Universität Stuttgart
Fakultät Informatik

Prüfer:	Prof. Dr.-Ing. U. G. Baitinger
Betreuer:	Dipl. Ing. Markus Bühler
Beginn:	1. November 1997
Ende:	20. März 1998
CR-Klassifikation	B2.2, B7.2

Studienarbeit Nr. 1677

Entwicklung eines Zeitmodells für einen symbolischen Simulator

Kai Kapp

Institut für Parallele und Verteilte Höchstleistungsrechner

Breitwiesenstraße 20-22

D-70565 Stuttgart

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungen

CMOS	Complementary Metal Oxide Semiconductor
ZDM	Zero-Delay-Modell
TDM	Transport-Delay-Modell
RDM	Real-Delay-Modell
MHD	Minimum Hamming Distance
VLSI	Very Large Scale Integration
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language

Einleitung

Kapitel 1

Lange Zeit beschränkten sich die Optimierungsverfahren beim Entwurf höchstintegrierter Schaltungen auf Schnelligkeit, geringen Flächenbedarf und minimale Kosten von Schaltkreisen. In Folge dessen wurden immer schnellere und kleinere Schaltungen mit entsprechend wachsender Integrationsdichte entwickelt, die allerdings eine erhöhte Wärmeentwicklung aufwiesen. Hohe Temperaturen beeinträchtigen jedoch die Zuverlässigkeit von Schaltkreisen, so daß der Schaltkreis entweder durch aufwendige Kühlvorrichtungen in einem sicheren Temperaturbereich gehalten oder die Ursache bekämpft werden muß. So versucht man, die hohe Wärmeentwicklung durch Senkung der Leistungsaufnahme zu minimieren. Zusätzlich hat die steigende Verbreitung von tragbaren Geräten wie Laptops und Mobiltelefonen dazu beigetragen, daß die Minimierung des Leistungsverbrauchs mittlerweile einen ähnlich hohen Stellenwert einnimmt wie die Minimierung des Flächenaufwands und die Schnelligkeit von Schaltkreisen.

1.1 Leistungsaufnahme in CMOS-Schaltkreisen

In diesem Kapitel wird dargelegt, welche Ursachen für die Leistungsaufnahme in Schaltkreisen mit CMOS-Technologie verantwortlich sind. Es wird gezeigt, daß sich das Problem der Abschätzung des durchschnittlichen Leistungsverbrauchs im allgemeinen auf die Berechnung der durchschnittlichen Schaltaktivität (der Anzahl der zu erwartenden Umschaltvorgänge pro Taktzyklus) der Gatter eines Schaltkreises reduzieren läßt. Das Kapitel schließt mit der Klassifizierung aller Arten von Schaltvorgängen, die bei einem Gatter auftreten können, ab.

1.1.1 Ursachen

Die Leistungsaufnahme in CMOS-Schaltkreisen verteilt sich auf folgende Ursachen:

- **Leckströme** treten in Dioden und Transistoren durch nicht ideales Verhalten auf und sind primär von der Herstellung abhängig.
- **Kurzschlußströme** fließen beim Schaltvorgang, während die komplementären Transistoren umschalten.
- **Umladen von Kapazitäten** an Gatterausgängen aufgrund von Transitionen, Hazards und Glitches (siehe Kapitel 1.1.2).

Während Leckströme dem sogenannten *statischen* Leistungsverbrauch zugeordnet werden, zählen Kurzschlußströme und das Umladen von Kapazitäten zum *dynamischen* Leistungsverbrauch, weil sie direkt mit den Signalwechseln in der Schaltung in Zusammenhang stehen. Da die Leistungsaufnahme durch Leckströme ausschließlich von der Herstellung und der gewählten Technologie abhängt, wird sie von weiteren Betrachtungen ausgeschlossen. Außerdem ist der Leistungsaufwand für Leck- und Kurzschlußströme so gering, daß er gegenüber dem Leistungsverlust durch das Umladen von Kapazitäten vernachlässigt werden kann.

Die durchschnittliche Leistungsaufnahme eines getakteten Schaltkreises beläuft sich nach [Mont 97] auf:

$$P = \sum_{\text{alle Gatter i}} \frac{1}{2 \cdot T_C} \cdot V_{DD}^2 \cdot C_i \cdot sw_i$$

V_{dd} repräsentiert die Versorgungsspannung, T_C den Taktzyklus der Schaltung, C_i die Lastkapazität und sw_i die Schaltaktivität des Gatters i. Alle Parameter der Gleichung mit Ausnahme der Schaltaktivität sind Konstanten, die von der gewählten Technologie, der Topologie und dem geometrischen Aufbau der Schaltung abhängen. Übrig bleibt die Berechnung der durchschnittlichen Schaltaktivität eines jeden Gatters.

1.1.2 Klassifizierung der Schaltvorgänge

In der Literatur findet man viele verschiedene Verwendungen der Begriffe Transitionen, Hazards und Glitches, so daß hier eindeutige Definitionen für deren weiterer Gebrauch in dieser Studienarbeit vorgestellt werden soll:

Definition 1-1: Transitionen:

> Bei Transitionen handelt es sich um Signaländerungen am Ausgang eines Gatters. Es bezeichnet einen logischen Übergang von 0 nach 1 bzw. von 1 nach 0. Dabei wird die Ausgangskapazität des Gatters jeweils vollständig geladen bzw. entladen. In dieser Studienarbeit wird ausschließlich positive Logik verwendet, so daß die logische 0 stets einem niedrigen und die logische 1 einem hohen Spannungspegel entspricht.

Im Beispiel in Abbildung 1-1 werden die Gatterlaufzeiten des Inverters und des UND-Gatters vollständig vernachlässigt. Die Gatterlaufzeit repräsentiert die Zeitspanne, die zwischen einer Signaländerung an den Eingängen eines Gatters und der entsprechenden Reaktion am Ausgang des Gatters vergeht. Im sogenannten Zero-Delay-Modell werden eben diese Gatterlaufzeiten vernachlässigt und als unendlich schnell angenommen (siehe auch Kapitel 4.2.1).

So finden in diesem Beispiel an den Ausgängen der Gatter insgesamt zwei Transitionen statt: Sowohl der Inverter als auch das UND-Gatter wechseln zum Zeitpunkt t=4 von 0 nach 1.

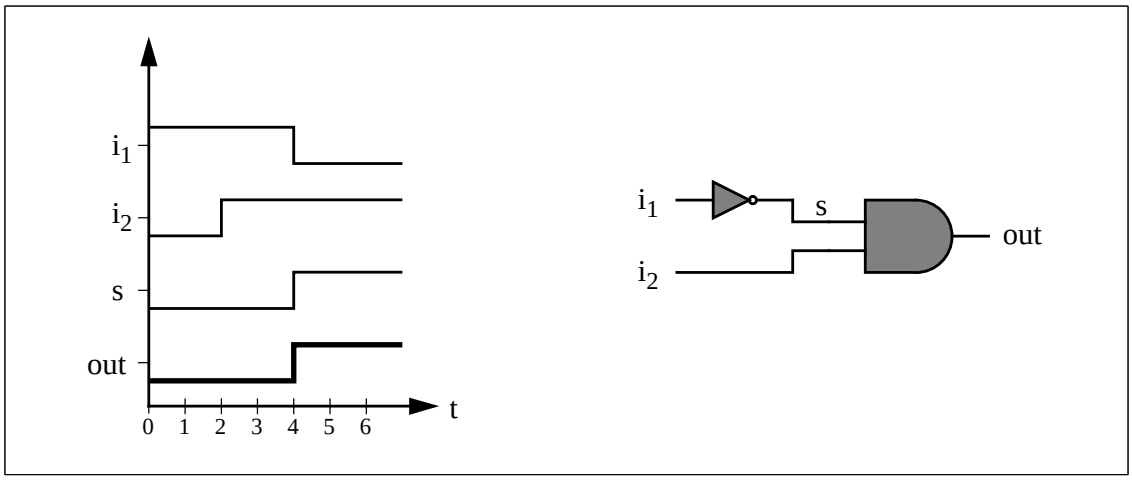


Abbildung 1-1: Beispiel für eine Transition

Definition 1-2: Hazards:

Hazards sind ein Spezialfall von Transitionen. Sie beschreiben ungewollte Signaländerungen, die aus Gatterlaufzeiten resultieren. Da das Zero-Delay-Modell Gatterlaufzeiten vernachlässigt, benutzen wir für dieses Beispiel das sogenannte Real-Delay-Modell (siehe auch Kapitel 4.4.1). In diesem Modell werden den Signalwechseln an den Ausgängen von Gattern bestimmte Verzögerungszeiten zugeordnet, was zu deutlich realistischeren Simulationsergebnissen führt.

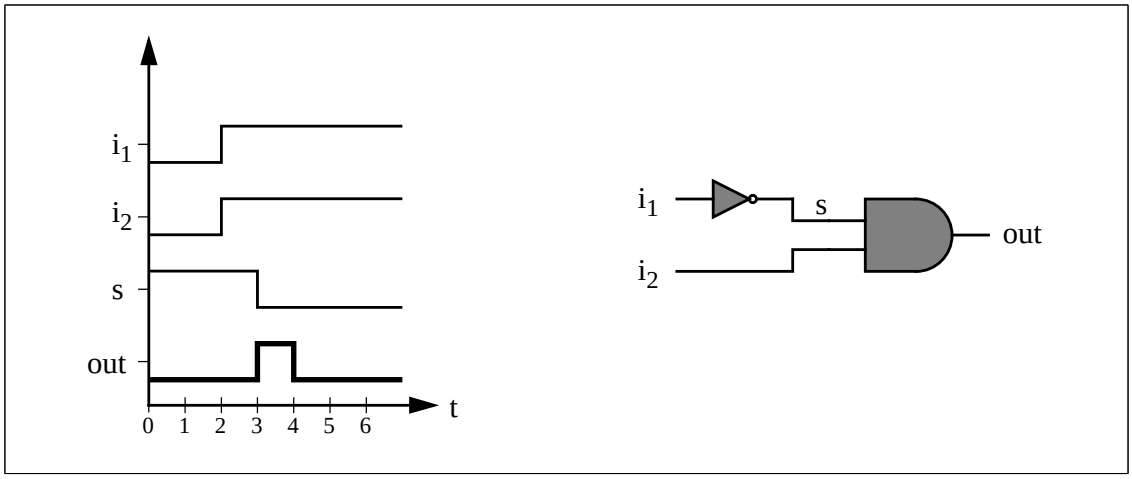


Abbildung 1-2: Beispiel für Hazards

Im Beispiel in Abbildung 1-2 haben beide Gatter eine Laufzeit von einer Zeiteinheit. Zum Zeitpunkt t=2 schalten beide Eingänge auf 1 um. Der Gatterlaufzeit zufolge braucht der Inverter eine Zeiteinheit, um den Wechsel des seines Eingangssignals durchzupropagieren. Währenddessen liegt an den Eingängen des UND-Gatters jeweils eine 1 an. Dadurch schaltet das UND-Gatter mit einer Zeiteinheit Verzögerung bei t=3 auf 1. Gleichzeitig schaltet der Inverter auf 0. Diese Signaländerung hat das UND-Gatter dann bei t=4 durchpropagiert

und schaltet wieder auf 0 zurück. Da nur das endgültige Ergebnis am Gatterausgang zur Funktion des Schaltkreises beiträgt, sind diese zwei Transitionen am Ausgang des UND-Gatters überflüssig. Derartige unerwünschte Transitionen werden als Hazards bezeichnet und allein durch Gatterlaufzeiten verursacht. Im Gegensatz dazu findet am Ausgang des Inverters in diesem Beispiel nur eine einzige (gewollte) Transition und kein Hazard statt.

Definition 1-3: Glitches:

Bei Glitches finden im Gegensatz zu Transitionen und Hazards keine vollständigen Signalübergänge statt. Eine Beschränkung auf die logisch-digitale Ebene ist hier nicht möglich und die Betrachtung analoger Spannungsübergänge auf der Schaltkreisebene wird erforderlich. Ein Glitch findet statt, wenn die Signaländerung am Ausgang von so kurzer Dauer ist, daß kein vollständiges Umladen der Kapazität, also keine Transition, erfolgen kann. So resultiert ein Glitch nur in einem kurzen Spannungsabfall bzw. -hub im Ausgangssignalverlauf des Gatters (siehe Abbildung 1-3). Durch die Sperrschichtkapazitäten des Transistors und die sich aus den Folgegattern ergebende Lastkapazität sind die Flanken in diesem Fall nicht unendlich steil wie bei der Betrachtung auf der logisch-digitalen Ebene (Bsp.: Abb. 1-2).

In Abbildung 1-3 ist der typische Spannungsverlauf am Gatterausgang während eines Glitches dargestellt: Die Spannung sinkt nicht vollständig auf den Low-Pegel ab. Der gestrichelte Verlauf gibt den idealen Verlauf ohne Berücksichtigung parasitärer Kapazitäten wieder und δ die Breite dieses Impulses. In Abhängigkeit von der Impulsbreite δ wird während der Simulation nach [WIL 98] die Höhe des Energieverbrauchs berechnet und entschieden, ob ein Glitch oder ob Transitionen stattgefunden haben.

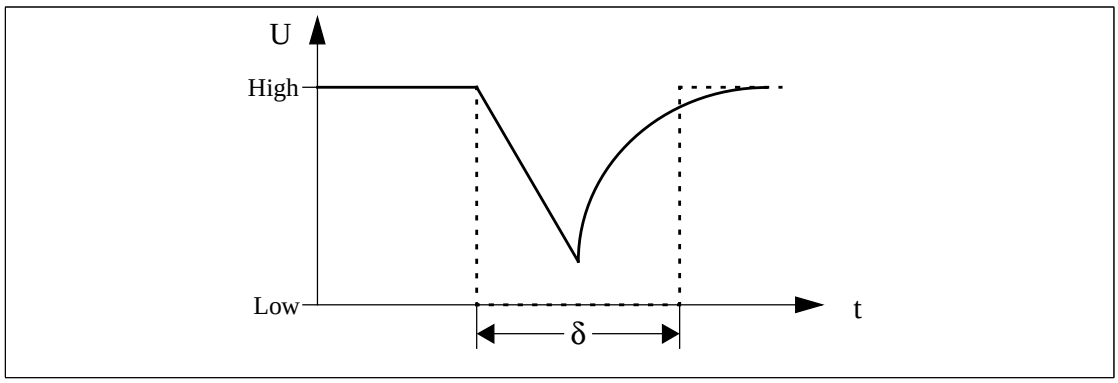


Abbildung 1-3: Auswirkungen eines Glitches am Gatterausgang

1.2 Simulationen

Um eine Schaltung zu entwickeln, die einen möglichst geringen Energiebedarf hat, ist eine genaue Abschätzung des Energieaufwandes unentbehrlich. Am exaktesten sind Schaltungssimulatoren auf der physikalischen Ebene (wie z.B. Spice), welche die Spannungs- und

Stromverläufe analog untersuchen. Allerdings haben diese Simulatoren bei größeren Schaltungen einen enormen Speicherbedarf und eine inakzeptabel hohe Laufzeit. Zusätzlich kann eine Simulation erst sehr spät während des Entwicklungsablaufs vorgenommen werden.

Im Vergleich dazu sind Simulatoren, die Schaltkreise auf der logischen Gatterebene untersuchen, um Größenordnungen schneller und das bei viel geringerem Speicherbedarf. Außerdem ermöglichen sie die Abschätzung des Energieaufwandes in einem früheren Stadium der Entwicklung, unabhängig von der Zieltechnologie. Die Schnelligkeit der Simulation erlaubt unter anderem die Abschätzung mehrerer Schaltungsalternativen mit dem Ziel diejenige mit dem geringsten Energiebedarf auszuwählen. Im Vergleich zu der Schaltungssimulation sind allerdings die Schätzergebnisse ungenauer.

Während bei den Simulatoren, die auf der Schaltkreisebene arbeiten, jeder Transistor für sich betrachtet wird, sind auf der Gatterebene mehrere Transistoren zu jeweils einem logischen Gatter zusammengefaßt. Eine zusätzliche Beschleunigung der Logiksimulatoren wird durch die Diskretisierung der Spannungswerte zu einer logischen 0 oder 1 erreicht. Ziel der Logiksimulatoren ist die Berechnung der zeitlichen Verläufe aller Signale in einer Schaltung. Aus den Signalverläufen läßt sich die Anzahl an aufgetretenen Transitionen ermitteln und daraus mit der Formel aus Kapitel 1.1.1 der durchschnittliche Energiebedarf des Schaltkreises.

Zwei Formen der logischen Schaltkreissimulation sollen im Folgenden näher betrachtet werden: Die Logiksimulation und die probabilistische Simulation.

1.2.1 Die Logiksimulation

Die Logiksimulation gehört mit zu den ersten Simulationsverfahren. Aus diesem Grund existiert eine weite Verbreitung von Logiksimulatoren in kommerziellen Entwicklungsumgebungen. Bei der Logiksimulation wird an den Primäreingängen der zu untersuchenden Schaltung eine Sequenz von Testvektoren angelegt. Jeder Testvektor enthält die Signalwerte aller Primäreingänge, welche diese zu einem bestimmten Taktzyklus als Simulationsstimuli annehmen. Zu jedem neuen Taktzyklus wird der nächste Testvektor angelegt und untersucht, wie oft die Gatter ihre Ausgabewerte ändern. Auf diese Weise wird die gesamte Testvektorsequenz durchpropagiert und anschließend die Signalaktivität jedes Gatters berechnet.

Die Logiksimulation liefert für die jeweils verwendete Testvektorsequenz exakte Ergebnisse bezüglich der Anzahl an Transitionen, welche an den Gatterausgängen stattfinden. Allerdings ist eine erschöpfende Simulation aller möglichen Testvektoren bei größeren Schaltungen unmöglich. Gegenstand der Forschung ist die Suche nach Testvektoren, die ein möglichst realitätsnahes Ergebnis liefern.

1.2.2 Die Probabilistische Simulation

Dieses Simulationsverfahren verwendet statt konkreten Signalwerten die Signalwahrscheinlichkeiten der Primäreingänge als Stimuli. Die Signalwahrscheinlichkeit beschreibt die Wahrscheinlichkeit, daß ein Signal einen bestimmten Wert annimmt. Durch die Schaltung werden in diesem Fall keine Signalwerte propagiert sondern die Wahrscheinlichkeiten für deren Auftreten. Dadurch ist nur ein einziger Simulationsdurchlauf erforderlich und nicht wie bei der Logiksimulation ein Durchlauf für jeden Testvektor. Sind zu den Primäreingängen zusätzlich die Schaltwahrscheinlichkeiten gegeben, so kann daraus auf ähnliche Art die Schaltwahrscheinlichkeit der Ausgänge berechnet werden.

Geht man bei den Eingangswahrscheinlichkeiten davon aus, daß sie statistisch unabhängig sind, so ist die Berechnung der Ausgangswahrscheinlichkeit eines Gatters sehr einfach. In Abbildung 1-4 werden beispielsweise zur Berechnung der Ausgangswahrscheinlichkeit eines UND-Gatters die Signalwahrscheinlichkeiten seiner Eingänge miteinander multipliziert.

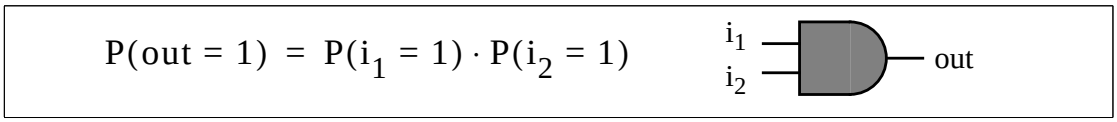


Abbildung 1-4: Signalwahrscheinlichkeiten bei statistischer Unabhängigkeit

Sobald allerdings die Eingangssignale aufgrund von Korrelationen (siehe Kapitel 1.3) nicht mehr statistisch unabhängig sind, gestaltet sich die Berechnung deutlich aufwendiger. So können für ein internes Gatter nicht mehr die Ausgangswahrscheinlichkeiten der Gatter an den Eingängen verwendet werden, sondern die Berechnung muß bis zu den Primäreingängen durch die gesamten Teilnetze, die an den Eingängen des zu untersuchenden Gatters liegen, zurückgeführt werden.

1.2.3 Die Symbolische Simulation

Der bestehende Simulator von [DD 98] ist ein symbolischer Simulator. Bei ihm werden wie in der Logiksimulation Signalwerte durch die Schaltung propagiert. Die Testvektoren werden dabei allerdings umgeordnet, um die Geschwindigkeit der Simulation zu erhöhen (siehe Kapitel 2.1.2). Ziel der symbolischen Simulation ist die Ermittlung der Anzahl an auftretenden Transitionen und Hazards bzw. der Signalwahrscheinlichkeiten und nicht der exakte Ausgabesignalverlauf des Schaltkreises, wie es bei der Logiksimulation der Fall ist.

1.3 Korrelationen

Korrelationen beschreiben Wechselbeziehungen zwischen Signalen eines Schaltkreises. Dabei unterscheidet man zeitliche und räumliche Korrelationen.

Definition 1-4: Zeitliche Korrelation

Hängt der Signalwert eines Eingangsknotens zum Taktzyklus i von seinem Wert zum vorherigen Taktzyklus i-1 ab, so gilt das Signal als zeitlich korreliert.

Definition 1-5: Räumliche Korrelation

Hängt der Signalwert eines Knotens von dem Wert eines anderen Knotens in dem Schaltkreis ab, so stehen die Signale in einer räumlichen Korrelation.

Durch Korrelationen wird die einfache Berechnung der Signalwahrscheinlichkeiten, wie sie in Abbildung 1-4 erfolgt, vereitelt. Die Signale sind nicht mehr statistisch unabhängig und die Ausgangswahrscheinlichkeiten sind sogenannte bedingte Wahrscheinlichkeiten.

1.4 Die Studienarbeit

1.4.1 Aufgabenstellung

Die Grundaufgabe der hier vorliegenden Arbeit war die Erweiterung eines in [DD 98] entwickelten Logiksimulators um ein Real-Delay-Modell. Der bestehende Simulator nutzt Analogien zwischen Mengenlehre und Wahrscheinlichkeitstheorie, um eine schnelle Simulation zeitlich und räumlich korrelierter Eingangssignale zu realisieren. Allerdings ist er bisher auf ein Zero-Delay-Modell beschränkt gewesen. Hazards und Glitches, die gerade durch Gatterlaufzeiten verursacht werden, konnten bei der Berechnung der Schaltaktivitäten der Gatter nicht berücksichtigt werden. Bei manchen Schaltungstypen wie Addierern oder Multiplizierern können allerdings Hazards einen Transitionsanteil von über 90% ausmachen!

In Kapitel 4.4 wird das Real-Delay-Modell vorgestellt, welches die Erfassung von Hazards unterstützt und in Kapitel 4.5 eine Fortentwicklung, die Glitch-Simulation, welche zusätzlich den durch Glitches nach [WIL 98] verursachten Energieaufwand berücksichtigt.

Eine Anforderung an die Durchführung der Arbeit war die Erstellung eines klaren, objektorientierten Konzepts (siehe Kapitel 3) und dessen Implementierung in C++. Zur Demonstration der Flexibilität und Erweiterbarkeit des objektorientierten Aufbaus wurden außerdem zwei zusätzliche Simulationsmodi entwickelt: Die Zero-Delay-Simulation (siehe Kapitel 4.2) wurde reimplementiert und eine weitere Simulationsart, die Transport-Delay-Simulation (siehe Kapitel 4.3), entwickelt.

Abschließend war eine Gegenüberstellung mit dem kommerziellen Logik-Simulator „Synopsys“ und dem ursprünglichen Simulator nach [DD 98] vorzunehmen.

1.4.2 Die verwendeten Mittel

Die Erweiterung wurde in reinem ANSI-C++ entwickelt. Dies ermöglicht eine vollständig plattformunabhängige Verwendung des entwickelten Codes. Als Programmierumgebung wurde VisualAge C++ von IBM unter OS/2 verwendet. Die endgültige Integration der

Erweiterung in den bestehenden Simulator von [DD 98] erfolgte mit dem GNU-C++-Compiler unter Solaris und Linux. Aufgrund der strikten Einhaltung des ANSI-C++-Standards war dazu keinerlei Modifikation des Programmcodes erforderlich.

1.5 Übersicht

Die weitere Arbeit ist folgendermaßen aufgebaut:

- In Kapitel 2 wird die in der Erweiterung verwendete Signalrepräsentation vorgestellt. Zunächst wird die logische Interpretation der Signaldarstellung erläutert und anschließend ihre Implementierung in C++.
- Kapitel 3 beschreibt ausführlich den objektorientierten Aufbau der Netzliste. Es wurde versucht real existierende Objekte möglichst naturgetreu mithilfe des Paradigmas der Objektorientierung in den Rechner abzubilden. In diesem Zusammenhang wird des weiteren die Verwendung virtueller Funktionen erläutert, welche immens zur Verständlichkeit, Übersichtlichkeit und Erweiterbarkeit des Programmcodes beitragen.
- In Kapitel 4 werden die vier Simulationsmodelle beschrieben, welche durch die Erweiterung realisiert werden: Die Zero-Delay-Simulation, die Transport-Delay-Simulation, die Real-Delay-Simulation und die Glitch-Simulation. Zu jeder Simulationsmethode wird zunächst das zugrundeliegende logische Modell erläutert. Die Vorgehensweise eines jeden Modells wird an einem ausführlichen Beispiel veranschaulicht und abschließend im Pseudocode zusammenfassend dargestellt.
- Ein Vergleich mit dem bestehenden Simulator von [DD 98] und dem kommerziellen Logiksimulator Synopsys folgt in Kapitel 5. In diesem Kapitel werden die Genauigkeit und die Geschwindigkeit der in dieser Arbeit vorgestellten Erweiterungen untersucht.
- Eine Auflistung der entwickelten Klassen und ihrer Methoden liegt in Anhang A vor. Die Klassendefinitionen sind nach den Dateien, in welchen sie definiert sind, geordnet.
- Anhang B enthält die erweiterten Optionen für den Aufruf des bestehenden Simulators von [DD 98].
- In Anhang C sind abschließend die Aufbereitung der Simulationsdaten und deren Simulation grafisch darstellt.

Die Signalrepräsentation

Kapitel 2

2.1 Die Semantik der Signaldarstellung

In der Simulation auf Gatterebene kann ein Signal die Werte 0 oder 1 annehmen. Mit jedem Signalwert wird ein bestimmter Zeitpunkt verknüpft. Die Zeitpunkte haben in diesem Fall nichts mit Realzeit zu tun, sondern dienen allein zum Einhalten von Kausalitäten. Über sie wird entschieden, in welchem Verhältnis ein Ereignis zu einem anderen steht: Es kann davor, gleichzeitig oder danach stattfinden. Ziel dieser Simulation ist nicht die Erstellung eines wirklichkeitsgetreuen Ausgabesignalverlaufs, sondern allein die Berechnung der Anzahl von Transitionen und Hazards, die während der gesamten Simulation auftreten.

2.1.1 Die Signalrepräsentation der Primäreingänge

Tabelle 2-1 zeigt das Beispiel einer Folge von Eingangsvektoren in einer Schaltung mit zwei Primäreingängen: PI_1 und PI_2. Jede Spalte repräsentiert eine Eingangsbelegung für die Simulation und wird als (Test-)Vektor bezeichnet. Zu jedem Taktzyklus liegt ein neuer Testvektor an den Primäreingängen an. Zum Beispiel liegt bei einer Testvektorsequenz wie in Tabelle 2-1 zum Taktzyklus 2 am Primäreingang 1 (PI_1) eine 0 und am Primäreingang 2 (PI_2) eine 1 an.

Vektornummer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
PI_1	0	0	1	0	1	1	0	1	0	1
PI_2	0	1	0	1	1	0	1	0	1	0

Tabelle 2-1: Beispiel einer Testvektorsequenz

Alle zu einem bestimmten Taktzyklus gehörigen Eingangssignale sind innerhalb einer Spalte angeordnet. Solange diese Werte nicht getrennt werden, ist die Einhaltung von räumlichen Korrelationen bereits sichergestellt. Um zusätzlich die Möglichkeit zu erhalten, die Testvektoren umzuordnen ohne zeitliche Korrelationen zu verletzen, wird die Darstellung aus Tabelle 2-1 um jeweils eine Zeile pro Eingangssignalverlauf zu Tabelle 2-2 ergänzt. Die

neuen Zeilen enthalten die Folgezustände des jeweiligen Primäreingangs zum nächsten Taktzyklus und werden aus den ursprünglichen, um eine Stelle nach links rotierten Werten gebildet.

Vektornummer	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
PI_1	0	0	1	0	1	1	0	1	0	1
PI_{zk1}	0	1	0	1	1	0	1	0	1	0
PI_2	0	1	0	1	1	0	1	0	1	0
PI_{zk2}	1	0	1	1	0	1	0	1	0	0

Tabelle 2-2: Ergänzung der Testvektoren um ihre Folgewerte

Nach diesem Schritt enthalten die Spalten alle Informationen über etwaige zeitliche und räumliche Korrelationen zwischen den Signalen und können somit beliebig umgeordnet werden.

2.1.2 Optimierung der Signalrepräsentation

Zur Optimierung der bestehenden Signalrepräsentation werden als erstes alle Testvektoren, welche öfter als einmal auftreten, zu einem einzigen Vektor zusammengefaßt. Jeder Vektor wird anschließend mit einer Gewichtung versehen, die der Anzahl seines Vorkommens entspricht. Im Beispiel werden die Vektoren 2, 7, 9 und 3, 6, 8 zu den Testvektoren 2 und 3 in Tabelle 2-3 zusammengefaßt und mit der Gewichtung 3 versehen. Da alle anderen Vektoren nur einmal vorkommen, erhalten sie die Gewichtung 1.

Vektornummer	**1**	**2**	**3**	**4**	**5**	**6**
PI_1	0	0	1	0	1	1
PI_{zk1}	0	1	0	1	1	0
PI_2	0	1	0	1	1	0
PI_{zk2}	1	0	1	1	0	0
Gewichtung	1	3	3	1	1	1

Tabelle 2-3: Zusammenfassung und Gewichtung der Testvektoren

Weitere Optimierungen sind denkbar. Zum Beispiel zielt die in [MB 98] vorgestellte Signalrepräsentation auf möglichst große Blöcke von nebeneinander liegenden Einsen ab. Die dort präsentierte Umsortierungsheuristik namens MHD (Minimum Hamming Distance) ergibt auf die Testvektorsequenz in Tabelle 2-3 angewandt folgende Darstellung:

Vektornummer	**1**	**2**	**3**	**4**	**5**	**6**
PI_1	0	1	1	1	0	0
PI_{zk1}	0	0	0	1	1	1
PI_2	0	0	0	1	1	1
PI_{zk2}	1	1	0	0	0	1
Gewichtung	1	3	1	1	3	1

Tabelle 2-4: Die mittels MHD optimierte Testvektorsequenz

2.1.3 Zeit in der Simulation

Bei der Simulation wird davon ausgegangen, daß zu Beginn eines neuen Taktzyklus alle Signaländerungen, die zu Beginn des vorhergehenden Taktzyklus angeregt wurden, vollständig bis zu den Primärausgängen der Schaltung durchpropagiert worden sind und sich die Schaltung in einem statischen Zustand befindet. Diese Voraussetzung ermöglicht die Betrachtung eines jeden Taktzyklus bzw. eines jeden Testvektors in seinem eigenen Zeitrahmen. Innerhalb eines Zeitrahmens wird t=0 als derjenige Zeitpunkt definiert, zu dem der aktuell zu untersuchende Taktzyklus beginnt und die Primäreingänge die Werte des zugehörigen Testvektors annehmen. Der Inhalt von Tabelle 2-4 wird in diesem Sinne folgendermaßen interpretiert:

Signal	**Zeitpunkt**	**1**	**2**	**3**	**4**	**5**	**6**
PI_1	t<0	0	1	1	1	0	0
	t=0	0	0	0	1	1	1
PI_2	t<0	0	0	0	1	1	1
	t=0	1	1	0	0	0	1

Tabelle 2-5: Zeitliche Interpretation der Testvektorsequenz aus Tabelle 2-4

Der obere Wert in einer Spalte eines Signals repräsentiert dessen Endwert während des vorhergehenden Taktzyklus. Dieser Zeitpunkt wird mit „t<0“ gekennzeichnet und stellt den Wert des Signals unmittelbar vor t=0 dar. Die Werte zu t<0 werden als *Anfangswerte* bezeichnet, da mit ihrer Hilfe die Ausgangswerte aller Gatter der Schaltung für den aktuell

zu simulierenden Taktzyklus berechnet werden. Zu t=0 beginnt definitionsgemäß der aktuell zu betrachtende Taktzyklus und die Primäreingänge erhalten jeweils den unteren Wert in der Spalte.

Testvektor 2 aus Tabelle 2-5 wird beispielsweise folgendermaßen interpretiert: Die Schaltung ist mit PI_1=1 und PI_2=0 seit unendlich langer Zeit eingeschwungen. Sie befindet sich in einem statischen Zustand und alle Gatterausgänge haben in Abhängigkeit der Primäreingänge wohldefinierte Werte angenommen. Nun ändern sich zum Zeitpunkt t=0 beide Primäreingänge und es gilt: PI_1=0 und PI_2=1. Diese Signaländerungen werden anschließend im Zuge der Simulation durch die gesamte Schaltung propagiert.

Eine Zeile in Tabelle 2-5 wird als *Zeitwert* bezeichnet. Jeder Zeitwert eines Signals besteht aus einem Zeitpunkt und den zugehörigen Signalwerten über alle Testvektoren. Jedes Gatter enthält nach der Simulation mindestens einen solchen Zeitwert: t<0.

2.1.4 Die Signalrepräsentation der Gatterausgänge

Während die Signaldarstellung von Primäreingängen definitionsgemäß nur zwei Zeitpunkte erlaubt, nämlich t<0 und t=0, können die Ausgänge von internen Gattern der Schaltung mehrere Zeitpunkte beinhalten. Wie bei Primäreingängen ist auch bei internen Signalen der erste Zeitpunkt definitionsgemäß „t<0“. Weitere Zeitpunkte ergeben sich durch Laufzeitverzögerungen und durch unterschiedliche Ankunftszeiten der Signale an den Eingängen eines Gatters. Somit könnten sich im Laufe einer Simulation die in Tabelle 2-6 vorgestellten Signale interner Gatter ergeben.

Signal	Zeitpunkt	1	2	3	4	5	6
s_{A1}	t<0	0	0	1	1	1	0
	t=1	1	1	0	1	1	1
	t=4	0	1	1	1	0	0
s_{A2}	t<0	1	1	0	1	1	0
	t=3	1	0	0	0	0	1
	t=8	1	0	1	1	0	1

Tabelle 2-6: Beispiel für Signalverläufe interner Gatter

Die verschiedenen Zeitpunkte stellen sich im Laufe der Simulation heraus (siehe Kapitel 4). Wie man sieht, ist diese Signaldarstellung keineswegs redundanzfrei. Der Testvektor 4 des Signals s_{A1} wäre schon allein durch den obersten Eintrag zum Zeitpunkt t<0 hinreichend spezifiziert. Allerdings ergibt sich durch diese Darstellung die Möglichkeit schnell und unkompliziert logische Verknüpfungen zwischen verschiedenen Signalen zeilenweise (also parallel, siehe Kapitel 2.1.5) durchzuführen. Außerdem sind redundanzärmere Signalreprä-

sentationen im allgemeinen deutlich speicheraufwendiger, da schon allein ein Zeiger auf eine Speicheradresse (beispielsweise für Zuordnungen zwischen Zeitpunkten und Signalwerten) mindestens 32 Bit Speicher benötigt.

In Tabelle 2-6 entspricht die Anzahl an Spalten der Spaltenanzahl der Primäreingänge (Tabelle 2-5). Dies resultiert aus der Tatsache, daß die Ausgangssignalverläufe der Gatter aus den Signalwerten der Primäreingänge abgeleitet werden. Abbildung 2-1 stellt den Signalverlauf der in Tabelle 2-6 vorgestellten Signale grafisch dar.

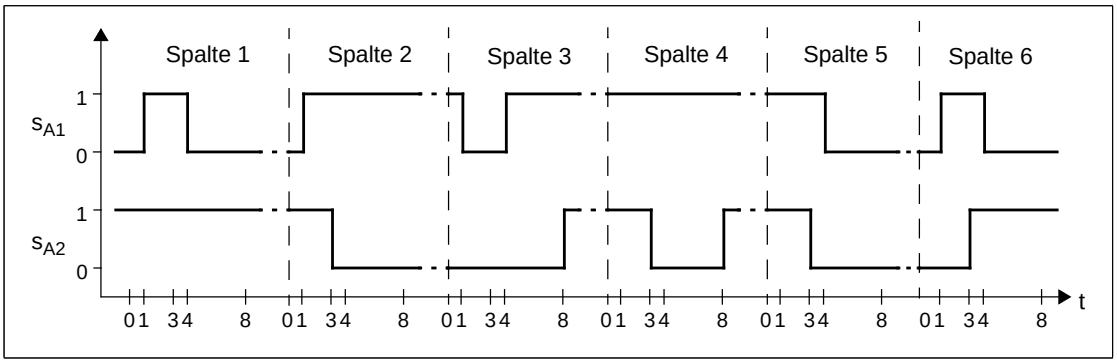


Abbildung 2-1: Grafische Darstellung der Signalverläufe aus Tabelle 2-6

Jeder Vektor wird für sich betrachtet und hat seinen eigenen Zeitrahmen entsprechend den vertikal gestrichelten Linien. Die Werte für die Zeit vor dem Nullpunkt entsprechen den statischen Anfangswerten. Mit t=0 schalten die Primäreingänge auf neue Werte und ab diesem Zeitpunkt wird das weitere Verhalten der Gatter betrachtet.

2.1.5 Die Verknüpfung von Signalen

Die Zuteilung eines eigenen Zeitsegments zu jedem Taktzyklus ermöglicht die *zeitparallele* Verknüpfung von Signalen über alle Taktzyklen hinweg. Zeitparallel bezieht sich dabei auf eine simultane Abarbeitung aller Taktzyklen, nicht aber deren einzelner Zeitpunkte. Die zeitparallele Verarbeitung bewirkt eine starke Beschleunigung der Simulation. Eine der Verknüpfungsmethoden, die von dem bestehenden Simulator von [DD 98] angeboten werden, realisiert das in [SCH 95] vorgestellte Modell. Dabei wird jede Zeile einer Signaltabelle als Bitkette dargestellt. Bei einer Maschine mit einer Wortbreite von 32 Bit werden somit bei jeder Verknüpfung 32 Bit echt parallel verknüpft! Generell ist die in dieser Arbeit vorgestellte Erweiterung auf alle zeitparallelen Logiksimulatoren anwendbar.

Als Beispiel seien die Signale s_{A1} und s_{A2} aus Tabelle 2-6 an den Eingängen eines UND-Gatters angeschlossen. Um die Ausgangswerte des UND-Gatters für t<0 zu bestimmen werden die entsprechenden Zeilen der beiden Signale UND-verknüpft (siehe Tabelle 2-7).

Signal	Zeitpunkt	1	2	3	4	5	6
s_{A1}	t<0	0	0	1	1	1	0
s_{A2}	t<0	1	1	0	1	1	0
UND	t<0	0	0	0	1	1	0

Tabelle 2-7: Zeilenweise Verknüpfung von Signalwerten

2.2 Die rechnerinterne Signaldarstellung

Signalverläufe wie in Tabelle 2-6 werden in Objekten der Klasse `Signalverlauf` gespeichert. Sie enthält den zeitlichen Verlauf eines Signals über alle Testvektoren hinweg. Die Klasse besteht aus einer einfach verketteten Liste von Zeitwerten, die nach den Zeitpunkten aufsteigend sortiert ist. Eine 0 in „next“ bezeichnet dabei das Ende der Liste. Ein Zeitwert besteht aus einem Zeitpunkt und den zugehörigen Werten des Signals über alle Testvektoren. Kursive Typnamen sollen andeuten, daß an der gegebenen Stelle eigentlich ein Zeiger auf den jeweiligen Typen existiert. Aus Gründen der Übersichtlichkeit wurden nur für den logischen Aufbau wichtige Zeiger explizit aufgeführt. Der erste Zeitwert ist immer der Anfangswert für t<0 und hat intern zur Identifikation als Zeitpunkt den Wert -1.

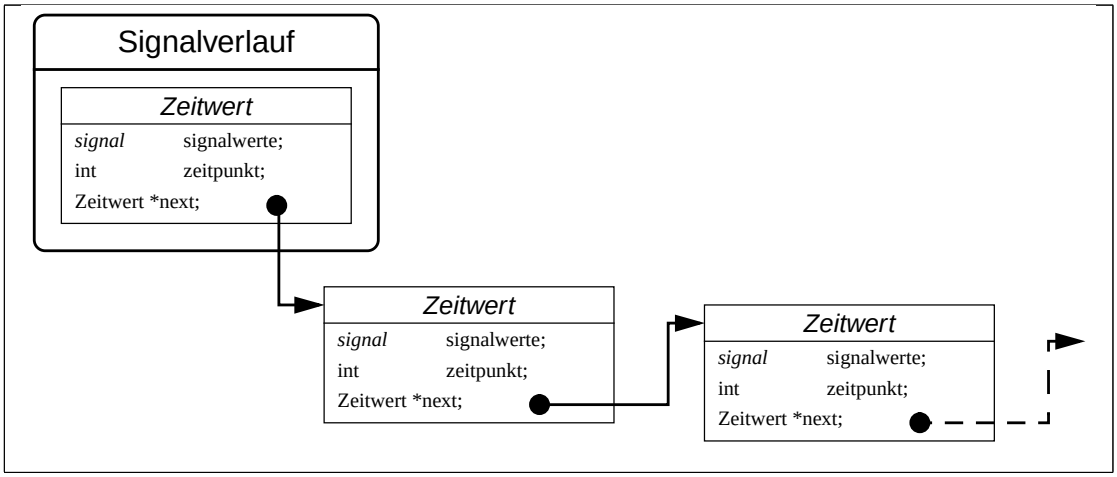


Abbildung 2-2: Die Klasse Signalverlauf

Die Signalwerte sind vom Typ *signal*, einer Klasse aus dem bestehenden Simulator von [DD 98]. Diese speichert die Signale entweder als Blöcke von Einsen (wie in [MB 98] beschrieben) oder als Bitketten nach [SCH 95]. Die gewünschte Darstellungsweise kann beim Simulationsstart festgelegt werden und ist größtenteils unabhängig von den in dieser Arbeit vorgestellten Erweiterungen.

Zur Verdeutlichung stellt Abbildung 2-3 die rechnerinterne Speicherung des Verlaufs des Signals s_{A1} aus Tabelle 2-6 als Beispiel dar:

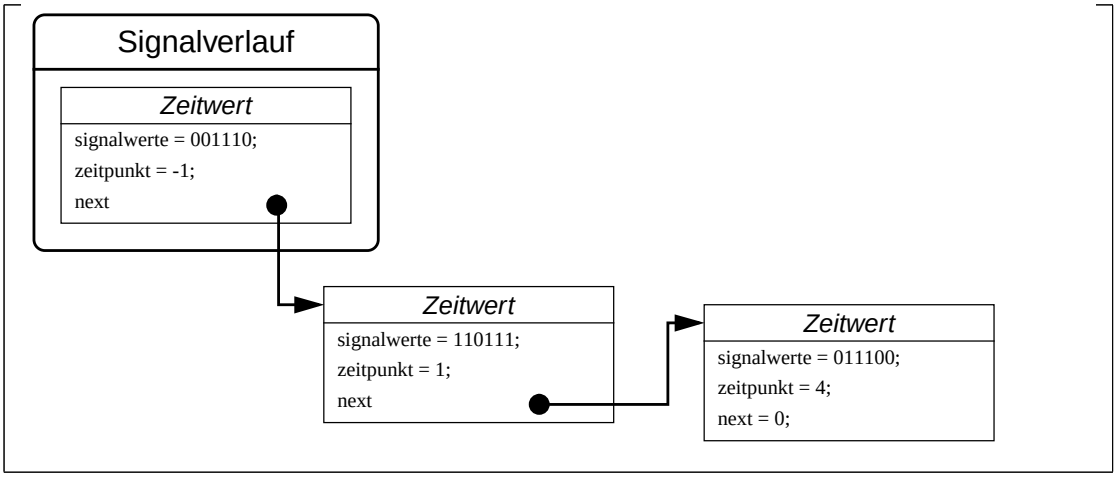


Abbildung 2-3: Interne Darstellung des Signals s_{A1}

Die Netzliste

Kapitel 3

In diesem Kapitel wird der rechnerinterne Aufbau des zu simulierenden Schaltkreises beschrieben. Um den Simulator verständlich und leicht erweiterbar zu halten, wurde auf eine möglichst naturgetreue Abbildung der Realität in eine objektorientierte Darstellung Wert gelegt. Der Schaltkreis wird als Netzliste beschrieben, die aus einer Vielzahl von Gattern besteht. Jedem Gatter ist ein bestimmter Komponententyp zugeordnet.

3.1 Komponenten

Der Gebrauch von Komponenten in der hier verwendeten Form ist dem in VHDL sehr ähnlich. In VHDL wird mit dem Konstrukt `component` ein Gattertyp definiert, der dann beliebig oft in der Schaltungsbeschreibung verwendet werden kann. Dies ist dual zu dem hier verwendeten Konzept, in dem jede Komponente einmal definiert wird und beliebig vielen Gattern zugeordnet werden kann.

Komponenten:	in VHDL	in dieser Arbeit

Deklaration	`component comp_name`
	`  [local_generic_clause]`
	`  [local_port_clause]`
	`end component;`
		Objekte der Klassen NOT, BUFFER, AND(n), NAND(n), OR(n), NOR(n), XOR(n) und XNOR(n)

Verwendung (Instanziierung)	`instantiation_label:`
	`  comp_name`
	`  [generic_map_aspect]`
	`  [port_map_aspect];`
		entsprechender Zeiger eines Objekts der Klasse Gatter auf die zugehörige Komponente

Tabelle 3-1: Dualität der Komponenten

Die Komponenten selbst als Klassen zu erzeugen scheitert an der Tatsache, daß dies während des Programmablaufs in C++ nicht möglich ist. Man könnte natürlich eine „Metaklasse“ entwerfen, mit der sich alle Gatter erzeugen ließen. Allerdings müßte man dann hohe Einbußen in Effizienz und Übersichtlichkeit des Programmcodes sowie einen größeren Speicheraufwand in Kauf nehmen.

3.1.1 Das Prinzip der virtuellen Funktionen

Um die Algorithmen, welche die Simulation der Gatter durchführen, klar und übersichtlich zu halten, werden sogenannte *virtuelle Funktionen* verwendet. Virtuelle Funktionen ermöglichen es in C++ dynamisch während des Programmablaufs einer Botschaft (d.h. einem Methodennamen) eine Methode zuzuordnen. Im Gegensatz dazu erfolgt die Bindung eines Namens an eine Funktion normalerweise zum frühestmöglichen Zeitpunkt, nämlich bei der Compilierung.

Im folgenden Beispiel soll die Funktionsweise virtueller Funktionen erläutert werden. Die Aufgabe soll darin bestehen, ein Feld zu definieren, das gleichzeitig Zeiger auf Elemente vom Typ `int`, `(char *)` und `float` enthalten kann. Zusätzlich soll jedes Element mit der gleichen Methode `print` ausgegeben werden können. Da dies direkt nicht möglich ist, braucht man als erstes einen einheitlichen Datentyp für die Feldelemente: Die Klasse `Element`.

```
class Element
{
  public:
    enum Datentyp { INTEGER, STRING, FLOAT };
    Element(int   x) { dtyp = INTEGER; intwert = x;};
    Element(char* x) { dtyp = STRING ; strwert = x;};
    Element(float x) { dtyp = FLOAT  ; fltwert = x;};

    void print(void);

  private:
    Datentyp dtyp;

    union {
      int   intwert;
      char* strwert;
      float fltwert;
    };
};

void Element::print(void) {
  switch (dtyp) {
    case INTEGER: cout << intwert; return;
    case STRING : cout << strwert; return;
    case FLOAT  : cout << fltwert; return;
  };
};
```

Listing 3-1: Klassische Lösung ohne virtuelle Funktionen

Eine Verwendung der Klasse `Element` ist wie folgt denkbar:

```
Element * feld[3];

feld[0] = new Element(12345);
feld[1] = new Element("Hello World");
feld[2] = new Element(67.89);

feld[0]->print();     // Ausgabe: 12345
feld[1]->print();     // Ausgabe: Hello World
feld[2]->print();     // Ausgabe: 67.89
```

Listing 3-2: Anwendung mit klassischer Lösung

Die klassische Lösung aus Listing 3-1 erinnert stark an C-Programmierstil. Der große Nachteil dieser Vorgehensweise liegt darin, daß jede Methode vom Datentyp abhängt und per `switch`-Anweisung über alle Möglichkeiten verzweigen muß. Zudem müssen bei Erweiterung um einen weiteren Datentyp alle Methoden geändert werden. Dies führt zu einem sehr unübersichtlichen und äußerst fehlerträchtigen Programm.

In C++ läßt sich dieses Problem elegant mittels virtueller Funktionen lösen:

```
class Element
{
  public:
    virtual void print(void) = 0;
};

class IntElement: public Element
{
  public:
    IntElement(int   x) { intwert = x;};
    void print(void)    { cout << intwert};
  private:
      int   intwert;
};

class StrElement: public Element
{
  public:
    StrElement(char* x) { strwert = x;};
    void print(void)    { cout << strwert};
  private:
      char*  strwert;
};
```

Listing 3-3: Neue Lösung mit virtuellen Funktionen

```
class FltElement: public Element
{
  public:
    FltElement(float x) { fltwert = x;};
    void print(void)    { cout << fltwert};
  private:
      float  fltwert;
};
```

Listing 3-3: Neue Lösung mit virtuellen Funktionen

In Listing 3-3 wird der Zusammenhang zwischen den unterschiedlichen Datentypen durch die gemeinsame Superklasse `Element` erzeugt, von der alle anderen Klassen abgeleitet sind. Das im Hauptprogramm zu erzeugende Datenfeld enthält auch hier Zeiger auf Objekte der Klasse `Element` wie bei der klassischen Lösung. Allerdings ist in diesem Fall die Klasse `Element` leer bis auf die virtuelle Funktion `print()`. Der Zusatz „`=0`“ weist darauf hin, daß die Definition der Methode nicht in dieser Klasse erfolgt: In diesem Fall wird `print()` als „reine virtuelle Funktion“ bezeichnet. Eine Klasse, die reine virtuelle Funktionen enthält, heißt „abstrakte Klasse“. Von abstrakten Klassen können keine Objekte instanziiert werden: Wie man in diesem Beispiel sieht, wäre ein Objekt der Klasse `Element` auch völlig sinnlos. Nun wird für jeden Datentyp eine eigene Klasse definiert, welche von der Klasse `Element` abgeleitet wird. In jeder dieser Klassen wird die virtuelle Funktion `print()` dem jeweiligen Datentyp entsprechend definiert.

Die gleiche Anwendung wie in Listing 3-2 sieht nun folgendermaßen aus:

```
Element * feld[3];

feld[0] = new IntElement(12345);
feld[1] = new StrElement(„Hello World“);
feld[2] = new FltElement(67.89);

feld[0]->print();     // Ausgabe: 12345
feld[1]->print();     // Ausgabe: Hello World
feld[2]->print();     // Ausgabe: 67.89
```

Listing 3-4: Anwendung mit neuer Lösung

Aufgrund der Realisierung von print() als virtuelle Funktion wird bei einem Aufruf von feld[0]->print() nicht die Methode `Element::print()` sondern die Methode der abgeleiteten Klasse `IntElement` aufgerufen. Es wird während des Programmablaufs festgestellt, zu welcher abgeleiteten Klasse das Objekt eigentlich gehört, obwohl es sich bei feld[0] nur um einen Zeiger auf die Superklasse `Element` handelt.

Im Gegensatz zur klassischen Lösung entfallen alle `switch`-Anweisungen. Jederzeit kann ein neuer Datentyp hinzugefügt werden, ohne daß an den anderen Klassen irgendetwas geändert werden muß. Der Aufbau ist klar und übersichtlich und alle datentyp-spezifischen Methoden sind in der jeweiligen Klasse vereint. Aufgrund der vielen Vorteile wurden diejenigen Klassen, welche in dieser Arbeit Komponententypen beschreiben, nach eben diesem Prinzip aufgebaut.

3.1.2 Die Komponentenklassen

Die im Rahmen dieser Arbeit entworfenen Komponentenklassen haben folgende Vererbungshierarchie:

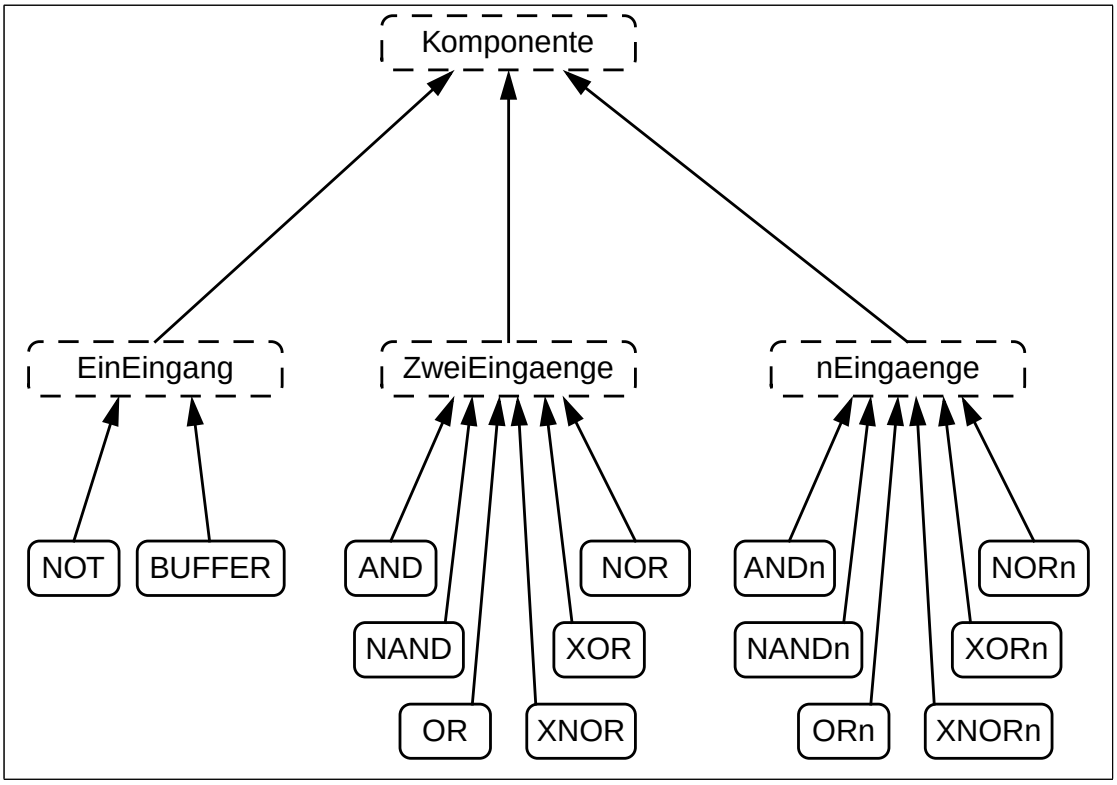


Abbildung 3-1: Vererbungshierarchie der Komponentenklassen

Die Pfeile sind von der Subklasse zu ihrer jeweiligen Superklasse gerichtet. Die gestrichelten Rahmen weisen auf abstrakte Klassen hin.

3.1.3 Die Klasse Komponente

Wie Abbildung 3-1 zu entnehmen ist, sind alle Komponentenklassen von der abstrakten Superklasse `Komponente` abgeleitet. Diese speichert die Gatterlaufzeiten für 0-1- und 1-0-Übergänge am Ausgang einer Komponente. Zusätzlich sind diverse statische Variablen enthalten, die beim Start einer Simulation initialisiert werden und dem vereinfachten Zugriff innerhalb aller unteren Komponentenklassen dienen. In erster Linie wird aber so die ständige dynamische Allozierung und Deallozierung dieser relativ umfangreichen Variablen während der Simulation vermieden, was bei Verwendung von `Auto`-Variablen unumgänglich wäre. Schließlich werden in dieser Klasse noch die Anzahl der Testvektoren und deren Gewichtung gespeichert. Die enthaltenen Methoden haben folgende Aufgabegebiete:

1. Verwaltung und Verarbeitung der statischen Variablen
2. Verwaltung und Ausgabe der Anzahl der Testvektoren und ihrer Gewichtung

3. Ausgabe der Gatterlaufzeiten
4. Berechnung des Energieverbrauchs für die Glitch-Simulation (rein virtuelle Funktionen)
5. Ermittlung der Anzahl der während der Simulation aufgetretenen Transitionen und Hazards
6. Ausgabe der Anzahl der Komponenteneingänge (rein virtuelle Funktion)
7. Auswahl der passenden Simulationsfunktion zur gewählten Art der Simulation
8. verschiedene Methoden der Simulation je nach Simulationsart (rein virtuelle Funktionen)
9. Durchführung der logischen Verknüpfung einer Komponente (rein virtuelle Funktion)

Die virtuellen Methoden zur Ausgabe der Anzahl der Komponenteneingänge und zur Durchführung der Simulation sind in den Klassen `EinEingang`, `ZweiEingaenge` und `nEingaenge` definiert; die Methode zur Durchführung der logischen Verknüpfung einer Komponente in den Klassen der untersten Hierarchieebene: `NOT`, `BUFFER`, `AND(n)`, `NAND(n)`, `OR(n)`, `NOR(n)`, `XOR(n)` und `XNOR(n)`. Gleichungen für die Berechnung des Energieverbrauchs wurden bisher in [WIL 98] nur für die Klassen `NOT`, `NAND(n)`, und `NOR(n)` entwickelt. In diesen Klassen werden die entsprechenden Methoden auch definiert.

3.1.4 Die Klassen EinEingang, ZweiEingaenge und nEingaenge

Die Klassifizierung der Komponenten nach Anzahl der Eingänge hilft die Effizienz der Simulation zu optimieren. Die Klasse `EinEingang` ist die Superklasse aller Klassen mit nur einem Eingang: `NOT` und `BUFFER`. Die Klasse `ZweiEingaenge` ist die Superklasse der Klassen mit genau zwei Eingängen: `AND`, `NAND`, `OR`, `NOR`, `XOR` und `XNOR`. Die Klasse `nEingaenge` entspricht der Klasse `ZweiEingaenge`, allerdings für Komponenten mit mehr als zwei Eingängen und ist Superklasse für `ANDn`, `NANDn`, `ORn`, `NORn`, `XORn` und `XNORn`. Die Klasse `ZweiEingaenge` wurde zusätzlich entwickelt, da ein hoher Anteil der verwendeten Komponenten nur zwei Eingänge haben und so im Vergleich zu der Klasse `nEingaenge` die Komplexität in der Simulation gesenkt werden kann. Es fallen einige Abfragen und Schleifen weg. Dies wirkt sich in einer gesteigerten Effizienz der Simulation aus.

Im Gegensatz zu den Klassen EinEingang und ZweiEingaenge wird die Anzahl der Gattereingänge bei der Klasse `nEingaenge` extra gespeichert. Alle drei Klassen enthalten Methoden mit folgenden Aufgaben:

- Ausgabe der Anzahl der Komponenteneingänge
- Definition der verschiedenen Simulationsmethoden

3.1.5 Die Klassen der untersten Hierarchieebene

Für jede unterstütze Verknüpfungsfunktion ist je eine Klasse vorhanden. Sie beinhalten alle die Definition der Methode zur Durchführung der logischen Verknüpfung einer Komponente. Von dieser Methode machen ausschließlich die Simulationsmethoden der Klassen `EinEingang`, `ZweiEingaenge` und `nEingaenge` Gebrauch. Zusätzlich enthalten die Klassen `NOT`, `NAND(n)`, und `NOR(n)` die von [WIL 98] entwickelten Methoden zur Berechnung des Energieverbrauchs für die Glitch-Simulation.

Klasse	logische Verknüpfung
NOT	Negation
BUFFER	Identität (nur Signalverzögerung)
AND(n)	Konjunktion
NAND(n)	Negation der Konjunktion
OR(n)	Disjunktion
NOR(n)	Negation der Disjunktion
XOR	Antivalenz
XORn	ungerade Parität (ungerade Anzahl an Eingängen mit dem Wert 1)
XNOR	Äquivalenz
XNORn	gerade Parität (gerade Anzahl an Eingängen mit dem Wert 1)

Tabelle 3-2: Logische Verknüpfung der Komponentenklassen

3.2 Gatter

Eine Netzliste, die einen Schaltkreis beschreibt, besteht im wesentlichen aus Gattern und deren Verbindungen. Wie schon in Tabelle 3-1 erwähnt, entspricht ein Objekt der Klasse `Gatter` der Instanziierung einer Komponente in VHDL.

3.2.1 Die Klasse Gatter

Die Klasse `Gatter` beinhaltet folgende Daten:

- ein Feld mit Zeigern auf die Gatter, die an den Eingängen angeschlossen sind. Die Dimension entspricht der durch den zugeordneten Komponententyp festgelegten Anzahl von Eingängen.
- einen Zeiger auf den Ausgabesignalverlauf, den das Gatter in Abhängigkeit von den Signalverläufen an seinen Eingängen produziert.
- die Anzahl der Gatter, die von diesem Gatter getrieben werden. Dieser Wert ist unter anderem für eine Simulation unter Beachtung des Energieverbrauchs durch Glitches nötig.
- die Anzahl der Gatter, die während dem Simulationsablauf noch getrieben werden müssen. Dieser Wert wird zu Beginn einer Simulation mit der Anzahl der getriebenen Gatter initialisiert und während der Simulation von jedem Gatter verringert, an dessen Eingang dieses Gatter angeschlossen ist. Sobald der Wert auf 0 sinkt, wird das Gatter im Simulationsablauf nicht mehr verwendet, der Ausgabesignalverlauf wird gelöscht und somit Speicher gespart.
- einen Zeiger auf den zugehörigen Komponententyp. Über diesen Zeiger kann die Anzahl der Eingänge des Gatters bestimmt werden. Zudem ist der Komponententyp für die Erzeugung des Ausgangssignalverlaufs in Abhängigkeit von der gewählten Simulationsart zuständig.
- einen Zeiger auf das nächste Gatter in der Auswertungsreihenfolge. Die Gatter sind in einer einfach verketteten Liste angeordnet. Die Anordnung garantiert, daß für ein Gatter in der Liste alle Gatter an seinen Eingängen davor eingereiht sind. Dadurch ist für die Simulation sichergestellt, daß die Signalverläufe der Gatter an den Eingängen des zu propagierenden Gatters schon berechnet worden sind. Aufgrund dieser Voraussetzung ist die Unterstützung sequentieller Schaltwerke nicht möglich.

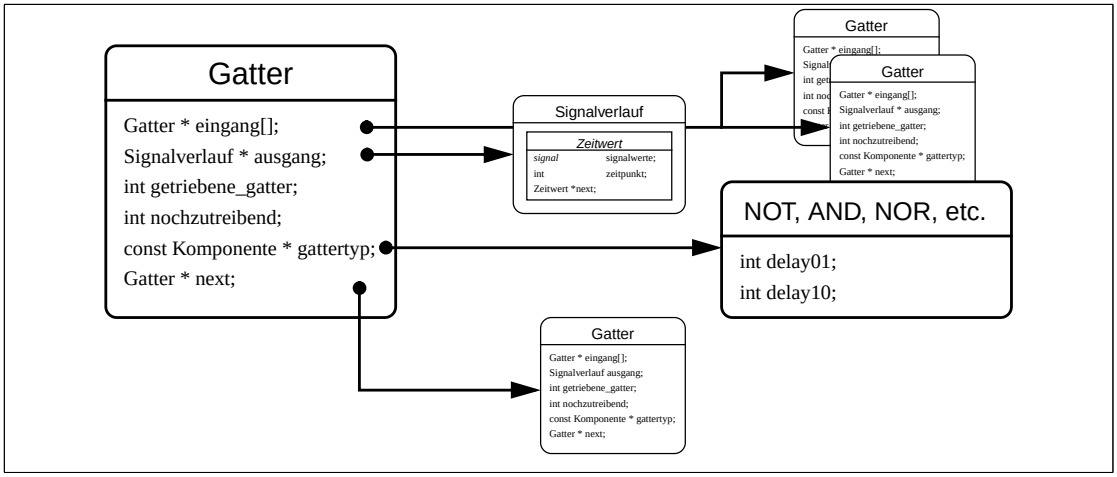


Abbildung 3-2: Die Klasse Gatter und ihre Verknüpfungen

Die Methoden in der Klasse `Gatter` erfüllen folgende Aufgaben:

- Zuordnung eines Ausgabesignalverlaufs (nur für Primäreingänge, siehe nächstes Kapitel)
- Initialisierung und Verminderung der Anzahl noch zu treibender Gatter und das Löschen des Ausgabesignalverlaufs sobald die Anzahl der zu treibenden Gatter den Wert 0 erreicht
- Simulation des Gatters. Dabei ruft das Gatterobjekt die Simulationsmethode des ihm zugeordneten Komponententyps auf (siehe Kapitel 3.4).
- Verwaltung und Ausgabe des Zeigers auf das nächste Gatter in der Auswertungsreihenfolge
- Ausgabe der Anzahl von Zeitwerten im Ausgabesignalverlauf.

3.2.2 Die Primäreingänge

Um den Simulationsablauf zu vereinfachen sind die Primäreingänge ebenfalls als Gatter realisiert. Zu Beginn einer Simulation werden die Simulationsstimuli als Signalverlauf aufbereitet und der Ausgabe dieser Gatter zugeordnet. Die Gatter in der Netzliste, welche direkt mit den Primäreingängen verbunden sind, werden nun stattdessen mit diesen speziellen “Primärgattern” verbunden. Dadurch erspart man sich eine gesonderte Behandlung der Primäreingänge. Alle existierenden Verbindungen bestehen nur zwischen Gattern. Für diese speziellen Gatter werden nur folgende Daten verwendet:

- der Zeiger auf den Ausgangssignalverlauf. Zu Beginn eines Simulationsablaufs wird aus den Testvektoren ein Signalverlauf für jeden Primäreingang erzeugt.
- die Anzahl der Gatter, die von diesem Gatter getrieben werden.
- die Anzahl der Gatter, die während dem Simulationsablauf noch getrieben werden müssen.

Die restlichen Daten bleiben ungenutzt und erhalten den Wert 0. Ebenso dürfen nur bestimmte Methoden auf einen Primäreingang angewandt werden:

- Zuordnung eines Ausgangssignalverlaufs
- Initialisierung und Verminderung der Anzahl noch zu treibender Gatter und das Löschen des Ausgabesignalverlaufs sobald die Anzahl der zu treibenden Gatter den Wert 0 erreicht

3.3 Netzliste

Die Klasse Netzliste besteht primär aus einer einfach verketteten Liste der Gatter des Schaltkreises. Die Information über die Verbindungen zwischen den Gattern sind in der Gatterklasse enthalten. Folgende Daten werden in der Klasse Netzliste gespeichert:

- die Anzahl der Gatter
- ein Zeiger auf den Einstiegspunkt in die Gatterliste

- die Anzahl an Primäreingängen und ein Feld mit Zeigern auf die entsprechenden Gatter (siehe Kapitel 3.2.2).
- das Ergebnis einer Simulation: Transitionen, Hazards und/oder Energieverbrauch
- der Simulationstyp

Die Methoden der Klasse Netzliste führen folgende Aufgaben durch:

- Aufbau der Komponententypen, der Gatter und die Zusammensetzung in einer Netzliste
- Simulation des Schaltkreises mit den übergebenen Testvektoren auf die gewünschte Simulationsart

Zusätzlich sind einige Methoden enthalten, welche die Umwandlung des bei [DD 98] verwendeten Netzlistenformates in die neue Darstellung unterstützten.

3.4 Die Verwendung virtueller Funktionen

Nachdem die Komponentenobjekte und die Netzliste eines Schaltkreises erzeugt worden sind, läuft die Simulation wie folgt ab: Wie schon in Kapitel 3.2.1 erläutert, sind die Gatter in einer verketteten Liste gespeichert. Für jedes Gatter ist sichergestellt, daß die an seinen Eingängen angeschlossenen Gatter weiter vorne in der Liste stehen. Werden die Gatter nun in der Reihenfolge, wie sie in der Liste gespeichert sind, durchsimuliert, so ist garantiert, daß die Ausgaben aller Gatter an den Eingängen des zu simulierenden Gatters bereits ermittelt worden sind.

3.4.1 Der Simulationsablauf eines Gatters

Zur Simulation eines Gatters wird die Methode `Gatter::propagiere()` aufgerufen. Da die Funktionsweise eines Gatters im jeweiligen Komponententyp verankert ist, wird die Simulationsaufforderung an die zugehörige Komponente weitergeleitet. Die Simulationsmethode erzeugt aus den an den Eingängen des Gatters anliegenden Signalverläufen einen dem Komponententyp entsprechenden Ausgabesignalverlauf.

In Tabelle 3-1 wurde bereits darauf hingewiesen, daß die Klasse `Gatter` ihre Zugehörigkeit zu einer bestimmten Komponente als Zeiger auf die Instanz der entsprechenden Komponentenklasse speichert. Man beachte, daß es sich zwecks Vereinheitlichung um einen Zeiger auf ein Objekt der Klasse `Komponente`, der Superklasse aller Komponentenklassen, handelt, obwohl es sich in Wirklichkeit um einen Zeiger auf ein Objekt der Klassen der untersten Hierarchieebene (`NOT`, `BUFFER`, `AND(n)`, `NAND(n)`, `OR(n)`, `NOR(n)`, `XOR(n)` oder `XNOR(n)`) handelt.

Listing 3-5 stellt den relevanten Code-Ausschnitt der Klasse `Gatter` dar. Das Gatter leitet die Simulationsaufforderung durch den Aufruf der Methode `Komponente::propagiere()` weiter.

```
class Gatter {
  Komponente * komponententyp;
  int propagiere(SimulationsArt simart){
    return (komponententyp->propagiere(simart)); };
};
```

Listing 3-5: Ausschnitt aus der Klasse Gatter

Wie man Listing 3-6 entnehmen kann, wählt die Klasse `Komponente` anhand der übergebenen Simulationsart die passende Simulationsmethode aus. Bei dieser handelt es sich nun um eine virtuelle Funktion. Dadurch wird automatisch während der Laufzeit der eigentliche Komponententyp (eine Klasse der untersten Hierarchieebene), bei dem die Simulationsmethode definiert ist, ermittelt und die entsprechende Methode ausgeführt.

```
class Komponente {
  int propagiere(SimulationsArt simart){
    switch (simart) {
      case ohneDelays:
        return propagiere_oD();
      case mitDelays:
        return propagiere_mD();
    };
  };

  virtual int propagiere_oD()=0;
  virtual int propagiere_mD()=0;
};
```

Listing 3-6: Ausschnitt aus der Klasse Komponente

Ohne Verwendung von virtuellen Funktionen müßte während der Simulation jedes Gatter eine Fallunterscheidung bezüglich des Komponententyps durchführen, dem es angehört, und anschließend eine entsprechende Methode zur Simulation des Gatters aufrufen. Diese Fallunterscheidung fällt aufgrund der virtuellen Funktionen weg.

Auch das Hinzufügen neuer Komponententypen ist durch die Verwendung virtueller Funktionen erheblich vereinfacht worden. Dies kann ohne jegliche Modifikation an den anderen Komponentenklassen vollzogen werden.

3.4.2 Beispiel des Simulationsablaufs eines Gatters

Listing 3-7 soll als Beispiel für den Aufruf virtueller Funktionen dienen. In Zeile 1 wird eine Nand-Komponente und in Zeile 2 ein Gatter instanziiert. Bei dem Gatter soll es sich um ein NAND-Gatter handeln. Daher wird in Zeile 3 dem Gatter die NAND-Komponente durch einen Verweis in `Gatter::komponententyp` zugeordnet. Im Laufe der Simula-

tion kommt es zum Aufruf der Methode `propagiere` (Zeile 4). Diese Anweisung stößt

```
1   NAND nandkomponente;
2   Gatter beispielgatter;

3   beispielgatter.komponententyp = &nandkomponente;
4   beispielgatter.propagiere(simart);
```

Listing 3-7: Beispiel zu virtuellem Funktionsaufruf

die Simulation des Gatters an. Wie aus Listing 3-5 ersichtlich bewirkt sie den Aufruf der Funktion komponententyp->propagiere(simart). Die Klasse `Komponente` entscheidet anhand der übergebenen Simulationsart, welche Simulationsmethode anzuwenden ist und ruft diese auf. Bei der gewählten Simulationsmethode handelt es sich um eine virtuelle Funktion. So wird zunächst dynamisch der tatsächliche Komponententyp bestimmt (`nandkomponente` = ein Objekt der Klasse `NAND`) und dessen Simulationsmethode ausgeführt.

Die Simulationsmodelle

Kapitel 4

Die in dieser Arbeit vorgestellte Erweiterung des bestehenden Simulators von [DD 98] bietet vier Arten der Simulation an:

- eine **Zero-Delay-Simulation** ohne Berücksichtigung von Gatterlaufzeiten,
- eine **Transport-Delay-Simulation** mit einer einheitlichen Gatterlaufzeit für 0-1- und 1-0-Übergänge am Ausgang eines Gatters,
- eine **Real-Delay-Simulation** mit verschiedenen Gatterlaufzeiten für 0-1- und 1-0-Übergänge am Gatterausgang und der Berücksichtigung träger Totzeiten und
- eine **Glitch-Simulation** mit verschiedenen Gatterlaufzeiten für 0-1- und 1-0-Übergänge am Gatterausgang, Berücksichtigung träger Totzeiten und zusätzlicher Berechnung des Energieverlusts durch Glitches.

Sie unterscheiden sich in ihren Laufzeiten und ihrer Genauigkeit bezüglich des zu schätzenden Energiebedarfs. Als Ergebnis liefern die Simulationen in Abhängigkeit der gewählten Simulationsarten die Anzahl an aufgetretenen Transitionen, Hazards oder direkt den im Schaltkreis aufgetretenen Energieverlust:

Simulationsart	Ausgabe
Zero-Delay	Anzahl an aufgetretenen Transitionen
Transport-Delay	Anzahl an aufgetretenen Transitionen und Hazards
Real-Delay	Anzahl an aufgetretenen Transitionen und Hazards
Glitch	Anzahl an aufgetretenen Transitionen und Hazards und absoluter Energieverbrauch

Tabelle 4-1: Ausgaben der verschiedenen Simulationsarten

Wie in Kapitel 1.1.1 beschrieben läßt die Anzahl der aufgetretenen Transitionen eine Abschätzung des Energiebedarfs der Schaltung zu.

Das Beispiel

Zur Verdeutlichung der Implementierungen wird jede Simulationsart anhand eines Beispiels erläutert. Simuliert wird ein UND-Gatter mit zwei Eingängen. Weiter wird angenommen, daß das Gatter an den Eingängen von anderen Gattern getrieben wird und nicht direkt mit den Primäreingängen der Schaltung verbunden ist. Dadurch kann es an den Eingängen zu komplexeren Signalverläufen kommen, als bei Primäreingängen möglich wären. Bei Primäreingängen sind ja definitionsgemäß nur ein Anfangswert für t<0 und ein Endwert für t=0 möglich.

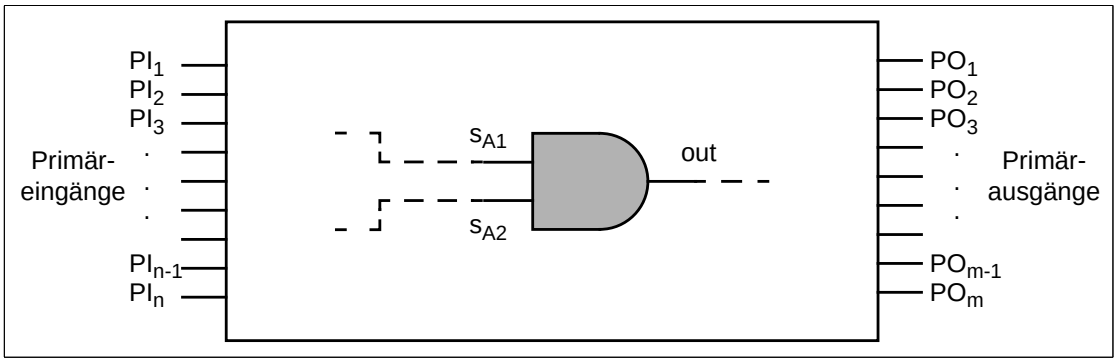


Abbildung 4-1: Internes UND-Gatter in einer Schaltung

Für die Eingänge werden die schon in Kapitel 2.1.4 vorgestellten Signale s_{A1} und s_{A2} verwendet:

Signal	Zeitpunkt	1	2	3	4	5	6
1. Eingang: s_{A1}	t<0	0	0	1	1	1	0
	t=1	1	1	0	1	1	1
	t=4	0	1	1	1	0	0
2. Eingang: s_{A2}	t<0	1	1	0	1	1	0
	t=3	1	0	0	0	0	1
	t=8	1	0	1	1	0	1
Gewichtung		1	3	1	1	3	1

Tabelle 4-2: Eingangssignale für Beispiel-Gatter

4.1 Allgemeiner Ablauf einer Simulation

Zu Anfang einer Simulation wird zunächst aus einer VHDL-Datei eine Netzliste erzeugt. Diese Netzliste kann dann beliebig oft mit unterschiedlichen Testvektorsequenzen unter den verschiedenen Modellen zur Simulation verwendet werden. Beim Start der Simulation wer-

den die Testvektoren mit ihrer Gewichtung übergeben und die Art der Simulation festgelegt. Die Testvektoren werden als Ausgangssignalverläufe derjenigen Gatter aufbereitet, welche die Primäreingänge der Schaltung repräsentieren (siehe auch Kapitel 3.2.2). Es folgt eine Iteration über alle Gatter entsprechend der Reihenfolge in der sie in der verketteten Liste abgespeichert sind. Wie in Kapitel 3.2.1 schon erwähnt, garantiert diese Reihenfolge, daß bei einem zu simulierenden Gatter alle Gatter an dessen Eingängen vorher in der Liste auftreten. Dadurch wird sichergestellt, daß die benötigten Gatter schon durchpropagiert wurden, ihr Ausgangssignalverlauf also bereits ermittelt wurde und an den Eingängen des nun zu simulierenden Gatters zur Verfügung steht. Jedes Gatter wird sukzessive über eine entsprechende Methode zur Simulation aufgefordert. Die Gatter geben diese Aufforderung an ihre zugehörigen Komponenten weiter (siehe Kapitel 3.4.1), welche schließlich den Ausgangssignalverlauf und die Anzahl an aufgetretenen Transitionen bzw. Hazards oder den absoluten Energieverbrauch -je nach gewählter Simulationsart- ermitteln.

Es folgt eine detaillierte Beschreibung der verschiedenen Simulationsmodi. Am Anfang wird jeweils der theoretische Hintergrund eines Modells erläutert, welches dann im zweiten Abschnitt anhand eines Beispiels veranschaulicht wird. Am Schluß enthält jede Beschreibung den Pseudocode, welcher die jeweilige Simulationsart realisiert. Dieser Pseudocode soll dazu dienen Unklarheiten und Mehrdeutigkeiten auszuräumen.

4.2 Die Zero-Delay-Simulation

4.2.1 Das Zero-Delay-Modell (ZDM)

In diesem Modell wird von jedem Gatter angenommen, daß es unendlich schnell schaltet: Alle Gatter haben eine Laufzeit von 0 Zeiteinheiten. Dies hat zur Folge, daß jeder Signalverlauf während der Simulation nur zwei Zeitwerte hat:

- einen Anfangswert für $t<0$ und
- einen Endwert für $t = 0$.

Durch diese Vereinfachung läuft die Simulation mit Abstand am schnellsten von allen vier Arten ab und hat den geringsten Speicherbedarf. Allerdings werden alle auftretenden Hazards und Glitches vernachlässigt.

4.2.2 Die Realisierung des Zero-Delay-Modells

Durch die eben erwähnte Einschränkung gestaltet sich die Realisierung auch sehr einfach und effizient. Zur Erläuterung dieser Simulationsart seien an den Eingängen des Beispielgatters die Signalverläufe aus Tabelle 4-3 gegeben. Diese Signalverläufe entsprechen den Signalen s_{A1} und s_{A2} aus Tabelle 4-2 im Verlauf einer ZDM-Simulation, d.h. alle Zwischenwerte werden ignoriert und der Endwert tritt bei $t=0$ ein.

Signal	Zeitpunkt	1	2	3	4	5	6
1. Eingang (s_{A1})	Anfanswert (t<0)	0	0	1	1	1	0
	Endwert (t=0)	0	1	1	1	0	0
2. Eingang (s_{A2})	Anfanswert (t<0)	1	1	0	1	1	0
	Endwert (t=0)	1	0	1	1	0	1

Tabelle 4-3: Signalverlauf an den Eingängen des UND-Gatters im ZDM

Zunächst werden alle Anfangswerte der Eingangssignale entsprechend der logischen Funktion der zugehörigen Komponente verknüpft und als Anfangswert des Ausgabesignalverlauf gespeichert. Eine UND-Verknüpfung der Anfangswerte des ersten und zweiten Eingangs ergibt:

Anfangswerte für t<0	1	2	3	4	5	6
1. Eingang	0	0	1	1	1	0
2. Eingang	1	1	0	1	1	0
Ausgang	0	0	0	1	1	0

Tabelle 4-4: Ermittlung des Anfangswerts des Ausgabesignalverlauf

Durch die Verknüpfung aller Endwerte der Eingabesignale erhält man den Endwert des Ausgabesignalverlaufs. Dieser tritt aufgrund der angenommen Gatterlaufzeit von 0 Zeiteinheiten zum Zeitpunkt t=0 ein:

Endwerte für t=0	1	2	3	4	5	6
1. Eingang	0	1	1	1	0	0
2. Eingang	1	0	1	1	0	1
Ausgang	0	0	1	1	0	0

Tabelle 4-5: Ermittlung des Endwerts des Ausgabesignalverlauf

Zur Berechnung der Anzahl an aufgetretenen Transitionen folgt die Exklusiv-Oder-Verknüpfung zwischen dem Anfangs- und Endwert des Ausgabesignalverlaufs. Die aus der Verknüpfung resultierenden Einsen geben an, ob bei dem jeweiligen Vektor im Laufe der Simulation eine Transition stattgefunden hat.

	1	**2**	**3**	**4**	**5**	**6**
Anfangswert	0	0	0	1	1	0
Endwert	0	0	1	1	0	0
Exklusiv-Oder	0	0	1	0	1	0

Tabelle 4-6: Exklusiv-Oder-Verknüpfung der Ausgabewerte

Zuletzt werden die resultierenden Werte der Exklusiv-Oder-Verknüpfung mit den Gewichtungen der Testvektoren aus Tabelle 4-2 multipliziert und alle Produkte aufsummiert.

	1	**2**	**3**	**4**	**5**	**6**
Exklusiv-Oder	0	0	1	0	1	0
Gewichtung der Testvektoren	1	3	1	1	3	1
aufgetretene Transitionen	0	0	1	0	3	0

Tabelle 4-7: Ermittlung der gewichteten Anzahl an Transitionen

Die Summe 1+3=4 entspricht der Anzahl der in diesem Gatter während der gesamten Simulation aufgetretenen Transitionen.

4.2.3 Das Zero-Delay-Modell im Pseudocode

```
Anfangswert = logische Verknüpfung der Eingangssignale bei t<0;
Erstellung des Ausgabesignalverlaufs mit Anfangswert als ersten Eintrag für t<0;
Endwert = logische Verknüpfung der Eingangssignale bei t=0;
Erweiterung des Ausgabesignalverlaufs um Endwert für t=0;
```

Listing 4-1: Pseudocode für Zero-Delay-Simulation

4.3 Die Transport-Delay-Simulation

4.3.1 Das Transport-Delay-Modell (TDM)

Dieses Modell trägt seine Bezeichnung aufgrund der Ähnlichkeit mit dem Transport-Delay-Mechanismus zur Signalpropagierung in VHDL (siehe [PA 90]). Der Mechanismus bewirkt eine zeitliche Verzögerung eines Ausgabesignals ohne Berücksichtigung der trägen Totzeit der Gatter. Im Gegensatz zu VHDL gilt hier zusätzlich die Einschränkung, daß die Laufzeiten eines jeden Gatters für 0-1- und 1-0-Übergänge jeweils identisch sind. Nur dadurch wird eine Beschleunigung im Vergleich zur Real-Delay-Simulation (siehe Kapitel 4.4) erreicht. Unterschiedliche Gatter können aber nach wie vor verschiedene Laufzeiten haben. Während beim Zero-Delay-Modell zu wenig Transitionen aufgrund der Vernachlässigung von Gatterlaufzeiten ermittelt werden, ist beim Transport-Delay-Modell das Gegenteil der Fall: Das Herausfallen von Transitionen aufgrund von trägen Totzeiten wird ignoriert und dadurch werden zu viele Transitionen ermittelt.

4.3.2 Die Realisierung des Transport-Delay-Modells

Als Beispiel diene wieder das UND-Gatter mit den Signalen s_{A1} und s_{A2} aus Tabelle 4-2 an den Eingängen. Wie beim Zero-Delay-Modell wird auch hier zuerst der Anfangswert des Ausgabesignalverlaufs durch Verknüpfung der Anfangswerte der Eingangssignale entsprechend der logischen Funktion des Komponententyps des Gatters ermittelt:

Anfangswerte für t < 0	**1**	**2**	**3**	**4**	**5**	**6**
1. Eingang	0	0	1	1	1	0
2. Eingang	1	1	0	1	1	0
Ausgang	0	0	0	1	1	0

Tabelle 4-8: Ermittlung des Anfangswerts des Ausgabesignalverlauf

Da nun Gatterlaufzeiten eine Rolle spielen und somit die Eingangssignalverläufe mehr als zwei Zeitpunkte aufweisen, ist nicht vorhersagbar, wann welcher Eingang als nächster seinen Signalwert ändert. Also wird erst einmal der früheste Zeitpunkt $t_{Simulation}$ ermittelt, zu dem mindestens einer der Eingänge einen neuen Signalwert erhält. Im Beispiel ändert sich als erstes der Eingang 1 zum Zeitpunkt 1. Die zu diesem Zeitpunkt gültigen Signalwerte aller Eingangswerte werden verknüpft und als Signalwert für den Ausgang vorgemerkt (siehe Tabelle 4-9).

Der Zeitpunkt, zu dem dieser Signalwert in Kraft tritt, ergibt sich aus der Addition des Zeitpunkts $t_{Simulation}$ und der Gatterlaufzeit. In diesem Beispiel betrage die Gatterlaufzeit zwei Zeiteinheiten. Somit wird die neue Ausgabe zum Zeitpunkt $t=t_{Simulation}+2=3$ wirksam.

$t_{Simulation}=1$	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=1	1	1	0	1	1	1
2. Eingang	t<0	1	1	0	1	1	0
Ausgang	$t_{Simulation}+2=3$	1	1	0	1	1	0

Tabelle 4-9: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}=1$

Anschließend wird $t_{Simulation}$ auf den nächsten Zeitpunkt gesetzt, zu dem sich eines der Eingabesignale ändert. Der nächste Signalwechsel findet bei t=3 am zweiten Eingang statt, also wird $t_{Simulation}$ auf 3 erhöht. Wieder werden alle zu diesem Zeitpunkt gültigen Eingangswerte verknüpft und mit einer Verzögerung von 2 Zeiteinheiten zum Ausgang durchpropagiert:

$t_{Simulation}=3$	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=1	1	1	0	1	1	1
2. Eingang	t=3	1	0	0	0	0	1
Ausgang	$t_{Simulation}+2=5$	1	0	0	0	0	1

Tabelle 4-10: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}=3$

Der nächste Zeitpunkt einer Änderung ist $t_{Simulation}=4$, zu dem wieder der erste Eingang neue Signalwerte erhält. Wie gehabt werden die Einganssignale verknüpft und um zwei Zeiteinheiten verzögert dem Ausgabesignalverlauf hinzugefügt:

$t_{Simulation}=4$	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=4	0	1	1	1	0	0
2. Eingang	t=3	1	0	0	0	0	1
Ausgang	$t_{Simulation}+2=6$	0	0	0	0	0	0

Tabelle 4-11: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}=4$

Die letzte Änderung findet bei t=8 am zweiten Eingang statt. Tabelle 4-12 zeigt das Ergebnis der Verknüpfung unter Berücksichtigung der Verzögerung.

$t_{Simulation}$=8	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=4	0	1	1	1	0	0
2. Eingang	t=8	1	0	1	1	0	1
Ausgang	$t_{Simulation}$+2=10	0	0	1	1	0	0

Tabelle 4-12: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=8

Beide Eingangssignale haben ihren Endwert erreicht und es ergibt sich der folgende Ausgabesignalverlauf:

Zeitpunkt	1	2	3	4	5	6
t<0	0	0	0	1	1	0
t=3	1	1	0	1	1	0
t=5	1	0	0	0	0	1
t=6	0	0	0	0	0	0
t=10	0	0	1	1	0	0

Tabelle 4-13: Der resultierende Ausgabesignalverlauf

An Tabelle 4-13 soll nun die Forderung nach identischen Gatterlaufzeiten für 1-0- und 0-1-Übergänge verdeutlicht werden. Man betrachte den Übergang von Zeitpunkt t=3 zu Zeitpunkt t=5. Bei den Testvektoren 2, 4 und 5 wechselt das Signal von 1 nach 0. Dagegen wechselt es beim Testvektor 6 von 0 zu 1. Hätte man nun unterschiedliche Gatterlaufzeiten für diese beiden Übergänge, so müßte man jeden Testvektor wie beim Real-Delay- und beim Glitch-Modell separat betrachten. Beim Transport-Modell hingegen haben beide Übergänge eine einheitliche Gatterlaufzeit von 2 Zeiteinheiten. Dadurch kann beim Ausgabsignalverlauf jede Zeile auf einmal erstellt werden.

Es folgen die Auswertung des Ausgabesignalverlaufs und die Berechnung der Anzahl an aufgetretenen Transitionen und Hazards: Alle Signalwerte des Ausgangs werden in ihrer zeitlichen Reihenfolge paarweise exklusiv-Oder-verknüpft und die resultierenden Einsen für jeden Vektor aufsummiert (siehe Tabelle 4-14).

Jede Summe entspricht der Anzahl der aufgetretenen Transitionen bei dem jeweiligen Testvektor. Die Hazardanzahl gleicht der Anzahl an Transitionen, soweit es sich dabei um eine gerade Zahl handelt. Ungerade Transitionszahlen werden um 1 vermindert der Hazardanzahl zugewiesen. Diese Unterscheidung findet statt, da bei einer Signaländerung zwischen dem Anfangs- und Endwert innerhalb eines Testvektors genau eine der Transitionen für den Signalwechsel verantwortlich ist. Alle weiteren Transitionen sind ungewollt und somit definitionsgemäß Hazards.

Ausgabesignalverlauf							Exklusiv-Oder-Verknüpfung					
t<0	0	0	0	1	1	0						
							1	1	0	0	0	0
t=3	1	1	0	1	1	0						
							0	1	0	1	1	1
t=5	1	0	0	0	0	1						
							1	0	0	0	0	1
t=6	0	0	0	0	0	0						
							0	0	1	1	0	0
t=10	0	0	1	1	0	0						
Summe (Transitionen)							2	2	1	2	1	2
Hazards							2	2	0	2	0	2

Tabelle 4-14: Berechnung der Transitionen und Hazards

Die Anzahl an Transitionen und Hazards pro Testvektor werden mit der dem Testvektor entsprechenden Gewichtung multipliziert und die Summe über alle Produkte gebildet. Die resultierenden Summen entsprechen der Transitions- und Hazardanzahl, die während der gesamten Simulation an diesem Gatter aufgetreten sind (siehe Tabelle 4-15).

Anzahl pro Testvektor	Transitionen	2	2	1	2	1	2	Gesamt-summe
	Hazards	2	2	0	2	0	2	
	Gewichtung	1	3	1	1	3	1	
Gewichtete Anzahl	Transitionen	2	6	1	2	3	2	16
	Hazards	2	6	0	2	0	2	12

Tabelle 4-15: Gesamtsumme der Transitionen und Hazards

4.3.3 Das Transport-Delay-Modell im Pseudocode

```
Ausgabe = logische Verknüpfung der Eingangssignale bei t<0;
Erstellung des Ausgabesignalverlaufs mit Ausgabe als ersten Eintrag für t<0;
t_Simulation = Zeitpunkt der ersten Signaländerung an den Gattereingängen;
REPEAT {
```

Listing 4-2: Der Pseudocode des Transport-Delay-Modells

```
    Ausgabe = logische Verknüpfung der zu t_Simulation gültigen Eingangssignale;
    Erweiterung des Ausgabesignalverlaufs um Ausgabe für t=t_Simulation+Verzögerung;
    t_Simulation = Zeitpunkt der nächsten Signaländerung an den Gattereingängen;
} UNTIL (es finden keine Signaländerungen mehr an den Gattereingängen statt);
```

Listing 4-2: Der Pseudocode des Transport-Delay-Modells

4.4 Die Real-Delay-Simulation

4.4.1 Das Real-Delay-Modell (RDM)

Das Real-Delay-Modell ist das exakteste Simulationsverfahren auf der logischen Ebene (siehe auch [KK 98]). Wie beim Transport-Delay-Modell werden auch in diesem Modell Gatterlaufzeiten berücksichtigt. Allerdings können sich nun die Laufzeiten für 1-0- und 0-1-Übergänge am Gatterausgang unterscheiden.

Zusätzlich wird ab jetzt die träge Totzeit in die Betrachtung miteinbezogen. Die träge Totzeit gibt die minimale Länge an, die der Ausgabeimpuls eines Gatters haben muß, um durch das Gatter durchpropagiert zu werden. Unterschreitet ein Impuls diesen Grenzwert, so wird er verschluckt. Diese Vorgehensweise dient zur Berücksichtigung der Zeit, die jedes Gatter zur Reaktion auf Signaländerungen an den Eingängen benötigt. Dabei wird angenommen, daß die Reaktionszeiten der Gatter genauso lang sind wie deren Verzögerungszeiten.

Durch die erhöhte Komplexität des Real-Delay-Modells benötigt diese Methode von den bisher vorgestellten Modellen die meiste Rechenzeit.

4.4.2 Die Realisierung des Real-Delay-Modells

Um eine Gegenüberstellung zu vereinfachen, werden in diesem Kapitel die gleichen Eingangssignalverläufe (aus Tabelle 4-2) wie beim Transport-Modell verwendet. Auch in diesem Modell werden als erstes die Anfangswerte aller Eingänge dem Komponententyp entsprechend verknüpft und als Anfangswert des Ausgabesignalverlaufs gespeichert (siehe Tabelle 4-8):

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0

Tabelle 4-16: Anfangswert des Ausgabesignalverlauf

Von nun an unterscheidet sich die Vorgehensweise beträchtlich von den bisher vorgestellten Verfahren. Da verschiedene Gatterlaufzeiten für 1-0- und 0-1-Übergänge zu beachten sind, muß nun, wie schon in Kapitel 4.3.2 angedeutet, jeder Übergang für sich betrachtet werden.

Transitionen, die am Ausgang aufgrund von Signaländerungen an den Gattereingängen auftreten, werden in diesen Modell nicht sofort durchgeführt. Für sie wird erst einmal berechnet, wann sie in Abhängigkeit von der Gatterlaufzeit bis zum Ausgang des Gatters durchpropagiert sein werden. Der berechnete Zeitpunkt wird in einem Feld namens *Zeitplan* vermerkt. Die Transition wird schließlich ausgeführt, sobald die Simulationszeit den entsprechenden Wert erreicht. Durch diese Vorgehensweise erhält man die Möglichkeit Glitches zu erkennen und entsprechende Ausgabeimpulse zu unterdrücken. Ein Glitch tritt nämlich genau dann auf, wenn im Zeitplan bereits eine durchzuführende Transition eingetragen ist und während dessen eine weitere Signaländerung auftritt.

Nach der Ermittlung des Anfangswert wird der früheste Zeitpunkt $t_{Eingabe}$ ermittelt, an dem mindestens eines der Eingangssignale seinen Wert ändert. In diesem Beispiel ändert das Signal s_{A1} zum Zeitpunkt 1 als erstes seinen Wert. Alle zu diesem Zeitpunkt gültigen Eingangssignale werden verknüpft:

$t_{Eingabe}=1$	Zeitpunkt	1	2	3	4	5	6
1. Eingang (s_{A1})	t=1	1	1	0	1	1	1
2. Eingang (s_{A2})	t<0	1	1	0	1	1	0
Ausgang		1	1	0	1	1	0

Tabelle 4-17: Ermittlung des Ausgabesignalverlauf für $t_{Eingabe}=1$

Nun kann allerdings für das Ergebnis kein pauschaler Zeitpunkt wie bei der Transport-Delay-Simulation angegeben werden, zu dem alle Ausgabewerte bis zum Gatterausgang durchpropagiert sein werden. Hier muß jeder Vektor untersucht werden, ob bei ihm eine Transition auftritt und falls ja, ob diese von 0 nach 1 oder von 1 nach 0 geht. Zu diesem Zweck wird der Anfangswert des Ausgabesignalverlaufs (Tabelle 4-8) mit dem Ergebnis der letzten Verknüpfung (bei $t_{Eingabe}=1$: Tabelle 4-17) verglichen.

Zeitpunkt	1	2	3	4	5	6
Anfangswert	0	0	0	1	1	0
Verknüpfung bei $t_{Eingabe}=1$	1	1	0	1	1	0

Tabelle 4-18: Vergleich des Anfangswerts und der Verknüpfung bei $t_{Eingabe}=1$

Bei den Vektoren 1 und 2 tritt ein Übergang von 0 nach 1 auf. Alle anderen Signale bleiben konstant. Es folgt die Berechnung der Zeitpunkte, zu denen diese Transitionen bis zum Gatterausgang durchpropagiert sein werden. Dazu seien in diesem Beispiel folgende Verzögerungszeiten gegeben:

Übergang von	Verzögerung
0 nach 1	3 Zeiteinheiten
1 nach 0	1 Zeiteinheit

Tabelle 4-19: Verzögerungszeiten für 0-1- und 1-0-Übergänge

Die gesuchten Zeitpunkte ergeben sich aus der Summe des aktuellen Simulationszeitpunkts $t_{Simulation}=1$ und der entsprechenden Verzögerungszeit. Für die beiden Vektoren 1 und 2 ist ein 1-0-Übergang zu planen (siehe Tabelle 4-18).

Dieser wird zum Zeitpunk $t_{Simulation}$ zuzüglich 3 Zeiteinheiten für die Verzögerung (siehe Tabelle 4-19) eintreten:

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
$t_{Simulation}$=1	**1 + 3 = 4**	**4**	-	-	-	-

Tabelle 4-20: Der Zeitplan für $t_{Simulation}$=1

Dem Zeitplan aus Tabelle 4-20 ist zu entnehmen, daß die nächste Änderung der Gatterausgabe zum Zeitpunkt 4 ansteht. Dieser Zeitpunkt wird als $t_{Ausgabe}$ bezeichnet.

Der Zeitpunkt des nächsten Ereignisses an den Gattereingängen $t_{Eingabe}$ wird nun auf 3 erhöht. Zu diesem Zeitpunkt ändert der 2. Eingang seinen Wert. Der Simulationszeitpunkt $t_{Simulation}$ wird auf den Zeitpunkt $t_{Eingabe}$ oder $t_{Ausgabe}$ gesetzt. Dies hängt davon ab, welches Ereignis zuerst eintritt. In diesem Fall ist $t_{Eingabe}<t_{Ausgabe}$ und somit $t_{Simulation}$=3. Wie gehabt werden die aktuell gültigen Eingangssignalwerte verknüpft:

$t_{Simulation}$=3	**Zeitpunkt**	**1**	**2**	**3**	**4**	**5**	**6**
1. Eingang	t=1	1	1	0	1	1	1
2. Eingang	t=3	1	0	0	0	0	1
Ausgang		1	0	0	0	0	1

Tabelle 4-21: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=3

Das Resultat wird mit dem Ergebnis der vorigen Verknüpfung (siehe Tabelle 4-17) verglichen:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
Verknüpfung bei $t_{Eingabe}$=1	1	1	0	1	1	0
Verknüpfung bei $t_{Eingabe}$=3	1	0	0	0	0	1

Tabelle 4-22: Vergleich der Verknüpfung bei $t_{Eingabe}$=1 und $t_{Eingabe}$=3

Es ergeben sich vier zu ändernde Signale: Für die Testvektoren 2, 4 und 5 ein Übergang von 1 nach 0 und für den Testvektor 6 ein Übergang von 0 nach 1. Diese Signaländerungen werden nun in den Zeitplan eingetragen.

In Tabelle 4-23 ist der Zeitplan nur zur Verdeutlichung zu zwei Simulationszeitpunkten dargestellt, um die vorzunehmenden Änderungen zu veranschaulichen. In Wirklichkeit besteht der Zeitplan nur aus einem eindimensionalen Feld.

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
vorher (Tabelle 4-20)	4	4	-	-	-	-
aktuell	4	**-**	-	**3+1=4**	**4**	**3+3=6**

Tabelle 4-23: Der Zeitplan bei $t_{Simulation}$=3

Zunächst wird der Testvektor 2 betrachtet: Man erkennt an der Tatsache, daß schon ein Wert (4) eingetragen ist, daß eine „alte“ Planung noch nicht in Kraft getreten ist. Die letzte Signaländerung wurde noch nicht vollständig durch das Gatter durchpropagiert. Es tritt somit ein Glitch auf und der entsprechende Impuls wird verschluckt: Der „alte“ Eintrag wird gelöscht und die „neue“ Planung ignoriert.

Bei den Testvektoren 4, 5 und 6 ist bisher keine Signaländerung geplant. Für die Testvektoren 4 und 5 wird eine Signaländerung zum Zeitpunkt $t_{Simulation}$+Verzögerung$_{1->0}$=4 und für Testvektor 6 eine Änderung zum Zeitpunkt $t_{Simulation}$+Verzögerung$_{0->1}$=6 eingetragen.

Anschließend werden wieder die Zeitvariablen aktualisiert: Der Zeitpunkt $t_{Ausgabe}$ bleibt nach wie vor 4. Der Zeitpunkt der nächsten Änderung der Eingabesignale $t_{Eingabe}$ beträgt nun ebenfalls 4 und somit wird auch die Simulationszeit auf $t_{Simulation}$=4 gesetzt.

Die Simulationszeit hat mittlerweile den Zeitpunk der nächsten Ausgabe erreicht. Die für diesen Zeitpunkt geplanten Änderungen werden jetzt durchgeführt und ein neuer Zeitwert dem Ausgabesignalverlauf hinzugefügt. Nach dem aktuellen Zeitplan (Tabelle 4-23) finden zum Zeitpunkt $t_{Simulation}$=4 Signaländerungen bei den Testvektoren 1, 4 und 5 statt:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t = $t_{Simulation}$=4	**1**	0	0	**0**	**0**	0

Tabelle 4-24: Der bisher ermittelte Ausgabesignalverlauf für $t_{Simulation}$=4

Die Änderungen werden durchgeführt und die entsprechenden Einträge aus dem Zeitplan gelöscht.

Nach der Aktualisierung der Gatterausgabe werden die momentan anliegenden Eingangssignalwerte verknüpft:

$t_{Simulation}=4$	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=4	0	1	1	1	0	0
2. Eingang	t=3	1	0	0	0	0	1
Ausgang		0	0	0	0	0	0

Tabelle 4-25: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}=4$

Erneut wird das Resultat mit dem Ergebnis der letzten Verknüpfung (Tabelle 4-21) verglichen:

Zeitpunkt	1	2	3	4	5	6
Verknüpfung bei $t_{Eingabe}=3$	1	0	0	0	0	1
Verknüpfung bei $t_{Eingabe}=4$	0	0	0	0	0	0

Tabelle 4-26: Vergleich der Verknüpfung bei $t_{Eingabe}=3$ und $t_{Eingabe}=4$

Für die Vektoren 1 und 6 ergibt sich eine Signaländerung von 1 nach 0. Diese Transitionen werden in den Zeitplan eingetragen:

Zeitplan	1	2	3	4	5	6
vorher	-	-	-	-	-	6
aktuell	**5**	-	-	-	-	-

Tabelle 4-27: Der Zeitplan bei $t_{Simulation}=4$

Bei Testvektor 1 ist noch kein Eintrag vorhanden. Es wird der Übergang für den Zeitpunkt $t=t_{Simulation}+Verzögerung_{1->0}=5$ geplant. Testvektor 6 enthält noch eine „alte“ Planung. Es handelt sich somit um einen Glitch und die für Zeitpunkt 6 geplante Signaländerung wird verschluckt.

Es folgt eine Aktualisierung des Zeitpunkts der nächsten Ausgabe zu $t_{Ausgabe}=5$. Die nächste Eingabe findet bei $t_{Eingabe}=8$ an Eingang 2 statt. Die Simulationszeit wird wieder auf den niedrigeren der beiden Werte gesetzt und beträgt somit $t_{Simulation}=5$.

Für den aktuellen Simulationszeitpunkt ist ein Eintrag im Zeitplan enthalten. Ein Signalwechsel ist für den ersten Testvektor vorgesehen. Die aktuelle Ausgabe wird entsprechend modifiziert und dem Ausgabesignalverlauf hinzugefügt:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t=4	1	0	0	0	0	0
$t = t_{Simulation}=5$	**0**	0	0	0	0	0

Tabelle 4-28: Der bisher ermittelte Ausgabesignalverlauf für $t_{Simulation}=5$

Der Eintrag wird aus dem Zeitplan gelöscht. Dieser enthält nun keine Einträge mehr. Somit ist $t_{Ausgabe}$ undefiniert. Die nächste Eingabe findet bei $t_{Eingabe}=8$ statt. Die Simulationszeit wird auf $t_{Simulation}=8$ gesetzt und die zu diesem Zeitpunkt anliegenden Eingangssignale werden verknüpft:

$t_{Simulation}=8$	**Zeitpunkt**	**1**	**2**	**3**	**4**	**5**	**6**
1. Eingang	t=4	0	1	1	1	0	0
2. Eingang	t=8	1	0	1	1	0	1
Ausgang		0	0	1	1	0	0

Tabelle 4-29: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}=8$

Wieder wird das Resultat mit dem Ergebnis der letzten Verknüpfung (Tabelle 4-25) verglichen:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
Verknüpfung bei $t_{Eingabe}=4$	0	0	0	0	0	0
Verknüpfung bei $t_{Eingabe}=8$	0	0	1	1	0	0

Tabelle 4-30: Vergleich der Verknüpfung bei $t_{Eingabe}=4$ und $t_{Eingabe}=8$

Es ergeben sich zwei Transitionen von 0 nach 1 in den Vektoren 3 und 4. Diese werden für den Zeitpunkt $t_{Simulation}$+Verzögerung$_{0->1}=11$ geplant und im Zeitplan eingetragen (siehe Tabelle 4-31). Damit erfolgt die nächste Ausgabe zum Zeitpunkt $t_{Ausgabe}=11$.

Die Zeitvariable $t_{Eingabe}$ wird im folgenden ignoriert, da keine weiteren Signaländerungen an den Eingängen des Gatters anstehen. Somit schreitet die Simulationszeit direkt zu $t_{Simulation}=t_{Ausgabe}=11$ fort. Alle für diesen Zeitpunkt geplanten Signaländerungen werden durchgeführt: In diesem Fall ändern sich die Signalwerte bei den Testvektoren 3 und 4.

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
aktuell	-	-	**11**	**11**	-	-

Tabelle 4-31: Der Zeitplan bei $t_{Simulation}$=8

Die Signaländerungen werden durchgeführt und die entsprechenden Einträge aus dem Zeitplan gelöscht. Da keine weiteren Ausgaben geplant sind erhält man den endgültigen Ausgabesignalverlauf:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t=4	1	0	0	0	0	0
t=5	0	0	0	0	0	0
t = $t_{Simulation}$=11	0	0	**1**	**1**	0	0

Tabelle 4-32: Der endgültige Ausgabesignalverlauf für $t_{Simulation}$=11

Jetzt müssen noch die Anzahl der aufgetretenen Transitionen und Hazards ermittelt werden. Dies geschieht auf die exakt gleiche Weise wie beim Transport-Delay-Modell (siehe Kapitel 4.3.2):

Ausgabesignalverlauf							Exklusiv-Oder-Verknüpfung					
t<0	0	0	0	1	1	0						
							1	0	0	1	1	0
t=4	1	0	0	0	0	0						
							1	0	0	0	0	0
t=5	0	0	0	0	0	0						
							0	0	1	1	0	0
t=11	0	0	1	1	0	0						
Summe (Transitionen)							2	0	1	2	1	0
Hazards							2	0	0	2	0	0

Tabelle 4-33: Berechnung der Transitionen und Hazards

Die Anzahl an Transitionen und Hazards pro Testvektor werden nun mit der dem Testvektor entsprechenden Gewichtung multipliziert und die Summe über alle Produkte gebildet.

Anzahl pro Testvektor	Transitionen	2	0	1	2	1	0	Gesamt-summe
	Hazards	2	0	0	2	0	0	
	Gewichtung	1	3	1	1	3	1	
Gewichtete Anzahl	Transitionen	2	0	1	2	3	0	8
	Hazards	2	0	0	2	0	0	4

Tabelle 4-34: Gesamtsumme der Transitionen und Hazards

Die resultierenden Werte entsprechen der Transitions- beziehungsweise Hazardanzahl, die während der gesamten Simulation an diesem Gatter aufgetreten sind.

4.4.3 Das Real-Delay-Modell im Pseudocode

```
aktuelle_Werte = logische Verknüpfung der Eingangssignale bei t<0;
aktuelle_Ausgabe = aktuelle_Werte;
Erstellung des Ausgabesignalverlaufs mit aktuelle_Werte als ersten Eintrag für t<0;
t_Eingabe = Zeitpunkt der ersten Signaländerung an den Gattereingängen;
REPEAT {
    t_Simulation = (t_Eingabe < t_Ausgabe) ? t_Eingabe : t_Ausgabe;
    IF (t_Simulation == t_Ausgabe ) {
        FOR (jedes Feld x im Zeitplan, dessen Eintrag t_Simulation gleicht) DO {
            Setze Feld x in aktuelle_Ausgabe auf seinen entgegengesetzten Wert;
            Entferne den Eintrag in Feld x des Zeitplans; };
        Erweitere den Ausgabesignalverlauf um aktuelle_Ausgabe für t=t_Simulation;};
    letzte_Werte = aktuelle_Werte;
    aktuelle_Werte = logische Verknüpfung der aktuell zu t_Simulation gültigen Eingangssignale;
    Vergleiche aktuelle_Werte mit letzte_Werte;
    FOR (jede Transition in Testvektor x zwischen aktuelle_Werte und letzte_Werte) DO {
        IF (das Feld x im Zeitplan enthält keinen Eintrag) {
            Trage t_Simulation + (Verzögerung_0->1 oder Verzögerung_1->0) in Feld x ein; }
        ELSE {
```

Listing 4-3: Pseudocode für Real-Delay-Modell

```
            Entferne Eintrag in Feld x des Zeitplans; }; };// Glitch
    t_Eingabe = Zeitpunkt der nächsten Signaländerung an den Gattereingängen;
    t_Ausgabe = Zeitpunkt der nächsten Signaländerung am Gatterausgang;
} UNTIL (t_Eingabe und t_Ausgabe verweisen auf keinen Zeitpunkt mehr);
```

Listing 4-3: Pseudocode für Real-Delay-Modell

4.5 Die Glitch-Simulation

4.5.1 Das Glitch-Modell

Das dieser Simulationsart zugrunde liegende Modell stammt aus [WIL 98] und wurde für die Implementierung im Rahmen der entwickelten Erweiterung geringfügig angepaßt. In [WIL 98] werden verschiedene Annäherungen für den Energieverbrauch eines Gatters im Falle eines Glitches oder einer Transition vorgestellt. Dabei hängt die Energie im Falle eines Glitches von der Impulsbreite δ ab (siehe Abbildung 4-2). Das Modell der zwei Geraden stellt sich als die geeignetste Methode heraus und wurde in diese Arbeit integriert.

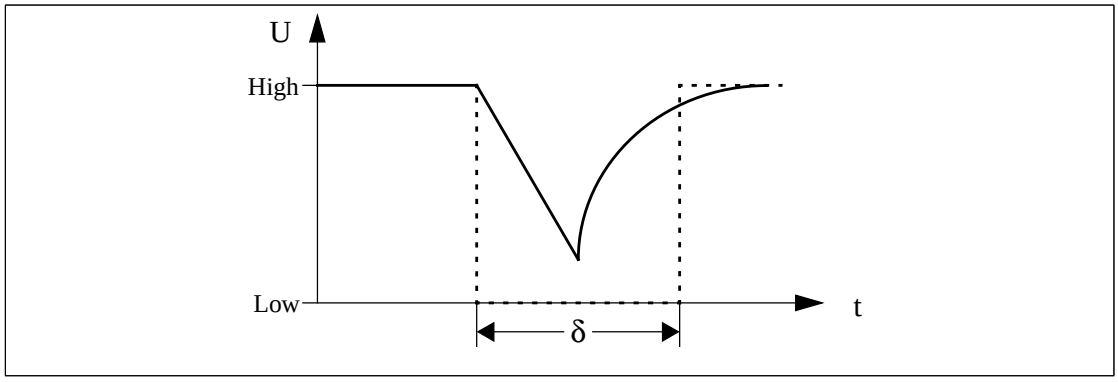


Abbildung 4-2: Impulsbreite δ eines Glitches

Im *Modell der zwei Geraden* wird der Energieverbrauch in Abhängigkeit von der Impulsbreite durch zwei Geraden angenähert:

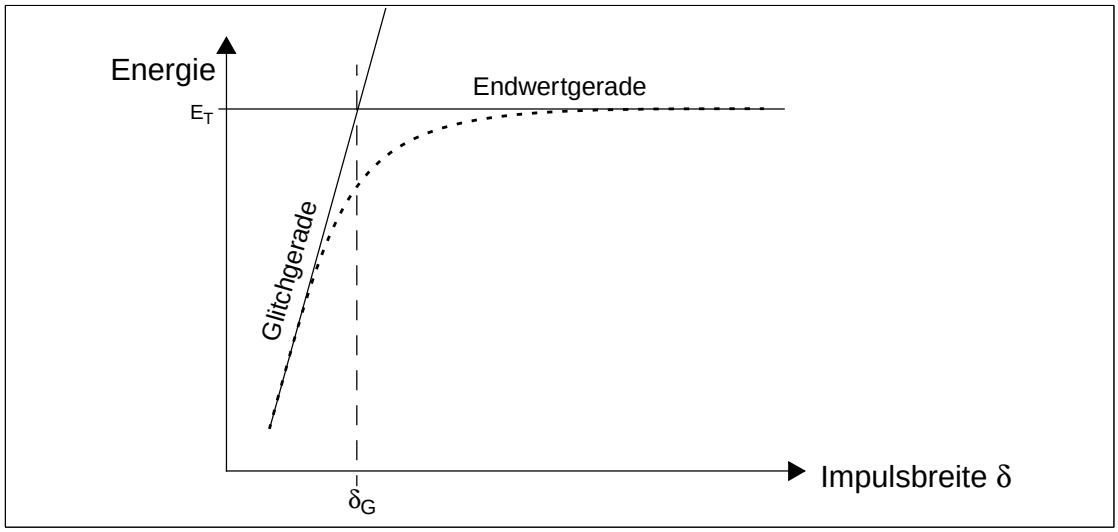


Abbildung 4-3: Annäherung des Energieverbrauchs durch zwei Geraden

In Abbildung 4-3 symbolisiert die gepunktete Linie den realen Energieverbrauch. Dieser wird angenähert durch eine Glitchgerade und eine Endwertgerade.

Die beiden Geraden hängen von folgenden Parametern ab:

- dem Gattertyp,
- der Anzahl der Eingänge des Gatters,
- dem Eingang des Gatters, an dem der Impuls auftritt,
- den parasitären Kapazitäten der Folgegatter
- und der Impulsrichtung, d.h. ob es sich um einen 0-1-0- oder einen 1-0-1-Impuls handelt.

Wie man Abbildung 4-3 entnehmen kann, hängt allein die Glitchgerade von der Impulsbreite δ ab. Die Endwertgerade steht für den konstanten Energiewert, der bei Umladung aller Sperrschichtkapazitäten und der Lastkapazitäten der Folgegatter aufgewendet werden muß. δ_G ist als Schnittpunkt der beiden Geraden definiert und wird als *Glitchpunkt* bezeichnet. Liegt die Breite δ eines Impulses unter δ_G, so liegt ein Glitch vor. Bei $\delta>\delta_G$ findet eine vollständige Transition statt. In diesem Fall repräsentiert δ_G die Gatterlaufzeit.

Die entsprechenden Annäherungsgeraden wurden in [WIL 98] nur für die Gattertypen NOT, NAND und NOR ermittelt. So werden in dieser Implementierung auch allein diese Gattertypen unterstützt.

Das in Kapitel 4.4 vorgestellte Real-Delay-Modell untersucht Gatter auf eventuelle Glitches an deren Ausgang. Die Betrachtungen in [WIL 98] dagegen beziehen sich auf die Gattereingänge. Zur Integration wurden daher die Geradenwerte der unterschiedlichen Eingänge aus [WIL 98] gemittelt und alle Untersuchungen auf Glitches mit den resultierenden Durchschnittsgeraden am Ausgang der Gatter vollzogen.

4.5.2 Die Realisierung des Glitch-Modells

Die Glitchbereiche der in [WIL 98] angegebenen Gleichungen liegen alle im Bereich von Pikosekunden. Daher werden die Werte der Zeitpunkte in diesem Modell ebenfalls als Pikosekunden interpretiert. Für die Vorführung des Glitch-Modells werden wieder die Eingangssignale s_{A1} und s_{A2} aus Tabelle 4-2 verwendet. Zur Anpassung der Größenverhältnisse wurden allerdings die Zeitpunkte mit 100 multipliziert (siehe Tabelle 4-35).

Zusätzlich werden die Geradengleichungen für das UND-Gatter des Beispiels benötigt. Die in Tabelle 4-36 aufgeführten Gleichungen sind rein fiktiv und für das Beispiel dahingehend modifiziert, daß sich dieselben Verzögerungseigenschaften wie beim Real-Delay-Modell ergeben. Es stehen δ für die Impulsbreite, ZB für die Zeitbasis (hier Pikosekunden) und C für die Lastkapazität. Die Lastkapazität wird aus der Anzahl der Folgegatter berechnet, wobei jedem Gatter standardmäßig eine Eingangskapazität von 10 Femtofarad zugeordnet wird (siehe Anhang A.1). In diesem Beispiel treibe das UND-Gatter zwei Folgegatter wodurch sich eine Lastkapazität von 20 Femtofarad ergibt.

Signal	Zeitpunkt	1	2	3	4	5	6
1. Eingang: s_{A1}	t<0	0	0	1	1	1	0
	t=100	1	1	0	1	1	1
	t=400	0	1	1	1	0	0
2. Eingang: s_{A2}	t<0	1	1	0	1	1	0
	t=300	1	0	0	0	0	1
	t=800	1	0	1	1	0	1
Gewichtung		1	3	1	1	3	1

Tabelle 4-35: Beispiel für Signalverlauf interner Gatter mit an das Glitch-Modell angepaßten Zeiten

Geradentyp	Funktion	Übergang
Glitchgeraden	$E_{G0} = (5,43 \cdot 10^{-3} \cdot \delta \cdot ZB) + 22 \cdot 10^{-15}$	1-0-1
	$E_{G1} = 1,883 \cdot 10^{-3} \cdot \delta \cdot ZB$	0-1-0
Endwertgerade	$E_T = 1,25 \cdot C + 5,4 \cdot 10^{-13} = 5,65 \cdot 10^{-13}$	bei C=20 fF

Tabelle 4-36: Beispiel für Glitch- und Endwertgeraden

Zu Beginn der Simulation werden die Glitchpunkte gg_0 für den 1-0-1-Übergang und gg_1 für den 0-1-0-Übergang berechnet. Diese entsprechen jeweils dem Schnittpunkt der Glitchgeraden und der Endwertgeraden: E_{G0} und E_{G1} werden mit E_T gleichgesetzt und es ergeben sich gg_0=100 und gg_1=300.

Als erstes werden in der Simulation die Anfangswerte aller Eingangssignale dem Komponententyp entsprechend verknüpft und als Anfangswert des Ausgabesignalverlaufs gespeichert (siehe Tabelle 4-8):

Zeitpunkt	1	2	3	4	5	6
t<0	0	0	0	1	1	0

Tabelle 4-37: Aktuelle Ausgabe für t<0

Die weitere Vorgehensweise ähnelt zum großen Teil dem im letzten Kapitel vorgestellten Real-Delay-Modell. Es wird zunächst der früheste Zeitpunkt $t_{Eingabe}$ ermittelt zu dem das erste Ereignis an den Eingangssignalen stattfindet. In diesem Fall ist $t_{Eingabe}$=100. Die Simulationszeit $t_{Simulation}$ wird auf diesen Zeitpunkt gesetzt und die entsprechenden Eingangssignale werden verknüpft (siehe Tabelle 4-38).

$t_{Simulation}$=100	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=100	1	1	0	1	1	1
2. Eingang	t<0	1	1	0	1	1	0
Ausgang		1	1	0	1	1	0

Tabelle 4-38: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=100

Das Resultat wird mit dem Anfangswert des Ausgabesignalverlaufs verglichen und auftretende Transitionen im Zeitplan eingetragen (siehe Tabelle 4-39 und Tabelle 4-40).

Zeitpunkt	1	2	3	4	5	6
Anfangswert	0	0	0	1	1	0
Verknüpfung bei $t_{Simulation}$=100	1	1	0	1	1	0

Tabelle 4-39: Vergleich des Anfangswerts und der Verknüpfung bei $t_{Simulation}$=100

Für die Vektoren 1 und 2 tritt ein Übergang von 0 nach 1 auf, der zum Zeitpunkt $t_{Simulation}+gg_1$=400 durch das Gatter durchpropagiert sein wird.

Zeitplan	1	2	3	4	5	6
$t_{Simulation}$=100	**400**	**400**	-	-	-	-

Tabelle 4-40: Der Zeitplan für $t_{Simulation}$=100

Jetzt werden die Zeitvariablen aktualisiert: $t_{Eingabe}$ beträgt nun 300 und $t_{Ausgabe}$=400. Die Simulationszeit $t_{Simulation}$ wird auf den kleineren der beiden Werte gesetzt und beträgt somit 300. Wiederum werden die zu diesem Zeitpunkt gültigen Eingangssignale verknüpft.

$t_{Simulation}$=300	Zeitpunkt	1	2	3	4	5	6
1. Eingang	t=100	1	1	0	1	1	1
2. Eingang	t=300	1	0	0	0	0	1
Ausgang		1	0	0	0	0	1

Tabelle 4-41: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=300

Es folgt die Gegenüberstellung mit dem Ergebnis der vorhergehenden Verknüpfung aus Tabelle 4-38:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
Verknüpfung bei $t_{Eingabe}$=100	1	1	0	1	1	0
Verknüpfung bei $t_{Eingabe}$=300	1	0	0	0	0	1

Tabelle 4-42: Vergleich der Verknüpfung bei $t_{Eingabe}$=100 und $t_{Eingabe}$=300

Der Vergleich ergibt vier Signalwechsel: Ein Übergang von 1 nach 0 für die Vektoren 2, 4 und 5 und ein Übergang von 0 nach 1 für den Testvektor 6. Im Zeitplan (Tabelle 4-40) ist allerdings für den zweiten Testvektor bereits ein Eintrag enthalten. Wie beim Real-Delay-Model wird dieser Eintrag aus dem Zeitplan entfernt und die neue Planung ignoriert. Zusätzlich wird der Energieverbrauch berechnet, der bei diesem Glitch auftritt. Für die Berechnung muß zunächst die Impulsbreite δ des Glitches ermittelt werden.

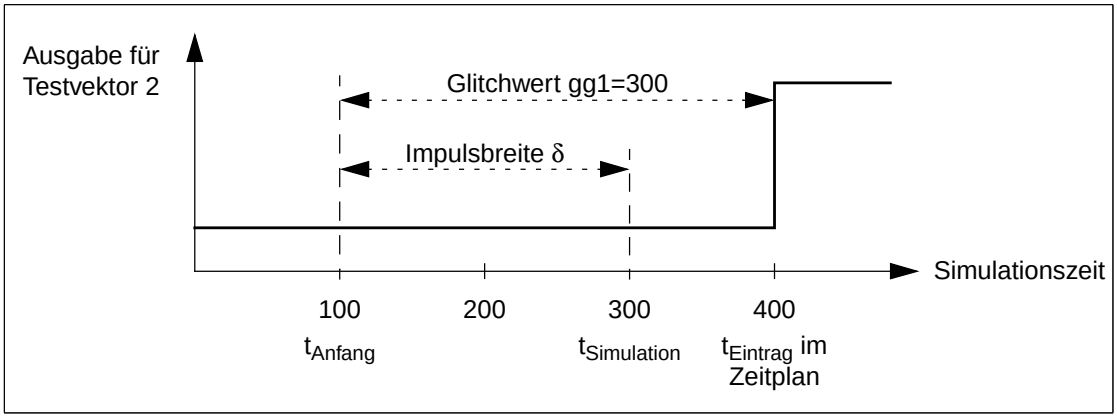


Abbildung 4-4: Berechnung der Glichtimpulsbreite bei Testvektor 2

Die Impulsbreite δ entspricht $t_{Simulation}$-t_{Anfang}. Der Anfangszeitpunkt des Glitchimpulses t_{Anfang} ergibt sich aus dem Eintrag im Zeitplan abzüglich des Glitchwerts gg_1. Der Glitch beginnt also zum Zeitpunkt $t_{Anfang}=t_{Eintrag}-gg_1$=100. Daraus resultiert eine Impulsbreite von δ=300-100=200. Eingesetzt in die Glitchgerade E_{G1} aus Tabelle 4-36 ergibt sich ein Energieaufwand von $3{,}766*10^{-13}$ Joule.

Die übrigen Signalwechsel werden wie gehabt in den Zeitplan eingetragen:

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
aktuell	400	-	-	**400**	**400**	**600**

Tabelle 4-43: Der Zeitplan bei $t_{Simulation}$=300

Es folgt die Aktualisierung der Zeitvariablen: $t_{Ausgabe}$ bleibt 400 und $t_{Eingabe}$ beträgt nun ebenfalls 400. Somit wird auch die Simulationszeit $t_{Simulation}$ auf 400 gesetzt. Für diesen Zeitpunkt sind drei Signalwechsel im Zeitplan vorgesehen.Die aktuelle Ausgabe wird entsprechend modifiziert und dem Ausgabesignalverlauf angehängt:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t = $t_{Simulation}$=400	**1**	0	0	**0**	**0**	0

Tabelle 4-44: Der bisher ermittelte Ausgabesignalverlauf für $t_{Simulation}$=400

Wie gehabt werden jetzt die zu diesem Zeitpunkt anliegenden Eingangssignalwerte verknüpft:

$t_{Simulation}$**=400**	**Zeitpunkt**	**1**	**2**	**3**	**4**	**5**	**6**
1. Eingang	t=400	0	1	1	1	0	0
2. Eingang	t=300	1	0	0	0	0	1
Ausgang		0	0	0	0	0	0

Tabelle 4-45: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=400

Erneut wird das Resultat mit dem Ergebnis der letzten Verknüpfung (Tabelle 4-41) verglichen (siehe Tabelle 4-46). Für die Vektoren 1 und 6 ergibt sich eine Signaländerung von 1

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
Verknüpfung bei $t_{Eingabe}$=300	1	0	0	0	0	1
Verknüpfung bei $t_{Eingabe}$=400	0	0	0	0	0	0

Tabelle 4-46: Vergleich der Verknüpfung bei $t_{Eingabe}$=300 und $t_{Eingabe}$=400

nach 0. Diese Änderungen werden in den Zeitplan eingetragen. Für den ersten Testvektor ist noch kein Eintrag vorhanden. Der entsprechende Übergang wird für $t=t_{Simulation}+gg_0=500$ geplant.

Für Testvektor 6 ist bereits eine Planung (für t=600) enthalten und der Impuls stellt sich als Glitch heraus. Die Impulsbreite δ beträgt $t_{Simulation}$-t_{Anfang}=400-(600-300)=100 Zeiteinheiten. Eingesetzt in E_{G1} ergibt dies 1,833*10^{-13} Joule. E_{Summe} ist nun 1,833*10^{-13}+3,766*10^{-13}=5,599*10^{-13} J. Anschließend wird der Eintrag aus dem Zeitplan gelöscht.

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
vorher	-	-	-	-	-	600
aktuell	**500**	-	-	-	-	-

Tabelle 4-47: Der Zeitplan bei $t_{Simulation}$=400

Die Zeitvariablen werden aktualisiert: $t_{Ausgabe}$ beträgt nun 500 und $t_{Eingabe}$=800. Der im Zeitplan vorgesehene Signalwechsel für den ersten Testvektor wird durchgeführt, der entsprechende Eintrag aus dem Zeitplan gelöscht und der Ausgabesignalverlauf um die neue Ausgabe ergänzt:

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t=400	1	0	0	0	0	0
t = $t_{Simulation}$=500	**0**	0	0	0	0	0

Tabelle 4-48: Der bisher ermittelte Ausgabesignalverlauf für $t_{Simulation}$=500

$t_{Ausgabe}$ ist nun undefiniert, da der Zeitplan keine Einträge mehr enthält. Somit schreitet die Simulationszeit $t_{Simulation}$ direkt zu $t_{Eingabe}$=800 fort.

Eine Verknüpfung der zum aktuellen Zeitpunkt gültigen Eingangssignale ergibt:

$t_{Simulation}$=800	**Zeitpunkt**	**1**	**2**	**3**	**4**	**5**	**6**
1. Eingang	t=400	0	1	1	1	0	0
2. Eingang	t=800	1	0	1	1	0	1
Ausgang		0	0	1	1	0	0

Tabelle 4-49: Ermittlung des Ausgabesignalverlauf für $t_{Simulation}$=800

Wieder wird das Resultat mit dem Ergebnis der letzten Verknüpfung verglichen (siehe Tabelle 4-50).

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
Verknüpfung bei $t_{Eingabe}$=400	0	0	0	0	0	0
Verknüpfung bei $t_{Eingabe}$=800	0	0	1	1	0	0

Tabelle 4-50: Vergleich der Verknüpfung bei $t_{Eingabe}$=400 und $t_{Eingabe}$=800

Es ergeben sich zwei Transitionen von 0 nach 1 in den Vektoren 3 und 4. Diese werden für den Zeitpunkt $t_{Simulation}+gg_1$=1100 vorgesehen und im Zeitplan eingetragen (siehe Tabelle 4-51). Damit ergibt sich die nächste Ausgabe zum Zeitpunkt $t_{Ausgabe}$=1100.

Zeitplan	**1**	**2**	**3**	**4**	**5**	**6**
aktuell	-	-	**1100**	**1100**	-	-

Tabelle 4-51: Der Zeitplan bei $t_{Simulation}$=800

Ein letztes Mal wird die Simulationszeit erhöht zu $t_{Simulation}=t_{Ausgabe}$=1100 und die entsprechenden Transitionen ausgeführt.

Zeitpunkt	**1**	**2**	**3**	**4**	**5**	**6**
t<0	0	0	0	1	1	0
t=400	1	0	0	0	0	0
t=500	0	0	0	0	0	0
$t = t_{Simulation}$=1100	**0**	0	**1**	**1**	0	0

Tabelle 4-52: Der endgültige Ausgabesignalverlauf

Die Berechnung der Transitionen und Hazards erfolgt identisch zu der Vorgehensweise beim Real-Delay-Modell (siehe Kapitel 4.4.2). Man erhält Tabelle 4-53.

Abschließend wird die Summe der aufgewendeten Energie um die für Transitionen verbrauchte Energie erhöht: $E_{Summe} = E_{Summe} + Anzahl_{Transitionen}*E_T$. Acht Transitionen sind aufgetreten (siehe Tabelle 4-34) und E_{Summe} beträgt insgesamt $5{,}08*10^{-12}$ J.

Ausgabesignalverlauf							Exklusiv-Oder-Verknüpfung					
t<0	0	0	0	1	1	0						
							1	0	0	1	1	0
t=400	1	0	0	0	0	0						
							1	0	0	0	0	0
t=500	0	0	0	0	0	0						
							0	0	1	1	0	0
t=1100	0	0	1	1	0	0						
Summe (Transitionen)							2	0	1	2	1	0
Hazards							2	0	0	2	0	0

Tabelle 4-53: Berechnung der Transitionen und Hazards

Anzahl pro Testvektor	Transitionen	2	0	1	2	1	0	Gesamt-summe
	Hazards	2	0	0	2	0	0	
	Gewichtung	1	3	1	1	3	1	
Gewichtete Anzahl	Transitionen	2	0	1	2	3	0	8
	Hazards	2	0	0	2	0	0	4

Tabelle 4-54: Gesamtsumme der Transitionen und Hazards

4.5.3 Das Glitch-Modell im Pseudocode

```
aktuelle_Werte = logische Verknüpfung der Eingangssignale bei t<0;
aktuelle_Ausgabe = aktuelle_Werte;
Energie = 0;
Erstelle Ausgabesignalverlauf mit aktuelle_Werte als ersten Eintrag für t<0;
t_Eingabe = Zeitpunkt der ersten Signaländerung an den Gattereingängen;
REPEAT {
    t_Simulation = (t_Eingabe < t_Ausgabe) ? t_Eingabe : t_Ausgabe;
    IF (t_Simulation == t_Ausgabe ) {
        FOR (jedes Feld x im Zeitplan, dessen Eintrag t_Simulation gleicht) DO {
```

Listing 4-4: Pseudocode für das Glitch-Modell

```
            Setze Feld x in aktuelle_Ausgabe auf seinen entgegengesetzten Wert;
            Entferne den Eintrag in Feld x des Zeitplans; };
        Erweitere den Ausgabesignalverlauf um aktuelle_Ausgabe für t=t_Simulation;};
    letzte_Werte=aktuelle_Werte;
    aktuelle_Werte=logische Verknüpfung der aktuell zu t_Simulation gültigen Eingangssignale;
    Vergleiche aktuelle_Werte mit letzte_Werte;
    FOR (jede Transition in Testvektor x zwischen aktuelle_Werte und letzte_Werte) DO {
        IF (das Feld x im Zeitplan enthält keinen Eintrag) {
            Trage t_Simulation + (Glitchpunkt_0->1 oder Glitchpunkt_1->0) in Feld x ein; }
        ELSE {
            δ_Glitch = t_Simulation - [Wert in Feld x - (Glitchpunkt_0->1 oder Glitchpunkt_1->0)];
            Energie += [Glitchgerade_1(δ_Glitch) oder Glitchgerade_0(δ_Glitch)];
            Entferne Eintrag in Feld x des Zeitplans; }; };
    t_Eingabe = Zeitpunkt der nächsten Signaländerung an den Gattereingängen;
    t_Ausgabe = Zeitpunkt der nächsten Signaländerung am Gatterausgang;
} UNTIL (t_Eingabe und t_Ausgabe verweisen auf keinen Zeitpunkt mehr);
Energie += Energie_Transition * Anzahl_Transitionen;
```

Listing 4-4: Pseudocode für das Glitch-Modell

Leistungsbewertung

Kapitel 5

In diesem Kapitel werden die Geschwindigkeit und die Genauigkeit der Simulationsarten der Logikebene diskutiert. Für eine Bewertung der Glitch-Simulation sei auf [WIL 98] verwiesen. Alle Simulationen wurden auf einer Sparc Ultra 1 mit 170 MHz und 192 MB Hauptspeicher unter Solaris 2.5 vollzogen. Für die Signalverarbeitung während der Simulation wurde die auf [SCH 95] basierende Methode gewählt.

5.1 Die Benchmark-Schaltkreise

Um nachvollziehbare Ergebnisse zu erhalten, wurden alle Testsimulationen an Schaltungen aus der ISCAS-85- und der ISCAS-89-Benchmark-Bibliothek vollzogen. Jeder Schaltung dienten 10.000 zufällig generierte Testvektoren als Simulationsstimuli. Die verwendeten Schaltkreise haben folgende Charakteristika:

Schaltkreisname	Anzahl der Primäreingänge	Anzahl der Gatter
c432	37	159
c499	42	201
c880	61	382
c1908	34	879
c3540	51	1668
c6288	33	2415

Tabelle 5-1: Kombinatorische Schaltnetze

Schaltkreisname	Anzahl der Primäreingänge	Anzahl der Gatter
s386	8	168
s820	19	299
s1196	15	552
s1494	9	656
s5378	36	2961
s9234	20	5830

Tabelle 5-2: Sequentielle Schaltwerke

5.2 Die Genauigkeit der Simulationsarten

Bei den Betrachtungen in diesem Kapitel wird die Genauigkeit der Simulationstypen an der Real-Delay-Simulation gemessen. Nur die Real-Delay-Simulation und die Glitch-Simulation ermitteln die Anzahl der auftretenden Transitionen exakt.

5.2.1 Die Genauigkeit der Zero-Delay-Simulation

Die Zero-Delay-Simulation ist die schnellste Simulationsart auf der logischen Ebene. Da bei dieser Methode die Gatterlaufzeiten nicht berücksichtigt werden, kann man allerdings Hazards nicht erfassen. Das prozentuale Fehlerausmaß ist in Abbildung 5-1 aufgeführt.

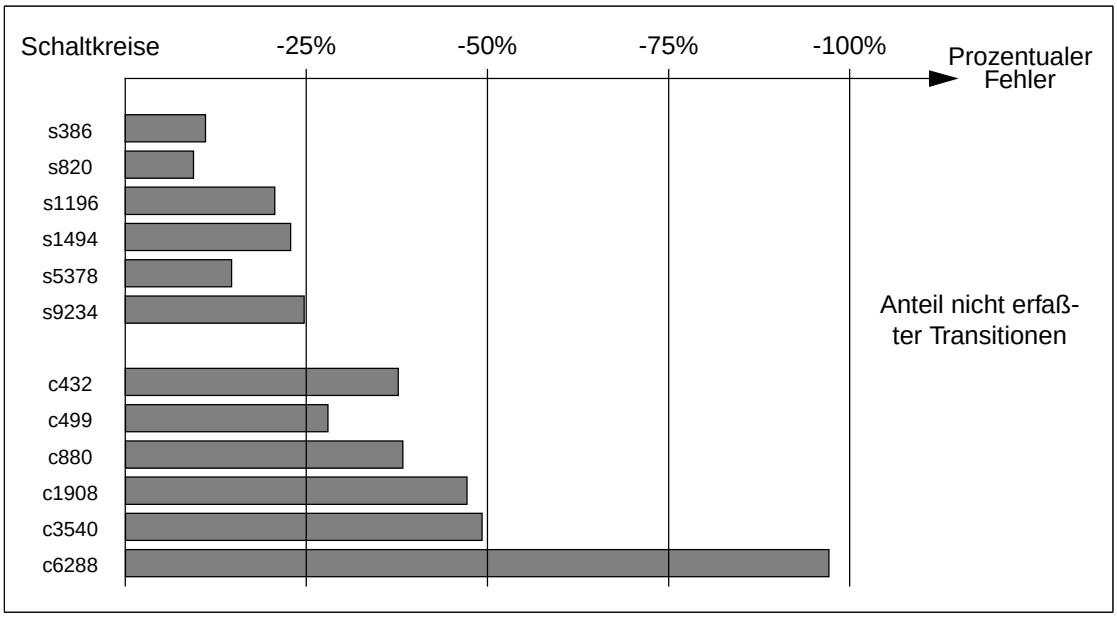


Abbildung 5-1: Relativer Fehler der Zero-Delay-Simulation

Ein Balken gibt in dieser Darstellung an, wieviel Prozent an Transitionen zu wenig erfaßt wurden. Durchschnittlich beträgt die Fehlerquote etwa -33%. Es werden also nur zwei drittel der auftretenden Transitionen erfaßt.

An Abbildung 5-1 erkennt man, daß die Vernachlässigung von Gatterlaufzeiten teilweise zu sehr großen Fehlern führt. Die Balken entsprechen den Hazard-Anteilen, die sich im Lauf einer Real-Delay-Simulation ergeben. Manche Schaltkreise, wie Multiplizierer und Addierer, haben im allgemeinen einen sehr hohen Hazard-Anteil an Transitionen. Beim Schaltkreis c6288, einem Multiplizierer, ermittelt beispielsweise die Zero-Delay-Simulation nur 9.269.106 Transitionen. In Wirklichkeit treten allerdings 317.774.630 Transitionen auf, also die 34-fache Menge!

5.2.2 Die Genauigkeit der Transport-Delay-Simulation

Bei der Transport-Delay-Simulation werden Gatterlaufzeiten berücksichtigt. Allerdings gilt die Einschränkung, daß ein Gatter keine unterschiedliche Laufzeit für einen 0-1- oder einen 1-0-Übergang haben kann. Zusätzlich wird der größte Fehler dadurch verursacht, daß Glitches nicht erkannt und die entsprechenden Transitionen nicht verschluckt werden. Daher werden bei der Transport-Delay-Simulation grundsätzlich zuviele Transitionen ermittelt. Ein Balken in Abbildung 5-2 gibt an, wieviel Prozent an Transitionen zuviel ermittelt wurden. Durchschnittlich beträgt die Fehlerquote +32%. Es werden also ein drittel zu viel Transitionen erfaßt.

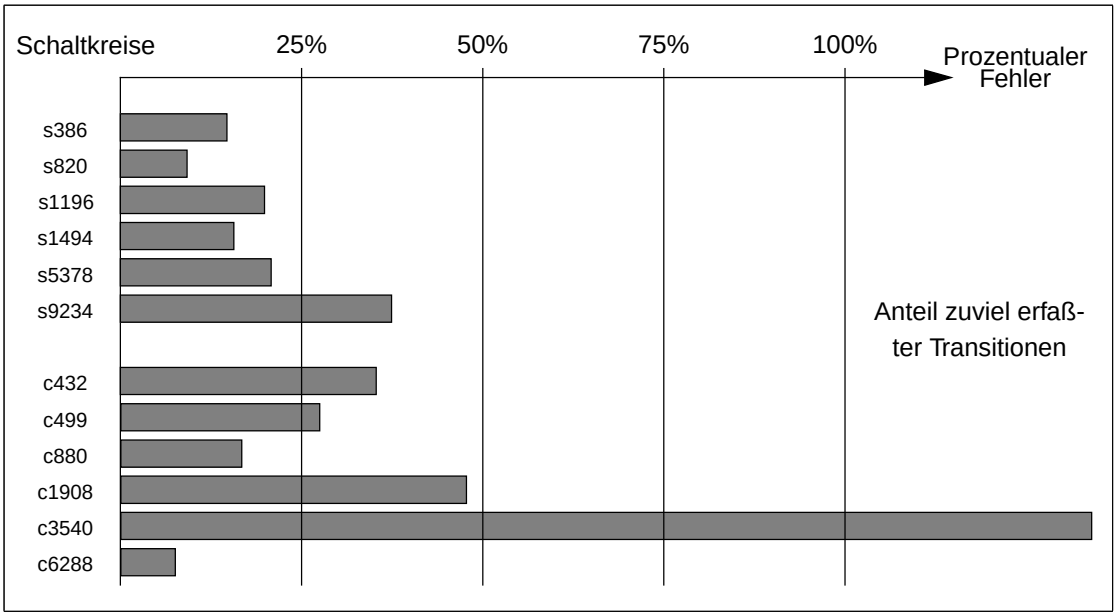


Abbildung 5-2: Relativer Fehler der Transport-Delay-Simulation

5.3 Die Laufzeiten der Simulationsarten

5.3.1 Vergleich mit dem bestehenden Simulator von [DD 98]

Der bestehende Simulator von [DD 98] war bisher auf ein Zero-Delay-Modell beschränkt. Dessen Laufzeit wird nun mit der im Zuge der Erweiterung entwickelten Variante (siehe Kapitel 4.2) verglichen. In Abbildung 5-3 werden die Geschwindigkeitsverhältnisse zwischen den beiden Varianten aufgezeigt. Bei der “0%”-Markierung sind beide Simulationen gleich schnell. Abweichungen davon geben die prozentualen Laufzeitunterschiede an. Es zeigt sich, daß die neu entwickelte Variante im Durchschnitt nur etwa 6% langsamer ist. Diese Differenz ist der Preis für den leichter wartbaren, konsequent objektorientierten Aufbau der neu entwickelten Variante.

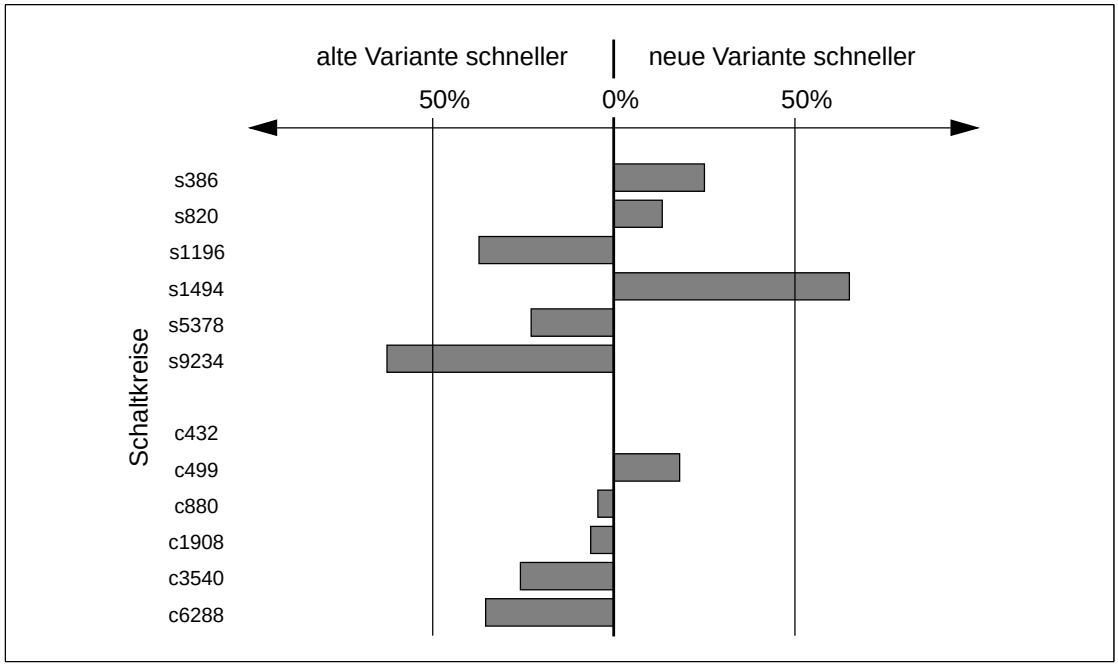


Abbildung 5-3: Laufzeitverhältnis der Zero-Delay-Simulation gegenüber der Variante in [DD 98]

5.3.2 Vergleich mit dem kommerziellen Simulator Synopsys

Für einen Laufzeitvergleich mit dem kommerziellen Logiksimulator Synopsys (Version 9708) wurde das Real-Delay-Modell herangezogen, da dieses Modell die komplexeste Simulationsart auf der logischen Ebene darstellt. Im Vergleich zu Synopsys ist die in dieser Arbeit implementierte Variante durchschnittlich 9,3-mal so schnell, wobei selbstverständlich beide das gleiche Ergebnis liefern. Allerdings gibt die Real-Delay-Simulation zusätzlich die Anzahl an aufgetretenen Hazards aus. In Abbildung 5-4 sind exemplarisch die Laufzeiten während der Simulation der sequentiellen Netze dargestellt:

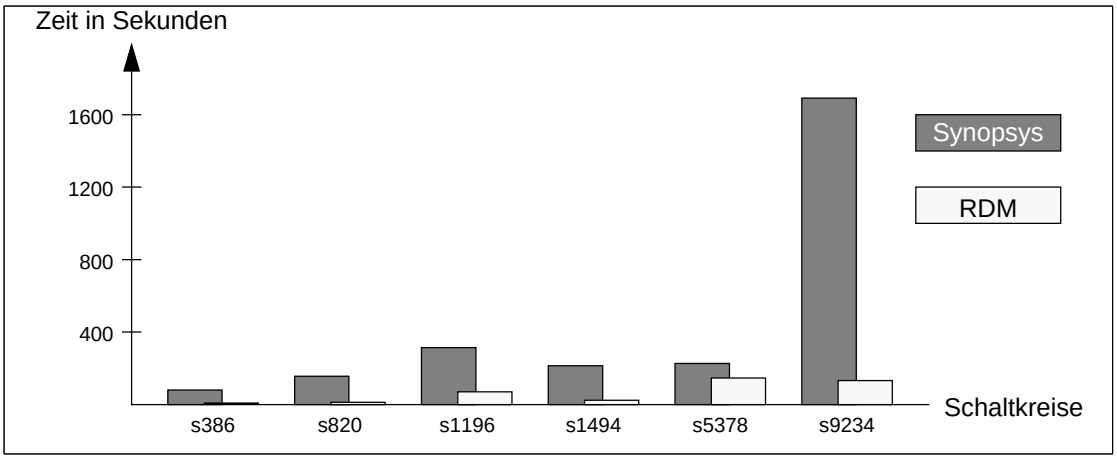


Abbildung 5-4: Laufzeitvergleich sequentieller Netzwerke zwischen Synopsys und dem RDM

Abbildung 5-5 stellt die Geschwindigkeitsverhältnisse zwischen Synopsys und der Real-Delay-Simulation dar. Dabei repräsentiert die horizontale Achse den Faktor, um welchen die Real-Delay-Simulation schneller ist als Synopsys.

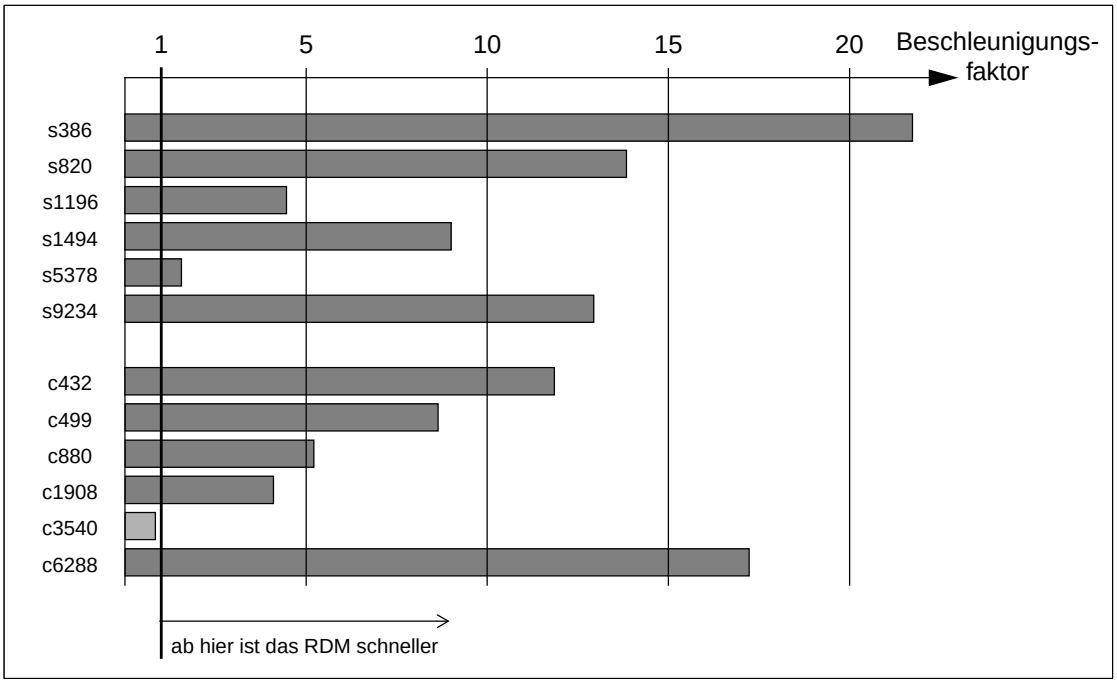


Abbildung 5-5: Geschwindigkeitsverhältnis zwischen Synopsys und RDM

Man erkennt, daß die Real-Delay-Simulation in der Regel um ein Vielfaches schneller ist als Synopsys. Durchschnittlich ergibt sich eine mehr als 9-fache Beschleunigung der Laufzeit. Nur in einem einzigen Fall (beim Schaltkreis c3540) braucht die Real-Delay-Simulation aus bisher ungeklärten Gründen ein wenig länger als Synopsys.

5.3.3 Vergleich der Logiksimulationsarten untereinander

Tabelle 5-3 zeigt noch einmal alle Logiksimulationsarten im Vergleich. Man erkennt, daß die höhere Laufzeit der Transport-Delay-Simulation gegenüber der Zero-Delay-Simulation nicht durch eine höhere Genauigkeit gerechtfertigt wird. Bildet man aber den Mittelwert aus den Transitionszahlen, die sich aus einer Zero-Delay- und einer Transport-Delay-Simulation ergeben, so erhält man eine durchschnittliche Abweichung gegenüber dem exakten Ergebnis von nur 0,6% (allerdings bei einzelnen Abweichungen von bis zu 45%, siehe Abbildung 5-6). Dabei laufen die Zero-Delay- und die Transport-Delay-Simulation zusammen immer noch durchschnittlich 11,5-mal so schnell wie die Real-Delay-Simulation ab.

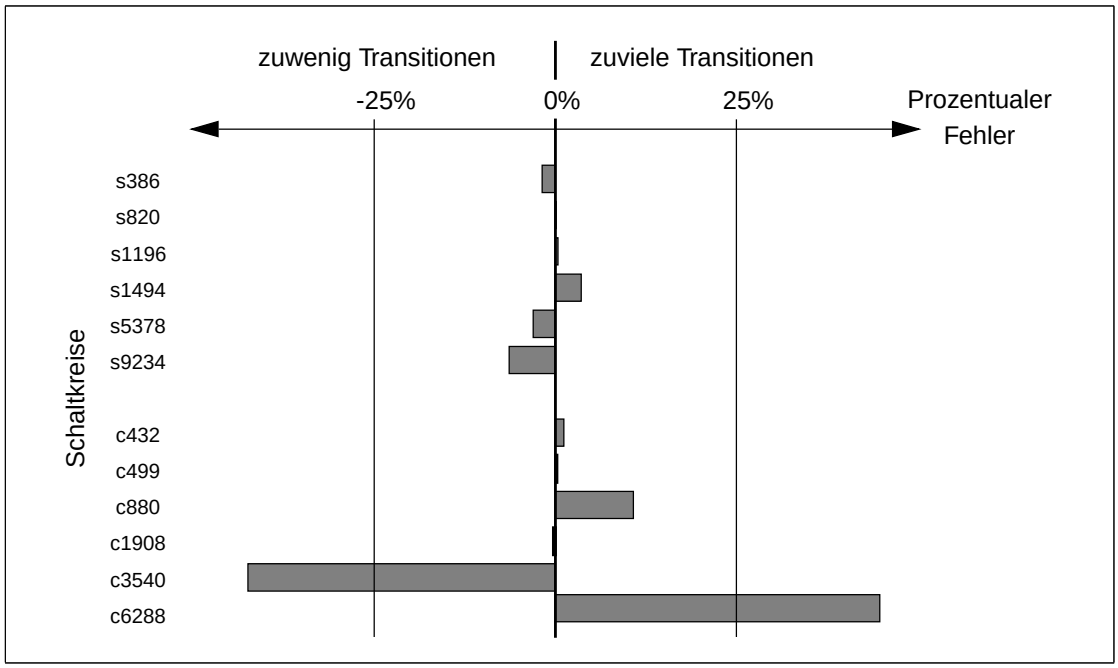


Abbildung 5-6: Fehlerrate des Durchschnitts aus den Ergebnissen der ZD- und TD-Simulation

Simulationsart	Genauigkeit	Durchschnittliche Laufzeit
Zero-Delay-Modell	33% Transitionen zuwenig erfaßt	1,48 s
Transport-Delay-Modell	32% Transitionen zuviel erfaßt	32,38 s
ZDM + RDM	0,6% Transitionen zuviel erfaßt	33,86 s
Real-Delay-Modell	exakte Anzahl an Transitionen erfaßt	343,24 s

Tabelle 5-3: Vergleich der Logiksimulationsarten

5.4 Absolute Simulationsergebnisse

In diesem Kapitel sind alle Simulationsergebnisse aufgeführt. *DD* bezieht sich dabei auf den bestehenden Simulator, während *KK* sich auf dessen in dieser Arbeit vorgestellten Erweiterung bezieht. Alle Zeitangaben sind in Sekunden gegeben.

Schaltung	Dauer mit DD	Dauer mit KK	Transitionen
s386	0,05	0,04	398410
s820	0,17	0,15	588652
s1196	0,22	0,35	1322598
s1494	0,33	0,2	1085450
s5378	1,89	2,45	5712176
s9234	3,58	9,57	2134528
c432	0,1	0,1	572226
c499	0,13	0,11	772538
c880	0,22	0,23	1237936
c1908	0,58	0,62	3529334
c3540	1,06	1,43	5556008
c6288	1,6	2,48	9269106

Tabelle 5-4: Simulation unter dem Zero-Delay-Modell

Schaltung	Dauer mit KK	Transitionen	Hazards
s386	0,66	513942	115532
s820	2,08	709720	121068
s1196	4,91	1997858	675260
s1494	2,93	1626826	541376
s5378	21,7	8088076	2375900
s9234	43,05	3894614	1760086
c432	1,15	1242296	670070
c499	1,29	1366450	593912
c880	3,6	2339808	1101872
c1908	9,85	9858534	6329200

Tabelle 5-5: Simulation unter dem Transport-Delay-Modell

Schaltung	Dauer mit KK	Transitionen	Hazards
c3540	215,55	25585974	20029966
c6288	81,74	341630104	332360998

Tabelle 5-5: Simulation unter dem Transport-Delay-Modell

Schaltung	Dauer mit Synopsys	Dauer mit KK	Transitionen	Hazards
s386	78,9	3,63	448046	49636
s820	154,2	11,14	649976	61324
s1196	314,2	70,42	1666576	343978
s1494	213,1	23,67	1406552	321102
s5378	226,4	145,52	6695870	983694
s9234	1692,6	130,9	2834228	699700
c432	195,1	16,47	918218	345992
c499	231,3	26,81	1072024	299486
c880	400,0	76,89	2004980	767044
c1908	859,8	210,18	6674932	3145598
c3540	1606,8	1930,06	10934100	5378092
c6288	25367,5	1473,14	317774630	308505524

Tabelle 5-6: Simulation unter dem Real-Delay-Modell

Dateien und Klassen

Anhang A

Dieser Anhang umfaßt eine ausführliche Auflistung der Dateien, welche die in dieser Arbeit entwickelten Erweiterung des bestehenden Simulators enthalten. Alle Klassen und deren Methoden werden beschrieben. Bei Funktionsparametern wurde dabei weniger auf deren Typzugehörigkeit als viel mehr auf deren Semantik in der zugehörigen Methode Wert gelegt. Die genaue Aufrufsyntax der Methoden kann dem Quell-Code entnommen werden.

A.1 Die Datei konstanten.h

Diese Datei enthält globale Konstantendefinitionen.

Konstante	Bedeutung
laenge_komp_name	Maximale Länge der im VHDL-Code definierten Komponentennamen.
Zeitbasis	Zeitgranularität in der Glitch-Simulation. Standardmäßig sind die Zahlenwerte der Zeitpunkte als Pikosekunden definiert. Dementsprechend lautet der Eintrag `Zeitbasis=1e-12`.
Durchschnittskapazitaet	Durchschnittliche Lastkapazität, die ein Folgegatter verursacht. Dieser Wert wird in der Glitch-Simulation verwendet und beträgt standardmäßig 10 fF.
SimulationsArt	Aufzählung über alle verfügbaren Simulationsarten: *ohneDelays* (Zero-Delay-Simulation), *mitDelays* (Transport-Delay-Simulation), *mit2Delays* (Real-Delay-Simulation) oder *mitGlitches* (Glitch-Simulation).
AusgabeArt	Aufzählung über alle verfügbaren Ausgabearten: *ohne*, *ToDoanzeige* oder *THsproGatter.*

Tabelle A-1: Konstantendefinitionen in konstanten.h

A.2 Die Dateien gatter.h und gatter.cpp

In diesen Dateien sind die Klassen `Zeitwert`, `Signalverlauf` und `Gatter` implementiert.

A.2.1 Die Klasse Zeitwert

Die Klasse `Zeitwert` dient zum Aufbau einer verketteten Liste zur Speicherung von Zeitwerten (siehe Kapitel 2.1.3). Ein Zeitwert besteht aus den Werten eines Signals für alle Testvektoren zu einem bestimmten Zeitpunkt.

Name	Datentyp	Zugriff	Beschreibung
signalwerte	signal*	public	enthält Werte über alle Testvektoren
zeitpunkt	int	public	Zeitpunkt, zu dem die Signalwerte Gültigkeit haben
next	Zeitwert*	public	Zeiger auf nächstes Listenelement

Tabelle A-2: Daten der Klasse Zeitwert

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	void
	Setzt alle Daten auf 0		
Destruktor	-	public	void
	Löscht alle Daten des Objekts		

Tabelle A-3: Methoden der Klasse Zeitwert

A.2.2 Die Klasse Signalverlauf

Diese Klasse dient zur Speicherung aller Zeitwerte eines Signals im Verlauf einer Simulation (siehe auch Kapitel 2.2). Sie enthält den Einstiegspunkt einer verketteten Liste von Zeitwerten und verwaltet die Liste unter Erhaltung der chronologischen Reihenfolge der Zeitwerte.

Name	Datentyp	Zugriff	Beschreibung
start	Zeitwert *	private	Einstiegspunkt der verketteten Liste von Zeitwerten

Tabelle A-4: Daten der Klasse Signalverlauf

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Signalwerte*
	Erstellt einen neuen Signalverlauf mit den übergebenen Signalwerten. Der erste Zeitpunkt ist definitionsgemäß „t<0“ (intern als -1 dargestellt).		
Destruktor	-	public	void
	Löscht die verkettete Liste von Zeitwerten		
neuer_zeitwert	void	public	Signalwerte*, Zeitpunkt
	Fügt neue Signalwerte in den Signalverlauf zu einem bestimmten Zeitpunkt ein. Dabei wird die verkettete Liste für einen schnelleren Zugriff nach Zeitpunkten aufsteigend sortiert. Es kommt zum Abbruch falls der Zeitpunkt kleiner als Null ist oder falls der Zeitpunkt bereits vorhanden ist, da beides auf einen Programmierfehler beim Aufrufer hinweist.		
operator[]	const Zeitwert*	public	Index
	Indizierter Zugriff auf die Zeitwerte: Ein Index von 0 liefert den Anfangswert bei „t<0“, 1 liefert den nächsten Zeitwert etc. in chronologischer Reihenfolge. Liegt der Index außerhalb der Grenzen, so wird ein Null-Pointer zurückgegeben.		

Tabelle A-5: Methoden der Klasse Signalverlauf

A.2.3 Die Klasse Gatter

Die Semantik der Klasse `Gatter` wird ausführlich in Kapitel 3.2 erläutert. Es folgt der konkrete Aufbau:

Name	Datentyp	Zugriff	Beschreibung
eingang	Gatter**	private	Zeigerfeld auf diejenigen Gatter, deren Ausgänge mit den Eingängen des aktuellen Gatters verbunden sind
ausgang	Signalverlauf*	private	Signalverlauf am Ausgang des Gatters. Dieser wird normalerweise während der Simulation ermittelt. Eine Ausnahme bilden die Primäreingänge (siehe Kapitel 3.2.2).

Tabelle A-6: Daten der Klasse Gatter

Name	Datentyp	Zugriff	Beschreibung
getriebene_gatter	int	private	Die Anzahl der Gatter, die am Ausgang des aktuellen Gatters hängen
nochzutreibend	int	private	Die Anzahl der Gatter, die während dem Simulationsablauf noch getrieben werden müssen. Sinkt dieser Wert auf 0, so wird der Ausgabesignalverlauf zur Speicherersparnis gelöscht.
gattertyp	const Komponente*	private	Zeiger auf den Komponententyp, dem das Gatter angehört
next	Gatter*	private	Zeiger auf das nächste Gatter in der verketteten Liste. Diese ist in der Klasse Netzliste verankert. Die Anordnung der Gatter in dieser Liste entspricht der Auswertungsreihenfolge während der Simulation.

Tabelle A-6: Daten der Klasse Gatter

Name	Datentyp	Zugriff	Parameter
1. Konstruktor	-	public	Zeigerfeld auf Gatter an Eingängen, Komponententyp
	Erstellt Gatter mit einem Zeigerfeld auf die Gatter an den Eingängen und ordnet einen Komponententyp zu. Die anderen Werte werden auf 0 gesetzt. Anschließend wird das Datum `getriebene_gatter` aller Gatter an den Eingängen um 1 erhöht.		
2. Konstruktor	-	public	void
	Erstellt ein Gatter, das einen Primäreingang der Schaltung repräsentieren soll (siehe Kapitel 3.2.2). Alle Daten werden auf 0 gesetzt. Der Ausgabesignalverlauf wird zu Beginn der Simulation mit der Methode `Setze_Ausgang` auf einen den Testvektoren entsprechenden Werteverlauf gesetzt.		
Destruktor	-	public	void
	Löscht alle Daten des Objekts		

Tabelle A-7: Methoden der Klasse Gatter

Name	Datentyp	Zugriff	Parameter
Setze_Ausgang	void	public	Signalverlauf
	Setzt den Ausgabesignalverlauf des Gatters bei Primäreingängen. Bei allen anderen Gattern führt der Aufruf zum Abbruch.		
init	void	public	void
	Setzt zu Beginn einer Simulation `nochzutreibend` = `getriebene_gatter`		
propagiere	TransHaz	public	SimulationsArt
	Stößt die Simulation des Gatters auf die übergebene Simulationsart an und gibt die Anzahl der Transitionen, Hazards bzw. den Energieaufwand zurück. Anschließend wird die Methode `treibe_mich` der Gatter an den Eingängen aufgerufen. Diese Methode reduziert bei jedem dieser Gatter das Datum `nochzutreibend` um 1.		
setze_next	void	public	Gatter*
	Setzt das Datum `next` des Gatters auf den übergebenen Wert		
naechstes	Gatter*	public	void
	Gibt den Wert des Datums `next` an den Aufrufer zurück		
treibe_mich	void	public	void
	Vermindert das Datum `nochzutreibend` um 1 und löscht den Ausgabesignalverlauf des Gatters, sobald dieser Wert Null erreicht		
print	void	public	void
	Gibt die Anzahl der Zeitwerte im Ausgabesignalverlauf aus.		

Tabelle A-7: Methoden der Klasse Gatter

Die Klassen `EinEingang`, `zweiEingaenge`, `nEingaenge`, `NOT`, `BUFFER`, `AND`, `ANDn`, `NAND`, `NANDn`, `OR`, `ORn`, `NOR`, `NORn`, `XOR`, `XORn`, `XNOR` und `XNORn` sind `friend` dieser Klasse.

A.3 Die Dateien komponentenbib.h, komponentenbib.cpp und kkonf

In diesen Dateien sind die Klassen `Komp_element`, `Kompx` und `Komponentenbib` implementiert. Sie dienen zur Erstellung der Komponententypen, die in dem zu simulierenden Schaltkreis vorkommen. In der Datei `kkonf` wird jedem Komponentennamen der VHDL-Datei, die den Schaltkreis beschreibt, die jeweilige logische Funktion, die Länge der Gatterlaufzeiten und eventuell die Anzahl der Eingänge zugeordnet.

A.3.1 Aufbau der Komponentenkonfigurationsdatei kkonf

Leere Zeilen sowie Zeilen, die mit einem „#“ beginnen, werden ignoriert. Alle sonstigen Zeilen müssen genau 5 Felder haben, die jeweils mit Leerzeichen oder Tabulatoren getrennt sein müssen.

Feld	Typ	Beschreibung
Komponentenname	char[]	Der Name der Komponente, mit dem sie in der VHDL-Datei definiert ist
Eingangsanzahl	int oder ’?’	Anzahl der Eingänge der Komponente. Falls sie nicht bekannt ist oder der Komponentenname in der VHDL-Datei als Makro mit variabler Eingangszahl verwendet wird: ’?’ eintragen. Dadurch wird die Eingangszahl dynamisch ermittelt.
Logische Verknüpfung	NOT, BUFFER, AND, NAND, OR, NOR, XOR, XNOR	Typ der logischen Verknüpfung, welche die Komponente realisiert
$Delay_{1->0}$	int	Gatterlaufzeit für einen Signalwechsel von 1 nach 0
$Delay_{0->1}$	int	Gatterlaufzeit für einen Signalwechsel von 0 nach 1. Dies ist außerdem der Wert, welcher bei der Transport-Delay-Simulation sowohl für den 0-1- als auch für den 1-0-Übergang verwendet wird.

Tabelle A-8: Aufbau der Datei kkonf

Es folgt die Konfigurationsdatei, die im Verlauf dieser Arbeit verwendet wurde:

```
#############################################################
#Komponentenkonfiguration nach IMS 0.8 um gate forest lib#
#############################################################
#
#############################################################
#Name  Eingaenge       Typ     Delay1->0   Delay0->1     #
#############################################################
invg           1       NOT        190        256
buffg          1       BUFFER     122        144
andg           2       AND        213        277
nandg          2       NAND       310        259
org            2       OR         240        271
norg           2       NOR        202        234
xorg           2       XOR        259        301
xnorg          2       XNOR       260        300
andg_n         ?       AND        213        277
nandg_n        ?       NAND       310        259
org_n          ?       OR         240        271
norg_n         ?       NOR        202        234
xorg_n         ?       XOR        259        301
xnorg_n        ?       XNOR       260        300
```

Listing A-1: Beispiel für eine Komponentenkonfiguration

A.3.2 Die Klasse Komp_element

Diese Klasse wird ausschließlich intern in der Klasse `Komponentenbib` verwendet. Sie dient zur Speicherung der Assoziation zwischen einem Komponentennamen und dem Zeiger auf das Objekt, das den zugehörigen Komponententyp repräsentiert. Die Klasse repräsentiert ein Element einer verketten Liste, welche die Klasse `Komponentenbib` verwaltet.

Name	Datentyp	Zugriff	Beschreibung
name	char*	private	Name der Komponente, unter dem sie in der VDHL-Datei definiert wurde
verweis	Komponente*	private	Zeiger auf das Objekt, welches den zugehörigen Komponententyp repräsentiert
next	Komp_element*	private	Verweis auf das nächste Objekt der verketteten Liste

Tabelle A-9: Daten der Klasse Komp_element

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Name, Verweis
	Setzt die Daten auf die übergebenen Werte und `next` auf 0		
Destruktor	-	public	void
	Löscht alle Daten des Objekts		

Tabelle A-10: Methoden der Klasse Komp_element

Die Klasse `Komponentenbib` ist `friend` dieser Klasse.

A.3.3 Die Klasse Kompx

Diese Klasse wird ebenso ausschließlich in der Klasse `Komponentenbib` verwendet. Objekte dieser Klasse speichern diejenigen Komponententypen, deren Eingangsanzahl nicht in der Komponentenkonfigurationsdatei enthalten sind. Beim Parsen der VHDL-Datei während des Aufbaus der Netzliste werden mit ihrer Hilfe die endgültigen Komponentenobjekte erstellt. Die Klasse repräsentiert ein Element einer verketten Liste, welche die Klasse `Komponentenbib` verwaltet.

Name	Datentyp	Zugriff	Beschreibung
name	char*	private	Name der Komponente, wie er in der VDHL-Datei definiert wurde
typ	NOT, BUFFER, AND, NAND, OR, NOR, XOR, XNOR	private	Typ der logischen Verknüpfung, welche die Komponente realisiert
delay01	int	private	Gatterlaufzeit für einen Signalwechsel von 0 nach 1
delay10	int	private	Gatterlaufzeit für einen Signalwechsel von 1 nach 0
next	Kompx*	private	Verweis auf das nächste Objekt der verketteten Liste

Tabelle A-11: Daten der Klasse Kompx

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Name, Logikfunktion, Gatterlaufzeiten
	Setzt die Daten auf die übergebenen Werte und `next` auf 0		
Destruktor	-	public	void
	Löscht alle Daten des Objekts		

Tabelle A-12: Methoden der Klasse Kompx

Die Klasse `Komponentenbib` ist `friend` dieser Klasse.

A.3.4 Die Klasse Komponentenbib

Diese Klasse erzeugt aus der Komponentenkonfigurationsdatei die Komponentenobjekte, die beim Aufbau der Netzliste den Gattern zugewiesen werden. Komponenten, deren Eingangsanzahl nicht in der Komponentenkonfiguration definiert sind, werden in einer Liste mit Objekten der Klasse `Kompx` gesichert und dynamisch beim Erstellen der Netzliste erzeugt.

Name	Datentyp	Zugriff	Beschreibung
anfang	Komp_element*	private	Einstiegspunkt der verketteten Liste von Objekten der Klasse `Komp_element`, welche die Assoziation zwischen Namen von Komponententypen und den Zeigern auf die entsprechenden Objekte speichern
ende	Komp_element*	private	Letztes Element der verketteten Liste zum schnelleren Einfügen von neuen Elementen
start	Kompx*	private	Einstiegspunkt der verketteten Liste von Komponententypen, deren Eingangsanzahl nicht in der Komponentenkonfigurationsdatei definiert ist

Tabelle A-13: Daten der Klasse Komponentenbib

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	void
	Setzt alle Daten auf 0		

Tabelle A-14: Methoden der Klasse Komponentenbib

Name	Datentyp	Zugriff	Parameter
Destruktor	-	public	void
	Löscht die beiden verketteten Listen		
liesKomponentenkonf	void	public	Name der Datei
	Liest die Komponentenkonfigurationsdatei zeilenweise ein und ruft für jede beschriebene Komponente `fuege_ein` oder `fuegex_ein` auf, je nachdem ob die Anzahl der Eingänge spezifiziert wurde oder nicht. Entspricht die Konfigurationsdatei nicht der Spezifikation (siehe Tabelle A-8) wird abgebrochen.		
erzeuge_Komponente	Komponente*	private	Name, Eingangszahl, Typname, Gatterlaufzeitzen
	Wählt in Abhängigkeit des logischen Typnamens und der Anzahl der Eingänge eine passende Komponentenklasse aus und erstellt ein Objekt dieser Klasse mit den entsprechenden Eingangszahlen und Gatterlaufzeiten. Die Adresse des neu erzeugten Objekts wird zurückgegeben.		
fuege_ein	Komponente*	private	Name, Eingangszahl, Typname, Gatterlaufzeitzen
	Erzeugt neue Komponentenobjekte mit Hilfe der Methode `erzeuge_Komponente` und fügt diese in die verkettete Liste der `Komp_element`-Objekte ein. Existiert bereits ein Objekt unter demselben Namen, so wird abgebrochen. Die Adresse des neu erzeugten Komponentenobjekts wird zurückgegeben.		
fuegex_ein	void	private	Name, Typname, Gatterlaufzeitzen
	Fügt die übergebenen Daten in die verkettete Liste der `Kompx`-Objekte ein. Existiert bereits ein Objekt unter demselben Namen, so wird abgebrochen.		

Tabelle A-14: Methoden der Klasse Komponentenbib

Name	Datentyp	Zugriff	Parameter
suche_x	Komponente*	public	Name, Eingangszahl
	Sucht den übergebenen Namen in der Liste der `Kompx`-Objekte. Wird er nicht gefunden erfolgt der Programmabbruch. Ansonsten wird für die übergebene Eingangszahl überprüft, ob bereits ein entsprechendes Komponentenobjekt erstellt wurde. Falls nicht, so wird es mit den im `Kompx`-Objekt vermerkten Charakteristika erzeugt. Anschließen wird der Zeiger auf das Komponentenobjekt zurückgeliefert.		
operator[]	Komponente*	public	Name
	Sucht das Komponentenobjekt mit dem übergebenen Namen in der `Komp_element`-Liste. Ist kein entsprechendes Objekt vorhanden, wird ein Zeiger auf 0 zurückgegeben.		

Tabelle A-14: Methoden der Klasse Komponentenbib

A.4 Die Dateien netzliste.h und netzliste.cpp

Diese Dateien definieren die einzige Klasse, die vom bereits existierenden Teil des Simulators von [DD 98] direkt benutzt wird. Beim Erstellen eines Objekts der Klasse `Netzliste` wird die Netzliste intern aufgebaut und steht über entsprechende Methoden zur Simulation bereit.

A.4.1 Die Klassen DDnachKK und port_gatter

Die Klasse `DDnachKK` dient zur Umwandlung der in [DD 98] gewählten Datenrepräsentation in eine sicherere, objektorientierte Darstellung mit weniger Redundanzen. Die Klasse `DDnachKK` beinhaltet eine eigene private Klasse namens `port_gatter`.

Die Klasse port_gatter

Hierbei handelt es sich um eine Hilfsklasse für `DDnachKK`. Sie dient zum Aufbau eines Elements einer verketteten Liste, die von `DDnachKK` verwaltet wird. Jedes Element speichert die Zuordnung zwischen einem Zeiger auf den Ausgang eines Gatters in der Darstellung von [DD 98] und dem entsprechenden Gatter in der hier gewählten Darstellung.

Name	Datentyp	Zugriff	Beschreibung
port	const port_list_t*	public	Zeiger auf den Ausgang eines Gatters in [DD 98]'s Darstellung
gatter	Gatter*	public	Zeiger auf korrespondierendes Gatter in der hier gewählten Darstellung
next	port_gatter*	public	Verweis auf das nächste Objekt der verketteten Liste

Tabelle A-15: Daten der Klasse port_gatter

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	void
	Setzt alle Daten auf 0		

Tabelle A-16: Methode der Klasse port_gatter

Die Klasse DDnachKK

Diese Klasse verwaltet die verkettete Liste der `port_gatter`-Elemente und ermöglicht den assoziativen Zugriff in Abhängigkeit von den Zeigerwerten der Gatter in [DD 98]'s Darstellung.

Name	Datentyp	Zugriff	Beschreibung
start	port_gatter*	private	Einstiegspunkt in die verkettete Liste von `port_gatter`-Elementen

Tabelle A-17: Daten der Klasse DDnachKK

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	void
	Setzt `start` auf 0		
Destruktor	-	public	void
	Löscht die verkettete Liste		

Tabelle A-18: Methoden der Klasse DDnachKK

Name	Datentyp	Zugriff	Parameter
fuege_ein	void	public	Gatterausgang bei [DD 98], entsprechendes Gatter hier
	Fügt eine Zuordnung zwischen dem Zeiger auf einen Gatterausgang bei [DD 98] und dem korrespondierenden Gatter in der hier gewählten Darstellung ein. Um den Suchaufwand zu verringern, werden die Elemente nach dem Zeigerwert von [DD 98] aufsteigend sortiert. Existiert bereits eine Zuordnung für einen bestimmten Gatterausgang, so wird abgebrochen.		
operator[]	Gatter*	public	Gatterausgang bei [DD 98]
	Ordnet assoziativ einem Zeiger auf einen Gatterausgang bei [DD 98] den Zeiger auf das entsprechende hier erzeugte Gatter zu. Existiert keine Zuordnung, so wird Null zurückgeliefert.		

Tabelle A-18: Methoden der Klasse DDnachKK

A.4.2 Die Klasse TransHaz

Dieser Datentyp wird von vielen Funktionen verwendet. Er erleichtert die Rückgabe von den in der Simulation gewonnenen Werten: Die Anzahl der Transitionen und Hazards bzw. einen Energiewert.

Name	Datentyp	Zugriff	Beschreibung
transitionen	long	public	Anzahl an Transitionen
hazards	long	public	Anzahl an Hazards
energie	double	public	Energieaufwand

Tabelle A-19: Daten der Klasse TransHaz

Name	Datentyp	Zugriff	Parameter
1. Konstruktor	-	public	void
	Setzt alle Variablen auf 0.		
2. Konstruktor	-	public	Transitionen, Hazards, Energiewert
	Setzt alle Variablen auf die übergebenen Werte		

Tabelle A-20: Methode der Klasse TransHaz

Name	Datentyp	Zugriff	Parameter
operator+=	TransHaz	public	const TransHaz
	Erhöht die Variablenwerte des TransHaz-Objekts um die Werte des rechts vom „+=“-Operanden stehenden TransHaz-Objekts.		

Tabelle A-20: Methode der Klasse TransHaz

A.4.3 Die Klasse Netzliste

Die Klasse Netzliste bildet die Schnittstelle zwischen dem bereits bestehenden Teil des Simulators von [DD 98] und den in dieser Arbeit vorgestellten Erweiterungen. Mit ihrer Hilfe wird die Netzliste der zu simulierenden Schaltung aufgebaut. Auch die eigentliche Simulation findet durch den Aufruf einer Methode dieser Klasse statt. Dabei müssen die gewünschte Simulationsmethode und die zu simulierenden Testvektoren übergeben werden.

Name	Datentyp	Zugriff	Beschreibung
anz_gatter	long	private	Anzahl der Gatter in der Netzliste (exklusive der Gatter, welche die Primäreingänge repräsentieren)
netz	Gatter*	private	Einstiegspunkt in die verkettete Liste der Gatter
primaereingaenge	Gatter**	private	Zeigerfeld auf diejenigen Gatter, welche die Primäreingänge der Schaltung repräsentieren
anz_primaereingaenge	int	private	Anzahl der Primäreingänge der Schaltung
resultat	TransHaz	private	Summe aller Transitionen und Hazards, bzw. des Energieverbrauchs
simulationstyp	SimulationsArt	private	Speichert die Art der Simulation während der Ausführung

Tabelle A-21: Daten der Klasse Netzliste

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Dateiname der Komponentenkonfiguration, Netzliste in [DD 98]'s Format
	Erstellt Komponentenobjekte unter Verwendung von Objekten der Klassen `Komponentenbib` und `DDnachKK`. Anschließend werden alle Gatter aus [DD 98]'s Netzliste in die hier verwendeten Gatterobjekte umgewandelt. Zusätzlich werden Gatter, welche die Primäreingänge repräsentieren, erzeugt. Dementsprechend werden alle Variablen des Objekts initialisiert.		
Destruktor	-	public	void
	Löscht die komplette Gatternetzliste sowie die Primäreingänge		
portliste	Gatter**	private	Eingangsanzahl, Portliste von [DD 98], DDnachKK-Bibliothek
	Wandelt eine Portliste (siehe [DD 98]) in ein Feld mit Zeigern auf die hier korrespondierenden Gatterobjekte unter Verwendung der übergebenen Zuordnungsbibliothek um und gibt einen Zeiger auf das neu erzeugte Feld zurück. Bei Fehlern wird eine Null zurückgeliefert.		
anzahl_ports	int	private	Portliste von [DD 98]
	Zählt die Anzahl der Ports in einer Portliste von [DD 98]		
simuliere	TransHaz	public	Anzahl der ungewichteten Testvektoren, gewichtete Testvektoren, Gewichtung, Ausgabeart, Simulationsart
	Wandelt zunächst die übergebenen Testvektorwerte in Ausgabesignalverläufe für diejenigen Gatter um, welche die Primäreingänge repräsentieren. Anschließend werden die Gatter sukzessive in Abhängigkeit von der gewählten Simulationsart durchsimuliert. Die übergebene Ausgabeart bestimmt, wieviel Informationen über jedes Gatter während der Simulation ausgegeben wird. Zurückgeliefert wird die Summe aller aufgetretenen Transitionen und Hazards, bzw. der gesamte Energieaufwand.		

Tabelle A-22: Methoden der Klasse Netzliste

A.5 Die Dateien komponenten.h und komponenten.cpp

In diesen Dateien werden die Komponentenklassen definiert. Die Komponentenobjekte werden aus der Komponentenkonfigurationsdatei von der Klasse `Komponentenbib` erzeugt. Diese Klassen und deren Verwendung werden ausführlich in Kapitel 3.1 beschrieben.

A.5.1 Die Klasse Komponente

Die abstrakte Klasse `Komponente` bildet die Superklasse aller Komponentenklassen. Sie wird eingehend in Kapitel 3.1.3 abgehandelt.

Name	Datentyp	Zugriff	Beschreibung
delay01	int	private	Gatterlaufzeit für einen Signalwechsel von 0 nach 1
delay10	int	private	Gatterlaufzeit für einen Signalwechsel von 1 nach 0
anzahl_vektoren	static int	private	Die Anzahl der Testvektoren
gewichtung	static const int*	protected	Die Gewichtung der Testvektoren
akt_ausgabe	static char*	protected	Entspricht im Pseudocode der Real-Delay- und der Glitch-Simulation dem Feld `aktuelle_Ausgabe` (s. Kapitel 4.4.3 und Kapitel 4.5.3)
vektorwert	static int*	protected	Allgemeines Feld mit unterschiedlicher Nutzung: einerseits dient es zur Zählung der Anzahl an aufgetretenen Transitionen und Hazards pro Vektor und andererseits zur Verwendung als *Zeitplan* für die Real-Delay- und die Glitch-Simulation.

Tabelle A-23: Daten der Klasse Komponente

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Gatterlaufzeiten
	Setzt die Gatterlaufzeiten auf die übergebenen Werte		

Tabelle A-24: Methoden der Klasse Komponente

Name	Datentyp	Zugriff	Parameter
Destruktor	virtual	public	void
	Löscht alle Daten des Objekts		
Delay01	int	public	void
	Gibt die Gatterlaufzeit für einen 0-1-Übergang zurück		
Delay01	int	public	void
	Gibt die Gatterlaufzeit für einen 1-0-Übergang zurück		
Eg0	virtual double	public	Impulsbreite
	Gibt den Energieverbrauch im Glitchbereich bei einem 1-0-1-Übergang in Abhängigkeit von der übergebenen Impulsbreite zurück, falls die Methode in einer Subklasse redefiniert wurde. Ansonsten tritt ein Fehler auf und es erfolgt der Programmabbruch.		
Eg1	virtual double	public	Impulsbreite
	Gibt den Energieverbrauch im Glitchbereich bei einem 0-1-0-Übergang in Abhängigkeit von der übergebenen Impulsbreite zurück, falls die Methode in einer Subklasse redefiniert wurde. Ansonsten tritt ein Fehler auf und es erfolgt der Programmabbruch.		
Et	virtual double	public	Anzahl der Folgegatter
	Gibt den Energieverbrauch einer Transition in Abhängigkeit von der Anzahl der Folgegatter zurück, falls die Methode in einer Subklasse redefiniert wurde. Ansonsten tritt ein Fehler auf und es erfolgt der Programmabbruch.		
gg0	virtual int	public	Anzahl der Folgegatter
	Gibt die Glitchgrenze beim 1-0-1-Übergang zurück, falls die Methode in einer Subklasse redefiniert wurde. Ansonsten tritt ein Fehler auf und es erfolgt der Programmabbruch.		
gg1	virtual int	public	Anzahl der Folgegatter
	Gibt die Glitchgrenze beim 0-1-0-Übergang zurück, falls die Methode in einer Subklasse redefiniert wurde. Ansonsten tritt ein Fehler auf und es erfolgt der Programmabbruch.		

Tabelle A-24: Methoden der Klasse Komponente

Name	Datentyp	Zugriff	Parameter
AnzahlVektoren	static int	public	void
	Gibt die Anzahl der (gewichteten) Testvektoren zurück		
setze_AnzahlVektoren	static void	public	Anzahl der Vektoren
	Setzt die Anzahl der Testvektoren auf den übergebenen Wert		
setze_Gewichtung	static void	public	Gewichtung der Testvektoren
	Setzt die Gewichtung der Testvektoren auf die übergebenen Werte		
init_vektorwert	void	protected	Wert
	Setzt alle Elemente des `vektorwert`-Feldes auf den übergebenen Wert (standardmäßig auf Null)		
statisch_init	static void	public	void
	Alloziert Speicher für die statischen Variablen `vektorwert` und `akt_ausgabe`		
statisch_loeschen	void	public	void
	Löscht die statischen Variablen `vektorwert` und `akt_ausgabe`		
anzahl_ths	TransHaz	protected	void
	Voraussetzung: In der Variablen `vektorwert` stehen vektorweise die Anzahl der während der Simulation aufgetretenen Transitionen. Diese Methode summiert alle Transitionen unter Berücksichtigung der Vektorgewichtungen auf und berechnet gleichzeitig die Summe der aufgetretenen Hazards. Diese Werte gibt sie anschließend als Objekt der Klasse TransHaz zurück.		
summe_ths	TransHaz	protected	Signalverlauf
	Zählt die Anzahl an aufgetretenen Transitionen während des Signalverlaufs und speichert diese in der Variablen `vektorwert`. Anschließend wird `anzahl_ths` aufgerufen und die Summe an Transitionen und Hazards zurückgegeben.		

Tabelle A-24: Methoden der Klasse Komponente

Name	Datentyp	Zugriff	Parameter
aktualisiere_ausgabe	int	protected	Simulationszeit $t_{Simulation}$
	Voraussetzung: Die Variable `vektorwert` enthält den Zeitplan in der Real-Delay- oder der Glitch-Simulation (s. Kapitel 4.4.2). In Abhängigkeit von der Simulationszeit werden durch diese Methode die Signalwerte der aktuellen Ausgabe (`akt_ausgabe`) dem Zeitplan entsprechend modifiziert.		
propagiere	TransHaz	public	Gatter, Simulationsart
	Wählt in Abhängigkeit von der gewählten Simulationsart die entsprechende Propagierungsfunktion für das Gatter und gibt deren Ergebnis zurück.		
propagiere_oD	virtual TransHaz	public	Gatter
	Abstrakte virtuelle Methode, welche die Simulation des Gatters nach der Zero-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen zurückgibt.		
propagiere_mD	virtual TransHaz	public	Gatter
	Abstrakte virtuelle Methode, welche die Simulation des Gatters nach der Transport-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards zurückgibt.		
propagiere_m2D	virtual TransHaz	public	Gatter
	Abstrakte virtuelle Methode, welche die Simulation des Gatters nach der Real-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards zurückgibt.		
propagiere_mG	virtual TransHaz	public	Gatter
	Abstrakte virtuelle Methode, welche die Simulation des Gatters nach der Glitch-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards und den Energieaufwand zurückgibt.		
anz_eingaenge	virtual int	public	void
	Abstrakte virtuelle Methode, welche die Anzahl der Komponenteneingänge zurückliefert		

Tabelle A-24: Methoden der Klasse Komponente

Name	Datentyp	Zugriff	Parameter
verknuepfe	virtual signal*	public	Feld von Signalwerten
	Abstrakte virtuelle Methode, welche je nach Komponente alle Signalwerte des übergebenen Feldes logisch verknüpft und die resultierenden Signalwerte zurückliefert		

Tabelle A-24: Methoden der Klasse Komponente

A.5.2 Die Klassen EinEingang, ZweiEingaenge und nEingaenge

Diese abstrakten Klassen werden ausführlich in Kapitel 3.1.4 vorgestellt.

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Gatterlaufzeiten
	Setzt die Gatterlaufzeiten auf die übergebenen Werte		
Destruktor	virtual	public	void
	Löscht alle Daten des Objekts		
propagiere_oD	virtual TransHaz	public	Gatter
	Definition der virtuellen Methode, welche die Simulation des Gatters nach der Zero-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen zurückgibt.		
propagiere_mD	virtual TransHaz	public	Gatter
	Definition der virtuellen Methode, welche die Simulation des Gatters nach der Transport-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards zurückgibt.		
propagiere_m2D	virtual TransHaz	public	Gatter
	Definition der virtuellen Methode, welche die Simulation des Gatters nach der Real-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards zurückgibt.		

Tabelle A-25: Methoden der Klassen EinEingang, ZweiEingaenge und nEingaenge

Name	Datentyp	Zugriff	Parameter
propagiere_mG	virtual TransHaz	public	Gatter
	Definition der virtuellen Methode, welche die Simulation des Gatters nach der Glitch-Delay-Simulation bewirkt und die Anzahl an aufgetretenen Transitionen und Hazards und den Energieaufwand zurückgibt.		
anz_eingaenge	virtual int	public	void
	Definition der virtuellen Methode, welche die Anzahl der Komponenteneingänge zurückliefert		

Tabelle A-25: Methoden der Klassen EinEingang, ZweiEingaenge und nEingaenge

A.5.3 Die Klassen NOT, BUFFER, AND, NAND, OR, NOR, XOR, XNOR, ANDn, NANDn, ORn, NORn, XORn und XNORn

Dies sind die Klassen der untersten Hierarchieebene (siehe Kapitel 3.1.2). Mit ihrer Hilfe werden die Komponentenobjekte, welche den Gattern zugeordnet werden, erzeugt.

Name	Datentyp	Zugriff	Parameter
Konstruktor	-	public	Gatterlaufzeiten, Eingangsanzahl (bei Komponenten mit mehr als zwei Eingängen)
	Setzt die Gatterlaufzeiten und die Anzahl der Eingänge (bei Komponenten mit mehr als zwei Eingängen) auf die übergebenen Werte		
Destruktor	virtual	public	void
	Löscht alle Daten des Objekts		
Eg0	virtual double	public	Impulsbreite
	Gibt den Energieverbrauch im Glitchbereich bei einem 1-0-1-Übergang in Abhängigkeit von der übergebenen Impulsbreite zurück, soweit die Werte für die Komponentenklassen schon berechnet wurden.		

Tabelle A-26: Methoden der Klassen der untersten Hierarchieebene

Name	Datentyp	Zugriff	Parameter
Eg1	virtual double	public	Impulsbreite
	Gibt den Energieverbrauch im Glitchbereich bei einem 0-1-0-Übergang in Abhängigkeit von der übergebenen Impulsbreite zurück, soweit die Werte für die Komponentenklassen schon berechnet wurden.		
Et	virtual double	public	Anzahl der Folgegatter
	Gibt den Energieverbrauch einer Transition in Abhängigkeit von der Anzahl der Folgegatter zurück, soweit die Werte für die Komponentenklassen schon berechnet wurden.		
gg0	virtual int	public	Anzahl der Folgegatter
	Gibt die Glitchgrenze beim 1-0-1-Übergang zurück, soweit die Werte für die Komponentenklassen schon berechnet wurden.		
gg1	virtual int	public	Anzahl der Folgegatter
	Gibt die Glitchgrenze beim 0-1-0-Übergang zurück, soweit die Werte für die Komponentenklassen schon berechnet wurden.		
verknuepfe	virtual signal*	public	Feld von Signalwerten
	Bewirkt die logische Verknüpfung aller Signalwerte des übergebenen Feldes in Abhängigkeit von der Komponentenklasse und liefert die resultierenden Signalwerte zurück.		

Tabelle A-26: Methoden der Klassen der untersten Hierarchieebene

A.6 Die Dateien signalkk1.h, signalkk2.h und signalkk2.cpp

Die Dateien `signalkk*` enthalten diejenigen Methoden, welche von der Implementierung der Klasse `signal` abhängen. Die Dateien sind als „include-Files“ für die Dateien `signal.h` und `signal.cpp` konzipiert, welche bei [DD 98] die Definition der Klasse `signal` beinhalten.

Name	Datentyp	Zugriff	Parameter
ink_vektorwert	void	public	Vektorfeld über alle Testvektoren
	Übergeben wird ein Vektorfeld, welches dieselbe Anzahl an Feldern hat wie Testvektoren existieren (Bsp.: die Variable `vektorwert` der Klasse `Komponente`). Der Wert eines jeden Feldes, dessen Signalwert 1 beträgt, wird um 1 erhöht.		
signal2ausgabe	void	public	Feld mit aktueller Ausgabe und dessen Dimension (= Anzahl der Testvektoren)
	Kopiert das Signal in die Feld-Darstellung um.		
ausgabe2signal	void	public	Feld mit aktueller Ausgabe, dessen Dimension (= Anzahl der Testvektoren), Signalmodus
	Kopiert die Feld-Darstellung in das Signal um. Der Signalmodus gibt an, ob die gepackte Signaldarstellung zu verwenden ist (siehe [DD 98]).		
zaehle_transitionen	int	public	Signal, Testvektorgewichtung
	Zählt die Anzahl der unterschiedlichen Stellen zwischen den beiden Signalen und gewichtet sie der Testvektorgewichtung entsprechend.		
1. plane_ereignisse	int	public	$t_{Simulation}$, letzter_Wert, aktuelle Ausgabe, Gatterlaufzeiten, Zeitplan
	Führt die Planung der auszuführenden Transitionen in der Real-Delay-Simulation wie in Kapitel 4.4.3 beschrieben durch. Die aktuellen und die letzten Signalwerte werden gegenübergestellt und auftretende Transitionen im Zeitplan zu den entsprechenden Zeitpunkten eingetragen. Der Zeitpunkt der nächsten Ausgabe wird zurückgeliefert.		

Tabelle A-27: zusätzliche Methoden der Klasse Signal

Name	Datentyp	Zugriff	Parameter
2. plane_ereignisse	int	public	$t_{Simulation}$, letzter_Wert, aktuelle Ausgabe, Gatterlaufzeiten, Testvektorgewichtung, Zeitplan, Komponente, Energiesumme
	Führt die Planung der auszuführenden Transitionen in der Glitch-Simulation wie in Kapitel 4.5.3 beschrieben durch. Die aktuellen und die letzten Signalwerte werden gegenübergestellt und auftretende Transitionen im Zeitplan zu den entsprechenden Zeitpunkten eingetragen. Zusätzlich wird der Energiebedarf der Glitches berechnet und zu der Energiesumme hinzuaddiert. Der Zeitpunkt der nächsten Ausgabe wird zurückgeliefert.		

Tabelle A-27: zusätzliche Methoden der Klasse Signal

Anwendung

Anhang B

B.1 Die neuen Optionen

Die in dieser Arbeit vorgestellten Erweiterungen wurden in den bestehenden Simulator namens ASSet von [DD 98] integriert. Eine umfangreiche Erläuterung der Optionen von ASSet ist in [DD 98] aufgeführt.

Diese Optionen wurden um folgende Punkte ergänzt:

- `-k0:` Simulation unter dem Zero-Delay-Modell
- `-k1:` Simulation unter dem Transport-Delay-Modell
- `-k2:` Simulation unter dem Real-Delay-Modell
- `-k3:` Simulation unter dem Glitch-Modell

Die Simulationsarten entsprechen exakt den in Kapitel 4 vorgestellten Varianten.

Verarbeitungsschema

Anhang C

C.1 Vorbereitung der Simulationsdaten

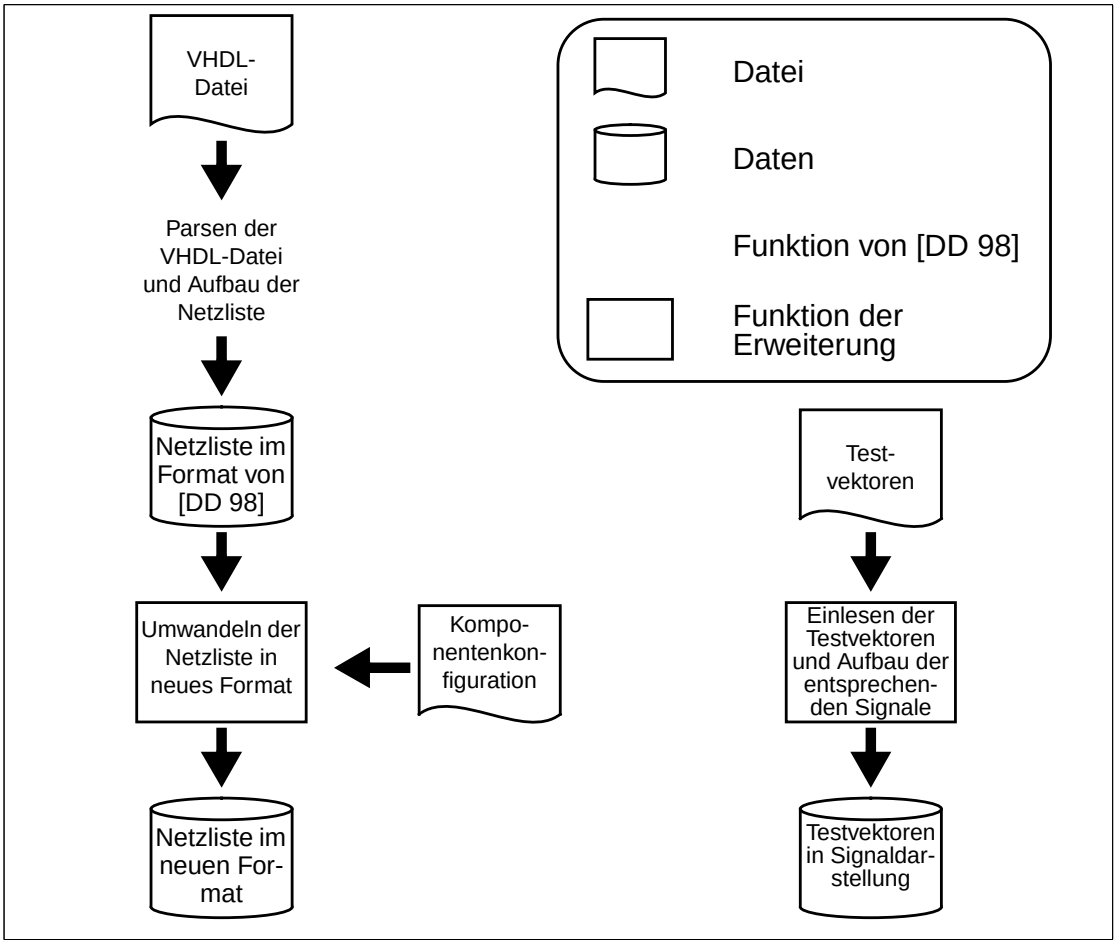


Abbildung C-1: Schema der Datenaufbereitung

C.2 Verarbeitung der Simulationsdaten

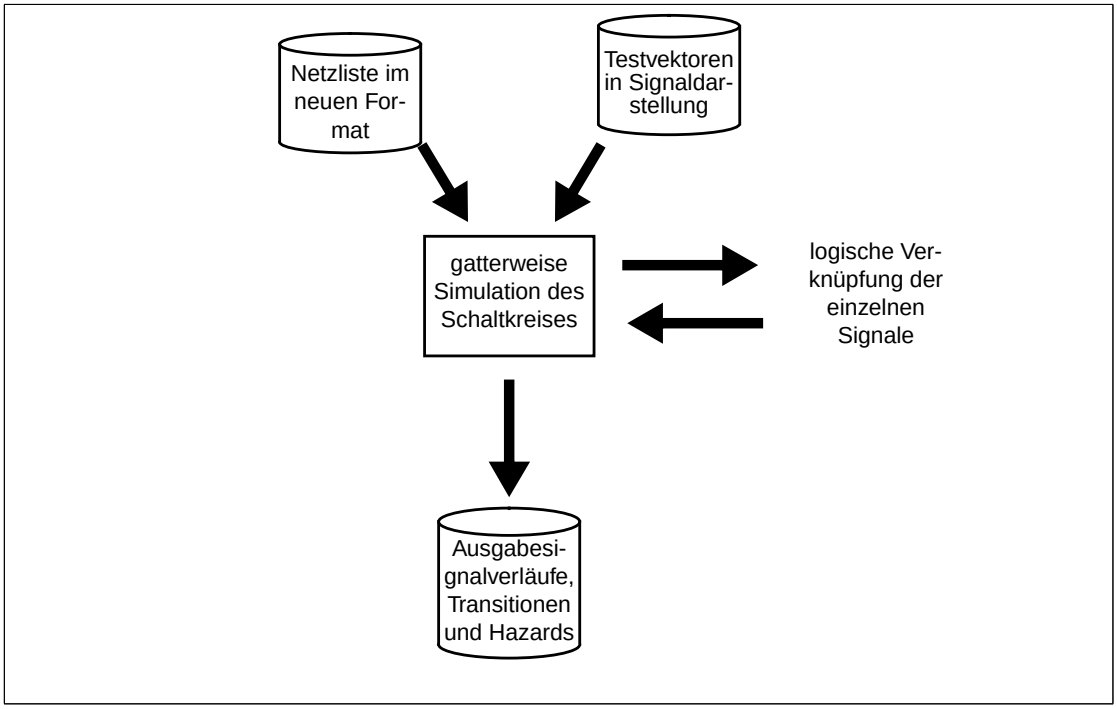
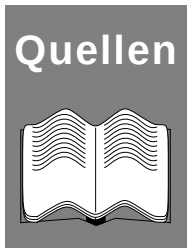


Abbildung C-2: Simulationsschema

Literaturverzeichnis



[MB 98] M.Bühler, D. Dallmann, U. G. Baitinger: *Switching Activity Analysis Using a Set Theoretical Approach;* GI/ITG/GME Workshop 1998, Paderborn, ISBN 3-931466-35-3

[Egg C++] Bernd Eggink: *C++ für C-Programmierer;* RRZN Hannover

[Mont 97] Monteiro et al. *Estimation of Average Switching Activity in Combinational Logic Circuits Using Symbolic Simulation;* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 16, No. 1, January 1997, pp. 121-127

[SCH 95] P.H.Schneider: *PAPSAS: A Fast Switching Activity Simulator;* PAT-MOS'95, 1995, pp. 351-360

[DD 98] Daniel Dallmann: *Entwicklung eines symbolischen Schaltungssimulators;* Diplomarbeit 1998, Universität Stuttgart, Fakultät Informatik

[KK 98] K. Kapp, M. Bühler, D. Dallmann, U.G. Baitinger: *TESA: Timeparallel Estimation of Switching Activity under a Real Delay Model;* 1998

[WIL 98] Christof Maluck: *Untersuchung von Hazards und Glitches in CMOS-Gattern;* Studienarbeit 1998, Universität Stuttgart, Fakultät Informatik

[PA 90] Peter J. Ashenden: *The VHDL Cookbook;* Dept. Computer Science, University of Adelaide, South Australia, 1990

Hiermit versichere ich, daß ich diese Arbeit selbstständig verfaßt und bei der Erstellung nur die angegebenen Hilfsmittel verwendet habe.

_____

Kai Kapp