Albert–Ludwigs–Universität Freiburg

Institut für Informatik

Lehrstuhl für Rechnerarchitektur
Prof. Dr. Bernd Becker

Diplomarbeit

Verifikation von Dividierern mit Word-Level-Decision-Diagrams

Achim D. Brucker

25. April 2000

千里の行も足下に始る。

Senri no kō mo ashimoto hi hajimaru.

Selbst ein Weg von tausend Meilen beginnt mit dem ersten Schritt.

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit allein und nur unter Verwendung der angegebenen Hilfsmittel verfasst habe.

Denzlingen, den 25. April 2000 Achim D. Brucker

Zusammenfassung

Da die späte Entdeckung von Fehlern in einem Schaltkreisdesign hohe Kosten verursacht, nimmt die Bedeutung der Verifikation und Validierung zu. Deutlich wurde dies 1994 mit dem „Pentium Bug“. Seit dieser Zeit werden verstärkt Verfahren zur Verifikation von arithmetischen Schaltkreisen, insbesondere der Division, untersucht.
Im Bereich der Hardwareverifikation sind Entscheidungsdiagramme die wichtigsten Datenstrukturen zur Repräsentation boolescher Funktionen. Allerdings konnte 1998 gezeigt werden, dass die Berechnungsstärke der bekannten Entscheidungsdiagramme nicht ausreicht, um die Division effizient darstellen zu können.
In dieser Arbeit wird ein neuer Ansatz zur Verifikation von Dividiererschaltkreisen vorgestellt, bei dem durch eine Transformation vermieden wird, die Division als Entscheidungsdiagramm darstellen zu müssen. Mit diesem Verfahren ist erstmals eine vollständig automatische Verifikation der nonrestoring Division nur durch den Einsatz von Entscheidungsdiagrammen möglich.
Stichwörter:
formale Verifikation, binäre Entscheidungsdiagramme, K∗BMD, nonrestoring Division

Abstract

Late detection of design errors typically results in higher costs, therefore the importance of design verification and validation increases. This was especially shown in 1994 by the “Pentium bug”. Since then the effort put into the verification of arithmetic circuits, particularly division, has increased.
In the area of the hardware verification decision diagrams are the most important data structures for the representation of boolean functions. However, in 1998 was shown that the representational power of any known decision diagram ist too weak to efficiently represent division.
In this work a new approach for the verification of divider circuits is introduced, which by a transformation avoids the representation of the division operation as decision diagram. With this approach it was the first time possible to verify the nonrestoring division automatically only by the application of decision diagrams.
Keywords:
formal verification, binary decision diagrams, K∗BMD, nonrestoring division

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

In den letzten Jahren hat der Computer im privaten Bereich verstärkt Einzug gehalten. In nahezu allen Bereichen unseres täglichen Lebens können und wollen wir auf die Hilfe der Informationstechnologien nicht mehr verzichten.

Am Anfang dieser Entwicklungen standen einfache mechanische Rechenmaschinen. Bereits 1623 baute Wilhelm Schickard eine Maschine, die alle vier Grundrechenarten beherrschte. Eine weitere wichtige Arbeit war die Additionsmaschine, die 1641 von Blaise Pascal gebaut und die 1673 von Gottfried W. Leibniz verbessert wurde. Allerdings erinnern erst die Entwürfe für einen programmgesteuerte Rechenautomaten, die 1834 von Charles Babbage vorgestellt wurden, entfernt an die Arbeitsweise heutiger Computer.

Der erste „Computer“, der einer breiten Bevölkerungsschicht bekannt wurde, war die elektromechanische Lochkartenmaschine, die 1890 von Hermann Hollerith gebaut wurde. Ohne die Hollerith-Maschine wäre die damalige Volkszählung in den USA nicht möglich gewesen. Der Deutsche Konrad Zuse stellte 1937 den ersten elektromechanischen Rechner, die Z1, vor. Wie die heute üblichen Computer arbeitete die Z1 bereits mit Binärzahlen. Als die ersten programmgesteuerten Rechner, die als direkte Vorläufer unserer heutigen Computer angesehen werden können, gelten die elektromechanische Z3 (1941) von Konrad Zuse und die elektronische (auf der Röhrentechnologie basierende) Rechenmaschine von John Atanasoff.

Bis Mitte des 20. Jahrhunderts waren Computer nur für Wissenschaftler von Interesse und einige Geschäftsleute waren auch der Meinung, dass dies noch lange so bleiben würde:

> *I think there's a world market for about five computers.*
> Thomas J. Watson (Chairman of the Board, IBM), 1943

Seinen Siegeszug begann der Computer 1945 mit der ENIAC[1], einem aus über 18000 Vakuumröhren bestehendem Rechner. Die ENIAC wog 30 Tonnen und benötigte 140 Kilowatt. Eine Addition dauerte $200\mu s$, eine Multiplikation $3ms$ und eine Division etwa $30ms$.

Erst als der 1947 erfundene Transistor die Vakuumröhre ablöste, konnte der Computer zu einem Werkzeug „für jedermann“ werden. Es sollte aber bis 1971 dauern, bis erste „Taschenrechner“ der Öffentlichkeit vorgestellt werden konnten. Zu diesem Zeitpunkt

[1] Electronic Numerical Integrator and Computer

Tabelle 1.1. Anzahl Transistoren gebräuchlicher Intel Prozessoren

Jahr	Prozessor	Anzahl Transistoren
1978	Intel 8086	29000
1982	Intel 80286	130000
1985	Intel 80386	275000
1989	Intel 80486	1200000
1992	Intel Pentium	3100000
1995	Intel Pentium Pro	5500000
1997	Intel Pentium II	7500000
1999	Intel Pentium III	9500000

war der Siegeszug der Computer nicht mehr aufzuhalten, aber erst 1980 wurde mit dem ZX80 von Sinclair erstmals ein Computer präsentiert, der in „großen“ Stückzahlen die Privathaushalte eroberte.

Die rasante Entwicklung der Computerindustrie in den letzten 25 Jahren lässt sich recht gut durch die Komplexität der Prozessoren veranschaulichen. In Tabelle 1.1 ist die Komplexität der gebräuchlichsten Prozessoren der Firma Intel durch die Anzahl der Transistoren wiedergeben.

Diese Entwicklung verursacht aber auch Probleme, denn zum einen steigt die Komplexität der Prozessoren stark, zum anderen wird gerade durch ihren vermehrten Einsatz in sicherheitsrelevanten Bereichen (beispielsweise die Medizin- und Militärtechnik) ein sehr hohes Maß an Zuverlässigkeit und Korrektheit gefordert. Dazu gehört sowohl die Korrektheit des Entwurfes, die durch die *Verifikation* und *Validierung* des Designs garantiert werden muss, als auch die Vermeidung von Fehlern bei der Produktion (*Testen*).

Der Glaube in Korrektheit moderner Computer wurde 1994 schwer erschüttert, als in dem damals modernsten Prozessor der Firma Intel, dem „Pentium“, ein Fehler in der Divisionseinheit entdeckt wurde. Zwar hatten auch nahezu alle anderen bis dahin gefertigten Prozessoren Fehler[2], diese waren aber meist nur einem kleinen Fachpublikum bekannt und traten nur unter sehr speziellen Umständen auf. Bei dem als „Pentium Bug“ oder „FDIV-Bug“ bekannt gewordenen Fehler konnte erstmal jeder Benutzer die Fehlfunktion leicht erkennen:

Beispiel (Pentium Bug) Sei $x = 4195835$ und $y = 3145727$. Betrachte nun

$$z = x - \frac{x}{y} \cdot y \; . \tag{1.1}$$

[2]Eine Übersicht der wichtigsten Fehler in den Prozessoren der x86 Linie von Intel bietet [24]. Aber auch die aktuellen Prozessoren sind nicht fehlerfrei, wie sich in den „Spezification Updates“ der Hersteller nachlesen lässt, so ist erst im Januar 2000 ein neuer „Rechenfehler“ des aktuellen Pentium III bekannt geworden. Er trägt die Bezeichnung E59 und ist in [1] dokumentiert.

Korrekt ist $z = 0$, während der (fehlerhafte) Intel Pentium 256 als Ergebnis liefert. □

Zum ersten Mal wurde ein Fehler eines Computerchips in allen Nachrichten und nicht nur in der Fachpresse publik gemacht. Zu Beginn versuchte Intel den Fehler herunterzuspielen, da bei er „Normalanwendern“ nur einmal in mehr als tausend Jahren auftreten würde (siehe [41]). Gleichzeitig erregte der Fehler aber besonders beim Fachpublikum großes Interesse. Innerhalb kürzester Zeit wurden sehr genaue Analysen des Fehlers und verschiedene Fehlermodelle (z.B. in [9, 16, 36]) veröffentlicht.

Durch den öffentlichen Druck und um einen weiteren Image-Verlust zu verhindern, startete Intel die bisher größte Umtauschaktion im IT-Bereich. Jeder Anwender konnte seinen Prozessor recht einfach durch ein neues Modell ohne FDIV-Bug austauschen. Die geschätzten Kosten der Umtauschaktion beliefen sich allein im 4. Quartal 1994 auf etwa 475 Millionen US Dollar.[3]

Seit diesem Zeitpunkt widmen sich vermehrt Wissenschaftler den Problemen, die bei der Verifikation von Schaltkreisen auftreten können. Konnten in den letzten Jahren in einigen Bereichen große Erfolge gefeiert werden, so hat sich die Verifikation von Dividierern als besonders schwierig erwiesen. Inzwischen konnte durch die Arbeiten von Nakanishi (中西正樹) [32] sowie Scholl, Becker und Weis [39, 40] nachgewiesen werden, dass die Darstellung der Division mit den üblichen Datenstrukturen nicht effizient möglich ist.

Die vorliegende Arbeit untersucht einen neuen Ansatz zur automatischen Verifikation von Dividiererschaltkreisen, bei dem die direkte Darstellung der Division vermieden wird. Hierzu werden Word-Level Diagramme als Datenstrukturen verwendet, die sich bereits zur Verifikation von Additions- und Multiplikationsschaltkreisen bewährt haben.

Aufteilung der Arbeit

Die Arbeit beginnt mit einer Darstellung der wichtigsten theoretischen Grundlagen, die zum Verständnis der weiteren Kapitel notwendig sind. Hierzu werden in Kapitel 2 sowohl boolesche Funktionen und Entscheidungsdiagramme definiert, als auch die gebräuchlichsten Zahlendarstellungen betrachtet. Die wichtigsten Algorithmen zur Division und Hinweise zu ihrer Implementierung als Schaltkreis werden in Kapitel 3 erklärt.

In Kapitel 4 werden die bisherigen Ansätze zur Verifikation von Dividiererschaltkreisen vorgestellt, sowie ein neue Idee zur automatischen Verifikation eingeführt. Die ersten Ergebnisse, die durch verschiedene Implementierungen des in Kapitel 4 vorgestellten Grundverfahrens erzielt wurden, werden in Kapitel 5 erläutert. Basierend auf den Ergebnissen aus Kapitel 5 wird in Kapitel 6 ein Verfahren zur automatischen Verifikation von Dividiererschaltkreisen und dessen Erfolge bei der nonrestoring Division vorgestellt.

Durch die geschickte Ausnutzung von Don't-Care-Belegungen ist eine weitere Variante des vorgestellten Verfahrens denkbar, die eventuell Vorteile bei der Adaption des

[3] Intel schloss das 4. Quartal 1994 trotzdem mit schwarzen Zahlen ab.

Verfahrens auf andere Designs bietet. Auf erste Untersuchungen hierzu wird in Kapitel 7 eingegangen.

Kapitel 8 fasst die in dieser Arbeit erzielten Resultate nochmals zusammen und gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen.

2. Grundlagen

In diesem Kapitel werden die wichtigsten Definitionen und Datenstrukturen eingeführt, die zur Verifikation von Schaltkreisen mit Entscheidungsdiagrammen[1] benötigt werden. Die hier vorgestellten Grundlagen lassen sich in einer ausführlichere Form z.B. in den einführenden Kapiteln von [11] und [30] nachlesen. Anschließend wird eine kurze Einführung in die Darstellung von Zahlen und deren Verarbeitung mittels Schaltkreisen gegeben.

Einen umfassenden Überblick der verschiedenen Methoden zur formalen Hardware-Verifikation bietet [17]. Eine Einführung in die Verifikation von arithmetischen Funktionen mit Entscheidungsdiagrammen kann in [7, 8] nachgelesen werden.

2.1. Boolesche Funktionen und Kofaktorbildung

2.1.1. Boolesche Funktionen

Die Grundlage der meisten Schaltkreisrealisierungen sind boolesche Funktionen. Boolesche Funktionen bilden eine Menge von booleschen Variablen, welche Werte der zweiwertigen Menge $\mathbf{B} = \{0, 1\}$ annehmen können, auf eine Teilmenge des $\mathbf{B}^m$ ab.

Definition 2.1.1 (boolesche Funktion) Eine Funktion $f : \mathbf{B}^n \to \mathbf{B}^m$ heißt boolesche Funktion mit n Eingängen (Variablen) und m Ausgängen. □

Werden die m Ausgänge geeignet zusammengefasst (siehe Abschnitt 2.3.2), können pseudoboolesche Funktionen definiert werden, die in die ganzen Zahlen $\mathbf{Z}$ abbilden.

Definition 2.1.2 (ganzzahlige Funktion) Eine Funktion $f : \mathbf{B}^n \to \mathbf{Z}$ heißt pseudoboolesche oder ganzzahlige Funktion mit n Variablen. □

Ist der Urbildbereich einer booleschen Funktion eine echte Teilmenge des $\mathbf{B}^n$, wird die Funktion als unvollständig spezifiziert bezeichnet.

Definition 2.1.3 (unvollständig spezifizierte boolesche Funktion) Eine Funktion $f : D \to \mathbf{B}^m$ mit $D \subset \mathbf{B}^n$ heißt unvollständig spezifizierte boolesche Funktion mit n Eingängen (Variablen). □

[1] engl. Decision Diagrams

Die Elemente der Menge $\mathbf{B}^n \backslash D$ werden als *don't cares* bezeichnet, somit zerfällt das Urbild einer booleschen Funktion $f : \mathbf{B}^n \rightarrow \mathbf{B}$ in die disjunkten Mengen, die auf 1 oder 0 abgebildet werden sowie die Don't Cares.

Definition 2.1.4 (ON-, OFF- und DC-Menge) Sei $D \subseteq \mathbf{B}^n$ der Definitionsbereich $\mathrm{DEF}(f)$ der (unvollständig spezifizierten) booleschen Funktion $f : D \rightarrow \mathbf{B}$. Die Menge

$$\mathrm{ON}(f) := \{\alpha | \alpha \in D \wedge f(\alpha) = 1\} \tag{2.1}$$

heißt die ON-Menge von f. Die Elemente der ON-Menge werden als *Minterme* bezeichnet. Die Menge

$$\mathrm{OFF}(f) := \{\alpha | \alpha \in D \wedge f(\alpha) = 0\} \tag{2.2}$$

heißt die OFF-Menge von f. Die Menge

$$\mathrm{DC}(f) := \mathbf{B}^n \backslash \mathrm{DEF}(f) \tag{2.3}$$

heißt die DC[2]-Menge von f. □

Bei der Realisierung einer booleschen Funktion als Schaltkreis (siehe z.B. [30] oder [44]) ist meist nur das Verhalten der Realisierung auf dem Definitionsbereich von Interesse, dies wird durch den Begriff der Erweiterung formalisiert. Beispielweise wird eine Funktion gesucht, die in einem bestimmten Kostenmaß eine besonders günstige Schaltkreisrealisierung ermöglicht und auf dem Definitionsbereich mit der Spezifikation identisch ist. Es wird also eine Erweiterung gesucht, die eine „günstige“ Schaltkreisrealisierung ermöglicht.

Definition 2.1.5 (Erweiterung) Eine Erweiterung einer unvollständig spezifizierten booleschen Funktion f ist eine (unvollständig spezifizierte) boolesche Funktion g mit

$$\mathrm{DEF}(f) \subseteq \mathrm{DEF}(g) \tag{2.4}$$

und

$$\forall \alpha \in \mathrm{DEF}(f) : g(\alpha) = f(\alpha) \; . \tag{2.5}$$

Eine Erweiterung einer booleschen Funktion f heißt vollständig, wenn g eine vollständig spezifizierte boolesche Funktion ist, d.h. wenn $\mathrm{DEF}(g) = \mathbf{B}^n$ gilt. □

Oft wird der Definitionsbereich einer booleschen Funktion mit Hilfe der charakteristischen Funktion angegeben, wobei die ON-Menge der charakteristischen Funktion von f genau $\mathrm{DEF}(f)$ darstellt.

[2] engl. Don't Care

Definition 2.1.6 (Charakteristische Funktion) Seien A, B Mengen mit $B \subseteq A$. Die charakteristische Funktion von B bezüglich A ist definiert als:

$$\chi_B(x) = \begin{cases} 1 & \text{falls} \quad x \in B \\ 0 & \text{falls} \quad x \notin B \end{cases} \tag{2.6}$$

□

Neben dem Defintions- oder Urbildbereich ist oft das Bild, genauer der Bildbereich, einer Funktion von Interesse. Der Bildbereich beschreibt die möglichen Funktionswerte, genauer: Die ON-Menge der den Bildbereich darstellenden Funktion enthält genau die Werte, auf die, unter Berücksichtigung des Definitionsbereiches, die Funktion f abbildet.

Definition 2.1.7 (Bildbereich) Das Bild $\chi_{\text{img}(f,C)}$ einer booleschen Funktion f bezüglich einer Definitionsmenge $C \subseteq \mathbf{B}^n$ ist definiert als

$$\chi_{\text{img}(f,C)}(y) = \begin{cases} 1 & \text{falls } \exists x \in C \bullet f(x) = y \\ 0 & \text{sonst} \end{cases} \tag{2.7}$$

□

In ähnlicher Weise kann das Bild eines Vektors boolescher Funktionen, und damit einer Funktion $f : \mathbf{B}^n \to \mathbf{B}^m$, definiert werden. In diesem Fall beschreibt die ON-Menge der den Bildbereich darstellenden Funktion genau die Tupel, auf welche der Funktionsvektor abbildet.

Definition 2.1.8 (Bildbereich eines Vektors boolescher Funktionen) Das Bild $\chi_{\text{img}(f,C)}$ eines Vektors $f = (f_1, \ldots, f_m)$ von booleschen Funktionen $(f_i : \mathbf{B}^n \to \mathbf{B}, 1 \leq i \leq m)$ bezüglich der Definitionsmenge $C \subseteq \mathbf{B}^n$ ist definiert durch

$$\chi_{\text{img}(f,C)}(y_1, \ldots, y_m) = \exists x \in \mathbf{B}^n \bullet \chi_C(x) \cdot \left(\prod_{i=1}^{m} (\overline{y_i \oplus f_i(x)}) \right) \tag{2.8}$$

□

2.1.2. Kofaktorbildung

Eine weit verbreitete Datenstruktur zur effizienten Darstellung und Manipulation von booleschen Funktionen sind Entscheidungsdiagramme. Bevor diese in Abschnitt 2.2 formal eingeführt werden können, sind noch einige Begriffe zu klären.

Definition 2.1.9 (Kofaktor) Sei $f : \mathbf{B}^n \to \mathbf{B}$ oder $f : \mathbf{B}^n \to \mathbf{Z}$. Für $c \in \mathbf{B}$ ist der Kofaktor von f nach $x_i = c$ gegeben durch:

$$f_i^c(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) := f(x_1, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n) \tag{2.9}$$

□

Häufig wird für den Kofaktor von f nach $x_i = c$ auch $f_{x_i=c}$ oder $f_{x_i}^c$ geschrieben.

Definition 2.1.10 (Boolesche Differenz) Als (boolesche) Differenz von f bezüglich x_i wird definiert:

$$f_i^2 := \begin{cases} f_i^1 \oplus f_i^0, & \text{falls } f : \mathbf{B}^n \to \mathbf{B} \\ f_i^1 - f_i^0, & \text{falls } f : \mathbf{B}^n \to \mathbf{Z} \end{cases} \tag{2.10}$$

□

Der Kofaktor $f_i^c : \mathbf{B}^n \to \mathbf{Z}$ ($f_i^c : \mathbf{B}^n \to \mathbf{B}$) einer Funktion $f : \mathbf{B}^n \to \mathbf{Z}$ ($f : \mathbf{B}^n \to \mathbf{B}$) hängt nur noch von $n-1$ Variablen ab. Der Wert des Kofaktors kann also in natürlicher Weise als Funktion vom Typ $\mathbf{B}^{n-1} \to \mathbf{Z}$ ($\mathbf{B}^{n-1} \to \mathbf{B}$) interpretiert werden. Diese Interpretation wird in den Shannonschen Entwicklungsätzen 2.1.1 und 2.1.2 ausgenutzt, um eine Funktion rekursiv in einfachere Teilfunktionen zu zerlegen.

Anmerkung 2.1.1 Die Bildung von Kofaktoren ist kommutativ, d.h. für $f : \mathbf{B}^n \to \mathbf{B}$ oder $f : \mathbf{B}^n \to \mathbf{Z}$ und $c, d \in \mathbf{B}, i \neq j$ gilt:

$$(f_i^c)_j^d = (f_j^d)_i^c \tag{2.11}$$

□

Mit Hilfe der Kofaktorbildung lassen sich für boolesche und pseudoboolesche Funktionen Zerlegungen in einfachere Funktionen angeben. Die im weiteren vorgestellten Entscheidungsdiagramme basieren auf der Idee, diese Zerlegung rekursiv fortzusetzen. Die hierbei verwendeten Zerlegungen sind in den Shannonschen Entwicklungssätzen beschrieben.

Satz 2.1.1 (Shannonscher Entwicklungssatz für boolesche Funktionen)
Sei $f : \mathbf{B}^n \to \mathbf{B}$ eine Funktion über der Variablenmenge $\chi = \{x_1, \ldots, x_n\} \subseteq \mathbf{B}^n$. Für alle $i \in \{1, \ldots, n\}$ gilt:

$$f = \begin{cases} \bar{x}_i f_i^0 + x_i f_i^1 & \textit{Shannon } (S_{\mathbf{B}}) \\ f_i^0 \oplus x_i f_i^2 & \textit{positiv Davio } (pD_{\mathbf{B}}) \\ f_i^1 \oplus \bar{x}_i f_i^2 & \textit{negativ Davio } (nD_{\mathbf{B}}) \end{cases} \tag{2.12}$$

□

Beweis: Durch Nachrechnen, ergibt sich $\forall x = (x_1, \ldots, x_n) \in \mathbf{B^n}$:

$S_{\mathbf{B}}$*:* Sei x fest und o.B.d.A gelte $a_i = 0, 1 \leq i \leq n$, so folgt:

$$\begin{aligned} f(x) &= \bar{x}_i \cdot f_i^0 + x_i \cdot f_i^1 \\ &= 1 \cdot f(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) + 0 \\ &= f(x) \end{aligned}$$

Für $a_i = 1$ ergibt sich eine analoge Betrachtung.

$pD_{\mathbf{B}}$: Es gilt $\bar{x}_i = x_i \oplus 1$, somit:

$$\begin{aligned} f(x) &= f_i^0 \oplus x_i \cdot f_i^2 \\ &= f_i^0 \oplus x_i \cdot \left(f_i^0 \oplus f_i^1\right) \\ &= (1 \oplus x_i) \cdot f_i^0 \oplus x_i \cdot f_i^1 \\ &= \bar{x}_i \cdot f_i^0 \oplus x_i \cdot f_i^1 \\ &= \bar{x}_i \cdot f_i^0 + x_i \cdot f_i^1 \end{aligned}$$

$nD_{\mathbf{B}}$: Es gilt $x_i = \bar{x}_i \oplus 1$, somit:

$$\begin{aligned} f(x) &= f_i^1 \oplus \bar{x}_i \cdot f_i^2 \\ &= f_i^1 \oplus \bar{x}_i \cdot \left(f_i^0 \oplus f_i^1\right) \\ &= (1 \oplus \bar{x}_i) \cdot f_i^1 \oplus \bar{x}_i f_i^0 \\ &= \bar{x}_i \cdot f_i^0 \oplus x_i \cdot f_i^1 \\ &= \bar{x}_i \cdot f_i^0 + x_i \cdot f_i^1 \end{aligned}$$

■

Satz 2.1.2 (Shannonscher Entwicklungssatz für ganzzahlige Funktionen) *Sei $f : \mathbf{B}^n \to \mathbf{Z}$ eine Funktion über der Variablenmenge $\chi = \{x_1, \ldots, x_n\} \subset \mathbf{B}^n$. Für alle $i \in \{1, \ldots, n\}$ gilt:*

$$f = \begin{cases} (1 - x_i) \cdot f_i^0 + x_i f_i^1 & \textit{Shannon } (S_{\mathbf{Z}}) \\ f_i^0 + x_i \cdot f_i^2 & \textit{positiv Davio } (pD_{\mathbf{Z}}) \\ f_i^1 + (1 - x_i) \cdot (-f_i^2) & \textit{negativ Davio } (nD_{\mathbf{Z}}) \end{cases} \tag{2.13}$$

□

Beweis: Analog zum Beweis von Satz 2.1.1. ■

Die Zerlegungen nach Satz 2.1.1 auf der vorherigen Seite und Satz 2.1.2 werden im folgenden als *Dekompositionstypen* bezeichnet. In [2] wurde gezeigt, dass für die Darstellung boolescher Funktionen (siehe 2.2.1 auf Seite 25) insgesamt nur 24 Dekompositionstypen möglich sind, die in drei Äquivalenzklassen zerfallen. Die vorgestellten Dekompositionstypen stellen jeweils einen Repräsentanten dieser Äquivalenzklassen dar. In der Literatur werden diese Repräsentanten oft als Kronecker-Dekompositionen bezeichnet.

Anmerkung 2.1.2 Durch Nachrechnen lässt sich zeigen, dass für Funktionen vom Typ $f : \mathbf{B}^n \to \mathbf{B}$ die Dekompositionen aus Satz 2.1.1 und Satz 2.1.2 äquivalent sind. Boolesche Funktionen und ihre Kofaktorzerlegungen lassen sich somit in natürlicher Weise in ganzzahligen Funktionen einbetten. □

2.2. Entscheidungsdiagramme

Definition 2.2.1 (DD) Ein Entscheidungsdiagramm (DD[3]) über einer Menge von Variablen $\chi := \{x_1, x_2, \ldots, x_n\} \subseteq \mathbf{B}^n$ ist ein gewurzelter, gerichteter, azyklischer und zusammenhängender Graph $G = (V, E, \mathit{index}, dt, \omega, T)$, mit folgenden Eigenschaften:

1. Der Wurzelknoten besitzt genau eine eingehende Kante.
2. Jeder Knoten $v \in V$ ist entweder ein terminaler Knoten oder ein nicht-terminaler Knoten.
3. Jeder terminale Knoten besitzt keine ausgehende Kanten und ist mit einem Element t der Terminalmenge T gekennzeichnet.
4. Jeder nicht-terminale Knoten $v \in V$ ist mit einer Variable $\mathit{index}(v) \in \chi$ markiert, also $\mathit{index} : V \to \chi$.
5. Jeder Variablen $x_i \in \chi$ des Entscheidungsdiagrammes ist ein *Dekompositionstyp* $dt(x_i) \in \{S_\mathbf{B}, pD_\mathbf{B}, nD_\mathbf{B}, S_\mathbf{Z}, pD_\mathbf{Z}, nD_\mathbf{Z}\}$ zugeordnet.
6. Jeder Kante $e \in E$ ist ein *Kantengewicht* $\omega(e)$ zugeordnet.
7. Jeder nicht-terminale Knoten $v \in V$ hat genau zwei ausgehende Kanten, deren Endpunkte mit $\mathit{low}(v), \mathit{high}(v) \in V$ bezeichnet werden. Die Kante zwischen v und $\mathit{low}(v)$ wird als *low*-Kante, die zwischen v und $\mathit{high}(v)$ als *high*-Kante bezeichnet.

□

Anmerkung 2.2.1 Meist gilt für die Terminalmenge $T \subseteq \mathbf{B}$ oder $T \subseteq \mathbf{Z}$. Weiter existieren Entscheidungsdiagramme, bei denen keine Kantengewichte ω ausgewertet werden. Üblicherweise werden diese dann auch nicht an die Kanten der Graphen geschrieben. Im Spezialfall, dass für alle Kanten $e \in E : \omega(e) \in \mathbf{B}$ gilt, werden die Kantengewichte als *Komplementmarken* bezeichnet. □

Definition 2.2.2 (Variablenordnung) Sei $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ eine beliebige Permutation auf der Menge $\{1, \ldots, n\}$. Als *Variablenordnung* des DDs G über χ wird die durch die Permutation π induzierte lineare Ordnung $<$ auf χ bezeichnet, die definiert wird durch $x_{\pi(i)} < x_{\pi(j)}$, genau dann wenn $i < j$ gilt. □

Definition 2.2.3 (Dekompositionstypliste) Als *Dekompositionstypliste* (DTL) des DDs G über χ wird die Liste der Dekompositionstypen $(dt(x_1), \ldots, dt(x_n))$ bezeichnet, die den Variablen x_i zugeordnet sind. □

[3] engl. Decision Diagram

Definition 2.2.4 (Eigenschaften von Entscheidungsdiagrammen) Sei G ein DD über χ mit der Terminalmenge T. Es gilt:

1. Als *Größe* oder *Kosten* $|G|$ von G wird die Anzahl innerer Knoten von G definiert.
2. Als *Level* x_i wird die Menge der Knoten von G bezeichnet, die mit der Variablen x_i markiert sind.
3. G ist *frei*, falls jede Variable x_i auf jedem Pfad von der Wurzel zu einem terminalen Knoten höchstens einmal vorkommt.
4. G ist *vollständig*, falls jede Variable x_i auf jedem Pfad von der Wurzel zu einem terminalen Knoten genau einmal vorkommt.
5. Ein freies DD G heißt *geordnet* falls auf jedem Pfad von der Wurzel zu einem Blatt jede Variable höchstens einmal, in der durch die Variablenordung $<$ vorgegebenen Reihenfolge $x_{\pi(1)}, \ldots, x_{\pi(n)}$, auftritt. Der Präfix „O“ (engl. ordered) bezeichnet geordnete Graphen, ein geordnetes DD wird dementsprechend als ODD bezeichnet.

□

2.2.1. Bit-Level Diagramme

Die Darstellung boolescher Funktionen als Bit-Level Diagramme soll eine effiziente Darstellung und Manipulation ermöglichen.

Definition 2.2.5 (BDD) Ein binäres Entscheidungsdiagramm (BDD[4]) ist ein DD über χ mit der Terminalmenge $T = \mathbf{B}$ und der festen DTL $d = (S_{\mathbf{B}}, \ldots, S_{\mathbf{B}})$. Bei BDDs werden keine Kantengewichte ausgewertet.

Die Funktion $f_G^d : \mathbf{B}^n \to \mathbf{B}$, die durch das BDD G über χ mit DTL d repräsentiert wird, ist wie folgt induktiv definiert:

1. Falls G nur aus einem mit 0 markierten (terminalen) Knoten besteht, dann ist G ein BDD für die konstante 0 Funktion.
2. Falls G nur aus einem mit 1 markierten (terminalen) Knoten besteht, dann ist G ein BDD für die konstante 1 Funktion.
3. Falls G die Wurzel v mit der Markierung x_i besitzt, dann ist G ein BDD für

$$\bar{x}_i f_{low(v)} + x_i f_{high(v)} \tag{2.14}$$

wobei $f_{low(v)}$ bzw. $f_{high(v)}$ die durch den BDD mit Wurzel $low(v)$ bzw. $high(v)$ repräsentierte Funktion ist. □

[4] engl. Binary Decision Diagram

Eine wesentlich allgemeinere Datenstruktur als BDDs stellen OKFDDs dar. Ihr Hauptmerkmal ist, dass alle binären Kronecker Dekompositionen zugelassen werden.

Definition 2.2.6 (OKFDD) Ein OKFDD[5] über χ ist ein geordnetes DD über χ mit der Terminalmenge $T = \mathbf{B}$ und einer festen DTL $d = (dt(x_1), \ldots, dt(x_n))$ mit $dt(x_i) \in \{S_\mathbf{B}, nD_\mathbf{B}, pD_\mathbf{B}\}, 0 < i \leq n$. Bei OKFDDs werden keine Kantengewichte ausgewertet.

Die Funktion $f_G^d : \mathbf{B}^n \to \mathbf{B}$, die durch das OKFDD G über χ mit DTL d repräsentiert wird, ist induktiv definiert:

1. Falls G nur aus einem mit 0 markierten (terminalen) Knoten besteht, dann ist G ein OKFDD für die konstante 0 Funktion.

2. Falls G nur aus einem mit 1 markierten (terminalen) Knoten besteht, dann ist G ein OKFDD für die konstante 1 Funktion.

3. Falls G die Wurzel v mit der Markierung x_i besitzt, dann ist G ein OKFDD für

$$\begin{cases} \bar{x}_i f_{low(v)} + x_i f_{high(v)} & \text{falls } dt(x_i) = S_\mathbf{B} \\ f_{low(v)} \oplus x_i f_{high(v)} & \text{falls } dt(x_i) = pD_\mathbf{B} \\ f_{low(v)} \oplus \bar{x}_i f_{high(v)} & \text{falls } dt(x_i) = nD_\mathbf{B} \end{cases} \tag{2.15}$$

□

In Abb. 2.1 ist ein OKFDD über $\chi = \{x_1, x_2, x_3, x_4\} \subseteq \mathbf{B}^4$ für die boolesche Funktion $f(x_1, x_2, x_3, x_4) = x_1x_2x_4 \oplus x_1x_2\bar{x}_3 \oplus x_1\bar{x}_3 \oplus \bar{x}_1x_2x_4$. Als Variablenordnung wurde $x_1 < x_2 < x_3 < x_4$ und als DTL $d = \{S_\mathbf{B}, pD_\mathbf{B}, nD_\mathbf{B}, S_\mathbf{B}\}$ gewählt.

Definition 2.2.7 (Reduktion von Bit-Level Diagrammen) Sei ein Bit-Level DD $G = (V, E, index, dt, \omega, T)$ ohne Kantenmarkierungen gegeben. Es werden folgende drei Reduktionstypen definiert:

Typ 1: Seien $v, w \in V$, $v \neq w$, es gelte $low(v) = low(w)$ und $high(v) = high(w)$.

Dann lenke alle Kanten, die auf w zeigen nach v um und lösche w und die beiden ausgehenden Kanten.

Typ 2: Sei $v \in V$ ein nicht-terminaler Knoten mit $dt\,(index(v)) = S_\mathbf{B}$, es gelte $low(v) = high(v)$.

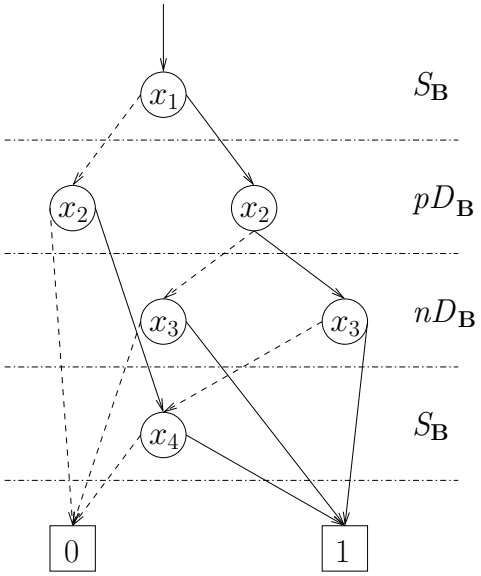
Dann lenke alle Kanten, die auf v zeigen nach $low(v)$ um und lösche v und die beiden ausgehenden Kanten.

Typ 3: Sei $v \in V$ ein nicht-terminaler Knoten mit $dt(index(v)) \in \{pD_\mathbf{B}, nD_\mathbf{B}\}$. Es gelte, daß die *high*-Kante von v die konstante 0-Funktion repräsentiert.

Dann lenke alle Kanten, die auf v zeigen nach $low(v)$ um und lösche v und die beiden ausgehenden Kanten.

[5] engl. Ordered Kronecker Funktional Decision Diagram

Abbildung 2.1. OKFDD der Funktion $x_1x_2x_4 \oplus x_1x_2\bar{x}_3 \oplus x_1\bar{x}_3 \oplus \bar{x}_1x_2x_4$



Reduktion vom Typ 1 werden auch Isomorphie-Reduktionen, Reduktionen vom Typ 2 Shannon-Reduktionen und Reduktionen vom Typ 3 Davio-Reduktionen genannt. □

Definition 2.2.8 (Reduzierte Bit-Level Entscheidungsdiagramme) Ein binäres DD heißt reduziert, wenn keine der Reduktionen aus Definition 2.2.7 mehr angewendet werden können. Reduzierte Entscheidungsdiagramme werden mit dem Präfix „R“ (engl. reduced) versehen. □

Satz 2.2.1 (Kanonitizät von OKFDDs)
Für eine feste Variablenordnung und feste DTL ist ein reduziertes OKFDD eine kanonische, d.h. eindeutige, Darstellung einer booleschen Funktion. □

Beweis: Für den Spezialfall der OBDDs wird der Beweis in [5] ausgeführt. Für OKFDDs kann der Beweis in [10] nachgelesen werden. ■

Anmerkung 2.2.2 Durch die Verwendung von Komplementkanten ($\omega : V \rightarrow \mathbf{B}$) (siehe z.B. [11]) kann die Darstellung noch kompakter gestaltet werden. Hierbei wird jeder Kante ein zusätzliches Bit zugeordnet. Ist dieses Bit gesetzt, so ist die am Knoten repräsentierte Funktion zu komplementieren. Es werden also von den Knoten und Kanten

Tabelle 2.1. Operationen auf Bit-Level DDs ohne Komplementkanten

Graph	$\neg G$	$F \wedge G$	$F \vee G$	$F \oplus G$	ITE(F, G, H)
OBDD	$O(\|G\|)$	$O(\|F\| \cdot \|G\|)$	$O(\|F\| \cdot \|G\|)$	$O(\|F\| \cdot \|G\|)$	$O(\|F\| \cdot \|G\| \cdot \|H\|)$
OKFDD	$O(\|G\|)$	$O(2^{\max(\|F\|,\|G\|)})^a$	$O(2^{\max(\|F\|,\|G\|)})^a$	$O(\|F\| \cdot \|G\|)$	$O(2^{\max(\|F\|,\|G\|,\|H\|)})^a$

[a] Bei konstant vielen Davio-Leveln, polynomiell.

Funktionen dargestellt. Es ist leicht einzusehen, dass für binäre DDs ein Terminalknoten ausreicht. Durch die Forderung, dass die *low*-Kanten[6] nicht komplementiert werden dürfen, bleibt die Kanonizität der Darstellung erhalten. □

Bei der Wahl der Variablenordnung und der DTL ist zu beachten, dass sich durch die Wahl einer „ungeschickten“ Variablenordnung (oder DTL) exponentielle Größenunterschiede ergeben können. Es lassen sich auch Funktionen angeben, die nur exponentiell große OBDD Darstellungen besitzen, als OKFDD aber effizient darstellbar sind. Eine zusammenfassende Darstellung über diese Problematik findet sich in [11]. Diesem Werk ist auch Tabelle 2.1 entnommen, in der das worst case Größenwachstum einiger boolescher Operationen auf OBDDs und OKFDDs gegenübergestellt wird.

2.2.2. Word-Level Diagramme

Word-Level Diagramme sind eine Erweiterung der Bit-Level-Diagramme, die der effizienten Darstellung und Manipulation pseudoboolescher Funktionen dienen.

Die einfachste Erweiterung der Bit-Level-Diagramme stellen MTBDDs dar, eine Art OBDD bei dem eine ganzzahlige Terminalmenge T zugelassen wird.

Definition 2.2.9 (MTBDD) Ein MTBDD[7] ist ein geordnetes DD über χ mit der Terminalmenge $T = \mathbf{Z}$ und der festen DTL $d = (S_{\mathbf{Z}}, \ldots, S_{\mathbf{Z}})$. Bei MTBDDs werden keine Kantengewichte ausgewertet.

Die Funktion $f_G^d : \mathbf{B}^n \to \mathbf{Z}$, die durch das MTBDD G über χ mit der DTL d repräsentiert wird, ist induktiv wie folgt definiert:

1. Falls G nur aus einem mit $t \in T$ markierten Knoten besteht, dann ist G ein MTBDD für die konstante t-Funktion.

2. Falls G die Wurzel v mit $index(v) = x_i$ besitzt, dann ist G ein MTBDD für

$$(1 - x_i) \cdot f_{low(v)} + x_i \cdot f_{high(v)} \tag{2.16}$$

□

[6] Die Wahl ist bei OBDDs beliebig, die Forderung, dass die *high*-Kanten nicht komplementiert werden, erhält in diesem Fall gleichfalls die Kanonizität.

[7] engl. Multi-Terminal Binary Decision Diagram

Da MTBDDs ohne Kantengewichte realisiert sind, lassen sich die booleschen Reduktionen aus Definition 2.2.7 auf Seite 26 leicht auf MTBDDs übertragen.

Definition 2.2.10 (Kantengewichte) Bei Word-Level Diagrammen werden additive und multiplikative Kantengewichte unterschieden:

additiv: Zu der am Knoten repräsentierten Funktion wird das Kantengewicht addiert.

multiplikativ: Die am Knoten repräsentierte Funktion wird mit dem Kantengewicht multipliziert.

Zudem besteht die Möglichkeit, Kanten mit einem Tupel (a, m) zu gewichten. Dabei wird a als additives und m als multiplikatives Gewicht interpretiert. In diesem Fall wird zuerst die am Knoten dargestellte Funktion mit m multipliziert und dieser Wert anschließend zu a addiert. □

Um arithmetische Funktionen über booleschen Variablen, insbesondere die Multiplikation, effizient darstellen zu können, wurden in [7, 8] ∗BMDs eingeführt.

Definition 2.2.11 (∗BMD) Ein ∗BMD[8] ist ein geordnetes DD über χ mit einer festen DTL $d = (pD_{\mathbf{Z}}, \ldots, pD_{\mathbf{Z}})$, der Terminalmenge $T = \{1\}$ und einer Gewichtsfunktion $\omega : V \to \mathbf{Z}$, die jeder Kante ein multiplikatives Gewicht $m \in \mathbf{Z}$ zuordnet.

Die Funktion $f_G^d : \mathbf{B}^n \to \mathbf{Z}$, die durch das ∗BMD G über χ mit der DTL d und der Gewichtsfunktion ω repräsentiert wird, ist induktiv definiert durch:

1. Falls G nur aus einem mit 1 gekennzeichneten terminalen Knoten besteht, auf den die Kante e mit $\omega(e) = m$ zeigt, dann ist G eine Repräsentation für die konstante Funktion mit dem Wert m.

2. Falls G die Wurzel $v \in V$ mit der Markierung $index(v) = x_i$ und der eingehenden Kante e mit $\omega(e) = m$ besitzt, dann ist G ein ∗BMD für

$$\langle m, f \rangle = m \cdot \left(f_{low(v)} + x_i f_{high(v)} \right) \tag{2.17}$$

□

Definition 2.2.12 (Normierung von ∗BMDs) Ein ∗BMD über χ heißt normiert, falls für die Kantengewichte an allen Knoten $v \in V$ mit der eingehenden Kante e und den ausgehenden Kanten e_{low} und e_{high} gilt[9]:

$$\gcd(\omega(e_{low}), \omega(e_{high})) = 1 \wedge ((\omega(e_{low}) > 0) \vee ((\omega(e_{low}) = 0) \wedge (\omega(e_{high}) \geq 0))) \tag{2.18}$$

□

[8] engl. Multiplicative Binary Moment Diagram

[9] Wobei die Funktion gcd() die Funktion des größten gemeinsamen Teilers repräsentiert.

Eine allgemeinere Word-Level Datenstruktur bilden die K∗BMDs, bei denen ähnlich wie bei den OKFDDs bei den Bit-Level Datenstrukturen, alle ganzzahligen Kronecker Dekomposition zugelassen werden.

Definition 2.2.13 (K∗BMD) Ein K∗BMD[10] ist ein geordnetes DD über χ mit der Terminalmenge $T = \{0\}$ und einer festen DTL $d = (dt(x_1), \ldots, dt(x_n))$ wobei $dt(x_i) \in \{S_{\mathbf{Z}}, pD_{\mathbf{Z}}, nD_{\mathbf{Z}}\}$, $0 < i \leq n$ und $\omega : V \to \mathbf{Z} \times \mathbf{Z}$ eine Gewichtsfunktion, die jeder Kante ein additives Gewicht $a \in \mathbf{Z}$ und ein multiplikatives Gewicht $m \in \mathbf{Z}$ zuordnet.

Die Funktion $f_G^d : \mathbf{B}^n \to \mathbf{Z}$, die durch das K∗BMD G über χ mit DTL d und der Kantengewichtsfunktion ω repräsentiert wird, ist induktiv definiert durch:

1. Falls G nur aus einem mit 0 gekennzeichneten (terminalen) Knoten und der Kante e mit $\omega(e) = (a, m)$ besteht, die auf diesen Knoten zeigt, dann ist G eine Repräsentation für die konstante Funktion mit dem Wert a.

2. Falls G die Wurzel $v \in V$ mit der Markierung $index(v) = x_i$ und der eingehenden Kante e mit $\omega(e) = (a, m)$ besitzt, dann ist G ein K∗BMD für

$$\langle (a, m), f \rangle = \begin{cases} a + m \cdot \big((1 - x_i) \cdot f_{low(v)} + x_i \cdot f_{high(v)}\big) & \text{falls } dt(x_i) = S_{\mathbf{Z}} \\ a + m \cdot \big(f_{low(v)} + x_i \cdot f_{high(v)}\big) & \text{falls } dt(x_i) = pD_{\mathbf{Z}} \\ a + m \cdot \big(f_{low(v)} + (1 - x_i) \cdot f_{high(v)}\big) & \text{falls } dt(x_i) = nD_{\mathbf{Z}} \end{cases} \tag{2.19}$$

□

Definition 2.2.14 (Normierung von K∗BMDs) Ein K∗BMD über χ heißt normiert, falls für die Kantengewichte an allen Knoten $v \in V$ mit der eingehenden Kante e mit $\omega(e) = (a, m)$ und den ausgehenden Kanten e_{low} und e_{high} mit $\omega(e_{low}) = (a_{low}, m_{low})$ sowie $\omega(e_{high}) = (a_{high}, m_{high})$, sowie $s(e) \in V \cup T$ der Knoten, auf den e zeigt, gilt:

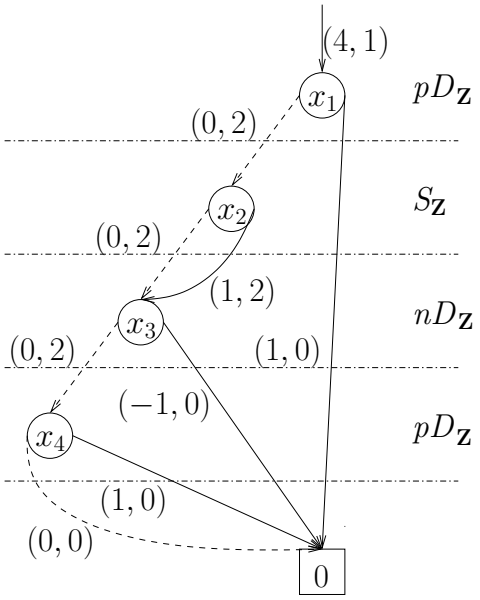
$$\begin{aligned} & \gcd(m_{low}, a_{high}, m_{high}) = 1 \\ \wedge\ & a_{low} = 0 \\ \wedge\ & (m_{low} > 0 \vee (m_{low} = 0 \wedge a_{high} > 0) \wedge (m_{low} = 0 \wedge a_{high} = 0 \wedge m_{high} \geq 0)) \\ \wedge\ & (m_{low} = 0 \Leftrightarrow s(e_{low}) \text{ terminaler Knoten}) \\ \wedge\ & (m_{high} = 0 \Leftrightarrow s(e_{high}) \text{ terminaler Knoten}) \end{aligned}$$

□

Als Beispiel für einen K∗BMD über $\chi = \{x_1, x_2, x_3, x_4\} \subseteq \mathbf{B}^4$ ist in Abb. 2.2 die Kodierung natürlicher Zahlen, also die Funktion $f(x_1, x_2, x_3, x_4) = \sum_{i=0}^{3} 2^i \cdot x_{i+1}$, dargestellt. Als Variablenordnung wurde $x_1 < x_2 < x_3 < x_4$ und als DTL $d = (pD_{\mathbf{Z}}, S_{\mathbf{Z}}, nD_{\mathbf{Z}}, pD_{\mathbf{Z}})$ gewählt.

[10] engl. Kronecker Multiplikative Binary Moment Diagram

Abbildung 2.2. Beispiel für ein K∗BMD



Anmerkung 2.2.3 Die in Definition 2.2.12 auf Seite 29 und 2.2.14 geforderten Normierungen für ∗BMDs und K∗BMDs lassen sich aus einem nicht normierten DD durch bottom-up Traversierung berechnen, siehe hierzu [21]. □

Auch bei Word-Level Datenstrukturen werden Reduktionen angewendet, um eine effiziente Darstellung zu erreichen. Es ergeben sich dabei dieselben Reduktionstypen wie bei den Bit-Level Darstellungen (siehe 2.2.7 auf Seite 26), wobei im Word-Level Fall die Kantengewichte besonders berücksichtigt werden müssen.

Definition 2.2.15 (Reduktion von Word-Level-Diagrammen) Sei ein Word-Level DD $G = (V, E, index, dt, \omega, T)$ mit Kantengewichten gegeben. Es werden folgende drei Reduktionstypen definiert:

Typ 1: Seien $v, w \in V$, $v \neq w$, nicht-terminale Knoten, sowie $e_{low(v)}$ die *low*-Kante von v (entsprechend $e_{high(v)}, e_{low(w)}, e_{high(w)}$). Es gelte:

$$\begin{aligned} index(v) &= index(w) \\ low(v) &= low(w) \\ high(v) &= high(w) \\ \omega(e_{low(v)}) &= \omega(e_{low(w)}) \\ \omega(e_{high(v)}) &= \omega(e_{high(w)}) \end{aligned}$$

Dann lenke alle Kanten, die auf w zeigen nach v um und berechne entsprechend der Semantik neue Kantengewichte. Anschließend lösche w und die beiden ausgehenden Kanten.

Typ 2: Sei $v \in V$ ein nicht-terminaler Knoten mit $dt\,(index(v)) \in \{S_{\mathbf{B}}, S_{\mathbf{Z}}\}$. Sei e_{high} die *high*-Kante und e_{low} die *low*-Kante von v. Es gelte $low(v) = high(v)$ und $\omega(e_{low}) = \omega(e_{high})$.

Dann lenke alle Kanten, die auf v zeigen nach $low(v)$ um und berechne entsprechend der Semantik neue Kantengewichte. Anschließend lösche v und die beiden ausgehenden Kanten.

Typ 3: Sei $v \in V$ ein nicht-terminaler Knoten, $dt(index(v)) \in \{pD_{\mathbf{Z}}, nD_{\mathbf{Z}}\}$. Es gelte, daß die *low*-Kante von v auf einen terminalen Knoten zeigt, der die konstante 0-Funktion repräsentiert.

Dann lenke alle Kanten, die auf v zeigen nach $low(v)$ um und berechne entsprechend der Semantik neue Kantengewichte. Anschließend lösche v und die beiden ausgehenden Kanten.

Reduktionen vom Typ 1 werden auch Isomorphie-Reduktionen, Reduktionen vom Typ 2 Shannon-Reduktionen und Reduktionen vom Typ 3 Davio-Reduktionen genannt. □

Definition 2.2.16 (Reduzierte Word-Level DDs) Ein Word-Level-DD heißt reduziert, wenn keine der Reduktionen aus Definition 2.2.15 mehr angewendet werden können. Reduzierte Entscheidungsdiagramme werden mit dem Präfix „R“ (engl. reduced) versehen. □

Es ist zu beachten, dass durch die Reduktion nicht normierte DDs enstehen können. Nach dem Anwenden von Reduktionen ist eine Normierung notwendig. Genauso kann durch das Normieren ein nicht reduzierter DD enstehen. Ein DD heißt reduziert und normiert, wenn keine Normierungsregeln und Reduktionen angewendet werden können.

Satz 2.2.2 (Kanonizität von Word-Level DDs)
Für eine feste Variablenordnung und DTL sind reduzierte und normierte ∗BMDs und K∗BMDs eine kanonische Darstellung pseudoboolescher Funktionen. □

Beweis: Für ∗BMDs siehe [7], eine Betrachtung für K∗BMDs kann in [12] nachgelesen werden. ■

Erstmals vorgestellt wurden K∗BMDs in [12], einen guten Überblick bietet auch [3]. Eine [21] entnommene Gegenüberstellung der Darstellungsgröße einiger Wortoperationen ist in Tabelle 2.2 wiedergegeben. Weitere Algorithmen für ∗BMDs finden sich in [15] und für K∗BMDs werden verschiedenen Algorithmen in [13, 14] vorgestellt.

Tabelle 2.2. Komplexität von Wortoperationen in Abhängigkeit der Wortbreite n

Graphtyp	X	$X+Y$	$X \cdot Y$	X^2	c^X
MTBDD	$O(2^n)$	$O(2^n)$	$O(4^n)$	$O(2^n)$	$O(2^n)$
*BMD	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$
K*BMD	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Anmerkung 2.2.4 Im weiteren werden nur ganzzahlige Kantengewichte betrachtet. Zwar lassen sich rationale Kantengewichte analog einführen, diese erschweren aber zum einen die theoretische Betrachtung und zum anderen werden sie von den verwendeten Word-Level DD-Paketen [19] und [23] nicht unterstützt. □

2.3. Zahlendarstellungen und Operationen auf Bitvektoren

Im folgenden werden kurz die wichtigsten Eigenschaften von Bit-Vektoren eingeführt, sowie erläutert wie Bit-Vektoren als Zahlen aufgefasst werden können.

2.3.1. Grundlegende Definitionen

Definition 2.3.1 Als Ordnung eines Bitvektors $a \in \mathbf{B}^n$ wird seine Länge definiert:

$$||a|| = ||(a_{n-1}, \ldots, a_0)|| = n \tag{2.20}$$

□

Definition 2.3.2 (Projektion) Als Projektion p der Stellen i bis j mit $0 \leq i \leq j < n$ eines Bitvektors $a \in \mathbf{B}^n$ wird definiert:

$$p_i^j\left((a_{n-1}, \ldots, a_j, \ldots, a_i, \ldots, a_0)\right) = (a_j, \ldots, a_i) \tag{2.21}$$

Die Projektion ist eine Funktion vom Typ $p_i^j : \mathbf{B}^n \rightarrow \mathbf{B}^{j-i+1}$ □

2.3.2. Zahlendarstellungen

Computer stellen Zahlen als Bitvektoren einer festen Länge dar. Um mit den bekannten Zahlenräumen ($\mathbf{N}, \mathbf{Z}, \mathbf{Q}$ oder $\mathbf{R}$) arbeiten zu können, sind geeignete Kodierungen notwendig, die in diesem Abschnitt für Teilmengen von $\mathbf{N}, \mathbf{Z}$ und $\mathbf{Q}$ vorgestellt werden.

2.3.2.1. Darstellung ganzer Zahlen

Definition 2.3.3 (Binärzahlen) Die übliche Interpretation eines Bitvektors als natürliche Zahl $\langle a \rangle \in \mathbf{N}$ wird durch folgende Funktion beschrieben:

$$\langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i \tag{2.22}$$

Es gilt: $\langle a \rangle \in \{0, \ldots, 2^n - 1\}$ □

Um den Zahlenbereich $\mathbf{Z}$ zu beschreiben, müssen zusätzlich negative Zahlen dargestellt werden. Zur Darstellung negativer Zahlen bieten sich drei Möglichkeiten an: Betrag und Vorzeichen, Einer-Komplement und die Darstellung im Zweier-Komplement.

Definition 2.3.4 (Betrag-Vorzeichen)

$$a = (-1)^{a_n} \cdot \langle a_{n-1}, \ldots, a_0 \rangle$$

Es gilt: $a \in \{-2^{n-1} - 1, \ldots, 2^{n-1} - 1\}$ □

Definition 2.3.5 (Einer-Komplement)

$$a = -a_n \cdot (2^n - 1) + \langle a_{n-1}, \ldots, a_0 \rangle$$

Es gilt: $a \in \{-2^n - 1, \ldots, 2^n - 1\}$ □

Definition 2.3.6 (Zweier-Komplement)

$$[a] = [a_n, \ldots, a_0] = -a_n \cdot 2^n + \langle a_{n-1}, \ldots, a_0 \rangle$$

Es gilt: $[a] \in \{-2^n, \ldots, 2^n - 1\}$ □

2.3.2.2. Darstellung von Festkommazahlen

Die Darstellung ganzer Zahlen lässt sich leicht erweitern, so dass Dualbrüche dargestellt werden können. Da in der Regel eine feste Anzahl von Nachkommastellen verwendet wird, werden diese Zahlen als Festkommazahlen bezeichnet.

Definition 2.3.7 (Festkommazahl) Die Darstellung eines Dezimalbruchs mit n Vorkomma und m Nachkommastellen in der Form

$$a = \sum_{i=-m}^{n} a_i \cdot 2^i$$

wird als Festkomma-Darstellung bezeichnet. □

Es gilt: $a \in \{2^{-m}, 2^{-m+1}, \ldots, 0, \ldots, 2^{n+1} - 2^{-m}\}$

Negative Festkommazahlen können entsprechend als Betrag-Vorzeichen, Einer-Komplement oder Zweier-Komplement kodiert werden. Vorzeichenbehaftete Festkommazahlen repräsentieren eine Teilmenge von $\mathbf{Q}$.

Anmerkung 2.3.1 Da die Kodierung der Festkommazahlen eine einfache Erweiterung der Darstellung ganzer Zahlen darstellt, können Schaltkreise, die auf natürlichen Zahlen arbeiten ohne Modifikation für die Berechnung von Festkommazahlen gleicher Bitbreite verwendet werden und umgekehrt. Hierzu muss lediglich die Interpretation des Bitvektors angepasst werden. □

2.3.2.3. Signed-Digit Zahlensysteme

In den bisher vorgestellten binären Zahlendarstellungen waren nur die Ziffern $\{0, 1\}$ zugelassen. Stattdessen kann aber auch folgende Ziffermenge für x_i zugelassen werden:

$$x_i \in \{\bar{1}, 0, 1\} \tag{2.23}$$

Wobei $\bar{1}$ gleich -1 ist. Da jede Ziffer bereits positiv oder negativ ist, wird kein zusätzliches Vorzeichenbit benötigt. Im Allgemeinen sind Signed-Digit Darstellungen redundant, d.h. eine Zahl hat mehrere Signed-Digit Darstellungen.

Definition 2.3.8 Sei die Ziffermenge

$$\Xi = \left\{\overline{(r-1)}, \overline{(r-2)}, \ldots, \bar{1}, 0, 1, \ldots, (r-2), (r-1)\right\} \quad \text{mit} \quad r \in \mathbf{N} \tag{2.24}$$

gegeben, wobei $\bar{x}_i = -x_i, x_i \in \Xi$ gilt. Die Darstellung eines Dezimalbruches mit n Vorkomma und m Nachkommastellen in der Form

$$a = \sum_{i=-m}^{n} x_i \cdot 2^i \quad \text{mit} \quad x_i \in \Xi \tag{2.25}$$

wird als *Signed-Digit Zahl* bezeichnet. □

Die Redundanzen, die bei der Darstellung ganzer Zahlen als Signed-Digit Zahlen auftreten (siehe auch Beispiel 2.3.1), können genutzt werden, um Schaltkreise zu optimieren[11]. Ein Schaltkreis, bei dem diese Zahlendarstellung zur Optimierung ausgenutzt wird, ist der in Abschnitt 3.2.1 auf Seite 49 vorgestellte SRT Dividierer.

[11] Je nach Anwendung wird hierbei die Tiefe (Laufzeit), Anzahl der Gatter (Fläche) oder eine Kombination hiervon optimiert.

Beispiel 2.3.1 (SD-Zahlen) Sei $(\bar{1}01)_2$ eine binäre Signed-Digit Darstellung einer ganzen Zahl. Es ergibt sich:

$$\begin{aligned}(10\bar{1})_2 &= 1 \cdot 2^2 + 0 \cdot 2^1 + (-1) \cdot 2^0 \\ &= (3)_{10} \\ &= 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ &= (011)_2\end{aligned}$$

2.3.3. Addition von Zweier-Komplement-Zahlen

Eine grundlegende Operation, die auch bei der Division Verwendung findet, ist die Addition. In diesem Zusammenhang, wird zuerst der Begriff der Sign-Extension eingeführt. Damit wird die Eigenschaft von Zahlen in Zweierkomplement-Darstellung bezeichnet, dass durch Verdoppeln des höchstwertigen Bits[12] die Interpretation (der dargestellte Zahlenwert) nicht geändert wird. Durch „iteriertes“ Anwenden dieses Sachverhaltes können damit Zahlen im Zweierkomplement auf eine beliebige Bitbreite erweitert werden ohne ihren Zahlenwert zu ändern.

Satz 2.3.1 (Sign-Extension)
Sei $a = (a_n, \ldots, a_0) \in \mathbf{B}^{n+1}$*, so gilt:*

$$[a_n, a_n, \ldots, a_0] = [a]$$

□

Beweis:

$$\begin{aligned}[a_n, a_n, \ldots, a_0] &= -a_n \cdot 2^{n+1} + \langle a_n, \ldots, a_0 \rangle \\ &= -a_n \cdot 2^{n+1} + \sum_{i=0}^{n} a_i \cdot 2^i \\ &= -a_n \cdot 2^n + \langle a_{n-1}, \ldots, a_0 \rangle \\ &= [a]\end{aligned}$$

Definition 2.3.9 (Überträge) Das Bit $c_i \in \mathbf{B}$ bezeichne den bei der Addition entstehenden Übertrag von Stelle i zu Stelle $i + 1$. □

Eine interessante Eigenschaft von Zahlen in Zweierkomplement-Darstellung ist, dass sich der bekannte Algorithmus zur Addition von natürlichen Zahlen direkt auf sie anwenden lässt.

[12] also dem Vorzeichenbit

Satz 2.3.2 (Addition von Zweier-Komplement-Zahlen)
Seien $a, b \in \mathbf{B}^{n+1}$, $c \in \mathbf{B}$ und $s \in \mathbf{B}^{n+1}$, so dass $\langle c_n, s\rangle = \langle a\rangle + \langle b\rangle + c$. Es ist

$$[a] + [b] + c \in \{-2^n, \ldots, 2^n - 1\} \quad \textit{genau dann wenn} \quad c_n = c_{n-1}.$$

In diesem Fall ist $[a] + [b] + c = [s]$. □

Beweis: Seien a_n und b_n die Vorzeichenbits der Operanden a und b. Weiter sei

$$\begin{aligned} a' &:= (a_{n-1}, \ldots, a_0) \\ b' &:= (b_{n-1}, \ldots, b_0) \end{aligned}$$

Somit kann die Addition $[a] + [b] + c$ getrennt betrachtet werden für die Gruppen von Bits:

$$\begin{aligned} \langle a'\rangle + \langle b'\rangle + c &= \langle a_{n-1}, \ldots, a_0\rangle + \langle b_{n-1}, \ldots, b_0\rangle + c \\ &= \langle c_{n-1}, s_{n-1}, \ldots, s_0\rangle \\ a_n + b_n + c_{n-1} &= \langle c_n, s_n\rangle \end{aligned}$$

Fall 1: $a_n = b_n = 0$
In diesem Fall sind $[a]$ und $[b]$ positiv, deswegen gilt $c_n = 0$.

$$\begin{aligned} c_{n-1} = 0 &\Leftrightarrow \langle a'\rangle + \langle b'\rangle + c \leq 2^n - 1 \\ &\Leftrightarrow [a] + [b] + c \leq 2^n - 1 \end{aligned}$$

Weiter gilt:

$$\begin{aligned} [a] + [b] + c &= \langle a'\rangle + \langle b'\rangle + c \\ &= \langle s_{n-1}, \ldots, s_0\rangle \\ &= [s] \quad (\text{da } s_n = 0) \end{aligned}$$

Fall 2: $a_n = b_n = 1$
Also sind $[a]$ und $[b]$ negativ, deswegen ist $c_n = 1$ und es gilt:

$$\begin{aligned} c_{n-1} = 1 &\Leftrightarrow \langle a'\rangle + \langle b'\rangle + c \geq 2^n \\ &\Leftrightarrow [a] + [b] + c = -2^{n+1} + \langle a'\rangle + \langle b'\rangle + c \geq -2^n \end{aligned}$$

Weiter gilt:

$$\begin{aligned} [a] + [b] + c &= -2^{n+1} + \langle a'\rangle + \langle b'\rangle + c \\ &= -2^{n+1} + \langle 1, s_{n-1}, \ldots, s_0\rangle \\ &= -2^n + \langle s_{n-1}, \ldots, s_0\rangle \\ &= [s] \ (\text{da } s_n = 1) \end{aligned}$$

Fall 3: $a_n \neq b_n$

O.B.d.A. gelte $a_n = 0$ und $b_n = 1$. Die Stelle n propagiert also den Übertrag von der Stelle $n-1$. Also gilt $c_n = c_{n-1}$.
Wegen

$$0 \leq \langle a' \rangle + \langle b' \rangle + c \leq 2^{n+1} - 1$$

tritt keine Bereichsüberschreitung auf. Weiter ist

$$\begin{aligned} [a] + [b] + c &= -2^n + \langle a' \rangle + \langle b' \rangle + c \\ &= -2^n + \langle c_{n-1}, s_{n-1}, \ldots, s_0 \rangle \\ &= -(1 - c_{n-1}) \cdot 2^n + \langle s_{n-1}, \ldots, s_0 \rangle \\ &= [\overline{c_{n-1}}, s_{n-1}, \ldots, s_0] \end{aligned}$$

und

$$\begin{aligned} s_n &= a_n \oplus b_n \oplus c_{n-1} \\ &= 1 \oplus c_{n-1} \\ &= \overline{c_{n-1}} \end{aligned}$$

Also gilt

$$[s] = [\overline{c_{n-1}}, s_{n-1}, \ldots, s_0]$$

■

Die Addition von Zweierkomplementzahlen kann also auf die Addition von Binärzahlen zurückgeführt werden.

Bei der Addition von natürlichen Zahlen kann die Bereichsüberschreitung bei der Addition leicht umgangen werden, indem durch Hinzunahme von c_n zum Ergebnis, also für $a, b \in \mathbf{B}^n$ und $s \in \mathbf{B}^{n+1}$ etwa $\langle a \rangle + \langle b \rangle = \langle c_n, a_{n-1}, \ldots, a_0 \rangle$, die Addition als Funktion $\mathbf{B}^n \times \mathbf{B}^n \to \mathbf{B}^{n+1}$ definiert wird. Wie dies bei Zahlen in Zweierkomplement-Darstellung erreicht wird, zeigt der folgende Satz.

Satz 2.3.3
Seien $a, b \in \mathbf{B}^{n+1}, c \in \mathbf{B}$ *und* $s \in \mathbf{B}^{n+2}$, *so dass* $\langle c_n, s_n, \ldots, s_0 \rangle = \langle a \rangle + \langle b \rangle + c$ *und*

$$s_{n+1} = \begin{cases} s_n & \text{wenn } c_n = c_{n-1} \\ c_n & \text{wenn } c_n \neq c_{n-1}. \end{cases}$$

In diesem Fall gilt: $[a] + [b] + c \in \{-2^{n+1}, \ldots, 2^{n+1} - 1\}$ *und somit:* $[a] + [b] + c = [s]$. □

BEWEIS: Für $c_n = c_{n-1}$ folgt die Aussage direkt aus Satz 2.3.1 und Satz 2.3.2. Die Aussage $[a] + [b] + c \in \{-2^{n+1}, \ldots, 2^{n+1} - 1\}$ folgt direkt, so dass nur die Behauptung $[a] + [b] + c = [s]$ nachgerechnet werden muss.

Fall 1: $a_n = b_n = 0$
In diesem Fall sind $[a]$ und $[b]$ positiv, deswegen gilt $c_n = 0$.

$$\begin{aligned} [a] + [b] + c &= \langle a' \rangle + \langle b' \rangle + c \\ &= \langle c_n, s_n, \ldots, s_0 \rangle \\ &= [s] \end{aligned}$$

Fall 2: $a_n = b_n = 1$
In diesem Fall sind $[a]$ und $[b]$ negativ, deswegen ist $c_n = 1$ und es gilt:

$$\begin{aligned} [a] + [b] + c &= -2^{n+1} + \langle a' \rangle + \langle b' \rangle + c \\ &= -2^{n+1} + \langle s_n, \ldots, s_0 \rangle \\ &= [s] \end{aligned}$$

Fall 3: $a_n \neq b_n$
In diesem Fall gilt immer $c_n = c_{n-1}$. ■

2.4. Auswahl von Leitungen eines Datenwortes

Beim Entwurf konkreter Implementierungen kommt es häufiger vor, dass Leitungen eines Datenbusses „weggelassen“ werden. Wird der Datenbus als Bitvektor aufgefasst, kann das Weglassen von Leitungen durch die Projektion beschrieben werden. Satz 2.4.1 beschreibt, wie sich im Falle der Zweier-Komplement Darstellung diese Projektion auf die dargestellte Zahl auswirkt.

Satz 2.4.1
Sei $a \in \mathbf{B}^{n+1}$. *Es gilt:*

$$[a_{i-1}, \ldots, a_0] \equiv [a] \pmod{2^i} \quad \textit{mit} \quad 1 \leq i \leq n \qquad \square$$

BEWEIS: Nach der Definition der Modular-Arithmetik genügt es zu zeigen, dass $2^i \mid ([a] - [a_{i-1}, \ldots, a_0])$ gilt. Betrachte hierzu die Differenz $[a] - [a_{i-1}, \ldots, a_0]$:

$$[a] - [a_{i-1}, \ldots, a_0] = \left(-2^n \cdot a_n + \sum_{j=0}^{n-1} 2^j \cdot a_j \right) - \left(-2^{i-1} \cdot a_{i-1} + \sum_{j=0}^{i-2} 2^j \cdot a_j \right) .$$

Aufgrund $1 \leq i \leq n$ lassen sich die Summen vereinfachen zu:

$$= -2^n \cdot a_n + 2^{i-1} \cdot a_{i-1} + \sum_{j=i-1}^{n-1} 2^j \cdot a_j$$
$$= -2^n \cdot a_n + \sum_{j=i}^{n-1} 2^j \cdot a_j \; .$$

Ein Ausmultiplizieren von 2^i ergibt wegen $i \leq n$:

$$= 2^i \cdot \left(-2^{n-i} + \sum_{j=i}^{n-1} 2^{j-i} \cdot a_j \right) \; .$$

■

Korollar 2.4.1
Sei $a \in \mathbf{B}^{n+1}$, so gilt:

$$\left[p_0^{i-1}(a)\right] \equiv [a] \pmod{2^i} \quad \textit{mit} \quad i \leq n$$

□

3. Algorithmen zur Berechnung der Division

Naiv betrachtet scheint die Division die schwierigste der vier Grundrechenarten zu sein. Im Gegensatz zu den anderen drei Operationen besteht das Ergebnis der Division aus zwei Komponenten, dem Quotienten und dem Rest.

Sei ein Dividend $X \in \mathbf{Z}$ und ein Divisor $D \in \mathbf{Z}$ gegeben, so ist sowohl ein Quotient $Q \in \mathbf{Z}$ als auch ein Rest $R \in \mathbf{Z}$ gesucht, für welche die Gleichung

$$X = Q \cdot D + R \quad \text{mit} \quad 0 \leq R < D \tag{3.1}$$

gilt. Oft wird die Division aufgeteilt in eine Operation $\div$ zur Berechnung des Quotienten und einer Operation mod zur Berechnung des Restes, so dass gilt:

$$X \div D = Q \quad \text{und} \quad X \bmod D = R \tag{3.2}$$

In diesem Kapitel werden die wichtigsten Algorithmen zur Berechnung der Division durch digitale Schaltkreise vorgestellt.

Eine sehr gute Einführung in das Gebiet der arithmetischen Algorithmen bietet [26]. Einen vergleichenden Überblick verschiedener Divisionsalgorithmen und ihre Implementationen wird in [34] gegeben.

3.1. Sequentielle Division

Die in diesem Abschnitt vorgestellten Schaltkreise berechnen die $\div$ und mod Operation für Festkommazahlen.

Bereits bei der Einführung der Kodierung für die ganzen Zahlen $\mathbf{Z}$ und den Festkommazahlen wurde darauf hingewiesen (vergleiche Anmerkung 2.3.1 auf Seite 35), dass Schaltkreise für Festkommazahlen, ohne Modifikation des Schaltkreises, auch mit ganzen Zahlen arbeiten können. Hierzu muss lediglich die Interpretation der Bitvektoren an den Ein- und Ausgängen angepasst werden.

Im folgenden wird zusätzlich zur der meist üblichen Beschreibung der Schaltkreise im Falle der Festkomma-Arithmetik auch die entsprechende Kodierung für den Fall der Ganzzahl-Arithmetik angegeben, da sich diese direkt auf Word-Level Diagramme zur Darstellung von Funktionen des Typs $\mathbf{B}^n \to \mathbf{Z}$, bzw. $\mathbf{B}^n \to \mathbf{Z}^2$, umsetzen lassen.

Werden die Ein- und Ausgaben des Dividieres als Festkommazahlen interpretiert, so gelten für die Kodierung die in folgender Definition eingeführten Bezeichnungen.

Definition 3.1.1 (Festkommazahlen) Es gelten folgende Bezeichnungen:

$$
\begin{aligned}
D &= \sum_{i=-(n-1)}^{0} 2^i \cdot d_i && \text{Divisor} \\
Q &= \sum_{i=-(n-1)}^{0} 2^i \cdot q_i && \text{Quotient} \\
q_j &\in \{0,1\} && j\text{-te Quotientenstelle} \\
R^{(0)} &= \sum_{i=-(n-1)}^{0} \left(2^i \cdot r_i^{(0)}\right) && \text{Dividend}
\end{aligned}
$$

□

Somit lässt sich die Division für Festkommazahlen beschreiben durch:

$$
\begin{aligned}
R^{(j+1)} &= 2 \cdot \left(R^{(j)} - q_j \cdot D\right) \\
q_{-j} &= \max\left\{ i \,\middle|\, (0 \le i \le 1) \wedge \left(R^{(j)} - i \cdot D \ge 0\right) \right\} \\
j &= 0, \ldots, n-1
\end{aligned}
$$

Ohne Beschränkung der Allgemeinheit seien D und R positiv (also $r_0^{(0)} = d_0 = 0$). Weiter gelte die Vorbedingung

$$0 < R^{(0)} < D\,. \tag{3.3}$$

Dies stellt keine echte Einschränkung dar, da diese Bedingung durch geeignetes Shiften der Eingabe erreicht werden kann. Sollen auch negative Zahlen zugelassen werden, eignet sich die Betrag-Vorzeichen Darstellung für $R^{(0)}$ und Q. In diesem Fall kann durch die vorgestellten Schaltkreise die Division für die Beträge berechnet werden. Das Vorzeichenbit des Quotienten ergibt sich dabei direkt, durch eine eine $\oplus$-Operation, aus den Vorzeichenbits von Divisor und Dividend.

Korollar 3.1.1
Sei R der Rest der Festkomma-Division, so folgt aus der Vorbedingung (3.3) für das Ergebnis der Division direkt:

$$0 \le Q < 1 \quad \textit{und} \quad 0 \le R < D$$

□

Um die vorgestellten Verfahren zur Division von Festkommazahlen auch für die Division von Binärzahlen (ganzen Zahlen) anwenden zu können, genügt es, die Ein- und Ausgabewerte entsprechend zu interpretieren. Dazu muss nur Definition 3.1.1 angepasst werden:

Definition 3.1.2 (ganze Zahlen) Es gelten folgende Bezeichnungen:

$$\begin{aligned} D &= \sum_{i=0}^{n-1} 2^i \cdot d_i && \text{Divisor} \\ Q &= \sum_{i=0}^{n-1} 2^i \cdot q_i && \text{Quotient} \\ q_j &\in \{0,1\} && j\text{-te Quotientenstelle} \\ R^{(0)} &= \sum_{i=n-1}^{2n-1} \left(2^i \cdot r^{(0)}_{i-(n-1)}\right) && \text{Dividend} \end{aligned}$$

□

Für die Vorbedingung (3.3) folgt damit:

$$0 < D < 2^n \quad \text{und} \quad 2^{n-1} \leq R^{(0)} < 2^{n-1} \cdot D \tag{3.4}$$

Entsprechend zu Korollar 3.1.1 ergibt sich:

Korollar 3.1.2
Sei R der Rest der Ganzzahl-Division, so folgt aus der Vorbedingung (3.4) für das Ergebnis der Division direkt:

$$0 \leq Q < 2^n \quad \textit{und} \quad 0 \leq R < D$$

□

3.1.1. Die restoring Division

Bei der restoring Division wird analog zur Division nach der „Schulmethode“ vorgegangen. Der Divisor wird solange vom aktuellen Partialrest abgezogen, bis ein negatives Ergebnis entsteht. In diesem Fall wird die letzte Subtraktion rückgängig gemacht und der Partialrest um eine Stelle nach links verschoben. Anschließend wird das Verfahren iterativ fortgesetzt. Somit ergibt sich mit den Bezeichnungen für Festkommazahlen:

Definition 3.1.3 (restoring Division)

$$\begin{aligned} R^{(j+1)} &= \begin{cases} 2 \cdot R^{(j)} & \text{falls } R^{(j)} - D < 0 \\ 2 \cdot \left(R^{(j)} - D\right) & \text{falls } R^{(j)} - D \geq 0 \end{cases} \\ q_{n-j} &= \begin{cases} 0 & \text{falls } R^{(j)} - D < 0 \\ 1 & \text{falls } R^{(j)} - D \geq 0 \end{cases} \\ j &= 0, \ldots, n-1 \end{aligned}$$

Diese Definition lässt sich durch Abbildung 3.1, welche die Auswahl des Quotientenbits q_{n-j} darstellt, veranschaulichen. Hierbei ist auf der x-Achse der um eine Stelle nach links geshiftete Partialrest R^j und auf der y-Achse der um eine Stelle nach rechts geshiftete Partialrest R^{j+1} aufgetragen. Das Diagramm veranschaulicht, dass im Falle $R^{(j)} < D$ das Quotientenbit q_{n-j} immer so gewählt wird, dass die Nachbedingung $R^{(j+1)} < D$ zugesichert werden kann. Da nach Voraussetzung $R^{(0)} < D$ gilt, wird sicher ein $R^{(n)}$ mit $R^{(n)} < D$ bestimmt. Somit gilt während des gesamten Verfahrens die Invariante $R^{(i)} < D$ für $0 \leq i \leq n$. Dadurch wird zudem sichergestellt, dass während der Berechnung keine Überläufe bei den Subtraktionen stattfinden können.

Beispiel 3.1.1 (restoring Division) Sei $R^{(0)} = (0101)_2 \cdot 2^3 = (0101000)_2 = (40)_{10}$ und $D = (0111)_2 = (7)_{10}$, also $-D = (1001)_2$. Es handelt sich um die Division von 4-Bit Zahlen. Somit ergibt sich:

$R^{(0)}$	0101000	
$-D$	1001	
	1110000	$< 0 \Rightarrow q_3 = 0$
$R^{(1)}$	101000	Linkshift von $R^{(0)}$ um eine Stelle, kein Überlauf
$-D$	1001	
	001100	$> 0 \Rightarrow q_2 = 1$
$R^{(2)}$	01100	Linkshift um eine Stelle, kein Überlauf
$-D$	1001	
	11110	$< 0 \Rightarrow q_1 = 0$
$R^{(3)}$	1100	Linkshift von $R^{(2)}$ um eine Stelle, kein Überlauf
$-D$	1001	
	0101	$> 0 \Rightarrow q_0 = 1$
$R^{(4)}$	0101	

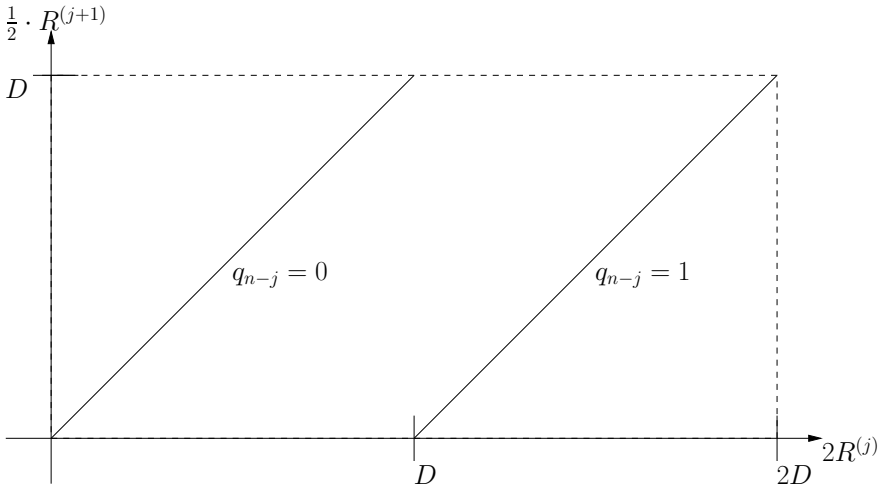
Daraus folgt: $Q = (0101)_2 = (5)_{10}$ und $R^{(4)} = (0101)_2 = (5)_{10}$. □

Zur Division zweier n-Bit Zahlen werden n Subtraktionen und höchstens n Additionen[1], insgesamt also $O(n)$ Operationen, benötigt. Eine einfache Implementierung für die restoring Division von 4 Bit Zahlen[2] ist in Abb. A.2 auf Seite 103 gegeben.

[1] Im statistischen Mittel ist mit $n/2$ tatsächlich notwendigen Additionen zu rechnen.

[2] Die Eingabe des Schaltkreises sind vorzeichenlose 3 Bit Zahlen, das Ergebnis sind vorzeichenbehaftete 4 Bit Zahlen.

Abbildung 3.1. Auswahl des Quotientenbits q_{n-j} bei der restoring Division



3.1.2. Die nonrestoring Division

Bei der nonrestoring Division werden negative Partialreste als Zwischenergebnisse zugelassen. Wird bei der restoring Division, falls ein negativer Partialrest auftrat, erst $2 \cdot D$ addiert und anschließend D subtrahiert, so wird im Fall der nonrestoring Division direkt D addiert. Ist der letzte Partialrest $R^{(n)}$ negativ, so wird nochmals D addiert, um einen positiven (Gesamt-) Rest $R^{(n+1)}$ zu erhalten.

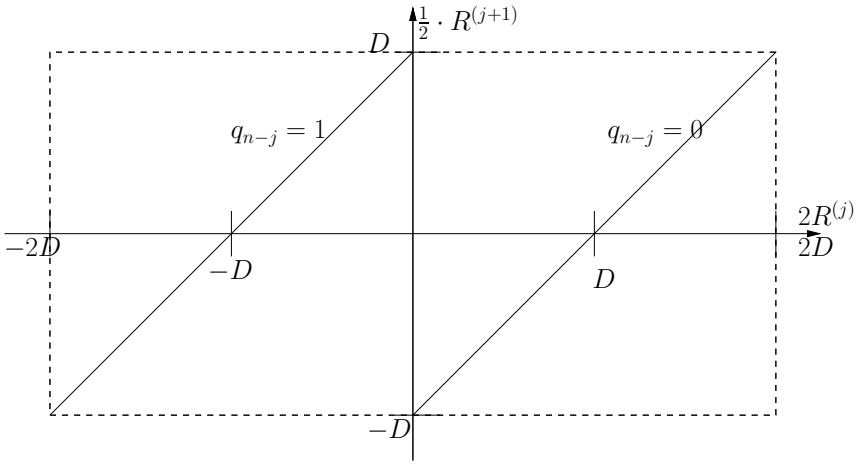
Mit den Bezeichungen für die Division von Festkommazahlen ergibt sich:

Definition 3.1.4 (nonrestoring Division)

$$
\begin{aligned}
R^{(j+1)} &= \begin{cases} 2 \cdot \left(R^{(j)} - D\right) & R^{(j)} \geq 0 \\ 2 \cdot \left(R^{(j)} + D\right) & R^{(j)} < 0 \end{cases} \\
q_{-j} &= \begin{cases} 0 & \text{falls } R^{(j+1)} \geq 0 \\ 1 & \text{falls } R^{(j+1)} < 0 \end{cases} \\
j &= 0, \ldots, n-1 \\
R^{(n+1)} &= \begin{cases} R^{(n)} & R^{(n)} \geq 0 \\ R^{(n)} + D & R^{(n)} < 0 \end{cases}
\end{aligned}
$$

In Abb. 3.2 ist die Regel zur Bestimmung des aktuellen Quotientenbits graphisch dargestellt. Die Verwandtschaft der restoring und nonrestoring Division tritt in dieser Darstellung besonders deutlich hervor (siehe Abb. 3.1). Es gilt in ähnlicher Form: Sei $|R^{(j)}| < D$,

Abbildung 3.2. Auswahl des Quotientenbits q_{-j} bei der nonrestoring Division

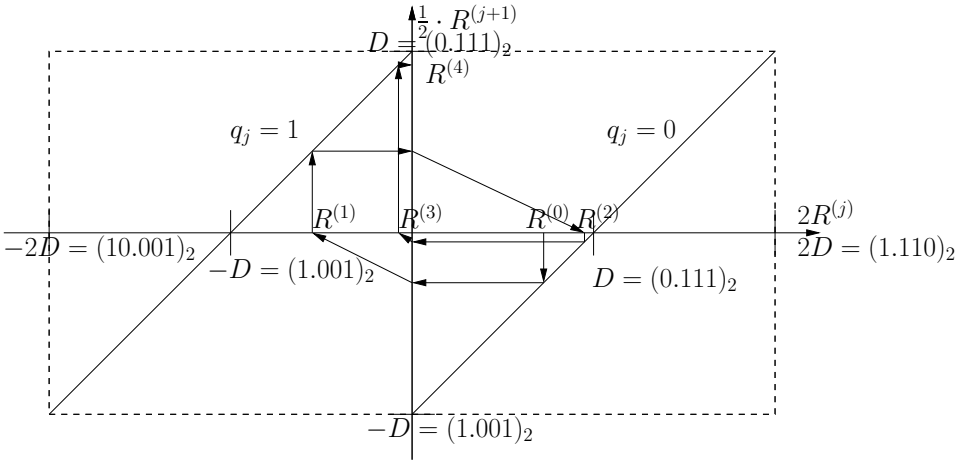


dann wird das Quotientenbit q_{-j} so gewählt, dass $|R^{(j+1)}| < D$ zugesichert werden kann. Da nach Voraussetzung $|R^{(0)}| < D$ gilt, wird ein $R^{(n+1)}$ mit $|R^{(n+1)}| < D$ bestimmt.

Beispiel 3.1.2 (nonrestoring Division) Sei $R^{(0)} = (0101)_2 \cdot 2^3 = (0101000)_2 = (40)_{10}$ und $D = (0111)_2 = (7)_{10}$, also $-D = (1001)_2$. Es handelt sich dabei um die Division von 4-Bit Zahlen. Die erste Stelle ist die Vorzeichenstelle. Somit ergibt sich:

$R^{(0)}$	0101000		
$-D$	1001		
	01110000		$< 0 \Rightarrow q_3 = 0$
$R^{(1)}$	110000		Linkshift um eine Stelle, kein Überlauf
$+D$	0111		
	1001100		$> 0 \Rightarrow q_2 = 1$
$R^{(2)}$	01100		Linkshift um eine Stelle, kein Überlauf
$-D$	1001		
0	11110	$< 0 \Rightarrow q_1 = 0$	
$R^{(3)}$	1110		Linkshift um eine Stelle, kein Überlauf
$+D$	0111		
	10101		$> 0 \Rightarrow q_0 = 1$
$R^{(4)}$	0101		$> 0 \Rightarrow R^{(5)} = R^{(4)}$

Abbildung 3.3. Beispiel der nonrestoring Division



Daraus folgt: $Q = (0101)_2 = (5)_{10}$ und $R^{(5)} = (0101)_2 = (5)_{10}$. Der Rechenweg dieses Beispiels ist in Abb. 3.3 dargestellt, dabei stellen die horizontalen Pfeile die Addition von $\pm D$ und die diagonalen Pfeile die Multiplikation mit zwei (Shift um eine Stelle nach links) dar. □

Bei der nonrestoring Division werden zur Division zweier n-Bit Zahlen genau n Subtraktionen und maximal eine Addition, also $O(n)$ Operationen, benötigt. Die Komplexität einer einzelnen Operation ist bei der nonrestoring Division geringer als bei der restoring Division[3], da keine Subtraktionen rückgängig gemacht werden müssen. Eine einfache Implementierung für die nonrestoring Division von 4 Bit Zahlen ist in Abb. A.3 auf Seite 105 gegeben.

Eine Variante der nonrestoring Division wird beispielsweise im K5 Prozessor von AMD (siehe [31]) eingesetzt.

3.1.3. Implementierungen

Bei der Implementierung ist zu beachten, dass die benötigte Multiplikation mit zwei einem Shift um eine Stelle nach links entspricht, die allein durch die Verdrahtung der einzelnen Stufen realisiert werden kann. Es müssen keine besonderen Schaltkreise zum Multiplizieren oder Shiften aufgebaut werden.

[3] Im statistischen Mittel über alle Programmläufe werden bei der nonrestoring Division $\frac{n-1}{2}$ Additionen im Vergleich zur restoring Division eingespart. Bei der Schaltkreisrealisierung entfallen je nach Realisierung n Addierer oder Multiplexer.

Weiter kann bei der Implementierung aufgrund der Vorbedingung 3.3 auf Seite 42 ein n-Bit Dividierer durch n CAS[4] Stufen der Breite n realisiert werden[5]. Durch das Shiften und Weglassen des führenden Bits geht keine relevante Information verloren. Zudem werden bei den Berechnungen der einzelnen Stufen keine relevanten Überläufe erzeugt. Dieser Sachverhalt wird in Satz 3.1.1 (für die Interpretation der Bitvektoren als Festkommazahlen) formalisiert.

Satz 3.1.1 (Breite der Dividiererstufen)
Für die nonrestoring und restoring Division gilt: Aufgrund der Vorbedingung

$$0 < R^{(0)} < D \tag{3.5}$$

können Schaltkreise zur restoring und nonrestoring Division von n Bit Festkommazahlen mit CAS-Zellen der Breite n realisiert werden. □

Beweis: Für die restoring Division ergibt sich die Aussage direkt aus dem Algorithmus[6], für den Fall der nonrestoring Division ergibt sich:

Sei $(a_n, a_{n-1}, a_{n-2}, \ldots, a_0) \in \mathbf{B}^{n+1}$ das Ergebnis einer CAS-Zelle (Stufe), es ergeben sich die folgenden vier Fälle:

1. Fall: $[1.1a_{n-2}a_{n-3}\ldots] < 0$. Nach dem Shiften ergibt sich somit $[11.a_{n-2}a_{n-3}\ldots] = [1.a_{n-2}a_{n-3}\ldots]$. Das Weglassen des führenden Bits entspricht der Umkehrung einer Sign-Extension.

2. Fall: $[1.0a_{n-2}a_{n-3}\ldots] < 0$. Nach dem Shiften ergibt sich somit $[10.a_{n-2}a_{n-3}\ldots]$. Die Führende „1“ braucht bei der Addition nicht berücksichtigt werden, da $R^{(j)} = [10.a_{n-2}a_{n-3}\ldots]$ gilt. Laut Algorithmus gilt sicher: $-1 < R^{(j)} < 0$, und damit $R^{(j)} + D = [11.b_{n-1}b_{n-2}, \ldots] = [1.b_{n-1}b_{n-2}]$. Somit kann die führende „1“ bereits vor der Addition weggelassen werden.

3. Fall: $[0.0a_{n-2}a_{n-3}, \ldots] \geq 0$. Nach dem Shiften ergibt sich somit $[00.a_{n-2}a_{n-3}, \ldots] = [0.a_{n-2}a_{n-3}, \ldots] = [0.a_{n-2}a_{n-3}, \ldots] = [0.a_{n-2}a_{n-3}, \ldots]$

4. Fall: $[0.1a_{n-2}a_{n-3}, \ldots]$. Somit ergibt sich nach dem Shiften um eine Stelle nach links $[01.a_{n-2}a_{n-3}, \ldots] = [1.a_{n-2}a_{n-3}, \ldots]$. Die Führende „0“ braucht bei der Addition wegen $0 < R^{(j)-D} < 1$ nicht berücksichtigt werden.

[4] engl. Controlled Add Sub

[5] Im Falle der nonrestoring Division wird noch die Stufe zur Korrektur negativer Reste benötigt, die auch die Breite n hat.

[6] Bei der restoring Division wird nur subtrahiert, wenn dadurch kein Überlauf ensteht, ansonsten wird nur geshiftet.

In jedem der vier Fälle gilt: Das Ausgangscarry ist gleich dem invertierten Vorzeichenbit. Bei Additionen von $R^{(j)}$ und D sind die Vorzeichen entgegengesetzt, es ensteht also kein Überlauf und damit ist das Ausgangscarry bei der Addition gleich dem negierten Vorzeichenbit. ■

Um den Einfluss der konstanten Breite der einzelnen Stufen untersuchen zu können, wird auch ein „verbreiterter“ nonrestoring Dividierer (siehe hierzu auch Abschnitt 5.4.1 auf Seite 66) betrachtet. Bei dessen Implementierung wird zum einen das führende Bit nach dem Shiften nicht weggelassen, sondern in die nächste Stufe hineingeführt. Zum anderen wird der Addierer/Subtrahierer so modifiziert, dass er Satz 2.3.3 auf Seite 38 genügt. Somit gilt für alle n-Bit Zahlen A, B in Zweierkomplementdarstellung: $A \pm B = C$, wobei C eine $n+1$-Bit Zahl in Zweierkomplementdarstellung ist. Durch diese Modifikation werden die Busse für die Partialreste bei jeder Stufe um 2 Bit breiter. Die letzte Stufe hat also die Breite $3n$. Ein nach dieser Idee aufgebauter nonrestoring Dividierer mit einer Breite von 4 Bit ist in Abbildung A.4 auf Seite 107 dargestellt.

Während der „schmale“ nonrestoring Dividierer einen Platzbedarf[7] von $6n^2 + 5n - 4$ hat, benötigt die „verbreiterte“ Variante Platz $12n^2 + 17n - 2$.

$$\lim_{n\to\infty} \frac{12n^2 + 17n - 2}{6n^2 + 5n - 4} = 2 \tag{3.6}$$

Asymptotisch benötigt der „breite“ Dividierer doppelt soviel Platz wie die schmale Variante, wobei bereits für Dividierer mit Bitbreiten von 16 der Faktor kleiner als 2.1 ist.

3.2. Schnelle Division

Es existieren zwei unterschiedliche Ansätze, um schnelle Algorithmen zur Division zu entwickeln. Die übliche Methode verwendet Additionen und Subtraktionen, die zweite Methode basiert auf der Multiplikation als Basisoperation. Beim ersten Ansatz ist die Anzahl auszuführender Schritte linear zur Wortbreite n, während bei der Verwendung der Multiplikation nur $\log_2 n$ (Multiplikations-) Operationen notwendig sind, um eine Genauigkeit von n Bits zu erreichen. Dafür ist die Multiplikationsoperation im Vergleich zur Additionsoperation aufwendiger.

3.2.1. SRT Division

Der nach seinen Erfindern[8] Sweeny [27], Robertson [37] und Tocher [43] benannte Algorithmus ist ein Vertreter der auf Addition und Subtraktion basierenden Divisions-

[7] Bei der Verwendung beliebiger Funktionen $\mathbf{B}^2 \to \mathbf{B}$ als Grundgatter. Die in der Schaltkreisrealisierung verwendeten Gatter sind in Abbildung A.1 auf Seite 102 dargestellt.

[8] Der Algorithmus wurde von den drei Autoren unabhängig voneinander entwickelt.

algorithmen. In diese Klasse[9] gehören im übrigen auch die vorgestellten restoring und nonrestoring Divisionsverfahren, im Gegensatz zu diesen erlaubt der SRT Dividierer eine „schnelle“ Division.

Eine Variante des SRT Dividierers wird z.B. im Pentium Chip von Intel (siehe [41]) zur Division von Fließkommazahlen eingesetzt. Im Allgemeinen gilt bei SRT-Divisionen für Festkommazahlen:

$$q_{-i} = \begin{cases} 1 & \text{falls } 2 \cdot R^{(i-1)} \geq D \\ 0 & \text{falls } -D \leq 2 \cdot R^{(i-1)} < D \\ \bar{1} & 2 \cdot R^{(i-1)} < -D \end{cases} \tag{3.7}$$

wobei sich der neue Partialrest berechnet als

$$R^{(i)} = 2 \cdot R^{(i-1)} - q_{-i} \cdot D \tag{3.8}$$

Die Schwierigkeit liegt darin, dass mit dieser Auswahl ein vollständiger Vergleich zwischen $2 \cdot R^{(i-1)}$ und D bzw. $-D$ notwendig ist. Wenn allerdings ein normalisiertes D betrachtet wird, welches die Bedingung $\frac{1}{2} \leq |D| < 1$ erfüllt, so wird der Bereich für $q_{-i} = 0$ eingeschränkt auf

$$-D \leq -\frac{1}{2} \leq 2 \cdot R^{(i-1)} < \frac{1}{2} \leq D\,. \tag{3.9}$$

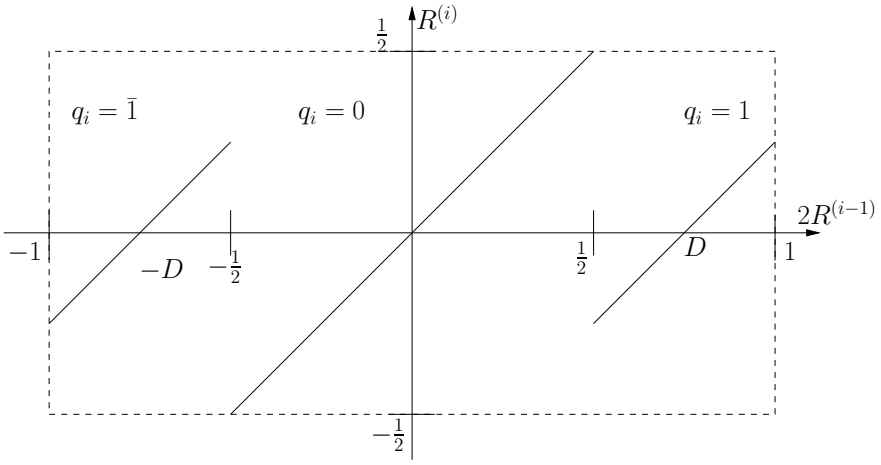
Somit muss der Partialrest $R^{(i-1)}$ nur mit $\frac{1}{2}$, bzw. $-\frac{1}{2}$ verglichen werden. Dabei kann ausgenutzt werden, dass bei der Darstellung als Zweierkomplement folgendes gilt: Eine Zahl ist genau dann größer oder gleich $\frac{1}{2}$, wenn die Zahl mit 0.1 beginnt. Andererseits gilt: Eine Zahl ist genau dann kleiner als $-\frac{1}{2}$, wenn die Zahl mit 1.0 beginnt. Es müssen also nur noch 2 Ziffern verglichen werden. Für die Auswahl des Quotientenbits ergibt sich.

$$q_{-i} = \begin{cases} 1 & \text{falls } 2 \cdot R^{(i-1)} \geq \frac{1}{2} \\ 0 & \text{falls } -\frac{1}{2} \leq 2 \cdot R^{(i-1)} < \frac{1}{2} \\ \bar{1} & \text{falls } 2 \cdot R^{(i-1)} < -\frac{1}{2} \end{cases} \tag{3.10}$$

Wird die Redundanz der gewählten Signed-Digit Darstellung ausgenutzt, so kann die Auswahl des Quotientenbits durch die Verwendung einer Lookup-Table deutlich beschleunigt werden. Die im Pentium Chip realisierte Variante der SRT-Division wird in den verschiedenen Arbeiten über den FDIV-Bug [9, 16, 36, 41] erläutert. Eine ausführliche Behandlung der verschiedenen Varianten von SRT Dividierern findet sich in [26].

[9] In der englischsprachigen Literatur werden diese Algorithmen auch als “digit recurrence algorithms” bezeichnet.

Abbildung 3.4. Auswahl des Quotientenbits q_i bei der SRT Division



Ähnlich wie bei der restoring und nonrestoring Division kann auch die Auswahl des Quotientenbits bei der SRT Division graphisch dargestellt werden (siehe Abb. 3.4). Das Diagramm veranschaulicht folgendes: Das Quotientenbit muss so gewählt werden, dass $|R^{(i)}| < |D|$ gilt, um die Konvergenz des Verfahrens mit einem Rest $R < |D|$ zu garantieren.

Beispiel 3.2.1 (SRT Division) Sei $R^{(0)} = (0.0101)_2 = \left(\frac{5}{16}\right)_{10}$ und $D = (0.1100)_2 = \left(\frac{3}{4}\right)_{10}$. Somit ergibt sich:

$R^{(0)}$	0.0101	
$2R^{(0)}$	0.1010	$\geq \frac{1}{2} \Rightarrow q_1 = 1$
$-D$	1.0100	
$R^{(1)}$	1.1110	
$2R^{(1)} = R^{(2)}$	1.1100	$\geq -\frac{1}{2} \Rightarrow q_2 = 0$
$2R^{(2)} = R^{(3)}$	1.1000	$\geq -\frac{1}{2} \Rightarrow q_3 = 0$
$2R^{(3)}$	1.0010	$< -\frac{1}{2} \Rightarrow q_4 = \bar{1}$
$+D$	0.1100	
$R^{(4)}$	1.1100	negativer Partialrest und positives D
$+D$	0.1100	Korrektur
$R^{(4)}$	0.1000	Rest nach Korrektur

Der Quotient vor der Korrektur ist $Q = 0.100\bar{1}$, nach der Korrektur gilt $Q = (0.0110)_2 = (3/8)_{10}$ und der Rest $(1/32)_{10}$. □

3.2.2. Division durch Multiplikation

Wird der Divisor D als Nenner und der Dividend N als Zähler des Quotienten Q betrachtet, so gilt $N/D = Q$. Diese Gleichung gilt insbesondere auch, wenn sowohl Zähler als auch Nenner mit den Faktoren $R^{(0)} \cdots R^{(m-1)}$ multipliziert (erweitert) werden. Falls die Faktoren $R^{(i)}$ so gewählt wurden, dass $D \cdot R^{(0)} \cdots R^{(m-1)} \rightarrow 1$ gilt, so konvergiert der Zähler gegen Q:

$$Q = \frac{N}{D} = \frac{N \cdot R^{(0)} \cdots R^{(m-1)}}{D \cdot R^{(0)} \cdots R^{(m-1)}} \rightarrow \frac{Q}{1} \tag{3.11}$$

Da nach Gleichung (3.11) nur der Quotient Q berechnet wird, ist eine weitere Berechnung für den Rest notwendig. Deswegen eignet sich dieses Vorgehen besonders für Fließkommazahlen, bei denen kein Rest berechnet werden muss.

Wesentlich bei dieser Methode ist die Auswahl der Faktoren $R^{(i)}$, so dass die Konvergenz des Nenners gegen 1 gesichert ist. Sei der Divisor ein binärer Bruch der Form $0.1\ldots$. Somit gilt $1/2 \leq D < 1$ und $D = 1 - y$, wobei $y \leq 1/2$. Wenn $R^{(0)}$ als $1 + y$ gewählt wird, dann ergibt sich für den Nenner $D^{(1)}$:

$$D^{(1)} = D \cdot R^{(0)} = (1 - y) \cdot (1 + y) = 1 - y^2 \tag{3.12}$$

Wegen $y^2 \leq 1/4$ gilt weiter $D^{(1)} \geq 3/4$. Damit hat $D^{(1)}$ die Form $D^{(1)} = 0.11\ldots$. Im Schritt 2 wird $R^{(1)} = 1 + y^2$ gewählt, somit gilt:

$$D^{(2)} = D^{(1)} \cdot R^{(1)} = (1 - y^2) \cdot (1 + y^2) = 1 - y^4 \tag{3.13}$$

Jetzt gilt $y^4 \leq 1/16$, und damit ist $D^{(2)}$ von der Form $D^{(2)} = 0.1111\ldots$. Allgemein gilt für den Schritt $i + 1$, dass der Divisor die Form $D^{(i)} = 1 - y_i$ mit $y_i = y^{2^i}$ hat. Wegen $y \leq 2^1$, hat $D^{(i)}$ mindestens 2^i führende Einsen. Der sich ergebende Faktor ist $R^{(i)} = 1 + y_i$ und somit wird $D^{(i+1)} = D^{(i)} \cdot R^{(i)}$ mindestens 2^{i+1} führende Einsen haben (auf einen ausführlichen Konvergenzbeweis wird hier verzichtet, siehe hierzu [26]).

Der Bruch Q wird solange mit $R^{(i)}$ erweitert, bis $D^{(i)}$ zu 1 (bzw. $0.111\ldots1$) konvergiert. Es ist leicht zu sehen, dass die Anzahl führender Einsen in jedem Iterationsschritt verdoppelt wird. Um eine Genauigkeit von n Stellen, also die Division für n-Bit Zahlen zu berechnen, beträgt somit die Zahl der Iterationen $m = \lfloor \log_2 n \rfloor$. Das Verfahren konvergiert quadratisch.

Der Athlon Prozessor von AMD verwendet beispielsweise einen Algorithmus zur Division, der die Multiplikation als Basisoperation verwendet. Das in diesem Fall eingesetzte Verfahren wird in [33] ausführlich vorgestellt.

4. Eine neue Idee zur Verifikation von Dividierern

In diesem Kapitel wird eine neue, DD-basierte Methode zur Verifikation von Dividierern vorgestellt, bei der durch eine geeignete Transformation vermieden wird, die Operationen $\div$ und mod explizit darstellen zu müssen.

4.1. Einführung in die DD basierte Schaltkreisverifikation

Bei der Verifikation von Schaltkreisen ist eine Spezifikation gegeben und es soll gezeigt werden, dass ein Schaltkreis (Implementierung) ihr entspricht. Meist ist die Spezifikation als boolesche Funktion gegeben. Um nun zu überprüfen, dass die Implementierung die Spezifikation erfüllt, wird aus dem Schaltkreis die Funktion konstruiert, die von ihm repräsentiert wird. Ein Vergleich mit der Funktion der Spezifikation ergibt dann eine Aussage über die Korrektheit der Implementierung.

Bei der Verifikation wird also eine Datenstruktur benötigt, mit der sich boolesche Funktionen effizient darstellen und vergleichen lassen. Der naive Ansatz, einfach alle möglichen Belegungen „auszuprobieren“ (vollständige Simulation), ist in der Praxis nicht anwendbar, da in der Regel exponentiell viele Berechnungen ausgeführt werden müssen.

Für viele Funktionen haben sich DDs als Datenstrukturen bewährt. Werden kanonische DD-Varianten (z.B. OKFDDs oder K*BMDs) verwendet, ist ein effizienter Vergleich der dargestellten Funktionen möglich. Bei geeigneter Implementierung ist der Identitätsvergleich sogar in konstanter Zeit realisierbar, siehe hierzu z.B. [11]. Das Aufbauen eines (kanonischen) DDs zu einem Schaltkreis wird als symbolische Simulation bezeichnet. Die Verifikation mittels DDs ermöglicht es auch, durch einfach $\oplus$-Berechnung von Schaltkreis und Spezifikation, diejenigen Eingaben zu ermitteln, bei denen sich ein unterschiedliches Ausgabeverhalten ergibt.

Die so durchgeführte DD basierte Verifikation kann nur effizient durchgeführt werden, wenn sowohl die Darstellung der Spezifikation (also im Korrektheitsfall auch die des Schaltkreises) effizient möglich ist. Zudem muss es auch eine effiziente Möglichkeit geben, aus dem Schaltkreis den DD der dargestellten Funktion zu berechnen.

4.2. Problematiken der Dividiererverifikation

Bereits 1991 wurde von Bryant [6] für OBDDs eine exponentielle untere Schranke zur Darstellung von Ganzzahl Multiplizieren bewiesen.

Durch die Einführung von BMDs und *BMDs in [7] und [8] wurde es möglich, effizient Multiplizierer darzustellen. Für eine spezielle Klasse von Multiplizierern konnte ein effizientes, *BMD-basiertes Verifikationsverfahren angegeben werden, bei dem während des gesamten Aufbaus des *BMDs eine polynomielle obere Schranke garantiert werden kann (siehe [25, 18]).

Inwiefern durch die Einführung weiterer Word-Level Diagramme eine effiziente Darstellung von Dividierern möglich ist, war lange Zeit ein offenes Problem. Erst 1998 zeigte Nakanishi (中西正樹) [32] eine exponentielle untere Schranke bei der Darstellung der Operationen $\div$ und mod für *BMDs. Unabhängig davon wurde von Scholl, Becker und Weis [39, 40] im selben Jahr ein noch allgemeineres Ergebnis bewiesen: Für alle bekannten Word-Level Datenstrukturen ist die Berechnungsstärke zu schwach, um $\div$ oder mod effizient darzustellen. Der von ihnen bewiesene, zentrale Satz sei hier nochmals ohne Beweis wiedergegeben:

Satz 4.2.1 (Darstellung der Division mit Word-Level Datenstrukturen)
*MTBDDs, EVBDDs, BMDs, *BMDs, K*BMDs und *PHDDs benötigen unabhängig von der Variablenordnung eine Größe von* $\Omega\left(2^{\frac{n}{16}}\right)$ *zur Darstellung der Operationen* mod *und* $\div$. □

Beweis: Siehe [39]. ■

Da bei den bisher üblichen Ansätzen zur DD-basierten Verifikation der zu verifizierende Schaltkreis und seine Spezifikation explizit als DD dargestellt werden, lässt sich aus diesen Ergebnissen schlussfolgern, dass sich Dividierer mit den bekannten DD-basierten Ansätzen nicht effizient und vollständig verifizieren lassen.

4.3. Bisherige Ansätze

Bisher basieren die meisten Ansätzen zur Verifikation von arithmetischen Schaltkreisen auf halbautomatischen Verfahren, die Theorem Proving und Model Checking Techniken einsetzen. Insbesondere für die Division konnte bisher kein effizientes Verfahren zur vollständigen DD-basierten Verifikation angegeben werden.

4.3.1. DD basierte Verfahren

Als einer der wenigen DD-basierten Ansätze zur Verifikation von Dividierern soll der in [4] angegebene Ansatz skizziert werden. Bryant zeigt in dieser Arbeit auf, dass bereits mit

der vorgestellten Methode der „Pentium Bug“ hätte vermieden werden können. Dazu untersucht er einen SRT-Dividierer und führt eine BDD-basierte Verifikation auf Bit-Level-Ebene durch. Die Methode basiert auf einem „gate-level-checker“-Schaltkreis, der zur Verifikation benötigt wird und ca. 4 mal so groß ist wie der eigentliche Dividierer.[1] Mit dieser Methoden lässt sich allerdings nur die Korrektheit einer Iteration (Stufe) des Dividierers nachweisen, dafür dauert die vollautomatische Verifikation nur etwa 10 Minuten.

4.3.2. Model Checking und Theorem Proving

Mittels Model Checking und Theorem Proving[2] wurden sowohl Dividierer, die auf Addition bzw. Subtraktion basieren, als auch solche, die die Multiplikation als Basisoperation verwenden verifiziert.

4.3.2.1. Addition und Subtraktion als Basisoperation

In [29] wird ein allgemeines Verfahren für die Klasse der auf Subtraktion basierenden Divisionsverfahren angeben. Der Beweis kann anschließend für bestimme Dividierer instanziiert werden, so wird z.B. der im Pentium eingesetzte Dividierer verifiziert.

O’Leary stellt in [35] eine Verifikation des im Intel Pentium Pro verwendeten Dividierers vor, der eine Weiterentwicklung des im Intel Pentium benutzten SRT Dividierers sein dürfte. Dabei wird das speziell auf die Hardware-Verifikation optimierte, formale Verifikationsystem Forte eingesetzt. Bemerkenswert ist, dass im Gegensatz zu den meisten anderen Arbeiten in diesem Bereich, der Dividierer auf Gate-Ebene verifiziert wurde. Interessant ist auch die Aussage, dass sich die Verifikation gelohnt hat, obwohl incl. Grundlagenforschung etwa 2.75 Mannjahre investiert wurden.

4.3.2.2. Multiplikation als Basisoperation

In [38] wird ein auf ACL2[3] basierender Beweis der Korrektheit der in [33] vorgestellten FPU des AMD K7 vorgestellt. Dieser Prozessor verwendet ein auf Multiplikation basierendes Verfahren.

Hierbei führt Russinoff eine Verifikation eines abstrakten Modells des Dividierers[4], wobei er die Frage offen lässt, wie die Äquivalenz von Gate-Ebene und dem verwendeten Modell sichergestellt wird. Der Beweis wurde mit dem nicht auf Hardware-Verifikation

[1] Die Größe eines Schaltkreises wird durch die Anzahl Gatter einer Bibliothek abgeschätzt.
[2] Eine Einführung in die Verwendung von logikbasierten Verfahren im Bereich der Hardware-Verifikation bietet [17].
[3] Applicative Common Lisp
[4] Russinoff verifiziert ein von den Entwicklern zur Verfügung gestelltes C-Programm zusammen mit einer Spezifikation der FPU.

spezialisierten Theorem Prover ACL2 durchgeführt und kann auch „per Hand“ nachgerechnet werden.

4.3.3. Vergleich der bisherigen Ansätze

Keiner der bisherigen Ansätze erlaubt eine vollständige und vollautomatische Verifikation von Dividierern, wie sie z.B. mittels DDs bei Addierern möglich ist. Wünschenswert sind Verfahren, die vollautomatisch auf Gate-Level Ebene arbeiten. Dabei bedeutet vollautomatisch, dass keine manuellen Eingriffe während der Verifikation notwendig sind. Verfahren, die auf Gate-Level Ebene, also auf den Netzlisten, arbeiten, sind zu bevorzugen, da diese Darstellung am besten die reale Implementierung repräsentiert.[5] Arbeitet ein Verfahren mit Netzlisten als Eingabe, so fallen zum einen Übersetzungs- und Interpretationsschritte in andere Beschreibungen weg, zum anderen kann die Verifikation sehr leicht in den Optimierungsprozess integriert werden. Denkbar sind hierbei sukzessive Vorgehensweisen, bei denen der Schaltkreis mittels CAE-Tools[6] nahezu automatisch entworfen wird. Nachdem der Schaltkreis verifiziert wurde, können von Hand (lokale) Optimierungen am Design vorgenommen werden. Existiert eine Möglichkeit vollautomatisch, auf Gate-Level-Ebene zu verifizieren, kann somit jede manuelle Optimierung sehr leicht auf ihre Korrektheit hin überprüft werden.

Eine vollautomatische Verifikation auf Gate-Level Ebene leistet nur der Ansatz von Bryant. Allerdings wird dabei nur ein Teil des Schaltkreises verifiziert. Bei den auf Model Checking und Theorem Proving basierenden Verfahren muss in der Regel ein Kompromiss gefunden werden zwischen dem Automatisierungsgrad und der Art und Weise der Schaltkreisbeschreibung.

4.4. Ein neuer Ansatz zur Verifikation von Dividierern

Da mit den bekannten DD-basierten Datenstrukturen die Funktionen $\div$ und mod nicht effizient dargestellt werden können, muss ein DD basiertes Verifikationsverfahren auf die direkte Darstellung des Dividiererschaltkreises verzichten. Um dies zu erreichen, wird der Dividiererschaltkreis transformiert und die eigentliche Verifikation im Transformationsraum ausgeführt. Dabei muss für die Transformation gelten:

- Die Implementierung (Schaltreis) ist genau dann korrekt, wenn die Verifikation des transformierten Schaltkreises ein positives Ergebnis liefert.
- Die Transformation muss effizient durchgeführt werden können.

[5] Im allgemeinen entspricht die Netzliste der Implementierung, bzw. kann leicht maschinell aus dieser erzeugt werden.

[6] Programme zum computerunterstützten Entwurf von Schaltkreisen, engl. computer aided engineering

In dem hier vorgestellten Verfahren wird als Transformation eine Proberechnung verwendet. Genauer wird ausgenutzt, dass Satz 4.4.1 gilt.

Satz 4.4.1
Sei C ein Schaltkreis mit den Eingängen $D, R^{(0)} \in \mathbf{B}^n$ und den Ausgängen $Q, R^{(n+1)} \in \mathbf{B}^n$, der die Division berechnet, so gilt:

$$\begin{aligned} &\Big(Q = R^{(0)} \div D \wedge R^{(n+1)} = R^{(0)} \bmod D\Big) \\ \Longleftrightarrow &\Big((Q \cdot D) + R^{(n+1)} = R^{(0)} \wedge 0 \leq R^{(n+1)} < D\Big) \end{aligned} \tag{4.1}$$

□

Als mögliche Transformation bietet es sich an, den Dividiererschaltkreis um einen „virtuellen“ Schaltkreis, der die Funktion $(Q \cdot D) + R^{(n+1)}$ berechnet zu erweitern. Somit ensteht der in Abbildung 4.1 auf der nächsten Seite skizzierte Schaltkreis. Aus Satz 4.4.1 folgt direkt:

Korollar 4.4.1 (Korrektheit)
Der Dividierer ist genau dann korrekt, wenn der Gesamtschaltkreis aus Abbildung 4.1 die Identität berechnet und $0 \leq R^{(n+1)} < D$ gilt. □

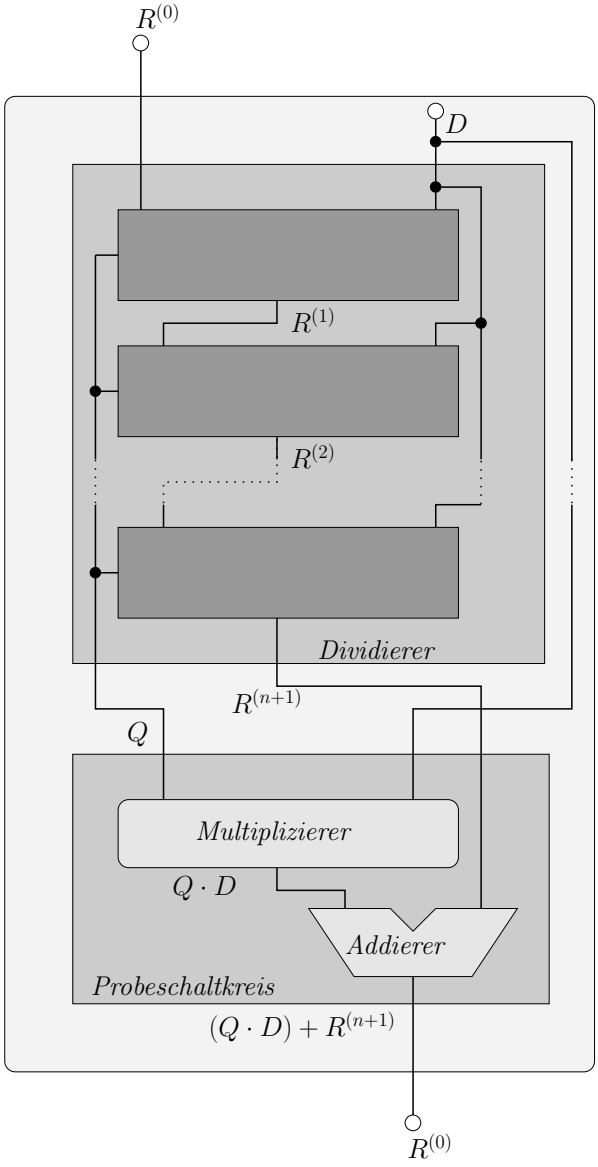
Die gewählte Transformation besteht also „nur“ aus einer Erweiterung des Dividierers an dessen Ausgängen. Deswegen ist auch im transformierten Raum die Darstellung der Funktionen $\div$ und $\bmod$ notwendig, falls, wie meist üblich, der DD zur Verifikation durch Syntheseoperationen von den Eingängen her aufgebaut wird.

Alternativ kann die DD Darstellung eines Schaltkreises durch *Rückwärtseinsetzen* (Substitution) aufgebaut werden. Hierbei wird bei den Ausgängen des Schaltkreises begonnen und sukzessive die Funktionalität der einzelnen Gatter hineinsubstituiert. Mit dieser Vorgehensweise konnten bereits effiziente DDs für Multiplizierer aufgebaut werden (siehe [25]).

Aus bestehenden Arbeiten (siehe auch Tabelle 2.2 auf Seite 33) ist bekannt, dass sich die „Transformationsfunktion“ $D \cdot Q + R^{(n+1)}$ effizient als K∗BMD darstellen lässt. Somit ist die Darstellung an der Stelle des Verfahrens, bei der mit der Substitution des eigentlichen Dividierers begonnen wird, klein. Im Gegensatz dazu müsste beim „Vorwärtsaufbauen“ an dieser Stelle der Dividiererschaltkreis dargestellt werden. Erste Überlegungen geben Anlass zur Hoffnung, dass nach der Substitution eines Blockes, der die Addition bzw. Subtraktion von D vom aktuellen Partialrest $R^{(i)}$ berechnet, wieder eine Funktion dargestellt wird, die der Transformationsfunktion sehr ähnlich ist.

Am Ende des Verfahrens wird bei einem korrekten Dividierer $R^{(0)}$ dargestellt, da der transformierte Schaltkreis die Identität auf $R^{(0)}$ berechnet. Das Verfahren beginnt mit der K∗BMD-Darstellung der Funktion $D \cdot Q + R^{(n+1)}$, die sicherlich effizient ist. Weiter ist bekannt, dass am Ende des Verfahrens $R^{(0)}$ dargestellt wird, wobei $R^{(0)}$ eine effiziente K∗BMD-Darstellung besitzt. Im folgenden bleibt zu untersuchen, wie sich die

Abbildung 4.1. Idee zur Verifikation von Dividiererschaltkreisen



darzustellenden DD-Größen im Verlauf des Verfahrens verhalten bzw. wie diese „klein“ gehalten werden können.

In einem weiteren Schritt muss die Nachbedingung $0 \leq R^{(n+1)} < D$ bewiesen werden. Dies ist beispielsweise durch eine Bildberechnung möglich. Da bei der Bildberechnung eine Funktion $\chi : \mathbf{B}^{2n-2} \rightarrow \mathbf{B}$ dargestellt werden muss, eignen sich für diesen Schritt OKFDDs als Datenstruktur. Auch hierbei muss eine Darstellung der Funktionen $\div$ und mod vermieden werden. Wird „stufenweise“ (siehe Abschnitt 5.4.3 auf Seite 70) vorgegangen, muß nur die Funktion einer Stufe, nicht aber der gesamte Dividierer, als Bit-Level Diagramm dargestellt werden. Auch hier besteht die Hoffnung, die Bildberechnung, ausgehend von der Vorbedingung, effizient berechnen zu können.

Basierend auf dieser Grundidee soll ein Verifikationsverfahren für Dividierer entwickelt werden, das DD basiert ist und die Verifikation in nichtexponentiellem Platzbedarf ermöglicht. Dabei soll besonders auf die Automatisierbarkeit der Lösung geachtet werden, so dass die Schaltkreise mit minimalem manuellem Aufwand und wenig zusätzlichem Wissen verifiziert werden können. Durch die verwendeten Techniken zum Aufbau des den Schaltkreis repräsentierenden DDs ist eine Arbeitsweise auf Gate-Level-Ebene direkt gegeben.

5. Untersuchungen zur Verifikation der nonrestoring Division

In diesem Kapitel wird ein DD-basiertes Verfahren untersucht, das die effiziente Verifikation der in Kapitel 3.1.2 vorgestellten nonrestoring Dividierer ermöglichen soll. Hierzu wird der Dividierer durch einen Schaltkreis der die „Probe“ berechnet erweitert, so dass bei geschickter Vorgehensweise zu keinem Zeitpunkt des Verfahrens die Funktion $\div$ oder mod explizit dargestellt werden muss.

5.1. Notation

Neben den bereits eingeführten Notationen werden im folgenden die Bezeichnungen aus Abb. 5.1 verwendet, um zu beschreiben an welcher Stelle sich das Verfahren befindet. Hierbei sind die Nummerierungen der Strukturen aufsteigend so gewählt, wie sie in der Berechnung von Q und $R^{(n+1)}$ abgearbeitet werden. Da bei der Verifikation die Entscheidungsdiagramme im ersten Schritt durch Substitution aufgebaut werden, beginnt die Verifikation bei den Strukturen mit maximaler Nummerierung. Die anschließend durchgeführte Bildberechnung arbeitet in aufsteigender Reihenfolge.

Die Aufteilung in Stufen (Blöcke), die die Basisoperation Addition bzw. Subtraktion berechnen, ergibt sich in natürlicher Weise aus Definition 3.1.4 auf Seite 45. Die Stufen, die abhängig vom Vorzeichen des vorherigen Partialrestes den Wert von D addieren oder subtrahieren, werden als „Controlled-Add-Sub-Cell“, kurz CAS-Zelle, bezeichnet.

5.2. Vorüberlegungen

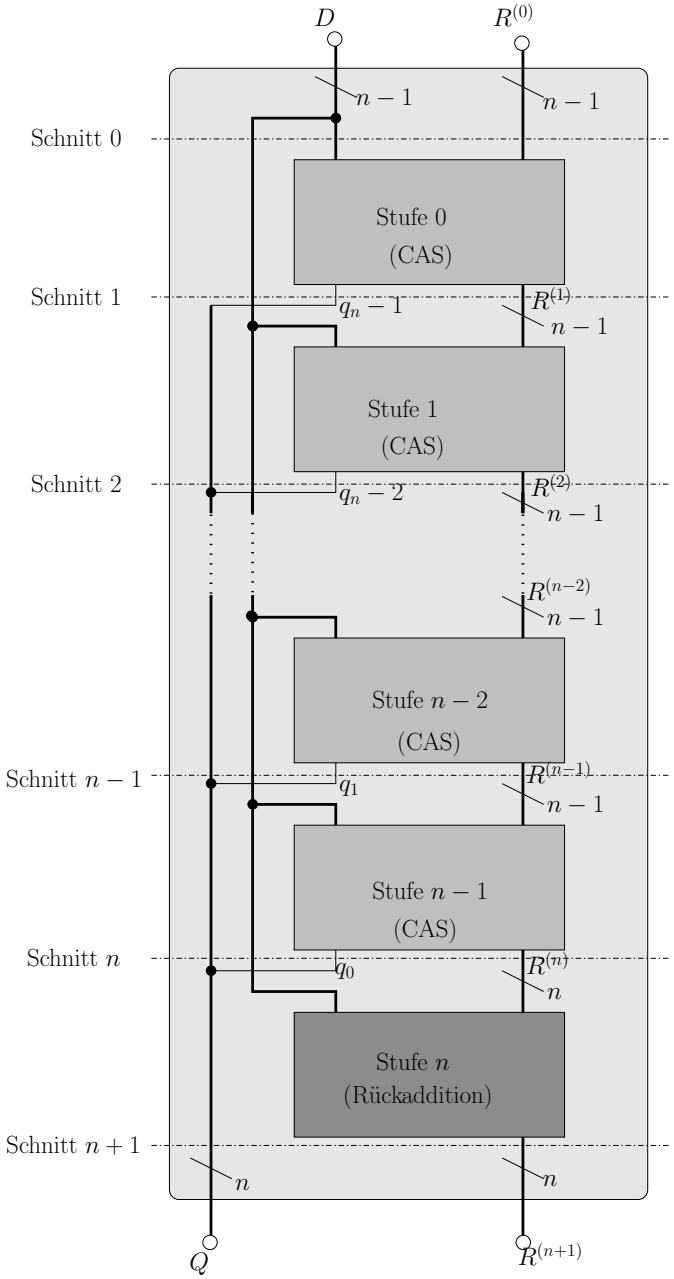
Bei der Verifikation der in Abschnitt 3.1.2 auf Seite 45 vorgestellten nonrestoring Division unter Berücksichtigung der Implementierungshinweise ergeben sich verschiedene Parameter und Problematiken, die bei der Umsetzung des in Abschnitt 4.4 auf Seite 56 vorgestellte Verfahrens berücksichtigt werden müssen:

- Die an den Eingängen des Dividierers geforderte Bedingung[1]:

$$0 < D < 2^n \quad \text{und} \quad 2^{n-1} \leq R^{(0)} \leq 2^{n-1} \cdot D \tag{5.1}$$

[1] Es wird die Division von ganzen Zahlen betrachtet.

Abbildung 5.1. Schematischer Aufbau eines n Bit nonrestoring Dividierers



muss bei der Verifikation berücksichtigt werden. Aus der Vorbedingung folgt direkt für die charakteristische Funktion $\chi(D, R^{(0)}) : \mathbf{B}^{2n-2} \rightarrow \mathbf{B}$ des Dividierers:

$$\chi(D, R^{(0)}) := \begin{cases} 1 & \text{falls} \quad 0 < D < 2^n \quad \text{und} \quad 2^{n-1} \leq R^{(0)} \leq 2^{n-1} \cdot D \\ 0 & \text{sonst} \end{cases} \tag{5.2}$$

Durch die Spezifikation des Dividierers wird nur sein Verhalten für den Definitionsbereich $\chi(D, R^{(0)}) = 1$ beschrieben. Dadurch kann auch nur für diesen Eingabebereich die Korrektheit der Division gefordert werden. Für die Korrektheit des Dividierers genügt es somit, dass der transformierte Schaltkreis auf dieser Eingabemenge die Identität auf $R^{(0)}$ darstellt. Dies wird in Satz 5.2.1 formalisiert.

- Als Transformation kann auch

$$D \cdot Q + R^{(n+1)} - R^{(0)} \tag{5.3}$$

verwendet werden. In diesem Fall stellt der Gesamtschaltkreis die konstante 0-Funktion dar, sofern der Dividierer korrekt ist.

- Die Bedingung $0 \leq R^{(n+1)} < D$ muss in einem zusätzlichen Schritt, z.B. durch eine Bildberechnung des Dividiererschaltkreises, gezeigt werden.

- Weiter sind die üblichen Parameter, die beim Einsatz von Entscheidungsdiagrammen die Darstellungsgröße entscheiden beeinflussen (z.B. Variablenordnung, Dekompositionstypliste, usw.), mit einzubeziehen. Die maximale Darstellungröße während dem Aufbau der Entscheidungsdiagramme,[2] wird zudem von der Durchlaufreihenfolge[3] bestimmt.

Satz 5.2.1 (Korrektheit der realen Implementierung)
Sei $\chi(D, R^{(0)}) : \mathbf{B}^{2n-2} \rightarrow \mathbf{B}$ *die charakteristische Funktion des Dividiererschaltkreises. Es gilt:*

$$\begin{aligned} & \Big((0 < D < 2^n) \wedge (2^{n-1} \leq R^{(0)} \leq 2^{n-1} \cdot D) \\ & \wedge (Q = R^{(0)} \div D) \wedge (R^{(n+1)} = R^{(0)} \bmod D) \Big) \\ \Longleftrightarrow & \Big(\chi(D, R^{(0)}) \cdot \Big(Q \cdot D) + R^{(n+1)} = R^{(0)} \Big) \wedge (0 \leq R^{(n+1)} < D) \Big) \end{aligned} \tag{5.4}$$

□

[2] Die maximale Größe wird meist als „Peak während des Aufbaus“ bezeichnet.
[3] Die Reihenfolge, in der die Gatter des Schaltkreises abgearbeitet werden.

Wird das Verfahren, wie in Abschnitt 4.4 auf Seite 56 beschrieben, durchgeführt, so muss auf Schnitt 0 nicht für alle möglichen Eingaben $R^{(0)}$ dargestellt werden. Es genügt, wenn $R^{(0)}$ auf dem Definitionsbereich dargestellt wird. Um dies zu berücksichtigen, genügt es beispielsweise, das Verfahren wie beschrieben durchzuführen und die auf Schnitt 0 dargestellte Funktion mit $\chi(D, R^{(0)})$ zu multiplizieren. Nach Satz 5.2.1 ist das Ergebnis dieser Multiplikation $R^{(0)}$, falls der Schaltkreis korrekt ist.

5.3. Untersuchungen zur Verifikation der Division

5.3.1. Erste Untersuchungen

Um einen ersten Eindruck zu bekommen, wurde das Verfahren mit folgenden Parametern implementiert:

1. Da sich die Zerlegungen $\{pD_{\mathbf{Z}}, nD_{\mathbf{Z}}\}$ als besonders geeignet für die Darstellung arithmetischer Funktionen erwiesen haben, wurde für alle Knoten die Zerlegung $pD_{\mathbf{Z}}$ gewählt.

2. Die charakteristische Funktion wurde erst am Ende des Verfahrens an den DD des transformierten Schaltkreises multipliziert. Da die charakteristische Funktion des Dividierers prinzipiell einen Bitvergleich darstellt, wurde hier ein OBDD als Datentyp gewählt.[4]

3. Als Variablenordung wurde eine verzahnte Sortierung gewählt, da sich gezeigt hat, dass sich mit dieser Sortierung sowohl die charakteristische Funktion als auch die Transformationsfunktion effizient darstellen lässt. Eine verzahnte Variablenordung bedeutet, dass sich die Bit-Level-Variablen der Operanden abwechseln. In diesem Fall wurde die Variablenordung so gewählt, dass sich auf Schnitt i die Sortierung

   $$q_{n-i} < r_{n-1}^{(i)} < d_{n-1} < r_{n-2}^{(i)} < d_{n-2} < \cdots < r_0^{(i)} < d_0$$

   ergibt.

Für eine erste Bewertung des Verfahrens genügt es, sich die Größen der darzustellenden DDs auf den Schnitten 0 bis $n+1$ zu betrachten. Mit den gewählten Parametern ergeben sich die in Tabelle 5.1 dargestellten Messwerte. In der Tabelle ist neben der maximalen DD-Größe während des Substituierens auch die Größe auf den einzelnen Schnitten in Anzahl der Knoten angegeben. Weiter ist auch die DD-Größe der charakteristischen Funktion $\chi(D, R^{(0)})$ und die zur Verifikation benötigte Zeit auf einer Ultra Sparc 1 angegeben. In der Tabelle sind die Schnitte in der Reihenfolge angegeben, wie sie im Verlauf

[4] Hierbei kann ausgenutzt werden, dass das verwendete DD-Paket (siehe [23]) Entscheidungsdiagramme mit unterschiedlicher DTL verknüpfen kann.

Tabelle 5.1. DD-Größen bei der Verifikation der nonrestoring Division

Bitbreite	3	4	5	6	7	8
maximale Größe	48	233	864	3301	12414	48121
Laufzeit/s	0.6	2	12	74	672	17518
Größe auf Schnitt 9						23
Größe auf Schnitt 8					20	49
Größe auf Schnitt 7				17	42	118
Größe auf Schnitt 6			14	35	94	978
Größe auf Schnitt 5		11	28	68	460	2455
Größe auf Schnitt 4	8	21	48	210	894	3933
Größe auf Schnitt 3	**14**	26	95	339	1286	5060
Größe auf Schnitt 2	13	**44**	132	453	1668	6391
Größe auf Schnitt 1	13	43	**152**	**540**	**1969**	**7468**
Größe auf Schnitt 0	7	16	32	86	321	1237
Größe von $\chi(D, R^{(0)})$	6	16	29	45	64	86

des Verfahrens abgearbeitet werden. Da das Verfahren auf der Substitution basiert, wird zuerst der Schnitt mit maximaler Nummerierung besucht.

Bei einer genaueren Betrachtung der Messwerte lassen sich bereits folgende Schlüsse ziehen:

1. Wie erwartet lässt sich die Transformationsfunktion (Schnitt $n + 1$) effizient darstellen.
2. Die charakteristische Funktion $\chi(D, R^{(0)})$ lässt sich effizient darstellen.
3. Auf den Schnitten 1 bis n ist mit den gewählten Parametern keine effiziente Darstellung möglich.
4. Während der Substitution ist ein deutlicher Peak in der DD-Größe (maximale Größe) erkennbar.

Die Messwerte zeigen deutlich, dass mit diesen Parametern nur kleine Dividierer Schaltkreise verifiziert werden können. Bereits bei einem Dividierer für 8 Bit Zahlen ist ein Zeitaufwand von fast 5 Stunden notwendig.

5.3.2. Weitere Untersuchungen

Durch Variation der Parameter des Verfahrens wurde versucht, die Darstellungsgröße auf den Schnitten zu minimieren. Dabei konnte teilweise auf bekannte DD-Techniken (siehe z.B. [11] und [22]) zurückgegriffen werden. Die wichtigste Technik ist das Sifting:

Variablensifting: Beim Variablensifting wird versucht eine „gute“ Variablenordnung zu finden. Hierzu werden die Variablen absteigend nach Levelgröße (Breite des DD) sortiert und anschließend die Variable mit der maximalen Levelgröße durch lokale Vertauschung an diejenige Postition gebracht, bei der die Größe des DD (lokal) minimal wird. Dieser Schritt kann iterativ für die weiteren Variablen angewendet werden.

DTL-Sifting: Bei OKFDDs und K*BMDs ist neben der Variablenordnung auch die Wahl der Dekompositionstypen entscheident. Deswegen wurde der Sifting-Algorithmus erweitert, so dass während des Verfahrens alle Kronecker Dekompositionen für jede Variable getestet werden. Wieder wird die Kombination mit (lokal) minimaler Größe des DD gewählt.

Leider konnte weder die Darstellungsgröße noch die Programmlaufzeit deutlich gemindert werden. Um ein effizientes Verfikationsverfahren angeben zu können sind deshalb weitere Modifikationen am Verfahren notwendig.

5.4. Der vollständige nonrestoring Dividierer

5.4.1. Die breite Variante des nonrestoring Dividierers

Um herauszufinden, warum die Funktionen auf den Stufen 1 bis n nicht wie erhofft effizient darstellbar sind, wurde eine Variante des nonrestorings Dividierers entworfen, bei der die Stufen keine konstante Bitbreite haben (siehe Abschnitt 3.1.3 auf Seite 47). Der Entwurf verwendet CAS-Zellen, die auf einer Implementierung der in Satz 2.3.3 auf Seite 38 beschriebenen Addition aufbauen. Zusätzlich wird ein „vollwertiger“ Shift ohne Überlauf realisiert. In Abb. A.4 auf Seite 107 ist eine Implementierung für 4 Bit Zahlen dargestellt. Die Hauptmerkmale dieses Dividierers sind:

- Als Eingangsbedingung für die Division von ganzen Zahlen genügt
$$D \neq 0 \,. \tag{5.5}$$
- Die CAS-Zellen i mit $0 < i \leq n$ (Stufe i) berechnen nach Konstruktion:
$$q_{n-i}(R^{(i)} - 2^{n-1-i} \cdot D) + (1 - q_{n-i})(R^{(i)} + 2^{n-1-i} \cdot D) \tag{5.6}$$
- Da die erste CAS-Zelle immer subtrahiert, vereinfacht sich die Funktion der CAS-Zelle 0 zu
$$R^{(0)} - 2^{n-1} \cdot D \,. \tag{5.7}$$
- Die letzte Stufe (Rückaddition) berechnet nach Konstruktion:
$$(1 - q_0)(R^{(n)} + D) + q_0 \cdot R^{(n)} \tag{5.8}$$

5.4.2. Betrachtung der Substitution

Aufgrund der speziellen Konstruktion des Dividierers ist eine theoretische Analyse der auf den Schnitten darzustellenden Funktionen des transformierten Schaltkreises möglich. Insbesondere gilt folgender Satz:

Satz 5.4.1 (Darstellung auf dem Schnitt)
Auf den Schnitten n bis 1 wird die Funktion

$$\left(\sum_{j=n+1-i}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(i)} + 2^{(n-i)} \cdot D \tag{5.9}$$

dargestellt. □

Beweis: Durch strukturelle Induktion über die Anzahl der Stufen:

Verankerung: Auf Schnitt $n+1$ wird die Transformationsfunktion

$$D \cdot Q + R^{(n+1)} \tag{5.10}$$

dargestellt. Durch Einsetzen von Gleichung 5.8 ergibt sich für die auf Schnitt $i = n$ dargestellte Funktion:

$$\begin{aligned} D \cdot Q + R^{(n+1)} &= D \cdot Q + \Big((1 - q_0) \cdot (R^{(n)} + D) + q_0 \cdot R^{(n)}\Big) \\ &= D \cdot Q + \Big(R^{(n)} + D - q_0 \cdot R^{(n)} - q_0 \cdot D + q_0 \cdot R^{(n)})\Big) \\ &= D \cdot Q - q_0 \cdot D + R^{(n)} + D \\ &= \left(\sum_{j=1}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(n)} + D \\ &= \left(\sum_{j=1}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(n)} + 2^0 \cdot D \end{aligned}$$

Induktion ($i+1 \to i$): Auf Schnitt $i+1$ wird nach Induktionsvoraussetzung die Funktion

$$\left(\sum_{j=n-i}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(i+1)} + 2^{(n-i-1)} \cdot D \tag{5.11}$$

dargestellt. Einsetzen der Gleichung 5.6 für Stufe i ergibt:

$$\begin{aligned}
& \left(\sum_{j=n-i}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(i+1)} + 2^{(n-i-1)} \cdot D \\
= & \left(\sum_{j=n-i}^{n-1} 2^j \cdot q_j\right) \cdot D + 2^{(n-i-1)} \cdot D \\
& + \Big(q_{n-i}(R^{(i)} - 2^{n-1-i} \cdot D) + (1 - q_{n-i})(R^{(i)} + 2^{n-1-i} \cdot D)\Big) \\
= & \left(\sum_{j=n-i}^{n-1} 2^j \cdot q_j\right) \cdot D + \Big(R^{(i)} - 2^{n-i} \cdot q_{n-i} \cdot D + 2^{n-1-i} \cdot D\Big) \\
& + 2^{(n-i-1)} \cdot D \\
= & \left(\sum_{j=n+1-i}^{n-1} 2^j \cdot q_j\right) \cdot D + R^{(i)} + 2^{(n-i)} \cdot D
\end{aligned}$$

Dabei gelte $\sum_{n}^{n-1} 2^j \cdot q_j := 0$. ■

Korollar 5.4.1 (Darstellung auf Schnitt 0)
Auf Schnitt 0 *wird*

$$R^{(0)} \tag{5.12}$$

dargestellt. □

Beweis: Nach Satz 5.4.1 wird auf Schnitt 1 die Funktion

$$R^{(1)} + 2^{(n-1)} \cdot D \tag{5.13}$$

dargestellt. Einsetzen von Gleichung 5.7 ergibt:

$$\begin{aligned}
R^{(1)} + 2^{(n-1)} \cdot D &= (R^{(0)} - 2^{n-1} \cdot D) + 2^{(n-1)} \cdot D \\
&= R^{(0)}
\end{aligned}$$

Wird die breite Variante der nonrestoring Division um einen Probeschaltkreis, der $Q \cdot D + R^{(n+1)}$ berechnet, erweitert, so berechnet der durch diese Transformation enstehende Gesamtschaltkreis wie erwartet die Identität auf $R^{(0)}$. ■

Tabelle 5.2. DD-Größen bei der Substitution der breiten nonrestoring Division

Bitbreite	3	4	5	6	7	8
maximale Größe	20	25	31	37	43	49
Laufzeit/s	0.6	2.0	2.6	2.4	3.6	5.4
Größe auf Schnitt 9						39
Größe auf Schnitt 8					34	43
Größe auf Schnitt 7				29	37	40
Größe auf Schnitt 6			24	31	34	37
Größe auf Schnitt 5		19	25	28	31	34
Größe auf Schnitt 4	14	19	22	25	28	31
Größe auf Schnitt 3	13	16	19	22	25	28
Größe auf Schnitt 2	10	13	16	19	22	25
Größe auf Schnitt 1	6	8	10	12	14	16
Größe auf Schnitt 0	2	3	4	5	6	7

Tabelle 5.3. Max. DD-Größen bei der Substitution der breiten nonrestoring Division

Bitbreite	3	4	5	6	7	8	12	16	24	32	48
maximale Größe	20	25	31	37	43	49	73	97	145	193	289
maximale Größe auf Schnitt	14	19	25	31	37	43	67	91	139	187	283

Bei diesem Dividierer gilt die Vermutung, dass auf den Schnitten Funktionen dargestellt werden, die der Transformationsfunktion sehr ähnlich sind. Da aus den in Abschnitt 5.3 durchgeführten Untersuchungen bereits bekannt ist, dass sich die Transformationsfunktion effizient darstellen lässt, sollte auch die Substitution dieses Dividierers effizient möglich sein. Um dies zu untersuchen, wurde wieder mit denselben Parametern (alle Knoten $pD_{\mathbf{Z}}$, verzahnte Variablenordnung) Messungen durchgeführt.

Die in Tabelle 5.2 wiedergegebenen Messwerte bestätigen die Vermutung sehr deutlich, dass bei dieser Variante der nonrestoring Division ein effizienter Aufbau des transformierten Schaltkreises möglich ist. Wieder sind in der Tabelle die Laufzeit, der maximale Peak während der Substitution und die Darstellungsgrößen auf den Schnitten aufgeführt. Bei dieser Vorgehensweise benötigt der Dividierer für 8 Bit Zahlen weniger als 6 Sekunden im Vergleich zu knapp 5 Stunden, eine Verbesserung um mehr als einen Faktor 8000. Auch bei der Betrachtung der maximalen DD Größe ergibt sich ein bemerkenswerter Unterschied, beträgt der Peak in der breiten Variante (wieder für 8 Bit Zahlen) gerade mal 49 Knoten, sind bei der Verifikation des „normalen“ Dividierers 48121 Knoten notwendig. Dies stellt immerhin noch eine Verbesserung um nahezu einen Faktor 1000 dar.

In Tabelle 5.3 sind nochmals die Größe des Peaks und die Größe der maximalen Darstellung auf einem Schnitt für die Substitution des breiten Dividierers dargestellt. Deutlich zeigt sich, dass mit dieser Vorgehensweise auch große Bitbreiten verarbeitet werden können. So beträgt der Peak bei einer Bitbreite von 48 nur 289 Knoten. Für die durchgeführten Messungen gilt sogar, dass sich die Größe des Peaks durch eine einfache lineare Funktion abschätzen lässt.

Vermutung 5.4.1 (DD-Größe während der Substitution) Bei der vorgestellten breiten Variante des nonrestoring Dividierers gilt: Für $n > 3$ lässt sich die maximale DD-Größe während der Substitution des transformierten Schaltkreises nach oben durch die Funktion $6n + 1$ abschätzen. □

5.4.3. Bildberechnung

Nachdem in diesem Fall die Überprüfung der Bedingung

$$D \cdot Q + R^{(n+1)} = R^{(0)} \tag{5.14}$$

effizient möglich ist, muss zur vollständigen Verifikation noch die Bedingung

$$0 \leq R^{(n+1)} < D \tag{5.15}$$

nachgewiesen werden. Dies kann z.B. durch eine Bildberechnung des Dividierers realisiert werden. Um während der Bildberechnung nicht die Operationen $\div$ und mod darstellen zu müssen, wird stufenweise vorgegangen. Dabei wird, unter Berücksichtigung des zugehörigen Definitionsbereiches[5], für jede Stufe das Bild berechnet. Bei der Bildberechnung gilt:

1. Der Definitionsbereich der ersten CAS-Zelle (Stufe 0) ist mit dem spezifizierten Definitionsbereich des Dividierers identisch. Für die Bildberechnung wird der durch die Eingangsbedingung

   $$0 < D < 2^n \quad \text{und} \quad 2^{n-1} \leq R^{(0)} < 2^{(n-1)} \cdot D \tag{5.16}$$

   beschriebene Definitionsbereich gewählt.

2. Das Bild img_i von Stufe i ist gleichzeitig der Definitionsbereich von Stufe $i + 1$.

Werden die Bild- und Definitionsbereiche als charakteristische Funktionen dargestellt, so ergibt sich folgender Algorithmus zur Berechnung des exakten Bildes bei der nonrestoring Division:

Eingabe: Netzliste des Dividierers.

[5] Der Definitionsbereich ist als charakteristische Funktion χ gegeben.

Ausgabe: Das exakte Bild des Dividierers, sowie das Bild auf jedem Schnitt. Die Bilder werden als charakteristische Funktionen in Form von OBDDs repräsentiert.

Algorithmus: Sei $\chi(D, R^{(i)})$ die charakteristische Funktion (der Definitionsbereich) auf Schnitt i, weiter sei $f_i(D, R^{(i)})$ die Funktion, die durch Stufe i berechnet wird.

1. Beginne auf Schnitt 0, also $i = 0$;
2. Bestimme die OBDD-Darstellung $f_i(D, R^{(i)})$ der Stufe i. Dies kann durch Substitution oder Vorwärtsaufbauen geschehen.
3. Sei $f_i^j(D, R^{(i)})$ der Bit-Level Ausgang der Stufe i, der $r_j^{(i+1)}$ berechnet. Führe die eigentliche Bildberechnung (siehe Definition 2.1.7 auf Seite 21 und Definition 2.1.8) durch, hierzu muss

$$\mathrm{img}_i = \exists R^{(i)} \bullet \chi(D, R^{(i)}) \cdot f_i(D, R^{(i)}) \cdot \prod_{i=0}^{n-1+2\cdot i} \left(\overline{f_i^j(D, R^{(i)}) \oplus r_j^{(i)}} \right) \tag{5.17}$$

   berechnet werden. Weitere Hinweise zur effizienten Implementierung der Bildberechnung bei OBDDs können in [28] nachgelesen werden.
4. Falls $i < n+1$ gilt, setze $\chi(D, R^{(i+1)}) := \mathrm{img}_i$ und fahre bei Schritt 2 fort.

Durch dieses Verfahren müssen nur die DDs für die charakteristischen Funktionen bzw. für die Bilder und die Stufen selbst aufgebaut werden. Die Bildberechnung wird mit derselben Variablenordnung wie die Substitution durchgeführt, allerdings werden als Datenstrukturen OBDDs verwendet.

Für die Verifikation des Dividierers genügt eine Bereichsabschätzung des Bildes, da nur die Bedingung

$$0 \leq R^{(n+1)} < D \tag{5.18}$$

gezeigt werden muss.

Satz 5.4.2 (Bildbereich auf den Stufen)
Für die breite nonrestoring Division gilt unter Berücksichtigung der Vorbedingung

$$0 < D < 2^n \quad \textit{und} \quad 2^{n-1} \leq R^{(0)} < 2^{(n-1)} \cdot D \tag{5.19}$$

für das auf Schnitt i mit $0 < i \leq n$ dargestellte Bild, $\text{img}_i = \text{img}(R^{(0)})$ *die Bereichsabschätzung*

$$-2^{n-i} \cdot D \leq \text{range}(\text{img}_i) < 2^{n-i} D \,. \tag{5.20}$$

Für Schnitt $n+1$ gilt die Bereichsabschätzung

$$0 \leq R^{(n+1)} < D \tag{5.21}$$

□

BEWEIS: Durch strukturelle Induktion über die Anzahl der Stufen:

Verankerung: Nach Gleichung 5.7 auf Seite 66 folgt aus der Vorbedinung für das Bild auf Schnitt 1:

$$\begin{array}{rccccc} & 2^{n-1} & \leq & R^{(0)} & < & 2^{n-1}D \\ \Leftrightarrow & 2^{n-1} - 2^{n-1}D & \leq & R^{(0)} - 2^{n-1}D & < & 2^{n-1}D - 2^{n-1}D \\ \Leftrightarrow & 2^{n-1}(1-D) & \leq & R^{(1)} & < & 0 \\ \Leftrightarrow & -2^{n-1}D & \leq & R^{(1)} & < & 0 \end{array}$$

Da $R^{(1)} < 0$ gilt, addiert die zweite Stufe. Somit folgt für den Bereich von img_2:

$$\begin{array}{rccccc} \Leftrightarrow & -2^{n-1}D + 2^{n-2}D & \leq & R^{(1)} + 2^{n-2}D & < & 2^{n-2}D \\ \Leftrightarrow & -2^{n-2}D & \leq & R^{(2)} & < & 2^{n-2}D \end{array}$$

Induktion ($i \to i+1$): Nach Induktionsvorrausetzung gilt für $1 < i < n$:

$$-2^{n-i}D \leq R^{(i)} < 2^{n-j}D$$

Es ergeben sich zwei Fälle:

1. $-2^{n-i}D \leq R^{(i)} < 0$, also $R^{(i+1)} = R^{(i)} + 2^{n-(i+1)}D$

$$\begin{array}{rccccc} & -2^{n-i}D & \leq & R^{(i)} & < & 0 \\ \Leftrightarrow & -2^{n-i}D + 2^{n-(i+1)}D & \leq & R^{(i)} + 2^{n-(i+1)}D & < & 2^{n-(i+1)}D \\ \Leftrightarrow & -2^{n-(i+1)}D & \leq & R^{(i+1)} & < & 2^{n-(i+1)}D \end{array}$$

2. $0 \leq R^{(i)} < 2^{n-j}D$, also $R^{(i+1)} = R^{(i)} - 2^{n-(i+1)}D$

$$\begin{array}{rccccc} & 0 & \leq & R^{(i)} & < & 2^{n-j}D \\ \Leftrightarrow & -2^{n-(i+1)}D & \leq & R^{(i)} - 2^{n-(i+1)}D & < & 2^{n-j}D - 2^{n-(i+1)} \\ \Leftrightarrow & -2^{n-(i+1)}D & \leq & R^{(i+1)} & < & 2^{n-(j+1)}D \end{array}$$

Tabelle 5.4. Bildberechnung des breiten nonrestoring Dividierers

Bitbreite	3	4	5	6	7	8
maximale Größe	38	94	148	202	256	310
Laufzeit/s	0.4	0.6	1.0	1.4	2.2	3.2
Größe auf Schnitt 0	6	12	18	24	30	36
Größe auf Schnitt 1	6	11	16	21	26	31
Größe auf Schnitt 2	14	22	30	38	46	54
Größe auf Schnitt 3	18	27	33	39	45	51
Größe auf Schnitt 4	12	32	41	47	53	59
Größe auf Schnitt 5		23	46	55	61	67
Größe auf Schnitt 6			33	60	69	75
Größe auf Schnitt 7				43	74	83
Größe auf Schnitt 8					53	88
Größe auf Schnitt 9						63

Die Behauptung für die Darstellung auf Stufe $n+1$ ist für den Fall $0 \leq R^{(n)}$ erfüllt, da in diesem Fall die Stufe zur Rückaddition die Identität berechnet. Für den Fall $R^{(n)} < 0$ gilt genauer

$$-D \leq R^{(n)} < 0$$

und die Stufe $n+1$ berechnet $R^{(n+1)} = R^{(n)} + D$, somit gilt

$$0 \leq R^{(n+1)} < D$$

■

Aus diesem Satz ergibt sich Korollar 5.4.2. Das Korollar überträgt die in Satz 3.1.1 auf Seite 48 getroffene Aussage, das auf den Schnitten n-Bit Zahlen zur Darstellung genügen, auf den breiten Dividierer. Beim breiten Dividierer berechnen die höherwertigen Bits lediglich eine Sign Extension.

Korollar 5.4.2
Unter den Voraussetzungen für Satz 5.4.2 auf Stufe $i \in \{1, n\}$,

$$r^{(i)}_{n-1+2\cdot i} = r^{(i)}_{n-1+2\cdot i-1} = \cdots = r^{(i)}_{n} = r^{(i)}_{n-1} \,, \tag{5.22}$$

gilt speziell für das Bild auf Stufe $n+1$:

$$r^{(i)}_{3n} = r^{(i)}_{3n} = \cdots = r^{(i)}_{n} = r^{(i)}_{n-1} = 0 \tag{5.23}$$

□

Tabelle 5.5. Übersicht der Bildberechnung beim breiten nonrestoring Dividierer

Bitbreite	3	4	5	6	7	8	12	16	24	32	48
maximale Größe	38	94	148	202	256	310	526	742	1174	1606	2470
maximales Bild	18	32	46	60	74	88	144	200	312	424	648
Bildgröße	12	23	33	43	53	63	103	143	223	303	463

In Tabelle 5.4 ist die Größe des maximalen Peaks während der Bildberechnung, sowie die Größe der Bilder auf den jeweiligen Schnitten angegeben. Die Messwerte auf den Schnitten sind wieder in der Reihenfolge angeben, in der sie vom Algorithmus abgearbeitet werden. Die Bildberechnung beginnt mit den Eingängen, also Schnitt 0, des Schaltkreises. Zusätzlich ist die auf einer Ultra Sparc 1 notwendige Rechenzeit zur Bestimmung des exakten Bildes des Dividierers aufgeführt. Wieder zeigt sich deutlich, dass auch die Größe des Peaks effizient handhabbar ist.

Tabelle 5.5 fasst nochmal die Größe des maximalen Peaks, die Größe des maximalen Bildes auf einer Stufe sowie die Größe des Bildes des Dividierers zusammen.

Vermutung 5.4.2 (Größe während der Bildberechnung) Für die vorgestellte breite Variante des nonrestoring Dividierers gilt: Die maximale DD-Größe während der Substitution des transformierten Schaltkreises lässt sich nach oben durch die Funktion $54n - 122$ abschätzen. □

5.5. Erste Rückschlüsse zur Verifikation der schmalen Implementierung

Nachdem bei der breiten Variante des Dividierers eine sehr effiziente vollautomatische Verifikation möglich ist, stellt sich die Frage, wie die neu gewonnenen Erkenntnisse helfen, reale Designs zu verifizieren.

Der Hauptunterschied zwischen den beiden Varianten der nonrestoring Division liegt in der Funktion der CAS-Zellen. Beim breiten Dividierer ist gewährleistet, dass bei dem Ergebnis der CAS-Zellen kein eventuell enstehender Überlauf verloren gehen kann. Genauer basieren die CAS-Zellen in diesem Design auf Korollar 2.3.3 auf Seite 38, so dass im mathematischen Sinne die Addition und Subtraktion in den ganzen Zahlen $\mathbf{Z}$ berechnet wird. Dagegen ergibt eine Analyse des Designs des schmalen Dividierers zusammen mit der Aussage von Satz 2.4.1 auf Seite 39, dass dort kongruent modulo 2^i gerechnet wird. Sei $f_i(D, R^{(i)})$ die von der CAS Zelle i der schmalen nonrestoring Division berechnete

Funktion, dann gilt lediglich:

$$f_i(D, R^{(i)}) \equiv q_{n-i} \cdot \left(R^{(i)} - 2^{n-1-i} \cdot D\right) + (1 - q_{n-i}) \cdot \left(R^{(i)} + 2^{n-1-i} \cdot D\right) \pmod{2^i} \tag{5.24}$$

Weiter gilt auch die Ungleichung:

$$f_i(D, R^{(i)}) \neq q_{n-i} \cdot \left(R^{(i)} - 2^{n-1-i} \cdot D\right) + (1 - q_{n-i}) \cdot \left(R^{(i)} + 2^{n-1-i} \cdot D\right) \tag{5.25}$$

Dieser funktionale Unterschied scheint zu einer exponentiellen Diskrepanz in der zur Verifikation benötigen DD-Größe zu führen.

Um ein Verfahren zur Verifikation der „normalen“ nonrestoring und restoring Division zu entwickeln, bieten sich zwei Ideen an:

1. Es wird die Tatsache ausgenutzt, dass sich der schmale Dividierer in natürlicher Weise in die verbreiterte Konstruktion einbetten lässt, d.h. zur Verifikation wird der Dividiererschaltkreis „virtuell“ verbreitert. Aus Korollar 5.4.2 ergibt sich direkt, dass beim so erweiterten, korrekten Dividierer die Erweiterung nur eine Sign-Extension durchführt. Somit folgt aus der Korrektheit der verbreiterten Variante die Korrektheit des schmalen Dividierers. Hierbei wird der schmale Dividierer durch eine zusätzliche Erweiterung in einen zum breiten Dividierer mit Probeschaltkreis äquivalenten Schaltkreis transformiert. Ein auf diese Grundidee aufbauendes Verfahren wird in Kapitel 6 vorgestellt.

2. Werden die führenden Bits bei der Berechnung der breiten nonrestoring Division betrachtet, so gilt nach Korollar 5.4.2 , dass nur die niederwertigsten n Bit maßgebend sind, da die höherwertigen Bits lediglich eine Sign-Extension darstellen. Dadurch kann die Darstellung des breiten Dividierers als eine Art DC-Belegung der DD-Darstellungen des schmalen Dividierers (auf den Schnitten) aufgefasst werden. Anders formuliert, die Funktionalität des breiten und des schmalen Dividierers unterscheidet sich auf den Schnitten nur für Eingaben, bei denen die Bedingung an die Eingänge (Gleichung 3.4 auf Seite 43) verletzt ist. Durch geschickte DC-Belegung kann versucht werden, die Darstellungen auf den Stufen klein zu halten. Erste Ergebnisse von Untersuchungen, die in diese Richtung zielen, sind in Kapitel 7 dargelegt.

6. DD-basierte Verifikation von Dividierern

In Kapitel 5 wurde eine spezielle Version der nonrestoring Division vorgestellt, bei der die Verifikation mit Platz $O(n)$ möglich ist. In diesem Kapitel sollen diese Ergebnisse verallgemeinert werden.

Ein wichtiger Schritt des vorgestellten Verfahrens ist die Transformation des Dividierer-Schaltkreises. In diesem Fall besteht die Transformation aus zwei Schritten: Erst wird jede Stufe erweitert, so dass sie einer Stufe des vorgestellten „breiten“ Dividierers entspricht und anschließend wird der Gesamtschaltkreis um die „Proberechnung“ ergänzt.

6.1. Notation

Um die Erweiterung der einzelnen Stufen möglichst allgemein beschreiben zu können, werden folgende Funktionen benötigt:

CAS Die CAS-Zelle berechnet eine Funktion vom Typ: $\mathbf{B}^n \times \mathbf{B}^n \times \mathbf{B} \to \mathbf{B}^n$. Sei $c \in \mathbf{B}$ und seien $a, b \in \mathbf{B}^n$ binäre Vektoren mit $A = [a]$ und $B = [b]$ so berechnet die CAS-Zelle

$$\left[p_0^{n-1}\left(c \cdot (A - B) + (1 - c) \cdot (A + B)\right)\right] . \tag{6.1}$$

Aus den vorherigen Kapiteln ist bekannt, dass ein n-Bit nonrestoring Dividierer in der Regel aus n CAS-Zellen der Bitbreite n und einer Zelle für die Rückaddition aufgebaut ist.

CAS$_{\mathbf{Z}}$ Die CAS$_{\mathbf{Z}}$-Zelle basiert auf Korollar 2.3.3 auf Seite 38. Es handelt sich um eine Funktion vom Typ $\mathbf{B}^n \times \mathbf{B}^n \times \mathbf{B} \to \mathbf{B}^{n+1}$. Sei $c \in \mathbf{B}$ und seien $a, b \in \mathbf{B}^n$ binäre Vektoren mit $A = [a]$ und $B = [b]$ so berechnet die CAS$_{\mathbf{Z}}$-Zelle

$$c \cdot (A - B) + (1 - c) \cdot (A + B) . \tag{6.2}$$

Ein wie in Abschnitt 5.4.1 beschriebener n-Bit nonrestoring Dividierer besteht aus n CAS$_{\mathbf{Z}}$ Zellen und einer Zelle für die Rückaddition.

XCAS$_\mathbf{Z}$ Eine i-Bit breite XCAS$_\mathbf{Z}$-Zelle[1] erweitert eine n-Bit breite CAS-Zelle zu einer $n+i$-Bit breiten CAS$_\mathbf{Z}$-Zelle. Diese Erweiterung wird, wie in Abb. 6.1 gezeigt, durchgeführt.

Die XCAS$_\mathbf{Z}$-Zelle besteht dabei intern aus einer i-Bit breiten CAS$_\mathbf{Z}$-Zelle, die zusätzlich zum Auswahleingang s ein Eingangs-Carry c_{in} berücksichtigt. Dieses Eingangscarry wird aus den Eingängen der n-Bit CAS-Zelle berechnet und entspricht genau einem etwaigen Ausgangsübertrag der CAS-Zelle.

Der zu $R^{(i)}$ korrespondierende Eingang der CAS$_\mathbf{Z}$-Zelle wird mit $E^{(i)}$ gekennzeichnet. Er entspricht genau dem Ausgang der $CAS_\mathbf{Z}$-Zelle in der XCAS$_\mathbf{Z}$-Zelle der vorangegangenen Stufe. Bei dem zu D korrespondierenden Eingang werden alle Leitungen auf 0 gelegt. Aufgrund der Voraussetzung $0 < D$ entspricht dies einer Sign-Extension von D auf die Gesamtbitbreite der XCAS$_\mathbf{Z}$-Zelle.

Für diese Erweiterung ist für Stufe i neben dem Eingangsignal D und dem Ausgangssignal Q des Dividierers nur der Zugriff auf das höchstwertigste Bit des Partialrestes $R^{(i-1)}$ notwendig. In Abbildung A.5 auf Seite 109 ist eine mögliche Implementierung der XCAS$_\mathbf{Z}$-Zelle gezeigt.

6.2. Verfahren zur automatischen Verifikation von nonrestoring Dividierern

6.2.1. Der Grundalgorithmus

Folgendes Vorgehen ermöglicht eine vollständige Verifikation der nonrestoring Division mit geringstmöglichen manuellen Eingriffen.

Eingabe: Netzliste des nonrestoring Dividierers. In der Netzliste sind neben den Eingängen D und $R^{(0)}$ sowie den Ausgängen Q und $R^{(n+1)}$ auch die Busse, an denen die Partialreste $R^{(i)}$ anliegen, markiert.

Die Markierung der $R^{(i)}$-Busse für $0 < i < n+1$ stellt den einzigen manuellen Eingriff dar. Ansonsten läuft das Verfahren vollautomatisch ab!

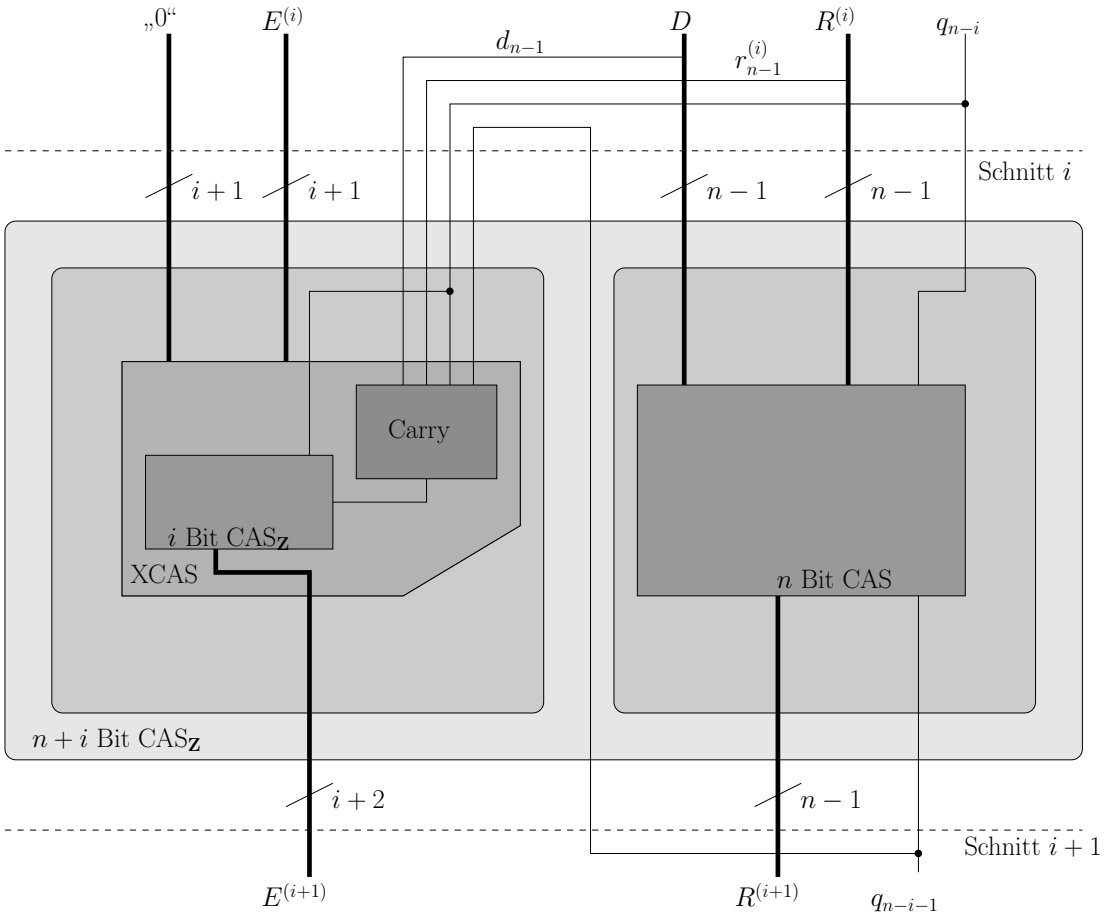
Ausgabe: Aussage über die Korrektheit des Dividieres. Im Falle eines fehlerhaften Schaltkreises ist eine ungefähre Abschätzung des Fehlerortes möglich.

Algorithmus:

1. Erweitere die einzelnen Stufen, so dass ein „virtueller“ breiter Dividierer entsteht. Hierzu wird die CAS-Zelle von Stufe i zu einer

[1] eXtensional CAS$_\mathbf{Z}$-Zelle

Abbildung 6.1. Erweiterung einer Stufe am Beispiel einer CAS-Zelle



XCAS$_{\mathbf{Z}}$-Zelle mit Bitbreite

$$\begin{cases} n+2i & \text{für } 0 \leq i < n \\ 3n-2 & \text{für } i = n \end{cases} \tag{6.3}$$

erweitert. Durch die Markierung der $R^{(i)}$-Busse ist diese Erweiterung ohne manuelle Eingriffe möglich.

2. In einem weiteren Schritt wird die Variablenordnung auf den Schnitten festgelegt. Als geeignet hat sich die Verzahnung von D- und $R^{(i)}$-Bussen erwiesen. Aufgrund der Markierung der $R^{(i)}$-Busse kann die Sortierung automatisch erfolgen.
3. Führe eine „stufenweise“ Bildberechnung, wie in Abschnitt 5.4.3 auf Seite 70 vorgestellt, auf dem erweiterten Dividierer durch. Auf Schnitt i mit $R^{(i)}$ und $E^{(i)}$ muss gelten:

   $$e_i^{(i)} = \cdots = e_0^{(i)} = r_{n-1}^{(i)} \tag{6.4}$$

   Speziell auf Schnitt $n+1$ muss die Bedingung

   $$e_{2n-1}^{(n+1)} = \cdots = e_0^{(n+1)} = r_{n-1}^{(n+1)} = 0 \tag{6.5}$$

   gelten. Zudem muss auch die Ungleichung

   $$R^{(n+1)} < D \tag{6.6}$$

   überprüft werden.
4. Erweitere „virtuell“ den (verbreiterten) Dividierer zusätzlich um die Proberechnung $Q \cdot D + [E^{(n+1)}, R^{(n+1)}]$, wobei

   $$[E^{(n+1)}, R^{(n+1)}] = [e_{2n-1}^{(n+1)}, \ldots, e_0^{(n+1)}, r_{n-1}^{(n+1)}, \ldots, r_0^{(n+1)}] \tag{6.7}$$

   gilt.
5. Durch Substitution (Rückwärtsaufbau) wird gezeigt, dass der Gesamtschaltkreis, bestehend aus dem verbreiterten Dividierer und der Proberechnung, die Identität darstellt. Durch die Markierung der $R^{(i)}$-Busse kann leicht sichergestellt werden, dass im Verlauf der Substitution die Stufen berücksichtigt werden.

   Am Ende des Verfahrens kann durch einen einfachen Isomorphie-Test nachgeprüft werden, dass der Gesamtschaltkreis $R^{(0)}$ darstellt. Zusätzlich kann auch effizient auf jeder Stufe die dargestellte Funktion mit der Soll-Funktion (siehe Satz 5.4.1 auf Seite 67) verglichen werden.

6.2.2. Eigenschaften des Verfahrens

Die Korrektheit des Verfahrens kann direkt aus den Ausführungen in Abschnitt 5.4 auf Seite 66 gefolgert werden. Hierbei ist besonders zu beachten, dass die Erweiterung $E^{(i)}$ aus Schritt 3 nur eine Sign-Extension des aktuellen Partialrestes darstellt. Nach Korollar 5.4.2 auf Seite 73 gilt für den korrekten Dividierer mit $E^{(i)} = [e_j^{(n)}, \ldots, e_0^{(i)}]$ und $R^{(i)} = [r_{n-1}^{(i)}, \ldots, r_0^{(i)}]$:

$$e_j^{(i)} = \cdots = e_0^{(i)} = r_{n-1}^{(i)} \,. \tag{6.8}$$

Diese Bedingung kann effizient durch eine Traversierung des darstellenden Graphen überprüft werden. Ist diese Bedingung auf Schnitt i verletzt, so kann daraus bereits gefolgert werden, dass eine der Stufen 0 bis i nicht korrekt implementiert ist. Mit sehr hoher Wahrscheinlichkeit liegt der Fehler in der Stufe i oder $i-1$. Anzumerken bleibt, dass auf Schnitt $n+1$ zudem $r_n^{(n+1)} = 0$ gilt und das Bild zusätzlich die Bedingung $0 \leq R^{(n+1)} < D$ erfüllen muss.

Auch bei der Substitution kann effizient auf jedem Schnitt ein Vergleich der dargestellten Funktion mit der im korrekten Fall dargestellten Funktion durchgeführt werden, um eine Fehlerlokalisation durchzuführen.

Das vorgestellte Verfahren bietet zudem die Möglichkeit, bei einem fehlerhaften Verhalten des Schaltkreises effizient die Eingaben zu bestimmen, bei denen Implementierung und Spezifikation abweichende Ergebnisse liefern. Wie bei den üblichen DD-basierten Ansätzen (siehe [10]) kann diese Eingabemenge durch eine $\oplus$-Operation ermittelt werden. Zu beachten ist, dass dieser Schritt sowohl für die Bildberechnung als auch für das Ergebnis der Substitution durchgeführt werden muss, um alle Eingaben zu erhalten, die ein Fehlverhalten hervorrufen.

Eine weitere wichtige Eigenschaft des Verfahrens ist die Robustheit gegenüber der genauen Implementierung der CAS-Zelle. In Abschnitt 6.2.4 und 6.2.5 wird eine Variante vorgestellt, die eine lineare DD-Größe bei beliebigen CAS-Implementierungen garantiert.

6.2.3. Messwerte

Um das Verfahren bewerten zu können, wurden auf einer Ultra Sparc 2 mit 1GB Hauptspeicher der nonrestoring-Dividierer nach dem beschriebenen Verfahren verifiziert. In Tabelle 6.1 sind die wichtigsten Messwerte wiedergeben:

- Die Eingabegröße wird in der ersten Zeile durch die Bitbreite des Dividierers angeben. Um einen Eindruck für die Größe der darstellenden Netzliste zu geben, ist in der zweiten Zeile die Anzahl der Gatter des Schaltkreises ohne Transformation aufgeführt.
- In der dritten Zeile ist die zur Verifikation benötigte Laufzeit in Sekunden angegeben.

- In der nächsten Zeile ist die (Peak-) Größe des maximalen DDs während des Verfahrens dokumentiert. Dieses Maximum tritt während der Bildberechnung auf.
- Die maximale Größe während der Substitution vermittelt einen Eindruck der Darstellungsgröße der Funktionen auf den Schnitten (siehe Satz 5.4.1 auf Seite 67).
- In der vorletzten Zeile ist die Größe des OBDD angegeben, der das Bild des verbreiterten Dividierers repräsentiert.
- Die letzte Zeile gibt die OBDD-Größe des maximale Bildes auf einem Schnitt wieder.

Diese Messwerte werden in Abb. 6.2 nochmals veranschaulicht. Hierbei ist deutlich der lineare Platzbedarf zu sehen. Die maximale DD-Größe während der Verifikation lässt sich, abhängig von der Bitbreite n für $n > 3$ exakt angeben als $54n - 112$. Für die betrachteten Bitbreiten ist ein akzeptables Laufzeitverhalten feststellbar.

Vermutung 6.2.1 Das beschriebene Verfahren zur Verifikation von Dividierer-Schaltkreisen erlaubt die Verifikation von n-Bit nonrestoring Dividierern mit der vorgestellten Struktur[2] mit einer maximalen DD-Größe von $54n - 112$ inneren Knoten. □

Da eine quadratisch wachsende Schaltkreisbeschreibung eingelesen und verarbeitet werden muss und die benötigten DD-Algorithmen im Worst-Case Verhalten exponentiell sind (siehe Tabelle 2.1 auf Seite 28 und 2.2 auf Seite 33), lässt sich hieraus keine Abschätzung für die Laufzeit ableiten. Allerdings lassen sich auf einer handelsüblichen Workstation mit dem hier vorgestellten Verfahren sogar 96-Bit Dividierer in weniger als 10 Stunden vollständig verifizieren.

6.2.4. Mögliche Erweiterungen und Optimierungen

Bisher wurde sowohl bei der Bildberechnung als auch bei der Substitution von einer flachen Netzliste ausgegangen, d.h. der komplette Schaltkreis ist nur aus zweiwertigen Funktionen aufgebaut. Experimente haben gezeigt, dass beim verbreiterten Dividierer folgende zwei Varianten bei der Substitution möglich sind, ohne eine Erhöhung des Platzbedarfes hervorzurufen:

Variante A

1. Für jeden Ausgang des Addierers innerhalb der CAS-Zelle wird ein K∗BMD aufgebaut, das von den Eingängen des Addierers abhängt und den Ausgang repräsentiert[3] (eine Funktion $f : \mathbf{B}^j \rightarrow \mathbf{B}$ mit $1 < j < 2n - 2$.).

[2] Gemeint ist hier der strukturelle Aufbau der nonrestoring Division in n CAS-Zellen und einer Zelle für die Rückaddition.

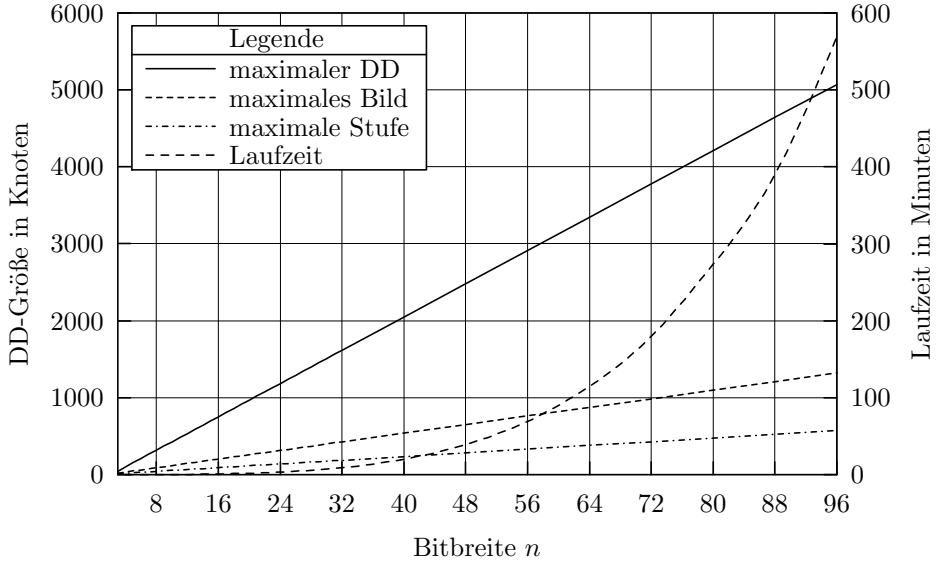
[3] Es wird die Projektion der Addition auf das gerade betrachtete Ausgangsbit berechnet.

Tabelle 6.1. Übersicht der Verifikation des nonrestoring Dividierers

Bitbreite	3	4	5	6	7	8	9	10
Schaltkreisgröße	65	112	171	242	325	420	527	646
Laufzeit/s	< 1	< 1	1	2	4	6	9	12
maximale Größe	41	104	158	212	266	320	374	428
Substitition (maximal)	23	31	39	48	57	66	66	75
Bildgröße	12	23	33	43	53	63	73	83
Bild (maximal)	20	34	48	62	76	90	104	118

Bitbreite	12	16	24	32	48	64	80	96
Schaltkreisgröße	920	1612	3572	6300	14060	24892	38796	55772
Laufzeit/s	21	59	199	540	2346	6868	16393	34167
maximale Größe	536	752	1184	1616	2480	3344	4208	5072
Substitution (maximal)	102	138	210	282	426	570	714	858
Bildgröße	103	143	223	303	463	623	783	943
Bild (maximal)	146	202	314	426	650	874	1098	1322

Abbildung 6.2. Auswertung der Verifikation des nonrestoring Dividierers



2. Die so aufgebauten K∗BMDs werden sequentiell in die Darstellung auf dem Schnitt substituiert.
3. Anschließend werden die $\oplus$-Gatter[4] der CAS-Zelle in die dargestellte Funktion substituiert.

Alternativ kann auch der als shared K∗BMD (eine Funktion $f' : \mathbf{B}^{2n+i+1} \rightarrow \mathbf{B}^{2n+i+3}$) dargestellte Addierer aufgebaut und die parallele Substitution des TUDD-Paketes (siehe [23]) verwendet werden.

Variante B

1. Für jeden Ausgang der Stufe i wird ein K∗BMD aufgebaut, das nur von den Eingängen der Stufe abhängt und den Ausgang repräsentiert.
2. Die so aufgebauten K∗BMDs werden sequentiell in die Darstellung auf dem Schnitt substituiert.

Alternativ kann auch die als shared K∗BMD dargestellte Stufe aufgebaut werden. Die Substitution der Stufe kann mit Hilfe der parallelen Substitution des TUDD-Paketes (siehe [23]) oder auch sequentiell, ein Ausgang nach dem anderen, durchgeführt werden.

Damit diese Varianten effizient durchfürbar sind, muss entweder jedes Bit (Ausgang) der Addition (bei Variante A) oder jedes Bit (Ausgang) der Stufe[5] (Variante B) effizient darstellbar sein.

6.2.5. Analyse des Platzverhaltens

Für eine Beispielimplementierung mit CAS-Zellen, die auf Carry-Ripple Addierern basieren, wurden bei den durchgeführten Experimenten ein linearer Platzbedarf benötigt. Das auch für CAS-Zellen, die auf anderen Addierern beruhen ein linearer Platzbedarf zu erwarten ist, lässt sich wie folgt motivieren:

Bildberechnung: Bei der Bildberechnung wird zuerst für jeden Ausgang der (breiten) CAS-Zelle ein OBDD aufgebaut. Da bei jeder Implementierung der CAS-Zelle dieselbe Funktionalität realisiert wird, ist nur der Aufbau dieser OBDDs kritisch. Es ist bekannt, dass sich Addierer und Subtrahierer effizient als OBDD darstellen und aufbauen lassen (siehe [5]). Wird die Variable s, die das Auswahlbit für Addition oder Subtraktion repräsentiert, als Top-Variable verwendet, ist klar, dass auch jeder Ausgang einer CAS-Zelle (jeder Stufe) effizient darstellbar[6] ist. Sobald die

[4] Im Falle der Rückaddition die ensprechenden $\wedge$-Gatter mit negiertem Eingang.
[5] genauer: einer CAS-Zelle
[6] In diesem Fall ist die „Addition“ gerade der *low*-Sohn der Top-Variablen und die „Subtraktion“ der *high*-Sohn der Top-Variablen.

Stufe dargestellt ist, lassen sich direkt die Ergebnisse aus Tabelle 6.1 für die Bildberechnung übernehmen. Somit ist also für jede (funktionsäquivalente) Realisierung der CAS-Zellen/Stufen ein linearer Platzbedarf für die Bildberechnung der nonrestoring Division zu erwarten.

Substitution: Auch bei der Substitution lässt sich aus den in Abschnitt 6.2.4 vorgestellten Erkenntnissen zusammen mit Satz 6.2.1 motivieren, dass sich der lineare Platzbedarf auch mit anderen Realisierungen der CAS-Zellen erreichen lässt. Nach Satz 6.2.1 ist es möglich, die Bitlevel-Ausgänge des Addierers in der CAS-Zelle komplett (z.B. durch Vorwärtsberechnen) aufzubauen und „auf einmal“ zu substituieren. Desweiteren zeigt dieser Satz, dass auch das Carry-Bit der Addition effizient dargestellt werden kann, so dass sich auch die XCAS-Zellen effizient verwenden lassen.

Weiter zeigt Vermutung 6.2.2, dass bei der Substitution zuerst jeder Ausgang einer Stufe als K*BMD aufgebaut werden kann (entweder „vorwärts“ mit Hilfe von Syntheseoperationen oder „rückwärts“ durch Substitution) und in die Darstellung auf dem Schnitt hineinsubstituiert werden kann, ohne dass sich die Platzabschätzung ändert.

Satz 6.2.1 (Darstellung der Ausgänge eines Addierers und Subtrahierers) *Seien* $X = \sum_{i=0}^{n-1} x_i, Y = \sum_{i=0}^{n-1} y_i$ *mit* $\{x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}\} \in \mathbf{B}^{2n}$*, so lassen sich alle Summen- und Übertragsbits der Addition* $X + Y$ *und der Subtraktion* $X - Y$ *mit der DTL* $d = (pD_{\mathbf{Z}}, \ldots, pD_{\mathbf{Z}})$ *und der Variablenordnung* $(x_{n-1}, y_{n-1}, \ldots x_0, y_0)$ *als shared K*BMD mit* $4n - 1$ *inneren Knoten darstellen.* □

Beweis: Durch strukturelle Induktion. Der Beweis wird nur für die Addition durchgeführt, für die Subtraktion ist der Beweis analog durchzuführen.

Für $n = 1$ ist die Aussage klar, betrachte nun getrennt den Induktionsschritt für die Darstellung der Summen- und Übertragsbits.

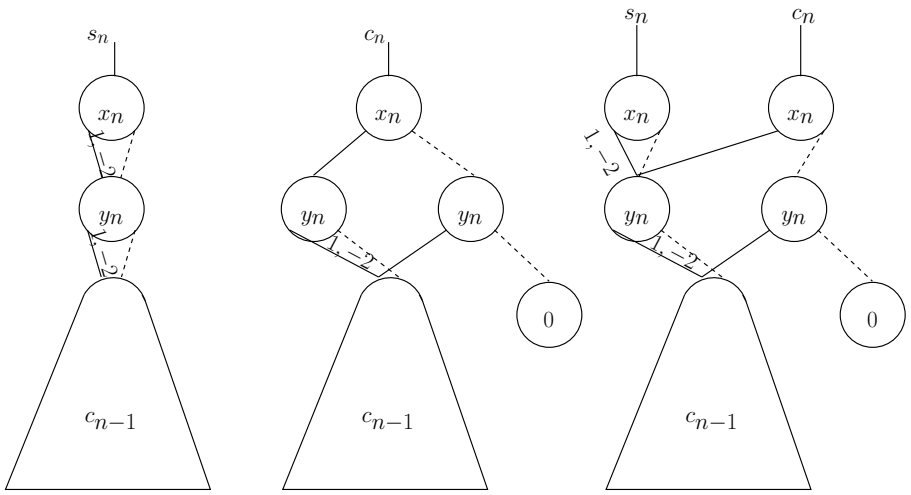
1. Darstellung des Summenbits s_n:
   Es gilt

$$s_n = x_n \oplus y_n \oplus c_{n-1} \tag{6.9}$$

   entsprechend ergibt sich durch Ausnutzen der Word-Level Darstellungen für die binären Operatoren:

$$\begin{aligned} s_n &= (x_n + y_n - 2 \cdot x_n \cdot y_n) + c_{n-1} - 2 \cdot ((x_n + y_n - 2 \cdot x_n \cdot y_n) \cdot c_{n-1}) \\ &= (c_{n-1} + y_n - 2 \cdot y_n \cdot c_{n-1}) + x_n \cdot (1 - 2 \cdot (c_{n-1} + y_n - 2 \cdot y_n \cdot c_{n-1})) \\ &= (c_{n-1} + y_n \cdot (1 - 2 \cdot c_{n-1})) + x_n \cdot (1 - 2 \cdot (c_{n-1} + y_n \cdot (1 - 2 \cdot c_{n-1}))) \end{aligned}$$

Abbildung 6.3. K∗BMD der Bit-Level-Darstellungen s_n und c_n



(a) s_n (b) c_n (c) s_n und c_n

Mit der gewählten DTL und Variabelsortierung ergibt sich daraus direkt die in Abb 6.3(a) gegebene DD-Darstellung und für die Annahme, dass c_{n-1} in $\Theta(n)$ darstellbar ist, folgt die Aussage direkt.

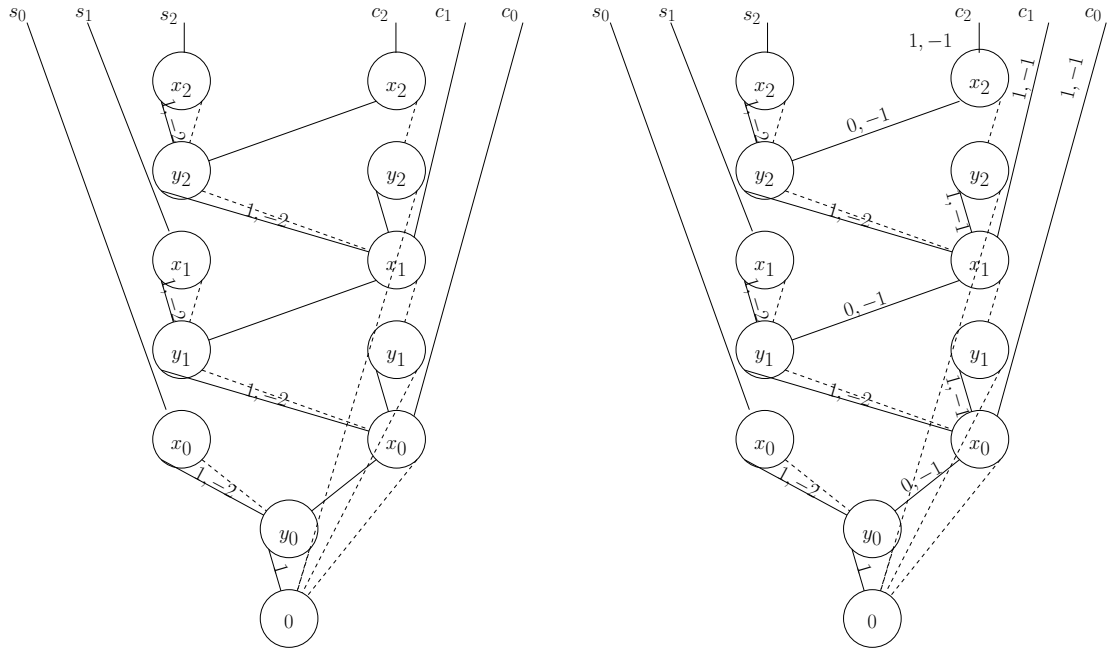
2. Darstellung des Übertragsbits c_n:
   Es gilt

$$c_n = (x_n \oplus y_n) \wedge c_{n-1} \vee x_n \wedge y_n \tag{6.10}$$

entsprechend ergibt sich durch Ausnutzen der Word-Level Darstellungen für die binären Operatoren:

$$\begin{aligned}
c_n &= x_n \cdot c_{n-1} + y_n \cdot c_{n-1} - 2 \cdot x_n \cdot y_n \cdot c_{n-1} + x_n \cdot y_n \\
&\quad -(x_n + y_n - 2 \cdot x_n \cdot y_n) \cdot c_n \cdot x_n \cdot y_n \\
&= x_n \cdot c_{n-1} + y_n \cdot c_{n-1} - 2 \cdot x_n \cdot y_n \cdot c_{n-1} + x_n \cdot y_n \\
&\quad -(x_n^2 \cdot y_n + x_n \cdot y_n^2 - 2 \cdot x_n^2 \cdot y_n^2) \cdot c_n \\
&= x_n \cdot c_{n-1} + y_n \cdot c_{n-1} - 2 \cdot x_n \cdot y_n \cdot c_{n-1} + x_n \cdot y_n \\
&\quad -(2 \cdot x_n \cdot y_n - 2 \cdot x_n \cdot y_n) \cdot c_n \\
&= y_n \cdot c_{n-1} + x_n \cdot (c - 2 \cdot y_n \cdot c_{n-1} + y_n) \\
&= (0 + y_{n-1} \cdot c_{n-1}) + x_n \cdot (c_{n-1} + y_n \cdot (1 - 2c_{n-1}))
\end{aligned}$$

Abbildung 6.4. Shared „Bit-Level“ K∗BMD eines 3 Bit Addierers und Subtrahierers



(a) $X + Y$ (b) $X - Y$

Mit der gewählten DTL und Variablenordnung ergibt sich daraus direkt die in Abb 6.3(b) gegebene DD-Darstellung und für die Annahme, dass c_{n-1} in $\Theta(n)$ darstellbar ist, folgt die Aussage direkt.

Durch Zusammenfassen der Induktionsschritte für s_n und c_n ergibt sich die in Abbildung 6.3(c) dargestellte Struktur des shared K∗BMD für s_n und c_n. ■

In Abbildung 6.4 ist der shared K∗BMD für alle Summen- und Carry-Bits eines 3 Bit Addierers (Abbildung 6.4(a)) und Subtrahierers (Abbildung 6.4(b)) dargestellt.

Experimentell lässt sich eine ähnliche Aussage für die Ausgänge der CAS-Zellen zeigen (siehe Tabelle 6.2), allerdings versagt hierbei die vorgestellte Beweis-Methodik. In der Tabelle ist in der ersten Spalte die Darstellungsgröße für das n-te Summenbit s_{n-1}, in der zweiten Spalte die des n-ten Übertragsbites c_{n-1} angegeben. In den folgenden beiden Spalten ist die Darstellungsgröße für einen shared K∗BMD der alle n Summenbits (bzw. Übertragsbits) darstellt, aufgeführt.

Tabelle 6.2. Bit-Level Darstellungsgröße der Ausgänge einer CAS-Zelle

n	$\lvert s_{n-1}\rvert$	$\lvert c_{n-1}\rvert$	$\lvert s_0, \ldots, s_{n-1}\rvert$	$\lvert c_0, \ldots, c_{n-1}\rvert$
4	24	29	35	38
5	32	37	50	51
8	60	65	95	90
9	68	73	110	103
16	132	137	215	194
17	140	145	230	207
24	204	209	335	298
25	212	217	350	311
32	276	281	455	402
33	284	289	470	415
48	420	425	695	610
49	428	433	710	623

Vermutung 6.2.2 (Bit-Level Darstellung der Ausgänge einer CAS-Zelle) Seien $X = \sum_{i=0}^{n-1} x_i, Y = \sum_{i=0}^{n-1} y_i$ mit $\{x_0, \ldots, x_{n-1}, y_0, \ldots, y_{n-1}\} \in \mathbf{B}^{2n}$ und $s \in \mathbf{B}$. Weiter sei die DTL $d = (pD_{\mathbf{Z}}, \ldots, pD_{\mathbf{Z}})$ und die Variablenordnung $(s, x_{n-1}, y_{n-1}, \ldots x_0, y_0)$ gegeben. Somit gilt:

- Das n-te Summenbit der CAS-Zelle $c \cdot (X - Y) + (1 - c) \cdot (X + Y)$ lässt sich als K∗BMD mit

$$\begin{cases} 9n - 12 & \text{falls } n > 2 \text{ und } n \text{ gerade} \\ 9n - 13 & \text{falls } n > 2 \text{ und } n \text{ ungerade} \end{cases} \tag{6.11}$$

  inneren Knoten darstellen. Als shared K∗BMD für die ersten n Summenbits werden $15n - 25$ innere Knoten benötigt.

- Das n-te Übertragsbit der CAS-Zelle $c \cdot (X - Y) + (1 - c) \cdot (X + Y)$ lässt sich als K∗BMD mit

$$\begin{cases} 9n - 7 & \text{falls } n \text{ gerade} \\ 9n - 8 & \text{falls } n > 2 \text{ und } n \text{ ungerade} \end{cases} \tag{6.12}$$

  inneren Knoten darstellen. Als shared K∗BMD für die ersten n Übertragsbits werden $13n - 14$ innere Knoten benötigt. □

6.3. Zusammenfassung

In diesem Kapitel wurde ein Verfahren motiviert, dass eine vollautomatische Verifikation der nonrestoring Division ermöglicht. Die vollständig DD-basierte Verifikation wird durch eine geeignete Transformation (Schaltkreiserweiterung) ermöglicht. Als einzige Zusatzinformation, neben den Ein- und Ausgängen des Schaltkreises, ist das Markieren der $R^{(i)}$ Busse notwendig. Diese Markierung wird zum einen für die Transformation, zum anderen für die Bildberechnung und die Aufrechterhaltung der gewünschten Variablenordnung benötigt.

Mit diesem Verfahren ist es erstmals möglich, Dividierer üblicher Bitbreiten (bis zu 96 Bit) vollständig zu verifizieren.

7. Don't Care Minimierung

Im vorigen Kapitel wurden die Ergebnisse aus Kapitel 5 verallgemeinert, indem zusätzlich zur Proberechnung eine Transformation angewendet wurde, die den schmalen Dividierer auf die breite Variante abbildet. In diesem Kapitel wird ein zweiter Ansatz motiviert, bei dem auch die Verwandtschaft der breiten und schmalen Implementierung der nonrestoring Division ausgenutzt wird. Hierbei soll durch „geschickte" Belegung des DC-Bereiches versucht werden, die maximalen DD-Größen während der Substitution „klein" zu halten. Dass solch eine „geschickte" DC-Belegung existiert, kann bereits aus den in Kapitel 5 vorgestellten Ergebnissen geschlussfolgert werden.

7.1. DC-Minimierung von Entscheidungsdiagrammen

Bei der DC-Minimierung wird versucht, für eine (boolesche) Funktion f eine vollständige Erweiterung (siehe Definition 2.1.5 auf Seite 20) g zu finden, so dass g in einem gewählten Kostenmaß „billiger" als f ist. Ein bekanntes Verfahren zur Minimierung der Monome in einer booleschen Funktion ist beispielsweise das McClusky Verfahren (siehe [30]). Bei der DC-Minimierung von Entscheidungsdiagrammen wird in der Regel eine Erweiterung gesucht, so dass die Anzahl innerer Knoten des darstellenden DDs minimal wird.

Definition 7.1.1 (DC-Minimierung von Entscheidungsdiagrammen) Sei ein DD F über $\chi \subseteq \mathbf{B}^n$ mit der Terminalmenge T und der festen DTL d gegeben. Weiter sei $x \in \mathbf{B}^n$. Gesucht ist ein DD G über χ mit der Terminalmenge T und der festen DTL d, so dass gilt:

$$f_G^d(x) = \begin{cases} f_F^d(x) & \text{falls} \quad x \in \mathrm{DEF}(f_F^d) \\ \text{beliebig} & \text{sonst} \end{cases} \tag{7.1}$$

und $|G| < |F|$. □

In der Regel sind die Mengen $\mathrm{DEF}(f_G^d)$ und $\mathrm{DC}(f_G^d)$ durch die charakteristische Funktion (siehe 2.1.6 auf Seite 21) $\chi_{\mathrm{DEF}(f_F^d)}$ gegeben, so dass sich Gleichung 7.1 schreiben lässt als:

$$f_G^d(x) = \begin{cases} f_F^D(x) & \text{falls} \quad \chi_{\mathrm{DEF}(f_F^d)}(x) = 1 \\ \text{beliebig} & \text{falls} \quad \chi_{\mathrm{DEF}(f_F^d)}(x) = 0 \end{cases} \tag{7.2}$$

Die Repräsentation der DC-Menge durch die charakteristische Funktion ist in der Regel zu bevorzugen, da sich diese in natürlicher Weise als OBDD darstellen lässt.

Während für die DC-Minimierung von OBDDs bereits verschiedene Verfahren bekannt und eingesetzt werden (siehe [20, 42]), sind für Word-Level Diagramme noch keine wirkungsvollen Verfahren bekannt. Insbesondere bei der Verwendung von $\{pD_{\mathbf{Z}}, nD_{\mathbf{Z}}\}$-Zerlegungen ist zu beachten, dass eine DC-Zuweisung keine lokale Operation mehr darstellt, sondern Änderungen global propagiert werden müssen.

7.2. Die Beziehung zwischen den Varianten der nonrestoring Division

In Abschnitt 5.4.3 auf Seite 70 wurde bereits mit Korollar 5.4.2 gezeigt, dass die höchstwertigsten Bits des Busses $R^{(i)}$ beim breiten Dividierer lediglich einer Sign-Extension der Darstellung beim schmalen Dividierer entsprechen.

Um zu motivieren, dass die Darstellung auf Stufe i des breiten Dividierers einer Art[1] „geschickte" DC-Belegung des schmalen Dividierers entspricht, wurde untersucht, wie sich die Darstellung des breiten Dividierers verhält, wenn die die Erweiterung repräsentierenden Variablen „gleichgesetzt" werden. Im Falle des korrekten Dividierers ist bekannt, dass die Erweiterung eine Sign-Extension berechnet, d.h. auf jedem Schnitt gilt, dass alle Variablen der Erweiterung den gleichen Wert (d.h. sie sind entweder alle gleich 0 oder alle gleich 1) haben. Aus Korollar 5.4.2 folgt, dass dieser Wert gleich dem Wert des höchstwertigen Bits von $R^{(i)}$, also r_{n-1} entspricht. Diese Eigenschaften führen zu folgender Vermutung:

Vermutung 7.2.1 Sei G ein geordnetes DD über der Variablenmenge $\chi = \{q_{n-i}, d_0, \ldots, d_{n-1}, r_0^{(i)}, \ldots, r_{n-1}^{(i)}, e_0^{(i)}, \ldots, e_{2\cdot i}^{(i)}\}$ mit der DTL $d = \{pD_{\mathbf{Z}}, \ldots, pD_{\mathbf{Z}}\}$. G sei die Darstellung auf Stufe i des verbreiterten n-Bit Dividierers. Weiter sei F ein DD über $\chi' = \{q_{n-i} d_0, \ldots, d_{n-1}, r_0^{(i)}, \ldots, r_{n-1}^{(i)}\}$, also $\chi' \subseteq \chi$, für die Darstellung auf Stufe i des schmalen n-Bit Dividierers. Dei DTL von D sei $d' = \{pD_{\mathbf{Z}}, \ldots, pD_{\mathbf{Z}}\}$.
Sei H das DD über χ', das aus G ensteht, wenn die Variablen $\{e_0^{(i)}, \ldots, e_{2\cdot i}^{(i)}\}$ der Erweiterung durch $r_{n-1}^{(i)}$ ersetzt (substituiert) werden. H besitzt die DTL d'. Weiter sei gefordert, dass die DDs F, G und H auf den gemeinsamen Variablen dieselbe Variablenordnung besitzen. Dann gilt:

1. H lässt sich durch geschickte DC-Belegung aus F berechnen, d.h. H und F stimmen auf dem Definitionsbereich von F überein. Somit ist H eine Erweiterung von F.

2. H ist effizient darstellbar. □

[1] Die Darstellung auf Schnitt i des breiten Dividierers ist von zusätzlichen Variablen, welche die Erweiterung repräsentieren, abhängig.

Tabelle 7.1. Substitution der Variablen der Erweiterung

Bitbreite	3	4	5	6	7	8	12	16	24	32	48
maximale Größe auf Schnitt	14	19	25	31	37	43	67	91	139	187	283
nach Substitution	7	11	15	19	23	27	43	59	91	123	187

Das H und F auf dem Definitionsbereich übereinstimmen lässt sich direkt aus den Aussagen von Satz 3.1.1 auf Seite 48 zusammen mit Satz 5.4.2 auf Seite 71 folgern. Bei der Argumentation wird ausgenutzt, dass der verbreiterte Dividierer und der schmale Dividierer denselben Algorithmus umsetzen und auf den Schnitten nur n-Bit Zahlen dargestellt werden müssen. Aus dieser Argumentation ergibt sich weiter, dass das Bild des schmalen Dividierers (dargestellt durch das DD F) aus Stufe i gerade dem Definitionsbereich (Care-Bereich) entspricht, unter dessen Berücksichtigung Gleicheit zwischen schmaler und verbreiteter Implementierung gefordert wird.

Die zweite Vermutung, dass die Erweiterung H effizient darstellbar ist, wurde von den Experimenten bestätigt. Es hat sich zudem gezeigt, dass auf allen Schnitten die Größe von H kleiner als die von G ist. In Tabelle 7.1 ist die maximale Größe auf dem Schnitt, vor und nach der Substitution (siehe hierzu Tabelle 5.3 auf Seite 69) dargestellt. Bei der gewählten, verzahnten Variablensortierung, lässt sich das Wachstum der Darstellung nach der Substition durch $4n - 5$ abschätzen.

7.3. Verifikation mit Hilfe der DC-Minimierung

In diesem Abschnitt soll ein Verfahren motiviert werden, dass eine effiziente Verifikation der nonrestoring Division ermöglicht, vorausgesetzt, es steht eine geeignete, effiziente Routine zur DC-Minimierung zur Verfügung.

7.3.1. Der Grundalgorithmus

Der folgende Algorithmus basiert auf dem bereits vorgestellten Verfahren, verwendet aber anstatt der Verbreiterung der Stufen (siehe Abbildung 6.1 auf Seite 79) die Idee der DC-Minimierung. Dadurch wird die Transformation wieder auf die Ergänzung des Dividierers um den Probeschaltkreis reduziert.

Eingabe: Netzliste des nonrestoring Dividierers. In der Netzliste sind neben den Eingängen D und $R^{(0)}$ sowie den Ausgängen Q und $R^{(n+1)}$ auch die Busse an denen die Partialreste $R^{(i)}$ anliegen markiert.

Die Markierung der $R^{(i)}$-Busse für $0 < i < n + 1$ stellt den einzigen manuellen Eingriff dar. Ansonsten läuft das Verfahren vollautomatisch ab!

Ausgabe: Aussage über die Korrektheit des Dividieres. Im Falle eines fehlerhaften Schaltkreises ist eine ungefähre Abschätzung des Fehlerortes möglich.

Algorithmus:

1. Die Variablenordnung auf den Schnitten wird festgelegt. Als geeignet hat sich die Verzahnung von D- und $R^{(i)}$-Bussen erwiesen. Aufgrund der Markierung der $R^{(i)}$-Busse kann die Sortierung automatisch erfolgen.
2. Führe eine „stufenweise" Bildberechnung, die in Abschnitt 5.4.3 auf Seite 70 vorgestellt wurde, auf dem Dividierer durch. Auf Schnitt i gilt folgende Beziehung zwischen dem Bild img_i und dem Don't-Care-Bereich DC_i:
$$\mathrm{DC}_i = \overline{\mathrm{img}_i} \tag{7.3}$$
3. Erweitere „virtuell" den Dividierer zusätzlich um die Proberechnung $Q \cdot D + R^{(n+1)}$.
4. Führe stufenweise die Substitution durch. Um die vermutlich exponentiellen Darstellungsgrößen auf den Schnitten (siehe Abschnitt 5.3 auf Seite 64) zu vermeiden, wird auf den Schnitten eine Don't-Care-Minimierung durchgeführt. Das Verfahren wird mit der Darstellung auf der Stufe, die durch die DC-Minimierung ermittelt wurde, fortgesetzt.

7.3.2. Korrektheit

Da die zusätzlich ausgeführten DC-Minimierungen keinen Einfluss auf die Funktionalität des Dividierers innerhalb des Definitionsbereiches haben, kann die Argumentation aus Abschnitt 5.4 auf Seite 66 direkt übernommen werden.

7.3.3. Platzverhalten

Im Idealfall findet die DC-Minimierungs-Routine auf jeder Stufe gerade die Belegung, die der Darstellung des breiten Dividierers nach der Substitution entspricht. In diesem Fall besteht die Hoffnung, dass der Platzbedarf des Verfahrens mit DC-Minimierung gegenüber dem Verfahren aus Kapitel 6 nur unwesentlich größer ist.

7.4. Bisherige Untersuchungen

Erste Experimente, bei denen versucht wurde alle DC-Stellen auf „0" zu setzen, indem auf jeder Stufe die Darstellung mit dem entsprechenden Bild multipliziert wird, brachten keine erkennbaren Erfolge, so dass die Entwicklung komplexerer Algorithmen notwendig ist.

Bei OBDDs ist die DC-Minimierung eine lokale Operation. Damit ist gemeint, dass eine DC-Zuweisung im *low*-Sohn eines Knotens keine Auswirkung auf die Darstellung der Funktion des *high*-Sohnes hat. Somit kann in diesem Fall die gesamte DC-Belegung in einfacher Weise rekursiv berechnet werden, indem zuerst eine „gute“ DC-Belegung für den *low*-Sohn und anschließend eine „gute“ DC-Belegung für den *high*-sohn ermittelt wird.

Bei Knoten mit einer Davio-Zerlegung ist dies leider nicht mehr der Fall. Eine DC-Belegung im *low*-Sohn ist keine lokale Operation mehr, sondern wirkt sich auch auf die Darstellung des *high*-Sohnes aus. Genauer gilt für die $pD_{\mathbf{Z}}$-Zerlegung[2] bei Betrachtung der Funktion f für die Zerlegung nach x_i:

$$\begin{aligned} f &= f_{low(x_i)} + x_i \cdot f_{high(x_i)} \\ &= f_i^0 + x_i \cdot f_i^2 \\ &= \underbrace{f_i^0}_{low\text{-Sohn}} + x_i \cdot \underbrace{\left(f_i^1 - f_i^0\right)}_{high\text{-Sohn}} \end{aligned}$$

Wird nun der *low*-Sohn, in diesem Fall also f_i^0, minimiert, so muss diese Änderung propagiert werden. Sei $f'_{low(x_i)} = f'^0_i$ der durch die DC-Zuweisung aus f_i^0 enstandene, neue *low*-Sohn. Damit der Wert von f auf dem Definitionsbereich nicht verändert wird, muss die Differenz aus neuem und altem *low*-Sohn zum *high*-Sohn addiert werden:

$$\begin{aligned} f' &= f'_{low(x_i)} + x_i \cdot \Big(f_{high(x_i)} + (f_{low(x_i)} - f'_{low(x_i)})\Big) \\ &= f_i^0 + x_i \cdot \left(f_i^2 + f_i^0 - f_i'^0\right) \\ &= f_i'^0 + x_i \cdot \left(f_i^1 - f_i'^0\right)\big) \\ &= f_i'^0 + x_i \cdot f_i'^2 \,. \end{aligned}$$

Im schlimmsten Fall sind durch die Propagation der Veränderung von f_i^0 mehr neue Knoten zur Darstellung des *high*-Sohnes notwendig, als durch die Minimierung von f_i^0 eingespart wurden. Deswegen muss vor jeder DC-Zuweisung eine Folgenabschätzung durchgeführt werden.

Die im Rahmen dieser Arbeit bisher entwickelten Ansätze erzielen zwar gute Einsparungseffekte auf einer Stufe. Leider wird aber nach der DC-Minimierung nicht die Funktion dargestellt, die bei der breiten Variante nach der Substitution der Variablen der Erweiterung, zu finden ist. Der verbleibende Unterschied führt während des Verfahrens zu einem nicht tragbaren Grössenwachstum, so dass diese Vorgehensweise im Moment nur für kleine Eingaben, bis etwa 8 Bit, sinnvoll angewendet werden kann.

Allerdings bieten die bisher erarbeiteten DC-Minimierungsansätze eine gute Basis zur allgemeinen DC-Minimierung von Word-Level-Diagrammen mit $pD_{\mathbf{Z}}$-Knoten.

[2] Eine analoge Betrachtung ergibt sich für die Zerlegungen $pD_{\mathbf{B}}$, $nD_{\mathbf{B}}$ und $nD_{\mathbf{Z}}$.

7.5. Ausblick

Der Bereich der Minimierung von Word-Level-Diagrammen durch Ausnutzung von DC-Bereichen bietet ein weites Forschungsfeld. In diesem Kapitel wurden erste Ergebnisse vorgestellt und gezeigt, wie diese Minimierungstechnik gewinnbringend beim vorgestellten DD basierten Verfahren zur Verifikation von Dividierern eingesetzt werden kann.

Weitere Arbeiten werden sich auf die Verbesserung der Heuristiken zur DC-Belegung konzentrieren, um zum einen diese Technik bei der Verifikation von Dividierern einsetzen zu können und zum anderen, um verschiedene allgemeine Verfahren zur DC-Minimierung bei K*BMDs zu untersuchen.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein neuer Ansatz zur Verifikation von Dividiererschaltkreisen untersucht. Dabei wird der Schaltkreis erweitert (transformiert), so dass der Gesamtschaltkreis die Identität darstellt. Somit muss an keiner Stelle der Verifikation die Funktion $\div$ oder mod dargestellt werden.

Speziell für die nonrestoring Division konnte eine Transformation angegeben werden, mit der eine DD-basierte Verifikation mit linearem Platzbedarf möglich ist. Mit Hilfe des vorgestellten Verfahrens konnten Dividierer mit einer Bitbreite von 96 Bit in weniger als 10 Stunden vollautomatisch verifiziert werden.

Das neu entwickelte Verifikationsverfahren unterscheidet sich grundlegend zu den in Kapitel 4.3 auf Seite 54 vorgestellten bisherigen Ansätzen:

- Im Gegensatz zu den bisherigen DD basierten Ansätzen ist erstmals eine vollständige Verifikation möglich.
- Im Gegensatz zu den auf Model Checking und Theorem Proving basierenden Verfahren sind keine manuellen Eingriffe während des Verfahrens notwendig.

Weiter konnte gezeigt werden, dass das Verfahren robust ist gegenüber Modifikationen innerhalb der einzelnen CAS-Zellen, z.B. bei der Verwendung von schnelleren Addierern, ist. Analog lässt sich auch motivieren, dass die restoring Division mit dem vorgestellten Verfahren effizient verifiziert werden kann.

Zudem wurde ein Verfahren aufgezeigt, dass mit einer einfacheren Transformation auskommt, dafür aber linear viele DC-Minimierungen benötigt. Momentan ist mit dieser Variante noch keine effiziente Verfikation möglich, es wurden bei den Untersuchungen für dieses Verfahren gute Algorithmen zur DC-Minimierung bei K∗BMDs mit $pD_{\mathbf{Z}}$-Knoten entwickelt.

Weitere Untersuchungen sollen zeigen, wie das Verfahren auf andere Dividierer mit der Addition und Subtraktion als Basisoperation übertragen werden kann. Am interessantesten dürfte hierbei die Verifikation des im Pentium eingesetzten SRT-Dividerers sein. Hierbei muss zusätzlich eine Look-Up-Table effizient als DD-dargestellt (siehe [4]) und in das Verfahren integriert werden.

Ein weiteres Gebiet ist die Verifikation von Dividierern, welche die Multiplikation als Basisoperation verwenden. Hierbei berechnet eine Stufe die Multiplikation und es ergeben sich andere Invarianten auf den einzelnen Schnitten. Momentan lässt sich noch

keine Aussage treffen, ob ein auf der vorgestellten Grundidee basierendes Verfahren auch bei dieser Klasse von Schaltkreisen Erfolg haben kann.

Gerade beim Übertragen des Verfahrens auf andere Dividierer könnte es nützlich sein auch die Variante mit DC-Minimierung einzusetzen, da hier keine Erweiterung auf jeder Stufe notwendig ist.

Zu untersuchen bleibt weiterhin, ob sich die Grundidee des Verfahrens, die Transformation des zu verifizierenden Schaltkreises mit Hilfe der inversen Funktion, auf andere strukturierte Schaltkreise übertragen lässt.

Anhang

A. Stromlaufpläne

Im folgenden sind die Stromlaufpläne diverser Dividierer wiedergeben. Es handelt sich dabei jeweils um 4-Bit breite Dividierer. Zu beachten ist, dass aufgrund der Vorbedingung $0 \leq D$ und $0 \leq R^{(0)}$ die Eingänge der ersten Stufe nur 3 Bit breit sind, da das Vorzeichenbit den konstanten Wert 0 hat. Im einzelnen zeigen die verschiedenen Abbildungen:

Abbildung A.1 zeigt eine Übersicht der in den folgenden Schaltkreisen verwendeten Gatter.

Abbildung A.2 zeigt einen 4 Bit restoring Dividerer, bei dem die Addition durch Carry-Ripple Addierer realisiert wird. Deutlich sind die 4 identischen Stufen zu sehen, die jeweils aus einer 4 Bit breiten CAS-Zelle mit nachgeschalteter Rückaddition bestehen.

Abbildung A.3 zeigt einen 4 Bit breiten nonrestoring Dividierer. Deutlich sind die 4 CAS-Zellen und die nachgeschaltete Rückaddition zu erkennen. Die Addierer sind als Carry-Ripple-Addierer implementiert.

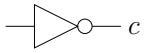
Abbildung A.4 zeigt die breite Variante des nonrestoring Dividierers. Besonders im direkten Vergleich zu Abb. A.3 ist die Verbreiterung der einzelnen Stufen besonders deutlich.

Abbildung A.5 zeigt, wie durch Verbreiterung der „normale“ nonrestoring Dividierer, wie er beispielsweise in Abb.A.3 dargestellt ist, in die breite Variante eingebettet werden kann.

Abbildung A.1. Übersicht der verwendeten Gatter

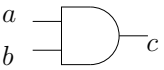
$c = \bar{a}$

a	c
0	1
1	0



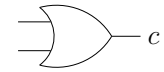
$c = a \wedge b$

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



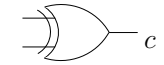
$c = a \vee b$

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



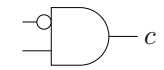
$c = a \oplus b$

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



$c = \bar{a} \wedge b$

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



$c = (s \wedge a) \vee (\bar{s} \wedge b)$

s	c
0	a
1	b

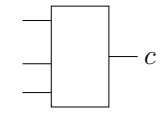


Abbildung A.2. 4 Bit restoring Dividierer

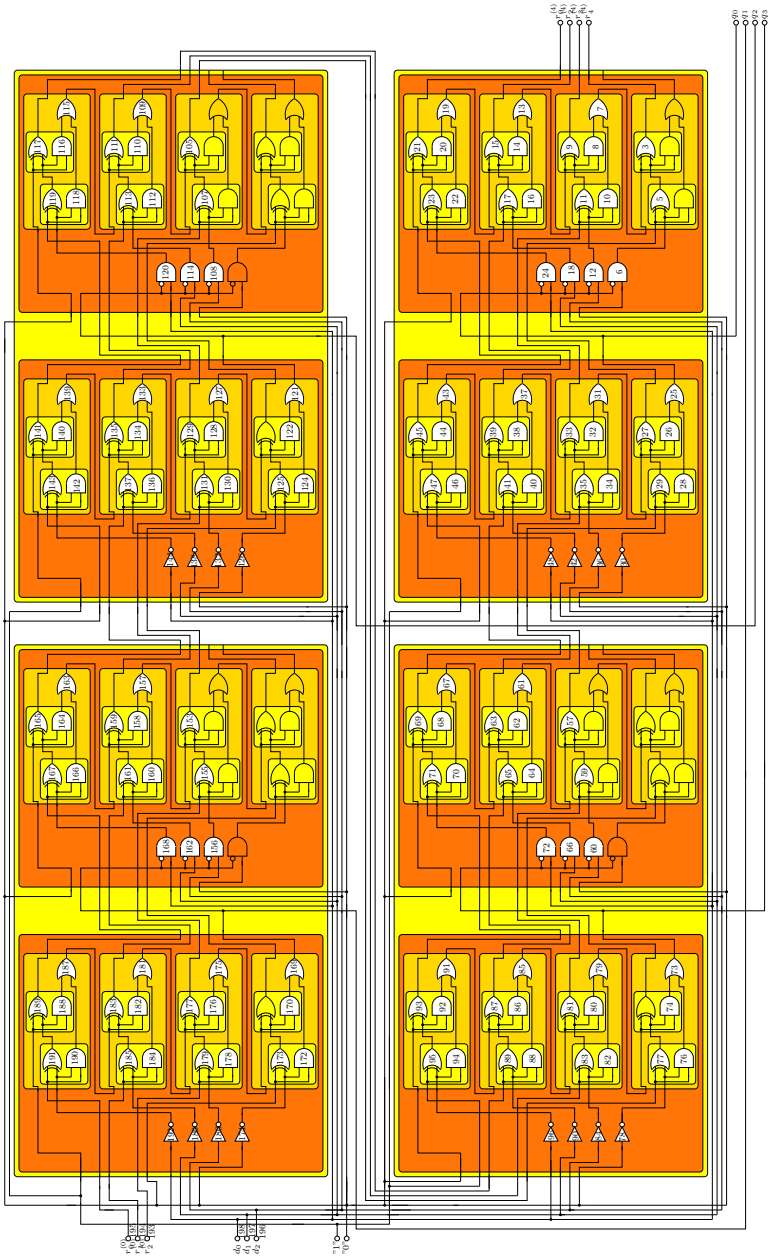


Abbildung A.3. 4 Bit nonrestoring Dividierer

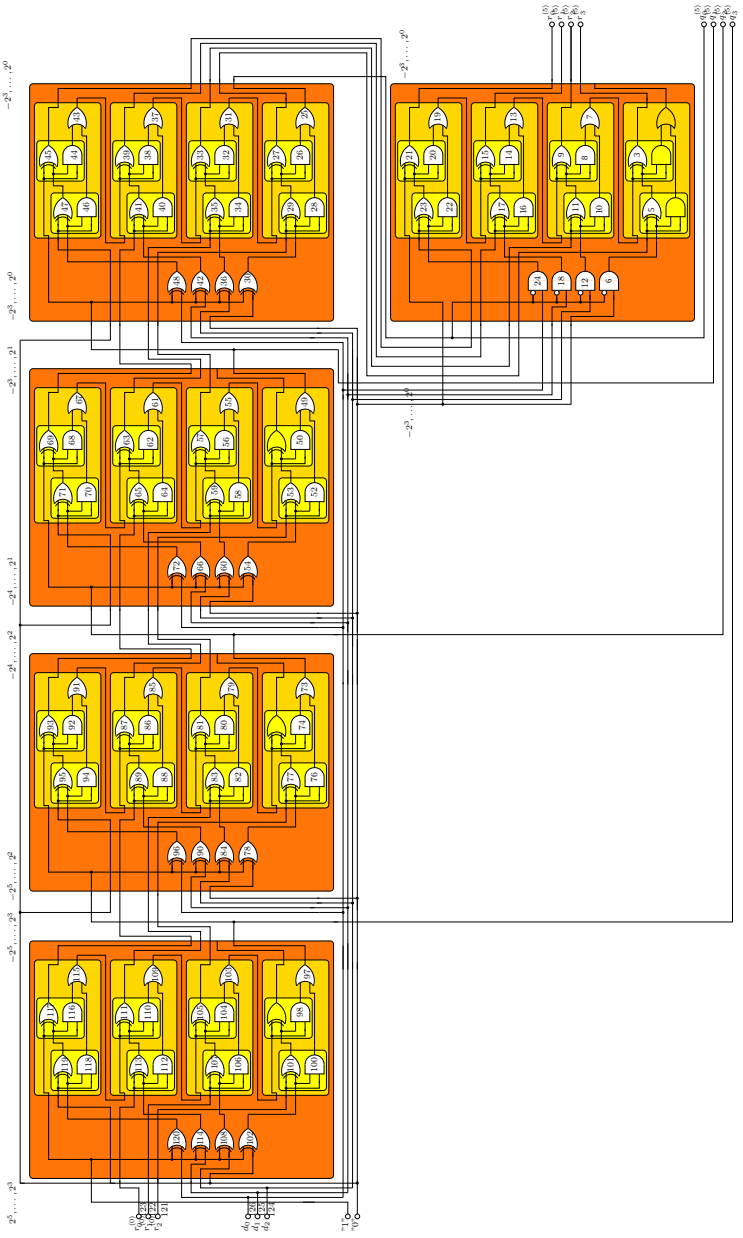


Abbildung A.4. Breite Variante des 4 Bit nonrestoring Dividierers

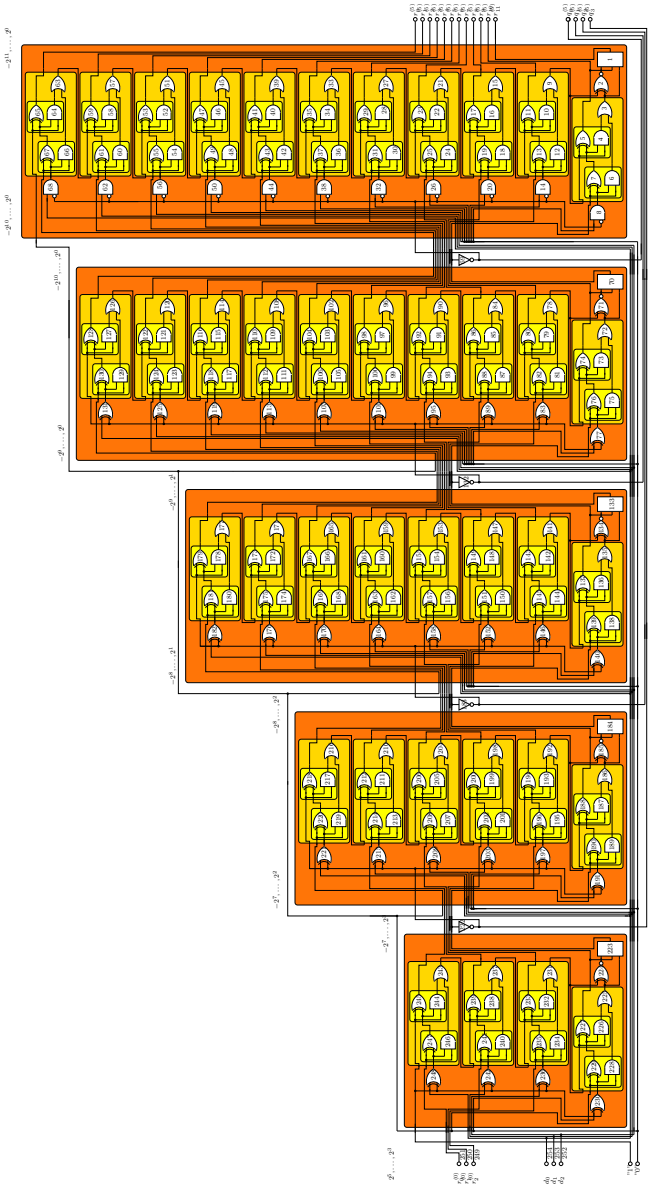
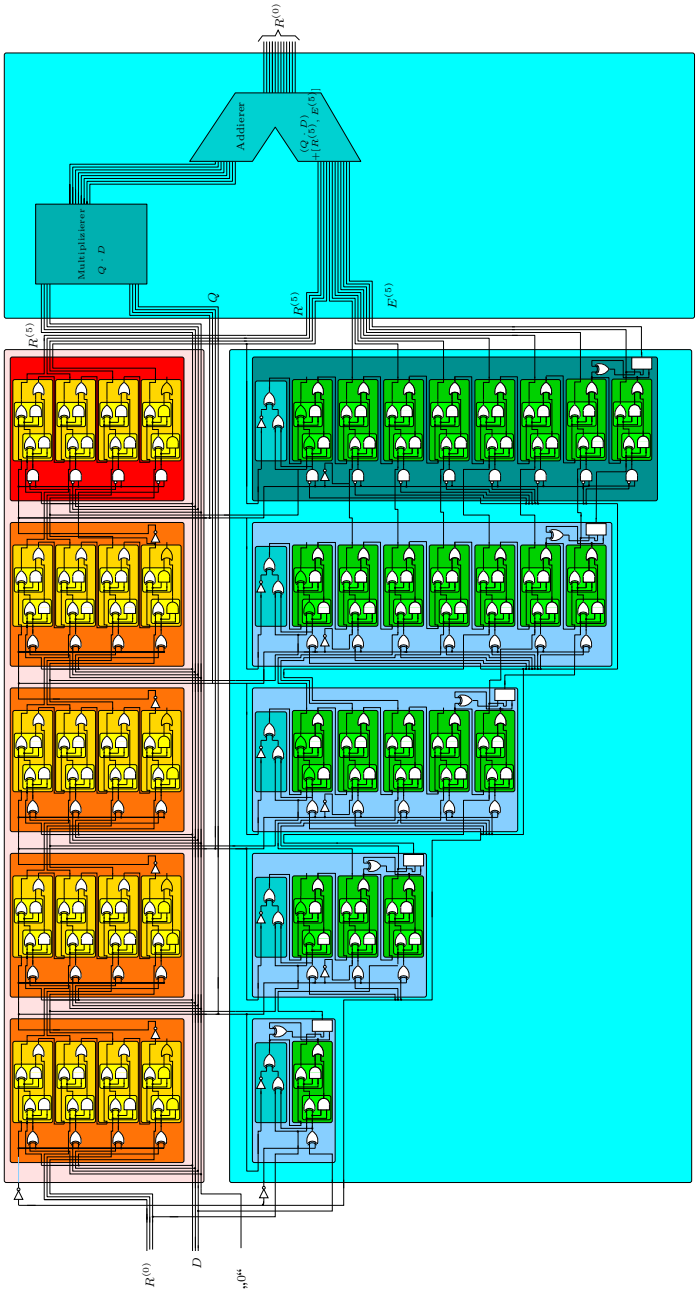


Abbildung A.5. 4 Bit nonrestoring Dividierer mit Verbreiterung



B. Über die CD

Inhalt der CD

Die CD enthält

- eine elektronische Version der Diplomarbeit,
- die Folien zu einem Vortrag über die Diplomarbeit,
- eine Umsetzung des vorgestellen Verifikationsverfahrens,
- Netzlisten für die nonrestoring Division verschiedener Bitbreiten,
- eine Sammlung von Dokumenten über den Pentium Bug,
- eine elektronische Version des Literaturverzeichnisses mit Verweisen zu den Orginalarbeiten.

Installation und Nutzung

Die CD ist unter allen Betriebsystemen nutzbar. Um die Handhabung zu vereinfachen ist die CD mit einer HTML-basierten Oberfläche ausgestattet.

Unter Windows ist die CD als so eingerichtet, dass nach dem Einlegen der CD automatisch die Datei `index.html` geladen wird. Auf allen anderen Betriebsystemen sollte die Datei `index.html` mit einem WWW-Browser betrachtet werden.

Literaturverzeichnis

[1] Pentium® III processor specification update. January 2000. ftp://download.intel.com/design/PentiumIII/specupdt/24445312.pdf.

[2] B. Becker and R. Drechsler. How many decomposition types do we need? In *European Design & Test Conf.*, pages 438–443. 1995. http://www.informatik.uni-freiburg.de/~drechsle/ps_test/DECOMP.ps.

[3] B. Becker, R. Drechsler, and R. Enders. On the computational power of bit-level and word-level decision diagrams. In *ASP Design Automation Conf.*, pages 461–467. 1997. http://ira.informatik.uni-freiburg.de/papers/Year_97/BDE_97.ps.gz.

[4] Randal E. Bryant. Bit-Level analysis of an SRT Divider Circuit. In *Design Automation Conf.*, pages 661–665. 1996. http://www.cs.cmu.edu/~bryant/pubdir/dac96b.ps.

[5] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986. http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.ps.

[6] R.E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Trans. on Comp.*, 40:205–213, 1991. http://www.cs.cmu.edu/~bryant/pubdir/ieeetc91.ps.

[7] R.E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. Technical report, CMU-CS-94-160, 1994. http://reports-archive.adm.cs.cmu.edu/anon/1994/CMU-CS-94-160.ps.

[8] R.E. Bryant and Y.-A. Chen. Verification of arithmetic functions with binary moment diagrams. In *Design Automation Conf.*, pages 535–541. 1995. http://www.cs.cmu.edu/~bryant/pubdir/dac95.ps.

[9] Tim Coe. Inside the Pentium FDIV Bug. *Dr. Dobbs Journal*, 20(4):129–135, April 1995.

[10] R. Drechsler. *Ordered Kronecker Functional Decision Diagrams und ihre Anwendungen*. Modell Verlag, Dissertation, J.W. Goethe-Universität, Frankfurt am Main, 1995. ISBN 3-9805033-0-5.

[11] R. Drechsler und B. Becker. *Graphenbasierte Funktionsdarstellung*. B.G. Teubner, Stuttgart, 1998. ISBN 3-519-02149-8.

[12] R. Drechsler, B. Becker, and S. Ruppertz. K*BMDs: A new data structure for verification. In *European Design & Test Conf.*, pages 2–8. 1996. `http://www.informatik.uni-freiburg.de/~drechsle/ps_test/KBMD.ps`.

[13] R. Drechsler, B. Becker, and S. Ruppertz. The K*BMD: A verification data structure. *IEEE Design & Test of Comp.*, pages 51–59, 1997.

[14] R. Drechsler, B. Becker, and S. Ruppertz. Manipulation algorithms for K*BMDs. In *Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, pages 4–18. 1997. `http://www.informatik.uni-freiburg.de/~drechsle/ps_test/KBMDSIFT2.ps`.

[15] R. Drechsler and S. Höreth. Manipulation of *BMDs. In *ASP Design Automation Conf.*, pages 433–438. 1998. `http://www.informatik.uni-freiburg.de/~drechsle/ps_test/BMD_SIFT.ps`.

[16] Alan Edelman. The mathematics of the pentium division bug. *SIAM*, 39(1):54–67, 1997. `http://epubs.siam.org/sam-bin/dbq/article/29395`.

[17] Aarti Gupta. Formal hardware verification methods: A survey. Technical Report CMU-CS-91-193, Carnegie Mellon University, Pittsburgh, October 1991.

[18] K. Hamaguchi, A. Morita, and S. Yajima. Efficient construction of binary moment diagrams for verifying arithmetic circuits. In *Int'l Conf. on CAD*, pages 78–82. 1995.

[19] Marc Herbstritt. *Implementation eines *BMD-Paketes*. Albert-Ludwigs-Universität, Freiburg, Oktober 1997. `http://www.informatik.uni-freiburg.de/~herbstritt/hiwi/kbmdpaket.html`. Aus dieser Arbeit enstand das K*BMD-Paket WLD.

[20] Y. Hong, P.A. Beerel, J.R. Burch, and K.L. McMillan. Safe BDD minimization using don't cares. In *Design Automation Conf.*, pages 208–213. 1997.

[21] S. Höreth. *Effiziente Konstruktion und Manipulation von binären Entscheidungsgraphen*. Dissertation, Technische Universität, Darmstadt, 1998.

[22] S. Höreth and R. Drechsler. Dynamic minimization of word-level decision diagrams. In *Design, Automation and Test in Europe*, pages 612–617. 1998. `http://www.informatik.uni-freiburg.de/~drechsle/ps_test/WORDSIFT.ps`.

[23] Stefan Höreth and Claudia Blank. *TUD Decision Diagram Package*. Darmstadt University of Technology, 1999. `http://www.rs.e-technik.th-darmstadt.de/~sth/demo.html`.

[24] Robert L. Hummel. *The Processor and Coprocessor*. PC Magazine Programmer's Technical Reference:. Ziff-Davis Press, Emerywille, CA, 1992. ISBN 1-56276-016-5.

[25] M. Keim, M. Martin, B. Becker, R. Drechsler, and P. Molitor. Polynomial formal verification of multipliers. In *VLSI Test Symp.*, pages 150–155. 1997. `http://ira.informatik.uni-freiburg.de/papers/Year_97/KMBDM_97.ps.gz`.

[26] Israel Koren. *Computer Arithmetic Algorithms*. Prentice-Hall, Inc., 1993. ISBN 0-13-151952-2. `http://www.ecs.umass.edu/ece/koren/arith/`.

[27] O. L. MacSorley. High-speed arithmetic in binary computers. *Proc. of IRE*, 49:67–91, January 1961.

[28] Christoph Meinel und Thorsten Theobald. *Algorithmen und Datenstrukturen im VLSI-Design*. Springer-Verlag, Berlin, März 1998. ISBN 3-540-63869-5.

[29] Paul S. Miner and Jr. James F. Leathrum. Verification of IEEE compliant subtractive division algorithms. *FMCAD*, October 1996. `http://www.ee.odu.edu/~leathrum/Formal_Methods/computer_arithmetic/fmcad.ps`.

[30] P. Molitor und C. Scholl. *Datenstrukturen und effiziente Algorithmen für die Logiksynthese kombinatorischer Schaltungen*. B.G. Teubner, Stuttgart, Leipzig, 1999.

[31] J. Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the AMD5$_K$86^{TM} floating-point division algorithm. Technical report, University of Berkeley, 1996. `http://devil.ece.utexas.edu:80/~lynch/divide/divide_paper.ps`.

[32] M. Nakanishi. *An Exponential Lower Bound on the Size of a Binary Moment Diagram Representing Division*. Master's thesis, Osaka University, February 1998. `http://www-kasi.ics.es.osaka-u.ac.jp/m-naka/paper/BMDdiv.ps`.

[33] Stuart F. Oberman. Floating point division and square root algorithms and implementation in the AMD-K7^{TM} microprocessor. Technical report, Advanced Micro Devices, Sunyvale, CA 94088, March 1997. `http://euler.ecs.umass.edu/paper/final/paper-139.ps`.

[34] Stuart F. Oberman and Michael J. Flynn. An analysis of division algorithms and implementations. Technical report, Standord University, Standford, California, July 1995. `ftp://umunhum.stanford.edu/tr/oberman.jul95.tr675.ps.Z`.

[35] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, Q1, 1999. `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.

[36] V.R. Pratt. Anatomy of the pentium bug. In *TAPSOFT'95*, volume 915 of *LNCS*, pages 97–107. Springer-Verlag, Aarhus, Denmark, 1995. `ftp://boole.standford.edu:/pub/anapent.ps.gz`.

[37] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, September 1958.

[38] David M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7^{TM} processor. *LMS JCM*, 1:148–200, December 1998. `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

[39] C. Scholl, B. Becker, and T.M. Weis. Word-level decision diagrams, wlcds and division. Technical Report 102, Albert-Ludwigs-University, Freiburg, May 1998. `http://ira.informatik.uni-freiburg.de/papers/Year_98/SBW_98.ps.gz`.

[40] C. Scholl, B. Becker, and T.M. Weis. Word-level decision diagrams, wlcds and division. In *Int'l Conf. on CAD*, pages 672–677. 1998. `http://ira.informatik.uni-freiburg.de/papers/Year_98/SBW_98b.ps.gz`.

[41] H. P. Sharangpani and Ph. D. M. I. Barton. Statistical analysis of floating point flaw in the pentium^{TM} processor(1994), November 1994. `http://www.intel.com/procs/support/pentium/fdiv/white11.pdf`.

[42] T.R. Shiple, R. Hojati, A.L. Sangiovanni-Vincentelli, and R.K. Brayton. Heuristic minimization of BDDs using don't cares. In *Design Automation Conf.*, pages 225–231. 1994.

[43] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quart. J. Mech. Appl Math.*, EC-10:662–670, 1961.

[44] Ingo Wegener. *The Complexity of Boolean Functions*. John Wiley & Sons Ltd., and B.G. Teubner, Stuttgart, 1987. `http://ls2-www.informatik.uni-dortmund.de/monographs/bluebook/`.

Index