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

Studienarbeit Nr. 2269

Hardware Entwurf eines eingebauten Selbsttests für digitale Schaltnetze

Stefan Bayha

Studiengang:	Informatik
Prüfer:	Prof. Dr. habil. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inf. Stefan Holst
begonnen am:	1. April 2010
beendet am:	30. September 2010
CR-Klassifikation:	B.5.1, B.5.3

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Motivation

Das Testen von Schaltnetzen war schon immer eine große Herausforderung um die korrekte Funktionsweise digitaler Schaltnetze nach der Produktion sowie im Betrieb der Schaltung sicher zu stellen.

Hieraus resultieren diverse Ansätze, Schaltnetze effizient und wirtschaftlich testen zu können. Ein Ansatz stellt der Built-in self-test (BIST) dar. Aufgrund der Integration der Testschaltung in die zu testende Schaltung ergeben sich unter anderem enorme Einsparungen an umfangreichen und teuren externen Testgeräten. Auch erschließt sich hieraus die Möglichkeit, sicherheitsrelevante Schaltnetze dem Tester nicht offen legen zu müssen. Ein solcher Ansatz zum Test sicherheitsrelevanter Schaltnetze stellt der während der Studienarbeit zu implementierende Built-in self-test dar. Diese Built-in self-test Architektur zeichnet sich durch eine hohe Einsparung an zusätzlich benötigter Hardware aus und kann somit platzsparend in zukünftigen Schaltungen integriert werden.

1.2 Aufgabenstellung

Ziel der Studienarbeit ist es, die in der Publikation 'Restrict Encoding for Mixed-mode BIST' [H+09] beschriebene mixed-mode Built-in self-test (BIST) Architektur (Abbildung 1.1) in einer Hardwarebeschreibungssprache zu implementieren.

Das Ergebnis der Implementierung stellt eine Architektur in der Hardwarebeschreibungssprache VHDL dar. Diese in VHDL entstandene Hardware galt es nach der Implementierung umfangreich zu validieren um Implementierungsfehler ausschließen zu können und die Korrektheit der Ausgaben sicher zu stellen. Die Validierung der Implementierung gestaltete sich durch eine Implementation eines Referenzmodells derselben BIST Architektur mittels einer Hochsprache. Dieses zur Validierung nötige Referenzmodell wurde in der Hochsprache Java implementiert.

Nach dem erfolgreichen Test des Entwurfs erfolgte eine Analyse der VHDL Implementierung. Diese wurde auf den benötigten Flächenbedarf, die Anzahl benötigter Zellen sowie den kritischen Pfad hin untersucht und dokumentiert.

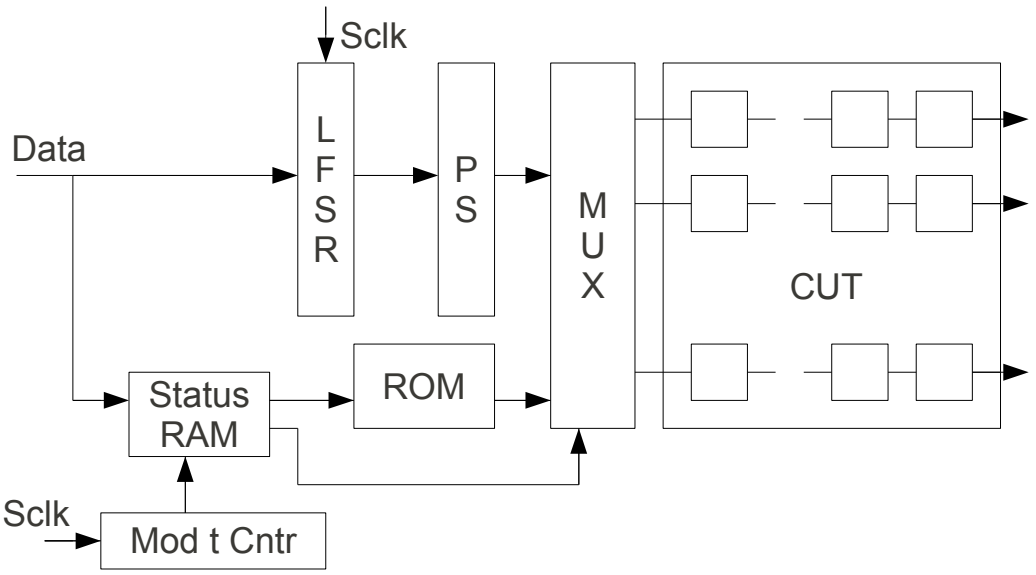

Abbildung 1.1: Aufbau des BIST

1.3 Gliederung der Ausarbeitung

Die Ausarbeitung der vorliegenden Studienarbeit gliedert sich in 5 Kapitel.

Das erste Kapitel enthält eine kurz gefasste Aufgabenstellung zu dieser Studienarbeit sowie eine Beschreibung der Gliederung der Ausarbeitung.

Im zweiten Kapitel werden die Grundlagen, welche für das Verständnis der vorliegenden Arbeit nötig sind, erörtert. Dabei werden zunächst Grundlagen über den Sinn und Zweck des Hardwaretests betrachtet. Im Anschluss daran werden Techniken zum Test von digitalen Schaltnetzen vorgestellt und deren Vor- und Nachteile diskutiert. Zum Abschluss dieses Kapitels, werden alle für den BIST notwendigen Komponenten mit ihren Funktionalitäten vorgestellt.

Das dritte Kapitel enthält eine Beschreibung über die Struktur sowie den Aufbau des zu implementierenden BIST. Hierbei wird grundlegend auf die in der Hardwarebeschreibung VHDL Implementierung sowie auf die Java Implementierung eingegangen und deren Struktur und Aufbau erläutert. Weiterhin enthält dieses Kapitel eine Beschreibung weiterer Software, welche für die automatische Validierung des BIST erstellt und genutzt wurde.

Das vierte Kapitel enthält experimentelle Ergebnisse, welche aus der VHDL Implementierung gewonnen wurden. Hierbei wurde die Flächeneffizienz sowie die kritische Pfadlänge der Schaltung im Bezug auf unterschiedliche Busbreiten dokumentiert.

Im fünften Kapitel befindet sich zum Abschluss der Dokumentation eine Zusammenfassung der Ausarbeitung.

2 Grundlagen

Immer komplexer werdende elektronische Baugruppen ziehen immer größere Fehlermöglichkeiten in der Produktion sowie im Entwurf mit sich. Hierbei besteht die Notwendigkeit, zum einen produktionsbedingte Fehler im Halbleiter ausfindig zu machen. Diese Fehler resultieren aus defekten Halbleitermaterialien oder Staubeinschlüssen während des Produktionsprozesses. Zum anderen müssen neu erstellte Schaltungen auf ihre Struktur untersucht werden. Entsprechende Designfehler müssen gefunden werden, um Änderungen an der Schaltung vornehmen zu können. Des Weitern existiert die Notwendigkeit an den Kunden ausgelieferte Schaltungen weiterhin während des gesamten Lebenszyklus der Schaltung testen zu müssen. Hierdurch ist es nicht mehr möglich externe Testgeräte zum Einsatz zu bringen. Hierfür muss eine effiziente Möglichkeit geschaffen werden die ausgelieferte Schaltung ohne externe Beschaltungen testbar zu machen.

Für das Detektieren unterschiedlicher Fehler existieren diverse Möglichkeiten. Die erste hier beschriebene Möglichkeit sieht vor, die zu testende Schaltung mittels eines externen Testgerätes (ATE) wie in 2.1 gezeigtem Aufbau zu testen.

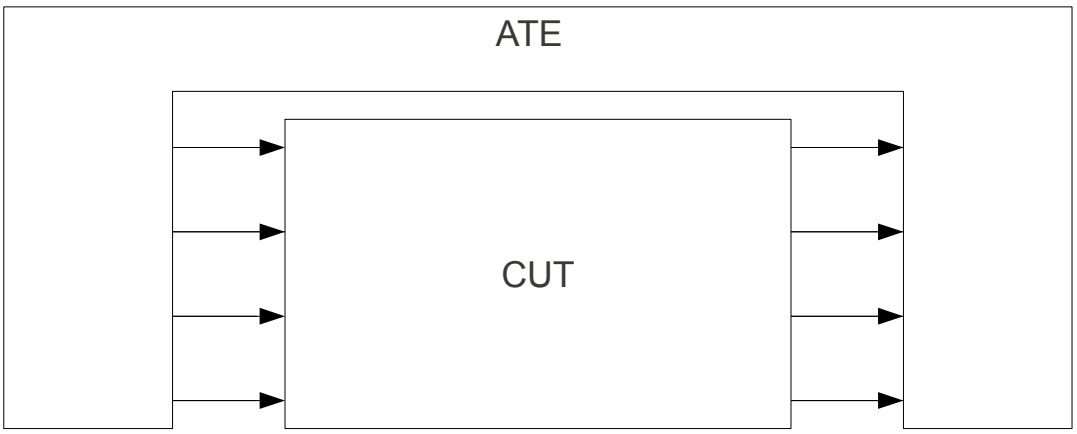

Abbildung 2.1: ATE

Bei dieser Methode handelt es sich um ein Testgerät, welches die zu testende Schaltung (CUT) auf Fehler überprüft. Der Aufbau dieses Verfahrens sieht ein automatisches Test Equipment (ATE) vor, welches sich extern des CUT befindet. Das ATE hat während des Tests die Aufgabe, an den CUT diverse Kombinationen an Testmustern anzulegen. Der CUT reagiert im folgenden auf diese Eingangsdaten mit der ihm vorgesehenen Implementierung. Die Ausgangsdaten des CUT werden daraufhin vom ATE wieder entgegen genommen und auf ihre Korrektheit untersucht. Ein Vorteil dieser Technik ist beispielsweise der geringe zusätzliche Flächenbedarf auf der Schaltung. Als nachteilig stellt sich jedoch die immer teurer werdenden ATEs heraus. Der Grund für den massiven Testkostenanstieg bei der Verwendung eines ATE ist, dass der ATE eine schnellere Berechnung als der CUT aufweisen muss. Bei modernen Schaltungen ist der CUT jedoch schon technologisch relativ ausgereift, sodass ein Übertreffen dessen Geschwindigkeit nur mit teuren Methoden möglich ist [WA99].

Eine Reaktion auf diese Nachteile stellen heute Testverfahren dar, welche die Schaltung per „Scan chains“ [Abbildung 2.2] auf Fehler untersuchen. Hierzu werden die FlipFlops, welche im normalen Betrieb der Schaltung die vorgesehene Logik darstellen, als sogenannte „Scan chains“ zusammen geschalten. Hierbei werden die schaltungsinternen FlipFlops zu einer Reihe hintereinander, als Kette beschaltet. Die am Anfang der n-FlipFlop langen Kette eingelesenen Testdaten werden durch die Kette bis ans Ende propagiert und dort anschließend vom ATE ausgewertet. Diese Durchpapagierung hat zur Folge, dass ein Testbit, in einer n-FlipFlop langen Scan Chain, erst nach n Taktzyklen der Auswerteeinheit zur Verfügung steht. Da dies eine lange und somit kostenintensive Testzeit hervorruft, werden die Scan chains parallel betrieben. Dadurch ist eine Reduzierung der Scan chain Länge möglich was eine Reduzierung der Testzeit zur Folge hat [H$^+$96].

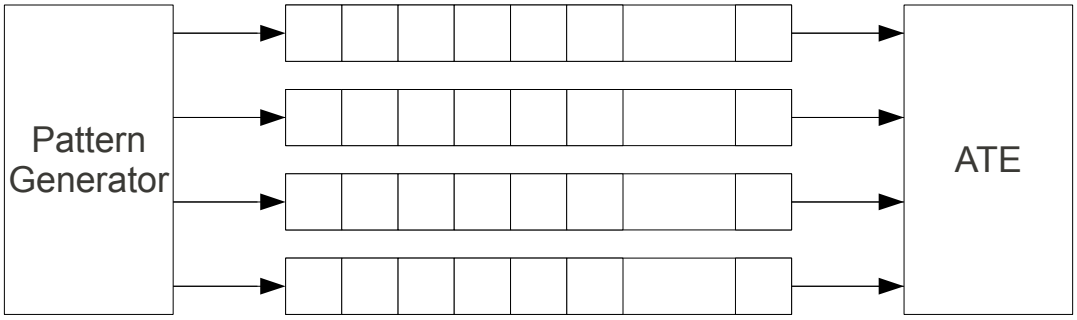

Abbildung 2.2: Aufbau paralleler Scan Chains

Ein auf diesen Ansatz aufbauendes Testverfahren stellt der „Built-in self-test“ dar. Hierbei wird zum Teil oder vollständig auf das externe Testgerät, welches den Testverlauf steuert und die Testvektoren erzeugt, verzichtet. Diese Methode bringt mehrere Vorteile mit sich.

Zum einen wird eine enorme Kostenreduzierung am Testgerät erzielt, da ein komplexes und somit kostenintensives Testgerät entfällt. Zum anderen wird durch dieses Verfahren ein Schutz des IP Cores möglich. Hierzu wird der BIST durchlaufen und das MISR meldet schließlich nur noch die korrekte Funktion des System-under-Test bzw. einen detektierten Defekt der untersuchten Schaltung.

Zum Einsatz kommen solche Verfahren unter anderem auf Smartcards bei den es wichtig ist, das geistige Eigentum des Chipdesins vor dem Kopieren und Widerverwenden durch die Konkurenz zu schützen [S+03][Wun98][B+01]. Weiterhin gestaltet sich der Vorteil, durch den Einsatz einer BIST Implementierung, dass hierdurch die funktionalen Vorgänge der implementierten Schaltung besser versteckt werden können. Hierdurch wird es möglich, dass bei sicherheitskritischen Anwendungen die funktionale Struktur nicht ohne erhöhten Aufwand nachvollzogen werden kann und ein Missbrauch erschwert wird.

Ein BIST gestaltet sich, wie in der Abbildung 2.3 zu sehen, durch die Erweiterung der Scan chains um einen Testvektoren Generator (PG) sowie einer Auswerteeinheit für die in den Scan chains detektierten Bitveränderungen. Als Testvektoren Generator kommt ein lineares Schieberegister (LFSR) mit oftmals nachgeschaltetem Phase Shifter (PS) zum Einsatz. Das lineare Schieberegister erzeugt pro Taktzyklus einen pseudozufälligen Testvektor. Der nachgeschaltete Phase Shifter dient der erhöhten Variierung der Testvektoren, da die im LFSR erzeugten Vektoren hohe Ähnlichkeit aufweisen.

Zur Auswertung der Vektoren kommt bei vielen BIST Architekturen ein „Multiple Input Signature Register“ (MISR) zum Einsatz. Dieses hat die Aufgabe aus allen Scan chain Ketten eine Signatur zu generieren, so dass nur ein einziges Signal als Ausgabe bereit steht. Dieses einzelne Signal repräsentiert je nach Belegung die korrekte Funktion des Circuit-under Test bzw. einen detektierten Fehler. Weiterhin kommen zum Teil ROM's zum Einsatz um neben dem Test mit Pseudozufallsvektoren auch die Fehler welche gegen Pseudozufallsvektoren resistent sind mittels deterministischen Testmustern erfassen zu können [TM96b]. Hierbei werden die für die deterministischen Testmuster nötigen Vektoren vor Implementierung des BIST errechnet und dem BIST bei dessen Implementierung im ROM gespeichert. Dieses Vorgehen erhöht die Wahrscheinlichkeit der Fehlererkennung.

In der vorliegenden Studienarbeit zu implementierenden und validierenden BIST sah die Architektur einen BIST mit den Komponenten LFSR, Phase Shifter, Status-RAM, ROM, Modulo-t-Counter und einem Multiplexer zur Generierung der Testvektoren für den CUT vor. Im folgenden werden diese in der Architektur enthaltenen Komponenten auf ihre theoretischen Hintergründe erörtert.

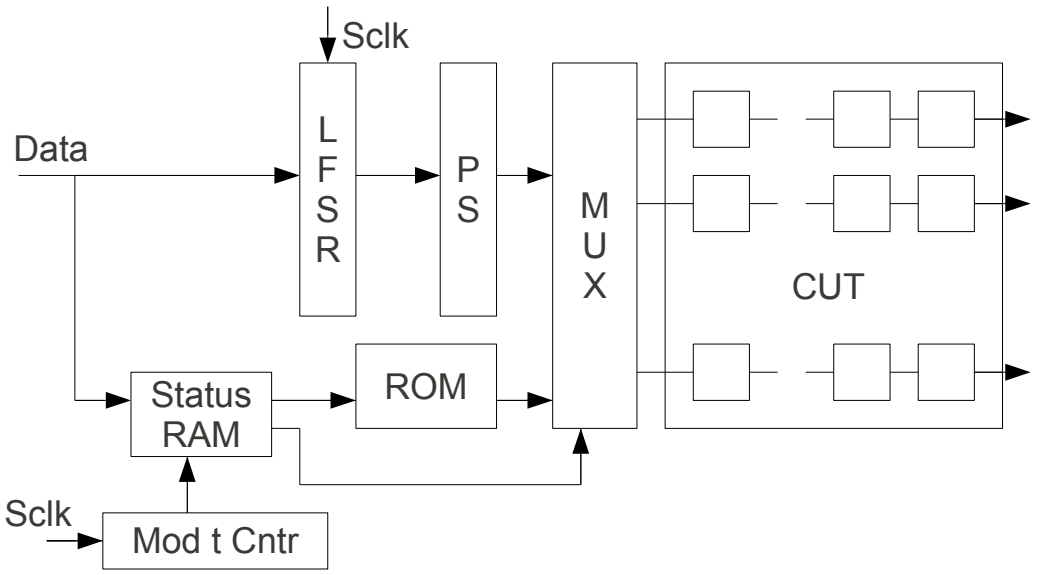

Abbildung 2.3: Decoder

2.1 LFSR

Ein lineares rückgekoppeltes Schieberegister (LFSR) findet in diversen Anwendungen wie beispielsweise in der Kryptografie, in Datenkompressionsverfahren, als Modulo-t-Counter und vielen weiteren Anwendungen eine große Anwendungsverbreitung. In einer BIST Architektur dient ein LFSR zur Erzeugung von Pseudozufallszahlen sowie im späteren Verlauf zur Erzeugung von Testsignaturen. Auf die Erstellung von Testsignaturen, wird im Kapitel MISR [2.6] detailliert eingegangen.

Ein LFSR, mit dem Ziel der Testmustererzeugung, hat die Aufgabe zu jedem Takt seinen Zustand zu ändern und somit pro Takt eine neue Pseudozufallszahl zu generieren. Diese so erzeugten Pseudozufallszahlen dienen im weiteren Verlauf des Testverfahrens als Testvektoren, indem sie de CUT als Eingangssignale dienen.

Als lineare Schieberegister sind zwei Arten von Implementierungen möglich:

1. Fibonacci LFSR
2. Galois-LFSR

Der erste LFSR-Typ wäre wie in Abbildung 2.4 dargestellt ein LFSR und nennt sich Fibonacci LFSR.

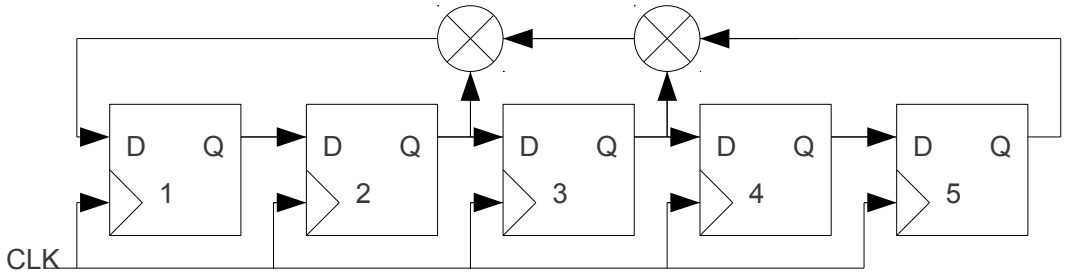

Abbildung 2.4: Fibonacci LFSR

Der Aufbau eines Fibonacci LFSR gestaltet sich durch Verkettung von aneinandergereihten Flip-Flops zu einem Schieberegister. Im rückgekoppelten Schaltnetz befinden sich XOR Bausteine, welche die Eingänge des D-FlipFlops beeinflussen und so deren Zustandsfolge definieren. Platziert werden die XOR Elemente gemäß der Vorgabe des ausgewählten Generatorpolynoms. An allen Stellen, an denen das Generatorpolynom eine *1* liefert, wird eine Rückkopplung über ein XOR Gatter realisiert. Im Gegensatz hierzu existieren so genannte Galois-LFSR. Die Abbildung 2.5 zeigt den Aufbau eines solchen Galois-LFSR grafisch.

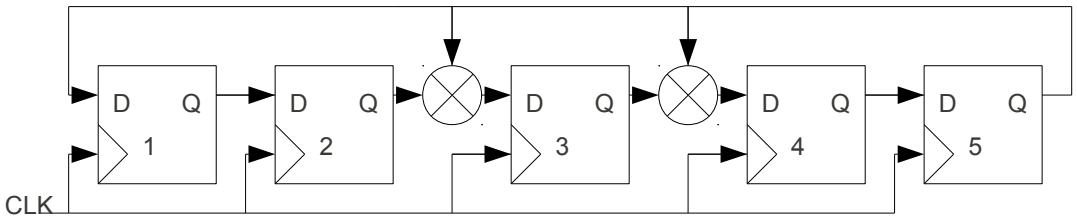

Abbildung 2.5: Galois LFSR

Galois-LFSR unterscheiden sich von den Fibonacci LFSR lediglich im strukturellen Aufbau. Ein Galois-LFSR besitzt die rückgekoppelten XOR Gatter nicht im rückgekoppelten Pfad, sondern direkt zwischen den D-FlipFlops.

Definiert werden beide Typen über ein Polynom mit dem Polynomgrad n.

Zustand n	LFSR Ausgabe
0	1 1 1
1	1 0 1
2	1 0 0
3	0 1 0
4	0 0 1
5	1 1 0
6	0 1 1

Tabelle 2.1: LFSR Zustände des primitiven Polynom $p(x) = x^3 + x + 1$

$$p(x) = x^n + ... + x^2 + x^1 + 1$$

Dieses Polynom wird charakteristisches Polynom oder Feedback Polynom genannt. Charakteristisch für den Einsatz von Polynomen in einem LFSR stellt sich die Tatsache dar, dass nicht durch alle existierenden Polynome bei einem Polynomgrad der Größe n in ihrer Zustandsfolge 2^n Zustände erreicht werden können. Jedoch stehen Polynome zur Verfügung, welche $2^n - 1$ Zustände erreichen. Bei dieser Menge an Polynomen handelt es sich um primitive Polynome.

Beide Arten von linear rückgekoppelten Schieberegister weisen trotz unterschiedlicher Implementierung des selben Polynoms die selbe Anzahl an Zuständen auf. Lediglich die Reihenfolge des Durchlaufes der Zustände unterscheidet sich. Beide LFSR Typen sind somit ineinander verführbar. Der Grund für die Überführbarkeit ist, dass zu jedem primitiven Polynom mit Grad > 2 ein Zwillingspolynom besteht.

Die Wahl des LFSR-Typs ist letztendlich von Faktoren wie der gewünschten Implementierungseffizienz und der Geschwindigkeit des LFSR abhängig. In Anbetracht der Flächeneffizienz können beispielsweise dünn besetzte Generatorpolynome zur Anwendung kommen um Chipfläche zu sparen. Bei dünn besetzten Generatorpolynomen handelt es sich um primitive Polynome, welche mit möglichst wenig XOR Elementen auskommen, um den Platzbedarf zu senken.

Gleichgültig welcher LFSR Typ jedoch zum Einsatz kommt, ist darauf zu achten, dass als Generatorpolynom für den LFSR ein primitives Polynom gewählt wird. Nur durch den Einsatz von primitiven Polynomen mit einem Grad n lassen sich $2^n - 1$ Zustände des LFSR erreichen. Das primitive Polynom $p(x) = x^3 + x + 1$ beispielsweise liefert mit einem Polynomgrad der Ordnung 3 die in der Tabelle 2.1 angegebenen Zustände.

LFSR Zustandsmengen mit 2^n Zustände sind nicht möglich. Dies resultiert aus der Tatsache, dass bei Erreichen der 0-Zustandsfolge kein Verlassen dieses Zustandes in einen anderen

Folgezustand als der 0-Folge möglich wäre. Dies hängt mit der Tatsache zusammen, dass die XOR Verknüpfung $0 \oplus 0$ immer eine 0 liefert. Andere Verknüpfungen als XOR sehen primitive Generatorpolynome nicht vor. Somit ist keine Veränderung des LFSR Zustandes mehr möglich. Der betroffene LFSR würde somit statisch nur noch die 0-Folge liefern. Als Spezialfall sind jedoch LFSR implementierbar, welche die 0-Folge beinhalten möglich. Hier wird jedoch eine zusätzliche Hardware nötig, um die 0-Folge und somit 2^n Zustände zu erreichen und aus dieser wieder zu entkommen. Dies beansprucht jedoch kostbare und teure Chipfläche.

Außerdem lässt sich jedes primitive Polynom in der Form schreiben.

$$P(x) = c_0 \oplus c_1^1 \oplus ...c_n^n$$

Das Berechnen primitiver Polynome benötigt relativ viel Rechenaufwand und somit relativ viel Zeit. Hierzu werden primitive Polynome in Look-Up-Tabellen bereitgehalten. Während der Implementierung von linearen Schieberegister aus primitiven Polynomen werden die Polynome daher entsprechenden Tabellen entnommen.

Da ein LFSR mit primitivem Polynom eine recht hohe Linearität aufweist hat dies zur Folge, dass nicht alle vom LFSR generierten Vektoren in der generierten Reihenfolge als sinnvolle Testvektoren in Betracht gezogen werden können. Um diese hohe Linearität zu beseitigen, existiert die Möglichkeit, ein Reseeding des LFSR durchzuführen. Hierbei bleibt das implementierte primitive Polynom erhalten. Lediglich kommen weitere XOR Elemente hinzu, welche über externe Eingänge verfügen und somit den aktuellen Zustand des LFSR beeinflussen können.

Ein weiterer Nachteil stellt die Tatsache dar, dass man nicht in der Lage ist alle Fehler mittels den von einem LFSR erzeugten Pseudozufallszahlen im CUT zu finden [AYM03]. Dies mindert die Fehlerüberdeckung. Umgangen werden kann dies Minderung der Fehleraufdeckung durch ein Redesign des CUT oder durch das Einbringen von Testpunkten in den CUT [Eic83][TM96a].

2.2 Phase shifter

Die vom LFSR gelieferten Zustände, genannt Vektoren, haben zwar den Vorteil aus einer effizienten und platzsparenden Technik zu stammen, haben jedoch den Nachteil, nicht alle benötigten Testvektoren zu enthalten [TM09][WA99][RT98]. Weiterhin können nicht alle Testsequenzen, genannt Pattern, erreicht werden oder werden nur dann erreicht, wenn lange Zeit keine für den Test nützlichen Vektoren generiert werden können. Den hierdurch entstandenen Zeitverlust beim Test versucht man mittels diversen Techniken zu kompensieren. Zum einen existieren Ansätze dem BIST einen Phase Shifter, welcher eine höhere Effizienz

der Testvektorengenerierung erzeugen soll, ein zu setzen. Zum anderen ist ein Reseeding [AY04] des LFSR oder gar die Veränderung des Generatorpolynoms möglich. Hierdurch wird erreicht, dass der LFSR nicht seine durch das Polynom vorgegebene Sequenzenfolge durchläuft, sondern eine Variierung der Sequenz ermöglicht wird. Ein weiterer Ansatz sieht vor, dem LFSR ein „Scan-Feed-Register“ zu implementieren [G+03]. Auch dieses variiert die LFSR Zustandsfolge und erhöht somit die Testeffizienz. Diese Verfahren bringen jedoch den Nachteil mit sich, sollte ein Vektor mehrmals in nachfolgenden Pattern benötigt werden, ist dies mittels der genannten Technik nur schwer realisierbar. Hierfür erhält der BIST ein ROM, welches die zuvor berechneten oft benötigten Vektoren enthält und kann diese gezielt im Testpattern platzieren.

Der Einsatz eines Phase Shifters hingegen gestaltet sich wie in Abbildung 2.6 dargestellt durch die Implementierung des Phase Shifters zwischen dem Pattern Generator z.B. einem vorangestellten LFSR und dem Circuit Under Test (CUT).

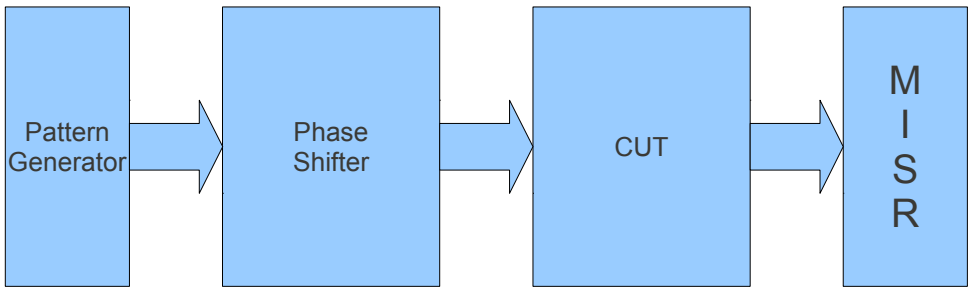

Abbildung 2.6: Phase Shifter

Der Aufbau eines Phase Shifter gestaltet sich durch den Einsatz von XOR-Gattern, welche die Ausgänge des LFSR mit dem CUT verbinden [Abbildung 2.7].

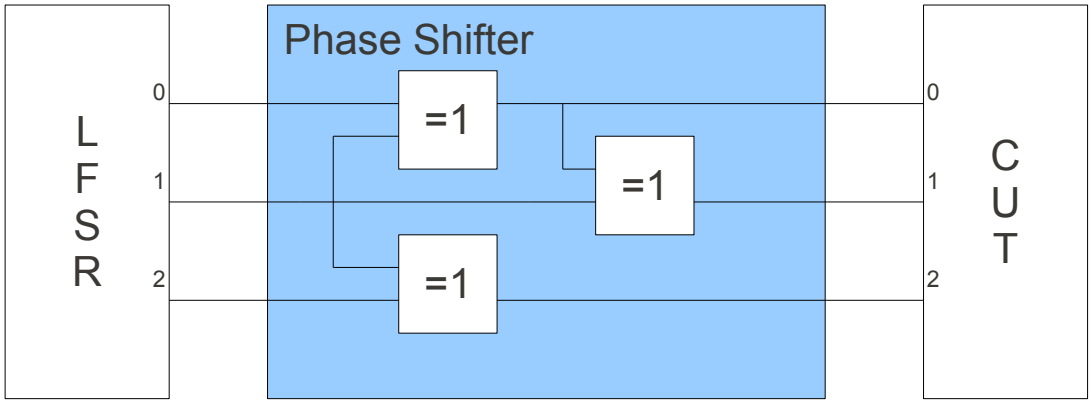

Abbildung 2.7: Phase Shifter Schaltung

Die Aufgabe des Phase Shifter ist es nun, den aktuellen LFSR Zustand gemäß der vom Phase Shifter dargestellten Transformationsvorschrift zu verändern. Die Abbildung 2.7 zeigt den typischen Aufbau eines Phase Shifters. Der in der Abbildung dargestellte Phase Shifter hat eine Bitbreite von 3 Bit. Als Transformationsvorschrift ist beim dargestellten Phase Shifter die nachfolgende Transformationsvorschrift realisiert:

$$x_0 = x_1 \oplus x_0$$

$$x_1 = x_1 \oplus x_1 \oplus x_0$$

$$x_2 = x_2 \oplus x_1$$

Im Anbetracht dieser Transformationsvorschrift liefert das im Unterkapitel LFSR [2.1] vorgestellte lineare Schieberegister mit seinen Zuständen wie in Tabelle 2.1 folgende in nachfolgenden Tabelle 2.2 ersichtlichen Zustände.

2.3 Status-RAM

Das Status-RAM hat in der vorliegenden Implementierung die Aufgabe die Adressierung des ROMs zu übernehmen. Hierbei wird das Status-RAM wie folgt betrieben. Das Status-RAM enthält 2^t Adressen. Jede Adresse bildet eine $t+1$ Bit große Adresse. Die ersten t Bits der Adresse werden genutzt um die Restrict Vektoren im nachfolgenden ROM aus zu wählen. Das *t+1* Bit wird benötigt um dem Multiplexer die Anweisung zu erteilen ob er die Ausgaben des Phase Shifters oder des ROMs an den CUT weiterleiten soll. Adressiert wird

LFSR Ausgabe	Phase Shifter Ausgabe
1 1 1	0 1 0
1 0 1	1 1 1
1 0 0	1 0 0
0 1 0	1 0 1
0 0 1	0 1 1
1 1 0	0 0 1
0 1 1	1 1 0

Tabelle 2.2: Transformation der LFSR Ausgabe per Phase Shifter

das Status-RAM selbst durch einen Modulo-t-Counter. Dieser erhöht pro Taktzyklus seinen Zustand. Dies hat zur Folge, dass das Status-RAM in aufsteigender Weise alle Speicherstellen adressiert bekommt. Der aktuelle Takt bestimmt somit aufgrund seines Einflusses auf die Speicherstelle des Status-RAMs welcher Adresse des ROMs selektiert wird. Beeinflusst werden kann die Adresse ROMs jedoch dadurch, dass zu jedem Taktzyklus die Möglichkeit besteht den Speicherinhalt des ROMs mittels dem Datenbus *a* zu verändern.

2.4 ROM

Ein BIST welcher seine Testvektoren mittels einem LFSR erzeugt, hat die Vorteile Pseudozufallszahlen Flächeneffizienz und mit kurzen Taktzyklen zu erzeugen. Jedoch können mittels einem LFSR erzeugten Pseudozufallszahlen nicht alle Fehler im CUT gefunden werden. Die erzeugten Zufallszahlen liefern hierzu zu wenig Informationen [WA99]. Um diese Einschränkung zu umgehen und sich nicht nur auf das Testen einer Teilmenge der möglichen Fehler beschränken zu müssen, welche durch die Pseudozufallszahlen abgedeckt werden, kommen Techniken zum Einsatz welche eine Erhöhung der Fehlerabdeckung bewirken. Zum einen ist es möglich durch Umgestaltung des CUT eine höhere Fehlererkennung zu bewirken. Des Weiteren kann ein Reseeding des LFSR Testvektoren ermöglichen um die sonst nicht möglichen Testmuster erzeugen zu können. Zum anderen wird durch den Einsatz eines ROMs die Fehlerabdeckung erhöht. Dies wird erreicht, indem das ROM Testvektoren speichert welche nicht durch das LFSR erzeugt werden können. Hierzu werden die hierfür benötigten Testvektoren im Voraus berechnet und im ROM gespeichert.

Ein ROM kann in einer BIST Architektur zum Einsatz falls der CUT mittels einem sogenannten *Restrict Vektor* [Abbildung 2.8] getestet werden muss. Hierbei handelt es sich um einen Vektor welcher mehrfach hintereinander in aufeinander folgenden Pattern für den

Test des CUT benötigt werden. Der Restrict Vektor wird hierzu im Vorfeld der Implementierung des BIST errechnet und im weiteren Verlauf statisch im ROM gespeichert. Sobald ein entsprechender Restrict Vektor für den Test benötigt wird, wird der entsprechende Vektor des ROMs selektiert und in den CUT eingegeben.

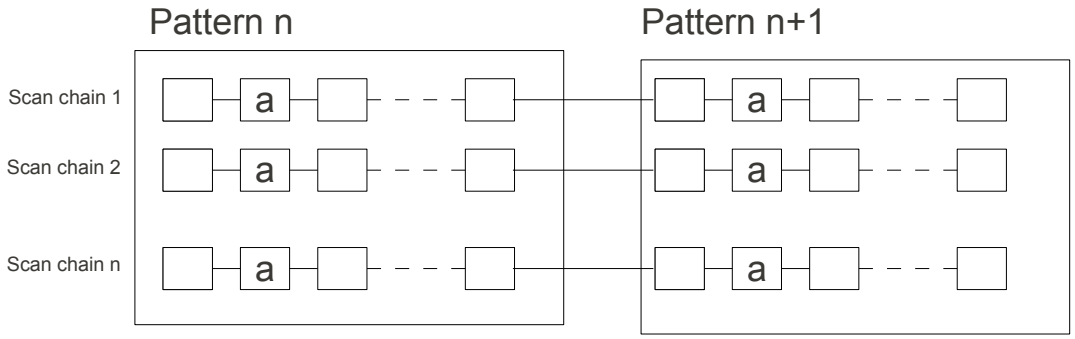

Abbildung 2.8: Restrict Vektor

2.5 MUX

Der Multiplexer dient in der vorliegenden BIST Architektur zur Selektion der Vektoren aus dem linear rückgekoppelten Schieberegister mit seinem nachgeschalteten Phase Shifter sowie den Vektoren aus dem ROM. Mittels dem Multiplexer wird es somit möglich, mit der BIST Architektur gemischte Testmuster in Form von Pseudozufallszahlen sowie deterministischen Testmustern für den Test zur Verfügung zu stellen. So wählt der Multiplexer die Pseudozufallszahlen aus dem vorangehenden Phase Shifter und die deterministischen Testmuster aus dem ROM.

Als Signal welchen der beiden Testmuster-Typen der MUX dem CUT zur Verfügung zu stellen hat entscheidet das im Status-RAM gespeicherte Auswahl-Bit für den Multiplexer.

2.6 MISR

In Build-in self-test Architekturen wird es nötig aufgrund des fehlenden externen Testgerätes eine Technik zur Auswertung der von der zu testenden Schaltung produzieren Vektoren in der Schaltung mit zu führen. Eine Technik stellt die Testvektorenauswertung mittels einem Multiple Input Signature Register (MISR) dar [Abbildung 2.9].

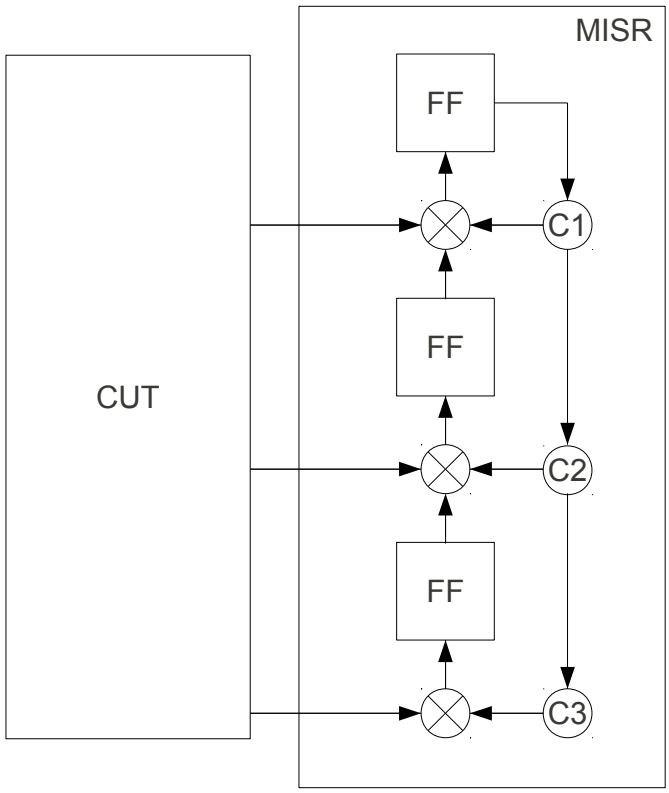

Abbildung 2.9: MISR Aufbau

Das MISR hat die Aufgabe die ihm vom CUT übergebenen Testvektoren zu einer kompakten Signatur zu komprimieren [L$^+$oo]. Ein Vorteil dieser Technik besteht darin, dass die gesamte Auswerteeinheit, welche aus den Vektoren aus dem CUT eine Signatur erzeugt, auf in der Schaltung integriert werden kann. Es entfallen somit umfangreiche Testgeräte. Ein Nachteil stellt allerdings die Tatsache dar, dass nur der Defekt des CUT nachgewiesen werden kann. Eine Lokalisierung des Defektes ist in der Regel nicht möglich, da die komprimierte Ausgabe der Testvektoren nicht mehr genügend Informationen für den Rückschluss des defekten Blockes aufweist [WA99].

Als problematisch stellt sich beim Einsatz eines MISR ebenfalls die Tatsache dar, dass er aufgrund der Kompression der Eingangsvektoren auf nur eine Ausgangssignatur nicht in der Lage sein wird alle Fehler welche im CUT auftreten feststellen zu können. Daraus ergibt sich die Notwendigkeit der Betrachtung wie effizient der eingesetzte MISR arbeitet. Unter der Annahme, ein Testlauf habe m Testpattern. Jedes Testpattern enthält n Testvektoren. Hieraus ergeben sich $n * m$ Eingangsvektoren für das MISR.

Diese $n * m$ Eingangsvektoren sind aufgrund ihrer booleschen Eigenschaften in der Lage 2^{n*m} boolesche Belegungen zu repräsentieren. Kommt nun ein n-Bit breites MISR zum Einsatz, so besitzt dieses MISR 2^n unterschiedliche Zustände. Als Resultat ergibt sich, dass die 2^{n*m} möglichen Eingangsbelegungen für das MISR auf 2^n MISR internen Zustände abgebildet werden müssen. Dies hat zur Folge, dass $\frac{2^{n*m}}{2^n}$ Eingangsmuster pro MISR Zustand repräsentiert werden. Schlussendlich resultiert daraus die Eigenschaft, dass $2^{m*n-n} - 1$ fehlerbehaftete Eingangsbelegungen auf die als richtig interpretierte MISR-Signatur abgebildet werden.

$$Fehlermaskierung = \frac{verdeckte\ Fehler}{Gesamtanzahl\ Fehler} = \frac{2^{m*n-n}-1}{2^{m*n}-1} \cong 2^{-n}$$

Als Beispiel sei ein 16 Bit breites MISR gewählt. Hieraus ergibt sich, folgende Berechnung:

$$Fehlermaskierung = \frac{verdeckte\ Fehler}{Gesamtanzahl\ Fehler} \cong 2^{-16}$$

Somik ist es möglich mittels einem 16 Bit breiten MISR eine Fehlermaskierung von ~99, 9998% zu erzielen.

3 Entwurf und Validierung

Im vorliegenden Kapitel werden die VHDL und Java Implementierungen des Built-in self-tests, welcher in der Abbildung 3.1 grafisch dargestellt ist, beschrieben. Realisiert wurden die Implementierungen in der Hardwarebeschreibungssprache VHDL und in der Hochsprache Java. Beiden Implementierungen liegen der gleichen Spezifikation zugrunde, sodass diese die selbe Funktion erbringen sollten. Um diese identische Funktion nachzuweisen, wurden beide Entwürfe umfangreich auf ihre übereinstimmenden Ausgaben geprüft. Eine Beschreibung der beiden Entwürfe ist im Kapitel 3.1 beschrieben. Der Aufbau und das Vorgehen der Validierung beider Implementierungen ist dem Unterkapitel 3.2 zu entnehmen.

3.1 Entwurf

3.1.1 VHDL

Die VHDL Implementierung gestaltet sich durch sieben Entities wie im Abbildung 3.1 grafisch dargestellt. Es handelt sich dabei um die Top Level Entität SR sowie sechs Entities welche die folgenden Komponenten funktional darstellen.

1. lineares Schieberegister (LFSR)
2. Phase Shifter (PS)
3. Status RAM
4. ROM
5. Multiplexer
6. Modulo-t-Counter

Ausgehend von der vorgegebenen Struktur des mixed-mode BIST, die von der vorliegenden Spezifikation in der Publikation [H+09] gefordert ist, wurden die Komponenten und deren Busbreiten implementiert. Die Spezifikation sieht die fünf Eingangssignale und Eingangsvektoren *a*, *d*, *wen*, *slct*, *slck* und *k* vor. Die erste Restriktion, welche der Spezifikation zu entnehmen ist, definiert den Datenbus *a*. Dieser Datenbus übernimmt zwei Funktionalitäten innerhalb des BIST. Zum einen dient er dazu, die Zustandsfolge des linearen Schieberegister

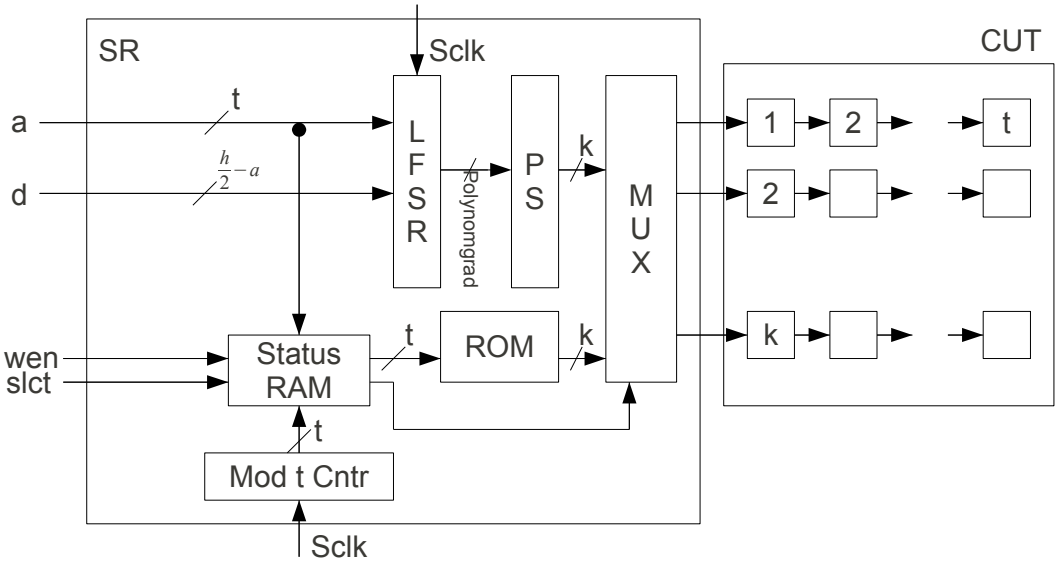

Abbildung 3.1: Struktur des VHDL Entwurfes

durch ein Reseeding beeinflussen zu können. Dies geschieht, wie in 2.1 beschrieben, indem er die im LFSR befindlichen XOR-Gatter mit externen Signalen beschaltet. Als weitere Aufgabe hat er die Funktion, das Status-RAM mit Informationen zu versorgen. Der Datenbus *a* bringt die Adressinformationen zusammen mit dem Signal *slct* in das Status-RAM ein. Durch die Busbreite des Datenbusses *a* ist bereits indirekt schon die Größe des ROMs vorgegeben. Dem RAM stehen pro Speicherstelle die Anzahl der Bits des Datenbusses *a* mit seinen *t* Bits, zusätzlich dem weiteren Bit *slct* zur Verfügung. Somit können im ROM 2^t Vektoren gespeichert werden. Das zusätzlich gespeicherte Bit *slct* dient im späteren Verlauf dem Multiplexer als Auswahlsignal, um zwischen der Ausgabe des Phase Shifters oder des ROMs zu selektieren. Weiterhin sieht die Spezifikation ein Write Enable Signal (*wen*) vor, durch das definiert werden kann zu welchem Taktzyklus das Status-RAM die Adresse für das ROM neu schreiben kann. Es kann somit das RAM nur verändert werden wenn das Write Enable Signal gesetzt wurde.

Als weiterer Parameter, welcher Teile der Implementierung betrifft, ist der Parameter *t* zu nennen. Der Parameter *t* stellt die Länge der Scan chains, also die Anzahl der Vektoren pro Pattern dar. Dieser Parameter wirkt sich unter anderem auf den Mod-t-Counter aus. Der Modulo-t-Counter hat in dieser Anwendung die Aufgabe, einen Zähler darzustellen. Mit jedem Taktsignal erhöht er seinen Zählerstand um einen Zustand und bildet damit Addressierung für das Status-RAM. Der Modulo-t-Counter mit seinem Parameter *t* stellt den Restklassenring $\mathbb{N}/2^t\mathbb{N}$ dar. Somit beginnt der Zähler bei Null und zählt anschließend in

aufsteigender Weise bis $2^t - 1$ um dann wieder bei Null zu beginnen. Durch Belegung dieses Parameters t ergibt sich zugleich eine Restriktion für das Status-RAM. Da die Busbreite t des Modulo-t-Counters als Addressierung der Speicherstellen im Status-RAM dient, sind somit im Status-RAM 2^t Speicherstellen adressierbar.

Durch die Belegung des Parameters t ist nun bekannt, dass 2^t Adressen für das ROM adressierbar sein müssen. Jede Adresse des Status-RAMs hat die Größe des t Bit breiten Datenbusses a sowie das Bit des Signales *wen*. Hieraus ergibt sich für das Status-RAM ein Array der Dimension $2^t * (t + 1)$.

Als weiterer Datenbus ist der Datenbus d vorgesehen. Dieser wird ausschließlich vom LFSR benötigt. Dieser Datenbus kommt dann zum Einsatz, wenn der Datenbus a nicht ausreicht um genügend Informationen durch den Datenbus a dem LFSR für das Reseeding zur Verfügung zu stellen. Der zusätzliche Datenbus d wird dann, wie auch schon der Datenbus a, über XOR-Gatter im LFSR verbunden.

Aufgrund der vielen assoziativen Abhängigkeiten gestaltet sich eine manuelle Anpassung der Parameter an die jeweiligen möglichen Bedingungen recht umfangreich. Es sind pro Parameteränderung mehrere hieraus resultierende Änderungen zu tätigen sein. Um diese Arbeit zu erleichtern und Fehler bei der Parametermodifikation zu vermeiden, wurden alle anzugebenden Parameter in die Datei *config.vhd* ausgelagert. Die einzelnen Komponenten der BIST Implementierung definieren hieraus ihre Busbreiten und Abhängigkeiten untereinander selbstständig.

Anzugebende Parameter:

1. Anzahl der Scan Chains
2. Anzahl der Vektoren pro Pattern
3. die Busbreite des Datenbus d
4. der Polynomgrad des Generatorpolynoms
5. Anzahl und Position der XOR Gatter für das Reseeding im LFSR

Neben den anzugebenden Parametern zur Bestimmung der Busbreiten wird im Anschluss noch das Editieren des LFSR-Polynoms sowie deren XOR-Gatter zum Reseeding nötig. Die Angabe des gewünschten LFSR Polynom ist in der Datei *lfsr.vhd* vor zu nehmen. Des Weiteren müssen dem Phase Shifter (*ps.vhd*) seine Ausprägung in Form einer Matrix implementiert werden. Dem Status-RAM sowie dem ROM (*ram.vhd* und *rom.vhd*) müssen ihre Initialisierungszustände mitgeteilt werden. Nach dem Anpassen dieser Parameter sowie der Angabe der Initialisierungszustände der Komponenten lässt sich die gesamte Architektur kompilieren.

3.1.2 Java Referenzmodell

Beim Java Referenzmodell handelt es sich um die Implementierung des BIST in der Hochsprache Java. Diese Implementierung wird nötig, um während der Validierung des in VHDL implementierten BIST eine Möglichkeit zu erhalten die VHDL Implementierung automatisiert zu testen. Hierzu werden während der Validierung die Ausgaben der VHDL sowie der Java Implementierung verglichen und beurteilt. Sollten beide Ausgaben nicht identische Werte liefern, so entspricht eine der beiden Implementierungen nicht der Spezifikation und weist somit Fehler auf.

Um eine Ähnlichkeit des Strukturierungen der beiden Implementierungen zu erlangen, wurde versucht, die Benennung der einzelnen Komponenten, so weit wie möglich identisch zu gestalten. Hierzu wurde die Hauptklasse der BIST Implementierung ebenfalls mit dem Namen *SR* versehen und findet ihre Implementierung in der Datei *sr.java* wieder. Die Java Implementierung erhält ihre Parameter weiterhin wie die VHDL Implementierung aus einer global gültigen Konfigurationsdatei. Diese trägt ebenfalls den Namen *conf* und ist in der Java-Datei *conf.java* zu finden. In dieser Konfigurationsdatei sind die selben Parameter wie in der VHDL Implementierung anzugeben. Des Weiteren ist es möglich, in dieser Datei global die Initialisierungszustände der im BIST enthaltenen Komponenten zu definieren. Diese globale Konfigurationsdatei erspart das Bearbeiten jeder einzelnen Komponente und erhöht die Übersichtlichkeit. Weiterhin ist es hier auch möglich neben der manuellen Definition der Initialisierungszustände eine Initialisierung mit Zufallszahlen zu wählen. Diese Möglichkeit kann durch die Einstellung in der Konfigurationsdatei oder über den Konsolenaufruf mit den Parametern erfolgen:

> *./partialseeding 'Anzahl der Scan Chains', 'Anzahl der Vektoren pro Pattern', 'Busbreite des Datenbusses d', 'Polynomgrad des Generatorpolynomes, 'Anzahl und Position der XOR Gatter'*

Durch diesen Aufruf erfolgt eine Zufallsinitialisierung der Komponenten gemäß den angegebenen Busbreiten. Nach dem Programmaufruf erstellt die Java Implementierung das Referenzmodell mit den angegebenen Parametern und führt anschließend die Berechnung durch. Zur Berechnung kommen dabei Eingangsvektoren, welche gezielt erstellt werden um gewünschte Testvektoren oder ganze Testsequenzen überprüfen zu können. Weiterhin können jedoch auch in der Konfigurationsdatei zufällige Eingangsvektoren erzeugt werden. Dies hat den Vorteil gegenüber der manuellen Testvektorenerstellung, dass das Modell durch viele zufällig erzeugte Vektoren auf seine Korrektheit hin untersucht werden kann.

3.2 Validierung

Das Ziel der Validierung eines Entwurfes ist es alle Fehler der Implementierung ausfindig zu machen, um diese vor dem Einsatz des Entwurfs beseitigen zu können. Das Ziel der Validierung besteht somit nicht darin mittels Durchführung von Tests zu zeigen, dass durch die gewählten Eingabeparameter keine Fehler mehr auftreten. Der Sinn der Validierung begründet sich vielmehr darin durch passende Auswahl der Testvektoren, möglichst viele Fehler im Entwurf ausfindig zu machen. Die manuelle oder automatische Testmustergenerierung zur Abdeckung aller auftretenden Fehler, erfolgt daher unter dem Aspekt der Betrachtung in welchen Teilen und welchen Zuständen die zu testende Software oder Hardware Fehler aufweisen könnte.

Um der Forderung einer möglichst hohen Testmusterabdeckung gerecht zu werden, müssen die in einer Hardwarebeschreibungssprache vorliegenden Entwürfe mit Hilfe möglichst vielen Testvektoren auf ihre Korrektheit hin untersucht werden. Hierzu ist es nötig, zu den Testvektoren der zu testenden Schaltung die zugehörigen Ausgabevektoren vorliegen zu haben. Problematisch gestaltet sich diese Forderung bereits bei einer geringen Anzahl an Testvektoren, da zu jedem Testvektor dessen Ausgangsvektor manuell berechnet werden muss. Dies gestaltet sich bei umfangreichen zu testenden Schaltungen bereits schwierig und zeitraubend. Die dargestellte Methode 3.2 stellt eine Möglichkeit zur effizienten, schnellen und automatischen Testmustergenerierung durch Erzeugung von zufälligen Testvektoren dar. Diese so erzeugten Testvektoren werden vom Device-under Test berechnet. Gleichzeitig werden sie von ein Referenzmodell welches die selbe Funktion wie das Device-under Test bereit hält ebenfalls berechnet. Nach Berechnung der Ausgabevektoren durch die zu testende Schaltung und das Referenzmodell werden beide Ausgaben verglichen und beurteilt ob Fehler in der Implementierung vorliegen. Das Referenzmodell sollte zur Vermeidung möglichst vieler Fehlermöglichkeiten während der Implementierung in einer Hochsprache und somit auf einer möglichst abstrakten Ebene erstellt werden.

3.2.1 Validierung der einzelnen Komponenten

Das konkrete Vorgehen zur Validierung des implementierten BIST gestaltete sich in mehreren Stufen. Noch vor dem Test aller Komponenten als Gesamtsystem wurden bereits während der Implementierungsphase des VHDL Entwurfes die einzelnen Komponenten getrennt von den anderen Komponenten getestet. Durch diese separaten Tests konnten frühzeitig Implementierungsfehler in den einzelnen Komponenten aufgedeckt werden. Vorteil dieser lokalen Tests ist, dass zum einen, bereits während der Implementierung der Komponente weitestgehend Fehler beseitigt werden können. Zum anderen gestaltet sich die Fehlerlokalisierung im gesamten System sich mitunter bei großen Systemen recht komplex. Aufgetretenen Fehler können nur schwer einer Komponente zuordnet. Weiterhin dienten diese lokalen Testbenches dazu, gefundene Fehler im Gesamtsystem welche während eines Testlaufs auftraten lokal

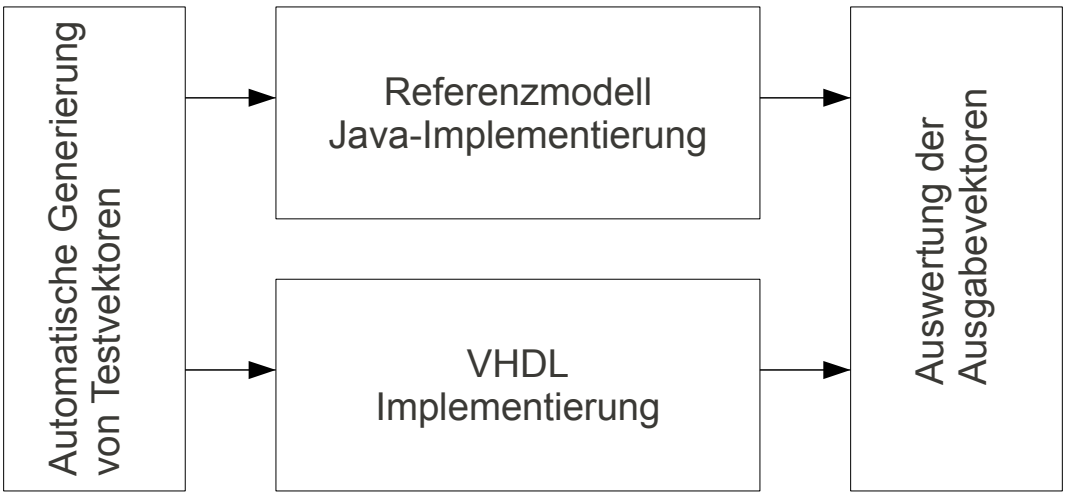

Abbildung 3.2: Aufbau einer Testanordnung mittels eines Referenzmodells

auf den fehlerhaft vermuteten Komponenten über eine begrenzte Anzahl an Testvektoren zu simulieren und somit schneller Fehler beseitigen zu können.

Aufgebaut waren die lokalen Testbenches wie in Abbildung 3.3 zu sehen. Die Testbench prüft die zu testende Komponenten, indem sie mögliche Eingangsvektoren der Komponente zur Bearbeitung übergibt. Nach Bearbeitung der Eingabe durch die zu testende Komponente nimmt die Testbench die Ausgabe entgegen und überprüft diese auf ihre Korrektheit. Geprüft wurden durch diese Vorgehensweise alle im BIST enthaltenen Komponenten mithilfe manuell erstellter Testvektoren und durch eine manuelle Berechnung der Ausgabevektoren. Auf dieser Ebene der lokalen Fehlerlokalisierung fand noch keine automatische Testmustergenerierung und Fehlerprüfung statt.

3.2.2 Validierung des Gesamtsystems

Durch den Einsatz von lokalen Testbenches wurden im VHDL Entwurf alle Komponenten schon während des Entwurfes auf ihre korrekte Funktionalität getestet. Als durch den Einsatz dieser lokalen Testbenches keine fehlerhafte Reaktion der einzelnen Komponenten mehr gefunden werden konnten, wurde die gesamte Implementierung auf ihre Korrektheit geprüft. Hierbei wurden alle Komponenten gemäß ihrer Abhängigkeiten untereinander verdrahtet und eine globale Testbench erstellt. Diese Testbench wurde zu Beginn noch mit manuellen Testvektoren versorgt. Ziel der manuellen Überprüfung war das Sicherstellen

Abbildung 3.3: Prinzipieller Aufbau einer Testbench

der korrekten Verdrahtung der Komponenten untereinander. Begonnen wurde hierzu mit einer von Hand erstellten globalen Testbench, welche die wichtigsten Testmuster, welche Fehler aufweisen könnten überprüfte. Weiterhin wurden anschließend ganze Sequenzen getestet bei denen angenommen wurde, dass die Implementierung zu einer fehlerhaften Ausgabe kommen könnte. Als auch durch diese manuelle Eingabe von Eingabemustern keine Fehler mehr festgestellt werden konnten, wurde mit der Realisierung eins automatisierten Testverfahrens begonnen. Diese Automatisierung bringt den Vorteil mit sich, dass enorm viele Testfälle und Testsequenzen automatisch generiert und getestet werden können. Auch solche hinter welchen man eigentlich keine Fehler vermutet oder Fehlermöglichkeiten welche man bei der manuellen Testmustererstellung übersehen hatte und somit keinem manuellen Test unterzogen hatte.

Nachdem auch dieser manuelle Gesamttest mit seiner nur begrenzten Anzahl an Testvektoren erfolgreich abgeschlossen war, wurde begonnen den Entwurf mittels vieler automatisch generierter Vektoren zu testen. Der verwendete automatisierte Test gestaltete sich wie in Abbildung 3.4 zu sehen durch den Einsatz eines weiteren Java Programms.

Dieses Java Programm hatte die Aufgabe nach dessen Aufruf im ersten Schritt eine BIST Architektur mit Zufallsparametern zu erstellen. Hierbei wurden die unten ersichtlichen Parameter frei gewählt.

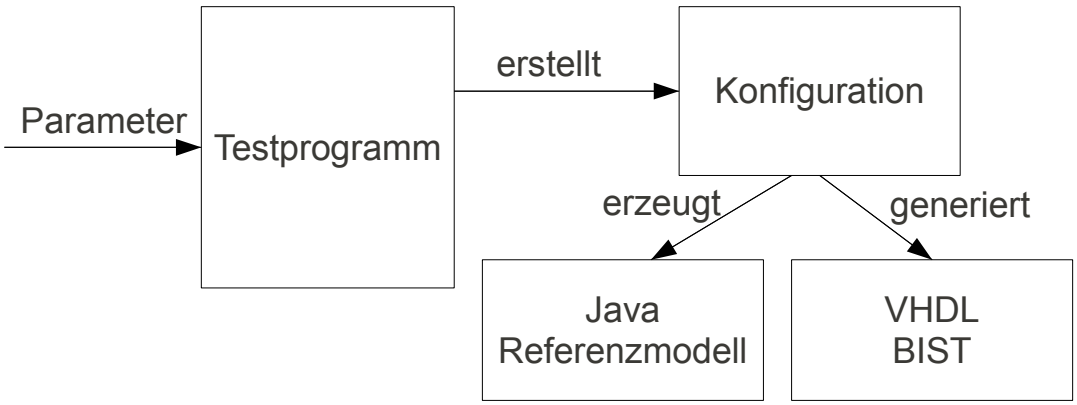

Abbildung 3.4: Aufbau des Testprogramms

1. Polynomgrad des Generatorpolynomes
2. Anzahl der Vektoren pro Pattern
3. Die zusätzliche Datenleitung *d*
4. Die Anzahl der von der Datenleitung *a* und *d* beschalteten XOR Gatter im LFSR
5. sowie die Größe des RAMs

Dies hatte zur Folge, dass im ersten Schritt eine Architektur entstand deren Busbreiten etc frei gewählt wurden. Im nächsten Schritt erstellte das Java Testprogramm die Initialisierungszustände der BIST Architektur. Hierbei wurden für das Status-RAM sowie das ROM ebenfalls eine Zufallsbelegung gewählt. Auch der Phase Shifter und seine Auswirkung auf die LFSR Ausgabe wurde hierbei zufällig festgelegt. Nachdem diese so zufällig erstellte Architektur in allen Punkten angelegt war, erfolgte eine Generierung des BIST in VHDL durch das Java Testprogramm. Abhängigkeiten wie die Busbreiten konnten hierbei über das Bearbeiten der global gültigen VHDL Datei *conf.vhd* erreicht werden. Komponenten wie der Phase Shifter, das Status-RAM und das ROM, mussten durch das Java Programm neu generiert werden. Dies hatte den Grund, dass die Komponenten wie beispielsweise das ROM ihre mit der Busbreite und dem Initialisierungszustand sich ändernden Bedingungen welche für jeden Test anfielen nicht aus einer Datei nach luden sondern direkt implementiert hatten.

Nach der Erstellung aller VHDL Dateien durch das Java Testprogramm, erstellte das Testprogramm die Java Implementierung durch Aufruf der entsprechend nötigen Klassen mit den global definierten Parametern wie sie auch schon zur Erstellung der VHDL Architektur

zu Einsatz gekommen waren. Im Anschluss daran begann das Testprogramm mit der Erstellung von zufällig gewählten Testvektoren. Diese wurden in einer Textdatei gespeichert, sodass beide BIST Implementierungen auf die gleichen Vektoren, bei ihrem Testlauf, zurückgreifen konnten. Im Anschluss an diesem Schritt der Testinitialisierung erfolgten noch das Einfügen der Restrict Vektoren in die Pattern. Hierzu wurde eine Liste angelegt welche Restrict Vektoren die Testsequenz in welche Patternfolge an welche Vektorposition im Pattern platzierte um diese nach Beendigung des Testlaufs wieder zu finden und deren Korrektheit feststellen zu können. Als einer der letzten Schritte des Testprogramms veranlasste dieses beide Implementierungen zur Berechnung mittels der vorgegebenen Testvektoren.

Als beide Implementierungen ihre Berechnungen abgeschlossen hatten, wurden beiden Ausgabedateien der beiden BIST Implementierungen mittels dem Unix-Programm *diff* auf Übereinstimmung untersucht. Sollten die beiden Ausgabedateien nun zu einem unterschiedlichen Ergebnis gekommen sein so war bekannt, dass ein Fehler in einer oder beiden Implementierungen vorlag. Sollte hiermit jedoch kein Fehler nachweisbar gewesen sein, so wurden noch durch das Java Testprogramm die beiden Ausgabedateien auf die korrekte Behandlung der Restrict Vektoren untersucht. Hierbei wurde getestet ob alle zuvor, vom Java Testprogramm, definierten Restrict Vektoren in korrekter Anzahl, in den richtigen Pattern und an richtiger Stelle zu finden waren.

Diese gesamte Testsequenz, wie im Zustandsgraphen in der Abbildung 3.5 zu sehen, wurde mehrfach wiederholt. Somit wurden mehrere Testsequenzen mit zufälligen Busbreiten, Generatorpolynomen, Testvektoren und sontigen zur Verfügung stehenden variablen Parametern voll automatisiert generiert und validiert. Letztendlich konnte keine Testsequenz mehr einen Fehler ausfindig machen. Hiermit ist anzunehmen, dass beide Implementierungen, die Java Implementierung wie auch die VHDL Implementierung die selben Berechnungen ausführen und der Spezifikation entsprechen.

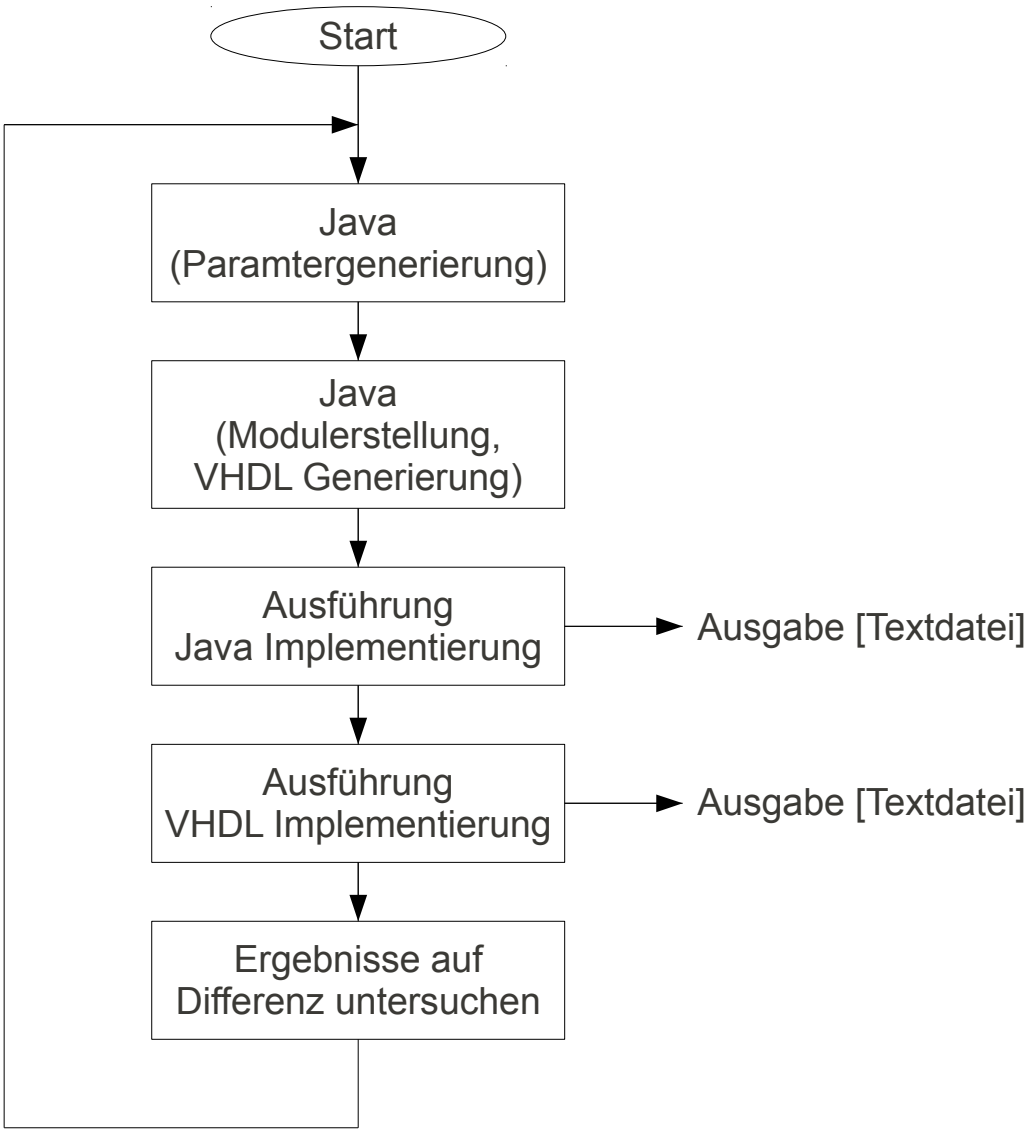

Abbildung 3.5: Vorgehen zur automatischen Validierung der Implementierungen

4 Experimentelle Ergebnisse

Im vorliegenden Kapiteln werden experimentelle Ergebnisse des in Abbildung 1.1 dargestellten und in VHDL implementierten BIST präsentiert. Der vorliegende BIST wird im folgenden auf seinen Flächenbedarf sowie den kritischen Pfad der synthetisieren Schaltung untersucht.

Um möglichst aussagekräftige Ergebnisse der experimentellen Auswertung zu erzielen, wurde entschieden die ROM Komponente nicht mit zu synthetisieren. Dies resultiert daraus, dass ein ROM nicht mittels Standardzellen implementiert wird. Somit wurde das ROM und dessen Größe nicht bei den Auswertungen im Bezug auf den benötigten Flächenbedarf sowie kritischen Pfad berücksichtigt. Der BIST wurde hierfür insofern verändert, dass das ROM während der Synthese aus der BIST Architektur entfernt wurde. Implementiert wurde jedoch eine Schnittstelle welche eine Verbindung des BIST mit einem externen ROM ermöglicht. Die Schnittstelle sah vor, die Adressleitung, für das ROM, welche ihren Ursprung im Status-RAM hat aus der BIST Architektur hinaus zu führen. Hinzu kam daraufhin ein Dateneingang zur BIST Architektur hinzu. Dieser Dateneingang stellt die Verbindung vom nun externen ROM zum BIST internen Multiplexer dar. Der Aufbau dieser bei der gesamten Synthese zum Einsatz kommenden leicht veränderten Architektur ist in der Abbildung 4.1 grafisch dargestellt.

Die Entscheidung für die Implementierung einer Schnittstelle begründete sich darin, dass nicht bekannt ist, wie das Synthese Tool reagiert, wenn das ROM entfernt wird und stattdessen die Verdrahtung des ROM nur mit statischen Logikpegeln definit wird. Es muss davon ausgegangen werden, dass das Synthese Tool ganze Bereiche der Architektur aufgrund der internen Belegungen mit den statischen Pegel optimieren würde. Hiermit wären wiederum keine aussagekräftigen experimentellen Ergebnisse möglich. Aus diesem Grunde entschied man sich für die Schnittstellenbildung, sodass die Optimierung des BIST Implementierung definitiv verhindert werden konnte.

Zur Synthese der Schaltung kommt während allen Synthesevorgängen ein kommerzielles Synthesewerkzeug zum Einsatz. Genutzt wird hierbei durchgängig die Standardzellenbibliothek *lsi_10k*. Weiterhin wird für alle Gatter ein maximaler Fan Out der Größe 4 festgelegt.

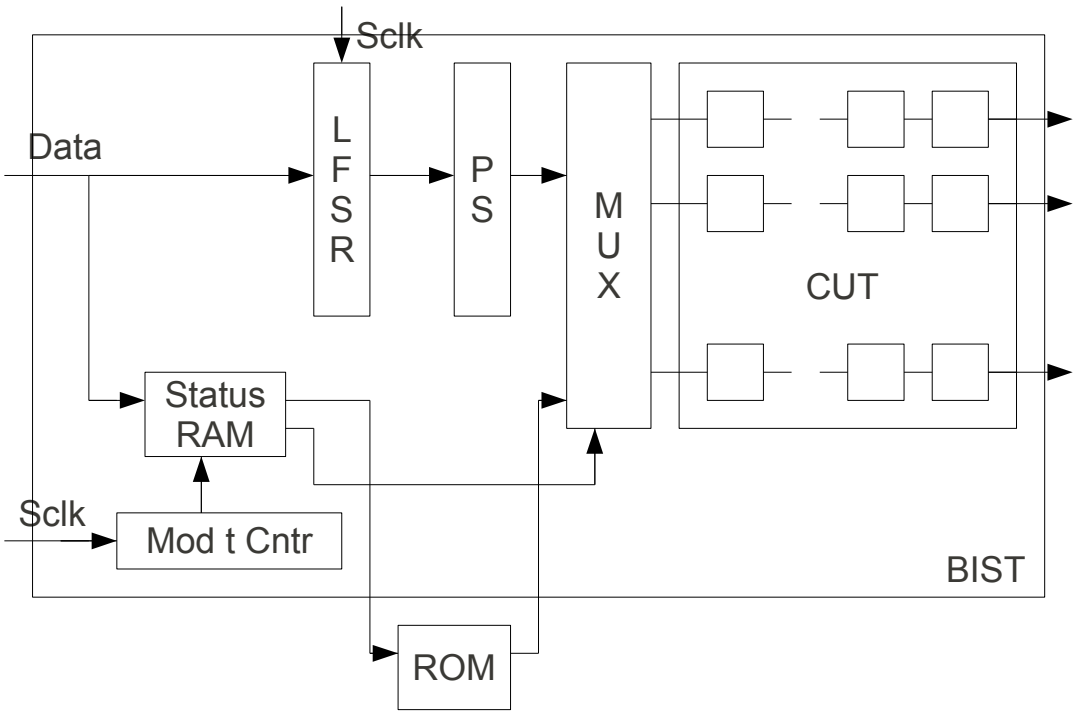

Abbildung 4.1: Synthese Aufbau

4.1 Flächenbedarf

In den nachfolgenden Unterkapitel erfolgt eine Flächenbedarfsanalyse. Hierzu werden alle einstellbaren Parameter der BIST Architektur auf ihre Auswirkung auf den Flächenbedarf untersucht. Variabel sind hierbei drei Parameter:

1. Anzahl der Scan chains
2. Anzahl der Vektoren pro Pattern
3. Der Polynomgrad des Generatorpolynomes für den LFSR

Aus diesen drei Parametern lassen sich alle assoziativ abhängigen Parameter wie Busbreiten, RAM-Größe etc der BIST Implementierung ableiten. Im folgenden wird detailliert auf die Auswirkung jedes einzelnen dieser genannten Parameter auf dessen Auswirkung auf den Flächenbedarf hin eingegangen. Ebenso werden Abhängigkeiten der einzelnen Parameter untereinander sowie deren Beeinflussung zueinander aufgezeigt.

Als kombiniertes Maß für den Flächenbedarf sowie die Anzahl der Elemente, kommt im gesamten Unterkapitel das sogenannte Gate-Count Verfahren zum Einsatz. Hierbei werden die Anzahl der Elemente sowie die Fläche welche die synthetisierte Schaltungen in Anspruch nehmen mit dem Flächenbedarf von NAND-Gattern verglichen. Als NAND Gatter, werden hierbei NAND Gatter mit zwei Eingängen, sogenannte NAND2 Gatter zum Größenvergleich herangezogen.

4.1.1 Anzahl der Scan chains

Architektur	Länge der Scan chains	Polynomgrad
2	4	32
3	8	32

Tabelle 4.1: Synthetisierte Architekturen - Flächenbedarf - Anzahl der Scan chains

Um die Abhängigkeit des Flächenbedarfs von der Anzahl der Scan chains zu untersuchen, werden zwei Architekturen [Tabelle 4.1] synthetisiert. Beide Architekturen haben einen Polynomgrad der Größe 32. Die erste Architektur hat hierbei vier Vektoren im Pattern, die zweite Architektur ist mit 8 Vektoren im Pattern ausgestattet. Variabel gestaltet sich nun die Anzahl der Scan chains. So wird für beide Architekturen eine Anzahl von 8, 16, 32, 64, 128, 512 und 1024 Scan chains gewählt. Die Ergebnisse des benötigten Flächenbedarfs sind in Tabelle 4.2 zu sehen.

Anzahl der Scan chains	Fläche Architektur 1	Fläche Architektur 2
8	1059	17990
16	1148	18083
32	1301	18230
64	1630	18568
128	2222	19148
512	6105	23041
1024	10473	27395

Tabelle 4.2: Ergebnis - Flächenbedarf - Anzahl der Scan chains

Diese tabellarisch dargestellten Ergebnisse sind im Schaubild 4.2 grafisch aufbereitet. Auf der x-Achse verdeutlicht die Grafik die Anzahl der synthetisierten Scan chains. Auf der y-Achse wird der benötigte Flächenbedarf aufgetragen. Zu erkennen ist, dass der benötigte Flächenbedarf bis zu einer Anzahl von 128 Scan chains nur eine geringe Flächenzunahme verursacht. Mit größer werdender Anzahl an Scan chains steigt der Flächenbedarf jedoch

stark an. Die Ursache für den starken Anstieg des Flächenbedarf liegt in der Tatsache, dass der Phase Shifter bei einer größer werdenen Anzahl von Scan chains starkt zunimmt. Der Phase Shifter beinhaltet bei einer Anzahl von n Scan chains mit einer Länge von t Bits eine Fläche von $n * t$.

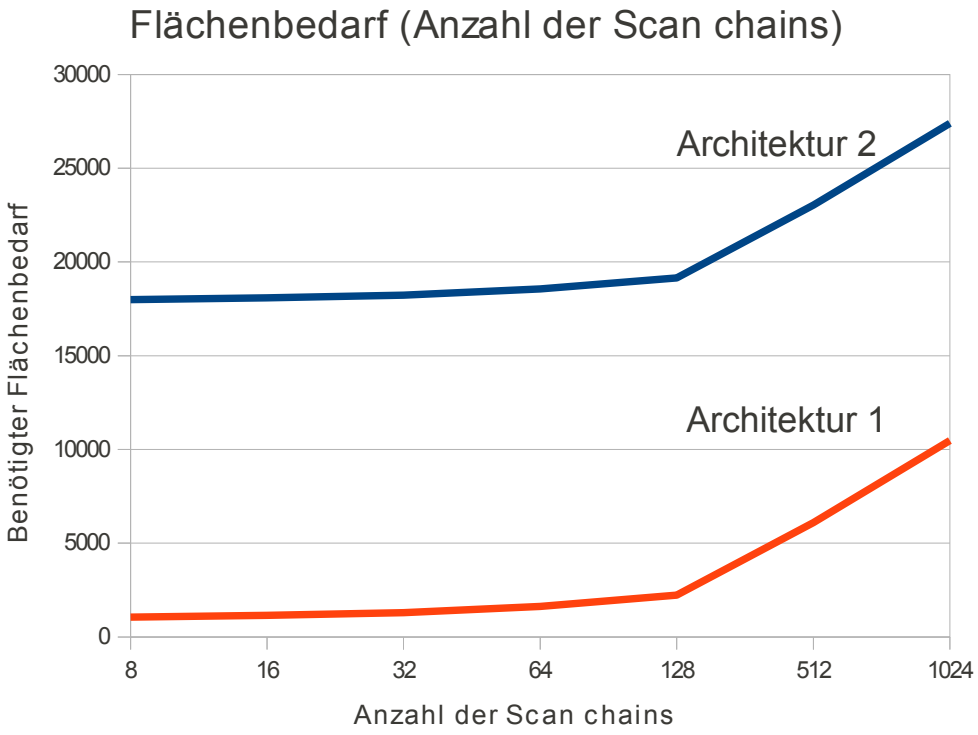

Abbildung 4.2: Flächenbedarf - Scan chains

4.1.2 Länge der Scan chains

Die Länge der Scan chains bildet eine weitere tragende Größe im Bezug auf den benötigten Flächenbedarf. Die Ursache hierin liegt in der Tatsache begründet, dass aufgrund der Implementierung die Länge der Scan chains und somit die Anzahl der Vektoren pro Pattern die Größe des Status-RAM bestimmen. Ist bei der Implementierung eine Länge Scan chain Länge mit t Bits gewählt, so sind t^2 Adressen im RAM speicherbar. Dieser Zusammenhang der Länge der Scan chains und die Größe des RAMs wirken sich quadratisch auf den benötigten

Flächenbedarf aus. Um diesen Sachverhalt zu zeigen, wird der BIST in vier Varianten synthetisiert. Die Tabelle 4.3 zeigt die vier synthetisierten Architekturen zusammenfassend. Die erste Architektur umfasst im Detail 64 Scan chains und hat den Polynomgrad 32. Die zweite Architektur ist mit 128 Scan chains versehen und umfasst ebenfalls einen Generatorpolynom vom Grad 32. Die dritte hat eine Anzahl von 512 Scan chains beim gleichen Polynomgrad. Die vierte Architektur ist ebenfalls mit 512 Scan chains versehen, wird jedoch mit einem Polynomgrad von 16 synthetisiert.

Architektur	Anzahl der Scan chains	Polynomgrad
1	64	32
2	128	32
3	512	32
4	128	16

Tabelle 4.3: Synthetisierte Architekturen - Flächenbedarf - Länge der Scan chains

Anzahl der Vektoren	Architektur 1	Architektur 2	Architektur 3	Architektur 4
1	941	1534	5421	1268
2	1032	1618	5509	1335
3	1220	1816	5692	1351
4	1630	2222	6105	1940
5	2256	2899	6733	2706
6	4242	4880	8716	4689
7	9004	9589	13471	9315
8	18567	19147	23040	18877
9	40061	40642	44521	40372
10	85547	86128	90014	85862
11	182970	183551	187437	183276

Tabelle 4.4: Ergebnis - Flächenbedarf - Länge der Scan chains

Synthetisiert werden diese vier Architekturen mit ihren unterschiedlichen Parametern auf die Länge der Scan chains. Hierbei werden die Längen und somit die Anzahl der Vektoren pro Pattern gemessen. Gewählt werden in allen vier Synthesen die Patterngrößen von 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 und 11 Vektoren pro Pattern. Die Tabelle 4.4 zeigt die Ergebnisse der Synthese zusammenfassend. Aus der Tabelle ist ersichtlich, welche Architektur bei der Synthese welche Anzahl an Zellen benötig. Hierbei ist in der ersten Spalte die Anzahl der Vektoren pro Pattern angegeben. In den weiteren Spalten ist zu sehen, welche Anzahl an Zellen bei der Synthese durch die unterschiedlichen Architekturen benötigt werden. So

benötigt beispielsweise die Synthese der Architektur 2 bei einer Anzahl von 7 Vektoren im Pattern und daraus resultierenden 7^2 möglichen RAM Einträgen einen Flächenbedarf von 9589 NAND2 Gattern bei der Synthese. Die Untersuchung aller Synthesen zeigt, dass bei allen Implementierungen der Flächenbedarf quadratissch, mit der Zunahme der Vektoren pro Pattern, wächst. Dieser Sachverhalt verdeutlicht die Grafik 4.3.

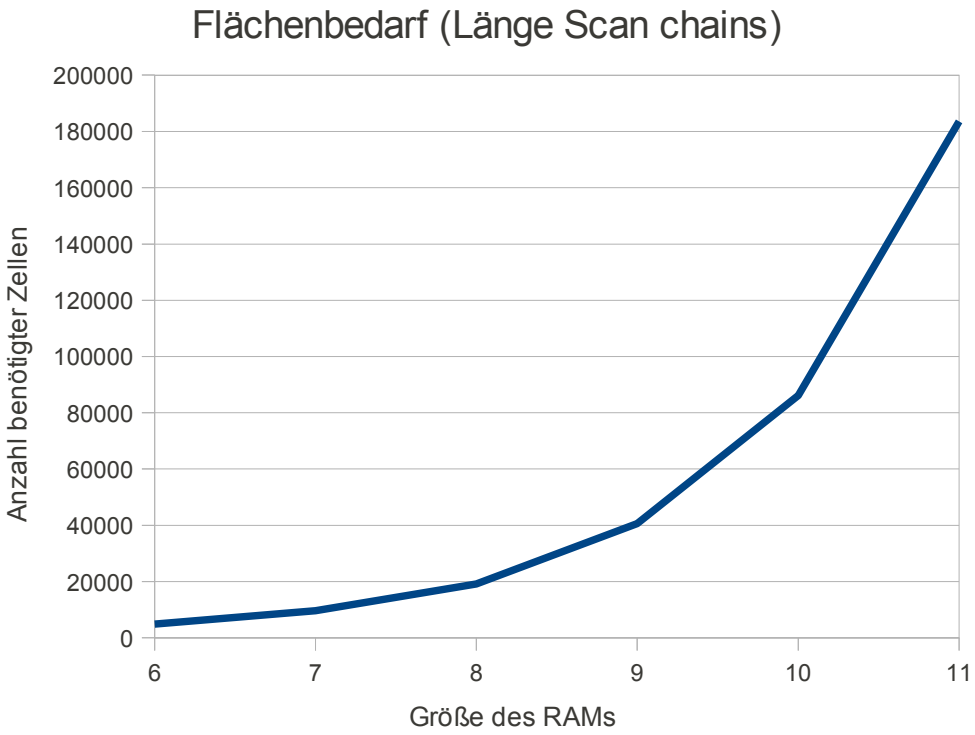

Abbildung 4.3: Flächenbedarf - Anzahl Vektoren pro Pattern

Das Diagramm 4.3 zeigt somit grafisch die texuell in Tabelle 4.4 dargestellten Ergebnisse. In der Grafik ist jedoch aufgrund der gewählten Auflösung nur eine Linie zu sehen. Die Linien der anderen Architekturen bilden aufgrund der Schrittweite der x und y-Achse eine identische Linie. Somit liegen alle 4 Linien übereinander.

Nummer	Anzahl der Scan chains	Länge der Scan chains
1	64	4
2	128	4
3	512	4

Tabelle 4.5: Synthetisierte Architekturen - Flächenbedarf - Generatorpolynom

4.1.3 Grad des Generatorpolynoms

Bei der Untersuchung des Flächenbedarfs für den Grad des Generatorpolynoms werden drei Architekturen ausgewählt. Diese drei Architekturen sind in Tabelle 4.5 zusammenfassend aufbereitet. Alle drei Architekturen weisen die selben Parameter auf. Lediglich die Anzahl der Scan chains unterscheiden sich, um die Vermutung nach zu beweisen, dass sich der Grad des Generatorpolynomes linear auf den Flächenbedarf auswirkt. Hierfür wird die erste Architektur mit 64 Scan chains, die zweite mit 128 und die dritte Architektur mit 512 Scan chains versehen. Für alle drei Architekturen wird im Anschluss der Polynomgrad variiert und der benötigte Flächenbedarf festgehalten. So werden alle drei Architekturen mit dem Polynomgrad 4, 8, 16, 32, 64 und 128 synthetisiert.

Die Ergebnisse der Synthesen verdeutlicht die Tabelle 4.6. Die Tabelle zeigt für jede der drei Architekturen den benötigten Flächenbedarf für die gewählt Anzahl an Vektoren pro Pattern. Die Anzahl der Vektoren pro Pattern ist hierbei in der ersten Spalte angegeben. Die Größe der benötigten Fläche ist den Spalten Architektur 1 bis 3 zu entnehmen.

Polynomgrad	Architektur 1	Architektur 2	Architektur 3
4	935	1097	2067
8	1193	1512	2908
16	1399	1940	4906
32	1630	2222	6105
64	1900	2688	7183
128	2442	3244	8609

Tabelle 4.6: Ergebnis - Flächenbedarf - Generatorpolynom

Die Ergebnisse der Synthese, welche in Tabelle 4.6 dargestellt sind, werden in der Abbildung 4.4 grafisch präsentiert. Die Abbildung zeigt deutlich, dass der Flächenzuwachs bei verändertem Grad des Generatorpolynoms linear ansteigt. Diese Syntheseergebnisse entsprechen dem erwarteten Verhalten des Flächenbedarf eines LFSR. Da sich die Struktur eines LFSR mit dem Polynomgrad n aus der Anzahl von n-XOR-Gattern und n-FlipFlops

zusammensetzt, sind $2n$ Zellen pro Polynomgrad zu erwarten. $2n$ repräsentiert eine lineare Funktion. Diese theoretische Annahme wird durch die Grafik bewiesen.

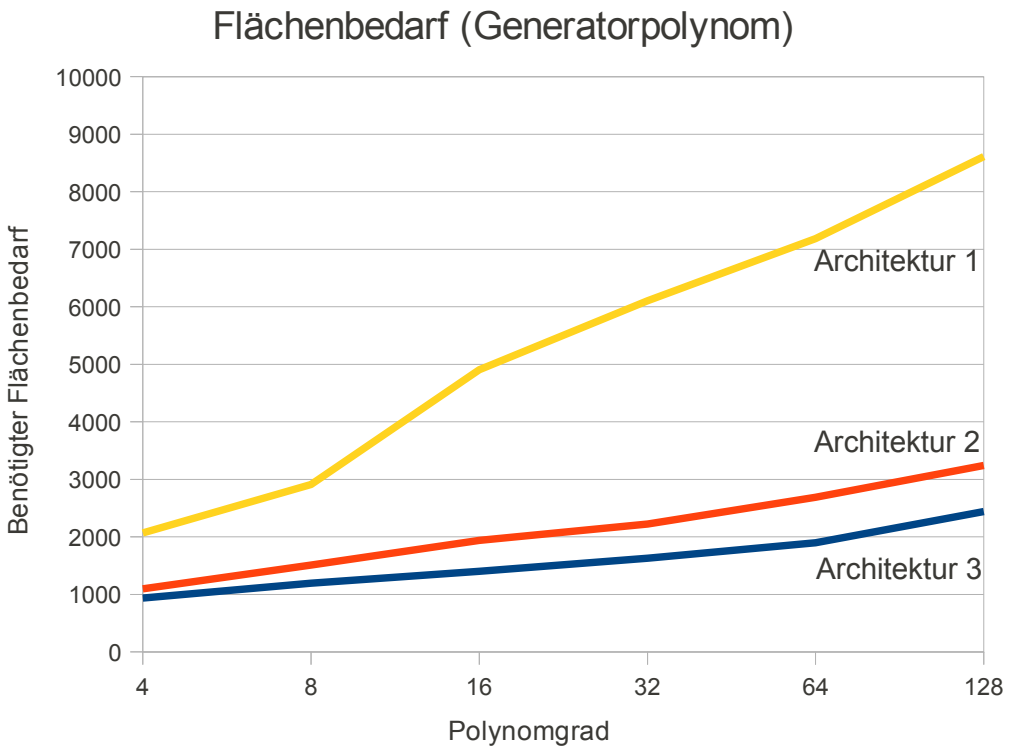

Abbildung 4.4: Flächenbedarf Generatorpolynom

4.2 Anzahl benötigter Zellen

Nach der Betrachtung des benötigten Flächenbedarfs im vorangegangenen Kapitel erfolgt nun die Betrachtung wie viele Zellen die variablen Schaltungen nach ihrer Synthese in Anspruch nehmen. Hierzu wird erneut auf die drei Parameter "Anzahl der Scan chains", "Anzahl der Vektoren pro Pattern" sowie auf den Parameter "Polynomgrad des Generatorpolynomes" eingegangen.

4.2.1 Anzahl der Scan chains

Zu Beginn erfolgt die Auswertung des Zellenbedarfs für den Parameter "Anzahl der Zellen". Hierfür werden zwei Architekturen, dargestellt in Tabelle 4.7, synthetisiert und ihren Zellenbedarf notiert. Gewählt werden für beide Architekturen ein Polynomgrad der Größe 32. Unterschiedliche Parameter weisen die beiden Architekturen in der Länge der Scan chains auf. So wird die erste Architektur mit vier Vektoren pro Pattern, die zweite mit acht Vektoren pro Pattern synthetisiert. Für beide Architekturen wurde der Zellenbedarf mit einer Anzahl von 8, 16, 32, 64, 128 und 512 Scan chains pro Architektur untersucht.

Architektur	Länge der Scan chains	Polynomgrad
1	4	32
2	8	32

Tabelle 4.7: Synthetisierte Architekturen - Anzahl benötigter Zellen - Anzahl der Scan chains

Anzahl der Scan chains	Zellenbedarf Architektur 1	Zellenbedarf Architektur 2
8	368	127
16	422	182
32	527	282
64	751	500
128	1133	877
512	2106	1867

Tabelle 4.8: Ergebnis - Anzahl benötigter Zellen - Anzahl der Scan chains

Das Ergebnis [Tabelle 4.8, Diagramm 4.5] dieser Synthese zeigt deutlich, dass für beide synthetisierten Architekturen der Zellenbedarf mit der Zunahme an Scan chains kontinuierlich ansteigt. Dieser Zellenbedarf ist damit zu begründen, dass zum einen der Multiplexer vergrößert und der Phase Shifter an die größer werdenden Busbreiten angepasst werden

muss. Der Phase Shifter vergrößert seinen Flächenbedarf um den Faktor $t * n$ wobei t die Länge der Scan chains und n die Anzahl der Scan chains darstellt.

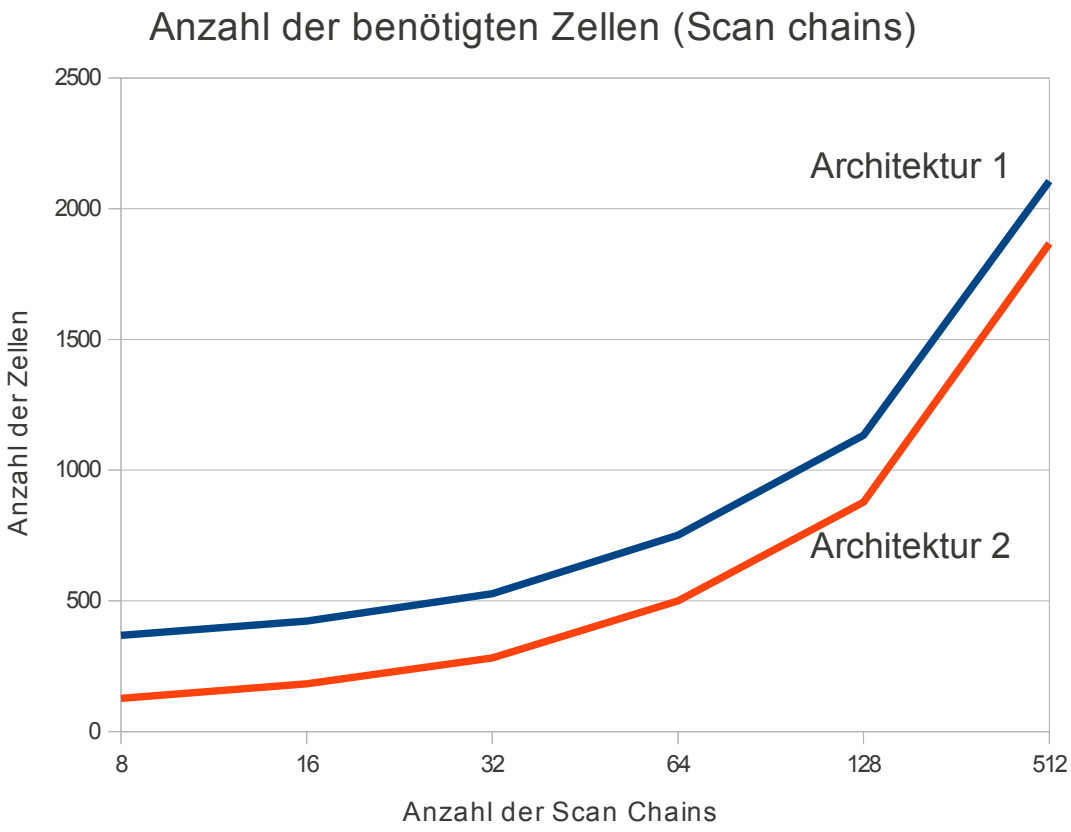

Abbildung 4.5: Anzahl Zellen - Anzahl der Scan chains

4.2.2 Länge der Scan chains

Zur Auswertung des Zellenbedarfs welche der Parameter "Länge der Scan chains" auf den Anzahl der benötigten Zellen der synthetisierten Schaltung hat, werden vier Architekturen erstellt. Zusammenfassend stellt die Tabelle 4.9 die gewählten Architekturen dar. Die erste Architektur, wie sie der Tabelle zu entnehmen ist, hat eine Anzahl von 64 Scan chains und ein Polynomgrad von 32. Die zweite Architektur hat 128 Scan chains und wie die erste Architektur ebenfalls einen Polynomgrad der Größe 32. Die dritte Architektur ist ausgestattet mit 512 Scan chains und einem Polynomgrad von ebenfalls einer Größe von 32. Die vierte Architektur wurde mit 128 Scan chains versehen. Besitzt aber im Gegensatz zu den drei vorangegangenen Architekturen ein Polynomgrad von 16. Diese Eigenschaft der vierten Architektur wird gewählt, um zu zeigen, dass der Zellenbedarf sich für die Anzahl aller Scan chains sowie diversen Polynomgraden analog verhält.

Architektur	Anzahl der Scan chains	Polynomgrad
1	64	32
2	128	32
3	512	32
4	128	16

Tabelle 4.9: Synthetisierte Architekturen - Anzahl benötigter Zellen - Länge der Scan chains

Anzahl der Vektoren	Architektur 1	Architektur 2	Architektur 3	Architektur 4
1	447	865	1839	763
2	480	898	1873	786
3	547	974	1937	837
4	721	1155	2106	1006
5	1041	1464	2380	1347
6	1774	2203	3117	2086
7	495	877	1862	777
8	500	876	1868	779
9	506	881	1873	782
10	509	890	1879	790
11	515	885	1882	790

Tabelle 4.10: Ergebnis - Anzahl benötigter Zellen - Länge der Scan chains

Nach der Synthese [Tabelle 4.10] aller gewählten Architekturen zeigt sich der in der Grafik 4.6 dargestellte Zellenbedarf für die einzelnen Architekturen.

Zu sehen ist deutlich, dass für alle gewählten Architekturen ein starker Zuwachs an Zellen im Bereich einer bis sechs Scan chains stattfindet. Im Nachfolgenden fällt dieser Zellenbedarf stark ab und bleibt ab einer Anzahl von sieben Scan chains nur noch minimal zu. Er bleibt fast schon konstant.

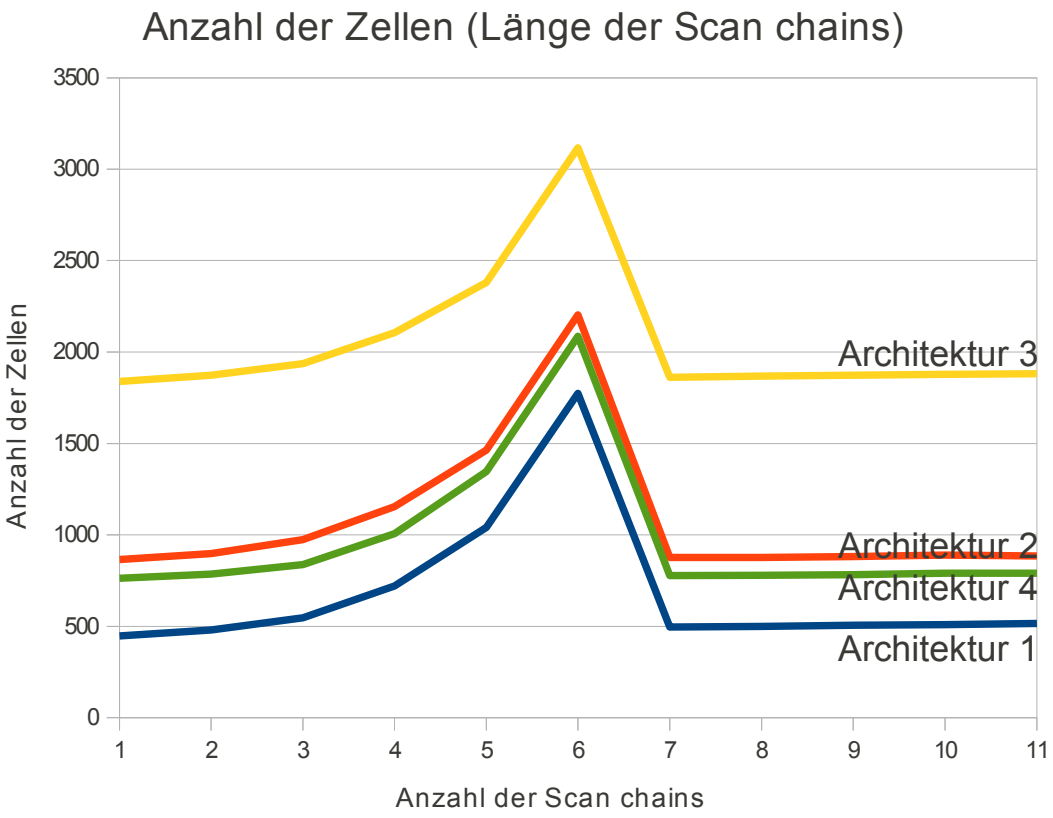

Abbildung 4.6: Anzahl Zellen - Länge der Scan chains

4.2.3 Grad des Generatorpolynoms

Zur Analyse der Anzahl der benötigten Zellen kommen bei der Betrachtung der Auswirkungen des Polynomgrads die in der Tabelle 4.11 beschriebenen Architekturen mit dem dargestellten Parametern zum Einsatz. Alle drei Architekturen haben eine Scan chain Länge der Größe 4. Die Anzahl der Scan chains wird unterschiedlich gewählt. So wird die erste der drei Architekturen mit einer Anzahl von 64 Scan chains ausgestattet. Die zweite Architektur erhält 128 Scan chains. Die dritte und letzte synthetisierte Architektur ist mit einer Anzahl von 512 Scan chains ausgestattet.

Zur Analyse der Anzahl der benötigten Zellen werden nun der Grad des Generatorpolynomes variiert. Hierbei werden die Polynomgrade 4, 8, 16, 32, 64, 128, 512 und 1024 gewählt. Die Ergebnisse der einzelnen Synthesen sind in der Tabelle 4.12 veranschaulicht.

Nummer	Anzahl der Scan chains	Länge der Scan chains
1	64	4
2	128	4
3	512	4

Tabelle 4.11: Synthetisierte Architekturen - Anzahl benötigter Zellen - Generatorpolynom

Polynomgrad	Architektur 1	Architektur 2	Architektur 3
4	451	613	1582
8	578	832	1739
16	649	1006	2071
32	751	1133	2106
64	782	1293	2150
128	891	1375	2207
512	1309	1803	2713
1024	1824	2315	3274

Tabelle 4.12: Ergebnis - Anzahl benötigter Zellen - Generatorpolynom

Die Betrachtung der Syntheseergebnisse [Grafik 4.7] des Zellenbedarfs zeigt mit wachsendem Grad des Generatorpolynoms eine konstante Zellenzunahme. Erklärt werden kann dies dadurch, dass bei der Implementierung nur der Parameter des Polynomgrads letztendlich eine Auswirkung auf die Größe des LFSR hat. Keine Komponente des implementierten BIST weist Abhängigkeiten vom Polynomgrad ab. Dies hat zur Folge, dass letztendlich nur das LFSR seinen Zellenbedarf an den Parameter zur Festlegung des Polynomgrads anpassen muss.

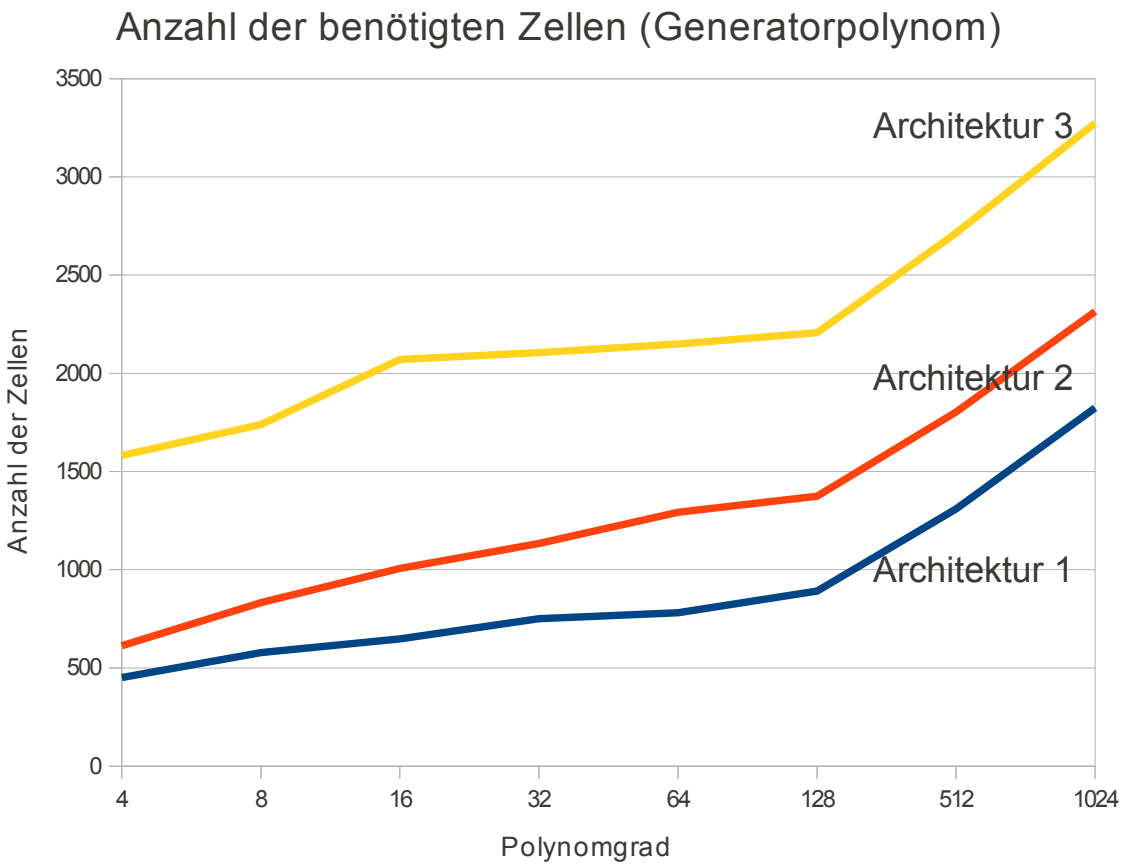

Abbildung 4.7: Anzahl Zellen - Generatorpolynom

4.3 Signallaufzeit

Neben der Betrachtung des Flächenbedarfs der Implementierung stand zusätzlich noch die Betrachtung der Signallaufzeiten an. Hierbei ist der kritische Pfad von Interesse. Betrachtet werden hierfür erneut synthetisierte Schaltungen welche im Bezug auf die Anzahl der Scan chains, die Länge der Scan chains sowie den Grad des Generatorpolynomes hin untersucht werden.

4.3.1 Anzahl der Scan chains

Zur Untersuchung der Abhängigkeiten der Anzahl der Scan chains und deren Signallaufzeiten mit ihrem kritischen Pfad werden drei Architekturen [Tabelle 4.13] entworfen und synthetisiert. Alle drei werden mit der Anzahl der Scan chains von 8, 16, 32, 64, 128, 512 und 1024 Scan chains synthetisiert und deren kritische Pfade in Tabelle 4.14 notiert. Alle drei Architekturen haben die Gemeinsamkeit eine Länge von vier Vektoren pro Pattern implementiert zu haben. Unterschiedlich gestaltet sich der Polynomgrad der Architekturen. Die erste Architektur wird mit einem Polynomgrad der Größe 16 synthetisiert. Für die zweite Architektur wird ein Polynomgrad der Größe 32 gewählt. Die dritte Architektur erhält einen Polynomgrad von 64.

Architektur	Länge der Scan chains	Polynomgrad
1	4	16
2	4	32
3	4	64

Tabelle 4.13: Synthetisierte Architekturen - Signallaufzeit - Anzahl der Scan chains

Anzahl der Scan chains	Architektur 1	Architektur 2	Architektur 3
8	27.26 ns	28.47 ns	24.68 ns
16	31.19 ns	30.01 ns	26.23 ns
32	42.18 ns	28.13 ns	30.78 ns
64	32.91 ns	35.45 ns	39.38 ns
128	37.50 ns	41.09 ns	40.95 ns
512	51.61 ns	40.45 ns	40.68 ns
1024	55.81 ns	44.38 ns	46.14 ns

Tabelle 4.14: Ergebnis - Signallaufzeit - Anzahl der Scan chains

Die Synthese der gewählten Architekturen mit ihren kritischen Pfaden macht deutlich, dass der kritische Pfad mit der Zunahme an Scan chains steigt. So werden bei der Synthese kritische Pfade von 25ns bis 55ns festgestellt [Grafik 4.8].

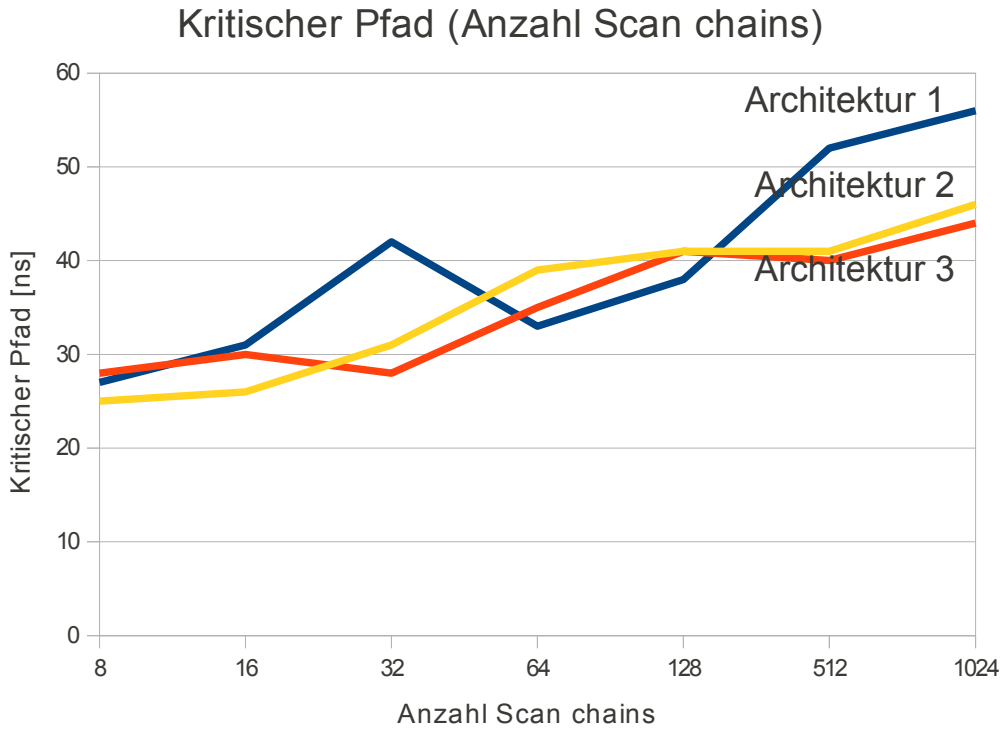

Abbildung 4.8: Kritischer Pfad Generatorpolynom

4.3.2 Länge der Scan chains

Zur Bestimmung der kritischen Pfade in Abhängigkeit der Länge der Scan chains sowie der Anzahl der Scan chains und dem Polynomgrad werden drei Architekturen [Tabelle 4.15] zur Synthese herangezogen. Alle drei Architekturen weisen einen Polynomgrad der Größe 32 auf. Die Anzahl der Scan chains unterscheiden sich in den einzelnen Architekturen. So wird die erste Architektur mit 64 Scan chains, die zweite Architektur mit 128 Scan chains und die dritte Architektur mit 512 Scan chains synthetisiert. Die Ergebnisse der Signallaufzeiten sind der Tabelle 4.16 zu entnehmen. Die Darstellung 4.9 stellt hierbei die Syntheseergebnisse grafisch dar.

Architektur	Anzahl der Scan chains	Polynomgrad
1	64	32
2	128	32
3	512	32

Tabelle 4.15: Synthetisierte Architekturen - Signallaufzeit - Länge der Scan chains

Länge der Scan chains	Architektur 1	Architektur 2	Architektur 3
1	25.86 ns	32.80 ns	39.44 ns
2	30.59 ns	31.81 ns	38.90 ns
3	31.25 ns	35.13 ns	38.25 ns
4	35.45 ns	41.09 ns	40.45 ns
5	43.81 ns	47.30 ns	51.55 ns
6	40.44 ns	44.11 ns	46.44 ns
7	46.70 ns	53.31 ns	59.32 ns
8	50.61 ns	51.84 ns	53.38 ns
9	56.43 ns	62.91 ns	67.16 ns
10	62.67 ns	66.35 ns	71.25 ns
11	59.95 ns	62.38 ns	75.33 ns
12	66.95 ns	69.81 ns	83.95 ns

Tabelle 4.16: Ergebnis - Signallaufzeit - Länge der Scan chains

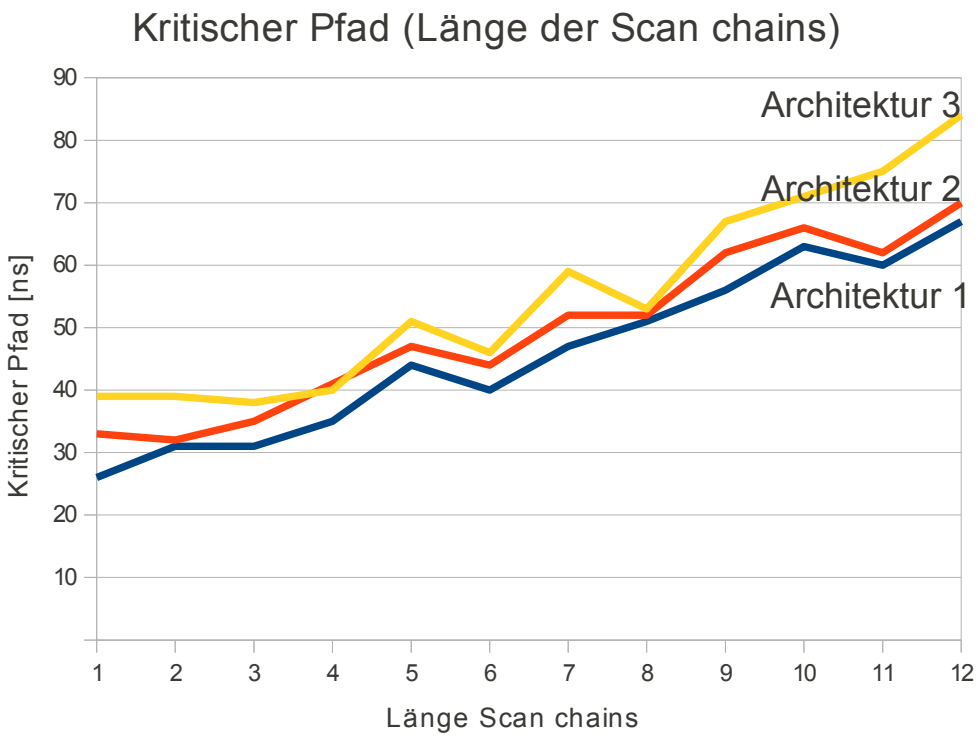

Abbildung 4.9: Kritischer Pfad Generatorpolynom

4.3.3 Grad des Generatorpolynoms

Architektur	Anzahl der Scan chains	Länge der Scan chains
1	64	4
2	128	4
3	512	4

Tabelle 4.17: Synthetisierte Architekturen - Signallaufzeit - Generatorpolynom

Nachfolgend erfolgt eine Betrachtung des kritischen Pfades unter der Veränderung des Grades des Generatorpolynomes. Hierzu werden drei Architekturen, wie in Tabelle 4.17 dargestellt, gewählt. Die erste Architektur enthält 64 Scan chains, die zweite Architektur 128 Scan chains und die dritte Architektur 512 Scan chains. Alle drei Architekturen enthalten des weiteren vier Vektoren pro Pattern. Synthetisiert werden diese drei Architekturen mit den Polynomgraden 4, 8, 16, 32, 64, 128, 512 und 1024. Die Ergebnisse der kritischen Pfade für jede dieser synthetisierten Schaltungen werden in der Tabelle 4.18 dargestellt. Diese Ergebnisse werden in der Darstellung 4.10 grafisch zusammengefasst.

Polynomgrad	Architektur 1	Architektur 2	Architektur 3
4	30.16 ns	33.22 ns	38.53 ns
8	53.08 ns	52.37 ns	59.09 ns
16	32.19 ns	37.50 ns	51.61 ns
32	35.45 ns	41.09 ns	40.45 ns
64	39.38 ns	40.95 ns	40.68 ns
128	37.5 ns	37.40 ns	48.89 ns
512	34.67 ns	38.19 ns	39.76 ns
1024	34.70 ns	34.39 ns	39.76 ns

Tabelle 4.18: Ergebnis - Signallaufzeit - Generatorpolynom

Der grafischen Darstellung ist deutlich zu entnehmen, dass für alle synthetisierten Architekturen bei einem Polynomgrad der Größe 8 der kritische Pfad sein Maximum erreicht. Mit größer werdenden Polynomgraden konvergieren alle kritischen Pfade zu einer Gesamtlaufzeit der Signale in einen Bereich von 30 bis 40 Nano Sekunden.

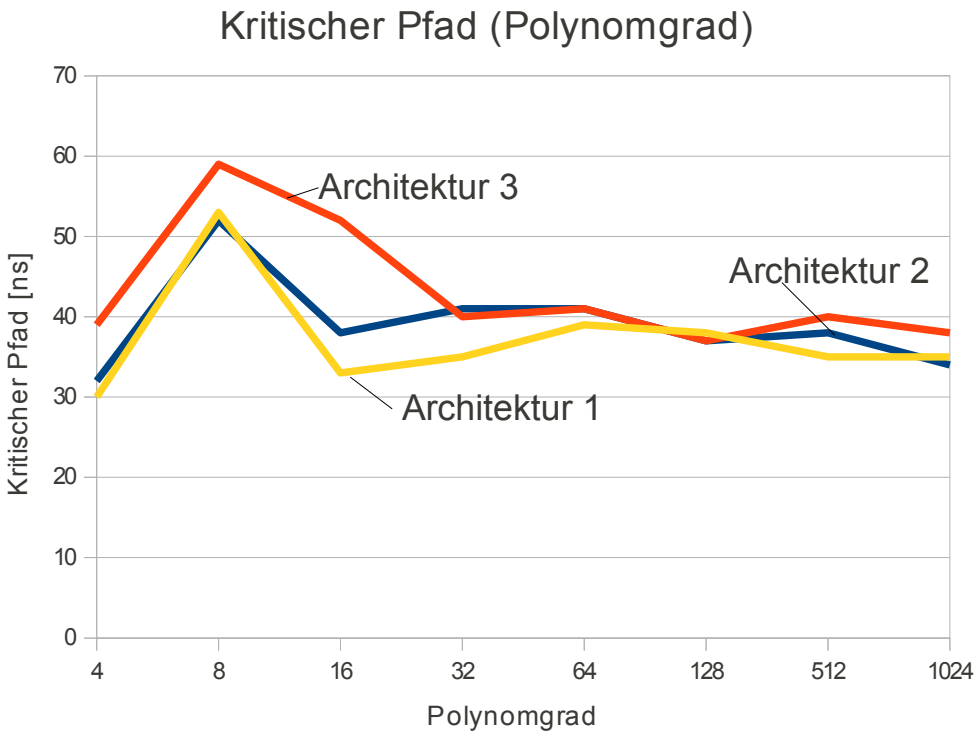

Abbildung 4.10: Kritischer Pfad Generatorpolynom

5 Zusammenfassung

Das Ziel der Studienarbeit war die Implementierung des Mixed-Mode BIST in der Hardwarebeschreibungssprache VHDL. Hierfür stand das Erstellen eines entsprechend der vorgegebenen Spezifikation entsprechenden Register-Transfer-Entwurfs an. Mittels diesem Register-Transfer-Entwurfs wurde die Implementierung der BIST Architektur in VHDL durchgeführt.

Diese Implementierung galt es anschließend umfangreich zu validieren um Fehler in der Implementierung entfernen zu können und eine korrekte Funktion der Architekturen gewährleisten zu könne. Validiert werden mussten hierfür alle laut der Spezifikation zulässigen Parameterkombinationen. Um die Korrektheit dieser Parameterspezifikationen zu zeigen, mussten alle aus der Implementierung heraus erstellbaren BIST Architekturen generiert und validiert werden. Dies wurde erreicht indem ein Referenzmodell des BIST erstellt wurde. Diese Referenzimplemtierung hatte die Aufgabe die selbe Funktion wie der in VHDL implementierte BIST zu repräsentieren. So wurden beide Implementierungen mittels automatischer Testmustergenerierung auf eine unterschiedliche Testvektorenausgabe hin untersucht um Fehler in den Architekturen eliminieren zu können.

Nach erfolgreichem Abschluss der Architektur Tests wurde eine Auswertung der BIST Architekturen auf ihren Flächen- und Zellenbedarf und ihre Signallaufzeiten durchgeführt. Hierbei war von Interesse welche Paramtergrößen sich bei Veränderung in welchem Maße auf den Flächenbedarf sowie die Signallaufzeiten auswirkten. Diese Ergebnisse wurden dokumentiert und ihre Ursachen erläutert.

Literaturverzeichnis

[AY04] A. A. Al-Yamani. "Deterministic built-in self-test for digital circuits". *Standford University*, April 2004.

[AYM03] A. A. Al-Yamani, E. J. McCluskey. "Seed encoding with LFSRs and cellular automata". *Design Automation Conference, 2003.*, Seite 560-565, Juli 2003.

[B+01] A. Benso, et al. "Online and offline BIST in IP-Core design". *International Test Conference*, vol. 18, Seite 1-8, September- Oktober 2001.

[Eic83] E. B. Eichelberger. "Random-pattern coverage enhancement and diagnosis for LSSD logic self-test". *IBM Journal of Research and Development*, Vol. 27, No. 3, Seite 265-272, Mai 1983.

[G+03] C. Galke, et al. "Kompaktierung von Testmustern für den Test von SoCs mittels einer Testprozessor-Architektur". *Ekompass-Workshop*, Seite 1-5, 2003.

[H+96] Y. Higami, et al. "Partially parallel scan chain for test length reduction by using retiming technique". *Test Symposium, 1996., Proceedings of the Fifth Asian*, Seite 94-99, 20-22 November 1996.

[H+09] A.-W. Hakmi, et al. "Restrict encoding for mixed–mode BIST". *27th IEEE VLSI Test Symposium*, Seite 179-184, 2009.

[L+00] B.-H. Lin, et al. "A fast signature computation algorithm for LFSR and MISR". *IEEE Transaction on computer-aided design of integrated circuits and systems*, VOL. 19, No. 9, Seite 1031-1040, September 2000.

[RT98] J. Rajski, J. Tyszer. "Design of phase shifters for BIST applications". *16th IEEE VLSI Test Symposium*, Seite 218-224, April 1998.

[S+03] A. D. Singh, et al. "Multimode scan: test per clock BIST for IP Cores". *ACM Transaction on Design Automation of Electronic Systems*, Vol. 8, No. 4, Seite 491-505, Oktober 2003.

[TM09] N. A. Touba, E. J. McCluskey. "Transformed pseudo-random patterns for BIST". *VLSI Test Symposium*, Seite 410-416, 30.April-3.Mai 1009.

[TM96a] N. Touba, E. McCluskey. "Test point insertion based on path tracing". *Proc. of VLSI Test Symposium*, Seite 2-8, 1996.

[TM96b] N. A. Touba, E. J. McCluskey. "Altering a pseudo-random bit sequence for mixed-mode scan BIST". *International Test Conference*, Oktober 1996.

[WA99] Y. Wu, S. M. I. Adham. "Scan-based BIST fault diagnosis". *"IEEE Transaction on Computer-Aided Design of integrated circuits an systems"*, vol. 18, no.2, Seite 203-211, Februar 1999.

[Wun98] H.-J. Wunderlich. "BIST for System-on-a-Chip". *Integration, the VLSI Journal*, vol. 26, no 1-2, Seite 55-78, Dezember 1998.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Stefan Bayha)