Fachhochschule Dortmund
University of Applied Sciences and Arts

ET

Bachelor Thesis
zur Erlangung des Akademischen Grades
Bachelor of Engineering

Konzepte zur Steigerung der Resilienz gegenüber strahleninduzierten Logikfehlern des MOPS-HUB FPGA Entwurfs im Kontrollsystem des ATLAS Pixeldetektors

Lucas Schreiter

Matrikelnummer 7201349

8. Mai 2023

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis
Zweitprüfer: M.E. Alexander Walsemann

Erklärung

Hiermit versichere ich, dass die von mir vorgelegte Prüfungsleistung selbständig und ohne unzulässige fremde Hilfe erstellt worden ist. Alle verwendeten Quellen sind in der Arbeit so aufgeführt, dass Art und Umfang der Verwendung nachvollziehbar sind.

Dortmund, 8.5.2023

Unterschrift

Abstract

Diese Arbeit beschäftigt sich mit zwei Konzepten zur Steigerung der Resilienz gegenüber strahleninduzierten Logikfehlern des MOPS-HUB FPGA Entwurfs im Kontrollsystem des ATLAS Pixeldetektors am CERN. Um die Genauigkeit und Zuverlässigkeit der Detektordaten zu gewährleisten, müssen die elektronischen Systeme robust und fehlertolerant gegenüber einer strahlenbelasteten Umgebung sein. Zum einen wird die Möglichkeit der partiellen Rekonfiguration von Xilinx FPGAs als Methode zur Fehlerbereinigung des FPGA Konfigurationsspeichers näher vorgestellt. Es wurde ein Testentwurf und ein Programm zur teilweisen Rekonfiguration des FPGA aus der Anwenderlogik heraus mittels ICAP entwickelt. Als zweites Konzept wurde sich mit der Anwendung von TMR auf den MOPS-HUB Entwurf beschäftigt. Es wurden Tools entworfen, welche den manuellen Aufwand der Implementierung von TMR reduzieren und bei der Validierung unterstützen.

This work deals with two concepts for increasing the resilience to radiation-induced logic errors of the MOPS-HUB FPGA design in the control system of the ATLAS pixel detector at CERN. To ensure the accuracy and reliability of the detector data, the electronic systems must be robust and fault-tolerant to a irradiated environment. Firstly, the possibility of partial reconfiguration of Xilinx FPGAs is presented in more detail as a method to correct errors in the FPGA configuration memory. A test design and a program for partial reconfiguration of the FPGA from the user logic using ICAP was developed. As a second concept, the implementation of TMR on the MOPS-HUB design. Tools were designed that reduce the manual effort of implementing TMR and support validation.

Inhaltsverzeichnis

Abkürzungen

ASIC	Application Specific Integrated Circuit
ATLAS	A Toroidal Large Hadron Collider Apparatus
BRAM	Block Random Access Memory
CAN	Controller Area Network
CERN	Conseil Européen pour la Recherche Nucléaire
CRC	Cyclic Redundancy Check
DCS	Detector Control System
DFX	Dynamic Function eXchange
DLC	Data Lenght Code
E-Link	Electrical Link
EMCI	Embedded Monitoring and Control Interface
FE	Front-End
FET	Field Effect Transistor
FPGA	Field Programmable Gate Arry
HDL	Hardware Description Language
ICAP	Internal Configuration Access Port
ID	Inner Detector
ILA	Integrated Logic Analyzer
IP	Intellectual Property
ITk	Inner Tracker

JTAG	Joint Test Action Group
LHC	Large Hadron Collider
LUT	Look Up Table
MoPS	Monitoring of Pixel Systems
MOPS-HUB	Monitoring of Pixel Systems Hub
PLAG	Placement Generator
PP	Patch Panel
RBT	Raw Bitfile
RTL	Register Transfer Level
SDC	Synopsys Design Constraints
SEE	Single Event Effect
SEEG	Single Event Effects Generator
SET	Single Event Transient
SEU	Single Event Upset
SRAM	Static Random Access Memory
TBG	Testbench Template Generator
tcl	Tool Command Language
TMR	Triple Modular Redundancy
TMRG	Triple Modular Redundancy Generator
UART	Universal Asynchronous Receiver Transmitter

1 Einleitung

Das A Toroidal Large Hadron Collider Apparatus (ATLAS) Experiment ist einer der vier Kollisionsdetektoren am Large Hadron Collider (LHC), der größte Teilchenbeschleuniger der Welt. Er wird vom europäischen Kernforschungszentrum Conseil Européen pour la Recherche Nucléaire (CERN) in Genf, Schweiz, betrieben. Der LHC ist ein ringförmiger Beschleuniger mit einer Länge von circa 27 km, welcher sich etwa 50 m bis 175 m unter der Erdoberfläche befindet. Im LHC werden zwei entgegengesetzte Protonenstrahlen innerhalb des Beschleunigerrings auf nahezu Lichtgeschwindigkeit beschleunigt und im Zentrum der vier Kollisionspunkte zur Kollision gebracht. Jeder Strahl besteht aus mehreren kleineren Gruppen von dicht gepackten Protonen, die als Bündel bezeichnet werden. Mitte des Jahres 2026 soll der LHC auf den High-Luminosity, HL-LHC, Betrieb umgestellt werden [5]. Durch die erhöhte Luminosität erhöht sich die durchschnittliche Anzahl der potentiellen Kollisionen am ATLAS Detektor von derzeit etwa 37 auf rund 200 Kollisionen pro Bündelkreuzung [16]. Eine Bündelkreuzung beschreibt das Zusammentreffen zweier Protonenbündel oder Strahlen, wobei es nicht zwangsläufig zu Kollisionen zwischen den einzelnen Protonen kommt.

ATLAS, das größte der vier Experimente, befindet sich am Standort Punkt 1 (Point 1), einem der vier Kollisionspunkte des LHC. Es ist einer von zwei Detektoren, welcher zur Beobachtung von Proton-Proton Kollisionen dient. ATLAS umfasst eine Länge von 44 m, einen Durchmesser von 25 m in zylindrischer Bauform und befindet sich 100 m unterhalb der Erdoberfläche. In Abbildung 1.1 ist die Konstruktionsweise schematisch dargestellt.

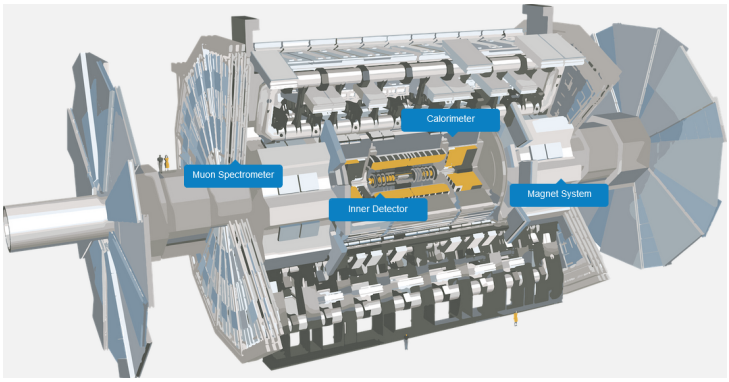


Abbildung 1.1: Schnittzeichnung des ATLAS Experimentes[1]

Der ATLAS Detektor ist in sechs Subsysteme unterteilt, die in Schichten um die Beamline angeordnet sind. Im Rahmen des geplanten Phase II Upgrades für den Hochluminositätsbetrieb sind wesentliche Verbesserungen am ATLAS-Detektor geplant, um die Leistungsfähigkeit des Experiments zu erhöhen und um den neuen Anforderungen gerecht zu werden. Unter anderem ist eine Neukonstruktion des Inner Detector (ID) geplant. Dieser ist schematisch in Abbildung 1.2 dargestellt.

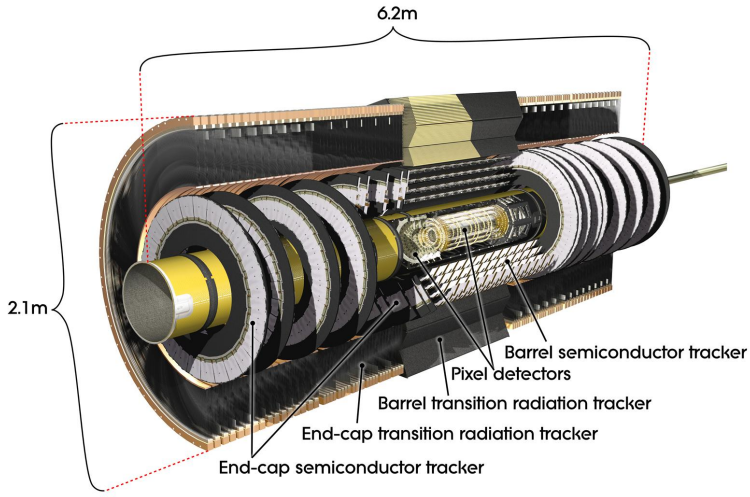


Abbildung 1.2: Schnittbild des Inner Detectors[17]

Die Aufgabe des ID besteht darin, Ladung, Impuls, sowie die Richtung von elektrisch geladenen Teilchen zu messen, welche bei den Kollisionen freigesetzt werden. Er wird im Rahmen des Phase II Upgrades durch einen neuen Detektor, den Inner Tracker (ITk), ersetzt. Herzstück bildet der Pixeldetektor, welcher der Beamline am nächsten ist. Wie der Name schon vermuten lässt, sind für die Detektion einzelne Pixel zuständig, welche in Pixelmodule gruppiert sind. Die neue Version des Pixeldetektors des ITk bietet unter anderem eine höhere Pixeldichte und eine verbesserte Leistung.

Aufgrund der Neukonstruktion des ITk wird zeitgleich ein neues Detector Control System (DCS) entwickelt, welches unter anderem für die Steuerung und Überwachung der einzelnen Pixelmodule des Pixeldetektors zuständig ist.

[1] Quelle: `https://atlas.cern/Discover/Detector`

Im Rahmen der Kooperation zwischen der Bergische Universität Wuppertal und der Fachhochschule Dortmund mit CERN wird ein strahlenharter Application Specific Integrated Circuit (ASIC) namens Monitoring of Pixel Systems (MoPS) entwickelt, dessen Aufgabe es ist, die Spannungsversorgung und Temperatur der Front-End (FE) Chips zu überwachen. Der FE ist ein weiterer strahlenharter ASIC, welcher für das Auslesen der Pixelsensoren zuständig ist und direkt an den Pixelmodulen montiert ist [15]. Der MoPS Chip kommuniziert mit dem DCS über einen Controller Area Network (CAN) Bus und kann bis zu 16 FEs überwachen [28][17][5].

1.1 MOPS-HUB FPGA Entwurf

Diese Arbeit befasst sich mit dem gleichnamigen Projekt Monitoring of Pixel Systems Hub (MOPS-HUB). MOPS-HUB ist ein Field Programmable Gate Arry (FPGA) Entwurf dessen Aufgabe es ist, die Überwachungsdaten der MoPS Chips zu aggregieren und an das DCS weiterzuleiten. MOPS-HUB ist in der Lage, 16 CAN Busse gleichzeitig zu verwalten. Pro CAN Bus können bis zu vier MoPS Chips teilnehmen. In Abbildung 1.3 ist das MOPS-HUB Netzwerk dargestellt. Für die CAN Funktionalität wird der CANakari[2] CAN Controller verwendet. Es handelt sich hierbei um eine ISO 11898 konforme CAN 2.0 B Controller Implementierung in VHDL und Verilog. Der CANakari findet bereits Einsatz im MoPS ASIC und wird daher auch im MOPS-HUB Entwurf verwendet.

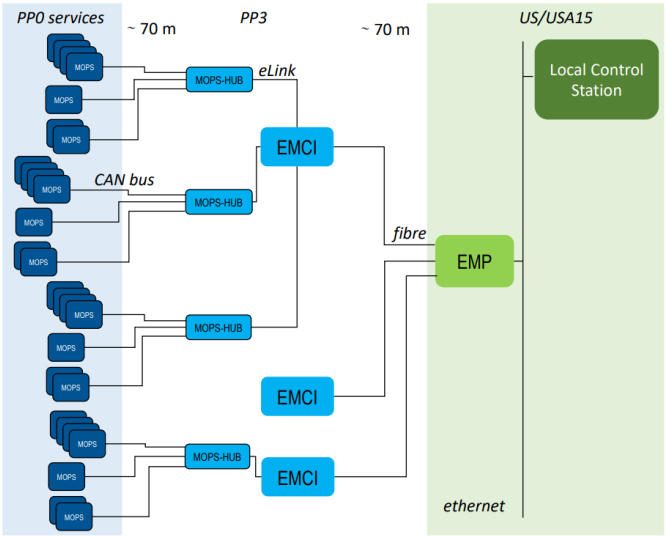


Abbildung 1.3: Einsatzzweck des MOPS-HUBs [3]

[2]Das CANakari Design sowie eine ausführliche Dokumentation findet sich auf GitHub: `https://github.com/mkaragou/Canakari`

Das eigentliche MOPS-HUB FPGA befindet sich am Standort Patch Panel (PP) 3. Dieser liegt rund 70m von dem ATLAS Detektor entfernt. Der Standort PP 0 bildet die direkte Verbindung zu den Pixelmodulen bzw. den MoPS Chips. Die agreggierten Datenströme der CAN Busse werden vom MOPS-HUB gebündelt und echt-differenziell über eine elektrische low-power, high-speed Verbindung namens Electrical Link (E-Link) an das Embedded Monitoring and Control Interface (EMCI) gesendet. E-Link wurde speziell von CERN für den Einsatz für Chip-to-Chip Kommunikation in einer strahlungsbelasteten Umgebung entwickelt. Das EMCI bildet die Schnittstelle zwischen dem MOPS-HUB und dem DCS - In Abbildung 1.3 grün dargestellt. Es hat die Aufgabe, alle E-Link Kanäle zu bündeln und über eine über eine optische Hochgeschwindigkeitsleitung über eine längere Strecke in Richtung DCS zu übertragen [3][18].

Der MOPS-HUB Entwurf besteht aus rund 122 Verilog Modulen (Inklusive CANakari) und wird zum Zeitpunkt der Arbeit aktiv in Hardware Description Language (HDL)-Designer Entwicklungsumgebung von Siemens EDA entwickelt. Mehr hierzu in Kapitel 3.4

1.1.1 MOPS-HUB FPGA

Bei dem MOPS-HUB FPGA handelt es sich um eine Static Random Access Memory (SRAM) basierte Artix-7 XCA200T von Xilinx in einer Ausführung mit einem FBG484 Gehäuse. Neben einem guten Kosten-Größenverhältnis wurde dieser Typ von FPGA bereits in anderen CERN Projekten am ATLAS Experiment verwendet. Dies ist von Vorteil, da bereits Daten zur Strahlenempfindlichkeit vorliegen [9][10].

Im Folgenden ist der Ressourcenbedarf des implementierten MOPS-HUB Entwurfs dargestellt. Zusätzlich sind die absolute und relative Ressourcennutzung der CANakaris, bezogen auf die MOPS-HUB Firmware, aufgeschlüsselt. Die Zahlen wurden aus der Entwicklungsumgebung Vivado des Anbieters Xilinx entnommen.

Ressource	**Verfügbar**	**MOPS-HUB**	**CANakari**
Lookup Table (LUT)	133800	32767 24,49%	17190 52,46%
Flip Flop (FF)	269200	18222 6,77%	10839 59,48%
Global Clock Buffer (BUFG)	32	1 3,13%	- -
IO Pins	285	69 24,21%	- -

Tabelle 1.1: Ressourcenbedarf des MOPS-HUB Entwurfs im MOPS-HUB FPGA.

1.2 Strahleneffekte im Kontext von FPGAs

Der Einsatz von integrierten Halbleiterschaltungen in Umgebungen mit einer hohen Strahlungsbelastung ist mit verschiedenen Herausforderungen und potenziellen Risiken verbunden. Eine der größten Herausforderungen ist die Empfindlichkeit gegenüber Single Event Effects (SEEs). Bei Single Event Effects handelt es sich um unvorhersehbare elektrische oder logische Verhaltensänderungen, welche durch das Auftreten eines energiereichen Partikels verursacht werden. Für diese Arbeit sind folgende Single Event Effects relevant:

- **Single Event Transient (SET):**
  Wenn ein ionisierendes Teilchen auf empfindliche Teile eines Field Effect Transistors (FETs) trifft, kann dies zu einer kurzzeitigen Änderung des Stromflusses durch den Transistor führen. Grundsätzlich sind zwei Szenarien zu betrachten: Zum einen kann der Einschlag eines hochenergetischen Teilchens in das FET Substrat zu einer Freisetzung von Elektron-Loch Paaren oder Defektelektronen führen. Durch die Anreicherung von Elektronen bzw. Defektelektronen kann der Kanal für eine kurze Zeit leitend werden. Zum anderen können sich Defektelektronen, welche durch einen Einschlag des Teilchens ins das Leitungsband befördert wurden, am Gate des Transistors sammeln und eine Modulation der Kanallänge hevorrufen. Diese plötzlichen Stromänderungen können Glitches im Digitalteil auslösen. Ein Single Event Transient (SET) verursacht oftmals temporäre Logikfehler.
- **Single Event Upset (SEU):**
  Ein Single Event Upset (SEU) beschreibt einen SET, dessen Impuls den Schwellwert eines Logikgatters oder einer Speicherzelle überschritten hat und somit einen falschen Wert annimmt. Ein SEU in einer Speicherzelle kann einen permanenten Logikfehler verursachen. Es handelt sich um einen Bitflip.

Selten sind SEU und SET von destruktiver Natur, können jedoch kritische Fehler in der Logik erzeugen. Vor allem SRAM basierte FPGAs sind besonders empfindlich gegenüber SEU und SET. Der Grund hierfür ist der SRAM basierte Konfigurationsspeicher, in welchem das Routing zwischen den Primitiven - wie Look Up Tables (LUTs) oder Flipflops - sowie die in den LUTs implementierten Logikfunktionen gespeichert ist. Ein SEU innerhalb des Konfigurationsspeichers kann möglicherweise zu einer Änderung in der programmierten Logik des FPGA führen [14]. Aufgrund der relativen Nähe des MOPS-HUB FPGAs zum ATLAS Experiment müssen Vorkehrungen getroffen werden, um die Resilienz des MOPS-HUB Entwurfs gegenüber strahleninduzierte Logikfehler zu steigern. Bei den in dieser Arbeit vorgestellten Konzepten handelt es sich um die partielle Rekonfiguration eines FPGAs sowie die Anwendung von Triple Modular Redundancy (TMR) auf den MOPS-HUB Entwurf.

2 Partielle Rekonfiguration

Die partielle Rekonfiguration ist eine bei FPGAs angewandte Technik, die es erlaubt, einen bestimmten Teil der Hardware im laufenden Betrieb neu zu konfigurieren, während der Rest des Systems ohne Beeinträchtigung weiterarbeitet. Dies steht im Gegensatz zu einer konventionellen Konfiguration, bei welcher das gesamte FPGA neu programmiert wird. Die Möglichkeit der dynamischen Rekonfiguration der Hardware erhöht die Flexibilität enorm, da Logikblöcke ohne systemweite Unterbrechung rekonfiguriert werden können und eine nahtlose Anpassung eines Systems an sich ändernde Anforderungen erfüllt werden kann. Durch die gemeinsame Nutzung von Ressourcen kann der Gesamtressourcenbedarf gesenkt werden, was letztendlich zu einem kostengünstigeren Gesamtsystem führt. Zusätzlich verbessert die partielle Rekonfiguration die modulare Upgrade Fähigkeit eines Systems.

Für diese Arbeit von Interesse ist die partielle Rekonfiguration als Methodik zur Fehlerbereinigung des Konfigurationsspeichers des FPGAs. Durch ein zyklisches partielles überschreiben der Konfiguration werden strahleninduzierte Fehler bereinigt, ohne dabei das Gesamtsystem zu beeinträchtigen. Die partielle Rekonfiguration wird von Xilinx unter anderem als Dynamic Function eXchange (DFX) bezeichnet und beworben.

Bei der partiellen Rekonfiguration wird zwischen dem statischen und rekonfigurierbaren Teil unterschieden. In Abbildung 2.1 ist das Grundprinzip dargestellt.

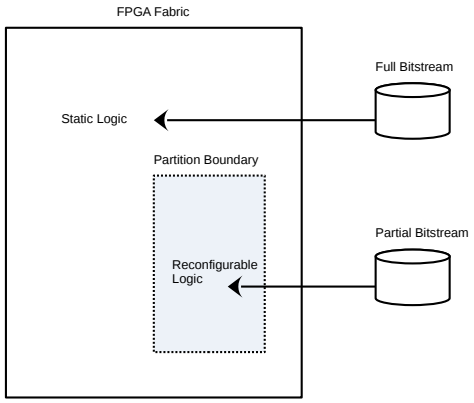


Abbildung 2.1: Schematische Darstellung einer partiellen Rekonfiguration

Bei dem dynamisch rekonfigurierbaren Teil handelt es sich um ein logischen Abschnitt eines Entwurfs, welcher vom Benutzer an einer physischen hierarchischen Grenze definiert wird. Der Bereich, den die hierarchische Grenze einschließt, wird auch als Partition bezeichnet. Ein Entwurf kann mehrere Partitionen aufweisen. Je Partition können mehrere rekonfigurierbare Module existieren, welche innerhalb der Partition implementiert werden sollen. Als statische Logik wird die Logik außerhalb der Partition bezeichnet, welche nicht dynamisch rekonfigurierbar ist und immer aktiv bleibt. Hierbei handelt es sich um die Top-Level Logik. In einem DFX Design werden mehrere Bitstreams generiert. Hierbei konfiguriert der volle Bitstream den gesamten FPGA, d.h. den statischen Teil und die Partition, der partielle Bitstream nur die Partition [25]. Die Anzahl der generierten Bistreams hängt von der Anzahl von Partition und rekonfigurierbaren Modulen ab. Zusätzlich ist es möglich, einen vollen Bitstream mit einer Greybox Definition der Partition zu erzeugen. Eine Greybox Definition beschreibt nur die Ports einer Partition, jedoch keine rekonfigurierbare Logik. Die Partition ist als leer anzusehen. Das Schreiben eines partiellen Bitstreams, welcher als Greybox definiert ist, würde vorhandene Logik innerhalb der Partition entfernen. Eine Anleitung zur Erstellung eines DFX-Designs sowie zur Erstellung und Planung von Partitionen in Vivado findet sich im Anhang A.1.

Während einer partiellen Rekonfiguration ist es empfehlenswert, die Pins der Partition von dem statischen Teil zu entkoppeln. Dies hat den Hintergrund, dass während einer Konfiguration undefinierte Zustände an den Partitionpins anliegen können, was letzendlich zu Glitches in der Logik führen kann. Dies kann bereits durch einen einfachen 2 in 1 Multiplexer verhindert werden:

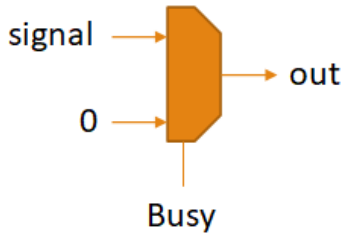


Abbildung 2.2: Mutliplexer zur Entkopplung während einer Rekonfiguration

Je nach Designkomplexität reicht ein Multiplexer jedoch nicht aus, um das Design vollständig zu entkoppeln. Sollten zum Beispiel Bus Systeme verwendet werden, müssen Mechanismen implementiert werden, um fehlerhafte Transaktionen zu verhindern und einen sicheren Abbruch der Kommunikation zu gewährleisten. Auch können Active Low Steuersignale bei dem Übergang in eine Partition Probleme bereiten.

Steuersignale wie Reset sind in der FPGA Architektur üblicherweise Active High, daher wird ein im Entwurf verwendetes Active Low Signal durch eine LUT invertiert. Die verwendeten LUTs für die Signalinversion können jedoch zu einer einzelnen LUT am I/O Element zusammengefasst werden. Die Ports einer Partition sind jedoch fest definiert, was dazu führen kann, dass viele einzelne Inverter LUTs innerhalb der Partition bestehen bleiben und nicht wie im Top-Level reduziert werden, welches das Routing und Timing negativ beeinflusst. Um dies zu vermeiden, sollte das Active Low Signal vor dem Eintritt in die Partition invertiert und im Topmodul des rekonfigurierbaren Moduls wieder invertiert werden. Auf diese Weise wird sichergestellt, dass nur eine LUT für die Signalinversion verwendet wird. Des Weiteren gibt es Limitierungen in Bezug darauf, welche Logikelemente in einem rekonfigurierbaren Modul verwendet werden können. Taktverändernde Logik, I/O sowie Hardwarespezifische Primitive wie zum Beispiel der Internal Configuration Access Port (ICAP) können nur im Top-Level existieren und sind deshalb nur im statischen Teil verfügbar [25].

Ziel dieses Kapitels ist die Entwicklung eines FPGA Testentwurfs, welcher in der Lage ist, Bitstream-Daten über eine serielle Schnittstelle zu empfangen und eine partielle Rekonfiguration selbständig durchzuführen. Zusätzlich wurde eine Benutzeroberfläche entwickelt, die es ermöglicht, die Bitstream-Daten von einem PC an den FPGA zu senden.

2.1 Internal Configuration Access Port

Der ICAP ist eine Primitive in Xilinx FPGAs, die es dem Benutzer ermöglicht, auf die Register der internen Konfigurationslogik aus der vom Benutzer im FPGA implementierten Logik zuzugreifen. Die Register haben eine Wortbreite von 32 Bit und ermöglichen dem Anwender den Zugriff auf den Konfigurationsspeicher sowie auf eine Vielzahl unterschiedlicher Statusinformationen. Dabei verhält sich der ICAP aus Sicht des Benutzers wie eine registerbasierte Schnittstelle. Aufgrund der begrenzt vorhandenen Dokumentation, die von Xilinx in Bezug auf das ICAP Interface zur Verfügung gestellt wird, hat das vorliegende Kapitel das Ziel, eine zentrale Zusammenstellung von Informationen zu diesem Interface bereitzustellen.

Der ICAP ist in allen 7-Series (ICAPE2) sowie UltraScale und UltraScale+ (ICAPE3) FPGA Bauelementen vorzufinden, jedoch auch in den älteren Modellen der Spartan-6 und Virtex-6 Produktserie [13][6]. Je nach Produktserie ist die ICAP Primitive jedoch anders ausgeführt. In diesem Kapitel wird ausschließlich die Version ICAPE2 (7-Series) behandelt.

Für ein besseres Verständnis ist in Quelltext 2.1 die Portbeschreibung bzw. die Vorlage für die Instanziierung gezeigt.

```
ICAPE2 #(
    // Specifies the pre-programmed Device ID value to be used
    // for simulation purposes.
   .DEVICE_ID(32'h3651093),
   // Specifies the input and output data width.
   .ICAP_WIDTH("X32"),
   // Specifies the Raw Bitstream (RBT) file to be parsed by
   // the simulation model.
   .SIM_CFG_FILE_NAME("NONE")

)
ICAPE2_inst (
   .O(O),         // 32-bit output: Configuration data output bus
   .CLK(CLK),     // 1-bit input: Clock Input
   .CSIB(CSIB),   // 1-bit input: Active-Low ICAP Enable
   .I(I),         // 32-bit input: Configuration data input bus
   .RDWRB(RDWRB)  // 1-bit input: Read/Write Select input
);
```

Quelltext 2.1: Instanzierung von ICAPE2 entnommen aus [22]

Neben den fünf Signalen für Takt, Steuerung und Daten besitzt die Instanz ein weiteres wichtiges Attribut *ICAP_WIDTH*. Dieses legt bei der Instanziierung die Breite des Datenein- und Datenausgangs fest und kann während der Laufzeit des FPGA nicht verändert werden. Die maximale Taktfrequenz des ICAPE2 ist mit 100MHz für die Geschwindigkeitsklassen -1, -2, -3 und 70MHz für die Low-Voltage-Varianten spezifiziert [4]. Bei einer Taktfrequenz von 100MHz und einer maximalen Busbreite von 32Bit ergibt sich somit eine theoretische Bandbreite von 3,2Gbit/s, da Daten zu jeder steigenden Taktflanke übernommen werden. Der Verilog oder VHDL Code für die Instanzierung lässt sich im Datenblatt [22] finden.

2.1.1 ICAP Simulation

Für Simulationszwecke gibt es noch zwei weitere interessante Attribute der ICAPE2 Instanz: *DEVICE_ID* und *SIM_CFG_FILE_NAME*. Mit *DEVICE_ID* kann die FPGA spezifische ID angegeben werden. Beispielsweise beträgt die ID für einen Artix-7 XC7A100TCSG324-1 Baustein 3631093. Es handelt sich hierbei um den FPGA, welcher für den Testentwurf verwendet wird. Mit *SIM_CFG_NAME* kann der Pfad zu einem Bistream im Raw Bitfile (RBT) Format angegeben werden. Somit wäre es für das Simulationsmodell möglich, simple Funktionen wie das Auslesen eines einzelnen Registers, dessen Daten im Bitstream hinterlegt sind, zu simulieren. Jedoch lieferte die Simulation keine sinnvolle Ergebnisse, da bei einem Lesevorgang der Ausgang auf Z liegt obwohl die RBT Datei erfolgreich geladen wird:

„Message: ICAPE2 on instance [...] has finished initialization. User can start read/write operation“.

Ein ähnlicher Fehler wurde bei der Recherche in einem Xilinx Forenbeitrag gefunden[3]. Der Pfad zur RBT Datei ist gültig, da ansonsten folgende Fehlermeldung geworfen wird:

„Error: The configure rbt data file for ICAPE2 instance was not found. Use the SIM_CFG_FILE_NAME parameter to pass the file name“.

Die Simulation stützt sich auf die Konfiguration Simulationsmodelle „SIM_CONFIGE2.v“ sowie „ICAPE2.v“, welche im Vivado Installationsverzeichnis unter „`/data/verilog/src/unisims`“ gefunden werden können. Eine direkte Einbindung in eine Testbench brachte jedoch keine Veränderung. Auch eine Testbench von Xilinx für SelectMap, die überraschenderweise und ohne jeglichen Hinweis eine Testbench für ICAPE2 inklusive einem Bitstream im RBT Format enthält, zeigt selbiges Verhalten [26][7]. Daher ist davon auszugehen, dass eine ICAPE2 Simulation in Vivado[4] nicht funktionsfähig ist, weshalb auf den Integrated Logic Analyzer (ILA) von Vivado zurückgegriffen werden muss.

2.1.2 Konfigurationslogik Register

Der Konfigurationsspeicher von Series-7 FPGAs ist in Frames segmentiert, die über den gesamten Baustein verteilt sind. Ein Frame ist das kleinste adressierbare Segment des Konfigurationsspeichers und besteht aus jeweils 101 32-Bit-Wörtern. Die kleinste adressierbare Einheit ist somit 32 Bit. Alle Befehle werden durch Schreiben und Lesen der Konfigurationsregister durchgeführt.

In einem Konfigurationsbitstream finden sich zwei Arten zur Adressierung der Register der Konfigurationslogik, welche von Xilinx als Packet Type 1 und 2 bezeichnet werden:

Header Type	**Opcode**[5]	**Register Address**	**Reserved**	**Word Count**
[31:29]	[28:27]	[26:13]	[12:11]	[10:0]
001	xx	RRRRRRRRRxxxxx	RR	xxxxxxxxxxx

Tabelle 2.1: Type 1 Packet, nach [2]

[3] `https://support.xilinx.com/s/question/0D52E00006hpPhiSAE/icap-simulation?language=en_US`

[4] Zum Zeitpunkt der Arbeit wurde mit Vivado 2021.2 gearbeitet

[4] 00 = No Operation, 01 = Read, 10 = Write, 11 = Reserved

Header Type	**Opcode**[5]	**Word Count**
[31:29]	[28:27]	[26:0]
010	xx	xxxxxxxxxxxxxxxxxxxxxxxxxxx

Tabelle 2.2: Type 2 Packet, nach [2]

Das Type 1 Packet wird zum Schreiben und Lesen von Registern verwendet. Der *Opcode* gibt hierbei an, ob es sich um eine Schreib- oder Leseoperation handelt. Zusätzlich kann *Opcode* die Instruktion „No Operation“ beinhalten. Dieses Packet wird auch als „NOOP“ bezeichnet und beinhaltet keinerlei Daten. Ein „NOOP“ weist die interne Konfigurationslogik an, das Packet zu ignorieren. Während 14 Bits für die Register Adressierung reserviert sind, werden nur die unteren 5 Bits verwendet. Die Reserved Bits sind als null anzusehen. Die Wortzahl gibt die Anzahl von Konfigurationsdatenwörter an, welche dem Packet Type 1 folgen. Das Type 2 Packet findet Anwendung, wenn die Anzahl der Wörter die Kodierungsmöglichkeit durch die verfügbaren 11 Bits überschreitet. Die Adresse aus dem letzten Type 1 Packet wird als Ziel wiederverwendet [2].

Insgesamt verfügt die interne Konfigurationslogik über 14 Register, von denen fünf aktiv zur Konfiguration des Konfigurationsspeichers verwendet werden. Eine Erläuterung aller einzelnen Register und deren Inhalte würde jedoch das Thema dieses Kapitels verfehlen, weshalb für weitere Informationen auf Datenblatt [2] verwiesen wird. Im Folgenden werden daher nur die für die Konfiguration wichtigsten Register vorgestellt.

NAME	**R/W**	**Adresse**	**Beschreibung**
CRC	R/W	00000	CRC Register
FAR	R/W	00001	Frame Adressierungs Register
FDRI	W	00010	Konfigurationsdatenregister, Input
FDRO	R	00011	Konfigurationsdatenregister, Ouput
CMD	R/W	00100	Command Register
IDCODE	R/W	01100	Hardware-ID
MFWR	W	01010	Mehrfach FDRI

Tabelle 2.3: Konfigurationsregister, nach [2]

Vor Beginn einer Konfiguration muss zunächst die FPGA spezifische ID in das Register *IDCODE* geschrieben werden. Stimmt diese ID nicht mit der ID des FPGAs überein, ist davon auszugehen, dass die Konfigurationsdaten nicht für das FPGA geeignet sind und der Konfigurationsvorgang dementsprechend abgebrochen wird. Das aktuell zu schreibende oder lesende Wort befindet sich im Register *FDRI* oder *FDRO*. Mit dem Register *FAR* wird der entsprechende Frame des Konfigurationsspeichers adressiert. Sobald ein Frame vollständig beschrieben wurde, wird die Adresse autoinkrementiert.

Das Register *CMD* instruiert die Konfigurationslogik. Vereinfacht ausgedrückt handelt es sich hierbei um eine Zustandsmaschine, die je nach Registerinhalt die angeforderte Aktion ausführt. Bei einer Konfiguration führt die Konfigurationslogik eine Cyclic Redundancy Check (CRC) Prüfung durch, sobald die Prüfsumme in das Register *CRC* geschrieben wird. Weicht diese Prüfsumme von den Bitstreamdaten ab, wird ein entsprechendes Flag gesetzt und die Konfiguration abgebrochen. Das Register *MFWR* wird zur Komprimierung des Bitstreams verwendet, siehe Kapitel 2.3.

Tatsächlich existiert der ICAPE2 in 7-Series FPGAs zweimal. Aus Datenblatt [20] geht hervor, dass die Doppelung für einen besseren Schutz gegenüber SEU implementiert ist. Im Register *CTL0* mit der Adresse 00101 kann mit dem Bit *ICAP_SELECT* zwischen den beiden Primitiven umgeschaltet werden [2].

2.1.3 Schreib- und Leseoperation

Am ICAPE2 Interface anliegende Daten werden solange ignoriert, bis ein Synchronisationswort am Dateneingangsbus erscheint. Das Synchronisationswort für 7-Series FPGAs lautet 0xAA995566. Erst nach diesem Wort kann eine Konfiguration oder Steuerung beginnen. Das Ende einer Sequenz wird mit dem Desynchronisationswort 0x0000000D gekennzeichnet. Im Gegensatz zum Synchronisationswort muss das Desynchronisationswort in das Register *CMD* geschrieben werden.

Die Schreib- und Leseoperationen von ICAPE2 sind denen der SelectMap Schnittstelle sehr ähnlich und sind in Abbildung 2.3 dargestellt.

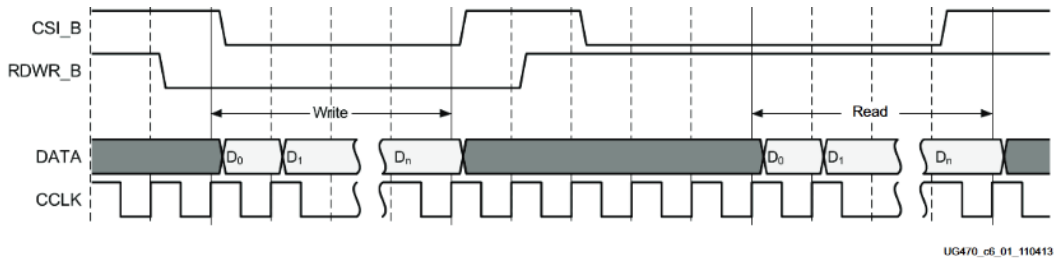


Abbildung 2.3: Schreib- und Leseoperation des ICAPE2 Interface, ursprünglich für SelectMap, entnommen aus [2]

Mit jeder positiven Taktflanke werden die Signale an den Eingängen übernommen, solange *CSIB* aktiv ist (csib = 0). Schreiboperationen (rdwrb = 0) können auf zwei Arten durchgeführt werden: kontinuierliches Schreiben und diskontinuierliches Schreiben. Kontinuierliches Schreiben setzt voraus, dass der Konfigurationsdatenstrom ununterbrochen zur Verfügung steht.

Wenn eine kontinuierliche Datenübertragung nicht aufrechterhalten werden kann, muss der Schreibvorgang pausiert werden, indem das *CSIB* zurückgesetzt wird (csib = 1). Alternativ listet die Xilinx Dokumentation [2] auch die Möglichkeit der Taktabeschaltung. Eine Leseoperation (rdwrb = 1) beginnt immer mit einer Schreiboperation, bei der ein Lesebefehl an das betreffende Register adressiert wird. Anschließend wird *RDWRB* auf Lesen getoggelt. Die Lesedaten sind drei Taktzyklen nach Aktivierung von *CSIB* am Ausgang deterministisch gültig. Während bei einer Konfiguration die Schreiboperationen durch den Bitstream vorgegeben sind, können Leseoperationen auf einzelne Register mit folgender Sequenz durchgeführt werden:

Bit	**Daten**	**Beschreibung**
Write	FFFFFFFF	Dummy
Write	AA995566	Sync Wort
Write	20000000	NOOP
Write	28000001	Type 1 Packet, lesend, CRC Register
Write	20000000	NOOP
Read		ICAP schreibt
Write	30008001	Type 1 Packet, schreibend, CMD Register
Write	0000000D	Desync Wort
Write	20000000	NOOP
Write	20000000	NOOP

Tabelle 2.4: Lesesequenz von Registern [2]

Mit dieser Sequenz kann Beispielsweise das interne *CRC* Register ausgelesen werden. Das Type 1 Packet für das Lesen des *CRC* Register setzt sich aus dem Header Type 1 (001), dem Opcode Lesen (01), die *CRC* Registeradresse (00000) und der Wortanzahl 1 zusammen. Anschließend muss ein *NOOP* geschrieben werden. Nach Abschluss wird *RDWRB* auf Lesen getoggelt und der Wert erscheint drei Taktzyklen später am Datenausgang. Anschließend erfolgt das Schreiben des Desynchronisationswortes in das *CMD* Register. Eine Liste der 14 verfügbaren Registern und ihre Adressen findet sich im Xilinx Datenblatt [2].

Während des Togglevorgangs von *RDWRB* muss *CSIB* deaktiviert sein, da *RDWRB* sich bei aktivem *CSIB* nicht ändern darf. Falls *RDWRB* doch bei aktivem *CSIB* getoggelt wird, kommt es zu einem *ABORT* d.h. Abbruch der Lese- oder Schreibsequenz. Bei einem *ABORT* handelt es sich um einen Interrupt, bei welchem ein Statusbyte auf den Ausgang gelegt wird [2]. Das Statusbyte liegt immer auf O[7:0], während O[31:8] High ist. Nach Ablauf der *ABORT* Sequenz kann eine Konfiguration nach einer Synchronisation wieder aufgenommen werden. Eine Aufschlüsselung des Abortbyte ist in Tabelle 2.5 angegeben.

Bit	Bezeichnung	Beschreibung
7	CFGERR_B	Konfigurationsfehler 0 = Fehler 1 = Kein Fehler
6	DALIGN	Sync-Wort empfangen 0 = Kein Sync-Wort empfangen 1 = Sync-Wort empfangen
5	RIP	Status Leseoperation 0 = Kein aktiver Lesevorgang 1 = Aktiver Lesevorgang
4	IN_ABORT_B	Status ABORT 0 = ABORT aktiv 1 = ABORT nicht aktiv
3:2	RSVD	Reserviert
1:0	11	Dauerhaft 11

Tabelle 2.5: ABORT Statusbyte nach [2]

Da das Datenblatt [2] keine genaue Aussage über das *ABORT* Verhalten von ICAPE2 tätigt, wurde der Ausgang mit ILA untersucht:

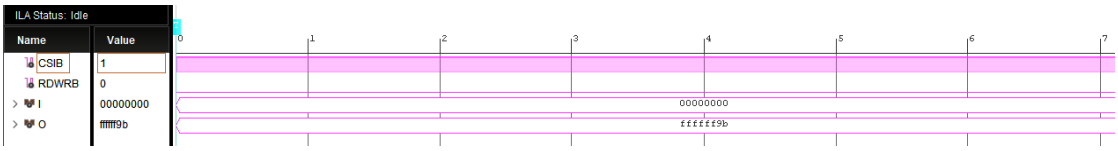


Abbildung 2.4: ICAPE2 ABORT Statuswort

Im Gegensatz zum SelectMap Interface liegt das *ABORT* Statuswort dauerhaft am Ausgang des ICAPE2 Interface an, solange der Ausgangsbus nicht anderweitig aktiv ist, siehe Abbildung 2.4. Dies ist möglich, da ICAP im Gegensatz zu SelectMap keinen einzelnen bidirektionalen Datenbus besitzt. Der Signalverlauf in Abbildung 2.4 wurde direkt nach der Konfigurierung des FPGAs aufgenommen. Wie erwartet sind die 24 höchstwertigen Bits High und die 8 niederwertigsten Bits das Statuswort.

Soll eine Konfiguration über ICAPE2 erfolgen, so ist darauf zu achten, dass die Bitreihenfolge in jedem Byte umgekehrt wird. Dies soll heißen, dass das niederwertigste Bit innerhalb eines Bytes zum höchstwertigen Bit wird und so fort. Beispielsweise wird aus dem Byte 0xAA das Byte 0x55. Dies gilt auch für gelesenen Daten mit Ausnahme des *ABORT* Statusbytes [2].

Für den Fall, dass das ICAP Interface mit einer Busbreite von weniger als 32 Bit konfiguriert ist, werden die höchstwertigen Stellen zuerst geschrieben.

2.2 STARTUP Primitiv

Das STARTUP Primitiv ermöglicht den Zugriff auf interne Konfigurationssignale der Konfigurationslogik. Im Folgenden ist die Vorlage für die Instanziierung gezeigt.

```
1  STARTUPE2 #(
2     // Activate program event security feature. Requires encrypted
          bitstreams.
3     .PROG_USR("FALSE"),
4     // Set the Configuration Clock Frequency(ns) for simulation.
5     .SIM_CCLK_FREQ(0.0)
6  )
7  STARTUPE2_inst (
8     // 1-bit output: Configuration main clock output
9     .CFGCLK(CFGCLK),
10    // 1-bit output: Configuration internal oscillator clock output
11    .CFGMCLK(CFGMCLK),
12    // 1-bit output: Active high output signal indicating the End
          Of Startup.
13    .EOS(EOS),
14    // 1-bit output: PROGRAM request to fabric output
15    .PREQ(PREQ),
16    // 1-bit input: User start-up clock input
17    .CLK(CLK),
18    // 1-bit input: Global Set/Reset input (GSR cannot be used for
          the port name)
19    .GSR(GSR),
20    // 1-bit input: Global 3-state input (GTS cannot be used for
          the port name)
21    .GTS(GTS),
22    // 1-bit input: Clear AES Decrypter Key input from Battery-
          Backed RAM (BBRAM)
23    .KEYCLEARB(KEYCLEARB),
24    // 1-bit input: PROGRAM acknowledge input
25    .PACK(PACK),
26    // 1-bit input: User CCLK input
27    .USRCCLKO(USRCCLKO),
28    // 1-bit input: User CCLK 3-state enable input
29    .USRCCLKTS(USRCCLKTS),
30    // 1-bit input: User DONE pin output control
31    .USRDONEO(USRDONEO),
32    // 1-bit input: User DONE 3-state enable output
```

```
33      .USRDONETS(USRDONETS)
34  );
```

Quelltext 2.2: Instanzierung von STARTUPE2 entnommen aus [22]

Die Aufmerksamkeit gilt dem `EOS` Signal. `EOS` indiziert das absolute Ende der Konfiguration und ist somit ideal für die Entkopplung einer Partition während einer Konfiguration geeignet. Für eine Erklärung der restlichen Signale wird auf Quelle [2] verwiesen, da die Erklärung dieser Signale keinen sinnvollen Beitrag zum Thema der partiellen Rekonfiguration darstellt.

2.3 Bitstream Formate

Xilinx 7-Series FPGAs können über verschiedene Schnittstellen auf unterschiedliche Weise konfiguriert werden. Aus diesem Grund existieren verschiedene Bitstream Formate, um den jeweiligen Anforderungen gerecht zu werden. Wie der Name Bitstream schon sagt, handelt es sich um einen Datenstrom, welcher direkt von der internen Konfigurationslogik ausgewertet wird. Üblicherweise werden Bitstream Formate im BIT Format verwendet, welches in binärer Form dargestellt ist. Öffnet man beispielsweise einen Bitstream im BIT Format mit einem Hex-Editor, so zeigt sich folgende binäre Struktur:

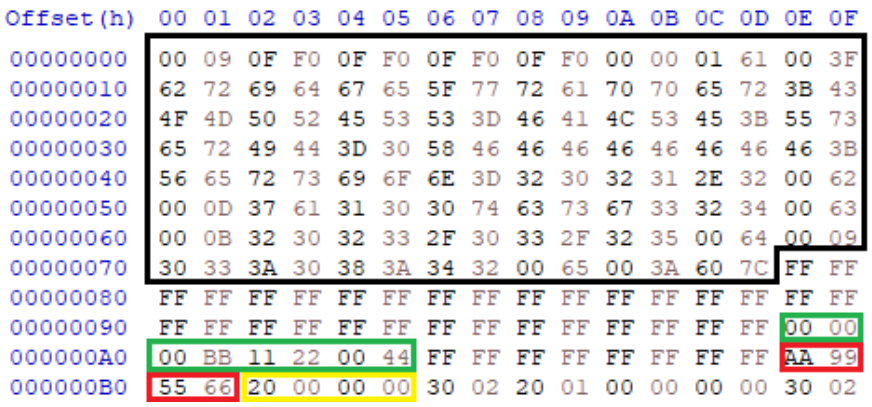


Abbildung 2.5: Struktur eines Bitstreams im BIT Format

Bei dem schwarz umrandeten Teil handelt es sich um einen Header von Vivado. In diesem werden Informationen wie Zeitstempel oder Toolversion abgespeichert. Ohne diesen Header verweigert Vivado die Programmierung eines FPGA. Bei dem grün markierten Bereich handelt es sich um eine Busbreitendetektion. Sie wird bei parallelen Konfigurationsmodi verwendet, um die Breite des Konfigurationsbusses automatisch zu erkennen. In rot markiert ist das bereits bekannte Synchronisationswort. Ab hier beginnt die eigentliche Konfiguration des FPGA.

In Gelb ist nochmals das bekannte NOOP Packet im Hex Format gekennzeichnet. Am Ende des Bitstreams findet sich noch das Desynchronisationswort, welches in der Abbildung 2.5 jedoch nicht dargestellt ist. Aufgrund der Anordnung des Bistreams entspricht die Länge in etwa der Größe des Konfigurationsspeichers. Die Transparenz des Bitstreams ist für eine individuelle Konfigurationslösung von Vorteil, da keine größere Datenverarbeitung stattfinden muss. Neben dem BIT Format existieren noch weitere Formate:

- **BIN:**
  Das BIN Format ist ähnlich zum BIT Format, jedoch fehlt der schwarz markierte Vivado Header in Abbildung 2.5. Für eine Konfiguration über ICAP ist dieser somit dem BIT Format vorzuziehen, da der Header Informationen enthält, die im Kontext der partiellen Konfiguration von keinem Interesse sind.

- **RBT:**
  Hierbei handelt es sich um ein ASCII äquivalent zu dem BIN Format mit identischen Inhalt. Der Inhalt ist ASCII codiert d.h. die Binärdaten werden als Hexadezimalzahlen aus den ASCII Zeichen 0 bis 9 sowie A bis F dargestellt

- **MCS:** Wie das RBT Format ist auch das MCS Format ASCII codiert. Jede Zeile repräsentiert einen Datensatz mit Konfigurationsdaten, Zieladresse und Prüfsumme. In diesem Format ist die Bitreihenfolge bereits invertiert.

- **HEX:** Ähnlich zum MCS Format, jedoch sind nur die Konfigurationsdaten enthalten. Die Invertierung der Bitreihenfolge ist optional und kann vom Benutzer gesteuert werden.

Zudem ermöglicht Vivado die Komprimierung des Bitstreams. Dabei werden identische Wörter, die mehrfach nacheinander auftreten, zusammengefasst. Um dies zu erreichen, wird das in Tabelle 2.3 dargestellte *MFWR* Register verwendet. In diesem Register wird eingetragen, wie oft der Inhalt des *FDRI* Registers übernommen werden soll. Die Kompression des Bitstreams hängt jedoch stark vom Design und der Größe ab [8]. Die Bitstream Kompression kann mit folgendem Vivado Constraint aktiviert werden:

```
1 set_property BITSTREAM.GENERAL.COMPRESS True [current_design]
```

Quelltext 2.3: Constraint für die Bitstream Komprimierung in Vivado

2.4 Testentwurf

In diesem Kapitel wird ein Entwurf zum Test der partiellen Rekonfiguration eines 7-Series FPGAs über das ICAPE2 Interface vorgestellt. Zur besseren Übersicht wird zunächst das Blockschaltbild gezeigt:

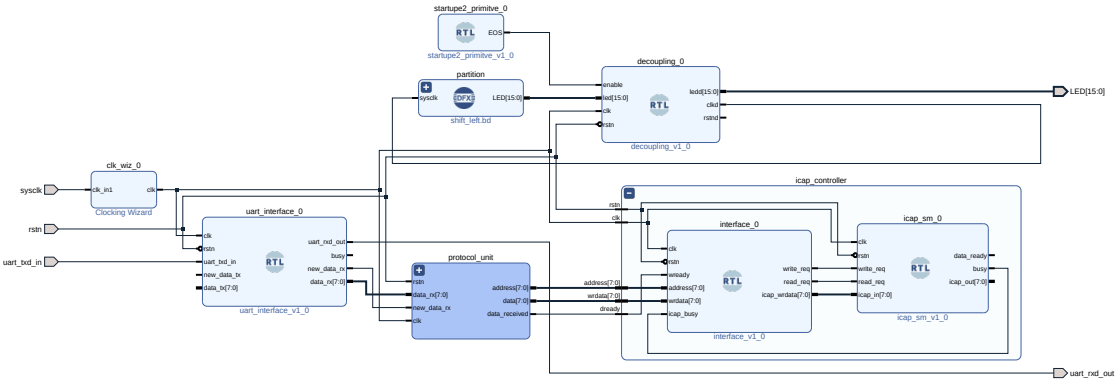


Abbildung 2.6: Block Diagramm des Testentwurf

Um die Bitstreamdaten an die ICAP Primitive senden zu können, wird ein Design benötigt, welches die Daten über Universal Asynchronous Receiver Transmitter (UART) von einem PC empfangen und verarbeiten kann. Der Entwurf setzt sich aus sieben Blöcken zusammen, wobei der ICAP Controller das Herzstück aufweist. Im Wesentlichen stellt der Entwurf eine Brückenlogik zwischen dem UART Interface und dem ICAP Controller dar. Das UART Interface wurde für die Arbeit zur Verfügung gestellt. Aufgrund der geringen Bandbreite von UART wird die Funktion des diskontinuierlichen Schreibens durch toggeln von csib verwendet.

2.4.1 ICAP Controller

Zunächst soll näher auf den ICAP Controller eingegangen werden. In Tabelle 2.6 ist die Portbeschreibung aufgelistet.

Port	Breite	In/Out	Beschreibung
rstn	1	In	Reset (active low)
clk	1	In	Taktsignal, max. 100MHz
address	8	In	Adressierung Lesen, Schreiben
wrdata	BUS_WIDTH	In	Schreibdaten
dready	1	In	Gültige Eingangsdaten
busy	1	Out	Aktiver Lesevorgang

Port	Breite	Beschreibung	Beschreibung
rdata	BUS_WIDTH	Out	Lesedaten

Tabelle 2.6: Portdefinition des ICAP Controller

Der ICAP Controller koordiniert die Schreib- und Leseoperation des ICAP Primitivs durch das Setzen der Signale `csib` und `rbwrb` und bietet ein registerbasiertes Interface für eine einfache Steuerung. Das Interface reagiert auf die Adressen 0x00 (Schreiben) und 0x01 (Lesen). Sobald `wready` high wird, werden die anliegenden Daten `wrdata` und `address` von der Protocol Unit übernommen. Im Falle eines Schreibvorgangs auf die Adresse 0x00 werden die Konfigurationsdaten in das interne Register `data_register` geschrieben. Das Interface signalisiert der ICAP_sm den erfolgreichen Empfang einer Schreibinstruktion durch das Signal `write_req`. In Abbildung 2.7 ist dieser Vorgang gezeigt.

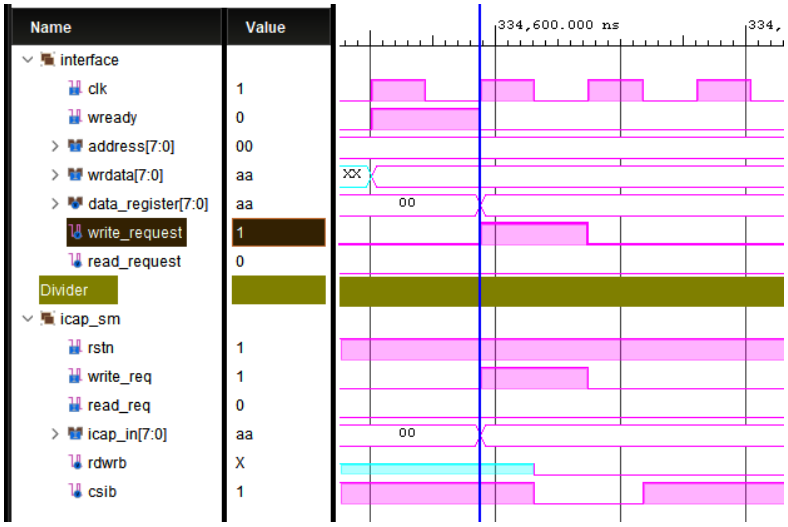


Abbildung 2.7: ICAP Schreibvorgang

Nach dem Erhalt von `write_req` werden die Signale `csib` und `rdwrb` entsprechend mit der nächsten negativen Taktflanke gesetzt. Das Setzen der Signale mit der negativen Taktflanke hat den Vorteil, dass eine ABORT Situation umgangen wird, siehe Kapitel 2.1.

Bei einem Lesezugriff auf die Adresse 0x01 wird ein Pseudo-Schreibzugriff auf das Register `data_register` ausgeführt. Hierbei wird `wrdata` ignoriert und stattdessen das Register mit einem Dummywort 0xFF gefüllt, da ein Lesezugriff nur durch togglen von `csib` und `rdwrb` geschieht. Analog zum Schreibzugriff wird ein Lesezugriff durch das Signal `read_req` signalisiert.

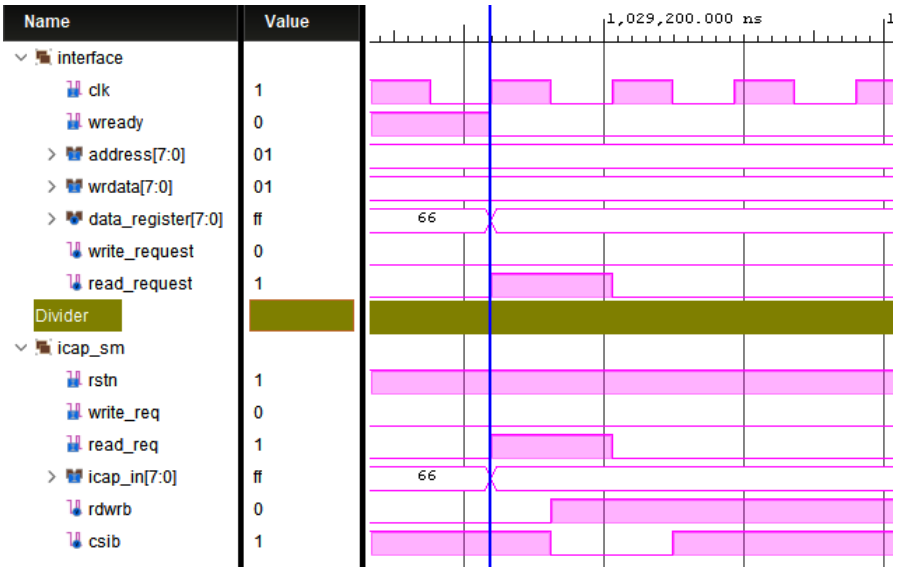


Abbildung 2.8: ICAP Lesevorgang

Gültige Lesedaten liegen drei Taktzyklen nach Aktivierung von `csib` am Ausgang an und können mit dem ILA beobachtet werden.

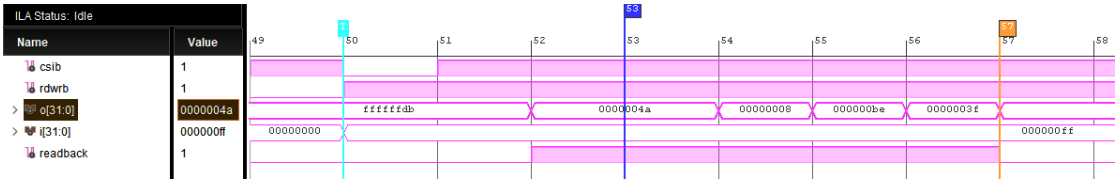


Abbildung 2.9: ICAP Lesevorgang aufgenommen mit ILA

Zusätzlich wird ein Signal `busy` erzeugt, welches intern den Namen `readback` trägt. Es ist solange high, wie das ICAP Primitiv Daten auf den Ausgangsbus legt. Dabei wird ausgenutzt, dass während eines normalen Lesevorgangs kein ABORT-Statusbyte am Ausgang anliegt.

2.4.2 UART Interface

Das bereitgestellte UART Interface kommuniziert mit einer Geschwindigkeit von 115200 Baud bei Verwendung eines Start- und Stoppbits und ohne Parität. Jedes Bit wird 100 mal abgetastet. Daraus ergibt sich ein Takt von 11,52MHz. Da es sich beim UART Interface um das langsamte Glied in der Kette handelt, wird der UART Takt als Systemtakt verwendet. Das UART Interface informiert die Protocol Unit über den Empfang einer neuen Nachricht mit dem Signal `new_data_rx`. Die empfangene Nachricht wird auf dem Bus `data_rx` abgelegt.

2.4.3 Clocking Wizard

Um den benötigten Takt von 11,52MHz zu erzeugen, wird auf die Clocking Wizard IP von Xilinx zurückgegriffen.

2.4.4 Protocol Unit

Für die Steuerung des Verhaltens des ICAP Controllers wurde ein byteorientiertes Protokoll entworfen um Schreib und Leseoperationen zu ermöglichen. Die Protokollunit ist hierbei für das Dekapsulieren der Nachricht zuständig. In Tabelle 2.7 ist das Rahmenformat einer Nachricht dargestellt.

Flag	**Address**	**DLC**	**Data**	**Flag**
0x7E	RRRRRRRx	255 - 0	xxxxxxxx	0x7E

Tabelle 2.7: Rahmenformat einer Nachricht

Die beiden Flag Bytes geben den Start und das Ende einer Nachricht an. Von den acht verfügbaren Bits im Adressfeld wird momentan nur eins genutzt. Die restlichen Bits sind als Nullen anzusehen. Ein Schreiben auf Adresse 0x00 startet eine Schreiboperation. Ein Schreiben auf Adresse 0x01 eine Leseoperation. Das Data Lenght Code (DLC) Feld gibt die Anzahl der zu sendenden Bytes an. Bei einer Leseoperation kann das DLC Feld auf null gesetzt werden, da jegliche Daten ignoriert werden. Die Länge des Datenfelds, welche im DLC Feld angegeben wird, kann bis zu 255 Bytes umfassen. Im folgenden ist das Zustandsdiagramm der Decapsulation Unit dargestellt:

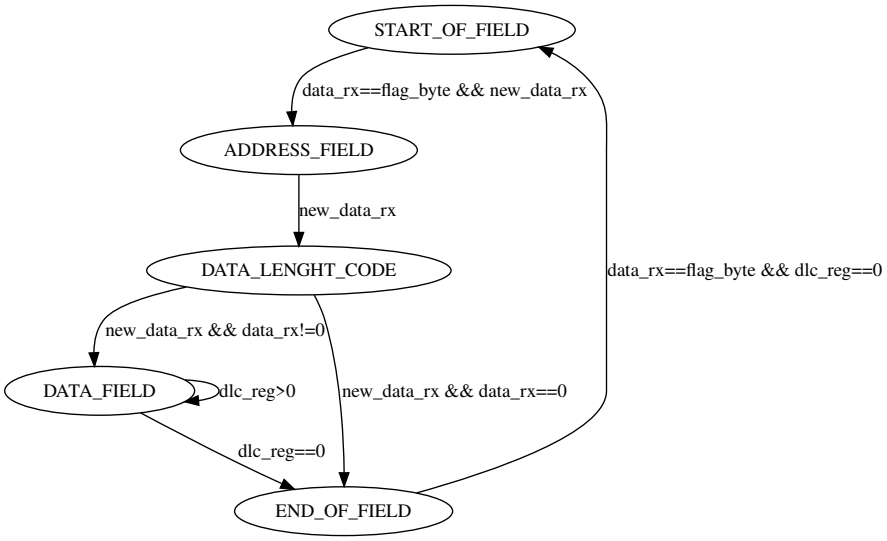


Abbildung 2.10: Zustandsdiagramm der Decapsulation Unit

Mit jedem Empfang einer neuen UART Nachricht wird die Statemachine aktiv. Zu Beginn befindet sich diese im Zustand `START_OF_FIELD`. Mit jedem neu empfangenen Datenwort wird geprüft, ob es sich um ein Flag Byte handelt. Ist dies der Fall, wechselt die Statemachine in den Zustand `ADDRESS_FIELD` und wartet auf den Empfang eines weiteren Bytes. Das empfangene Address Byte wird auf den Ausgang `address` gelegt und der Zustand um eins auf `DATA_LENGHT_CODE` inkrementiert. Im Zustand `DATA_LENGHT_CODE` wird das empfangene DLC Byte gespeichert und als Zähler für die kommenden Data Bytes verwendet. Entspricht das DLC Byte null, so ändert sich der Zustand zu `END_OF_FIELD`. Entspricht das DLC Byte ungleich null, so wechselt der Zustand zu `DATA_FIELD`. Im Zustand `DATA_FIELD` wird der Zählerstand bei jedem empfangenen Byte um eins dekrementiert. Das empfangene Byte wird auf den Ausgang `data` gelegt. Die zuvor empfangene Adresse bleibt erhalten. Das Signal `data_received` wird high, um den ICAP Controller über den Empfang eines neuen Data Bytes zu informieren. Erreicht der Zählerstand den Wert null, wird in den Zustand `END_OF_FIELD` gewechselt. Sobald ein Flag Byte empfangen wurde, wechselt der Zustand wieder zu `START_OF_FIELD`, wo der Empfang einer neuen Nachricht abgewartet wird. Die Überspringung von `DATA_FIELD` ist für den Fall einer Leseoperation. Wie bereits erwähnt sollte das DLC Feld in diesem Fall null entsprechen.

2.4.5 Decoupling

Die Decoupling Unit übernimmt die Entkopplung der Ein- und Ausgänge der Partition. Wie bereits in Abbildung 2.2 gezeigt, handelt es sich auch hier um einen 2 in 1 Multiplexer. Als Enable Signal dient das `EOS`-Signal des STARTUPE2 Primitives.

2.4.6 Partition

Hierbei handelt es sich um die partiell rekonfigurierbare Partition. Es wurden zwei verschiedene rekonfigurierbare Module erstellt. Eine Anleitung findet sich im Anhang A.1. Es handelt sich um zwei 16 Bit-Schieberegister zur Visualisierung der partiellen Rekonfiguration, von denen eines nach links und das andere nach rechts schiebt.

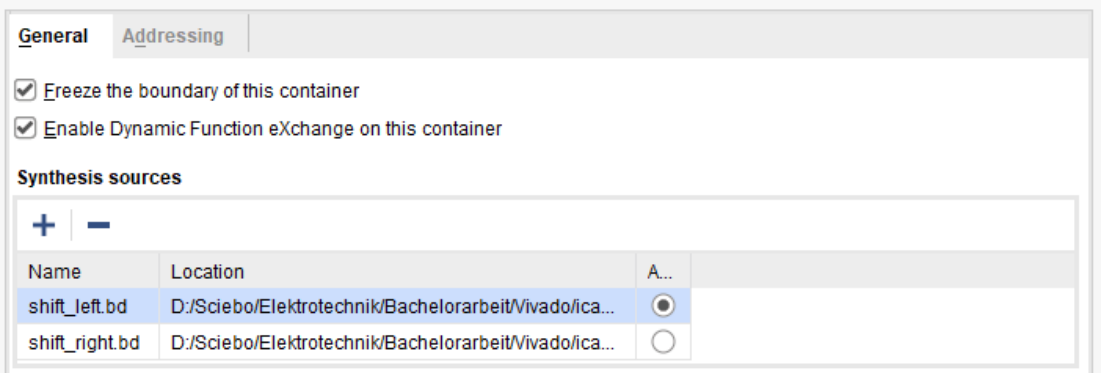


Abbildung 2.11: Rekonfigurierbare Module der Partition

Insgesamt wurden vier Bitstreams erzeugt. Jeweils zwei vollständige Bitstreams, die entweder `shift_left` oder `shift_right` standardmäßig in der Partition enthalten. Zusätzlich gibt es zwei partielle Bitstreams für `shift_left` und `shift_right` mit denen die Partition rekonfiguriert werden kann.

2.4.7 Startupe2

In diesem Block ist die STARUPE2 Primitive instanziiert. Sie dient zur Erzeugung des `EOS` Signals (Siehe Kapitel 2.2), welches als Enable Signal für die Decoupling Unit dient.

2.4.8 Testbench

Für den Testentwurf wurde eine Testbench geschrieben, welche mittels einer Task Funktion Nachrichten nach dem in Tabelle 2.7 dargestellten Schema assembliert. Ein weiterer Task ist für die Serialisierung der Nachricht und die UART Rahmenbildung zuständig. Wie bereits in Kapitel 2.1 erwähnt, konnte jedoch keine Simulation der ICAP Primitive durchgeführt werden, weshalb die Simulation anhand des Verhaltens in Abbildung 2.3 abgeglichen und später das Verhalten mit dem ILA untersucht wurde.

2.5 ICAP Benutzeroberfläche

Im diesem Kapitel soll die grafische Oberfläche zur Versendung von Bitstreamdaten näher vorgestellt werden. Es handelt sich hierbei um ein Python Skript, welches sich auf die Bibliotheken pySerial sowie PySide6 stützt. Die Bibliothek pySerial ermöglicht die Kommunikation mit seriellen Schnittstellen in Python. Die Bibliothek PySide6 wird für die Erstellung von grafischen Benutzeroberflächen unter der Verwendung von Qt in Python verwendet. Qt ist ein plattformübergreifendes Toolkit, welche für die Erstellung von Desktop- sowie und mobilen Anwendungen verwendet wird. Im folgenden ist die Benutzeroberfläche gezeigt.

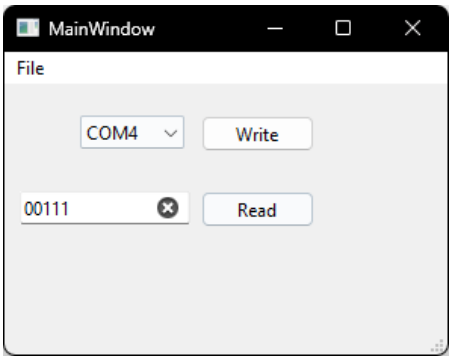


Abbildung 2.12: ICAP Benutzeroberfläche

Unter dem Menüpunkt `File` kann eine partielle Bitstream Datei geladen werden. Es werden nur Bitstreams im BIN und BIT Format akzeptiert. Die Bitreihenfolge wird automatisch verändert. Mit dem Reiter neben dem Button `Write` kann der entsprechende COM Port ausgewählt werden. Mit `Write` wird die Rekonfiguration gestartet. Im Adressfeld kann die 5 Bit Adresse des Registers spezifiziert werden, welches ausgelesen werden soll, siehe Tabelle 2.1. Das Skript erzeugt automatisch das Type 1 Packet. Mit dem Button `Read` wird der Leseprozess gestartet.

2.5.1 Klasse SerialServer

Die Klasse `SerialServer` ist für das Öffnen und Schließen der seriellen Schnittstelle zuständig. Außerdem sind Methoden zum Lesen und Schreiben von Daten implementiert. Wie bereits im Kapitel 2.4.2 erwähnt, wird mit 115200 Baud kommuniziert.

2.5.2 Klasse GUI

In der Klasse `GUI` (`Ui_MainWindow`) ist die Benutzeroberfläche implementiert. In ihr finden sich alle nötigen Elemente, um die Benutzeroberfläche anzusprechen.

2.5.3 Klasse App

Die Klasse `App` bildet das Hauptgerüst des Programms. In ihr sind die Funktionen für das Schreiben und Lesen implementiert, welche beim Drücken der Buttons `Write` oder `Read` ausgeführt werden. Wenn eine Bitstream Datei geöffnet wurde, wird diese in 255 großen Bytepacketen geladen. Zeitgleich wird die Bitreihenfolge in jedem Byte verändert. Anschließend wird eine Nachricht nach dem Schema in Tabelle 2.7 assembliert und der Klasse SerialSever zum Versenden übergeben.

Bei einem Lesevorgang wird zuerst die 5 Bit Adresse in das Type 1 Packet eingerahmt. Anschließend wird die Befehlsequenz für einen Lesevorgang gestartet und wieder der Klasse `SerialServer` zum schreiben übergeben.

2.5.4 Test-FPGA

Für den Test einer partiellen Rekonfiguration wurde das Nexys 4 DDR Board von Digilent verwendet. Es handelt sich hierbei um das in Kapitel 2.1 erwähnte Artix-7 XC7A100T FPGA. Praktischer Weise existiert auf dem FPGA Board bereits ein FTDI FT2232HQ USB zu UART Brückenbaustein, weshalb das FPGA Board per USB an den PC angeschlossen werden kann. Beim Testentwurf werden die LEDs oberhalb der 16 Schalter zur Visualisierung der partiellen Konfiguration verwendet. Ein rekonfigurierbares Modul schiebt nach links, während das andere nach rechts schiebt. Der Ausgang der Schieberegister sind mit den 16 LEDs verbunden und erzeugen somit ein Lauflicht. Zusätzlich wird der Taster `CPU_RESET` als Reset verwendet.

Um den beschriebenen Testentwurf zu testen, muss zunächst das FPGA über ein USB Kabel mit einem PC verbunden werden. Bevor ein partieller Bitstream über die Benutzeroberfläche auf den FPGA geschrieben werden kann, muss das FPGA vorher mit dem statischen Teil, d.h. mit einem vollständigen Bitstream, über Vivado konfiguriert werden. Nach Abschluss der Konfiguration läuft das LED-Lauflicht je nach Wahl des Bitstreams entweder nach links oder rechts.

In der ICAP Benutzeroberfläche sollte das FPGA als COM Port auftauchen. Wenn nun über die ICAP Benutzeroberfläche ein partieller Bitstream geladen wird, welcher in die gegensätzliche Richtung läuft (Shiftregister links oder rechts), sollte dies nach Abschluss der partiellen Rekonfiguration sichtbar werden. Während der partiellen Rekonfigurations erlischt das Lauflicht für einen Moment. Anhand der Änderung der Richtung des Lauflichtes ist erkennbar, dass eine partielle Rekonfiguration erfolgreich durchgeführt wurde.

2.6 Partielle Rekonfiguration des MOPS-HUB Entwurfes

Im Vorfeld dieser Arbeit wurde bereits eine partielle Rekonfiguration des kompletten MOPS-HUB Entwurfs unter Verwendung des MOPS-HUB FPGAs erfolgreich getestet und durchgeführt. Es muss jedoch erwähnt werden, dass die partielle Rekonfiguration zu diesem Zeitpunkt mit Joint Test Action Group (JTAG) über Vivado durchgeführt wurde, da der Testentwurf sich zu dem Zeitpunkt noch in Entwicklung befand. Da sowohl JTAG als auch ICAP auf die interne Konfigurationslogik zugreifen, lässt sich mit sehr hoher Gewissheit sagen, dass dies auch über das ICAP Interface möglich wäre, da gezeigt werden konnte, dass eine partielle Rekonfiguration mit dem Testentwurf möglich ist.

Wie bereits in Kapitel 2 beschrieben, gibt es bei der Erstellung eines partiellen Designs wenige Limitierungen in Form von taktverändernder Logik, I/O sowie Hardwarespezifischen Primitiven. Der komplette MOPS-HUB Entwurf konnte in einer Partition konfiguriert werden, lediglich die taktverändernde Logik (Clocking Wizard IP) sowie I/O Elemente (SelectIO Wizard IP) mussten in den statischen Teil platziert werden. Daher sind nur wenige Änderungen am Entwurf erforderlich. Im Falle einer partiellen Rekonfiguration über das ICAP Interface müsste der ICAP Controller, eine geeignete Brückenlogik sowie ein Decoupling analog zum gezeigeten Testentwurf für die Verarbeitung von Nachrichten in den MOPS-HUB Entwurf implementiert werden.

Eine partielle Rekonfiguration überschreibt die Logik innerhalb einer Partition, somit würde sich die partielle Rekonfiguration als Methodik zur Bereinigung von Fehlern im FPGA Konfigurationsspeicher anbieten. Jedoch verbleibt der statische Part ungeschützt. Somit wäre es erforderlich, beispielsweise ein strahlenhartes Speichermedium in der Nähe des MOPS-HUB FPGAs zu platzieren, von welchem das FPGA die statische Konfiguration in einem Fehlerfall laden kann. Der eigentliche MOPS-HUB Entwurf könnte über die E-Link Verbindung (Siehe Kapitel 1.1) von einem entfernten Ort an das MOPS-HUB FPGA geschickt und partiell rekonfiguriert werden.

Ein Vorteil der physischen Separierung von statischem und rekonfigurierbarem Teil ist die Erhaltung der Datenintegrität des rekonfigurierbaren Teils, da dieser keine erhöhte Strahlung am Standort PP 3 ausgesetzt wäre. Zusätzlich könnte der MOPS-HUB Entwurf aus der Ferne aktualisiert werden. Ein weiterer Vorteil ist die geringe Größe des statischen Teils, vorrausgesetzt die Bitstream Komprimierung wurde aktiviert sowie die Generierung eines Bitstreams mit einer Greybox Definition der Partition. Da eine Greybox Definition keinerlei Logik enthält, beschränkt sich die Bitstreamgröße nur auf den statischen Part des Entwurfs, welcher im Falle des MOPS-HUB Entwurfs die genannten Intellectual Property (IP)s sowie den ICAP Controller, Decoupling und die E-Link Kommunikationsschnittstelle Logik beschreibt. Es bleibt zu erwähnen, dass keine Aussagen zur strahlenhärte der internen Konfigurationslogik und dem ICAP Interface getätigt werden können, da Recherchen keine sinnvollen Ergebnisse liefern.

3 Triple Modular Redundancy

Bei TMR handelt es sich um eine Methode, um integrierte Schaltungen gegenüber strahleninduzierten Logikfehler durch Fehlertoleranz zu schützen. Der Grundgedanke von TMR basiert auf die Einbringung von Redundanz auf Hardwareebene: Die Resilienz von kritischen Systemen wird durch eine dreifache Replizierung d.h. einer Triplizierung der Logikschaltung gesteigert. In Kombination mit einer Mehrheitsabstimmung als Entscheidungsmechanismus zwischen den Ausgangssignalen der drei Ausführungen der Logikschaltung, auch Voting genannt, kann das System trotz eines Fehlers in einer der drei Ausführungen immer noch eine wahrheitsgemäße Antwort liefern. Tritt jedoch in zwei Ausführungen ein Fehler auf, kann dieser nicht mehr korrigiert werden, sondern wird als korrekt angenommen. Im Falle des MOPS-HUB Entwurfs wird der Wert des Mehrheitsvotums bei jedem Taktzyklus von allen drei Flipflops über einen Feedback Pfad übernommen, um einen potentiellen Fehler zu eliminieren. Ohne diese Rückkopplung würden Fehler zwar erkannt, aber nicht korrigiert. Obwohl TMR eine solide und übliche Methode ist, um ein Design gegen strahlungsinduzierte Fehler zu schützen, hat TMR je nach gewählter Implementierungsstrategie einen starken Anstieg auf die Anzahl benötigter Logikressourcen zur Folge. Außerdem können nicht alle Elemente eines FPGA-Designs geschützt werden. Dazu gehören FPGA spezifische Ressourcen wie beispielsweise das ICAP-Primitiv oder ein IP, dessen Quellcode nicht offen zugänglich ist. Außerdem gibt es Schwierigkeiten bei der Triplizierung von großen Speicherblöcken wie Block Random Access Memory (BRAM). Hier empfiehlt sich z.B. die Verwendung einer Kanalcodierung in Kombination mit einem Speichercontroller zur aktiven Fehlerkorrektur.

Ursprünglich war für den MOPS-HUB Entwurf eine volle Triplizierung des MOPS-HUB Entwurfs bei 32 CAN Bussen angedacht, da der im MoPS verwendete CANakari bereits vollständig tripliziert ist. Dieser Ansatz wurde jedoch schnell verworfen, da frühe Ergebnisse zeigten, dass die benötigten Logikressourcen die Kapazität der MOPS-HUB FPGA nahezu ausreizen und übersteigen werden. Zu diesem Zeitpunkt war das TMR Voting bisher nur im CANakari Design und nicht im MOPS-HUB Entwurf selbst implementiert:

Ressource	**Verfügbar**	**MOPS-HUB**
Lookup Table (LUT)	133800	121045 - 90,47%
Flipflop (FF)	269200	76014 - 28,24%

Ressource	Verfügbar	MOPS-HUB
Global Clock Buffer (BUFG)	32	7 - 21,88%

Tabelle 3.1: Ressourcenbedarf des vollständig triplizierten MOPS-HUB Entwurfs mit 32 CAN Bussen auf der MOPS-HUB FPGA.

Aus diesem Grund wurde eine speziellere Triplizierungsart gewählt, welche nur die Speicherinhalte schützt bzw. korrigiert. Da der CANakari Controller einen Großteil des MOPS-HUB Entwurf ausmacht, siehe Tabelle 1.1, kann nicht auf die bereits bestehende TMR Implementation des CANakari zurückgegriffen werden. Des Weiteren wurde die Anzahl der CANakaris von 32 auf 16 halbiert. In Abbildung 3.1 ist schematisch der gewählte Ansatz der partiellen Implementierung von TMR gezeigt.

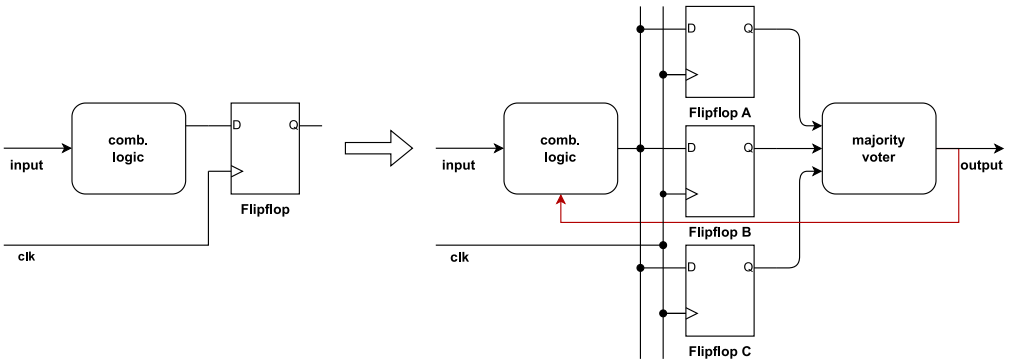


Abbildung 3.1: Schematische Darstellung der partiellen TMR Implementierung des MOPS-HUB Entwurfs samt CANakari

Diese Art von TMR reduziert zwar den Bedarf an LUTs erheblich, da die Mehrheitsentscheidung und die kombinatorische Logik nicht mehr dreifach ausgeführt ist. Dies führt jedoch zum Nachteil, dass beim Auftreten eines SET im Abstimmungsnetzwerk oder in der kombinatorischen Logik alle drei Speicherinhalte korrumpiert werden, wenn im selbigen Moment auch die Flipflops einen neuen Wert übernehmen. Es bleibt abzuwarten, ob diese Triplizierungsstrategie am Standort PP3 einen ausreichenden Schutz bietet. Zum aktuellen Zeitpunkt der Arbeit kann keine Abschätzung der Fehlerrate Werte genannt werden, da bisher noch keine Untersuchungen unternommen wurden.

In Tabelle 3.2 ist der Ressourcenbedarf der zum Zeitpunkt der Arbeit aktuellsten Version des MOPS-HUB Entwurfs mit 16 CAN Bussen mit und ohne partielle TMR Implementierung gezeigt. Bei den gezeigten Ergebnissen handelt es sich bereits um ein implementiertes Design.

Ressource	Verfügbar	MOPS-HUBTMR	TMRErr	MOPS-HUB
Lookup Table (LUT)	133800	54536 40,76%	81779 61,12%	32767 24,49%
Flipflop (FF)	269200	56723 21,07%	- -	18222 6,77%
Global Clock Buffer (BUFG)	32	1 3,13%	- -	1 3,13%

Tabelle 3.2: Ressourcenbedarf des aktuellen partiell triplizierten MOPS-HUB Entwurf mit 16 CAN Bussen auf der MOPS-HUB FPGA.

Erwartungsgemäß nach Abbildung 3.1 ist der Ressourcenbedarf der Flipflops um das Dreifache gestiegen. Als Faustformel für die Ermittlung des Zuwachses der benötigten LUTs kann die Anzahl der Flipflops und der LUTs des nicht triplizierten Entwurfs addiert werden. Dies hat den Hintergrund, dass für jedes Flipflop, welches tripliziert wird, eine LUT mit drei Eingängen für die Mehrheitsabstimmung verwendet wird, wie in Abbildung 3.2 zu sehen ist. Es ist jedoch zu beachten, dass eine Votinginstanz in der Regel noch ein Fehlersignal *tmrErr* aufweist. Der zusätzliche Ressourcenverbrauch für die Implementierung des Fehlersignals ist in Tabelle 3.2 in der Spalte „TMRErr“ angegeben. Weitere Informationen zum Error Signal finden sich in Kapitel 3.1.3.

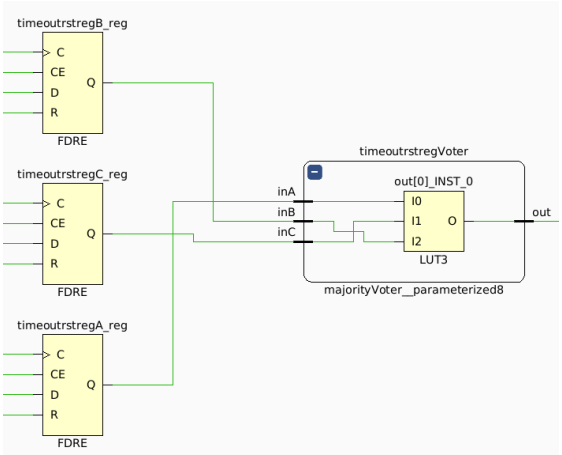


Abbildung 3.2: Implementierter Voter

Für die Anwendung von TMR auf den MOPS-HUB FPGA Entwurf sind eigene Skripte entwickelt worden, deren Zielsetzung es ist, den zeitlichen und manuellen Aufwand zu reduzieren.

Da der MOPS-HUB Entwurf sich zum Zeitpunkt der Arbeit noch aktiv in der Entwicklung befindet, mussten Lösungen gefunden werden, um unabhängig vom aktuell im HDL-Designer eingepflegten Entwicklungsstand des MOPS-HUB Entwurfs, schnell Ergebnisse zu liefern, an Hand derer die Umsetzbarkeit und der Ressourcenaufwand bewertet werden kann.

3.1 Triple Modular Redundancy Generator Toolkit

Für die Umsetzung einer Triplizierung muss der HDL Quellcode eines digitalen Systementwurfs angepasst werden. Eine manuelle Implementierung einer dreifachen Strukturierung ist je nach Designgröße jedoch sehr zeitintensiv, fehleranfällig und komplex. Zur Unterstützung wird deshalb auf ein Toolkit Namens Triple Modular Redundancy Generator (TMRG)[6] zurückgegriffen, welches am CERN entwickelt worden ist [12]. Das Toolkit ist eine Sammlung von Python basierten Werkzeugen, die den Prozess einer validen TMR Implementierung unterstützen. Namensgebend und Hauptbestandteil des Toolkits ist das TMRG Tool zum generieren von triplizierten Verilog Code. Für die Unterstützung bei der Validierung sowie Implementierung eines TMR Designs bietet das Toolkit die folgenden Routinen:

- **Single Event Effects Generator (SEEG):**
  Der SEEG erzeugt aus einer finalen Netzliste eines TMR Designs eine Verilog-Datei mit mehreren Verilog Tasks. Die Tasks ermöglichen die Simulation von SET und SEU durch die Erzwingung von vorgegebenen Werte an Netzen und in Flipflops mittels der Verilog *force* Anweisung.

- **Placement Generator (PLAG):**
  Dieses Skript unterstützt die physische Platzierung und Distanzierung von triplizierten Logikzellen.

- **Testbench Template Generator (TBG):**
  Dieses Skript erzeugt eine exemplarische Verilog Testbench, welche auf den vom SEEG generierten Tasks aufbaut.

Das TMRG Toolkit ist zwar ursprünglich für die Anwendung auf ASIC Designs ausgerichtet, kann jedoch auch für die Triplizierung von FPGA Entwürfen verwendet werden [12]. Die oben genannten Routinen spielen für die Bearbeitung des Themas eine untergeordnete Rolle, wurden aber der Vollständigkeit halber erwähnt. Weiterhin ist zu erwähnen, dass mit den von Vivado generierten Verilog Netzlisten keine brauchbaren Ergebnisse erzielt werden konnten.

[6] Zum Zeitpunkt der Arbeit wurde die aktuellste Version „e2805f5c6799c210349d674807823064dd88f4a5“ benutzt. Probleme die bei der Bearbeitung aufgetreten sind, können bereits behoben sein

Es wird eine Standard Cell Library für Flipflops und Latches benötigt, welche eher für ASIC Designs üblich sind. Obwohl der HDL-Code in einem FPGA Design auch in logische Elemente wie Gatter und Flipflops übersetzt wird, gibt es an sich keine Standardzellenbibliotheken sondern Bibliotheken mit FPGA Primitiven. Im Vivado Installations Verzeichnis finden sich zwar Bibliotheken für die Flipflop und Latch Primitive, jedoch erlaubt das Format dieser Bibliotheken keine automatische Anwendung des SEEG Tools. Zudem sind die oben erwähnten Routinen zum Zeitpunkt der Arbeit wenig dokumentiert.

3.1.1 Triple Modular Redundancy Generator

Das TMRG Tool ist in der Lage, Verilog HDL Code zu parsen und eine triplizierte Version nach Vorgaben des Benutzers zu generieren. Die Vorgaben beziehen sich hierbei auf die Elemente eines Entwurfs, welche verdreifacht werden sollen, sowie bei welchen Signalen ein Voting stattfinden soll. Realisiert werden die Vorgaben über eine Festlegung von Direktiven, welche innerhalb jedes Verilog Moduls und oder in einer optionalen Konfigurationsdatei platziert werden. Des Weiteren gibt es die Möglichkeit, diese in Form von Kommandozeilenargumenten bei der Ausführung des Skriptes zu übergeben. Dieser Ansatz wurde aber im Rahmen dieser Arbeit wegen der geringen Flexibilität nicht weiter verfolgt.

Eine Triplizierung geschieht immer innerhalb eines Verilog Moduls. Es müssen alle Modulinstanzen zur Laufzeit des TMRG Tools bekannt sein. Im Falle eines Blocks innerhalb eines Entwurfs, dessen HDL Quellcode nicht bekannt ist, muss eine Moduldefinition der Ein- und Ausgangsports angegeben werden. Ein vom TMRG Tool verarbeitetes Modul trägt das Suffix `TMR` im Modulbezeichner. Triplizierte Elemente tragen die Suffixe `A`, `B`, `C`.

Das TMRG Tool reagiert sehr empfindlich auf falsch gesetzte Direktiven. Zum Zeitpunkt der Erstellung dieser Arbeit existieren Verilog Ausdrücke, welche das Tool nicht verarbeiten kann. Oftmals wird zwar trotzdem Verilog Code generiert, dieser ist aber nicht synthetisierbar und fehlerhaft. Diese Probleme werden im weiteren Verlauf der Arbeit vereinzelt wieder aufgegriffen und Lösungsansätze aufgezeigt.

3.1.2 TMRG Direktive und Konfigurationsdatei

Im folgenden soll eine Übersicht zu den Direktiven und der Konfigurationsdatei des TMRG Tools gegeben werden, wie sie auch für die Triplizierung des MOPS-HUB Entwurfs verwendet werden. Für ein tieferes Verständis wird jedoch auf die TMRG Dokumentation verwiesen [12].

Eine Direktive wird in Form eines Verilog Kommentars beginnend mit `//` und gefolgt vom Schlagwort `tmrg` angegeben. Somit werden folglich alle Kommentare, welche mit `tmrg` starten vom Tool als eine Anweisung gewertet.

Das `default` Direktiv in Quelltext 3.1 steuert das Standardverhalten für das gesamte Verilog Modul. Der Parameter `triplicate` oder `do_not_triplicate` gibt an, ob standardmäßig eine Triplizierung stattfinden soll oder nicht.

```
1 //tmrg default triplicate/do_not_triplicate
```

Quelltext 3.1: Steuerung des Standardverhalten

Ausnahmen zum obigen Standardverhalten können mittels den Anweisungen aus dem Quelltext 3.2 erzeugt werden. Als Beispiel dienen die zwei Signal `cnt` vom Typ *reg* und `state` vom Typ *wire*.

```
1 reg cnt;
2 wire state;
3 //tmrg triplicate cnt
4 //tmrg do_not_triplicate state
```

Quelltext 3.2: Ausnahmen des Standardverhaltens

Während mit der `default` Anweisung zwar die Triplizierung eines Moduls unterbunden werden kann, wird das entsprechende Modul von TMRG dennoch verarbeitet. Um dies zu unterbinden, muss das Direktiv `do_not_touch` platziert werden. Dies ist vor allem hilfreich, wenn sich der Modulbezeichner nicht verändern darf und keine Triplizierung gewünscht ist:

```
1 //tmrg do_not_touch
```

Quelltext 3.3: Ausschließen eines Moduls

Die Direktive `tmr_error` macht das Error Signal der Voter Instanzen für die nächsthöhere Hierarchieebene zugänglich. Wenn mehrere Voter in einem Modul instanziiert wurden, werden die Signale mit einer ODER Funktion verknüpft. Eine Deklaration von `tmrError` macht das Error Signal bereits im nicht triplizierten Zustand und innerhalb des Moduls nutzbar. Die Zuweisung `0` dient zur Fehlervermeidung, da im nicht triplizierten Zustand `tmrError` noch keine Funktion erfüllt. Weitere Details zur Voter Definition und dem Error Signal finden sich im Kapitel 3.1.3. Vorab soll erwähnt werden, das mit der aktuellsten Version des TMRG Toolkit zum Zeitpunkt der Arbeit fehlerhafter Verilog Code bei Verwendung der `tmr_error` Direktive produziert wird.

```
1 //tmrg tmr_error True/False
2 wire tmrError=1'b0;
```

Quelltext 3.4: Nutzung des tmrError Signals

Zwar kann bereits mit den gezeigten Anweisungen die Triplizierung eines kompletten Designs vollständig beschrieben werden, jedoch ist es in vielen Fällen sinnvoll eine Konfigurationsdatei zu erstellen. In einer Konfigurationsdatei können dieselben Direktive beschrieben werden. Darüber hinaus bietet die Verwendung einer Konfigurationsdatei die Möglichkeit, globale Einstellungen in Bezug auf das Verhalten des TMRG Tools zu wählen. Dies ist vor allem bei komplexeren Entwürfen, deren Hierarchie aus einer Vielzahl an Modulen besteht, nützlich.

In Quelltext 3.5 ist das Schema einer Konfigurationsdatei dargestellt. Sie ist in die Bereiche `[tmrg]`, `[global]` und `[modul]` aufgeteilt.

```
1  [tmrg]
2  tmr_dir =
3  rtl_dir =
4  sdc_generate = True/False
5  files =
6
7  [global]
8  default = triplicate/do_not_triplicate
9  tmr_error = True/False
10 top_module =
11
12 [modul]
13 default : triplicate/do_not_triplicate
14 element : triplicate/do_not_triplicate
```

Quelltext 3.5: Schema der Konfigurationsdatei

Der `[tmrg]` Block ermöglicht die Übergabe von Parametern, die sonst nur über die Kommandozeile übergeben werden können. `tmr_dir` sowie `rtl_dir` in Zeile 2 und 3 spezifiziert die Speicherpfade für den Output sowie den Input der Verilog Quelldateien. Mit der Option `sdc_generate` wird die Generierung einer Synopsys Design Constraints (SDC) Datei gesteuert, die dem Synthesetool zur Vermeidung der Zusammenführung vermeintlicher redundanter Logik übergeben werden kann. Da sich diese Liste jedoch oft als unvollständig und fehlerhaft erwies, wird diese Option nicht genutzt (Siehe Kapitel 3.5). Mit `files` können weitere Verilog Quelldateien, welche nicht im Pfad `rtl_dir` liegen, hinzugefügt werden. Wenn eine Liste von Dateien angegeben werden soll, müssen die Einträge dieser Liste mit einem Leerzeichen seprariert werden

Der `[global]` Block steuert das Verhalten über die komplette Designhierarchie anstatt wie bisher nur innerhalb eines Moduls. Die `default` Direktive gilt somit auch außerhalb eines einzelnen Moduls. Die Anweisung `tmr_error` ist bereits bekannt und führt bis zum Top-Modul. Mit `top_module` kann das Verilog Top-Modul definiert werden.

Dies ist aber selten von Nöten, da das TMRG Tool selbst in der Lage ist, das Top-Modul zu ermitteln.

Der `[modul]` Block beinhaltet die Direktiven, welche innerhalb des Moduls gelten. `modul` steht hierbei als Platzhalter für den eigentlichen Modulbezeichner. Die Direktiven bleiben die gleichen, jediglich die Schreibweise ändert sich.

Es bleibt zu erwähnen, das sich die Direktive innerhalb der Konfigurationsdatei nicht mit denen im Verilog Quellcode überschneiden sollten. Dies wird zu Fehlern bei der Ausführung des TMRG Tools führen und somit in fehlerhaften Verilog Code resultieren.

3.1.3 TMRG Voting und Fanout

Bisher wurde der Begriff Voting benutzt, ohne detailliert auf diesen einzugehen. Wenn ein nicht tripliziertes Signal einem triplizierten Signal zugewiesen wird, platziert das TMRG Tool automatisch einen Fanout. Es handelt sich hierbei um eine einfache Zuordnung eines Signals auf die drei Signale mit den Suffixen `A`, `B`, `C`, welche durch die Triplizierung entstanden sind.

Wie bereits erwähnt, reicht eine reine dreifach Strukturierung eines Designs nicht aus, um diesen vor strahleninduzierten Logikfehlern zu schützen. Erst in Kombination mit einer Mehrheitsabstimmung zwischen den drei Komponenten und dem Mehrheitsvotum als Feedback können Fehler korrigiert und eine Fehlerfortpflanzung verhindert werden. Im folgenden ist die Voter Definition gezeigt. Diese wurde für das MOPS-HUB Projekt leicht verändert.

```
1  module majorityVoter #(
2    parameter WIDTH = 1
3  )(
4    input wire  [WIDTH-1:0] inA,
5    input wire  [WIDTH-1:0] inB,
6    input wire  [WIDTH-1:0] inC,
7    output wire [WIDTH-1:0] out,
8    output reg              tmrErr
9  );
10   assign out = (inA&inB) | (inA&inC) | (inB&inC);
11   `ifdef tmrErr
12   always @(inA or inB or inC) begin
13     if (inA!=inB || inA!=inC || inB!=inC)
14       tmrErr = 1;
15     else
16       tmrErr = 0;
17   end
18   `endif
```

```
19 endmodule
```

Quelltext 3.6: Voter Definition

Neben dem Mehrheitsvotum verfügt eine Votinginstanz über ein Fehlersignal `tmrErr`. Sobald ein Signal von den beiden anderen abweicht, wird das Signal `tmrErr` auf High gesetzt. Somit signalisiert `tmrErr` einen SEU in einem der drei Flipflops. In Verbindung mit einem Zähler könnte somit die SEU Rate bestimmt werden. Neu hinzugefügt wurde die Compiler Anweisung `'ifdef`, um die Implementierung der Fehlersignallogik im Voter zu unterbinden, sofern das Fehlersignal nicht genutzt wird. Diese Änderung wurde vorgenommen, um die Synthese- sowie Simulationszeit zu verringern. Zusätzlich ist es mit in dieser Arbeit verwendeten TMRG Toolkit Version[6] nicht möglich, das Error Signal sinnvoll zu nutzen, da die Verwendung des `tmr_error` Direktivs fehlerhaften Verilog Code erzeugt. Es werden invalide Modulinstanzen generiert. Des Weiteren findet das Fehlersignal im MOPS-HUB Entwurf keine Verwendung.

In Quelltext 3.7 ist ein Beispiel anhand des `erbcount2` Moduls des CANakari aufgeführt. `erbcount2TMR` besitzt nur den Port `tmrError`.

```
1  module erbcount2TMR(
2    input wire  clock ,
3    input wire  reset ,
4    input wire  elevrecb ,
5    output reg  erb_eq128 ,
6    output  tmrError
7  );
8  [...]
9  assign tmrError =  counterTmrError|edgedTmrError;
10 ------------------------------------------------
11 erbcount2TMR erb_count (
12     .clock(clock),
13     .reset(resetsig),
14     .elevrecb(elevrecb),
15     .erb_eq128(erb_eq128_i),
16     .tmrErrorA(erb_counttmrErrorA),
17     .tmrErrorB(erb_counttmrErrorB),
18     .tmrErrorC(erb_counttmrErrorC)
19     );
```

Quelltext 3.7: Fehlerhafte Modulinstanzen bei Verwendung des `tmr_error` Direktivs

In Abbildung 3.3 ist die gleiche Voting Zelle wie in Kapitel 3 dargestellt nur dieses mal mit dem zugehörigen Error Signal.

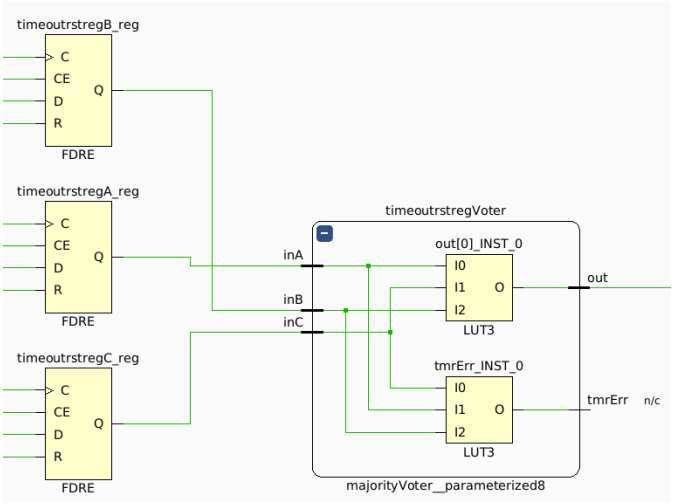


Abbildung 3.3: Implementierter Voter wie in Abbildung 3.2, mit Error Signal

Für die Instanziierung der Voter gibt es zwei verschiedene Ansätze. Wenn ein tripliziertes Signal einem nicht triplizierten Signal zugewiesen wird, wird automatisch ein Voter an diesem Signal platziert. Für den Fall, dass der Voter selbst in einer dreifacher Ausführung existieren soll, muss folgende Zuweisung getätigt werden:

```
1 reg cnt;
2 //tmrg triplicate cnt
3 wire cntVoted = cnt;
```

Quelltext 3.8: Einführung von Voting Instanzen

Sobald ein tripliziertes Element, hier `cnt`, einem Signal zugeordnet wird, welches den selben Bezeichner und den Suffix `Voted` trägt, werden drei Voter `cntVoted`A, `cntVoted`B, `cntVoted`C instanziiert.

3.2 Validierung von TMR

Aufgrund der gewählten TMR Implementierung des MOPS-HUB Entwurfs ist zu erwarten, dass der Ausgang jedes Flipflops mit einer Voterzelle verbunden ist, siehe Abbildung 3.1. Ein guter Indikator ist hierbei der Blick auf den Ressourcenbedarf des TMR Designs. Bei einer validen TMR Implementierung sollte der Bedarf an Flipflops um den Faktor drei angestiegen sein, wie bereits in Kapitel 3 diskutiert.

Für die Unterstützung der Validierung wurde ein Tool Command Language (tcl) Skript namens reg2 geschrieben, welches durch die Netzlisten von Vivado iteriert und das Ausgangsnetz jedes Flipflop Primitivs untersucht. Ist das Ausgangsnetz eines Flipflops mit einer Voterzelle verbunden, wird das Primitiv zunächst als tripliziert angesehen. Es werden die Vivado *get_ ** Befehle verwendet, um Informationen aus der Netzliste zu extrahieren. Für weitere Informationen über die *get_ ** Befehle und ihre Parameter wird auf die Vivado tcl Dokumentation verwiesen[24].

Zunächst soll der Output des Skriptes gezeigt werden:

```
1 <modulbezeichner, bezeichner>
2 monitor_busTMR , mopshub_top_16bus0/monitor_pp30/renaC_reg
```

Quelltext 3.9: Output des reg2 tcl Skript

Wenn ein Primitiv als tripliziert befunden wurde, wird der Output in eine Datei *triplicated_ *.txt* abgespeichert. Wenn das Primitiv als nicht tripliziert bewertet wurde, in die Datei *not_ triplicated_ *.txt*. Neben dem jeweiligen Element wird auch der zugehörige Modulbezeichner kommasepariert aufgeführt. Die Schrägstriche \ im Elementbezeichner stehen jeweils für eine Hierarchiestufe.

Das Skript durchläuft jeweils die Elaborated (Ausgearbeitete), Synthesized (Logische) und Implemented (Physische) Netzliste. Es werden somit sechs Listen erstellt:

- **Elaborated Netzliste:**
  triplicated_rtl.txt, not_triplicated_rtl.txt
- **Synthesized Netzliste:**
  triplicated_synth.txt, not_triplicated_snyth.txt
- **Implemented Netzliste:**
  triplicated_impl.txt, not_triplicated_impl.txt

Die Elaborated Netzliste basiert direkt auf dem vom Benutzer erstellten Verilog Code. Lediglich nicht verwendete Logik wird entfernt. Sie eignet sich daher besonders gut zum Vergleich mit der Synthesized oder Implemented Netzliste, da durch einen einfachen Abgleich festgestellt werden kann, ob bei der Synthese oder Implementierung Elemente entfernt wurden. Vivado interpretiert die Elemente einer Elaborated Netzliste anders als die der Synthesized oder Implemented Netzlisten. Deshalb muss zwischen den beiden Primitive Gruppen *RTL_ REGISTER* für die ausgearbeitete Netzliste und *FLOP_ LATCH* für die logische und physische Netzliste unterschieden werden. Leider sind diese Gruppen in den Xilinx Dattenblättern [21][22][1] nicht detailliert dokumentiert. Oft werden Primitive Gruppen beschrieben, welche in Vivado gar nicht existieren oder umgekehrt.

Es ist aber davon auszugehen, das *FLOP_LATCH* alle Primitive vom Typ Flipflop und Latch umfasst, während *RTL_REGISTER* das Pendant für die ausgearbeitete Netzliste ist.

Im folgenden soll näher auf die wichtigsten Eigenschaften des tcl Skriptes eingegangen werden. Mit den Befehlen aus Quelltext 3.10 werden die gesuchten Primtive aus der jeweiligen Netzliste gefiltert und namentlich absteigend in einer Liste sortiert gespeichert:

```
1 set leafcells [lsort -unique -decreasing -dictionary\
2 [get_cells -hierarchical -filter\
3 {IS_PRIMITIVE == true && PRIMITIVE_GROUP == RTL_REGISTER}]]
4 set leafcells [lsort -unique -decreasing -dictionary\
5 [get_cells -hierarchical -filter\
6 {IS_PRIMITIVE == true && PRIMITIVE_GROUP == FLOP_LATCH}]]
```

Quelltext 3.10: Bestimmung des übergeorneten Moduls eines Elementes

Für die Bestimmung des Ausgangnetzes einer Primitiven muss zunächst der Ausgangspin bestimmt werden:

```
1 set pin [get_pins -of_objects [get_cells $leafcell] -filter {
     DIRECTION == OUT}]
```

Quelltext 3.11: Bestimmung der Output Pins

In Vivado besitzt jeder Pin eine Reihe an Eigenschaften. Die Richtung eines Pins ist unter dem Parameter `DIRECTION` angegeben. Für die Bestimmung des Netzes an einem Pin kann die Vivado Funktion *get_nets* verwendet werden:

```
1 set net [get_nets -of_objects [get_pins $pin]]
```

Quelltext 3.12: Bestimmung des Netzes an einem Pin

Um ein Primitiv später einem Verilog Modul zuordnen zu können, in welchem das Element instanziiert ist, wird der Originalbezeichner des Elternteils bestimmt:

```
1 if {[catch {set name [get_property ORIG_REF_NAME\
2                          [get_property PARENT\
3                          [get_cells $leafcell]]]} errorstring
4                       ]} {
5     set name [lindex [find_top] 0]
6 }
```

Quelltext 3.13: Bestimmung des übergeorneten Moduls eines Elementes

Dies erweist sich jedoch als kniffelig, da eine Primitive, keine direkte Eigenschaft für den Bezeichner des übergeordneten Moduls besitzt. Über die Eigenschaft `PARENT` ist aber ein nachvollziehbarer hierarchischer Pfad zum übergeordneten Elternteil gegeben.

Das Elternteil besitzt das Attribut `ORIG_REF_NAME`, welches schlussendlich den originalen Modulbezeichner angibt. Eine besondere Herangehensweise ist jedoch nötig wenn es sich bei dem übergeordneten Modul um das Top-Modul handelt. Da ein Top-Modul an der Spitze der Hierarchie steht, wird es auch nicht instanziiert. Es existiert somit keine `PARENT` Eigenschaft und es würde stattdessen ein Fehler bei der Abfrage generiert werden. Aus diesem Grunde findet zuerst eine `if` Abfrage mit einem `catch errorstring` Statement statt. Die Bedingung wird wahr, wenn ein Element kein Attribut `PARENT` besitzt. Anstatt einen Fehler zu generieren, wird dieser aufgefangen. Wenn dies der Fall ist, wird mit `find_top` eine Liste der Top-Module abgefragt. Das aktive Top-Modul befindet sich hierbei immer im Index 0: `lindex [find_top] 0`.

Neben einer Validierung wird der Output des Skriptes als Input für die automatische Platzierung der Direktiven des TRMG Tools verwendet, siehe Kapitel 3.3.

3.3 TMRG_CFG_GEN Skript

Im Rahmen dieser Arbeit wurde ein Python-Skript entwickelt, welches die notwendigen Direktiven für die Anwendung des TMRG Toolkits aus der von reg2 generierten Elementliste erzeugt und diese in den entsprechenden Verilog Modulen platziert. Die Direktive werden jeweils vor dem ersten *always* Block platziert und standardmäßig mit dem Trennzeichen `//***Voter***` gekennzeichnet:

```
1 //***Voter***
2 //This block was generated by tmrg_cfg_gen
3 //tmrg triplicate reg_i
4 wire [15:0] reg_iVoter = reg_i;
5 //***Voter***
```

Quelltext 3.14: Beispiel einer Direktiv generierung

Zusätzlich wird eine Konfigurationsdatei nach dem Schema in Quelltext 3.5 generiert:

```
1  [tmrg]
2  tmr_dir = ../hdl/output/tmr
3  rtl_dir = ../hdl/output
4  sdc_generate = False
5  files =
6
7
8  [global]
9  default = do_not_triplicate
10 tmr_error = false
11 top_module =
12
13
```

```
14 [accmaskreg2]
15 default = do_not_triplicate
```

Quelltext 3.15: Beispiel einer generierten Konfigurationsdatei

Das Skript trägt den Namen tmrg_cfg_gen und gliedert sich in drei Klassen und eine Hauptfunktion `main`, welche die Funktionalität der einzelnen Klassen zusammenfasst. Die Aufgabe des Skripts ist es, den manuellen Aufwand, der mit der Definition von Direktiven verbunden ist, durch Automation zu reduzieren.

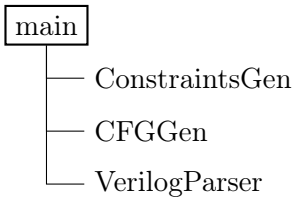


Abbildung 3.4: Struktur des tmrg_cfg_gen Skriptes

Im folgenden wird auf die wichtigste Funktionalität jeder Klasse näher eingegangen.

3.3.1 Klasse ConstraintsGen

Die Python-Klasse `ConstraintsGen` ist für die Generierung von TMRG Toolkit Direktiven zuständig. Die Klasse selbst enthält vier Hilfsmethoden. Die Methode `parse` ist für die Einlesung und Verarbeitung der von reg2 generierten Listen zuständig. Sie selbst ruft die Methode `add_to_dict` auf, welche die Elemente Modulbezeichner und Signalname in einem Python Dictionary speichert. Auf diese Weise kann jedem Modulbezeichner (Key) eine Liste von instanzierten Flipflops (Value) zugeordnet werden. Die Methode `parse` weist hierbei einige Besonderheiten auf:

Der Ausdruck in Quelltext 3.16 ignoriert jegliche Register, welche von Vivado während der Synthese repliziert worden sind und deshalb das Suffix *_rep* tragen. Diese sind im Register Transfer Level (RTL) Code nicht enthalten. Sie sind daher für die weitere Verarbeitung nicht von Interesse.

```
1 "elink_proc_in_dec8b10b, mopshub_top_32bus0/.../ISK_r_reg[1]
     _rep__0"
2
3 if re.search(r"_rep.*$", line):
4     continue
```

Quelltext 3.16: Register Replikation

Unter der Replikation versteht sich ein Optimierungsprozess des Synthesetools, welcher die Aufteilung eines Registers in mehrere Register mit demselben Inhalt bewirkt, um das Timing in zeitkritischen Pfade einzuhalten oder um eine höhere Parallelität zu erreichen [19]. Folgende Abfrage in 3.17 überprüft, ob ein Element einem *generate* block angehört.

```
"tcrc_cell2, mopshub_top_32bus0/.../transmitcrc/genblk1[0].reg_i/
   q_i_reg"

tmp = re.search(r"genblk\d+\[(\d+)]", line)
if tmp and int(tmp.group(1)) > 0:
    continue
```

Quelltext 3.17: Verilog Generate Block

Die *generate* Funktion in Verilog ermöglicht die dynamische Instanzierung von mehreren Modulinstanzen, deren Verhalten durch Parameter verändert werden können. Da im RTL Code jedoch nur eine Instanz von einem *generate* block existiert, werden alle Elemente, welche nicht dem ersten instanzierten *generate* block angehören, ignoriert.

Während der Synthese werden die Zustände und Übergänge von Zustandsautomaten erkannt und extrahiert. Je nach verwendetem Kodierungsalgorithmus ändert sich die synthetisierte Schaltung [27]. Je nach Algorithmus fügt Vivado diesen als Präfix dem Elementnamen hinzu. Der Name des Elements wird mit der Liste `fsm` verglichen. Wenn eine Übereinstimmung vorliegt, wird der Präfix entfernt.

```
"bus_control_SM, mopshub_top_32bus0/.../
   FSM_sequential_current_state_reg[0]"

fsm = ["FSM_onehot_", "FSM_sequential_", "FSM_johnson_",
       "FSM_gray_"]

for fsm in self.fsm:
    if fsm in value[0]:
        value[0] = value[0].split(fsm, 1)[-1]
```

Quelltext 3.18: FSM Extrahierung

Zuletzt wird mit Vergleich 3.19 die Größe eines Registers oder Busses überprüft. Ist kein Suffix für die Tiefe vorhanden, wird die Tiefe auf Null gesetzt, ansonsten wird der höchste Wert übernommen.

```
"top_led_enable_SMTMR, top_led_for_synth0/.../csm_timerC_reg[7]"

value[1] = int(value[1][1:-2]) if value[1][1:-2].isdigit() else 0
```

Quelltext 3.19: Registerlänge Vergleich

Die Methode `constraints_gen` erzeugt aus den Werten des Dictionarys `dict` die bekannten Direktiven sowie die Voter Zuweisungen wie sie in Quelltext 3.14 zu sehen sind.

3.3.2 Klasse: VerilogParser

Die Aufgabe der Klasse `VerilogParser` ist das Parsen von Verilog Code mit Hilfe von regulären Ausdrücken (Regex). Die Klasse selbst enthält vier Hilfsmethoden. Ziel dieser Klasse ist es, den Modulbezeichner sowie die Zeile zu identifizieren, in welcher der erste always Block eines Verilog Moduls beginnt. An dieser Stelle werden später die Direktiven für das TMRG Tool eingefügt. Verilog Kommentare sowie Blockkommentare werden mittels der Methode `__handle_comments` gefiltert.

3.3.3 Klasse: CFGGen

Die Klasse `CFGGen` ist für die Generierung der Konfigurationsdatei zuständig. Diese wird nach dem Schema in `config_template` sowie `module_template` erzeugt. Mit den Hilfsmethoden `add_files` und `add_modules` können Dateinamen sowie Modulbezeichner hinzugefügt werden. Die Dateipfade sowie das *default* Verhalten und weitere Instruktionen können über die Übergabeparameter der Klasse gesteuert werden.

3.3.4 Klasse: Main

Die Klasse `main` fügt die gesamte Funktionalität der Klasse zusammen. Die Funktionen `read_file`, `write_file` und `file_handling` übernehmen das Lesen und Schreiben von Dateien. Die Funktion `fsm_fix` ist eine Zwischenlösung, welche aus einer vorherigen Version des Skriptes übernommen wurde:

```
1 def fsm_fix(lines):
2     re_fsm = re.compile(r"(?i)^\s*(reg\s*\[[0-9]+:[0-9]+])\s*(
          current_state),\s*(next_state)\s*;")
3     for i, line in enumerate(lines[:]):
4         match = re_fsm.match(line)
5         if match:
6             lines[i] = match.group(1) + ' ' + match.group(2) + ';
                  ' + match.group(1) + ' ' + match.group(3) + ';' + '
                  \n'
7     return lines
```

Quelltext 3.20: HDL Designer state register fix

Sie extrahiert aus den Verilog Dateien das Schema `reg [:] current_state, next_state` und splittet diese Deklaration in für das TMRG Tool lesbare Format `reg [:] current_state; reg [:] next_state;`.

3.4 HDL-Designer

Die Entwicklung des MOPS-HUB FPGA Entwurfs findet im HDL-Designer von Siemens EDA statt. HDL-Designer ist eine HDL basiertes Entwicklungswerkzeug und bietet umfassende Unterstützung für Simulation, Verifikation, Visualisierung und grafische Erstellung von digitalen Systemen. Im MOPS-HUB Entwurf wird es vor allem für die abstrakte grafische Planung und Erstellung von Zustands- und Blockdiagrammen verwendet. HDL-Designer ist in der Lage diese Diagramme in Verilog HDL Code umzuwandeln. Mit den vorgefundenen Standardeinstellung ist es jedoch nicht möglich diesen Verilog Code fehlerfrei mit dem TMRG Tool zu parsen. Das Problem lässt sich weitestgehend durch zwei Einstellungen in HDL-Designer lösen. In Abbildung 3.5 sind die nötigen Einstellungen bezüglich der Verilog Generierung gezeigt. Das gezeigte Fenster kann in HDL-Designer unter dem Menüreiter *Option* unter dem Eintrag *Verilog* und dem Reiter *Style* gefunden werden.

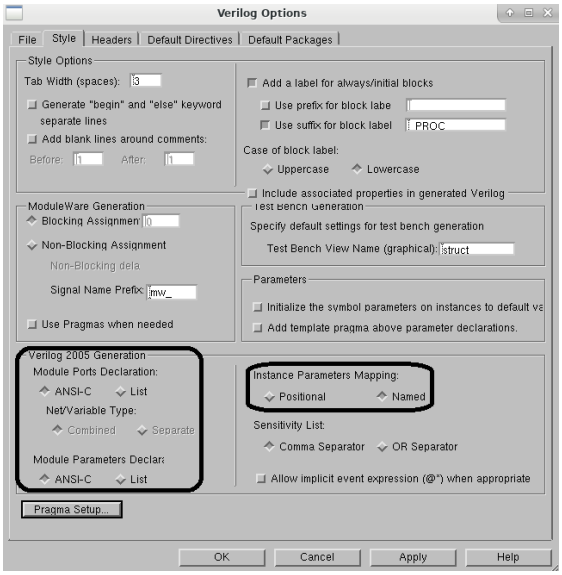


Abbildung 3.5: Verilog Optionen in HDL-Designer

Bei der `ANSI-C` Deklaration handelt es sich um eine neuere Syntaxform der Portdeklaration seit Verilog-2001 (IEE Std 1364-2001)[11]. Es ermöglicht eine komplette Deklarierung eines Ports innerhalb des Modulkopf.

```
1 ANSI-C Style seit Verilog-2001 (IEE Std 1364-2001)
2 module test(
3     input a,
4     output reg b
5 );
```

```
6
7  endmodule
8
9  List Style in Verilog-1995 (IEEE Std 1364-1995)
10 module test (a,b);
11 input [1:0] a;
12 reg b;
13 endmodule
```

Quelltext 3.21: Verilog ANSI-C und List Syntax Deklaration

Obwohl die Dokumentation des TMRG Toolkits [12] beide Syntaxformen erlaubt, führte die Verwendung der ANSI-C-Deklaration bei der Verwendung des TMRG Tools zu einer geringeren Anzahl von Parsingfehlern. Zusätzlich müssen die Parameterwerte innerhalb einer Modulinstanz namentlich zugewiesen werden anstatt der Reihenfolge entsprechend. Bei der Generierung von Statemachines gibt es jedoch noch ein weiteres Problem. HDL-Designer deklariert die Zustandsvektoren von Statemachines in einer Liste statt getrennt. Dies wurde schon im Kapitel 3.3.4 aufgegriffen und wird von dem tmrg_cfg_gen Skript behoben.

3.5 Synthese eines TMR Entwurfs

Aus der Sicht eines Synthesetools ist die Einführung einer dreifachen Redundanz optimierungsbedürftig, da die drei Instanzen identisch zu einander sind und den Ressourcenbedarf erhöhen. Dies hat zur Folge, dass die eigentlich gewünschte redundante Logik während der Synthese reduziert wird. Um dies zu verhindern, wurde im Rahmen dieser Arbeit ein Satz von Constraints für Vivado entwickelt, welche die Reduktion der Logik verhindern. Es werden die Vivado `get_*` Befehle verwendet, um Informationen aus der Netzliste zu extrahieren [24]. Der Vorgang ist hierbei recht simpel. Zunächst werden alle Voterzellen aus der Netzliste gefiltert und mit der Eigenschaft `KEEP_HIERARCHY SOFT` versehen. Interessanterweise listet die Xilinx Vivado Dokumentation jedoch nur *true/false* als mögliche Werte auf [23]. Während laut Dokumentation `KEEP_HIERARCHY TRUE` den Erhalt der benutzerdefinierten Hierarchie im HDL Code einer Instanz erzwingt und strikt keine Optimierung über Hierarchiegrenzen hinweg zulässt, erlaubt `KEEP_HIERARCHY SOFT` eine Optimierung während der Place and Route Phase. Dies hat zur Folge, dass die Hierarchie einer Instanz innerhalb der logischen Netzliste bestehen bleibt, jedoch alle etwaige ungenutzte Elemente innerhalb der Instanz während Place and Route entfernt werden. Im Falle der Voterzelle würde als Beispiel ein nicht verwendetes Error Signal und die Logik des Voters, die das Signal generiert, entfernt werden, während die eigentliche Voting Funktionalität unberührt bleibt. Neben den Voterzellen werden die Netze, welche mit den Eingangspins der Voterzelle verbunden sind, mit der Eigenschaft `KEEP true` versehen.

Das Attribut `KEEP` ist hierbei ähnlich zu `KEEP_HIERARCHY`, bezieht sich jedoch auf ein einzelnes Signal. Da der Ausgang jedes Flipflops mit einem Eingang einer Voterzelle verbunden ist, wird durch das `KEEP` Attribut des Netzes auch die Erhaltung des Flipflops sichergestellt.

```
1  set cells_voter [get_cells -quiet -hierarchical
2                   -filter NAME =~ "*Voter*" &&
3                   IS_PRIMITIVE == false}
4                   ]
5
6  set_property KEEP_HIERARCHY SOFT [get_cells $cells_voter ]
7
8  set pins_voter [get_pins -of_objects [get_cells $cells_voter ]
9                  -filter {NAME =~ "*inA*" || NAME =~ "*inB*" ||
10                 NAME =~ "*inC*"}
11                 ]
12
13 set_property KEEP true [get_nets -segments -of_objects
14                           [get_pins $pins_voter ]
15                           ]
```

Quelltext 3.22: Vivado Constraints

Es empfiehlt sich bei den Constraints in Vivado die Verwendung auf die Synthese zu beschränken:

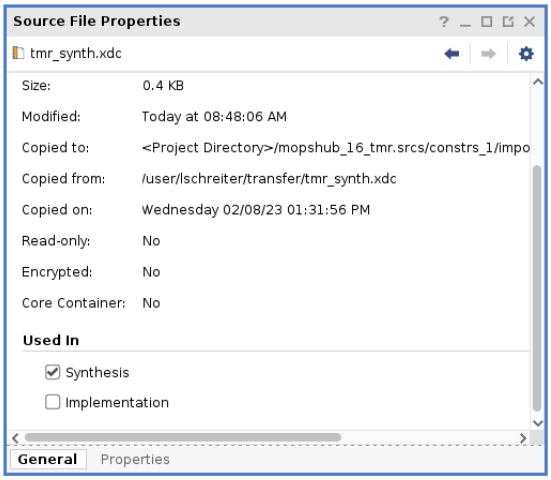


Abbildung 3.6: Beschränkung der Constrainst nur auf die Synthese in Vivado

An Hand eines einfachen Beispiels soll die Wirkungsweise der Constraints gezeigt werden. In Abbildung 3.8 ist ein tripliziertes drei Bit Schieberegister dargestellt.

Die Constraints waren während der Synthese aktiv. Der Verilog Quellcode für die Schaltung findet sich im Anhang A.2.

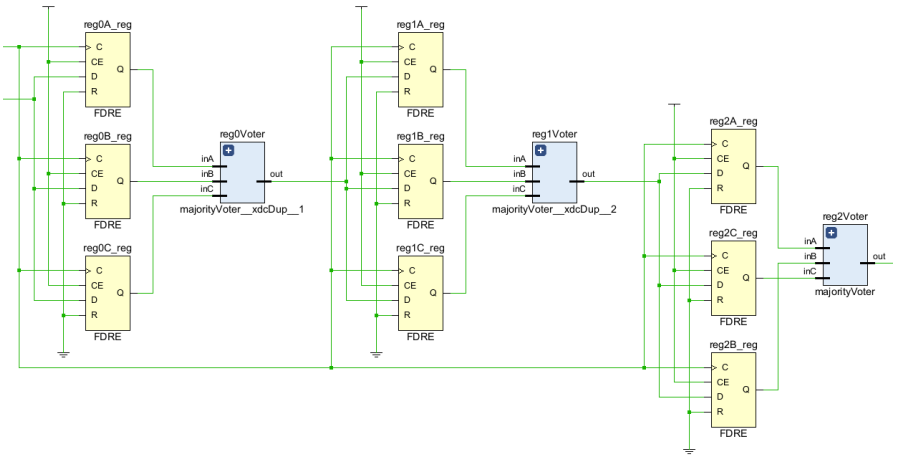


Abbildung 3.7: Wirkweise der Constraints an einem triplizierten 3 Bit Schieberegister

Wie erwartet ist die Triplizierung der Flipflops erhalten geblieben. In Abbildung 3.8 ist die gleiche Schaltung dargestellt, hier wurden jedoch die Constraints abgeschaltet. Die komplette Schaltung inklusive Voting ist reduziert worden.

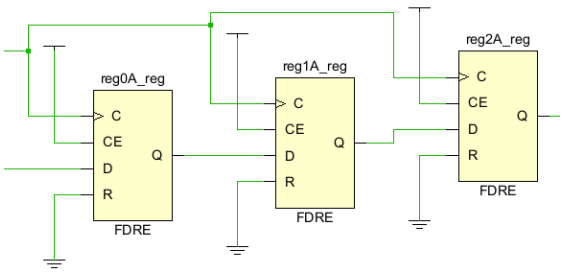


Abbildung 3.8: Selbiges 3 Bit Schieberegister ohne aktive Constraints

3.6 Triplizierung eines Moduls aus dem MOPS-HUB Entwurf

In diesem Kapitel soll die Vorgehensweise einer Triplizierung mit tmrg_cfg_gen und reg2 erläutert werden. Als Beispiel dient die sync_bytes Statemachine des MOPS-HUB Entwurfs. Nachdem die Verilog Datei in Vivado geladen wurde, muss zunächst das tcl Skript mit dem Befehl `source "../reg2.tcl"` aus der tcl Konsole heraus gestartet werden. Das Skript öffnet automatisch die Vivado Designviews und erstellt die Textdateien in dem Verzeichnis, in welchem das Skript ausgeführt wird. Für die Implementierung von TMR ist hierbei die Datei *not_triplicated_rtl.txt* von Interesse. Hierbei handelt es sich um die Liste aller Flipflop Primitive, welche der Elaborated Netzliste entnommen wurden. Der Inhalt ist wie folgt:

```
1 sync_bytes_SM, sel_cnt_cld_reg[2]
2 sync_bytes_SM, sel_cnt_cld_reg[1]
3 sync_bytes_SM, sel_cnt_cld_reg[0]
4 sync_bytes_SM, current_state_reg[3]
5 sync_bytes_SM, current_state_reg[2]
6 sync_bytes_SM, current_state_reg[1]
7 sync_bytes_SM, current_state_reg[0]
```

Quelltext 3.23: Triplizierungsbeispiel - Nicht triplizierte Flipflops

Anhand der Liste ist zu erkennen, dass im Modul `sync_bytes_SM` die Register `sel_cnt_reg` sowie `current_state_reg` nicht mit einer Voterzelle verbunden sind. Dies ist bei einem nicht triplizierten Verilog Code zu erwarten. Mit der Liste kann nun tmrg_cfg_gen gestartet werden. Das Python Skript erwartet hierbei die *not_triplicated_rtl.txt* im Arbeitsverzeichnis von *main.py*. Des Weiteren muss das Verilog Modul in dem Ordner HDL im Arbeitsverzeichnis platziert werden. Nun kann main.py ausgeführt werden. Das Skript generiert eine Konfigurationsdatei für das TMRG Tool sowie eine neue Verilog Datei in „hdl/output“ mit der Kennung *VTR*. Im folgenden ist der Inhalt der Konfigurationsdatei gezeigt:

```
1  [tmrg]
2  tmr_dir = ../hdl/output/tmr
3  rtl_dir = ../hdl/output
4  sdc_generate = False
5  files =
6
7  [global]
8  default = do_not_triplicate
9  tmr_error = false
10 top_module =
11
```

```
12 [sync_bytes_SM]
13 default = do_not_triplicate
```

Quelltext 3.24: Triplizierungsbeispiel - Konfigurationsdatei

Des Weiteren ein Ausschnitt der neu erstellten Verilog Datei:

```
1  reg [3:0] current_state; reg [3:0] next_state;
2
3  //----------------------------------------------------------------
4  // Next State Block for machine csm
5  //----------------------------------------------------------------
6
7
8  //***Voter***
9  //This block was generated by tmrg_cfg_gen
10 //tmrg triplicate sel_cnt_cld
11 wire [2:0] sel_cnt_cldV = sel_cnt_cld;
12 //tmrg triplicate current_state
13 wire [3:0] current_stateV = current_state;
14 //***Voter***
```

Quelltext 3.25: Triplizierungsbeispiel - Verilog Code

Die Direktiven wurden erfolgreich in den Verilog Code platziert. Des Weiteren wurden die Deklaration der Zustandsvektoren erfolgreich gesplittet. Nun kann die Konfigurationsdatei im Ordner „config“ dem TMRG Tool übergeben werden. Nachdem das TMRG Tool seine Arbeit verrichtet hat, kann der triplizierte Verilog Code wieder in Vivado geladen und mit den Constraints implementiert werden.

Nach einem weiteren Durchlauf von reg2 ergibt sich folgendes Resultat Blick in der Datei *triplicated_impl.txt*:

```
sync_bytes_SMTMR, sel_cnt_cldC_reg[2]
sync_bytes_SMTMR, sel_cnt_cldC_reg[1]
sync_bytes_SMTMR, sel_cnt_cldC_reg[0]
sync_bytes_SMTMR, sel_cnt_cldB_reg[2]
sync_bytes_SMTMR, sel_cnt_cldB_reg[1]
sync_bytes_SMTMR, sel_cnt_cldB_reg[0]
sync_bytes_SMTMR, sel_cnt_cldA_reg[2]
sync_bytes_SMTMR, sel_cnt_cldA_reg[1]
sync_bytes_SMTMR, sel_cnt_cldA_reg[0]
sync_bytes_SMTMR, current_stateC_reg[3]
sync_bytes_SMTMR, current_stateC_reg[2]
sync_bytes_SMTMR, current_stateC_reg[1]
sync_bytes_SMTMR, current_stateC_reg[0]
sync_bytes_SMTMR, current_stateB_reg[3]
sync_bytes_SMTMR, current_stateB_reg[2]
sync_bytes_SMTMR, current_stateB_reg[1]
sync_bytes_SMTMR, current_stateB_reg[0]
sync_bytes_SMTMR, current_stateA_reg[3]
sync_bytes_SMTMR, current_stateA_reg[2]
sync_bytes_SMTMR, current_stateA_reg[1]
sync_bytes_SMTMR, current_stateA_reg[0]
```

Quelltext 3.26: Triplizierungsbeispiel - Triplizierte Flipflops

Die Liste zeigt, dass alle Flipflops erfolgreich mit einem Voter verbunden worden sind. Beim derzeitigen Stand der Arbeit ist tmrg_cfg_gen noch nicht in der Lage, das Mehrheitsvotum als Feedback in die Logik zu integrieren. Dies muss derzeit manuell erfolgen. Die automatische Platzierung der Direktive nimmt jedoch bereits einen Großteil der Arbeit ab. Das Modul *sync_bytes_SM* sowie die triplizierte Version werden der Arbeit beigefügt. Aufgrund der Länge des Verilog-Codes ist dieser nicht im Anhang enthalten. In Quelltext 7 sind Beispiele aus sync_bytes_SM des Feedbacks dargestellt. Immer dann wenn ein tripliziertes Element verglichen, einem neuen Signal zugewiesen oder inkrementiert wird, ist die Verwendung des gevoteten Signals erforderlich.

```
if ((efifoDout ==Kchar_eop)&&(sel_cnt_cldV == max_byte))
    next_state = ST_EOF;
------------------------------------------------------------
sel_cnt_cld = sel_cnt_cldV +1;
------------------------------------------------------------
sel_cnt = sel_cnt_cldV;
```

Quelltext 3.27: Mehrheitsvotum Feedback

3.7 Fehler in der CANakari TMR Implementierung

Während der Arbeit wurden Fehler in der Triplizierung des MoPS CANakari entdeckt, da die Triplikation des CANakari eigens für den MOPS-HUB geändert werden musste. Ein Großteil der Fehler wurde bereits durch das Anwenden des reg2 Skript auf das triplizierte CANakari Design aufgedeckt. Im folgenden ist der Inhalt der Liste *not_triplicated_rtl* gezeigt.

```
1  prescale2TMR, prescaler/Prescale_ENC_reg
2  [...]
3  stuffing2TMR, MediumAccessControl/stuff/countC_reg[2]
4  [...]
5  stuffing2TMR, MediumAccessControl/stuff/BufC_reg
6  [...]
7  destuffing2TMR, MediumAccessControl/destuff/countC_reg[2]
8  [...]
9  destuffing2TMR, MediumAccessControl/destuff/buffC_reg
10 [...]
```

Quelltext 3.28: Nicht triplizierte Elemente des vollständig triplizierten CANakari

Insgesamt sind folgende Fehler aufgefallen:

- **Modul CLK_Counter:**
  Counter_C_iVoted: Ungevotetes Signal in if Abfrage
- **Modul destuffing2:**
  count, Buf nicht gevotet
- **Modul erbcount2:**
  counterVoted: Ungevotetes Signal in if Abfrage
- **Modul Phasenfehler_Reg:**
  Ausgang e_k sollte e_k_i_Voted zugewiesen werden anstatt e_k_i
- **Modul prescale2:**
  Prescale_EN nicht gevotet
- **Modul stuffing2:**
  count, Buf nicht gevotet

Da CANakari wie MOPS-HUB ebenfalls mit Hilfe von tmrg_cfg_gen tripliziert wurde, treten die im Quelltext 11 aufgelisteten Fehler im teilweise triplizierten CANakari nicht auf, da sich tmrg_cfg_gen strikt an die vorgegebene Flipflop-Liste hält (Siehe auch Kapitel 3.6). Alle genannten Fehler wurden in der MOPS-HUB CANakari Version behoben.

4 Ausblick

In dieser Arbeit wurde erfolgreich ein Testentwurf und die dazugehörige Benutzeroberfläche entwickelt, um ein 7-Series FPGA teilweise heraus der Benutzerlogik aus zu rekonfigurieren. Die von Xilinx zur Verfügung gestellten Informationen über die Schnittstelle sind an einigen Stellen nicht sehr präzise. Hier stellt sich die Frage, inwieweit welche Erweiterungen des Controllers sinnvoll wären, da dies stark vom Einsatzzweck abhängt. Verbesserungen an der Benutzeroberfläche könnten zwar vorgenommen werden, andererseits stellt die ICAP GUI keine Alternative zu bereits bestehenden Konfigurationsmöglichkeiten wie z.B. der JTAG-Programmierung über Vivado dar. Letztendlich handelt es sich um ein Testdesign im Rahmen einer Machbarkeitsstudie. Erwähnenswert ist, dass das gesamte MOPS-HUB Design erfolgreich mit einer partiellen Rekonfiguration in Hardware getestet wurde. Die Stärken der partiellen Rekonfiguration und der ICAP-Schnittstelle zeigen sich vor allem bei der Behebung von strahlungsinduzierten Logikfehlern im Konfigurationsspeicher durch zyklisches Neuschreiben. Sehr interessant könnte die Kombination von ICAP mit der ECC/CRC-Funktion von Xilinx FPGAs sein. Die ECC/CRC-Funktion ist in der Lage, die genaue Adresse eines fehlerhaften Frames im Konfigurationsspeicher anzugeben. Anhand dieser Adresse könnte der Inhalt mittels ICAP neu geschrieben werden. Hier stellt sich die Frage, ob und wie eine Korrektur eines einzelnen Frames möglich ist und wie sich ein Frame innerhalb des statischen Teils der Logik verhält. Ein FPGA Design könnte auch in mehrere kleine rekonfigurierbare Partitionen aufgeteilt werden, so dass kleinere Teile eines Designs partiell rekonfiguriert werden können, um einen Ausfall durch Rekonfiguration weiter zu minimieren. ICAP bietet auch eine elegante Lösung, um Funktionsblöcke während der Laufzeit auszutauschen, um so den Gesamtressourcenverbrauch zu reduzieren. Außerdem bietet ICAP eine gute Möglichkeit, einen Entwurf aus der Ferne zu aktualisieren, ohne jegliche externe Beschaltung des FPGAs.

Im Rahmen der Triplizierung des MOPS-HUB Projektes wurden Skripte entwickelt, die den manuellen Aufwand reduzieren. Es wurde bereits an einer neuen Version von tmrg_cfg_gen gearbeitet, die statt regulärer Ausdrücke PyParsing zur aktiven Syntaxanalyse verwendet.

Die Idee dahinter ist eine vollständige Automatisierung einer Triplizierung anhand einer Liste von Flipflop Primitiven, wie sie von reg2 erzeugt wird. Hierzu wurde ein bereits existierender Verilog Parser von dem PyParsing GitHub Repository[7] um weitere Funktionalität erweitert. Die Triplizierung des MOPS-HUB Projekts ist während dieser Arbeit noch nicht abgeschlossen. Auch steht noch eine Simulation von SEU durch Beinflussung der Votingnetze während der Simulation für die Verifizierung aus. Des Weiteren muss sich durch Tests zeigen, ob die gewählte TMR Strategie ausreichenden Schutz am Standort PP 3 bietet und inwiefern SEUs im Konfigurationsspeicher der FPGA eine Rolle spielen.

[7] `https://github.com/pyparsing/pyparsing/blob/master/examples/verilogParse.py`

Abbildungsverzeichnis

Tabellenverzeichnis

Quelltextverzeichnis

Literaturverzeichnis

[1] *7 Series FPGAs Configurable Logic Block User Guide*. 2016. URL: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB.

[2] *7 Series FPGAs Configuration User Guide*. 2023. URL: https://docs.xilinx.com/r/en-US/ug470_7Series_Config.

[3] Lucas Schreiter Ahmed Qamesh und Theodor Fischer. *ATLAS sub-detector Phase-II Upgrade: MOPS-HUB FPGA Firmware specification*. Techn. Ber. 2022, S. 44.

[4] *Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics*. 2022. URL: https://docs.xilinx.com/v/u/en-US/ds181_Artix_7_Data_Sheet.

[5] Collaboration ATLAS. *Letter of Intent for the Phase-II Upgrade of the ATLAS Experiment*. Techn. Ber. Draft version for comments. Geneva: CERN, 2012. URL: https://cds.cern.ch/record/1502664.

[6] *AXI HWICAP v3.0 Product Guide*. 2016. URL: https://docs.xilinx.com/v/u/en-US/pg134-axi-hwicap.

[7] *Configuration simulation testbenches*. 2021. URL: https://support.xilinx.com/s/article/53632?language=en_US.

[8] *How does the bitstream compress option work*. 2021. URL: https://support.xilinx.com/s/article/16996?language=en_US.

[9] Xueye Hu. „Design and Test of a Signal Packet Router Prototype for the ATLAS NSW sTGC Detector“. In: (2016). URL: https://cds.cern.ch/record/2119894.

[10] Xueye Hu u. a. „A multi-layer SEU mitigation strategy to improve FPGA design robustness for the ATLAS muon spectrometer upgrade“. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 939 (2019), S. 30–35. ISSN: 0168-9002. DOI: https://doi.org/10.1016/j.nima.2019.05.045. URL: https://www.sciencedirect.com/science/article/pii/S0168900219306874.

[11] *IEEE Standard Verilog® Hardware Description Language*. 2001. URL: https://www.eg.bucknell.edu/~csci320/2016-fall/wp-content/uploads/2015/08/verilog-std-1364-2005.pdf.

[12] S. Kulis. „Single Event Effects mitigation with TMRG tool“. In: *Journal of Instrumentation* 12.01 (2017), S. C01082. URL: http://stacks.iop.org/1748-0221/12/i=01/a=C01082.

[13] *LogiCORE IP AXI HWICAP (v2.03.a) Data Sheet*. 2012. URL: https://docs.xilinx.com/v/u/en-US/ds817_axi_hwicap.

[14] *NSEU Mitigation in Avionics Applications Application Note*. 2010. URL: https://docs.xilinx.com/v/u/en-US/xapp1073_NSEU_Mitigation_Avionics.

[15] Joleen Pater. *The ATLAS Pixel Detector Upgrade at the HL-LHC*. Techn. Ber. Geneva: CERN, 2020. DOI: 10.22323/1.373.0011. URL: https://cds.cern.ch/record/2709133.

[16] *Technical Design Report for the ATLAS Inner Tracker Pixel Detector*. Techn. Ber. Geneva: CERN, 2017. DOI: 10.17181/CERN.FOZZ.ZP3Q. URL: https://cds.cern.ch/record/2285585.

[17] „The ATLAS Experiment at the CERN Large Hadron Collider“. In: *Journal of Instrumentation* 3.08 (Aug. 2008), S08003. DOI: 10.1088/1748-0221/3/08/S08003. URL: https://dx.doi.org/10.1088/1748-0221/3/08/S08003.

[18] Nathan Triplett. *ATLAS Pixel Detector System Test*. Techn. Ber. Iowa State University, 2017. URL: https://cds.cern.ch/record/2285585.

[19] *UltraFast Design Methodology Guide for FPGAs and SoCs*. 2022. URL: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology.

[20] *UltraScale Architecture Configuration*. 2022. URL: https://www.xilinx.com/content/dam/xilinx/support/documents/user_guides/ug570-ultrascale-configuration.pdf.

[21] *UltraScale Architecture Libraries Guide*. 2022. URL: https://docs.xilinx.com/r/en-US/ug974-vivado-ultrascale-libraries.

[22] *Vivado Design Suite 7 Series FPGA and Zynq-7000 SoC Libraries Guide*. 2022. URL: https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries.

[23] *Vivado Design Suite Properties Reference Guide*. 2022. URL: https://docs.xilinx.com/r/en-US/ug912-vivado-properties.

[24] *Vivado Design Suite Tcl Command Reference Guide*. 2022. URL: https://docs.xilinx.com/r/en-US/ug835-vivado-tcl-commands.

[25] *Vivado Design Suite User Guide: Dynamic Function eXchange*. 2022. URL: https://docs.xilinx.com/r/en-US/ug909-vivado-partial-reconfiguration.

[26] *Vivado Design Suite User Guide: Logic Simulation*. 2022. URL: https://docs.xilinx.com/r/en-US/ug900-vivado-logic-simulation/SelectMAP-Simulation.

[27] *Vivado Design Suite User Guide: Synthesis*. 2022. URL: https://docs.xilinx.com/r/en-US/ug901-vivado-synthesis.

[28] Alexander Walsemann u. a. „A CANopen based prototype chip for the Detector Control System of the ATLAS ITk Pixel Detector“. In: *PoS* TWEPP2019 (2020), S. 013. DOI: 10.22323/1.370.0013. URL: https://cds.cern.ch/record/2724951.

A Anhang

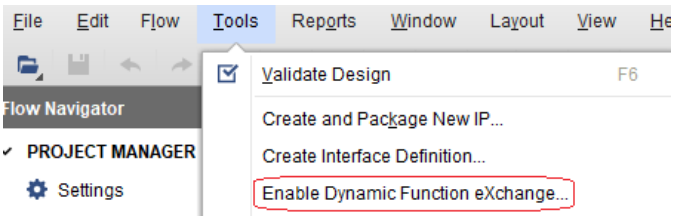
CD und USB Stick mit folgendem Inhalt:

- Thesis im PDF Format
- Bitstreams
- constraints
- icap_gui
- Quellen
- reg2
- sync_bytes_sm
- tmrg_cfg_gen
- Vivado Projekte:
  - icap_controller
  - constraints_showcase

Aufgrund der begrenzten Speicherkapazität der CD sind alle Dateien mit Ausnahme der Bachelor-Thesis im 7zip-Format komprimiert.

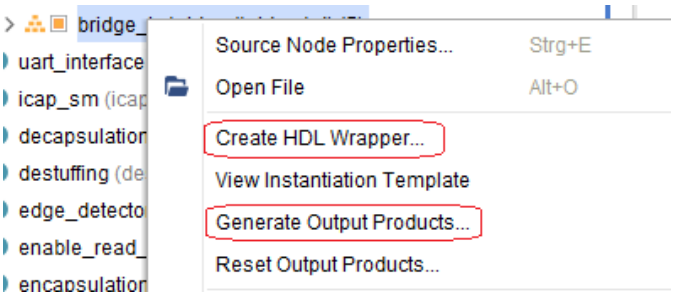
A.1 Anleitung für die Erstellung eines DFX Projektes

Diese Anleitung beschreibt die Erstellung eines DFX Projektes. Zunächst sollte das Projekt gesichert werden, da die Aktivierung von DFX nicht mehr rückgängig gemacht werden kann. Für DFX wird mindestens Vivado 2013.3 oder neuer benötigt. Unter dem Menüreiter *Tools*, *Enable Dynamic Function eXchange* kann das Projekt umgewandelt werden.

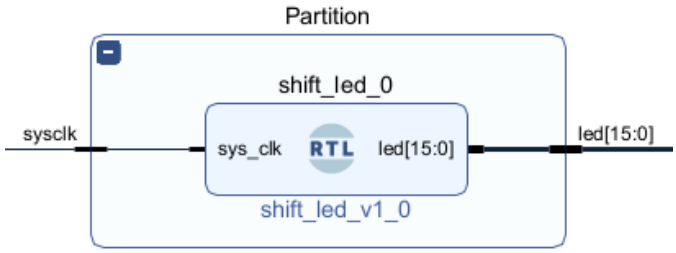


Hier erzeugt Vivado nochmals eine Warnung, dass dieser Schritt unwiderrufbar ist.

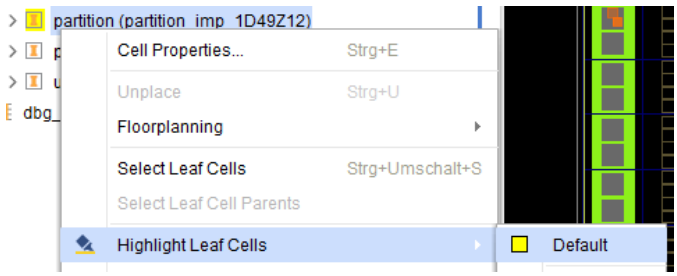
Für die Erstellung von Partitionen in einem DFX Projekt bestehen zwei Möglichkeiten. Es wird zunächst der Weg über ein Block Design gezeigt. Falls nicht bereits schon geschehen, muss ein HDL Wrapper sowie dessen Output Product generiert werden.



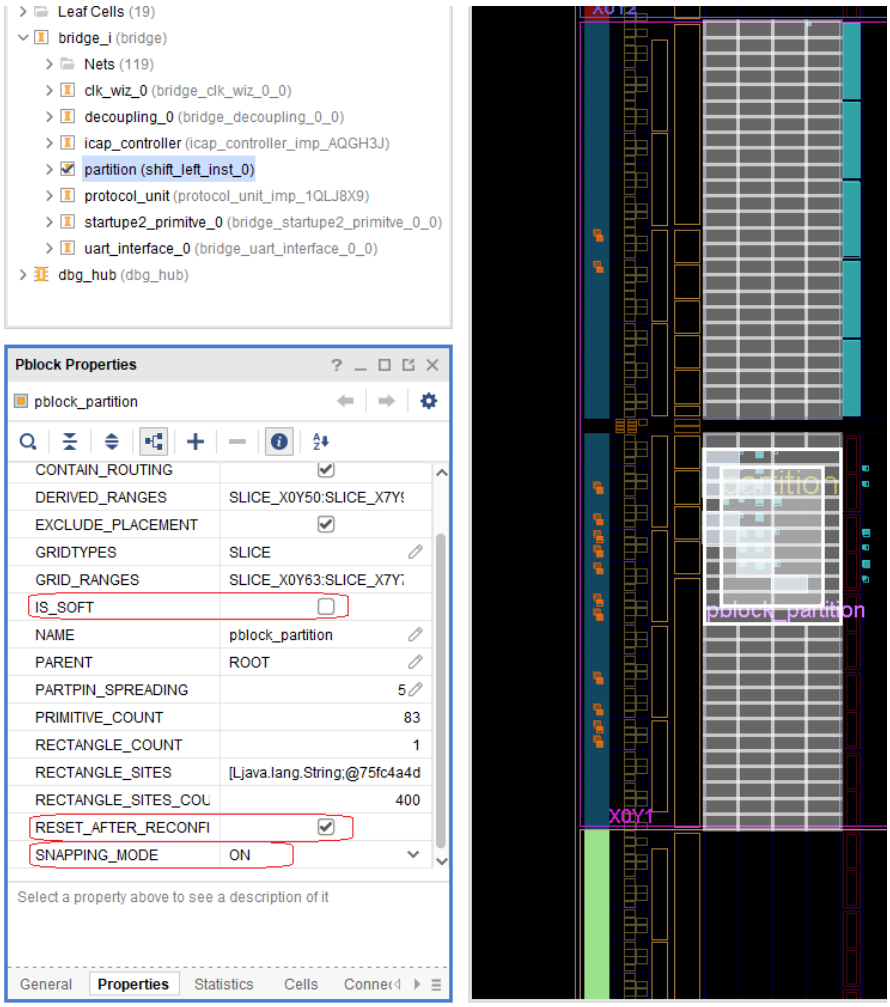
Nachdem der Prozess abgeschlossen ist, muss als nächstes eine neue Hierarchie in dem Block Design erstellt werden. Eine Hierarchie kann im Block Design mit Rechtsklick über das Kontextmenü *Create Hierarchy* erzeugt werden. Der Hierarchie sollte ein sinnvoller Name zugewiesen werden, da es sich hier um die Partition handeln wird. In dieser Hierarchie muss nun die Schaltung platziert werden, welche rekonfiguriert werden soll. Hier ist es wichtig die Ein- und Ausgänge zu verbinden. Das Ergebnis könnte in etwa so aussehen:



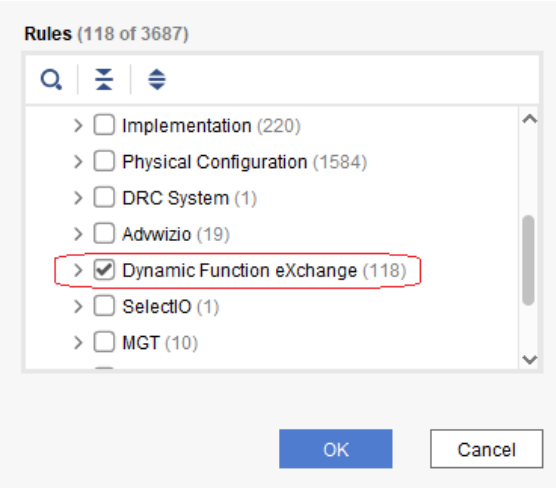
Ab diesem Punkt sollte das Projekt implementiert werden, um die Grenzen einer Partition zu planen. Dazu wird das Implementierte Design göffnet und die betroffene Zelle farblich markiert. Hierzu kann man sich an den Namen orientieren, welche der Hierarchie vergeben wurde.



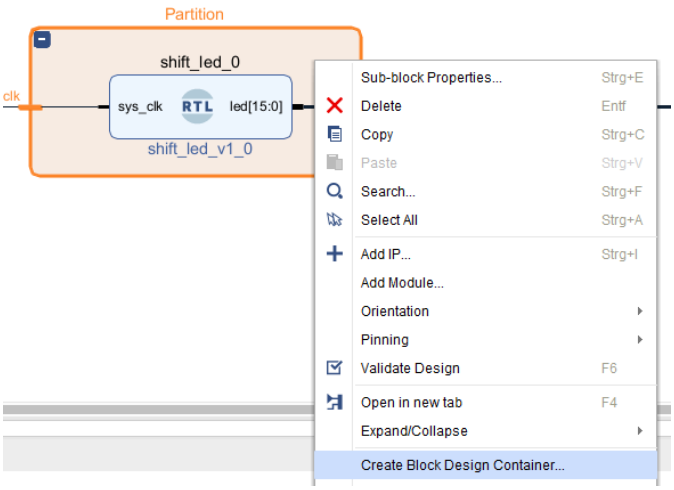
Die implementierten Primitiven (Leaf Cells) sollten nun farblich markiert sein. Der Sinn dahinter ist die Erstellung einer Partitionsgrenze in der Nähe der durch Vivado platzierten Primitven, um etwaige Timing oder Routing Probleme zu verhindern. Mit Rechtsklick auf die Zelle wird *Floorplanning* und *Draw Pblock* ausgewählt, um eine Partitionsgrenze zu ziehen. Diese sollte nicht zu groß und nicht zu klein sein. Beim loslassen wird eine kurze Übersicht erstellt, welche Elemente sich innerhalb der Partitionsgrenzen befinden. In den *Pblock Properties* unter dem Reiter `Statistics` findet sich eine detaillierte Auflistung der verfügbaren Ressourcen innerhalb der Partition.



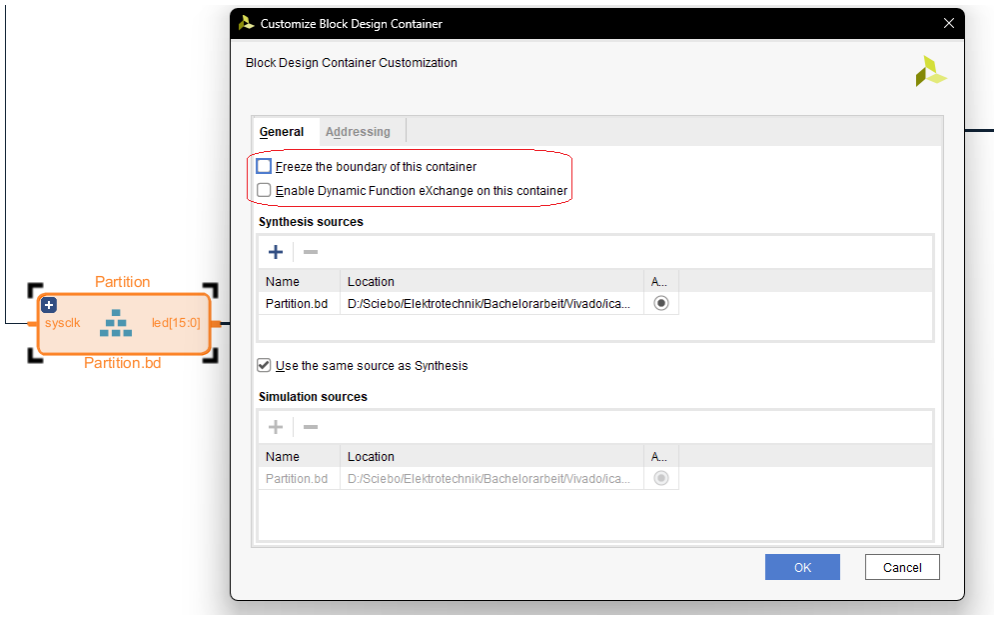
Im nächsten Schritt muss unbedingt das Häkchen bei *IS_SOFT* im Reiter `Properties` entfernt werden. *IS_SOFT* kann dazu führen, dass die Partitionsgrenze nicht streng eingehalten werden. Dies funktioniert jedoch nicht im Falle einer partiellen Rekonfiguration. Zusätzlich kann es sehr nützlich sein, den Haken bei *RESET_AFTER_RECONFIG* zu setzen. Somit wird nach einer Rekonfiguration die Logik auf seine Anfangswerte initialisiert. Hierfür muss jedoch die Partition die obere und untere Grenze einer Clock Region berühren. Ein weiteres nützliches Attribut ist *SNAPPING_MODE*. *SNAPPING_MODE* erlaubt Vivado die automatische Anpassung einer Region im Bezug zum Ressourcenbedarf. Zum Abschluss empfiehlt es sich einen DRC Report erstellen zu lassen. Hierzu wird oben in der Menüleiste *Reports* und dann *Report DRC* gewählt. Es öffnet sich ein Fenster, in dem folgende Regelen aktiviert werden sollten:



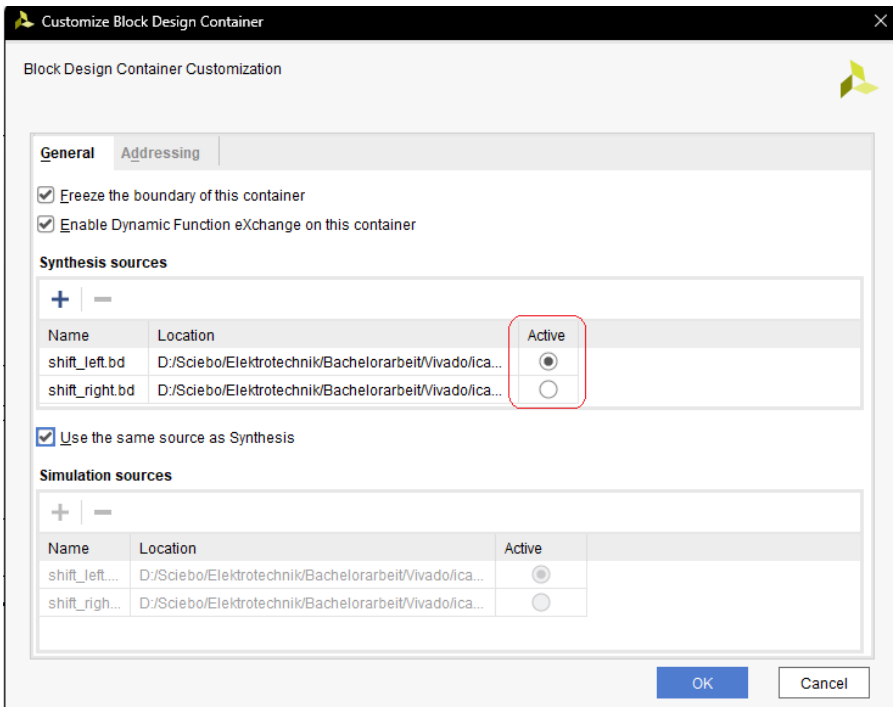
Wenn alles richtig konfiguriert wurde, werden keine Violations gefunden. Im Abschluss werden die Constraints mit STRG + S gespeichert. Nun kann die Hierarchie in ein Block Design Container umgewandelt werden. Vivado wird zunächst eine Meldung ausgeben, dass das Block Design nicht validiert ist. Hierzu muss einmal F6 gedrückt und ggf. den Anweisungen gefolgt werden.



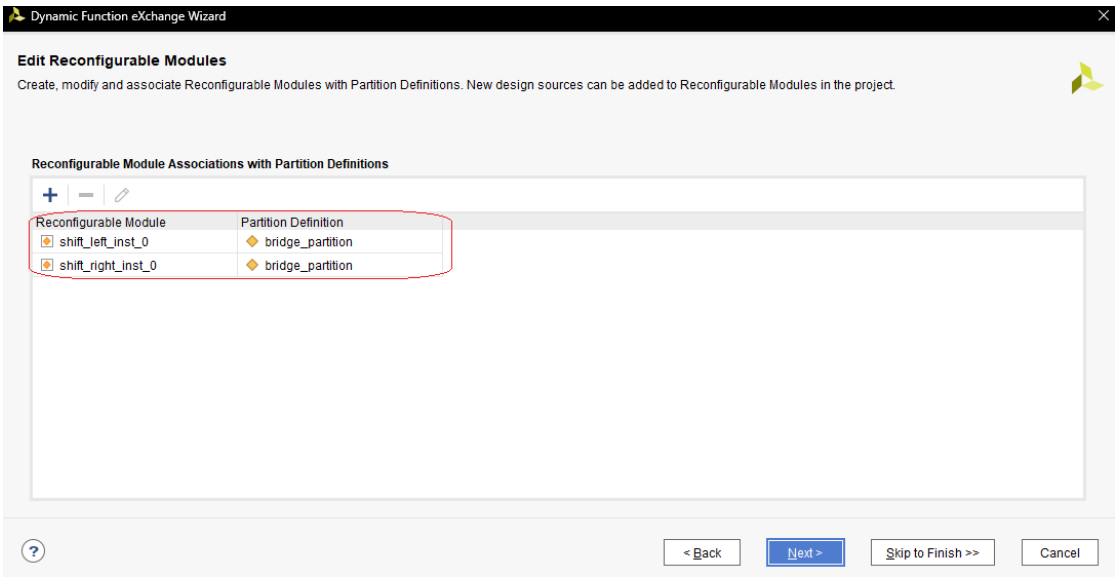
Auch hier sollte ein gut gewählter Name gewählt werden, da es sich hierbei um die Benennung des ersten rekonfigurierbaren Moduls handelt. Bei Bestätigung wird ein neues Blockschaltbild generiert, welches die platzierte Schaltung beinhaltet. Ein Doppelklick auf den Container öffnet die Optionen. Es müssen die beiden markierte Häkchen gesetzt werden. Nun ist aus dem Container ein DFX Container geworden, auch das Logo ändert sich zu DFX.



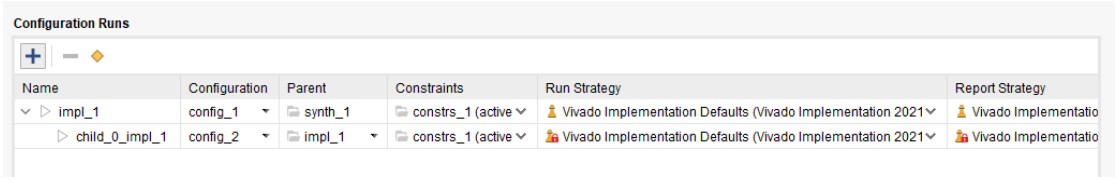
Sollen weitere Rekonfigurierbare Module erstellt werden, können diese per Rechtsklick über Partition und *Create Reconfigurable Module* hinzugefügt werden. Es öffnet sich ein leeres Block Design mit den Ein und Ausgangports. Hier kann nach belieben eine weitere Schaltung platziert werden. Anschließend muss die Konfiguration validiert und gespeichert werden. Mit einem Doppelklick auf die Partition kann das aktive rekonfigurierbare Modul verändert werden.



Zuletzt müssen noch Runs für die einzelnen rekonfigurierbaren Module erzeugt werden. Hierzu wird in der Menüleiste *Tools* und dann *Dynamic Function eXchange Wizard* ausgewählt. Wenn alles richtig gemacht wurde, sollten hier die rekonfigurierbaren Module und deren Partition stehen.



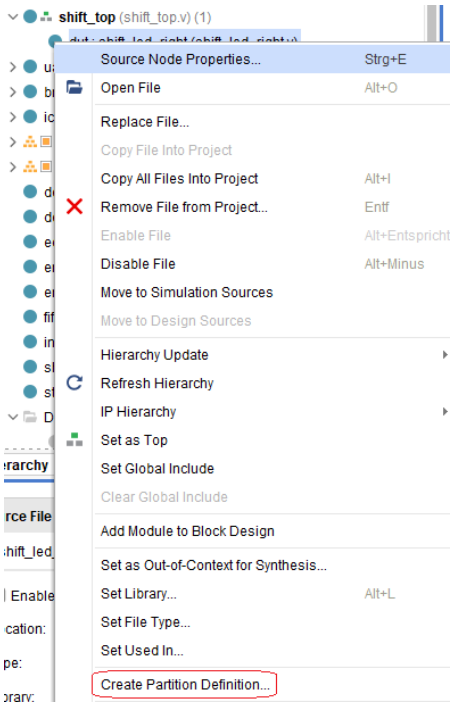
Im nächsten Schritt wird auf *automatically create configurations* geklickt. Je nach Anzahl der Partitionen/Rekonfigurierbare Module sollten hier die einzelnen Configs erzeugt worden sein. Im nächsten Schritt wird der Vorgang wiederholt. Das Ergebnis sollte ungefähr so aussehen:



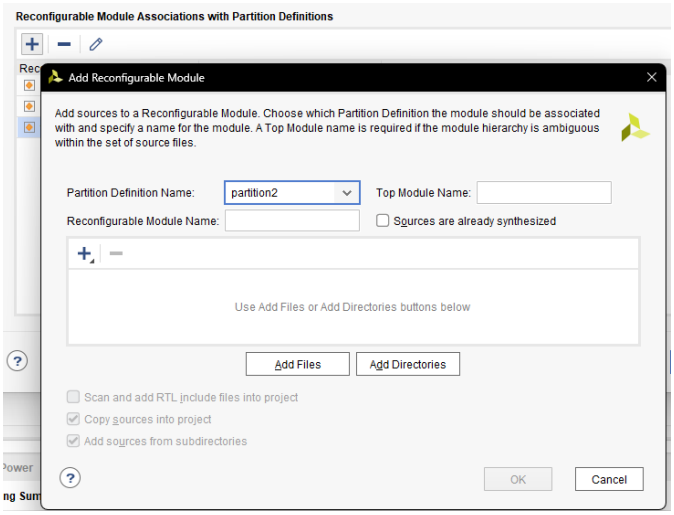
Anschließend wird auf Next und Finish geklickt.

Für den Fall, dass kein Block Design benutzt wird, gibt es eine zweite Möglichkeit eine Partition in einer Design Hierarchie zu erstellen. Die Grenzen einer Partition sollten bereits geplant sein, die Schritte sind identisch wie zuvor beschrieben. Hierzu wird die Partition der gewünschten Hierarchieebene geplant. Das Modul welches rekonfiguriert werden soll, muss sich unterhalb eines aktiven Top Moduls befinden.

Nachdem eine Partitionsgrenze erstellt wurde, wird mit der rechten Maustaste auf das gewünschte Modul geklickt und der Eintrag *Create Partition Definition* gewählt:

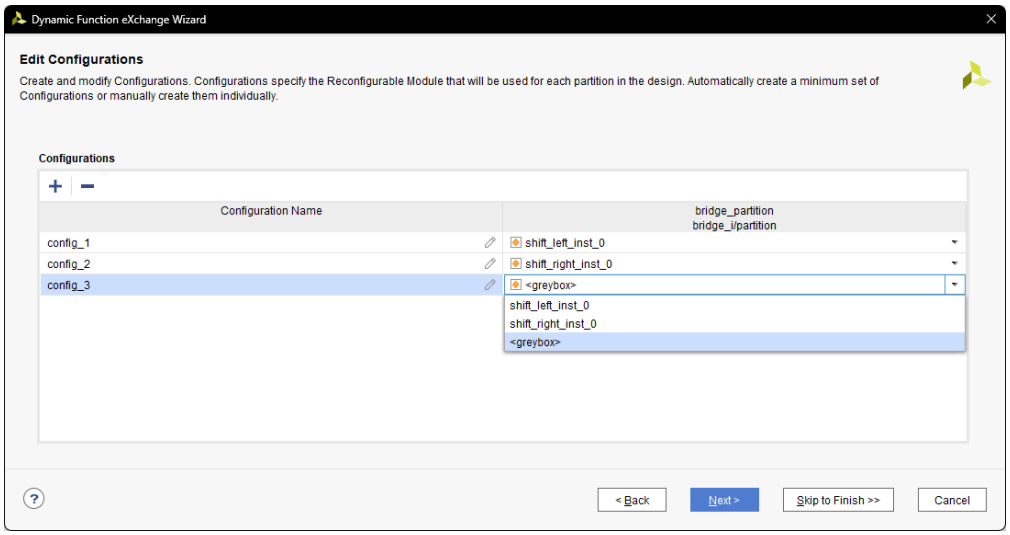


Es öffnet sich ein neues Fenster. Hier wird wieder ein geeigneter Name für die Partition gewählt. Wenn alles richtig gemacht wurde, taucht neben den Modulbezeichnern eine gelbe Raute auf. Wenn *Dynamic Function eXchange Wizard* wieder geöffnet wird, sollte dort das Modul zu erkennen sein. Falls weitere rekonfigurierbare Module zu der Partition hinzugefügt werden sollen, muss wieder der *Dynamic Function eXchange Wizard* geöffnet werden.



Hierzu wird auf das Plus Symbol geklickt, die richtige Partition ausgewählt und Module über `Add Files` oder `Add Directories` hinzugefügt. Die restlichen Schritte zur Erstellung der Configs und Runs bleiben dieselben.

Ist eine Greybox Definition einer Partition erwünscht, d.h. eine leere Partition, so kann im *Dynamic Function eXchange Wizard* an der Stelle *Edit Configurations* über das Plus Symbol eine weitere Configuration hinzugefügt werden. Anschließend auf die Liste klicken und den Eintrag `greybox` auswählen.



Sobald Änderungen an den Partitionen vorgenommen wurden, ist es wichtig im *Dynamic Function eXchange Wizard* an der Stelle *Edit Configurations Runs* die veralteten Configuration Runs über das Minus Symbol zu löschen und über den Button *automatically create configurations* neue Configuration Runs zu generieren.

A.2 Triplizierter Verilog Quellcode eines drei Bit Schieberegisters

```
1  module testTMR(
2    input  clk ,
3    input  d ,
4    output  q
5  );
6  wire [0:0] reg1VC;
7  wire [0:0] reg1VB;
8  wire [0:0] reg1VA;
9  wire [0:0] reg0VC;
10 wire [0:0] reg0VB;
11 wire [0:0] reg0VA;
12 wire dC;
13 wire dB;
14 wire dA;
15 wire clkC;
16 wire clkB;
17 wire clkA;
18 wor reg2TmrError;
19 wire reg2;
20 wor reg1TmrError;
21 wire reg1;
22 wor reg0TmrError;
23 wire reg0;
24 reg  reg0A ;
25 reg  reg0B ;
26 reg  reg0C ;
27 reg  reg1A ;
28 reg  reg1B ;
29 reg  reg1C ;
30 reg  reg2A ;
31 reg  reg2B ;
32 reg  reg2C ;
33 wire [0:0] reg2V =  reg2;
34 wire [0:0] reg1V =  reg1;
35 wire [0:0] reg0V =  reg0;
36
37 always @( posedge clkA )
38   begin
39     reg0A <= dA;
40     reg1A <= reg0VA;
41     reg2A <= reg1VA;
42   end
```

```
43
44 always @( posedge clkB )
45   begin
46     reg0B <= dB;
47     reg1B <= reg0VB;
48     reg2B <= reg1VB;
49   end
50
51 always @( posedge clkC )
52   begin
53     reg0C <= dC;
54     reg1C <= reg0VC;
55     reg2C <= reg1VC;
56   end
57 assign q =  reg2V;
58
59 majorityVoter reg0Voter (
60     .inA(reg0A),
61     .inB(reg0B),
62     .inC(reg0C),
63     .out(reg0),
64     .tmrErr(reg0TmrError)
65     );
66
67 majorityVoter reg1Voter (
68     .inA(reg1A),
69     .inB(reg1B),
70     .inC(reg1C),
71     .out(reg1),
72     .tmrErr(reg1TmrError)
73     );
74
75 majorityVoter reg2Voter (
76     .inA(reg2A),
77     .inB(reg2B),
78     .inC(reg2C),
79     .out(reg2),
80     .tmrErr(reg2TmrError)
81     );
82
83 fanout clkFanout (
84     .in(clk),
85     .outA(clkA),
86     .outB(clkB),
87     .outC(clkC)
88     );
```

```
89
90  fanout dFanout (
91      .in(d),
92      .outA(dA),
93      .outB(dB),
94      .outC(dC)
95      );
96
97  fanout reg0VFanout (
98      .in(reg0V),
99      .outA(reg0VA),
100     .outB(reg0VB),
101     .outC(reg0VC)
102     );
103
104 fanout reg1VFanout (
105     .in(reg1V),
106     .outA(reg1VA),
107     .outB(reg1VB),
108     .outC(reg1VC)
109     );
110 endmodule
111
112 module majorityVoter #(
113   parameter WIDTH = 1
114 )(
115   input wire  [WIDTH-1:0] inA,
116   input wire  [WIDTH-1:0] inB,
117   input wire  [WIDTH-1:0] inC,
118   output wire [WIDTH-1:0] out,
119   output reg              tmrErr
120 );
121   assign out = (inA&inB) | (inA&inC) | (inB&inC);
122   'ifdef tmrErr
123   always @(inA or inB or inC) begin
124     if (inA!=inB || inA!=inC || inB!=inC)
125       tmrErr = 1;
126     else
127       tmrErr = 0;
128   end
129   'endif
130 endmodule
131
132 module fanout #(
133   parameter WIDTH = 1
134 )(
```

```
135   input wire  [WIDTH-1:0] in,
136   output wire [WIDTH-1:0] outA,
137   output wire [WIDTH-1:0] outB,
138   output wire [WIDTH-1:0] outC
139 );
140   assign outA = in;
141   assign outB = in;
142   assign outC = in;
143 endmodule
```

Quelltext A.1: Verilog ANSI-C und List Syntax Deklaration