Selbsttest mit Akkumulatoren

Dipl.-Ing. Frank Mayer

Selbsttest mit Akkumulatoren

Zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften von der Fakultät für Informatik der Universität Karlsruhe genehmigte Dissertation von Frank Mayer aus Ludwigshafen/Rhein.

Tag der mündlichen Prüfung:	10. Februar 2000
Erster Gutachter:	Prof. Dr.-Ing. D. Schmid
Zweiter Gutachter:	Prof. Dr. rer. nat. H. Schmeck

Diese Arbeit ist in elektronischer Form verfügbar.
http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=2000/informatik/3

Vorwort

Die vorliegende Arbeit entstand in den Jahren 1995-2000 im Rahmen eines von der DFG finanzierten Forschungsprojekts am Institut für Rechnerentwurf und Fehlertoleranz der Fakultät für Informatik an der Universität Karlsruhe.

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die an der Erstellung dieser Arbeit beteiligt waren. Mein Dank gilt vor allem meinem Hauptreferenten Herrn Prof. Dr.-Ing. D. Schmid für die Betreuung der Arbeit und für den großen Freiraum, den er mir während all der Jahre am Institut ließ. Ebenso bedanke ich mich bei Herrn Prof. Dr. rer. nat. H. Schmeck für die Übernahme des Korreferats und die sorgfältige Durchsicht meiner Arbeit.

Ganz besonders möchte ich mich bei meinem Kollegen Dr. Albrecht Ströle bedanken für dessen fachlichen Beistand und seine stoische Geduld bei unzähligen Gesprächen. Weiterhin bedanke ich mich bei all meinen Kolleginnen und Kollegen für die tolle Zeit am Institut, an die ich mich stets gerne erinnern werde. Nicht zuletzt danke ich meinen Eltern, ohne deren Unterstützung diese Arbeit nicht denkbar gewesen wäre.

Karlsruhe, im Februar 2000 *Frank Mayer*

Inhalt

1 Einleitung

1.1 Motivation

Der Entwurf hochintegrierter digitaler Schaltungen ist heute weitgehend automatisiert. Dadurch ist es möglich, komplexe Schaltkreise mit mehreren Millionen Transistoren in vergleichsweise kurzer Zeit zu entwickeln. Die Herstellung der durch die fortschreitende Miniaturisierung stetig feiner werdenden Chipstrukturen wird allerdings immer schwieriger. So können schon kleinste Staubpartikel, wie sie selbst in Reinsträumen nicht zu vermeiden sind, zu fehlerhaften Strukturen führen. Jeder Prozeßschritt ist ein Herd für Fehlerquellen. Durch die Vielzahl der benötigten Prozeßschritte lassen sich selbst bei optimierten Produktionsverhältnissen defekte Chips nicht vermeiden.

Defekte Chips sind nicht nur ein Sicherheitsrisiko, sondern verursachen auch Reparaturkosten. Ist ein Chip bereits in eine Baugruppe eingebaut oder wird sein Defekt sogar erst innerhalb eines großen Systems beim Kunden erkannt, steigen diese Kosten rapide an. Die Lieferbedingungen für Chips sehen daher meist einen hohen Qualitätsstandard vor, den der Chipproduzent einhalten muß. Um diesen Standard zu erreichen, wird jeder einzelne Chip nach der Produktion getestet, und fehlerhafte Chips werden ausgemustert. Die Anforderungen an einen solchen Test sind im wesentlichen zweifach. Defekte Chips sollen mit hoher Wahrscheinlichkeit erkannt werden, und der Test eines Chips soll zusätzlich in möglichst kurzer Zeit abgeschlossen sein, bei Massenprodukten bereits nach wenigen Sekunden.

1.1.1 Testautomaten

Eine Möglichkeit für den Test ist die Verwendung eines externen Testautomaten. Solche Automaten sind in der Lage, vorprogrammierte Testmuster an die Chipanschlüsse anzulegen und die resultierenden Testantworten an den Chipausgängen zu lesen und zu analysieren. Ein Testmuster kann dabei Defekte innerhalb der Schaltung aufspüren, indem es bei Anwesenheit bestimmter Defekte eine andere Testantwort erzeugt als im im fehlerfreien Fall. Durch Vergleich der tatsächlichen Testantworten mit den z.B. durch Simulation gewonnenen Antworten für den fehlerfreien Fall lassen sich danneinzelne Defekte diagnostizieren.

Mit zunehmender Komplexität der Chips wird es immer schwieriger, Defekte im Inneren der Schaltung über die vergleichsweise wenigen Chipeingänge zu aktivieren und ein Fehlverhalten bis an die Chipausgänge zu propagieren. Will man zudem Defekte entdecken, die sich nur bei Betriebsgeschwindigkeiten auswirken, muß der Testautomat mit modernster Technologie arbeiten, um Testmuster mit ausreichender Geschwindigkeit bereitstellen zu können. Gerade bei neuen, schnellen Chiptechnologien ist dies ein sehr teures Unterfangen.

1.1.2 Selbsttest

Eine Alternative zum Test mit einem externen Testautomaten ist der Selbsttest. Hierbei wird die benötigte Testhardware auf dem Chip selbst integriert. Bild 1.1 zeigt den schematischen Aufbau beim Selbsttest.

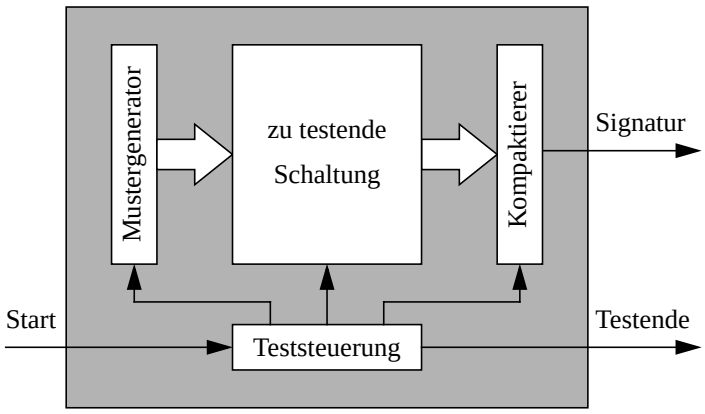


Bild 1.1: Schema eines Selbsttests

Im Testbetrieb erzeugt der Mustergenerator Testmuster, die an die Eingänge der zu testenden Schaltung geleitet werden. Da das Speichern der fehlerfreien Testantworten auf dem Chip vom Hardware-Aufwand her nicht zu vertreten ist, werden die Testantworten stattdessen meist zu einer Signatur kompaktiert. Die Signatur zeigt schließlich an, ob während des Tests ein Defekt erkannt worden ist oder nicht. Der Ablauf des Tests wird von der Teststeuerung kontrolliert. Auf ein Startsignal hin schaltet sie vom Normalbetrieb in den Testbetrieb um und initialisiert den Mustergenerator, den Kompaktierer und die zu testende Schaltung. Darauf startet sie die Mustergenerierung und die Kompaktierung und koordiniert den Transport der Testmuster zur Schaltung bzw. den der Testantworten zum Kompaktierer. Zuletzt gibt sie die Signatur aus und signalisiert das Testende.

Der Selbsttest bietet gegenüber dem externen Test viele Vorteile. Zum einen spart man sich den teuren Testautomaten. Ferner ist der Test bei Betriebsgeschwindigkeit kein Problem, da die Testhardware in derselben Technologie aufgebaut ist wie der Rest des Chips. Komplexe, schwer zu testende Schaltungen können im Testbetrieb in weniger komplexe, einfach zu testende Teilschaltungen partitioniert werden. Diese Teilschaltungen können dann unabhängig voneinander und je nach Anzahl der verwendeten Mustergeneratoren und Kompaktierer in einer oder in mehreren Testsitzungen nacheinander getestet werden. Die externe Testansteuerung über ein Start- und ein Endesignal ist denkbar einfach. Dadurch eignet sich der Selbsttest auch hervorragend für den hierarchischen Test und für die Diagnose ganzer Baugruppen und Systeme z.B. im Rahmen der „Boundary-Scan Norm“ [IEEE90]. Durch den Selbsttest ist es auch möglich, Chips im späteren Einsatz z.B. während Betriebspausen zu testen.

Den Vorteilen des Selbsttests stehen einige Nachteile gegenüber. Durch den Informationsverlust bei der Kompaktierung sind die Diagnosemöglichkeiten sehr beschränkt. Im allgemeinen läßt sich anhand der Signatur nur feststellen, ob ein Test fehlgeschlagen ist oder nicht. Die Ursache eines Scheiterns läßt sich jedoch nicht rekonstruieren. In den meisten Fällen, speziell beim Aussortieren defekter Chips nach der Produktion, ist diese Information jedoch völlig ausreichend.

Die Kompaktierung beinhaltet noch einen weiteren Nachteil. Mehrere fehlerhafte Testantworten können sich in ihrer Wirkung derart aufheben, so daß die Signatur am Testende derjenigen im fehlerfreien Fall entspricht, man spricht dann von Fehlermaskierung. Bei den für den Selbsttest verwendeten Kompaktierungsmethoden ist die Wahrscheinlichkeit einer solchen Maskierung jedoch meist vernachlässigbar gering.

Ein schwererwiegender Nachteil sind Verzögerungen der Signallaufzeiten durch die zusätzlichen Gatter, die mit der Testhardware in die Datenpfade und Steuerleitungen einer Schaltung eingebaut werden. Liegen solche Gatter im kritischen Pfad, sinkt die maximal mögliche Betriebsgeschwindigkeit. Gerade bei Hochgeschwindigkeitsschaltungen läßt sich das in der Regel nicht vermeiden, da solche Schaltungen entwurfsbedingt eine Vielzahl kritischer Pfade enthalten.

Der wohl gravierendste Nachteil beim Selbsttest ist der zusätzliche Flächenbedarf für die Testhardware. Durch die vergrößerte Chipfläche steigt die Wahrscheinlichkeit, daß ein Chip nach der Produktion defekt ist, und damit der Ausschuß. Dies schlägt sich unmittelbar in den Herstellungskosten nieder. Ein Beispiel hierfür ist in [Thom96] beschrieben. Steigt die Fläche eines Mikroprozessor-

chips durch zusätzliche Testhilfsmittel um 15 %, müssen 47 % mehr Wafer produziert werden, um die gleiche Anzahl funktionsfähiger Chips ausliefern zu können.

Um den zusätzlichen Flächenbedarf zu mindern, werden schaltungseigene Register zu sogenannten Testregistern ausgebaut. Ein bekanntes Beispiel hierfür ist der „Built-In Logic Block Observer“, kurz BILBO [KöMZ79]. Dieses multifunktionale Testregister kann in vier verschiedenen Betriebsmodi arbeiten: außer als normales Register noch als Mustergenerator, Kompaktierer und als Schieberegister. Es benötigt allerdings immer noch etwa den doppelten Flächenbedarf eines einfachen Registers und enthält zusätzliche Gatter in den Datenleitungen, die die Signallaufzeiten vergrößern [WaMc86a, OhWM87].

1.1.3 Akkumulatoren

Erst in den letzten Jahren wurden Akkumulatoren als Alternative zu herkömmlichen Testregistern untersucht [RaTy93a/b, GuRT94, Strö95a/b, Strö96a/b]. Den schematischen Aufbau eines Akkumulators zeigt Bild 1.2.

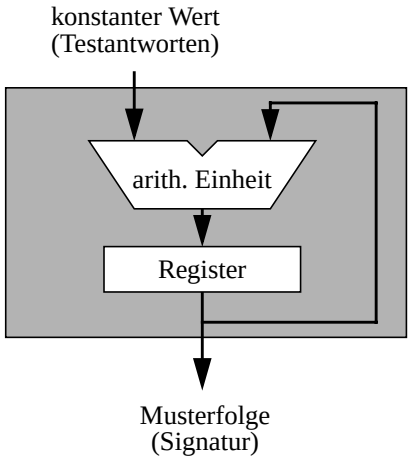


Bild 1.2: Akkumulator

Der Inhalt eines Registers wird auf einen der beiden Eingänge einer arithmetischen Einheit, z.B. eines Addierers, geleitet. Dort wird er mit dem Wert am anderen Eingang zu einem neuen Registerwert verknüpft. Ist der Wert am anderen Eingang konstant, arbeitet der Akkumulator als Mustergenerator. Werden an den anderen Eingang Testantworten geleitet, werden sie vom Akkumulator zu einer Signatur kompaktiert.

Es konnte gezeigt werden, daß Akkumulatoren in den Punkten Fehlererfassung und Fehlermaskierung ebenso für den Selbsttest geeignet sind wie Testregister. In vielen Schaltungen, besonders in

Datenpfaden für die digitale Signalverarbeitung, sind die erforderlichen Bestandteile von Akkumulatoren bereits vorhanden und können allein durch geeignetes Ansteuern zu Akkumulatoren konfiguriert werden. In diesen Fällen kann die sonst nötige Testhardware eingespart werden, so daß zusätzlicher Flächenbedarf und erhöhte Signallaufzeiten wie bei Testregistern entfallen.

Akkumulatoren bieten offensichtlich Vorteile gegenüber Testregistern. Allerdings befaßte sich die Forschung bisher überwiegend mit der prinzipiellen Eignung von Akkumulatoren für den Selbsttest. Im Gegensatz zu Testregistern, deren Einsatz mittlerweile von vielen Entwurfswerkzeugen unterstützt wird, werden die Potentiale von Akkumulatoren während der verschiedenen Entwurfsphasen noch kaum genutzt.

1.2 Ziel und Gliederung der Arbeit

Ziel der vorliegenden Arbeit ist es, Akkumulatoren beim automatischen Einbau eines Selbsttests in einen Chip zu berücksichtigen und ihre Möglichkeiten dabei voll auszunutzen.

Es wird ein Verfahren vorgestellt, das neben Testregistern auch Akkumulatoren für den Einbau eines Selbsttests auf *Register-Transfer-Ebene* (s. Abschnitt 2.1.2) verwendet. Das Verfahren erzeugt zunächst alle Akkumulatorstrukturen, die sich aus den vorhandenen Schaltungsmodulen durch gezieltes Ansteuern konfigurieren lassen. Diese Akkumulatoren werden zusammen mit den Registern der Schaltung als potentielle Mustergeneratoren bzw. Kompaktierer angesehen. Mithilfe dieser Mustergeneratoren und Kompaktierer wird die Schaltung in Teilschaltungen zerlegt, die - wiederum durch geeignetes Ansteuern - ein oder mehrere ihrer Module selbständig testen können. Teilschaltungen, die parallel arbeiten können, werden zu größeren Teilschaltungen vereinigt. Von der Vielzahl der möglichen Teilschaltungen werden schließlich diejenigen ausgewählt, die zusammen alle Module der Schaltung in möglichst kurzer Zeit testen können, und zwar bei minimalem Hardware-Mehraufwand für Testregister. Zuletzt wird für jede ausgewählte Teilschaltung die Ansteuerung so optimiert, daß die Teststeuerung möglichst einfach und/oder die Testzeit möglichst kurz wird. Das Verfahren zeichnet sich nicht nur durch den Einsatz von Akkumulatoren aus, es hebt auch viele Einschränkungen anderer Verfahren auf, so daß auch ohne Akkumulatoren oft noch Testregister eingespart werden.

Ein zweites Verfahren zielt auf die Minimierung der Dauer einer Testsitzung. Der konstante Eingabewert und der Startwert des Registers eines akkumulatorbasierten Mustergenerators werden

gezielt an die zu testende Teilschaltung angepaßt. Die resultierende Musterfolge ist in der Lage, alle modellierten kombinatorischen Fehler mit deutlich weniger Mustern und damit in kürzerer Testzeit zu überprüfen, als es bei zufällig gewählten Eingabe- und Startwerten möglich wäre, selbst wenn mithilfe von Fehlersimulationen von vielen solcher Zufallswerte nur die besten ausgewählt werden.

Teile beider Verfahren wurden bereits auf internationalen Konferenzen [MaSt97c, MaSt98b, MaSt00] und nationalen Workshops [MaSt97a/b, MaSt98a, MaSt99] veröffentlicht.

Die Arbeit ist folgendermaßen gegliedert: Im nächsten Kapitel werden Grundlagen des Selbsttests erläutert. Ebenso wird ein Abriß des Standes der Forschung auf dem Gebiet des Entwurfs selbsttestbarer Schaltungen gezeichnet. Dabei werden die für den Selbsttest spezifischen Probleme vorgestellt und, sofern für das weitere Verständnis der Arbeit notwendig, bekannte Lösungsansätze umrissen. In Kapitel 3 bzw. 4 werden das Konfigurationsverfahren für den Selbsttest respektive das Optimierungsverfahren für kurze Testmusterfolgen vorgestellt und die Ergebnisse diskutiert. Die Arbeit schließt mit einer Zusammenfassung.

2 Grundlagen und Stand der Technik

Dieses Kapitel gibt eine kurze Einführung in den Selbsttest sowie in Verfahren, die den Selbsttest während der verschiedenen Phasen des automatischen Schaltungsentwurfs berücksichtigen. Die Grundlagen umfassen den klassischen Schaltungsentwurf, soweit er für das weitere Verständnis dieser Arbeit notwendig ist, Klassen von Selbsttestkonfigurationen sowie den Aufbau und die wesentlichen Eigenschaften von Mustergeneratoren und Kompaktierern auf der Basis von Testregistern bzw. Akkumulatoren. Es folgen Methoden zur Anpassung eines Mustergenerators an eine zu testende Schaltung. Das Kapitel schließt mit bekannten Verfahren zur Berücksichtigung und zum Einbau der verschiedenen Selbsttestkonfigurationen während der Schaltungssynthese.

2.1 Automatische Schaltungssynthese

Bild 2.1 zeigt die Entwurfsphasen bei der automatischen Schaltungssynthese. Ausgangspunkt ist eine algorithmische Verhaltensbeschreibung der Schaltungsfunktion. Diese wird in mehreren Syntheseschritten zunächst zu einer Strukturbeschreibung auf Register-Transfer-Ebene, dann auf Gatterebene und schließlich zu einer Layout-Beschreibung verfeinert.

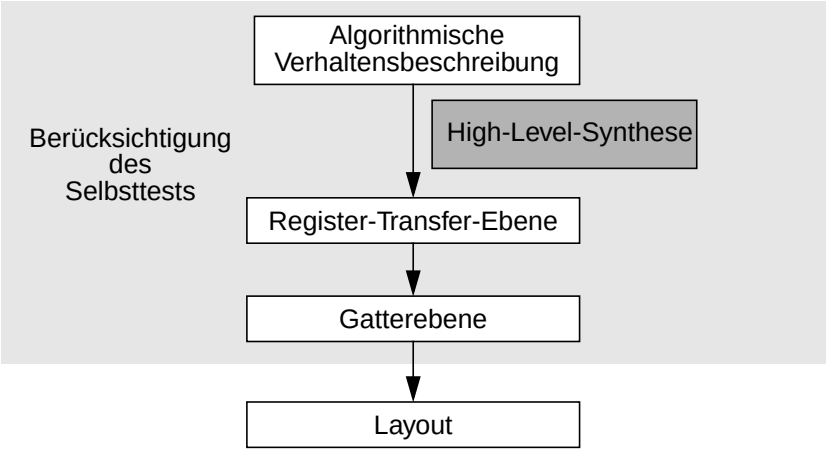


Bild 2.1: Entwurfsphasen der automatischen Schaltungssynthese

Im folgenden wird nur auf diejenigen Entwurfsebenen und Syntheseschritte näher eingegangen, bei denen die bekannten Verfahren für den Selbsttest ansetzen. Dies sind die Beschreibungen auf algo-

rithmischer Ebene, Register-Transfer- und Gatterebene sowie die Synthese einer Register-Transfer-Struktur aus einer algorithmischen Verhaltensbeschreibung – die sogenannte „High-Level-Synthese“.

2.1.1 Algorithmische Ebene

Ausgangspunkt der Schaltungssynthese ist hier eine algorithmische Beschreibung der Schaltungsfunktion mithilfe einer Programmiersprache wie z.B. C [KeRi83] oder VHDL [IEEE87]. Ein solches Programm verknüpft üblicherweise Schritt für Schritt ganzzahlige oder reellwertige Konstanten bzw. Variablen mit den typischen arithmetischen, relationalen und logischen Operatoren und enthält meist noch Verzweigungskonstrukte (if ... then ... else, switch ... case) und Schleifen (while, repeat ... until, for). Bild 2.2 zeigt einen Ausschnitt eines C-Programms, das die Quadratwurzel, $y = \sqrt{x}$, mithilfe eines Iterationsverfahrens mit einer maximalen Abweichung des Quadrats von 0.001 bestimmt [BrSe87].

```
. . .
y = 0.2 + 0.9 * x;
while (|y*y - x| > 0.001) {
    y = 0.5 * (y + x/y);
}
. . .
```

Bild 2.2: Algorithmus zur Berechnung der Quadratwurzel

2.1.2 Register-Transfer-Ebene

Auf Register-Transfer-Ebene wird eine Schaltung in ein Steuerwerk und ein Operationswerk unterteilt (Bild 2.3). Das Operationswerk besteht aus Standardmodulen, die über Leitungsbündel miteinander verbunden sind. Standardmodule sind dabei Register, Multiplexer, Funktionseinheiten wie z.B. Addierer, Subtrahierer, Multiplizierer, Komparatoren oder ganze arithmetisch-logische Einheiten, sogenannte ALUs (Arithmetic Logic Units). Das Operationswerk ist in der Lage, die Variablen der Verhaltensbeschreibung in Registern zu speichern und über Funktionseinheiten entsprechend zu verknüpfen.

Das Steuerwerk steuert dabei den Datenfluß und sorgt für einen korrekten Ablauf, indem es die Ladesignale von Registern, die Auswahlsignale von Multiplexern und eventuelle Steuersignale

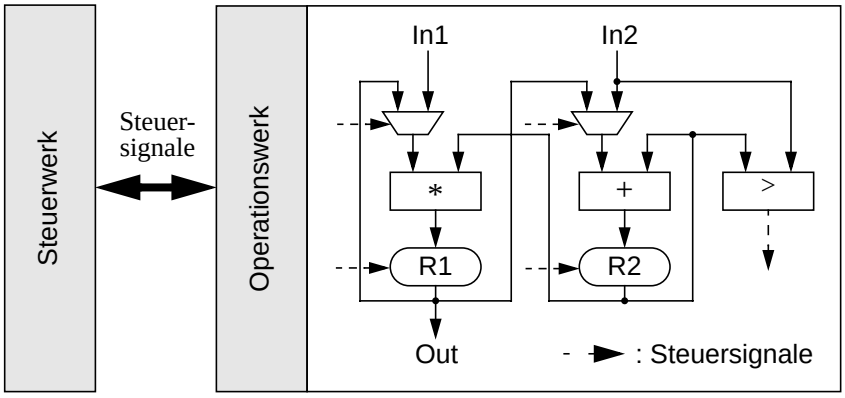


Bild 2.3: Schaltungsbeschreibung auf Register-Transfer-Ebene

anderer Module geeignet ansteuert und auf Steuersignale vom Operationswerk reagiert. Das Steuerwerk wird auf Register-Transfer-Ebene durch sein Verhalten beschrieben, meist als Zustandsübergangstabelle eines endlichen Automaten [McCa92].

2.1.3 Gatterebene

Auf Gatterebene wird eine Schaltung als Netzwerk von einzelnen Flipflops und logischen Gattern wie UND, ODER usw. beschrieben, die über einzelne Signalleitungen miteinander verbunden sind (Bild 2.4).

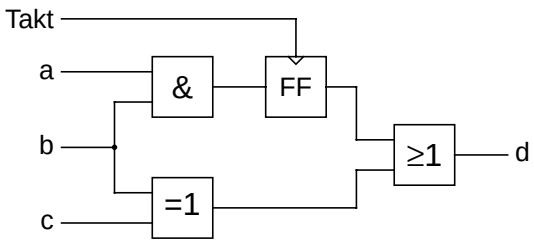


Bild 2.4: Netzwerk auf Gatterebene

2.1.4 High-Level-Synthese

Ziel der High-Level-Synthese ist es, eine möglichst optimale Register-Transfer-Struktur aus der algorithmischen Verhaltensbeschreibung einer Schaltung zu erzeugen. Optimierungsziele sind meist minimaler Hardware-Aufwand und/oder maximaler Datendurchsatz. Die High-Level-Syn-

these gliedert sich in einzelne Syntheseschritte, die im folgenden kurz beschrieben werden sollen. Ausführliche Einführungen finden sich z.B. in [PaKn89] und [McPC90].

Die einzelnen Aufgabenbereiche sind in Bild 2.5 dargestellt.

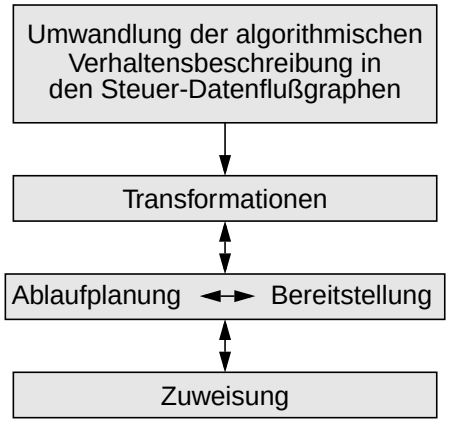


Bild 2.5: Syntheseschritte der High-Level-Synthese

Der erste Schritt bei der High-Level-Synthese ist die Umsetzung der algorithmischen Verhaltensbeschreibung in eine interne Darstellung des Steuer- und Datenflusses z.B. als *Steuer-Datenfluß-graph*. Die Knoten des Graphen stellen dabei die Operationen dar, die Kanten entsprechen den in der Verhaltensbeschreibung angegebenen Variablen bzw. zusätzlichen internen Variablen zur Repräsentierung von Zwischenergebnissen. Oft wird Kanten mit gemeinsamem Ursprung dieselbe Variable zugeordnet. Bild 2.6 zeigt den Datenflußgraph einer einfachen Verhaltensbeschreibung ohne Verzweigungskonstrukte und damit ohne Steuergraph. Die Beschriftung der Kanten zwischen den Operationen mit internen Variablen wurde der Übersichtlichkeit halber weggelassen.

D = A∗C + B∗C;

E = (A + B) ∗ (C + A − B);

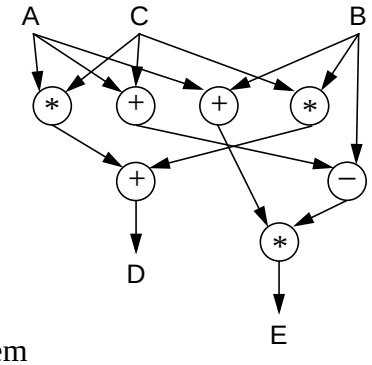


Bild 2.6: Verhaltensbeschreibung mit zugehörigem Datenflußgraph

Im nächsten Schritt wird der Steuer-Datenflußgraph mithilfe von funktionserhaltenden *Transformationen* vereinfacht. Bei der *Ablaufplanung* werden die Operationen des Graphen einzelnen Steu-

erschritten bzw. Taktzyklen zugewiesen. Die sogenannte *Bereitstellung* legt fest, welche und wieviele Funktionseinheiten verfügbar sind, und ebenso, wieviele Register und Steuerschritte verwendet werden dürfen. Bereitstellung und Ablaufplanung können in beliebiger Reihenfolge, auch gleichzeitig ausgeführt werden. Viele Systeme koppeln beide Schritte, um bessere Ergebnisse zu erzielen [GiKn84, PaKn87, Camp88]. Während der *Zuweisung* werden die verschiedenen Variablen und Operationen einzelnen Registern bzw. Funktionseinheiten zugewiesen, diese schließlich durch Leitungsbündel, Busse und Multiplexer miteinander verbunden.

Tatsächlich beeinflussen sich alle Syntheseschritte gegenseitig und müßten gleichzeitig betrachtet werden, um ein optimales Ergebnis zu erhalten. Die bekannten High-Level-Synthese-Werkzeuge bearbeiten hingegen mehrere Bereiche separat und verwenden Heuristiken, um den Rechenaufwand akzeptabel zu halten [PaKn89, RoCa85, DRSC86, TSEN88].

Im folgenden wird anhand des Beispiels aus Bild 2.6 der Einfluß der einzelnen Syntheseschritte auf die resultierende Register-Transfer-Beschreibung demonstriert.

2.1.4.1 Transformationen

Eine algorithmische Verhaltensbeschreibung ist i.a. für Menschen leicht lesbar geschrieben, jedoch nicht zur direkten Umsetzung in Hardware geeignet. Es ist daher wünschenswert, die ursprüngliche Beschreibung zu optimieren. In der Regel werden bereits vom Compilerbau bekannte Transformationen angewendet wie Konstantenpropagierung, Löschen von wirkungslosen Anweisungen, Ausrollen von Schleifen usw. Andere Transformationen sind mehr hardware-spezifisch. Beispielsweise kann die Addition um 1 durch eine Inkrementoperation ersetzt werden, oder Multiplikationen mit Zweierpotenzen werden durch Schiebeoperationen realisiert.

```
D = C * (A + B);
E = C * (A + B)  +  (A + B) * (A - B);
```

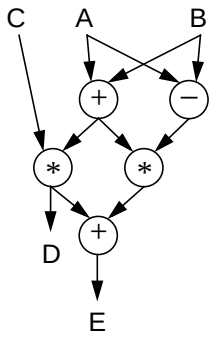


Bild 2.7: Vereinfachter Graph nach Transformationen

Bild 2.7 zeigt die Verhaltensbeschreibung von Bild 2.6 nach zweimaliger Anwendung des Distributivgesetzes. Der Graph wurde im Vergleich zum ursprünglichen Graphen durch die Transformation deutlich vereinfacht, ohne daß sich die Schaltungsfunktion geändert hat. Zusätzlich hat sich die Zahl der Operationen verringert.

2.1.4.2 Ablaufplanung und Bereitstellung

Das Resultat der High-Level-Synthese soll eine synchrone Schaltung sein, d.h. während eines Taktzyklus werden eine oder mehrere Operationen parallel ausgeführt und die Ergebnisse zu Beginn des nächsten Zyklus in Registern zwischengespeichert. Ziel der Ablaufplanung ist es, die Operationen des Steuer-Datenflußgraphen einzelnen Taktzyklen bzw. Steuerschritten so zuzuteilen, daß unter Beibehaltung der Datenabhängigkeiten eine möglichst kostengünstige Schaltung entsteht. Bei den Kosten werden normalerweise der Hardware-Aufwand und die Berechnungsgeschwindigkeit gegeneinander abgewogen. Zwei Operationen, die parallel im selben Steuerschritt durchgeführt werden sollen, benötigen jeweils eine eigene Funktionseinheit. Führt man sie in verschiedenen Steuerschritten aus, können sie sich eine Funktionseinheit teilen, sofern diese in der Lage ist, beide Operationstypen auszuführen. Im letzteren Fall kann eine Funktionseinheit eingespart werden auf Kosten zusätzlicher Steuerschritte, d.h. zu Lasten der Berechnungsgeschwindigkeit.

Wieviele Operationen parallel ausgeführt werden können, hängt von der Anzahl der jeweils verfügbaren Funktionseinheiten ab, die bei der Bereitstellung festgelegt wird. Bei der Bereitstellung wird ebenfalls die Anzahl der Steuerschritte vorgegeben. Bei den gängigen Verfahren werden diese Zahlen i.a. nicht fest vorgegeben, sondern vielmehr automatisch in gewissem Rahmen variiert, um eine optimale Lösung zu erhalten.

Bild 2.8 zeigt zwei verschiedene Ablaufpläne für den Steuer-Datenflußgraphen aus Bild 2.7. Der erste Ablaufplan benötigt drei Steuerschritte. Unter der Voraussetzung, daß jede Funktionseinheit nur je einen einzigen Operationstyp ausführen kann, benötigt dieser Ablauf neben einem Addierer und einem Subtrahierer noch zwei Multiplizierer, da die beiden Multiplikationen im selben Steuerschritt durchgeführt werden. Im Gegensatz dazu kommt der zweite Ablaufplan mit nur einem Multiplizierer aus, da alle Operationen verschiedenen Steuerschritten zugeteilt sind. Allerdings braucht er zwei zusätzliche Steuerschritte.

Anhand des Ablaufplans und des Steuer-Datenflußgraphen läßt sich auch der Mindestbedarf an Registern ablesen. Tatsächlich steht jede Kante im Graphen, die eine Steuerschrittgrenze – eine

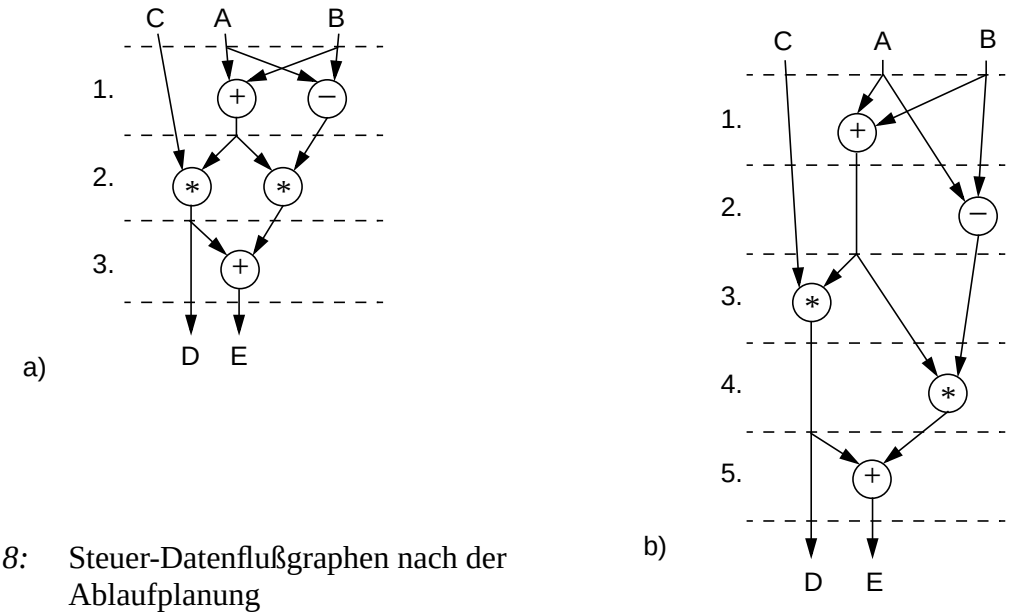


Bild 2.8: Steuer-Datenflußgraphen nach der Ablaufplanung

gestrichelte Linie – schneidet, für eine Variable, die in einem Register gespeichert werden muß. Somit benötigt der Graph mit den fünf Steuerschritten mindestens vier Register, da zu Beginn des zweiten Steuerschritts neben den Werten der Variablen A, B und C noch das Ergebnis der Addition als zusätzliche Variable gespeichert werden muß. Beim anderen Graphen genügen dagegen schon drei Register. Auch die Zahl der Register wird bei der Bereitstellung bestimmt und bei den Hardware-Kosten berücksichtigt.

2.1.4.3 Zuweisung

Bei der Zuweisung wird jede Operation einer Funktionseinheit und jede zu speichernde Variable einem Register zugewiesen. Dabei können Operationen aus verschiedenen Steuerschritten derselben Funktionseinheit zugeordnet werden, falls diese die erforderlichen Operationstypen ausführen kann. Ebenso können Variablen, die nicht gleichzeitig aktiv sind, vom selben Register gespeichert werden. Zuletzt werden Leitungsbündel hinzugefügt, wobei jede Kante im Steuer-Datenflußgraphen einem Leitungsbündel zugeordnet wird, das die entsprechenden Ein- und Ausgänge der beteiligten Funktionseinheiten und Register miteinander verbindet. Müssen dabei mehrere Ausgänge mit demselben Eingang verbunden werden, ist ein zusätzlicher Multiplexer bzw. ein Bus erforderlich. Das Ergebnis ist die Strukturbeschreibung des Operationswerkes auf Register-Transfer-Ebene. Durch die vielfältigen Zuweisungsmöglichkeiten gibt es auch bei bekannter Anzahl von Funktionseinheiten und Registern noch einen großen Spielraum für verschiedene Verbindungsaufbauten mit unterschiedlichem Flächenbedarf.

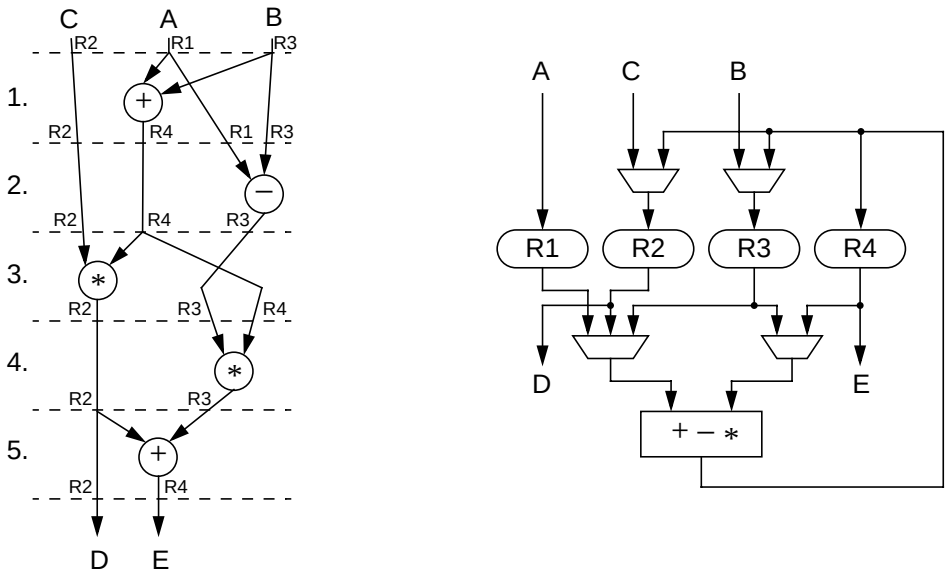


Bild 2.9: Steuer-Datenflußgraph nach Ablaufplanung und Zuweisung sowie zugehöriges Operationswerk

Bild 2.9 zeigt nocheinmal den Graphen aus Bild 2.8b und ein mögliches Operationswerk. Zur Verdeutlichung wurden die Kanten mit den Namen der Register beschriftet, die die verschiedenen Zwischenwerte im jeweiligen Steuerschritt speichern. Außerdem wurden bei der Multiplikation im vierten Steuerschritt die Eingänge vertauscht. Als Funktionseinheit dient eine komplexe Einheit, die alle erforderlichen drei Operationstypen ausführen kann. Die Lade- und Steuersignale wurden der Übersicht halber weggelassen.

Nachdem die Ablaufplanung und die Zuweisungen mit dem Verbindungsaufbau vollzogen sind, läßt sich anhand der gewonnenen Informationen die Ansteuerung für die Lade- bzw. Steuersignale der Register, Multiplexer, Bustreiber und eventueller arithmetisch-logischer Funktionseinheiten ableiten. Anhand dieser Ansteuerungsbeschreibung kann das Steuerwerk synthetisiert werden.

Betrachtet man das Operationswerk aus Bild 2.9, so erkennt man, daß durch das Zusammenlegen mehrerer Operationen bzw. Variablen auf eine Funktionseinheit bzw. auf ein Register Zyklen entstanden sind, die im Steuer-Datenflußgraphen noch nicht enthalten waren. Normalerweise enthalten bereits die Steuer-Datenflußgraphen Zyklen, die im Operationswerk widergespiegelt werden. Zyklen spielen bei den Entwurfsverfahren unter Berücksichtigung des Selbsttests eine große Rolle, und zwar abhängig von der gewählten Selbsttestst rategie. Die verschiedenen Selbsttestst rategien sollen im folgenden beschrieben werden.

2.2 Selbstteststrategien

Die heutigen Schaltungen sind i.a. zu komplex, als daß sie allein über die Schaltungseingänge und -ausgänge in akzeptabler Zeit – bei Massenprodukten wenige Sekunden – ausreichend viele Fehler testen könnten, d.h. oft über 95% der betrachteten Fehler. Schaltungen, die in akzeptabler Zeit nicht ausreichend getestet werden können, nennt man schwer testbar. Um den Test zu erleichtern, untergliedert man solche Schaltungen in kleinere, leichter testbare Teilschaltungen, die unabhängig voneinander überprüft werden. Dabei unterscheidet man zwei Vorgehensweisen. Beim „Test pro Scan“-Schema werden die Testmuster seriell in mehreren Taktzyklen über ein Schieberegister, den sogenannten Prüfpfad, eingelesen und an die Eingänge der Teilschaltungen angelegt. Dann folgt der eigentliche Test mit dem jeweiligen Testmuster, wobei die Testantwort wieder vom Schieberegister aufgenommen wird. Anschließend wird die Testantwort seriell ausgelesen und gleichzeitig das nächste Testmuster eingelesen.

Beim „Test pro Takt“-Schema wird in jedem Taktzyklus ein neues Testmuster direkt an den Eingängen der Teilschaltungen erzeugt und parallel an diese angelegt. Die Testantwort wird im selben Takt am Ausgang parallel abgelesen und verarbeitet.

Neben der Art und Weise, wie Testmuster an Teilschaltungen angelegt werden, unterscheidet man auch die Art der erzeugten Muster. Beim Test mit deterministisch bestimmten Testmustern werden die Testmuster anhand der Schaltung und eines Fehlermodells berechnet. Der Mustergenerator erzeugt nur diese Muster. Im Gegensatz dazu steht der Test mit erschöpfenden bzw. pseudoerschöpfenden Mustermengen und zufälligen Mustern. Hier sind die Muster nicht an eine spezielle Schaltung oder ein Fehlermodell angepaßt.

Im folgenden werden die beiden Testschemata vorgestellt. Anschließend werden die verschiedenen Methoden der Mustererzeugung diskutiert.

2.2.1 Selbsttest mit „Test pro Scan“-Schema

Beim „Test pro Scan“-Schema werden die schaltungseigenen Flipflops zu sogenannten Scan-Flipflops erweitert. Die Erweiterung besteht im wesentlichen aus einem Multiplexer, über den der Eingang des Flipflops im Testbetrieb mit dem Ausgang eines anderen Scan-Flipflops verbunden ist. Die Scan-Flipflops bilden im Testbetrieb ein großes Schieberegister, den Prüfpfad, der die Schal-

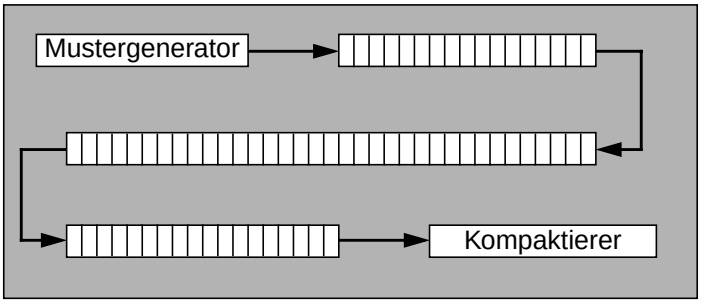


Bild 2.10: Selbsttest mit Prüfpfad

tung in viele Teilschaltungen segmentiert. Beim Selbsttest wird der Eingang des Prüfpfads von einem auf der Schaltung integrierten Mustergenerator seriell gespeist. Die Antworten werden ebenfalls seriell von einem integrierten Kompaktierer aufgenommen (Bild 2.10).

Die ersten Verfahren schalteten alle Flipflops einer Schaltung zu einem Prüfpfad zusammen [BaMc82, LeBl84]. Der Hardware-Mehraufwand wird in [LeBl84] mit 10 % bis15 % angegeben. Andere Verfahren weisen nur ausgewählte Flipflops einem partiellen Prüfpfad zu, um Hardware einzusparen. Dennoch kann die Anzahl der Scan-Flipflops sehr groß werden. Beim Ultra Sparc 1 Prozessor enthält der Prüfpfad beispielsweise 22 000 Scan-Flipflops [LEVI95]. Da die Testmuster und Testantworten seriell ein- bzw. ausgelesen werden, können Verzögerungsfehler, die sich nur bei Betriebsgeschwindigkeit auswirken, praktisch nicht erkannt werden. Außerdem können in akzeptabler Zeit nur vergleichsweise wenig Testmuster verwendet werden.

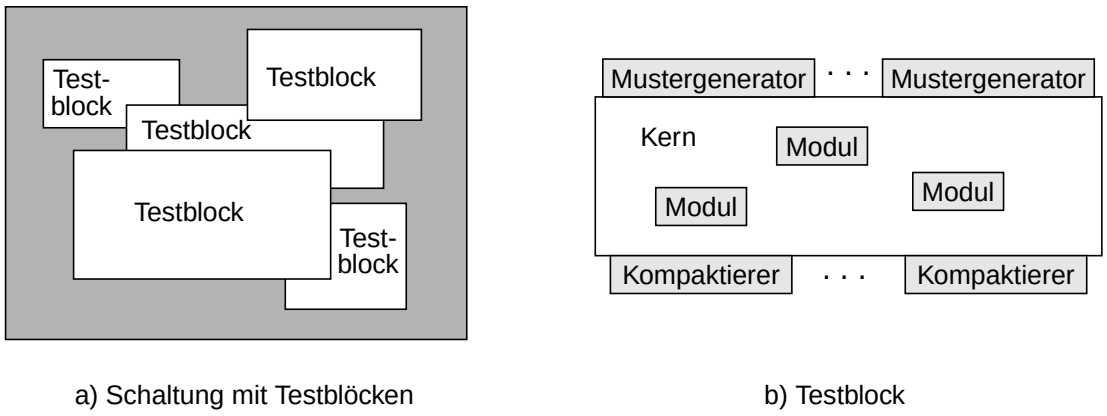
2.2.2 Selbsttest mit „Test pro Takt“-Schema

Beim „Test pro Takt“-Schema werden i.a. mehrere Mustergeneratoren und Kompaktierer in die Schaltung eingebaut, die mit jedem Takt ein neues Testmuster erzeugen bzw. eine Testantwort aufnehmen. Der Test läuft in Betriebsgeschwindigkeit ab, Verzögerungsfehler können demnach erkannt werden. Die Mustergeneratoren und Kompaktierer segmentieren die Schaltung in mehrere *Testblöcke* (s.u.). Während einer *Testsitzung* wird jeweils ein Testblock aktiviert, der dann in der Lage ist, einzelne Schaltungsmodule zu testen. Können mehrere Testblöcke parallel arbeiten, dürfen sie auch gemeinsam während einer Testsitzung aktiviert werden. Der Test aller Schaltungsmodule besteht i.a. aus einer Abfolge mehrerer Testsitzungen.

Im nächsten Abschnitt wird zunächst auf den Test mit Testblöcken eingegangen. Darauf wird der sogenannte „Test mit zirkulärem Prüfpfad“ als Spezialfall des „Test pro Takt“-Schemas behandelt.

2.2.2.1 Test mit Testblöcken

Bild 2.11a zeigt schematisch eine Schaltung mit mehreren Testblöcken. Ein Testblock besteht dabei aus einem *Kern* sowie Mustergeneratoren an seinen Eingängen und Kompaktierern an seinen Ausgängen (Bild 2.11b). Der Kern enthält ein oder mehrere zu testende Module sowie Transportpfade (hier nicht im einzelnen gezeigt), die benötigt werden, um Testmuster von den Eingängen zu diesen Modulen zu leiten bzw. die Testantworten der Module zu den Ausgängen. Einzelne Testblöcke können dabei überlappen, d.h. Schaltungsmodule gemeinsam haben.



a) Schaltung mit Testblöcken

b) Testblock

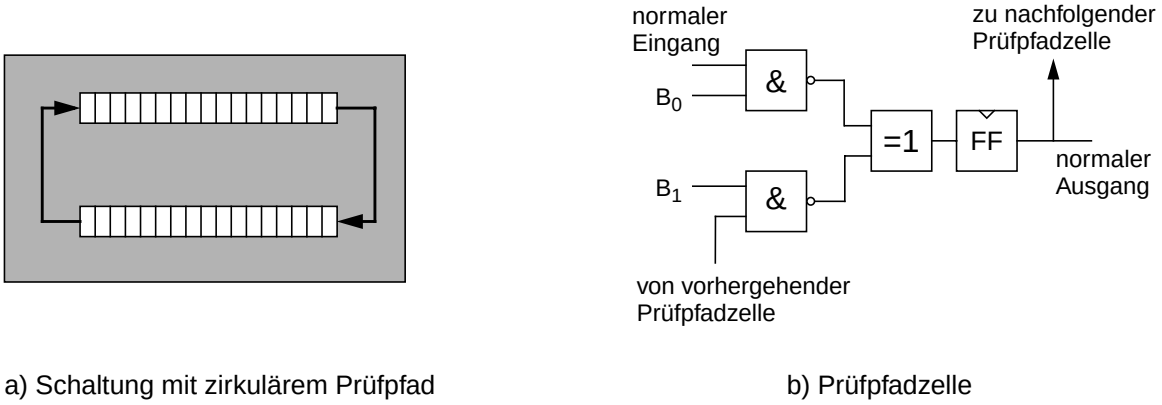
Bild 2.11: Test mit Testblöcken

Beim Test mit Testblöcken werden meist multifunktionale Testregister eingesetzt, die pseudozufällige Muster generieren und Testantworten kompaktieren können (s. Abschnitt 2.3). Dazu werden vorhandene Register zu Testregistern erweitert. Der Hardware-Aufwand ist dabei etwas größer als der Ausbau der gleichen Anzahl von Flipflops zu Scan-Flipflops. Außerdem ist die Teststeuerung wegen der verschiedenen Testsitzungen komplexer als die beim Test mit Prüfpfad.

Der wesentliche Vorteil beim Test mit Testblöcken ist, daß mit jedem Takt ein neues Testmuster erzeugt und eine Testantwort verarbeitet wird. Dadurch können auch bei mehreren Testsitzungen große Anzahlen pseudozufälliger Muster in akzeptabler Zeit angelegt werden. Beim Test mit Prüfpfad hingegen muß man sich i.a. auf eine vergleichsweise sehr geringe Zahl von Testmustern beschränken. Damit die Schaltung auch mit wenigen Testmustern ausreichend getestet werden kann, müssen die Testmuster erst manuell oder mit speziellen Werkzeugen berechnet werden. Der Hardware-Aufwand zur Generierung einer solchen spezifischen Testmustermenge kann dabei so groß werden, daß der Selbsttest mit Prüfpfad aus Kostengründen unaktraktiv wird.

2.2.2.2 Test mit zirkulärem Prüfpfad

Ein Spezialfall des „Test pro Takt“-Schemas ist der Test mit zirkulärem Prüfpfad [KrPi89]. Der Prüfpfad bildet hier einen Ring, wobei die im Prüfpfad enthaltenen Testmuster gleichzeitig als Signatur der Testantworten dienen (Bild 2.12a).



a) Schaltung mit zirkulärem Prüfpfad

b) Prüfpfadzelle

Bild 2.12: Zirkulärer Prüfpfad

Im Testbetrieb wird dazu die Testantwort am Eingang eines Flipflops zusätzlich mit dem Ausgang des im Pfad vorhergehenden Flipflops über ein XOR-Gatter verknüpft. Bild 2.12b zeigt den Aufbau einer Prüfpfadzelle mit den Betriebsarten *Rücksetzen* (B1 = 0, B_0 = 0), *Normalbetrieb* (B_1 = 0, B_0 = 1), *Schieben* (B_1 = 1, B_0 = 0) und *Testbetrieb* (B_1 = 1, B_0 = 1). Laut [POLB88] benötigt eine solche Zelle 77 % mehr Fläche als ein einfaches Flipflop.

Der Hardware-Aufwand beim zirkulären Prüfpfad ist geringer als beim Test mit Testblöcken. Allerdings kann aufgrund von Korrelationen aufeinanderfolgender Muster die Zahl der erzeugten Muster zu gering sein, um die Schaltung ausreichend zu testen. Ebenso besteht eine erhöhte Gefahr der Fehlermaskierung. In jedem Fall muß durch Fehlersimulation überprüft werden, ob die Schaltung ausreichend getestet wird. Die Fehlersimulation ist dabei sehr aufwendig, da sie anstelle kleiner, unabhängiger Testblöcke die komplette Schaltung umfaßt.

2.2.3 Deterministische Musterbestimmung

Wie schon oben erwähnt, werden bei der deterministischen Mustererzeugung die Testmuster anhand der zu testenden Schaltung und eines gegebenen Fehlermodells berechnet. Als Fehlermodell wird meist das sehr einfach zu behandelnde *Haftfehlermodell* angenommen [Eldr59]. Es setzt

auf der Gatterebene an und betrachtet zwei Arten von Fehlern an den Anschlüssen eines jeden Gatters oder Flipflops. Ein „ständig 1“-Fehler setzt den logischen Wert eines Anschlusses konstant auf 1, ein „ständig 0“-Fehler entsprechend auf 0.

Bei der automatischen Testmustererzeugung (automatic test pattern generation, ATPG) wird für jeden einzelnen Fehler ein Testmuster berechnet, d.h. ein Muster, das aufgrund des Fehlers eine andere Ausgabe erzeugen würde als im fehlerfreien Fall. Mehrfache Fehler bleiben wegen des aufgrund ihrer großen Anzahl immensen Rechenaufwands i.a. unberücksichtigt. Allerdings werden mit den Testmustern für einzelne Fehler auch die meisten mehrfachen Fehler erkannt [AbBF90].

Für Schaltnetze existiert eine Reihe von Werkzeugen zur automatischen Testmustererzeugung. Der Rechenaufwand bei bekannten Verfahren wie dem D-Algorithmus [Roth66], PODEM [Goel81] und FAN [FuSh83] wächst für reale Schaltungen im Schnitt quadratisch bis kubisch mit der Schaltungsgröße. Prinzipiell ist die Bestimmung eines Testmusters jedoch ein NP-vollständiges Problem [IbSa75], die Rechenzeit kann also exponentiell ansteigen.

Für Schaltwerke ist die Testmusterbestimmung noch wesentlich aufwendiger, da die Ausgabe außer von den Eingängen zusätzlich von den internen Flipflops abhängt. Daher ist meist eine ganze Musterfolge nötig, um einen Fehler zu aktivieren und ein fehlerhaftes Verhalten bis zu den Schaltungsausgängen zu propagieren. Ein bekanntes Werkzeug ist z.B. HITEC [NiPa91].

Der Test mit deterministischen Mustern war ursprünglich für den Einsatz eines externen Testautomaten gedacht. Die Testmuster werden dabei parallel an die Schaltungseingänge angelegt und evt. zusätzlich in einen Prüfpfad eingespeist. Hier ist die geringe Anzahl von Testmustern von Vorteil, die für den Test aller Haftfehler benötigt wird, da durch sie die Testdauer klein gehalten wird.

Die kleine Testmusteranzahl hat jedoch auch einen Nachteil, der darin beruht, daß die gängigen Werkzeuge zur Testmustergenerierung aus Aufwandsgründen nur Haftfehler berücksichtigen. Tatsächlich werden die in modernen CMOS-Schaltungen auftretenden Defekte wie z.B. Kurzschlüsse innerhalb von Gattern bzw. zwischen Gatteranschlüssen oder Unterbrechungen einzelner Leitungen von Haftfehlern nur unzureichend modelliert [FeSh88]. Wegen der geringen Anzahl von Testmustern kann es daher passieren, daß eine Schaltung trotz hoher Fehlererfassung bei den Haftfehlern nicht ausreichend getestet wird [MAJC91].

Beim Selbsttest muß die deterministisch bestimmte Testmustermenge von einem Mustergenerator auf der Schaltung selbst erzeugt werden. Im einfachsten Fall werden die Testmuster in einem ROM

gespeichert und nacheinander ausgelesen. Ein solches ROM kommt jedoch i.a. wegen des hohen Flächenbedarfs kaum in Frage. Ausgefeiltere Verfahren nutzen aus, daß Testmuster meist nur teilweise spezifiziert sein müssen, um Fehler zu detektieren. Auf diese Weise lassen sich Mustergeneratoren mit deutlich geringerem aber immer noch beträchtlichem Hardware-Aufwand nutzen [HELL95, WuKi96].

2.2.4 Erschöpfende und pseudoerschöpfende Mustermengen

Eine Alternative zum Test mit deterministischen Testmustern ist der erschöpfende bzw. pseudoerschöpfende Test. Beim erschöpfenden Test werden alle Eingangsbelegungen aufgezählt, z.B. mit einem einfachen Zähler. Auf diese Weise lassen sich alle kombinatorischen Fehler eines Schaltnetzes mit geringem Hardware-Aufwand testen. Da die Mustermenge jedoch exponentiell mit der Anzahl der Eingänge steigt, ist der erschöpfende Test aus Zeitgründen auf Schaltungen mit höchstens 20 bis 30 Eingängen beschränkt. Für den Test mit Prüfpfad ist er daher ungeeignet.

In vielen Fällen hängt ein einzelner Schaltungsausgang nur von einem Teil der Schaltungseingänge ab. Die Teilschaltung, die den Ausgang beeinflußt, nennt man einen Kegel. Beim pseudoerschöpfenden Test [McBo81] werden anstelle der gesamten Schaltung nur die Kegel erschöpfend getestet. Auf diese Weise läßt sich die Testzeit oft erheblich verkürzen. Allerdings können nur die kombinatorischen Fehler innerhalb der einzelnen Kegel garantiert detektiert werden, manche Kurzschlüsse zwischen den Kegeln jedoch nicht.

2.2.5 Zufällige Muster

Viele Schaltungen lassen sich auch mit zufälligen Mustern testen. Die Voraussetzung dafür ist, daß sich jeder Fehler mit sehr vielen Testmustern detektieren läßt. In diesem Fall ist die Wahrscheinlichkeit sehr groß, alle Fehler innerhalb einer kurzen Musterfolge und somit in kurzer Testzeit zu entdecken. Methoden zur Schätzung und zur exakten Berechnung der Fehlererkennungswahrscheinlichkeiten finden sich u.a. in [Wund85, ChHu90, Wund91].

Bei schwerer detektierbaren Fehlern ist es oft hilfreich, gewichtete Zufallsmuster zu verwenden. Hier können die Wahrscheinlichkeiten, daß ein bestimmtes Bit den Wert 0 bzw. 1 annimmt, verschieden sein. Zur Veranschaulichung wird im folgenden ein UND-Gatter mit n Eingängen betrachtet. Um einen „ständig 1“-Fehler an einem der n Eingänge des Gatters zu überprüfen, muß an den

Eingang eine 0 und an alle anderen Eingänge eine 1 angelegt werden. Nur bei diesem Eingangsmuster verursacht der Fehler einen anderen Ausgabewert als im fehlerfreien Fall. Gleichzeitig testet ein solches Eingangsmuster einen „ständig 1“-Fehler am Gatterausgang – tatsächlich ist hierfür jedes Muster geeignet, das mindestens ein Bit auf 0 setzt. Für einen beliebigen „ständig 0“-Fehler an einem Eingang müssen hingegen alle Eingangsbits auf 1 gesetzt werden. Dieses Testmuster überprüft auch als einziges einen „ständig 0“-Fehler am Ausgang.

Ein Test auf alle Haftfehler muß offensichtlich die n+1 obengenannten Testmuster enthalten. Seien p_1 bzw. p_0 die Wahrscheinlichkeiten, daß an einem Eingangsbit der Wert 1 bzw. 0 anliegt. Die Wahrscheinlichkeit für ein Eingangsmuster mit einer einzigen 0 ist bei n Eingangsbits somit $P_{eine\_0} = p_1^{n-1} \cdot p_0$. Für das Muster mit keiner 0 gilt $P_{keine\_0} = p_1^{n}$. Bei gleichverteilten Mustern wäre die Auftrittswahrscheinlichkeit für ein beliebiges Muster 2^{-n} und damit bei großen n sehr gering. Die für einen vollständigen Haftfehler erforderliche Testlänge – die Anzahl anzulegender Muster – ist dann i.a. sehr groß. Betrachtet man hingegen eine beliebige Bitposition bei allen n+1 benötigten Testmustern, so erkennt man, daß jedes Bit in n Testmustern den Wert 1 annimmt, jedoch nur einmal den Wert 0. Offensichtlich ist es für ein kurze Testlänge sinnvoll, gewichtete Zufallsmuster zu verwenden, deren einzelne Bits eher den Wert 1 annehmen als den Wert 0. Für n = 32 erhält man kurze Testlängen mit $p_1 = \sqrt[32]{\frac{1}{2}}$ und damit $P_{eine\_0} \approx 0{,}011$ und $P_{keine\_0} = 0{,}5$ [Wund91, S. 291].

Die Testlängen bei Zufallsmustern sind normalerweise wesentlich größer als bei deterministisch bestimmten Testmustern. Der Zufallstest eignet sich daher vor allem für das „Test pro Takt“-Schema, bei dem die Testlänge eine weit geringere Rolle spielt als beim „Test pro Scan“-Schema. Durch die große Musteranzahl werden allerdings auch viele Fehler detektiert, die sich nicht durch Haftfehler modellieren lassen. Im Vergleich zum Test mit deterministischen Mustern ist der Hardware-Aufwand zur Erzeugung (pseudo)zufälliger Muster sehr gering (s. Abschnitt 2.3) und damit für den Selbsttest meist vertretbar.

2.3 Mustergeneratoren und Kompaktierer für den Selbsttest

Mustergeneratoren und Kompaktierer werden beim Selbsttest mit auf dem Chip integriert. Durch den zusätzlichen Flächenbedarf steigen die Herstellungskosten. Wie oben erwähnt, sind Mustergeneratoren für den Selbsttest mit deterministischen Mustern i.a. sehr teuer. Im folgenden wird daher nur auf die für den Selbsttest geeigneteren Mustergeneratoren für (pseudo-)erschöpfende und pseudozufällige Muster eingegangen.

Häufig beim Selbsttest eingesetzte Mustergeneratoren und Kompaktierer sind sogenannte Testregister, bei denen ein vorhandenes Register für Testzwecke erweitert wird. Relativ neu ist der Einsatz von Akkumulatoren. Werden sie allein aus schaltungseigenen Modulen konfiguriert, kommen sie praktisch ohne zusätzliche Hardware aus. Testregister und Akkumulatoren werden in den nächsten beiden Abschnitten näher beschrieben.

2.3.1 Testregister

Testregister bauen sehr häufig auf linear rückgekoppelten Schieberegistern (LRSR) auf, bei denen einzelne Flipflopausgänge über XOR-Verknüpfungen auf Flipflopeingänge rückgekoppelt werden. Bild 2.13 zeigt als Beispiel ein einfaches LRSR mit einer externen XOR-Verknüpfung.

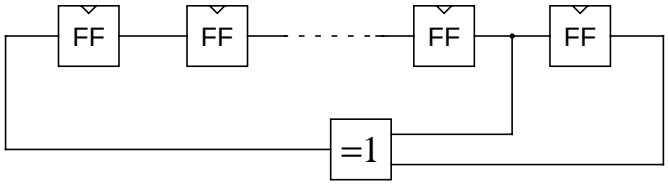


Bild 2.13: Linear rückgekoppeltes Schieberegister mit externer XOR-Verknüpfung

Aufgrund der Rückkopplung durchläuft das LRSR eine zyklische Folge von Zuständen. Bei manchen Rückkopplungen wird der Zyklus maximal groß, bei n Bits enthält er dann 2^n-1 Muster, d.h. bis auf ein Muster lassen sich alle Muster erschöpfend aufzählen. Der 0-Zustand, bei dem alle Bits den Wert '0' haben, wird immer auf sich selbst abgebildet, d.h. er kann nicht in einem maximalen Zyklus enthalten sein. Es ist jedoch möglich, ein solches maximalperiodisches LRSR mit zwei zusätzlichen Gattern zu einem vollständigen, rückgekoppelten Schieberegister (complete feedback shift register [McCl86, WaMc86b]) zu erweitern, das alle 2^n Muster zyklisch erzeugen kann.

Bild 2.14 zeigt zwei einfache Methoden, mithilfe eines LRSR pseudoerschöpfende Muster zu erzeugen. In Bild 2.14a speist ein maximalperiodisches LRSR ein Schieberegister. In jedem Takt liegt ein neues Muster am Ausgang des LRSR und des Schieberegisters an. Bei einem w-bit-LRSR und geeigneter Wahl der Rückkopplungsfunktion lassen sich in einer Schaltung praktisch alle Kegel mit maximal w Eingangsbits erschöpfend testen [BaCR83]. Eine andere Methode zeigt Bild 2.14b. Hier sind mehrere Kopien der Zustandsflipflops des LRSR aneinandergereiht. Auf diese

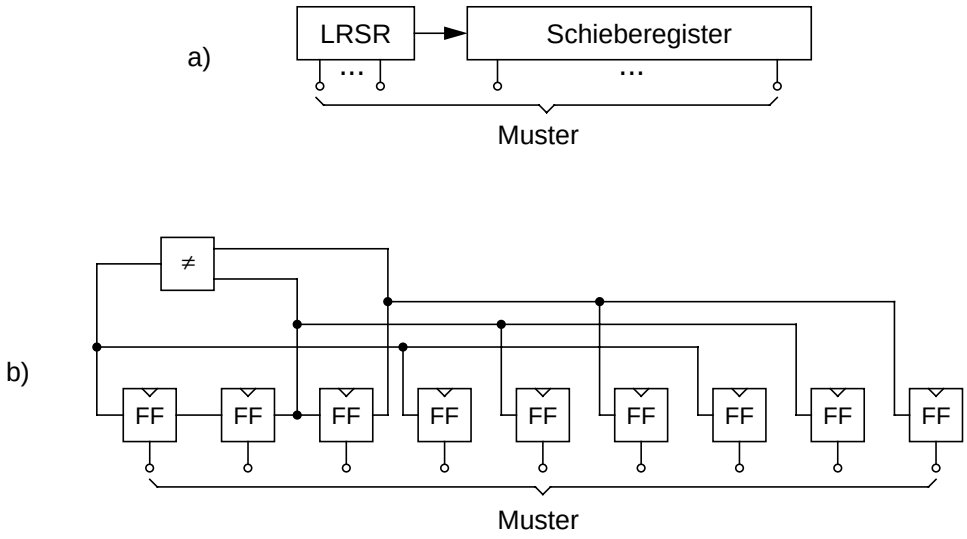


Bild 2.14: Beispiele zur Erzeugung pseudoerschöpfender Muster

Weise ist es möglich, mit einem w-bit-LRSR alle Kegel mit w aufeinanderfolgenden Eingangsbits nahezu erschöpfend zu testen [BeFR97].

Außer für die Erzeugung erschöpfender oder pseudoerschöpfender Musterfolgen läßt sich ein LRSR auch für den Zufallstest einsetzen. Die Bitfolgen eines maximalperiodischen LRSR haben nämlich ähnliche Eigenschaften wie zufällige Bitfolgen, man nennt sie pseudozufällig [Golo67]. Generell ist ein LRSR für das „Test pro Scan“- wie für das „Test pro Takt“-Schema geeignet. Betrachtet man nur den Ausgang eines einzigen Flipflops erhält man einen seriellen Mustergenerator. Für das „Test pro Takt“-Schema wird dagegen der komplette Registerinhalt als paralleles Muster verwendet.

Auf der Basis eines LRSR beruht auch das parallele Signaturregister (multiple input signature register, MISR), das zur Kompaktierung eingesetzt wird (Bild 2.15).

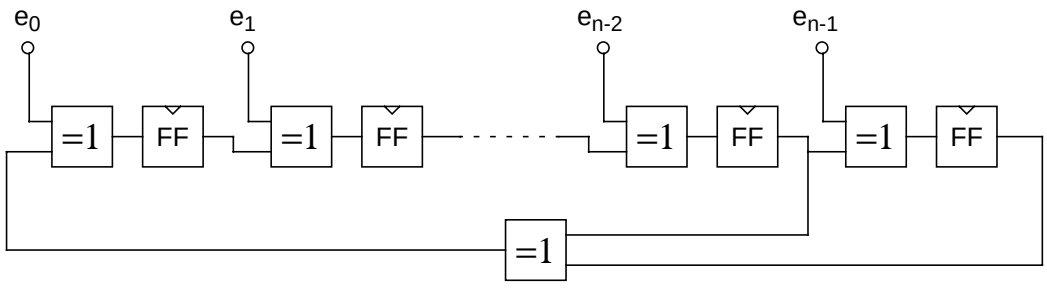
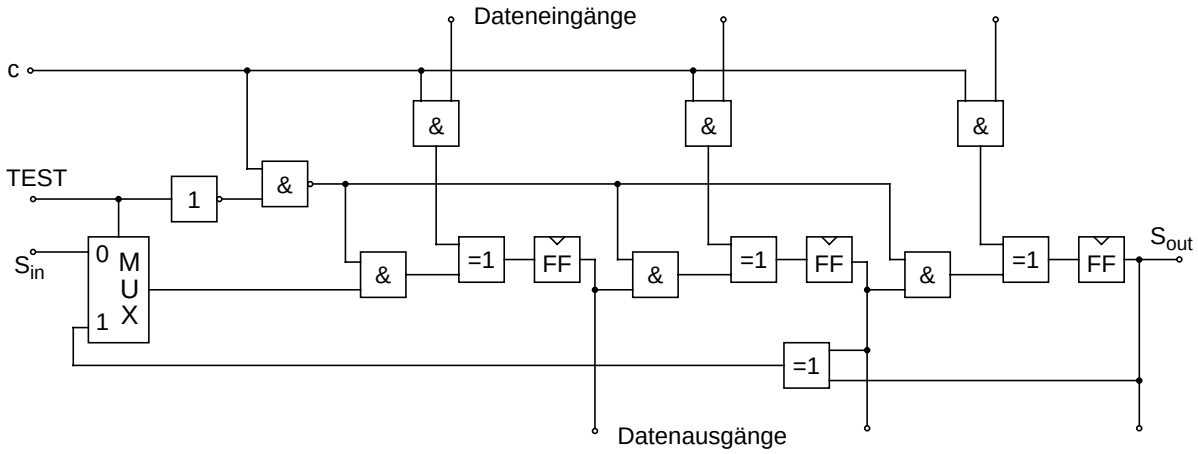


Bild 2.15: Beispiel eines n-bit-MISR

Die Testantworten werden dabei über die Eingänge eingegeben. Das serielle Signaturregister (single input signature register, SISR) mit nur einem Eingang e_0 ist ein Spezialfall davon.

Wie schon oben erwähnt, können sich Fehler in den Testantworten durch die Kompaktierung gegenseitig aufheben, so daß am Testende die Signatur des fehlerfreien Falls im MISR enthalten ist, obwohl Fehler aufgetreten sind. Man spricht dann von Fehlermaskierung. Unter der Voraussetzung, daß aufeinanderfolgende Testantworten statistisch unabhängig voneinander erzeugt werden und mindestens zwei verschiedene Abweichungen auftreten, strebt die Wahrscheinlichkeit für eine Fehlermaskierung für Schaltnetze mit rein kombinatorischen Fehlern bei einem n-bit-MISR jedoch schnell gegen den Wert 2^{-n}, sofern die Rückkopplungsfunktion wie bei einem LRSR mit maximalem Zyklus gewählt wurde [DaWW90, KaPI93]. Dieser Wert ist bei den für den Zufallstest üblichen Testlängen eine sehr gute Näherung [DaWW90] und bei den Bitbreiten typischer MISRs vernachlässigbar gering.



Betriebsart	Steuersignale	
	TEST	c
Normalbetrieb	0	1
Schiebebetrieb	0	0
Mustererzeugung	1	0
Signaturanalyse	1	1

Bild 2.16: 3-bit-BILBO

Ein wohlbekanntes Beispiel für ein multifunktionales Testregister ist das BILBO (built-in logic block observer) [KöMZ79]. In einer Variante kann es außer als normales Register noch als Schieberegister, MISR und LRSR arbeiten (Bild 2.16).

Benötigt man neben dem Normalbetrieb nur noch die Mustererzeugung bzw. die Signaturanalyse, lassen sich einige Gatter einsparen. Eine Erweiterung des BILBO ist das sogenannte *Concurrent* BILBO (CBILBO) [WaMc86a]. Im Gegensatz zum BILBO kann dieses gleichzeitig als LRSR und MISR arbeiten, allerdings auf Kosten eines zusätzlichen Registers und zusätzlicher Steuerlogik. Komplette Testregister lassen sich auch direkt in Leitungsbündel integrieren, wo keine schaltungseigenen Register verfügbar sind. Man spricht dann von transparenten Testregistern. Mithilfe eines Multiplexers kann man zwischen der normalen Verbindung und einem Pfad über das Testregister wählen. Die Zusatzkosten entstehen hier durch den Multiplexer und das komplette Testregister.

2.3.2 Akkumulatoren

Wie bereits erwähnt, stellen Akkumulatoren eine Alternative zu Testregistern dar. Werden sie allein aus schaltungseigenen Modulen konfiguriert, fällt praktisch kein zusätzlicher Hardware-Aufwand an. Ebenfalls treten keine Geschwindigkeitseinbußen aufgrund erhöhter Signallaufzeiten auf. Bild 2.17 zeigt nocheinmal den prinzipiellen Aufbau eines Akkumulators. Zur Mustererzeugung wird der Registerinhalt fortwährend auf einen Eingang der arithmetischen Einheit propagiert, dort mit einer Konstanten v verknüpft, und das Ergebnis wieder im Register gespeichert. Im Falle der Kompaktierung treten anstelle der Konstanten die Testantworten.

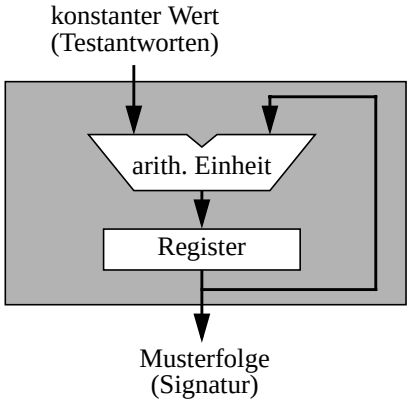


Bild 2.17: Akkumulator

Die Rückkopplungsschleife eines Akkumulators darf weitere Module wie Multiplexer, Register und sogar zusätzliche arithmetische Einheiten enthalten, sofern nur der Registerinhalt zur arithme-

tischen Einheit propagiert werden kann. Akkumulatoren mit mehreren Registern können dabei verschiedene Aufgaben in verzahnter Weise ausführen, z.B. Mustererzeugung in einem Taktzyklus und Kompaktierung in einem anderen Zyklus.

Als arithmetische Einheiten eignen sich vor allem Addierer und Subtrahierer [RaTy93a/b, GuRT94, Strö95a/b, Strö96a/b]. Allerdings muß die Rückkopplungsschleife bei Subtrahierern über den Minuendeneingang verlaufen, andernfalls enthielte die Musterfolge nur maximal zwei verschiedene Muster. Multiplizierer sind nur bedingt geeignet [Strö95b, Strö96b]. Bild 2.18 zeigt verschiedene Addierer sowie die erzeugten Musterfolgen.

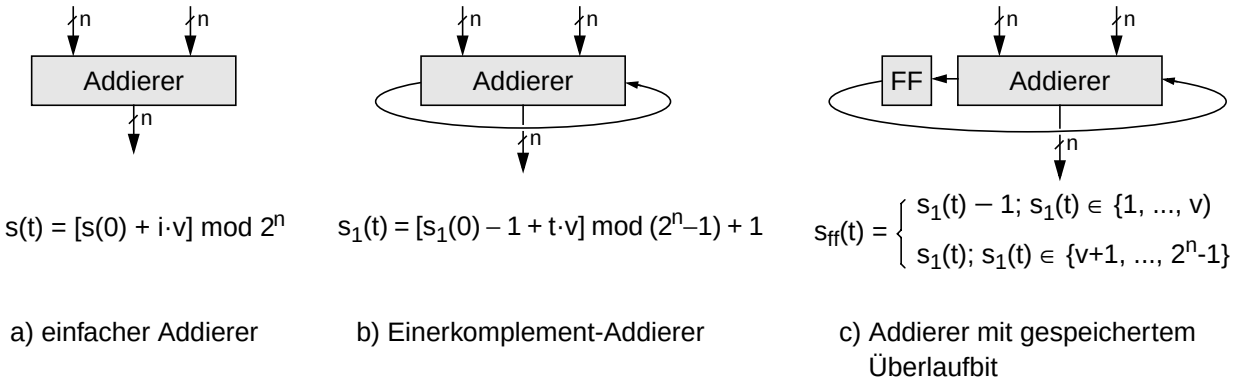


Bild 2.18: Verschiedene Addierer mit Musterfolgen

Ein einfacher n-bit-Addierer (Bild 2.18a) generiert Muster $s(t) = [s(0) + t \cdot v] \bmod 2^n$, wobei der Startwert s(0) der Initialwert des Registers ist. Ein Einerkomplement-Addierer, bei dem das Überlaufbit direkt mit dem Übertragseingang des niedrigstwertigen Bits verbunden ist, erzeugt die Folge $s_1(t) = [s_1(0) - 1 + t \cdot v] \bmod (2^n - 1) + 1$, sofern $v \neq 0$ gilt (Bild 2.18b). Tatsächlich benötigt ein solcher Addierer ein zusätzliches Gatter in der Rückkopplung, um mögliche Oszillationen zu vermeiden. Wird das Überlaufbit in einem zusätzlichen Flipflop zwischengespeichert (Bild 2.18c) , erhält man mit $v \neq 0$ und $v \neq 2^n - 1$: $s_{ff}(t) = s_1(t) - 1$, wenn $s_1(t) \in \{1, 2, ..., v\}$ gilt, bzw. $s_{ff}(t) = s_1(t)$ für $s_1(t) \in \{v + 1, ..., 2^n - 1\}$, wobei $s_1(t)$ wieder das Muster des Einerkomplement-Addierers ist.

Für alle drei Addierer sind durch geeignete Wahl des Startwertes und der Konstanten v erschöpfende Musterfolgen möglich. Obwohl die Folgen nicht die pseudozufälligen Eigenschaften haben wie maximalperiodische LRSR, ist die Fehlererfassung vergleichbar mit denen von LRSRs, wenn man gleiche Testlängen voraussetzt [Strö95a/b, Strö96b]. Beim einfachen Addierer erhält man unabhängig vom Startwert Zyklen der Länge 2^n, wenn für den größten gemeinsamen Teiler von v und 2^n gilt: $ggT(v, 2^n) = 1$. Die Konstante v muß also lediglich ungerade sein. Beim Einerkomple-

ment-Addierer sind nur höchstens 2^n-1 Muster zyklisch erzeugbar. Dafür muß gelten: ggT(v, 2^n-1) = 1. Der Wert 0 ist in keinem Zyklus enthalten. Wählt man ihn jedoch als Startwert, ist der Nachfolgewert Teil des Zyklus, und alle 2^n Muster können generiert werden. Das gleiche gilt für den Addierer mit gespeichertem Überlaufbit. Hier ist statt 0 der Wert v nicht im Zyklus enthalten.

Die Musterfolgen der Addierer lassen sich bijektiv auf die Musterfolgen der entsprechenden Subtrahierer abbilden. Subtrahierer eignen sich demnach ebenso gut als Mustergeneratoren wie Addierer. Eine genauere Beschreibung der Musterfolgen von Akkumulatoren mit Addierern bzw. Subtrahierern findet sich in [Strö95a/b, Strö96b, Strö98b].

In [GuRT94] werden Akkumulatoren für die Erzeugung pseudoerschöpfender Musterfolgen betrachtet, wobei man sich wie in [BeFR97] auf Kegel mit aufeinanderfolgenden Eingangsbits beschränkt. Als arithmetische Einheit wird ein einfacher Addierer verwendet. Anhand umfangreicher Simulationen werden Startwerte und Konstanten für verschiedene Kegelbreiten so optimiert, daß für jeden Kegel alle möglichen Eingangsbelegungen nach möglichst wenigen Mustern abgedeckt werden.

In [RaTy93a/b, Strö96a] wurden die Kompaktiereigenschaften der obengenannten Akkumulatoren mit Addierern und Subtrahierern untersucht. Addierer und Subtrahierer ohne Rückkopplung des Überlauf- bzw. Unterlaufbits erwiesen sich dabei als für die Kompaktierung ungeeignet, da die Fehlermaskierungswahrscheinlichkeit für Fehler, die nur die höherwertigen Bits beeinflussen, relativ groß ist. Für Akkumulatoren mit Rückkopplung des Überlauf- bzw. Unterlaufbits ist der Grenzwert der Fehlermaskierungswahrscheinlichkeit hingegen $\frac{1}{2^n-1}$, sofern die Bitbreite n des Akkumulators so gewählt ist, daß 2^n-1 eine Primzahl ist. Auch wenn 2^n-1 keine Primzahl ist, gilt der Grenzwert $\frac{1}{2^n-1}$ für viele Schaltungsfehler. Für die anderen Fehler ist er ein Vielfaches davon. Bei geeigneter Wahl der Testlänge, z.B. einer Zweierpotenz, kann jedoch für manche dieser Fehler eine Maskierung ganz vermieden werden. Akkumulatoren zeigen demnach ähnlich gute Kompaktierungseigenschaften wie MISRs.

In [Strö98a] wurden die Eigenschaften der Akkumulatoren zur bitseriellen Mustererzeugung bzw. Kompaktierung untersucht. Die Ergebnisse zeigten, daß Akkumulatoren als serielle Kompaktierer ebenso geeignet sind wie SISR. Für die serielle Mustererzeugung sind die Akkumulatoren mit nur einem Addierer (Subtrahierer) dagegen nicht zu gebrauchen. Tatsächlich variiert die Anzahl der in den Bitsequenzen aufeinanderfolgenden Einsen bzw. Nullen nicht oder kaum. Komplexere Akku-

mulatoren, die neben einem Addierer (Subtrahierer) auch einen Multiplizierer enthalten, sind hingegen für die serielle Mustererzeugung sehr gut geeignet [RaRT97, Strö98a].

Akkumulatoren bieten wie multifunktionale Testregister alle Funktionen, die für den Selbsttest einer Schaltung notwendig sind. Testregister sind gewiß universeller einsetzbar, da jedes Register einer Schaltung zu einem Testregister ausgebaut werden kann. Jedoch überall dort, wo in einer Schaltung geeignete arithmetische Einheiten bereits vorhanden sind, haben Akkumulatoren den Vorteil des geringeren Hardware-Mehraufwandes. Zudem vermindern sie die maximal mögliche Taktfrequenz nicht.

2.3.3 Verfahren zur Erzeugung kurzer Testlängen

Testregister wie Akkumulatoren können (pseudo)erschöpfende Musterfolgen erzeugen, die sich auch für den Zufallstest eignen. Bei größeren Schaltungen mit vielen Eingangsbits werden die benötigten Testlängen für den (pseudo)erschöpfenden Test jedoch oft zu groß. Beim Zufallstest spielt die Schaltungsgröße eine eher untergeordnete Rolle, wichtig ist vor allem, wie leicht sich die Fehler entdecken lassen. Existieren für einen Fehler nur wenige Testmuster, ist die Chance gering, ihn mit einer kurzen Musterfolge zu detektieren. Für Schaltungen mit solchen *harten* Fehler wurden verschiedene Verfahren entwickelt, die durch geschickte Auswahl der Musterfolge oder mit zusätzlicher Hardware eine akzeptable Testlänge erzielen.

Die Verfahren konzentrieren sich dabei i.a. auf kombinatorische Fehler, d.h. Fehler, die sich durch ein einziges Testmuster erkennen lassen. Zwar existieren auch Verfahren für Verzögerungsfehler [WuFu95, VPNH95], die auf dem Zufallstest basieren. Verzögerungsfehler benötigen jedoch Paare von aufeinanderfolgenden Testmustern, um entdeckt zu werden. Bei $2^n \cdot (2^n - 1)$ möglichen Musterpaaren für n Eingabebits wird die Testlänge deshalb trotz der Verfahren schon bei wenigen Eingabebits zu lang. Im folgenden werden daher nur Verfahren für kombinatorische Fehler beschrieben.

2.3.3.1 Verfahren für Testregister

Viele Verfahren passen die Musterfolgen von Testregistern mit zusätzlicher Hardware an eine Schaltung an. In der Regel wird eine Schaltung vorab analysiert, die Testmuster werden für die betrachteten Fehler bestimmt und dann für die eigentliche Anpassung herangezogen.

Ein bekanntes Verfahren ist die Verwendung gewichteter Zufallsmuster (s. Abschnitt 2.2.5). In [Wund87] wird die sogenannte GURT-Konfiguration (Generator of Unequiprobable Random Tests) beschrieben, bei der die einzelnen Bits unterschiedliche 0- und 1-Wahrscheinlichkeiten aufweisen können. Sie wird als paralleler Mustergenerator eingesetzt und besteht im wesentlichen aus mehreren maximalperiodischen LRSRs, die jeweils über eine boolesche Funktion f ein Schieberegister speisen (Bild 2.19).

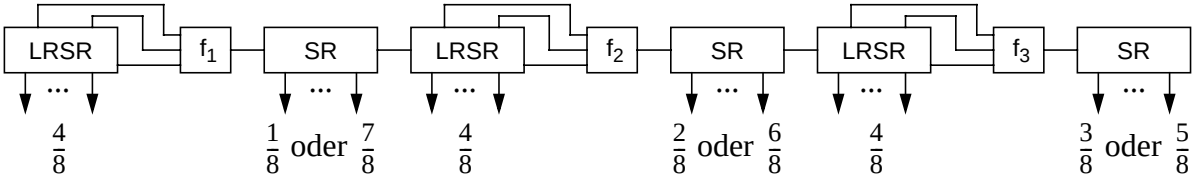


Bild 2.19: GURT-Konfiguration für die Gewichte $\frac{1}{8}$, ..., $\frac{7}{8}$

Die Funktion f verknüpft einige Zustandsbits eines LRSR. Sie wird so gewählt, daß die Anzahl ihrer Minterme geteilt durch die Anzahl aller möglichen Minterme die gewünschte 1-Wahrscheinlichkeit ergibt. Bei drei Eingängen a_1, a_2 und a_3 kann man mit f $\in$ {$a_1a_2a_3$, a_2a_3, $\overline{a_1 \vee a_2a_3}$, a_3, $a_1 \vee a_2a_3$, $\overline{a_2a_3}$, $\overline{a_1a_2a_3}$} die sieben 1-Wahrscheinlichkeiten $\frac{1}{8}$, $\frac{2}{8}$, ..., $\frac{7}{8}$ einstellen. Beim GURT genügen die Gewichte $\frac{1}{8}$, $\frac{2}{8}$, $\frac{3}{8}$ und $\frac{4}{8}$. Die übrigen drei Gewichte erhält man an den negierten Ausgängen der Schieberegisterzellen.

Anstelle der LRSR lassen sich auch Akkumulatoren denken. Die 0- und 1-Wahrscheinlichkeiten bleiben dadurch unverändert. Allerdings wurden die Korrelationen zwischen einzelnen Bitpositionen bei Akkumulatoren noch nicht untersucht.

In [ToMc95] ist ein Verfahren beschrieben, das bei vorgegebener Testlänge alle betrachteten Fehler detektiert. Dies wird durch ein zusätzliches logisches Netzwerk zwischen dem Mustergenerator und der zu testenden Schaltung erreicht. Das Netzwerk transformiert eine gegebene Musterfolge so, daß für jeden Fehler mindestens ein Testmuster erzeugt wird. Zur Bestimmung des Netzwerkes wird die Musterfolge simuliert. Jedes Muster, das einen Fehler das erste Mal entdeckt, wird auf sich selbst abgebildet, die übrigen Muster werden so transformiert, daß mit einem möglichst kleinen Netzwerk eine komplette Fehlererfassung garantiert ist. Da das Verfahren unabhängig vom Typ des Mustergenerators arbeitet, ist es sowohl für Testregister als auch Akkumulatoren zu gebrauchen.

Ein anderer Ansatz verzichtet auf zusätzliche Hardware und optimiert stattdessen den Startwert eines maximalperiodischen LRSR für verschiedene zufällig ausgewählte Rückkopplungsfunktio-

nen, so daß alle Fehler bei möglichst kurzer Testlänge entdeckt werden [LeGB95]. Von den optimierten Musterfolgen wird schließlich die kürzeste ausgewählt. Die Startwertoptimierung beruht dabei auf der Berechnung diskreter Logarithmen, die sehr zeitaufwendig werden kann. Weitere Verfahren zur Testlängenverkürzung finden sich in [PoRe93] und [SaMc90].

2.3.3.2 Verfahren für Akkumulatoren

Im Gegensatz zu Testregistern sind für Akkumulatoren noch kaum Verfahren zur Anpassung von Musterfolgen an eine gegebene Schaltung bekannt. Eine erst kürzlich vorgestellte Methode für Akkumulatoren mit einfachem Addierer verwendet mehrere kurze Musterfolgen, um alle Fehler innerhalb einer vorgegebenen Testlänge zu erfassen [DoWu98]. Ziel ist es, mit möglichst wenigen Musterfolgen auszukommen, da für jede Folge der Startwert und der konstante Eingabewert gespeichert werden müssen. Ausgangspunkt sind wieder die Testmuster aller betrachteten Fehler. Der Kern des Verfahrens ist, auf der Grundlage sogenannter ROBDDs (reduced ordered binary decision diagrams [Brya86]) Musterfolgen derselben vorgegebenen Länge zu bestimmen, die eine gegebene Menge von Testmustern detektieren können. Der Speicher- und Rechenaufwand dafür ist jedoch für praktische Anwendungen nicht akzeptabel. Deshalb schränken mehrere Heuristiken den Suchraum von vornherein extrem ein, entsprechend sinkt die Qualität der gefundenen Lösung.

2.3.3.3 Leicht testbare Schaltungen

Die bisher vorgestellten Verfahren hatten stets das Ziel, die erzeugte Musterfolge an die zu testende Schaltung anzupassen. Eine andere Möglichkeit besteht darin, die zu testende Schaltung selbst so zu gestalten, daß eventuelle Fehler leicht zu detektieren sind. In diesem Fall ist es möglich, ausreichend viele (d.h. oft >95%) der betrachteten Fehler einer Schaltung mit kurzen Folgen von zufälligen Mustern zu testen. Im folgenden wird eine solche Schaltung kurz *zufallstestbar* genannt.

Eine Methode zur Steigerung der Testbarkeit ist der Einbau von sogenannten Testpunkten in die Schaltung selbst [HaFr73]. Die Idee ist dabei, schwer zugängliche Leitungen innerhalb der Schaltung durch zusätzliche Schaltungseingänge bzw. -ausgänge leichter einstellen und beobachten zu können. Der Hardware-Mehraufwand beträgt dafür oft nur etwa 1 % [SeTS91]. Bei den Testpunkten unterscheidet man sogenannte Beobachtungspunkte und Steuerpunkte. Ein Beobachtungspunkt stellt eine zusätzliche Leitung zwischen einer zu beobachtenden Leitung v innerhalb der Schaltung und einem zusätzlichen Schaltungsausgang dar (gestrichelte Linie in Bild 2.20a). Dadurch kann

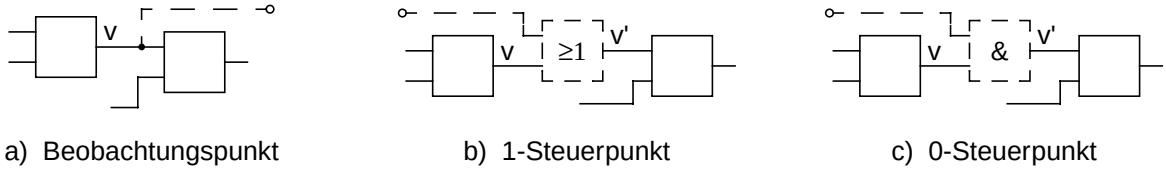


Bild 2.20: Beobachtungspunkt (a) und Steuerpunkte zur Einstellung der Werte 1 (b) bzw. 0 (c)

jede Änderung auf der Leitung v an einem Schaltungsausgang beobachtet werden. Außerdem wird die Beobachtbarkeit aller Pfade zwischen Schaltungseingängen und der Leitung v erhöht.

Mit Steuerpunkten kann man eine Leitung v gezielt auf einen Wert 1 bzw. einen Wert 0 setzen (gestrichelter Teil in Bild 2.20b bzw. c). Dafür wird ein zusätzlicher Schaltungseingang über ein ODER- bzw. UND-Gatter mit der Leitung v verknüpft. Steuerpunkte werden dann eingesetzt, wenn einzelne Leitungen nur schwer über die Schaltungseingänge mit Werten 0 bzw. 1 zu belegen sind. Beim Zufallstest werden auch an die zusätzlichen Schaltungseingänge zufällige Werte angelegt. Dadurch kann die Beobachtbarkeit mancher Leitungen beeinflußt werden. Beispielsweise kann bei einer 1-Wahrscheinlichkeit P_1 am zusätzlichen Schaltungseingang in Bild 2.20b bzw. c der Wert von v nur noch mit Wahrscheinlichkeit $1–P_1$ bzw. P_1 am Knoten v' beobachtet werden. Viele Verfahren berücksichtigen diesen Sachverhalt beim Einbau von Testpunkten [SeTS91, ToMc96, ChLi95]. Sie versuchen oft auch zusätzliche Schaltungseingänge bzw. -ausgänge einzusparen, indem sie solche Eingänge über Multiplexer mit einem normalen Schaltungseingang verknüpfen bzw. mehrere zusätzliche Schaltungsausgänge mit XOR-Gattern zusammenfassen [IyBr89]. In [ChLi95] werden zudem Testpunkte bevorzugt in nicht zeitkritische Pfade eingebaut, um Einbußen bei der maximalen Betriebsgeschwindigkeit zu vermeiden.

Eine andere Methode, Schaltungen zufallstestbar zu machen, benutzt Logiktransformationen [ChPK95, ChGu94]. Ein Beispiel einer solchen Transformation zeigt Bild 2.21.

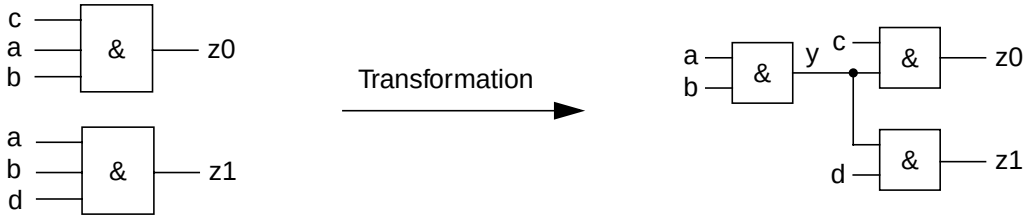


Bild 2.21: Erhöhung der Testbarkeit durch Transformation einer Schaltung

Vor der Transformation ist jeder Haftfehler an den Eingangsleitungen a und b der beiden Gatter jeweils nur durch ein einziges Testmuster zu detektieren, ein „ständig 0“-Fehler auf der Leitung a am oberen UND-Gatter z.B. mit dem Testmuster abc. Nach der Transformation sind die Eingangsleitungen a und b bzw. die Ausgangsleitung y des neu hinzugekommenen Gatters mit mindestens zwei Testmustern zu erfassen. Ein „ständig 0“-Fehler auf a z.B. mit Mustern abc und abd. Für die übrigen Leitungen hat sich die Zahl der Testmuster nicht verändert. Insgesamt wurde die Testbarkeit der Schaltung jedoch gesteigert. Nicht immer lassen sich schwer testbare Fehler durch solche Transformationen vermeiden. Das Verfahren in [ToMc99] kombiniert deshalb die Transformation einer Schaltung mit dem Einbau von Testpunkten.

2.4 Schaltungsentwurf unter Berücksichtigung des Selbsttests

In den vorigen Abschnitten wurden zum einen der automatische Schaltungsentwurf beschrieben, zum anderen wurden verschiedene Teststrategien sowie spezielle für den Selbsttest entwickelte Testhardware vorgestellt. Dieser Abschnitt befaßt sich mit der Integration der verschiedenen Selbstteststrategien in die Entwurfsphasen der herkömmlichen Schaltungssynthese. Der Selbsttest hat heutzutage in alle Entwurfsstadien Einzug gehalten, beginnend bei der algorithmischen Verhaltensbeschreibung bis hinab zur Gatterebene und darunter. Veränderungen der algorithmischen Verhaltensbeschreibung bzw. Eingriffe während der High-Level-Synthese haben dabei großen Einfluß auf den späteren Hardware-Mehraufwand für den Test sowie auf die Testzeit, weil sie sich auf die Schaltungsstruktur auswirken. Die Optimierungsmöglichkeiten auf Register-Transfer-Ebene bzw. Gatterebene sind geringer, da die Schaltungsstruktur bereits festliegt. Allerdings lassen sich zusätzliche Kosten besser abschätzen und somit gute Lösungen von schlechteren sauberer trennen.

Die verschiedenen Vorgehensweisen zur Berücksichtigung des Selbsttests hängen sehr stark von der gewählten Teststrategie ab und unterscheiden sich für den Test mit Prüfpfad, den Test mit zirkulärem Prüfpfad bzw. den Test mit Testblöcken (s. Abschnitt 2.2). Die Art der Testmustererzeugung, deterministisch oder zufällig, spielt dagegen eine geringere Rolle. In den folgenden Abschnitten werden für jede Selbstteststrategie die Grundideen zur Integration des Selbsttests in eine Schaltung anhand verschiedener existierender Verfahren beleuchtet.

2.4.1 Entwurf für den Test mit Prüfpfad

Die Idee, einen Prüfpfad in eine Schaltung für Testzwecke einzubauen, resultierte ursprünglich aus den wachsenden Problemen beim externen Test von immer komplexer werdenden Schaltwerken. Bei diesen Schaltungen ist es oft sehr schwierig, einen Fehler tief im Inneren der Schaltung allein über die Schaltungseingänge zu aktivieren und ein fehlerhaftes Verhalten an den Schaltungsausgängen zu beobachten. Auch mit automatischen Werkzeugen zur Bestimmung deterministischer Testmuster für Schaltwerke lassen sich dann für viele Fehler keine Testmuster mehr in akzeptabler Zeit berechnen, selbst wenn nur das einfache Haftfehlermodell zugrunde gelegt wird.

Durch den Prüfpfad werden im Testmodus quasi zusätzliche Schaltungseingänge und -ausgänge geschaffen, da alle Flipflops des Prüfpfads beliebig beschrieben und ausgelesen werden können. Werden alle Flipflops einer Schaltung in einem Prüfpfad vereinigt, wird die Schaltung während des Tests zu einem Schaltnetz. Deterministische Testmuster lassen sich dann leicht berechnen. Die Verbindung aller Flipflops einer Schaltung ist jedoch sehr teuer, meist wird deshalb nur ein Teil der Flipflops zu einem sogenannten partiellen Prüfpfad zusammengefaßt. Ziel ist es, die Schaltung mit einem möglichst kurzen und damit kostengünstigen Prüfpfad leicht testbar zu machen. Die Frage ist, welche Flipflops ausgewählt werden müssen, so daß eine Schaltung leicht testbar wird.

Dazu wurden empirische Untersuchungen durchgeführt, und es wurde festgestellt, daß der Berechnungsaufwand zur Bestimmung eines Testmusters bei Schaltwerken exponentiell mit der Anzahl von Flipflops innerhalb der Zyklen zunimmt [ChAg90]. Das Ziel vieler Verfahren ist es daher, mit möglichst wenigen Scan-Flipflops alle Zyklen während des Tests aufzubrechen, um eine azyklische Schaltung zu erhalten.

In [DePo94a] werden werterhaltende Operationen (z.B. Addition mit 0) in die algorithmische Verhaltensbeschreibung eingefügt, damit bestimmte Variablen demselben Register zugewiesen werden können. Die Variablen sind dabei so ausgewählt, daß später durch das Register Zyklen auf Register-Transfer-Ebene aufgebrochen werden können. Bei der eigentlichen Registerzuweisung werden die Variablen dann so auf Register verteilt, daß möglichst wenige Register zu Scan-Registern ausgebaut werden müssen, um alle Zyklen aufzubrechen [DePR93, LeJW93]. Ist die Register-Transfer-Struktur bereits gegeben, lassen sich mit dem Verfahren in [StCD93] Register zum Aufbrechen aller Zyklen auswählen, so daß der Hardware-Mehraufwand möglichst klein bleibt. Die Verfahren in [ChAg90] und [LeRe90] betrachten Zyklen auf Gatterebene. Zyklen mit nur einem Flipflop werden außer acht gelassen, da diese nur geringe Probleme bei der Testmusterberechnung verursachen.

Neben dem Aufbrechen von Zyklen existiert auch eine Reihe anderer Ansätze, um die Zahl benötigter Scan-Flipflops gering zu halten. Beispielsweise lassen sich nur unvollständig spezifizierte Verzweigungskonstrukte (IF, CASE) in der algorithmischen Verhaltensbeschreibung für Testzwecke erweitern, um die *Steuerbarkeit* und *Beobachtbarkeit* einer Schaltung allgemein zu erhöhen [VTAA93]. Ein Moduleingang ist dabei vollkommen steuerbar, wenn er sich über die Schaltungseingänge auf beliebige Werte einstellen läßt. Ein Modulausgang ist vollkommen beobachtbar, wenn sich eine Änderung des Ausgangswertes immer auch an einem Schaltungsausgang bemerkbar macht. In [Gu95] wird eine Schaltung auf Register-Transfer-Ebene anhand von Testbarkeitsmaßen in Partitionen unterteilt, wobei nur die Register an den Eingängen und Ausgängen einer Partition zu Scan-Registern oder alternativ zu Testregistern ausgebaut werden müssen. Das Verfahren in [BhDS97] arbeitet ebenfalls auf der Register-Transfer-Ebene. Es bestimmt alle Register, die allein durch eine geeignete Musterfolge an den Schaltungseingängen mit beliebigen Werten geladen werden können, bzw. deren Werte sich zu den Schaltungsausgängen propagieren lassen. Dabei wird vorausgesetzt, daß alle Register einen Resetzustand besitzen und Multiplexer über ein zusätzliches Signal, das zwischen Normal- und Testbetrieb unterscheidet, angesteuert werden. In [AbKR91] werden Flipflops auf Gatterebene anhand der Anzahl der Fehler sortiert, die unentdeckt blieben, wenn das Flipflop als nicht steuerbar und unbeobachtbar betrachtet werden würde. Flipflops mit großer Anzahl solcher Fehler werden bevorzugt in einen Prüfpfad aufgenommen. Stattdessen analysiert das Verfahren in [ChPa91c] die Stellen, an denen keine Testmuster für Fehler durch ein ATPG-Werkzeug berechnet werden konnten.

2.4.2 Entwurf für den Test mit zirkulärem Prüfpfad

Beim zirkulären Prüfpfad ist der Prüfpfadinhalt Testmuster und gleichzeitig auch Signatur. Mit jedem Takt wird der Inhalt um eine Stelle weitergeschoben und mit der Testantwort desselben Taktes verknüpft. Da aufeinanderfolgende Testmuster beim zirkulären Prüfpfad stark miteinander korreliert sein können, besteht dann allerdings die Gefahr, daß zuwenige verschiedene Testmuster erzeugt werden, um die Schaltung ausreichend zu testen. Außerdem kann die Fehlermaskierungswahrscheinlichkeit zu groß werden. Ziel der Verfahren zur Integration eines zirkulären Selbsttest ist es daher, solche Korrelationen zu vermeiden.

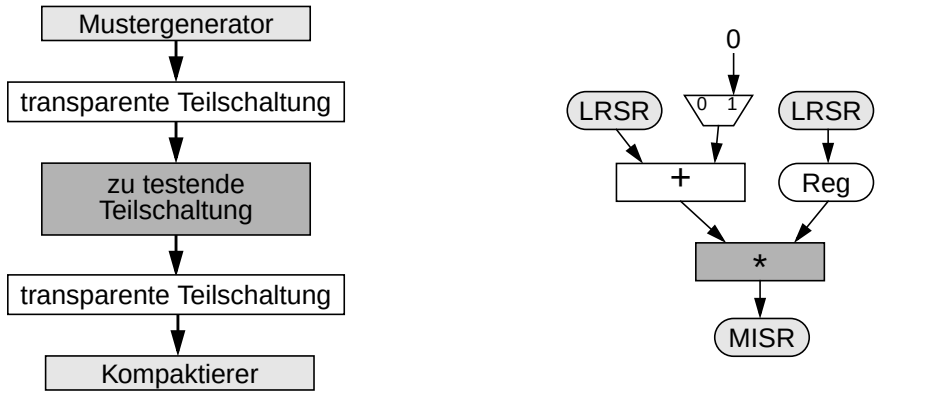
In [CaPa95a] werden zwei Korrelationsmechanismen auf Register-Transfer-Ebene betrachtet (*adjacency* und *shifting)*. Beide Arten rühren von der Nachbarschaft zweier Flipflops im Prüfpfad her. Die Korrelationen werden vermieden, indem die Reihenfolge ganzer Register im Prüfpfad und

ebenso die Schieberichtung für jedes Register geeignet gewählt wird. Um nicht alle Register in den zirkulären Prüfpfad aufnehmen zu müssen, werden in [CaPa95b] die Testbarkeitsmaße *Zufälligkeit*, *Zustandsabdeckung* und *Transparenz* mithilfe eines Markov-Ketten-Modells berechnet. Register müssen nur dann aufgenommen werden, wenn die Testbarkeitsmaße einen vorgegeben Schwellwert unterschreiten. Das Verfahren eignet sich auch zum Plazieren von Testregistern.

In [AvMc93] werden die Eingangsfunktionen aller Scan-Flipflops auf Korrelationen hin überprüft und die Art der Korrelation klassifiziert. Bei geeigneten Klassen wird die Scan-Funktion eines Flipflops, d.h. die XOR-Verknüpfung, so verändert, daß eine Fehlermaskierung dort nicht mehr auftreten kann. Das Verfahren arbeitet überwiegend auf Gatterebene, teilweise jedoch auch auf Register-Transfer-Ebene.

2.4.3 Entwurf für den Test mit Testblöcken

Beim Test mit Testblöcken wird der Eingang einer zu testenden Teilschaltung durch einen Mustergenerator gespeist, während die Testantworten von einem Kompaktierer aufgenommen werden. Mustergenerator und Kompaktierer erzeugen mit jedem Takt ein neues Testmuster bzw. kompaktieren eine komplette Testantwort (Bild 2.22a).



a) allgemeine Struktur b) auf Register-Transfer-Ebene

Bild 2.22: Testblöcke

In der Regel liegt ein Mustergenerator oder Kompaktierer nicht unmittelbar am Eingang bzw. Ausgang der zu testenden Schaltung. Meist ist ein Pfad dazwischengeschaltet, der die Testmuster bzw. Testantworten transportiert. Ein solcher Pfad kann dabei über weitere Teilschaltungen verlaufen,

die dann das transportierte Muster jedoch nicht unzulässig verfälschen dürfen. Auf Register-Transfer-Ebene sind die zu testenden Teilschaltungen meist Module mit bekannter Funktion, z.B. Addierer, Multiplexer usw. Solche Module haben oft mehrere Eingänge, d.h. Eingabeworte. Anstatt alle Eingänge zusammen durch einen breiten Mustergenerator zu speisen, verwendet man meist mehrere schmalere Mustergeneratoren (für jedes Eingang einen) und nimmt in Kauf, daß die Länge des maximalen Musterzyklus abnimmt (s. auch Abschnitt 4.5). Ebenso verlaufen die Transportpfade über solche Module. Bild 2.22b zeigt einen Testblock auf Register-Transfer-Ebene, der einen Multiplizierer testet. Als Mustergeneratoren und Kompaktierer wurden Testregister verwendet. Der Addierer im Transportpfad zum linken Multipliziereingang mußte für den Transport durchlässig gemacht werden. Dazu wird hier sein neutraler Wert 0 über einen Schaltungseingang zu seinem *Seiteneingang* propagiert, d.h. zu dem Eingang, der nicht im Transportpfad liegt.

Das Ziel beim Test mit Testblöcken ist, diejenigen Testblöcke auszuwählen, die alle Module einer Schaltung mit möglichst wenigen Testregistern, d.h. geringem Hardware-Mehraufwand testen. Die Testablaufszeit soll ebenfalls klein gehalten werden. Dabei geht man üblicherweise davon aus, daß alle Module durch zufällige Musterfolgen mit akzeptabler Länge getestet werden können. Viele Verfahren betrachten zudem Register und Multiplexer nicht explizit als zu testende Module, da diese mit sehr wenigen Mustern zu testen sind und meist beim Test der anderen Module mitgetestet werden. Ist dies bei der gefundenen Lösung doch nicht der Fall, werden sie im nachhinein gesondert berücksichtigt.

In den folgenden Abschnitten werden zum einen existierende Verfahren zur Plazierung von Testregistern vorgestellt, zum anderen Verfahren, die gezielt die Testablaufszeit optimieren. Zunächst soll jedoch ein Spezialfall beim Test mit Testblöcken betrachtet werden, bei dem Testregister nur an den Schaltungseingängen bzw. Schaltungsausgängen integriert werden. Diese Verfahren sind ursprünglich für den externen Test mit deterministischen Mustern für das „Test pro Takt“-Schema entwickelt worden, lassen sich jedoch auch für den Test mit zufälligen Mustern einsetzen.

2.4.3.1 Testregister an den „Schaltungsrändern“

Werden Testregister nur an den Schaltungseingängen bzw. -ausgängen integriert, also an den Schaltungsrändern, müssen die Eingänge aller zu testenden Module über die Schaltungseingänge beliebig eingestellt und die Modulausgänge an den Schaltungsausgängen beobachtet werden können. Es gilt also, geeignete Transportpfade zwischen den Schaltungsrändern und den Modulen zu konfigurieren. Viele Verfahren berücksichtigen dies bereits während der High-Level-Synthese.

In [LWJA92] werden Variablen den Registern so zugewiesen, daß jedes Register eine Variable eines Schaltungseingangs bzw. -ausgangs speichert. Der Effekt ist, daß nach der Synthese jedes Register direkt mit einem Schaltungseingang bzw. -ausgang verbunden ist; die Transportpfade werden dadurch sehr einfach und kurz und bestehen i.a. nur aus einem Register und einem Multiplexer. Variablen von Schaltungseingängen bzw. -ausgängen sind dabei die Werte, die an Schaltungseingängen angelegt bzw. an Schaltungsausgängen abgelesen werden, und damit sind sie vollkommen steuerbar bzw. beobachtbar. Somit sind auch alle Register vollkommen steuerbar bzw. beobachtbar, wodurch die Testbarkeit der Schaltung gesteigert wird, d.h. die Schaltung ist leichter zu testen. In [LeWJ92] wird die Ablaufplanung so gestaltet, daß durch die Variablenzuweisung von [LWJA92] möglichst viele Variablen vollkommen steuerbar bzw. kontrollierbar werden. Das Verfahren in [HsRP96] betrachtet Verzweigungskonstrukte wie IF, WHILE und FOR, bei denen sich die Verzweigung nur schwer über Schaltungseingänge beeinflussen läßt. Zweige, die nur schwer ausgewählt werden können, sind meist auch schwer zu testen. Durch ein zusätzliches Steuersignal kann eine Verzweigung im Testbetrieb gezielt gewählt werden, wodurch die Testbarkeit steigt.

In [ChSa92] und [ChKS94] werden die Variablen eines Steuer-Datenflußgraphen in vollständig steuerbare (beobachtbare*)* und nicht vollständig steuerbare (beobachtbare*)* Variablen eingeteilt. Die algorithmische Verhaltensbeschreibung wird dann so verändert, daß im Testbetrieb jeder nicht vollkommen steuerbaren (beobachtbaren) Variablen der Wert einer vollkommen steuerbaren (beobachtbaren) Variablen zugewiesen wird. Damit wird jede Variable im Testbetrieb über Schaltungseingänge vollkommen steuerbar bzw. kann an Schaltungsausgängen beobachtet werden, und zwar unabhängig von der anschließenden Schaltungssynthese.

In [BhJh94] und [FIHR95] werden gezielt Transportpfade zwischen Schaltungseingängen bzw. -ausgängen und den zu testenden Modulen konfiguriert. Die Zuweisung von Operationen bzw. Variablen wird so gestaltet, daß für jeden Eingang bzw. Ausgang eines zu testenden Moduls ein solcher Transportpfad existiert. Notfalls wird im Steuer-Datenflußgraphen eine zusätzliche Verbindung zu einem Schaltungseingang bzw. -ausgang hinzugefügt.

Das Verfahren von [BhJh94] wurde auch auf Register-Transfer-Ebene übertragen und an verschiedene Schaltungstypen angepaßt [GhRJ96...99, GhJB98]. Andere Verfahren zur Konfigurierung von Transportpfaden auf Register-Transferebene wurden in [DePo94b], [RaLJ98] und [PaBa97] vorgestellt. Ein Verfahren, das die Steuerbarkeit bzw. Beobachtbarkeit durch zusätzliche Multiplexer auf Gatterebene erhöht ist in [CRBP93] beschrieben.

2.4.3.2 Testregisterplazierung innerhalb der Schaltung

Im Gegensatz zum Test mit Testregistern an den Schaltungsrändern, bei dem die Testregister bereits vorgegeben sind und der Transport von Testmustern bzw. Testantworten zu bzw. von den internen Modulen das eigentliche Problem darstellt, spielen Transportpfade bei der Plazierung von Testregistern im Inneren der Schaltung eine untergeordnete Rolle. Da alle internen Register zu Testregistern erweitert werden können, stehen i.a. viele sehr einfach gestaltete Transportpfade zur Verfügung. Die Schwierigkeit besteht darin, eine Plazierung zu finden, die mit einer möglichst geringen Zahl von Testregistern auskommt. Da sich die verschiedenen Testregistertypen im Hardware-Mehraufwand unterscheiden, versucht man zudem, teure Typen wie CBILBOs oder transparente Testregister zu vermeiden.

Wie beim Test mit Prüfpfad verursachen Zyklen Schwierigkeiten. Diesmal sind es jedoch die Zyklen mit nur einem Register. Dann besteht nämlich die Möglichkeit, daß ein Eingang und ein Ausgang eines Moduls nur über ein und dasselbe Register zu steuern bzw. zu beobachten sind. Dazu muß das Register zu einem transparenten Testregister, oft sogar zu einem CBILBO erweitert werden, da andernfalls ein ausreichender Test des Moduls aufgrund von Korrelationen nicht mehr garantiert ist. Einige Verfahren versuchen deshalb schon während der Zuweisung von Operationen und Registern bei der High-Level-Synthese solche Zyklen zu vermeiden [Avra91, MuSJ92].

Andere Verfahren konzentrieren sich auf die Anzahl der Testregister. In [KiTH98] werden die Variablen so den Registern zugeordnet, daß die Zahl der Testregister bei vorgegebener Anzahl von Testblöcken minimal wird. Dabei werden nur Testblöcke betrachtet, bei denen die Transportpfade allenfalls über einen Multiplexer führen. Derselben Einschränkung unterliegen die Verfahren in [PaGB98a/b]. Dort werden werterhaltende Operationen, sogenannte I-Nodes, in den Steuer-Datenflußgraphen eingefügt [PaGB98a], bzw. die Zuweisung von Operationen und Variablen wird so gesteuert, daß die Chancen auf eine geringe Anzahl letztendlich benötigter Testregister steigen [PaGB98b]. In [PaCH91] werden Schaltungen so synthetisiert, daß sie sich allein aus sogenannten *Testable Functional Blocks (TFB)* zusammensetzen. Ein solcher TFB besteht aus einer ALU mit Registern an jedem Eingang und einem Register am Ausgang, wobei alle Register verschieden sein müssen, damit auf CBILBOs verzichtet werden kann. Zwischen den Registern und der ALU dürfen Multiplexer liegen. Jeder TFB kann zu einem Testblock ausgebaut werden, der seine ALU testet. In [HaPa93] wurden die Einschränkungen etwas gelockert und die TFB zu *extended TFB (XTFB)*

erweitert. Hier dürfen mehrere Register vom Ausgang der ALU gespeist werden, und nur eines dieser Register muß sich von allen Registern an den Eingängen unterscheiden.

Die Verfahren in [AbBr85], [BhPa89], [KiTH91] und [LiNB91] plazieren Testregister auf Register-Transfer-Ebene. Sie erlauben Transportpfade, die über mehrere Register, Multiplexer und Busse führen, sogenannte *I-Paths* [AbBr85], die ein Muster unverändert weiterleiten können.

Die bisher vorgestellten Methoden zur Testregisterplazierung betrachten immer nur ein zu testendes Modul pro Testblock. In vielen Fällen können die Testantworten eines Moduls jedoch auch als Testmuster für ein anderes Modul weiterverwendet werden. Testregister lassen sich dann einsparen. In [JoBa86] wird das Konzept der *translucency* (Durchsichtigkeit) eingeführt. Es zieht in Betracht, daß Abweichungen einer Musterfolge an einem Eingang eines Addierers oder Subtrahierers immer auch am Ausgang sichtbar werden, sofern die Musterfolge am anderen Eingang bekannt ist. Im Gegensatz zu den I-Pfaden, die ein Muster unverändert transportieren, genügt es bei durchsichtigen Pfaden, daß sie für Eingangsänderungen durchlässig sind. In [ThAb89] werden die Testbarkeitsmaße *Zufälligkeit* und *Beobachtbarkeit* verwendet. Hier wird berücksichtigt, daß bei Modulen wie Multiplizierern die Zahl möglicher Ausgangsmuster nicht der maximalen Anzahl denkbarer Ausgangsbelegungen entsprechen kann und nicht alle Eingangsänderungen am Ausgang eines Pfades registrierbar sind. Dies wird als Reduzierung der Zufälligkeit bzw. Beobachtbarkeit beschrieben. Testblöcke werden auf Register-Transfer-Ebene so gebildet, daß die Zufälligkeit von Testmustern bzw. die Beobachtbarkeit von Testantworten einen bestimmten Schwellwert nicht unterschreiten. In [ChPa91a] wird zusätzlich eine eventuelle Beeinträchtigung der Testlänge mit einbezogen und in [ChPa91b] bereits während der High-Level-Synthese berücksichtigt.

Alle oben beschriebenen Methoden verwenden Testblöcke, bei denen die Steuersignale von Multiplexern während des Testbetriebs konstant gehalten werden und Register in jedem Taktzyklus einen neuen Wert laden. Alle Transportpfade sind somit während des gesamten Testbetriebs komplett durchgeschaltet. In [RaJL99] werden die Pfade hingegen aus Teilpfaden zusammengesetzt, wobei Multiplexer zwischen verschiedenen Teilpfaden hin- und herschalten können. Ferner dürfen Register einen Wert über mehrere Takte speichern. Auch werden Transportpfade betrachtet, bei denen Addierer und Subtrahierer durch konstante Werte an den Seiteneingängen durchlässig gemacht werden, was dem Konzept der *translucency* [JoBa86] entspricht. Für diese konstanten Werte werden eigene Transportpfade vorgesehen. Alles in allem wird so die Zahl möglicher Pfade auf Register-Transfer-Ebene stark erhöht, was sich in einer Verringerung der Testregisterzahl auszahlt.

2.4.3.3 Testablaufplanung

Von der Wahl der Testblöcke hängt es ab, wie lang der Test aller Module letztlich dauert. Der Test wird dazu in einzelne Testsitzungen aufgeteilt, die nacheinander ausgeführt werden. Manche Testblöcke können gemeinsam innerhalb einer Testsitzung arbeiten, ohne Konflikte auf den Steuer- bzw. Datenleitungen zu verursachen. Sie sind dann miteinander kompatibel. Durch die parallele Abarbeitung wird Testzeit eingespart. Nicht miteinander kompatible Testblöcke müssen hingegen verschiedenen Testsitzungen zugeordnet werden. Je nachdem wie lange die einzelnen Testblöcke zum Test ihrer Module brauchen, kann die Dauer der verschiedenen Testsitzungen variieren. Ziel der Testablaufplanung ist es, die Testblöcke so auf Testsitzungen aufzuteilen, daß die gesamte Testdauer minimal wird.

Einige Verfahren gehen davon aus, daß die Testblöcke alle gleichlang aktiviert werden. Die Testdauer hängt dann nur von der Anzahl der Testsitzungen ab. In [HaOr94a] und [VaOr95] wird z.B. die High-Level-Synthese derart gesteuert, daß die Chancen steigen, miteinander kompatible Testblöcke zu erhalten. Die Transportpfade innerhalb der Testblöcke sind dabei während des gesamten Testbetriebs durchgeschaltet. Zwei Testblöcke sind also nicht kompatibel, wenn sie verschiedene Pfade über einen gemeinsamen Multiplexer schalten. In [OrHa93] werden solche Testblöcke auf Register-Transfer-Ebene erzeugt und auf möglichst wenige Testsitzungen verteilt. Hebt man die Einschränkung der konstanten Ansteuerung von Multiplexern auf, kann ein Multiplexer mehrere Teilpfade abwechselnd schalten. Dann können jedoch nicht in jedem Takt neue Testmuster an die Module angelegt werden. Durch den geringeren Durchsatz an Testmustern steigt die Testdauer für einen Testblock. In [HaOr94b] wird deshalb die Ansteuerung eines Testblocks so bestimmt, daß der Durchsatz an Testmustern möglichst groß ist. In [ChYu92] und [JoPP89] wird ein möglichst kurzer Testablaufplan für Testblöcke mit unterschiedlichen Aktivierungszeiten erstellt. Sie erlauben dabei, daß ein neuer Testblock aktiviert wird, während der Testbetrieb eines kompatiblen Testblocks noch nicht abgeschlossen ist. Das Verfahren in [CrKS88] läßt sogar zu, daß der Testbetrieb eines Testblocks unterbrochen wird, um ihn zu einem späteren Zeitpunkt fortzusetzen. Dadurch werden noch kürzere Testdauern möglich, allerdings steigt der Aufwand für die Teststeuerung.

2.4.4 Entwurf mit Akkumulatoren

Im Gegensatz zum Selbsttest mit Testregistern, der von vielen Werkzeugen beim Schaltungsentwurf unterstützt wird, existieren für Akkumulatoren noch recht wenige Verfahren. In [MKRT95]

wird eine Schaltung so synthetisiert, daß sie allein durch Akkumulatoren pseudoerschöpfend getestet werden kann. Um dies zu erreichen, werden Akkumulatoren bereits in den Steuer-Datenfluß-graphen eingebaut. Bei Bedarf wird dafür eine Kante vom Ausgang eines Additionsoperators zu einem seiner Eingänge hinzugefügt. Bei manchen Schaltungen werden dadurch zusätzliche Funktionseinheiten für Akkumulatoren nötig, manchmal wird auch der Verbindungsaufwand sehr groß. In solchen Fällen ist der Selbsttest mit Testregistern vorzuziehen.

Beim *ABIST*-Szenario (Arithmetic Built-In Self-Test) in [ADHA95, RaRT97] wird ein Mikroprogramm für einen digitalen Signalprozessor angegeben, mit dem der Prozessorkern allein mit Akkumulatoren getestet wird. In [GiPZ96] wird ein Multiplizierer mit pseudoerschöpfenden Mustern von LRSRs gespeist, und die Testantworten werden von einem Akkumulator aufgenommen. Es wurde gezeigt, daß bei dieser speziellen Konfiguration neben dem Multiplizierer auch der Akkumulator mitgetestet wird.

Das bisher wohl einzige Verfahren, das Akkumulatoren und Testregister miteinander kombiniert, ist *OptiBIST* [BeFR98]. Es plaziert Mustergeneratoren und Kompaktierer auf Register-Transfer-Ebene. Allerdings testen die betrachteten Testblöcke immer nur ein Modul, Transportpfade bleiben während des gesamten Testbetriebs eines Testblocks durchgeschaltet. Ferner werden nur einfache Akkumulatoren mit einem Register in der Rückkopplungsschleife berücksichtigt.

3 Konfiguration eines Selbsttests mit Akkumulatoren

In diesem Kapitel wird ein neues Verfahren vorgestellt, das Akkumulatoren für den Selbsttest des Operationswerks einer synchronen Schaltung einsetzt, um den Hardware-Mehraufwand für Testregister zu minimieren. Ausgangspunkt ist die Register-Transfer-Ebenen-Beschreibung des Operationswerks – im folgenden ist mit einer Schaltung stets ihr Operationswerk gemeint. Anhand der Beschreibung werden Akkumulatoren innerhalb der Schaltung allein durch geeignetes Ansteuern konfiguriert. Zusammen mit potentiellen Testregistern werden Testblöcke gebildet und schließlich die Blöcke ausgewählt, die alle Module der Schaltung bei minimalem Hardware-Mehraufwand und kurzer Testdauer testen. Die gefundene Lösung enthält sowohl die Information, welche Register zu welchen Testregistertypen erweitert werden müssen, als auch alle Daten, mit denen das Teststeuerwerk durch klassische Verfahren der Steuerwerkssynthese erzeugt werden kann.

Neben dem Einsatz von Akkumulatoren war es das Ziel, die Einschränkungen existierender Verfahren zur Plazierung von Testregistern auf Register-Transfer-Ebene aufzuheben, um die Möglichkeiten zur Einsparung von Testhardware besser ausnützen zu können. Im folgenden werden deshalb erst verschiedene Einschränkungen klassifiziert und ihre Auswirkungen auf die Testhardware beschrieben. Anschließend wird das neue Verfahren vorgestellt.

3.1 Einschränkungen herkömmlicher Verfahren

Die meisten Verfahren zur Testregisterplazierung auf Register-Transfer-Ebene erzeugen eine Reihe von Testblöcken und suchen dann diejenigen aus, die für einen kostengünstigen Test am besten geeignet sind. Da die Anzahl aller möglichen Testblöcke in einer Schaltung üblicherweise viel zu groß für eine umfassende Betrachtung ist, beschränkt man sich auf bestimmte Typen von Testblökken. Im folgenden werden diese Typen drei Klassen von Einschränkungen zugeteilt.

Testblöcke der ersten Klasse konzentrieren sich auf je ein zu testendes Modul. Ferner transportieren Pfade nur Testmuster oder Testantworten und müssen konstant durchgeschaltet bleiben, solange der zugehörige Testblock aktiviert ist. Auf Testblöcke dieser Klasse beschränken sich z.B. die Verfahren in [AbBr85], [BhPa89], [KiTH91] und [LiNB91] sowie das bisher einzige Verfahren, das neben Testregistern auch Akkumulatoren berücksichtigt [BeFR98].

In der zweiten Klasse können Testblöcke mehrere Module gleichzeitig testen, indem Testantworten des einen Moduls als Testmuster eines anderen Moduls weiterverwendet werden. Um dies zu erreichen, verwenden die Verfahren das Konzept der *translucency* [JoBa86] bzw. Testbarkeitsmaße wie *Zufälligkeit* und *Beobachtbarkeit* [ThAb89, ChPa91a].

In der dritten Klasse wird die Einschränkung konstant durchgeschalteter Transportpfade aufgehoben. Außerdem sind Transportpfade für konstante Werte an Seiteneingängen von Funktionseinheiten erlaubt, mit denen die Funktionseinheit für den anderen Eingang transparent wird. Das Verfahren in [RaJL99] berücksichtigt auch Testblöcke dieser Klasse.

Der Einfluß der verschiedenen Einschränkungen auf den Hardware-Mehraufwand für Testregister soll an einem Beispiel demonstriert werden. Bild 3.1a zeigt eine kleine Schaltung auf Register-Transfer-Ebene mit zwei Eingängen (Ein1, Ein2) und einem Ausgang (Aus). Sie enthält zwei Addierer ($+_1$, $+_2$), fünf Register (R1 ... R5) und einen Multiplexer mit einem konstanten Eingangs-

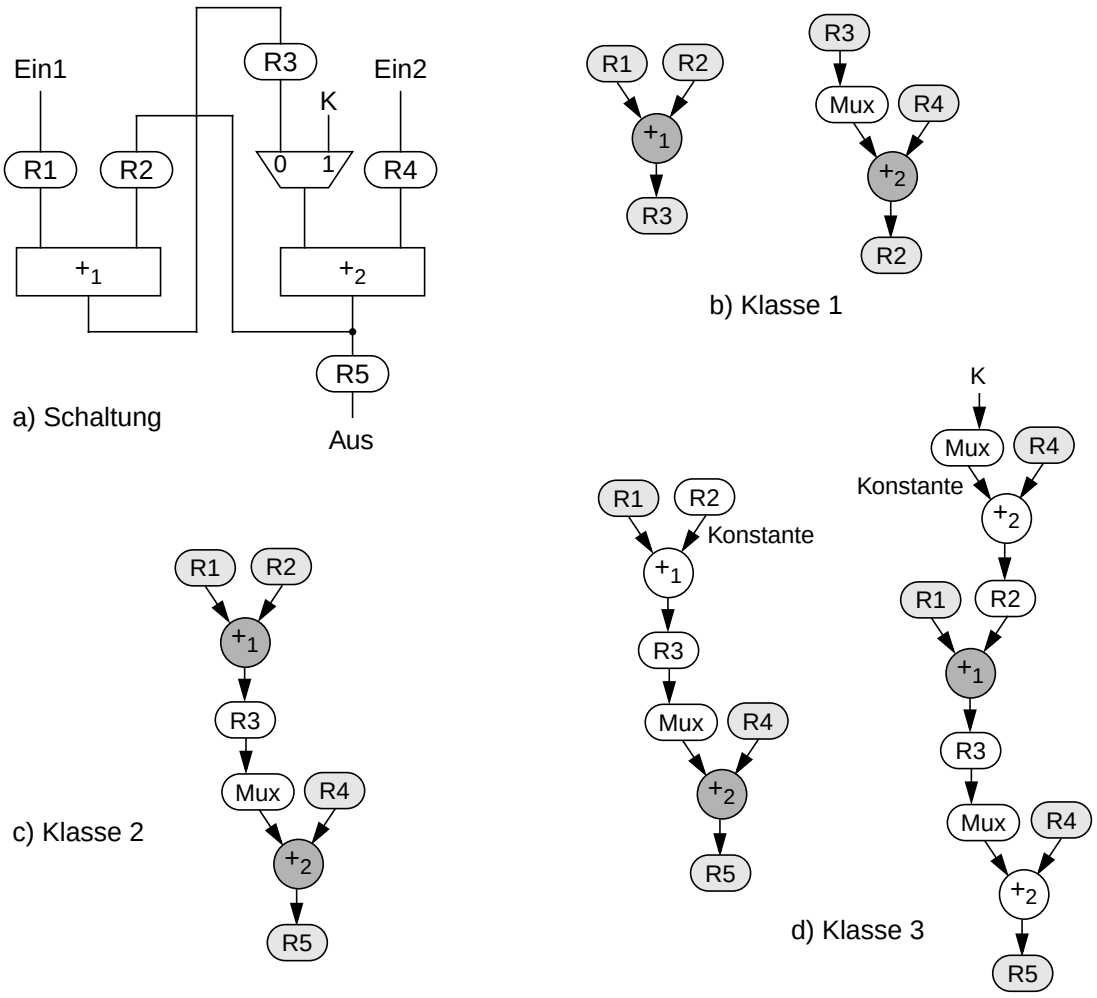


Bild 3.1: Beispielschaltung mit Datenflüssen von Testblöcken aller drei Einschränkungsklassen

wert K. Der Übersichtlichkeit halber wurden die Steuersignale der Register und des Multiplexers weggelassen. In Bild 3.1b bis d sind für jede der drei Einschränkungsklassen die Datenflüsse derjenigen Testblöcke dargestellt, die beide Addierer mit geringstem Hardware-Mehraufwand testen können. Dabei waren als Testregister nur BILBOs erlaubt sowie einfache Testregister, die außer im Normalbetrieb noch als LRSR bzw. MISR arbeiten können. Für den Rest des Kapitels sind mit LRSR und MISR immer solche Testregister gemeint.

Unter den Einschränkungen der Klasse 1 benötigt man zwei Testblöcke, wobei die Register R1 und R4 zu LRSRs erweitert werden müssen, R2 und R3 dagegen zu BILBOs (Bild 3.1b). Dürfen mehrere Module pro Testblock getestet werden, braucht R2 statt zu einem BILBO nur noch zu einem billigeren LRSR erweitert zu werden, das BILBO R3 kann durch das ebenfalls billigere MISR R5 ersetzt werden (Bild 3.1c). In Bild 3.1d wurde hingegen nur die Einschränkung auf konstant durchgeschaltete Transportpfade aufgehoben, und Pfade durften auch konstante Werte zu Seiteneingängen von Funktionseinheiten propagieren. Der linke Testblock testet beispielsweise den Addierer $+_2$, wobei der Transportpfad zwischen dem LRSR R1 und dem linken Eingang von $+_2$ über den Addierer $+_1$ verläuft. Damit die Musterfolge am Ausgang von $+_1$ zufällig bleibt, wird an den rechten Eingang von $+_1$ ein beliebiger konstanter Wert angelegt. Dazu wird das Register R2 vor der entsprechenden Testsitzung mit diesem Wert initialisiert. Der Wert bleibt dann während der gesamten Testsitzung in R2 gespeichert. Ähnlich sieht es beim zweiten Testblock aus. Hier wird der Addierer $+_2$ durch die Konstante K für die Zufallsmuster des LRSR R4 durchlässig gemacht. Die Testantworten von $+_1$ werden ebenfalls über $+_2$ geleitet. Dazu muß der Multiplexer zwischen seinen beiden Eingängen, d.h. zwischen der Konstante K und den Testantworten, zum richtigen Zeitpunkt umschalten. Die Zufallsmuster von R4 maskieren dabei keine eventuellen Fehler in den Testantworten. Zum Test beider Addierer müssen nur die Register R1 und R4 zu LRSRs und das Register R5 zu einem MISR ausgebaut werden. Der Hardware-Aufwand für Testregister ist hier minimal.

3.2 Verfahren zur Konfiguration eines Selbsttests

In diesem Abschnitt wird das neue Verfahren zum Einbau eines Selbsttests in eine synchrone Schaltung auf Register-Transfer-Ebene beschrieben. Neben dem zusätzlichen Einsatz von Akkumulatoren zeichnet es sich dadurch aus, daß die oben genannten Einschränkungen, denen andere Verfahren bei der Bestimmung von Testblöcken unterliegen, aufgehoben werden. Tatsächlich werden Testblöcke aller drei Klassen zugelassen. Konstante Werte dürfen dabei über Schaltungseingänge eingespeist oder in Registern der Schaltung gespeichert werden. Die Register müssen dazu vor der

Aktivierung eines Testblocks geeignet initialisiert werden. Im folgenden wird vorausgesetzt, daß das Teststeuerwerk alle Register, falls nötig, über die Schaltungseingänge mit den erforderten Werten initialisieren kann und ebenso die Signaturen am Ende einer Testsitzung über die Schaltungsausgänge lesen kann. Letztlich besteht die Aufgabe darin, geeignete Transportpfade zu konfigurieren - gegebenenfalls mit zusätzlicher Hardware. Diese Problematik ist wohlbekannt und kann von einer Reihe von Verfahren gelöst werden [BhDS97, GhJB98, RaLJ98]. In vielen Fällen ist dafür kein oder nur ein geringer Hardware-Mehraufwand nötig, besonders wenn man berücksichtigt, daß Testregister mit Schiebebetrieb seriell beschrieben und ausgelesen werden können.

Für den Test mehrerer Module durch einen Testblock werden Testbarkeitsmaße verwendet, die mit denen in [ChPa91b] verwandt sind. Die Transportpfade innerhalb von Akkumulatoren bzw. Testblöcken werden nicht mehr konstant durchgeschaltet, sondern können sich aus Teilpfaden zusammensetzen, die nur zu bestimmten Zeiten aktiv sind. Daher unterliegt die Ansteuerung eines Akkumulators bzw. Testblocks bestimmten Bedingungen, die die korrekte Funktionsweise garantieren. Um diese Bedingungen zu erhalten, wurde eine Methode entwickelt, die benachbarte Module zu immer komplexeren Teilschaltungen zusammensetzen kann und gleichzeitig die Ansteuerungsbedingungen für eine gewünschte Funktionsweise bestimmt. Im folgenden wird zunächst diese Methode beschrieben. Darauf werden mit ihrer Hilfe Akkumulatoren und schließlich ganze Testblöcke konfiguriert (Abschnitte 3.2.2 und 3.2.3). Anschließend werden von den gefundenen Testblöcken diejenigen ausgewählt, mit denen alle Module der Schaltung bei minimalen Testkosten getestet werden können. Es folgt ein Verfahren, das für jeden dieser Testblöcke eine konkrete Ansteuerung bestimmt, die alle Ansteuerungsbedingungen erfüllt mit dem Ziel, das Teststeuerwerk möglichst einfach zu halten. Der Abschnitt schließt mit der Diskussion experimenteller Ergebnisse.

3.2.1 Kombination benachbarter Module und Komponenten

Wie oben erwähnt, werden Akkumulatoren und Testblöcke durch schrittweises Verschmelzen benachbarter Module innerhalb einer gegebenen synchronen Schaltung gewonnen. Die Grundlage dafür bildet eine Bausteinbibliothek, in der für Standardmodule wie Register, Multiplexer, Addierer usw. verschiedene Verhaltensbeschreibungen gespeichert sind. Eine Verhaltensbeschreibung gibt dabei an, wie die Eingänge und Steuersignale eines Moduls angesteuert werden müssen, damit ein vorgegebenes Ausgangsverhalten garantiert ist. Da synchrones Verhalten vorausgesetzt wird, können sich Steuersignale sowie die Werte an den Eingängen bzw. Ausgängen von Modulen nur syn-

chron zum Systemtakt ändern, z.B. bei den ansteigenden Taktflanken. Zur Beschreibung des Verhaltens werden charakteristische Funktionen und verschiedene Werteklassen verwendet, die im Anschluß erläutert werden. Es folgen konkrete Verhaltensbeschreibungen von Standardmodulen, wie sie auch in der Bausteinbibliothek enthalten sind. Zuletzt wird auf die Verschmelzung benachbarter Module eingegangen.

3.2.1.1 Charakteristische Funktionen und Werteklassen

Das Verhalten eines Signals, d.h. eines Steuersignals, eines Signals am Moduleingang oder Modulausgang, wird durch Paare von Werten und charakteristischen Funktionen beschrieben. Ein Paar „w: F“ gibt an, daß ein Signal den Wert w annehmen soll, und zwar in Taktzyklen, die durch die Funktion F spezifiziert sind. F(t) = 1 bedeutet, daß der Wert im Zyklus t angenommen werden muß, F(t) = 0 erlaubt auch beliebige andere Werte. Ein Paar „w: ...011000100...“ an einem Ausgang O steht beispielsweise für die Musterfolge „...-ww---w--...“ von Werten w und „don’t cares“ an diesem Ausgang. Mit dem zusätzlichen Paar „q: ...100001001...“ an O erhält man die Folge „...qww--qw-q...“. Natürlich kann ein Signal immer nur einen Wert pro Taktzyklus annehmen. Bei zwei Paaren „w: F_w“ und „q: F_q“, $w \neq q$, für dasselbe Signal müssen F_w und F_q demnach *disjunkt* sein, um einen Konflikt zu vermeiden, d.h. F_w und F_q dürfen niemals gleichzeitig 1 sein. Sind die Verläufe beider Funktionen bekannt, läßt sich sofort feststellen, ob sie disjunkt sind oder nicht. Sind sie nicht bekannt, wird die Disjunktheit durch die zusätzliche Disjunktheitsbedingung „$\forall t: F_w(t) + F_q(t) < 2$“, kurz „$F_w + F_q < 2$“, erzwungen. Tatsächlich arbeitet das hier vorgestellte Verfahren zunächst mit rein symbolischen Funktionen, deren konkrete Verläufe noch nicht bekannt sind. Der Grund dafür ist, daß der Hardware-Mehraufwand für Testregister allein von der Struktur der Testblöcke abhängt, nicht jedoch von ihrer Ansteuerung. Die konkreten Verläufe werden erst für die Synthese des Teststeuerwerks interessant (s. Abschnitt 3.2.5).

Bisher wurden lediglich symbolische Signalwerte w und q verwendet. In der Tat ist es gerade bei Moduleingängen und -ausgängen nicht notwendig, alle Einzelwerte zu unterscheiden. Oft genügt es, Klassen von Werten anzugeben. Im folgenden werden die Werte bzw. Werteklassen vorgestellt, die für die Konfiguration von Akkumulatoren bzw. Testblöcken wichtig sind.

Ein Register übernimmt den Eingangswert, wenn sein *Ladesignal* aktiv (1) ist, und speichert seinen Wert, solange das Ladesignal inaktiv (0) ist. Das Auswahlsignal eines n:1-Multiplexers aktiviert einen von n Eingängen. Es kann die Werte 0 bis n-1 annehmen. Andere Steuersignale z.B. zur Auswahl der Funktionsweise einer ALU werden ähnlich behandelt.

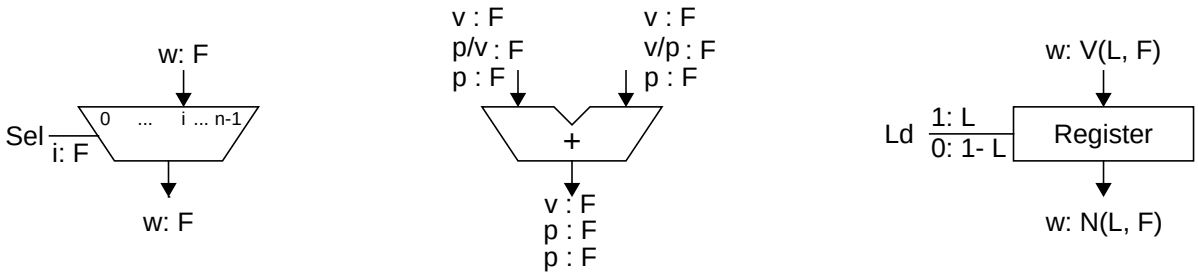
Für die Moduleingänge und -ausgänge genügen wenige Werte und Wertklassen. Ein Akkumulator generiert eine Folge von Testmustern, indem er eine weitgehend beliebige Konstante akkumuliert. Kompaktierer aktualisieren anhand der Testantworten ihre Signaturen. Wegen der Verwendung von Testbarkeitsmaßen wird zwischen Testantworten und Testmustern nicht unterschieden. In Transportpfaden werden Addierer und Subtrahierer durchlässig für Zufallsmuster, wenn an ihrem Seiteneingang eine beliebige Konstante anliegt. Sie werden sogar vollständig transparent, d.h. verändern den propagierten Wert nicht, wenn eine 0 am Seiteneingang (beim Subtrahierer am Subtrahendeneingang) anliegt. Multiplizierer und Dividierer benötigen dazu eine 1 (beim Dividierer am Divisoreingang). Andere Konstanten verzerren bei Multiplizierern und Dividierern die Muster i.a. so stark, daß sie nicht mehr zufällig genug sind. In vielen Schaltungen wird nur das obere Halbwort des Multipliziererausgangs genutzt. Der Multiplizierer kann dann nicht transparent geschaltet werden. Legt man jedoch den Wert ‘1...1‘ an, bei dem alle Bits gesetzt sind, wirkt dieser Multiplizierer wie ein Dekrementierer. Lediglich der Eingabewert 0 bleibt unverändert. In diesem Fall ist der Multiplizierer noch durchlässig genug für Zufallsmuster. Insgesamt benötigt man neben den konkreten konstanten Werten 0, 1 und ‘1...1‘ nur noch folgende Klassen:

Klasse v:	„(weitgehend) beliebige konstante Werte“
Klasse p:	„Testmuster und Testantworten“
Klasse s:	„(zwischenzeitliche und endgültige) Signaturen“.

Die Klassen können bei Bedarf in Unterklassen aufgeteilt werden, beispielsweise um zwischen Mustern von verschiedenen Mustergeneratoren zu unterscheiden oder zwischen beliebigen Konstanten, z.B. am Seiteneingang eines Addierers, und Konstanten von Akkumulatoren, die individuelle, aber noch unbekannte Werte besitzen.

3.2.1.2 Verhaltensbeschreibungen von Standardmodulen

Mithilfe der (symbolischen) charakteristischen Funktionen und der vorgestellten Werte bzw. Werteklassen läßt sich das Verhalten von Standardmodulen leicht beschreiben. Bild 3.2 zeigt einige Beispiele. Ein Multiplexer erzeugt beliebige Werte w in Taktzyklen mit F(t) = 1, wenn sie während derselben Taktzyklen am i-ten Eingang anliegen und dieser Eingang gleichzeitig durch das Auswahlsignal Sel durchgeschaltet ist (Bild 3.2a). Durch die Addition zweier Konstanten entsteht eine neue Konstante. Bei einem Testmuster p hingegen entsteht wiederum ein Testmuster, egal ob man ein anderes Testmuster oder eine Konstante hinzu addiert (Bild 3.2b).



a) Multiplexer b) Addierer c) Register

Bild 3.2: Verhaltensbeschreibungen von Multiplexer, Addierer und Register

Im Gegensatz zu rein kombinatorischen Modulen wie Multiplexern oder Addierern kann ein Register Werte speichern. Dazu beschreiben wir das Verhalten am Registereingang in Abhängigkeit vom gewünschten Ausgangsverhalten (gegeben durch die Funktion F) und dem Ladesignal Ld (Bild 3.2c). Sei L die charakteristische Funktion für das aktivierte Ladesignal, Ld = 1, und 1-L die Funktion für Ld = 0. Dann signalisiert L(t) = 1, daß der Wert, der in Taktzyklus t am Registereingang anliegt, im darauffolgenden Taktzyklus t+1 gespeichert werden wird und am Registerausgang erscheint. Bei L(t) = 0 hingegen bleibt der Ausgangswert von Zyklus t auch im Zyklus t+1 erhalten. Die Funktion V(L, F) beschreibt den Sachverhalt, daß ein Wert, der am Ausgang erscheinen soll, vorab gespeichert werden muß, und zwar wenn das Ladesignal zuletzt aktiv war (ein Beispiel folgt unten). N(L, F) gibt an, daß ein Wert solange am Ausgang anliegt, bis ein neuer Wert gespeichert wird. Ein Wert kann deshalb länger am Registerausgang erscheinen als die Funktion F fordert. Tatsächlich kann mit der Funktion V(L, F) nur das Ausgangsverhalten zu Zeitpunkten t > 0 beeinflußt werden, wobei t = 0 der erste Taktzyklus nach der Initialisierung einer Teilschaltung sei, die das Register beinhaltet. Es wird gefordert, daß F(0) = N(L, F)(0) gilt. Bei Bedarf muß das Register also mit dem entsprechenden Wert initialisiert worden sein.

Wie die Funktionen V(L, F) und N(L, F) für Zeitpunkte t > 0 bestimmt werden, macht Bild 3.3 anhand der konkreten Verläufe F = „...000100010...“ und L = „...010010100...“ deutlich.

Entsprechend der Funktion F soll ein bestimmter Wert in den Zyklen t_2 und t_5 am Ausgang erscheinen. Bezüglich L speichert das Register nur Werte, die in den Zyklen t_1, t_3 und t_4 am Registereingang anliegen. Diese Werte erscheinen folglich in den Zyklen $t_1+1 \leq t \leq t_3$, $t_3+1 \leq t \leq t_4$ bzw. $t \geq t_4+1$ am Ausgang. Damit der gewünschte Wert an den durch F vorgegebenen Zyklen am Ausgang anliegt, muß er sich offensichtlich in den Zyklen t_1 und t_4 am Eingang befinden. Die zugehörige Funktion V(L, F) zeigt Bild 3.3a, die resultierende Ausgangsfunktion N(L, F) Bild 3.3b.

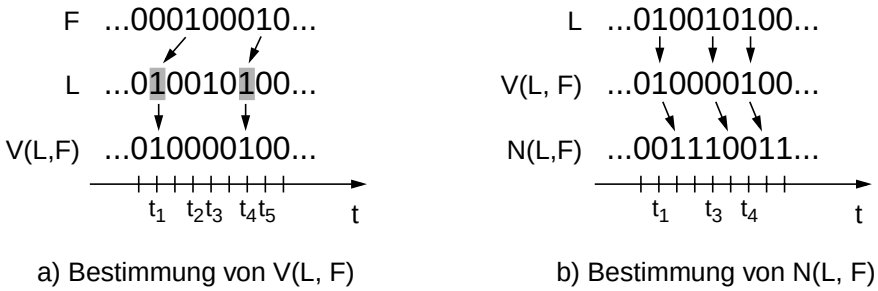


Bild 3.3: Bestimmung der Funktionen V(L, F) und N(L, F)

Das obige Registerverhalten beschreibt nur, wie Werte vom Eingang zum Ausgang des Registers propagiert werden können. Ein Register kann jedoch auch einen Initialwert dauerhaft speichern oder als Testregister arbeiten. Wird ein Wert v dauerhaft gespeichert, liegt der Wert in jedem Taktzyklus am Registerausgang an, während das Ladesignal immer deaktiviert bleibt. Am Ausgang erhält man das Verhalten „v: **1**“ und für das Ladesignal „1: **0**“, wobei immer **1**(t) = 1 und **0**(t) = 0 gilt. Ein LRSR erzeugt an seinem Ausgang fortlaufend Muster einer Musterfolge unabhängig von der Belegung am Eingang. Mit dem Ladesignal wird lediglich gesteuert, wie lange ein einzelnes Muster am Ausgang erscheint. Ein MISR verarbeitet jeden neu gespeicherten Eingangswert zu einer neuen Signatur. Es interpretiert daher Muster am Eingang als Testantworten, wann immer das Ladesignal aktiv ist. Bild 3.4 zeigt das Verhalten eines Registers zur dauerhaften Speicherung einer Konstanten, sowie das Verhalten eines LRSR bzw. eines MISR. Für ein BILBO ist ein eigenes Verhalten nicht notwendig, da es innerhalb eines Testblocks immer entweder als LRSR, MISR oder normales Register arbeitet und das entsprechende Verhalten ausgewählt werden kann.

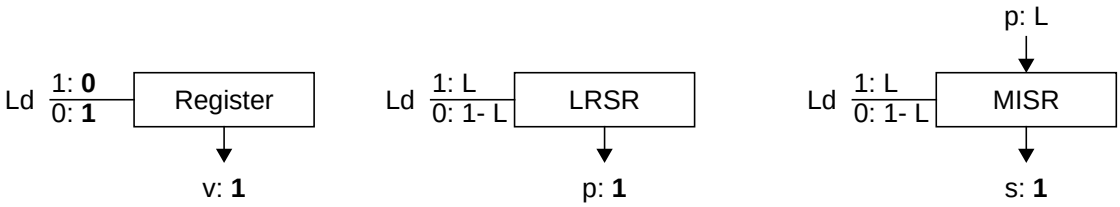


Bild 3.4: Verhalten eines Registers, das eine Konstante v dauerhaft speichert, eines LRSR und eines MISR

Ein Modul kann nur getestet werden, wenn alle seine Eingänge mit voneinander unabhängigen Testmusterfolgen gespeist werden. Das entsprechende *Testverhalten* erzeugt demnach Testmuster p am Modulausgang durch andere Testmuster p an allen Eingängen. Viele Module sind auch in der Lage, Konstanten und Zufallsmusterfolgen zu propagieren, ohne dabei selbst getestet zu werden. Register und Multiplexer verändern dabei den propagierten Wert nicht. Arithmetische und logische

Module lassen sich oft durch eine bestimmte Konstante transparent schalten, so daß auch sie den propagierten Wert unverändert lassen. Zufallsmuster und Testantworten brauchen nicht unverändert weiterpropagiert zu werden. Wichtig ist nur, daß sich Änderungen dieser Muster in den meisten Fällen auch nach der Propagierung bemerkbar machen (s. auch *Zufälligkeit* und *Beobachtbarkeit* in Abschnitt 3.2.3.1). Manche arithmetischen Funktionen besitzen dafür ein zusätzliches, *durchlässiges* Propagierungsverhalten. Arithmetische und logische Funktionen können Konstanten erzeugen, indem sie zwei andere Konstanten miteinander verknüpfen. Solange der Wert der Konstanten am Ausgang beliebig sein darf, spielt der Wert der Konstanten an den Eingängen keine Rolle. Auch die speziellen Werte 0, 1 und '1...1' lassen sich meist durch viele Kombinationen von Konstanten erzeugen, aber nicht durch alle. Da in Teilschaltungen die Eingangswerte selbst zusammengesetzt sein können, ist es möglich, daß sich aufgrund von Abhängigkeiten zwischen den verschiedenen Konstanten keine gültige Kombination mehr einstellen läßt. Das gleiche gilt für die konstanten Werte von Akkumulatoren. Diese müssen so gewählt werden können, daß ein maximalperiodischer Musterzyklus entsteht. Außerdem hängt von ihrer Wahl die erzeugte Musterfolge und damit die Fehlererfassung bzw. Testlänge ab. Die Einstellmöglichkeiten sollten daher nicht zu sehr eingeschränkt werden. Das hier vorgestellte Selbsttestverfahren läßt grundsätzlich ein Modulverhalten zu, bei dem eine Konstante aus zwei beliebigen anderen Konstanten zusammengesetzt wird. Eventuell unzulässige Abhängigkeiten zwischen Konstanten werden beim Verschmelzen von Modulen überprüft.

Tabelle 3.1 zeigt die verschiedenen Verknüpfungsmöglichkeiten beim Propagierungsverhalten typischer arithmetischer und logischer Standardmodule. Es wurde dabei von rein kombinatorischen Schaltungen ausgegangen; die charakteristischen Funktionen an den Ein- und Ausgängen sind somit identisch und wurden weggelassen.

	Add.	Sub.	Mul.	Mul^O	Div.	Div^R	UND	ODER	XOR
transparente Verknüpfungen	w/0 w 0/w w	w/0 w	w/1 w 1/w w	–	w/1 w	–	w/'1..1' w '1..1'/w w	w/0 w 0/w w	w/0 w 0/w w
durchlässige Verknüpfungen	w/v w v/w w	w/v w v/w w	v/v v	'1..1'/w w w/'1..1' w v/v v	v/v v	w/'1..1' w v/v v	–	–	w/v w v/w w

Tabelle 3.1: Verknüpfungen beim Propagierungsverhalten typischer Module

Ein transparentes Verhalten „x/y z“ bedeutet, daß ein Wert x am linken Eingang des Moduls mit einem Wert y am rechten Moduleingang zu einem Wert z am Ausgang verknüpft werden kann. Bei

lediglich durchlässigem Verhalten, stehen x, y, und z für bestimmte Werteklassen. Der Wert w steht stellvertretend für die Werteklassen v, p und s, d.h. für Konstanten, Testmuster (-antworten) und Signaturen. Bei Subtrahierern und Dividierern wurde der Subtrahend bzw. Divisor am rechten Eingang angenommen. Mul^O steht für den Multiplizierer bei dem nur die obere Hälfte des Ausgangs genutzt wird. Div^R ist die Division, bei der nur der ganzzahlige Rest betrachtet wird.

Nicht alle Standardmodule sind rein kombinatorische Schaltungen. Multiplizierer und vor allem Dividierer benötigen oft mehrere Taktzyklen bis ihre Berechnung abgeschlossen ist. Manchmal ist zusätzlich eine Pipeline eingebaut. Eine einzelne Berechnung dauert dann zwar immer noch mehrere Zyklen, allerdings kann oft schon in jedem Zyklus mit einer neuen Berechnung begonnen werden. Auch für solche Module läßt sich ein Verhalten leicht angeben. Bild 3.5 zeigt das Testverhalten zweier Multiplizierer, die das Ergebnis erst mit einem Taktzyklus Verzögerung liefern.

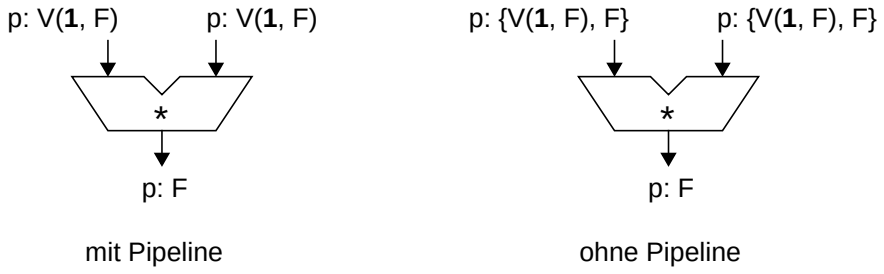


Bild 3.5: Multiplizierer mit einem Taktzyklus Verzögerung

Der linke Multiplizierer besitzt eine Pipeline, die Eingangsbelegungen dürfen mit jedem Taktzyklus wechseln. Beim rechten Multiplizierer ohne Pipeline müssen die Eingangsbelegungen für mindestens zwei Zyklen, V(**1**, F) und F bzw. kurz {V(**1**, F), F}, stabil anliegen. Tatsächlich besagt „p: {V(**1**, F), F}“ nur, daß während zwei aufeinanderfolgender Zyklen Testmuster anliegen, nicht jedoch, daß diese Muster identisch sind. Dies läßt sich i.a. erst entscheiden, wenn die Teilschaltung bekannt ist, in der der Multiplizierer letztlich eingesetzt wird. Durch geeignete Wahl der Signalverläufe bei der Ansteuerung kann man dann garantieren, daß ein und dasselbe Muster für mehrere Taktzyklen an einem Eingang bzw. Ausgang anliegt.

3.2.1.3 Kombinationsmechanismus

Mithilfe der Verhaltensbeschreibungen lassen sich benachbarte Komponenten, die ein oder mehrere Module beinhalten können, systematisch zu komplexeren Komponenten kombinieren. Vorausset-

zung ist, daß das Verhalten der benachbarten Komponenten bekannt ist. An den Eingang einer gegebenen Komponente A mit gewünschtem Verhalten wird ein Modul B folgendermaßen hinzugefügt: Man wählt für B ein Verhalten aus der Bausteinbibliothek, das den Wert liefert, der am Eingang von A angelegt werden soll. Die ursprüngliche Funktion F am Ausgang von Modul B ersetzt man dann einfach durch die Funktion, die am Eingang von A gefordert wird. Auf diese Weise läßt sich das Verhalten von B an den Kontext von A anpassen. Analog läßt sich natürlich statt eines Moduls B eine vorab zusammengesetzte Komponente B an A anpassen, sofern der Ausgangswert von B mit dem Eingangswert von A kompatibel ist. Bild 3.6 demonstriert als Beispiel die Kombination eines Registers mit einem 2:1 Multiplexer am Registereingang.

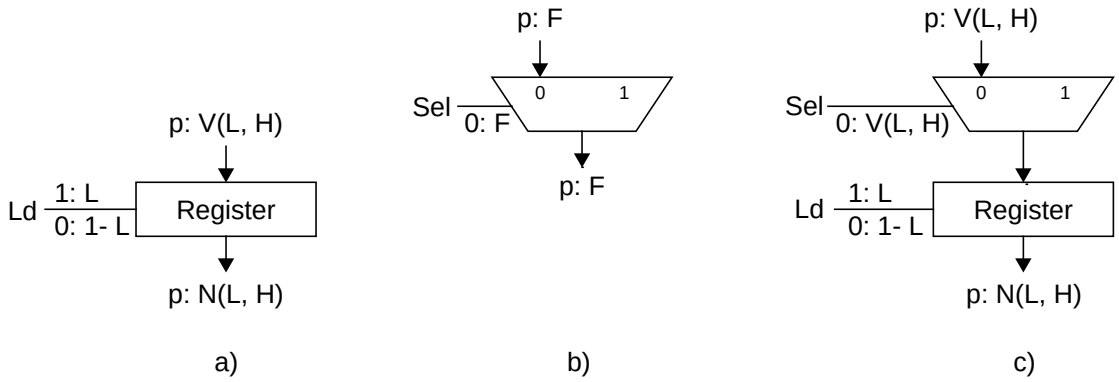


Bild 3.6: Kombination eines Registers mit einem Multiplexer

Das Register soll Testmuster p an seinem Ausgang bezüglich einer charakteristischen Funktion H erzeugen. Die Funktion V(L, H) spezifiziert somit die Taktzyklen, in denen die Muster am Registereingang anliegen müssen (Bild 3.6a). Die Muster sollen nun über den Eingang 0 des Multiplexers zum Registereingang propagiert werden. Dazu wählt man das entsprechende Multiplexerverhalten aus der Bausteinbibliothek (Bild 3.6b). Indem man die Funktion F des Multiplexers durch V(L, H) ersetzt, erhält man die Ansteuerung der zusammengesetzten Komponente (Bild 3.6c).

3.2.2 Konfigurierung von Akkumulatoren

Mithilfe der Kombination benachbarter Module lassen sich komplexe Komponenten zusammensetzen. Dabei muß natürlich die Struktur der gewünschten Komponente berücksichtigt werden. Bild 3.7 zeigt nocheinmal die Struktur eines Akkumulators.

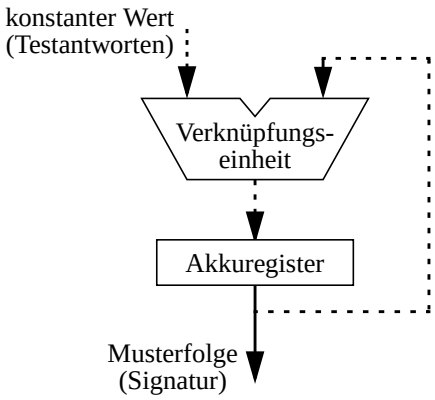


Bild 3.7: Akkumulatorstruktur

Das Grundgerüst des Akkumulators bilden die *arithmetische Verknüpfungseinheit* – im folgenden kurz *Verknüpfungseinheit* genannt – und das *Akkuregister*. Die Verknüpfungseinheit ist die arithmetische Einheit, die ein Testmuster bzw. eine Signatur mit einer Konstanten bzw. einer Testantwort zu einem neuen Testmuster bzw. einer neuen Signatur verknüpft. Im Akkuregister wird das neue Testmuster bzw. die neue Signatur gespeichert. Der Ausgang der Verknüpfungseinheit ist durch einen Propagierungspfad über das Akkuregister auf einen geeigneten Eingang der Verknüpfungseinheit rückgekoppelt. Der Einfachheit halber wird gefordert, daß dieser Rückkopplungspfad vollständig transparent ist, was prinzipiell nicht notwendig ist. Jedoch wird dadurch das spätere Einstellen einer spezifischen Musterfolge für eine kurze Testlänge bzw. die Simulation zur Bestimmung der Fehlererfassung vereinfacht. Außerdem werden nur Rückkopplungspfade betrachtet, die selbst keine zusätzlichen Zyklen beinhalten, d.h. über jedes Modul höchstens einmal verlaufen. Die Rückkopplung kann ferner mehrere Register enthalten, wobei jedes Register Akkuregister sein könnte. Akkumulatoren, die sich nur in der Wahl des Akkuregisters unterscheiden, sind jedoch von der Struktur und ihrem Verhalten her gleichwertig und müssen nicht gesondert betrachtet werden. Das Akkuregister wird deshalb als das erste Register definiert, das der Verknüpfungseinheit im Rückkopplungspfad folgt. Der Ausgang des Akkuregisters wird gleichzeitig als Ausgang des Akkumulators betrachtet. An ihm sollen Testmuster p bzw. Signaturen s entsprechend einer vorgegebenen charakteristischen Funktion H erscheinen.

Die wesentliche Aufgabe bei der Konfigurierung eines Akkumulators besteht darin, eine geeignete Verknüpfungseinheit mit passendem Akkuregister innerhalb einer Schaltung zu finden und je einen transparenten, zyklenfreien Pfad zwischen dem Ausgang der Verknüpfungseinheit und dem Akkuregistereingang bzw. zwischen dem Akkuregisterausgang und dem Rückkopplungseingang der Verknüpfungseinheit zu konfigurieren, der Testmuster p bzw. Signaturen s transportieren kann. Der

hier vorgestellte Algorithmus bestimmt zunächst eine mögliche Verknüpfungseinheit und wählt ein Modul mit transparentem Verhalten an deren Ausgang aus. Dieses Modul bildet den Anfang eines transparenten Propagierungspfades, der schrittweise verlängert wird. Dazu wird er jeweils mit einem Modul an seinem Ausgang verschmolzen, für das ebenfalls ein transparentes Verhalten ausgewählt wurde. Diese Vorgehensweise wird solange fortgesetzt, bis der Propagierungspfad mit einem Register endet. Dieses Register ist das Akkuregister und erhält das gewünschte Ausgangsverhalten „p: H“ („s: H“). Eventuell erforderliche Konstanten an Seiteneingängen des Propagierungspfades bleiben zunächst unbeachtet. Nachdem der Propagierungspfad vollendet ist, wird die Verknüpfungseinheit mit geeignetem Verhalten an dessen Eingang angefügt. Darauf wird am Rückkopplungseingang der Verknüpfungseinheit Schritt für Schritt der zweite transparente Propagierungspfad erzeugt, bis dessen Eingang vom Akkuregister gespeist wird. Die Rückkopplung kann dann geschlossen werden. Zuletzt werden noch Propagierungspfade für eventuelle Konstanten an den Seiteneingängen der Rückkopplung bzw. an der Verknüpfungseinheit hinzugefügt. Konstante Werte können dabei über die Schaltungseingänge angelegt werden, oder sie werden von Registern geliefert, die vor einer Testsitzung mit einem konstanten Wert initialisiert worden sind und diesen

Bestimme die Menge VE aller als Verknüpfungseinheit geeigneten arithmetischen Module;

Für alle Verknüpfungseinheiten VE $\in$ VE:

Erzeuge am Ausgang von VE alle transparenten, zyklenfreien Propagierungpfade, die mit einem Akkuregister enden, dem das Kompaktiererverhalten „s: H“ am Ausgang zugewiesen wurde. Wann immer ein solcher Propagierungspfad P erzeugt wurde:

Für jedes Kompaktierungsverhalten für VE:

Füge VE am Eingang des Propagierungspfades P hinzu;

Erzeuge alle transparenten, zyklenfreien Pfade am Rückkopplungseingang von VE, die vom Akkuregister gespeist werden. Wann immer ein solcher Pfad erzeugt wurde:

Schließe die Rückkopplung;

R_K := der aktuelle Kompaktiererrumpf;

R_M := eine zum Mustergeneratorrumpf transformierte Kopie von R_K;

Erweitere R_K bzw. R_M um alle möglichen Kombinationen von Propagierungspfaden für konstante Werte an den jeweiligen Seiteneingängen. Wann immer eine solche Kombination gefunden wurde:

Speichere den vollständigen Kompaktierer A_K bzw. den Mustergenerator A_M.

Ende.

Bild 3.8: Algorithmus zur Konfiguration aller Akkumulatoren in einer Schaltung

während der Testsitzung speichern. Bild 3.8 zeigt den gesamten Algorithmus zur Konfiguration von Akkumulatoren.

Sieht man von den Propagierungspfaden für konstante Werte ab, betrachtet man also nur den *Rumpf* eines Akkumulators, unterscheiden sich die Strukturen eines Mustergenerators und eines Kompaktierers nicht. Der Unterschied liegt lediglich darin, daß in der Rückkopplung einmal Testmuster p statt Signaturen s transportiert werden und am Seiteneingang der Verknüpfungseinheit ein konstanter Wert v statt einer Testantwort p erwartet wird. Es genügt daher, die Struktur eines Akkumulatorrumpfes nur einmal zu erzeugen, und erst dann die Trennung zwischen Mustergenerator und Kompaktierer vorzunehmen.

Zur abschließenden Demonstration des Algorithmus soll ein Mustergenerator innerhalb der Schaltung von Bild 3.9a konfiguriert werden. Der Addierer ist die einzige arithmetische Einheit, die als Verknüpfungseinheit eingesetzt werden kann, die Menge VE enthält also nur ein Element. Der Ausgang des Addierers speist nur ein einziges Modul, einen Multiplexer. Der Algorithmus würde zuerst das Kompaktiererverhalten betrachten, d.h. Signaturen über den Multiplexer propagieren. Da hier nur ein Mustergenerator betrachtet werden soll, wird der Übersichtlichkeit halber nur das Verhalten eines Mustergenerators betrachtet, der Multiplexer propagiert also Testmuster p (Bild 3.9b). Am Multiplexerausgang befindet sich nur das Register R1, das Akkuregister. Sein Ausgangsverhalten wird mit „p: H“ vorgegeben und der bisherige Propagierungspfad, der Multiplexer, an seinem Eingang hinzugefügt (Bild 3.9c). Nun wird das Verhalten der Verknüpfungseinheit bestimmt und die Verknüpfungseinheit mit dem Propagierungspfad verschmolzen. Von den beiden Möglichkeiten, eine Konstante mit einem Testmuster zu verknüpfen, ist hier nur diejenige gezeigt, bei der die Konstante am linken Eingang der Verknüpfungseinheit anliegt (Bild 3.9d). Nun muß die Rückkopplung fertiggestellt werden. Dazu wird der Multiplexer am Rückkopplungseingang der Verknüpfungseinheit angefügt. Tatsächlich würde erst untersucht, ob eine Rückkopplung über den Eingang 0 des Multiplexers verlaufen kann. Dies ist nicht der Fall, deswegen wird der Eingang 1 zur Propagierung ausgewählt. Sein Verhalten ist „p: $V(L_1, H)$“. Da der Eingang 1 vom Akkuregister gespeist wird, kann die Rückkopplung geschlossen werden. Das Akkuregister muß Testmuster zumindest in den Taktzyklen liefern, die durch $V(L_1, H)$ spezifiziert sind. Um dies zu garantieren, muß die Ausgangfunktion $N(L_1, H)$ des Akkuregisters die Funktion $V(L_1, H)$ *überdecken*, d.h. die zusätzliche Überdeckungsbedingung „$\forall t$: $N(L_1(t), H(t)) \geq V(L_1(t), H(t))$“, kurz „$N(L_1, H) \geq V(L_1, H)$“, muß später von einer konkreten Ansteuerung erfüllt werden (Bild 3.9e). Zu diesem Zeitpunkt ist der Rumpf des Mustergenerators fertiggestellt. Wie schon gesagt, würde der Algorithmus zuerst den

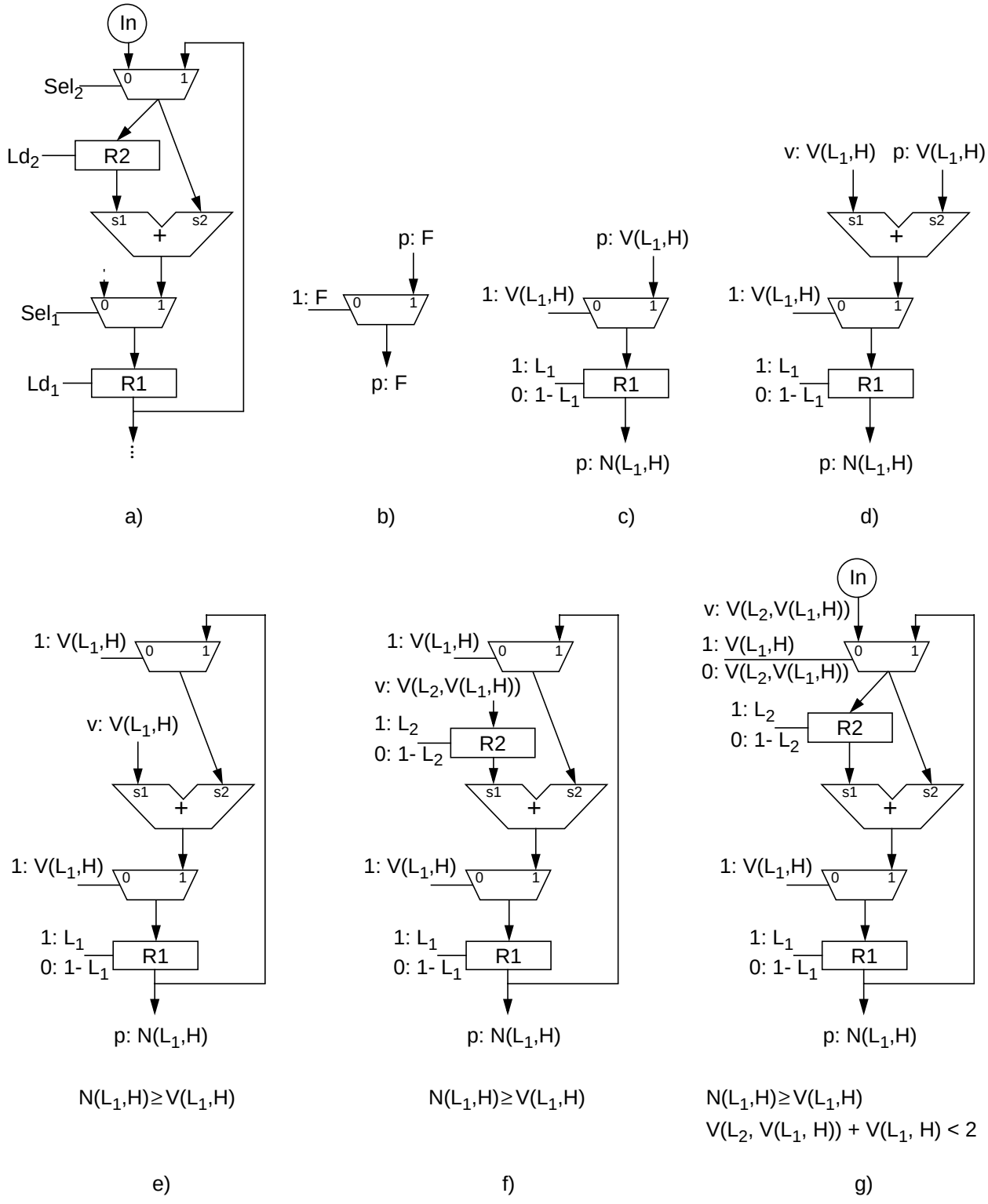


Bild 3.9: Erzeugung eines Mustergenerators (b,c,d,e,f, g) innerhalb einer Schaltung (a)

Rumpf eines Kompaktierers erzeugen und dann das Verhalten des Kompaktierers in das eines Mustergenerators transformieren, d.h. überall die Signaturen s durch Testmuster p ersetzen und die Testantwort p durch eine Konstante v.

Es fehlt nun noch der Propagierungspfad für die Konstante am Seiteneingang der Verknüpfungseinheit – die Rückkopplung selbst hat im Beispiel keine weiteren Seiteneingänge. Das Register R2 speist den Seiteneingang. Mit dem Verhalten, bei dem R2 die Konstante während einer Testsitzung speichert, erhält man einen möglichen Mustergenerator. In Bild 3.9f wurde stattdessen das Propagierungsverhalten für R2 gewählt. Der Propagierungspfad wird um den Multiplexer erweitert und über dessen Eingang 0 geführt, den der Schaltungseingang In schließlich mit der benötigten Konstante versorgen kann (Bild 3.9g). Bei diesem Mustergenerator schaltet jedoch der Multiplexer zwischen den beiden Eingängen 1 und 0 hin und her. Um Konflikte zu vermeiden, muß deshalb die Disjunktheitsbedingung „$V(L_2, V(L_1, H)) + V(L_1, H) < 2$“ erfüllt sein (vgl. Abschnitt 3.2.1.1).

Zuletzt würde der Eingang 1 des oberen Multiplexers zur Propagierung der Konstanten untersucht. Da R1 jedoch das einzige Register in der Rückkopplung ist, kann es nur Testmuster speichern und propagieren. Andernfalls ginge das Testmuster aus der Rückkopplung verloren und die Musterfolge wäre unterbrochen (vgl. auch Abschnitt 3.2.5). R1 blockiert daher jeden weiteren Propagierungspfad über Eingang 1 des Multiplexers.

3.2.3 Konfigurierung von Testblöcken

Testblöcke werden konfiguriert, indem Mustergeneratoren und Kompaktierer mit Testblockkernen kombiniert werden. Zuerst extrahiert man alle potentiellen Mustergeneratoren und Kompaktierer, die innerhalb der gegebenen Schaltung für einen Test verwendet werden können. Akkumulatoren werden dabei mit der oben beschriebenen Methode konfiguriert, für die Register wählt man einfach das Verhalten für LRSRs bzw. MISRs. Mustergeneratoren mit demselben Ausgang werden zu Gruppen zusammengefaßt. Dasselbe geschieht mit Kompaktierern, die Testantworten an Eingängen aufnehmen, die vom selben Ausgang gespeist werden.

Darauf werden alle Kerne mit nur einem Ausgang konfiguriert, sogenannte *primitive* Kerne. Diese werden schließlich miteinander kombiniert und zu *zusammengesetzten* Kernen mit mehreren Ausgängen verschmolzen. Immer wenn ein neuer Kern gefunden worden ist, wird er mit den verschiedenen Kombinationen von angrenzenden Mustergeneratoren und Kompaktierern zu Testblöcken erweitert. Die nächsten beiden Abschnitte beschreiben, wie primitive und zusammengesetzte Kerne konstruiert werden. Beide Verfahren beruhen auf *Branch-and-bound*-Techniken. Zur effizienten Einschränkung des Suchraums werden verschiedene Methoden und Heuristiken verwendet. Abschnitt 3.2.3.3 stellt diese vor.

3.2.3.1 Primitive Testblockkerne und Testblöcke

Ausgangspunkt für einen Testblockkern ist ein Modul, dessen Ausgang mit einer Gruppe von Kompaktierern verbunden ist. Für das Modul wird ein beliebiges Verhalten gewählt, das Testmuster am Modulausgang erzeugt. Je nach Modul und Verhalten müssen weitere Werte, Testmuster oder Konstanten, zu den Moduleingängen geleitet werden. Für jeden Eingang, der Testmuster verlangt, wird Schritt für Schritt ein eigener Propagierungspfad erstellt, dasselbe gilt für eventuelle Seiteneingänge dieser Pfade. Eingänge für Konstanten bleiben vorerst unberücksichtigt, d.h. ähnlich wie bei Akkumulatoren betrachtet man zunächst nur Rümpfe möglicher Testblockkerne bzw. Testblöcke. Letztlich erhält man so einen Datenflußbaum, dessen Blätter Eingänge sind, die mit Testmustern oder Konstanten belegt werden müssen. Die Wurzel ist das zuerst gewählte Modul, über dessen Ausgang die Testantworten in einen Kompaktierer gespeist werden. Durch Betrachtung aller möglichen Wurzeln, systematisches Hinzufügen von Modulen, z.B. nach einem Breitensuch-Schema, und Berücksichtigung aller geeigneten Verhalten dieser Module, lassen sich alle Datenflußbäume aufbauen.

Immer wenn alle Eingänge für Testmuster mit Mustergeneratoren verbunden werden können, ist der Rumpf eines Testblockkerns vollendet, und man erweitert ihn zu Testblöcken. Dazu kombiniert man die Mustergeneratoren und Kompaktierer der beteiligten Gruppen an den Eingängen bzw. am Wurzelmodulausgang, verschmilzt die verschiedenen Kombinationen mit dem Rumpf des Kerns und erzeugt zuletzt die Propagierungspfade für die konstanten Seiteneingänge.

Im folgenden wird die Konfiguration eines primitiven Kerns an einem Beispiel demonstriert. Zuvor soll jedoch erst auf die Frage eingegangen werden, welche Module eigentlich von einem Testblock getestet werden können, d.h. von ihm *testbar* sind. Bisher ist ein Testblock nichts anderes als ein Datenflußbaum, der die von Mustergeneratoren erzeugten Musterfolgen miteinander zu einer einzigen Musterfolge an seinem Ausgang verknüpft. Diese Musterfolge wird von einem Kompaktierer zu einer Signatur verarbeitet. Nun enthält ein Testblock mehrere Module, und nicht jedes dieser Module kann von ihm getestet werden. Damit ein Modul getestet werden kann, müssen vielmehr einige Voraussetzungen erfüllt sein.

Die in den Abschnitten 2.3.1 und 2.3.2 gemachten Aussagen zur Fehlererfassung und Maskierungswahrscheinlichkeit bei Testregistern und Akkumulatoren beruhen darauf, daß Fehler nur in dem zu testenden Modul auftreten, die angelegten Musterfolgen sowie die Kompaktierer jedoch von diesen Fehlern unbeeinflußt bleiben. Um diese Voraussetzungen zu erfüllen, werden Module grundsätzlich

als vom Testblock nicht testbar angesehen, wenn sie innerhalb des Testblocks Teil eines Mustergenerators oder Kompaktierers sind oder mehr als einmal im Datenflußbaum des Testblockkerns vorkommen. Ein Modul kann ferner nur getestet werden, wenn sein Testverhalten gewählt wurde, d.h. an allen Moduleingängen müssen Testmuster anliegen – eine Ausnahme sind Eingänge, die schon im Normalbetrieb der Schaltung nur mit einer Konstante belegt werden können.

Die einzelnen Module werden als zufallstestbar angesehen. Für den Test eines Moduls müssen die Musterfolgen an den Eingängen somit zufällig genug sein, und Fehler in den Testantworten am Modulausgang müssen ausreichend genau vom Kompaktierer beobachtet werden können. Dazu werden die Testbarkeitsmaße *Zufälligkeit* und *Beobachtbarkeit* eingeführt, die Werte zwischen 0 (überhaupt nicht zufällig bzw. beobachtbar) und 1 (vollkommen zufällig bzw. beobachtbar) annehmen können. Die Zufälligkeit der Musterfolgen an den Eingängen eines testbaren Moduls bzw. die Beobachtbarkeit am Modulausgang müssen vorgegebene Schwellwerte überschreiten. Die Schwellwerte dienen dabei als grobe Hilfsmittel zur Abwägung zwischen Testhardware bzw. Fehlererfassung und Testzeit. Je höher die Schwellwerte angesetzt werden, umso größer ist die Fehlererfassung bzw. umso kürzer ist die benötigte Testlänge, die man für den Test der Module erwarten kann. Mit geringeren Schwellwerten läßt sich unter Umständen Testhardware einsparen. Die Wahl der Schwellwerte hängt letztlich von der Erfahrung und den Zielen des Anwenders ab, für die Experimente in Abschnitt 3.2.6 wurden sie z.B. mit 0,9 für die Zufälligkeit und 0,8 für die Beobachtbarkeit relativ hoch angesetzt.

Die Zufälligkeit der Muster an einem Moduleingang mit n bit ist hier definiert als das Verhältnis der Zahl verschiedener an ihm einstellbarer Eingangsbelegungen und der Anzahl aller denkbaren 2^n Eingangsbelegungen. Die Beobachtbarkeit eines Ausgangs ist die Chance, daß ein Bitwechsel im Ausgangsmuster sich irgendwann am Kompaktierereingang bemerkbar macht, so daß ihn der Kompaktierer registrieren kann. Genauer gesagt ist es die Anzahl aller Paare (a, i) von Ausgangsmustern a und Bitpositionen i, bei denen ein Bitwechsel an der i-ten Position im Muster a bis zum Kompaktierereingang propagiert werden kann, geteilt durch die Anzahl aller Paare (a, i). Die exakten Zufälligkeits- bzw. Beobachtbarkeitswerte sind meist nur schwer zu bestimmen. Stattdessen wird hier dem Ansatz aus [ChPa91a] gefolgt, bei dem der Einfluß eines einzelnen Moduls jeweils isoliert betrachtet wird.

An den Ausgängen von Mustergeneratoren wird dabei die Zufälligkeit als maximal angenommen, ebenso die Beobachtbarkeit von Testantworten am Ausgang eines Testblockkerns, da er direkt mit einem Kompaktierer verbunden ist. Jedes Modul, über das eine Musterfolge propagiert wird, kann

die Zufälligkeit bzw. Beobachtbarkeit der Folge um einen Faktor Z bzw. B, $0 \leq Z, B \leq 1$, vermindern. Der Faktor Z ist hier als das Verhältnis zwischen der maximalen Anzahl verschiedener, über die Moduleingänge einstellbarer Ausgangsmuster, und der Anzahl aller denkbaren Ausgangsmuster definiert. Bei einem Multiplizierer mit 2n Ausgangsbits ist das z.B. die Zahl möglicher Produkte geteilt durch die Anzahl aller denkbaren 2^{2n} Ausgangsbelegungen. Der Faktor B ist hingegen die Anzahl der Paare (e, i) von Eingangsmustern e und Bitpositionen i, bei denen ein Wechsel des Bits i im Muster e das Ausgangsmuster verändert, geteilt durch die Anzahl aller Paare (e, i). Die Werte für Z und B sind für jedes Modul und jedes Verhalten analytisch bzw. per Monte-Carlo-Simulation bestimmt worden und ebenfalls in der Bausteinbibliothek gespeichert. Tabelle 3.2 enthält für verschiedene arithmetische und logische Operationen mit einer Eingangswortbreite von 16 bit die Faktoren Z und B, und zwar jeweils für das Testverhalten, das transparente Propagierungsverhalten und das durchlässige Propagierungsverhalten (vgl. Tabelle 3.1). Mul^O ist wieder die Multiplikation, bei der nur das obere Halbwort des Produkts berücksichtigt wird, Div^R der Rest bei der ganzzahligen Division.

		Add.	Sub.	Mul.	Mul^O	Div.	Div^R	UND	ODER	XOR
Testverhalten	Z B	1,0 1,0	1,0 1,0	0,21 0,93	0,99 0,93	1,0 0,15	0,99 0,76	1,0 0,5	1,0 0,5	1,0 1,0
transparentes Verhalten	Z B	1,0 1,0	1,0 1,0	0,0 1,0	–	1,0 1,0	–	1,0 1,0	1,0 1,0	1,0 1,0
durchlässiges Verhalten	Z B	1,0 1,0	1,0 1,0	–	0,99 1,0	–	0,99 1,0	–	–	1,0 1,0

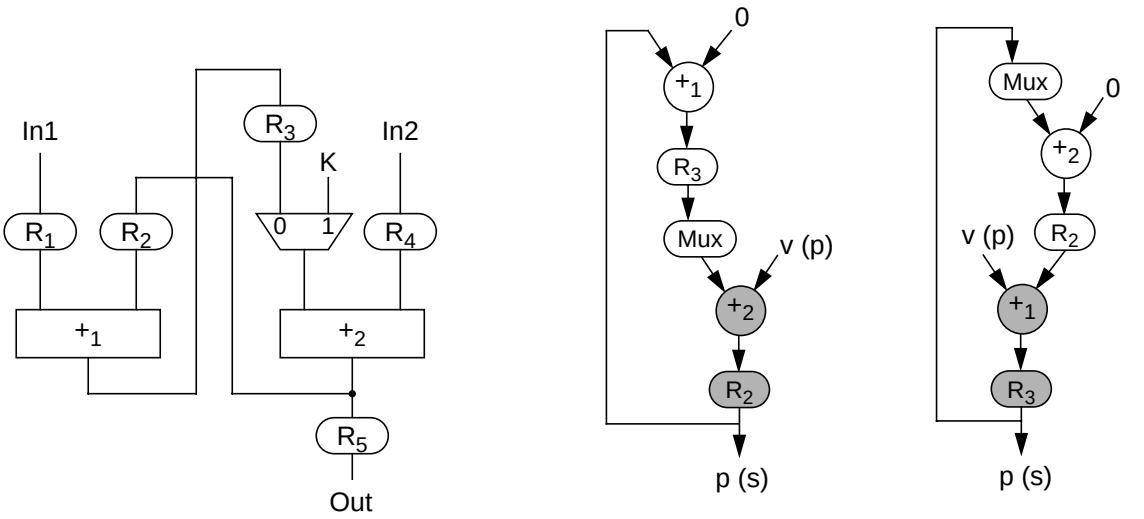
Tabelle 3.2: Faktoren Z und B für typische arithmetische und logische Operatoren

Die Zufälligkeit der Musterfolge am Ausgang eines Moduls wird nun folgendermaßen bestimmt. Zuerst berechnet man die Zufälligkeit der Musterfolgen an den Moduleingängen. Von den erhaltenen Zufälligkeitswerten wählt man den größten aus und multipliziert ihn mit dem Zufälligkeitsfaktor Z des Modulverhaltens. Das Produkt ist die Zufälligkeit der resultierenden Musterfolge am Ausgang des Moduls. Analog wird anhand der Beobachtbarkeit von Testantworten am Modulausgang, die Beobachtbarkeit von Testantworten an den Moduleingängen bestimmt.Um Korrelationen zwischen Musterfolgen zu vermeiden, die zu verminderter Fehlererfassung bzw. erhöhter Fehlermaskierungswahrscheinlichkeit führen können, müssen die Musterfolgen am Eingang eines Moduls unabhängig voneinander sein, andernfalls wird die Zufälligkeit am Ausgang und die Beobachtbarkeit an den Eingängen auf den Minimalwert 0 gesetzt. Zwei Folgen werden dabei als unab-

hängig voneinander angesehen, wenn sie nicht von den Folgen derselben Mustergeneratoren abhängen. Da die Zufälligkeits- bzw. Beobachtbarkeitswerte der Mustergeneratoren bzw. des Kompaktierers vorgegeben sind, lassen sich alle Zufälligkeits- bzw. Beobachtbarkeitswerte an den Ein- und Ausgängen der Module innerhalb des Testblocks bestimmen. Die hier vorgestellte Methode zur Berechnung von Testbarkeitsmaßen ist relativ einfach und vergleichsweise schnell. Alternativen sind z.B. in [ThAb89, ChPa91b] beschrieben.

Mit der oben beschriebenen Methode läßt sich für jeden Testblock überprüfen, welche Module von ihm getestet werden können und welche nicht. In der Tat beschränkt sich das hier vorgestellte Selbsttestverfahren – wie viele andere Verfahren auch – auf den Test von Funktionseinheiten. Multiplexer und Register sind leicht testbar und werden meist beim Test der Funktionseinheiten mit getestet. Sie werden deshalb nicht explizit auf Testbarkeit hin untersucht. Bei Bedarf läßt sich für sie die Fehlererfassung nachträglich steigern, z.B. durch Hinzunahme geeigneter Testblöcke, Einbau von Testpunkten oder den Test mit wenigen deterministischen Mustern. Bei Fehlern in Testregistern ist dabei zu beachten, daß nicht alle dieser Fehler im Testbetrieb mitgetestet werden. Testregister müssen daher auch im Normalbetrieb getestet werden.

Im folgenden soll anhand des Beispiels in Bild 3.10 die Konfiguration von primitiven Testblockkernen und Testblöcken demonstriert werden.



a) Schaltung b) A_{MG1} bzw. A_{K1} c) A_{MG2} bzw. A_{K2}

Bild 3.10: Beispielschaltung (a) mit den Datenflüssen zweier Akkumulatorrümpfe (b,c)

Ausgangspunkt ist die Schaltung in Bild 3.10a, die bereits in Abschnitt 3.1 vorgestellt wurde. Die Schaltung enthält zwei verschiedene Akkumulatorrümpfe, die sich sowohl als Mustergeneratoren

(A_{MG1} bzw. A_{MG2}) und Kompaktierer (A_{K1}, A_{K2}) verwenden lassen. Ihre Datenflüsse sind in Bild 3.10b/c dargestellt, die schraffierten Module sind die jeweilige arithmetische Verknüpfungseinheit bzw. das Akkuregister. Zuerst werden die Mustergeneratoren und Kompaktierer gruppiert. Wie schon oben beschrieben, bilden Mustergeneratoren, die ihre Musterfolgen am selben Modulausgang erzeugen, eine Gruppe. Dasselbe gilt für Kompaktierer, die Testantworten vom selben Modulausgang übernehmen. Ein Register R_i kann dabei als LFSR L_i oder als MISR M_i arbeiten. An den Schaltungseingängen bzw. -ausgängen können zusätzliche Mustergeneratoren bzw. Kompaktierer plaziert werden. Der Übersicht halber werden sie hier weggelassen. Man erhält schließlich die Mustergeneratorgruppen $\{L_1\}$, $\{L_2, A_{MG1}\}$, $\{L_3, A_{MG2}\}$, $\{L_4\}$ und $\{L_5\}$ und für Kompaktierer die Gruppen $\{M_1\}$, $\{A_{K2}\}$, $\{M_2, M_5\}$, $\{M_3\}$, $\{A_{K1}\}$ und $\{M_4\}$.

Nun werden die Testblockkerne konfiguriert (Bild 3.11). Jedes Modul, das eine Gruppe von Kompaktierern speist, ist ein potentielles Wurzelmodul. In diesem Beispiel wird nur der Addierer $+_2$ als Wurzelmodul betrachtet. Für den Addierer steht das Testverhalten mit Testmustern an allen Eingängen zur Verfügung, ebenso sind die Propagierungsverhalten verfügbar, die neben einem Testmuster noch eine Konstante v an einem Seiteneingang benötigen. Der Spezialfall v = 0, bei dem der Addierer transparent ist, braucht nicht extra berücksichtigt werden, da die Propagierungspfade innerhalb eines Testblockkerns im Gegensatz zu denen in Akkumulatoren lediglich durchlässig sein müssen. Für das Beispiel wurde nur das Testverhalten betrachtet. Da der Addierer als Wurzelmodul fungiert, wird sein Verhalten mit „p: T“ vorgegeben. Er generiert also Testantworten p entsprechend einer symbolischen Funktion T (Bild 3.11a).

Zuerst wird ein Propagierungspfad für den linken Eingang von $+_2$ gesucht: es werden Testmuster über den 0-Eingang des Multiplexers geleitet (Bild 3.11b). Da beide Eingänge der resultierenden Teilschaltung von Mustergeneratoren gespeist werden, nämlich von den Gruppen $\{L_3, A_{MG2}\}$ bzw. $\{L_4\}$, liegt der Rumpf eines Testblockkerns vor. Er wird zu einem Testblock erweitert. Als Kompaktierer eignen sich M_2 und M_5, im Beispiel wird nur M_5 betrachtet. Als Mustergeneratoren werden L_3 und L_4 eingesetzt, das Ergebnis ist der Testblock von Bild 3.11c, der den Addierer $+_2$ testen kann. An den Ausgängen der LRSR L_3 und L_4 liegen in jedem Taktzyklus Muster p an, die zugehörige Ausgangsfunktion ist somit **1**. Damit die Muster zu den geforderten Zyklen entsprechend T anliegen, müssen die Ausgangsfunktionen der LRSRs die Eingangsfunktion T überdecken, d.h. die Überdeckungsbedingung „$\mathbf{1} \geq$ T“ muß erfüllt werden. Bei der Kompaktierung ist es hingegen nicht unbedingt nötig, daß alle Testantworten verarbeitet werden, eine ausreichende Anzahl genügt. Für die Konfiguration von Testblöcken reicht es, wenn die Funktion X am Testantworteingang eines

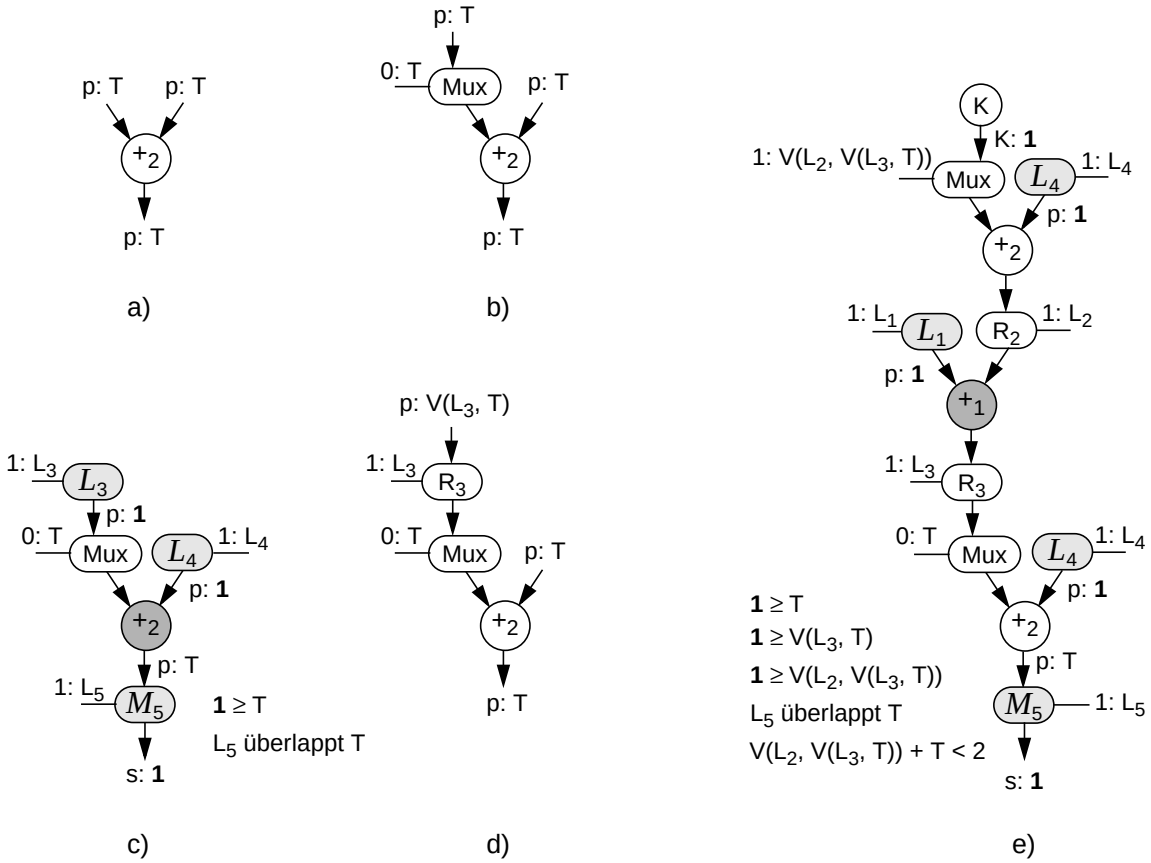


Bild 3.11: Konfiguration einiger Testblockkerne und Testblöcke

Kompaktierers die Ausgangsfunktion T des Wurzelmoduls lediglich *überlappt*, d.h. die Überlappungsbedingung „∃t: X(t) = T(t) = 1“ oder „X überlappt T“ muß gelten. Für den Kompaktierer M_5 heißt das „L_5 überlappt T“. Für eine ausreichende Anzahl von kompaktierten Testantworten sorgt das später vorgestellte Verfahren zur Bestimmung einer konkreten Ansteuerung (s. Abschnitt 3.2.5).

Anstelle von L_3 ist auch A_{MG2} als Mustergenerator denkbar. Allerdings ist das einzige zu testende Modul des Kerns, der Addierer $+_2$, auch Bestandteil von A_{MG2} und würde damit nach den oben festgelegten Voraussetzungen für testbare Module als nicht testbar angesehen. Der resultierende Testblock würde also keine Funktionseinheit testen und wäre überflüssig.

Nachdem alle Testblöcke für den Kern aus Bild 3.11b erzeugt worden sind, wird der Kern erweitert. Tatsächlich würden erst alle Propagierungspfade für den rechten Eingang von $+_2$ untersucht, bevor mit Propagierungspfaden am 0-Eingang des Multiplexers fortgefahren werden würde. Diese Schritte werden hier übersprungen und gleich das Register R_3 am 0-Eingang hinzugefügt (Bild 3.11d). Nach weiteren Kernen und Testblöcken erhält man schließlich den Testblock in Bild 3.11e.

Dabei wurde für den Addierer $+_1$ wieder das Testverhalten ausgewählt. Die Musterfolge für den rechten Eingang stammt diesmal vom LRSR L_4, wobei die Muster wieder über den Addierer $+_2$ geleitet werden. Diesmal wurde jedoch das Propagierungsverhalten für $+_2$ verwendet. Die erforderliche Konstante am linken Seiteneingang wird dazu vom konstanten Eingang K über den 1-Eingang des Multiplexers transportiert. Um Kurzschlüsse am Addierer $+_2$ bzw. dem Multiplexer zu vermeiden, muß die Disjunktheitsbedingung „V(L_2, V(L_3, T)) + T < 2“ erfüllt werden.

Der Testblock testet diesmal nur den Addierer $+_1$. Tatsächlich wird für das Wurzelmodul, den Addierer $+_2$, auch das Testverhalten verwendet. Ebenso werden die Musterfolgen an beiden Eingängen nach den obigen Voraussetzungen als unkorreliert angesehen, da die Folge am linken Eingang außer von LRSR L_4 auch von L_1 abhängt, die Folge am rechten Eingang dagegen nur von L_4. Die Folgen sind auch zufällig genug, da sie neben den transparenten Registern und dem Multiplexer nur über Addierer propagiert werden, die die Zufälligkeit und Beobachtbarkeit nicht beeinträchtigen. Insoweit ist der Addierer $+_2$ testbar. Allerdings wird er zweimal im Datenfluß des Testblockkerns eingesetzt. Somit erfüllt er nicht alle Voraussetzungen eines testbaren Moduls.

3.2.3.2 Zusammengesetzte Testblockkerne und Testblöcke

In jeder Testsitzung wird ein Testblock für eine bestimmte Anzahl von Taktzyklen aktiviert. Manche der primitiven Testblöcke können als *zusammengesetzter* Testblock statt in mehreren Testsitzungen hintereinander parallel während einer einzigen Testsitzung arbeiten. Mithilfe von zusammengesetzten Testblöcken ist es demnach möglich, Testzeit einzusparen.

Wie bei den primitiven Testblöcken werden zusammengesetzte Testblöcke durch Erweiterung ihrer Kerne gebildet. Ein zusammengesetzter Kern besteht aus mehreren primitiven Kernen. Ein Kern vom *Grad n* ist dabei die Kombination von n primitiven Kernen. Tatsächlich werden für die Kerne vom Grad n erst die Kerne vom Grad n-1 erzeugt und diese dann mit primitiven Kernen kombiniert. Man erhält also zuerst alle Kerne vom Grad 2, baut dann die Kerne vom Grad 3 auf usw. Bei der Kombination zweier Kerne faßt man jeweils die Ansteuerungsbedingungen zusammen und überprüft sie auf Widersprüche. Natürlich gelten weiterhin die Voraussetzungen für testbare Module. Es ist daher möglich, daß zwei Kerne mehr Module testen können, wenn man sie einzeln statt als zusammengesetzten Kern betrachtet.

Nur die Testblöcke *lohnender* Kerne können Teil eines optimalen Tests sein. Ein primitiver Block lohnt sich z.B. nur, wenn er mindestens ein Modul testen kann. Andernfalls ist er nutzlos. Zusam-

mengesetzte Testblöcke werden nur dann als lohnend angesehen, wenn sie folgende Bedingungen erfüllen:

Sei k_n ein Kern vom Grad n, der sich aus zwei Kernen k_i und k_j zusammensetzt. Der Kern k_n lohnt sich genau dann, wenn

- k_i und k_j selbst lohnend sind, und
- k_n läßt sich zu einem Testblock erweitern, der mindestens n Module testen kann, und
- ein Test, der diesen Testblock aktiviert ist bezüglich einer gegebenen Kostenfunktion (s. Abschnitt 3.2.4) billiger als jeder Test mit zwei Testblöcken für k_i und k_j.

Die ersten beiden Bedingungen sollen Kerne vermeiden, die unnötig viele primitive Kerne beinhalten. In diesem Fall, könnte man einen primitiven Kern herausnehmen, ohne die Zahl testbarer Module eines seiner Testblöcke zu verringern. Durch den zusätzlichen Kern würden allenfalls die Testkosten steigen. Die dritte Bedingung führt dazu, daß ein Testblock des zusammengesetzten Kerns die von ihm testbaren Module in kürzerer Zeit testen kann, als wenn man die Testblöcke, aus denen er sich zusammensetzt, nacheinander aktiviert. In der Tat ist es möglich, daß man trotz der parallelen Abarbeitung von Testblöcken keine Reduktion der Testzeit erhält. Zur Verdeutlichung wird die Schaltung von Bild 3.12a betrachtet.

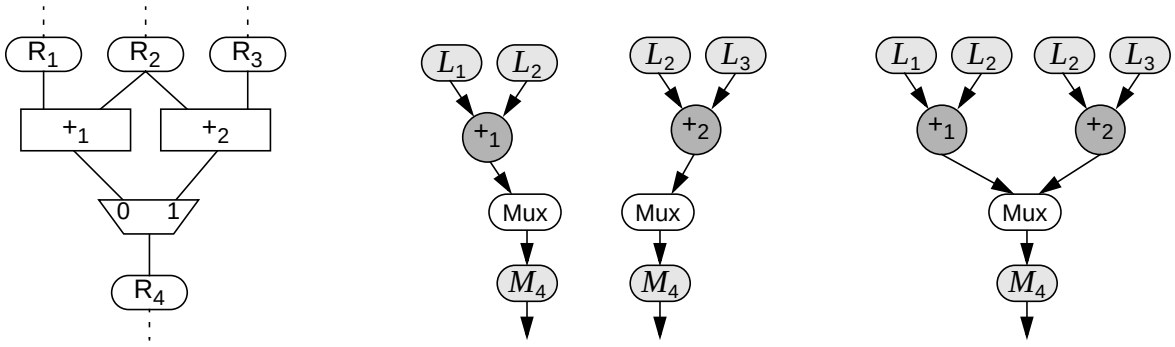


Bild 3.12: Beispiel für einen zusammengesetzten Testblock ohne Testzeitverkürzung

Bild 3.12b zeigt die Datenflüsse zweier Testblöcke, die den Addierer $+_1$ bzw. den Addierer $+_2$ testen können. Werden die LRSRs und das MISR mit dem globalen Taktsignal angesteuert, produzieren die LRSRs mit jedem Taktzyklus ein neues Testmuster. Das MISR kann dann in jedem Taktzyklus eine neue Testantwort kompaktieren. Benötigen beide Addierer für einen ausreichenden Test die-

selbe Anzahl N von Zufallsmustern, dauert der Test für jeden Testblock N Taktzyklen, insgesamt also 2N Zyklen. In Bild 3.12c ist der Datenfluß des Testblocks für den zusammengesetzten Kern gezeigt. Der Multiplexer kann hier immer nur eine Testantwort weiterleiten, entweder die von $+_1$ oder die von $+_2$. Der Test beider Addierer benötigt demnach wiederum mindestens 2N Taktzyklen, eine Einsparung an Testzeit wird nicht erreicht, die Ansteuerung für den Multiplexer ist sogar komplexer geworden. In diesem Fall ist der Test mit den getrennten Testblöcken dem Test mit zusammengesetztem Testblock vorzuziehen.

3.2.3.3 Einschränkung des Suchraums

Die Zahl aller möglichen Testblöcke innerhalb einer Schaltung ist gewöhnlich zu groß, um sie vollständig zu untersuchen. Aus Aufwandsgründen muß der Suchraum deshalb geeignet eingeschränkt werden. Verschiedene Methoden helfen dabei, die Suche auf möglichst wenige Bereiche zu konzentrieren, aus deren Testblöcken sich dennoch ein optimaler oder möglichst guter Test zusammenstellen läßt. Einige davon werden im folgenden vorgestellt.

Ein Beispiel ist die bereits erwähnte Methode, zunächst den Rumpf einer Teilschaltung zu erzeugen und erst danach die Propagierungspfade für Konstanten. Viele Akkumulatoren bzw. Testblockkerne unterscheiden sich nur in der Wahl dieser Propagierungspfade, nicht jedoch in ihrem Rumpf. Werden statt der Rümpfe komplette Teilschaltungen zu Testblöcken oder zusammengesetzten Kernen verschmolzen, hat man zum einen den Aufwand zu bestimmen, der für alle diese Teilschaltungen bzw. die verschiedenen Propagierungsmöglichkeiten für Konstanten nötig ist. Zudem verursachen diese Propagierungspfade beim Verschmelzen oft Widersprüche bei den Ansteuerungsbedingungen, obwohl die Rümpfe selbst miteinander kompatibel wären. Weit effizienter ist es, nur die Rümpfe von Akkumulatoren und Testblockkernen zu betrachten. Für jeden Rumpf wird lediglich überprüft, ob alle Konstanten widerspruchsfrei propagiert werden können. Andernfalls ließe sich der Rumpf niemals zu einer funktionsfähigen, vollständigen Teilschaltung erweitern. Erst nachdem der Rumpf eines Testblocks fertiggestellt ist, werden die verschiedenen Propagierungsmöglichkeiten für Konstanten untersucht. Auf diese Weise reduziert man nicht nur die Zahl der insgesamt zu untersuchenden Propagierungspfade, sondern auch die Zahl der möglichen Verschmelzungen zur Erzeugung aller Testblöcke bzw. der zusammengesetzten Kerne.

In der Tat sind die Kosten eines Testblocks weitgehend unabhängig von der Wahl der Propagierungspfade für Konstanten (vgl. Abschnitt 3.2.4). Es genügt daher, lediglich eine einzige Propagierungsmöglichkeit zu betrachten. Bei der Wahl dieser Möglichkeit ist jedoch Vorsicht geboten, da

ein testbares Modul nach den obigen Voraussetzungen nicht gleichzeitig in einem Propagierungspfad für eine Konstante enthalten sein kann. Dazu wird eine hier nicht näher erläuterte Heuristik eingesetzt, die möglichst schnell eine Propagierungsmöglichkeit findet, bei der die Zahl testbarer Module nicht oder möglichst wenig reduziert wird.

Ein anderer Problemfall sind Rekonvergenzen in der Schaltungsstruktur. Bild 3.13a zeigt als Beispiel einen Schaltungsausschnitt mit einer Rekonvergenz. Vom Ausgang eines Moduls M1 führt ein

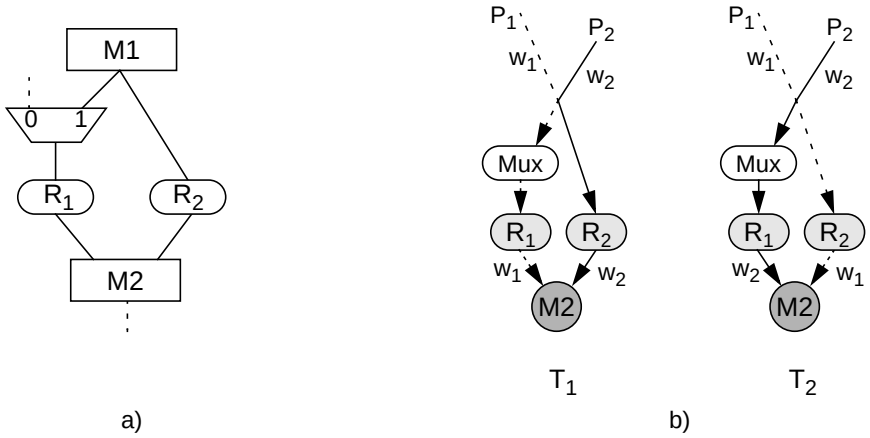


Bild 3.13: Schaltungsausschnitt (a) mit den zugehörigen Datenflüssen zweier äquivalenter Testblöcke T_1 und T_2 (b)

transparenter Pfad zum linken Eingang des Moduls M2 und ebenso einer zum rechten Eingang. Seien P_1 und P_2 zwei Propagierungspfade, die den Wert w_1 bzw. w_2 zum Ausgang von M_1 transportieren, von wo er beliebig zum linken oder rechten Eingang von M2 weiterpropagiert werden kann. Sei T_1 ein Testblock, bei dem der Wert w_1 über P_1 zum linken Eingang von M2 geführt wird und w_2 über P_2 zum rechten Eingang. T_2 sei nun ein Testblock, der sich von T_1 nur darin unterscheidet, daß der Wert w_1 zwar wieder über P_1, diesmal jedoch zum rechten Eingang geleitet und entsprechend w_2 über P_2 zum linken Eingang (Bild 3.13b). Offensichtlich kann der Testblock T_1 innerhalb einer Testsitzung durch den Testblock T_2 ersetzt werden, ohne die Kosten für den Test oder die Zahl der getesteten Module zu verändern. Hinsichtlich des Selbsttests sind T_1 und T_2 *äquivalent*. Es genügt daher, nur einen der beiden Testblöcke zu konfigurieren. Deshalb wird die Schaltung auf solche Rekonvergenzen hin untersucht und die Konfiguration äquivalenter Testblöcke weitgehend vermieden.

Eine weitere Heuristik zur Verminderung des Rechenaufwands nutzt die Tatsache, daß ein Testblock nicht weiter berücksichtigt werden muß, wenn ein anderer Testblock dieselben Module testen kann und als Teil eines Tests die Testkosten auf keinen Fall erhöhen würde. Der erste Testblock ist

dann im Vergleich zum zweiten Testblock *zu teuer.* Entsprechend ist ein Testblockkern zu teuer, wenn alle seine Testblöcke gegenüber den Testblöcken eines anderen Kerns zu teuer sind. Die Heuristik unterläßt die Konfigurierung zu teurer Testblöcke und Kerne bereits bei der Bestimmung der primitiven Kerne. Zwar ist es denkbar, daß ein zusammengesetzter Testblock im Vergleich zu anderen Testblöcken nicht zu teuer ist, obwohl er einen zu teuren primitiven Kern enthält. Dieser Fall ist jedoch sehr selten, und es ist sehr unwahrscheinlich daß dieser Testblock die bereits reduzierten Testkosten auch noch weiter senken könnte. Außerdem könnte durch ihn nur die Testzeit verkürzt werden. Die Hardware-Kosten blieben unverändert.

3.2.4 Auswahl der besten Testblöcke

Nach der Konfiguration der primitiven und zusammengesetzten Testblöcke müssen für einen bezüglich dieser Testblöcke optimalen Test diejenigen Testblöcke ausgewählt werden, bei denen die Testkosten minimal werden. Formal läßt sich das Optimierungsproblem wie folgt darstellen:

Sei B_{tot} die Menge aller konfigurierten primitiven und zusammengesetzten Testblöcke b. Sei *TM* die Menge aller von diesen Testblöcken testbaren Module, wobei Multiplexer und Register nicht berücksichtigt werden. Damit gilt $TM = \bigcup_{b \in B_{tot}} \text{getestete\_Module(b)}$. Ein optimaler Test entspricht dann der Teilmenge $B \subseteq B_{tot}$, bei der $\bigcup_{b \in B} \text{getestete\_Module(b)} = TM$ gilt und die Kosten C(*B*) bezüglich einer gegebenen Kostenfunktion minimal sind. In den folgenden beiden Abschnitten werden die Kostenfunktion eingeführt und das Verfahren beschrieben, das die optimale Teilmenge *B* bestimmt.

3.2.4.1 Kostenfunktion

Mithilfe der Kostenfunktion $C = C_{HW} + \alpha \cdot C_T$ werden der Hardware-Mehraufwand und die Testzeit miteinander in Beziehung gesetzt. Dabei schätzt C_{HW} die zusätzlichen Hardware-Kosten ab, C_T ist ein Kostenmaß für die erwartete Dauer eines Tests. Durch den Gewichtsfaktor α kann der Schwerpunkt der Optimierung zwischen geringen Hardware-Kosten (kleines α) und möglichst kurzer Testzeit (großes α) verschoben werden.

Als Hardware-Kosten werden nur die Ausbaukosten für Mustergeneratoren und Kompaktierer berücksichtigt. Die Kosten für konstante Werte an den Seiteneingängen von Modulen, die für den Transport von Testmustern benötigt werden, sind weitgehend unabhängig von der Auswahl der

Testblöcke für einen Test. In den meisten Fällen genügen nämlich die Konstanten 0 für Addition, Subtraktion, ODER und XOR, 1 für Multiplikation und Division sowie '1...1' für den Multiplizierer, bei dem nur das obere Halbwort des Produkts betrachtet wird, für den Rest bei der Division und für die UND-Verknüpfung. Viele Testblöcke liefern Konstanten an Seiteneingänge, besonders solche, die Bestandteil eines kostengünstigen Tests sein können. Kostengünstige Tests brauchen oft dieselbe Anzahl von Konstanten. Zudem sind die Kosten zur Bereitstellung einer Konstanten im Vergleich zu den Kosten für den Ausbau eines Registers zu einem Testregister gering. Bei der Optimierung werden daher Kostendifferenzen bei den Konstanten meist von denen bei Testregistern überblendet. Das hier vorgestellte Verfahren zur Auswahl der besten Testblöcke vernachlässigt deshalb die Kosten für solche Konstanten.

Das Teststeuerwerk läßt sich zum jetzigen Zeitpunkt noch nicht synthetisieren, und seine Kosten lassen sich damit nicht berechnen. Sie lassen sich auch nur schwer abschätzen, denn für die gegebenen Testblöcke sind nur Ansteuerungsbedingungen mit symbolischen charakteristischen Funktionen bekannt, jedoch keine konkreten Ansteuerungen. Allerdings zeigen die Untersuchungen in [MaSt98a] bzw. [MaSt98b], daß bei Akkumulatoren der Aufwand für optimierte Ansteuerungen kaum differiert. Außerdem wiegt wegen der meist großen Datenwortbreiten bei den Schaltungen zur digitalen Signalverarbeitung, die hier wegen ihrer Anzahl arithmetischer Module und damit möglicher Akkumulatoren bevorzugt betrachtet werden, der Ausbau zu Testregistern oft weit schwerer als eventuelle Differenzen beim Teststeuerwerk. Der Aufwand für das Teststeuerwerk wird daher ebenfalls als konstant angesehen und bei der Optimierung nicht berücksichtigt.

Der Test besteht aus einer Abfolge von Testsitzungen. In jeder Testsitzung wird genau ein Testblock aktiviert. Im Gegensatz zu anderen Verfahren, die erst bei der Testablaufplanung parallel arbeitende Testblöcke zu berücksichtigen versuchen, wurde dieses Problem schon mit der Erzeugung der zusammengesetzten Testblöcke erledigt. Die Testzeit hängt nun im wesentlichen davon ab, wie lang eine Testsitzung braucht, um für jedes Modul die erforderliche Anzahl von Testantworten zu erhalten und zu kompaktieren. Die Reihenfolge der Testsitzungen ist dabei egal. Die genaue Zahl notwendiger Testantworten läßt sich i.a. nur durch eine aufwendige Simulation der einzelnen Testblöcke bestimmen. Zudem können in der Regel verschiedene Testblöcke dasselbe Modul testen bzw. teilweise testen. In diesem Fall müßten bei der Optimierung mehrere Testsitzungen gleichzeitig betrachtet werden, um die kürzeste Testzeit zu erhalten. Eine solche Optimierung ist i.a. zu rechenaufwendig. Hier wird deshalb einfach davon ausgegangen, daß in jeder Testsitzung dieselbe Anzahl von Testantworten kompaktiert wird. C_T ist dann die Summe der Zeitkosten der verwende-

ten Testblöcke. Ein Testblock b, der in jedem Taktzyklus eine neue Testantwort erzeugt und kompaktiert, bekommt dabei die Zeitkosten $C_T(b) = 1$ zugewiesen. Da innerhalb eines Testblocks zwischen einzelnen Propagierungspfaden bzw. Teilpfaden hin- und hergeschaltet werden darf, ist es nicht immer möglich, mit jedem neuen Zyklus eine neue Testantwort zu erhalten. Schon wenn mindestens ein Modul n verschiedene Werte – neben Testmustern z.B. auch Konstanten oder Testantworten verschiedener Module – transportieren muß, kann eine neue Antwort nur noch maximal jeden n-ten Taktzyklus am Kompaktierer anliegen. In diesem Fall erhöht sich die Testzeit für das Modul um mindestens den Faktor n. Die Maximalzahl verschiedener Werte, die irgendein Modul im Testblock b transportieren muß, dient daher als Abschätzung für $C_T(b)$.

3.2.4.2 Bestimmung des optimalen Tests

Das Problem, eine optimale Auswahl $B \subseteq B_{tot}$ an Testblöcken zu finden, kann als Überdeckungsproblem mit zusätzlicher Kostenfunktion beschrieben werden. Dabei ist die Menge *TM* aller testbaren Module einer Schaltung die Menge, die überdeckt werden muß, und zwar mit Teilmengen TM_b = testbare_Module(b). Das ganze Problem wird hier als Problem der ganzzahligen linearen Optimierung formuliert – oft kurz ILP-Problem genannt (ILP von *integer linear programming*) [Neum75]. Die Formulierung besteht dabei aus einem System linearer Ungleichungen für ganzzahlige Variablen und einer Kostenfunktion, die ebenfalls linear bezüglich dieser Variablen ist. Es gilt dann, eine Variablenbelegung zu finden, bei der die Ungleichungen erfüllt sind und gleichzeitig die Kostenfunktion minimal wird. Ein Algorithmus zur Lösung dieses Problems ist z.B. der GomoryII-Algorithmus, der auf dem dualen Simplex-Verfahren beruht [Neum75]. Hier wird nur auf die ILP-Formulierung eingegangen. Zunächst werden die benötigten Variablen vorgestellt.

Für jeden Testblock $b \in B_{tot}$ wird eine binäre Variable $b_b \in \{0, 1\}$ verwendet, die angibt, ob der Testblock zur Auswahl eines Tests gehört ($b_b = 1$) oder nicht ($b_b = 0$). Ebenso werden für Register r binäre Variablen l_r, m_r und lm_r eingeführt. Hier zeigt l_r, m_r, $lm_r = 1$ an, daß das Register für den Test zu einem LRSR l_r, MISR m_r oder zu beidem, d.h. einem BILBO lm_r, ausgebaut werden muß. Außerdem gilt: $lm_r = l_r \wedge m_r$. Analog beschreiben die Variablen mg_a, k_a bzw. mgk_a Akkumulatoren a, die als Mustergeneratoren mg_a, als Kompaktierer k_a oder als Kombination mgk_a von beidem eingesetzt werden sollen. Auch hier gilt: $mgk_a = mg_a \wedge k_a$. Mithilfe dieser Variablen läßt sich die Kostenfunktion $C = C_{HW} + \alpha \cdot C_T$ erstellen.

Für jeden ausgewählten Testblock b fallen Zeitkosten $C_T(b)$ an, die wie in Abschnitt 3.2.4.1 beschrieben vorab und allein anhand des Testblocks bestimmt werden können. Die Ausbaukosten

für LRSRs, $C_{LRSR}(r)$, für MISRs, $C_{MISR}(r)$ und BILBOs, $C_{BILBO}(r)$ fallen höchstens einmal an, unabhängig davon, ob ein Testregister nur in einem oder in mehreren ausgewählten Testblöcken enthalten ist. Die Abhängigkeit dieser Kosten vom jeweiligen Register r dient lediglich dazu, vorhandene Register, die nur zu Testregistern ausgebaut zu werden brauchen, von Testregistern zu unterscheiden, die komplett z.B. an Schaltungseingängen oder -ausgängen hinzugefügt werden müssen. Bei letzteren sind die Kosten entsprechend höher. Bei BILBOs muß darauf geachtet werden, daß trotz der Bedingung $lm_r = l_r \wedge m_r$ neben den Kosten C_{BILBO} nicht zusätzlich noch Kosten C_{LRSR} bzw. C_{MISR} in der Kostenfunktion anfallen. Dazu werden die Differenzkosten $C\Delta_{BILBO} = C_{BILBO} - C_{LRSR} - C_{MISR}$ eingeführt. Akkumulatoren werden analog wie Testregister behandelt. Man erhält die Kosten C_{MG}, C_K und C_{MGK} für Mustergeneratoren, Kompaktierer bzw. mustergenerierende und kompaktierende Akkumulatoren sowie die Differenzkosten $C\Delta_{MGK} = C_{MGK} - C_{MG} - C_K$. Sei R die Menge aller Register und A die Menge aller Akkumulatoren. Die komplette Kostenfunktion $C(B)$ zeigt dann Gleichung (3.1).

$$\begin{aligned} C(B) = \quad & \sum_{r \in R} (C_{LRSR}(r) \cdot l_r + C_{MISR}(r) \cdot m_r + C\Delta_{BILBO}(r) \cdot lm_r) + \\ & \sum_{a \in A} (C_{MG} \cdot mg_a + C_K \cdot k_a + C\Delta_{MGK} \cdot mgk_a) + \\ & \alpha \cdot \sum_{b \in B} (C_T(b) \cdot b_b) \end{aligned} \tag{3.1}$$

Wie man sieht, ist die Kostenfunktion eine Linearkombination der Variablen. Es fehlen jetzt noch die Ungleichungen, die die verschiedenen Randbedingungen für die binären Variablen beschreiben.

Ein Test muß für jedes testbare Modul $m \in TM$ mindestens einen Testblock aktivieren, der dieses Modul testet. Sei B_m die Menge der Testblöcke, die das Modul m testen. Dann muß gelten:

$$\forall m \in TM: \qquad \sum_{b \in B_m} b_b \geq 1 \tag{3.2}$$

Ein Akkumulator wird in einem Test als Mustergenerator bzw. Kompaktierer verwendet, wenn er in mindestens einem der ausgewählten Testblöcke als Mustergenerator bzw. Kompaktierer arbeitet. Ebenso werden Register nur einmal zu einem Testregister ausgebaut, selbst wenn sie in mehreren Testblöcken als Testregister eingesetzt werden.

Sei B_{mg_a} die Menge der Testblöcke, die einen Akkumulator mg_a enthalten. Für die Variable mg_a eines jeden dieser Mustergeneratoren gilt dann der logische Ausdruck: $mg_a = \bigvee_{b \in B_{mg_a}} b_b$. Dieser

Ausdruck muß nun durch lineare Ungleichungen ausgedrückt werden. Da die Variablen b_b der Testblöcke binäre Variablen sind, gilt $0 \leq \sum_{b \in B_{mg_a}} b_b \leq |B_{mg_a}|$, wobei $|B_{mg_a}|$ die Mächtigkeit der Menge B_{mg_a} ist. Für $\sum_{b \in B_{mg_a}} b_b > 0$ muß $mg_a = 1$ gelten, sonst $mg_a = 0$. Dies erreicht man durch die Ungleichung (3.3):

$$0 \geq \sum_{b \in B_{mg_a}} b_b - mg_a \cdot |B_{mg_a}| > -|B_{mg_a}| \tag{3.3}$$

Für $\sum_{b \in B_{mg_a}} b_b > 0$ erzwingt die Ungleichung, daß $mg_a = 1$ gilt. Für $\sum_{b \in B_{mg_a}} b_b = 0$ ist die Ungleichung bei $mg_a \leq 0$ erfüllt, und da mg_a eine binäre Variable ist, gilt folglich $mg_a = 0$. Der hier demonstrierte Fall für mustergenerierende Akkumulatoren mg_a, läßt sich analog auf kompaktierende Akkumulatoren k_a bzw. LRSRs l_r und MISRs m_r übertragen. Seien B_{mg_a} und B_{k_a} die Mengen von Testblöcken, die den mustergenerierenden bzw. kompaktierenden Akkumulator mg_a bzw. k_a beinhalten. Für LRSRs l_r und MISRs m_r seien B_{l_r} und B_{m_r} die Mengen der zugehörigen Testblöcke. Dann müssen gemäß Gleichung (3.3) die folgenden Ungleichungen erfüllt sein:

$$\forall a \in A: \quad 0 \geq \sum_{b \in B_{mg_a}} b_b - mg_a \cdot \left|B_{mg_a}\right| > -\left|B_{mg_a}\right| \tag{3.4}$$

$$\forall a \in A: \quad 0 \geq \sum_{b \in B_{k_a}} b_b - k_a \cdot \left|B_{k_a}\right| > -\left|B_{k_a}\right| \tag{3.5}$$

$$\forall r \in R: \quad 0 \geq \sum_{b \in B_{l_r}} b_b - l_r \cdot \left|B_{l_r}\right| > -\left|B_{l_r}\right| \tag{3.6}$$

$$\forall r \in R: \quad 0 \geq \sum_{b \in B_{m_r}} b_b - m_r \cdot \left|B_{m_r}\right| > -\left|B_{m_r}\right| \tag{3.7}$$

Für die Akkumulatoren mgk_a, die sowohl als Mustergeneratoren als auch als Kompaktierer arbeiten sollen, bzw. die BILBOs lm_r gelten die folgenden Aussagen: $mgk_a = mg_a \wedge k_a$ bzw. $lm_r = l_r \wedge m_r$. Für diese binären Variablen erhält man die Ungleichungen:

$$\forall a \in A: \quad 2 > mg_a + k_a - 2mgk_a \geq 0 \tag{3.8}$$

$$\forall r \in R: \quad l_r + m_r - 2lm_r \geq 0 \tag{3.9}$$

Die Gleichungen (3.2) und (3.4) bis (3.9) stellen das komplette System linearer Ungleichungen des ILP-Problems dar, sieht man einmal von den Wertebereich einschränkenden Bedingungen für binäre Variablen ab. Grundsätzlich sind die Variablen eines ILP-Problems nämlich ganzzahlig, für jede binäre Variable x muß deshalb noch die Ungleichung $0 \leq x \leq 1$ erfüllt sein.

3.2.5 Bestimmung einer konkreten Ansteuerung

Nachdem die besten Testblöcke für einen Test ausgewählt worden sind, muß für jeden dieser Testblöcke eine konkrete Ansteuerung bestimmt werden, die die Ansteuerungsbedingungen des Testblocks erfüllt. Anhand dieser Ansteuerung kann letztlich das Teststeuerwerk mit konventionellen Werkzeugen zur Schaltungssynthese erzeugt werden. Von der Wahl der Ansteuerung hängt die Komplexität und damit der Hardware-Aufwand für das Teststeuerwerk ab. Außerdem bestimmt sie den Durchsatz an Testmustern und Testantworten in einem Testblock und somit die Testzeit.

Das hier vorgestellte Verfahren bestimmt für jeden Testblock eine im Hinblick auf Hardware-Aufwand und Testzeit optimierte Ansteuerung. Zuerst wird jedoch auf die Darstellung konkreter Ansteuerungen eingegangen. Anschließend wird gezeigt, wie sich symbolische Ansteuerungsbedingungen für die Bestimmung konkreter Ansteuerungen vereinfachen lassen. Zuletzt wird das eigentliche Verfahren vorgestellt und an einem Beispiel demonstriert.

3.2.5.1 Darstellung konkreter Ansteuerungen

Ist ein Testblock erst einmal aktiviert, sind Mustergenerierung und Kompaktierung zyklische Prozesse. Die charakteristischen Funktionen zeigen dann ein periodisches Verhalten. Zur Darstellung solcher periodischen Funktionen werden einfach die Werte einer einzigen Periode aufgezählt. Startwert ist dabei der Wert im Zyklus t_0, mit dem der periodische Prozeß beginnt. Beispielsweise steht [011] für 011011011... und [1] für die schon oben beschriebene konstante Funktion **1**. Die Periode π einer periodischen Funktion F ist dabei definiert als die kleinste natürliche Zahl, für die gilt: $\forall t \geq t_0$: $F(t+\pi) = F(t)$.

3.2.5.2 Vereinfachung symbolischer Ansteuerungsbedingungen

Normalerweise sind viele konkrete Ansteuerungen für eine symbolische Ansteuerungsfunktion denkbar. Manchmal schränken die verschiedenen Ansteuerungsbedingungen jedoch die Wahlmög-

lichkeiten derart ein, daß für eine symbolische Ansteuerungsfunktion nur noch eine einzige konkrete Ansteuerung möglich ist. Ein Beispiel ist die Überdeckungsbedingung $N(L, H) \geq V(L, H)$, die bei Akkumulatoren auftritt, deren Rückkopplungspfad nur ein Register, das Akkuregister, beinhaltet. Tatsächlich gilt der folgende Satz:

Satz: $N(L, H) \geq V(L, H) \Leftrightarrow N(L, H) = [1]; \quad L, H \neq [0]$

Beweis: Ein Register kann einen bestimmten Wert nur bei aktiviertem Ladesignal übernehmen. Der Wert erscheint dann ab dem folgenden Taktzyklus am Registerausgang. Für ein beliebiges propagierendes Register gelten offensichtlich folgende Aussagen:

$$V(L, H)(t) = 1 \quad \Rightarrow \quad L(t) = 1 \tag{3.10}$$

$$V(L, H)(t) = 1 \quad \Rightarrow \quad N(L, H)(t+1) = 1 \tag{3.11}$$

$$L(t) = 1 \ \wedge \ V(L, H)(t) = 0 \quad \Rightarrow \quad N(L, H)(t+1) = 0 \tag{3.12}$$

Ein Wert bleibt solange gespeichert, bis ein neuer Wert geladen wird, d.h. es gilt:

$$L(t) = 0 \quad \Rightarrow \quad N(t) = N(t+1) \tag{3.13}$$

Angenommen es gelte $N(L, H) \geq V(L, H)$ und $N(L, H) \neq [1]$. Dann existiert ein Zyklus t_1 mit $N(L, H)(t_1) = 0$. $N(L, H)$ bleibt wegen der Aussagen (3.10) bis (3.13) solange 0, bis wieder $V(L, H)(t_2) = 1$, $t_2 \geq t_1$, gilt. Da alle Funktionen periodisch sind und wegen $L, H \neq [0]$ existiert ein solcher Zyklus t_2. Für diesen Zyklus gilt noch $N(L, H)(t_2) = 0$. Somit überdeckt $N(L, H)$ im Widerspruch zur Annahme $V(L, H)$ nicht. Es gilt also $N(L, H) \geq V(L, H) \Rightarrow N(L, H) = [1]$. Umgekehrt überdeckt die Funktion [1] jede beliebige Funktion, und der Satz stimmt.

❑

Bei einem primitiven Akkumulator mit einem einzigen Register im Rückkopplungspfad kann man daher die symbolische Funktion $N(L, H)$ durch die konkrete Funktion [1] ersetzen. Man erhält dann die Bedingung $[1] \geq V(L, H)$. Eine Bedingung $[1] \geq F$ ist offensichtlich immer erfüllt, man kann sie weglassen. Für primitive Akkumulatoren läßt sich ein weiterer Satz zur Vereinfachung anwenden, so daß die Funktion $V(L, H)$ entfällt:

Satz: $N(L, H) = [1] \Leftrightarrow V(L, H) = L$

Beweis: Angenommen es gelte $N(L, H) = [1]$ und $V(L, H) \neq L$, dann existiert ein Zyklus t mit $L(t) = 1$ und $V(L, H)(t) = 0$. Wegen Aussage (3.12) gilt dann $N(L, H)(t+1) = 0$ und folglich $N(L, H) \neq [1]$ im Widerspruch zur Annahme. Es gilt also $N(L, H) = [1] \Rightarrow V(L, H) = L$. Umgekehrt wird

mit V(L, H) = L immer derselbe Wert geladen und erscheint folglich immer am Ausgang. Der Satz ist somit korrekt.

❑

3.2.5.3 Optimierungsverfahren

Wie schon oben erwähnt, optimiert das hier vorgestellte Verfahren die Ansteuerung eines Testblocks im Hinblick auf den Hardware-Aufwand für das Teststeuerwerk und die Testzeit. Das Teststeuerwerk steuert dabei die sogenannten *Steuersignale* des Testblocks. Neben den Steuersignalen wie Ladesignale für Register, Auswahlsignale für Multiplexer usw. sind das noch die Schaltungseingänge, über die evt. Daten in den Testblock eingespeist werden müssen. Tatsächlich stellt das gefundene Ergebnis i.a. einen guten Kompromiß zwischen Hardware-Aufwand und Testzeit dar. Ein anderes Verfahren, das speziell die Testzeit minimiert, ist in [StMa99] beschrieben.

Das vorliegende Optimierungsverfahren sucht für alle Steuersignale eines Testblocks Ansteuerungen, so daß die längste Periode möglichst kurz ist. Die Heuristik beruht auf folgenden Beobachtungen:

- Ansteuerungen mit kurzen Perioden lassen sich i.a. mit weniger Hardware erzeugen als solche mit langen Perioden. Z.B. entspricht die Funktion [1] einfach einem konstanten Signal. Für die Periode 2 ([01], [10]) genügt ein Toggle-Flipflop. Für Perioden 3 und 4 benötigt man hingegen schon mindestens zwei Flipflops, usw.
- Je kürzer die Perioden, um so größer die Chance, daß die Ansteuerungen mehrerer Signale identisch sind und somit Steuerhardware eingespart werden kann. Gleichzeitig steigt die Wahrscheinlichkeit, daß die Steuerhardware eines Testblocks zumindest teilweise auch für einen anderen Testblock verwendet werden kann.
- Bei kurzen Perioden ist der Durchsatz an Testmustern und Testantworten i.a. höher als bei langen Perioden. Die Testzeit ist dann kürzer.

Alle Ansteuerungsbedingungen sind von den unabhängigen Funktionen H an den Ausgängen der Testblockkerne sowie von den Funktionen L für die Ladesignale von Registern abgeleitet. Das Optimierungsverfahren bestimmt nun für einen bereits symbolisch vereinfachten Testblock die Menge aller unabhängigen symbolischen Funktionen H und L. Ausgehend von diesen Funktionen ermittelt es mit einem Branch-and-Bound-Algorithmus eine konkrete Ansteuerung, die alle Ansteuerungsbedingungen erfüllt und bei der die längste auftretende Periode minimal ist. Bei

einem zusammengesetzten Testblock, der sich aus mehreren unabhängig voneinander ansteuerbaren Testblöcken zusammensetzt, wird die Ansteuerung für jeden dieser Testblöcke einzeln optimiert. Den groben Aufbau des Optimierungsalgorithmus zeigt Bild 3.14.

```
Lege eine obere Grenze π_max für Perioden fest;
Zerlege den Testblock in unabhängige Testblöcke;
Für jeden dieser unabhängigen Testblöcke:
    Λ := Menge der unabhängigen symbolischen Funktionen in den Ansteuerungsbedingungen;
    Optimiere(Λ, Λ);
Fasse die optimierten Ansteuerungen der unabhängigen Testblöcke zusammen;
Ende.

Prozedur Optimiere(Λ, Γ)
    Falls Λ ≠ {},                                              /* Lege Perioden fest */
        L := erstes Element von Λ;
        Λ := Λ / {L};
        Für jede Periode π von 1 bis π_max:
            Periode π_L von L := π;
            Falls π = 1, L := [1];                    /* Periode [0] ergibt keinen Sinn */
            Falls keine Widersprüche für Ansteuerungsbedingungen,
                Optimiere(Λ, Γ);
    sonst:                              /* Alle Perioden festgelegt. Bestimme Ansteuerungen */
        Falls Γ≠ {},
            L := erstes Element von Γ;
            Γ := Γ / {L};
            Für jede Ansteuerung a ≠ [0] mit Periode π_L:
                Ansteuerung von L := a;
                Falls keine Widersprüche in Ansteuerungsbedingungen,
                    Optimiere({}, Γ);
        sonst:                                                 /* Ansteuerung komplett */
            π_m := Längste Periode aller Steuerfunktionen;
            Falls π_m < π_max ∨ (π_m = π_max ∧ noch kein Zwischenoptimum gefunden),
                Aktuelle Ansteuerung ist neues Zwischenoptimum;
                π_max := π_m – 1;
    return;
```

Bild 3.14: Algorithmus zur Optimierung der Ansteuerung eines Testblocks

Jedesmal, wenn die Periode oder Ansteuerung einer Funktion festgelegt worden ist, werden die Ansteuerungsbedingungen auf Widersprüche hin untersucht. Oft lassen sich solche Widersprüche bereits allein anhand der Kenntnis der Perioden feststellen.

Angenommen man hat n Funktionen F_1, ..., F_n, die alle disjunkt sein müssen. Für eine Funktion F_i mit der Periode π_i gilt in mindestens $\frac{1}{\pi_i}$-tel aller Taktzyklen $F_i(t) = 1$. Da disjunkte Funktionen niemals im selben Zyklus den Wert 1 annehmen, gilt offensichtlich: $\sum_{i=0}^{n} \frac{1}{\pi_i} \leq 1$, andernfalls müßten mindestens zwei der n Funktionen einander überlappen. O.B.d.A. seien nun die ersten m Perioden π_1, ..., π_m bekannt. Mit der Obergrenze π_{max} für alle Perioden erhält man:

$$\sum_{i=0}^{m} \frac{1}{\pi_i} \leq 1 - \frac{n-m}{\pi_{max}} \tag{3.14}$$

Jedesmal wenn die Periode einer weiteren dieser n Funktionen festgelegt wird, läßt sich anhand dieser Gleichung feststellen, ob die Disjunktheitsbedingungen für die n Funktionen verletzt werden.

Die obige Bedingung ist notwendig, aber nicht hinreichend für die Disjunktheit. Eine weitere Bedingung, mit der die Disjunktheit zweier Funktionen F und G überprüft werden kann, beruht auf dem *größten gemeinsamen Teiler (ggT*) ihrer Perioden π_F und π_G:

Satz: $ggT(\pi_F, \pi_G) = 1 \Rightarrow$ F überlappt G

Beweis: Sei $ggT(\pi_F, \pi_G) = 1$. Wegen F, G $\neq [0]$ existiert ein Zyklus f bzw. g mit $F(f) = G(g) = 1$. Da F und G zudem periodisch sind, gilt: $\forall m \in \mathbb{N}_0$: $F(m \cdot \pi_F + f) = 1$ bzw. $\forall n \in \mathbb{N}_0$: $G(n \cdot \pi_G + g) = 1$. F überlappt G folglich auf jeden Fall, wenn für beliebige f und g die Gleichung

$$m \cdot \pi_F + f = n \cdot \pi_G + g \tag{3.15}$$

gelöst werden kann. Aus Gleichung (3.15) erhält man:

$$m \cdot \pi_F = n \cdot \pi_G + g - f$$

Diese Gleichung ist lösbar, wenn die folgende Gleichung lösbar ist:

$$(m \cdot \pi_F) \bmod \pi_G = (g - f) \bmod \pi_G \tag{3.16}$$

Die Funktion $G(m) = (m \cdot \pi_F) \bmod \pi_G$ ist eine periodische Funktion mit der Periode π_G, weil π_F, $2\pi_F$, ..., $(\pi_G\text{-}1) \cdot \pi_F$ nicht durch π_G teilbar sind. Zudem sind alle Funktionswerte von G(m) innerhalb einer Periode verschieden. G(m) erzeugt folglich die Werte 0, 1, ..., π_G-1, also alle Werte, die die rechte

Seite der Gleichung (3.16) annehmen kann. Es existiert demnach immer eine Lösung für Gleichung (3.16) und folglich für Gleichung (3.15). Damit ist die Behauptung des Satzes bewiesen.

❑

Nachdem alle Perioden festgelegt sind, werden die konkreten Ansteuerungen der Funktionen spezifiziert. Oft schränkt die Spezifizierung einer Funktion andere Funktionen so ein, daß unspezifizierte Funktionen bereits teilweise mitspezifiziert werden können. I.a. sind solche Teilspezifikationen einfache Schlußfolgerungen der Aussagen (3.10) bis (3.13) für Register bzw. der Definitionen von Überdeckungs-, Überlappungs- und Disjunktheitsbedingungen. Gilt beispielsweise $F \geq G$ und $G(t_1) = 1$, dann muß offensichtlich auch $F(t_1) = 1$ gelten. Bei bekanntem G kann man somit für F alle Spezifikationen mit $F(t_1) = 0$ von vornherein ausschließen.

Nach jeder Spezifizierung einer Funktion F ist für jede Periode die Anzahl e der Zyklen mit $F(t) = 1$ bekannt. Damit läßt sich z.B. Gleichung (3.14) verfeinern. Seien also $F_1, ..., F_n$ wieder n disjunkte Funktionen, wobei m davon, $F_1, ..., F_m$, spezifiziert seinen. Für eine Funktion F_i mit Periode π_i sei e_i die Anzahl der Zyklen pro Periode mit $F_i(t) = 1$. Dann muß gelten: $\sum_{i=0}^{m} \frac{e_i}{\pi_i} \leq 1 - \frac{n-m}{\pi_{max}}$. Andernfalls sind die n Funktionen nicht alle disjunkt.

Mit den oben beschriebenen Methoden läßt sich der Suchraum üblicherweise soweit einschränken, daß eine konkrete Ansteuerung nach kurzer Zeit gefunden wird. Voraussetzung dafür ist, daß eine solche Ansteuerung überhaupt existieren kann. Tatsächlich existiert immer eine Lösung, solange der Datenfluß eines Testblocks keine Zyklen enthält. Solche Zyklen treten jedoch bei Akkumulatoren auf, bei denen der Inhalt des Akkuregisters wegen der Rückkopplung von sich selbst abhängt. Problematisch wird es, wenn alle Register innerhalb des Rückkopplungspfades eines Akkumulators ebenfalls zur Propagierung anderer Muster verwendet werden, d.h. wenn für jedes dieser Register Disjunktheitsbedingungen beachtet werden müssen. In diesem Fall kann es sein, daß bei der Propagierung dieser Muster immer der Inhalt des Akkuregisters aus dem Akkumulatorzyklus verloren gehen muß, der Akkumulator also nicht arbeiten kann. Es tritt dann ein Widerspruch zwischen den Disjunktheitsbedingungen der beteiligten Register und der Überdeckungsbedingung auf, mit der der Zyklus geschlossen wird. Gerade bei Akkumulatoren mit mehreren Registern im Rückkopplungspfad sind solche Widersprüche anhand der symbolischen Funktionen oft nur sehr schwer zu entdecken, und es besteht die Gefahr, daß bei einem unentdeckten Widerspruch viel Rechenzeit für die sinnlose Suche nach einer konkreten Ansteuerung vergeudet wird. Deshalb werden nur solche Testblöcke betrachtet, bei denen jeder Akkumulator mindestens ein Register seines Rückkopplungspfades exklusiv für sich reserviert hat, d.h. über das keine anderen Muster als der Akkuregi-

sterinhalt propagiert werden. Ein solches Register schneidet quasi den Akkumulatorzyklus, da es den Akkuregisterinhalt beliebig lange speichern kann, ohne die Funktion des übrigen Testblocks zu beeinträchtigen. Die Erzeugung und Kompaktierung aufeinanderfolgender Testantworten ist dadurch zeitlich entkoppelt, die Existenz einer konkreten Ansteuerung garantiert.

Der gesamte Optimierungsalgorithmus soll nun an dem bereits vorgestellten Testblock aus Bild 3.11e erläutert werden. Die vorgegebene Funktion am Ausgang des Testblockkerns war T, die Register des Testblocks wurden mit den Funktionen L_1, ..., L_5 angesteuert, der 0-Eingang des einzigen Multiplexers mit T, der 1-Eingang mit $V(L_2, V(L_3, T))$. Die Ansteuerungsbedingungen waren:

$$[1] \geq T$$

$$[1] \geq V(L_3, T)$$

$$[1] \geq V(L_2, V(L_3, T))$$

$$L_5 \text{ überlappt } T$$

$$V(L_2, V(L_3, T)) + T < 2$$

Da die Funktion [1] alle Funktionen überdeckt, sind die drei Überdeckungsbedingungen immer erfüllt und brauchen nicht weiter berücksichtigt zu werden. Der Testblock ist nicht aus anderen Testblöcken zusammengesetzt, die Menge der unspezifizierten, unabhängigen Funktionen in den übrigen beiden Ansteuerungsbedingungen ist $\Lambda = \{L_5, T, L_2, L_3\}$. Für die obere Periodengrenze gelte $\pi_{max} = 5$.

Zuerst werden der Reihe nach die Perioden bestimmt. Zu Beginn erhält L_5 die Periode $\pi_5 = 1$, damit gilt $L_5 = [1]$. Die Funktion T muß disjunkt zu $V(L_2, V(L_3, T))$ sein. Wegen Gleichung (3.14) gilt dann $\pi_T > 1$, die kleinste Periode für T ist somit $\pi_T = 2$. Die Funktionen L_2 und L_3 erhalten wie L_5 die Periode π_2, $\pi_3 = 1$ und sind somit identisch [1]. Nachdem die Perioden festgelegt sind, werden die Funktionen spezifiziert. Die einzige noch nicht spezifizierte Funktion ist T mit den möglichen Ansteuerungen [10] und [01]. Sei T = [10], dann erhält man $V(L_3, [10]) = [01]$ und $V(L_2, V(L_3, T)) = V([1], [01]) = [10] = T$. Damit ist jedoch die Disjunktheitsbedingung $V(L_2, V(L_3, T)) + T < 2$ verletzt. Das gleiche passiert mit T = [01]. Für die gewählten Perioden existiert keine gültige Spezifikation, deshalb wird mit der Periode $\pi_3 = 2$ für L_3 fortgefahren. Diesmal erhält man mit T = [10] und $L_3 = [10]$ schließlich $V(L_2, V(L_3, T)) = [01]$ und die Disjunktheitsbedingung ist erfüllt. Da

L_5 = [1] auch die Funktion T überlappt, hat man eine erste gültige Lösung gefunden. Die längste Periode dieser Lösung ist $\pi_m = 2$. Die Obergrenze wird daher auf $\pi_{max} = \pi_m - 1 = 1$ erniedrigt. Da $\pi_T > 1$ gelten muß, kann keine bessere Lösung mehr gefunden werden. Man erhält also die optimierten Spezifikationen L_5, L_2 = [1], T, L_3 = [10] und V(L_2, V(L_3, T)) = [01]. Die noch nicht spezifizierten Funktionen L_1 und L_4 dürfen beliebig gewählt werden, z.B. L_1, L_4 = [1]. Mit Ausnahme von L_3 können die Ladesignale also ständig aktiviert bleiben. Das Multiplexerauswahlsignal beginnt mit dem 0-Eingang und wechselt dann mit jedem Zyklus zwischen den beiden Eingängen hin und her, während für L_3 das invertierte Auswahlsignal verwendet wird.

3.2.6 Experimentelle Ergebnisse

Das komplette Verfahren zur Konfiguration eines optimierten Selbsttests unter Berücksichtigung von Akkumulatoren wurde im Rahmen dieser Arbeit als C-Programm implementiert und auf verschiedene Benchmarkschaltungen aus der Literatur zur High-Level-Synthese angewendet. Alle Experimente wurden auf einer SUN Sparc Ultra 10 durchgeführt, zur Lösung des ILP-Problems bei der Auswahl der besten Testblöcke kam das Programm „lp_solve“ der Eindhoven University of Technology zum Einsatz, das dort für jedermann frei erhältlich ist [Berk99].

Als Beispielschaltungen wurden die IIR-Filter (IIR: infinite impulse response) *IIRa* (paralleler IIR-Filter vierter Ordnung), *IIRb* und *IIRc* (beides kaskadierte IIR-Filter vierter Ordnung), der digitale elliptische Wellenfilter fünfter Ordnung *EWF* und der Differentialgleichungslöser *Paulin* gewählt. Alle Schaltungen sind die Endprodukte verschiedener High-Level-Synthese Werkzeuge. Ihre Register-Transfer-Ebenen Beschreibungen wurden [PoDR95a, Bild 2], [PoDR95b, Bilder 3b, 2b und 8] bzw. [GhRJ96, Bild 1] entnommen. Zur Schaltung Paulin wurde ein zusätzlicher 2:1-Multiplexer hinzugefügt, um die Testbarkeit des Subtrahierers zu verbessern. Die Schaltung EWF_1 ist identisch mit EWF, allerdings durften die Testblöcke jeweils nur ein Register pro Registerbank verwenden. Eine Registerbank ist dabei eine Ansammlung von Registern, von denen pro Taktzyklus nur ein Register ausgelesen und ein Register beschrieben werden können. Schaltung EWF+ ist EWF mit einem zusätzlichen Multiplexereingang, der mit dem Ausgang eines Addierers verbunden wurde.

Tabelle 3.3 zeigt für jede Schaltung die Anzahl der Schaltungseingänge (Ein) und -ausgänge (Aus) sowie der konstanten Eingänge (Konst), ferner die Zahl der Addierer und Subtrahierer (+ –), der Multiplizierer (*), Register (Reg) und Multiplexer (Mux). Spalte *FE* gibt die Zahl der explizit zu

Schaltung	Ein	Aus	Konst	+ –	*	Reg	Mux	FE	AR_k	AR_v
IIRa	1	1	3	4	3	17	9	7	4	8
IIRb	1	1	4	2	3	12	5	5	1	1
IIRc	1	1	4	2	3	12	8	5	3	11
EWF	1	1	3	3	3	23	13	6	19	83
EWF_1	1	1	3	3	3	23	13	6	6	14
EWF+	1	1	3	3	3	23	13	6	19	143
Paulin	2	2	1	2	2	7	11	4	3	6

Tabelle 3.3: Daten der Beispielschaltungen

testenden funktionalen Einheiten an, hier die Summe der Addierer, Subtrahierer und Multiplizierer (vgl. Abschnitt 3.2.3.1). Die beiden letzten Spalten enthalten die Zahl der gefundenen Akkumulatorrümpfe bei konstant durchgeschalteten Propagierungspfaden (AR_k) bzw. bei variabler Ansteuerung (AR_v). Alle Multiplizierer außer diejenigen von Paulin haben einen konstanten Eingang, d.h. einen Eingang, der nur über konstante Eingänge gespeist werden kann. Für den Test dieser Multiplizierer genügt es, nur den jeweils anderen Eingang mit Testmustern zu versorgen.

Für jede der Schaltungen wurden vier Experimente durchgeführt. Im ersten Experiment konnte ein primitiver Testblock jeweils nur eine einzige Funktionseinheit FE testen, und Propagierungspfade mußten konstant durchgeschaltet bleiben. Im zweiten Experiment wurden Testbarkeitsmaße hinzugezogen, ein primitiver Testblock konnte also mehrere Module auf einmal testen. Im dritten Experiment wurde zusätzlich die Beschränkung auf konstant durchgeschaltete Pfade aufgehoben, außerdem konnten Konstanten zu Seiteneingängen von Modulen transportiert werden. Allerdings durfte ein Pfad nicht mehrfach über dasselbe Modul verlaufen, d.h. die Schaltungsstruktur eines Testblocks durfte neben den notwendigen Zyklen in Akkumulatoren keine weiteren Zyklen enthalten. Im vierten Experiment waren auch solche Strukturzyklen erlaubt – im Unterschied zu Strukturzyklen sind Zyklen im Datenfluß von Testblöcken immer auf die Zyklen ihrer Akkumulatoren beschränkt. Bei allen Experimenten mußte für die Zufälligkeit ζ bzw. die Beobachtbarkeit β eines Testmusters am Eingang bzw. Ausgang eines testbaren Moduls $\zeta \geq 0{,}9$ und $\beta \geq 0{,}8$ gelten.

Bild 3.15 zeigt für jedes der Experimente den Hardware-Mehraufwand für die Mustergeneratoren und Kompaktierer bezogen auf den Mehraufwand eines LRSR, d.h. ein LRSR hat den Mehraufwand 1. Für MISRs und BILBOs wurden die Anzahlen der zusätzlichen Transistoren pro Registerzelle mit denen eines LRSR verglichen. Diese sind 12 Transistoren bei einem LRSR, 13 bei einem

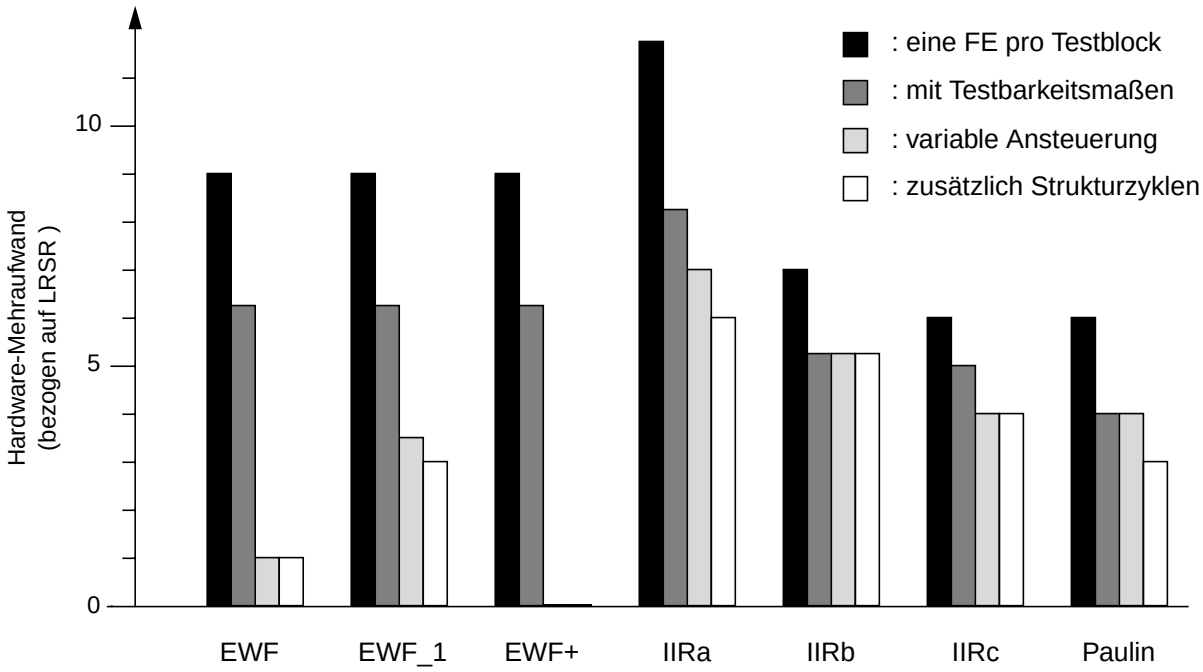


Bild 3.15: Hardware-Mehraufwand für Testregister

MISR ohne Schiebebetrieb und 15 bei einem BILBO. Der relative Mehraufwand ist dann 1,08 für ein MISR und 1,25 bei einem BILBO. Die Zusatzhardware für die Rückkopplung und die Auswahl der Betriebsarten wurde vernachlässigt. Sie fällt im Vergleich zur Erweiterung der üblicherweise vielen Registerzellen pro Register kaum ins Gewicht. Da die betrachteten Schaltungen oft selbst als Teil einer größeren Schaltung arbeiten, wurde angenommen, daß an den Schaltungseingängen und -ausgängen bereits Register anliegen. Die Erweiterungskosten für externe und interne Testregister sind dann identisch. Da Akkumulatoren praktisch ohne zusätzliche Hardware auskommen, wurden ihre Kosten vernachlässigt. Der Gewichtungsfaktor α der Kostenfunktion wurde sehr klein gewählt, die Testzeit spielte also nur eine untergeordnete Rolle.

Die Ergebnisse zeigen eine deutliche Reduktion des Mehraufwands durch die Verwendung von Testbarkeitsmaßen (jeweils erste und zweite Säule). Bei den Schaltungen EWF, EWF_1 und EWF+ brachten die variable Ansteuerung und die Konstanten an Seitenangängen eine weitere merkliche Verbesserung (dritte Säule). Bei den Schaltungen IIRa und IIRc fiel dieser Effekt geringer aus, da die konstanten Eingänge der Multiplizierer nur mit Werten gespeist werden konnten, die zum Transport von Testmustern bzw. -antworten über den jeweils anderen Eingang ungeeignet waren. Diese Multiplizierer blockierten daher viele potentielle Propagierungspfade. Bei der Schaltung IIRb waren es sogar zuviele für eine weitere Hardware-Reduktion. Die vergleichsweise kleine Schaltung Paulin beinhaltet hingegen ohnehin nur wenige Propagierungsmöglichkeiten. Diese sind erst wieder ausreichend, wenn Strukturzyklen zugelassen werden. Auch bei den Schaltungen

EWF_1 und IIRa läßt sich der Hardware-Aufwand mit Strukturzyklen noch einmal verringern (vierte Säule), bei EWF und EWF+ konnte immerhin die Testzeit verkürzt werden (s. Zeitkosten C_T in Tabelle 3.5).

Bild 3.16 zeigt die Einsparung an Testregistern, die durch die Berücksichtigung von Akkumulatoren erzielt werden konnten. Bei den Schaltungen EWF_1, IIRa, IIRb, IIRc und Paulin war nur eine leichte Verringerung des Hardware-Mehraufwands durch Akkumulatoren erreichbar. Die Ursache hierfür liegt in der geringen Anzahl möglicher Akkumulatorrümpfe bei diesen Schaltungen (s. Tabelle 3.3). Meist konnten lediglich BILBOs durch billigere LRSRs oder MISRs ersetzt werden. Bei EWF und EWF+ war die Einsparung dagegen wegen der Vielzahl verschiedener Akkumulatorrümpfe deutlich größer. In den Experimenten 3 und 4 fielen bei EWF zwei BILBOs weg, bei EWF+ konnte sogar auf alle Testregister, zwei BILBOs und ein LRSR, verzichtet werden.

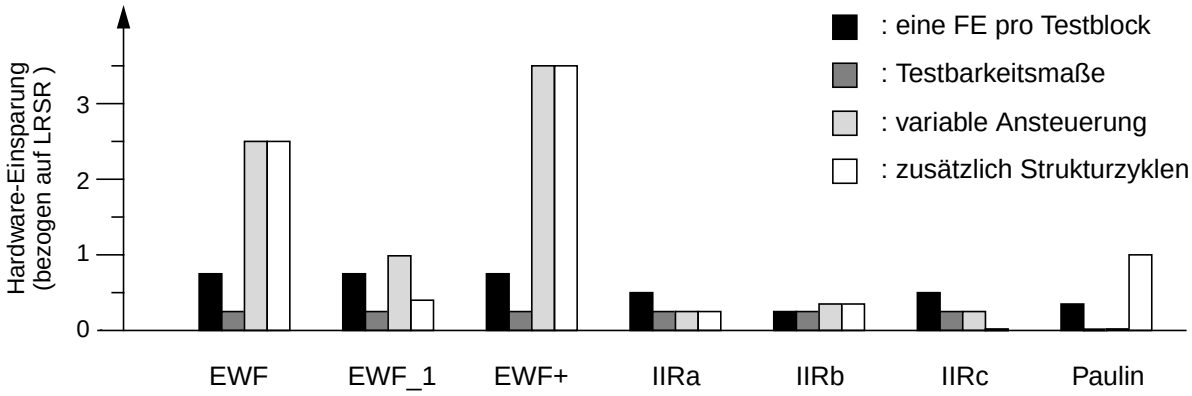


Bild 3.16: Hardware-Einsparung durch Akkumulatoren

Die Rechenzeiten für die vier Experimente geben die Tabellen 3.4 und 3.5 wieder. Die Zeiten zur Bestimmung der primitiven bzw. der zusammengesetzten Testblockkernrümpfe (*prim*. bzw. *zus. TKR*) sind getrennt aufgelistet, und zwar in Stunden, Minuten und Sekunden. Bei Zeiten unter einer Minute sind auch die Zehntelsekunden angegeben. Die Rechenzeiten zur Bestimmung der Akkumulatoren fehlen hier. Sie betrugen höchstens wenige Sekunden. Nach der Konfiguration der primitiven Kerne wurde jedesmal der Algorithmus zur Auswahl der besten Testblöcke gestartet, um bereits vor dem Zusammensetzen der Testblockkernrümpfe eine gute obere Kostenschranke zu erhalten, mit der die Zahl lohnender Testblöcke weiter eingeschränkt werden kann. Dadurch wird sowohl das Zusammensetzen als auch die abschließende Auswahl der Testblöcke beschleunigt. Spalte *ILP* gibt die Rechenzeit zum Lösen des zwischenzeitlichen ILP-Problems an, die jeweils darauffolgende Spalte die Anzahl der dabei zu berücksichtigenden Testblöcke. Die Zeiten für das

Schaltung	Experiment 1					Experiment 2				
	Rechenzeit (Std:Min:Sek)			Anz. TB	C_T	Rechenzeit (Std:Min:Sek)			Anz. TB	C_T
	prim. TKR	zus. TKR	ILP			prim. TKR	zus. TKR	ILP		
IIRa	0,5	1,2	0,6	20	4	1:32	9:44	7:47	687	3
IIRb	0,2	1,0	0,6	18	4	8,4	2,2	2,5	150	3
IIRc	0,8	0,1	0,5	12	5	42,6	16,1	31,3	321	3
EWF	1,0	2,9	1,2	21	4	26,4	8,5	1:27	236	4
EWF_1	0,4	0,1	0,6	7	4	7,1	1,3	0,8	79	4
EWF+	1,1	2,8	1,6	21	4	30,7	9,2	2:29	261	4
Paulin	0,5	1,2	0,6	21	3	2,3	0,1	0,5	21	1

Tabelle 3.4: Rechenzeiten für Experimente 1 und 2

Schaltung	Experiment 3					Experiment 4				
	Rechenzeit (Std:Min:Sek)			Anz. TB	C_T	Rechenzeit (Std:Min:Sek)			Anz. TB	C_T
	prim. TKR	zus. TKR	ILP			prim. TKR	zus. TKR	ILP		
IIRa	7:26	25:21	18,9	1042	3	13:27	10:30	8,8	1087	4
IIRb	11,5	11,5	4,0	192	3	22,1	11,7	3,9	192	3
IIRc	27:38	4:35	4,1	438	4	53:03	3:05	7,8	662	4
EWF	4:11	3,5	1,6	444	10	4:17:52	3,5	1,6	780	8
EWF_1	1:10	1,5	1,0	137	8	7:09	2,8	1,5	169	7
EWF+	18,1	1,8	0,7	6	11	33:35	6,1	0,8	10	9
Paulin	33,5	0,5	0,5	35	1	5:00	0,6	0,7	76	3

Tabelle 3.5: Rechenzeiten für Experimente 3 und 4

abschließende ILP-Problem lagen wegen der nun bekannten Kostenobergrenze höchstens in derselben Größenordnung, meist waren es nur Bruchteile einer Sekunde. Die jeweils letzte Spalte enthält als Maß für die Testzeit die Zeitkosten C_T (s. Abschnitt 3.2.4.1).

Wie zu erwarten, sind die Rechenzeiten für Experiment 1 wegen der vielen Einschränkungen und folglich geringen Zahl möglicher Testblöcke am kürzesten. Wenige Sekunden genügen, um einen optimierten Test zu erhalten. Die Testblöcke sind hier auch am kleinsten, was die Chancen steigen läßt, daß mehrere Testblöcke parallel arbeiten können, um Testzeit einzusparen. Mit Ausnahme von Schaltung IIRc konnten deshalb bei optimierten Hardware-Kosten durch zusammengesetzte Test-

blöcke noch einmal 20 ... 57 % an Testzeit eingespart werden. Tatsächlich führte das Zusammensetzen bei den drei anderen Experimenten mit wenigen Ausnahmen zu keiner Verkürzung der Testzeit. Hier waren die lohnenden primitiven Testblöcke im Verhältnis zur gesamten Schaltung einfach zu groß. Möglicherweise ist es sinnvoll, durch eine kleine Stichprobe das Verhältnis zwischen lohnenden und nicht lohnenden zusammengesetzten Testblöcken abzuschätzen. Ist das Verhältnis zu schlecht, wird auf das Zusammensetzen zu Gunsten der Rechenzeit verzichtet.

Die Tabellen verdeutlichen auch einen anderen Zusammenhang. Je mehr Einschränkungen bei der Suche nach primitiven Testblöcken aufgehoben werden, um so größer ist die Rechenzeit. Bei Experiment 4 wird die Rechenzeit maximal und liegt i.a. bei mehreren Minuten bis zu einer bzw. vier Stunden bei IIRc bzw. EWF. Gute Lösungen in deutlich kürzerer Zeit erhält man jedoch schon bei Experiment 3, bei dem keine Strukturzyklen zugelassen sind.

Der Anstieg in der Rechenzeit drückt sich auch in der gefundenen Anzahl lohnender primitiver Testblöcke aus, die für das zwischenzeitliche ILP-Problem berücksichtigt werden müssen. Einzige Ausnahme ist die Schaltung EWF+. Hier wurde erkannt, daß eine Lösung ohne Testregister existiert, sobald eine variable Ansteuerung zugelassen wird. Der Algorithmus betrachtete daher nur noch Testblöcke ohne Testregister, was die Rechenzeit deutlich verkürzt. Andernfalls wäre die Rechenzeit immer etwas länger als bei der geringfügig einfacheren Schaltung EWF. Im Gegensatz dazu sind die Zeiten für das Zusammensetzen der Testblockkerne bzw. zur Lösung des ILP-Problems weit unabhängiger von der Zahl lohnender primitiver Testblöcke. Tatsächlich scheint es hier wichtiger zu sein, wie nah die zwischenzeitlich bestimmte Kostenobergrenze an den endgültigen Kosten liegt, wie viele Lösungen ähnlich geringe Kosten aufweisen bzw. wie teuer einzelne primitive Testblöcke im Vergleich zur Kostenobergrenze sind. Je mehr eine optimierte Lösung aus der Masse möglicher Lösungen herausragt, um so stärker kann der Suchraum eingeschränkt werden.

Die Testzeit nimmt in der Regel ab, sobald mehrere Module von einem Testblock getestet werden dürfen, und steigt wieder bei variablen Ansteuerungen. Der Algorithmus zur Bestimmung einer optimalen konkreten Ansteuerung ist nur bei variabler Ansteuerung von Nutzen, bei konstant durchgeschalteten Propagierungspfaden wird jedes Signal entsprechend der charakteristischen Funktion **1** angesteuert. Tabelle 3.6 zeigt für die Experimente 3 und 4 mit variabler Ansteuerung für jeden ausgewählten Testblock die längste Periode und in Spalte C_T die Summe der Perioden als tatsächliche Zeitkosten C_T. Die Rechenzeit zur Bestimmung der optimalen Ansteuerung ist in Spalte *Zeit* in Minuten und Sekunden wiedergegeben. Maximal fünf Testblöcke wurden ausgewählt, die längste Periode ist vier. Die tatsächlichen Zeitkosten sind identisch mit den abgeschätzten Zeitko-

Schaltung	Experiment 3							Experiment 4					
	Testblock					C_T	Zeit	Testblock				C_T	Zeit
	1	2	3	4	5			1	2	3	4		
IIRa	1	1	1	–	–	3	0,0	1	2	1	–	4	0,1
IIRb	1	1	1	–	–	3	0,0	1	1	1	–	3	0,0
IIRc	1	1	1	1	–	4	0,0	1	1	1	1	4	0,0
EWF	2	2	4	2	–	10	45:44	3	3	2	–	8	0,6
EWF_1	1	3	1	3	–	8	0,1	1	2	3	1	7	0,1
EWF+	3	1	3	2	3	11	5,9	4	3	2	–	9	0,6
Paulin	1	–	–	–	–	1	0,0	2	1	–	–	3	0,1

Tabelle 3.6: Perioden der konkreten Ansteuerungen für Experimente 3 und 4

sten aus Tabelle 3.5, ein Hinweis für die hohe Qualität der Abschätzung. Die Rechenzeit betrug fast durchweg nur Bruchteile einer Sekunde. Augenfällige Ausnahme ist mit 45 Minuten Schaltung EWF in Experiment 3. Ursache war der dritte ausgewählte Testblock. Löscht man jedoch diesen Testblock und wiederholt die Auswahl der besten Testblöcke, so erhält man eine kostenidentische Lösung bereits nach Bruchteilen einer Sekunde. Dieser Befund legt nahe, die Berechnung einer konkreten Ansteuerung bei ausbleibendem Erfolg nach relativ kurzer Zeit abzubrechen und stattdessen mit einer neuen Auswahl von Testblöcken fortzufahren, bis für jeden Testblock eine konkrete Ansteuerung gefunden worden ist.

Die Ergebnisse aller durchgeführten Experimente bestätigen die Effizienz des hier vorgestellten Ansatzes. Durch die Kombination von Akkumulatoren und Testregistern zusammen mit dem Konzept der Testbarkeitsmaße, einer variablen Ansteuerung und eigenen Transportpfaden für Konstanten zu Seiteneingängen von Modulen kommt ein Selbsttest i.a. mit deutlich geringerem Hardware-Mehraufwand aus als ein Selbsttest, der nicht diese Variationsbreite an Testblöcken zuläßt. Das neue Verfahren dürfte daher den herkömmlichen Verfahren zur Konfiguration eines Selbsttests, die nicht diese Vielseitigkeit aufweisen, bezüglich des Hardware-Aufwandes meist überlegen sein.

Das vorgestellte Verfahren arbeitet bei kleinen bis mittelgroßen Schaltungen relativ schnell. Mit guten Ergebnissen ist bereits nach einigen Minuten zu rechnen. Problematisch wird es, wenn eine Schaltung sehr viele potentielle Propagierungspfade aufweist, was die Zahl möglicher Testblockstrukturen und damit die Rechenzeit stark ansteigen läßt. Hier ist es sinnvoll, den Suchraum einzuschränken, z.B. durch den Verzicht auf Strukturzyklen wie in Experiment 3. Eine Zeitersparnis

brächte auch die Beschränkung der maximal erlaubten Pfadlänge zwischen Mustergeneratoren und Kompaktierern. Bei kurzen Pfadlängen erhielte man recht schnell eine, wenn auch teure, aber komplette Selbsttestkonfiguration. Durch schrittweise Erhöhung der erlaubten Pfadlänge erzielte man schnell kleine Verbesserungen. Ein solches Vorgehen eignete sich besonders zur Optimierung der Selbsttestkonfiguration bei vorgegebener Rechenzeit.

Der Einsatz des Verfahrens beschränkt sich auf das Operationswerk einer Schaltung. Für Steuerwerke ist es wegen des Mangels an Standardmodulen nicht geeignet. Wegen der vergleichsweise geringen Anzahl an Flipflops in einem Steuerwerk ist hier ein serieller Test mit Prüfpfad sinnvoll. Zur Mustergenerierung und Kompaktierung können die bereits für den Test des Operationswerkes konfigurierten Mustergeneratoren und Kompaktierer verwendet werden. Testregister sind dabei für die serielle Mustergenerierung wie für die serielle Kompaktierung einsetzbar, Akkumulatoren mit einer einzigen arithmetischen Verknüpfungseinheit nur für die Kompaktierung. Für die serielle Mustererzeugung sind komplexere Akkumulatoren notwendig [Strö98a], die bei Bedarf mit einer modifizierten Version des Algorithmus aus Abschnitt 3.2.2 konfiguriert werden könnten.

4 Optimierung der Testlänge bei Akkumulatoren

Im vorangegangenen Kapitel wurde eine Methode für den Selbsttest einer Schaltung mithilfe von Akkumulatoren beschrieben. Dabei wurde vorausgesetzt, daß sich alle Module der Schaltung leicht testen lassen, jeder der betrachteten Schaltungsfehler ist dann durch eine Vielzahl von Testmustern zu detektieren. In diesem Fall spielt die konkrete Wahl der vom Mustergenerator erzeugten Musterfolge nur eine untergeordnete Rolle. Solange die Folge nur genügend verschiedene Muster enthält, kann ein Modul normalerweise mit einer vergleichsweise kurzen Folge und damit in kurzer Zeit getestet werden. Wird ein Akkumulator als Mustergenerator eingesetzt, kann der Startwert und der konstante Eingabewert, von denen ja die generierte Musterfolge abhängt (s. Abschnitt 2.3.2), weitgehend beliebig gewählt werden. Lediglich bei der Konstanten sollte man darauf achten, daß die Zykluslänge maximal ist. Bei Bedarf simuliert man verschiedene Folgen und wählt die mit der kürzesten Testlänge aus.

Ganz anders sieht es aus, wenn einzelne Fehler eines Moduls nur mit wenigen Mustern zu testen sind. Solche „harten“ Fehler verursachen bei beliebig gewählten Folgen große Testlängen, z.T. wird die Testlänge so lang, daß an einen praktikablen Einsatz nicht mehr zu denken ist. Ist dies der Fall, muß man gezielt eine geeignete Musterfolge auswählen, will man auf weitere Hardware zur Vereinfachung des Tests verzichten.

In [LeGB95] werden dazu der Startwert und die Rückkopplung eines LRSR optimiert (vgl. Abschnitt 2.3.3.1). Das Verfahren ist sehr speicherintensiv, vor allem jedoch wegen der häufigen und schwierigen Berechnung diskreter Logarithmen so aufwendig, daß oft erst nach mehreren Stunden Rechenzeit eine Verbesserung gegenüber der einfachen Simulation verschiedener Musterfolgen erreicht werden kann. Für Akkumulatoren waren bisher keine Verfahren bekannt, die ohne den Einsatz zusätzlicher Hardware auskommen.

In Rahmen dieser Arbeit wurde ein solches Verfahren für Akkumulatoren entwickelt. Ausgehend von den Testmustern der betrachteten Fehler optimiert es jeweils den Startwert eines Akkumulators für zufällig gewählte konstante Eingabewerte. Dabei ist garantiert, daß alle modellierten Fehler innerhalb der gefundenen Testlänge detektiert werden. Das Verfahren erzielt bereits nach wenigen Minuten Testlängen, die deutlich kürzer sind als diejenigen, die in gleicher Rechenzeit durch Simulation zufälliger Musterfolgen gewonnen werden können.

Im folgenden wird das neue Verfahren vorgestellt. Dabei werden vorerst nur solche Testkonfigurationen betrachtet, bei denen ein einziger Mustergenerator die Testmuster für ein zu testendes Modul liefert. Solche Konfigurationen findet man häufig in digitalen Prozessoren, bei denen die Muster über Busse zu vielen anderen Teilschaltungen geleitet werden können. Später wird auch auf Module mit zwei separaten Mustergeneratoren eingegangen, eine typische Konfiguration, wenn der Selbsttest wie im vorigen Kapitel auf Register-Transfer-Ebene eingebaut wird.

Zunächst wird das Optimierungsproblem formal spezifiziert und verschiedene Optimierungsmöglichkeiten werden beleuchtet. Die beiden darauf folgenden Abschnitte beschreiben ein Verfahren, das den Startwert bei vorgegebener Konstante optimiert und dabei die auftretende Datenflut wirkungsvoll begrenzt. In einem weiteren Abschnitt werden die dabei gewonnenen Mechanismen zu einem automatischen Verfahren kombiniert. Darauf wird das Verfahren auf Module mit zwei separaten Mustergeneratoren erweitert. Den Abschluß bilden wieder experimentelle Ergebnisse.

4.1 Problemspezifikation

Die Suche nach der kürzesten Musterfolge läßt sich folgendermaßen spezifizieren. Sei $F = \{f_0, f_1, ..., f_{n-1}\}$ eine Menge von kombinatorischen Fehlern der zu testenden Schaltung. Jeder dieser Fehler sei durch mindestens ein Testmuster detektierbar. Eine Menge T_j, $0 \le j < n$, sei nun die Menge von Testmustern, die den Fehler f_j detektieren können. Ziel ist es, eine Musterfolge s(t), $t \ge 0$, des Akkumulators zu finden, so daß die Teilfolge s(0), s(1), ..., s(l-1) bei minimaler Länge l für jeden Fehler $f_j \in F$ mindestens ein Testmuster $s(t) \in T_j$ enthält.

Ein Akkumulator mit Addierer modulo 2^k erzeugt die Teilfolge s(0), [s(0) + v] mod 2^k, [s(0) + 2·v] mod 2^k, ..., [s(0) + (l-1)·v] mod 2^k. Die Folge hängt allein vom Startwert s(0) und vom konstanten Eingabewert v ab. Die Konstante v bestimmt offensichtlich die Reihenfolge der Muster innerhalb der Musterfolge, während s(0) lediglich den Beginn der Folge festlegt. Dabei ist es unerheblich, ob ein einfacher Addierer verwendet wird, Addierer mit rückgekoppeltem Überlaufbit oder die entsprechenden Subtrahierer (s. Abschnitt 2.3.2). Bild 4.1 verdeutlicht den Einfluß der beiden Größen auf die erforderliche Testlänge l an einem kleinen Beispiel mit drei Fehlern und einem Akkumulator mit einfacher 3-bit-Addition. Bild 4.1a zeigt die Testmuster, mit denen sich die drei Fehler f_0, f_1 und f_2 überprüfen lassen. In Bild 4.1b und c sind die kompletten Musterzyklen für zwei verschiedene Paare (s(0), v) gegeben. Neben den einzelnen Mustern stehen die Fehler, die durch das Muster getestet werden.

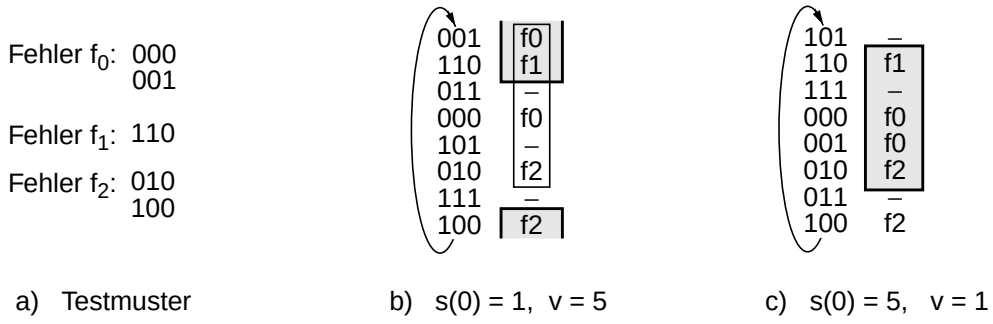


Bild 4.1: Einfluß von s(0) und v auf die Testlänge

In Musterzyklus b) wurde als Startwert s(0) = 1 (binär 001) gewählt. Man sieht, die resultierende Folge muß mindestens sechs Muster enthalten, um jeden Fehler wenigstens einmal zu testen (schmaler Kasten). Wählt man stattdessen als Startwert s(0) = 4 (binär 100), ist die Testlänge mit drei Mustern minimal (grauer Kasten). Bei vorgegebener Konstante v hängt die resultierende Testlänge offensichtlich stark vom Startwert ab. Für den Musterzyklus c) wurde eine andere Konstante verwendet, v = 1. Hier ist die kürzeste Testlänge $l = 5$ statt $l = 3$ für Zyklus b). Ist die Konstante einmal festgelegt, bestimmt sie die untere Grenze für die Testlänge.

Beide Größen, der Startwert und die Konstante, haben offensichtlich entscheidenden Einfluß auf die kürzestmögliche Testlänge. Sind beide Werte vorgegeben, so bleibt immer noch das Problem, die erforderliche Testlänge zu ermitteln. Ein Weg ist die Bestimmung der Fehlererfassung mittels Fehlersimulation. Der Aufwand dafür hängt aber wesentlich von der Länge der simulierten Musterfolge ab, die ja gerade bei harten Fehlern sehr groß werden kann. Der hier vorgestellte Ansatz beschreitet daher einen anderen Weg. Er berechnet die Anzahl der Schritte, die im Zyklus zwischen den Testmustern und einem Referenzmuster liegen, und bestimmt mithilfe dieser Distanzen die Testlänge. Wie die beiden nächsten Abschnitte zeigen, läßt sich dadurch die Rechenzeit um Größenordnungen verringern.

Aus Aufwandsgründen ist es meist nicht möglich, bei einem k-bit-Akkumulator alle 2^{2k} möglichen Kombinationen von Startwert und Konstante zu untersuchen. Stattdessen wird eine Größe festgehalten und die andere Größe für diesen Spezialfall optimiert. Dieses Problem ist weit weniger komplex. In der Tat ist es wesentlich effizienter, für eine vorgegebene Konstante v den Startwert zu berechnen als umgekehrt. Wenn nämlich nur die Startwerte wechseln, ändern sich die Distanzen der Testmuster zueinander nicht, was sich gut bei der Testlängenbestimmung ausnutzen läßt. Ein konstanter Startwert dagegen liefert für die Optimierung der Konstanten weit weniger nützliche Information. Hier wird deshalb folgende Vorgehensweise verfolgt:

- Wähle eine zufällige Konstante v für maximale Zykluslänge aus (vgl. Abschnitt 2.3.2)
- Bestimme den optimalen Startwert für diese Konstante
- Wiederhole die beiden Schritte, bis eine Zeitschranke überschritten wird

Eine Konstante für maximale Zykluslänge findet man i.a. nach höchstens drei Versuchen. Der erste Schritt stellt daher kein zeitliches Problem dar. Im folgenden Abschnitt wird daher nur das Verfahren zur Bestimmung des optimalen Startwerts bei vorgegebener Konstante vorgestellt.

4.2 Optimierung des Startwerts

Bei der Optimierung des Startwerts wird vorausgesetzt, daß die Konstante v und damit der Musterzyklus fest vorgegeben ist. Zuerst wird ein beliebiger Fehler, z.B. f_0, zum *Referenzfehler* erklärt. Aus seiner Testmustermenge T_0 wird dann ein Testmuster r als *Referenzmuster* ausgewählt. Für jedes der $|T_0|$ möglichen Referenzmuster wird nur das Teilproblem betrachtet, den optimalen Startwert einer Musterfolge zu finden, die dieses Referenzmuster sowie Testmuster für alle anderen Fehler umfaßt. Der Startwert der kürzesten Folge ist das gesuchte Optimum.

Damit ein Fehler f_j von einer Folge mit Referenzmuster r erfaßt wird, müssen bei der Startwertoptimierung nicht alle $|T_j|$ Testmuster dieses Fehlers berücksichtigt werden. Es genügt, die beiden Testmuster aus T_j zu kennen, die im Musterzyklus das Referenzmuster am engsten „umrahmen“. Sie werden kurz als *Vorgänger* und als *Nachfolger* bezeichnet. Da das Referenzmuster in jedem Fall innerhalb der gesuchten Musterfolge liegt, muß auch mindestens eines dieser beiden Testmuster Teil der Folge sein. Sind nun für alle Fehler f_1, ..., f_{n-1} die Vorgänger und Nachfolger bekannt, läßt sich der optimale Startwert mit einer einfachen Fenstertechnik bestimmen. Ein Fenster entspricht dabei einer Teilfolge, der Beginn des Fensters dem Startwert und die Fensterlänge der Testlänge.

Bild 4.2 zeigt ein Beispiel mit den vier Fehlern f_0, ..., f_3, dem Referenzmuster ‡ aus T_0 und den Vorgängern bzw. Nachfolgern #, ◊ und o aus den Testmustermengen T_1, T_2 und T_3. Der Musterzyklus

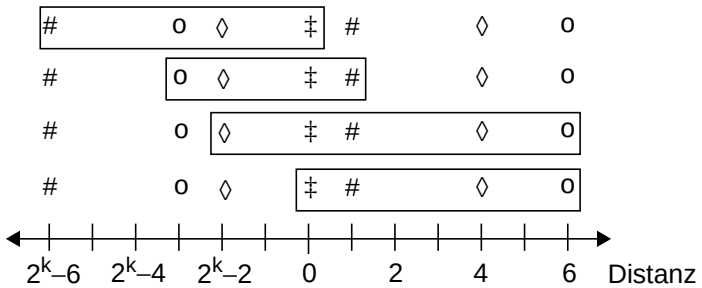


Bild 4.2: Beispiel für die Fenstertechnik

erstreckt sich in horizontaler Richtung, gezeigt ist nur ein kleiner Ausschnitt. Das erste Fenster am unteren Rand von Bild 4.2 beginnt mit dem Referenzmuster und hat die Länge $l = 7$. Man verschiebt nun den linken Rand Schritt für Schritt zum jeweils nächsten Vorgänger und verkürzt das Fenster am rechten Rand immer soweit, daß gerade noch jeder Fehler abgedeckt ist. Das kürzeste Fenster (hier das zweitoberste mit der Länge $l = 5$) ist die gesuchte Musterfolge, der Beginn dieses Fensters der optimale Startwert für das betrachtete Teilproblem.

Um den Vorgänger und Nachfolger bestimmen zu können, muß man in der Lage sein, die Distanz der Muster zum jeweiligen Referenzmuster zu berechnen. Die *Distanz* eines Musters y bezüglich eines Musters x, $\delta(x, y)$, ist die Anzahl der Taktschritte im Musterzyklus zwischen dem Auftreten von x und y. Trete x zum Zeitpunkt i auf, $x = s(i)$, dann gilt $y = s(i+\delta)$ für ein bestimmtes $\delta \in \{0, 1, ..., 2^{k-1}\}$. Für einen einfachen k-bit Addierer gilt entsprechend Abschnitt 2.3.2:

$$s(i+\delta) = [s(i) + \delta \cdot v] \bmod 2^k$$

Löst man nach δ auf, erhält man:

$$[\delta \cdot v] \bmod 2^k = [s(i+\delta) - s(i)] \bmod 2^k$$

$$\delta = [(s(i+\delta) - s(i)) \cdot v^{-1}] \bmod 2^k$$

$$\delta = [(y - x) \cdot v^{-1}] \bmod 2^k$$

Dabei ist v^{-1} die Inverse zu v, und es gilt $(v \cdot v^{-1}) \bmod 2^k = 1$. Es gilt nun folgender Satz:

Satz: $ggT(a, b) = 1 \Rightarrow$ Es existiert eine eindeutige Inverse a^{-1}, $0<a^{-1}<b$, mit $(a \cdot a^{-1}) \bmod b = 1$

Beweis: Der größte gemeinsame Teiler ggT(a, b) läßt sich immer als Linearkombination von a und b darstellen, d.h. es existieren zwei ganze Zahlen r und s, so daß gilt: $ggT(a, b) = r \cdot a + s \cdot b$ [Kobl87, S.13]. Mit $ggT(a, b) = 1$ erhält man:

$$r \cdot a + s \cdot b = 1$$

$$(r \cdot a) \bmod b = 1$$

$$\Rightarrow \quad r = a^{-1}$$

Nun ist $E(r) = (r \cdot a) \bmod b$ eine Funktion, die entsprechend dem Beweis des Satzes aus Abschnitt 3.2.5.3 wegen $ggT(a, b) = 1$ periodisch alle Werte zwischen 0 und b-1 annehmen kann, d.h. es existiert genau ein r, $0 < r < b$, mit $(r \cdot a) \bmod b = 1$. Die Inverse $r = a^{-1}$ ist somit eindeutig.

❑

Aufgrund des Satzes existiert eine eindeutige Inverse v^{-1}, wenn $ggT(v, 2^k) = 1$ gilt, also genau dann, wenn der Musterzyklus maximal ist. Nun werden aber ohnehin nur Konstanten v betrachtet, bei

denen ein maximaler Musterzyklus entsteht. Die Distanz δ läßt sich also bei Kenntnis der Inversen v^{-1} berechnen. Die Inverse erhält man mithilfe des Euklidischen Algorithmus [Kobl87, S.12 f.].

Die Distanzen lassen sich in ähnlicher Weise auch für Addierer mit rückgekoppeltem Überlaufbit und für die verschiedenen Subtrahierertypen berechnen, sofern die Muster s(i) = x und s(i+δ) = y im Musterzyklus enthalten sind:

Einerkomplement-Addierer, $0<s(t)<2^k$, $ggT(v, 2^k-1)=1$, $(v\cdot v^{-1}) \bmod (2^k-1)=1$:

$$y = [x - 1 + \delta\cdot v] \bmod (2^k-1) + 1$$

$$\Rightarrow \quad \delta = [(y - x)\cdot v^{-1}] \bmod (2^k-1)$$

Addierer mit gesp. Überlaufbit, $0 \le s(t)<2^k$, $s(t)\neq v$, $ggT(v, 2^k-1)=1$, $(v\cdot v^{-1}) \bmod (2^k-1)=1$:
Seien s'(t) die Zustände des Einerkomplement-Addierers. Dann gilt:

$$s'(t) = \begin{cases} [s(t)+1] \bmod 2^k; & s(t) \in \{0, ..., v-1\} \\ s(t); & s(t) \in \{v+1, ..., 2^k-1\} \end{cases}$$

$$\delta = [(y' - x')\cdot v^{-1}] \bmod (2^k-1)$$

Subtrahierer, $0 \le s(t)<2^k$, $ggT(v, 2^k)=1$, $(v\cdot v^{-1}) \bmod 2^k=1$:

$$y = [x - \delta\cdot v] \bmod 2^k$$

$$\Rightarrow \quad \delta = [(x - y)\cdot v^{-1}] \bmod 2^k$$

Subtrahierer modulo 2^k-1, $0 \le s(t)<2^k-1$, $ggT(v, 2^k-1)=1$, $(v\cdot v^{-1}) \bmod (2^k-1)=1$:

$$y = [x - \delta\cdot v] \bmod (2^k-1)$$

$$\Rightarrow \quad \delta = [(x - y)\cdot v^{-1}] \bmod (2^k-1)$$

Subtr. mit gesp. Unterlaufbit, $0 \le s(t)<2^k$, $s(t)\neq 2^k-v-1$, $ggT(v, 2^k-1)=1$, $(v\cdot v^{-1}) \bmod (2^k-1)=1$:
Seien s'(t) die Zustände des Subtrahierers modulo 2^k-1. Dann gilt:

$$s'(t) = \begin{cases} s(t); & s(t) \in \{0, ..., 2^k-v-2\} \\ [s(t)-1] \bmod 2^k; & s(t) \in \{2^k-v, ..., 2^k-1\} \end{cases}$$

$$\Rightarrow \quad \delta = [(x' - y')\cdot v^{-1}] \bmod (2^k-1)$$

Die obigen Distanzformeln lassen sich alle auf folgende allgemeine Form bringen:

$$\delta\cdot v = [a - b] \bmod N$$

Tatsächlich läßt sich die Distanz d auch dann berechnen, wenn ggT(v, N) > 1 gilt. Man dividiert dazu beide Seiten der Gleichung durch ggT(v, N) und erhält:

$$\delta \cdot v' = \frac{a - b}{ggT(v, N)} \bmod N'$$

Da nun ggT(v', N') = 1 gilt, existiert wieder eine eindeutige Inverse zu v' bezüglich N', und die Distanz δ läßt sich analog zu den obigen Gleichungen bestimmen. Wegen ggT(v, N) > 1 ist der Musterzyklus nicht mehr maximal. Das macht sich darin bemerkbar, daß nicht mehr alle Muster im Zyklus enthalten sind. In diesem Fall ist (a – b) kein Vielfaches von ggT(v, N), und es existiert keine ganzahlige Lösung für δ.

Mit den Formeln zur Distanzbestimmung läßt sich der Vorgänger und Nachfolger eines Fehlers bezüglich des Referenzmusters r einfach aus den Distanzen δ(m, r) bzw. δ(r, m) für jedes Testmuster m ermitteln. Testmuster, die nicht im Musterzyklus vorkommen, werden ignoriert. Bei den hier betrachteten maximalen Zyklen ist das jedoch höchstens ein Testmuster, das gegebenenfalls als

```
Prozedur Startwertoptimierung(v)        /* Liefert den optimalen Startwert s_opt und die kürzeste
                                           Länge l_opt bei gegebener Konstante v */
l_opt := 2^k + 1;                       /* Optimale Länge Initialisieren */
D_vor := D_nach := {};                  /* Mengen der Vorgänger- bzw. Nachfolgerdistanzen */
Für alle Referenzmuster r ∈ T_0:
    Für jeden Fehler f_j außer dem Referenzfehler:
        δ^j_vor := 2^k + 1; δ^j_nach := 2^k + 1;   /* Vorgänger- und Nachfolgerdistanzen
                                                      initialisieren */
    Für alle Testmuster m ∈ ⋃_{i=1}^{n-1} T_i :    /* Bestimme alle Vorgänger und Nachfol-
                                                      gerdistanzen */
        Für jeden Fehler f_j, der mit m detektiert werden kann:
            Falls δ(m, r) < δ^j_vor ,   /* Die Distanzen δ werden bezüglich v berechnet */
                δ^j_vor := δ(m, r);
            Falls δ(r, m) < δ^j_nach ,
                δ^j_nach := δ(r, m);
        D_vor := D_vor ∪ {δ^j_vor}; D_nach := D_nach ∪ {δ^j_nach};
    (s, l) := Fenstertechnik(D_vor, D_nach);   /* Startwert s und kürzeste Länge l bestimmen */
    Falls l_opt > l,
        l_opt := l; s_opt := s;             /* Optimale Länge l_opt und optimalen
                                               Startwert s_opt aktualisieren */
return (s_opt, l_opt);
```

Bild 4.3: Hauptprozedur der Startwertoptimierung

Startwert des Akkumulators berücksichtigt werden kann. Durch fortwährendes Aktualisieren der Muster mit den zwischenzeitlich geringsten Abständen bleiben am Schluß der tatsächliche Vorgänger und Nachfolger übrig. Der gesamten Algorithmus zur Optimierung des Startwerts bei vorgegebener Konstante besteht aus einer Hauptprozedur (Bild 4.3) und der Fenstertechnik (Bild 4.4).

```
Prozedur Fenstertechnik(D_vor, D_nach)          /* Liefert das Anfangsmusters m_opt und
                                                   die Länge l_opt des kürzesten Fensters */
   M:= {};                                      /* Menge abgedeckter Vorgänger */
   I_vor := Liste aller Fehlerindizes i, 0<i<n;
   I_nach := I_vor;
   l_opt := 1 + Maximum von allen δ^i_nach ∈ D_nach;   /* Erste Fensterlänge initialisieren */
   m_opt := Referenzmuster r;                   /* Erster Startwert ist Referenzmuster */
   Sortiere I_vor und I_nach so, daß Indizes mit kleineren Distanzen δ^i_vor ∈ D_vor bzw. δ^i_nach ∈ D_nach
   vor solchen mit größeren Distanzen stehen;
Sprungmarke:                                    /* Optimiere Fenster */
   Falls Liste I_vor noch Elemente enthält,
      a := erstes Element der Liste I_vor; Entferne erstes Element aus I_vor;
      M := M ∪ m_a;                             /* m_a ist Vorgänger mit Distanz δ^i_vor */
      Solange I_nach noch Elemente enthält:
         b := letztes Element der Liste I_nach;
         Falls m_b ∈ M,                         /* m_b ist Nachfolger mit Distanz δ^b_nach */
            Entferne letztes Element aus I_nach;
      Falls I_nach = {},
         δ^b_nach := 0;                         /* Referenzmuster ist letzter Nachfolger */
      Falls l_opt > δ^a_vor + δ^b_nach + 1,     /* Optimales Fenster aktualisieren */
         l_opt := δ^a_vor + δ^b_nach + 1;
         m_opt := m_a;
         gehe zur Sprungmarke;
   return (s_opt, l_opt);
```

Bild 4.4: Algorithmus der Fenstertechnik

Die Hauptprozedur berechnet für jedes Referenzmuster zunächst die Vorgänger- und Nachfolgerdistanzen, optimiert mithilfe dieser Distanzen und der Fenstertechnik den Startwert und die Testlänge für das jeweilige Referenzmuster und aktualisiert gegebenenfalls den Startwert und die Länge der gesuchten besten Musterfolge. Die meiste Rechenzeit benötigt die Hauptprozedur zur Bestimmung der Vorgänger und Nachfolger, da für jeden Fehler die Distanzen aller Testmuster zum Refe-

renzmuster berechnet werden müssen. Üblicherweise existieren dabei selbst für die härtesten Fehler einer Schaltung noch vergleichsweise viele Testmuster. Die Fenstertechnik ist demgegenüber sehr schnell, da jetzt nur noch zwei Muster pro Fehler berücksichtigt zu werden brauchen – der Vorgänger und der Nachfolger. Die Hauptprozedur läßt sich jedoch beschleunigen, indem man Testmuster, die dieselben Fehler detektieren können, zu Gruppen von disjunkten *Testwürfeln* zusammenfaßt. Ein Testwürfel ist dabei ein Muster, das unspezifizierte Bitstellen, *don't cares*, beinhalten kann. Die Testmuster werden nun gruppenweise abgearbeitet, und zwar in einer Reihenfolge, in der man möglichst schnell die Vorgänger und Nachfolger möglichst vieler Fehler erhält. Jedem Testwürfel wird ein eigener Vorgänger und Nachfolger zugewiesen. Der Vorgänger und Nachfolger eines Fehlers werden aus den Vorgängern und Nachfolgern der Testwürfel bestimmt, die den Fehler testen können. Sind nun ein Vorgänger und Nachfolger eines Fehlers f bekannt, kann die Bearbeitung einer Testwürfelgruppe abgebrochen werden, sobald feststeht, daß der Vorgänger und der Nachfolger der Gruppe näher am Referenzmuster liegen werden als der Vorgänger bzw. der Nachfolger des Fehlers f. Die der Testgruppe zugeordneten Fehler werden dann von der resultierenden Musterfolge garantiert erfaßt, andernfalls würde auch der Fehler f nicht detektiert. Die Fehler der Testgruppe brauchen dann auch bei der Fenstertechnik nicht weiter berücksichtigt zu werden. Sind hingegen sowohl der Vorgänger wie auch der Nachfolger des Fehlers f weiter vom Referenzmuster entfernt als die bisher kürzeste Musterfolge lang ist, kann die resultierende Musterfolge keine Verbesserung bringen. In diesem Fall darf sogar die komplette Optimierung des Startwerts für das aktuelle Referenzmuster abgebrochen werden.

Obwohl mit den beschriebenen Methoden der Algorithmus zur Bestimmung eines optimalen Startwerts deutlich beschleunigt werden kann, ist der Rechenaufwand i.a. immer noch zu groß, um eine kurze Musterfolge in akzeptabler Zeit zu finden. Im folgenden Abschnitt werden daher Heuristiken vorgestellt, die bereits nach wenigen Minuten Rechenzeit gute Lösungen liefern.

4.3 Heuristiken zur Verringerung des Rechenaufwandes

Der Rechenaufwand zur Bestimmung des optimalen Startwerts hängt überwiegend von der Anzahl der Testmuster ab, deren Distanz zum Referenzmuster bestimmt werden muß. In der Regel ist die Zahl der Testmuster viel zu groß, wenn alle modellierten Schaltungsfehler bei der Distanzberechnung hinzugezogen werden. Besonders die leicht testbaren Fehler tragen dann oft eine riesige Menge an Testmustern bei. Nun sind es aber gerade die harten Fehler, die für die Länge der benötigten Musterfolge verantwortlich sind. Wegen der vergleichsweise geringen Zahl an Testmustern

ist die Chance, einen harten Fehler durch eine vorgegebene Musterfolge aufspüren zu können, geringer als bei leichten Fehlern. Bei Musterfolgen, die nicht alle Fehler detektieren können, sind die nicht getesteten Fehler daher vorwiegend harte Fehler. Zur Reduzierung des Rechenaufwands ist es somit naheliegend, bei der Optimierung nur harte Fehler zu betrachten. Ist die Menge der harten Fehler angemessen gewählt, ist die optimierte Musterfolge in der Regel lang genug, um auch alle übrigen, leichten Fehler mitzutesten. Sichergestellt wird das mit einer abschließenden Fehlersimulation der gefundenen Musterfolge.

Nun sind aber gerade zu Beginn des Verfahrens die gefundenen Testlängen i.a. noch weit vom tatsächlichen Optimum entfernt, und die Fehlersimulation für solche großen Testlängen ist relativ zeitaufwendig. Andererseits ist die Wahrscheinlichkeit, alle leichten Fehler mitzutesten, bei großen Testlängen sehr hoch und eine Fehlersimulation im Vergleich zur Rechenzeit nur wenig nützlich. Ein sinnvoller Kompromiß ist, auf Fehlersimulationen zu verzichten, solange eine optimierte Musterfolge eine Länge Λ überschreitet, bei der es sehr wahrscheinlich ist, auch alle leichten Fehler zu detektieren. Die Frage ist dann, wo man die Grenze zwischen leichten und harten Fehlern ziehen muß, damit auch alle leichten Fehler mit einer geforderten Mindestwahrscheinlichkeit P_G von einer Musterfolge der Länge Λ entdeckt werden sollen.

Da ein Testmuster oft mehrere verschiedene Fehler testen kann, sind die Fehlererfassungswahrscheinlichkeiten miteinander korreliert, was die ganze Betrachtung wesentlich erschwert. Wie schon oben erwähnt, hängt die erforderliche Testlänge überwiegend von dem am schwersten zu detektierenden Fehler einer zu untersuchenden Fehlermenge ab. Je stärker die Korrelationen sind, um so stärker wird die Abhängigkeit von diesem Fehler. Für eine gute Abschätzung der Grenze zwischen leichten und harten Fehlern genügt es daher i.a., die Fehler isoliert zu betrachten. Sei also f_j ein beliebiger Fehler mit der Testmustermenge T_j. Die Wahrscheinlichkeit p_j, daß ein beliebiges Muster einer Zufallsfolge gleichverteilter Muster den Fehler f_j detektiert, ist:

$$p_j = \frac{|T_j|}{2^k}$$

Die Wahrscheinlichkeit $P(\Lambda, f_j)$, den Fehler mit einer Zufallsfolge der Länge Λ zu erfassen, ist:

$$P(\Lambda, f_j) = 1 - (1 - p_j)^{\Lambda}$$

Nach p_j aufgelöst erhält man:

$$p_j = 1 - \sqrt[\Lambda]{1 - P(\Lambda, f_j)}$$

Als Abschätzung für die Grenze werden nun alle Fehler f_j mit $P(\Lambda, f_j) \leq P_G$ als *hart bezüglich der Länge* Λ und der „Grenzwahrscheinlichkeit“ P_G bezeichnet. Damit gilt für harte Fehler:

$$p_j \leq 1 - \sqrt[\Lambda]{1 - P_G} \tag{4.1}$$

Je größer P_G gewählt wird, um so mehr Fehler werden als hart eingestuft, und die als leicht klassifizierten Fehler können mit immer mehr Testmustern detektiert werden. Damit steigt mit P_G auch die Wahrscheinlichkeit, daß die bei der Optimierung unberücksichtigt gebliebenen leichten Fehler wie gefordert ebenfalls von der optimierten Musterfolge erfaßt werden.

Bisher wurde immer vorausgesetzt, daß alle Testmuster und damit die Anzahl der Testmuster pro Fehler bekannt sind. Tatsächlich müssen die Testmuster für ein gegebenes zu testendes Modul auch erst berechnet werden. Die Bestimmung eines Testmusters pro Fehler ist mit modernen Verfahren in der Regel recht schnell (vgl. Abschnitt 2.2.3), die Berechnung aller Testmuster hingegen meist zu langwierig. Einfacher ist es, eine kurze Zufallsfolge der Länge λ zu simulieren. Dabei nicht erkannte Fehler werden als hart betrachtet, die übrigen als leicht. Auf diese Weise kann man die Testmusterbestimmung auf die so klassifizierten harten Fehler beschränken. Es gilt nun, die Simulationslänge λ so festzulegen, daß möglichst alle entsprechend der Gleichung (4.1) als leicht einzustufenden Fehler bei der Simulation entdeckt werden und alle harten Fehler nicht. Dabei wird gefordert, daß die harten Fehler mit einer großen Mindestwahrscheinlichkeit $P_{korrekt}$ richtig eingestuft werden. Betrachtet man alle Fehler wieder isoliert und von den gemäß Gleichung (4.1) harten Fehlern nur den leichtesten Fehler f_h, dann gilt: $P_{korrekt} = (1 - p_h)^{\lambda} = \sqrt[\Lambda]{1 - P_G}^{\lambda}$. Damit erhält man $P_{korrekt} = (1 - P_G)^{\lambda/\Lambda}$ und nach λ aufgelöst:

$$\lambda = \frac{\log(P_{korrekt})}{\log(1 - P_G)} \cdot \Lambda \tag{4.2}$$

Mit dieser Gleichung wird die Simulationslänge λ zur Bestimmung der harten Fehler in einen statistischen Zusammenhang mit der Grenzlänge Λ gebracht, die entscheidet, ob der Rechenaufwand zur Kontrolle einer Musterfolge durch Fehlersimulation gerechtfertigt ist oder nicht. Die Wahrscheinlichkeiten $P_{korrekt}$ und P_G dienen zur Abwägung zwischen der Rechenzeit für eine Kontrollsimulation und dem Risiko einer Verringerung der Fehlererfassung unabhängig von dem zu testenden Modul. Einen guten Kompromiß erhält man mit $P_{korrekt} = P_G = 99\ \%$ und damit $\lambda = 0{,}00218 \cdot \Lambda$. Entscheidenden Einfluß auf die Rechenzeit hat auch die Wahl der Länge Λ. In der Tat wird Λ im Wechselspiel mit der Startwertoptimierung automatisch an das zu testende Modul angepaßt. Diese Anpassung ist Gegenstand des folgenden Abschnitts.

4.4 Komplettes Optimierungsverfahren

In diesem Abschnitt werden die bisher entwickelten Ideen zu einem Verfahren kombiniert, das automatisch den Startwert s(0) und die Konstante v für eine möglichst kurze Musterfolge bestimmt. Wie im letzten Abschnitt erläutert, steuert der Parameter Λ die Anzahl der bei der Startwertoptimierung betrachteten harten Fehler. Λ dient dabei als Abschätzung für die endgültige, optimierte Testlänge. Je genauer diese Abschätzung ist, um so effektiver arbeitet die Startwertoptimierung. Da die optimierte Testlänge apriori nicht bekannt ist, wird Λ Schritt für Schritt an das zu testende Modul angepaßt.

Das Optimierungsverfahren beginnt dazu mit einem großen Wert $\Lambda = \Lambda_{max}$. Die Simulationslänge λ ist dann ebenfalls relativ groß, die Zahl der bei der Simulation nicht entdeckten harten Fehler demnach klein. Nach der Bestimmung der Testmuster für die so gewonnenen harten Fehler startet die Suche nach einer kurzen Musterfolge. Dazu wird nacheinander für zufällige Konstanten v, die einen maximalen Zyklus erzeugen, der Startwert optimiert. Die Startwertoptimierung ist wegen der geringen Anzahl von harten Fehlern und damit Testmustern noch recht schnell, es können daher viele Konstanten in kurzer Zeit untersucht werden. Jedesmal, wenn eine neue, kürzere Folge gefunden worden ist und ihre Länge den Wert Λ unterschreitet, wird per Fehlersimulation überprüft, ob auch alle leichten Fehler detektiert werden und die Musterfolge wirklich zu gebrauchen ist.

Zu Beginn sind die optimierten Testlängen wegen der wenigen harten Fehler jedoch meist zu kurz, um alle Fehler erfassen zu können. Verlängert man eine solche Musterfolge bis alle Fehler detektiert werden, zeigt sich in aller Regel, daß die jetzige Testlänge keine Verbesserung mehr darstellt. Das Verfahren arbeitet in dieser Phase zwar schnell, aber ineffizient. Deshalb wird nach mehreren fehlgeschlagenen Kontrollsimulationen – als günstig haben sich drei Fehlschläge hintereinander erwiesen – die Optimierung unterbrochen und die Menge der harten Fehler erweitert, indem Λ um einen bestimmten Betrag verringert wird. Als praktikabler Ansatz hat sich dabei die Beschränkung der Werte Λ auf Werte der Menge $\{10^x, 2\cdot10^x, 5\cdot10^x\}$ herausgestellt, wobei x eine natürliche Zahl ist. Die erneute Suche braucht durch die Verringerung von Λ zwar mehr Zeit pro Konstante, liefert dafür aber häufiger Verbesserungen. Auf diese Weise wird Λ recht schnell an die endgültige verbesserte Testlänge angepaßt. Das Verfahren endet, wenn eine vorgegebene Zeitschranke überschritten wird. Bild 4.5 zeigt eine Übersicht des gesamten Verfahrens.

Das Verfahren liefert die beste gefundene Kombination von Konstante v_{opt} und Startwert s_{opt} zusammen mit der Testlänge l_{opt}, bei der alle Fehler der Menge F erfaßt werden. Bei kleiner Zeitschranke sollte bei der Startwertoptimierung gemäß Abschnitt 4.2 die Anzahl der zu untersuchen-

```
v_opt := 1; s_opt := 0;                    /* Initiale Werte für Konstante und Startwert */
λ := λ_max;                                /* Initiale Simulationslänge */
l_opt := Λ_max; Λ := Λ_max;                /* Maximale Testlänge */

Solange Zeitschranke nicht überschritten:
    Bestimme alle harten Fehler bezüglich Λ durch Simulation der Länge λ;
    Bestimme alle Testmuster für die harten Fehler;
    fehlschläge := 0;

    Solange fehlschläge < 3 und Zeitschranke nicht überschritten:
        Wähle eine Konstante v, die maximale Zykluslänge garantiert;
        Bestimme den besten Startwert s_v und die Testlänge l_v unter
        alleiniger Berücksichtigung der harten Fehler

        Falls (l_v < l) und (l_v < Λ),
            Führe Kontrollsimulation mit Startwert s_v und l_v Mustern durch;

            Falls alle Fehler f∈ F detektiert wurden,
                v_opt := v; s_opt := s_v;          /* kürzere Musterfolge gefunden */
                l := l_v;
                fehlschläge := 0;

            sonst:
                fehlschläge := fehlschläge + 1;

    Verringere Λ und λ;

Ende
```

Bild 4.5: Algorithmus für komplettes Optimierungsverfahren

den Referenzmuster pro Referenzfehler beschränkt werden. Tatsächlich wird selbst bei nur je einem einzigen Referenzmuster der Wegfall der übrigen durch die nun größere Anzahl untersuchter Konstanten mehr als kompensiert.

Obwohl bisher nur Akkumulatoren als Mustergeneratoren betrachtet worden sind, läßt sich das Verfahren ebenso auf Testregister anwenden. Statt einer Konstanten v werden zufällige Rückkopplungen gewählt, die einen maximalen Zyklus garantieren. Solche Rückkopplungen sind z.B. in [PeWe72] bis zu einer Registerbreite von 34 bit tabelliert. Die Distanzen lassen sich wie in [LeGB95] mit diskreten Logarithmen berechnen, die allerdings rechnerisch weit schwieriger zu handhaben sind als die Distanzformeln bei den Akkumulatoren.

4.5 Module mit zwei separaten Mustergeneratoren

Das Optimierungsverfahren beruht darauf, daß ein zu testendes Modul von einem einzigen Mustergenerator gespeist werden kann. Auf Register-Transfer-Ebene werden Mustergeneratoren jedoch meist so in eine Schaltung eingebaut, daß jeder Moduleingang einer Funktionseinheit an einen eigenen Mustergenerator angeschlossen ist. Ein Beispiel ist das Verfahren aus Kapitel 3. Prinzipiell kann das Optimierungsverfahren auch auf Module mit zwei Eingängen erweitert werden. Darunter fallen z.B. typische Module wie arithmetische Einheiten, ALUs, Komparatoren und logische Operatoren.

Um das Optimierungsverfahren auf Module mit zwei Eingängen anwenden zu können, denkt man sich die beiden Muster an den Eingängen zu einem einzigen Muster doppelter Breite zusammengefaßt, das an einem Modul mit nur einem Eingang anliegt. Zwei k-bit-Muster x_1 und x_0 bilden somit zusammen das 2k-bit-Muster $X = x_1 \cdot 2^k + x_0$ (Bild 4.6).

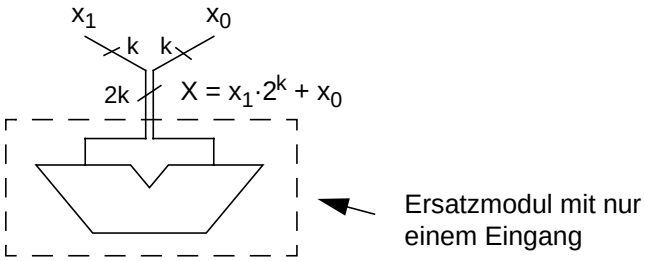


Bild 4.6: Zusammenfassung zweier Muster zu einem einzigen Muster

Der Musterzyklus der zusammengesetzten Musterfolge sollte maximal lang sein. Die Mustergeneratoren für die Muster x_1 bzw. x_0 erzeugen selbst zyklische Musterfolgen mit den Perioden π_1 bzw. π_0. Die Periode Π der zusammengesetzten Musterfolge ist offensichtlich das kleinste gemeinsame Vielfache der Perioden π_1 und π_0, es gilt also $\Pi = kgV(\pi_1, \pi_0)$ und somit:

$$\Pi = \frac{\pi_1 \cdot \pi_0}{ggT(\pi_1, \pi_0)} \tag{4.3}$$

Das Produkt $\pi_1 \cdot \pi_0$ wird maximal für $\pi_1 = \pi_0 = 2^k$. Allerdings ist $ggT(2^k, 2^k) = 2^k$ ebenfalls maximal, und man erhält nur $\Pi = 2^k$. Das nächstgrößte Produkt ist $2^k \cdot (2^k - 1)$, und wegen $ggT(2^k, 2^k - 1) = 1$ erhält man die maximale Periode $\Pi_{max} = 2^k \cdot (2^k - 1)$. Für eine zusammengesetzte Folge mit maximaler Zykluslänge muß also einer der Mustergeneratoren eine Folge mit Periode 2^k erzeugen, der andere eine Folge mit Periode $2^k - 1$. Nun können sowohl Akkumulatoren als auch LRSRs beide Zykluslängen generieren. Für die Zykluslänge $\pi = 2^k$ lassen sich Akkumulatoren mit einfachem

Addierer bzw. Subtrahierer sowie vollständige LRSRs (vgl. Abschnitt 2.3.1) verwenden, für Zykluslängen $\pi = 2^k-1$ die Addierer bzw. Subtrahierer mit Rückkopplung des Überlauf- bzw. Unterlaufbits oder ein normales LRSR.

Bei der Startwertoptimierung muß man lediglich die Distanz zwischen zwei Mustern berechnen können, die Art des Mustergenerators ist im Prinzip egal. Die Distanz $\Delta(X, Y)$ zwischen zwei zusammengesetzten Mustern $X = x_1 \cdot 2^k + x_0$ und $Y = y_1 \cdot 2^k + y_0$ erhält man wie folgt:

Seien $\delta_1 = \delta(x_1, y_1)$ und $\delta_0 = \delta(x_0, y_0)$ die Distanzen zwischen den Mustern x_1 und y_1 bzw. zwischen x_0 und y_0 in den Musterfolgen der beiden Mustergeneratoren. Für die Distanz Δ zwischen X und Y muß dann gelten: $\Delta = \delta_1 + n \cdot \pi_1$ und gleichzeitig $\Delta = \delta_0 + m \cdot \pi_0$, wobei n und m natürliche Zahlen sind. Es gilt also:

$$\begin{aligned} & \delta_1 + n\cdot\pi_1 = \delta_0 + m\cdot\pi_0 \\ \Leftrightarrow \quad & n\cdot\pi_1 - m\cdot\pi_0 = \delta_0 - \delta_1 \\ \Rightarrow \quad & [n\cdot\pi_1] \bmod \pi_0 = [\delta_0 - \delta_1] \bmod \pi_0 \\ \Leftrightarrow \quad & n = [(\delta_0 - \delta_1)\cdot\pi_1^{-1}] \bmod \pi_0 \end{aligned}$$

Dabei ist π_1^{-1} wieder die Inverse zu π_1 mit $(\pi_1^{-1} \cdot \pi_1) \bmod \pi_0 = 1$. Eine eindeutige Inverse existiert wiederum genau dann, wenn gilt: $ggT(\pi_1, \pi_0) = 1$. Das ist bei maximaler Periode $\Pi = \Pi_{max}$ der zusammengesetzten Musterfolge der Fall. Die Inverse läßt sich wie bei der Distanzberechnung bei Akkumulatoren mit dem Euklidischen Algorithmus berechnen. Damit erhält man als Distanz Δ:

$$\Delta = \delta_1 + ([(\delta_0 - \delta_1)\cdot\pi_1^{-1}] \bmod \pi_0)\cdot\pi_1 \qquad (4.4)$$

Wie schon in Abschnitt 4.2 erläutert, kann der Wert n auch berechnet werden, wenn $ggT(\pi_1, \pi_0) > 1$ gilt und der Musterzyklus nicht maximal ist. Man muß dann darauf achten, daß nicht mehr für alle Paare (δ_0, δ_1) eine ganzzahlige Lösung für n und damit eine sinnvolle Lösung für Δ existiert.

Die Distanz zwischen zwei zusammengesetzten Mustern X und Y kann auch bestimmt werden, wenn die Typen der beiden Mustergeneratoren verschieden sind. Wichtig ist nur, das die Distanzberechnungsformel für jeden Mustergenerator bekannt ist. Tatsächlich dürfen sogar die Bitbreiten der Mustergeneratoren verschieden sein. Die Gleichungen (4.3) und (4.4) gelten dann immer noch, und auch bei unterschiedlichen Bitbreiten k und h gilt die Voraussetzung $ggT(\pi_1, \pi_0) = ggT(2^k, 2^h-1) = 1$ und ebenso $\Pi_{max} = 2^k \cdot (2^h-1)$. Das Optimierungsverfahren kann daher statt auf einzelne Module auch auf komplexere Teilschaltungen, z.B. die Testblockkerne aus Kapitel 3 angewendet werden. Eventuell müssen mehrere LRSRs bzw. mehrere Akkumulatoren mithilfe zusätzlicher Lei-

tungen zu einem großen LRSR bzw. Akkumulator verknüpft werden. Bei LRSRs ist das prinzipiell immer möglich, bei Akkumulatoren in der Regel nur, wenn sie alle primitiv sind, also neben dem Akkuregister kein weiteres Register in der Rückkopplungsschleife enthalten ist. Außerdem müssen die Verknüpfungseinheiten vom selben Typ sein.

Prinzipiell läßt sich das Verfahren auch bei mehr als zwei Mustergeneratoren einsetzen. In diesem Fall erhält man für jedes Paar von Mustergeneratoren eine Gleichung analog zu Gleichung (4.4). Zur Bestimmung der Distanz Δ muß dann das ganze Gleichungssystem gelöst werden. Allerdings nimmt die Länge des maximalen Zyklus im Verhältnis zur Anzahl aller denkbaren Muster immer mehr ab, je mehr Mustergeneratoren betrachtet werden. Damit können immer mehr Testmuster nicht mehr erzeugt werden, und die Chance sinkt, eine kurze Musterfolge zu finden.

4.6 Experimentelle Ergebnisse

Das komplette Verfahren wurde in der Programmiersprache C implementiert und anhand mehrerer Experimente bewertet. Zur Bestimmung der Testmuster und zur Fehlersimulation wurde das System ATALANTA [LeHa91] verwendet und zur Kopplung mit dem vorgestellten Optimierungsverfahren entsprechend modifiziert. Als Testschaltungen wurden zunächst die kombinatorischen ISCAS'85-Schaltungen [BrFu85] untersucht. Die meisten dieser Schaltungen waren schon mit kurzen Zufallsfolgen vollständig zu testen. Für das hier vorgestellte Verfahren waren sie hingegen nicht geeignet, da selbst die härtesten Fehler noch mehrere Zehnmillionen Testmuster besaßen. Stattdessen wurde auf die Berkeley-Schaltungen [BHMS84] zurückgegriffen, die dazu als mehrstufige Schaltnetze synthetisiert wurden. Tabelle 4.1 zeigt die Charakteristika der untersuchten Schaltungen auf Gatterebene. Die Spalte *Tiefe* gibt dabei die Anzahl der Gatter entlang des längsten Pfades zwischen einem Schaltungseingang und einem Schaltungsausgang an. Die Spalte Fehler enthält die Anzahl der modellierten Fehler, das System ATALANTA betrachtet dabei nur Haftfehler. In der letzten Spalte ist die Zahl der Fehler angegeben, die durch ATALANTA als *redundant* klassifiziert wurden. Dies sind entweder echt redundante Fehler, für die aufgrund der Schaltungsstruktur kein Testmuster existiert, oder Fehler, für die ATALANTA kein Testmuster finden konnte.

Alle Experimente wurden auf einer SUN Sparc Ultra 1 durchgeführt und waren auf 10 Minuten Rechenzeit begrenzt. Die Testmusterbestimmung durch ATALANTA betrug dabei allenfalls wenige Sekunden. Als Mustergenerator wurde ein Akkumulator mit einfachem Addierer vorgesehen. Der Parameter Λ war auf Werte der Menge $\{10^x, 2{\cdot}10^x, 5{\cdot}10^x\}$ beschränkt, wobei x eine natürliche Zahl ist. Als Λ_{max} wurde das zweitgrößte $\Lambda \leq 10^7$ gewählt, bei dem harte Fehler auftraten.

Schaltung	# Eingänge	# Ausgänge	# Gatter	Tiefe	# Fehler	# red.
b3	32	19	518	15	1 125	58
b4	33	22	220	10	528	0
bc0	21	11	973	17	2 637	225
chkn	29	7	540	15	1 014	0
cps	24	102	960	44	1 924	0
exep	29	62	415	10	1 105	0
in3	34	29	336	15	740	17
in4	32	20	600	16	1 231	100
in5	24	14	237	12	627	6
in7	26	10	169	12	327	12
jbp	36	57	433	18	923	0
misj	35	12	92	5	107	0
signet	39	8	279	9	615	0
vg2	25	8	109	9	202	0
vtx1	27	6	135	10	238	0
x6dn	38	5	323	13	713	31
x9dn	27	7	116	13	210	0

Tabelle 4.1: Charakteristika der Berkeley-Schaltungen

Beim größten Λ mit harten Fehlern ist deren Anzahl i.a. zu gering, um brauchbare Musterfolgen zu liefern, die auch die leichten Fehler erfassen können. Für die Bestimmung von λ wurde $P_G = P_{korrekt}$ = 99 % festgelegt. Die Wahrscheinlichkeit, daß ein leichter Fehler innerhalb einer Zufallsfolge der Länge Λ erfaßt wird, war demnach größer als 99 %, ebenso die Wahrscheinlichkeit, daß ein tatsächlich harter Fehler durch die Simulation richtig klassifiziert wurde. Für die maximal erlaubte Simulationslänge λ ergibt sich dann gemäß Gleichung (4.2): $\lambda = 0{,}00218 \cdot \Lambda$. Pro Referenzfehler wurde immer nur ein Referenzmuster verwendet, als Referenzfehler der härteste Fehler mit kleinster Testmustermenge gewählt.

Zur Bewertung des Optimierungsverfahrens wurden dessen Ergebnisse mit der Optimierung durch reine Fehlersimulation verglichen. Startwert und Konstante waren jeweils zufällig, die Testlänge wurde durch Fehlersimulation der resultierenden Musterfolge bestimmt. Überschritt die Simulationslänge die bis dato kürzeste Testlänge oder aber die Obergrenze $\Lambda_{max} = 10^7$, brach das Verfahren die Simulation der aktuellen Musterfolge ab und fuhr mit der nächsten zufällig gewählten Folge fort. Die Zeitschranke lag wieder bei 10 Minuten.

Tabelle 4.2 zeigt die Ergebnisse der beiden Verfahren. Angegeben sind die größten Anzahlen der harten Fehler f_{hart}, die bei der Startwertoptimierung berücksichtigt werden mußten – diese Zahl

Schaltung	# f_{hart}	# harte Testmuster	neues Verfahren	Simulations-verfahren	Durch-schnitt	# Konst.	# Folgen
b3	54	851 968	193 537	439 922	1 555 772	1 590	75
b4	24	6 553 600	26 349	27 068	98 055	36	2 313
bc0	45	688	61 441	98 512	250 129	555 920	172
chkn	57	9 392	3 023421	7 969 797	33 342 654	52 510	7
cps	219	3 136	601 037	1 101 768	3 377 164	168 242	21
exep	20	15 616	510 253	1 915 931	5 380 164	33 531	28
in3	28	1 376 256	157 185	314 248	1 288 110	177	182
in4	46	843 776	217 601	388 775	1 551 100	1 228	63
in5	75	157 696	5 350	6 170	34 928	4 227	5 822
in7	31	66 560	9 235	13 691	250 258	4 555	4 151
jbp	50	213 909 504	34 837	12 106	68 838	1	2 594
misj	16	624 951 296	–	511	4 142	0	15 585
signet	–	$> 2^{26}$	–	3 434	16 663	0	7 055
vg2	21	14 688	16 699	27 079	163 915	39 946	3 745
vtx1	43	45 792	80 321	141 124	599 413	16 793	742
x6dn	60	$> 2^{30}$	–	11 539	78 398	0	2 841
x9dn	22	62 208	27 838	43 229	199 125	13 435	2 384

Tabelle 4.2: Testlängen beim neuen Verfahren im Vergleich zum Simulationsverfahren

wächst mit kleiner werdendem Λ. Die folgende Spalte gibt die Gesamtzahl der Testmuster für diese harten Fehler wieder. Anschließend sind die Testlängen des Optimierungsverfahrens und des Simulationsverfahrens dargestellt, darauf, zum Vergleich, die Durchschnittslängen von zufällig gewählten, nicht optimierten Musterfolgen. Zur Bestimmung dieser Werte wurden jeweils 100 Musterfolgen simuliert. Die beiden letzten Spalten enthalten die Zahl der beim Optimierungsverfahren untersuchten Konstanten bzw. die Zahl der beim Simulationsverfahren untersuchten Musterfolgen.

Die optimierten Testlängen sind im Vergleich mit den Durchschnittslängen erheblich kürzer. Dies demonstriert den Optimierungsspielraum, der bei allen Schaltungen gegeben war. Für die Testzeit ist besonders die absolute Anzahl eingesparter Testmuster wichtig. Diese Anzahl beträgt bei ohnehin kurzen Testlängen i.a. nur Bruchteile gegenüber den Einsparungsmöglichkeiten bei großen Testlängen, selbst wenn die relative Einsparung bei letzteren geringer ausfällt. Die Stärken des neuen Optimierungsverfahrens liegen nun gerade bei Schaltungen mit besonders schwer zu entdekkenden Fehlern, d.h. wenn die Testlängen und damit das absolute Einsparungspotential besonders groß sind. Bei diesen Schaltungen ist die Zahl der für die harten Fehler zu berücksichtigenden Testmuster i.a. vergleichsweise gering, und das Optimierungsverfahren arbeitet besonders effektiv. Z.B.

konnten bei der Schaltung *chkn* mit einer Testlänge von mehreren Millionen Mustern über 50 000 Konstanten untersucht werden, beim Simulationsverfahren dagegen nur sieben. Die absolute Einsparung des Optimierungsverfahrens gegenüber der Optimierung durch Fehlersimulation war hier mit fast viermillionen Mustern am größten.

Die Vorteile des Optimierungsverfahrens schwinden, je kürzer die benötigte Testlänge ist. Bei kurzen Testlängen ist die Fehlersimulation selbst schnell genug, um ausreichend viele Musterfolgen zu simulieren. Umgekehrt sinkt beim Optimierungsverfahren die Zahl der betrachteten Konstanten, je mehr Testmuster für die harten Fehler berücksichtigt werden müssen. Bei Schaltung *jbp* waren das bereits über Zweihundertmillionen Testmuster, und die Startwertoptimierung für die erste Konstante benötigte schon 15 Minuten – mehr als die vorgegebene Zeitschranke. Hier wird auch deutlich, daß eine ausreichende Anzahl von Konstanten weit wichtiger ist als nur eine große Zahl von Startwerten. Obwohl die Startwertoptimierung bei *jbp* 2^{36} Startwerte untersucht, ist die gefundene Testlänge fast dreimal länger als die der Fehlersimulation mit ihren nur 2 600 Startwerten aber ebensovielen Konstanten. Bei den Schaltungen *misj* und *x6dn* war die Zahl der Testmuster für das Optimierungsverfahren zu groß, bei *signet* wurden nicht einmal alle Testmuster bestimmt – der härteste Fehler hatte schon mehr als 2^{26}. Allerdings ist bei diesen Schaltungen das Einsparungspotential an Testzeit wegen der kurzen Testlängen ohnehin nur sehr gering.

Tabelle 4.3 zeigt für einige Beispiele anhand des Optimierungsverfahrens den Einfluß der Konstantenanzahl auf die gefundene Testlänge. Pro Konstante wurde jeweils ein zufällig gewähltes Referenzmuster berücksichtigt. Man sieht, daß in der Regel schon nach mehreren hundert Konstanten mit einem guten Ergebnis zu rechnen ist. Eine weitere Verbesserung ist nur noch bei deutlich steigendem Aufwand möglich.

Schaltung	Anzahl Konstanten				
	10	100	1 000	10 000	100 000
bc0	61 441	61 441	61 441	61 441	61 441
chkn	6 416 306	5 318 052	4 402 302	3 531 455	3 023 421
cps	873 213	873 213	873 213	774 143	601 037
exep	1 721 325	613 365	552 381	552 381	510 253
vg2	31 932	31 932	31 359	22 231	16 699
vtx1	132 980	111 233	80 321	80 321	41 938
x9dn	59 649	59 649	40 826	27 838	20 225

Tabelle 4.3: Einfluß der Konstantenanzahl auf die gefundene Testlänge

Einen weiteren interessanten Befund liefert Tabelle 4.4. Sie zeigt den Einfluß der pro Referenzfehler betrachteten Referenzmusteranzahl auf die gefundene Testlänge. Bei jedem Experiment wurde ein und dieselbe Konstante betrachtet. Die Zahl der harten Fehler war für jede Schaltung so angepaßt, daß die gefundenen Testlängen stets unter Λ_{max} lagen. Die letzte Spalte enthält die maximal mögliche Zahl von Referenzmustern. Bei dieser Anzahl wurden alle Referenzmuster aufgezählt, sonst zufällig ausgesucht.

Schaltung	Anzahl Referenzmuster						
	1	2	10	20	100	1 000	max
bc0	155 649	155 649	114 689	106 497	–	–	16
chkn	8 692 790	7 040 623	6 093 401	5 525 533	5 294 199	–	32
cps	873 213	873 213	873 213	–	–	–	8
exep	2 335 989	2 335 989	2 087 897	2 087 897	2 039 285	2 039 285	256
vg2	33 409	33 409	33 409	33 409	32 592	29 660	864
vtx1	197 480	197 480	197 480	195 169	195 169	183 073	864
x9dn	88 193	78 977	53 569	53 569	40 328	29 754	1 728

Tabelle 4.4: Testlänge in Abhängigkeit von der Referenzmuster-Anzahl

Wie zu erwarten, sinkt auch hier die Testlänge mit größer werdender Zahl der Referenzmuster. Die Optimierungsmöglichkeiten werden jedoch stark von der gewählten Konstanten beeinflußt. Das belegt der Vergleich der jeweils besten Ergebnisse aus Tabelle 4.3 und 4.4.

In praktisch allen Fällen können aus Aufwandsgründen nicht alle Konstanten und Referenzmuster erschöpfend untersucht werden. Stattdessen wird die Optimierung nach einer vorgegebenen Zeitschranke abgebrochen. Je mehr Referenzmuster dabei pro Konstante betrachtet werden, um so geringer wird die Zahl der berücksichtigten Konstanten und umgekehrt. Für die Experimente wurden einige Schaltungen ausgetauscht, um auch den Einfluß sehr kleiner Konstantenanzahlen dokumentieren zu können. Tabelle 4.5 enthält die Testlängen bei wachsender Zahl von Referenzmustern, wenn die Rechenzeit auf fünf Minuten begrenzt war. Unter den Testlängen ist die Zahl der untersuchten Konstanten angegeben.

Wie man sieht, hat die Anzahl der überprüften Referenzmuster pro Referenzfehler nur einen geringen Einfluß auf das Ergebnis, solange nur genügend Konstanten berücksichtigt werden können. Wird die Zahl der Konstanten sehr gering, verschlechtert sich i.a. die gefundene Lösung. Die Beschränkung auf ein Referenzmuster ist daher vernünftig, da dann die meisten Konstanten untersucht werden.

Schaltung		Anzahl Referenzmuster						
		1	2	10	20	100	1 000	max.
b3	Testlänge	193 537	209 409	193 281	226 305	254 465	254 465	8 192
	# Konstanten	1 170	1 062	280	33	28	3	
chkn	Testlänge	3 531 455	3 456 454	3 589 890	4 500 918	3 561 392		32
	# Konstanten	11 861	6 058	972	579	481		
cps	Testlänge	601 037	674 573	622 669				8
	# Konstanten	88 856	9 499	20 005				
exep	Testlänge	552 381	425 985	553 857	575 098	588 461	558 769	256
	# Konstanten	34 822	18 613	1 632	792	345	156	
in3	Testlänge	157 185	219 745	258 561	315 185	–	–	16 384
	# Konstanten	71	39	6	3	0	0	
in4	Testlänge	217 601	276 993	229 377	209 153	293 121	436 737	8 192
	# Konstanten	2 039	320	185	93	15	2	

Tabelle 4.5: Einfluß der betrachteten Referenzmuster auf das gefundene Optimum

Zum Abschluß ist festzuhalten, daß das hier vorgestellte Optimierungsverfahren bei schwer testbaren Schaltungen die besten Resultate erzielt. Gerade bei diesen Schaltungen mit den ohnehin großen Testlängen ist eine Reduzierung der Testzeit jedoch notwendig und wegen des hohen Einsparpotentials auch besonders sinnvoll. Bei leicht testbaren Schaltungen mit kurzen Testlängen fällt die Verbesserung gegenüber der Optimierung durch reine Fehlersimulation geringer aus, ist aber auch nicht mehr so wichtig. Da der Rechenaufwand des Optimierungsverfahrens mit der Zahl der Testmuster für die härtesten Fehler steigt, ist es für manche dieser Schaltungen mit sehr vielen solcher Testmuster nicht mehr sinnvoll einsetzbar. In diesem Fall kann man mit dem Verfahren aus [StMa97] weitere Verbesserungen durch die Verwendung mehrerer Konstanten und Startwerte pro Schaltung erzielen.

5 Zusammenfassung

Bei der Produktion integrierter Schaltkreise lassen sich Defekte auf einzelnen Chips nicht vermeiden. Diese fehlerhaften Chips werden mithilfe eines Schaltungstests erkannt und aussortiert. Neben einer hohen Fehlererfassung wird dabei auch eine kurze Testzeit gefordert. Bei den immer komplexer werdenden Schaltungen gewinnt der Selbsttest zunehmend an Bedeutung, bei dem die benötigte Testhardware integraler Bestandteil einer Schaltung ist. Die Testhardware setzt sich aus Mustergeneratoren, Kompaktierern und einer Teststeuerung zusammen. Als Mustergeneratoren und Kompaktierer werden meist interne Register der Schaltung zu sogenannten Testregistern erweitert, was jedoch die Chipkosten erhöht. Eine Reihe existierender Verfahren versucht deshalb, die Anzahl der notwendigen Testregister klein zu halten, um so die Kosten hierfür zu minimieren.

In den letzten Jahren haben sich Akkumulatoren als mögliche, kostengünstige Alternative zu Testregistern erwiesen. Im Gegensatz zu Testregistern fällt bei Akkumulatoren praktisch keine zusätzliche Hardware an. Sie müssen jedoch erst durch eine geeignete Ansteuerung aus einzelnen Schaltungsmodulen konfiguriert werden, was allerdings in vielen Schaltungen, besonders solchen zur digitalen Signalverarbeitung, möglich ist. Trotz der offensichtlichen Vorteile werden Akkumulatoren bei den herkömmlichen Selbsttestverfahren noch kaum genutzt. Ziel dieser Arbeit war es deshalb, Akkumulatoren beim Selbsttest eines Chips zu berücksichtigen und ihre Potentiale voll auszunutzen. Dazu wurden zwei Verfahren entwickelt.

Das erste Verfahren betrachtet Datenpfade auf Register-Transfer-Ebene und minimiert durch den Einsatz von Akkumulatoren die Anzahl notwendiger Testregister. Zusätzlich hebt es typische Einschränkungen anderer Verfahren auf und erreicht dadurch eine weitere Senkung der Ausbaukosten. Wesentliche Punkte sind dabei der Einsatz von Testbarkeitsmaßen zur Wiederverwendung von Testantworten als Testmuster, beliebige Freiheitsgrade bei der Ansteuerung von Steuersignalen und der Transport von konstanten Werten an Seiteneingänge von Modulen, um diese transparent bzw. durchlässig zu schalten. Neben der Plazierung von Testressourcen wird ein Testablaufplan für eine kurze Testzeit erstellt. Dabei wird berücksichtigt, daß während einer Testsitzung mehrere Teiltests parallel stattfinden können. Die bei der Plazierung gewonnenen Randbedingungen für die Ansteuerung werden dafür genutzt, um die gesamte Testzeit abzuschätzen. Schließlich wird aus den Randbedingungen für die einzelnen Testsitzungen eine konkrete Ansteuerung bestimmt, die einen

Kompromiß zwischen einfacher Teststeuerung und kurzer Testzeit darstellt. Aus der so gewonnenen Steuerinformation läßt sich die Teststeuerung mit Standardwerkzeugen zur Schaltwerkssynthese erzeugen. Das Verfahren konfiguriert somit einen kompletten Selbsttest auf Register-Transfer-Ebene. Die experimentellen Ergebnisse bestätigen die Überlegenheit des Verfahrens gegenüber herkömmlichen Verfahren. Trotz seiner Vielseitigkeit werden sehr gute Ergebnisse bereits nach wenigen Minuten Rechenzeit gefunden.

Das zweite Verfahren konzentriert sich auf die Verringerung der Testzeit bei schwer testbaren Schaltungen mit großen Testlängen. Es ist in der Lage, einen Mustergenerator ohne zusätzliche Hardware so an die zu testende Teilschaltung anzupassen, daß alle kombinatorischen Schaltungsfehler durch eine möglichst kurze Musterfolge erfaßt werden können. Dazu optimiert es die beiden Parameter, von denen die erzeugte Musterfolge eindeutig festgelegt wird – bei einem Akkumulator sind das der Startwert und die konstante Eingabe. Die Besonderheit des Verfahrens liegt in der sehr schnellen Berechnung der Testlänge, deren Aufwand anders als bei der üblicherweise sonst eingesetzten Fehlersimulation nicht mit der Testlänge wächst. Mithilfe der analytischen Testlängenberechnung wird für gegebene konstante Eingabewerte der optimale Startwert bestimmt. Trotz der zusätzlichen Startwertoptimierung können mit dem Verfahren meist deutlich mehr konstante Eingabewerte untersucht werden, als es durch reine Fehlersimulation selbst ohne optimierte Startwerte möglich wäre. Auf diese Weise kann die Testlänge bereits nach wenigen Minuten Rechenzeit stark verkürzt werden. Das Verfahren arbeitet besonders effizient bei Akkumulatoren, kann jedoch auch zur Anpassung von LRSRs eingesetzt werden. Wegen der weit komplexeren Berechnungen bei LRSRs sind allerdings dort nur geringere Verbesserungen zu erwarten. Das Verfahren kann ferner Teilschaltungen behandeln, die von zwei unabhängigen Mustergeneratoren gespeist werden.

6 Literatur

AbBF90 M. Abramovici, M. A. Breuer, A. D. Friedman, “Digital Systems Testing and Testable Design,” Computer Science Press, Freeman, New York, 1990

AbBr85 M. S. Abadir, M. A. Breuer, “A Knowledge-Based System for Designing Testable VLSI Chips,” in IEEE Design&Test, vol. 2, no. 4, Aug. 1985, pp. 56-68

AbKR91 M. Abramovici, J. J. Kulikowski, R. K. Roy, “The Best Flip-Flops to Scan,” in Proc. International Test Conference, 1991, pp. 166-173

ADHA95 S. Adham, M. Kassab, N. Mukherjee, K. Radecka, J. Rajski, J. Tyszer, “Arithmetic Built-In Self Test for Digital Signal Processing Architectures,” in Proc. Custom Integrated Circuits Conference, 1995, pp. 659-662

AvMc93 L. J. Avra, E. J. McCluskey, “Synthesising for Scan Dependence in Built-In Self-Testable Designs,” in Proc. International Test Conference, 1993, pp. 734-743

Avra91 LaNae Avra, “Allocation and Assignment in High-Level Synthesis for Self-Testable Data Paths,” in Proc. International Test Conference, 1991, pp. 463-472

BaCR83 Z. Barzilai, D. Coppersmith, A. L. Rosenberg, “Exhaustive Generation of Bit Patterns with Applications to VLSI Self-Testing,” in IEEE Transactions on Computers, vol. 32, no. 2, Feb. 1983, pp. 190-194

BaMc82 P. H. Bardell, W. H. McAnney, “Selt-Testing of Multichip Logic Modules,” in Proc. International Test Conference, 1982, pp. 190-194

BeFR97 D. Berthelot, M. L. Flottes, B. Rouzeyre, “A new TPG structure for Datapath cores,” in 1st IEEE International Workshop on Testing Embedded Core-Based Systems (TECS ’97), 1997, pp. 2.1.1-2.1.6

BeFR98 D. Berthelot, M. L. Flottes, B. Rouzeyre, “OptiBIST: A Tool for BISTing Datapaths,” in Compendium of papers IEEE European Test Workshop, 1998, pp. 123-127

Berk99 M. Berkelaar, “lp_solve (version 2.3),” Eindhoven University of Technology, 1999, ftp://ftp.ics.ele.tue.nl/pub/lp_solve

BhDS97 S. Bhattacharya, S. Dey, B. Sengupta, “An RTL Methodology to Enable Low Overhead Combinational Testing,” in Proc. European Design and Test Conference, 1997, pp. 146-152

BhJh94 S. Bhatia, N. K. Jha, “Behavioral Synthesis for Hierarchical Testability of Controller/Data Path Circuits with Conditional Branches,” in Proc. International Conference on Computer Design, 1994, pp. 91-96

BHMS84 R. K. Brayton, G. D. Hachtel, C. McMullen, A. Sangiovanni-Vincentelli, “Logic Minimization Algorithms for VLSI Synthesis,” Boston: Kluwer, 1984

BhPa89 S. Bhawmik, P. Palchaudhuri, “Expert system to configure global design for testability structure in a VLSI circuit,” in Microprocessors and Microsystems, 1989, pp. 462-472

BrFu85 F. Brglez, H. Fujiwara, “A neutral netlist of 10 combinational benchmark circuits and a target translator in FORTRAN,” in Proc. International Symposium on Circuits and Systems, 1985

BrSe87 I. N. Bronstein, K. A. Semendjajew, „Taschenbuch der Mathemathik“, Verlag Harri Deutsch Thun und Frankfurt/Main, 23. Auflage, 1987, S.745

Brya86 R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipulation,“ in IEEE Transactions on Computers, vol. C-35, August 1986, pp. 677-691

Camp88 R. Camposano, “Structural Synthesis in the Yorktown Silicon Compiler,” in VLSI ’87, VLSI Design of Digital Systems, Amsterdam, The Netherlands: North Holland, 1988, pp. 61-72

CaPa95a J. Carletta, C. Papachristou, “Structural Constraints for Circular Self-Test Paths,” in Proc. VLSI Test Symposium, 1995, pp. 486-491

CaPa95b J. Carletta, C. Papachristou, “Testability Analysis and Insertion for RTL Circuits Based on Pseudorandom BIST,” in Proc. International Conference on Computer Design, 1995, pp. 162-167

ChAg90 K.-T. Cheng, V. D. Agrawal, “A Partial Scan Method for Sequential Circuits with Feedback,” in IEEE Transaction on Computers, vol. 39, no. 4, April 1990, pp. 544-548

ChGu94 C.-H. Chiang, S. K. Gupta, “Random Pattern Testable Logig Synthesis,” in Proc. International Conference on CAD, 1994, pp. 125-128

ChHu90 S. Chakravarty, H. H. Hunt, “On Computing Signal Probability and Detection Probability of Stuck-at Faults,” in IEEE Transactions on Computers, vol. 39, no. 11, Nov. 1990, pp. 1369-1377

ChKS94 C.-H. Chen, T. Karnik, D. G. Saab, “Structural and Behavioral Synthesis for Testability Techniques,” in Transactions on CAD, vol. 13, no. 6, June 1994, pp. 777-785

ChLi95 K.-T. Cheng, C.-J. Lin, “Timing Driven Test Point Insertion for Full-Scan and Partial-Scan,” in Proc. International Test Conference, 1995, pp. 506-514

ChPa91a S. S. K. Chiu, C. A. Papachristou, “A Built-In Self-Testing Approach for Minimizing Hardware Overhead,” in Proc. International Conference on Computer Desing, 1991, pp. 282-285

ChPa91b S. Chiu, C. A. Papachristou, “A Design for Testability Scheme with Applications to Data Path Synthesis,” in Proc. 28th Design Automation Conference, 1991, pp. 271-277

ChPa91c A. Chickermane, J. H. Patel, “A Fault Oriented Partial Scan Design Approach,” in Proc. International Conference on CAD, 1991, pp. 400-403

ChPK95 M. Chatterjee, D. K. Pradhan, W. Kunz, “LOT: Logic Optimization with Testability – New Transformations Using Recursive Learning,” in Proc. International Conference on CAD, 1995, pp. 318-325

ChSa92 C.-H. Chen, D. G. Saab, “Behavioral Synthesis for Testability,” in Proc. International Conference on CAD, 1992, pp. 612-615

ChYu92 C.-I. H. Chen, J. T. Yuen, “Concurrent Test Scheduling in Built-In Self-Test Environment,” in Proc. International Conference on Computer Design, 1992, pp. 256-259

CRBP93 V. Chickermane, E. M. Rudnick, P. Banerjee, J. H. Patel, “Non-Scan Design-for-Testability Techniques for Sequential Circuits,” in Proc. 30th Design Automation Conference, 1993, pp. 236-241

CrKS88 G. L. Craig, C. R. Kime, K. K. Saluja, “Test Scheduling and Control for VLSI Built-In Self-Test,” in IEEE Transactions on Computers, vol. 37, no. 9, Sep. 1988, pp. 1099-1109

DaWW90 W. Daehn, T. W. Williams, K. D. Wagner, “Aliasing errors in linear automata used as multiple-input signature analyzers,” in IBM Journal of Research and Development, vol. 34, no. 2/3, March/May 1990, pp.363-380

DePo94a S. Dey, M. Potkonjak, “Transforming Behavioral Specifications to Facilitate Synthesis of Testable Designs,” in Proc. International Test Conference, 1994, pp. 184-193

DePo94b S. Dey, M. Potkonjak, “Non-Scan Design-For-Testability of RT-Level Data Paths,” in Proc. International Conference on CAD, 1994, pp. 640-645

DePR93 S. Dey, M. Potkonjak, R. K. Roy, “Exploiting Hardware Sharing in High-Level Synthesis for Partial Scan Optimization,” in Proc. International Conference on CAD, 1993, pp. 20-25

DRSC86 H. DeMan, J. Rabaey, P. Six, L.Claesen, “Cathedral II: A Silicon Compiler for Digital Signal Processing,” in IEEE Design&Test, vol. 3, no. 6, Dec. 1986, pp. 13-25

DoWu98 R. Dorsch, H.-J. Wunderlich, “Accumulator Based Deterministic BIST,” in Proc. International Test Conference, 1998, pp. 412-421

Eldr59 R. D. Eldred, “Test Routines Based on Symbolic Logical Statements for Combinational Logic Nets,” in Journal of the ACM, vol. 6, no. 1, 1959, pp. 33-36

FeSh88 F. J. Ferguson, J. P. Shen, “Extraction and Simulation of Realistic CMOS Faults with Inductive Fault Analysis,” in Proc. International Test Conference, 1988, pp. 475-484

FlHR95 M. L. Flottes, D. Hammad, B. Rouzeyre, “High-Level Synthesis for Easy Testability,” in Proc. European Design Automation Conference, 1995, pp. 198-206

FuSh83 H. Fujiwara, T. Shimono, “On the Acceleration of Test Generation Algorithms,” in IEEE Transactions on Computers, vol. 32, no. 12, Dec. 1983, pp. 1137-1144

GhJB98 I. Ghosh, N. K. Jha, S. Bhawmik, “A BIST Scheme for RTL Controller-Data Paths Based on Symbolic Testability Analysis,” in Proc. 35th Design Automation Conference, 1998, pp.554-559

GhRJ96 I. Ghosh, A. Raghunathan, N. K. Jha, “A Design for Testability Technique for RTL Circuits Using Control/Data Flow Extraction,” in Proc. International Conference on CAD, 1996, pp. 329-336

GhRJ97 I. Ghosh, A. Raghunathan, N. K. Jha, “Design for Hierarchical Testability of RTL Circuits Obtained by Behavioral Synthesis,” in IEEE Transactions on CAD, vol. 16, no. 9, Sep. 1997, pp. 1001-1014

GhRJ98 I. Ghosh, A. Raghunathan, N. K. Jha, “A Design-for-Testability Technique for Register-Transfer Level Circuits Using Control/Data Flow Extraction,” in IEEE Transactions on CAD, vol. 17, no. 8, Aug. 1998, pp. 706-723

GhRJ99 I. Ghosh, A. Raghunathan, N. K. Jha, “Hierarchical Test Generation and Design for Testability Methods for ASPP’s and ASIP’s,” in IEEE Transactions on CAD, vol. 18, no. 3, March 1999, pp. 357-370

GiKn84 E. F. Girczyc, J. P. Knight, “An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling,” in Proc. International Conference on Computer Design, 1984, pp. 726-731

GiPZ96 D. Gizopolos, A. Paschalis, Y. Zorian, “An Effective BIST Scheme for Datapaths,” in Proc. International Test Conference, 1996, pp. 76-85

Goel81 P. Goel, “An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits,” in IEEE Transactions on Computers, vol. 30, no. 3, March 1981, pp. 215-222

Golo67 S. W. Golomb, “Shift Register Sequences,” Holden-Day, San Francisco, 1967

Gu95 X. Gu, “RT Level Testability-Driven Partitioning,” in Proc. VLSI Test Symposium, 1995, pp. 176-181

GuRT94 S. Gupta, J. Rajski, J. Tyszer, “Test Pattern Generation Based on Arithmetic Operations,” in Proc. International Conference on CAD, 1994, pp. 117-124

HaFr73 J. P. Hayes, A. D. Friedman, “Test Point Placement to Simplify Fault Detection,” in Proc. International Symposium on Fault-Tolerant Computing (FTCS-3), 1973, pp.73-78

HaOr94a I. G. Harris, A. Orailoglu, “Microarchitectural Synthesis of VLSI Designs with High Test Concurrency,” in Proc. 31st Design Automation Conference, 1994, pp.206-211

HaOr94b I. G. Harris, A. Orailoglu, “Fine-Grained Concurrency in Test Scheduling for Partial-Intrusion BIST,” in Proc. European Design and Test Conference, 1994, pp. 119-123

HaPa93 H. Harmanani, C. A. Papachristou, “An Improved Method for RTL Synthesis with Testability Tradeoffs,” in Proc. International Conference on CAD, 1993, pp. 30-35

HELL95 S. Hellebrand, J. Rajski, S. Tarnick, S. Venkataraman, B. Courtois, “Built-In Test for Circuits with Scan Based on Reseeding of Multiple-Polynomial Linear Feedback Shift Registers,” in IEEE Transaction on Computers, vol. 44, no. 2, Feb. 1995, pp. 223-233

HsRP96 F. F. Hsu, E. M. Rudnick, J. H. Patel, “Enhancing High-Level Control-Flow for Improved Testability,” in Proc. International Conference on CAD, 1996, pp. 322-328

IbSa75 O. H. Ibarra, S. K. Sahni, “Polynomially Complete Fault Detection Problems,” in IEEE Transactions on Computers, vol. 24, no. 3, March 1975, pp. 245-249

IEEE87 IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987, Dec. 10, 1987

IEEE90 IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-1990, May 21, 1990

IyBr89 V. S. Iyengar, D. Brand, “Synthesis of Pseudo-Random Pattern Testable Designs,” in Proc. International Test Conference, 1989, pp. 501-508

JoBa86 N. A. Jones, K. Baker, “An Intelligent Knowledge-Based System Tool for High-Level BIST Design,” in Proc. International Test Conference, 1986, pp. 743-749

JoPP89 W.-B. Jone, C. A. Papachristou, M. Pereira, “A Scheme for Overlaying Concurrent Testing of VLSI Circuits,” in Proc. 26th Design Automation Conference, 1989, pp. 531-536

KaPI93 T. Kameda, S. Pilarski, A. Ivanov, “Notes on Multiple Input Signature Analysis,” in IEEE Transactions on Computers, vol. 42, no.2, Feb. 1993, pp. 228-234

KeRi83 B. W. Kernighan, D. M. Ritchie, „Programmieren in C“, Hanser Verlag, Wien, 1983

KiTH91 K. Kim, J. G. Tront, D. S. Ha, “BIDES: A BIST Design Expert System,” in Journal of Electronic Testing: Theory and Applications, vol. 2, no. 2, 1991, pp. 165-179

KiTH98 H. B. Kim, T. Takahashi, D. S. Ha, “Test Session Oriented Built-in Self-testable Data Path Synthesis,” in Proc. International Test Conference, 1998, pp. 154-163

Kobl87 N. Koblitz: “ A Course in Number Theory and Cryptography,” New York: Springer, 1987

KöMZ79 B. Könemann, J. Mucha, G. Zwiehoff, “Built-In Logic Block Observation Techniques,” in Proc. Test Conference, 1979, pp. 37-41

KrPi89 A. Krasniewski, S. Pilarski, “Circular Self-Test Path: A Low-Cost BIST Technique for VLSI Circuits,” in IEEE Transactions on CAD, vol. 8, no. 1, Jan. 1989, pp. 46-55

LeBl84 J. J. LeBlanc, “LOCST: A Built-In Self-Test Technique,” in IEEE Design&Test, vol. 2, no. 4, Nov. 1984, pp. 45-52

LeGB95 M. Lempel, S. K. Gupta, M. A. Breuer, “Test Embedding with Discrete Logarithms,” in IEEE Transactions on CAD, vol. 14, no. 5, May 1995, pp. 554-566

LeHa91 H. K. Lee, D. S. Ha, “An Efficient Forward Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation,” International Test Conference, 1991, pp. 946-955

LeJW93 T.-C. Lee, N. K. Jha, W. H. Wolf, “Behavioral Synthesis of Highly Testable Data Paths under the Non-Scan and Partial Scan Environments,” in Proc. 30th Design Automation Conference, 1993, pp. 292-297

LeRe90 D. H. Lee, S. M. Reddy, “On Determining Scan Flip-Flops in Partial-Scan Designs,” in Proc. International Conference on CAD, 1990, pp. 322-325

LEVI95 M. E. Levitt et al., “Testability, Debuggability, and Manufacturability Features of the UltraSPARC^{TM}-I Microprocessor,” in Proc. International Test Conference, 1995, pp.157-166

LeWJ92 T.-C. Lee, W. H. Wolf, N. K. Jha, “Behavioral Synthesis for Easy Testability in Data Path Scheduling,” in Proc. International Conference on CAD, 1992, pp. 616-619

LiNB91 S.-P. Lin, C. A. Njinda, M. A. Breuer, “A Systematic Approach for Designing Testable VLSI Circuits,” in Proc. International Conference on CAD, 1991, pp. 496-499

LWJA92 T.-C. Lee, W. H. Wolf, N. K. Jha, J. M. Acken, “Behavioral Synthesis for Easy Testability in Data Path Allocation,” in Proc. International Conference on Computer Design, 1992, pp. 29-32

MAJC91 P. C. Maxwell, R. C. Aitken, V. Johansen, I. Chiang, “The Effect of Different Test Sets on Quality Level Prediction: When is 80% Better than 90% ?” in Proc. International Test Conference, 1991, pp. 358-364

MaSt00 F. Mayer, A. P. Ströle, “A Versatile BIST Technique Combining Test Registers and Accumulators,” in Proc. International Conference on VLSI Design (VLSI Design 2000), 2000, pp. 412-415

MaSt97a F. Mayer, A. P. Ströle, „Bestimmung kurzer Testmusterfolgen beim Selbsttest mit Akkumulatoren“, in Tagungsband 9. ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen, Bremen, 1997, S. 53-56 (Berichte Elektrotechnik der Universität Bremen, Report 1/97)

MaSt97b F. Mayer, A. P. Ströle, „Testzeitverkürzung beim Selbsttest mit Akkumulatoren“, in Tagungsband 8. E.I.S.-Workshop, Hamburg, 1997, S. 77-86 (Tagungsband 1997 als GMD-Studie 318 erschienen)

MaSt97c F. Mayer, A. P. Ströle, “Test Length Reduction for Accumulator-Based Self-Test,” in Proc. International Symposium on Circuits and Systems (ISCAS), 1997, pp. 2705-2708

MaSt98a F. Mayer, A. P. Ströle, „Konfigurierung von arithmetischen Mustergeneratoren und Kompaktierern in Register-Transfer-Schaltungen“, in Tagungsband 10. ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen, Herrenberg, 1998

MaSt98b F. Mayer, A. P. Ströle, “Configuring Arithmetic Pattern Generators and Response Compactors from the RT-Moduls of a Circuit,” in Proc. 7th Asian Test Symposium, 1998, pp. 15-20

MaSt99 F. Mayer, A. P. Ströle, „Konfigurierung und Ablaufplanung für den Selbsttest mit Akkumulatoren“, in Tagungsband 11. ITG/GI/GMM-Workshop Testmethoden und Zuverlässigkeit von Schaltungen und Systemen, Potsdam, 1999

McBo81 E. J. McCluskey, S. Bozorgui-Nesbat, “Design for Autonomous Test,” in IEEE Transactions on Computers, vol. 30, no. 11, Nov. 1981, pp. 866-874

McCa92 T. R. McCalla, “Digital Logic and Computer Design,” Macmillan Publishing Company, New York, 1992

McCL86 E. J. McCluskey, “Logic Design Principles with Emphasis on Testable Semicustom Circuits,” Prentice-Hall, Englewood Cliffs, 1986

McPC90 M. C. McFarland, A. C. Parker, R. Camposano, “The High-Level Synthesis of Digital Systems,” in Proc. of the IEEE, vol. 78, no. 2, Feb. 1990, pp.301-318

MKRT95 N. Mukherjee, M. Kassab, J. Rajski, J. Tyszer, “Arithmetic Built-In Self Test for High-Level Synthesis,” in Proc. VLSI Test Symposium, 1995, pp. 132-139

MuSJ92 A. Mujumdar, K. Saluja, R. Jain, “Incorporating Testability Considerations in High-Level Synthesis,” in Proc. International Symposium on Fault-Tolerant Computing (FTCS-22), 1992, pp. 272-279

Neum75 K. Neumann, „Operations Research Verfahren, Band 1“ Carl Hanser Verlag München Wien, 1975

NiPa91 T. M. Niermann, J. H. Patel, “HITEC: A Test Generation Package for Sequential Circuits,” in Proc. European Design Automation Conference, 1991, pp. 214-218

OhWM87 M. J. Ohletz, T. W. Williams, J. P. Mucha, “Overhead in Scan and Self-Test Designs,” in Proc. International Test Conference, 1987, pp. 460 -470

OrHa93 A. Orailoglu, I. G. Harris, “Test Path Generation and Test Scheduling for Self-Testable Designs,” in Proc. International Conference on Computer Design, 1993, pp. 528-531

PaBa97 C. Papachristou, M. Baklashov, “A Test Synthesis Technique Using Redundant Register Transfers,” in Proc. International Conference on CAD, 1997, pp. 414-420

PaCH91 C. A. Papachristou, S. Chiu, H. Harmanani, “A Data Path Synthesis Method for Self-Testable Designs,” in Proc. 28th Design Automation Conference, 1991, pp. 378-384

PaGB98a I. Parulkar, S. K. Gupta, M. A. Breuer, “Introducing Redundant Computations in a Behavior for Reducing BIST Resources,” in Proc. 35th Design Automation Conference, 1998, pp. 548-553

PaGB98b I. Parulkar, S. K. Gupta, M. A. Breuer, “Scheduling and Module Assignment for Reducing BIST Resources,” in Proc. Desing, Automation and Test in Europe, 1998, pp. 66-73

PaKn87 P. G. Paulin, J. P. Knight, “Force-Directed Scheduling in Automatic Data Path Synthesis,” in Proc. 24th Design Automation Conference, 1987, pp. 195-202

PaKn89 P. G. Paulin, J. P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s,” in IEEE Transactions on CAD, vol. 8, no. 6, June 1989, pp. 661-679

PeWe72 W. W. Peterson, E. J. Weldon, “Error-Correcting Codes,” 2. Edition, MIT Press, Cambridge, 1972

PoDR95a M. Potkonjak, S. Dey, R. K. Roy, “Considering Testability at Behavioral Level: Use of Transformations for Partial Scan Cost Minimization Under Timing and Area Constraints,” in IEEE Transactions on CAD, vol. 14, no. 5, May 1995, pp. 531-546

PoDR95b M. Potkonjak, S. Dey, R. K. Roy, “Behavioral Synthesis of Area-Efficient Testable Designs Using Interaction Between Hardware Sharing and Partial Scan,” in IEEE Transactions on CAD, vol. 14, no. 9, Sep. 1995, pp. 1141-1154

POLB88 M. M. Pradhan, E. J. O’Brien, S. L. Lam, J. Beausang, “Circular BIST with Partial Scan,” in Proc. International Test Conference, 1988, pp. 719-729

PoRe93 I. Pomeranz, S. M. Reddy, “A Learning-Based Method to Match a Test Pattern Generator to a Circuit-Under-Test,” in Proc. International Test Conference, 1993, pp. 998-1007

RaLJ98 S. Ravi, G. Lakshminarayana, N. J. Jha, “TAO: Regular Expression based High-Level Testability Analysis and Optimization,” in Proc. International Test Conference, 1998, pp. 331-340

RaJL99 S. Ravi, N. J. Jha, G. Lakshminarayana, “TAO-BIST: A Framework for Testability Analysis and Optimization of RTL Circuits for BIST,” in Proc. VLSI Test Symposium, 1999, pp.398-406

RaRT97 K. Radecka, J. Rajski, J. Tyszer, “Arithmetic Built-In Self-Test for DSP Cores,” in IEEE Transactions on CAD, vol. 16, no. 11, Nov. 1997, pp. 1358-1369

RaTy93a J. Rajski, J. Tyszer, “Test Responses Compaction in Accumulators with Rotate Carry Adders,” in IEEE Transactions on CAD, vol. 12, no. 4, April 1993, pp. 531-539

RaTy93b J. Rajski, J. Tyszer, “Accumulator-Based Compaction of Test Responses,” IEEE Transactions on Computers, vol. 42, no. 6, June 1993, pp. 643-650

RoCa85 W. Rosenstiel, R. Camposano, “Synthesizing Circuits from Behavioral Level Specifications,” in Proc. 7th International Conference on Computer Hardware Description Languages and their Applications, C. Koomen and T. Moto-oka, Eds., North-Holland, August, 1985, pp. 391-402

Roth66 J. P. Roth, “Diagnosis of Automata Failures: A Calculus and a Method,” in IBM Journal of Research and Development, vol. 10, no. 4, July 1966, pp. 278-291

SaMc90 J. Savir, W. H. McAnney, “A Multiple Seed Linear Feedback Shift Register,” in Proc. International Test Conference, 1990, pp. 657-659

SeTS91 B. H. Seiss, P. M. Trouborst, M. H. Schulz, “Test Point Insertion for Scan-Based BIST,” in Proc. European Test Conference, 1991, pp. 253-262

StCD93 J. Steensma, F. Catthoor, H. De Man, “Partial Scan at the Register-Transfer Level,” in Proc. International Test Conference, 1993, pp. 488-497

StMa97 A. P. Ströle, F. Mayer, “Methods to Reduce Test Application Time for Accumulator-Based Self-Test,” in Proc. VLSI Test Symposium, 1997, pp. 48-53

StMa99 A. P. Ströle, F. Mayer, “Test Scheduling with Loop Folding and Its Application to Test Configurations with Accumulators,” erscheint in Proc. 8th Asian Test Symposium, Nov. 1999

Strö95a A. P. Ströle, “A Self-Test Approach Using Accumulators as Test Pattern Generators,” in Proc. International Symposium on Circuits and Systems (ISCAS), 1995, pp. 2120-2123

Strö95b A. P. Ströle, „Testmustererzeugung mit arithmetischen Funktionseinheiten“, 7. E.I.S.-Workshop, Chemnitz, 1995, S. 109-118 (Tagungsband 1996 als GMD-Studie 280 erschienen)

Strö96a A. P. Ströle, “Test Response Compaction Using Arithmetic Functions,” in Proc. VLSI Test Symposium, 1996, pp. 380-386

Strö96b A. P. Ströle, “Arithmetic Pattern Generators for Built-In Self-Test,” in Proc. International Conference on Computer Design, 1996, pp. 131-134

Strö98a A. P. Ströle, “Bit Serial Pattern Generation and Response Compaction Using Arithmetic Functions,” in VLSI Test Symposium, 1998, pp. 48-54

Strö98b A. P. Ströle, „Entwurf selbsttestbarer Schaltungen“, Teubner-Texte zur Informatik Band 27, Teubner Stuttgart, 1998

ThAb89 K. Thearling, J. Abraham, “An Easily Computed Functional Level Testability Measure,” in Proc. International Test Conference, 1989, pp. 381-390

Thom96 K. M. Thompson, “Intel and the Myths of Test,” in IEEE Design&Test, vol. 13, no. 1, 1996, pp. 79-81

ToMc95 N. A. Touba, E. J. McCluskey, “Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST,” in Proc. International Test Conference, 1995, pp. 674-682

ToMc96 N. A. Touba, E. J. McCluskey, “Test Point Insertion Based on Path Tracing,” in Proc. International Test Conference, 1994, pp. 2-8

ToMc99 N. A. Touba, E. J. McCluskey, “RP-SYN: Synthesis of Random Pattern Testable Circuits with Test Point Insertion,” in IEEE Transactions on CAD, vol. 18, no. 8, Aug. 1999, pp. 1202-1213

TSEN88 C. Tseng, R. Wei, S. G. Rothweiler, M. M. Tong, A. K. Bose, “Bridge: A Versatile Behavioral Synthesis System,” in Proc. 25th Design Automation Conference, 1988, pp. 415-420

VaOr95 M. Vahidi, A. Orailoglu, “Testability Metrics for Synthesis of Self-Testable Designs and Effective Test Plans,” in Proc. VLSI Test Symposium, 1995, pp. 170-175

VPNH95 I. Voyiatzis, A. Paschalis, D. Nikolos, C. Halatsis, “Accumulator-Based BIST Approach for Stuck-Open and Delay Fault Testing,” in Proc. Eropean Design and Test Conference, 1995, pp. 431-435

VTAA93 P. Vishakantaiah, T. Thomas, J. A. Abraham, M. S. Abadir, “AMBIANT: Automatic Generation of Behavioral Modifications For Testability,” in Proc. International Conference on Computer Design, 1993, pp. 63-66

WaMc86a L.-T. Wang, E. J. McCluskey, “Concurrent Built-in Logic Block Observer (CBILBO),” in Proc. International Symposium on Circuits and Systems, 1986, pp. 1054-1057

WaMc86b L.-T. Wang, E. J. McCluskey, “Complete Feedback Shift Register Design for Built-In Self-Test,” in Proc. International Conference on CAD, 1986, pp. 56-59

WuFu95 B. Wurth, K. Fuchs, “A BIST Approach to Delay Fault Testing with Reduced Test Length,” in Proc. European Design Automation Conference, 1995, pp. 418-423

WuKi96 H.-J. Wunderlich, G. Kiefer, “Bit-Flipping BIST,” in Proc. International Conference on CAD, 1996, pp.337-343

Wund85 H.-J. Wunderlich, “PROTEST: A Tool for Probabilistic Testability Analysis,” in Proc. IEEE Design Automation Conference, 1985, pp. 204-211

Wund87 H.-J. Wunderlich, “Self Test Using Unequiprobable Random Patterns,” in Proc. International Symposium on Fault-Tolerant Computing (FTCS-17), 1987, pp. 258-263

Wund91 H.-J. Wunderlich, „Hochintegrierte Schaltungen: Prüfgerechter Entwurf und Test“, Springer, Berlin, 1991