Simulationsbasierte Bewertung fehlertoleranter Festkommarecheneinheiten

Modellierung in VHDL und kritische Betrachtung der Bewertungsmethodik

Der Technischen Fakultät der Universität Erlangen-Nürnberg

zur Erlangung des Grades

Doktor-Ingenieur

vorgelegt von

Oliver Tschäche

Erlangen, Dezember 2000

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	18. Dezember 2000
Tag der Promotion:	17. September 2001
Dekan:	Prof. Dr. Harald Meerkamm
Berichterstatter:	Prof. Dr. Mario Dal Cin
	Prof. Dr. Wolfram H. Glauert

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dal Cin stellvertretend für den IfI 3 und dessen Mitarbeitern für die Unterstützung bedanken, die ich in meiner Zeit an der Uni-Erlangen erfuhr.

Volkmar Sieh danke ich für die vielen Diskussionsstunden und den Teamgeist, der die Grenzen dieser Dissertation weit überschreitet.

Susann Allmaier, Eva Seeberger und Ilka Hilgenböker haben mich gelehrt, meine Gedanken nicht nur in Worte zu fassen, sondern auch in Taten umzusetzen. Herzlichen Dank.

Graham Horton gebührt mein besonderer Dank: Durch seine Fragen gewinne ich die Freiheit, aus vielen verschiedenen Perspektiven auf mein Leben zu sehen. Dadurch gewinne ich die Möglichkeit, mich selbst zu erkennen und bewußter zu leben.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Anforderungen

Das Vertrauen in automatische Dienstleistungen wächst. Mittlerweile verlassen sich viele Millionen Menschen auf Telefon, Verkehrsleitsysteme, Internet, Flugzeugelektronik und Steuerungen, die uns z.B. ein Computer, ein Mikrokontroller oder ein einfacher Taschenrechner abnimmt. Hersteller dieser Anlagen bieten Verträge an, die Ausfallraten von weniger als eine Stunde Downtime im Jahr versprechen, obwohl es bislang kaum Verfahren gibt, diese Raten exakt zu berechnen. Hier verläßt man sich Großenteils auf Erfahrungen und Qualität des Ingenieures, der aus Abschätzungen eine Zahl 'berechnet'. Wird bei der Entwicklung eines Systems von Seite der Spezifikation eine maximale Ausfallrate vorgeschrieben, die vom rein funktionalen System nicht erfüllt werden kann, muß der Ingenieur zu Fehlertoleranzmaßnahmen greifen, deren Integration die Güte des Systems erhöht.

Ein Beispiel, das in naher Zukunft realisierbar scheint, sind medizinische Anwendungen, bei denen der Chirurg nur noch im virtuellen Operationssaal steht und sein Patient viele Kilometer entfernt auf dem OP-Tisch liegt. Für diese Anwendung ist ein Signalverarbeitungssystem notwendig, das mit geringer Latenz die Bilddaten vom OP-Tisch zum Chirurgen transportiert und andererseits sicher die Aktionen des Chirurgen zum Roboterarm überträgt. Störungen bei der Bildübertragung und der Robotersteuerung können sich tödlich auswirken. Als weiteres Beispiel sei hier die Bordelektronik eines Flugzeugs aufgegriffen. Versagt hier die Elektronik, stehen ebenfalls Menschenleben auf dem Spiel, wenn ein Fehler zum Absturz führt. Auch Atomkraftwerke liegen in der Hand digitaler Steuerungen. Ein Versagen dieser sollte möglichst vollkommen ausgeschlossen werden können, da hier im Falle eines Gaus sogar ganze Landstriche verwüstet werden.

Zu den hohen Anforderungen an die Zuverlässigkeit eines Systems kommt z.B. für moderne Jets noch die Anforderung hinzu, den Rechenbedarf für die Steuerung der Maschine zu stillen, um das Flugzeug in einer stabilen Fluglage zu halten (der Pilot lenkt die Maschine 'nur' noch). Bei der Bildübertragung ist es üblich, das hohe Datenvolumen durch Kompression zu reduzieren. Soll diese Aufgabe On-The-Fly mit geringer Latenz durchgeführt werden, ist auch

hier Hardware hoher Leistung erforderlich.

Wie man anhand der Beispiele erkennen kann, werden an Systeme nicht nur hohe Anforderungen an die korrekte Durchführung der Aufgaben sondern zusätzlich auch hohe Anforderungen an die Geschwindigkeit, mit der diese Leistung erfüllt werden, gestellt. Diese Arbeit setzt an diesen Punkten an.

1.2 Einordnung dieser Arbeit

Ein Teil dieser Arbeit beschäftigt sich auf dem Gebiet des Entwurf digitaler Teil-Systeme, die oben genannten Anforderungen gerecht werden. Die betrachteten Teil-Systeme sind Festkommarechenwerke, die zum einen aufgrund von Pipeline-Methoden einen hohen Rechendurchsatz erzielen und zum anderen durch Fehlertoleranzmaßnahmen mit hoher Zuverlässigkeit korrekt durchgeführte Operationen leisten sollen. Die Systeme sind in VHDL modelliert, um mit einem üblichen Prozeß bis zu einem Silizium-Prototypen kommen zu können. Der Entwurfsweg wurde bis zur Abbildung auf eine Standardzellenbibliothek gegangen. Auf diesen Abbildungen wurden alle Simulationen durchgeführt. Genauergesagt wurden in C VHDL-Generatoren programmiert, die die VHDL-Modelle der Festkommarecheneinheiten für verschiedene Genauigkeiten generieren. Die Funktionalität wurde u.a. basierend auf [15] und [5] entwickelt. Die Fehlertoleranz ist vollständig in die Hardware der Systeme integriert, weshalb man sie im Sinne von [9] in die Klasse der Self-Checking-Checkers einordnen kann. Die Systeme organisieren die Korrektur eventuell fehlerhafter Ergebnisse selbst. Die implementierten Methoden wurden aus [9] und [10] abgeleitet.

Während für die Bewertung der Effizienz bezüglich Rechenleistung dieser Systeme bereits etablierte Methoden zur Verfügung stehen, wird für die Bewertung der Fehlertoleranz das nicht so erschlossene Gebiet der simulationsbasierten Fehlerinjektion, siehe u.a. [12], betreten. Ein weiterer Teil dieser Arbeit betritt dieses Neuland und weist auf Unsicherheiten bei der Interpretation der Daten offen hin, die mit den Methoden aus [13] und [14] gewonnen wurden. Es wird ein Weg gezeigt, den ermittelten Werten die Fakten zu entlocken und darauf korrekte Interpretationen aufzubauen.

Um die Leistungsfähigkeit und Fehlertoleranz der Entwicklung zu bewerten, werden mit bekannten Werkzeugen konkrete Zahlen für diese Parameter bestimmt. Da die untersuchten Modelle relativ komplex sind, mußte jedoch an diesen Verfahren ebenfalls Hand angelegt werden, um in absehbarer Zeit zu Ergebnissen kommen zu können. Hier liefert diese Arbeit auf dem Gebiet der Optimierung simulationsbasierter Ermittlung einige Ideen, um Datenvolumen und benötigte Zeit dieser Simulationen um Größenordnungen zu reduzieren.

1.3 Inhaltsübersicht

Kapitel 2 beschreibt die Algorithmen, auf der die Implementierungen basieren. Es wurden 8 Funktionen ausgewählt, die mittels Bit- und CORDIC-Algorithmen in einem Festkommare-

chenwerk implementiert wurden: Multiplikation, Division, Quadratwurzel, Logarithmus, Exponentialfunktion, Sinus, Cosinus und Arcustangens.

In Kapitel 3 werden drei Grundtypen dieser Festkommarecheneinheiten entwickelt. Einer arbeitet bitseriell, während die beiden anderen, der bitparallele und bitredundante Ansatz, ihre Daten parallel verarbeiten. Die parallelen Ansätze unterscheiden sich in der Kodierung der Operanden. Dieser Entwicklungsprozeß nimmt einen großen Anteil dieser Arbeit ein. Die Implementierung der Bit- und CORDIC-Algorithmen wurde für funktionale Recheneinheiten ohne Fehlertoleranzmaßnahmen durchgeführt, die mit 8, 16, 32, 64 oder 128 Bit Genauigkeit arbeiten. Die Daten für den Rechendurchsatz und Chipflächenbedarf sind für alle drei Ansätze gegenübergestellt und grafisch aufgearbeitet.

Kapitel 4 stellt basierend auf den drei Ansätzen Fehlertoleranzmaßnahmen vor, die eine Implementierung direkt in Hardware erlauben. Daraus entstanden zusätzlich zu den drei rein funktionalen Entwürfen jeweils drei mit dem entsprechenden FT-Verfahren: Zwei Rollback-Verfahren, Micro- and Macro-Rollback, und ein TMR-Verfahren. Daraus folgt, daß ein Pool von 12 verschiedenen Implementierungen zu bewerten ist. Ein Rollback-Verfahren schied bereits im Vorfeld aus, so daß letztlich 11 Varianten gegeneinander verglichen wurden.

Kapitel 5 betrachtet die Fehlertoleranzeigenschaften der 11 Modelle. Die Methodik, mit der die entsprechenden Zahlen ermittelt wurden, wird eingehend dargestellt. Dabei wird besonders auch auf die Aussagekraft und Konfidenz der ermittelten Werte eingegangen. An dieser Stelle werden auch die ’wunden’ Punkte der Analysemethodik (simulationsbasierte Fehlerinjektion) beleuchtet.

1.4 Beitrag dieser Arbeit

- Basierend auf [15] und [5] werden drei Varianten einer Festkommarecheneinheiten entwickelt. Diese wurden in VHDL implementiert. Die Bewertung erfolgte mit dem Tool ’Design-Analyzer’ von Synopsys.
- Gleichzeitig werden Fehlertoleranzmaßnahmen abgeleitet aus z.B. [9] und [10] auf deren Eignung zur Integration für diese drei Varianten untersucht. Es werden mehrere Fehlertoleranzmechanismen implementiert, die mit dem Tool ’VERIFY’, siehe [13] und [14], bewertet werden.
- Da die Bewertung durch VERIFY im üblichen Verfahren sämtliche, verfügbare Rechenresourcen gesprengt hätte, wurde das Verfahren an sich modfiziert, um die Simulation zu beschleunigen und die erzeugten Datenmengen zu reduzieren.
- Der Vergleich verschiedener Varianten ist ein weiterer Punkt, den diese Arbeit beleuchtet. Es wird kritisch betrachtet, wie sehr man den Zahlen für die Fehlertoleranzcharakteristika vertrauen darf, an welcher Stelle bei der Bewertung von Fehlertoleranzeigenschaften Fehler gemacht werden können und wo durch kritische Annahmen die Methodik zur Berwertung an sich Bedenken auslöst.

Kapitel 2

Implementierung der Bit/CORDIC Algorithmen

In diesem Kapitel wird beschrieben, welche Algorithmen zur Implementierung einer Festkommaarithmetik eingesetzt wurden, wie sie funktionieren und wie sie am effizientesten in Hardware realisiert werden können. Eine differenzierte Betrachtung der alternativen Implementierungsmöglichkeiten rundet dieses Kapitel ab und begründet gleichzeitig die weitere Gliederung dieser Arbeit.

Im Rahmen des DFG-Projekts „3D-Smart-Pixels-Rechner“ wurden die Weichen für den Inhalt dieses Projekts bereits in einem sehr frühen Stadium gestellt. Um nahtlos an die Vorarbeiten der Projektpartner anzuknüpfen, wurde entschieden, die Festkommaarithmetik basierend auf Bit- bzw. CORDIC-Algorithmen zu implementieren.

2.1 Funktionsweise

Dieses Kapitel beschreibt die Funktionsweise der Algorithmen, auf denen die Implementierung des Rechenwerks basiert. Die Algorithmen wurden an der Friedrich-Schiller-Universität Jena im Rahmen einer Diplomarbeit [7] von der streng mathematischen Form in eine Form transformiert, die gewissen Rahmenbedingungen genügt. Diese werden im Kapitel 2.2 erläutert.

Bei den Bit- bzw. CORDIC-Algorithmen handelt es sich um iterative Verfahren zur Berechnung von Multiplikation, Division, Wurzel-, Exponential- und Logarithmusfunktion mit den Bitalgorithmen und den trigonometrischen Funktionen Sinus, Cosinus und Arcustangens mit den CORDIC-Algorithmen. Generell können mit den Bit- bzw. CORDIC-Algorithmen noch viele andere Funktionen realisiert werden. Bei den acht oben genannten handelt es sich um eine Auswahl von Funktionen, die häufig gebraucht werden und die gleichzeitig auch die Basis für die Berechnung komplexerer Funktionen bilden. Die beiden Algorithmen bedienen sich in einer Iteration der gleichen Abfolge dreier einfacher Operationen:

- Shift-Operation: In den einzelnen Iterationsstufen werden die Operanden der vorange-

gangenen Stufe nicht nur so benötigt, wie sie diese liefert, sondern auch um eine gewisse Stellenanzahl n bei binärer Zahlendarstellung nach rechts verschoben. Dies entspricht einer Multiplikation des Operanden mit 2^{-n}.

- Addition/Subtraktion zweier Operanden: Verglichen zur Shift-Operation und der im folgenden Punkt dargestellten Berechnung der Selektionsbedingung ist die Addition/Subtraktion zweier Operanden die komplexeste Funktion.

- Berechnung der Selektionsbedingung, bei der überprüft wird, ob das Ergebnis einer bestimmten Addition/Subtraktion ≥ 0 ist.

Der Unterschied zwischen Bit- und CORDIC-Algorithmen liegt in der unterschiedlichen Verarbeitung der neu berechneten Parameter aufgrund der Selektionsbedingung:

- Für die Bitalgorithmen werden, wenn die Selektionsbedingung erfüllt ist, die neu berechneten Parameter an die nächste Iterationsstufe übergeben. Ist die Selektionsbedingung nicht erfüllt, werden die Parameter, die dieser Stufe als Operanden übergeben wurden, als Ergebnis geliefert.

- Bei den CORDIC-Algorithmen werden generell die neu berechneten Parameter an die nächste Iterationsstufe übergeben. Über die Selektionsbedingung wird hier entschieden, ob die Addition bzw. Subtraktion in der nächsten Stufe vertauscht werden sollen.

In Tabelle 2.1 sind die Anfangsbedingungen, die Iterationsvorschriften, die Selektionsbedingung und deren Auswirkung für die Zuweisung des Ergebnisses einer Iterationsstufe für die Bitalgorithmen bzw. die Auswahl der Operation für die CORDIC-Algorithmen dargestellt, wobei die Einträge bereits an die Rahmenbedingungen, die im Kapitel 2.2 erläutert sind, angepaßt sind.

Wie man in Tabelle 2.1 sieht, benötigen die gewählten Funktionen maximal drei Iterationsvariablen x_i, y_i und z_i. Je nach Funktion, die berechnet werden soll, werden die Variablen x_0, y_0 und z_0 vor der ersten Iteration mit den Operanden der geforderten Funktion, 0, 1 oder κ belegt, wobei gilt: $\kappa = \prod_{i=0}^{\infty} \sqrt{1+(2^{-i})^2}$. Die Selektionsbedingung ist von der Variablen y_{tmp} bzw. y_i abhängig und ist erfüllt, wenn der entsprechende Wert positiv ist. Das Ergebnis der gewünschten Funktion wird in der Variablen x_i abgelegt. Wie in [7] nachgelesen werden kann, konvergieren die Bitalgorithmen für Operanden im Bereich [0.5 , 1.5] und die CORDIC-Algorithmen im Bereich $[0\ ,\ \frac{\pi}{4}\ ]$ zufriedenstellend. Stößt man den Iterationsprozeß für diese Operanden an, gewinnt man pro Iterationsstufe etwa ein Bit Genauigkeit.

Die Iterationsvorschriften zur Berechnung eines neuen Wertes für eine Variable bestehen aus einer Addition bzw. Subtraktion des alten Wertes dieser Variablen und einer Konstanten oder einem um i geshifteten Wert einer der drei Eingangsvariablen. Mit „Shiften“ ist in diesem Zusammenhang gemeint, daß eine Variable mit 2^{-i} multipliziert wird. Betrachtet man die Variable im binären Zahlensystem, entspricht dies einer Verschiebung des Kommas nach links. Diese Shift-Operation wird in der Tabelle 2.1 durch den Operator $\gg$ symbolisiert.

Tabelle 2.1: Algorithmen

Funktion	*Initialisierung*	*Iterationsvorschrift*	*Selektionsbedingung*	*wahr*	*falsch*
$\ln(1+\alpha)$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = 1$	$x_{tmp} = x_i \quad +\ln(1+2^{-i})$ $y_{tmp} = y_i \quad -(z_i \gg i)$ $z_{tmp} = z_i \quad +(z_i \gg i)$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$ $z_{i+1} = z_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$ $z_{i+1} = z_i$
e^{α}	$x_0 = 1$ $y_0 = \alpha$	$x_{tmp} = x_i \quad +(x_i \gg i)$ $y_{tmp} = y_i \quad -\ln(1+2^{-i})$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\sqrt{\alpha}$	$x_0 = 0$ $y_0 = \alpha$	$x_{tmp} = x_i \quad +(0.5 \gg i)$ $y_{tmp} = y_i \quad -((x_i \gg i) +$ $+ \ (0.25 \gg 2i))$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\alpha \cdot \beta$	$x_0 = 0$ $y_0 = \alpha$	$x_{tmp} = x_i \quad +(\beta \gg i)$ $y_{tmp} = y_i \quad -(1 \gg i)$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\frac{\alpha}{\beta}$	$x_0 = 0$ $y_0 = 1$	$x_{tmp} = x_i \quad +(\alpha \gg i)$ $y_{tmp} = y_i \quad -(\beta \gg i)$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\sin\alpha$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = \kappa$	$x_{i+1} = x_i \quad \pm(z_i \gg i)$ $y_{i+1} = y_i \quad \mp\arctan(2^{-i})$ $z_{i+1} = z_i \quad \mp(x_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation
$\cos\alpha$	$x_0 = \kappa$ $y_0 = \alpha$ $z_0 = 0$	$x_{i+1} = x_i \quad \mp(z_i \gg i)$ $y_{i+1} = y_i \quad \mp\arctan(2^{-i})$ $z_{i+1} = z_i \quad \pm(x_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation
$\arctan\alpha$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = 1$	$x_{i+1} = x_i \quad \pm\arctan(2^{-i})$ $y_{i+1} = y_i \quad \mp(z_i \gg i)$ $z_{i+1} = z_i \quad \pm(y_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation

Die einzige Ausnahme in diesem System ist die Wurzelfunktion, die nur mit Additionen realisierbar scheint. Im Kapitel 2.2 wird aber gezeigt, daß auch diese Funktion im Prinzip in das Schema Shift-Operation gefolgt von Addition bzw. Subtraktion und abschließender Selektion paßt.

2.2 Rahmenbedingungen

Dieses Kapitel beschreibt die Rahmenbedingungen, denen die Algorithmen und deren Operanden genügen müssen, damit eine effiziente Implementierung in einem integrierten Schaltkreis möglich ist. In dieser Arbeit wird ausschließlich die Berechnung einer Mantisse mit feststehendem Komma behandelt, was der komplexe Teil einer Realisierung solcher Fließkommaoperationen [1] ist.

[1] Im Rahmen des Projektes ist geplant, die Berechnung der Charakteristik parallel zur Berechnung der Mantisse durchzuführen, um die volle Funktionalität einer Fließkommaeinheit bieten zu können, worauf aber in dieser Arbeit nicht eingegangen wird.

Tabelle 2.2: Kodierung der Operanden

Binäre Kodierung	*Dezimaler Wert*
011.11111	3.96875
001.00000	1.0
000.10000	0.5
000.00001	0.03125
000.00000	0.0
111.11111	-0.03125
111.00000	-1.0
100.00000	-4.0

2.2.1 Kodierung und Wertebereiche der Operanden

Die Operanden werden in einer Festkommaarithmetik im binären Zweierkomplement kodiert, wobei das Most-Significant-Bit (MSB) das Vorzeichenbit ist, die nächsten zwei Bits als Stellen links vom Komma zu interpretieren sind und die restlichen Bits Nachkommastellen sind. Wird im Folgenden von z.B. 16 Bit Genauigkeit gesprochen, bedeutet das, daß davon 1 Bit Vorzeichen, 2 Bits Vorkommastellen und die restlichen 13 Bits Nachkommastellen sind.

Aufgrund dieser Kodierung liegt ein Operand immer im Wertebereich zwischen -4.0 einschließlich und $+4.0$ ausschließlich. Je nach Genauigkeit n ist die Differenz zweier benachbarter Zahlen 2^{3-n}. In Tabelle 2.2 sind Beispiele für die Kodierung mit 8 Bit Genauigkeit nach diesem System dargestellt.

Die in dieser Arbeit verwendeten und in Kapitel 2.1 dargestellten Algorithmen benötigen ausschließlich Additions-, Subtraktions- und Shiftoperationen. Um die Addition und Subtraktion effizient in Hardware zu realisieren, ist die Wahl, die Operanden im Zweierkomplement zu kodieren, ideal, da sowohl positive als auch negative Operanden ohne weiteren Aufwand mit bekannten Addierern (Ripple-Carry-Adder oder Conditional-Sum-Adder) verarbeitet werden können und eine Subtraktion hauptsächlich durch Invertieren des zweiten Operanden realisiert werden kann. Bei einer Shiftoperation werden die Bits eines Operanden nach rechts geschoben, wobei das Least-Significant-Bit (LSB) herausfällt und das neue MSB entsprechend dem Vorzeichen des zu shiftenden Operanden ergänzt wird. Wird ein Operand um mehrere Bits nach rechts geschoben, wird dieser Vorgang entsprechend wiederholt.

2.2.2 Iterationstiefe

Die Iterationstiefe hängt von der geforderten Genauigkeit ab. Betrachtet man die Algorithmen von Tabelle 2.1 mit dem Ziel im Auge, die Iterationstiefe für eine Genauigkeit n zu ermitteln, fällt auf, daß der rechte Operand jeder Summe bzw. Differenz immer von der aktuellen Iterationsstufe i abhängt. Diese Operanden lassen sich in zwei Klassen einteilen:

- Konstanten: Damit sind die Summanden gemeint, die ausschließlich von i abhängen und

keinen Bezug zu den Iterationsvariablen x_i, y_i und z_i bzw. zu den Parametern α oder β haben. Betrachtet man die Grenzwerte $\lim_{i\to\infty} \ln(1+2^{-i})$ und $\lim_{i\to\infty} \arctan 2^{-i}$ ist das Ergebnis jeweils 2^{-i}, wobei die Funktionswerte der beiden Funktionen immer kleiner sind als dieser Grenzwert. D.h. für die Iterationsstufe i ist beim Summanden frühestens das i.Bit nach dem Komma 1, alle Bits links von diesem sind 0. Dies gilt auch für die Konstante $0.5 \gg i$ jeder Iterationsstufe.

- Geshiftete Operanden: Damit sind die Summanden gemeint, die aus der Shiftoperation der Iterationsvariablen x_i, y_i und z_i bzw. der Parameter α oder β hervorgehen. Da Iterationsvariablen und Parameter aufgrund des Wertebereichs maximal in der zweiten Stelle links vom Komma ein relevantes Bit enthalten, enthalten die geshifteten Operanden der i.Iterationsstufe frühestens an der Stelle $i-1$ rechts vom Komma ein relevantes Bit. Die Bits links davon enthalten nur das Vorzeichenbit.

Um die maximale Genauigkeit der Algorithmen auszuschöpfen, müssen so lange weitere Iterationsstufen durchlaufen werden, bis ein relevantes Bit nicht mehr mit der gewählten Kodierung und Genauigkeit dargestellt werden kann. Die schärfere Bedingung wird durch die geschifteten Operanden formuliert, da bei diesen bereits an der Stelle $i-1$ rechts vom Komma relevante Bits zu erwarten sind, während die Konstanten frühestens an der Stelle i relevante Bits liefern.

Da für eine Genauigkeit von n Stellen $n-3$ Bits nach dem Komma verfügbar sind, wird in der Iterationsstufe $(n-3)+1$ je nach Eingangswert eventuell noch eine Änderung bewirkt. Um die maximale Genauigkeit der Algorithmen auszuschöpfen, muß die Iterationstiefe $n-2$ gewählt werden.

2.3 Implementierung

In diesem Abschnitt werden zwei Methoden beschrieben, wie die Algorithmen in Funktionseinheiten umgesetzt und welche Schnittstellen an den Funktionseinheiten benötigt werden. Es werden zwei Implementierungsvarianten untersucht. Die eine Variante besteht aus einem einzigen Funktionsblock, dessen Ausgänge rückgekoppelt sind und auf diese Weise mehrere Iterationen auf den Operanden ermöglicht. In der anderen Variante wird für jede einzelne Iterationsstufe ein Funktionsblock generiert. In den folgenden Unterkapiteln werden diese Varianten veranschaulicht.

Beide Varianten bieten in ihrer Implementierung jeweils alle acht genannten Funktionen. Die Auswahl der zu berechnenden Funktion geschieht über ein Signal, das den Index der gewünschten Funktion enthält. Die Zuordnung von Index zu Funktion ergibt sich aus der Reihenfolge, in welcher die Funktionen in der Tabelle 2.1 aufgeführt sind, wobei die oberste Funktion, $\ln(1+\alpha)$, den Index 0 erhält und aufsteigend bis zur untersten Funktion, $\arctan\alpha$ mit Index 7, indiziert wird.

Für die Implementierungen der Algorithmen ist in Abbildung 2.1 ein Blockschaltbild dargestellt. Es zeigt die Art, auf welche die Schnittstellen zu den Dateneingaben realisiert werden.

In dem Block, der mit dem Schriftzug „Initialisierung“ versehen ist, werden die Eingangsvariablen x_0, y_0 und z_0 entsprechend der Initialisierungsspalte der Tabelle 2.1 belegt. Die Belegung hängt vom Wert der Operanden α und β und dem Index für die Auswahl der Funktion ab. Dieser Vorgang ist für beide Implementierungsmethoden identisch.

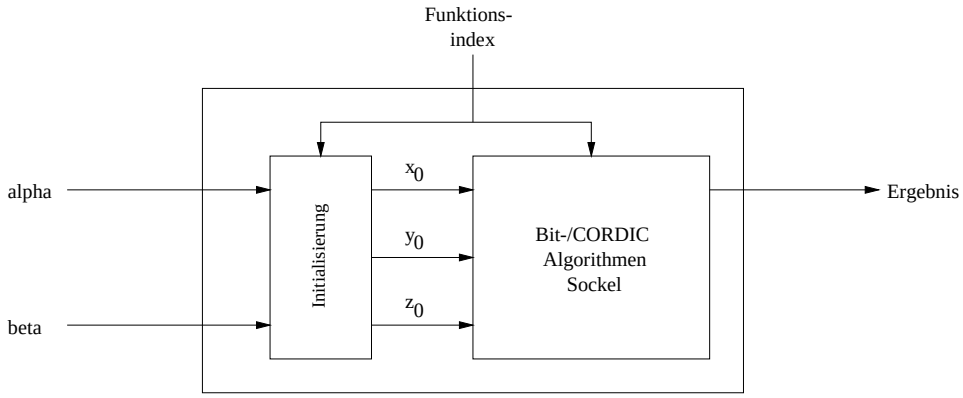

Abbildung 2.1: Wirt für Implementierungsmöglichkeiten

Der Block mit der Beschriftung „Bit-/CORDIC-Algorithmen Sockel“ ist die Schnittstelle zu den beiden Implementierungsvarianten. An dieser Stelle kann, wie in einen Sockel, eine der in den folgenden Kapiteln beschriebenen Implementierungsvarianten eingesetzt werden.

2.3.1 Implementierung in einem einzelnen Knoten

In diesem Unterkapitel wird ein Einsatz für den „Bit-/CORDIC-Algorithmen Sockel“, siehe Abbildung 2.1, gezeigt, der die genannten Algorithmen innerhalb eines Knotens integriert. In diesem Knoten wird eine einzelne Iteration durchgeführt. Damit der vollständige Algorithmus abgearbeitet wird, werden die Ergebnisse zeitlich solange wieder zum Eingang rückgekoppelt, bis die nötige Iterationstiefe erreicht ist. Ein Blockschaltbild dieser Implementierungsvariante ist in 2.2 dargestellt.

Da bei dieser Einzelknoten-Lösung der Knoten die Berechnung für alle Iterationsstufen durchzuführen hat und in verschiedenen Iterationsstufen verschiedene Anforderungen an die Funktionalität des Knotens gestellt werden, muß der Knoten noch davon in Kenntnis gesetzt werden, für welche Stufe er die Berechnungen durchführen muß. Dies wird durch die Steuereinheit realisiert. Mit dieser Information kann er dann die der Iterationsstufe entsprechenden geshifteten Operanden und Konstanten generieren.

Außerdem müssen für die erste auszuführende Iteration die von außen anliegenden Daten an den Knoten gelegt werden und nicht die rückgekoppelten, vorher berechneten Parameter. Hierfür sind die mit MUX gekennzeichneten Multiplexer verantwortlich. Die Steuereinheit generiert hierfür ein entsprechendes Signal.

Wie schon aus den vorangegangenen Ausführungen abgeleitet werden kann, handelt es sich bei der Steuereinheit im Prinzip um einen Iterationszähler. Er steuert je nach Iterationsstufe die Multiplexer und den Knoten.

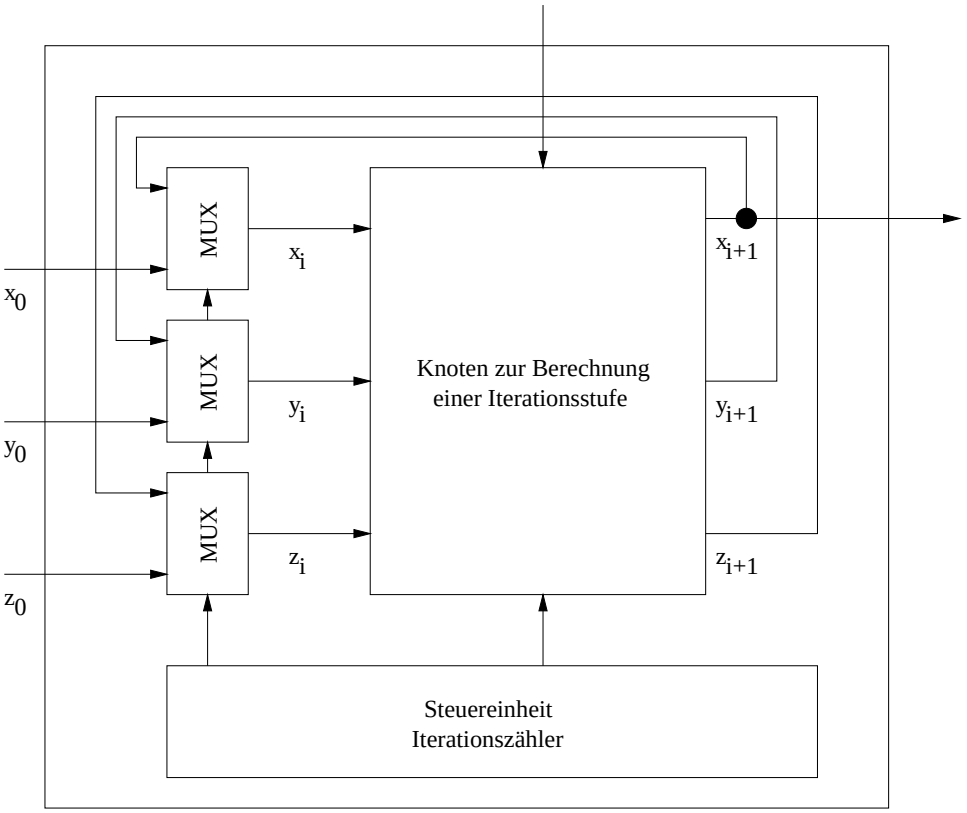

Abbildung 2.2: Einzelknoten

Der Inhalt des Knotens wird im Kapitel 2.3.3 behandelt.

2.3.2 Implementierung in seriell verschalteten Knoten

Dieses Kapitel beschreibt einen Einsatz für den „Bit-/CORDIC-Algorithmen Sockel“, siehe Abbildung 2.1, der die Algorithmen durch seriell verschaltete Knoten realisiert. Je nach geforderter Genauigkeit n werden, wie in Kapitel 2.2.2 beschrieben, $n-2$ Knoten hintereinander geschaltet, wobei jeder Knoten für exakt eine Iterationsstufe zuständig ist.

Im Gegensatz zur Implementierungsvariante in einem Knoten ist ein zeitliches Auffächern der Parameterberechnung mit diesem Ansatz nicht nötig, weshalb, wie in Abbildung 2.3 zu sehen ist, für den seriellen Ansatz keine Steuereinheit benötigt wird.

Die Größe der Knoten in Abbildung 2.2 und 2.3 symbolisiert die unterschiedliche Mächtigkeit, die die beiden Knoten erfüllen müssen. Während für die Lösung in einem einzelnen Knoten für jede Iterationsstufe Konstanten und die entsprechend der Stufe verschiedene Schiebeoperation der Operanden im Knoten integriert werden müssen, ist für den seriellen Ansatz für jeden Knoten jeweils nur eine bestimmte Konstante und eine bestimmte Schiebeoperation, eben die für die Iterationsstufe, nötig.

Eine differenzierte Betrachtung der beiden Ansätze findet in Kapitel 2.3.4 statt, nachdem im

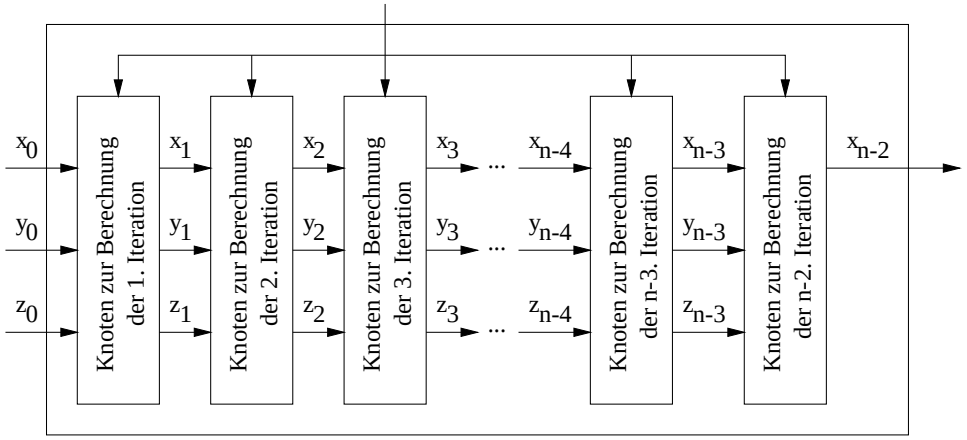

Abbildung 2.3: Seriell verschaltete Knoten

folgenden Kapitel Aufschluß über die Funktionalität der Knoten gegeben wird.

2.3.3 Struktur der Knoten

Beide Implementierungsvarianten ähneln sich, wenn man die Funktionalität der Knoten zur Berechnung einer Iterationsstufe betrachtet. In Abbildung 2.4 werden die Funktionseinheiten und der Datenfluß in diesen dargestellt.

Alle Blöcke, die in Abbildung 2.4 dargestellt sind, werden von der Kontrolleinheit durch Steuersignale, die in der Abbildung der Übersichtlichkeit halber nicht dargestellt sind, in einen bestimmten Modus geschaltet. Die Kontrolleinheit erhält über den Pfad „fkt“ die Information, welche der acht Funktionen berechnet werden soll, und über den Pfad „stage“, für welche Iterationsstufe dieser Knoten die entsprechende Operation durchführen soll. Der Pfad „fkt“ wird in Abbildung 2.1 durch den von oben kommenden Pfeil symbolisert, der durch den Sockel über die Implementierungsvariante an den Knoten weitergereicht wird. Der Pfad „stage“ wird in der Implementierungsvariante mit einem Knoten vom Steuerwerk benutzt, um dem Knoten die aktuelle Iterationsstufe mitzuteilen. In der Variante seriell angeordneter Knoten ist dieser Pfad fest auf einen Wert gelegt, der jener Stufe entspricht, in der ein Knoten fixiert ist.

Die Funktionsweise der Blöcke, die mit „Shifter“ bezeichnet sind, hängt von der Iterationsstufe ab. Betrachtet man Tabelle 2.1 erkennt man, daß die Parameter x_i, y_i und z_i jeweils um die gleiche Anzahl Bits, nämlich i, in einer Iterationsstufe verschoben werden müssen.

Vergleicht man die Iterationsvorschriften in Tabelle 2.1 erkennt man, daß für jede beliebige der acht Funktionen sich der neue Wert aus der Summe oder Differenz des alten Wertes und einer Variablen ergibt, wobei die Variable von der zu erfüllenden Funktion abhängt. Betrachtet man beispielsweise die Iterationsvorschrift für den Parameter x_i der Funktion $\ln(1+\alpha)$ und $\sin\alpha$, wird im Falle der Logarithmusberechnung die Konstante $\ln(1+2^{-i})$ und im Falle der Sinusberechnung der um i geshiftete Parameter z_i addiert bzw. subtrahiert. Dies wird mit einem Multiplexer realisiert, der aus den acht möglichen Summanden den auswählt, der die geforderte Funktion erfüllt. In Abbildung 2.4 wird diese Auswahl durch den Block „Funktionsaus-

wahl“ symbolisiert. Die eingehenden acht Datenpfade entsprechen den Summanden für eine bestimmte Funktion. Die entsprechenden Verbindungen zu den Quellen wurden zur Förderung der Übersichtlichkeit weggelassen.

Der mit „Addierer/Subtrahierer“ bezeichnete Block führt je nach zu erfüllender Funktion eine Addition oder eine Subtraktion aus. Dies ist im Falle der Bitalgorithmen für die Erfüllung der Funktion immer die gleiche Operation, siehe Tabelle 2.1. Im Falle der CORDIC-Algorithmen hängt es davon ab, ob die Selektionsbedingung der vorangegangenen Stufe erfüllt war.

Der Block „Selektion“ wird nur dann aktiv, wenn eine Funktion mit einem Bitalgorithmus realisiert wird. In diesem Fall wird in Abhängigkeit der Selektionsbedingung, die im Block „Selektionsbedingung“ generiert wird, entweder der neu berechnete Wert als Ergebnis herausgeführt oder, wenn die Selektionsbedingung nicht erfüllt ist, der alte Wert. Für CORDIC-Algorithmen werden immer die neu berechneten Werte weitergegeben.

Für die Berechnung der acht Funktionen werden, wie in Tabelle 2.1 zu sehen ist, häufiger Konstanten benötigt. Diese werden im mit „Konstanten“ bezeichneten Block generiert. Zu Gunsten der Übersichtlichkeit verlassen repräsentativ für alle Konstanten nur zwei Datenpfade diesen Block.

2.3.4 Vergleich der Implementierungsvarianten

In diesem Unterkapitel wird differenziert betrachtet, wie sich die beiden Implementierungsvarianten der Knoten bei der Einknotenlösung von dem der seriell verschalteten Knoten unterscheiden.

Für die weiteren Überlegungen in diesem Kapitel wird vorrausgesetzt, daß ein integrierter Schaltkreis eine Reihe von Funktionseinheiten beherbergt, die parallel jeweils eine der acht Funktionen auf Operanden ausführen. Im Falle der Einzelknotenlösung heißt das, daß mehrere dieser Knoten mit den entsprechenden Steuereinheiten auf einem Schaltkreis integriert werden. Im Falle der seriell verschalteten Knoten bedeutet es die Integration mehrerer Reihen von verketteten Knoten.

Je mehr parallel rechnende Einheiten auf einem Schaltkreis fester Größe, in Bezug auf die Chipfläche, integriert werden können, desto mehr Operationen können parallel ausgeführt werden. Läßt sich der Highlebel Algorithmus ideal parallelisieren, erzielt man einen linearen Speedup. Dies hat zur Folge, daß das System umso schneller arbeitet, je mehr parallel rechnende Einheiten auf einem Chip integriert werden können. Benötigt die Integration eines Knotens weniger Fläche, können mehr Knoten auf der gleichen untergebracht werden. Deshalb gilt es die Implementierungsvariante zu wählen, die am wenigsten Fläche benötigt.

Ein weiterer relevanter Faktor zur Bestimmung der günstigeren Implementierungsvariante ist die Zeit, die der entsprechende Ansatz benötigt, um eine komplette Berechnung durchzuführen. Da beide Varianten quasi identische Knoten, die demnach auch identische Durchlaufzeiten haben, verwenden und sich nur darin unterscheiden, an welchem Ort diese berechnet werden, ist die Zeit, die für die vollständige Berechnung einer der acht Funktionen benötigt wird, gleich und kann daher in den folgenden Überlegungen vernachlässigt werden. Mit dem Ort der Berech-

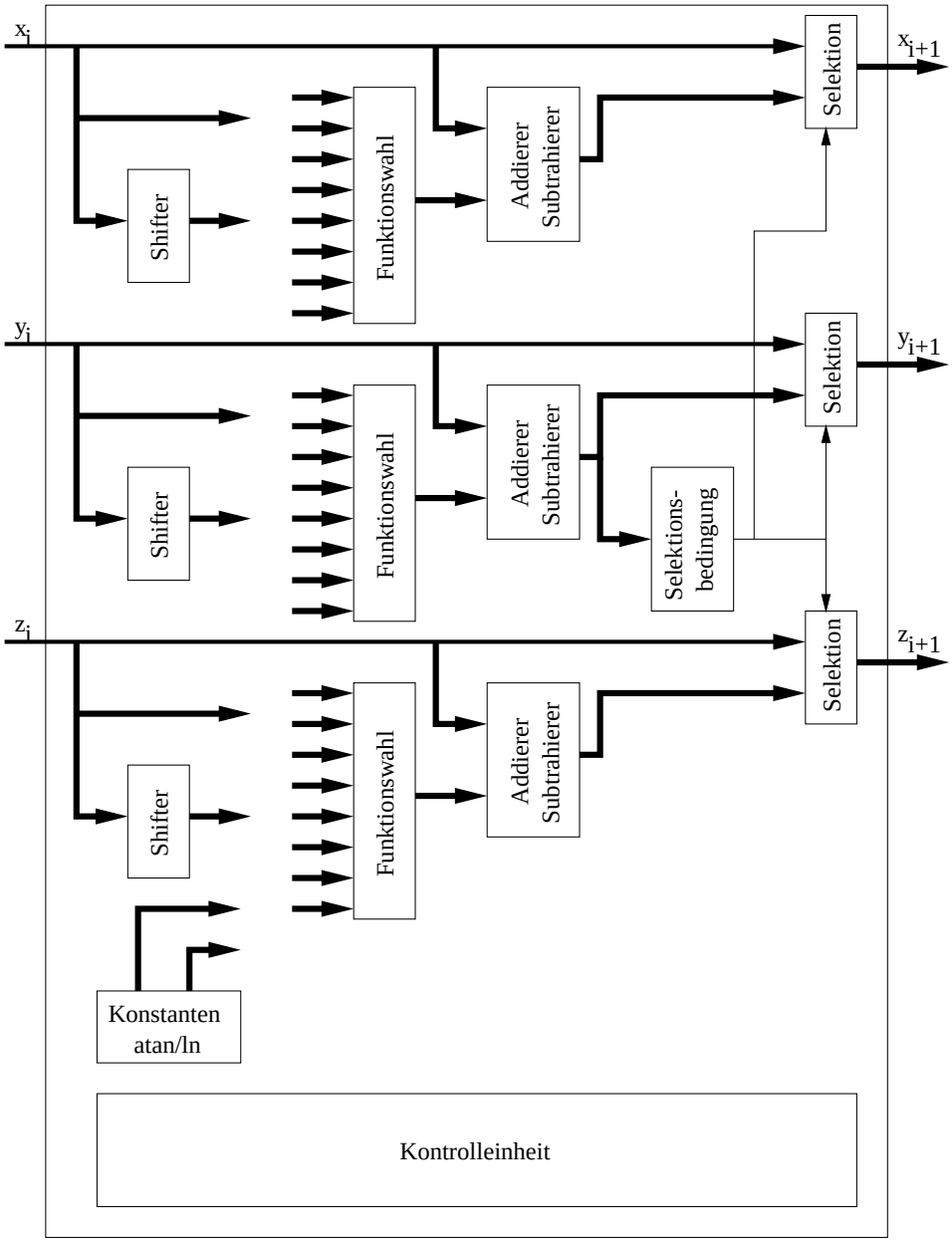

Abbildung 2.4: Struktur eines Knotens

nung ist gemeint, daß im Falle der Einzelknotenlösung die $n-2$ Iterationen auf einem Knoten berechnet werden, während im seriell verschaltetem Fall die $n-2$ Iterationen nacheinander in $n-2$ Knoten berechnet werden.

Realisiert man die Kette seriell verschalteter Knoten als Pipeline, so daß im zeitlichen Abstand einer Durchlaufzeit eines Knotens immer wieder neue Berechnungen angestoßen werden, entspricht der serielle Ansatz mit $n-2$ Knoten einem Feld von $n-2$ Einzelknoten, die parallel arbeiten. Somit sind die Rechenkapazitäten, berechnete Funktionen pro Zeit, beider Ansätze identisch.

Die Effizienz der Ansätze hängt deshalb nur von der Komplexität eines Knotens ab. Vergleicht man die Blockschaltbilder der Einzelknotenlösung, siehe Abbildung 2.2, mit der Lösung seriell verschalteter Knoten, siehe Abbildung 2.3, fallen in der Einzelknotenlösung die Multiplexer und die Steuereinheit auf, die bei der seriell verschalteten Variante nicht benötigt werden. Zusätzlich sind bei der Einzelknotenlösung die Blöcke „Shifter“ und „Konstanten“ wesentlich größer: Der Block „Shifter“ muß im Einzelknotenansatz sämtliche Schiebeoperationen für jede Iterationsstufe beherrschen, während er für die seriell verschaltete Lösung nur eine einzelne Schiebeoperation bieten muß. Der Block „Konstanten“ muß ebenfalls im Einzelknotenfall mehr Konstanten beherbergen, nämlich die Konstanten für alle Iterationsstufen, während im seriell verschalteten Fall jeweils nur ein Satz Konstanten generiert werden muß.

Eine mögliche Optimierung für die Einzelknotenlösung wäre, mehrere parallel arbeitende Knoten zu synchronisieren und alle Knoten mit einem einzigen „Konstanten“-Block zu versorgen. Damit wäre die Einzelknotenlösung der seriell verschalteten Lösung auch in Bezug auf die „Konstanten“-Blöcke konkurrenzfähig. Die größeren „Shift“-Blöcke der Einzelknotenlösung bleiben aber weiterhin der Nachteil dieses Ansatzes.

Daraus wird geschlossen, daß der Ansatz mit seriell verschalteten Knoten aufgrund seiner Rechenleistung in Bezug zur benötigten Chipfläche der günstigere Ansatz ist.

2.4 Optimierung

Basierend auf den Algorithmen, die in Tabelle 2.1 dargestellt sind und die in dieser Form in [1] nachgeschlagen werden können, und den in Kapitel 2.3.4 ermittelten Vorzügen der seriell verschalteten Implementierungsvariante wurden vereinfachte Spezifikationen eines entsprechenden Automaten in VHDL modelliert. Durch das vertiefte Verständnis, das aus diesem Entwurf gewonnen wurde, gelang es, die Folgerungen von Kapitel 2.3.4 zu bestätigen und vor allem die Algorithmen von Tabelle 2.1 weiter zu optimieren, so daß mit weniger Gattern ausgekommen werden kann.

Wie in Tabelle 2.1 zu sehen ist, werden für die Multiplikation und Division die beiden Operanden α und β in jeder Iterationsstufe gebraucht. Dies bedeutet für die Verarbeitungspipeline, daß zusätzlich zu den Parametern x_i, y_i und z_i zwei weitere Variablen in jeder Stufe gespeichert werden müssen. Die Speicherung geschieht im Schaltkreis durch Flipflops. Da die sonstigen logischen Operationen, die in einem Knoten benötigt werden, im Verhältnis zu den Flipflops als gering einzustufen sind, entspräche die Einsparung der Speicher für die Operanden α und β eine Reduzierung der Flipflops um 40%. In Tabelle 2.3 sind die entsprechenden Optimierungen bei Multiplikation und Division eingetragen.

Für die Multiplikation ist die Optimierung trivial, da hier im Gegensatz zu den meisten anderen Funktionen nur zwei, x_i und y_i, der drei Iterationsparameter x_i, y_i und z_i in Gebrauch sind. Hier kann einfach z_i zum Durchschleifen des Operanden β verwendet werden. Folgerichtig ergibt sich dann ein z_{i+1} ausschließlich aus z_i. Dadurch kann in der Iterationsvorschrift für x_{i+1} β durch z_i ersetzt werden. Damit ist der Operand β in den Iterationsvorschriften der Multiplikation eliminiert.

Tabelle 2.3: Optimierte Algorithmen

Funktion	*Initialisierung*	*Iterationsvorschrift*	*Selektionsbedingung*	*wahr*	*falsch*
$\ln(1+\alpha)$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = 1$	$x_{tmp} = x_i \quad +\ln(1+2^{-i})$ $y_{tmp} = y_i \quad -(z_i \gg i)$ $z_{tmp} = z_i \quad +(z_i \gg i)$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$ $z_{i+1} = z_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$ $z_{i+1} = z_i$
e^{α}	$x_0 = 1$ $y_0 = \alpha$	$x_{tmp} = x_i \quad +(x_i \gg i)$ $y_{tmp} = y_i \quad -\ln(1+2^{-i})$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\sqrt{\alpha}$	$x_0 = 0$ $y_0 = \alpha$	$x_{tmp} = x_i \quad +(0.5 \gg i)$ $y_{tmp} = y_i \quad -((x_i \gg i) +$ $+ \ (0.25 \gg 2i))$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$
$\alpha \cdot \beta$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = \beta$	$x_{tmp} = x_i \quad +(z_i \gg i)$ $y_{tmp} = y_i \quad -(1 \gg i)$ $z_{tmp} = z_i \quad +0$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$ $z_{i+1} = z_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$ $z_{i+1} = z_i$
$\frac{\alpha}{\beta}$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = \beta$	$x_{tmp} = x_i \quad +(1 \gg i)$ $y_{tmp} = (y_i - z_i) \ll 1$ $z_{tmp} = z_i \quad +0$	$y_{tmp} \geq 0$	$x_{i+1} = x_{tmp}$ $y_{i+1} = y_{tmp}$ $z_{i+1} = z_{tmp}$	$x_{i+1} = x_i$ $y_{i+1} = y_i$ $z_{i+1} = z_i$
$\sin\alpha$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = \kappa$	$x_{i+1} = x_i \quad \pm(z_i \gg i)$ $y_{i+1} = y_i \quad \mp\arctan(2^{-i})$ $z_{i+1} = z_i \quad \mp(x_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation
$\cos\alpha$	$x_0 = \kappa$ $y_0 = \alpha$ $z_0 = 0$	$x_{i+1} = x_i \quad \mp(z_i \gg i)$ $y_{i+1} = y_i \quad \mp\arctan(2^{-i})$ $z_{i+1} = z_i \quad \pm(x_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation
$\arctan\alpha$	$x_0 = 0$ $y_0 = \alpha$ $z_0 = 1$	$x_{i+1} = x_i \quad \pm\arctan(2^{-i})$ $y_{i+1} = y_i \quad \mp(z_i \gg i)$ $z_{i+1} = z_i \quad \pm(y_i \gg i)$	$y_i \geq 0$	obere Operation	untere Operation

Für die Division ist eine Modifikation des Bitalgorithmus' nötig, da hier beide Operanden α und β in die Iterationsvorschriften eingehen und nur ein freier Iterationsparameter z_i zur Verfügung steht. Der neue Algorithmus hat mit den Bitalgorithmen nur noch die Selektion aus neu berechneten und alten Werten gemeinsam, was ihn aber nicht mehr in das Klassifizierungsschema passen läßt. Dies wird in Tabelle 2.3 mit der zusätzlichen Zeilenabgrenzung symbolisiert. Der Algorithmus entspricht dem, den man anwenden würde, wenn man per Hand binär dividieren würde. Ist der Rest der Subtraktion von Divisor vom Dividenden positiv, wird der neu berechnete Wert für den ganzzahligen Anteil x_i, bei dem an der entsprechenden Stelle eine 1 vermerkt wird, und den Rest y_i an die nächste Iterationsstufe übergeben. Ist die Differenz negativ, wird der alte Wert für den ganzzahligen Anteil, in dem in diesem Fall eine 0 vermerkt wird, und der alte Rest weitergegeben. In beiden Fällen wird der Rest vor der Übergabe an die nächste Stufe um ein Bit nach links geschoben. Das Blockschaltbild in Abbildung 2.4 wird also am Ausgang y_{i+1} mit einem Funktionsblock ergänzt, der im Falle einer Division den vom Selektionsblock ausgehenden Wert um eine Stelle nach links verschiebt, wie in Abbildung 2.5 zu sehen ist. Im

Parameter z_i wird wie bei der Multiplikation ein Operand, für die Division der Divisor, von Iterationsstufe zu Iterationsstufe durchgeschleift.

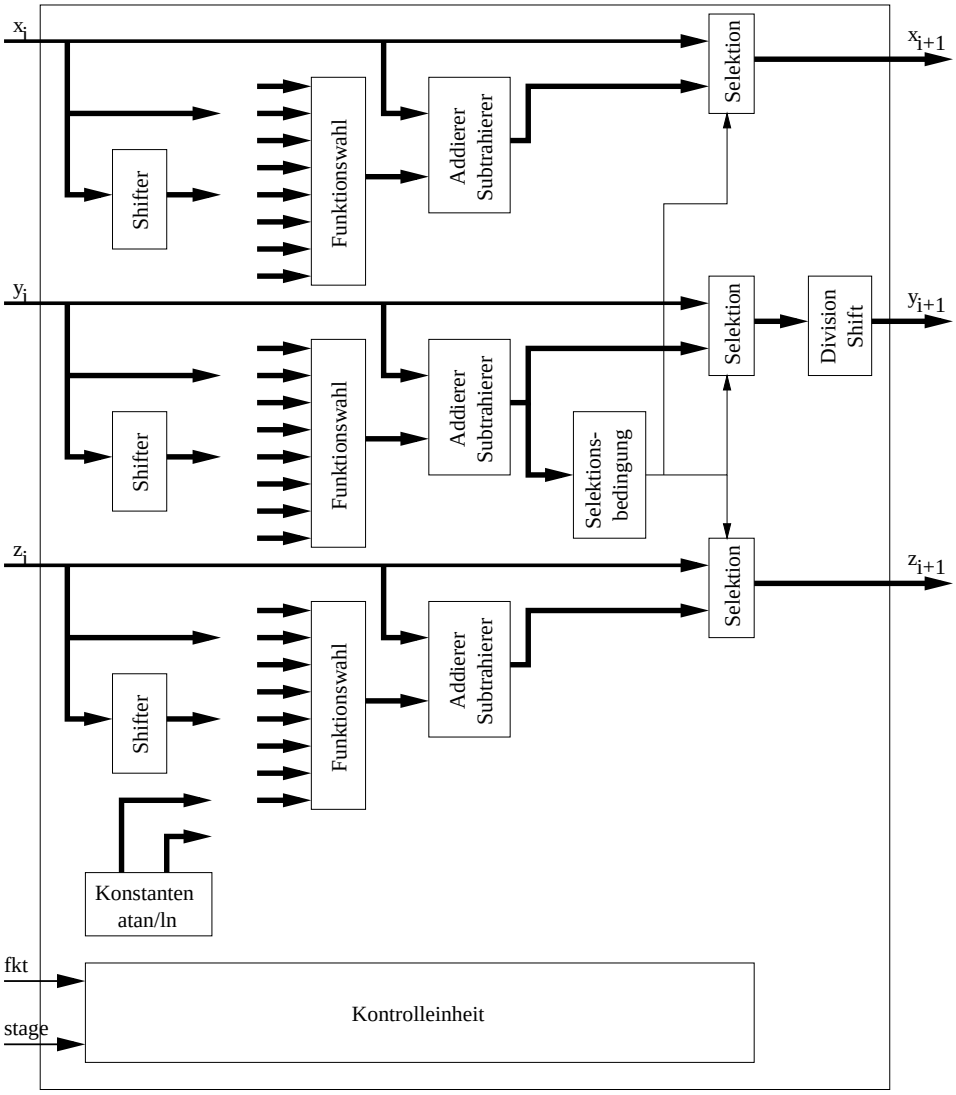

Abbildung 2.5: Struktur eines optimierten Knotens

Auf diese Weise ist es gelungen, die Operanden α und β vollständig aus den Iterationsvorschriften zu eliminieren, wodurch 40% der Flipflops zum Speichern der Parameter innerhalb einer Iterationsstufe eingespart werden.

2.5 Validierung

In diesem Abschnitt geht es nicht um die formale Verifikation der korrekten Funktion, z.B. der Konvergenz, der Bit- und CORDIC-Algorithmen, oder korrekten Implementierung, sondern es geht darum, für die Implementierung Testmuster zu finden, bei deren erfolgreicher Abarbeitung eine korrekte Implementierung angenommen wird.

Das Ziel dieser Arbeit war, mehrere Modelle eines integrierten Schaltkreises zu entwerfen und den günstigsten auszuwählen. Der Entwurf ging bis auf Gatterebene, so daß aufgrund des Modells mit Tools zur Erstellung von Maskenlayout automatisch ein integrierter Schaltkreis erzeugt werden kann. Um die Funktionalität des Gatterlayouts zu überprüfen (und später auch die korrekte Arbeitsweise des Chips), werden Testmuster benötigt, die eine möglichst hohe Fehleranzahl abdecken. Für das Modell heißt das, daß dessen Antworten auf die Stimuli mit den erwarteten Antworten verglichen werden. Stimmen die beiden Antworten für alle Testmuster überein, wird der Chip für funktionsfähig erklärt. Sowohl in der Simulation als auch beim Testen des fertigen Chips wird die Anordnung in Abbildung 2.6 benutzt. Damit dienen die Testmuster während der Simulation zum Test des Modells und in der Produktionsphase zum Test des Chips.

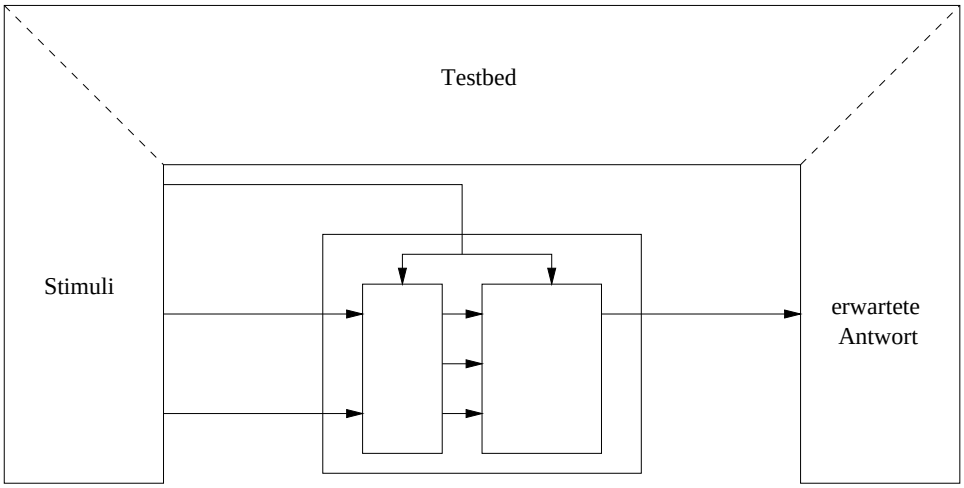

Abbildung 2.6: Anordnung für den Test

Bei den Testmustern handelt es sich um pseudo-zufällig erzeugte Bitfolgen, den Stimuli. Um auch schon in diesem Stadium eine möglichst hohe Fehlerabdeckung zu bekommen, wurde darauf geachtet, daß jede der acht Funktionen gleich häufig getestet wird, um den Test der verschiedenen Funktionen gleichmäßig abzudecken. Zusätzlich wurden die zufälligen Werte für die Operanden gegebenenfalls durch andere ersetzt, die den Rahmenbedingungen von Kapitel 2.2 genügen. In Abbildung 2.6 werden die Stimuli im linken Ast des Testbeds generiert.

Um die erwarteten, korrekten Antworten zu berechnen, wurden die Algorithmen in der Programmiersprache C implementiert, siehe Anhang A. Das C-Programm berechnet nicht nur das Endergebnis einer Funktion, sondern auch alle Zwischenergebnisse x_i, y_i und z_i aller Iterationsstufen. Diese Zwischenergebnisse ermöglichen eine effiziente Fehlersuche in den VHDL-Modellen, da man nicht erst am Ende die erwartete Antwort validieren kann, sondern nach jeder einzelnen Iteration am Ausgang jedes Knotens. Abbildung 2.6 zeigt auch, wie das Endergebnis wieder dem Teil des Testbeds zugeführt wird, das die Antwort des Systems mit der erwarteten Antwort vergleicht.

Die Programmiersprache C wurde gewählt, weil sie dem Autor dieser Arbeit am vertrautesten ist und somit eine schnelle Entwicklung der geforderten Testmuster erwartet wurde. VHDL wurde für die Testmustererzeugung nicht gewählt, weil, wie in Kapitel 3 zu erfahren ist, die

VHDL-Modelle der Implementierungen nicht direkt in VHDL spezifiziert wurden sondern von einem Modellgenerator, der ebenfalls in C spezifiziert ist. In diesen Modellgenerator ist die Erzeugung der Testbench, die auf oben genannten Testmustern basiert, integriert.

Im Zentrum der Abbildung 2.6 steht der modellierte Schaltkreis, der die Funktion berechnet. Dieser Block entspricht dem, der in Abbildung 2.1 zu sehen ist.

Diese Testmuster werden in Kapitel 4.3 in Bezug auf ihre Tauglichkeit zum Funktionstest des Chips untersucht. In Kapitel 5.1.1 werden diese Stimuli herangezogen, um die Fehlertoleranz-charakteristika zu berechnen.

Kapitel 3

Realisierung der Knoten

Nachdem im vorangegangenen Kapitel 2 die Funktionsweise und Anpassung der Bit- und CORDIC-Algorithmen an die digitale Realisierung dargestellt wurden, beschäftigt sich dieses Kapitel mit der Integration einer einzelnen Iterationsstufe auf Gatterebene. Wie schon in Kapitel 2 eingeführt, wird dieses Modul Knoten genannt. Das funktionale Blockschaltbild eines Knotens ist bereits in Abbildung 2.5 dargestellt worden. Verbindet man mehrere Knoten in einer Kette, wie in der Implementierungsvariante von Kapitel 2.3.2 beschrieben wird, wobei die Ausgaben des Knotens i die Eingaben des Knotens $i+1$ sind, ist das gleichbedeutend mit einer Aneinanderreihung von Iterationsstufen, die die Funktionalität der Algorithmen realisieren, siehe auch Abbildung 2.3.

Es wurden drei verschiedene Typen, bitparallel, bitredundant und bitseriell, dieser Knoten in VHDL nahe Gatterebene modelliert. Dieses Kapitel liefert eine detaillierte Beschreibung dieser Typen. Überdies werden die drei Knotentypen gegeneinander verglichen, welcher Typ unter welchen Bedingungen die kürzeste Latenzzeit, die effizienteste Nutzung der Chipfläche, den höchsten Durchsatz an Rechenoperationen und die günstigste Zahl der Ein-/Ausgabeleitungen bietet.

Um maximale Flexibilität im Ergebnis des Entwurfs zu gewährleisten, wurde kein VHDL-Modell der Knoten entwickelt, das eine feste Anzahl Bits für die Genauigkeit vorgibt, sondern es wurde ein VHDL-Modell Generator für jeden Knotentyp in der Programmiersprache C entwickelt, bei dem die Genauigkeit als Parameter eingeht. Die Generatoren können VHDL-Modelle für 8 bis 1024 Bits Genauigkeit erzeugen. Die Funktionalität der Modelle wurde mit der Testbench, die in Kapitel 2.5 beschrieben ist, für die Genauigkeit von 8 und 16 Bits validiert, indem die entsprechenden Modelle in einer Simulation in der Testbench auf korrekte Ausgaben für die Testmuster überprüft wurden.

Die untere Genauigkeit von 8 Bits wurde gewählt, weil sie die niedrigste Genauigkeit ist, um sinnvolle Berechnungen mit den Knoten durchzuführen. Bei der späteren Analyse mit dem Fehlerinjektor VERIFY, siehe auch [12] in einem späteren Kapitel, sind die anfallenden Datenmengen proportional zur Anzahl der Gatter. Um mit dem Fehlerinjektor aussagekräftige Werte zu ermitteln, müssen viele ($\gg 5000$), je mehr desto besser, siehe Kapitel 5.1.4, Fehlerinjektionsexperimente durchgeführt werden. Das Produkt aus Anzahl der Gatter und Experimente ist ein

Maß für die anfallenden Datenmengen. Um diese mit aktuellen Rechensystemen handhaben zu können, wurde die untere Grenze für die Genauigkeit möglichst tief, auf 8 Bit, gelegt.

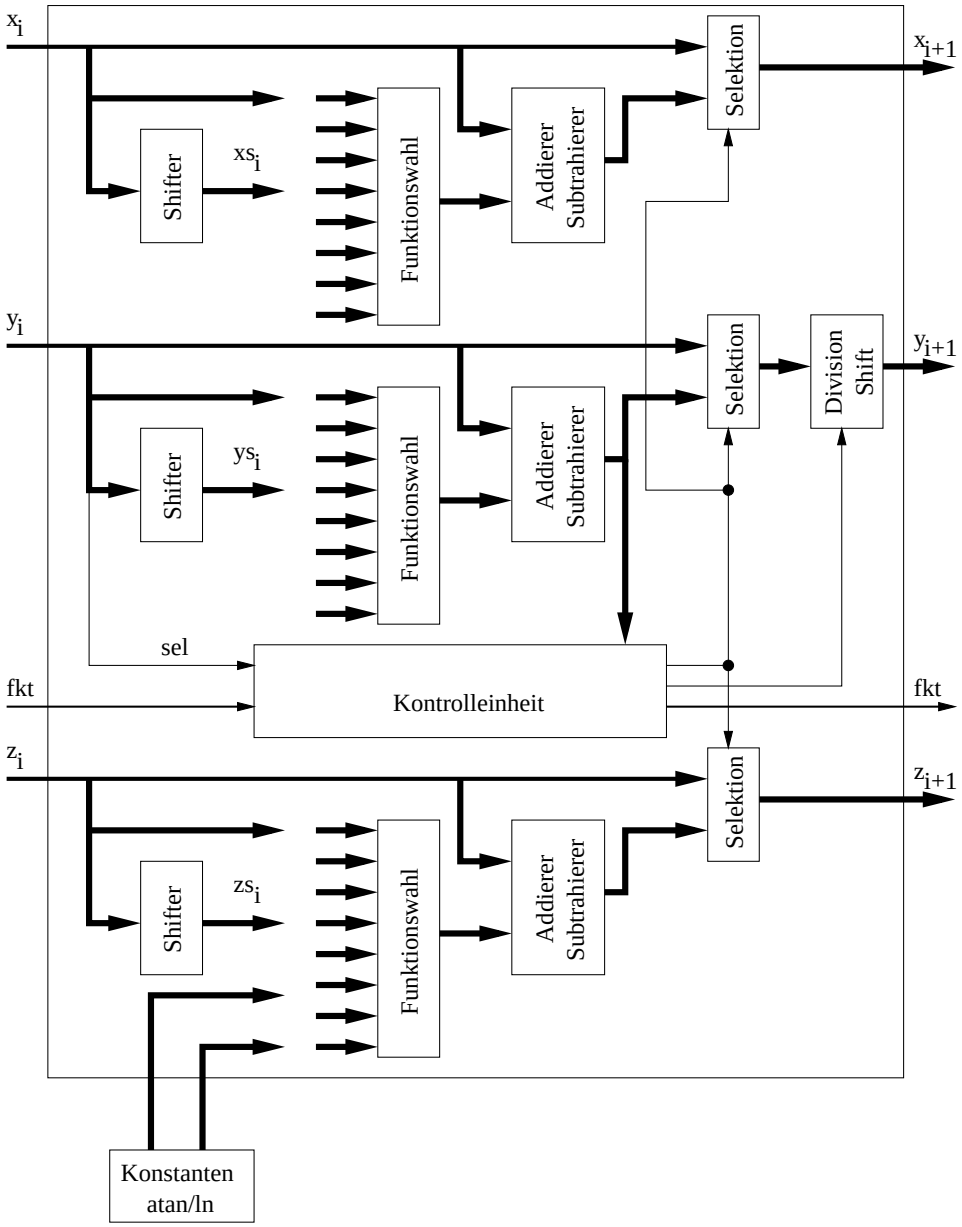

Abbildung 3.1: Blockschaltbild der Implementierung

Alle drei Implementierungen der Knoten sind als Pipeline, siehe [5] Seite 379 ff., organisiert. Das bedeutet, daß sobald eine Berechnung einer der acht Funktionen den ersten Knoten verläßt, um dem zweiten übergeben zu werden, ist der erste Knoten wieder frei, um die Berechnung der nächsten Funktion durchzuführen. Im Idealfall sind alle Knoten mit der Berechnung auch verschiedener Operationen beschäftigt.

Im Vergleich zu Abbildung 2.5 erscheint in Abbildung 3.1 noch das Signal *sel*, das aus dem Parameter y_i generiert wird. Für den Fall einer anstehenden Funktion, die einen CORDIC-

Algorithmus benötigt, dient dieses Signal zur Auswahl der Operation für diesen Schritt.

Die Information, für welche Stufe ein Knoten arbeitet, ist für die Implementierungsvarianten fest verdrahtet, da ein Knoten immer für die gleiche Iterationsstufe zuständig ist. Deshalb ist in Abbildung 3.1 im Vergleich zu Abbildung 2.5 der Pfad „stage“ nicht mehr zu finden.

Ein weiterer Unterschied der Blockschaltbilder in Abbildung 2.4 und 3.1 liegt in der Art, wie die Konstanten generiert werden. Der Block zur Generierung der Konstanten ist in Abbildung 2.4 noch im Knoten selbst angesiedelt, während in Abbildung 3.1 dieser Block ausgelagert ist. Da geplant ist, mehrere parallel rechnende Pipelines auf einem Chip zu integrieren, wäre eine ähnliche Optimierung möglich, wie sie schon für die Einzelknotenlösung in Kapitel 2.3.4 vorgeschlagen wurde. Synchronisiert man diese Pipelines, kann man die Konstantengeneratoren mehrerer Knoten der gleichen Iterationsstufe durch einen globalen Konstantengenerator für alle Knoten dieser Stufe ersetzen. Durch die Modellierung auf diese Art sind beide Design-Alternativen einfach zu realisieren, was wiederum der Flexibilität des Entwurfs zuträglich ist.

In den folgenden Kapiteln werden für jeden Knotentyp in einer Tabelle seine charakteristischen Daten zusammengefaßt. In der ersten Spalte dieser Tabelle ist die Genauigkeit n vermerkt, für welche die folgenden Werte ermittelt wurden. Die nächsten drei Spalten enthalten Daten, die mit einem Synthesetool durch die Abbildung des VHDL-Modells auf eine Standardzellenbibliothek entstanden sind: Die Fläche A, die maximale Durchlaufzeit T und der längste Pfad P. Die Werte für die Fläche und Durchlaufzeit sind abhängig vom verwendeten Prozeß und deshalb ohne Einheit dargestellt. Sie dienen nur zum Vergleich der einzelnen Knotentypen. Der maximale Pfad enthält die Anzahl der Gatter für den längsten Pfad. In der fünften Spalte IO wird schließlich die Anzahl der Ein-/Ausgabeleitungen präsentiert.

Die letzten vier Spalten enthalten Werte, die aus den Daten der vorangegangenen Spalten berechnet wurden. Der Rechendurchsatz R zeigt die Anzahl maximal möglicher Operationen (vollständig gefüllte Pipeline) in 1000 Zeiteinheiten auf einer Chipfläche von 1e6 Einheiten. Die Zahlen 1000 und 1e6 wurden gewählt, damit die resultierenden Daten im Bereich zwischen 1 und 10000 liegen und somit ein einfaches Vergleichen innerhalb der Tabellen möglich ist. Die Größe R wurde gewählt, um schnelle Implementierungen mit hohem Flächenbedarf mit langsameren aber kleineren Implementierungen vergleichen zu können. Ist z.B. eine Implementierung halb so groß und braucht die doppelte Zeit zur Berechnung einer Operation, sind beide Implementierungen unter der Vorraussetzung idealer Parallelisierung gleichwertig, da man von der halb so großen Variante zwei Einheiten auf der gleichen Fläche unterbringen kann.

Die Latenz L beschreibt die Zeit, die von der Eingabe der Operanden bis zur Ausgabe des Ergebnisses vergeht. Die Durchlaufzeiten T und P beschreiben hingegen nur, wie lange ein einzelner Knoten mit einer Operation beschäftigt ist. Diese Zahl gewinnt an Bedeutung, wenn die Pipeline nicht ideal gefüllt werden können. Dies kann bei Algorithmen passieren, die nicht ideal parallelisiert werden können und deshalb teilweise auf Ergebnisse gewartet werden muß, bevor man die nächste Operation anstoßen kann. Die Indizes T und P zeigen, welche Größe für die Durchlaufzeit hergenommen wurde. T steht für die vom Synthesetool aufgrund der Gatterfanin und -fanouts berechneten Zeit, P steht für eine Zeit, die sich ergeben würde, wenn man die Durchlaufzeit jedes Gatters auf eine Zeiteinheit setzen würde.

Die Größen R_P und L_P wurden zugefügt, um den Gewinn abzuschätzen, der sich ergeben würde,

wenn man keinen Standardzellenentwurf als Integrationsprozeß zugrunde legt, sondern ein ideales Full-Custom-Design. Ein großer Nachteil des Standardzellenentwurfs ist, daß die Treiber der Ausgänge, die viele Eingänge treiben, zu hohen Durchlaufzeiten führen. Beim Full-Custom-Design hat man die Möglichkeit, diese stark belasteten Ausgänge mit stärkeren Treiberstufen auszurüsten, um diesem Effekt entgegen zu wirken. Im Idealfall hat dann jedes Gatter eine konstante Durchlaufzeit. Die Größen R_P und L_P enthalten die Werte für diese konstante Gatterlaufzeit von 1 Zeiteinheit. Sie entsprechen somit der Anzahl des längsten Gatterpfades, über den ein Signal läuft.

Das Synthesetool, mit dem oben genannte Werte bestimmt wurden, ist der „design analyzer“ von Synopsys. Die dem Syntheseprozeß zugrunde liegende Gatterbibliothek ist die von Synopsys mitgelieferte Standardzellenbibliothek *„lsi_10k“*. Es wurde für jeden Knotentyp jeweils eine Synthese für die Genauigkeiten $8, 16, 32, 64$ und 128 Bit durchgeführt und eine Analyse gestartet, um oben genannte Werte mit dem Tool zu berechnen.

Die Latenz hat eine Bedeutung, wenn in einer Applikation Datenabhängigkeiten bestehen, die ein Warten auf einen zu berechnenden Wert erfordern. Unter diesen Umständen können die Pipelines mehrerer parallel arbeitender Knoten nicht optimal gefüllt werden. Stattdessen gehen Berechnungszyklen während der Wartezeit verloren. Geht man davon aus, daß eine Applikation die Pipelines optimal nutzen kann, ist für die Performance des Systems allein die Anzahl der Rechenoperationen pro Zeit und Chipfläche ausschlaggebend. Im Allgemeinen, auch für nicht optimal parallelisierte Applikation, muß für die Bewertung der Effizienz die Latenz mit einbezogen werden. Eine Implementierung ist demnach besser, je geringer die Latenz und je größer die Anzahl der Rechenoperationen pro Zeit und Chipfläche ist.

3.1 Bitparallele Implementierung

In diesem Unterkapitel wird der Aufbau eines Knotens beschrieben, der die Iterationsvariablen x_i, y_i und z_i an seinem Eingang parallel annimmt und sie intern ebenfalls parallel verarbeitet. Parallel bedeutet, daß alle Bits der Variablen gleichzeitig am Eingang des Knotens anliegen. Die Parameter sind dabei wie in Kapitel 2.2.1 beschrieben kodiert.

Um die Pipelinefunktion der Knoten im Verbund zu ermöglichen, werden in jedem Knoten Flipflops eingesetzt, die die Operanden bis zum nächsten Taktimpuls stabil halten. Für diese bitparallele Implementierungsvariante sind die Flipflops für die Funktionsparameter im Block „Selektion“ von Abbildung 3.1 enthalten. Die Anzahl beträgt 3 mal der Anzahl der Bits eines Operanden für die Parameter x_{i+1}, y_{i+1} und z_{i+1}. Die Kontrolleinheit liefert die Flipflops für die geforderte Funktion, der die ausgegebenen Parameter unterzogen werden sollen. Für die acht Möglichkeiten bei der Funktionsauswahl werden weitere 3 Flipflops benötigt. Da das Selektsignal einem Bit im Parameter y_{i+1} entspricht, braucht es kein eigenes Flipflop. Darüberhinaus gibt es keine weiteren speichernden Elemente in einem bitparallelen Knoten.

In Abbildung 3.2 ist die Kontrolleinheit in Kombination mit einer der drei Berechnungseinheiten, hier die für y_i, die für x_i und z_i wurden weggelassen, aus Abbildung 3.1 detailliert dargestellt. Wie man in Abbildung 3.2 erkennt, wurde die Kontrolleinheit auf die drei Blöcke „Add-

/Sub-Kontrolle“, „Selektions-Kontrolle“ und „Div.-Shift-Kontrolle“ aufgeteilt. Alle einfach gerahmten Blöcke enthalten reine Schaltnetze. Nur der doppelt gerahmte Block „Flipflops“ enthält die oben beschriebenen, speichernden Elemente. Die Umsetzung für die Berechnungseinheiten der Parameter x_i und z_i ist analog zu der dargestellten y_i.

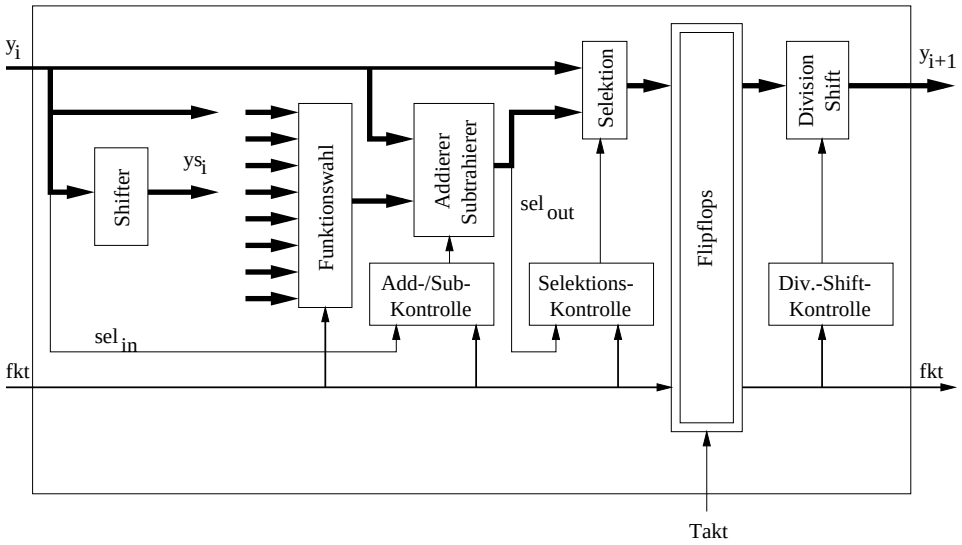

Abbildung 3.2: Schaltbild der bitparallelen Implementierung

Die Anzahl der Ein-/Ausgabeleitungen beträgt jeweils $3 \cdot n + 3$ Ein- und Ausgänge für die Datensignale und einen Takteingang. Dabei resultiert die $3 \cdot n$ aus der Anzahl der Eingangsparameter mit der Genauigkeit n plus den 3 Leitungen, die für die Funktionsauswahl fkt benötigt werden. Da die Konstanten ausschließlich von der Iterationsstufe, für die ein Knoten eingesetzt wird, abhängen, und für jede Iterationsstufe ein Knoten existiert, werden die $2 \cdot n$ Leitungen nicht zu den Eingängen gezählt, da sie im Knoten hart verdrahtet werden könnten.

In Tabelle 3.1 sind die Eingangs des Kapitels 3 beschriebenen Parameter eingetragen. Bei der bitparallelen und, wie später in Abschnitt 3.2 beschrieben, bitredundanten Implementierung wird in einem Takt eine Iteration des Algorithmus' durchgeführt. Da für eine komplette Berechnung $n-2$ Iterationen nötig sind, ergibt sich für die Latenz: $L_x = x \cdot (n-2)$.

Da das Design so ausgelegt ist, daß die Knoten in Form einer Pipeline bestehend aus $n-2$ Knoten ($A_{Pipeline} = (n-2) \cdot A_{Knoten}$) arbeiten, kann zu jedem Takt eine Berechnung angestoßen werden. Der Rechendurchsatz R, die Anzahl der Operationen, die in 1.000 Zeiteinheiten auf 1.000.000 Flächeneinheitein ausgeführt werden können, ergibt sich deshalb aus: $R_x = \frac{1e+9}{x \cdot (n-2) \cdot A}$.

Durch Verwendung des „Conditional-Sum-Adders“, siehe [5] Seite 78 ff., wäre eine theoretische Durchlaufzeit zu erwarten gewesen, die nicht linear, sondern logarithmisch mit einem Offset zur Genauigkeit n ansteigt. Dieser theoretisch logarithmische Anstieg ist zu erkennen, wenn man die Anzahl der Gatter für den längsten Pfad betrachtet: Bei Verdoppelung der Genauigkeit wird der Pfad um ungefähr ein Gatter länger. Da aber in den Stufen des Addierers von einem Ausgangstreiber bei steigender Genauigkeit immer mehr Eingänge getrieben werden müssen und bei einem Standardzellenentwurf die Ausgangstreiberleistung konstant ist, geht der Gewinn durch das Design verloren, weil für solche Signale mehr Zeit gebraucht wird, um

Tabelle 3.1: Charakteristik für bitparallele Implementierung

n	A	T	P	IO	R_T	R_P	L_T	L_P
008	1731	24.20	15	28/27	3978.65	6418.89	145.20	90.00
016	3924	40.75	16	52/51	446.69	1137.68	570.50	224.00
032	8872	73.58	17	100/99	51.06	221.00	2207.40	510.00
064	19752	151.05	19	196/195	5.40	42.97	9365.10	1178.00
128	43636	290.94	21	388/387	.62	8.66	36658.44	2646.00

einen Signalwechsel durchzuführen. Um diesen gravierenden Nachteil auszugleichen, ist ein Full-Custom-Design nötig, damit die Ausgangstreiber der kritischen Gatter in ihrer Leistung entsprechend verstärkt werden können. Eine Näherung an diese Bedingungen ist in den mit P indizierten Größen abgeschätzt.

Der qualitative Unterschied der Abhängigkeit der Durchlaufzeit von der Genauigkeit n ist in Abbildung 3.3 dargestellt. Die obere Kurve beschreibt die Durchlaufzeit wie sie von Synopsys für das Standardzellenlayout berechnet wurde und wie sie auch schon in Tabelle 3.1 in der Spalte T aufgeführt ist. Die untere Kurve entspricht der Durchlaufzeit, wenn man für jedes Gatter nur eine Zeiteinheit für die Durchlaufzeit veranschlagen würde. Dies entspricht der optimalen Lösung eines Full-Custom-Designs. In dieser Abbildung soll nicht aufgezeigt werden, daß ein gewisser Offset zwischen den beiden Kurven liegt. Die Aufmerksamkeit soll vielmehr auf den qualitativen Unterschied in der Krümmung der Kurven gelenkt werden. In der doppelt logarithmischen Darstellung in Abbildung 3.3 ist zu erkennen, daß die Kurven deutlich verschiedene Steigungen haben. Analysiert man die Kurven in einer linearen Darstellung, steigt die obere Kurve des Standardzellenlayouts etwa linear, während die Kurve für das optimierte Full-Custom-Design nur logarithmisch steigt, was theoretisch aufgrund des „Conditional-Sum-Adders“ auch zu erwarten war.

Abgesehen vom Addierer schlägt bei dieser Implementierungsvariante noch zu Buche, daß bei der Funktionsauswahl der Eingangsparameter nicht nur einzelne Signale selektiert werden müssen, sondern immer ein ganzer Vektor. Um dies zu realisieren, werden eine Anzahl von Multiplexern benötigt, die der Genauigkeit n entspricht. Dies hat zur Folge, daß das Steuersignal für die Selektion eines Vektors n Multiplexer treiben muß. An dieser Stelle wirkt sich wieder der Standardzellenentwurf nachteilig aus, da die Stärke des Ausgangstreibers begrenzt ist und nicht an die höheren Anforderungen angepaßt wird.

3.2 Bitredundante Implementierung

Dieses Kapitel beschreibt die bitredundante Implementierungsvariante für einen Knoten, die analog zur bitparallelen die Bits der Operanden parallel verarbeitet. Der Unterschied zur bitparallelen Methode liegt in der Kodierung der Parameter. Bei der bitparallelen Variante werden für eine Genauigkeit von n Bits auch n Bits für einen Parameter benötigt. Bei dieser Variante wird jeweils ein Bit des Operanden durch zwei dargestellt, weshalb $2 \cdot n$ Bits für einen Operan-

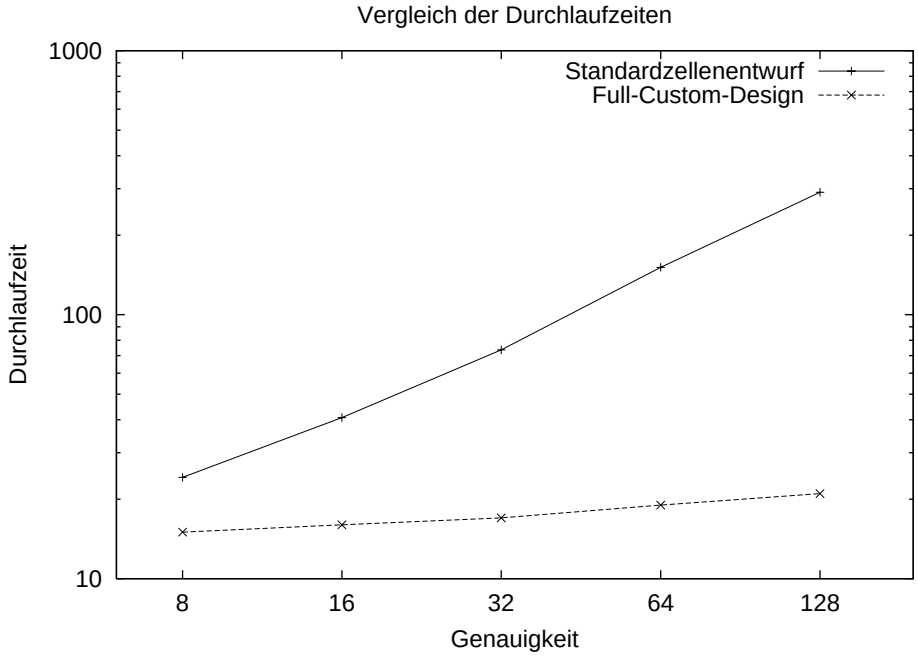

Abbildung 3.3: Vergleich der Durchlaufzeiten für die bitparallele Variante

den benötigt werden. Da sich die beiden Implementierungsvarianten nur in der Kodierung ihrer Parameter unterscheiden, gilt für beide das in Abbildung 3.2 dargestellte Blockdiagramm.

Die Kodierung wurde gewählt, weil sich dadurch der im „Addierer/Subtrahierer“- Funktionsblock enthaltene Addierer erheblich vereinfacht und durch die Kodierung Auswirkungen auf die Fehlertoleranzeigenschaften des Systems erwartet werden. Im bitparallelen Ansatz wurde ein „Conditional sum adder“ eingesetzt, der in $\log_2 n$ Schritten die Summe zweier Operanden bildet. Für die bitredundante Kodierung wird an dieser Stelle ein „Borrow save adder“ basierend auf Signed-Digit-Kodierung, siehe [5] Seite 107 ff., eingesetzt, der die Addition in einem Schritt unabhängig von der Genauigkeit n durchführt.

Der Nachteil der bitredundanten Variante ist, daß die Selektionsbedingung nicht mehr durch Testen des höchstwertigen Bits des Operanden y_i ermittelt werden kann. Aufgrund der Kodierung gehen im ungünstigsten Fall alle Bits in die Entscheidung ein. Deshalb ist die Ermittlung der Selektionsbedingung an einen relativ hohen Gatteraufwand geknüpft. Da die Selektionsbedingung an zwei verschiedenen Stellen gebraucht wird, einmal zur Selektion der Addition bzw. Subtraktion im Falle der CORDIC-Algorithmen, siehe sel_{in} in Abbildung 3.2, und einmal für die Selektion neuer bzw. alter Operanden für die Bitalgorithmen, siehe sel_{out} in Abbildung 3.2, müßte ein Block zur Bestimmung der Selektionsbedingung zweimal implementiert werden: Einmal zur Berechnung von sel_{in} und einmal zur Berechnung von sel_{out}. Durch ein weiteres Flipflop und jeweils ein Ein- und Ausgabesignal kann eine dieser Einheiten eingespart werden. Das entsprechende Blockschaltbild ist in Abbildung 3.4 dargestellt.

Der Rechendurchsatz der bitredundanten Implementierung wird bei steigender Genauigkeit n nur durch den Block, der die Selektionsbedingung berechnet, erhöht. Da im Gegensatz zur bitparallelen Variante der Addierer eine konstante Durchlaufzeit hat, trägt dieser nicht zur

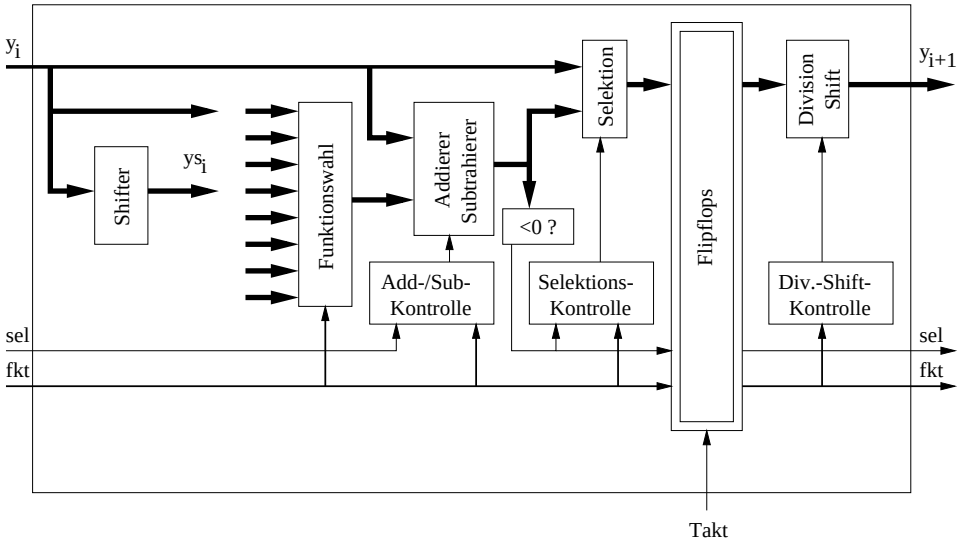

Abbildung 3.4: Schaltbild der bitredundanten Implementierung

Tabelle 3.2: Charakteristik für bitredundante Implementierung

n	A	T	P	IO	R_T	R_P	L_T	L_P
008	2102	38.30	24	52/51	2070.22	3303.73	229.80	144.00
016	4195	63.74	29	100/99	267.13	587.14	892.36	406.00
032	8379	111.31	35	196/195	35.73	113.66	3339.30	1050.00
064	16667	210.27	64	388/387	4.60	15.12	13036.74	3968.00
128	33427	384.09	77	772/771	.61	3.08	48395.34	9702.00

Erhöhung der Durchlaufzeit bei steigender Genauigkeit n bei.

Die Anzahl der Ein-/Ausgabeleitungen beträgt jeweils $3 \cdot 2 \cdot n + 4$ Ein- und Ausgänge für die Datensignale und einen Takteingang. Dabei resultiert die $3 \cdot 2 \cdot n$ aus der Anzahl der Eingangsparameter mit der Genauigkeit n, die durch $2 \cdot n$ Bits kodiert dargestellt werden, plus den 3 Leitungen, die für die Funktionsauswahl fkt benötigt werden, und der Leitung für das Selektsignal. Da die Konstanten ausschließlich von der Iterationsstufe, für die ein Knoten eingesetzt wird, abhängen, und für jede Iterationsstufe ein Knoten existiert, werden die $2 \cdot 2 \cdot n$ Leitungen nicht zu den Eingängen gezählt, da sie im Knoten hart verdrahtet werden können.

In Tabelle 3.2 sind die Eingangs des Kapitels 3 beschriebenen Parameter eingetragen. Bei der bitparallelen und bitredundanten Implementierung wird in einem Takt eine Iteration des Algorithmus' durchgeführt. Da für eine komplette Berechnung $n-2$ Iterationen nötig sind, ergibt sich für die Latenz: $L_x = x \cdot (n-2)$.

Da das Design so ausgelegt ist, daß die Knoten in Form einer Pipeline ($A_{Pipeline} = (n-2) \cdot A_{Knoten}$) bestehend aus $n-2$ Knoten arbeiten, kann zu jedem Takt eine Berechnung angestoßen werden. Der Rechendurchsatz R, die Anzahl der Operationen, die in 1.000 Zeiteinheiten auf 1.000.000 Flächeneinheiten ausgeführt werden können, ergibt sich deshalb aus: $R_x = \frac{1e+9}{x \cdot (n-2) \cdot A}$.

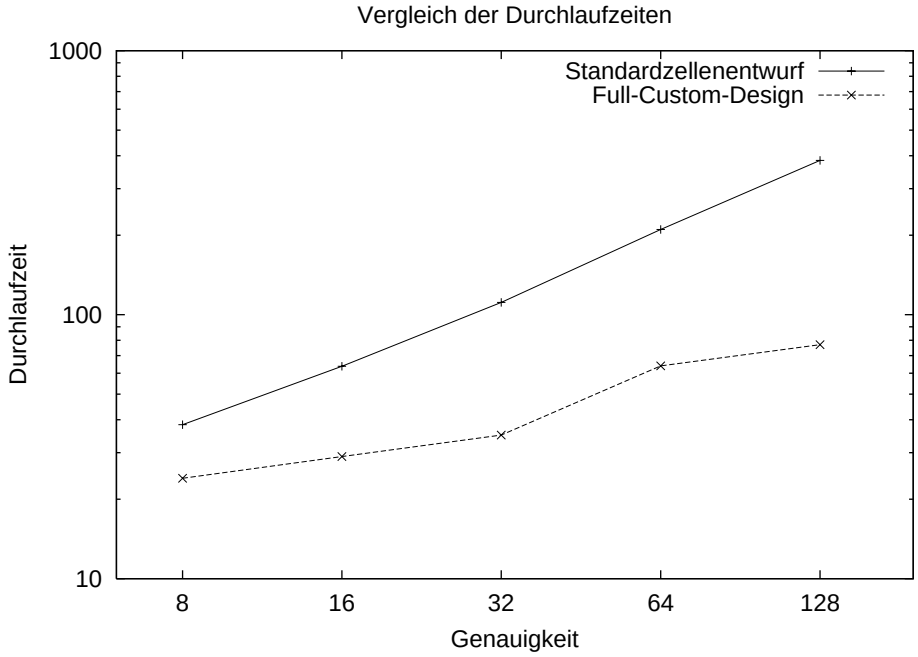

Abbildung 3.5: Vergleich der Durchlaufzeiten für die bitredundante Variante

Bei dieser Implementierungsvariante kann die Addition aufgrund der Kodierung in konstanter, von der Genauigkeit *n* unabhängiger Zeit durchgeführt werden. Dafür gibt es einen quasi logarithmischen Anteil bei der Einheit, die die Selektionsbedingung berechnet. Hier ist ein Oder mit einer Anzahl von Eingängen nötig, die der Genauigkeit *n* entspricht. Da in der Gatterbibliothek jedoch nur Oder-Gatter mit maximal 8 Eingängen enthalten sind, baut das Synthesetool ein entsprechend großes Oder-Gatter aus mehreren, hintereinandergeschalteten kleinen Oder-Gattern auf.

Bei der bitredundanten Variante trägt besonders der Block „< 0 ?“ zur Durchlaufzeiterhöhung für wachsende Genauigkeiten bei, siehe Abbildung 3.4. Neben dem Problem, das breite Oder aufzulösen, hat das Synthesetool die Schwierigkeit, für diesen Block ein adäquates Mapping auf die Standardzellenbibliothek zu finden. In beiden Fällen ist die Ursache in der Funktionalität dieses Blocks zu finden: Viele Signale treiben eine große Anzahl von Gattern. Hier wählt das Synthesetool den Weg, die Durchlaufzeit in einem erträglichen Rahmen zu halten. Dazu wird scheinbar für Genauigkeiten bis 32 Bit ein anderer Algorithmus für das Mapping verwendet, als für die Genauigkeiten ab 64 Bit. In Abbildung 3.5 sind die Kurven für die von Synopsys berechneten Durchlaufzeiten graphisch dargestellt. Die obere Kurve zeigt, daß bei steigender Genauigkeit die Durchlaufzeit regelmäßig steigt. Die untere Kurve stellt die Durchlaufzeit dar, die resultieren würde, wenn man jedes Gatter mit einer Durchlaufzeit von einer Zeiteinheit belegen würde. In der unteren Kurve ist eine Unregelmäßigkeit zwischen 32 und 64 Bit zu erkennen, die vermutlich das Ergebnis einer ungewöhnlichen Erhöhung des Gatteraufwandes durch das Synthesetool ist, um die Durchlaufzeit für das Standardzellenlayout in einem akzeptablen Rahmen zu halten.

Außerdem ist in Abbildung 3.5 erkennbar, daß, wie auch für die bitparalle Variante, eine Op-

timierung durch ein Full-Custom-Design eine qualitative Verbesserung bringen würde, da die Kurve für den Standardzellenentwurf linear steigt, während die für ein ideales Full-Custom-Design nur logarithmisch steigt.

Ein weiterer Nachteil, der sich auf die Durchlaufzeit auswirkt, ist, daß bei der Funktionsauswahl ein ganzer Vektor selektiert werden muß, was, wie auch schon bei der bitparallelen Implementierung, beim Standardzellenentwurf dazu führt, daß das Selektionssignal viele Multiplexer treiben muß. Bei der bitredundanten Implementierung mutlipliziert sich dieser Effekt noch, da die Vektoren aufgrund der gewählten Kodierung doppelt so breit sind.

3.3 Bitserielle Implementierung

In diesem Kapitel wird die Implementierung eines bitseriellen Knotens beschrieben. Im Gegensatz zur bitparallelen und bitredundanten Variante werden die Parameter seriell, Bit für Bit zeitlich nacheinander beginnend beim LSB (Least Significant Bit), an den Eingängen des Knotens hineingeschoben. Da für dieses Verfahren eine Kontrolleinheit benötigt wird, die die Anzahl der bereits verarbeiteten Bits registriert, werden mehr als nur die Flipflops zur Speicherung der Parameter und Funktion von Stufe zu Stufe benötigt. Der Kontroll-Block ist für diesen Ansatz deshalb kein Schaltnetz sondern ein Schaltwerk. Der serielle Bitstrom ist dabei wie in Kapitel 2.2.1 beschrieben und auch für die bitparallele Methode angewendet kodiert. Das Blockschaltbild für diese Variante ist in Abbildung 3.6 dargestellt.

Das Signal „sync“ dient zur Synchronisation der Kontrolleinheit mit den anliegenden Daten, damit die Kontrolleinheit im Falle des ersten anliegenden LSB z.B. einen gespeicherten Übertrag im Block „Addierer/Subtrahierer“ rücksetzen kann und im Falle des letzten MSB das Ergebnis des Addierers, der den Parameter y berechnet, als Selektionsbedingung verarbeiten kann.

Der auffälligste Unterschied zu den beiden anderen Varianten ist, daß die Eingangssignale x_i, y_i und z_i keine Vektoren der Breite der Genauigkeit n sind, sondern aufgrund der seriellen Verarbeitung unabhängig von der Genauigkeit nur ein Bit transportieren. Dadurch vereinfachen sich die einzelnen Funktionsblöcke erheblich. Der Block „Funktionsauswahl“, der für die parallelen Ansätze n bzw. $2 \cdot n$ 8-nach-1 Multiplexer enthält, muß für die bitserielle Variante nur noch einen Multiplexer bieten. Der „Addierer/Subtrahierer“-Block enthält nur noch einen Volladdierer. Zusätzlicher Aufwand entsteht durch die komplexere Kontrolleinheit und vor allem durch die zweiten Schieberegister zum temporären Speichern der Eingangsparameter x_i, y_i und z_i, die gebraucht werden, falls die Selektionsbedingung für die Bit-Algorithmen die Ausgabe der alten Werte fordert.

Ein weiterer Unterschied, der beim Übergang von der parallelen Verarbeitung aller Bits zur seriellen Verarbeitung ins Auge fällt, ist, daß die Flipflops zum Speichern der Operanden nicht mehr parallel geladen werden, sondern als Schieberegister arrangiert sind. Diese Anordnung ist nötig, da sowohl für die Bit- als auch CORDIC-Algorithmen vor der Ausführung der nächsten Iterationsstufe bekannt sein muß, ob die Selektionsbedingung erfüllt ist. Da diese Aussage erst getroffen werden kann, wenn das MSB (Most Significant Bit) berechnet ist, müssen die anderen Bits solange gespeichert werden.

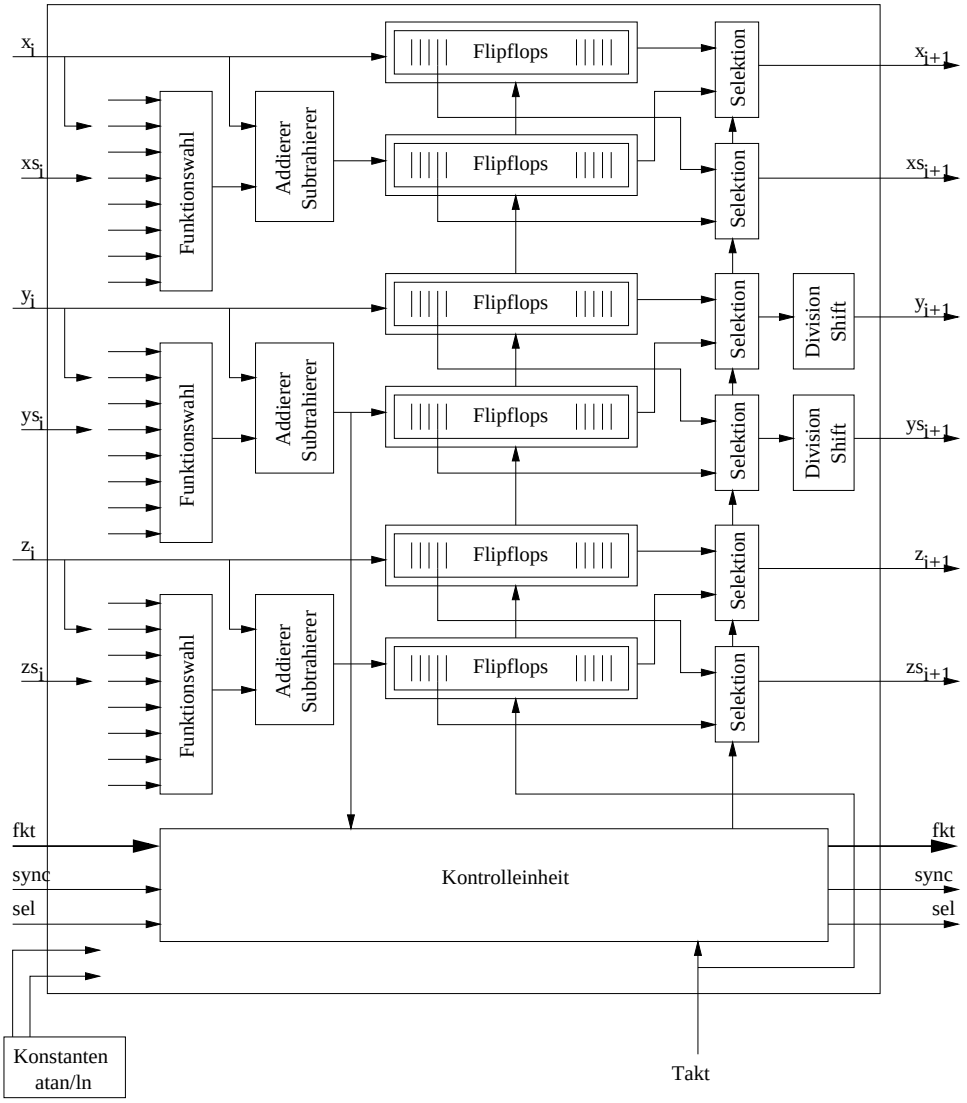

Abbildung 3.6: Schaltbild der bitseriellen Implementierung

Ein weiterer Unterschied zu den beiden anderen Varianten ist, daß die geshifteten Parameter nicht mehr aus den ungeshifteten erzeugt werden können, da man zum Zeitpunkt, wenn das LSB der Parameter anliegt, noch keinen Zugriff auf in der Zahlendarstellung weiter links liegender Bits hat. Da aber im vorangehenden Knoten wegen der Bestimmung der Selektionsbedingung bereits alle Bits eines Parameters bekannt sind, kann man die geshifteten Signale am Ausgang des vorangehenden Knotens erzeugen, indem das der Iterationsstufe entsprechende geshiftete Bit zusätzlich herausgeführt wird. Dies wird in Abbildung 3.6 dargestellt, indem der „Shift“-Block, der bei den anderen Varianten jeweils am Eingang eines Knotens war, durch jeweils ein Signal xs_{i+1}, ys_{i+1} und zs_{i+1} am Ausgang dargestellt wird, das im Schieberegister das gewünschte Bit abgreift. Dies entspricht der Anschauung, daß einem Knoten am Eingang nicht nur die Parameter x_i, y_i und z_i zur Verfügung gestellt werden, sondern zusätzlich deren geshiftete Pendants xs_i, ys_i und zs_i. Analog zu den zusätzlichen Signalen am Eingang, werden die geshifteten Parameter, wie eben beschrieben, am Ausgang unter den Namen xs_{i+1}, ys_{i+1}und

Tabelle 3.3: Charakteristik für bitserielle Implementierung

n	A	T	P	IO	R_T	R_P	L_T	L_P
008	752	13.45	14	14/11	2059.76	1978.85	645.60	672.00
016	1147	13.96	15	14/11	278.80	259.47	3127.04	3360.00
032	1931	13.96	15	14/11	38.64	35.96	13401.60	14400.00
064	3499	13.96	15	14/11	5.15	4.80	55393.28	59520.00
128	6635	13.96	15	14/11	.66	.62	225146.88	241920.00

zs_{i+1} geliefert.

Der Rechendurchsatz gemessen in Iterationen pro Zeit ist linear zur Genauigkeit n. Der längste Pfad zwischen zwei Flipflops in dieser Variante ist zwar konstant, da aber für jedes zusätzliche Bit Genauigkeit ein weiteres Flipflop im Schieberegister eingefügt wird, ist die tatsächliche Zeit zur Berechnung einer Iteration $n \cdot Durchlaufzeit$.

Die Anzahl der Ein-/Ausgabeleitungen für den bitseriellen Ansatz ist eine von der Genauigkeit n unabhängige Konstante: 11 Ein- und Ausgangssignale für die Parameter, deren geshiftete Pendants, der Funktionsauswahl, einem Selektsignal und einem Synchronisationssignal, einen weiteren Eingang für den Takt und, im Gegensatz zu den parallelen Varianten, zwei Eingänge für die Konstanten. Da die Konstanten für diesen Ansatz ebenfalls einen Automaten benötigen, der die Konstante seriell Bit für Bit hinausschiebt, können diese Signale nicht verdrahtet werden.

In Tabelle 3.3 sind die Eingangs des Kapitels 3 beschriebenen Parameter eingetragen. Bei der bitseriellen Implementierung wird in n Takten eine Iteration des Algorithmus' durchgeführt. Da für eine komplette Berechnung $n-2$ Iterationen nötig sind, ergibt sich für die Latenz: $L_x = n \cdot x \cdot (n-2)$.

Da das Design so ausgelegt ist, daß die Knoten in Form einer Pipeline ($A_{Pipeline} = (n-2) \cdot A_{Knoten}$) bestehend aus $n-2$ Knoten arbeiten, kann zu jedem n-ten Takt eine Berechnung angestoßen werden. Der Rechendurchsatz R, die Anzahl der Operationen, die in 1000 Zeiteinheiten auf 1000000 Flächeneinheiten ausgeführt werden können, ergibt sich deshalb aus: $R_x = \frac{1e+9}{n \cdot x \cdot (n-2) \cdot A}$.

Im Gegensatz zur bitparallelen und bitredundanten Implementierung ist die maximale Durchlaufzeit und Pfadlänge bei der bitseriellen quasi konstant, da sich für diese Implementierungsvariante für zusätzliche Genauigkeitsbits keine Erweiterungen bei der Funktionsauswahl und den Addier-/Selektionseinheit ergibt, sondern nur eine Erweiterung der Schieberegister und des Steuerautomaten.

3.4 Vergleich der Knotentypen

Dieses Kapitel vergleicht die drei dargestellten Knotentypen und ermittelt unter welchen Umständen welche Variante die günstigste ist. Für die Qualität auf funktionaler Ebene sind

Tabelle 3.4: Vergleich des Rechendurchsatzes

	R_T			R_P		
n	par	red	ser	par	red	ser
008	3978.65	2070.22	2059.76	6418.89	3303.73	1978.85
016	446.69	267.13	278.80	1137.68	587.14	259.47
032	51.06	35.73	38.64	221.00	113.66	35.96
064	5.40	4.60	5.15	42.97	15.12	4.80
128	.62	.61	.66	8.66	3.08	.62

zwei Größen relevant: Der Rechendurchsatz pro Chipfläche und die Latenz von Auftrag zu Ergebnis, die Zeit, die die Berechnung einer Funktion benötigt. Diese Größen sind jeweils in den vorangegangenen Kapiteln in den Tabellen 3.1, 3.2 und 3.3 eingetragen.

Eine Anforderung wird an alle Implementierungsvarianten gleichermaßen gestellt: Die Versorgung der Flipflops mit dem Taktsignal. Da sehr viele Flipflops mit dem gleichen Signal versorgt werden müssen, ist dies kein triviales Problem. Gerade bei den Schieberegistern des bitseriellen Ansatzes ist ein korrektes Timing der Takteingänge für die Funktion des Schaltkreises nötig. Da diese Hürde jedoch für alle Implementierungsvarianten gleichermaßen überwunden werden muß, wird an dieser Stelle nur darauf hingewiesen, daß für die bitserielle und bitredundante Variante im Gegensatz zur bitparallelen Variante ca. doppelt so viele Flipflops zu versorgen sind. Ansonsten wird dieses Problem nicht weiter beleuchtet.

Im folgenden wird davon ausgegangen, daß mehrere Pipelines parallel auf einem Schaltkreis integriert sind und parallel arbeiten. Wird auf diesem System z.B. eine Fast Fourier Transformation (FFT) berechnet, hängt die Effizienz der Berechnung davon ab, wie gut diese Funktionalität parallelisiert werden konnte, um die Pipelines optimal zu füllen. Gelingt dies zu 100%, hängt die Leistung (Anzahl der Operationen pro Zeiteinheit) des Systems nicht von der Latenz ab, die durch die Ermittlung eines Ergebnisses, das für weitere Berechnung nötig ist, auftritt. In diesem Fall ist die Variante mit dem höchsten Rechendurchsatz pro Chipfläche die günstigste. In Abbildung 3.7 und 3.8 sind jeweils visualisierte Daten der drei Implementierungsvarianten im Vergleich zu sehen. Abbildung 3.7 zeigt die Rechendurchsätze, für den Standardzellenentwurf, Abbildung 3.8 die für ein optimales Full-Custom-Design.

Da die Kurven in bestimmten Bereichen sehr dicht beieinander liegen, sind sie auch in Form einer Tabelle dargestellt. In Tabelle 3.4 ist jeweils der Rechendurchsatz der drei Implementierungsvarianten in Abhängigkeit von der Genauigkeit n gegenübergestellt. Die erste Spalte zeigt an, für welche Genauigkeit die folgenden Werte dieser Zeile gültig sind. Das folgende Tripel enthält die Werte für den Rechendurchsatz der drei Imlementierungsvarianten, der mit den vom Synthesetool berechneten Durchlaufzeiten bestimmt wurde, das nächste Tripel den Rechendurchsatz für die drei Varianten, wenn man den längsten Pfad als Maß für die Latenz einer Berechnung verwendet. Das erste Tripel steht für den Entwurf mit einer Standardzellenbibliothek, das zweite für eine Näherung im Falle eines Full-Custom-Designs.

Die Kurven in Abbildung 3.7 zeigen, daß für einen Standardzellenentwurf der beste Rechendurchsatz für den bitparallelen Ansatz zu erwarten ist, während sich der bitredundante und

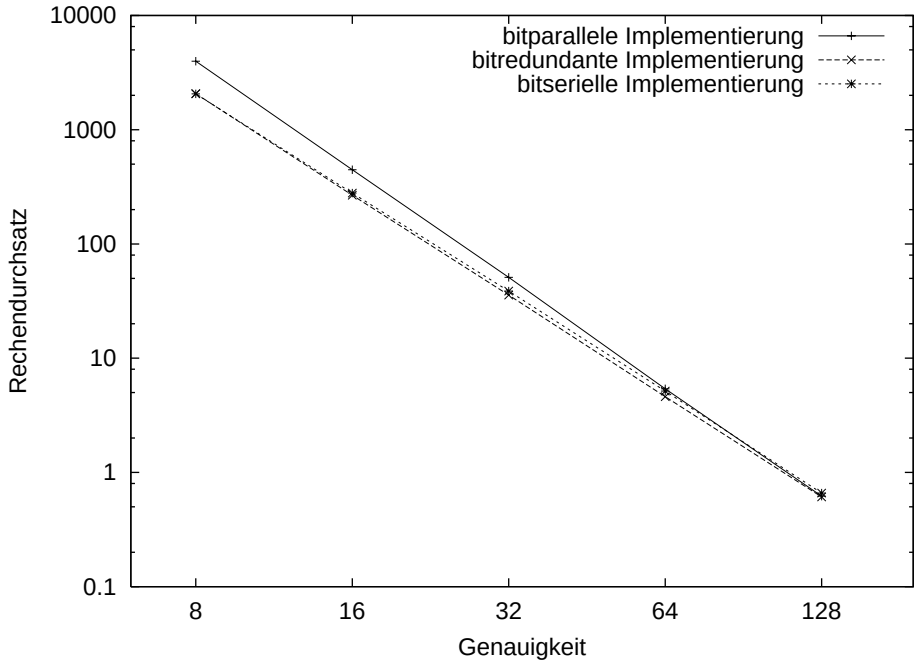

Abbildung 3.7: Vergleich der Rechendurchsätze der Standardzellenentwürfe

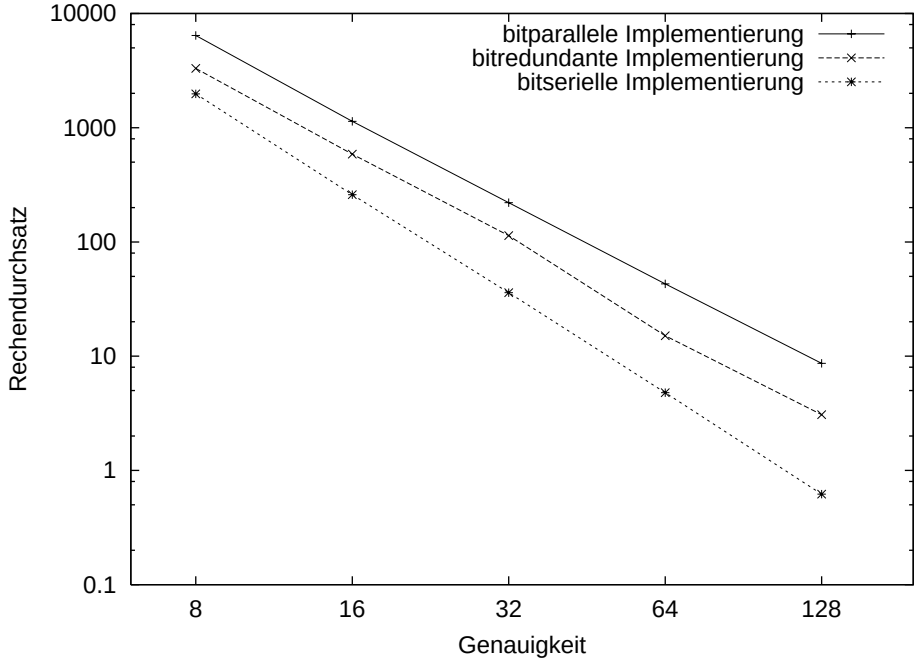

Abbildung 3.8: Vergleich der Rechendurchsätze der Full-Custom-Designs

bitserielle nicht viel nehmen aber unterhalb des bitparallelen liegen. Mit steigender Genauigkeit liegen die drei Varianten immer enger beieinander, bis sie schließlich für 128 Bit Genauigkeit quasi zusammenfallen.

Für den optimierten Rechendurchsatz, der für ein Full-Custom-Design zu erwarten und in Abbildung 3.8 dargestellt ist, liegt der bitparallele Ansatz gegenüber dem bitredundanten ca. um den Faktor zwei vorn. Da der bitredundante Ansatz doppelt so viele Flipflops zur Speicherung der Operanden braucht, ist zu folgern, daß die Implementierung des „Borrow-Save-Adders“ mit der zusätzlichen Logik zur Berechnung der Selektionsbedingung dem Aufwand für den „Conditional-Sum-Adder“ entspricht.

Vergleicht man den bitparallelen Ansatz mit dem bitseriellen, ist er je nach Genauigkeit um den Faktor 3 bis 10 besser, wobei das Verhältnis für größere Genauigkeiten steigt.

Vergleicht man die Kurven von Abbildung 3.7 mit 3.8 kann man durch den Unterschied im Gefälle der Kurven erkennen, daß die Kurven für das Full-Custom-Design für die bitparallele und bitredundante Implementierung nicht so steil fallen wie für den Standardzellenentwurf, wodurch eine höhere Effizienz für die Full-Custom-Design-Systeme zu erwarten ist. Allerdings wird darauf hingewiesen, daß die Kurven für das Full-Custom-Design nur eine Näherung für den optimalen Fall darstellen und deshalb nur eine obere Schranke für die Güte gesehen werden können. Eine Aussage über die tatsächliche Effizienz eines Full-Custom-Design kann erst getroffen werden, wenn eine Abbildung auf Gatter mit stärkeren Treibern durchgeführt wurde.

Gelingt es nicht, die Pipelines zu 100% zu beschäftigen, geht die Latenz und der durchschnittliche Füllgrad der Pipelines in die Berechnung des Rechendurchsatzes ein. Für diesen Fall könnte man beliebig mit den Werten für Latenz und Füllgrad spielen, um Werte für den Rechendurchsatz zu ermitteln. Darauf wurde in dieser Arbeit verzichtet, da von algorithmischer Seite immer eine optimale, 100 prozentige Auslastung der Pipelines angestrebt wird. Da für Echtzeit-Systeme die Latenz jedoch eine Rolle spielen kann, wird sie vergleichend für die Implementierungsvarianten in Tabelle 3.5 dargestellt. Ähnlich wie in Tabelle 3.4 enthält diese Tabelle sieben Spalten. Die erste gibt wieder an, für welche Genauigkeit die Werte einer Zeile gültig sind. Die nächsten beiden Tripel enthalten die Latenzen für die jeweils drei Implementierungsvarianten, wobei die Werte des ersten Tripels für einen Standardzellenentwurf gültig sind und das zweite Tripel eine Näherung für ein Full-Custom-Design.

In Abbildung 3.9 und 3.10 sind die Daten aus Tabelle 3.5 graphisch aufgearbeitet. Da die Latenz indirekt auch in den Rechendurchsatz eingeht, fallen die Kurven entsprechend aus: Auch wenn die Latenz das ausschlaggebende Kriterium für die Wahl der Implementierungsvariante ist, schneidet die bitparallele Variante dicht gefolgt von der bitredundanten am besten ab: Sie liefern die Ergebnisse mit geringerer Latenz. Vergleicht man Full-Custom-Design mit Standardzellenlayout, läßt sich, wie auch bei der Betrachtung des Rechendurchsatzes, aus der unterschiedlichen Steigung der Geraden ableiten, daß das Full-Custom-Design dem Standardzellenentwurf vorzuziehen ist, weil die Latenzen für den Standardzellenentwurf eine größere Steigung haben, als für das Full-Custom-Design und somit eine wesentliche Verbesserung der Latenzen bewirken würde.

Wenn ein möglichst einfacher Entwurfsprozeß im Vordergrund für die Implementierung der Bit-Algorithmen steht, ist die bitserielle Variante die günstigste. Für die bitserielle Variante kann auch für hohe Genauigkeiten n ein Standardzellenentwurf zur Integration der Algorithmen auf einem Chip hergenommen werden, da mit steigender Genauigkeit keine optimierenden Eingriffe in Treiberstufen einzubringen sind, da immer eine konstante Anzahl von Gattern, z.B. die

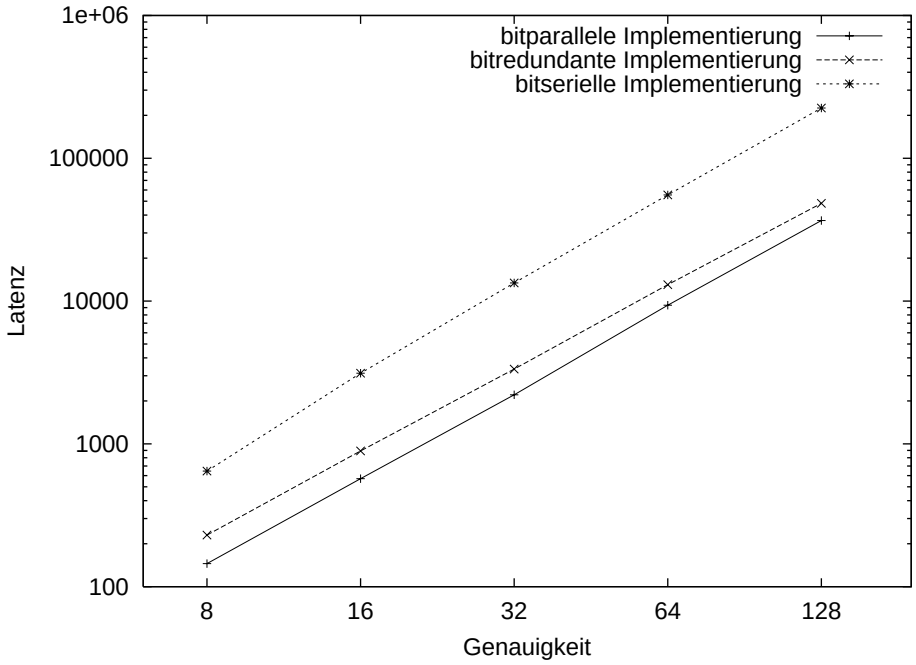

Abbildung 3.9: Vergleich der Latenz der Standardzellenentwürfe

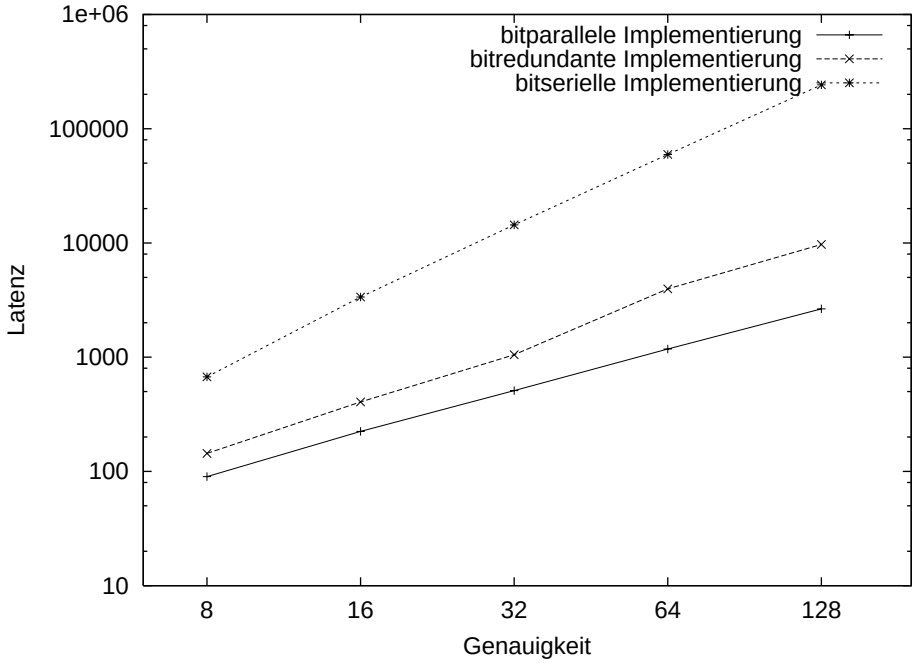

Abbildung 3.10: Vergleich der Latenz der Full-Custom-Designs

Multiplexer für die Funktionsauswahl, geschaltet werden muß.

Betrachtet man die Integrierbarkeit der Knoten in einem Netz, wie es zur Erfüllung der Funktionalität der Algorithmen nötig ist, ist kein zusätzlicher Aufwand erforderlich, wenn man die

Tabelle 3.5: Vergleich der Latenzen

	L_T			L_P		
n	par	red	ser	par	red	ser
008	145.20	229.80	645.60	90.00	144.00	672.00
016	570.50	892.36	3127.04	224.00	406.00	3360.00
032	2207.40	3339.30	13401.60	510.00	1050.00	14400.00
064	9365.10	13036.74	55393.28	1178.00	3968.00	59520.00
128	36658.44	48395.34	225146.88	2646.00	9702.00	241920.00

Tabelle 3.6: Vergleich der Anzahl von Ein-/Ausgabeleitungen

n	par	red	ser
008	28/27	52/51	14/11
016	52/51	100/99	14/11
032	100/99	196/195	14/11
064	196/195	388/387	14/11
128	388/387	772/771	14/11

Knoten einfach nur in Reihe verschaltet, um die Iterationsstufen nacheinander durchlaufen zu können. Wird jedoch z.B. gefordert, daß die Knoten in einem rekonfigurierbaren Netz eingebettet sind, so daß im Falle eines fehlerbehafteten Knotens dieser Umgangen werden kann, gewinnt die Anzahl der Ein-/Ausgabeleitungen pro Knoten an Bedeutung. Ein Rekonfigurationsschaltwerk für die bitserielle Variante kommt mit sehr wenigen, von der Genauigkeit n unabhängigen konstanten Anzahl von Umschalt-Komponenten aus, während beim bitparallelen und bitredundanten Ansatz die Anzahl der Umschalt-Komponenten mit steigender Genauigkeit n aufgrund zunehmender Anzahl von umzuschaltenden Signalen linear wächst. Eine Gegenüberstellung der Anzahl der Ein-/Ausgabeleitungen der Varianten ist in Tabelle 3.6 dargestellt.

Sind in einer Standardzellenbibliothek bereits Komponenten mit starken Treibern zur Umschaltung der Multiplexer, mit denen mehrere Bit breite Vektoren ausgewählt werden können, enthalten, ist man dem Full-Custom-Designs einen großen Schritt näher.

Fazit: Sieht man von der leichteren Integrierbarkeit der bitseriellen Variante in einem rekonfigurierbaren Netz einmal ab, ist die bitparallele Variante die effizienteste. Sie schlägt die beiden anderen Varianten in den Disziplinen Rechendurchsatz und Latenz. Bezieht man stärkere Treiber in die Abwägung mit ein, fällt der Vorteil noch weiter zugunsten der bitparallelen Variante aus. Die bitredundante Variante folgt beim Rechendurchsatz ca. um den Faktor 2, die bitserielle ca. um den Faktor 4 bis 10. Bei der Latenz unterscheiden sich bitparalleler und bitredundanter Ansatz ca. um Faktor 2-5, während sich bitparalleler und bitserieller Ansatz um den Faktor 10-100 unterscheiden.

Kapitel 4

Implementierung der Netze

Dieses Kapitel beschäftigt sich mit der Verbindungstopologie zwischen den einzelnen Knoten, die im vorangegangenen Kapitel 3 dargestellt wurden. Es werden vier Möglichkeiten gegenübergestellt, wobei in diesem Kapitel nur entwurfsspezifische Parameter, wie z.B. Funktionalität, Komplexität oder resultierende Chipfläche betrachtet werden. Eine Analyse der Fehlertoleranzcharakteristiken wird im Kapitel 5 dargestellt. Jede Netztopologie wird bis auf eine Ausnahme mit jedem der drei Knotentypen evaluiert werden, woraus folgt, daß letztlich verschiedene Systeme gegeneinander verglichen werden müßten. Da sich eine Netztopologie nur auf sehr ineffiziente Weise mit dem bitseriellen Knoten hätte implementieren lassen, wurde von der Implementierung dieser einen Kombination abgesehen und somit resultieren nur 11 statt 12 verschiedene Systeme.

Da das Ziel verfolgt wird, Bit-/CORDIC-Algorithmen direkt in Hardware zu integrieren, die zusätzlich durch Fehlertoleranz-Mechanismen ausfallsicherer sein soll, wird die Wahl der FT-Mechanismen durch den erforderlichen Hardwareaufwand, den eine solche Maßnahme mit sich bringt, beschränkt. In Kapitel 4.4.4 wird anhand eines Beispiels beschrieben, welche Randbedingungen zur Ablehnung der Implementierung einer Methode führen kann.

In diesem Kapitel werden nur Fehlertoleranzmaßnahmen dargestellt, die basierend auf einem Vergleich mindestens zweier parallel arbeitender Knoten feststellen, ob ein Ergebnis als korrekt gewertet wird. Es wurden auch Verfahren, z.B. Residuen- und Hamming-Code, untersucht, um direkt im Knoten eine Fehlerdetektion oder gar -korrektur zu ermöglichen. Der Hardware-Overhead für diese Verfahren überstieg jedoch den Faktor zwei bei weitem, so daß nur vergleichende Verfahren, die einen Hardware-Overhead um den Faktor zwei mit sich bringen, Einzug in diese Arbeit fanden.

Die in diesem Kapitel dargestellten Varianten zur Integration der Bit-/CORDIC-Algorithmen in einem Modul werden unabhängig vom Typ des Netzwerks mit einem einheitlichen Interface zum Anwender versehen. Abgesehen von der erleichterten Implementierung dieser Moduln in einem System bietet das einheitliche Interface den Vorteil, alle Varianten der Netze mit den Varianten der Knoten gegenüberzustellen, ohne die ermittelten Parameter wie z.B. Durchsatz auf evtl. unterschiedliche Interfaces normieren zu müssen. Das einheitliche Interface aller Netztopologien hat den weiteren Vorteil, daß jedes Netz mit seinen Knoten aufgrund der gleichen

Testbench auf korrekte Funktion untersucht werden kann, also nur einmal Testpattern und ihre erwarteten Antworten entwickelt werden müssen, mit denen dann alle Kombinationen von Knoten und Netztopologie validiert werden können. Die Beschreibung des Interfaces findet sich in Kapitel 4.1.

Im Kapitel 4.2 wird darauf eingegangen, wie die verschiedenen Systeme auf korrekte Funktion validiert wurden. Das Kapitel 4.3 beschreibt die Strategie, wie der fertige Chip auf Fabrikationsfehler hin testbar gemacht wurde. Im Gegensatz zu konventioneller, nicht fehlertoleranter, Hardware wird man hier mit zusätzlichen Problemen konfrontiert.

In den folgenden Kapiteln werden die verschiedenen Netztopologien vorgestellt. Das in Kapitel 4.4.1 beschriebene, funktionale Netz verbindet lediglich die Knoten, damit sie die in Kapitel 2 beschriebene Funktionalität erfüllen können. Darüber hinaus wird dieses Netz im Kapitel 5 dazu benutzt, die Fehlertoleranzeigenschaften der verschiedenen Knotentypen, bitparallel, bitredundant und bitseriell, zu analysieren.

Die beiden Roll-Backward Netze integrieren ein Fehlertoleranzverfahren, das transiente Fehler korrigiert, indem es Fehler aufgrund einer Duplizierung der Funktionen und anschließendem Vergleich erkennt und die Operation, bei der der Fehler auftrat, solange wiederholt, bis die Operation korrekt durchgeführt wurde. Diese Netze werden in den Kapiteln 4.4.2 und 4.4.3 dargestellt. Sie unterscheiden sich in der Anzahl, um wieviele Stufen das Rollback durchgeführt wird: Beim Macro-Rollback ist lediglich ein Komparator am Ende der Pipeline dafür verantwortlich, einen Fehler zu erkennen, und gegebenenfalls eine fehlerhafte Operation nochmal von ganz vorne anzustoßen. Beim Micro-Rollback existiert hinter jeder Stufe der Pipeline ein Komparator zur Fehlererkennung und entsprechenden Rekalkulation auf feinerer Granularität als beim Macro-Rollback.

Ursprünglich war geplant, alternativ zum Roll-Backward ein Roll-Forward Netz zu implementieren, um die beim Roll-Backward auftretenden Latenzen im Fehlerfall zu minimieren. In der Entwurfsphase dieses Netzes hat sich jedoch herausgestellt, daß ein simpler Komparator zur Erkennung der Fehler nicht ausreicht, weil bei einem Roll-Forward zusätzlich zur Fehlerdetektion eine Fehlerlokalisation benötigt wird. Entsprechende Implementierungsversuche einer Hardwareerweiterung, die diese Funktionalität bietet, ergaben, daß sich zum hohen Entwurfsaufwand für die Steuerautomaten dieses Netzes ein Hardware-Overhead gesellt, der den Chipflächenbedarf weit mehr als um den Faktor drei, vermutlich 4 bis 5, wachsen lassen würde. Durch diesen immensen Overhead motiviert, wurde ein entsprechendes TMR-Netzwerk implementiert, das mit einer etwas mehr als 3-fachen Chipfläche auskommt. Aufgrund der einfacher strukturierten Voter im Vergleich zu den komplexen Steuerautomaten des Roll-Forward-Netzes bietet es bessere Fehlertoleranzeigenschaften: Sowohl die Latenz bei transienten Fehlern als auch Korrekturverhalten bei permanenten Fehlern sind günstiger. Eine detaillierte Beschreibung des TMR-Netzes findet sich in Kapitel 4.4.5.

Im abschließenden Kapitel 4.5 werden charakteristische Daten wie z.B. Chipflächenbedarf, Latenzen, Entwurfskomplexität usw. zusammengetragen. Die Analyse der Fehlertoleranzcharakteristika der Netze und eine entsprechende Bewertung wird dann im folgenden Kapitel 5 durchgeführt.

4.1 Interface der Netze

Um die zahlreichen Implentierungsvarianten gegeneinander vergleichen und einfach testen zu können, wurde die Entwurfsentscheidung gefällt, alle Varianten mit einem einheitlichen Interface zu versehen, welches in diesem Kapitel beschrieben wird.

Die Netze wurden, wie auch schon die Knoten, durch ein C-Programm erzeugt, das als Eingabe die Rechengenauigkeit, mit der gerechnet werden soll, die Anzahl der Iterationsstufen, i.a. 2 weniger als die Anzahl der Bits der Genauigkeit, und die Anzahl der Bits für den Index erhält. In dieser Arbeit wurden Netze mit 8 Bit Rechengenauigkeit, 6 Iterationsstufen und 8 Bits für den Index untersucht.

Das gewählte Interface ähnelt dem, welches im RISC-Prozessor von Motorola M88k für die Integer-, Floating-Point- und Data-Unit eingesetzt wurde. Es ist in der Lage, pro Taktzyklus eine Operation abzusetzen und dementsprechend nach einer gewissen Latenz die Ergebnisse ebenso nach jedem Taktzyklus zu erzeugen, natürlich nur unter der Bedingung, daß der Funktionsblock zwischen der Ein- und Ausgabe in der Lage ist, Daten in entsprechender Geschwindigkeit zu verarbeiten. Für jede abgesetzte Operation wird ein Index durch die Stufen der Pipeline gezogen, der am Ausgang der Pipeline die Anzeige ermöglicht, in welchem Register des Prozessors, in dem diese Pipeline integriert ist, ein Ergebnis abzulegen ist. Dieser Mechanismus kann bei Integration der Pipeline in einen Prozessor für einen Scoreboard-Mechanismus verwendet werden.

Abbildung 4.1: Blockschaltbild des Interface

Die Eingabe besteht aus 2 Operanden zu je 8 Bit, einem 8 Bit breiten Index, einem 3 Bit breiten Vektor zur Auswahl der Operation, log, exp, sin usw. und den beiden Handshake-Signalen action und busy, siehe auch 4.1. Die Ausgabe des Interfaces besitzt nur die Handshake-Signale: ready, welches durch den 8 Bit breiten Index und das 8 Bit Ergebnis ergänzt wird. Das Zeitverhalten des Interfaces wird über die Handshake-Signale abgehandelt. Mit der Aktivierung des Signals action müssen gleichzeitig Operanden, Index und Operationsauswahl bereitgestellt werden, die zu steigender Flanke des Taktsignals vom Netz übernommen werden, siehe 4.2. Zu diesem Zeitpunkt muß im Netz auch entschieden werden, ob zur nächsten steigenden Flanke des Taktsignals wieder Daten angenommen werden können. Wenn ja, bleibt busy passiv, wenn

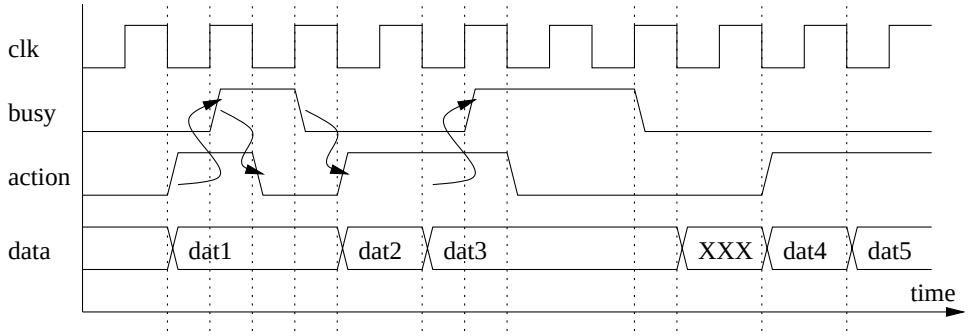

Abbildung 4.2: Zeitverhalten des Interface

nein, muß busy aktiviert werden, worauf die Daten am Eingang solange gehalten werden, bis busy wieder passiv wird. Am Ausgang des Netzes wird durch das Signal ready angezeigt, ob Daten zur Speicherung anliegen. Ist ready zum Zeitpunkt einer fallenden Flanke des Taktsignals clk aktiviert, müssen das Ergebnis und der Index abgeholt werden. Ist clk nicht aktiviert, liegen keine Daten zur Verarbeitung an.

Der im vorangegangenen beschriebene Teil des Interface ist allen Implementierungsvarianten gemeinsam, was, wie schon gesagt, es ermöglicht, alle Netze in ein und dieselbe Testbench einzubetten. Zu diesem Teil des Interfaces kommen noch die Schnittstellen zu den Knoten hinzu. Dazu trägt jedes Netz entsprechend der Anzahl der zu integrierenden Knoten quasi einen Sockel, in den ein Knoten hineingesteckt wird. Dieser Bereich von Ein- und Ausgängen ist abhängig vom verwendeten Knotentyp und variiert in der Anzahl parallel arbeitender Knoten. Detaillierte Beschreibungen dieser Teile der Netz-Interfaces befinden sich in den entsprechenden Unterkapiteln zu den Netzen.

4.2 Validierung der Modelle

Dieses Kapitel stellt dar, wie die generierten VHDL-Modelle auf korrekte Funktion hin validiert wurden und welche Maßnahmen schon beim Design beachtet wurden, um möglichst schnelle Entwicklung zu erzielen.

Zunächst wurden die in Kapitel 2 dargestellten Algorithmen in der Programmiersprache C implementiert. Das C-Programm wurde überprüft, indem es für eine Reihe von Operationen die Ergebnisse berechnet hat und diese dann mit den entsprechenden Operationen des Rechners verglichen wurden.

Mit Hilfe dieses C-Programms wurden Stimuli und expected Responses für die Netze generiert. Somit wurden die gleichen Stimuli in alle Testbenches integriert. Da alle Netze auf diese Weise mit den gleichen Stimuli erregt wurden und von ihnen gleiche Antworten zu erwarten waren, war es möglich, die alle Netze mit einem C-Programm auf korrekte Funktion zu prüfen, ohne mit neuen Fehlern durch falsch implementierte Stimuli oder expected Responses rechnen zu müssen.

Die Idee, alle VHDL-Modelle von einem einzigen C-Programm generieren zu lassen, in Kombination mit der Disziplin identische Teile der VHDL-Modelle vom gleichen C-Code erzeugen zu lassen, hatte zur Folge, daß bei der Fehlerbehebung im C-Programm oft mehrere Modelle gleichzeitig korrigiert wurden, obwohl ein Fehler nur in einem VHDL-Modell entdeckt wurde.

Die Fehlersuche in den VHDL-Modellen selbst wurde durch den VHDL-Simulator von Modeltech unterstützt. Dieser bietet ein GUI, indem man den zeitlichen Verlauf aller Signale visualisiert bekommt. Um die Netze funktional zu validieren, wurden sie in eine Testbench, siehe Abbildung 4.3, eingebettet, die einerseits eine Reihe von Stimuli generiert und andererseits auch die Antworten des Netzes, dem Ist-Zustand, mit den erwarteten Antworten (expected responses) vergleicht und Auskunft über Übereinstimmung gibt. Da die Netze alle mit dem gleichen Interface versehen sind, mußte nur eine Testbench entwickelt werden, mit der alle Netze überprüft werden konnten.

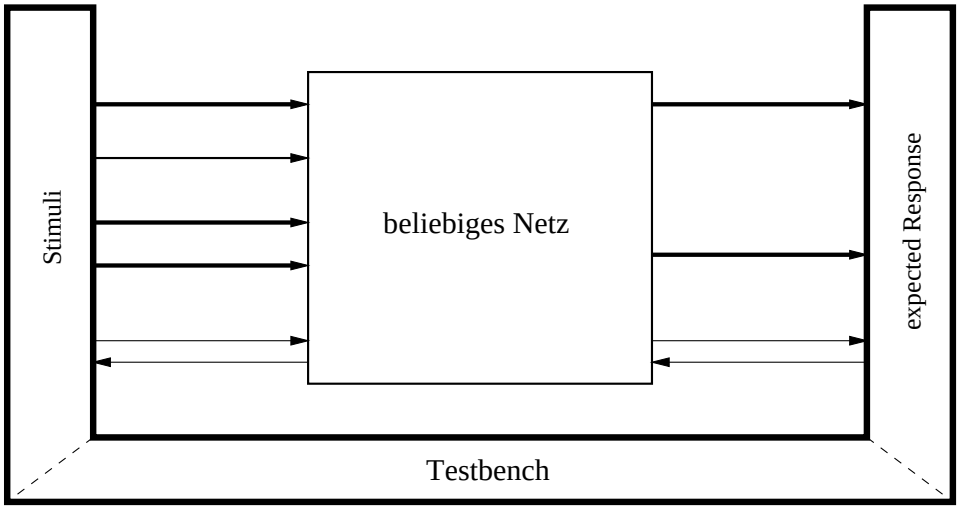

Abbildung 4.3: Struktur der Testumgebung

Für die Stimuli wurden 64 Operationen zufällig mit der Einschränkung, daß alle möglichen acht Funktionen (log, exp, sqrt, Multiplikation, Division, sin, cos, arctan) gleich häufig auftreten, ausgewählt. Die Stimuli werden von einem weiteren C-Programm erzeugt, das sich eines Pseudo-Zufallszahlengenerators bedient. Der Pseudo-Zufallszahlengenerator hat die nützliche Eigenschaft, daß er immer wieder die gleiche Reihe von Zufallszahlen erzeugt, was den Vorteil hat, daß die im folgenden beschriebenen Experimente auch zu späteren Zeitpunkten rekonstruierbar sind, weil bei jedem Aufruf des C-Programms die gleichen zufälligen Stimuli generiert werden. Allerdings sind auf verschiedenen Rechner-Architekturen verschiedene Zufallszahlengeneratoren implementiert, wodurch man auf einem Intel-PC andere Stimuli erhält, als auf einer Sparc-Sun. In den folgenden Kapiteln wurden die Stimuli ausschließlich auf der Sparc-Architektur generiert, weil sowohl der VHDL-Compiler/-Simulator von Modeltech als auch das Synthese-Tool, welches im folgenden Kapitel eingesetzt wird, nur auf dieser Architektur verfügbar waren.

Mit den quasi zufällig generierten Stimuli als Eingabe für das eingangs dieses Kapitels beschriebene C-Programm werden die expected Responses berechnet und ebenfalls der Testbench

Tabelle 4.1: Deckungsrate der Stimuli

Implementierungsvariante	Faultcoverage
Bitparallele Implementierung	47.0 % (14057 von 29930)
Bitredundante Implementierung	63.4 % (18544 von 29270)
Bitserielle Implementierung	69.5 % (6886 von 9910)

zugefügt. Da am Anfang des Entwurfstadiums auch Fehler in den Knoten behoben werden mußten, wurde die Testbench erweitert, um auch die Signale an den Ausgängen der im Netz eingebetteten Knoten beobachten zu können. Dies war nur für das funktionale Netz nötig, da in allen anderen Netzen die gleichen Knoten eingesetzt werden wie für das funktionale.

Um die korrekte Funktion von Fehlertoleranzmechanismen zu überprüfen, müssen diese aktiviert werden, damit man das daraus resultierende Verhalten beobachten und somit validieren kann. Ein Feature des Modeltech VHDL-Simulators bietet die Möglichkeit, Signalwerte manuell während der Simulation zu verändern, was quasi die Injektion eines Fehlers ermöglicht. Diese Möglichkeit kombiniert mit dem Wissen, welche Fehler von einer FT-Maßnahme kompensiert werden sollen, bietet wieder die Vergleichsmöglichkeit von Soll- und Ist-Zustand der Implementierung des Systems im Fehlerfall. Dadurch kann die Funktion der FT-Maßnahme validiert werden, indem ermittelt wird, ob eine Fehlertoleranzmaßnahme für Fehler, die eine Aktivierung der FT-Maßnahme laut Entwurf erzwingen, auch anspringt.

Die Validierung der VHDL-Modelle wurde ausschließlich simulativ durch Vergleich des Soll- und Ist-Zustandes für zufällig gewählte Stimuli durchgeführt. Aufgrund der Komplexität des Systems kam eine formale Verifikation nicht in Frage.

4.3 Test des Chips

Dieses Kapitel beschäftigt sich mit dem Test des gefertigten Chips. Es zeigt die Qualität der gewählten Stimuli für den Fall, daß man sie zum Testen eines Chips heranziehen möchte. Da das Ziel dieser Arbeit ist, FT-Maßnahmen auf Hardware-Ebene gegeneinander zu vergleichen, wurde in den Entwurf von Testpattern für die Fertigung des Chips nicht viel Energie investiert.

Um die Deckungsrate der in Kapitel 4.2 beschriebenen Stimuli in Bezug auf das Stuck-At-Fehlermodell zu bestimmen, wurde aus den VHDL-Modellen der funktionalen Netze jeweils ein Gattermodell synthetisiert. Für das Gattermodell in Kombination mit einer allgemeinen Gatterbibliothek, die auch die Injektion von Stuck-At-Fehlern über die gesamte Simulationszeit erlaubt, wurde in jeweils einer Simulation für jeden möglichen Stuck-At-Fehler bestimmt, ob dieser die expected Responses verfälscht. Die Auswertung der Simulationen hat ergeben, daß die Abdeckung je nach Knotentyp weit unter 70% liegt, für einen Fertigungsprozeß also nicht ausreicht. Die genauen Zahlen können Tabelle 4.1 entnommen werden. In Abhängigkeit vom Knotentyp ist der prozentuale Anteil angegeben und in Klammern die Anzahl entdeckter und möglicher Fehler.

Fehlertolerante Hardware muß unter Berücksichtigung der Erkennbarkeit, ob ein Fehlertoleranzmechanismus anspringt, entworfen werden. Wäre dies nicht erkennbar, könnte beim Test des Chips nicht unterschieden werden, ob eine Fehlertoleranzmaßnahme einen Fertigungsfehler kompensiert oder ob der Chip wirklich fehlerfrei arbeitet. Im Falle der Kompensation eines Fertigungsfehlers ist das Netz in den meisten Fällen nicht mehr in der Lage, weitere Fehler zu kompensieren, und damit ist der Schaltkreis nicht funktionstüchtig im Sinne der Spezifikation, daß einzelne Fehler toleriert werden. Bei den fehlertoleranten Varianten der Netze ist deshalb ein Signal herausgeführt, das den Blick auf die Aktivität der FT-Mechanismen ermöglicht.

4.4 Netztopologien

In diesem Abschnitt werden die Netztopologien betrachtet, die zur Implementierung in die engere Auswahl fielen. Zunächst wird ein rein funktionales Netz, siehe Kapitel 4.4.1, beschrieben, das zum Debuggen der VHDL-Modelle diente und bei den Fehlertoleranzbetrachtungen als Referenzimplementierung zu den mit FT-Maßnahmen ausgestatteten Netzen dient.

Die dem funktionalen Netz folgenden Unterabschnitte behandeln Netze, die in der Lage sind, bestimmte transiente Fehler zu erkennen und durch Wiederholung der Operationen, siehe [10] unter dem Schlagwort „Time Redundancy“ ab Seite 43, zu korrigieren. Es wurden zwei Möglichkeiten dieser Verfahren implementiert: Macro-Rollback, siehe Kapitel 4.4.2, und Micro-Rollback, siehe Kapitel 4.4.3. Von der Implementierung eines Rollforward-Verfahrens wurde aus den in Kapitel 4.4.4 erwähnten Gründen abgesehen.

Eine im Sinne von [10] „passive hardware redundancy“ implementierte FT-Maßnahme, die sowohl transiente als auch permanente Fehler korrigieren kann, rundet diesen Abschnitt im Kapitel 4.4.5 mit dem TMR-Netz ab.

4.4.1 Funktionales Netz

In diesem Kapitel wird die einfachste Art und Weise, die Knoten verschiedener Iterationsstufen zu verbinden, dargestellt. Diese Netztopologie wurde hauptsächlich aus zwei Gründen implementiert:

Bei der Entwicklung der Knoten ermöglicht eine einfache Struktur der Verbindungstopologie zwischen den Knoten eine effiziente Möglichkeit, Entwurfsfehler in den Knoten aufzuspüren, da aufgrund der Einfachheit des Netzes, die Knoten mußten quasi nur miteinander verbunden werden, Fehler darin nahezu auszuschließen waren oder schnell aufgedeckt werden konnten.

Ein weiterer Grund für die Implementierung dieser Netztopologie ist, daß die Fehlertoleranzeigenschaften der verschiedenen Implementierungen, bitparallel, bitredundant und bitseriell, an sich hervortreten, was eine Bewertung der FT-Eigenschaften dieser Varianten untereinander erlaubt.

Durch Vergleich der FT-Charakteristiken der mit FT-Mechanismen versehenen Netze mit dem

für diese Netze ermittelten Referenzwert ist eine klare Aussage über die Effizienz einer FT-Maßnahme möglich. Ohne dieses Netz würde nicht klar hervorgehen, ob die Kompensierung eines Fehlers einer FT-Maßnahme zugeschrieben werden kann oder ob die Implementierung der Algorithmen oder die verwendeten Algorithmen selbst bereits fehlertolerantes Verhalten an den Tag legen.

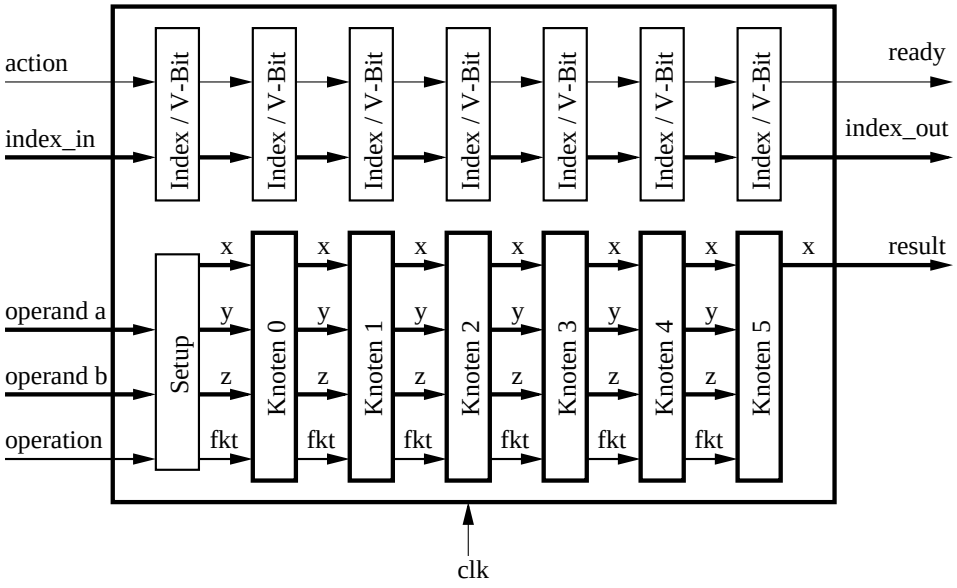

Abbildung 4.4: Blockschaltbild der funktionalen Netze

Die Struktur des funktionalen Netzes ist in Abbildung 4.4 dargestellt. Da das funktionale Netz zu jedem Takt bereit für neue Daten ist, wurde das Signal *busy*, da es niemals aktiv ist, weggelassen. Die stärker umrandeten Boxen, die mit Knoten 0 bis 5 durchnummeriert sind, kann man sich als Sockel vorstellen, in denen der jeweilige Knotentyp der erforderlichen Iterationsstufe steckt. Somit gibt es drei verschiedene funktionale Netze: Eines mit bitparallelen, eines mit bitredundanten und eines mit bitseriellen Knoten. Bei den funktionalen Netzen sind die Ausgänge des Knotens n mit den Eingängen des Knotens $n+1$ verbunden, um die Pipeline der Knoten herzustellen.

Im Setup-Block wird aufgrund der geforderten Operation *operation* entschieden, wie die Signale *x*, *y* und *z* für den Knoten 0 aus den Operanden *a* und *b* erzeugt werden. Wie auch in den Knoten werden in der Setup-Box die Eingaben gespeichert, was sie zum ersten Glied der Pipeline macht.

Für jede Stufe der Pipeline wird in der obenliegenden Box Index/V(alid)-Bit vermerkt, ob die gespeicherten Daten der einzelnen Stufen gültig sind (Valid-Bit) und welcher Index zu ihnen gehört. Da die Setup-Box die erste Stufe der Pipeline ist, ist auch für sie vermerkt, ob dort gespeicherte Daten gültig sind und welchen Index sie haben. Der Index und das Valid-Bit werden wie die Daten für die parallel verarbeitenden bitparallelen und bitredundanten Knoten zu jedem Takt, für die bitseriellen Knoten jeden 9. Takt eine Stufe weiter geschoben, wobei sich die neun Takte aus einem Takt für Übermittlung der geforderten Operation und acht weitere für die 8 Bit Operanden zusammensetzen.

Tabelle 4.2: Chipflächenbedarf der funktionalen Netze

Typ	Netz ohne Knoten	Knoten	Gesamt
Bitparallele Implementierung	608	$6 \cdot 1731$	10994
Bitredundante Implementierung	1104	$6 \cdot 2102$	13716
Bitserielle Implementierung	1265	$6 \cdot 752$	5777

Während für den bitparallelen und bitredundanten Ansatz der Index und V-Bit Mechanismus identisch ist und auch die Aufbereitung der Daten *x*, *y* und *z* aus *operation*, *a* und *b* aufgrund des parallel verarbeitenden Charakters ähnlich ist, wurde beim bitseriellen Ansatz in der Setup-Box ein Mechanismus integriert, der die parallel anfallenden Operanden nicht nur entsprechend umwandelt, sondern auch Bit für Bit mit dem LSB beginnend in den Knoten 0 schiebt. Da am Ende der bitseriellen Pipeline die Daten nicht parallel vorliegen, mußte der bitserielle Ansatz am Ende der Pipeline durch ein Schieberegister ergänzt werden, das die Ergebnis-Bits der Reihe nach einsammelt und wenn sie vollständig im Register enthalten sind, parallel ausgibt. Diese Einheit ist in Abbildung 4.4 nicht dargestellt, da sie nur für die bitserielle Variante des funktionalen Netzes, nicht aber für die anderen beiden implementiert werden muß.

Der Flächenbedarf dieser einfachen Netze ist in Tabelle 4.2 dargestellt. Der hohe Bedarf der bitseriellen Imlementierung ist auf die zusätzlichen Automaten und Schieberegister zurückzuführen, die für die Realisierung des allen Netzen gemeinsamen Netzwerkinterfaces benötigt werden.

Der Implementierungsaufwand, die Zeit, die für die Modellierung der Netze gebraucht wurde, hängt hauptsächlich von der Komplexität eventuell benötigter Steuerautomaten ab. Da im bitparallelen und bitredundanten Ansatz quasi keine Automaten zur Steuerung der Pipeline benötigt wurde, konnten diese etwa 4 bis 5 Mal schneller entwickelt werden, als das Netz für die bitserielle Variante, für die einerseits eine Umwandlung der parallelen in serielle Daten und am Ende der Pipeline wieder zurück integriert werden mußte und zusätzlich ein Steuerautomat entwickelt werden mußte, der die Steuerung des Netzwerkinterfaces übernimmt. Bei den beiden parallelen Ansätzen kam man ohne diesen Automaten aus, weil diese Netze zu jedem Takt neue Daten verarbeiten können, während der bitserielle Ansatz pro Takt nur ein Bit der Daten verarbeiten kann und man deshalb z.B. bei der Übernahme von Daten am Netzeingang zwischen zwei Operationen Wartezyklen einfügen muß, um das Herausschieben der Bits vom Pufferregister in die Knoten abzuwarten.

4.4.2 Macro-Rollback Netz

Dieses Unterkapitel enthält die Beschreibung eines Netzes, das mit einer Fehlertoleranz-Maßnahme ausgestattet ist, um einzeln auftretende, transiente Fehler korrigieren zu können. Es bedient sich der Methode, eine Operation solange zu wiederholen, bis sie korrekt durchgeführt wurde. Diese Methode wurde aus [10] in Kapitel „time redundancy“ abgeleitet. Im Gegensatz zur im Kapitel 4.4.3 beschriebenen Methode, die nach jeder Stufe der Pipeline die berechneten Daten überprüft, wird hier nur einmal am Ende der Pipeline eine Prüfung vorgenommen und

bei falschen Daten die Berechnung ab der 1. Stufe erneut angestoßen. Aus diesem Grund wurde dieses Netz ’Macro-Rollback Netz’ getauft, weil die gesamte Operation wiederholt wird.

Dieses Netz wurde implementiert, weil es das Rollback-Verfahren mit geringem Hardware-Aufwand integriert. Im Prinzip wird ein funktionales Netz verdoppelt und mit einem Komparator nach den Ergebnissen ergänzt, um festzustellen, ob beide Pipelines zum gleichen Ergebnis gekommen sind. Eine simple Logik entscheidet im Falle gleicher Ergebnisse, diese auszugeben und gleichzeitig am Eingang zu signalisieren, daß eine neue Berechnung angestoßen werden kann. Im Falle ungleicher Ergebnisse, wird der Eingang für die Übernahme neuer Operanden gesperrt und die Berechnung der als fehlerbehaftet durchgeführten Operation statt der neuen durchgeführt.

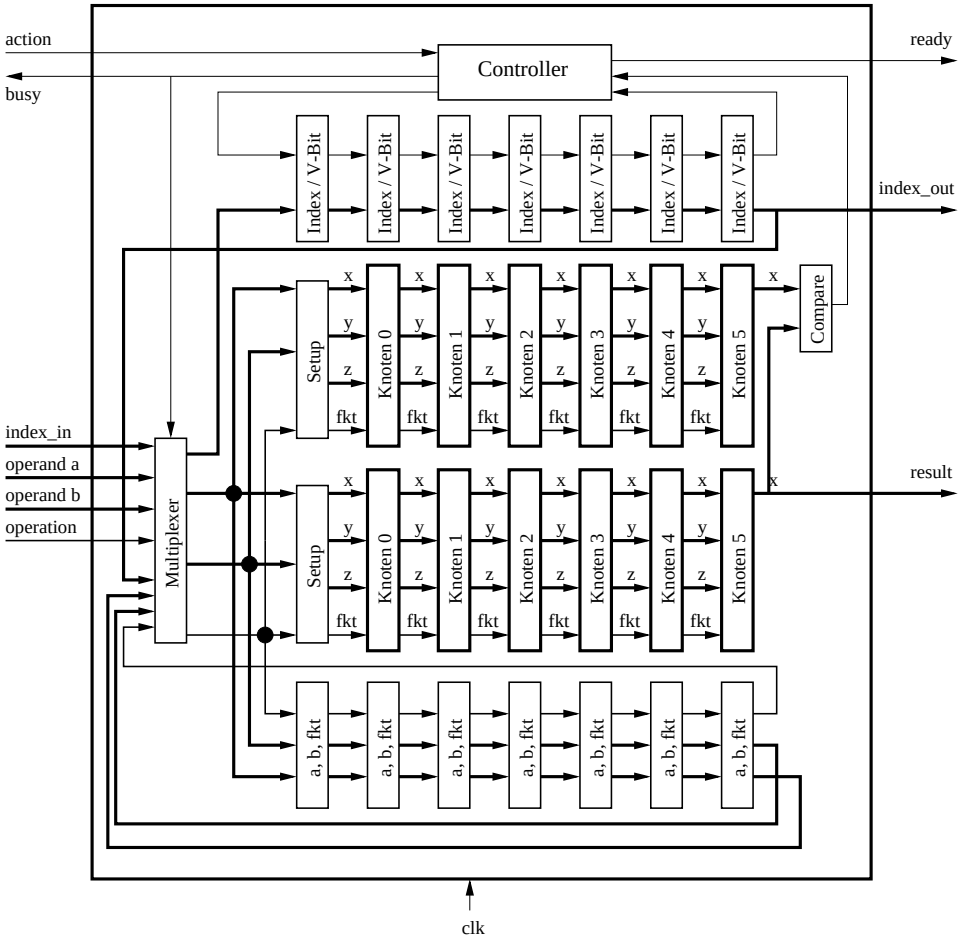

Abbildung 4.5: Blockschaltbild der Macro-Rollback Netze

Dieses Verfahren hat den Vorteil, daß für die einzelnen Stufen der Pipeline keine Steuermechanismen integriert werden müssen. Es kommt damit aus, jeweils am Eingang und am Ausgang zusätzliche Hardware zu ergänzen. In Abbildung 4.5 ist die Struktur des gesamten Netzes dargestellt. Vergleicht man es mit Abbildung 4.4 erkennt man die zusätzlichen Blöcke.

Im unteren Teil der Abbildung 4.5 ist zu sehen, daß die Pipeline der Knoten verdoppelt wurde. Dies ist erforderlich, um am Ende der Pipeline bestimmen zu können, ob ein berechnetes Ergeb-

Tabelle 4.3: Chipflächenbedarf der Macro-Rollback-Netze

Typ	Netz ohne Knoten	Knoten	Gesamt
Bitparallele Implementierung	1665	$12 \cdot 1731$	22437
Bitredundante Implementierung	2657	$12 \cdot 2102$	27881
Bitserielle Implementierung	3516	$12 \cdot 752$	12540

nis korrekt ist. Geht man davon aus, daß zu einem Zeitpunkt nur ein Fehler auftritt, kann man die Korrektheit des Ergebnisses bestimmen, indem man die Berechnung zweimal durchführt. Sind die Ergebnisse gleich, was im Block Compare ermittelt wird, signalisiert der Block Controller am Ausgang des Netzes die Gültigkeit der Daten.

Um im Fehlerfall eine Berechnung wiederholen zu können, ist es nötig, die Eingangsdaten mitzuschleifen, um sie für diesen Fall parat zu haben. Dafür existieren die ebenfalls in Form einer Pipeline angelegten Blöcke, die mit 'a, b, fkt' bezeichnet sind. Es handelt sich bei dieser Reihe von Blöcken um ein Schieberegister, wie es auch für die Index/V-Bit Blöcke verwendet wird.

Am Eingang wählt ein Multiplexer aus, ob neue Daten vom Eingang des Netzes oder alte Daten, die aufgrund eines detektierten Fehlers nochmal berechnet werden müssen, für die nächste Berechnung geholt werden. Ist das Signal *busy* aktiv, werden die alten Daten erneut der Berechnung zugeführt, ist es nicht aktiv, können neue vom Eingang des Netzes geladen werden.

Im Block Controller wird aufgrund der Gültigkeit der Daten in der letzten Stufe der Pipeline, dem V-Bit, und dem Ergebnis des Vergleichs beider Ergebnisse entschieden, was am Eingang des Netzes am Signal *busy* und am Ausgang des Netzes am Signal *ready* signalisiert wird. Sind die Ergebnisse der Berechnungseinheiten identisch, ist *ready* aktiv, um die Gültigkeit der Daten anzuzeigen, und *busy* nicht aktiv, um zu zeigen, daß das Netz bereit ist, neue Daten zu verarbeiten. Fällt der Vergleich negativ aus, wird durch ein nicht aktiv am Signal *ready* signalisiert, daß die Daten am Ausgang des Netzes ungültig sind, während am Signal *busy* durch Aktivität am Eingang des Netzes signalisiert wird, daß im Moment keine neuen Daten verarbeitet werden können, weil die Berechnungseinheiten zur Rekalkulation alter Daten gebraucht werden. Der Block Controller ist ein reines Schaltnetz und enthält keine Automaten.

Der Flächenbedarf für diese Netze ist in Tabelle 4.3 dargestellt. Wie auch beim funktionalen Netz fällt der hohe Anteil des Netzes für den bitseriellen Ansatz auf, der wieder auf die zusätzlichen Automaten und Schieberegister für die Realisierung der allen Knoten gemeinsamen Netzwerkinterfaces zurückzuführen ist.

Wie schon am Blockschaltbild in Abbildung 4.5 für diese Netztopologie zu erkennen ist, genügte die Duplizierung der Pipeline ergänzt mit wenigen Blöcken und einem einfachen Steuermechanismus, um die Fehlertoleranzmaßnahme Macro-Rollback in Hardware zu implementieren. Dadurch lag die Zeit zum Entwurf und Test dieses Netztyps bei knapp drei Mann-Tagen. Allerdings sind in dieser Zeit bereits mit einem Mann-Tag die Überlegungen enthalten, auch das Verhalten des Netzes im Fehlerfall zu untersuchen. Da bei konventioneller Nutzung des VHDL-Simulators keine Fehler im Netz auftreten, würde bei der Validierung des Netzes nie-

mals der fehlertolerante Mechanismus zum Tragen kommen. Um nun auch die fehlertoleranten Mechanismen zu aktivieren, wurden während der Simulation Signalwerte so verändert, daß ein Fehler provoziert wurde, der wiederum die FT-Maßnahme aktiviert hat.

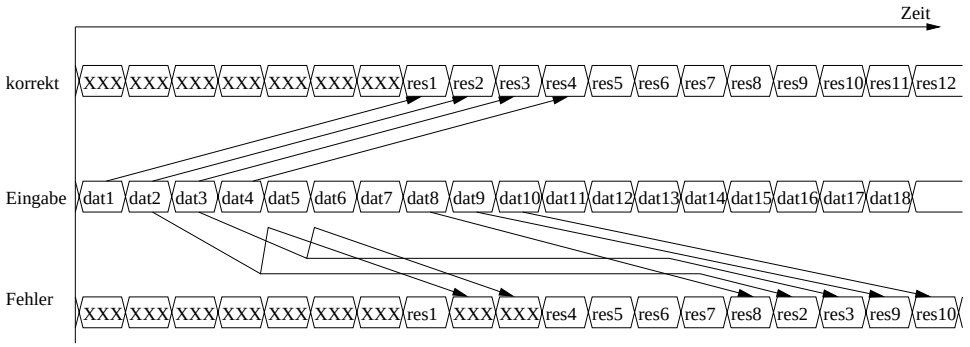

Abbildung 4.6: Timing Diagramm der Macro-Rollback Netze

Das zeitliche Verhalten der Macro-Rollback Topologie im Fehlerfall hat nur Auswirkungen auf die Latenz von Eingabezeitpunkt zur Ausgabe des Ergebnisses. Dauert die Berechnung einer Funktion im fehlerfreien Fall eine Zeit t_{norm}, wird sie im fehlerbehafteten Fall zusätzlich die Dauer $t_{duration}$ des Fehlers zuzüglich eines weiteren Durchlaufs t_{norm} benötigen, während alle folgenden Berechnungen um t_{norm} verspätet sind. In Abbildung 4.6 ist das Verhalten grafisch dargestellt: Nach rechts ist die Zeit aufgetragen. In der Mitte ist das Eingabesignal dargestellt. Es soll für jeden Taktzyklus eine Operation mit den Daten dat_i angestoßen werden. Die mit korrekt bezeichnete Ergebnisausgabe zeigt, daß mit einer Latenz von 7 Stufen die Ergebnisse res_i in der gleichen Reihenfolge vom Netz ausgegeben werden.

Im Fehlerfall, im Diagramm mit Fehler gekennzeichnet, tritt während der Berechnung ein Fehler auf, der die Berechnung des Datensatzes dat_2 und dat_3 stört. Im Diagramm ist deshalb zu erkennen, daß die Ergebnisse res_2 und res_3 im Ausgabestrom fehlen. Für die zugehörigen Daten dat_2 und dat_3 wird eine erneute Berechnung durchgeführt, was die Eingabe von dat_9 und dat_{10} verzögert.

4.4.3 Micro-Rollback Netz

In diesem Kapitel wird eine Netztopologie beschrieben, die analog zur Topologie, die in Kapitel 4.4.2 dargestellt wurde, transiente Fehler tolerieren kann, indem eine Berechnung solange wiederholt wird, bis sie korrekt durchgeführt wurde. Im Gegensatz zum Macro-Rollback, bei welchem nur am Ende der Pipeline eine Überprüfung der Ergebnisse stattfindet, wird beim Micro-Rollback nach jeder Stufe der Pipeline dieser Test durchgeführt. Der Name Micro-Rollback wurde gewählt, weil das Rollback auf einer Stufe feinerer Granularität erfolgt, als dies beim Macro-Rollback der Fall ist. Diese Methode wurde genauso wie die für das Macro-Rollback aus [10] abgeleitet.

Dieses Netz wurde implementiert, um die langen Latenzen für die Berechnung der Operationen, in denen ein Fehler auftritt, zu verringern. Dies geschieht, relativ zum Macro-Rollback gesehen,

auf Kosten der Operationen, die direkt nach fehlerbehafteten Operationen durchgeführt werden, da bei Fehlerdetektion die Pipeline genau an der Stelle angehalten wird, an der sich ein Fehler ausgewirkt hat. Die Macro-Rollback-Topologie hatte den Vorteil, daß die sich bereits in der Pipeline befindenden Operationen noch ohne zusätzliche Latenz ausgegeben wurden.

Hintergrund für die Entscheidung, die Latenzen möglichst gering zu halten, ist ein Echtzeitsystem, bei dem Daten per Spezifikation nur eine maximale zusätzliche Latenz aufweisen dürfen. Bedenkt man, daß bei diesem Beispiel nur mit 8 Bit Genauigkeit gerechnet wird, was eine Pipeline von 6 Stufen zur Folge hat, ist diese Überlegung nicht unbedingt relevant, da sich ein Ergebnis nur um 9 statt um 2 Zyklen verspätet. Betrachtet man jedoch ein System, das z.B. mit 24 Bit Genauigkeit rechnet, also 22 Iterationsstufen besitzt, wächst die Latenz bereits auf 25 statt 2 Zyklen an, was dann je nach Spezifikation die Micro-Rollback-Topologie erfordert, damit nicht eine über 10-fache Latenz auftritt.

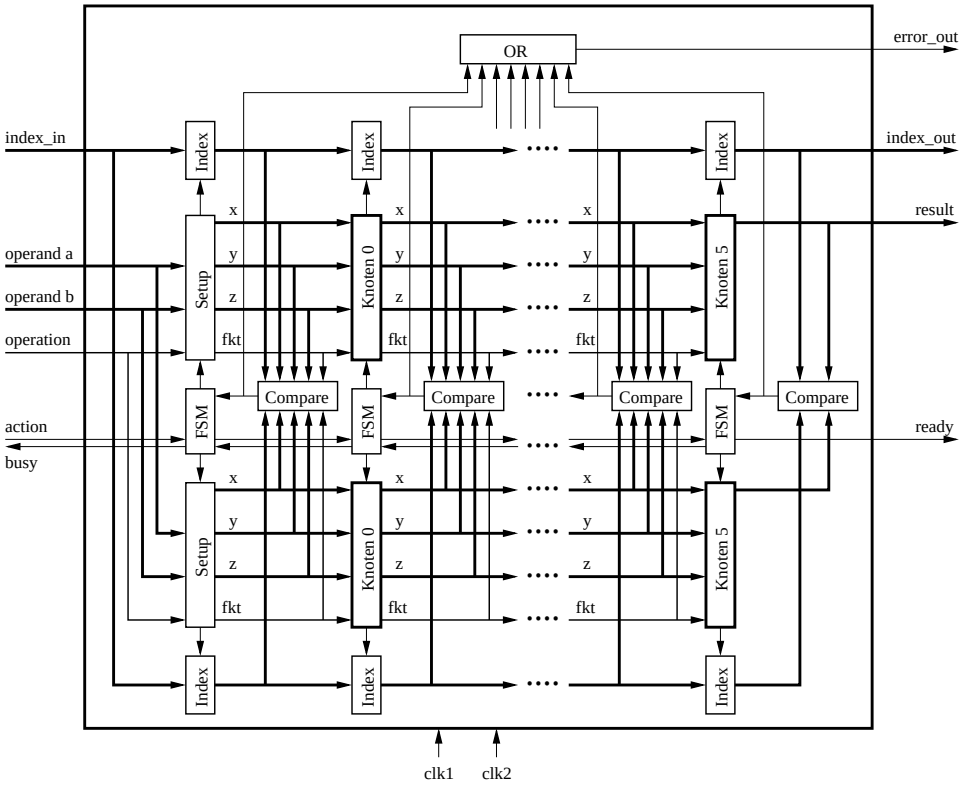

Abbildung 4.7: Blockschaltbild der Micro-Rollback Netze

Das Micro-Rollback wird realisiert, indem die Knoten, wie auch schon beim Macro-Rollback, verdoppelt werden und zusätzlich für jede Stufe der Pipeline ein Komparator, Block *compare*, zum Vergleich der in jeder Stufe berechneteten Operanden und ein Steuerautomat, Block *FSM*, zugefügt werden, siehe Abbildung 4.7. Im Gegensatz zum Macro-Rollback wurde hier beim Micro-Rollback die Übernahme der Operanden für jede Stufe der Pipeline realisiert, indem der Steuerautomat das Taktsignal für die Flipflops dieser Stufe generiert, weshalb dieses Netz mit zwei verschränkten Takten *clk*1 und *clk*2 versorgt werden muß. Das Signal *clk*1 wird für die Zustandsübergänge der Steuerautomaten benutzt, während *clk*2, evtl. maskiert durch ein Signal des Steuerautomaten, für die Übernahme der Daten in eine Stufe der Pipeline zuständig ist.

Diese Methode ist aus den Checkpoint-Verfahren entstanden. Bei diesen Verfahren werden Checkpoints festgelegt, zu denen berechnete Daten gesichert werden. Im Falle eines auftretenden, detektierten Fehlers werden die berechneten Daten verworfen. Stattdessen wird auf die Daten des letzten Checkpoints zurückgegriffen, um die Operation zu wiederholen. Für das Micro-Rollback wird im Prinzip ein Checkpoint nur mit neuen Daten geladen, wenn sein Nachfolger die alten Daten bereits in einem korrekten Checkpoint gespeichert hat. Ist dies nicht der Fall, wird immer wieder vom Checkpoint aus weitergerechnet, bis eine korrekte Berechnung durchgeführt wurde. Siehe dazu [10] Seite 160, „Rollback Recovery using Checkpoints“.

Wie im Vergleich der Struktur des Micro-Rollback-Netzes, Abbildung 4.7, zur Struktur des funktionalen Netzes, Abbildung 4.4, zu sehen ist, fehlen die Blöcke, die das Valid-Bit enthalten und die zeigen, ob die Operanden einer Stufe zu einer aktiven Berechnung gehören. Diese Information wurde in die Steuerautomaten verschoben.

Es sind mehrere Algorithmen für die Steuerautomaten der Pipeline-Stufen denkbar. Ein einfacher Ansatz wäre, das Ergebnis des Vergleichs am Ausgang der Stufen als Indikator zu verwenden, so daß die folgende Stufe die Daten nur bei identischen Signalen am Komparator übernimmt. Dies hat jedoch den gravierenden Nachteil, daß z.B. im Falle eines Stuck-At-Fehlers am Eingang eines Flipflops einer Pipelinestufe unter Umständen am Ausgang Daten generiert werden, die nicht mit der Referenzstufe übereinstimmen und somit zu einem ständigen Anzeigen eines Fehlers für die nächste Stufe führen, woraus folgt, daß das Netz blockiert und völlig funktionsuntüchtig wird.

Um dieses Phänomen auszuschließen, wurde ein Algorithmus für den Steuerautomaten entwickelt, der die Möglichkeit bietet, der vorangehenden Stufe zu signalisieren, daß sie keine neuen Daten aufnehmen soll, um in der aktuellen Stufe in zwei Phasen korrekte Daten in die Flipflops der Pipeline-Stufe zu laden. In der ersten Phase werden die Daten in die Flipflops geladen und gleichzeitig der vorangehenden Stufe signalisiert, die Daten solange zu halten, bis sichergestellt ist, daß identische Daten in die Flipflops der Stufe geladen wurden. Im fehlerfreien Fall führt dieses Verfahren jedoch zu einer Halbierung des Rechendurchsatzes, da für eine Berechnung nun zwei Takte statt einem benötigt werden. Im ersten Zyklus wird validiert, ob die Daten korrekt geladen wurden. Im zweiten Zyklus wird der folgenden Stufe wiederum gezeigt, daß nun die gelieferten Daten validiert werden können.

Für diesen Algorithmus ist in Abbildung 4.8 der Steuerautomat visualisiert. Er hat drei Eingangssignale, $action_in$ zur Signalisierung zu verarbeitender Daten am Eingang, $busy_in$ zur Signalisierung zum Halten der Ausgabe und *comp*, dem Ergebnis des Vergleichs am Ausgang der Stufe, zuzüglich des Signals *reset* zur Initialisierung des Automaten. Der Automat hat drei Ausgabesignale: $busy_out$ zum Signalisieren, daß die vorangehende Stufe die Daten einen weiteren Zyklus halten soll, $action_out$ zum Signalisieren, daß die folgende Stufe neue Daten am Eingang verarbeiten soll, und $fetch$ zur Steuerung des Ladevorgangs der Flipflops der aktuellen Stufe. Während das Signal $fetch$ ausschließlich vom Zustand des Automaten abhängt, hängen die Signale $action_out$ und $busy_out$ zusätzlich von den Eingangssignalen $action_in$ und $busy_in$ ab. Im Zustandsdiagramm in Abbildung 4.8 wird dieses Moore-/Mealy- Verhalten symbolisiert, indem das Signal $fetch$ direkt im Zustandsknoten dargestellt wird und die Signale $action_out$ und $busy_out$ an den Kanten.

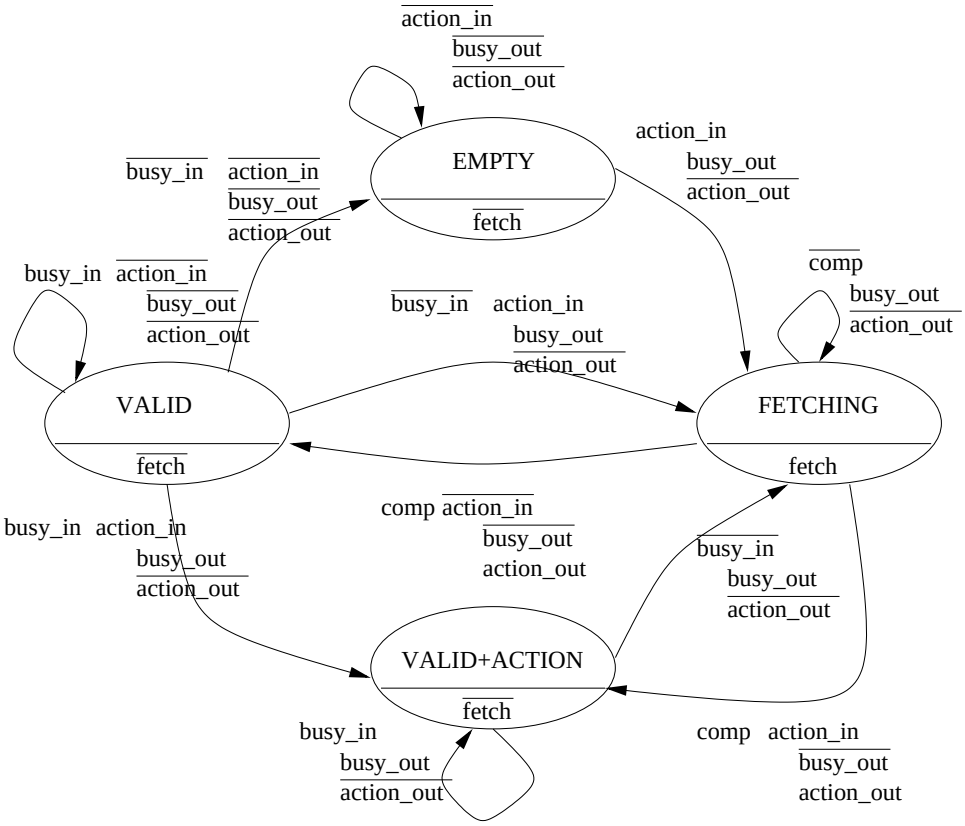

Abbildung 4.8: Zustandsdiagramm des Steuerautomaten

Bei dem in Abbildung 4.8 dargestellten Automaten ist zu beachten, daß je nach Anzahl der Stufen einer Pipeline aufgrund des Mealy-Charakters des Automaten sich Laufzeitprobleme ergeben können. Im Falle einer zu 100% gefüllten Pipeline löst bei einem Fehler in der letzten Stufe der Pipeline der zugehörige Steuerautomat dieser Stufe das Stoppen aller Stufen aus, wobei das entsprechende Signal durch die Steuerautomaten aller Stufen läuft. Das kann vermieden werden, indem die einzelnen Steuerautomaten bei der Synthese zu einem Block zusammengefaßt werden, damit das Synthesetool die Möglichkeit hat, dies zu optimieren.

Die Micro-Rollback-Topologie wurde nur in Ansätzen mit den bitseriellen Knoten durchgeführt, da aufgrund der komplexen Steuerung der Eingaben an jedem Knoten in Kombination mit dem seriellen Datentransfer ein Hardware-Aufwand erforderlich gewesen wäre, der vergleichbar mit der in Kapitel 4.4.5 vorgestellten Topologie gewesen wäre. Zusätzlich hätten sich die Latenzen verdoppelt, was für die auf kleine Latenzen ausgelegte Topologie als K.O.-Kriterium gewertet wurde und somit die Implementierung der Micro-Rollback-Topologie nur für die bitparallelen und bitredundanten Knoten aber nicht für bitserielle Knoten durchgeführt wurde.

Der Flächenbedarf der Micro-Rollback-Netze ist in Tabelle 4.4 dargestellt. Die bitserielle Variante wurde für diese Netz-Topologie nicht realisiert und fehlt deshalb in Tabelle 4.4.

Der Implementierungsaufwand für diese Topologie lag bei etwa 10 Mann-Tagen, wobei etwa 2 Mann-Tage dafür aufgebracht wurden, die Argumente gegen die Implementierung mit

Tabelle 4.4: Chipflächenbedarf der Micro-Rollback-Netze

Typ	Netz ohne Knoten	Knoten	Gesamt
Bitparallele Implementierung	1792	$12 \cdot 1731$	22564
Bitredundante Implementierung	3182	$12 \cdot 2102$	28406
Bitserielle Implementierung	-	-	-

den bitseriellen Knoten aufzudecken. Trotz der recht einfachen Struktur des Micro-Rollback-Ansatzes, wie er in Abbildung 4.7 zu betrachten ist, war viel Energie erforderlich, einen sinnvollen Algorithmus für die Steuerautomaten zu entwickeln.

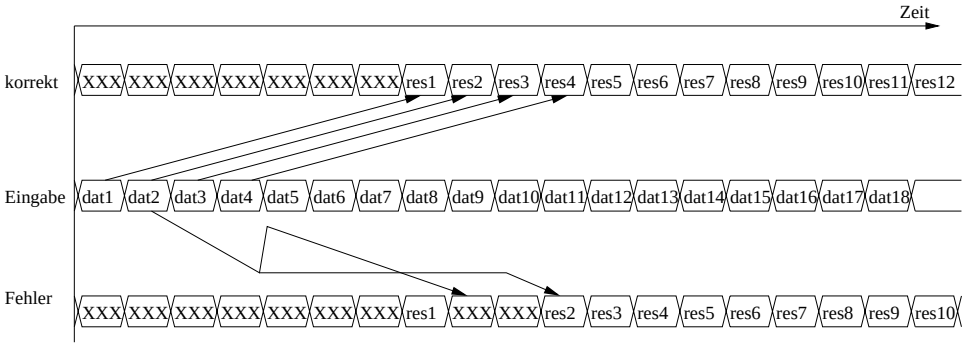

Abbildung 4.9: Timing Diagramm der Micro-Rollback Netze

Betrachtet man die Latenz von Operandeneingabe bis Ergebnisausgabe des Micro-Rollback-Netzes, erkennt man in Abbildung 4.9, daß sie im Fehlerfall alle Ergebnisse um die gleiche Anzahl von Zyklen verzögern und die Reihenfolge der Ergebnisausgabe erhalten bleibt.

4.4.4 Roll-Forward Netz

Das Roll-Forward-Verfahren ist wie auch das Roll-Backward in der Lage, einzelne, transiente Fehler zu tolerieren. Für diese Topologie werden die Funktionsblöcke ebenfalls dupliziert. Im Gegensatz zum Roll-Backward wird bei dieser Topologie jedoch bei der Erkennung eines Fehlers durch den Vergleich sowohl mit den falschen als auch mit den korrekten Werten solange weitergerechnet, bis der Fehler lokalisiert werden konnte, worauf der Datenstrang, der mit den fehlerhaften Werten gerechnet hat, für ungültig erklärt wird. Dadurch wird eine zusätzliche Latenz im Fehlerfall vermieden, wenn der Fehler nicht in der letzten Stufe der Pipeline auftritt. Diese Methode wurde aus [10] Seite 181 ff., Kapitel „Forward Recovery Schemes“ abgeleitet.

Bei der Implementierung dieses Verfahrens in Hardware hat sich jedoch schon bei den ersten Entwurfsschritten herausgestellt, daß eine Verdoppelung der Hardware nicht ausreicht. Zusätzlich zur Verdoppelung ist noch Hardware für die Lokalisation des Fehlers erforderlich, da es bei diesem Verfahren nicht genügt nur zu ermitteln, daß ein Fehler aufgetreten ist. Es wurden verschiedene Codierungsverfahren, wie z.B. Residuen- und Hammingcode, untersucht, um Fehler an den Knotenausgängen zu erkennen. Der Hardware-Aufwand für die Fehlerdetektion lag je-

doch mindestens beim 1.5-fachen, was in Anbetracht der Verdoppelung der Pipelines zu einer Verdreifachung der Hardware führen würde, was wiederum im Hinblick auf ein TMR-System, siehe Kapitel 4.4.5, als K.O.-Kriterium für die Implementierung dieses Verfahrens gewertet wurde.

Trotz der Entscheidung, dieses Verfahren nicht zu implementieren, wird im folgenden kurz theoretisch dargestellt, was den Ingenieur bei der Implementierung dieses Verfahrens auf Hardware-Ebene erwartet hätte und unter welchen Umständen dieses Verfahren doch sinnvoll auf Hardware-Ebene einsetzbar sein könnte.

Knackpunkt bei der Implentierung dieses Verfahrens ist die Effizienz, mit der Fehler in einem Knoten entdeckt werden können. Die Knoten selbst führen einfache Additionen oder Subtraktionen durch. Wenn es eine Kodierung der Operanden gibt, die im Hinblick auf die Abbildung in Hardware weniger als 50% Overhead erzeugen würde, wäre die Roll-Forward-Methode günstiger als TMR. Es wurden jedoch sowohl in der Literatur als auch durch eigene Ansätze keine Verfahren gefunden, die dies für den Fall der Bit/CORDIC- Algorithmen ermöglichen. Andererseits wäre der Einsatz anderer als der Bit/CORDIC-Algorithmen denkbar, die ein günstigeres Verhalten bezüglich der Fehlerlokalisation zeigen. Davon wurde in dieser Arbeit, die die Integration der Bit/CORDIC-Algorithmen genauer untersuchen soll, jedoch abgesehen.

Ein Problem bei der Implementierung des Roll-Forward-Verfahrens ist die Handhabung eines Fehlers in der letzten Stufe der Pipeline. Tritt dort ein Fehler auf, kann dieser nicht durch eine folgende Berechnung behoben werden. Dies könnte kompensiert werden, indem man nach der letzten Stufe noch eine nicht dem Roll-Forward unterliegende Stufe ergänzen würde, die am Ausgang des Netzes dann eine fortlaufende Folge von Ergebnissen liefert. Eine andere Lösung für das Problem eines Fehlers in der letzten Stufe der Pipeline wäre, in dieser Stufe ein Roll-Back zu integrieren, was aber den gravierenden Nachteil hätte, daß im Fehlerfall die Pipeline komplett gestoppt werden müßte und somit der Vorteil des konstanten Eingabe-Datenstroms dieser Topologie verloren ginge.

4.4.5 TMR-Netz

In diesem Kapitel wird beschrieben, wie sich das TMR-Verfahren in der Implementierung verhält. Im Gegensatz zu den Roll-Back-Verfahren ist es in der Lage, auch permanente Fehler zu kompensieren. Dabei weist es keine zusätzlichen Latenzen im Fehlerfall auf. Diese erhöhte Fehlertoleranz gegenüber den anderen, hier behandelten Netzen hat ihren Preis, den man schon in der Struktur der Netze erkennt: Anstatt einer Verdoppelung der Knotenanzahl erfordert TMR eine Verdreifachung. Dieses Verfahren wird in [10] im Kapitel „Hardware Redundancy“ auf Seit 8 ff. unter dem Punkt „passive hardware redundancy“ beschrieben.

Dieses Netzwerk wurde zur Realisierung in Hardware als Ersatz für das Roll-Forward-Netz ausgesucht, weil es bei höchstens gleichem Hardware-Overhead die Vorzüge des Roll-Forward (fortwährender Datenfluß und keine Latenzen im Fehlerfall) besitzt und auch permanente Fehler toleriert.

Der strukturelle Aufbau des TMR-Netzes ist in Abbildung 4.10 dargestellt. Wie gegenüber dem

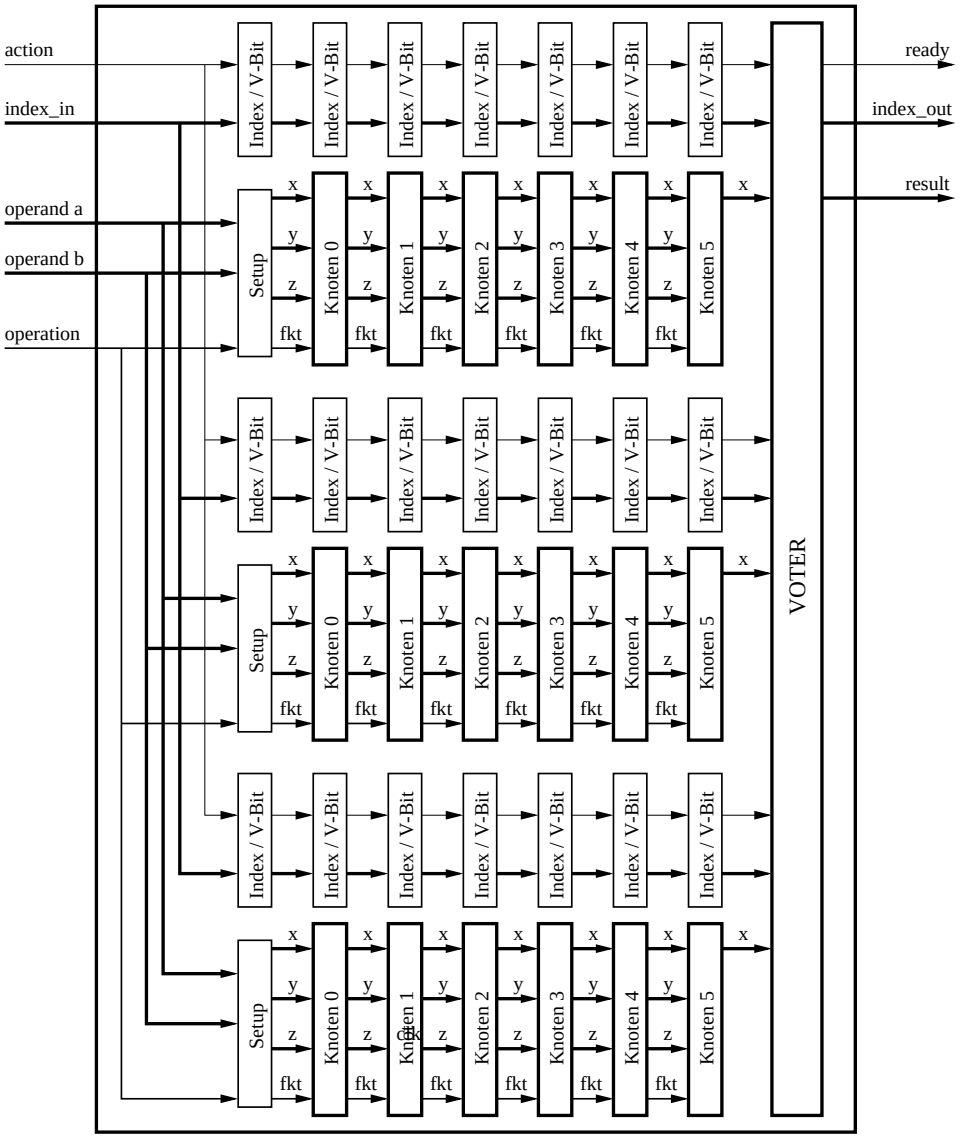

Abbildung 4.10: Blockschaltbild der TMR Netze

Tabelle 4.5: Chipflächenbedarf der TMR-Netze

Typ	Netz ohne Knoten	Knoten	Gesamt
Bitparallele Implementierung	2325	$18 \cdot 1731$	33483
Bitredundante Implementierung	3813	$18 \cdot 2102$	41649
Bitserielle Implementierung	4050	$18 \cdot 752$	17586

rein funktionalen Netz in Abbildung 4.4 zu sehen ist, besteht es aus drei parallel arbeitenden funktionalen Netzen, die am Ausgang durch einen Voter ergänzt werden, der aufgrund der drei Antworten entscheidet, was letztlich ausgegeben werden soll.

Der Flächenbedarf für die Implementierungsvarianten dieser Netz-Topologie ist in Tabelle 4.5

aufgetragen. Wie bei den anderen Netzen auch fällt die bitserielle Variante durch ihren hohen Bedarf auf, der auf die zusätzlichen Automaten und Schieberegister zur Realisierung des allen Netzen gemeinsamen Interfaces zurückzuführen ist.

Im Gegensatz zu den Roll-Back-Netzen ist für das TMR-Netz der Entwurf eines Automaten zur Steuerung nicht erforderlich. Einzige Neuheit gegenüber dem funktionalen Netz ist der Voter, der eine einfache zwei aus drei Auswahl durchführt. Die Entwicklungsarbeit für das TMR-Netz schlägt dementsprechend mit weniger als einem viertel Mann-Tag zu Buche.

Das TMR-Netz verhält sich im fehlerfreien Fall wie das funktionale Netz: Nach einer konstanten Latenz nach der Eingabe werden die Ergebnisse ausgegeben. Im fehlerbehafteten Fall glänzt es mit den gleichen Eigenschaften, nur daß die Ausgaben wesentlich häufiger korrekt sind, als dies beim funktionalen Netz der Fall ist.

4.5 Vergleich der Netztopologien

In diesem Kapitel werden die verschiedenen Netztopologien noch einmal gegenübergestellt und direkt miteinander in Bezug auf Flächenbedarf, Implementierungsaufwand und Verhalten im fehlerlosen und -behafteten Fall verglichen.

Für die späteren Produktionskosten eines Schaltkreises ist der Implementierungsaufwand nur dann entscheidend, wenn nur wenige dieser Bauteile hergestellt werden sollen. Die Entwicklungskosten, in Mann-Tagen gemessen und in Tabelle 4.6, können als Anhaltspunkt für ein Kriterium zur Entscheidung für oder gegen eine Maßnahme dienen. Allerdings sind diese Daten nur mit Vorsicht auf Realisierungen, denen andere Strukturen als die hier zeitlich streng linear arbeitende Pipeline zugrunde liegen, zu übertragen, weil sich bei anderen Strukturen durchaus Nebeneffekte, die hier nicht absehbar sind, bemerkbar machen könnten. Außerdem ist zu beachten, daß die Realisierung der bitseriellen Ansätze in den Roll-Back/-Forward-Verfahren einen nicht-linearen und somit schlecht extrapolierbaren Anteil einnahmen. Dies ist auf die Steuerautomaten der Knoten, die das Shiften der Signale überwachen, zurückzuführen, weil die Steuerautomaten der Netzwerke auf die Steuerautomaten der Knoten abgestimmt werden mußten. Ausgehend von einem rein funktionalen Netz lag das TMR-Netz mit einem Entwicklungsaufwand von $1/4$ Mann-Tag an der Spitze. Das Macro-Roll-Back folgt mit 2 Mann-Tagen, während das Micro-Roll-Back-Netz mit einer fünffachen Entwicklungszeit von 10 Mann-Tagen auftritt. Die Entwicklungszeit für das Roll-Forward-Verfahren kann nur geschätzt werden und liegt selbst unter der Voraussetzung, daß entsprechende Verfahren zur Fehlerlokalisation bereits vorhanden sind, vermutlich weit jenseits von 10 Mann-Tagen.

In Tabelle 4.7 ist der Flächenbedarf jeder Lösung dargestellt. Wie zu erkennen ist, macht sich bei den Rollback-Topologien der zusätzliche Aufwand über die Duplizierung der Knoten hinaus kaum bemerkbar, so daß man auch gesamt gesehen mit einer Verdoppelung des Hardware-Aufwandes eine Hardware bekommt, die in der Lage ist, transiente Fehler zu kompensieren. Auch bei den TMR-Topologien spricht die Tabelle 4.7 für einen Hardware-Overhead, der über die Verdreifachung kaum hinaus geht, so daß man für das in dieser Arbeit dargestellte Problem die Aussage, daß ein TMR-System nur eine Verdreifachung der Hardware zur Folge hat, vertre-

Tabelle 4.6: Vergleich der Entwicklungskosten

Topologie	Kosten
macro-rollback	2
micro-rollback	10
roll-forward	$\gg 10$
tmr	$1/4$

Tabelle 4.7: Vergleich des Flächenbedarfs

Topologie	bitparallel	bitredundant	bitseriell
funktional	10994	13716	5777
macro-rollback	22437	27881	12540
micro-rollback	22564	28406	-
tmr	33483	41649	17586

ten kann. Eventueller Ausreißer dieser beiden Folgerungen ist die bitserielle Realisierung der Micro-Rollback-Topologie, die bereits im Vorfeld aufgrund des erwarteten, hohen Chipflächenbedarfs nicht implementiert wurde. Das Rollforward-Verfahren, welches hier ebenfalls nicht dargestellt wurde, ist ein weiterer Kandidat, der mindestens den Overhead eines TMR-Systems besitzt, aber nur transiente Fehler korrigieren kann und ebenfalls im Vorfeld ausgesiebt wurde.

Das Zeitverhalten der verschiedenen Topologien ist im fehlerfreien Fall zwischen allen Netzen bis auf das Micro-Roll-Back-Netz, das mit halber Taktfrequenz arbeitet, identisch. Während die Roll-Backward-Netze nur transiente Fehler durch Wiederholung einer Operation korrigieren können, ist das TMR-Netz in der Lage, auch permanente Fehler zu korrigieren. Im Fehlerfall zeigt nur noch das TMR-Netz das zum funktionalen Netz identische, meist fehlerlose Verhalten. Sowohl Macro- als auch Micro-Roll-Back verzögern die Ausgabe der Ergebnisse nach einem Fehler, wobei beim Macro-Rollback-Netz die Latenz 9 Zyklen beträgt, während das Micro-Rollback-Netz nur 1 Zyklus nach verschwinden des Fehlers benötigt. Zusätzlich sei angemerkt, daß das Micro-Rollback-Netz die Latenz eines Zyklus nach verschwinden des Fehlers immer aufweist, während beim Macro-Rollback die Verzögerung linear zur Anzahl der Iterationsstufen steigt und deshalb für höhere Genauigkeiten größer wird.

Tabelle 4.8: Vergleich des Rechendurchsatzes

	R_T			R_P		
Topologie	par	red	ser	par	red	ser
funktional	3758.62	1903.59	1608.73	6063.91	3037.81	1545.54
macro-rollback	1841.70	936.47	741.12	2971.28	1494.45	712.01
micro-rollback	915.67	459.58	-	1477.28	733.41	-
tmr	1234.13	626.90	528.47	1991.06	1000.42	507.71

Ein weiteres wichtiges Kriterium zur Bewertung der Netze ist der erzielbare Rechendurchsatz, die Operationen pro Zeit und Chipfläche. Da alle Topologien außer der Micro-Rollback-Topologie das gleiche Zeitverhalten wie die funktionale Variante zeigen, wurden die Werte aus Tabelle 3.4 auf die neuen Chipflächen umskaliert. Die Umskalierung wurde durchgeführt, indem die Werte aus Tabelle 3.4 mit der Fläche, die die entsprechenden Knoten einnehmen, multipliziert wurden und dieser Wert durch den eigentlichen Flächenbedarf der Knoten zuzüglich des Netzes geteilt wurden. Im Falle der Micro-Rollback-Topologie wurde der Wert wegen der doppelt so langen Zyklendauer nochmals halbiert. Die resultierenden Werte sind in Tabelle 4.8 aufgelistet. Wie man sieht, verhält sich der Rechendurchsatz näherungsweise antiproportional zum Chipflächenbedarf, weshalb der Durchsatz für die Macro-Rollback Netze etwa halb so groß ist, wie für das funktionale Netz. Der Durchsatz der Micro-Rollback Netze ist wegen des halben Taktes nur ein viertel des Durchsatzes des funktionalen Netzes. Besonders sei hier auf den Durchsatz des Micro-Rollback verglichen mit dem TMR Netz hingewiesen: Obwohl die benötigte Chipfläche des Micro-Rollback Netzes kleiner ist, als die des TMR Netzes, ist aufgrund des halben Taktes beim Micro-Rollback Netz der Rechendurchsatz durchwegs kleiner als der des TMR Netzes.

In Tabelle 4.8 kann man auch nochmal erkennen, daß der Overhead für die bitseriellen Netze deutlich größer ist, als dies für die parallel verarbeitenden der Fall ist. Beträgt der Verlust des Rechendurchsatzes bei den parallelen Varianten etwa 10% im Vergleich zu Tabelle 3.4 ohne Netzinterface, ist der Unterschied für die bitseriellen Netze im Bereich um 20 %.

Die Ermittlung und der Vergleich der Fehlertoleranzeigenschaften der implementierten Systeme wird im folgenden Kapitel 5 dargestellt.

Kapitel 5

FT-Analyse

In diesem Kapitel wird beschrieben, welche Schritte unternommen wurden, um eine Bewertung des Verhaltens im Fehlerfall vornehmen zu können und vor allem einen Vergleich der Realisierungen durchführen zu können, der eine Auswahl der besten Lösung unter gewissen Randbedingungen ermöglicht.

Um die VHDL-Modelle bezüglich ihrer Fehlertoleranzeigenschaften zu bewerten, wurde der Weg der simulationsbasierten Fehlerinjektion eingeschlagen. Im Kapitel 5.1 wird dargestellt, wie das Simulationsszenario aussieht, um die Charakteristiken der Netz-Topologien bezüglich Fehlertoleranz zu ermitteln.

Aufgrund der im Simulationsszenario dargestellten Methode, werden in Abschnitt 5.2 die Daten zusammengetragen, die bei den Simulationen der Topologien ermittelt wurden. Zusätzlich wird für die Topologien überrissen, welche Fehler besondere Auswirkungen hatten.

Abschnitt 5.3 behandelt die Transformation der in 5.2 ermittelten Wahrscheinlichkeiten eines Fehlverhaltens für die injizierten Fehler in Ausfallraten der Systeme. Basierend auf den Ausfallraten werden in Abschnitt 5.4 die Implementierungen gegenübergestellt und allgemeine Schlußfolgerungen gezogen.

5.1 Evaluierungs-Szenario

Dieses Kapitel beschreibt das Szenario, unter welchem die Bewertung der Fehlertoleranzeigenschaften der verschiedenen Netz-Topologien durchgeführt wurde. Es ist in mehrere Unterkapitel aufgeteilt, um die verschiedenen Aspekte, die in Abbildung 5.1 erkennbar sind, übersichtlich darzustellen.

Die Eingaben in diesem Szenario sind eine Testbench, in die das System eingebettet ist und die in Kapitel 5.1.1 dargestellt wird, die abstrakten Systembeschreibungen für die Netz-Topologien und Knoten, die bereits in Kapitel 3 und 4 dargestellt wurden, und eine Gatterbibliothek, aufgrund derer das VHDL-Modell der abstrakten Systembeschreibungen auf ein VHDL-Modell auf Gatterebene abgebildet wurde. Die Gatter der Gatterbibliothek sind ebenfalls in einem er-

weiterten VHDL modelliert und enthalten das Fehlermodell. Darauf wird in Kapitel 5.1.2 eingegangen.

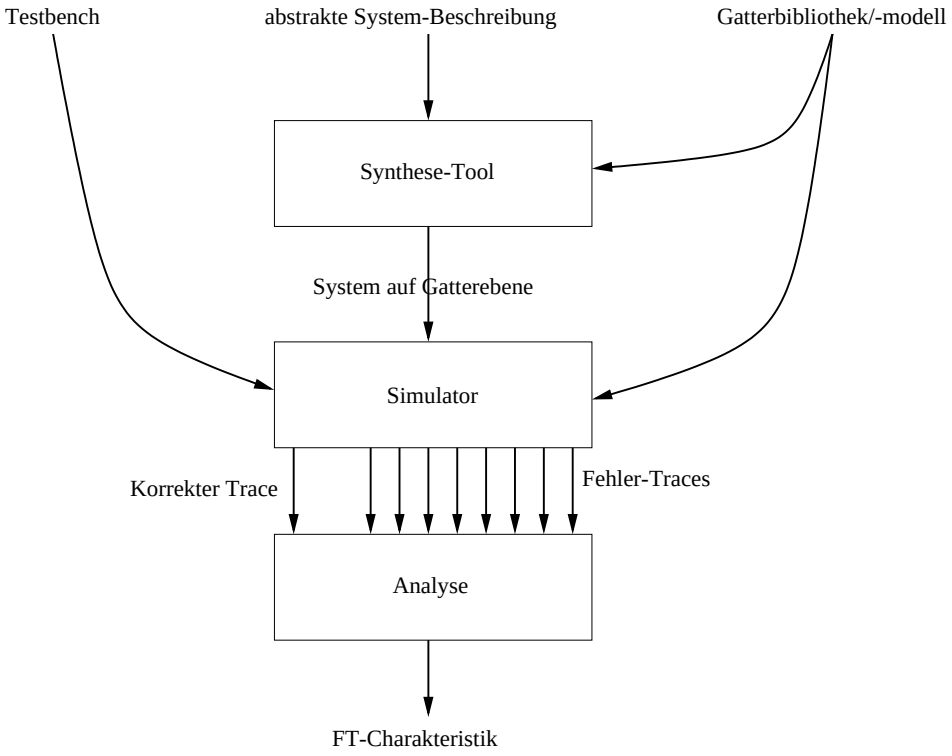

Abbildung 5.1: Blockschaltbild des Evaluierungsszenarios

In Abbildung 5.1 ist in den Blöcken dargestellt, welche Tools bei der Evaluierung verwendet werden. An den Kanten ist angeschrieben, welche Daten transportiert werden. Ausgehend von den abstrakten Systembeschreibungen der Netztopologien wird von einem Synthese-Tool, wie schon in den Kapiteln 3 und 4 beschrieben, jeweils eine Systembeschreibung generiert, die einer Transformation des Systems auf eine Gatterliste entspricht. Dieser Schritt wurde mit dem Design Analyzer von Synopsys durchgeführt und wird nicht weiter beschrieben, weil keine Probleme damit auftraten und es ein Standardschritt im Schaltkreisentwurf ist. Zur Gatterbibliothek, auf die das abstrakt beschriebene System abgebildet wurde, wird im Kapitel 5.1.2 im Zusammenhang mit dem Fehlermodell Stellung genommen.

Der Block Simulator besteht im wesentlichen aus einem Tool, das den VHDL-Quelltext simulieren kann. Da bei der Simulation Fehler in das System injiziert werden sollen, war es nötig, entsprechende Vorrichtungen zu schaffen. Dies, weitere Probleme und welche Ausgaben dieser Block liefert, werden in Kapitel 5.1.3 behandelt.

In Kapitel 5.1.4 werden zunächst verschiedene Arten von Fehlverhalten spezifiziert, aufgrund derer dann Methoden dargestellt werden, die bei einer Analyse der Traces eine Klassifizierung für jeden injizierten Fehler erlauben. Die Ausgabe dieses Blocks ist ein einfaches Abzählen, wie viele Fehler welches Fehlverhalten zur Folge hatten, was als FT-Charakteristik bezeichnet wird.

5.1.1 Testbench

Um ein datenverarbeitendes System bewerten zu können, sind Stimuli und deren bekannte Antworten erforderlich, aufgrund derer evaluiert werden kann, ob das System korrekt antwortet. In diesem Kapitel wird beschrieben, wie die Testbench, in die das System zur Evaluierung eingebettet ist, modelliert wurde und wie diese Stimuli gewählt wurden.

Die Struktur der Testbench wurde bereits in Abbildung 4.3 in Kapitel 4.2 mit einem eingebetteten Netz als DUT (device under test) dargestellt. Allerdings wurde in diesem Kapitel die Testbench dafür eingesetzt, die VHDL-Modelle auf korrekte Funktion zu validieren. In diesem Kapitel wird die gleiche Testbench angewendet, um damit während der Simulation mit Fehlerinjektion Daten zu ermitteln, die später Aussagen über die Charakteristik der Netz-Topologien bezüglich der Fehlertoleranz erlauben.

Die Charakteristik hängt von den angelegten Stimuli, oder äquivalent ausgedrückt, von der Workload, unter der das System arbeitet, ab, siehe z.B. [11] oder [3]. Man stelle sich zum Beispiel die hier beschriebenen Systeme unter einer Workload vor, die zu hohen Anteilen Sinus-Berechnungen betreibt, und ein System, daß gerade für Sinus-Berechnungen besonders häufig ausfällt. Die unter dieser Konfiguration ermittelten Charakteristika lassen sich nicht auf andere Workloads, Stimuli, übertragen, die beispielsweise bevorzugt Multiplikationen durchführen.

In dieser Arbeit wurden deshalb Stimuli gewählt, die mit einer zufälligen, gleichverteilten Abdeckung Fehlverhalten unter den verschiedensten Bedingungen aufdecken sollen. Einzige Einschränkung bei der Auswahl der 64 Testpatterns war ein gleichmäßiges, 8 maliges Anstoßen jeder der 8 implementierten Funktionen der in dieser Arbeit behandelten Einheiten. Die Operationen wurden mit Werten durchgeführt, die quasi gleich verteilt sind. Auf diese Weise wäre es möglich, im Nachhinein bei der Auswertung zu ermitteln, wie sich zum Beispiel Fehler auf die Sinusberechnungen ausgewirkt haben, indem man einfach nur die Stimuli und deren erwartete Antworten betrachtet, die eine Sinusoperation ausgeführt haben.

Ein Problem bei der Simulation dieser Systeme mit Fehlerinjektion ist die Datenflut, die in Kapitel 5.1.3 erläutert wird. Ursprünglich sollten die Ausgangssignale der Netze mitprotokolliert werden, was jedoch nicht nur zu Datenmengen im Terabyte-Bereich geführt hätte, sondern auch zu Problemen bei der Auswertung, die wenigstens proportional zur Datenmenge an Zeit benötigt hätte. Deshalb war es nötig, die bei der Simulation anfallenden Daten zu komprimieren, was mit einem direkten Vergleich der Ausgaben eines Netzes mit den erwarteten Antworten gelöst wurde: Ein Prozeß in der Testbench hat ständig den Ausgang des DUT abgefragt. Wenn das DUT Aktivität signalisierte, wurde nur der Zeitpunkt und die eventuelle Abweichung der Daten von den erwarteten Daten ausgegeben. Die erwarteten Antworten wurden mit einem C-Programm, das die Bit-/CORDIC-Algorithmen auf den Stimuli durchführt, berechnet.

Die Testbench, in VHDL modelliert, wurde ebenfalls durch ein C-Programm generiert, um Fehler bei der Übertragung der erwarteten Antworten vom C-Programm ins VHDL-Programm zu vermeiden. Die Testbench zusammen mit den Gattermodellen und dem eigentlichen System sind die Eingaben für den VHDL-Simulator.

5.1.2 Gatterbibliothek und Fehlermodell

Dieses Kapitel beschreibt die Methode, wie das System, das rein unter funktionalen Gesichtspunkten ohne Möglichkeiten zur Fehlerinjektion modelliert wurde, einem Verfahren zur Ermittlung von FT-Charakteristiken unterzogen werden kann. Zusätzlich wird das Fehlermodell beschrieben, aufgrund dessen Fehler injiziert wurden.

Zu einem Schritt beim Schaltkreisentwurf eines Standardzellen-Designs zählt die Abbildung eines abstrakten VHDL-Modells auf eine Standardzellenbibliothek, die beschreibt, welche Gatter verfügbar sind. In dieser Arbeit wurde die Bibliothek *lsi_10k*, die von Synopsys mitgeliefert wird, verwendet.

Um Aussagen über die Qualität der FT-Charakteristiken treffen zu können, ist es nötig, die Fehlerinjektion auf möglichst konkreter Ebene durchzuführen, siehe [13] und [14]. In dieser Arbeit wurde dies auf Gatterebene durchgeführt, was zusätzlich den Vorteil hat, daß die Beschreibung des eigentlichen Systems und dessen Abbildung auf das Gattermodell nicht durch zusätzliche Beschreibung von Saboteuren, siehe [8], [6], [2] oder [4], belastet werden muß, was den Entwicklungsprozeß der Systeme komplizierter und somit schwieriger und langwieriger gemacht hätte. Die Fehler werden über die Beschreibung der Gatter in das System eingebracht.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE work.faulty_stuck_at_01.ALL;
ENTITY IV IS
        GENERIC( delay : time := 1 ns );
        port( A : in std_logic;  Z : out std_logic);
END IV;

ARCHITECTURE behaviour OF IV IS
        SIGNAL s0_a, s1_a : boolean;
        SIGNAL s0_z, s1_z : boolean;
BEGIN
        iv_func : PROCESS( A, s0_a, s1_a, s0_z, s1_z ) BEGIN
                Z <= stuck_at_01_std_ulogic (
                        NOT ( stuck_at_01_std_ulogic(A, s0_a, s1_a) ),
                        s0_z, s1_z
                    ) AFTER delay;
        END PROCESS;
END behaviour;
```

Abbildung 5.2: VHDL-Modell eines Inverters mit Fehlerinjektionsmöglichkeit

Dafür wurde für jedes Gatter, das das Synthese-Tool bei der Abbildung auf die Zellbibliothek verwendet hat, ein VHDL-Modell erstellt, das zusätzlich die Möglichkeit bietet, über interne Signale im Gatter, Stack-At 0 und 1 Fehler an jedem Ein- und Ausgang aller Gatter zu verursachen. Bei Flipflops besteht zusätzlich die Möglichkeit, über ein internes Signal einen Bitflip-

Fehler auszulösen. Um einen Fehler zu injizieren, muß dann das entsprechende Fehlersignal im Gattermodell während der Simulation aktiviert werden. Um Häufigkeit und Dauer der Aktivierung bei der Simulation festlegen zu können, wurde der VHDL-Simulator des Tools VERIFY benutzt, der aufgrund eines erweiterten VHDL-Wortschatzes die Einführung stochastischer Signale, die hier zur Injektion transienter Fehler benutzt wurden, erlaubt. In Abbildung 5.2 ist der VHDL-Quelltext für einen Inverter dargestellt. Die vier Signale *s*0_*a*, *s*1_*a*, *s*0_*z* und *s*1_*z* werden vom VHDL-Simulator auf *false* initialisiert. Der Funktion *stuck_at_01_std_ulogic* wird als erstes Argument der normale Wert eines Signals *A* übergeben. Dieser wird zurückgegeben, wenn die beiden folgenden Boolean-Signale *s*0_*a* und *s*1_*a* den Wert *false* enthalten. Wird eines der beiden letzteren Signale *true*, liefert die Funktion entweder eine 0, wenn *s*0_*a true* ist, oder eine 1, wenn *s*1_*a true* ist. Das gleiche Prozedere wird für den Ausgang durchgeführt. Läßt man die Simulation laufen, ohne die Boolean-Signale jemals mit *true* zu belegen, entspricht dies dem fehlerfreien Lauf, was eine Verwendung üblicher VHDL-Compiler/-Simulatoren ermöglicht.

Mit den Steuersignalen zur Auslösung von Gatterfehlern und VERIFY als stochastischer Steuerautomat zum Aktivieren eines Fehlers in einem Gatter ist es nun möglich, damit die Klasse der Bitflip-Fehler und die Klasse transienter Stuck-At-Fehler abzudecken. Die Fehlerraten für beide Fehlerklassen wurden auf den gleichen Wert gesetzt, so daß die Wahrscheinlichkeit der Aktivierung für jeden Fehler gleich groß ist. Die Fehlerdauer bei den transienten Stuck-At-Fehlern ist exponential verteilt und beträgt im Mittel 100 ns, was einem Taktzyklus entspricht.

Das Fehlermodell, das hier gewählt wurde, basiert leider nicht auf praktischen Erfahrungen. Es wurde trotz allem gewählt, weil bisher keine in der Praxis fundierten Statistiken über Fehlerursachen zu finden waren. Dies liegt daran, daß im normalen Betrieb Hardwarefehler extrem selten sind. Selbst wenn sie auftreten und ein Fehlverhalten des Systems verursachen, wird das System eher durch einen Reset wieder in einen definierten Zustand versetzt, als der Ursache für das Fehlverhalten auf den Grund zu gehen. Bei der Komplexität moderner Systeme sprengt der Aufwand für ein solches Unternehmen vermutlich auch alle Dimensionen, wenn eine eindeutige Lokalisation der Fehlerquelle in Ort und Zeit überhaupt eindeutig bestimmbar ist.

Ein in der Praxis anerkanntes Fehlermodell ist das Stuck-At-Fehlermodell, das allerdings für Fehlerinjektionsexperimente mit anderem Hintergrund, nämlich dem Test integrierter Schaltungen, erfolgreich eingesetzt wird, um die Fault-Coverage der Testpatterns simulativ zu bestimmen. Hier werden Signale permanent von Beginn bis Ende der Simulation auf 0 (stuck at 0) oder 1 (stuck at 1) gelegt und anhand der Ausgaben des Systems ermittelt, ob eine Abweichung zu den erwarteten Antworten, die ohne injizierten Fehler ermittelt werden, auftritt. Mit diesem Modell im Kopf könnte man sich vorstellen, daß ein Stuck-At-Fehler nicht nur permanent durch einen Kurzschluß verursacht auftritt, sondern z.B. beim Einschlag elektrisch geladener Teilchen auch transient erscheinen könnte, wobei leider nichts über die Raten und die resultierenden Stuck-At-Folgen solcher Einschläge bekannt ist. Diese hängen vermutlich von der Rate der Teilchen und der Kapazität der Leitung ab.

Bei der in dieser Arbeit gewählten Methode wird die Fehlerrate für alle möglichen Fehler auf den gleichen Wert gesetzt, was für die Simulation die Folge hat, daß alle Fehler mit der gleichen Wahrscheinlichkeit aktiviert werden. Daher wird im folgenden darauf verzichtet, bei den ermittelten Ausfallraten für die Systeme ein konkretes Zeitmaß anzugeben. Da es aber für alle Modelle das gleiche ist, können die resultierenden Werte miteinander verglichen werden, um

bessere von schlechteren Systemen zu unterscheiden, ohne jedoch die absolute Aussage wie z.B. 1 Ausfall pro Jahr zu fällen.

Die mittlere Fehlerdauer von 100 ns wurde gewählt, weil die Flipflops des Systems mit einem 10 MHz Takt arbeiten und diese somit alle 100 ns neue Daten speichern. Auf diese Weise werden Fehler injiziert, die mit relativ hoher Wahrscheinlichkeit auch Auswirkungen zeigen. Würde man die Fehlerdauer auf 1 ns reduzieren, würden sich nur selten Fehler im System auswirken. Auch hier sei darauf hingewiesen, daß diese Art der Fehlerinjektion zwar üblich ist, daß aber die Übertragbarkeit der ermittelten Fehlverhalten auf reale Fehler nur auf wackligen Beinen steht, da geringe Änderungen am Fehlermodell gravierende Änderungen im Fehlverhalten verursachen können, siehe [13].

5.1.3 Simulation

Dieses Kapitel beschreibt die Methode, mit der die Daten für eine Bestimmung der Fehlertoleranz-Charakteristik eines Systems gewonnen werden, und zwei Probleme, die bei der Simulation der Systeme auftraten und wie sie gelöst wurden: Wie die langen Simulationszeiten verkürzt und wie die großen Datenmengen bewältigt wurden.

Um später Aussagen über die Zuverlässigkeit der Netz-Topologien treffen zu können, wurde jedes System einmal ohne injizierten Fehler, als Referenz zur Ermittlung, ob sich ein Fehler ausgewirkt hat, simuliert und die Ausgaben als korrekter Trace mitprotokolliert, was durch den linken, abgesetzten, einzelnen Pfeil in Abbildung 5.1 visualisiert werden soll. Darauf folgend wurden 10000 weitere Simulationen für jedes System durchgeführt, wobei jeweils ein Fehler dem Fehlermodell entsprechend zu einer bestimmten Zeit eine bestimmte Dauer aktiviert wurde und das Ergebnis dieser Simulationen ebenfalls in jeweils einem Fehler-Trace, Gruppe von Pfeilen rechts in Abbildung 5.1, mitprotokolliert wurde. Dies ergibt 10001 Traces pro Netz-Topologie. Die Erläuterung der Anzahl von 10000 Fehlern sowie die Frage, wie aus diesen Traces die FT-Charakteristik des untersuchten Systems bestimmt wurde, befindet sich in Kapitel 5.1.4.

Ursprünglich war geplant, das Tool VERIFY, welches bereits Simulation und Auswertung ermöglicht, für die Evaluierung der Netz-Topologien zu verwenden. Mit VERIFY ist es möglich, das zufällige Auftreten von Fehlern zu simulieren. Für die Dauer der Simulation und die Daten, die während einer Simulation anfallen, gilt, daß sie linear zur Anzahl der injizierten Fehler steigen.

Da VERIFY in den Traces jeden Signalwechsel speichert und aufgrund der Komplexität der Systeme sehr viele Signalwechsel stattfanden, stieg der Speicherbedarf für diese Methode jenseits der Tera-Byte Region. Dieses Problem wurde gelöst, indem die Trace-Daten modifiziert wurden: Es wurden nicht mehr die Signalwechsel protokolliert, sondern mittels der VHDL-Report-Anweisung Ausgaben erzeugt, die Aufschluß über die Reaktionen des Systems gaben. Dies wurde realisiert, indem in der Testbench ein Prozeß den Ausgang des Netzes beobachtet hat. Dieser Prozeß hat nur noch die Ergebnisse mit zugehörigem Index und Zeitpunkt, wann sie geliefert wurden, protokolliert. Mit dem Wechsel von Signalwechseln zu den berechneten Ergebnissen und dem zugehörigen Index und Zeit gelang es, die Datenmengen auf ein faßbares

Volumen von zunächst etwa 10 Gigabyte zu drücken. Wie die erzeugten Daten aussehen und in welches Format sie konvertiert wurden, wird im Anhang B geschildert.

Ein großes Problem stellten die langen Simulationszeiten dar, die von VERIFY benötigt wurden. Pro injiziertem Fehler benötigte die Simulation etwa 5 Minuten. Bei 11 Netztopologien mit jeweils 10000 Fehlern entspricht dies einer Rechenzeit von gut einem Jahr. Da der kommerzielle VHDL-Simulator von Modeltech bei der Simulation etwa um den Faktor 10-50 schneller ist, wurde nach einem Weg gesucht, die Simulation mit diesem durchzuführen. Das Problem, das sich dabei stellte, war, daß es sich bei diesem Simulator um einen normalen VHDL- Simulator handelt, der nicht zur Fehlerinjektion vorbereitet ist. Er bietet jedoch die Möglichkeit, im Batch-Modus gesteuert zu werden. Der Simulator von VERIFY wurde nun so modifiziert, daß er nur noch die Zeitpunkte und -dauern der Fehlerinjektionen bestimmt und ausgibt, aus denen dann ein Script erzeugt wird, das wiederum den Modeltech-Simulator steuert und ihn veranlaßt, die Simulationen mit jeweils einer Aktivierung eines Fehlersignals pro Lauf durchzuführen. Durch paralleles Arbeiten auf 5 Maschinen konnte mit dieser Hybrid-Methode aus VERIFY- und Modeltech-Simulator die Simulationszeit auf knapp 4 Tage reduziert werden. Im Anhang ist in Abschnitt C dargestellt, wie die Batch-Skripten für den Modeltech-VHDL-Simulator erzeugt werden und aussehen.

Um den Rollback-Verfahren in der Simulation die Möglichkeit zur Wiederherstellung zu geben, ist es nötig, die Systeme um eine Zeit länger als benötigt zu simulieren. Wenn sich z.B. bei einem Macro-Rollback-Netz ein Fehler auf das letzte Pattern auswirkt und eine Rekalkulation angestoßen wird, wird das korrekte Ergebnis eine vollständige Durchlaufzeit der Pipeline später geliefert. Der Zeitpunkt des Simulationsendes wurde für alle Netze adäquat gewählt.

5.1.4 Analyse

Dieses Kapitel beschreibt, wie aus den Traces Informationen gewonnen werden können, die Aufschluß über die FT-Charakteristik eines Systems liefern. Dazu werden mehrere Fehlverhalten spezifiziert. Außerdem wird erläutert, wie eng die Konfidenz für die ermittelten Werte ist.

Da bei einer Datenmenge von 10 GB nicht mehr mit einer schnellen Auswertung zu rechnen ist, wurde zunächst nach einer Möglichkeit gesucht, die Daten weiter zu komprimieren. In einem ersten Schritt wurden alle 10000 Fehler-Traces jeweils mit dem korrekten Trace verglichen und in einem File festgehalten, wie für jeden injizierten Fehler der Unterschied des Verhaltens zum korrekten Durchlauf aussieht. Die resultierende Datenmenge beträgt ca. 10 MB, was einem Kompressionsdivisor von etwa 1000 entspricht. Auf dieser Datenmenge läßt sich bequem interaktiv agieren. Die Wartezeiten zur Ermittlung der in Kapitel 5.2 dargestellten Daten beträgt nur wenige Sekunden.

Für die Transformation der Unterschiede zwischen den Fehler-Traces und dem korrekten Trace in eine Bewertung zur FT-Charakteristik wurden mehrere Klassen von Fehlverhalten für die Netz-Topologien spezifiziert:

- Das schwerwiegendste Fehlverhalten ist ein vollständiges Blockieren und somit Ausfall der Pipeline nach einem Fehler. Dies kann aus den Traces ermittelt werden, indem nach

dem Auftreten eines Fehlers keine Ergebnisse mehr geliefert werden. Dieses Verhalten wird im folgenden Blocking genannt.

- Ein weiteres Fehlverhalten ist, wenn ein oder mehrere Ergebnisse fehlen. Dies kann ermittelt werden, indem in den Fehler-Traces nachgesehen wird, welche Indizes bei den Ausgaben nicht geliefert wurden. Dieses Verhalten wird im folgenden Missing genannt.

- Gleichbedeutend zum Fehlen von Ergebnissen ist das Liefern falscher Ergebnisse einzustufen. Dies kann ermittelt werden, indem jedes Ergebnis eines Fehler-Traces mit dem entsprechend dem Index gewählten Ergebnis des korrekten Trace verglichen wird. Wobei hier noch ermittelt werden könnte, um wieviel Prozent der Ist-Wert vom Soll-Wert abweicht. Dieses Verhalten wird im folgenden Drifting genannt.

- Ein für die Rollback-Verfahren zu erwartendes Fehlverhalten ist die zeitliche Verschiebung einiger Ergebnisse. Dies kann aus den Traces ermittelt werden, indem für ein Paar aus Index und Ergebnis der Vergleich von Fehler-Trace mit korrektem Trace zwar positiv ausfällt, aber die Zeiten, zu denen diese Daten eintrafen, unterschiedlich sind. Dieses Verhalten wird im folgenden Delayed genannt.

Die FT-Charakteristik einer Netz-Topologie wird in Kapitel 5.2 durch einfaches abzählen, wie viele von den 10000 Fehlern welches Verhalten hervorriefen, ermittelt. Dabei sei darauf hingewiesen, daß die ermittelten Wahrscheinlichkeiten nur mit einer gewissen Konfidenz den tatsächlichen Wahrscheinlichkeiten entsprechen. Bei 10000 Experimenten liegt die tatsächliche Wahrscheinlichkeit, laut Tschebyscheff-Ungleichung, für das Auftreten des Verhaltens mit 90%iger Sicherheit in einem Intervall von $\pm 1.6\%$ um den ermittelten Wert. Dies wirkt sich besonders bei Werten aus, die nahe den 100% liegen. Wird zum Beispiel bei 99% aller injizierter Fehler kein Fehlverhalten diagnostiziert, liegt die tatsächliche Wahrscheinlichkeit mit einer Sicherheit von 90% in dem Intervall von 97.4% bis 100%.

5.2 Bewertung der Realisierungen

In diesem Kapitel findet sich die Auswertung der Simulationsergebnisse. In erster Linie wird dargestellt, mit welchen Anteilen welches Fehlverhalten verursacht wurde. In einem zweiten Schritt wird aufgrund der Fehler, die das Fehlverhalten bewirkt haben, analysiert, wie man den Schaltkreis besser konstruieren könnte.

An dieser Stelle sei darauf hingewiesen, daß die Genauigkeit von 4 Stellen für das Auftreten der Verhaltensweisen nur gewählt wurde, um erkennen zu können, wie viele der 10000 Experimente zu diesem Verhalten führten. Die tatsächliche Auftrittswahrscheinlichkeit unterliegt der Tschebyscheff- Ungleichung und liegt, wie in Kapitel 5.1.4 dargestellt, in einem Intervall um den ermittelten Wert herum. Je kleiner das Intervall, desto weniger wahrscheinlich ist es, daß die wirkliche Auftrittswahrscheinlichkeit in diesem Intervall liegt.

Weiterhin sei darauf hingewiesen, daß die in den folgenden Kapiteln zu den verschiedenen Implementierungsvarianten ermittelten Wahrscheinlichkeiten nur eine Aussage liefern, wie viele

Tabelle 5.1: Fehlerauswirkungen im funktionalen Netz

Typ des Netzes	Blocking	Missing	Drifting	Correct	Delayed	Faults
Bitparallele Implementierung	-	0.51 %	8.14 %	91.35 %	-	30174
Bitredundante Implementierung	-	0.70 %	11.74 %	87.56 %	-	29664
Bitserielle Implementierung	-	1.38 %	6.86 %	91.76 %	-	10521

von den 10000 zufällig ausgewählten Fehlern zu welchem Verhalten führen, weshalb die Werte auch nicht direkt miteinander verglichen werden können. Um vergleichbare Zahlen zu produzieren, müssen aus diesen Wahrscheinlichkeiten Ausfallraten berechnet werden. In diese Ausfallraten geht die Wahrscheinlichkeit, mit der ein Fehler ein Fehlverhalten ausgelöst hat, ein und zusätzlich die Anzahl der möglichen Fehler. Dies ist in Kapitel 5.3 beschrieben.

Die Analyse der Simulationsdaten ist in den folgenden Unterkapiteln 5.2.1-5.2.4 passend zu der jeweiligen Netz-Topologie dargestellt. In jeweils einer Tabelle zu jeder Topologie wird in den Spalten aufgenommen, für wie viele der 10000 injizierten Fehler die drei Implementierungsvarianten einer Netz-Topologie welches Verhalten an den Tag legten. Die Spalte ganz rechts enthält die Anzahl möglicher Fehler im Design. Diese Zahl wird in Kapitel 5.3 benötigt, um die Ausfallraten aus den Wahrscheinlichkeiten zu ermitteln.

Die ermittelten Simulationsdaten bieten die Möglichkeit, weiter als hier dargestellt ins Detail zu gehen. Im Detail gesehen sind die Fehler nach Art des injizierten Fehlers, Stuck At 0/1 oder Bitflip, und nach dem Ort, im Knoten, Netzwerk oder deren Subkomponenten, in denen sie auftreten, gegliedert. Um die Tabellen nicht unnötig auszuweiten wurde in den folgenden Kapiteln auf die Darstellung aller Daten verzichtet und stattdessen nur für jede Implementierungsvariante die Anzahl der Fehlverhalten für alle Arten und Orte, auf welche bzw. an denen Fehler injiziert wurden, aufsummiert dargestellt. Wenn die weitere Aufschlüsselung der Fehler zu interessanten Folgerungen führen, werden entsprechende Teile der detaillierten Ergebnisse dargestellt. Im Anhang D sind Ausdrucke dargestellt, wie fein-granularere Daten gewonnen wurden.

5.2.1 Funktionales Netz

In diesem Abschnitt wird dargestellt, wie die drei funktionalen Implementierungen auf injizierte Fehler reagieren, siehe Tabelle 5.1. Da diese Netze nicht rückgekoppelt sind, tritt hier weder Blocking- noch Delayed-Fehlverhalten auf.

Diese Werte sind nur als Referenz zu gebrauchen. Später werden die Ausfallraten der mit Fehlertoleranz erweiterten Modelle gegenüber diesen verglichen, um zu ermitteln, ob die Fehlertoleranzmaßnahmen wirklich einen Vorteil bringen.

Auffällig an dieser Tabelle ist die kleine Anzahl möglicher Fehler für die bitserielle Variante. Da die Anzahl der Fehler proportional zur Anzahl der Gatter ist und die bitserielle Variante die günstigste bezüglich der Chipfläche und somit des Gatterbedarfs, ist dies nicht überraschend.

Tabelle 5.2: Fehlerauswirkungen im Macro-Rollback Netz

Typ des Netzes	Blocking	Missing	Drifting	Correct	Delayed	Faults
Bitparallele Implementierung	-	0.34 %	0.24 %	99.42 %	7.78 %	61123
Bitredundante Implementierung	-	0.56 %	0.23 %	99.21 %	11.19 %	60103
Bitserielle Implementierung	-	0.61 %	0.14 %	99.25 %	5.84 %	23873

Der relativ hohe Wert für das Missing-Fehlverhalten der bitseriellen gegenüber den anderen beiden Varianten ist ebenfalls auf die geringere Komplexität des seriellen Entwurfs zurückzuführen: Da das Missingverhalten jeweils auf ein Kippen eines Bits eines Indexregisters, die in allen drei Varianten gleich viele Gatter beanspruchen, zurückzuführen ist und in der seriellen bei der Auswahl der Fehler weniger Möglichkeiten, etwa ein drittel, zur Verfügung standen, wurde die Injektion von Fehlern in den Indexteil des Netzes etwa um den Faktor drei begünstigt. Rechnet man dies mit ein, bekommt man einen Wert von 0.46 für die bitserielle Variante. Kalkuliert man nun noch ein, daß ein Intervall von $\pm 0.12\%$ $0.70 - 0.46/2$ mit der Tschebyscheff-Ungleichung nicht mehr zu erfassen ist, sind die drei Werte als sehr ähnlich in diesem Kontext einzustufen.

Wesentlich mehr Aufmerksamkeit ist den stark divergierenden Werten für das Drifting-Fehlverhalten zu widmen, da sie sich nahe an der Grenze mit einer Sicherheit von über 90% bewegen. Hier fällt das Hervorstechen der bitredundanten Variante mit 11.74% gegenüber 8.14% bei nahezu gleicher Anzahl insgesamt injizierbarer Fehler auf. Selbst detaillierte Analysen nach Fehlerart und Fehlerort ließen keinen Rückschluß auf z.B. die doppelte Anzahl von Flipflops im bitredundanten gegenüber dem bitparallelen Fall zu, weshalb dieser Effekt der anderen Additionsmethode zugeschrieben wird.

Eine detaillierte Analyse, siehe Anhang D, des Fehlerortes hat ergeben, daß das Missing-Fehlverhalten ausschließlich für Fehler folgte, die in der Netz-Topologie injiziert wurden, was wegen der Plazierung der Indexregister außerhalb der Knoten zu erwarten war.

5.2.2 Macro-Rollback

In diesem Kapitel werden die Simulationsergebnisse der drei Macro-Rollback Netze in Tabelle 5.2 dargestellt. Da dieses Netz rückgekoppelt ist, ist hier theoretisch Blocking-Fehlverhalten denkbar, wurde aber nicht beobachtet.

Die Verhältnisse für die Anzahl injizierbarer Fehler sind etwa gleich zur funktionalen Netztopologie: Bitparallel und bitredundante Variante beinhalten jeweils etwa dreimal so viele mögliche Fehler wie die bitserielle.

Da das angewendete FT-Verfahren durch Wiederholung einer Operation Fehler zu kompensieren versucht, treten hier im Falle erkannter Fehler Delays für berechnete Werte auf. In der Spalte Correct sind diese zu den nicht-verzögerten bereits aufaddiert.

Tabelle 5.3: Fehlerauswirkungen im Micro-Rollback Netz

Typ des Netzes	Blocking	Missing	Drifting	Correct	Delayed	Faults
Bitparallele Implementierung	0.06 %	0.41 %	0.06 %	99.47 %	7.13 %	62048
Bitredundante Implementierung	0.11 %	0.48 %	0.12 %	99.29 %	10.51 %	62044

Betrachtet man die Fehlerorte, fällt auf, daß nun alle Drifting-Fehlverhalten erzeugenden Fehler im Netzwerk liegen, während kein Fehler innerhalb eines Knotens zu einem Drifting-Fehlverhalten führte, was den Schluß zuläßt, daß diese von den Komparatoren dieser Topologie erkannt und durch ein Rollback erfolgreich korrigiert wurden.

Im Gegensatz zum funktionalen Netz wurde für diese Netztopologie auch Missing-Fehlverhalten für Fehler beobachtet, die nicht in das Netzwerk sondern in Knoten injiziert wurden. Dieses Fehlverhalten trat nur für die parallelen Varianten auf und die Fehler waren stets in der letzten Stufe der Pipeline in einem Flipflop injiziert worden, das den Wert für den Operanden x hielt. Die Analyse ergab, daß aufgrund des Vergleichs der parallel arbeitenden Pipelines das Signal, welches am Ausgang des Netzes die Gültigkeit der Daten signalisiert, zwar ein 'ungültig' signalisiert, aber am Eingang aufgrund der Laufzeit nicht die Daten zur Wiederholung der Operation sondern neue Daten geladen werden. Dadurch geht eine gesamte Operation in seltenen Fällen verloren. Dieses Verhalten ist auch für die bitserielle Variante vorhanden, wurde aber aufgrund seiner geringen Wahrscheinlichkeit nicht beobachtet.

5.2.3 Micro-Rollback

In diesem Kapitel werden die Simulationsergebnisse der drei Micro-Rollback Netze in Tabelle 5.3 dargestellt. Da diese Netz-Topologie wie auch schon die des Macro-Rollback rückgekoppelt ist, ist hier Blocking-Fehlverhalten zu erwarten und im Gegensatz zum Macro-Rollback auch beobachtbar.

Da die bitserielle Variante für diese Netz-Topologie ineffizient eingestuft und nicht implementiert wurde, siehe Kapitel 4.4.3, existieren dafür auch keine Simulationsdaten.

Für die Micro-Rollback Netze sind die Anzahl möglicher Fehler für die beiden implementierten Varianten nahezu identisch. Die leicht erhöhten Wahrscheinlichkeiten für ein Fehlverhalten der injizierten Fehler in die bitredundante Variante liegen in derart engen Intervallen, daß sie aufgrund der Tschebyscheff-Ungleichung keine verläßlichen Folgerungen erlauben.

Im Gegensatz zu allen anderen in dieser Arbeit dargestellten Netz-Topologien wurde durch Implementierung einer FT-Maßnahme auch ein neues Fehlverhalten erzeugt. Die Micro-Rollback-Topologie zeigt in seltenen Fällen Blocking-Verhalten, was bedeutet, daß die Einheit für den weiteren Betrieb völlig unbrauchbar wird. Die Ursache für dieses Verhalten ist in allen Fällen ein Bitflipfehler in einem Flipflop eines Knotens, das einen Ausgabeoperanden gespeichert hat. Geschieht dies zu einem Zeitpunkt, nachdem der Steuerautomat für die Stufe in den Zustand gesprungen ist, indem er gültige, korrekte Daten in den Knoten annimmt, und bevor die nächste

Tabelle 5.4: Fehlerauswirkungen im TMR Netz

Typ des Netzes	Blocking	Missing	Drifting	Correct	Delayed	Faults
Bitparallele Implementierung	-	0.03 %	0.01 %	99.96 %	-	91319
Bitredundante Implementierung	-	0.07 %	0.06 %	99.87 %	-	89789
Bitserielle Implementierung	-	0.06 %	0.03 %	99.91 %	-	32826

Stufe diese Daten übernommen hat, wird dieser nächsten Stufe ständig die Gültigkeit korrekter Daten signalisiert, obwohl fehlerbehaftete Daten anliegen, was letztlich zum Blockieren dieser Stufe und somit der gesamten Pipeline führt.

Betrachtet man das Missing-Verhalten, erkennt man, daß wie auch schon bei der Macro-Rollback-Topologie dieses durch Fehler in den Knoten erzeugt werden kann. Beim Micro-Rollback wurde dieses Verhalten jedoch nicht nur bei Fehlern in der letzten Stufe erzeugt sondern in beliebigen. Die Ursache für dieses Verhalten war, daß der Steuerautomat der Pipelinestufe einen unerlaubten Zustandsübergang aufgrund eines Laufzeitproblems vollzog: Der Steuerautomat enthält u.a. zwei Flipflops, in denen die vier Zustände kodiert werden. Während die Übergangsfunktion für das eine Flipflop 3 Gatterlaufzeiten beansprucht, wurde die Berechnung für das andere in nur zwei Gatterlaufzeiten vollzogen. Der Fehler trat nun zu einem Zeitpunkt auf, bei dem er sich auf das eine Flipflop auswirkte, während das andere korrekt geschaltet wurde, was sich in einem unerlaubten Zustandsübergang auswirkte, bei dem aus einer Validmarkierung für eine Stufe eine Emptymarkierung wurde und somit eine Operation verloren ging.

Das Drifting-Fehlverhalten wird bei dieser Netz-Topologie, wie auch bei der Macro-Rollback-Topologie, ausschließlich von Fehlern im Netz erzeugt. Fehler in den Knoten führen nicht zu diesem Fehlverhalten, was zum Schluß führt, daß die FT-Maßnahme diese Fehler erfolgreich korrigiert.

5.2.4 TMR

In diesem Kapitel werden die Simulationsergebnisse der drei TMR Netze in Tabelle 5.4 dargestellt. Diese Netz-Topologie ist frei von Rückkopplungen, weshalb hier kein Blocking-Fehlverhalten zu erwarten war und auch nicht auftritt. Da bei dieser FT-Maßnahme die Korrektur von Fehlerauswirkungen durch eine zwei aus drei Auswahl erreicht wird und nicht durch eine Wiederholung der Rechenoperation, wurde durch die injizierten Fehler kein Delay-Fehlverhalten provoziert.

Wegen der Tschebyscheff-Ungleichung können die durchwegs geringen Wahrscheinlichkeiten für ein Fehlverhalten eines injizierten Fehlers bzw. die hohe Kompensationswahrscheinlichkeit nicht so gut gewertet werden, wie es auf den ersten Blick scheint, weil die Sicherheit für z.B. die 99.96%ige Kompensationswahrscheinlichkeit mit einer Wahrscheinlichkeit von 10% sogar weniger als 98.36% beträgt. Um diesen Werten zuverlässigen Halt zu geben, müßte die Anzahl der Experimente um ein Vielfaches erhöht werden.

Wie der Name Triple-Modular-Redundancy schon sagt, werden bei dieser Netz-Topologie die Komponenten verdreifacht, was zu einer entsprechenden Verdreifachung der Anzahl der möglichen Fehler führt. Im Gegensatz zu den anderen beiden fehlertoleranten Ansätzen wurden hier auch die Komponenten des Netzwerkes verdreifacht, so daß Auswirkungen von Fehlern nur noch möglich sind, wenn sie in den Votern oder danach injiziert werden. Bei der Analyse der Fehlerorte wurde dementsprechend festgestellt, daß Fehler, die ein Fehlverhalten provozierten, ausschließlich in den Votern plaziert waren. Alle anderen Fehler wurden kompensiert.

5.3 Bestimmung der Ausfallraten

Nachdem in den vorangegangenen Kapiteln statistisch erfaßt wurde, mit welcher Wahrscheinlichkeit bei 10000 injizierten Fehlern welches Fehlverhalten ausgelöst wird, wird in diesem Kapitel beschrieben, wie man diese Wahrscheinlichkeiten in Ausfallraten umrechnet, um die verschiedenen Topologien basierend auf den ermittelten Wahrscheinlichkeiten vergleichen zu können. Der resultierende Vergleich ist ebenfalls in diesem Kapitel dargestellt.

Wie schon erwähnt, wird durch die in den vorangegangenen Kapiteln dargestellten Wahrscheinlichkeiten nur beschrieben, wie viele von den 10000 zufällig ausgesuchten Fehlern sich mit einem Fehlverhalten auswirkten. Diese Größen können nicht miteinander verglichen werden, was anhand eines Beispiels im folgenden erläutert wird: Angenommen ein System bietet 20000 mögliche Fehler, von denen sich bei einer zufälligen Auswahl von 1000 Fehlern 100 mit einem Fehlverhalten auswirken. Würde man diesem System funktionslose Gatter hinzufügen, die weitere 20000 mögliche Fehler bieten, von denen sich aber keiner auswirkt, würden bei Wiederholung des Experiments mit 10000 zufällig gewählten Fehlern nur noch ca. 50 Fehler zu einem Fehlverhalten führen, weil nur noch 5000 Fehler in den relevanten Teil des Systems und die anderen 5000 in den sinnlosen Teil des Systems injiziert werden.

Um vergleichbare Werte zu erzielen, ist eine Umrechnung dieser Wahrscheinlichkeiten, mit der ein Fehler zu einem Fehlverhalten führt, in Ausfallraten des Gesamtsystems nötig. Die Ausfallrate aufgrund eines Fehlers ergibt sich aus dem Produkt der Fehlerrate und dessen Wahrscheinlichkeit, ein Fehlverhalten auszulösen. Um die Ausfallrate des gesamten Systems zu ermitteln, muß man über die Ausfallraten aller Fehler summieren. Da die Fehlerraten für alle Fehler gleich groß ($1/Zeiteinheit$) gewählt wurden, hängt die Ausfallwahrscheinlichkeit nur noch von der ermittelten Wahrscheinlichkeit multipliziert mit der Anzahl möglicher Fehler ab. In den Tabellen 5.5, 5.6, 5.7, 5.8, 5.9, 5.10 und 5.11 sind diese Werte für alle Architekturen zusammengetragen. In den ersten drei Tabellen 5.5, 5.6 und 5.7 sind die Daten nach den drei Knotentypen bitseriell, -parallel und -redundant sortiert, um die verschiedenen FT-Maßnahmen direkt gegeneinander vergleichen zu können, während die Tabellen 5.8, 5.9, 5.10 und 5.11 nur eine andere Anordnung der ermittelten Werte enthalten. In diesen können die verschiedenen Knotentypen verglichen werden.

Auch an dieser Stelle sei wieder darauf hingewiesen, daß die berechneten Ausfallraten auf statistisch ermittelten Daten basieren, die aufgrund der endlichen Anzahl von Experimenten nur zur Abschätzung dienen können. Um die Verläßlichkeit der Ausfallraten abschätzen zu können, wurde jeweils in der letzte Zeile der Tabellen 5.5-5.11 die Abweichung angegeben, mit

Tabelle 5.5: Ausfallraten der bitparallelen Systeme

Fehlerverhalten	Funk	Macro	Micro	TMR
Blocking	0.00	0.00	37.23	0.00
Missing	153.89	207.82	254.40	27.40
Drifting	2456.16	146.70	37.23	9.13
Delayed	0.00	4755.37	4424.02	0.00
Abweichung	482.78	977.97	992.77	1461.10

Tabelle 5.6: Ausfallraten der bitredundanten Systeme

Fehlerverhalten	Funk	Macro	Micro	TMR
Blocking	0.00	0.00	68.25	0.00
Missing	207.65	336.58	297.81	62.85
Drifting	3482.55	138.24	74.45	53.87
Delayed	0.00	6725.53	6520.82	0.00
Abweichung	474.62	961.65	992.70	1436.62

der ein Intervall für die Fehlerraten berechnet werden kann, das aufgrund der Tschebyscheff-Ungleichung mit 90%iger Sicherheit der Realität entspricht. Dabei fällt auf, daß die Intervallgröße die eigentlichen Raten teilweise um Größenordnungen übersteigt. Die Intervallgröße wurde ermittelt, indem die Anzahl der Fehler eines Modells mit 0.016 (1.6%), dem Wert für 90%ige Sicherheit nach Tschebyscheff, multipliziert wurde.

5.4 Auswertung und Folgerungen

Um das günstigste Verfahren im Allgemeinen zu ermitteln, ist die Frage von Relevanz, welche Fehlverhalten für den Betrieb tolerierbar sind. Als Kostenfaktor bei der Produktion des Chips spielt der Rechendurchsatz bezüglich der Chipfläche die größte Rolle. Im Folgenden wird ausgehend von einem geforderten FT-Verhalten des Systems herausgestellt, welches das günstigste ist.

Tabelle 5.7: Ausfallraten der bitseriellen Systeme

Fehlerverhalten	Funk	Macro	TMR
Blocking	0.00	0.00	0.00
Missing	145.19	145.63	19.70
Drifting	721.74	33.42	9.85
Delayed	0.00	1394.18	0.00
Abweichung	168.34	381.97	525.22

Tabelle 5.8: Ausfallraten der funktionalen Systeme

Fehlerverhalten	bitparallel	bitredundant	bitseriell
Blocking	0.00	0.00	0.00
Missing	153.89	207.65	145.19
Drifting	2456.16	3482.55	721.74
Delayed	0.00	0.00	0.00
Abweichung	482.78	474.62	168.34

Tabelle 5.9: Ausfallraten der Macro-Rollback Systeme

Fehlerverhalten	bitparallel	bitredundant	bitseriell
Blocking	0.00	0.00	0.00
Missing	207.82	336.58	145.63
Drifting	146.70	138.24	33.42
Delayed	4755.37	6725.53	1394.18
Abweichung	977.97	961.65	381.97

In Anbetracht der Tatsache, daß die TMR-Netze durchwegs bessere Ausfallraten bieten, als das Micro-Rollback-Netz und das TMR-Netz zusätzlich einen um etwa 30% höheren Rechendurchsatz, siehe Tabelle 4.8, als das Micro-Rollback-Netz bietet, ist das TMR dem Micro-Rollback-System generell vorzuziehen.

Vergleicht man innerhalb einer Netztopologie die verschiedenen Varianten, siehe Tabelle 5.8-5.11, schneiden die bitparallelen Varianten bis auf die Ausnahme in Tabelle 5.9 nicht nur tendenziell sondern teilweise auch mit hoher Konfidenz besser ab als die bitredundanten Ansätze. Da die bitredundanten Ansätze bezüglich Rechendurchsatz um knapp 50% schlechter sind als die bitparallelen, ist generell den bitparallelen Varianten vor den bitredundanten der Vorzug zu geben. Vergleicht man bitparallele und bitserielle Netze, zieht sich ausgehend vom funktionalen Netz, siehe Tabelle 5.8, ein Vorteil der bitseriellen Netze durch alle Netztopologien. Der Preis für diesen Vorteil beträgt ca. 50% des Rechendurchsatzes, um den die bitparallelen Netze besser sind.

Geht man von der schärfsten Forderung aus, daß keines der vier Fehlverhalten im System ak-

Tabelle 5.10: Ausfallraten der Micro-Rollback Systeme

Fehlerverhalten	bitparallel	bitredundant
Blocking	37.23	68.25
Missing	254.40	297.81
Drifting	37.23	74.45
Delayed	4424.02	6520.82
Abweichung	992.77	992.70

Tabelle 5.11: Ausfallraten der TMR Systeme

Fehlerverhalten	bitparallel	bitredundant	bitseriell
Blocking	0.00	0.00	0.00
Missing	27.40	62.85	19.70
Drifting	9.13	53.87	9.85
Delayed	0.00	0.00	0.00
Abweichung	1461.10	1436.62	525.22

zeptabel ist, entspricht die Fehlerrate der Summe der Fehlerraten über alle Fehlverhalten. Unter diesen Bedingungen schneiden die TMR-Systeme mit hoher Konfidenz um Größenordnungen besser ab, als alle anderen. Die Rollback-Verfahren schneiden unter diesen Bedingungen sogar schlechter ab, als die rein funktionalen Netze, weil sie aufgrund der Duplizierung der Knoten auch die Fehlerraten im Gesamtsystem verdoppeln, was aber nur durch eine Verzögerung der Ergebnisse abgefangen wird. Der Preis für die erhöhte Fehlertoleranz des TMR-Netzes im Vergleich zum funktionalen Netz ist ein Verlust des Rechendurchsatzes von mehr als 66%, siehe Tabelle 4.8. Hier gilt es dann abzuwägen, wie viel die wesentlich bessere Ausfallrate der TMR-Systeme gegenüber den rein funktionalen Systemen wert ist.

Geht man davon aus, daß nur eine Verzögerung, Delayed-Fehlverhalten, der Ergebnisse akzeptabel ist, was letztlich einer Summierung über die restlichen drei Fehlverhalten entspricht, werden die Macro-Rollback-Netze attraktiver: Sie haben zwar noch höhere Ausfallraten als die TMR-Netze, aber diese liegen im Bereich schlechter Konfidenz, so daß sie nur noch als Tendenzen gewertet werden dürfen. Sie sind aber mit hoher Konfidenz wesentlich ausfallsicherer als die rein funktionalen Netze. Gepaart mit den Werten für den Rechendurchsatz, etwa 50% höher als der der TMR-Netze, bieten die Macro-Rollback-Netze für diesen Fall eine ernstzunehmende Alternative zu den TMR-Netzen.

Geht man davon aus, daß wenigstens ein Ergebnis geliefert werden muß, wenn es auch falsch ist, wenn also nur Blocking- und Missing-Fehlverhalten erlaubt ist, sticht wieder das TMR-System mit der niedrigsten Ausfallrate allerdings geringer Konfidenz heraus. Die Rollback-Verfahren liegen tendenziell mit leicht höheren Ausfallraten schlechter als das funktionale Netz, wodurch die Entscheidung wieder auf die Wahl des funktionalen oder TMR-Systems eingeschränkt wird. Hier muß wieder abgewogen werden, ob die zusätzliche Sicherheit des TMR-Systems die um ca. den Faktor 3 höheren Rechendurchsatz des funktionalen Netzes wert ist.

Geht man davon aus, daß nur ein Totalausfall der Pipeline nicht toleriert werden darf, scheiden nur die Rollback-Systeme aus. Hier wurde das Macro-Rollback-Verfahren mit genannt, weil durch die Einführung der zusätzlichen Steuerautomaten davon ausgegangen werden kann, daß auch hier, wie beim Micro-Rollback-Netz, eine Verklemmung einsetzen könnte, obwohl sie während der Experimente nicht beobachtet werden konnte. Für diesen Fall ist dann das rein funktionale Netz, das sich durch dreifachen Rechendurchsatz gegenüber dem TMR-Netzes auszeichnet, das günstigste.

Weiterhin fällt auf, daß im Falle des Missing-Fehlverhaltens die Fehlerraten für die beiden Rollback-Topologien tendenziell steigen. Tendenziell deshalb, weil die Intervalle für die Aus-

fallraten bei 90%iger Sicherheit gut doppelt so groß sind wie für das jeweils funktionale Netz. Durch die Lokalisation der Fehlerursachen konnten diese Tendenzen bestätigt werden: Im Falle der Rollback-Netze wurden neue Möglichkeiten für Fehlerauswirkungen dieser Art geschaffen und somit das Missing-Fehlverhalten begünstigt.

Für das Drifting-Fehlverhalten kann trotz großer Sicherheitsintervalle festgehalten werden, daß alle Topologien mit FT-Maßnahmen wesentlich ausfallsicherer arbeiten, als die funktionalen Netze. Tendenziell sieht es so aus, als wären die TMR-Topologien etwa 3 Mal ausfallsicherer als die Micro-Rollback-Verfahren, welche wiederum zwei bis drei mal ausfallsicherer als die Macro-Rollback-Verfahren sind. Allerdings ist die Konfidenz der Werte sehr zweifelhaft, was besonders am Beispiel des bitparallelen TMR-Systems deutlich wird: Glänzt es mit einer Ausfallrate von gut 9 pro Zeiteinheit, beträgt das Intervall für 90%ige Sicherheit dieses Wertes knapp 1500 Ausfälle pro Zeiteinheit.

Beim Delayed-Fehlverhalten ist gut zu erkennen, daß für die Rollback-Verfahren die Ausfallrate etwa doppelt so hoch ist, wie für das Drifting-Fehlverhalten der entsprechenden funktionalen Netze. Dies läßt den Schluß zu, daß Fehler in den Knoten durch die ebenfalls Duplizierung der Knotenhardware der FT-Maßnahme von Drifting nach Delay transformiert werden. Diese Aussage hat auch Bestand, wenn man mit einkalkuliert, daß die Konfidenz für die entsprechenden Werte in einem Intervall liegt, da sich die Intervallgrenzen verschiedener Topologien nicht überlappen sondern relativ weit voneinander entfernt sind. Diese Aussage ist auch noch für die TMR-Topologien haltbar, deren Intervallgrenzen sich trotz der Größe der Intervalle nicht überschneiden.

Um verläßliche Aussagen über die FT-Charakteristiken dieser Systeme treffen zu können, muß die Größe der Intervalle in eine Größenordnung gebracht werden, die der der ermittelten Ausfallraten entspricht oder besser noch wesentlich kleiner ist. Dies ist abhängig von der Anzahl durchgeführter Fehlerinjektionen. Am Beispiel der Drifting-Rate für das bitparallele TMR-Netz bedeutet dies, daß etwa 25 Mal so viele Experimente durchgeführt werden müßten, was sowohl die Driftingrate von 9.13 als auch die Abweichung von 1461.10 in eine Zahl um die 250 transformieren müßte. Da die benötigte Rechenzeit linear mit der Anzahl der Experimente steigt, würde diese unter den Randbedingungen der hiesigen Rechensysteme 100 statt 4 Tage beanspruchen.

Betrachtet man die funktionalen Ansätze, fällt auf, daß alle drei etwa identische Raten für das Missing-Fehlverhalten zeigen, während jedoch das bitserielle beim Drifting-Fehlverhalten mit einer Ausfallrate hoher Konfidenz glänzt. Hierraus könnte die Hypothese abgeleitet werden, daß Realisierungen mit geringerem Gatteraufwand fehlertoleranter sind. Dies ist tendenziell auch in den fehlertoleranten Implementierungen für das Drifting-Fehlverhalten zu erkennen. Das Missing-Fehlverhalten, welches seine Ursache nicht in der Realisierung der Knoten hat, sondern vom Netzwerk, in dem die Knoten eingebettet sind, abhängt, bleibt von diesem Faktum unbeeinflußt, was oben genannte These unterstützt. Da es in dieser Arbeit jedoch gelang, fehlertolerantere Implementierungen durch Erhöhung des Gatteraufwands zu realisieren, bleibt nur der Schluß, daß unter bisher nicht klar definierbaren Randbedingungen die Reduzierung der Gatteranzahl eine Erhöhung der Fehlertoleranz mit sich bringt, so wie es eine Erhöhung ebenso tut.

Als gravierendstes Ergebnis der statistisch simulativen Analyse ist zu erwähnen, daß durch Integration einer Fehlertoleranzmaßnahme ein neues Fehlverhalten kreiert wurde, das ohne die FT-Maßnahme nicht auftreten konnte: Bei der Micro-Rollback-Topologie wurden in seltenen Fällen Auswirkungen von Fehlern beobachtet, die zum Blockieren und damit zum Totalausfall des Systems führten, während alle anderen Topologien, selbst die rein funktionale, ohne FT-Maßnahme, nach einer Refraktärzeit nach dem Verschwinden der Fehlerursache wieder normal arbeiteten. Zusätzlich hat die Integration der FT-Maßnahme das Missing-Fehlverhalten begünstigt und somit das System verschlechtert.

Die letzten beiden Absätzen zeigen, daß zusätzliche Funktionalität (zusätzliche Gatter) den ersehnten Fehlertoleranz-Vorteil nicht unbedingt erzielt, sondern daß hier behutsam auf Randbedingungen geachtet werden muß, die es in zukünftigen Arbeiten genauer herauszuarbeiten gilt. Durch effizientere Analysemethodik kombiniert mit höherer Rechenkapazität könnten eindeutigere Hinweise, an welcher Stelle hier zu suchen ist, erzielt werden. Ein anderer Weg, diese Randbindungen zu ermitteln, könnte über den rein theoretischen Ansatz erfolgreich verlaufen. Aber auch auf diesem Weg muß noch einige Denkarbeit geleistet werden.

Kapitel 6

Schluß

In diesem Kapitel wird kurz zusammengefaßt, was in dieser Arbeit geleistet wurde und welche Schlußfolgerungen aus den Ergebnissen abgeleitet werden können. Es wurden Digitalsysteme, Festkommaeinheiten, entwickelt, die Mutliplikation, Division, Wurzel, Logarithmus, Exponential-, Sinus-, Cosinus- und Arcustangensfunktion basierend auf Bit- und CORDIC-Algorithmen, siehe Kapitel 2, effizient bezüglich Chipfläche und Rechendurchsatz berechnen können. Dazu wurden diese Algorithmen in 11 verschiedenen Ansätzen implementiert, die alle mit dem gleichen Interface versehen waren, um einen direkten Vergleich der Implementierungsvarianten zu gewährleisten, siehe Kapitel 3 und 4. Die Effizienzbetrachtung der Implementierung galt nicht nur den üblichen Werten wie Rechendurchsatz und Chipfläche, sondern auch der Charakteristik unter der Randbedingung auftretender Fehler, siehe Kapitel 5. Die Werte für benötigte Chipfläche und erwartetem Rechendurchsatz wurden durch ein Synthese-Tool bestimmt, das aus den Modellen der Festkommaeinheiten Gatterlisten generierte. Die Werte für die Fehlertoleranz-Charakteristiken der Implementierungsvarianten wurden simulativ durch Fehlerinjektion in einem Hybridverfahren ermittelt. Die in der Fehlerinjektion ermittelten Werte wurden kritisch auf ihre Aussagekraft überprüft und deren Brauchbarkeit dargestellt. Zusätzlich wurde ein Auge auf die Übertragbarkeit der FT-Charakteristik auf andere Architekturen geworfen.

6.1 Zusammenfassung

Abschnitt 2 liefert die Grundlagen für die Bit- und CORDIC- Algorithmen, die herangezogen wurden, um die mathematischen Funktionen Multiplikation, Division, Wurzel, Logarithmus, Exponential-, Sinus-, Cosinus- und Arcustangensfunktion in einem Digitalsystem zu realisieren, wobei aufgrund der günstigeren Eigenschaften entschieden wurde, die Algorithmen in Struktur einer Pipeline zu implementieren, siehe Kapitel 2.3.4. Dadurch ergaben sich Optimierungsmöglichkeiten, die in Kapitel 2.4 dargestellt sind.

Durch die Pipelinestruktur ergab sich die Differenzierung zwischen Knoten, die den Berechnungseinheiten einer Stufe der Pipeline entsprechen, und Netzen, der Verbindungsstruktur zwi-

schen den Knoten.

Kapitel 3 stellt dar, auf welche Arten eine einzelne Stufe der Pipeline implementiert wurde. Es wurden eine bitparallele, siehe Unterkapitel 3.1, eine bitredundante, siehe Unterkapitel 3.2, und eine bitserielle, siehe Unterkapitel 3.3, Möglichkeit realisiert und als Knotentyp für die Verbindungsstrukturen zur Verfügung gestellt. Abschließend wurde in Unterkapitel 3.4 gegenübergestellt, welche Charakteristik die verschiedenen Knotentypen bezüglich Rechendurchsatz und Chipflächenbedarf zeigen.

Kapitel 4 beschreibt die Verbindungsstrukturen (Netztopologien) zwischen den Knoten. Mit dem Ziel im Auge, fehlertolerante Hardware mit möglichst geringen Hardware-Overhead zu produzieren, gelang es nicht, in die Knoten mit vertretbarem Chipflächenzuwachs Fehlertoleranzmaßnahmen zu integrieren, die z.B. eine Fehlerdetektion durchführten. Deshalb wurde für alle fehlertoleranten Implementierungen der Weg eingeschlagen, Berechnungen doppelt, in zwei unabhängigen Knoten, durchzuführen und aufgrund eines Komparators bei Gleichheit die Ergebnisse für richtig zu erklären. Es wurden fünf Netztopologien untersucht, wobei für eine, siehe Unterkapitel 4.4.4, im Vorfeld aufgrund eines zu großem erwarteten Hardwareaufwands entschieden wurde, diese nicht zu realisieren. Es wurden Modelle für eine rein funktionale Verbindungsstruktur ohne FT-Maßnahmen, siehe Unterkapitel 4.4.1, zwei verschiedene Rollback-Verfahren, siehe 4.4.2 und 4.4.3, und ein TMR-Verfahren, siehe Unterkapitel 4.4.5, für jeweils jeden Knotentyp (nur die bitserielle Knotenvariante wurde beim Micro-Rollback-Verfahren ausgelassen) implementiert, für die jeweils mit einem Synthese-Tool der erwartete Hardwareaufwand durch eine Abbildung auf eine Gatterbibliothek berechnet wurde. Um die 11 Implementierungsvarianten miteinander vergleichen zu können, wurde das Netzinterface vereinheitlicht, siehe Unterkapitel 4.1.

Kapitel 5 beschreibt die Methode zur Ermittlung der Aussagen über die Fehlertoleranz-Charakteristiken der 11 Varianten und stellt die mit dieser Methode ermittelten Daten dar. Dazu wurde das Evaluierungs-Szenario, siehe Unterkapitel 5.1, dargestellt und Randbedingungen für die Ermittlung der Bewertung durchgeführt. Es handelt sich um eine simulationsbasierte Fehlerinjektion, dessen größtes Manko die langen Simulationszeiten zur Ermittlung der geforderten Werte sind. Der Vergleich der verschiedenen Varianten, siehe Unterkapitel 5.3, hat ergeben, daß allein die Form der Implementierung die FT-Charakteristik beträchtlich beeinflussen kann. Betrachtet man allein die 3 Varianten ohne Fehlertoleranzmaßnahmen, glänzt die bitserielle gegenüber der bitparallelen mit einer Ausfallrate, die gut drei Mal besser ist und gegenüber der bitredundanten Variante knapp fünf Mal besser ist, ohne FT-Maßnahmen angewendet zu haben.

Ein generelles Problem bei der simulationsbasierten, quasi experimentellen Auswertung ist die Konfidenz der ermittelten Werte, siehe Kapitel 5.1.4. Da die zu injizierenden Fehler nur zufällig ausgewählt wurden, besteht eine Kluft zwischen den ermittelten und tatsächlichen Wahrscheinlichkeiten für einen Systemausfall. Da manche dieser Werte nahe der 99.9% Marke lagen, aber bei 10000 Experimenten aufgrund der Tschebyscheffungleichung das Intervall für 90%ige Sicherheit bei $\pm 1.6\%$ liegt, ist für diese Werte die Konfidenz nicht ausreichend.

Aufgrund des eben geschilderten Konfidenz-Problems ergaben sich bei der Bewertung der Systeme, siehe Unterkapitel 5.4, Aussagen, die mit hoher Wahrscheinlichkeit zutreffen, weil sie in den entsprechenden Konfidenz-Intervallen liegen, und andere Aussagen, die aufgrund der

Konfidenz nur mit geringerer Wahrscheinlichkeit zutreffen und die deshalb nur als Tendenzen gewertet werden dürfen.

6.2 Ergebnisse und Folgerungen

In diesem abschließenden Abschnitt werden die Ergebnisse von Kapitel 5.4 aufgegriffen und weitere Folgerungen die simulationsbasierte Fehlerinjektion im Allgemeinen betreffend aufgestellt.

Als erstes sei darauf hingewiesen, daß die Fehlerinjektion basierend auf einem Fehlermodell durchgeführt wurde, für das keine in der Praxis beobachtete Fundierung existiert. Aus welchem Grund es dennoch gewählt wurde, ist in Kapitel 5.1.2 nachzuschlagen.

Gravierendstes Ergebnis der dargestellten Untersuchungen ist, daß durch die Einführung von Fehlertoleranzmaßnahmen auch neue Ausfallmöglichkeiten des Systems geschaffen und Fehlverhalten begünstigt wurden: Bei den Rollback-Netzen trat der Fall ein, daß durch den zusätzlichen Steuerautomaten ein Deadlock in der Pipeline aktiviert wurde, der bei den anderen Verfahren nicht möglich ist. Dieser Deadlock hat jedoch zum Totalausfall der Berechnungseinheit geführt, während die anderen Einheiten nach einer gewissen Refraktärzeit nach Verschwinden des Fehlers wieder korrekt arbeiteten. Dies unterstreicht die Notwendigkeit in einem Entwicklungsprozeß, nach einem Entwurfsschritt, der Fehlertoleranzmaßnahmen in ein Modell integriert, auch einen Verifikations- oder Validierungsschritt einzubauen, der solche Folgen erkennt. Dabei ist jedoch zweifelhaft, ob die simulationsbasierte Fehlerinjektion die richtige Methode für einen derartigen Verifikationsschritt sein kann. Die Antwort auf diese Frage hängt davon ab, mit welcher Konfidenz die Antwort gegeben werden soll. Soll die Sicherheit jenseits der 99% liegen, sind über 250000 Experimente nötig. Unter den Bedingungen in dieser Arbeit ist die Antwort ein klares nein, da für das einfache System in dieser Arbeit nur für 6 oder 11, je nach Variante, von 10000 Fehlern das entsprechende Verhalten beobachtet werden konnte. Zieht man nun noch in Betracht, daß die Fehler zufällig ausgewählt wurden, ist leicht denkbar, daß ein solches Verhalten von dieser statistischen Methode nicht unbedingt erfaßt wird, besonders wenn die Auftrittswahrscheinlichkeiten kleiner werden, was durch komplexere Systeme begünstigt wird, weil bei mehr Fehlermöglichkeiten die Selektion eines Fehlers, der das Fehlverhalten anstößt, unwahrscheinlicher wird. Allerdings versagen analytische Verfahren aufgrund ihres Speicher- oder Rechenbedarfs schon bei sehr einfachen Systemen und scheiden deshalb für die hier dargestellten Systeme völlig aus, wenn es nicht gelingt, ein System in für die formale Verifikation faßbare Subsysteme zu hierarchisieren. Eine aktuelle Lösung dieses Problems könnte in einer Kombination beider Verfahren liegen. Ergänzt man die Erfahrungen eines erfahrenen Ingenieurs mit diesen Verifikationsansätzen, sollte eine möglichst gute Abschätzung für das Ausfallverhalten möglich sein, wobei das Prädikat ’gut’ für den Ingenieur auch in der Richtung gemeint ist, die ermittelten Werte kritisch zu betrachten.

Weiterhin läßt sich aus den Simulationsergebnissen ersehen, daß eine Fehlertoleranzmaßnahme zwar ein Fehlverhalten verbessert, aber gleichzeitig ein anderes verschlechtern kann. Obwohl diese Folgerung in den Rahmen der Unsicherheit fällt, die durch große, überlappende Konfidenzintervalle bedingt sind, gelang es, motiviert durch diesen Hinweis, die zusätzlich versteckt

eingeführten Schwachstellen aufzudecken und somit eindeutig der Fehlertoleranzmaßnahme zuzuordnen.

Als am besten bezüglich der Ausfallrate dastehende Fehlertoleranzmaßnahme sind die TMR-Systeme zu nennen. Sie glänzen mit den niedrigsten Ausfallraten bezüglich aller hier dargestellten Fehlverhalten. Allerdings ist auch hier auf die mangelnde Konfidenz der ermittelten Werte hinzuweisen.

An dieser Stelle sei auch nochmal auf die Methodik zur Bewertung von fehlertolerantem Verhalten hingewiesen: Um Systeme bezüglich ihres fehlertoleranten Verhaltens bewerten zu können, genügt es nicht, nur die Wahrscheinlichkeiten der injizierten Fehler zu ermitteln, die ein Fehlverhalten verursachen. Um die Systeme vergleichen zu können, werden Fehlerraten benötigt, die man erhält, indem die ermittelten Wahrscheinlichkeiten mit der Anzahl insgesamt möglicher Fehler multipliziert werden.

Als weiteres gravierendes Ergebnis sei hier festgehalten, daß allein durch Modifikationen am Design ohne Integration einer Fehlertoleranzmaßnahme eine um den Faktor 3 bessere Ausfallrate erzielt werden konnte. Dies resultierte aus der Komplexität des Schaltkreises: Der bitserielle Ansatz bot nur ca. 10000 Fehlermöglichkeiten, während der bitparallele und -redundante ca. 30000 enthielten, was sich, wie in Kapitel 5.3 beschrieben, auf die Fehlerrate auswirkt.

Abschließend sei gesagt, daß die Methode der simulationsbasierten Fehlerinjektion Bewertungen mit hoher Konfidenz nur ermitteln kann, wenn eine genügend hohe Anzahl an Experimenten durchgeführt wird. Will man in den Bereich jenseits der 99.9% vordringen, sind mehr als 100000 Experimente notwendig, um eine Konfidenz zu erreichen, die mit einer Wahrscheinlichkeit mehr als 90% Sicherheit bietet, daß der reale Wert auch tatsächlich jenseits der 99.9% liegt. In dieser Arbeit wurde bereits die Parallelisierung der Simulationen beschrieben, und es wurden auf 4 Rechnern 10000 Experimente in 4 Tagen durchgeführt, wobei der Speedup quasi linear zur Anzahl der Rechner ist. Dies ermöglicht bereits heute für Systeme mit einem Komplexitätsgrad von ca. 5000 bis 15000 Gattern die Berechnung der Ausfallraten, wenn man entsprechend Rechenkraft einsetzt. Für eine Konfidenz jenseits der 99.99% müssen neue Wege eingeschlagen werden.

Anhang A

Stimuli-Generator

In diesem Kapitel befinden sich die Quellen für ein Programm, das in der Programmiersprache C verfaßt ist. Es generiert die Stimuli und erwarteten Antworten auf diese Stimuli. Zusätzlich werden die Konstanten für die geforderte Genauigkeit generiert.

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <math.h>

#include  "ld2str.h"
#include  "stimuli.h"

static int  rand64bits;

char    **stimuli_a, **stimuli_b, **stimuli_fkt;

static long double my_rand()
{
  int    i;
  long double  rnd,div;

  rnd = 0.0;
  div=1.0/(long double)RAND_MAX;
  for( i=0 ; i<rand64bits ; i++ ) {
    rnd += (long double)rand()*div;
    div /= (long double)RAND_MAX;
  }
  if( rnd > 1.0 ) {
    fprintf( stderr, "ERROR: rnd>1.0\n" );
    exit(1);
```

```
  }
  return rnd;
}

void stimuli_init(int acc, int num)
{
  int    i,j,k;
  long double  a;

  srand( 0x3e2db7 );      /* oh yäh, set it to a very stran-
ge number ;-) */
  i=log( (double)RAND_MAX ) / log(2);  /* get number of possi-
ble bits */
  rand64bits=64/i+1;      /* number of random numbers ma-
king more than 64 bits */

  stimuli_a   = (char**)malloc( num * sizeof(char*) );
  stimuli_b   = (char**)malloc( num * sizeof(char*) );
  stimuli_fkt = (char**)malloc( num * sizeof(char*) );
  if( stimuli_a == NULL || stimuli_b == NULL || stimu-
li_fkt == NULL ) {
    fprintf( stderr, "memory error\n" );
    exit(1);
  }
  for( i=0 ; i<num ; i++ ) {
    stimuli_a[i] = ld2str( my_rand() );
  }

  for( i=0 ; i<num ; i++ ) {
    stimuli_b[i] = ld2str( my_rand() );
  }
  j=0;
  for( i=0 ; i<num ; i++ ) {
    if( j==0 ) {
      j=255;
    }
    do {
      k=(int)(my_rand()*8.0);
    } while( ((1<<k) & j) == 0 );
    stimuli_fkt[i] = ld2str_fkts_bvec[k];
    j &= ~(1<<k);
  }
}
```

Anhang B

Tracefile Datenformat

In Abschnitt 5.1.3 wurde dargestellt, daß einige Schritte nötig waren, um die Datenflut, mit der man bei der Simulation dieser Modelle konfrontiert wurde, zu bewältigen. In diesem Abschnitt werden die Inhalte der verschiedenen Datenfiles kurz dargestellt, um einen Einblick zu bekommen, welche Schritte für ein Komprimieren dieser Daten nötig waren und wie sie in dieser Arbeit realisiert wurden.

Von einer Darstellung der Daten, wie sie VERIFY bietet, wird hier abgesehen, weil sie sofort aufgrund des riesigen Platzbedarfs verworfen wurde.

Die Ausgaben des VHDL-Simulators sind in Abbildung B.1 am Beispiel des bitparallelen, funktionalen Netzes *network_simple_par* dargestellt. Die Größe eines Tracefiles beträgt zwischen 10 und 20 Megabyte. Der dargestellte Bereich ist nur ein Ausschnitt der Simulation eines einzigen Fehlers. Anfangs der Simulation werden viele Warnings erzeugt. Die ersten und letzten 3, bevor sinnvoll simuliert wird, sind in Abbildung B.1 dargestellt. Die Warnings resultieren aus dem Mapping auf die Gatterbibliothek, die mit Signalen des Typs *std_logic* arbeitet, während das Ursprungsmodell mit dem Typ *bit* modelliert war. Da die Initialisierung der *std_logic*-Variablen *undefined* ist, diese Werte an den Ports aber nach Typ *bit* gewandelt werden, werden an dieser Stelle Warnings erzeugt, da diese Konvertierung nicht möglich ist. Nachdem alle Flipflops initialisiert sind, verschwindet dieses Problem.

Ab dem Zeitpunkt 900 ns wird alle 100 ns ein Ergebnis geliefert, das die Testbench abnimmt und eine entsprechende Notiz (Note) erzeugt, in der vermerkt wird, ob das Datum zu dem geforderten Index richtig ist (Vergleich mit expected response).

Zum Zeitpunkt 2269 ns wird ein Fehlersignal und somit ein Fehler aktiviert und 89 ns weitersimuliert. Zum Zeitpunkt 2269+89 ns wird der Fehler wieder deaktiviert. Im folgenden ist zu erkennen, daß zum Zeitpunkt 2700 ns der Testbench ein Ergebnis präsentiert wird, daß nicht zum Index 18 paßt.

Die anderen Meldungen, daß in X_OUT einer gewissen Stufe ein fehlerbehafteter Operand ausgegeben wurde, wurden nur generiert, um zu sehen, ob die Bit-/CORDIC-Algorithmen von selbst Fehler korrigieren. Würden sie dies tun, würden diese Meldungen spätestens, wenn ein Ergebnis die Pipeline verläßt, verschwinden. Dieses Verhalten wurde zwar nie beobachtet, ist

aber denkbar.

Wie schon dargestellt, ist der Platzbedarf der vom Simulator erzeugten Trace-Files zu groß und kann aufgrund der gigantischen Größe auch nicht interaktiv analysiert werden. Deshalb wurde aus den Trace-Files von Abbildung B.1 mittels eines Parsers ein neues Format generiert, das wesentlich leichter zu handhaben ist. Dafür wurde jeder Trace eines Fehlerinjektionsexperiments mit dem fehlerlosen Lauf verglichen und nur noch gespeichert, welche Auswirkungen ein injizierter Fehler hat. Durch Zerstörung des Index bei einer Berechnung kommt es z.B. zum Fehlen eines Wertes, während gleichzeitig ein anderes Ergebnis mit dem fehlerhaften Index überschrieben wird. Das neue, bereits mit dem korrekten Simulationslauf verglichene Datenformat ist in Abbildung B.2 dargestellt. Durch diese Kompression wurden die Traces etwa um den Faktor 1000 kleiner. Die resultierenden Datenfiles haben eine Größe von 10 bis 25 Kilobyte. Der hohe Kompressionsfaktor ist hauptsächlich auf die vielen Warnings zu Beginn eines Experiments zurückzuführen. Da diese irrelevanten Daten im neuen Format nicht mehr erscheinen, konnte dieser hohe Faktor erreicht werden.

Aus dem Datenformat von Abbildung B.2 können die verschiedenen Fehlverhalten der Systeme in wenigen Sekunden ermittelt werden.

```
restart
run 2269
.
.

** Warning: UN-CONVERTIBLE VALUE
   Time: 4 ns  Iteration: 0  Instance: /tb_net_simple_node_par/network_simple_par
** Warning: UN-CONVERTIBLE VALUE
   Time: 4 ns  Iteration: 0  Instance: /tb_net_simple_node_par/network_simple_par
** Warning: UN-CONVERTIBLE VALUE
   Time: 4 ns  Iteration: 0  Instance: /tb_net_simple_node_par/network_simple_par
** Note: GOT RESULT FOR PATTERN 0
   Time: 900 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 1
   Time: 1 us  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 2
   Time: 1100 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 3
   Time: 1200 ns  Iteration: 3  Instance: /tb_net_simple_node_par
.
.
** Note: GOT RESULT FOR PATTERN 12
   Time: 2100 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 13
   Time: 2200 ns  Iteration: 3  Instance: /tb_net_simple_node_par
force -freeze /tb_net_simple_node_par/node_par_2/x_adder/gen_stage_2_bit_7/u19/s0_z t
run 89
** Note: GOT RESULT FOR PATTERN 14
   Time: 2300 ns  Iteration: 3  Instance: /tb_net_simple_node_par
force -freeze /tb_net_simple_node_par/node_par_2/x_adder/gen_stage_2_bit_7/u19/s0_z f
se 0
run 4962
** Note: GOT RESULT FOR PATTERN 15
   Time: 2400 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: X_OUT STAGE 2 PATTERN 18 FAULTY COUNTER=23
   Time: 2400 ns  Iteration: 4  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 16
   Time: 2500 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: X_OUT STAGE 3 PATTERN 18 FAULTY COUNTER=24
   Time: 2500 ns  Iteration: 4  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 17
   Time: 2600 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: X_OUT STAGE 4 PATTERN 18 FAULTY COUNTER=25
   Time: 2600 ns  Iteration: 4  Instance: /tb_net_simple_node_par
```

Abbildung B.1: Ein Tracefile des Modeltech Simulator (Fortsetzung nächste Seite)

```
** Note: GOT RESULT FOR PATTERN 18
   Time: 2700 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: FAULTY RESULT FOR PATTERN 18
   Time: 2700 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: X_OUT STAGE 5 PATTERN 18 FAULTY COUNTER=26
   Time: 2700 ns  Iteration: 4  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 19
   Time: 2800 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 20
   Time: 2900 ns  Iteration: 3  Instance: /tb_net_simple_node_par
** Note: GOT RESULT FOR PATTERN 21
   Time: 3 us  Iteration: 3  Instance: /tb_net_simple_node_par
```

Forsetzung von Abbildung B.1

```
fault #7: /tb_net_macrorb_node_par/node_par_a_5/u107/s0_c 5577 150
        no effect
fault #8: /tb_net_macrorb_node_par/node_par_a_5/u150/s1_b 6928 85
        no effect
fault #9: /tb_net_macrorb_node_par/node_par_b_4/y_adder/u20/s1_a 1853 19
        no effect
fault #10: /tb_net_macrorb_node_par/network_macrorb_par/index_1_reg_2_label/s
        pattern 50: MISSING
        pattern 54: FAULTY
        pattern 54: got at 6000 (golden: 6400)
fault #11: /tb_net_macrorb_node_par/node_par_b_0/u143/s0_z 4611 150
        no effect
fault #12: /tb_net_macrorb_node_par/node_par_a_2/u107/s1_c 585 233
        no effect
```

Abbildung B.2: Komprimierte Simulationsdaten

Anhang C

Fehlerinjektor

In Abschnitt 5.1.3 wurde beschrieben, daß die langen Simulationszeiten von VERIFY umgangen wurden, indem der kommerzielle VHDL Simulator von Modeltech eingesetzt wurde. Da dieser Simulator nicht die Möglichkeit bietet, Fehler zu injizieren, wurde ein Hybridverfahren entwickelt, das die Vorteile beider Simulatoren vereint.

```
1378 216 /node_par_c_5/y_adder/gen_stage_2_bit_7/u13/s0_c[0001a0a1]
5748 45 /node_par_b_0/u155/s1_z[00004848]
3629 36 /network_tmr_par/u207/s0_c[0000139b]
6120 167 /node_par_a_4/z_adder/gen_stage_0_bit_3/u4/s0_a[00013a48]
```

Abbildung C.1: Ausgabe der Fehlerinjektionsdaten von VERIFY

VERIFY wurde eingesetzt, um die Fehlerinjektion zu arrangieren, während die eigentliche Simulation mit dem Modeltech Simulator durchgeführt wurde. Dazu wurde mit VERIFY eine Liste von 10000 zu injizierenden Fehlern erzeugt, in der in jeder Zeile die Daten für ein Fehlerinjektionsexperiment enthalten sind. In Abbildung C.1 ist ein Teil der Ausgabe einer Fehlerliste für das bitparallele TMR-Netz dargestellt. Es existiert für jede Implementierungsvariante eine solche Liste mit 10000 Einträgen.

In der ersten Spalte ist der Zeitpunkt, in Nanosekunden, zu dem ein Fehlersignal aktiviert werden soll, vermerkt. In der zweiten Spalte ist die Dauer der Aktivierung, ebenfalls in Nanosekunden, vermerkt. Die letzte Spalte enthält den Pfad durch das Design zum Fehlersignal eines Gatters. Die in eckigen Klammern angegebene Zahl ist ein interner Pointer von VERIFY zu dem Fehlersignal, der hier irrelevant ist.

Aus dieser Fehlerliste wurde mittels eines Parsers, geschrieben in C, ein Batchscript für den Modeltech VHDL-Simulator erzeugt, woraus der zu Abbildung C.1 passende Teil in Abbildung C.2 dargestellt ist. Aus einer Zeile in Abbildung C.1 wurden fünf Zeilen für das Batchscript generiert. Am Beispiel der ersten Zeile soll gezeigt werden, wie die Ermittlung für einen Trace unter Fehlerinjektion erwirkt wurde: Zunächst wird bis zum Zeitpunkt der Fehleraktivierung das Modell ohne injizierten Fehler simuliert (VHDL initialisiert Boolean-Variablen mit *false*), im Falle der ersten Zeile 1378 ns. Zu diesem Zeitpunkt wird das Fehlersignal auf *true* gesetzt

```
run 1378
force -freeze /tb_net_tmr_node_par/node_par_c_5/y_adder/gen_stage_2_bit_7/u13/s0_c tr
run 216
force -freeze /tb_net_tmr_node_par/node_par_c_5/y_adder/gen_stage_2_bit_7/u13/s0_c fa
se 0
run 5726
restart
run 5748
force -freeze /tb_net_tmr_node_par/node_par_b_0/u155/s1_z true 0
run 45
force -freeze /tb_net_tmr_node_par/node_par_b_0/u155/s1_z false 0
run 1527
restart
run 3629
force -freeze /tb_net_tmr_node_par/network_tmr_par/u207/s0_c true 0
run 36
force -freeze /tb_net_tmr_node_par/network_tmr_par/u207/s0_c false 0
run 3655
restart
run 6120
force -freeze /tb_net_tmr_node_par/node_par_a_4/z_adder/gen_stage_0_bit_3/u4/s0_a tru
run 167
force -freeze /tb_net_tmr_node_par/node_par_a_4/z_adder/gen_stage_0_bit_3/u4/s0_a fal
se 0
```

Abbildung C.2: Batchscript für den Modeltech Simulator

und somit der Fehler aktiviert. Mit diesem aktivierten Fehler wird das System dann die Fehlerdauer, 216 ns in dem Beispiel, weitersimuliert. Danach wird das Fehlersignal wieder auf *false* gesetzt und damit der Fehler deaktiviert. Danach wird noch bis zum Zeitpunkt 7320 simuliert. Zu diesem Zeitpunkt sind alle Stimuli durch das Netz spätestens bearbeitet. Danach wird der Simulator zurückgesetzt und die nächste Zeile und somit der nächste Fehler der Fehlerliste simuliert.

Da pro injizierten Fehler ein Tracefile der Größe von etwa 100 kByte erzeugt wird, war es nicht möglich, alle 10000 Fehler auf einmal innerhalb eines Batchscripts zu berechnen, da dies in einer Datenmenge von etwa 1 Terabyte Simulationsdaten enden würde, was auf hiesigen Systemen nicht gespeichert werden kann. Deshalb wurde die Simulation der 10000 Fehler in Paketen zu je 100 Fehlern durchgeführt, wobei direkt nach der Erzeugung eines 10 Megabyte Traces die Daten mit einem weiteren Verfahren komprimiert wurden. Pro Modell sind demnach 100 Batchfiles erzeugt worden, deren Inhalt in Abbildung C.2 dargestellt ist.

Anhang D

Detaillierte Strukturierung der Ergebnisse

Die in den Abschnitten 5.2.1-5.2.4 dargestellten Tabellen enthalten lediglich eine Aufsummierung aller Fehlerarten an jedem Ort des Systems. Für eine detailliertere Analyse sind jedoch detaillierte Angaben notwendig gewesen. Im folgenden wird dargestellt, aus welcher Form über die Tabellen hinausgehende Aussagen gefolgert wurden.

In Abbildung D.1 ist ein Teil der Statistik über die Experimente im bitparallelen, funktionalen Netz dargestellt. In den Tabellen 5.1-5.4 sind nur die Statistiken über alle möglichen Orte, im Knoten oder Netzwerk, und alle möglichen Fehler, Stuckat 0, Stuckat 1 und Bitflip, summiert angegeben. Diese Summe ist bei den Ausgabefiles in der obersten Zeile zu erkennen: 0 Blocking, 51 Missing, 814 Drifting, 9135 Correct und 0 Delayed, bei einer Anzahl von 10000 Fehlern, vergleiche auch Tabelle 5.1 die Zeile für die bitparallele Variante des funktionalen Netzes.

In den Klammern hinter der Anzahl der Fehler ist aufgeschlüsselt, wie viele von dieser Anzahl Stuckat 0, Stuckat 1 und Bitflipfehler sind. Betrachtet man wieder die erste Zeile, werden die 51 Missing-Fehlverhalten von 14 Stuckat 0, 16 Stuckat 1 und 21 Bitflipfehlern herausgefordert.

Der Fehlerort wird erkennbar, indem man das VDHL-Modell als einen Baum betrachtet, an dessen Wurzel die Testbench liegt, in der wiederum ein Netzwerk und seine Knoten liegt. Verzweigungen sind durch ’+’ Zeichen gekennzeichnet. Stufen höherer Hierarchie enthalten die Summe aller Teilbäume unterhalb dieser, weshalb in der obersten Zeile die gesamte Statistik zu erkennen ist.

In diesem Beispiel ist in der Wurzelkomponente *tb_net_simple_node_par* eine Komponente namens *network_simple_par* enthalten, in der alle 51 Fehler, die das Missingverhalten provozierten, enthalten sind. Für das Driftingfehlverhalten sind allerdings nur 19 von den 814 in dieser Komponente enthalten. Die restlichen 795 sind auf andere Komponenten verteilt, die hier nicht mehr dargestellt sind.

Mit Hilfe dieser Ausschlüsselung ist es möglich, genau herauszufinden welcher Fehler und welcher Fehlertyp an welchem Ort welches Fehlverhalten herausgefordert hat.

```
+-/tb_net_simple_node_par
| # number of injected faults | blocking | missing | drifting | cor-
rect | delayed
| 10000   0 (  0   0   0)  51 ( 14  16  21) 814 (395 386  33) 9135 (4589 4517  29)
|   +-/tb_net_simple_node_par/network_simple_par
|   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   | 10000   0 (  0   0   0)  51 ( 14  16  21)  19 (  6   8   5) 226 (103 120   3)
|   |   +-/tb_net_simple_node_par/network_simple_par/u53
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0   1
|   |   |   +-/tb_net_simple_node_par/network_simple_par/u53/s1_a
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0
|   |   +-/tb_net_simple_node_par/network_simple_par/b_reg_3_label
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  1   0
|   |   |   +-/tb_net_simple_node_par/network_simple_par/b_reg_3_label/s0_d
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  1
|   |   +-/tb_net_simple_node_par/network_simple_par/a_reg_0_label
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   1 (  0   0   1)   1 (  0   1
|   |   |   +-/tb_net_simple_node_par/network_simple_par/a_reg_0_label/s1_cp
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0
|   |   |   +-/tb_net_simple_node_par/network_simple_par/a_reg_0_label/flip
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   1 (  0   0   1)   0 (  0
|   |   +-/tb_net_simple_node_par/network_simple_par/valid_reg_1_label
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   2 (  0   2
|   |   |   +-/tb_net_simple_node_par/network_simple_par/valid_reg_1_label/s1_cp
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   2 (  0
```

Abbildung D.1: Detaillierte Fehlerstatistik (Forsetzung nächste Seite)

```
| # number of injected faults | blocking | missing | drifting | cor-
rect | delayed


|   |   +-/tb_net_simple_node_par/network_simple_par/u74
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0   1
|   |   |   +-/tb_net_simple_node_par/network_simple_par/u74/s1_c
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0
|   |   +-/tb_net_simple_node_par/network_simple_par/valid_reg_6_label
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
|   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   2 (  1   1
|   |   |   +-/tb_net_simple_node_par/network_simple_par/valid_reg_6_label/s0_d
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  1
|   |   |   +-/tb_net_simple_node_par/network_simple_par/valid_reg_6_label/s1_q
|   |   |   | # number of injected faults | blocking | mis-
sing | drifting | correct | delayed
|   |   |   | 10000   0 (  0   0   0)   0 (  0   0   0)   0 (  0   0   0)   1 (  0
|   |   +-/tb_net_simple_node_par/network_simple_par/index_1_reg_5_label
|   |   | # number of injected faults | blocking | missing | drif-
ting | correct | delayed
```

Fortsetzung von Abbildung D.1

Literaturverzeichnis

[1] Dietmar Fey, Bernd Kasche, Christian Burkert, and Oliver Tschäche. Specification for a reconfigurable optoelectronic VLSI processor suitable for digital signal processing. *Applied Optics*, 37(2):284–295, January 1998.

[2] K.K. Goswami and R.K. Iyer. Depend: A design environment for prediction and evaluation of system dependability. *Proceedings 9th Digital Avionics Systems Conference*, Oktober 1990.

[3] J. Güthoff and V. Sieh. Improving the efficiency of fault injection based dependability evaluation. *Proceedings 25th Symposium on Fault Tolerant Compunting (FTCS-25)*, pages 196–206, Juni 1995.

[4] A. Hein and K.K. Goswami. Combined performance and dependability evaluation with conjoint simulation. *Proceedings 7th European Simulation Symposium*, pages 365–369, Oktober 1995.

[5] Kai Hwang. *Computer Arithmetic: Principles, Architecture and Design*. John Wiley & Sons, 1979.

[6] J.A.Clark and D.K.Pradhan. React: A synthesis and evaluation tool for fault-tolerant multiprocessor architectures. *Proceedings Annual Reliability and Maintainability Symposium, IEEE Press*, pages 428–435, 1993.

[7] Bernd Kasche. Untersuchung geeigneter Algorithmen für Smart-Pixel Rechenwerke. Master's thesis, Friedrich Schiller Universität Jena, April 1995.

[8] S. Kumar, R.H. Klenke, and J.H. Aylor. Adept: A unified system level modeling design environment. *Proceedings 1st Annual RASSP Conference*, pages 114–123, August 1994.

[9] Edward J. McCluskey. *Logic design principles*. Englewood Cliffs, NJ Prentice-Hall, 1986.

[10] Dhiraj K. Pradhan. *Fault-Tolerant Computer System Design*. Prentice Hall PTR, 1995.

[11] R.H.Iyer and D.Rosetti. A measurement based model for workload-dependance of cpu-errors. *IEEE Transactions on Computers*, 35:511–519, Juni 1986.

[12] Volkmar Sieh. *Effiziente Erstellung und Auswertung von Rechnermodellen zur detaillierten Zuverlässigkeitsanalyse*. PhD thesis, Friedrich Alexander Universität Erlangen-Nürnberg, April 1998.

[13] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. Comparing different fault models using verify. *Dependable Computing and Fault-Tolerant Systems*, 11:63–80, January 1997.

[14] Volkmar Sieh, Oliver Tschäche, and Frank Balbach. Verify: Evaluation of reliability using vhdl models with integrated fault descriptions. *Extended Abstracts 8th European Workshop Dependable Computing (EWDC-8)*, 11:63–80, January 1997.

[15] Otto Spaniol. *Arithmetik in Rechenanlagen*. Teubner, 1987.

Lebenslauf

18.April 1969	Geboren in Erlangen
1975 – 1979	Besuch der Grundschule in Dechsendorf bei Erlangen
1979 – 1988	Besuch des Albert-Schweitzer-Gymnasiums in Erlangen
6/1988	Abitur
1988 – 1996	Studium der Elektro-Technik in Erlangen
2/1996	Diplom in Elektro-Technik der Friedrich-Alexander-Universität Erlangen
3/1996 – 8/2000	Angestellt am Lehrstuhl III des Instituts für Informatik der Universität Erlangen
10/2000 –	Tätig als freiberuflicher Berater im Bereich Informationstechnologie