Fachhochschule Dortmund
Fachbereich Elektrotechnik
Studiengang Bachelor Elektrotechnik

Fachhochschule Dortmund
University of Applied Sciences and Arts

Bachelorarbeit
zur Erlangung des akademischen Grades
Bachelor of Engineering

Härtung eines KI Hardware Beschleunigers gegen strahleninduzierte Bitfehler

Hardening of an AI hardware accelerator against radiation-induced bit errors

Yunus Tas

Erstprüfer: Prof. Dr.-Ing. Michael Karagounis
Zweitprüfer: M. Sc. Felix Schneider

23. August 2023

Erklärung

Hiermit versichere ich, dass die von mir vorgelegte Prüfungsleistung selbständig und ohne unzulässige Hilfe erstellt worden ist. Alle verwendeten Quellen sind in meiner Arbeit als wörtliche und sinngemäße Übernahmen aus anderen Werken kenntlich gemacht, damit Art und Umfang der Verwendung nachvollziehbar sind.

Dortmund, 23. August 2023

_____

Unterschrift

Abstract

In dieser Arbeit wird die Strahlenhärtung eines KI Hardware Beschleunigers beschreiben, in dem das Design mit dem Triple Modular Redundancy Generator Toolset (TMRG) vollständig tripliziert wird. Anschließend wird das triplizierte Design mit einer statischen und einer dynamischen Verifikation auf die korrekte Art der Triplizierung und seiner Funktionsweise untersucht. Zuletzt werden Simulationen mit drei verschiedenen Injektionstypen durchgeführt, in dem die tatsächliche Funktion der Voter durch Injektion von Single Event Upsets geprüft wird.

English translation:

In this work, the radiation hardening of an AI hardware accelerator is described in which the design is fully triplicated using the Triple Modular Redundancy Generator Toolset (TMRG). Then, the triplicated design is examined with a static and a dynamic approach to verify the correct way of triplicating and its operation. Finally, simulations with three different injection types are performed, in which the actual function of the voter is tested by injecting single event upsets.

Danksagung

Ich möchte mich hiermit bei Herrn Prof. Dr.-Ing. Michael Karagounis und Herrn Felix Schneider für die Betreuung und Unterstützung bei diesem Projekt und bei der Verfassung dieser Arbeit herzlich bedanken.

Zusätzlich möchte ich meinen Eltern für die finanzielle und mentale Unterstützung danken, die mir ermöglicht hat, mich komplett auf dieses Projekt und das Verfassen dieser Arbeit zu konzentrieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingsverzeichnis

Tabellenverzeichnis

Kürzungsverzeichnis

ANN	Artificial Neural Network
CAECO	Configurable Accelerator Engine for Convolution Operations
CNN, ConvNet	Convolutional Neural Network
DNN	Deep Neural Network
FCN	Fully Convolutional Network
GHDL	G Hardware Design Language
HDL	Hardware Description Language
LEC	Logic Equivalence Checking
PLAG	Placement Generator
POMAA	Pareto-Optimal MAchine learning ASIC
POOMA	Pareto Optimal Machine Learning Accelerator
ReLU, relu	Rectified Linear Unit
SDC	Synopsys Design Constraints
SEE	Single Event Effects
SEEG	Single Event Effect Generator
SET	Single Event Transient
SEU	Single Event Upset
TCL	Tool Command Language
TBG	Testbench Generator
TMRG	Triple Modular Redundancy Generator
VHDL	Very High Speed Intergrated Circuit Hardware Description Language
XDC	Xilinx Design Constraints
Yosys	Yosys Open Synthesis Suite

1 Einleitung

Künstliche Neuronale Netze sind aktuell bei der Erkennung, Analyse und Verarbeitung komplexer Prozesse und Systemen essenziell. Durch Maschinelles Lernen ist es möglich große Mengen an Informationen auszuwerten und durch bestimmte Trainingsmethoden wiederkehrende Muster zu erkennen. Dazu gehören beispielsweise bestimmte Wetterbedingungen, Materialfehler in der Produktion oder die Objekterkennung in Bild- und Videodateien [15].

Eine Art von künstlichen neuronalen Netzen sind faltungsneuronale Netze, die durch Faltung bestimmte Merkmale aus einer Eingangsgröße extrahieren und in einer Ausgangsgröße speichern. Diese Netze sind sehr bekannt und werden häufig zur Erkennung und Verarbeitung von Bild-, Video- und Audiodateien eingesetzt. In diesem Zusammenhang beschäftigt sich diese Arbeit mit einem KI Hardware Beschleuniger namens Configurable Accelerator Engine for Convolution Operations (CAECO), der eindimensionale Faltungsneuronale Netze, auch bekannt als Fully Convolutional Network, durch seine Hardware-Architektur ausführt. Dieser Beschleuniger wurde im Rahmen des Pareto Optimal Machine Learning Accelerator (POOMA) mit dem Ziel entwickelt, ihn als Neural Network IP-Core zusammen mit einem RISC-V-Prozessorkern als hybriden ASIC zu verwenden.

Um dieses Ziel zu erreichen, muss der CAECO in einer strahlungsharten Version entwickelt werden, wobei in dieser Arbeit die vollständige Triplizierung des Beschleunigers beschrieben wird und die gewonnenen Erkenntnisse und Ergebnisse anhand von Methoden und Materialien erläutert werden.

In Kapitel 1 wird das Thema eingeleitet und erläutert, was die Motivation und das Ziel dieser Arbeit darstellt.

In Kapitel 2 werden die theoretischen Grundlagen sowie verwendete Materialien vorgestellt, die für das Verständnis der Inhalte wesentlich sind.

In Kapitel 3 wird der CAECO-Beschleuniger vorgestellt und seine Architektur und Modul-Hierarchie im Detail erläutert.

In Kapitel 4 wird das Konzept der Triple Modular Redundancy (dreifache modulare Redundanz) erläutert, indem einige Tools vorgestellt werden, die es ermöglichen, Designs zu triplizieren.

In Kapitel 5 wird die vollständige Triplizierung des Beschleunigers anhand von Quellcodes und Abbildungen dargestellt.

In Kapitel 6 werden zwei Verifikationsmethoden für das vollständig triplizierte Design des Beschleunigers in statischer und dynamischer Weise vorgestellt.

In Kapitel 7 werden verschiedene Injektionstypen von Single Event Effekten anhand von Quellcodes und Simulationsergebnissen erläutert.

Kapitel 8 entspricht einer Zusammenfassung von Erkenntnissen und Ergebnissen, welche mit einem Ausblick abgeschlossen wird.

2 Grundlagen

2.1 Convolutional Neural Network

Ein Convolutional Neural Network (CNN, ConvNet) ist ein künstliches faltungsneuronales Netz, das seine Funktion an biologischen Prozessen anlehnt, die im Gehirn stattfinden. CNNs können im Themenbereich des Machine Learnings beziehungsweise des Maschinellen Lernens zugeordnet werden und sind eine der am häufigsten verwendeten Artificial Neural Networks (ANNs) für die Verarbeitung von Bild- und Audiodaten [1].

2.1.1 Grundlagen

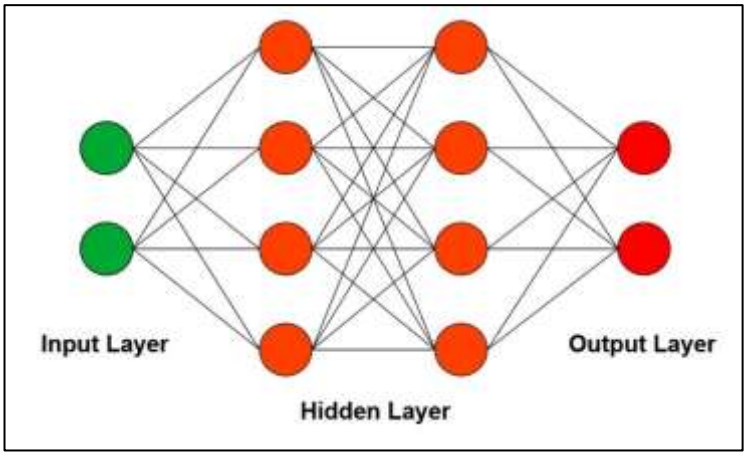

Abbildung 2.1: Struktur eines künstlichen neuronalen Netzes (ANN).

In Abbildung 2.1 ist ein beispielhaftes ANN dieser Art, das über einen Input Layer (Eingangsschicht), zwei Hidden Layer (verborgene Schicht) und einen Output Layer (Ausgangsschicht) verfügt. Diese Layer bestehen aus künstlichen Neuronen, deren Funktion durch Gleichung 2.1 mathematisch beschrieben werden kann [5].

$$o = \varphi\left(\sum_{i=0}^{n} x_i w_i + w_0\right) \tag{2.1}$$

Im Input Layer befinden sich Neuronen, die Informationen aufnehmen und an die Neuronen im Hidden Layer weitergeben. Mit der weitergeleiteten Information wird jedem Neuron x_i ein Gewicht w_i zugeordnet. Mit dem Bias-Wert w_0, auch Bias-Neuron genannt, können die Daten entsprechend der verwendeten Aktivierungsfunktion angepasst werden. Die Aktivierungsfunktion φ bestimmt den Zustand des Neurons durch einen definierten Schwellenwert. Wenn der Schwellwert erreicht ist, wird der Zustand des Neurons dem Output Layer o weitergegeben.

2.1.2 Aufbau

Ein typisches CNN besteht aus drei verschiedenen Arten von Schichten: Convolutional Layer (Faltungsschicht), Pooling Layer (Pooling-Schicht) und Fully-Connected Layer (völlig verbundene Schicht). Die Anzahl und Reihenfolge der Faltungs- und Pooling-Schichten

variiert je nach Anwendung. Netze mit mehreren Faltungsschichten werden als Deep Convolutional Neural Networks (DCNNs) bezeichnet.

In einer Faltungsschicht wird der Filter mit dem Input der Schicht gefaltet. Dadurch können je nach Wahl des Filters bestimmte Merkmale extrahiert werden. Bei Bildern können beispielsweise Farben und Formen erkannt werden. Die Filter selbst werden als Matrizen für zweidimensionale Eingaben oder als Vektoren für eindimensionale Eingaben (Höhe × Breite) gewählt und können eine dreidimensionale Eingabe mit einer bestimmten Tiefe (Höhe × Breite × Tiefe) verarbeiten.

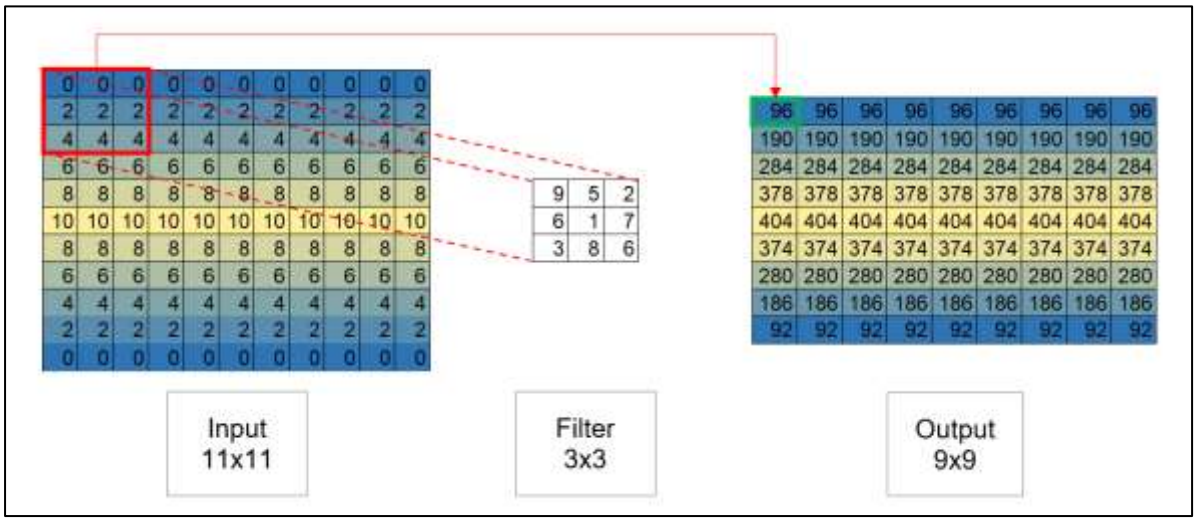

Abbildung 2.2: Faltung eines 11x11 Inputs mithilfe eines 3x3 Filters zu einem 9x9 Output

Abbildung 2.2 beschreibt die Faltung einer Inputgröße mit Hilfe eines vordefinierten Filters. Der rot markierte 3x3-Zahlenblock wird diskret gefaltet, indem das Skalarprodukt mit der Faltungsmatrix berechnet wird [1]. Der sogenannte Stride legt die Bewegung des Filters fest, wie viele Pixel der Filter nach rechts bzw. unten im Input verschoben wird. Diese Faltungsprozedur wird wiederholt, nachdem eine Schrittweite (Stride) nach rechts verschoben wurde. Sobald das Filter am rechten Rand angelangt ist, wird es mit einer Schrittlänge nach unten zum linken Rand verschoben, so dass die nächste Reihe von Eingaben berechnet werden kann.

Bei Anwendung dieser Methode auf die Eingangsdaten in Abbildung 2.2 wird der 11x11 Input in einen 9x9 Output umgewandelt. Die Größe des Outputs ändert sich mit der Größe des Filters. Bei tieferen Netzwerken mit mehreren hintereinander geschalteten Schichten wird die Größe jeder Schicht durch die Anwendung des Filters reduziert, was zu dem Problem führt, dass mit zunehmender Schichttiefe weniger Informationen zur Verfügung stehen [2].

Damit die Größe des Inputs im Output nach der Faltung erhalten bleibt, wird das „Zero Padding“-Verfahren angewendet, wie in Abbildung 2.3 zu sehen ist. Dabei werden bei den Input-Daten entlang der Kanten „Nullen“ hinzugefügt beziehungsweise „aufgefüllt“, damit der Filter diese bei der Rechnung mit einbezieht.

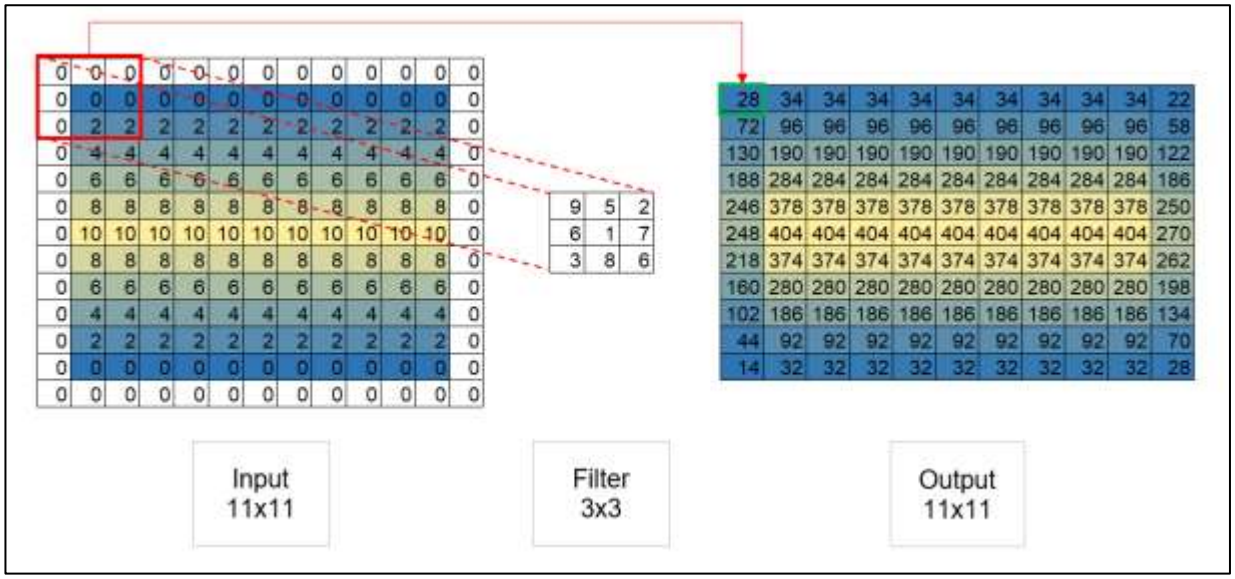

Abbildung 2.3: Faltung eines 11x11 Inputs mithilfe eines 3x3 Filters in einem 11x11 Output mit Zero Padding

Die Anwendung des Zero Paddings in einem Input wird auch „Same Padding“ genannt, während das Weglassen der Methode als „Valid Padding“ bezeichnet wird. Bei der Wahl größerer Faltungsfilter ist es auch möglich, den Außenbereich des Inputs mehrmals mit Nullen für die Erhaltung der Größe aufzufüllen [3].

Nach einer Faltungsschicht folgt oft eine Pooling-Schicht. Hierfür werden je nach CNN und Anwendung die zwei Verfahren Max Pooling und Average Pooling herangezogen. Bei diesen Methoden werden überflüssige Informationen verworfen. Beim Max Pooling wird bei einer bestimmten Gruppe an Zahlen beziehungsweise Neuronen, die das Filter momentan verarbeitet, die höchste Zahl oder das aktivste Neuron ausgewählt. Beim Average Pooling oder auch Mean Pooling genannt, wird der Mittelwert der Zahlen berechnet und in die Pooling-Schicht eingefügt [4].

Beispielhaft geht bei einem 2x2 Filter bis zu 75% der Information verloren, wobei dies auch Vorteile bietet. Auf Grund des verringerten Speicherbedarfs und der erhöhten Berechnungsgeschwindigkeit können Daten schneller verarbeitet und auch tiefere Netzwerke erzeugt werden, mit denen komplexere Aufgaben verarbeitet werden können.

Von den beiden Methoden ist Max Pooling am stärksten verbreitet, da das Mean Pooling sich bei Anwendungen als weniger effizient erwiesen hat und bei Max Pooling, die Neuronen in die Schicht eingefügt werden, die am aktivsten sind und wichtige Merkmale des Inputs (z.B. Formen, Kanten, Ecken in einem Bild) beinhalten. In Abbildung 2.4 ist die Max Pooling Methode in einem Beispiel dargestellt, während in Abbildung 2.5 die Mean Pooling Methode verwendet wird.

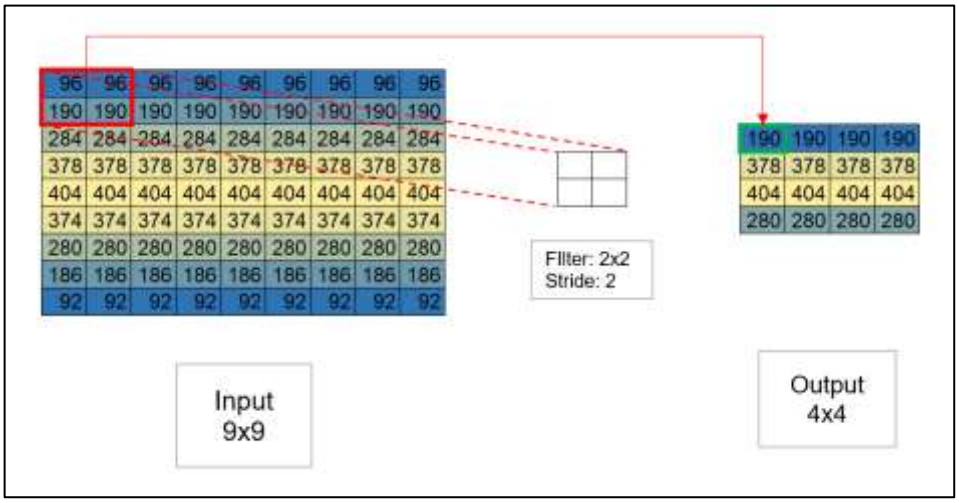

Abbildung 2.4: Umwandlung des 9x9 Inputs in einen 4x4 Output mit der Max Pooling Methode, durch einen 2x2 Filter mit einer Schrittlänge (Stride) von 2

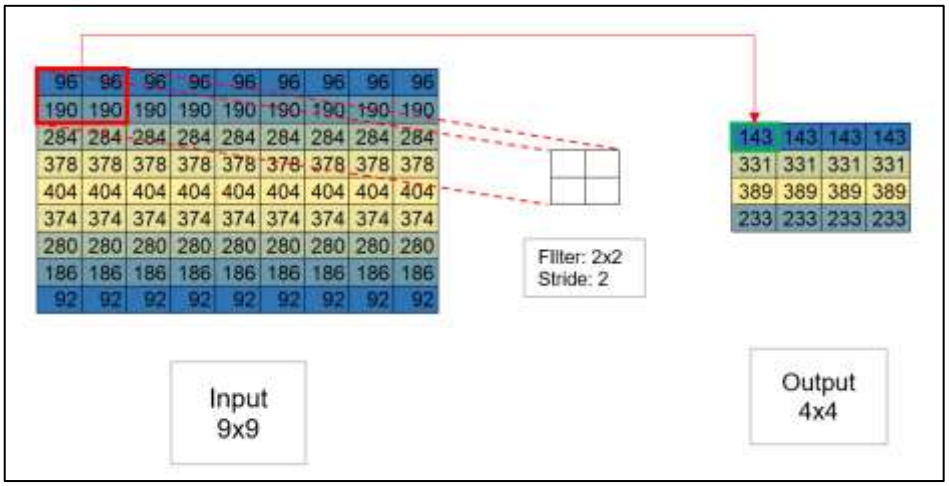

Abbildung 2.5: Umwandlung des 9x9 Inputs in einen 4x4 Output mit der Mean Pooling Methode, durch einen 2x2 Filter mit einer Schrittlänge (Stride) von 2

In den Abbildungen 2.3 und 2.4 wird ein 9x9 Input in einen 4x4 Output mittels Max/Mean Pooling umgewandelt. Da der Input eine Dimension von 9x9 und der Filter eine Schrittlänge (Stride) von 2 hat, wird die äußerste rechte Spalte des Inputs verworfen. Dies kann durch die Wahl eines Inputs mit gerader Länge und Höhe oder eines Filters mit ungerader Länge und Höhe vermieden werden.

Nach einer bestimmten Anzahl von Convolutional- und Pooling-Layer wird häufig ein Fully-Connected-Layer als abschließende Schicht eines Netzes verwendet. In dieser Schicht werden die Eingangsdaten aus den Convolutional- und Pooling-Layern zunächst in einen Vektor umgewandelt, was als „Flattening“ bezeichnet wird. Um sicherzustellen, dass die Ausgabe die richtigen Daten erhält, werden die Daten durch eine Aktivierungsfunktion überprüft (siehe Kapitel 2.1.1). Beispielsweise hat die Rectified Linear Unit (ReLU) einen Schwellwert, bei dem positive Werte ausgegeben werden, während negative Werte durch Null ersetzt werden.

2.1.3 Aufbau eines Fully Convolutional Networks

Neben zweidimensionalen Faltungsnetzen (CNN) existieren auch eindimensionale vollständige Faltungsnetze wie das Fully Convolutional Network (FCN). Ein Unterschied zwischen einem FCN und einem CNN besteht darin, dass ein FCN nur aus Faltungsschichten besteht und Außerdem können FCNs auch transponierte Faltungsschichten (Transposed Convolutional Layers) besitzen, bei denen die verwendeten Filter bei einer bestimmten Eingangsgröße eine größere Ausgangsgröße erzeugen, so dass dieser Schichttyp auch als Deconvolutional Layer oder Entfaltungsschicht bezeichnet wird, wobei der Filter die Inputgröße in eine Outputgröße „entfaltet“ [7].

Beginnend mit dem Eingang des FCN werden die Eingangsdaten durch einen Filter mit einer erweiterten Faltung (dilated convolution) geleitet und an die Faltungsschicht übergeben. Bei dieser Methode werden nicht alle Daten berücksichtigt, wodurch die Rechenleistung und der Speicherbedarf minimiert werden. Die Dilatationsrate (dilation rate) bestimmt dabei, wie stark der Filter aufgeweitet wird und wie viele Eingangsdaten berücksichtigt werden [19]. Abbildung 2.6 zeigt ein Beispiel für eine geweitete Faltung mit zwei unterschiedlichen Dilatationsraten.

Es ist zu erkennen, dass bei einer Erhöhung der Dilatationsrate der 3x1 Filter zu einem 5x1 Filter aufgeweitet wird und „Löcher“ im Filter entstehen. Diese „Löcher“ werden bei der Faltung nicht berücksichtigt, so dass die Faltungsschicht kleiner wird. Um die Verkleinerung der Faltungsebenen zu verhindern, kann Zero Padding verwendet werden, so dass alle Faltungsebenen gleich groß sind.

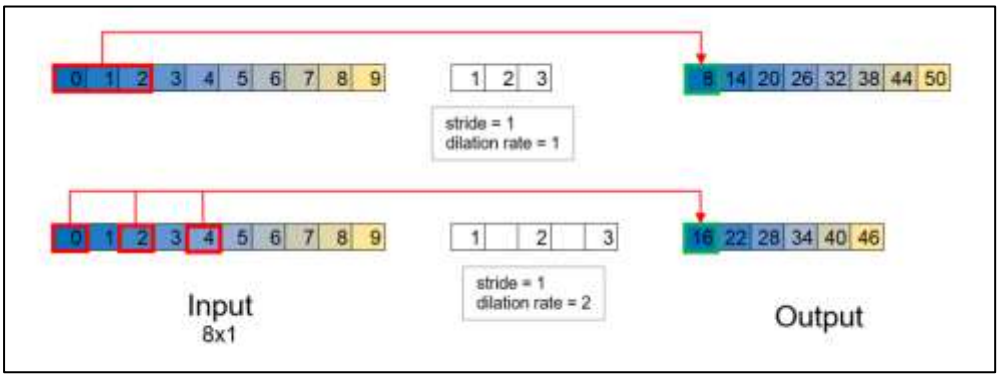

Abbildung 2.6: Beispiel der geweiteten Faltung (dilated convolution) durch die Veränderung der Dilatationsrate (dilation rate)

2.1.4 Anwendung von CNN

CNNs werden am häufigsten bei der Verarbeitung von Bild- und Audiodateien verwendet und kommen z.B. im Sicherheitsbereich zur Gesichts- und Spracherkennung zum Einsatz. Darüber hinaus werden CNNs bei Versicherungsfällen für die Schadensanalyse von Objekten und Naturkatastrophen eingesetzt.

2.2 Single Event Effects

Single Event Effekte (SEE) werden in Halbleiterbauelementen durch ionisierte Strahlung in Form von Alphastrahlung, Neutronenstrahlung oder kosmischer Strahlung ausgelöst, indem ein Teilchen in das Bauelement eindringt oder es durchquert. Dabei gibt es destruktive Effekte (hard errors), die zu einer dauerhaften Schädigung des Bauelementes führen und nicht-destruktive Effekte (soft errors), die zu einem temporären Defekt führen [10] [12] [20]. In dieser Arbeit werden nur die nicht-destruktiven Effekte behandelt. Diese sind Single Event Upsets (SEU) und Single Event Transients (SET).

Ein Single Event Upset entsteht durch das Invertieren eines Bits (Bitflip) in Speichern oder Registern durch ionisierende Strahlung. Je nach Auftreten kann dieser Effekt von einem kleinen Fehler, bei dem der Fehler durch eine Korrektur wieder behoben werden kann, bis zu einem Systemausfall führen [11]. Neben SEUs gibt es auch Multiple Event Upsets (MEUs), bei denen mehrere Bits gleichzeitig invertiert werden.

Ein Single Event Transient wird durch eine Signalstörung an einem Logikgatter erzeugt. Die Auswirkung eines SET kann ganz unterschiedlich sein. Bestenfalls klingt der SET ab bis zu nächsten aktiven Flanke des Taktsignals und hat dann keinen Einfluss auf eine synchrone Schaltung. Schlimmstenfalls breitet sich die Störung im System aus, z.B. wenn Takt oder Resetsignale betroffen sind oder ein nachgeschaltetes Flip-Flop einen verfälschten Logikzustand abspeichert [14].

2.3 VHDL und Verilog

VHDL (Very High Speed Intergrated Circuit Hardware Description Language) und Verilog sind die meistgenutzten Hardwarebeschreibungssprachen, mit denen digitale Systeme textbasiert beschrieben werden können. Einer der bekannten Anwendungen dieser Sprachen ist der Entwurf von digitalen Schaltungen in ASICs oder FPGAs. Der markanteste Unterschied zu Programmiersprachen wie C, Ada, Java usw. besteht darin, dass kein sequenzieller Programmablauf, sondern parallel bzw. nebenläufig arbeitende Hardware beschrieben wird.

2.4 GHDL-Yosys-Plugin

Das GHDL-Yosys-Plugin ist ein von den Entwicklern des Open-Source Synthesewerkzeugs Yosys (Yosys Open Synthesis Suite) entwickeltes Plugin, welches GHDL in Yosys für den Import von VHDL-Designs verwendet. GHDL ist ein VHDL-Simulator und kann VHDL-Designs analysieren, kompilieren und simulieren, während Yosys Verilog-Designs analysieren, synthetisieren und in Hardwarebeschreibungssprachen wie RTLIL (Register Transfer Level Intermediate Language), BLIF (Berkeley Logic Interchange Format), EDIF (Electronic Design Interchange Format), etc. umwandeln und exportieren kann. Yosys verwendet bei diesen Prozessen generische RTLIL-Datenstrukturen. Diese RTLIL-Datenstrukturen können auch

über das GHDL-Plugin befüllt werden. Somit ist es mit dem GHDL-Yosys-Plugin auch möglich, importierte VHDL-Designs in Verilog-Designs umzuwandeln bzw. im Verilog Format zu exportieren. Für weitere Informationen wird auf Quelle [16] verwiesen.

Das Plugin wurde in einer vorangehenden Arbeit für die Übersetzung des CAECO-Beschleunigers von VHDL in Verilog verwendet. GHDL war jedoch nicht in der Lage, einige VHDL-Designs wie die Module des Beschleunigers korrekt und fehlerfrei nach Verilog zu übersetzen, so dass diese Module manuell übersetzt werden mussten.

2.5 Questasim/Modelsim

Questasim ist ein von Siemens EDA (vorher als Mentor Graphics bekannt) entwickelter HDL-Simulator, der ein zur Simulation von in VHDL, SystemVerilog oder Verilog geschriebene Entwürfe unabhängig oder in Verknüpfung mit anderen Simulatoren verwendet werden kann.

Zusätzlich kann ein hierarchischer Entwurf wie zum Beispiel der CAECO, mit einer Testbench verifiziert werden. Das Verhalten einzelner Signale kann in einer Waveform dargestellt und durch Zeitangaben genauer analysiert werden. So ist es auch möglich, Unterschiede zwischen zwei identischen Designs zu erkennen und zu beheben. Mehr dazu unter Quellen [21] und [22].

2.6 Xilinx Vivado

Xilinx Vivado (auch Xilinx Design Suite genannt) ist ein von Xilinx Inc. (jetzt ein Tochterunternehmen von AMD) entwickelter HDL-Simulator, der HDL-Designs simulieren, synthetisieren und implementieren kann. Der Simulator ähnelt Questasim und ermöglicht durch die Darstellung von Signalen in einem Waveform-Fenster eine genauere Analyse. Dabei können Simulatoren wie ModelSim/Questasim, Synopsys VCS oder Cadence Xcelium mit Vivado verbunden werden. Während der Synthese und Implementierung können Constraints als benutzerdefinierte Spezifikationen in Form von SDC- oder XDC-Dateien angepasst werden, so dass das Design benutzerdefiniert synthetisiert und implementiert werden kann. Zusätzlich verfügt Vivado über eigene Bauteilbibliotheken für die Synthese und Implementierung, es können aber auch designspezifische Bibliotheken für das eigene Design hinzugefügt werden. Für weitere Informationen wird auf Quelle [23] verwiesen.

3 Configurable Accelerator Engine for Convolution Operations

Die Configurable Accelerator Engine for Convolution Operations (CAECO) ist ein konfigurierbarer Beschleuniger, der im Rahmen des Pareto Optimal Machine Learning Accelerator (POOMA) Projekts entwickelt worden ist und das Hauptthema dieser Arbeit darstellt. Die Informationen in diesem Kapitel beziehen sich auf die Quellen [6] und [9].

Das POOMA-Projekt selbst, wurde von der Fachhochschule Dortmund in Zusammenarbeit mit der Universität Bremen bearbeitet, mit dem Ziel einen hybriden ASIC in der 22FDX/FDSOI-Technologie von Globalfoundries mit einem RISC-V Prozessorkern und einem Machine Learning IP-Core zu entwickeln. Der RISC-V ist dabei für die Ausführung von allgemeinen Steuerungs- und Kommunikationsaufgaben verantwortlich, während die rechenintensive Ausführung eines Neuronalen Netzes vom IP-Core übernommen wird [8]. Als Machine Learning IP-Core wurde der CAECO entwickelt, bei dem es sich um eine konfigurierbare Hardware-Architektur zur Ausführung eindimensionaler CNNs handelt.

3.1 Architektur

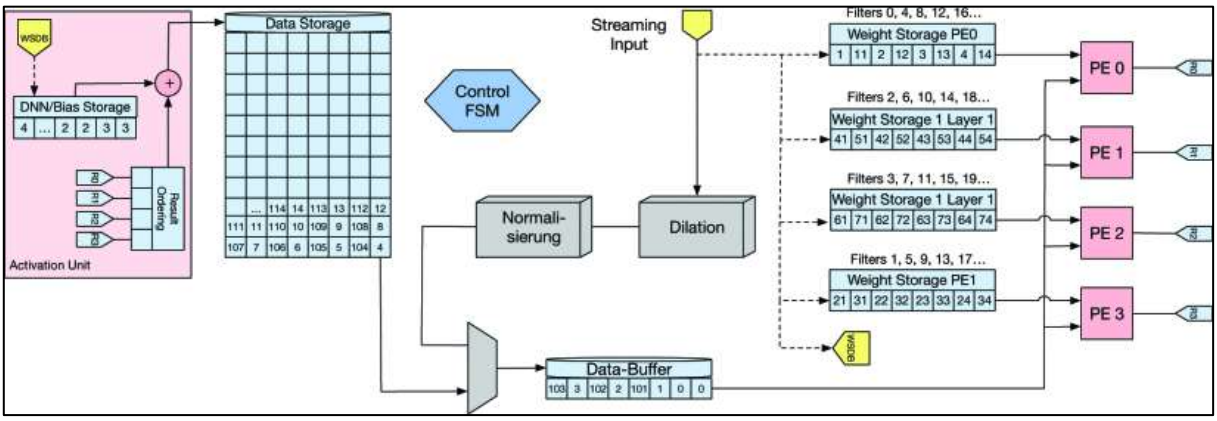

Abbildung 3.1: Schematischer Aufbau von CAECO [6]

Wie in Abbildung 3.1 dargestellt, besteht der CAECO aus folgenden Elementen [6]:

- **Streaming Input:** CAECO bezieht die Eingangs-, Gewichts- und Biasdaten über einen AXI-Bus (Advanced eXtensible Interface). Die Eingangsdaten werden nach Dilatation und Normalisierung in den Datenpuffer geschrieben, während die Gewichts- und Biasdaten direkt in den vier „Weight“-Speichern und dem DNN/Bias-Speicher der Aktivierungseinheit (Activation Unit) abgelegt werden.
- **Datenpuffer (Data Buffer):** speichert Daten aus dem Streaming Input oder dem Daten Speicher (Data Storage) zwischen die in Faltungsoperationen wiederverwendet werden.
- **Gewichtsspeicher (Weight Storage):** besteht aus vier Speicherelementen für Gewichtsdaten und einem Element für den DNN/Bias-Speicher mit einer Datenbreite von jeweils 8 Bit pro Element. Während die vier Speicherelemente Gewichte für die eindimensionale Convolutional Layer speichern, speichert der DNN/Bias-Speicher

Gewichte und Bias-Werte für den fully-connected Layer. Diese können jederzeit durch die Verbindung des Streaming Inputs neu konfiguriert werden.

- **Daten Speicher (Data Storage):** besteht aus vier Speicherelementen mit einer Datenbreite von 32 Bit pro Element. Der Datenspeicher speichert die von den Prozesselementen (PEs) verarbeitete Ergebnisse, nachdem diese Daten durch die ReLU-Aktivierungsfunktion unter Verwendung der Daten aus dem DNN/Bias-Speicher behandelt worden sind.
- **Prozesselemente (PE):** verarbeiten die Daten aus dem Datenpuffer durch Multiply-Accumulate-Rechenoperationen und geben die Ergebnisse zur Aktivierungseinheit weiter.
- **Control FSM:** eine Zustandsmaschine (Finite State Machine), die alle oben benannten Elemente kontrolliert und durch den Übergang in entsprechende Zustände koordiniert.

3.2 Hierarchie

Die Hierarchie des konfigurierbaren Beschleunigers besteht neben dem Hauptmodul caeco.vhd aus mehreren Untermodulen, die in Tabelle 2.1 aufgelistet ist.

Modul	Funktion
caeco.vhd	Hauptmodul mit der top entity „caeco“; Steuert und koordiniert alle anderen Untermodule, die im Hauptmodul instanziiert sind.
caeco_pack.vhd	Package bestehend aus Konstanten, Allzweckfunktionen und Definitionen für Simulation, Synthese, Quantisierung etc.
caeco_pe.vhd	Prozesselement des CAECO, in dem verschiedene Berechnungen und Prozesse stattfinden.
caeco_debug_paths.vhd	Package für den Zugriff der auf die Dateien mit dem ECG-Datensätzen für GHDL, Cadence und Vivado
caeco_weight_storage.vhd	Speichert für „weights“ bzw. Gewichte und Bias-Werte des Neuronalen Netzes; Besitzt eine ASIC-Architektur für einen SRAM-Speicher, eine Debug-Architektur für eine technologieunabhängige Simulation und eine Xilinx-Architektur für die Implementierung im FPGA.
caeco_data_storage.vhd	Datenspeicher, der Daten und Zwischenergebnisse zwischen den Schichten des Neuronalen Netzes speichert. Besitzt wie das Modul caeco_weight_storage auch drei verschiedene Architekturen.
caeco_data_buffer.vhd	Speichert häufig verwendete Daten in einem Puffer für einen schnellen Zugriff.
conv_pkg.vhd	Package bestehend aus Funktionen für Datentypumwandlung, Rundung etc.

Tabelle 3.1: Übersicht auf alle CAECO-Module mit deren individuellen Funktionen.

Neben den in Tabelle 3.1 gelisteten Modulen wurde eine Testbench namens caeco_tb in SystemVerilog entwickelt, die für den Simulationsverlauf der CAECO-Module zuständig ist. Für die Testbench werden auch Dateien mit den ECG-Datensätzen und mem-Dateien

benötigt, welche die Eingabedaten und Gewichte in den Speichern des Beschleunigers vorinitialisieren und Parameter für die Simulation bereitstellen.

Zusätzlich wurde ein Shell-Skript simulate_caeco für Modelsim geschrieben, sodass beim Start des Programms in der Befehlszeile auch eine direkte Simulation mit der Testbench möglich ist, ohne die Notwendigkeit manuell ein Projekt anzulegen.

3.2.1 Hauptmodul caeco

Das Hauptmodul ist im Vergleich zu den anderen Untermodulen mit insgesamt 1836 Zeilen Code das größte Modul des CAECO. Die ist durch die verwendeten Zustandsmaschine (engl. Finite State Machine) begründet, die den Großteil des Moduls ausmacht.

Die entity des Hauptmoduls, welche im Listing 3.1 zu sehen ist, hat sieben Eingangsports, für die Eingabe von Daten, die Ausführung von Berechnungen und das Ein-/Ausschalten der internen Strukturen des Moduls, während drei Ausgangsports für die Ausgabe der Endergebnisse des Neuronalen Netzes zuständig sind. Neben der Port-Liste besitzt sie auch generische konstante Variablen, die in den Ports und Signalen in der Architektur des Moduls Anwendung finden.

```
48 entity caeco is
49     generic (
50         DATA_WIDTH : positive := 32; -- width of the input data, might be
   different from the internally used data width
51         RESULT_WIDTH : positive := 32 -- width of the result port, when a more
   precise prediction than the pure class prediction is required
52     );
53     port (
54         -- the data in port implements the AXI stream protocol
55         DIN : in std_logic_vector(DATA_WIDTH - 1 downto 0);
56         DIN_VALID : in std_logic;
57         DIN_READY : out std_logic;
58         DIN_LAST : in std_logic;
59         -- output ports for the result (RESULT: class that is predicted,
   RESULT_VALID: all computations are finished, we have a valid prediction
   available)
60         RESULT : out std_logic_vector(RESULT_WIDTH - 1 downto 0);
61         RESULT_VALID : out std_logic;
62         -- what should the CAECO do? Store the NN weights or start a prediction
   cycle?
63         -- (i.e., read in data and perform all computations)
64         CMD : in std_logic_vector(1 downto 0);
65         -- enable or disable the internal structures
66         EN : in std_logic;
67         CLK : in std_logic;
68         RSTN : in std_logic
69     );
70 end caeco;
```

Listing 3.1: Entity des CAECO-Hauptmoduls.

In der nachfolgenden Architektur des Moduls werden zuerst in den Zeilen 73 bis 261 Signale für die Festlegung und Auswertung der Zustände in der Zustandsmaschine, für die Verbindung der Untermodule, die Steuerung des Datenflusses und die Ausführung der Berechnung in den Prozesselementen, sowie die finale Klassifizierung der Ergebnisse nacheinander deklariert.

Einige der deklarierten Signale werden, wie im Listing 3.2 dargestellt, mit den Ein- und Ausgangports der instanziierten Untermodule verbunden, damit die Ergebnisse bzw. Daten der Untermodule im Hauptmodul und umgekehrt übertragen und verarbeitet werden können.

```
329   -- the data buffer.. nothin special here
330      data_buffer : entity work.caeco_data_buffer(single_memory)
331      generic map(
332          DATA_WIDTH => DATA_INTERNAL_WIDTH)
333      port map(
334          DIN => s_buffer_din,
335          WE => s_buffer_we,
336          DOUT => s_buffer_dout,
337          RE => s_buffer_re,
338          LAYER_INDEX => std_logic_vector(s_layer_index),
339          NEXT_SAMPLE => s_next_sample,
340          NEXT_FILTER => s_next_filter,
341          CLK => clk,
342          RSTN => s_buffer_rstn
343      );
```

Listing 3.2: Die Instanziierung des Untermoduls caeco_data_buffer im Hauptmodul caeco

Bei der Instanziierung der Module wird für die vier Speicherelemente des Gewichtsspeichers eine generate for-Schleife mit dem Namen „weight_array“ verwendet, die mit den Prozesselementen verbunden werden und Gewichte für die Faltungsschicht beziehungsweise Convolutional Layer bevorraten. Für den DNN/Bias-Speicher wird, wie in Listing 3.2 zu sehen, eine normale Instanziierung mit dem Namen „weight_storage_dnn“ vorgenommen, um die Gewichte des DNN-Layers und alle Bias-Elemente zu speichern. Bei den Prozesselementen wird ebenfalls eine generate for-Schleife mit dem Namen „pe_pipeline“ verwendet, in der zwei generate if-Bedingungen eingeführt worden sind. Eine generate if-Bedingung prüft die Instanziierung des Elements pe_0 bei der Bedingung Index i = 0 während alle anderen Elemente pe_1 bis pe_3, bei Erfüllung der generate if-Bedingung i > 0 instanziiert werden. Der Aufbau des weight_arrays ist im Listing 3.3 und der Aufbau des pe_pipelines im Listing 3.4 dargestellt.

```
350          weight_array : for i in 0 to NUM_PES-1 generate

352          -- write weights, when we want to exchange them
353          s_weight_we_valid(i) <= '1' when s_weight_we(i) = '1' and DIN_VALID='1'
     else
354                                  '0';

356          -- throughout the implementation, we used different architectures..
357          -- these are mapped by including the correct implementation and default
     binding
358          weight_storage_i : entity work.caeco_weight_storage
359          generic map (
360              WEIGHT_WIDTH => WEIGHT_WIDTH,
361              ADDR_WIDTH => WEIGHT_ADDR_WIDTH,
362              WEIGHT_ID => i)
363          port map (
364              W_OUT => s_current_weights(i),
365              W_ADDR => std_logic_vector(s_weight_addr),
366              W_WE => s_weight_we_valid(i),
367              W_IN => s_weight_in,
368              W_RE => s_weight_re(i),
369              CLK => clk,
370              RSTN => rstn
371          );

373      end generate;
```

Listing 3.3: Generate-Statement weight_array für die Instanziierung der Speicherelemente des Moduls caeco_weight_storage (Zeilen 350-373).

```
414     ------------------------------------------------------------------------
415      -- Instation of the PEs (that operate in parallel)
416     ------------------------------------------------------------------------

418     pe_pipeline : for i in 0 to NUM_PES - 1 generate

420          -- for serial computations, we ocassionally use the first PE only
421         generate_pe_0 : if i = 0 generate
422             pe_0 : caeco_pe
423             generic map(
424                 DATA_WIDTH => DATA_INTERNAL_WIDTH,
425                 WEIGHT_WIDTH => WEIGHT_WIDTH,
426                 RESULT_WIDTH => PE_RESULT_WIDTH,
427                 CTRL_WIDTH => PE_CTRL_WIDTH)
428             port map (
429                 X_IN => s_pe_x_in,
430                 W => s_current_weight_pe_0,
431                 Y => s_results(i),
432                 Y_VALID => s_results_valid_z1(i),
433                 CTRL => s_ctrl,
434                 EN => s_pe_en(i),
435                 LAYER_INDEX => s_layer_index,
436                 CLK => CLK,
437                 RSTN => s_pe_rstn);
438
439         end generate;
          . . .
463     end generate pe_pipeline;
```

Listing 3.4: Generate-Statement pe_pipeline für die Instanziierung der Prozesselemente des Moduls caeco_pe (Zeilen 418-463)

Der caeco_data_storage wird erst am Ende der Datei nach der Zustandsmaschine ab der Zeile 1502 instanziiert.

Nach den Instanziierungen befindet sich im Code ab der Zeile 470 ein Prozessblock namens „dilated convolution“. Dieser Prozess wird verwendet, um nicht jeden Datenpunkt zu verarbeiten, sondern nur Datenpunkte bei einer gewissen Schrittweite in die Faltung mit einzubeziehen. Der Prozess wird aktiviert, wenn die Schrittweite (dilation stride) länger als eins gewählt wird. Wenn diese Bedingung nicht erfüllt ist und die Schrittlänge gleich eins ist, bleibt der Prozessblock mit der Meldung „dilation is not activated“ inaktiv.

Im Prozessblock „combined_sample_inputs“ (Zeile 545 ff.) werden die Inputdaten in zwei „Gruppen“ aufgeteilt, für den Fall, dass der Dateneingang DIN aus zwei Datenpunkten besteht. Dafür muss die Konstante DOUBLE_SAMPLE_INPUTS, die im caeco_pack deklariert worden ist, gesetzt sein. Falls die Konstante nicht gesetzt ist, wird der Prozessblock „single_sample_inputs“ (Zeile 582 ff.) aktiviert.

Anschließend wird eine Normalisierung der Daten durchgeführt (Zeile 598 ff.), bei der die Offsets korrigiert und die Datenpunkte skaliert werden. Bei der Aktivierung des Prozessblocks „data_normalization_active“ wird zuerst mit einem Generate-Statement eine Offset-Normalisierung der Daten ausgeführt, der die Daten mit einem Normalisierungsfaktor von einer Binärzahl in eine Dezimalzahl umwandelt. Danach wird mit den geänderten Daten eine Offset-Korrektur mit einer resize-Funktion durchgeführt. Anschließend folgt eine Skalierung der Input-Daten zu einer optimalen und vorkonfigurierten Anzahl und Breite der Fraktionsbits mit der convert_type-Funktion vom conv_pkg Package, indem die verarbeiteten Daten als

skalierte Zwischenergebnisse des Prozesses in die Funktion eingefügt werden. Der Prozessblock selbst wird erst aktiv, wenn die Konstante NORMALIZE_INPUT_DATA den „true“ beziehungsweise wahr besitzt. Falls dies nicht der Fall ist, bleibt dieser Prozessblock deaktiviert.

Alle bisher erwähnten Prozesse werden in generate-Blöcken behandelt und gegebenenfalls instanziiert. Das bedeutet, dass die Instanziierung des Moduls nur erfolgt, wenn die if-else-Bedingung erfüllt wird.

In den Zeilen 653 bis 687 befinden sich noch weitere when-else-Bedingungen, die funktional mit if-else-Bedingungen identisch sind, aber thematisch bzw. funktional in keine der zuvor genannten Prozessen eingeordnet werden können.

Ab der Zeile 698 befindet sich mit 1480 Zeilen Quellcode der größte Prozessblock des Hauptmoduls. Dieser Prozessblock entspricht der Zustandsmaschine, welche für die Kontrolle und Koordinierung der Untermodule und Prozesse zuständig ist.

Zu Beginn dieses Prozessblocks, der in Listing 3.5 zu sehen ist, werden zuerst die next_sample und einige counter_full-Variablen deklariert, die bei positiven Flanken von CLK bzw. negativen Flanken von RSTN die Reihenfolge der Zustände koordinieren. Anschließend wird eine if-elseif-Anweisung verwendet, um bei RSTN = 0 (active-low reset) die Signale und Register auf ihre voreingestellte Werte zu setzen. Bei einer steigenden Taktflanke von CLK bzw. bei RSTN = 1 behalten einige Signale ihre momentanen Werte und Einstellungen, während andere Signale auf null zurückgesetzt werden. Zusätzlich wird die case-Anweisung ausgeführt, in der 19 von 20 Zustände für die Steuerung des Faltungsneuronalen Netzes verwendet werden.

```
698   process (CLK, RSTN)
699           -- some helper variables to control the flow of computation
700           variable next_sample : boolean;
701           variable sample_counter_full : boolean;
702           variable step_counter_full : boolean;
703           variable conv_counter_full : boolean;
704           variable bunch_counter_full : boolean;
705       begin
706           if RSTN = '0' then
707               -- set all registers to predefined default values
                   . . .
745           elsif (rising_edge(CLK)) then
747               -- performing various default assignments to set or perserve the
      state of internal signals
                   . . .
797               case dnn_state is
                   -- building and performing the neural network with its 19 different
                   states from IDLE to FINISHING
                   . . .
1486              end case;
1487          end if;
1488      end process;
```

Listing 3.5: Aufbau und Übersicht der Zustandmaschine per case-Anweisung in einem Prozessblock (Zeilen 698 - 1488)

Vor der Erklärung der einzelnen Zustände der Zustandsmaschine, wird zuerst die Vorgehensweise beim Übergang eines Zustands zu einem anderen beschrieben. Dazu wird

beispielhaft der Übergang vom Zustand ZERO_PADDING_BEGIN zu READ_SAMPLES_BEGIN betrachtet, der im Listing 3.6 zu sehen ist. In Abbildung 3.2 ist zusätzlich zur Visualisierung des Überganges eine Waveform dargestellt, welche den Verlauf von relevanten Variablen und Signale zeigt.

Zu Beginn des Zustands ZERO_PADDING_BEGIN werden in den Zeilen 847 und 849 sample_counter_full und step_counter_full Bedingungen zugewiesen, bei denen die Zählvariablen s_sample_counter und s_step_counter Werte erhalten. Die Bedingung ist erfüllt, wenn s_step_counter und s_sample_counter den zuvor definierten und zugewiesenen Wert annehmen, so dass die counter_full-Signale auf „1“ schalten. Zwischen den Zeilen 851 und 871 befindet sich ein if-elsif-Konstrukt, das aus drei verschiedenen Fällen besteht. Im ersten Fall (Zeilen 851 - 859) findet ein Zustandsübergang statt, wenn beide counter_full-Signale aktiv sind. Die Zähler s_step_counter und s_sample_counter werden auf null zurückgesetzt und das Signal dnn_state wechselt in den Zustand READ_SAMPLES_BEGIN. Im zweiten Fall (Zeile 860 - 865) wird s_sample_counter einmal inkrementiert und s_step_counter auf null zurückgesetzt, wenn step_counter_full aktiv ist. Die Zustandsmaschine verbleibt dabei im Zustand ZERO_PADDING_BEGIN. Im letzten Fall (Zeilen 866 - 871) zählt s_step_counter weiter und der aktuelle Zustand bleibt ebenfalls erhalten, wenn keines der counter_full Signale aktiv ist.

```
841  when ZERO_PADDING_BEGIN =>
842  -- this is the first real computing state, where we apply zero padding to the data
     to preserve the size after the convolutions
843  -- we use the different counters to count the filter steps and number of samples
     that we have to insert
844        s_storage_re <= '1'; -- read data from the data storage

846        -- count the number of zero samples we have to insert
847        sample_counter_full := s_sample_counter =
     NUM_PAD_SAMPLES_START(to_integer(s_layer_index)) - 1;
848        -- count the number of input data channels we have to use in zero padding
849        step_counter_full := s_step_counter =
     NUM_CHANNELS(to_integer(s_layer_index)) - 1;

851        if sample_counter_full and step_counter_full then
852        -- we have padded all channels (and added enough zeros to the current
     channel)
853              s_sample_counter <= (others => '0');
854              s_storage_dout_address <= s_storage_dout_address + 1;
855              s_step_counter <= (others => '0');
856              s_weight_addr <=
       WEIGHT_STORAGE_START_POSITIONS(to_integer(s_layer_index));
857              s_dilation_active <= '1';
858              dnn_state <= READ_SAMPLES_BEGIN;
859              state_debug <= 2;
860        elsif step_counter_full then
861              -- we have added enough samples in the current channel
862              dnn_state <= ZERO_PADDING_BEGIN;
863              s_sample_counter <= s_sample_counter + 1;
864              s_step_counter <= (others => '0');
865              state_debug <= 1;
866        else
867              -- continue as usual, add zeros to the begin of the data
868              s_step_counter <= s_step_counter + 1;
869              dnn_state <= ZERO_PADDING_BEGIN;
870              state_debug <= 1;
871        end if;
```

Listing 3.6: Aufbau des Zustandes ZERO_PADDING_BEGIN in einer case-Anweisung

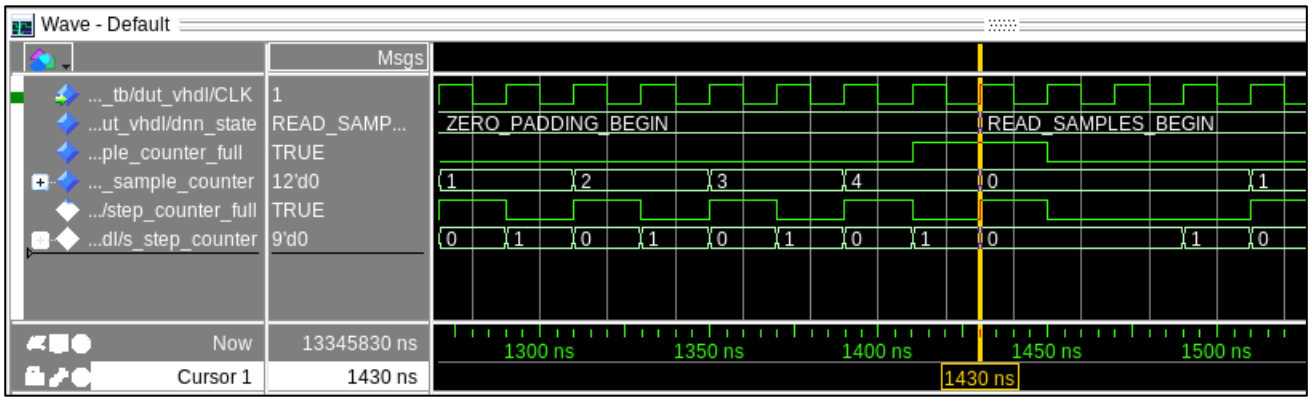

Abbildung 3.2: Visualisierung des Übergangs zwischen den Zuständen ZERO_PADDING_BEGIN und READ_SAMPLES_BEGIN in einer Waveform.

Das faltungsneuronale Netz wird mit Hilfe von 20 Zuständen gesteuert, welche Einfluss auf die Bearbeitung der Daten in 6 verschiedenen Schichten nehmen, die mit dem LAYER_INDEX von 0 bis 4 und 7 nummeriert sind.

In Tabelle 3.2 ist eine Überschicht der Schichten mit den zugehörigen Zuständen dargestellt.

LAYER_INDEX (hexadezimal)	Schicht	Zugehörige Zustände (in Reihenfolge)
0	1. Faltungsschicht (Convolutional Layer)	ZERO_PADDING_BEGIN, READ_SAMPLES_BEGIN, CONV_LAYER, READ_SAMPLES, CONV_NEXT_BUNCH_IDLE, ZERO_PADDING_END, ZERO_PADDING_END_NEXT_BUNCH_IDLE, READ_SAMPLES_ZERO_PADDING_END, CONV_NEXT_LAYER
1	2. Faltungsschicht (Convolutional Layer)	
2	3. Faltungsschicht (Convolutional Layer)	
3	Pooling-Schicht (Pooling Layer)	MAX_POOLING_STARTUP, MAX_POOLING, MAX_POOLING_NEXT_SAMPLE, MAX_POOLING_FINALIZE,
4	Völlig verbundene Schicht (fully-connected Layer)	FC_STARTUP, FC_LAYER, FC_NEXT_NEURON, FC_FINALIZE
7	Leerlaufschicht/Output	IDLE (allererster Zustand), FINISHING (allerletzter Zustand)

Tabelle 3.2: Übersicht der Schichten und der Zustände des eindimensionalen faltungsneuronalen Netzes

Auf Grund der Tatsache, dass drei Faltungsschichten in diesem Netz verwendet werden, kann man es auch als Deep Fully Convolutional Network (DFCN) oder tiefes faltungsneuronales Netz bezeichnen.

In den Zuständen INITIALIZE_STORE_WEIGHTS und STORE_WEIGHTS werden die Gewichte und Bias-Werte in den Gewichtsspeichern abgelegt. In der verwendeten Simulation werden diese Zustände der Zustandsmaschine jedoch nicht angesteuert, da von vorinitialisierten Gewichtsspeichern ausgegangen wird.

Neben der Zustände zur Steuerung der Schichten existieren noch die Zustände IDLE und FINISHING. Der Zustand IDLE ist ein Leerlaufszustand. Hier wird auf eine Änderung des

Eingangssignals CMD gewartet. Bei CMD = „01“ erfolgt ein Zustandsübergang zu ZERO_PADDING_BEGIN beziehungsweise bei „10“ ein Übergang zu STORE_WEIGHTS. Im Zustand FINISHING werden die berechneten Endergebnisse angezeigt und anschließend in den Zustand IDLE gewechselt.

Die Steuerung der drei Faltungsschichten (Zeilen 841 - 1262) besitzen einen identischen Ablauf. Zunächst wird mit dem Zustand ZERO_PADDING_BEGIN begonnen. In diesem Zustand wird Zero Padding auf die Daten aus dem Datenspeicher angewendet. Dadurch bleibt die Größe aller in den drei Schichten verarbeiteten Datensätze gleich. Falls Zero Padding angewendet wird, wird im Zustand READ_SAMPLES_BEGIN eine bestimmte Anzahl an Datenpunkten ausgelesen und im Datenpuffer für die Bearbeitung in späteren Zustände gespeichert. Dabei wird beim Auslesen auch Zero Padding auf die Daten angewendet. Anschließend wird in den „Hauptzustand“ CONV_LAYER für die Steuerung der Faltungsschicht übergegangen. Hier werden die vom Datenpuffer und Gewichtsspeicher gespeicherten Daten und Gewichte für die Berechnungen der Prozesselemente verwendet. Nach den ersten Berechnungen im CONV_LAYER, wird in den Zustand CONV_NEXT_BUNCH_IDLE mit jedem Impuls des Signals step_counter_full gewechselt. Hier werden die Zwischenstände der letzten Faltungsberechnungen von Daten und Gewichte im Datenpuffer für die nächste Faltungsberechnung zwischengespeichert. Danach geht die Zustandsmaschine wieder in den Zustand CONV_LAYER über. Neben dem Zustand CONV_NEXT_BUNCH_IDLE gibt es noch den Zustand READ_SAMPLE, der erst bei jedem Hochzählen des s_sample_counter Signals und bei jedem Impuls des Signals step_counter_full aktiviert wird. Dieser Zustand ähnelt dem Zustand READ_SAMPLES_BEGIN bis auf den Unterschied, dass die berechneten Daten statt Zero Padding, mit den vorherigen Daten kombiniert im Datenpuffer gespeichert werden.

Wie in Abbildung 3.3 zu sehen ist, wiederholt sich bei jedem Zustandsübergang zu CONV_LAYER dieser Zyklus, wenn s_sample_counter hochgezählt wird.

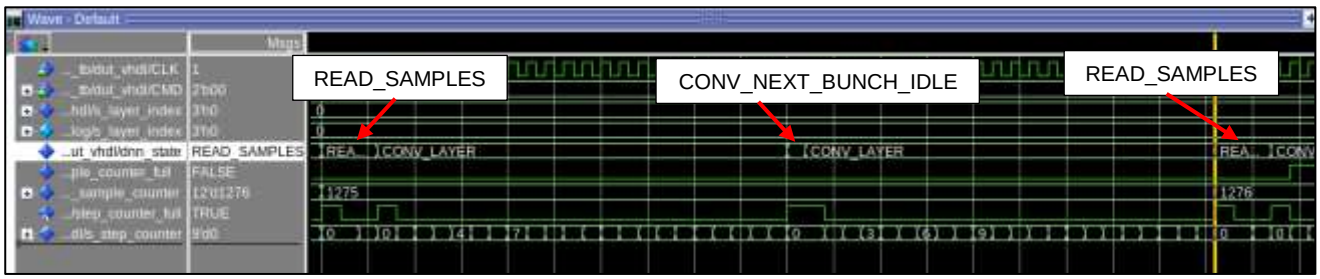

Abbildung 3.3: Die Berechnung und Verarbeitung der Daten im Zustand CONV_LAYER mit den Zuständen READ_SAMPLES und CONV_NEXT_BUNCH_IDLE

Nachdem das Signal s_sample_counter den erforderlichen Wert für sample_counter_full erreicht hat, geht die Zustandsmaschine vom Zustand CONV_LAYER in den Zustand ZERO_PADDING_END über, bei dem mit einem bestimmten Filter die Faltung mit dem Zero Padding ausgeführt wird. Dadurch werden die im CONV_LAYER verarbeiteten Daten in der Faltungsschicht gespeichert. Wie in Abbildung 3.3 zu sehen, gibt es auch bei

ZERO_PADDING_END Zyklen, bei denen die Zustände ZERO_PADDING_END_NEXT_BUNCH_IDLE und READ_SAMPLE_ZERO_PADDING_END vorkommen. Diese beiden Zustände sind funktional den Zuständen CONV_NEXT_BUNCH_IDLE und READ_SAMPLES bis auf die Anwendung des Zero Paddings am Ende des Prozesses sehr ähnlich. Zuletzt wird der Zustand CONV_NEXT_LAYER ausgeführt, in dem die verarbeiteten Endergebnisse der Faltungsschicht für die Pooling-Schicht weiterverwendet werden.

Nach den drei Faltungsschichten kommt die Pooling-Schicht zum Einsatz (Zeilen 1264 - 1361). Hier wird mit dem Zustand MAX_POOLING_STARTUP das Max-Pooling-Verfahren vorbereitet. Es wird gewartet, bis die endgültigen Ergebnisse der Faltungsschichten zur weiteren Verwendung eintreffen. Im Zustand MAX_POOLING wird danach das Max Pooling bei den Daten ausgeführt, in dem abhängig vom Filter der aktivste beziehungsweise größte Wert gespeichert wird. Dadurch gehen bis zu 75% der Daten verloren. In jedem Impuls des Signals sample_counter_full im Zustand MAX_POOLING, wird der Zustand MAX_POOLING_NEXT_SAMPLE aktiv, bei dem die Pooling-Werte gespeichert werden. Nach der Speicherung wird der Zustand MAX_POOLING aktiv und setzt das Max Pooling-Verfahren bis zum nächsten Impuls von sample_counter_full fort. Wenn die Signale sample_counter_full und step_counter_full gesetzt sind, nimmt die Zustandsmaschine zuletzt den Zustand MAX_POOLING_FINALIZE an. Hier wird gewartet, bis alle Daten mit Max Pooling verarbeitet wurden, sodass die finalen Daten in der völlig verbundenen Schicht (fully-connected Layer) weiterverwendet werden.

Als Letztes steuert die Zustandsmaschine die Vorgänge in der völlig verbundenen Schicht (Zeilen 1363 - 1455), angefangen mit dem Zustand FC_STARTUP, bei dem die gespeicherten Daten und Gewichte aus der Pooling-Schicht und dem Gewichtsspeicher abgerufen und in die Pipeline der Prozesselemente gespeist werden, damit die Verarbeitung in dieser Schicht starten kann. Im Zustand FC_LAYER werden Klassen beziehungsweise „classes“ berechnet, indem jeder einzelner Datenwert eines Neurons mit einem korrespondierendem Gewichtswert multipliziert und alle verarbeiteten Inputdaten des Neurons akkumuliert, also angesammelt werden, um die Ausgangsgröße bzw. Endergebnis der Klassifizierung zu erhalten. Zwischendurch wird durch Setzen des Signals sample_counter_full der Zustand FC_NEXT_NEURON aktiv, bei dem die nächsten Neuronen für die Verarbeitung im FC_LAYER abgerufen werden. Sobald die Ausgangsgröße aller Neuronen verarbeitet wurden, erreicht die Zustandsmaschine als letztes den Zustand FC_FINALIZE, bei dem auf die Endergebnisse der Neuronen für den Zustand FINISHING gewartet wird.

Nach der Zustandmaschine sind noch Prozesse wie logging_relu, max_pooling_proc und result_collector im Code des Hauptmoduls vorhanden. Im logging_relu Prozess (Zeile 1546 ff.) wird der Layer-Index der Schicht des neuronalen Netzes angezeigt, in der die ReLU-Funktion angewendet werden soll. Vor dem Prozess werden durch Zuweisungen die Bias-Werte hinzugefügt, umgewandelt, in die Hälfte geteilt und zugewiesen, weil die Bias-Werte aufgrund ihrer 16-Bit-Breite eine größere Wertedifferenz als andere Daten aufweisen und bei ihrer Verarbeitung hohe Verluste auftreten können. Nach dem logging_relu Prozess (Zeile

1566 ff.) wird mithilfe der convert_type-Funktion die ReLU-Funktion auf die entsprechende Schicht angewendet, indem die bearbeiteten Werte eingeschränkt und auf eine bestimmte Bit-Breite gerundet werden.

Im max_pooling_proc, also dem vorletzten Prozess des Hauptmoduls (Zeile 1588 ff.), werden die vom Max Pooling-Verfahren gefilterten Werte zusätzlich angepasst bzw. für die spätere Sammlung der Ergebnisse aufbereitet.

Im letzten Prozess des Hauptmoduls result_collector (Zeile 1613 ff.) werden die Ergebnisse der drei Faltungsschichten und der Pooling-Schicht gesammelt, und für die Ausgabe als Endergebnisse verarbeitet. Hierbei werden im Faltungsteil des neuronalen Netzes, also während die Zustandsmaschine mit der Steuerung einer Faltungsschicht beschäftigt ist, die gesammelten Daten synchronisiert und mit Schieberegistern von den Prozesselementen zum Datenspeicher übertragen.

Im Prozess generate_results_log (Zeile 1768 ff.) werden die Ergebnisse der Signale s_activation für die Aktivierungswerte (activation), s_storage_din für die normalisierten bzw. skalierten Daten im Datenspeicher (caeco_data_storage), s_relu für die von der ReLU-Funktion umgewandelten Werte (relu) und die Zeiteinheiten (ts) in Textdateien gespeichert.

3.2.2 Untermodul caeco_pe

Das Untermodul caeco_pe ist das Prozesselement des CAECO und wird im Hauptmodul caeco als Instanz pe_pipeline verwendet. In diesem Modul finden arithmetische Operationen und Berechnungen statt.

```
30 entity caeco_pe is
31     generic (
32         DATA_WIDTH      : positive;
33         WEIGHT_WIDTH    : positive;
34         RESULT_WIDTH    : positive;
35         CTRL_WIDTH      : positive);
36     port (
37         X_IN : in std_logic_vector(DATA_WIDTH-1 downto 0);
38         W : in std_logic_vector(WEIGHT_WIDTH-1 downto 0);
39         Y : out std_logic_vector(RESULT_WIDTH-1 downto 0);
40         Y_VALID : in std_logic;
41         CTRL : in std_logic_vector(CTRL_WIDTH-1 downto 0);
42         EN : in std_logic;
43         LAYER_INDEX : in unsigned(bits_for_size(NUM_TOTAL_LAYERS)-1 downto 0);
44         CLK : in std_logic;
45         RSTN : in std_logic
46    );
47 end caeco_pe;
```

Listing 3.7: Die entity beziehungsweise Entität des Untermoduls caeco_pe

Die entity aus dem Listing 3.7 besteht aus 8 Eingangsports, einem Ausgangsport und 4 generische Konstanten.

In der Architektur des Moduls, werden zuerst einige Signale für Daten, Gewichte, Zwischenergebnisse der Berechnungen und die Koordination der Schichten deklariert (Zeilen 51 - 68). Die Signale werden anschließend im Berechnungsprozess verwendet (Zeilen 77 - 172), wo für die einzelnen Schritte des Prozesses temporäre Daten oder

Akkumulierungswerte als Signale deklariert werden. Diese temporären Werte werden am Ende der Rechnung den zugehörigen Signalen zugewiesen. Für die Berechnung und Anpassung der Daten wird die convert_type-Funktion angewendet. Damit die Berechnung oder Akkumulierung der Daten und Produkte korrekt zur richtigen Schicht (LAYER_INDEX) oder Berechnung zugewiesen werden können, wird das Signal s_ctrl_z1 für die Steuerung und Koordinierung verwendet. Falls die Berechnung und Akkumulierung erfolgt ist, werden die temporären Werte als Ergebnis dem Signal s_result zugewiesen, der mit dem Ausgangsport verbunden ist. Im Listing 3.8 ist die verkürzte Version des Berechnungsprozesses dargestellt.

```
77      -- main computation process,
78      mac_process : process(CLK, RSTN)
79          variable temp_acc : signed(PE_ACC_WIDTH-1 downto 0);
80          variable temp_prod : signed(TEMP_PROD_WIDTH-1 downto 0);
81      begin
82          if RSTN = '0' then
                -- signals will be turned to zero or IDLE state
90          elsif rising_edge(clk) then
                -- signals holding their default value or state
98              if EN = '1' then
99                  if s_ctrl_z1 = WX or s_ctrl_z1 = WXA then
100                     -- compute product
101                     temp_prod := s_w * s_x_in;
102                     -- round product, depending on the layer
103                     if LAYER_INDEX = 0 then
104                         s_prod <=
                        signed(convert_type(std_logic_vector(temp_prod),. . .));
107                     elsif LAYER_INDEX = 1 then  . . .
111                     elsif LAYER_INDEX = 2 then  . . .
115                     elsif LAYER_INDEX = 4 then  . . .
119                     else
120                         s_prod <= (others => '0');
121                     end if;
122                 elsif s_ctrl_z1 = A then
123                     -- just do accumulation and store a new value in s_prod
                        -- same procedure as above, but with the signal s_x_in instead
                        of temp_prod
140                     else
141                         s_prod <= (others => '0');
142                     end if;
143                 end if;
                    -- dealing with product, accumulator and computing a result
                        . . .
163             else
164                 -- default assignments: store the current values
170             end if;
171         end if;
172     end process;
```

Listing 3.8: Der Berchnungsprozess im caeco_pe, wo die eingegangenen Daten hier verarbeitet werden

3.2.3 Untermodule caeco_weight_storage und caeco_data_storage

Während der caeco_weight_storage die Gewichte und Bias-Werte des neuronalen Netzes speichert und für die Berechnungen zur Verfügung stellt, speichert der caeco_data_storage die Zwischenergebnisse der einzelnen Schichten des Netzes.

Bei diesen beiden Modulen sind in den Listings 3.9 und 3.10 nur die entity gelistet, da sie drei verschiedene Architekturen besitzen. Diese weisen verschiedene Funktionen auf:

ASIC-Architektur:

- **caeco_weight_storage_asic:** Besitzt eine Signal-Deklaration Namens s_addr und eine Instanziierung eines SRAM-Speichers mit einer Kapazität von 1024x8 Bits. Das Signal s_addr wird mit dem Eingangsport W_ADDR verbunden, während die Portliste des SRAM-Speichers mit der Portliste der entity verbunden wird.
- **caeco_data_storage_asic:** Initialisiert für das Hauptmodul vier SRAM-Speicher mit einer Kapazität von 4096x32 Bits pro Element, in dem Zwischendaten des neuronalen Netzes gespeichert und für spezielle Prozesse wiederverwendet werden.

DEBUG-Architektur:

- **caeco_weight_storage_debug:** Ladet Gewichte aus mem-Dateien und speichert sie für die Verwendung beziehungsweise Austausch in einem adressdefinierten oder größendefinierten als Array beschreibenen RAM-Speicher.
- **caeco_data_storage_debug:** Erzeugt einen einfachen RAM-Speicher als Array, wo Daten gespeichert und verwendet werden können.

XILINX-Architektur:

- Bei beiden Modulen findet sich eine leere Architektur, die keinerlei Funktionen besitzt. Diese Module werden bei der Synthese mit dem Synthesewerkzeug von Xilinx Vivado automatisch mit Speicherprimitiven ersetzt, welche in Xilinx Bausteinen vorhanden sind.

```
31 entity caeco_weight_storage is
32     generic (
33         WEIGHT_WIDTH : positive;
34         ADDR_WIDTH : positive;
35         WEIGHT_ID  : natural := 0;
36         WEIGHT_STORAGE_SIZE  : natural := 0
37     );
38     port (
39         W_OUT : out std_logic_vector(WEIGHT_WIDTH - 1 downto 0);
40         W_IN : in std_logic_vector(WEIGHT_WIDTH - 1 downto 0);
41         W_ADDR : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
42         W_RE : in std_logic;
43         W_WE : in std_logic;
44         CLK : in std_logic;
45         RSTN : in std_logic
46     );
47 end caeco_weight_storage;
```

Listing 3.9: Die entity des Untermoduls caeco_weight_storage

```
32  entity caeco_data_storage is
33      generic (
34          DATA_WIDTH : positive; -- width of the stored elements
35          ADDR_WIDTH : positive); -- width of the adress
36      port (
37          DIN : in std_logic_vector(DATA_WIDTH - 1 downto 0);
38          DIN_ADDR : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
39          WE : in std_logic;
40          DOUT : out std_logic_vector(DATA_WIDTH - 1 downto 0);
41          DOUT_ADDR : in std_logic_vector(ADDR_WIDTH - 1 downto 0);
42          RE : in std_logic;
43          CLK : in std_logic;
44          RSTN : in std_logic
45      );
46  end caeco_data_storage;
```

Listing 3.10: Die entity des Untermoduls caeco_data_storage

3.2.4 Untermodul caeco_data_buffer

Das Modul ist ein Puffer, der Daten speichert, die in den Prozessen wiederverwendet werden sollen. Das Ziel besteht darin den Zugriff auf die Daten im Puffer schneller und effizienter als auf Daten im Datenspeicher zu gestalten.

Der Puffer ist als Ringspeicher mit zwei Prozessblöcken (writer, reader) aufgebaut, der an einer Stelle Daten speichert und an anderer Stelle Daten ausgibt. Falls der Speicher voll ist, werden die ältesten Daten von neueren Daten überschrieben. Im Prozessblock „writer“ (Zeilen 65 - 79) werden die Daten im Puffer geschrieben, während im Prozessblock „reader“ (Zeilen 84 - 109) die Daten an der gewünschten Ausleseposition ausgelesen werden. Bei der Wiederverwendung von Daten wird die vorherige Ausleseposition gelöscht. Im Listing 3.11 ist die entity des Moduls dargestellt.

```
29  entity caeco_data_buffer is
30      generic (
31          DATA_WIDTH : positive);
32      port (
33          -- data write ports
34          DIN : in std_logic_vector(DATA_WIDTH - 1 downto 0);
35          WE : in std_logic;
36          -- data read ports
37          DOUT : out std_logic_vector(DATA_WIDTH - 1 downto 0);
38          RE : in std_logic;
39          -- meta information: do we continue with the next sample/filter, in which
    layer are we currently?
40          LAYER_INDEX : in std_logic_vector(bits_for_size(NUM_TOTAL_LAYERS) - 1
    downto 0);
41          NEXT_SAMPLE : in std_logic;
42          NEXT_FILTER : in std_logic;
43          -- general ports
44          CLK : in std_logic;
45          RSTN : in std_logic
46      );
47  end caeco_data_buffer;
```

Listing 3.11: Die entity des Untermoduls caeco_data_buffer

3.2.5 Paketmodule caeco_pack, caeco_debug_paths und conv_pkg

Die Module caeco_pack, caeco_debug_paths und conv_pkg sind Paketmodule, welche mit den Schlagworten „package“ und „package body“ analog zu entity und architecture in

regulären Modulen definiert werden. Im Baustein „package“ werden Funktionen und Konstanten deklariert, während sie im „package body“ implementiert bzw. verwendet werden.

Das Modul caeco_pack beinhaltet generische Konstanten und Funktionen, die in den CAECO-Modulen verwendet werden. Im Modul conv_pkg ist die komplexe convert_type-Funktion implementiert, die im Hauptmodul caeco und im Untermodul caeco_pe mehrmals für die Berechnung der Daten in den Prozesselementen verwendet wird. Im Modul caeco_data_paths wurden für den Zugriff auf die ecg- (Datensätze) und den mem-Dateien (Gewichte) während einer Simulation mit einer Testbench Dateipfade als Konstanten hinterlegt.

4 Triple Modular Redundancy

Die Methode der dreifach modularen Redundanz (Triple Modular Redundancy, TMR) verleiht einem elektronischen Design Immunität gegen Single Event Effekte, in dem die Logik des Designs tripliziert wird. Die Triplizierung kann benutzerdefiniert erfolgen, wobei bei partiellen Triplizierungen sogenannte Voter und Fanouts eingesetzt werden (Abbildung 4.1), die eine Verbindung zwischen triplizierten und nicht-triplizierten Signalen herstellen.

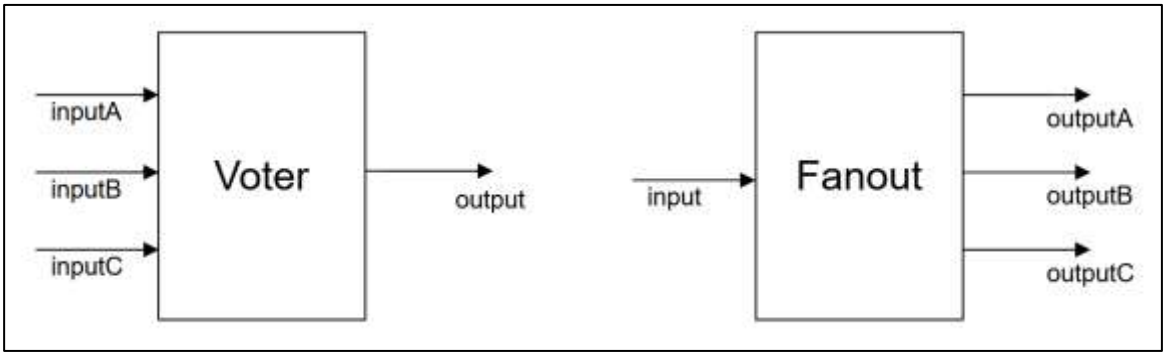

Abbildung 4.1: Darstellung eines Voters und eines Fanouts.

Wie in Abbildung 4.1 zu sehen ist, wird ein Voter verwendet, um ein tripliziertes Signal mit einem nicht-triplizierten Signal zu verbinden, während es bei einem Fanout umgekehrt ist.

Durch die Triplizierung des Entwurfs kann ein Schutz gegen SEEs eingeführt werden. Bei einem Fehler in einem von drei Speicherelementen kann das triplizierte Design weiterhin funktionieren. Dafür werden MajorityVoter am Ausgang der Logik bzw. der FlipFlops implementiert. Diese Voter sorgen dafür, dass der korrekte Wert am Output ausgegeben wird, wenn eine Mehrheit der triplizierten Signale diesen Wert besitzt. Der vom MajorityVoter ausgegebene Wert wird an die Logik weitergeleitet und in die Speicherelemente zurückgekoppelt, um eine Akkumulation von Fehlern zu verhindern, so dass eine Korrektur erfolgt. Abbildung 4.2 zeigt schematisch eine vollständige Triplizierung mit drei MajorityVoter. Dieser Korrekturmechanismus funktioniert nur so lange, wie nur ein von drei Signalen verfälscht worden ist.

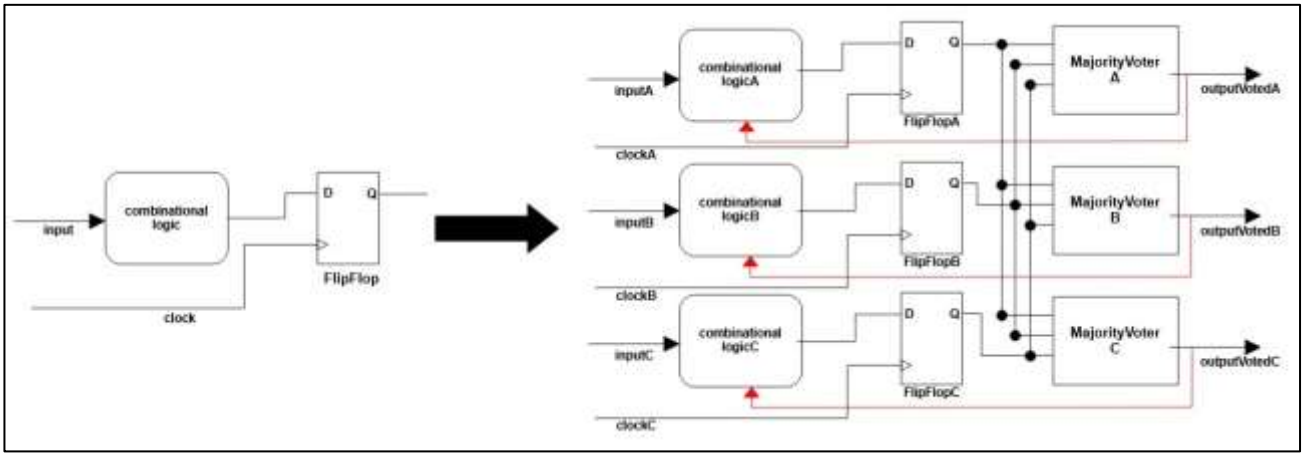

Abbildung 4.2: Darstellung einer vollständigen TMR-Implementierung mit drei MajorityVoter.

In dieser Arbeit wird für den Beschleuniger die vollständige Triplizierung angewendet, die Abbildung 4.2 dargestellt ist. Dies wird mit dem Triple Modular Redundancy Generator Toolset (TMRG) realisiert, das in Kapitel 4.1 näher vorgestellt wird.

4.1 Triple Modular Redundancy Generator Toolset (TMRG)

Der Triple Modular Redundancy Generator (TMRG) ist ein von CERN-Mitarbeitern entwickeltes Toolset, welches für die Triplizierung und Verifikation von Designs gegen Single Event Effekte verwendet wird. Das Toolset besteht aus vier verschiedenen Modulen: TMRG (Triple Module Redundancy Generator), SEEG (Single Event Effect Generator), PLAG (Placement Generator) und TBG (Testbench Generator). In diesem Kapitel wird auf die Quellen [12] und [13] verwiesen.

Mit dem TMRG-Modul ist es möglich, ein Design benutzerdefiniert zu triplizieren, um eine Immunität gegen SEUs (Single Event Upsets) und SETs (Single Event Transients) zu gewährleisten. Dazu werden Constraints bzw. Direktiven verwendet, wobei die Direktiven als Kommentare in den Quellcode geschrieben werden. Neben der Verwendung von Direktiven im Quellcode gibt es zwei weitere Möglichkeiten, Constraints zu verwenden.

Direktive im Quellcode (niedrige Priorität)	Konfigurationsdatei (mittlere Priorität)	Argumente in einer Befehlszeile (hohe Priorität)

`module counter(clk, rstn, out);`
`// tmrg default triplicate`
`input clk;`
`input rstn;`
`output [3:0] out;`
	`[counter]`
	`default = triplicate`
	`. . .`
		`$ tmrg counter.v -w "default triplicate counter"`
		`. . .`

Tabelle 4.1: Arten zu der Verwendung von Constraints im TMRG-Modul

Wie in Tabelle 4.1 gezeigt, kann eine Konfigurationsdatei erstellt und die Constraints dort eingetragen werden. Andererseits ist es auch möglich, die Constraints in der Befehlszeile mit der Option `-w` einzugeben. Dabei besitzt die Eingabe der Constraints in der Befehlzeile die höchste Priorität der vorhandenen drei Möglichkeiten, so dass diese vom TMRG-Modul zuerst ausgeführt werden.

Tabelle 4.2 listet neben `tmrg default triplicate` weitere Direktiven mit unterschiedlichen Funktionen auf.

Direktive	Funktion
`// tmrg default triplicate`	Tripliziert das Design vollständig
`// tmrg default do_not_triplicate`	Das Design wird nicht tripliziert
`// tmrg triplicate net_name`	Tripliziert das Signal net_name
`// tmrg do_not_triplicate net_name`	Das Signal net_name wird nicht tripliziert
`// tmrg translate on/off`	Der betroffene Bereich wird bei der Triplizierung ausgeblendet und für den triplizierten Design nicht berücksichtigt

Direktive	Funktion
`// tmrg majority_voter_cell name`	Legt den Namen der majority_voter_cell mit „name“ fest
`// tmrg fanout_cell name`	Legt den Namen der fanout_cell mit „name“ fest

Tabelle 4.2: Liste der Direktiven im TMRG-Modul

Zusätzlich erzeugt TMRG Voter und Fanouts (Listing 4.1), wenn an bestimmten Stellen des Designs triplizierte Signale mit nicht triplizierten Signalen verbunden werden müssen oder wenn das Design nicht vollständig tripliziert ist.

```
module majorityVoter #(
  parameter WIDTH = 1
)(
  input wire  [WIDTH-1:0] inA,
  input wire  [WIDTH-1:0] inB,
  input wire  [WIDTH-1:0] inC,
  output wire [WIDTH-1:0] out,
  output reg              tmrErr
);
  assign out = (inA&inB) | (inA&inC) |
(inB&inC);
  always @(inA or inB or inC) begin
    if (inA!=inB || inA!=inC || inB!=inC)
      tmrErr = 1;
    else
      tmrErr = 0;
  end
endmodule
```

```
module fanout #(
  parameter WIDTH = 1
)(
  input wire  [WIDTH-1:0] in,
  output wire [WIDTH-1:0] outA,
  output wire [WIDTH-1:0] outB,
  output wire [WIDTH-1:0] outC
);
  assign outA = in;
  assign outB = in;
  assign outC = in;
endmodule
```

Listing 4.1: Aufbau eines majorityVoters und eines Fanouts im TMRG-Modul

```
module counter(clk, rstn, out);
// tmrg default triplicate
input clk;
input rstn;
output [3:0] out;

always@(posedge clk) begin
    if (!rstn) begin
        out <= 0;
    end
    else begin
        out <= out + 1;
    end
end
endmodule
```

```
module counterTMR(clkA, clkB, clkC,
rstnA, rstnB, rstnC, outA, outB,
outC);
input clkA, clkB, clkC;
input rstnA, rstnB, rstnC;
output [3:0] outA;
output [3:0] outB;
output [3:0] outC;

always @( posedge clkA ) begin
    if (!rstnA) begin
        outA <= 0;
    end
    else begin
        outA <= outA+1;
    end
end

always @( posedge clkB ) begin
    . . .
end

always @( posedge clkC ) begin
    . . .
end
endmodule
```

Listing 4.2: Quellcode des normalen Zählers counter (links) und des triplizierten Zählers counterTMR (rechts)

Als Beispiel zeigt Listing 4.2 einen normalen Zähler, der vollständig tripliziert wurde. Die vollständig triplizierte Version des Zählers wurde vom TMRG-Modul mit dem Befehl `tmrg`

`filename.v` und der Anweisung `tmrg default triplicate` erzeugt und mit der Abkürzung TMR am Ende des Modulnamens gekennzeichnet.

Für den Schutz vor Single Event Effekten ist diese triplizierte Version jedoch nicht ausreichend und muss daher angepasst werden. Dabei werden Signale mit speziellen Namenskonventionen eingesetzt, um Voter für das triplizierte Signal zu generieren. Ein betroffenes Signal wird wie folgt gevotet:

```
reg [3:0]net;
wire [3:0]netVoted = net;
```

Es ist zu beachten, dass das gevotete Signal als wire-Typ deklariert und in der gleichen Zeile dem ursprünglichen Signal zum gevoteten Signal zugewiesen werden kann. In Listing 4.3 ist ein Beispiel dargestellt, bei dem das gevotete Signal countVoted im linken Quellcode eingeführt wird.

```
module counter(clk, rstn, out);
// tmrg default triplicate
input clk;
input rstn;
output [3:0] out;

reg [3:0]count;
wire [3:0]countVoted = count;

always@(posedge clk) begin
    if (!rstn) begin
        count <= 0;
    end
    else begin
        count <= countVoted + 1;
    end
end
assign out = countVoted;

endmodule
```

```
module counterTMR(clkA, clkB, clkC, rstnA,
rstnB, rstnC, outA, outB, outC);
wor countTmrErrorC, countTmrErrorB,
countTmrErrorA;
wire [3:0] countVotedC;
wire [3:0] countVotedB;
wire [3:0] countVotedA;
input clkA, clkB, clkC;
input rstnA, rstnB, rstnC;
output [3:0] outA;
output [3:0] outB;
output [3:0] outC;
reg  [3:0] countA ;
reg  [3:0] countB ;
reg  [3:0] countC ;

always @( posedge clkA ) begin
    if (!rstnA) begin
        countA <= 0;
    end
    else begin
        countA <= countVotedA+1;
    end
end

always @( posedge clkB ) begin
  . . .
end

always @( posedge clkC ) begin
  . . .
end
assign outA =  countVotedA;
assign outB =  countVotedB;
assign outC =  countVotedC;

majorityVoter #(.WIDTH(4)) countVoterA (
    .inA(countA),
    .inB(countB),
    .inC(countC),
    .out(countVotedA),
    .tmrErr(countTmrErrorA));

majorityVoter #(.WIDTH(4)) countVoterB (
    . . .
    );

majorityVoter #(.WIDTH(4)) countVoterC (
    . . .
    );
endmodule
```

Listing 4.3: Umwandlung des Moduls counter (links) durch Triplikation und Voting in das Modul counterTMR (rechts)

Hier ist es wichtig, dass das gevotete Signal immer auf der rechten Seite von Zuweisungen in getakteten Always-Blöcke und in Assign-Statements eingetragen wird, wenn im ursprünglichen Code die normale Version des Signals an der Stelle verwendet wurde. Außerdem muss in if- und case-Anweisungen anstelle der normalen Version immer das gevotete Signal verwendet werden. Mit der Triplikation des linken Quellcodes (Modul counter) werden neben der vollständigen Triplikation des Quellcodes (Modul counterTMR) majorityVoter für das Signal countVoted automatisch vom TMRG-Modul erzeugt. Dabei sind die Signale countA, countB und countC einmal mit jedem der drei majorityVoter verbunden. Dies verleiht dem Design eine hohe Sicherheit gegen Single Event Effekte, da bei Störung eines dieser Voter oder eines dieser Signale immer noch zwei Voter bzw. zwei Signale fehlerfrei sind.

In Abbildung 4.3 ist das in Listing 4.3 triplizierte Design counterTMR schematisch dargestellt.

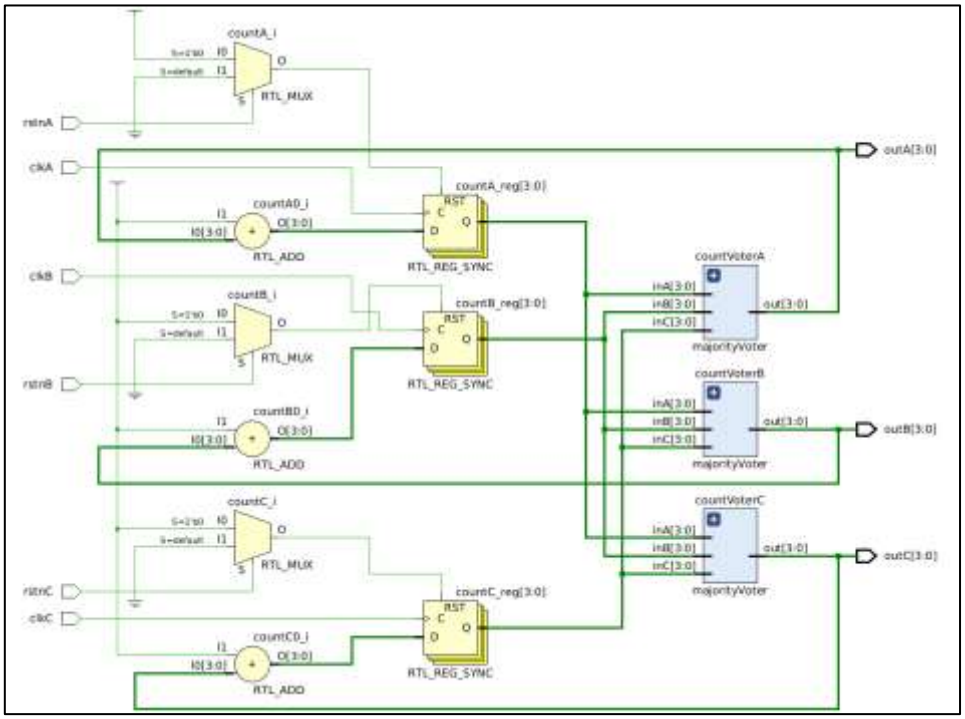

Abbildung 4.3: Schematische Darstellung des triplizierten Designs counterTMR

Nachdem das Design korrekt tripliziert wurde, stellt sich die Frage, ob das Design nach der Triplizierung die gleiche Funktion wie der ursprüngliche Entwurf besitzt und ob ein ausreichender Schutz gegen Single Event Effekte vorhanden ist. Zu diesem Zweck wird das SEEG-Modul verwendet, das eine Reihe von Verilog Tasks erzeugt, mit denen geeignete Netze für eine Simulation von SEUs und SETs forciert bzw. erzwungen und wieder freigegeben werden können.

Damit das SEEG-Modul diese Tasks erzeugen kann, wird eine synthetisierte Netzliste des triplizierten Designs und eine annotierte Bauteilbibliothek benötigt. Die synthetisierte Netzliste kann durch ein Synthesetool wie z.B. Xilinx Vivado erzeugt und exportiert werden, wobei es hierfür in Vivado zwei Möglichkeiten angeboten werden. Der Export kann entweder mit dem

tcl-Befehl write_verilog oder über das GUI-Fenster mit der Funktion „Export Netlist“ durchgeführt werden. Während des Synthesevorgangs wird durch Optimierungsalgorithmen die Anzahl der benötigten Gatter reduziert. Da durch die Triplizierung redundante Logik eingefügt wird, besteht die Gefahr, dass durch diesen Optimierungsvorgang die Redundanz entfernt wird. Um dies zu verhindern, wird eine SDC-Datei benötigt, die Constraints enthält, mit der die redundante Logik in einem Design während der Synthese von der Optimierung ausgenommen wird. Diese Datei wird mit dem triplizierten Design automatisch erzeugt oder kann mit dem Befehl `tmrg --sdc-generate filename.v` generiert werden. Listing 4.4 zeigt, wie eine SDC-Datei mit den entsprechenden Constraints aufgebaut ist.

```
set sdc_version 1.3

set tmrgSucces 0
set tmrgFailed 0
proc constrainNet netName {
  global tmrgSucces
  global tmrgFailed
  # find nets matching netName pattern
  set nets [dc::get_net $netName]
  if {[llength $nets] != 0} {
    set_dont_touch $nets
    incr tmrgSucces
  } else {
    puts "\[TMRG\] Warning! Net(s) '$netName' not found"
    incr tmrgFailed
  }
}

constrainNet /countA[*]
constrainNet /countB[*]
constrainNet /countC[*]
constrainNet /countVotedA[*]
constrainNet /countVotedB[*]
constrainNet /countVotedC[*]


    puts "TMRG successful  $tmrgSucces failed $tmrgFailed"
```

Listing 4.4: Aufbau einer SDC-Datei mit den entsprechenden Constraints für das Modul counterTMR

Bei der Synthese werden Konstanten in Attribute umgewandelt, Attribute für Place & Route erzeugt und ein Modul mit dem Namen „glbl“ am Ende des Quellcodes hinzugefügt. Diese Attribute und das Modul müssen für das SEEG-Modul auskommentiert werden, da das SEEG-Modul andernfalls Probleme mit dem Parsing der Netzliste hat und eine Fehlermeldung ausgibt. Abbildung 4.4 zeigt ein Beispiel für ein Parsingfehler.

```
p/caeco_tmr/xilinx_prims_tmrg.v -o counter_see.v counter_r2g.v
[ERROR ] Error in file 'counter_r2g.v' around line '14'.
[ERROR ]
[ERROR ] (* NotValidForBitStream *)
[ERROR ] ^
[ERROR ] Expected end of text, found '(' (at char 53), (line:14, col:1)
[ERROR ] Error during parsing
```

```
12  `timescale 1 ps / 1 ps
13
14  (* NotValidForBitStream *)
15  module counterTMR
16     (clkA,
17      clkB,
```

Abbildung 4.4: Auftreten eines Parsingfehlers bei der Ausführung des SEEG-Moduls, wobei sich der Fehler auf das Attribut in Zeile 14 bezieht.

Für die benötigte Bauteilbibliothek existiert ein Python-Skript, welches in der Lage ist, eine Bibliothek zu erzeugen, die durch das SEEG-Modul prozessiert werden kann. Der Befehl dafür lautet:

```
$ ./tmrg/tmrg/gen_tmrg_libs.py ./path/of/comp_library.v ./lib_tmrg.v
```

Mit diesem Befehl erzeugt das Python-Skript gen_tmrg_libs.py eine Version der Datei comp_library.v, mit dem Namen lib_tmrg.v, die mit dem SEEG-Tool verwendet werden kann. Das Python-Skript funktioniert jedoch im Zusammenspiel mit den Modellen der Xilinx-Primitiven nicht fehlerfrei, so dass stattdessen die Bauteile manuell eingetragen und annotiert werden müssen. Listing 4.5 zeigt einen Vergleich, wie das FlipFlop FDRE vor und nach der Umwandlung mit dem Python-Skript aussieht.

```
///// component FDRE ////
(* BOX_TYPE="PRIMITIVE" *) // Verilog-2001
module FDRE (Q, C, CE, D, R);
  parameter [0:0] INIT = 1'b0;
  parameter [0:0] IS_C_INVERTED = 1'b0;
  parameter [0:0] IS_D_INVERTED = 1'b0;
  parameter [0:0] IS_R_INVERTED = 1'b0;
   output Q;
   input C;
   input CE;
   input D;
   input R;
endmodule
```

```
module FDRE (
   output Q;
   input C;
   input CE;
   input D;
   input R;
endmodule
```

Listing 4.5: Fehlerhafte Konvertierung des FlipFlops FDRE mit dem Python-Skript gen_tmrg_libs.py

Bei der manuellen Erstellung der Bauteilbibliothek muss beachtet werden, dass alle Module vorhanden sein müssen, die auch in der synthetisierten Netzliste verwendet werden. Dabei müssen die FlipFlops mit dem unteren Befehl für eine Injektion von SEU-Effekten annotiert werden.

```
// tmrg seu_set net
```

Nachdem die synthetisierte Netzliste und die annotierte Bauteilbibliothek erstellt wurden, kann die Datei mit den Tasks für die Single Event Effekte mit dem SEEG-Modul durch Eingabe des folgenden Befehls erzeugt werden.

```
$ seeg -l ./xilinx_prims_tmrg.v -o ./counter_see.v ./counter_r2g.v
```

Mit der Option -l wird die gewünschte Bibliothek aufgerufen und mit der Option -o kann der Name und Pfad der erzeugten Datei angegeben werden. Wichtig ist auch, dass der Dateiname der synthetisierten Netzliste den Zusatz „r2g“ enthält. Ansonsten wird eine Fehlermeldung ausgegeben, dass der Zusatz „r2g“ nicht im Namen enthalten ist.

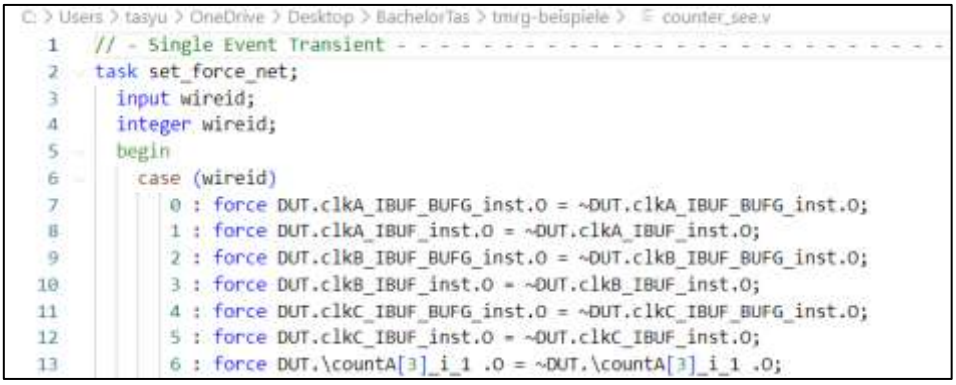

Abbildung 4.5: Übersicht des Tasks set_force_net, bei dem die gelisteten FlipFlops für eine SET-Simulation forciert werden

Abbildung 4.5 zeigt einen Task namens `set_force_net`, in dem die aufgelisteten Netze für eine SET-Simulation forciert, also umgeschaltet werden. Neben der Task `set_force_net` gibt es noch die Tasks `set_release_net`, `set_display_net` und `set_max_net`, wobei `set_release_net` die forcierten Netze mit dem Befehl release freigibt, `set_display_net` die aufgelisteten Netze im Debug-Fenster anzeigt und `set_max_net` die Gesamtanzahl der Netze dem Signal wireid zuordnet. Listing 4.6 zeigt die oben genannten Tasks in Kurzfassung.

```
// - Single Event Transient - - - - - - - - - - - - - - - - - - - - - - - -
task set_force_net;
  input wireid;
  integer wireid;
  begin
    case (wireid)
       0 : force DUT.clkA_IBUF_BUFG_inst.O = ~DUT.clkA_IBUF_BUFG_inst.O;
      . . .
    endcase
  end
endtask

task set_release_net;
  input wireid;
  integer wireid;
  begin
    case (wireid)
       0 : release DUT.clkA_IBUF_BUFG_inst.O;
       . . .
    endcase
  end
endtask

task set_display_net;
  input wireid;
  integer wireid;
  begin
    case (wireid)
      0 : $display("DUT.clkA_IBUF_BUFG_inst.O");
      . . .
    endcase
  end
endtask

task set_max_net;
  output wireid;
  integer wireid;
  begin
    wireid=44;
  end
endtask
```

Listing 4.6: Übersicht über die Tasks zur Simulation von Single Event Transients

Die Übersicht in Listing 4.6 gilt auch für Single Event Upsets, bei denen die Tasks statt `set_force_net` `seu_force_net` heißen und statt Netze FlipFlops geschaltet werden. Es gibt auch Tasks, die mit `see_` gekennzeichnet sind und bei denen beide Effekte (SEU und SET) forciert und freigegeben werden können. Nach den Tasks erzeugt das SEEG-Modul noch eine kommentierte Liste, in der die FlipFlops noch einmal mit einer zugehörigen Nummer aufgelistet sind.

Wenn ein tripliziertes Design in einer realen Umgebung implementiert werden soll, müssen einige Punkte beachtet werden. Einer dieser Punkte ist, dass die MajorityVoter das Routing der Instanzen der triplizierten FlipFlops kurz halten. Dadurch liegen die Instanzen sehr nahe beieinander, was dazu führen kann, dass ein Teilchen mehr als ein FlipFlop treffen kann und dadurch ein Fehler erzeugt wird, der durch TMR nicht korrigiert werden kann. Um dies zu verhindern, ermöglicht das PLAG-Modul die Einführung eines minimalen Abstands zwischen den Instanzen triplizierter FlipFlops, der vom Benutzer eingestellt werden kann. Um diese Datei zu erzeugen, wird in der Befehlszeile folgendes eingegeben:

```
$ plag -l ./xilinx_prims_tmrg.v -c ./plag.cfg -o ./counter_plag.tcl counter_r2g.v
```

Das PLAG-Modul erzeugt eine TCL-Datei, in welche die oben genannten Zuweisungen eingetragen werden. Dazu benötigt PLAG wie beim SEEG-Modul die annotierte Bauteilebibliothek, eine synthetisierte Netzliste und eine Konfigurationsdatei, in der die in der synthetisierten Netzliste instanziierten Bauteile eingetragen sind. PLAG sucht nach Instanzen der eingetragenen Bauteile und sortiert diese in drei Instanzgruppen tmrGroupA, tmrGroupB und tmrGroupC. Listing 4.7 zeigt die Struktur der von PLAG erzeugten TCL-Datei, in der die Instanzen der synthetisierten Netzliste (counter_r2g) in Instanzgruppen sortiert sind.

```
addInstToInstGroup tmrGroupA {counterTMR/\countA[3]_i_1 }
addInstToInstGroup tmrGroupA {counterTMR/\countA_reg[0] }
addInstToInstGroup tmrGroupA {counterTMR/\countA_reg[1] }
addInstToInstGroup tmrGroupA {counterTMR/\countA_reg[2] }
addInstToInstGroup tmrGroupA {counterTMR/\countA_reg[3] }
addInstToInstGroup tmrGroupB {counterTMR/\countB[3]_i_1 }
addInstToInstGroup tmrGroupB {counterTMR/\countB_reg[0] }
addInstToInstGroup tmrGroupB {counterTMR/\countB_reg[1] }
addInstToInstGroup tmrGroupB {counterTMR/\countB_reg[2] }
addInstToInstGroup tmrGroupB {counterTMR/\countB_reg[3] }
addInstToInstGroup tmrGroupC {counterTMR/\countC[0]_i_1 }
addInstToInstGroup tmrGroupC {counterTMR/\countC[1]_i_1 }
addInstToInstGroup tmrGroupC {counterTMR/\countC[2]_i_1 }
addInstToInstGroup tmrGroupC {counterTMR/\countC[3]_i_1 }
addInstToInstGroup tmrGroupC {counterTMR/\countC[3]_i_2 }
. . .
```

Listing 4.7: Struktur der TCL-Datei counter_plag.tcl, die vom Modul PLAG erzeugt wurde.

Neben der Triplizierung und des Votings mit TMRG, der Erstellung von Tasks zur Simulation von Single Event Effects mit SEEG und der Erstellung von Instanzgruppen für Placement & Routing mit PLAG ist das Toolset auch in der Lage, mit dem TBG-Modul Single Event Effekte mit Hilfe der Tasks des SEEG-Moduls zu simulieren, wobei eine generische Testbench mit

Hilfe des nicht-triplizierten Designs erzeugt wird. In diesem Fall wird in der Befehlszeile folgendes eingegeben:

```
$ tbg ./counter.v -o ./counter_tbg.v
```

Die erzeugte Testbench mit dem Namen counter_tbg.v enthält neben der Instanziierung des normalen Zählers counter auch die triplizierte Version. Zusätzlich ist es mit der Testbench möglich, die Single Event Effekte in einer Simulation durch SEE-Injektion darzustellen, die in Listing 4.8 zu sehen ist.

```
// - - - - - - - - - - - - Single Event Effect section - - - - - - - - - - - -
`ifdef SEE
  reg     SEEEnable=0;          // enables SEE generator
  reg     SEEActive=0;          // high during any SEE event
  integer SEEnextTime=0;        // time until the next SEE event
  integer SEEduration=0;        // duration of the next SEE event
  integer SEEwireId=0;          // wire to be affected by the next SEE event
  integer SEEmaxWireId=0;       // number of wires in the design which can be affected by
SEE event
  integer SEEmaxUpaseTime=1000; // 1 ns  (change if you are using different timescale)
  integer SEEDel=100_000;       // 100 ns (change if you are using different timescale)
  integer SEECounter=0;         // number of simulated SEE events

  `include "see.v"

  // get number of wires which can be affected by see
  initial
    see_max_net (SEEmaxWireId);
  always begin
      if (SEEEnable) begin
          // randomize time, duration, and wire of the next SEE
          SEEnextTime = #(SEEDel/2) {$random} % SEEDel;
          SEEduration = {$random} % (SEEmaxUpaseTime-1) + 1;  // SEE time is from 1 -
MAX_UPSET_TIME ns
          SEEwireId   = {$random} % SEEmaxWireId;

          // wait for SEE
          #(SEEnextTime);

          // SEE happens here! Toggle the selected wire.
          SEECounter=SEECounter+1;
          SEEActive=1;
          see_force_net(SEEwireId);
          see_display_net(SEEwireId); // probably you want to comment this line ?
          #(SEEduration);
          see_release_net(SEEwireId);
          SEEActive=0;
      end
      else
        #10;
    end
`endif
```

Listing 4.8: Die generierte Testbench verfügt über eine SEE-Injektion, in der die generierten Tasks mit der include-Direktive verwendet werden.

Bei der Erzeugung von Designs mit dem Toolset ist zu beachten, dass die Designs zuerst simuliert und verifiziert werden muss, bevor das Design für die Triplizierung verwendet werden kann. Der Grund hierfür ist, dass TMRG aus den fehlerhaften Designs triplizierte Versionen erzeugen kann und diese Fehler entsprechend ebenfalls tripliziert werden. Es ist möglich, dass TMRG diese Fehler beim Parsen meldet, aber die Meldungen sind meist undurchsichtig.

In dieser Arbeit wurde das TMRG-Toolset für die vollständige Triplizierung des CAECO-Beschleunigers verwendet, wobei die Logik vollständig tripliziert und alle FlipFlops (außer den Speicher-FlipFlops) gevotet wurden. Für die Umsetzung wurden die Module TMRG für die Triplikation und das Voting und SEEG für die Simulation von Single Event Effekten verwendet, während das Modul TBG für die Beschleunigerhärtung ausgelassen wurde. Außerdem ist das Tool PLAG auf die ASIC und nicht auf die FPGA Implementierung ausgelegt und wurde deswegen ebenfalls nicht angewendet. Für die Synthese des triplizierten Designs in Vivado konnte die vom TMRG-Modul generierte SDC-Datei (Listing 4.4) nicht verwendet werden, da es sich um eine Synopsys Design Constrains-Datei handelt, die nicht die für die Synthese in Vivado erforderlichen Attribute besitzt, so dass Vivado im Optimierungsvorgang die Redundanz wegoptimiert. Stattdessen werden einige Dateien des TMRG Config Generators verwendet, die in Kapitel 4.2 beschrieben sind.

4.2 TMRG Config Generator

Der TMRG Config Generator ist ein Tool, das den Prozess der Triplikation des TMRG-Moduls unterstützt und die korrekte Triplizierung des Designs überprüft. In dieser Arbeit wurden nur zwei Dateien verwendet. Dabei handelt es sich um das TCL-Skript reg2 und die Xilinx Design Constraints-Datei (XDC) tmr_synth [17].

Die Datei tmr_synth.xdc enthält Constraints, die sicherstellen, dass die Redundanz des triplizierten Designs während der Synthese mit Vivado erhalten bleibt. Listing 4.9 zeigt den Quellcode der Datei. Mit get_cells werden Zellen des Designs gelistet, bei denen es sich um Voter handelt. Mit dem Attribut KEEP_HIERARCHY werden diese Voter aus dem Design vor der Optimierung geschützt [18]. Als nächstes werden die Pins inA, inB und inC der Voter mit dem Befehl get_pins herausgefiltert. Abschließend werden mit get_nets die mit diesen Pins verbundenen Netze ermittelt, um sie vor der Optimierung mit KEEP zu schützen [18].

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

Listing 4.9: Der Quellcode der Datei tmr_synth.xdc.

Das TCL-Skript reg2.tcl überprüft, ob der Ausgang eines FlipFlops tatsächlich mit einem MajorityVoter verbunden ist. Dazu durchsucht das Skript alle verfügbaren FlipFlops in einem triplizierten Design und listet die untersuchten FlipFlops in zwei verschiedenen Textdateien mit den Namen „triplicated_rtl“ und „not_triplicated_rtl“ auf. Alle FlipFlops, deren Ausgang mit einem MajorityVoter verbunden sind, werden in der Datei triplicated_rtl.txt aufgelistet,

während die FlipFlops, deren Ausgang nicht mit einem MajorityVoter verbunden sind, in der Datei not_triplicated_rtl.txt aufgelistet werden. Es können auch synthetisierte oder implementierte Designs untersucht werden, so dass die erzeugten Textdateien triplicated_synth oder triplicated_impl anstelle von triplicated_rtl heißen. Da die Auflistung der FlipFlops in der triplizierten und der synthetisierten Version des Designs jedoch identisch ist, wird nur die normale bzw. die RTL-Version verwendet.

Zum Zeitpunkt dieser Arbeit wurde das reg2-Skript erweitert, da bei der Verifikation des triplizierten Beschleunigers im Vergleich des Original-Quellcodes Unterschiede auftraten, so dass die Endergebnisse der Simulation nicht übereinstimmten. Diese Unterschiede wurden durch FilpFlops verursacht, bei denen der Ausgang nicht nur mit MajorityVoter sondern auch mit weiteren Logikzellen verbunden waren. Dies ist auf Fehler bei der Einführung des Votings zurückzuführen und deutet darauf hin, dass im Code das ungevotete statt das gevotete Signal bei Zuweisungen, if-else Abfragen usw. verwendet wurden. Daher wurde reg2 so angepasst, dass derartige Fehler erkannt werden können. Der angepasste Quellcode wird in Kapitel 6.2 näher beschrieben.

5 Triplizierung des Beschleunigers

In diesem Kapitel wird die vollständige Triplizierung des CAECO-Beschleunigers beschrieben, wobei detailliert auf die Art der Triplizierung des Hauptmoduls und der Untermodule sowie auf die am Beschleuniger vorgenommenen Änderungen eingegangen wird. Dabei wird die Verilog-Version des Beschleunigers als Ausgangslage verwendet, welche im Rahmen der Betrieblichen Praxis aus VHDL übersetzt wurde. Die in diesem Kapitel verwendeten Listings und Informationen beruhen auf den Quellen [9] und [12].

In allen Modulen wurde die vollständige Triplizierung der Logik und der FlipFlops angewendet, wobei welche drei MajorityVoter für den zusätzlichen Schutz gegen SEE-Effekte vorgesehen werden.

```
reg [3:0]net;
wire [3:0]netVoted = net;
```

Die SRAM-Speicher sram_weight_storage_1024x8 und sram_data_storage_4096x32 sowie das Speicherarray `memory` im caeco_data_buffer_v werden nicht tripliziert, sondern sollen in einem Folgeprojekt zu Dual-Port-Speichern umgewandelt werden. Der zweite Port soll dann für ein zyklisches Daten-Scrubbing in Kombination mit einem Kanal-Code wie z.B. Hamming-Code verwendet werden.

Listing 5.1 zeigt Beispiele, bei dem normale Signale durch gevoteten Signalen ersetzt werden.

```
reg [3:0]net1;
reg [5:0]net2;
reg [3:0]net3;
. . .

if (!net1) begin
    net2 <= 0;
end
else begin
    net2 <= net2;
end

assign net1 = net3;
. . .
```

```
reg [3:0]net1;
reg [5:0]net2;
reg [3:0]net3;

wire [3:0]net1Voted = net1;
wire [5:0]net2Voted = net2;
wire [3:0]net3Voted = net3;
. . .

if (!net1Voted) begin
    net2 <= 0;
end
else begin
    net2 <= net2Voted;
end

assign net1 = net3Voted;
. . .
```

Listing 5.1: Voting-Beispiele, bei dem die normalen Register von den gevoteten Signalen ersetzt werden.

Der Quellcode des Beschleunigers wird so geändert, dass zwei Taktsignale clk und clkT eingeführt werden. Über den Input Port clk wird das Taktsignal eingekoppelt und dem Signal clkT zugewiesen. Das Signal clkT wird überall dort verwendet, wo der Entwurf tripliziert wurde, während das Signal clk im nicht-triplizierten Teil des Entwurfs zur Anwendung kommt. Das Signal clkT wird durch ein Fanout auf drei Signale aufgefaltet. Daraus folgt auch, dass die Untermodule zwei Eingänge für die Taktsignale clk und clkT erhalten, wobei die Module

caeco_data_buffer und caeco_data_storage beide Taktsignale benötigen. Listing 5.2 zeigt die Instanziierung des Moduls caeco_data_buffer_v mit beiden Taktsignalen.

```
423 caeco_data_buffer_v #(
424     .DATA_WIDTH(DATA_INTERNAL_WIDTH)
425 ) data_buffer (
426     .DIN(s_buffer_din),
427     .WE(s_buffer_we),
428     .DOUT(s_buffer_dout),
429     .RE(s_buffer_reVoted),
430     .LAYER_INDEX(s_layer_indexVoted),
431     .NEXT_SAMPLE(s_next_sampleVoted),
432     .NEXT_FILTER(s_next_filterVoted),
433     .CLK(clk),
434     .CLKT(clkT),
435     .RSTN(s_buffer_rstn)
436     );
```

Listing 5.2: Instanziierung des Untermoduls caeco_data_buffer_v im Hauptmodul caeco_v, mit zwei Taktsignalen.

Die Triplizierung des Beschleunigers erfolgt über eine Konfigurationsdatei mit dem Namen caeco.cfg, in der Optionen für das Hauptmodul und die Untermodule individuell eingestellt werden können, wie z.B. die Nicht-Triplizierung des Speichers oder einzelner Signale. Der Aufbau der Konfigurationsdatei ist in Listing 5.3 dargestellt. Dabei wird die triplizierte Version des Beschleunigers im Ordner rtl_verilog_tmr gespeichert, der im Repository neu eingefügt wird.

```
1  [tmrg]
2  tmr_dir = ./rtl_verilog_tmr
3  rtl_dir = ./rtl_verilog
4  sdc_generate = true
5  files = caeco_v.v caeco_pe_v.v caeco_data_storage_v.v caeco_weight_storage_v.v
   caeco_data_buffer_v.v sram_data_storage_4096x32_v.v
   sram_weight_storage_1024x8_v.v

7  [global]
8  #default = triplicate

10 [caeco_v]
11 default = triplicate
12 clk = do_not_triplicate
13 clkT = do_not_triplicate
14 #s_current_weights = do_not_triplicate
   . . . # other modules
42 [sram_data_storage_4096x32_v]
43 default = do_not_triplicate

45 [sram_weight_storage_1024x8_v]
46 default = do_not_triplicate
```

Listing 5.3: Struktur der Konfigurationsdatei caeco.cfg für die Triplizierung des Beschleunigers.

Nach der vollständigen Triplizierung muss das Design verifiziert werden, um sicherzustellen, dass das triplizierte Design die gleiche Funktionalität wie das Original besitzt. Hierfür gibt es zwei Verifizierungsmethoden, eine mit einer Testbench-Simulation in Questasim und eine mit dem reg2 TCL-Skript, die in Kapitel 6 näher beschrieben werden.

5.1 Triplizierung der Untermodule

5.1.1 caeco_pe_v

Das Prozesselement caeco_pe_v ist das einzige Modul des Beschleunigers, das mit dem GHDL-Yosys-Plugin über RTLIL-Datenstrukturen nach Verilog übersetzt wurde und für die Triplizierung verwendet wird. Zusätzlich zu den gevoteten Signalen enthält das Verilog-Modul eine generierte Funktion mit dem Namen s_4557, wobei die Eingänge der Funktion a, b und s nicht tripliziert werden. Listing 5.4 zeigt die Funktion mit ihrer Anwendung.

```
 92   function [1:0] s_4557 ;
 93     input [1:0] a;
 94     input [7:0] b;
 95     input [3:0] s;
 96     (* parallel_case *)
 97     casez (s)
 98       4'b???1:
 99         s_4557  = b[1:0];
100       4'b??1?:
101         s_4557  = b[3:2];
102       4'b?1??:
103         s_4557  = b[5:4];
104       4'b1???:
105         s_4557  = b[7:6];
106       default:
107         s_4557  = a;
108     endcase
109   endfunction
110   assign _04_ = s_4557(2'h0, 8'he4, { _03_, _02_, _01_, _00_ });
```

Listing 5.4: Die erzeugte Funktion s_4557 mit ihrer Anwendung in einem Assign-Statement

5.1.2 caeco_weight_storage_v

Das Modul caeco_weight_storage_v wurde auch vom GHDL-Yosys-Plugin automatisch ins Verilog übersetzt. Weil aber die Speicher zukünftig für das Daten-Scrubbing erweitert werden müssen, muss das Modul zur besseren Lesbarkeit ebenfalls manuell umgewandelt werden. Dabei wird die ASIC-Architektur verwendet, weil die SRAM-Speicher der Module caeco_weight_storage_v und caeco_data_storage_v in dieser Architektur instanziiert sind.

Bei der manuellen Übersetzung des Moduls war die ursprüngliche Idee, die VHDL-Version des SRAM-Gewichtsspeichers SRAM_WEIGHT_STORAGE_1024x8.vhd mit dem Modul zu verknüpfen. Damit die Gewichte in den Speicher gelangen, wurde im Modul caeco_weight_storage_v die Funktion `$readmemh()` verwendet, um die hexadezimal in den mem-Dateien hexadezimal dargestellten Gewichte einzulesen und zu initialisieren.

```
$readmemh("./caeco/sim/mem/weight_array_mem_file_0.mem", sram.ram);
```

Im obigen Quellcode stellt „sram.ram“ das Speicherarray `ram` des SRAM-Speichers dar, wobei der SRAM-Speicher als „sram“ im Modul instanziiert ist.

Bei der Simulation in Abbildung 5.1 wurde jedoch festgestellt, dass die Funktion `$readmemh()` die Gewichte nicht in den SRAM-Speicher einlesen konnte, so dass der Speicher RAM während der gesamten Simulation x-Werte anzeigte. Dies lag zum einen daran, dass es nicht möglich ist, die Gewichte mit einer Verilog-Funktion aus einem Verilog-Modul in einen VHDL-Array zu laden und zum anderen daran, dass die Funktion `$readmemh()` direkt in den

Speicher integriert werden muss, so dass die Verilog-Version des SRAM-Speichers verwendet werden musste, wie sie in Listing 5.5 dargestellt ist.

Abbildung 5.1: Vergleich zweier RAM-Speicher der Originalen (dut_vhdl) und der Verilog-Version (dut_vlog) im Waveform-Fenster.

Die Listings 5.5 und 5.6 stellen eine neue Lösung für die Initialisierung der Gewichte dar. Beginnend in Zeile 2 wird ein Parameter mit dem Namen INIT_FILE deklariert, der in Zeile 21 als String-Parameter verwendet wird. Der String enthält den Dateipfad der mem-Datei. Da es aber fünf verschiedene mem-Dateien gibt, werden diese in eine generate if-Abfrage eingebunden, die in Listing 5.6 dargestellt ist. In der if-Abfrage wird der Wert des Parameters `WEIGHT_ID` überprüft. Im Fall `WEIGHT_ID == 0` erhält `INIT_FILE` den String „./caeco/sim/mem/weight_array_mem_file_0.mem“, so dass der Funktion `$readmemh()` dieser String als Dateipfad übergeben wird und die Gewichte der mem-Datei in den RAM-Speicher eingelesen werden können.

Abgesehen von der Initialisierung der Gewichte besitzt das Modul keine Register, die gevotet werden müssen, so dass nur das Taktsignal clk benötigt wird.

```
1  module sram_weight_storage_1024x8_v #(
2  parameter INIT_FILE = ""
3  )(clk, cen, rdwen, deepsleep, powergate, a, d, bw, q);

5  parameter WORD_SIZE = 1024;
6  parameter DATA_WIDTH = 8;
7  parameter ADDR_WIDTH = 10;

9  input clk;          // clock input
10 input cen;          // chip enable
11 input rdwen;            // read/write enable
12 input deepsleep;
13 input powergate;
14 input [ADDR_WIDTH-1:0]a;        // adress input
15 input [DATA_WIDTH-1:0]d;        // data input
16 input [DATA_WIDTH-1:0]bw;
17 output reg [DATA_WIDTH-1:0]q;       // data output

19 reg [DATA_WIDTH-1:0] ram[0:WORD_SIZE-1];
21 initial $readmemh(INIT_FILE, ram);

23 always @(posedge clk) begin
24     if (!cen)       // chip enable active low
25     begin
26         if (!rdwen) begin   // active low for write
27             ram[a] <= d;
28         end
29         else begin
30             q <= ram[a];
31         end
32     end
33 end
35 endmodule
```

Listing 5.5: Die Verilog-Version des SRAM-Speichers sram_weight_storage_1024x8_v.

```
22  generate
23      if (WEIGHT_ID == 0) begin
24          sram_weight_storage_1024x8_v #(
25              .INIT_FILE("/user/ytas/Desktop/caeco/sim/mem/weight_array_mem_file_0.mem")
26          ) sram (
27              .clk(clk),
28              .cen(s_cen),
29              .rdwen(s_rdwen),
30              .deepsleep(1'b0),
31              .powergate(1'b0),
32              .a(s_addr),
33              .d(w_in),
34              .bw(8'b11111111),
35              .q(w_out)
36          );
37      end
38      if (WEIGHT_ID == 1) begin
            . . .
52      end
53      if (WEIGHT_ID == 2) begin
            . . .
67      end
68      if (WEIGHT_ID == 3) begin
            . . .
82      end
83      if (WEIGHT_ID == 4) begin
            . . .
97      end
98  endgenerate
```

Listing 5.6: Ein generate if-konstrukt im Modul caeco_weight_storage_v, das benötigt wird, um die Gewichte zu initialisieren.

5.1.3 caeco_data_storage_v

Wie beim Modul caeco_weight_storage_v muss die ASIC-Architektur des Moduls caeco_data_storage_v für das Daten-Scrubbing und die Instanziierung des SRAM-Speichers sram_data_storage_4096x32_v manuell nach Verilog übersetzt werden. Bei der Triplizierung des Moduls werden nur die Register `s_dout_addr_low_bits_z1` und `s_dout_addr_high_bits_z1` gevotet, die im Quellcode als FilpFlops dargestellt sind, wie in Listing 5.7 zu sehen ist.

```
27  wire [1:0]s_dout_addr_low_bits;
28  reg [1:0]s_dout_addr_low_bits_z1;
29  wire [1:0]s_dout_addr_high_bits;
30  reg [1:0]s_dout_addr_high_bits_z1;
    . . .
36  // Signals for Voting
37  wire [1:0]s_dout_addr_low_bits_z1Voted = s_dout_addr_low_bits_z1;
38  wire [1:0]s_dout_addr_high_bits_z1Voted = s_dout_addr_high_bits_z1;
    . . .
94  always @(posedge clkT, negedge rstn) begin
95      if(rstn == 0) begin
96          s_dout_addr_low_bits_z1 <= 0;
97          s_dout_addr_high_bits_z1 <= 0;
98      end
99      else begin
100         s_dout_addr_low_bits_z1 <= s_dout_addr_low_bits;
101         s_dout_addr_high_bits_z1 <= s_dout_addr_high_bits;
102     end
103 end
```

Listing 5.7: Das Voting der Register s_dout_addr_high_bits_z1 und s_dout_addr_low_bits_z1 im Modul caeco_data_storage_v.

Listing 5.8 zeigt die erzeugten MajorityVoter für die gevoteten Signale `s_dout_addr_low_bits_z1Voted` und `s_dout_addr_high_bits_z1Voted`.

```
635 majorityVoter #(.WIDTH(2)) s_dout_addr_high_bits_z1VoterA (
636     .inA(s_dout_addr_high_bits_z1A),
637     .inB(s_dout_addr_high_bits_z1B),
638     .inC(s_dout_addr_high_bits_z1C),
639     .out(s_dout_addr_high_bits_z1VotedA),
640     .tmrErr(s_dout_addr_high_bits_z1TmrErrorA)
641     );

643 majorityVoter #(.WIDTH(2)) s_dout_addr_low_bits_z1VoterA (
644     .inA(s_dout_addr_low_bits_z1A),
645     .inB(s_dout_addr_low_bits_z1B),
646     .inC(s_dout_addr_low_bits_z1C),
647     .out(s_dout_addr_low_bits_z1VotedA),
648     .tmrErr(s_dout_addr_low_bits_z1TmrErrorA)
649     );

651 majorityVoter #(.WIDTH(2)) s_dout_addr_high_bits_z1VoterB (
        . . .
657     );

659 majorityVoter #(.WIDTH(2)) s_dout_addr_low_bits_z1VoterB (
        . . .
666     );

668 majorityVoter #(.WIDTH(2)) s_dout_addr_high_bits_z1VoterC (
        . . .
675     );

677 majorityVoter #(.WIDTH(2)) s_dout_addr_low_bits_z1VoterC (
        . . .
684     );
```

Listing 5.8: Erzeugte MajorityVoter für die Signale s_dout_addr_low_bits_z1Voted und s_dout_addr_high_bits_z1Voted.

5.1.4 caeco_data_buffer_v

Für das Modul caeco_data_buffer_v müssen vier Register tripliziert und gevotet werden: `read_counter`, `read_counter_buffer`, `write_counter` und `s_dout`. Das Speicherarray `memory` soll jedoch nicht tripliziert werden, befindet sich aber mit dem Register `write_counter` im selben Always-Block. Aus diesem Grund muss ein zweiter Always-Block eingeführt werden. Listing 5.9 stellt die Separation der Register `memory` und des Signals `write_counter` bzw. `write_counterVoted` in zwei Always-Blöcken dar.

```
50 always @(posedge CLK, negedge RSTN) begin
51     if (!RSTN) begin
52         for (i = 0; i <= MAX_MEM_DEPTHS-1; i = i + 1) begin
53             memory[i] <= 0;
54         end
55     end
56     else begin
57         if (WE) begin
58             memory[write_counterVoted] <= DIN;
59         end
60     end
61 end
62
63 always @(posedge CLKT, negedge RSTN) begin
64     if (!RSTN) begin
65         write_counter <= 0;
66     end
67     else begin
68         write_counter <= write_counterVoted;
69         if (WE) begin
70             write_counter <= write_counterVoted + 1;
71         end
72     end
73 end
```

Listing 5.9: Separation der Register memory und write_counter in zwei Always-Blöcke für die Triplizierung und Voting des caeco_data_buffer_v

5.2 Triplizierung des Hauptmoduls

Im Hauptmodul caeco_v werden bis zu 54 Register gevotet, die als FlipFlops fungieren. Unter diesen Registern ist ein Register namens `s_results_buffer` als Bit-Array deklariert, das im Gegensatz zum Speicherarray `memory` im caeco_data_buffer_v gevotet werden muss. Da aber die direkte Zuweisung eines Speicherarrays bei der Deklaration zu einem Syntaxfehler führt, muss überlegt werden, wie ein Speicherarray so gevotet werden kann, dass das TMRG-Modul tatsächlich MajorityVoter für dieses Array generiert.

Tabelle 5.1 zeigt Fallunterscheidungen verschiedener Deklarationen. Es ist zu sehen, dass das TMRG-Modul nur MajorityVoter erzeugt, wenn das gevotete Signal `netVoted` als wire-Typ deklariert und bei der Deklaration das normale Register `net` zugewiesen wird.

Deklaration	Ergebnis des TMRG-Moduls
`reg [3:0]net;` `wire [3:0]netVoted = net;`	Die Triplizierung wird fehlerfrei ausgeführt, bei dem drei majorityVoter erzeugt werden.
`reg [3:0]net;` `reg [3:0]netVoted = net;`	Es wird ein Parsingfehler ausgegeben.

`reg [3:0]net;`
`wire/reg [3:0]netVoted;`
`assign netVoted = net;`
	Die Triplizierung wird fehlerfrei durchgeführt, aber es werden keine MajorityVoter erzeugt.

`reg [3:0]net[0:N-1];` `wire/reg [3:0]netVoted[0:N-1] = net;`	Es wird ein Parsingfehler ausgegeben.

`reg [3:0]net[0:N-1];`
`generate`
`  for (i = 0; i <= N-1; i = i + 1)`
`  begin`
`  wire [3:0]netVoted = net[i];`
`  end`
`endgenerate`
	Die Triplizierung wird fehlerfrei durchgeführt, aber es werden keine MajorityVoter erzeugt.

Tabelle 5.1: Übersicht verschiedener Deklarationen von gevoteten Signalen.

Eine andere Möglichkeit, MajorityVoter für diese Array-Konstrukte zu erzeugen, besteht darin, zwei Signale für das normale und das gevotete Bit-Array zu deklarieren, wie in Listing 5.10 gezeigt. Hier werden für `s_results_buffer` und `s_results_bufferVoted` zwei Signale mit den Namen `s_results_buff` und `s_results_buffVoted` in einer generate for-Schleife deklariert. Dabei wird der Index von `s_results_buffer` `s_results_buff` zugewiesen, während `s_results_buffVoted` die Werte von `s_results_buff` erhält. Die for-Schleife endet dann mit einer Assign-Zuweisung, die die Werte von `s_results_buff` dem Index des Bit-Arrays `s_results_bufferVoted` zuweist.

```
reg [PE_RESULT_WIDTH-1:0]s_results_buffer[0:NUM_PES-1];
wire [PE_RESULT_WIDTH-1:0]s_results_bufferVoted[0:NUM_PES-1];

genvar i;
generate
    for (i = 0; i <= NUM_PES-1; i = i + 1) begin
        wire [PE_RESULT_WIDTH-1:0]s_results_buff = s_results_buffer[i];
        wire [PE_RESULT_WIDTH-1:0]s_results_buffVoted = s_results_buff;

        assign s_results_bufferVoted[i] = s_results_buffVoted;
    end
endgenerate
```

Listing 5.10: Voting des Bit-Arrays s_results_buffer nach s_results_bufferVoted mit einer generate for-Schleife.

Hier ist die Idee, die MajorityVoter durch Indizierung der Signale innerhalb der generate for-Schleife zu generieren. Aber auch diesem Ansatz entsteht das Problem, dass das TMRG-Modul die MajorityVoter außerhalb der generate for-Schleife generiert, so dass der generierte Quellcode nicht funktioniert. Diese falsche Voter-Instanziierung ist den TMRG-Entwicklern bekannt und in der Dokumentation ist der Hinweis enthalten, dass die Architektur des TMRG-Moduls derzeit nicht in der Lage ist, diese Art von Triplizierung durchzuführen [12]. Daher wurde die generate for-Schleife in triplizierter Form in eine Verilog-Header-Datei (caeco_pack_tmr.vh) geschrieben, wobei der MajorityVoter mit der obigen generate for-Schleife erzeugt und dann extrahiert wird, um ihn in die neue Schleife einzufügen. Dabei wird die obige generate for-Schleife mit den tmrg translate on/off-Flags von der Triplizierung ausgenommen, so dass die Header-Datei caeco_pack_tmr.vh nach der Triplizierung mit der include-Anweisung hinzugefügt werden muss. Listing 5.11 zeigt den triplizierten Entwurf aus Listing 5.10 mit drei MajorityVoter.

```
 2  generate
 3    for(i = 0; i <= NUM_PES-1; i = i + 1)
 4      begin
 5        wire [PE_RESULT_WIDTH-1:0] s_results_buffA =  s_results_bufferA[i] ;
 6        wire [PE_RESULT_WIDTH-1:0] s_results_buffB =  s_results_bufferB[i] ;
 7        wire [PE_RESULT_WIDTH-1:0] s_results_buffC =  s_results_bufferC[i] ;
 8        wor s_results_buffTmrErrorA;
 9        wire [PE_RESULT_WIDTH-1:0] s_results_buffVotedA;
10        wor s_results_buffTmrErrorB;
11        wire [PE_RESULT_WIDTH-1:0] s_results_buffVotedB;
12        wor s_results_buffTmrErrorC;
13        wire [PE_RESULT_WIDTH-1:0] s_results_buffVotedC;
14        assign s_results_bufferVotedA[i]  =  s_results_buffVotedA;
15        assign s_results_bufferVotedB[i]  =  s_results_buffVotedB;
16        assign s_results_bufferVotedC[i]  =  s_results_buffVotedC;

18      majorityVoter #(.WIDTH(((PE_RESULT_WIDTH-1)>(0)) ? ((PE_RESULT_WIDTH-1)-
    (0)+1) : ((0)-(PE_RESULT_WIDTH-1)+1))) s_results_buffVoterA (
19      .inA(s_results_buffA),
20      .inB(s_results_buffB),
21      .inC(s_results_buffC),
22      .out(s_results_buffVotedA),
23      .tmrErr(s_results_buffTmrErrorA)
24      );
    . . .
34      majorityVoter #(.WIDTH(((PE_RESULT_WIDTH-1)>(0)) ? ((PE_RESULT_WIDTH-1)-
    (0)+1) : ((0)-(PE_RESULT_WIDTH-1)+1))) s_results_buffVoterC (
        . . .
40      );
41      end

43  endgenerate
```

Listing 5.11: Der triplizierte Entwurf der generate for-Schleife in Listing 5.10 in der Haederdatei caeco_pack_tmr.vh.

Wie am Ende des Kapitels 3.2.1 beschrieben, enthält das Hauptmodul den Initial-Block logging_relu und den Generate-Block generate_results_log, in denen die Daten in Textdateien gespeichert werden. Die Architektur dieser Blöcke ist nicht synthetisierbar bzw. enthält nicht synthetisierbaren Quellcode, so dass die Blöcke mit den Flags tmrg translate on/off von der Triplizierung ausgenommen werden. Ein anderer Grund für das Ausblenden der genannten Blöcke ist, dass beim Triplizierungsvorgang bis zu vier Integer-Signale als unbekannte Signale nicht berücksichtigt werden und die Blöcke generell die Berechnungen nicht beeinflussen. Abbildung 5.2 zeigt Warnmeldungen, welche die integer-Signale beim Triplizierungsvorgang verursachen, während in Listing 5.12 die Initialblocks logging_relu mit den entsprechenden Flags ausgeblendet wird.

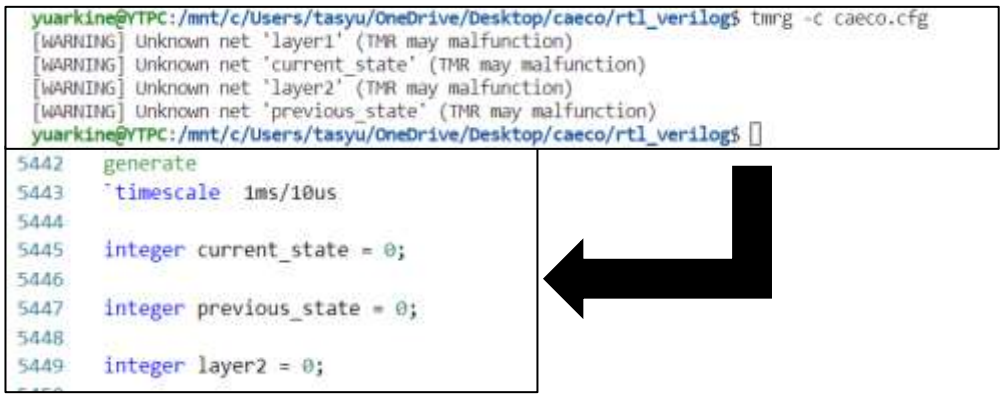

Abbildung 5.2: Warnmeldungen der Integer-Signale, welche als Unbekannte Signale nicht im Triplizierungsvorgang berücksichtigt werden.

```
1618 // tmrg translate off

1620 initial begin // logging_relu
1621         activation_out = $fopen("Act_Out.txt", "w");
1622         if (clk) begin
1623             if (layer1 != s_layer_index) begin
1624                 $fwrite(activation_out, "Layer: %d", s_layer_index);
1625                 layer1 <= s_layer_index;
1626             end
1627             if (1 == s_storage_we) begin
1628                 //$fdisplay(activation_out, "Value: %d", s_activation);
1629                 $fwrite(activation_out, "%d", s_activation, ", 0x%h", s_relu);

1631             end
1632         end
1633         $fclose(activation_out);
1634 end
1635 // tmrg translate on
```

Listing 5.12: Ausblenden des nicht-synthetisierbaren initial-Blocks logging_relu mit tmrg translate on/off-Flags.

6 Verifikation des triplizierten Designs

Für die Verifizierung des triplizierten Beschleunigers stehen zwei verschiedene Methoden zur Verfügung. Einerseits kann das Design dynamisch mit der SystemVerilog-Testbench caeco_tb_ecg_file_tmr simuliert werden, wobei die triplizierten bzw. gevoteten Signale mit dem Original verglichen und auf Unterschiede untersucht werden können. Zum anderen kann das Design statisch mit dem TCL-Skript reg2 dahingehend untersucht werden, ob die Ausgänge aller FlipFlops nur mit jeweils drei MajorityVotern verbunden sind.

Die Verifikation wurde mit beiden Methoden durchgeführt und die Ergebnisse werden in diesem Kapitel ausführlich beschrieben.

6.1 Verifizierung mit dem Testbench caeco_tb_ecg_file_tmr

Die SystemVerilog-Testbench caeco_tb_ecg_file_tmr stellt eine Version der Testbench caeco_tb_ecg_file dar, in der die tmr-Version neben den Instanzen dut_vhdl für die Originalversion und dut_vlog für die Verilog-Version auch eine Instanz mit dem Namen dut_vlog_tmr enthält, in der die triplizierte Verilog-Version des Beschleunigers instanziiert ist. Listing 5.13 zeigt das instanziierte Hauptmodul caeco_vTMR als dut_vlog_tmr.

```
114     generate if (RunVlogParallel == 1) begin : generate_vlog_tmr
115         logic result_valid_vlogA, result_valid_vlogB, result_valid_vlogC,
            din_ready_vlogA, din_ready_vlogB, din_ready_vlogC;

117         caeco_vTMR dut_vlog_tmr (
118             .clk                (clk),
119             .rstnA               (rstn),
120             .rstnB               (rstn),
121             .rstnC               (rstn),
122             .enA                 (1'b1),
123             .enB                 (1'b1),
124             .enC                 (1'b1),
126             .cmdA                (cmd),
127             .cmdB                (cmd),
128             .cmdC                (cmd),
130             .dinA                (din),
131             .dinB                (din),
132             .dinC                (din),
133             .din_validA          (din_valid),
134             .din_validB          (din_valid),
135             .din_validC          (din_valid),
136             .din_readyA          (din_ready_vlogA),
137             .din_readyB          (din_ready_vlogB),
138             .din_readyC          (din_ready_vlogC),
139             .din_lastA           (din_last),
140             .din_lastB           (din_last),
141             .din_lastC           (din_last),
143             .result_validA       (result_valid_vlogA),
144             .result_validB       (result_valid_vlogB),
145             .result_validC       (result_valid_vlogC)
146         );
147     end
148     endgenerate
```

Listing 6.1: Das instanziierte triplizierte Modul caeco_vTMR mit dem Namen "dut_vlog_tmr" in der Testbench.

Damit für eine Verifizierung die Endergebnisse der triplizierten Version in der Konsole erscheinen, werden die gevoteten Signale s_result_first_classVotedA, -B und -C und

s_result_second_classVotedA, -B und -C in einer `$write()`-Funktion, wie in Listing 6.2 dargestellt, eingetragen. Im Listing ist zu sehen, dass für die Ergebnisse der Verilog-Version die gevoteten Signale s_result_first_class und s_result_second_class verwendet werden.

```
$write("Got VHDL results:\t %d, %d\n", $signed(dut_vhdl.s_result_first_class),
$signed(dut_vhdl.s_result_second_class));

if ((RunVlogParallel == 1)) begin
      $write("Got Verilog results:\t %d, %d\n",
      $signed(generate_vlog.dut_vlog.s_result_first_classVoted),
      $signed(generate_vlog.dut_vlog.s_result_second_classVoted));
end

if ((RunVlogParallel == 1)) begin
      $write("Got triplicated Verilog results: %d, %d,\n\t\t\t\t %d, %d,\n\t\t\t\t
%d, %d\n",
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_first_classVotedA),
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_second_classVotedA),
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_first_classVotedB),
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_second_classVotedB),
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_first_classVotedC),
      $signed(generate_vlog_tmr.dut_vlog_tmr.s_result_second_classVotedC));
end
```

Listing 6.2: Ausgabe der Endergebnisse der drei Versionen mit einer $write()-Funktion.

Zu diesem Zweck wurde auch eine tmr-Version des Shell-Skripts simulate_caeco_tmr.sh erstellt, in der die triplizierten Module des Beschleunigers für die Simulation mit der Testbench hinzugefügt werden. Dabei wird der Pfad des Ordners, der die triplizierten Module enthält, als RTLD_VLOG_TMR benannt. Listing 6.3 zeigt eine Übersicht der Dateien im Quellcode des Shell-Skripts.

```
RTLD_VHDL=$(pwd)/../rtl/
RTLD_VLOG=$(pwd)/../rtl_verilog/
RTLD_VLOG_TMR=$(pwd)/../rtl_verilog_tmr/


rm -rf ./.work/*
mkdir -p .work/results

files_vh="\

    # files from the original VHDL version of CAECO
. . .
"
files_v="\
    $RTLD_VLOG/sram_data_storage_4096x32_v.v \
    $RTLD_VLOG/sram_weight_storage_1024x8_v.v \
    $RTLD_VLOG/caeco_weight_storage_v.v \
    $RTLD_VLOG/caeco_data_storage_v.v \
    $RTLD_VLOG/caeco_data_buffer_v.v \
    $RTLD_VLOG/caeco_pe_v.v \
    $RTLD_VLOG/caeco_v.v \
    $RTLD_VLOG_TMR/sram_data_storage_4096x32_vTMR.v \
    $RTLD_VLOG_TMR/sram_weight_storage_1024x8_vTMR.v \
    $RTLD_VLOG_TMR/caeco_data_storage_vTMR.v \
    $RTLD_VLOG_TMR/caeco_weight_storage_vTMR.v \
    $RTLD_VLOG_TMR/caeco_data_buffer_vTMR.v \
    $RTLD_VLOG_TMR/caeco_pe_vTMR.v \
    $RTLD_VLOG_TMR/caeco_vTMR.v \
"

files_sv="\
    $(pwd)/caeco_tb_ecg_file_tmr.sv \
"
```

Listing 6.3: Übersicht der aufgelisteten Dateien im Shell-Skript simulate_caeco_tmr.sh.

Unter Verwendung der Testbench caeco_tb_ecg_tmr und des Shell-Skripts caeco_simulate_tmr wird die triplizierte Version des Beschleunigers verifiziert, indem in der Simulation die Signale der Verilog-Version und der Original VHDL-Version vergleichen werden und eventuelle Unterschiede anhand von Zeitangaben ausgegeben werden. Abbildung 6.1 zeigt eine Waveform, bei der die Signale dnn_stateA, -B, und -C und dnn_stateVotedA, -B, -C der triplizierten Version im Vergleich zu den anderen Versionen des Beschleunigers auf mögliche Unterschiede untersucht werden, während Abbildung 6.2 als Beispiel die Endergebnisse des Vergleichs in der Konsole zeigt.

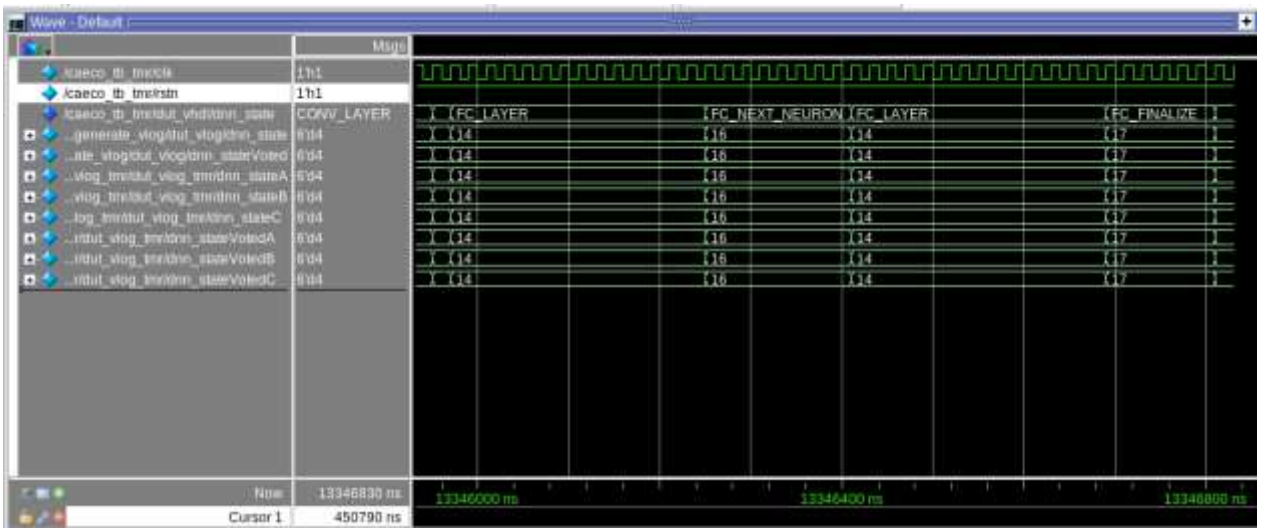

Abbildung 6.1: Das Signal dnn_state wird in einer Waveform dargestellt, die das Signal in allen drei Beschleunigervarianten zeigt.

```
# Got VHDL results:    3259,  -1782
# Got Verilog results:    3259,  -1782
# Got triplicated Verilog results:   3259,  -1782,
#                                    3259,  -1782,
#                                    3259,  -1782
# Result is correct on VHDL
# Result is correct on Verilog
# Result is correct on triplicated Verilog
# ** Note: $stop    : /user/ytas/Desktop/caeco/sim/caeco_tb_ecg_file_tmr.sv(368)
#    Time: 13346830 ns  Iteration: 1  Instance: /caeco_tb_tmr
# Break in Module caeco_tb_tmr at /user/ytas/Desktop/caeco/sim/caeco_tb_ecg_file_tmr.sv line 368
```

Abbildung 6.2: Die identischen Ergebnisse der drei Beschleunigerversionen in der Konsole.

Die Endergebnisse zeigen, dass alle drei CAECO-Versionen den gleichen Wert generieren. Dies ist zunächst eine Bestätigung dafür, dass durch die Triplizierung keine funktionalen Fehler eingeführt worden sind.

Um zusätzlich sicherzustellen, dass alle benötigten Signale gevotet worden sind, kann ein TCL-Skript mit dem Namen „reg2“ des TMRG Config Generators verwendet werden, das im nächsten Kapitel beschrieben wird

6.2 Verifizierung mit dem Skript reg2.tcl

Um die Triplizierung des Entwurfs mit Hilfe des reg2 TCL-Skript zu verifizieren, muss in Vivado ein Projekt angelegt werden, in dem das triplizierte Design importiert wird. Abbildung 6.3 zeigt die Projektübersicht.

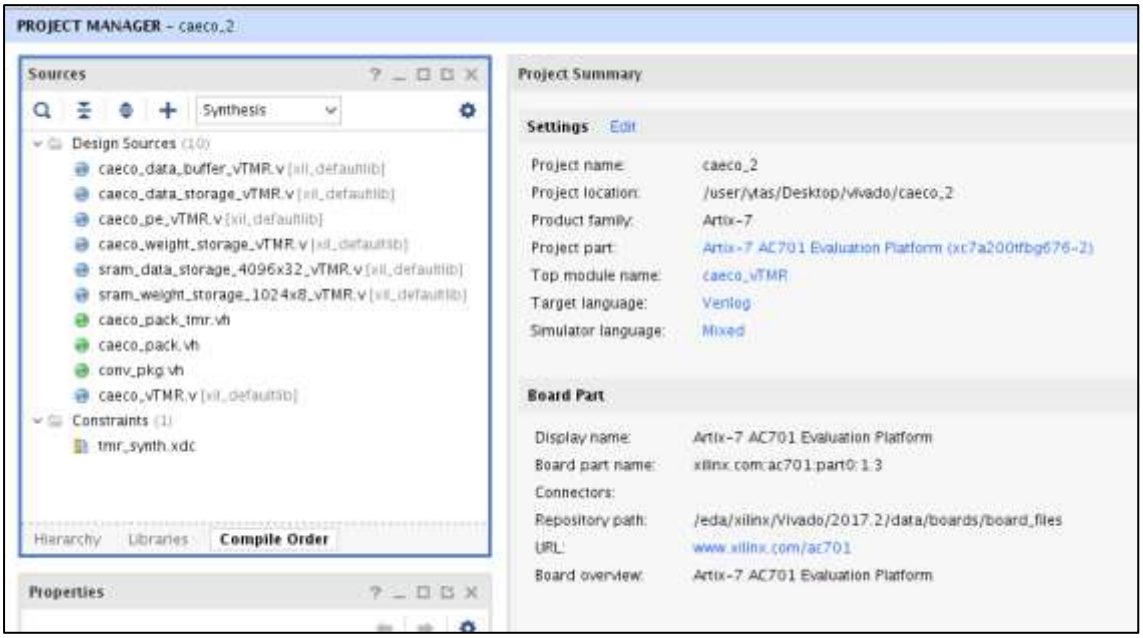

Abbildung 6.3: Die Übersicht des angelegten Vivado-Projekts.

Mit dem angelegten Projekt kann das reg2-Skript durch den Befehl `source` in der TCL-Konsole ausgeführt werden.

```
source ./tmrg_cfg_gen-main/reg2/reg2.tcl
```

```
61 #Opens Elaborated Design
62 synth_design -rtl
63 #Get all leafcells that are primitive and belong to RTL_REGISTER group
64 set leafcells [lsort -unique -decreasing -dictionary [get_cells -hierarchical
   -filter {IS_PRIMITIVE == true && PRIMITIVE_GROUP == RTL_REGISTER}]]
65 get_triplicated_and_not_triplicated $leafcells "./triplicated_rtl.txt"
   "./not_triplicated_rtl.txt" "./triplication_faults_rtl.txt"
66 close_design
```

Listing 6.4: Öffnen des elaborierten Designs und Abrufen aller primitiven FlipFlops aus der Gruppe RTL_REGISTER.

Bei der Ausführung öffnet das reg2-Skript zunächst das elaborierte Design, wie in Listing 6.4 gezeigt. In Zeile 64 werden alle Zellen (Leafcells) des triplizierten Designs gelistet, die der Gruppe RTL_REGISTER angehören. Diese primitiven FlipFlops werden dann in Zeile 65 durch den Prozess get_triplicated_and_not_triplicated untersucht, dessen Quellcode in Listing 6.5 dargestellt ist. Die untersuchten FlipFlops werden anschließend in drei verschiedenen Textdateien (triplicated_rtl, not_triplicated_rtl und triplication_faults) aufgelistet.

```
 4  set countVotingcells 3

 6  proc get_triplicated_and_not_triplicated {leafcells triplicated not_triplicated
    triplication_faults} {

 8      global countVotingcells

10      #Creates and opens files in working directory
11      #Truncate to zero if it exists. If it does not exist, create a new file.
12      set triplicated_log [open $triplicated w]
13      set not_triplicated_log [open $not_triplicated w]
14      set triplication_faults [open $triplication_faults w]

16      foreach leafcell $leafcells {
            . . .
32          if {([llength $voter] == 3 && $countVotingcells == 3) || ($voter ne "" &&
            $countVotingcells == 1)} {
33              if {[llength $cells] - [llength $voter] == 1} {
34                  # If yes, add to "triplicated"
35                  puts "$leafcell is indeed triplicated"
36                  puts $triplicated_log "$name, $leafcell"
37              } else {
38                  puts "$leafcell has too many output connections, check logs
                    for further details"
39                  puts $triplication_faults "$name, $leafcell, \[[join $cells ", "]\]"
40              }
41          } else {
42              # If not, add to "not_triplicated"
43              puts "$leafcell sadly is not triplicated"
44              puts $not_triplicated_log "$name, $leafcell"
45          }
            . . .
55      }
56      close $triplicated_log
57      close $not_triplicated_log
58      close $triplication_faults
59  }
```

Listing 6.5: Die Struktur des Prozessblocks get_triplicated_and_not_triplicated.

In Listing 6.5 wird in Zeile 4 eine Variable countVotingcells deklariert und mit dem Wert 3 belegt. In Zeile 8 wird countVotingcells mit dem Befehl global für den Prozessblock freigegeben, da keine außerhalb des Prozessblocks deklarierten Variablen im Prozessblock verwendet werden dürfen. In Zeile 12 bis 14 werden drei verschiedene Textdateien mit den Namen triplicated_log, not_triplicated_log und triplcation_faults erzeugt, mit open geöffnet und mit der Option w für Schreibrechte freigegeben. Ab Zeile 16 ist ein foreach-Konstrukt dargestellt, das mit der Angabe $leafcells jedes einzelne FlipFlop in einem Design untersucht. Dazu werden die FlipFlops mit einem if-Konstrukt unterschieden.

In Zeile 32 besteht die äußere if/else-Bedingung aus zwei Bedingungen, die durch ein ODER verknüpft werden. Die Bedingung ist erfüllt, wenn die Anzahl der verbundenen Voter im FlipFlop (llength $voter) und der Wert der Variablen countVotingcells gleich sind oder wenn der Name des Voters ungleich dem leeren String ist ($voter ne “”) und die Variable countVotingcells gleich 1 ist. In diesem Fall wird auf die innere if-else-Bedingung geprüft. Wenn das FlipFlop die äußere Bedingung nicht erfüllt, wird in Zeile 43 die Meldung „$leafcell sadly is not triplicated“ in die TCL-Konsole geschrieben, während in Zeile 44 das FlipFlop ($leafcell) mit dem Namen des Moduls ($name) in der Textdatei $not_triplicated_log gespeichert wird.

Die innere if-else-Bedingung in Zeile 33 prüft, ob der Ausgang eines FlipFlops tatsächlich nur mit drei MajorityVoter und keinen weiteren Zellen verbunden ist, indem die Anzahl aller Zellen (FlipFlops, Voter, etc.) von der Anzahl der Voter subtrahiert wird, so dass als Ergebnis die Zahl 1 herauskommt. Ist die Bedingung erfüllt, wird das FlipFlop in Zeile 35 mit der Meldung „$leafcell is indeed triplicated“ in die TCL-Konsole geschrieben und in Zeile 36 mit dem zugehörigen Modul in der Textdatei $triplicated_log gespeichert. Wenn der Ausgang des FlipFlops tatsächlich außer mit Votern noch mit weiteren Logikzellen verbunden ist, so dass die in der Bedingung geprüfte Differenz größer als 1 ist, wird das FlipFlop in Zeile 38 mit der Meldung „$leafcell has too many output connections, check logs for further details“ in die Konsole geschrieben. Außerdem wird das FlipFlop in Zeile 39 wird zusammen mit dem zugehörigen Modul und den verbundenen Zellen in der Textdatei $triplication_faults gespeichert.

In den Zeilen 56 bis 58 endet der Prozessblock get_triplicated_and_not_triplicated mit dem Schließen der Textdateien durch den Befehl close.

Abbildung 6.4 zeigt ein Anwendungsbeispiel für das reg2-Skript, in dem in Zeile 1176 des Hauptmoduls caeco_v das normale Signal anstelle des gevoteten Signals second_time_before_max_poolingVoted verwendet wird.

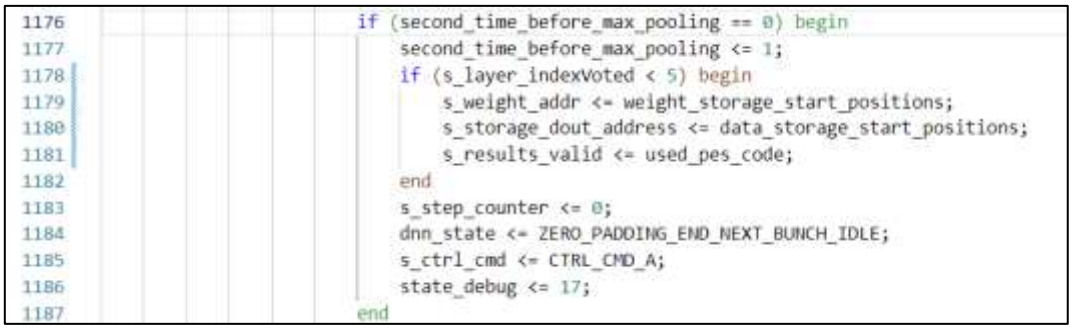

Abbildung 6.4: Injektion eines Votingfehlers in eine if-Abfrage in Zeile 1176, bei der das Signal second_time_before_max_poolingVoted nicht verwendet wird.

Der Code in Abbildung 6.4 führt dazu, dass die triplizierten Varianten des Signals second_time_before_max_pooling mit der Meldung in der Konsole erscheinen, dass diese Signale unzulässige Verbindungen besitzen. Abbildung 6.5 zeigt die entsprechende Meldung und in Abbildung 6.6 den relevanten Auszug aus der Textdatei triplication_faults_rtl.

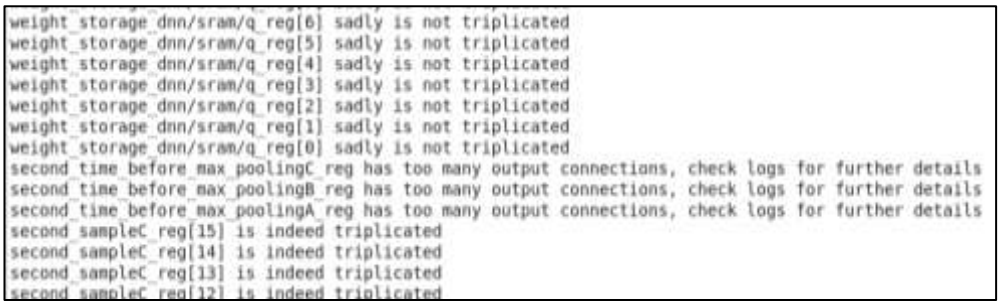

Abbildung 6.5: Meldungen einzelner Signale des TCL-Skripts reg2 in der TCL-Konsole, in der unzulässige Verbindungen für das Signal second_time_before_max_pooling gemeldet werden.

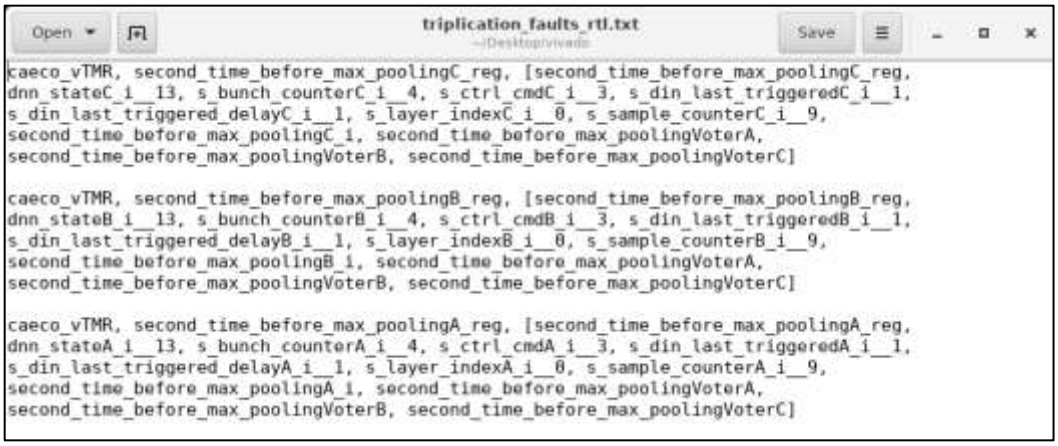

Abbildung 6.6: Die Textdatei triplication_faults_rtl.txt, in der das triplizierte Signal second_time_before_max_pooling mehrere Verbindungen aufweist.

In Abbildung 6.6 wird das oben genannte triplizierte Signal mit dem Modulnamen caeco_vTMR in der Textdatei triplication_faults_rtl.txt gespeichert, so dass in caeco_v das ursprüngliche Signal untersucht und korrigiert werden kann.

Die Untersuchung der Triplizierung mit Hilfe des reg2-Skripts in einem großen Design kann sehr zeitaufwendig sein. Die Untersuchung des Beschleunigers nimmt bis zu 40 Minuten in Anspruch.

7 Simulation von Single Event Effekten

Für die Simulation von SEEs wird die triplizierte Version des Beschleunigers synthetisiert wobei durch die Constraints in der XDC-Datei tmr_synth die Redundanz des Entwurfs beim Optimierungsvorgang erhalten bleibt. Wie in Kapitel 4.1 beschrieben, gibt es in Vivado zwei Möglichkeiten eine synthetisierte Netzliste zu exportieren. In diesem Projekt wird die Netzliste mit dem folgenden Befehl in der TCL-Konsole exportiert:

```
write_verilog -mode funcsim ./caeco/rtl_verilog_r2g/caeco_r2g.v
```

Dabei wird im Repository des Beschleunigers ein neuer Ordner mit dem Namen rtl_verilog_r2g angelegt, dem die Netzliste als caeco_r2g.v hinzugefügt wird. Mit der Option `-mode funcsim` wird eine Netzliste ausgegeben, die für eine Funktionssimulation verwendet werden kann, aber nicht mehr synthetisierbar ist [24].

Mit der Option „Export Netlist“ im GUI-Fenster kann die gleiche Netzliste exportiert werden, wie wenn der Befehl `write_verilog` ohne die Option `-mode funcsim` verwendet wird [24]. Wie jedoch in Listing 7.1 zu sehen ist, erzeugt dieser Befehl einen MajorityVoter, bei dem der Port tmrErr mit dem Signal \<const0> verknüpft wird. Dabei ist \<const0> als wire-Typ deklariert, während der Port tmrErr nicht deklariert ist, was zu einem Syntaxfehler im Quellcode führt.

With `write_verilog` command, via „Export Netlist“ option — a)

```
module majorityVoter__xdcDup__9
   (inA,
    inB,
    inC,
    out,
    .tmrErr(\<const0> ));
  input [15:0]inA;
  input [15:0]inB;
  input [15:0]inC;
  output [15:0]out;
  output \<const0> ;

  wire \<const0> ;
  (* RTL_KEEP = "yes" *) wire [15:0]inA;
  (* RTL_KEEP = "yes" *) wire [15:0]inB;
  (* RTL_KEEP = "yes" *) wire [15:0]inC;
  wire [15:0]out;
```

With `write_verilog` command + option `-mode funcsim` — b)

```
module majorityVoter__xdcDup__9
   (inA,
    inB,
    inC,
    out,
    tmrErr);
  input [15:0]inA;
  input [15:0]inB;
  input [15:0]inC;
  output [15:0]out;
  output tmrErr;

  wire \<const0> ;
  (* RTL_KEEP = "yes" *) wire [15:0]inA;
  (* RTL_KEEP = "yes" *) wire [15:0]inB;
  (* RTL_KEEP = "yes" *) wire [15:0]inC;
  wire [15:0]out;

  assign tmrErr = \<const0> ;
```

*Listing 7.1: Vergleich der Optionen **a)** über „Export Netlist“ im GUI-Fenster und **b)** über die Option -mode des Befehls write_verilog.*

Mit der exportierten Netzliste caeco_r2g.v können im SEEG-Modul die Verilog-Tasks für die Simulation von SETs und SEUs erzeugt werden, wobei eine annotierte Bauteilbibliothek namens xilinx_prims_tmrg.v verwendet wird, die im Ordner rtl_verilog_r2g zu finden ist. In dieser Bibliothek sind für die SEUs alle FilpFlops annotiert (siehe Kapitel 4.1), während für die SETs keine Annotation erforderlich ist.

Die erzeugten Verilog-Tasks werden dann für drei verschiedene Simulationen verwendet, in denen verschiedene Injektionen von SEUs durchgeführt werden, die in den Kapiteln 7.1 bis 7.3 beschrieben werden. Dazu wird wiederum eine neue Testbench mit dem Namen caeco_tb_ecg_file_r2g und ein Shell-Skript mit dem Namen simulate_caeco_r2g.sh erstellt, in dem die Simulationen in der Testbench caeco_tb_ecg_file_r2g durchgeführt und die benötigten Dateien in das Shell-Skript eingefügt werden. In der neuen Testbench wird die synthetisierte Netzliste mit dem Hauptmodul caeco_vTMR unter dem Namen DUT in die Testbench eingefügt. Dabei werden alle Bauteile der UNISIM-Bibliothek von Vivado [25] in einer Datei mit dem Namen sim_prims.v konkateniert, da für die instanziierten Bauteile in der Netzliste die annotierte Bibliothek xilinx_prims_tmrg.v nicht ausreicht. Zusätzlich wird das Modul glbl in der Testbench instanziiert, da dieses Modul Signale enthält, die in allen Bauteilen der UNISIM-Bibliothek verwendet werden. Die drei genannten Dateien caeco_r2g.v, sim_prims.v und glbl.v werden für die Simulation in das Shell-Skript eingefügt, wie in Listing 7.2 zu sehen ist.

```
RTLD_VHDL=$(pwd)/../rtl/
RTLD_VLOG_R2G=$(pwd)/../rtl_verilog_r2g/

rm -rf ./.work/*
mkdir -p .work/results

files_vh="\
    $RTLD_VHDL/conv_pkg.vhd \
    $RTLD_VHDL/caeco_pack.vhd \
    $RTLD_VHDL/impl/caeco_debug_paths.vhd \
    $RTLD_VHDL/caeco_data_buffer.vhd \
    $RTLD_VHDL/impl/SRAM_DATA_STORAGE_4096x32.vhd \
    $RTLD_VHDL/impl/SRAM_WEIGHT_STORAGE_1024x8.vhd \
    $RTLD_VHDL/caeco_data_storage.vhd \
    $RTLD_VHDL/impl/caeco_data_storage_asic.vhd \
    $RTLD_VHDL/caeco_weight_storage.vhd \
    $RTLD_VHDL/impl/caeco_weight_storage_debug.vhd \
    $RTLD_VHDL/caeco_pe.vhd \
    $RTLD_VHDL/caeco.vhd \
    $(pwd)/legacy_dont_use.vhd \
"

files_v="\
    $RTLD_VLOG_R2G/glbl.v \
    $RTLD_VLOG_R2G/sim_prims.v \
    $RTLD_VLOG_R2G/caeco_r2g.v \
"

files_sv="\
    $(pwd)/caeco_tb_ecg_file_r2g.sv \
"
```

Listing 7.2: Übersicht der aufgelisteten Dateien im Shell-Skript simulate_caeco_r2g.sh.

In dieser Arbeit wurden für die Simulation zwei Verilog-Tasks verwendet, wobei der Task `seu_force_net` die SEUs injiziert bzw. die gelisteten Netze forciert und der Task `seu_release_net` diese wieder freigibt. Für die Durchführung der drei Injektionstypen wird eine Verilog-Headerdatei namens caeco_seeg.vh erstellt, die mit einer include-Direktive in die Testbench caeco_tb_ecg_file_r2g eingebunden wird. Wie in Listing 7.3 zu sehen, sind in der Headerdatei acht Tasks vorhanden. Die Tasks mit den Buchstaben A, B und C enthalten jeweils die A-, B- oder C-Varianten der triplizierten Signale, während die Tasks

`seu_force_net` und `seu_release_net` alle Varianten enthalten. Dabei wurden alle Netze, die nicht tripliziert sind, entfernt.

```
task seu_force_netA; . . .
endtask

task seu_force_netB; . . .
endtask

task seu_force_netC; . . .
endtask

task seu_force_net; . . .
endtask

task seu_release_netA; . . .
endtask

task seu_release_netB; . . .
endtask

task seu_release_netC; . . .
endtask

task seu_release_net; . . .
endtask
```

Listing 7.3: Übersicht der Verilog-Headerdatei caeco_seeg.

Zusätzlich wurde in der Testbench ein Parameter namens sim_number deklariert. Jeder der drei Injektionstypen ist in eine if-Bedingung eingebunden, in der abgefragt wird, ob sim_number den für die Durchführung einer bestimmten Injektion erforderlichen Wert besitzt. Wenn, wie in Listing 7.4 zu sehen, sim_number diesen Wert hat, wird der jeweilige Injektionstyp ausgeführt.

7.1 Normale Injektion

```
137     parameter sim_number = 1;
. . .
216         if (sim_number == 1) begin
217             for (i = 0; i < 618; i = i + 1) begin
218             seu_force_netA(i);
219             end
220         end
```

Listing 7.4: Die Injektion aller A-Netze mit dem Task seu_force_setA() in einer for-Schleife.

In dieser Injektion wird verifiziert, ob die Korrektur über die MajorityVoter tatsächlich funktioniert, wenn das A-Netz kontinuierlich forciert wird, indem der Wert des B- und C-Netzes ausgegeben wird. Abbildung 7.1 zeigt dazu eine Waveform, bei der das Signal dnn_stateA durchgehend forciert wird, während dnn_stateB und dnn_stateC normal weiterlaufen. Es ist zu erkennen, dass alle drei MajorityVoter dnn_stateVotedA, B und C im Vergleich zum Original identische Werte ausgeben.

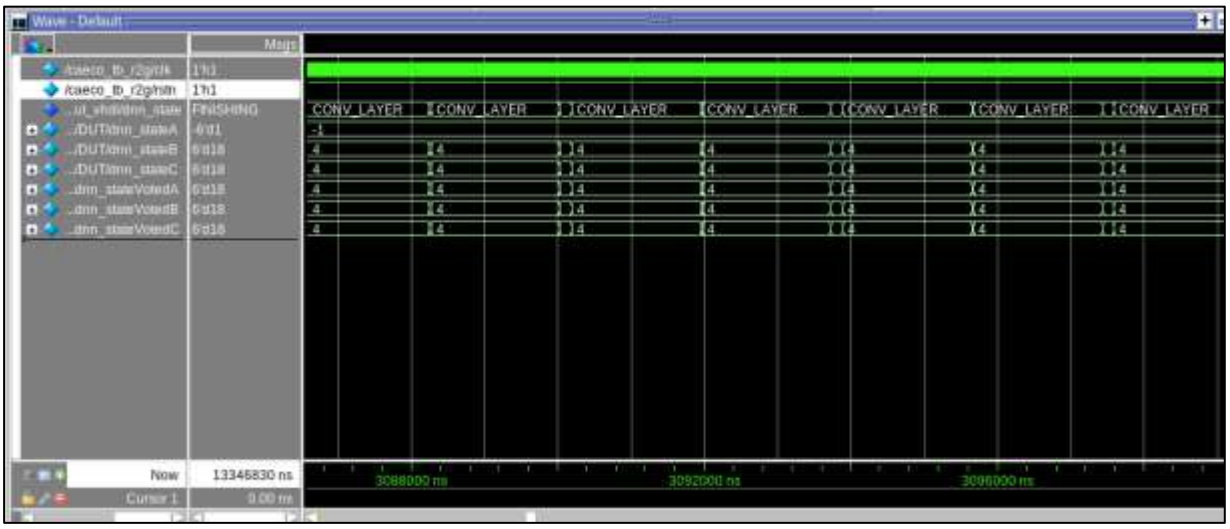

Abbildung 7.1: Darstellung der fortlaufenden Injektion eines SEU im Signal dnn_stateA.

Im Falle einer kontinuierlichen Forcierung aller A- und B-Netze würden die MajorityVoter die forcierten Werte weitergeben, so dass die Endergebnisse der Netzliste nicht mehr identisch wären, wie in Abbildung 7.2 dargestellt. Dabei ist zusätzlich zu erkennen, dass das Signal dnn_stateC, das als C-Netz nicht forciert ist, einen Nullwert ausgibt.

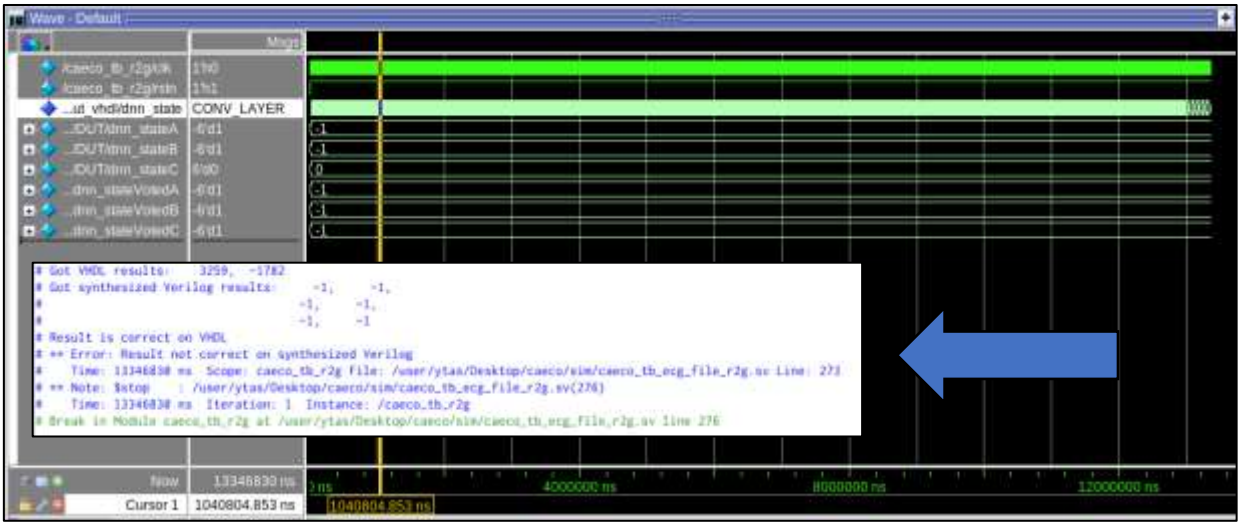

Abbildung 7.2: Kontinuierliche Forcierung aller A- und B-Netze mit dem Resultat, dass die Endergebnisse nicht mit dem Original übereinstimmen.

7.2 Sequenzielle Injektion

Bei der sequenziellen Injektion werden Signale aller Netzvarianten (A, B, C) forciert, indem in jedem zweiten Taktsignal bei einer fallenden Taktflanke eine Netzvariante forciert und im gleichen Takt wieder freigegeben wird, so dass bei der nächsten fallenden Taktflanke dasselbe mit der nächsten Netzvariante geschieht. Diese Art der Injektion erfolgt zyklisch. Durch diese Injektion wird geprüft, ob die eingebaute Rückführung des MajorityVoters tatsächlich funktioniert und der Wert nach der Injektion korrigiert wird. Listing 7.5 zeigt den Initialblock in der Testbench caeco_tb_ecg_file_r2g, der diese Art der Injektion durchführt.

```
144 if (sim_number == 2) begin
145     initial begin
146         if (RunVlogParallel == 1) begin
147             #1250;
148             forever @(negedge clk) begin
149                 for (i = 0; i < 618; i = i + 1) begin
150                     seu_force_netA(i);
151                 end
152                 #1;
153                 for (i = 0; i < 618; i = i + 1) begin
154                     seu_release_netA(i);
155                 end
156                 @(negedge clk);
157                 for (i = 0; i < 618; i = i + 1) begin
158                     seu_force_netB(i);
159                 end
160                 #1;
161                 for (i = 0; i < 618; i = i + 1) begin
162                     seu_release_netB(i);
163                 end
164                 @(negedge clk)
165                 for (i = 0; i < 618; i = i + 1) begin
166                     seu_force_netC(i);
167                 end
168                 #1;
169                 for (i = 0; i < 618; i = i + 1) begin
170                     seu_release_netC(i);
171                 end
172             end
173         end
174     end
175 end
```

Listing 7.5: Der Quellcode, in dem die sequenzielle Injektion zyklisch ausgeführt wird.

Die Prozeduren in den Zeilen 156 und 164 stellen sicher, dass die Injektion der nächsten Netzvariante auf der fallenden Taktflanke erfolgt. Die "#1" in den Zeilen 152, 160 und 168 verzögern die Freigabe der Netze um eine Nanosekunde durch die in Zeile 1 der Testbench beschriebene `timescale`. Die zyklische Ausführung erfolgt auf Grund der forever-Anweisung in Zeile 148.

Abbildung 7.3 stellt die sequenzielle Injektion in einer Waveform dar und zeigt, dass durch die Korrektur der Werte von dnn_stateA, B und C die Rückführung der MajorityVoter tatsächlich in der synthetisierten Netzliste vorhanden ist und durch die identischen Endergebnisse der Netzliste im Vergleich zum Original auch funktioniert.

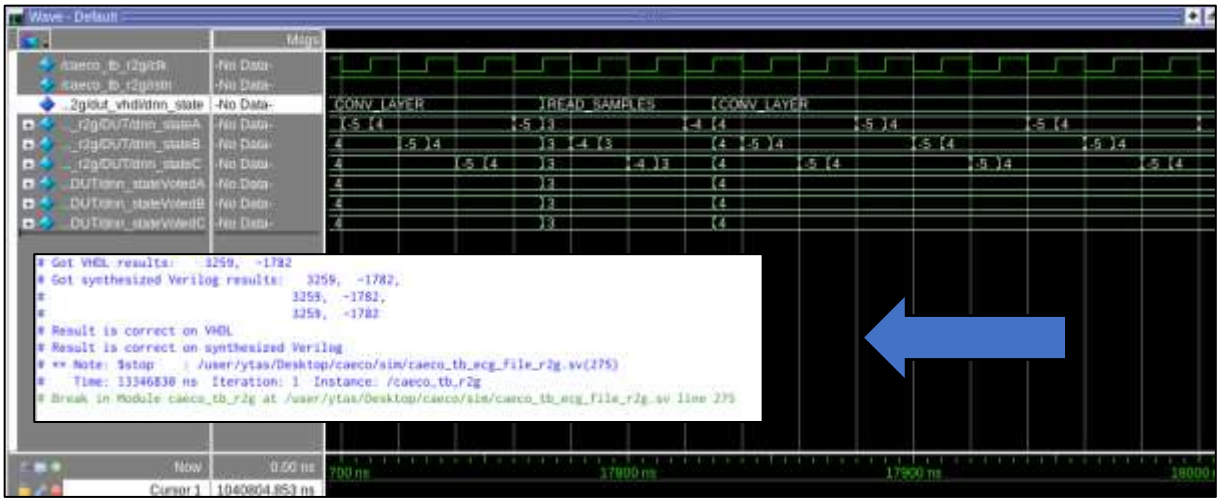

Abbildung 7.3: Sequenzielle Forcierung aller Netzvarianten mit dem Resultat, dass die Endergebnisse mit dem Original übereinstimmen.

7.3 Zufällige Injektion

Bei dieser Art der Injektion wird in jedem zweiten Taktsignal bei einer fallenden Taktflanke ein einzelnes Signal zufällig forciert und im gleichen Takt wieder freigegeben. Listing 7.6 zeigt den Quellcode, in dem die Injektion zyklisch durchgeführt wird.

```
176 else if (sim_number == 3) begin
177     initial begin
178         if (RunVlogParallel == 1) begin
179             #1250;
180             forever @(negedge clk) begin
181                 i = $urandom_range(0, 1853);
182                 seu_force_net(i);
183                 #1;
184                 seu_release_net(i);
185                 //@(negedge clk);
186             end
187         end
188     end
189 end
```

Listing 7.6: Der Quellcode, in dem die zufällige Injektion zyklisch ausgeführt wird.

Um ein bestimmtes Signal zufällig zu forcieren, müssen Zufallszahlen erzeugt werden. In Zeile 181 wird dem Integer i die Funktion `$urandom_range()` zugewiesen, die Pseudozufallszahlen in einem bestimmten Zahlenbereich erzeugt. Der Integer wird dann den Tasks `seu_force_net` und `seu_release_net` zugewiesen, in denen ein beliebiges Signal bei einer fallenden Flanke forciert bzw. freigegeben wird. Diese Tasks enthalten alle Signale der Netzvarianten A, B und C, jedoch nicht die Signale, die nicht tripliziert wurden, so dass die Gesamtzahl 1853 beträgt.

Abbildung 7.4 zeigt, dass die Signale `dnn_stateA[2]` und `dnn_stateA[5]` zufällig injiziert wurden. Wie bei der sequenziellen Injektion funktioniert die Rückkopplung des MajorityVoter, indem der Wert nach der Injektion korrigiert wird.

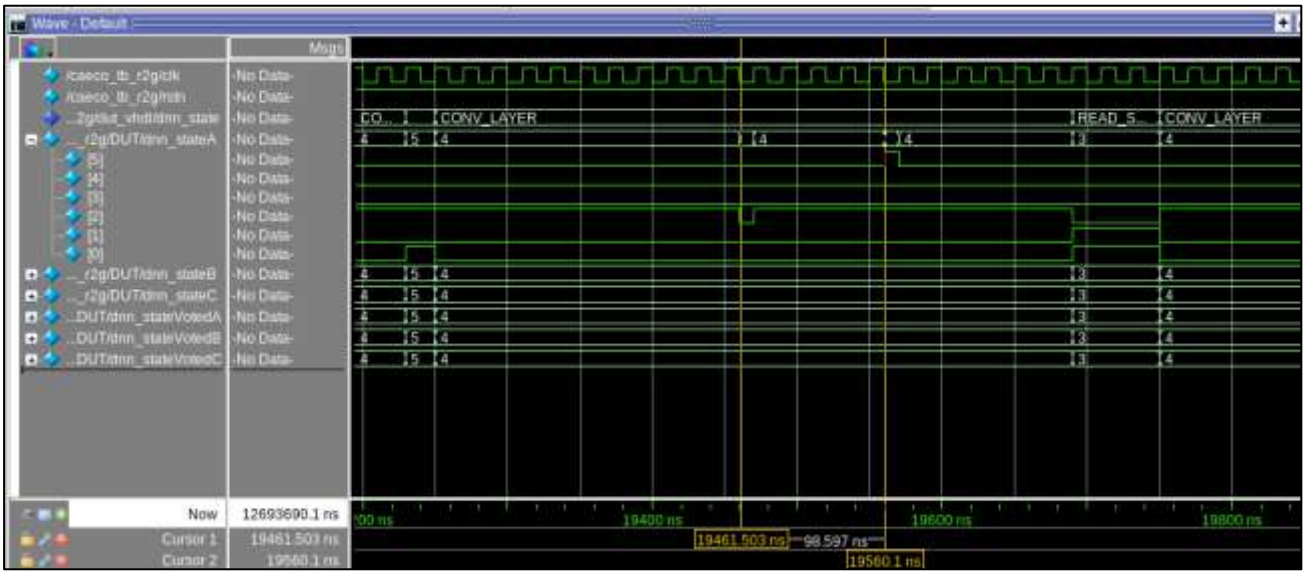

Abbildung 7.4: Simulation der zufälligen Injektion, bei der die Signale dnn_stateA[2] und dnn_stateA[5] zufällig forciert und freigegeben werden.

8 Zusammenfassung und Ausblick

Das Ziel dieser Arbeit war es, den CAECO-Beschleuniger durch TMR gegen nicht-destruktive Single Event Effekte zu sichern. Zunächst wurde gezeigt, wie jedes einzelne Modul des Beschleunigers vollständig tripliziert werden kann und wie das Voting der Signale korrekt durchgeführt werden kann. Dabei gab es Signaltypen, bei denen eine andere Vorgehensweise angewendet werden musste, so dass z.B. für das Bit-Array s_results_buffer eine eigene Headerdatei vorgesehen wurde. Außerdem mussten einige Module für die SRAM-Speicher manuell geschrieben werden, da die automatisch generierten Versionen nicht für die Triplizierung bzw. Strahlungshärtung geeignet waren. Dazu wurde der Quellcode des Beschleunigers so angepasst, dass in den einzelnen Untermodulen neben dem ursprünglichen Taktsignal clk des Hauptmoduls auch die triplizierte Signalversion clkT verwendet wird.

Für die Überprüfung der Triplizierung wurde eine statische Verifikation mit dem TCL-Skript reg2 und eine dynamische Verifikation mit der SV-Testbench caeco_tb_ecg_file_tmr in Questasim durchgeführt. Bei der statischen Verifikation wurde z.B. überprüft, ob der Ausgang eines FlipFlops tatsächlich nur mit den drei MajorityVoter verbunden ist. Durch die dynamische Simulation sollte geprüft werden, ob durch die Triplizierung funktionale Fehler eingeführt worden sind.

Anschließend wurde das triplizierte Design für eine Simulation von Single Event Effekten synthetisiert und Verilog-Tasks mit Hilfe des SEEG-Moduls und der annotierten Bibliothek xilinx_prims_tmrg erstellt, welche die aufgelisteten Signale forcen und freigeben. Es wurden drei Injektionstypen vorgestellt, für die der entsprechende Quellcode und die Simulation gezeigt wurden.

Die Ergebnisse zeigen, dass das vollständig triplizierte Design des Beschleunigers tatsächlich die gleiche Funktionalität wie das ursprüngliche Design aufweist und darüber hinaus gegen Single Event Effekte geschützt ist.

Ausgehend von dieser Erkenntnis wäre der nächste logische Schritt in einem Folgeprojekt die Simulation von Single Event Transients mit den in Kapitel 7 genannten Injektionstypen, da aus Zeitgründen nur mit Single Event Upsets simuliert wurde. Anschließend muss der Speicher gehärtet werden, indem die Blockspeicher in Dual-Port-Speicher verwendet wird. Mit Hilfe eines Kanalcodes wie z.B. dem Hamming-Code kann durch Data-Scrubbing d.h. dem zyklischen Auslesen der SRAM-Daten Fehler erkannt und korrigiert werden. Nach der Verifikation des Speichers wird schließlich der vollständig strahlungsresistente CAECO-Beschleuniger mit dem bereits triplizierten RISC-V-Prozessorkern verbunden.

9 Literaturverzeichnis

[1]: Convolutional Neural Network, Wikipedia, https://de.wikipedia.org/wiki/Convolutional_Neural_Network

[2]: deeplizard, Convolutional Neural Networks (CNNs) explained, https://deeplizard.com/learn/video/YRhxdVk_sIs

[3]: deeplizard, Zero Padding in Convolutional Neural Networks explained, https://deeplizard.com/learn/video/qSTv_m-KFk0

[4]: deeplizard, Max Pooling in Convolutional Neural Networks explained, https://deeplizard.com/learn/video/ZjM_XQa5s6s

[5]: Diego Unuzeta, Fully Connected Layer vs. Convolutional Layer: Explained (Abschnitt „What Is a Fully Connected Layer?“), BuiltIn 2022, https://builtin.com/machine-learning/fully-connected-layer

[6]: H. Wöhrle, M. De Lucas Alvarez, F. Schlenke, A. Walsemann, M. Karagounis and F. Kirchner, "Surrogate Model based Co-Optimization of Deep Neural Network Hardware Accelerators," 2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), 2021, pp. 40-45, doi: 10.1109/MWSCAS47672.2021.9531708.

[7]: Ajitesh Kumar, Transposed Convolution vs. Convolution Layer: Examples (Abschnitt „What are Convolutional Transpose Layers?“), https://vitalflux.com/transposed-convolution-vs-convolution-layer-examples/

[8]: POMAA, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, https://robotik.dfki-bremen.de/de/forschung/projekte/pomaa/

[9]: CAECO, Learning_Chips_Lab, GitLab, https://gitlab.ikt.fh-dortmund.de:9443/learning_chips_lab/caeco (Zugriff nur als Mitglied möglich!)

[10]: Single Event Effects, NASA/GSFC Radiation Effects & Analysis, https://radhome.gsfc.nasa.gov/radhome/see.htm

[11]: Richard Jung, Radiation Qualification of the Cologne Chip GateMate A1 FPGA (Abschnitt 2.2.3), PubliDo 2023, https://opus.bsz-bw.de/fhdo/frontdoor/index/index/searchtype/latest/docId/3364/start/9/rows/10 , https://opus.bsz-bw.de/fhdo/frontdoor/deliver/index/docId/3364/file/Master_Thesis_Richard_Jung-1.pdf (PDF)

[12]: Triple Module Redundancy Generator (TMRG), GitLab, https://gitlab.cern.ch/tmrg/tmrg, https://tmrg.web.cern.ch/tmrg/ (Dokumentation)

[13]: Szymon Kulis, Single Event Upsets mitigation techniques, CERN EP-ESE Electronics Seminars 2016, https://indico.cern.ch/event/465343/

[14]: K. J. Hass and J. W. Ambles, "Single event transients in deep submicron CMOS," (Abschnitt „Introduction“), 42nd Midwest Symposium on Circuits and Systems (Cat. No.99CH36356), Las Cruces, NM, USA, 1999, pp. 122-125 vol. 1, doi: 10.1109/MWSCAS.1999.867224.

[15]: Künstliche Neuronale Netze (Abschnitt: Anwendung), Wikipedia, https://de.wikipedia.org/wiki/K%C3%BCnstliches_neuronales_Netz#Anwendung

[16]: GHDL-Yosys-Plugin, Github, https://github.com/ghdl/ghdl-yosys-plugin, http://ghdl.github.io/ghdl/using/Synthesis.html (Dokumentation)

[17]: TMRG Config Generator, Github, https://github.com/mkaragou/TMRG_Config_Generator

[18]: Vivado Design Suite Properties Reference Guide (UG912) 2023.1 English, https://docs.xilinx.com/r/en-US/ug912-vivado-properties

[19]: Akshay Atam, Dilated Convolution [explained] (Abschnitt: Introduction to Dilated Convolution), Opengenus IQ 2023, https://iq.opengenus.org/dilated-convolution/

[20]: Single Event Effect, Wikipedia, https://de.wikipedia.org/wiki/Single_Event_Effect

[21]: Questa Advanced Simulator, Siemens Software, https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[22]: ModelSim User's Manual, https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user (PDF)

[23]: Vivado Design Suite User Guide: Designing with IP (UG896) 2023.1 English, https://docs.xilinx.com/r/en-US/ug896-vivado-ip/Vivado-Design-Suite-Documentation

[24]: Vivado Design Suite Tcl Command Reference Guide (UG835) 2023.1 English, https://docs.xilinx.com/r/en-US/ug835-vivado-tcl-commands/Introduction

[25]: Vivado Design Suite User Guide: Logic Simulation (UG900) 2021.1 English, https://docs.xilinx.com/r/2021.1-English/ug900-vivado-logic-simulation