Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz
Prof. Dr.-Ing. D. Schmid

Am Zirkel 2
Postfach 6980
76128 Karlsruhe
Tel. +49 721/ 608-3960

Studienarbeit

Verifikation der diskreten Cosinus-Transformation mittels Modellprüfung

von

Jens Zimmermann

Betreuer: Prof. Dr.-Ing. D. Schmid
Betreuende Mitarbeiter : Dr. rer. nat. Klaus Schneider
Dr. rer. nat. Michaela Huhn

Tag der Anmeldung: 01. 08. 1998
Tag der Abgabe: 31. 10. 1998

Erklärung

Hiermit versichere ich, daß ich die vorliegende Studienarbeit selbständig verfaßt und keine anderen Quellen und Hilfsmittel als die im Literaturverzeichnis aufgeführten verwendet habe.

Karlsruhe, den 31. Oktober 1998

Inhaltsverzeichnis

1. Einleitung

1.1 Hardware-Verifikation

Ziel der Hardware-Verifikation ist die Vermeidung von Entwurfsfehlern. Ein nachträgliches Beheben von Entwurfsfehlern ist sehr teuer und zeitaufwendig. Ein Beispiel hierfür ist der Fehler im Pentium-Prozessor, dessen Beseitigung die Firma Intel ca. 100 Millionen Dollar gekostet hat.

Bei der Hardware-Verifikation wird ein mathematischer Beweis geführt, daß eine Implementierung ihrer Spezifikation genügt. Es kann allerdings nur die Korrektheit der Implementierung gegenüber einer formellen Spezifikation bewiesen werden. Der Beweis der Korrektheit gegenüber einer Spezifikation die z.B. in Umgangssprache verfaßt ist, ist nicht möglich. Einen allgemeinen Überblick über die Hardware-Verifikation liefern [6] und [7].

Zur formalen Verifikation existieren zwei Hauptansätze: Zum einen der Theorembeweis, bei dem für zwei logische Formeln gezeigt wird, daß die eine aus der anderen folgt, und zum anderen die Modellprüfung [8], bei der für eine Spezifikation gezeigt werden muß, daß sie auf einer Interpretation der Logik, in der sie verfaßt ist zu dem Wert *wahr* evaluiert werden kann.
Eine Möglichkeit der Spezifikation von Systemen ist die temporale Logik CTL [9], bei der Systeme durch endliche Transitionssysteme modelliert werden und die eine effiziente Modellprüfung erlaubt.

Das in dieser Arbeit betrachtete Fallbeispiel einer Verifikation mittels Modellprüfung ist die diskrete Cosinus-Transformation. Diese ist ein wichtiger Kompressionsalgorithmus der z. B. bei JPEG, MPEG und zur Sprachkompression verwendet wird.

1.2 Datenkompression

Heutzutage kommen im Bereich der Bildverarbeitung hochauflösende Grafiken mit einigen Millionen Farben vor. So benötigt z. B. ein Bild mit einer Auflösung von 800 x 600 Punkten und einer Farbtiefe von 32 Bit einen Speicherplatz von 1,83 MByte. Da dies bei einer größeren Anzahl von Bildern schnell zu einer riesigen Datenmenge führt, muß man sich Verfahren überlegen, die zu einer Reduktion der Datenmenge führen.

Durch die Kompression von Daten erreicht man deutliche Einsparungen beim benötigten Speicherplatz. Ein weiterer Vorteil ist, daß bei der Übertragung von komprimierten Daten in lokalen Netzwerken oder über das Internet weniger Zeit benötigt und das Datennetz weniger belastet wird. Dies wird vor allem dann deutlich, wenn man Bildfolgen oder ganze Videosequenzen übertragen will.

Im Vergleich zu Programmen oder z. B. Datensätzen aus Datenbanken ist es bei der Komprimierung von Bildern nicht so wichtig, daß die Daten nach der Dekompression wieder exakt mit den Originaldaten übereinstimmen. Allgemein existieren zur Datenkompression die verschiedensten Kompressionsverfahren wie z. B. Huffman-Kodierung oder Shannon-

Fano-Kodierung. Diese Kodierungsverfahren sind in der Lage, Daten durch eine Kodierung in eine Darstellung zu überführen, die weniger Speicherplatz benötigt. Aus dieser Darstellung lassen sich die Originaldaten immer noch exakt rekonstruieren. Bei der Bildkompression kann man sich ferner den Umstand zu Nutze machen, daß das menschliche Auge nur eine begrenzte Zahl von Farben wahrnehmen kann, so daß eine exakte Rekonstruktion gar nicht notwendig ist. Für die Bildkompression benötigt man Kompressionsverfahren, die in der Lage sind, Farbabstufungen so anzugleichen, daß für das menschliche Auge kein sichtbarer Qualitätsverlust zum Original besteht, aber trotzdem mehr Speicherplatz eingespart wird als z. B. bei einer Huffman-Kodierung. Eines der gängigsten Kodierverfahren zur Bildkompression ist JPEG.

Im Jahre 1982 wurde bei der ISO eine Expertengruppe zusammengerufen, um einen Bildkompressionsstandard zu entwickeln. Das Ziel war ein progressives Datenkompressionsverfahren zu entwickeln, das in der Lage ist, mit der Datenrate von ISDN (64 Kbits/sec.) standzuhalten. Das komprimierte Bild sollte bereits nach einer Sekunde erkennbar sein und die Bildqualität sollte sich danach kontinuierlich verbessern, bis ein Bild ohne erkennbaren Qualitätsverlust entstanden ist. 1986 schloß sich die Expertengruppe der ISO mit einer weiteren Expertengruppe der CCITT zusammen. Nach dem Zusammenschluß wurde die Gruppe dann JPEG (Joint Photographic Experts Group) genannt.

Der JPEG-Kompressionsstandard sollte eine große Zahl an Funktionen enthalten. Dies waren sequentielle Kodierung, progressive Kodierung, Kodierung mit Verlust, verlustfreie Kodierung und die Möglichkeit der Hardware-Implementierung mit 64Kbit/s. Im Juni 1987 standen dann drei mögliche Techniken zur Kompression zur Auswahl, von denen dann schließlich 1988 die adaptive diskrete Cosinus-Transformation aufgrund der besten Bildqualität ausgewählt wurde. Es wurden weiterhin die funktionalen Anforderungen an die Bildkompression festgehalten. Im Jahre 1992 war die Arbeit an der JPEG-Kompression dann beendet.
Ein genauer Überblick über die Geschichte von JPEG findet sich in Kapitel 19 von [3].

1.3 Ziel der Arbeit

Ziel dieser Arbeit ist es zu ermitteln in wie weit man heutzutage in der Lage ist, Schaltungen und Systeme mit komplexen Datenpfaden mit Hilfe bestehender Modellprüfungsverfahren zu verifizieren.

Hierzu werden zwei verschiedene Versionen der diskreten Cosinus-Transformation mittels Modellprüfung gegeneinander verifiziert. Die diskrete Cosinus-Transformation ist ein wichtiger Bestandteil der JPEG-Kompression und wird daher in nahezu jedem bildverarbeitenden System eingesetzt. Es soll herausgefunden werden, bis zu welchen Eingangsbitbreiten und Schaltungsgrößen der unterschiedlichen Implementierungen eine Verifikation noch mit vertretbarem Aufwand möglich ist.

Es soll weiterhin herausgefunden werden, ob die beiden verwendeten Versionen der diskreten Cosinus-Transformation an allen Ausgängen die gleichen Ergebnisse liefern oder ob die Ergebnisse an den Ausgängen möglicherweise voneinander abweichen. Derartige Abweichungen kommen dadurch zustande, daß die mathematische Definition der DCT auf

reellen Zahlen beruht während Schaltungsrealisierungen Zahldarstellung begrenzter Genauigkeit verwenden. Durch verschiedene Berechnungen können sich somit Rechenungenauigkeiten ergeben. Sollten die Ergebnisse an einigen der Ausgänge voneinander abweichen, so soll herausgefunden werden, ob man mittels Modellprüfung zeigen kann, daß diese Differenz immer kleiner als eine zu ermittelnde obere Schranke ist.

Zur Durchführung der Modellprüfungsexperimente wurde auf den Modellprüfer SMV der Carnegie Mellon Universität (Pittsburg, USA) zurückgegriffen.

1.4 Gliederung der Arbeit

Im folgenden Abschnitt wird die Notwendigkeit der Hardware-Verifikation erläutert. Danach wird die temporale Logik CTL vorgestellt, die zur Spezifikation von Systemen dient. Systeme werden hierbei durch endliche Transitionssysteme (Kripke Strukturen) modelliert. CTL erlaubt nun eine effiziente Modellprüfung [8], welche auf der effizienten Speicherung von Zustandsmengen durch binäre Entscheidungsdiagramme [2] beruht. Daher befaßt sich Abschnitt 4 mit binären Entscheidungsdiagrammen. Der Abschnitt nach den binären Entscheidungsdiagrammen erläutert die mathematische Definition der diskreten Cosinus-Transformation sowie einige Optimierungen. Der letzte Abschnitt befaßt sich dann mit der praktischen Durchführung der Verifikation der beiden gewählten diskreten Cosinus-Transformationen sowie mit den experimentell ermittelten Ergebnissen.

2. Hardware-Verifikation

Bei Hardware- und auch bei Softwareprodukten können während der Entwicklung Fehler auftreten, die im späteren Betrieb zu Funktionsstörungen führen können. Diese Funktionsstörungen können eine Menge Geld kosten oder im schlimmsten Fall sogar Menschenleben gefährden. Während die Beseitigung eines Fehlers und die Auslieferung der neuen Version bei Softwareprodukten noch relativ einfach und billig zu bewerkstelligen ist, ist dies bei Hardwareprodukten weitaus kostspieliger.

Die Änderung eines Entwurfs auf einer hohen Abstraktionsebene ist hierbei nicht so aufwendig, wie die Änderung auf einer niedrigen Abstraktionsebene, da in einem solchen Fall viel Entwurfsarbeit noch einmal wiederholt werden müsste.
Das Ersetzen einer bereits gefertigten und ausgelieferten Komponente ist bei Hardwareprodukten eine sehr teure Angelegenheit. Ein gutes Beispiel hierfür ist der Fehler im Divisionswerk des Pentium-Prozessors von Intel. Die Beseitigung dieses Fehlers hat Intel ca. 100 Millionen Dollar gekostet.

Bei elektronischen Schaltungen können Fertigungsfehler, Spezifikationsfehler oder Entwurfsfehler zu Funktionsstörungen führen. Während Fertigungsfehler direkt nach der Fertigung durch Testen der Schaltung erkannt werden können, gibt es Entwurfsfehler, die möglicherweise erst nach vielen Betriebsstunden auftreten. Um diese Fehler zu vermeiden, benötigt man geeignete formale Modelle, um mit diesen die Fehlerfreiheit des Entwurfs gegenüber der Spezifikation mathematisch nachzuweisen.

Die vollständige Simulation aller möglichen Eingangsbelegungen, bei der alle internen Zustände der Schaltung erreicht werden, ist im Prinzip eine Möglichkeit der Verifikation der Schaltung, ist jedoch bei Schaltungen mit einer wachsenden Anzahl von Eingängen sehr schnell nicht mehr durchführbar, da die Anzahl der möglichen Eingangsbelegungen einer Schaltung exponentiell mit der Zahl der Eingänge und der Zahl der internen Zustände wächst.

Um zu zeigen, daß eine Implementierung der Spezifikation genügt, kann man sich statt dessen eines mathematischen Beweises bedienen in dem man zeigt, daß sich die Implementierung für alle Eingaben und alle Zeiten gemäß der Spezifikation verhält. Die Simulation wird dann zur Entdeckung von Spezifikationsfehlern verwendet.

Spezifikation sowie auch Implementierung werden vom Menschen vorgegeben. Beide müssen vollständig und fehlerfrei so beschrieben werden, daß mit ihnen ein formaler Beweis möglich ist. Bei der Spezifikation muß genau geklärt sein, welche Eigenschaften nachgewiesen werden sollen. Hierzu können z. B. funktionale Korrektheit, das Echtzeitverhalten, Sicherheitseigenschaften, Lebendigkeitseigenschaften oder Fairneß-Eigenschaften gehören. Besonders bei der Spezifikation muß darauf geachtet werden, daß diese das Gewünschte auch wirklich beschreibt. Spezifikationen sind nämlich meistens nicht in einer formalen Sprache gegeben, sondern in Umgangssprache verfaßt. Diese informellen Spezifikationen beinhalten eine Menge von Fehlerquellen, da sie oft unvollständig, unstrukturiert oder nicht eindeutig sind. Besonders bei der Umsetzung der informellen Spezifikation in eine formelle Darstellung müssen Fehler unbedingt vermieden werden. Die Verifikation kann nämlich nur die Korrektheit der Implementierung gegenüber der formellen Spezifikation überprüfen, nicht jedoch die Korrektheit gegenüber der informellen Spezifikation.

Die Umsetzung der Implementierung in einen zur Verifikation geeigneten Formalismus ist kein so großes Problem, da sie oftmals schon in einem maschinenlesbaren[1] Format wie z.B. VHDL vorliegt.

Die Verifikation von Hardware ist auch dann notwendig, wenn eine Schaltung in irgendeiner Hinsicht optimiert wurde. Es muß sichergestellt sein, daß sich die optimierte Schaltung für alle Eingaben und alle Zeiten genauso verhält wie die ursprüngliche Schaltung. Es ist jedoch genauso wichtig, auch die VHDL-Programme, welche Ausgangspunkt der Synthese sind, zu verifizieren.

Zur Verifikation von Hardware existieren verschiedene Verifikationsarten, wie die horizontale oder die vertikale Verifikation sowie verschiedene Verifikationsformalismen [6,7]. Besonders wichtig sind temporale Logiken wie CTL, LTL oder CTL* [9]. Im nächsten Abschnitt soll einer dieser Formalismen, nämlich die temporale Logik CTL beschrieben werden.

3. CTL

Bei temporalen Logiken wie CTL [10] werden Systeme durch sogenannte Kripke-Strukturen modelliert. Ein Beipiel einer Kripke-Struktur zeigt Abb. 1.

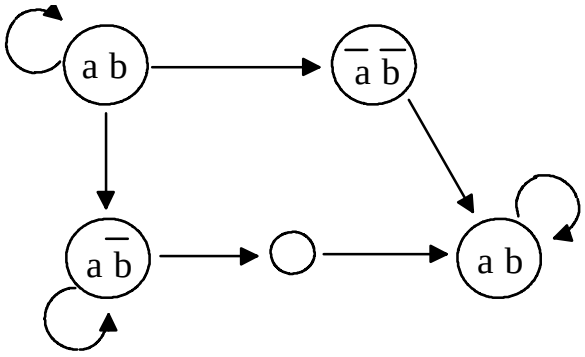


Abb. 1: Kripke Struktur zur Systemmodellierung

Eine solche Struktur besteht aus endlich vielen Zuständen, welche durch Transitionen miteinander verbunden sind. Jeder Zustand beschreibt einen Systemzustand und ist mit den Variablen markiert, die in dem Zustand gelten sollen. Die Markierung entspricht einer Eingabebelegung sowie der Belegung eines internen Zustands.

Jeder Pfad entspricht dem Ablauf des Systems unter einer Eingabesequenz. Jeder Zustand ist ein Systemzustand. Die Variablen, welche in einem Zustand gelten, beschreiben diesen Zustand, wobei es möglich ist, daß verschiedene Zustände dieselben Variablen aufweisen

[1] Ein solches maschinenlesbares Format muß eine eindeutig definierte Syntax, sowie eine eindeutig definierte Semantik haben. Bei VHDL existiert derzeit offiziell allerdings nur eine informelle Semantik.

(siehe Abb. 1). Der Folgezustand eines Zustandes wird nichtdeterministisch ausgewählt. Es wird davon ausgegangen, daß eine Transition eine Zeiteinheit verbraucht.

Temporale Logiken erlauben nun, Aussagen über diese Systemzustände bzw. Pfade zu machen.

Dazu verwenden diese Logiken neben den aussagenlogischen Operatoren zwei Arten von modalen Operatoren, die dem zugrundeliegenden Zeitmodell entsprechen. Die eine Art der Operatoren macht Aussagen über einen bestimmten Pfad, d.h. Aussagen über zeitliche Beziehungen zwischen Variablen auf den Zuständen. Eine Formel kann für jeden Zustand eines Pfades gelten (G oder *generally*), nur für mindestens einen möglichen Folgezeitpunkt des Pfades (F oder *sometimes*), oder man kann über den unmittelbaren Folgezustand Aussagen machen (X oder *next*). Man kann außerdem zwei Formeln derart miteinander verknüpfen, daß die erste solange wahr sein muß, bis die zweite gilt (U oder *until*).
Die zweite Art von Operatoren macht Aussagen darüber, ob die Formel auf allen Folgepfaden (A oder *for All paths*) oder nur auf einem einzigen Folgepfad (E oder *there Exists a path*) gelten muß.

CTL ist eine verzweigende temporale Logik, d. h. in jedem Zustand können Aussagen über alle oder einen Pfad gemacht werden. Die Auswahl zwischen diesen möglichen Pfaden erfolgt nichtdeterministisch.

Die Syntax von CTL beruht auf zwei Formelklassen. Es gibt Zustandsformeln, die in einem bestimmten Zustand wahr werden und Pfadformeln, deren Wahrheitswert von einem gegebenen Pfad einer Struktur abhängt.

Definition der Syntax von CTL: Sei V die Menge der aussagenlogischen Variablen, mit denen die Kripke-Struktur markiert ist, dann definiert man die Menge der Zustandsformeln rekursiv:

i.) Ist $\varphi \in V$, so ist φ eine CTL-Formel.
ii.) Sind φ und ψ CTL-Formeln, so sind $\neg\varphi$ und $\varphi \vee \psi$ CTL-Formeln.
iii.) Sind φ und ψ CTL-Formeln, so sind $EX\varphi$, und $E(\varphi U \psi)$ CTL-Formeln.

Zur besseren Schreibweise wird zusätzlich noch der Und-Operator eingeführt:

$$\varphi \wedge \psi :\Leftrightarrow \neg(\varphi \vee \psi)$$

Weitere Operatoren lassen sich alle mit EX und EU ausdrücken.

$AX\varphi \leftrightarrow \neg EX(\neg\varphi)$; für alle unmittelbaren Folgezustände gilt φ
$AG\varphi \leftrightarrow \neg E(trueU(\neg\varphi))$; für alle Pfade gilt immer φ
$AF\varphi \leftrightarrow A(trueU\varphi)$; für alle Pfade gilt in mindestens einem Folgezustand φ
$EF\varphi \leftrightarrow E(true\ U\ \varphi)$; es existiert ein Pfad auf dem in mindestens einem Folgezustand φ gilt
$A(\varphi U \psi) \leftrightarrow \neg E(\neg\psi U \neg\varphi \wedge \neg\psi) \wedge \neg\ EG(\neg\psi)$; für alle Pfade gilt solange φ bis ψ gilt
$EG\varphi \leftrightarrow \neg A(trueU(\neg\varphi))$; es gibt einen Pfad auf dem immer φ gilt

Bei CTL muß jedem Operator F, G, X, U ein Pfadquantor A oder E vorangestellt werden. Damit sind nur die obigen acht Basisoperatoren AX, EX, AG, EG, AF, EF, AU und EU in CTL zulässig.

Ist neben einer temporallogischen Formel auch eine Struktur gegeben, so muß zur Modellprüfung gezeigt werden in welchen Zuständen der Struktur die Formel gilt. Für die Logik CTL existieren effiziente Modellprüfungsverfahren, die auf Fixpunktcharakterisierungen der temporalen Operatoren beruhen. Grundlage dieser Fixpunktcharakterisierungen sind Funktionen, die Mengen von Zuständen einer Struktur wieder auf Zustandsmengen abbilden [11]. Diese Zustandsmengen werden durch charakteristische Funktionen repräsentiert, die selbst wieder Boolesche Funktionen sind. Man benötigt also eine effiziente Darstellung von Booleschen Funktionen. Diese effiziente Darstellung kann durch binäre Entscheidungsdiagramme geschehen.

4. Binäre Entscheidungsdiagramme

Aus einer Wertetabelle kann man den Funktionswert einer booleschen Funktion $f : B^n \rightarrow B$ für eine konkrete Variablenbelegung direkt ablesen. Die Darstellung der gesamten Funktionstabelle erfordert jedoch einen Speicheraufwand der Größenordnung $O(2^n)$. Eine aussagenlogische Formel andererseits ist eine sehr kompakte Darstellung einer Booleschen Funktion. Der Test, ob eine aussagenlogische Formel eine Tautologie ist oder ob sie überhaupt erfüllbar ist, ist jedoch NP-vollständig [12].

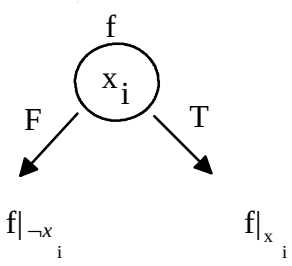
OBDDs erlauben hingegen eine Darstellung für Boolesche Funktionen, die meist weniger Speicherplatz benötigt als Wertetabellen, bei der aber der Erfüllbarkeits- bzw. Gültigkeitstest einfacher ist als bei der Formeldarstellung.

Betrachtet man eine Boolesche Funktion $f(x_1,...,x_n)$, so nennt man den Ausdruck $f(x_1,...,x_{i-1},T,...,x_n)$ den Kofaktor von f nach x_i (geschrieben $f|_{x_i}$). Der Ausdruck $f(x_1,...,x_{i-1},F...,x_n)$ ist der Kofaktor von f nach $\neg x_i$ (geschrieben $f|_{\neg x_i}$).

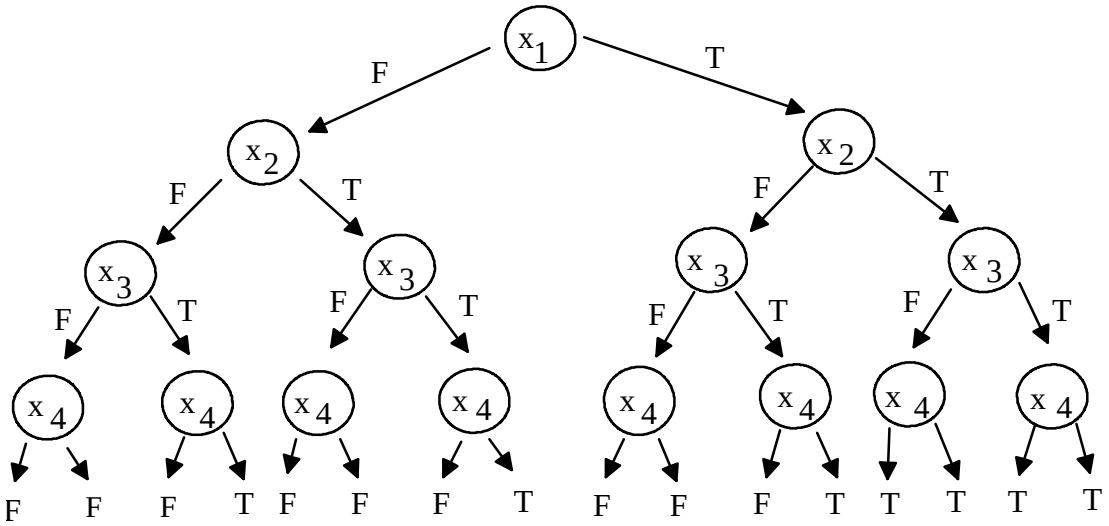
Es gilt der Entwicklungssatz [1]: $f(x_1,...,x_i,...,x_n) = (x_i \wedge f|_{x_i}) \vee (\neg x_i \wedge f|_{\neg x_i})$

Man kann also eine Funktion f mit n Variablen auf zwei neue Funktionen mit n-1 Variablen zurückführen.

Diese Kofaktorbildung läßt sich auch graphisch veranschaulichen:



Beispiel 1: binäres Entscheidungsdiagramm der Formel $f(x_1,x_2,x_3,x_4) := x_1x_2 \vee x_3x_4$



Entwickelt man eine Boolesche Funktion nach allen Variablen und verwendet die obige graphische Notation, so erhält man eine Baumdarstellung der Funktion an deren Blättern die Funktionswerte für die 2^n verschiedenen Belegungen zu finden sind. Der Baum entspricht also einer vollständigen Wertetabelle. Diese Darstellung besitzt gegenüber den Wertetabellen noch keinen nennenswerten Vorteil. Die Baumstruktur kann jedoch redundante Teile, wie z.B. identische Teilbäume oder Entscheidungsknoten, die nicht benötigt werden, enthalten, die sich eliminieren lassen. Hierdurch erreicht man eine wesentlich kompaktere Darstellung.

Diese kompakte Darstellung nennt man ein binäres Entscheidungsdiagramm [2].

Bei fester Variablenordnung ist das binäre Entscheidungsdiagramm eindeutig [2]. Der Tautologietest läßt sich dann auf einen Vergleich mit dem 1-Blatt reduzieren.
Da geordnete und reduzierte binäre Entscheidungsdiagramme eindeutig sind, läßt sich mit ihnen der semantische Vergleich zweier Formeln durch einen syntaktischen Vergleich der binären Entscheidungsdiagramme durchführen d.h. syntaktischer und semantischer Vergleich

sind hier identisch. Der semantische Vergleich zweier aussagenlogischer Formeln ist dagegen wesentlich aufwendiger als deren syntaktischer Vergleich.

Die Größe der binären Entscheidungsdiagramme hängt entscheidend von der gewählten Variablenordnung ab (siehe Experimente).

5. Die diskrete Cosinus-Transformation

Die diskrete Cosinus-Transformation (DCT) ist ein wichtiger Bestandteil von verschiedenen Datenkompressionsverfahren. Sie wird z.B. zur Bildkompression eingesetzt und ist Bestandteil der JPEG- und MPEG-Kompression bei der große Einsparungen in Bezug auf den benötigten Speicher erzielt werden, ohne daß dabei sichtbare Qualitätsverluste hingenommen werden müßten.

5.1 Die eindimensionale DCT

Die eindimensionale DCT ist definiert als lineare Funktion $\Phi : \Re^8 \rightarrow \Re^8$

$$\begin{pmatrix} y_0 \\ \vdots \\ y_7 \end{pmatrix} := \underbrace{\begin{pmatrix} a_{0,0} & \cdots & a_{0,7} \\ \vdots & \cdots & \vdots \\ a_{7,0} & \cdots & a_{7,7} \end{pmatrix}}_{=: \mathcal{A}} \cdot \begin{pmatrix} x_0 \\ \vdots \\ x_7 \end{pmatrix}$$

wobei die $a_{i,j} := \frac{1}{2}\cos((2j+1)i\frac{\pi}{16})$ *für* $i > 0$ *und* $a_{0,j} := \frac{1}{2\sqrt{2}}$.

Die Matrix A ist orthonormal und nicht damit singulär, so daß die Funktion bijektiv ist. Die inverse Matrix einer orthonormalen Matrix ist die transponierte Matrix: $A^{-1} = A^t$. Die Elemente der inversen Matrix $B = A^{-1}$ sind somit $\beta_{i,j} := \alpha_{j,i}$.

Die Implementierung der inversen DCT ist der Implementierung der DCT also sehr ähnlich.

5.2 Die zweidimensionale DCT

Da Bilder zweidimensional sind, benötigt man zur Bildkompression die zweidimensionale DCT.

Die zweidimensionale DCT ist eine lineare Funktion $\Psi : \Re^8 \times \Re^8 \rightarrow \Re^8 \times \Re^8$

$$y_{i,j} := \sum_{k=0}^{7} \left(\sum_{l=0}^{7} x_{k,l} ; \alpha_{j,l} \right) ; \alpha_{i,k}$$

Die zweidimensionale DCT läßt sich durch eindimensionale DCT's berechnen. Mit einer Übergangsmatrix U läßt sie sich schreiben als:

$$u_{j,k} := \sum_{l=0}^{7} x_{k,l} ; \alpha_{j,l} \qquad y_{i,j} := \sum_{k=0}^{7} u_{j,k} ; \alpha_{i,k} \qquad y = AXA^{-1}$$

Die zweidimensionale DCT läßt sich also durch 16 eindimensionale DCT implementieren. Hierbei werden 8 DCT's für die Zeilen und 8 für die Spalten benötigt. Dies ist jedoch nicht so effizient wie echte zweidimensionale Ansätze [3]. Dennoch existieren sehr effiziente eindimensionale Verfahren, welche auch zu fast optimalen zweidimensionalen Lösungen führen.

Im folgenden wird daher nur noch die eindimensionale DCT betrachtet.

5.3 Optimierungen der eindimensionalen DCT

Die Implementierung der eindimensionalen DCT kann vereinfacht werden, wenn man Symmetrien ausnutzt, die sich aus der Verwendung der trigonometrischen Funktionen ergeben. Alle benutzten trigonometrischen Werte lassen sich auf 8 Basiswerte reduzieren.

Sei $C(k) := \cos(k\frac{\pi}{16})$ und $S(k) := \sin(k\frac{\pi}{16})$ für $k \in Z$, dann gelten die folgenden Gleichungen:

1. C(-k) = C(k)
2. C(k) = C(k mod 32)
3. C(k) = C(32-k)
4. C(k) = -C(16-k)
5. $C(0) = 1, C(4) = \frac{\sqrt{2}}{2}, C(8) = 0$
6. S(k) = C(8-k)

Die erste Gleichung erlaubt die Reduktion der Werte aus Z auf Werte $k \geq 0$. Die zweite Gleichung reduziert diese Werte auf Werte aus {0, ...,31}, die dritte auf Werte aus {0, ...,16}, und die vierte Gleichung erlaubt die Reduktion auf Werte aus {0, ...,8}. Gleichung 5 stellt einige Spezialfälle dar. Gleichung 6 erlaubt die Darstellung von S(k) durch C(k).

Durch Benutzung dieser 6 Gleichungen läßt sich die eindimensionale DCT wie folgt darstellen, wozu nur noch C(1), ..., C(7) benötigt werden.

$2y_0 := C(4)(x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7)$
$2y_1 := C(1)x_0+C(3)x_1+C(5)x_2+C(7)x_3-C(7)x_4-C(5)x_5-C(3)x_6-C(1)x_7$
$2y_2 := C(2)x_0+C(6)x_1-C(6)x_2-C(2)x_3-C(2)x_4-C(6)x_5+C(6)x_6+C(2)x_7$
$2y_3 := C(3)x_0-C(7)x_1-C(1)x_2-C(5)x_3+C(5)x_4+C(1)x_5+C(7)x_6-C(3)x_7$
$2y_4 := C(4)x_0-C(4)x_1-C(4)x_2+C(4)x_3+C(4)x_4-C(4)x_5-C(4)x_6+C(4)x_7$
$2y_5 := C(5)x_0-C(1)x_1+C(7)x_2+C(3)x_3-C(3)x_4-C(7)x_5+C(1)x_6-C(5)x_7$
$2y_6 := C(6)x_0-C(2)x_1+C(2)x_2-C(6)x_3-C(6)x_4+C(2)x_5-C(2)x_6+C(6)x_7$
$2y_7 := C(7)x_0-C(5)x_1+C(3)x_2-C(1)x_3+C(1)x_4-C(3)x_5+C(5)x_6-C(7)x_7$

Abb. 2: Eindimensionale DCT als Gleichungssystem.

Zur Nutzung gemeinsamer Teilterme werden diese Gleichungen nun folgendermaßen zusammengefaßt:

$L_{0,0} := x_0 + x_7$	$L_{1,0} := L_{0,0} + L_{0,3}$	$2y_0 := C(4)(L_{1,0} + L_{1,1})$
$L_{0,1} := x_1 + x_6$	$L_{1,1} := L_{0,1} + L_{0,2}$	$2y_1 := C(1)L_{0,7} + C(3)L_{0,6} + C(5)L_{0,5} + C(7)L_{0,4}$
$L_{0,2} := x_2 + x_5$	$L_{1,2} := L_{0,0} - L_{0,3}$	$2y_2 := C(2)L_{1,2} + C(6)L_{1,3}$
$L_{0,3} := x_3 + x_4$	$L_{1,3} := L_{0,1} - L_{0,2}$	$2y_3 := C(3)L_{0,7} - C(7)L_{0,6} - C(1)L_{0,5} - C(5)L_{0,4}$
$L_{0,4} := x_3 - x_4$		$2y_4 := C(4)(L_{1,0} - L_{1,1})$
$L_{0,5} := x_2 - x_5$		$2y_5 := C(5)L_{0,7} - C(1)L_{0,6} + C(7)L_{0,5} + C(3)L_{0,4}$
$L_{0,6} := x_1 - x_6$		$2y_6 := C(6)L_{1,2} - C(2)L_{1,3}$
$L_{0,7} := x_0 - x_7$		$2y_7 := C(7)L_{0,7} - C(5)L_{0,6} + C(3)L_{0,5} - C(1)L_{0,4}$

Abb. 3: Eindimensionale DCT nach Zusammenfassen von gemeinsamen Subtermen

Diese Implementierung benötigt nur noch 22 Multiplikationen und 28 Additionen statt der vorher nötigen 64 Multiplikationen und 56 Additionen.
Das Wiederverwenden gemeinsamer Subterme, die durch die Ausnutzung der Symmetrien von cos und sin entstanden sind, führt zu einer ersten einfachen Optimierung.

Eine weitere Optimierung kann durch die Verwendung einer auf den trigonometrischen Gesetzen beruhenden Rotationsoperation erreicht werden. Die Rotation um einen Winkel α ist definiert als:

$$\begin{pmatrix} y_0 \\ y_1 \end{pmatrix} := \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix}$$

Diese Form der Rotation benötigt 4 Multiplikationen und 2 Additionen. Da Multiplikationen bei der Darstellung mit OBDDs einen exponentiellen Aufwand verursachen, sind sie bei der

Verifikation besser zu vermeiden. Es werden daher die folgenden Formeln zur Berechnung der Rotation benutzt:

$$l := \cos(\alpha)(x_0 + x_1)$$
$$y_0 := l + (\sin(\alpha) - \cos(\alpha))x_1$$
$$y_1 := -(\sin(\alpha) + \cos(\alpha))x_0 + l$$

Hierbei werden 3 Multiplikationen und 3 Additionen benötigt. Die Anzahl der Additionen steigt zwar an, dafür sinkt aber die Anzahl der Multiplikationen, welche i.d.R. weit mehr Kosten und Aufwand verursachen.
Im Folgenden sei

$$ROT_0(x_0, x_1, \alpha) := x_0 C(\alpha) + x_1 S(\alpha)$$
und $$ROT_1(x_0, x_1, \alpha) := -x_0 S(\alpha) + x_1 C(\alpha),$$

zur Berechnung wird jedoch die oben erwähnte schnellere Variante der Rotation eingesetzt.

Die DCT nach Löffler, Ligtenberg und Moschytz [4] läßt sich nun folgendermaßen darstellen:

$L_{0,0} := x_0 + x_7$
$L_{0,1} := x_1 + x_6$
$L_{0,2} := x_2 + x_5$
$L_{0,3} := x_3 + x_4$
$L_{0,4} := x_3 - x_4$
$L_{0,5} := x_2 - x_5$
$L_{0,6} := x_1 - x_6$
$L_{0,7} := x_0 - x_7$

$L_{1,0} := L_{0,0} + L_{0,3}$
$L_{1,1} := L_{0,1} + L_{0,2}$
$L_{1,2} := L_{0,1} - L_{0,2}$
$L_{1,3} := L_{0,0} - L_{0,3}$
$L_{1,4} := ROT_0(L_{0,4}, L_{0,7}, 3)$
$L_{1,5} := ROT_0(L_{0,5}, L_{0,6}, 1)$
$L_{1,6} := ROT_1(L_{0,5}, L_{0,6}, 1)$
$L_{1,7} := ROT_1(L_{0,4}, L_{0,7}, 3)$

$L_{2,0} := L_{1,1} + L_{1,0}$
$L_{2,1} := L_{1,1} - L_{1,0}$
$L_{2,2} := ROT_0(L_{1,2}, L_{1,3}, 6)$
$L_{2,3} := ROT_1(L_{1,2}, L_{1,3}, 6)$
$L_{2,4} := L_{1,4} + L_{1,6}$
$L_{2,5} := L_{1,7} - L_{1,5}$
$L_{2,6} := L_{1,4} - L_{1,6}$
$L_{2,7} := L_{1,7} + L_{1,5}$

$z_0 := L_{2,0}$
$z_1 := L_{2,4} + L_{2,7}$
$z_2 := 2\ C(4)\ L_{2,2}$
$z_3 := 2\ C(4)\ L_{2,5}$
$z_4 := L_{2,1}$
$z_5 := 2\ C(4)\ L_{2,6}$
$z_6 := 2\ C(4)\ L_{2,3}$
$z_7 := L_{2,7} - L_{2,4}$

Abb. 4: Eindimensionale DCT nach Verwendung von Rotationen

Um diese DCT zu implementieren benötigt man lediglich 29 Additionen, 13 Multiplikationen und 4 Shifter-Operationen. Diese DCT ist also weitaus billiger als die erste Version. Obwohl

die Zahl der Additionen steigt und zusätzlich 4 Shifter benötigt werden, ist die DCT insgesamt billiger, da die Anzahl der Multiplikationen erheblich gesunken ist.

Es gibt noch weitere Optimierungen, die die Berechnung der DCT noch schneller machen. Eine solche stammt von Arai, Agui und Nakajiama [3]. Bei dieser Implementierung benötigt man 23 Additionen und 5 Multiplikationen. Zwar steigt die Zahl der benötigten Additionen im Vergleich zu der Variante von Löffler, Ligtenberg und Moschytz wieder etwas an, aber dafür sinkt die Zahl der benötigten Multiplikationen auf weniger als die Hälfte, was einen erheblich geringeren Aufwand bei der Implementierung und eine höhere Geschwindigkeit bedeutet.

6. Durchführung

6.1 Ziele und Ansätze

In dieser Arbeit sollen unterschiedliche Implementierungen der eindimensionalen diskreten Cosinus-Transformation gegeneinander verifiziert werden. Es soll untersucht werden, ob die beiden Implementierungen für gleiche Eingaben auch die gleichen Ausgaben liefern. Aufgrund von Rechenungenauigkeiten, welche durch die begrenzte Genauigkeit der von der Schaltung verwendeten Zahldarstellung gegeben sind, ist jedoch auch zu erwarten, daß eventuell Ausgänge nicht exakt gleich sein werden. Da die Verifikation für mehrere unterschiedliche Bitbreiten erfolgen soll, ist eine erschöpfende Simulation aller möglichen Eingangsbelegungen und der dazugehörigen Ausgangsbelegungen für die beiden Implementierungen nicht machbar. Bei 4 Bit Eingangsbitbreite erhält man bereits $2^{8 \cdot 4} = 2^{32}$ verschiedene Belegungen! Zur Verifikation wird daher Modellprüfung eingesetzt.

Die Experimente wurden auf einer SUN Ultra 1 Creator 3D mit 512 MB Hauptspeicher unter dem Betriebssystem SUN Solaris 2.5.5 durchgeführt. Der zur Verifikation verwendete Modellprüfer ist SMV Version 2.4.3 [11] mit einer vorgegebenen Cachegröße von 1046429 Byte. Sollte im Folgenden eine andere Software oder eine andere Version des SMV verwendet worden sein, so wird dies ausdrücklich erwähnt.

Die beiden verschiedenen Versionen der DCT, die implementiert wurden, sind die DCT aus Abb. 3 und die DCT aus Abb. 4. Diese beiden Versionen der DCT wurden nur für ganze Zahlen implementiert, da Hardware-Implementierungen keine Darstellung der reelen Zahlen erlauben. Dabei wird eine Festpunktdarstellung verwendet, bei der keine Nachkommastellen auftreten (Integer-DCT).

Da die Verifikation für unterschiedliche Bitbreiten durchgeführt werden soll, ist es von Vorteil, wenn man die entsprechenden Implementierungen der DCT für die jeweilige Bitbreite automatisch erzeugen kann. Dies kann z. B. bei der Fehlersuche behilflich sein, da man Fehler in einer Implementierung mit einer geringen Bitbreite leichter entdecken kann. Es ist außerdem sehr einfach, nach Beseitigung eines Fehlers die Implementierungen für die anderen Bitbreite wieder zu erzeugen. Man erspart sich so die Korrektur von Hand in allen anderen Implementierungen.

Zur automatischen Erzeugung der verschiedenen Implementierungen wurde für jede der beiden Versionen der DCT ein C-Programm implementiert, das in der Lage ist, eine Datei in SMV-Syntax zu erzeugen, die eine Beschreibung der jeweiligen DCT für eine vorgegebene Bitbreite enthält. Diesem Modul muß dann zur Erzeugung einer konkreten Implementierung nur noch die gewünschte Bitbreite übergeben werden. Die generischen Module basieren auf einer Bibliothek, die die gängigsten Schaltungen wie z. B. Addierer, Multiplizierer, Vergleicher, Shifter, Multiplexer usw. wiederum als generische Module enthält.

Die SMV-Beschreibungen enthalten dann für jedes benutzte Modul aus dieser Bibliothek eine konkrete Instanz des Moduls.

Die erste Version der DCT, die implementiert wurde, ist die DCT aus Abb. 3. Eine schematische Darstellung der Schaltung ist in Abb. 5 zu sehen. Hierbei ist anzumerken, daß in der Implementierung die Gleichung

$$2y_3 := C(3)L_{0,7} - C(7)L_{0,6} - C(1)L_{0,5} - C(5)L_{0,4}$$

umgewandelt wurde zu

$$2y_3 := C(3)L_{0,7} - C(7)L_{0,6} - (C(1)L_{0,5} + C(5)L_{0,4})$$

was die Berechnung eines Zweierkomplements erspart.

Die andere implementierte Version der DCT ist die aus Abb. 4. Eine schematische Darstellung dieser Implementierung findet sich in Abb. 6.

Die Bitbreiten an den Ausgängen der zu vergleichenden Schaltungen sind unterschiedlich groß. Dies kommt daher, daß die Bausteine, aus denen die Schaltungen aufgebaut sind, feste Bitbreiten für die Eingänge und Ausgänge haben. So hat z. B. ein Addierer mit einer Eingangsbitbreite von n Bit eine Ausgangsbitbreite von n+1 Bit und ein Multiplizierer mit 2n Eingängen hat auch 2n Ausgänge.

Unterschiede in der Bitbreite kommen durch zusätzliche oder fehlende Skalierungen.

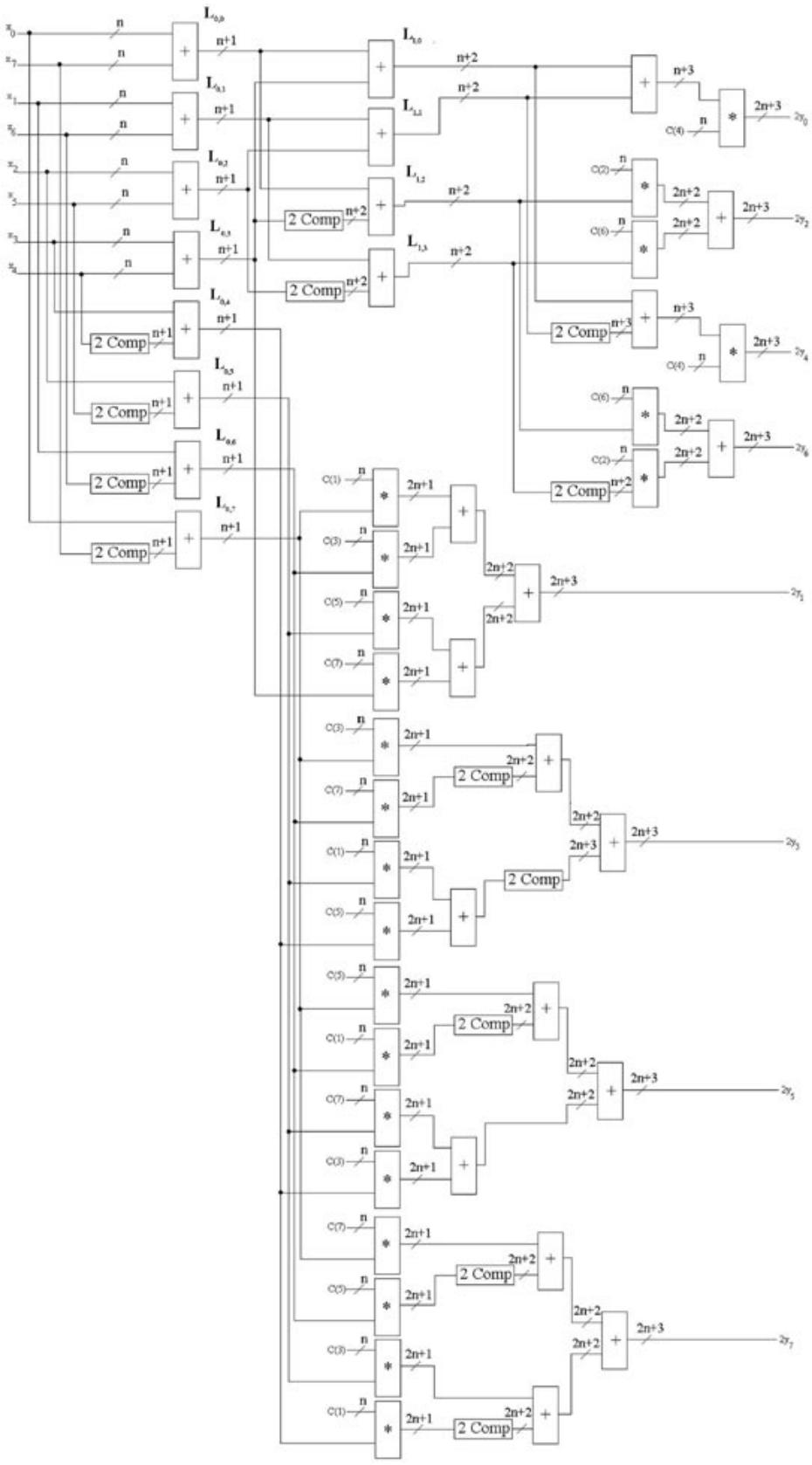


Abb.5: Schematische Darstellung der DCT aus Abb. 3

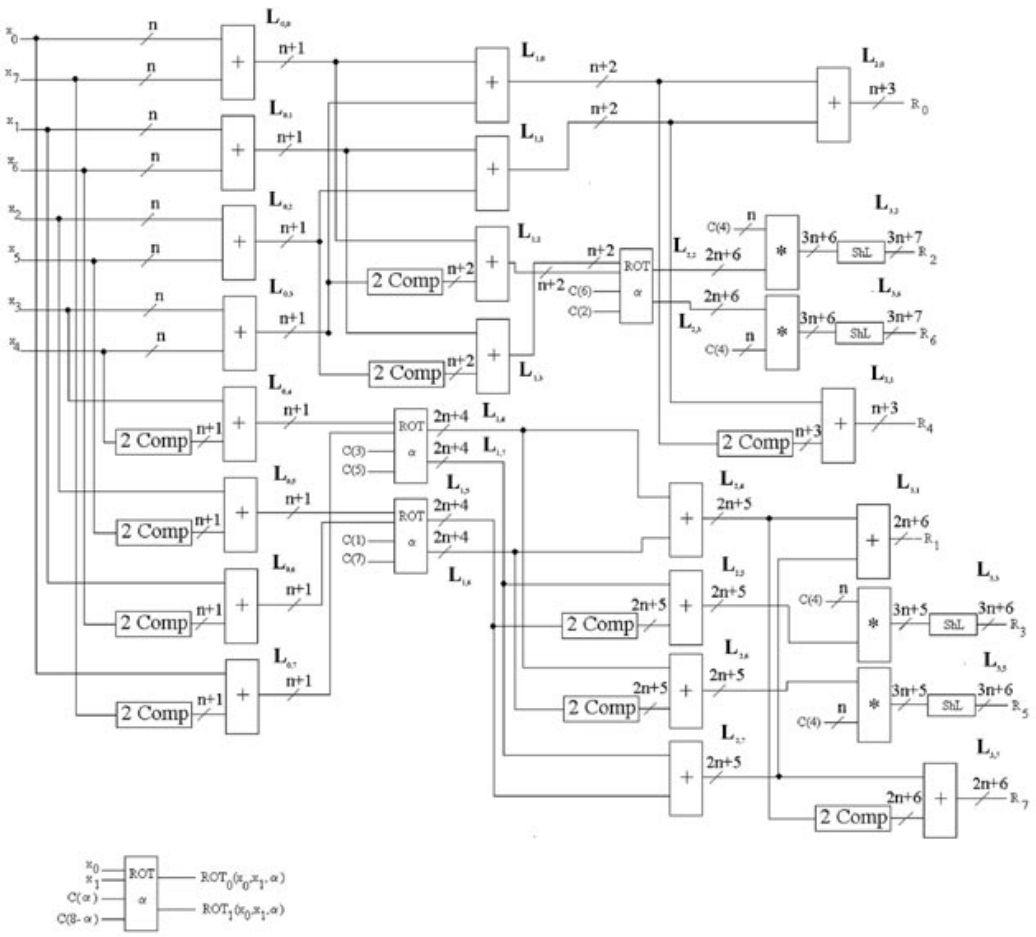


Abb. 6: Schematische Darstellung der DCT aus Abb. 4

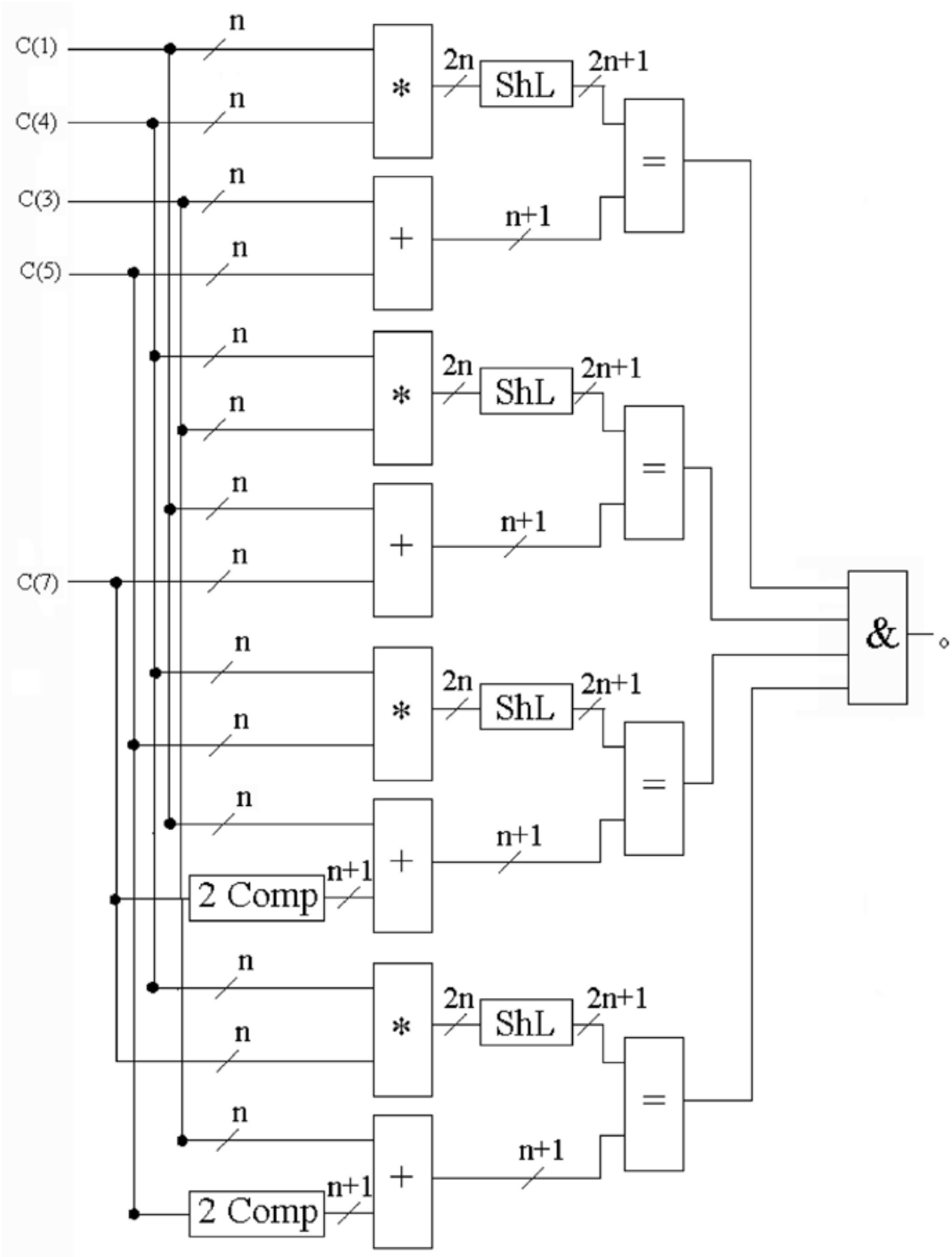


Abb. 7: Schematische Darstellung der Additiontheoreme

6.2 Ermittlung der Konstanten für die Additionstheoreme

Um diese beiden Versionen der DCT vollständig zu implementieren und dann miteinander vergleichen zu können, werden noch konkrete Werte für die Konstanten C(1), C(2), C(3), C(4), C(5), C(6) und C(7) benötigt. Nach Definition ist C(i):=cos($\frac{i\pi}{16}$), so daß alle gesuchten C(i) irrationale Zahlen sind. Diese sind somit nicht mit endlich vielen Bits darzustellen. Eine genaue Analyse ergibt jedoch, daß zur Verifikation lediglich folgende Additionstheoreme notwendig sind:

$$\cos(x) + \cos(y) = 2\cos(\tfrac{x+y}{2})\cos(\tfrac{x-y}{2})$$
$$\cos(x) - \cos(y) = 2\sin(\tfrac{x+y}{2})\sin(\tfrac{x-y}{2})$$

Daher stellt sich nun die Frage, ob es n-Bit Zahlen $\widetilde{C}(1), \widetilde{C}(3), \widetilde{C}(5), \widetilde{C}(7)$ gibt, welche folgende Gesetze erfüllen.

Zur Ermittlung der Werte werden nur die folgenden Gleichungen, die sich aus den Additionstheoremen ergeben, benötigt. Diese Gleichungen werden nun in der Schreibweise der DCT dargestellt.

$$(1) \quad 2\widetilde{C}(4)\widetilde{C}(1) := \widetilde{C}(3) + \widetilde{C}(5)$$
$$(2) \quad 2\widetilde{C}(4)\widetilde{C}(3) := \widetilde{C}(1) + \widetilde{C}(7)$$
$$(3) \quad 2\widetilde{C}(4)\widetilde{C}(5) := \widetilde{C}(1) - \widetilde{C}(7)$$
$$(4) \quad 2\widetilde{C}(4)\widetilde{C}(7) := \widetilde{C}(3) - \widetilde{C}(5)$$

Abb. 8: Additionstheoreme

Für diese vier Gleichungen wurde nun ebenfalls ein generisches SMV-Modul geschaffen, das in der Lage ist, ein SMV-Modul für eine vorgegebene Bitbreite zu erzeugen. Eine schematische Darstellung dieser Schaltung ist in Abb. 7 zu sehen.

Dieser SMV-Beschreibung der Additionstheoreme wurde eine Spezifikation angefügt, die besagt, daß es keine 5 von Null verschiedene ganzen Zahlen gibt, die die obigen Gleichungen erfüllen. SMV versucht dann diese Spezifikation zu beweisen, was bedeutet, daß es für diese Bitbreite eben keine solche Zahlen gibt oder andernfalls einen Gegenbeweis anzutreten indem 5 solche Zahlen genannt werden. Sollte SMV in der Lage sein für eine gewisse Bitbreite n solche Zahlen zu finden, die alle 4 Gleichungen erfüllen, so gelten diese Zahlen natürlich auch für alle größeren Bitbreiten $m \geq n$. Die triviale Lösung, bei der alle 5 Zahlen gleich Null sind, wollen wir natürlich ausschließen.

Die Ergebnisse finden sich in Tabelle 1 sowie in den Abbildungen 9 und 10. Das Experiment zur Ermittlung der Konstanten für 12 Bit wurde nach über 400 Stunden Rechenzeit ohne Ergebnis abgebrochen.

Bitbreite	BDD-Knoten	Bytes	Spezifikation erfüllt ?
2	957	17.498.112	ja
4	26.599	18.087.936	ja
6	577.803	27.262.976	ja
8	11.731.678	206.110.720	ja

Tabelle 1: Experimentelle Ergebnisse zur Ermittlung von Konstanten für die Additionstheoreme

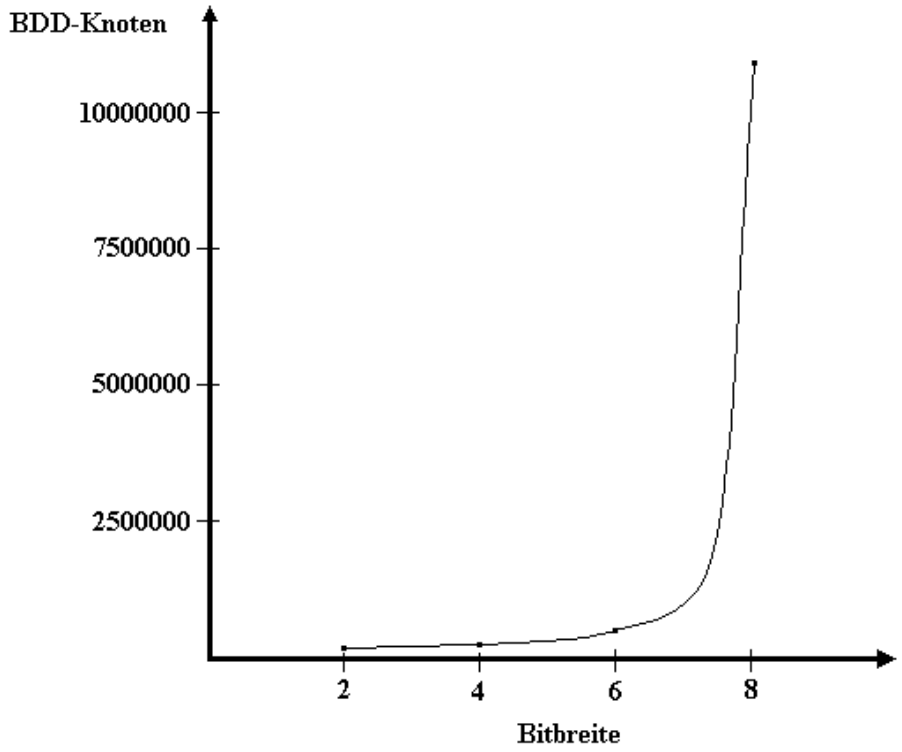


Abb.9: Benötigte BDD-Knoten für Additionstheoreme mit SMV 2.4.3

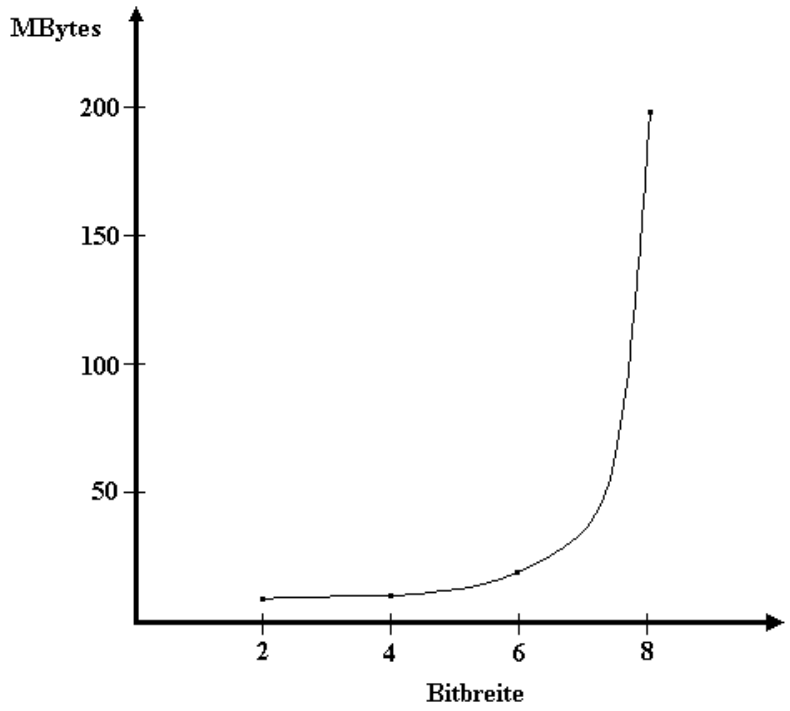


Abb.10: Benötigte MBytes für Additionstheoreme mit SMV 2.4.3

Um den Ressourcenbedarf bei den nächsten Durchläufen zu senken, wird zunächst versucht, eine bzgl. Ressourcenbedarf optimierte Ordnung der Variablen zu bestimmen. Hierzu wird die SMV-Version 2.4.4 verwendet, da die Version 2.4.3 nicht in der Lage ist, eine solche Umordnung vorzunehmen. Die jeweiligen neuen Variablenordnungen werden dann später in Versuchen verwendet. Die Ergebnisse der Durchläufe der Additionstheoreme mit SMV-Version 2.4.4 finden sich in Tabelle 2 sowie in den Abbildungen 11 und 12. In Tabelle 3 ist die jeweilige neue Variablenordnung dargestellt.

Bitbreite	Systemzeit	BDD-Knoten	Bytes	Spezifikation erfüllt ?
2	0.17 s	957	17.498.112	ja
4	0.19 s	6.777	17.891.328	ja
6	0.26 s	51.694	19.464.192	ja
8	0.58 s	553.328	35.651.584	ja

Tabelle 2: Ressourcenverbrauch bei SMV-Durchläufen für die Additionstheoreme mit Version 2.4.4

Bitbreite	Neue Variablenordnung (der Index bezeichnet die Bitposition)
2	$C(3)_1$, $C(3)_0$, $C(5)_0$, $C(5)_1$, $C(4)_0$, $C(4)_1$, $C(1)_0$, $C(1)_1$, $C(7)_1$, $C(7)_0$
4	$C(1)_3$, $C(7)_3$, $C(1)_2$, $C(7)_2$, $C(3)_3$, $C(5)_3$, $C(3)_2$, $C(5)_2$, $C(3)_1$, $C(5)_1$, $C(5)_0$, $C(3)_0$, $C(4)_0$, $C(4)_1$, $C(4)_2$, $C(4)_3$, $C(1)_1$, $C(1)_0$, $C(7)_1$, $C(7)_0$
6	$C(3)_4$, $C(5)_4$, $C(3)_5$, $C(5)_5$, $C(3)_3$, $C(1)_5$, $C(5)_3$, $C(5)_0$, $C(3)_0$, $C(5)_1$, $C(5)_2$, $C(4)_4$, $C(3)_1$, $C(3)_2$, $C(4)_5$, $C(4)_3$, $C(4)_2$, $C(4)_1$, $C(4)_0$, $C(7)_5$, $C(1)_4$, $C(7)_4$, $C(1)_3$, $C(7)_3$, $C(1)_2$, $C(7)_2$, $C(1)_1$, $C(1)_0$, $C(7)_1$, $C(7)_0$
8	$C(3)_6$, $C(5)_6$, $C(3)_7$, $C(5)_7$, $C(3)_5$, $C(5)_5$, $C(3)_4$, $C(5)_4$, $C(4)_7$, $C(5)_0$, $C(3)_0$, $C(4)_6$, $C(4)_5$, $C(4)_4$, $C(3)_3$, $C(5)_3$, $C(4)_3$, $C(4)_2$, $C(4)_1$, $C(4)_0$, $C(5)_1$, $C(3)_1$, $C(5)_2$, $C(3)_2$, $C(1)_7$, $C(7)_7$, $C(1)_6$, $C(7)_6$, $C(1)_5$, $C(7)_5$, $C(1)_4$, $C(7)_4$, $C(1)_3$, $C(7)_3$, $C(1)_2$, $C(7)_2$, $C(1)_1$, $C(1)_0$, $C(7)_1$, $C(7)_0$

Tabelle 3: Mit SMV Version 2.4.4 ermittelte neue Variablenordnungen

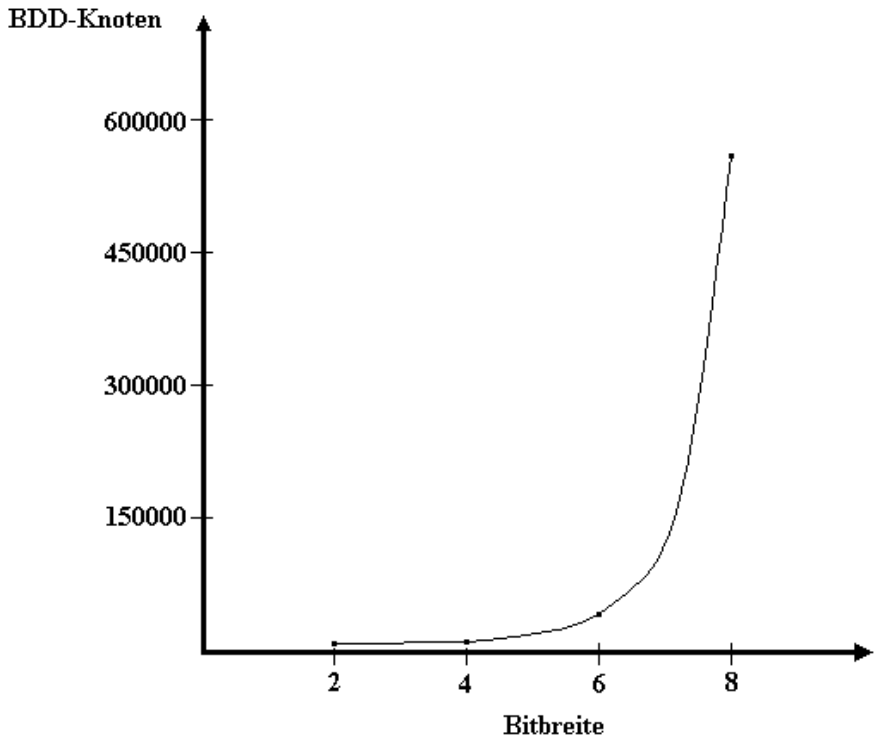


Abb.11: Benötigte BDD-Knoten für Additionstheoreme mit SMV 2.4.4

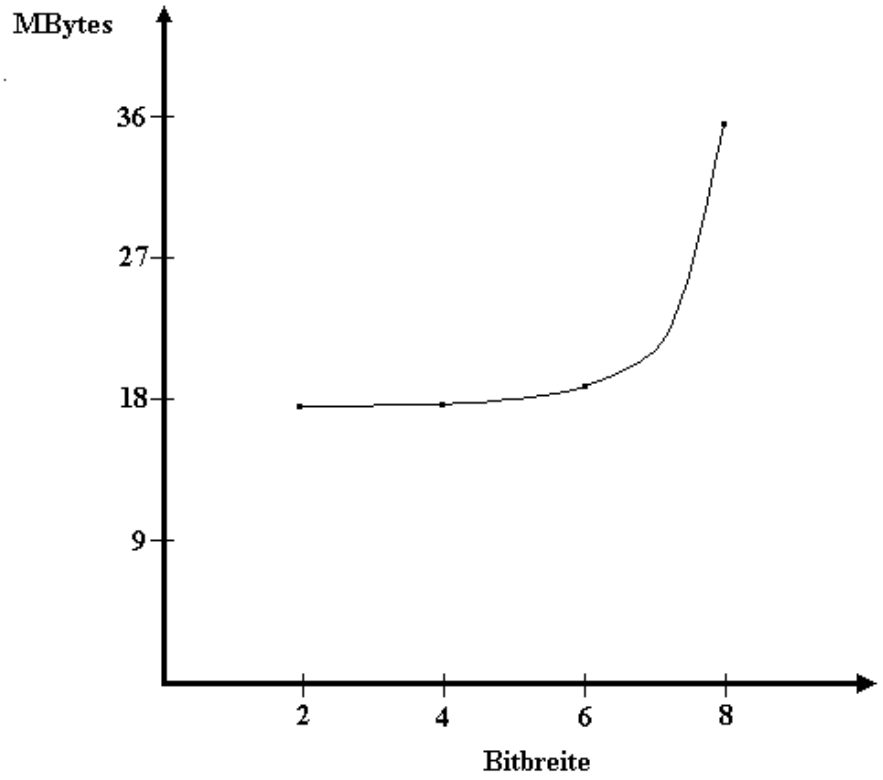


Abb.12: Benötigte MBytes für Additionstheoreme mit SMV 2.4.4

Die neuen Variablenordnungen wurden dann als Eingabe für einen erneuten Durchlauf der Additionstheoreme mit SMV Version 2.4.3 benutzt. Die Ergebnisse dieses erneuten Durchlaufs finden sich in Tabelle 4 sowie den Abbildungen 13 und 14.

Bitbreite	BDD-Knoten	Bytes	Spezifikation erfüllt ?
2	633	17.498.112	ja
4	9.531	17.825.792	ja
6	69.138	19.005.440	ja
8	608.462	28.180.480	ja

Tabelle 4: Ressourcenverbrauch für erneute SMV-Durchläufe der Additionstheoreme mit Version 2.4.3

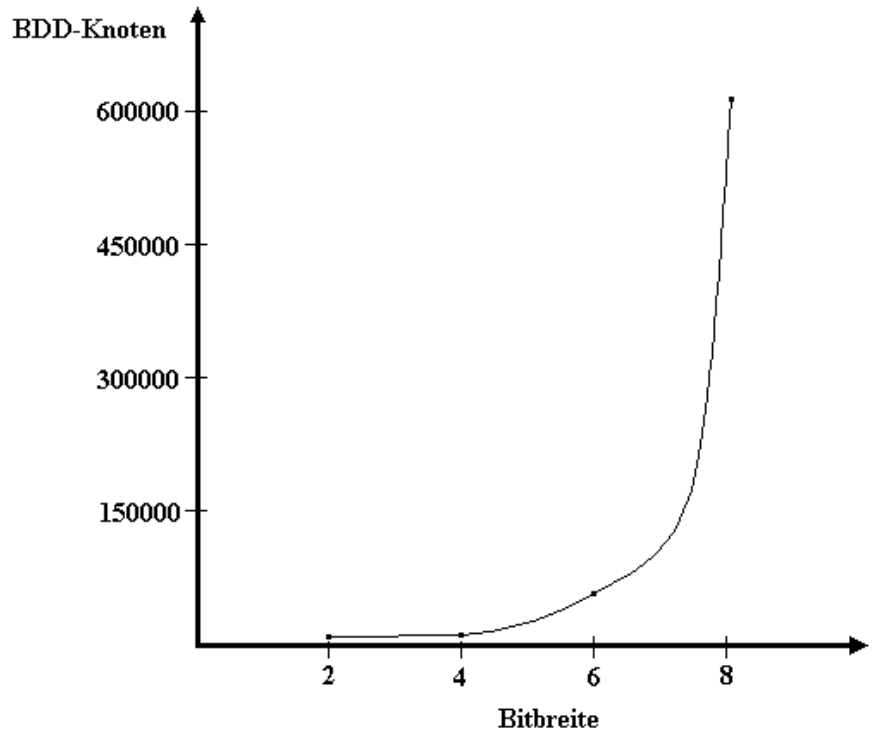


Abb. 13: Benötigte BDD-Knoten für neuen Durchlauf der Additionstheoreme mit SMV 2.4.3

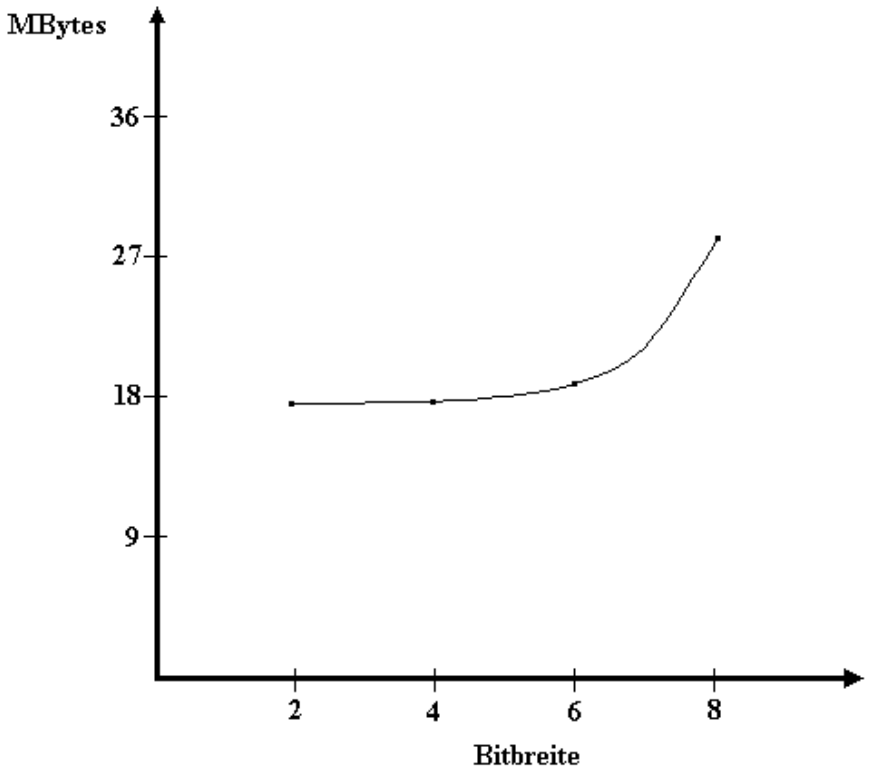


Abb. 14: Benötigte MBytes für neuen Durchlauf der Additionstheoreme mit SMV 2.4.3

Da sich die Spezifikation jedesmal als wahr herausstellte, ist es naheliegend, daß es möglicherweise gar keine von Null verschiedenen ganze Zahlen gibt, die die Additionstheoreme erfüllen. Dies muß nun bewiesen werden.

Wenn man 2C(4) als Parameter betrachtet, so ergeben die 4 Gleichungen aus Abb. 8 ein lineares Gleichungssystem. Für den Fall, daß $2C(4) = \pm 2$ ergibt sich für die anderen Zahlen C(1) = C(3) = C(5) = C(7) = 0, was nicht erlaubt ist. Im folgenden muß also $2C(4) \neq \pm 2$ gelten.

Wenn man nun die Gleichungen (2) und (3) addiert, so erhält man

$$(5) \quad 2C(4)(C(3)+C(5)) = 2C(1)$$

und aus der Addition von (1) und (4) erhält man

$$(6) \quad 2C(4)(C(1)+C(7)) = 2C(3).$$

Nun wird Gleichung (5) mit 2 multipliziert und nach 2C(3) aufgelöst:

$$2C(3) = \frac{4C(1)-4C(4)C(5)}{2C(4)}$$

Die sich ergebende Gleichung wird dann für C(3) in (6) eingesetzt und nach einigen Umformungen ergibt sich:

$$(7) \quad C(1) = \frac{4C(4)C(5)+4C(4)^2C(7)}{4-4C(4)^2}$$

Diese Gleichung läßt sich nun wiederum in (6) einsetzen und es ergibt sich:

$$(8) \quad C(3) = \frac{4C(4)^2C(5)+4C(4)C(7)}{4-4C(4)^2}$$

Die beiden Gleichungen (7) und (8) werden nun benutzt, um in den Gleichungen (1) - (4) die Werte C(1) und C(3) zu eliminieren. Es ergeben sich die folgenden Gleichungen:

$$(9) \quad (8C(4)^2-4)C(5)+(8C(4)^3-4C(4))C(7) = 0$$
$$(10) \quad (8C(4)^3-4C(4))C(5)+(8C(4)^2-4)C(7) = 0$$
$$(11) \quad (8C(4)^3-4C(4))C(5)+(8C(4)^2-4)C(7) = 0$$
$$(12) \quad (8C(4)^2-4)C(5)+(8C(4)^3-4C(4))C(7) = 0$$

Da (9) = (12) und (10) = (11) bleiben also nur zwei verschiedene Gleichungen übrig.

Wenn man nun (9) mit $8C(4)^3-4C(4)$ und (10) mit $8C(4)^2-4$ multipliziert und danach voneinander abzieht, ergibt sich folgende Gleichung:

$$(13) \quad 64C(4)^6-128C(4)^4+80C(4)^2-168C(4)^2-4 = 0$$

Es gilt weiterhin:

$$64C(4)^6 - 128C(4)^4 + 80C(4)^2 - 16 = (4C(4)^2 - 2)^2(4C(4)^2 - 4)$$

$$= (2C(4) - \sqrt{2}\,)^2(2C(4) + \sqrt{2}\,)^2(2C(4) - 2)(2C(4) + 2)$$

Nach der Voraussetzung $2C(4) \neq \pm 2$ bleiben als einzige Werte für 2C(4) noch $2C(4) = \pm\sqrt{2}$. Dies in (7) und (8) eingesetzt ergibt schließlich die beiden folgenden Lösungen:

a) $C(1) = \sqrt{2}\,C(5) + C(7)$
$C(3) = C(5) + \sqrt{2}\,C(7)$
$2C(4) = \sqrt{2}$

b) $C(1) = -\sqrt{2}\,C(5) + C(7)$
$C(3) = C(5) - \sqrt{2}\,C(7)$
$2C(4) = -\sqrt{2}$

Es existieren also unendlich viele reelle Lösungen für diese Konstanten, aber keine mit ausschließlich ganzen Zahlen. Es müssen nun also geeignete ganze Zahlen zur Approximation dieser Konstanten bestimmt werden. Weiterhin stellt sich die Frage, wie groß dann die Differenz der Ergebnisse der beiden unterschiedlichen Versionen der DCT werden kann.

6.3 Verifkation der beiden DCT-Versionen

Zur Verifikation der beiden Versionen der DCT wurden die Schaltungsbeschreibungen in einer Datei vereinigt. Eine schematische Darstellung ist in Abb. 15 zu sehen.

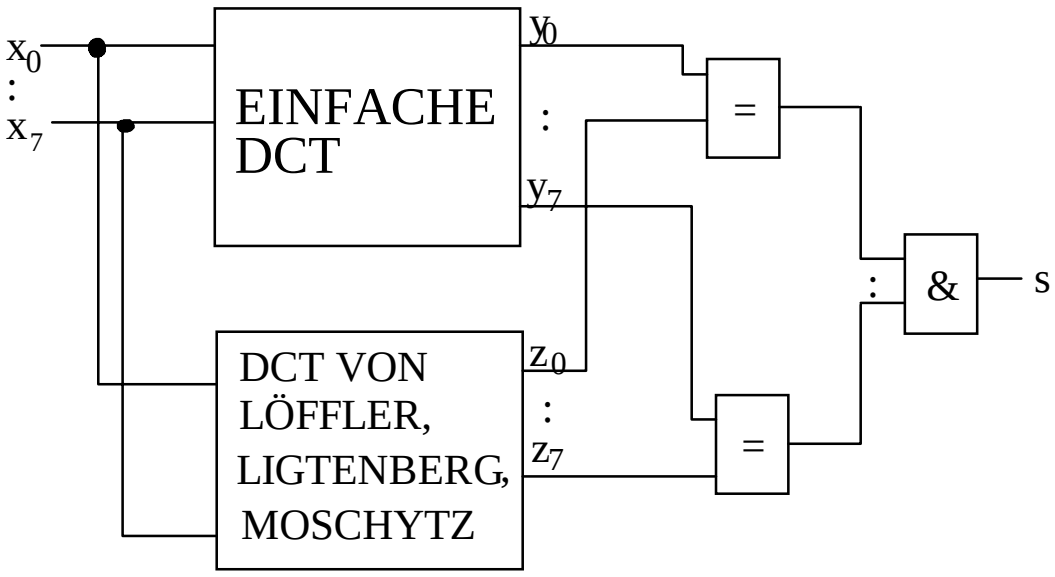


Abb. 15: Schematische Darstellung der Schaltungsbeschreibung zur Verifikation der beiden DCT-Versionen.

Dann wurde eine Spezifikation erzeugt, die besagt, daß die Ausgänge der beiden Instanzen gleich sind. Da die Bitbreiten der beiden Versionen für die jeweiligen Ausgänge unterschiedlich groß sind, wurde zum Vergleich das jeweilige Minimum der beiden Bitbreiten genommen. SMV versucht dann ein Gegenbeispiel zu finden und jedesmal, wenn dies gelingt, wird die Spezifikation entsprechend verändert. Den Ressourcenverbrauch der Verifikation für 2 Bit findet sich in Tabelle 5.

Bitbreite	BDD-Knoten	Bytes
2	31.645	21.757.952

Tabelle 5: Ressourcenverbrauch der Verifikation für 2 Bit

Für die Eingangsbitbreite von 2 Bit konnten folgende Gleichungen verifiziert werden:

$$2y_0 = C(4)z_0,$$
$$2y_2 = C(4)z_2,$$
$$2y_3 = C(4)z_3,$$
$$2y_4 = C(4)z_4,$$
$$2y_5 = C(4)z_5,$$
$$2y_6 = C(4)z_6$$

Für die Ausgangspaare $2y_1$ und $C(4)*z_1$, sowie $2y_7$ und $C(4)*z_7$ konnte keine Gleichheit gezeigt werden.

Für eine Eingangsbitbreite von 4 Bit ist der Zeitbedarf schon so groß, daß sich diese Verifikation nicht mehr durchführen läßt. Auch durch eine Änderung der Variablenordnung läßt sich der Zeitbedarf nicht signifikant senken. Daher muß ein anderer Ansatz zur Verifikation benutzt werden, bei dem man höhere Eingangsbitbreiten erreichen kann. Die Laufzeit der Verifikation hängt stark von der Anzahl der Eingangsvariablen, aber auch von der Anzahl der verwendeten Module und deren Bitbreite ab.Um die Laufzeit zu verringern müssen also die Bitbreiten sowohl an den Eingängen, als auch im Inneren der Schaltung verringert werden. Um dies zu erreichen, kann man Schaltungen mit Hilfe des Chinesischen Restsatzes modifizieren.

6.4 Reduktion mittels Chinesischem Restsatzes

Der chinesische Restsatz:

Seien m_1 , m_2 , ..., m_n positive ganze Zahlen, die paarweise teilerfremd sind.
Sei $m = m_1 \cdot m_2 \cdot ... \cdot m_n$ und seien b, i_1 , i_2 , ..., i_n ganze Zahlen, dann gibt es eine ganze Zahl i, so daß gilt:

$$b \le i \le b + m \quad \text{und } i \equiv i_j \ (mod\ m_j) \qquad für\ \ 1 \le j \le n$$

Dieser Satz läßt sich nun auf die Verifikation anwenden. Die Schaltungsbeschreibungen werden so geändert, daß die einzelnen Bausteine ihre Ergebnisse modulo m ausgeben. Ebenso werden auch die Ergebnisse an den Ausgängen der Schaltungen modulo m ausgegeben.

Zum Ändern der Schaltungsbeschreibungen werden die folgenden mathematischen Gesetze benötigt:

$$((i \bmod m) + (j \bmod m)) \bmod m \equiv (i + j) \bmod m$$
$$((i \bmod m) - (j \bmod m)) \bmod m \equiv (i - j) \bmod m$$
$$((i \bmod m) \cdot (j \bmod m)) \bmod m \equiv (i \cdot j) \bmod m$$

Schematische Darstellungen der geänderten Schaltungen finden sich in Abb. 16 und 17.

Sei n die Bitbreite an den Eingängen der Schaltungen.

Um die Verifikation mit Hilfe des Chinesischen Restsatzes durchführen zu können, benötigt man nun eine Menge $P = \{p_1, ..., p_j\}$ von paarweise teilerfremden Zahlen mit $\widetilde{P} = \prod m_k$. Es muß gelten:

$$\widetilde{P} > 2^{2n+3}$$

Wenn man die Schaltungen nun für jedes p_j modulo p_j verifizieren kann, dann folgt mit dem Chinesischen Restsatz, daß die Spezifikation auch ohne die Verwendung dieses Satzes erfüllt wäre.

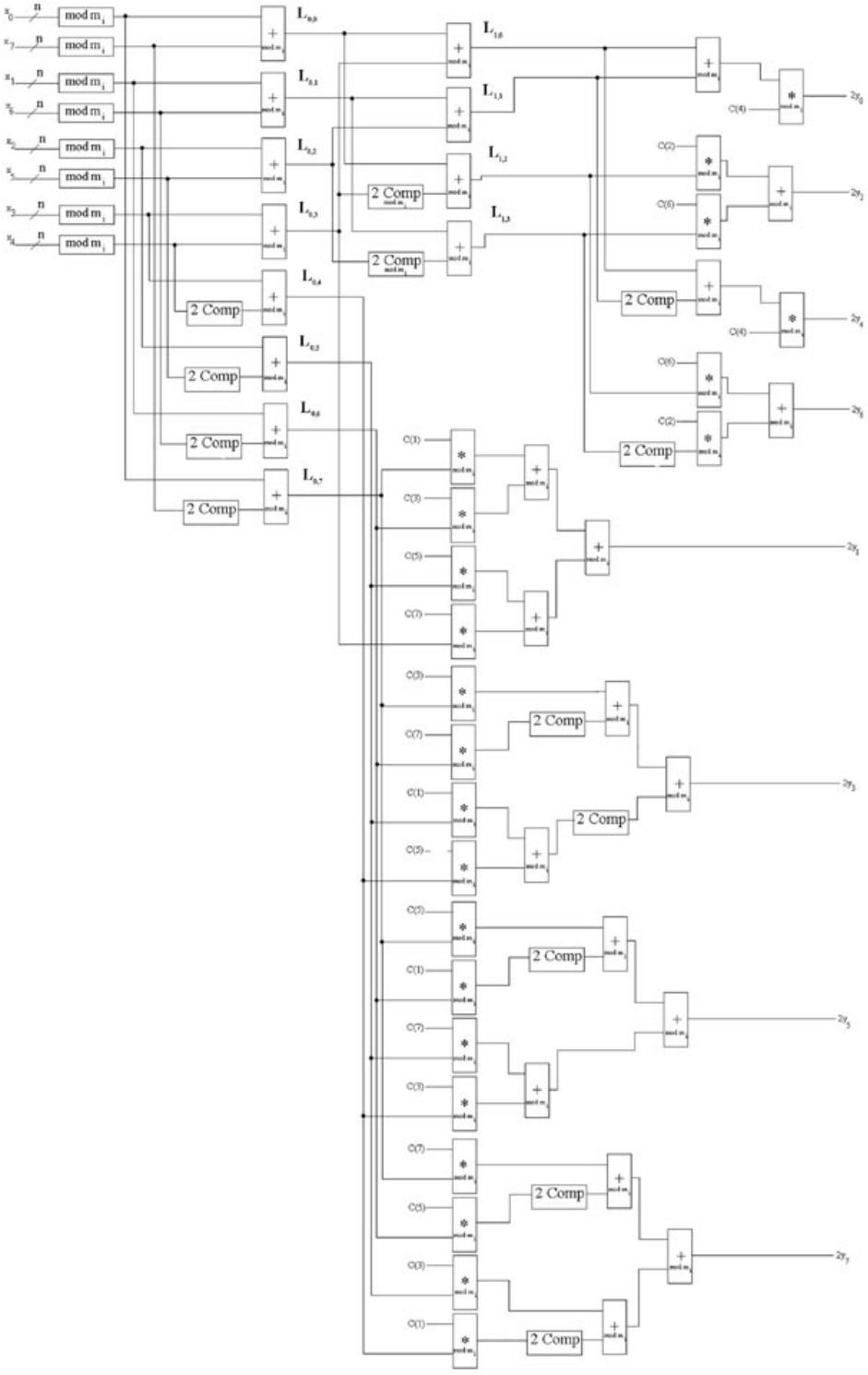


Abb. 16: Schematische Darstellung der DCT aus Abb. 3 nach Verwendung des Chinesischen Restsatzes

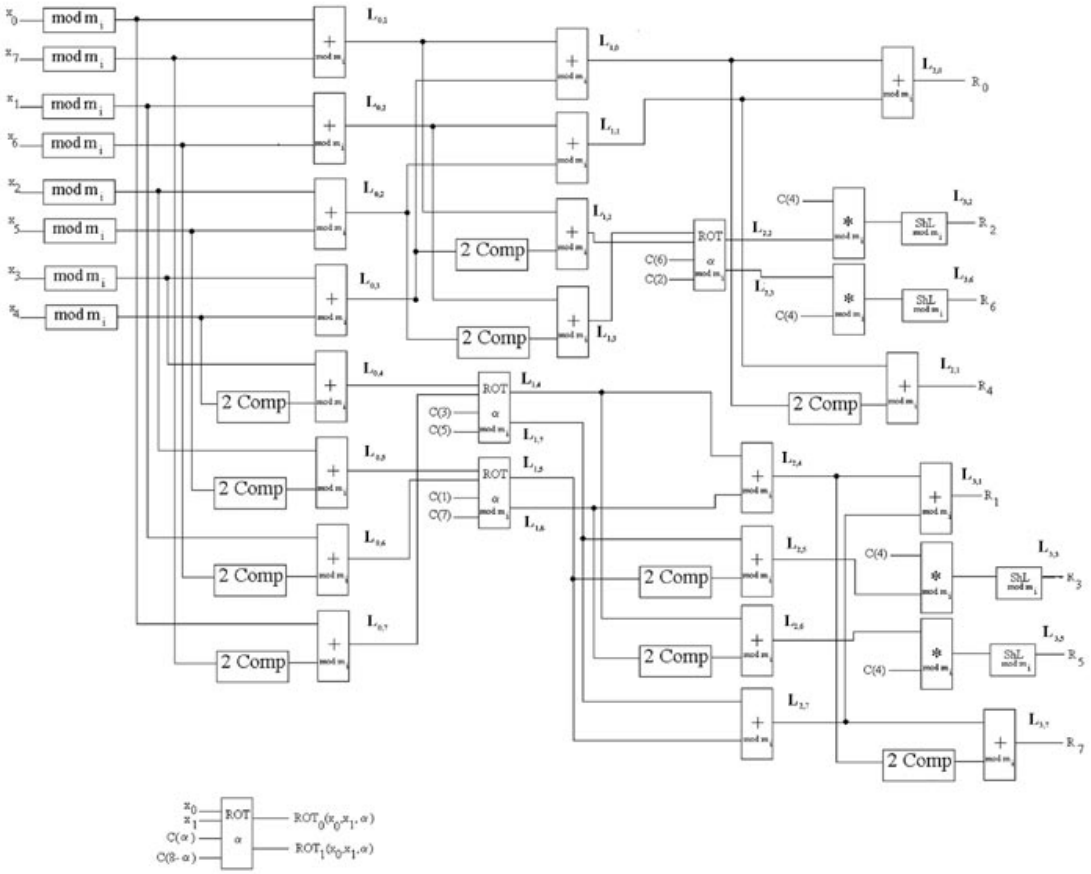
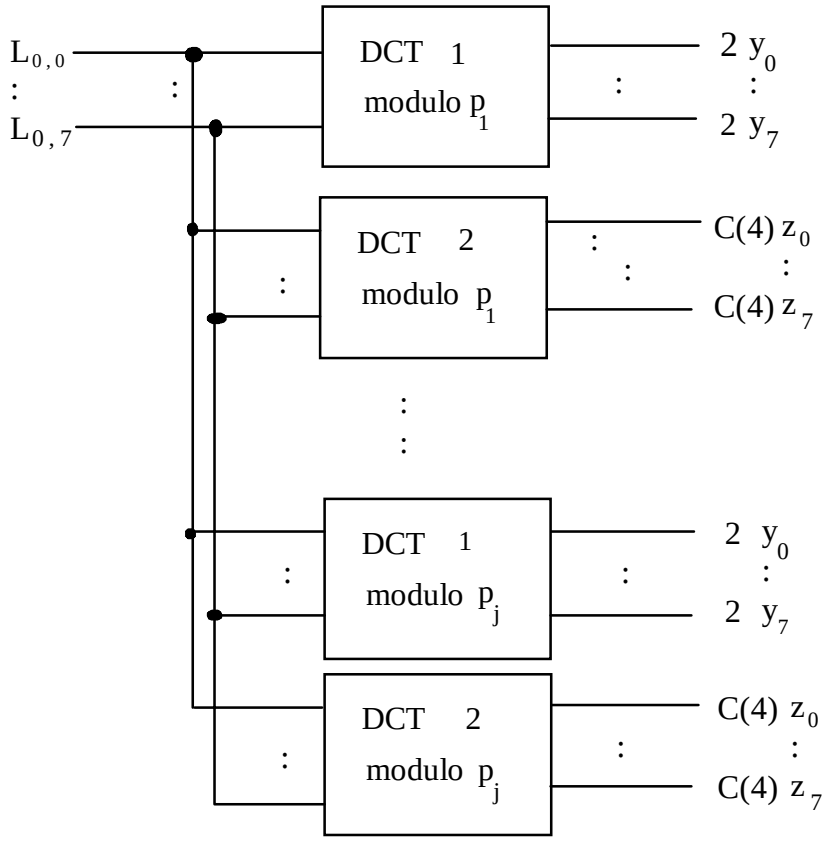


Abb. 17: Schematische Darstellung der DCT aus Abb.4 nach Verwendung des Chinesischen Restsatzes

Zur Verifikation wurde nun wieder ein C-Programm implementiert, das nach Eingabe der gewünschten Bitbreite und einer Zahlenbasis diese zuerst auf ihre Eignung für die Verifikation prüft und dann eine SMV-Beschreibung erzeugt. In dieser SMV-Beschreibung befinden sich die zu verifizierenden DCT-Versionen modulo jeder Zahl aus der eingegebenen Zahlenbasis.
In den Abbildungen 3 und 4 ist zu sehen, daß die Terme $L_{0,0}$, $L_{0,1}$, $L_{0,2}$, $L_{0,3}$, $L_{0,4}$, $L_{0,5}$, $L_{0,6}$ und $L_{0,7}$ in beiden verwendeten Versionen der DCT gleich sind. Daher kann man diese Teilterme bei der Verifikation weglassen und die jeweiligen Ergebnisse als Eingänge der jeweiligen DCT betrachten. Hierdurch wird eine Verringerung der Zahl der Eingangsvariablen von 8n auf $8 \cdot \lceil ld(p_i) \rceil$ erreicht. Eine schematische Darstellung ist in Abb. 18 zu sehen.



DCT 1 = einfache DCT aus Abb. 3

DCT 2 = DCT von Loeffler, Ligtenberg und Moschytz. (Abb. 4)

Abb. 18: Schematische Darstellung der SMV-Beschreibung zur Verifikation mit Chinesischem Restsatz.

Für die beiden verwendeten DCT-Versionen konnten folgende Gleichungen verifiziert werden.

$$2y_0 = C(4)z_0$$
$$2y_2 = C(4)z_2$$
$$2y_3 = C(4)z_3$$
$$2y_4 = C(4)z_4$$
$$2y_5 = C(4)z_5$$
$$2y_6 = C(4)z_6$$

Die Laufzeiten sowie der Ressourcenverbrauch bei den BDD-Knoten und den benötigten Bytes ist in den Tabellen 6, 7 und 8 zu sehen.

An den Stellen, an denen sich in den Tabellen ein * befindet, trat bei der Verifikation ein Fehler auf. In einem solchen Fall wurde dann die jeweilige Datei in mehrere kleine Dateien geteilt, die einzeln verifiziert wurden. Es wurde dann nicht der jeweilige Ausgang für alle Zahlen aus der Basis parallel verifiziert, sondern die Basis wurde in mehrere Teile zerlegt, für die die Verifikation nacheinander durchgeführt wurde.

Bitbreite	Ausgang 0	Ausgang 2	Ausgang 3	Ausgang 4	Ausgang 5	Ausgang 6
10	361.82	523.65	1054.25	336.83	1075.63	503.94
11	445.47	644.37	*	392.78	*	565.53
12	485.36	713.96	*	453.96	*	661.71
13	774.4	1096.51	*	720.98	*	1031.64
14	884.04	1262.2	*	830.6	*	1190.23
15	1329.17	1854.3	*	1234.17	*	1752.14
16	1496.87	2076.37	*	1407.45	*	1973.41
17	1675.19	*	*	1585.99	*	2219.68

Tabelle 6: Benötigte Laufzeit zur Verifikation der beiden DCT-Versionen in Sekunden

Bitbreite	Ausgang 0	Ausgang 2	Ausgang 3	Ausgang 4	Ausgang 5	Ausgang 6
10	510.994	545.714	267.711	443.795	268.991	409.161
11	510.925	904.699	*	443.678	*	693.629
12	523.921	874.123	*	448.313	*	672.379
13	877.062	1.085.330	*	663.397	*	832.091
14	767.123	1.174.143	*	729.415	*	935.153
15	1.082.133	1.424.903	*	876.262	*	1.155.686
16	1.107.811	1.396.318	*	888.366	*	1.089.533
17	1.138.045	*	*	913.521	*	1.196.082

Tabelle 7: Benötigte BDD-Knoten zur Verifikation der beiden DCT-Versionen

Bitbreite	Ausgang 0	Ausgang 2	Ausgang 3	Ausgang 4	Ausgang 5	Ausgang 6
10	26.345.472	29.032.448	30.801.920	24.838.144	30.801.920	26.476.544
11	28.114.944	36.700.160	*	26.738.688	*	32.899.072
12	30.146.560	38.141.952	*	28.573.696	*	34.471.936
13	41.156.608	47.185.920	*	37.158.912	*	42.729.472
14	41.484.288	50.790.400	*	40.370.176	*	46.530.560
15	52.625.408	61.407.232	*	48.693.248	*	56.623.104
16	55.377.920	63.438.848	*	51.314.688	*	58.064.896
17	58.195.968	*	*	54.132.736	*	62.259.200

Tabelle 8: Benötigte Bytes zur Verifikation der beiden DCT-Versionen

Tabelle 9 enthält die Ergebnisse der einzelnen SMV-Beschreibungen für die Verifikation von Ausgang 3 und Tabelle 10 enthält die Ergebnisse der einzelnen SMV-Beschreibungen für die Verifikation von Ausgang 5. In Tabelle 11 befinden sich die Ergebnisse der einzelnen SMV-Beschreibungen für die Verifikation von Ausgang 2 für 17 Bit.

Bitbreite	Basis	User-Time in Sekunden	BDD-Knoten	Bytes
11	7, 11, 13, 17	307.69	127.083	17.891.328
	19, 20, 23	249.65	70.469	15.400.960
12	7, 11, 13, 17	354.14	48.375	17.694.720
	19, 20, 23	287.93	231.421	18.939.904
13	7, 11, 13, 17	404.76	56.261	18.939.904
	19, 23, 29, 30	604.54	446.275	28.246.016
14	7, 11, 13, 17	458.64	71.870	20.316.160
	19, 23, 29, 30	693.44	438.261	29.425.664
15	7, 11, 13, 17, 19	900.43	183.837	28.770.304
	23, 29, 30, 31	778.57	552.533	32.571.392
16	7, 11, 13	268.95	62.436	16.711.680
	17, 19	200.34	175.698	16.252.928
	23, 29, 30, 31	859.58	320.712	30.277.632
17	7, 11, 13	300.89	60.345	17.367.040
	17, 19	221.69	179.935	16.973.824
	23, 29, 30, 31	936.36	178.659	29.229.056

Tabelle 9.: Ressourcenverbrauch bei der Verifikation von Ausgang 3

Bitbreite	Basis	User-Time in Sekunden	BDD-Knoten	Bytes
11	7, 11, 13, 17	309.06	122.024	17.760.256
	19, 20, 23	249.58	70.064	15.400.960
12	7, 11, 13, 17	355.18	44.579	17.629.184
	19, 20, 23	289.66	226.718	18.874.368
13	7, 11, 13, 17	405	53.400	18.939.904
	19, 23, 29, 30	604.03	449.504	28.311.552
14	7, 11, 13, 17	472.31	77.368	20.381.696
	19, 23, 29, 30	700.6	431.693	29.360.128
15	7, 11, 13, 17, 19	899.95	181.177	28.770.304
	23, 29, 30, 31	782.65	538.492	32.374.784
16	7, 11, 13	268.49	59.815	16.646.144
	17, 19	200.72	168.381	16.187.392
	23, 29, 30, 31	860.31	304.802	30.015.488
17	7, 11, 13	300.84	59.952	17.367.040
	17, 19	221.43	169.077	16.842.752
	23, 29, 30, 31	929.5	177.495	292.290.656

Tabelle 10.: Ressourcenverbrauch bei der Verifikation von Ausgang 3

Bitbreite	Basis	User-Time in Sekunden	BDD-Knoten	Bytes
17	7, 11, 13, 17, 19	597.91	595.481	33.488.896
	23, 29, 30, 31	504.89	743.418	33.488.896

Tabelle 11.: Ressourcenverbrauch bei der Verifikation von Ausgang 2 für 17 Bit

Für die Ausgänge $2y_1$ und $C(4)z_1$ sowie $2y_7$ und $C(4)z_7$ konnte die Gleichheit nicht gezeigt werden, da hier Rechenungenauigkeiten durch Approximation der Cosinus-Werte auftraten. Dies kommt daher, daß innerhalb dieser Gleichungen zweimal mit einer Cosinus-Konstanten multipliziert wird. Dadurch kommen die Rundungsfehler, die durch das Reduzieren der Konstanten auf n Bit entstehen, stärker zum Tragen.

Um mit Hilfe der Modellprüfung eine Fehlerschranke für die Differenz des jeweiligen Ausgangs bei beiden DCT-Versionen zu ermitteln wurde der folgende Ansatz versucht:

Von dem jeweiligen Ausgang der beiden DCT-Versionen werden jedesmal die beiden Differenzen DCT1-DCT2 und DCT2-DCT1 gebildet. Im folgenden sei

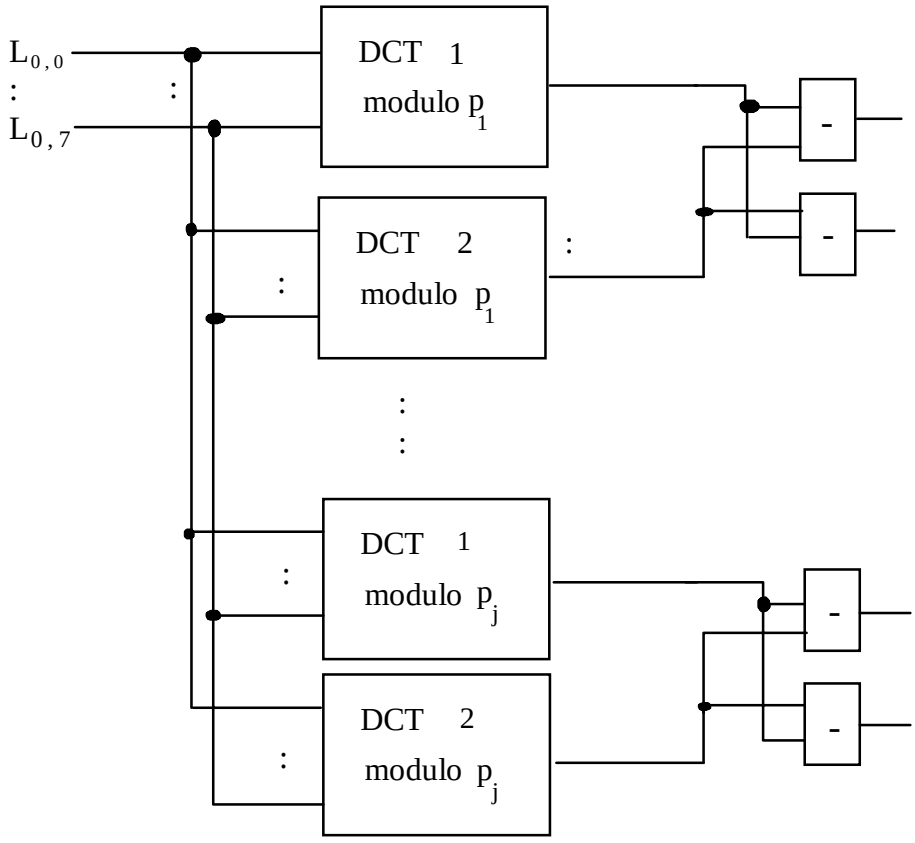
$$\delta(p_i) = (DCT1) mod\ p_i - (DCT2) mod\ p_i, \text{ und}$$
$$\varepsilon(p_i) = (DCT2) mod\ p_i - (DCT1) mod\ p_i$$

die Differenz der beiden DCT-Versionen.

Wenn man nun zeigen kann, daß für die Residuen-Basis $\{p_1, \ldots, p_j\}$ gilt:

$$((\delta(p_1) = \delta(p_2)) \wedge (\delta(p_2) = \delta(p_3)) \wedge \ldots \wedge (\delta(p_{j-1}) = \delta(p_j))) \vee$$
$$((\varepsilon(p_1) = \varepsilon(p_2)) \wedge (\varepsilon(p_2) = \varepsilon(p_3)) \wedge \ldots \wedge (\varepsilon(p_{j-1}) = \varepsilon(p_j))),$$

dann ist der Betrag der Differenz der beiden Ausgänge kleiner als die kleinste Zahl der eingegebenen Basis. Eine schematische Darstellung zeigt Abb. 19.



DCT 1 = einfache DCT aus Abb. 3

DCT 2 = DCT von Loeffler, Ligtenberg und Moschytz. (Abb. 4)

Abb. 19: Schematische Darstellung der SMV-Beschreibung zur Ermittlung der oberen Schranke der Differenz für die Ausgänge 1 und 7

Diese Spezifikation konnte jedoch nicht bewiesen werden, da der Fehler mit der Eingangsbitbreite stark anwächst. Die Schaltungen verwenden nämlich eine Festpunktdarstellung, welche auf der folgenden Konvertierung von $\Re$ beruht: $f : \Re \to Z$ mit $f(x) := \lfloor x \cdot 2^n \rfloor$. Multiplikation mit einer Cosinus-Konstanten führt in dieser Zahldarstellung somit zu einer impliziten Multiplikation mit 2^n , so daß sich vorhandene Fehler ebenfalls um den Faktor 2^n vergrößern. Die Cosinus-Konstanten entstanden, indem der echte Cosinus mit 2^n multipliziert und dann nach n Bit abgeschnitten wurde.

Die Ergebnisse an den Ausgängen 1 und 7 der beiden DCT-Versionen sind also nicht vergleichbar, solange man nur die Ergebnisse modulo eines p_i hat. Es ist jedoch möglich, die Ergebnisse aus der Darstellung modulo der p_i wieder in die normale Notation zurückzuwandeln. In der normalen Notation ohne die Verwendung des Chinesischen Restsatzes ließen sich die Ergebnisse an den Ausgängen 1 und 7 nämlich vergleichen. Das Zurückwandeln der Werte läßt sich entweder parallel oder sequentiell bewerkstelligen. Die parallele Variante hat den Nachteil, daß wieder eine große Zahl von Multiplizierern benötigt wird, was ja eigentlich vermieden werden sollte, und man dadurch einen großen Teil der erzielten Ressourcenersparnis wieder aufgibt. Da BDD's eine Normalform darstellen, wird hier als Resultat wieder dasselbe BDD enstehen, welches man ohne Verwendung des Chinesischen Restsatzes erhalten hätte. Die sequentielle Variante hat wiederum den Nachteil, daß in diesem Fall zusätzlich ein Controller und mehrere Register benötigt werden, was den Aufwand bei der Verifikation auch beträchtlich steigert. Eine experimentelle Analyse wurde hierzu jedoch nicht mehr durchgeführt.

Es ist jedoch anzunehmen, daß das Zurückwandeln der Ergebnisse in die normale Notation ist also nicht mit vertretbarem Aufwand zu machen ist. Daher läßt sich also über die Ergebnisse der Ausgänge 1 und 7 der beiden DCT-Versionen lediglich sagen, daß sie nicht gleich sind. Die Existenz einer oberen Schranke konnte mit Hilfe des SMV jedoch nicht gezeigt werden.

7. Schlußfolgerungen

Es ließ sich zeigen, daß Modellprüfung auch auf die Schaltungsklasse der diskreten Cosinus-Transformation anwendbar ist. Es ist jedoch nicht ohne weiteres möglich größere Eingangsbitbreiten zu verifizieren, da schon bei einer Eingangsbitbreite von 4 Bit die Verifikation nicht mehr mit vernünftigem Zeitbedarf durchzuführen ist. Es ist notwendig Abstraktionsverfahren wie den Chinesischen Restsatz anzuwenden, um auch größere Eingangsbitbreiten verifizieren zu können.

Nach der Anwendung des Chinesischen Restsatzes kann eine Rückkonvertierung in die normale Notation durchgeführt werden, um auch Fehlerschranken für die Ausgänge 1 und 7 verifizieren zu können. Da Modellprüfer eine Normalform herstellen bringt eine parallele Rückkonvertierung keinen Gewinn, da das resultierende BDD dasselbe ist, das man ohne Verwendung des Chinesischen Restsatzes erhalten würde. Es ist also erforderlich eine sequentielle Rückkonvertierung durchzuführen.

8. Literaturverzeichnis

[1] SHANNON, C. E. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal, 1948.*

[2] BRYANT, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers, C-35(8):677-691, August 1986.*

[3] PENNEBAKER,W., AND MITCHELL, J. *JPEG Still Image Data Compression Standard*. Van Nostrand Reinhold, ISBN 0-442-01272-1, 1993.

[4] LOEFFLER, C., LIGHTENBERG, A. AND MOSCHYTZ, G. Practical fast 1-D DCT algorithms with 11 multiplications. In *International Conference on Acoustics, Speech, and Signal Processing 1989* (ICASSP '89) (1989), pp. 988-999.

[5] KROPF, T. Hardware Verifikation / Entwurf und Werkzeuge zum Entwurf korrekter Schaltungen und Systeme. Skriptum zur Vorlesung WS 1997/98. Universität Karlsruhe, Institut für Rechnerentwurf und Fehlertoleranz.

[6] GUPTA, A. Formal Hardware Verification Methods: A Survey. In *Journal of Formal Methods in System Design, Volume 1, 1992*, pp. 151-238.

[7] YOELI, M. Formal Verification of Hardware Design. *IEEE Computer Society Press 1990.*

[8] CLARKE, E. M., EMERSON, E. A. AND SISTLA, A. P. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. In *ACM Transactions on Programming Languages and Systems, Volume 8, April 1986 Number 2*, pp. 244-263.

[9] EMERSON, E. A. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science, Volume B,* pp 996-1072. Elsevier Science Publishers, 1990.

[10] CLARKE, E. M. AND EMERSON, E. A. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Workshop on Logics of Programs - Lecture Notes in Computer Science, Volume 131*, pp. 52-71. Springer Verlag, 1981.

[11] MCMILLAN, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[12] GAREY, M. R. AND JOHNSON, D. S. Computers and Intractability: A Guide to the Theory of NP-Completeness. In *Studienreihe Informatik*. W. H. Freeman and Company, 1979.