Institut für Technische Informatik

Bachelorarbeit Nr. 182

Software-basierter Selbsttest von Peripherie-Komponenten

Jochen Bäßler

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer/in:	Dipl.-Inf. Dominik Ull
Beginn am:	21. Oktober 2014
Beendet am:	6. Mai 2015
CR-Nummer:	B.4.5

Kurzfassung

Software-basierte Selbsttest (SBST) Verfahren werden zumeist für das Testen von Mikroprozessoren eingesetzt, lassen sich jedoch auch auf Peripheriekomponenten anwenden. Der Vorteil von SBST, gegenüber Hardware-basierten Ansätzen besteht dabei im Verzicht auf spezielle Testhardware und Hochgeschwindigkeitstestgeräte und der Tatsache, dass Tests in der natürlichen Betriebsumgebung (engl. In-System) und bei normaler Betriebsfrequenz (engl. At-Speed) ablaufen. Peripheriekomponenten nehmen in vielen Systemen einen erheblichen Teil der Chipfläche ein, werden teilweise für sicherheitskritische Aufgaben eingesetzt und müssen folglich ausgiebig getestet werden.

Um strukturelle SBST-Verfahren erfolgreich auf diesen Typ von Komponenten anzuwenden, müssen Maßnahmen getroffen werden um deren geringe Beobacht- und Kontrollierbarkeit zu erhöhen, da andernfalls die erzielte Fehlerabdeckung der Verfahren zu niedrig ausfällt.

In dieser Arbeit werden zwei unterschiedliche Ansätze untersucht, um die strukturelle Fehlerabdeckung von SBST-Verfahren auf Kommunikationsperipheriekomponenten zu verbessern. Der erste Ansatz zielt auf eine verbesserte Kontrollierbarkeit der verwendeten Komponente ab. Dazu wird ein Loopback-basierter Mechanismus implementiert. Um darüber hinaus eine bessere Beobachtbarkeit zu erreichen wird als zweiter Ansatz der Zustand ausgewählter internen Signale dem System sichtbar gemacht.
Eine beispielhafte Anwendung der vorgestellten Methode auf die I^2C-Komponente eines RISC-Prozessors zeigt die Wirksamkeit der verwendeten Maßnahmen zur Verbesserung der strukturellen Fehlerabdeckung.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Motivation 7
1.2 Übersicht 8

2 Grundlagen **9**
2.1 Architektur von Mikrocontrollern 9
2.2 Test von Mikrocontrollern 15
2.3 Software-basierter Selbsttest 22

3 Implementierung **29**
3.1 Implementierung des Testsystems 29
3.2 Anpassungen zur Verbesserung der Haftfehlerabdeckung 35
3.3 Testprogrammzeugung 39

4 Experimente **49**
4.1 Synthese 49
4.2 Experimentelle Untersuchung der sequentiellen Tiefe 49
4.3 Experimentelle Ergebnisse 51
4.4 Analyse der vorgeschlagenen Maßnahmen 52
4.5 Analyse der Templates 55

5 Fazit **57**

6 Ausblick **59**

Literaturverzeichnis **61**

1 Einleitung

1.1 Motivation

Mit der Verdopplung der Integrationsdichte von integrierten Schaltkreisen innerhalb von zwei Jahren, nach dem Moore'schen Gesetz, steigt ebenfalls der Aufwand für deren Test kontinuierlich an. Automatische Testgeräte (engl. Automatic Test Equipement, ATE) mit hoher Betriebsfrequenz führen aufgrund sehr hoher Anschaffungskosten und schneller Alterung zu hohen Testkosten. Selbsttests umgehen die Notwendigkeit von teuren Testgeräte, wodurch sich Testkosten verringern lassen.
Hardware-basierte Selbsttest-Verfahren fügen in das Design der Komponente testspezifische Hardware ein um einen Selbsttest zu ermöglichen. Diese Zusatzhardware erhöht jedoch die Chipfläche der Schaltung und kann zu Performanceverlusten führen.

Eine alternative Testmethode bieten Software-basierte Selbsttests (SBST), die programmierbare Komponenten des Systems ausnutzen um Testvektoren an die Zielkomponente anzulegen und durch Abgleich mit Sollwerten eventuell vorhandene Fehler zu erkennen. SBST benötigt weder teure Testgeräte, noch das Vorhandensein spezifischer Teststrukturen im Hardwaredesign und ermöglichen das Testen unter den normalen Betriebsbedingungen der Schaltung - bei normaler Betriebsfrequenz und innerhalb des umgebenden Systems.

Zudem bieten SBST-Methoden erhebliches Potenzial zur Wiederverwendbarkeit über Herstellungstests hinaus, z. B. für Anwendungen in Feldtests, in der Rückläuferanalyse oder als periodische Selbsttestlösung zur Erhöhung der Verlässlichkeit von Systemen in sicherheitsrelevanten Bereichen.
Es sind eine ganze Reihe von SBST-Methoden für das Testen von Mikroprozessoren [CD01] [KPGX07] [PGSR10] [BBF+10] [PG05] [KMT+08] bekannt, allerdings beschränkt sich der Einsatz von SBST nicht nur auf Prozessoren. Stattdessen lässt sich der SBST-Ansatz auch auf Cache-Strukturen [TKPG14] [DCPS11] und Peripheriekomponenten [GPR+10a] [GHS+12] [APGP07a] [BSS+07] anwenden. Eine große Zahl der Peripheriekomponenten implementieren weitverbreitete, standardisierte Kommunikationsprotokolle wie Ethernet, USB (engl. Universal Serial Bus), I^2C (engl. Inter Integrated Curcuit) oder UART (engl. Universal Asynchronous Receiver Transmitter).
Peripheriekomponenten stellen eine besondere Herausforderung für die Entwicklung von effizienten Testprogrammen dar, da diese Komponenten für gewöhnlich eine deutlich geringere Beobacht- und Kontrollierbarkeit aufweisen als dies bereits bei Mikroprozessoren der Fall ist [GPR+10a]. Folglich muss für die effiziente Erzeugung von Testvektoren für Peripheriekomponenten zunächst eine Möglichkeit gefunden werden die Beobacht- und Kontrollierbarkeit zu erhöhen. Anschließend können SBST-Techniken Testprogramme erzeugen, die diese Komponenten mit einer hohen Fehlerabdeckung testen.

1.2 Übersicht

In dieser Arbeit wird eine Methode zur Generierung von Testmustern für strukturellen SBST von Peripheriekomponenten vorgestellt und Maßnahmen entworfen, welche die Beobachtbarkeit und Kontrollierbarkeit der internen Signale einer solchen Komponente erhöhen können. An einer beispielhaften Implementierung wird zudem gezeigt, wie sich diese Maßnahmen auf die strukturelle Fehlerabdeckung der vorgestellten Methode auswirken.
Dazu wird der OpenSource Prozessor miniMIPS[1] genutzt, eine dazu kompatible I^2C-Komponente implementiert und die Anwendung der Methodik und die verwendeten Maßnahmen besprochen. Die Arbeit gliedert sich dabei wie folgt:

Kapitel 2 behandelt die Grundlagen zu Mikrocontrollern (Abschnitt 2.1) und dem I^2C-Protokoll (Abschnitt 2.1.2). Unterkapitel 2.2 erklärt die Grundlagen zum Test von Mikrocontrollern. Im Abschnitt 2.2.3 wird das ATPG-Verfahren (engl. Automatic Test Pattern Generation) erläutert. Anschließend wird in Unterkapitel 2.2.5 näher auf bekannte Testverfahren wie ATE-Testing und Scan-basierte Testverfahren eingegangen.
Im Unterkapitel 2.3 wird das Grundprinzip, die wichtigsten Schritte und einige Varianten des SBST-Verfahrens erklärt. Die Anwendung von SBST auf Peripheriekomponenten wird im Abschnitt 2.3.3 behandelt und einige bestehende Arbeiten und deren Ergebnisse vorgestellt.

Kapitel 3 behandelt zunächst im Abschnitt 3.1 die Implementierung des Testsystems und im Folgenden (3.2) die Umsetzung der vorgestellten Maßnahmen zur Erhöhung der Beobacht- und Kontrollierbarkeit. In Unterkapitel 3.3.1 wird die Erzeugung struktureller Testmuster für die ausgewählte I^2C-Komponente diskutiert. Zur Umwandlung dieser Testmuster in ein ausführbares Programm werden in Abschnitt 3.3.2 zwei unterschiedliche Test-Templates vorgestellt.

Die Ergebnisse der experimentellen Anwendung auf dem Testsystem werden in Kapitel 4 besprochen. Kapitel 5 beschließt die Arbeit mit einer Zusammenfassung und einem Kapitel 6 gibt einen Ausblick auf mögliche weitere Arbeiten im Zusammenhang von SBST für Peripheriekomponenten.

[1] http://opencores.org/project,minimips

2 Grundlagen

Im Folgenden werden die für diese Arbeit wichtigsten Grundlagen vorgestellt. In Kapitel 2.1 wird ein kurzer Überblick über die Architektur von Mikrocontrollern und deren abstrakten Aufbau gegeben und anschließend in 2.1.2 das I^2C-Protokoll erläutert. Abschnitt 2.2 behandelt das Testen von Mikrocontrollern. Dazu werden die Grundlagen von Fehlermodellierung, DFT und ATPG erklärt und kurz verschiedene Testverfahren besprochen. Im Abschnitt 2.3 wird das Prinzip und verschiedene Methodiken des Software-basierten Selbsttests (SBST) erläutert und im Unterkapitel 2.3.3 werden bisherige Arbeiten zum Software-basierten Selbsttest von Peripheriekomponenten vorgestellt und zusammengefasst.

2.1 Architektur von Mikrocontrollern

2.1.1 Aufbau

Abbildung 2.1 zeigt eine vereinfachte Übersicht über den Aufbau und die typischen Komponenten eines Mikrocontrollers.

Systembus: Die einzelnen Komponenten eines Mikrocontroller-Systems kommunizieren über den Systembus miteinander. Klassischerweise besteht ein Systembus aus drei Bussen: der Datenbus ermöglicht den Austausch von Daten zwischen den Komponenten und besitzt n parallele Datenleitungen, wobei n der verwendeten Wortbreite entspricht. Der Adressbus spezifiziert das Ziel der übertragenen Daten und besteht aus der Menge an Leitungen, die benötigt werden um den adressierbaren Speicherbereich zu codieren. Der Steuerbus fasst alle benötigten Kontrollsignale zusammen und wird von einer Steuereinheit betrieben.

Prozessor: Prozessoren stellen die zentrale Komponente eines jeden Systems dar und sind für dessen Funktionalität entscheidend. Die Menge aller unterstützten Befehle eines Prozessors wird als Befehlssatz (engl. Instruction Set, IS) bezeichnet.

Ablaufsteuerung: In einem System gibt es eine Reihe von Komponenten zur Steuerung des Programmablaufs. So werden beispielsweise Interrupt Controller eingesetzt um das sog. Polling zu vermeiden. Polling bezeichnet eine Warteschleife, während jener der Prozessor auf die Reaktion eines externen Funktionsblocks (z. B. einer Peripheriekomponente) wartet. Durch die Verwendung von Unterbrechungen (engl. Interrupts) kann der Prozessor, während er auf eine angeforderte Reaktion wartet, Instruktionen anderer Programme durchführen. Trifft die Reaktion ein, so wird ein Interrupt durch den Interrupt Controller generiert und der Prozessor kann daraufhin mit der Ausführung des ursprünglichen Programms fortfahren. Durch diese Technik lässt sich der Befehlsdurchsatz des Prozessors erhöhen.

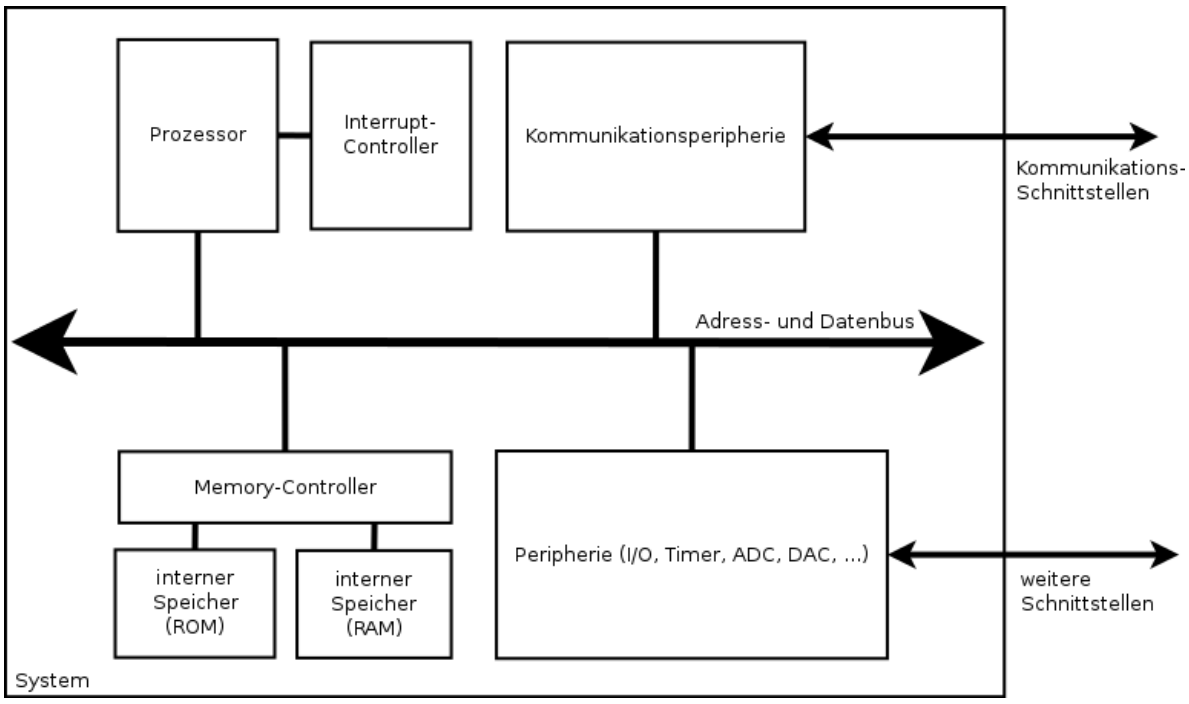


Abbildung 2.1: Abstrakte Architektur eines Mikrocontrollers

Speicher: Um Daten zu speichern und Programme zu halten benötigt ein System Speicher. Je nach Art der zu speichernden Werte und deren Verwendungszweck werden verschiedene Speichertypen in einem System verwendet. Unterscheiden lassen sich Speichertypen in solche auf die ausschließlich lesend zugegriffen werden kann (engl. Read-Only Memory, ROM) und jene auf die lesend und schreibend zugegriffen werden kann (engl. Random Access Memory, RAM). Zudem wird zwischen flüchtigen (engl. Volatile) und nicht-flüchtigen (engl. Non-Volatile) Speichertypen unterschieden, hierbei entscheidet die Tatsache ob gespeicherte Werte verloren gehen wenn die Stromversorgung unterbrochen wird oder nicht. Von besonderer Bedeutung in modernen Systemen sind die Speicherarten SRAM/DRAM (engl. Static / Dynamic Random Access Memory) für die Verwendung als Cache bzw. als Arbeitsspeicher um flüchtige Daten zu halten und EEPROM (engl. Electrical Erasable Read Only Memory) und Flash Speicher um die Systemfirmware, Konfigurationsdaten u.a. feststehende Werte nicht-flüchtig zu speichern.
Um Speicherzugriffe auf unterschiedliche Speicher zu verwalten werden Memory Controller eingesetzt. Somit lassen sich unterschiedliche Speichertypen über denselben Speicherbus adressieren, während die Anbindung der einzelnen Speicherelemente durch den Memory Controller übernommen wird.

Oftmals wird der Systembus auch als sogenannte MMI/O (engl. Memory Mapped Input Output) Schnittstelle für angeschlossenen Komponenten genutzt. Für dieses Verfahren werden bestimmte Speicheradressen, welche sich außerhalb des Systemspeichers befinden, auf bestimmte Ein- oder Ausgänge von Komponenten abgebildet. Mithilfe von gewöhnlichen Speicherbefehlen kann der Mikroprozessor des Systems somit auf die Ports der angeschlossenen Komponenten

zugreifen. Mit dieser Technik lassen sich spezielle Schnittstellen und Befehle für jede einzelne angeschlossene Komponente einsparen.

Peripheriekomponenten: Peripherie bezeichnet abstrakt alle Hardware die es ermöglicht Daten in ein System einzugeben oder auszulesen. Peripheriekomponenten lassen sich weiter in Systemperipherie und Kommunikationsperipherie aufteilen [GHS+12]. Systemperipheriekomponenten unterstützen oder entlasten den Prozessor des Systems, während Kommunikationskomponenten ein Kommunikationsprotokoll implementieren und so die Verbindung des Systems mit externen Systemen ermöglichen. Systemperipheriekomponenten sind dabei stärker mit dem Mikrocontroller verzahnt und stellen eine große Herausforderung für Testverfahren dar, insbesondere für nichtinvasive Verfahren, da deren Beobacht- und Kontrollierbarkeit noch geringer ausfällt, als die von Kommunikationsperipheriekomponenten [GHS+12] [GPR+10b]. Ein Beispiel für Systemperipherie stellen Direct Memory Access Controller (DMA) dar, die es ermöglichen große Datenmengen zwischen zwei Speicherquellen zu transferieren, ohne dabei den Prozessor, nach anfänglicher Konfiguration, zu belasten. Somit lässt sich der Prozessor entlasten und der Datentransfer unter Umständen erhöhen [GPR+10b].

Kommunikationskomponenten: Unter diese Kategorie fallen Peripheriekomponenten die standardisierte Kommunikationsprotokolle implementieren um eine Kommunikation mit externen Geräten über eine spezifizierte Schnittstelle zu erlauben. In vielen Mikrocontrollern sind eine ganze Reihe dieser Schnittstellen verbaut, folglich nehmen diese eine erhebliche Chipfläche ein [GHS+12].

Kommunikationsprotokolle werden grundsätzlich nach Übertragungsrichtung (uni- bzw. bidirektional), nach Übertragungsart (seriell bzw. parallel) sowie nach deren Zeitverhalten (synchron bzw. asynchron) und nach deren Topologie unterschieden.

Unidirektionale Protokolle erlauben den Datentransfer nur in eine spezifische Richtung, während bidirektionale Protokolle Kommunikation in beide Richtungen zulassen. Der überwiegende Teil der Kommunikationsprotokolle die für Peripherieschnittstellen eingesetzt werden ist bidirektional.

Die Übertragungsart eines Protokolls wird in seriell und parallel unterschieden und definiert sich über die Anzahl der übertragenen Datenbits pro Zeiteinheit. Überträgt eine Kommunikationskomponente pro Zeitschritt nur ein Bit, so handelt es sich dabei um eine serielle Übertragung. Serielle Protokolle benötigten folglich mindestens n Zeitintervalle für die Übertragung von n Datenbits. Bekannte Beispiele serieller Kommunikationsprotokolle sind USB, Ethernet Schnittstellen, CAN (engl. Controller Area Network), I^2C und SPI (engl. Serial Peripheral Interface). Parallele Kommunikationskomponenten besitzen dagegen mehrere physikalische Datenleitung und können somit pro Zeitintervall mehrere Datenbits gleichzeitig übertragen. Aufgrund von unterschiedlichen Signallaufzeiten und hoher Störanfälligkeit der parallelen Leitungen werden parallele Protokolle hauptsächlich für Hochgeschwindigkeitsanwendungen eingesetzt. Zu parallelen Protokollen zählen PCI (engl. Peripheral Component Interconnect) und die größte Zahl aller Systembusimplementierungen.

Die Topologie eines Kommunikationsprotokolls bezeichnet die Art und Weise wie Kommunikationsteilnehmer miteinander verbunden sind. Typischerweise wird zwischen Bus-, Ring,- Stern,- Baum- und Maschen-Topologie unterschieden. Der Vorteil von Bus-Strukturen ist, dass

beliebig viele Teilnehmer über nur eine Busleitung miteinander verbunden werden können. Bekannte Vertreter der Bus-Topologie sind Ethernet, CAN und I^2C. Da alle Kommunikationsteilnehmer über nur eine Bus-Leitung miteinander Verbunden sind, kann zu jeder Zeit maximal ein Teilnehmer auf den Bus schreiben, jedoch alle gleichzeitig Daten vom Bus lesen. Folglich enthalten Protokolle von Bus-Topologien zugriffsregulierende Elemente. Diese Zugriffsregulierung kann explizit erfolgen, in dem eine Arbitrationslogik eingesetzt wird, oder implizit durch die Verwendung von hierarchischen Beziehungen zwischen den angeschlossenen Komponenten. So unterscheiden manche Protokolle (z. B. I^2Coder USB) zwischen Master- und Slave-Kommunikationsteilnehmern, die unterschiedliche Rechte, z. B. zum initiieren einer Kommunikation, besitzen. Bei hierarchischen Bus-Topologien kann die Verwaltung der Buszugriffe durch den Master erfolgen, oder sogar auf diesen beschränkt werden (Kommunikation nur zwischen Master und Slave, nicht jedoch zwischen Slaves).

Eine weiteres Unterscheidungsmerkmal von Kommunikationsprotokollen ist die Abhängigkeit von einem Taktsignal im Fall von synchronen Protokollen oder der Verzicht darauf bei asynchroner Kommunikation. Letztere sind Ereignis-basiert, d.h. es wird über bestimmte Ereignisse eine zeitweilige Synchronizität der Kommunikationspartner erreicht, ohne dass diese ein gemeinsames Taktsignal teilen. Beiden Komponenten müssen deshalb die Parameter der Kommunikation bekannt sein. Zu diesen Parametern gehören u. a. die Übertragungsrate, die Verwendung und Position von Paritätsbits und die Reihenfolge der Bits - niederwertigstes (engl. Least Significant Bit - LSB) oder höchstwertiges (engl. Most Significant Bit - MSB) Bit zuerst. Eine bekannte asynchrone Komponente ist die UART-Schnittstelle.
Bei synchronen Protokollen teilen die Kommunikationspartner dagegen ein gemeinsames Taktsignal für Synchronisationszwecke. Je nach Topologie des Protokolls wird dieses Taktsignal nur von einer Masterkomponente - im hierarchischen Fall - oder vom aktuellen Transmitter - nicht-hierarchischer Fall - erzeugt.

2.1.2 Das I^2C Protokoll

Im folgenden Abschnitt wird das I^2C- Protokoll, als Beispiel für eine Kommunikationsschnittstelle eines Mikrokontrollers, vorgestellt. Bei I^2C handelt es sich um ein einfaches, kostengünstiges Zweidraht-Bussystem zur bidirektionalen, synchronen, seriellen und hierarchischen Kommunikation zwischen einem Master- und mehreren Slavekomponenten. Übertragungen sind dabei stets byteorientiert und nutzen Empfangsbestätigungen durch den Empfänger. Das Konzept von I^2C stammt aus dem Jahr 1982 und wurde von Philips Semiconductors (inzwischen NXP Semiconductors) entwickelt, um die interne Kommunikation zwischen verschiedenen Integrierten Schaltungen zu verbessern und standardisieren. Die aktuellste I^2C-Spezifikation [NXP14] (Version 6) vom April 2014 sieht vier verschiedene (bidirektionale) Übertragungsraten vor: Standard-mode (SM) bis zu 100 kbit/s, Fast-mode (FM) bis zu 400 kbit/s, Fast-mode Plus (FM+) bis zu 1 Mbit/s und High-Speed mode (HS-mode) mit bis zu 3,4 Mbit/s.

Topologie

Das I^2C-Protokoll unterscheidet zwei hierarchisch getrennte Typen von Kommunikationsteilnehmern: Master und Slave. Slaves sind passive Komponenten und somit nicht in der Lage, im Gegensatz zum Master, eine Kommunikation zu initiieren, ihren Kommunikationspartner zu wählen oder eine begonnene Übertragung zu beenden. Aufgrund des bidirektionalen Protokollcharakters sind sowohl Master- als auch Slavekomponenten in der Lage als Sender oder Empfänger zu agieren. Die Wahl der Kommunikationsrichtung wird dabei zu Beginn der Kommunikation durch den Master festgelegt. Dabei ist jedoch zu beachten, dass dem Master in beiden Fällen die Kontrolle der Kommunikation vorbehalten ist, auch wenn er die Rolle des Empfängers einnimmt. Ein I^2C-Netzwerk besteht typischerweise aus einem Master und mehreren Slavekomponenten die über einen gemeinsamen Bus verbunden sind (vgl. Blockschaubild 2.2). Der I^2C-Bus, über den alle angeschlossenen Komponenten verbunden sind, besteht aus lediglich zwei Leitungen: die erste überträgt die Daten (engl. Serial Data, SDA) und kann von Master- und Slavekomponenten getrieben werden, die zweite den seriellen Takt (engl. Serial Clock, SCL), der ausschließlich vom Master gesetzt wird. Jede I^2C-Komponente, unabhängig ob Master oder Slave, besitzt eine eindeutige, 7 Bit-lange Adresse innerhalb des Busses. Es lassen sich somit theoretisch $2^7 = 128$ Komponenten an einen Bus anschließen. Aufgrund von 16 reservierten Adressen, reduziert sich diese Anzahl auf 112. I^2C-kompatible Schaltkreise lassen eine gewisse Anpassung ihrer Adresse zu, um die Verwendung mehrerer identischer Schaltelemente innerhalb eines Netzwerks zu ermöglichen, ohne dass es dabei zu Adresskonflikten kommt.

I^2C-Übertragungen sind byteorientiert, d. h. es wird mindestens ein Byte übertragen. Die Bits innerhalb eines Bytes werden dabei nach dem MSB-first Prinzip übertragen. Nach jedem Byte fordert das Protokoll zudem eine Empfangsbestätigung (engl. Acknowledge, ACK) durch den Empfänger, wodurch sich kurzzeitig das Sende-Empfangs-Verhältnis der beiden Kommunikationspartner umdreht. Bei Empfang eines ACK setzt der Sender die aktuelle Kommunikation fort. Wenn dagegen eine negative Empfangsbestätigung (engl. Not Acknowledge, NACK) auftritt, so wird der aktuelle Kommunikationsvorgang durch den Sender abgebrochen.

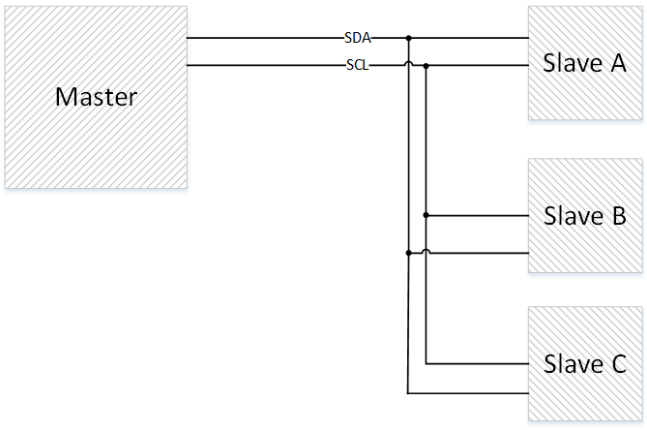


Abbildung 2.2: Topologie eines I^2CNetzwerks.

Wird ein ACK empfangen so beginnt die Übertragung des nächsten Datenbytes, wenn weitere Daten übertragen werden sollen, oder die Kommunikation durch den Master beendet. Fällt diese Empfangsbestätigung jedoch negativ aus (Empfang eines NACK) wird die Kommunikation durch den Master abgebrochen.

Um den Beginn bzw. das Ende einer Übertragung anzuzeigen, werden zwei Bedingungen formuliert: Start und Stopp (vgl. Abb. 2.3). Diese können lediglich durch den Master erzeugt werden. Die Startbedingung ist definiert durch eine fallende Flanke auf der SDA-Leitung während SCL konstant den logischen Wert '1' hält (High-Phase). Die Stoppbedingung ist komplementär definiert, durch eine steigende SDA Flanke während der High-Phase von SCL. Damit diese Bedingungen nicht fälschlicherweise erkannt werden, darf sich der Wert von SDA während einer validen Datenübertragung nicht ändern, solange sich SCL in der High-Phase befindet.

Ablauf einer Kommunikation

Im folgenden Anschnitt wird der protokollgerechte Ablauf einer Kommunikation anhand der schematischen Skizze 2.3 erklärt.

Eine I^2C-Kommunikation beginnt stets durch das Setzen der Startbedingung durch den Master (in der Abbildung grün markiert). Das erste Byte, welches zu Beginn einer Kommunikation durch den Master gesendet wird, setzt sich aus der 7-Bit Adresse des Ziels und dem Operationsbit (engl. read/write, R/W) zusammen. R/W Bit '1' ist definiert als Leseoperation, '0' als Sendeoperation. Bestätigt die adressierte Komponente ihre Bereitschaft, durch die Übertragung eines ACK (in der Skizze orange markiert), so beginnt die eigentliche Datenübertragung. Ein ACK ist dabei definiert als eine logische '0' auf der SDA-Leitung. Empfängt der Master dagegen eine logische '1' auf SDA (Definition von NACK) bedeutet dies entweder eine nicht vergebene Adresse oder einen nicht kommunikationsbereiten Slave. Folglich wird der Kommunikationsversuch durch den Master beendet.

Anschließend werden, gemäß der gewählten Übertragungsrichtung, Datenbytes übertragen. Nach jedem übertragenen Byte stoppt der Sender für eine SCL-Periode und wartet auf ein ACK des Empfängers (vgl. Abb. 2.3, Takt 8), bevor das nächste Byte übertragen wird. Empfängt er stattdessen ein NACK, so bricht er die Kommunikation ab.

Nach der Übertragung des letzten Bytes wird die Kommunikation durch das Setzten der Stoppbedingung (in der Abbildung rot markiert) durch den Master beendet. Eine Besonderheit stellt dabei eine lesende Masterkomponente dar. Da nur dem Master die Anzahl der zu lesen Bytes bekannt ist, muss der übertragende Slave nach dem letzten Byte unterbrochen werden. Dies geschieht durch die Übertragung eines NACK nach dem letzten Byte (vgl. Abb. 2.3, blaue Markierung), was zu einem Abbruch der Übertragung durch den Slave führt.

Erweiterungen

Es gibt eine Reihe von Erweiterungen des einfachen I^2C-Protokolls.
Dazu gehört die Möglichkeit des Multi-Master Betriebs, also dem Vorhandensein von mehr als einer Masterkomponente im I^2C-Netzwerk. Dazu wird Arbitrierungslogik eingesetzt, um Zugriffskonflikte

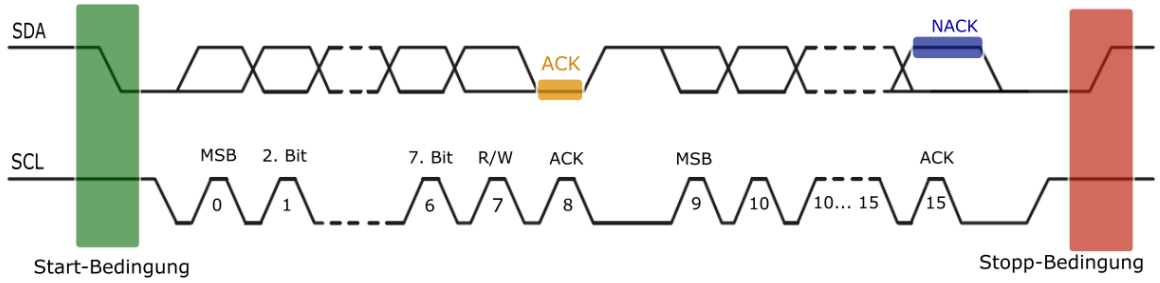


Abbildung 2.3: Schematischer Ablauf einer I^2C-Datenübertragung.

auf den gemeinsamen I^2C-Bus aufzulösen.
Um die Anzahl der möglichen Komponenten innerhalb eines I^2C-Busses zu erhöhen, existiert eine Erweiterung, welche den Adressraum von sieben auf zehn Adressbits erhöht. Diese Erweiterung ist rückwärts-kompatibel, es können folglich weiterhin 7-Bit Komponenten verwendet werden. Laut der NXP-Spezifikation [NXP14] wird diese Erweiterung jedoch bislang selten eingesetzt.

Sollte ein Slave nach einem übertragenen Byte ausgelastet sein, kann er die Taktleitung auf Low halten, bis er für das nächste Byte bereit ist. Dieses Verfahren wird Taktdehnung (engl. clock stretching) genannt, birgt jedoch die Gefahr, dass ein Fehler in einem Slave das gesamte I^2C-Netzwerk blockiert, da ein konstanter LOW-Wert auf SCL nicht aufgehoben werden kann.

Ein zusätzlicher Übertragungsmodus (Ultra-Fast Mode, UFm) sieht zudem eine Anpassung für unidirektionale Übertragung mit einer Übertragungsrate von bis zu 5 Mbit/s vor.

2.2 Test von Mikrocontrollern

Als hochkomplexe, mikroelektronische Systeme sind Mikrokontroller anfällig für eine breite Spanne an Fehlertypen. Zugleich werden sie immer häufiger in sicherheitskritischen Bereichen eingesetzt, in denen höchste Anforderungen an die Zuverlässigkeit gelten. Folglich sind strukturierte, wirtschaftliche und qualitativ hochwertige Testverfahren notwendig um diese Anforderungen zu erfüllen und gleichzeitig den Kunden niedrige Preise bei guter - oder im sicherheitskritischen Fall bei höchster - Qualität anbieten zu können.

Dazu werden unterschiedliche Verfahren für die verschiedenen Testanwendungsfelder (Herstellungstest, Feldtest, Rückläuferanalyse) eingesetzt, da sich die Anforderungen je nach Fall zum Teil erheblich unterschieden. Unter Herstellungstest (engl. Manufactoring Test) versteht man den Test eines fertigen Chips nach Abschluss dessen Fertigung. Dabei steht die Fehlerabdeckung und insbesondere die Testzeit im Vordergrund, da lange Testzeiten zu hohen Herstellungskosten führen. Beim Feldtest (engl. In-Field Test) ist eine hohe Fehlerabdeckung, kurze Testzeit und eine nicht-invasive Testmethode gefragt, um das zu testende Gerät in seiner natürlichen Betriebsumgebung zu testen. Dies ist wichtig um nicht reproduzierbare Fehlerbilder zu vermeiden. Das Auftreten dieser NFF (eng. No Failure Found) genannten Fehler hängt von den genauen Betriebsbedingungen ab und kann deshalb u.U. nach dem Ausbau des fehlerhaften Systems nicht reproduziert werden [JRW14]. Rückläuferanalyse beschäftigt

sich mit der Fehleranalyse von funktionsunfähigen Geräten um Rückschlüsse auf die Fehlerursache zu erhalten und so Fehlerquellen zu beheben. Dabei ist die Testzeit verhältnismäßig irrelevant, ebenso können problemlos invasive Methoden eingesetzt werden, und eine hohe Testabdeckung ist entscheidend.

2.2.1 Fehlermodellierung

Fehler lassen auf unterschiedlichen Ebenen definieren und modellieren.

Defekt (engl. Defect) bezeichnet einen physikalischen Fehler. Ein Defekt kann während des Fertigungsprozesses auftreten, durch eine spätere Beschädigung des Systems erfolgen oder durch Alterungserscheinungen hervorgerufen werden. Ein Fertigungsfehler könnte eine zu dünn aufgebrachte Siliziumschicht an einem Transistor-Gate sein.

Ein Fehler (engl. Fault) ist die Modellierung eines Fehlers mittels eines Fehlermodells. Ein Beispiel dafür ist das Haftfehlermodell.

Fehlerhafter Systemzustand (engl. Error) beschreibt einen fehlerhaften internen Zustand der z. B. durch einen aktiven Fehler entstehen kann. Ein Beispiel für einen solchen Zustand wäre ein falscher Wert in einem internen Register.

Systemausfall (engl. Failure) beschreibt die (fehlerhafte) Abweichung der Funktionalität eines Systems von der erwarteten, korrekten Funktionalität [ALR$^+$01].

Eines der am häufigsten genutzten Fehlermodelle ist das Haftfehlermodell (engl. Stuck-At Fault Model). Das Haftfehlermodell bezieht sich auf die Darstellung einer Schaltung auf Gatterebene und nimmt an, dass Fehler lediglich an den Ein- oder Ausgängen eines Gatters oder an den primären Eingängen (engl. Primary Input, PI) bzw. den primären Ausgängen (engl. Primary Output, PO) der Schaltung auftreten und zwar in der Form eines konstanten logischen Werts. Folglich existieren zwei unterschiedliche Haftfehler: Stuck-at-0 und Stuck-at-1. Eine Schaltung mit n möglichen Haftfehlerpositionen besitzt eine Fehlerliste (eine Liste aller Modellfehler der Schaltung) mit maximal $2n$ Einträgen. [BA01, S. 63, 71] Die Anzahl der Haftfehler einer gegebenen Schaltung lässt sich durch Verfahren zur Fehlerkollabierung (engl. Fault Collapsing) reduzieren. Dazu werden äquivalente Haftfehler definiert: [BA01]

Äquivalenz: Zwei Fehler sind äquivalent, wenn sie nicht voneinander unterschieden werden können. So kann z. B. ein Stuck-at-0 Fehler am Eingang eines Inverters nicht von einem Stuck-at-1 Fehler an dessen Ausgang unterschieden werden und umgekehrt. Alle äquivalente Fehler lassen sich zu disjunkten Äquivalenzklassen zusammenfassen.

Beim fault collapsing wird aus jeder Äquivalenzklasse nur ein einziger Vertreter als Repräsentant in die Fehlerliste der untersuchten Schaltung aufgenommen. Durch die Anwendung von fault collapsing kann die Größe der Haftfehlerliste einer Schaltung im Schnitt um 50 bis 60% [WWW06, 45] reduziert werden.

2.2.2 Funktionale und strukturelle Tests

Testverfahren lassen sich abstrakt in zwei Kategorien einordnen.

Funktionale Testverfahren nutzen keine Informationen über den strukturellen Aufbau einer Schaltung, sondern lediglich Informationen über die Funktionalität des zu testenden Systems. Es handelt sich bei ihnen folglich um Blackbox-Tests.
Dazu wird bei funktionalen Testverfahren eine bestimmte Belegung an die PIs angelegt. Eine solche Belegung wird Testvektor genannt. Die daraus resultierenden Werte an den POs können mittels Simulation der fehlerfreien Schaltung, für den gegebenen Testvektor, bestimmt werden [WWW06, S. 41]. Diese Soll-Werte werden anschließend mit den tatsächlichen Werten der POs verglichen. Abweichende Werte bedeuten folglich die Existenz eines Fehlers in der Schaltung. Eine Menge zusammenhängender Eingabe- und Ausgabetestvektoren wird als Testmuster bezeichnet.

Eine rein kombinatorische Schaltung lässt sich durch testen aller 2^n möglichen Testvektoren erschöpfen Testen (engl. Exhaustive Testing), dies ist jedoch unwirtschaftlich für große n, aufgrund der Tatsache dass die Anzahl der Belegungen exponentiell mit der Zahl der primären Eingänge wächst. Bei sequentiellen Schaltungen ist das Problem noch größer, da selbst das Anlegen aller 2^n möglichen Testvektoren nicht das Erreichen aller internen Zustände garantiert [WWW06]. Es müssen folglich Sequenzen von Eingangsbelegungen erzeugt werden, um schwer zu testende Fehler abzudecken.

Ein Problem funktionaler Ansätze ergibt sich aus dem Mangel an strukturellen Informationen, welche eine Angabe der Testabdeckung struktureller Fehler unmöglich macht. Folglich ist es schwierig gesicherte, qualitative Angaben zu funktionalen Testmethoden anzugeben [PGSR10].

Strukturelle Testverfahren beziehen zusätzlich zu den funktionalen auch strukturelle Informationen über die Schaltung mit ein und benutzen Fehlermodelle um die Anzahl aller betrachteten Fehler bestimmen zu können. Strukturelle Informationen und Fehlermodelle ermöglichen es diese Testverfahren qualitativ einzuschätzen und zu vergleichen.

Die Fehlerabdeckung (engl. Fault Coverage, FC) eines Verfahrens ist definiert als die Anzahl der abgedeckten Fehler im Bezug zur Gesamtzahl der Fehler in der betrachteten Schaltung [WWW06, S. 41].

$$FC = \frac{\#\ abgedeckte\ Fehler}{Gesamtfehlerzahl}$$

Um die Effizienz eines Testverfahrens (engl. Fault Detection Effeciency, FDE) zu bewerten kann die Gleichung um redundante Fehler erweitert werden [WWW06, S. 41]. Redundante Fehler (engl. Redundant Fault, RF) verändern die Eingabe-Ausgabe-Funktion der betrachteten Schaltung nicht und können folglich nicht durch Testvektoren aufgedeckt werden [BA01].

$$FDE = \frac{\#\ abgedeckte\ Fehler}{Gesamtfehlerzahl - \#RF}$$

2.2.3 Automatische Testmustererzeugung

Das Aufstellen von Testvektoren und ganzen Testmustern für eine - nicht-triviale - gegebene Schaltung ist eine komplexe Aufgabe. Deshalb werden für diese Aufgabe ATPG-Programme eingesetzt. Es gibt sowohl für funktionale als auch für strukturelle Testmethoden (vgl. Abschnitt 2.2.2) ATPG-Programme. Im Folgenden wird strukturelles ATPG vorgestellt, da im Laufe dieser Arbeit strukturelle Testmuster erzeugt werden sollen. Neben funktionalen und strukturellen ATPG-Programmen wird zudem zwischen kombinatorischen und sequenziellen ATPG-Verfahren unterschieden.

Kombinatorisches ATPG

Kombinatorische ATPG-Verfahren behandeln ausschließlich rein kombinatorische Schaltungen. Ein Fehler in einer kombinatorischen Schaltungen lässt sich mit einem Testmuster testen, welches genau einen Testvektor und einen zugehörigen Ergebnisvektor enthält [BA01]. Die bevorzugte ATPG-Methode für kombinatorische Schaltungen ist Pfadsensibilisierung (engl. path sensitization) und nutzt eine Darstellung der Schaltung auf Gatterebene. Dabei wird ein Fehler in die Schaltung eingefügt und durch eine passende Wahl der Eingangbelegung dafür gesorgt, dass dieser Fehler aktiviert (d.h. der Fehler führt zu einer Wertänderung an einem Gatter) und propagiert wird (d.h. die fehlerhafte Wertänderung führt an einem oder mehreren POs der Schaltung zu einem abweichenden Wert, verglichen mit einer fehlerfreien Ausführung).
Das Problem der Pfadsensibilisierung lässt sich auf das Boolesche Erfüllbarkeitsproblem zurückführen und ist folglich ein NP-Vollständiges Problem. Das bedeutet, dass nicht-heuristische ATPG-Algorithmen exponentielle Laufzeit besitzen.

Während der Berechnung von Testmustern decken ATPG-Programme zusätzlich redundante Hardware auf, d.h. Hardware deren Entfernung aus der Schaltung keinen Einfluss auf deren Funktionalität besitzt. Durch das Entfernen von redundanter Hardware kann die Chipfläche und der Energiebedarf der Schaltung verringert und gleichzeitig die maximale Taktfrequenz (durch verkürzte Verzögerungszeiten) erhöht werden. Dabei muss beachtet werden, dass redundante Hardware auf Designebene zusätzliche Zwecke erfüllen kann (z. B. Erhöhung der Fehlertoleranz) und folglich nicht entfernt werden darf.

Sequenzielles ATPG

Das Erzeugen von Testmustern für sequenzielle Schaltungen ist komplexer als die Erzeugung für kombinatorische Schaltungen [BA01]. Im Gegensatz zu Testmustern für kombinatorische Schaltungen, bestehen Testmuster für sequentielle Schaltungen im Allgemeinen aus mehr als nur einem Paar von Testvektoren, welche in einer fest vorgegebenen Reihenfolge angelegt werden müssen.
Zunächst muss die Schaltung jedoch in einen bekannten Zustand versetzt werden, bevor der eigentliche Test begonnen werden kann. Dieses sog. Initialisierungsproblem ist nicht-trivial lösbar und erfordert erheblichen Aufwand, sowohl im Bezug auf Rechenaufwand als auch auf die Menge der erzeugten Testmuster.

Für den Test einer sequentiellen Schaltung ist die betrachtete sequentielle Tiefe des Tests entscheidend. Wird eine zu niedrige sequentielle Tiefe betrachtet, können bestimmte Verhalten einer Schaltung nicht beobachtet werden. Folglich können Fehler in diesem Schaltungsbereich nicht getestet werden, was zu einer niedrigen Fehlerabdeckung führt. Eine höhere sequentielle Tiefe ermöglicht somit ein ausgiebigeres Testen der gegebenen sequentiellen Schaltung, bis zu einem Punkt an dem jegliches sequentielles Verhalten der Schaltung untersucht werden kann. Je mehr Takte jedoch betrachtet werden, desto größer wird der Zustandsraum der Schaltung und desto aufwendiger wird folglich die Testerzeugung und die Länge der Testmuster. Lange Testmuster führen wiederum zu großen Testprogrammen und langer Testdauer.
Die Wahl der sequentiellen Tiefe für eine gegebene Schaltung erfordert deshalb eine genaue Untersuchung, um denjenigen Wert zu bestimmen, ab dem eine weitere Erhöhung der Tiefe nur noch geringe, oder gar keine, Verbesserung der Haftfehlerabdeckung bewirkt. Je nach Testfall (vgl. Kapitel 2.2) kann dieser Kompromiss mehr zu Gunsten der Laufzeit, Programmgröße oder Fehlerabdeckung optimiert werden.

Eine sequenzielle Schaltung lässt sich für eine feste sequenzielle Tiefe (eine entsprechende Anzahl an Takten) mittels des sog. TFE-Verfahrens (engl. Time Frame Expansion) [WWW06][BA01] in eine kombinatorische Schaltung umwandeln. Durch dieses Verfahren lässt sich folglich das Problem des sequenziellen ATPG auf ein kombinatorisches ATPG abbilden. TFE Verfaren duplizieren den kombinatorischen Block A für jeden Takt, wobei alle Signale, Gatter und Ein- bzw. Ausgänge entsprechend der sequentiellen Tiefe umbenannt werden (vgl. Abb. 2.4). Je nach Algorithmus werden die im aktuellen Takt nicht benötigte Schaltungselemente nicht dupliziert. Es existieren für TFE mehrere denkbare Varianten, die eingesetzt werden können um bestimmte Eigenschaften einer sequenziellen Schaltung zu modellieren.

Sollen die PIs der Schaltung über alle Takte hinweg statisch sein, so kann dies durch TFE modelliert werden, indem die PIs der duplizierten Blöcke nicht umbenannt werden (vgl. Abb. 2.5). Sollen die PIs dagegen in jedem Takt änderbar sein (vgl. Abb. 2.6) können die PIs in jedem Takt umbenannt werden. Auch kann die zeitliche Beobachtbarkeit der POs der zu testenden Schaltung eingeschränkt sein. In Abb. 2.5 wird eine TFE-Variante gezeigt, bei der lediglich die POs des letzten betrachteten Taktes

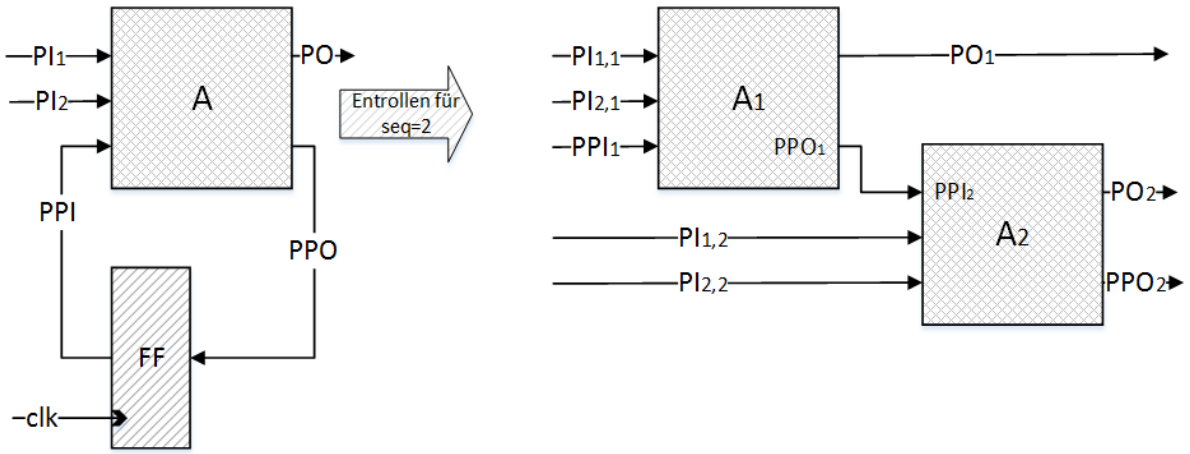


Abbildung 2.4: Abstrakte Anwendung des TFE-Verfahrens.

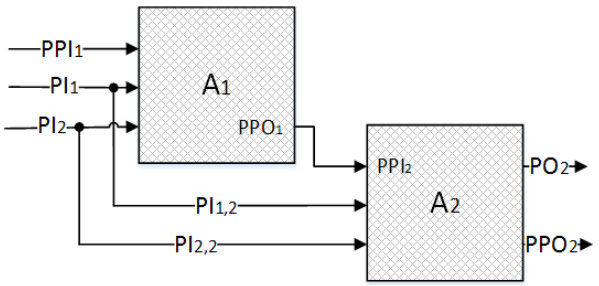


Abbildung 2.5: TFE-Variante mit konstanten PIs und nicht sichtbaren POs der Zwischentakte.

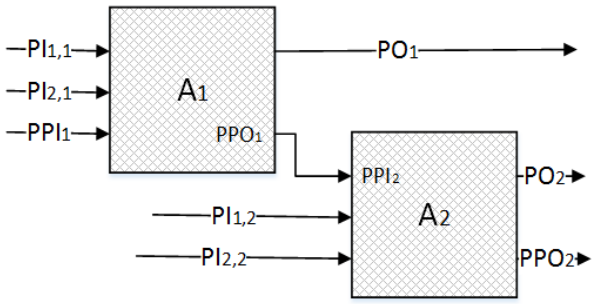


Abbildung 2.6: TFE-Variante mit duplizierten und umbenannten PIs und POs.

beobachtet werden können Abb. 2.6 zeigt dagegen eine Variante, bei der die POs in jedem Takt beobachtbar sind.
Diese verschiedenen TFE-Varianten führen zu den unterschiedlichen Template-Varianten, die im Laufe der Arbeit (in den Kapitels 3.3.2 und 3.3.2) vorgestellt werden.

Abbildung 2.4 zeigt TFE angewandt auf eine abstrakte, sequentielle Schaltung, für eine sequentielle Tiefe von 2. Sie besteht aus dem rein kombinatorischen Block A mit zwei PIs, einem PO und dem Flipflop FF. Der Eingang des Flipflops wird mit PPO (engl. Pseudo Primary Output) oder auch als NS (engl. Next State) bezeichnet und der Ausgang mit PPI (engl. Pseudo Primary Input) oder PS (engl. Present State) [BA01]. Das Zwischenergebnis von A_1, welches in der sequentiellen Schaltung in FF gespeichert ist, wird über PPO_1 an A_2 weitergeleitet. Die so entstehende kombinatorische Schaltung ist funktional äquivalent zur ursprünglichen sequenziellen Schaltung, allerdings um den Faktor der sequenziellen Tiefe vergrößert (im Beispiel verdoppelt).

Ausgangspunkt für ATPG-Programme, bei Anwendung auf eine TFE-Schaltung, ist der letzte betrachtete Takt. Die Bedingungen für eine Propagierung des untersuchten Fehlers an einen PO werden aus der Schaltung und den PIs und PPIs bestimmt. Anschließend werden diese Bedingungen im vorherigen Takt auf Erfüllbarkeit untersucht. Wird kein Konflikt entdeckt, wird dieses Verfahren bis zu Takt 0 wiederholt und ergibt somit einen validen Testvektor. Sollte ein Konflikt entdeckt werden, muss im vorherigen Schritt (engl. Backtracking) eine andere Sensibilisierungsvariante untersucht werden. Dieses Verfahren wird durch eine maximale Anzahl an Versuchen beschränkt. Übersteigt die Untersuchung eines Fehlers dieses Limit (engl. Abort Limit) wird die Untersuchung abgebrochen.

Durch abgebrochene Fehler sinkt die Fehlerabdeckung, da diese Fehler als untestbar eingestuft werden, obwohl sie gegebenenfalls, mit größerem Aufwand, testbar sind.

Funktionale Nebenbedingungen

Grundsätzlich erzeugen ATPG-Programme Testmuster in der Annahme, dass jeder PI der Schaltung in jedem Takt mit einem Testmuster belegt werden kann und die Belegung der einzelnen PIs frei wählbar ist. Ebenso wird angenommen, dass jeder PO in jedem Takt gelesen werden kann. Da dies jedoch abhängig, von Testverfahren und der gegebenen Schaltung, nicht immer umsetzbar ist, muss das ATPG bei der Erzeugung der Muster entsprechend eingeschränkt werden. Diese Einschränkungen werden funktionale Nebenbedingungen (engl. Constraints) genannt.

2.2.4 Design for Testability

Bei DFT (engl. Design For Testability) handelt es sich um spezielle Anpassungen beim Entwurf digitaler Schaltkreise, um deren Testbarkeit zu erhöhen [WWW06]. Die Idee des Prinzips stammt aus den 1970er Jahren und wurde zuerst angewandt um den Test von schwer zu testenden Schaltungsbereichen und die Erzeugung von Testvektoren zu vereinfachen. Essentiell ist dabei das Abwägen zwischen dem Nutzen einzelner DFT-Maßnahmen und den Hardwarekosten, die dadurch verursacht werden. Der Nutzen von DFT-Maßnahmen ist die Erleichterung bzw. Ermöglichung spezieller Testverfahren oder das Erschließen von, andernfalls nicht testbaren, Bereichen der Schaltung und kann oftmals durch eine erhöhte Testabdeckung gemessen werden. Zudem verringern DFT-Maßnahmen im Allgemeinen die Testzeit und senken somit die Testkosten, insbesondere beim Herstellungstest. Viele der im nächsten Abschnitt vorgestellten Testmethoden sind davon abhängig, dass beim Entwurf der Schaltung spezifische DFT-Anpassungen durchgeführt werden.

2.2.5 Testverfahren

Eine zentrale Anforderung an Testmethoden für performanceorientierte Mikrocontroller ist das Testen bei normaler Betriebsfrequenz, da viele Fehler nur bei maximaler Betriebsfrequenz auftreten, so z. B. deep-submicron Verzögerungsfehler (engl. Delay Faults) bei aktuellen Fertigungstechniken [PGSR10]. Testverfahren lassen sich in invasive (engl. Intrusive) und nicht-invasive (engl. Non-Intrusive) unterscheiden. Invasive Methoden bieten die von der Industrie geforderten sehr hohe Testabdeckung und zeichnen sich durch gut dokumentierte und skalierbare Verfahren aus [APGP07b]. Diese Methoden fordern zumeist das Vorhandensein spezieller DFT-Hardware. Diese kann allerdings bei High-End Mikrocontrollers zu nicht-tragbaren Einschnitten bei der Performance oder dem Platzbedarf der Schaltung zur Folge haben [AGP+09].
Typischerweise werden im Rahmen des Herstellungstests ATEs eingesetzt, um gefertigte Chips zu testen. Die Betriebsfrequenz von ATEs steigt jedoch weniger rasch an, als die Frequenz von High-End Mikrocontrollern, es entsteht somit eine sich stetig vergrößernde Lücke, welche die Testzeit einzelner Chips, und somit die Testkosten, erhöht [KPGX07]. Hochgeschwindigkeits-ATEs sind außerordentlich teuer (mehrere Millionen Dollar) und bedingt durch die schnelle Entwicklung schnell veraltet

[KLC+02]. Folglich ist at-speed Testen von High-End Mikrocontrollern mit ATEs häufig unwirtschaftlich [KPGX07].
Eine weit verbreitete Methode stellen scanbasierte Hardwaretests dar. Das Prinzip von Scan ermöglicht die vollständige Beobacht- und Kontrollierbarkeit aller internen Flipflops einer Schaltung [BA01]. Durch dieses Verfahren lässt sich das Initialisierungsproblem für sequenzielles ATPG lösen und so die Anzahl der Testmuster reduzieren. Dies führt zu verringerter Testzeit und damit zu verringerten Testkosten.
Um ein Design scan-kompatibel zu machen werden alle Flipflops der Schaltung durch sog. Scan-Flipflops ersetzt, die in einen spezifizierten Testmodus versetzt werden können. In diesem Modus bilden die Scan-Flipflops ein, oder mehrere, Shiftregister (engl. Scan Register). Die Eingänge und Ausgänge der Scan-Register werden zu PIs bzw. POs der Schaltung. Folglich können, durch das Anlegen geeigneter Eingabefolgen, interne Flipflops in beliebige Zustände versetzt werden (Kontrollierbarkeit) und deren Zustand durch Shiften an dem PO sichtbar gemacht werden (Beobachtbarkeit).
Die Vorteile dieser Methoden bestehen in der sehr hohen erreichbaren Haftfehlerabdeckung (nahezu 100%), guter Skalierbarkeit des Verfahrens und der Möglichkeit ein vorhandenes Design automatisiert in ein scan-kompatibles umzuwandeln. Das Verfahren kann zudem als Selbstest implementiert werden und somit die Verwendung von teuren ATEs vermeiden. Bei einer solchen Implementierung werden lediglich externe Geräte zum initiieren des Testmodus benötigt.
Scan-Verfahren besitzen zwei hauptsächliche Nachteile: Erhöhung der benötigten Chipfläche (engl. Area Overhead) und Verringerung der maximalen Taktfrequenz (engl. Performance Overhead). Der Overhead liegt bei beiden Fällen für gewöhnlich zwischen 5 und 10% [BA01, S. 477]. Ein weiterer Nachteil liegt in der deutlich erhöhten Energieaufnahme während des Testmodus, bedingt durch die hohe Schaltaktivität der Scan-Flipflops. Diese signifikant erhöhte Energieaufnahme führt zu erhöhter Wärmeerzeugung, welche im schlimmsten Fall die Schaltung beschädigen kann. Mit den von Wunderlich und Gerstendörfer [GW00] vorgeschlagenen Anpassungen lässt sich dieser Nachteil jedoch erheblich reduzieren.

2.3 Software-basierter Selbsttest

SBST ist ein Testverfahren, welches weder High-Speed ATEs noch spezielle DFT-Maßnahmen benötigt. Dazu werden programmierbare Komponenten einer Schaltung ausgenutzt, um Testmuster mithilfe von Testprogrammen an die zu testende Komponente (engl. Device Under Test, DUT) anzulegen. Die Ausgabewerte des DUT können anschließend mit den Soll-Werten der Testmuster abgeglichen werden. Da die Ausführung dieser Testprogramme zu den normalen Betriebsumgebung der Schaltung stattfindet, läuft SBST stets unter normaler Betriebsfrequenz und innerhalb des Gesamtsystems ab.
Da bei SBST auf DFT-Maßnahmen verzichtet werden kann, wird weder die Chipfläche des Systems erhöht, noch dessen Performance beeinflusst, noch die Energieaufnahme während des Tests gesteigert. Zusätzlich bietet SBST die interessante Möglichkeit das DUT während des Betriebs (engl. Online-Test) zu testen und so die Verlässlichkeit des Gesamtsystems zu erhöhen. Da es sich um eine nicht-invasive Testmethode handelt lassen sich SBST Methoden in allen Testfällen einsetzten und fertige Testprogramme sind jederzeit veränderlich, was den SBST Ansatz-stark wiederverwendbar macht [PGSR10].

2.3.1 Allgemeine Durchführung

SBST-Verfahren beschäftigen sich mit der Erzeugung und Durchführung von Test-Programmen, die sich den Instruktionen des Mikroprozessors bedienen, um Fehler zu aktivieren und sichtbar zu machen. Die Ausführung von SBST-Programmen ist rein funktional. Allgemein gliedern sich die Durchführung von SBST-Verfahren in die folgenden drei Schritte:

1. Laden des Testprogramms in den Systemspeicher des DUT. Dazu können existierende Programmier- und Debug-Schnittstellen genutzt werden. Da dieser Schritt dabei nicht unter Betriebsfrequenz ablaufen muss, kann der Gebrauch von schnellen und teuren ATEs vermieden werden. Für periodische Selbsttests kann das compilierte Testprogramm alternativ bereits im Systemspeicher des Mikrocontrollers vorliegen.
2. Ausführen des Testprogramms durch den Mikroprozessor des Systems. Ein Testprogramm setzt sich dabei aus mehreren Testmustern zusammen. Jedes Testmuster besteht dabei aus einem Konfigurationsblock, der die Komponenten in einen definierten Zustand versetzt, und einem Funktionsblock der eine bestimmte Funktionalität der Komponente aktiviert. Beobachtbare Ausgabewerte werden während der Ausführung, für den späteren Abgleich, in den Speicher geladen. Dieser Schritt läuft unter normaler Betriebsfrequenz der Komponente ab.
3. Auslesen der Testresultate. Die im vorherigen Schritt gespeicherten Daten können nun durch ein Testgerät ausgelesen und mit den Sollwerten der Testmuster abgeglichen werden um mögliche Fehler aufzudecken. Alternativ kann auch der Mikroprozessor des Systems diesen Schritt übernehmen. Für periodische Selbsttest können Sollwerte im Systemspeicher abgelegt werden und mit den gespeicherten Ausgabewerten des vorherigen Schritts abgeglichen werden.

2.3.2 Erzeugen von Testprogrammen

Das Erzeugen von Testprogrammen ist eine komplexe Aufgabe, die stark von dem Testfall und den vorhandenen Informationen über die zu testende Schaltung abhängt. Während die Testdurchführung rein funktional abläuft, kann für die Erzeugung der Programme durchaus auf strukturelle Informationen zurückgegriffen werden. Dementsprechend unterschieden sich funktionale SBST-Methoden von strukturellen lediglich durch die Art der genutzten Informationen, die während der Erzeugung der Testprogramme zum Einsatz kommen, nicht jedoch bei der Ausführung - diese läuft immer funktional ab.

Funktionaler SBST

Funktionale Methoden nutzen zur Testprogrammerzeugung lediglich Informationen über den Instruktionssatz (engl. Instruction Set Architecture, ISA) des Prozessors und sind folglich selbst bei Komponenten nutzbar, bei denen keine strukturellen Informationen vorhanden sind. Zentrales Problem der rein funktionalen Ansätze ist, wie bereits in Abschnitt 2.2.2 besprochen, der Mangel an strukturellen Informationen, die eine Angabe der Testabdeckung struktureller Fehler unmöglich macht [PGSR10].

Funktionale Methoden nutzen entweder funktionale ATPG-Programme, randomisierte oder Feedback-basierte Verfahren zur Erzeugung von Testmustern [PGSR10].

Struktureller SBST

Strukturelle SBST-Methoden nutzen Beschreibungen der zu testenden Schaltung auf Register-Transfer-Ebene (engl. Register Transfer Level, RTL), in Form von Hardwarebeschreibungssprachen (engl. hardware description language, HDL), oder synthetisierte Gatternetzlisten. Die Nutzung von Beschreibungen auf Gatterebene erlaubt das Aufstellen der Haftfehlerliste (vgl. Abschnitt 2.2.1) und wird deshalb von dem Großteil der nachfolgend vorgestellten Methoden verwendet.

Grundsätzlich läuft die Erzeugung von strukturellen Testprogrammen dabei in zwei Schritten ab:

1. Erzeugen von Testmustern. Damit ein SBST-Programm einen Fehler an einem logischen Block aufdecken kann, muss mithilfe von Instruktionen ein geeigneter Testvektor an das DUT angelegt werden, sodass an dem fehlerhaften Block ein abweichender Wert erzeugt wird (Fehleraktivierung). Dieser muss anschließend an eine beobachtbaren Bereich der Schaltung propagiert wird. Da nahezu jede größere Schaltung sequentielle Komponenten enthält [BA01], ist das Problem der Testmustererzeugung für SBST-Programme ein Spezialfall des sequentiellen, strukturellen ATPG-Problems (vgl. Kapitel 2.2.3), erweitert um die funktionalen Nebenbedingungen von SBST. Die wichtigste funktionale Nebenbedingung von SBST ist die Einschränkung, dass Testvektoren sich durch eine oder mehrere Instruktionen, die Teil der ISA des betrachteten Systems sind, an das DUT anlegen lassen müssen.
2. Erzeugen des Testprogramms. Die im vorherigen Schritt erzeugten Testmuster werden anschließend zu einem Testprogramm zusammengefasst. Für diesen Zweck werden gewöhnlich parametrisierte Vorlagen (engl. Templates) verwendet, in die sich die erzeugten Testmuster einsetzen lassen. Der Einsatz von parametrisierten Templates erleichtert die Umwandlung struktureller Testvektoren in ein ausführbares, binäres Testprogramm. Sollten die Templates weitere funktionale Nebenbedingungen für die Erzeugung der Testmuster aufstellen, so müssen diese von dem ATPG-Programm während der Testmustererzeugung bekannt sein. Sieht ein Template z. B. nur in jedem zweiten Takt ein schreibenden Zugriff auf die PIs vor (da im nächsten Takt die korrelierenden Werte der POs ausgelesen werden), muss diese Einschränkung dem ATPG-Programm als Constraint übergeben werden.

Im Folgenden werden einige strukturelle SBST-Methoden zur Erzeugung von Testmustern besprochen. Nach Gizopoules et. al. [PGSR10] lassen diese sich in Hierarchische und RTL-Methoden einteilen.
Hierarchische SBST-Methoden nutzen die hierarchischer Struktur des DUT aus, um Testprogramme nach einem Teile-und-Herrsche Prinzip (engl. divide and conquer) zu erzeugen [GVA06][LJ07][CRRD03][WWC+06][CWLG07]. Im Divide-Schritt wird dazu das DUT in dessen einzelne Module aufgeteilt. Anschließend werden Testvektoren für jedes Teilmodul mithilfe eines strukturellen ATPG-Programms erzeugt. Die das Modul umgebende Schaltung wird auf einem höheren Abstraktionslevel miteinbezogen, oder mithilfe einer Reihe von ATPG-Constraints abgebildet. Durch diese Abbildung wird die Wahrscheinlichkeit erhöht Testvektoren (je Teilmodul) zu erhalten, die sich mittels Instruktionen an das Gesamtsystem anlegen lassen. Diejenigen Testvektoren die sich

mittels von Befehlen der ISA an die Gesamtschaltung anlegen lassen, werden im Conquer-Schritt zum SBST-Testprogramm zusammengefasst .

Eine ganze Reihe von Verfahren [CCRS00][KMT+08][BSS+07] beschäftigt sich mit der Entwicklung und Anwendung von Metriken auf RTL-Ebene, da komplexe Schaltungen auf Gatternebene einen erheblichen Rechenaufwand für die Erzeugung von Testmustern per ATPG bedeuten. Die untersuchten Metriken sollen einen Kompromiss zwischen möglichst hoher Abstraktion einerseits und hoher Aussagekraft, bezüglich bestehender Fehlermodellen anderseits, bieten. Vorgeschlagene Metriken messen z. B. den abgedeckten Prozentsatz der Verzweigungen (engl. Branch Coverage), Ausdrücke (engl. Statement Coverage), Bedingungen (engl. Condition Coverage) oder die Anzahl an Signal- oder Registeränderungen (engl. Toggle Coverage) einer gegebenen HDL-Beschreibung durch ein gegebenes Testprogramm [KMT+08][BSS+07]. Große kombinatorische Komponenten korrelieren jedoch schlecht mit RTL Metriken, was ihren Nutzen für viele Komponenten, oder Teilmodule, einschränkt [BSS+07].

Einschränkungen von SBST

Neben den funktionalen Nebenbedingungen des SBST-Paradigmas können je nach Anwendungsfall und Testsystem weitere Einschränkungen hinzukommen, die ein Abwägen zwischen der erzielbaren Fehlerabdeckung und den zusätzlichen Anforderungen notwendig machen. Soll das Testprogramm z. B. während des Herstellungstest Anwendung finden, so ist die Laufzeit des Testprogramms von vorrangiger Bedeutung, dagegen im Fall der Rückläuferanalyse nicht entscheidend. Die Größe des verbauten Systemspeichers kann ebenfalls ein einschränkendes Kriterium für die maximale Länge der Testprogramme darstellen. Ebenfalls können SBST-Verfahren auf den Energieverbrauch der entwickelten Testprogramme hin optimiert werden [Zho09]. Sind dagegen keine strukturellen Informationen über das DUT verfügbar, so können strukturelle SBST-Methoden nicht eingesetzt werden und es muss auf funktionale Methoden zurückgegriffen werden.

2.3.3 Anwendung auf Peripheriekomponenten

Tests für Peripheriekomponenten zu entwickeln stellt eine anspruchsvolle Aufgabe dar, da die Kontrollier- und Beobachtbarkeit interner Signale gegenüber Mikroprozessoren im Allgemeinen deutlich niedriger ist [BSS+07][GHS+12]. Hinzu kommt dass die Kommunikationsgeschwindigkeit meist erheblich langsamer ist als die Taktrate des Mikroprozessors, was zu langen Wartezeiten und somit zu langer Gesamttestzeit führt [AGP+09].
Generell gehen SBST-Methoden für Peripheriekomponenten davon aus, dass alle anderen Komponenten des Systems, insbesondere betrifft dies die für SBST benötigten Komponenten Mikroprozessor und Systemspeicher, fehlerfrei sind. Dies kann entweder durch Hardware-basierte Testmethoden oder durch die vorherige Anwendung von SBST-Methoden auf den Mikroprozessor und den Speicher erreicht werden.
Um Kommunikationskomponenten erfolgreich zu testen muss eine protokollgerechte Kommunikation stattfinden [AGP+09]. Es gibt im wesentlichen zwei Ansätze diese Kommunikation zu ermöglichen: externe Testgeräte oder die Nutzung einer Loopback-Struktur. Die erste Variante sieht die Nutzung eines ATEs vor, welches als Kommunikationspartner für das DUT fungiert. Dabei ist zu beachten,

dass das ATE zumindest dieselbe Betriebsfrequenz wie eine entsprechende Kommunikationskomponente besitzen muss. Während dieser Ansatz für Herstellungstest oder Rückläuferanalyse durchaus geeignet ist, kann er keinesfalls für Feldtests genutzt werden. Als zweite Möglichkeit kann deshalb ein Loopback-Mechanismus implementiert werden, der es ermöglicht gesendete Werte anschließend auszulesen und mit den ursprünglichen Daten auf Übertragungsfehler abzugleichen. Abstrakt gesehen wird bei einem Loopback die Sendeeinheit der Komponente mit der Empfangseinheit verbunden, sodass gesendete Werte von der Schaltung empfangen werden können. Diese Variante bietet die Möglichkeit von Feldtests oder periodischen Selbsttests, allerdings macht sie, abhängig vom Kommunikationsprotokoll, gewisse DFT-Maßnahmen erforderlich. Durch einen solchen Loopback kann folglich die Beobachtbarkeit und die Kontrollierbarkeit von Kommunikationskomponenten stark erhöht werden. In vielen Komponenten sind bereits Loopback-Mechanismen vorgesehen um Programmierer zu unterstützten [AGP+09], was die Notwendigkeit von zusätzlichen Designanpassungen beseitigt. Je nach Kommunikationsprotokoll kann ein Loopback einfach implementiert werden, z. B. bei UART oder benötigt bestimmte Hardwarezusammensetzungen, z. B. hierarchische Protokolle benötigen sowohl Master,- als auch Slavekomponenten um ein Loopback zu erreichen.

Ein früher, rein funktionaler Ansatz von Jayaraman et. al. [SA98] erreichte eine relativ geringe Haftfehlerabdeckung von 67,89% [SA98] auf einer einfachen UART-Komponente und zeigte einige der zentralen Problemstellungen für SBST auf Kommunikationsperipheriekomponenten auf.

Gizopoulos et. al [APGP07b] präsentierten eine systematische, deterministische Methodik welche eine sehr gute Fehlerabdeckung, bei kleiner Testprogrammgröße, erreicht. Die Methode erfordert jedoch manuellen Aufwand zur Testmustererzeugung und ausführliche Kenntnisse über interne Funktionen der Komponenten. Die Methode erfordert die Identifikation der zentralen Teilmodule der Kommunikationskomponente und pro Modul das Aufstellen von Test-Templates. Je ein Template wird für die verschiedenen Betriebsmodi der Komponente, für die Anbindung des DUT an den Systembus, für jede FIFO-Struktur (engl. First In First Out) und für Fehlerbehandlungslogik erzeugt. Dabei finden sich FIFO Elemente zumeist bei seriellen Kommunikationskomponenten im Sende- bzw. Empfangsteil. Diese Templates können abschließend zu einem Gesamttestprogramm zusammengefasst werden.
Experimentelle Anwendung dieser Methode auf einer UART und einer Ethernet Kommunikationskomponente erreichen eine Haftfehlerabdeckung von 93,74% und 91,7% [APGP07a].

Ein weiterer Ansatz von Reorda et. al. [BSS+07] nutzt einen evolutionären Algorithmus um mithilfe von Metriken auf HDL-Ebene sog. Testblöcke zu erzeugen und iterativ zu verbessern, die in der Lage sind das DUT zu konfigurieren und zu testen. Die erzielten Ergebnisse sind durch die Verwendung des evolutionären Algorithmus heuristisch, allerdings können durch dieses Verfahren sehr schnell Testprogramme erzeugt werden. Die zur Optimierung der Testblöcke vorgeschlagenen Metriken - Statement, Branch, Condition, Expression und Toggle Coverage - bestimmen die prozentuale Abdeckung gegebener Blöcke anhand der HDL-Beschreibung der Komponente.
Eine experimentelle Anwendung auf eine UART und Ethernet Komponenten erreichte eine Haftfehlerabdeckung von 86,35% und 86,57% [AGP+09]. Die verwendeten Komponenten waren identisch zu jenen die bei der experimentellen Auswertung der Methode von Gizopoulos et. al. Anwendung fanden.

Diese beiden Methoden wurden in [AGP+09] zu einer zusammengefasst. Dazu werden die funktionalen Informationen, die in [APGP07b] eingeholt werden, genutzt um den Erzeugungsprozess des evolutionären Algorithmus zu verbessern. Der evolutionäre Algorithmus von [BSS+07] wird einzeln

für die in [APGP07b] beschriebenen Teilmodule der jeweiligen Kommunikationskomponente angewandt. Dies verringert den Suchraum für den Algorithmus erheblich, was zu effizienteren Testblöcken innerhalb weniger Iterationen führt [AGP+09].
Testergebnisse dieser hybriden Methode zeigen für UART und Ethernet eine Haftfehlerabdeckung von 93,13% und 91,70% [AGP+09]. Insbesondere zeigen die Ergebnisse eine Verbesserung bei der Erzeugung der Testblöcke für FIFO-Strukturen und eine erheblich verringerten Aufwand zur Erzeugung der Testprogramme (verglichen mit dem Aufwand der deterministischen Methode von [APGP07b]).

Systemperipheriekomponenten sind im Allgemeinen noch komplexer zu testen als Kommunikationskomponenten, da deren Beobacht- und Kontrollierbarkeit äußerst gering ist [GHS+12] und Fehler in diesen Komponenten die Ausführung des Testprogramms beeinflussen kann. Zudem erfordern sie sorgfältige Konfiguration weiterer Systemkomponenten.
Die Methode [DBG03] zielt auf das Testen von Randfällen (engl. Corner Cases) von Systemperipheriekomponenten ab. Dazu wird das DUT in eine abstrakte Zustandsmaschine übersetzt und mithilfe eines abdeckungsorientierten Algorithmus durchlaufen. Anschließend können aus der abstrakten Zustandsfolge die Testvektoren erzeugt werden. Dabei dient die Zahl der durchlaufenen Zustände als high-level Metrik.
Ein weiterer Ansatz [GHS+12] für Systemperipheriekomponente stellt aus Informationen über die Zielkomponente einen formalen Konfigurationsgraphen (engl. Configuration Graph) auf. Aus diesem lassen sich, mithilfe eines Rundlauf-Algorithmus (engl. Visiting Algorithm), Konfigurationspfade berechnen, welche die Konfigurationen der einzelnen Komponenten bestimmen. Für jeden solchen Pfad kann ein Test-Template erzeugt werden. Diese garantieren eine Konfiguration, die sich mithilfe von Prozessorinstruktionen anlegen lässt, für alle Systemkomponente die benötigt werden um ein bestimmtes Verhalten der Systemperipheriekomponente zu beobachten.

Im Gegensatz zu den heuristischen Methoden von [AGP+09] und [BSS+07] wird in dieser Arbeit eine deterministische, strukturelle SBST-Methode vorgestellt und die Wirksamkeit von einfachen Maßnahmen zur Verbesserung der Kontrollier- und Beobachtbarkeit auf die erreichbare Haftfehlerabdeckung betrachtet. Ebenfalls wird der Einfluss von funktionalen Nebenbedingungen einer Schaltung und von zwei unterschiedlichen Template-Varianten auf die Haftfehlerabdeckung, Testdauer und Programmgröße untersucht.

3 Implementierung

Im folgenden Kapitel wird zunächst in Unterkapitel 3.1 das genutzte Testsystem besprochen und auf die Implementierung der zu testenden Kommunikationskomponente eingegangen. Die Abschnitte 3.1.1 und 3.1.2 stellt die beiden Teilkomponenten (Master und Slave) der I^2C-Peripheriekomponente vor. Zudem wird in Unterkapitel 3.1.4 das Mapping der I^2C-Ports auf MMI/O Adressen besprochen.
Im zweiten Abschnitt (3.2) werden die vorgenommenen Maßnahmen vorgestellt, welche die Beobacht- und Kontrollierbarkeit der Kommunikationskomponente erhöhen und somit die strukturelle Fehlerabdeckung verbessern. Dieser ist in zwei Teile gegliedert: der erste Abschnitt (3.2.1) bespricht die Implementierung des Loopback-Mechanismus während im Unterkapitel 3.2.2 diejenigen interne Signale vorstellt werden, die zur Verbesserung der Beobachtbarkeit dem System sichtbar gemacht werden. Zuletzt werden im Abschnitt 3.2.3 die untersuchten Kombinationen der vorgestellten Maßnahmen vorgestellt, deren Ergebnisse im Kapitel 4 besprochen werden.
In Unterkapitel 3.3 wird schließlich die Anwendung der strukturellen SBST Methode auf die vorgestellte Kommunikationsperipheriekomponenten gezeigt. Abschnitt 3.3.1 beschreibt die Erzeugung der Testmuster mittels eines sequenziellen ATPG-Programms. Zuletzt wird im Unterkapitel 3.3.2 kurz erklärt, welche Schritte unternommen werden müssen, um ein fertiges Testprogramm aus den erzeugten Testmustern zu erzeugen. Dazu werden zwei unterschiedliche Test-Template Varianten besprochen.

3.1 Implementierung des Testsystems

Als Testumgebung für diese Arbeit wird der Open-Source Mikroprozessor miniMIPS[1] mit einem kompatiblen RAM-Modul benutzt. Bei miniMIPS handelt es sich um einen 32-Bit RISC Mikroprozessor mit fünfstufiger Pipeline, Bypassing-Komponente und Sprungvorhersage. Die Synthese der miniMIPS-Referenz erreicht eine Taktrate von 50MHz. Als zu testende Kommunikationsperipheriekomponente wird ein einfacher I^2C-Master und ein dazu kompatibler Slave verwendet, die jeweils per MMI/O Controller an den Prozessor angeschlossen sind.

Abbildung 3.1 zeigt ein Blockschaubild des Testsystems mit besonderem Augenmerk auf der Schnittstelle der I^2C-Komponente. Prozessor, Speichermodule und MMI/O Controller sind dabei über einen Systembus verbunden, welcher dem Speicherprotokoll des miniMIPS entspricht. Zusätzlich zeigt die Abbildung den Clock Divider, welcher durch Teilung der Systemtaktfrequenz den langsameren Peripherietakt erzeugt. Durch diese Teilung lässt sich die Übertragungsrate der Komponente einstellen. Der aktuelle Pegel des Peripherietaktes kann mittels MMI/O Zugriff auf des $sync$-Signal abgefragt werden und dient der Synchronisierung von Software und Peripheriekomponente.

[1] http://opencores.org/project,minimips

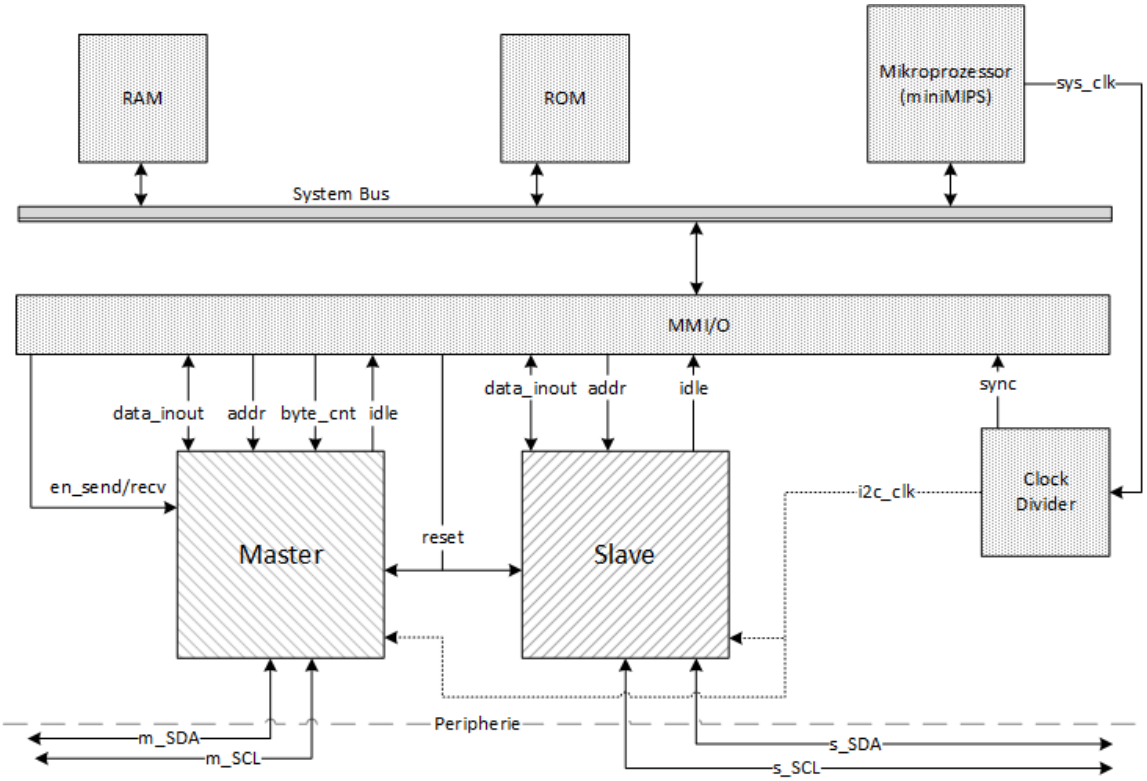


Abbildung 3.1: Blockschaubild des Testsystems.

3.1.1 I^2CMaster

Die Masterkomponente ist entsprechend der NXP Spezifikation [NXP14] implementiert. Die in Unterkapitel 2.1.2 genannten Erweiterungen des I^2C-Protokolls sind nicht implementiert.

Schnittstelle

Das Schaubild 3.1 zeigt die Schnittstelle der Masterkomponente. Das Taktsignal (*i2c_clk*) der Komponente wird durch den Clock Divider erzeugt, während das Reset-Signal (*reset*) softwareseitig gesetzt werden kann. Die Kontrolleingänge *en_send* und *en_recv* spezifizieren (exklusiv) die gewünschte Operation (Senden bzw. Empfangen), zudem führt das Setzen eines der Signale zum Start der gewählten Übertragung. Die Anbindung der Komponente an den I^2C-Bus geschieht über die Signale *m_SDA* und *m_SCL* die den I^2C-Busleitungen SDA und SCL entsprechen. Mit *data_inout* kann auf das interne Datenregister zugegriffen werden, während der *addr* Eingang die 7-Bit I^2C-Adresse des Ziels der Übertragung und *byte_cnt* die Menge der zu übertragenden Bytes spezifiziert. Die Implementierung kann bis zu 8 Bytes infolge übertragen, anschließend muss eine neue Kommunikation gestartet werden. Das *idle*-Signal dient als Bereitschaftssignal für den Systembus. Ein logischer Wert 1 auf dieser

Leitung signalisiert das Erreichen des Startzustands, nicht jedoch eine erfolgreich abgeschlossene Kommunikation.

Zustandsautomat

Abbildung 3.2 zeigt den Zustandsautomaten der Masterkomponente. Die Masterkomponente erzeugt gemäß [NXP14] das I^2C-Taktsignal auf der SCL-Leitung.
Der Startzustands des Systems erwartet das Setzen von *en_send* bzw. *en_recv* um mit einer Übertragung zu beginnen. Während sich die Komponente im *IDLE*-Zustand befindet, hat das *idle*-Signal den Wert 1. Nachdem *en_send* oder *en_recv* gesetzt sind, durchläuft die Komponente die *START*-Zustände und generiert dabei die in Abschnitt 2.2 beschriebene Startbedingung. Anschließend wird mit der Übertragung des ersten Bytes begonnen, welches sich aus der an *addr* angelegten Zieladresse (7 Bit) und dem R/W-Bit zusammensetzt. Das R/W Bit geht aus der gewählten Operation hervor - *en_recv* steht für eine Leseoperation und ist als R/W $= 1$ definiert, *en_send* für eine Schreiboperation und erzeugt R/W $= 0$. Die Übertragung des (Adress-)Bytes läuft dabei nahezu identisch zur Übertragung eines Datenbytes ab, nutzt folglich auch dieselben Zustände und unterscheidet sich lediglich durch das Setzen eines internen Statussignals.
Die Abfolge der Zustände zur Übertragung eines Bytes besteht aus den beiden *SEND*- und den *GET_D_ACK*-Zuständen (im Folgenden mit Sendeschleife bezeichnet und analog Empfangsschleife für den Empfang). Die *SEND*-Zustände werden dabei achtmal durchlaufen und das interne Datenregister vom höchstwertigen zum niederwertigsten Bit ausgelesen. Das aktuell gelesene Bit wird über die SDA-Leitung übertragen. Anschließend empfängt der Master das ACK-Bit in den beiden Zuständen *GET_D_ACK*. Wird ein NACK empfangen, geht der Master in den Fehlerzustand *FAIL* und bricht die Übertragung durch die in Kapitel 2.2 beschriebene Stoppbedingung ab. Ist das interne Adressierung-Statussignal (*addr_flag*) gesetzt und wird ein ACK empfangen, beginnt die Komponente die eigentliche Datenübertragung. Entsprechend dem R/W-Bit wird in die Sende- oder Empfangsschleife gesprungen.

Dabei läuft der Transmittermodus ähnlich zur Übertragung der Adresse und des R/W-Bits ab. Nach jeder - mit einem ACK quittierten - Sendeschleife wird zusätzlich der Byte-Zähler dekrementiert und entweder die Kommunikation beendet, durch einen Übergang in den STOPP-Zustand, oder das nächste Byte gesendet - durch Rücksprung zu *SEND_1*.

Die Empfangsschleife besteht aus den *READ*-Zuständen die, vergleichbar mit den *SEND* Zuständen, byteweise durchlaufen werden. Das aktuelle Bit wird dabei von der SDA-Leitung in das interne Datenregister geschrieben. Nach jedem Durchlauf wird der Byte-Zähler dekrementiert. Anschließend sendet der Master in den Zuständen *SEND_D_ACK* über SDA eine logische 0 - das Daten ACK gemäß Kapitel 2.1.2 - wenn weitere Bytes gelesen werden sollen, oder eine logische 1 - ein NACK - um die Übertragung des Slaves nach dem letzten Byte abzubrechen. Nach dem übertragenen NACK wird die Kommunikation ordnungsgemäß durch den Übergang in den *STOP*-Zustand - welcher die I^2C-Stoppbedingung erzeugt - beendet.

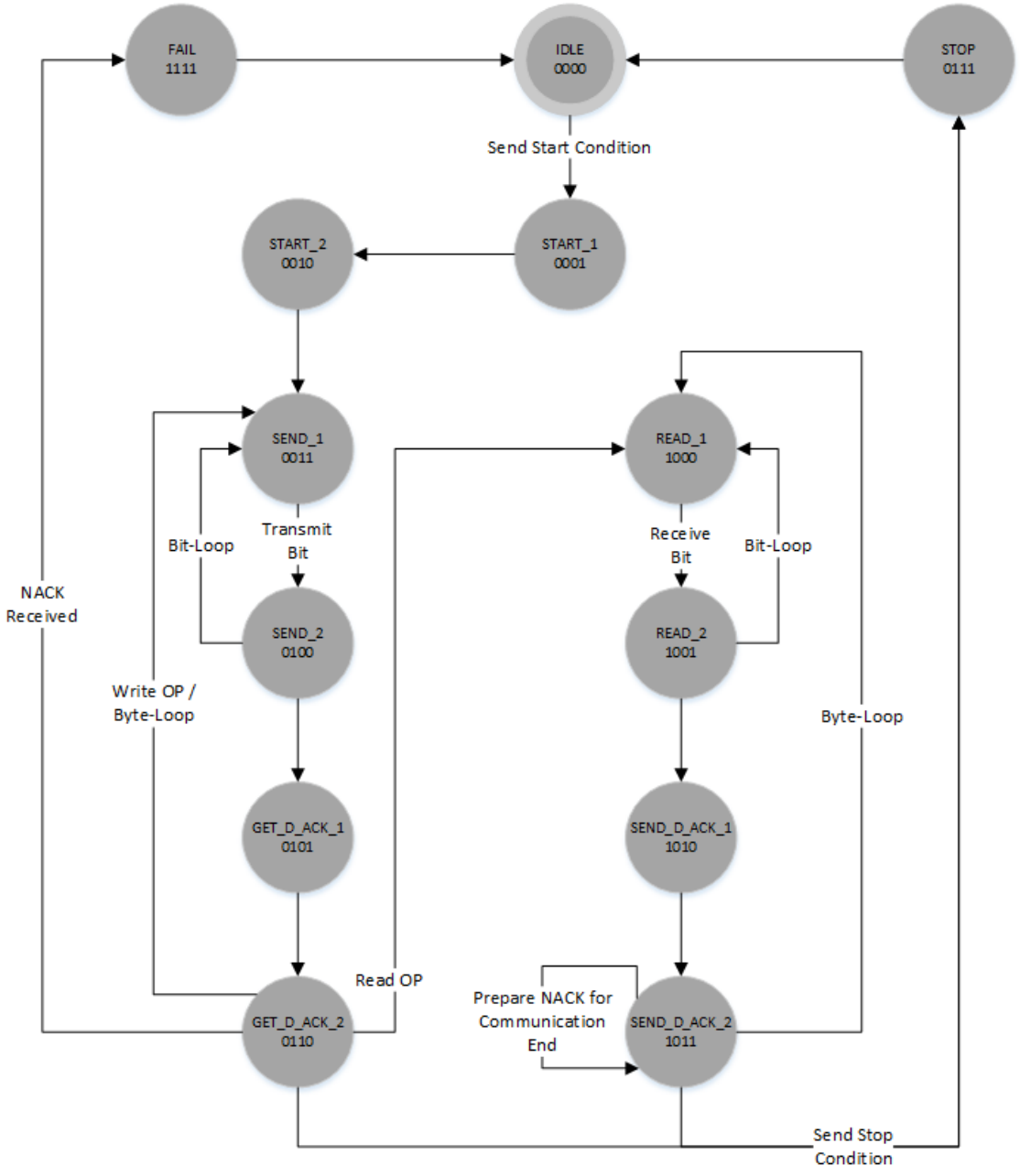


Abbildung 3.2: Kommentierter Zustandsautomat der I^2C-Master Komponente.

3.1.2 I^2CSlave

Die Slavekomponente ist ebenfalls entsprechend der NXP-Spezifikation [NXP14] implementiert.

Schnittstelle

Das Taktsignal (*i2c_clk*) der Komponente wird durch den Clock Divider erzeugt, während das Reset-Signal (*reset*) softwareseitig gesetzt werden kann. Die Anbindung der Komponente an den I^2C-Bus geschieht über die Signale *s_SDA* und *s_SCL* die den I^2C-Busleitungen SDA und SCL entsprechen. *data_inout* erlaubt den Zugriff auf das interne Datenregister, während mithilfe der *addr* Eingangs die 7-Bit I^2C-Adresse der Komponente - softwareseitig - festlegt werden kann. Das *idle*-Signal dient als Bereitschaftssignal für den Systembus. Ein logischer Wert 1 auf dieser Leitung signalisiert den Startzustands, nicht jedoch eine erfolgreich abgeschlossene Kommunikation.

Zustandsautomat

Abbildung 3.3 zeigt den Zustandsautomaten der Slavekomponente. Der Startzustand der Komponente wartet auf den Empfang der I^2C-Startbedingung und löst anschließend den Zustandsübergang nach *READ_1* aus um die adressierte I^2C-Adresse zu empfangen. Dabei lauft der Empfang der Zieladresse identisch zum Empfang von Daten ab. Folglich können dieselben Zustände genutzt werden, wobei ein internes Statussignal (*addr_flag*) die beiden Fälle unterscheidet.
Die Empfangsschleife besteht aus den zwei *SEND*-Zuständen die achtmal durchlaufen werden und dabei das interne Datenregister vom höchstwertigen zum niederwertigsten Bit beschreibt. Das aktuelle Bit wird von der SDA-Leitung gelesen und in das Register an die Position eingetragen. Nach dem Durchlaufen der Sendeschleife befindet sich die Slavekomponente im Zustand *SEND_D_ACK*. Ist *addr_flag* gesetzt wird die empfangene Adresse (die sieben höchstwertigen Bits) mit der durch *addr* zugewiesenen I^2C-Slaveadresse abgeglichen. Sind beide Adressen identisch, wird ein ACK über SDA gesendet und entsprechend dem R/W-Bit (dem niederwertigsten empfangenen Bit) in die Sende - oder Empfangsschleife gesprungen. Stimmen die Adressen nicht überein wird ein Zustandsübergang in den Startzustand ausgelöst. Ist *addr_flag* dagegen nicht gesetzt befindet sich die Komponente in der Daten-Empfangsschleife. Bei Erreichen des *SEND_D_ACK* Zustands generiert die Komponente ein ACK, um den Empfang der Daten zu bestätigen. Danach verbleibt die Komponente in diesem Zustand bis zum Beginn der nächsten Übertragung - es folgt ein Zustandsübergang nach *READ_1* - oder bis zum Empfang der I^2C-Stoppbedingung. Diese löst einen Zustandsübergang nach *IDLE* aus.
Die Sendeschleife besteht aus den *SEND*-Zuständen die, äquivalent zur Empfangsschleife, achtmal durchlaufen werden und somit ein Byte senden. Das aktuelle Bit wird dabei auf die SDA-Leitung gelegt. Nach jedem Durchlauf wird der Byte-Zähler dekrementiert. Anschließend erwartet der Slave im Zustand *GET_D_ACK* das Daten-ACK des Master, übertragen via SDA. Wird ein ACK empfangen, folgt das nächste Byte, andernfalls (NACK-Empfang) wird die Kommunikation abgebrochen und ein Zustandsübergang nach *IDLE* vorgenommen.

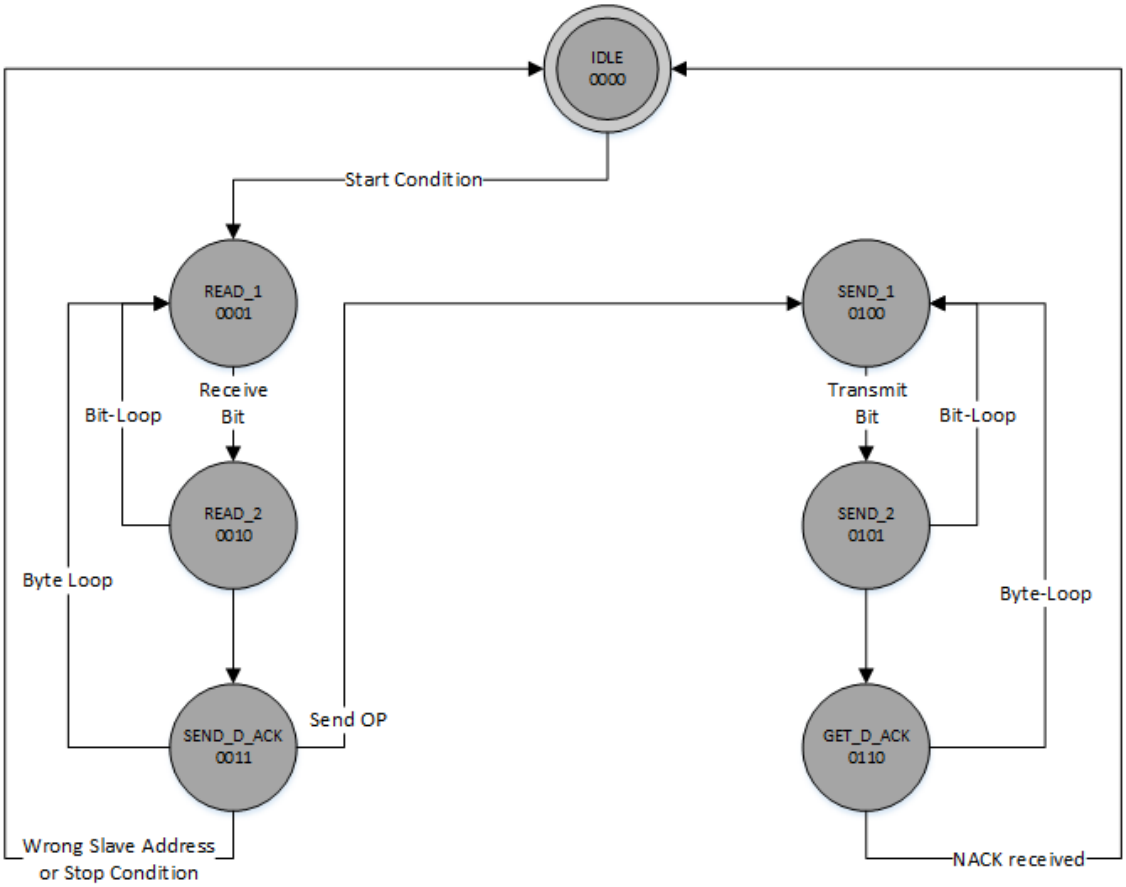


Abbildung 3.3: Kommentierter Zustandsautomat der I^2C-Slave Komponente.

3.1.3 Taktteilung

Kommunikationskomponenten werden gewöhnlich mit einem deutlich langsameren Takt (Peripherietakt) betrieben, als z. B. der Mikroprozessor des Systems (Systemtakt). Damit Kommunikationskomponenten dennoch synchron zum System arbeiten, können Taktteiler (engl. Clock Divider) eingesetzt werden. Dies vermeidet zudem zusätzlich benötigte Logik um ein neues Taksignal zu generieren.
Taktteiler zählen den Systemtakt mit und erzeugen, entsprechend einem gegebenen Teiler, den Peripherietakt.
Im Fall von Kommunikationskomponenten kann durch die Anpassung dieses Multiplikators die Übertragungsrate gewählt werden.

3.1.4 Memory Mapped Input Output

Das Mapping der Ports des DUT auf Speicheradressen beeinflusst die Anzahl der nötigen Speicherzugriffe die für das Anlegen bzw. Auslesen von Testvektoren benötigt werden. Werden die n Ports

einer betrachteten Komponente jeweils auf eine MMI/O Adresse abgebildet, so werden *n* Speicheroperationen benötigt, um alle Eingangssignale der Schaltung zu setzen. Werden jedoch mehrere Ports auf dieselbe Adresse abgebildet kann die Zahl der benötigten Speicherzugriffe erheblich gesenkt werden. Deshalb werden möglichst viele Interface-Signale auf eine einzige MMI/O Adresse gemappt, wobei die Anzahl der pro Speicheradresse zusammengefassten Ports durch die Wortbreite des Systems beschränkt ist.

Für die Anbindung der Master- und Slavekomponente werden zwei MMI/O Adressen für Eingabe und eine weitere für Ausgabewerte benötigt.

MMI/O Mapping

Die Abbildung 3.4, 3.5 und 3.6 zeigt die Bit-Verteilung von den MMI/O Adressen auf das I^2C-Interface (vgl. Kapitel 3.1).
Für die Anbindung der in im nächsten Kapitel vorgestellten Maßnahmen zur verbesserten Beobachtbarkeit wird eine weitere MMI/O Adresse benötigt. Abbildung 3.7 zeigt das Bit-Mapping der einzelnen Maßnahmen innerhalb dieser MMI/O Adresse. Zur Verringerung der benötigten MMI/O Zugriffe wird das lb_{enable}-Signal, welches die Loopback-Komponente aktiviert, gemeinsam mit dem Interface der Slavekomponente, auf ein Datenwort abgebildet.

3.2 Anpassungen zur Verbesserung der Haftfehlerabdeckung

Abbildung 3.8 zeigt die vorgenommen Anpassungen des Systems im Überblick. Um die Durchführung der Experimente zu vereinfachen, wurde eine Schaltung implementiert in der alle vorgeschlagenen Maßnahmen zur Verbesserung der Kontrollier- und Beobachtbarkeit (blau markiert) eingefügt sind (vgl. Abbildung 3.8).
Die rot eingezeichneten Steuersignale und Gatter ermöglichen eine Auswahl der verwendeten Maßnahmen für Testzwecke mithilfe von ATPG Nebenbedingungen (vgl. Kapitel 3.2.3) und sind nicht Teil der untersuchten Implementierungen. Diese Signale werde im folgenden als ATPG-Steuersignale

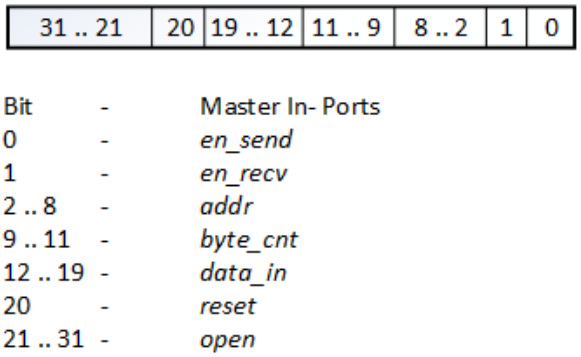
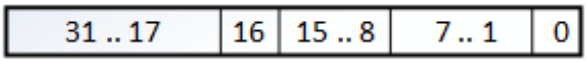
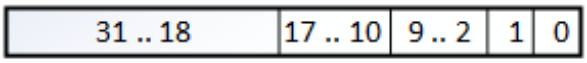


Abbildung 3.4: Bit-Mapping der MMI/O Adresse auf die Eingänge der I^2CMasterkomponente.



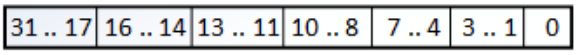
Bit	-	Slave In- Ports
0	-	*reserved (lb_en)*
1 .. 7	-	*addr*
8 .. 15	-	*data_in*
16	-	*reset*
17 .. 31	-	*open*

Abbildung 3.5: Bit-Mapping der MMI/O Adresse auf die Eingänge der I^2CSlavekomponente.



Bit	-	System Out- Ports
0	-	*m_idle*
1	-	*s_idle*
2 .. 9	-	*m_data_out*
10 .. 17	-	*s_data_out*
18 .. 31	-	*open*

Abbildung 3.6: Bit-Mapping der MMI/O Adresse auf die Ausgänge der I^2CKomponenten.



Bit	-	Modification Out- Ports
0	-	*err*
1 .. 3	-	*m_cnt*
4 .. 7	-	*m_state*
8 .. 10	-	*i2c_bus*
11 .. 13	-	*s_cnt*
14 .. 16	-	*s_state*
17 .. 31	-	*open*

Abbildung 3.7: Bit-Mapping der MMI/O Adresse auf die Maßnahmen zur verbesserten Beobachtbarkeit.

bezeichnet. Dazu wurde jedes Signal der Verbesserungen mit einem zusätzlichen Steuersignal an ein UND-Gatter angeschlossen, dessen Ausgabewert als finales Steuersignal genutzt wird. Wird folglich ein ATPG-Steuersignal als Nebenbedingung in der Form $a = 0$ auf den logischen Wert 0 gezwungen, so steht die entsprechende Maßnahme für den folgenden Test nicht zur Verfügung. So kann z. B. durch das Einfügen einer Nebenbedingung für die Steuerleitung lb_{enable} auf den Wert 0 die Benutzung der Loopback-Komponente deaktiviert werden.

3.2.1 Loopback

Eine häufig genutzte Methode um die Beobacht- und Kontrollierbarkeit von Kommunikationsperipheriekomponenten zu erhöhen ist das Einfügen eines Loopbacks, eine Möglichkeit übertragene Daten vor und nach der Übertragung einsehen zu können. Durch das Einfügen einer sehr einfachen Loopback-Komponente in das in Kapitel 3.1 vorgestellte Testsystem werden die beiden - bislang nicht verbundenen - I^2C-Module an einen gemeinsamen I^2C-Bus angeschlossen. Über das lb_{enable}-Signal kann zwischen diesem Testmodus (die Komponenten sind über einen internen I^2C-Bus verbunden) und dem Normalbetrieb (die Komponenten sind an ihre jeweiligen externen Busse angeschlossen) gewechselt werden.

Die Beobachtbarkeit wird erhöht, da durch diese Erweiterung alle Übertragungen zwischen Master- und Slavekomponente beobachtet werden können. So wird es möglich die ursprünglichen mit den empfangenen Daten abzugleichen und dadurch Fehler aufzudecken. Stimmen die so erzeugten Daten überein, ist zudem eine protokollgerechte Übertragung abgelaufen. Folglich können bestimmte Fehler in der Protokoll-Logik mittels der Loopback-Komponenten entdeckt werden, wenn sich diese Fehler auf die empfangenen Daten der Gegenseite auswirken.
Die Kontrollierbarkeit interner Signale wird durch ein Loopback erhöht, da dies den vollständigen Ablauf einer Kommunikation ermöglicht. Durch die Möglichkeit den Loopback während einer Übertragung abzubrechen, wird die Fehlerbehandlungslogik testbar. So lässt sich beispielsweise der Verbindungsabbruch während der Kommunikation, resultierend in einem empfangenen NACK bei der Datenbestätigung, oder ein nicht antwortender Slave bei der Adressierung, simulieren und so die zuständige Logik aktivieren und testen. Ob diese Möglichkeit beim ATPG-Prozess zum Einsatz kommt, hängt jedoch von dem verwendeten Template, und den daraus resultierenden Nebenbedingungen ab.

Diese Maßnahme führt zu einem nur sehr geringen Hardware Overhead. Für jede der insgesamt acht I^2C-Busleitungen wird ein 2×1 Multiplexer benötigt, jeweils mit lb_{enable} als Steuersignal, um die Anbindung der beiden I^2C-Komponenten zwischen Testmodus und Normalbetrieb zu wechseln.

3.2.2 Sichtbarmachung interner Signale

Um die Beobachtbarkeit des internen Zustands der Komponente zu erhöhen, können interne Signale dem System sichtbar gemacht werden. Diese Signale werden oft auch zu Debugzwecke in die MMI/O Schnittstelle integriert. Im Folgenden werden diese Signale einzeln erklärt und kurz die Motivation hinter der Wahl der gewählten Signale vorgestellt.

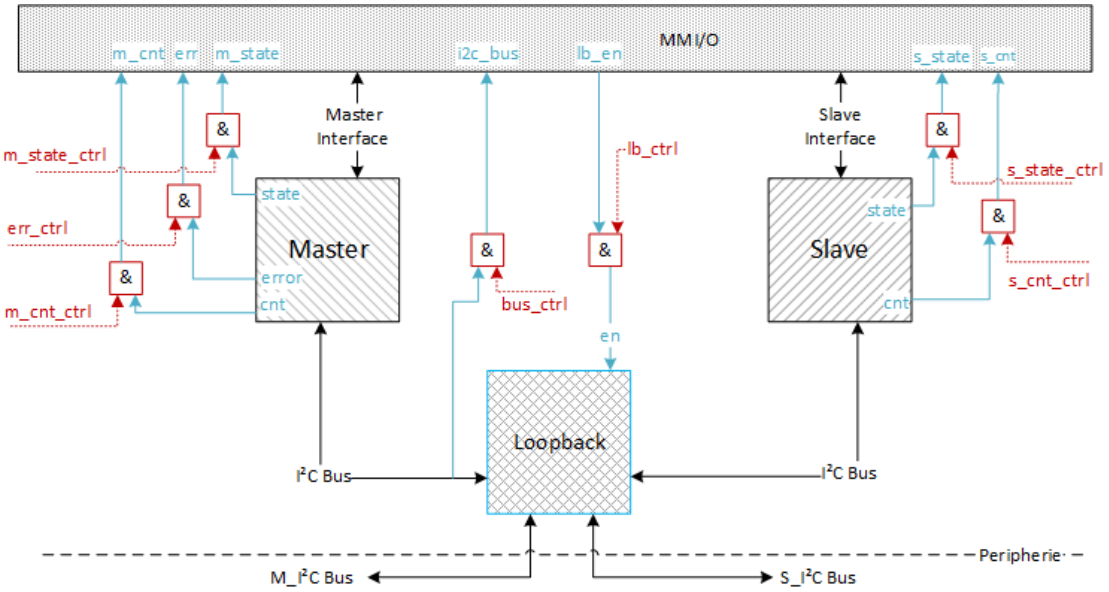


Abbildung 3.8: Modifiziertes Testsystem.

State Diese Erweiterung wird sowohl auf die Master- als auch auf die Slavekomponente angewandt und gibt den - binär codierten - Zustand aus. Die vorgestellten Zustandsautomaten der Master- bzw. Slavekomponente zeigen die Codierung der Zustände. Durch diese Maßnahme lassen sich z. B. Fehler in der internen Umsetzung der Zustandsautomaten erkennen bzw. von anderen Haftfehlern des Systems unterscheiden. Für die Ausgabe des aktuellen Zustands der Masterkomponente werden 4-Bit benötigt, beim Zustand des Slaves lediglich 3-Bit.

I^2C-Bus Um die protokollgerechte Kommunikation der Komponenten nachvollziehen zu können, werden die Bussignale des I^2C-Busses an der MMI/O Schnittstelle sichtbar gemacht. Dazu werden die I^2C-Signale (SDA und SCL) der Masterkomponente abgegriffen und dem System sichtbar gemacht.

Bit-Zähler Beide I^2C-Komponenten enthalten jeweils eine Sende- und Empfangsschleife mit einem 8-Bit Zähler um die Zahl der empfangenen bzw. gesendeten Bits zu zählen. Durch Ausgabe dieses Zählersignals, über ein 3-Bit Signal, können Haftfehler in der Zählerlogik entdeckt werden, welche u.a. zu Endlos-Übertragungen führen können.

Error Als Alternative zur codierten Ausgabe aller Zustände, können einzelne Zustände ausgegeben werden. Als Beispiel wird ein Signal gewählt, dass das Erreichen des Fehlerzustands der Masterkomponente anzeigt.

3.2.3 Untersuchte Konfigurationen

Um die Wirksamkeit der einzelnen Maßnahmen zu untersuchen, werden verschiedene Kombinationen der vorgestellten Eingriffe unter bestimmten Bedingungen untersucht. Tabelle 3.1 zeigt alle

untersuchten Varianten und die ATPG-Steuersignale, die für diese jeweils gesetzt werden müssen. Grundsätzlich werden zwei verschiedene Konfigurationen betrachtet:

- N Bei Versuchsbezeichnungen die mit diesem Kürzel beginnen wird davon ausgegangen, dass kein Kommunikationspartner im I^2C-Netzwerk vorhanden ist. Folglich kann keine Datenübertragung erfolgen. Im Testsystem werden diese Versuche umgesetzt indem die Loopback-Komponente deaktiviert wird.
- O Bei dieser Konfiguration kann der Kommunikationspartner (der I^2C-Slave) beobachtet und kontrolliert werden. Es entsteht folglich ein Loopback, da gesendete bzw. empfangene Daten des DUT mit den korrespondierenden Daten des Slave abgeglichen werden können. Dazu wird das lb_{enable}-Signal auf eine logische 1 gesetzt.

3.3 Testprogrammzeugung

Entsprechend der in Abschnitt 2.3.2 vorgestellten, allgemeinen Methode zur Erzeugung struktureller SBST-Programme gliedert sich die im Folgenden beschriebene Methode in zwei Abschnitte: Zuerst wird in Unterkapitel 3.3.1 die Erzeugung von Testmustern besprochen und anschließend im Abschnitt 3.3.2 die Umwandlung der Testmuster in ein Testprogramm diskutiert.

3.3.1 Testmuster

Das SAT-basierte ATPG wandelt die ursprünglich sequentielle Zielschaltung durch Anwendung von TFE in eine rein kombinatorischen Darstellung um, welche die Originalschaltung für eine bestimmte

Versuchskürzel	lb_en	m_cnt_en	err_en	m_state_en	i2c_Bus_en	s_state_en	s_cnt_en
$NONE$	0	0	0	0	0	0	0
N_cnt_m	0	1	0	0	0	0	0
N_err_m	0	0	1	0	0	0	0
N_state_m	0	0	0	1	0	0	0
N_bus	0	0	0	0	1	0	0
O_lb	1	0	0	0	0	0	0
$O\_lb\|cnt_m$	1	1	0	0	0	0	0
$O\_lb\|err$	1	0	1	0	0	0	0
$O\_lb\|state_m$	1	0	0	1	0	0	0
$O\_lb\|bus$	1	0	0	0	1	0	0
$O\_lb\|state_s$	1	0	0	0	0	1	0
$O\_lb\|cnt_s$	1	0	0	0	0	0	1
$FULL$	1	1	1	1	1	1	1

Tabelle 3.1: Untersuchte Varianten und korrespondierende ATPG-Steuersignale.

Anzahl von Taktschlägen repräsentiert. Die funktionalen Nebenbedingungen für die Testmustererzeugung setzten sich aus einer Vielzahl von Faktoren zusammen und müssen als Nebenbedingung an das ATPG-Programm übergeben werden.

Funktionale Nebenbedingungen

Grundsätzlich muss die zentrale funktionale Nebenbedingung von SBST beachtet werden, d.h. Testmuster müssen sich mithilfe von ISA-Instruktionen an das DUT anlegen lassen.

Ein wichtiger Aspekt spielt das Timing-Verhalten der betrachteten Komponenten. Ohne zusätzliche Einschränkungen erzeugt ein ATPG-Programm Testmuster unter der Annahme, dass alle gegebenen PIs in jedem Takt geschrieben und alle POs in jedem Takt beobachtet werden können. Daraus folgt, dass ein Testprogramm innerhalb eines Taktschlags des DUT an alle PIs Testvektoren anlegen muss und zusätzlich aktuelle Ausgabewerte der POs, für den späteren Abgleich, abspeichern oder mit den Sollwerten der Testmuster abgleichen muss. Diese Aufgabe kann nur durchgeführt werden, wenn die untersuchte Komponente eine erheblich geringere Taktfrequenz als der Mikrocontroller besitzt. Diese Bedingung lässt sich allgemein durch folgende Formel darstellen:

$$f_{DUT} < \frac{f_{sys}}{(t_{PI} + t_{PO} + t_I)} \tag{3.1}$$

f_{DUT} Taktfrequenz der zu testenden Komponente

f_{sys} Taktfrequenz des Prozessors

t_{PI} Anzahl der Takte die benötigt werden um alle PIs der Komponente mit gegebenen Testvektoren zu belegen.

t_{PO} Anzahl der Takte die benötigt werden um alle POs der Komponente auszulesen und abzuspeichern bzw. mit den zugehörigen Sollwerten des Testmusters abzugleichen.

t_I Anzahl der Takte die für zusätzliche Instruktionen benötigt werden (z. B. das Laden von Konstanten, Sprungbefehle oder Vergleichsoperationen).

Bei dieser Abschätzung muss je nach Template auch die Ausführungszeit von Speicherbefehlen abgeschätzt werden. Dazu wird Formel 3.1 um die maximale Dauer für die Ausführung eines Speicherbefehls (t_{sp}) erweitert. Der Wert m beschreibt dabei die Anzahl der Speicherbefehle im Template. Werden in einem Test-Template Speicherbefehle genutzt, kann folglich dessen Laufzeit nicht deterministische angegeben werden.

$$f_{DUT} < \frac{f_{sys}}{(t_I + (x * m))} \tag{3.2}$$

Formel 3.2 stellt somit die obere Schranke für die Verwendung von Testmustern dar, welche von der Voraussetzung ausgehen, dass in jedem Peripherietakt Testvektoren an die PIs des DUT angelegt und Ergebniswerte der POs ausgelesen werden können. Erfüllt das gegebene System diese Formel nicht, so muss das ATPG eingeschränkt werden, oder m verkleinert werden. Diese Einschränkung

kann offensichtlich die Kontrollier- und Beobachtbarkeit der Komponente verringern und hat somit direkten Einfluss auf die Fehlerabdeckung der so erzeugten Testmuster.

Ein weiterer kritischer Aspekt ist die Synchronisierung zwischen Testprogramm und dem DUT. Da Kommunikationskomponenten im Allgemeinen eine langsamere Taktfrequenz als der Mikroprozessor des Systems besitzen, muss der Ablauf eines SBST-Programms an die Geschwindigkeit der Kommunikationskomponente angepasst sein. Andernfalls könnten Testvektoren zu schnell oder zu langsam angelegt werden. Die Taktdifferenz zwischen System- und Peripherietakt kann als Anhaltspunkt dienen um synchrone SBST-Programme zu entwickeln. Eine besondere Herausforderung stellen in diesem Zusammenhang die Verwendung von Speicherzugriffen dar, da deren Dauer nicht konstant ist. Das Testsystem enthält für Synchronisierungszwecke ein *sync*-Signal, welches durch den Clock Divider erzeugt wird (vgl. Abschnitt 3.1.3). Mithilfe dieses Signals können Templates entworfen werden, die eine bestimmte sequentielle Tiefe abwarten, bevor sie die Ausgabewerte der Peripheriekomponente auslesen, oder die auf Taktflanken des Peripherietakts reagieren.

Soll ein Testvektor je Peripherietakt angelegt und die aktuellen Ausgabewerte überprüft werden, so muss die Kommunikationskomponente nicht nur Formel 3.2 genügen, sondern zusätzlich eine Synchronisierungsmöglichkeit besitzen. Ist dies nicht der Fall, kann nur ein Testmuster verwendet werden, das zu Beginn des Programms Werte anlegt und nach Ablauf einer vorgegeben Taktzahl die Ausgabewerte der Schaltung ausliest. Eine Beobachtung bzw. Kontrolle der Schaltung zwischen Beginn und Ende des Testmusters ist in einem solchen Fall nicht möglich. Diese beiden Fälle führen zu den zwei unterschiedlichen, im folgenden Abschnitt vorgestellten, Templates *simple* und *extended*.

3.3.2 Testprogramm

Im Folgenden werden zwei parametrisierte Templates vorgestellt mit denen sich die, im vorherigen Schritt entwickelten, Testmuster an die I^2C-Komponente anlegen lassen. Die Verwendung von parametrisierten Templates reduziert den manuellen Aufwand für den Entwurf von finalen Testprogrammen. Mithilfe eines Pre-Compilers lassen sich die parametrisierten Werte der Templates durch die entsprechenden Werte der Testmuster ersetzen. Die vorgestellten Templates nutzen die Syntax des x86 NASM Pre-Prozessors [NAS15] um die Verwendung von Parametern und lokale bzw. globale Sprungmarken zu definieren.

Das *simple*-Template (siehe Abschnitt 3.3.2) kann angewendet werden, sollte das untersuchte System Formel 3.2 nicht erfüllen, oder keine Synchronisierungsmöglichkeit bestehen. Das *extended*-Template (siehe Abschnitt 3.3.2) geht davon aus, dass beide Bedingung erfüllt sind und ist in der Lage ein Testvektor pro Peripherietakt anzulegen und die Ausgabewerte dieses Taktes zu prüfen.

Globale Testprogrammbedingungen

Sowohl das *simple* als auch *extended* Template nutzen einige globale Bedingungen, die das Testprogramm sicherstellen muss. Im Wesentlichen sind dies Anforderungen von bestimmten Registerinhalten oder globale Sprungadressen, die von allen Templates genutzt und nicht verändert werden. Die Register werden dabei wie folgt verwendet:

$0: Hält den Wert 0.

$2: Hält die *master_in* MMI/O Adresse.

$3: Hält die *slave_in* MMI/O Adresse.

$4: Hält die *system_out* MMI/O Adresse.

$5: Hält die *modification_out* MMI/O Adresse.

fail: Bezeichnet eine - über alle Templateausführungen hinweg - globale Sprungmarke die einen Fehler-Handler enthält. Wird während der Ausführung des Templates ein Fehler i, DUT entdeckt, so wird zu dieser Sprungmarke gesprungen. Ein möglicher Fehler-Handler wird in Listing 3.1 gezeigt. Im Fehlerfall wird eine -1 in das Rückgaberegister $30 gelegt und zu einer in $31 gespeicherten Rücksprungadresse gesprungen.

ok: Bezeichnet eine - über alle Templateausführungen hinweg - globale Sprungmarke die den Handler für einen fehlerfreien Fall enthält. Wird während der Ausführung des Templates kein Fehler entdeckt, so wird zu dieser Sprungmarke gesprungen.

```
#Failure Handler, write -1 into $30 and exit
fail: xori $30, $0, -1
jr $31
```

Listing 3.1: Beispiel eines Fehler-Handlers

Einfaches Template

Das *simple* Testmakro ist in der Lage ein Testmuster an die I^2C-Komponente anzulegen, eine als Übergabeparameter gegebene Zahl an I^2C-Takten zu warten und die Werte der POs auszulesen und mit den durch das Testmuster gegebenen Soll-Werten abzugleichen. Abbildung 3.9 zeigt die Anwendung des *simple*-Templates im Bezug zum Peripherietakt I^2C_{clk} und dem Taktzähler der sequentiellen Tiefe. Sollte eine Abweichung auftreten wird in den globalen Fehler-Handler gesprungen. Die Einschränkungen (Inputs und Outputs können nur zu Beginn gesetzt bzw. im letzten Takt gelesen werden) des Templates müssen bei der Testmustererzeugung berücksichtigt werden. Dazu werden beim TFE-Verfahren statische PIs verwendet und die Beobachtbarkeit der POs zeitlich auf den letzten Takt der betrachteten Schaltung beschränkt. (vgl. Unterabschnitt: Sequenzielles ATPG).

Als Übergabeparameter erhält das Template ein Testmuster, wobei die vier Testvektoren als Halbworte übergeben werden. Dies ist notwendig um die Testvektoren als Immediate-Werte laden zu können. Zusätzlich erhält das Template die sequentielle Tiefe, ebenfalls in Form von zwei Halbworten, als Übergabeparameter.

Das *simple* Template besteht aus vier Schritten: *Preparation, Pattern, Wait und Compare.*
Im *Preparation*-Schritt werden die übergebenen Eingabevektoren und die sequentielle Tiefe in die Register $10 bis $12 geladen.
Im *Pattern*-Schritt werden Testvektoren an die PIs der Schaltung angelegt. Zunächst wird der Testvektor für die *slave_in* MMI/O Adresse angelegt, anschließend der Testvektor der *master_in* Schnittstelle.

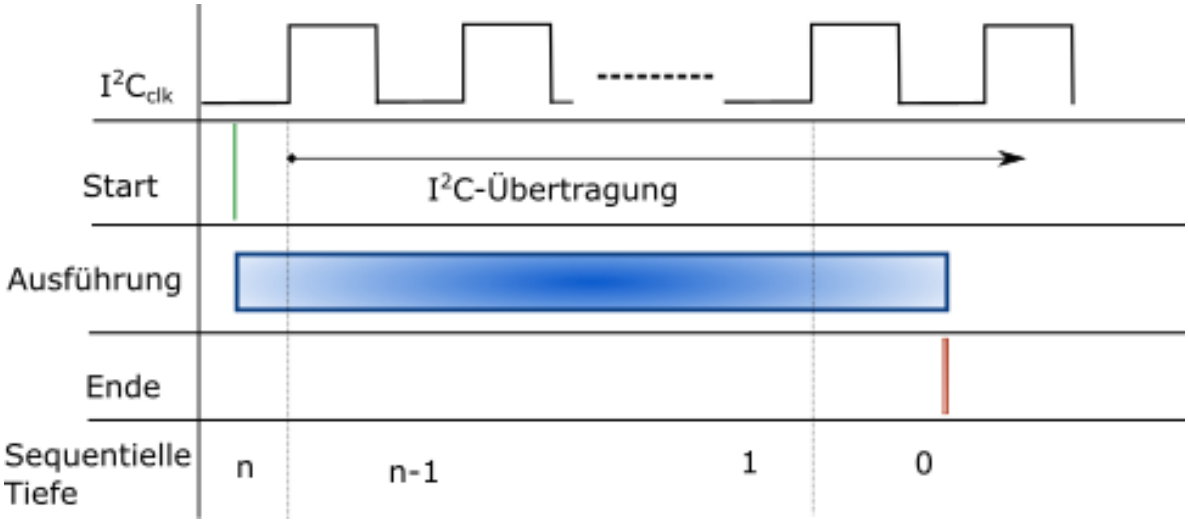


Abbildung 3.9: Ausführungen des *simple* Templates in Abhängigkeit zum Peripherietakt.

Diese Reihenfolge ist entscheidend, da *master_in* die Signale *en_recv* und *en_send* belegt, die für den Start einer I^2C-Übertragung verantwortlich sind. Die Schritte *Preparation* und *Pattern* entsprechen *Start* in Abb. 3.9.
Im anschließenden *Wait*-Schritt wird die sequentielle Tiefe des Testmusters abgezählt. Dies kann mit verschiedenen Maßnahmen erreicht werden. Ist die genaue Zahl der Takte bekannt, die der Peripherietakt gegenüber dem Systemtakt langsamer ist, so kann dies durch einfaches Einfügen einer *nop*-Schleife erreicht werden. Ist der Peripherietakt zumindest um den Faktor zehn langsamer (Anzahl der Instruktionen des *Wait*-Schritts) und ist das *sync*-Signal vorhanden, so kann die implementierte *Wait*-Schleife genutzt werden. Diese dekrementiert den übergebenen Zähler der sequentiellen Tiefe stets auf die fallende Taktflanke des Peripherietakts, wie in Abb. 3.9 zu erkennen ist. Zu Beginn der Schleife werden die POs der Schaltung in die temporären Register $11 und $12 gelesen, es sind als immer die Ausgabewerte der aktuell betrachteten Periode gespeichert.
Im abschließenden *Compare*-Schritt werden die ausgelesenen Werte der POs mit den Soll-Werten des Testmusters verglichen. Wird eine Abweichung festgestellt, wird in den globalen Fehler-Handler gesprungen. Dieser Schritt entspricht *Ende* in Abb. 3.9.

```
#Paramters:  (M_high,M_low,S_high,S_low,sys_high,sys_low,mod_high,
#                       mod_low,seq_high,seq_low)
%macro simple 10

##### Preparation - Step:
lui $10, %9
xori $10, $10, %10 #Load Sequential Depth Value in $10
lui $11, %3
xori $11, $11, %4 #Load Slave Input Values in $11
lui $12, %1
xori $12, $12, %2 #Load Master Input Values in $12
```

```
##### Pattern - Step: #####
sw $11, 0($3) #Write Slave Pattern to slave_input MMI/O Address
sw $12, 0($2) #Write Master Pattern to master_input MMI/O Address

##### Wait - Step: #####
%%wait2:                #Wait for sync='1' then cnt-- until cnt = 0
lw $11, 0($4)           #load System_out Values to $11
lw $12, 0($4)           #load modification_out Values to $12
andi $13, $11, 1
bgtz $13, sync_high     #if sync=1 goto sync_high
bgtz $15, wait2         #if sync=0 and stat=1 gogo wait2
addi $10, $10, -1       #decrement seq_depth counter
addi $15, $15, 1        #stat = 1
bgez $10, %%compare     #if seq_depth >= 0 goto compare
%%sync_high
and $15, $15, $0        #stat = 0
bgez $15, wait2         #Goto wait2

##### Compare - Step: #####
%%compare:
lui $13, %5
xori $13, $13, %6 #Compare system_output to set values
bne $13, $11, fail #if not equal, jump to the failure handler
lui $13, %7
xori $13, $13, %8 #Compare modification_output to set values
bne $11, $12, fail #if not equal, jump to the failure handler
#####
blez $0, ok #Fail Free: jump to ok-Handler
%endmacro
```

Listing 3.2: Testmakro: *simple*

Erweitertes Template

Das *extended* Testmakro ist in der Lage ein Testmuster für einen Takt an die I^2C-Komponente anzulegen, die Werte der POs auszulesen und mit den durch das Testmuster gegebenen Soll-Werten abzugleichen. Eine Abfolge dieser Templates ermöglicht es, in jedem betrachteten Takt Werte anzulegen, auszulesen und zu vergleichen. Folglich kann für TFE eine Variante mit nicht-statischen PIs und in jedem Takt beobachtbaren POs gewählt werden (vgl. Abschnitt 2.2.3). Abbildung 3.10 zeigt eine aufeinanderfolgende Ausführungen von *extended*-Templates im Bezug zum I^2C-Peripherietakt. Wie die Abbildung zeigt, beginnt die Templateausführung sobald eine vorherige Templateausführung abgeschlossen ist und das *sync*-Signal den Wert 0 annimmt. So kann sichergestellt werden, dass alle POs der Komponenten konstant anliegen und das Änderungen der PIs zu keiner Störung der Komponente führen.

Folglich muss der, mithilfe von Formel 3.2, berechnete Wert für die Taktdifferenz verdoppelt werden (WKS-Abtasttheorem), da die Ausführung des Templates innerhalb eines halben Peripherietaktes durchgeführt wird. Somit ergibt sich durch Einsetzen der Anzahl von Instruktionen des *extended*-Templates eine Taktdifferenz von:

$$f_{DUT} < \frac{f_{sys}}{(17 * 2)} \tag{3.3}$$

Folglich kann das *extended*-Template angewendet werden, wenn der Clock Divider mindestens eine Taktdifferenz von 34 vorsieht. Für die Übertragung eines Bytes benötigt die vorgestellte I^2C-Komponente 20 Takte. Somit ergibt sich für das gegebene Testsystem und einer I^2C-Übertragungsrate von $100kbit/s \approx 12$kbyte/s (NM) eine Mindestaktrate von:

$$f_{DUT} \geq 12\text{kbyte/s} * 20\text{byte} = 240kHz \tag{3.4}$$

Durch Einsetzten der Taktfrequenz des miniMIPS-Prozessors ergibt sich die - erfüllte - Ungleichung 3.5:

$$240kHz < \frac{50000\text{kHz}}{34} \quad \square \tag{3.5}$$

Folglich lässt sich das *extended*-Template für die vorgestellte Testumgebung nutzen. Auch für den FM Übertragungsmodus (400 kbit/s) ist die Ungleichung erfüllt ($1000kHz < 1470,59kHz$). Soll eine Komponente mit einem schnelleren Übertragungsmodus (FM+ oder HS-mode) getestet werden, muss auf das *simple*-Template zurückgegriffen werden.

Als Übergabeparameter erhält das Template ein Testvektor je MMI/O Adresse, diese werden als Halbworte übergeben, um als Immediate-Werte geladen werden zu können. Das *extended* Template gliedert sich in drei Schritte: (*Wait, Pattern und Compare*).
Im *Wait*-Schritt wird die Synchronisierung mit der I^2C-Komponente abgewartet. Dazu wird das *sync*-Signal via Polling ausgelesen. Hat dieses Signal den logischen Wert '0' - befindet sich der Peripherietakt folglich in der *LOW*-Phase - wird die Templateausführung begonnen. Hat das *sync* den Wert '1' so wird in eine Polling-Schleife gesprungen, bis *sync* den Wert '0' annimmt und erst anschließend gestartet.
Im *Pattern*-Schritt werden die übergebenen Testvektoren an die PIs der Schaltung angelegt. Zunächst wird der Testvektor für die *slave_in* MMI/O Adresse angelegt, anschließend der Testvektor der *master_in* Schnittstelle. Diese Reihenfolge ist lediglich für die Ausführung des ersten Templates von Bedeutung, da *master_in* die Signale *en_recv* und *en_send* belegt, die eine I^2C-Übertragung starten.
Im anschließenden *Compare*-Schritt werden die aktuell anliegenden Werte der POs ausgelesen und mit den Soll-Werten des Testmusters verglichen. Sollte ein ausgelesener Wert von den Sollwerten abweichen, wird in den globalen Fehler-Handler *fail:* gesprungen, andernfalls nach *ok:*.

```
%macro advanced 8
#Paramters: (M_high,M_low,S_high,S_low,sys_high,sys_low,mod_high,mod_low)

#####Wait - Step: (I=3)
%%wait:          #wait for sync=0
lw $10, 0($4) #load system_out values to $10
andi $11, $10, 1
```

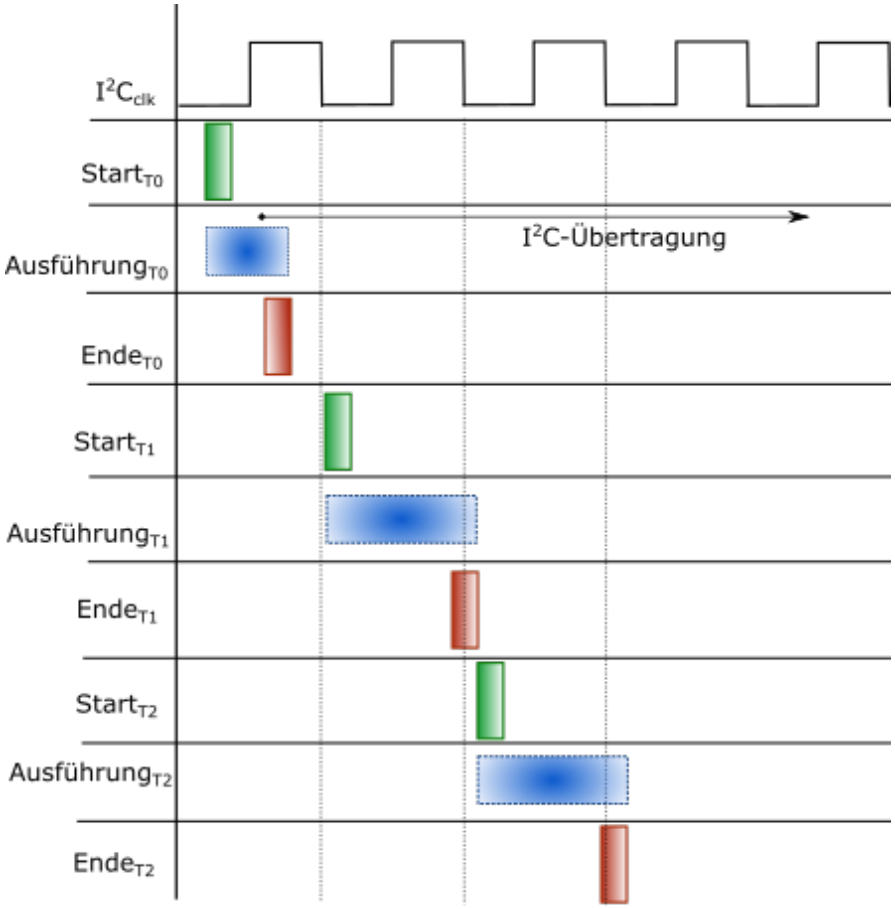


Abbildung 3.10: Abfolge von Ausführungen des *extended* Templates im Bezug zum Peripherietakt I^2C_{clk}.

```
bgtz $11, %%wait  #syn>0 wait

#####Pattern - Step: (I=6)
lui $11, %3
xori $11, $11, %4 #Load Slave Input Values to $11
sw $11, 0($3) #Write Slave Input Patterns to slave_input MMI/O Address
lui $11, %1
xori $11, $11, %2 #Load Master Input Values to $11
sw $11, 0($2) #Write Slave Input Patterns to slave_input MMI/O Address

#####Compare - Step: (I max = 8)
lui $11, %5
xori $11, $11, %6 #load set values of system_out to $11
bne $10, $11, fail #if not equal jump to the failure handler
```

```
lw $11, 0($5) #Load current modification_out values
lui $12, %7
xori $12, $12, %8 #load set values of mod_out to $12
bne $11, $12, fail #if not equal jump to the failure handler

#Fail Free
blez $0, ok
%endmacro
```

Listing 3.3: Testmakro: *extended*

Finales Testprogramm

Im Folgenden wird besprochen wie aus den beiden vorgestellten, parametrisierten Templates und den erzeugten Testmustern ein finales Testprogramm generiert werden kann.
Zunächst müssen die globalen Testprogrammbedingungen in ein Programmskelett übertragen werden. Anschließend werden die Parameter für die einzelnen Templateinstanzen aus den erzeugten Testmustern gewonnen. Die Instanzen der Templates werden im Testprogramm nacheinander mit den verschiedenen Parametern aufgerufen.
Die beiden vorgestellten Tempaltes unterscheiden sich, aus Sicht der finalen Testprogrammerzeugung, lediglich in der Zahl der Template-Instanzen, die pro Testmuster benötigt werden. Während bei der Nutzung des *simple*-Templates nur eine Instanz pro Testmuster erzeugt werden muss, werden bei der Nutzung des *extended*-Templates eine, von der sequentiellen Tiefe linear abhängige, Anzahl an Instanzen erzeugt. Folglich unterscheidet sich die Testprogrammgröße bei Verwendung der beiden Templates erheblich voneinander. Formel 3.6 zeigt die Abschätzung der Testprogrammgröße ($n_{simple|Speicherbedarf}$) bei Verwendung des *simple*-Template. Die Programmgröße hängt dabei im Wesentlichen von der Anzahl der erzeugten Testmuster (n_{Muster}), der Anzahl der Instruktionen des *simple*-Templates ($n_{Template}$) und der Anzahl der Instruktionen die im Programmskelett zur Initialisierung ($n_{Initialisierung}$) und zum Beenden des Programms (n_{Ende}) benötigt werden, ab.

(3.6) $$n_{simple|Speicherbedarf} = n_{Initialisierung} + n_{Muster} * n_{Template} + n_{Ende}$$

Durch Einsetzen der Anzahl der *simple*-Instruktionen (25), der benötigten Zahl an Instruktionen für die Initialisierung (9) und das Beenden des Testprogramms (1), in die Formel 3.6 ergibt sich die Abschätzung 3.7, in Abhängigkeit der Testmusteranzahl. Durch Multiplikation mit 4 ergibt sich aus den Instruktionen der benötigte Speicherbedarf des Testprogramms in Bytes.

(3.7) $$n_{simple|Speicherbedarf} \geq (25 * n_{Muster} + 10) * 4$$

Die Programmgröße bei Verwendung des *extended*-Tempalte lässt sich ähnlich berechnen, wobei als zusätzlicher Faktor die sequenzielle Tiefe (n_{seq}) hinzukommt, da für jeden betrachteten Takt eines Testmusters ein Template benötigt wird. Damit ergibt sich Formel 3.8

(3.8) $$n_{extended|Speicherbedarf} = n_{Initialisierung} + n_{Muster} * n_{Template} * n_{seq} + n_{Ende}$$

Durch Einsetzen (Instruktionszahl des *extended*-Template ist 17, alle anderen Werte identisch zu 3.7) ergibt sich die Abschätzung 3.9 der Programmgröße in Abhängigkeit von der Anzahl der Testmuster und der betrachteten sequentiellen Tiefe.

(3.9) $n_{extended|Speicherbedarf} \geq (17 * n_{Muster} * n_{seq} + 10) * 4$

Die Laufzeit des Testprogramms ($t_{Laufzeit}$) je nach Verwendung des Template-Typs variiert weniger stark als die Programmgröße, da die bestimmenden Faktoren der Gleichung für beide Template-Varianten identisch sind. Diese Faktoren sind die betrachtete sequentielle Tiefe (n_{seq}) und die Anzahl der erzeugten Testmuster (n_{Muster}). Da die Testlaufzeit in Systemtakten gemessen wird, ist zudem der Teilungsfaktor zwischen Systemtakt und Peripherietakt ($k_{Taktteiler}$) von Bedeutung. Dieser ist definiert als $k_{Taktteiler} = \frac{f_{System}}{f_{Peripherie}}$. Zusätzlich muss die Laufzeit des Programmskeletts berücksichtigt werden ($t_{Initialisierung}$ und t_{Ende}).

Im Fall des *simple*-Template werden alle bis auf zwei Instruktionen (vgl. *syn_high*-Sprungmarke des Listings 3.2) insgesamt einmal vor bzw. nach dem Ablauf der betrachteten sequentiellen Tiefe ausgeführt. Dieser konstante Wert muss bei der Abschätzung beachtet werden ($t_I = 23$). Damit ergibt sich Formel 3.10:

(3.10) $t_{simple|Laufzeit} = t_{Initialisierung} + n_{Muster} * (n_{seq} * k_{Taktteiler} + t_I) + t_{Ende}$

Unter der Annahme, dass die Instruktionen des Programmskeletts in einem Takt ablaufen, ergibt sich durch Einsetzen als Abschätzung der Laufzeit des *simple*-Template die Formel 3.11:

(3.11) $t_{simple|Laufzeit} = n_{Muster} * (n_{seq} * k_{Taktteiler} + 23) + 10$

Da alle Instruktionen des *extended*-Template parallel zum Peripherietakt ablaufen, muss lediglich die erste Templateausführung ($t_{ex} = 17$) je Testmuster eingerechnet werden (da diese vor Beginn der Peripherieoperation ausgeführt wird). Alle weiteren Ausführungen sind bereits durch das Einbeziehen der sequentiellen Tiefe eingerechnet. Somit ergibt sich Formel 3.12:

(3.12) $t_{extended|Laufzeit} = t_{Initialisierung} + n_{Muster} * (n_{seq} * k_{Taktteiler} + t_{ex}) + t_{Ende}$

Durch Einsetzen, analog zu 3.11, ergibt sich die Abschätzung für die Laufzeit des *extended*-Template 3.13:

(3.13) $t_{extended|Laufzeit} = n_{Muster} * (n_{seq} * t_{Taktteiler} + 17) + 10$

Aus dem Vergleich der Abschätzungen 3.7, 3.9, 3.11 und 3.13 geht hervor, dass das *extended*-Template bei gleicher Anzahl an erzeugter Testmuster und derselben betrachteten sequentiellen Tiefe, eine etwas kürze Laufzeit, bei deutlich vergrößertem Speicherbedarf aufweist, verglichen mit dem *simple*-Template. In späteren Experimenten (vgl. Kapitel 4) zeigt sich jedoch, dass mit dem *extended*-Template eine höhere Fehlerabdeckung als mit dem *simple*-Template erreicht werden kann.

4 Experimente

Im Folgenden wird experimentell die Wirksamkeit der in Kapitel 3.2 vorgestellten Maßnahmen zur Verbesserung der Beobacht- und Kontrollierbarkeit anhand des in Kapitel 3.1 beschriebenen Testsystems untersucht. Zunächst werden dazu in Abschnitt 4.1 die Ergebnisse der Synthese des Testsystems vorgestellt. Anschließend wird in Unterkapitel 4.2 experimentell die sequentielle Tiefe untersucht, welche zum erreichen einer bestimmten Fehlerabdeckung notwendig ist. Im Unterkapitel 4.3 werden die experimentellen Ergebnisse der in Kapitel 3.2.3 beschriebenen Versuchsreihe vorgestellt. Als Kriterien dienen u.a. die in Kapitel 2.3 vorgestellten Qualitätsmerkmale: prozentuale Fehlerabdeckung, Anzahl der erzeugten Testmuster, Testprogrammgröße und Testdauer. Im Anschluss werden in Unterkapitel 4.4 die vorgeschlagenen Maßnahmen anhand der experimentellen Ergebnisse analysiert und deren Einfluss auf die erreichbare Haftfehlerabdeckung diskutiert. Zuletzt werden in Kapitel 4.5 die beiden Template-Varianten bewertet.

4.1 Synthese

Die in den Unterkapiteln 3.1.1 und 3.1.2 vorgestellte I^2C-Komponente wurde, zusammen mit der in Abschnitt 3.2.1 besprochenen Loopback-Einheit, mithilfe des Synthesetools Design Compiler von Synopsys hierarchisch synthetisiert. Das synthetisierte Gatternetz (vor Anwendung von TFE) besitzt 634 Gatter und 51 Flip-Flops. Die Haftfehlerliste der Schaltung hat, vor fault collapsing, 4324 Einträge. In der kollabierten Fehlerliste sind noch 1836 Haftfehler enthalten. Diese Fehler verteilen sich auf die I^2C-Masterkomponente (904), den I^2C-Slave (713) und die Loopback-Komponente (11). Die verbleibenden Haftfehler (208) sind Teil des Testsystems (vorwiegend aufgrund der ATPG-Steuersignale und der zusätzlichen UND-Gatter, vgl. Kapitel 3.2) und werden im Folgenden nicht weiter untersucht.

Damit liegt die Komplexität der betrachteten Schaltung unter der UART-Komponente die in [APGP07b] und [BSS+07] genutzt wird.

4.2 Experimentelle Untersuchung der sequentiellen Tiefe

Die Gesamthaftfehlerabdeckung des *FULL*-Versuchs wird, für beide Template-Varianten, jeweils für die sequentiellen Tiefen 0 bis 120 untersucht. Abbildung 4.1 stellt die prozentuale Abdeckung aller Haftfehler (y-Achse) der getesteten Schaltung in Abhängigkeit der sequentiellen Tiefe (x-Achse) dar. Der *FULL*-Versuch wurde gewählt, da bei diesem die maximale Haftfehlerabdeckung erreicht werden kann. Da die Abhängigkeit zwischen Haftfehlerabdeckung und sequentieller Tiefe der beiden

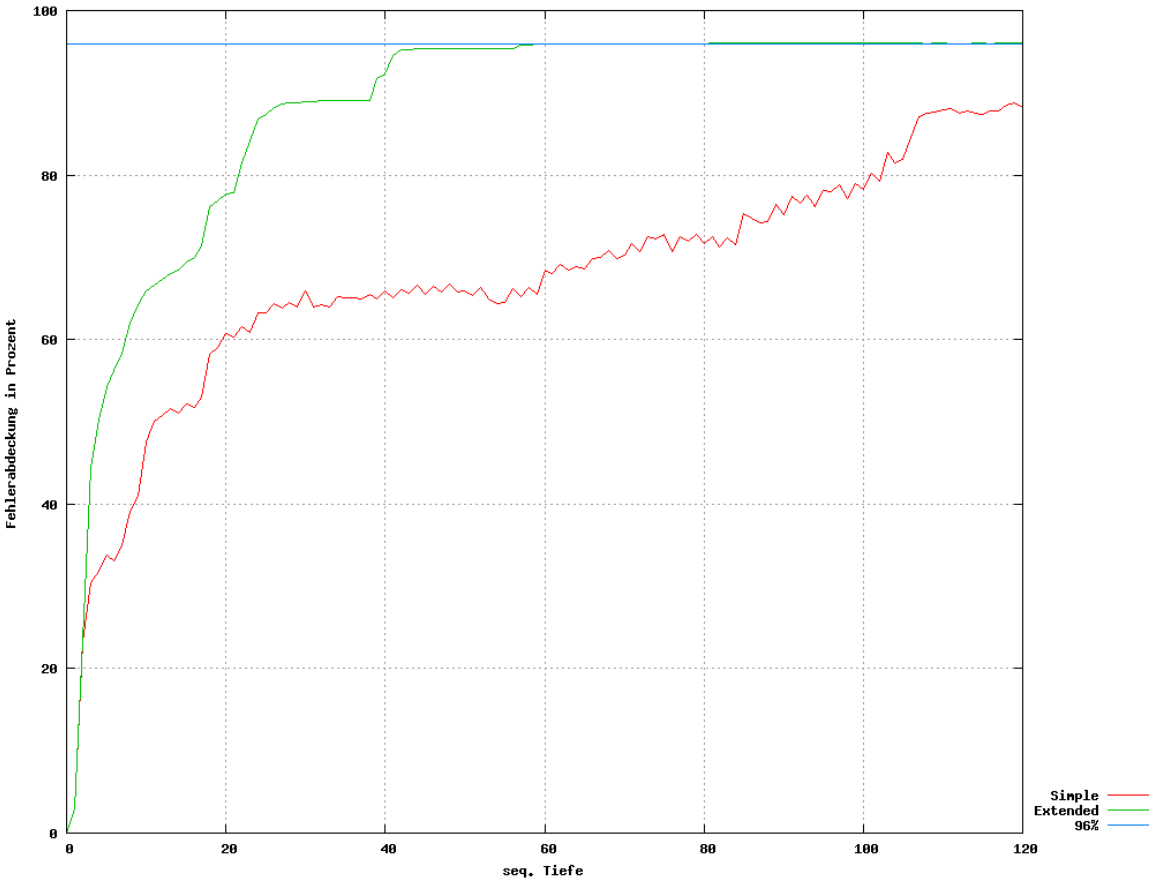


Abbildung 4.1: Prozentuale Fehlerabdeckung des *FULL*-Versuchs.

verwendeten Templates sich erheblich voneinander unterscheiden, werden sie getrennt besprochen.

simple-Template

Die Haftfehlerabdeckung bei Verwendung des *simple*-Template verhält sich uncharakteristisch. Typischerweise sinkt bei der Erhöhung der sequentiellen Tiefe die erzielte Fehlerabdeckung nicht ab, es sei denn, es treten abgebrochene Fehler auf (vgl. Abschnitt 2.2.3), dies ist jedoch im vorliegenden Fall nicht der Grund. Das Verhalten lässt sich dagegen anhand der Einschränkung des *simple*-Templates erklären.
Im Gegensatz zur Verwendung des *extended*-Template, kann das *simple*-Template nicht den Ablauf der Kommunikation verfolgen, sondern lediglich eine Momentaufnahme der Kommunikation nach Ablauf der sequentiellen Tiefe betrachten. Folglich ist es möglich, dass ein Fehler, der z. B. in Takt 25 beobachtbar ist, im nächsten Takt bereits nicht mehr sichtbar und damit nicht testbar ist.

Bei einer sequentiellen Tiefe von 120 erreicht das *simple*-Template eine Gesamthaftfehlerabdeckung von 88,7%. Ein vergleichbarer Wert wird bei Verwendung des *extended*-Template bereits bei einer sequentiellen Tiefe von 27 erreicht.
Da die Fehlerabdeckung des *simple*-Template bis zur Grenze von 120 weiterhin steigt, wird im folgenden diese Tiefe für die *simple*-Templates genutzt.

Extended-Template

Die Haftfehlerabdeckung bei Verwendung des *extended*-Template verhält sich wie erwartet, in Form eines sprunghaften, beschränkten Wachstums.

Anhand des Zustandsautomaten der Masterkomponente (vgl. Abb. 3.2) lassen sich die Sprünge, ebenso wie die Bereiche ohne temporäre Anstiege (im Folgenden als Ebene bezeichnet) erklären.
Während der ersten 21 Takte wird die Sendeschleife (zur Adressierung der Slavekomponente) einmal traversiert. Die Sendeschleife (vgl. Abschnitt 3.1.1) nimmt einen großen Teil der Masterkomponente ein, folglich ermöglicht der Durchlauf dieser Schleife bereits das Testen eines großen Teils der Schaltung (77.8356%).
Der anschließende Anstieg bis zum Erreichen der ersten Ebene erklärt sich durch die Möglichkeit die verschiedenen Verzweigungen des Zustands *GET_D_ACK_2*, in die Empfangsschleife, zurück zum Beginn der Sendeschleife oder zum Abbruch der Kommunikation im Fehlerfall, zu aktivieren. In der folgenden Ebene (zwischen sequentieller Tiefe 24 und 38) bleibt die Gesamtfehlerabdeckung bei 89% konstant. Dieser Stillstand erklärt sich durch wiederholtes Erreichen der *SEND* bzw. *READ* Zustände in der Sende- bzw. Empfangsschleife, während dessen können keine neuen Fehler aktiviert werden.
Ab Tiefe 39 steigt die Abdeckung weiter bis auf einen Wert von 95,4% bei Tiefe 45. Dieser Anstieg erklärt sich durch die mögliche Aktivierung der Byte-Schleife der Sendeschleife und eines protokollgerechten Beendens der Kommunikation (Erreichen des *STOP*-Zustand). Der Wert stellt einen guten Kompromiss zwischen Fehlerabdeckung und Laufzeit dar. Eine Verdopplung der Tiefe auf (90) erreicht lediglich einen Wert von 96.0648%, eine Steigerung von unter 0,64%, bei doppelter Testzeit, je Testmuster.
Für bessere Fehlerabdeckung kann eine sequentielle Tiefe von 60 gewählt werden. Mit diesem Wert kann eine Gesamtfehlerabdeckung von 95,95% erreicht werden. Diese Steigerung um 0,55% erklärt sich durch die Aktivierung höherer Bits des Byte-Zählers.

Die scheinbare Verringerung bei sehr hoher sequentiellen Tiefe erklärt sich durch das Auftreten von abgebrochenen Fehler (vgl. Abschnitt 2.2.3). Für die finale Testreihe wurde aufgrund dieser Beobachtung ein höheres Limit für den Abbruch der Fehleruntersuchung des ATPG-Programms gewählt.

4.3 Experimentelle Ergebnisse

Die in Kapitel 3.2.3 vorgestellten Versuche wurden auf dem Testsystem mit beiden Template-Varianten, für eine sequentielle Tiefe von 60 (*extended*-Template) und 120 (*simple*-Template), untersucht. Die experimentellen Ergebnisse sind in der Tabellen I^2C-Master (4.1), I^2C-Slave (4.2) und Loopback (4.3) aufgelistet. Dabei enthält die erste Spalte der Tabelle stets die Größe der kollabierten Haftfehlerliste

Versuchskürzel	#Haftfehler$_M$	FC_{si120}	#Testmuster$_{si}$	FC_{ex60}	#Testmuster$_{ex}$
$NONE$	904	51.6593%	14	59.9558%	17
N_cnt_m	904	56.4159%	16	60.9513%	15
N_err_m	904	52.3230%	13	62.2788%	15
N_state_m	904	59.4027%	14	70.0221%	17
N_bus	904	61.0619%	17	74.1150%	17
O_lb	904	79.9779%	19	84.5133%	32
$O\_lb\|cnt_m$	904	80.1991%	16	84.9558%	27
$O\_lb\|err$	904	80.6416%	22	86.5044%	32
$O\_lb\|state_m$	904	85.9513%	21	92.5885%	31
$O\_lb\|bus$	904	81.3053%	21	85.7301%	31
$O\_lb\|state_s$	904	80.0885%	20	84.5133%	37
$O\_lb\|cnt_s$	904	80.0885%	20	84.5133%	38
$FULL$	904	87.721%	17	95.2434%	32

Tabelle 4.1: Ergebnisse der Masterkomponente.

der jeweiligen Komponente.
Bei Anwendung aller Maßnahmen lässt sich die Haftfehlerabdeckung der betrachteten Schaltung, abhängig von der untersuchten Komponente und des verwendeten Template, um 40% (Masterkomponente, *extended*-Template) bis zu 76% (Slavekomponente, *simple*-Template) verbessern.
Bei der Testerzeugung wurden keine Fehler abgebrochen, nachdem das Limit, nach der experimentellen Bestimmung der sequentiellen Tiefe, erhöht wurde.

Die Testerzeugung wurde parallelisiert (ein Task je Versuchsversion und Template, 24 insgesamt) auf mehreren Computern (ausgestattet mit je einem Core i7 2600@3,4 GHz und 32GB RAM) ausgeführt. Dabei unterschied sich die Testerzeugungsdauer der beiden in Kapitel 3.3.2 vorgestellten Templates erheblich voneinander. Die parallele Ausführung aller Versuch war bei Anwendung des *simple*-Template, innerhalb von maximal elf Minuten abgeschlossen.
Je nach Testbedingungen dauerte die Testerzeugung bei Verwendung des *extended*-Template bis zu 3,5 Stunden. Dies lässt sich durch die erheblich größere Zahl an zu betrachtenden PIs und POs der Schaltung erklären.

4.4 Analyse der vorgeschlagenen Maßnahmen

Im Folgenden werden die einzelnen Maßnahmen auf deren Wirksamkeit untersucht. Dazu werden die Versuche, welche die Verwendung einer oder mehrerer Maßnahmen vorsehen, mit den beiden Vergleichsreihen ($NONE$ und O_lb) abgeglichen. Diese stellen die unangepasste Schaltung mit (O_lb) bzw. ohne ($NONE$) aktivierter Loopback-Komponente dar.

Versuchskürzel	#Haftfehler$_S$	FC_{si120}	#Testmuster$_{si}$	FC_{ex60}	#Testmuster$_{ex}$
$NONE$	713	14.4460%	2	20.8976%	2
N_cnt_m	713	14.4460%	2	20.8976%	2
N_err_m	713	14.4460%	2	20.8976%	3
N_state_m	713	14.4460%	2	20.8976%	2
N_bus	713	14.4460%	2	20.8976%	2
O_lb	713	85.4137%	27	89.0603%	36
$O\_lb\|cnt_m$	713	85.4137%	27	89.0603%	39
$O\_lb\|err$	713	85.4137%	25	89.0603%	34
$O\_lb\|state_m$	713	85.4137%	28	89.0603%	41
$O\_lb\|bus$	713	86.1150%	28	89.9018%	35
$O\_lb\|state_s$	713	89.9018%	27	96.3534%	38
$O\_lb\|cnt_s$	713	85.8345%	24	89.7616%	39
$FULL$	713	90.6031%	27	97.8962%	33

Tabelle 4.2: Ergebnisse der Slavekomponente.

Versuchskürzel	#Haftfehler$_{LB}$	FC_{si120}	#Testmuster$_{si}$	FC_{ex60}	#Testmuster$_{ex}$
$NONE$	11	27.2727%	1	27.2727%	2
N_cnt_m	11	27.2727%	1	27.2727%	1
N_err_m	11	27.2727%	1	27.2727%	2
N_state_m	11	27.2727%	1	27.2727%	1
N_bus	11	36.3636%	2	36.3636%	2
O_lb	11	90.9091%	3	100%	3
$O\_lb\|cnt_m$	11	90.9091%	3	100%	4
$O\_lb\|err$	11	90.9091%	3	100%	4
$O\_lb\|state_m$	11	90.9091%	3	100%	3
$O\_lb\|bus$	11	100%	4	100%	2
$O\_lb\|state_s$	11	90.9091%	3	100%	3
$O\_lb\|cnt_s$	11	90.9091%	3	100%	3
$FULL$	11	100%	4	100%	2

Tabelle 4.3: Ergebnisse der Loopback-Komponente.

4.4.1 Loopback

Die Ergebnisse zeigen deutlich, wie entscheidend die Verwendung einer Loopback-Komponente (vgl. Abschnitt 3.2.1) für die Fehlerabdeckung beim Test einer Peripheriekomponente ist. Wird diese Anpassung nicht verwendet so wird nur eine maximale Fehlerabdeckung von 74,1% (Master) bzw. 20,9% (Slave) erreicht. Dagegen liegt die geringste, erreichte Fehlerabdeckung, bei Nutzung der Loopback-Komponente, bereits bei 84,5% (Master) bzw.. 89,1% (Slave).
Ohne zusätzliche Maßnahmen (im Fall O_{lb}) erhöht die Loopback-Komponente die Haftfehlerabdeckung um mindestens 24% (Master) bzw. 69% (Slave), verglichen mit $NONE$.

4.4.2 Maßnahmen zur Verbesserung der Beobachtbarkeit

Die Wirksamkeit der Maßnahmen zur verbesserten Beobachtbarkeit des internen Zustands (vgl. Kapitel 3.2) variiert je nach Versuchskontext, nach der verwendeten Template-Variante und nach deren Abhängigkeit von der Loopback-Komponente.
Die beiden wirksamsten der vorgestellten Maßnahmen sind die codierte Ausgabe des internen Zustandsautomaten einerseits und die Sichtbarmachung des I^2C-Bus anderseits. Dabei bewirkt die Ausgabe des Zustands eine deutliche Steigerung der Fehlerabdeckung mit (etwa 8%) und ohne (bis zu 10%) aktivierter Loopback-Komponente. Die Wirksamkeit der Zustandsausgabe unterscheidet sich bei Master- und Slavekomponente. Die Ausgabe des Slavezustands erhöht die Fehlerabdeckung der Slavekomponente lediglich um bis zu 7%. Dies ist auf einen trivialen Zusammenhang zwischen der Größe des Zustandsautomaten und der betrachteten Schaltung zurückzuführen.
Interessant ist, dass bereits die Ausgabe eines Zustands-Bit (in den Versuchen der *Error*-Zustand des Masters) eine deutliche Verbesserung (bis zu 2,3%) der Fehlerabdeckung ermöglicht.

Die Beobachtbarkeit des I^2C-Bus verliert bei aktivierter Loopback-Komponente stark an Wirksamkeit (unter 1%), verglichen mit einer hohen Steigerung der Fehlerabdeckung bei deaktiviertem Loopback (bis 14%). Interessant ist, dass die Beobachtung des Busses die vollständige Prüfung der Loopback-Komponente bei Verwendung des *simple*-Templates erlaubt.

Die Ausgabe der internen Schleifenzähler bewirkt bei Verwendung des *extended*-Templates nur eine geringe Verbesserung der Fehlerabdeckung (etwa 1%), während im Fall der Verwendung des *simple*-Templates und deaktivierter Loopback-Komponente eine Steigerung der Fehlerabdeckung von etwa 5% erreicht wird.
Dies lässt sich durch die Tatsache erklären, dass Fehler im Schleifenzähler einfach zu beobachtende Auswirkungen auf die Funktionalität der Komponente besitzen (führt z. B. dazu dass stets dasselbe Bit übertragen bzw. gelesen wird und die Sende- bzw. Empfangsschleife nicht verlassen wird). Diese Auswirkungen lassen sich leicht feststellen, wenn der Zustand ausgegeben wird, der Bus beobachtbar ist oder ein Loopback zwischen I^2C-Master und Slavebesteht. Da die Ausgabe des codierten Zustands im vorliegenden Fall nahezu dieselbe Anzahl an Pins benötigt, wie die Ausgabe des Schleifenzählers, ist diese Ausgabe des Zustands klar vorzuziehen.

4.5 Analyse der Templates

Der Einfluss der in Kapitel 3.3.2 vorgestellten Templates auf die erreichbare Fehlerabdeckung, die betrachtete sequentielle Tiefe und die Anzahl der erzeugter Testmuster wird aus der experimentellen Bestimmung der sequentiellen Tiefe (in Abschnitt 4.2) und den experimentellen Ergebnissen (in den Tabellen 4.1, 4.2 und 4.3) deutlich.

Bei Nutzung des *extended*-Templates kann, in nahezu allen betrachteten Fällen, eine deutlich höhere Fehlerabdeckung erreicht werden, als bei Verwendung des *simple*-Templates. Die Ausnahme stellt dabei lediglich das Testen der Loopback-Komponente dar. Ist diese deaktiviert, können mit beiden Templates dieselben Fehler getestet werden.
Bemerkenswert ist die Tatsache, dass das *extended*-Template auf einer sequentiellen Tiefe von 60 eine bessere Haftfehlerabdeckung erreicht, als das *simple*-Template bei der doppelten sequentiellen Tiefe 120. Allerdings werden in den meisten Fällen, bei Verwendung des *extended*-Templates, deutlich mehr Testmuster erzeugt, als bei Verwendung des *simple*-Template.

4.5.1 Programmgröße

Die Programmgröße bei Verwendung des *simple*-Templates ist erheblich geringer, als die des *extended*-Templates. Dies geht eindeutig aus den Formeln 3.7 und 3.9 hervor. Zusätzlich werden bei Anwendung des *extended*-Templates auch mehr Testmuster erzeugt.

Wie groß dieser Unterschied tatsächlich ausfällt, wird im Folgenden am Beispiel des *FULL*-Versuchs berechnet. Dazu werden die erzeugten Testmuster für den Test der Masterkomponente (vgl. Tabelle 4.1) in die Formeln 3.7 und 3.9 eingesetzt.

(4.1) $n_{simple|Speicherbedarf} \geq (25 * 17 + 10) * 4 = 1740Bytes = 1,7kBytes$

(4.2) $n_{extended|Speicherbedarf} \geq (17 * 32 * 60 + 10) * 4 = 130600Bytes = 127,6kBytes$

4.5.2 Laufzeit

Für die untersuchten Fälle (*simple*$_{120}$ und *extended*$_{60}$) zeigt sich bei Vergleich der Formeln 3.11 und 3.13, dass *extended*-basierte Testprogramme interessanterweise eine kürze Laufzeit besitzen. Dies lässt sich mit der Beobachtung erklären, dass nie mehr als die doppelte Zahl von Testmustern für das *extended*-Template erzeugt werden, die sequentielle Tiefe jedoch nur halb so groß ist wie bei Verwendung des *simple*-Templates.

Beispielhaft für diese Beobachtung wird die Laufzeit der beiden Templates für den Testfall *FULL*, angewandt auf die Masterkomponente, mithilfe der Formeln 3.11 bzw. 3.13 berechnet. Für $k_{Taktteiler}$ wird der in Abschnitt 3.3.2 berechnete Mindestwert für das *extended*-Template ($k_{Taktteiler} = 34$) verwendet.

(4.3) $t_{simple|120} = 17 * (120 * 34 + 23) + 10 = 69761 \quad Systemtakte$

(4.4) $t_{extended|60} = 32 * (60 * 34 + 17) + 10 = 65834$ *Systemtakte*

Bei einer Taktrate von 50 MHz des miniMIPS Prozessors ergibt sich damit:

(4.5) $t_{simple|120} = \frac{69761}{50000000} = 1,4ms$

(4.6) $t_{extended|60} = \frac{65834}{50000000} = 1,3ms$

5 Fazit

Bei der Anwendung von SBST-Techniken auf Peripheriekomponenten kann die erreichbare Haftfehlerabdeckung durch Maßnahmen zur Verbesserung der Kontrollierbarkeit und Beobachbarkeit interner Signale erheblich verbessert werden.
Experimentelle Ergebnisse für die Anwendung der strukturellen SBST Methode auf der implementierten I^2C-Komponente zeigen eine erreichbare Gesamtfehlerabdeckung von 96%, bei Verwendung aller vorgeschlagenen Maßnahmen gegenüber einer Abdeckung von 37% ohne diese Anpassungen. Die erfolgversprechendste Maßnahme ist dabei das Einfügen eines Loopbacks. Diese erlaubt ein intensives Testen der Komponente, da Schaltungsbereiche getestet werden, die bei einer einzelnen Komponente nicht aktiviert werden können. Durch die Sichtbarmachung diverser internen Signale lässt sich die Haftfehlerabdeckung, je nach Testfall, um weitere drei bis neun Prozent erhöhen. Von den untersuchten Maßnahmen zur Erhöhung der Beobachtbarkeit der internen Signale erweist sich die codierte Ausgabe des aktuellen Zustandsautomaten der Komponenten als besonders wirksam.

Als Resultat wurden zwei unterschiedliche Test-Templates entworfen, die in Abhängigkeit von den funktionalen Nebenbedingungen, eingesetzt werden können um Testprogramme zu erzeugen. Die Laufzeit, Programmgröße und erreichbare Fehlerabdeckung der Templates unterscheidet sich erheblich voneinander und zeigt den Einfluss funktionaler Nebenbedingungen auf die Testerzeugung.
Erlauben die Nebenbedingungen die Durchführung eines Testprogramms, welches in jedem Takt die PIs beschreiben und POs auslesen kann, so lässt sich eine höhere Testabdeckung erzielen, bei einer niedrigeren betrachteten sequentiellen Tiefe und höherer Wirksamkeit der vorgestellten Maßnahmen. Allerdings steigt dabei der benötigte Speicherplatz des Testprogramms erheblich an.

6 Ausblick

Um die Vergleichbarkeit der vorgestellten strukturellen SBST-Methode für Kommunikationsperipheriekomponenten mit bestehenden Arbeiten zu erhöhen, könnte eine zukünftige Arbeit die vorgestellte Methode auf eine der in [APGP07b] und [BSS+07] genutzten Peripheriekomponenten angewandt werden.
Von Interesse ist dabei, neben der erreichbaren Haftfehlerabdeckung, vor allem wie sich die vorgeschlagenen Verbesserungen auf komplexere Schaltungen auswirken und der Unterschied in der Fehlerabdeckung der unterschiedlichen Template-Varianten.

Aufgrund der herausragenden Bedeutung einer Loopback-Komponente auf die erreichbare Haftfehlerabdeckung, könnte zudem die Implementierung solcher Komponenten für verschiedene Kommunikationsprotokolle stattfinden.

Des Weiteren kann untersucht werden, wie sich ein nicht-beobachtbarer und nicht-kontrollierbarer Slave, der über den I^2C-Bus an den Master angeschlossen ist, als Kommunikationspartner auf die erreichbare Haftfehlerabdeckung auswirken würde.

Außerdem kann eine weitere Optimierung der vorgeschlagenen Templates untersucht werden, z. B. wie sich der Speicherplatzbedarf des *extended*-Templates verringern lässt, ohne die erreichbare Fehlerabdeckung signifikant zu reduzieren.

Literaturverzeichnis

[AGP+09] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, M. S. Reorda. "Test program generation for communication peripherals in processor-based SoC devices". *IEEE Transactions on Design & Test of Computers*, 26(2):52–63, 2009. (Zitiert auf den Seiten 21, 25, 26 und 27)

[ALR+01] A. Avizienis, J.-C. Laprie, B. Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001. (Zitiert auf Seite 16)

[APGP07a] A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis. "A functional self-test approach for peripheral cores in processor-based SoCs". In *Proceeding of International On-Line Testing Symposium*, S. 271–276. IEEE, 2007. (Zitiert auf den Seiten 7 und 26)

[APGP07b] A. Apostolakis, M. Psarakis, D. Gizopoulos, A. Paschalis. "Functional processor-based testing of communication peripherals in systems-on-chip". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):971–975, 2007. (Zitiert auf den Seiten 21, 26, 27, 49 und 59)

[BA01] M. L. Bushnell, V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory, and Mixed-Signal VLSI Circuits*. Kluwer, 2001. (Zitiert auf den Seiten 16, 17, 18, 19, 20, 22 und 24)

[BBF+10] O. Ballan, P. Bernardi, G. Fontana, M. Grosso, E. Sánchez. "A Fault Grading Methodology for Software-Based Self-Test Programs in Systems-on-Chip". In *Proceedings of International Workshop on Microprocessor Test and Verification*, S. 43–46. IEEE, 2010. (Zitiert auf Seite 7)

[BSS+07] L. Bolzani, E. Sánchez, M. Schillaci, M. S. Reorda, G. Squillero. "An automated methodology for cogeneration of test blocks for peripheral cores". In *Proceedings of International On-Line Testing Symposium*, S. 265–270. IEEE, 2007. (Zitiert auf den Seiten 7, 25, 26, 27, 49 und 59)

[CCRS00] F. Corno, G. Cumani, M. S. Reorda, G. Squillero. "An RT-level fault model with high gate level correlation". In *Proceedings of High-Level Design Validation and Test Workshop*, S. 3–8. IEEE, 2000. (Zitiert auf Seite 25)

[CD01] L. Chen, S. Dey. "Software-based self-testing methodology for processor cores". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):369–380, 2001. (Zitiert auf Seite 7)

[CRRD03] L. Chen, S. Ravi, A. Raghunathan, S. Dey. "A scalable software-based self-test methodology for programmable processors". In *Proceedings of Annual Design Automation Conference*, S. 548–553. ACM, 2003. (Zitiert auf Seite 24)

[CWLG07] C.-H. Chen, C.-K. Wei, T.-H. Lu, H.-W. Gao. "Software-based self-testing with multiple-level abstractions for soft processor cores". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):505–517, 2007. (Zitiert auf Seite 24)

[DBG03] J. Dushina, M. Benjamin, D. Geist. "Semi-formal test generation and resolving a temporal abstraction problem in practice: industrial application". In *Proceedings of Asia and South Pacific Design Automation Conference*, S. 699–704. ACM, 2003. (Zitiert auf Seite 27)

[DCPS11] S. Di Carlo, P. E. Prinetto, A. Savino. "Software-based self-test of set-associative cache memories". *IEEE Transactions on Computers*, 60(7):1030–1044, 2011. (Zitiert auf Seite 7)

[GHS+12] M. Grosso, W. P. Holguin, E. Sánchez, M. S. Reorda, A. Tonda, J. V. Medina. "Software-Based Testing for System Peripherals". *Journal of Electronic Testing*, 28(2):189–200, 2012. (Zitiert auf den Seiten 7, 11, 25 und 27)

[GPR+10a] M. Grosso, H. Perez, D. Ravotto, E. Sanchez, M. S. Reorda, J. V. Medina, et al. "A software-based self-test methodology for system peripherals". In *Proceedings of European Test Symposium*, S. 195–200. IEEE, 2010. (Zitiert auf Seite 7)

[GPR+10b] M. Grosso, W. Perez, D. Ravotto, E. Sanchez, M. Reorda, J. Medina. "Functional test generation for DMA controllers". In *Proceedings of Latin American Test Workshop*, S. 1–6. IEEE, 2010. (Zitiert auf Seite 11)

[GVA06] S. Gurumurthy, S. Vasudevan, J. A. Abraham. "Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor". In *Proceedings of International Test Conference*, S. 1–9. IEEE, 2006. (Zitiert auf Seite 24)

[GW00] S. Gerstendörfer, H.-J. Wunderlich. "Minimized power consumption for scan-based BIST". *Journal of Electronic Testing*, 16(3):203–212, 2000. (Zitiert auf Seite 22)

[JRW14] A. Jutman, M. S. Reorda, H.-J. Wunderlich. "High Quality System Level Test and Diagnosis". In *Proceedings of Asian Test Symposium*, S. 298–305. IEEE, 2014. (Zitiert auf Seite 15)

[KLC+02] A. Krstic, W.-C. Lai, K.-T. Cheng, L. Chen, S. Dey. "Embedded software-based self-test for programmable core-based designs". *IEEE Transactions on Design & Test of Computers*, 19(4):18–27, 2002. (Zitiert auf Seite 22)

[KMT+08] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, D. Gizopoulos. "Hybrid-SBST methodology for efficient testing of processor cores". *IEEE Transactions on Design & Test of Computers*, 25(1):64–75, 2008. (Zitiert auf den Seiten 7 und 25)

[KPGX07] N. Kranitis, A. Paschalis, D. Gizopoulos, G. Xenoulis. "Software-based self-testing of embedded processors". In *Processor Design*, S. 447–481. Springer, 2007. (Zitiert auf den Seiten 7, 21 und 22)

[LJ07] L. Lingappan, N. K. Jha. "Satisfiability-based automatic test program generation and design for testability for microprocessors". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(5):518–530, 2007. (Zitiert auf Seite 24)

[NAS15] The NASM Development Team. *"NASM — The Netwide Assembler"*, 2015. Rev. 6. (Zitiert auf Seite 41)

[NXP14] NXP Semiconductors. *"I2C-bus specification and user manual"*, 2014. Rev. 6. (Zitiert auf den Seiten 12, 15, 30, 31 und 33)

[PG05] A. Paschalis, D. Gizopoulos. "Effective software-based self-test strategies for on-line periodic testing of embedded processors". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24(1):88–99, 2005. (Zitiert auf Seite 7)

[PGSR10] M. Psarakis, D. Gizopoulos, E. Sanchez, M. S. Reorda. "Microprocessor software-based self-testing". *IEEE Transactions on Design & Test of Computers*, 27(3):4–19, 2010. (Zitiert auf den Seiten 7, 17, 21, 22, 23 und 24)

[SA98] J. Shen, J. A. Abraham. "Native mode functional test generation for processors with applications to self test and design validation". In *Proceedings of International Test Conference*, S. 990–999. IEEE, 1998. (Zitiert auf Seite 26)

[TKPG14] G. Theodorou, N. Kranitis, A. Paschalis, D. Gizopoulos. "Software-Based Self-Test for Small Caches in Microprocessors". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(12):1991–2004, 2014. (Zitiert auf Seite 7)

[WWC+06] C.-P. Wen, L. Wang, K.-T. Cheng, et al. "Simulation-based functional test generation for embedded processors". *IEEE Transactions on Computers*, 55(11):1335–1343, 2006. (Zitiert auf Seite 24)

[WWW06] L.-T. Wang, C.-W. Wu, X. Wen. *VLSI Test Principles and Architectures: Design for Testability*. Academic Press, 2006. (Zitiert auf den Seiten 16, 17, 19 und 21)

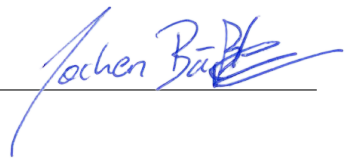
[Zho09] J. Zhou. *Software-Based Self-Test under Memory, Time and Power Constraints*. University of Stuttgart, 2009. (Zitiert auf Seite 25)

Alle URLs wurden zuletzt am 05. 05. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Remseck, den 05.05.2015



Ort, Datum, Unterschrift