Generierung von Logikschaltungen zum effizienten Lösen von Differentialgleichungen auf FPGAs

Silas Bartel

Bayreuth Reports on Parallel and Distributed Systems

No. 13, August 2020

University of Bayreuth
Department of Mathematics, Physics and Computer Science
Applied Computer Science 2 – Parallel and Distributed Systems
95440 Bayreuth
Germany

Phone: +49 921 55 7701
Fax: +49 921 55 7702
E-Mail: brpds@ai2.uni-bayreuth.de

Bachelorarbeit

Generierung von Logikschaltungen zum effizienten Lösen von Differentialgleichungen auf FPGAs

---

Generation of logic circuits for efficient solving of differential equations on FPGAs

Silas Bartel

25. August 2020

UNIVERSITÄT
BAYREUTH

Lehrstuhl für Angewandte Informatik II
Institut für Informatik

Bachelorarbeit

Generierung von Logikschaltungen zum effizienten Lösen von Differentialgleichungen auf FPGAs

Generation of logic circuits for efficient solving of differential equations on FPGAs

Silas Bartel
Mat-Nr.: 1399215

1. Prüfer Dr. Matthias Korch
Lehrstuhl für Angewandte Informatik II
University of Bayreuth

2. Prüfer Prof. Dr. Thomas Rauber
Lehrstuhl für Angewandte Informatik II
University of Bayreuth

Betreuer Dr. Matthias Korch

Bayreuth, der 25. August 2020

Silas Bartel
Generierung von Logikschaltungen zum effizienten Lösen von Differentialgleichungen auf FPGAs
Bachelorarbeit, 25. August 2020
Prüfer: Dr. Matthias Korch and Prof. Dr. Thomas Rauber
Betreuer: Dr. Matthias Korch

University of Bayreuth
Institut für Informatik
Lehrstuhl für Angewandte Informatik II
Universitätsstraße 30
95447 Bayreuth

Zusammenfassung

Zusammenhänge aus Natur und Technik in Simulationsmodellen werden im Allgemeinen durch Differentialgleichungssysteme beschrieben. Das effiziente Lösen von diesen Differentialgleichungssystemen ist eine wichtige Aufgabe im High Performance Computing (HPC)-Umfeld. Zur Beschleunigung solcher Berechnungen stellt die Verwendung von Field Programmable Gate Arrays (FPGAs) einen vielversprechenden Ansatz dar. Mithilfe von Hardwarebeschreibungssprachen können sehr effiziente Logikentwürfe umgesetzt werden. Ihr hardwarenahes Konzept verhindert jedoch gleichzeitig eine weite Verbreitung unter Wissenschaftlern und Software-Ingenieuren. Der Einsatz von High Level Synthese (HLS)-Tools verspricht eine schnellere und einfachere Entwicklung, verbunden mit dem Risiko einer geringeren Performance und eines erhöhten Ressourcenbedarfs der finalen Umsetzung. Selbst bei der Verwendung von HLS-Tools benötigt der Benutzer spezielle Kenntnisse über FPGAs und Schaltungsentwicklung. Um eine weite Verbreitung von FPGAs in HPC-Anwendungen zu ermöglichen, wird ein einfach zu bedienendes Werkzeug zum Erstellen von perfomanten Schaltungen benötigt. Im Rahmen dieser Arbeit wird ein *Logikgenerator* vorgestellt, der in der Lage ist aus einfach zu handhabenden Konfigurationsdateien automatisch optimierte Löserschaltungen zu erzeugen. Weder manuelle Entwicklungsarbeit, noch spezielle FPGA- oder Programmierkenntnisse sind erforderlich. Zur Evaluation der entwickelten Software wurden Schaltungen zur Umsetzung unterschiedlicher Lösungsmethoden generiert und jeweils mit einer alternativen handoptimierten HLS-Implementierung verglichen. Die durch diesen neuen Ansatz, in dieser Arbeit, automatisch erzeugte Logik erreicht eine $21,7$- bis 774-fache schnellere Berechnung als die HLS-Alternative.

Abstract

The solving of ordinary differential equations (ODEs) used to describe simulation models is a main task in the high performance computing (HPC) domain. Field programmable gate arrays (FPGAs) are a promising platform to be used as accelerators for such calculations. While the usage of hardware description languages (HDLs) can produce very efficient logic designs, their unique concept is hard to adopt for scientists or software engineers. High-level synthesis tools (HLS) promise faster development although at the risk of lower performance and increased resource consumption of the final design. But even when using HLS tools the user requires special knowledge about FPGAs and circuit design. In order to reach a wide adoption of FPGAs in HPC applications the need of simple to use tools, which enable performant designs, was identified beforehand. This thesis proposes a framework able to automatically generate specific and optimized solver logic from easy to handle configuration files. No manual development, nor special FPGA or programming knowledge is required. To measure the capability of the proposed tool the performance was evaluated for different solver methods and compared to an alternative hand optimized HLS implementation. The logic generated by this new approach is 21.7 to 774 times faster than its HLS counterpart.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Diffenrentialgleichungssysteme werden zur Beschreibung vieler Zusammenhänge in Natur und Technik verwendet. Da sich für viele dieser Systeme jedoch keine explizite Lösung angeben lässt, ist die Verwendung numerischer Berechnungsmethoden notwendig. Im High Performance Computing (HPC)-Umfeld wird seit langem an einer effizienten Umsetzung dieser geforscht. Neben der Verwendung von CPUs ist auch die Verwendung von GPUs als Hardware inzwischen etabliert. In den letzten Jahren haben Field Programmable Gate Arrays (FPGAs) durch die Markteinführung und stärkere Verbreitung von Beschleunigerkarten auch im HPC-Bereich auf sich aufmerksam gemacht. Die durch den Nutzer frei verschaltbaren Logikbausteine sind seit Jahren in elektrotechnischen Anwendungen vertreten. Durch die neue Verbreitung von FPGAs in der Informatik wird aktuell intensiv geforscht, wie sich diese im HPC-Umfeld effizient verwendet lassen. Eine wesentliche Forschungsfrage an dieser Schnittstelle zwischen Elektrotechnik und Informatik ist, wie sich der individuelle Entwicklungsaufwand für jede neue Konfiguration reduzieren lässt.

1.2 Zielsetzung

Im Rahmen dieser Arbeit soll die Möglichkeit der automatischen Erzeugung optimierter Logikschaltungen zum Lösen von Differentialgleichungen auf FPGAs untersucht werden. Dabei wird das Ziel verfolgt, dass der Entwicklungsaufwand nur in den einmalig zu erstellenden Tools liegt und nicht bei jeder Änderung des Differentialgleichungssystems oder der Lösungsmethode anfällt. Die Schaltungsgenerierung auf Basis einer Konfigurationsdatei soll die Nutzung von FPGA-Beschleunigern auch Nutzern ohne FPGA- oder programmierspezifischer Kenntnisse ermöglichen. Die erzeugten Schaltungen sollen automatisch optimiert werden und somit eine performante Alternative zu anderen Ansätzen schaffen. In einem Vergleich mit dem aktuellen Stand der Forschung, speziell mit einem High Level Synthese (HLS)-Ansatz, wird eine Einordnung dieser Arbeit angestrebt. Unter dem Aspekt der Performance

wird neben dem Lösen eines einzelnen Anfangswertproblems auch das parallele Lösen mehrerer Anfangswertprobleme als Ziel verfolgt.

Neben diesen problemorientierten Zielen soll die Arbeit als Beispiel für die Nutzung von FPGAs zum Beschleunigen numerischer Berechnungen dienen und die dafür notwendigen Konzepte und Überlegungen beleuchten. Die Zielgruppe dieser Ausarbeitung sind unter anderem auch Informatikstudenten ohne spezielles Vorwissen. Daher ist ein ausführlicher Grundlagenteil in FPGAs, Hardwarebeschreibungssprachen und alle notwendigen Spezifika vorgesehen.

Auflistung der Ziele:

1. Automatische Generierung von Logikschaltungen aus Konfigurationsdateien
2. Nutzbarkeit ohne FPGA-spezifische Kenntnisse
3. Automatische Optimierung der Berechnungen zum Erreichen hoher Performance
4. Vermittlung der notwendigen Grundlagen

Um die zeitlichen Rahmenbedingungen einer Bachelorarbeit einhalten zu können, wurden mehrere Einschränkungen festgelegt. Somit begrenzt sich diese Arbeit auf explizite Runge-Kutta-Verfahren als Lösungsmethoden für Differentialgleichungen. Eine Ausweitung auf andere mathematischen Lösungsverfahren ist möglich. Die internen Signale arbeiten mit einer konfigurierbaren Festkommazahlendarstellung. Eine Ausweitung auf die Unterstützung von Gleitkommazahlen ist vorgesehen und mit moderatem Aufwand möglich. Zusätzlich wird sich auf die Unterstützung einer bestimmten FPGA-Beschleunigerkarte (siehe 5.1) beschränkt. Die Verwendung einer Beschleunigerkarte eines anderen Herstellers oder in einer eingebetten Anwendung erfordert Anpassungen.

Auflistung der Einschränkungen:

1. Begrenzung auf explizite Runge-Kutta-Verfahren
2. Festkommazahlendarstellung für alle Signale
3. Unterstützung nur für eine spezifische Hardware

1.3 Aufbau

Im Folgenden wird ein kurzer Überblick über den Aufbau der Arbeit und den Inhalt der einzelnen Kapitel vermittelt.

Zu Beginn der Arbeit (Kapitel 2) wird eine Einführung in die zum Verständis der Arbeit notwendigen Grundlagen der Schaltungsentwicklung gegeben und auf die Funktionsweise von FPGAs eingegangen. Auch die für diese Arbeit relevanten mathematischen Zusammenhänge werden erklärt. Das Kapitel kann wahlweise übersprungen oder als Nachschlagwerk genutzt werden.

In Kapitel 3 wird auf den aktuellen Stand der Forschung eingegangen. Dabei werden relevante Paper vorgestellt und diskutiert.

Im Anschluss wird in Kapitel 4 auf das Konzept dieser Arbeit eingegangen. Dieses wird auch im Vergleich zu den in Kapitel 3 vorgestellten Papern eingeordnet. Neben einer Übersicht über die einzelnen in dieser Arbeit implementierten Software- und Logikkomponenten wird auch die Benutzerschnittstelle vorgestellt.

Auf die Implementierung einzelner Details des Konzeptes wird in Kapitel 5 eingegangen. Neben der Vorstellung der verwendeten Hardware und Software liegt ein Fokus darauf, wie mathematische Berechnungen für den FPGA optimiert und auf ihm implementiert werden. Aber auch auf das Interface zwischen FPGA und CPU wird genauer eingegangen.

In Kapitel 6 wird das Ergebnis dieser Arbeit evaluiert und mit einem alternativen Ansatz verglichen.

Am Ende der Arbeit werden in Kapitel 7 die Erkenntnisse diser Arbeit zusammengefasst und ein Ausblick auf weitere mögliche Forschungsthemen gegeben.

2 Grundlagen

In diesem Kapitel werden die zum Verständnis dieser Arbeit benötigten Grundlagen vermittelt. Das Kapitel kann wahlweise vollständig übersprungen oder als Nachschlagwerk genutzt werden. Zu Beginn wird eine allgemeine Einführung in die Entwicklung von Logikschaltungen gegeben. Im Anschluss werden die Funktionsweise von FPGAs und die mathematischen Grundlagen zum Lösen von gewöhnlichen Differentialgleichungen erläutertert.

2.1 Schaltungsentwicklung

Die Komplexität von integrierten Schaltungen hat in den letzten Jahrzehnten enorm zugenommen. Als ein Beispiel dient die Anzahl von Transistoren in Mikroprozessoren, welche im Vergleich zu 1970 um mehr als das 10^7-fache gestiegen ist. Bereits seit dem letzten Jahrzehnt sind mehrere Milliarden Transistoren [@Alc19] in einem Prozessor üblich. Die klassische Schaltungsentwicklung und somit die Herstellung gezeichneter Entwürfe zur Dokumentation und Fertigung wurde von Computer Aided Design (CAD) Verfahren abgelöst. Die zur Schaltungsentwicklung verwendete Untermenge wird Electronic Design Automation (EDA) genannt. Nur so ist die Entwicklung moderner und komplexer Digitalschaltungen möglich. Im Folgenden wird auf die relevanten Grundlagen zur Schaltungsentwicklung für programmierbare Logikdevices eingegangen.

2.1.1 Abstraktionsebenen

Im modernen Schaltungsentwicklungsprozess unterscheidet man zwischen mehreren Abstraktionsebenen, auf denen ein Design beschrieben werden kann. In Abbildung 2.1 sind die gängigen Abstraktionsebenen dargestellt, auf welche im Folgenden auch nochmal einzeln eingegangen wird.

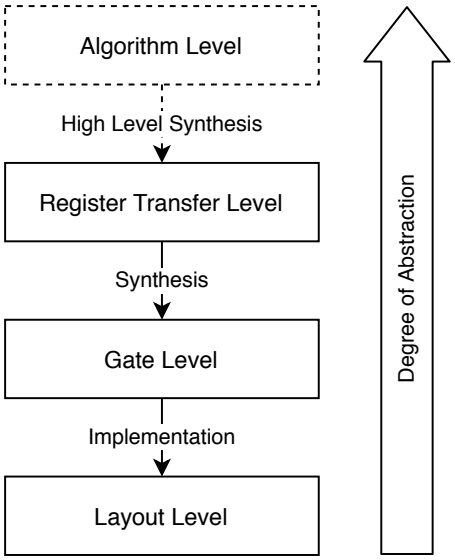

Abb. 2.1.: Abstraktionsebenen Schaltungsentwicklung

Layout Level

Das sogenannte Layout Level ist die niedrigste Abstrahierungsebene welche bei der Beschreibung einer Schaltung zum Einsatz kommt. Neben der Verschaltung der einzelnen Elemente ist auch die genaue räumliche Umsetzung in der Hardware spezifiziert.

Gate Level

Im Gate Level wird das Design mithilfe einer Netzliste beschrieben, in der die Verschaltung einzelner Logikgatter (and, or, xor, etc.) und Speicherblöcke definiert wird. Netzlisten werden meist automatisiert erstellt. Durch den Vorgang der Implementierung kann aus einer Netzliste eine Beschreibung auf dem Layout Level erzeugt werden. Dabei wird die Position der Elemente und der Verschaltung manuell oder automatisch festgelegt.

Register Transfer Level

Das Register Transfer Level (RTL) ist eine leicht abstrahierte und lesbare, meist in textueller aber auch grafischer Form verfasste, Beschreibung des Signalflusses eines Designs, aus dem durch Synthese eine Netzliste generiert werden kann. Im Gegensatz zum Gate Level ist die gemeinsame Beschreibung von mehreren Bitsignalen als ein mehrbittiges Signal möglich. Auch das mehrmalige Instantiieren von zu Modulen zusammengefasster Logik ist gängig. Die Abstrahierung des Register Transfer Levels bricht, im Gegensatz zu höheren Leveln, mit keinen konzeptuellen Eigenheiten der Hardwarebeschreibung.

Algorithm Level

Eine Lösungsstrategie für eine Aufgabe wird meist als Algorithmus, eine sequentielle Abfolge von Instruktionen, definiert. Eine solche Beschreibung ist auf dem Algorithm Level angesiedelt. Um die Einstiegshürde in die Entwicklung von Digitalschaltungen zu senken sowie die Entwicklungszeit zu verkürzen, wird seit mehreren Jahren unter anderem mit der sogenannten High Level Synthese (HLS) versucht, automatisiert Schaltungsbeschreibungen im RTL aus Algorithmen zu erzeugen. Dieser Prozess ist aufgrund fehlender Timing Informationen, wie etwaig gewünschter Parallelität oder Implementierungsdetails, jedoch komplex. Auf die Herausforderungen der HLS sowie die Unterschiede zur Hardwarebeschreibung auf dem RTL wird in Abschnitt 2.1.3 nochmals eingegangen.

2.1.2 Arten von Logik

Bei der Entwicklung von Digitalschaltungen unterscheidet man grundsätzlich zwei Arten von Logik: kombinatorische und sequentielle Logik. Reale Schaltungen bestehen aus einer Mischung beider Logikarten. Im Folgenden wird auf die einzelnen Merkmale beider Arten eingegangen.

Kombinatorische Logik

Bei kombinatorischer Logik wird der aktuelle Zustand der Ausgänge nur durch den derzeitigen Zustand der Eingänge definiert. Ändert sich ein Eingangssignal zu einem beliebigen Zeitpunkt, so ändern sich alle durch kombinatorische Logik abhängigen Signale unmittelbar. Anders formuliert besitzt kombinatorische Logik keinen internen Speicher und keine Rückkopplungen. Typische Anwendungen sind (En-/) Decoder und (De-) Multiplexer aber auch numerische Operationen, wie Multiplikation oder Addition.

Sequentielle Logik

Im Gegensatz zu kombinatorischer Logik sind die Ausgänge sequentieller Logik durch die Serie der Zustände der Eingänge definiert. Man unterscheidet zwischen takt- und ereignisgesteuerter Logik sowie einer Mischform aus beidem. Bei taktgesteuerter Logik wird die zeitliche Diskretisierung der Eingangssignale durch ein zusätzliches Taktsignal definiert. Man spricht deswegen auch von synchronen Schaltungen. Im Gegensatz dazu wird ereignisgeteuerte Logik auch asynchron genannt, da sie bei Aktivierung vorher festgelegter Signale sofort ihren Zustand ändert. Für das Speichern und Bereithalten von Zwischen- und Ausgangssignalen sind in beiden Fällen Register zuständig. Sequentielle Logik besitzt somit im Gegensatz zu kombinatorischer Logik eine Art Gedächtnis oder Zustand. Mögliche Beispiele sind einfache Zähler aber auch komplexe Zustandsautomaten.

2.1.3 Hardwarebeschreibung auf dem RTL

Eine zentrale Rolle bei der Entwicklung von integrierten Schaltungen nehmen die sogenannten Hardwarebeschreibungssprachen ein. Sie dienen zur formalen Beschreibung von Schaltungen, meist in textueller oder grafischer Form. Die so definierten Designs können automatisiert verifiziert und durch logische Synthese sowie Implementierung in die gewünschte Zielform übersetzt werden. Neben der Erstellung von Application Specific Integrated Circuits (ASICs) können auch Field Programmable Gate Arrays (FPGAs) als Zielplattform gewählt werden. Auch wenn die Anzahl der Hardwarebeschreibungssprachen hoch ist, haben sich zwei Sprachen als Industriestandard durchgesetzt. *Verilog* im amerikanisch und *VHDL* im europäisch geprägten Raum. Beide Sprachen bieten einen ähnlichen Funktionsumfang und Abstrahierungsgrad auf dem Register Transfer Level (RTL). Fast alle EDA-Tools unterstützen diese beiden Grundsprachen. So transkribiert der Großteil der höheren Sprachen, ob grafisch oder textuell, meistens nach *Verilog* oder *VHDL* und erlauben so die Nutzung verbreiteter Analyse-, Simulations- und Synthese-Tools. Vor allem Synthese-Tools für FPGAs sowie das Format des generierten Bitstreams sind im Allgemeinen proprietär. Somit bleibt nur die Verwendung der von dem FPGA-Hersteller vorgesehenen Software, welche oft nur *Verilog* und *VHDL* als Eingabe versteht.

Differenzierung von Kontrollfluss- und Datenflusssprachen

Die Beschreibung von Hardware unterscheidet sich grundsätzlich vom klassischem Programmieren, bei dem es darum geht einen sequentiell abzuarbeitenden Algorithmus zu beschreiben. Prozessoren nach der Von-Neumann-Architektur arbeiten eine Sequenz von Instruktionen ab. Imperative Programmiersprachen bieten Konstrukte wie Funktionsaufrufe, bedingte Ausführung und Schleifen, welche diesen Instruktionsfluss verändern können. So wird bei einer nicht erfüllten *if* Bedingung der Befehlszähler erhöht, um auf die nächsten auszuführenden Instruktionen zu springen. Die eigentliche Arbeit wird in mit Daten gefüllten Registern erledigt, welche durch die einzelnen Instruktionen verändert werden können. Dieser Ansatz ist in Abbildung 2.2 dargestellt und kann beschrieben werden als ein Fluss an Instruktionen, der durch Daten fließt. Die Daten bleiben an einem Ort und sind konstant, solange sie nicht durch den Instruktionsfluss geändert werden. Sprachen, die diesem Konzept folgen, werden Kontrollflusssprachen (engl. *Control Flow Languages*) genannt.

Durch Hardwarebeschreibungssprachen wird der Aufbau des Schaltungsdesigns in Form der Verbindungen einzelner Logiksignale und Register untereinander beschrieben. Im Gegensatz zu Programmiersprachen sind Hardwarebeschreibungssprachen

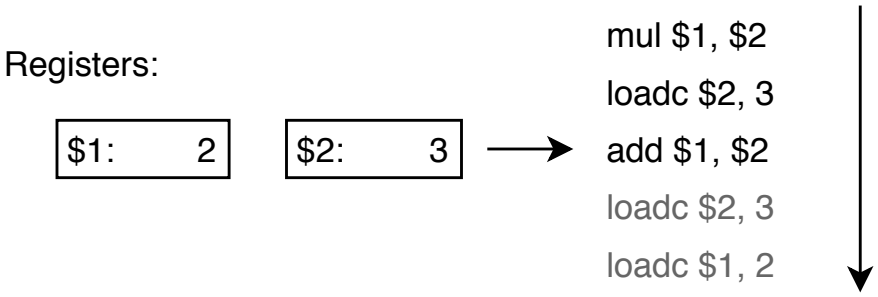

Abb. 2.2.: Kontrollfluss visualisiert

Datenflusssprachen. Bei diesen wird eine feste Verkettung von Operationen beschrieben, durch die einzelne Daten fließen und so verarbeitet werden. Dies ist in Abbildung 2.3 visualisiert. Elektrische Schaltungen oder Wasser, welches durch ein Netzwerk an Leitungen fließt, kann als Analogie herangezogen werden. Dieser grundlegende Konzeptunterschied erfordert ein Umdenken bei der Entwicklung von Schaltungen im Vergleich zu klassischer Software.

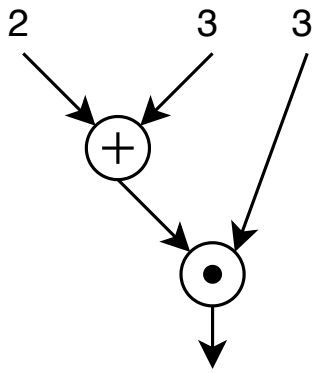

Abb. 2.3.: Datenfluss visualisiert

Abgrenzung zur High Level Synthese

Wenngleich klassische Programmiersprachen somit nicht zum Beschreiben von Hardware verwendet werden können, kommen sie bei der Entwicklung von Digitalschaltungen dennoch zum Einsatz. In Kontrollflusssprachen definierte Algorithmen lassen sich mithilfe von HLS in eine Schaltungsbeschreibung nach dem Datenflussprinzip übersetzen. Die erzeugte Hardwarebeschreibung liegt dann meistens in *Verilog* oder *VHDL*, also einer Hardwarebeschreibungssprache, auf dem RTL vor. Die Eingabe zur HLS erfolgt, unter Verwendung von Bibliotheken oder Frameworks (z.B. Open-

Cl [@Intc]), im Allgemeinen in ANSI C / C++ oder SystemC. Der grundlegende Konzeptunterschied zwischen Kontroll- und Datenflusssprachen sowie fehlende Informationen über Timing und Implementierungsdetails stellen auch für moderne HLS Software eine große Herausforderung dar. Einer eventuell schlechteren Performance oder Ressourcennutzung des resultierenden Designs steht ein deutlich geringerer Entwicklungsaufwand gegenüber. Die Abwägung zur Nutzung von HLS sollte von Fall zu Fall erfolgen. Auch eine kombinierte Nutzung von handgeschriebenem RTL-Code mit einzelnen übersetzten Algorithmen kann sinnvoll sein.

2.1.4 Workflow

Simulation und Verifikation

Grundsätzlich ähnelt der Workflow von Hardwareprojekten dem von Softwareprojekten in vielen Bereichen. Die Vorgehensweise bei Projektmanagement und Spezifikation ist vergleichbar. In industriellen Projekten kann jedoch häufig eine etwas konservativere Herangehensweise festgestellt werden. Im Gegensatz zur Softwareentwicklung ist Debugging von integrierten Schaltungen zeitaufwendig und komplex. Es kann nicht auf die internen Signale zugegriffen oder die Ausführung pausiert werden. Aus diesem Grund wird der Simulation und Verifikation des Designs ein sehr hoher Stellenwert zugeschrieben. Neben der allgemeinen Funktionalität können so auch in der Praxis schwer nachzustellende Ausnahmefälle geprüft werden. Vorwiegend werden Modultests (engl. *Unittests*) zur Überprüfung einzelner funktionaler Einheiten verwendet. Das Device Under Test (DUT) wird dafür meist auf dem RTL, gemeinsam mit einem zusätzlich beschriebenen, digitalen Prüfstand (engl. *Testbench*), der für das Steuern der Eingangs- und dem Überwachen der Ausgangssignale zuständig ist, simuliert. Neben vielen kommerziellen Simulatoren kann für Verilog insbesondere der unter der *GNU General Public License (GPL)* stehende *Icarus Verilog* [@Wil] empfohlen werden. Neben der automatisierten Überprüfung der Schnittstellen können bei der Simulation auch die internen Signale manuell analysiert werden. Dazu werden sogenannte Impulsdiagramme eingesetzt, die den Signalverlauf über die Zeit darstellen. In Abbildung 2.4 ist ein Beispiel für ein Impulsdiagramm, mit der ebenfalls unter der *GPL* stehenden Software *GTKWave* [@ByB], dargestellt.

Timing Analyse

Zusätzlich zu der funktionalen Verifikation der Designs muss auch überprüft werden, ob das durch die Implementierung generierte Layout den zeitlichen Einschränkungen

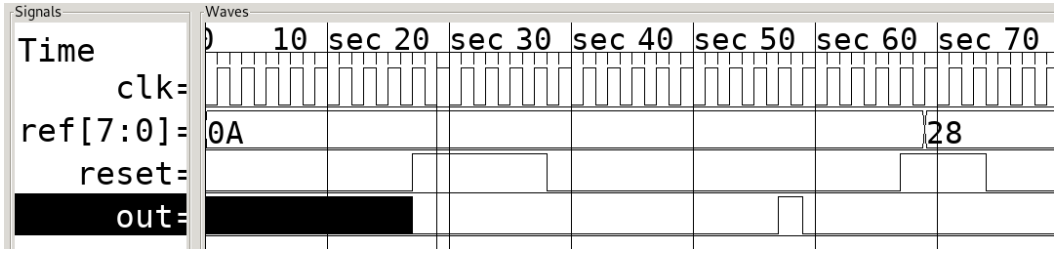

Abb. 2.4.: Impulsdiagramm - Quelle: Screenshot von *GTKWave* [@ByB]

(engl. *Timing Constraints*) gerecht wird. Dabei wesentlich ist die als Gatterlaufzeit (engl. *Propagation Delay*) bezeichnete Dauer, welche angibt, wie viel Zeit ein Signal vom Eingang der Schaltung bis zum Ausgang benötigt. Je komplexer die interne kombinatorische Logik und somit umso länger die Signalpfade in der Hardware, desto größer ist die benötigte Gatterlaufzeit. Für eine klassische synchrone Schaltung (nur Eintaktpfade (engl. *One Cylce Paths*)), deren zeitliche Zustände über ein Taktsignal (engl. *Clock Signal*) diskretisiert werden, gilt die logische Einschränkung, dass die Gatterlaufzeit zwischen allen Registern kleiner als die Periode des Taktes sein muss. Wird diese zeitliche Schranke nicht eingehalten, kann es vorkommen, dass eine Signaländerung noch nicht das Zielregister erreicht hat und somit das Register für den nächsten Takt einen alten falschen Wert enthält. Das Layout zeigt somit ein anderes Verhalten als erwartet. Kann kein Layout generiert werden, welches die zeitlichen Schranken einhält, muss der Entwickler das Design oder die Taktraten anpassen. Als *Timing Analyse* bezeichnet man die automatische Überprüfung über die Einhaltung aller zeitlichen Einschränkungen des Layouts. Dafür wird die Gatterlaufzeit aller Signale für alle wesentlichen Betriebspunkte, vorallem für unterschiedliche Temperaturen, ermittelt und mit den Vorgaben verglichen. Die Layoutvorgaben werden üblicherweise in dem als Industriestandard etablierten Synopsys Design Constraints (SDC) Format angegeben.

2.2 Field Programmable Gate Array

Ein Field Programmable Gate Array (übersetzt: im Feld programmierbare Gatter-Anordnung) ist ein integrierter Schaltkreis, der seine interne Verschaltung ändern kann, um unterschiedliche Logikfunktionen abzubilden. Dies ermöglicht die Nutzung von applikationsspezifischen Schaltungen ohne dafür spezielle integrierte Schaltkreise (ASICs) zu benötigen, bei deren Fertigung hohe Werkzeugkosten anfallen.

Ein Prozessor (hier von Von-Neumann-Architektur) ist eine Logikschaltung, die abhängig von einem während Laufzeit veränderbaren Befehlsstrom unterschiedli-

che Operationen sequentiell ausführt. Dabei werden unterschiedliche Teilbereiche der Rechenwerk-Schaltung genutzt, um zum Beispiel Addition oder Multiplikation auszuführen. Ein FPGA kann, unter Annahme ausreichender Ressourcen, zu jeglicher Logikschaltung verschaltet werden. So kann ein FPGA auch einen Prozessor beherbergen. Da dieser dann in veränderbarer Hardware vorliegt, nennt man ihn Softcore-Prozessor. Um ein Problem auf einem Prozessor zu lösen, wird eine Handlungsabfolge definiert, die einzelne sequentiell abzuarbeitenden Schritte enthält (Algorithmus). Der Durchsatz ist, vereinfacht betrachtet, begrenzt durch die zum Abarbeiten eines einzelnen Schritts benötigte Dauer und der Anzahl der Schritte. Ein klassischer Prozessor ist ein unveränderbarer integrierter Schaltkreis (selbe Hardware), der durch seinen Aufbau zur Laufzeit unterschiedliche Aufgaben erfüllen kann. Anstelle eines generischen Prozessors kann auf einen FPGA auch eine spezifische Logikschaltung zum Lösen eines Problems geladen werden. Alle Pfade der Aufgabe, welche parallel gelöst werden sollen, können physisch parallel aufgebaut werden. An jeder Stelle, an der eine numerische Operation benötigt wird, kann diese direkt implementiert werden. Der Durchsatz einer solchen spezialisierten Schaltung ist im Gegensatz zu dem Durchsatz eines Prozessors somit nur durch den Pfad mit der längsten Laufzeit begrenzt und nicht durch die Summe der benötigten Zeit für alle Teilschritte. Dies führt meist zu einem Performancevorteil und gleichzeitig höherer Engergieeffizienz. Im Folgenden wird auf den Aufbau von FPGAs und die Funktionsweise der logischen Blöcke sowie der Konfigurierung eingegangen.

2.2.1 Aufbau

Aus dem Namen Field Programmable Gate Array lässt sich bereits der Aufbau in Form einer Gatteranordnung ableiten. Als Grundelement kommen die sogenannten logischen Blöcke zum Einsatz, welche in der Lage sind simple Logikfunktionen abzubilden. Durch die Verschaltung mehrerer logischer Blöcke können komplexere Schaltungen gebildet werden. An den Rändern des Gatters gibt es spezielle I/O-Blöcke (Input-/Output- Blöcke), die den Signalpegel der internen Signale zur Kommunikation mit der Umgebung anpassen. Zusätzlich enthalten FPGAs weitere hart verschaltete Komponenten, da diese so weniger Platz benötigen und schneller getaktet werden können als eine verschaltete Alternative aus logischen Blöcken. Zu diesen zählen Random-Access Memory (RAM), aber auch Digital Signal Processing (DSP) Blöcke, welche meistens numerische Operationen (vorwiegend Multiplikationen) implementieren. In Abbildung 2.5 ist der Aufbau eines FPGAs dargestellt. Neben den einzelnen logischen Blöcken sowie den RAM- und DSP-Blöcken fallen vor allem die schwarzen Verbindungslinien auf. Diese stehen symbolisch für mehrere

Signalleitungen, die dort verlaufen. Das sogenannte Interconnect ist konfigurierbar und ermöglicht somit die dynamische Verschaltung zwischen den einzelnen Komponenten.

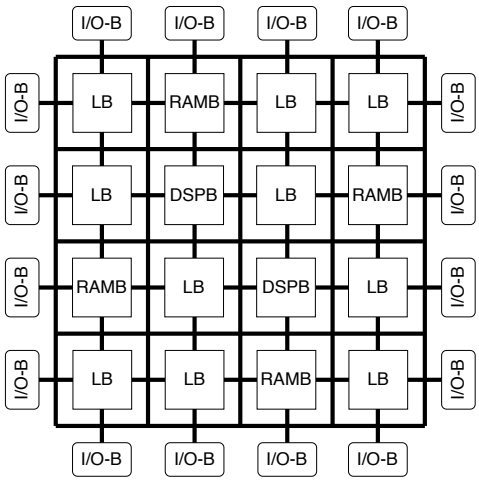

Abb. 2.5.: Aufbau eines FPGAs

2.2.2 Funktionsweise logischer Block

Der logische Block ist die Basiskomponente eines FPGAs. Seine Aufgabe ist die Abbildung einer simplen Logikfunktion. Seine Aus- und Eingänge können mit weiteren logischen Blöcken zur Gesamtlogikschaltung verbunden werden. Die Komplexität und Implementierungsweise eines logischen Blocks hängt vom Hersteller und dem FPGA selbst ab. Man spricht von einem n-bittigen logischen Block, wenn dieser n Eingänge hat und somit eine Binärfunktion mit n Variablen umsetzen kann.

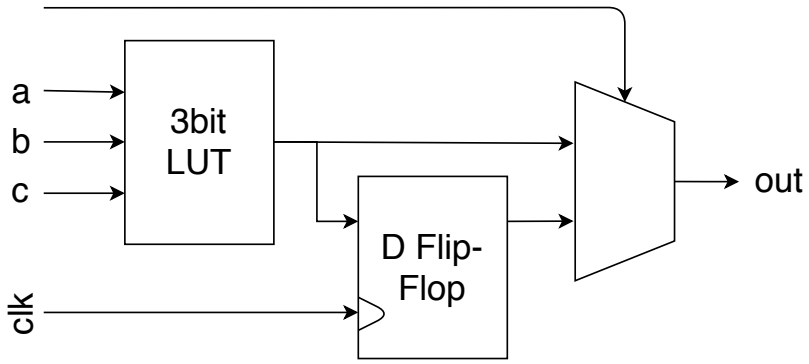

Abb. 2.6.: 3-bit Look-Up-Table (LUT) basierter logischer Block - Quelle: [Wan, S. 7]

Im Folgenden wird beispielhaft eine Look-Up-Table (LUT) basierte Lösung vorgestellt. Dabei enthält jeder n-bittige logische Block eine über n Signale addressierbare LUT.

Dadurch sind 2^n Bits ansprechbar. Folglich kann für jede mögliche Kombination der Variablen einer n-bittigen Binärfunktion das Ergebnis direkt abgespeichert werden. Dieses Verfahren wird anhand der 3-bittigen logischen Blocks aus Abbildung 2.6 und der folgenden Binärfunktion verdeutlicht:

$$out := (a \wedge b) \vee c$$

In Tabelle 2.1 werden alle möglichen Kombination der Variablen (Eingänge) und das jeweilige Ergebnis der Binärfunktion dargestellt. In der LUT müssen nur die 8 Ergebnisse (orange markiert) gespeichert werden. Diese können direkt an den $2^3 = 8$ über die Eingänge addressierbaren Orten abgespeichert werden. Dies zeigt, dass die LUT jede Binärfunktion mit 3 Variablen abbilden kann.

Tab. 2.1.: Beispiel Wahrheitstabelle 3-bit logischer Block

a	b	c	out
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Neben der LUT zum Auflösen der Binärfunktion enthält der logische Block aus Abbildung 2.6 auch noch einen D Flip-Flop, der als Dateneingang die Ausgabe der LUT erhält sowie einen Multiplexer, welcher entweder den Ausgang der LUT direkt oder den Ausgang des D Flip-Flops weiterreicht. Nur bei aktivem externen Trigger-Signal updatet sich der Ausgang des Flip-Flops. Dies erlaubt es, den Ausgang des logischen Blockes nur zu bestimmten diskreten Zeitpunkten zu aktualisieren und somit sequentielle Logik zu implementieren. Durch ein externes Steuersignal kann der Multiplexer angesteuert werden und somit dieses Verhalten aktiviert oder deaktiviert werden.

2.2.3 Funktionsweise Konfigurierung

Die Schaltung, welche ein FPGA abbildet, wird bestimmt durch die Inhalte der LUTs und den Status der Steuerleitungen für die Multiplexer aller logischen Blöcke sowie die Kombination der logischen Blöcke untereinander. Die Summe dieser

Informationen wird Konfiguration genannt. Eine weit verbreitete Vorgehensweise ist die Vorhaltung der Konfiguration in integrierten RAM-Zellen. Dabei kodieren die einzelnen Bits bestimmte Konfigurationsentscheidungen, wie den Zustand der Transistoren des Interconnects oder den Wert der Steuerleitungen der Multiplexer aller logischen Blöcke. Auch die Daten der LUTs können dort hinterlegt werden. In Abbildung 2.7 ist dieses Verfahren dargestellt. Die Reihenfolge der einzelnen Bits im RAM, welche somit eindeutig die Konfiguration des FPGAs bestimmt, wird auch Bitstream genannt.

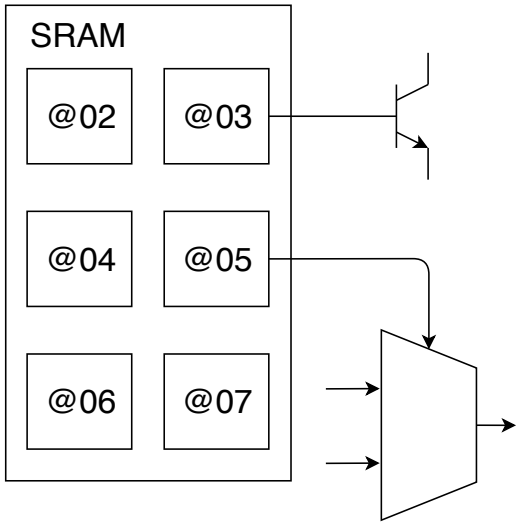

Abb. 2.7.: RAM-basierte Konfiguration

Um eine Logikschaltung auf einem FPGA zu realisieren, muss ein solcher pro Schaltung und FPGA individueller Bitstream erzeugt werden. Ein wesentlicher Teil dieses Vorgangs, der Implementierung genannt wird, ist auch die Erkennung und Zuordnung von Logik-Anteilen, welche durch DSP- oder RAM-Blöcke abgebildet werden können. Um dies zu ermöglichen muss bei der Entwicklung von Logik für FPGAs auf bestimmte Richtlinien Rücksicht genommen werden [Inte]. Ein Bitstream kodiert dann für die gegebene Hardware alle notwendigen Konfigurationsentscheidungen, das Format ist allerdings meist proprietär. Je nach FPGA kann der Bitstream auf persistenten internen oder externen Speicher geflasht werden. Beim Start wird die Konfiguration dann in den internen RAM geladen und somit alle Komponenten wie gewollt verschaltet. Manche FPGAs unterstützten zur Laufzeit eine Rekonfiguration einzelner Logikteile (engl. *Partial Reconfiguration*), dabei werden nur die für den jeweiligen Bereich verantwortlichen Teile des Bitstreams geladen.

2.3 Mathematische Grundlagen

In diesem Kapitel sollen die dieser Arbeit zugrundeliegenden mathematischen Zusammenhänge erläutert werden. Dafür wird eine Einführung in Diffenretialgleichungen sowie numerischen Lösungsmethoden für diese vermittelt. Wie bereits im Abschnitt Zielsetzung festgelegt, wird sich dabei auf explizite Runge-Kutta-Verfahren zur Lösung von Anfangswertproblemen gewöhnlicher Differentialgleichungen beschränkt.

2.3.1 Gewöhnliche Differentialgleichungen

Eine Differentialgleichung definiert eine mathematische Beziehung zwischen einer gesuchten Funktion, den Variablen, von denen die Funktion abhängt, sowie Ableitungen der Funktion. Eine Vielzahl von Zusammenhängen in Natur und Technik lassen sich durch Differentialgleichungen beschreiben. Als typische Beispiele dienen mechanische Bewegungen oder Schwingungen in der Physik, die Evolutionstheorie in der Biologie, die Bewegung von Himmelskörpern in der Astronomie oder die Kinetik von Reaktionen in der Chemie. Die Gleichung 2.1 ist ein Beispiel für eine sehr einfache Differentialgleichung:

$$y'(x) = 2y(x) \tag{2.1}$$

In diesem Fall ist die gesuchte Funktion $y(x)$, die unabhängige Variable wäre x und mit $y'(x)$ ist eine Ableitung der gesuchten Funktion vertreten. Eine Funktion y, die für alle Werte der unabhängigen Variablen die Differentialgleichung erfüllt, wird Lösung genannt. Im Falle unseres Beispiels (Gleichung 2.1) ist die Funktion $y(x) = e^{2x}$ eine mögliche Lösung, wie durch Einsetzen geprüft werden kann. Allgemein unterscheidet man zwischen gewöhnlichen und partiellen Differentialgleichungen. Eine gewöhnliche Differentialgleichung (engl. *Ordinary Differential Equation* oder auch *ODE*) liegt dann vor, wenn die gesuchte Funktion nur von einer unabhängigen Variable abhängt, wie es auch in unserem Beispiel (Gleichung 2.1) der Fall ist. Im Falle von partiellen Differentialgleichungen (engl. *Partial Differential Equation* oder auch *PDE*) treten durch die Abhängigkeit von mehreren Variablen partielle Ableitungen auf. Im Rahmen dieser Arbeit werden ausschließlich gewöhnliche Differentialgleichungen betrachtet. Die Ordnung einer Differentialgleichung entspricht der höchsten Ordnung der auftretenden Ableitungen. Ist eine Differentialgleichung so formuliert, dass die höchste Ableitung alleine auf einer Seite steht, wird

sie explizit genannt. Sonst spricht man von einer impliziten Differentialgleichung. Eine explizite Differentialgleichung der 1. Ordnung hat somit allgemein folgende Form mit der stetigen Funktion $f\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$:

$$y'(x) = f(x, y(x)) \tag{2.2}$$

Im Fall unseres Beispiels (Gleichung 2.1) gilt $f(x, y(x)) = 2y(x)$. In den meisten Fällen betrachtet man jedoch statt einer einzelnen Differentialgleichung ein Differentialgleichungssystem. Bei einem expliziten Differentialgleichungssystem mit mehreren unbekannten Funktionen müssen alle Gleichungen die selbe unabhängige Variable besitzen und können alle gesuchten Funktionen und deren Ableitungen enthalten. Als ein Beispiel für ein Gleichungssystem dienen die beiden Lotka-Volterra Gleichungen (A.1.2), welche auch im Rahmen der Arbeit verwendet werden:

$$\begin{aligned} y_1'(x) &= 0.1 \cdot y_1(x) - 0.2 \cdot y_1(x) \cdot y_2(x) \\ y_2'(x) &= -0.2 \cdot y_1(x) + 0.4 \cdot y_1(x) \cdot y_2(x) \end{aligned} \tag{2.3}$$

Das Gleichungssystem erster Ordnung 2.3 besteht aus zwei expliziten Differentialgleichungen, die beide von der Variable x abhängen. Beide Gleichungen enthalten die gesuchten Funktionen y_1 und y_2. Im Folgenden wird eine Differentialgleichung auch in der Form $y' = f(x, y)$ angegeben mit der gesuchten Funktion $y = (y_1, \ldots, y_n)\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}^n$ und $f = (f_1, \ldots, f_n)\colon \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$, welches die rechte Seite des Systems darstellt. Üblicherweise gibt es unendlich viele Lösungen für eine Differentialgleichung. Für das einfache Beispiel der Gleichung 2.1 gibt es neben der bereits angegebenen Lösung $y(x) = e^{2x}$ noch unendlich viele weitere Lösungen: $y(x) = ke^{2x}$ mit $k \in \mathbb{R}$. Durch weitere Bedingungen können die Lösungen auf einzelne Funktionen eingeschränkt werden. Zum Beispiel durch die Vorgabe eines Anfangswertes, also der Forderung dass $y(x)$ für ein gegebenes x_0 einen bestimmten Wert y_0 annimmt: $y(x_0) = y_0$. Wird nach einer Lösung für die Beispiel-Differentialgleichung 2.1 mit der zusätzlichen Anfangswertbedingung $y(0) = 3$ gefragt, dann ist die einzige Lösung $y(x) = 3e^{2x}$. Das Lösen einer Differentialgleichung mit zusätzlicher Anfangswertbedingung wird Anfangswertproblem genannt. Im Falle unserer Definition für Differentialgleichungssysteme gilt die Anfangsbedingung $y(x_0) = y_0$ wobei $y_0 = (y_{1,0}, \ldots, y_{n,0}) \in \mathbb{R}^n$ ist.

2.3.2 Euler-Verfahren

Nur in seltenen Fällen kann für ein Differentialgleichungssystem eine explizite Lösung angegeben werden. Daher werden Anfangswertprobleme oft in numerischer Form gelöst. Dabei wird der Funktionsverlauf der Lösungsfunktion näherungsweise Punkt für Punkt bestimmt. Meist ist der Abstand zwischen den Abtastpunkten äquidistant und wird Schrittweite h genannt. Der einfachste Vertreter solcher Lösungsverfahren ist das explizite Euler-Verfahren mit dem die Näherungswerte η_k der exakten Lösung $y(x)$ für $k = 0, 1, 2, \ldots$ für ein explizites Differentialgleichungssystem ermittelt werden können. Im Folgenden ist das Iterationsverfahren dargestellt:

$$\begin{aligned} \eta_0 &= y_0 \\ \eta_{k+1} &= \eta_k + h \cdot f(x_k, y_k) \\ x_{k+1} &= x_k + h \end{aligned} \tag{2.4}$$

Das Euler-Verfahren berechnet folglich den jeweils nächsten Näherungswert auf Basis des Wertes und der Steigung des vorherigen Schrittes. Somit ist das Lösungsverfahren inhärent sequentiell, zur Berechnung des Wertes η_{k+1} muss der Näherungswert η_k bereits vorher berechnet worden sein. In Abbildung 2.8 ist die exakte Lösung ($y(x) = 3e^{2x}$) für das Beispiel Anfangswertproblem mit der Differentialgleichung 2.1 und der Vorbedingung $y(0) = 3$ sowie zwei Approximationen durch das Euler-Verfahren mit unterschiedlicher Schrittweite dargestellt. Dabei ist der Einfluss der Schrittweite auf die Genauigkeit der Näherungswerte zu erkennen. Wird die Schrittweite geringer gewählt und werden somit mehr Schritte im gleichen Zeitraum berechnet, steigt die Genauigkeit der Approximation.

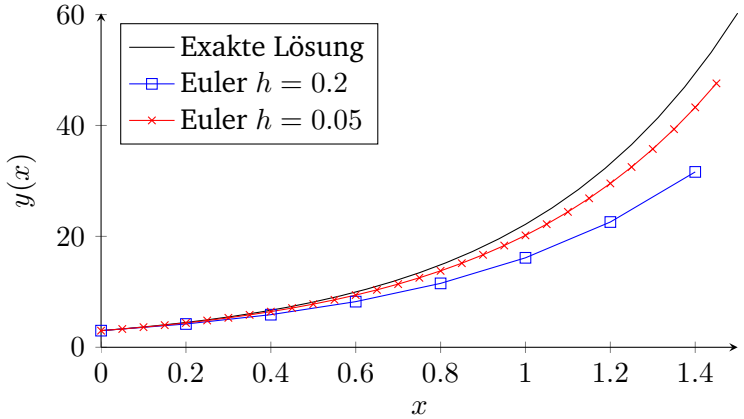

Abb. 2.8.: Darstellung des Euler-Verfahrens für unterschiedliche Schrittweiten

2.3.3 Runge-Kutta-Verfahren

Die Runge-Kutta-Verfahren sind eine Klasse von Lösungsverfahren, zu denen auch das Euler-Verfahren gehört. Allgemein ist die Idee dabei, die Funktion $f(x, y)$ an, je nach Variante, einer oder mehreren vordefinierten Stellen im Intervall $[x_k, x_{k+1}]$ auszuwerten. Ein s-stufiges Runge-Kutta-Verfahren nimmt zur Berechnung eines Näherungswertes s Auswertungen der Funktion $f(x, y)$ vor. Das bereits vorgestellte Euler-Verfahren ist als 1-stufiges Verfahren einer der einfachsten Vertreter der Klasse. In jedem Schritt eines s-stufigen Runge-Kutta-Verfahrens wird zuerst für jede Stufe ein Stufenvektor $v_1, \ldots, v_s \in \mathbb{R}^n$ berechnet:

$$v_j = f\left(x_k + h \cdot c_j, \eta_k + h \cdot \sum_{l=1}^{s} a_{jl} v_l\right) \quad \text{für } j = 1, \ldots, s \tag{2.5}$$

Dabei sind die Koeffizienten $c = (c_1, \ldots, c_s) \in \mathbb{R}^s$, $A = (a_{kl} \in \mathbb{R}^{s \times s})$ sowie $b = (b_1, \ldots, b_s) \in \mathbb{R}^s$, welche zur Berechnung des Näherungswertes benötigt werden, von dem individuell verwendeten Runge-Kutta-Verfahren abhängig. Nach der Berechnung der Stufenvektoren kann die Approximation des Punktes wie folgt berechnet werden:

$$\eta_{k+1} = \eta_k + h \cdot \sum_{j=1}^{s} b_j k_j \tag{2.6}$$

Die Koeffizienten eines Verfahrens werden üblicherweise im sogenannten Runge-Kutta-Tableau (Butcher-Schema / -Tableau, engl. *Butcher Array*) angegeben. Gleichung 2.7 stellt die allgemeine Form eines Runge-Kutta-Tableaus dar, welches alle benötigten Informationen zur Implementierung eines bestimmten Runge-Kutta-Verfahrens enthält. Auch bei Runge-Kutta-Verfahren unterscheidet man zwischen expliziten und impliziten Verfahren, welche sich stark in ihren numerischen Eigenschaften und somit auch ihrem Anwendungsfall unterscheiden. Diese Arbeit beschränkt sich auf explizite Runge-Kutta-Verfahren. Für diese gilt, dass die Einträ-

ge $a_{jl} = 0$ für alle $l \geq j$ sind, daher enthält das Runge-Kutta-Tableau nur Nullen oberhalb der Diagonale.

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b^T} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{2.7}$$

Die in dieser Arbeit verwendeten Verfahren sind in Form ihrer Runge-Kutta-Tableaus im Anhang unter A.2 zu finden.

3 Stand der Forschung

Bereits 2005 wurde von Osana et al. [Osa+05] ein Framework zur Simulation von, auf Differentialgleichungen basierenden, biochemischen Zellprozessen auf einem FPGA vorgeschlagen. Die Wahl eines FPGAs als Plattform wird dabei als kompakte und kosteneffektive Lösung herausgestellt. Durch die starke Eingrenzung des Anwendungsfeldes und somit der geringen Diversität an relevanten Differentialgleichungen (19), konnten die Autoren für jede Funktion einen Löser (engl. *Solver*) in einer Hardwarebeschreibungssprache (Verilog) implementieren. Das entwickelte Framework ist neben der Ansteuerung auch für die Zusammenstellung des FPGA Designs, aus den handgeschriebenen Lösern und weiteren statischen Komponenten, verantwortlich. Die Nutzung fortgeschrittener Lösungsmethoden, wie Heun oder Runge-Kutta anstatt des verwendeten Euler-Verfahrens, wird im Ausblick vorgeschlagen.

Auch Fasih et al. [Fas+09] haben sich 2009 mit der Frage beschäftigt, wie man Differentialgleichungssysteme effizient auf FPGAs lösen kann. In ihrer Arbeit haben sie am Beispiel der Rössler Gleichungen (A.1.1), nach ihren Worten, eine neue Methode zum Lösen von Differentialgleichungen höherer Ordnung demonstriert. Ihren Ansatz leiten sie dabei von klassischen Analogrechnern ab, die notwendigen Berechnungen sollen möglichst direkt in Hardware nachgebaut werden. Zur Integration wird das klassische Euler-Verfahren verwendet, auch wenn dieses keine Nennung in der Arbeit findet. Eine Auseinandersetzung mit anderen numerischen Lösungsmethoden findet nicht statt. Die sequentielle Logik des vorgeschlagenen Lösers arbeitet mit einem gemeinsamen Taktsignal. In jedem Zyklus des Taktes wird die vollständige Funktionsauswertung sowie ein Schritt des Euler-Verfahrens durchgeführt. Der Umgang mit komplexeren Gleichungssystemen, deren Funktionsauswertung nicht mithilfe rein kombinatorischer Logik innerhalb eines Taktschrittes möglich ist, wird von Fasih et al. nicht behandelt. Die Autoren empfehlen die Nutzung einer, der gewünschten und benötigten Genauigkeit angepassten, Festkomma-Darstellung zur Repräsentation aller interner Signale. Dies ermögliche eine Ressourceneinsparung sowie schnellere Berechnungen im Vergleich zu einer Gleitkomma-Darstellung. Bei der Wahl der Implementierung wählen die Autoren die gleiche Vorgehensweise wie Osana et al. [Osa+05]. Der vorgeschlagene Löser wird manuell durch Verilog Code beschrieben. Im Ausblick wird die automatische Generierung von Lösern aus einem Datenfluss, in textueller oder visueller Darstellung, vorgeschlagen.

Des Weiteren beschäftigen sich Stamoulias et al. [Sta+17] in ihrer Arbeit von 2017 mit der Entwicklung von FPGA-Schaltungen zum Lösen von Differentialgleichungen. Bei der vorgeschlagenen Methodik kommt jedoch im Gegensatz zu [Fas+09] und [Osa+05] keine Hardwarebeschreibungssprache zum Einsatz. Die Nutzung von HLS soll eine einfachere Anpassbarkeit an andere Differentialgleichungssysteme und Lösungsmethoden ermöglichen. Gegebenenfalls eintretende Nachteile wurden hierbei nicht untersucht. Im Rahmen der Arbeit wurden acht Löser für die Lotka-Volterra Gleichungen (A.1.2) von Hand implementiert, welche sich in der verwendeten Gleitkomma-Darstellung (einfache und doppelte) sowie der Lösungsmethode unterscheiden: Euler (A.2.1), ModEuler (A.2.2), Heun (A.2.3) und SSPRK3 (A.2.4). Die Autoren vergleichen die vorgeschlagenen Löser unter dem Gesichtspunkt der Performance und Ressourcennutzung ausführlich. Ein Vergleich zu in einer Hardwarebeschreibungssprache formulierten Lösern findet nicht statt. Durch zusätzliche Direktiven wird das HLS-Tool angewiesen, die Berechnungen auf dem FPGA in Form einer Pipeline aufzubauen und somit die parallele Abarbeitung mehrerer unabhängiger Anfangswertprobleme innerhalb eines Lösers zu ermöglichen. Die Gesamtperformance bei paralleler Abarbeitung wird weiter gesteigert, indem die Anzahl dieser von Hand implementierten Löser in den Schaltungen erhöht wird. Bei einem Vergleich mit einer Single-Core CPU konnte eine Beschleunigung der Berechnung um das bis zu 14-fache ermittelt werden.

Einen grundlegend anderen Ansatz verfolgen Huang et al. [HVG11]. Anstatt, wie bei [Osa+05], [Fas+09] sowie [Sta+17] für jedes Gleichungssystem und die verwendete Lösungsmethode eine spezifische Schaltung zu entwickeln, wird in dieser Arbeit ein spezieller Softcore-Prozessor für FPGAs vorgeschlagen, der darauf optimiert ist, effizient Differentialgleichungssysteme zu lösen. Die Autoren stellen den signifikanten, menschlichen Aufwand zur Entwicklung von Schaltungen auf dem RTL als Hindernis zur breiteren Adaption dar. HLS-Tools vertragen sich laut den Autoren nicht mit der hohen Komplexität mancher physikalischer Modelle und generieren Schaltungen mit einem höheren Ressourcenbedarf als notwendig. Eine schlechtere Performance wird von den Autoren im Hinblick auf eine Vermeidung des je Gleichungssystem und Lösungsmethode notwendigen spezifischen Entwicklungsaufwandes in Kauf genommen. Das *Differential Equation Processing Element (DEPE)*, wie der Softcore-Prozessor getauft wurde, verwendet intern eine Festkomma-Darstellung.

4 Konzept

Im Rahmen dieser Arbeit wird ein alternatives Konzept zum effizienten Lösen von Differentialgleichungen auf FPGAs vorgeschlagen: die automatisierte Generierung von Logikschaltungen auf dem RTL. Die große Hürde bei der Nutzung von FPGAs als Rechenbeschleuniger ist der hohe manuelle Entwicklungsaufwand, der für jedes neue Differentialgleichungssystem und jede neue Lösungsmethode anfällt. Selbst bei der Nutzung von HLS-Tools, wie durch Stamoulias et al. [Sta+17] vorgeschlagen, ist die manuelle Implementierung und anschließende Optimierung der Algorithmen zur Nutzung auf FPGAs notwendig und erfordert somit neben Programmier- auch FPGA-spezifische Kenntnisse. In dem hier vorgeschlagenen Konzept wird, nach Bereitstellung einer Konfigurationsdatei, in der Gleichungssystem und gewünschte Lösungsmethode spezifiziert sind, automatisiert eine optimierte Schaltung zum Lösen erzeugt. Von Seiten des Anwenders sind keine Programmier- oder gar FPGA-Kenntnisse erforderlich. Die grundlegende Herangehensweise kann auf andere numerische Probleme übertragen werden.

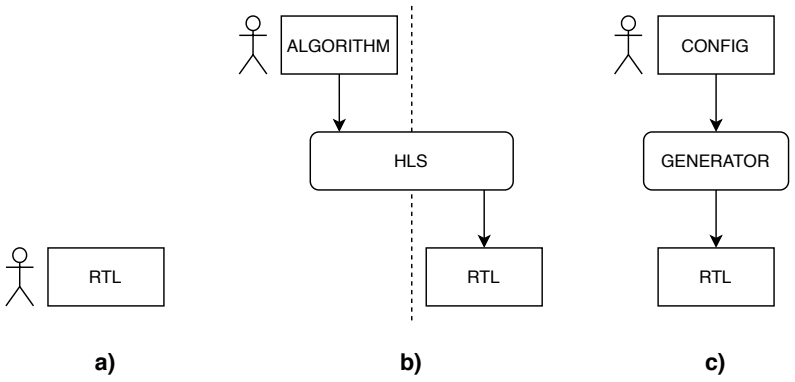

Abb. 4.1.: Workflow Vergleich: *(a)* handimplementierter RTL Code, *(b)* HLS Ansatz, *(c)* generierter RTL Code

In Abbildung 4.1 werden die Workflows der unterschiedlichen Ansätze aus Stand der Technik und dieser Arbeit dargestellt. Sowohl Osana et al. [Osa+05] als auch Fasih et al. [Fas+09] verfolgen einen Ansatz, bei dem der Nutzer die Logik zum Lösen der Differentialgleichungen in einer Hardwarebeschreibungssprache auf dem RTL von Hand implementiert *(a)*. Bei der Herangehensweise von Stamoulias et al. [Sta+17] *(b)* implementiert der Nutzer einen Algorithmus manuell in einer Kontroll-

flusssprache. Dieser wird dann durch HLS automatisiert in eine RTL-Repräsentation übersetzt. Auch das in dieser Arbeit vorgestellte Konzept *(c)* führt eine weitere Abstrahierungsschicht oberhalb des RTL ein. Der Nutzer spezifiziert dort in einer Konfigurationsdatei alle notwendigen Informationen, die zur automatischen Generierung des Lösers benötigt werden. Die als Teil dieser Arbeit vorgeschlagene Softwarekomponente zum Erzeugen von RTL-Beschreibungen der Löser-Schaltungen aus den Konfigurationsdateien, wird im Folgenden *Logikgenerator* genannt. Durch den sehr umfassenden Anspruch, Algorithmen in Hardware übersetzen zu können und dem damit einhergehenden notwendigen Paradigmenwechsel von einem Kontrollfluss- zu einem Datenflussmodell, ist HLS ein überaus komplexer Prozess. Der in dieser Arbeit vorgestellte Logikgenerator hingegen hat ein eingeschränktes Anwendungsgebiet (Ziel: Generierung von Lösern für Differentialgleichungen). Die Einschränkung ermöglicht einen hohen Optimierungsgrad.

Um die Vorteile eines FPGAs zu nutzen, erzeugt der entwickelte Generator die Logik für alle Berechnungen möglichst parallel in Hardware. Register und interner RAM werden verwendet, um Daten lokal vorzuhalten und somit Speicherzugriffe zu reduzieren. Im Rahmen dieser Arbeit wurde aus zeitlichen Gründen nur eine Unterstützung für Festkommazahlen implementiert. Dabei kann die Anzahl der binären Vor- und Nachkommastellen frei gewählt werden. Eine zukünftige Unterstützung für Gleitkommazahlen, einfacher und doppelter Präzision, ist vorgesehen und umsetzbar. Alle Berechnungen werden zum Zeitpunkt der Generierung automatisiert optimiert. So werden zum Beispiel konstante Berechnungen durch das Ergebnis und Multiplikationen der Festkommazahlen mit einem Faktor von 2 durch eine Bit-Schiebe-Operation ersetzt. Bei diesen Anpassungen entfallen die sonst notwendigen Berechnungen vollständig. Um nicht nur ein einzelnes Anfangswertproblem, sondern auch mehrere simultan, schnell lösen zu können, ist jede Lösereinheit vollständig als Berechnungspipeline aufgebaut. Dadurch kann die Logik in jedem Taktzyklus genutzt werden. Um die parallele Performance weiter zu steigern, können die vom Logikgenerator erzeugten Schaltungen mehrere solcher unabhängigen Lösereinheiten enthalten.

Die im Rahmen dieser Arbeit entwickelte Software wurde für den Einsatz auf einer *Intel Arria 10 GX Acceleration Card* [@Intd] entwickelt. Zum Einsatz auf einer Beschleunigerkarte eines anderen Herstellers oder in eingebetteten Systemen sind Anpassungen notwendig. Das zugrundeliegende Konzept ist allerdings allgemeingültig.

4.1 Komponenten

Die im Rahmen der Arbeit entwickelte Software lässt sich in drei Bereiche unterteilen, deren Funktion wird im Folgenden vorgestellt:

Framework Das *Framework* besteht aus mehreren Bibliotheken und Logikkomponenten, welche zur Umsetzung numerischer Berechnungen dienen. Neben der Bereitstellung von Hilfsfunktionen wird eine eingebettete Domain Specific Language (DSL) zur Beschreibung des Datenflusses von Berechnungen zur Verfügung gestellt. Die so formulierten Berechnungen mit ihren Abhängigkeiten untereinander können automatisiert optimiert und in parallele Pipeline-Logik übersetzt werden.

Logikgenerator Das *Logikgenerator*-Modul instantiiert auf Basis der gegebenen Konfiguration die Logik für einen Löser. Die numerischen Berechnungen zum Lösen des Anfangswertproblems sind in der durch das *Framework* zur Verfügung gestellten DSL formuliert. Unveränderbar in die Logik eines Lösers eingearbeitet sind, das durch die Konfiguration angegebene Differentialgleichungssystem sowie die dort definierte Lösungsmethode. Zur Laufzeit können jedoch die Startwerte, Schrittweite und -größe der zu lösenden Anfangswertprobleme angepasst werden. Der Generator erzeugt eine RTL-Beschreibung der Schaltung in Verilog und stößt automatisch die Synthese sowie Implementierung dieser an. Der erzeugte Bitstream wird mit weiteren Informationen, die unter anderem von der *Runtime* benötigt werden, angereichert und in einem speziellen Dateiformat (Dateiendung: .slv) gespeichert.

Runtime Die *Runtime*-Komponente stellt ein Interface für die Nutzung der generierten Löser dar. Neben einer Programmierschnittstelle werden auch Kommandozeilenprogramme zum Benchmarken und manuellen Lösen einzelner Anfangswertprobleme zur Verfügung gestellt. Als Eingabe dient eine .slv-Datei, aus der die notwendigen Informationen und der Bitstream zum Konfigurieren des FPGAs extrahiert werden.

4.2 Aufbau der Logikschaltungen

In Abbildung 4.2 ist stark vereinfacht der Aufbau einer einzelnen Lösereinheit dargestellt. Die Pfeile stellen den Datenfluss zwischen den Komponenten dar. Eine Lösereinheit besteht aus einer, durch das gewählte Lösungsverfahren festgelegten,

Anzahl an Stufen. Bei der in Abbildung 4.2 dargestellten Lösereinheit kommt ein dreistufiges Runge-Kutta-Verfahren zum Lösen eines Systems aus drei Differentialgleichungen (k, l, m) zum Einsatz. Die einzelnen Stufen werden von den Daten sequentiell nacheinander durchlaufen. Innerhalb einer Stufe wird für jede Komponente des Differentialgleichungssystems der jeweilige Stufenvektor (2.5) berechnet. Ein Durchlauf durch den Löser und somit durch alle Stufen entspricht einem Schritt des Runge-Kutta-Verfahrens. Ein einzelnes Anfangswertproblem durchläuft einen Löser mehrfach, solange bis die gewünschte Anzahl der Schritte absolviert wurde.

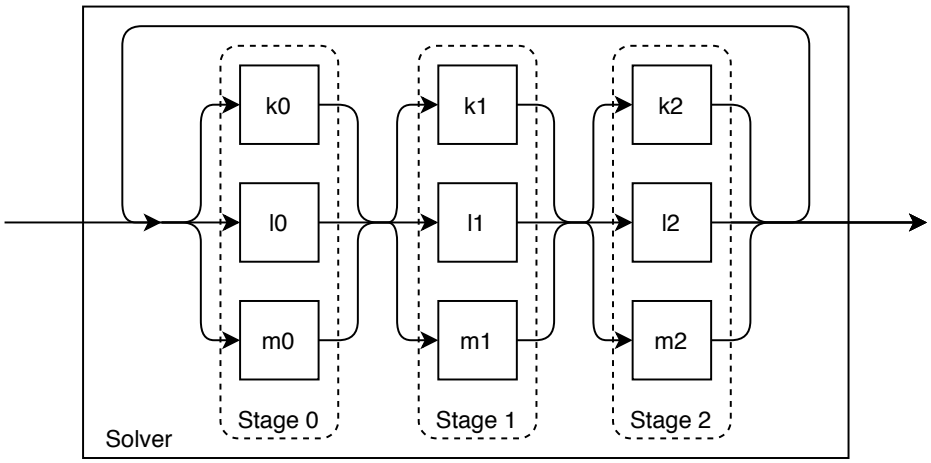

Abb. 4.2.: Aufbau einer Lösereinheit

Um die parallele Performance der Gesamtschaltung zu verbessern, kommen meist mehrere Lösereinheiten zum Einsatz. Weitere Logikkomponenten sind notwendig, um die zu berechnenden Anfangswertprobleme intern zu verteilen und den Datenaustausch mit der CPU zu gewährleisten. In Abbildung 4.3 sind diese dargestellt. Der *Host RAM Handler* ist für das Laden und Speichern der Anfangswertprobleme sowie der Ergebnisse im Hauptspeicher der CPU zuständig. Ein Anfangswertproblem besteht dabei aus einem Datensatz mit den Startwerten der unabhängigen Variable, den Startwerten der Systemgleichungen sowie der gewünschten Schrittweite und der gewünschten Anzahl der Schritte. Durch die Angabe einer Kennung kann das Ergebnis später wieder einem bestimmten Anfangswertproblem zugeordnet werden. Der *Dispatcher* ist für die Zuweisung der Datenpakete an die einzelnen Lösereinheiten sowie die Weiterleitung der Ergebnisse an den *Host RAM Handler* zuständig. Dieser überträgt dann für jedes gelöste Anfangswertproblem die angegebene Kennung, den finalen Wert der unabhängigen Variable sowie die berechneten Näherungswerte.

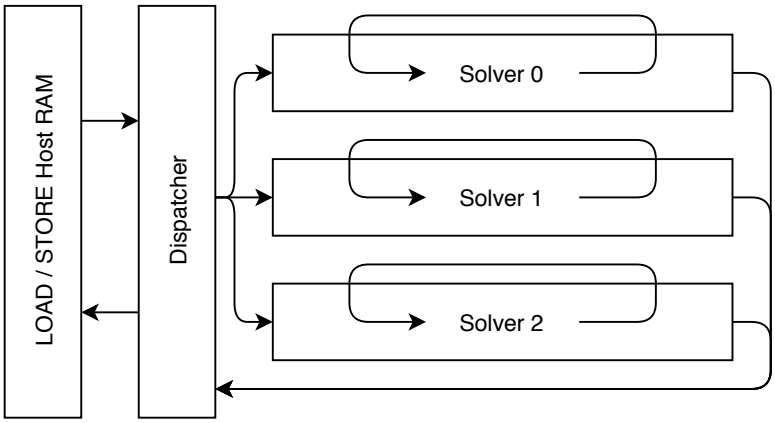

Abb. 4.3.: Aufbau einer vollständigen Löserschaltung

4.3 Benutzerschnittstelle

Die im Rahmen dieser Arbeit vorgeschlagene Software hat zum Ziel sich durch eine möglichst einfache Handhabung auszuzeichnen. Die Nutzung soll ohne FPGA- oder Programmierkentnisse möglich sein. Im Folgenden werden alle notwendigen Schritte zur Nutzung dargelegt.

4.3.1 Konfiguration

Die Konfiguration erfolgt in einer oder mehreren Konfigurationsdateien. Aufgrund ihrer einfachen und lesbaren Syntax, wird YAML [@YAM] als Auszeichnungssprache eingesetzt. Durch die Möglichkeit zum Aufteilen der Konfigurationsdateien können einzelne Teile der Konfiguration, wie zum Beispiel die Lösungsmethode, ausgelagert und leichter wiederverwendet werden.

```
1 method:
2   A: [[],
3       [1]]
4   b: [0.5, 0.5]
5   c: [0, 1]
```

Listing 4.1: Konfiguration der Heun Lösungsmethode

Die Lösungsmethode wird unter dem Schlüssel (engl. *Key*) *method* konfiguriert. Dafür müssen die charakteristischen Koeffizienten A, b und c des Runge-Kutta-Tableaus angegeben werden. Wie im Abschnitt Zielsetzung beschrieben, wird sich im Rahmen der Arbeit aus zeitlichen Gründen auf explizite Runge-Kutta-Verfahren beschränkt. In

Listing 4.1 wird beispielhaft die Konfiguration der Heun Lösungsmethode dargestellt. Dabei wird folgendes Runge-Kutta-Tableau angegeben:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b^T} \end{array} = \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}$$

Bei der Konfiguration des Anfangswertproblems (Key: *problem*) ist nur die Angabe des Differentialgleichungssystems in einem Array unter dem Key *components* notwendig. Dabei können die Variablen x und y (Liste aller y-Werte des Systems) verwendet werden. Die Startwerte für x und y sowie die Schrittweite h und Schrittanzahl n können optional angegeben werden und dienen dann als Standardwerte für die *Runtime*-Komponente. In Listing 4.2 wurde folgendes Differentialgleichungssystems (Lotka-Volterra Gleichungen A.1.2) angegeben:

$$\frac{dy_0}{dt} = f(y_0, y_1, t)$$

$$\frac{dy_1}{dt} = g(y_0, y_1, t)$$

$$f(y_0, y_1, t) = 0.1 \cdot y_0 - 0.2 \cdot y_0 \cdot y_1$$

$$g(y_0, y_1, t) = -0.2 \cdot y_0 + 0.4 \cdot y_0 \cdot y_1$$

```
problem:
    components:
      - 0.1 * y[0] - 0.2 * y[0] * y[1]
      - -0.2 * y[1] + 0.4 * y[0] * y[1]
```

Listing 4.2: Konfiguration des Anfangswertproblems

Zusätzlich kann durch weitere Optionen Einfluss auf den generierten Löser genommen werden. Durch den Schlüssel *nbr_solver* kann spezifiziert werden, wie viele parallele Lösungseinheiten generiert werden sollen. Unter dem Schlüssel *numeric* kann die verwendete Zahlenrepräsentation konfiguriert werden. In Listing 4.3 sind die Standardwerte der Optionen dargestellt, welche verwendet werden, falls keine anderen Werte angegeben werden.

```
nbr_solver: 1

numeric:
    type: 'fixed'  # Currently nothing else supported
    fixed_point_signed: True
    fixed_point_fraction_size: 41
```

```
7       fixed_point_nonfraction_size : 12
```

Listing 4.3: Standardwerte weiterer Konfigurationsoptionen

Eine vollständige Liste aller Konfigurationsparameter wird im Anhang unter A.3 aufgeführt.

4.3.2 Generierung

Zur Interaktion mit dem *Logikgenerator* und der *Runtime* steht ein Kommandozeilentool zur Verfügung. Für die Erstellung eines Lösers muss das *build* Kommando mit den Konfigurationsdateien als Parametern aufgerufen werden. In Listing 4.4 ist der Aufruf zum Erzeugen eines Lösers für das Lotka-Volterra (Predator-Prey) Anfangswertproblem dargestellt. Dabei wird angenommen, dass die beiden Konfigurationsdateien den Inhalt der Listings 4.1 und 4.2 enthalten.

```
>> rtlode build heun.yaml predator-prey.yaml
```

Listing 4.4: Generierung eines Solvers

Der Aufruf erzeugt die Logikschlatung für den Löser auf dem RTL und stößt anschließend Synthese und Implementierung an. Das Ergebnis, der fertige Bitstream für den Löser sowie weitere Informationen, wird in einer .slv-Datei gespeichert.

4.3.3 Ausführung

Das selbe Kommandozeilentool kann auch für die Ausführung eines Lösers genutzt werden. Dafür wird nur die .slv-Datei benötigt, sie enthält alle notwendigen Informationen. Somit kann Generierung und Ausführung auch auf unterschiedlichen Systemen stattfinden. In Listing 4.5 ist beispielhaft das Lösen eines Anfangswertproblems dargestellt.

```
>> rtlode run heun_predator-prey.slv \
              --runtime_config='{x: 0, y: [0, 2], n: 60, h: 0.17}'
```

Listing 4.5: Ausführen eines Solvers

Für komplexere Anwendungen wird eine *Python*-Programmierschnittstelle zur Verfügung gestellt. Als weiteres Unterkommando steht auch *benchmark* bereit, welches eine Performance-Messung für einen Löser vornimmt. Dabei wird, neben der benötigten Zeit zum Lösen eines Anfangswertproblems, auch das parallele Lösen mehrerer Probleme evaluiert.

5 Implementierung

Im folgenden Kapitel wird zunächst ein Überlick über Implementierung im Allgemeinen gegeben. Anschließend werden einzelne, für die Thematik besonders relevante, Details genauer beleuchtet. Handelt es sich dabei um Performanceverbesserungen, werden die Unterschiede analysiert. Eine übergreifende Evaluierung findet in Kapitel 6 statt.

5.1 Verwendete Hardware

Für diese Arbeit wurde eine *Intel Programmable Acceleration Card* [@Intd] verwendet. Diese PCIe-Erweiterungskarte enthält einen *Intel Arria 10 GX 1150* FPGA sowie Arbeitsspeicher und zusätzliche Schnittstellen. Der FPGA beinhaltet $1,15$ Millionen logische Blöcke, $65\ Mb$ internen RAM sowie 3036 DSP-Blöcke [Intb]. Ein DSP-Block kann mit $27\ bit$ Festkomma- und Gleitkommazahlen einfacher Präzision verwendet werden. Durch das Zusammenschalten mehrerer Blöcke kann mit Gleitkommazahlen doppelter Präzision sowie mit genaueren Festkommazahlen gerechnet werden. Im Allgemeinen dient die Erweiterungskarte als Rechenbeschleuniger in Servern, insbesondere im Bereich des High Performance Computings (HPCs).

5.2 Verwendete Software

Für seine Beschleunigerkarten bietet *Intel* mit der *Open Programmable Acceleration Engine (OPAE)* ein umfangreiches Framework an. Neben Tools zur Entwicklung sowie Linuxtreibern, um auf die angeschlossenen FPGAs zuzugreifen, und Bibliotheken hat *Intel* auch Schnittstellen zur Kommunikation zwischen FPGA und CPU definiert. Für diese Schnittstellen werden neben einer Softwarebibliothek in *C* und *Python* auch die notwendigen Schaltungen auf Seiten der Hardware zur Verfügung gestellt. Eine Schaltung, die für eine Beschleunigerkarte entwickelt wurde und die geforderten Schnittstellen implementiert, wird Accelerator Function Unit (AFU) genannt. Die

verwendeten FPGAs unterstützen eine partielle Rekonfiguration zur Laufzeit. Dadurch kann ein Programm auf der CPU flexibel neue AFUs zur Verwendung auf den FPGA laden. Zur Synthese sowie Implementierung der Schaltungen greift die von *Intel* entwickelte OPAE auf eine Installation von *Quartus Prime* [@Inta] zurück. Bei *Quartus Prime* handelt es sich um die Entwicklungssoftware für programmierbare Hardware von *Intel*.

Als Hardwarebeschreibungssprache wird in dieser Arbeit MyHDL [@MyH] verwendet. Die Open-Source-Bibliothek ermöglicht Hardwarebeschreibung und Verifikation auf dem RTL in *Python*. *Python* ist für eine klare Syntax und eine hohe Entwicklungsgeschwindigkeit bekannt. Durch die weite Verbreitung der Sprache kann auf zahlreiche Tools und Bibliotheken zurückgegriffen werden. Dies ermöglicht eine zeitgemäße, software-ähnliche, Vorgehensweise bei der Schaltungsentwicklung. So können leistungstarke Entwicklungsumgebungen genutzt und, für die in MyHDL entwickelte Hardware, Modultests mithilfe des *Python Unit Testing Frameworks* erstellt werden. Für das Verständnis ist an dieser Stelle wichtig klarzustellen, dass hierbei keine HLS zum Einsatz kommt. *Python* selbst ist eine Kontrollflusssprache und eignet sich somit nicht zur Beschreibung von Hardware. Vielmehr bietet MyHDL in *Python* spezielle Sprachkonstrukte an, die sich sehr nah an klassischen Hardwarebeschreibungssprachen wie *Verilog* oder *VHDL* orientieren. Hardware die unter Verwendung dieser MyHDL-Funktionen beschrieben wurde kann autmatisch nach *Verilog* oder *VHDL* transkribiert werden und erlaubt somit eine einfache Integration in den Standard-Workflow der Schaltungsentwicklung. Ein als Teil von MyHDL zur Verfügung stehender Simulator ermöglicht effizientes Testen der beschriebenen Logik. Zusätzlich kann MyHDL gemeinsam mit einem klassischen externen Simulator (z.B. *Icarus Verilog* [@Wil]) für die Verifikation von *Verilog*-Code verwendet werden.

5.3 Framework

5.3.1 Beschreibung von Berechnungen als Datenfluss

Das Framework-Modul bildet die Basis für alle numerischen Berechnungen in den Logikschaltungen. Eine der Hauptaufgaben ist die Bereitstellung einer eingebetteten Domain Specific Language (DSL), welche genutzt werden kann, um die Berechnungen in Form eines Datenflusses zu beschreiben. Dadurch können die numerischen Berechnungen dann vom Framework weiter verarbeitet und automatisiert in Hardware übersetzt werden. Ein Datenfluss lässt sich gut als Graph (Graphentheorie)

modellieren. Dabei sind die Knoten einzelne Operationen und die Pfade stellen die Abhängigkeiten der Operationen untereinander dar. In Abbildung 5.1 ist ein vereinfachtes Klassendiagramm mit den grundlegenden Elementen zur Modellierung der Datenflüsse in dieser Arbeit dargestellt. Die Klasse *Node* repräsentiert dabei eine Operation und somit einen Knoten im Graphen. Die dazugehörige Hardwarebeschreibung der Logik wird im Feld *logic* gespeichert. Als Eingabe für eine Operation können Objekte der Klassen *PipeConstant* und *PipeSignal* verwendet werden. Die Klasse *PipeSignal* verknüft ein MyHDL Signal, welches die Hardwarebeschreibung einer Verbindung ist, mit einer *Node* (Feld: *producer*), die dieses Signal ansteuert. Eine *Node* kann mehrere Signale als Ausgabe ansteuern und somit für mehrere *PipeSignals* im Feld *producer* gelistet sein. Die Operationen, von denen eine *Node* abhängt, ist somit die Menge aller in ihren Eingaben des Types *PipeSignal* im Feld *producer* gelisteten Knoten.

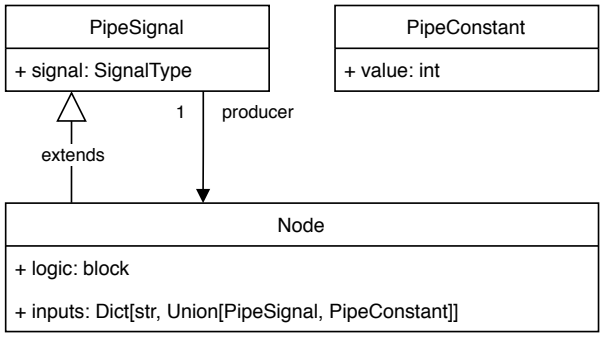

Abb. 5.1.: Vereinfachtes Klassendiagramm der grundlegenden Bauelemente für die Datenflussgraphen

In Listing 5.1 ist die Beschreibung eines einfachen Datenflusses mithilfe der DSL dargestellt. Die auftretenden Variablen sind vom Type *PipeSignal*. Das Framework stellt grundlegende mathematische Funktionen zur Verfügung. Durch eine Überladung der Operatoren können diese neben einem expliziten Aufruf (Zeile 2 und 5) auch implizit verwendet werden (Zeile 3 und 4). Zur automatischen Wandlung der Konstanten in die richtige Darstellungsform, z.B. Festkommazahlen unterschiedlicher Genauigkeit, und zur Vermeidung von Fehlern wird die Verwendung der Klasse *PipeConstant* erzwungen. Entscheidend ist dabei hervorzuheben, dass der dargestellte Code nicht die Berechnungen durchführt, sondern diese nur beschreibt.

```
1 def calc(a, b):
2     add1 = add(a, PipeConstant.from_float(3))
3     add2 = add1 + b
4     mul1 = add1 * PipeConstant.from_float(5)
5     mul2 = mul(add2, a)
6     return (mul1, mul2)
```

Listing 5.1: Definition einer Berechnung als Datenfluss

Zusätzlich zu den grundlegenden mathematischen Funktionen stellt das Framework spezielle *Nodes* als Schnittstelle zwischen klassischer Logik und den durch die DSL beschriebenen Berechnungen zur Verfügung. In Listing 5.2 ist die Verwendung dieser Schnittstellenkomponenten (*PipeInput* und *PipeOutput*) zur Einbindung der Berechnungen aus Listing 5.1 dargestellt.

```
1 data_in = PipeInput(in_valid, a=a_in_signal, b=b_in_signal)
2 res = calc(data_in.a, data_in.b)
3 data_out = PipeOutput(out_busy, a=res[0], b=res[1])
```

Listing 5.2: Schnittstelle zwischen DSL und anderer Logik

Der vollständige Datenfluss aus Listing 5.1 in Kombination mit Listing 5.2 kann als Graph, wie in Abbildung 5.2 dargestellt, visualisiert werden. Dadurch sind die Abhängigkeiten der einzelnen Operationen untereinander gut zu erkennen. Bevor die Multiplikation aus Zeile 5 (Listing 5.1) berechnet werden kann, muss die Addition aus Zeile 3 und somit auch die Addition aus Zeile 2 abgeschlossen sein. Die Zeile 3 und 4 aus Listing 5.1 können hingegen gleichzeitig berechnet werden.

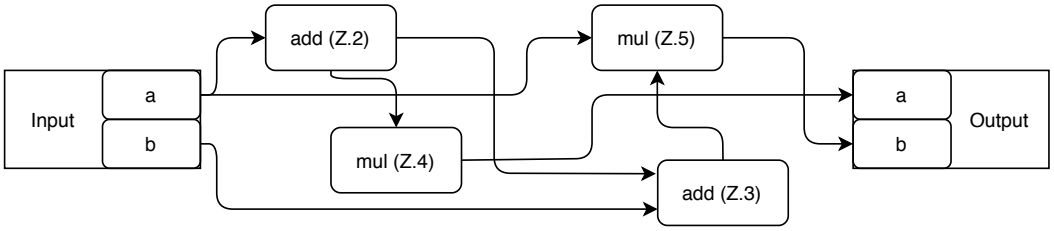

Abb. 5.2.: Darstellung des Datenflusses als Graph

5.3.2 Übersetzung der Berechnungen in Hardware

Um Berechnungen variabler Komplexität automatisiert und ohne Verstöße gegen die zeitlichen Grenzen in Hardware übersetzen zu können, wird die Ausführungszeit der Logik einer *Node* auf eine Taktperiode festgelegt. Komplexere Berechnungen können und sollten daher als eigener Teildatenfluss, unter Verwendung der durch das Framework zur Verfügung gestellten Grundfunktionen, implementiert werden. Ein naiver Ansatz zur Konvertierung des Datenflusses (Lst. 5.1 und 5.2) in Hardware ist die Formulierung der Bedingung mit der Ausführung einzelner Operationen zu warten, bis alle vorherigen Operationen abgeschlossen sind. Um dies zu realisieren, könnte man für jede *Node* ein boolesches Signal einführen, welches das Ende ihrer Berechnung signalisiert. Eine *Node* überwacht dieses Signal für alle ihre Eingänge und kann dadurch mit dem Start der eigenen Berechnungen warten bis alle vorhergehenden abgeschlossen sind. Die für die Berechnung der Ergebnisse für ein Set an

Eingaben benötigte Anzahl Taktzyklen wird durch den längsten Pfad bestimmt. In der gegebenen Beispiel-Berechnung sind das drei Taktzyklen. Dabei ist festzustellen, dass bei dieser Art der Implementierung die Logikverschaltungen der einzelnen Operationen nur in einem Taktzyklus genutzt werden und die restliche Zeit brach liegen. Die Zeit, in der auf die vorhergehenden Berechnungen gewartet wird, sowie alle Taktzyklen nach Ende der Berechnung sind ungenutzt.

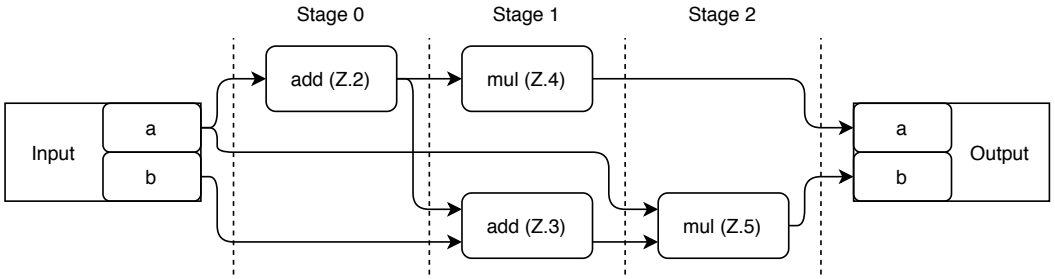

Abb. 5.3.: Vorsortierung der Pipeline des Datenflusses

Der in dieser Arbeit zur Implementierung gewählte Ansatz übersetzt den Datenfluss in Form einer Pipeline. In der so generierten Logik können mehrere Datensätze parallel berechnet werden. Abgesehen von der benötigten Zeit zum Einlaufen, sind dadurch alle Operationen in jedem Taktzyklus aktiv. Die einzelnen *Nodes* werden nach ihren Abhängigkeiten sortiert und der frühestmöglichen Stufe (engl. *stage*), bei der alle Abhängigkeiten bereits in vorherigen Stufen berechnet wurden, zugeordnet. Für den Beispiel-Datenfluss ist dies in Abbildung 5.3 dargestellt. Ein Datensatz durchläuft eine Stufe pro Taktperiode. Signale, die eine oder mehrere Stufen überspringen, müssen deshalb mithilfe von Registern zwischengespeichert werden. Diese Register werden automatisiert in den Datenflussgraphen eingefügt. In Abbildung 5.4 ist der transformierte Graph dargestellt.

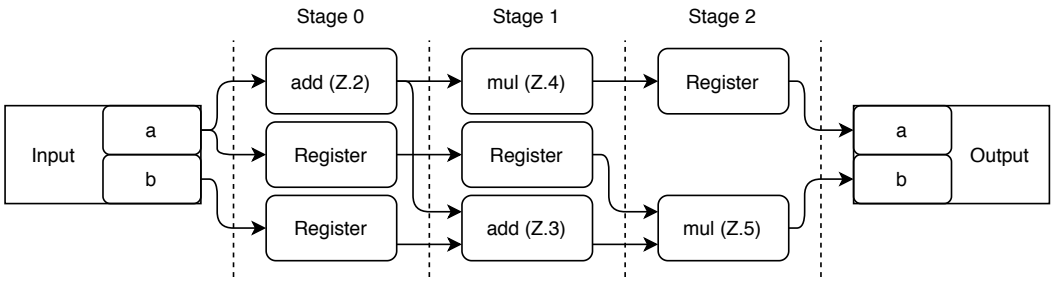

Abb. 5.4.: Pipeline des Datenflusses mit Registern

Bei der Implementierung einer Pipeline ist die Art des Umgangs mit einem Halt der Pipeline wesentlich. Ein solcher kann auftreten, wenn die nachfolgende Logik keine weiteren Daten verarbeiten kann. Auch der Leerlauf einer Pipeline, da zum

Beispiel am Eingang keine neuen Daten zur Verfügung stehen, muss berücksichtigt werden. Der in dieser Arbeit gewählte Ansatz wird *buffered handshake* [@Tec] genannt. Die organisatorische Logik für den Pipelinebetrieb ist in den einzelnen Stufen implementiert. Ob die aktuelle Eingabe gültig (engl. *valid*) ist, teilt die vorherige Stufe mit einem booleschen Signal mit. Ein weiteres boolesches Signal zeigt an, ob die nächste Stufe gerade beschäftigt (engl. *busy*) ist und somit keine Daten empfangen kann. Ein Datentransfer zwischen zwei Stufen findet nur statt, wenn die nächste Stufe nicht beschäftigt ist und die aktuelle Stufe ihre Daten für gültig erklärt. Da bei einem Halt der Pipeline in jedem Taktzyklus nur eine vorhergehende Stufe über das *busy*-Signal benachrichtigt werden kann, müssen alle *Nodes* einer Stufe in der Lage sein einen Satz Eingabewerte, die aktuelle Ausgabe der vorhergehenden Stufe, zwischenzuspeichern. Die Logik einer Stufe weist nach Auswertung der Signale ihre *Nodes* an, ob die neuen Daten verarbeitet und direkt weitergegeben, das Ergebnis zwischengespeichert oder der Speicher geleert werden soll. In Abbildung 5.5 sind die Kontrollsignale zwischen den Stufen der Pipeline sowie den Eingabe- und Ausgabe-*Nodes* abgebildet.

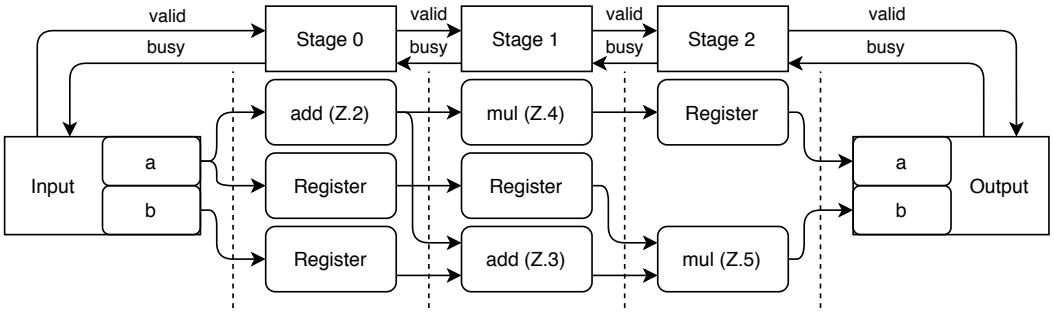

Abb. 5.5.: Fertige Pipeline des Datenflusses

Beim Aufbau der Graphen aus dem Datenfluss werden nur *Nodes*, deren Ergebnisse weiter verwendet werden, platziert. In Tabelle 5.1 sind die benötigten Taktzyklen pro Lösungsschritt der beiden beschriebenen Implementierungsvarianten gegenübergestellt. Durch die Implementierung als Pipeline wird keine Verbesserung der Performance beim Lösen eines einzelnen Anfangswertproblems im Vergleich zur naiven Implementierung erreicht. Betrachtet man jedoch die Performance beim Lösen mehrerer Anfangswertprobleme, sinkt die benötigte Zeit erheblich. Komplexere Lösungsmethoden führen zwar mehr Berechnungen pro Schritt durch, können aber dadurch auch mehr Anfangswertprobleme gleichzeitig lösen. Unabhängig von der verwendeten Lösungsmethode benötigt die Pipeline bei voller Auslastung pro weiterem Anfangswertproblem nur einen zusätzlichen Taktschritt. Daraus resultiert folgender Zusammenhang: Je mehr Anfangswertprobleme gelöst werden, desto

weiter nähern sich die benötigten Takte pro Schritt und Anzahl der Anfangswertprobleme dem Wert 1 an.

Tab. 5.1.: Vergleich der Taktzyklen pro Schritt für die Lotka-Volterra (Predator-Prey) Gleichungen: naive Implementierung vs. Pipeline-Implementierung

	Euler		RK4	
Anzahl Anfagswertprobleme	*1*	*100*	*1*	*100*
Naive Implementierung	6,2	607,4	24,2	2407,7
Pipeline-Implementierung	6,2	108,2	24,2	109,8

Der Bedarf an FPGA-Ressourcen der Pipeline-Implementierung ist durch die zusätzlich benötigten Logikverschaltungen und Register größer als bei der naiven Implementierung. Allerdings unterscheidet sich die Anzahl der benötigten DSP-Blöcke nicht. In der Anwendung hat sich gezeigt, dass bei berechnungsintensiven Schaltungen die Anzahl der DSP-Blöcke der limitierende Faktor sind. Eine optimale Ausnutzung dieser, wie in der vorliegenden Arbeit durch die Pipeline-Implementierung realisiert, ist in Allgemeinen anzustreben.

5.3.3 Optimierung der Operatoren

Da jegliche Berechnungen immer auf den durch das Framework zur Verfügung gestellten Grundoperatoren basieren, ist es wichtig diese zu optimieren. Die Ziele sind dabei die Minimierung der benötigten DSP-Blöcke sowie Performance-Verbesserungen durch Einsparen von Taktzyklen oder Berechnungen. Um kombinatorische Logik, die keinen eigenen Taktzyklus benötigt, effizient in der Pipeline anordnen zu können, wird eine neue Variante einer *Node* eingeführt, die *CombNode*. Die bestehende *Node* wird im Folgenden *SeqNode* genannt. Abhängig von dem Typ und den Werten der Parameter werden unterschiedliche Hardwareschaltungen beschrieben oder Berechnungen ausgeführt. Nachfolgend werden solche Optimierungen am Beispiel der Multiplikation dargelegt. Die beiden Parameter der Multiplikation werden a und b genannt. Wie bei der Implementierung der DSL festgelegt, können diese vom Typ *PipeConstant* oder *PipeSignal* sein. Das Verhalten der Multiplikations-Funktion unterscheidet sich für unterschiedliche Typen- und Wertepaare der Parameter. Untenstehend ist das jeweilige Verhalten anhand von Bedingungen beschrieben. Die erste zutreffende Bedingung ist ausschlaggebend. Um die Bedingungen zu vereinfachen, wird davon ausgegangen, dass Parameter a immer vom Typ *PipeConstant* ist, sobald einer der beiden Parameter vom Typ *PipeConstant* ist.

***a* vom Typ *PipeConstant* und *a.value == 0*:** Unabhängig von b wird beim Aufruf der Multiplikations-Funktion immer eine *PipeConstant* mit dem Wert 0 zurückgegeben.

***a* vom Typ *PipeConstant* und *b* vom Typ *PipeConstant*:** Bei der Multiplikation ist das Ergebnis eine *PipeConstant* mit dem Wert $a \cdot b$.

***a* vom Typ *PipeConstant* und *a.value & (a.value - 1) == 0*:** In *a.value* ist genau ein Bit gesetzt. Somit liegt eine Multiplikation oder Division um ein Vielfaches von 2 vor. Eine arithmetische Verschiebung um n Bits nach links ist äquivalent zu einer Multiplikation mit 2^n. Die Variable n kann auch negativ sein, dann wird eine arithmetische Verschiebung nach rechts ausgeführt und somit dividiert. Die Implementierung eines arithmetischen Shifts um einen konstanten Wert in Hardware ist, ohne Ressourcen zu benötigen, durch eine andere Verschaltung der einzelnen Signale möglich. Im Vergleich zu einer Implementierung mit einem DSP-Block werden sowohl Ressourcen (der DSP-Block) als auch Zeit eingespart. Die Funktion gibt ein *PipeSignal* zurück, welches von einer *CombNode* angesteuert wird.

in allen anderen Fällen: Die Multiplikation wird mithilfe eines DSP-Blockes implementiert. Das Ergebnis des Aufrufes ist ein *PipeSignal*, welches von einer *SeqNode* angesteuert wird.

Tab. 5.2.: Vergleich der generierten Pipelines mit und ohne Optimierung für die Lotka-Volterra (Predator-Prey) Gleichungen

	nbr_stages	reg	add	sub	mul_dsp	mul_by_shift
Euler	8	68	7	1	13	/
Euler opt.	5	39	5	1	8	/
Einsparung	*37,5%*	*42,6%*	*28,6%*	*0,0%*	*38,5%*	
Heun	15	165	12	2	25	/
Heun opt.	11	116	10	2	16	4
Einsparung	*26,7%*	*29,7%*	*16,7%*	*0,0%*	*36,0%*	
RK4	31	519	28	4	56	/
RK4 opt.	23	325	20	4	40	5
Einsparung	*25,8%*	*37,4%*	*28,6%*	*0,0%*	*28,6%*	

In Tabelle 5.2 sind Informationen über die Bestandteile der generierten Pipelines für unterschiedliche Lösungsmethoden, je einmal mit Optimierungen und einmal ohne Optimierungen, gelistet. Zusätzlich ist die prozentuale Einsparung, welche durch die Optimierungen der Grundoperatoren erreicht wird, angegeben. Da jede Stufe einer Pipeline genau einen Taktzyklus lang rechnet, entspricht die Anzahl der Stufen (engl. *nbr_stages*) den benötigten Taktzyklen zum Durchlauf der Pipeline. Eine Einsparung der Stufen, wie zum Beispiel um $37,5\%$ beim Euler-Verfahren führt beim Lösen

eines einzelnen Anfangswertproblems zu einer Performanceverbesserung um den gleichen prozentualen Anteil. Aber auch bei den benötigten DSP-Blöcken können für alle Verfahren signifikante Einsparungen festgestellt werden. Das Wissen, dass sich Multiplikation oder Division mit einem Vielfachen von 2 sehr effizient in Hardware implementieren lässt, kann bei der Auswahl einer Lösungsmethode berücksichtigt werden. Je mehr Koeffizienten des Butcher-Tableaus der Bedingung gerecht werden, desto ressourceneffizienter und performanter lässt sich ein Lösungsverfahren auf einem FPGA implementieren.

5.4 Logikgenerator

5.4.1 Parsen der Differentialgleichungen

Der Logikgenerator arbeitet mit einer Konfiguration, aus der individuelle Löserschaltungen erzeugt werden sollen. Eine zentrale Aufgabe ist dabei das automatisierte Übersetzen der angegebenen Gleichungssysteme in Hardware. In der Konfiguration ist das Differentialgleichungssystem in Textform beschrieben (siehe Listing 4.2). Die rechten Seiten der in expliziter Form angegebenen Gleichungen, werden durch den Logikgenerator in eine Hardwarebeschreibung übersetzt. In Listing 5.3 ist die Funktion angegeben, welche diese Aufgabe erfüllt.

```
1 def expr(expression: str,
2          scope: Dict[str, Union[PipeSignal, List[PipeSignal]]]):
3     tree = get_expr_grammar().parseString(expression, parseAll=True)
4     return _generate_logic(tree[0], scope)
```

Listing 5.3: Implementierung des *Expression Parsers*

Die Funktion `expr()` benötigt zwei Parameter. Als erster Parameter (`expression`) wird der Ausdruck, der geparst werden soll, angegeben. Zusätzlich wird über den Parameter `scope` ein Dictionary, in dem Variablennamen auf *PipeSignals* oder Listen mit *PipeSignals* aufgeführt werden, erwartet. Unter Verwendung der Bibliothek *PyParsing* [@PyP20] wird in der Funktion `get_expr_grammar()` eine Grammatik für die Ausdrücke formuliert. Die Ausdrücke müssen einer Infixnotation folgen und dürfen als Operanden neben Dezimalzahlen auch Variablennamen, optional mit einem Index in eckigen Klammern, enthalten. Mit dem Aufruf der Bibliotheksfunktion `ParserElement.parseString()` wird ein Ableitungsbaum für den übergebenen Ausdruck erzeugt. Die Funktion `_generate_logic()` arbeitet diesen rekursiv ab

und beschreibt nach einer Fallunterscheidung die einzelnen Berechnungen mithilfe der durch das Framework zur Verfügung gestellen DSL. Bei Verwendung eines Variablennamen im Ausdruck wird das entsprechende Objekt aus dem Dictionary `scope` verwendet. Ist ein Index in eckigen Klammern angegeben, wird auf das entsprechende Element in der Liste zugegriffen.

5.4.2 Implementierung der Runge-Kutta-Verfahren

Die als Teil des Frameworks entwickelte Datenfluss-DSL ermöglicht eine sehr direkte Implementierung der für die Runge-Kutta-Verfahren notwendigen Berechnungen. Die allgemeine Vorgehensweise wird im Folgenden am Beispiel der Berechnungen für die Stufenvektoren verdeutlicht. Wie bereits im Abschnitt 2.3.3 der Grundlagen zu den Runge-Kutta-Verfahren dargelegt, dient die Gleichung 5.1 zur Berechnung der Stufenvektoren $v_1, \ldots, v_s \in \mathbb{R}^n$ eines s-stufigen Runge-Kutta-Verfahrens.

$$v_j = f\left(x_k + h \cdot c_j, \eta_k + h \cdot \sum_{l=1}^{s} a_{jl} v_l\right) \quad \text{für } j = 1, \ldots, s \tag{5.1}$$

Die in Listing 5.4 dargestellte Funktion `stage()` dient der Berechnung eines Stufenvektors. Ihr wird über den Parameter `config` eine zum Zeitpunkt der Generierung festgelegte Stufenkonfiguration übergeben. Diese enthält neben dem Index j der Stufe auch die Koeffizienten a_{jl} für $l = 1, \ldots, s$ und c_j sowie die rechten Seiten (`config.components`) und die Größe des Gleichungssystems n. Zusätzlich erhält die Funktion über die Parameter `h` und `x` je ein Signal und über `y` eine Liste mit Signalen der Größe n. Die Signale der vorhergehenden Stufenvektoren werden in einer verschachtelten Liste über den Parameter `v` zur Verfügung gestellt. In Zeile 2 wird die Berechnung des x-Wertes für die Funktionsauswertung definiert. Dabei muss nur auf die Umwandlung der Konstante in den richtigen Datentyp geachtet werden. Ab Zeile 3 werden die Berechnungen des y-Vektors beschrieben. Die Funktion `dot_product()`, welche über die vom Framework zur Verfügung gestellten Basisoperationen implementiert ist, berechnet das Skalarprodukt der beiden ihr übergebenen Vektoren. In Zeile 8 findet die eigentliche Funktionsauswertung statt. Dabei wird die bereits vorgestellte Funktion `expr_parser.expr()` für jede rechte Seite des expliziten Gleichungssystems aufgerufen. Das *PipeSignal* des x-Wertes sowie eine Liste an *PipeSignals*, der y-Vektor, wird als Mapping-Dictionary übergeben. Die Umsetzung der restlichen Berechnungen der Runge-Kutta-Verfahren erfolgt analog.

```
def stage(config, h, x, y, v):
    rhs_x = x + h * PipeConstant.from_float(config.c)
    rhs_y = [y[i] + h * dot_product(
            [PipeConstant.from_float(el) for el in config.a],
            [el[i] for el in v[:config.stage_index]]
        ) for i in range(config.system_size)]

    return [expr_parser.expr(rhs_expr, {
        'x': rhs_x,
        'y': rhs_y
    }) for rhs_expr in config.components]
```

Listing 5.4: Implementierung einer Stufe der Runge-Kutta-Verfahren

5.4.3 .slv Dateiformat

Der Logikgenerator erzeugt Verschaltungen für den FPGA, welche dann nach Synthese und Implementierung in Form eines Bitstreams vorliegen. Die Tools der OPAE speichern diesen, zusammen mit für den Konfigurationsprozess benötigten Informationen, in einer .gbs-Datei. Für die spätere Nutzung der so generierten AFU durch die in dieser Arbeit zur Verfügung gestellte Runtime-Software werden weitere Angaben benötigt. Sowohl die verwendete Zahlendarstellung als auch die Größe des Systems haben einen Einfluss auf das Interface. Um eine möglichst einfache Handhabung und das Vorgenerieren von Lösern mit geringerem Verwaltungsaufwand zu ermöglichen, wird für diese Arbeit ein neues Dateiformat entwickelt, welches die für die Runtime benötigten Informationen mit der .gbs-Datei kombiniert. Der Aufbau ist in Abbildung 5.6 skizziert. Nach einer statischen ID, welche das Dateiformat identifiziert, wird die Länge des Konfigurationsheaders in vier Byte gespeichert. Darauf folgt der in JSON codierte Header, mit allen später benötigten zusätzlichen Informationen. Im Anschluss wird die ursprüngliche .gbs-Datei eingebettet. Eine Bibliothek zum einfachen Packen und Entpacken der .slv-Dateien wurde implementiert.

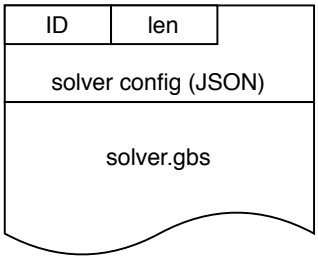

Abb. 5.6.: Aufbau des .slv-Dateiformates

5.4.4 Interface zwischen CPU und AFU

Die OPAE stellt das Core Cache Interface (CCI) zur Kommunikation zwischen der CPU und einer AFU auf dem FPGA zur Verfügung. Gründsätzlich bieten sich zwei Möglichkeiten zum Datenaustausch. Die Control / Status Registers (CSRs) sind ein Adressraum, für den die CPU Schreib- und Leseanfragen stellen kann. Die AFU ist nur ein reaktiver Teilnehmer. Die genaue Implementierung auf Seiten der AFU ist dem Entwickler überlassen. Die zweite verfügbare Variante des Datenaustausch ist das Lesen und Schreiben aus und in den RAM der CPU. Hierbei kann die AFU die Übertragung von bis zu vier Cache-Lines an oder von einer RAM-Adresse gleichzeitig anstoßen. Eine Cache-Line ist bei aktuellen Prozessoren 512 Bits $= 64$ Bytes groß. Da die AFU nicht ohne weiteres auf die virtuelle Speicherverwaltung der CPU zugreifen kann, benötigt sie für alle Anfragen die physische Adresse. Damit ein allozierter Speicherbereich an der gleichen physischen Adresse verbleibt, muss die entsprechende RAM-Page von der Runtime-Software angepinnt werden. Die Größe eines dem FPGA zugänglichen Speicherbereiches ist somit durch die Page-Größe (Standard: 4096 Bytes) limitiert. Sollen größere Speicherbereiche zugänglich gemacht werden, ist die Verwendung der sogenannten Hugepages notwendig [@Lin].

Den in dieser Arbeit generierten Lösern müssen die zu berechnenden Anfangswertprobleme übermittelt werden. Zusätzlich muss die AFU die Ergebnisse am Ende der Berechnung an die CPU zurückübertragen. Zur Realisierung dieses Datenaustausches werden zwei Speicherbereiche im RAM der CPU verwendet. Einer wird von der Runtime mit den Anfangswerten beschrieben und dient somit als Eingabe für die AFU. Diese ruft immer vier Cache-Lines, im Folgenden ein Chunk genannt, gemeinsam ab. Der zweite Speicherbereich wird für die Ausgabe der Ergebnisse genutzt. Für die physikalischen Adressen der beiden Bereiche stellt die AFU zwei Control-Register an fest definierten Adressen zur Verfügung. Zusätzlich werden in weiteren Registern die Anzahl der zu lesenden Chunks festgelegt sowie der Lösungsprozess gestartet und dessen Status übermittelt.

```
1 class InputData(StructDescription, metaclass=...):
2     id = BitVector(num.INTEGER_SIZE)
3     x_start = BitVector(num.TOTAL_SIZE)
4     y_start = List(system_size, BitVector(num.TOTAL_SIZE))
5     h = BitVector(num.TOTAL_SIZE)
6     n = BitVector(num.INTEGER_SIZE)
7     _bit_padding = BitVector(len_padding)
```

Listing 5.5: Beschreibung des Datenformats der Eingabedaten (gekürzt)

Das Datenformat der Eingabedaten ist in Listing 5.5 definiert. Mithilfe der im Rahmen der Arbeit entwickelten `StructDescription`-Bibliothek können bitgenau gepackte Datentypen beschrieben werden. Für die Anfangswertprobleme wird neben der Angabe der Startwerte für x und y sowie der Schrittweite h und der Anzahl an Schritten n auch eine ID festgelegt. Über diese lässt sich später ein Ergebnis-Datensatz einem bestimmten Anfangswertproblem zuordnen. Eine gleichbleibende Reihenfolge zwischen Eingabe- und Ausgabedaten ist nicht gewährleistet. Mithilfe eines optionalen Paddings wird die Gesamtlänge eines Eingabe-Datensatzes auf die nächste Bytegrenze erweitert. Dies vereinfacht die Handhabung auf Seiten der CPU.

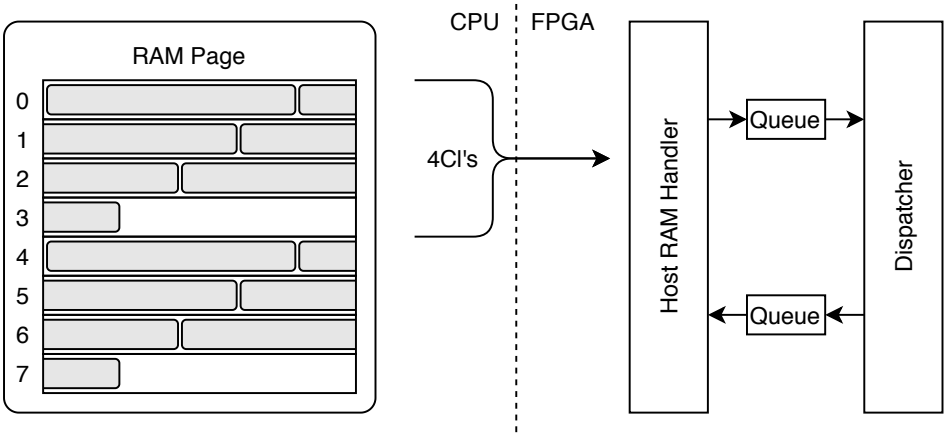

Abb. 5.7.: Laden der Anfangswertprobleme

Abbildung 5.7 visualisiert das Laden der Anfangswertprobleme. Die Datensätze werden im Eingabe-Speicherbereich von der Runtime so abgelegt, dass keine Chunkgrenzen überschritten werden. Dies vereinfacht die Logik auf Seiten des FPGAs, da die Positionen der einzelnen Datensätze somit relativ zum Chunk konstant bleiben. Am Ende eines Chunks bleiben demnach Teile des Speicherbereiches ungenutzt. Dies ist in Abbildung 5.7 in der 3. und 7. Cache-Line zu sehen. In den generierten Lösern auf dem FPGA ist der Host-RAM-Handler für das Management der Hauptspeicherzugriffe zuständig. Dieses Modul lädt, sobald der Löseprozess gestartet wurde, nacheinander je einen Chunk an Daten aus dem Eingabe-Speicherbereich. Sobald alle vier Cache-Lines eingetroffen sind, werden die einzelnen Datensätze separiert und in einer Warteschlange (engl. *Queue*) abgelegt. Diese dient der Entkopplung der Löserlogik von den Speicherzugriffen. Das Dispatcher-Modul liest dann die Datensätze aus der Queue und verteilt sie an die einzelnen Lösungseinheiten. Das Speichern der Ergebnisse funktioniert analog. Sobald alle Eingabedatensätze verarbeitet und die Ergebnisse an die CPU übertragen wurden, signalisiert die AFU dies mit einer Änderung des Wertes im Status-Register.

6 Evaluierung

Im Rahmen dieser Arbeit wird ein alternatives Konzept zum effizienten Lösen von Differentialgleichungen auf FPGAs vorgestellt. In der Arbeit von Stamoulias et al. [Sta+17] aus dem Jahr 2017 wurde die Wahl der Nutzung von HLS-Tools mit einer reduzierten Entwicklungszeit begründet. Mit dem neu vorgeschlagenen Konzept in der hier vorliegenden Arbeit wird der spezifische Entwicklungsaufwand auf das Beschreiben des Lösers in einer Konfigurationsdatei gesenkt. Aus dieser kann der entwickelte *Logikgenerator* automatisiert eine Schaltung erzeugen. Ein direkter Vergleich mit [Sta+17] ist wegen dem nicht zugänglichem Quelltext und einer anderen vorliegenden Hardware leider nicht möglich. Jedoch sind die Angaben zur Laufzeit und der verwendeten Hardware ausreichend genau, um einen Vergleich mit einigen Einschränkungen durchführen zu können. Die Angaben im Folgenden beziehen sich, falls nicht anders angegeben, auf das in [Sta+17] verwendete Differentialgleichungssystem, die Lotka-Volterra Gleichungen (A.1.2). In Tabelle 6.1 sind die wichtigsten Ressourcen der beiden verwendeten FPGAs gegenübergestellt. Der in [Sta+17] genutzte *Xilinx Kintex UltraScale KU060* besitzt, im Vergleich zu dem in dieser Arbeit verwendeten *Intel Arria 10 GX 1150*, neben einer geringeren Anzahl an logischen Blöcken und einem kleineren internen RAM, auch weniger der wichtigen DSP-Blöcke.

Tab. 6.1.: Vergleich der verwendeten FPGAs - Daten-Quelle: [Xil20]

	DSP-Blöcke	Logische Blöcke	Interner RAM
Xilinx Kintex UltraScale KU060	2760	725550	9,1 Mb
Intel Arria 10 GX 1150	3036	1150000	65 Mb

Um die Hardwareunterschiede auszugleichen, soll die Ressourcennutzung der in dieser Arbeit generierten Schaltungen zum Zweck der Vergleichsbarkeit begrenzt werden. Eine Auslastung aller FPGA-Ressourcen ist in der Praxis im Allgemeinen nicht möglich, da die Komplexität der Implementierung erheblich ansteigt und als Folge dessen das Einhalten der definierten Zeitschranken meist nicht möglich ist. Aus diesem Grund wird bei allen folgenden Vergleichen die maximale Anzahl der parallelen Lösereinheiten so beschränkt, dass weniger als 75% der 2760 auf dem *Xilinx Kintex UltraScale* zur Verfügung stehenden DSP-Blöcke benötigt werden. Dadurch wird kein Vorteil aus dem größeren FPGA geschöpft. Stamoulias et al. [Sta+17]

verwenden in ihrer Arbeit Gleitkommazahlendarstellungen einfacher und doppelter Präzision. Der *Logikgenerator* unterstützt zur Zeit nur Festkommazahlendarstellungen. Um trotzdem einen Vergleich in dieser Arbeit ermöglichen zu können, wurden die Vor- und Nachkommastellen so gewählt, dass bei einer Multiplikation die gleiche Anzahl an DSP-Blöcken, wie für eine Gleitkommadarstellung doppelter Präzision, benötigt wird. Zeitlich macht die Nutzung einer Gleitkommazahlendarstellung keinen Unterschied. In Tabelle 6.2 sind die benötigten DSP-Blöcke pro Lösereinheit für unterschiedliche Lösungsmethoden dargestellt.

Tab. 6.2.: Vergleich benötigter DSP-Blöcke pro Lösereinheit

	Multiplikationen	DSP-Blöcke	Max. Anzahl parallel
Euler	8	64	32,3
Heun	16	128	16,2
SSPRK3	30	240	8,6

Mit steigender Komplexität des Lösungsverfahrens und somit höherem Ressourcenbedarf pro Lösereinheit, sinkt auch die maximale Anzahl verwendbarer paralleler Lösereinheiten, um die 75%-Grenze einzuhalten. Dabei sollte jedoch beachtet werden, dass jede Lösereinheit intern eine Pipeline zur Berechnung nutzt und somit zusätzlich parallel mehrere Anfangswertprobleme lösen kann. In Tabelle 6.3 ist die benötigte Zeit zum Lösen von 10k Anfangswertproblemen für unterschiedliche generierte Schaltungen den Ergebnissen des HLS-Ansatzes aus [Sta+17] (einfache (SFP) und doppelte (DFP) Präzision der Gleitkommazahlen) gegenüber gestellt. Die Schrittweite und Anzahl von Schritten für jede Methode orientiert sich dabei an den Angaben aus [Sta+17] und ist so abgestimmt, dass eine vergleichbare Genaugigkeit bei den Ergebnissen erreicht wird.

Tab. 6.3.: Benötigte Zeit in [ms] zum Lösen von 10k Anfangswertproblemen mit folgender Anzahl an Schritten: Euler 10k, Heun 60 und SSPRK3 12

	Anzahl paralleler Lösereinheiten						[Sta+17]	
	1	2	4	8	16	32	SFP	DFP
Euler	758,18	379,10	192,49	96,20	48,89	26,80	128	676
Heun	4,61	2,33	1,50	1,19	1,19	/	3,79	6,63
SSPRK3	1,20	1,20	1,21	1,20	/	/	2,22	94,79

Beim Euler-Verfahren ist zu erkennen, dass eine Steigerung der Anzahl paralleler Lösereinheiten die benötigte Berechnungszeit stetig reduziert. Bei 32 parallelen Lösereinheiten, der festgelegten Grenze um die Hardwareunterschiede auszugleichen, wird eine um das ≈ 25-fache schnellere Berechnung erreicht im Vergleich zu dem Löser mit doppelter Gleitkommagenauigkeit aus [Sta+17]. Selbst der Löser einfacher Präzision ist um das $\approx 4,7$-fache langsamer. Auch unter Verwendung der

anderen Lösungsmethoden wird eine schnellere Berechnung als in [Sta+17] erzielt. Jedoch ist zu erkennen, dass eine Steigerung der Anzahl paralleler Lösereinheiten sowohl bei der Heun- als auch bei der SSPRK3-Methode nicht in eine geringere Berechnungszeit resultiert. Der Wert von $\approx 1,2\ ms$ wird nicht unterschritten. Der gering gewählte Endwert des Anfangswertproblems führt in Kombination mit einer im Vergleich zum Euler-Verfahren großen Schrittweite, bedingt durch die besseren numerischen Eigenschaften der mehrstufigen Verfahren, zu einer sehr geringen Schrittanzahl (Heun: 60, SSPRK3: 12). Dadurch ist der limitierende Faktor nicht mehr die Berechnung, sondern das Interface zwischen CPU und FPGA, welches nicht in der Lage ist die Daten der 10k Anfangswertprobleme ausreichend schnell zu übertragen. Um die Performance der Berechnungen in den generierten Logikschaltungen betrachten zu können, wurden erneut 10k Anfangswertprobleme mit den unterschiedlichen Verfahren gelöst. Die Anzahl an Schritten ist jedoch nun kostant für alle Verfahren bei 10k und entspricht somit der des Euler-Verfahrens aus [Sta+17] und Tabelle 6.3. In Tabelle 6.4 sind die Ergebnisse dieser Betrachtung dargestellt. Die Zeiten des Heun- und SSPRK3-Verfahrens für die Schaltungen des HLS-Ansatzes aus [Sta+17] wurden aus der benötigten Zeit pro Schritt erechnet.

Tab. 6.4.: Benötigte Zeit in [ms] zum Lösen von 10k Anfangswertproblemen mit je 10k Schritten

	Anzahl paralleler Lösereinheiten						[Sta+17]	
	1	2	4	8	16	32	SFP	DFP
Euler	758,18	379,10	192,49	96,20	48,89	26,80	128	676
Heun	763,74	382,46	189,80	97,71	50,87	/	632	1105
SSPRK3	763,74	391,78	202,42	102,05	/	/	1850	78992

Wie erwartet führt eine Erhöhung der Anzahl paralleler Lösereinheiten nun auch bei der Heun- und SSPRK3-Methode zu einer Verbesserung der Performance. Betrachtet man die schnellsten generierten Schaltungen im Vergleich mit den Ergebnissen aus [Sta+17] ist festzustellen, dass der Faktor zwischen der benötigten Zeit für Lösungsmethoden steigender Komplexität in dieser Arbeit deutlich geringer ausfällt. Der Geschwindigkeitsvorteil ist bei komplexeren Verfahren somit deutlich höher. Die Schaltung zur SSPRK3-Methode für Gleitkommazahlen doppelter Präzision aus [Sta+17] zeigt eine schlechte Performance und ist damit ein deutlicher Ausreißer. In Tabelle 6.5 ist der Faktor der Beschleunigung beim Lösen der Anfangswertprobleme im Vergleich zu dem HLS-Ansatz aus [Sta+17] dargestellt. Wie weiter oben beschrieben, wurden die Parameter der Festkommazahlendarstellung so gewählt, dass sie mit der Gleitkommadarstellung doppelter Präzision vergleichbar ist. Reduziert man die Genauigkeit der Festkommazahlendarstellung oder führt eine Gleitkommazahlendarstellung einfacher Präzision ein, können mehr Lösereinheiten parallel

eingesetzt werden und somit der Löseprozess weiter beschleunigt werden. Vor allem bei komplexeren Runge-Kutta-Verfahren kann der in dieser Arbeit vorgestellte *Logikgenerator* einen deutlichen Performancevorteil gegenüber dem HLS-Ansatz erreichen. Im direkten Vergleich sind die generierten Schaltungen dieser Arbeit somit je nach Lösungsmethode um das $21,7$- bis 774-fache schneller. Selbst im Vergleich zu einer weniger präzisen Zahlendarstellung kann eine um das $4,8$- bis $18,1$-fache schnellere Berechnung erreicht werden.

Tab. 6.5.: Erreichte Beschleunigung des Lösungsverfahrens im Vergleich zu [Sta+17]

	SFP	DFP
Euler	4,78	25,22
Heun	12,42	21,72
SSPRK3	18,13	774,05

7 Zusammenfassung

In dieser Arbeit wurde eine neuartige Herangehensweise zum Lösen von Differentialgleichungen auf FPGAs vorgestellt. Durch den hier entwickelten *Logikgenerator* können Schaltungen automatisiert erzeugt werden. Der Entwicklungsaufwand muss dabei nur einmalig geleistet werden. Durch eine einfache Benutzerschnittstelle können Schaltungen für andere Differentialgleichungssysteme oder Lösungsmethoden ohne FPGA- und programmierspezifische Kenntnisse erzeugt werden. Dadurch wird nicht nur der bisher notwendige spezifische Entwicklungsaufwand je Lösungsansatz (Differentialgleichungssystem und Lösungsmethode) eingespart, sondern auch der mögliche Nutzerkreis deutlich erweitert. Die Logik der Löser wird als Datenfluss beschrieben. Durch diesen, im Vergleich zur HLS, hardwarenäheren Ansatz ist eine performante Implementierung und anschließende Optimierung möglich. Wie die Evaluierung zeigt, sind diese automatisiert erzeugten Schaltungen beim Lösen der Anfangswertprobleme um das $21,7$- bis 774-fache schneller als vergleichbare Schaltungen, die mit für FPGAs handoptimierten Algorithmen unter Verwendung von HLS-Tools [Sta+17] erzeugt wurden. Mit steigender Stufenzahl des verwendeten Runge-Kutta-Verfahrens und damit einhergehender steigender Komplexität der notwendigen Berechnungen, steigt der Geschwindigkeitsvorteil der durch den hier vorgestellten *Logikgenerator* erzeugten Schaltungen. Zusätzlich wurde in einem ausführlichen Grundlagenteil ein allgemeiner Einstieg in die Schaltungsentwicklung für FPGAs unter unterschiedlichen Gesichtspunkten vermittelt. Die zu Beginn der Arbeit gesetzten Ziele konnten somit erreicht werden.

Ist das Ziel eine einzelne performante Implementierung einer Berechnung zu erzeugen (zum Beispiel bei der Entwicklung eines ASICs) kann sich eine manuelle Optimierung auf dem RTL lohnen. Zur Einsparung von Entwicklungsaufwand für einzelne Schaltungen jeglicher Form können HLS-Tools verwendet werden. Für repetitive Aufgaben wird jedoch die Vorgehensweise dieser Arbeit, auf Basis einfacher Konfigurationsdateien Schaltungen automatisiert zu erzeugen, empfohlen. Der notwendige Entwicklungsaufwand muss nur einmalig geleistet werden und die generierten Schaltungen sind im Vergleich zu dem HLS-Ansatz performanter.

Ausblick

Ausgehend von dem hier entwickelten *Logikgenerator* sind weitere Anpassungen und Erweiterungen denkbar. Mithilfe einer Unterstützung für Gleitkommazahlen kann die Benutzerfreundlichkeit weiter erhöht werden, da weniger Vorabüberlegungen zur Dimensionierung der Datentypen anfallen. Als Alternative kann eine dynamische Anpassung der Genauigkeit der verwendeten Festkommazahlentypen zu einer Ressourceneinsparung führen. Auch die Unterstützung weiterer Lösungsverfahren, wie zum Beispiel implizite Runge-Kutta-Verfahren und *Parallel Iterated Runge-Kutta (PIRK)* Verfahren, oder die Implementierung einer Schrittweitenkontrolle, können ein Bestandteil zukünftiger Forschungsarbeit sein. Ein weiteres größeres Forschungsfeld ist die effiziente Unterstützung von Stencil-Lösern.

In Kapitel 5.3.3 wurde auf die effiziente Implementierbarkeit von Multiplikationen und Divisionen mit einem Vielfachen von 2 bei Festkommazahlen eingegangen. In diesem Zusammenhang wäre es auch lohnenswert, unterschiedliche Runge-Kutta-Verfahren und ihre Möglichkeiten zur effizienten Implementierung auf FPGAs zu untersuchen. Die Entwicklung spezieller mathematischer Verfahren bietet sich an.

Durch eine klare Trennung und Weiterentwicklung des *Framework*-Teils der hier vorliegenden Arbeit könnte dieser als allgemeine Basis für die Implementierung von Berechnungen auf FPGAs verwendet werden. Eine Vielzahl von weiteren Optimierungen der Übersetzungslogik ist denkbar.

Literatur

[Fas+09] A. Fasih, T. Do Trong, J. C. Chedjou und K. Kyamakya. “New Computational Modeling for Solving Higher Order ODE Based on FPGA”. In: *2009 2nd International Workshop on Nonlinear Dynamics and Synchronization*. Juli 2009, S. 49–53 (zitiert auf den Seiten 21–23).

[HH15] Sarah Harris und David Harris. *Digital Design and Computer Architecture: ARM Edition*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2015.

[HVG11] Chen Huang, Frank Vahid und Tony Givargis. “A Custom FPGA Processor for Physical Model Ordinary Differential Equation Solving”. In: *IEEE Embedded Systems Letters* 3.4 (Dez. 2011), S. 113–116 (zitiert auf Seite 22).

[Intb] Intel. “Intel® Arria® 10 Core Fabric and General Purpose I/Os Handbook”. In: (), S. 336 (zitiert auf Seite 31).

[Inte] Intel. “Recommended HDL Coding Styles”. In: *Quartus II Handbook Version 13.1* (). URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/qts_qii51007.pdf (besucht am 16. Aug. 2020) (zitiert auf Seite 15).

[Osa+05] Y. Osana, T. Fukushima, M. Yoshimi et al. “A Framework for ODE-Based Multimodel Biochemical Simulations on an FPGAs”. In: *International Conference on Field Programmable Logic and Applications, 2005.* Aug. 2005, S. 574–577 (zitiert auf den Seiten 21–23).

[RR00] Thomas Rauber und Gudula Rünger. *Parallele und verteilte Programmierung*. Springer-Lehrbuch. Berlin Heidelberg: Springer-Verlag, 2000.

[Sta+17] Ioannis Stamoulias, Matthias Möller, Rene Miedema et al. “High-Performance Hardware Accelerators for Solving Ordinary Differential Equations”. In: *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies* (Bochum, Germany). HEART2017. New York, NY, USA: Association for Computing Machinery, 2017. URL: https://doi.org/10.1145/3120895.3120919 (zitiert auf den Seiten 22, 23, 45–49).

[Wan] Haibo Wang. “FPGA Logic Cells and Architecture”. URL: https://www.engr.siu.edu/haibo/ece428/notes/ece428_logcell.pdf (besucht am 4. Dez. 2018) (zitiert auf Seite 13).

[Xil20] Xilinx. “UltraScale Architecture and Product Data Sheet: Overview (DS890)”. In: (2020), S. 52 (zitiert auf Seite 45).

Webseiten

[@Alc19] Paul Alcorn. *AMD Ryzen 9 3900X and Ryzen 7 3700X Review: Zen 2 and 7nm Unleashed - Tom's Hardware | Tom's Hardware*. 2019. URL: https://www.tomshardware.com/reviews/ryzen-9-3900x-7-3700x-review,6214.html (besucht am 4. Aug. 2020) (zitiert auf Seite 5).

[@ByB] ByBell. *GTKWave*. URL: http://gtkwave.sourceforge.net/ (zitiert auf den Seiten 10, 11).

[@Inta] Intel. *FPGA Design Software - Intel® Quartus® Prime*. URL: https://www.intel.com/content/www/de/de/software/programmable/quartus-prime/overview.html (besucht am 15. Aug. 2020) (zitiert auf Seite 32).

[@Intc] Intel. *Intel® FPGA SDK for OpenCL™ Software Technology*. URL: https://www.intel.com/content/www/de/de/software/programmable/sdk-for-opencl/overview.html (besucht am 4. Aug. 2020) (zitiert auf Seite 10).

[@Intd] Intel. *Intel® Programmable Acceleration Card with Intel Arria® 10 GX FPGA - Overview*. URL: https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/acceleration-card-arria-10-gx/overview.html (besucht am 13. Aug. 2020) (zitiert auf den Seiten 24, 31).

[@Lin] Linux. *Linux Hugepage Support*. URL: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt (besucht am 20. Aug. 2020) (zitiert auf Seite 42).

[@MyH] MyHDL. *MyHDL*. URL: http://www.myhdl.org/ (besucht am 15. Aug. 2020) (zitiert auf Seite 32).

[@PyP20] PyParsing. *Pyparsing/Pyparsing*. 19. Aug. 2020. URL: https://github.com/pyparsing/pyparsing (besucht am 19. Aug. 2020) (zitiert auf Seite 39).

[@Tec] Gisselquist Technology. *Strategies for Pipelining Logic*. URL: https://zipcpu.com/blog/2017/08/14/strategies-for-pipelining.html (besucht am 17. Aug. 2020) (zitiert auf Seite 36).

[@Wil] Stephen Williams. *Icarus Verilog*. URL: http://iverilog.icarus.com/ (zitiert auf den Seiten 10, 32).

[@YAM] YAML Project. *YAML: YAML Ain't Markup Language*. URL: https://yaml.org/ (besucht am 16. Juli 2020) (zitiert auf Seite 27).

Abbildungsverzeichnis

Tabellenverzeichnis

Listing-Verzeichnis

Abkürzungsverzeichnis

FPGA Field Programmable Gate Array 8
ASIC Application Specific Integrated Circuit 8
LUT Look-Up-Table 13
DSP Digital Signal Processing 12
RAM Random-Access Memory 12
CAD Computer Aided Design 5
EDA Electronic Design Automation 5
RTL Register Transfer Level 6
HLS High Level Synthese 7
DUT Device Under Test 10
GPL GNU General Public License 10
SDC Synopsys Design Constraints 11
DSL Domain Specific Language 25
HPC High Performance Computing 31
OPAE Open Programmable Acceleration Engine 31
AFU Accelerator Function Unit 31
CCI Core Cache Interface 42
CSR Control / Status Register 42
PIRK Parallel Iterated Runge-Kutta 50

Anhang

A

A.1 Differentialgleichungssysteme

A.1.1 Rössler

$$
\begin{aligned}
x'(t) &= -y(t) - z(t) \\
y'(t) &= x(t) + ay(t) \\
z'(t) &= b + z(t) \cdot (x(t) - c)
\end{aligned}
$$

A.1.2 Lotka-Volterra (Predator-Prey)

$$
\begin{aligned}
y_1'(x) &= 0.1 \cdot y_1(x) - 0.2 \cdot y_1(x) \cdot y_2(x) \\
y_2'(x) &= -0.2 \cdot y_1(x) + 0.4 \cdot y_1(x) \cdot y_2(x)
\end{aligned}
$$

A.2 Runge-Kutta-Verfahren

Im Folgenden werden die für die Arbeit relevanten Runge-Kutta-Verfahren mit ihren Koeffizienten in Form des Butcher-Tableaus angegeben, dabei gilt:

$$
\begin{array}{c|c}
\mathbf{c} & A \\
\hline
 & \mathbf{b^T}
\end{array}
$$

A.2.1 Euler

$$
\begin{array}{c|c}
0 & \\
\hline
 & 1
\end{array}
$$

A.2.2 ModEuler

0		
$1/2$	$1/2$	
	0	1

A.2.3 Heun

0		
1	1	
	$1/2$	$1/2$

A.2.4 SSPRK3

0			
1	1		
$1/2$	$1/4$	$1/4$	
	$1/6$	$1/6$	$2/3$

A.2.5 RK4

0				
$1/2$	$1/2$			
$1/2$	0	$1/2$		
1	0	0	1	
	$1/6$	$1/3$	$1/3$	$1/6$

A.3 Konfiguration Logikgenerator

In Listing A.1 sind alle möglichen Konfigurationsparameter des Logikgenerators aufgelistet. Die Konfiguration kann in mehrere Dateien aufgeteilt werden und erlaubt so die einfache Wiederverwendung einzelner Teile. Manche der Parameter, wie zum Beispiel die Startwerte für das Anfangswertproblem, werden erst zur Laufzeit benötigt. Sind sie zum Zeitpunkt der Generierung angegeben, werden sie in der generierten .slv-Datei hinterlegt und somit als Standardwerte für die *Runtime* verwendet.

```
# Specifies the amount of parallel solvers to be generated.
nbr_solver: 1

# Internal numeric value representation
numeric:
    type: 'fixed'  # Currently nothing else supported
    fixed_point_signed: True
    fixed_point_fraction_size: 41
    fixed_point_nonfraction_size : 12

# Definition of runge kutta method to be used.
method:
  A: [[],
      [0.5],
      [0, 0.5],
      [0, 0, 1]]
  b: [0.1666666667, 0.3333333334, 0.3333333334, 0.1666666667]
  c: [0, 0.5, 0.5, 1]

# Specifies the initial value problem to be solved
problem:
    x: 0  # Only needed at runtime
    y:  # Only needed at runtime
      - 2
      - 1
    h: 0.17  # Only needed at runtime
    n: 60  # Only needed at runtime
    components:
      - 0.1 * y[0] - 0.2 * y[0] * y[1]
      - -0.2 * y[1] + 0.4 * y[0] * y[1]
```

Listing A.1: Konfigurationsparameter des Logikgenerators

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Bachelorarbeit, die anderen Quellen im Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Die Arbeit wurde bisher nirgends in gleicher oder ähnlicher Form zur Erlangung eines akademischen Grades eingereicht.

Bayreuth, 25. August 2020

Silas Bartel