Institut für Technische Informatik

Abteilung Rechnerarchitektur

Universität Stuttgart
Pfaffenwaldring 47
D - 70569 Stuttgart

Diplomarbeit Nr. 3354

Effiziente mehrwertige Logiksimulation verzögerungsbehafteter Schaltungen auf datenparallelen Architekturen

Alexander Schöll

Studiengang:	Informatik
Prüfer:	Prof. Dr. Hans-Joachim Wunderlich
Betreuer:	Dipl.-Inform. Claus Braun
	Dipl.-Inf. Stefan Holst
	Dipl.-Inf. Michael Kochte
begonnen am:	1. Juni 2012
beendet am:	1. Dezember 2012
CR-Klassifikation:	B.8.0, C.1.4, C.4, D.1.3, I.6.4, J.6

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

In der Entwurfsautomatisierung von hochintegrierten Schaltungen ist die Validierung eine zentrale Aufgabe. Ziel der Validierung ist die Untersuchung des Entwurfs einer hochintegrierten Schaltung auf Korrektheit, bevor der Entwurf in Form von Hardware realisiert wird. Die Spezifikation der hochintegrierten Schaltung formuliert ein erwartetes Verhalten. Dieses erwartete Verhalten wird mit dem Verhalten des Entwurfs verglichen. Die Validierung kann auf sämtlichen Abstraktionsebenen eines Entwurfs erfolgen. Dies reicht von der Validierung auf der Systemebene über die Register-Transfer-Ebene, zur Gatter-Ebene bis hin zur Validierung des Layouts [1].

Zwischen dem Wachstum der Komplexität hochintegrierter Schaltungen und der einhergehenden Entwurfsdauer existiert eine Diskrepanz. Die Komplexität hochintegrierter Schaltungen steigt nach dem "*Mooreschen Gesetz*" [2][3] exponentiell an. Dieses Gesetz beschreibt eine Verdopplung der Komplexität innerhalb einer hochintegrierten Schaltung in fortlaufenden Zeiträumen von 18 Monaten. Demgegenüber steht die Entwurfsdauer. Die Entwurfsdauer, die den Zeitraum zwischen der Entwicklung der Spezifikation und der Auslieferung der realisierten hochintegrierten Schaltung beschreibt, verringert sich nicht ausreichend genug, um in gleichbleibenden Zeiträumen dem exponentiellen Wachstum der Komplexität der Entwürfe begegnen zu können. Genauer unterliegt die Entwurfsproduktivität einem linearen Wachstum, der mit einer Verringerung der Entwicklungszeit um den Faktor 1,6 in einem Zeitraum von 18 Monaten [4] angegeben wird. Die Nachfrage des Marktes nach hochintegrierten Schaltungen legt der Entwurfszeit jedoch Grenzen auf (engl. *time-to-market constraints*). Die Folge ist eine zu niedrige Entwurfsproduktivität, was als *Design Productivity Gap* bezeichnet wird [4]. Innerhalb des Entwurfs einer hochintegrierten Schaltung werden 70% der Entwurfszeit für die Validierung aufgewendet [5]. Eine Verkürzung der aufgewendeten Zeit für die Validierung ist daher unentbehrlich.

Eine Methode, die das Verhalten von Schaltungsentwürfen hochintegrierter Schaltungen analysiert, ist die Simulation. Eine Simulation zeichnet sich durch die Fähigkeit zur punktuellen modellbasierten Untersuchung des Verhaltens eines Schaltungsentwurfs aus. Damit steht die Simulation im Kontrast zu einem elektrischen Experiment, in welchem die Realisierung des Schaltungsentwurfs auf ihr Verhalten hin untersucht wird. Die überwiegende Mehrheit der internen Zustände ist innerhalb eines Experimentes nicht direkt beobachtbar. Die Durchführung einer Simulation beginnt mit dem Anlegen von Stimuli an die Eingänge des betrachteten Schaltungsentwurfs. Anhand des vorgesehenen Simulationsmodells werden die Stimuli innerhalb einer Repräsentation des Schaltungsentwurfs ausgewertet und zu den Ausgängen propagiert. Durch die Erweiterung des Simulationsmodells um

ein Verzögerungsmodell kann das zeitliche Verhalten des Schaltungsentwurfs untersucht werden. Die fortlaufende Simulation verschiedener Sequenzen von Stimuli führt zu einer Menge von Ausgaben, die zusammengefasst das Verhalten des Schaltungsentwurfs unter den entsprechenden Eingabeszenarien beschreiben [6].

Die Validierung von Schaltungsentwürfen stellt eine Herausforderung dar. Die Abstraktion dieser Schaltungsentwürfe resultiert in strukturellen Netzbeschreibungen auf Gatterebene. Für größere Schaltungen bewegt sich die Anzahl der Gatter hierbei im Millionenbereich. Die Berechnungszeit, die bei der Validierung jener Netzbeschreibungen in kommerziellen Simulationswerkzeugen anfällt, wächst proportional mit der Anzahl der eingesetzten Gatter sowie der Anzahl der Simulationsszenarien [7].

Ein Teilaspekt der Validierung ist die Untersuchung des zeitlichen Verhaltens einer Schaltung. Eine Methode zur Durchführung dieser Abschätzung ist die Simulation auf Gatterebene. Hierzu existiert eine Reihe von Anwendungsbereichen, wie die verzögerungsbehaftete Fehlersimulation [8], die Untersuchung der Zuverlässigkeit [9], die Abschätzung des erwarteten Energieverbrauchs [10], Alterungsanalysen [11] sowie Testmengencharakterisierungen [12]. Das zeitliche Verhalten einer Schaltung ist von Variationen geprägt, welche beispielsweise durch nominale Änderungen der Materialparameter im Herstellungsprozess entstehen. Darüber hinaus werden Variationen durch Alterungsprozesse sowie Umgebungsparameter, welche in Form von Temperatur- oder Spannungsvariationen auftreten, begünstigt [13].

Müssen in den zuvor genannten Anwendungsbereichen Verzögerungsvariationen betrachtet werden, so ist es erforderlich, eine Vielzahl von Simulationen durchzuführen, welche jeweils unterschiedliche Ausprägungen von Variationen untersuchen. Eine sequentielle Durchführung sämtlicher Simulationen führt zu einer Vervielfachung der benötigten Zeit. Es ist also unentbehrlich, mittels innovativer Simulationstechniken den entstehenden Rechenaufwand zu reduzieren.

1.2 Ziele der vorliegenden Arbeit

In dieser Arbeit wird eine Lösung vorgestellt, welche die aufgewendete Zeit für die Durchführung einer Vielzahl verzögerungsbehafteter Simulationen signifikant reduziert. Die Lösung basiert auf der Verwendung von hochperformanten datenparallelen Architekturen, wie sie unter anderem in Grafikprozessoren implementiert sind. Die Vorgehensweise, welche die Ausführung allgemeiner Aufgaben auf Grafikprozessoren vorsieht, wird als GPGPU-Modell (*General Purpose Computation on Graphics Processing Units*) beschrieben [14].

Ein Ziel dieser Arbeit ist die Abbildung eines Algorithmus für verzögerungsbehaftete Simulationen auf Gatterebene und benötigte Datenstrukturen auf eine GPGPU-Architektur sowie deren Optimierung hinsichtlich der spezifischen Architektureigenschaften. In einer vorbereitenden Organisationsphase wird die Netzbeschreibung des Schaltungsentwurfs topologisch sortiert. Hierbei werden die einzelnen Gatter anhand der maximalen Pfadlänge ausgehend von den Eingängen angeordnet.

Gatter, die sich auf einer gemeinsamen topologischen Ebene befinden, besitzen zu einander keine Abhängigkeiten und können daher parallel ausgewertet werden. Ein weitergehendes Ziel ist die Verwendung der Parallelität der GPGPU-Architektur für die gleichzeitige Durchführung einer Vielzahl paralleler Simulationen. Hierzu werden Abhängigkeiten innerhalb der Datenstrukturen für die Repräsentation einer Schaltung untersucht und aufgelöst.

Die vorliegende Arbeit basiert auf einer *Ausgangsimplementierung* [15], deren Anwendungsbereich auf die Simulation kurzer endlicher Stimulifolgen begrenzt ist. Die Kapazität des Speichers begrenzt die Anzahl der Stimuli innerhalb endlicher Stimulifolgen. Eine Simulation über einen beliebig großen Zeitraum, welcher zum Beispiel zur hinreichend genauen Abschätzung des Einflusses von Variationen unentbehrlich ist, ist bisher nicht durchführbar. Ein wesentliches Ziel dieser Arbeit ist die vollständige Auflösung dieser Einschränkung. Hierzu wird ein kontinuierlicher Simulationsansatz entwickelt. Dies wird unter anderem durch eine optimierte dynamische Speicherverwaltung erreicht, so dass der Speicher von ausgewerteten Stimuli freigeben wird, um fortlaufend weitere Stimuli auswerten zu können. Zu jedem Simulationszeitpunkt betrachtet die Simulation ausschließlich denjenigen Teil der Stimulifolgen, welcher an diesem Simulationszeitpunkt ausgewertet wird.

Die Ausgangsimplementierung verwendet innerhalb der Simulation eine zweiwertige Boolesche Algebra. Ein weiteres Ziel dieser Arbeit ist die Verwendung einer vierwertigen Booleschen Algebra. Die Werte dieser Booleschen Algebra umfassen die Signalwerte 0 und 1, den undefinierten Zustand X sowie den hochohmigen Zustand Z. Im initialen Zustand einer Simulation können sequentielle Komponenten als nicht initialisiert und damit ohne definierten Wert betrachtet werden. Undefinierte Werte werden mit dem undefinierten Zustand X beschrieben. Der hochohmige Zustand Z wird innerhalb der Simulation von hardwareeffizienten Bussystemen benötigt. Diese Bussysteme können den gegenseitigen Ausschluss der Übertragung durch Tri-State-Elemente realisieren, welche den hochohmigen Zustand Z erzeugen können.

Die Simulationsimplementierung soll im Rahmen einer Validierung gegen ein kommerzielles Simulationswerkzeug ihre Korrektheit bestätigen, in dem sie das Verhalten von unterschiedlichen Schaltungen äquivalent berechnet. Im Zusammenhang der Validierung der entwickelten Simulationsimplementierung wird die Laufzeit der Auswertung der Validierungsszenarien dokumentiert und einander gegenübergestellt. Die zu entwickelnde parallele Herangehensweise steht sequentiellen Herangehensweisen kommerzieller Simulationswerkzeuge mit dem Ziel gegenüber, eine signifikant geringere Berechnungszeit bei gleichen Validierungsszenarien zu erreichen. Eine Analyse soll die Erreichung dieses Ziel bestätigen.

2 Gegenwärtiger Entwicklungsstand der Simulation auf datenparallelen Architekturen

Zur Simulation von Schaltungen auf Gatterebene existieren eine Reihe von Ansätzen. Hierbei werden verzögerungsfreie Ansätze von verzögerungsbehafteten Ansätzen unterschieden. Verzögerungsfreie Simulatoren verfolgen ebenen-orientierte (*plain simulation*) sowie ereignisgesteuerte Methoden (*event-driven simulation*). Ebenen-orientierte Methoden werten die Gatter einer Schaltung iterativ in topologischen Ebenen aus. Hierbei werden in jedem Simulationsdurchlauf sämtliche Gatter ausgewertet. Eine ebenen-orientierte Simulation kann in ein kompiliertes Programm übersetzt und ausgeführt werden (*Compiled Code Simulation* [13]). Dies führt zu signifikanten Laufzeitreduktionen, da eine aufwendige Interpretation innerhalb der Simulation außen vor gelassen wird [16]. Ein Nachteil der ebenen-orientierten Methode ist der hohe Anteil an redundanten Gatterauswertungen. Innerhalb großer Schaltungen erfährt nur ein geringer Teil der Gatter Änderungen (1 - 10 Prozent) [7]. Ereignisgesteuerte Methoden werten ausschließlich Gatter aus, an denen eine Änderung an den Eingängen des Gatters vorliegt. Hierbei werden jedoch komplexe dynamische Untersuchungen zur Bestimmung der Auswertungsreihenfolge von Gattern benötigt [17]. Ereignisgesteuerte Methoden werden darüber hinaus in verzögerungsbehafteten Simulationsalgorithmen eingesetzt. Ein Ereignis definiert hierbei einen Signalwechsel an einem bestimmten Zeitpunkt. In diesem Zusammenhang kann ein Verzögerungsmodell eingesetzt werden, welches die erzeugten Ereignisse an den einzelnen Gattern verzögert [13].

Die ebenen-orientierte Methode zeichnet sich durch einen geringen Anteil von Schleifen und Verzweigungen aus, wodurch sie sich für eine Parallelisierung anbietet. Die Parallelisierung der ereignisorientierten Methode besitzt einen zusätzlichen Aufwand, welcher von der dynamischen Bestimmung der Auswertungsreihenfolge sowie von der Verwaltung von Datenstrukturen ausgeht [18]. Im Folgenden werden Beiträge vorgestellt, in denen Methoden zur Parallelisierung der vorgenannten Simulationsansätze entwickelt wurden.

Im Beitrag von Suresh et. al. [18] werden drei Lösungsansätze vorgestellt, von denen zwei in einem Vergleich gegeneinander abgewogen werden. Gegenstand dieses Beitrags ist die Beschleunigung der funktionalen Validierung von Schaltungsentwürfen auf Gatterebene. Die Lösungsansätze umfassen eine datenparallele Methode, eine Pipeline-Methode sowie eine ereignisgesteuerte Methode. Sämtliche Methoden basieren auf einer Simulation, welche zwei Phasen beinhaltet. Innerhalb der ersten Phase werden die Gatter des Schaltungsentwurfs topologisch sortiert und in Ebenen gleicher Ordnung zusammengefasst. Auf einem Pfad zwischen zwei Gattern, dessen Pfadlänge mindestens eine topologische Ebene überspannt, werden stellvertretende Gatter eingefügt, die Stimuli über den Pfad propagieren. Die zweite Phase beinhaltet die eigentliche Simulation. Im Rahmen der daten-

parallelen Methode werden Stimuli, welche in einer Menge von Vektoren vorliegen, auf je einer topologischen Ebene parallel ausgewertet. Diese Methode sieht vor, eine Ebene nach der anderen auf der datenparallelen Architektur auszuwerten. Die Pipelining-Methode erweitert die Herangehensweise der datenparallelen Methode. Der Unterschied liegt in der parallelen Auswertung sämtlicher topologischen Ebenen innerhalb der Pipelining-Methode. Die Speichereinheiten, die zwischen den topologischen Ebenen angeordnet sind, werden innerhalb der Pipelining-Methode dupliziert, da ein gleichzeitiges Lesen und Beschreiben einer einzigen Speichereinheit zu Konflikten führt. Zwischen den Simulationszyklen alternieren die Speichereinheiten zwischen den topologischen Ebenen. Die dritte ereignisgesteuerte Methode verfügt über eine Liste für jeden Gattereingang, in der eine anstehende Aktivität dokumentiert wird. Anhand dieser Liste werden für jede Ebene nur Gatter ausgewertet, bei denen eine Aktivität vorliegt. Die Ausrichtung der Lösungsansätze auf die reine funktionale Validierung schließt eine Verwendung innerhalb von Untersuchungen zum erwarteten Energieverbrauch aus, da hierzu die Berechnung von Aktivitäten unter der Anwendung eines Verzögerungsmodells erforderlich ist.

Der Beitrag von Chatterjee et. al. [7] stellt eine ereignisgesteuerte Simulation auf Gatterebene vor. Die Simulation ist in zwei Phasen unterteilt. In der ersten Phase wird eine Segmentierung der Netzbeschreibung in so genannte Makrogatter durchgeführt. Die zweite Phase umfasst die eigentliche Simulation. Ein Makrogatter beinhaltet eine Teilmenge der Gatter, welche in der Netzbeschreibung untereinander verbunden sind. Die Eingänge eines Makrogatters werden mittels einer Sensitivitätsliste beobachtet. Falls ein Ereignis an die Eingänge des Makrogatters propagiert wird, wird dies in der Sensitivitätsliste dokumentiert. In der Folge werden ausschließlich Makrogatter simuliert, an deren Eingängen ein Ereignis aufgetreten ist. Innerhalb der Erzeugungsphase der Makrogatter werden einzelne Gatter, die zu zwei Makrogattern gehören, dupliziert, um eine unabhängige und daher parallele Ausführung verschiedener Makrogatter erreichen zu können. Die einzelnen Gatter innerhalb eines Makrogatters werden anschließend ausbalanciert. Ziel dieses Schrittes ist die gleichmäßige Verteilung von Gattern auf sämtlichen Ebenen des Makrogatters um eine gleichbleibende Anzahl von Gattern auf jeder Ebene zu erreichen. Hierdurch wird die parallele Ausführung begünstigt, da Leerläufe auf Ebenen mit wenigen Gattern reduziert werden. Die ereignisgesteuerte Simulation erreicht eine Beschleunigung von 13x im Vergleich mit einem kommerziellen Simulationswerkzeug. Dieser Beitrag berücksichtigt kein Verzögerungsmodell. Daher ist er für eine präzise Untersuchung des erwarteten zeitlichen Verhaltens nicht in ausreichendem Maße geeignet.

In einen weiteren Beitrag von Chatterjee et. al. [17] wird eine analoge Herangehensweise zur Zusammenfassung von Gattern vorgestellt. Darin werden Gatter, die Stimuli zu einem gemeinsamen Ausgang propagieren, innerhalb eines Kegels zusammengefasst. Innerhalb der ersten Phase der Simulation werden entsprechend dieses Ansatzes sämtliche Gatter auf Kegel abgebildet. Für die Abbildung der Gatter auf die GPGPU-Architektur werden darüber hinaus Gruppen(*engl. Cluster*) gebildet, welche die Kegel zusammenfassen. Gatter werden dupliziert, wenn sie zu mehreren Kegeln gehören, um eine parallele Auswertung verschiedener Gruppen zu ermöglichen. In Experimenten wurde eine Duplizierung bei 15% sämtlicher Gatter beobachtet. 7% der Gatter wurden hierbei mehrfach dupliziert.

Anschließend wird innerhalb der Gruppen analog zum ersten Beitrag eine Balancierung durchgeführt, womit die Anzahl der Gatter auf verschiedenen Ebenen angeglichen und darüber hinaus die Anzahl der Ebenen reduziert wird. Diese Herangehensweise erreicht eine Beschleunigung von 14,4x im Vergleich mit einem kommerziellen Simulationswerkzeug. Dieser Ansatz ist ebenfalls innerhalb einer präzisen Untersuchung des erwarteten zeitlichen Verhaltens nicht hinreichend genug geeignet, da die Berücksichtigung eines Verzögerungsmodells innerhalb der Simulation außen vor bleibt.

Gulatir und Khatri beschreiben in ihrem Beitrag [19] einen Lösungsansatz für die Abbildung einer Monte Carlo Simulation auf GPGPU-Architekturen. Zweck dieser Abbildung ist die statistische statische Untersuchung des zeitlichen Verhaltens von Schaltungen auf Gatterebene. Der Beitrag verfolgt damit das Ziel, eine möglichst genaue Bewertung der Verzögerungen durchzuführen, die von Variationen beeinflusst werden. Innerhalb der parallelen Implementierung werden eine Reihe von verschiedenen Simulationen an einem einzelnen Gatter parallel durchgeführt. Hierzu wird eine Menge von Stichproben erzeugt, die mittels einer statischen Untersuchung des zeitlichen Verhaltens untersucht werden. Innerhalb dieser Untersuchung wird für jeden Stimulus an den Eingängen eines Gatters die Ankunftszeit um die spezifische Verzögerung erhöht, welche in der Folge eine Änderung des Signalwertes am Ausgang bewirkt. Die Herangehensweise sieht nun vor, das Maximum sämtlicher errechneter Zeitpunkte zu propagieren. Die Implementierung dieses Ansatzes erreicht eine Beschleunigung von 260x im Vergleich mit einem kommerziellen Simulationswerkzeug. Dieser Beitrag unterschätzt Einflüsse durch Hazards, da eine Propagierung der jeweils maximalen Verzögerungszeit Stimuli filtert, die zu Hazards führen. Aktivitäten innerhalb von Schaltungen, die von Hazards ausgelöst werden, sind jedoch relevant für die Untersuchung des erwarteten Energieverbrauchs.

Im Beitrag von Wang et. al. [20] wurde der Chandy-Misra-Byrant-Algorithmus [21] im Kontext einer diskreten ereignisgesteuerten Simulation angewandt. Kern dieses Ansatzes ist die Unterteilung des Schaltungsentwurfs in interaktive Komponenten, die als logische Prozesse bezeichnet werden. Diese logischen Prozesse führen die Simulation in ihrer Schaltungspartition unter einer lokalen Simulationszeit durch. Die einzelnen logischen Komponenten propagieren Ereignisse über Nachrichten. Jede logische Komponente verfügt zur Auswertung von Ereignissen über eine Nachrichten-Schlange, in welcher ankommende Nachrichten abgelegt werden. Die einzelnen Nachrichten verfügen über Zeitstempel, mit denen die Reihenfolge der eintreffenden Ereignisse bestimmt wird. Zur Vermeidung von Verklemmung (*engl. Deadlock*) innerhalb von zyklischen Abhängigkeiten zwischen logischen Komponenten werden leere Nachrichten (*null-messages*) verwendet. Die Ausführung der einzelnen Prozesse wird parallel durchgeführt, da die Verarbeitung der Nachrichten durch die einzelnen Prozesse unabhängig voneinander ist. Die datenparallele Implementierung erreicht im Vergleich mit einer sequentiellen Implementierung des Ansatzes eine Beschleunigung von 29,2x.

Im Mittelpunkt des Beitrages von Sen et. al. [22] stehen Und-Inverter-Graphen (*And-Inverter-Graphs*). Im vorbereitenden Schritt zu einer Simulation werden sämtliche Gatter eines Schaltungsentwurfs in Und-Inverter-Gatter überführt. Innerhalb des Beitrages werden zwei Vorgehensweisen zur Bildung von Clustern in Schaltungsentwürfen vorgestellt. Das Ziel der Bildung von Clustern ist die parallele Evaluierung jener Cluster, durch die gezielte Vermeidung von Abhängigkeiten. Die erste Vorgehensweise zur Bildung von Clustern begrenzt die Anzahl der Gatter pro Cluster durch einen Schwellenwert. Ausgehend von den Ausgängen der Schaltung werden fortlaufend Gatter einer Ebene zum Cluster hinzugefügt, bis die Anzahl der Gatter den Schwellenwert übersteigt. Weitere Cluster werden ausgehend von den Eingängen der erzeugten Cluster erzeugt. Gatter, die zu mehreren Clustern hinzugefügt wurden, werden dupliziert. Die zweite Vorgehensweise zur Bildung von Clustern ergänzt die erste Vorgehensweise um einen Verflechtungsschritt. Hierzu werden Cluster mit dem Ziel verflechtet, eine möglichst gleichmäßige Anzahl von Gattern pro Cluster zu erreichen. Unterschiedlich dimensionierte Cluster führen zu Leerlaufzeiten auf der datenparallelen Architektur. Die verflochtenen Cluster werden anschließend ausbalanciert, wobei die Anzahl der Gatter innerhalb der einzelnen Ebenen in eine gleichmäßige Form überführt wird. Im Vergleich mit einem sequentiellen Simulationswerkzeug erreicht die Implementierung, welche die erste Vorgehensweise zur Bildung von Clustern verfolgt eine Beschleunigung von 5x. Die Implementierung, welche die zweite Vorgehensweise realisiert, erreicht eine Beschleunigung von 21x.

3 Grundlagen zur kontinuierlichen Simulationsimplementierung

3.1 Simulationsmodell verzögerungsbehafteter Schaltungen auf Gatterebene

Struktur von Schaltungen

Das Simulationsmodell, welches der vorliegenden Arbeit zugrunde liegt, abstrahiert Schaltungen auf Gatterebene. Die Schaltungen werden als azyklischer gerichteter Graph dargestellt. Die Knoten des Graphen repräsentieren Gatter, sowie Ein- und Ausgänge der Schaltung. Die Kanten des Graphen repräsentieren Signale einer Schaltung. Die Menge der Gatter besteht aus kombinatorischen Gattern mit höchstens zwei Eingängen. In Tabelle 3.1 sind die logischen Gatter angeführt, welche explizit im Simulationsmodell abgebildet wurden.

logisches Gatter	BUF	AND	NAND	OR
deutsche Bezeichnung	Puffer	Und	Nicht Und	Oder
logisches Gatter	NOR	XOR	XNOR	NOT
deutsche Bezeichnung	Nicht Oder	Exklusiv-Oder	Nicht Exklusiv-Oder	Nicht

Tabelle 3.1: Im Simulationsmodell explizit abgebildete logische Gatter

Stimuli

Innerhalb der Simulation werden Stimuli als Datentyp zur Darstellung von Signalwechseln verwendet. Ein Stimulus beinhaltet die Information über den Zeitpunkt des Signalwechsels sowie den anschließenden Wert des Signals. Fortlaufende Signalwechsel werden in einer Stimulifolge zusammengefasst, in welcher die Stimuli aufsteigend nach dem Zeitpunkt angeordnet werden. Die Menge der Signalwerte, die ein Stimulus annehmen kann, basiert auf einer vierwertigen Booleschen Algebra. Die Menge der Signalwerte umfasst die Signalwerte 0 und 1, den undefinierten Zustand X sowie dem hochohmigen Zustand Z.

Verzögerungsmodell

Innerhalb des Simulationsmodells wird ein Verzögerungsmodell angewandt. Dies ergänzt die Untersuchung des funktionalen Verhaltens um die Untersuchung des zeitlichen Verhal-

ten einer Schaltung. Jedes Gatter einer Schaltung verfügt über explizite Verzögerungsparameter zu jedem Eingang. Für jeden Eingang werden drei Parameter verwendet. Darin sind Verzögerungen für Signalübergänge zu den Signalwerten 0 und 1, sowie für Signalübergänge zum undefinierten Zustand X definiert (engl. *Transport Delay*). Verzögerungsparameter, die einen Übergang zum hochohmigen Zustand Z vorsehen, wurden außen vor gelassen, da der Signalwert Z an den Ausgängen der momentan verwendeten Gatter (siehe Tabelle 3.1) nicht erzeugt werden kann. Sämtliche Parameter sind unabhängig zu einander und können daher individuelle Verzögerungsszenarien an sämtlichen Signalen innerhalb der Schaltung modellieren.

Variationen von Verzögerungen

Das Simulationsmodell ist fähig, Verzögerungen zu variieren. Die explizit angegebenen Verzögerungen werden hierbei verlängert beziehungsweise verkürzt. Der angewandte Variationsfaktor resultiert hierbei aus einer Normalverteilung. Der exakte Einfluss der Variation ist für jede einzelne Verzögerungsart durch Modellparameter explizit definierbar. Wie zuvor erläutert entstehen Variationen des zeitlichen Verhaltens unter anderem durch Variationen im Herstellungsprozess, welche zu nominalen Änderungen der Materialparameter führen. Darüber hinaus beeinflussen Temperatur- oder Spannungsvariationen, welche aus der direkten Einsatzumgebung der Schaltung stammen, die Verzögerungseigenschaften der Schaltung [13].

Trägheiten

Das Simulationsmodell bildet Trägheiten an den Eingängen von Gattern ab (engl. *Inertial Delay*). Damit ein Signalwechsel am Eingang eines Gatters zu einer Propagierung führt, muss es die Trägheit überwinden. Das heißt, dass ein Signal eine definierte Mindestzeit stabil anliegen muss, damit es propagiert werden kann [23]. Das Verhalten des Simulationsmodells mit und ohne Berücksichtigung von Trägheiten ist in Abbildung 3.1 dargestellt.

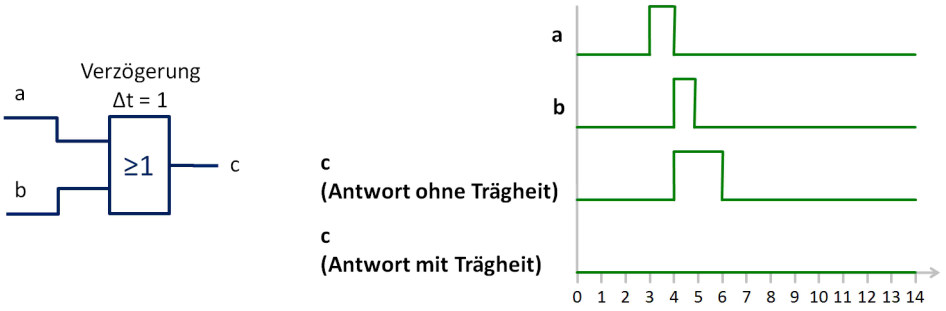


Abbildung 3.1: Modellierung von Trägheiten in einer Schaltung (engl. *Inertial Delay*)

3.2 Grundlagen zur Implementierung des Simulationsmodells

3.2.1 Das Simulationsframework ADAMA

Das *ADAMA*-Simulationsframework (abk. Adaptive Diagnosis of Arbitrary Manifold Artifacts) ist ein Projekt innerhalb des Institutes für Technische Informatik der Universität Stuttgart. Das Simulationsframework bietet Funktionalitäten an, welche zur Untersuchung des Verhaltens von Schaltungen genutzt werden können. Dies umfasst eine umfangreiche Schaltungs- und Stimuliverarbeitung. Die Simulationsimplementierung, welche in der vorliegenden Arbeit vorgestellt wird, erweitert dieses Simulationsframework.

Das *ADAMA*-Framework ist durch eine umfangreiche Erweiterungsfähigkeit gekennzeichnet. Dies spiegelt sich im Design der Software durch das Merkmal einer Auftragsverwaltung wieder. Aufträge werden in der Programmiersprache Java erstellt und können über die Kommandozeilenschnittstelle von *ADAMA* ausgeführt werden. Den Aufträgen steht durch diese Herangehensweise die gesamte Funktionalität von *ADAMA* zur Verfügung.

Die Verarbeitung und Simulation des Verhaltens von Schaltungen innerhalb des *ADAMA*-Frameworks orientiert sich am vorgestellten Simulationsmodell. Die Darstellung von Stimuli wird im *ADAMA*-Framework mittels Eingabemustern durchgeführt, die in Blöcken von 64 Mustern zusammengefasst werden. *ADAMA* bietet darüber hinaus Funktionalitäten für die Validierung von Simulationsimplementierungen an. Die Funktionalitäten ermöglichen die Verwendung der Schaltungen in kommerziellen Simulationswerkzeugen, welche im Bereich der Entwurfsautomatisierung (engl. *Electronic Design Automation - EDA*) eingesetzt werden. Hierzu können Schaltungen, die in den Datenstrukturen des *ADAMA*-Frameworks gehalten werden, in Hardwarebeschreibungsprachen wie *VHDL* oder *Verilog* exportiert werden. Darüber hinaus besteht die Möglichkeit Schaltungen, welche in Hardwarebeschreibungsprachen formuliert sind, zu importieren [24].

3.2.2 Das Simulationswerkzeug ModelSim

Zur Validierung der entwickelten Simulationsimplementierung wird das kommerzielle Simulationswerkzeug *ModelSim* des Herstellers *Mentor Graphics* eingesetzt. Die Kernbestandteile von *ModelSim* umfassen Funktionalitäten zur Simulation von Schaltungen auf verschiedenen Abstraktionsebenen. *ModelSim* umfasst darüber hinaus Methoden zum Import von Schaltungen, welche in Hardwarebeschreibungssprachen wie *VHDL* oder *Verilog* vorliegen sowie die Unterstützung von Skripten zur Automatisierung von Simulationen [25]. Die im Rahmen der vorliegenden Arbeit entwickelte Simulationsimplementierung wurde gegen das Simulationswerkzeug *ModelSim* ausführlich für eine Vielzahl von Schaltungen validiert. Die Methoden der Validierung und die Ergebnisse dieser Schritte werden im Kapitel 5 vorgestellt.

3.2.3 Parallelisierung

Ein zentrales Ziel der vorliegenden Arbeit ist die Erreichung einer größtmöglichen Effizienz innerhalb der Implementierung des vorgestellten Simulationsmodells. Zur Erfüllung dieses Ziels wird die Methode der Parallelisierung angewandt [26]. Sequentielle Algorithmen werden parallelisiert, in dem Teilprobleme innerhalb des Algorithmus identifiziert werden, die zu einander keine Abhängigkeiten besitzen. Die parallele Ausführung jener Teilprobleme führt zu einer Reduzierung der Laufzeit des Algorithmus.

Im Bereich der Hardwarearchitekturen existieren verschiedene Herangehensweisen zur Parallelisierung. Hierzu werden Hardwarestrukturen auf die parallele Ausführung von Algorithmen optimiert. Für die explizite Verwendung dieser Hardwarearchitekturen existieren Programmierschnittstellen, welche die darunter liegenden Architekturen abstrahieren. Hierdurch wird die Ausnutzung jener Architekturen durch gezielte Erweiterungen auf der Softwareebene gewährleistet.

- CPU-Architekturen
  - Multi-Core-Architekturen
  - Simultaneous Multithreading
  - SIMD-Erweiterungen [27]
- GPU-Architekturen
  - GPGPU
    - Nvidia Compute Unified Device Architecture [28]
    - AMD Accelerated Parallel Processing [29]
    - Khronos Group OpenCL [30]
    - Microsoft DirectCompute [31]

Das GPGPU-Modell

Das Akronym *GPGPU* steht stellvertretend für *General Purpose Computation on Graphics Processing Unit* [32] und beschreibt ein Konzept zur Verwendung von Grafikprozessoren außerhalb ihrer eigentlichen Aufgaben. Diese sind im Bereich der Beschleunigung von Berechnungen zur Bildsynthese definiert. Durch die freie Programmierung von Grafikprozessoren können die spezifischen Eigenschaften der Architekturen von Grafikprozessoren genutzt werden.

Der Vergleich von CPU und GPU-Architekturen zeigt einen Unterschied in den Ansätzen zur Parallelisierung auf. Dies wird innerhalb der Kategorisierung zur Granularität der Parallelität deutlich. Die Parallelisierung im Bereich der CPU-Architekturen findet auf der Ebene der Prozesse sowie Threads statt. Die Parallelisierung dieser Instruktionsstrukturen wird durch den Einsatz von Multi-Core- sowie Simultaneous Multithreading-Konzepten

durchgeführt. Mit Erweiterungen, die im Hinblick auf die Taxonomie nach *Flynn* [33] in der Kategorie *Single Instruction Multiple Data (SIMD)* definiert sind, begegnen CPU-Architekturen datenparallelen Algorithmen. Im Vergleich zu CPU-Architekturen ist die Orientierung an datenparallelen Algorithmen bei GPU-Architekturen signifikant ausgeprägter. Die Granularität der Parallelität ist bei GPU-Architekturen auf der Ebene der jeweiligen Instruktionspfade definiert, die auf den Berechnungseinheiten des Grafikprozessors ausgeführt werden. Darüber hinaus wird die Ausprägung der GPU-Architekturen zugunsten datenparalleler Algorithmen deutlich, wenn die Aufteilung von Prozessorfläche betrachtet wird. CPU-Architekturen werden von Caches und aufwändigen Einheiten zur Organisation des Instruktionsflusses in Form von Pipelining sowie Super-Skalarität dominiert. GPU-Architekturen hingegen sind von einer Vielzahl skalarer Recheneinheiten geprägt. Die Organisation der einzelnen Instruktionsflüsse wird innerhalb von Berechnungseinheiten zusammengefasst [32][34].

Die CUDA-Architektur

Die in der vorliegenden Arbeit verwendete datenparallele Architektur ist die *CUDA*-Architektur. Die Implementierung des vorgestellten Simulationsmodells, sowie die einhergehende Validierung und Analyse wurden auf der *CUDA*-Architektur durchgeführt. Darüber hinaus wurde die Implementierung für die Verwendung als Simulationswerkzeug innerhalb der *CUDA*-Architektur vorbereitet.

CUDA steht als Akronym für *Compute Unified Device Architecture* und beschreibt eine seit 2006 entwickelte Architektur der Firma *Nvidia* [28]. Diese Architektur umfasst Hardware in Form von sogenannten *CUDA*-fähigen Grafikprozessoren, sowie Software, die eine Abstraktion der datenparallelen Architektur durchführen und darüber hinaus die freie Programmierung der Grafikprozessoren im Rahmen des *GPGPU*-Modells ermöglichen. Die *CUDA*-Architektur umfasst eine Reihe von Bestandteilen, die sich in Eigenschaften der Hardwarearchitekturen sowie in Softwareschnittstellen widerspiegeln [35].

- Grafikprozessoren, die im Hinblick auf das *GPGPU*-Modell frei programmierbar gestaltet sind und daher als parallele Berechnungseinheiten verwendet werden können.
- Schnittstellen zum jeweilig eingesetzten Betriebssystem, mit denen die Initialisierung, Konfiguration und Interaktion mit der Hardware von Seiten der Software durchgeführt wird.
- Treiber, die in der Benutzerebene des Betriebssystems die Aufgabe besitzen, eine Programmierschnittstelle zur Hardware für Entwickler anzubieten.
- Ein Befehlssatz, der für die parallele Ausführung von Programmteilen verwendet wird. Dieser Befehlssatz wird unter der Abkürzung *PTX* referenziert (*Parallel Thread Execution*).

Im Rahmen von Implementierungsarbeiten, welche innerhalb der CUDA-Architektur durchgeführt werden, stehen zwei Programmierschnittstellen zur Verfügung.

- Die Programmierschnittstelle auf Geräteebene kann als Schnittstelle zu einer ausgewählten Applikationsschnittstelle verwendet werden. Ausgewählte Applikationsschnittstellen sind hierbei *DirectX*, *OpenCL* sowie die zur *CUDA*-Architektur zugeordneten *CUDA Driver API*. Die Implementierung der Programmteile, die parallel ausgeführt werden, erfolgt hierbei innerhalb der ausgewählten Applikationsschnittstellen.
- Eine weitere Programmierschnittstelle steht durch die Benutzung der Erweiterungen zur Programmiersprache *C* zur Verfügung (*C for CUDA*). Hierbei werden hardwarenahe Aufgaben des Grafikprozessors abstrahiert und müssen daher nicht explizit behandelt werden. Darüber hinaus stehen im Rahmen der Implementierung sämtliche Konzepte der Programmiersprache *C* in Form von Typen und Funktionen zur Verwendung auf dem Grafikprozessor bereit.

Die Abstraktion der CUDA-Architektur

Zur Parallelisierung von Implementierungen, welche im Rahmen der *CUDA*-Architektur verwendet werden sollen, müssen die Konzepte zur Abstraktion der Hardware berücksichtigt werden [34].

Die Berechnungseinheiten eines Grafikprozessors, welche die *CUDA*-Architektur implementieren, können nach der Taxonomie von *Flynn* in die Kategorie *SPMD* (*Single Programm Multiple Data*) eingeteilt werden. Eine Berechnungseinheit führt eine Folge von Instruktionen auf einer Menge von Daten aus. Eine Instruktionsfolge, welche auf den Daten angewandt wird, wird als *Thread* bezeichnet. Die Menge der *Threads*, die eine Berechnungseinheit unter der Anwendung einer Instruktionsfolge bearbeitet, wird als *Warp* bezeichnet. Die spezifischere Einteilung in die Kategorie *SPMD* im Hinblick auf die vorgenannte Einteilung *SIMD* rührt daher, dass die Ausführung der Instruktionen von Verzweigungen abhängen kann. Verzweigungen führen dazu, dass die Menge der Threads in Abhängigkeit der erfüllten Bedingung in Partitionen unterteilt wird. Die Partition, welche die Bedingung nicht erfüllt, führt die Instruktionen innerhalb der Verzweigung nicht aus.

Um die Ausführung eines datenparallelen Algorithmus auf beliebig großen Datenmengen durchführen zu können, die einem Vielfachen eines *Warps* entsprechen, können *Warps* zu *Blöcken* zusammengefasst werden. Die Anzahl der *Threads* in einem *Block* ist durch die Kapazität eines Prozessorkerns begrenzt. Um eine beliebige Menge an verfügbaren Prozessorkernen an die Ausführung einer Instruktionsfolge zu binden, können *Blöcke* zu *Gitter* verbunden werden. Im Rahmen der *CUDA*-Architektur wird eine Instruktionsfolge, die auf den beteiligten Berechnungseinheiten eines *Gitters* ausgeführt wird, als *Kernel* bezeichnet. Die größtmögliche Anzahl der Blöcke ist nicht durch Parameter der Hardware begrenzt. Werden mehr *Blöcke* erzeugt, als der Grafikprozessor Berechnungseinheiten zur Ausführung bereit stellen kann, werden die Blöcke sequentiell ausgeführt.

Grafikprozessoren, die eine Implementierung der *CUDA*-Architektur beinhalten, werden im Kontext ihrer Spezifikationen und Eigenschaften klassifiziert. Das Ergebnis dieser

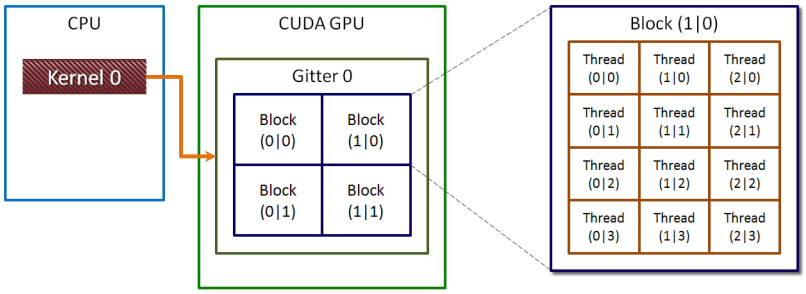


Abbildung 3.2: Die Abstraktion der Berechnungseinheiten in der CUDA-Architektur

Klassifizierung ist ein Index, der im Rahmen der *CUDA*-Architektur als *Compute Capability* beschrieben wird. Dieser Index definiert darüber hinaus die Generation, zu der ein Grafikprozessor gehört. Durch die Kenntnis der Generation des eingesetzten Grafikprozessors können im Implementierungsprozess gezielte Optimierungen im Hinblick auf die verfügbaren Eigenschaften vorgenommen werden [36].

Die Speicherhierarchie der CUDA-Architektur

Die Speicherhierarchie der *CUDA*-Architektur ist von Strukturen geprägt, welche aus den Optimierungen der eigentlichen Aufgaben des Grafikprozessors resultieren. Diese Strukturen wurden vollständig in die *CUDA*-Architektur abgebildet und stehen für die Implementierung von Lösungen innerhalb des GPGPU-Modells zur freien Verfügung. Die Prozessorkerne beinhalten einen Registerspeicher, der in partitionierter Form den einzelnen *Threads* zur Verfügung steht. Dieser Registerspeicher hat einen Umfang von 8000 32-Bit Register in Grafikprozessoren der *Compute Capability* 1.0 bis hin zu 64000 32-Bit Register in Grafikprozessoren der *Compute Capability* 3.0 [36]. Darüber hinaus besitzt jeder Thread Zugriff zu einem lokalen Speicher, der im Hauptspeicher des Grafikprozessors angelegt wird. Die *Threads* eines Blocks besitzen Lese- und Schreibrechte in einem gemeinsamen Speicher. Darüber hinaus stehen drei weitere Speicher im Rahmen des Hauptspeichers des Grafikprozessors zur Verfügung, die aus Sicht der *Threads* unterschiedliche Zugriffsmöglichkeiten besitzen. Der globale Speicher ist für sämtliche *Threads* sichtbar und darüber hinaus für die *Threads* schreib- und lesefähig. Der Textur- sowie Konstantenspeicher ist für *Threads* ausschließlich lesbar definiert. Einen Überblick über sämtliche Speicherarten im Kontext der zugehörigen Verhaltensparameter ist in Tabelle 3.2 dargestellt [34].

Speicher	Caching	Zugriffsart	Zugriffszeit
Register	Nein	Lesen/Schreiben	22 Zyklen
Lokaler Speicher	Ja*	Lesen/Schreiben	400-800 Zyklen
Gemeinsamer Speicher	Nein	Lesen/Schreiben	22 Zyklen
Globaler Speicher	Ja*	Lesen/Schreiben	400-800 Zyklen
Konstantenspeicher	Ja	Lesen	400-800 Zyklen
Texturspeicher	Ja	Lesen	400-800 Zyklen

Tabelle 3.2: Übersicht zur Speicherhierarchie der *CUDA*-Architektur
*Ab *Compute Capability* 2.0 [36]

Optimierung der Effizienz im Hinblick auf die Verwendung der CUDA-Architektur

Eine detaillierte Analyse der eingesetzten Architektur ist unter der Zielsetzung zur Erreichung einer größtmöglichen Effizienz unentbehrlich. Die *CUDA*-Architektur besitzt limitierende Faktoren innerhalb der Ausführung von Instruktionsfolgen sowie in einhergehenden Speicherzugriffen. Diese limitierenden Faktoren müssen explizit bei der Entwicklung von Implementierungen berücksichtigt werden [37].

- Die Anzahl der global durchgeführten Speicherzugriffe innerhalb der parallel auszuführenden Instruktionsfolgen muss minimiert werden. Wie aus Tabelle 3.2 hervorgeht sind Zugriffe in den globalen Speicher von einer Latenz geprägt, die einem Vielfachen jener Latenz entspricht, die bei einem Zugriff von Registern oder gemeinsamen Speicher entsteht.
- Die Verwendung von arithmetischen Instruktionen, die über einen geringen Durchsatz zugunsten höherer Genauigkeit verfügen, sollte reduziert werden, falls für das Endergebnis eine hohe Genauigkeit nicht relevant ist.
- Die Anzahl divergierender Kontrollflüsse innerhalb der zu einer Berechnungseinheit zugeordneten Instruktionsfolgen muss minimiert werden. Durch Verzweigungen entstehende divergierende Kontrollflüsse werden sequentiell ausgeführt. Dies führt dazu, dass die Berechnungseinheit die ihr zugeordneten Daten partitioniert und Instruktionen nur auf jener Partition ausführt, welche die Bedingung der Verzweigung erfüllt. Die Berechnungszeit erhöht sich somit um die sequentiell ausgeführten Berechnungszeitanteile.

3.3 Die Ausgangsimplementierung

Eine wesentliche Grundlage der Arbeit ist die im Rahmen des *ADAMA*-Simulationsframeworks existierende Simulationsimplementierung. Wie zuvor erläutert besitzt diese Ausgangsimplementierung eine Reihe von Beschränkungen. Diese Beschränkungen führen zu einer verringerten Anwendungsfähigkeit im Rahmen der Prognose des erwarteten zeitlichen Verhaltens. Im Folgenden wird dieser Implementierungszwischenstand detailliert

analysiert und die existierenden Beschränkungen vorgestellt. Im Anschluss daran sind sämtliche Beschränkungen zusammengefasst aufgelistet, von denen ausgehend Lösungsansätze vorgestellt werden. Deren Realisierung wird im Kapitel 4 vorgestellt.

Im Folgenden wird die im *ADAMA*-Simulationsframework bestehende Simulationsimplementierung als *Ausgangsimplementierung* bezeichnet, während die im Rahmen dieser Diplomarbeit erarbeiten Methoden, Lösungsansätze sowie Implementierungen als *CWT-Sim (Continuous Wave Timing Simulation - verzögerungsbehaftete Simulation kontinuierlicher Stimulifolgen* bezeichnet werden.

3.3.1 Einsatzumgebung

Die Ausgangsimplementierung liegt im *ADAMA*-Simulationsframework in einer datenparallelen Implementierung, sowie für Zwecke der Validierung als sequentielle Implementierung vor. Die sequentielle Implementierung wurde analog zum *ADAMA*-Simulationsframework in der Programmiersprache *Java* entwickelt. Die datenparallele Implementierung liegt als externe Anwendung vor, welche zur Kommunikation mit dem *CUDA*-Applikationsframework durchführt. Daher sind diese Implementierungsteile in der Programmiersprache *C* entwickelt.

Die Ausgangsimplementierung besitzt eine Schnittstelle zur Prozesskommunikation, um innerhalb des *ADAMA*-Simulationsframework eine Kommunikation mit der externen Anwendung durchführen zu können. Die externe Anwendung wird zur Durchführung von Simulationen als Unterprozess erzeugt, wobei gemeinsame Speicherbereiche im Hauptspeicher genutzt werden, um Simulationsdaten von der *Java*-Umgebung in den Speicher der GPU zu schreiben oder zu lesen.

3.3.2 Darstellung von Stimuli

Ein Implementierungsdetail, welches zur Untersuchung des Verhaltens von Schaltungen grundlegend beeinflusst, ist die eigentliche Darstellung von Stimuli. Das *ADAMA*-Simulationsframework benutzt zur Darstellung von Stimuli Muster (*engl. Patterns*). Die Ausgangsimplementierung verwendet für Stimuli einen Datentyp, in welchem die ausschließliche Repräsentation von Änderungen eines Signals im zeitlichen Verlauf repräsentiert werden. Ein Vergleich der beiden Ansätze ist in Abbildung 3.3 dargestellt.

Der in diesem Kontext verwendete Datentyp *Wave*, welcher innerhalb der Schnittstelle zum *ADAMA*-Simulationsframework zur Ein- und Ausgabe von Stimulifolgen verwendet wird, sieht eine zweiwertige Boolesche Algebra für die Signalwerte vor. Innerhalb des Datentyps werden die Signalwerte nicht explizit gespeichert, sondern implizit über einen im Kontext gespeicherten initialen Signalwert berechnet. Diese Berechnung wird in einer trivialen Form durchgeführt, da ausgehend vom initialen Wert eine Signaländerung in jedem Fall nur einen möglichen nachfolgenden Signalwert besitzt. Entweder findet ein Signalwechsel vom logischen Wert von 0 nach 1 statt, oder umgekehrt von 1 nach 0. Dies ist im Hinblick auf die einzusetzenden Architekturen samt dem Kontext der Speicherorganisation sehr effizient, da

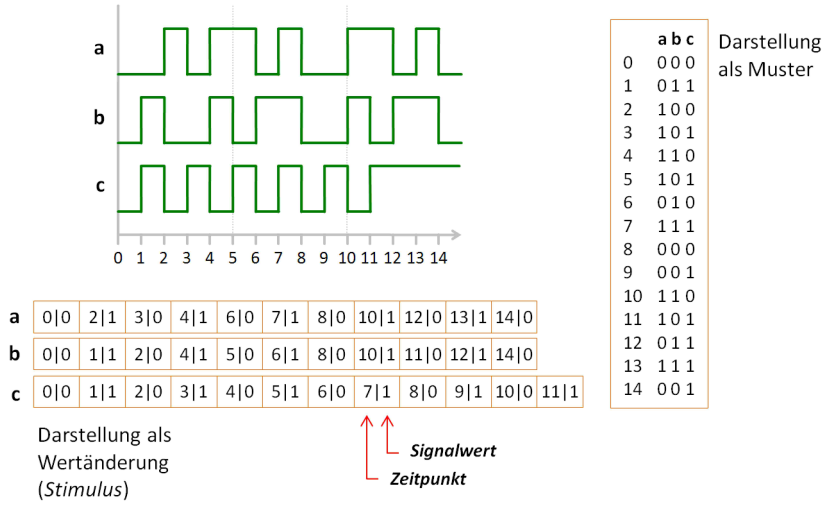


Abbildung 3.3: Vergleich der Darstellung zeitlicher Werteverläufe als Folge von Signaländerungen (Stimuli) und als fortlaufende Muster

die Auswertung der Stimulifolgen durch eine Iteration über eine Stimulifolge durchgeführt wird. Hierbei kann der Signalwert durch das triviale Invertieren des Signalwertes bestimmt werden, wodurch eine dazugehörige Speicheranfrage unnötig wird. Eine direkte Folge dieser Herangehensweise ist daher die Reduktion von Speicherzugriffen.

Während innerhalb der Schnittstelle zum Framework der Datentyp *Wave* als verkettete Liste implementiert wurde und damit Informationen zur Anzahl der Stimuli im Kontext der verketteten Liste bereitstehen, wurde in der datenparallelen *CUDA*-Implementierung stattdessen ein Feld-Datentyp (*engl. Array*) verwendet. Das Konzept zur Verwendung der einzelnen Elemente des Arrays ist in Abbildung 3.4 dargestellt. Das Ende einer Stimulifolge wird durch einen speziellen Wert an einem vordefinierten Zeitpunkt definiert (10^{38}), welcher in der praktischen Anwendung unwahrscheinlich erreichbar ist. Die Auswertung unterliegt der Annahme, dass der initiale Signalwert jeder beliebigen Stimulifolge mit 0 beginnt. Vor der Überführung der Stimulifolge aus der verketteten Liste in das Array wird im Falle des initialen Wertes 1 ein zusätzlicher Stimulus mit dem Wert -10^{35} eingefügt. Das Array enthält zusätzliche Elemente für Informationen, die während des Auswertungsvorganges erzeugt werden. Das erste Element des Arrays wird dazu genutzt, mögliche Überläufe bei der Auswertung von Stimuli an den Gattern zu dokumentieren, um so in einem weiteren Schritt eine Behandlung des Überlaufs durchführen zu können. Das zweite Element wird zur Erfassung der Aktivität des Signals verwendet. Hierzu wird die Anzahl der erzeugten Stimuli am Ausgang eines Gatters ermittelt, die bei der Auswertung der Stimulifolgen an den Eingängen des Gatters entstehen.

Die Ausgangsimplementierung ist fähig, eine Vielzahl von Simulationsinstanzen parallel auszuwerten. Innerhalb der Simulationsinstanzen wird die zuvor erläuterte Variation von Verzögerungszeiten an sämtlichen Gattern der Schaltung angewandt. Hierzu werden

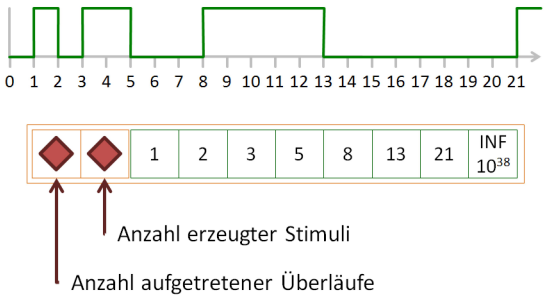


Abbildung 3.4: Darstellung von Stimulifolgen in der Ausgangsimplementierung

die Datenstrukturen der Stimulifolgen einer Schaltung, entsprechend der Anzahl der zu betrachtenden Simulationsinstanzen, vervielfältigt. Jede Instanz einer Stimulifolge wird einem Signal in einer Schaltungsinstanz zugeordnet.

Im datenparallelen Teil der grundlegenden Simulationsimplementierung werden sämtliche Instanzen einer Stimulifolge in einem gemeinsamen Array abgelegt. Durch die Bildung von Teilarrays werden die einzelnen Stimulifolgen eines Signals wiedergewonnen. Die Anordnung der einzelnen Stimuli innerhalb des gemeinsamen Arrays erfolgt durch Zusammenfassung von Stimuli gleicher Ordnung. Dieser Ansatz begründet sich in der gesteigerten räumlichen Kohärenz, welche bei der Auswertung der Stimulifolgen an den Gattern eine gesteigerte Effizienz erzielt. Die Ausgangsimplementierung führt die Auswertung der Stimulifolgen in einer parallelen Vorgehensweise aus. Die Parallelität der Ausführung führt zu gleichartigen Bearbeitungsfortschritten in den Stimulifolgen, so dass zu beliebigen Zeitpunkten innerhalb der Bearbeitung Stimuli gleicher Ordnung aus dem Speicher geladen werden. Eine Steigerung der räumlichen Kohärenz führt in diesem Kontext unmittelbar zur Steigerung der Effizienz der Implementierung.

Um innerhalb einer Stimulifolge die dazugehörigen Stimuli durchschreiten zu können, muss das Array ausgehend vom ersten Stimuli in einem konstanten Abstand durchschritten werden. Wie zuvor erwähnt, werden sämtliche Signale einer Schaltung im gemeinsamen Array vorgehalten. Um die Stimulifolge eines bestimmten Signals aufzufinden, kann das Array ebenfalls im konstanten Schritten durchquert werden. Die Schrittweite wird in den folgenden Algorithmen als *Signalschritt* bezeichnet. Das Konzept zur Anordnung von Stimulifolgen in einem gemeinsamen Array ist in Abbildung 3.5 dargestellt.

3.3.3 Darstellung von Gattern und Signalen

Schaltungen entstehen durch die Verbindung von einzelnen Gattern durch Signale. Die Repräsentation von Schaltungen in Form einer Datenstruktur orientiert sich in der Folge anhand eines gerichteten Graphen $G = (K, E)$. Dieser Graph enthält Knoten K, die den Gattern entsprechen, sowie Kanten E, welche die Signale repräsentieren, die jene Gatter verbinden. Zur Simulation des Verhaltens einer beliebigen Schaltung, wird die Struktur der

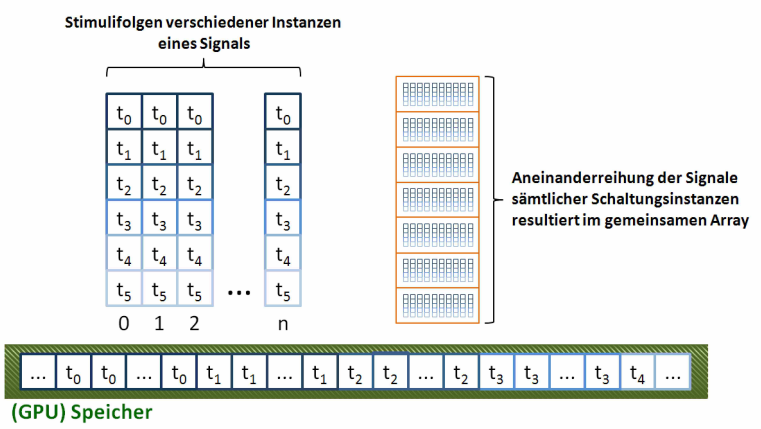


Abbildung 3.5: Darstellung der Speicherorganisation unter der Verwendung verschiedener Simulationsinstanzen in der Ausgangsimplementierung

Schaltung in einem vorbereitenden Schritt in die interne Datenstruktur überführt, aus dem jener Graph resultiert. Hiermit wird das Simulationsmodell der Schaltung beschrieben. Als Grundlage der Simulation wird die Schaltung topologisch sortiert. Die dabei entstehenden Ebenen werden im Rahmen der Simulation sequentiell ausgewertet.

Durch die topologische Sortierung wird die Ordnung zur Propagierung von Stimuli durch die Schaltung sichergestellt. Konflikte, die durch eine beliebige Anordnung der Gatter in der Datenstruktur entstehen können und in der Folge Abhängigkeiten verletzen, werden durch die topologische Sortierung aufgelöst, da die Bearbeitungsreihenfolge der topologischen Sortierung die Abhängigkeiten der Signalpropagierung zwischen den Gattern berücksichtigt. Diese Aufgaben werden durch Funktionalitäten des *ADAMA*-Simulationsframeworks vorgenommen.

Im Rahmen des Verfahrens zur topologischen Sortierung wird neben der Anordnung der Gatter-Informationen in topologischen Ebenen die Zuweisung von Speichereinheiten für die jeweiligen Signale durchgeführt. Im Rahmen der Ausgangsimplementierung wird die topologische Sortierung unter dem Ziel der Minimierung des benötigten Speichers für die Repräsentation der Signale durchgeführt. Hierzu wird die Annahme zugrunde gelegt, dass einerseits Signale bei einem Auswertungsvorgang vollständig ausgewertet werden und andererseits vollständig ausgewertete Signale nicht mehr benötigt werden. Diese können in der Folge überschrieben werden. In Abbildung 3.6 ist die Wiederbenutzung von Signalen im Rahmen der Auswertung der zugehörigen Stimuli anhand der Schaltung *C17* dargestellt.

Die Datenstruktur zur Darstellung der Gatter im Rahmen der grundlegenden Simulationsimplementierung orientiert sich am zuvor vorgestellten Simulationsmodell. Jedes Gatter verfügt unabhängig von der tatsächlichen Anzahl von Eingängen sowie Ausgängen, drei Signalindizes, mit denen zwei Eingangssignale sowie ein Ausgangssignal bestimmt werden. Die zugehörigen Signale definieren sich über die zuvor vorgestellten Stimulifolgen, die

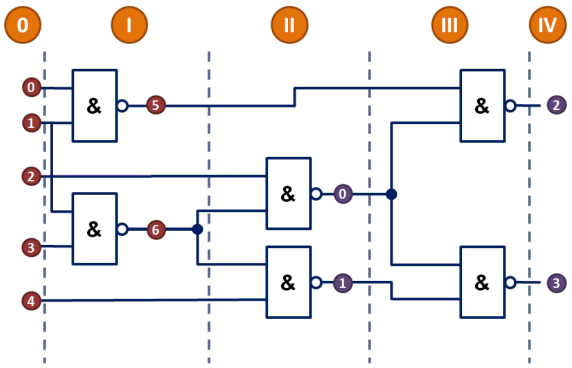


Abbildung 3.6: Schaltung C17 in Form der topologischen Sortierung

zusammengefasst in einem gemeinsamen Array vorliegen und durch das Durchschreiten in einem konstanten *Signalschritt*-Abstand wiedergefunden werden können. Im Rahmen der Auswertung der Stimulifolgen wird der Gattertyp benötigt, um die zugehörige Boolesche Funktion an den anliegenden Stimulifolgen anwenden zu können. Der Gattertyp wird in Form einer Ganzzahl explizit in der Datenstruktur vorgehalten.

Zur verzögerungsbehafteten Simulation der Schaltung, werden für jeden Eingang eine Reihe von Parametern individuell vorgehalten. Dies umfasst die Angabe der Verzögerungszeiten für die steigende und fallende Signalflanke, sowie ein Parameter, der den Einfluss der Varianz der errechneten Verzögerungszeit unter der Normalverteilung festlegt. Darüber hinaus wird für jeden Eingang das Trägheitsmodell in Form von zwei Schwellenwerten angegeben. Hierbei wird die Trägheit zwischen der steigenden und fallenden Signalflanke differenziert.

Gatter	
z, a, b	: int
op	: int
delay_a[2]	: int
variance_a[2]	: int
delay_b[2]	: int
variance_b[2]	: int
peak_threshold[4]	: int

Abbildung 3.7: Datenstrukur zur Repräsentation von Gattern

Die Menge der Gatter einer Schaltung wird fortlaufend in einem Array gespeichert. Die im Rahmen der topologischen Sortierung der Schaltung entstandene Anordnung der Gatter bleibt in diesem Zusammenhang erhalten. Die einzelnen Gatter einer topologischen Ebene können über einen Anfangsindex, sowie der Anzahl der Gatter jener Ebene bestimmt werden.

3.3.4 Auswertung von Stimulifolgen an beliebigen Gattern

Der Algorithmus zur Auswertung von Stimulifolgen an den Eingängen beliebiger Stimulifolgen ähnelt in der Ausgangsimplementierung Scan-Line-Algorithmen. Scan-Line-Algorithmen lösen geometrische Probleme, in dem sie eine sortierte Problemmenge durchqueren und im Falle des Auftretens von Elementen eine Teilantwort des Problems erzeugen [38].

Für die Lösung des Problems, auf welche Art die Auswertung von Stimulifolgen durchgeführt wird, wurde das Problem in Form eines geometrischen Problems abstrahiert. Diese Abstraktion ist in Abbildung 3.8 dargestellt. Innerhalb der Eingabe von Stimulifolgen liegen die einzelnen Stimuli in einer sortierten Reihenfolge vor. Die Sortiergrundlage ist der Zeitpunkt des Auftretens der einzelnen Stimuli innerhalb der jeweiligen Stimulifolgen.

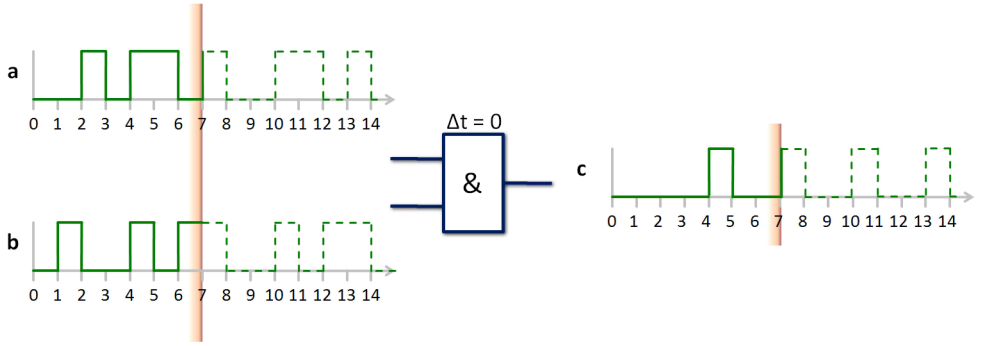


Abbildung 3.8: Veranschaulichung des Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern im Kontext der Abstraktion zu einem geometrischen Problem

Der Ansatz zur Lösung des Problems besteht darin, die Stimulifolgen an den Eingängen der Gatter anhand von fortlaufenden Zeitpunkten gleichzeitig zu durchqueren und im Falle eines Auftretens von Signalübergängen eine Behandlung dieses Überganges durchzuführen. Die Behandlung besteht darin, anhand der gegebenen vorherigen und nachfolgenden Signalwerte im Kontext der Booleschen Funktion des Gatters festzustellen, wie sich das Signal am Ausgang der Schaltung zu gegebenem Zeitpunkt verhält. Falls das Signal am Ausgang des Gatters nach diesem Auswertungsvorgang eine Änderung erfährt, wird ein neuer Stimulus am Ende der Stimulifolge eingefügt.

Die Effizienz der gesamten Simulationsimplementierung hängt signifikant vom eigentlichen Laufzeitverhalten des Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern ab (siehe Algorithmus 1), da dieser Algorithmus ein Kernbestandteil der Ausgangsimplementierung ist. Das Laufzeitverhalten des Algorithmus ist von einem linearen Komplexität geprägt.

Das lineare Laufzeitverhalten resultiert aus der Implementierung der einzelnen Bestandteile des Algorithmus. Das Durchschreiten der Stimulifolgen entspricht der Iteration über eine

Algorithmus 1 Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern

```
OP = Boolesche Funktion des Gatters
STIMULICAP = Maximale Anzahl an Stimuli in einer Stimulifolge
function AUSWERTUNG(a, b, z)
    i ← 0;
    j ← 0;
    k ← 0;
    letzte_transition ← −1;
    while k < STIMULICAP ∧ (i < STIMULICAP ∨ j < STIMULICAP) do
        m ← min(a[i].zeit, b[j].zeit);
        if a[i].zeit < b[j].zeit then
            i ← i + 1;
        else
            j ← j + 1;
        end if
        if letzte_transition ≠ a[i].wert OP b[j].wert then
            z[k] ← a[i].wert OP b[j].wert
            k ← k + 1;
        end if
    end while
end function
```

endliche Liste der Länge n. Da zwei Stimulifolgen betrachtet werden, umfasst das größte anzunehmende Ausmaß beider Stimulifolgen $2n$. Die Berücksichtigung der vorherigen und nachfolgenden Signalwerte werden im Kontext des Durchschreitens der Stimulifolgen gelesen und stehen daher ohne einen weiteren Aufwand für die Auswertungsaufgabe zur Verfügung. Die eigentliche Auswertungsaufgabe anhand der Booleschen Funktion wird in einer konstanten Zeit $\mathcal{O}(k)$ durchgeführt. Das Anhängen eines weiteren Stimulus an die Stimulifolge am Ausgang eines beliebigen Gatters erfolgt während des Durchschreitens der Liste. Die größte anzunehmende Zahl an Auswertungsvorgängen in Kombination mit der Erzeugung von neu einzufügenden Stimuli, entsteht dann, wenn jeder gelesene Stimulus einen Signalwechsel am Ausgang des betrachteten Gatters erzeugt. Ein Beispiel hierfür ist ein Szenario, in dem ein Exklusiv-Oder-Gatter definiert ist, sowie zwei Stimulifolgen, die abwechselnd ihren Signalwert ändern. Dieses Szenario ist in Abbildung 3.9 dargestellt.

In der Analyse des Laufzeitverhaltens bleibt somit die Durchschreitung der Stimulifolgen ($\mathcal{O}(2 * n)$) sowie die Auswertungsdurchführung ($\mathcal{O}(k)$) zu berücksichtigen. Diese einzelnen Bestandteile sind durch eine lineare Laufzeit geprägt. Das Laufzeitverhalten beträgt somit $\mathcal{O}(2 * n + n * k)$. Als Resultat ergibt sich somit in der Gesamtbetrachtung ein lineares Laufzeitverhalten.

3.3.5 Implementierungsdetails zur Auswertung von Stimuli an beliebigen Gattern

Innerhalb der Realisierung des zuvor vorgestellten Auswertungsalgorithmus beinhaltet die Ausgangsimplementierung weitergehende Schritte. Die Implementierung verfügt über eine

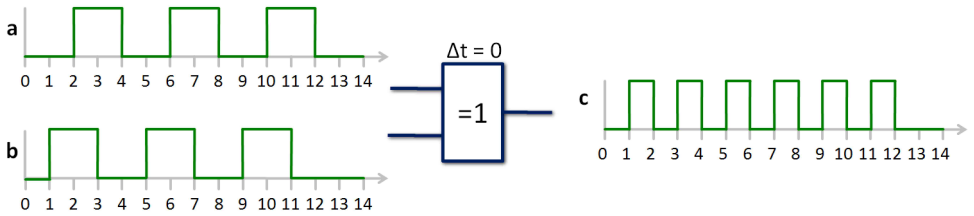


Abbildung 3.9: Beispielszenario, unter dem bei jedem Signalwechsel an den Eingängen ein Signalwechsel an den Ausgängen entsteht

Initialisierungsphase, die nicht im vorgestellten Algorithmus 1 dargestellt ist. In der Anwendung innerhalb der datenparallelen *CUDA*-Architektur wird die Initialisierung parallel sowohl auf der Menge der Gatter einer topologischen Ebene als auch auf den der Menge der verschieden Instanzen der jeweiligen Gatter ausgeführt. In der Initialisierungsphase werden die Kontextinformationen für eine Kombination aus Gatter und Simulationsinstanz geladen. Diese Kontextinformationen, aus denen die Signalinformationen über Eingänge und Ausgänge des betrachteten Gatters hervorgehen, bestimmen die Position der zu verarbeitenden Stimulifolgen innerhalb des Arrays, in welcher die Stimulifolgen zusammengefasst abgelegt sind. Wie bereits erläutert, werden in diesem Array sämtliche Stimulifolgen der einzelnen Signale in konstantem Abstand zu einander vorgehalten. Einzelne Instanzen von Stimulifolgen verschiedener Schaltungsinstanzen liegen fortlaufend im Array vor.

Anschließend wird die Bestimmung der initialen Signalwerte, sowie die Prüfung des ersten erzeugten Signalwertes durchgeführt. Wie bereits erläutert, werden die Signalwerte der einzelnen Stimuli innerhalb von Stimulifolgen ausgehend vom Signalwert 0 implizit bestimmt. Führt jedoch die Auswertung zweier Signalwerte mit dem Wert 0 zu einer Signaländerung zum Signalwert 1 wird in diesem Schritt ein zusätzliches Stimuli eingefügt. Hiermit werden Stimulifolgen berücksichtigt, die abweichend mit dem Signalwert 1 beginnen. Die Initialisierungsphase wird mit der Bestimmung der zeitlichen Werte der initialen Stimuli abgeschlossen. Im Falle von Gattern, die zwei Eingänge besitzen, werden die jeweiligen ersten Stimuli gelesen, wobei deren zeitliche Werte mit der im Verzögerungsmodell angegebenen Verzögerung addiert werden. Gatter, die nur über einen Eingang verfügen, werden entsprechend nur am ersten Eingang ausgewertet.

In Algorithmus 2 ist die Initialisierungsphase dargestellt. Aus Gründen der besseren Übersicht wurde die Ermittlung der Überläufe und Anzahl von Stimuli außen vor gelassen.

Algorithmus 3 stellt den Auswertungsteil des Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern vor, wie er in der Ausgangsimplementierung realisiert wurde. Um das Ende einer Stimulifolge zu kennzeichnen, wird der letzte Stimulus mit dem speziellen Zeitwert 10^{38} gekennzeichnet. Innerhalb der Implementierung wird die Iteration fortgesetzt, bis beide Stimulifolgen vollständig durchquert wurden. Diese Bedingung wird erfüllt, wenn die jeweiligen letzten Stimuli gelesen wurden und somit der Zeitwert 10^{38} als kleinster Zeitwert der beiden zuletzt gelesenen Stimuli auftritt.

Algorithmus 2 Initialisierungsphase des Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern

```
DELAY = Verzögerung eines Gatters
STIMULICAP = Maximale Anzahl an Stimuli in einer Stimulifolge

/* Hinweis: Die Bestimmung des aktuellen Gatter, den dazugehörigen Signalen, sowie
der aktuellen Simulationsinstanz ist abhängig von der eingesetzten Architektur */
Gatter ← GatterListe[GatterIdx];
slot ← aktuelle Simulationsinstanz;

/* Bestimmung der Ein- und Ausgänge des Gatters */
a ← Gatter.a ∗ signalschritt + slot;
b ← Gatter.b ∗ signalschritt + slot;
z ← Gatter.z ∗ signalschritt + slot;

/* Bestimmung der Signalwerte */
a_wert ← 0;
b_wert ← 0;
z_wert ← (0 gate.op 0);

/* Falls die Stimulifolge abweichend mit 1 beginnt,... */
if z_wert = 1 then
    /* ...füge zusätzliches Stimuli mit Zeitpunkt −10^35 ein */
    waves[z] ← −10^35;
    Schiebe z in der Stimulifolge um eine Stelle
end if

/* Bestimmung der Zeitpunkte */
a_zeit ← waves[a] + DELAY;
if gate.op ≠ NOT ∧ gate.op ≠ BUF then
    b_zeit ← waves[b] + DELAY;
end if
```

Die Bestimmung der jeweiligen Signalwerte an einem Zeitpunkt wird implizit, durch die Invertierung des Signalwertes des zuletzt betrachteten Stimuli, durchgeführt. Um die Trägheit innerhalb von Schaltungen (*Inertial Delay*) zur berücksichtigen, werden eingetragene Stimuli nachträglich entfernt, falls ein nachfolgender Stimulus einen zu geringen zeitlichen Abstand zum eingetragenen Stimulus aufweist. Zur Bestimmung dieses zeitlichen Abstandes wird der Zeitwert des zuletzt erzeugten Stimulus bei jedem Erzeugungsvorgang gespeichert. Aus der Differenz des Zeitwertes des zu erzeugenden Stimuli zum gespeicherten Zeitwertes wird die Bedingung zur Überwindung der Trägheit der Schaltung geprüft. Im Fall der Nichtüberwindung der Trägheit wird die Überschreibung des ersten beteiligten Stimulus vorbereitet. Hierzu wird der Index der Einfügeposition um eine Stelle zurückgesetzt. Zusätzlich wird die Überschreibung in der nächsten Iteration explizit sichergestellt, um die Korrektheit der Auswertung zu gewährleisten. Dies begründet sich durch die Beteiligung von genau zwei Stimuli an einem Signalausschlag, der unter der Trägheit der

Algorithmus 3 Auswertungsphase des Algorithmus zur Auswertung von Stimulifolgen an beliebigen Gattern

```
DELAY = Verzögerung eines Gatters
STIMULICAP = Maximale Anzahl an Stimuli in einer Stimulifolge
FILTER_SCHWELLE = Zeitraum, den Signale stabil überdauern müssen, um eine Propagierung auszulösen (Trägheitsschwelle)
/* Auswertung der Stimulifolgen */
m ← min(a_zeit, b_zeit);
while z < STIMULICAP ∧ m ≠ 10^38 do
    if b_zeit < a_zeit then
        Schiebe b in der Stimulifolge
        Invertiere b_wert
        b_zeit ← waves[b] + DELAY;
    else
        Schiebe a in der Stimulifolge
        Invertiere a_wert
        a_zeit ← waves[a] + DELAY;
    end if
/* Falls mit dem neuen Signalwert eine Änderung des Signalwertes einhergeht... */
    if z_wert ≠ (a_wert gate.op b_wert) then
/* Falls das aktuelle Signal die Trägheit der Schaltung überdauert... */
        if letzte_transition = −500 ∨ (m − letzte_transition) ≥FILTER_SCHWELLE then
            /* ... wird der Signalwechsel dokumentiert */
            waves[z] ← m;
            Schiebe z in der Stimulifolge um eine Stelle
        else
            letzte_transition ← −500;
            Schiebe z in der Stimulifolge um eine Stelle zurück
        end if
    end if
end while
```

Schaltung nicht propagiert wird.

3.3.6 Durchführung von Simulationen

Die Durchführung einer Simulation in der Ausgangsimplementierung umfasst neben der eigentlichen Auswertung von Stimulifolgen an Gattern vorbereitende sowie nachbereitende Schritte. Die grundlegende Simulationsimplementierung bietet Funktionalitäten an, mit denen eine Überführung von Eingabemustern in Stimulifolgen durchgeführt werden kann. Hierzu werden zwei unterschiedliche Herangehensweisen angeboten. Zum einen kann eine Überführung auf Seiten des *ADAMA*-Frameworks im Kontext der CPU-Architektur verfolgt werden sowie zum anderen im Kontext der datenparallelen GPGPU-Architektur. Der Unterschied der beiden Herangehensweisen bestimmt sich im Grad der Parallelität. Während die Überführung der Muster für jeden Eingang der Schaltung sowie für jede

Schaltungsinstanz im Kontext des ADAMA-Frameworks sequentiell durchgeführt wird, führt die Verwendung der datenparallelen Funktionalitäten zu einer für jeden Eingang sowie Schaltungsinstanz parallel ausgeführten Überführung. Darüber hinaus ist es möglich, diesen Überführungsschritt außen vor zu lassen und statt dessen explizit Stimulifolgen zu erzeugen und an die zu simulierende Schaltung anzulegen.

Bevor die eigentliche Simulation durchgeführt wird, wird die Initialisierung der Simulationsumgebung ausgeführt, sofern dies nicht im Rahmen der Überführung von Mustern zu Stimulifolgen geschehen ist. Hierzu zählen die Allokation von Speicher, sowie die Bestimmung der Größen sämtlicher Datenstrukturen. In der datenparallelen Ausführung wird an dieser Stelle die Anwendung zur Kommunikation mit dem *CUDA*-Applikationsframework als Unterprozess ausgeführt, sowie Kommunikationspfade und gemeinsamer Speicher erzeugt und initialisiert. Darüber hinaus wird die Initialisierung dazu genutzt, die Informationen über die Gatter, sowie die dazugehörigen Kontextinformationen über das Verzögerungsmodell bereitzustellen.

Nach dem Abschluss der Initialisierung wird nun die eigentliche Simulation der Schaltung unter den angelegten Eingaben durch eine Iteration über die topologischen Ebenen der Schaltung durchgeführt. Die angelegten Eingaben, die in Form von Stimulifolgen dargestellt sind, werden an den Gattern einer topologischen Ebene entsprechend Algorithmus 2 und 3 ausgewertet. Im Rahmen der Auswertung der Stimulifolgen werden eventuell auftretende Überläufe in den Stimulifolgen an den Ausgängen der Gatter geprüft und dokumentiert.

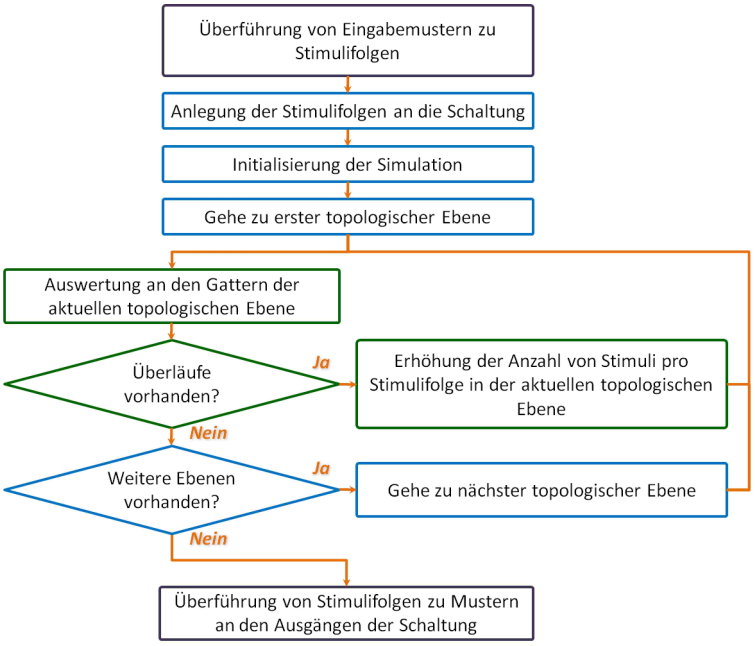


Abbildung 3.10: Ablauf der Durchführung einer Simulation in der Ausgangsimplementierung

Im Falle auftretender Überläufe innerhalb einer topologischen Ebene werden die Datenstrukturen der Stimulifolgen an den Ausgängen der Gatter jener topologischen Ebene vergrößert. Anschließend wird die Auswertung für jene topologische Ebene von Grund auf wiederholt, wobei zuvor erzeugte Stimuli überschrieben werden. Dieser Vorgang wird für jede topologische Ebene im Einzelnen solange fortgesetzt, bis sämtliche Überläufe berücksichtigt wurden. Resultierend werden somit sämtliche Stimuli innerhalb der Stimulifolgen an den Eingängen der Gatter ausgewertet. Sobald sämtliche Überläufe behandelt wurden, wird die Auswertung an der nachfolgenden topologischen Ebene fortgesetzt. Dieser Vorgang wird weitergeführt, bis alle Stimulifolgen innerhalb der topologischen Ebenen auswertet wurden.

Den Abschluss der Simulation bildet die Überführung der Stimulifolgen zu Mustern. Im Rahmen dieses Schrittes werden die Stimulifolgen an den Ausgängen der Schaltung an definierten Zeitpunkten zu Mustern übersetzt. Die Wahl dieser Zeitpunkte wird nicht von der Simulationsimplementierung bestimmt, sondern ist vom Benutzer vorgegeben. Der abschließende Schritt zur Überführung von Stimulifolgen zu Mustern kann im Rahmen der Ausführung einer Stimulation außen vor gelassen werden. Stattdessen können die Stimulifolgen von den Ausgängen der Schaltung direkt gelesen werden. Der vorgestellte Simulationsablauf ist in Abbildung 3.10 dargestellt.

3.4 Zusammenfassung der analysierten Beschränkungen

Im Folgenden werden die Beschränkungen der Ausgangsimplementierung zusammengefasst. Zu jeder erfassten Beschränkung werden Lösungsansätze erläutert, deren Realisierung im Kapitel 4 detailliert vorgestellt werden.

Datenstruktur für kontinuierliche Stimulifolgen Innerhalb der Ausgangsimplementierung begrenzt die Kapazität des Speichers die Menge der Eingaben, welche in einer Simulation verwendet werden können. Die Verarbeitung von Eingaben der Ausgangsimplementierung führt zur Konsequenz, dass sämtliche Eingaben zu Beginn der Simulation im Speicher vorhanden sein müssen. Eine fortlaufende Simulation von Eingabemengen, die diese Kapazität überschreiten, ist ausschließlich durch eine fortlaufende Wiederholung der gesamten Simulation unter der Anwendung weiterer Eingaben möglich. Wie zuvor erläutert, führt diese Beschränkung zu einer nicht effizienten Lösung. Zur Steigerung der Effizienz dieser Prognosen ist es erforderlich neben einer Vielzahl von Simulationen unter verschiedenen Variation die Simulation beliebig dimensionierter Simulationszeiträume durchführen zu können. Für die weiterentwickelte Simulationsimplementierung leitet sich hieraus der Anspruch ab, Eingaben kontinuierlich verarbeiten zu können. Die Funktionalitäten, welche die Stimuli an die Eingänge der Schaltung anlegen, müssen die Fähigkeit besitzen, im laufenden Simulationsprozess weitere Stimuli verarbeiten zu können Hierbei muss die Möglichkeit ausgeschlossen werden, bestehende Stimulifolgen an den Eingängen der Schaltung überschreiben zu können. Als Folge würde das Simulationsergebnis falsifiziert werden.

Effiziente Behandlung von übergelaufenen Stimuli Die Ausgangsimplementierung begegnet Überläufen in drei Schritten. Im Zuge der Auswertung von Stimulifolgen werden sämtliche Stimulifolgen auf das Vorhandensein von Überläufen innerhalb einer topologischen Ebene überprüft. In einem zweiten Schritt werden Erweiterungen von Stimulifolgen der betroffenen Gatter durchgeführt. Anschließend werden in einem dritten Schritt Neuauswertungen der Stimulifolgen an sämtlichen Gattern jener topologischen Ebene durchgeführt. Sämtliche Schritte werden wiederholt, bis die Stimulifolgen die benötigte Dimensionierung erreicht haben. Im Rahmen der Auswertung von Stimulifolgen an beliebigen Gattern sind Überläufe ein fortwährend auftretendes Ereignis, falls an den Eingängen der Schaltung kontinuierlich Eingaben hinzugefügt werden. Die Behandlung dieses Ereignisses erfordert eine effiziente Herangehensweise. Die Ausgangsimplementierung führt im Falle der kontinuierlichen Neuanlage von Eingaben zu einer fortlaufenden wiederholten Neuauswertung der Stimulifolgen. Dies führt dazu, dass erzeugte Stimulifolgen aus vergangenen Simulationsschritten ebenfalls neu ausgewertet werden. Für die weiterentwickelte Simulationsimplementierung leitet sich hieraus die Anforderung ab, eine wiederholte Auswertung von Stimuli an beliebigen Gattern zu vermeiden, um die Effizienz zu steigern.

Korrekte Auswertung der finalen Stimuli Die Ausgangsimplementierung geht innerhalb der Auswertung von Stimulifolgen an beliebigen Gattern von der Annahme aus, dass der jeweilig letzte Stimulus einer Stimulifolge den Signalwert für den gesamten restlichen betrachteten Zeitraum der Auswertung bestimmt. In einer kontinuierlichen Simulation ist diese Annahme nicht korrekt. Das Anlegen weiterer Stimulifolgen kann dazu führen, dass eine Stimulifolge an einem Eingang eines beliebigen Gatters um weitere Stimuli ergänzt wird, die den Signalwert im weiteren zeitlichen Verlauf verändern. Stimuli, die unter der Annahme eines stabilen Signalwertes erzeugt wurden, könnten unter falschen Voraussetzungen erzeugt worden und damit nicht korrekt sein. Für CWTSim leitet sich hieraus die Anforderung ab, die Auswertung vorzeitig zu unterbrechen, falls im Falle des Vorhandensein zweier Stimulifolgen, eine Stimulifolge Stimuli besitzt, zu denen in der zweiten Stimulifolge keine zeitlichen Informationen existieren. Darüber hinaus leitet sich die Anforderung ab, dass die Auswertungsalgorithmen der weiterentwickelten Simulationsimplementierung die Fähigkeit besitzen, unterbrochene Bearbeitungsfortschritte wieder aufnehmen zu können. Im Kontext beliebig groß dimensionierter Stimulifolgen besteht der Anspruch, die Wiederaufnahme der Auswertung effizient und daher unabhängig von der Dimensionierung von Stimulifolgen zu realisieren.

Trägheiten im Falle kontinuierlicher Stimuli Die Ausgangsimplementierung propagiert an den Gattern erzeugte Stimulifolgen vollständig durch die topologischen Ebenen. Im Rahmen einer kontinuierlichen Simulation kann diese Herangehensweise im Kontext der Trägheitskriterien (*Inertial Delay*) der Schaltung zu falschen Simulationsergebnissen führen. Wird im kontinuierlichen Fall die gesamte Stimulifolge propagiert, kann es vorkommen, dass der letzte propagierte Stimulus in Kombination mit einem nachfolgenden eingefügten Stimulus dazu führt, dass die beiden Stimuli die

Trägheitsbedingung verletzten und entfernt werden müssen. Der bereits propagierte Stimulus kann im Verlauf der topologischen Ebenen zu weiteren erzeugten Stimuli führen, die ebenfalls behandelt werden müssen. Daraus folgt die Anforderung an CWTSim dieses Ereignis zu berücksichtigen, um korrekte Simulationsergebnisse zu erzeugen.

Höherwertige Boolesche Algebra Die Ausgangsimplementierung verwendet im Rahmen der Darstellung und Auswertung einer Stimulifolge eine zweiwertige Boolesche Algebra. Eine zweiwertige Boolesche Algebra ist jedoch nicht fähig, Zustände zu simulieren, die in sequentiellen Schaltungen sowie in Bussystemen auftreten. Für die Erweiterung des Simulationsmodells soll eine vierwertige Booleschen Algebra verwendet werden.

Speicherverwaltung In der Ausgangsimplementierung findet eine Wiederverwendung von Speichereinheiten innerhalb der Repräsentation von Signalen statt. Diese Wiederverwendung wird im Rahmen der topologischen Sortierung des Schaltungsgraphen festgelegt. In einer kontinuierlichen Simulation kann diese Bedingung im Rahmen der topologischen Sortierung nicht angenommen werden, da unvollständige Bearbeitungsfortschritte innerhalb der Stimulifolgen während des Simulationsprozesses ausführlich berücksichtigt werden müssen. Die topologische Sortierung, welche innerhalb von CWTSim angewandt wird, darf daher keine Wiederverwendung von Signalen im Kontext von Speichereinheiten durchführen.

4 Realisierung und Implementierung der kontinuierlichen Simulation - CWTSim

4.1 Einführung

Die Simulationsimplementierung, welche durch die Realisierung des vorgestellten kontinuierlichen Simulationsmodells entstand, wird in dieser Arbeit mit der Bezeichnung *CWTSim* referenziert (*abk. Continuous Wave Timing Simulation - verzögerungsbehaftete Simulation kontinuierlicher Stimulifolgen*). Die Implementierung besteht aus einer Menge von Datenstrukturen und Algorithmen, welche im folgenden Kapitel detailliert vorgestellt werden. *CWTSim* beinhaltet eine Schnittstelle zu *ADAMA*, womit Befehle und Eingaben an die Implementierung von *CWTSim* übergeben werden. Darüber hinaus wurde *CWTSim* auf die datenparallele *CUDA*-Architektur abgebildet. Die Entwicklungen, welche in den folgenden Kapiteln vorgestellt werden, werden im Kontext dieser *datenparallelen Implementierung* erläutert. Zur Validierung dieser datenparallelen Implementierung wurde eine Referenzimplementierung entwickelt, welche eine Abbildung der datenparallelen Implementierung in die Programmiersprache Java darstellt. Diese führt die Algorithmen der datenparallelen Implementierung sequentiell aus.

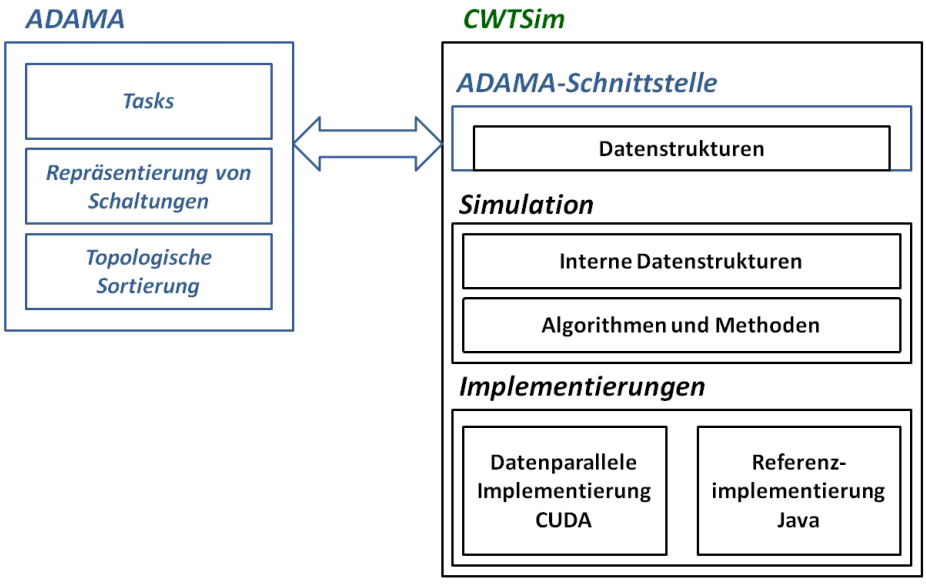


Abbildung 4.1: Übersicht zu CWTSim

4.1.1 Schnittstelle zu ADAMA

In Abbildung 4.2 sind die Datentypen dargestellt, welche als Schnittstelle zum ADAMA-Framework entwickelt wurden. Diese Datentypen umfassen *MVFloat (Multi-Valued Float)*, *MVWave* und *MVWaveBundle*. Der Datentyp *MVFloat* dient zur effizienten Repräsentation eines Stimulus. Dieser Datentyp wird in Kapitel 4.2.1 erläutert. Der Datentyp *MVWave* repräsentiert eine Stimulifolge. Eine Stimulifolge beinhaltet eine nach der Zeit aufsteigend sortierte Menge von Stimuli. Dieser Datentyp wird in Kapitel 4.2.2 erläutert. Eine Menge von Stimulifolgen, wird zu einem so genannten *Signalbund* zusammengefasst. Eine Simulationsinstanz repräsentiert eine dieser Variationen. Der Datentyp *MVWaveBundle* beinhaltet sämtliche Stimulifolgen eines *Signalbundes*. Dieser Datentyp wird in Kapitel 4.2.4 vorgestellt. Die einzelnen Funktionen des Task *CWTSim*, der die algorithmische Schnittstelle zum ADAMA-Framework darstellt, werden in Kapitel 8.1 dargestellt.

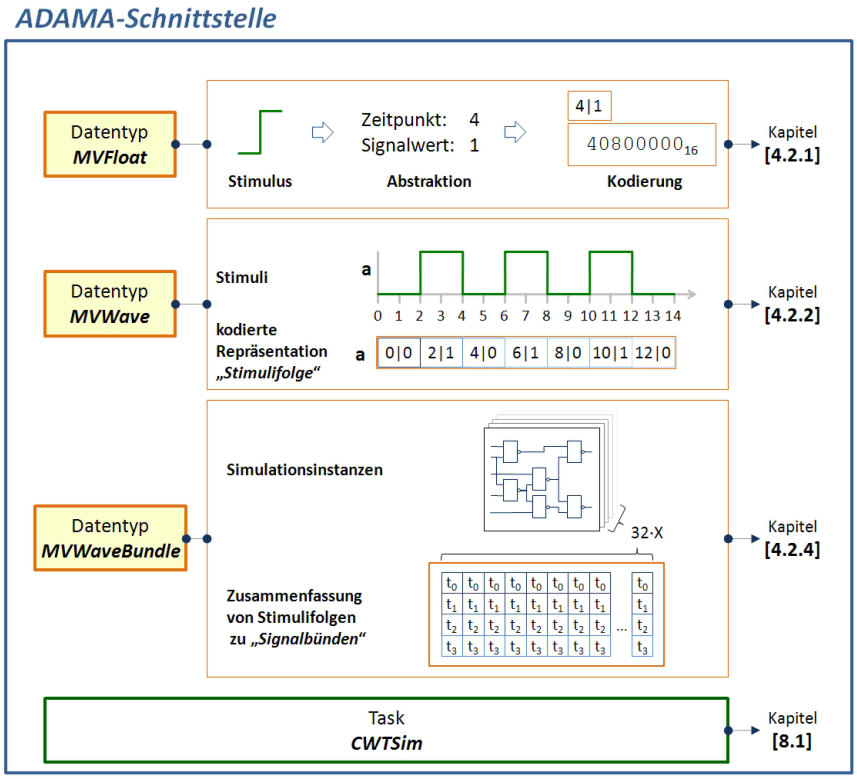


Abbildung 4.2: Übersicht der Datentypen, welche in der Schnittstelle zu *ADAMA* eingesetzt werden

4.1.2 Übersicht zu internen Datenstrukturen

Innerhalb einer Simulation werden eine Reihe von Datenstrukturen verwendet, welche in Abbildung 4.3 dargestellt sind. Diese Datenstrukturen sind im Einzelnen das *Stimuli-Array*, die *Signalbundkarte* sowie die *Signalkarte*. Das *Stimuli-Array* umfasst eine Teilmenge der Stimuli innerhalb einer Simulation. Diese Teilmenge beinhaltet ausschließlich Stimuli, die zu einem gegebenen Simulationszeitpunkt verwendet werden. Diese Datenstruktur wird in Kapitel 4.2.3 detailliert vorgestellt. Die *Signalbundkarte* dient zur Verkettung einzelner Signalbünde. Die einzelnen Stimuli eines Signales werden fortlaufend über mehrere endliche Stimulifolgen hinweg gespeichert. Die Verkettung der Stimulifolgen erfolgt über die Verkettung der Signalbünde, in denen sich die Stimulifolgen befinden. Innerhalb der *Signalbundkarte* werden sämtliche Verkettungen festgehalten. Diese Datenstruktur wird in Kapitel 4.2.4 erläutert. Die *Signalkarte* repräsentiert die einzelnen Signale innerhalb einer Schaltung. Sie dokumentiert Informationen zu verwendeten Signalbünden sowie Kontextinformationen zu den Auswertungsfortschritten innerhalb der verwendeten Stimulifolgen. Diese Datenstruktur wird in Kapitel 4.2.5 erläutert.

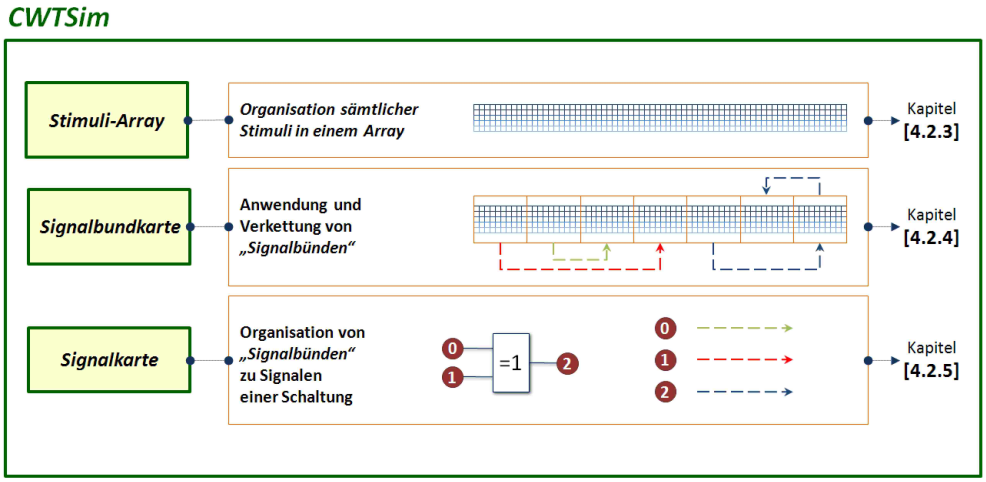


Abbildung 4.3: Übersicht zu Datenstrukturen, die innerhalb einer Simulation verwendet werden

4.1.3 Übersicht zum Ablauf einer kontinuierlichen Simulation

Der Ablauf einer kontinuierlichen Simulation unterteilt sich in eine Reihe von einzelnen Operationen. Die Gesamtübersicht der einzelnen Operationen ist in Abbildung 4.4 dargestellt. Vor dem Beginn der Simulation werden die zuvor erläuterten Datenstrukturen initialisiert. Dies wird in Kapitel 4.3.2 vorgestellt. Anschließend wird die kontinuierliche Simulation eingeleitet, die fortlaufend Eingaben in die internen Datenstrukturen hinzufügt und auswertet. Das Hinzufügen von Eingaben innerhalb der kontinuierlichen Simulation wird in Kapitel 4.3.1 erläutert. Die einhergehenden Operationen auf den Datenstrukturen, welche durch das Hinzufügen von neuen Daten erforderlich sind, werden in Kapitel 4.2.6 erläutert. Die Auswertung von Stimuli an den einzelnen Gattern innerhalb einer Simulation wird in Kapitel 4.3.3 vorgestellt. Nach dem Abschluss der Auswertung können die Ergebnisse aus den internen Datenstrukturen der Simulation in die Datenstrukturen von ADAMA überführt werden. Die einhergehenden Operationen werden in Kapitel 4.3.4 vorgestellt. Nach der Simulation sämtlicher Eingaben ist der Abschluss der Simulation erforderlich. Die kontinuierliche Simulation geht in ihren Ansätzen fortwährend von einem Zwischenstand in der Simulation aus. Der Abschluss einer Simulation muss daher gesondert betrachtet werden. Die einzelnen Operationen, die den Abschluss einer vollständigen Simulation gewährleisten, werden in Kapitel 4.3.5 behandelt.

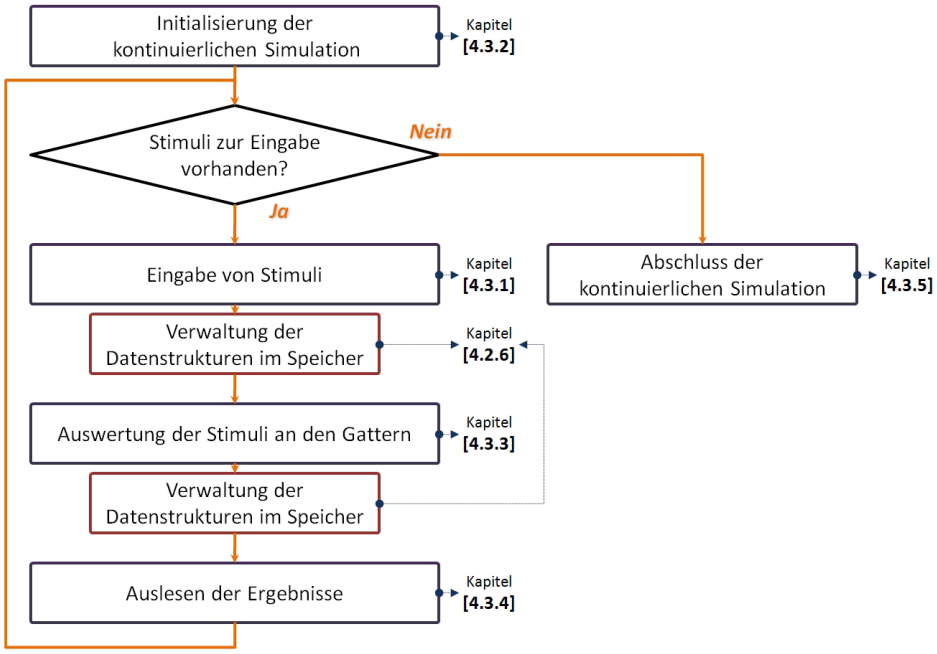


Abbildung 4.4: Übersicht zum Ablauf der einzelnen Operationen innerhalb einer kontinuierlichen Simulation

4.2 Datenstrukturen von CWTSim

4.2.1 Datentyp zur Darstellung eines mehrwertigen Stimulus

Innerhalb einer Simulation werden Signalwechsel durch Stimuli gekennzeichnet. Das Simulationsmodell definiert einen Stimulus als einen Signalwechsel, der an einem eindeutigen Zeitpunkt auftritt. Wie zuvor erläutert, betrachtet das Simulationsmodell der Ausgangsimplementierung eine zweiwertige Boolesche Algebra. Darauf basierend, berechnet der Auswertungsalgorithmus den Signalwert eines Stimulus implizit. Ausgehend von einem initialen Signalwert wird der gegenwärtige Signalwert bestimmt, in dem der Signalwert zwischen 0 und 1 abgewechselt wird.

CWTSim betrachtet eine vierwertige Boolesche Algebra. Darin kann ein Signalwechsel zu einem von drei Signalwerten führen. Bei der Verwendung von mindestens drei Signalwerten innerhalb einer Booleschen Algebra ist eine implizite Berechnung des Signalwertes nicht möglich. Dieser Zusammenhang ist in Abbildung 4.5 dargestellt. Während jeder Zustand innerhalb einer zweiwertigen Booleschen Algebra genau einen definierten Nachfolgezustand besitzt, besitzt ein Zustand in einer dreiwertigen Boolschen Algebra zwei Nachfolgezustände. Um diesem Nichtdeterminismus zu begegnen, ist es erforderlich, den nachfolgenden Signalwert an jedem Signalwechsel explizit anzugeben.

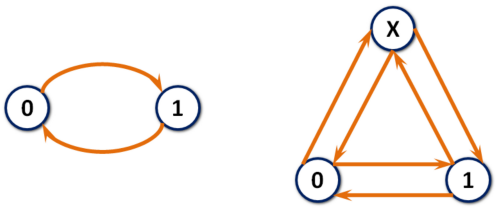


Abbildung 4.5: Vergleich der Zustandsdiagramme einer zweiwertigen Booleschen Algebra und einer dreiwertigen Booleschen Algebra

Der Datentyp, welcher in der Ausgangsimplementierung die Repräsentation von Stimuli durchführt, besitzt keine Funktionalitäten zur expliziten Speicherung von Signalwerten. Daher kann dieser Datentyp in CWTSim nicht übernommen werden. Für die Realisierung von CWTSim wurden zwei Konzepte für einen Datentyp entwickelt. Die Konzepte werden im Folgenden vorgestellt. Darüber hinaus wird die Analyse zu den Eigenschaften der Konzepte erläutert. Grundlage beider Konzepte ist das Ziel, die vierwertige Boolesche Algebra vollständig zu repräsentieren. Das Zustandsdiagramm zu dieser Algebra ist in Abbildung 4.6 dargestellt.

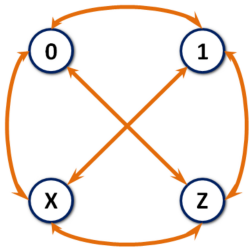


Abbildung 4.6: Zustandsdiagramm einer vierwertigen Booleschen Algebra

Das erste Konzept sieht vor, den Datentyp als Tupel zweier Datentypen zu implementieren. Hierzu wird der Zeitpunkt als Gleitkommazahl einfacher Genauigkeit nach dem IEEE754-Standard gespeichert. Der Signalwert wird in Form einer positiven Ganzzahl repräsentiert. Das Lesen sowie Schreiben von Stimuliwerten aus dem Speicher ist in Algorithmus 4 dargestellt.

Algorithmus 4 Lesen und Schreiben von Stimuli nach dem ersten Konzept

```
function LESESTIMULI(idx)
    StimuliWert ← stimuli[2 * idx];
    StimuliZeit ← stimuli[2 * idx + 1];
end function
function SCHREIBESTIMULI(idx)
    stimuli[2 * idx] ← StimuliWert;
    stimuli[2 * idx + 1] ← StimuliZeit;
end function
```

Das zweite Konzept benutzt ebenfalls eine Gleitkommazahl einfacher Genauigkeit für die Repräsentation für den Zeitpunkt. Der Unterschied besteht darin, dass der Signalwert in die Gleitkommazahl codiert wird. Die beiden höchstwertigen Bits, die das Vorzeichen und das höchstwertige Bit des Exponenten darstellen, werden durch die Codierung des Signalwertes überschrieben. Somit können vier diskrete Werte dargestellt werden. Die Lese- und Schreibalgorithmen des zweiten Konzeptes sind in Algorithmus 5 dargestellt.

Algorithmus 5 Lesen und Schreiben von Stimuli nach dem zweiten Konzept

```
function LESESTIMULI(idx)
    StimuliWert ← stimuli[idx];
    StimuliZeit ← StimuliWert;
    StimuliZeit ← StimuliZeit  AND  3FFFFFFF16;
    StimuliZeit ← StimuliZeit  OR   4000000016;
    StimuliWert ← StimuliWert  AND  C000000016;
end function
function SCHREIBESTIMULI(idx)
    StimuliZeit ← StimuliZeit  AND  3FFFFFFF16;
    StimuliZeit ← StimuliZeit  OR   StimuliWert;
    stimuli[idx] ← StimuliZeit;
end function
```

In Tabelle 4.1 sind die Eigenschaften der jeweiligen Konzepte gegenübergestellt. Die erfassten Eigenschaften wurden analysiert und in Vor- und Nachteile eingeteilt. Die Analyse wurde hierzu im Kontext der Architekturparameter vorgenommen, welche die Effizienz der Implementierung auf der datenparallelen Architektur maßgeblich bestimmen.

	Konzept 1: 2-Tupel	**Konzept 2: Codierung einer Gleitkommazahl**
Vorteile	Keine Zwischenschritte zur Benutzung der Werte erforderlich	Gewinn von 2 Informationen pro 32 Bit-Speicheranfrage
	Zahlenbereich nach IEEE754 vollständig vorhanden	Geringere Speicherbandbreite notwendig, um Stimuli vollständig zu lesen
		Vollständige Nutzung sämtlicher Bits des Gleitkommaanteils
Nachteile	Nichtbenutzung von 30 Bit im vierwertigen Fall bei 32 Bit-Darstellung	Rechenzeitverlust durch Kodierung sowie Dekodierung
	Negative IEEE754-Zahlen werden im Simulationsmodell nicht verwendet	Einschränkung des darstellbaren Zahlenbereichs nach IEEE754

Tabelle 4.1: Vergleich der Konzepte zur Darstellung eines Stimulus

In Tabelle 4.2 ist die erwartete Anzahl an Zyklen pro Lese- bzw. Schreibvorgang der beiden Konzepte dargestellt. Das erste Konzept weist im direkten Vergleich eine signifikant höhere Anzahl an Zyklen auf, welche hauptsächlich durch die Zugriffe innerhalb des globalen Speichers entstehen. Pro Stimulus werden hierbei zwei Zugriffe benötigt. Im zweiten Konzept wird lediglich eine 32-Bit-Leseoperation pro Stimuli innerhalb des globalen Speichers benötigt. In der Folge wurde das zweite Konzept realisiert.

	Konzept 1: 2-Tupel	Konzept 2: Codierung einer Gleitkommazahl
Operationen im globalen Speicher (Stimulus Lesen)	2	1
Operationen in lokalen Registern (Stimulus Lesen)	2	9
Operationen im globalen Speicher (Stimulus Schreiben)	2	1
Operationen in lokalen Registern (Stimulus Schreiben)	2	6
Anzahl der Zyklen (Globaler Speicher)	$1600 - 3200$	$800 - 1600$
Anzahl der Zyklen (lokale Register)	88	330
Summe der Zyklen	$1688 - 3288$	$1030 - 1930$
Durchschnitt	2488	1480

Tabelle 4.2: Vergleich der Konzepte innerhalb der Anwendung der *CUDA*-Architektur

In Abbildung 4.7 ist der Kodiervorgang an einer Gleitkommazahl nach *IEEE754*-Standard dargestellt. Bei der Kodierung geht die Information über die beiden höchstwertigen Bits verloren. Daher ist es erforderlich, innerhalb der Dekodierung einen vordefinierten Wert einzusetzen. Dieser vordefinierte Wert kann vier unterschiedliche Zustände annehmen, da er aus zwei Bits besteht. Da im Simulationsmodell nur positive Zeitpunkte verwendet werden, wird das Bit, welches für das Vorzeichen reserviert ist, auf 0 gesetzt. Hierdurch wird nach *IEEE754* eine positive Gleitkommazahl dargestellt. Das zweite Bit repräsentiert das höchstwertige Bit im Exponenten der Gleitkommazahl. Tabelle 4.3 zeigt den darstellbaren Zahlenbereich in Abhängigkeit von der Wahl des höchstwertigen Bits.

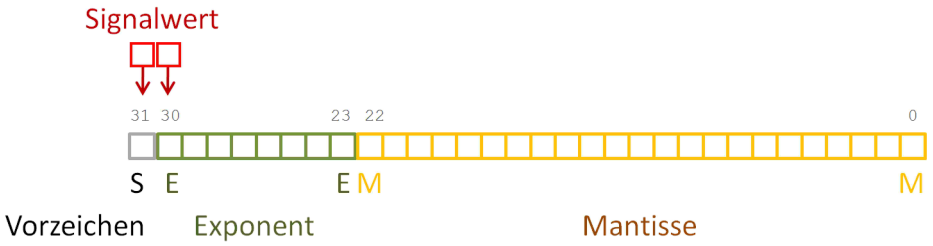


Abbildung 4.7: Kodierung eines Stimulus im Kontext des IEEE-754 Standards

	Codierung 00	Codierung 01
Kleinste darstellbare Zahl (normalisiert)	$0,0$	$2,0$
Größte darstellbare Zahl (normalisiert)	$1-2^{-24} \approx 1.\bar{9}$	$2^{127} * (1-2^{-24}) \approx 3,40282 * 10^{38}$

Tabelle 4.3: Darstellbarer Zahlenbereich des Stimulidatentyps in Abhängigkeit der Maskierung

Für das Simulationsmodell ist der Zahlenbereich ≥ 2.0 in der Realisierung von größerem Nutzen. Dies begründet sich darin, dass in der Simulation überwiegend positive ganzzahlige Zeitpunkte betrachtet werden. Daher kommt die Codierung 01 in der Dekodierung zum Einsatz.

Der erzeugte Datentyp wurde in der Schnittstelle zu *ADAMA* als Klasse *MVFloat* implementiert. In der datenparallelen Implementierung wurde dieser Datentyp nicht explizit umgesetzt. Stattdessen werden Instanzen des Datentyps verwendet, der eine Gleitkommazahl einfacher Genauigkeit nach IEEE754 darstellt (*float*). Auf diesen Instanzen werden die zuvor erläuterten Kodier- und Dekodieroperationen angewendet.

4.2.2 Mehrwertige Stimulifolgen

Eine Menge von Stimuli, welche den Wert eines Signals innerhalb eines Zeitraumes bestimmen, werden in so genannten *Stimulifolgen* zusammengefasst. In der Schnittstelle zu ADAMA wurde hierzu der Datentyp *MVWave* entwickelt, dessen Klassendiagramm in Abbildung 4.8 dargestellt ist. Dieser Datentyp enthält eine verkettete Liste von einzelnen *MVFloat*-Instanzen. Die Funktionalitäten des Datentyps *MVWave* umfassen die Erzeugung von mehrwertigen Stimulifolgen, die Manipulation und Entfernung einzelner Stimuli sowie die Kodierung und Dekodierung von Zeitpunkt und Signalwert zu Stimuli.

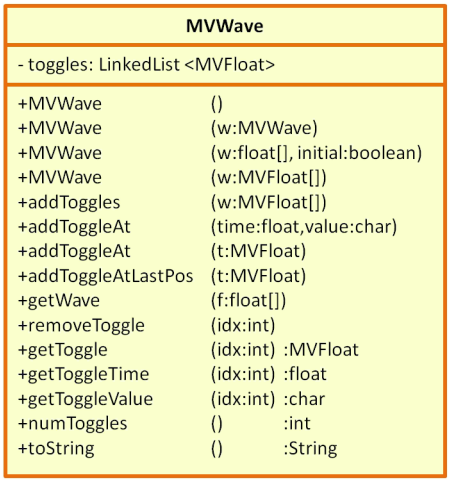


Abbildung 4.8: Klassendiagramm zum Datentyp *MVWave*

Wie zuvor erläutert, ist die Anzahl der Stimuli, welche die Ausgangsimplementierung behandeln kann, begrenzt. Zu Beginn einer Simulation müssen sämtliche Stimuli in den Datenstrukturen der Ausgangsimplementierung vorhanden sein. Diese Beschränkung entsteht durch die endliche Kapazität des Speichers. Es ist nicht möglich, Speicher von ausgewerteten Stimuli für weitere Stimuli freizugeben.

CWTSim begegnet dieser Beschränkung mit einem kontinuierlichen Simulationskonzept. In der Folge besteht die Möglichkeit potentiell unbegrenzten Stimulifolgen an beliebigen Schaltungen zu simulieren. Hierzu werden beliebig große Stimulifolgen in Abschnitte endlicher Größe zerlegt. Diese Teilfolgen werden kontinuierlich simuliert. Ein Beispiel für eine Zerlegung einer Stimulifolge ist in Abbildung 4.9 dargestellt.

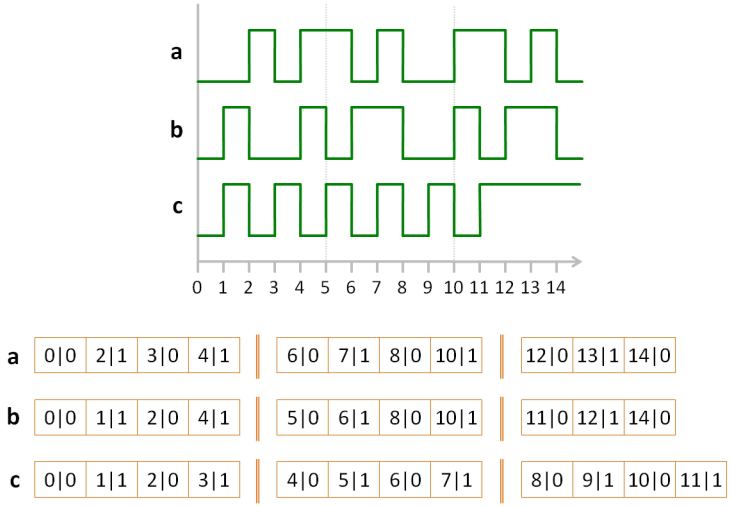


Abbildung 4.9: Zerlegung von Stimulifolgen in Teilfolgen mit maximal vier Stimuli

4.2.3 Datenparallele Implementierung von mehrwertigen Stimulifolgen

In der datenparallelen Implementierung von CWTSim wird ein Array (*Feld*) verwendet, in welchem Stimuli gemeinsam abgelegt werden. Für jedes Signal einer Schaltung wird zunächst eine Stimulifolge in diesem Array angelegt. In Abhängigkeit von der Anzahl der Simulationsinstanzen werden die Stimulifolgen entsprechend vervielfältigt angelegt.

Durch die gezielte Bildung von Teilarrays innerhalb dieses *Stimuli-Arrays* werden einzelne Stimuli zu Stimulifolgen zusammengefasst. Zu jeder Stimulifolge werden Kontextinformationen gespeichert, die zur Auswertung der Stimulifolgen an beliebigen Gattern sowie zur Verwaltung des Speichers benötigt werden. In Abbildung 4.10 sind diese Kontextinformationen am Beispiel einer Stimulifolge dargestellt. Das erste Element dieses Teilarrays enthält eine Aussage über den Auswertungsfortschritt zur zugehörigen Stimulifolge. Dieser Fortschritt wird in Form einer positiven Ganzzahl repräsentiert. Die Zustände, die dieser Fortschritt beschreibt, sind in Tabelle 4.4 aufgeführt. Der zweite Eintrag des Teilarrays enthält eine Ganzzahl, welche den Index des letzten verwendeten Elementes innerhalb der nachfolgenden Stimulifolge speichert. Im Rahmen einer Auswertung dieser Stimulifolge wird dieser Index als Zeiger verwendet. Er dient dazu, ein vorzeitiges Ende der Stimulifolge innerhalb eines Teilarrays zu bestimmen.

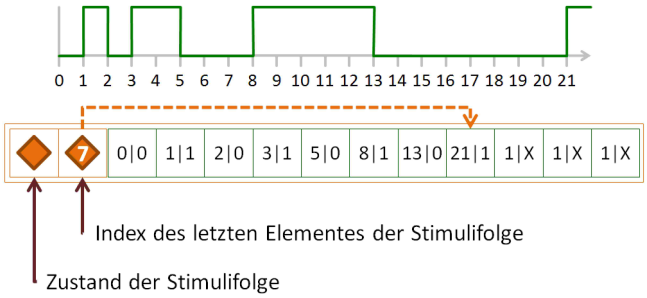


Abbildung 4.10: Darstellung einer Stimulifolge im Hauptspeicher des Grafikprozessors

Zahlenwert X	**Zustand**	**Maßnahme**
-1	Keine bisherige Auswertung	-
≥ 1	gegenwärtige Auswertung durch X nachfolgende Gatter	Auswertung fortsetzen
0	Auswertung durch nachfolgende Gatter abgeschlossen	Freigabe der Stimulifolge

Tabelle 4.4: Zustände einer Stimulifolge im Kontext des Auswertungsfortschritts

4.2.4 Organisation von Stimulifolgen im Kontext von Simulationsinstanzen

Wie zuvor erläutert werden die Stimulifolgen in der datenparallelen Implementierung entsprechend der Anzahl von Simulationsinstanzen vervielfältigt. Jedes Signal besitzt für jede Simulationsinstanz genau eine Stimulifolge. Die Menge der Stimulifolgen, die zu einem Signal zugeordnet ist, wird im Folgenden als *Signalbund* bezeichnet.

Analog zur Ausgangsimplementierung werden die einzelnen Stimuli innerhalb des Stimuli-Arrays in einer optimierten Form angeordnet. Hierzu werden die Stimuli, welche zu einer Ordnung gehören, innerhalb der *Signalbünde* fortlaufend angeordnet. Diese Methode steigert die räumliche Kohärenz der Schreib- und Leseoperationen innerhalb der parallelen Auswertung von Stimulifolgen. Im Verlauf der Auswertung werden die einzelnen Stimuli innerhalb einer Stimulifolge fortlaufend ausgewertet. Durch die parallele Auswertung verschiedener Stimulifolgen entsteht eine parallele Durchquerung der Stimuli entsprechend der jeweiligen Ordnung. In der Folge werden sämtliche Stimuli, welche zu einer Ordnung gehören, parallel im Hauptspeicher gelesen oder geschrieben. Dieser Zusammenhang ist in Abbildung 3.5 (Kapitel 3.3.3) dargestellt.

Die Stimulifolgen innerhalb der Signalbünde verfügen über eine begrenzte Zahl von Stimuli. Diese Anzahl ist als Simulationsparameter frei wählbar. CWTSim repräsentiert Mengen von Stimuli, die einen größeren Umfang als eine Stimulifolge besitzen, durch die Verkettung von Signalbünden. Durch die Verkettung zweier Signalbünde steht jeder Stimulifolge des ersten Signalbundes eine weitere Stimulifolge im zweiten Signalbund zur Verfügung. In diesen hinzugefügten Stimulifolgen können weitere Stimuli repräsentiert

werden. Die Verkettung von Signalbünden lässt sich beliebig fortsetzen. Die Verkettung von Signalbünden wird in zwei Anwendungsfällen erforderlich. Der erste Anwendungsfall tritt im zuvor beschriebenen Fall auf, wenn eine Anzahl von Stimuli über die Dimensionierung einer Stimulifolge hinausgeht. Ein weiterer Anwendungsfall tritt auf, wenn innerhalb der Auswertung von Stimulifolgen Überläufe entstehen. Die Verkettung zweier Stimulifolgen ist in Abbildung 4.11 veranschaulicht.

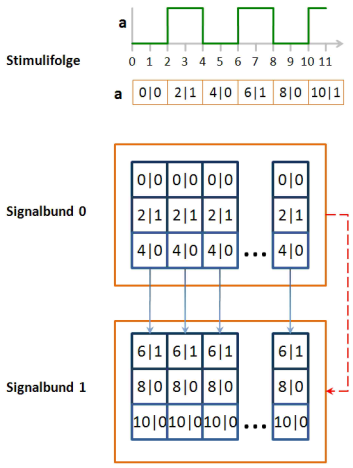
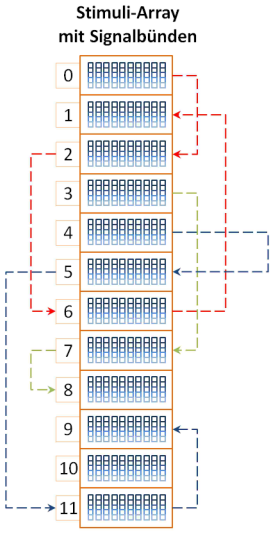


Abbildung 4.11: Verknüpfung von Stimulifolgen innerhalb zweier Signalbünde

Die Dokumentation sämtlicher Verkettungen wird durch eine Datenstruktur vorgenommen, die als *Signalbundkarte* bezeichnet wird. Diese Datenstruktur ist in Form eines Array implementiert. Innerhalb dieses Arrays werden Kontextinformationen festgehalten, die sämtliche Signalbünde im Stimuli-Array beschreiben. Die einzelnen Elemente der Datenstruktur enthalten folgende Informationen zu einem Signalbund:

- Index des Signals, zu welchem der Signalbund gehört
- Index des vorhergehenden Signalbundes
- Index des nachfolgenden Signalbundes

Die Datenstruktur, welche innerhalb der *Signalbundkarte* angewendet wird, ist beispielhaft in Abbildung 4.12 dargestellt.



Signalbundkarte

Idx	0	1	2	3	4	5	6	7	8	9	10	11
Signal	0	0	0	1	2	2	0	1	1	2	-	2
Vorherig	-	6	0	-	-	4	2	3	7	11	-	5
Nachfolgend	2	-	6	7	5	11	1	8	-	-	-	9

Abbildung 4.12: Repräsentation von Stimulifolgen innerhalb von Signalbünden sowie der übergeordneten Organisation durch eine Signalbundkarte

4.2.5 Verwaltung der Menge von Signalen einer Schaltung

Die Menge der Signale innerhalb einer Schaltung wird in einer Datenstruktur verwaltet, die als *Signalkarte* bezeichnet wird. Für jede Simulationsinstanz steht innerhalb der Signalkarte eine Menge von Kontextinformationen bereit. Die Menge an Kontextinformationen der Signalkarte ist in Abbildung 4.13 dargestellt. Hauptaufgabe der Signalkarte ist die Referenzierung der Signalbünde, welche von einem Signal verwendet werden. Hierzu wird der Index des ersten und letzten verwendeten Signalbundes gespeichert, welche die Stimulifolgen des Signals enthalten.

Die weiteren Kontextinformationen dienen der Dokumentation des Auswertungsfortschritts. Der Auswertungsfortschritt wird aus der Sicht der Gatter durchgeführt. Jedes Gatter zeichnet für dessen Eingänge die Auswertungsfortschritte explizit auf. Die erfassten Fortschritte werden in den Kontextinformationen des Signals gespeichert, welches sich am Ausgang des Gatters befindet. Diese Vorgehensweise ist in Abbildung 4.14 dargestellt.

Der Auswertungsfortschritt am Ausgang des Gatters wird als *letzter betrachteter Signalbund* festgehalten. Die Auswertungsfortschritte an den Eingängen der Gatter werden als *letzter betrachteter Signalbund A* beziehungsweise als *letzter betrachteter Signalbund B* dokumentiert. Innerhalb der Signalbünde wird die Position des zuletzt betrachteten Stimulus innerhalb der beiden Eingänge erfasst. Dies wird als *Fortschritt im Signal A* beziehungsweise *Fortschritt im Signal B* gespeichert. Diese Methode ermöglicht eine unmittelbare Fortsetzung der Auswertung nach jeder weiteren kontinuierlichen Hinzufügung von weiteren Stimuli.

Die Aufzeichnung der Auswertungsfortschritte an den Eingängen eines Gatters durch die

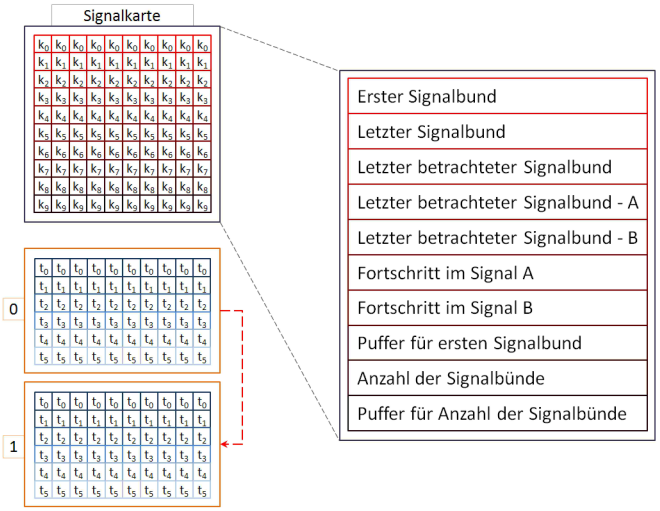


Abbildung 4.13: Darstellung der Struktur der Signalkarte sowie der Organisation der einzelnen Informationen zu Stimulifolgen im Hauptspeicher des Grafikprozessors

Signalkarte an den Ausgängen des Gatters ist unentbehrlich. Falls jedes Signal dessen Auswertungsfortschritt selbst aufzeichnet, würde dies zu Fehlern in der kontinuierlichen Simulation führen. Diese Fehler treten in der Gegenwart von Verzweigungen auf. Verzweigungen führen dazu, dass bestimmte Eingänge von Gattern ein Signal teilen. Dieses gemeinsam genutzte Signal wird jedoch höchstens durch eine Verkettung von Stimulifolgen repräsentiert. Dementsprechend stehen die Kontextinformationen zur Speicherung der Auswertungsfortschritt ebenfalls höchstens in einer Instanz zur Verfügung. Die Gatter besitzen über existierende Verzweigungen an ihren Eingängen keine Informationen. Daher werten sie die zugehörige Stimulifolge aus, als wäre keine Verzweigung vorhanden. Falls di-

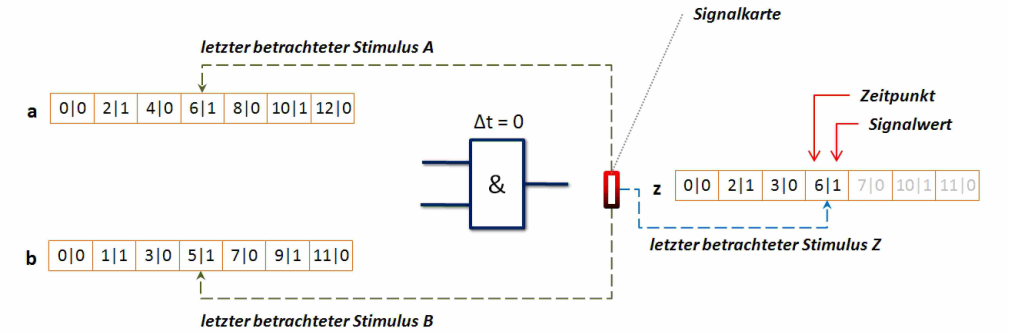


Abbildung 4.14: Veranschaulichung der Aufzeichnung von Auswertungsfortschritten an den Eingängen beliebiger Gatter

vergierende Auswertungsfortschritte auftreten, kann höchstens ein Auswertungsfortschritt dokumentiert werden. Sämtliche übrige Auswertungsfortschritte bleiben unbehandelt. Wird die Auswertung nach dem Hinzufügen weiterer Stimuli fortgesetzt, werden sämtliche Gatter einen gemeinsamen Auswertungsfortschritt verwenden. In der Folge führt dies zu falschen Ergebnissen. Somit ist die Dokumentation der Auswertungsfortschritte durch die Signalkarte an den Ausgängen des Gatters die notwendige Methode. Eine Veranschaulichung der erläuterten Problematik ist in Abbildung 4.15 dargestellt.

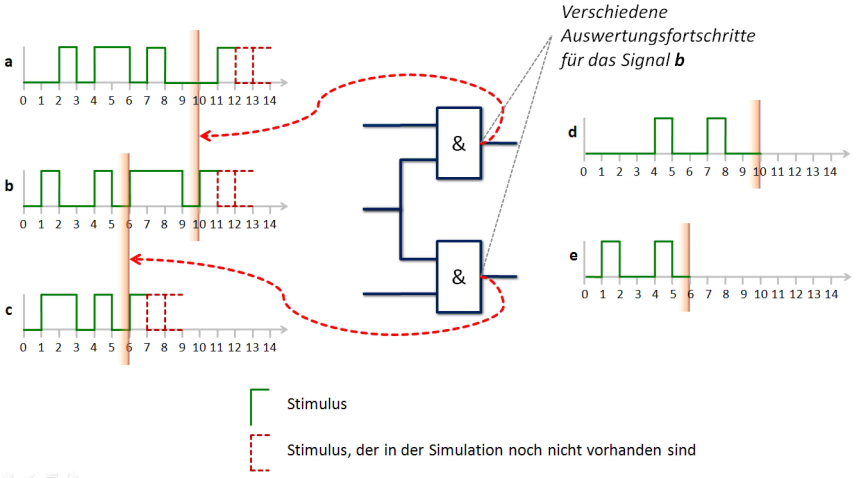


Abbildung 4.15: Veranschaulichung der auftretenden Problematik im Falle divergierender Bearbeitungsfortschritte im Kontext von Verzweigungen

Die *Signalkarte* enthält darüber hinaus die Anzahl der Signalbünde, welche an ein Signal vergeben wurden. Zwei weitere Einträge dienen der Vermeidung von nichtdeterministischen Effekten, welche durch die Parallelität innerhalb der Ausführung von sämtlichen Auswertungs- und Verwaltungsalgorithmen entstehen können. Hierzu wird ein initialer Zustand festgehalten, von dem aus nur ein möglicher Zustandsübergang möglich ist. Zur Vermeidung von nichtdeterministischem Verhalten wird der vorgefundene Zustand mit dem dokumentierten initialen Zustand verglichen. Falls der vorgefundene Zustand dem initialen Zustand entspricht, wird der vorgesehene Zustandsübergang durchgeführt.

Im Detail umfassen diese Informationen den Index des Signalbundes, welcher zur Freigabe bestimmt ist sowie die Anzahl der Signalbünde, welche an ein Signal vergeben wurden. Beide Informationen werden während des Hinzufügens sowie während der Freigabe von Signalbünden benötigt. Die zugehörigen Algorithmen werden in einer parallelen Form ausgeführt, wodurch der Ausschluss nichtdeterministischen Verhaltens zwingend erforderlich ist.

Die einzelnen Elemente der Signalkarte werden innerhalb des Speichers in einer Form angeordnet, die Analog zur Organisation innerhalb des Stimuli-Array entwickelt wurde. Hierzu werden die Elemente gleicher Ordnung zusammengefasst. Innerhalb der datenparallelen Durchführung wird hierdurch die räumliche Kohärenz bei sämtlichen anfallenden

Speicherzugriffen ausgenutzt. Hieraus resultiert eine signifikant verringerte Anzahl an Speicherzugriffen, welche im Hauptspeicher des Grafikprozessors durchgeführt werden. Eine Folge ist die direkte Steigerung der Effizienz.

4.2.6 Verwaltung der Datenstrukturen innerhalb einer kontinuierlichen Simulation

Die vorgestellte Datenstruktur erfährt während der kontinuierlichen Ausführung der Simulation fortlaufend Anforderungen zur Erweiterung des verwalteten Speichers. Diese Anforderungen entstehen durch die fortlaufende Eingabe von weiteren Stimuli an den Eingängen der Schaltung sowie durch Überläufe, die innerhalb der Auswertung von Stimulifolgen auftreten. Neben der fortlaufenden Erweiterung des Speichers ist eine begleitende Freigabe von ungenutzten Speichers erforderlich. Ohne die begleitende Freigabe ungenutzten Speichers wäre ein Überlauf des gesamten genutzten Speichers während einer kontinuierlichen Simulation unausweichlich.

Zu Beginn einer Simulation beinhaltet das Stimuli-Array eine Reihe von Signalbünden. Die Anzahl dieser Signalbünde entspricht der Anzahl der Signale innerhalb der zu untersuchenden Schaltung. Der Rest des Stimuli-Arrays entspricht in der Größe einem Vielfachen des benötigten Speichers eines Signalbundes. Dieser *freie Speicherplatz* innerhalb des Stimuli-Arrays wird im Falle der vorgenannten Ereignisse dazu genutzt, weitere Signalbünde für ein Signal anzulegen und zu initialisieren.

Die Information über verwendete sowie nicht verwendete Signalbünde findet sich in der *Signalbundkarte* wieder. Falls innerhalb des Stimuli-Array Bereiche für freie Signalbünde zur Verfügung stehen, so sind diese Bereiche in der *Signalbundkarte* für eine spätere Belegung durch Signalbünde gekennzeichnet. Falls Signalbünde nicht verwendet werden, verfügen sie über kein zugeordnetes Signal. Dieser Zusammenhang wird verwendet, um freie Signalbünde an eine Kette von Signalbünden hinzuzufügen. Der initiale Zustand der verschiedenen Datenstrukturen zu Beginn einer Simulation ist in Abbildung 4.16 dargestellt.

In der datenparallelen Implementierung von CWTSim wird hierzu eine parallele Suche auf der Datenstruktur der *Signalbundkarte* durchgeführt. In einem ersten vorbereitenden Schritt wird hierzu innerhalb sämtlicher Stimulifolgen aller zuletzt betrachteten Signalbünde parallel festgestellt, ob Überläufe aufgetreten sind. Für jede einzelne Stimulifolge steht ein Thread zur Verfügung, der diese Feststellung durchführt. Ein Überlauf tritt ein, wenn sämtliche Stimuli der Stimulifolge des letzten Signalbundes innerhalb der verketteten Datenstruktur verwendet werden. Somit stehen keine weiteren Elemente in der betreffenden Stimulifolge zur Verfügung, um weitere Stimuli aufnehmen zu können. Im Falle eines aufgetretenen Überlaufs wird ein Überlauf-Indikator (*engl. Flag*) für den betreffenden Signalbund gesetzt. Die Implementierung dieser Herangehensweise ist in Algorithmus 6 dargestellt.

Die anschließende parallele Suche wird für sämtliche Stimulifolgen eines Signalbundes gemeinsam durchgeführt. Das heißt, dass jedem Überlauf ein Thread zugeordnet wird, der eine parallele Suche nach einem weiteren Signalbund durchführt. Eine Optimierung

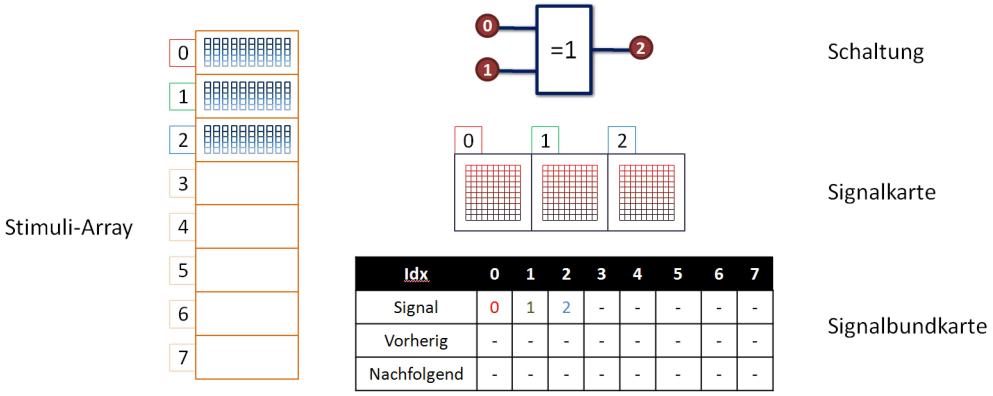


Abbildung 4.16: Veranschaulichung des initialen Zustandes des Stimuli-Array zu Beginn einer Simulation

Algorithmus 6 Feststellung von Überläufen in einer Stimulifolge

```
/* Hinweis: Die Bestimmung des aktuellen Signals, sowie der aktuellen Simulationsin-
stanz ist abhängig von der eingesetzten Architektur */
signal ← aktuelles_Signal;
slot ← aktuelle_Simulationsinstanz;

/* Suche nach der Stimulifolge des letzten Signalbündels */
Letzter_Signalbund ← Signalkarte[signal][slot].letzter_Signalbund;
Stimuli_Folge ← Letzter_Signalbund ∗ signalschritt + slot;

/* Untersuchung der gefundenen Stimulifolge auf Überläufe */
if Sämtliche Stimuli von Stimuli_Folge belegt? then
    Setze allgemeinen Überlauf-Indikator
    Setze Überlauf-Indikator für aktuellen Signalbund
end if
```

der parallelen Suche wird erreicht, wenn sämtliche Threads, die an der Suche beteiligt sind, nicht gleichzeitig ein Element der *Signalbundkarte* untersuchen. In diesem Fall wird durch die verwendeten Synchronisationsmechanismen sichergestellt, dass nur ein Thread den gemeinsam betrachteten Signalbund erhält. Sämtliche andere Threads setzen die Suche am nachfolgenden Element der *Signalbundkarte* fort, wobei ebenfalls nur ein Thread den betrachteten Signalbund erhalten kann. Im besten anzunehmenden Fall entspricht die Anzahl der Iterationen der Anzahl der beteiligten Threads. Im Fall einer verteilten Suche, wie sie in Abbildung 4.17 dargestellt ist, betrachtet die Menge an Threads in jedem Iterationsschritt eine zueinander disjunkte Menge an Signalbünden.

Im besten anzunehmenden Fall wird nur eine einzige Iteration benötigt, um sämtlichen Threads ein Signalbund zuzuweisen. Im Falle eines weiteren Suchablaufs durchsucht ein

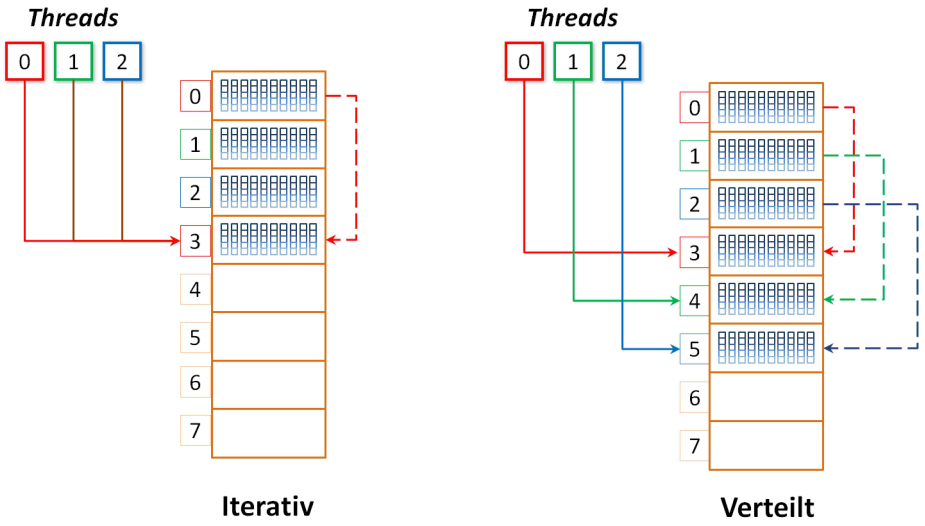


Abbildung 4.17: Vergleich der parallelen Suche in einer iterativen Form und in einer verteilten Form

einzelner Thread nicht direkt aufeinander folgende Elemente. Stattdessen durchquert der Thread die *Signalbundkarte* in einem konstanten Abstand. Dieser konstante Abstand entspricht der Anzahl der Signale, welche in der zu untersuchenden Schaltung vorhanden sind. Hat der Thread nach einer vollständigen Durchquerung der gesamten *Signalbundkarte* keinen frei verfügbaren Signalbund gefunden, setzt er die Suche in aufeinander folgenden Elementen der *Signalbundkarte* fort. Durch diese Herangehensweise wird sichergestellt, dass Endlosschleifen vermieden werden, falls dennoch Signalbünde zur Verfügung stehen, die durch die verteilte Suche ansonsten nicht gefunden würden. Wird ein unbenutzter Signalbund gefunden und einem Thread zugewiesen, so hängt er diesen Signalbund an die verkettete Liste der Signalbünde des entsprechenden Signals. Hierzu werden die Einträge zur Signalzugehörigkeit, sowie der vorangegangenen und nachfolgenden Signalbünde für den gefunden Signalbund in der *Signalbundkarte* initialisiert. In der *Signalkarte* wird für das entsprechende Signal der Eintrag über den letzten Signalbund auf den neu hinzugefügten Signalbund geändert. Darüber hinaus wird die Anzahl der Signalbünde, die zum Signal gehören, erhöht. Die Implementierung hierzu ist in Algorithmus 7 dargestellt.

Abschließend werden sämtliche neu hinzugefügte Signalbünde initialisiert. Die Initialisierung in der datenparallelen Implementierung von CWTSim wird hierbei parallel auf sämtlichen Stimulifolgen aller beteiligten Signalbünde durchgeführt. Hierzu werden die Kontextinformationen der Stimulifolgen für eine spätere Verwendung innerhalb der Auswertung der Stimulifolgen vorbereitet. Der Zustand der Stimulifolge ist zum Initialisierungszeitpunkt unbenutzt, weshalb dieser mit -1 gekennzeichnet wird. Der letzte Stimulus in der zugehörigen Stimulifolge existiert zum Initialisierungszeitpunkt noch nicht. Daher wird der Zeiger für den letzten Stimulus vor das erste Element der Stimulifolge gesetzt, welches den ersten Stimulus beinhalten wird. Die Implementierung ist in Algorithmus 8

dargestellt.

Wie zuvor erläutert, werden Signalbünde innerhalb der kontinuierlichen Simulation nach ihrer Benutzung wieder freigegeben. Der aktuelle Zustand eines Signalbundes spiegelt sich in der Benutzung der darin enthalten Stimulifolgen wider. Falls mindestens eine Stimulifolge benutzt wird, gilt der Signalbund ebenfalls als benutzt. Sind hingegen sämtliche Stimulifolgen eines Signalbundes ausgewertet und daher ungenutzt, kann der Signalbund für eine spätere Wiederverwendung freigegeben werden. In der datenparallelen Implementierung wird für die Erkennung des Zustandes der Signalbünde pro Stimulifolge ein Thread verwendet, der dessen Zustand ausliest. Sofern eine Stimulifolge verwendet wird, wird ein Indikator für das entsprechende Signal gesetzt. Falls während der Auswertung sämtlicher Stimulifolgen eines Signalbundes der Indikator unverändert bleibt, kann jener Signalbund in einem weiteren Schritt freigegeben werden.

Für diese Aufgabe wird pro Signalbund ein Thread verwendet. Dieser Thread prüft im ersten Schritt den Indikator, des zugehörigen Signals. Falls eine Freigabe erforderlich ist, werden die Kontextinformationen zum Signalbund in der *Signalbundkarte* auf initiale Werte zurückgesetzt. Hierdurch kann der Signalbund in einer späteren Suche nach freien Signalbunden als solcher explizit erkannt werden. Die Kontextinformationen zum Signal, welches den Signalbund freigibt, werden entsprechend in der *Signalkarte* geändert. Hierzu wird der Index des ersten verwendeten Signalbundes auf den zweiten Signalbund in der Kette der verwendeten Signalbunde geändert. Darüber hinaus wird die Information über die Anzahl der Signalbunde, welche vom Signal benutzt werden, verringert.

Die Erkennung und Freigabe von ungenutzten Signalbünden wird in der datenparallelen Implementierung an den Eingängen der Gatter durchgeführt. Diese Herangehensweise begründet sich darin, dass ausschließlich die Auswertung der Signalbünde an den Eingängen von Gattern dazu führen kann, dass die betrachteten Signalbünde freigegeben werden können. Diese Herangehensweise führt jedoch in der Gegenwart von Verzweigungen zu fälschlicherweise mehrfach durchgeführten Freigaben, falls mehr Threads an der Freigabe beteiligt sind, als die datenparallele Architektur gleichzeitig ausführen kann. In diesem Fall führt die datenparallele Architektur sämtliche Threads, welche über die maximale Anzahl hinausgehen, fortlaufend sequentiell aus. Hierbei kann es vorkommen, dass mehrere Gatter, die an einer Verzweigung beteiligt sind, versuchen einen gemeinsam genutzten Signalbund freizugeben. Der Freigabeindikator zeigt nicht an, welcher Signalbund eines Signals zur Freigabe bestimmt ist. Falls nun die Threads der jeweiligen Gatter zu verschiedenen sequentiellen Zeitpunkten ausgeführt werden, wird die Freigabe von Signalbünden an diesem Signal mehrfach durchgeführt, wodurch fälschlicherweise Signalbünde freigegeben werden, die nicht zur Freigabe vorgesehen sind. Die Lösung des Problems ist die explizite Angabe des Indizes zum betreffenden Signalbund neben der Angabe des Signals, in welchem die Freigabe stattfindet. Hierzu prüfen die die jeweiligen Threads den Index des ihnen vorliegenden Signalbundes, bevor sie eine Freigabe durchführen. Die Implementierung der vorgestellten Herangehensweise zur Erkennung von ungenutzten Signalbünden ist in Algorithmus 9 dargestellt. Die Implementierung zur Freigabe von Signalbünden ist in Algorithmus 10 dargestellt.

Algorithmus 7 Erweiterung einer Kette von Signalbünden um einen weiteren Signalbund

```
Anzahl_Signale = Anzahl der Signale innerhalb der Schaltung
Anzahl_Signalbünde = Größtmögliche Anzahl Signalbünde innerhalb des Stimuli-Array

/* Hinweis: Die Bestimmung des aktuellen Signals, sowie der aktuellen Simulationsin-
stanz ist abhängig von der eingesetzten Architektur */
signal ← aktuelles_Signal;
slot ← aktuelle_Simulationsinstanz

/* Die Erweiterung der Kette wird nur durchgeführt, wenn sie erforderlich ist */
if Überlauf-Indikator für aktuellen Signalbund gesetzt? then

    Letztes_Register ← Signalkarte[signal][slot].letzter_Signalbund;
    SignalbundIdx ← (Letzter_Signalbund + Anzahl_Signale) mod Anzahl_Signalbnde;

    while Neuer Signalbund nicht gefunden do
        if Signalbundkarte[SignalbundIdx].signal = −1 then
            Atomische Operation: Versuche Signalbundkarte[SignalbundIdx] zu setzen
            if Signalbundkarte[SignalbundIdx] exklusiv erhalten? then
                Signalbundkarte[SignalbundIdx].signal ← signal
                Signalbundkarte[SignalbundIdx].next ← −1
                Signalbundkarte[SignalbundIdx].prev ← Letzter_Signalbund

                Signalbundkarte[Letzter_Signalbund].next ← SignalbundIdx

                Signalkarte[signal][slot].letzter_Signalbund ← SignalbundIdx;
                Signalkarte[signal][slot].Anzahl_Signalbnde++;
            end if
        end if

        if Signalbundkarte noch nicht vollständig durchquert? then
            SignalbundIdx ← (Letzter_Signalbund + Anzahl_Signale) mod Anzahl_Signalbnde
        else
            SignalbundIdx ← (Letzter_Signalbund + 1) mod Anzahl_Signalbnde
        end if

    end while
end if
```

4.2.7 Darstellung von Gattern

Die Repräsentation von Gattern innerhalb von CWTSim entspricht der Herangehensweise, welche die Ausgangsimplementierung verfolgt. Innerhalb der Analyse der Ausgangsimplementierung wurden keine Beschränkungen innerhalb der Repräsentation von Gattern festgestellt, welche eine effiziente Prognose zum erwarteten Energieverbrauch einer Schaltung negativ beeinflussen.

Algorithmus 8 Initialisierung von Signalbünden

```
/* Hinweis: Die Bestimmung des aktuellen Signals, sowie der aktuellen Simulationsinstanz ist abhängig von der eingesetzten Architektur */
signal ← aktuelles_Signal;
slot ← aktuelle_Simulationsinstanz;

/* Die Initialisierung eines Signalbundes wird nur durchgeführt, wenn sie erforderlich ist */
if Überlauf-Indikator für aktuellen Signalbund gesetzt? then
    /* Suche nach der Stimulifolge des letzten Signalbündels */
    Letzter_Signalbund ← Signalkarte[signal][slot].letzter_Signalbund;
    Stimuli_Folge ← Letzter_Signalbund ∗ signalschritt + slot;

    Stimuli_Array[Stimuli_Folge] ← −1;
    Stimuli_Array[Stimuli_Folge + 1] ← Stimuli_Folge + 1;
end if
```

Algorithmus 9 Feststellung des Zustandes innerhalb der Signalbünde am Eingang eines Gatters

```
/* Hinweis: Die Bestimmung des aktuellen Gatters, sowie der aktuellen Simulationsinstanz ist abhängig von der eingesetzten Architektur */
Gatter ← aktuelles_Gatter;
slot ← aktuelle_Simulationsinstanz;

/* Bestimmung der Signale an den Eingängen des Gatters */
signal_a ← Gatter.a;
signal_b ← Gatter.b;

/* Bestimmung der jeweiligen Stimulifolgen an den Eingängen des Gatters */
Erster_Signalbund_a ← Signalkarte[signal_a][slot].erster_Signalbund;
Stimuli_Folge_a ← Erster_Signalbund_a ∗ signalschritt + slot;
Erster_Signalbund_b ← Signalkarte[signal_b][slot].erster_Signalbund;
Stimuli_Folge_b ← Erster_Signalbund_b ∗ signalschritt + slot;

/* Untersuchung des Zustandes der Stimulifolgen */
if waves[Stimuli_Folge_a] > 0 then
    Setze Indikator, welcher eine Benutzung von Signal A angibt
    Setze Indikator, welcher eine Benutzung von Signal A angibt
    Signalkarte[signal_a].PufferSignalIdx ← Erster_Signalbund_a
end if
if waves[Stimuli_Folge_b] > 0 then
    Setze Indikator, welcher eine Benutzung von Signal B angibt
    Signalkarte[signal_b].PufferSignalIdx ← Erster_Signalbund_b
end if
```

Algorithmus 10 Freigabe ungenutzter Signalbünde (Beispiel für den ersten Eingang)

```
/* Hinweis: Die Bestimmung des aktuellen Gatters ist abhängig von der eingesetzten
Architektur */
Gatter ← aktuelles_Gatter;

/* Bestimmung der Signale an den Eingängen des Gatters */
signal_a ← Gatter.a;

/* Falls das Signal einen unbenutzten Signalbund besitzt, wird es freigegeben */
if Indikator für Signal A gesetzt? then
    Erster_Signalbund_a ← Signalkarte[signal_a].erster_Signalbund;

    /* Prüfung, ob der vorliegende Signalbund zur Freigabe bestimmt ist */
    if Signalkarte[signal_a].PufferSignalIdx = Erster_Signalbund_a then
        Zweiter_Signalbund ← Signalbundkarte[Erster_Signalbund_a].next;
        Signalkarte[signal_a].erster_Signalbund ← Zweiter_Signalbund;

        Signalbundkarte[Erster_Signalbund_a].signal ← −1;
        Signalbundkarte[Erster_Signalbund_a].next ← −1;
        Signalbundkarte[Erster_Signalbund_a].prev ← −1;

        Signalbundkarte[Zweiter_Signalbund_a].prev ← −1;
    end if
end if
```

4.3 Detaillierte Vorgehensweise innerhalb einer kontinuierlichen Simulation

4.3.1 Kontinuierliche Hinzufügung weiterer Stimuli

Das Anlegen von Stimulifolgen sowie die fortlaufende Erweiterung jener Stimulifolgen um weitere Stimuli kann nicht trivial durchgeführt werden. Zunächst müssen eine Reihe vorbereitender Aufgaben erfüllt werden. Da die Stimuli fortlaufend in die zuvor vorgestellten Datenstrukturen eingefügt werden, ist es erforderlich, zunächst die Position des zuletzt eingefügten Stimuli zu finden. Da eine verkettete Datenstruktur zur Repräsentation der *Signalbünde* eingesetzt wird, muss für jedes Signal eine Durchquerung bis zum letzten verwendeten *Signalbund* eines Signals vorgenommen werden. Daraufhin muss die Anzahl der Stimuli, welche in die entsprechende Stimulifolge noch eingesetzt werden können, bestimmt werden. Ist die Anzahl der Stimuli größer als die Anzahl der verfügbaren Elemente in der vorhandenen Stimulifolge, müssen weitere *Signalbünde* angelegt und verwendet werden. Im Anschluss daran können die neu einzufügenden Stimuli eingefügt werden.

In CWTSim wurden Funktionalitäten implementiert, die diese Aufgaben effizient durchführen. Hierdurch wird die Datenstruktur der *Signalbünde* abstrahiert. Dem Benutzer einer Simulation steht ein Puffer zur Verfügung, in welchem Stimulifolgen für die Eingänge der

Schaltung eingefügt werden können. Dieser Puffer entspricht einer Menge von Signalbünden, welche die Signale an den Eingängen der Schaltung repräsentieren. Die Positionierung von Stimuli bleibt während des gesamten Simulationsablaufs konstant. Innerhalb jedes *Signalbundes* stehen entsprechend der Anzahl an Simulationsinstanzen Stimulifolgen zur Verfügung, welche die Eingaben für je eine Simulationsinstanz repräsentieren. Einzufügende Stimuli werden ohne Berücksichtigung des bisherigen Zustandes der Datenstrukturen innerhalb der Simulation eingefügt. Eine Veranschaulichung dieses Lösungsansatzes in Abbildung 4.18 dargestellt.

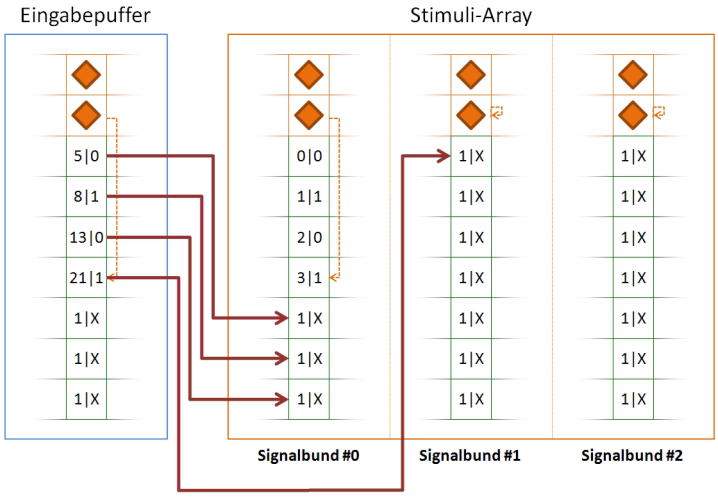


Abbildung 4.18: Veranschaulichung der Überführung von Stimuli aus dem Eingabepuffer in das Stimuli-Array

Vor dem unmittelbaren Beginn einer Simulationsdurchführung werden die Stimuli, welche sich im Puffer befinden, in die bestehenden Datenstrukturen der Simulation eingefügt. Die *Signalkarte*, in der die Fortschritte zur Auswertung der einzelnen Stimulifolgen gespeichert werden, muss für das erneute Einfügen weiterer Stimuli zunächst vorbereitet werden. Sämtliche Fortschrittsinformationen jener Signale, welche die Eingänge repräsentieren, werden in diesem Schritt zurückgesetzt. Der Fortschritt innerhalb des Puffers wird in diesem Zusammenhang auf den jeweiligen ersten Stimulus gesetzt. Darauf folgend werden die jeweiligen letzten Stimuli in den bestehenden Stimulifolgen ermittelt. Diese letzten Stimuli befinden sich in der verketteten Liste der *Signalbünde* in jenem *Signalbund*, in welchem zuletzt ein Schreibvorgang durchgeführt wurde. Diese Information wird in der Signalkarte für jedes Signal explizit gespeichert und wird an dieser Stelle verwendet, um den betreffenden *Signalbund* und den darin enthaltenen zuletzt geschriebenen Stimuli wiederzufinden. Im Anschluss werden die einzelnen Stimuli aus dem Puffer an die zuvor bestimmten freien Positionen innerhalb der Stimulifolge kopiert. Die Implementierung des beschriebenen Vorgangs ist in Algorithmus 12 dargestellt. Im Fall des Auftretens von Überläufen wird der Kopiervorgang unterbrochen. Die bis dahin verzeichneten Fortschritte werden in der *Signalkarte* dokumentiert. Hierdurch können die übrig gebliebenen Stimuli innerhalb des

Puffers in einer späteren Fortsetzung des Kopiervorgangs eindeutig bestimmt werden. Der Überlauf wird durch die bereits beschriebenen Vorgehensweisen zur Erweiterung der Datenstrukturen behandelt. Zunächst werden die einzelnen Überläufe innerhalb sämtlicher Stimulifolgen für den zugehörigen *Signalbund* bestimmt. Falls in einem *Signalbund* ein Überlauf aufgetreten ist, werden im Anschluss verfügbare *Signalbünde* innerhalb des Stimuli-Array gesucht und an die verkettete Liste der *Signalbünde* eines Signals angehängt. Sämtliche Stimulifolgen in den einzelnen neu hinzugewonnenen *Signalbünden* werden abschließend initialisiert. Diese Vorgehensweise ist in Abbildung 4.19 dargestellt.

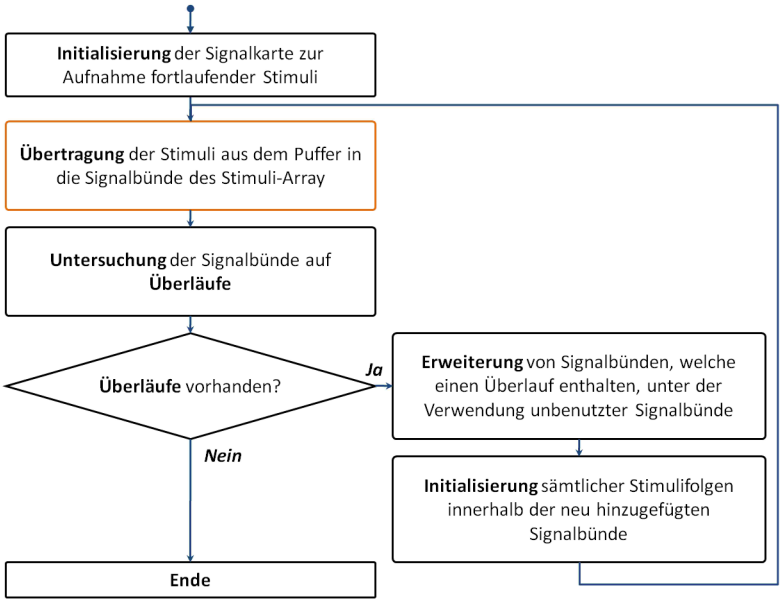


Abbildung 4.19: Darstellung zum Ablauf der Übertragung von Stimulifolgen aus dem Puffer in die Signalbünde im Kontext der Speicherverwaltung

Algorithmus 11 Durchquerung von Stimuli innerhalb verketteter Signalbünde

```
transition ← Abstand zwischen zwei Elementen einer Stimulifolge
function NextStimulus(Stimulus_Idx, Signalbund_Idx)
    Stimulus_Idx = Stimuli_Idx + transition;
    if Stimulus_Idx aus Stimulifolge übergelaufen? then
        if Signalbundkarte[Signalbund_Idx].next ≠ −1 then
            Signalbund_Idx ← Signalbundkarte[Signalbund_Idx].next;
            Stimulus_Idx ← Signalschritt ∗ Signalbund_Idx + slot;
        end if
    end if
end function
```

Algorithmus 12 Übertragung der Stimulifolgen aus dem Puffer in die Signalbünde des Stimuli-Array

```
/* Hinweis: Die Bestimmung des aktuellen Signals aus der Menge der Eingänge
innerhalb der Schaltung, sowie die Bestimmung der aktuellen Simulationsinstanz ist
abhängig von der eingesetzten Architektur */
Signal_Eingang ← Schaltungseingaenge[aktueller_Eingang];
slot ← aktuelle_Simulationsinstanz;

/* Bestimmung des Signalbundes, in welchem die bestehende Stimulifolge endet */
aktueller_Signalbund ← Signalkarte[Signal_Eingang][slot].Letzter_betrachter_Signalbund;
/* Bestimmung des Indizes an welchem weitere Stimuli eingetragen werden */
input_wave_idx ← waves[aktueller_Signalbund ∗ Signalschritt + slot + transition];
/* Ermittlung des bisherigen Kopierfortschritts */
input_buffer_idx ← Signalkarte[Signal_Eingang][slot].Fortschritt_A;
/* Ermittlung des letzten einzutragenden Stimuli */
input_buffer_last ← waves_input[Signal_Eingang ∗ Signalschritt + slot
+transition];

/* Überführung der Stimuli in die bestehende Simulation */
prev_input_wave_idx ← input_wave_idx
NextStimuli(input_wave_idx, aktueller_Signalbund);
while input_buffer_idx und input_wave_idx nicht übergelaufen? do
    waves[input_wave_idx] ← waves_input[input_buffer_idx];
    prev_input_wave_idx ← input_wave_idx;
    NextStimuli(input_wave_idx, aktueller_Signalbund);
    input_buffer_idx ← input_buffer_idx + transition;
end while

/* Speicherung des Fortschrittes */
Signalkarte[Signal_Eingang][slot].Letzter_betrachter_Signalbund ←
aktueller_Signalbund;
Signalkarte[Signal_Eingang][slot].Fortschritt_A ←
input_buffer_idx;
waves[aktueller_Signalbund ∗ Signalschritt + slot + transition] ←
prev_input_wave_idx;
```

4.3.2 Vorbereitende Schritte zu Beginn des Simulationsablaufs

Vor dem Beginn einer kontinuierlichen Simulation wird eine Initialisierung innerhalb der zuvor vorgestellten Datenstrukturen durchgeführt. Darin werden die initialen Nutzungszustände der einzelnen Stimuli bestimmt. In sämtlichen topologischen Ebenen werden hierzu die Nutzungszustände der Stimulifolgen, welche die Stimuli für die Eingänge der Gatter beinhalten, inkrementiert. Die Nutzungszustände der Stimulifolgen enthalten als Resultat die Anzahl der Gatter, welche die Stimulifolge als Eingang verwenden. Wie zuvor erläutert, wird der Nutzungszustand innerhalb der Simulation dazu verwendet, Freigaben von Signalbünden zu ermöglichen. Wurde eine Stimulifolge von einem Gatter vollständig ausgewertet, dekrementiert das Gatter den Nutzungszustand.

Die Initialisierung gewährleistet darüber hinaus den initialen Zustand der Schaltung. Der initiale Zustand der Schaltung definiert sich aus der Menge der Signalwerte, welche innerhalb der Schaltung vor der Eingabe von Stimuli bestehen. Hierzu wird der erste Stimulus jeder Stimulifolge, welches an die Eingänge der Schaltung angelegt wurde, als initialer Zustand verwendet. Die Initialisierung der Schaltung wird rein funktional durchgeführt. Es werden somit keine Verzögerungsparameter innerhalb der Initialisierung berücksichtigt. Ein Beispiel für eine durchgeführte Initialisierung ist in Abbildung 4.20 dargestellt.

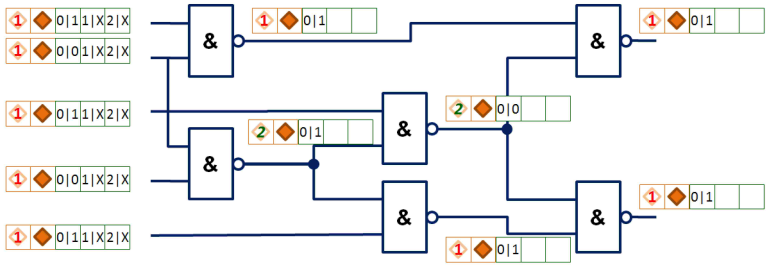


Abbildung 4.20: Beispiel einer durchgeführten Initialisierung an der Schaltung C17

4.3.3 Auswertung von kontinuierlichen Stimulifolgen an beliebigen Gattern

Die Herangehensweise zur Auswertung von kontinuierlichen Stimulifolgen in CWTSim orientiert sich an der Herangehensweise, welche in der Ausgangsimplementierung verfolgt wird. Die Analyse dieser Herangehensweise zeigt eine lineare Komplexität innerhalb des Laufzeitverhaltens auf. Die Auswertung von Stimulifolgen ist eine Aufgabe, die das vollständige Durchschreiten von Listen erfordert. Die minimal erreichbare Laufzeit, welche innerhalb des Durchschreitens einer Liste auftritt, wächst linear mit der Anzahl der Elemente, welche sich innerhalb der Liste befinden. Eine geringere Komplexität ist nicht erreichbar. Dies begründet sich darin, dass ein Algorithmus, der die Aufgabe besitzt eine Liste vollständig auswerteten, kein Element der Liste außen vor lassen darf. Das lineare Laufzeitverhalten entspricht somit einer optimalen Herangehensweise.

Wie bereits erläutert, ist die Herangehensweise der Ausgangsimplementierung, welche die Auswertung von Stimulifolgen durchführt, nicht fähig, eine Simulation über potentiell nicht endliche Stimulifolgen durchzuführen. Im Folgenden werden die entwickelten Methoden vorgestellt, welche diese Diskrepanz auflösen.

Der Algorithmus, welcher die Auswertung von Stimulifolgen an beliebigen Gattern durchführt, ist in CWTSim in drei Phasen unterteilt. Die erste Phase des Algorithmus gewährleistet die Fortführung der Auswertung. Darin werden die bisher erreichten Fortschritte innerhalb der Auswertung der Stimulifolgen ermittelt. Ziel dieser Phase ist die vollständige Wiederherstellung des Auswertungskontextes. Dieser Auswertungskontext wird in der zweiten Phase benötigt. Diese Phase umfasst die eigentliche Auswertung der Stimulifolgen. Die dritte Phase des Algorithmus tritt ein, wenn die Auswertung der Stimulifolgen für die gegenwärtigen Eingaben vollständig durchgeführt wurde. Innerhalb dieser Phase wird der erreichte Auswertungsfortschritt dokumentiert. Die Ergebnisse der dritten Phase werden in der fortlaufenden Simulation zur Wiederaufnahme der Auswertung benötigt. Diese drei Phasen werden im Folgenden detailliert erläutert.

Initialisierungsphase des Auswertungsalgorithmus

Die erste Phase des Auswertungsalgorithmus, in welchem der gegenwärtige Fortschritt der Auswertung ermittelt wird, ist in Algorithmus 13 dargestellt. In einem ersten Schritt wird der Fortschritt am Ausgang des Gatters ermittelt. Hierbei wird der Index des Stimulibundes ermittelt, in welchem im zurückliegenden Simulationszwischenschritt Stimuli erzeugt wurden. Ausgehend von der Position des zuletzt eingetragenen Stimulus, werden in der anschließenden Auswertungsphase weitere erzeugte Stimuli hinzugefügt. Anschließend werden die Kontextinformationen zu den Eingängen des Gatters ermittelt. Hierzu wird der Signalbund ermittelt, aus dem zuletzt Stimuli für den betreffenden Eingang gelesen wurden. Weitere Stimuli wurden zwischen den Auswertungsfortschritten von diesem Stimuli ausgehend hinzugefügt. Die Auswertung wird somit an dem Stimulus fortgeführt, an dem zuletzt der Auswertungsfortschritt unterbrochen wurde. Die Wiederaufnahme der

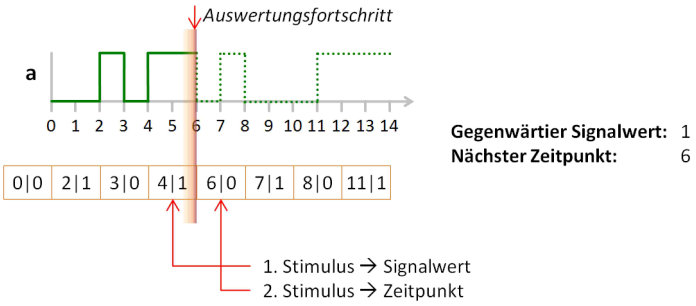


Abbildung 4.21: Wiederaufnahme der Auswertung innerhalb einer Stimulifolge

Auswertung innerhalb der Stimulifolgen an den Eingängen der Gatter ist in Abbildung

4.21 dargestellt. In diesem Schritt werden die Zeitpunkte und Signalwerte der gegenwärtig betrachteten Stimuli wiederhergestellt. Der erste Stimulus innerhalb der Stimulifolgen beschreibt den initialen Signalwert der Signale. Der zweite Stimulus in der Stimulifolge markiert den Beginn der bis dahin nicht ausgewerteten Stimuli. In der Folge wird der Zeitpunkt der auszuwertenden Stimuli aus dem zweiten Stimulus der Stimulifolge bestimmt.

Algorithmus 13 Initialisierungsphase des Algorithmus zur Auswertung von Stimulifolgen innerhalb der erweiterten Simulationsimplementierung

```
/* Hinweis: Die Bestimmung des aktuellen Gatters, den dazugehörigen Signalen,
sowie die Bestimmung der aktuellen Simulationsinstanz ist
abhängig von der eingesetzten Architektur */
Gatter ← GatterListe[GatterIdx];
slot ← aktuelle_Simulationsinstanz;

/* Bestimmung der Kontextinformationen für den Ausgang des Gatters */
Signalbund_Z ← Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund;
Z_Stimuli ← Signalbund_Z ∗ signalschritt + slot;
Z ← waves[Z_Stimuli + transition];

/* Bestimmung der Kontextinformationen für den ersten Eingang des Gatters */
Signalbund_A ← Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund_A;
A_Stimuli ← Signalbund_A ∗ signalschritt + slot;
A ← Signalkarte[Gatter.z][slot].Fortschritt_A;
A_Last ← waves[A_Stimuli + transition];
/* Bestimmung der Kontextinformationen für den zweiten Eingang des Gatters */
Signalbund_B ← Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund_B;
B_Stimuli ← Signalbund_B ∗ signalschritt + slot;
B ← Signalkarte[Gatter.z][slot].Fortschritt_B;
B_Last ← waves[B_Stimuli + transition];

/* Der erste Stimulus innerhalb der Stimulifolgen definiert den initialen Zustand des
Simulationsfortschrittes */
A_Wert ← Dekodiere_Wert(waves[A]);
B_Wert ← Dekodiere_Wert(waves[B]);
Z_Wert ← GatterFkt(A_Wert, B_Wert, Gatter.op);

/* Der zweite Stimulus innerhalb der Stimulifolgen definiert die ersten auszuwertenden
Stimuli */
NextStimulus(A, Signalbund_A);
A_Zeit ← Dekodiere_Zeit(waves[A]);
NextStimulus(B, Signalbund_B);
B_Zeit ← Dekodiere_Zeit(waves[B]);

/* Bestimmung der Variationen im Verzögerungsmodell */
Variiere(Verzoegerungen, slot);
```

Auswertung von Stimulifolgen

Die Auswertung der Signalwerte von Stimuli ist in Algorithmus 14 dargestellt. Für jede Boolesche Funktion hält die datenparallele Implementierung von CWTSim eine Tabelle im Konstantenspeicher des Grafikprozessors bereit. Diese Tabellen enthalten 16 Werte, die jeweils einem Auswertungsszenario zugeordnet sind. Die Funktion *GatterFKT* erhält hierbei die dekodierten Signalwerte aus den Stimuli. Das Ergebnis der Dekodierung wird in einem 32-Bit-Register hinterlegt. Der Signalwert wird hierbei durch die beiden höchstwertigen Bits innerhalb dieses Registers dargestellt. Das Auswertungsszenario wird bestimmt, in dem die Bits, welche die Signalwerte darstellen, konkateniert werden. In der Folge entsteht ein Wert, der über vier Bit verfügt und somit sämtliche Signalwertkombinationen repräsentiert. Der entstehende Wert wird als Index innerhalb der Tabelle verwendet, aus dem das Auswertungsergebnis gelesen wird.

Die Verwendung des Konstantenspeichers ist effizient, da Inhalte des Speichers innerhalb eines Caches auf dem Grafikprozessor zwischengespeichert werden. Im Vergleich zur Nutzung des Hauptspeichers des Grafikprozessors werden die benötigten Zyklen reduziert.

Algorithmus 14 Auswertung von Stimuliwerten an einem Gatter unter Verwendung der vierwertigen booleschen Algebra

```
function GatterFkt(A_Wert, B_Wert, OP)
    Eingabe ← (A_Wert»28 ∨ B_Wert»30);
    switch OP
    case Puffer:
    case Und:
        return Und_Tabelle[Eingabe];
    case Nicht:
    case Nicht-Und:
        return Nicht-Und_Tabelle[Eingabe];
    case Oder:
        return Oder_Tabelle[Eingabe];
    case Nicht-Oder:
        return Nicht-Oder_Tabelle[Eingabe];
    case Exklusiv-Oder:
        return Exklusiv-Oder_Tabelle[Eingabe];
    case Nicht-Exklusiv-Oder:
        return Nicht-Exklusiv-Oder_Tabelle[Eingabe];
    endswitch
end function
```

Die zweite Phase des Auswertungsalgorithmus, in welchem die eigentliche Auswertung der Stimulifolgen durchgeführt wird, ist in Algorithmus 15 dargestellt. Dieser Algorithmus iteriert über die Stimulifolgen an den Eingängen des Gatters, bis eines der folgenden Ereignisse eintritt:

- Die Stimulifolge am Ausgang des Gatters ist vollständig mit Stimuli gefüllt. Es stehen für den übergeordneten Signalbund keine nachfolgenden Signalbünde bereit.
- Eine der beiden Stimulifolgen an den Eingängen des Gatters wurde vollständig ausgewertet. Darüber hinaus wurde geprüft, ob der Signalbund, welcher der Stimulifolge übergeordnet ist, nachfolgende Signalbünde besitzt.

Im Gegensatz zur Ausgangsimplementierung wird die Auswertung bereits abgeschlossen, wenn mindestens eine Stimulifolge an den Eingängen der Stimulifolgen vollständig ausgewertet wurde. Die übrigen nicht vollständig ausgewerteten Stimuli verbleiben, bis die vollständig ausgewertete Stimulifolge in einem nachfolgenden Simulationsdurchgang weitere Stimuli beinhaltet, unter denen die zuvor übrig gebliebenen Stimuli ausgewertet werden können. Diese Vorgehensweise ist in Abbildung 4.22 dargestellt.

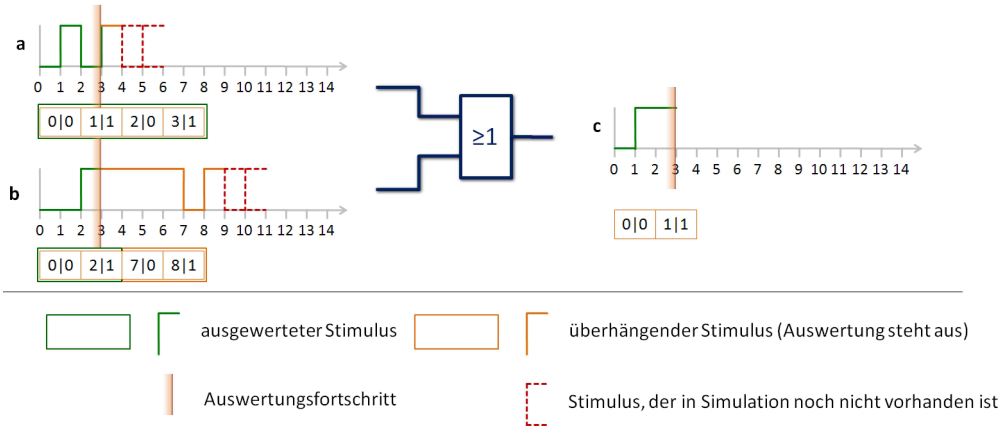


Abbildung 4.22: Veranschaulichung des vorzeitigen Abschluss der Auswertung

Eine nachfolgende Stabilität in der Stimulifolge, welche bereits vollständig ausgewertet wurde, darf in einer kontinuierlichen Simulation nicht angenommen werden. Durch diese Herangehensweise können Stimuli erzeugt werden, welche innerhalb späterer Simulationsdurchläufe ersetzt werden müssten. Die angenommene Stabilität innerhalb der Stimulifolge wird hierbei durch nachfolgend hinzugefügte Stimuli widerlegt.

Innerhalb der Iteration über die Stimulifolgen wird jeweils derjenige Stimulus betrachtet, welcher den gegenwärtigen minimalen Zeitpunkt aufweist. Der Signalwert des Stimulus sowie der gegenwärtige Signalwert der zweiten Stimulifolge wird anschließend unter der booleschen Funktion des Gatters ausgewertet. Falls diese Auswertung zu einem Signalwechsel am Ausgang des Gatters führt, wird der Zeitpunkt des auftretenden Signalwechsels bestimmt. Hierzu wird das Verzögerungsmodell des Gatters angewendet. Falls der ermit-

telte Zeitpunkt ein Stimulus erzeugt, welcher die zeitliche Mindestdistanz zum zuletzt eingefügten Stimulus erfüllt, wird der Stimulus in die Stimulifolge am Ausgang des Gatters eingefügt. Die zeitliche Mindestdistanz folgt aus dem Trägheitskriterium, welches für jedes Gatter explizit definiert ist. Falls ein Stimulus dieses Trägheitskriterium nicht erfüllt, folgt daraus, dass der aktuelle Stimulus sowie der zuvor eingefügte Stimulus einen Signalausschlag erzeugen, welcher vom Gatter nicht propagiert wird. In der Folge wird der zuvor eingefügte Stimulus entfernt. Da ausschließlich die direkte Folge von zwei Stimuli das Trägheitskriterium verletzen können, wird ein darauf nachfolgender dritter Stimulus ohne Prüfung des Trägheitskriteriums eingesetzt. Die Prüfung des Trägheitskriteriums wird daraufhin wieder fortgesetzt.

Ein Stimulus, welches nicht eingefügt wird, da es zu keinem Signalwechsel führt, zeigt die Stabilität des Signals auf, welche mindestens bis zum Eintritt des Stimulus gültig ist. Der Auswertungsalgorithmus von CWTSim dokumentiert diese Stabilitäten, um sie nach Abschluss der Auswertungsphase in der Stimulifolge am Ausgang des Gatters zu dokumentieren. Daher werden Zeitpunkt und Signalwert des jeweiligen letzten Stimulus, welches nicht eingefügt wurde, dokumentiert. Der Zeitpunkt wird um das Ausmaß des Trägheitskriteriums verringert. Dies verhindert Entfernungen von Stimuli, welche das Trägheitskriterium nicht verletzten.

Abschluss des Auswertungsalgorithmus

Die dritte Phase des Auswertungsalgorithmus, in welchem der erreichte Fortschritt innerhalb der Auswertung von Stimulifolgen für nachfolgende Simulationsdurchläufe dokumentiert wird, ist in Algorithmus 16 dargestellt. In einem ersten Schritt wird die Stabilität des Signals am Ausgang des Gatters festgehalten. Hierzu wird ein weiterer Stimulus am Ausgang eingefügt, der denselben Signalwert besitzt, wie der Stimulus, welcher zuletzt hinzugefügt wurde.

Diese Herangehensweise gewährleistet die Propagierung der Information, in wie fern ein Signal zeitlich stabil ist. In nachfolgenden topologischen Ebenen werden hierdurch Überhänge in der Auswertung von Stimulifolgen reduziert. Verkürzte Überhänge führen zu einem verringerten Bedarf an Speicher, der für Überhänge benötigt wird. Dies führt dazu, dass eine größere Anzahl von Signalbünden innerhalb des Stimuli-Arrays während einer Simulation zur Verfügung stehen.

Anschließend wird der erreichte Fortschritt, welcher innerhalb der Stimulifolgen erreicht wurde, in der Signalkarte dokumentiert. Für den Fortschritt am Ausgang des Gatters wird die Position des vorletzten hinzugefügten Stimulus dokumentiert. Der zuletzt hinzugefügte Stimulus kann in einer nachfolgenden Simulationsdurchführung durch ein Stimulus, welcher die Trägheitsbedingung verletzt, nachträglich entfernt werden. Daher wird der letzte Stimulus erzeugt, ohne dessen Existenz zu dokumentieren. In nachfolgenden topologischen Ebenen wird dieser Stimulus in der Folge nicht ausgewertet.

Algorithmus 15 Auswertungsphase des Algorithmus zur Auswertung von Stimulifolgen innerhalb der erweiterten Simulationsimplementierung

```
NextStimulus(Z, Signalbund_Z);
Letzte_Transistion ← Dekodiere_Zeit(waves[Z]);
m ← min(A_Zeit, B_Zeit);

while Z ≠ −1 ∧ A ≤ A_Last ∧ B ≤ B_Last do
    if A_Zeit < B_Zeit then
        A_Wert ← Dekodiere_Wert(waves[A]);
        NextStimulus(A, Signalbund_A);
        A_Zeit ← Dekodiere_Zeit(waves[A]);
    else
        B_Wert ← Dekodiere_Wert(waves[B]);
        NextStimulus(B, Signalbund_B);
        B_Zeit ← Dekodiere_Zeit(waves[B]);
    end if
    Neuer_Z_Wert ←GatterFkt(A_Wert, B_Wert, Gatter.op);

    /* Ein weiterer Stimulus wird eingefügt, falls ein Signalwechsel auftritt */
    if Neuer_Z_Wert ≠ Z_Wert then
        Neue_Z_Zeit ← m;

        /* Falls das aktuelle Signal das Trägheitskriterium erfüllt... */
        if Letzte_Transistion = 10^38 ∨ Neue_Z_Zeit − m ≥TRÄGHEITSSCHWELLE then
            /* ... wird der Signalwechsel dokumentiert */
            Z_Wert ← Neuer_Z_Wert;
            Letzte_Transistion ← Neue_Z_Zeit;
            waves[Z] =Kodiere(m, Neuer_Z_Wert);
            NextStimulus(Z, Signalbund_Z);
        else
            Letzte_Transistion ← 10^38;
            Schiebe Z um eine Stelle zurück;
            Z_Wert ← Dekodiere_Wert(waves[Z]);
        end if
    else
        /* Falls die Auswertung zu keinem Signalwechsel führt,
        wird die Stabilität des Signals dokumentiert */
        Kandidat_Letzter_Wert ← Neuer_Z_Wert;
        Kandidat_Letzte_Zeit ← m−TRÄGHEITSSCHWELLE;
    end if

    m ← min(A_Zeit, B_Zeit);

end while
```

Falls der Stimulus in nachfolgenden Simulationsdurchläufen entfernt wird, ist es nicht erforderlich, weitere Stimuli ebenfalls zu entfernen, welche durch diesen Stimulus bedingt, in nachfolgenden topologischen Ebenen erzeugt wurden. Diese Problematik ist in Abbildung 4.23 dargestellt.

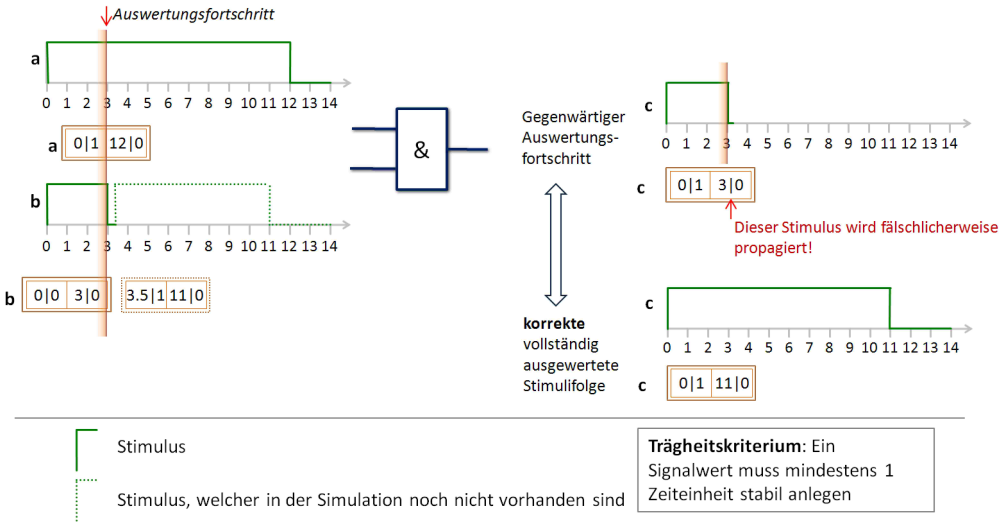


Abbildung 4.23: Veranschaulichung der entstehenden Problematik durch eine Propagierung des letzten erzeugten Stimulus

Algorithmus 16 Dokumentation des Fortschrittes innerhalb des Algorithmus zur Auswertung von Stimulifolgen

```
/* Dokumentation von stabilen Signalwerten, welche in der Auswertungsphase nicht
erfasst wurden */
if Letzte_Transistion ≠ 10^38 ∧ Z ≠ −1 then
    if Z_Wert =Kandidat_Letzter_Wert∧Letzte_Transition <Kandidat_Letzte_Zeit then
        waves[Z] ←Kodiere(Kandidat_Letzte_Zeit,Z_Wert);
        NextStimulus(Z, Signalbund_Z);
    end if
end if
/* Dokumentation des Fortschrittes innerhalb der Auswertung - A_Prev, B_Prev, Z_Prev
entsprechen dem vorangegangenen Index von A, B und Z */
Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund_A ← Signalbund_A;
Signalkarte[Gatter.z][slot].Fortschritt_A ← A_Prev;

Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund_B ← Signalbund_B;
Signalkarte[Gatter.z][slot].Fortschritt_B ← B_Prev;

Signalkarte[Gatter.z][slot].Letzter_betrachter_Signalbund ← Signalbund_Z;
waves[Z_Stimuli + transition] ← Z_Prev;
```

Falls ein Überlauf in der Stimulifolge am Ausgang des Gatters auftritt, wird zunächst geprüft, ob ein weiterer Signalbund mit dem übergeordneten Signalbund verknüpft ist, um innerhalb des weiteren Signalbundes die Auswertung fortzusetzen. Ist dies nicht der Fall, wird die Auswertung vorzeitig unterbrochen. Anschließend wird der erreichte Fortschritt in der Auswertung dokumentiert. Der Überlauf wird durch dieselben Methoden behandelt, welche die Überläufe innerhalb der kontinuierlichen Hinzufügung von Eingaben behandeln. Zunächst werden die einzelnen Überläufe innerhalb sämtlicher Stimulifolgen für den zugehörigen Signalbund bestimmt. Falls in einem Signalbund ein Überlauf aufgetreten ist, werden im Anschluss verfügbare Signalbünde innerhalb des Stimuli-Array gesucht und an die verkettete Liste der Signalbünde eines Signals angehängt. Sämtliche Stimulifolgen, welche in den neu hinzugewonnenen Signalbünden angelegt wurden, werden abschließend initialisiert. Diese Vorgehensweise ist in Abbildung 4.24 dargestellt.

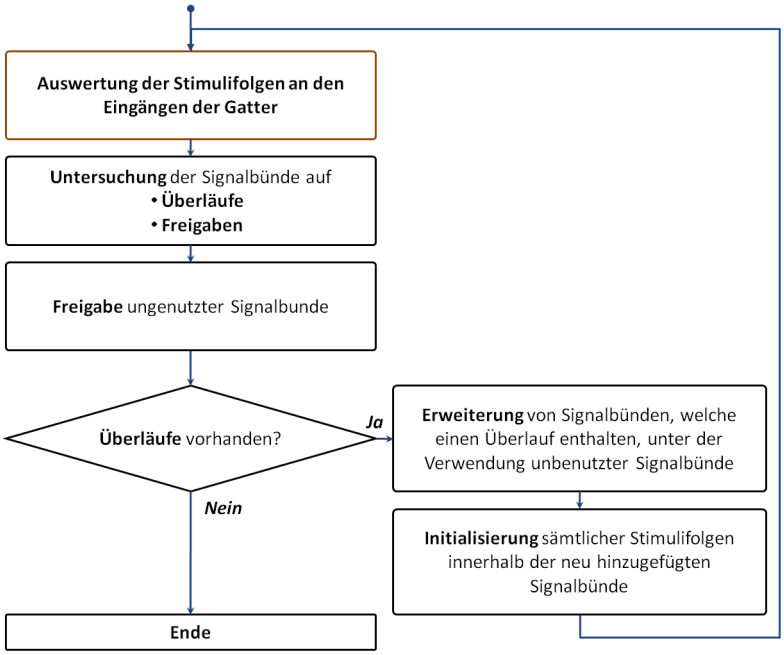


Abbildung 4.24: Auswertung von Stimulifolgen im Kontext der Verwaltung der zugehörigen Datenstrukturen

Datenparallele Implementierung des Auswertungsalgorithmus

CWTSim erhält die Beschreibung einer Schaltung in ihrer topologisch sortierten Repräsentation. Wie zuvor erläutert, werden die Funktionalitäten zur Repräsentation von Schaltungen sowie zur topologischen Sortierung von Gattern innerhalb des ADAMA-Frameworks bereitgestellt. Zwischen Gattern, welche in einer topologischen Ebene zusammengefasst sind, existieren keine Abhängigkeiten. Die datenparallele Implementierung von CWTSim führt daher den Auswertungsalgorithmus parallel auf sämtlichen Gattern einer topologischen Ebene aus. Darüber hinaus werden sämtliche Simulationsinstanzen, die von einem Gatter existieren, parallel ausgeführt.

Zwischen den einzelnen topologischen Ebenen existieren Abhängigkeiten. Stimulifolgen, welche in einer topologischen Ebene erzeugt werden, können erst darauf folgend an nachfolgenden topologischen Ebenen ausgewertet werden. CWTSim betrachtet daher einzelne topologische Ebenen sequentiell. Die Parallelität innerhalb Ausführung des Auswertungsalgorithmus ist in Abbildung 4.25 dargestellt.

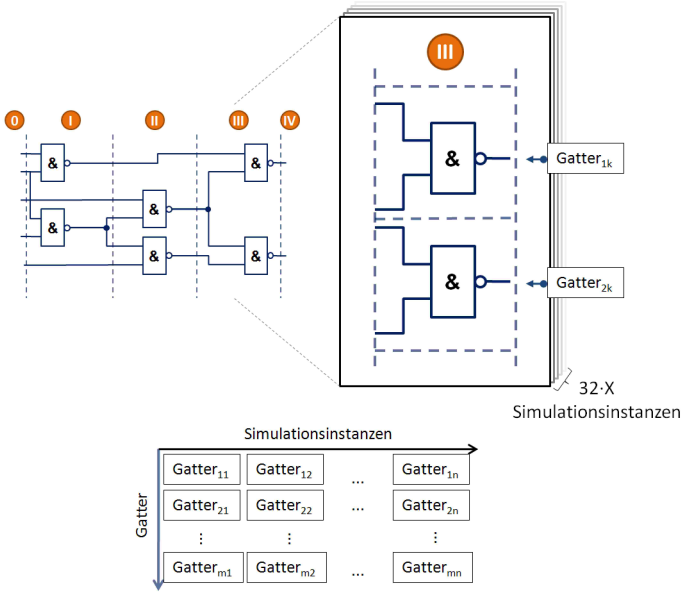


Abbildung 4.25: Veranschaulichung der parallelen Auswertung von Stimulifolgen innerhalb von topologischen Ebenen und Simulationsinstanzen

4.3.4 Ausgabe von Ergebnissen einer Simulation

Zur Ausgabe von Stimuli, welche an die Ausgängen der Schaltung propagiert wurden, existiert ein Ausgabenpuffer, der eine analoge Form des Eingabenpuffers darstellt. Dieser wurde in Kapitel 4.3.1 vorgestellt. Der Ausgabenpuffer verfügt für jeden Ausgang der Schaltung sowie für jede einhergehende Simulationsinstanz eine Stimulifolge. CWTSim verfügt über Funktionalitäten, mit denen der Ausgabepuffer gefüllt wird. Von diesem Ausgabenpuffer aus können die Simulationsergebnisse gelesen und innerhalb der Schnittstelle zu ADAMA weiterverarbeitet werden.

4.3.5 Finalisierung des Simulationsablaufs

Der zuvor vorgestellte Auswertungsalgorithmus geht während der Auswertung von Stimulifolgen von der Annahme aus, dass fortlaufend zu jedem Zeitpunkt weitere Stimuli eingefügt werden können. Dieser Algorithmus ist dafür konzipiert, in jedem Fall weitere Simulationsdurchläufe zu erwarten. Ein trivialer Abschluss der Simulation ist nicht möglich.

Eine Simulation wird abgeschlossen, in dem sämtliche Überhänge, die während der Simulation erzeugt wurden, vollständig ausgewertet werden. Hierzu müsste der Auswertungsalgorithmus die Signalstabilität innerhalb jeder vollständig ausgewerteten Stimulifolge annehmen. Eine Methode, die den Abschluss der Simulation gewährleistet, setzt weitere algorithmische Details in den Auswertungsalgorithmus ein. Hierdurch würde der Auswertungsalgorithmus um einen Modus ergänzt, in dem die Stimulifolgen vollständig ausgewertet werden.

In einer kontinuierlichen Simulation ist der Abschluss einer Simulation ein Ereignis, welches signifikant seltener auftritt, als die zuvor vorgestellte Fortsetzung der Auswertung. Sämtliche hinzugefügte algorithmische Details würden ausschließlich zum Abschluss der Simulation benötigt werden. In den überwiegenden Ausführungsszenarien würden diese algorithmischen Details zu Effizienzeinbußen führen, da ein möglicher Moduswechsel zu Beginn jeder Ausführung geprüft werden müsste.

CWTSim schließt eine Simulation ab, in dem es in sämtliche Stimulifolgen an den Eingängen einer topologischen Ebene ein Stimulus einsetzt, dessen Zeitpunkt weit über den Zeitpunkten liegt, welche in der praktischen Anwendung verwendet werden. Der zuvor eingefügte Stimulus innerhalb jeder Stimulifolge wird hierdurch implizit stabil, da dessen Signalwert bis zu dem Zeitpunkt des Stimulus reicht, welches von CWTSim eingesetzt wurde. Anschließend wird ein Simulationsdurchlauf durchgeführt, in dem sämtliche Überhänge durch die speziell eingesetzten Stimuli ausgewertet werden. Beim Auslesen der vollständig ausgewerteten Stimulifolgen an den Ausgängen der Schaltung werden die speziell eingefügten Stimuli herausgefiltert.

5 Validierung

5.1 Übersicht

Die Korrektheit von CWTSim wurde durch eine Validierung festgestellt. Innerhalb dieser Validierung wurden die Simulationsergebnisse von CWTSim mit den Simulationsergebnissen des kommerziellen Simulationswerkzeugs *ModelSim* verglichen. Eine Übersicht zur Validierung in Abbildung 5.1 dargestellt.

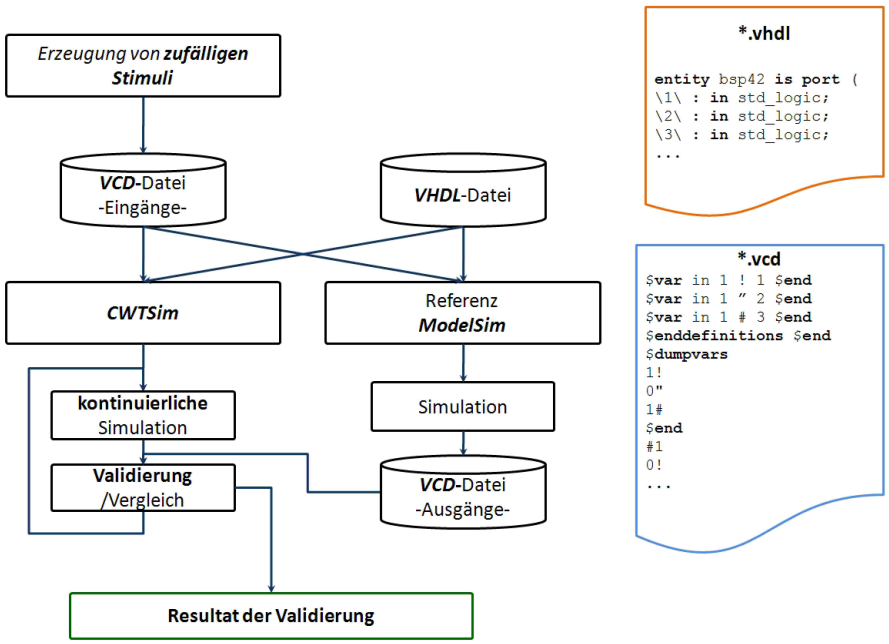


Abbildung 5.1: Übersicht der Validierung

Die Schnittstelle zwischen CWTSim und *ModelSim* wurde über Dateien hergestellt, welche im VCD-Format (*Value Change Dump*) angelegt werden. Das VCD-Format ist im IEEE-Standard 1364 festgehalten [39]. Dieses Format beschreibt vierwertige Stimulifolgen unter der Verwendung von ASCII-Zeichen. Für sämtliche Ein- und Ausgänge einer Schaltung lassen sich beliebige Stimulifolgen definieren. Innerhalb der Validierung werden VCD-Dateien an zwei Stellen eingesetzt, die im Folgenden näher erläutert werden.

Die Validierung beginnt mit der Erzeugung von zufälligen Stimulifolgen, die als Eingabemenge für die Validierung verwendet werden. Der Task CWTSim besitzt hierzu einen

Modus, welcher zufällige Stimulifolgen erzeugt und in einer VCD-Datei hinterlegt. Der entsprechende Kommandozeilen-Befehl ist im Folgenden dargestellt.

```
Syntax:
./adama cwtsim <Schaltung> -vcdfile <Ausgabe VCD-Datei>
-vcdts <Simulationszeitraum>
-vcdmts <maximaler zeitlicher Abstand zwischen Stimuli> -opmode 0

Beispiel:
./adama cwtsim c17.lg -vcdfile c17.vcd -vcdts 10000000 -vcdmts 10 -opmode 0
```

Die erzeugte VCD-Datei wird im Anschluss von *ModelSim* als Eingabe zur Simulation verwendet, wobei erzeugte Ausgaben in eine weitere VCD-Datei geschrieben werden. Diese VCD-Datei dient CWTSim als Grundlage zur Validierung der Simulationsergebnisse. Im Folgenden sind die Kommandozeilen-Befehle aufgeführt, die innerhalb von *ModelSim* erforderlich sind, um eine Simulation, samt einhergehender Umleitung der Ergebnisse in eine VCD-Datei, durchzuführen.

```
Syntax:
>vsim -t <Zeitauflösung> -vcdread <VCD-Datei> <Schaltungsentität>
>vcd file <VCD-Datei für die Ausgaben>
>vcd add -out <Schaltungsentität>/*
>run -all

Beispiel:
>vsim -t ns -vcdread c17.vcd c17
>vcd file c17o.vcd
>vcd add -out c17/*
>run -all
```

Im Anschluss daran führt CWTSim eine kontinuierliche Simulation auf Grundlage der VCD-Datei aus, in denen die zufälligen Stimulifolgen definiert sind. Hierbei werden fortlaufend kontinuierlich erzeugte Stimulifolgen mit den Simulationsergebnissen von ModelSim verglichen. Im Folgenden sind die Kommandozeilen-Befehle aufgeführt, die eine Validierung in CWTSim veranlassen. Als Eingaben werden hierbei die VCD-Datei, welche die zufälligen Stimulifolgen enthält, sowie die VCD-Datei, welche die Simulationsergebnisse von *ModelSim* enthält, als Parameter übergeben.

```
Syntax:
./adama cwtsim <Schaltung> -k <Pfad zur datenparallelen Implementierung>
 -vcdfile <VCD-Datei -Eingaben-> -vcdreffile <VCD-Datei -Referenzausgaben->
 -opmode 2

Beispiel:
./adama cwtsim c17.lg -k cwt_cuda_sm20:0 -vcdfile c17.vcd
 -vcdreffile c17o.vcd -opmode 2
```

5.2 Vorgehensweise der Validierung

CWTSim führt die Validierung der Simulation kontinuierlich durch. Hierzu werden die Eingabestimuli fortlaufend simuliert und mit Referenzstimuli verglichen. Der Ablauf der einzelnen Validierungsschritte ist in Abbildung 5.2 dargestellt. Eine Validierung beginnt mit dem Einlesen von Eingabe- und Referenzstimuli. Nach der Initialisierung der Simulation werden die initial propagierten Signalwerte mit den Referenzstimuli verglichen. Daraufhin werden in jedem Validierungsdurchlauf weitere Eingabestimuli sowie Referenzstimuli gelesen, sofern sämtliche vorhandene Stimuli validiert wurden. In jedem Durchlauf werden hierbei die eingelesenen Referenzstimuli mit den gegenwärtig simulierten Stimuli verglichen. Sollten sich Stimuli unterscheiden, ist die Validierung nicht bestanden.

Die Validierung ist bestanden, sofern sämtliche Stimuli zwischen CWTSim und der Referenz äquivalent sind und darüber hinaus beide Simulationsimplementierungen die gleiche Anzahl von Stimuli erzeugt haben.

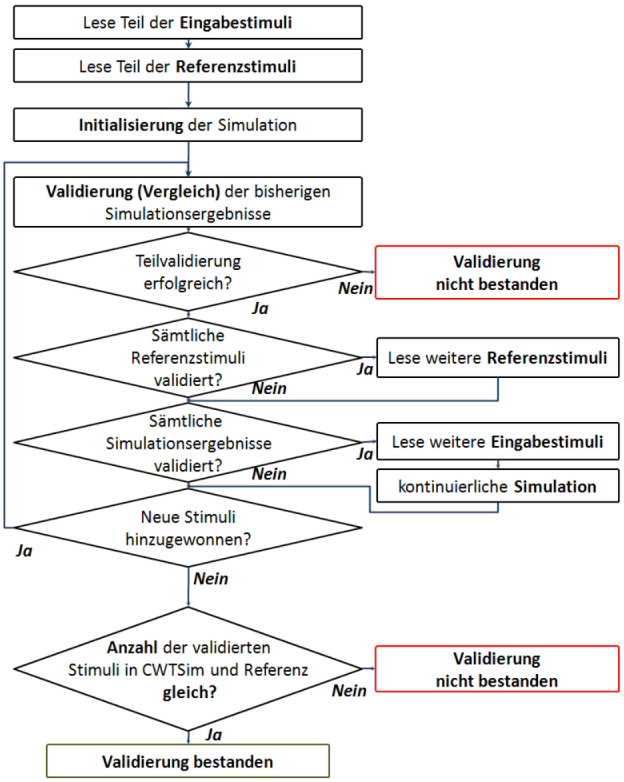


Abbildung 5.2: Ablauf der Validierung

6 Analyse

Das folgende Kapitel untersucht das Laufzeitverhalten der entwickelten kontinuierlichen Simulation CWTSim. Für jede Schaltung wurde eine Reihe von Simulationen durchgeführt, die sich in ihrer Konfiguration voneinander unterscheiden. Eine Konfiguration entsteht durch die Kombination zweier Simulationsparameter:

1. Die Anzahl der Stimuli pro Stimulifolge
2. Die Anzahl der parallel ausgewerteten Simulationsinstanzen

In jeder Simulation wurden eine Million zufällige Stimuli für jeden Eingang der untersuchten Schaltung erzeugt und kontinuierlich simuliert. Die erzeugten Stimulifolgen wurden in VCD-Dateien gespeichert, die als Eingabe an CWTSim sowie an *ModelSim* übergeben wurden. Für jede untersuchte Schaltung wurde eine Simulation über 54 Konfigurationen durchgeführt, welche in Tabelle 6.1 dargestellt sind. Hierbei wurden neun verschiedene Parameter für die Größe einer Stimulifolge, sowie sechs verschiedene Parameter für die Anzahl der Simulationsinstanzen verwendet. In Abhängigkeit von der Konfiguration wird der Speicherbedarf im initialen Zustand einer Simulation bestimmt. In Konfigurationen, in denen die Anzahl der Simulationsinstanzen mehr als den gesamten verfügbaren Speicher der Zielarchitektur in Anspruch nehmen, konnte die parallele Auswertung sämtlicher Simulationsinstanzen nicht durchgeführt werden. In diesem Fall muss die Menge der Simulationsinstanzen in Partitionen aufgeteilt und nacheinander simuliert werden. Diese Fälle wurden in der nachfolgenden Analyse außen vor gelassen, da das Hauptaugenmerk der Analyse auf vollständig parallel durchgeführten Simulationen beruht.

		Anzahl Stimuli pro Stimulifolge								
		8	16	32	64	128	256	512	1024	2048
Simulationsinstanzen	32	*untersuchte Konfigurationen*								
	64									
	128									
	256									
	512									
	1024									

Tabelle 6.1: Übersicht zu den eingesetzten Konfigurationen

Die Untersuchung wurde auf vier Nvidia Tesla C2070 durchgeführt, welche jeweils über 6GB RAM verfügen. Die Spezifikationen zur Zielhardware sind in Kapitel 8.2 aufgeführt. Das System umfasst zwei Intel Xeon X5680-Prozessoren, die einen Takt von 3,33 GHz

besitzen. Der Hauptspeicher des Systems umfasst 96 GB RAM. Die einzelnen Simulationen wurden vollständig auf den Grafikprozessoren durchgeführt.

Die Beschleunigung gegenüber ModelSim wurde aus dem Quotienten der ermittelten Laufzeiten berechnet:

$$\text{Beschleunigung} = \frac{t_{ModelSim}}{t_{CWTSim}}$$

6.1 Übersicht der betrachteten Schaltungen

In Tabelle 6.2 sind die Eigenschaften der betrachteten Schaltungen dargestellt. Darüber hinaus ist zu jeder Schaltung die größte erreichte Beschleunigung angegeben. Die untersuchten Schaltungen stammen aus der Menge der ISCAS'85 und ISCAS'89-Schaltungsentwürfe sowie aus der Menge der industriellen Schaltungsentwürfe, die von NXP stammen. Bei der Auswahl der Schaltungen wurde eine große Bandbreite an Schaltungsgrößen angestrebt.

Schaltung	Gatter	Eingänge	Ausgänge	Ebenen	Max. Beschleunigung
C17	6	5	2	3	27,9x
S400	214	24	27	13	55,9x
C432	216	36	7	29	53,1x
C499	246	41	32	14	45,0x
C1908	1057	33	25	44	137,3x
C2670	1476	233	140	39	97,3x
C5315	2973	178	123	52	119,6x
C7552	4043	207	108	45	167,8x
S9234	5944	247	250	61	98,1x
S15850	10211	611	684	87	72,7x
S38584	21462	1464	1730	59	52,1x
P100k	96685	5902	5829	105	45,1x
P81k	108991	4029	3952	55	61,1x
P279k	287935	18074	17831	150	26,8x

Tabelle 6.2: Übersicht über betrachtete Schaltungen

6.2 Untersuchung zur minimalen Anzahl von parallelen Simulationsinstanzen

Zunächst wurde das Laufzeitverhalten von CWTSim unter der minimalen Anzahl von parallelen Simulationsinstanzen untersucht. Die Ergebnisse dieser Untersuchung sind in Abbildung 6.1 dargestellt. Die minimale Anzahl von Simulationsinstanzen, die CWTSim

parallel berechnen kann, beträgt 32. Diese Zahl leitet sich aus der Granularität der Ausführungseinheiten innerhalb des Grafikprozessors ab. Die Datenstrukturen einer Simulation wurden zur Aufnahme von 32 Stimuli pro Stimulifolge eingestellt.

Ein Ergebnis dieser Untersuchung ist die Beobachtung, dass CWTSim mit steigender Anzahl von Gattern innerhalb einer Schaltung eine zunehmende Beschleunigung erreicht. Da in sämtlichen Fällen die Anzahl der parallelen Simulationsinstanzen gleich ist, begründet sich die zunehmende Beschleunigung innerhalb der Simulationsinstanzen wie folgt: In jeder Simulationsinstanz werden die Gatter einer topologischen Ebene parallel ausgeführt. In der Untersuchung der Schaltung C17, welche mit sechs Gattern äußerst klein ist, wurde eine Beschleunigung von $1,2$x ermittelt. Den Kontrast bildet die Schaltung P81k, welche 108991 Gatter besitzt. Hierbei wurde die maximale Beschleunigung von $29,6$x ermittelt.

Kleine Schaltungen nutzen nur geringe Teile des Grafikprozessors. In der Folge fallen die Effizienzvorteile signifikant geringer aus, als bei Schaltungen, die eine vergleichsweise große Anzahl von Gattern besitzen. In der Simulation dieser großen Schaltungen werden sämtliche Ausführungseinheiten des Grafikprozessors ausgelastet, wodurch die einhergehende Parallelität die Laufzeit gegenüber der sequentiellen Auswertung (ModelSim) reduziert.

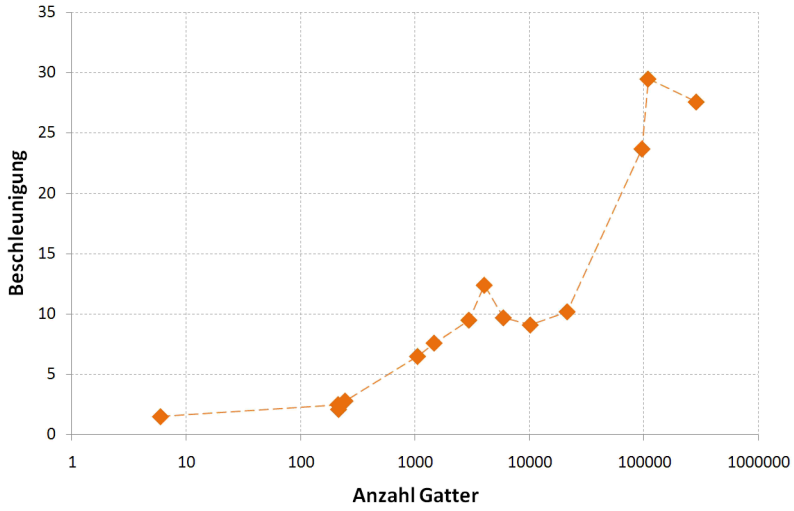


Abbildung 6.1: Beschleunigungen in der Konfiguration 32 Stimuli pro Stimulifolge und 32 parallele Simulationsinstanzen

6.3 Untersuchung der maximal erreichten Beschleunigung

Weiterhin wurde das Laufzeitverhalten von CWTSim unter der maximalen Anzahl von parallelen Simulationsinstanzen untersucht. Die Ergebnisse dieser Untersuchung sind in Abbildung 6.2 dargestellt. Die maximale Anzahl von Simulationsinstanzen, die angesetzt

wurde, beträgt 2048. In sämtlichen Simulationen wurden bis zu 1024 Stimuli pro Stimulifolge angelegt. Wie zuvor erläutert, wächst mit zunehmender Größe der Schaltungen die Größe des Speichers, welcher benötigt wird, um sämtliche Signalbünde unter einer gegebenen Konfiguration darstellen zu können. In der Mehrheit der betrachteten Schaltungen lag die größte Konfiguration, welche vollständig im Speicher der Zielarchitektur abgebildet werden konnte, unter der größten gewählten Konfiguration (2048 Simulationsinstanzen und 1024 Stimuli pro Stimulifolge).

Im Vergleich zur sequentiellen Auswertung wird die Laufzeit signifikant reduziert. Die kleinste ermittelte Beschleunigung beträgt 26,8x in der Schaltung P279k. Die höchste Beschleunigung wurde für Schaltung C7552 ermittelt. Hier wurde eine Beschleunigung von 167,8x ermittelt. Die Schaltung C7552 enthält 4043 Gatter. Die Konfiguration, die diese höchste Beschleunigung erzielt hat, enthält 256 parallele Simulationsinstanzen sowie 256 Stimuli pro Stimulifolge. Diese liegt somit unter der zuvor erläuterten größten gewählten Konfiguration.

Mit fortlaufender Erhöhung der Anzahl von Gattern verringert sich die größtmögliche Konfiguration. In der Folge verringert sich die größte untersuchte Anzahl von parallelen Simulationsinstanzen sowie die größte Anzahl von Stimuli pro Stimulifolge. In Tabelle 6.3 ist die maximale Anzahl an parallel durchgeführten Instanzen dargestellt. Diese maximal erreichte Anzahl von parallelen Simulationsinstanzen sinkt fortlaufend mit steigender Anzahl von Gattern in einer Schaltung. In der Simulation der Schaltung C7552, an welcher die größte Beschleunigung gemessen wurde, konnten hierbei lediglich 256 parallele Simulationsinstanzen betrachtet werden.

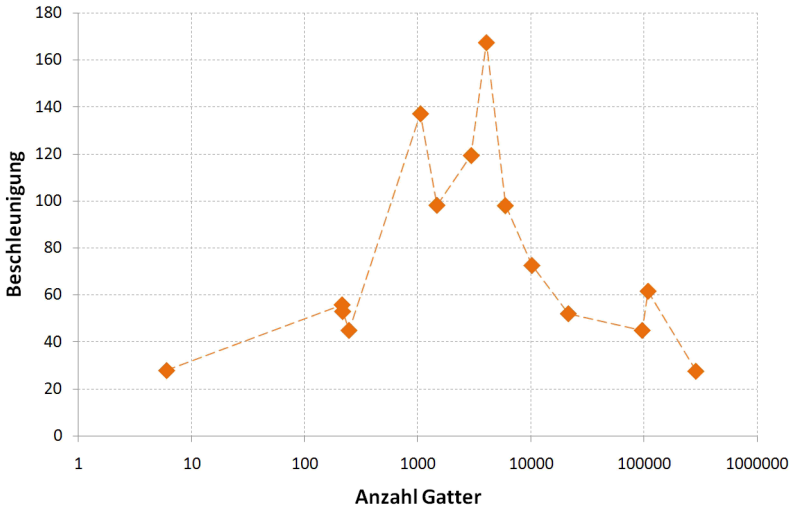


Abbildung 6.2: Vergleich der maximal erreichten Beschleunigungen in Abhängigkeit der Anzahl von Gatter

Schaltung	Gatter	Max. parallele Simulations-instanzen
C17	6	2048
S400	214	1024
C432	216	1024
C499	246	512
C1908	1057	256
C2670	1476	256
C5315	2973	256
C7552	4043	256
S9234	5944	128
S15850	10211	128
S38584	21462	64
P100k	96685	32
P81k	108991	32
P279k	287935	32

Tabelle 6.3: Übersicht zu maximalen parallelen Simulationsinstanzen

6.4 Detaillierte Auswertung

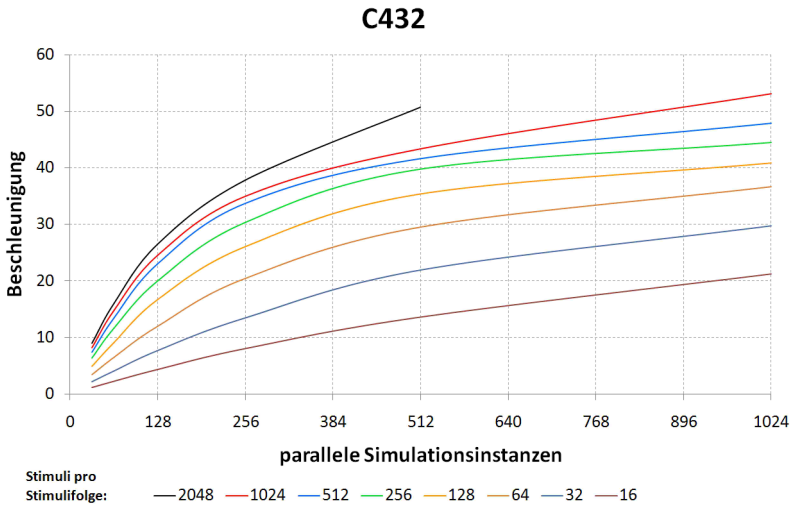
Im Folgenden werden die Beschleunigungen von sämtlichen Konfigurationen in den untersuchten Schaltungen diskutiert. Die Ergebnisse sind in den nachfolgenden Abbildungen sowie darüber hinaus in Kapitel 8.3 dargestellt. Die Ergebnisse der Schaltungen P81k, P100k sowie P279k wurden außen vor gelassen, da die Menge der durchführbaren Konfigurationen in diesen Schaltungen einen zu geringen Umfang aufweist.

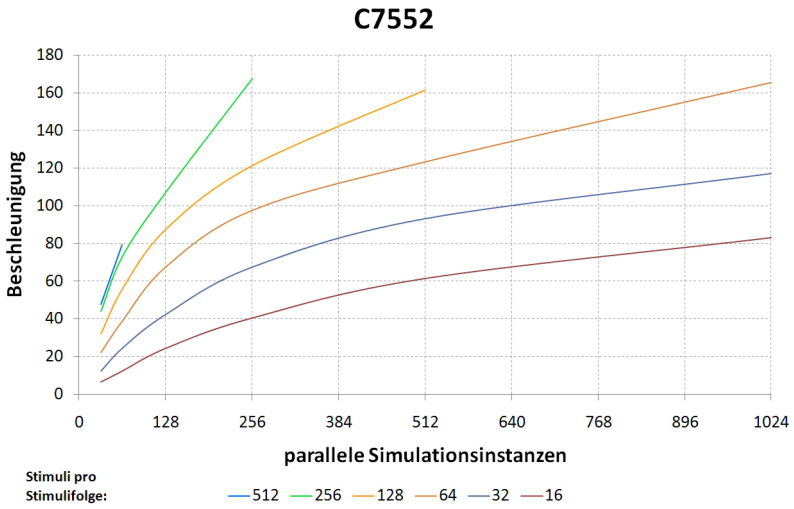
In dieser Untersuchung lässt sich ein Verhalten beobachten, welches in sämtlichen untersuchten Schaltungen auftritt. Mit steigender Anzahl von parallel ausgewerteten Simulationsinstanzen steigt die Beschleunigung proportional an. Wie zuvor erläutert, begründet sich dies in der zunehmenden Ausnutzung sämtlicher Ausführungseinheiten des Grafikprozessors, wodurch eine höhere Auslastung gewährleistet wird.

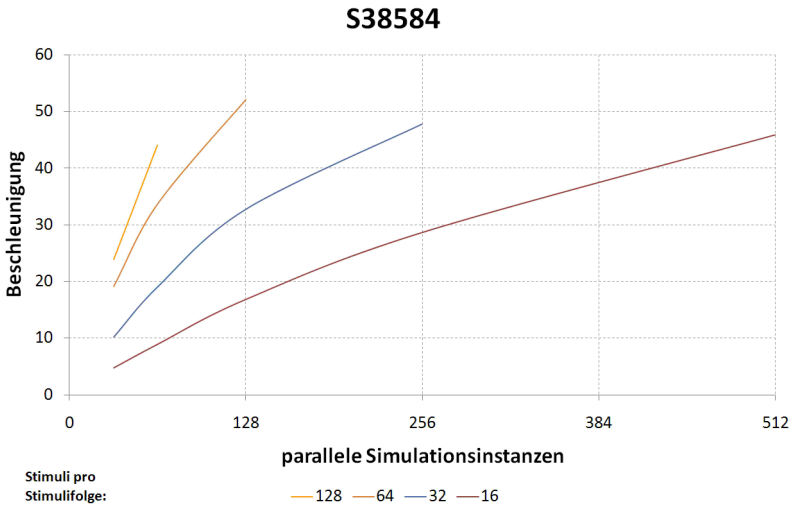
In Konfigurationen, in denen eine Vielzahl von Simulationsinstanzen und umfangreichen Stimulifolgen gewählt wurden, kann eine obere Grenze der Beschleunigung beobachtet werden. Diese resultiert aus der Begrenzung der Bandbreite zum Hauptspeicher des Grafikprozessors. Insgesamt erfährt die Beschleunigung eine Sättigung, da der verfügbare Speicher Konfigurationen einschränkt, in denen größere Anzahlen von parallelen Simulationsinstanzen verwendet werden. Dies resultiert in einer verringerten Anzahl von Stimuli pro Stimulifolge sowie parallel ausgewerteten Simulationsinstanzen.

Darüber hinaus steigt die Beschleunigung mit zunehmender Anzahl von Stimuli pro Stimulifolge an. Dies begründet sich darin, dass größere Stimulifolgen die Anzahl an

benötigten Stimulifolgen verringert, die erzeugt werden müssen, um sämtliche Stimuli eines Signals vollständig erfassen zu können. Mit der steigenden Anzahl von Stimuli in einer Stimulifolge sinkt die Anzahl der Überläufe innerhalb des Auswertungsvorgangs. In der Folge verringert sich der Verwaltungsaufwand, welcher zur Behandlung der Überläufe nötig ist. Aus der verringerten Laufzeit, welche hieraus folgt, resultiert eine Erhöhung der Beschleunigung.







6.5 Zusammenfassung der Ergebnisse

Aus der Untersuchung zur minimalen Anzahl von parallelen Simulationsinstanzen geht hervor, dass CWTSim mit steigender Anzahl von Gattern eine zunehmende Beschleunigung erreicht. Die größte ermittelte Beschleunigung beträgt in dieser Untersuchung 29,6x. Innerhalb dieser Untersuchung hat die limitierte Speichergröße keinen Einfluss auf die ausgewerteten Simulationen.

Im Gegensatz hierzu übt die limitierte Speichergröße einen erheblichen Einfluss auf die Untersuchung der maximal erreichten Beschleunigung aus. Hierdurch konnten Konfigurationen, welche über eine sehr große Anzahl von Simulationsinstanzen sowie Stimuli pro Stimulifolge verfügen, mit steigender Anzahl von Gattern nicht ausgewertet werden. Die größte ermittelte Beschleunigung beträgt 167,8x gegenüber der sequentiellen Auswertung.

Aus der detaillierten Auswertung geht hervor, dass die Beschleunigung mit der Anzahl der Stimuli pro Stimulifolgen sowie mit der Anzahl der parallel ausgewerteten Simulationsinstanzen steigt. Darüber hinaus wurde eine Limitierung der Beschleunigung beobachtet, die aus einer Kombination von begrenzter Speicherbandbreite sowie begrenztem Speicherplatz entsteht.

7 Zusammenfassung

Die Validierung von Schaltungsentwürfen nimmt bis zu 70 Prozent der Entwurfsdauer in Anspruch. Validierungsaufgaben wie Fehlersimulationen, Alterungsanalysen, Untersuchungen zum Energieverbrauch, Testmengencharakterisierungen sowie die Bewertung der Zuverlässigkeit erfordern hochperformante verzögerungsbehaftete Logiksimulationen. Die Verzögerungen innerhalb hochintegrierter Schaltungen sind von Variationen geprägt. Die Berücksichtigung von Variationen innerhalb der Validierungsaufgaben erhöht den Aufwand nochmals erheblich.

Die Ausgangsimplementierung einer verzögerungsbehafteten Logiksimulation, die in der vorliegenden Arbeit detailliert untersucht wurde, besitzt Beschränkungen, welche die Anwendbarkeit innerhalb der vorgenannten Validierungsaufgaben reduzieren. Eine wesentliche Beschränkung der Ausgangsimplementierung ist die Verwendung von endlichen Stimulifolgen, die eine effiziente Simulation über einen potentiell unbegrenzten Zeitraum nicht ermöglicht.

In der vorliegenden Arbeit wurde die Simulationsimplementierung *CWTSim* entwickelt, welche die Anforderungen der Validierungsaufgaben erfüllt und die Beschränkungen der Ausgangsimplementierung auflöst. CWTSim verfolgt den Ansatz einer kontinuierlichen Simulation von Stimulifolgen auf Gatterebene. In einer kontinuierlichen Simulation werden Stimulifolgen iterativ simuliert, wobei es nicht erforderlich ist, gesamte Stimulifolgen im Kontext der Simulation vorzuhalten. In der Folge ermöglicht CWTSim die kontinuierliche Simulation von potentiell unbegrenzten Stimulifolgen. Darüber hinaus ist CWTSim fähig, verschiedene Simulationsinstanzen parallel zu simulieren. Jede Simulationsinstanz kann hierbei eine verzögerungsbehaftete Schaltung repräsentieren, in der Variationen das Verzögerungsverhalten beeinflussen.

CWTSim wurde unter den Kriterien einer größtmöglich erreichbaren Effizienz entwickelt. Die Implementierung wurde parallelisiert und auf eine datenparallele Architektur abgebildet. Hierzu wurde die CUDA-Architektur eingesetzt, welche die Verwendung des GPGPU-Modells abstrahiert. Neben der parallelen Ausführung verschiedener Simulationsinstanzen wird innerhalb jeder Simulationsinstanz eine feingranulare Parallelisierung angewandt. CWTSim fasst die Auswertung an den Gattern innerhalb einer topologischen Ebene zusammen. Sämtliche Gatter, die sich innerhalb einer topologischen Ebene befinden, besitzen zueinander keine Abhängigkeiten und werden daher parallel ausgewertet.

Der Simulationsansatz von CWTSim leitet sich gleichermaßen aus der Kategorie der ebenenorientierten Simulation (*plain simulation*) sowie aus der Kategorie der ereignisgesteuerten Simulation (*event-driven simulation*) ab. Analog zu einer ebenen-orientierten Simulation wertet CWTSim sämtliche Gatter in den topologischen Ebenen aus. Der Auswertungsumfang hängt von der Menge und Art der Stimuli an den Eingängen der Gatter ab, was dem

Ansatz einer ereignisgesteuerten Simulation folgt.

Zur Optimierung von CWTSim wurden eine Reihe verschiedener Methoden angewandt. Die Darstellung von Stimuli wurde optimiert, so dass die Kontextinformationen eines Stimulus in einem kompakten Datentyp zusammenfasst sind. In der Folge wird die Anzahl der erforderlichen Speicherzugriffe reduziert. In einem weiteren Ansatz wurde die Anordnung der Stimuli innerhalb des Speichers optimiert. Hierdurch werden räumliche und zeitliche Kohärenzen bei Speicherzugriffen erzeugt, welche in der Ausführung effizient genutzt werden und Zugriffslatenzen reduzieren. Für eine effiziente Wiederaufnahme der Auswertung, welche nach jeder Hinzufügung von weiteren Stimuli durchgeführt wird, wurde eine Datenstruktur entwickelt, die den Auswertungsfortschritt innerhalb jedes Signals effizient erfasst. Hierdurch werden aufwendige Methoden zur Suche nach Auswertungsfortschritten vermieden. Für die parallele Erweiterung von Stimulifolgen wurde ein Ansatz entwickelt, der eine verteilte Herangehensweise zur Suche nach freien Speicherbereichen anwendet. In der Folge werden Konflikte in der parallelen Suche vermieden und somit die Anzahl von Suchdurchläufen reduziert. Weiterhin wurde die Auswertung von Stimulifolgen darauf optimiert, die Verwendung von Caches in der Speicherhierarchie implizit zu berücksichtigen, was ebenfalls zur Reduzierung von Latenzen innerhalb von Speicherzugriffen beiträgt.

Die benötigte Zeit, welche zur Simulation einer Vielzahl von Verzögerungsvariationen aufgewendet werden muss, wird durch CWTSim signifikant reduziert. CWTSim erreicht Beschleunigungen bis zu 168x im Vergleich zur sequentiellen Analyse mit einem kommerziellen Simulationswerkzeug.

8 Anhang

8.1 Befehlsumfang des ADAMA-Task CWTSim

Im Folgenden wird der Befehlsumfang des ADAMA-Task CWTSim vorgestellt.

Parameter	**Beschreibung**
-k	Pfad zur datenparallelen Implementierung. Die Angabe von 'java' führt zur Ausführung der sequentiellen Referenzimplementierung
-m	Umfang des Speichers, welcher in der Simulation verwendet wird
-c	Anzahl der Stimuli pro Stimulifolge
-i	Anteil des Speichers, der während der Initialisierung zur Erzeugung von Signalbünden verwendet werden darf
-d	Verzögerung sämtlicher Gatter. Alternativ kann ein Pfad zu einer Datei angegeben werden, in der einzelne Verzögerungsparameter hinterlegt sind.
-f	Trägheitsschwelle sämtlicher Gatter
-opmode 0	Erzeugung einer VCD-Datei, die zufällige Stimulifolgen für jeden Eingang einer Schaltung beinhaltet
-opmode 1	Ausgabe einer VCD-Datei
-opmode 2	Validierung von CWTSim gegen die Ergebnisse eines externen Simulationswerkzeugs
-opmode 3	Validierung der datenparallelen Implementierung gegen die sequentielle Implementierung innerhalb von CWTSim
-opmode 4	Durchführung einer reinen Simulation auf Grundlage einer VCD-Datei, welche die Eingabemenge enthält
-vcdfile	Pfad zu einer VCD-Datei, welche eine Eingabemenge enthält
-vcdreffile	Pfad zu einer VCD-Datei, welche die Ergebnisse eines externen Simulationswerkzeugs enthält
-vcdts	Simulationszeitraum bei der Erzeugung von VCD-Dateien
-vcdmts	Maximaler zeitlicher Abstand zwischen zwei Stimuli bei der Erzeugung von VCD-Dateien
-vcdob	Ausschließliche Erzeugung von Stimuli mit den Signalwerten 0 und 1

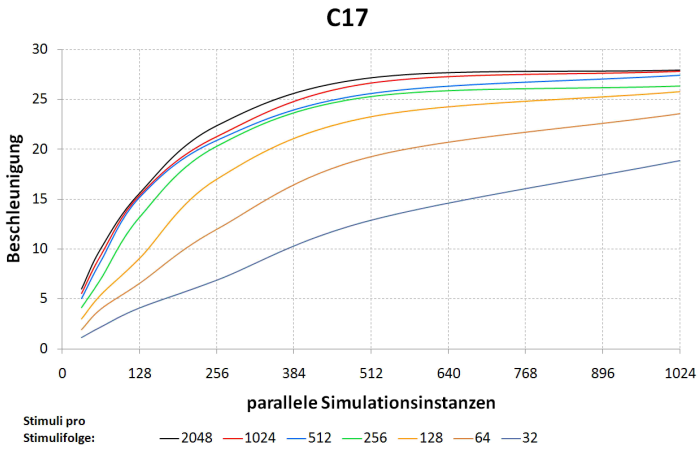
Tabelle 8.1: Befehlsumfang des ADAMA-Task CWTSim

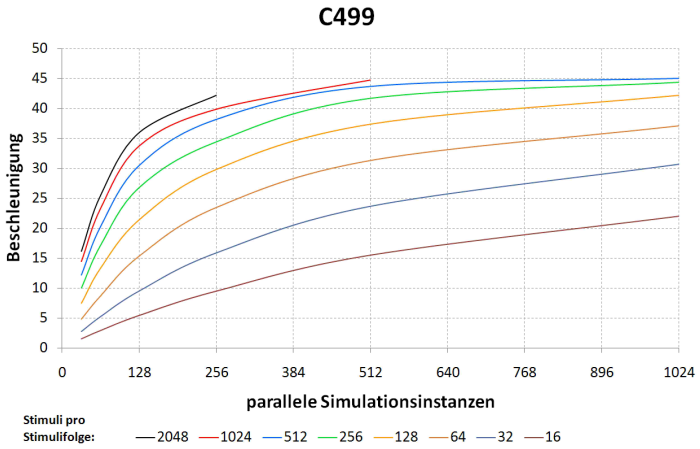
8.2 Spezifikationen der Zielhardware

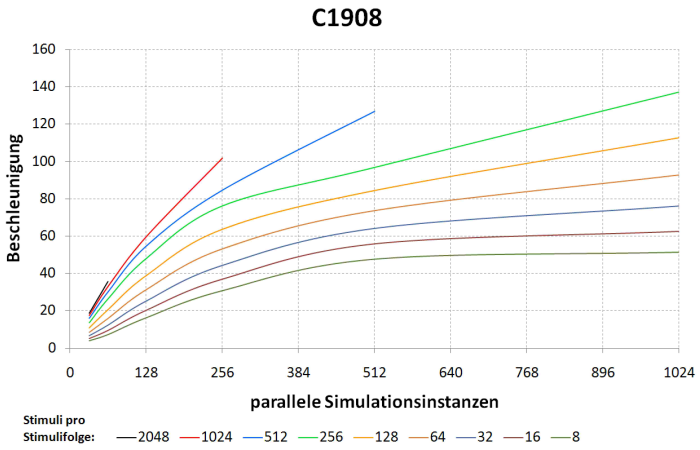
Art der Spezifikation	**Eigenschaft**
Grafikkarte	Nvidia Tesla C2050/C2070
Grafikprozessor	Fermi
Compute Capability	2.0
Anzahl Streaming-Multiprozessoren	14
Anzahl 32-Bit-Register pro Multiprozessor	1536
Maximale Anzahl von Threads	21504
L1-Cache pro Multiprozessor	16-48 KB
L2-Cache	768 KB
Maximale Anzahl parallel ausgeführter Kernel	16
Speicherschnittstelle	384 Bit
Speicherbandbreite	144 GB/s
Takt der Multiprozessoren	1,15 GHz
Theoretische Leistungsfähigkeit (Single Precision)	1,03 TFlops
Größe des Hauptspeichers	3 GB
Art des Hauptspeichers	GDDR5
Verlustleistung	238 Watt

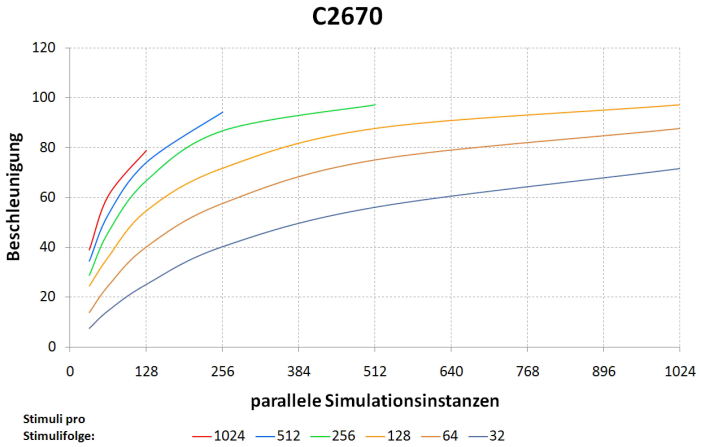
Tabelle 8.2: Spezifikationen zur Zielhardware [36] [40]

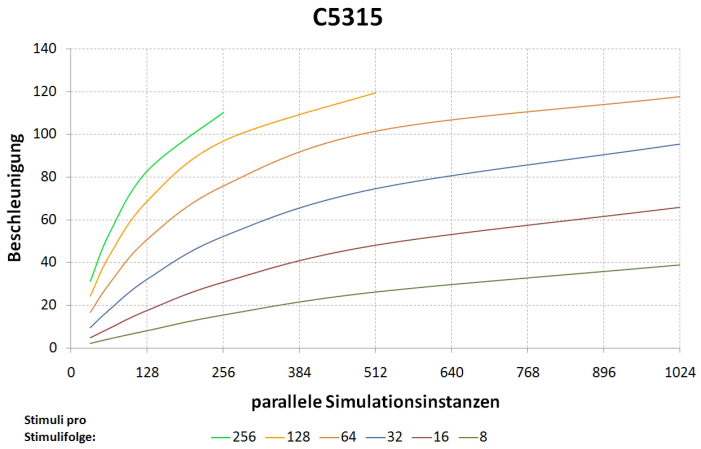
8.3 Anhang zur detaillierten Auswertung

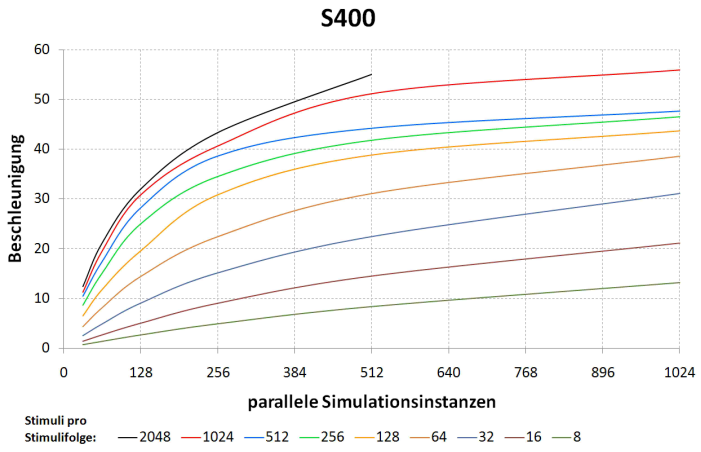


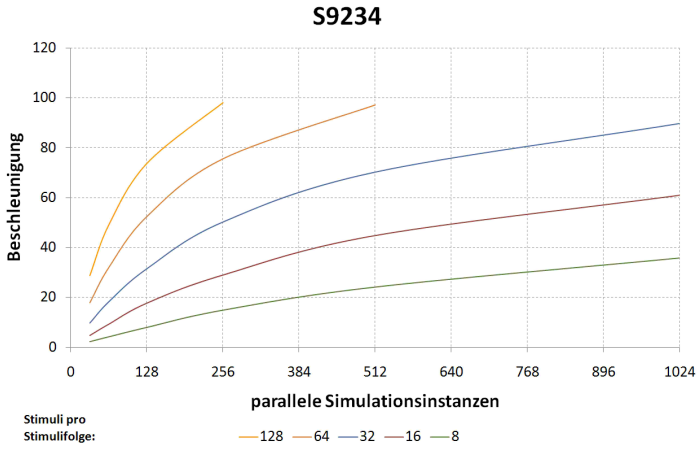


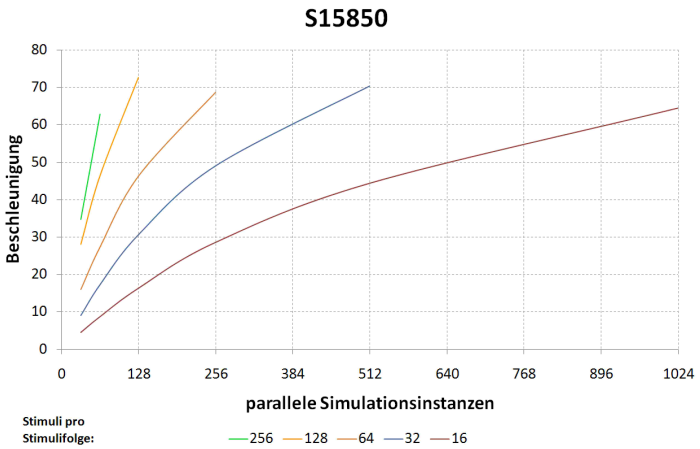



C2670
Beschleunigung
parallele Simulationsinstanzen
Stimuli pro Stimulifolge:
1024 512 256 128 64 32


C5315
Beschleunigung
parallele Simulationsinstanzen
Stimuli pro Stimulifolge:
256 128 64 32 16 8


S400
Beschleunigung
parallele Simulationsinstanzen
Stimuli pro Stimulifolge:
2048 1024 512 256 128 64 32 16 8


S9234
120
100
80
60
40
20
0
Beschleunigung
0
128
256
384
512
640
768
896
1024
parallele Simulationsinstanzen
Stimuli pro Stimulifolge:
128
64
32
16
8


S15850
80
70
60
50
40
30
20
10
0
Beschleunigung
0
128
256
384
512
640
768
896
1024
parallele Simulationsinstanzen
Stimuli pro Stimulifolge:
256
128
64
32
16

Literaturverzeichnis

[1] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009, ch. 8, pp. 451–462.

[2] F. Kesel and F. K. R. Bartholomae, *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs*. Oldenbourg Verlag, 2006, ch. 1.1, p. 5.

[3] G. Moore, "Cramming More Components Onto Integrated Circuits," *Proceedings of the IEEE*, vol. 86 (1), pp. 82–85, 1998.

[4] W. Ecker, W. Mueller, and R. Doemer, *Hardware-dependent Software: Principles and Practice*. Springer Science + Business Media B.V., 2009, ch. 1.1.2, p. 4.

[5] A. Iniguez, "Vector Language: A Proposed Verification Methodology for Intellectual-Property Cores," *14th Annual IEEE International ASIC/SOC Conference*, vol. 14, pp. 70–75, 2001.

[6] S. Hauck, "Multi-FPGA Systems," Ph.D. dissertation, University of Washington, 1995.

[7] D. Chatterjee, A. DeOrio, and V. Bertacco, "Event-Driven Gate-Level Simulation with GP-GPUs," *ACM/IEEE Design Automation Conference (DAC '09)*, vol. 46, pp. 557–562, 2009.

[8] S. Bose, H. Grimes, and V. Agrawal, "Delay Fault Simulation with Bounded Gate Delay Mode," *IEEE International Test Conference (ITC 2007)*, pp. 1–10, 2007.

[9] Y. Kawakami, J. Fang, H. Yonezawa, N. Iwanishi, L. Wu, A. I.-H. Chen, N. Koike, P. Chen, C.-S. Yeh, and Z. Liu, "Gate-Level Aged Timing Simulation Methodology for Hot-Carrier Reliability Assurance," *Proceedings of the Asia and South Pacific-Design Automation Conference*, pp. 289–294, 2000.

[10] D. Rabe, "Accurate Power Analysis of Integrated CMOS Circuits on Gate Level," Ph.D. dissertation, Carl von Ossietzky Universitaet Oldenburg, 2001.

[11] D. Lorenz, M. Barke, and U. Schlichtmann, "Efficiently Analyzing the Impact of Aging Effects on Large Integrated Circuits," *Reliability and variability of semiconductor devices and ICs (ICMAT)*, vol. 52 (8), pp. 1546–1552, 2011.

[12] N. Ahmed, M. Tehranipoor, and V. Jayaram, "Supply Voltage Noise Aware ATPG for Transition Delay Faults," *25th IEEE VLSI Test Symposium*, pp. 179–186, 2007.

[13] S. Sapatnekar, *Timing*. Kluwer Academic Publishers, 2004, p. 113.

[14] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879–899, 2008.

[15] S. Holst, E. Schneider, and H.-J. Wunderlich, "Scan Test Power Simulation on GPGPUs," in *IEEE Asian Test Symposium (ATS12), Niigata, Japan*, November 19-22, 2012.

[16] R. B. Reese and M. A. Thornton, *Introduction to Logic Synthesis Using Verilog Hdl*. Morgan & Claypool Publishers, 2006, ch. 1.6, p. 29.

[17] D. Chatterjee, A. DeOrio, and V. Bertacco, "GCS: HighPerformance Gate-Level Simulation with GP-GPUs," *Design, Automation and Test in Europe Conference and Exhibition (DATE '09)*, vol. 5, pp. 1332–1337, 2009.

[18] L. Suresh, N. Rameshan, M. S. Gaur, M. Zwolinski, and V. Laxmi, "Acceleration of Functional Validation using GPGPU," *Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)*, vol. 6, pp. 211–216, 2011.

[19] K. Gulati and S. P. Khatri, "Accelerating Statistical Static Timing Analysis Using Graphics Processing Units," *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 260–265, 2009.

[20] Y. Z. B. Wang and Y. Deng, "Distributed Time, Conservative Parallel Logic Simulation on GPUs," *ACM/IEEE Design Automation Conference (DAC '09)*, vol. 47, pp. 761–766, 2010.

[21] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. 5, pp. 440–452, 1979.

[22] A. Sen, B. Aksanli, and M. Bozkurt, "Speeding Up Cycle Based Logic Simulation Using Graphics Processing Units," *International Journal of Parallel Programming*, vol. 39, pp. 639–661, 2010.

[23] A. Miczo, *Digital Logic Testing and Simulation*. John Wiley & Sons, 2003, p. 60.

[24] S. Holst, *HowTo Get Things Done With ADAMA*, 2011.

[25] ModelSim Webseite. [Online]. Available: http://www.mentor.com/products/fpga/simulation/modelsim

[26] T. Rauber and G. Ruenger, *Parallele Programmierung*. Springer-Verlag Berlin Heidelberg, 2007, p. 722.

[27] *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2012. [Online]. Available: http://download.intel.com/products/processor/manual/325462.pdf

[28] CUDA Webseite. [Online]. Available: https://developer.nvidia.com/what-cuda

[29] AMD Accelerated Parallel Processing Webseite. [Online]. Available: http://developer.amd.com/tools/hc/AMDAPPSDK/Pages/default.aspx

[30] OpenCL - The open standard for parallel programming of heterogeneous systems. [Online]. Available: www.khronos.org/opencl/

[31] *GPGPU Computing Horizons: Developing and Deploying for Microsoft Windows*, Microsoft Corporation, 2010. [Online]. Available: http://download.microsoft.com/download/

A/6/C/A6C0223B-9460-4346-8DC0-B6BCFD9269B4/MSFT_GPGPU_whitepaper_FINAL.PDF

[32] A. Nischwitz, M. Fischer, P. Haberaecker, and G. Socher, *Computergrafik und Bildbearbeitung: Computergrafik und Bildverarbeitung: Band I: Computergrafik: 1*. Vieweg+Teubner Verlag, 2011, pp. 481–486.

[33] H. El-Rewini and M. Abd-El-Barr, *Advanced Computer Architecture and Parallel Processing*. John Wiley & Sons, 2005, ch. 1.2, p. 4.

[34] *NVIDIA CUDA C Programming Guide 4.2*, NVIDIA Corporation, 701 San Tomas Expressway, Santa Clara, CA 95050. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[35] *NVIDIA CUDA Architecture*, 1st ed., NVIDIA Corporation, 701 San Tomas Expressway, Santa Clara, CA 95050, April 2009. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf

[36] *Anhang F des NVIDIA CUDA C Programming Guide 4.2*, NVIDIA Corporation.

[37] *Kapitel 5 des NVIDIA CUDA C Programming Guide 4.2*, NVIDIA Corporation.

[38] T. Ottmann and P. Widmayer, *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag Heidelberg, 2012, p. 478.

[39] *IEEE Standard Verilog Hardware Description Language* - 1364-2001, IEEE Computer Society Std.

[40] *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi*, 1st ed., NVIDIA Corporation, 701 San Tomas Expressway, Santa Clara, CA 95050, 2009.

Sämtliche Einträge zu Webseiten wurden zuletzt am 29. November 2012 abgerufen.

Abbildungsverzeichnis

Algorithmenverzeichnis

Tabellenverzeichnis

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Stuttgart, den 1. Dezember 2012