Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Entwicklung einer Tcl-basierten Methode zur Fehlerinjektion auf Xilinx-FPGAs

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science (M.Sc.)

vorgelegt von

Murat Simsek

3348138

Erstgutachter:	Prof. Dr. Oliver Bringmann
Zweitgutachter:	Prof. Dr. Wolfgang Rosenstiel
Betreuer:	Dr. Thomas Schweizer

15. Juli 2015

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Masterarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Ort, Datum Unterschrift

Zusammenfassung

In dieser Arbeit wurde eine Tcl-App implementiert, welche den regulären Vivado Design Flow um die Möglichkeit zur Injektion von permanenten Fehlern erweitert. Die Vivado Design Suite ist eine integrierte Designumgebung, die zur Implemetierung von Anwendungen auf die neuen voll programmierbaren Xilinx-FPGAs der Serie 7 entwickelt wurde. Vivado enthält eine Reihe von Tcl-Befehlen zur Kontrolle des gesamten Designablaufs bei der Entwicklung digitaler Schaltungen.

Die implementierte Tcl-App ist in der Lage, basierend auf dem Vivado-Designflow mittels Tcl-Befehlen Stuck-at-Fehler in beliebige Schaltungen zu injizieren und das fehlerbehaftete Design per Simulation oder Emulation zu verifizieren. Das entwickelte Verfahren basiert auf dem Ansatz der Serial Fault Emulation, bei der die Injektion von Fehlern in der RTL-Netzliste stattfindet. Bei dieser Methode wird eine Schaltung für jeden zu injizierenden Fehler synthetisiert und nach der Platzierung und der Verdrahtung auf einem FPGA zur Ausführung gebracht. Diese zeitintensiven Prozesse werden hingegen bei der hier entwickelten Methode nur zu Beginn der Fehlerinjektion einmalig durchgeführt, indem die Injektion der Fehler auf das synthetisierte bzw. implementierte Design verlagert wird. Diese stellt eine erhebliche Verbesserung gegenüber der generischen Serial Fault Emulation dar und wird durch das in dieser Arbeit entwickelte Mapping erreicht.

Das Mapping ermöglicht die Zuweisung von Fehlern an Input-Ports und -Pins des RTL-Designs. Diese Objekte werden darauffolgend zur Fehlerinjektion im FPGA-Design referenziert. Dies erfolgt durch die vom Mapping erzeugten Injektoren, welche die erforderlichen Informationen wie den Namen der Selektion, den Typ der zu injizierenden Fehler usw. für das Mapping bereitstellt. Basierend auf den erzeugten Injektoren werden alle zuvor selektierten Ports und Pins im FPGA-Design lokalisiert. Anschließend wird die komponentenbasierte Anpassung des FPGA-Designs abhängig vom ausgewählten Fehler vorgenommen.

Mit der vorliegenden Implementierung der Fehlerinjektion stehen alle Vorteile der Vivado Design Suite zur Verfügung. Neben der Fehleremulation kann ein fehlerbehaftetes Design auch in den früheren Entwicklungsphasen wie z.B. in der Post-Synthesis-Phase simuliert werden.

für die Menschen,
die selbst das Exil mit ihrem Lächeln
verschönern können.[*]

[*]sürgünlükte bile yaşamı
gülüşleriyle güzelleştirebilen insanlara

Inhaltsverzeichnis

Abbildungsverzeichnis **v**

Tabellenverzeichnis **vii**

Abkürzungsverzeichnis **vii**

1 Einleitung **1**

1.1 Zielsetzung 1

1.2 Struktur dieser Arbeit 2

2 Grundlagen **5**

2.1 Field Programmable Gate Arrays 5

2.1.1 Aufbau der Xilinx 7-Series FPGA 5

2.1.2 Configurable Logic Block (CLB) 6

2.1.3 Look-Up Table (LUT) 9

2.1.4 Block-RAM 9

2.1.5 Input/Output Blocks (IOBs) 10

2.2 Vivado Design Suite 10

2.2.1 Designablauf 10

2.2.2 Design-Checkpoints 11

2.2.3 Design-Constraints 11

2.2.4 Design-Netlist 12

2.2.5 Design-Objekte 12

2.3 Haftfehlermodell 13

3 Stand der Technik **15**

4 Implementierung **17**

4.1 Terminologie 18

4.2 Ablauf der Fehlerinjektion in Vivado 19
4.3 Mapping 21
4.3.1 Injektoren für Ports 23
4.3.2 Injektoren für Modul-Pins 24
4.3.3 Injektoren für Leaf-Pins 26
4.3.3.1 UNISIM-Instanzen und Register 26
4.3.3.2 Logische Gatter 27
4.3.3.3 Multiplexer 27
4.4 Fehlerinjektion 29
4.4.1 Logische Konstante 29
4.4.2 Look-Up Table 30
4.4.2.1 Boolesche Algebra 31
4.4.2.2 INIT to DNF 31
4.4.2.3 LUT-Basierte Fehlerinjektion 33
4.4.2.4 DNF to INIT 34
4.4.3 Read-Only Memory 34
4.4.4 Block-RAM 35
4.4.5 Weitere Instanzen 35

5 Ablauf der Fehlerinjektion 37
5.1 Verzeichnisstruktur und Ablaufkontrolle 37
5.1.1 Anlegen von Testmodulen 39
5.2 Injektoren 41
5.3 Phasen der Fehlerinjektion 43
5.4 Tcl-basierter Designflow 46
5.5 GUI-basierter Designflow 47

6 Ergebnisse 51
6.1 Testumgebung 52
6.2 Abdeckungsgrad des Mappings 52

6.2.1 Diskussion der Ergebnisse 54
6.3 Ausführungszeiten 55
6.3.1 Diskussion der Ergebnisse 59

7 Ausblick **63**

Literaturverzeichnis **65**

A Referenzschaltung **67**
A.1 2-Bit-Addierer in Verilog 68
A.2 Tcl-Code des Ablaufs der Fehlerinjektion 72

Abbildungsverzeichnis

1.1 Ablauf der Fehlerinjektion im Vivado 3

2.1 Struktur (links) und Anordnung (rechts) der CLBs eines FPGAs der Serie 7 [Xil14a] 6
2.2 Aufbau einer SLICEM [Xil14a] 7
2.3 Fast Carry Logic Path in einem Slice [Xil14a] 8
2.4 Dual-Port Block-RAM in einem FPGA der Serie 7 [Xil14b] . . . 9
2.5 Tcl-Befehle des Designflow in Project-Mode und Non-Project-Mode [Xil14f] 11
2.6 Fist-Class Design-Objekte in Vivado 12

3.1 Vergleich der Fehlerinjektion bei der Serial Fault Emulation (1) und dem in [Pet12] entwickelten Verfahren (2) 16

4.1 Design-Objekte einer Schaltung 19
4.2 Ablauf der Fehlerinjektion in Vivado 20
4.3 Mapping der RTL-Netzliste auf das FPGA-Design 22
4.4 Beispiel für die Selektion eines Modul-Pins zur Fehlerzuweisung 25
4.5 Keep 28

5.1 Verzeichnisstruktur des implementierten Tools 38
5.2 Ablauf der funktionalen Simulation in Vivado [Xil14h] 46
5.3 Zustandsdiagramm des Ablaufs der Fehlerinjektion in Vivado . . 47
5.4 Simulation der fehlerfreien Referenzschaltung 50
5.5 Simulation der Referenzschaltung mit einem Stuck-at-1-Fehler . 50

6.1 Darstellung der Ergebnisse des Mappings 54
6.2 Ablauf der Fehlerinjektion in Post-Synthesis-Phase 55
6.3 Ablauf der Fehlerinjektion in Post-Implementation-Phase 56

6.4 Zeitverhalten der Fehlersimulation und der Fehleremulation in unterschiedlichen Phasen des Designflows 60
6.5 Vergleich der Laufzeiten der Platzierung, Verdrahtung und des partiellen Routings 61
6.6 Vergleich der Laufzeiten der Simulation und Emulation 61

A.1 Schaltplan eines 2-Bit-Addierers 67

Tabellenverzeichnis

4.1 Ableitung des INIT-Wertes aus der Wertetabelle der logischen Funktion `I0 & !I2 + !I1 & I2`. 32
4.2 Ableitung des INIT-Wertes aus der Wertetabelle der durch einen Stuck-at-0-Fehler modifizierten Funktion `!I1 & I2` 34

6.1 Die Ergebnisse des Mappings bei ausgewählten Testschaltungen 53
6.2 Ausführungszeiten der Post-Synthesis-Simulation (in Millisekunden) 57
6.3 Ausführungszeiten der Post-Synthesis-Emulation (in Millisekunden) 58
6.4 Ausführungszeiten der Post-Implementation-Simulation (in Millisekunden) 58
6.5 Ausführungszeiten der Post-Implementation-Emulation (in Millisekunden) 58
6.6 Ausführungszeiten des regulären Vivado Designflows 59

Abkürzungsverzeichnis

ASCII	American Standard Code for Information Interchange
ASIC	Application-Specific Integrated Circuit
CAD	Computer Aided Design
CLB	Configurable Logic Block
DBF-Injektion	Design-basierte Fehlerinjektion
DCP	Xilinx Design Checkpoints
DSP	Digital Signal Processing
EDIF	Electronic Design Interchange Format
FIFO	First-In First-Out
FPGA	Field Programmable Gate Array
GND	Ground, Massepotential
HDL	Hardware Description Language
LUT	Look-Up Table
MBF-Injektion	Mapping-basierte Fehlerinjektion
RAM	Random Access Memory
ROM	Read-Only Memory
RTL	Register-Transfer-Level
SRAM	Static Random Access Memory
SoC	Systems-on-Chip
Tcl	Tool Command Language
VCC	Positive Versorgungsspannung
Vivado	Xilinx Vivado Design Suite
XDC	Xilinx Design Constraints
XDL	Xilinx Design Language

1 Einleitung

Die steigende Integrationsdichte und fortschreitende Miniaturisierung in der Halbleitertechnologie führt zu einer erhöhten Anfälligkeit gegenüber Defekten bei integrierten digitalen Systemen, die heutzutage in fast allen Bereichen des alltäglichen Lebens Anwendung finden. Um dennoch einen zuverlässigen Betrieb gewährleisten zu können, sind Verfahren zur Validierung der Zuverlässigkeit von integrierten Schaltkreisen notwendig, mit deren Hilfe der Einfluss von Fehlern auf die Schaltung bewertet werden können.

Zur Bewertung der Zuverlässigkeit und der erreichten Fehlertoleranz bei der Entwicklung der integrierten Schaltkreise werden häufig simulationsbasierte oder emulationsbasierte Fehlerinjektionstechniken eingesetzt. Bei diesen Methoden wird ein Fehler in ein Zielsystem eingebracht, um das Systemverhalten in einem Fehlerfall zu untersuchen. Während der Fehlerinjektion muss dann eine geeignete Testumgebung die Eingangssignale der Testbench bereitstellen und sowohl die Ansteuerung der Fehlerinjektion als auch die Auswertung der Ausgangssignale übernehmen.

Die hohe Ausführungsgeschwindigkeit und die Flexibilität von SRAM-basierten Field Programmable Gate Arrays (FPGAs) machen diese auch für die Bewertung der Zuverlässigkeit und der Fehlertoleranz interessant. Obwohl SRAM-basierte FPGAs hauptsächlich für die logische Emulation von Application Specific Integrated Circuits (ASICs) verwendet werden, können diese Bausteine durch die Möglichkeit der Rekonfiguration auch zur Emulation von Schaltungen im Fehlerfall verwendet werden. Im Gegensatz dazu können simulationsbasierte Ansätze schon in frühen Phasen des Entwurfsprozesses angewendet werden und ermöglichen dadurch die Senkung der Kosten für die Behebung eines Fehlers. Weiterhin können mit dieser Methode verschiedenste Fehlerarten analysiert werden, jedoch ist die Simulationsgeschwindigkeit sehr gering.

1.1 Zielsetzung

Das Ziel dieser Arbeit ist es, ein Tool Command Language (Tcl) Werkzeug basierend auf der neuen Xilinx-Designumgebung zu entwickeln, welche die Injektion permanenter Fehler in beliebigen Schaltungen ermöglicht. Zur Entwicklung ihrer neuen FPGAs der Serie 7 hat die Firma Xilinx eine neue Entwicklungsumgebung namens Vivado Design Suite (kurz: Vivado) implementiert. Diese neue Designumgebung bietet einen neuen Tcl-basierten Designfluss und ersetzt alle bisherigen Werkzeuge wie ngdbuild, par und bitgen, welche mit der ISE Design Suite zur Verfügung gestellt wurden.

Die bekannten CAD-Frameworks wie RapidSmith und TORC, die zur Erforschung und Entwicklung von FPGA-Designs implementiert wurden, basieren auf der Xilinx Design Language (XDL), welche mit der Einführung von Vivado abgelöst wurde. Die XDL-Datei stellt die Konfiguration und Platzierung eines Designs im ASCII-Format dar und dient als Eingabe bei CAD-Frameworks und zur Anpassung des FPGA-Designs.

Basierend auf XDL und RapidSmith wurde bereits in [Pet12] eine Java-Bibliothek entwickelt, welche in der Lage ist, permanente Fehler in beliebige Schaltungen zu injizieren. In der vorliegenden Arbeit soll eine äquivalente Tcl-basierte Methode für die neue Vivado-Designumgebung entwickelt werden. Dabei soll in der ersten Phase die Realisierbarkeit eines Fehlerinjektionssystems mit reinen Tcl-Befehlen untersucht werden. In der zweiten Phase soll eine Tcl-App zur Injektion von permanenten Fehlern implementiert werden.

Wie es schon in [Pet12] detailliert beschrieben ist, erweitert der zu entwickelnde/implementierte Ansatz die Serial Fault Emulation, bei der die Hardwarebeschreibung für jeden zu injizierendem Fehler angepasst, synthetisiert und implementiert wird. Dabei führt die erneute Durchführung der Synthese und Implementierung für jeden injizierten Fehler bei großen Schaltungen zu langen Laufzeiten. Daher werden diese zeitaufwändige Prozesse bei dem implementierten Ansatz nur einmalig durchgeführt, um auch bei großen Schaltungen eine schnellere Fehlerinjektion zu ermöglichen. Dieser wird durch das so genannte Mapping realisiert. Die Abbildung 1.1 zeigt den implementierten Ablauf der Fehlerinjektion in der Vivado-Designumgebung.

1.2 Struktur dieser Arbeit

Die vorliegende Arbeit besteht aus sieben Kapiteln, deren Inhalt nachfolgend kurz zusammengefasst wird.

Kapitel 2 gibt einen Überblick über den grundlegenden Aufbau der Xilinx-FPGAs der Serie 7 sowie einen Einstieg in die Vivado Design Suite. Im letzten Teil dieses Kapitels wird das Thema Fehlermodelle sowie die Fehlersimulation und Fehleremulation betrachtet.

Kapitel 4 stellt den Hauptteil dieser Arbeit dar. Nachdem einige Begriffe erläutert wurden, wird im zweiten Abschnitt das Mapping ausführlich vorgestellt, welches den Kernteil des implementierten Ansatzes ausmacht. Im letzten Abschnitt erfolgt die Darstellung der Implementierung zur Fehlerinjektion.

Kapitel 5 beschreibt detailliert den Ablauf der Fehlerinjektion, die den Vivado Design Flow um die Möglichkeit der Injektion von permanenten Fehlern erweitert.

Im Kapitel 6 werden die ermittelten Ergebnisse des in dieser Arbeit umgesetzten Ansatzes in Bezug auf einige Testschaltungen dargestellt und disku-

tiert.

Anschließend wird in Kapitel 7 eine kurze Zusammenfassung der Arbeit sowie ein Überblick auf mögliche Erweiterungen des implementierten Tools gegeben.

Im Anhang findet sich ein Schaltplan für einen 2-Bit-Volladdierer sowie deren Verilog- und Testbench-Code, die als Referenzschaltung in vielen Stellen dieser Arbeit für die graphische Darstellung einzelner Schritte verwendet wurde.

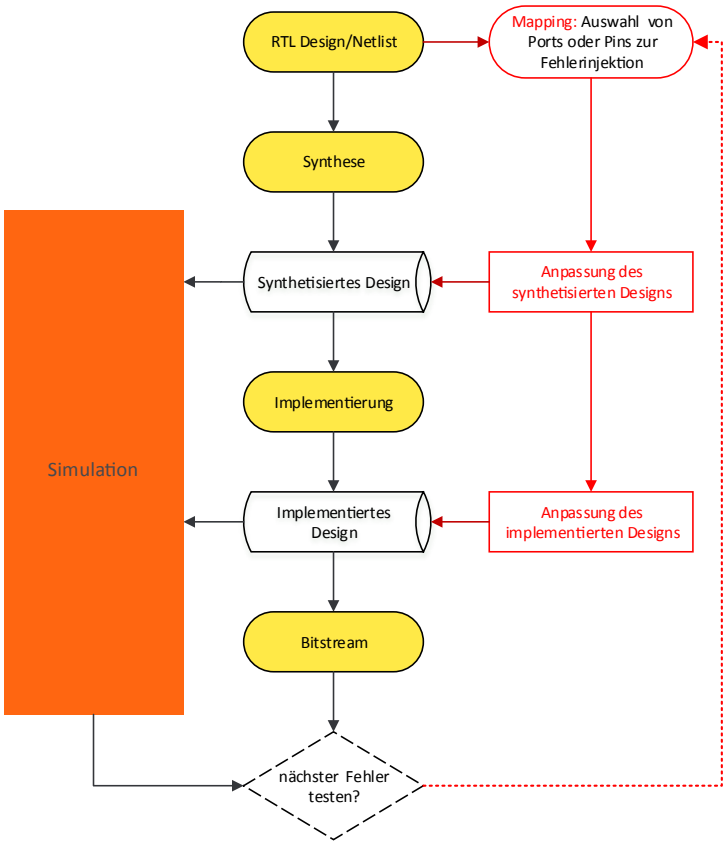

Abbildung 1.1: Ablauf der Fehlerinjektion im Vivado

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe beschrieben, die zum Verständnis dieser Arbeit notwendig sind und in den folgenden Kapiteln häufig verwendet werden.

2.1 Field Programmable Gate Arrays

Field Programmable Gate Array (FPGA) wird als programmierbare integrierte Schaltung bezeichnet, welche durch die Konfiguration vom Benutzer ihre Funktion erhalten. In der Praxis finden FPGAs häufig als Prototypen für die Emulation von komplexeren Schaltungen, Eingebetteten Systemen oder Rechnerbeschleuniger Anwendung [LB13]. Ein FPGA besteht aus in einer Matrix angeordneten Logik-Blöcken und einem zwischen diesen liegenden Verbindungsnetzwerk (Schaltmatrix). Er erhält seine spezifische Funktion durch die Programmierung dieser Blöcke und der Schaltmatrix. Außerdem verbinden die Input/Output-Blöcke, die mit den Pins des FPGAs verbunden sind, das FPGA nach außen.

Technologisch werden FPGAs nach Art der Speicherung ihrer Konfiguration in unterschiedliche Klassen unterteilt. Bei den meist verbreiteten SRAM-basierten FPGAs wird die Konfiguration in SRAM-Speicherzellen gespeichert. Die gleichen Bausteine der SRAM-basierten FPGAs können dabei durch Rekonfiguration beliebig oft für die Implementierung einer neuen Funktionalität verwendet werden. Bei Flash-basierten FPGAs wird die Hardwarekonfiguration in nichtflüchtigen Speichern hinterlegt und ihre Bausteine werden nach dem Einschalten programmiert. Bei Antifuse-basierten FPGAs werden bei der Konfiguration die leitenden Verbindungen durch Anlegen der Programmierspannung dauerhaft erzeugt. Daher fehlt bei diesen FPGAs die Programmierbarkeit. Nachfolgend wird der Aufbau der neuen Xilinx-FPGAs der Serie 7 vorgestellt.

2.1.1 Aufbau der Xilinx 7-Series FPGA

Die derzeit aktuelle Xilinx Produktfamilie besteht aus Virtex-7-, Kintex-7- und Artix-7-FPGAs und dem Zynq-7000 All Programmable SoC. Alle FPGAs der Serie 7 beinhalten unter anderem Logik-Zellen, Block-RAMs, DSP (Digital Signal Processing) Slices, Takt-Seuereinheiten (Clock Management Tiles, CMT) und Blöcke für PCI-Express [Xil14c].

In Folgendem werden die wesentlichen Ressourcen der 7-Serie FPGAs vorgestellt.

2.1.2 Configurable Logic Block (CLB)

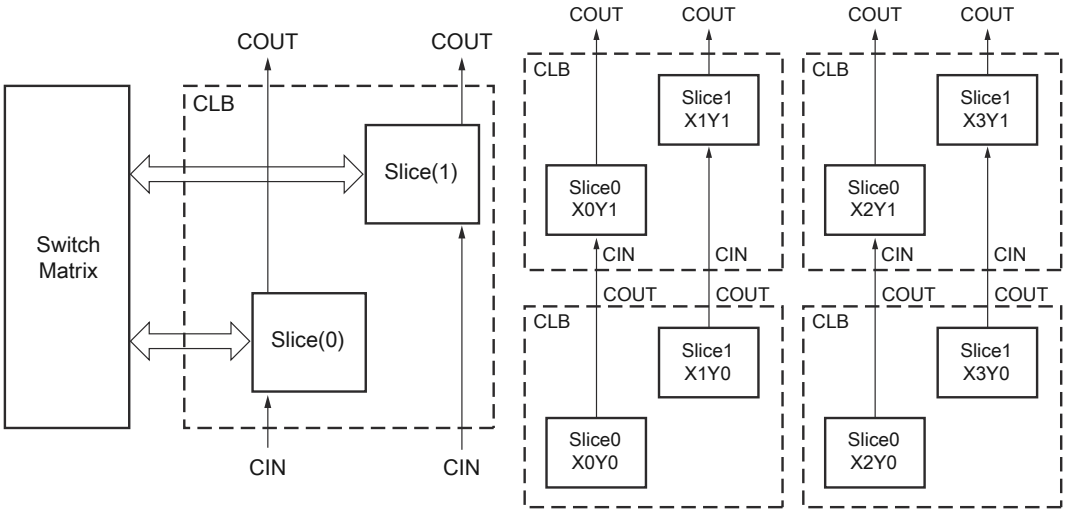

Abbildung 2.1: Struktur (links) und Anordnung (rechts) der CLBs eines FPGAs der Serie 7 [Xil14a]

Die konfigurierbaren Logikblöcke sind die Grundbausteine eines FPGAs, aus denen die logischen Schaltungen aufgebaut sind. Sie können individuell konfiguriert und wie auf der Abbildung 2.1 links über die programmierbare Schaltmatrix miteinander verbunden werden. Jede CLB enthält zwei Slices (SLICEL und SLICEM) zur Implementierung kombinatorischer und sequentieller (getakteter) Logik. Jede dieser Slices besteht wiederum aus vier Look-Up-Tabellen mit sechs Eingängen, acht Speicherelementen (Flip-Flops und Register), Multiplexer zur Verschachtlung der einzelnen Elemente und einer schnellen Carry-Logik für arithmetische Funktionen. Abbildung 2.1 rechts zeigt die matrixförmige Anordnung der CLBs. Dabei sind einzelne Slices in Spalten organisiert und werden unten links beginnend durchnummeriert. Der Aufbau der Slices wird am Beispiel von SLICEM in Abbildung 2.2 dargestellt.

Die CLBs implementieren die meiste Logik in einem FPGA. Die LUT in SLICEM können als Distributed-RAM oder Schieberegister konfiguriert werden. Zusätzlich zur LUTs enthält jede Slice eine eigene Carry-Logik zur schnellen Berechnung der arithmetischen Funktionen wie Addition und Subtraktion. Pro Slice gibt es einen Carry-Pfad, der vier Carry-Bits produzieren kann. Wie in der Abbildung 2.3 dargestellt ist können durch schnelle Carry-Signale CIN und COUT die benachbarten CLBs zum Aufbauen breiter Addition oder Subtraktion verschachtelt werden.

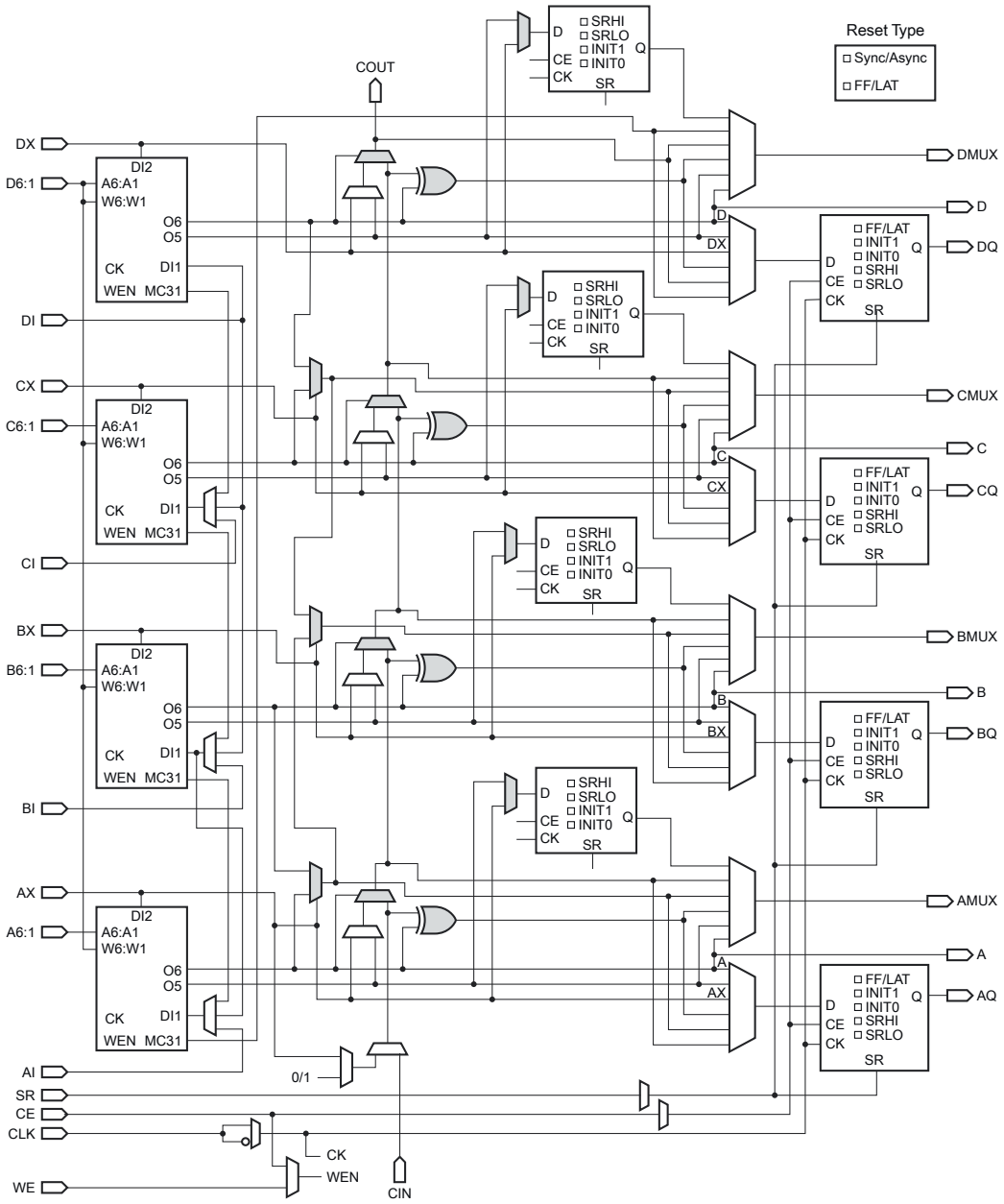

Abbildung 2.2: Aufbau einer SLICEM [Xil14a]

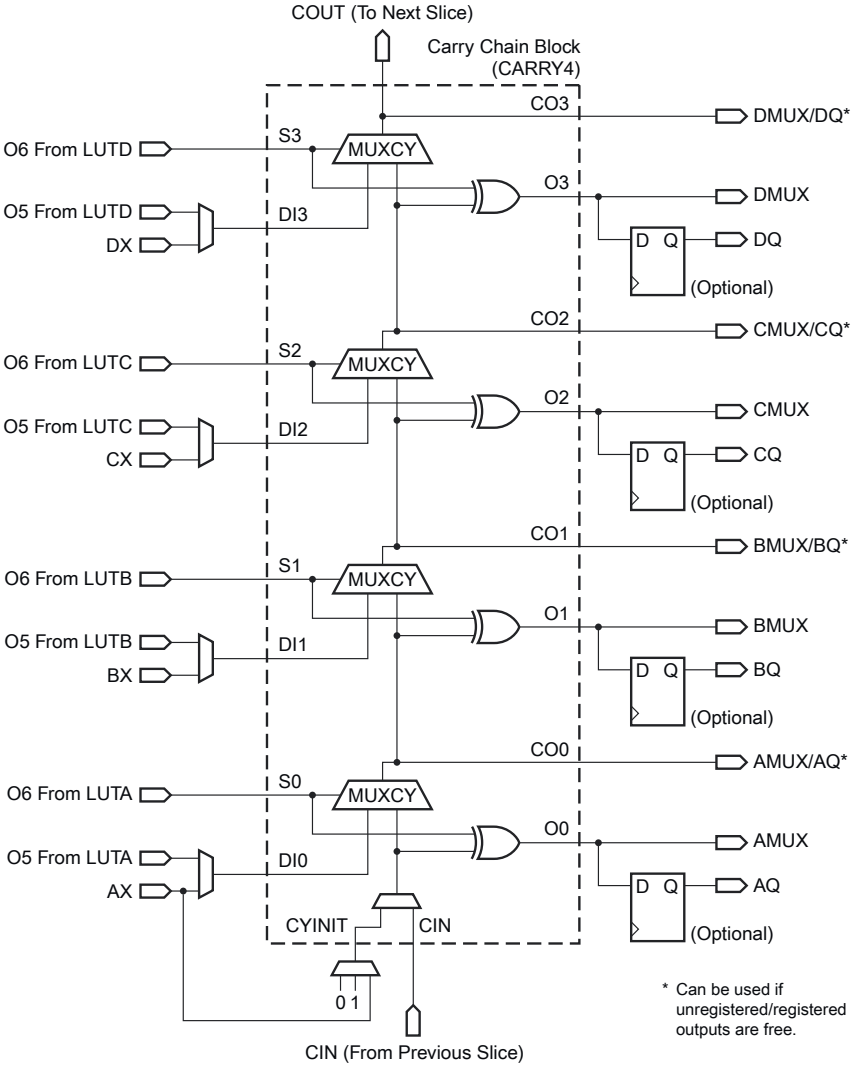

Abbildung 2.3: Fast Carry Logic Path in einem Slice [Xil14a]

2.1.3 Look-Up Table (LUT)

Look-Up Tables werden zur Implementierung der booleschen Funktionen in CLBs verwendet. In FPGAs der Serie 7 existieren in jeder Slice vier voneinander unabhängige LUTs (A, B, C und D) mit sechs Eingängen (A1 bis A6) und zwei Ausgängen (O5 und O6) [Xil15].

Ein LUT mit k Eingängen (LUTk) ist als eine beliebige Gatterfunktion mit k Variablen in Tabellenform konfigurierbar und besitzt 2^k Speicherplätze. In Slices existieren zusätzlich drei Multiplexer (`F7AMUX`, `F7BMUX` und `F8MUX`), durch die LUTs zur Realisierung komplexerer Logikfunktionen zusammengeschaltet werden. Beispielsweise wird eine logische Funktion mit 7 Variablen durch die Kombination der LUTs A und B mittels `F7AMUX` realisiert.

Außerdem werden LUTs in SLICEM zur Implementierung von RAM-Ressourcen, die als Distributed-RAM bezeichnet, verwendet.

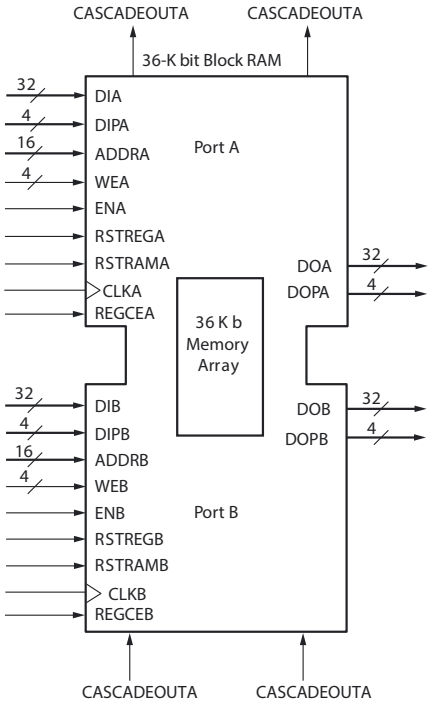

Abbildung 2.4: Dual-Port Block-RAM in einem FPGA der Serie 7 [Xil14b]

2.1.4 Block-RAM

Neben Logikblöcken existieren in den FPGAs der Serie 7 eine Reihe von Block-RAMs bis zur einer Größe von 36Kb. Dabei lassen sich jedes dieser Speicherelemente entweder als einzelne 36Kb RAM oder als zwei unabhängige 18Kb RAMs

konfigurieren. Block-RAMs bieten schnelle und flexible Speicherung von großen Datenmengen und werden auch als ROM- oder FIFO-Speicher konfiguriert. Der Zugriff auf die Speicherzellen erfolgt durch die Daten- und Adressports (PortA und PortB) wie in der Abbildung 2.4 dargestellt ist. Dabei können diese Blöcke noch weiter zerlegt werden, um verschiedene Datenwortbreiten zu definieren.

2.1.5 Input/Output Blocks (IOBs)

Input/Output-Blöcke befinden sich im Randbereich des FPGAs und stellen die Verbindung zwischen den internen Signalen und den externen Eingangs- und Ausgangssignalen. Zum Zwischenspeichern der Ein- und Ausgangssignale in der IOBs sind Flip-Flops vorhanden. Die Eigenschaften der IOBs wie IO-Standards, Treiberstärke und Kommunikationsrichtung sind programmierbar. Des weiteren sind IOBs als Input, Output oder bidirektional (Tri-State) konfigurierbar.

2.2 Vivado Design Suite

Die Vivado Design Suite ist die neue Entwicklungsumgebung der Firma Xilinx, die im Jahr 2012 auf dem Markt gebracht wurde. Die Vivado Design Suite wurde von Grund auf neu entwickelt, um die Design-Produktivität für die nächsten Generationen von All-Programmable-Bausteinen zu erhöhen [Xil12]. Dieser Abschnitt richtet sich an die Xilinx Vivado-Dokumentationen und stellt die wichtigen Funktionen des Vivado-Tools kurz vor, die für dieser Arbeit relevant waren.

2.2.1 Designablauf

Vivado bietet dem Anwender sowohl GUI- als auch Skript-basierten Designablauf, die nach Komplexität und Anforderung des zu entwickelnden Designs eingesetzt werden können. Vivado lässt sich vollständig mittels Tcl-Befehlen bedienen, die auch zu einem automatisierten Designablauf als Tcl-Skripte zusammengestellt werden. Dabei wird das zu entwickelnde Design im Hauptspeicher erstellt und die vollständige Verwaltung der Design-Daten erfolgt durch den Benutzer. Im Gegensatz dazu übernimmt Vivado die automatische Verwaltung des Designs und der Quelldateien in Projekt-basiertem Designablauf, der sowohl Skript-basiert als auch in GUI-Mode ausgeführt werden kann. In GUI-Mode stellt noch der Flow-Navigator zum Verwalten von einzelnen Design-Schritten wie Synthese und Implementierung zur Verfügung.

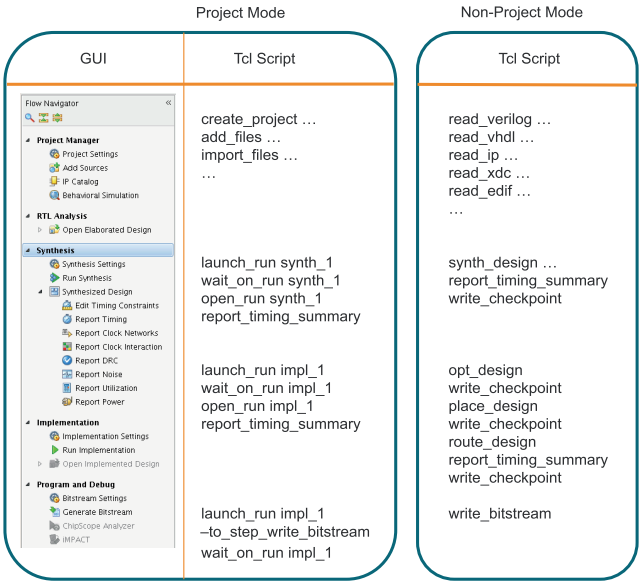

Abbildung 2.5: Tcl-Befehle des Designflow in Project-Mode und Non-Project-Mode [Xil14f]

2.2.2 Design-Checkpoints

In Vivado kann in jeder Entwicklungsphase der Zustand eines aktuellen Designs als so genannte Design-Checkpoints (DCP) gesichert werden. Sie dienen zum Speichern und Wiederherstellen der Designdatenbank mit allen enthaltenen Daten, wie Netzliste (edf), Constraints (xdc), Platzierung- und Routing-Information (xdef). Eine Checkpoint-Datei kann mittels den Tcl-Befehlen *write_checkpoint* und *read_checkpoint* geschrieben bzw. gelesen werden.

2.2.3 Design-Constraints

Constraints sind physikalische oder zeitliche Einschränkungen, die das Verhalten eines zu entwickelnden Designs spezifizieren. Xilinx Design Constraints (XDC) definieren Anforderungen an ein Design, die bei einem FPGA-Entwurfs berücksichtigt werden, um die Performance-Ziele zu erreichen. Beispielsweise können mittels Timing-Constraints Taktsignale definiert oder durch Physical-Constraints die zu verwendeten I/O-Pins auf dem FPGA festgelegt werden. Die Definition von Constraints erfolgt mittels Tcl-Befehlen in der XDC-Dateien, die ebenso mit den Tcl-Befehlen *read_xdc* und *write_xdc* gelesen bzw. geschrieben werden.

2.2.4 Design-Netlist

Eine Netzliste ist eine rein textuelle Beschreibung eines Designs. Darin sind alle Komponenten wie Zellen, Ports, Pins und Netze einer Schaltung vollständig enthalten. Dabei existiert in jeder Entwurfsphase eine Design-Netzliste wie z.B. synthetisierte oder implementierte Netzliste. Mittels Netzliste lassen sich mit einem Fremdtool entwickelten Design in Vivado importieren. Dabei ist EDIF das wichtigste Austauschformat für Netzlisten. Die aktuelle Netzliste kann mit *write_edif* als EDIF-Datei exportiert werden. Entsprechend kann eine Netzliste mittels *read_edif* gelesen werden. Ähnlich zu EDIF kann eine Netzliste in VHDL- oder Verilog-Format gespeichert werden. Die Elementen einer Netzliste sind in Abbildung 2.6 dargestellt.

2.2.5 Design-Objekte

Vivado basiert auf einem gemeinsam genutzten und skalierbaren Datenmodell. Dieses einzelne Datenmodell wird in jeder Stufe des Designablaufs benutzt und verschafft dem Anwender im Designprozess viel früher einen Einblick in entscheidende Design-Metriken (wie z.B. Timing, Leistungsbedarf, Ressourcenausnutzung und Verschaltungsengstellen) [Xil12]. Die Datenbasis eines Entwurfs wird dabei in einer Datenbank im Hauptspeicher verwaltet. Die Designdatenbank kann während des Designablaufes interaktiv mit Tcl-Befehlen analysiert, bearbeitet und deren aktueller Inhalt in jedem Design-Schritt als sogenannter Checkpoint zur späteren Wiederverwendung gespeichert werden.

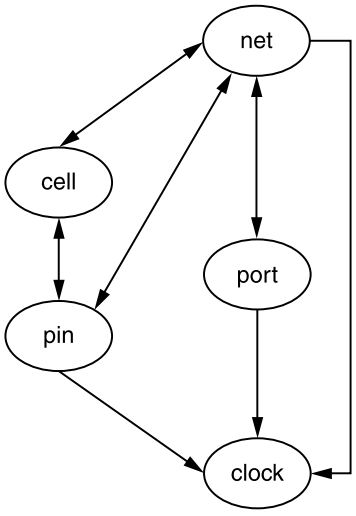

Abbildung 2.6: Fist-Class Design-Objekte in Vivado

Die Vivado-Datenstruktur besteht aus modularen Tcl-Objekten, die das Design, die logische Netzliste oder das Ziel-FPGA repräsentieren. Jedes einzel-

ne Objekt besitzt dabei eine Menge von beschreibbaren oder nur lesbaren Properties (Eigenschaften), die mit *get_property* oder *set_property* gelesen bzw. gesetzt werden. Die Abbildung 2.6 zeigt die grundlegenden und wichtigsten Vivado Design-Objekte und ihren Beziehungen.

Für jedes Design-Objekt existiert eine entsprechende `get`-Methode, mit der es angesprochen werden kann. Dabei ermittelt die jeweilige *get_**-Methode mit der Option *-of_objects* alle Objekte, die mit einem spezialisierten Objekt in Beziehung stehen. Des weiteren existieren Tcl-Methoden wie *create_** und *remove_** für jeweilige Design-Objekte. Sowohl einzelne Tcl-Befehle als auch jeder Design-Prozess operiert auf der Design-Netzliste und modifiziert sie oder erstellt eine neue Netzliste.

2.3 Haftfehlermodell

Fehlermodelle beschreiben die möglichen auftretenden Fehler in einem System und deren Auswirkungen auf die elektrische und logische Systemeigenschaften. Sie werden zur Bewertung der Auswirkungen von unterschiedlichen Fehlern eingesetzt. Als Fehler wird das Fehlverhalten einer Schaltung (bzw. eines Systems) bezeichnet, bei dem die Schaltung von ihrem spezifizierten Verhalten abweicht.

Das Haftfehlermodell ist bei der Fehlerinjektion das am einfachsten und am meisten eingesetzte Fehlermodell, welches auch die Grundlage für diese Arbeit bildet. Bei einem Haftfehler (Stuck-at-Fehler) wird angenommen, dass ein internes Signal einer digitalen Schaltung ständig einen logischen Wert 0 oder 1 annimmt, so dass keine Signaländerung möglich ist.

Um die Auswirkungen der in einem geeigneten Fehlermodell spezifizierten Fehler zu untersuchen, werden diese durch Modifikation in einer zu entwickelnde Schaltung eingebracht. Danach wird der Schaltung mit einem geeigneten Testsatz simuliert oder auf einem FPGA emuliert, um den in der Schaltung vorliegenden und in einem Fehlermodell spezifizierten Fehler zu erkennen. Wenn sich das Ergebnis der Simulation oder der Emulation von dem zuvor ermittelten korrekten Ausgangswerten unterscheidet, so ist der eingefügte Fehler Mithilfe des verwendeten Testvektoren erkennbar. Wird dieser Vorgang für jeden einzelnen Fehler und mit einer bestimmten Menge von Testvektoren durchgeführt, so lassen sich alle Testvektoren bestimmen, bei denen dieser Fehler erkannt wird [Woj88].

3 Stand der Technik

Zur Analyse der Zuverlässigkeit und der erreichten Fehlertoleranz existieren bereits viele Ansätze und standardisierte Fehlerinjektiontechniken. Eine kurze Einführung in der Literatur [BRF+96, JDR09, DN11, CHD99] für FPGA-basierte Verfahren und Werkzeuge findet man in [Pet12]. Darüber hinaus wird in [Pet12] ein Verfahren beschrieben, auf welches die im Rahmen dieser Arbeit implementierte Methode basiert.

Beim [Pet12] beschriebenen Verfahren werden Defekte in den Modulen und Ports der Netzliste auf RT-Ebene selektiert und diese direkt auf das Design übertragen. Dieses Verfahren basiert auf der Serial Fault Emulation [BRF+96], welches eine Schaltung zur Fehlerinjektion anpasst und diese auf einem FPGA zur Ausführung bringt. Dabei werden bei diesem Ansatz für jeden zu injizierenden Fehler alle Kompilierungs- und Syntheseprozesse durchgelaufen, um die Konfiguration des FPGA-Designs zu erhalten. Da die erneute Kompilierung und Synthese für jeden Defekt erhebliche Zeit in Anspruch nehmen, wurde bei diesem Verfahren eine statische Abbildung entwickelt, die die Serial Fault Emulation um die Überführung von Elementen der Netzliste auf das FPGA-Design erweitert. Somit wird unabhängig von der Anzahl der Fehlerinjektionen eine einmalige Durchführung der Kompilierung und Synthesen erreicht, welche bei der Serial Fault Emulation für jeden zu injizierenden Fehler erneut durchgelaufen werden. In Abbildung 3.1 ist der Ablauf der beiden Ansätze mit den zugehörigen einzelnen Phasen dargestellt.

In [Pet12] ist eine XDL und Rapidsmith basierte Java-Bibliothek implementiert, welche zum Einlesen sowie zum Anpassen des FPGA-Designs bei einer Fehlerinjektion auf beliebige Bauteile verwendet wird. In dieser Implementierung werden Ports von Bauteilen, wie z.B. der Dateneingang eines Flip-Flops, referenziert und dort permanente Defekte der Fehlermodelle Stuck-at-0 und Stuck-at-1 selektiert und auf das Design übertragen.

Die in [Pet12]mit mehreren Testschaltungen (ISCAS-Benchmarks, MIPS-Prozessor, CRC-Hardwarebeschleuniger) durchgeführten Tests zeigten einen Geschwindigkeitszuwachs gegenüber der generischen Serial Fault Emulation um den Faktor 2 bis 44.

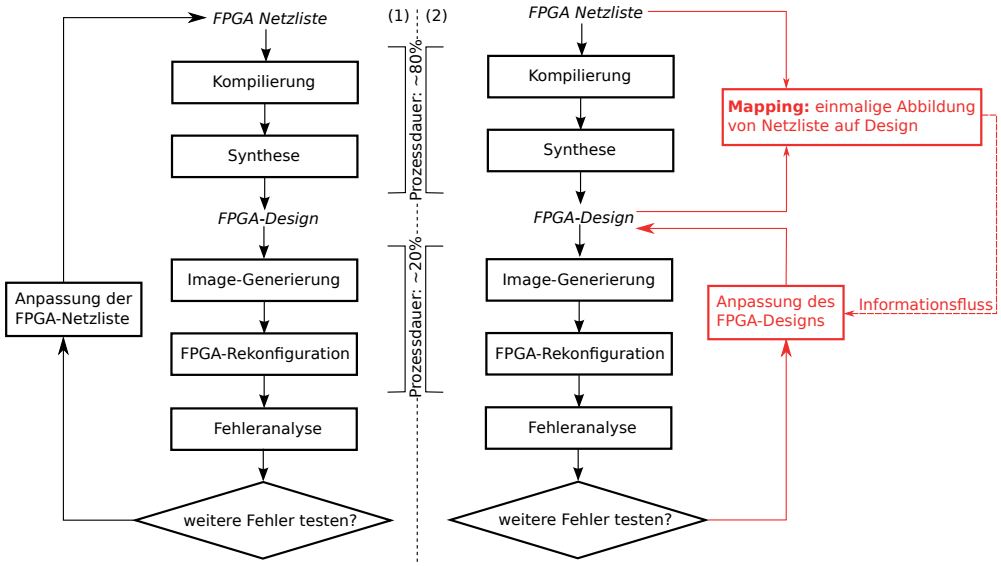

Abbildung 3.1: Vergleich der Fehlerinjektion bei der Serial Fault Emulation (1) und dem in [Pet12] entwickelten Verfahren (2)

4 Implementierung

Dieses Kapitel beschreibt die Implementierung einer Tcl-basierten Fehlerinjektion in der Vivado-Designumgebung. Dabei werden permanente (Stuck-at) Fehler gezielt in einer zu entwickelnden Schaltung injiziert und diese per Simulation oder Emulation analysiert. Ein wesentlicher Vorteil der Fehlerinjektion in der Vivado-Designumgebung ist die Möglichkeit der Simulation. Zusätzlich zur Fehleremulation kann die Fehlersimulation nach jedem Design-Schritt durchgeführt werden. Sie kann beispielsweise nach der Generierung der synthetisierten Netzliste ohne Zeitverhalten ausgeführt werden. Damit kann bezüglich der Komplexität der entwickelnden Schaltung gezielt einer dieser Methoden benutzt werden.

Beim implementierten Verfahren erfolgt die Zuweisung der Stuck-at-Fehler auf der RT-Ebene, d.h. in der aus der Verhaltensbeschreibung generierten generischen RTL-Netzliste. Die Fehlerinjektion wird dagegen im synthetisierten (Post-Synthesis) oder implementierten (Post-Implementation) Design an Primitiven (sogenannte Leaf-Zellen) oder an Makrozellen (z.B. Block-RAM) durchgeführt. Dies beschleunigt den Ablauf der Fehlerinjektion zu jedem einzufügenden Fehler erheblich, weil die Synthese und Implementierung nicht für jeden Durchgang durchgeführt werden müssen. Dieser Vorgang wird anhand des Mapping-Prozesses (kurz: Mapping) realisiert.

In Abhängigkeit von Mapping und Modifikation der zu testenden Schaltung sind folgende Varianten der Fehlerinjektion zu unterscheiden:

MBF-Injektion (Mapping-basierte Fehlerinjektion) Bei dieser Variante werden Stuck-at-Fehler auf Basis der RTL-Selektion an Input-Objekte (Ports und Pins) zugewiesen. Anschließend werden die Komponenten des FPGA-Designs mit Hilfe vom Mapping zur Injektion von Fehlern referenziert und darauffolgend modifiziert.

DBF-Injektion[1] (Design-basierte Fehlerinjektion) Bei dieser Variante werden Stuck-at-Fehler ohne Mapping direkt in das FPGA-Design eingefügt. Dabei werden die Komponenten des FPGA-Designs zur Fehlerinjektion selektiert und anschließend abhängig vom ausgewählten Fehler modifiziert.

Bei einigen Schaltungen kann nicht ein hundertprozentiges Mapping durchgeführt werden. Durch die Kombination beider Varianten kann jedoch eine vollständige Fehlerüberdeckung erreicht werden.

[1] Diese Bezeichnung wurde aus [Pet12] übernommen.

4.1 Terminologie

Um die Elemente einer zu entwickelnden Schaltung und die Phasen der Fehlerinjektion deutlich zu kennzeichnen, finden folgende Begriffe in dieser Arbeit häufig Verwendung:

Design Ein Design enthält alle Komponenten einer zu entwickelnden Schaltung wie Gatter, Register, Netze, etc. Ein Design ist durch abstrakte Modelle beschrieben und kann daher mit EDA-Werkzeugen simuliert, verifiziert und als Schaltplan dargestellt werden [Hop99]. In jeder Phase des Schaltungentwurfs existiert ein Design in Form einer Netzliste mit angewendeten zeitlichen und physikalischen Randbedingungen (Constraints).

Das Design wird in unterschiedlichen Entwicklungsphasen in Vivado wie folgt benannt:

RTL-Design Enthält Schaltungselemente in Form einer generischen Netzliste auf der RTL-Ebene, die durch die Analyse und Kompilierung des HDL-Code erzeugt wird. Das RTL-Design (bzw. RTL-Netzliste) Ist die Grundlage für die Selektion und Fehlerzuweisung der Objekte.

Post-Synthesis Design Ist ein durch Logiksynthese der RTL-Netzliste generiertes Design. Dabei werden die Elemente der generischen Netzliste auf die Komponenten des Ziel-FPGAs, unter Berücksichtigung der zeitlichen Randbedingungen, abgebildet.

Post-Implementation Design Vollständig platziertes und geroutetes Design mit zeitlichen und physikalischen Randbedingungen.

Modul Module ermöglichen die Strukturierung eines Designs. Ein Modul wird in Verilog als `module` deklariert. In VHDL ist ein Modul eine `Entity` und wird in einer aufrufenden Architektur als ein `component` deklariert. Module können durch Instanziierung mehrerer Submodule hierarchisch aufgebaut sein. Somit wird die Hierarchie eines Designs definiert. Jedes Design besteht aus mindestens einem Modul (sogenannte Top-Level-Modul).

Leaf-Zellen Sind auf dem Ziel-FPGA verfügbare Xilinx-Komponenten. Diese Elemente werden durch UNISIM-Bibliothek zu Verfügung gestellt und können direkt im HDL-Code instanziiert werden. Die UNISIM-Bibliothek enthält Design-Elemente wie LUTs, Flip-Flops, Buffer-Instanzen usw.

Makro-Zellen Sind Funktionsblöcke, die aus mehreren Leaf-Zellen bestehen. Diese Instanzen werden durch UNIMACRO-Bibliothek bereitgestellt. Typische Xilinx-Makros sind Block-RAMs, Addierer, Multiplier usw.

Port Ein Port definiert die Schnittstelle des Top-Level Moduls einer Schaltung. Je nach Datenflussrichtung kann es sich bei Ports um einen Eingang (`IN`), Ausgang (`OUT`) oder einen bidirektionalen Port (`INOUT`) handeln. Ein Port kann skalar sein, d.h. aus einer einzigen Verbindung bestehen oder als Busport mit fester Busbreite mehrere Signale zusammenfassen.

Pin Sind Verbindungspunkte der Designelemente wie Leaf-Zellen oder Module, an denen Netze angeschlossen sind. Damit werden diese Design-Elemente durch Netze miteinander verbunden. Bei der Implementierung werden Pins in folgenden Unterklassen unterteilt, die beim Mapping unterschiedlich behandelt werden:

Modul-Pin Bezeichnet Ein- und Ausgabepin der Module.

Leaf-Pin Bezeichnet Ein- und Ausgabepin der Leaf-Zellen.

Abbildung 4.1 stellt oben benannte Design-Elemente dar, die für die Fehlerinjektion relevant sind.

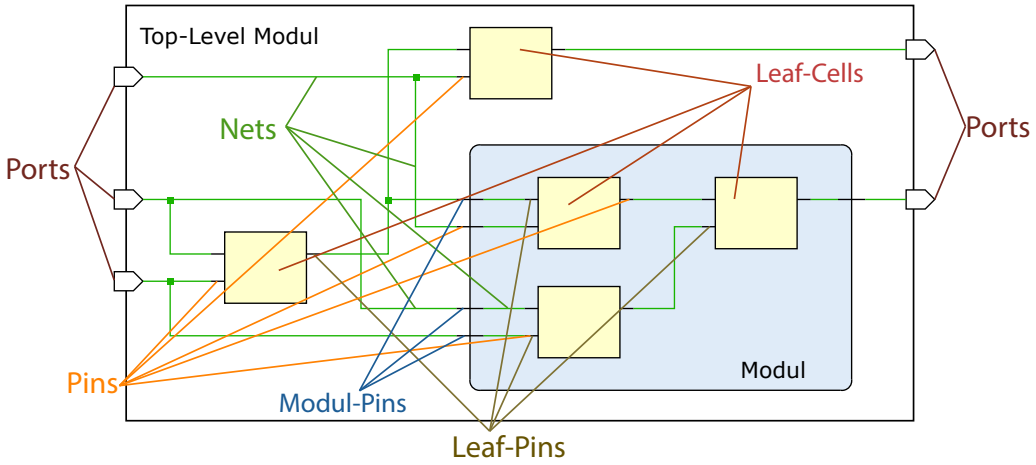

Abbildung 4.1: Design-Objekte einer Schaltung

4.2 Ablauf der Fehlerinjektion in Vivado

Dieser Abschnitt verschafft einen kurzen Überblick zum Ablauf der Fehlerinjektion in der Vivado-Designumgebung. Im nächsten Kapitel wird dieser Ablauf mit den zugrundeliegenden Datenstrukturen detailliert beschrieben.

Bei dem implementierten Ansatz wird das Injizieren von Defekten nach der Synthese oder Implementierung durchgeführt. Dabei wird der Synthese- und Implementierungsschritt, welcher die größte Zeit des Ablaufes der Fehlerinjektion in Anspruch nimmt, nur einmalig ausgeführt. Dieser Vorgang wird

mittels `Netlist` und `Checkpoint` erreicht, indem das synthetisierte Design als `Netlist`-Datei und das implementierte Design als `Checkpoint`-Datei gespeichert wird. Mittels dieser Dateien wird der zuvor gespeicherte Zustand von synthetisiertem bzw. implementiertem Design in Vivado wiederhergestellt.

Die Abbildung 4.2 stellt den gesamten Ablauf der Fehlerinjektion in Vivado und die entsprechenden Tcl-Befehle in den jeweiligen Schritten dar.

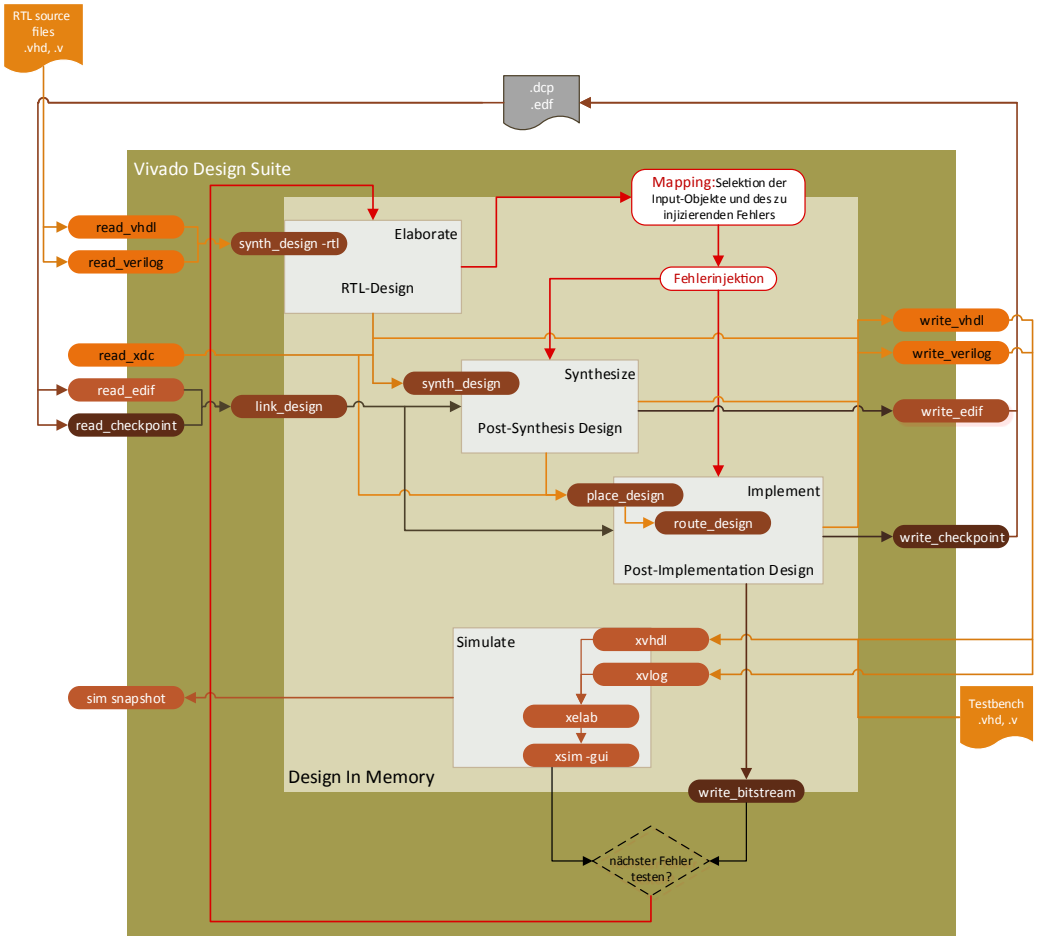

Abbildung 4.2: Ablauf der Fehlerinjektion in Vivado

Die `Netlist`-Datei wird zum Speichern des synthetisierten Designs mit allen enthaltenen Instanzen und deren Verschachtelung verwendet, um später im Ablauf der Fehlerinjektion das Post-Synthese Design in den Hauptspeicher zu laden. Die `Checkpoint`-Datei wird zum Sichern des aktuellen Zustands des platzierten und gerouteten Designs verwendet, um ähnlich wie das synthetisierte Design das Post-Implementation Design zu laden. Im Gegensatz zu einer `Netlist`-Datei beinhaltet `Checkpoint`-Datei zusätzlich die Constraints und die Platzierung- und Routing-Informationen eines Designs. Basierend auf

diesen Dateien ist ein einmaliges Synthetisieren und Implementieren des Designs möglich, welches beim implementierten Ansatz vorausgesetzt ist.

Das implementierte Verfahren setzt außerdem den Erhalt der hierarchischen Struktur der entwickelnden Schaltung voraus, mit der ein hoher Fehlerüberdeckungsgrad erreicht wird. Um die Flattening des Designs bei der Synthese zu verhindern, muss die Syntheseoption `flatten_hierarchy` auf `none` gesetzt werden. Damit entspricht die synthetisierte bzw. implementierte Netzliste der hierarchischen Struktur des RTL-Designs. Wenn die synthetisierte Netzliste eine flache Struktur hat, d.h. keine Modulgrenzen beibehält, so werden die Selektionen an Modul-Pins durch das Mapping nicht im FPGA-Design referenzierbar. In diesem Fall funktioniert das Mapping nur für Ports und Leaf-Pins. Möchte man hingegen eine DBF-Injektion durchführen, so spielt die Designhierarchie keine Rolle, weil bei der DBF-Injektion kein Mapping erforderlich ist und die Referenzierung direkt an den Leaf-Pins vorgenommen werden kann.

Die andere Möglichkeit um die Designhierarchie im FPGA-Design zu erhalten, ist das Synthese-Attribut `KEEP_HIERARCHY`. Dieses Attribut verhindert die Optimierungen entlang der Hierarchiestufen. `KEEP_HIERARCHY`-Attribut kann wie folgt auf Module oder Instanzen gesetzt werden [Xil14i]:

```
(* keep_hierarchy = "yes" *) module adder_2bit (in1,...);
(* keep_hierarchy = "yes" *) MyModule M1 (.in1(in1),...);
```

Liegt das synthetisierte bzw. implementierte Design vor, so können die implementierten Tcl-Befehle zur Fehlerinjektion ausgeführt werden. Dabei wird im ersten Schritt ein Stuck-at-Fehler an Port- oder Pin-Objekte zugewiesen, indem entsprechende Injektoren (siehe Abschnitt 5.2) erzeugt werden.

Im zweiten Schritt erfolgt das Referenzieren des zuvor ausgewählten Objektes in das Post-Synthesis oder Post-Implementation Design zur Fehlerinjektion mittels erzeugter Injektoren. Anschließend werden die Änderungen an den Komponenten des FPGA-Designs in Abhängigkeit von ausgewählten Stuck-at-Fehler vorgenommen. Im Folgenden werden nun diese Teilschritte detailliert beschrieben.

4.3 Mapping

Mapping wird in dieser Arbeit als Verfahren bezeichnet, bei dem Ports oder Pins im synthetisierten oder implementierten Design auf Basis der RTL-Selektion zur Injektion von Fehlern referenziert werden. Mapping ist die Basis der implementierten Methode zur Fehlerinjektion. Daher hängt eine vollständige Fehlerinjektion auf Basis der Netzliste vom Mapping-Prozess ab. Andererseits hängt das Mapping, das im Folgenden beschrieben wird, von vorhandenen Elementen der zu testenden Schaltung ab.

Die Abbildung 4.3 stellt eine schematische Darstellung des Mappings dar. Der erste Schritt des Mappings ist die Erzeugung der Injektoren durch die Selektion von Port- oder Pin-Objekte in der RTL-Netzliste. Dabei wird jedes Objekt entsprechend seines Typs unterschiedlich behandelt. Nach Zuweisung

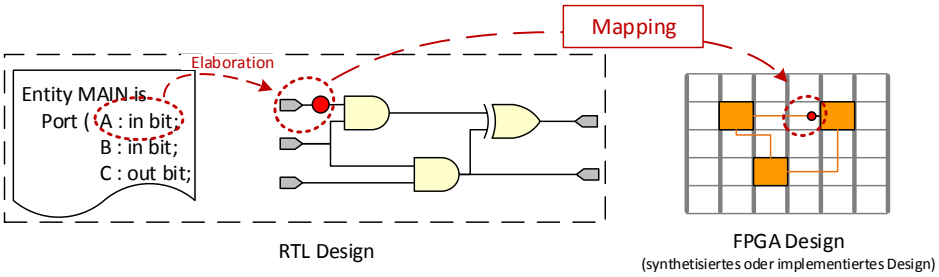

Abbildung 4.3: Mapping der RTL-Netzliste auf das FPGA-Design

der Defekte an Input-Ports oder -Pins werden im zweiten Schritt des Mappings diese Elemente im FPGA-Design gesucht und lokalisiert, um sie für die DBF-Injektion zu referenzieren. Während der Schaltungssynthese von Vivado werden die Namen einiger Elemente der synthetisierten Netzliste sich von den Namen der Elemente im RTL-Design unterscheiden. Darüber hinaus werden einige Elemente wie Register und Instanzen der arithmetischen Operationen im RTL-Design durch Optimierung entfernt oder auf eine FPGA-Ressource (wie DSP-Instanz) abgebildet. Daher ist eine hundertprozentige Abdeckung des Mappings nicht bei allen Schaltungen möglich. Durch die Kombination der NBF- und DBF-Injektion wird eine vollständige Fehlerüberdeckung erreicht, wenn das Mapping für einige Elemente der Netzliste nicht erfolgreich ist, diese jedoch im FPGA-Design vorhanden sind. Auf die entsprechenden Synthese- bzw. Implementationsoptionen wird in den folgenden Abschnitten eingegangen.

Wie schon oben erwähnt, müssen die selektierten Objekte zur Fehlerinjektion im FPGA-Design auffindbar sein. Jede Komponente im Design besitzt Properties, die die elektrischen und logischen Eigenschaften einer Komponente definieren. Eine dieser Properties ist der Name des Design-Objektes. Jedes Design-Objekt in Vivado besitzt einen eindeutigen Namen, mit dem die Objekte über die `get_*`-Befehle einen Zugriff erhalten.

Das Finden der zu injizierenden Objekte wie Ports, Modul- oder Leaf-Pins in FPGA-Design, erfolgt durch namensspezifische Suche im Design.

Die Suche nach einem bestimmten Objekt bei Default findet über seinen Namen im aktuellen Level statt. Durch das Einschalten der Option `-hierarchical` bei `get_*`-Methoden werden Objekte beginnend vom Top-Level in jeder Hierarchieebene gesucht. Das Durchsuchen der Design-Hierarchie

für bestimmte Objekte kann zusätzlich mit Optionen wie `-filter` und `-regexp` spezialisiert werden [Xil14j]. Damit sind Objekte trotz Umbenennungen bei der Vivado-Synthese im FPGA-Design referenzierbar.

Während mit `get_*`-Methoden die Design-Objekte angesprochen werden, kann mittels `get_property` die Eigenschaften eines Objekts abgefragt werden. Die Richtung der Ports (bzw. Pins) kann z.B. mittels

```
get_property DIRECTION [get_ports <port_name>]
```

abgerufen werden. Ob es sich bei Pins um einen Modul- oder einen Leaf-Pin handelt, kann wie folgt mit der `IS_LEAF`-Property festgestellt werden:

```
get_property IS_LEAF [get_pins <pin_name>]
```

Bei der Zuweisung der Defekte an Port- oder Pin-Objekte werden Injektoren mit unterschiedlichen Informationen erzeugt. Dabei werden alle selektierten Objekte nach ihrem Typ (Port oder Pin) mittels *get_ports $allSelObjects* und *get_pins $allSelObjects* gefiltert. Die selektierten Pins werden weiterhin nach ihrem Typ (Modul- oder Leaf-Pins) mittels *get_pins $allSelPins -filter {IS_LEAF}* in zwei Gruppen eingeteilt. Alle diese werden dann wie unten beschrieben unterschiedlich behandelt.

Des weiteren werden Zell-Objekte, zwischen welchen bei der Selektion eines Leaf-Pins oder bei der Anpassung des FPGA-Designs unterschieden werden muss, durch die Properties `PRIMITIVE_GROUP und PRIMITIVE_TYPE` angesprochen.

4.3.1 Injektoren für Ports

Wie oben erwähnt, wird für jede Selektion der Elemente in der Netzliste unabhängig von deren Typen eine Instanz des Injektors erzeugt. Bei der Zuweisung eines Injektors an einem Port wird der Name des Ports als `-selInput` gespeichert. Des Weiteren wird `-isPort` auf '1' gesetzt, um sich die Selektion als Eingangsport zu merken. Entsprechend des am Port zuzuweisenden Stuck-at-Fehlers, wird das Feld `-faultType` auf `true` oder auf `false` gesetzt.

Ist ein Skalar Port selektiert, so wird nur ein Injektor erzeugt. Im Gegensatz dazu wird, wenn ein Port vom Datentyp Vektor ausgewählt ist, diesem die höchste Bitposition eines Defekts zugewiesen. Schaltet man die Option `-inject_multiple` bei der Zuweisung von Defekten durch die Methode `inject_faults` an Ports oder Pins, so wird jedes einzelne Bit dieser Objekte den Injektoren gleichen Typs zugewiesen.

Das Referenzieren der zu injizierenden Leaf-Pins, die durch selektierte Ports getrieben werden, erfolgt über die Methode *mapping*. Dabei werden alle Input-Pins der Leaf-Zellen für Port-Injektoren ermittelt und das Feld `-leafPins` der jeweiligen Injektoren durch diese ermittelten Pins ersetzt.

Die Leaf-Pins, die mit den selektierten Input-Ports verbunden sind, werden mit der folgenden Anfrage ermittelt:

```
set topNet [get_nets -top -seg -of [get_ports $selPort]]
lappend allConnLeafPins [get_pins -leaf -of [get_nets
   $topNet]]
```

Listing 4.1: Ermitteln aller Leaf-Pins, die von einem selektierten Port referenziert werden

Während der Synthese fügt Vivado Buffer-Instanzen (I/O-Buffer) zum Design ein, um die Eingangs- bzw. Ausgangssignale zu puffern. Das Ermitteln der Leaf-Pins, wie in Listing 4.1 zu sehen ist, gibt die Input-Pins der eingefügten I/O-Buffer zurück. Daher muss die Anfrage für Leaf-Pins im FPGA-Design noch weiter angepasst werden. Dies geschieht durch Ignorieren der Buffer-Instanzen, indem das ermittelte Netz `topNet` durch das an Output-Pin (bzw. Input-Pins beim Selektion eines Ports vom Typ `INOUT`) angeschlossenen Netz-Segment ersetzt wird. Dies erfolgt durch die rekursive Methode *skipping_all_buffer*.

Anschließend wird die Anzahl der gefundenen Leaf-Pins mit der Anzahl der Leaf-Pins, die in der RTL-Netzliste durch selektierte Ports angesprochen sind zum Feedback verglichen. Damit kann man feststellen, ob alle Leaf-Pins im FPGA-Design gefunden wurden.

Nach der Aktualisierung der Injektoren (d.h. Finden der selektierten Leaf-Pins und Referenzieren deren Parent-Zellen zur Fehlerinjektion) übernimmt die Methode *modify_netlist* das Ändern des FPGA-Designs in Abhängigkeit vom selektierten Fehler.

4.3.2 Injektoren für Modul-Pins

Bei Pins gibt das Attribut `IS_LEAF` an, ob es sich um die Ein- und Ausgabepins logischer Primitive (Gatter, Flip-Flops, etc.) oder eines hierarchischen Moduls handelt. Es wird, zusätzlich zu den Injektoren für Ports, bei Modul-Pins das Attribut `-isLeaf` auf `false` gesetzt. Somit kann man in einer *if*-Anweisung leicht identifizieren, ob es sich um einen Leaf-Pin oder Modul-Pin handelt, da bei der Suche nach den ausgewählten Pins unterschiedliche Informationen in Betracht gezogen werden.

Ein Pin kann wie Ports aus einer einzelnen Verbindung (vom Datentyp `bit`) oder als ein Bus aus mehreren Signalen (vom Datentyp `bit_vector`) bestehen. Wie bei Ports können Pins einzeln oder im Falle eines Bus gleichzeitig zur Injektion ausgewählt werden. Wenn ein bestimmtes `bit` eines Pins (bzw. Ports) vom Typ Bus injiziert wird, so kann diese mit der Option `-bus_bit` an der Methode *inject_fault* angegeben werden. Die Abbildung 4.4 veranschaulicht das Ermitteln der Leaf-Pins bei der Erzeugung von Injektoren für einen Modul-Pin. Falls man einen Defekt an den Pin `in1` des Moduls `M1` zuweisen will,

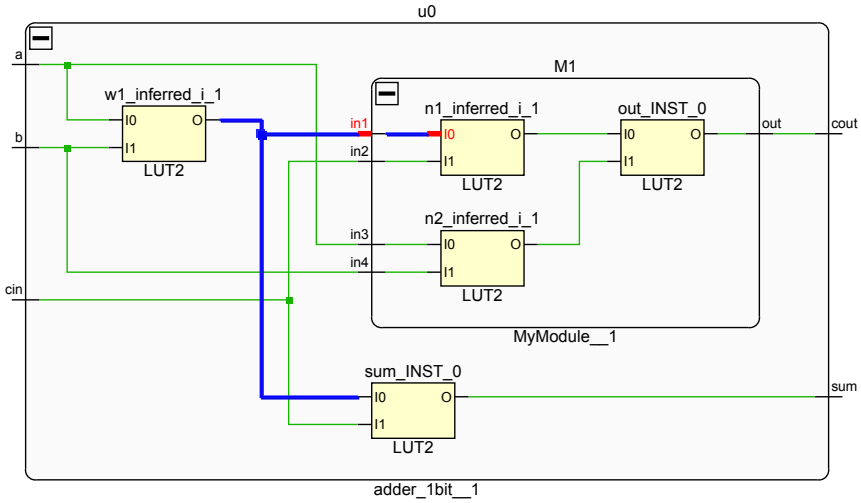

Abbildung 4.4: Beispiel für die Selektion eines Modul-Pins zur Fehlerzuweisung

so wird eine Instanz des Injektors namens `::ares::faults::u0_M1_in1` mit den Informationen `-selInput u0/M1/in1`, `-isport 0` und `-isLeaf 0` erzeugt. Um die Leaf-Pins zu identifizieren, die durch die ausgewählten Modul-Pins betrieben werden, muss zunächst das Netz, das an dem ausgewählten Pin `in1` angeschlossen ist, selektiert werden. Dies erfolgt durch die folgende Abfrage:

```
set net [get_nets -of [get_pins u0/M1/in1]]
```

Die Leaf-Pins, die an diesem Netz verbunden sind, werden mit folgender Abfrage angesprochen:

```
set allConnLeafPins [get_pins -leaf -of $net]
```

Dabei gibt diese Abfrage nicht nur den gesuchten Pin `u0/M1/n1_inferred_i_1/I0` innerhalb der Modulgrenze zurück, sondern auch die Pins `u0/w1_inferred_i_1/O` und `u0/sum_INST_0/I0`. Bei komplexeren Schaltungen ist ein Netz mit mehreren Pins auf unterschiedlichen Hierarchieebenen verbunden. Somit werden alle Pins mit der obigen Abfrage zurückgegeben.

Um den Leaf-Pin `u0/M1/n1_inferred_i_1/I0`, der innerhalb der Modulgrenze `u0/M1` liegt, zu referenzieren, muss noch die Abfrage wie folgt verfeinert werden:

```
set parentCell [get_property PARENT_CELL [get_pins u0/M1/
   in1]]
set allConnLeafPins [get_pins -leaf -of $net -filter "
   DIRECTION==IN && NAME =~ \$parentCell/*"]
```

Wenn durch die Logikoptimierung keine Instanzen der Leaf-Zellen zusammengeführt bzw. entfernt werden, so erfolgt ein vollständiges Mapping bei der Selektion von Modul-Pins.

4.3.3 Injektoren für Leaf-Pins

Da die Vivado-Synthese die Elemente der RTL-Netzliste auf unterschiedliche Primitiven der Zielplattform abbildet und sie in einigen Fällen neu benennt, ist das Finden der zu referenzierenden Objekte im FPGA-Design mit einfachen Tcl-Befehlen nicht möglich. Für die Pins unterschiedlicher Primitiven wurde ein Verfahren entwickelt, welches auf Beobachtungen mehrerer Designs und Informationen aus der Xilinx Vivado Dokumentationen basiert. Die Suche der zu injizierenden Objekte im FPGA-Design wurde Mithilfe der bei Tcl-Befehlen integrierten regulären Ausdrücken realisiert. Dabei wird der Name des Parent-Cell des selektierten Pins in einen regulären Ausdruck überführt. Im FPGA-Design werden dann Design-Objekte ermittelt, deren Namen der durch die RTL-Selektion definierten Textmuster entsprechen. Werden mehrere Objekte ermittelt, deren Namen den gesuchten Textmuster entsprechen, so werden diese nach weiteren Informationen, welche durch weitere Attribute der Injektoren bereitgestellt werden, aussortiert.

Bei direkter Selektion der Pins von Leaf-Zellen wurde kein hundertprozentiges Mapping erreicht. Sowohl der Grund eines unvollständigen Mappings dieser Objekte als auch das Verfahren selbst wird unten am Beispiel einiger Design-Objekte dargestellt. Da eine große Anzahl unterschiedlicher Design-Objekte existieren und das Mapping für diese Objekte ähnlich funktioniert, werden hier nur Register, logische Gatter und Multiplexer repräsentiert.

4.3.3.1 UNISIM-Instanzen und Register

Die in der generischen RTL-Netzliste vorhandenen Elemente werden durch das Synthese-Tool von Vivado auf die Komponenten des Ziel-FPGAs, die durch Xilinx UNISIM-Bibliothek bereitgestellt werden, abgebildet. Diese Komponenten können direkt im HDL-Code instanziiert werden. Die Namen der UNISIM-Module sowie der Register (bzw. Latches) werden während der Synthese nicht geändert. Bei der Lokalisierung werden die Namen dieser Objekte mit Hilfe der Injektoren unverändert an die Methode *mapping* weitergereicht.

Da der Registersatz in RTL-Schematic überlappt angezeigt wird und die Selektion eines bestimmten Registers in der GUI umständlich ist, muss bei Multi-Bit-Registern der Pin der selektierten Register mit der Option `-bus_bit` angegeben werden.

4.3.3.2 Logische Gatter

Die logischen Gatter wie AND, OR, XOR, etc. werden durch das Synthese-Tool auf LUT-Instanzen des Ziel-FPGAs abgebildet. Dabei werden diese Komponenten unterschiedlich umbenannt. Im Allgemeinen werden diese bei der Kompilierung des HDL-Code nach ihrem Output-Signal benannt. Des weiteren wird der Name auf ein Literal `i` und auf einen numerischen Index, wie in Abbildung 4.5 zu sehen ist, erweitert. In der synthetisierten Netzliste werden dann die Namen der logischen Gatter auf das Teilwort `INST` oder `inst`, durch einen Unterstrich getrennt, ergänzt. Wenn das Gatter mit einem Output-Port verbunden ist, wird noch das Wort `OBUF` und ein Index hinzugefügt. Die Namen der selektierten logischen Gatter werden auf einen regulären Ausdruck abgebildet, der alle diese Umbenennungen abdeckt. Der Ausdruck wird beim Mapping als Eingabe mit der Option `-regexp` für das Suchen eingegeben. Der Name des Pins `sum_i/I0` wird beispielsweise in dem Textmuster `sum(_OBUF){0,1}_(INST|i)_.*/I0` konvertiert. Dabei gibt der Ausdruck `(_OBUF){0,1}` an, dass das Teilwort *_OBUF* einmal oder gar nicht vorkommt. Der Ausdruck `(INST|i)` besagt, dass in den gesuchten Namen entweder die Teilzeichenkette `INST` oder das Literal `i` vorkommt. Anschließend folgt eine beliebige Zeichenfolge (`.*`), die auch Zahlen enthalten kann.

Des weiteren führt das Vivado Synthese-Tool Optimierungen durch. Dabei werden Ressourcen entfernt oder zusammengeführt. Falls ein Ausgangssignal eines logischen Gatters ein anderes Gatter antreibt, so werden diese in eine größere LUT zusammengefasst. Um die Wegoptimierung von Signalen zu verhindern, müssen diese im HDL-Code mit `KEEP`-Attribut, wie auf der Abbildung 4.5 dargestellt, annotiert werden.

Im Gegensatz zum `KEEP`-Attribut (bzw. `KEEP_HIERARCHY`) ist das Attribut `DONT_TOUCH` „forward“ annotiert und schließt angesetzte Instanzen auch bei Implementierung aus weiteren Optimierungen aus. Das `DONT_TOUCH`-Attribut kann auf Signale, Module und Komponenten gesetzt werden.

Wie auf der Abbildung 4.5 dargestellt ist, werden die mit `KEEP`-Attribut annotierte Signale sowohl in der RTL-Netzliste als auch im FPGA-Design unterschiedlich benannt. Daher sind für die Lokalisierung dieser Objekte noch weitere Informationen erforderlich. Die Injektoren für Leaf-Pins speichern zusätzlich zu Port- bzw. Modul-Pin-Injektoren noch den Namen des Ausgangssignals von Logikgatter, das solche Fälle abdecken soll. Im FPGA-Design wird in diesem Fall nach Namen gesucht, welche aus dem Namen des Output-Signals und den Suffix `inferred` sowie aus weiteren Substrings bestehen.

4.3.3.3 Multiplexer

Die Multiplexer sind Schaltungselemente, die abhängig von einem Steuersignaleingang S einen von mehreren Dateneingängen auf den Datenausgang

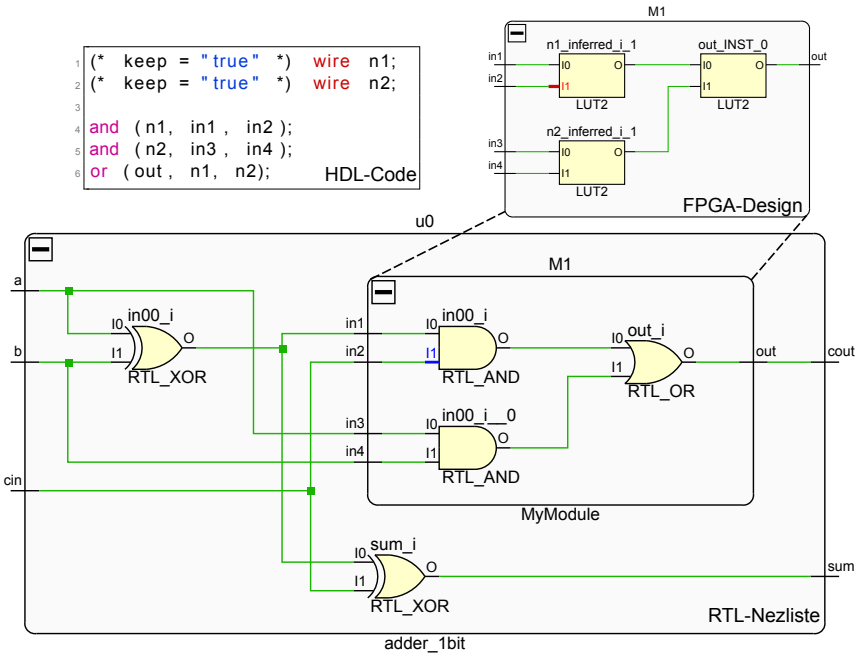

Abbildung 4.5: Keep

schalten. Multiplexer werden zur Implementierung von Schaltfunktionen eingesetzt. Beispielweise kann mittels MUXF7 Multiplexer in Kombination mit zwei LUT6-Instanzen eine logische Funktion bis zur 12 Eingangsvariablen oder ein 8-zu-1-Multiplexer definiert werden [Xil14d].

Wird ein Multiplexer vom Typ MUXFX instanziiert, so bleibt der Name des Steuereingangs unverändert. Bei einigen Schaltungen werden diese durch das Implementierungs-Tool auf LUT3-Primitive abgebildet. In diesem Fall muss beim Mapping der Steuereingang S auf den nächsten verfügbaren LUT3-Eingang abgebildet werden. Bei instanziierten Multiplexer wird ein vollständiges Mapping erreicht.

Die generische Multiplexer, die aus der Verhaltensbeschreibung für selektive Signalzuweisung erstellt werden, werden durch das Synthese-Tool auf LUT-Instanzen des Ziel-FPGAs abgebildet. Ein 4x1-Multiplexer wird z.B. durch eine LUT6-Instanz realisiert. Dabei werden die Steuereingänge S0 und S1 auf die LUT6-Eingängen I4 und I5 abgebildet. Die größere Multiplexer werden durch die Kombination mehrerer LUT6-Instanzen mit Hilfe der Multiplexer MUXF7 und MUXF8 realisiert.

Es ist nicht gelungen ein vollständiges Mapping bei größeren Multiplexer, die auf mehrere LUT-Instanzen abgebildet werden, zu erreichen. Genauso konnte das Mapping bei vielen kleinen Multiplexer (die aus *if*- oder *case*-Blöcken generiert werden), die auf einem einzigen FPGA-Ressource abgebildet werden, nicht vollständig abgedeckt werden. Daher funktioniert das Mapping

für diese Multiplexer durch die Selektion der Leaf-Pins nur teilweise. Diese Elemente können dennoch über Ports bzw. Modul-Pins zur Fehlerinjektion ausgewählt werden.

Das am Beispiel von logischen Gatter und Multiplexer vorgestellte Verfahren für Leaf-Pins wurde für eine Reihe von anderer Elementen intensiv untersucht und entsprechend abgedeckt.

4.4 Fehlerinjektion

Obwohl Stuck-at-Fehler an Input-Objekte wie Ports oder Pins zugewiesen werden, wird die Änderung des FPGA-Designs zur Fehlerinjektion Komponentenbasiert vorgenommen. Sowohl bei den Port-Injektoren als auch bei den Modul-Pin-Injektoren werden zunächst die Leaf-Pins ermittelt, die durch die selektierten Ports (bzw. Modul-Pins) getrieben werden. Die Injektion wird dann entsprechend des Instanztyps des Parent-Cells der betroffenen Leaf-Pins durchgeführt. Dabei sind zwei Arten der Anpassung des Designs für den jeweiligen Leaf-Pin zu unterscheiden:

- Durch Ändern des Attributenwertes einer Instanz (wie bei LUTs oder ROMs), zu welcher der zu injizierte Leaf-Pin gehört.
- Durch Verbinden des betroffenen Leaf-Pins mit einem VCC- oder GND-Netz (wie bei Flip-Flops oder RAMs).

Aufgrund des geänderten Routingstatus (bzw. der neu angelegten logischen Netze) muss beim Verbinden eines Pins an einem logischen Netz (logisch 1 oder logisch 0) anschließend ein partielles Routing durchgeführt werden.

4.4.1 Logische Konstante

Logische Konstante werden in einem Design zur Erzeugung der logischen Werte 1 und 0 gebraucht. Vivado stellt dafür zwei globale logische Konstanten (bzw. physikalische Netze) namens `<const0>` und `<const1>` für ein statisches High-Signal (VCC) und ein statisches Low-Signal (GND) zur Verfügung. Diese Netze werden bei einigen Design-Elementen für die Injektion von Stuck-at-1- oder Stuck-at-0-Fehler verwendet. Die logische Netze eines FPGA-Designs werden durch die Pins *HARD0* und *HARD1* der benachbarten TIEOFFs[2] getrieben.

Die Fehlerinjektion wird bei einigen Design-Elementen durch das Anpassen der Netze vorgenommen. Dafür wird der injizierte Pin aus dem bestehenden Netz herausgelöscht und entsprechend des selektierten Stuck-at-Fehlers zu einem `<const0>`- oder `<const1>`-Netz hinzugefügt. Sollte kein <const0>- oder

[2]FPGA-Ressourcen für die Bereitstellung von Versorgungsspannung oder Masse

<const1>-Netz existieren, so wird ein neues globales Netz entsprechend des Fehlertyps durch die Methode *get_logic_one* oder *get_logic_zero* generiert. Die Anbindung eines Pins an einem generierten oder vorhandenen logischen Netz übernimmt die Methode *pin_tie_logic fault <pin_name>*. Der Tcl-Code im folgenden Listing generiert einen globalen Netz für den logischen Wert 1.

```
create_cell -reference VCC faultVCC
set vcc [get_cells -quiet faultVCC]
create_net -verbose faultOne
set stuckAtOne [get_nets -quiet faultOne]
connect_net -net $stuckAtOne -objects [get_pins -of $vcc]
```

Listing 4.2: Generieren eines logischen Netzes für die Injektion eines Stuck-at-1-Fehlers

Nach der Anpassung der Netze in einem Design entstehen Routingkonflikte. Aus diesem Grund muss sowohl das modifizierte alte Netz als auch das neu angelegte logische Netz neu geroutet werden. Dabei erfolgt das Routing des globalen logischen Netzes durch den Tcl-Befehl *route_design -physical_nets* und der modifizierten Netze mittels *route_design -nets <nets_name>*. Das partielle Routing der Netze nimmt viel Zeit in Anspruch und führt somit zu einem großen Geschwindigkeitsnachteil.

Alternativ zu den oben beschriebenen Implementierungen kann die logische Konstante für das Stuck-at-Fehlermodell mittels Dummy-LUT1-Instanzen erzeugt werden. Im Folgenden wird die Fehlerinjektion auf unterschiedliche Design-Komponenten beschrieben.

4.4.2 Look-Up Table

Eine Look-Up Table (LUT) realisiert eine beliebige boolesche Funktion mit k Variablen, die durch einen INIT-Attribut festgelegt ist. Diese entspricht einer Wahrheitstabelle mit 2^k möglichen Eingangskombinationen. LUTs werden in den SRAM-Zellen gespeichert. Eine LUT mit 2^k SRAM-Zellen bildet 2^{2^k} unterschiedliche Funktionen ab, die durch das INIT-Attribut initialisiert werden.

Ein permanentes Signal an einer der LUT-Eingänge ändert die von dieser LUT implementierte logische Gleichung und somit den Ausgangswert. Eine Fehlerinjektion an einer LUT-Instanz kann durch die Änderung der Initialisierungsparameter durchgeführt werden. Für die Ermittlung und Änderung des INIT-Wertes werden in [Xil14a] zwei Methoden vorgeschlagen:

Logic Table Method Bei dieser Methode wird eine Wahrheitstabelle mit allen möglichen Eingangskombinationen erstellt und aus deren Ausgangswerten (aus der Bitfolge der Ausgangsspalte) der INIT-Wert festgelegt.

Equation Method Hier wird für jeden Eingang der LUT entsprechende Parameter mit Wahrheitswerten definiert, die dann für die Bestimmung

des INIT-Wertes bei der Instanziierung der LUT-Komponente verwendet werden. Danach wird die logische Gleichung der LUT als INIT-String in die LUT-Instanz eingefügt, aus dem der INIT-Wert berechnet wird.

Da bei dem implementierten Ansatz der FPGA-Netzliste zu Beginn der Netzliste generiert und die LUT-Instanzen durch das Synthese-Tool automatisch erstellt werden, wurde bei dieser Arbeit die erste oben genannte Methode implementiert. Im Folgenden wird die Implementierung der Fehlerinjektion an einer LUT genauer beschrieben.

4.4.2.1 Boolesche Algebra

Da die LUTs in Form boolescher Gleichungen vorliegen, folgt in diesem Teil eine kurze Einführung der Booleschen Algebra.

Die Boolesche Algebra besteht aus

- einer Menge der aussagenlogischen Variablen $x_1, x_2, \ldots, x_n$, die nur die Wahrheitswerte aus $\{0, 1\}$ annehmen können,
- den logischen Operatoren NICHT, UND, ODER und XOR $(!, \&, |, \hat{})$ und
- den Klammern „(“ und „)“.

Die in einer LUT enthaltene logische Funktion wird in sogenannten Normalformen beschrieben. In der Booleschen Algebra sind die disjunktive Normalform (DNF) und die konjunktive Normalform (KNF) gebräuchlich. Eine Formel ist in disjunktiver Normalform, wenn sie von der Form $\{K_1, \ldots, K_m, m \geq 0\}$ ist und jedes $K_j, 1 \leq j \leq m$ ein Minterm ist. Ein Minterm besteht aus konjunktiver Verknüpfung (AND) aller Literalen $\{L_1, \ldots, L_n\}$, wobei ein *Literal* eine aussagenlogische Variable oder ihre Negation ist. Eine Formel ist in konjunktiver Normalform, wenn sie ähnlich wie bei DNF von der Form der Konjunktion von Disjunktionen von Literalen ist.

Die von einer LUT definierte logische Funktion kann in disjunktiver Normalform (bzw. in KNF) als Disjunktion der Minterme aus allen Kombinationen der Eingangsvariablen erhalten werden. Dabei wird die Funktion in eine semantisch äquivalente Formel transformiert.

4.4.2.2 INIT to DNF

Um die von einer LUT definierte boolesche Gleichung zu verarbeiten, muss das INIT-Attribut dieser LUT in eine Formel überführt werden. Ein INIT-Wert ist eine Konstante mit folgender Syntax:

$$< size >'< radix >< value >$$

`size` eine dezimale Zahl, die die Bitbreite festlegt.

`radix` einer der Buchstaben `b`, `d`, `h`, die für `binär`, `dezimal` und `hexadezimal` stehen.

`value` der Wert der Konstante in jeweiliger Basis.

Wie in der Tabelle 4.2 dargestellt, werden bei den LUT-Instanzen die boolesche Gleichungen durch den hexadezimalen Initialisierungswert angegeben. Die Bitfolge der Ausgangsspalte (unten beginnend) ergibt dual `0101 1100`. Dies entspricht den hexadezimalen Initialisierungswert $3A$. Die von einer LUT

Inputs			Outputs		
I2	I1	I0	`O=I0 & !I2 + !I1 & I2`		
0	0	0	0	`4'hA`	INIT[0]
0	0	1	1		INIT[1]
0	1	0	0		INIT[2]
0	1	1	1		INIT[3]
1	0	0	1	`4'h3`	INIT[4]
1	0	1	1		INIT[5]
1	1	0	0		INIT[6]
1	1	1	0		INIT[7]
			INIT = `8'h3A`		

Tabelle 4.1: Ableitung des INIT-Wertes aus der Wertetabelle der logischen Funktion `I0 & !I2 + !I1 & I2`.

definierte logische Funktion in DNF kann aus dem INIT-Wert wie folgt ausgelesen werden:

- Eine Wahrheitstabelle mit n Eingangsvariablen wird mittels *truthtable n* als eine Liste generiert.
- Die Methode *init2bin init* ermittelt aus dem `<value>`-Feld des INIT-Wertes die Bitfolge der Output-Spalte und aus dem `<size>`-Feld die Anzahl der Eingänge.
- Die Methode *1-bitsposn bitvec* bestimmt alle Positionen der 1-Bit in der Output-Spalte, aus deren Eingangsvariablen die Minterme erzeugt werden.
- Die Bildung des Minterms für jede Zeile der Wertetabelle, in der eine 1 als Funktionswert steht, übernimmt die Methode *row2minterm row*. Dabei werden Eingansvariablen, für die eine 0 in der Eingangsvariablenspalte

steht, negiert. Eingansvariablen, für die eine 1 in der Eingangsvariablenspalte steht, werden nicht-negiert in die Konjunktion aller Eingangsvariablen aufgenommen.

- Alle Minterme werden untereinander ODER-verknüpft und repräsentieren somit die jeweilige INIT-äquivalente boolesche Funktion, die von der Methode *init2clauseset numinputs bits* als eine Liste der Minterme generiert wird. Eine Klausel ist die Konjunktion von Literalen, das hier als Begriff für die Disjunktion von Literalen statt von Minterm gebraucht wird.

Die Funktion, die durch die Tabelle 4.2 gegeben ist, wird wie folgt in Klauselform aufgestellt:

```
F = {{I0 !I1 !I2} {I0 I1 !I2} {!I0 !I1 I2} {I0 !I1 I2}}
```

Listing 4.3: Die der INIT-Wert 8'h3A äquivalente logische Formel in disjunktiver Normalform

4.4.2.3 LUT-Basierte Fehlerinjektion

Um Fehler einer Look-Up-Table zuzuweisen, muss die logische Gleichung dieser LUT und somit der INIT-Wert geändert und neu gesetzt werden. Nachdem der INIT-Wert - wie oben beschrieben - in einer Formel in DNF (in Klauselmenge) umgewandelt wird, kann nun der Stuck-at-Fehler übernommen werden. Ein Stuck-at-Fehler bedeutet eine Belegung der Inputvariablen auf den Wert 0 oder 1. Die Vereinfachung der Formel durch die Belegung eines Stuck-at-Fehlers von einer Eingangsvariable wird durch die Methode *simplify clauseset variable assignment* wie folgt durchgeführt:

- Ist an einem der Eingangsports eines Gatters Stuck-at-0 selektiert, so wird die durch das Gatter dargestellte Klauselmenge folgendermaßen vereinfacht:
  - lösche Klauseln, in denen der Literal positiv vorkommt und
  - lösche negative Literale aus allen Klauseln.
- Ist an einem der Eingangsports eines Gatters Stuck-at-1 selektiert, so wird der durch das Gatter dargestellte Term folgendermaßen vereinfacht:
  - lösche Klauseln, in denen der Literal negativ vorkommt und
  - lösche positive Literale aus allen Klauseln.

Die Klauselmenge im Listing 4.3 vereinfacht mit Stuck-at-0 an der Input-Variable `I0` sieht dann folgendermaßen aus:

```
F'= {{!I1 I2}}
```

4.4.2.4 DNF to INIT

Nachdem, wie oben beschrieben, eine INIT-äquivalente Formel generiert und entsprechend des Stuck-at-Fehlers vereinfacht wurde, folgt anschließend die Konvertierung dieser Formel in einen INIT-Wert. Dies erfolgt mit der Methode *expr2init numinputs expression*. Dabei wird wieder eine Tabelle mit allen möglichen Eingangskombinationen erstellt und der vereinfachte Formel dementsprechend ausgewertet. Die Bitfolge der Ausgangsspalte der vereinfachten Formel entspricht dann dem neuen Initialisierungswert. Folgende Tabelle zeigt den neuen ermittelten INIT-Wert für einen Stuck-at-0-Fehler am Tabelleneingang `I0`:

Inputs			Outputs		
I2	I1	I0	`O'=!I1 & I2`		
0	0	0	0	`4'h0`	INIT[0]
0	0	1	0		INIT[1]
0	1	0	0		INIT[2]
0	1	1	0		INIT[3]
1	0	0	1	`4'h3`	INIT[4]
1	0	1	1		INIT[5]
1	1	0	0		INIT[6]
1	1	1	0		INIT[7]
			INIT = `8'h30`		

Tabelle 4.2: Ableitung des INIT-Wertes aus der Wertetabelle der durch einen Stuck-at-0-Fehler modifizierten Funktion `!I1 & I2`

Das Auslesen und Neusetzen des INIT-Attributs erfolgt durch die *get_property INIT <lut_name>* und *set_property INIT <value> <lut_name>* Methoden.

4.4.3 Read-Only Memory

Ein Read-Only Memory (ROM) ist ein nichtflüchtiger Speicher, dessen Inhalt mit wahlfreiem Zugriff nur gelesen werden kann. Bei FPGAs der Serie 7 können durch die Verschachtlung von LUT-Instanzen größere ROM-Speicherblöcke gebildet werden. ROMs können in einem Design sowohl durch die Instanziierung als auch durch das Setzen des `ROM_STYLE`-Attributs auf `distributed` auch im HDL-Code erzeugt werden.

Der Inhalt eines ROM-Speicherblocks werden wie bei LUTs durch das INIT-Attribut festgelegt. Daher funktioniert die Fehlerinjektion für ROM-Instanzen mit dem bei LUTs detailliert beschriebenen Verfahren der booleschen Algebra.

Die größeren ROM-Blöcke, die aus zwei oder mehr LUT-Instanzen bestehen, sind als Makrozellen definiert. Bei Makrozellen sind die enthaltenen Leaf-Zellen nicht bearbeitbar und haben nur lesbare Attribute. Daher wird bei der Injektion nur die Makrozelle modifiziert, wenn auch die Selektion zur Fehlerinjektion an Leaf-Zellen stattfinden sollte.

Bei ROMs kann sowohl der ganze Inhalt durch die Option `inject_multiple` mit Selektion aller Eingänge als auch einzelne Eingänge auf einen ausgewählten Stuck-at-Fehler gesetzt werden.

4.4.4 Block-RAM

In einem FPGA-Design können neben LUTs, die sich als Speicher konfigurieren lassen, auch eine Vielzahl an Block-RAMs verwendet werden. Ähnlich wie bei ROMs können Block-RAM-Ressourcen manuell durch die Instanziierung der RAM-Primitiven oder durch die HDL-Codierung mit dem Datentyp `RAM_TYPE` implementiert werden. Abhängig von der HDL-Codierung unterscheidet das Synthese-Tool die zwei Synchronisationsarten Read-First und Write-First. Bei Read-First-Mode werden die Daten aus dem Speicher eingelesen, bevor die aktuellen Daten des Eingangsbusses in den Speicher geladen werden. In Write-First-Mode werden die Daten des Eingangsbusses taktsynchron auf dem Datenausgang gelegt und gleichzeitig in den Speicher geschrieben.

Bei der Block-RAMs sind Fehler in den Adress- und Dateneingängen injizierbar. Beispielsweise kann Lesen und Schreiben des Block-Rams durch Injizieren vom Stuck-at-0-Fehler in den WE (Write Enable) -Pins abgeschaltet werden. Die Fehlereffekte sind abhängig vom Synchronisationsart taktsynchron oder zeitlich versetzt am Datenausgang erkennbar.

4.4.5 Weitere Instanzen

Die Fehlerinjektion für alle weiteren Instanzen, die hier nicht beschrieben sind, erfolgt durch Verknüpfen derer Eingänge mit den entsprechenden logischen Konstanten. Beispielsweise sind alle Eingänge, außer der Takteingang eines Flip-Flops, injizierbar. Der zu injizierende Input-Pin des Flip-Flops wird vom bestehenden Netz abgelöst und mit einem entsprechenden logischen Netz verbunden.

5 Ablauf der Fehlerinjektion

Dieses Kapitel stellt eine vollständige Beschreibung des Designflows der Fehlerinjektion dar, die im Rahmen dieser Arbeit implementiert und mehrfach getestet wurde. Vivado bietet unterschiedliche Entwicklungsabläufe eines Designs. Zum einen durch den graphischen Modus, bei dem alle Befehle in einem sogenannten Flow-Navigator ausgeführt werden, zum anderen durch die Eingabe der Befehle in der Vivado Tcl-Shell. Bei der letzteren Methode können nützliche Tcl-Skripte entwickelt und als eine Tcl-App zur Verfügung gestellt werden. Solche Apps können dann in die Vivado-Designumgebung integriert werden und somit die Kapazität von Vivado erweitern. Die implementierte Tcl-App zur Fehlerinjektion kann sowohl im Tcl-Mode, als auch im GUI-Mode ausgeführt werden.

Der erste Teil dieses Kapitels beschreibt den Non-Project Designflow für den schnellen und automatisierten Ablauf der Fehlerinjektion, sowie die erforderlichen Datenstrukturen für das Mapping und dem Designablauf. Im zweiten Teil wird der GUI-basierte Designflow mit der implementierten Tcl-Methoden dargestellt. In Vivado ist der Designablauf in zwei Modi verfügbar, nämlich im Project-Mode und im Non-Project-Mode. Beide Modi bieten unterschiedliche Vorteile wie automatische Verwaltung von dem Projekt durch Vivado in Project-Mode oder volle Kontrolle des Designablaufes durch Tcl-Befehlen in Non-Project-Mode.

In Folgenden wird die Verzeichnisstruktur der Implementierung vorgestellt, bevor der Ablauf der Fehlerinjektion detailliert beschrieben wird.

5.1 Verzeichnisstruktur und Ablaufkontrolle

Abbildung 5.1 gibt einen Überblick über die Verzeichnisse und die darin befindlichen Ein- und Ausgabedateien, welche während der Ausführung des Tools generiert werden. Für einen fehlerfreien Ablauf muss diese Ordnerstruktur beibehalten werden. Der Ordner `Scripts` enthält alle Skripte, die in dieser Arbeit zur Fehlerinjektion entwickelt wurden. Diese werden im Startskript `fault.tcl` durch die `source`-Anweisung wie folgt zur Ausführung gebracht:

```
source tclDir/<script_name.tcl>
```

Das Hauptskript `fault.tcl` im `faultutils` Ordner enthält die notwendigen Variablen zur Kontrolle des Ablaufes der Fehlerinjektion, die im folgenden aufgeführt und beschrieben werden.

`part` Definiert das Ziel-FPGA, auf der das Design ausgeführt wird.

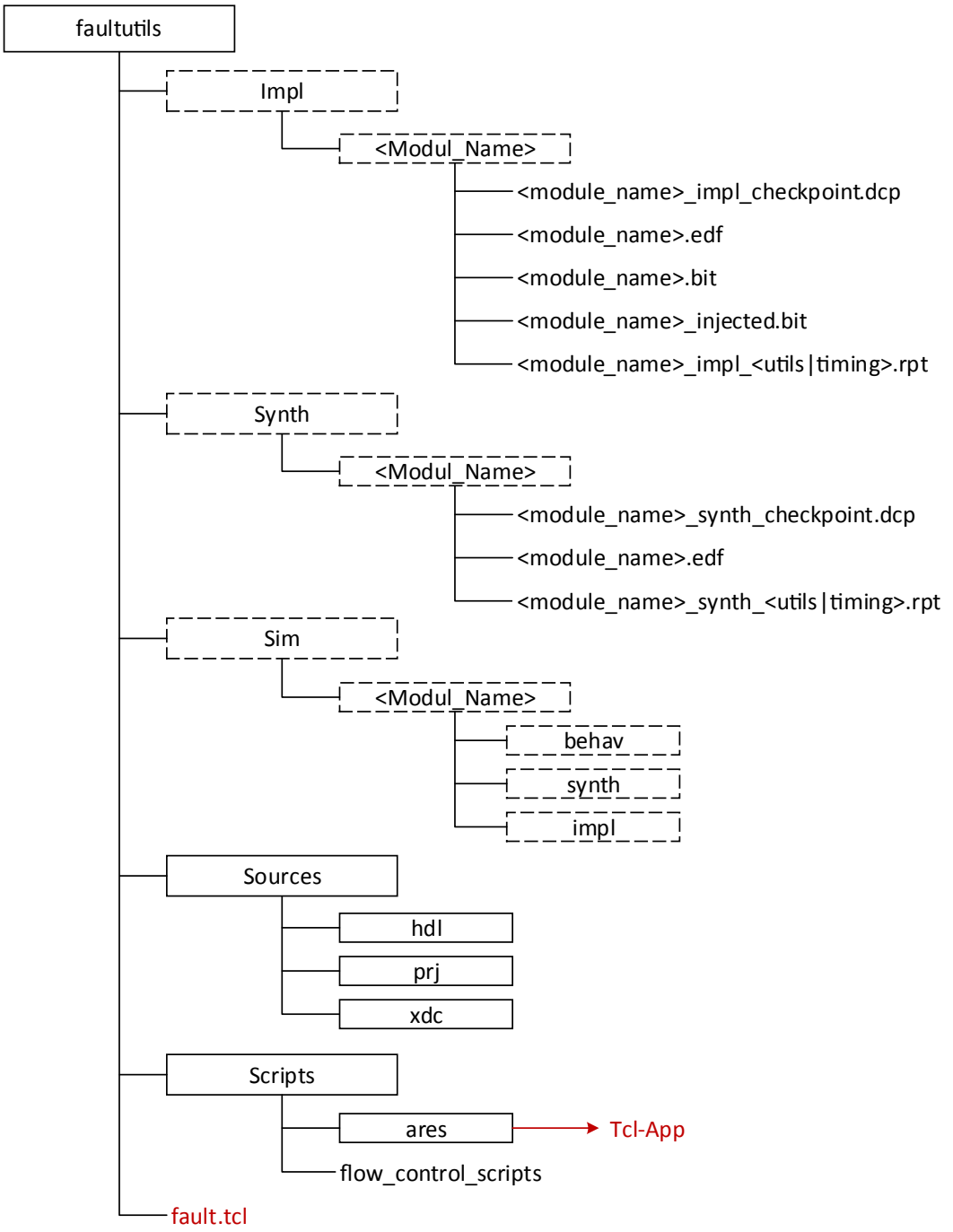

Abbildung 5.1: Verzeichnisstruktur des implementierten Tools

```
1 set device        "xc7z020"
2 set package       "clg484"
3 set speed         "-1"
4 set part         \$device\$package\$speed
```

Listing 5.1: Angaben zur Zielplattform

In dieser Arbeit wurde als Testplattform ein *ZedBoard* verwendet, das auf dem *Xilinx Zynq-7000 All Programmable SoC* basiert. Die Implementierung setzt aber keine konkreten FPGA-Familien voraus, außer die, die in der Vivado-Designumgebung verfügbar sind.

runFault Steuert die Art des Ablaufs. Ist die Variable auf 0 gesetzt, wird das Tool als reguläre RTL-to-Bitstream Designflow ohne Fehlerinjektion ausgeführt. Setzt man diese hingegen auf 1, so wird der implementierte Flow der Fehlerinjektion gestartet.

phase Steuert den Ablauf der Fehlerinjektion. Mit dieser Variable kann festgelegt werden, ob das Injizieren von Defekten in das Post-Synthese oder Post-Implementation Design stattfinden soll (siehe Abbildung 5.3). Dies erfolgt durch das Setzen der Variable `phase` auf `postSynth` oder `postImpl`.

runSim Startet die Post-Synthesis oder Post-Implementation Fehlersimulation, welche mit der Variable `phase` definiert ist.

synthDir, implDir, simDir Definieren den Pfad zum Output-Ordner zum Speichern der Dateien (Synthetisierte Netzliste, Checkpoint- und Bitstream-Datei etc.), die während der Ausführung der Tcl-Skripten generiert werden.

sourceDir Definiert den Pfad zu Input-Ordnern, in dem die Quelldateien wie HDL-, XDC- und PRJ-Dateien angelegt sind.

tclDir Definiert den Pfad zum `Scripts`-Ordner, in dem alle Tcl-Skripte zur Kontrolle der einzelnen Designschritte sowie das Tcl-App zur graphischen Fehlerinjektion befinden.

In dem nächsten Abschnitt des Skriptes `fault.tcl` werden einige Testmodule mit unterschiedlichen Input-Methoden von Quelldateien und deren Attribute definiert.

5.1.1 Anlegen von Testmodulen

Die Informationen der zu testenden Schaltungen wie Top-Level-Modul, Synthese- und Implementierungsoptionen, Pfade zu Quelldateien, Namen der

zu speichernden Dateien usw. werden in einem mehrdimensionalen Array namens `<modulName>Attribute`[1] verwaltet. Hier sind einige wichtige Attribute zu benennen:

Die Attribute `rtlDesign`, `synthDesign` und `implDesign` legen die Design-Namen fest. Das Design samt den zugehörigen Namen werden durch die Anweisungen, wie *synth_design -rtl -name <rtl_design> ...* und *link_design -name <synth_design/impl_design> ...*, in den Hauptspeicher geladen. Diese Default-Namen werden zur schnellen Umschaltung zwischen RTL-Design und Post-Synthesis oder Post-Implementation Design benötigt. Wenn das Design des jeweiligen Schrittes schon im Hauptspeicher initialisiert ist, kann dies mittels *current_design <design_name>* schnell neue geöffnet werden.

Das Setzen und das Auslesen eines Attributes von einem Modul erfolgt durch die Methoden *set_attribute name attribute value* und *get_attribute name attribute*. Die folgenden Attribute legen beispielsweise die Synthese-Option und den Namen der Checkpoint-Datei des synthetisierten Designs fest. Es werden für die Implementierung und andere Output-Dateien entsprechende Defaultwerte spezifiziert.

```
1 set_attribute $name "synthOptions" "-flatten_hierarchy
    none -no_lc"
2 set_attribute $name "synthCheckpoint" "${modulName}
    _synth_checkpoint.dcp"
```

Listing 5.2: Festlegen der Synthese-Optionen sowie der Name der Synthese-Checkpoint-Datei

Die Methode *create_module <moduleName>* erstellt für jedes zu testendem Modul ein Array `<modulName>Attribute`, mit dem oben benannten Defaultwerten an. Das Attribut `moduleName` definiert den Pfad zum Input/Output-Ordner des angelegten Moduls. Das Attribut `topModule` definiert hingegen den Namen des Top-Level-Moduls der Schaltung, der bei vielen Tcl-Befehlen als Argument erwartet wird. Wird das Top-Modul nicht definiert, so wird als Top-Level-Modul `moduleName` übernommen.

Nun folgt anhand der Referenzschaltung (siehe Anhang A) das Einlesen der Quelldateien für das zu erstellende Projekt.

```
1 set adder2Bit "2bitAdder"
2 set topModule "adder_2bit"
3 create_module $adder2Bit
4 set_attribute $adder2Bit   topModule $topModule
5 set_attribute $adder2Bit   prj       $prjDir/${topModule}
    .prj
```

[1] Diese Datenstruktur mit einiger Default-Attribute für Synthese und Implementierung wurde teilweise aus [Xil14e] übernommen.

```
6 set_attribute $adder2Bit   xdc       [list $xdcDir/${
    topModule}.xdc]
```

Listing 5.3: Anlegen eines VHDL-Modul für 2-Bit-Volladdierer

Im Listing 5.3 werden die Quelldateien aus einem Projekt-Datei, die durch das Attribut `prj` definiert sind, eingelesen. Eine Projekt-Datei ist eine Textdatei mit der Endung `.prj`, in der alle VHDL- und Verilog-Dateien einer Schaltung aufgelistet sind. Wenn dieses Attribut gesetzt ist, wird diese Datei geparst und alle darin befindlichen HDL-Dateien mittels *read_verilog* oder *read_vhdl* in den Hauptspeicher geladen. Eine Zeile der `prj`-Datei enthält folgende Quelldatei-Informationen [Xil13]:

```
verilog|vhdl <library_name> {<file\_name>.v|.vhd}
```

wobei

`verilog|vhdl` angibt, ob die Quelldatei eine Verilog- oder VHDL-Datei ist,

`<library_name>` definiert die Arbeitsbibliothek, in der HDL-Dateien kompiliert werden und

`<file_name>` die HDL-Datei festlegt, die kompiliert werden soll. Dabei können die Dateien als absoluter oder relativer Pfad angegeben werden.

Alternativ zur einer Projekt-Datei können die Quelldateien als Pfad zum Unterordner, in dem diese sich schon befinden oder wie folgt in einer Liste definiert werden:

```
1 set_attribute $top  vhdl  [list $rtlDir/$top/${top}.vhd \
2                             $rtlDir/$top/half_adder.vhd]
```

Es empfiehlt sich, für jede Schaltung ein separates Unterverzeichnis anzulegen. Am Ende der `fault.tcl` wird schließlich durch den Aufruf von *run_fault <modulName>* der ganze Ablauf gestartet.

Der implementierte Designflow zur Fehlerinjektion erweitert den Vivado Designflow wie auf Abbildung 5.3 ersichtlich ist. Dafür waren folgende Datenstrukturen erforderlich, die in folgendem erklärt werden:

5.2 Injektoren

Um den Port- oder Pin-Objekten der RTL-Netzliste einen Fehler zuzuweisen, werden entsprechende Injektoren erzeugt. Ein Injektor ist eine globale Variable vom Typ `Record`, ähnlich wie die Datenstruktur `struct` in C. Sie speichert für jeden zu injizierenden Port oder Pin den zugewiesenen Fehler und weitere Informationen, die in Listing 5.4 aufgelistet sind. Injektoren werden für ein

erfolgreiches Mapping sowie für das Ändern des Designs in Abhängigkeit des selektierten Fehlers benötigt.

Bei jedem Start des Tools wird die Injektor-Datenstruktur einmalig erstellt. Während der Fehlerinjektion wird dann für jeden zugewiesenen Fehler eine Instanz vom erstellten Injektor erzeugt. Diese Instanzen (sogenannte Injektoren) dienen zur Grundlage der komponentenbasierter Fehlerinjektion und existieren bis zur Modifikation des Designs.

Zum einen werden die selektierten Objekte durch das Mapping im FPGA-Design mit Hilfe der durch die Injektoren bereitgestellten Informationen referenziert. Zum anderen wird das FPGA-Design abhängig von der Art des zu injizierenden Fehlers angepasst.
Der Injektor-Datenstruktur hat folgenden Aufbau:

```
1  struct::record define injector {
2              selInput
3              isPort
4              isLeaf
5              driverPin
6              parentCell
7              refPinName
8              outNet
9              leafPins
10             faultType
11         }
```

Listing 5.4: Datenstruktur des Injektors

Im Folgenden werden alle Parameter der Injektoren kurz erläutert:

name Definiert Name des zu erstellenden Injektors. Ein Injektor hat einen eindeutigen Namen und wird aus den hierarchischen Namen des selektierten Objekts geleitet. Dabei werden die vorkommenden Hierarchie-Separatoren, sowie öffnende eckige Klammer durch Unterstrich ersetzt. Weiterhin werden schließende eckige Klammer aus dem geleiteten Namen entfernt.

Die Bestimmung der eindeutigen Namen von Injektoren erfolgt durch die Methode *gen_unique_injector_name selPin isPort*. Damit kann man auch leicht auf dem für das jeweilige Input bereitgestellten Injektor zugreifen.

selInput Kennzeichnet hierarchischen Namen der selektierten Input-Objekte.

isPort Kennzeichnet das selektierte Port-Objekt.

isPin kennzeichnet das selektierte Leaf-Pin-Objekt.

driverPin Treiber-Pin der Netze, an der der selektierte Pin gebunden ist.

parentCell Kennzeichnet eindeutig die Zelle, zu den der selektierte Pin gehört.

outNet Kennzeichnet den Ausgangsnetz des Parent-Cells.

leafPins Liste aller zu injizierenden Leaf-Pins. Bei einer direkten Selektion eines Leaf-Pins wird dessen hierarchischen Name in Form eines regulären Ausdrucks in diesem Feld gespeichert. Beim Mapping werden alle Leaf-Pins der jeweiligen Injektoren durch die gefundene Pins im FPGA-Design ersetzt.

faultType Spezifiziert den Typ des zu injizierenden Stuck-at-Fehlers.

Die Injektoren werden durch *create_injector args* erzeugt und deren Inhalt kann durch den Aufruf von *print_all_injectors* bzw. *print_injector name* ausgegeben werden. Die Methode *set_injector_attribute name attribute value* setzt den Wert eines Attributes für ein gegebenes Injektor. Mit dieser Methode kann z.B. die gleiche Selektion mit einem anderen Fehler-Typ betrachtet werden, ohne die Selektion im RTL-Design neu durchzuführen. Ähnlich kann der Wert eines Attributs aller Injektoren mittels *set_all_injector_attribute attribute value* auf dem gleichen Wert gesetzt werden.

5.3 Phasen der Fehlerinjektion

Um einen automatisierten Ablauf der Fehlerinjektion - basierend auf dem Vivado Non-Project Designflow - zu ermöglichen, wurde der auszuführende Tcl-Code im Array `faultFlow` gespeichert. Die Felder des `faultFlow`-Arrays enthalten die jeweiligen Codeabschnitte, die in den Zuständen des endlichen Zustandsautomaten in Abbildung 5.3 ausgeführt werden.

Mit diesem Zustandsautomat wird der zyklische Ablauf der Fehlerinjektion realisiert, der aus jedem beliebigen Zustand durch die Anweisung *resume_flow <step>* weiter ausgeführt werden kann. Dieser ist sehr hilfreich, falls durch einen Fehler der Flow abgebrochen wird. Darüber hinaus können die jeweiligen Schritte durch *run_flow step* separat ausgeführt werden.

Der implementierte Designflow der Fehlerinjektion besteht aus folgenden Schritten:

select Enthält Tcl-Code zur Generierung des RTL-Designs aus den im Hauptspeicher eingelesenen Quelldateien. Durch Ausführung von *synth_design -rtl <...>* wird RTL-Quellcode in eine generische Netzliste umgewandelt und das RTL-Design im Hauptspeicher geöffnet. Schließlich wird ein Input-Port oder -Pin der generischen RTL-Netzliste selektiert. Die Selektion kann auch mit mehreren Elementen der RTL-Netzliste erfolgen.

Die Methode *inject_fault -fault stuck-at-0/1 <...>* (siehe 5.9) erzeugt für die selektierten Port- und Pin-Objekte entsprechende Injektoren. Dabei kann die Selektion durch das Starten der GUI mit *start_gui* in RTL-Schematic vorgenommen werden. Dafür muss aber nach *start_gui* zusätzlich der *return*-Befehl eingefügt werden, um die Weiterführung des nachfolgenden Codes zu verhindern. Da so die Ausführung des Designflows unterbrochen wird, muss nach der graphischen Selektion der Objekte der Flow mittels *resume_flow* fortgesetzt werden. Verzichtet man hingegen auf eine graphische Selektion der Elemente in RTL-Schematic, so können mittels Tcl-Befehlen (*get_ports* und *get_pins*) Port- und Pin-Objekte selektiert werden, ohne den Ablauf der Fehlerinjektion zu unterbrechen.
Folgendes Listing zeigt den im Zustand `select` ausgeführten Tcl-Code:

```
synth_design -rtl -name $rtlDesign -top $topModule
   -part $part
inject_fault -fault stuck-at-1 -objects u1/sum_i/I1
```

Listing 5.5: Elaborating des Designs und Selektion eines Leaf-Pins sowie einem Stuck-at-1-Fehler

synth Lädt mittels *read_edif* das synthetisierte und als Netlist-Datei angelegte Design in den Hauptspeicher. Dabei wird die Netlist-Datei ohne die Initialisierung des Designs in den vorhandenen Datensatz des aktuellen Projekts hinzugefügt. Daher muss noch der Tcl-Befehl *link_design* ausgeführt werden, um das synthetisierte Design zu initialisieren und es in den Hauptspeicher zu laden. Das zuvor synthetisierte und als Netlist-Datei angelegte Design wird wie im folgenden Listing in den Projekt geladen.

```
read_edif $synthDir/$moduleName/${topModule}.edf
link_design -name $synthDesign -top $topModule -part
   $part
```

Listing 5.6: Laden des zuvor synthetisierten Designs in den Hauptspeicher

impl Ähnlich zu `synth` wobei das implementierte Design, das als Checkpoint-Datei vorliegt, in den Hauptspeicher geladen und anschließend geöffnet wird. Der Tcl-Code in folgendem Listing initialisiert das zuvor implementierte und als Checkpoint gespeicherten Design und lädt es in das aktuelle Projekt.

```
read_checkpoint $implDir/$moduleName/$implCheckpoint
   -part $part
```

```
link_design -name $implDesign -top $topModule -part
   $part
```

Listing 5.7: Laden des zuvor implementierten Designs in den Hauptspeicher

inject Führt die Änderungen im Design abhängig vom selektierten Defekt durch. Im ersten Schritt werden Anhand des Mapping-Prozesses alle Pins der betroffenen Primitiven referenziert und den Injektoren im RTL-Design zugewiesen. Das Mapping führt ein Update auf alle Injektoren durch, indem alle in der RTL-Netzliste selektierten Leaf-Pins durch im FPGA-Design gefundenen Pins ersetzt werden. Danach wird durch die Methode *modify_netlist <phase>* die komponentenbasierte Änderungen im Design vorgenommen.

Das Argument `phase` legt fest, ob die Injektion von Defekten in dem synthetisierten oder in dem implementierten Design stattfinden soll. Findet die Fehlerinjektion nach der Synthese statt, so wird das Design noch platziert und geroutet, bevor aus diesem noch der Bitstream generiert wird. Findet die Fehlerinjektion hingegen nach der Implementierung statt, so müssen noch bestimmte Netzrouten neu angepasst werden.

Da die Änderungen im Design komponentenbasiert vorgenommen werden, erfolgt ein partielles Routing nach der Injektion von Fehlern wie folgt:

- Bei LUT-Instanzen ist kein Routing notwendig, da deren Anpassung zur Fehlerinjektion durch das Ändern ihrer INIT-Attribute vorgenommen werden.
- Bei anderen Instanzen muss ein partielles Routing durchgeführt werden, da deren Anpassung zur Fehlerinjektion durch das Betreiben der injizierenden Pins mit einem GND- oder VCC-Netz erfolgt.

simulate Startet die funktionale Post-Synthesis bzw. Post-Implementation Simulation mit injizierten Fehlern. Folgende Abbildung zeigt dabei die auszuführende Befehle der Simulation:

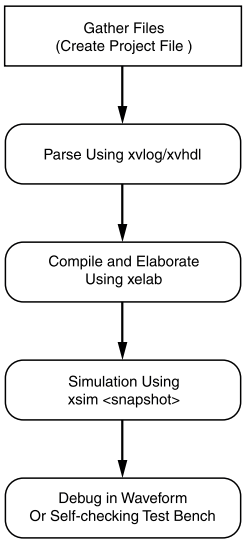

Abbildung 5.2: Ablauf der funktionalen Simulation in Vivado [Xil14h]

emulate Generiert den Bitstream, der anschließend zur Fehleremulation auf das FPGA übertragen wird. Hier kann der graphische Hardware-Manager wie in `select` mittels *start_gui* benutzt werden, um die Programmierung des FPGAs im GUI durchzuführen.

resume Startet den nächsten Ablauf der Fehlerinjektion oder beendet ihn.

Der komplette Tcl-Code von `faultFlow` befindet sich im Anhang.

5.4 Tcl-basierter Designflow

Abbildung 5.3 zeigt den Ablauf der Fehlerinjektion als ein endlicher Zustandsautomat. Jeder Zustand stellt dabei notwendige Schritte im Designflow dar. Die entwickelten Skripte basieren auf dem in [Xil14f] vorgestellten Designflow, sowie in [Xil14j, Xil14i, Xil14g].
In Non-Project-Mode werden Quelldateien eines Designs Anhand der *read_** Befehlen eingelesen. Zum Verwalten der Designdaten wird eine Datenbank mit dem ganzen Projekt und dessen Eigenschaften im Hauptspeicher erstellt, die interaktiv mit Tcl-Befehlen manipuliert werden können. Beispielsweise ist das Ändern des Attributwertes oder das Setzen neuer Attribute eines Designobjektes möglich. Der aktuelle Status der Designdatenbank im Hauptspeicher kann nach jedem Schritt des Designflows mittels *write_checkpoint* gesichert werden, welche hier zum Speichern des fehlerfreien Designs angewendet wurden. Bevor der in Abbildung 5.3 dargestellte Designflow gestartet wird, werden alle Dateien des zu erstellenden Designs mittels *read*-Befehlen gelesen und in den Hauptspeicher geladen, die schließlich durch den Aufruf von *synth_design*

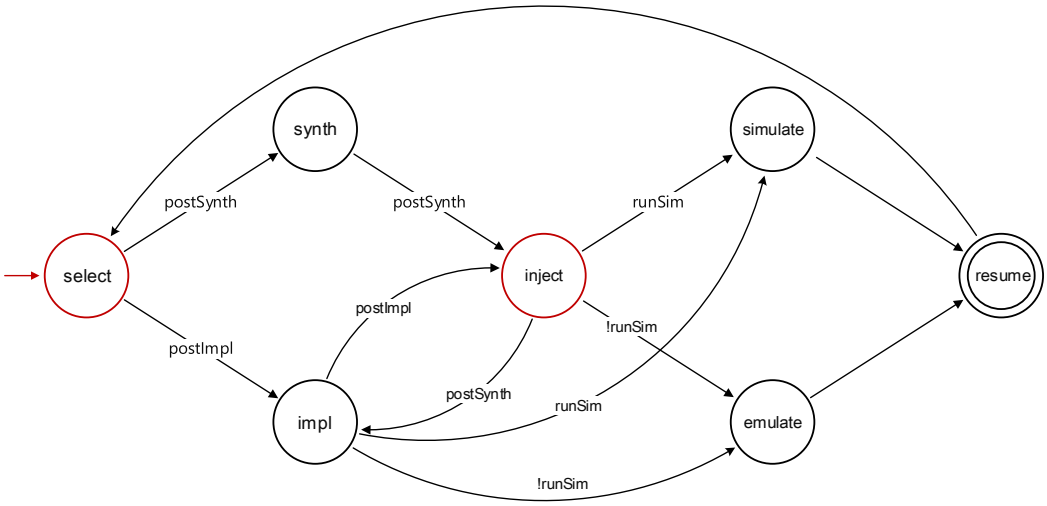

Abbildung 5.3: Zustandsdiagramm des Ablaufs der Fehlerinjektion in Vivado

kompiliert werden. Mit der Anwendung der in XDC-Dateien definierten Beschränkungen werden die aus der HDL-Beschreibung generierten Komponenten auf die ausgewählten FPGA-Architektur abgebildet.

Beim ersten Start des Faultflows wird das Design synthetisiert und danach implementiert (Place&Route). Das synthetisierte bzw. implementierte Design wird als Design-Checkpoint gespeichert. Das Ergebnis des synthetisierten bzw. implementierten FPGA-Designs wird in den jeweiligen Schritten des Designflows zur Fehlerinjektion durch *read_checkpoint* in den Hauptspeicher geladen.
Den Designflow der Fehlerinjektion in Tcl-Mode führt man wie folgt in der Windows-Konsole aus:

```
cd .../faultutils
vivado -mode tcl -source ./fault.tcl -notrace
```

In jedem Schritt des Faultflows werden jeweilige Unterordner im Hauptverzeichnis des entwickelten Tools erzeugt.

5.5 GUI-basierter Designflow

Im Gegensatz zum automatisierten Ablauf der Fehlerinjektion im Tcl-Mode müssen die Schritte des Ablaufs der Fehlerinjektion im GUI-Modus manuell durchgeführt werden. Im graphischen Modus kann das Tool mit allen Vorteilen von Vivado genutzt werden. Beispielsweise kann man über den Flow-Navigator sehr schnell zwischen Designs wechseln und den aktuellen Zustand des Design graphisch betrachten, was aber im Batch-Modus über Tcl-Befehle erfolgt.

Die entwickelten Tcl-Skripte zur Fehlerinjektion für den GUI-Mode wurden in Form einer Tcl-App zusammengefasst und wie in Abbildung 5.1 kennzeichnet ist, in einem separaten Unterordner gespeichert. Die entwickelte Tcl-App erweitert bestehende Befehlssätze von Vivado um neue Tcl-Befehle, die sowohl in Batch-Mode als auch in GUI-Mode ausgeführt werden können.

Um die entwickelten Tcl-Methoden zu initialisieren, muss das Tcl-App wie folgt in die Vivado-Designumgebung integriert werden:

```
source .../ares/faults/faults.tcl
package require ::ares::faults
```

Listing 5.8: Initialisierung des Tcl-Apps in Vivado

Die Methoden sind unter Namespace *::ares::faults::* implementiert und können nach der Initialisierung in der Tcl-Shell ausgeführt werden. Nachfolgend sind die erforderliche Tcl-Befehlen mit allen möglichen Parametern zur Fehlerinjektion aufgelistet.

Folgender Tcl-Befehl weist Injektoren die selektierten Input-Elemente der RTL-Netzliste mit einem Stuck-at-Fehler zu:

```
inject_faults

Description:
Responsible for assigned faults in the input port or input pin
   in the design.

Syntax:
 inject_faults [-fault <stuck-at-0|stuck-at-1>]
               [-objects <args>] [-inject_multiple]
               [-bus_bit <arg>] [-multiple_stuck_at]
               [-design_based] [-help]

Usage:
  Name                          Description
  ---------------------------------------
  [-fault]              Possible fault types to inject in
                        an input ports and pins
                        Default:stuck-at-1
  [-objects]            Input port/pin or list of the input
                        ports and pins to be injected
                        Default:Selected input objects in rtl
                        schematic
  [-bus_bit]            Inject specified bit of a bus
                        Default: Most significant bit
  [-inject_multiple]    Sets a list of input ports/pins to
                        inject the same fault type
  [-multiple_stuck_at] Allow injecting multiple stuck-at faults
                        (injection of different fault types)
  [-design_based]       Allow fault injection in the fpga design
                        without mapping
```

```
  [-help]              Print this help message
```

Listing 5.9: Tcl-Befehl für die Selektion der Elemente des RTL-Designs sowie des zu injizierenden Fehlers

Nachdem die Injektoren durch die Methode *::ares::faults::inject_faults* erzeugt wurden, können nun Änderungen in synthetisierten bzw. implementierten Designs mit folgendem Befehl vorgenommen werden:

```
modify_netlist

Description:
Injecting faults into the FPGA design. Synthesized or
   implemented netlist will be modified.

Syntax:
 modify_netlist [-phase|-p <synth|impl>] [-help|-h]

Usage:
  Name                Description
  ------------------------------
  [-phase <arg>]      Defines design netlist in which fault will
                      be injected.
                      Default: impl
  [-help]             Print this help message
```

Listing 5.10: Tcl-Befehl für die Änderung des synthetisierten oder implementierten Designs zur Fehlerinjektion

Die Methode *::ares::faults::modify_netlist* führt Änderungen im Design abhängig von den ausgewählten Stuck-at-Fehler durch. Dabei wird zunächst das Mapping durchgeführt, um die referenzierbare Pins zu lokalisieren. Um das Mapping ohne die Modifikation des FPGA-Designs zu testen, kann der Befehl *::ares::faults::mapping* separat durchgeführt werden.

Bei der DBF-Injektion wird sowohl die Selektion der Inpjut-Objekte als auch die Anpassung der Elemente eines synthetisierten oder implementierten Design zur Fehlerinjektion mittels *::ares::faults::inject_faults* durch Angabe des Argument *-design_based* durchgeführt.

Anschließend veranschaulichen die folgenden Abbildungen die Simulation eines fehlerfreien und eines fehlerbehafteten Designs mit der entwickelten Tcl-App:

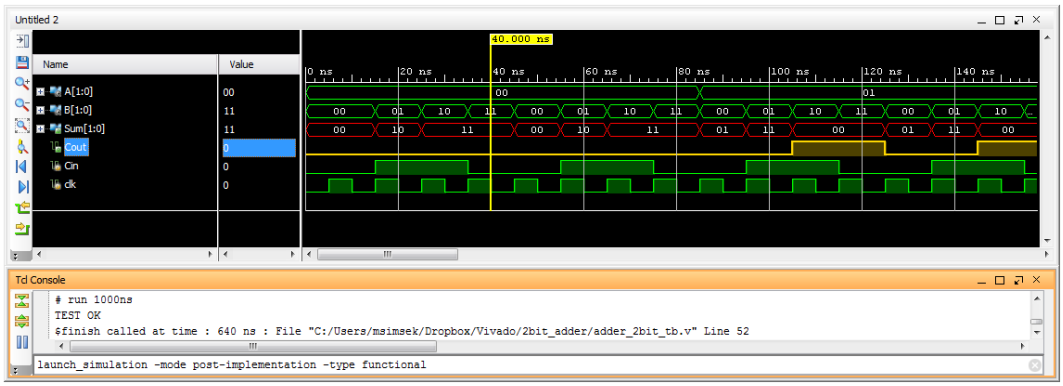

Abbildung 5.4: Simulation der fehlerfreien Referenzschaltung

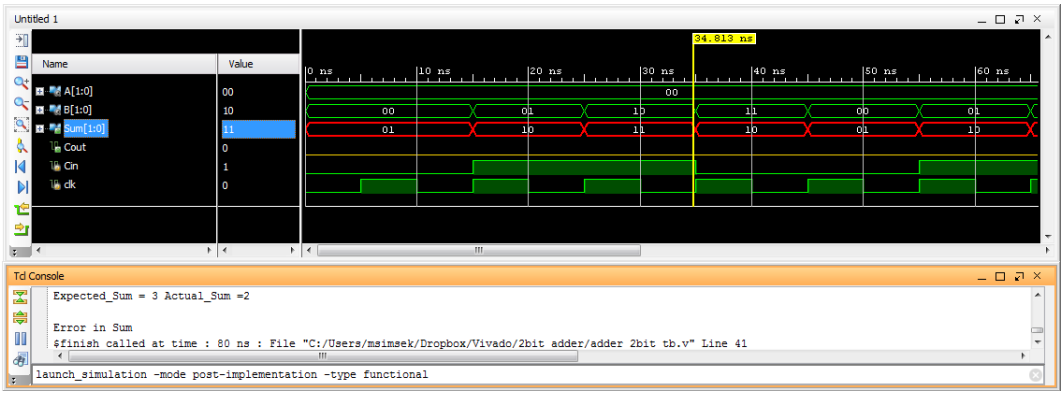

Abbildung 5.5: Simulation der Referenzschaltung mit einem Stuck-at-1-Fehler

6 Ergebnisse

In diesem Kapitel wird das implementierte Verfahren zur Fehlerinjektion bewertet. Dabei wird zunächst der Abdeckungsgrad des Mappings untersucht, da dieser den Kernteil der Fehlerinjektion ausmacht. Danach werden die Ausführungszeiten für die Fehlersimulation und -emulation des Post-Synthesis oder Post-Implementation Designs analysiert und miteinander verglichen. Die Tests wurden auf folgenden Schaltungen aus unterschiedlichen Kategorien durchgeführt:

2Bit-Adder ist ein 2-Bit-Volladdierer als Referenzschaltung (A.1). Dieser wurde an vielen Stellen zum Veranschaulichen verschiedener Design-Schritte verwendet. Zusätzlich wurde der HDL-Code mit Synthese-Attributen zur Absicherung der Design-Hierarchie und zur Verhinderung der Optimierung auf Signalen erweitert.

PicoBlaze Der PicoBlaze[1] ist ein 8-Bit Mikrocontroller der Firma Xilinx und ist in VHDL (bzw. Verilog) implementiert. Er deckt ein großes Spektrum der FPGA-Ressourcen (wie LUTs, Flip-Flops, Speichermodule, etc.) ab und wurde in einer kleinen Schaltung zum Ein- und Ausschalten der LEDS auf dem Zedboard eingebunden. Diese Schaltung besteht ausschließlich aus Xilinx-Primitiven und liegt als Strukturbeschreibung vor.

S27 Diese Schaltung wurde aus der ISCAS'85-Benchmark-Schaltungen[2] ausgewählt und wie folgt angepasst:

- Da das Verilog-Primitiv `nmos` im Modul `ddf` nicht synthetisierbar ist, wurde das Modul wie in [Pet12] geändert.
- Die logischen Gatter (AND, OR, NAND, NOR, NOT) wurden wie folgt am Beispiel des `NAND`-Gatters durch eigene Implementierung ersetzt:

```
1 module _NAND2 (O, I1, I2);
2     input I1, I2;
3     output O;
4
5     (* DONT_TOUCH = "yes" *) wire d;
6
7     assign d = I1 & I2;
8     assign O = (*@ $\sim$*)d;
9 endmodule
```

[1] http://www.xilinx.com/products/intellectual-property/picoblaze.html
[2] http://www.pld.ttu.ee/~maksim/benchmarks/iscas89/verilog

```
10
11 (* DONT_TOUCH = "yes" *) module _NOT (O, I);
12     input I;
13     output O;
14
15     assign O = (*@ $\sim$*)I;
16 endmodule
17 ...
```

Listing 6.1: Eigene Implementierung des logischen `NAND`-Gatters

Ohne `DONT_TOUCH`-Attribut wird das Signal `d` durch Synthese-Tools bei der Optimierung aus dem Design entfernt. Außerdem verhindert diese Anweisung die Wegoptimierung der `not`-Module während der Implementierung. Damit wird das Zusammenführen des `and`- und `not`-Gatters in einem LUT verhindert und bleiben im FPGA-Design erhalten. Diese Schaltung besteht ausschließlich aus Flip-Flops und logischen Gattern.

UART Diese Schaltung wurde aus den Xilinx Workshop-Dateien ausgewählt. Sie empfängt User-Eingaben per Drücken des Push-Buttons auf dem Zed-Board und schaltet entsprechend die obere oder untere Hälfte der LEDs ein. Diese Schaltung besteht aus einer großen Anzahl von Multiplexern, Register und arithmetischen Operationen, die ausschließlich aus der Verhaltensbeschreibung generiert werden. Sie wurde zur Veranschaulichung des Worst-Case beim Mapping ausgewählt.

6.1 Testumgebung

Die Messungen wurden auf dem folgenden Rechner durchgeführt:
AMD FX-6100 Six-Core Processor, 3.30 GHz
4.00 GB Arbeitspeicher
Bei der Implementierung und den Tests wurde Vivado 2014.4 genutzt.

6.2 Abdeckungsgrad des Mappings

In diesem Abschnitt wird das Ergebnis des implementierten Mapping-Prozesses dargestellt. Das Mapping ermöglicht die Referenzierung der Input-Objekte der Schaltung zur Fehlerinjektion im FPGA-Design. Dabei basiert diese Referenzierung auf die Selektion der Design-Objekte in der RTL-Netzliste. Es können nicht immer alle selektierten Elemente der RTL-Netzliste (wie z.B bei Multiplexern oder bei arithmetischen Operatoren der Fall ist) im FPGA-Design

gefunden werden, da einige dieser Elemente nach der Synthese (bzw. Implementierung) entfernt oder auf ein einziges Bauteil der Zielarchitektur (wie DSP48-Instanz) abgebildet werden. In diesem Fall können keine MBF-basierte Defekte an diese Bauteile zugewiesen werden. Deshalb ist das Mapping eine Kennzahl zur Abdeckung der implementierten Verfahren der Fehlerinjektion. Der Abdeckungsgrad des Mappings lässt sich durch das Verhältnis der Anzahl der referenzierbaren Leaf-Pins im FPGA-Design zur Gesamtanzahl der selektierbaren Leaf-Pins in der RTL-Netzliste bestimmen.

Tabelle 6.1 stellt die Ergebnisse der ausgewählten Testschaltungen dar. Dabei wurden die Ergebnisse für Ports, Modul-Pins und Leaf-Pins separat aufgelistet. Bei der Fehlerinjektion werden Leaf-Zellen abhängig von den ausgewählten Stuck-at-Fehler modifiziert. Aus diesem Grund werden bei selektierten Ports oder Modul-Pins nicht die Anzahl der gefundenen Ports oder Modul-Pins betrachtet, sondern die Anzahl der Pins der Leaf-Zellen, welche durch ein Netz mit diesen Objekten verbunden sind. Die folgenden Abkürzungen in der Tabelle stehen für:

P Die Anzahl der vorhandenen Input-Ports der Top-Level-Moduls oder -Pins der vorhanden Module (bzw. Primitiven) im Design.

S Die Anzahl der über Ports (bzw. Modul-Pins) oder direkt selektierbaren Leaf-Pins in der RTL-Netzliste. Die Clock-Pins und die Leaf-Pins die durch VCC oder GND getrieben werden, werden bei implementierten Verfahren nicht injizierbar. Daher wurden diese von der Selektion zur Fehlerzuweisung ausgeschlossen.

F Die Anzahl die durch Mapping referenzierbare Leaf-Pins im FPGA-Design.

Der Test für Mapping wurde nur auf der implementierten Netzliste ausgeführt. Die Ergebnisse auf der Post-Synthesis-Netzliste sind identisch mit den ermittelten Werten.

Tabelle 6.1: Die Ergebnisse des Mappings bei ausgewählten Testschaltungen

Schaltung	**Ports**			**Modul-Pins**			**Leaf-Pins**		
	P	**S**	**F**	**P**	**S**	**F**	**P**	**S**	**F**
Adder-2Bit	5	10	10	14	14	14	22	22	22
S27	7	4	4	24	21	21	29	26	26
PicoBlaze	29	340	340	474	420	420	1623	486	486
UART	3	3	3	17	58	58	468	417	155

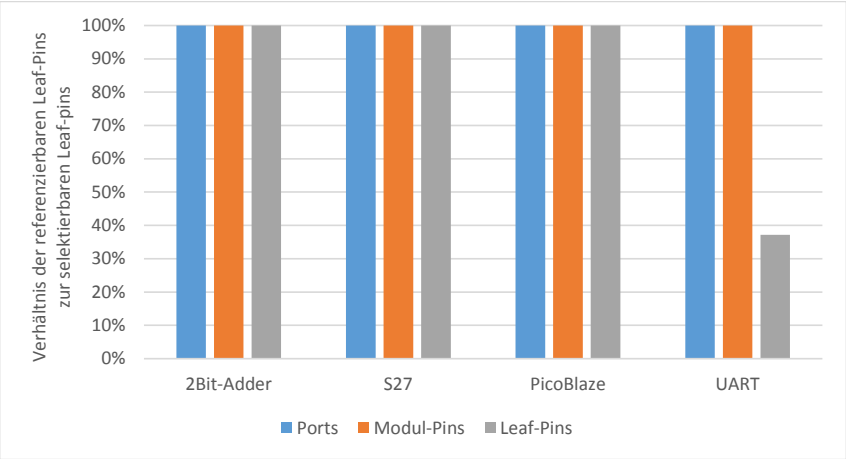

Abbildung 6.1: Darstellung der Ergebnisse des Mappings

6.2.1 Diskussion der Ergebnisse

Falls die synthetisierte oder implementierte Netzliste aus der RTL-Netzliste eins zu eins abgebildet wird, d.h. wenn alle Optimierungen ausgeschaltet sind und alle Zellen der RTL-Netzliste im FPGA-Design vorhanden sind, funktioniert das Mapping vollständig.

Wie schon aus den Ergebnissen ersichtlich ist, wurden bei den Schaltungen (2Bit-Adder und S27), die vollständig mit den benötigten Syntheseoptionen annotiert wurden, ein hundertprozentiges Mapping erreicht. Auch bei Schaltungen (z.B. PicoBlaze), die ausschließlich durch Instanziierung von Xilinx-Primitiven bestehen, d.h. strukturell aufgebaut sind, wird ein vollständiges Mapping erreicht. PicoBlaze liegt als Strukturbeschreibung vor und besteht ausschließlich aus den Xilinx-Primitiven des Ziel-FPGAs.

Wie schon bei der Schaltung UART der Fall ist, sinkt die Anzahl referenzierbarer Objekte erheblich, falls viele Komponenten der RTL-Netzliste auf einem einzigen Bauteil des Ziel-FPGAs abgebildet bzw. wegoptimiert werden. UART liegt als Verhaltensbeschreibung vor und viele Elemente wie z.B. Multiplexer der generischen Netzliste der UART-Schaltung werden aus den *always*- oder *if*-Blöcken generiert. Diese Elemente werden durch das Synthese-Tool auf eine LUT-Instanz des Ziel-FPGAs abgebildet.

Wird die Hierarchie der Schaltung durch Optimierungsattribute gesichert, so bleiben alle Module in der RTL-Netzliste nach Synthese (bzw. Implementierung) im FPGA-Design erhalten. Da die Namen der Modul-Pins in der RTL-Netzliste und FPGA-Design äquivalent sind, kann man dies ausnutzen, um bei Leaf-Pins eine hundertprozentige Mapping zu erreichen. Beispielsweise können bei der UART-Schaltung die nicht durch Mapping referenzierbaren Primitive in eigene Module eingekapselt werden. Die Selektion kann dann bei diesen Bauteilen an Modul-Pins vorgenommen werden, bei denen im allgemeinen ein

vollständiges Mapping erreicht wird.

6.3 Ausführungszeiten

Wie schon in den vorherigen Kapiteln detailliert beschrieben wurde, erweitert die implementierte Methode der deterministische Vivado-Designflow um die Selektion der Input-Objekte in der RTL-Netzliste und die Injektion von Fehlern im FPGA-Design. Dabei kann das fehlerbehaftete Design entweder mittels Simulation oder Emulation analysiert werden.

Sowohl Fehlersimulation als auch Fehleremulation bestehen aus gleich ablaufenden Teilschritten. Abhängig von der ausgewählten Fehlerinjektion-Phase (Post-Synthesis oder Post-Implementation) wird im Anschluss der Fehlerinjektion entweder die Simulation oder die Emulation gestartet. Die Tabellen 6.2, 6.3, 6.4 und 6.5 zeigen die ermittelten Ausführungszeiten (in Millisekunden) der einzelnen Methoden (Fehlersimulation und Fehleremulation) für jede dieser Phasen.

Bei dem Test wurden die Testschaltungen 2Bit-Adder und PicoBlaze ausgewählt. Dabei wurde die Ausführungszeit jeder Methode mit einer unterschiedlichen Anzahl von *Stuck-at-1* Fehlern gemessen. Die Fehler wurden in einem Ablauf parallel injiziert. Bei einer sequentiellen Fehlerinjektion addieren sich diese Zeiten auf und machen bei der Auswertung der getesteten Methoden keinen großen Unterschied. Bei der Referenzschaltung wurden die Fehler an den Eingängen der logischen Gatter zugewiesen. Bei PicoBlaze wurden die Fehler an den Komponenten wie Flop-Flops, Gatter und Block-RAMs verteilt zugewiesen. Beim Test mit einem einzelnen Stuck-at-Fehler wurde ein Flip-Flop ausgewählt.

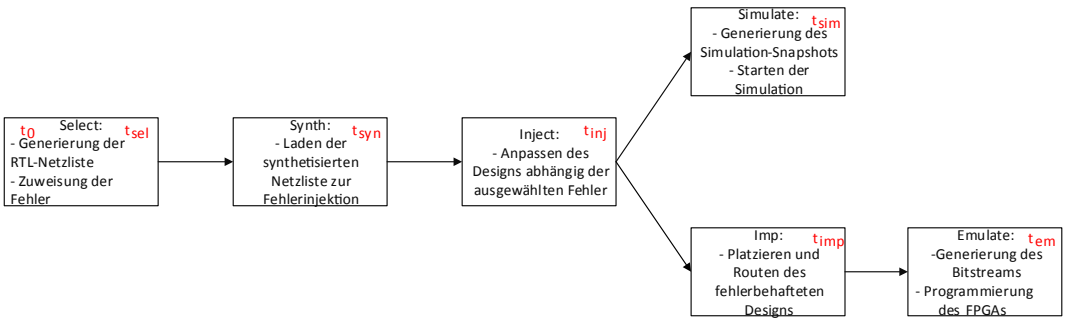

Abbildung 6.2: Ablauf der Fehlerinjektion in Post-Synthesis-Phase

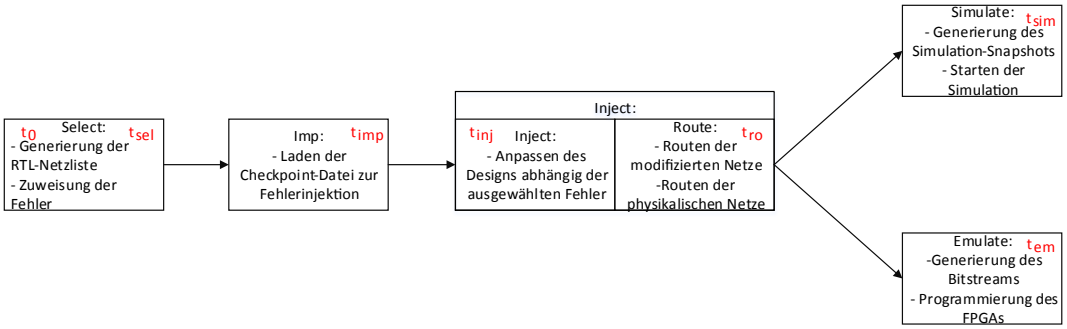

Abbildung 6.3: Ablauf der Fehlerinjektion in Post-Implementation-Phase

Wie in der Abbildung 5.3 zu sehen ist, besteht der Ablauf der Fehlerinjektion aus unterschiedlichen Teilschritten. Bei dem Test wurde die Laufzeit für alle beteiligten Prozesse gemessen, die in diesen Teilschritten durchgeführt werden. Sowohl Fehlerinjektion als auch Fehleremulation bestehen dabei aus gleich ablaufenden Schritten. Diese sind unter anderem Selektion der Input-Objekte zur Fehlerinjektion, Anpassung des FPGA-Designs abhängig vom ausgewählten Fehlertyp, sowie Laden des zuvor synthetisierten (bzw. implementierten) Designs. Die Abbildung 6.2 stellt diesen Ablauf für die Post-Synthesis Phase exemplarisch dar. Die Fehlerinjektion in der Post-Implementation Phase läuft hingegen mit *postImpl* beschrifteten Pfaden des Fehlerablauf-Automaten (siehe 5.3) durch und wird in Abbildung 6.3 dargestellt.

In der obigen Abbildung sind noch die Zeitpunkte der Messungen für jeden durchlaufenen Teilschritt gekennzeichnet. Sowohl die Laufzeiten einzelner Phasen als auch die Methoden Simulation und Emulation wurden folgendermaßen abgekürzt:

S Laufzeiten der Fehlerinjektion für Simulationsmethode.

E Laufzeiten der Fehlerinjektion für Emulationsmethode.

t_{sel} Laufzeit für Generierung der RTL-Netzliste und Zuweisung von Injektoren an Input-Ports oder -Pins.
Ausgeführte Prozesse: *synth_design -rtl*, *inject_fault*

t_{syn} Laufzeit für das Öffnen des zuvor synthetisierten Designs.
Ausgeführte Prozesse: *read_edif*, *link_design*

t_{imp} Laufzeit für das Öffnen des zuvor implementierten Designs.
Ausgeführte Prozesse: *read_checkpoint*, *link_design*

t_{inj} Laufzeit für die Anpassung des FPGA-Designs abhängig von dem ausgewählten Stuck-at-Fehler.
Ausgeführte Prozesse: *mapping*, *modify_netlist*

t_{ro} Laufzeit für das Routen der modifizierten und physikalischen („logic one“ und „logic zero“) Netze.
Ausgeführte Prozesse: *route_design -nets <nets_name>*, *route_design -physical_nets*

t_{sim} Laufzeit für das Starten der Simulation.
Ausgeführte Prozessen: *write_vhdl -mode funcsim*, *xelab*, *xsim*

t_{em} Laufzeit für das Starten der Emulation.
Ausgeführte Prozesse: *write_bitstream*, *program_hw_devices*

Es ist zu beachten, dass die Messungen bis zum Zeitpunkt des Simulations- bzw. Emulations-Starts durchgeführt wurden. D.h., es wurden keine Laufzeiten vom Ablauf der Simulation bzw. Emulation ermittelt, um die beiden Methoden gegenüberzustellen. Die Messungen dienen nur zur Bewertung des implementierten Verfahrens abhängig von der selektierten Phase und der eingesetzten Methode der Fehlerinjektion.

Folgende Tabellen zeigen die Messungen der einzelnen Prozesse für jede Methode und Phase des Ablaufs der Fehlerinjektion:

Tabelle 6.2: Ausführungszeiten der Post-Synthesis-Simulation (in Millisekunden)

Schaltung		t_{sel}	t_{syn}	t_{inj}	t_{sim}	t_{sum}
2Bit-Adder	1	18238	1807	81	26550	46676
	5	16977	1838	420	40748	59983
	10	16525	1890	1479	26254	46148
	20	18155	1868	6005	26383	52411
PicoBlaze	1	18099	2110	77	31821	52107
	5	16994	1896	474	36224	55588
	10	17450	1892	1524	25398	46264
	20	16748	1804	6155	25181	49888

Tabelle 6.3: Ausführungszeiten der Post-Synthesis-Emulation (in Millisekunden)

Schaltung		$\mathbf{t_{sel}}$	$\mathbf{t_{syn}}$	$\mathbf{t_{inj}}$	$\mathbf{t_{imp}}$	$\mathbf{t_{em}}$	$\mathbf{t_{sum}}$
2Bit-Adder	1	17854	1896	74	56322	71972	148044
	5	17883	1872	572	59183	68691	147629
	10	17982	1872	1661	57968	68047	145870
	20	16285	1764	5552	56258	69805	144113
PicoBlaze	1	16447	1868	252	52580	63640	134537
	5	16329	1777	464	52661	63447	134214
	10	16432	1867	1535	53492	64304	136095
	20	16760	1921	6953	58602	65341	142625

Tabelle 6.4: Ausführungszeiten der Post-Implementation-Simulation (in Millisekunden)

Schaltung		$\mathbf{t_{sel}}$	$\mathbf{t_{imp}}$	$\mathbf{t_{inj}}$	$\mathbf{t_{ro}}$	$\mathbf{t_{sim}}$	$\mathbf{t_{sum}}$
2Bit-Adder	1	17152	2008	78	82120	29493	130851
	5	17320	2094	508	88598	31872	140392
	10	18303	2086	1761	83557	27190	132897
	20	18291	2054	6103	82363	30592	139403
PicoBlaze	1	18440	2327	71	96743	31225	148806
	5	18707	2377	527	96897	30625	149133
	10	18672	2336	1661	96875	33723	153267
	20	18672	2333	6922	102480	38355	168762

Tabelle 6.5: Ausführungszeiten der Post-Implementation-Emulation (in Millisekunden)

Schaltung		$\mathbf{t_{sel}}$	$\mathbf{t_{imp}}$	$\mathbf{t_{inj}}$	$\mathbf{t_{ro}}$	$\mathbf{t_{em}}$	$\mathbf{t_{sum}}$
2Bit-Adder	1	17865	2393	80	79408	24208	123954
	5	16574	2302	398	76896	25543	121714
	10	16955	2365	1909	45070	21994	136791
	20	15443	1772	5300	69883	21345	113744
PicoBlaze	1	15593	2076	62	86571	61695	165998
	5	17334	2255	471	91657	22660	134378
	10	18670	2526	1602	97959	22836	143593
	20	18029	2258	6848	97107	23401	147643

Tabelle 6.6 enthält die Laufzeiten der Prozesse wie Synthese, Implementie-

rung und Simulation des Vivado Designflows. Der Vivado reguläre Designflow duchläuft diese Prozesse für jeden zu testenden Fehler, wie dies auch bei der generischen Serial Fault Emulation der Fall ist.

Tabelle 6.6: Ausführungszeiten des regulären Vivado Designflows

Schaltung		$\mathbf{t_{syn}}$	$\mathbf{t_{sim}}$	$\mathbf{t_{imp}}$	$\mathbf{t_{sim}}$	$\mathbf{t_{sum}}$
2Bit-Adder	Synth	45100	34113	...	...	79213
	Impl	41198	...	87794	25757	154749
PicoBlaze	Synth	44131	29209	...	...	73340
	Impl	43328	...	88543	30782	162653

6.3.1 Diskussion der Ergebnisse

Die Messwerte aus den obigen Tabellen zeigen, dass die Geschwindigkeit des Ablaufes der Fehlerinjektion fast doppelt so hoch ist als bei dem regulären Vivado Designflow. Bei PicoBlaze ist der Geschwindigkeitszuwachs geringer als beim 2Bit-Adder, da bei dieser Schaltung das rechenintensive partielle Routing ein zeitbegrenzender Faktor ist.

Im allgemeinen ist aber bei größeren Schaltungen ein größerer Geschwindigkeitsgewinn zu erwarten. Im Gegensatz zu der generischen Serial Fault Emulation, bei den die Synthese, Platzierung und Routing für jede zu testenden Fehler durchgeführt wird, erreicht das implementierte Verfahren durch das Laden der synthetisierten Netzliste- bzw. implementierten Checkpoint-Datei geringere Ausführungszeiten. Diese Dateien werden zu Beginn der Ausführung einmalig in den Hauptspeicher geladen. Das Wechseln zwischen RTL und des zuvor geladenen Designs erfolgt dann sehr schnell für jeden zu testenden Fehler. Des weiterhin nimmt das Starten des RTL-Designs (t_{sel}) zur Selektion der Input-Objekte einen großen Anteil bei der ermittelten Gesamtlaufzeit ein, welches auch nur einmalig ausgeführt wird und für die restlichen zu testenden Fehler vernachlässigbar ist. Um eine grobe Abschätzung zu erhalten, wurden die Messwerte für den regulären Designflow von Vivado nur mit einem einzigen Stuck-at-Fehler ermittelt. Daher sind die oben benannten Zeitfaktoren in den Messwerten, sowie der erhebliche Geschwindigkeitszuwachs bei vorliegender Implementierung nicht sichtbar.

Darüber hinaus sind die Generierung des Bitstreams (bzw. Simulation-Snapshot) und das partielle Routing in Post-Implementation-Phase bei der entwickelten Methode zeitbestimmende Größen. Wird die emulationsbasierte Fehlerinjektion in Post-Synthesis-Phase durchgeführt und das Design jedes mal platziert und geroutet, so ergibt sich weiterhin einen zeitlichen Vorteil gegenüber der generischen Serial Fault Emulation. Wie es in Abbildung 6.4 zu sehen ist hängt die Größe der Ausführungszeit einerseits von der angewendeten

Methode, andererseits von der eingesetzten Phase der Fehlerinjektion ab. Die folgende Abbildung vergleicht sowohl diese Methoden als auch die Phase der Fehlerinjektion miteinander.

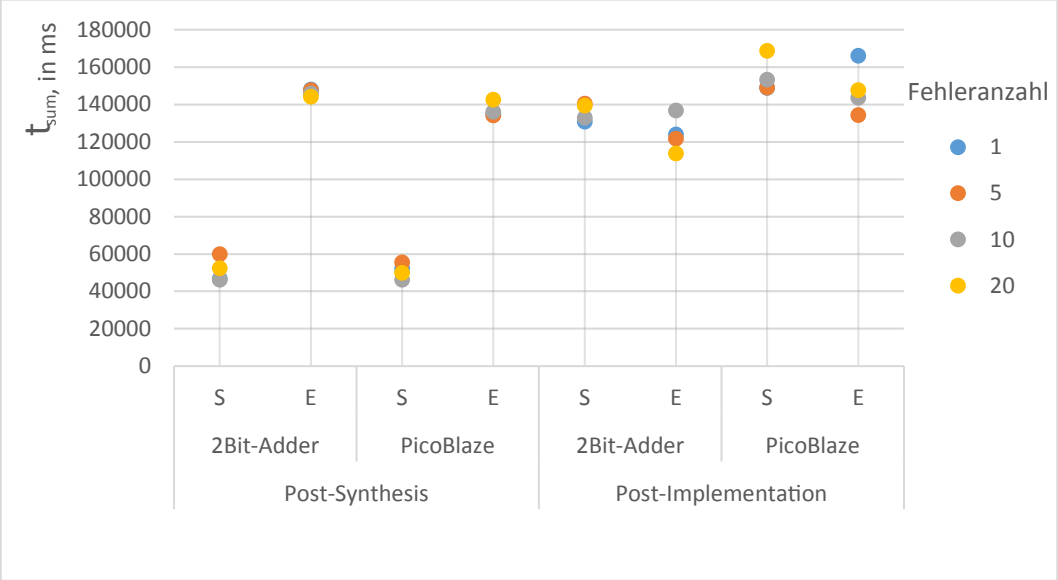

Abbildung 6.4: Zeitverhalten der Fehlersimulation und der Fehleremulation in unterschiedlichen Phasen des Designflows

Aus Abbildung 6.4 geht hervor, dass die Laufzeiten der eingesetzten Methoden sehr unterschiedlich sind, falls die Injektion von Fehlern in unterschiedlichen Phasen stattfindet. Unabhängig von der zu testenden Schaltung hat die Fehlersimulation einen großen Vorteil, falls diese in der Post-Synthesis-Phase eingesetzt wird. In der Post-Synthesis-Phase ist die synthetisierte Netzliste direkt simulierbar im Gegensatz zur Fehleremulation in dieser Phase, da die Schritte wie Platzierung und Verdrahtung nicht durchgelaufen werden müssen.

Deutlich ist ein Unterschied der Zeiten für den 2Bit-Adder und dem PicoBlaze in der Post-Implementation-Phase zu erkennen. Der Grund dafür ist, dass der 2Bit-Adder ausschließlich aus den logischen Gattern besteht und die Fehlerinjektion im FPGA-Design durch das Ändern des INIT-Attributs der LUTs vorgenommen wird. Somit wird kein Routing durchgeführt. Beim PicoBlaze werden die Fehler in unterschiedlichen Komponenten injiziert. Die injizierenden Pins dieser Komponente werden bei der Modifizierung mittels globalen Netzen durch die logische Konstante getrieben. Dabei werden die bestehenden Netze dieser Pins gelöscht, wenn sie keine Pins mehr treiben. Andernfalls werden sie modifiziert. Daher müssen diese und die angelegten globalen Netze nach der Anpassung des FPGA-Designs zur Fehlerinjektion neu geroutet werden. Somit nimmt die Ausführungszeit bei dieser Schaltung mit ansteigender Anzahl der injizierenden Fehlern zu.

Einen Vergleich der Platzierung, dem Routing und dem partiellen Routing der Netze des entwickelnden Designs zeigt die Abbildung 6.5. Das Routen der modifizierten, sowie neu angelegten physikalischen Netze für logische „1“ oder „0“ dauert länger als das Platzieren und Routen des gesamten Designs. Dies liegt wahrscheinlich an den internen Algorithmen des Place- und Route-Werkzeugs von Vivado.

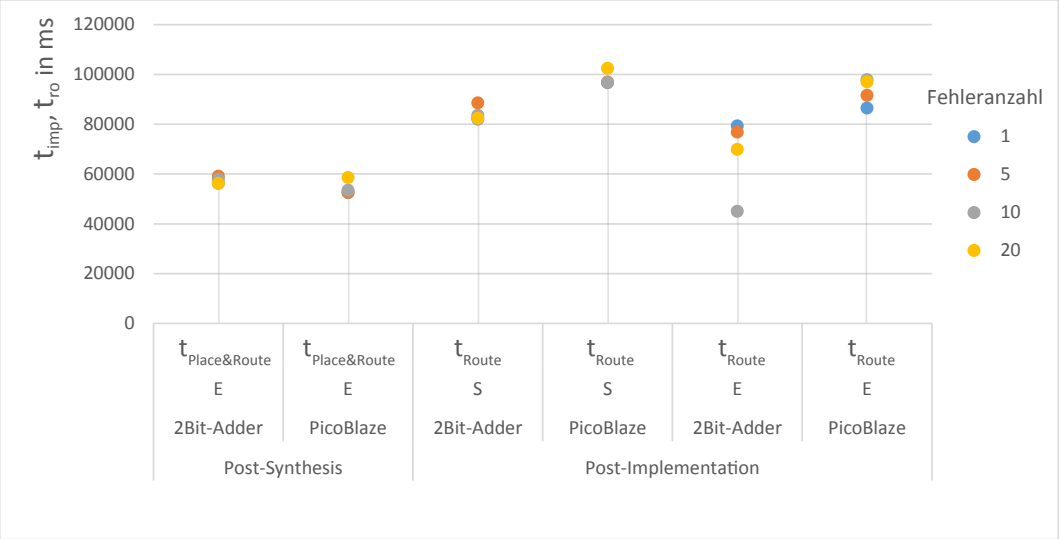

Abbildung 6.5: Vergleich der Laufzeiten der Platzierung, Verdrahtung und des partiellen Routings

Anschließend zeigt die nachfolgende Abbildung einen Vergleich der Laufzeiten für Simulation und Emulation.

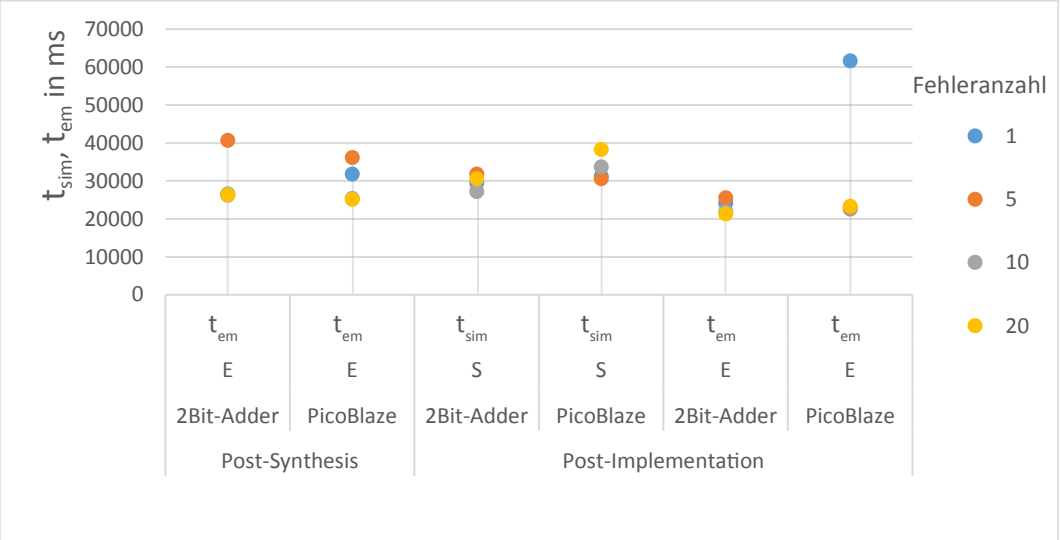

Abbildung 6.6: Vergleich der Laufzeiten der Simulation und Emulation

Die Messungen zeigen, dass das Starten der Simulation genauso lange wie die Generierung des Bitstreams und der Konfigurierung des FPGAs dauert. Wie oben schon angesprochen, wurden die Messungen an dem Zeitpunkt durchgeführt, bei dem die Simulation bzw. Emulation startbereit war. Da die Implementierung für das Anlegen der Testvektoren bei der Fehleremulation fehlt, konnten keine Vergleiche der Laufzeiten für die Durchführung der Simulation und Emulation mit den Testvektoren ermittelt werden.

7 Ausblick

Die Vivado Design Suite ist eine leistungsfähige Entwicklungsumgebung für die Xilinx-FPGAs und -SoCs. Diese neue Designumgebung basiert auf der mächtigen Skriptsprache Tcl, die als Industriestandard bei vielen EDA-Tools eingesetzt wird. Das Ziel dieser Arbeit bestand in der Implementierung eines Tcl-Tools zur Injektion permanenten Fehler in der Vivado-Designumgebung. Dabei war der Ausgangspunkt der in [Pet12] detailliert beschriebene und als eine Java-Bibliothek implementierte Ansatz.

Die entwickelte Tcl-App ermöglicht die Stuck-at-Fehler in beliebige digitale Schaltungen zu injizieren.

Basierend auf dem Vivado Designflow kann dabei die Fehlerinjektion in unterschiedlichen Entwicklungsphasen stattfinden. Beispielsweise können Defekte im vollständig platzierten und gerouteten Design injiziert und das fehlerbehaftete Design anschließend auf einem FPGA emuliert werden. Die vorliegende Implementierung hat außerdem den Vorteil der Fehlersimulation innerhalb der Vivado-Designumgebung. Ein FPGA-Design mit einem Stuck-at-Fehler kann neben der Emulation sowohl nach der Synthese als auch nach der Implementierung in Vivado simuliert werden. Die vorliegende Implementierung ist gekennzeichnet durch die zielgenaue Kombination beider genannten Methoden.

Trotz des Geschwindigkeitszuwachses des implementierten Verfahrens bestehen noch weitere Möglichkeiten der Beschleunigung der Tcl-basierten Methode. Zum einen kann nach einer genaueren Performanzanalyse die Fehlerinjektion in die Post-Implementierung-Phase vorverlegt werden, um den Nachteil des zeitaufwendigen partiellen Routings zu umgehen. Da die Implementierung in zwei Schritten aus Platzierung und Routing besteht, kann die Fehlerinjektion beispielsweise in der Post-Platzierung-Phase stattfinden und das modifizierte Design komplett geroutet werden. Somit hat das Routing-Tool mehr Freiheitsgrade im Vergleich zu einem partiellen Routing. Alternativ zu den physikalischen Netzen können die logischen Konstanten 1 und 0 für die Stuck-at-Fehler durch die LUT1-Instanzen generiert werden.

Zum anderen kann die Programmierung des FPGAs durch eine partielle Rekonfiguration beschleunigt werden. Die partielle Rekonfiguration ermöglicht die dynamische Änderung der Komponenten eines Designs zur Laufzeit. Die Rekonfiguration kann dabei durch ein partielles Bitfile vorgenommen werden, anstatt das gesamte Bitfile für jeden zu injizierenden Fehler neu auf das FPGA zu übertragen. Das Datenblatt [Xil15] von Xilinx beschreibt den Ablauf einer partiellen Rekonfiguration.

Die Implementierung lässt sich weiterhin auf neue Funktionen erweitern. Die Fehlerinjektion bei Block-RAMs kann beispielsweise auf die Möglichkeit zur Fehlerinjektion bei Bit-Flips erweitert werden. Darüber hinaus ist eine

Verbesserung des Abdeckungsgrades beim Mapping für die primitiven Zellen möglich. Bei der Referenzierung dieser Objekte können noch andere Properties wie der Typ dieser Objekte in Betracht gezogen werden. Damit kann beim Mapping gezielt nach Elementen im FPGA-Design gesucht werden. Dafür ist aber eine genauere Studie des Synthese-Tools von Vivado erforderlich, um die Abbildung zwischen den Elementen der generischen Netzliste und der FPGA-Netzliste zuordnen zu können.

Letztendlich wäre eine weitere Erweiterung die automatisierte Generierung der Testvektoren für zu testende Module bei der Fehlersimulation und der Fehleremulation. Eine Tcl-App namens Design-Utilities, welche als Open-Source Anwendung über den Xilinx-Tcl-Store in Vivado zugänglich ist, enthält bereits ein Tcl-Skript zur Erstellung von Testbenches für VHDL und Verilog. Dieses Skript erzeugt nur das Grundgerüst einer Testbench. Jedoch kann es als Grundlage für die Generierung der Testvektoren bei der Fehlersimulation erweitert werden. Die Generierung der Testvektoren für die Fehleremulation kann mittels der Xilinx Zynq-7000-Plattform realisiert werden, welche neben der programmierbaren Logik noch einen Dual-Core-ARM-Prozessor enthält.

Literaturverzeichnis

[BRF+96] Luc Burgun, Frédéric Reblewski, Gérard Fenelon, Jean Barbier, and Olivier Lepape. Serial fault emulation. In *DAC*, pages 801–806, 1996.

[CHD99] Kwang-Ting Cheng, Shi-Yu Huang, and Wei-Jin Dai. Fault emulation: A new methodology for fault grading. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 18(10):1487–1495, 1999.

[DN11] Carson Dunbar and Kundan Nepal. Using platform fpgas for fault emulation and test-set generation to detect stuck-at faults. *Journal of Computers*, 6(11), 2011.

[Hop99] Bernhard Hoppe. *ASIC-Design : Realisierung von VLSI-Systemen mit Mentor V8*. Springer, 1999.

[JDR09] M. Jeitler, M. Delvai, and S. Reichor. Fuse - a hardware accelerated hdl fault injection tool. In *Programmable Logic, 2009. SPL. 5th Southern Conference on*, pages 89–94, April 2009.

[LB13] W. Lange and M. Bogdan. *Entwurf und Synthese von Eingebetteten Systemen: Ein Lehrbuch*. Oldenbourg Wissenschaftsverlag, 2013.

[Pet12] Dustin Peterson. Entwicklung und Evaluierung eines Systems zur Injektion permanenter Fehler in FPGA-Schaltungen. Master's thesis, Universität Tübingen, sep 2012.

[Woj88] H. Wojtkowiak. *Test und Testbarkeit digitaler Schaltungen*. Viweg+Teubner Verlag, 1988.

[Xil12] Xilinx. *White Paper: Vivado Design Suite: WP416 (v1.1)*, jun 2012.

[Xil13] Xilinx. *XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices UG687 (v 14.5)*, mar 2013.

[Xil14a] Xilinx. *7 Series FPGAs Configurable Logic Block User Guide UG474 (v1.7)*, nov 2014.

[Xil14b] Xilinx. *7 Series FPGAs Memory Resources User Guide UG473 (v1.11)*, nov 2014.

[Xil14c] Xilinx. *7 Series FPGAs Overview DS180 (v1.16.1)*, dez 2014.

[Xil14d] Xilinx. *Vivado Design Suite 7 Series FPGA and Zynq-7000 All Programmable SoC Libraries Guide UG953 (v2014.3)*, oct 2014.

[Xil14e] Xilinx. *Vivado Design Suite Tutorial: Hierarchical Design UG946 (v2013.4)*, dez 2014.

[Xil14f] Xilinx. *Vivado Design Suite User Guide: Design Flows Overview UG892 (v2014.3)*, oct 2014.

[Xil14g] Xilinx. *Vivado Design Suite User Guide: Implementation UG904 (v2014.3)*, oct 2014.

[Xil14h] Xilinx. *Vivado Design Suite User Guide Logic Simulation UG900 (v2014.3)*, oct 2014.

[Xil14i] Xilinx. *Vivado Design Suite User Guide: Synthesis UG901 (v2014.3)*, oct 2014.

[Xil14j] Xilinx. *Vivado Design Suite User Guide: Using Tcl Scripting UG894 (v2014.3)*, oct 2014.

[Xil15] Xilinx. *Vivado Design Suite User Guide Partial Reconfiguration UG909 (v2015.1)*, apr 2015.

A Referenzschaltung

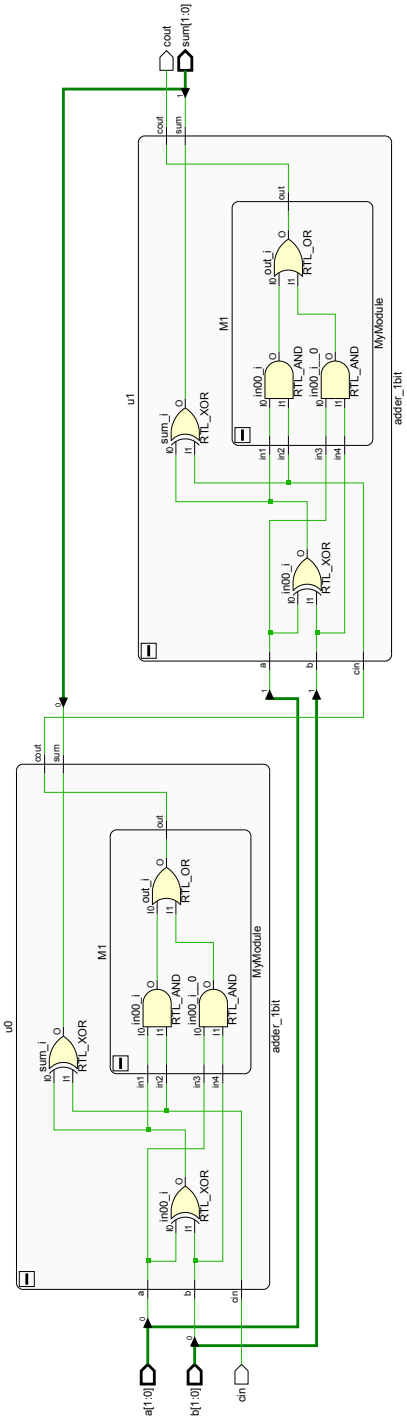

Abbildung A.1: Schaltplan eines 2-Bit-Addierers

A.1 2-Bit-Addierer in Verilog

```
1  `timescale 1ns / 1ps
2  //----------------------------------------------------
3  //
4  //  Module Name  : adder_2bit
5  //  Parent   : None
6  //  Children : adder_1bit
7  //
8  //  Description:
9  //     2-Bit-Volladdierer
10 //
11
12 module adder_2bit(
13     input [1:0] a,
14     input [1:0] b,
15     input cin,
16     output [1:0] sum,
17     output cout
18     );
19
20     wire [1:0] cvect;
21
22     adder_1bit u0(a[0], b[0], cin, sum[0], cvect[0]);
23     adder_1bit u1(a[1], b[1], cvect[0], sum[1], cvect[1])
          ;
24
25     assign cout = cvect[1];
26 endmodule
```

Listing A.1: Modulbeschreibung eines 2-Bit-Addierers

```
1  `timescale 1ns / 1ps
2  //----------------------------------------------------
3  //
4  //  Module Name : adder_1bit
5  //  Parent      : adder_1bit
6  //  Children    : None
7  //
8  //  Description:
9  //     1-Bit-Volladdierer
10 //
11
12
13 module adder_1bit(
14     input a,
```

```
15      input b,
16      input cin,
17      output sum,
18      output cout
19      );
20
21      (* dont_touch = "yes" *) wire w1;
22
23      assign w1 = a ^ b;
24      assign sum = w1 ^ cin;
25
26     MyModule M1(w1, cin, a, b, cout);
27
28 endmodule
29
30  module MyModule(
31      input in1,
32      input in2,
33      input in3,
34      input in4,
35      output out
36      );
37
38      (* dont_touch = "yes" *) wire n1;
39      (* dont_touch = "yes" *) wire n2;
40
41      and (n1, in1, in2);
42      and (n2, in3, in4);
43      or (out, n1, n2);
44
45 endmodule
```

Listing A.2: Strukturbeschreibung eines Addierers

```
1 `timescale 1ns / 1ps
2
3 module stim (A, B, clk, c_stim_in, s_in, c_in);
4    output   [1:0] A, B;
5    input c_stim_in, clk, c_in; // Takt
6    input [1:0]s_in;
7    integer i, j; // Laufindizes
8
9    reg [1:0] A, B;
10
11   wire [1:0]test_sum;
12   wire test_carry;
13
```

```
14    assign test_clk = ~clk; // inverser Takt
15    assign {test_carry,test_sum} = A+B+c_stim_in;
16
17    // Berechnung der Erwartungswerte
18    initial
19    begin // Anfangswerte
20       A = 0;
21       B = 0;
22    end
23
24    always // Doppelschleife fuer Eingabemuster
25    begin
26        for (i=0;  i < 8; i = i+1)
27       begin
28            for (j=0;  j < 8; j = j+1)
29                @(posedge clk)
30             // Ausgabeaktualisierung
31             begin
32                A =  i;
33                B =  j;
34             end
35          @(posedge test_clk)
36          begin  // Ueberpruefung der Ergebnisse
37             if ((test_sum - s_in)!= 0)
38             begin
39                $display ("Expected_Sum = %d Actual_Sum =%
                     d\n",test_sum,s_in);
40                $display ("Error in Sum");
41                $finish; // Im Fehlerfall wird Test
                     beendet
42             end
43               if ((test_carry - c_in)!= 0)
44             begin
45                $display ("Error in Carry");
46                $display ("Expected_carry = %d Test_Carry
                     =%d\n",test_carry,c_in);
47                $finish; // Im Fehlerfall wird Test
                     beendet
48                end
49           end
50       end
51        $display("TEST OK"); // Test durchgelaufen
52       $finish;
53    end
54
55 endmodule
```

```
56
57 module adder_2bit_tb (Sum, Cout);
58    output   [1:0] Sum;
59    output Cout;
60
61    wire [1:0] A, B;
62    reg Cin, clk;
63
64    initial
65    begin
66       Cin = 0;
67       clk = 0;
68    end
69
70    // Frequenzhalbierung
71    always    #5 clk = ~clk;
72
73    always @(posedge clk)
74    begin
75       @(posedge clk)
76       Cin = ~Cin;
77    end
78
79    // Instanziiere Addierer und Stimuli-Erzeugung
80    adder_2bit In1 (A, B, Cin, Sum, Cout);
81    // implizite Liste
82    stim In2 (A, B, clk, Cin, Sum, Cout);
83
84 endmodule
```

Listing A.3: Testbench für 2-Bit-Addierer

A.2 Tcl-Code des Ablaufs der Fehlerinjektion

```
array set faultFlow {

    select {# STEP#1: Open elaborated rtl design to
       select any input pin or port
        puts "\n*** STEP: select ***"
        puts "====================\n"

        if {[string equal $phase "synth"]} {
            set step synth
        } else {
            set step impl
        }

        # Run rtl synthesis
        puts "*** INFO: Open the RTL Design to select any
            pin or port to inject a fault."
        # if no elaborated design is open
        if {[catch {current_instance .}]} {
            eval synth_design -rtl -name $rtlDesign -top
               $topModule -part $part $quiet
        } else {
            # elaborated design already open, set it as
               current design
            if { [get_designs -quiet $rtlDesign]  != {} }
                {
                current_design $rtlDesign
                unhighlight_objects
            } else {
                eval synth_design -rtl -name $rtlDesign
                   -top $topModule -part $part $quiet
            }
        }

        # # Select input objects
        # set ports [get_ports -filter {DIRECTION==IN}]
        # set modulePins [get_pins -filter {!IS_LEAF &&
           DIRECTION!=OUT && !IS_CLOCK}]
        set leafPins [get_pins -hier -filter {IS_LEAF &&
           DIRECTION!=OUT && !IS_CLOCK}]

        ::ares::faults::inject_faults -f stuck-at-0 -o [
           lindex $leafPins 0]

```

```
35          set currentStep select
36      }
37
38      synth {# STEP#3: Open or reload synthesized design
39          puts "\n*** STEP: synth ***"
40          puts "==================\n"
41
42          if {[string equal $phase "synth"]} {
43              set step inject
44
45              if { [get_designs -quiet $synthDesign] != {}
                    } {
46                # synthesized design already opened, set it
                      as current design
47                current_design $synthDesign
48                refresh_design
49              } else {
50                  # open synthesized design to inject a
                      fault
51                  if {[file exists $synthDir/$moduleName/${
                      topModule}.edf]} {
52                      puts "*** INFO: Open synthesized
                          netlist of module '$topModule."
53                      read_edif $synthDir/$moduleName/${
                          topModule}.edf -quiet
54                      link_design -name $synthDesign -top
                          $topModule -part $part -quiet
55                      puts "*** INFO: linking synthesized
                          design is completed."
56                  } else {
57                      puts "*** INFO: Netlist file '${
                          topModule}.edf' not found in
                          $synthDir/$moduleName"
58                      run_synth $moduleName
59                  }
60              }
61          } else {
62              set step impl
63          }
64
65          set currentStep synth
66      }
67
68      impl {# STEP#4: Open or reload implemented design
69          puts "\n*** STEP: impl ***"
70          puts "=================\n"
```

```
71
72          if {[string equal $phase "synth"]} {
73              if {$runSim} {
74                  set step simulate
75              } else {
76                  set step emulate
77              }
78               run_impl $moduleName [expr {!$runFault}]
79          } else {
80              set step inject
81
82              # If implemented design already loaded
                   in-memory
83              if { [get_designs -quiet $implDesign] != {} }
                    {
84                  current_design $implDesign
85                  refresh_design
86              } else {# open implemented design to inject a
                    fault
87                  if {[file exists $implDir/$moduleName/
                       $implCheckpoint]} {
88                      puts "*** INFO: Open implemented
                           checkpoint of module '$moduleName'
                           ."
89                      read_checkpoint $implDir/$moduleName/
                           $implCheckpoint -part $part -quiet
90                      link_design -name $implDesign -top
                           $topModule -part $part -quiet
91                      puts "*** INFO: linking implemented
                           design is completed."
92                  } else {
93                      puts "*** INFO: Implemented
                           Checkpoint $implCheckpoint not
                           found in $implDir/$moduleName"
94                      run_impl $moduleName $runFault
95                  }
96              }
97          }
98
99          set currentStep impl
100     }
101
102     inject {# STEP#5: Modify design netlist for the fault
            injection
103         puts "\n*** STEP:inject ***"
104         puts "==================\n"
```

```
105
106         # Define next step and load correct design if no
               synthesized or implemented design is open
107         if {[string equal $phase "synth"]} {
108             if {$runSim} {
109                 set step simulate
110             } else {
111                 set step impl
112             }
113             if {[current_design] != [get_designs -quiet
                   $synthDesign]} {
114                 run_step synth
115             }
116         } else {
117             if {$runSim} {
118                 set step simulate
119             } else {
120                 set step emulate
121             }
122             if {[current_design] != [get_designs -quiet
                   $implDesign]} {
123                 run_step impl
124             }
125         }
126
127         puts "*** INFO: Modify synthesized/implemented
               netlist to inject a fault."
128         if {[catch {::ares::faults::modify_netlist -phase
                $phase} errorstring]} {
129             puts $errorstring
130             return
131         }
132
133         # fix route conflicts and re-route nets
134         if {[string equal $phase "impl"]} {
135             ::ares::faults::modify_netlist::fix_route_status

136         }
137
138         set currentStep inject
139     }
140
141     simulate {# STEP#6: Generate the simulation netlist
           und start xsim
142         puts "\n*** STEP: simulate ***"
143         puts "======================\n"
```

```
144         set step resume
145
146         # Starting simulation
147         if {$runSim} {
148             puts "*** INFO: Start simulation of module '
                    $moduleName'."
149             set cwd [pwd]
150             if {[string equal $phase "synth"]} {
151                 if {[catch {eval run_sim $moduleName
                        synth} errMsg]} {
152                     puts $errMsg
153                     cd $cwd
154                 }
155             } else {
156                 if {[catch {eval run_sim $moduleName impl
                        } errMsg]} {
157                     puts $errMsg
158                     cd $cwd
159                 }
160             }
161         }
162
163         set currentStep simulate
164     }
165
166     emulate {# STEP#8: Generate the bitstream and program
            the device.
167         puts "\n*** STEP: emulate ***"
168         puts "====================\n"
169         set step resume
170
171         puts "*** INFO: Generating injected bitstream and
                programming device."
172         if {[file exists $outputDir/$faultBitfile]} {
173             file delete -force $outputDir/$faultBitfile
174         }
175
176         if {$bitstream} {
177             write_bitstream -force -file $outputDir/
                    $faultBitfile -quiet
178         }
179
180         # program hardware device
181         program_device $outputDir/$faultBitfile
182
183         set currentStep emulate
```

```
184     }
185
186     resume {
187         puts "\n*** STEP: resume ***"
188         puts "====================\n"
189         set currentStep emulate
190
191         while {1} {
192             puts -nonewline "Inject next fault\[yes/no\]?
                    : "
193             flush stdout
194             gets stdin reinject
195             set reinject [string trim $reinject]
196             switch $reinject {
197                 yes { break }
198                 no { break  }
199                 default { puts unknown; continue }
200             }
201         }
202
203         if { $reinject } {
204             set step select
205             if {!$isFlowInLoop} {
206                 resume_flow $step
207             }
208         } else {
209             [expr {$isFlowInLoop ? [break] : [return]}]
210         }
211     }
212 }
```

Listing A.4: Tcl-Code für den Non-Project-basierten Designflow