Möglichkeiten und Grenzen der automatischen SBST Generierung für einfache Prozessoren – Fallstudie des Testprozessors T5016tp

C. Galke, T. Koal, H. T. Vierhaus
BTU Cottbus, Lehrstuhl Technische Informatik

Abstrakt
Software-basierte Selbsttest (SBST) Konzepte für Prozessoren werden zunehmend interessant nicht nur durch die At-Speed Test Problematik. Auch bezüglich Stromaufnahme und Testzeit bietet dieses Testkonzept gegenüber dem Standard Verfahren wie etwa Scan-Test Vorteile. Als grundsätzlich problematisch ist die Erzeugung solcher software-basierten Testroutinen anzusehen, da bislang kein geeigneter einheitlicher Entwurfsprozess vorliegt.
Deshalb wurde exemplarisch für einen einfachen 16-bit Prozessorkern sowohl eine manuelle rein funktional erstellte SBST und eine automatisch generierte auf Strukturinformationen basierende SBST untersucht um die Möglichkeiten und Grenzen eines solchen Ansatzes aufzuzeigen.

1. Einführung

Der Test von hochintegrierten Schaltungen erfolgt derzeit fast ausschließlich per Scan-Test [1,2]. Einerseits liegt dies sicherlich an der dadurch erreichten Fehlerüberdeckung und der Verfügbarkeit von zahlreichen kommerziellen Testmustergeneratoren. Jedoch erfordern ständig kleiner werdende Strukturgrößen und damit in der Komplexität wachsende Systeme immer umfangreichere Herstellungstests. Somit sind Testzeit und Testdatenmengen, welche für den Scan-Test notwendig sind eigentlich heute schon nicht mehr wirtschaftlich tragbar. Grundlegend ändern daran auch zahlreiche Ansätze bezüglich Testdatenkomprimierung und Vorschläge zum Einsatz von On-Chip Testcontroller für die Ansteuerung einer Vielzahl von Scan-Ketten nichts [3].
Andererseits haben einige Autoren mit ihren Untersuchungen gezeigt, das teilweise recht gute Fehlerüberdeckungen mittels SBST erreicht werden können [4,5]. Dabei stellt sich letztendlich die Frage wie gut ein automatischer Ansatz im Bezug auf eine manuell erzeugte Routine ausfallen kann. Dies stellt den Schwerpunkt dieser Untersuchungen dar.
Diese Ausarbeitung hat die folgende unterteilt. Im Abschnitt 2 wird das verwendete Prozessormodell für die Fallstudie vorgestellt. Anschließend werden die beiden unterschiedlichen Ansätze für die Erzeugung der SBST Routinen beschrieben. Unter Punkt 4 werden die Ergebnisse der beiden Methoden dargelegt und verglichen. Dabei werden die Möglichkeiten und Grenzen der automatischen Erzeugung aufgezeigt. Mit einer kurzen Zusammenfassung wird die Ausarbeitung abgeschlossen.

2. Prozessormodell T5016tp

Der untersuchte Prozessor T5016tp dient in einem übergeordneten hierarchischem Testansatz für Systems-on-a-chip als eine Kerninstanz. Um den Zusatzaufwand für das Gesamtsystem gering zu halten handelt es sich um einen 16-Bit Prozessor der neben einem RISC-ähnlichen Befehlssatz spezielle Hardwarefunktionen beinhaltet.
So können vier der 16 Register im Datenpfad mittels Spezialbefehlen als 2 extern Linear Feedback Shift Register (LFSR) oder als Multiple Input Signature Register (MISR) oder wahlweise kombiniert verwendet werden [3].

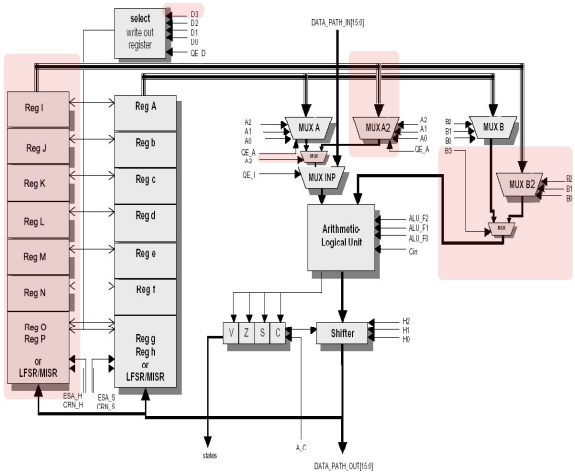

Abb. 1.: T5016tp Datenpfadstruktur für verbesserte funktionale Testbarkeit

Da für das Einsatzgebiet eine gute funktionale Testbarkeit notwendig ist, sind diese Strukturen dupliziert und parallel durch die entsprechenden Befehle ansteuerbar. Dies ermöglicht eine einfache Fehlererkennung innerhalb der duplizierten Strukturen, durch den Vergleich der Registerinhalte, sofern diese mit identischen Daten betrieben werden.

Weiterhin befinden sich im Datenpfad neben dem Registerfile die funktionalen Einheiten ALU und Shifter für die Festkommaarithmetik des Prozessors.
Um den Anforderungen des hierarchischen Testkonzeptes gerecht zu werden, besitzt der Prozessor vier parallele jeweils 16-Bit breite Ports und einen seriellen Ein/Ausgabe-Port. Die Steuerung des Datenaustausches über die Schnittstellen erfolgt über spezielle Port-Befehle.
Dabei kann ein spezieller TRS-Modus für 2 Ports aktiviert werden, in welchem ein schneller Vergleich zweier 8-Bit Werte möglich ist. Der TRS-Controller kann durch multiplen Vergleich über mehrere Takte erkennen, ob eine vorübergehende oder eine permanente Abweichung der beiden Werte vorliegt, wobei ein Wert als Golden Vector verwendet wird. Diese Funktion dient der Unterstützung der Fehleranalyse für den Online-Test.
Da die Schnittstellen im funktionalen Selbsttest somit nur sehr beschränkt testbar sind, wurden zusätzliche Multiplexer Strukturen integriert. Diese ermöglichen im speziell konfigurierbaren Test-Modus die an die Schnittstellen geschriebenen Werte wieder in den Prozessor für eine Überprüfung zu leiten. Damit bleibt die Schnittstelle letztendlich trotzdem nicht funktional testbar, aber es ermöglicht eine Überprüfung der Anbindung der Schnittstelle sowohl in die Ausgabe als auch in die Eingabe-Richtung.
In dieser Form ist der Testprozessor T5016tp für einen Software-basierten Selbsttest verbessert aber nicht optimal ausgelegt, aber mit einer Komplexität von zirka 3100 Gatter-Äquivalenten ist der Aufwand für die Implementierung relativ gering.

3. Die SBST Routinen

3.1. manuelle Erzeugung

Auf Grundlage der RT-Level Beschreibung und des Befehlssatzes erfolgte die manuelle Erzeugung einer SBST Routine für den Prozessor. Die Routine ist dabei so angelegt, das jeder Mikrobefehl mindestens einmal ausgeführt wird [11].
Die Struktur der Routine ist so aufgebaut, dass zuerst das Registerfile geprüft wird. Anschließend erfolgt die Überprüfung der einzelnen funktionalen Befehle mittels Ausführung und dem Vergleich mit fest in der Routine integrierten Referenzwerten. Den Abschluss der Routine bildet die Überprüfung der IO-Schnittstellen. Da diese trotz der hardware-technischen Veränderungen immer noch durch die Kerneigenschaften von Schnittstellen durch einen Selbsttest nicht optimal testbar sind, wurden an dieser Stelle detaillierte Einblicke in die Struktur des Designs notwendig. Zusätzlich wurde eine Flusskontrolle integriert, welche über die einzelnen Abschnitte der Routine eine spezielle Berechnung ausführt. Dabei wird nur das korrekte Berechnungsdatum erreicht, wenn alle Abschnitte genau einmal ausgeführt werden. Nachfolgend wird noch etwas detaillierter auf die einzelnen umgesetzten Konzepte eingegangen.
Die Analyse von Registerinhalten ist nicht trivial, da der LOAD / STORE- Befehlssatz einen direkten Vergleich von Registern mit Speicherzellen nicht zu lässt. Deshalb erfolgen Vergleiche stets mit dem Inhalt eines anderen Registers. Für den Test von zwei Registern Ra und Rb wird außerdem noch ein Register Rp benötigt, das eine Sprungadresse für den Fehlerfall enthält. Der komplette Test aller 16 Register erfordert insgesamt 121 Zeilen Code und 246 Byte. Damit ist der einfache Register-Test schon relativ aufwändig. Allein für den Test eines Registers werden sieben verschiedene Befehle und zwei zusätzliche Register benötigt. Für den Test auf Bridge-Fehler zwischen den Bits eines Registers werden alternierende Bitfolgen 0101... bzw. 1010... verwendet. Der Vergleichstest zur Entdeckung von Kopplungseffekten von Registern und Register-Bits gegeneinander umfasst 185 Zeilen Code und entspricht 486 Byte. Überprüft werden dabei nicht nur die Bit-Abhängigkeiten der Register, sondern indirekt auch die Multiplexer, internen Busse so wie der Kopierbefehl, welcher dabei verwendet wird.
Für das Testen der kompletten Multiplexer-Strukturen vor der ALU wird der Befehl zum Löschen der Registerinhalte, sowie der Additions- und Kopierbefehl benötigt. Dabei wird das Quellregister des Additionsbefehls, welches auch das Zielregister ist, mit dem 0-Vektor initialisiert. Das zweite Register erhält einen von Null verschiedenen zufälligen Wert. Nach der Additionsoperation befindet sich dieses Datum im Zielregister. Nun wird das zweite Register zurückgesetzt und durch wiederholtes Ausführen einer entsprechend gleichartigen Befehlsfolge unter Verwendung der Register k und k+1 wird der Ausgangswert durch das komplette Registerfile geschoben. Abschließend erfolgt eine Überprüfung des Anfangs- und Endwertes und bei einer

Abweichung wird ein entsprechende Fehlerindikator ausgelöst. Bei Fehlerfreiheit erfolgt die Ausführung noch einmal in umkehrter Richtung.

Die Erkennung der korrekten Ausführung einzelner Mikrobefehle erfolgt in der Regel durch Ausführung und Vergleich mit einem Referenzwert, der zuvor als Konstante geladen wurde. Da viele Befehle aber quasi paarweise auftreten (z.B. Inkrement und Dekrement) ist teilweise kein spezieller Referenzwert notwendig, da das Ergebnis eines früheren Tests noch in einem Register gehalten werden kann. Auf diese Weise lässt sich das Einlesen von Konstanten und damit die Größe der Selbsttest-Routine minimieren. Für die Spezialbefehle mit den LFSR und MISR Funktionalitäten erfolgt der Test durch den Betrieb der beiden Strukturen mit gleichen Konfigurationen und Daten. Eine Fehlererkennung erfolgt erneut durch den direkten Vergleich der erzeugten Werte beider Komponenten.

Die Konfiguration der parallelen Schnittstellen wird im sogenannten Port Config Register (PCR) festgelegt. Die Parallelports selbst sind zumindest partiell testbar. Für den Testbetrieb sind entsprechende zusätzliche Multiplexer vorhanden, die ebenfalls über das PCR konfiguriert werden. Somit sind Fehler die auf den Strukturen bis zur direkten Schnittstelle hin vorhanden testbar, jedoch das eigentliche IO-Interface nicht. Hier wäre eine Interaktion mit externen Baugruppen z. B. Bus-Kopplern notwendig. Grund für diese Einschränkung ist der hohe Aufwand für die Kopplung bidirektionaler Ports. Mit dem beschriebenen Szenario kann jedoch für jeden Port ein getesteter Betriebsmodus gesichert werden.

Die einzelnen Tests auf strukturelle Fehler und die Funktionstests sind nicht vollständig im Sinne einer funktionalen Fehlerüberdeckung. Deshalb wurde die gesamte Routine in 30 unterschiedliche Teilbereiche organisiert, welche den einzelner Fehlertypen und –orten entsprechen. Diese Subroutinen sind nicht komplett voneinander unabhängig, sondern es werden häufig benutzte Registerwerte einer erfolgreich durchlaufenen Teilroutine in der folgenden wiederverwendet. Dies dient einerseits erneut der Minimierung des Einlesens von Konstanten und anderseits erfolgt zusätzlich in jedem Abschnitt eine spezielle Berechnung unter Verwendung einigen dieser Werte, die unabhängig von dem in diesem Bereich durchgeführten Test sind. Am Ende der letzten Subroutine wird somit nur dann das korrekte Ergebnis der vollständigen Berechnung geliefert, wenn alle Teile fehlerfrei und komplett ausgeführt wurden. Mit dieser Flusskontrolle sind unter anderem fehlerhafte Sprünge über Teilbereiche oder -funktionen hinweg erkennbar.

3.2. Ansatz zur automatischen Generierung

Der Ansatz beruht zum einen auf dem Erzeugen funktional erreichbarer Testmuster mittel Constraint-ATPG für bestimmte Blöcke und zum anderen auf dem Abbilden dieser auf Software-Templates [8,9]. Die somit erzeugten Befehlssequenzen jedes einzelnen Blockes bzw. Baugruppe können dann in geeigneter Weise verbunden werden und stellen die endgültige Testsoftware dar.
Zum Testen einzelner Blöcke sind Befehlssequenzen gesucht, welche die entsprechenden Testmuster an Baugruppen anlegen und danach die Testantworten auffangen bzw. weiterverarbeiten. Unter Umständen sind mehrere Befehle notwendig, um ein Testmuster anzulegen oder Antworten zu speichern. Denn typischerweise werden nicht alle Ein- und Ausgänge eines Blockes durch einen einzelnen Befehl angesteuert. Zusätzlich bestehen die Templates aus weiteren Befehlen, die entweder festgelegte Testmuster als Immediate-Werte laden oder den Kontrollfluss der gesamten Routine gewährleisten.

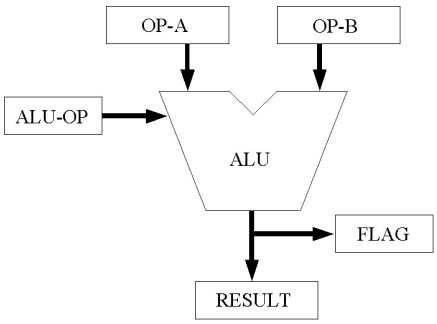

Abb. 2: ALU mit gepufferten Ein- und Ausgängen

Ein Software-Template für die in Abbildung 2 dargestellte ALU könnte wie in Tabelle 1 aufgezeigt aussehen. In diesem exemplarischen Template dienen die ersten 3 Befehle zum Initialisieren der ALU. Die beiden folgenden Instruktionen sollen das Ergebnis der Operation und damit mögliche Fehler innerhalb der ALU, welche sich auf die Berechnung ausgewirkt haben, durch das

Schreiben der Ergebnisse in den Speicher für eine Auswertung beobachtbar gestalten.

Tabelle 1: ALU Software-Template

Operation	Bemerkungen
set [OP-A], {data};	lade OP-A mit Daten
set [OP-B], {data};	lade OP-B mit Daten
{ALU-OP} [RESULT];	Ausführen der ALU-Operation
save [RESULT], {MEM1};	RESULT speichern
save [FLAG], {MEM2};	FLAG speichern

Um eine hohe Ausbeute bei der Fehlerüberdeckung zu erreichen sind kommerzielle ATPG Tools unverzichtbar. Jedoch ist der ATPG Prozess für SBST Konzepte nur bedingt geeignet, denn beispielsweise sind nicht alle Wertekombinationen funktional einstellbar und kommen somit als Testmuster nicht in Frage. Beim Beispiel der ALU könnten also bestimmte ALU Operationscodes nicht spezifiziert sein. Demzufolge darf das ATPG Tool diese ALU Ansteuerungen nicht verwenden, da eine anschließende Abbildung auf das Software-Template nicht möglich wäre. Mit Hilfe der „Constraint- ATPG“ kann genau das Verhalten erzwungen werden und eine Testmustergenerierung unter Bedingungen erfolgen. Das Prinzip, welches dahinter steht, beruht auf dem Modellieren von funktionalen Beschränkungen für bestimmte Designpunkte. Angenommen der 3 Bit Operationscode der ALU lässt die Kombinationen 110 und 111 nicht zu. Eine Constraint Modellierung dafür könnte folgendermaßen aussehen:

```
add atpg functions alu_111 AND op[2] op[1] ~op[0]
add atpg functions alu_110 AND op[2] op[1]  op[0]
add atpg functions no_alu_op AND alu_111 alu_110
add atpg constraints 0 no_alu_op
```

Nach der Definition der Funktionen, die den nicht gewünschten ALU Operationen entsprechen (alu_111, alu_110), können dann diese Funktionen explizit verboten werden (no_alu_op). Mit dieser Art der ATPG ist es möglich funktional gültige Testmuster zu generieren.
Nachdem Templates und Testmuster erzeugt wurden, ist ein Abbildungsvorgang zu starten, der jedes Testmuster mit dem entsprechenden Software-Template verbindet und dadurch die komplette SBST Routine erstellt. Hierbei werden die Platzhalter im Template mit den entsprechenden Werten aus dem Testmuster aufgefüllen. Die Menge der gefüllten Templates sind miteinander so zu verknüpfen, dass eine ablauffähige Software entsteht. Diese legt dann systematisch ein Testmuster nach dem anderen an eine Baugruppe an und sichert die Testantworten im Speicher.

Um die endgültige Güte der erzeugten Routine zu ermitteln, ist eine anschließende Einzelfehlersimulation unumgänglich. Denn die ermittelte Fehlerüberdeckung des Testmustergenerators kann von der tatsächlichen abweichen. Diese Abweichung kann sowohl durch Maskierungen aber auch durch die Tatsache entstehen, dass der Ablauf der Software unter Umständen mehr Testmuster an ein Modul anlegt als das ATPG Tool erzeugt hat.

4. Ergebnisse

Die manuell erzeugte SBST Routine ist mit ca. 1,5k Worten recht kompakt und erzielt eine Fehlerüberdeckung bezüglich „collapsed“ Stuck-At Fehler für den Datenpfad (DP) von ca. 92% (8472 der 9197 Fehler) und ca. 83%. für den Kontrollpfad (CP). Die Gesamtfehlerüberdeckung ist mit ca. 78% durch die im Selbsttest recht schlecht zu prüfenden Schnittstellen in der IO-Einheit etwas geringer.
Aufgrund der einfachen Struktur und der beschränkten Komplexität des verwendeten Prozessors wurden zwei verschiedene automatische Generierungsansätze betrachtet. Zum einem erfolgte die Generierung der erforderlichen strukturellen Testmuster bezogen auf die gesamte Prozessorstruktur als Einheit (Auto. SBST Proz.). Dies erfordert dann Templates die den kompletten Zustand des Prozessors einstellen. Diese sind mit ca. 80 Befehlen pro erzeugtem Testmuster recht umfangreich.

Tabelle 2: Fehlerüberdeckungen der SBST Routinen der unterschiedlichen Ansätze

	% Fehlerüberdeckung				
	T5016tp #SA: 17882	DP #SA: 9197	CP #SA: 3279	IO #SA: 5141	PI-PO #SA: 265
Man. SBST	78.4	92.1	82.9	52.7	44.5
Auto. SBST Proz.	68.3	84.3	80.2	33.2	45.7
Auto. SBST Modular	62.3	95.6	63.9	0.0	94.0
Auto. SBST Modular +	64.5	95.9	75.3	0.0	94,0

Wie in der Tabelle 2 zu sehen ist erreicht die so erzeugte SBST Routine zwar eine Fehlerüberdeckung für den Kontrollpfad der vergleichbar mit der manuell erzeugten ist. Jedoch werden die beiden anderen logischen Einheiten (DP, IO) deutlich schlechter überdeckt. Wobei die IO-Einheit auf grund der allgemein recht geringen funktionalen Testbarkeit besonders schlecht abschneidet.
Weiterhin resultiert die Abbildung der 90 erzeugten Muster aus der ATPG durch den hohen Initialisierungsaufwand in einer Routinenlänge von ca. 8k Worte (Tabelle 3). Dies stellt auch den Grund für die nahezu identische Güte der Routine für den Kontrollpfad dar. Es werden somit halt fünfmal so viele Befehle und Daten durch den Kontrollpfad geleitet wie bei der manuellen Routine.

Tabelle 3: SBST Routinen Eigenschaften der unterschiedlichen Ansätze

	SBST Routine	
	#Takte	#16-bit Worte
Man. SBST	3163	1474
Auto. SBST Proz.	28080	8150
Auto. SBST Modular	3157	1306
Auto. SBST Modular +	3355	1385

Der andere untersuchte Ansatz betrachtet nur einzelne Module (Auto. SBST Modular) womit sich die notwendigen Initialisierungen recht stark einschränken lassen und somit die Länge der Routine deutlich reduziert. Die Teilroutinen für die einzelnen Module werden aneinander gehängt und ergeben die Gesamtroutine. Für die ATPG werden dabei immer nur die zuvor noch nicht mitentdeckten Fehlerpunkte des Moduls verwendet. Dieses Vorgehen entspricht dem Ziel der Generierung einer kompakten SBST Routine für das in Module unterteilte Gesamtdesign. Damit wird nach jedem Modul eine Fehlersimulation nicht nur mit den Fehlerpunkten des aktuellen Moduls sondern aller noch nachflogend zu berücksichtigenden Blöcke notwendig. Exemplarisch wurde dies für die Basiskomponenten im Datenpfad durchgeführt, da diese den größten Anteil am Fehleruniversum des Prozessors haben. Die Ergebnisse für die einzelnen Blöcke deuten darauf hin, das mittels des Modularen Konzeptes eine automatische Generierung mit vergleichbaren Größenordnungen zu der manuell erzeugten SBST Routine bezüglich Speicherbedarf und Fehlerüberdeckung erreichbar ist. In der Tabelle 4 sind die erzielten Überdeckungen für die betrachteten Blöcke dargestellt. So werden für die ALU und die Registerfile-Multiplexer (MUX) mit 2-6 % mehr Fehlerüberdeckung besserer Ergebnisse erzielt. Jedoch sind die Ergebnisse für den Shifter und die LFSR/MISR-Strukturen etwas schlechter als beim manuellen Ansatz.

Tabelle 4: Fehlerüberdeckungen der SBST Routinen für die Datenpfad Module

	% Fehlerüberdeckung			
	ALU #SA: 657	Shifter #SA: 362	MUX #SA: 3279	LFSR/MISR #SA: 1484
Man. SBST	95,8	84,5	86,0	95,6
Auto. SBST Proz.	95,0	46,7	73,6	94,1
Auto. SBST Modular	98,0	82,3	92,1	94,8
Auto. SBST Modular ++	99,3	85,2	95,0	92,9

In der letzten Zeile sind zum Vergleich die erzielten Ergebnisse für die Betrachtung der Module als separate Einheit, unter Verwendung der jeweils kompletten Modul-Fehlerliste für die Erzeugung der Routinen, dargestellt. Die Unterschiede bezüglich des Modular Ansatzes hängen dabei mit der erhöhten abgebildeten Testmusteranzahl aufgrund der immer kompletten Modulfehlerlisten zusammen. Dabei ist erkennbar, das die Ergebnisse bis auf den Test der MUX-Struktur nicht signifikant über denen der manuell geschriebenen Routine liegen.

Tabelle 5: SBST Routinen Eigenschaften für die Datenpfad Module

		SBST Routine	
		#Takte	#16-bit Worte
Auto. SBST Modular	MUX	2273	955
	ALU	293	115
	Shifter	209	82
	LFSR/MISR	385	154
Auto. SBST Modular ++	MUX	2273	955
	ALU	775	193
	Shifter	535	138
	LFSR/MISR	2883	513

In Tabelle 5 sind für beiden Ansätze Modular und Modular++ die Größen der Teilroutinen aufgezeigt. Dabei wurden die Routinen beim Modular Ansatz in der Reihenfolge MUX, ALU, Shifter und LFSR/MISR erzeugt. Damit ergeben sich zum teil deutlich kürzere Teilroutinen beginnend mit der ALU gegenüber dem Modular++ Ergebnissen. Die Zusammenfassung der Modular++

Teilroutinen übersteigt mit einer Wortanzahl von 1799 schon deutlich die Größe der manuellen SBST Routine.

Wie in Tabelle 2 zu sehen ist, erreicht die Modular Routine mit ca. 62% eine um fast 16% geringere Gesamtfehlerüberdeckung als die manuell geschriebene Routine. Dabei entspricht die Länge mit 1306 Wörter nur ca. 88% des Aufwandes dieser. Um die Fehlerüberdeckung für den Kontrollpfad vergleichbarer zu gestalten wird die SBST Routine des Modular Ansatzes um noch nicht enthaltenen Befehle des Befehlssatzes erweitert (Auto. SBST Modular+). Dies erfolgte durch einfaches an einander hängen der bei der Generierung nicht genutzten Instruktionen. Als Ergebnis steigt die Fehlerüberdeckung des Kontrollpfades auf 75% (Tabelle 2). Somit bewirkt eine recht triviale Befehlssequenz eine Verbesserung von 12% und erreicht nun eine ähnliche Güte bezüglich des Kontrollpfades wie die manuell erzeugte Routine. Jedoch hat diese lokale Verbesserung nur einen sehr geringen Einfluss auf die Gesamtfehlerüberdeckung.
Aufgrund der eingeschränkten funktionalen Testbarkeit der IO-Schnittstellen wurde auf eine Erweiterung der Routine für diese Blöcke zu diesem Zeitpunkt verzichtet. Damit sind die Ergebnisse grundsätzlich nicht exakt vergleichbar. Jedoch lässt sich trotzdem feststellen, das eine automatische Erzeugung für einfache Prozessoren mit ähnlichen Fehlerüberdeckungen und Größenordnungen machbar ist. Bei Prozessoren mit umfangreichen IO-Schnittstellen erfordert dies jedoch wohl auch weiterhin teilweise manuellen Einsatz um die allgemein schlecht funktional testbaren Strukturen in das SBST Konzept mit zu integrieren.

5. Zusammenfassung

Es wurde zwei auf struktureller Constraint-ATPG beruhende SBST Routinen automatisch erzeugt und bezüglich der erreichten Fehlerüberdeckung und der Routinenlänge mit einer manuell erzeugten Selbsttest-Routine verglichen. Für die Datenpfad Komponenten welche eine gute funktionale Testbarkeit aufweisen konnten bessere Ergebnisse erzielt werden. Dies liegt an den Vorteilen einer ATPG gegenüber dem rein funktionalen Designer-Wissen. Für Prozessoren deren Datenpfad den Großteil des Fehleruniversums ausmacht sollte der Ansatz immer hinreichend gute Ergebnisse liefern und einem manuellen Erzeugen deutlich überlegen sein. Dagegen ist für sehr spezielle Strukturen wie die IO-Interfaces ein automatischer Ansatz kaum praktikabel.

6. Literatur

[1] J. Rajski, J. Tyszer et al., “Embedded Deterministic Test for Low-Cost Manufacturing Test”, IEEE International Test Conference, 2002, pp. 301-310

[2] K. Tumin, C. Vargas, R. Patterson, C. Nappi, “Scan vs. Functional Testing - A Comparative Effectiveness Study on Motorola’s MMC2107TM”, IEEE International Test Conference, 2001, pp. 443

[3] C. Galke, M. Grabow, H.T. Vierhaus, “Perspectives of Combining on-line and off-line Test Technology for Dependable Systems on a Chip“, International Online Test Synposium, 2003, pp.183-188

[4] F. Corno, G. Gumani, M. Sonza Reorda, and G. Squillero, “Fully Automatic Test Program Generation for Microprocessor Cores”, IEEE Design, Automation & Test in Europe Conference, 2003, pp. 1006-1011

[5] F. Corno, M. Sonza Reorda, G. Squillero, and M. Violante, “On the Test of Microprocessor IP Cores”, IEEE Design, Automation & Test in Europe Conference, 2001, pp. 209-213

[6] C. Galke, H. Schwabe, H. Fröschke, H. T. Vierhaus, R. Kothe, “Processor Design for Functional Self Test: -A Strategy and it’s Limits-“, Dresdner-Arbeitstagung-Schaltungs-und Systementwurf 2005

[7] H. Schwabe, C. Galke, H. T. Vierhaus, “Ein funktionales Selbsttest-Konzept für einfache Prozessorstrukturen-Fallstudie des Testprozessors T5016tp“, ITG Testmethoden und Zuverlässigkeit von Schaltungen und Systemen 2004

[8] R. S. Tupuri and J. A. Abraham, “A Novel Functional Test Generation Method for Processors using Commercial ATPG”, IEEE International Test Conference, 1997, pp. 743-752

[9] J. Zhou, H-J. Wunderlich, “Software-Based Self-Test of Processors under Power Constraints“, IEEE Design, Automation & Test in Europe Conference, 2006, pp 430-435

[10] A. Gärtner, “Optimierte Selbsttestprogramme für Mikroprozessoren”, Verlag TÜV Rheinland, 1985