

Bachelorarbeit

Parallele Berechnung der Mandelbrotmenge mit grafischer Ausgabe auf einem FPGA

von

Daniel Wagner



Betreuer:

Dr. Jano Gebelein

Lehrstuhl für Infrastruktur und Rechnersysteme in
der Informationsverarbeitung

Institut für Informatik

Johann Wolfgang Goethe-Universität
Frankfurt am Main

9. August 2017

Inhaltsverzeichnis

1	Einleitung	8
1.1	Motivation	8
1.2	Arbeitsumgebung	8
1.2.1	Software	8
1.2.2	Hardware	9
1.3	Ziele	10
2	Stand der Technik	11
2.1	FPGAs	11
2.2	Komplexe Zahlen	12
2.2.1	Rechnen mit komplexen Zahlen	12
2.3	Mandelbrotmenge	13
2.3.1	Definition	13
2.3.2	Geschichtliches	14
2.3.3	Erzeugen der Bilder	15
3	Ansatz	17
3.1	Modularisierung	17
3.1.1	Modularisierung in VHDL	17
3.2	Automaten	17
3.2.1	Moore-Automaten in VHDL-Syntax	18
3.3	IP-Cores	19
3.4	Systemeigenschaften	19
4	Implementierung	21
4.1	Top-Modul	21
4.2	Math-Modul	24
4.3	Komplexe-Arithmetic Logic Unit (ALU)	27
4.4	Joystick-Controller	30
4.5	LED-Controller	30
4.6	Hierarchie	31
5	Ergebnisse	33
5.1	Der Hardwareentwurf	33
5.2	Joystick und Display	34
5.3	Beurteilung der Parallelisierungseffizienz	34
5.3.1	Bottleneck Math-Modul	35

6	Zusammenfassung und Ausblick	38
6.1	Was war erfolgreich?	38
6.2	Wie kann aufbauend verfahren werden?	38
6.3	Mögliche Anwendungszwecke der Arbeit	38
7	Anhang	39

Abkürzungsverzeichnis

ALU Arithmetic Logic Unit

Ein Hardwarebaustein, der ausschließlich zur logischen Berechnung von arithmetischen Ausdrücken dient.

ASCII American Standard Code for Information Interchange

Eine rechnerinterne binär-Kodierung für Zeichen, die zum Beispiel von der Tastatur eingegeben werden können.

BRAM Block Random Access Memory (RAM)

Eine konfigurierbare Unterklasse des RAMs, die mit verschiedenen Wortbreiten genutzt werden kann.

CPU Central Processing Unit

Ein Hardwarebaustein, der Programmcode ausführt. Er beherbergt alle primäre Recheneinheiten und koordiniert diese.

DSP Digital Signal Processor

EPROM Erasable Programmable Read-Only Memory

Siehe Programmable Read-Only Memory (PROM), allerdings besteht die Möglichkeit, den beschriebenen Speicher wieder freizugeben.

FPGA Field Programmable Gate Array

Ein Chip, in den logische Schaltungen geladen werden können. Somit ist er als konfigurierbare Hardware zu verstehen.

GAL Generic Array Logic

Ein Halbleiterbaustein, der über komplett konfigurierbare Verknüpfungen zwischen seinen Ein- und Ausgängen verfügt.

HDL Hardware Description Language

Eine hardwarebeschreibende, so genannte „low-level“ Beschreibungssprache, mit der integrierte Schaltungen beschrieben werden können. Sie kann genutzt werden, um ein Field Programmable Gate Array (FPGA) zu konfigurieren.

ID Identifikationsnummer

Eine Zahl, die eindeutig mit einem Objekt oder einer Entität korrespondiert.

IDE Integrated Development Environment

Ein Programm, das Programmierern ermöglicht, Quellcode zu kompilieren und zu testen. Darüber hinaus werden hilfreiche Werkzeuge für diesen Vorgang bereitgestellt.

IP-Core Intellectual Property-Core

Ein Funktionsblock, der vom Hersteller angeboten wird, um Funktionalitäten in ein eigenes Design einzubringen.

LED Light Emitting Diode

Ein Halbleiter-Bauelement, das bei Stromfluss in Durchlassrichtung ein Licht ausstrahlt.

LUT Look-Up Table

MHz Megahertz

1 Megahertz $\hat{=}$ 1.000.000 Hertz. Ein Megahertz entspricht somit dem Ablauf einer Million Zyklen pro Sekunde.

PCB Printed Circuit Board

Eine Leiterplatte, auf der integrierte Schaltungen physikalisch realisiert werden.

Pmod Peripheral Module

Ein Schnittstellen-Standard. Er dient in dieser Bachelorarbeit als Schnittstelle zwischen dem FPGA und dem Joystick oder dem Light Emitting Diode (LED) Display.

PROM Programmable Read-Only Memory

Ein Festwertspeicher, in den einmalig Daten geschrieben werden können. Diese Daten bleiben auch ohne Versorgungsspannung erhalten.

RAID Redundant Array of Independent Disks

Bezeichnet ein Verfahren, bei dem Massenspeicher in einen Verbund geschaltet werden. Dies kann, je nach Umsetzung, zu einer höheren Datensicherheit oder Schreib- oder Lesegeschwindigkeit führen.

RAM Random Access Memory

Bezeichnet einen Speicher, der über eine Speicheradresse direkt angesprochen werden kann. Er dient meist als Hauptspeicher für Recheneinheiten.

SPI Serial Peripheral Interface

Ein Bus-System, das dazu dient, mit Peripherie zu kommunizieren.

UART Universal Asynchronous Receiver Transmitter

Schnittstelle, die zum seriellen Senden und Empfangen von Daten dient. In dieser Bachelorarbeit wird sie zur Kommunikation zwischen dem Computer und dem FPGA genutzt.

VHDL Very High Speed HDL

Eine Hardwarebeschreibungssprache. Sie wird in dieser Bachelorarbeit zum Konfigurieren des FPGAs genutzt.

Abbildungsverzeichnis

1.1	Das FPGA	9
1.2	Der Joystick	10
1.3	Das Display	10
2.1	Die komplexe Ebene	12
2.2	Visualisierung der Julia-Mengen	15
2.3	Erste Visualisierung der Mandelbrotmenge	15
2.4	Die Mandelbrotmenge	16
4.1	Das Schaltbild des Top-Moduls	21
4.2	Der Automat des Top-Moduls	22
4.3	Das Schaltbild des Math-Moduls	24
4.4	Der Automat des Math-Moduls	25
4.5	Das Schaltbild der komplexen ALU	27
4.6	Der Automat der komplexen ALU	28
4.7	Die Hierarchie der Top-Level Ebene	31
4.8	Die Hierarchie des Math-Moduls	32
5.1	8 ALUs auf dem Artix-7	33
5.2	Display und Joystick am FPGA	34
5.3	Graph der exponentiellen Regression	36
5.4	Das in der Simulation berechnete Bild	37
5.5	Weitere Bilder	37
7.1	Place-Design mit 16 ALUs nicht möglich	39
7.2	Entwurf mit 2 ALUs in der Simbox	39
7.3	Entwurf mit 4 ALUs in der Simbox	39
7.4	Entwurf mit 8 ALUs in der Simbox	40

Tabellenverzeichnis

5.1	Ergebnisse der Simulation	35
5.2	Exponenten-Regression über den Speed-Up Faktor	36

Zusammenfassung

In dieser Bachelorarbeit wird ein VHDL-Entwurf vorgestellt, der parallel eine grafische Ausgabe der Mandelbrotmenge berechnet. Es wird auf das Rechnen mit komplexen Zahlen, sowie die Definition der Mandelbrotmenge, Automaten in VHDL-Syntax und die Möglichkeit durch Modularisierung eine einfache, theoretisch beliebig-fache Parallelisierung zu erzielen, eingegangen. Zudem wird die Problematik der praktisch begrenzten Anzahl der konfigurierbaren Logikzellen auf dem FPGA aufgezeigt. Abschließend werden Simulationsergebnisse genutzt, um die Effizienz der Parallelisierung des Entwurfs einschätzen zu können.

Abstract (English)

A VHDL implementation to parallelly compute an visual image of the Mandelbrot set is being presented in the following bachelor thesis. The definition of the aforementioned mentioned set, the laws of complex numbers, as well as the implementation of finite state machines in VHDL syntax and lastly the advantages of modularized programming regarding parallelization is being concerned. Furthermore, a problem regarding the limited amount of configurable cells inside the FPGA is shown up in this thesis. Finally, the data resulting from the simulatoin is evaluated in order to estimate the efficiency provided by parallelizing the design.

Kapitel 1

Einleitung

1.1 Motivation

In dem Modul „Mathe für die Informatik 1: Analysis und lineare Algebra“, das Teil der Basismodule des Bachelors an der Goethe Universität Frankfurt am Main ist, wird das Rechnen mit komplexen Zahlen vorgestellt. Eine praktische Anwendung zur Ergänzung der Lehrveranstaltung [1] kann durch die hier geleistete Vorarbeit dargestellt werden.

Einer der bekanntesten Anwendungsfälle der komplexen Zahlen findet sich in Verbindung mit der Mandelbrotmenge. Die Darstellung ist, verglichen mit anderen mathematischen oder geometrischen Konstrukten, aufgrund der unendlichen Vielfalt eher bemerkenswert.

In einem anderen Basismodul des Bachelor Studienganges, dem „Grundlagen von Hardwaresystemen - Praktikum“, werden FPGAs vorgestellt. Es wird dort die konfigurierbare Hardware eingeführt, die es möglich macht, jegliche digitale Berechnungen zu implementieren. Dazu wird eine sogenannte Hardware Description Language (HDL) verwendet. In diesem Praktikum wird die Hardwarebeschreibungssprache Very High Speed HDL (VHDL) gelehrt [1].

Um das aus diesen beiden Veranstaltungen gelehrt Wissen praktisch zusammenzuführen, liegt es nahe, die Berechnung der Mandelbrotmenge auf einem solchen FPGA zu implementieren und sie in ihrer einzigartigen Vielfalt auszugeben. Darüber hinaus besteht ein Forschungsspekt darin, herauszufinden, inwiefern sich diese Berechnung parallelisieren lässt.

1.2 Arbeitsumgebung

1.2.1 Software

Der VHDL-Code wurde in der Xilinx Vivado 2016.4 Umgebung, installiert auf einem Windows 7 Laptop, entwickelt. Die genutzte Hardwarebeschreibungssprache ist VHDL. Vivado allein übernimmt hierbei alle wichtigen Rollen wie das Aufzeigen von Syntaxfehlern als Integrated Development Environment (IDE), das Synthetisieren des Codes zu einer Netzliste, das Routen des VHDL Entwurfs, sowie das Erzeugen eines Schaltkreislays. Ebenfalls wird das Generieren des Bitstreams und das Konfigurieren der Zielplattform, des FPGAs, von Vivado übernommen.

1.2.2 Hardware

1.2.2.1 FPGA

Als Zielplattform dient das von Digilent entwickelte Nexys 4 DDR Board. Das Nexys 4 basiert auf dem Xilinx Artix-7 FPGA Chip und bietet weitreichende Peripherie, mit der es möglich wird, die entwickelten Anwendungen sehr vielseitig zu implementieren. Wesentlich sind die konfigurierbaren Zellen auf dem Artix-7 Chip, die es ermöglichen das Board mit einem Entwurf zu laden. Dazu verfügt der Chip über FlipFlops, Block RAM (BRAM), Digital Signal Processor (DSP)s, und Look-Up Table (LUT)s [2].

In dieser Bachelorarbeit sind die relevanten Schnittstellen:

- Universal Asynchronous Receiver Transmitter (UART) Schnittstelle
- 16 LEDs
- 16 Schalter
- 5 Knöpfe
- 2 Peripheral Module (Pmod) Schnittstellen

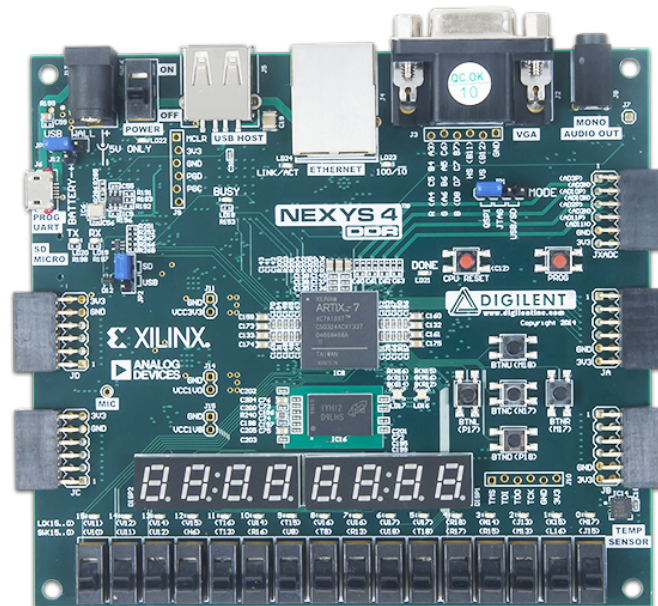


Abbildung 1.1: Digilent Nexys4 Board mit Artix-7 Chip

1.2.2.2 Joystick

Zur Eingabe wird ein Pmod Joystick von Digilent verwendet. Dieser verfügt für die Eingabe über 3 Knöpfe, sowie einen 2 Achsen Joystick. Außerdem werden 2 LEDs zur Ausgabe genutzt. Der Joystick wird über ein Serial Peripheral Interface (SPI) betrieben.

Der Stick dient zur primären Navigation durch die Mandelbrotmenge. Wird er bis zum Anschlag in eine der vier zum Printed Circuit Board (PCB) senkrechten Richtungen bewegt, ändert sich die Position des sichtbaren Ausschnitts der Mandelbrotmenge. Durch Herunterdrücken des Joysticks wird zoomt.

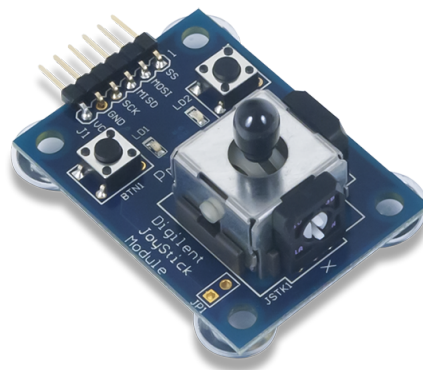


Abbildung 1.2: Digilent Pmod Joystick [3]

1.2.2.3 LED Display

Zur grafischen Ausgabe der berechneten Mandelbrotmenge wird ein Pmod LED Display von Digilent verwendet. Das Display verfügt über 6144 Pixel, 94 Pixel in der Breite und 64 Pixel in der Höhe. Jeder Pixel hat eine 16 Bit Farbaufösung, das heißt, jeder Pixel kann bis zu 65.536 verschiedene Farben darstellen.

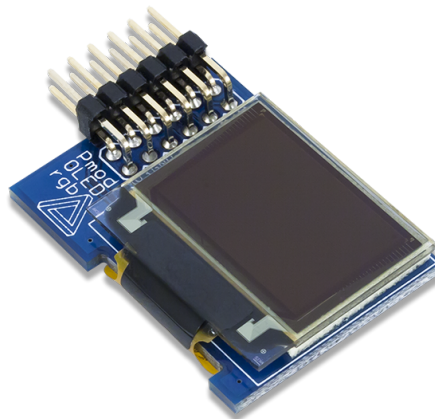


Abbildung 1.3: Digilent Pmod LED Display, das Zielgerät der grafischen Ausgabe.[4]

1.3 Ziele

Ziele dieser Bachelorarbeit bestehen darin,

- einen Hardwareentwurf für FPGAs zu entwickeln, der darauf spezialisiert ist, die Mandelbrotmenge zu erkunden
- einen Joystick zur Eingabe einzubinden, um durch die Mandelbrotmenge navigieren zu können und diese auf einem Display auszugeben
- zu einer Beurteilung über Effizienz der Parallelisierbarkeit der Berechnung zu kommen

Kapitel 2

Stand der Technik

2.1 FPGAs

Das erste Field Programmable Gate Array wurde 1985 von Ross Freeman entwickelt [5]. Davor allerdings stand eine lange Entwicklungszeit, anfangend 1960 mit dem ersten Feldeffekttransistor über den ersten konfigurierbaren Speicher, dem 1970 entwickelten PROM und einem Jahr später, 1971, dem wiederbeschreibbaren Erasable Programmable Read-Only Memory (EPROM) bis hin zum Generic Array Logic (GAL) im Jahre 1983 [6].

Seitdem wird an immer leistungstärkeren FPGAs geforscht, um immer vielseitigere und anspruchsvollere Entwürfe realisieren zu können. Aktuelle FPGAs sind soweit konfigurierbar, dass sie zu einer eigenständigen Central Processing Unit (CPU) konfiguriert werden können. Dem Programmierer sind hierbei keine konventionellen Auflagen gegeben. Er darf frei nach Belieben seine CPU entwerfen und sie ganz präzise für einen bestimmten Anwendungsfall zuschneiden. Dadurch ergibt sich ein grundlegender Vorteil: wird zum Beispiel in einem Anwendungsfall ausschließlich dividiert, so besteht die Möglichkeit, alle auf dem FPGA verfügbaren Logikgatter auf das Dividieren zu optimieren. Eine CPU, die in Desktop-Rechnern verbaut ist, verfügt nur über universelle, statische logische Einheiten. Diese allerdings in großer Zahl, sodass sie beliebige Anwendungen in schnellem Kontextwechsel ausführen kann. Das macht sie zwar vielseitig einsetzbar, aber die gesamte Rechenkraft kann nicht ausschließlich dem Dividieren zugeordnet werden, sodass in diesem beispielhaften Anwendungsfall der Einsatz eines FPGA sinnvoller ist. Somit lässt sich prinzipiell schließen: Der Anwendungsfall entscheidet, ob eine CPU oder ein FPGA Verwendung findet. Kann eine digitale Schaltung eine gegebene Aufgabe lösen, eignet sich ein FPGA mit entsprechender Konfiguration. Steht allerdings bei Inbetriebnahme des Gerätes der Anwendungsfall nicht fest, so eignet sich eine „Alleskönner“-CPU, die über diverse Recheneinheiten verfügt.

Eine konkrete Anwendung für FPGAs findet sich in der Forschung an Redundant Array of Independent Disks (RAID)-Controllern, die ohne Batterie auskommen sollen [7]. Neue Technologie, die diese Ansprüche erfüllen soll, kann flexibel mit einem FPGA getestet und eingesetzt werden. Die Möglichkeit, FPGAs neu zu konfigurieren, erspart dabei kostspieliges Austauschen der Hardware.

In dieser Bachelorarbeit wird daher das FPGA so konfiguriert, dass es parallel die Mandelbrotmenge (siehe Kapitel 2.3) berechnet.

Für nähere Informationen zu dem in dieser Bachelorarbeit verwendeten FPGA, siehe Kapitel 1.2.2.1 (FPGA).

2.2 Komplexe Zahlen

Die Menge der komplexen Zahlen \mathbb{C} ist eine Erweiterung der Menge der reellen Zahlen \mathbb{R} um eine imaginäre Einheit i , die folgende elementare Eigenschaft aufweist:

$$i^2 = -1$$

Dadurch erhält zum Beispiel die Gleichung $x^2 + 1 = 0$, die im reellen Zahlenraum \mathbb{R} keine Lösung hat, eine Lösungsmenge $\mathbb{L} = \{i\}$. Damit hat jede algebraische Gleichung positiven Grades über die komplexen Zahlen \mathbb{C} eine Lösung.

Auch in der Physik oder der Elektrotechnik finden die komplexen Zahlen \mathbb{C} Anwendung. Darauf wird aber in dieser Arbeit nicht näher eingegangen.

Alle komplexen Zahlen $c \in \mathbb{C}$ sind von der Form: $c = a + b \cdot i$, wobei $a, b \in \mathbb{R}$ und i die komplexe Einheit darstellt. a ist als Realteil einer komplexen Zahl c , und $b \cdot i$ als Imaginärteil von c zu verstehen.

Die komplexen Zahlen lassen sich in einem zweidimensionalen Graphen darstellen, indem die horizontale Achse den Realteil, und die vertikale Achse den Imaginärteil der komplexen Zahl beschreibt. Dieser Graph nennt sich auch die „Komplexe Ebene“.

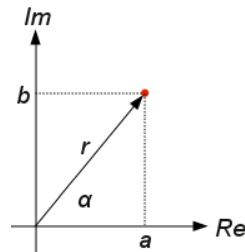


Abbildung 2.1: Die komplexe Ebene und eine komplexe Zahl $a + b \cdot i$ [8]

2.2.1 Rechnen mit komplexen Zahlen

Es werden die folgenden arithmetischen Operationen auf komplexen Zahlen betrachtet:

- Addition
- Subtraktion
- Multiplikation
- Betragsrechnung

2.2.1.1 Addition

Seien $c_0 := a + b \cdot i$ und $c_1 := c + d \cdot i$ zwei komplexe Zahlen, so ist ihre Summe

$$c_0 + c_1 = (a + b \cdot i) + (c + d \cdot i) = (a + c) + (b + d) \cdot i$$

Vereinfacht lässt sich sagen, dass sich jeweils die Realteile, sowie die Imaginärteile der beiden komplexen Zahlen addieren.

2.2.1.2 Subtraktion

Die Subtraktion verläuft analog zur Addition. Seien weiterhin $c_0 := a + b \cdot i$ und $c_1 := c + d \cdot i$ zwei komplexe Zahlen, so ist ihre Differenz

$$c_0 - c_1 = (a + b \cdot i) - (c + d \cdot i) = (a - c) + (b - d) \cdot i$$

Auch hier ist zu beobachten, dass sich jeweils die Realteile, sowie die Imaginärteile voneinander subtrahieren.

2.2.1.3 Multiplikation

Seien weiterhin $c_0 := a + b \cdot i$ und $c_1 := c + d \cdot i$ zwei komplexe Zahlen, so ist ihr Produkt

$$c_0 \cdot c_1 = (a + b \cdot i) \cdot (c + d \cdot i) = (a \cdot c - b \cdot d) + (a \cdot d + b \cdot c) \cdot i$$

Das Quadrat einer komplexen Zahl $c_0 := a + b \cdot i$ lässt sich vereinfacht wie folgt darstellen:

$$c_0^2 = (a + b \cdot i) \cdot (a + b \cdot i) = a^2 + 2 \cdot a \cdot b \cdot i + (b \cdot i)^2 = a^2 + 2 \cdot a \cdot b \cdot i - b^2 = a^2 - b^2 + 2 \cdot a \cdot b \cdot i$$

2.2.1.4 Betragsrechnung

Sei $c_0 := a + b \cdot i$ eine komplexe Zahl, so lässt sich ihr Betrag, also die Distanz von c_0 zum Ursprung der komplexen Ebene wie folgt berechnen:

$$|a + b \cdot i| = \sqrt{a^2 + b^2}$$

2.3 Mandelbrotmenge

2.3.1 Definition

Die Mandelbrotmenge \mathbb{M} ist eine Menge von komplexen Zahlen in der komplexen Ebene. Eine komplexe Zahl c , ist genau dann in der Mandelbrotmenge, wenn sie unter unendlich langer Iteration der Mandelbrotschen Formel zu einem bestimmten Wert konvergiert. Existiert kein Konvergenzwert, so divergiert c unter Iteration der Mandelbrotschen Formel und liegt nicht in der Mandelbrotmenge. Die Formel, die auf einer komplexen Zahl iteriert wird, ist wie folgt definiert:

Definition 1: *Mandelbrotsche Formel*

$$\begin{aligned} z_0 &= 0 + 0 \cdot i \\ z_{n+1} &= z_n^2 + c \end{aligned}$$

Sei für eine beliebige Zahl $c \in \mathbb{C}$ festzustellen, ob sie divergiert oder konvergiert, so wird diese Formel auf c so lange iteriert, bis der Betrag von c einen bestimmten Wert $absValMax$ überschreitet. Dieser Wert ist frei aus $\mathbb{R}_{>0}$ zu wählen.

2.3.1.1 Iterieren

- z_0 : Gemäß der Mandelbrotschen Formel ist der Wert jeder komplexen Zahl c in der ersten Iteration stets 0.
 $\implies z_0 = 0 + 0 \cdot i$
- z_1 : In der zweiten Iteration wird der Wert von z_0 quadriert. Anschließend wird der Wert von der Zahl c addiert.
 $\implies z_1 = 0^2 + c = c$
- In der zweiten Iteration wird z_1 , also c selbst, quadriert. Danach wird wieder die Konstante c addiert.
 $\implies z_2 = c^2 + c$
- In der dritten, und jeder folgenden Iteration, wird ebenso weiter verfahren.
 $\implies z_3 = (c^2 + c)^2 + c$
 $\implies z_4 = ((c^2 + c)^2 + c)^2 + c$
 $\implies z_5 = (((c^2 + c)^2 + c)^2 + c)^2 + c$
 $\implies z_6 = \dots$

2.3.1.2 Abbruchkriterien: Konvergenz und Divergenz

Mit der Iteration wird so lange verfahren, bis der Betrag von c die gewählte obere Schranke *absValMax* überschreitet. Sollte allerdings der Wert von c unter Iteration zu einem bestimmten Wert kleiner als *absValMax* konvergieren, so kann *absValueMax* nie erreicht werden. Es liegt der erste Fall von Konvergenz vor. Die Zahl c kann aber auch unter Iteration zwischen zwei Werten, die beide jeweils kleiner als *absValMax* sind, alternieren, oder sich unendlich lange chaotisch um eine komplexe Zahl kleiner als *absValMax* bewegen. Auch in diesen Fällen nimmt man an, dass c konvergiert. Um in den Fällen der Konvergenz nicht unendlich lange zu iterieren, wird eine Obergrenze der Iterationsanzahl, ein Wert *iterationMax* benötigt. Wird dieser erreicht ohne Divergenz festzustellen, so stellt das das hinreichende Kriterium für Konvergenz dar und das Iterieren auf c wird beendet. Sollte dies für eine komplexe Zahl c gelten, so liegt sie in der Mandelbrotmenge.

2.3.2 Geschichtliches

Die Mandelbrotmenge wurde nach Benoît Mandelbrot, einem französisch-US-amerikanischen Mathematiker benannt [9]. Eine erste Veröffentlichung, in der eine farbige und detailreiche grafische Darstellung der Mandelbrotmenge zu sehen ist, und bei der auf die Selbstähnlichkeit und auf das Wiederkehren der Julia-Mengen eingegangen wird, stammt bereits aus dem Jahre 1981/82 [10].

Erste Forschungen bezüglich dem Verhalten von Zahlen in der komplexen Ebene unter Iteration von polynomiellen Funktionen wurden schon viel früher, im Jahre 1906 von Pierre Fatou veröffentlicht [11]. Zu dieser Zeit war es noch nicht möglich, das Erforschte mit elektronischen Hilfsmitteln zu berechnen und in irgendeiner Form grafisch auszugeben. Was Pierre Fatou erforschte, steht in unmittelbarem Zusammenhang mit den heutzutage so genannten Julia-Mengen [12].

Die erste visuelle Darstellung der Mandelbrotmenge wurde 1978/1979 von Robert W. Brooks und Peter Matelski veröffentlicht [13]. Diese Darstellung bedient sich noch keinen Farben, bloß dem American Standard Code for Information Interchange (ASCII) Zeichen "*", das einer konvergierenden komplexen Zahl entspricht. Divergierende komplexe Zahlen erhalten in dieser Darstellung kein Symbol und bilden somit die weiße Fläche in der Darstellung.

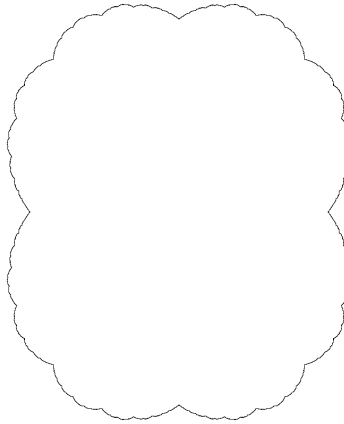


Abbildung 2.2: Visualisierung einer Julia-Menge, die bereits 1906 von Fatou untersucht wurde.



Abbildung 2.3: Die erste veröffentlichte Visualisierung der Mandelbrotmenge [13]

2.3.3 Erzeugen der Bilder

Um die farbkodierte Darstellung der Mandelbrotmenge umzusetzen, wird ein Ausschnitt der komplexen Ebene auf die anzeigende Bildfläche gelegt. Nun erhält jeder Pixel eine komplexe Zahl c , die an der Stelle des Pixels zur komplexen Ebene korrespondiert. Anschließend wird auf c gemäß der Mandelbrotschen Formel iteriert. Es werden die Iterationen gezählt, die es gebraucht hat, um Divergenz festzustellen, also bis der Betrag von c die Obergrenze $absValuaMax$ überschritten hat. Die Anzahl der benötigten Iterationen wird im Folgenden als *Iterationswert* bezeichnet. Jedem Iterationswert wird eine Farbe zugewiesen, die dann an dem Pixel angezeigt wird. Diese Farben werden in einer Farbpalette hinterlegt. Da diese unabhängig von den Berechnungen ist, besteht hier die freie Wahl über die Farben, die angezeigt werden. Daraus resultieren beliebig viele verschiedene farbliche Repräsentationen der Mandelbrotmen-

ge.

Wird unter Iteration die obere Iterationsgrenze *iterationMax* erreicht, so stellt dies das hinreichende Kriterium dar, um Konvergenz anzunehmen. Für jede Zahl c wird dann angenommen, dass sie in der Mandelbrotmenge liegt. Es wird konventionell die Farbe schwarz an dieser Stelle angezeigt.

Definition 2: *Iterationswert*

Der Iterationswert beschreibt die Anzahl an Iterationen, die benötigt wurden, bis der Betrag einer komplexen Zahl den Wert *absValueMax* überschritten hat.

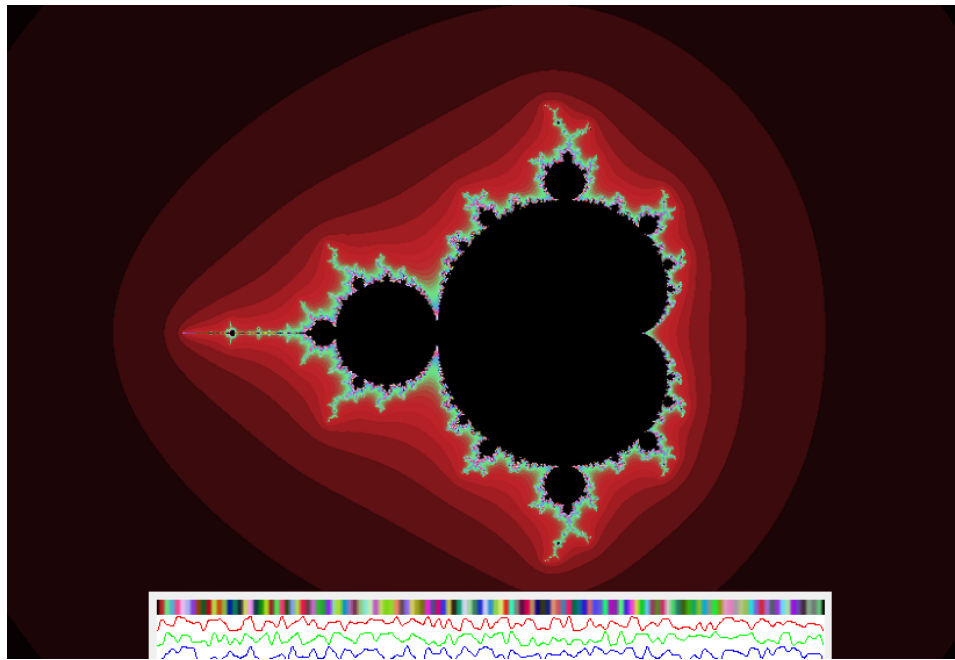


Abbildung 2.4:

Die grafische Darstellung der Mandelbrotmenge mit zugehöriger Farbpalette darunter. Im äußeren Bereich des Bildes sind die Grenzen zwischen den Iterationswerten gut zu erkennen.

Kapitel 3

Ansatz

3.1 Modularisierung

Das modulare Programmieren ist ein Programmierparadigma, das sich dadurch charakterisiert, dass ein größeres Programm aus kleineren Untereinheiten, so genannten Modulen, besteht. Es bietet sich die Möglichkeit, die einzelnen Module gezielt zu testen und sie im Bedarfsfall zu erweitern, ändern oder auszutauschen. Wichtig dabei ist, dass die Schnittstelle zu den anderen Modulen nicht geändert wird. Sollte eine Änderung der Schnittstelle aufgrund größeren Umstrukturierungen an einem Modul nötig sein, so muss ebenfalls die Schnittstelle auf der korrespondierenden Seite so angepasst werden, sodass beide Schnittstellen wieder übereinstimmen. Sollte im Vorhinein Funktionalität und Schnittstelle der einzelnen Module bekannt sein, so können die Module parallel entwickelt werden.

Durch mehrfache Instantiierung eines Moduls besteht die Möglichkeit, die gleiche Rechnung parallel auszuführen. Dazu muss dieses Modul nur ein einziges Mal geschrieben werden und es bedarf eines hierarchisch übergeordneten Modul, das genügend Schnittstellen bereitstellt und alle Module koordinieren kann. Eine Obergrenze für die Anzahl der instantiierten Module besteht dabei theoretisch nicht.

Somit bietet die modulare Programmierung zwei wesentliche Vorteile: einfache Parallelisierung, in Anwendung und Entwicklung, sowie flexible Designs aufgrund der einfachen Austauschbarkeit der Module.

Wie in diesem Kontext die Modularisierung umgesetzt wurde, siehe Kapitel „Implementierung“ (Kapitel 4).

3.1.1 Modularisierung in VHDL

In VHDL werden Hardwarebausteine (also die Module) in strikter Trennung zwischen Schnittstelle („Entity“) und Verhalten („Architecture“) beschrieben. In der „Entity“-Umgebung werden die Ein- und Ausgänge des Bausteins definiert. In der Architektur besteht die Möglichkeit, mit Hilfe der `component` Anweisung andere Bausteine zu instantiieren. Versorgt man die Ein- und Ausgänge des instantiierten Moduls mit Signalen aus der übergeordneten Architektur, lässt sich eine Hierarchie unter den Hardwarebausteinen erzeugen. Daraus resultiert implizit eine modulare Programmierung.

3.2 Automaten

Um die Funktionalität der Module strukturiert und übersichtlich zu implementieren, werden endliche Automaten genutzt. In VHDL eignet sich eine Implementierung eines Moore-Automaten. Ein Moore-Automat ist ein endlicher, deterministischer Automat, dessen Ausgabe nur von seinem aktuellen Zustand abhängt. Er besteht aus

- einem Eingabealphabet Σ , eine Menge von Zeichen, die gelesen werden können. Da Eingabe auch aus Zeichenfolgen bestehen dürfen, gilt für alle Eingabeworte $w : w \in \Sigma^*$
- einer nicht-leeren Menge von Zuständen Q
- einer Transitionsfunktion $\delta : Q \times \Sigma \rightarrow Q$, die einem aktuellen Zustand $q \in Q$ gepaart mit einer Eingabe $w \in \Sigma$ einen Folgezustand $q^+ \in Q$ zuweist.
- einem Startzustand $q_0 \in Q$, der zu Beginn vom Automaten angenommen wird
- einer Menge $F \subseteq Q$ von akzeptierenden Zuständen. Befindet sich nach Verarbeiten der Eingabe der Automat in einem Zustand $q \in F$, so wird die Eingabe als akzeptiert angesehen. Terminiert der Automat in einem Zustand $q \notin F$, so ist die Eingabe als verworfen anzusehen. Im Rahmen der Implementierung spielt das Akzeptieren von Eingaben keine Rolle, daher wird F stets die leere Menge \emptyset sein.

Um einen Moore-Automaten eindeutig zu beschreiben, bedarf es der Definition dieser 5 Eigenschaften. Konventionell wird demnach ein Moore-Automat M als Quintupel $M = (Q, \Sigma, \delta, q_0, F)$ angegeben.

3.2.1 Moore-Automaten in VHDL-Syntax

In VHDL lassen sich Moore-Automaten mit Hilfe der Aufzählungstypen (**type**-Anweisung), der Prozessumgebung (**process**-Anweisung) und des **case-when**-Statements realisieren. Die Aufzählungstypen erlauben es, Zustände mit einem lesbaren Bezeichner zu versehen. Auf der Hardware allerdings werden Zustände weiterhin binär kodiert. Die Prozessumgebung ermöglicht es, mit High-Level-Programmieranweisungen wie **if-then-else**, oder **case-when** Statements übersichtlich in allen möglichen Zuständen Eingaben zu verarbeiten und daraufhin einen Folgezustand zu ermitteln. Hier wird dem Programmierer das mühsame Dekodieren von Zuständen abgenommen.

Moore-Automaten lassen sich mit verschieden vielen Prozessen implementieren. In dieser Implementierung wurde entschieden, 3 Prozesse zu nutzen:

- Der erste Prozess dient dazu, synchron zum Taktsignal in einen ermittelten Folgezustand zu wechseln. Auch das Verarbeiten eines Reset-Signals wird hier realisiert.
- Der zweite Prozess bestimmt die Ausgabe für den aktuellen Zustand.
- Der dritte Prozess ist der einzige asynchrone Prozess. Er bestimmt den Folgezustand und kann auf Eingaben reagieren.

Es ist auch möglich einen Moore-Automaten mit einem einzigen asynchronen Prozess zu realisieren. Dabei ergeben sich andere Syntheseergebnisse und eventuelle Vor- und Nachteile in der Optimierung der Schaltung und dem daraus resultierendem Timing.

Listing 3.1: Beispielhafte Deklaration der Zustandsmenge $Q = \{start, c_calc7\}$ und des Startzustandes $q_0 = start$ in VHDL

```

1 type example_states is (start , c_calc7);
2 signal e_state : example_states := start;
3 signal e_state_next : example_states;
```

Listing 3.2: Beispielhafte Implementierung des ersten Prozesses

```

1  next_state: process (clk) is begin
2      if rising_edge(clk) then
3          if (reset = '1') then
4              math_state <= idle;
5          else
6              math_state <= math_state_next;
7          end if;
8      end if;
9  end process;

```

3.3 IP-Cores

Intellectual Property-Core (IP-Core)s sind bereits fertig geschriebene Code-Blöcke, die Dritte zur lizenzierten Nutzung bereitstellen. In dieser Implementierung kommen folgende IP-Cores zum Einsatz:

- **Floating-Point**
Zum Rechnen mit Gleitkommazahlen. Arithmetische Operationen wie die Addition, Subtraktion und Multiplikation werden von Floating-Point übernommen. Floating Point nimmt zwei Operanden als Bitvektor entgegen und gibt einen Bitvektor als Ergebnis zurück. Die genaue Kommunikation erfolgt über einzelne Pins, die über die Gültigkeit der Daten auf den Vektoren Auskunft gibt. Die korrekte Ansteuerung von Floating-Point muss implementiert werden.
- **Clocking Wizard**
Zum Erzeugen sauberer Clock-Signale. Clocking-Wizard nimmt ein Clock-Signal entgegen und kann daraus verschiedene Clock-Signale, auch anderer Frequenz erzeugen. Auch hier muss die korrekte Ansteuerung implementiert werden.

3.4 Systemeigenschaften

Um die Aufgabenstellung zu erfüllen, müssen folgende Systemeigenschaften implementiert werden:

1. Um die Mensch-Computer-Interaktion sicher zu stellen, müssen der Joystick und das LED-Display korrekt angesprochen werden
2. die komplexe Ebene muss auf dem LED Display ausgegeben werden
3. Es müssen sinnvolle Werte für die Variablen *absValueMax* und *iterationMax* gefunden werden
4. Es müssen beliebig viele komplexe ALUs (im Folgenden z.T. auch nur „ALU“) mit einer komplexen Zahl des sichtbaren Bildbereichs versorgt werden
5. Es muss eine parallelisierbare ALU entwickelt werden, die die Mandelbrotsche Formel auf einer komplexen Zahl iterieren kann
6. Die errechneten Iterationswerte für jeden Pixel müssen einer Farbe zugeordnet werden, die dann auf dem Display an der korrekten Stelle angezeigt werden

Ansätze, um die kleineren Ziele zu erreichen:

- (1.) Die Mensch Computer-Interaktion wird von den Modulen Joystick-Controller und LED-Controller übernommen. Genauer wird in Abschnitt 4.4 (Joystick-Controller), und 4.5 (LED-Controller) erläutert.
- (2.) Um die komplexe Ebene auf dem LED Display auszugeben, wird lediglich der Wert der komplexen Zahl in der unteren linken Ecke des Displays gespeichert. Von dort aus wird auf den Realwert dieser Zahl ein Zoomfaktor addiert. Ist der rechte Rand des Displays erreicht (nach 96 Pixeln, 95 Additionen), wird der ursprüngliche Realwert wiederhergestellt und der Zoomfaktor wird auf den Imaginärteil addiert. Nach diesem Schema wird weiter verfahren, bis der obere rechte Pixel erreicht wird. So wird lediglich unter der Verwendung einer Addiereinheit das ganze Display nach und nach abgearbeitet.
- (3.) Der Wert für *absValueMax* muss aus $\mathbb{R}_{>0}$ stammen. Um unnötige komplizierte Rechnungen zu vermeiden, wurde entschieden, den Wertebereich, aus dem *absValueMax* stammen darf, auf $\mathbb{N}_{>0}$ zu begrenzen. Da die Mandelbrotmenge etwa um den Ursprung der komplexen Ebene zentriert ist, und der betragsmäßig davon am weitesten entfernte Punkt bei $2 + 0 \cdot i$ liegt, eignet sich 2 als Wert für *absValueMax*. In dieser Implementierung wurde entschieden, eben 2 als Wert zu benutzen. Andere Implementierungen, die 100 (oder ähnlich große Zahlen) als Wert für *absValueMax* gewählt haben, kommen zur scheinbar gleichen visuellen Darstellung, allerdings sind die Iterationswerte höher, da das Abbruchkriterium für die Iteration später erreicht wird. Dies führt zu längeren Berechnungszeiten, ohne sichtbar anderem Ergebnis. Der Wert für *iterationMax* stammt aus $\mathbb{N}_{>0}$. Je höher dieser Wert gewählt wird, desto länger dauern die Berechnungen für die Bildpunkte, vor allem jene, an denen der Wert der komplexen Zahl konvergiert. Dafür können bei hoher Zoomstufe schärfere Bilder der Mandelbrotmenge erzeugt werden. Als Kompromiss zwischen Rechenzeit und Qualität wurde *iterationMax* auf 1.000 gesetzt.
- (4.) Die korrekte Ansteuerung der ALUs übernimmt das Hardwaremodul "Math-Modul,.". Genauer ist Abschnitt 4.2 (Math-Modul) zu entnehmen.
- (5.) Das Iterieren der Mandelbrotschen Formel auf einer gegebenen komplexen Zahl c übernimmt das Hardwaremodul "Komplexe-ALU,.". Näheres ist Abschnitt 4.3 (Komplexe-ALU) zu entnehmen.
- (6.) Die Bits für den Farbwert eines Pixels können direkt aus den Bits des Iterationswerts bestimmt werden.

Kapitel 4

Implementierung

Im Folgenden wird die genaue Implementierung dargestellt. Es folgt eine stichpunktartige Übersicht über die Funktionalität jedes Moduls, das zugehörige Schaltbild, die grafische Automaten Darstellung und eine Prosa-Erklärung für das Verhalten des Automaten. Zur grafischen Darstellung ist anzumerken, dass die impliziten Eigenkanten in jedem Zustand zur Lesbarkeit ausgespart wurden. Die Eigenkanten sorgen dafür, dass der aktuelle Zustand nicht gewechselt wird, sollte keines der dargestellten Übergangskriterien erfüllt sein. Ebenfalls ausgespart wurden alle Kanten, die von jedem Zustand in einen „Resetzustand“ führen.

Die gezeigten Automaten wurden in VHDL geschrieben, der zugehörige Code befindet sich im Anhang.

4.1 Top-Modul

Das Hardwaremodul Top-Modul stellt das Herzstück der Implementierung dar. Unter seine Aufgaben fällt das

- Verarbeiten der Ausgabe des Joystick-Controllers
- Anpassen des aktuell sichtbaren Bildausschnitts
- Versorgen des LED-Controllers mit gültigen Farb- und Koordinatenwerten
- Starten des Math-Moduls
- Auswerten der Ergebnisse des Math-Moduls



Abbildung 4.1: Das Schaltbild des Top-Moduls

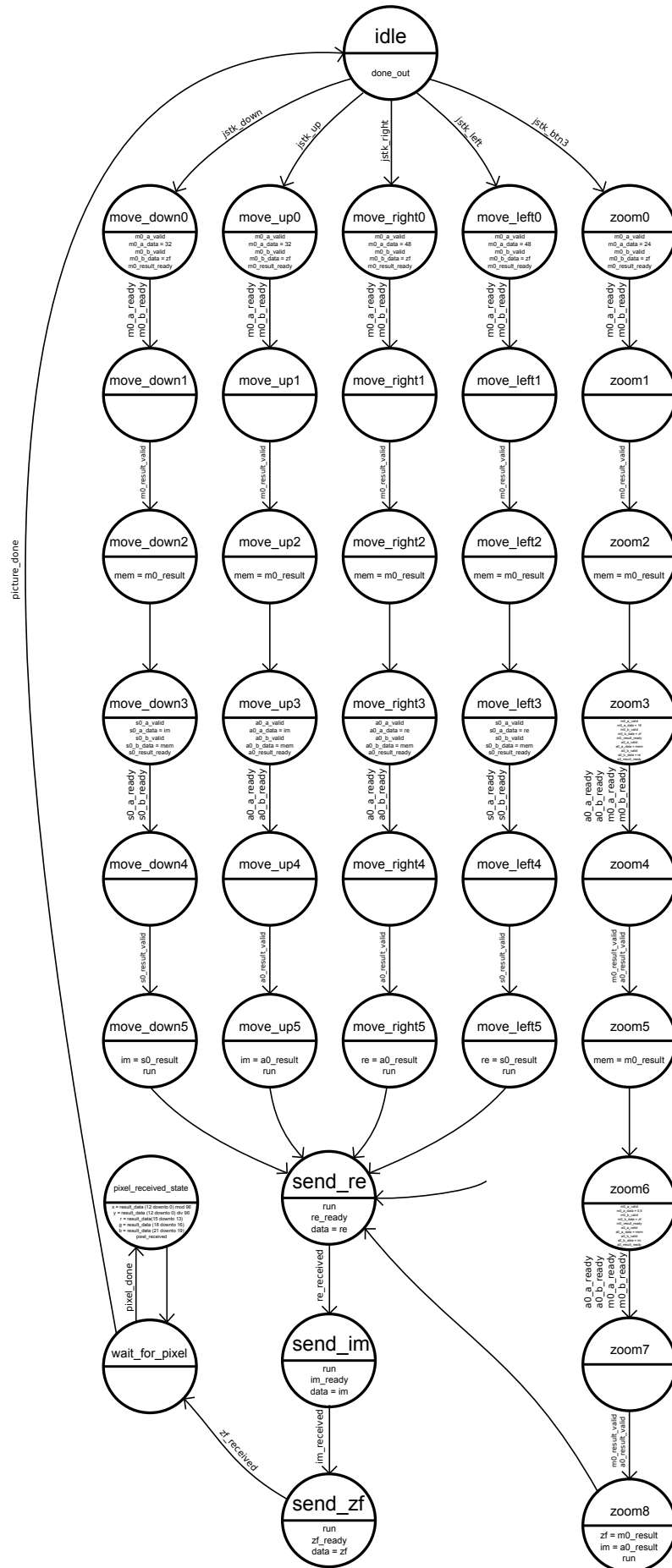


Abbildung 4.2: Der Automat des Top-Moduls

Verarbeiten der Ausgabe des Joystick-Controllers

Wenn sich der Automat des Top-Moduls im Zustand „idle“ befindet, kann auf Signale des Joystick-Controllers reagiert werden. Signalisiert der Joystick-Controller, dass der Joystick in eine zum PCB senkrechte Richtung bewegt wurde, so wird über die Zustände $move_dir_i$ mit $dir \in \{up, down, left, right\}$, $i \in \{0, \dots, 5\}$ der Wert der komplexen Zahl in der linken unteren Ecke des Displays angepasst.

Anpassen des aktuell sichtbaren Bildausschnitts

Da das Display eine Breite von 96 Pixeln hat, realisiert die Addition des 48-Fachen des Zoomfaktors (der jeweils einem Schritt von $\frac{1}{96}$ auf dem Display entspricht) auf den Realteil der komplexen Zahl in der unteren linken Ecke eine Verschiebung um 50% nach rechts. Eine Verschiebung gleicher Größe nach links wird realisiert, indem das 48-Fache des Zoomfaktors von dem Realteil der komplexen Zahl in der unteren linken Ecke subtrahiert wird.

Da das Display eine Höhe von 64 Pixeln hat, genügt es, das 32-Fache des Zoomfaktors auf den Imaginärteil der komplexen Zahl in der unteren linken Ecke zu addieren oder von ihm zu subtrahieren, um den Bildausschnitt um 50% nach oben oder unten zu bewegen.

Das Errechnen des 32-Fachen, beziehungsweise 48-Fachen, des Zoomfaktors übernimmt das Modul „floating_point_multiply“. Die darauffolgende Addition oder Subtraktion übernimmt das Modul „floating_point_adder“ beziehungsweise „floating_point_subtractor“. Aufgrund der Datenabhängigkeit ist eine Parallelisierung hier nicht möglich.

Der Zoomvorgang wird durch die Zustände $zoom_i$ mit $i \in \{0, \dots, 8\}$ realisiert. Ein einfacher Zoom kann durch das Senken des Zoomfaktors umgesetzt werden. Dies entspricht einer kleineren Schrittweite zwischen den komplexen Zahlen für die der Iterationswert berechnet wird. Um einen mittigen Zoom umzusetzen, muss auch der Wert der komplexen Zahl in der unteren linken Ecke des Displays angepasst werden. Diese Anpassung muss relativ zum aktuellen Zoomfaktor errechnet werden. Es wurde entschieden, den Zoomfaktor mit jedem Zoom zu halbieren. Der Realteil der komplexen Zahl in der unteren linken Ecke wird dabei um das 24-Fache des aktuellen Zoomfaktors erhöht, der Imaginärteil um das 16-Fache des aktuellen Zoomfaktors. Dies sorgt dafür, dass in die Mitte des Bildausschnitts gezoomt wird, also nach dem Zoom keine ungewollte Verschiebung entsteht.

Versorgen des LED-Controller mit gültigen Farb- und Koordinatenwerten

Das Top-Modul stellt insgesamt 5 Datenbusse für den LED-Controller zur Verfügung. Die Farbwerte des aktuell errechneten Pixels werden über die jeweils 3 Bit breiten Datenbusse „r“, „g“ und „b“ übermittelt. Der 7 Bit breite Datenbus „x“ übermittelt Informationen über die aktuelle X-Koordinate. Der 6 Bit breite Datenbus „y“ über die Y-Koordinate.

Starten des Math-Moduls

Startzustand des Top-Moduls ist „send_re“. Auf diesen folgen „send_im“ und „send_zf“. Diese 3 Zustände versorgen das Math-Modul mit aktuellen Daten über den Wert der komplexen Zahl in der unteren linken Ecke des Displays. Um Leitungen zu sparen, werden die 3 Werte „re“, „im“ und „zf“ seriell über den Datenbus „data“ übertragen. Die Steuerleitungen „re_ready“, „re_received“, „im_ready“, „im_received“, „zf_ready“ und „zf_received“ dienen dazu, die Daten kontrolliert zu übertragen. Anschließend wird dem Math-Modul befohlen, ein neues Bild zu berechnen. Dazu dient die Steuerleitung „run“, auf die das Math-Modul mit Starten der Berechnungen reagiert.

Sobald eine Eingabe über den Joystick gemacht wird, werden neue Werte für die komplexe Zahl in der linken unteren Ecke des Displays berechnet. Im Falle eines Zooms, wird auch der Zoomfaktor angepasst. Diese Vorgänge enden immer in dem Zustand „send_re“, sodass nach Eingabe über den Joystick stets ein neues Bild berechnet wird.

Auswerten der Ergebnisse des Math-Moduls

Das Math-Modul reicht die Ergebnisse der komplexen ALUs an das Top-Modul weiter. Dies erfolgt über einen 23 Bit breiten Bus, dessen untere 10 Bits den Iterationswert eines Pixels speichert. Die Nummer des Pixels wird in den oberen 13 Bit gespeichert. Die untersten 3 Bits des Iterationswertes werden genutzt, um den Rotanteil des Bildpunktes zu bestimmen. Die 3 nächsten Bits bestimmen den Grünanteil und die nächsten 3 Bits den Blauanteil.

4.2 Math-Modul

Das Hardwaremodul Math-Modul stellt die Verbindung zwischen dem Top-Modul und den beliebig vielen komplexen ALUs dar. Ihre konkreten Aufgaben sind das

- Erhalten von dem Wert der komplexen Zahl in der aktuell unteren linken Ecke des Displays
- Berechnen der 6144 komplexen Zahlen im sichtbaren Ausschnitt der komplexen Ebene
- Aufteilen der 6144 zu berechnenden komplexen Zahlen auf beliebig viele ALUs
- Versenden der Ergebnisse der ALUs mit einer Identifikationsnummer (ID)

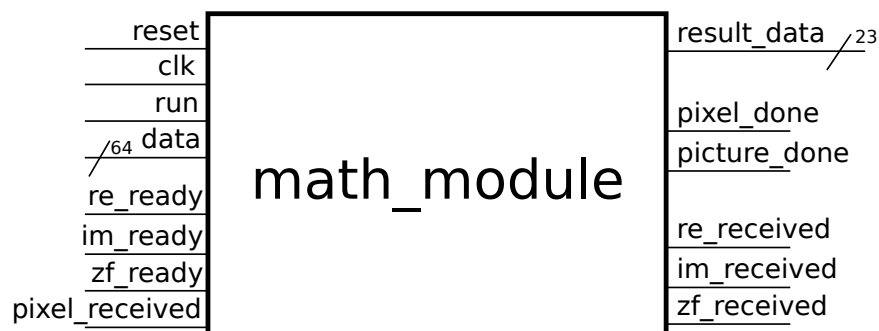


Abbildung 4.3: Das Schaltbild des Math-Moduls

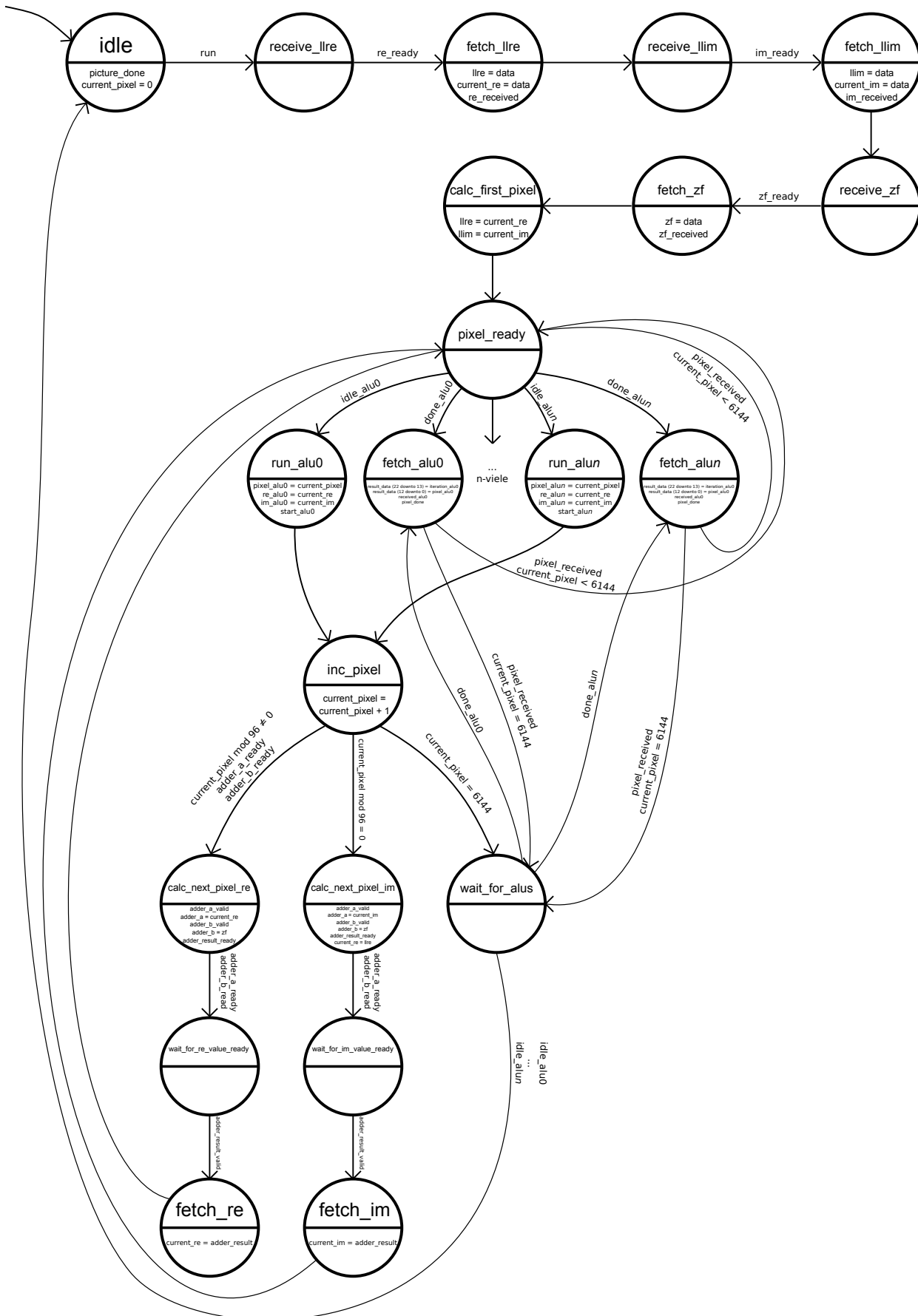


Abbildung 4.4: Der Automat des Math-Moduls

Erhalten von dem Wert der komplexen Zahl in der aktuell unteren linken Ecke des Displays

Die Zustände „receive_re“, „fetch_re“, „receive_im“, „fetch_im“, „receive_zf“ und „fetch_zf“ dienen zum kontrollierten Empfangen der Information über den aktuellen Wert der komplexen Zahl in der unteren linken Ecke des Displays, sowie die Schrittweite zwischen jedem Pixel, dem Zoomfaktor.

Berechnen der 6144 komplexen Zahlen im sichtbaren Ausschnitt der komplexen Ebene

Da das Display 6144 Pixel hat, müssen 6144 korrespondierende komplexe Zahlen errechnet werden. Die erste komplexe Zahl befindet sich in der unteren linken Ecke. Die vom Top-Modul übermittelten Daten sind für den ersten Pixel bereits gültig. Die nächsten 95 komplexen Zahlen werden durch Addition des Zoomfaktors auf den Realteil der jeweils aktuellen komplexen Zahl errechnet. Somit wird die unterste Zeile des Displays bearbeitet. Diese Berechnungen werden von den Zuständen „calc_next_pixel“, „calc_next_pixel_re“, „wait_for_re_value_ready“ und „fetch_re“ realisiert. Der nächste Pixel befindet sich in der vorletzten Zeile des Displays, in der Spalte ganz links. Um die komplexe Zahl für diesen Pixel zu errechnen, wird der ursprüngliche Realteil wiederhergestellt und der Zoomfaktor auf den Imaginärwert addiert. Somit wird ein „line-break“ umgesetzt. Dies übernehmen die Zustände „calc_next_pixel“, „calc_next_pixel_im“, „wait_for_im_value_ready“ und „fetch_im“.

Aufteilen der 6144 zu berechnenden komplexen Zahlen auf beliebig viele ALUs

Die Zustände „pixel_ready“ und „run_alu_i“ sorgen dafür, dass untätige ALUs mit Aufgaben versorgt werden, sobald eine komplexe Zahl zur Berechnung verfügbar ist. Ist eine ALU fertig, so wird dies über die Steuerleitung „done_alu_i“ signalisiert. Der Automat des Math-Moduls wechselt dann in einen Zustand „fetch_alu_i“, der das Ergebnis der ALU holt und sichert. Der Zustand „wait_for_alus“ wird am Ende der Berechnung eines Bildes eingenommen. Er stellt die Ausnahme dar, dass nach dem Sichern des Ergebnisses der ALU keine weiteren Aufträge erzeugt werden müssen. Von ihm aus gelangt das Math-Modul wieder in den Ausgangszustand „idle“, in dem es auf neue Werte für die komplexe Zahl in der unteren linken Ecke des Displays wartet, um anschließend mit der Neuberechnung eines Bildes beginnen zu können.

Vorsehen der Ergebnisse der ALUs mit einer ID

Jeder Pixel, der berechnet wird, wird mit einer ID versehen. Die ID ist in diesem Fall die Nummer des berechneten Pixels. Dies dient später dazu, einem errechneten Iterationswert einem Pixel zuzuordnen, sollten die ALUs in unterschiedlicher Reihenfolge ihre Berechnung beenden und ihr Ergebnis ausgeben.

4.3 Komplexe-ALU

Die komplexe ALU (im Folgenden z.T. auch nur “ALU,”) iteriert die Mandelbrotsche Formel auf einer gegebenen komplexen Zahl c . Sie gibt einen Iterationswert für c zurück. Die Iterationswerte liegen im Intervall $[0, 999]$ wobei die 0 einen konvergierenden Pixel beschreibt.

Unter ihre genauen Aufgaben fällt das

- Erhalten einer komplexen Zahl
- Iterieren, bis entweder Konvergenz oder Divergenz festgestellt wird
- Zurückgeben eines Iterationswertes

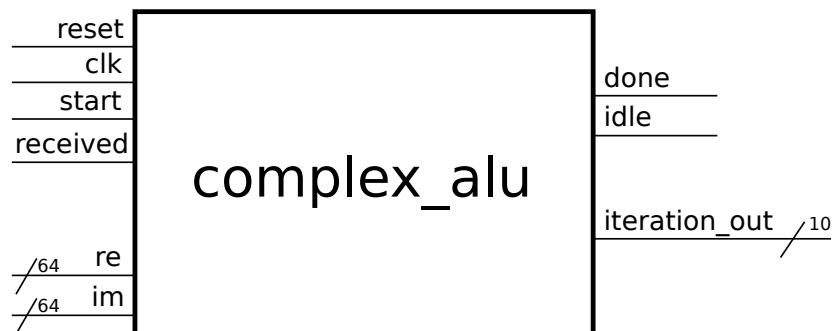


Abbildung 4.5: Das Schaltbild der komplexen ALU

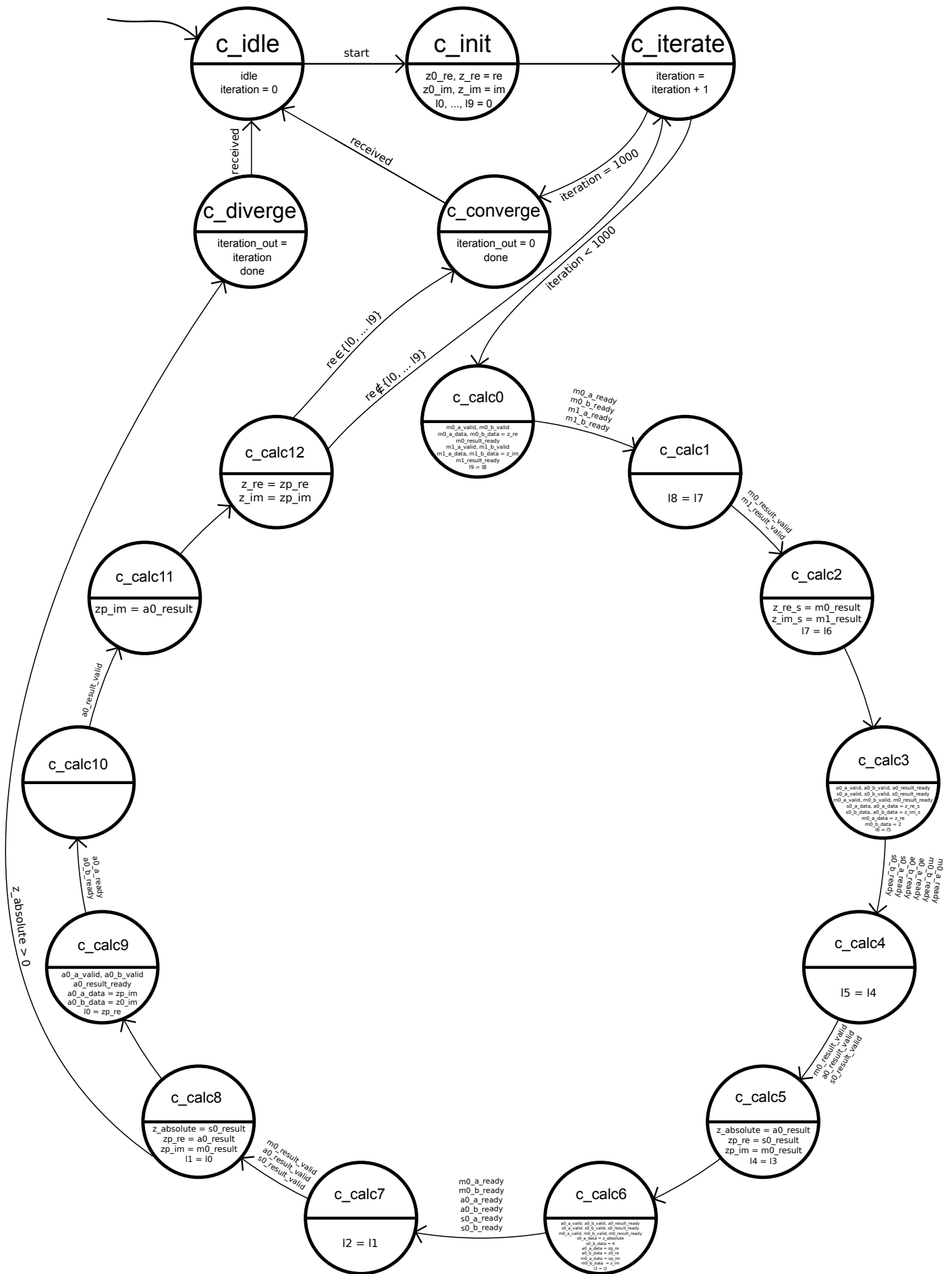


Abbildung 4.6: Der Automat der komplexen ALU

Erhalten einer komplexen Zahl

Im Gegensatz zum Math-Modul erhält die Komplexe-ALU die Werte der komplexen Zahl parallel. Es werden 2 64-Bit Leitungen genutzt, um die Daten zu übertragen. Daher reicht es aus, einen einzigen Zustand „init“ zur kontrollierten Übertragung zu nutzen. Die erhaltenen Werte werden in Registern „z0_re“ und „z0_im“ gespeichert.

Iterieren

Das Iterieren auf einer komplexen Zahl c wird von den Zuständen „c_iterate“, „c_calc $_i$ “ mit $i \in \{0, \dots, 12\}$ durchgeführt. Zu den nötigen Berechnungen gehören das Quadrieren von c , das anschließende Addieren der Konstanten c auf sein Quadrat, das Ermitteln des Betrages von c und das Prüfen, ob $|c|$ der Wert *absValueMax* überschritten wurde.

In den Zuständen „c_calc0“, „c_calc1“ und „c_calc2“ wird das Quadrat des Real- und Imaginärteils mit den beiden Multiplizierern parallel berechnet. Die Ergebnisse re^2 und im^2 werden in Signalen *z_re_s* und *z_im_s* gespeichert.

Die Zustände „c_calc3“, „c_calc4“ und „c_calc5“ berechnen parallel $re^2 + im^2$, $re^2 - im^2$ und $2 \cdot re$ mit Hilfe des Addierers, des Subtraktors und des Multiplizierers. Die Summe $re^2 + im^2$ ist das Quadrat des Betrages. Es gilt $re^2 + im^2 = |c|^2$. Dieses Ergebnis wird in einem Signal *z_absolute* gespeichert. Die Differenz $re^2 - im^2$ ist der Realteil von c^+ , allerdings muss noch der Realteil von c addiert werden. Die Differenz wird in einem Signal *zp_re* zwischengespeichert. Die Multiplikation $2 \cdot re$ ist das erste Produkt der Rechnung $2 \cdot re \cdot im$, was die erste Rechnung für den Imaginärteil von c^+ darstellt. Dieses Ergebnis wird in einem Signal *zp_im* zwischengespeichert.

Die Zustände „c_calc6“, „c_calc7“ und „c_calc8“ berechnen parallel $(2 \cdot re) \cdot im$, *z_absolute* - 4 und $re + zp_re$. Das Produkt $(2 \cdot re) \cdot im$ entspricht dem Imaginärteil von c^+ , allerdings muss auch hier noch der Imaginärteil von c addiert werden. Das Ergebnis wird in *zp_im* zwischengespeichert und überschreibt dort den alten Wert $2 \cdot re$. Um zu prüfen, ob *absValueMax* überschritten wurde, wird *z_absolute* - *absValueMax*² gerechnet und überprüft, ob das Ergebnis positiv ist. Um einen weiteren IP-Core für die Berechnung einer Wurzel einer Gleitkommazahl zu sparen, wurde entschieden, nicht $\sqrt{z_absolute}$ auszurechnen und mit *absValueMax* zu vergleichen, sondern $z_absolute = |c|^2$ mit *absValueMax*² zu vergleichen. So kommt die Subtraktion von *absValueMax*² = 2² = 4 zustande. Um einen Komparator-IP-Core zu sparen, wird das Vorzeichen der Differenz als ausschlaggebend betrachtet. Gilt $|c|^2 - \text{absValueMax}^2 > 0$, so liegt c nicht in der Mandelbrotmenge. Der Automat wechselt dann in einen Zustand „c_diverge“. Die Summe $re + zp_re$ stellt den fertig berechneten Realteil der komplexen Zahl c^+ dar.

Die Zustände „c_calc9“, „c_calc10“ und „c_calc11“ berechnen den Imaginärteil von c^+ . Der fertig berechnete Imaginärteil wird im Register *zp_im* abgelegt.

Zustand „c_calc12“ implementiert die Konvergenzerkennung. Die genaue Realisierung ist im nächsten Punkt erläutert. Folgezustand ist „c_converge“. Lag kein hinreichendes Kriterium für Konvergenz vor, so wird der Iterationswert erhöht, die inaktuellen Werte für *z_re* und *z_im* werden mit z^+_re und z^+_im überschrieben und Folgezustand ist „c_iterate“. In „c_iterate“ wird der aktuelle Iterationswert mit *iterationMax* verglichen. Sollte die Obergrenze *iterationMax* erreicht worden sein, so wechselt der Automat in Zustand „c_converge“. Andernfalls wird der komplette Zyklus erneut durchlaufen.

Erkennen von Konvergenz

Das Erkennen von Konvergenz gestaltet sich komplizierter als das Erkennen von Divergenz. Hierzu reicht nicht eine einfache Subtraktion, es müssen 3 verschiedene Fälle abgedeckt werden:

- Die komplexe Zahl hat unter Iteration ihren Konvergenzwert erreicht
- Die komplexe Zahl gerät unter Iteration in einen Zyklus
- Der Iterationswert *iterationMax* wurde überschritten.

Tritt der erste Fall ein, so ist $c^+ = c$. Im Zustand „c_calc_12“ wird überprüft, ob die Werte für *re* und *im* mit den Ergebnissen der Iteration *zp_re* und *zp_im* übereinstimmen. Ist dies der Fall, so ist Konvergenz garantiert. Beispiele, wie der Ursprung der komplexen Ebene ($0+0 \cdot i$) werden somit in nur einer Iteration als konvergierend gezeigt. In diesem Fall werden also 999 Iterationen gespart.

Den zweiten Fall zu prüfen stellt sich schwieriger dar. Um garantiert zu erkennen, ob eine komplexe Zahl *c* unter Iteration der Mandelbrotschen Formel in einen Zyklus gerät, müsste man jede komplexe Zahl c_{i-1} bis zur aktuellen Iteration *i* abspeichern und überprüfen, ob in den *i* – 1 Iterationen zuvor dieser Wert für *c* bereits errechnet wurde. In Programmiersprachen wie Java wäre dies mit Hilfe der Datenstruktur „Sets“ [14] schnell implementiert. Eine solche Umsetzung ist in VHDL nicht ohne Weiteres möglich. Daher werden bloß die Realteile der letzten 10 errechneten komplexen Zahlen gespeichert, um Zyklen bis zur Größe von 10 zu erkennen. Dazu dienen die Signale l_i mit $i \in \{0, \dots, 9\}$. Im Zustand „c_calc_0“ wird der Wert von 19 mit dem Wert von 18 überschrieben. In den Folgezuständen „c_calc_1“ bis „c_calc_9“ passiert diese Verschiebung mit den restlichen Signalen. Somit steht in jeder Iteration im Zustand „c_calc_12“ der Realteil der letzten 10 komplexen Zahlen zum Vergleichen zu Verfügung. Sollte der neu errechnete Realteil mit einem der Werte von l_i mit $i \in \{0, \dots, 9\}$ übereinstimmen, so ist der Folgezustand „c_converge“.

Zurückgeben eines Iterationswertes

Der Iterationswert wird über das Signal „iteration_out“ an das Math-Modul ausgegeben. Dieser Bus hat eine Breite von 10 Bit. Diese Breite ergibt sich nach folgender Gleichung durch Einsetzen des Wertes 1.000 für *iterationMax*:

$$\text{Datenbusbreite} = \lceil \log_2(\text{iterationMax}) \rceil = \lceil \log_2(1000) \rceil = \lceil 9,965784\dots \rceil = 10$$

Im Zustand „c_converge“ wird ein 10-Bit Nullvektor auf den Bus gelegt. Dies zeigt an, dass die komplexe Zahl in der Mandelbrotmenge liegt und führt dazu, dass der Pixel an dieser Stelle schwarz bleibt.

Im Zustand „c_diverge“ wird der aktuelle Iterationswert in binär auf den Ausgabebus „iteration_out“ gelegt. Die weitere Verarbeitung dieser Daten übernimmt dann das Math-Modul.

4.4 Joystick-Controller

Das Hardwaremodul Joystick-Controller implementiert den Treiber für den Pmod Joystick. Das Drücken des Joysticks bis zum Anschlag in eine zum PCB senkrechte Richtung signalisiert dem Top-Modul, eine neue komplexe Zahl dem unteren linken Bildpunkt zuzuweisen. Somit kann durch die Mandelbrotmenge navigiert werden. Auch das Betätigen des Zoomes wird durch dieses Modul dem Top-Modul signalisiert. Der Joystick-Controller wurde weitestgehend der Xilinx Demo File [15] entnommen und leicht angepasst, sodass der Joystick für diese Anwendung geeignet genutzt werden kann.

4.5 LED-Controller

Das Hardwaremodul LED-Controller implementiert den Treiber für das Pmod LED Display. Es nimmt eine X-Koordinate aus dem Intervall $[0,95]$ vom Datenbus „x“, sowie eine Y-Koordinate aus dem Intervall $[0,63]$ vom Datenbus „y“ des Top-Moduls entgegen. Die Farbwerte für die Pixel liegen auf den jeweils 3-Bit breiten Datenbussen „r“, „g“ und „b“ des Top-Moduls. Eine lauffähige Implementierung dieses Moduls war im Zeitrahmen der Bachelorarbeit leider nicht erfolgreich.

4.6 Hierarchie

Den Abbildungen ist die Hierarchie der Module zu entnehmen. Hierbei sind die grau gezeichneten Leitungen für die abgebildete Hierarchie irrelevant. Sie enden in der Logik der Automaten oder dienen zur Kommunikation und Datenübertragung zwischen den Modulen.

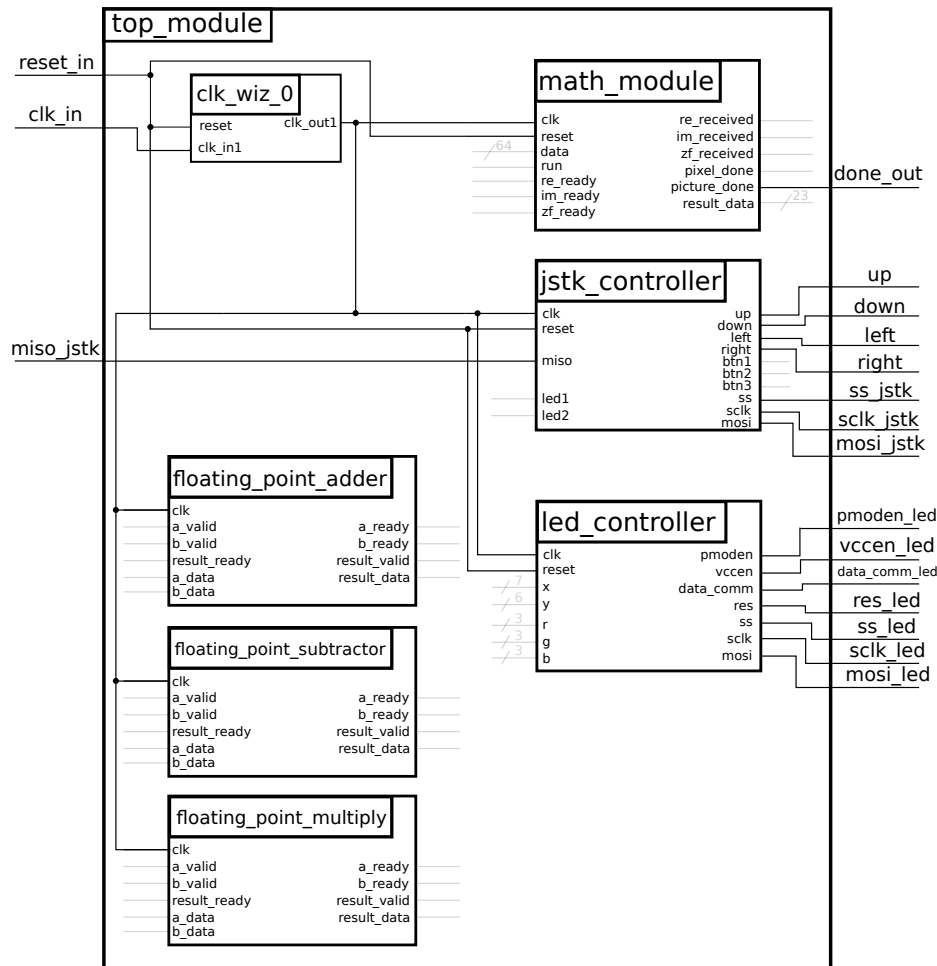


Abbildung 4.7: Die Hierarchie der Top-Level Ebene

Das Top-Modul verbindet alle anderen Module miteinander und stellt damit die höchste Ebene dar. Ihre Ein- und Ausgänge werden direkt auf die Pins des Boards gelegt. Das Clock-Signal kommt vom Quarz des Boards und endet in einem IP-Core „Clocking-Wizard“. Er erzeugt ein sauberes und kontrollierbares Clock-Signal, das alle weiteren Module versorgt. Die Frequenz von 100 Megahertz (MHz) bleibt erhalten. Der Joystick Controller erhält über die SPI Schnittstelle „miso_jstk“ Informationen vom Joystick, der über den Pmod Anschluss „JA“ angeschlossen ist.

Der LED-Controller versorgt das LED Display, das über den Pmod-Port „JC“ angeschlossen wird.

Die Module „floating_point_adder“, „floating_point_subtractor“ und „floating_point_multiply“ dienen dazu, den Wert der komplexen Zahl in der unteren linken Ecke des Displays, sowie den Zoomfaktor anzupassen. Je nach dem, ob nach rechts, links, oben oder unten navigiert wurde, oder ob gezoomt werden soll.

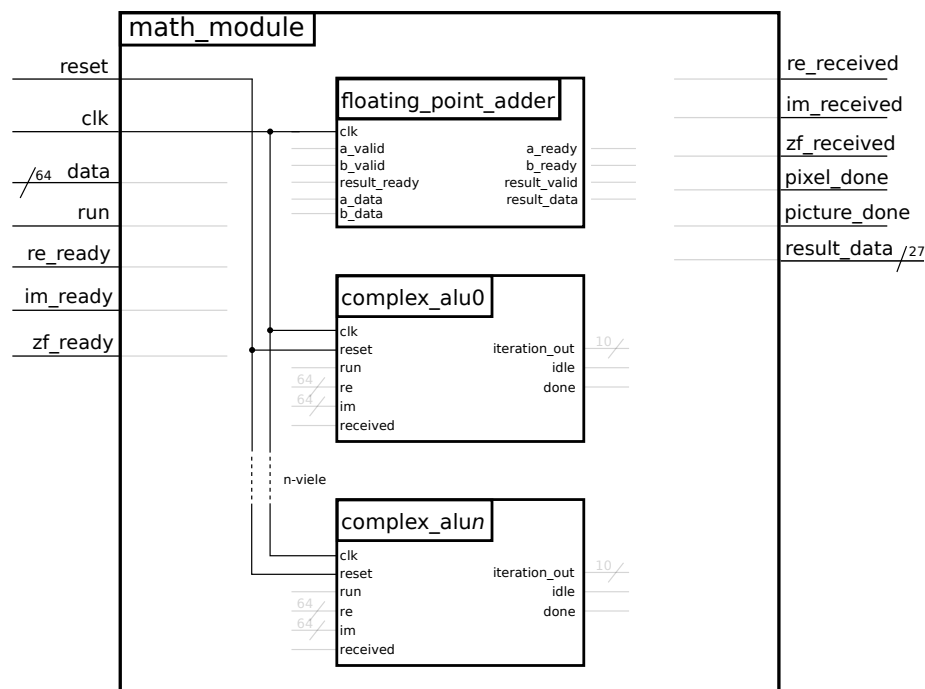


Abbildung 4.8: Die Hierarchie des Math-Moduls

Das Math-Modul besitzt einen „floating_point_adder“, um den Zoomfaktor auf den Wert komplexer Zahlen zu addieren. Darüber hinaus steuert es beliebig viele Komplexe-ALUs. Für jede Komplexe-ALU werden Signale benötigt, die sie mit Informationen über die komplexe Zahl (aufgeteilt auf die beiden Datenbusse „re“ und „im“), auf der zu iterieren ist, versorgt. Die Leitungen „received“, „idle“ und „done“ dienen zur Kommunikation zwischen den Komplexe-ALUs und dem Math-Modul.

Kapitel 5

Ergebnisse

Im Folgenden werden die Ergebnisse der Implementierung präsentiert. Es wird einzeln auf die in 1.3 definierten Ziele eingegangen.

5.1 Der Hardwareentwurf

Es wurde erfolgreich ein VHDL-Entwurf für FPGAs entwickelt, der ohne Errors synthetisiert, implementiert und in das FPGA geladen werden kann. Die Synthese entfernt bei der Optimierung einige Register, was zu ungewolltem Verhalten führen kann. Das Fortbestehen der Korrektheit des Entwurfs festzustellen, war zeitlich nicht möglich. Die Synthese, die Optimierung und die Implementierung, gefolgt vom Schreiben des Bitstreams eines Designs mit nur einer einzigen ALU, dauern in der genutzten Version von Vivado insgesamt rund 13 Minuten. Für jede kleine Änderung am Code fällt dieser gesamte Prozess erneut an und benötigt für jede hinzukommende ALU noch mehr Zeit.

Gemäß der Simulation kann der Hardwareentwurf auf einem Ausschnitt der komplexen Ebene die Mandelbrotsche Formel iterieren und dazu ein farbiges Bild der Mandelbrotmenge ausrechnen. Das Anpassen des Bildausschnitts wurde erfolgreich implementiert und verläuft wie erwartet. Das stellt das Erkunden der Mandelbrotmenge sicher.

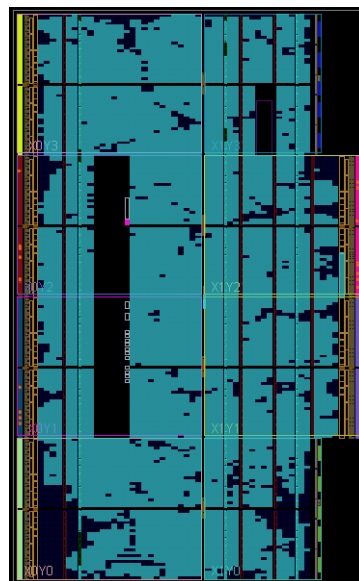


Abbildung 5.1: Der Hardwareentwurf mit 8 ALUs auf dem Artix 7. Hellblau entspricht belegten Logikzellen, dunkelblau ist frei. 95% der DSPs werden genutzt.

5.2 Joystick und Display

Die Kommunikation mit dem Joystick wird erfolgreich vom Joystick-Controller implementiert. Die Eingabe kann vom Top-Modul verarbeitet werden und führt in der Simulation zu einem Anpassen des aktuell sichtbaren Ausschnitts der komplexen Ebene. Auf der Hardware werden die vier LEDs rechts genutzt, um korrekt verarbeitete Eingaben aufzuzeigen (Siehe Abbildung 5.2 - Drücken nach rechts lässt die LED ganz rechts aufleuchten).

Die Implementierung des LED-Controllers war leider nicht erfolgreich. Das Einlesen in den über SPI gesteuerten Solomon Systech SSD1331 Display Controller [16], sowie das anschließend erfolgreiche Ansprechen des Displays war im Zeitrahmen der Bachelorarbeit nicht möglich.

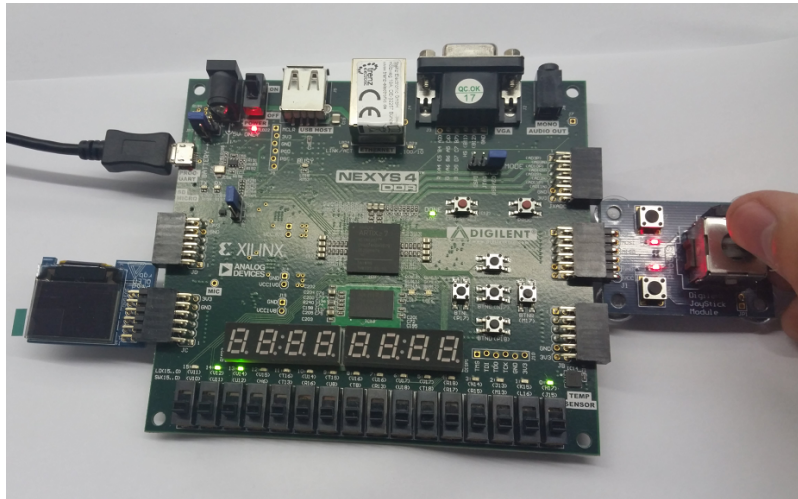


Abbildung 5.2: Der Joystick ließ sich erfolgreich als Eingabegerät einbinden. Das Display bleibt leider dunkel.

5.3 Beurteilung der Parallelisierungseffizienz

Auf dem FPGA ist es aufgrund der praktisch beschränkten Anzahl der konfigurierbaren Hardwarebausteine nicht möglich, mehr als 8 Komplexe-ALUs unterzubringen (Siehe Anhang, 7.1). Das Fortbestehen der Korrektheit war im Zeitrahmen der Arbeit nicht möglich, weshalb eine Zeitmessung am Board nicht sinnvoll ist. Damit dennoch ein möglichst aussagekräftiges Urteil über Effizienz der Parallelisierung zustande kommt, werden die Simulationsergebnisse betrachtet. Dort besteht theoretisch keine Obergrenze für die Anzahl der parallel arbeitenden ALUs, was gegenüber der Implementierung auf der Hardware ein großer Vorteil ist. Ebenfalls besteht die Möglichkeit, bis auf Nanosekunden genau die Zeit des theoretischen Ablaufes zu messen. Die Simulationsergebnisse stellen daher eine konstruktive Alternative da, um die Effizienz der Parallelisierung abschätzen zu können.

Spalte 1 der Tabelle 5.1 gibt Auskunft über die Anzahl der implementierten ALUs in der jeweiligen Simulation.

Spalte 2 zeigt die in der Simulation vergangene Zeit, die es gedauert hat, bis alle 6144 Pixel des Bildes fertig berechnet wurden.

Der Speed-Up Faktor steht in Spalte 3. Die Zeit, die in der Simulation mit nur einer ALU vergangen ist, wird als Referenzwert genommen. Damit steht jedes parallelisierte Ergebnis dem rein sequenziell berechneten Ergebnis gegenüber.

In Spalte 4 steht die Größe der .wdb Datei. Dies ist die Datei, die das Simulationstool anlegt und dort Informationen über die laufende Simulation ablegt. Je länger eine Simulation läuft,

desto größer wieder diese Datei.

Die letzte Spalte gibt Information über die in der Realität vergangene Zeit, die die Simulation lief.

ALUs	Simulationszeit	Speed-Up Faktor	Größe der .wdb Datei	Rechenzeit
1	2,980427328 s	100,0000%	925.129.378.000 Bytes	87h 14m 59s
2	1,499020169 s	198,8825%	500.163.076.000 Bytes	39h 33m 52s
4	0,749917795 s	397,9743%	265.453.892.000 Bytes	20h 19m 45s
8	0,3789402895 s	786,8651%	132.201.554.000 Bytes	12h 42m 21s
16	0,195372699 s	1525,5086%	72.444.012.000 Bytes	7h 32m 02s
32	0,992802226 s	3002,0353%	51.261.537.000 Bytes	4h 01m 29s
64	0,051410618 s	5797,2991%	44.927.012.000 Bytes	3h 04m 51s
128	0,027015148 s	11032,4301%	41.329.587.000 Bytes	2h 44m 52s

Tabelle 5.1: Ergebnisse der Simulation

Eine Exponenten-Regression über den Speed-Up Faktor ergibt folgende Funktion:

$$SpeedUpFaktor \approx 102.1063968 \cdot AnzahlALUs^{0.9710699028}$$

Aus dem Exponenten der obigen Funktion lässt sich herauslesen, wie effizient die Hinzunahme einer weiteren komplexen ALU ist. Ein Exponent von 1 bedeutet maximale Effizienz: jede weitere Komplexe-ALU, addiert ihre komplette Rechenkraft zur bestehenden Gesamtrechenkraft hinzu. Der hier ermittelte Exponent lässt sich also als Effizienzfaktor bezeichnen und zeigt eine überraschend gute Effizienz bei Hinzunahme einer weiteren ALU. Tabelle 5.2 zeigt, dass der Funktionswert der Regression bis 64 ALUs leicht unterschätzend ist. Das heißt, dass der Effizienzfaktor bis dorthin noch größer als der obig ermittelte Wert ist. Bei 128 ALUs ist der Regressionswert jedoch deutlich überschätzend, was vermuten lässt, dass ab dann die Effizienz jeder weiteren ALU sinkt. Es ist zu erwarten, dass spätestens ab 6144 ALUs der Effizienzfaktor sehr schnell gegen 0 geht, da jede weitere Komplexe-ALU keine Bildpunkte zum Berechnen zugeweiht bekommen kann, demnach im Ruhezustand bleibt und schließlich die Berechnung nicht weiter beschleunigt.

5.3.1 Bottleneck Math-Modul

Die Frage, ob die hier vorgestellte Implementierung „perfectly parallel“ ist, lässt sich im Hinblick auf das Math-Modul leicht beantworten. Der Lösungsweg eines Problems wird als „perfectly parallel“ bezeichnet, wenn der Effizienzfaktor 1 ist. Durch jede weitere parallel ausgeführte Rechnung muss sich also die hinzukommende Rechenleistung zur bereits bestehenden Gesamtrechenleistung hinzuaddieren. Dies ist meist dann der Fall, wenn jede Rechnung, sowie jedes Ergebnis dieser Rechnungen in keine Abhängigkeit zueinander stehen und untereinander keine Kommunikation benötigen. Wie Tabelle 5.2 und der errechnete Effizienzfaktor zeigen, ist dies in dieser Implementierung nicht der Fall. Dies liegt am Design des Math-Moduls, das die ALUs steuert.

Das Math-Modul ist so implementiert, dass das Holen und Sichern der Ergebnisse fertiger ALUs höher priorisiert ist, als das Starten untätiger ALUs. In den Simulationen ist gut zu sehen, dass höchstens 5 ALUs gestartet werden, wenn alle laufenden ALUs nach genau einer Iteration bereits Divergenz festgestellt haben. Dies passiert zum Beispiel in den untersten 3 Zeilen in Abbildung 5.4. Erst wenn ein Bereich der Mandelbrotmenge berechnet wird, in dem die komplexen Zahlen höhere Iterationswerte besitzen, werden mehrere ALUs gestartet. Eine

solche Engstelle, die die Effektivität potentieller Rechenleistung hemmt, wird „Bottleneck“ genannt. Der hier vorliegende „Bottleneck“ kommt zustande, da das Starten einer ALU, sowie das Holen und Sichern ihrer Ergebnisse nur sequenziell ausgeführt werden kann und eine Zeitspanne $t > 0s$ dauert.

Durch Änderungen am Code ist es möglich, das Starten einer untätigen ALU höher zu priorisieren. Ob dies das Problem des „Bottlenecks“ lindern oder fördern würde, oder ob es den „Bottleneck“ bloß verschieben würde, wäre zu prüfen.

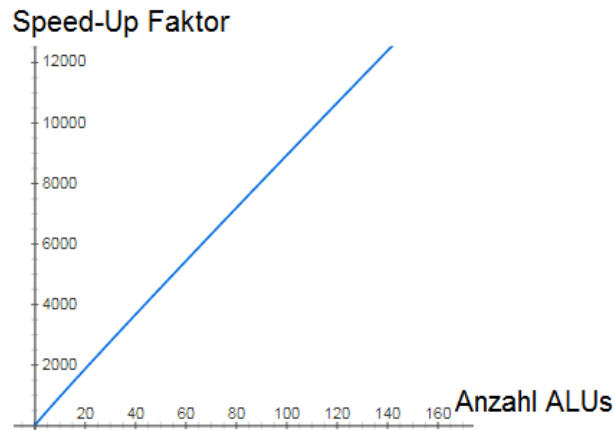


Abbildung 5.3: Graph der exponentiellen Regression

ALUs	Speed-Up Faktor	Regressionswert	Fehler	Fehler im Durchschnitt
1	100	102,1063968	2,106396819	2,106396819
2	198,8825	200,1585377	2,158537677	2,132467248
4	397,9743	392,3695425	-4,630457512	-0,121841053
8	786,8651	769,1595855	-16,84041451	-4,3014843815
16	1525,5086	1507,778775	-17,22122543	-6,8854325912
32	3002,0353	2955,689399	-46,31060091	-13,4562939776
64	5797,2991	5794,0196348	-2,980365783	-11,959732807
128	11032,4301	11357,98083	325,9808257	30,282837006375

Tabelle 5.2: Exponenten-Regression über den Speed-Up Faktor

Tabelle 5.2 stellt die tatsächlichen Werte des Speed-Up Faktors mit denen der Regression gegenüber. In Spalte 3 wird der Wert der Regressionsfunktion für die Anzahl der ALUs aus jeder Zeile aufgelistet. Die Werte in Spalte 4 werden errechnet, indem der tatsächliche Speed-Up Faktor vom zugehörigen Regressionswert abgezogen wird. Dadurch wird ersichtlich, ob die Regressionsfunktion den tatsächlichen Speed-Up Faktor über- oder unterschätzt. Ein negativer Wert bedeutet, dass der tatsächliche Speed-Up Faktor besser ist, als die Regressionsfunktion angibt. Ein positiver Wert bedeutet, dass der tatsächliche Wert schlechter als der durch die Regressionsfunktion angegebene Wert ist. Die letzte Spalte zeigt die durchschnittliche Abweichung von dem Wert der Regression zum tatsächlichen Speed-Up Faktor. Je näher die durchschnittliche Abweichung an der 0 ist, desto höher ist die Güte der Regressionsfunktion anzusehen. Hier ist gut zu erkennen, dass ab 128 komplexen ALUs der Wert der Regressionsfunktion stark überschätzt (um rund 3%). Dies lässt vermuten, dass ab dieser Anzahl von ALUs die Effizienz jeder weiteren hinzuzunehmenden ALU deutlich nachlässt.

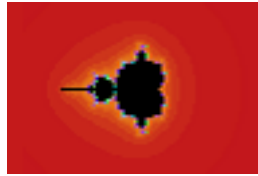


Abbildung 5.4: Das in der Simulation berechnete Bild (Originalgröße)

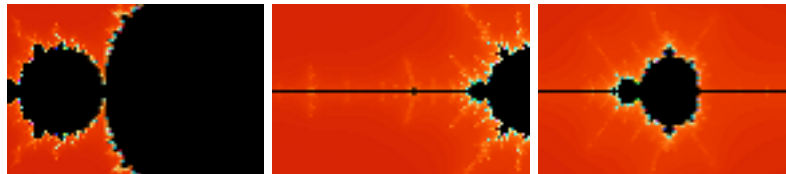


Abbildung 5.5: Beispielhafte Erkundung der Mandelbrotmenge. Das linke Bild wird erzeugt, wenn vom Startpunkt aus (siehe 5.4) 2 mal gezoomt wird. Das mittlere Bild ist zu sehen, wenn danach 2 mal nach links navigiert wird. Weiteres 3-maliges Zoomen und 2 maliges Navigieren nach links lässt das rechte Bild entstehen.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Was war erfolgreich?

Die Brücke zwischen den in Kapitel 1.1 (Motivation) vorgestellten Modulen des Bachelorstudiengangs Informatik der Goethe Universität Frankfurt am Main konnte erfolgreich geschaffen werden.

Die gelehrte Theorie der komplexen Zahlen konnte mit der praktischen Anwendung der FPGAs verbunden werden. Durch den Hardwareentwurf, der grundlegende arithmetische Berechnungen mit komplexen Zahlen, sowie das Iterieren der Mandelbrotschen Formel auf einer komplexen Zahl beherrscht, wird der gelehrte Inhalt der Veranstaltung „Mathe für die Informatik 1: Analysis und lineare Algebra“ durch praktische Anwendung greifbarer.

Die Parallelisierung zeigte eine überraschend gute Skalierbarkeit mit bis zu 128 parallel arbeitenden komplexen ALUs.

Der Joystick wurde erfolgreich in den Hardwareentwurf eingebunden, seine Funktionalität auf der Hardware konnte über die Ausgabe per LEDs gezeigt werden.

6.2 Wie kann aufbauend verfahren werden?

Um die Darstellung der Mandelbrotmenge visuell auszugeben, bedarf es noch der Implementierung des LED-Controllers. Dies würde zugleich zeigen, ob die Korrektheit der Implementierung auf der Hardware weiterhin besteht.

6.3 Mögliche Anwendungszwecke der Arbeit

Der vorgestellte Entwurf kann in der Veranstaltung „Mathe für die Informatik 1: Analysis und lineare Algebra“ genutzt werden, um dem gelehrten theoretischen Wissen über komplexe Zahlen eine praktische Anwendung zu verleihen. Der von Studenten anfangs womöglich schwer aufzufassende Inhalt erhält dadurch eine grafische, leicht begreifbare Anwendung über den visuellen Lernkanal.

Das Math-Modul kann jede Art von ALU betreiben, die eine passende Schnittstelle hat. Daher besteht die Möglichkeit, auch andere mathematische grafisch darstellbare Objekte parallel zu berechnen. Auch die Umsetzung einer parallel rechnenden Grafik-Engine ist umsetzbar.

Durch Anpassung einiger Werte im Math-Modul lässt sich auch die Bildfläche von größeren Displays berechnen.

Kapitel 7

Anhang

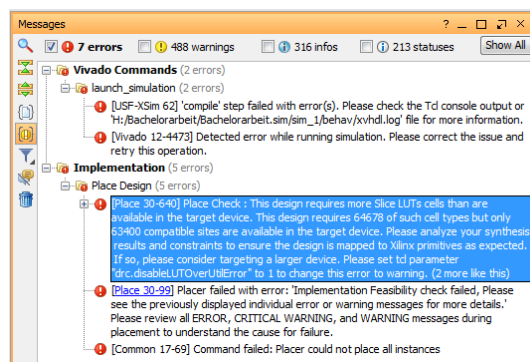


Abbildung 7.1: 16 ALUs lassen sich nicht auf dem Artix-7 FPGA unterbringen.

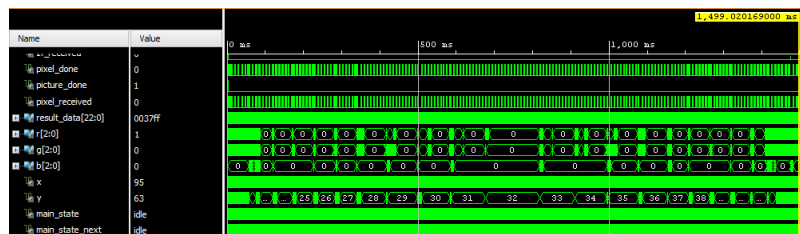


Abbildung 7.2: Entwurf mit 2 ALUs in der Simbox

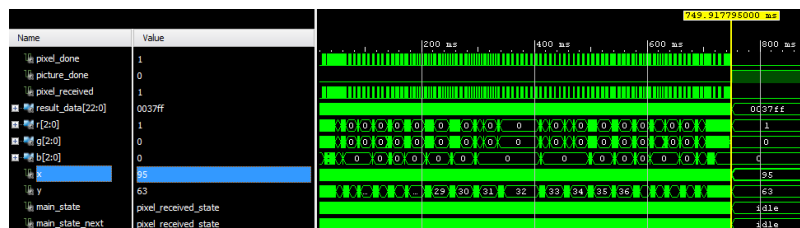


Abbildung 7.3: Entwurf mit 4 ALUs in der Simbox



Abbildung 7.4: Entwurf mit 8 ALUs in der Simbox

Die anderen Simulationen sehen den gezeigten Screenshots sehr ähnlich und werden daher nicht gezeigt.

Der Quellcode steht hier zum Download bereit:

m-data.dyndns.org/rados/studium.htm

Literaturverzeichnis

- [1] “Ordnung der Johann Wolfgang Goethe-Universität Frankfurt am Main für den Bachelorstudiengang Informatik,” Juli 2012. [Online]. Available: http://www.cs.uni-frankfurt.de/images/pdf/informatik/bachelor2/bachelorordnung_neu.pdf
- [2] *Nexys4 DDR™ FPGA Board Reference Manual*, Digilent, April 2016.
- [3] *PmodJSTK™ Reference Manual*, Digilent, Mai 2016.
- [4] *Nexys4 DDR™ FPGA Board Reference Manual*, Digilent, April 2016. [Online]. Available: <https://reference.digilentinc.com/reference/pmod/pmodoledrgb/reference-manual>
- [5] C. Bobda, *Introduction to Reconfigurable Computing*, 2007.
- [6] K. Franz. History of the FPGA. [Online]. Available: <https://blog.digilentinc.com/history-of-the-fpga/>
- [7] “Providing Battery-Free, FPGA-Based RAID Cache Solutions,” August 2010. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01141-raid-cache.pdf
- [8] Komplexe Zahlen Calculator. [Online]. Available: <http://de.numberempire.com/images/complex-numbers.png>
- [9] Mandelbrot set. [Online]. Available: https://en.wikipedia.org/wiki/Mandelbrot_set
- [10] A. Douady and J. H. Hubbard, *ETUDE ´ DYNAMIQUE DES POLYNOMES ^ COMPLEXES*.
- [11] P. Fatou, *Séries trigonométriques et séries de Taylor*, 1906.
- [12] Pierre fatou. [Online]. Available: https://en.wikipedia.org/wiki/Pierre_Fatou
- [13] R. Brooks and J. P. Matelski, “The dynamics of 2-generator subgroups of $psl(2, \mathbb{C})$,” in *Riemann surfaces and related topics: Proceedings of the 1978 Stony Brook conference*, 1981.
- [14] C. Ullenboom, *Java ist auch eine Insel*, 9th ed., Bonn, 2011. [Online]. Available: http://openbook.rheinwerk-verlag.de/javainssel9/javainssel_13_005.htm#mj45263b87fd44c62f2cde668164897a93
- [15] Pmod JSTK Example Code. [Online]. Available: https://reference.digilentinc.com/pmod/pmod/jstk/example_code
- [16] Pmod OLEDrgb Reference Manual. [Online]. Available: <https://reference.digilentinc.com/reference/pmod/pmodoledrgb/reference-manual>