HAW HAMBURG

Bachelorarbeit

Jonas Fuhrmann

Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Jonas Fuhrmann

Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Tim Tiedemann
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 20. November 2018

Jonas Fuhrmann

Thema der Arbeit

Implementierung einer Tensor Processing Unit mit dem Fokus auf Embedded Systems und das Internet of Things

Stichworte

Maschinelles Lernen, Tensor Processing Unit, FPGA, VHDL, Embedded Systems, Internet of Things, Computer Vision, Künstliche Intelligenz

Kurzzusammenfassung

Maschinelles Lernen findet immer mehr Anwendung in unserem Alltag, aber auch sicherheitskritische Systeme werden immer häufiger mit ML-Verfahren ausgestattet. Diese Arbeit gibt einen Einblick in die Realisierung eines Machine-Learning-Co-Prozessors für Embedded Systems und IoT-Geräte. Dabei wurde eine skalierbare Architektur mit Anlehnung an Google's Tensor Processing Units umgesetzt. Kleinere Systeme können so mit diesem Beschleuniger ausgestattet werden und neben der parallelen Ausführung von ML-Modellen noch andere Echtzeitaufgaben übernehmen.

Jonas Fuhrmann

Title of Thesis

Implementation of a Tensor Processing Unit with focus on Embedded Systems and the Internet of Things

Keywords

Machine Learning, Tensor Processing Unit, FPGA, VHDL, Embedded Systems, Internet of Things, Computer Vision, Artificial Intelligence

Abstract

Machine learning is more and more applied in our everyday life, but also safety critical systems are increasingly equipped with ML procedures. This paper gives an insight into the implementation of a machine learning co-processor for embedded systems and IoT devices. A scalable architecture based on Google's Tensor Processing Units was implemented. This allows smaller systems to be equipped with this accelerator and to perform other real-time tasks in addition to the parallel execution of ML models.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

AMBA Advanced Microcontroller Bus Architecture.

AR Augmented Reality.

ASIC Application Specific Integrated Circuit.

AXI Advanced eXtensible Interface.

BRAM BlockRAM.

BSP Board Support Package.

CISC Complex Instruction Set Computer.

CNN Convolutional Neural Net.

DMA Direct Memory Access.

DSP Digital Signal Processing.

FIFO First In First Out.

FPGA Field Programmable Gate Array.

HLS High-Level Synthesis.

IoT Internet of Things.

IP Core Intellectual Property Core.

LUT Lookup-Tabelle.

LUTRAM Verteilter Speicher.

ML Maschinelles Lernen.

MLP Multilayer Perceptron.

MSB Most Significant Bit.

MXU Matrix Multiply Unit.

ReLU Rectifier Linear Unit.

RISC Reduced Instruction Set Computer.

RTL Register Transfer Level.

SDK Software Development Kit.

SoC System-on-a-Chip.

TPU Tensor Processing Unit.

V$_{HSIC}$HDL Very High Speed Integrated Circuit Hardware Description Language.

VPU Vision Processing Unit.

VR Virtual Reality.

Symbolverzeichnis

C° Maßeinheit der Temperatur.

OPS Einheit für Operationen pro Sekunde.

W Internationale Einheit für Leistung.

1 Einleitung

1.1 Motivation

Maschinelle Lernverfahren finden immer häufiger ihren Weg in Internet of Things (IoT) und mobile Geräte. Ob der persönliche Assistent mit Amazon Alexa und dem Google Assistant oder markerloses Tracking für Augmented Reality (AR) auf Smartphones oder Virtual Reality (VR)/AR-Headsets, Anwendungen sind immer wieder in neuen Formen zu sehen. Aber auch in sicherheitskritischen Branchen wird Maschinelles Lernen (ML) und die möglichst schnelle Ausführung von ML-Modellen immer wichtiger. Autonomes Fahren basiert häufig stark auf diesen Verfahren und kann bei zu langsamem Reagieren oder falschen Entscheidungen des Systems lebensgefährlich werden. Ebenso wird in der Medizin immer mehr mit ML gearbeitet um bestimmte Krankheitsfälle schneller einschätzen zu können als es Menschen möglich ist.

1.2 Zielsetzung

Diese Arbeit befasst sich damit einen ML-Co-Prozessor, ähnlich einer Tensor Processing Unit, zu planen, implementieren und zu evaluieren. Dabei soll der Prozessor ausreichend klein und sparsam sein, sodass dieser in Embedded Systems oder IoT Geräten verwendet werden kann, als auch eine schnelle Ausführung von ML-Modellen bieten. Host-Systeme können so neben den vom Co-Prozessor ausgeführten ML-Modellen, andere, eventuell wichtigere Prozesse ausführen. Der Co-Prozessor wird dabei auf einem modernen Field Programmable Gate Array (FPGA) mit integriertem System-on-a-Chip (SoC) getestet, wobei die Ressourcen des FPGAs sparsam und zugleich sinnvoll eingesetzt werden sollen. Entsprechende Software für die Zugriffe des Co-Prozessors und das Exportieren und Aufbereiten von ML-Modellen ist ebenfalls Teil dieser Arbeit.

1.3 Bedarf

Im Rahmen dieser Arbeit wurden Meinungen von Unternehmen zu diesem Thema eingeholt. Hierbei wurde auf die Nutzbarkeit solcher Co-Prozessoren eingegangen.

> "As machine learning expands into more and more real time use cases and privacy and mobile network latency limiting the ability to transfer data to a data center we will need better hardware and software to make it easier to move machine learning to the mobile client."
> - Jörg Heilig, VP Software Engineering bei Uber

> "We see that most new data is now generated by mobile and IoT devices, and to be able to use this data for real-time decision-making, it is paramount that the processing can happen within these devices."
> - Andreas Schmidt Jensen, Head of Mobile Intelligence Technology bei LINK Mobility Group ASA

1.4 Aufbau der Ausarbeitung

Im Anschluss folgt eine Erläuterung der Grundlagen, um Begrifflichkeiten zu klären und für das notwendige Verständnis zu sorgen. Ebenso wird hier der Stand der Technik erläutert. Daraufhin folgt die Konzeption, wobei auf die Grundlagen zurückgegriffen wird. Dabei wird der grundlegende Aufbau des Co-Prozessors entsprechend erläutert, visualisiert und auf die Arbeitsweise eingegangen. Der anschließende Teil befasst sich mit der Implementierung und dem Einsatz des FPGAs. Im vorletzten Teil wird eine Evaluation des Prozessors vorgenommen, wobei neben Parametern wie der Geschwindigkeit, Skalierbarkeit und Genauigkeit auch auf den Stromverbrauch eingegangen wird. Die Arbeit wird mit einer Zusammenfassung abgeschlossen, wobei Ausblicke auf Weiterentwicklungen erläutert werden.

2 Grundlagen

2.1 Matrixmultiplikation

Eine Matrixmultiplikation beschreibt eine Verknüpfung von Matrizen. So entsteht ein Element des Matrixproduktes durch die Summierung der elementweisen Multiplikation zweier Vektoren der zu verknüpfenden Matrizen. Dabei gilt die Multiplikation von Zeilen- und Spaltenvektoren, wobei die Spaltenzahl der ersten Matrix mit der Zeilenzahl der zweiten Matrix übereinstimmen muss [11]. Gegeben seien zwei Matrizen A und B, dann ergibt sich das Matrixprodukt C wie folgt:

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1n} \\ A_{21} & A_{22} & \dots & A_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ A_{l1} & A_{l2} & \dots & A_{ln} \end{bmatrix}, B = \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1m} \\ B_{21} & B_{22} & \dots & B_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ B_{n1} & B_{n2} & \dots & B_{nm} \end{bmatrix}$$

$$C = A \cdot B$$
$$= \begin{bmatrix} (A_{11} \cdot B_{11} + A_{12} \cdot B_{21} + \dots + A_{1n} \cdot B_{n1}) & \dots & (A_{11} \cdot B_{1m} + A_{12} \cdot B_{2m} + \dots + A_{1n} \cdot B_{nm}) \\ (A_{21} \cdot B_{11} + A_{22} \cdot B_{21} + \dots + A_{2n} \cdot B_{n1}) & \dots & (A_{21} \cdot B_{1m} + A_{22} \cdot B_{2m} + \dots + A_{2n} \cdot B_{nm}) \\ \vdots & \ddots & \vdots \\ (A_{l1} \cdot B_{11} + A_{l2} \cdot B_{21} + \dots + A_{ln} \cdot B_{n1}) & \dots & (A_{l1} \cdot B_{1m} + A_{l2} \cdot B_{2m} + \dots + A_{ln} \cdot B_{nm}) \end{bmatrix}$$

2.2 Diskrete Faltung

Bei einer diskreten Faltung handelt es sich um einen Operator aus der Signal- und Bildverarbeitung, dabei wird ein Faltungskern über das Bild oder das Signal gefahren. Die übereinander liegenden Werte werden dann multipliziert und aufsummiert. Dies ist dann ein Element des Ausgangssignals. f, g seien Signale mit einem endlichen, diskreten Definitionsbereich, dann ist die Faltung $(f * g)(n)$ definiert als [29]:

$$(f * g)(n) = \sum_{k=0}^{N-1} f(k) \cdot g(n-k)$$

Handelt es sich um 2 dimensionale Signale, so kann die Faltung $(f * g)(n_x, n_y)$ ermittelt werden mit [29]:

$$(f * g)(n_x, n_y) = \sum_{k_x=0}^{N_x-1} \sum_{k_y=0}^{N_y-1} f(k_x, k_y) \cdot g(n_x - k_x, n_y - k_y)$$

Die diskrete Faltung $(f * g)(n)$ lässt sich auch als Matrixprodukt bzw. Matrix-Vektor-Produkt beschreiben [29]:

$$f = \begin{bmatrix} f_0 & f_1 & \dots & f_{m-1} \end{bmatrix}, g = \begin{bmatrix} g_0 & g_1 & \dots & g_{n-1} \end{bmatrix}$$

$$f * g = f \cdot G$$

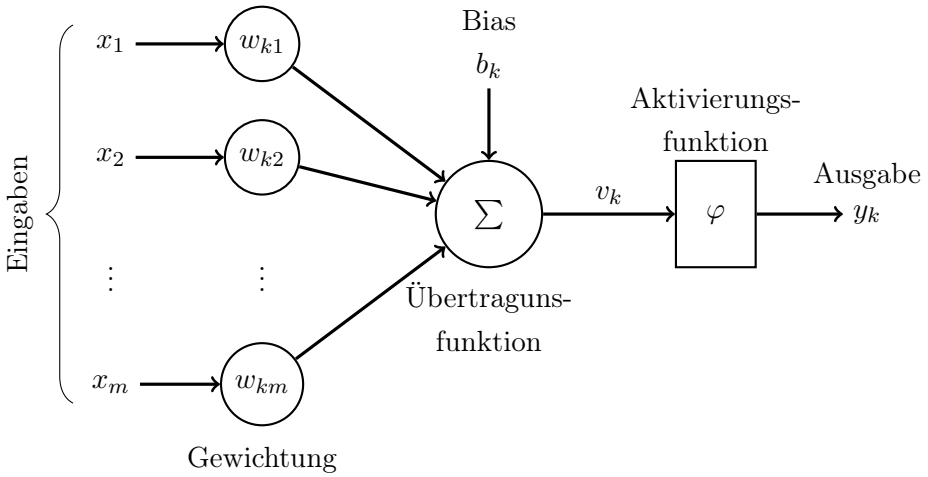
$$\text{mit } G = \begin{bmatrix} g_0 & g_1 & g_2 & \dots & g_{n-1} & 0 & 0 & 0 & \dots & 0 \\ 0 & g_0 & g_1 & g_2 & \dots & g_{n-1} & 0 & 0 & \dots & 0 \\ 0 & 0 & g_0 & g_1 & g_2 & \dots & g_{n-1} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & g_0 & g_1 & g_2 & \dots & g_{n-1} & 0 \\ 0 & \dots & 0 & 0 & 0 & g_0 & g_1 & g_2 & \dots & g_{n-1} \end{bmatrix}$$

2.3 Machine Learning

Der Bereich des maschinellen Lernens (ML) untersucht die Rechenprozesse, die dem Lernen bei Mensch und Maschine zugrunde liegen [16]. Hierbei werden Repräsentationen von Lernprozessen erforscht und angewendet.

2.3.1 Künstliche Neuronale Netze

Bei künstlichen neuronalen Netzen handelt es sich um ein System, welches von der Funktionsweise eines Gehirns abgeleitet wurde [12]. Einheiten eines solchen Systems sind Neuronen, die Verbindungen mit anderen Neuronen eingehen und so einen Datenfluss definieren. So kann eine Neurone beliebig viele Eingaben besitzen. Jede Eingabe einer Neurone wird mit einem Gewicht versehen. Dieses stellt die Signifikanz einer Eingabe dar. Die gewichteten Eingaben werden dann durch eine Übertragungsfunktion akkumuliert und die daraus resultierende Netzeingabe in eine Aktivierungsfunktion gegeben, welche bestimmt wie stark die Ausgabe ausfällt. Zusätzlich kann ein Bias hinzugeführt werden um die Netzeingabe zu erhöhen oder zu senken [12].

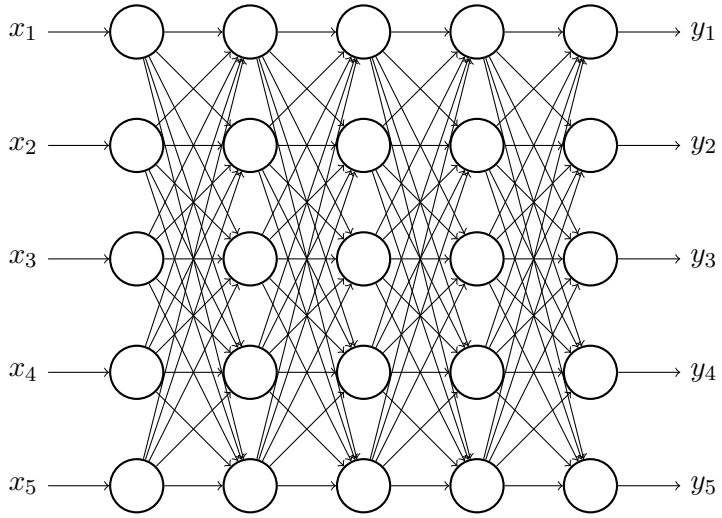


Abgeleitet von [12].
Der Bias b_k kann auch als Gewicht $w_k0 = b_k$ mit konstanter Eingabe $x_0 = 1$ angesehen werden.

Abbildung 2.1: Darstellung eines künstlichen Neurons

Multilayer Perceptron

Multilayer Perceptrons (MLP) sind künstliche neuronale Netze bei denen die Neuronen in Schichten angeordnet werden, dabei gibt es immer eine Eingabe- und Ausgabeschicht als auch beliebig viele Zwischenschichten (hidden layer). Alle Ausgaben einer Schicht werden mit den Eingaben der nächsten Schicht verbunden. Ein- und Ausgabeschichten werden abhängig von der Anwendung modelliert. MLPs können als Graph dargestellt werden, wobei ein Knoten des Graphs eine Neurone repräsentiert [12].



Abgeleitet von [12].
Die Anzahl der Neuronen einer Schicht kann beliebig sein.

Abbildung 2.2: Darstellung eines Multilayer Perceptrons als Graph

Convolutional Neural Nets

Convolutional Neural Nets (CNN) bestehen häufig aus vielen Schichten, basieren aber im Gegensatz zu MLPs auf Faltungen. So wird die Netzeingabe der Neuronen hier mittels einer Faltung der Eingaben und einem verhältnismäßig kleinen Faltungskern berechnet. Gewichte einer Schicht werden geteilt, was bedeutet dass alle Eingaben mit demselben angelernten Faltungskern verrechnet werden. Nach der Faltung folgt die Aktivierung der Neurone. Es können mehrere Faltungskerne pro Schicht trainiert werden. Die Ergebnisse der verschiedenen Faltungen werden in einer Feature Map abgelegt. Neben den Faltungsschichten werden schließlich sogenannte Pooling-Schichten eingeführt. Diese helfen dabei überflüssige Informationen zu verwerfen. Dabei werden Features ungefähr lokalisiert und es folgt eine Zusammenfassung der Daten in der Umgebung. Ein verbreitetes Beispiel ist das Max-Pooling, welches die Eingaben z.B. in 2x2 Matrizen einteilt und nur den größten Wert in diesen Regionen als Ausgabe bereitstellt. Netze können mit einer Fully-Connected Schicht abgeschlossen werden. Diese Schicht folgt wieder dem Prinzip des MLPs und wird häufig für Klassifizierungen verwendet [12].

2.3.2 Lernphase

Nach Mitchell wird das Lernen eines Systems wie folgt definiert:

> "Ein Computerprogramm soll aus Erfahrung E in Bezug auf eine Aufgabe T und eine Leistungskennzahl P **lernen**, wenn sich seine Leistung auf T, gemessen an P, mit Erfahrung E verbessert."
> - Tom M. Mitchell, 1997 [19]

Bei neuronalen Netzen bewirkt ein Lernalgorithmus die Anpassungen der Gewichte.

Supervised Learning

Supervised Learning basiert auf der Verfügbarkeit von Lerndaten. Dies bedeutet es existieren Datensätze für bekannte Eingaben des Netzes und dessen Lösungen (Labels). Der Term Supervised kommt von dem Prinzip des Lehrers, welcher auf Basis der Lerndatensätze die Eigenschaften des Systems anpasst [6].

Reinforcement Learning

Reinforcement Learning setzt auf Trial und Error. Hierbei wird dem Lernalgorithmus die Korrektheit der Ausgabe als Feedback vermittelt. In der Regel nimmt Reinforcement Learning mehr Zeit in Anspruch als Supervised Learning [30].

Unsupervised Learning

Für das Unsupervised Learning wird im Gegensatz zum Supervised oder Reinforcement Learning auf bekannte Datensätze oder auf Feedback verzichtet. Stattdessen versucht das System Regelmäßigkeiten bzw. Strukturen in den Daten selbstständig zu erkennen [6].

Verteilung mit Federated Learning

Federated Learning sorgt für eine Verteilung des Lernprozesses. Ein Koordinator hält dabei ein allgemein gültiges Modell. Teilnehmer des Federated Learning Prozesses beziehen das aktuelle Modell vom Koordinator und trainieren das Modell mit lokalen Daten. So können Modelle auch mit sensiblen Daten trainiert werden, ohne dass diese das Gerät verlassen. Anschließend werden die Änderungen des Modells dem Koordinator mitgeteilt. Dieser vereinheitlicht die Änderungen am Modell über alle mitgeteilten Änderungen und wendet diese an. Das neue aktualisierte Modell kann dann wieder von Teilnehmern abgeholt werden [18].
Neben dem Vorteil des Trainierens mit sensiblen Daten kann ein lokal aktualisiertes Modell sofort verwendet werden, ohne dass der Koordinator benötigt wird.

Federated Learning wird beispielsweise auf Android Geräten mit der Tastatur Gboard von Google verwendet, um Vorschläge zu verbessern.

2.3.3 Inference

Mit Abschließung der Lernphase entsteht ein sogenanntes Inference Modell. Dieses Modell wird dann für die Evaluierung unbekannter Daten verwendet. Im Falle des neuronalen Netzes beinhaltet dies die Gewichte und die Bekanntheit der genutzten Aktivierungsfunktionen.

2.3.4 Multilayer Perceptron als Matrizen

MLPs können mittels Matrixmultiplikationen berechnet werden. Gewichte werden dazu als Matrix W und Eingaben als Matrix X notiert. Das Resultat ist die Ausgabe V. Jede Spalte von W repräsentiert dabei die Gewichte jeder Eingabe einer Neurone und jede Zeile von X stellt eine Gesamteingabe in das Netz dar. Diese Darstellung wird für jede Schicht des Netzes angelegt [27]:

$$V_k = X_k \cdot W_k$$

Das Ergebnis V wird dann elementweise in die Aktivierungsfunktion $\varphi(v)$ übergeben, womit man die Ausgabe Y erhält [27]:

$$Y_k = \begin{bmatrix} \varphi(V_{11}) & \varphi(V_{12}) & \varphi(V_{13}) & \dots & \varphi(V_{1m}) \\ \varphi(V_{21}) & \varphi(V_{22}) & \varphi(V_{23}) & \dots & \varphi(V_{2m}) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \varphi(V_{m1}) & \varphi(V_{n2}) & \varphi(V_{n3}) & \dots & \varphi(V_{nm}) \end{bmatrix}$$

2.3.5 Aktivierungsfunktionen

Aktivierungsfunktionen sorgen für die Signifikanz der Ausgabe einer Neurone. Um ein lernfähiges System zu modellieren sollten diese nicht-linear sein. In der Praxis haben sich einige Aktivierungsfunktionen durchgesetzt. Am bekanntesten ist die Sigmoid Funktion und der Rectifier Linear Unit (ReLU). Es gibt aber auch andere Funktionen wie tanh oder Abwandlungen der ReLU Funktion. Die Funktionen Sigmoid s und ReLU r sind ohne Parameter wie folgt definiert [39]:

$$s(x) = \frac{1}{1 + e^{-x}}$$

$$r(x) = \begin{cases} x & \text{für } x > 0 \\ 0 & \text{sonst} \end{cases}$$

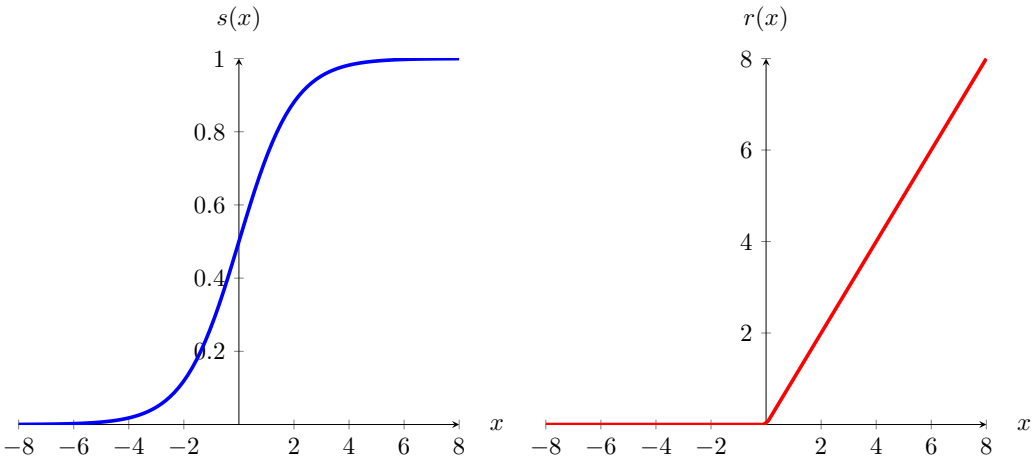


Abbildung 2.3: Sigmoid (Blau) und ReLU Funktion (Rot)

2.4 Festkommaarithmetik

In der Festkommaarithmetik wird (meist im Dualsystem) eine feste Position für das Komma gewählt. Stellen vor dem Komma gehören dann zum Integer und Stellen hinter dem Komma zur Fraction.

2.4.1 Q-Format

Das Q-Format ist ein Format für Festkommaarithmetik, hier gibt es wie beim Integer ein signed und unsigned Format. Das signed Format besteht auch hier aus dem Zweier-Komplement. Eine Festkommazahl wird im Q-Format angegeben mit $QI.F$ für signed und $QuI.F$ für unsigned, wobei I die Anzahl der Binärstellen des Integers und F die der Fraction darstellt [26]. Ist der Integer 0-Bit, so muss dieser nicht angegeben werden. Ein Beispiel für eine 8-Bit Festkommazahl mit 4-Bit Integer und Fraction:

$$\begin{aligned} Qu4.4 : x &= 0010.1101_2 \\ &= (0 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 + 1 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 1 \cdot \frac{1}{16})_{10} \\ &= 2.8125_{10} \end{aligned}$$

Festkommazahlen im Q-Format können wie gewöhnliche Integer behandelt werden. So ist die Größe des Integer-Teils bei einer Addition um 1 größer als der größte Integer-Teil der beiden Summanden und der Fraction-Teil so groß wie der größere der beiden Fraction-Teile. Bei einer Multiplikation wird die Größe der Integer- und Fraction-Teile addiert, andernfalls kann es zu einem Überlauf kommen oder die Genauigkeit wird geringer. Formal lassen sich diese Zusammenhänge wie folgt ausdrücken [26]:

$$\begin{aligned} &Qn.m + Qx.y \mapsto Qn + 1.y \text{ für } n \geq x, m \leq y \\ &Qn.m \cdot Qx.y \mapsto Qn + x.m + y \end{aligned}$$

2.5 Fließkommaarithmetik

Bei der Fließkommaarithmetik wird das Komma entsprechend einer Struktur angepasst. So kann das Komma bei der Ausführung von Operationen verschoben werden [28].

2.5.1 IEEE 754

IEEE 754 ist ein Standard für Fließkommazahlen. Bei der Single Precision Repräsentation werden 32-Bit verwendet. Diese werden eingeteilt in 23-Bit Mantisse, 8-Bit Exponent und 1-Bit Sign Flag [28].

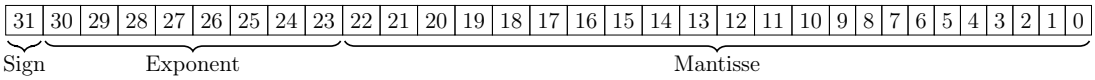


Abbildung 2.4: Aufbau des IEEE 754 Single Precision Standards

Das Sign Flag bestimmt das Vorzeichen der Fließkommazahl. Der Exponent bestimmt die Stelle des Kommas, dieser ist um einen Bias verschoben (hier 127). Die Mantisse ist der eigentliche Wert der Zahl. Zahlen sind immer normalisiert, was bedeutet, dass der Exponent so verschoben ist, sodass die Mantisse mit einer 1 beginnt und das Komma nach der 1 angesiedelt ist. Nach subtrahieren des Bias vom Exponenten e_b, das Verschieben des Kommas nach rechts um die vom Exponenten e angegebenen Stellen und Berücksichtigung des Vorzeichens s kann der Wert f der Mantisse m ermittelt werden [28]:

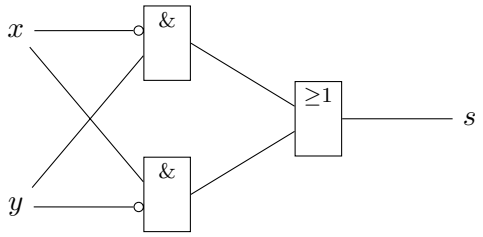
$$e = e_b - 127$$
$$f = (-1)^s \cdot m \cdot 2^e$$

2.6 Register Transfer Level

Register Transfer Level (RTL) beschreibt eine Abstraktionsebene in der Hardwaremodellierung. Hierbei wird ein Datenfluss zwischen Registern definiert. Diese Art von Logik wird kombinatorische Logik genannt. Die Beschreibung von Registern wird sequentielle Logik genannt [32].

2.6.1 Logikgatter und Lookup-Tabellen

Kombinatorische Logik kann mit Logikgattern realisiert werden. Diese bestehen aus Transistoren, wobei es verschiedene Typen von Logikgattern gibt. In der Regel stehen Gatter wie AND, OR und NOT zur Verfügung, wobei sich aus Kombination der Gatter andere Gatter wie NAND, NOR oder XOR realisieren lassen. Aus Kombination von Logikgattern kann jede beliebige Logikfunktion realisiert werden [20].



Zwei AND Gatter mit einem negierten Eingang und ein OR Gatter.

Abbildung 2.5: XOR Gatter

Kombinatorische Logik kann ebenfalls mit Lookup-Tabellen (LUT) realisiert werden [4]. Hierbei handelt es sich um einen kleinen Speicher, bei dem der Eingang eine Art Adresse darstellt. Der Ausgang verändert sich dann abhängig der Kombination des Einganges. Der Speicher erlaubt die Programmierung der Logikfunktion einer LUT.

2.6.2 Flip-Flops und Latches

Flip-Flops und Latches sind 1-Bit Datenspeicher, wobei Latches pegelgesteuert und Flip-Flops taktflankengesteuert sind, was bedeutet, dass der angelegte Wert nur mit der Änderung des Taktes übernommen wird. In RTL-Logik werden Flip-Flops als sequentielle Logik verwendet [10].

2.7 Field Programmable Gate Array

FPGAs fallen unter die Klasse der programmierbaren Logik. Dabei handelt es sich um Microchips mit der Funktionalität Logikfunktionen zu rekonstruieren. Hierbei wird mittels Interconnects auf verschiedene Ressourcen des FPGAs zurückgegriffen [2].

2.7.1 Fabric

Das Fabric eines FPGAs beinhaltet Ressourcen für die Rekonstruierung von RTL-Logik. Dabei besitzt das Fabric neben Flip-Flops auch Lookup-Tabellen.

2.7.2 Logikblöcke

Häufig besitzen FPGAs neben Standardkomponenten des Fabrics noch weitere Blöcke mit festen, häufig genutzten Logikfunktionen.

Digital Signal Processing Block

Digital Signal Processing (DSP)-Blöcke sind Blöcke mit fester Logikfunktion, welche häufig in der Signalverarbeitung Verwendung finden. So besitzen diese hauptsächlich einen verhältnismäßig großen Multiplizierer und Addierer, womit sich Operationen wie Multiply-Add oder Multiply-Accumulate realisieren lassen. Aufgrund der festen Logik sind DSP-Blöcke um ein vielfaches schneller als das FPGA Fabric [34].

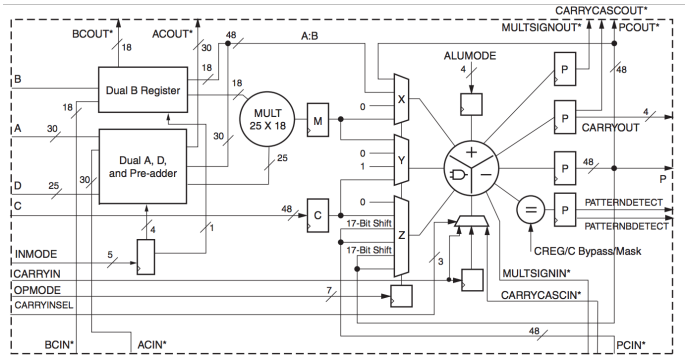


Abbildung 2.6: Aufbau eines DSP-Blocks in Xilinx FPGAs [34]

BlockRAM

Bei BlockRAM (BRAM) handelt es sich um Logikblöcke mit einem verhältnismäßig großen, synchronen Speicher. So können diese Blöcke im selben Taktsystem verwendet werden wie die RTL-Logik. BRAM kann kombiniert werden um beliebig breiten oder tiefen Speicher zu realisieren [33].
Speicher kann auch mit LUTs als verteilter Speicher oder Flip-Flops realisiert werden, was aber nicht für einen großen Speicher geeignet ist.

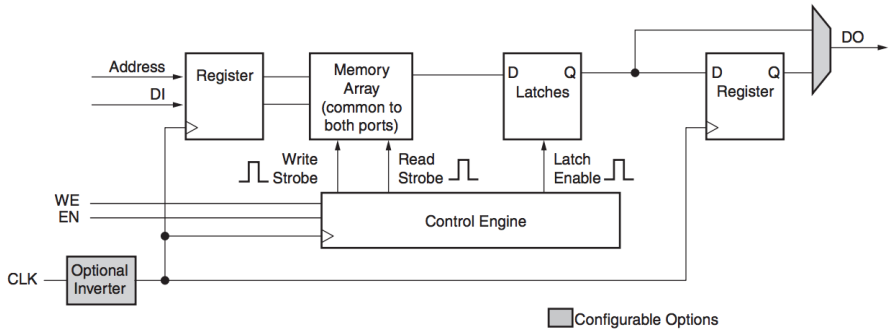


Abbildung 2.7: Aufbau eines BlockRAMs in Xilinx FPGAs [33]

2.8 Hardwarebeschreibungssprachen

Es gibt verschiedene Ansätze für die Definition von RTL-Logik. Eine Möglichkeit ist eine Beschreibung durch Hardwarebeschreibungssprachen.

2.8.1 V$_{HSIC}$HDL

Very High Speed Integrated Circuit Hardware Description Language (V$_{HSIC}$HDL) ist eine Hardwarebeschreibungssprache, wobei nicht einzelne Komponenten eingebunden werden, sondern das Verhalten der gewünschten Schaltung beschrieben wird. V$_{HSIC}$HDL weist in der Syntax Ähnlichkeiten zu der Sprache Pascal auf.

2.8.2 Verilog

Verilog ist eine Hardwarebeschreibungssprache mit einer ähnlichen Funktionsweise wie V$_{HSIC}$HDL. Im Gegensatz zu V$_{HSIC}$HDL besitzt diese Sprache aber ähnliche Konstrukte wie C.

2.8.3 High-Level Synthesis

Im Gegensatz zu klassischen Hardwarebeschreibungssprachen wird bei der High-Level Synthesis (HLS) auf Programmiersprachen zurückgegriffen. Mit Hilfe von speziellen Bibliotheken kann dann die Funktionalität von Hardware in Form von Algorithmen definiert werden [35]. Allerdings gibt man bei Nutzung von HLS Kontrolle über die Ressourcen und die genaue Architektur ab. Ein Beispiel ist Xilinx Vivado HLS (C/C++).

2.9 Hardwaresimulation

Um die korrekte Verhaltensweise von modellierter Hardware zu garantieren wird eine Hardwaresimulation ausgeführt. Hierbei handelt es sich um eine Umgebung, bei der die Funktionalität der Hardware in Software rekonstruiert wird [37]. Ergebnisse können dann mittels Waveforms evaluiert werden, hierbei werden die Werte verschiedener Signale der Hardware in Abhängigkeit der Zeit angezeigt. Um Testfälle auszuwerten werden sogenannte Testbenches implementiert mit denen die Eingänge der zu testenden Komponente entsprechend der Anforderungen stimuliert werden.

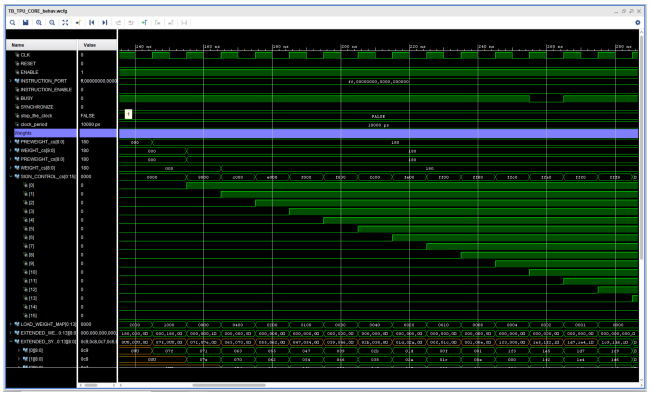


Abbildung 2.8: Auswertung einer Hardwarekomponente in der Simulation

2.10 Hardwaresynthese

Die Hardwaresynthese realisiert eine Umsetzung von definierter RTL-Logik zu einer konkreten Implementierung mit verfügbaren Hardwareressourcen [38]. Dies geschieht meistens in 2 übergreifenden Schritten (für Xilinx FPGAs).

2.10.1 Synthese

Die Synthese wendet Optimierungen an der RTL-Logik an und übersetzt diese in Hardwarekomponenten (LUTs, Flip-Flops, DSP und BRAM bei Xilinx FPGAs). Hierbei wird der Ort der Komponenten aber noch nicht festgelegt [38].

2.10.2 Implementation

Die Implementation adaptiert das synthetisierte Design an ein konkretes FPGA, hierbei werden hinsichtlich der Platzierung (Placing) und Verbindung (Routing) von Komponenten noch Optimierungen angewendet [36].

2.11 Stand der Technik

2.11.1 Tensor Processing Unit

Bei Tensor Processing Units (TPU) von Google handelt es sich um ML-Co-Prozessoren zur Verarbeitung von Inference-Modellen [15]. Im Kern der Architektur steht dabei eine Matrix Multiply Unit (MXU) für eine schnelle Berechnung der Netzeingaben, diese arbeitet parallel und kann bis zu 256 Eingaben gleichzeitig verrechnen. Für größere Matrixmultiplikationen als mit 256x256 Matrizen, stehen Akkumulatoren zur Verfügung um partielle Matrixmultiplikationen zu vereinen. Nach der Berechnung der Netzeingabe wird diese aus den Akkumulatoren in eine Aktivierungseinheit gegeben, welche (teils fest verdrahtete) Aktivierungsfunktionen ausführen kann.
Neben den Akkumulatoren gibt es weiteren Speicher für Gewichtungen und Ein-/Ausgaben der Netzschichten, wobei der Gewichtungsspeicher Off-Chip DDR3 RAM ist. Der Ein- und Ausgabespeicher ist ein On-Chip-Speicher und wird hier Unified Buffer genannt.
Für CNNs wird Normalisierung und Pooling unterstützt. Angeschlossen wird die TPU über eine PCI-Express Schnittstelle.

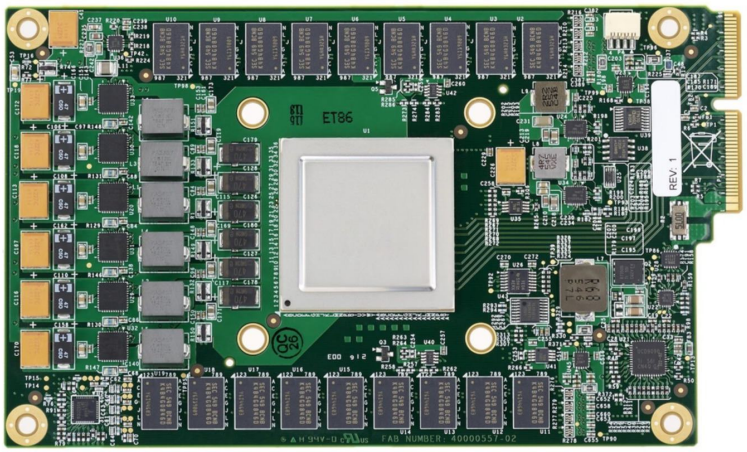


Abbildung 2.9: Erste Generation der Tensor Processing Units [15]

Während die erste Generation der TPUs nur Inferences mit 8/16-Bit signed und unsigned Integer unterstützt, können mit der 2. und 3. Generation Modelle mit Fließkomma Zahlen trainiert und evaluiert werden [7]. Hierzu sind jedoch keine Veröffentlichungen bekannt. Google nutzt diese TPUs innerhalb seiner Datenzentren für beschleunigtes Trainieren und Evaluieren. Bis dato sind diese Versionen nicht im freien Handel verfügbar.

2.11.2 Edge TPU

Edge TPU ist ein Application Specific Integrated Circuit (ASIC) von Google und ähnlich der ersten Generation der TPUs, so können diese 8/16-Bit Inference-Modelle berechnen. Welche Größe die MXU der Edge TPUs besitzt ist noch nicht bekannt. Diese Art der TPUs soll vor allem in Embedded Systems Anwendung finden [8].

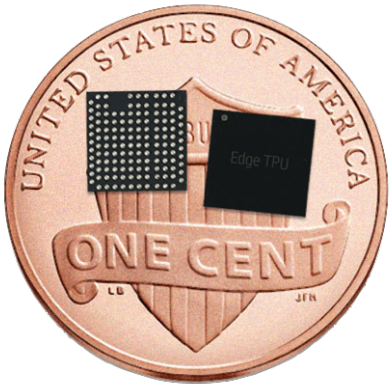


Abbildung 2.10: Edge TPU auf einem Penny [8]

2.11.3 OpenTPU

OpenTPU ist eine Open-Source Reimplementierung der TPU Architektur. Dabei wurde auf Festkommaarithmetik gesetzt. Die Hardwaremodellierung wurde in PyRTL (Bibliothek für Python) vorgenommen [31]. Mit PyRTL lässt sich Verilog Code generieren. Ob diese Hardwaremodellierung jemals auf einem FPGA zum Einsatz kam ist nicht bekannt und aufgrund von nicht abgeschlossenen Komponenten unwahrscheinlich.
Die Implementierung scheint in der Aktivierungsfunktion fehlerhaft zu sein, außerdem wird ein DDR-Speicherinterface verlangt (dieses ignoriert allerdings physikalische Umstände des Speichers, wie Latenzen). Eine optimierte Nutzung der Ressourcen eines FPGAs kann ebenfalls nicht garantiert werden, was die Umsetzung für Embedded Systems und IoT Geräte mit verhältnismäßig kleinen FPGAs ausschließt.

2.11.4 NVIDIA CUDA

Bei CUDA handelt es sich um eine Architektur von NVIDIA für parallele Berechnungen. Dabei werden die Grafikprozessoren von NVIDIA Grafikkarten verwendet [21]. Diese sind in der Regel in Desktop-Computern oder Servern zu finden, sind aber teilweise auch in mobilen Prozessoren/SoCs integriert, wie z.B. dem NVIDIA Tegra K1 oder X1.
Mit NVIDIA CUDA kann das Trainieren und Evaluieren von ML-Modellen beschleunigt werden.

2.11.5 NVIDIA Tensor Core

NVIDIA Tensor Core sind spezielle Kerne in NVIDIA Grafikkarten für die Beschleunigung von Matrixmultiplikationen und Faltungen [25]. Dabei sind diese schneller als NVIDIA CUDA Kerne, da weniger Overhead in der Ausführung besteht und mehr Daten parallel verarbeitet werden können. Ein Nachteil ist, dass diese Kerne nur für diese Operationen nutzbar sind.

2.11.6 NVIDIA Jetson

NVIDIA Jetson ist eine Plattform speziell für die Entwicklung und Nutzung von ML-Anwendungen [23]. Hierbei handelt es sich um Module und Entwicklerboards mit mobilen SoCs. Diese integrieren neben einem ARM-Prozessor, Grafikeinheiten mit CUDA Kernen. Es stehen verschiedene Ausführungen zur Verfügung, wie Jetson K1 (mit Tegra K1 SoC) und Jetson X1/X2. Hauptanwendungen sind Embedded Systems und mobile Systeme.

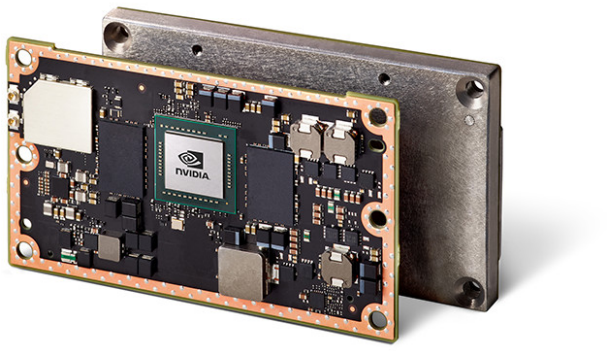


Abbildung 2.11: NVIDIA Jetson TX2 Modul [23]

2.11.7 NVDLA

NVIDIA Deep Learning Accelerator (NVDLA) ist eine Open-Source Implementierung eines ML-Co-Prozessors für Inferences [22]. Die Architektur weicht von der der TPU ab, besitzt aber ähnliche Komponenten. Dabei nutzt die Implementierung Off-Chip Speicher, welcher nicht unbedingt auf jedem Board/FPGA verfügbar ist.

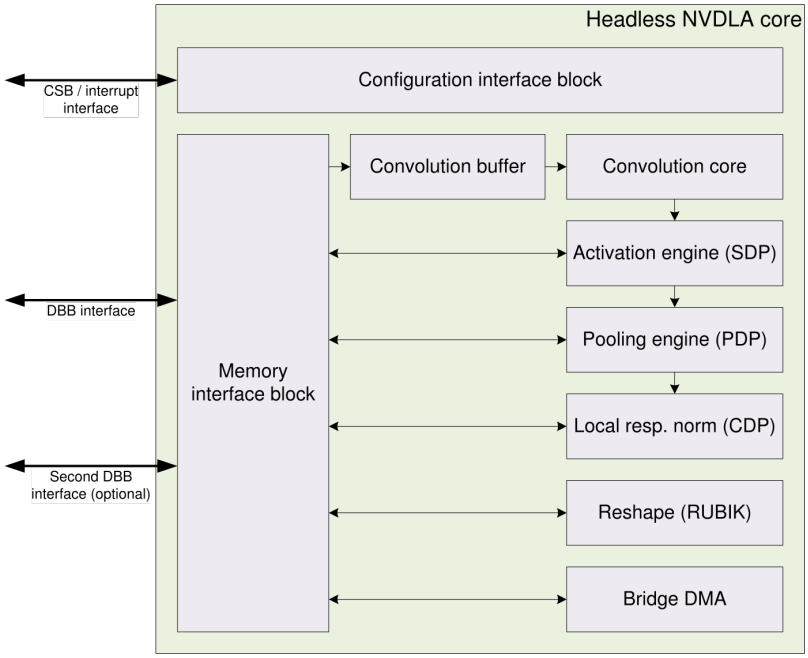


Abbildung 2.12: NVDLA Architektur [22]

2.11.8 NVIDIA Xavier

NVIDIA Xavier ist ein SoC speziell entwickelt für autonome Maschinen [24]. Dabei integriert dieser neben 8 ARM-Kernen auch 512 GPU Kerne mit Tensor Kernen und 2 NVDLA Co-Prozessoren. Xavier SoCs sind ebenfalls als Jetson Entwicklerboard verfügbar.

2.11.9 Intel Movidius Myriad

Intel Movidius Myriad ist eine Vision Processing Unit (VPU) [13]. Dabei handelt es sich um einen Microprocessor, der ML-Modelle beschleunigt. Die VPU integriert eine Neural Compute Engine, mit der DNN/CNN Applikationen berechnet werden können. Die Architektur erlaubt Berechnungen mit 16-Bit Fließkomma- und 8-Bit Festkommazahlen, wobei die Hardware energieeffizient arbeitet. Vor allem soll diese VPU Verwendung in der Bildverarbeitung finden. Neben der Hardware wird ein Software Development Kit (SDK) mitgeliefert um die Hardware optimal zu nutzen und bereits trainierte Netze zu portieren.

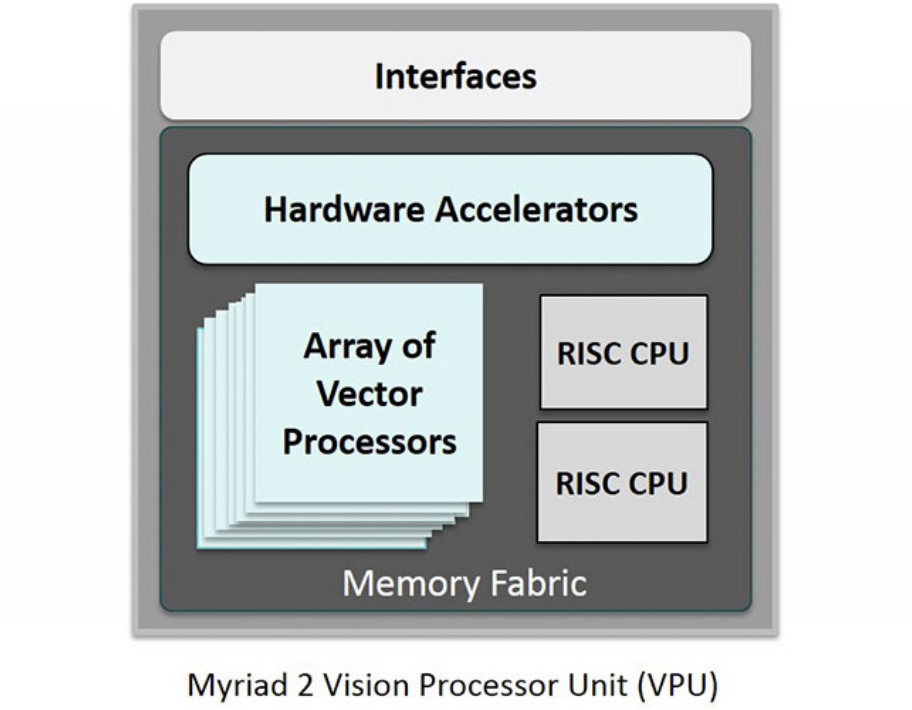


Abbildung 2.13: Angedeutete Architektur der Myriad 2 VPU [13]

3 Konzeption

Im Folgenden werden die Konzepte für die Realisierung einer TPU für Embedded Systems und IoT Geräte beschrieben, hierzu wird zunächst die Architektur der bestehenden TPUs erläutert und Überlegungen einer Adaptierung dargestellt. Zur Veranschaulichung werden Ausschnitte mit V$_{HSIC}$HDL-ähnlichem Pseudocode gezeigt.

3.1 Architektur der Tensor Processing Unit

Da es sich bei der Architektur der TPU um eine Co-Prozessor Architektur handelt besitzt diese keine Möglichkeit eines eigenständigen Programmablaufs, stattdessen werden Instruktionen von einem Host-System in einen FIFO-Speicher abgelegt. Das Design ähnelt einem Floating-Point-Co-Prozessor mehr als einem herkömmlichen Grafikprozessor[15].

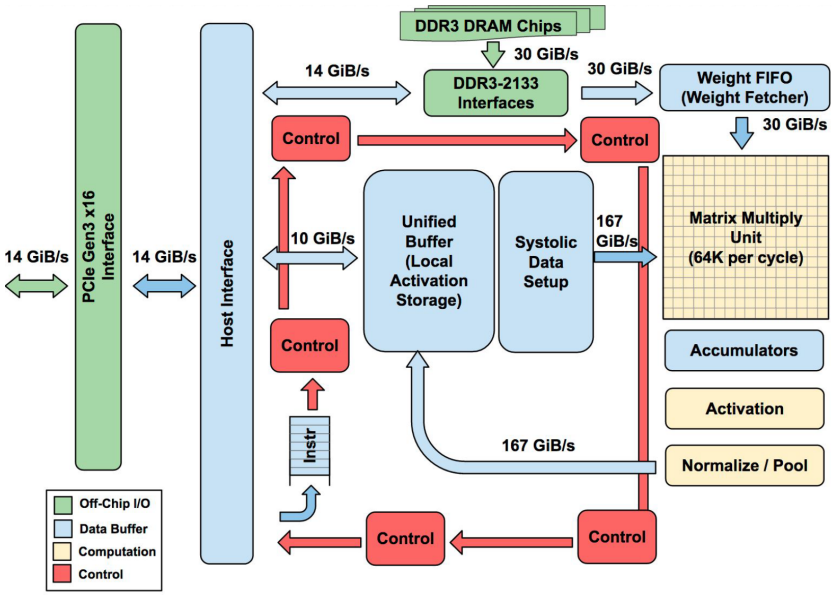


Abbildung 3.1: Architektur einer Tensor Processing Unit [15]

3.1.1 Quantisierung

Da die ursprüngliche Architektur der TPU 8-Bit/16-Bit Integer unterstützt ist eine Quantisierung der Netze notwendig, welche in der Regel mit Floating-Point trainiert werden. Es ist zu vermuten, dass die TPU ähnliche Quantisierungsmechanismen anwendet wie es bei TensorFlow (Lite) der Fall ist. Dabei wird der Exponent der Fließkommazahlen ersetzt mit einem »Scale« S und einem »Zero-Point« Z. Der reale (Fließkomma) Wert r kann dann aus dem quantisierten Wert q konstruiert werden mit [14]:

$$r = S(q - Z)$$

r und S sind Fließkommazahlen, q und Z hingegen unsigned Integer.
Elemente einer Matrix besitzen das selbe S und Z. Matrixmultiplikationen lassen sich dann wie folgt berechnen [14]:

$$\begin{aligned} S_3(q_3^{(i,k)} - Z_3) &= \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1) S_2(q_2^{(j,k)} - Z_2) \\ \Longrightarrow q_3^{(i,k)} &= Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2) \text{ mit } M \coloneqq \frac{S_1 S_2}{S_3} \end{aligned}$$

Das Verhältnis M ist der einzige Wert der keinen Integer-Typ darstellt. Dieser wird offline berechnet und liegt immer in dem Intervall $(0, 1)$ [14]. M kann als Festkommazahl aufgefasst und multipliziert werden. S und Z werden abhängig vom benötigten Wertebereich der Ein- und Ausgaben gewählt. Z_1 und Z_2 lassen sich mit der Substitution weiterer Matrizen aus der Matrixmultiplikation entfernen und als Vektoraddition auffassen.
Dieses Verfahren wird Integer-Arithmetic-Only Quantization genannt [14].

3.1.2 Unified Buffer

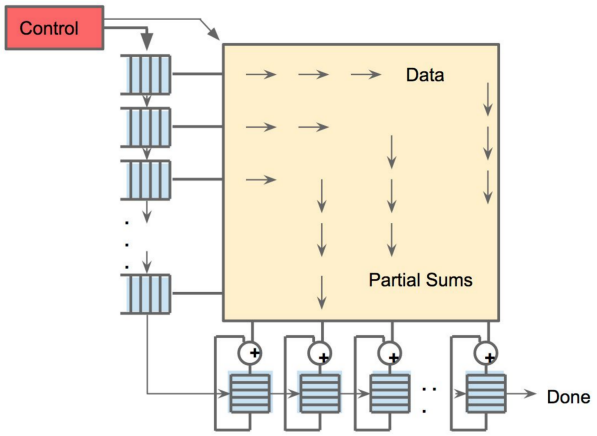
Der Unified Buffer ist ein On-Chip Speicher für Ein- und Ausgaben der Netze. Dabei besitzt dieser 3 Ports für die Übertragung von Daten vom/zum Host-System, lesen der Daten für Matrixmultiplikationen und schreiben von Daten nach der Aktivierung [15]. Die Busbreiten betragen 256 Byte.

3.1.3 Weight-Speicher/Weight-FIFO

Der Weight-Speicher ist ein Off-Chip DDR Speicher, um eine hohe Anzahl von Gewichten speichern zu können. Konkret kommen hier 8-GiB DDR3 Speicher zum Einsatz, womit sich eine hohe Anzahl von Netzen als auch tiefe Netze speichern lassen. Das Speicherinterface kann vom Host-System und einem Weight-First In First Out (FIFO) angesprochen werden. Dabei puffert der Weight-FIFO die eingehenden Daten des Speicherinterfaces und legt diese an einen 256 Byte breiten Bus an [15].

3.1.4 Matrix Multiply Unit

Die MXU ist ein systolisches Array von Multiply-Add Komponenten [15]. Diese besitzen 3 Eingänge für ein Gewicht, eine Eingabe und Akkumulation. Gewichte besitzen 2 Register, damit der Weight-FIFO bereits neue Gewichte laden kann während schon eine Matrixmultiplikation stattfindet. Eingaben werden abgetaktet und mit den geladenen Gewichten multipliziert. Das resultierende 16-Bit Ergebnis wird in einem Pipeline-Register gespeichert. Der Ausgang des Registers wird dann mit dem Akkumulationseingang addiert und in ein Ergebnis-Register abgelegt, wobei durch die Addition eine um 1-Bit größere Busbreite als der größte Bus entsteht.
Eine Reihe mit der Zeilenzahl 256 dieser Multiply-Add Einheiten werden konkateniert, wobei der Ausgang, die partielle Summe, mit dem Akkumulationseingang der nächsten Einheit verbunden wird. Die komplette Einheit wird von 256 paralleler Reihen gebildet. Gewichte werden zeilenweise vorgeladen. Busbreiten der Ein- und Ausgänge liegen bei 256 Byte. Die Größe der Ausgänge wird auf 32-Bit getrimmt.



Die Eingänge werden von dem Systolic Data Setup getrieben und die Ergebnisse in den Akkumulatoren gespeichert.

Abbildung 3.2: Datenfluss der partiellen Summen in der MXU [15]

3.1.5 Systolic Data Setup

Aufgrund der Pipeline-Register der MXU müssen die Eingaben des Unified Buffers »diagonal« eingespeist werden, sodass die Eingänge jeder Zeile um einen Takt zur vorherigen Zeile verzögert werden, hierdurch entsteht eine Latenz von 256 Taktzyklen [15].

3.1.6 Akkumulatoren

Die Akkumulatoren sind ein Speicher mit der Möglichkeit ein Ergebnis an eine Adresse zu überschreiben oder zu akkumulieren. Daten der zu schreibenden Adresse werden dabei gelesen, mit den neuen Daten durch einen Addierer geschleust und an dieselbe Adresse gespeichert. Die Größe der Akkumulatoren beträgt dabei 32-Bit [15].

3.1.7 Aktivierung

Die Aktivierungs-Pipeline kann verschiedene Aktivierungsfunktionen mit den Ergebnissen in den Akkumulatoren ausführen [15]. Resultate können dann in dem Unified Buffer abgelegt werden.

3.1.8 Pooling

Neben der Aktivierung ist eine Pooling-Pipeline vorhanden, diese kann optional Operationen, wie das Max-Pooling, auf aktivierte Ausgaben ausführen [15].

3.1.9 Instruktionssatz

Aufgrund der unterschiedlichen Ausführungsweisen der Komponenten besitzt die TPU einen Complex Instruction Set Computer (CISC) Instruktionssatz. Dabei gibt es keine klassische Reduced Instruction Set Computer (RISC) Pipeline und Instruktionen sprechen unterschiedliche Komponenten an, wobei die Ausführungsdauer variiert [15]. Um Ausführungen zu wiederholen ist ein Repeat-Feld vorhanden.
Es gibt insgesamt 12 Instruktionen, wobei 5 für die Ausführung von Netzen benötigt werden [15].

Read_Host_Memory

Read_Host_Memory liest per Direct Memory Access (DMA) Daten über das Host-Interface und speichert diese in den Unified Buffer [15].

Read_Weights

Read_Weights liest Gewichte aus dem Weight-Speicher und lädt diese über den Weight-FIFO in die MXU [15].

MatrixMultiply/Convolve

MatrixMultiply/Convolve führt mit den geladenen Gewichten und Daten aus dem Unified Buffer eine Matrixmultiplikation oder eine Faltung aus. Dabei können $B \cdot 256$ Eingaben mit 256×256 Gewichten verrechnet werden, wobei $B \cdot 256$ (partielle) Summen in den Akkumulatoren gespeichert werden [15].

Activate

Activate führt eine nicht-lineare Funktion auf die Netzeingaben aus, dabei dienen die Akkumulatoren als Netzeingabe. Ausgaben werden dann wieder im Unified Buffer gespeichert. Pooling kann ebenfalls durch Activate ausgeführt werden [15].

Write_Host_Memory

Write_Host_Memory schreibt Daten aus dem Unified Buffer in den Host-Speicher [15].

Weitere Instruktionen

Es gibt noch weitere Instruktionen, dazu gehören alternate host memory read/write, set configuration, 2 Typen von synchronization, interrupt host, debug-tag, nop und halt [15].

3.1.10 Steuerwerke

Die Steuerwerke verarbeiten die Instruktionen und steuern die dazu benötigten Hardwarekomponenten. Instruktionen werden per FIFO-Speicher vom Host-System eingespeist [15].

3.1.11 Host-Interface

Das Host-Interface adaptiert die PCI-Express Schnittstelle an die verschiedenen Speicherschnittstellen (Instruktions-FIFO, Weight-Speicher-Interface, Unified Buffer). Zugriffe per DMA werden unterstützt [15].

3.2 Adaptierung für Embedded Systems und IoT Geräte

Dieser Abschnitt beschäftigt sich mit dem eigentlichen Konzept dieser Arbeit, dabei wird das Konzept der TPU aufgegriffen und auf die Anforderungen von Embedded Systems und IoT Geräte zugeschnitten.

3.2.1 Quantisierung

Um Berechnungen mit Floating-Point zu vermeiden wird die Quantisierung im Gegensatz zur TPU nicht mittels Integer-Arithmetic-Only Quantization vorgenommen sondern in Festkommaarithmetik. 8-Bit Eingaben und Gewichte haben dann das Format Q8 oder Qu8. Die Wertebereiche liegen dann bei:

$$W_{Q8} = \{\frac{x}{128} | x \in \mathbb{N}, -128 \leq x \leq 127\}$$
$$W_{Qu8} = \{\frac{x}{256} | x \in \mathbb{N}, 0 \leq x \leq 255\}$$

3.2.2 Unified Buffer als BlockRAM

Der Unified Buffer wird so beschrieben, dass dieser einem konkatenierten BRAM genügt. Die Synthese kann die Logik dann mit BRAM Komponenten ersetzen. Da auf den meisten FPGAs kein BRAM mit 3 Ports unterstützt wird, müssen die beiden verfügbaren Ports so vermuxt werden, dass ein Zugriff vom Host (Master-Port) die Zugriffe der anderen Ports überschreibt. Der Host kann nicht unbedingt alle Daten gleichzeitig schreiben (z.B. 32-Bit Zugriffe), weshalb der Master-Port Write-Strobes besitzt. Die Architektur des Unified Buffers kann wie folgt definiert werden:

```
architecture BEH of UNIFIED_BUFFER is
  ⋮
  type RAM_TYPE is array(0 to TILE_WIDTH-1, 0 to MATRIX_WIDTH-1)
      of BYTE_TYPE;
  shared variable RAM  : RAM_TYPE;
begin
  ⋮
  OVERRIDE:
  process(MASTER_EN, EN0, EN1, MASTER_ADDRESS, ADDRESS0,
    ADDRESS1) is
  begin
    if MASTER_EN = '1' then
      EN0_OVERRIDE <= MASTER_EN;
      EN1_OVERRIDE <= MASTER_EN;
      ADDRESS0_OVERRIDE <= MASTER_ADDRESS;
      ADDRESS1_OVERRIDE <= MASTER_ADDRESS;
    else
      EN0_OVERRIDE <= EN0;
      EN1_OVERRIDE <= EN1;
      ADDRESS0_OVERRIDE <= ADDRESS0;
      ADDRESS1_OVERRIDE <= ADDRESS1;
    end if;
  end process OVERRIDE;
  ⋮
```

Abbildung 3.3: Deklaration des Speichers für den Unified Buffer und Vermuxung der Ports für den Master-Zugriff

```
  ⋮
  PORT0:
  process(CLK) is
  begin
    if CLK'event and CLK = '1' then
      if EN0_OVERRIDE = '1' then
        for i in 0 to MATRIX_WIDTH-1 loop
          -- Loop through all strobes
          if MASTER_WRITE_EN(i) = '1' then
            RAM(ADDRESS0_OVERRIDE)(i) := MASTER_WRITE_PORT(i);
          end if;
        end loop;
          READ_PORT0 <= RAM(ADDRESS0_OVERRIDE);
      end if;
    end if;
  end process PORT0;

  PORT1:
  process(CLK) is
  begin
    if CLK'event and CLK = '1' then
      if EN1_OVERRIDE = '1' then
        if WRITE_EN1 = '1' then
          RAM(ADDRESS1_OVERRIDE) := WRITE_PORT1;
        end if;
        MASTER_READ_PORT <= RAM(ADDRESS1_OVERRIDE);
      end if;
    end if;
  end process PORT1;
  ⋮
end architecture BEH;
```

Abbildung 3.4: Port0 und Port1 des Unified Buffers

Für Matrixmultiplikationen wird Port0 gelesen. Ergebnisse der Aktivierung werden über Port1 geschrieben. Wenn der Master-Port aktiv ist werden Adressen von Port0 und Port1 ignoriert. Die Größe des Speichers und der Busbreiten wird per Generic angegeben. Ein Ausschnitt eines Blockdiagramms des interpretierten V_{HSIC}HDL Codes der RTL-Analyse ist in Anhang A.2 zu sehen.

3.2.3 Weight Buffer als BlockRAM

Anstatt eines externen Speichers wird auch für die Gewichtungen BRAM verwendet. Einerseits wird der Durchsatz durch Vermeidung eines Off-Chip Speichers erhöht, da BRAM synchron arbeitet, außerdem besitzen nicht alle FPGAs/Boards einen Off-Chip Speicher bzw. -Interface.
Dieser Weight Buffer kann ähnlich definiert werden wie der Unified Buffer. Hierbei kann aber auf den Master-Port verzichtet werden, da nur 2 parallele Ports (Schreiben des Host-Systems und Lesen für Matrixmultiplikation) benötigt werden.

3.2.4 Matrix Multiply Unit

Multiply-Add/Multiply-Accumulate als DSP-Blöcke

Die MXU wird nach dem bereits genannten Konzept der TPU realisiert. Die hierfür benötigten Multiply-Add Einheiten sehen wie folgt aus:

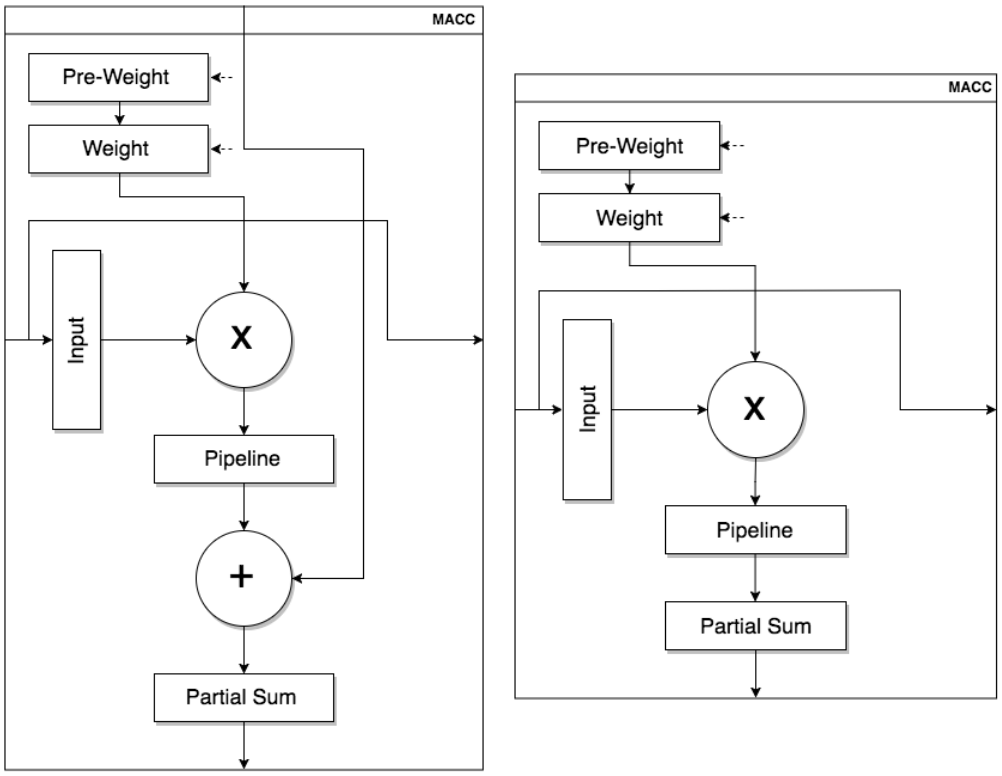


Abbildung 3.5: Blockdiagramm der Multiply-Add Einheit mit und ohne Addierer

Da die erste Zeile der Multiply-Add Einheiten keine Addition ausführen muss, ist der Addierer optional. Dies kann über Generics bestimmt werden, die die Busbreiten der Ein- und Ausgänge bestimmen. Demnach gibt es 3 Fälle die unterschieden werden müssen:

1. Einheit ohne Summen-Eingang und Addierer.
2. Standard Einheit mit Summen-Eingang und Addierer, die Busbreite des Ausgangs wird um 1-Bit erhöht.
3. Einheit mit Summen-Eingang und Addierer, der Ausgang ist auf 32-Bit begrenzt.

Aufgrund des Aufbaus der Einheiten können diese von der Synthese als DSP-Blöcke eingebunden werden. Eine Beschreibung ist auf Abbildung 3.6 zu sehen.

```
architecture BEH of MACC is
  ⋮
begin
  ⋮
  MUL_ADD:
  process(INPUT_cs, WEIGHT_cs, PIPELINE_cs, LAST_SUM) is
  begin
    PIPELINE_ns <= std_logic_vector(signed(INPUT) * signed(WEIGHT));

    -- Only ONE case will get synthesized!
    if LAST_SUM_WIDTH > 0 and LAST_SUM_WIDTH < PARTIAL_SUM_WIDTH then
      -- Concatination needed for PARTIAL_SUM to be 1 bit wider
      PARTIAL_SUM <= std_logic_vector(
        signed(PIPELINE_cs(PIPELINE_cs'HIGH) & PIPELINE_cs) +
        signed(LAST_SUM(LAST_SUM'HIGH) & LAST_SUM)
      );
    elsif LAST_SUM_WIDTH > 0 and LAST_SUM_WIDTH=PARTIAL_SUM_WIDTH then
      -- Same bit width
      PARTIAL_SUM <= std_logic_vector(
        signed(PIPELINE_cs) +
        signed(LAST_SUM)
      );
    else -- LAST_SUM_WIDTH = 0 --> no LAST_SUM input
      PARTIAL_SUM <= PIPELINE_cs;
    end if;
  end process MUL_ADD;
  ⋮
end architecture BEH;
```

Abbildung 3.6: Beschreibung der adaptierbaren Multiply-Add Einheit

Da signed und unsigned Integer unterstützt werden sollen, müssen, um dieselbe Arithmetik und somit Hardware zu nutzen, die Gewichte und Eingaben um 1-Bit erweitert werden. Die 8-Bit Werte werden dann, je nachdem ob signed oder unsigned gerechnet werden soll, auf 9-Bit sign oder zero extended. Die Hardware rechnet dann immer signed. Ein Ausschnitt eines Blockdiagramms der RTL-Analyse ist in Anhang A.3 zu sehen.

Aufbau

Die MXU besteht aus einem 2D Array der Multiply-Add Komponenten, dabei handelt es sich immer um eine $n \times n$ Matrix. Die erste Zeile besteht aus Komponenten ohne Summen-Eingang. Ist die Matrix kleiner als 16×16, so müssen die Ausgaben auf 32-Bit sign oder zero extended werden, andernfalls werden diese auf 32-Bit getrimmt.
Gewichte können vorgeladen werden, wobei immer eine komplette Zeile der Einheiten gleichzeitig vorgeladen wird. Hierfür sind Zeilen adressierbar. Sollen Gewichte durch eine Matrixmultiplikation verrechnet werden, so müssen diese mit Eintreffen der Eingaben in das Weight-Register geladen werden. Hierzu wird nur ein activate Signal benötigt. Dieses wird $n - 1$ Takte verzögert. Jedes Register der Verzögerung wird dann mit den Zeilen der Matrix verbunden, sodass mit jedem Takt immer die nächste Zeile geladen wird. Die erste Zeile wird mit dem direkt anliegenden activate Signal verbunden.

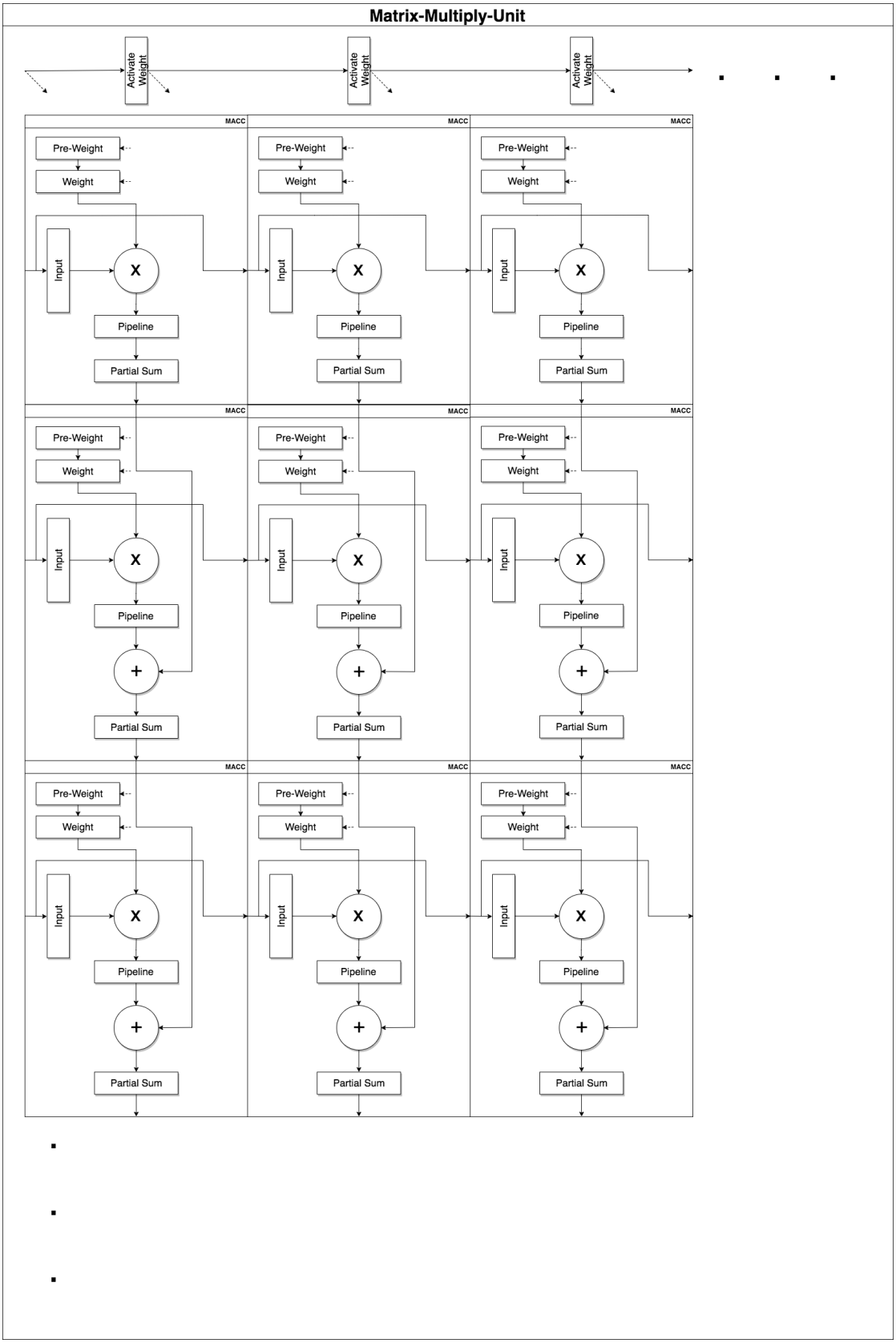


Abbildung 3.7: Blockdiagramm der Matrix Multiply Unit

3.2.5 Systolic Data Setup

Um die Daten des Unified Buffers »diagonal« in die MXU zu geben wird jede Zeile durch das Systolic Data Setup (SDS) verzögert. Dies wird durch Register realisiert, die sequentiell angeordnet sind. Die erste Zeile muss nicht verzögert werden. Abbildung 3.8 zeigt den Aufbau der SDS Komponente. Ein Ausschnitt eines Blockdiagramms der RTL-Analyse ist in Anhang A.4 zu sehen.

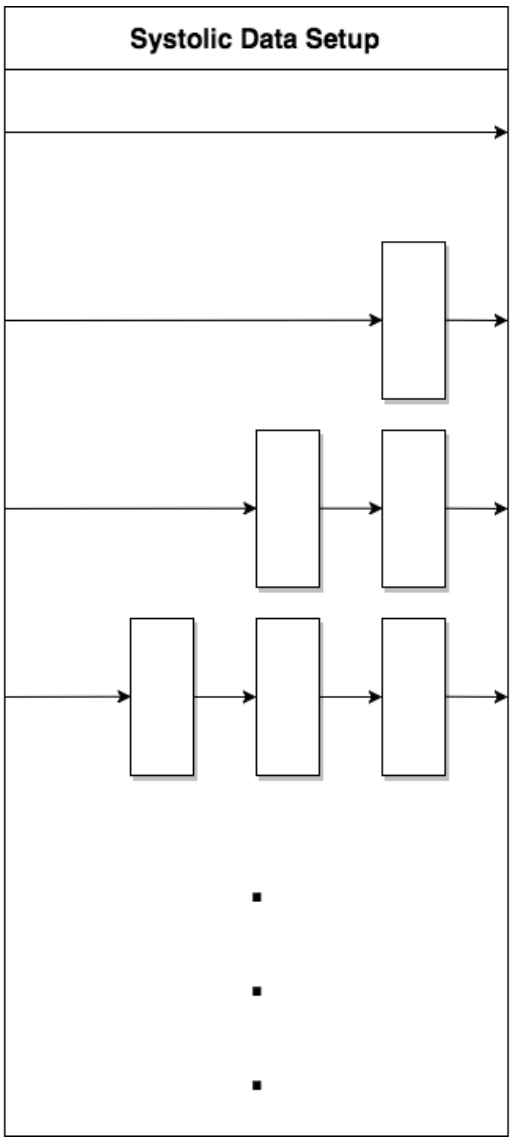
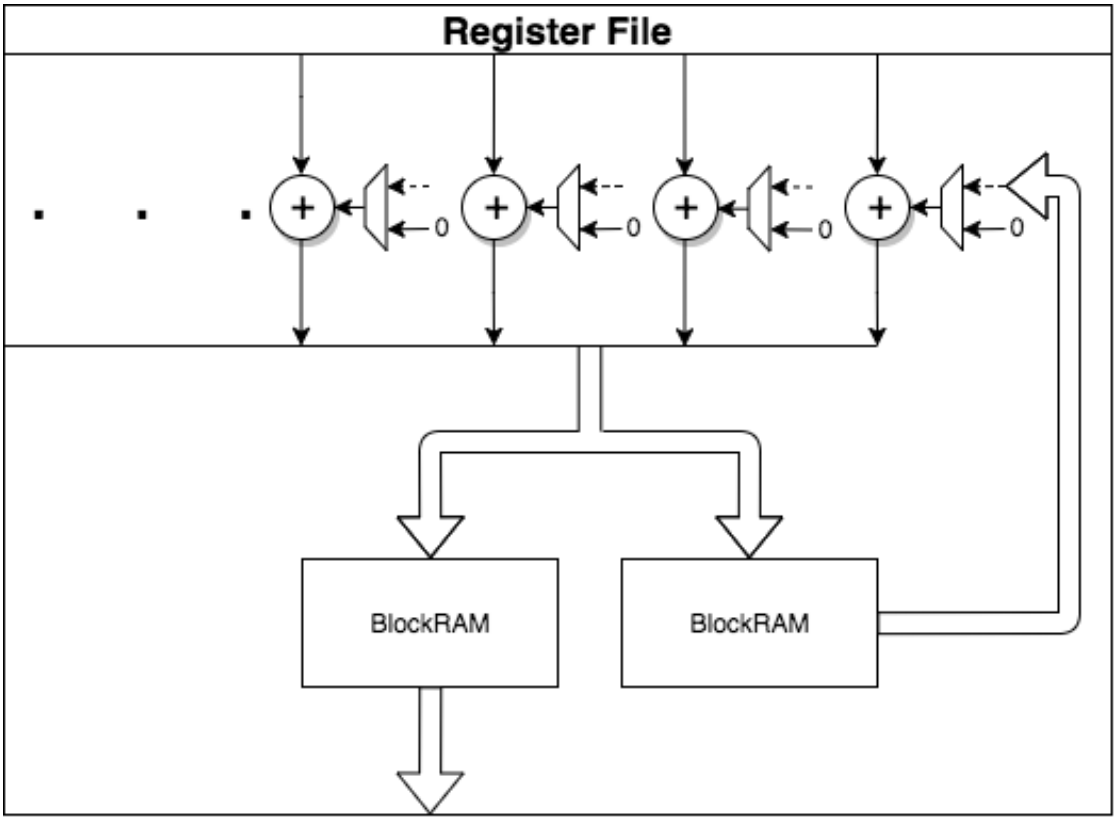


Abbildung 3.8: Aufbau des Systolic Data Setups

3.2.6 Akkumulatoren als BlockRAM und DSP-Blöcke

Für die Unterstützung von Matrixmultiplikationen mit Matrizen größer als $n \times n$ werden Akkumulatoren bereit gestellt. Dabei handelt es sich um Register, dessen Wert überschrieben oder aufsummiert werden kann. Da in dieser Architektur eine höhere Anzahl von Registern vorteilhaft ist, wird hier auf BRAM gesetzt. Für die Addierer werden DSP-Blöcke verwendet. Der Eingang wird dann über den DSP-Block entweder mit 0 oder mit dem Inhalt des BRAMs an derselben Adresse addiert. Das Ergebnis wird dann an die übergebene Adresse gespeichert. Da hierfür schon 2 Ports (gleichzeitig schreiben und lesen) benötigt werden, muss für den Ausgang der Akkumulatoren der Speicher redundant gehalten werden, so ist es möglich einen weiteren Port zu akquirieren.



Mögliche Pipeline Register und der Adress-Eingang werden nicht dargestellt.

Abbildung 3.9: Blockdiagramm der Akkumulator Register

Eine mögliche Umsetzung des gezeigten Blockdiagramms wird in Abbildung 3.10 und 3.11 gezeigt.

```
architecture BEH of REGISTER_FILE is
  ⋮
begin
  ⋮
  DSP_ADD:
  process(DSP_ADD_PORT0_cs, DSP_ADD_PORT1_cs) is
  begin
    for i in 0 to MATRIX_WIDTH-1 loop
      DSP_RESULT_PORT_ns(i) <= std_logic_vector(
        unsigned(DSP_ADD_PORT0_cs(i)) +
        unsigned(DSP_ADD_PORT1_cs(i))
      );
    end loop;
  end process DSP_ADD;

  ACC_MUX:
  process(ACCUMULATE_PORT, ACCUMULATE) is
  begin
    if ACCUMULATE= '1' then -- Input should be accumulated
      DSP_ADD_PORT1_ns <= ACCUMULATE_PORT;
    else
      DSP_ADD_PORT1_ns <= (others => (others => '0'));
    end if;
  end process ACC_MUX;
  ⋮
```

Pipeline Register werden hier ignoriert.

Abbildung 3.10: Beschreibung der DSP-Blöcke und der Vermuxung des Addierer-Einganges

```
  ⋮
  ACC_PORT0:
  process(CLK) is
  begin
    if CLK'event and CLK = '1' then
      if ENABLE = '1' then
        if ACC_WRITE_EN = '1' then
          ACCUMULATORS(ACC_WRITE_ADDRESS) := DSP_RESULT_PORT_cs;
          ACCUMULATORS_COPY(ACC_WRITE_ADDRESS) := DSP_RESULT_PORT_cs;
        end if;
      end if;
    end if;
  end process ACC_PORT0;

  ACC_PORT1:
  process(CLK) is
  begin
    if CLK'event and CLK = '1' then
      if ENABLE = '1' then
        ACC_READ_PORT <= ACCUMULATORS(ACC_READ_ADDRESS);
        ACCUMULATE_PORT <= ACCUMULATORS_COPY(ACC_ACCU_ADDRESS);
      end if;
    end if;
  end process ACC_PORT1;
  ⋮
end architecture BEH;
```

Pipeline Register werden hier ignoriert.

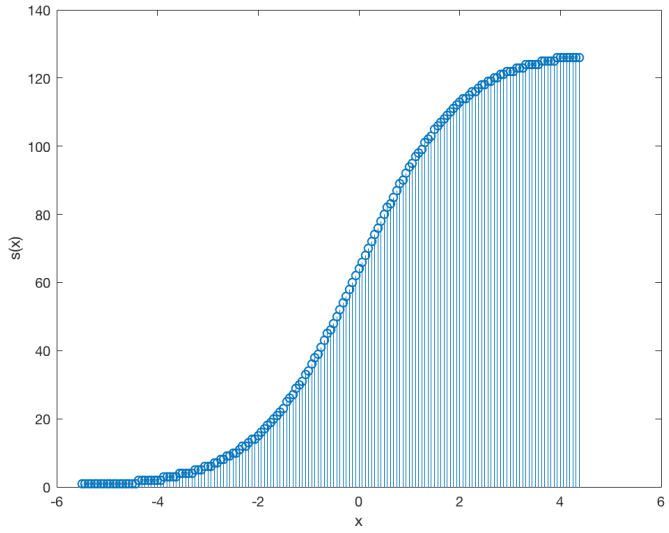
Abbildung 3.11: Beschreibung der redundanten Akkumulator Register

3.2.7 Aktivierung

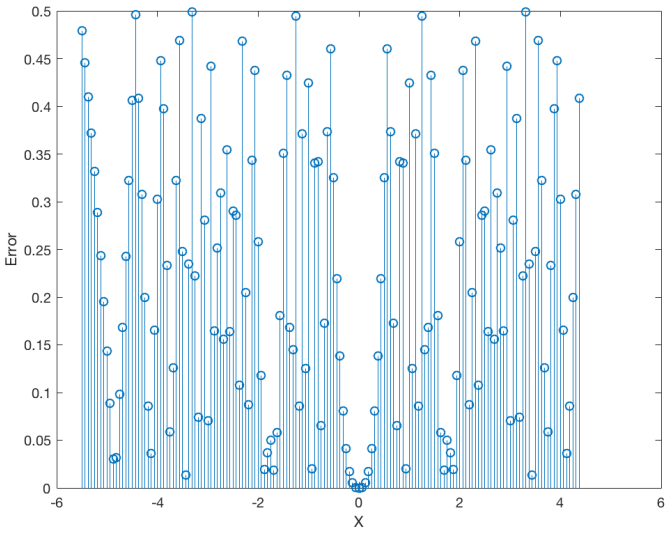
Im Rahmen dieser Arbeit werden 2 Aktivierungsfunktionen umgesetzt, Sigmoid und ReLU Funktion.

Sigmoid als Lookup-Tabelle/ROM

Die Sigmoid Funktion liegt (fast) in demselben Wertebereich wie die Typen der Berechnungen, daher muss diese nicht modifiziert werden. Für die Umsetzung wird die Funktion als Lookup-Tabelle aufgestellt, hierbei kann die Lösung der Funktion für die Eingabe ausgelesen werden. Ein Vorteil ist, dass der Wert nicht berechnet werden muss. Um allerdings eine gleichwertige Genauigkeit zu gewährleisten werden entsprechend viele Ressourcen benötigt. Um Ressourcen zu sparen wird auf die hohe Genauigkeit verzichtet und die Ausgabe der Funktion quantisiert.
Die beiden Tabellen (für signed und unsigned) werden auf 8-Bit Adressen begrenzt. Die akkumulierten Eingaben besitzen eine Busbreite von 32-Bit und sind dementsprechend formatiert als $Q16.16$ oder $Qu16.16$.
Die Tabelle für signed besitzt das Format $Q4.4$ mit der minimalen Eingabe von -5.5 und der maximalen Eingabe von 4.375. Alle kleineren Werte liegen bei 0 und alle größeren bei $\frac{127}{128}$. Hier wird ein Fehler akzeptiert, da es nicht möglich ist 1 abzubilden. Die Quantisierung für signed Sigmoid ist auf Abbildung 3.12a zu sehen, wobei mit einem Faktor von 128 gearbeitet wird. Der Quantisierungsfehler, also die Abweichung der quantisierten Funktion zur ursprünglichen Funktion, wird in Abbildung 3.12b gezeigt.



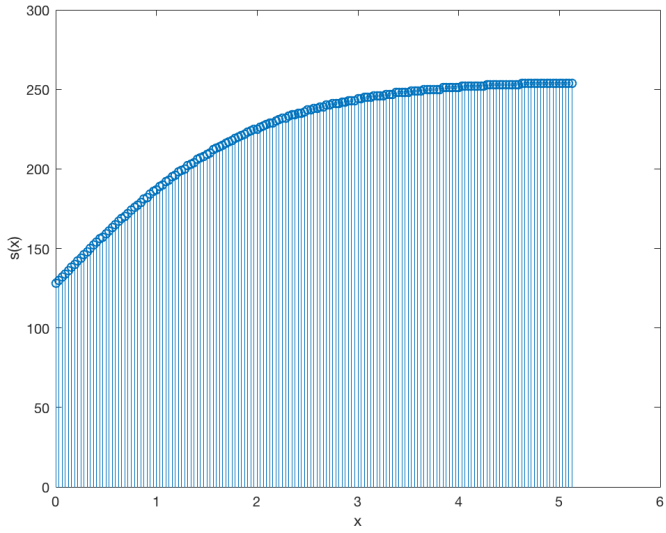
(a) Quantisierte Sigmoid Funktion für signed Integer



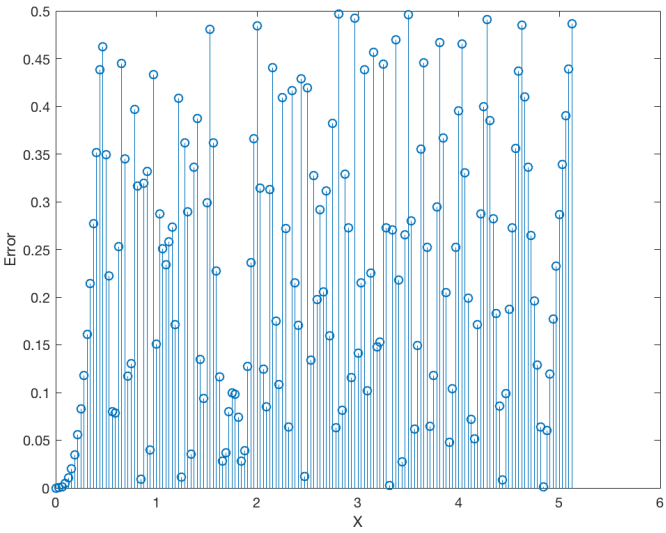
(b) Quantisierungsfehler der Sigmoid Funktion für signed Integer

Abbildung 3.12: Visualisierung der Lookup-Tabelle für signed Integer

Für die Tabelle für unsigned Integer kann der Typ $Qu3.5$ gewählt werden. Hier beginnt die Tabelle bei 0 und endet bei 5.125. Alle weiteren Werte liegen bei $\frac{255}{256}$. 1 kann nicht abgebildet werden. Abbildung 3.13a zeigt die Quantisierung, wobei der Quantisierungsfehler auf Abbildung 3.13b zu sehen ist. Hierbei wurde mit einem Faktor von 256 gearbeitet.



(a) Quantisierte Sigmoid Funktion für unsigned Integer



(b) Quantisierungsfehler der Sigmoid Funktion für unsigned Integer

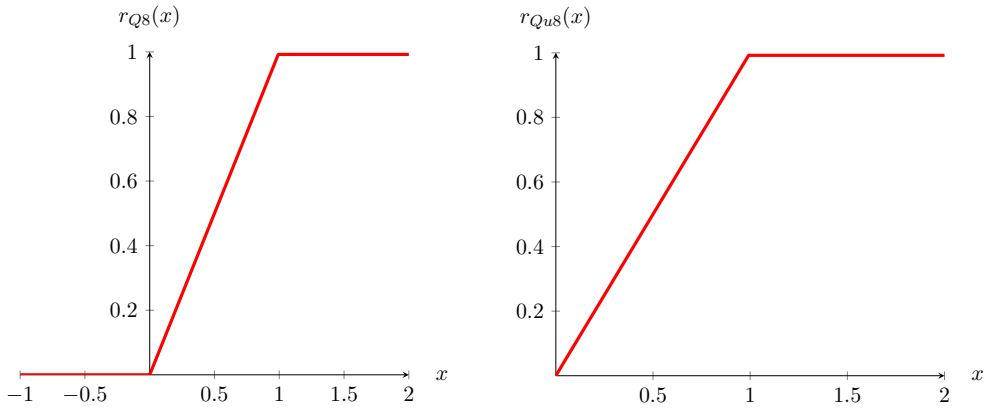
Abbildung 3.13: Visualisierung der Lookup-Tabelle für unsigned Integer

Die Fraction der Eingaben der Tabellen werden von $Q16/Qu16$ auf $Q4/Qu5$ gerundet, wodurch die Eingabe zusätzlich quantisiert wird.

ReLU mit oberer Begrenzung

Da die ReLU Funktion im Gegensatz zur Sigmoid Funktion nicht konvergiert muss diese begrenzt werden, damit der Wertebereich den Datentypen nicht übersteigt. Hierfür wird ein Maximalwert von $\frac{127}{128}$ bzw. $\frac{255}{256}$ verwendet - dem Maximalwert der Datentypen. Somit handelt es sich hierbei um eine stückweise-lineare Funktion und wird beschrieben durch:

$$r_{Q8}(x) = min(max(0, x), \frac{127}{128})$$
$$r_{Qu8}(x) = min(x, \frac{255}{256})$$



(a) Signed ReLU Funktion mit dem Maximalwert $\frac{127}{128}$

(b) Unsigned ReLU Funktion mit dem Maximalwert $\frac{255}{256}$

Abbildung 3.14: ReLU Funktion mit Begrenzungen

Da die Eingabe im Format $Q16.16/Qu16.16$ vorliegt, die Ausgabe aber wieder innerhalb von $Q8/Qu8$ liegen muss, werden auch hier die Eingaben (im linearen Teil gleich der Ausgabe) quantisiert.

Beschreibung der Aktivierung

Die Deklaration der Lookup-Tabellen und die Quantisierung der Eingaben wird in Abbildung 3.15 umgesetzt. Die ReLU und Sigmoid Aktivierungsfunktionen sind in Abbildung 3.16 und 3.17 zu sehen.

```
architecture BEH of ACTIVATION is
  -- Values of the lookup tables
  -- 0 to 164 --> in Qu3.5: 0 to 5.125
  constant SIGMOID_UNSIGNED : INTEGER_ARRAY_TYPE(0 to 164) := (...);
  -- -88 to 70 --> in Q4.4: -5.5 to 4.375
  constant SIGMOID_SIGNED : INTEGER_ARRAY_TYPE(-88 to 70) := (...);
  ⋮
begin
  ROUND:
  process(INPUT_REG_cs, SIGNED_NOT_UNSIGNED_REG_cs(0)) is
  begin
    -- input quantization adds highest non-mappable fractional bit
    for i in 0 to MATRIX_WIDTH-1 loop
      -- ReLU quantization - Q(u)16.8 input range
      RELU_ROUND_REG_ns(i) <=
        INPUT_REG_cs(i)(4*BYTE_WIDTH-1 downto 1*BYTE_WIDTH) +
        INPUT_REG_cs(i)(1*BYTE_WIDTH-1);
      -- Sigmoid quantization
      if SIGNED_NOT_UNSIGNED_REG_cs(0) = '0' then
        -- unsigned - Qu16.5 input range
        SIGMOID_ROUND_REG_ns(i) <=
          INPUT_REG_cs(i)(4*BYTE_WIDTH-1 downto 2*BYTE_WIDTH-5) +
          INPUT_REG_cs(i)(2*BYTE_WIDTH-6);
      else
        -- signed - Q16.4 input range
        SIGMOID_ROUND_REG_ns(i) <= (
          INPUT_REG_cs(i)(4*BYTE_WIDTH-1 downto 2*BYTE_WIDTH-4) +
          INPUT_REG_cs(i)(2*BYTE_WIDTH-5)
        ) & '0'; -- same register like unsigned, needs one bit more
      end if;
    end loop;
  end process ROUND;
  ⋮
```

Abbildung 3.15: Deklaration der Lookup-Tabellen und Ausführung der Eingabe-Quantisierung

```
⋮
RELU_ACTIVATION:
process(SIGNED_NOT_UNSIGNED_REG_cs(1), RELU_ROUND_REG_cs) is
  variable SIGNED_NOT_UNSIGNED_v : std_logic;
  variable RELU_ROUND_v : RELU_ARRAY_TYPE(0 to MATRIX_WIDTH-1);
begin
SIGNED_NOT_UNSIGNED_v := SIGNED_NOT_UNSIGNED_REG_cs(1);
RELU_ROUND_v := RELU_ROUND_REG_cs;
  for i in 0 to MATRIX_WIDTH-1 loop
    if SIGNED_NOT_UNSIGNED_v = '1' then
      if signed(RELU_ROUND_v(i)) < 0 then
        RELU_OUTPUT(i) <= (others => '0');
      elsif signed(RELU_ROUND_v(i)) > 127 then -- Bounded ReLU
        RELU_OUTPUT(i) <= x"7F";
      else
        RELU_OUTPUT(i) <= RELU_ROUND_v(i)(BYTE_WIDTH-1 downto 0);
      end if;
    else
      if unsigned(RELU_ROUND_v(i)) > 255 then -- Bounded ReLU
        RELU_OUTPUT(i) <= x"FF";
      else
        RELU_OUTPUT(i) <= RELU_ROUND_v(i)(BYTE_WIDTH-1 downto 0);
      end if;
    end if;
  end loop;
end process RELU_ACTIVATION;
⋮
```

Abbildung 3.16: Ausführung der ReLU Funktion mit Maximalwert

```
  ⋮
  SIGMOID_ACTIVATION:
  process(SIGNED_NOT_UNSIGNED_REG_cs(1), SIGMOID_ROUND_REG_cs) is
    variable SIGNED_NOT_UNSIGNED_v : std_logic;
    variable SIGMOID_ROUND_v: SIGMOID_ARRAY_TYPE(0 to MATRIX_WIDTH-1);
  begin
    SIGNED_NOT_UNSIGNED_v := SIGNED_NOT_UNSIGNED_REG_cs(1);
    SIGMOID_ROUND_v := SIGMOID_ROUND_REG_cs;
    for i in 0 to MATRIX_WIDTH-1 loop
      if SIGNED_NOT_UNSIGNED_v = '1' then -- Signed
        if signed(SIGMOID_ROUND_v(i)(20 downto 1)) < -88 then
          SIGMOID_OUTPUT(i) := x"00";
        elsif signed(SIGMOID_ROUND_v(i)(20 downto 1)) > 70 then
          SIGMOID_OUTPUT(i) := x"7F";
        else
          -- One bit less for signed
          SIGMOID_OUTPUT(i) := SIGMOID_SIGNED(
            SIGMOID_ROUND_v(i)(20 downto 1)
          );
        end if;
      else -- Unsigned
        if unsigned(SIGMOID_ROUND(i)) > 164 then
          SIGMOID_OUTPUT(i) := x"FF";
        else
          SIGMOID_OUTPUT(i) := SIGMOID_UNSIGNED(SIGMOID_ROUND_v(i));
        end if;
      end if;
    end loop;
  end process SIGMOID_ACTIVATION;
  ⋮
end architecture BEH;
```

Abbildung 3.17: Ausführung der quantisierten Sigmoid Funktion

Die Register der Ausgaben der Aktivierungsfunktionen werden abhängig von einem Steuersignal vermuxt, welches die Aktivierungsfunktion festlegt.

3.2.8 Instruktionssatz

Es gibt zwei Typen von Instruktionen. Diese besitzen unterschiedlich formatierte Bitfelder, haben aber die gleiche Länge. Neben dem Standard-Instruktionstyp wird ein Weight-Instruktionstyp zum Laden von Gewichten eingeführt. Der Standard-Instruktionstyp besitzt die Felder OP-Code (Angabe der Operation), Length (Länge der Daten), Accumulator Address und Buffer Address.

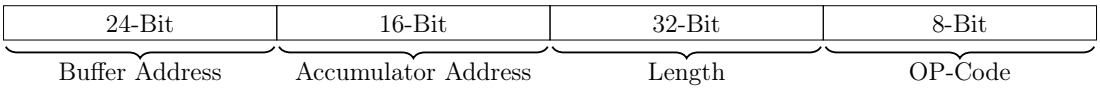


Abbildung 3.18: Standard-Instruktionstyp

Um eine skalierbare Architektur zu gewährleisten wurden die Adress- und Längenfelder entsprechend größer gewählt. So kann eine verhältnismäßig große TPU immer noch denselben Instruktionssatz verwenden.
Weight-Instruktionen fassen die Adressfelder zu einem großen Adressfeld zusammen, da diese für die Ausführung nicht benötigt werden und der Weight-Speicher in der Regel größer ist.

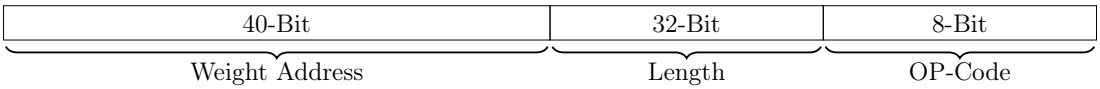


Abbildung 3.19: Weight-Instruktionstyp

Der OP-Code wird prioritätsgesteuert ausgelesen, so wird dieser in 6 mögliche Ausführungen eingeteilt. Das höchste gesetzte Bit markiert dabei den Ausführungstypen, alle anderen niederen Bits können nach Belieben verwendet werden. Ausnahmen bilden dabei der nop und der synchronize Befehl.

OP-Code	**Ausführung**	**Buffer Address**	**Accumulator Address**	**Length**
`00000000`	nop	don't care	don't care	don't care
`0000001X`	halt	don't care	don't care	don't care
`00001XXX`	read_weights	uses all 40 Bits	uses all 40 Bits	used
`001XXXXX`	matrix_multiply	used	used	used
`1XXXXXXX`	activate	used	used	used
`11111111`	synchronize	don't care	don't care	don't care

Tabelle 3.1: Priorisierung der Ausführungstypen und Nutzung der Bitfelder

read_weights

read_weights liest Gewichte aus dem Weight-Speicher. Dabei wird das 40-Bit breite Adressfeld Weight Address verwendet. Die Anzahl der in die MXU zu ladenden Gewichte steht im Length-Feld. Der OP-Code gibt an ob es sich um signed oder unsigned Integer handelt, da eine sign Extension vorgenommen werden muss. Die Adresse zum Laden der Gewichte in der MXU wird auf 0 zurückgesetzt, wenn diese größer ist als $N - 1$.

matrix_multiply

matrix_multiply liest mit Hilfe von Buffer Address Eingabedaten aus dem Unified Buffer und schleust diese durch die MXU. Die Accumulator Address gibt dabei an, unter welcher Adresse die Netzeingaben in den Akkumulatoren zwischengespeichert werden. Die Länge der Eingaben/Netzeingabe wird aus Length ausgelesen. Ob es sich um eine signed oder unsigned Matrixmultiplikation handelt und akkumuliert wird steht dabei im OP-Code.

activate

Der activate Befehl schleust Netzeingaben der Länge Length durch die Aktivierung und speichert diese wieder im Unified Buffer. Dabei werden die Felder Accumulator Address, Buffer Address und Length verwendet. Die Aktivierungsfunktion steht im OP-Code.

synchronize

Synchronize wartet auf die Abschließung aller noch anstehenden Instruktionen und löst dann einen Interrupt des Host-Systems aus. Dieser Befehl wird genutzt um den Host zu informieren, dass Berechnungen beendet wurden und dieser die Ergebnisse auslesen kann.

3.2.9 Steuerwerke

Die Ausführung der Instruktionen wird in den Steuerwerken umgesetzt. Da es 3 grundlegende Ausführungen gibt werden dementsprechend 3 Steuerwerke eingeführt. Diese werden wiederum von einem Koordinator angesteuert, welcher die Instruktionen dekodiert und dem entsprechendem Steuerwerk übergibt.

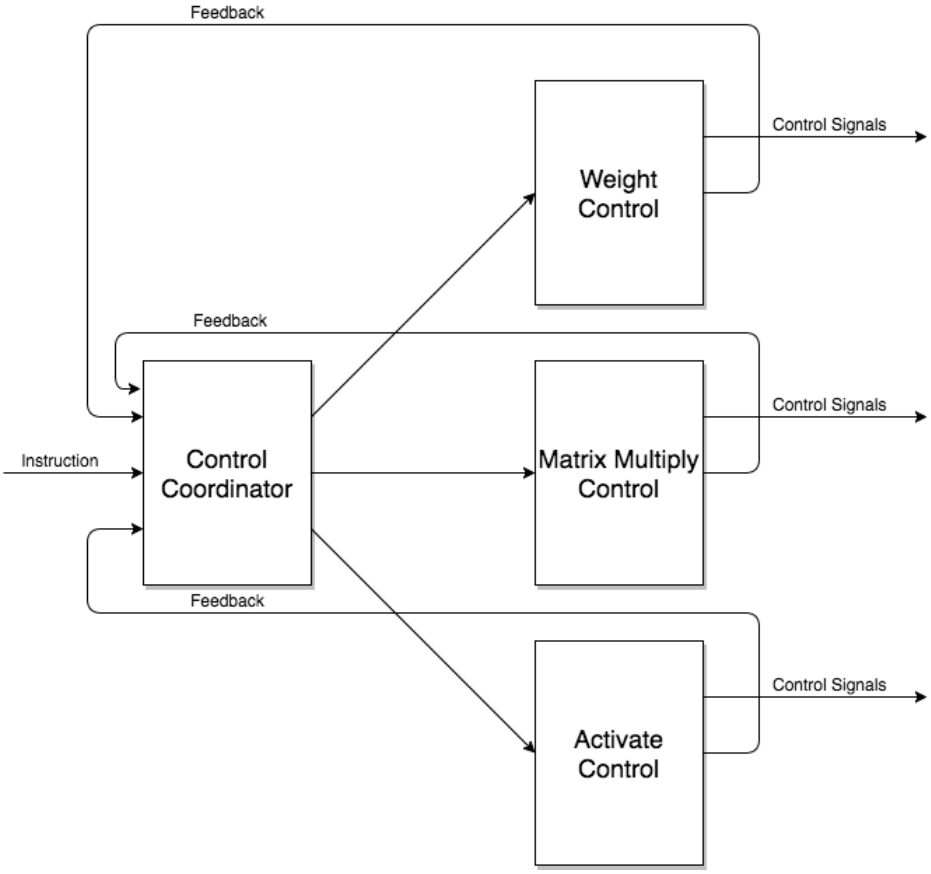


Abbildung 3.20: Aufbau der Steuereinheit

Die Steuerwerke besitzen ladbare Zähler für die momentanen Adressen der verschiedenen Komponenten. Mit Hilfe von Busy Signalen (Feedback) kann der Koordinator ausmachen, ob eine Instruktion noch ausgeführt wird. Kommt eine neue Instruktion während das benötigte Steuerwerk noch in Benutzung ist, so wartet der Koordinator bis die Instruktion abgearbeitet wurde, bevor dieser wiederum die neue Bearbeitung zulässt.

3.2.10 Instruktions-FIFO als LUTRAM

Da die Ausführungsdauer variiert und das Host-System parallel zu der TPU arbeitet wird ein FIFO-Speicher für die Instruktionen eingeführt. Hier werden neue Instruktionen reihenfolgenerhaltend zwischengespeichert und dann vom Koordinator abgerufen. Da der BRAM in anderen Komponenten verwendet wird, wird der Speicher für den FIFO in LUTRAM realisiert. Dies verhindert einen performanten FIFO bei höheren Tiefen, da der FIFO aber nicht sonderlich tief sein muss ist dies akzeptabel. Ein Ausschnitt eines Blockdiagramm der RTL-Analyse ist in Anhang A.5 zu sehen.

3.2.11 AMBA AXI-Interface

Für eine On-Chip Kommunikation wird der Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface (AXI) Standard verwendet [1]. Dieser wird in den meisten FPGAs/SoCs unterstützt. Im Rahmen der Arbeit wird dabei auf AXI4-Lite gesetzt. Dies erlaubt eine voll funktionstüchtige AXI Schnittstelle mit einer möglichst simplen Logik, wobei auf beschleunigte Zugriffe (burst mode) verzichtet wird. Die wichtigsten Signale der Schnittstellenbeschreibung werden im Folgenden kurz erläutert.

AXI4-Lite Schnittstellenbeschreibung

Die AXI4-Lite Schnittstelle unterstützt 5 Kanäle - Write address channel, Write data channel, Write response channel, Read address channel und Read data channel. Jeder Kanal besitzt eigene Signale für den Kontrollfluss. Daten und Adressen haben immer eine Breite von 32/64-Bit [1].

Global	**Write address channel**	**Write data channel**	**Write response channel**	**Read address channel**	**Read data channel**
ACLK	AWVALID	WVALID	BVALID	ARVALID	RVALID
ARESETn	AWREADY	WREADY	BREADY	ARREADY	RREADY
-	AWADDR	WDATA	BRESP	ARADDR	RDATA
-	AWPROT	WSTRB	-	ARPROT	RRESP

Tabelle 3.2: AXI4-Lite Schnittstellensignale [1]

VALID Signale signalisieren dem Kommunikationspartner, wenn ein Wert auf dem Übertragungsbus stabil anliegt, während READY zusagt das Datum abgeholt zu haben, sobald VALID angelegen hat. Bei falscher Nutzung der Steuersignale kann es zu einem Deadlock kommen.

Umsetzung mittels Automaten

Die richtige Nutzung der Steuersignale kann mit einem Automaten umgesetzt werden, dabei wird immer zuerst die Adresse und dann das dazugehörige Datum gelesen/geschrieben. Der Automat ändert die Steuersignale bei Eintritt in einen Zustand, somit handelt es sich um einen Moore-Automaten.

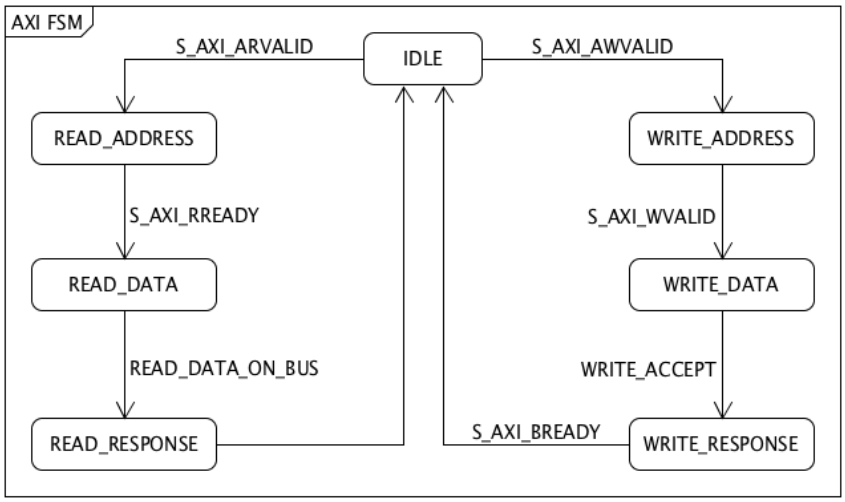


Abbildung 3.21: Zustandsautomat der AXI-Schnittstelle

Adressraum

Da es 3 unterschiedliche Speicher gibt, die vom Host-System beschrieben/ausgelesen werden, muss der Adressraum aufgeteilt werden. Die Zugehörigkeit des aufgeteilten Adressraumes wird dann anhand des höchstgesetzten Bits der Adresse ausgemacht. Hierbei wird der größte Speicher (Weight-Speicher) mit dem größten Most Significant Bit (MSB) zuerst gemapped. So muss für die Adressraumaufteilung nur das höchste bekannte Bit ausgewertet werden. Ist dieses gesetzt so wird der zweitgrößte Speicher (Unified Buffer) gemapped und das zugehörige MSB ausgewertet. Ist auch dieses Bit gesetzt so wird entsprechend in den Instruktions-FIFO geschrieben.
Diese Funktionsweise entspricht einer Prioritätslogik.
4 Bytes der Speicherein-/ausgänge werden immer zusammengefasst übertragen.

4 Umsetzung auf einem FPGA

Die Umsetzung des Konzepts wird in V$_{HSIC}$HDL vorgenommen. Evaluiert wird auf einem Xilinx Zynq 7020 mit integriertem SoC. Zunächst werden die Implementierungen in der Simulation getestet. Ist das Design in der Simulation evaluiert worden, so kann das gesamte Design inklusive dem SoC mittels eines Block Designs definiert werden. Die Synthese kann dann den SoC mit einbeziehen.

4.1 Evaluation in der Simulation

4.1.1 Matrix Multiply Unit

Das folgende Bild zeigt die Evaluation einer 4×4 MXU in der Simulation. Hier ist besonders die Diagonalität der Eingaben und das Laden der Gewichte an die Gewichtadressen zu sehen. Das Ergebnis wird wieder parallel ausgegeben.

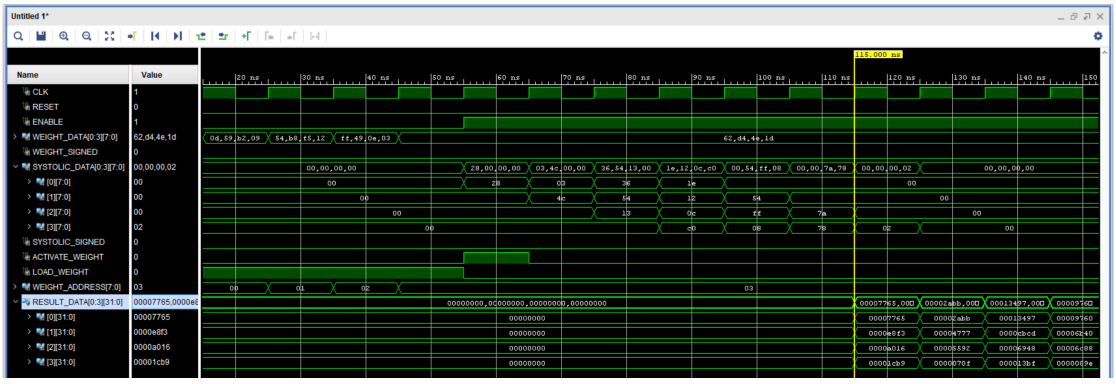


Abbildung 4.1: Simulation einer 4×4 MXU

Die Testbench vergleicht automatisiert das Ergebnis mit vordefinierten Matrizen, um Fehler zu erkennen. Im Anhang A.6 ist diese Abbildung vergrößert dargestellt.

4.1.2 Aktivierung

Die Abbildung 4.2 zeigt die Simulation der Sigmoid Aktivierungsfunktion. Dabei wird zuerst signed und dann unsigned ausgegeben. Es sind Stufen in den Funktionen zu erkennen, was der Quantisierung verschuldet ist. Der Ausschlag am Ende der Funktionen ist ein Überlauf der beim Runden entsteht, da hier der Maximalwert als Eingabe in Form von Äquivalenzklassen getestet wird.

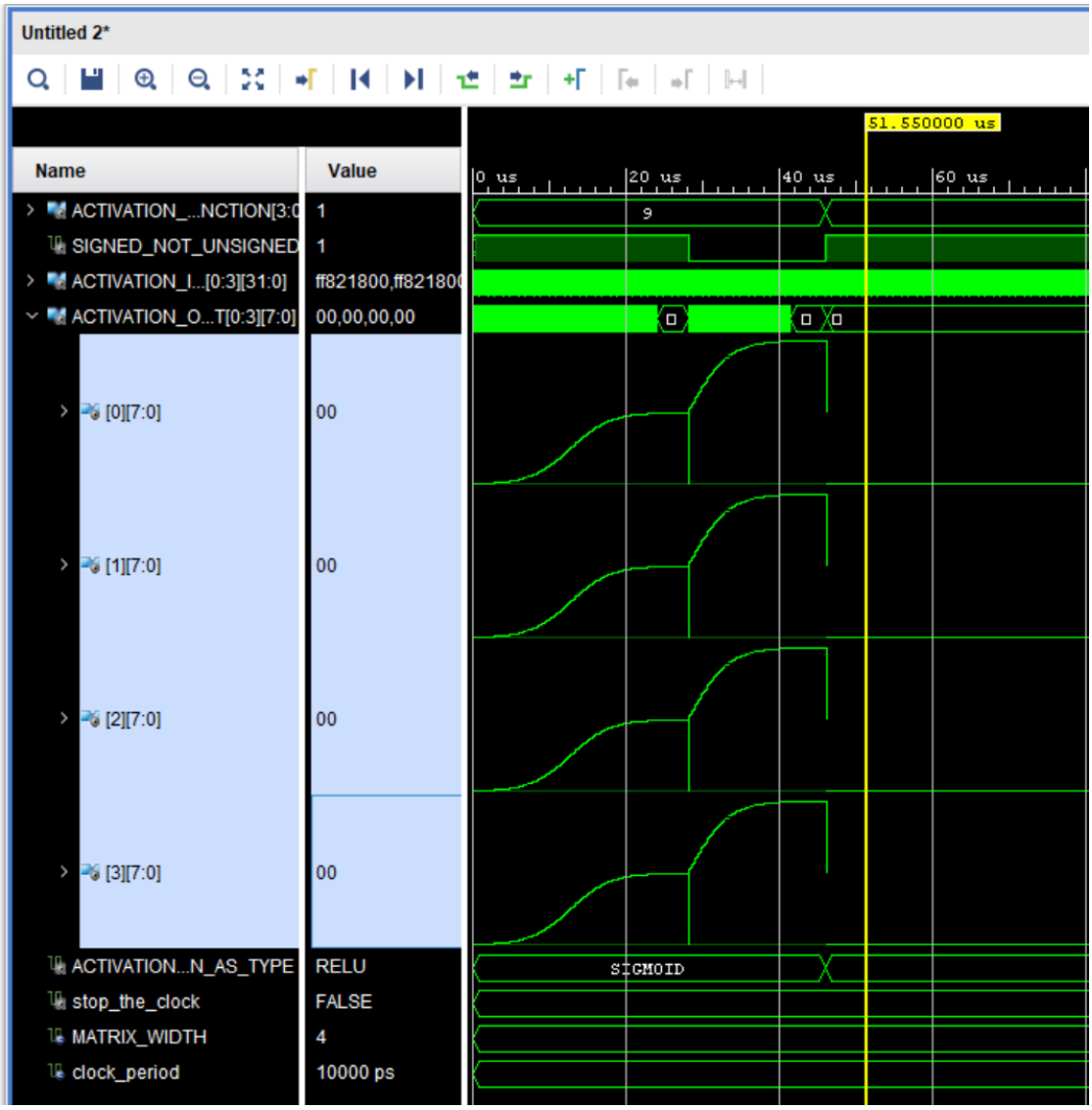


Abbildung 4.2: Simulation der un-/signed Sigmoid Aktivierungsfunktion

In Abbildung 4.3 ist die signed ReLU Aktivierungsfunktion zu sehen.

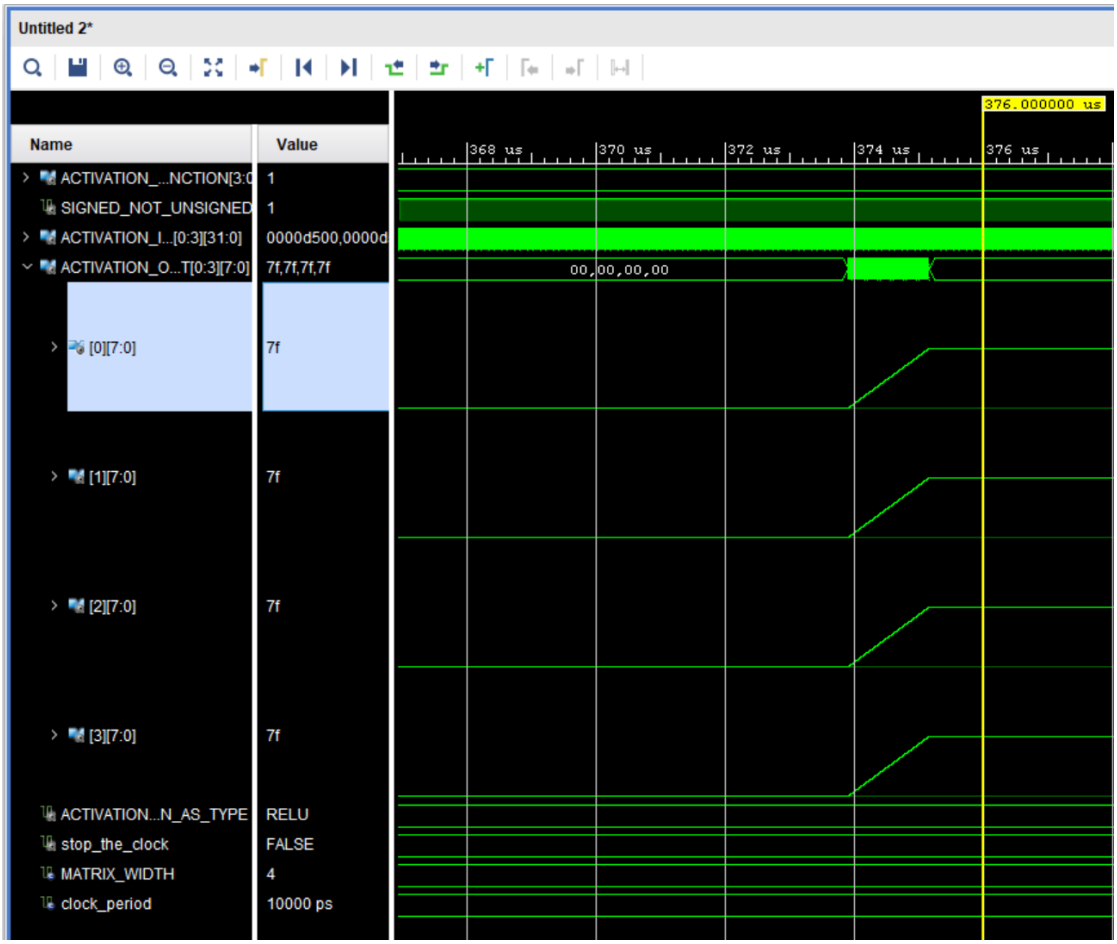


Abbildung 4.3: Simulation der signed ReLU Aktivierungsfunktion

Abbildung 4.4 zeigt die Simulation der unsigned ReLU Aktivierungsfunktion. Diese beginnt aufgrund des Wertebereiches bei 0.

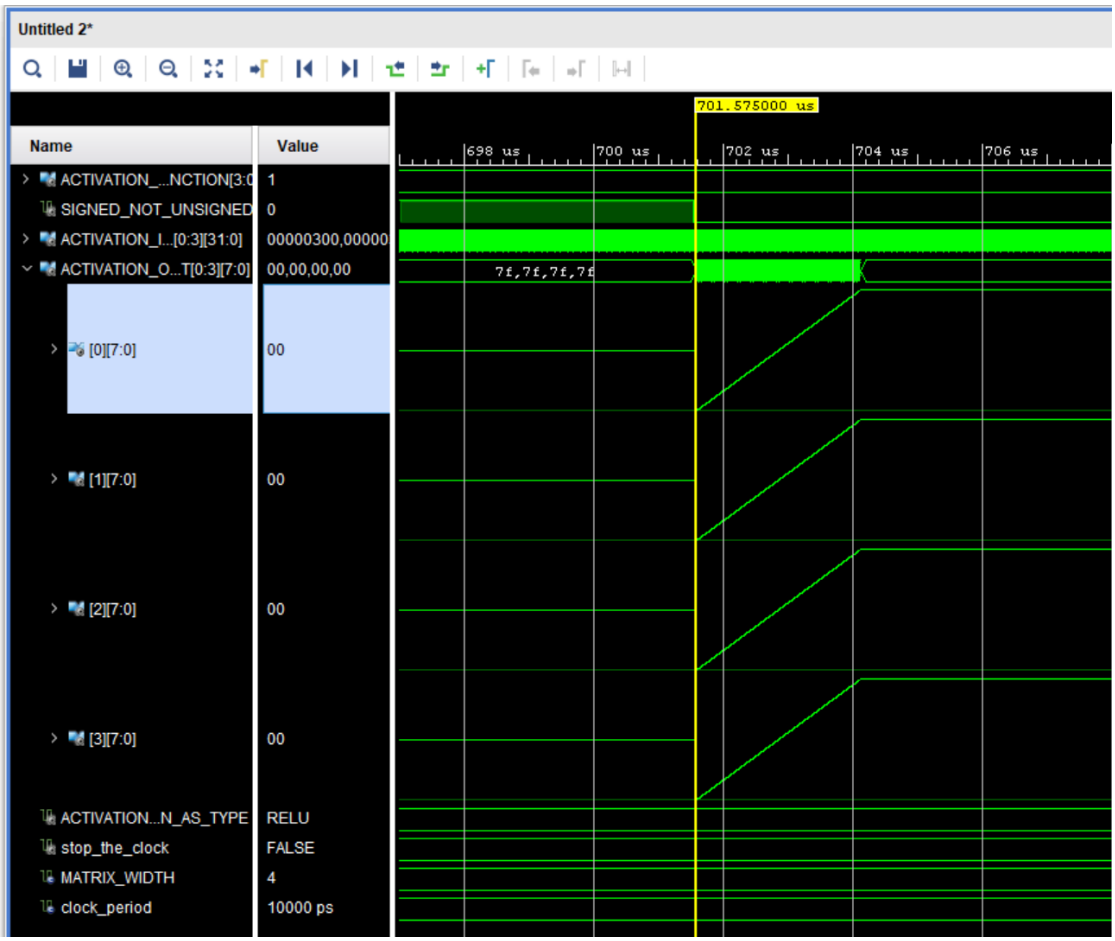


Abbildung 4.4: Simulation der unsigned ReLU Aktivierungsfunktion

4.2 Erstellen des Block Designs

Für das vollständige Design inklusive des SoCs wird ein Block Design erstellt. Hierfür wird zunächst ein Intellectual Property Core (IP Core) der TPU erstellt, wobei der V_{HSIC}HDL Code gekapselt wird. Tools (hier Vivado) sind die Formen der Schnittstellen wie AXI bekannt. Neben vorgefertigten IP Cores kann dann der soeben erstellte IP Core zu einem Block Design hinzugefügt werden. Dabei wird der Zynq SoC, ein AXI Interconnect für Adressauflösungen, ein System Reset und die TPU hinzugefügt. Die Verbindungen kann das Tool automatisch vornehmen.
Die TPU wird über den AXI Interconnect an den AXI Master Port des Zynqs angeschlossen. Das Synchronize interrupt Signal wird hingegen mit dem Interrupt Eingang verbunden.
Der Zynq 7020 besitzt genügend Ressourcen für eine TPU mit 14×14 MXU, 32768×14 Byte Weight-Speicher, 4096×14 Byte Unified Buffer und $512 \times 14 \times 4$ Byte Akkumulatoren.

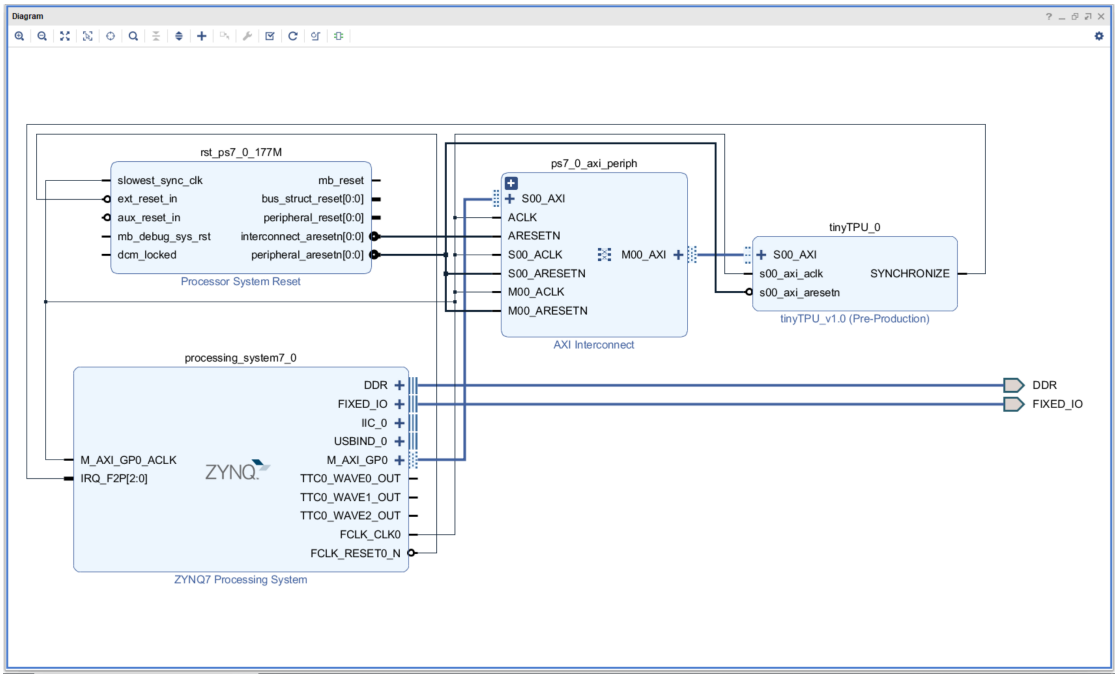


Abbildung 4.5: Block Design mit SoC und TPU als Peripherie

4.3 Synthese und Implementation

Die Synthese wird auf Performance eingestellt, zusätzlich wurde Retiming (Verschieben von Registern) und Behalten von äquivalenten Registern eingeschaltet. Dies hat sich für die spätere Implementation als hilfreich erwiesen. Die Schätzung der Ressourcennutzung ist in Tabelle 4.1 zu sehen. Dabei wurde eine Taktfrequenz von $177,\overline{77}$MHz angestrebt.

Resource	**Estimation**	**Available**	**Utilization in %**
LUT	4013	53200	7.53
LUTRAM	240	17400	1.38
FF	7219	106400	6.78
BRAM	139	140	99.29
DSP	218	220	99.09
BUFG	1	32	3.13

Tabelle 4.1: Schätzung der Ressurcennutzung seitens der Synthese

Das von der Synthese optimierte Design wird von der Implementation auf das konkrete FPGA angepasst und weiter optimiert. Die endgültige Ressourcennutzung zeigt Tabelle 4.2. Ein Floorplan ist in Anhang A.1 zu sehen.

Resource	**Utilization**	**Available**	**Utilization in %**
LUT	3985	53200	7.49
LUTRAM	239	17400	1.37
FF	7144	106400	6.71
BRAM	139	140	99.29
DSP	218	220	99.09
BUFG	1	32	3.13

Tabelle 4.2: Ressurcennutzung nach der Implementation

Der Worst Negative Slack beträgt dabei 22ps und bezeichnet den kritischen Pfad. Das Design benötigt eine Leistung von 1,838W wobei der SoC 1,493W der Gesamtleistung benötigt. Abbildung 4.6 zeigt die Leistungsaufnahme verschiedener Komponenten.

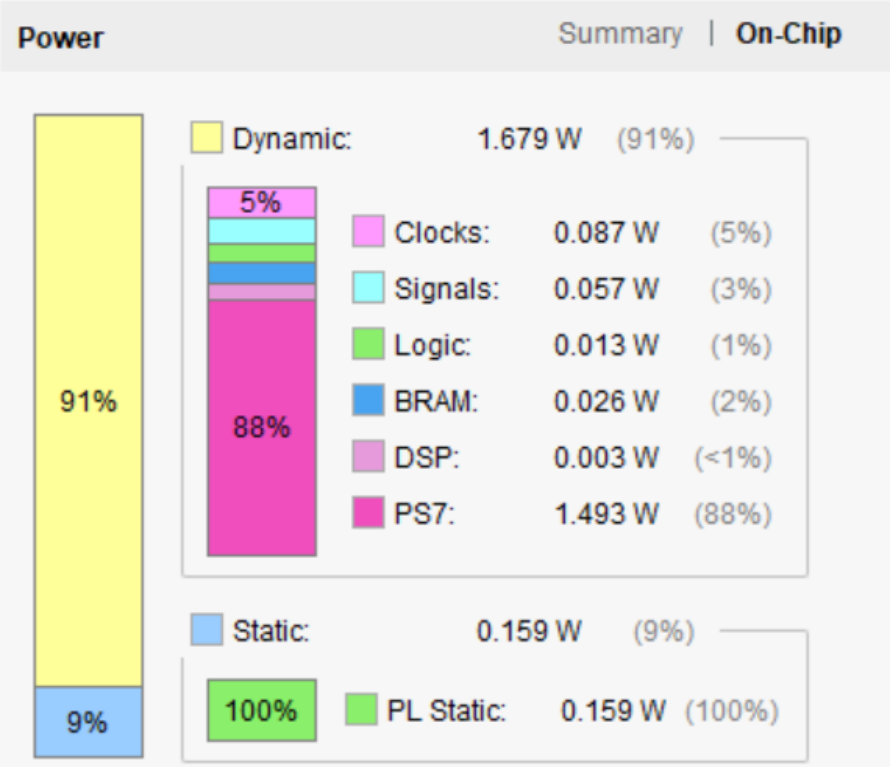


Abbildung 4.6: Leistungsaufnahme des Designs

4.4 Board Support Package

Die Zugriffe auf die TPU von dem Prozessor des SoCs werden mittels Board Support Package (BSP) unterstützt, hierbei handelt es sich um Funktionen umgesetzt in C. Dabei können gesamte Gewichte-/Eingabe-Vektoren und Instruktionen mit 32-Bit Zugriffen übertragen werden. Abbildung 4.7 zeigt die Definition der Typen zur Übertragung in einem C ähnlichen Pseudocode.

```
typedef union tpu_vector {
  // Assignment array
  uint8_t byte_vector[TPU_VECTOR_SIZE];
  // Transfer array
  uint32_t transfer_vector[TPU_VECTOR_PADDING/sizeof(uint32_t)];
} tpu_vector_t;

typedef union __attribute__((__packed__)) instruction {
  // Assignment structure
  struct {
    uint8_t op_code;
    uint8_t calc_length[4];
    union {
      struct {
        uint8_t acc_address[2];
        uint8_t buf_address[3];
      };
      uint8_t weight_address[5];
    };
  };
  // Transfer structure
  struct {
    uint32_t lower_word;
    uint32_t middle_word;
    uint16_t upper_word;
  };
} instruction_t;
```

Abbildung 4.7: Definition der Typen für TPU-Zugriffe

Instruktionen werden per Schreibzugriff in den FIFO-Speicher übertragen. Abbildung 4.8 zeigt die Übertragung. Bei WRITE_32 und READ_32 handelt es sich um Macros zum Lesen und Schreiben an eine Adresse.

```
int32_t write_instruction(instruction_t *instruction) {
  WRITE_32(TPU_INSTRUCTION_BASE+TPU_LOWER_WORD_OFFSET,
    instruction->lower_word);
  WRITE_32(TPU_INSTRUCTION_BASE+TPU_MIDDLE_WORD_OFFSET,
    instruction->middle_word);
  WRITE_16(TPU_INSTRUCTION_BASE+TPU_UPPER_WORD_OFFSET,
    instruction->upper_word);

  return 0;
}
```

Abbildung 4.8: Schreibzugriff auf den Instruktions-FIFO

Die Schreibzugriffe für den Weight Speicher ist in Abbildung 4.9 zu sehen. Da Gewichte nicht wieder gelesen werden müssen wird auf einen Lesezugriff verzichtet. Die Adressen beziehen sich immer auf einen kompletten Vektor.

```
int32_t write_weight_vector(tpu_vector_t *weight_vector,
  uint32_t weight_address) {
  if(weight_address >= WEIGHT_BUFFER_SIZE) return EFAULT;

  weight_address <<= (uint32_t)(ceil(log2(TPU_VECTOR_SIZE)));

  for(uint32_t i = 0; i < TPU_VECTOR_SIZE; i+=sizeof(uint32_t)) {
    WRITE_32(TPU_WEIGHT_BUFFER_BASE+weight_address+i,
      weight_vector->transfer_vector[i/sizeof(uint32_t)]);
  }

  return 0;
}
```

Abbildung 4.9: Schreibzugriff auf den Weight-Speicher

Schreibzugriffe für den Unified Buffer werden lesend und schreibend unterstützt, da hier die Eingaben als auch die Ausgaben des Netzes abgelegt werden.

```
int32_t write_input_vector(tpu_vector_t *input_vector,
  uint32_t buffer_address) {
  if(buffer_address >= UNIFIED_BUFFER_SIZE) return EFAULT;

  buffer_address <<= (uint32_t)(ceil(log2(TPU_VECTOR_SIZE)));

  for(uint32_t i = 0; i < TPU_VECTOR_SIZE; i+=sizeof(uint32_t)) {
    WRITE_32(TPU_UNIFIED_BUFFER_BASE+buffer_address+i,
      input_vector->transfer_vector[i/sizeof(uint32_t)]);
  }

  return 0;
}

int32_t read_output_vector(tpu_vector_t *output_vector,
  uint32_t buffer_address) {
  if(buffer_address >= UNIFIED_BUFFER_SIZE) return EFAULT;

  buffer_address <<= (uint32_t)(ceil(log2(TPU_VECTOR_SIZE)));

  for(uint32_t i = 0; i < TPU_VECTOR_SIZE; i+=sizeof(uint32_t)) {
    output_vector->transfer_vector[i/sizeof(uint32_t)] =
      READ_32(TPU_UNIFIED_BUFFER_BASE+buffer_address+i);
  }

  return 0;
}
```

Abbildung 4.10: Schreib- und Lesezugriff auf den Unified Buffer

5 Evaluation

Dieses Kapitel beschäftigt sich mit der Evaluation der umgesetzten TPU. Dabei wird auf das genutzte Modell und dessen Ausführung eingegangen, als auch auf die Skalierbarkeit, die Perfomance und die Leistungsaufnahme. Zum Schluss wird ein direkter Vergleich mit der Ausführung in TensorFlow vorgenommen.

5.1 Training in TensorFlow

Für das Trainieren des Netzes wird TensorFlow [9] mit Keras [3] verwendet. Als Trainingsdatensatz wird der MNIST Datensatz genutzt.

5.1.1 MNIST Datensatz

Beim MNIST Datensatz von handgeschriebenen Ziffern [17] handelt es sich um einen Datensatz, gedacht um ML-Modelle zu trainieren. MNIST wird häufig für Benchmarks verwendet um die Genauigkeit eines Netzes zu bewerten.
Der Datensatz eignet sich zur Klassifizierung der handgeschriebenen Ziffern, dabei besteht eine Ziffer immer aus einem Bild mit 28×28 Pixel in Graustufen. Der Datensatz beinhaltet 70000 Samples, wovon 60000 Labels zum Trainieren besitzen und 10000 zur Evaluation gedacht sind.

5.1.2 Modell

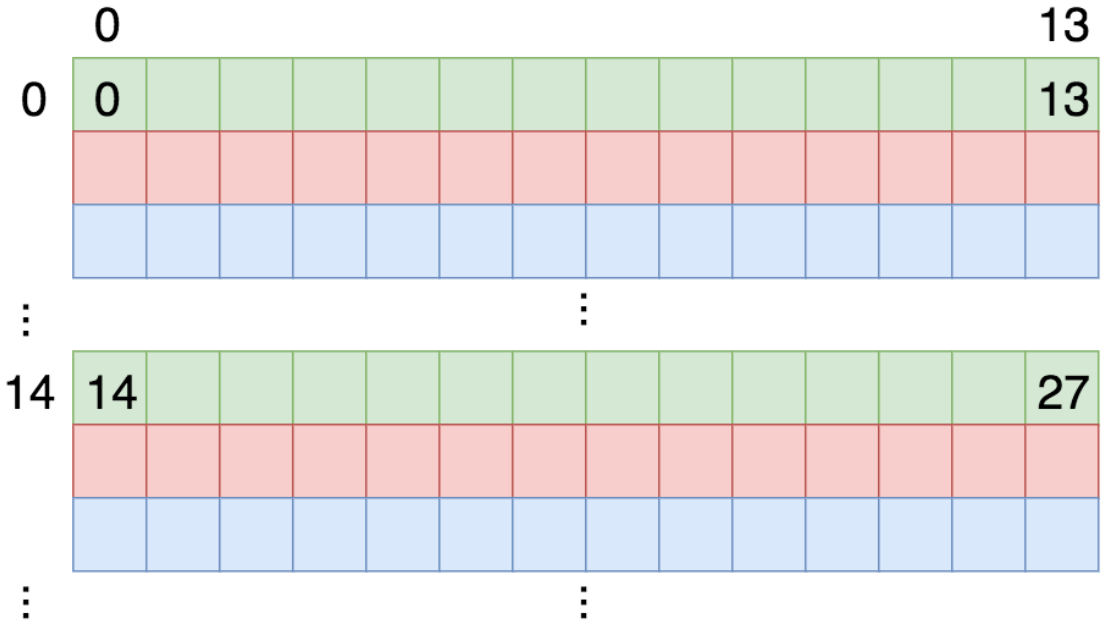
Das ML-Modell für die Evaluation besitzt 2 Schichten, wobei die erste Schicht 504 (durch 14 teilbar) und die zweite Schicht als Ausgabeschicht 10 Neuronen besitzt. So kann das Modell nachher auf der TPU in einem Durchgang komplett ausgerechnet werden, ohne dass Daten aus den Akkumulatoren aktiviert werden müssen, bevor eine Schicht abgeschlossen wurde. Die Gewichte werden auf die Maximalwerte begrenzt, sodass diese garantiert innerhalb des Wertebereichs liegen. Es wird zur Vereinfachung immer signed gerechnet, da unsigned Berechnungen nur Sinn machen, wenn es in einer Matrixmultiplikation keine negativen Eingaben und Gewichte gibt, sodass im positiven Bereich Genauigkeit gewonnen werden kann. Die Evaluation wird mit der Sigmoid Aktivierung durchgeführt. Da ein vollwertiger Bias nur durch Rekonstruktion erreicht werden kann wird das Netz ohne Bias trainiert.

5.1.3 Quantisierung der Gewichte

Die Gewichte werden für die Ausführung auf der TPU quantisiert, da diese von TensorFlow mit Floating-Point trainiert werden.

5.1.4 Exportieren des Modells

Das Modell wird als CSV-Datei exportiert. Die Gewichte werden dann aufgeteilt in $14 \times m$ Matrizen. Ist die Matrix nicht durch 14 teilbar, so werden die letzten Matrizen mit Nullen aufgefüllt. Die Eingaben werden als $m \times 14$ Matrizen aufgeteilt. Aufgrund der Architektur der TPU können 14 Eingaben parallel berechnet werden, weshalb Teilvektoren der Länge 14 der 14 Eingaben hintereinander im Unified Buffer stehen müssen. Dies wird in Abbildung 5.1 visualisiert.



Daten einer Farbe gehören zu dem selben Eingabevektor. Es werden 14 Byte breite Daten adressiert.

Abbildung 5.1: Visualisierung der Eingaben im Unified Buffer

5.2 Laden und Ausführen des Modells

Ein Programm auf dem Zynq liest Vektoren der Länge 14 von der SD-Karte aus und lädt diese in den Speicher der TPU. Instruktionen werden ebenfalls in einem ähnlichen Format ausgelesen und in den Instruktions-FIFO abgelegt. Die letzte Instruktion ist immer ein synchronize Befehl. Nachdem der dazugehörige Interrupt ausgelöst wurde liest das Programm die Ergebnisse aus dem Unified Buffer und speichert diese als CSV-Datei auf die SD-Karte.

5.3 Skalierbarkeit der TPU

Die TPU wird neben der Größe von 14×14 ebenfalls in kleineren Größen bei gleicher Frequenz synthetisiert, um die Leistungsaufnahme und Ressourcennutzung abschätzbar zu machen. Die tatsächliche Geschwindigkeit mit Matrixmultiplikationen verschiedener Größen wird allerdings auf der 14×14 TPU getestet (z.B. rechnen mit 6×6 Matrixmultiplikationen).

5.3.1 Ressourcennutzung

Die Ressourcennutzung wird für 5 Größen der TPU evaluiert, hierzu wurde das Design in diesen Größen synthetisiert und implementiert. Tabelle 5.1 zeigt die Ressourcennutzung der evaluierten Größen. Um bei der größten TPU auf die vorgegebene Frequenz zu kommen repliziert die Synthese Logik häufiger, was für kleinere Größen aufgrund der freieren Nutzbarkeit von Ressourcen nicht nötig ist. Dies wirkt sich auch auf die Dauer der Implementation aus.

Größe	**6**	**8**	**10**	**12**	**14**
LUT	802	873	936	970	3985
LUTRAM	93	109	125	141	239
FF	2112	2536	3031	3519	7144
BRAM	49	68	86.5	105.5	139
DSP	47	77	115	161	218
BUFG	1	1	1	1	1

Tabelle 5.1: Ressourcennutzung verschiedener TPU Größen

5.3.2 Leistungsaufnahme

Die geschätzte Leistungsaufnahme in W der 5 evaluierten Größen der TPU sind neben den Temperaturen in C° in Tabelle 5.2 zu sehen. Dabei wird auch die Leistung des Prozessors dargestellt.

Größe	**6**	**8**	**10**	**12**	**14**
Leistung des Prozessor in W	1.493				
Dynamische Leistung der TPU in W	0.04	0.051	0.061	0.071	0.186
Statische Leistung der TPU in W	0.146	0.148	0.15	0.152	0.159
Temperatur in C°	44.4	44.5	44.7	44.8	46.2

Tabelle 5.2: Leistungsaufnahme verschiedener TPU Größen

5.3.3 Theoretische Geschwindigkeit

Die theoretische Geschwindigkeit wird in OPS errechnet. Da eine Addition/Multiplikation als eine Operation eines herkömmlichen Prozessors aufgefasst werden kann ergibt sich dann in Abhängigkeit von N eine theoretische Geschwindigkeit von:

$$OPS = (2N^2 + N)Operations * 177,\overline{77}MHz$$

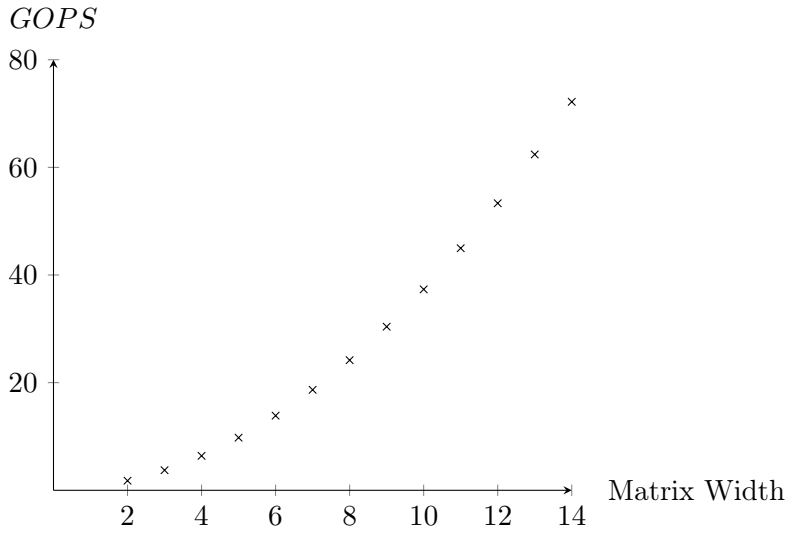


Abbildung 5.2: Theoretische Geschwindigkeit der TPU in $GOPS$ in Abhängigkeit der MXU Größe

Da die Aktivierungsfunktion unterschiedlich ausfallen und nicht unbedingt in Operationen aufgefasst werden kann wird diese in der theoretischen Abschätzung vernachlässigt.

5.3.4 Gemessene Geschwindigkeit

Die tatsächliche Geschwindigkeit wird anhand der Ausführungsdauer des beschriebenen Modells gemessen. Hierbei wurde auf einen Zähler innerhalb der TPU zurückgegriffen, der beim ersten Eintreffen einer Instruktion das Zählen pro Takt startet und beim synchronize Interrupt den Zählwert beibehält. Die Ausführungsdauer kann aufgrund der Übertragung der Instruktionen unterschiedlich ausfallen, weshalb die Messung für alle Testdaten wiederholt wird (715 Ausführungen). Abbildung 5.3 zeigt die Ergebnisse der Messungen der TPU mit 14×14 MXU in einem Histogramm.

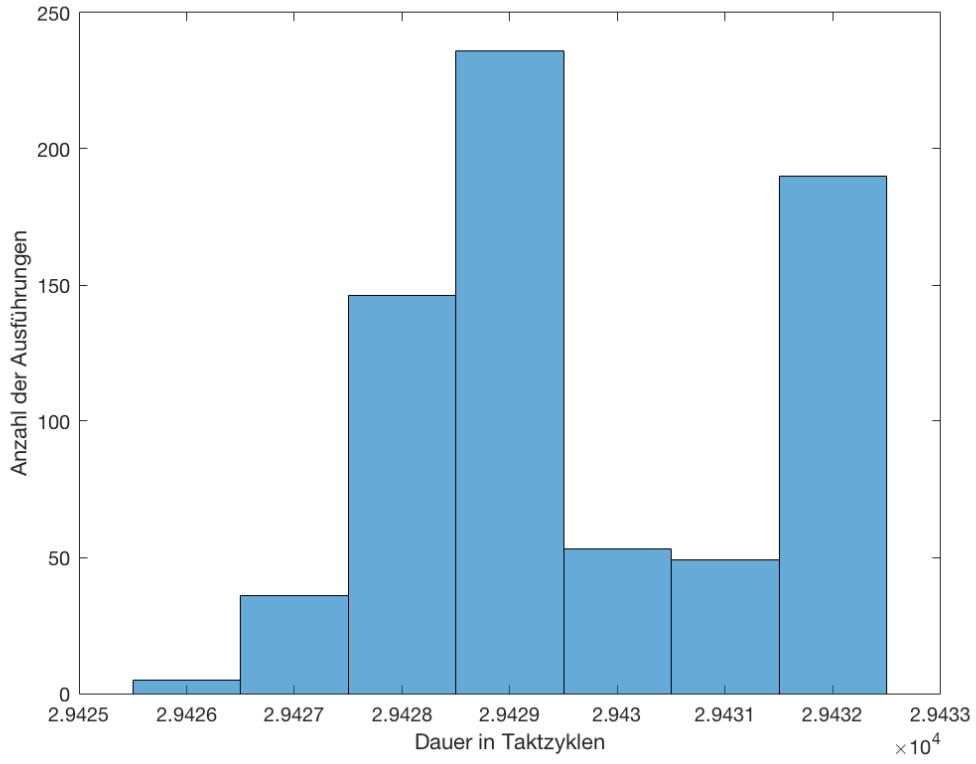


Abbildung 5.3: Häufigkeitsverteilung von Zeitmessungen der Ausführung des Testmodells mit 14×14 MXU

Für eine Auswertung verschiedener TPU Größen wurden Messungen mit derselben Hardware (14×14 TPU) aber verschiedenen Größen von Matrizen vorgenommen. Dabei wird immer das Mittel von 10 Messungen verwendet, was sich aufgrund der kleinen Verteilung für einen Vergleich als ausreichend erweist. Tabelle 5.3 zeigt die Messungen, wobei 5 verschiedene Größen betrachtet werden. Geschwindigkeiten nicht vorhandener Messungen lassen sich eventuell hieraus abschätzen. Da die TPU N Eingabevektoren parallel berechnen kann, wird in der Tabelle auch die Berechnungsdauer pro Vektor dargestellt und mit herkömmlichen Prozessoren (mit TensorFlow in Floating-Point) verglichen.

Tensor Processing Unit mit $177,\overline{77}MHz$						**Intel**	**BCM2837**
Matrix Größe	**6**	**8**	**10**	**12**	**14**	**Core i5**	**4x ARM**
Anzahl der						**-5287U**	**Cortex-A53**
Instruktionen	431	326	261	216	186	mit	mit
Dauer in μs	383,746	289,014	234,277	194,222	165,528	$2,9GHz$	$1.2GHz$
Dauer pro							
Vektor in μs	63,958	36,127	23,428	16,185	11,823	62	763

Tabelle 5.3: Zeitmessungen des Testmodells mit verschiedenen MXU Größen

5.4 Vergleich mit TensorFlow

Für eine Auswertung der Ergebnisse wird die Genauigkeit der Berechnungen mit TensorFlow verglichen. TensorFlow kommt mit den quantisierten Gewichtungen und Floating-Point Berechnungen auf eine Genauigkeit von $97,86\%$, die TPU hingegen auf $97,73\%$. Fehler der Ausgaben werden in Abbildung 5.4 gezeigt. Hierbei wird auf den maximalen und durchschnittlichen Fehler eingegangen.

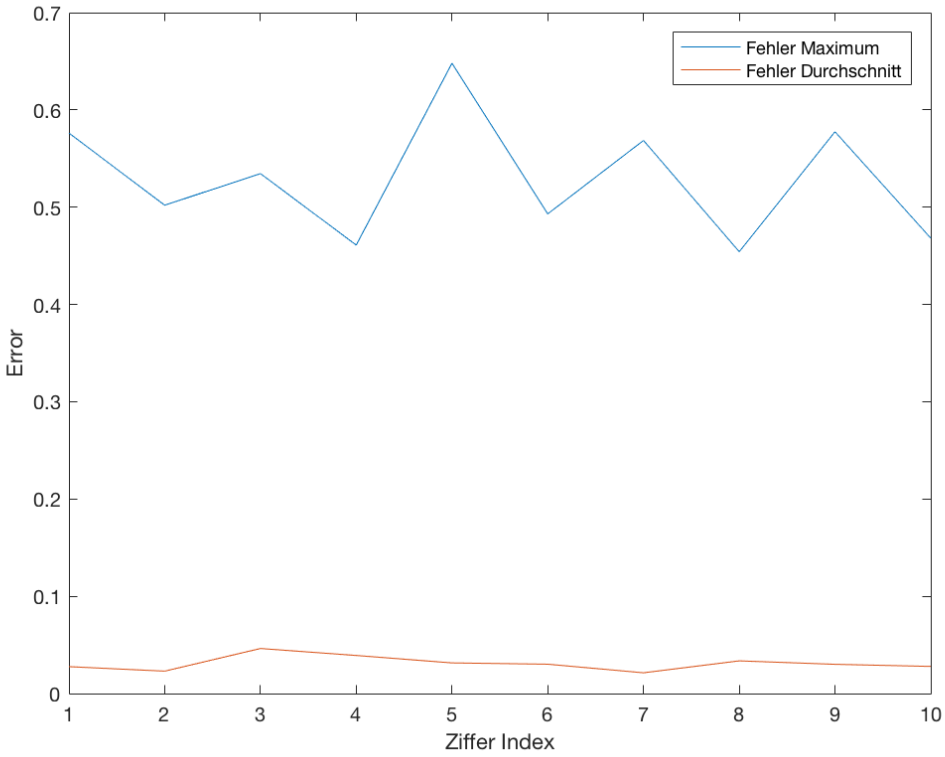


Abbildung 5.4: Fehler der Ausgaben im Vergleich zu TensorFlow

Während der maximale Fehler verhältnismäßig groß ausfällt, zeigt der Durchschnittsfehler eine geringere Auswirkung auf die Ergebnisse. Da der Fehler Regelmäßigkeiten aufweist (Abbildung 5.5) kann dieser während des Trainings angelernt werden. Hierzu müssten während des Trainings die Netzeingaben und die Aktivierungsfunktionen äquivalent zur TPU quantisiert werden.

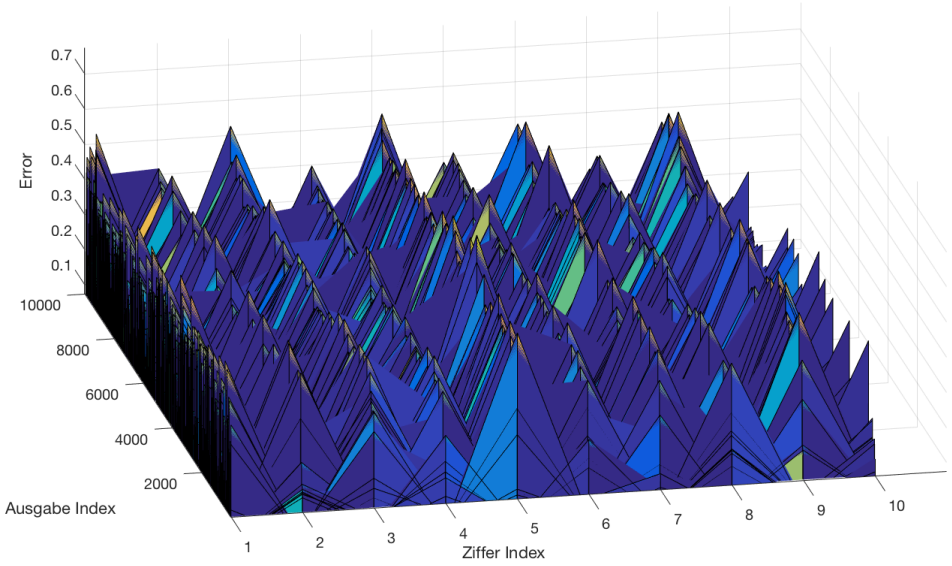


Abbildung 5.5: Erkennbare Regelmäßigkeiten des Fehlers

6 Zusammenfassung und Fazit

Diese Arbeit beschäftigt sich mit der Implementierung eines ML-Co-Prozessors. Dabei wurde die Architektur an die der Tensor Processing Units angelehnt und auf die Bedürfnisse von Embedded Systems und IoT Geräte zugeschnitten. So ist dieser Co-Prozessor je nach Verfügbarkeit von Ressourcen skalierbar und auch für Kleinstgeräte nutzbar. Eine Umsetzung wurde in V_{HSIC}HDL realisiert und auf einem FPGA evaluiert. Dank der AXI-Schnittstelle des Co-Prozessors kann dieser in verschiedensten Umgebungen eingesetzt werden.

Für die Evaluation wurde Ressourcennutzung, Leistungsaufnahme und Geschwindigkeit verschiedener Größen des Co-Prozessors getestet und anhand eines Testmodells ausgewertet. Für einen Vergleich mit herkömmlichen Prozessoren wurde TensorFlow mit dem selben Modell eingesetzt. Hier wurde auch auf die Genauigkeit und Abweichung zu TensorFlow eingegangen.

6.1 Fazit

Aus dieser Arbeit entstand ein einsatzbereiter ML-Co-Prozessor. Im Vergleich zu herkömmlichen Prozessoren zeigt die Ausführungsdauer klare Vorteile der Architektur. Vor allem für Embedded Systems und IoT Geräte kann dieser Co-Prozessor sinnvoll sein, da die Umgebung nicht immer den nötigen Platz für vollwertige Grafikkarten oder leistungsstarke CPUs bietet. Der Stromverbrauch hält sich in der Abschätzung durchaus in Grenzen und fällt im Gegensatz zu dem ARM-SoC gering aus. Ein ASIC als SoC mit einer solchen TPU ist vorstellbar, hier würde der Vorteil eines geringeren Stromverbrauchs und einer deutlich höheren Taktrate die Nutzbarkeit noch erhöhen.

6.2 Ausblicke

Im folgenden wird auf Möglichkeiten der Erweiterungen eingegangen, um die Performance und Genauigkeit der entstandenen Lösung zu verbessern.

6.2.1 Ausschöpfung der DSP-Blöcke

Da die möglichst hohen Taktraten von den DSP-Blöcken abhängen und nur eine begrenzte Anzahl von DSP-Blöcken zur Verfügung steht sollte die Nutzung der einzelnen Blöcke noch optimiert werden. Da es sich um 8-Bit Integer handelt, die Eingänge der DSP-Blöcke aber wesentlich breiter sind, ist es möglich mit nur einem Block 2 Multiply-Add Einheiten zu erstellen. Hierzu muss ein Eingang des Multiplizierers mit 2 8-Bit (+1-Bit) Eingängen konkateniert werden, wobei diese um 18-Bit verschoben sind, damit sich die Ergebnisse nicht beeinflussen (da $Q9 * Q9 \mapsto Q18$). Nicht alle DSP-Block-Versionen besitzen genügend breite Eingänge, um dies zu realisieren. Mitunter kann dieses Verfahren beim DSP48E2 (27-Bit Eingang) umgesetzt werden [5]. In der Evaluation standen lediglich DSP48E1 (25-Bit Eingang) Blöcke zur Verfügung.

6.2.2 Entfernen der unsigned Komponenten

Das Nutzen von unsigned Integern hat sich während der Evaluation als umständlich erwiesen. Da der negative Bereich einfach abgeschnitten wird ist es nur sinnvoll auf unsigned Berechnungen zu setzen, wenn Gewichtungen und Eingaben einer Schicht nicht vorzeichenbehaftet sind. Es wird zwar dann im positiven Bereich Genauigkeit gewonnen, dies kann allerdings nur selten ausgenutzt werden. Das Entfernen der unsigned Berechnungen kann Ressourcen sparen, somit auch der Geschwindigkeit helfen und ermöglicht die Nutzung der DSP48E1 Blöcke als 2 Multiply-Add Einheiten, da die Breite von $Q9$ auf $Qu8$ reduziert werden kann.

6.2.3 Integer-Arithmetic-Only Quantization

Um den Wertebereich nicht beschränken zu müssen könnte eine Unterstützung für Integer-Arithmetic-Only Quantization nachgeliefert werden. Diese vereinfacht die Nutzung der TPU, da Netze beim Training nicht eingegrenzt werden müssen.

6.2.4 Floating-Point Version

Für eine höhere Genauigkeit sollte eine Floating-Point Version in Betracht gezogen werden. Zwar werden aufgrund des Overheads der Berechnungen mehr Ressourcen fällig, dies kann sich aber für Anwendungen lohnen, bei denen eine hohe Genauigkeit hilfreich ist. Floating-Point Berechnungen können mit Hilfe von DSP-Blöcken umgesetzt werden, allerdings benötigt dies erheblich mehr Blöcke als Festkommaarithmetik.

6.2.5 Pooling und weitere Aktivierungsfunktionen

Um vollwertige CNNs zu unterstützen kann die Aktivierung um Pooling erweitert werden. Der OP-Code bietet hierzu noch genügend ungenutzte Bits. Ebenso können andere Aktivierungfunktionen nachgeliefert werden.

6.2.6 Learning

Die TPU bietet eine gute Grundlage, um diese um Lernalgorithmen zu erweitern. Dies erfordert allerdings eine Abwandlung der Architektur und eine Erweiterung des Instruktionssatzes.

Literaturverzeichnis

[1] ARM Limited: *AMBA AXI and ACE Protocol Specification*. 2011

[2] Brown, Stephen D. ; Francis, Robert J. ; Rose, Jonathan ; Vranesic, Zvonko G.: *Field-Programmable Gate Arrays*. Springer Science and Business Media, 2012

[3] Chollet, François u. a.: *Keras: The Python Deep Learning library*. – URL https://keras.io/. – Zugriffsdatum: 11.11.2018

[4] Cong, Jason ; Ding, Yuzheng: Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays / University of California, Los Angeles. 1996. – Forschungsbericht

[5] Fu, Yao ; Wu, Ephrem ; Sirasao, Ashish: 8-Bit Dot-Product Acceleration / Xilinx, Inc. 2017. – Forschungsbericht

[6] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016

[7] Google, Inc.: *Cloud TPU*. – URL https://cloud.google.com/tpu/. – Zugriffsdatum: 10.11.2018

[8] Google, Inc.: *Edge TPU*. – URL https://cloud.google.com/edge-tpu/. – Zugriffsdatum: 10.11.2018

[9] Google, Inc.: *TensorFlow*. – URL https://www.tensorflow.org/. – Zugriffsdatum: 11.11.2018

[10] Harris, David ; Harris, Sarah: *Digital Design and Computer Architecture*. Morgan Kaufmann, 2010

[11] Harville, David A.: *Matrix Algebra From a Statistician's Perspective*. Springer, 1997

[12] Haykin, Simon: *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1994

[13] Intel Corporation: *Vision Processing Unit*. – URL https://www.movidius.com/solutions/vision-processing-unit. – Zugriffsdatum: 10.11.2018

[14] Jacob, Benoit ; Kligys, Skirmantas u. a.: Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference / Google, Inc. 2017. – Forschungsbericht

[15] Jouppi, Norman P. ; Young, Cliff u. a.: In-Datacenter Performance Analysis of a Tensor Processing Unit / Google, Inc. 2017. – Forschungsbericht

[16] Langley, Pat: *Elements of Machine Learning*. Morgan Kaufmann, 1996

[17] Lecun, Yann ; Cortes, Corinna ; Burges, Christopher J. C.: *THE MNIST DATABASE of handwritten digits*. – URL http://yann.lecun.com/exdb/mnist/. – Zugriffsdatum: 11.11.2018

[18] McMahan, Brendan ; Ramage, Daniel: *Federated Learning: Collaborative Machine Learning without Centralized Training Data*. April 2017. – URL https://ai.googleblog.com/2017/04/federated-learning-collaborative.html. – Zugriffsdatum: 08.11.2018

[19] Mitchell, Tom M.: *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997

[20] Ndjountche, Tertulien: *Digital Electronics 1: Combinational Logic Circuits*. John Wiley and Sons, 2016

[21] NVIDIA Corporation: *CUDA*. – URL https://www.nvidia.de/object/cuda-parallel-computing-de.html. – Zugriffsdatum: 10.11.2018

[22] NVIDIA Corporation: *NVDLA*. – URL http://nvdla.org/. – Zugriffsdatum: 10.11.2018

[23] NVIDIA Corporation: *NVIDIA JETSON AGX-SYSTEME*. – URL https://www.nvidia.com/de-de/autonomous-machines/embedded-systems-dev-kits-modules/. – Zugriffsdatum: 10.11.2018

[24] NVIDIA Corporation: *NVIDIA Jetson AGX Xavier Developer Kit*. – URL https://developer.nvidia.com/embedded/buy/jetson-xavier-devkit. – Zugriffsdatum: 10.11.2018

[25] NVIDIA Corporation: *Tensor Core*. – URL https://www.nvidia.com/de-de/data-center/tensorcore/. – Zugriffsdatum: 10.11.2018

[26] Oberstar, Erick L.: *Fixed-Point Representation and Fractional Math*. 2007

[27] Oh, Kyoung-Su ; Jung, Keechul: GPU implementation of neural networks / School of Media, College of Information Science, Soongsil University. 2004. – Forschungsbericht

[28] Overton, Michael L.: *Numerical Computing with IEEE Floating point Arithmetic*. SIAM, 2001

[29] Schröder, Hartmut: *Mehrdimensionale Signalverarbeitung: Band 1: Algorithmische Grundlagen für Bilder und Bildsequenzen*. Springer-Verlag, 2013

[30] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning: An Introduction*. The MIT Press, 2015

[31] UCSB ArchLab: *OpenTPU Project*. – URL https://github.com/UCSBarchlab/OpenTPU. – Zugriffsdatum: 10.11.2018

[32] Vahid, Frank: *Digital Design with RTL Design, Verilog and VHDL*. John Wiley and Sons, 2010

[33] Xilinx, Inc.: *7 Series FPGAs Memory Resources*. 2016

[34] Xilinx, Inc.: *7 Series DSP48E1 Slice*. 2018

[35] Xilinx, Inc.: *UltraFast High-Level Productivity Design Methodology Guide*. 2018

[36] Xilinx, Inc.: *Vivado Design Suite User Guide Implementation*. 2018

[37] Xilinx, Inc.: *Vivado Design Suite User Guide Logic Simulation*. 2018

[38] Xilinx, Inc.: *Vivado Design Suite User Guide Synthesis*. 2018

[39] Zhang, C. ; Woodland, P.C.: DNN Speaker Adaption Using Parameterised Sigmoid and ReLU Hidden Activation Functions / Cambridge University Engineering. 2016. – Forschungsbericht

A Anhang

A.1 Floorplan

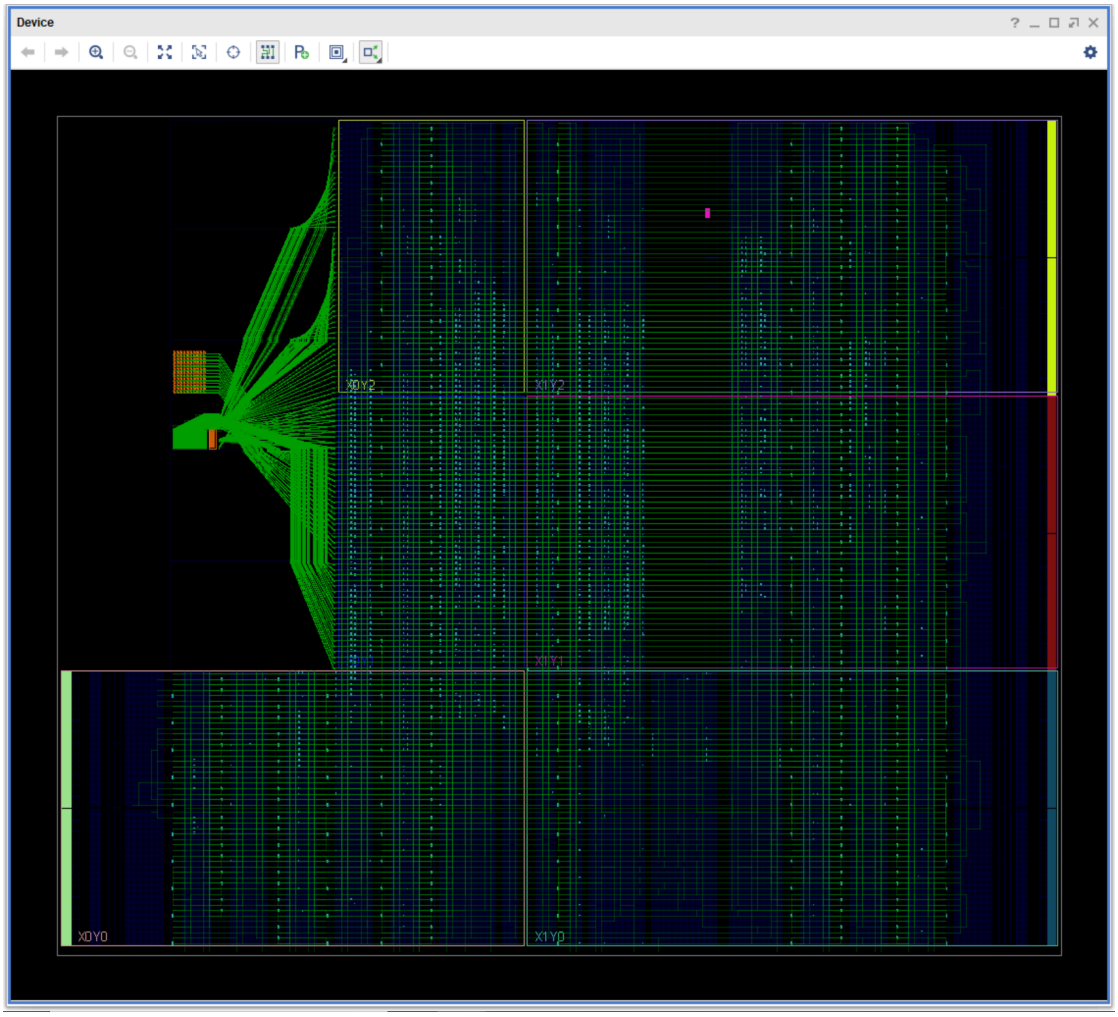


Abbildung A.1: Floorplan der FPGA-Nutzung

A.2 RTL-Analyse

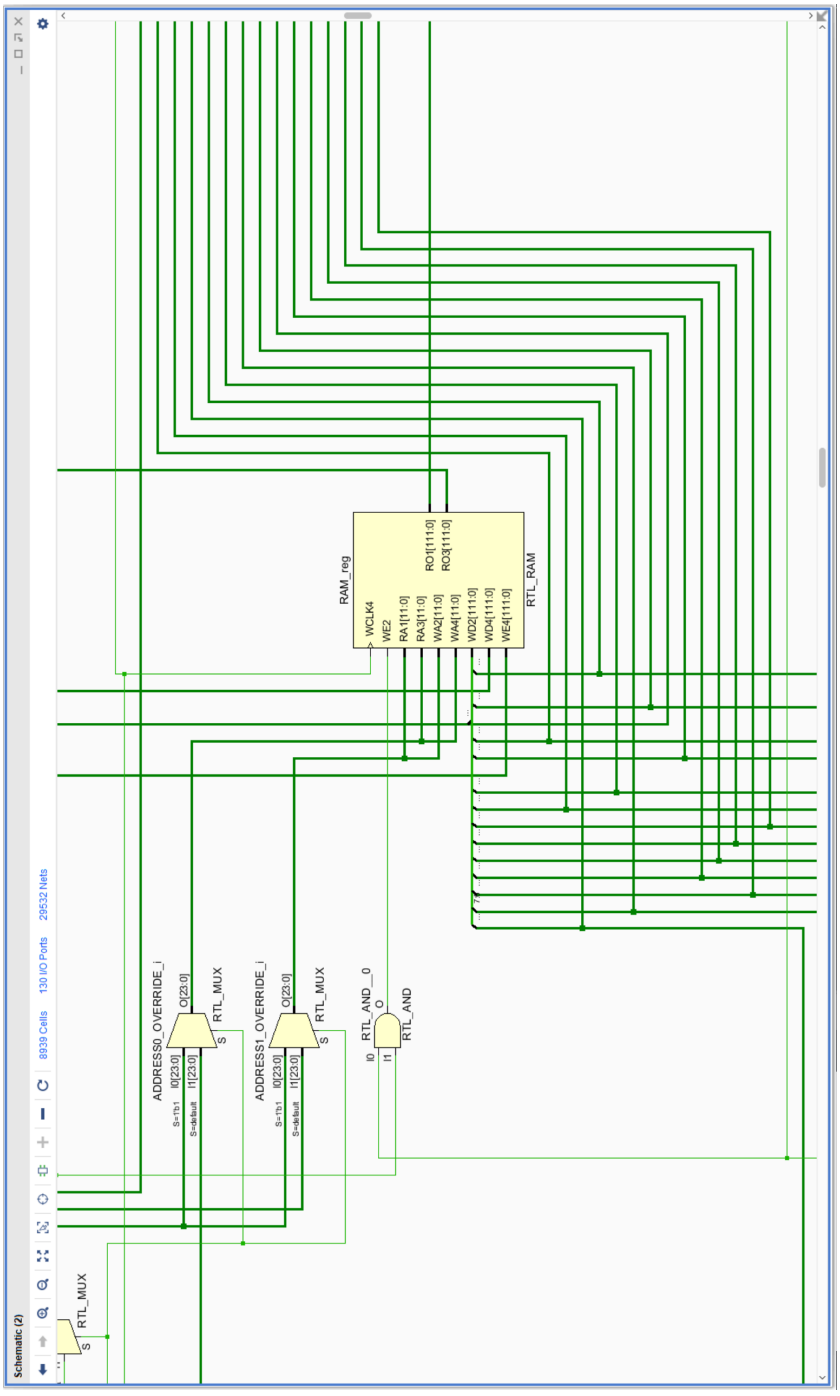


Abbildung A.2: Speicher des Unified Buffers in der RTL-Analyse

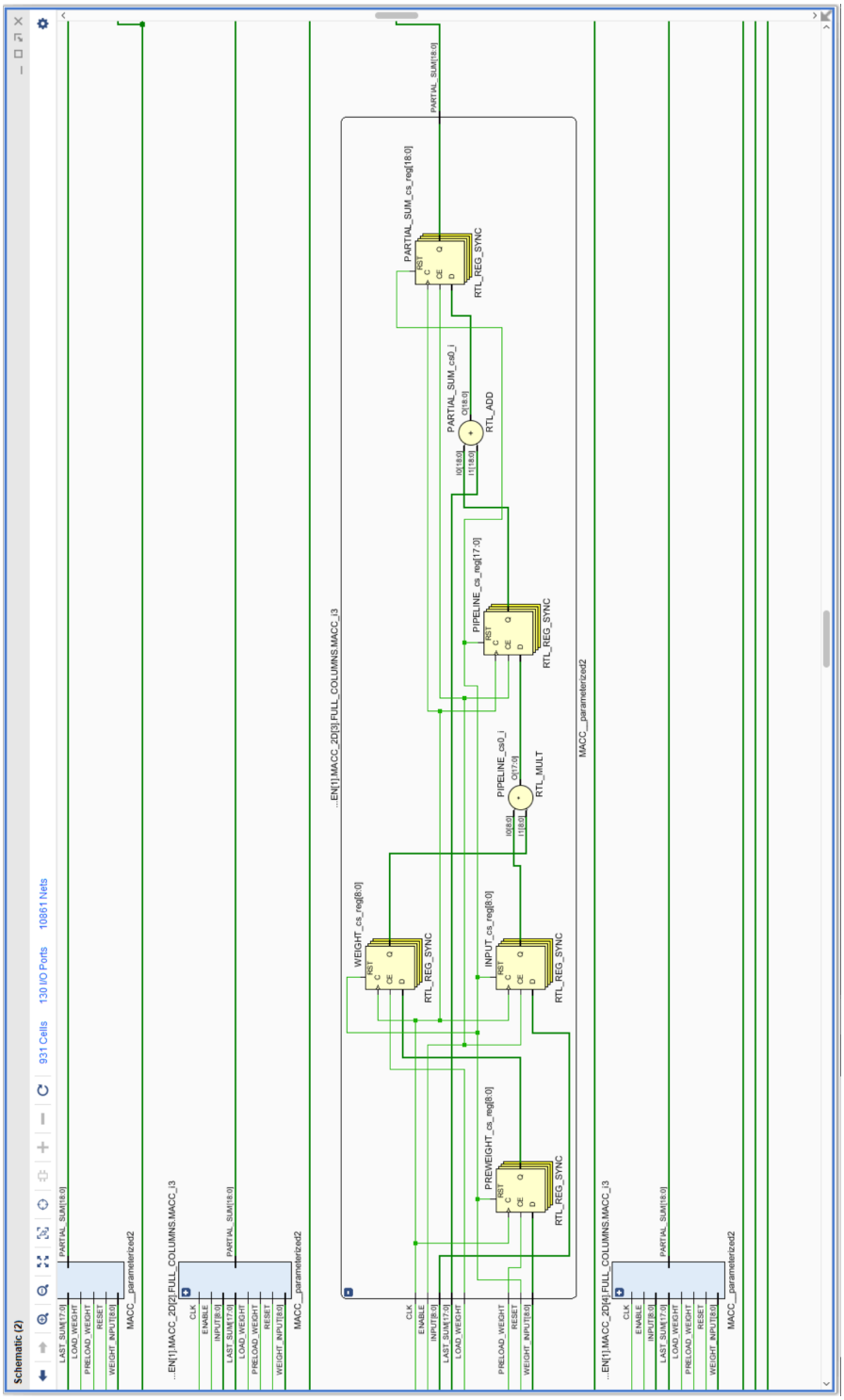


Abbildung A.3: Multiply-Add Einheiten der MXU in der RTL-Analyse

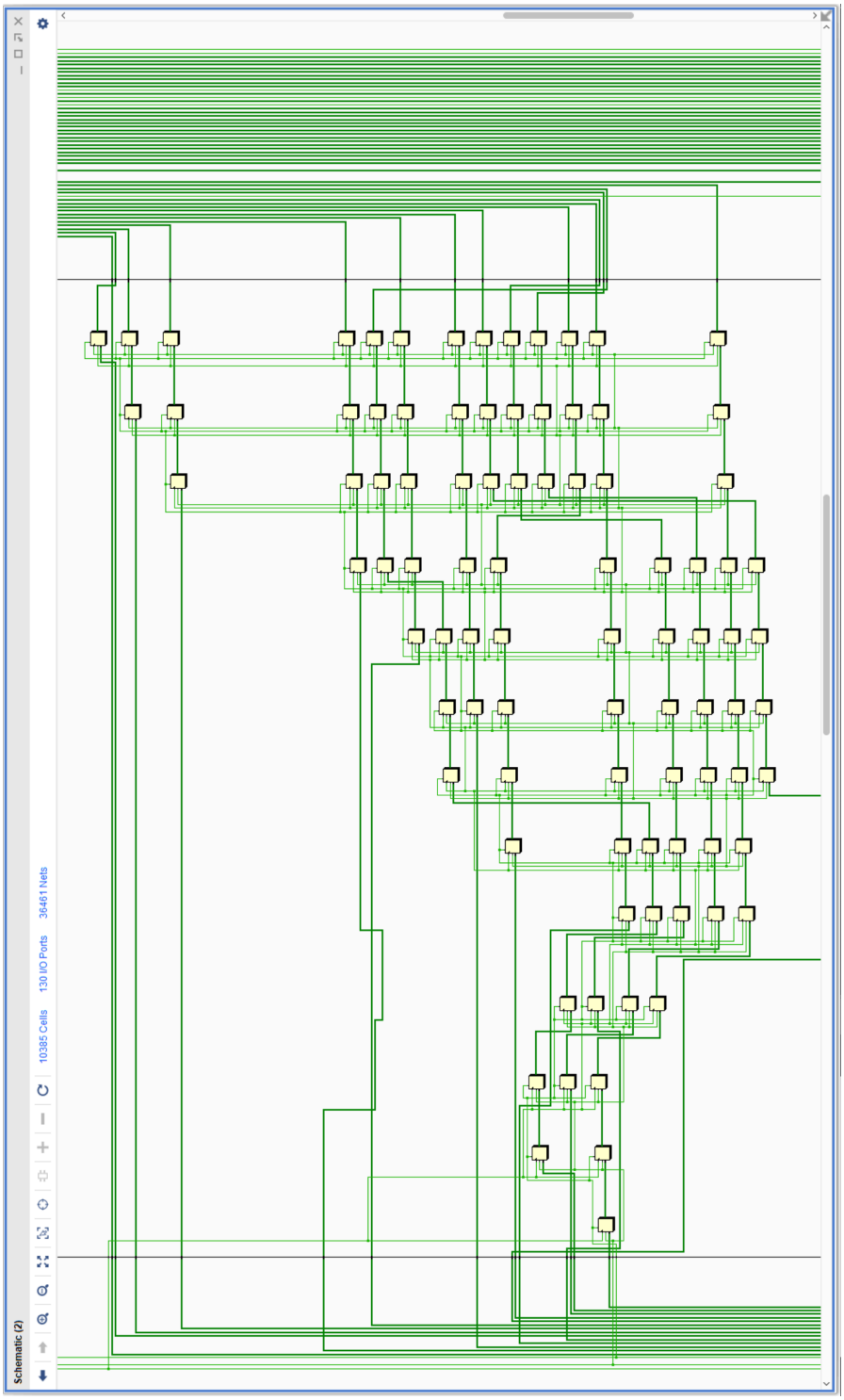


Abbildung A.4: Systolic Data Setup in der RTL-Analyse

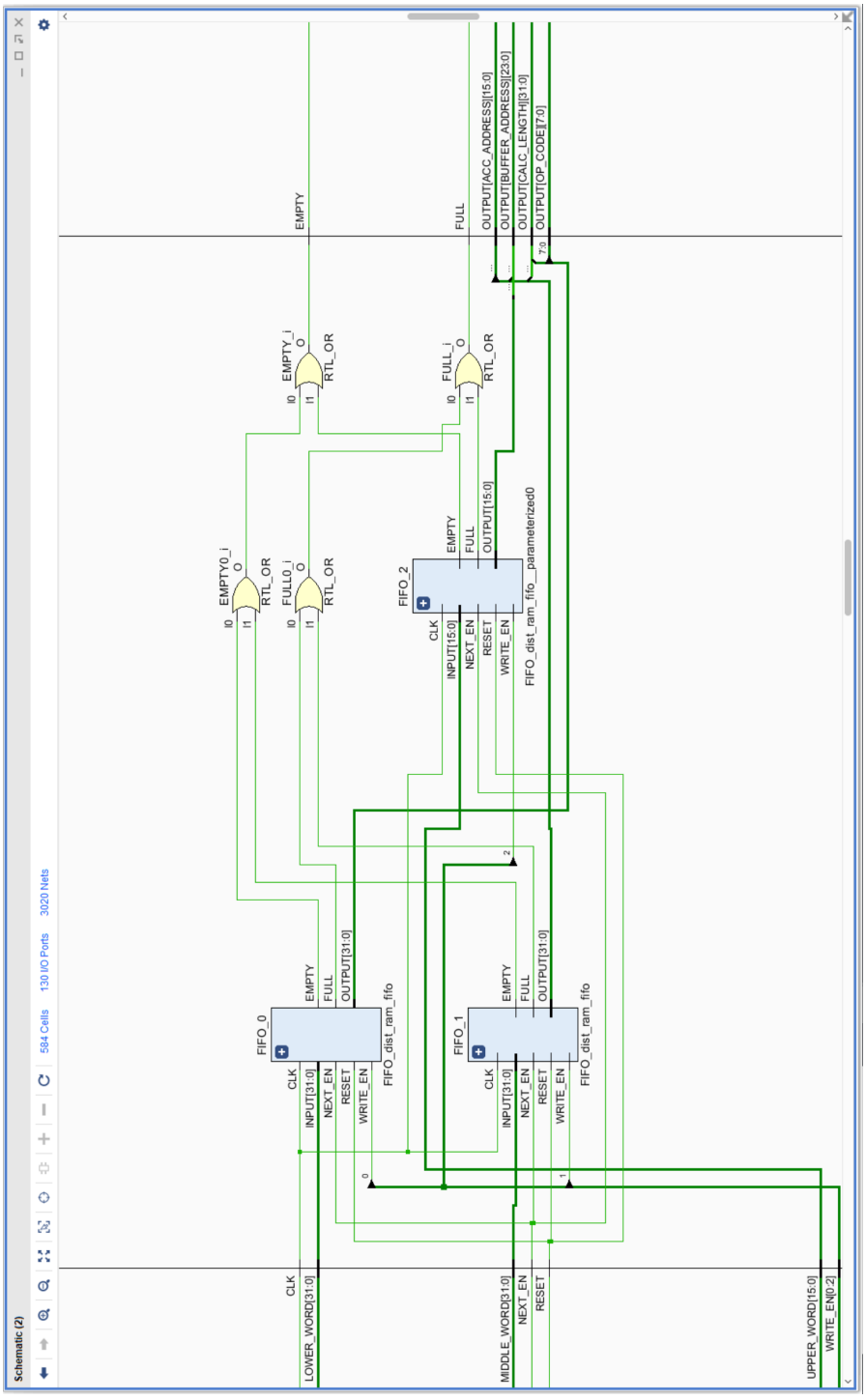


Abbildung A.5: FIFO-Speicher (3 32-/16-Bit Komponenten) für Instruktionen in der RTL-Analyse

A.3 Simulation

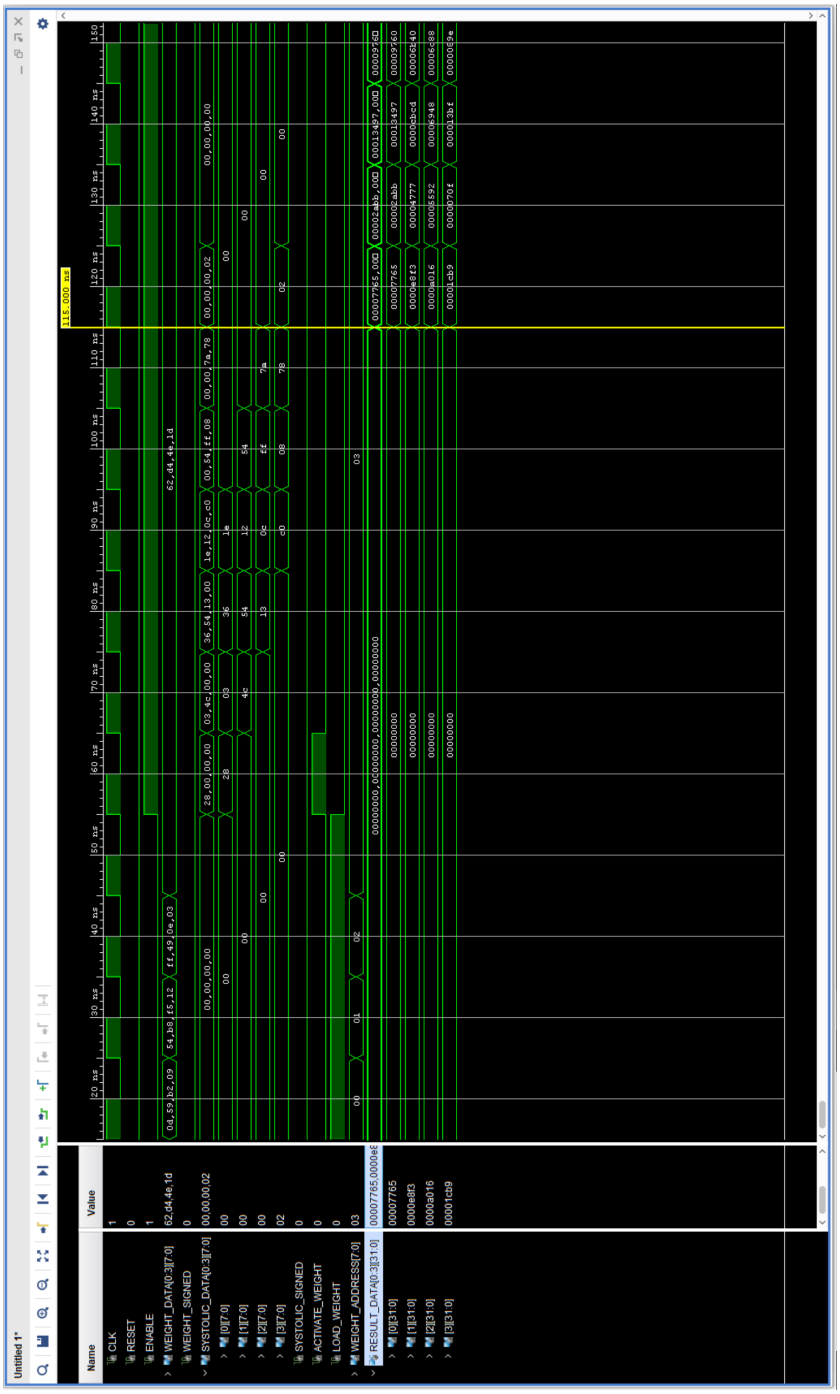


Abbildung A.6: Simulation einer 4×4 MXU

Glossar

Application Specific Integrated Circuit Application Specific Integrated Circuits sind auf eine Anwendung spezialisierte, integrierte Schaltkreise und sind demnach nicht mehr veränderbar.

Augmented Reality Augmented Reality bezeichnet die Fähigkeit eines Systems virtuelle Objekte in der realen Welt abzubilden. Häufig werden hierzu 3D Objekte über ein Kamerabild gerendert.

Board Support Package Ein Board Support Package definiert Bibliotheken für die Unterstützung von bestimmter Hardware.

Complex Instruction Set Computer Complex Instruction Set Computer bezeichnet Rechner, die einen komplexen Instruktionssatz aufweisen.

Direct Memory Access Direct Memory Access erlaubt Peripherie-Geräten die direkte Kommunikation mit dem Hauptspeicher, ohne einen Umweg über die CPU.

Embedded System Systeme, häufig integriert für Überwachungs-, Steuerungs- und Regelungsaufgaben oder Signal- und Datenverarbeitung. Dabei ist nicht unbedingt erkennbar, dass es sich um ein Computersystem handelt.

Floorplan Eine schematische Darstellung der Platzierung von Hardwareressourcen auf einem Chip.

Internet of Things Internet of Things bezeichnet physische Gegenstände, die miteinander vernetzt werden.

Reduced Instruction Set Computer Reduced Instruction Set Computer bezeichnet Rechner, mit einfacherem Instruktionssatz, was zur Erhöhung der Rechenleistung führen kann.

Software Development Kit Eine Sammlung von Programmierwerkzeugen und Bibliotheken, die der Entwicklung dienen.

System-on-a-Chip Ein System-on-a-Chip ist ein vollwertiges Computersystem auf einem einzigen Chip/ASIC.

Virtual Reality Virtual Reality bezeichnet die Wahrnehmung der Wirklichkeit in einer virtuellen Umgebung.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Ort	Datum	Unterschrift im Original