Effizienter Entwurfsfluss durch neue Verfahren der Logiksynthese und Technologieabbildung von VHDL-Hardwarebeschreibungen

vorgelegt von
Diplom-Ingenieur
Jan Gutsche

Von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. rer. nat. Otto Manck
Berichter:	Prof. Dr.-Ing. Hans-Ulrich Post
	Prof. Dr.-Ing. Hans Liebig

Tag der wissenschaftlichen Aussprache: 20. Mai 2005

Berlin 2005
D83

Danksagung

Die vorliegende Arbeit entstand im Rahmen meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Technische Informatik und Mikroelektronik der Technischen Universität Berlin.

Ich danke allen, die in verschiedener Weise zum Gelingen dieser Arbeit beigetragen haben. Herrn Prof. Dr.-Ing. Hans-Ulrich Post für die gegebenen Freiräume und Tipps, Herrn Prof. Dr.-Ing. Hans Liebig für die sofortige Bereitschaft, das zweite Gutachten zu übernehmen, für das entgegengebrachte Interesse an der Arbeit und die wertvollen Gespräche ein herzliches Dankeschön.

Weiterhin danke ich Nico Moser, meinem Lieblings-Ex-Tutor, der mich auch als gleichgestellter Kollege bei quengelnden Fragen nicht vor die Tür gesetzt hat, Till Neunast und Prof. Dr. Matthias Menge, die Kollegen von der Konkurrenz, die sich für jedes meiner Problemchen spontan haben begeistern lassen, Silvia Rabe, Theresa Schadow, Torsten Sadowski und Konstantin Gründger die mir selbstlos beim Feinschliff geholfen haben, einem gewissen „Kindergarten“ für den nötigen Abstand, wenn es mal wieder klemmte und ich keine Sonne mehr gesehen habe. Last but not least danke ich meiner Familie für die Geduld, wenn ich doch nicht so viel Zeit hatte, wie ich mir immer vorgenommen habe.

Berlin 23. Mai 2005

Zusammenfassung

Neben der tatsächlichen Leistungsfähigkeit von Synthesewerkzeugen hinsichtlich Synthesegeschwindigkeit und Güte der produzierten Ergebnisse ist auch die durch den Benutzer notwendige Interaktion zur Erzeugung eines befriedigenden Produktes ein wichtiger Faktor bei der Frage, wie schnell eine vorgegebene Aufgabe umgesetzt werden kann (Time to Market).

Zwei grundlegende Designschwächen bestehender Synthesewerkzeuge werden untersucht, welche eine permanente Aufmerksamkeit seitens des Entwicklers erfordern.

Die erste Designschwäche betrifft die automatische Logiksynthese. Sie kann zu einem unterschiedlichen Verhalten der generierten Schaltung im Vergleich zu einer vorher durchgeführten Simulation führen. Diese Synthesefehler sind oft nur schwer zu erkennen. Die Vermeidung solcher Fehler seitens des Entwicklers ist nur bei der strengen Einhaltung eines gegebenen Coding-Styles möglich. Dieser wird von den meisten Werkzeugen nur oberflächlich geprüft und verkompliziert zusätzlich die Beschreibung bestimmter Baugruppen erheblich. Es werden die Ursachen für dieses Verhalten beschrieben und ein alternatives vom Coding-Style unabhängiges Logiksyntheseverfahren „SibaS“ (Simulation-based-Synthesis) vorgestellt. Dieses Verfahren verwendet, insbesondere bei der Synthese von sequentiellen Schaltungen, einen anderen Ansatz als aktuelle VHDL-Syntheseverfahren. Im Gegensatz zu diesen wird dabei nicht versucht, aus der Struktur der beschriebenen Schaltung auf den Schaltungstyp zu schließen, stattdessen wird die zu übersetzende Beschreibung einer speziellen Simulation unterworfen und anhand der Ergebnisse eine entsprechende Schaltung generiert.

Die zweite Designschwäche betrifft die Schnittstelle zwischen Logiksynthese und Technologieabbildung. Die Verwendung von VHDL an dieser Stelle erlaubt zwar die beliebige Kombination von Werkzeugen, führt aber zum Verlust abstrakter Designinformation, wie z. B. von arithmetischen Strukturen, was durch den Einsatz zusätzlicher Werkzeuge und damit verbundener Entwicklungszeit ausgeglichen werden muss. Die Integration von Logiksynthese und Technologieabbildung ermöglicht es, eine neue Schnittstelle „A-RTL“ (Arithmetic-RTL) zu schaffen, um diese abstrakten Informationen zu bewahren und innerhalb der Technologieabbildung „ALTeM“ (Arithmetic and Logic Technology Mapping) zu nutzen.

Die entwickelten Verfahren wurden innerhalb einer beispielhaften Implementierung eines kompletten Synthesewerkzeugs „Square-Dance“ integriert und untersucht. Im Vergleich zu den verfügbaren VHDL-Synthesewerkzeugen zeichnet sich Square-Dance durch eine fehlerfreie Synthese, unabhängig vom verwendeten Coding-Style, und die Möglichkeit der unmittelbaren Generierung effizienter arithmetischer Strukturen aus.

Inhaltsverzeichnis

Symbol- und Abkürzungsverzeichnis

A-RTL	Arithmetic-RTL
ALU	Arithmetic and Logic Unit
ALTeM	Arithmetic and Logic Technology Mapping
ANSI	American National Standards Institute
ASIC	Application Specific Integrated Circuit
BDD	Binary Decision Diagram
BDT	Binary Decision Trees
BLIF	Berkeley Logic Interchange Format
CAD	Computer Aided Design
D-FF	Delay-Flipflop
DE-FF	Dual-Edge-Flipflop
DR-FF	Dual-Rail-Flipflop
DNF	Disjunktive Normalform
EMV	Elektromagnetische Verträglichkeit
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FR-FF	Flanken-Reset-Flipflop
IEEE	Institute of Electrical and Electronic Engineers
IP	Intellectual Property
KISS	State Assignment Program for PLA-based Finite-State Machines
KNF	Konjunktive Normalform
NP	Nichtdeterministisch Polynomiell
OBDD	Ordered Binary Decision Diagram
RAM	Random Access Memory
RTL	Register Transfer Level
SLIF	Stanford Logic Interchange Format
SibaS	Simulation-based-Synthesis
RS-FF	Reset-Set-Flipflop
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WORM	Write Once Read Many

1 Einleitung

1.1 Motivation

Der Entwurf digitaler Schaltungen ist durch eine hohe, ständig zunehmende Komplexität der zu bewältigenden Aufgaben gekennzeichnet. Dies ist bedingt durch die Forderung, immer leistungsfähigere Schaltungen mit gleicher oder geringerer Entwicklungszeit zu entwerfen. Diese Aufgabe gelingt nur, wenn die Entwicklung der entsprechenden Entwurfswerkzeuge mit der Technologieentwicklung Schritt hält.

Die Leistungsfähigkeit von Synthesewerkzeugen lässt sich unter anderem auf folgende Faktoren zurückführen:

1. Bewältigung der Datenmenge bzw. Rechenzeit
   Bei der Bewältigung der zu verarbeitenden Datenmenge ist die verwendete Hardware (Speicher, Rechengeschwindigkeit) nur ein Faktor bei der Frage, wie schnell ein gegebenes Design synthetisiert werden kann. Zusätzlich spielen auch die im Synthesewerkzeug realisierten Datenstrukturen und Algorithmen eine Rolle. Die reine Rechenzeit zum Bewältigen vieler Teilschritte während des Syntheseprozesses hängt in der Regel nicht linear von der vorgegebenen Designgröße ab. Hier sind ständig neue Verwaltungskonzepte und Algorithmen gefragt, da in der Regel davon auszugehen ist, dass die Leistungsfähigkeit der verwendeten Hardware nur linear zur Komplexität des zu bearbeitenden Designs ist.

2. Interaktionsmöglichkeiten und -notwendigkeiten
   Interaktionsmöglichkeiten sind Schnittstellen, die es dem Entwickler erlauben, zusätzlich Einfluss auf den Syntheseprozess zu nehmen, indem vom Standardprozess abweichende Randbedingungen an unterschiedlichen Stellen eingebracht werden können. Dies kann etwa die Angabe von anderen Geschwindigkeitsvorgaben in einem einzelnen Untermodul im Vergleich zu den globalen Vorgaben sein. Interaktionsnotwendigkeiten sind zwingend notwendige Schritte, ohne die der Syntheseprozess nicht durchgeführt werden kann bzw. die Ergebnisse nicht zu verwerten sind. Dies sind beispielsweise die Angaben der Ausgangsbeschreibungen und der Zieltechnologie. Sowohl die Reduzierung der Interaktionsnotwendigkeiten als auch die Schaffung zusätzlicher Interaktionsmöglichkeiten können den Entwurfsprozess optimieren.

3. Qualität der Ausgangsdaten
   Unter der Qualität der Ausgangsdaten ist der Grad der Brauchbarkeit der von der Synthese gelieferten Ergebnisse zu verstehen. Ein Aspekt ist die Frage, wie effizient ein vorgegebenes Design unter Berücksichtigung vorgegebener Randbedingungen nach der Synthese realisiert ist. Ein weiterer Punkt betrifft die grundlegende Funktionalität, die aufgrund zum gegenwärtigen Zeitpunkt immer noch auftretender Synthesefehler nicht immer gegeben ist und sich unmittelbar auf die Entwicklungszeit auswirkt.

4. Qualität der Eingangsdaten
   Die Qualität der Eingangsdaten bezieht sich hauptsächlich auf die Form der vom Synthesewerkzeug verarbeitbaren Eingangsdaten. Die Möglichkeit, zunehmend abstraktere Beschreibungen verarbeiten zu können (sei es direkt im Synthesewerkzeug oder durch neue vorgeschaltete Werkzeuge wie High Level Compiler), entbindet den Entwickler von dem Durchführen entsprechender Transformationen.

Eine Verkürzung der Entwurfszeit lässt sich durch das Einwirken auf einen oder mehrere Punkte erzielen. Der Einsatz von leistungsfähigerer Hardware und zusätzlicher Spezialsoftware (Punkt 1 und 3) kann nur kurzfristig eine Lösung darstellen. Langfristig ist die permanente Überarbeitung und Anpassung der Werkzeuge an neue Anforderungen (wie beispielsweise umfangreichere Schaltungen, neue Fertigungs- und Schaltungstechnologien) notwendig, um mit der Entwicklung Schritt halten zu können.

1.2 Eigener Beitrag

Die neuen Verfahren zur Reduzierung der Entwurfszeit wurden in dem Entwurfswerkzeug Square-Dance für die Logiksynthese und Technologieabbildung integriert. Der Aufbau ist stark an konventionelle Werkzeuge angelehnt, so dass das Werkzeug ohne Modifikationen in bestehende Designprozesse einzugliedern ist. Square-Dance zeichnet sich, bedingt durch das entwickelte Verfahren SibaS (siehe „Logiksyntheseverfahren SibaS“ auf Seite 33), durch eine vergleichsweise genaue Synthese aus, welche die Zeit der Suche nach Synthesefehlern reduziert. Zusätzlich wurde durch den Einsatz von SibaS der synthetisierbare Sprachumfang über den sog. Coding-Style (siehe „Logiksynthese aus VHDL-Beschreibungen“ auf Seite 12) hinaus erweitert, wodurch die Anzahl der durchzuführenden Anpassungen des Hardwaredesigns an das Synthesewerkzeug verringert wird. So werden im Gegensatz zu verfügbaren Werkzeugen den Coding-Style verletzende Beschreibungen mit Square-Dance entweder korrekt übersetzt oder mit einer aussagekräftigen Meldung abgelehnt.

Im Unterschied zu vorhandenen Werkzeugen wurden Logiksynthese und Technologieabbildung in einem Werkzeug zusammengeführt. Die Verschmelzung der normalerweise voneinander getrennten Werkzeuge erlaubt es, abhängig von der gestellten Aufgabe, auf unterschiedlichen Abstraktionsebenen zu arbeiten. So wird im Vergleich zu anderen Werkzeugen beispielsweise die Synthese von arithmetischen Strukturen der Technologieabbildung ALTeM (siehe „Technologieabbildung ALTeM“ auf Seite 59) überlassen, da diese aufgrund der dort verfügbaren Technologiedaten dann bessere Strukturen erzeugen kann, als es die Logiksynthese mit einer generischen Architektur vermag. Es ist so innerhalb der allgemeinen Synthese beispielsweise möglich, effizientere arithmetische Strukturen als mit vergleichbaren Werkzeugen zu erzeugen, weshalb an diesem Punkt auf den Einsatz von zusätzlichen Werkzeugen zum Erzeugen spezieller Strukturen (Coregeneratoren) verzichtet werden kann. Jedes Verfahren für sich kann je nach Aufgabe die durch den Entwickler durchzuführenden Schritte deutlich reduzieren.

1.3 Übersicht

Das folgende Kapitel enthält einen Überblick über den Stand der Technik. In Kapitel 3 werden die im weiteren Verlauf benötigten grundlegenden Verfahren der Verarbeitung boolescher Funktionen behandelt. Die Kapitel 4 bis 6 behandeln im Einzelnen die neuen Syntheseverfahren, welche im Rahmen des Synthesewerkzeugs Square-Dance implementiert wurden. Der Aufbau von Square-Dance ist in Abbildung 1.1 gegeben. Square-Dance besteht aus zwei Programmen. Der library_compiler wandelt eine gegebene VHDL-Technologiebeschreibung in eine für den design_compiler verwertbare Beschreibung um. Die entsprechenden Verfahren sind in Kapitel 3.2.3 bzw. Kapitel 7.1 beschrieben. Aufgabe des design_compilers ist die Übertragung einer in VHDL gegebenen RTL-Beschreibung in eine technologiespezifische Netzliste. Der design_compiler setzt sich zusammen aus der Logiksynthese SibaS (Kapitel 4), der Technologieabbildung ALTeM (Kapitel 5) und einer speziellen Datenbank A-RTL (Kapitel 6), welche die Schnittstelle zwischen Logiksynthese, Technologieabbildung und dem library_compiler darstellt. Kapitel 7 behandelt die Funktionalität und Bedienung aus der Sicht des Benutzers und bewertet die implementierten Verfahren. Die Arbeit schließt mit einem Ausblick auf weitergehende Entwicklungen der vorgestellten Verfahren und der Implementierungen ab.

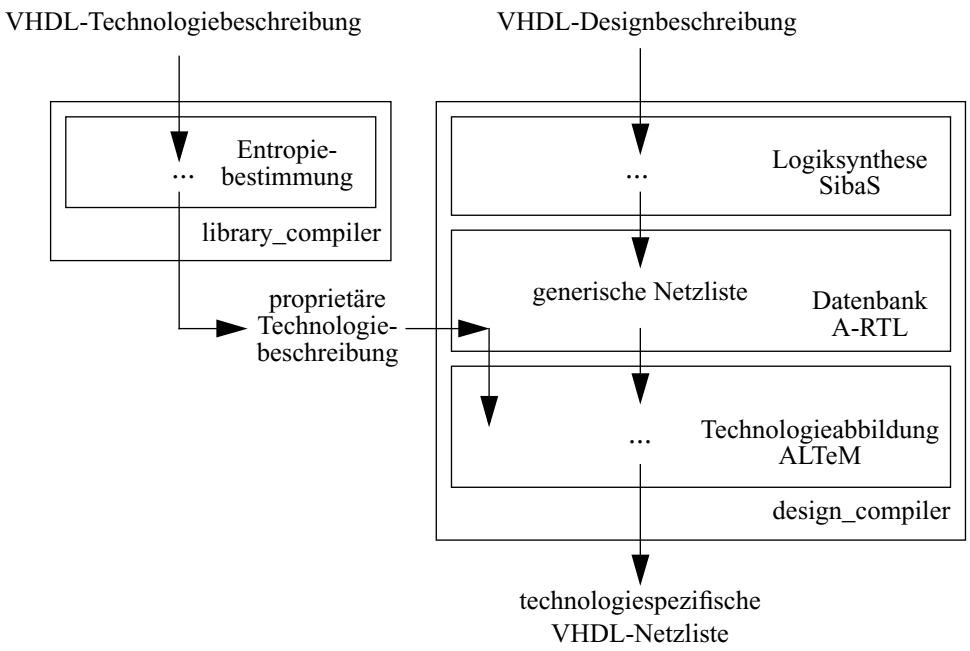


Abbildung 1.1 *Aufbau des Square-Dance-Synthesewerkzeugs.*

2 Stand der Technik

2.1 Entwurf digitaler Schaltungen

2.1.1 Hierarchisches Modell und zugehöriger Synthesefluss

Es haben sich unterschiedliche Hierarchiemodelle [53, 56] bewährt, um die verschiedenen Randparameter eines Projektes geordnet und kontextbezogen bearbeiten zu können. Je nach bevorzugtem Vorgehen beim Entwurf fließen die zu erfüllenden Randbedingungen auf den unterschiedlichen Abstraktionsebenen in das Design ein. Die Komplettierung des Designs auf einer Schicht erzeugt zusätzlich neue Randbedingungen, die es auf der nächsten Schicht zu berücksichtigen gilt.

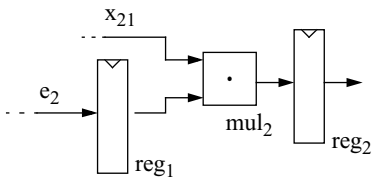
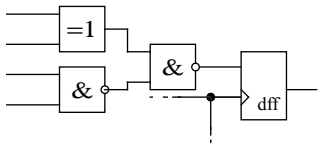
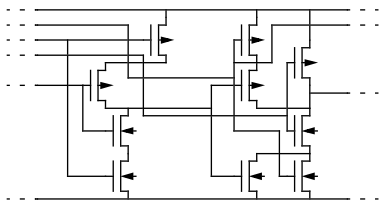
Es existieren unterschiedliche Modelle, um den Entwicklungsprozess zu veranschaulichen. Je nach Bedarf können einzelne Schichten des Modells feiner untergliedert werden, um auf spezielle Probleme aufmerksam zu machen, oder es wird eine grobe Gliederung vorgenommen, um den Prozess als Ganzes einfacher erfassen zu können.

Im Kontext des Hardwaredesigns versteht man unter Synthese den Übergang von der formalen Beschreibung eines Verhaltens zu einer dieses Verhalten realisierenden Struktur [39]. Dieses Verständnis hat sich bis heute durchgesetzt, auch wenn hierbei ursprünglich nicht ausschließlich das Übersetzen als vollautomatischer Prozess gemeint war. Prinzipbedingt kann nur bei der Abstraktionsreduzierung beim Top-Down-Entwurf von einem Syntheseprozess gesprochen werden.

Tabelle 2.1: *Hierarchische Gliederung eines digitalen Systems aus funktionaler Sicht und typische Werkzeuge zur Bearbeitung und Synthese*

Abstraktionsebene	Werkzeuge		Beispiel
	Bearbeitung	Synthese	
System	Texteditor (Word, Framemaker) Graphikeditor (Illustrator, xfig)	-	DVI ▶ Video Konverter ▶ RGB
Algorithmisch	Programmeditor (Visual Studio, Emacs) Statecharteditor (CodeModeller, Unimodeler, Cossap)	High-Level-Synthese (PtolemyII, ARC, Daisy)	$y_i = e_i \cdot \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$

Tabelle 2.1: *Hierarchische Gliederung eines digitalen Systems aus funktionaler Sicht und typische Werkzeuge zur Bearbeitung und Synthese*

Abstraktionsebene	Werkzeuge		Beispiel
	Bearbeitung	Synthese	
RTL	Programmeditor (Emacs) Integrierte Entwicklungs-Plattform (ISE)	Logiksynthese (Design Vision, Leonardo Spectrum, Square-Dance)	
Gatter	Schaltungseditor (Orcad, ISE, Cadence)	Technologie-abbildung (LeonardoSpectrum, RASP, Cisy, Square-Dance)	
Transistor/ Technologieebene	Schaltungseditor (Cadence)	Zellgeneratoren	

Eine gebräuchliche hierarchische Gliederung eines digitalen Systems aus rein funktionaler Sicht mit den jeweils zugehörigen Werkzeugen zur Bearbeitung und Synthese ist in Tabelle 2.1 dargestellt. Die einzelnen Ebenen sind wie folgt charakterisiert:

1. Systemebene
   Ausgehend von der Idee für eine Schaltung wird auf der Systemebene die eigentliche Funktion des Systems spezifiziert und die globalen Randparameter wie z. B. Geschwindigkeit, Ein- und Ausgangssignale werden definiert. Die Beschreibung erfolgt in schriftlicher Form oder mit Hilfe von Graphiken und stellt eine Verständigungsgrundlage für die am Projekt beteiligten Entwickler dar. Zum Erstellen der Beschreibung kommen beliebige Text- und Graphikeditoren zum Einsatz.
   Bei der zugehörigen Systemsynthese wird die formale Spezifikation in mehrere Komponenten unter Berücksichtigung vorgegebener Randparameter untergliedert. Dieser Schritt kann, bedingt durch die praktisch beliebige Repräsentation der Ausgangsdaten, nur sehr eingeschränkt automatisiert werden. Teil der Systemsynthese ist es dementsprechend auch, die Spezifikation in eine durch Synthesewerkzeuge verwertbare Form zu bringen.

2. Algorithmische Beschreibung
   In der Algorithmischen Beschreibung wird das System durch eine Reihe nebenläufiger Prozesse beschrieben. Grundlegende Funktionseinheiten der Schaltung werden ermittelt und durch allgemeine Blöcke dargestellt. Die einzelnen Blöcke können über Signale Daten austauschen. Aus allen ermittelten Elementen lässt sich eine Grundstruktur des zu entwerfenden Systems ableiten.
   Aufgabe der Algorithmischen Synthese ist die Transformation eines Verhaltensmodells oder einer algorithmischen Beschreibung in eine Struktur auf Register-Transfer-Ebene (RTL). Es werden beispielsweise die benötigten Zustandsautomaten zur Ansteuerung der einzelnen Module generiert. Dies kann je nach Teilkomponente automatisch durch sog. High-Level-Compiler geschehen. Stehen entsprechende IP-Bibliotheken zur Verfügung, können schon auf dieser Ebene fertige Schaltungsteile eingesetzt und getestet werden.
   Im universitären Umfeld gibt es zahlreiche Ansätze, die aufbauend auf UML eine direkte Hardwaresynthese [1, 50], zum Teil auch verbunden mit einer Hardware/Software-Partitionierung [4], durchführen. Ein aktueller Trend ist vor allem SystemC [111]. Es handelt sich dabei um eine Klassenbibliothek, mit der in C++ beschriebene Algorithmen auf Hardwarestrukturen abgebildet werden können. Vor allem durch die vom Werkzeug unabhängige Schnittstelle und immer bessere Unterstützung durch kommerzielle Anbieter (Fortes mit Cyntesizer [86], Synopsys mit CoCentric [84] und Mentor Graphics mit Modelsim[45]) erscheint es wahrscheinlich, dass sich SystemC mittelfristig gegenüber alternativen Ansätzen durchsetzen wird. Weitere Entwicklungen von High-Level-Compilern für andere Hochsprachen gibt es zum Beispiel für Java [2, 14] oder C [59]. Bestrebungen zur High-Level-Synthese, ausgehend von graphisch repräsentierten synchronen Datenflussgraphen, gibt es beispielsweise im kommerziellen Umfeld von Synopsys mit Cossap oder als freie Implementierung mit PtolemyII [108].

3. Register-Transfer-Ebene
   In der RTL (Register-Transfer)-Ebene werden die Eigenschaften der Schaltung durch Transfers der Daten zwischen Registern und datenverarbeitenden Operationen spezifiziert. Aus den Operationen werden die benötigten elementaren Funktionseinheiten (Codierer, Multiplexer, ALU usw.) abgeleitet, instantiiert und eingebunden. Kontroll- und Datenfluss werden zusammengeführt und die entsprechenden Steuerstrukturen erzeugt.
   Die zugehörige Register-Transfer-Synthese lässt sich in Datenpfadsynthese und Kontrollpfadsynthese untergliedern:

   - Die Datenpfadsynthese ermittelt, für welche Signale Register zur Speicherung angelegt werden müssen. Für nicht zu speichernde Signale werden kombinatorische Logik- und Verbindungsleitungen vorgesehen.
   - Die Kontrollpfadsynthese erzeugt die Steuerung der Register-Transfers. Entsprechend werden auch hier Zustandsautomaten angelegt, um Treiber und Multiplexer anzusteuern. Das Übergangsnetzwerk der Automaten wird durch boolesche Gleichungen repräsentiert.

Das Ergebnis ist eine generische Netzliste mit den vollständigen miteinander verwobenen Daten und Kontrollflüssen. Schaltnetze werden als boolesche Gleichungen repräsentiert oder bestehen aus Elementen einer technologieunabhängigen Bibliothek.
Die nachfolgende Logiksynthese optimiert die durch boolesche Gleichungen beschriebenen kombinatorischen Netzwerke, generiert für die Automaten eine geeignete Zustandskodierung und setzt diese zusammen mit allen aus der Register-Transfer-Synthese generierten Registern und Multiplexern in eine generische Zieltechnologie um.
Im Unterschied zur System- und Algorithmischen Synthese wird die Register-Transfer-Synthese, wenn überhaupt, nur für spezielle Fälle ohne Synthesewerkzeug durchgeführt. Die zahlreichen (zum Teil speziell für die Register-Transfer-Synthese entwickelten) Repräsentationsformate wurden in den letzten Jahren durch hauptsächlich zwei Formate verdrängt, welche von den wichtigsten Entwurfswerkzeugen unterstützt werden. Zum einen ist dies VHDL (siehe „Die Hardwarebeschreibungssprache VHDL“ auf Seite 10), zum anderen Verilog [107], eine 1985 auf der Automated Integrated Design Systems vorgestellte Sprache, die 1995 zum IEEE-Standard 1364 [71] wurde. Verilog wurde im Unterschied zu VHDL speziell für die Synthese entwickelt. Dies erleichtert die Entwicklung von entsprechenden Synthesewerkzeugen im Vergleich zu VHDL-Synthesewerkzeugen erheblich, weshalb auch freie Synthesewerkzeuge, wie zum Beispiel Icarus Verilog [94], für eine Verilog-basierte Synthese verfügbar sind. Freie Implementierungen für eine VHDL-basierte Logiksynthese sind, mit Ausnahme des in dieser Arbeit vorgestellten Systems Square-Dance, nicht bekannt. Kommerzielle Synthesewerkzeuge, wie z. B. Mentor Graphics Leonardo Spectrum [96], Synopsys DC Expert [87], Xilinx ISE [95] oder Magma Blast RTL [81], unterstützen in der Regel beide Standards.

4. Gatterebene
   Die Gatterebene kann weitestgehend direkt aus der RTL-Beschreibung abgeleitet werden. Die einzelnen Funktionseinheiten sind hier in die grundlegenden Gatter zerlegt und durch Signale verbunden.
   Aufgabe der Synthese auf dieser Ebene ist die Technologieabbildung (Technology Mapping). Es werden hierbei die generischen Netzlisten auf eine vorgegebene Technologie abgebildet. Neben der funktionalen Übereinstimmung der Ausgangsbeschreibung mit der vorgegebenen Netzliste müssen zusätzlich die physikalischen Randbedingungen berücksichtigt werden. Die Randbedingungen ergeben sich aus globalen Vorgaben (maximale Stromaufnahme, Geschwindigkeit, Platzbedarf) bzw. aus der Technologie selbst (Fanin/Fanout). An die Technologieabbildung schließt die Platzierung und Verdrahtung, d. h. das Fixieren der Funktionseinheiten an einen bestimmten Platz auf dem zu entwerfenden Baustein sowie die geeignete Verbindung über elektrische Leitungen an. Die Platzierung und Verdrahtung kann losgelöst von der Technologieabbildung in eigenen Werkzeugen erfolgen [49]. Insbesondere bei den kommerziellen Werkzeugen wie z. B. Synopsys FPGAExpress [92], Magma Blast Fusion [80] sind beide Schritte in der Regel in einem Werkzeug integriert. Freie Werkzeuge zur Technologieabbildung gibt es nur vereinzelt, da die hierfür benötigten Technologiedaten für nichtkommerzielle Anwendungen nur schwer verfügbar sind. Dementsprechend arbeiten diese Werkzeuge wie z. B. SIS [16], Alliance [113] oder Electric [88] oft mit generischen oder veralteten Technologien, die zumeist in einem werkzeugspezifischen Format gegeben

sind. Ein freies Werkzeug für die Programmierung von FPGAs und CPLDs ist Rasp [109], wobei auch dieses nur mit offengelegten Programmierspezifikationen wie z. B. für die XC3000 FPGA Serie [114] von Xilinx sinnvoll einsetzbar ist.

5. Transistorebene
   Die Transistorebene wird nur unter bestimmten Bedingungen in die funktionale Sichtweise mit eingebunden. Auf dieser Ebene wird primär auf elektrische Probleme, wie z. B. den Entwurf und die Anbindung analoger Schaltungsteile an das Design und Anforderungen an die Peripherie der Schaltung eingegangen. Zumeist werden zu diesem Zweck vom Hersteller vorgegebene Schaltungsteile eingesetzt, so dass der aufwändige Handentwurf, der eher einen Ausnahmefall darstellt, entfällt. Zur Unterstützung von Entwicklern bei der Einführung neuer Technologien können sog. Zellgeneratoren Teile des Entwurfsprozesses automatisieren. Sie erlauben eine dynamische Anpassung an spezielle Geometrievorgaben, beispielsweise Längen-/Breitenverhältnisse oder Orientierung der Anschlusspins. Abhängig von benötigten Geschwindigkeiten und Treiberstärken kann auch eine automatische Dimensionierung der Transistoren erfolgen.

Je nach verwendetem Entwurfsprozess kommen noch zusätzliche Ebenen wie z. B. Geometrie hinzu, oder es werden, bedingt durch entsprechend mächtige Werkzeuge, einzelne Ebenen übersprungen.

Die Problematik der kurzen Entwicklungszeiten erfordert hohe Disziplin und planvolles Vorgehen auf der Seite des Entwicklers. Das Überspringen eines vorgesehenen Schrittes in der dargestellten Hierarchie kann möglicherweise die Entwicklungszeiten insgesamt verkürzen. Ein auftretender Fehler bedeutet dann jedoch oft eine zeitaufwändige Fehleranalyse, die den Erfolg des entsprechenden Projektes in Frage stellen kann.

2.1.2 Designstrategien

Zur Bearbeitung eines größeren Projektes wird zwischen zwei grundsätzlichen Designstrategien unterschieden:

- Beim *Bottom-up-Entwurf* wird auf der untersten Ebene begonnen. Transistoren werden zu elementaren Logikgattern verbunden, welche beim nächsten Schritt zu komplexeren Gattern verschaltet werden. Register werden aus Flipflops zusammengesetzt und mit den elementaren Schaltnetzen zu den grundlegenden Schaltwerken verbunden. Der Vorteil dieser Strategie ist darin zu sehen, dass man schon sehr früh im Entwurf zu erfüllende Randparameter (wie beispielsweise eine zu erreichende Geschwindigkeit) berücksichtigen kann. Zusätzlich ist es möglich, schon nach den ersten Designschritten mit der Low-Level Simulation zu beginnen und auf prozessbedingte Probleme einzugehen. In der Regel fällt es jedoch schwer, ein solches Projekt als Ganzes zu überschauen. Insbesondere bei umfangreichen Designs, wo z. B. auch mehrere unabhängige Entwickler an jeweils unterschiedlichen Teilen arbeiten, kommt es schnell zu Problemen, wenn es darum geht, die einzelnen Komponenten zusammenzufügen. Da spezielle Schwierigkeiten erst während des Entwurfsprozesses erkannt werden, ist es sehr pro-

blematisch, schon vorab Schnittstellen zwischen den einzelnen Komponenten zu vereinbaren.

- Der *Top-down-Entwurf* versucht die Probleme des Bottom-up-Entwurfs zu umgehen. Es wird auf der obersten Ebene begonnen und man durchläuft den Entwurfsprozess abwärts. Eine Strukturierung des Projektes geschieht sehr früh, so dass es problemlos möglich ist, das Projekt zu partitionieren, um es entsprechend auf mehrere Entwickler aufzuteilen. Die auf einer Ebene komplettierten Daten werden mit auf die nächste Stufe übernommen und dort weiter verfeinert und konkretisiert. Nachteil des Verfahrens ist, dass erst relativ spät feststeht, ob die gegebenen Randbedingungen (Laufzeit, Platzbedarf usw.) eingehalten werden können. Die Qualität einer frühen Abschätzung ist stark abhängig vom durchzuführenden Projekt und dem Erfahrungsstand der Entwickler. Gegebenenfalls ist ein Redesign notwendig, bei dem auf allen Ebenen (Jojo-Entwurf) versucht wird, das Projekt derart anzupassen, dass die Randbedingungen auf unterster Ebene erfüllt werden.

Beim tatsächlichen Entwurf werden häufig beide Designstrategien vermischt. Entsprechend der verwendeten Synthesewerkzeuge, deren Aufgabe zumeist eine Konkretisierung des Designs ist, wird von oben durch die Hierarchie gearbeitet. Zumeist ist jedoch bei Entwurfsbeginn schon klar, welches die kritischen Elemente des Designs sind. Ein gezieltes Low-Level-Design an diesen Punkten erlaubt es, trotz des globalen Top-down-Entwurfes, frühzeitig Aussagen über das endgültige Verhalten der Schaltung zu treffen.

2.2 Die Hardwarebeschreibungssprache VHDL

VHDL wurde im Auftrag des amerikanischen Verteidigungsministeriums zwischen 1983 und 1985 im Rahmen des VHSIC-Programms entworfen. Ziel war die Schaffung einer Sprache zur eindeutigen Beschreibung komplexer Systeme und deren Schnittstellen mit der Möglichkeit des Austausches von Modellen und Entwürfen zwischen voneinander unabhängigen Entwicklergruppen. 1987 wurde die Sprache zum ersten IEEE-Standard für Hardware-Beschreibungssprachen (IEEE 1076-1987). Entsprechend der IEEE-Richtlinien, wonach ein Standard alle fünf Jahre überarbeitet werden muss, wurde 1993 die Version 1076-1993 und 2002 die Version 1076-2002 definiert. Beide Versionen beinhalten vorwiegend kosmetische Änderungen. Inzwischen ist VHDL auch als ANSI-Standard definiert.

VHDL hat sich in den letzten Jahren neben Verilog zum De-facto-Standard als Sprache zur Beschreibung elektronischer Hardware entwickelt. Sie wird zur Beschreibung von digitalen Schaltungen sowohl bei der Simulation als auch bei der Synthese verwendet. Des Weiteren ist es mittlerweile üblich, VHDL als Austauschmedium zwischen den unterschiedlichen Entwicklungswerkzeugen zu verwenden.

Bedingt durch den ursprünglichen Anspruch der Sprache als reine Beschreibung von Systemen fehlen individuelle Sprachkonstrukte, die auf die Besonderheiten einer Abstraktionsebene eingehen, und müssen oft aufwändig anders modelliert werden. Beispielsweise gibt es für die Beschreibung von Zustandsmaschinen, einer elementaren Einheit der RTL-Beschrei-

bung, keine eindeutigen Sprachkonstrukte und es muss auf Elemente aus höheren Abstraktionsebenen ausgewichen werden, wobei die Eindeutigkeit verloren geht und die Fehleranfälligkeit der Beschreibung entsprechend hoch ist.

Der große Vorteil von VHDL gegenüber zahlreichen anderen Beschreibungssprachen ist die Möglichkeit, das zu entwickelnde System auf unterschiedlichen Abstraktionsebenen zu beschreiben. So kann in VHDL ein reines Verhaltensmodell erstellt werden, welches während des Entwurfsprozesses immer weiter bis zur Netzliste verfeinert wird, ohne die Entwicklungsumgebung zu verlassen. Unterschiedliche Modelle einer Komponente können mehrfach auf unterschiedlichem Abstraktionsniveau formuliert und beliebig mit anderen Komponenten zusammengefügt werden, um sie beispielsweise gemeinsam zu simulieren. Eine Transformation des gesamten Designs in eine neue Umgebung bei Änderung der Sichtweise ist also nicht notwendig. So können auftretende Fehler bei der Verfeinerung einzelner Module sehr schnell aufgedeckt werden, da jede Komponente beliebig innerhalb einer bereits verifizierten Umgebung getestet werden kann.

Bedingt durch die aufgeführten Punkte ergeben sich jedoch auch einige Probleme. Die Möglichkeit, sich auf verschiedenen Abstraktionsebenen zu bewegen, bedeutet gleichzeitig, dass es sehr vielfältige Sprachkonstrukte geben muss, um ein System zu beschreiben. Dieses erleichtert einerseits die Beschreibung eines Systems, führt jedoch andererseits zu einer auffällig hohen Zahl funktionaler Synonyme. Als Beispiel sind Listing 2.1 bis Listing 2.5 mit jeweils unterschiedlichen Beschreibungen für einen 1-aus-2 Multiplexer gegeben. Listing 2.4 und Listing 2.5 sind dabei sequentielle Beschreibungen und dürfen nur innerhalb eines Prozesses, einer Funktion oder einer Prozedur verwendet werden. Die anderen Beschreibungen können innerhalb einer Architekturbeschreibung beliebig positioniert werden.

```
1.  out <= a when sel = '0' \
      else b;
```

Listing 2.1 *Multiplexer, erste Variante.*

```
1.  with sel select
2.    out <= a when '0',
3.    b when others;
```

Listing 2.2 *Multiplexer, zweite Variante.*

```
1.  out <= (b and sel) or
2.         (a and not sel);
```

Listing 2.3 *Multiplexer, dritte Variante.*

```
3.   if sel = '0' then
4.    out <= a;
5.   else
6.    out <= b;
```

Listing 2.4 *Multiplexer, vierte Variante.*

```
1.  case sel is
2.    when '0' => out <= a;
3.    when others => out <= b;
4.  end case;
```

Listing 2.5 *Multiplexer, fünfte Variante*.

2.3 Logiksynthese aus VHDL-Beschreibungen

Der hohe Sprachumfang von VHDL erschwert das Entwickeln geeigneter Synthesewerkzeuge, welches auch relativ früh dokumentiert wurde [43]. Daneben gibt es auch zahlreiche Sprachelemente, die nur auf einer hohen Abstraktionsebene sinnvoll einsetzbar sind. So existieren Anweisungen, mit denen ein die Beschreibung interpretierender Simulator gesteuert werden kann, um beispielsweise abhängig von bestimmten Zuständen Meldungen auszugeben, Fehlerroutinen auszuführen oder auch die Simulation abzubrechen (`assert, report, severity`). Eine Synthese solcher Anweisungen ist nicht sinnvoll und wird auch von keinem Synthesewerkzeug durchgeführt. Bei anderen Sprachelementen ist diese Entscheidung weniger eindeutig. Eine Synthese von Metaanweisungen, die abhängig von bestimmten Zuständen alternative Schaltungselemente in ein Design einfügen, ist vorstellbar, wird aber nicht durchgängig praktiziert.

Diesen prinzipiellen Differenzen zwischen simulierbaren und synthetisierbaren VHDL-Beschreibungen wird mit sog. Coding-Style-Beschreibungen, welche Bestandteil jedes Synthesewerkzeugs sind, Rechnung getragen [102]. In diesen werden alle grundlegenden synthetisierbaren Sprachelemente und deren Anwendungsbereiche aufgeführt. Im September 1999 wurde der IEEE-Standard 1076.6-1999 veröffentlicht [73], um eine Portierbarkeit zwischen Werkzeugen unterschiedlicher Anbieter sicherzustellen. Dieser stellt zum einen eine Schnittmenge der bis dahin veröffentlichten Coding-Style-Beschreibungen dar und fixiert somit auch die Sprachelemente, die in zukünftigen Synthesewerkzeugen unterstützt werden müssen, vereinheitlicht zum anderen aber auch die unterschiedlichen Lösungen der Werkzeuge, fehlende Sprachelemente auszugleichen. So sind auch Sprachelemente aufgeführt, die es erlauben einen Syntheseprozess zu steuern, die jedoch selbst nicht Bestandteil der Sprache VHDL sind. So deaktiviert beispielsweise die Anweisung: `synthesis off` innerhalb eines VHDL-Kommentars den laufenden Syntheseprozess. Die Fixierung des synthesefähigen Sprachbereichs löst das Problem der unter den Werkzeugen austauschbaren Beschreibungen, verursacht gleichzeitig aber auch eine Reihe neuer Probleme. So ist es beispielsweise sehr aufwändig, Schaltwerke, die auf unterschiedliche Taktflanken reagieren (z. B. Dual-Edge-Flipflops), unter Einhaltung der im Standard beschriebenen Richtlinien zu beschreiben. Vorschläge für eine entsprechende Erweiterung des Standards existieren [40], sie sind bis heute jedoch noch nicht umgesetzt.

Als weitaus problematischer hat sich die Synthese von Beschreibungen erwiesen, die den IEEE-Standard 1076.6-1999 verletzen. Diese Verletzungen - unabhängig davon, ob sie bewusst eingesetzt werden, um beispielsweise vom Standard nicht unterstützte Schaltungselemente einzusetzen oder unbewusst als tatsächliche Programmierfehler auftauchen - werden nur selten in Schaltungen, die der Simulation entsprechen, umgesetzt [46, 25].

Als Beispiel ist in Listing 2.6 ein Flipflop beschrieben. Die Syntheseergebnisse der heute verfügbaren Werkzeuge sind jedoch meist Latches, wie z. B. in Abbildung 2.1 als Logikdiagramm des Synopsys-Syntheseergebnisses zu sehen ist. In diesem Logikdiagramm ist das Taktsignal `clk` über einen funktionslosen Multiplexer mit dem Enable-Eingang des generischen Speicherelements verbunden. Das Datensignal `d` ist mit dem Dateneingang `data_in` des Speicherelements verbunden. Alle weiteren Eingänge des Speicherelements,

einschließlich des Takteingangs, sind fest mit Masse verbunden. Mit dieser Beschaltung verhält sich das Speicherelement wie ein Latch. Grund für dieses Verhalten ist im gegebenen Beispiel die durch die Sensitivitätsliste implizit gegebene Bedingung *clk'event,* welche von den Werkzeugen nicht berücksichtigt wird. Nur zum Teil wird in gegebenem Fall zusätzlich eine Warnung ausgegeben, dass die Beschreibung bestimmten Bedingungen nicht genügt.

```
1.  bsp: process( clk )
2.  begin
3.    if clk = '1' then
4.      z <= d;
5.    end if;
6.  end process bsp;
```

Listing 2.6 *VHDL Prozess zur Beschreibung eines Flipflops.*

```
1.  bsp: process( clk )
2.  begin
3.    z <= d;
4.  end process bsp;
```

Listing 2.7 *VHDL Prozess zur Beschreibung eines Dual-Edge-Flipflops.*

Als weiteres Beispiel ist in Listing 2.7 die Beschreibung eines Dual-Edge-Flipflops gegeben. Dieses Bauelement ist als eigene Zelle in aktuellen Technologien verfügbar (z. B. Coolrunner von Xilinx), kann jedoch unter Einhaltung des Coding-Styles praktisch nicht beschrieben werden[41].

Die Verantwortung, ein dem Coding-Style entsprechendes fehlerfreies Design zu erstellen, wird somit vollständig auf den Entwickler übertragen. Sieht der verwendete Compiler keine Mechanismen vor, diese Abweichungen aufzudecken, ist die Postsynthesesimulation die derzeit einzige Möglichkeit, Synthesefehler aufzudecken. Eine Analyse der auftretenden Fehler ergibt, dass vor allem die Prozesssynthese unzuverlässig arbeitet. Dokumentationen zur Funktionsweise der Synthese für Prozesse oder freie Implementierungen sind praktisch nicht verfügbar, so dass sich nur indirekt durch Analyse der Synthesefehler auf die Ursache des Problems schließen lässt.

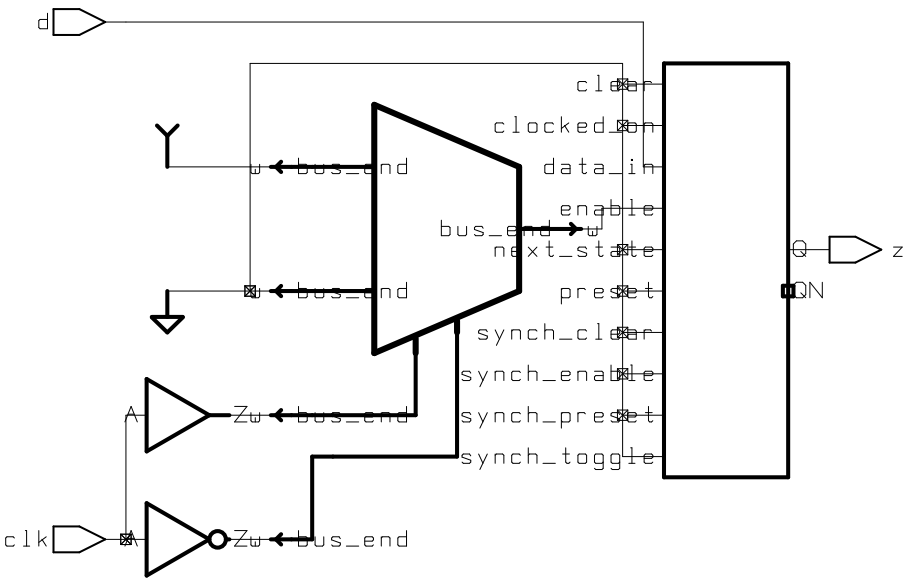


Abbildung 2.1 *Synopsys Syntheseergebnis von Listing 2.6.*

Die Synthese arithmetischer Strukturen wird bei den heute verfügbaren Werkzeugen im Rahmen der normalen Logiksynthese durchgeführt. Es wird ausgenutzt, dass in den jeder VHDL-Beschreibung zugrunde liegenden Packages Abbildungen arithmetischer Funktionen in logische Funktionen beschrieben sind. Wie in Abbildung 2.2 dargestellt, ist es ausreichend, nur die grundlegenden logischen Funktionen zu synthetisieren, da alle abstrakteren Funktionen auf diesen aufbauen.

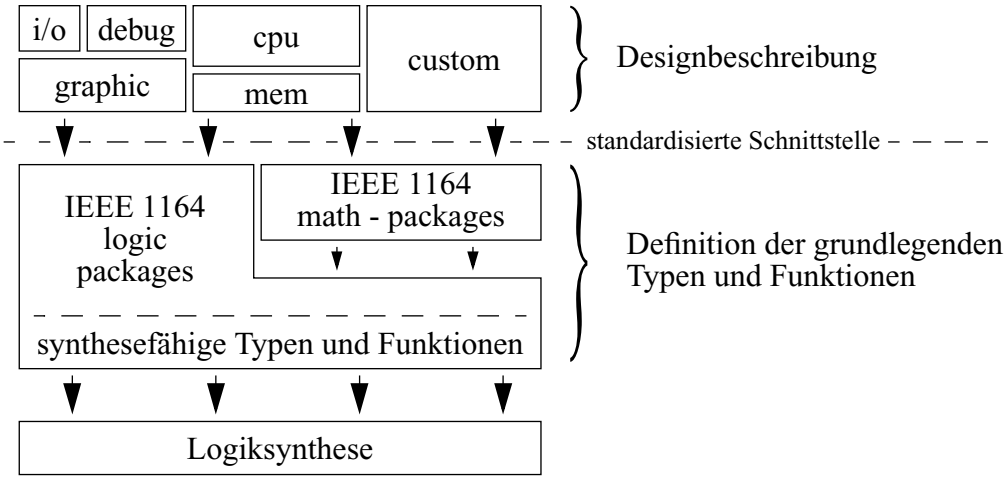


Abbildung 2.2 *Abhängigkeit eines Designs von standardisierten Packages und Anbindung an die Synthese.*

Als Beispiel ist in Listing 2.6 ein Auszug aus dem IEEE-1164-Package STD_LOGIC_UNSIGNED aufgeführt. Die Addition für vorzeichenlose ganze Zahlen wird in diesem auf elementare boolesche Operationen abgebildet. Die Implementierung einer speziellen Synthese für die Addition ist dementsprechend nicht notwendig. Der Vorteil dieser Vorgehensweise liegt in der vergleichsweise einfach durchzuführenden Synthese. Weiterhin ist es problemlos möglich, die vorhandenen Packages selbst um komplexe Funktionen zu erweitern und ohne Änderung an der Synthesesoftware sofort zu nutzen. Nachteilig ist der Verlust der abstrakten Information. Entsprechend der Formulierung eines Addierers als Carry-Ripple-Addierer, kann die Synthese Addierer ausschließlich in dieser Form generieren. Als Beispiel ist in Abbildung 2.3 das Syntheseergebnis für einen 8-Bit-Addierer für den SMC 0.13µm Prozess [79] von Artisan gegeben. In der Bibliothek des Technologieprozesses sind mehrere spezielle Zellen verfügbar, um beispielsweise Carry-select-Addierer aufzubauen. Zusätzlich sind 4-Bit-Addierer direkt als Zellen verfügbar. Aufgrund der Festlegung durch die VHDL-Packages auf Carry-Ripple-Addierer ist die Generierung von Addierern in anderen Schaltungstechniken innerhalb der Synthese nicht möglich. Diese Lücke wird von den Coregeneratoren gefüllt. Sie ermöglichen das Generieren von spezifischen Schaltungen wie zum Beispiel alle Arten von arithmetischen Einheiten beliebiger Tiefe, aber auch komplexere Schaltungen wie Signalfilter oder komplette Microcontroller. In der Abbildung 2.4 ist als Beispiel der Screenshot des Konfigurationsfensters mit allen Einstellmöglichkeiten für Addierer eines Coregenerators von Xilinx abgebildet. Die Coregeneratoren sind dabei direkt für die Zieltechnologie zugeschnitten und können entsprechend der gegebenen Anforderungen an die zu erzeugende Schaltung auch die bestmögliche Schaltung generieren.

```
1.  function "+"(L: UNSIGNED; R: UNSIGNED) return UNSIGNED is
2.    constant length: INTEGER := max(L'length, R'length);
3.  begin
4.    return unsigned_plus(CONV_UNSIGNED(L, length),
5.                         CONV_UNSIGNED(R, length));
6.  end;
7.
8.  function unsigned_plus(A, B: UNSIGNED) return UNSIGNED is
9.    variable carry: STD_ULOGIC;
10.   variable BV, sum: UNSIGNED (A'left downto 0);
11. begin
12.   if (A(A'left) = 'X' or B(B'left) = 'X') then
13.     sum := (others => 'X');
14.     return(sum);
15.   end if;
16.   carry :='0';
17.   BV := B;
18.   for i in 0 to A'left loop
19.     sum(i):=A(i) xor BV(i) xor carry;
20.     carry:=(A(i) and BV(i)) or (A(i) and carry)\
                                   or(carry and BV(i));
21.   end loop;
22.   return sum;
23. end;
```

Listing 2.6 *Auszug aus dem IEEE-1164-VHDL-Package:* `STD_LOGIC_UNSIGNED`*; Abbildung der Addition für vorzeichenlose ganze Zahlen in logische Strukturen.*

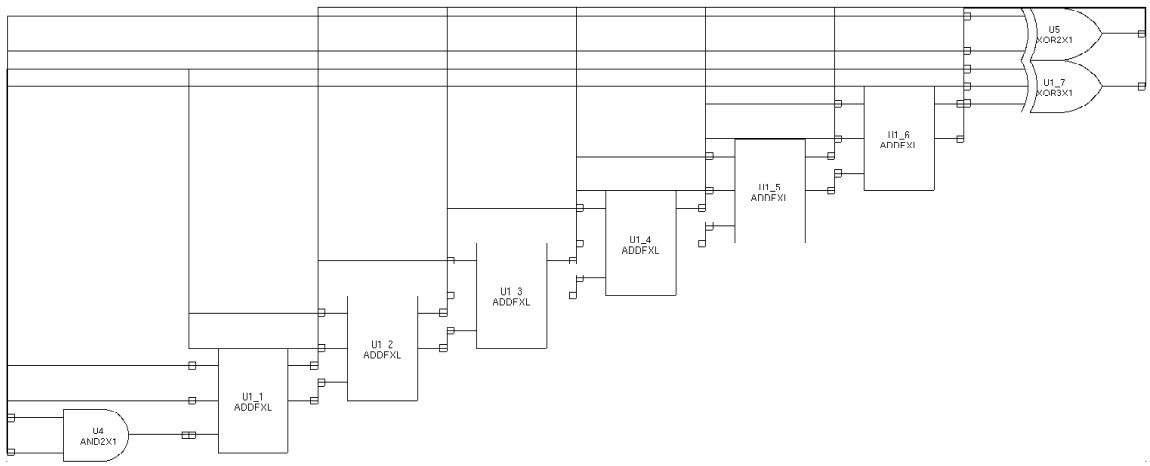


Abbildung 2.3 *Realisierung eines 8-Bit-Addierers im TSMC 0.13µm Prozess von Artisan.*

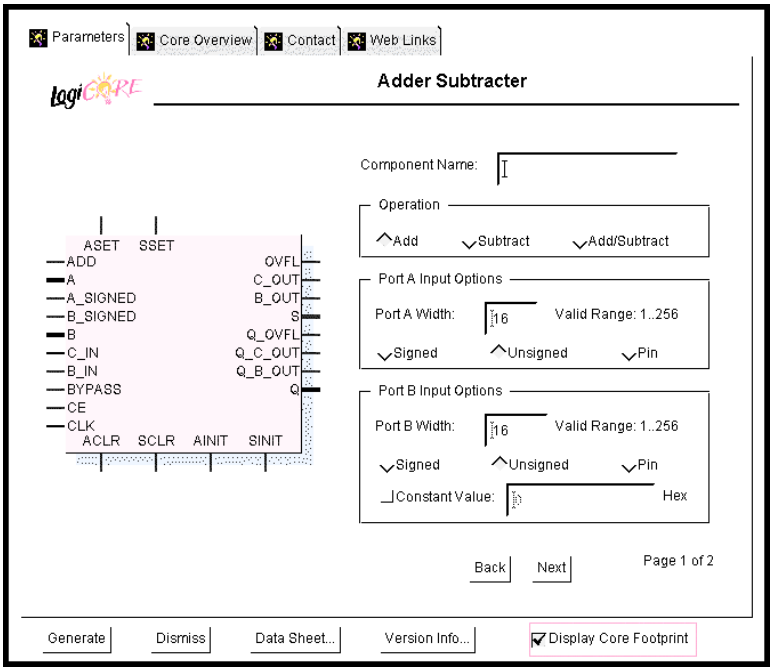


Abbildung 2.4 *Screenshot des CORE Generators[95] von Xilinx.*

Der Vorteil der vergleichsweise einfachen Logiksynthese in Verbindung mit auf das Design optimal zugeschnittenen Komplexzellen steht dem zusätzlichen Zeitaufwand insbesondere beim Entwurf von vergleichsweise einfachen Schaltungen gegenüber. Der Verzicht auf den Einsatz von Coregeneratoren, um Entwurfszeit zu sparen, schlägt sich dann auch in größeren und langsameren Schaltungen mit vergleichsweise hoher Stromaufnahme nieder.

2.4 Technologiebeschreibungen

Gut spezifizierte und gebräuchliche Ausgangsformate für Technologiebeschreibungen sind praktisch nicht verfügbar. Verfügbare Formate, wie sie zumeist an in Universitäten entwickelten experimentellen Synthesewerkzeugen wie beispielsweise SIS [26] zum Einsatz kommen (BLIF, KISS, SLIF), sind zwar gut dokumentiert, werden jedoch im industriellen Umfeld praktisch nicht eingesetzt. Verfügbare Bibliotheksbeschreibungen in diesen Formaten beschränken sich auf wenige, zumeist veraltete Technologieprozesse, wie z. B. den LSI 10k Prozess. In der Regel handelt es sich jedoch um selbst entworfene Zellbeschreibungen, die unter Umständen sogar auf das jeweilige im zugehörigen Werkzeug behandelte Problem zugeschnitten wurden.

Zellbeschreibungen für gebräuchliche kommerzielle Werkzeuge sind, wenn überhaupt, nur in einem geschlossenen Format zugänglich. Ausgangspunkt für dieses geschlossene Format ist ein zumeist offenes Format, welches beim Anbieter eines Technologieprozesses mit speziellen zum Synthesewerkzeug gehörenden Compilern in das interne Format umgewandelt wird. Als Beispiel ist im Abschnitt A.1 auf Seite 113 eine stark gekürzte Beschreibung einer Zelle im Synopsys-Library-Format gegeben. Sie enthält alle für die Technologieabbildung und Optimierung benötigten Informationen. Beschreibungen aktueller Technologien sind in dieser an sich optimalen Form jedoch nur schwer verfügbar.

Aus diesem Grund wurde als Ausgangsformat für die Zellbeschreibungen im Square-Dance-Entwicklungssystem VHDL nach der IEEE-1076.4-Spezifikation [74] gewählt. Diese Form der Beschreibung (im Anhang A.2 auf Seite 115 ist als Beispiel die Zellbeschreibung für ein Komplexgatter des ce81-Prozesses aufgeführt) ist vornehmlich für die Simulation nach der Synthese gedacht. Folglich enthält sie nur für die Simulation relevante Daten wie Funktionalität und Timing. Diese Form der Beschreibung muss jedoch unabhängig vom Synthesewerkzeug und Technologieprozess spätestens zur Simulation der synthetisierten Schaltung zwingend vorliegen und ist demzufolge unabhängig vom Ausgangsformat immer verfügbar.

2.5 Schaltungstechniken

2.5.1 Synchrone Schaltungen

Bei den meisten Logikentwürfen handelt es sich um synchrone, sequentielle Schaltwerke. Die primären Eingänge (siehe Abbildung 2.5) führen in ein kombinatorisches Schaltnetz und werden dort mit den Signalen aus einem Speicher verknüpft. Das Schaltnetz generiert sowohl die Ausgangssignale als auch den in den Speicher zu schreibenden Folgezustand.

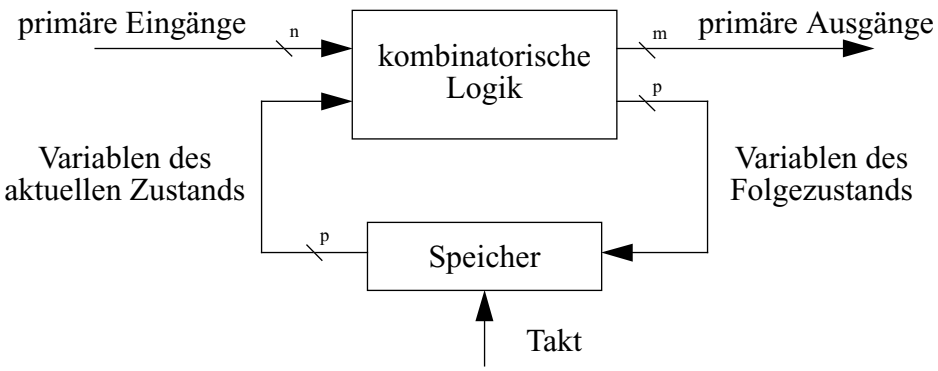


Abbildung 2.5 *Allgemeiner Aufbau eines synchronen Schaltwerks.*

In jeder Signalrückkopplung einer synchronen Schaltung befindet sich also mindestens ein Speicherelement. In jeder Teilschaltung wird in der Regel das gleiche Taktsignal verwendet, dessen Zykluszeit sich zumeist am langsamsten Schaltwerk der Schaltung orientiert. Der Entwurf synchroner Schaltungen garantiert die Funktion der Schaltung, solange diese maximale Signallaufzeit entsprechend berücksichtigt wird. Verglichen mit den weiteren hier aufgeführten Schaltungstechniken gestalten sich der Entwurf und die Verifikation der Funktion einer synchronen Schaltung relativ einfach. Die Möglichkeit, zu jedem Zeitpunkt Aussagen über den Zustand eines synchronen Schaltwerkes zu machen, erfordert jedoch, abhängig von der Komplexität der Gesamtschaltung, erheblichen Aufwand beim Entwurf der Taktnetze. Das Taktnetz muss zum einen garantieren, dass das Taktsignal bei allen Speicherele-

menten quasi gleichzeitig erscheint, zum anderen genug Treiberleistung zur Verfügung stellen, um sie tatsächlich auch gleichzeitig schalten zu können. Das Problem der Verdrahtung ist aufgrund der Mehrebenenverdrahtung weitgehend in den Hintergrund gerückt. Der insbesondere bei High-Speed-Schaltungen erhebliche Leistungsbedarf der Taktbäume und die Synchronisationsprobleme sind jedoch immer noch ein gegenwärtiges Problem.

Praktisch alle derzeitig verfügbaren kommerziellen Werkzeuge sind für den synchronen Entwurf ausgelegt, und bei dem überwiegenden Teil aller Entwürfe handelt es sich auch um synchrone Schaltungen.

2.5.2 Asynchrone Schaltungen

Eine Lösung, die bei synchronen Schaltungen auftretenden Probleme zu umgehen, sind asynchrone Schaltungen. Nachteilig sind hier die Empfindlichkeiten asynchroner Schaltungen gegen laufzeitbedingte Impulse, sog. Hazards, welche beim Entwurf verstärkt zu berücksichtigen sind und diesen entsprechend aufwändiger machen.

Im Gegensatz zu synchronen Schaltungen benötigen asynchrone Schaltungen kein globales Taktnetz, woraus sich vergleichsweise gute EMV-Eigenschaften und sehr geringe Ruheleistungen ergeben, welche zum Beispiel für den Mobilfunk interessant sind. Zusätzlich zeichnen sich asynchrone Schaltungen, je nach verwendetem Entwurfsansatz, durch eine höhere Geschwindigkeit und/oder geringeren Leistungsbedarf aus. Da der asynchrone Schaltungsentwurf in den letzten Jahren insbesondere durch die Entwicklungen im Mobilfunkbereich wieder an Bedeutung gewonnen hat [48, 23], wurden auch asynchrone Schaltungstechniken hinsichtlich ihrer Tauglichkeit zur Synthese aus VHDL-Hardwarebeschreibungen untersucht.

Es gibt zahlreiche Ansätze für den Entwurf asynchroner Schaltungen [30], die zum Teil auch neue Technologiebibliotheken erfordern. Es werden an dieser Stelle nur drei exemplarisch aufgeführt, für die entsprechende Entwurfsumgebungen bzw. Synthesewerkzeuge verfügbar sind:

- asynchrone Burst-Mode-Automaten
  Der Aufbau von asynchronen Burst-Mode-Automaten entspricht weitestgehend dem Aufbau von synchronen Schaltwerken. Im Unterschied zu diesen wird der Speicher durch speziell dimensionierte Verzögerungsglieder ersetzt. Je nach Anwendung kann auch die Verzögerung der kombinatorischen Logik selbst ausreichen, um auf zusätzliche Verzögerungsglieder verzichten zu können. Für die kombinatorische Logik kommen nur hazardfreie boolesche Funktionen in Frage. Da sich nicht alle Funktionen hazardfrei berechnen lassen, können nicht alle möglichen Automaten direkt als Burst-Mode-Automaten abgebildet werden. In [37] ist ein Verfahren beschrieben, Zustände beliebiger Automaten so umzucodieren, dass die Zustandsübergänge im booleschen Raum voneinander getrennt sind. Da jedoch auch die Rückkopplung an sich hazardfreier Logiken sog. sequentielle Hazards verursachen kann, können auch bei dieser Form je nach Schaltung Verzögerungsglieder in den Rückführungsleitungen notwendig sein, welches eine technologieunabhängige Beschreibung einer solchen Schaltung

erschwert. Eine Entwurfsumgebung für asynchrone Burst-Automaten ist in [36] beschrieben. Als Ausgangspunkt für eine Synthese von Burst-Mode-Automaten hat sich in den letzten Jahren die sog. Burst-Mode-Beschreibung [20,51], eine dem Petri-Netz ähnliche Beschreibung, etabliert. Die Generierung einer solchen Beschreibung innerhalb der VHDL-Logiksynthese wäre für Teilschaltungen möglich, aber unverhältnismäßig aufwändig.

- Micropipelines
  Micropipelines [61] stellen eine Alternative zu den im synchronen Design verwendeten Pipelines dar. Der typische FIFO-artige Aufbau von Micropipelines ist in Abbildung 2.6 dargestellt. Die abgebildeten Register können das anliegende Datenwort direkt auf den Ausgang durchschalten, so dass bei leerer Pipeline die Daten ungepuffert die komplette Pipeline durchlaufen. Erst wenn ein Datum durch nachfolgende Einheiten noch nicht verarbeitet werden kann, werden die jeweiligen Zwischenergebnisse der nachfolgenden Daten in jeweils der nächsten freien Stufe festgehalten.
  Im Gegensatz zum Burst-Mode-Automaten kann auf einen explizit hazardfreien Entwurf verzichtet werden

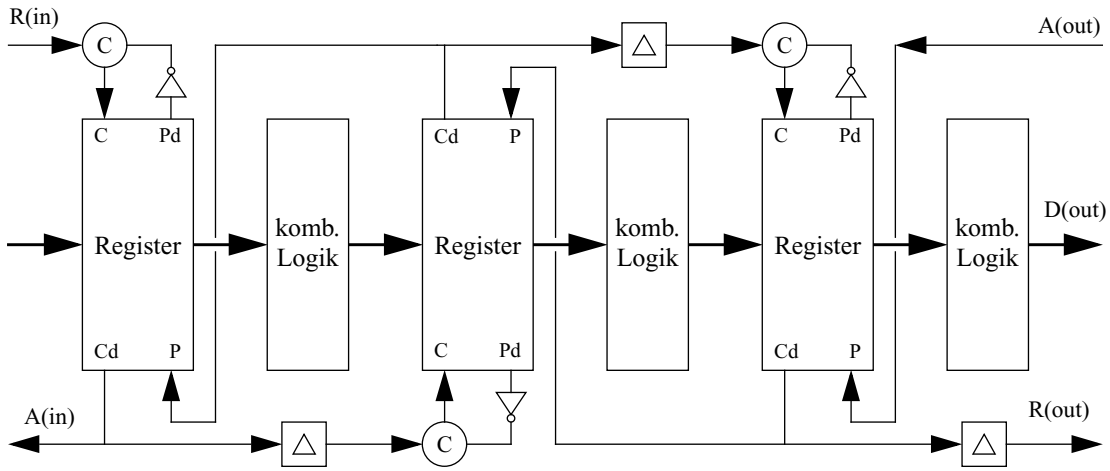


Abbildung 2.6 *Prinzipschaltung einer Micropipeline.*

Im Gegensatz zu den anderen asynchronen Schaltungstechniken existiert mit der AMULET-Prozessorserie [77, 22] ein umfangreiches und gut dokumentiertes Referenzdesign, um diese Schaltungstechnik hinsichtlich einer automatischen Synthese beurteilen zu können. Der AMULET1 ist eine Reimplementierung des ARM6-Prozessors [78] mit Micropipelines, wobei versucht wurde, die grundlegende Struktur zu erhalten. Zusätzlich wurde bei der Wahl der Fertigungstechnologie die des synchronen Vergleichsdesigns beibehalten. Obwohl der AMULET1 nicht durch automatische Synthesewerkzeuge generiert wurde, stellt das Design durch den gewählten Entwurfsprozess doch eine gute Referenz dar, was mit einem automatischen Werkzeug, ausgehend von einem gegebenen synchronen Design, hinsichtlich der Performance zu erwarten ist. Der AMULET1-Prozessor erreicht etwa 70% der Rechenleistung des ARM6-Prozessors bei gleicher Verlustleistung und etwa 20% höherem Platzbedarf.

Der 1996 fertig gestellte AMULET2-Prozessor ist ein komplettes Redesign des AMULET1-Prozessors. Die Grundstruktur wurde so geändert, dass die durch die Schaltungstechnologie gegebenen Besonderheiten stärker berücksichtigt werden konnten. Die Rechenleistung liegt zwischen der eines ARM7- und eines ARM8-Prozessors bei etwa gleicher Verlustleistung wie der ARM8-Prozessor. Bemerkenswert beim AMULET2-Prozessor sind die mit 3µW außergewöhnlich geringe Ruheleistung und das ausgezeichnete EMV-Verhalten.
Als Zielschaltungstyp, ausgehend von synchronen VHDL-Beschreibungen, sind Micropipelines aufgrund der starken Technologieabhängigkeiten zum gegenwärtigen Zeitpunkt ungeeignet. Eine Entwurfsumgebung und weitergehende Dokumentation sind in [65] beschrieben.

- dual-rail-encoding
  Unter Verwendung von dual-rail-encoding (z. B. [19]) aufgebaute Schaltungen kommen ebenfalls ohne globalen Takt aus. Das Synchronisationsproblem bei dieser Technik wird im Vergleich zu den anderen asynchronen Lösungen auf rein logischer Ebene gelöst. Der Datenfluss wird dabei mit den zugehörigen Synchronisationssignalen verflochten. Nachteilig wirkt sich der damit verbundene höhere Hardwareaufwand aus.
  Im Vergleich zu anderen asynchronen Techniken lehnt sich der Entwurfsverlauf stark an den synchronen Entwurf an und kann in der Regel mit vorhandenen Technologien realisiert werden [63]. Dual-rail-encoding wird im vorgestellten Synthesewerkzeug unterstützt, weitergehende Details sind themenspezifisch in die Arbeit mit eingebunden.

Aufgrund der Entwurfsproblematik wird heutzutage meist auf komplett asynchrone Designs verzichtet. So existieren nur vereinzelt komplexe Schaltungen, die komplett asynchron aufgebaut sind, wie z. B. die oben erwähnte ARM-kompatible AMULET-Prozessorserie. Wenn asynchrone Schaltungen verwendet werden, so sind es meist spezielle Schaltungen, die als Teilkomponenten in ein größeres synchrones Design eingebunden werden.

2.6 Einsatz von Datenbanken beim VLSI-Entwurf

Aufgabe einer Datenbank ist im Allgemeinen eine anwendungsunabhängige, dauerhafte Speicherung und Verwaltung von Daten [18]. Die Datenbank kann den Anwendungen dabei unterschiedliche Sichten auf die Daten bereitstellen, die Daten automatisch auf Konsistenz überprüfen, gleichzeitigen Zugriff mehrerer Benutzer ermöglichen und die Datensicherung automatisieren.

Im Kontext des VLSI-Entwurfs kann eine Datenbank auf unterschiedlichen Ebenen mit unterschiedlichen Aufgaben eingesetzt werden, von denen drei Einsatzgebiete exemplarisch behandelt werden.

- Datenbank zur Projektdatenverwaltung: Die Datenbank hat auf dieser Ebene primär die Aufgabe, alle projektspezifischen Daten zu ordnen und zu sichern. Je nach Einsatz können Abhängigkeiten verschiedener Projektdateien von unterschiedlichen Werkzeugen automatisch erkannt und bei Verletzung entsprechende Aktionen ausgelöst werden. Auf diese Weise kann die Datenbank beispielsweise für eine automatische Versionsverwaltung eingesetzt werden. Eine Schnittstelle zu den einzelnen Werkzeugen oder Schnittstellen, die der Datenbank eine Interpretation der Projektdaten erlauben, ist auf dieser Ebene nicht zwingend notwendig. Dementsprechend sind auch keine Einschränkungen hinsichtlich des einsetzbaren Datenbank-Managementsystems [100,101] gegeben.

- Datenbank als Werkzeugschnittstelle: In diesem Kontext wird die Datenbank als Schnittstelle zwischen Entwurfswerkzeugen benutzt, um die verschiedenen Sichtweisen auf ein Projekt direkt in Beziehung zu setzen. Zur Herstellung dieser Beziehungen müssen die jeweiligen Projektdaten direkt durch die Datenbank interpretierbar sein. Der Einsatz eines solchen Systems vereinfacht den Entwurf, da durchgeführte Änderungen auf anderen Abstraktionsebenen sofort sichtbar sind, schränkt jedoch die Auswahl der Werkzeuge stark ein, da entsprechend kompatible Schnittstellen in jedem Werkzeug vorausgesetzt werden. Stand der Technik sind sowohl Systeme, die auf speziell für den VLSI-Entwurf entwickelten Datenbanken aufbauen (Cadance Encounter digital IC design platform [89], PLAYOUT [60]), als auch Systeme, die auf bestehenden Standards [69] aufbauen [29].

- Datenbank zur werkzeuginternen Datenrepräsentation: Durch die in Datenbanken gegebenen Mechanismen zur effizienten Datenverwaltung sind diese auch für die werkzeuginternen Datenstrukturen interessant. Aus Effizienzgründen ist in diesem Kontext der Einsatz eines eigenen Datenbankprozesses nicht sinnvoll, weshalb entsprechende Programmbibliotheken verfügbar sind, die gewohnte Schnittstellen und Algorithmen bieten und dann im Kontext des Entwurfswerkzeugs lauffähig sind [97]. Moderne Programmiersprachen [82] bieten in der Regel von sich aus spezielle Datenorganisationsstrukturen [110], so dass beim Entwurf der Datenbank auf externe Prozesse oder Programmbibliotheken ganz verzichtet werden kann.

3 Theoretische Grundlagen

Ein hoher Anteil der bei der Transformation einer Hardwarebeschreibung in eine andere Abstraktionsebene durchgeführten Berechnungen besteht aus der Manipulation und Speicherung der zugrunde liegenden booleschen Gleichungen. Als wichtigste Randbedingung ist dabei die Erhaltung der Funktionalität einer gegebenen Schaltung sicherzustellen. Im Folgenden werden die im weiteren benötigten theoretischen Grundlagen für die innerhalb der Logiksynthese und Technologieabbildung durchführbaren Operationen behandelt und ein Verfahren für eine geeignete Repräsentation boolescher Funktionen innerhalb der Synthese vorgestellt.

3.1 Beschreibung von Schaltungsfunktionen mittels Boolescher Algebra

Die theoretische Grundlage des digitalen Schaltungsdesigns bildet die Boolesche Algebra mit zwei Elementen. Die Bedeutung dieses Zusammenhangs wurde sehr früh erkannt. Erste Überlegungen, die Boolesche Algebra in elektronischen Schaltnetzen zu verwenden, gehen auf den österreichischen Physiker Paul Ehrenfest (1880 - 1933) zurück [17]. Er schlug 1910 vor, mittels Boolescher Algebra die Schaltmatrizen von Telefonnetzen zu implementieren. Diese Technik wurde bis heute ständig weiter entwickelt und hat sich weitgehend gegenüber konkurrierenden Entwicklungen durchgesetzt.

Folgende Konventionen gelten im Weiteren für die Darstellung logischer Funktionen:

Definition 3.1: *Die Operationen* +, •, $\overline{\ }$ *angewendet auf die Menge* {0, 1} *sind wie folgt definiert:*

$a + b = \max\{a, b\}$
$a \bullet b = \min\{a, b\}$
$\overline{0} = 1$
$\overline{1} = 0$

($\{0, 1\}$, +, •, $\overline{\ }$,0, 1) *ist dann eine Boolesche Algebra, die als Schaltungsalgebra bezeichnet wird. Im Folgenden wird die Menge* {0, 1} *mit* $\mathbf{B}$ *bezeichnet. Ein Term* a • b *kann mit* ab *abgekürzt werden.*

Definition 3.2: *Eine Funktion mit* n *Variablen* $f : \mathbf{B}^n \rightarrow \mathbf{B}$ *wird als Schaltungsfunktion bezeichnet. Die Menge aller Funktionen mit* n *Variablen wird mit* $\mathbf{B}_n$ *bezeichnet.*

Jede Schaltungsfunktion ist eine boolesche Funktion. Die Bezeichnungen boolesche Funktion und Schaltungsfunktion werden deshalb im Weiteren synonym verwendet.

Ist **C** ein Schaltnetz mit n Eingangssignalen und m Ausgangssignalen, kann dessen Funktionalität als m-Tupel $f = (f_1, ... , f_m)$ von Schaltungsfunktionen beschrieben werden. Die Menge aller Funktionen mit n Eingangssignalen und m Ausgangssignalen wird mit $\mathbf{B}_{n,m}$ bezeichnet.

3.2 Eindeutige Repräsentation boolescher Funktionen

3.2.1 Motivation

Die Prüfung der funktionalen Äquivalenz zweier Funktionen ist eine fundamentale Aufgabe in fast allen Teilbereichen der Hardwaresynthese und Verifikation. Als NP (Nichtdeterministisch Polynomiell)-vollständiges Problem [12] wird versucht, an spezielle Probleme angepasste Lösungen zu finden, um mittels der problemspezifischen Randbedingungen den Suchraum einzuschränken. Die Grundlage der meisten verfügbaren Lösungen bilden graphentheoretische Ansätze [55]. Ein Beispiel sind die sog. BDDs (Binary Decision Diagrams) [3] bzw. auch deren Weiterentwicklungen, wie z. B. OBDDs (Ordered Binary Decision Diagrams), BDTs (Binary Decision Trees) usw. [44]. Ziel dieser Entwicklungen ist neben dem Vergleich auch die Optimierung boolescher Funktionen und damit verbunden deren effiziente und eindeutige Speicherung. Aktuell angebotene Programmbibliotheken wie z. B. das Paket BuDDy [24] erlauben eine effektive Funktionsanalyse, insbesondere auch komplexer Terme. Bei dieser, wie auch bei allen anderen Implementierungen, hängt die Leistungsfähigkeit jedoch sehr stark von der zu verarbeitenden Funktion selbst und von den Ressourcen, insbesondere dem verfügbarem Speicherplatz, ab.

In der vorliegenden Arbeit wurde ein anderer Ansatz gewählt, da besonders bei der Abbildung einer Schaltung auf eine bestimmte Zieltechnologie (Technology Mapping) je nach Problem eine hohe Zahl vergleichsweise kleiner Schaltungen möglichst ressourcenschonend verarbeitet werden muss.

3.2.2 Funktionstabellen und elementare Operationen

Jede kombinatorische Funktion kann prinzipiell vollständig mit allen möglichen Belegungen der Argumente und den zugehörigen Funktionswerten als Tabelle dargestellt werden. Die Repräsentation einer Funktion als Funktionstabelle erlaubt für kleine Argumentzahlen ($n \leq 9$) eine sehr effiziente Speicher- und Verarbeitungsmöglichkeit. Eine Funktionstabelle mit n Variablen hat 2^n Zeilen. Bei einer Variablenzahl von 9 sind dies 512 Bit. Zusätzlich zu den Funktionswerten sind die Variablen selbst auch zu sichern, was bei großen Tabellen aber nur einen Bruchteil des Speicherbedarfs ausmacht. Mit jeder hinzukommenden Variablen verdoppelt sich der Speicherbedarf und macht diese Form der Speicherung ineffizient. Im vorliegenden Programmpaket kommen prinzipbedingt komplexere Terme nicht vor (die Variablenanzahl der Logikzellen in gegenwärtig verfügbaren Technologien ist in der Regel kleiner als 9), weshalb auf die Implementierung einer alternativen Repräsentation verzichtet wurde.

Folgende grundlegende Operationen können durchgeführt werden:

- Konkatenation
  Zwei Funktionen f_1 und f_2 mit $f_1 = f(f_2)$ werden zu einer Funktion f_{12} zusammengefasst. Die Menge der Variablen der Funktion f_{12} ist die Vereinigung beider Variablenmengen abzüglich f_2. Der Rechenaufwand ist linear zur Länge der zu erstellenden Tabelle ($O(2^n)$). Beim Ermitteln der Variablenmenge muss darauf geachtet werden, dass die Ausgangsmengen nicht zwingend disjunkt sein müssen. Andernfalls werden überbestimmte Funktionen erzeugt, welche für die aktuelle Aufgabe keinen praktischen Wert haben.

- Variablentausch
  Zum eindeutigen Identifizieren einer Funktion ist es notwendig, die Variablen der Funktionstabelle in eine definierte Reihenfolge zu bringen. Bei einem tatsächlichen Umordnen der Tabelle muss jede Zeile der Tabelle mindestens einmal ausgelesen werden. Der minimale Rechenaufwand entspricht also der Anzahl der Zeilen einer Wertetabelle ($O(n^2)$). Insbesondere bei der Berechnung der eindeutigen Reihenfolge der Eingangsvariablen ist eine mehrfache Umordnung notwendig, was insbesondere bei umfangreichen Tabellen einen erheblichen Rechenaufwand erfordert. Es wurde deshalb ein Ansatz gewählt, der ohne eine tatsächliche Umordnung der Wertetabellen auskommt. Jede mögliche Belegung der Variablen kann als duale Zahl interpretiert werden, welche die für den entsprechenden Funktionswert gegebene Zeile der Funktionstabelle angibt. Jede Variable repräsentiert dabei eine bestimmte Stelle in dem dualen Zeilenindex der Funktionstabelle (im Beispiel Tabelle 3.1: $Zeile = 4x_2 + 2x_1 + x_0$) Wird diese Stelligkeit beim Umordnen der Tabelle gespeichert, braucht die Tabelle selbst nicht umgeordnet zu werden (siehe Tabelle 3.2), da sie implizit die Ergebnistabelle (siehe Tabelle 3.3) enthält.

Tabelle 3.1:
Ausgangstabelle für die logische Funktion: $f_1 = (x_2 + x_1) x_0$

Zeile	x_2	x_1	x_0	f_1
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	0
7	1	1	1	1

Tabelle 3.2:
Funktionstabelle nach dem Variablentausch von x_1 *und* x_0

Zeile	x_2	x_0	x_1	f_1
0	0	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	1	1	1
4	1	0	0	0
5	1	1	0	1
6	1	0	1	0
7	1	1	1	1

Tabelle 3.3:
Implizit durch Tabelle 3.2 gegebene Ergebnistabelle

Zeile	x_2	x_0	x_1	f_1
0	0	0	0	0
2	0	0	1	0
1	0	1	0	0
3	0	1	1	1
4	1	0	0	0
6	1	0	1	0
5	1	1	0	1
7	1	1	1	1

Unabhängig vom Umfang der Tabelle hat ein Variablentausch deshalb konstanten Rechenaufwand (O(1)). Ein zusätzlicher Aufwand ergibt sich nur, wenn die Funktionstabelle als Ganzes ausgelesen werden soll, um sie beispielsweise richtig geordnet als Teil einer Bibliothek zu sichern. Muss nur auf einzelne Funktionswerte zugegriffen werden, ist der Aufwand beim Zugriff gleich dem Aufwand beim Zugriff auf eine geordnete Tabelle, da auch bei dieser die Zeile nach gleicher Formel berechnet werden muss.

- Eliminierung nicht benötigter Variablen
  Ändert sich der Funktionswert bei der Negation einer Variablen für beliebige Belegungen der übrigen Variablen nicht, ist die entsprechende Variable bedeutungslos für die Funktion und kann eliminiert werden (siehe Beispiel Tabelle 3.4 bzw. Tabelle 3.5). Im einfachsten Fall wird jede Variable für sich getestet. Da für jede Variable alle Funktionswerte einmal ausgelesen werden müssen, beträgt der Gesamtrechenaufwand ($O(n \cdot 2^n)$). Die bei der Klassifizierung einer Funktion ermittelten Entropiewerte (siehe „Entropie als Ordnungskriterium der Funktionsvariablen“ auf Seite 26) können jedoch an dieser Stelle direkt benutzt werden, da nicht benötigte Variablen immer einen konstanten Entropiewert aufweisen.

Tabelle 3.4: *Funktionstabelle für die Funktion:* $f_2 = x_2 x_0$; x_1 *hat keinen Einfluss*

x_2	x_1	x_0	f_2
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabelle 3.5: *Sich aus Tabelle 3.4 ergebende Funktionstabelle nach Eliminierung von* x_1

x_2	x_0	f_2
0	0	0
0	1	0
1	0	0
1	1	1

3.2.3 Entropie als Ordnungskriterium der Funktionsvariablen

Zum Identifizieren einer bekannten Funktion, um beispielsweise alle einer gegebenen Funktion entsprechenden Zellen in der Technologiebibliothek zu ermitteln, ist es notwendig, diese in eine eindeutige Form zu bringen. Bei n Variablen sind n! Variablenanordnungen möglich und entsprechend viele Repräsentationen einer Funktion existieren als Tabelle. Beim vorliegenden Verfahren wird ausgenutzt, dass der Informationsgehalt (Entropie) einer Variablen vergleichsweise einfach berechnet werden kann und nicht durch ihre Stellung in der Funktionstabelle beeinflusst wird. Derart nach Entropie geordnete Wertetabellen sind eindeutig, d. h. gleiche Funktionen stimmen mit ihren Werten in den jeweiligen geordneten Tabellen überein.

In der Informationstheorie wird der Informationsgehalt e(x) einer Variablen x bei n verschiedenen Variablenwerten und der Wahrscheinlichkeit $p_m(x)$, dass ein Funktionswert mit dem Variablenwert m zum Zielfunktionswert gehört, durch die Entropie wie folgt berechnet:

$$e(x) = -\sum_{m=1}^{n} p_m(x) \cdot \ln \frac{1}{p_m(x)} \qquad \text{(Gl. 3-1)}$$

Im vorliegenden Fall ist die Zahl der möglichen Variablenwerte und Funktionswerte entsprechend der Basis des zugrunde liegenden dualen Zahlensystems gleich 2. Die Gesamtwahrscheinlichkeit für alle möglichen Funktionswerte einer Funktion entspricht 1, so dass die Wahrscheinlichkeit für das Auftreten eines Funktionswertes aus der Wahrscheinlichkeit des negierten Funktionswertes berechnet werden kann:

$$p_0(x) = 1 - p_1(x) \qquad \text{(Gl. 3-2)}$$

Für die grundsätzliche Unterscheidung der Entropien der einzelnen Variablen ist die absolute Entropie der einzelnen Variablen nicht notwendig. Dementsprechend ist es ausreichend, nur eine der beiden Wahrscheinlichkeiten für die Entropieberechnung heranzuziehen. Mit der gleichen Argumentation kann auf die Berechnung des Logarithmus verzichtet werden.

Die Wahrscheinlichkeit $p_{y,r}(x)$ eines Ereignisses r für eine gegebene Variable x berechnet sich aus der Anzahl der untersuchten Fälle (für alle Variablen: $(n-1)^2$) und der Anzahl des Auftretens eines Ereignisses $r_{x,y}$:

$$p_{x,y} = \frac{r_{x,y}}{(n-1)^2} \qquad \text{(Gl. 3-3)}$$

Die Anzahl der untersuchten Fälle ist bei einer gegebenen Funktionstabelle für alle Variablen gleich. Sie braucht für eine grundsätzliche Unterscheidung also ebenfalls nicht mit herangezogen zu werden. Der Rechenaufwand für die Berechnung der qualitativ reduzierten Entropien der einzelnen Variablen reduziert sich somit auf elementares Auszählen der einzelnen Fälle ($O(n^2)$).

Abhängig von der Funktion können Variablen gleiche Entropiewerte aufweisen. Ist dies der Fall, wird zunächst untersucht, ob die fraglichen Variablen funktional gleich sind. So ändern sich beispielsweise bei den Grundfunktionen (and, or, xor usw.) die Funktionswerte der entsprechenden Wertetabelle nicht, wenn die Variablen vertauscht werden. Ist dies der Fall, kann die Sortierung abgebrochen werden. Haben die Variablen hingegen unterschiedliche Funktionalität mit gleicher Entropie, wird eine bedingte Entropie berechnet. Hierzu werden alle anderen Variablen auf einen konstanten Wert (im Programm willkürlich 1) gesetzt und die sich daraus ergebende reduzierte Tabelle berechnet. Die reduzierte Tabelle wird dann

rekursiv dem gleichen Prozess wie die Ausgangstabelle unterworfen. Ist auch nach diesem Schritt keine Unterscheidung möglich, wird diejenige Ordnung der fraglichen Variablen gewählt, die den größten Wert der als Zahl interpretierten Funktionswerte der Tabelle aufweist.

3.3 Synthesegerechte Schaltungsbeschreibung mit VHDL

Eine Hardwarebeschreibung in VHDL besteht im Allgemeinen aus einer Entity-Deklaration, (siehe Zeile 1 bis 10 in Beispiel Listing 3.1) und einer oder mehrerer Architekturen (siehe Zeile 9 bis 14 bzw. Zeile 16 bis 22 in Listing 3.1). In der Entity-Deklaration wird die Schnittstelle des zu beschreibenden Hardwaremoduls spezifiziert. Die zugehörigen Architekturen enthalten die eigentliche Implementierung des Moduls, wobei mehrere Implementierungen für ein Modul in jeweils eigenen Architekturen zulässig sind. Grundsätzlich sind alle in einer Architektur aufgeführten Anweisungen nebenläufig und werden bei einer Synthese dementsprechend jeweils als parallel laufende Hardware umgesetzt. Ausnahmen bilden Anweisungen, die innerhalb eines sequentiell auszuführenden Blocks (Funktions-, Prozedur- oder Prozessblock) stehen. Architekturen und Entities können in Bibliotheken gebündelt und über das Schlüsselwort „`use`“ in anderen Architekturen als Komponente benutzt werden. Auf diese Weise kann ein hierarchisches Design vergleichsweise einfach in VHDL realisiert werden.

```
1.  entity add is
2.    port (
3.      a, b, c   : in  std_logic;
4.      sum, carry: out std_logic);
5.  end add;
6.
7.  architecture verhalten of add is
8.  begin  -- verhalten ( nicht synthesefähig )
9.    carry & sum <= a + b + c after 20 ns;
10. end verhalten;
11.
12. architecture funktion of add is
13. begin  -- funktion ( synthesefähig )
14.   sum <= a xor b xor c;
15.   carry <= (a and b) or (( a xor b ) and c );
16. end funktion;
```

Listing 3.1 *Beispielhafte VHDL-Entity-Deklarationen mit zwei zugehörigen Architekturen. Die Architektur* `funktion` *ist synthesefähig, die Architektur* `verhalten` *ist nicht synthesefähig.*

Bedingt durch die Entstehungsgeschichte als reine Modellierungssprache (siehe „Die Hardwarebeschreibungssprache VHDL“ auf Seite 10) ist nur ein Teil des Sprachumfangs von VHDL überhaupt für die Synthese geeignet. So gibt es beispielsweise Anweisungen für Bildschirmausgaben oder Dateioperationen, die im Kontext einer Hardwaresynthese nicht

zu verwerten sind. Im Listing 3.1 ist die Architektur `verhalten` nicht synthesefähig, da die Angabe von Verzögerungszeiten für die Synthese nicht zulässig ist.

Besonders problembehaftet für die Synthese ist die Instantiierung von Speicherelementen, da explizite Anweisungen für die Verwendung dieser in VHDL nicht existieren. Im Jahr 1999 wurde der IEEE-Standard 1076.6 verabschiedet, der die von der Synthese zu unterstützenden Varianten explizit aufzählt, um die Zahl der Varianten, die dieses Problem umgehen gering zu halten und eine einheitliche Grundlage für alle VHDL-Synthesewerkzeuge zu schaffen. Vom Standard abweichende Beschreibungen müssen von einem Synthesewerkzeug nicht akzeptiert werden, sollten jedoch, wenn sie nicht trotzdem korrekt übersetzt werden können, beim Auftreten ausdrücklich abgelehnt werden. Dieses Verhalten ist zum gegenwärtigen Zeitpunkt nur für spezielle Konstellationen von Werkzeug und Beschreibung gegeben.

Von den verschiedenen Varianten, sequentielle Algorithmen (einschließlich Speicherelementen) zu beschreiben, ist die Beschreibung innerhalb eines Prozesses die gängigste. Der Standard definiert vier grundsätzliche Formen für Prozesse, die eingehalten werden sollten, um eine fehlerfreie Synthese sicherzustellen:

- Flankengetriggerte Schaltwerke mit Sensitivitätsliste
  Flipflops für ein in einem Prozess aufgeführtes Signal oder eine Variable sollen generiert werden, wenn folgende Bedingungen zutreffen:
  - Das Signal oder die Variable hat eine synchrone Zuweisung;
  - Es gibt keinen Ausführungspfad, bei dem eine synchrone Zuweisung den von einer vorhergehenden asynchronen Zuweisung gesetzten Wert überschreibt, es sei denn, die asynchrone Zuweisung ist eine Zuweisung auf sich selbst;
  - Es ist möglich, alle Ausführungspfade zu den Zuweisungen statisch zu nummerieren;
  - In der Sensitivitätsliste sind sowohl das Taktsignal als auch alle Signale, die asynchrone Zuweisungen beeinflussen, aufgeführt;
  - Die Signalflanke des Taktsignals wird nur innerhalb von Bedingungen aufgeführt und bei Signalflanken innerhalb von Bedingungen muss es sich immer um die gleiche Flanke desselben Signals handeln.

  Ein Beispiel, welches den vorgegebenen Eigenschaften entspricht, ist in Listing 3.2 gegeben.

```
1.  edge: process: ( clk, reset )
2.  begin
3.    if reset = '1' then
4.      Q <= 0;
5.    elsif rising_edge( clk ) and enable = '1' then
6.      Q <= D;
7.    end if;
8.  end process;
```

Listing 3.2 *Synthesegerechter Prozess eines flankengetriggerten Schaltwerks mit Sensitivitätsliste.*

- Flankengetriggerte Schaltwerke mit wait-Anweisung
  Diese können, wie im Standard IEEE 1076.6 beschrieben, in die erste Form (flankengetriggerte Schaltwerke mit Sensitivitätsliste) transformiert werden. Nach der Transformation muss die Beschreibung für eine Synthese den zur ersten Form zugehörigen Bedingungen genügen. Ein entsprechendes Beispiel ist in Listing 3.3 gegeben:

```
1.  edge2: process
2.  begin
3.    wait on set, reset, clock until set = '1' or \
              reset = '1' or rising_edge( clock );
4.    if reset = '1' then
5.      Q <= '0';
6.    elsif set = '1' then
7.      Q <= '1';
8.    elsif rising_edge( clock ) then
9.      Q <= D;
10.   end if;
11. end process;
```

Listing 3.3 *Synthesegerechter Prozess eines flankengetriggerten Schaltwerks mit wait-Anweisung.*

- Pegelgetriggerte Schaltwerke
  Latches für ein in einem Prozess aufgeführtes Signal oder eine Variable sollen generiert werden, wenn folgende Bedingungen zutreffen:
  - Das Signal oder die Variable hat eine explizite Zuweisung;
  - Das Signal oder die Variable hat keine Zuweisung, die eine Signalflanke in einer zugehörigen Bedingung beinhaltet;
  - Es gibt Ausführungspfade, in denen es keine explizite Zuweisung für die Variable oder das Signal gibt.

  Ein entsprechendes Beispiel ist in Listing 3.4 gegeben:

```
1.  level1: process( d, enable )
2.    variable q_temp : bit;
3.  begin
4.    if enable = '1' then
5.      q_temp := d;
6.    else
7.      q_temp := q_temp;
8.    end if;
9.    q <= q_temp;
10. end process;
```

Listing 3.4 *Synthesegerechter Prozess eines pegelgetriggerten Schaltwerks.*

- Kombinatorische Logik
  Jeder Prozess, der weder eine Signalflanke noch eine wait-Anweisung enthält, modelliert entweder ein pegelgetriggertes Schaltwerk oder eine kombinatorische Logik.
  Kombinatorische Logik für ein Signal oder Variable soll generiert werden, wenn folgende Bedingungen zutreffen:
  - Es existiert in jedem Ausführungspfad mindestens eine Zuweisung für das Signal oder die Variable;
  - Das Signal oder die Variable wird in allen Ausführungspfaden nicht gelesen, bevor sie beschrieben wird;
  - Alle Signale oder Variablen, die in einer Zuweisung gelesen werden, müssen in der Sensitivitätsliste aufgeführt werden.

  Ein entsprechendes Beispiel ist in Listing 3.5 gegeben:

```
1.  combi: process( b )
2.  begin
3.    if a = '1' then
4.      c <= b;
5.    else
6.      c <= not b;
7.    end if;
8.  end process;
```

Listing 3.5 *Synthesegerechter Prozess mit kombinatorischer Logik.*

3.4 Dual-rail-encoding

Neben der Synthese synchroner Schaltungen ist im Synthesewerkzeug Square-Dance zusätzlich die Möglichkeit gegeben, Schaltungen in dual-rail-Technik zu synthetisieren. Ausgangspunkt ist dabei eine von der Logiksynthese SibaS generierte synchrone Schaltung. Die Schaltung wird innerhalb der Technologieabbildung ALTeM in Register und kombinatorische Schaltnetze zerlegt und getrennt in eine entsprechende dual-rail-Schaltung übersetzt (siehe „Dual-rail-encoding“ auf Seite 77).

Tabelle 3.6: *Signalverschlüsselung beim dual-rail-encoding*

Signalleitungen		Zustand
h	l	
0	0	u
0	1	0
1	0	1
1	1	f

Beim dual-rail-encoding werden für jedes logische Signal zwei Signalleitungen vorgesehen, welche das Signal, wie in Tabelle 3.6 dargestellt, verschlüsseln. Neben den normalen booleschen Zuständen **0** und **1** lässt sich so ein dritter Zustand **u** für „undefiniert“ verschlüsseln. Ein „high“ auf einer Signalleitung signalisiert ein gültiges Datum mit dem der Signalleitung zugewiesenen Wert. Der Fall, dass beide Signalleitungen „high“ sind, ist nicht zulässig, kann aber für eine Fehlererkennung **f** benutzt werden. Der hazardfreie Entwurf für die kombinatorische Logik gestaltet sich vergleichsweise einfach, da jedes Signal auch negiert vorliegt. In den Tabellen 3.7 - 3.9 sind die logischen Grundfunktionen gegeben.

Tabelle 3.7: *Dual-rail-Wertetabelle für logisch und*

and	0	1	u	f
0	0	0	0	f
1	0	1	u	f
u	0	u	u	f
f	f	f	f	f

Tabelle 3.8: *Dual-rail-Wertetabelle für logisch oder*

or	0	1	u	f
0	0	1	u	f
1	1	1	1	f
u	u	1	u	f
f	f	f	f	f

Tabelle 3.9: *Dual-rail-Wertetabelle für logisch nicht*

not	0	1	u	f
	1	0	u	f

Jeder Signalwechsel am Eingang bzw. Ausgang eines kombinatorischen dual-rail-Schaltnetzes muss zur Vermeidung undefinierter Zustände vom oder zum Zustand **u** erfolgen. Es ergibt sich der in Tabelle 3.10 dargestellte Ablauf:

Tabelle 3.10: *Ablauf der Signalwechsel am dual-rail-Schaltnetz*

Zustand	Signaleingänge	Signalausgänge
1	alle sind **u**	alle werden **u**
2	einige werden **1** oder **0**	alle bleiben **u**
3	alle sind **1** oder **0**	alle werden **1** oder **0**
4	einige werden **u**	alle bleiben **1** oder **0**

Die für den korrekten Ablauf erforderliche Synchronisation wird von den dem Schaltnetz vor- bzw. nachgeschalteten Registern übernommen. Da jedes Register seinen eigenen Zustand kennt, ist eine Synchronisationsleitung ausreichend, mit der ein vollständig im Register eingetroffenes Datum an das vorgehende Register quittiert wird.

4 Logiksyntheseverfahren SibaS

4.1 Überblick

Aufgabe des Logiksyntheseverfahrens SibaS (Simulation-based-Synthesis) ist es, eine gegebene Hardwarebeschreibung in VHDL in eine generische Netzliste (A-RTL) zu übertragen. Der Aufbau des Syntheseverfahrens und die Einordnung in den Syntheseprozess ist in Abbildung 4.1 dargestellt. Die abzubildenden Hardwarebeschreibungen werden zunächst

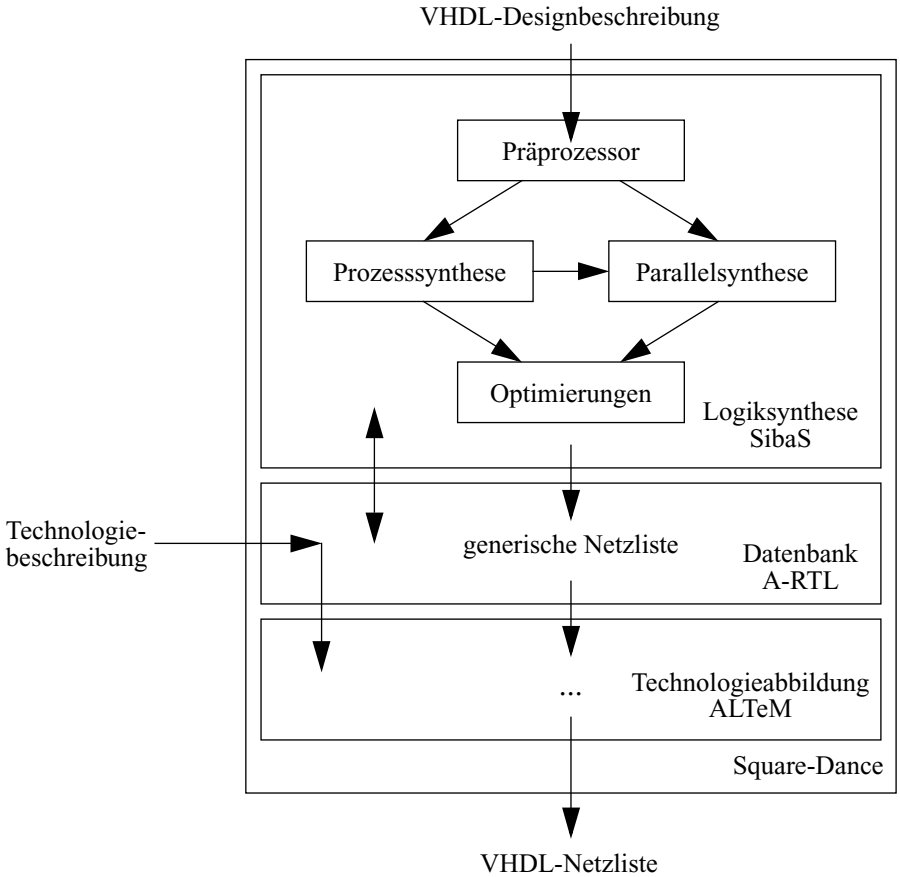


Abbildung 4.1 *Aufbau des Logiksyntheseverfahrens SibaS und Einordnung in den Entwurfsfluss.*

im Präprozessor einer syntaktischen und semantischen Prüfung unterzogen und sequentiell je nach Kontext der Beschreibung an die Prozesssynthese bzw. Parallelsynthese zur Übersetzung weitergereicht. Die dort generierten generischen Netzlisten werden im Optimierungsblock wieder zusammengefügt und optimiert. Die optimierte Netzliste kann dann von der Technologieabbildung weiter verarbeitet werden.

4.2 Präprozessor

Aufgabe des Präprozessors ist die syntaktische und semantische Prüfung der gegebenen Beschreibungen. Neben der Prüfung, ob die Beschreibung in sich widerspruchsfrei ist (z. B. ob alle Variablen und Signale korrekt deklariert und verwendet werden), wird eine Testsynthese (die sog. Kopfsynthese), durchgeführt, um festzustellen, ob die gegebenen Beschreibungen zueinander widerspruchsfrei sind und keine Beschreibungen fehlen.

Ausgehend von der globalen Instanz wird für alle in dieser Beschreibung verwendeten Instanzen anhand der jeweiligen Komponentendeklaration die zugehörige Entity-Deklaration generiert. Aus der im Listing 4.1 gegebenen Instantiierung `ierr` der Komponente

```
1.  architecture ver of uart is
2.
3.   component reg2
4.    generic( length : integer := 8 );
5.    port( d: in  std_logic_vector(length-1 downto 0);
6.          we: in  std_logic;
7.       reset: in  std_logic;
8.      regout: out std_logic_vector(length-1 downto 0));
9.    end component;
10.
11.   (...)
12.
13. begin
14.
15.   ierr : reg2
16.     generic map(4)
17.     port map(d(3 downto 0), weier, reset, ier);
18.
19.   (...)
20.
21. end ver;
```

Listing 4.1 *VHDL Architektur mit Komponentendeklaration und zugehöriger Instantiierung.*

`reg2` wird die in Listing 4.2 gegebene Entity-Deklaration generiert. Die Instantiierung wird als korrekt akzeptiert, wenn die generierte Entity-Deklaration mit der tatsächlichen Entity-Deklaration nach Auswertung der vorgegebenen generischen Daten übereinstimmt. Im Beispiel ist die tatsächliche Entity-Deklaration in Listing 4.3 gegeben. Nach Auswertung des generischen Teils (Zeile 2 in Listing 4.3) mit den vorgegebenen Werten (Zeile 16 in Listing 4.1) ergibt sich eine zu Listing 4.2 äquivalente Beschreibung. Der Vorgang wird unterbrochen und eine entsprechende Meldung ausgegeben, wenn die generierte Entity-Deklaration mit der tatsächlich vorhandenen nicht übereinstimmt bzw. keine Entity-Deklaration vorhanden ist.

```
1.  entity reg2 is
2.   port(d: in std_logic_vector( 3 downto 0 );
3.       we: in std_logic;
4.    reset: in std_logic;
5.   regout: out std_logic_vector( 3 downto 0 ));
6.  end reg2;
```

Listing 4.2 *Aus Listing 4.1 für* `ierr` *durch den Präprozessor generierte Entity-Deklaration für* `reg2.`

```
1.  entity reg2 is
2.    generic( length : integer := 8 );
3.    port( d: in std_logic_vector( length-1 downto 0 );
4.         we: in std_logic;
5.      reset: in std_logic;
6.     regout: out std_logic_vector( length-1 downto 0 ));
7.  end reg2;
```

Listing 4.3 *Entity-Deklaration von* `reg2.`

Sind für alle in der aktuellen Architektur verwendeten Instanzen passende Beschreibungen vorhanden, wird die Logiksynthese rekursiv für jede Instanz durchgeführt. Die eigentliche Logiksynthese der aktuellen Architektur wird also erst durchgeführt, wenn für jede verwendete Instanz die Logiksynthese erfolgreich abgeschlossen wurde, also die generischen Netzlisten für alle Instanzen vorliegen.

Die Beschreibung wird vom Präprozessor anweisungsweise an die Parallelsynthese übergeben, welche diese in eine entsprechende Schaltung übersetzt. Sequentielle Blöcke werden als Ganzes an die Prozesssynthese weitergeleitet. Die von den Syntheseeinheiten generierten Schaltungen werden anhand der Signal- bzw. Variablennamen miteinander verschaltet. Jedes Signal bzw. Variable darf dabei nur eine Quelle haben, kann aber beliebig viele Senken besitzen. Wurde die Übersetzung als Ganzes erfolgreich abgeschlossen, wird das generierte Netz in Form einer innerhalb der A-RTL-Datenbank angelegten Datenstruktur (siehe „Datenbank A-RTL“ auf Seite 85) an den Optimierer weitergegeben.

4.3 Prozesssynthese

4.3.1 Überblick

Bedingt durch die Entwicklung der Sprache VHDL als reine Spezifikationssprache ist die Synthese von sequentiellen VHDL-Beschreibungen (im Folgenden als Prozesse bezeichnet) problematisch. Ziel der Sprachentwicklung war es, den Entwickler in seiner Arbeit umfassend zu unterstützen. Für unterschiedliche Aufgaben stehen dem Entwickler eigene spezielle Sprachelemente zur Verfügung, die es erlauben, gegebene Probleme vergleichsweise

einfach zu lösen. Die Vielfalt der verfügbaren Sprachelemente ermöglicht es jedoch auch, vielfältige Beschreibungsmöglichkeiten für gleiche Problemstellungen zu finden. Die in der Beschreibung des Verfahrens gewählten Beispiele entsprechen nicht unbedingt dem Standard IEEE 1076.6-2004. Ziel ist die Demonstration unterschiedlicher Beschreibungsweisen um zu zeigen, wie das vorgestellte Verfahren mit diesen umgeht.

Der grundlegende Aufbau von Prozessen widerspricht den Strukturen, wie sie in digitalen Schaltungen realisiert sind. Ein Prozess besteht aus einer Folge von Anweisungen, die zeitlich sequentiell auszuführen sind. In digitalen synchronen Schaltungen werden in einem Taktzyklus alle aktuellen Anweisungen parallel ausgeführt. Die Forderung, dass sich aus einer gegebenen RTL-Beschreibung die Hardwarestruktur unmittelbar ableiten lässt, ist bei der Verwendung von Prozessen somit nicht gegeben. Demgegenüber existieren jedoch keine alternativen Sprachelemente zur Beschreibung von Speicherelementen wie z. B. Flipflops oder Latches. Die Prozesssynthese ist somit ein notwendiger Bestandteil jeder VHDL-RTL-Synthesesoftware. Dementsprechend sind Prozessbeschreibungen Bestandteil des Standards IEEE 1076.6-2004, jedoch sind die für die verschiedenen Schaltungsarten zu verwendenden Formen vorgegeben (siehe „Logiksynthese aus VHDL-Beschreibungen“ auf Seite 12). Die Art der auftretenden Synthesefehler verfügbarer Werkzeuge legt die Vermutung nahe, dass es sich bei der konventionellen Prozesssynthese um eine Mustererkennung handelt. Es wird dabei versucht, bei einer gegebenen Beschreibung die im Standard beschriebenen Sprachschablonen wiederzuerkennen und die Beschreibung entsprechend einzupassen. Diese Vorgehensweise ist einfach und nahe liegend, versagt aber bei Beschreibungen, die sich nicht an das vorgegebene Schema halten und verursacht die oben aufgeführten Probleme.

Die Prozesssynthese des Logiksyntheseverfahrens SibaS verwendet einen eigenen Ansatz. Die SibaS-Prozesssynthese besteht im Prinzip aus einem für diese Aufgabe entwickelten Simulationskern, der die Prozessbeschreibung in eine interne Verhaltensbeschreibung ohne sequentielle Informationen überführt. Durch den Schritt über die Simulation ist eine vollständige Lösung von Coding-Style-Beschreibungen möglich. Die weiteren Syntheseschritte bauen ausschließlich auf den Simulationsergebnissen auf, so dass die vom Entwickler gewählte Form der zu übersetzenden Beschreibungen keinen Einfluss auf die Synthese selbst hat. Unabhängig von den sich daraus ergebenden Konsequenzen ist die vollständige Kompatibilität mit den bestehenden Werkzeugen gegeben, da es nicht möglich ist, unter Einhaltung des Coding-Style Schaltungen zu beschreiben, die nicht synthesefähig sind.

Das SibaS-Prozesssyntheseverfahren setzt sich aus folgenden Schritten zusammen, die linear abgearbeitet werden:

1. Erstellung der Unterprozesse
2. Trennung kombinatorischer von sensitiven Bedingungen
3. Erstellen der Wertetabellen
4. Bestimmung des Schaltungstyps und der Steuersignale
5. Erstellen der Schaltung
6. Abschließende Überprüfungen und Optimierungen

4.3.2 Erstellung der Unterprozesse

In einem Prozess ist es möglich, beliebig viele Schaltungen unterschiedlichen Typs parallel zu beschreiben. Jedes in einem Prozess erzeugte Signal kann jedoch nur von genau einer Schaltung eines Typs generiert werden. Im ersten Schritt wird also für jedes in einem Ausgangsprozess erzeugte Signal ein eigener Unterprozess generiert. Dieser Schritt trennt automatisch die im Prozess ineinander verzahnten unterschiedlichen Schaltungstypen voneinander und vereinfacht so die nachfolgenden Syntheseschritte. Dieser Schritt wird auf syntaktischer Ebene durchgeführt (Program Slicing [11,64]).

Im Listing 4.4 als Ausgangsprozess wird beispielsweise für die Signale `c`, `d` und `e` jeweils ein eigener Unterprozess (siehe Listing 4.5 bis Listing 4.7) erzeugt, welche zusammen den Ausgangsprozess ersetzen. Alle im Ausgangsprozess aufgeführten Bedingungen bleiben in den Unterprozessen unverändert erhalten bzw. werden eliminiert, sofern der Körper der Bedingung leer ist. Die im Listing 4.4 miteinander verzahnten unterschiedlichen Schaltungstypen sind in den generierten Unterprozessen unabhängig voneinander beschrieben. Listing 4.5 beschreibt ein Schaltnetz, Listing 4.6 ein flankengetriggertes Schaltwerk und Listing 4.7 ein pegelgetriggertes Schaltwerk.

```
1.  test: process( b, a, e )
2.  begin
3.    c <= '0';
4.    if b = '0' then
5.      if a = '1' then
6.        c <= e;
7.        e <= d;
8.        if a'event then
9.          d <= c;
10.       end if;
11.     end if;
12.   else
13.     d <= '1';
14.   end if;
15. end process test;
```

Listing 4.4 *Ausgangsprozess.*

```
1.  t/c: process( b, a, e )
2.  begin
3.    c <= '0';
4.    if b = '0' then
5.      if a = '1' then
6.        c <= e;
7.      end if;
8.    else
9.    end if;
10. end process t/c;
```

Listing 4.5 *Unterprozess von Listing 4.4 für Signal c.*

```
1.  t/d: process( b, a, e )
2.  begin
3.    if b = '0' then
4.      if a = '1' then
5.        if a'event then
6.          d <= c;
7.        end if;
8.      end if;
9.    else
10.     d <= '1';
11.   end if;
12. end process t/d;
```

Listing 4.6 *Unterprozess von Listing 4.4 für Signal d.*

```
1.  t/e: process( b, a, e )
2.  begin
3.    if b = '0' then
4.      if a = '1' then
5.        e <= d;
6.      end if;
7.    else
8.    end if;
9.  end process t/e;
```

Listing 4.7 *Unterprozess von Listing 4.4 für Signal e.*

4.3.3 Trennung kombinatorischer von sensitiven Bedingungen

In diesem Schritt werden die einzelnen Unterprozesse in ein internes Format übersetzt. Es werden hierzu die Zuweisungen zusammen mit den jeweils zu erfüllenden Bedingungen ermittelt. Ziel ist die Lösung von der VHDL-Syntax und die Aufbereitung der Daten für die im nächsten Schritt durchzuführende Simulation.

Ein grundlegender Bestandteil der VHDL-Beschreibung von Prozessen ist die Liste der prozessauslösenden Signale. Ein Prozess wird genau dann ausgeführt, wenn sich ein Signal der Sensitivitätsliste ändert (Signalflanke). Des Weiteren können auch im Prozesskörper selbst Pegeländerungen von Signalen in Bedingungen abgefragt werden. Zustandsänderungen von den Signalen werden selbst wie Zustände behandelt, um diese Sprachelemente mit in den Syntheseprozess zu übernehmen. Neben den statischen logischen Zuständen (wahr/ falsch) kommen noch zwei dynamische (fallende/steigende Flanke) hinzu. Bedingt durch die beiden zusätzlichen Zustände der Signale werden die Wertetabellen im weiteren Syntheseprozess mit 4^n umfangreicher als gewöhnlich. Es wird eine Trennung von kombinatorischen und sensitiven Bedingungen vorgenommen, um dieses Problem zu reduzieren:

Für jede Signalzuweisung im generierten Unterprozess werden der rechte Ausdruck der Zuweisung (im Folgenden ϕ_x (mit $x \in N$)) und die kombinatorischen und sensitiven Bedingungen ermittelt. ϕ_x kann ein logischer Ausdruck, eine Konstante oder ein Literal sein.

Eine kombinatorische Bedingung α_x ist die logische Verknüpfung von sog. kombinatorischen Tupeln. Ein kombinatorisches Tupel besteht aus einem Literal λ_x und einem dem Literal zugeordneten Attribut μ_x mit $\mu_x \in (1, 0)$. Sie wird im Folgenden mit der Form $(\lambda_x \text{‘} \mu_x)$ dargestellt. Eine VHDL-Bedingung kann in eine oder mehrere kombinatorische Bedingungen übertragen werden, wenn kein Signal der Bedingung in der Sensitivitätsliste des Prozesses aufgeführt wird. Im Listing 4.8 werden beispielsweise die VHDL-Bedingungen in Zeile 4 und 5 in jeweils eine kombinatorische Bedingung (a‘1) und (b‘0) übertragen.

```
1.  sen: process(clk)
2.  begin
3.  if clk = '1' and clk'event then
4.      if a = '1' then
5.        if b = '0' then
6.          seq. Zuweisung x;
7.     [...]
8.  end process;
```

Listing 4.8 *Beispiel für die Ermittlung kombinatorischer Bedingungen.*

Eine sensitive Bedingung β_x ist die logische Verknüpfung von sog. sensitiven Tupeln. Ein sensitives Tupel besteht aus einem Literal κ_x und einem dem Literal zugeordneten Attribut ν_x mit $\nu_x \in (r, f, h, l)$. Sie wird im Folgenden mit der Form $(\kappa_x \circ \nu_x)$ dargestellt. Die Elemente der Wertemenge von ν_x haben dabei folgende Bedeutungen für das im Tupel zugeordnete Literal:

- r: steigende Flanke,
- f: fallende Flanke,
- h: logisch wahr,
- l: logisch falsch.

Eine VHDL-Bedingung kann in eine oder mehrere kombinatorische Bedingungen übertragen werden, wenn alle in der Bedingung vorkommenden Signale entweder in der Sensitivitätsliste des Prozesses erscheinen oder als konstant anzusehen sind (z. B. Listing 4.6 Zeile 3 bis 5).

VHDL-Bedingungen, die keiner der beiden Kategorien angehören (siehe Listing 4.9 Zeile 3), werden (wenn möglich) getrennt. So kann beispielsweise der Ausdruck '`if a and b then`' mit a als sensitivem und b als nichtsensitivem Signal in die sensitive Bedingung '`if a then`' und die nichtsensitive Bedingung '`if b then`' aufgeteilt werden. Die Bestimmung der kombinatorischen Bedingungen ist aus den gegebenen Beschreibungen direkt ablesbar. Im Listing 4.8 werden beispielsweise die Bedingungen aus Zeile 4 und 5 für die Zuweisung x zu (a'1)•(b'0) zusammengefasst.

Für die Bestimmung der sensitiven Bedingungen gelten zusätzliche Regeln:

- Für jedes Signal der Sensitivitätsliste wird zusätzlich zu jeder Zuweisung die Bedingung $(\kappa_x \circ r) + (\kappa_x \circ f)$ in die Liste der sensitiven Bedingungen aufgenommen. Durch diesen Schritt wird die Information des prozessauslösenden Signals in den Syntheseprozess übernommen. Die gleiche Bedingung wird ebenfalls für das VHDL-Attribut `event` sensitiver Signale verwendet.
- Zu jeder Bedingung, in denen ein Pegel abgeprüft wird ($\nu_x \in (h, l)$), wird zusätzlich die Bedingung für die zugehörige Flanke zugeordnet. So wird beispielsweise für die Zeile 4 im Listing 4.5 die Form $(b \circ l) + (b \circ f)$ zu den sensitiven Bedingungen hinzugefügt. Dies ist notwendig, da die Zustände nach VHDL-Standard nicht unabhängig voneinander sind.

Für jede Zuweisung werden die kombinatorischen und sensitiven Bedingungen getrennt mittels boolescher Logik zusammengefasst und für den nächsten Schritt gespeichert.
Für die Zuweisung x im Listing 4.8 ergeben sich folgende Bedingungen:

- sensitiv: $[(clk \circ r) + (clk \circ f)] \bullet [(clk \circ h) + (clk \circ r)] \bullet [(clk \circ r) + (clk \circ f)]$
  wird zu $(clk \circ r)$,
- kombinatorisch: $(a'1) \bullet (b'0)$.

Das Ergebnis der Umformung ist eine geordnete Liste Φ. Ein Eintrag in der Liste besteht aus der Zuweisung ϕ_x und den zugeordneten Bedingungen α_x und β_x. Die Bedingungen können leer sein. Als Beispiel ist die aus dem Listing 4.9 erzeugte Liste Φ in Tabelle 4.1 dargestellt.

```
1.  bsp: process( clk, rst )
2.  begin
3.  if rising_edge(clk) and cs then
4.    if a = '1' then
5.      z <= c;
6.    else
7.      z <= d and e;
8.    end if;
9.  end if;
10. if rst = '0' then
11.   z <= '0';
12. end if;
13. end process;
```

Listing 4.9 *Zu übersetzender VHDL-Quelltext.*

Tabelle 4.1: *Aus Listing 4.9 generierte Liste* Φ

x	ϕ_x	α_x	β_x	
1	c	(cs'1)•(a'1)	(clk°r)	siehe Listing 4.9 Zeile 5
2	d and e	(cs'1)•(a'0)	(clk°r)	siehe Listing 4.9 Zeile 7
3	'0'		(rst°f)+(rst°l) • [(clk°r)+(clk°f)]	siehe Listing 4.9 Zeile 11

4.3.4 Erstellen der Wertetabellen

Ausgehend von der im letzten Abschnitt erzeugten Liste Φ wird in diesem Schritt die eigentliche Simulation durchgeführt. Die durch die Liste Φ repräsentierte Schaltung wird dabei mit allen möglichen Belegungen der Eingangssignale beschaltet und die ermittelten Ausgangswerte in speziellen Wertetabellen zur weiteren Verarbeitung gespeichert. Durch diesen Schritt geht der in der Ausgangsliste noch vorhandene sequentielle Charakter der Repräsentation verloren. In den durch die Simulation generierten Wertetabellen werden automatisch charakteristische Strukturinformationen erzeugt, die im nachfolgenden Schritt entschlüsselt werden können, um die eigentliche Schaltung zu generieren.

Anhand der Menge aller in Φ vorkommenden sensitiven Literale (κ_0 - κ_n) wird als erste Wertetabelle die sog. Haupttabelle erstellt.

Die Haupttabelle enthält alle möglichen Zustände der sensitiven Literale. Die letzte Spalte enthält einen von drei möglichen Einträgen, der ein entsprechendes Verhalten der Schaltung unter der in der Zeile beschriebenen Bedingung spezifiziert. Die Einträge können sein:

- das Symbol Ω, welches bedeutet, dass kein Zustandswechsel stattfindet (speichern),
- eine Zuweisung ϕ_x, die einen entsprechenden Zustandswechsel symbolisiert,
- der Verweis auf eine Untertabelle mit den zusätzlich zu erfüllenden kombinatorischen Bedingungen.

Im initialen Zustand ist zunächst in allen Zellen der letzten Spalte das Symbol Ω eingetragen. Nacheinander werden alle Einträge von Φ entsprechend der jeweils aktuellen sensitiven Bedingung β_x in die Wertetabelle übertragen. Ist die jeweils zugehörige kombinatorische Bedingung α_x leer, werden alle Werte der letzten Spalte entsprechend dem aktuellen ϕ_x gesetzt. Ist die aktuelle kombinatorische Bedingung α_x nicht leer, so wird eine neue Untertabelle nach dem Verfahren für die Haupttabelle mit den kombinatorischen Literalen (λ_0 - λ_n) erstellt. Existiert diese Tabelle bereits, wird nur ϕ_x entsprechend der durch α_x beschriebenen Funktion eingetragen. Alte Werte werden analog der Haupttabelle durch die neuen Werte überschrieben. Zur Vereinfachung der weiteren Schritte wird entsprechend zum ersten Tabellensatz ein zweiter erstellt. Die Haupttabelle wird hier anhand der kombinatorischen, die zugehörigen Untertabellen anhand der sensitiven Literale erstellt. Ist α_x leer, werden alle Zeilen der Haupttabelle entsprechend β_x und ϕ_x modifiziert. Mit der aus Listing 4.9 erzeugten Liste Φ ergeben sich Tabelle 4.2 als Haupttabelle der sensitiven Literale und Tabelle 4.3 als einzige Untertabelle für die kombinatorischen Literale.

Tabelle 4.2: *Sensitive Haupttabelle von Listing 4.9*

rst	clk	Funktion/ komb. Tabelle
l	l	Ω
l	h	Ω
l	r	'0'
l	f	'0'
h	l	Ω
h	h	Ω
h	r	⇒ *Tabelle 4.3*
h	f	Ω
r	l	Ω
r	h	Ω
r	r	⇒ *Tabelle 4.3*
r	f	Ω
f	l	'0'
f	h	'0'
f	r	'0'
f	f	'0'

Tabelle 4.3: *Zu Tabelle 4.2 zugehörige kombinatorische Untertabelle*

cs	a	Funktion
0	0	Ω
0	1	Ω
1	0	d and e
1	1	c

Auf die Darstellung der kombinatorischen Haupttabelle und zugehörigen sensitiven Untertabellen wird verzichtet, da sie auch für einfache Beispiele sehr komplex und unübersichtlich werden.

4.3.5 Bestimmung des Schaltungstyps und der Steuersignale

Anhand der im letzten Abschnitt generierten Tabellen werden aufgrund charakteristischer Muster, die im Folgenden beschrieben sind, alle ausgezeichneten Signale identifiziert. Diese erlauben dann den Rückschluss auf den zugrunde liegenden Schaltungstyp und dessen Komponenten. Mit diesen Informationen kann eine vorläufige Schaltung erstellt werden, die im nächsten Schritt überprüft und angepasst wird.

Folgende Schaltungselemente können identifiziert werden:

- sensitive bzw. kombinatorische Reset- und Set-Signale,
- sensitive bzw. kombinatorische Enable-Signale,
- Taktsignale (Single- und Dual-Edge),
- kombinatorischer Rest.

Die Zuordnung der Literale zu den einzelnen Schaltungselementen wird anhand von charakteristischen Mustern in der Tabelle vorgenommen. So wird beispielsweise für die Suche nach einem sensitiven Set-Resetsignal für jedes sensitive Literal geprüft, ob für einen der beiden möglichen Pegel alle Funktionseinträge identisch und konstant sind. Funktionseinträge, die dabei ausschließlich von statischen Zuständen der Literale abhängen, bleiben hierbei unberücksichtigt. Sind die Funktionseinträge für den entgegengesetzten Pegel mit den Funktionseinträgen des zugeordneten Pegels bei jeweils gleichem Attribut aller übrigen Literale der Tabelle identisch, ist das aktuelle Signal ein asynchrones Set- bzw. Resetsignal. Je nach Wert der Konstante beim Resetpegel wird das Signal in die Liste der asynchronen Set- bzw. Resetsignale aufgenommen.
In Tabelle 4.4 wird beispielsweise geprüft, ob `rst` ein Set-Resetsignal ist. Im dargestellten Fall sind die Funktionseinträge für den Pegel (rst° l) alle '0', also identisch und konstant. Die Funktionseinträge für den entgegengesetzen Pegel (rst° h) sind für jeweils gleiche Attribute des übrigen Literals `clk` identisch (Ω für (clk° l), Ω für (clk° h), *Tabelle 4.3* für (clk° r) und Ω für (clk° f)). Anhand der Konstante beim Resetpegel (rst° l) entscheidet sich der Typ des Set-Resetsignals. Im gegebenen Fall ist die Konstante '0', also ist `rst` ein asynchrones Resetsignal.

Wurde ein Resetsignal gefunden, wird das Literal aus allen Tabellen gelöscht und nach weiteren Set-Resetsignalen gesucht. Im Beispiel entsteht nach dem Löschen des Signals `rst` aus Tabelle 4.4 eine neue sensitive Haupttabelle: Tabelle 4.5. Die kombinatorische Untertabelle Tabelle 4.3 bleibt unverändert erhalten. Analog wird auch in der kombinatorischen Haupttabelle gesucht. Gefundene Set-Resetsignale werden in die Liste der synchronen Set- bzw. Resetsignale übernommen. Es sind beliebig viele Set-Resetsignale pro Unterprozess möglich.

Tabelle 4.4: *Sensitive Haupttabelle; Muster für asynchrones Reset*

rst	clk	Funktion/ komb. Tabelle
l	l	Ω
l	h	Ω
l	r	’0’
l	f	’0’
h	l	Ω
h	h	Ω
h	r	⇒ *Tabelle 4.3*
h	f	Ω
r	l	Ω
r	h	Ω
r	r	⇒ *Tabelle 4.3*
r	f	Ω
f	l	’0’
f	h	’0’
f	r	’0’
f	f	’0’

konstante Funktion bei konstantem Pegel für aktuelles Literal (*rst*)

jeweils gleiche Funktion bei gleichen Attributen aller anderen Literale (hier nur *clk*)

Aktuelles Literal

Tabelle 4.5: *Sensitive Haupttabelle nach dem Löschen von rst*

clk	Funktion/ komb. Tabelle
l	Ω
h	Ω
r	⇒ *Tabelle 4.3*
f	Ω

Wie bei der Suche nach Set-Resetsignalen existieren entsprechende Muster für die Erkennung der Enable-Signale und des Taktes. Bei den Enable-Signalen wird ebenso wie bei den Set-Resetsignalen getrennt nach synchronen und asynchronen Signalen gesucht. Die Suche wird abgebrochen, wenn in keiner Tabelle mehr ein Ω vorhanden ist oder kein ausgezeichnetes Signal mehr gefunden wird. Die finale sensitive Haupttabelle nach der Identifikation des Taktes ist in Tabelle 4.6 dargestellt, die zugehörige kombinatorische Untertabelle in Tabelle 4.7.

Tabelle 4.6: *Sensitive Haupttabelle nach Eliminierung des Signals clk*

Funktion/ komb. Tabelle
⇒ *Tabelle 4.7*

Tabelle 4.7: *Zu Tabelle 4.6 zugehörige kombinatorische Untertabelle*

a	Funktion
0	d and e
1	c

4.3.6 Erstellen der Schaltung

Ist in den finalen Tabellen noch ein Ω vorhanden, lässt sich das gegebene Problem nicht mit den verfügbaren Schaltungsformen darstellen. Die gegebene VHDL-Beschreibung muss in dem Fall abgelehnt werden bzw. es muss auf alternative Syntheseverfahren ausgewichen werden. Beispiele entsprechender nicht synthesefähige Schaltungen liegen zur Zeit nicht vor.

Ist in den finalen Tabellen kein Ω mehr vorhanden, werden die finalen Tabellen in eine kombinatorische Schaltung umgewandelt. Dazu werden Zeilen der sensitiven Haupttabelle, die von dynamischen Zuständen abhängen, gelöscht (in Tabelle 4.8 grau hinterlegt). Es entsteht eine einfache, kombinatorische Tabelle, welche mit den kombinatorischen Untertabellen zu einer einzigen kombinatorischen Tabelle verschmolzen wird. Im Beispiel wird Tabelle 4.8 mit dazugehöriger Tabelle 4.9 zu Tabelle 4.10 zusammengefasst.

Tabelle 4.8: *Sensitive Haupttabelle nach Bestimmung der Steuersignale*

b	Funktion/ komb. Tabelle
l	⇒ *Tabelle 4.9*
h	’1’
r	⇒ *Tabelle 4.9*
f	’0’

Tabelle 4.9: *Zu Tabelle 4.8 zugehörige kombinatorische Untertabelle*

a	Funktion
0	d and e
1	c

Tabelle 4.10: *Aus Tabelle 4.8 und Tabelle 4.9 zusammengefasste kombinatorische Tabelle*

a	b	Funktion
0	0	d and e
0	1	’1’
1	0	c
1	1	’1’

Eine Optimierung der resultierenden Tabelle (wie z. B. eine Eliminierung für die Schaltung irrelevanter Eingangsvariablen) ist möglich, jedoch in der Regel nicht notwendig. Die verschmolzene Tabelle wird mit den Mitteln des logischen Entwurfs in eine kombinatorische Schaltung umgesetzt (siehe Abbildung 4.2).

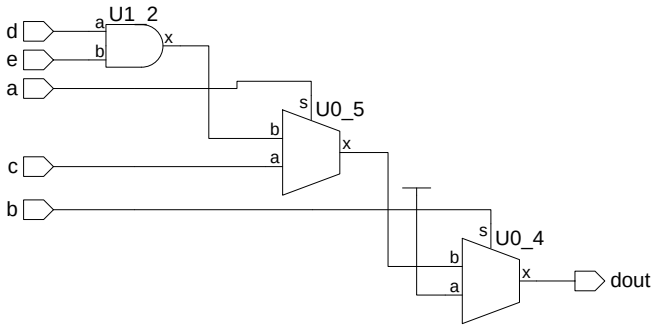


Abbildung 4.2 *Aus Tabelle 4.10 generierte kombinatorische Schaltung.*

Abhängig von den im vorhergehenden Schritt identifizierten Steuersignalen wird der Schaltungstyp bestimmt. Wurde ein Taktsignal identifiziert, handelt es sich um ein flankengetriggertes Schaltwerk. Es wird ein Flipflop instantiiert, dessen Dateneingang mit dem Ausgang des kombinatorischen Netzes beschaltet wird. Abhängig von den übrigen identifizierten Steuersignalen werden für das Flipflop entsprechende Enable bzw. Resetsignale erzeugt. Wurden mehrere Enable oder Resetsignale identifiziert, werden diese zu jeweils einem einzigen Signal kombinatorisch verknüpft.

Wurde kein Taktsignal gefunden, werden die ermittelten Enable-Signale kombinatorisch verknüpft und mit dem Enable-Eingang eines zu instantiierenden Latches beschaltet. Der Dateneingang des Latches wird analog zum Flipflop mit dem Ausgang des kombinatorischen Netzes beschaltet. Die optionalen Set- und Reset-Eingänge des Latches werden entsprechend der im Abschnitt 4.3.5 ermittelten Set- und Resetsignale beschaltet.

Wurden keine Steuersignale identifiziert, wird das ermittelte kombinatorische Netz unverändert übernommen. In Abbildung 4.4 ist das Synopsys-Syntheseergebnis des Listing 4.9 aufgeführt. Als einziges Eingangssignal wird in der Synopsys-Schaltung das Signal `rst` verwendet. In dieser Form ist die Schaltung nicht zu verwerten. In Abbildung 4.3 ist das entsprechende fehlerfreie Square-Dance-Syntheseergebnis abgebildet.

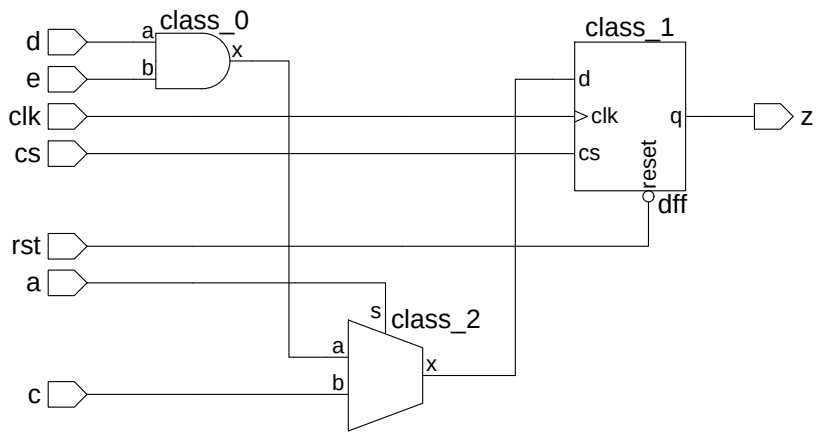


Abbildung 4.3 *Logikdiagramm des Square-Dance-Syntheseergebnisses von Listing 4.9.*

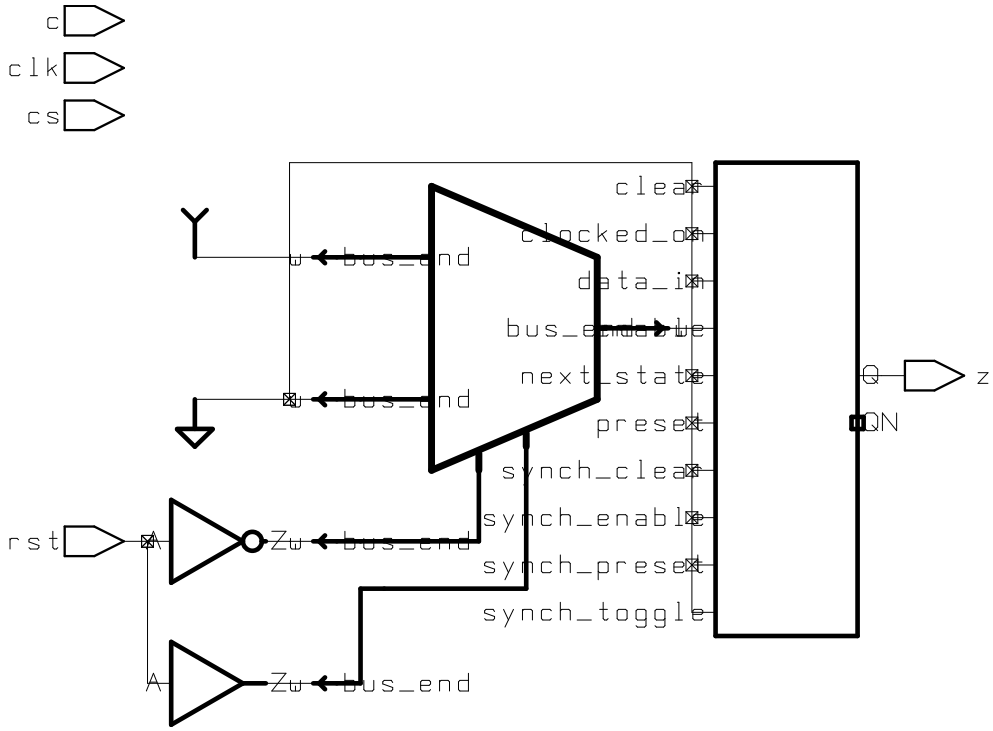


Abbildung 4.4 *Logikdiagramm des Synopsys-Syntheseergebnisses von Listing 4.9.*

4.3.7 Abschließende Überprüfungen und Optimierungen

Im letzten Schritt werden die synthetisierten Teilschaltungen der im Abschnitt 4.3.2 generierten Unterprozesse auf Konsistenz überprüft und zu einer einzigen Schaltung zusammengefügt.

Die einzelnen Unterschaltungen gehören, auch wenn sie unabhängig voneinander synthetisiert werden, funktional zusammen. Es wird überprüft, wie weit die synthetisierten Schaltungen voneinander abhängig sind, d. h. welche generierten Ausgangssignale in anderen Unterschaltungen als Eingangssignale weiterverwendet werden (siehe Abbildung 4.5). Werden solche Netze gefunden und sind die vorhandenen Speicherelemente mit denselben Steuersignalen beschaltet, können die Speicherelemente zu Registern zusammengefasst werden. Auf diese Weise werden die im Abschnitt 4.3.2 voneinander getrennten Elemente von Signalvektoren wieder zusammengefügt. Die ursprüngliche Reihenfolge der Bits im zu instantiierenden Register geht durch diesen Schritt verloren, was in Hinsicht auf ein optimales Mapping jedoch bedeutungslos ist. Tatsächlich werden für die als logisch zusammengehörig ermittelten Speicherelemente keine Register erzeugt, sondern allein die Möglichkeit, dass dieses an dieser Stelle sinnvoll ist, für die nachfolgende Technologieabbildung vorgemerkt.

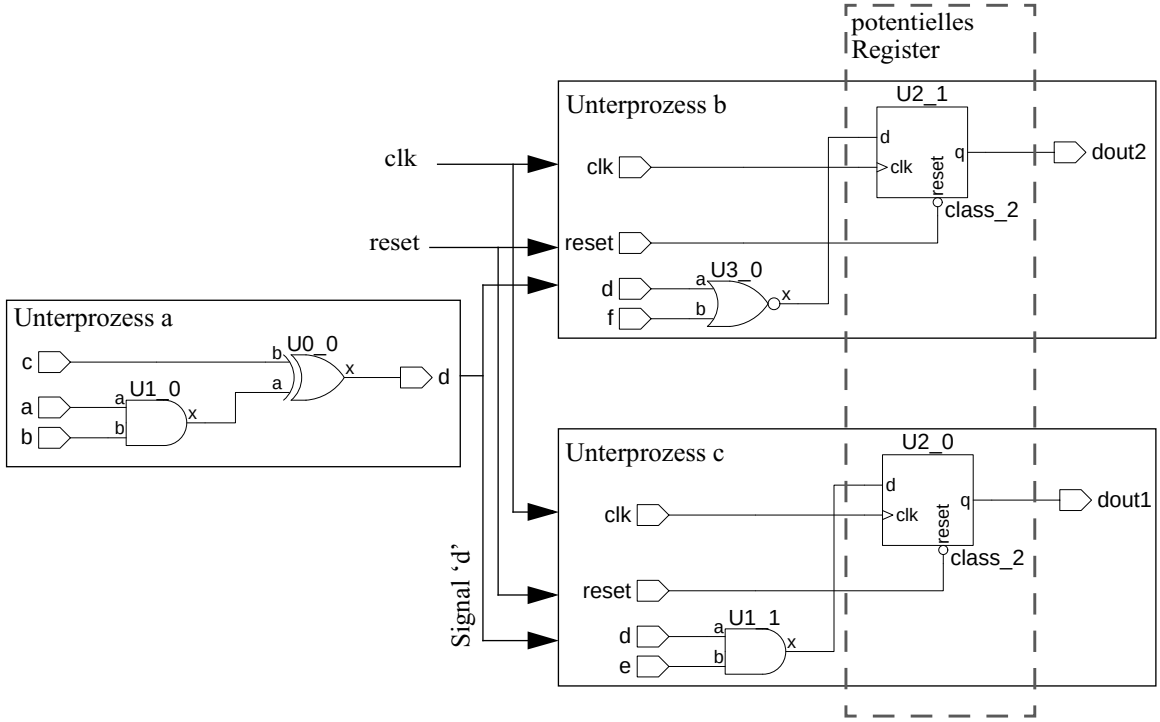


Abbildung 4.5 *Identifikation von Registern beim Zusammenführen der Unterschaltungen.*

Als letzter Schritt wird die generierte Schaltung mit der Ausgangsbeschreibung verglichen. Es wird geprüft, ob die gegebene VHDL-Beschreibung eine gültige Beschreibung für das synthetisierte Netz darstellt. So müssen beispielsweise alle Eingangssignale eines pegelgetriggerten Schaltwerkes oder eines Schaltnetzes in der Sensitivitätsliste des Prozesses aufgeführt sein bzw. alle nicht in der Sensitivitätsliste aufgeführten Eingangssignale unbedingt von den sensitiven Signalen abhängen. In der vorliegenden Implementierung werden nur die bis zu diesem Punkt synthetisierten Netze auf eine entsprechende Abhängigkeit überprüft. Genaugenommen ist dieses Vorgehen nicht ausreichend, da die entsprechenden Abhängigkeiten erst bei der weiteren Synthese durch hinzukommende Schaltungselemente hergestellt werden könnten. Entsprechende fehlerhafte Beispielbeschreibungen konnten jedoch nicht gefunden werden, weshalb in diesem Fall auf die Implementierung weitergehender Synthesetechniken verzichtet wurde und ggf. eine entsprechende Fehlermeldung ausgegeben wird.

4.3.8 Prozesssynthese bei umfangreichen Sensitivitätslisten

Die Länge der beim Syntheseprozess erzeugten sensitiven Haupttabelle hängt direkt von der Anzahl der für die Kodierung der in der Sensitivitätsliste aufgeführten Signale benötigten Signalleitungen ab. Bei der Beschreibung von komplexen, rein kombinatorischen Prozessen, z. B. solchen, in denen sehr breite Vektoren verknüpft werden, können die erzeugten Tabellen dementsprechend sehr speicherintensiv werden, da in kombinatorischen Prozessen grundsätzlich alle verwendeten Signale aufgeführt werden müssen. Bei Beschreibungen von Schaltwerken hingegen hat die Aufführung bestimmter Signale in der

Sensitivitätsliste keinen Einfluss auf die Funktionalität. Bei gleicher Funktion werden jedoch auch hier bei ungünstiger Prozessbeschreibung die entsprechenden Wertetabellen sehr umfangreich.

Dieses Problem lässt sich dadurch beheben, dass nur Signale in die sensitiven Tabelle aufgenommen werden, für die auch im entsprechenden VHDL-Prozess mindestens eine Bedingung existiert, in der das fragliche Signal ausgewertet wird. Hierdurch gehen die durch die Sensitivitätsliste mitgeführten impliziten Bedingungen verloren, was jedoch erst dann zur Wirkung kommt, wenn mit dem hier vorgestellten Verfahren keine Lösung gefunden werden kann. So ist es beispielsweise bei einem flankengetriggerten Schaltwerk unerheblich, ob die Signale des kombinatorischen Teils sensitiv sind oder nicht. Eine weitere Möglichkeit der Reduktion ist, für komplexe Bedingungen (z. B. ein Vergleich zweier Vektoren) ein einzelnes stellvertretendes Literal für die gesamte Bedingung zu generieren. Diese Technik wird grundsätzlich auch bei der Synthese von Zustandsmaschinen ("`select ... with`" bzw. "`case ... is`" Statements) eingesetzt. Hier erscheinen in den Tabellen nur die minimierten Zustandsvariablen.

4.3.9 Generalisierung

Ein generalisiertes Syntheseverfahren erlaubt die Synthese einer gegebenen Beschreibung auch dann, wenn die Beschreibung von den konventionellen Verfahren abgelehnt wird. Bei diesem Verfahren wird die Struktur einer sequentiellen VHDL-Beschreibung auf Gatterebene nachgebildet. Der Schaltungsaufwand und die erzielten Laufzeiten der synthetisierten Schaltungen steigen bei diesem Vorgehen überproportional im Vergleich zu den konventionellen Verfahren. Dadurch bedingt ist der Einsatz des generalisierten Verfahrens nur dann sinnvoll, wenn die alternativen Syntheseverfahren keine Lösung finden. Die in diesem Fall gegebenen Beschreibungen sind dann in der Regel Schaltwerke, in denen ausgezeichnete Signale wie Takt, Reset oder Enable nicht vorhanden bzw. nicht als solche zu identifizieren sind. Entsprechend ist die Synthese dann nur als asynchrones Schaltwerk möglich.

Jede nichtkombinatorische sequentielle VHDL-Beschreibung hat die in Listing 4.10 dargestellte Struktur bzw. ist in eine oder mehrere Beschreibungen dieser Struktur transformierbar. Die sensitiven Signale sind mit $\{s_0, s_1, ... s_n\} \in S$ bezeichnet und die Bedingungen mit $\{b_0, b_1, ... b_m\} \in B$. Diese Bedingungen sind sensitiv, hängen dementsprechend jeweils von mindestens einem sensitiven Signal ab. Nichtsensitive Bedingungen der Ausgangsbeschreibung werden mit den von ihnen abhängigen Transitionen kombinatorisch verknüpft, werden so als Bestandteil der Transitionssignal-erzeugenden kombinatorischen Netze $\{c_0, c_1, ... c_m\} \in C$ mit berücksichtigt, ohne gesondert behandelt werden zu müssen. Sensitive Bedingungen, die mehr als eine Signalflanke beschreiben, werden in mehrere Bedingungen aufgespalten, so dass jede Bedingung genau eine Signalflanke beschreibt. Die prinzipielle Grundstruktur der erzeugten Schaltungen ist in Abbildung 4.6 dargestellt.

.

```
1.  p0: process( s0, s1, ... ,sn )
2.  begin
3.    if b0 then
4.      t <- c0;
5.    end if;
6.    if b1 then
7.      t <- c1;
8.    end if;
9.    [...]
10.   if bm then
11.     t <- cm;
12.   end if;
13. end process p0;
```

Listing 4.10 *Grundstruktur eines nichtkombinatorischen Prozesses.*

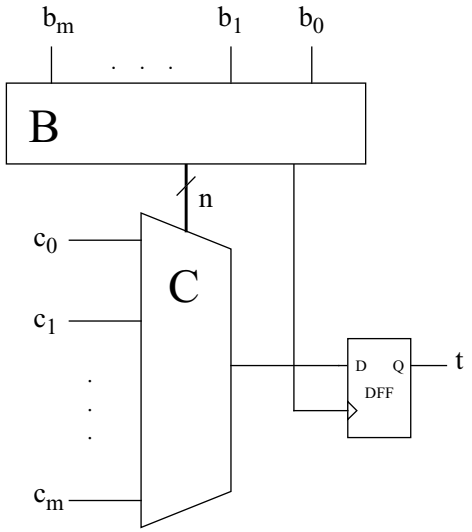


Abbildung 4.6 *Prinzipielle Schaltungsstruktur von Prozessen beim generalisierten Verfahren.*

Abhängig von der letzten gültigen Signalflanke der Eingangssignale b_1, b_2, ... b_m wird durch den Schaltungsblock B eine Adresse erzeugt, die das zu b_x entsprechende Signal c_x durchschaltet. Das Ausgangssignal muss gepuffert werden, da Signaländerungen am Ausgang nur in Abhängigkeit der Signale b_1, b_2, ... b_m zulässig sind. In der dargestellten Form ist die Schaltungsstruktur nur für den technologieabhängigen Entwurf zu verwerten. Das Taktsignal für das Ausgangs-Flipflop muss zeitlich verzögert zur Adressgenerierung für den Multiplexer C erfolgen, was ohne Kenntnisse der Zieltechnologie nicht befriedigend realisiert werden kann. Entsprechend ist die Grundstruktur zu modifizieren.

Aufgabe des Flankendecoders (Block B in Abbildung 4.6) ist es, die am Eingang auftretenden Signalflanken in kombinatorisch weiterverarbeitbare Pegel umzusetzen. Über zeitliche Randparameter oder die Reihenfolge der am Eingang auftretenden Signalflanken kann dabei keine Aussage getroffen werden. Als Basis für den Decoder wird das Flanken-Reset-Flipflop (im Folgenden FR-FF, siehe Abbildung 4.7) verwendet [27]. Das FR-FF hat ein ähnliches Verhalten wie ein RS-Flipflop. Im Unterschied zu diesem wird der Zustand des Flipflops nicht durch an den Eingängen anliegende Pegel gesteuert, sondern ausschließlich durch Signalflanken. Eine positive Signalflanke am Eingang S (Set) setzt das Flipflop unabhängig vom anliegenden Signalpegel am Eingang C; eine positive Signalflanke am Eingang C (Clear) löscht das Flipflop unabhängig vom anliegenden Signalpegel am Eingang S (siehe Abbildung 4.8) Eine mögliche Realisierung des FR-FF auf Basis von konventionellen Bausteinen ist in Abbildung 4.9 aufgeführt.

Der Zustand des Flipflops ist intern in einem RS-Flipflop gespeichert. Die zum Setzen und Löschen des Zustandswertes nötigen Pegel werden durch zwei unabhängige D-Flipflops generiert, wenn am jeweiligen Takteingang eine steigende Signalflanke registriert wird.

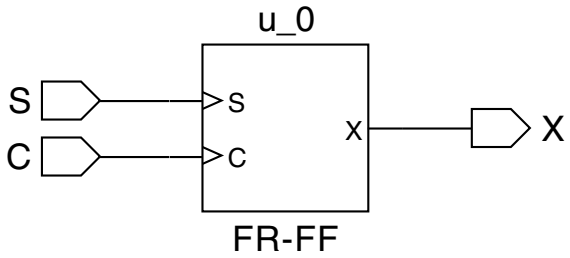


Abbildung 4.7 *Gattersymbol des FR-FF.*

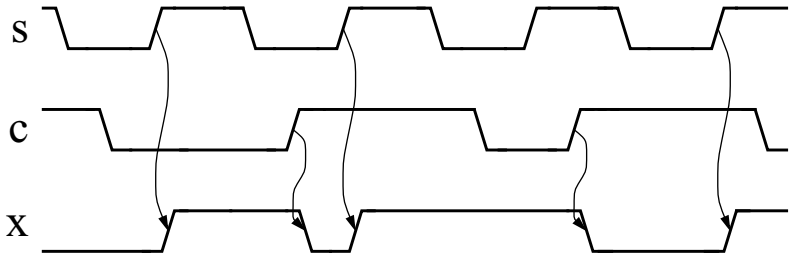


Abbildung 4.8 *Impulsplan für einen typischen Signalverlauf am FR-FF.*

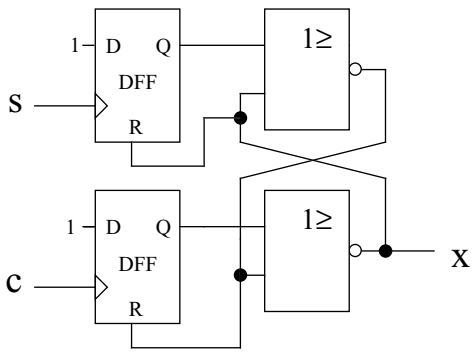


Abbildung 4.9 *Implementierungsvariante des FR-FF.*

Nach erfolgtem Setzen bzw. Löschen des RS-Flipflop wird das auslösende D-Flipflop zurückgesetzt. Als Resetsignal wird der jeweils im RS-Flipflop rückgekoppelte Signalpfad verwendet. Auf diese Weise wird sichergestellt, dass das Reset am D-Flipflop erst dann ausgelöst wird, wenn das RS-Flipflop in einem stabilen Zustand ist. In der dargestellten Form eignet sich die Schaltung auch für die Implementierung mit hazarderzeugenden Gattertechnologien, wie sie z. B. in Form von Look-Up-Tables in FPGAs Verwendung finden.

In Listing 4.11 ist ein Beispiel für einen nichtkombinatorischen Prozess und der dazugehörige Flankendecoder (Abbildung 4.10) gegeben. Für jede im Prozess aufgeführte Bedingung wird ein eigenes modifiziertes FR-FF instantiiert.

```
1.  p0: process( a, b )
2.  begin
3.    if  rising_edge( a ) then t <- out0;
4.    end if;
5.    if falling_edge( a ) then t <- out1;
6.    end if;
7.    if  rising_edge( b ) then t <- out2;
8.    end if;
9.    if falling_edge( b ) then t <- out3;
10.   end if;
11. end process p0;
```

Listing 4.11 *Beispielprozess für das generalisierte Syntheseverfahren von VHDL-Beschreibungen.*

Die positive Flanke am D-FF eines Blockes (gestrichelt in Abbildung 4.10 dargestellt) generiert einen positiven Pegel am Set-Eingang des zugehörigen RS-FF. Dieser Pegel setzt das RS-FF. Die Rückkopplung des RS-FF wird erst aktiv, wenn die RS-FF in den anderen Blöcken erfolgreich gelöscht wurden. Ist die Rückkopplung aktiv, wird durch diese das auslösende D-FF gelöscht. Als Vereinfachung zum unmodifizierten FR-FF kann auf das Clear D-FF verzichtet werden. Zum Löschen des RS-FF werden die Pegel der Rückkopplungen der RS-FF in den weiteren Blöcken verwendet. Als weiterer Unterschied zum unmodifizierten FR-FF tastet das D-FF den negativen Datenausgang am zugehörigen RS-FF ab. Dies verhindert, dass der Block bei zu schnellen Pegelwechseln am Eingang in einen unzulässigen Zustand übergeht. Aus gleichem Grund ist im Gegensatz zum spezifizierten Ausgangsprozess eine zusätzliche Resetleitung notwendig, um das Schaltwerk zu Beginn in einen zulässigen Zustand zu versetzen.

Neben der steigenden Komplexität für jede hinzukommende Bedingung nimmt auch die Laufzeit zu, da für die Bestimmung des aktuellen Zustandes jedes Blockes die Zustände aller anderen Blöcke berücksichtigt werden müssen. Das Setzen eines Blockes ist nur dann möglich, wenn alle Blöcke gelöscht sind.

In Abbildung 4.11 ist ein Simulationsergebnis der Realisierung der Schaltung im ce81-Prozess von Fujitsu dargestellt. Der Eingang a ist mit einem Takt von 20 Mhz, der Eingang b mit 20,8 Mhz beschaltet. Die Signalausgänge zeigen die jeweils letzte aufgetretene Signalflanke der beiden Eingänge an. Ein High-Pegel an einem Ausgang kann beim Auftreten der entsprechenden Flanke erst dann erzeugt werden, wenn der Pegel des Ausgangssignals der zeitlich davorliegenden Flanke gelöscht wurde. Folgen die auftretenden Flanken an den Eingängen zu schnell aufeinander (im Beispiel t<=3ns), ist keine Differenzierung der Flanken voneinander möglich, und es werden keine entsprechenden Pegel am Ausgang generiert. Die Ausgangssignale werden erst dann wieder generiert, wenn die Auflösungszeit überschritten wird. Entsprechend der Schaltung erhöhen zusätzlich zu detektierende Flanken diese Auflösungszeit.

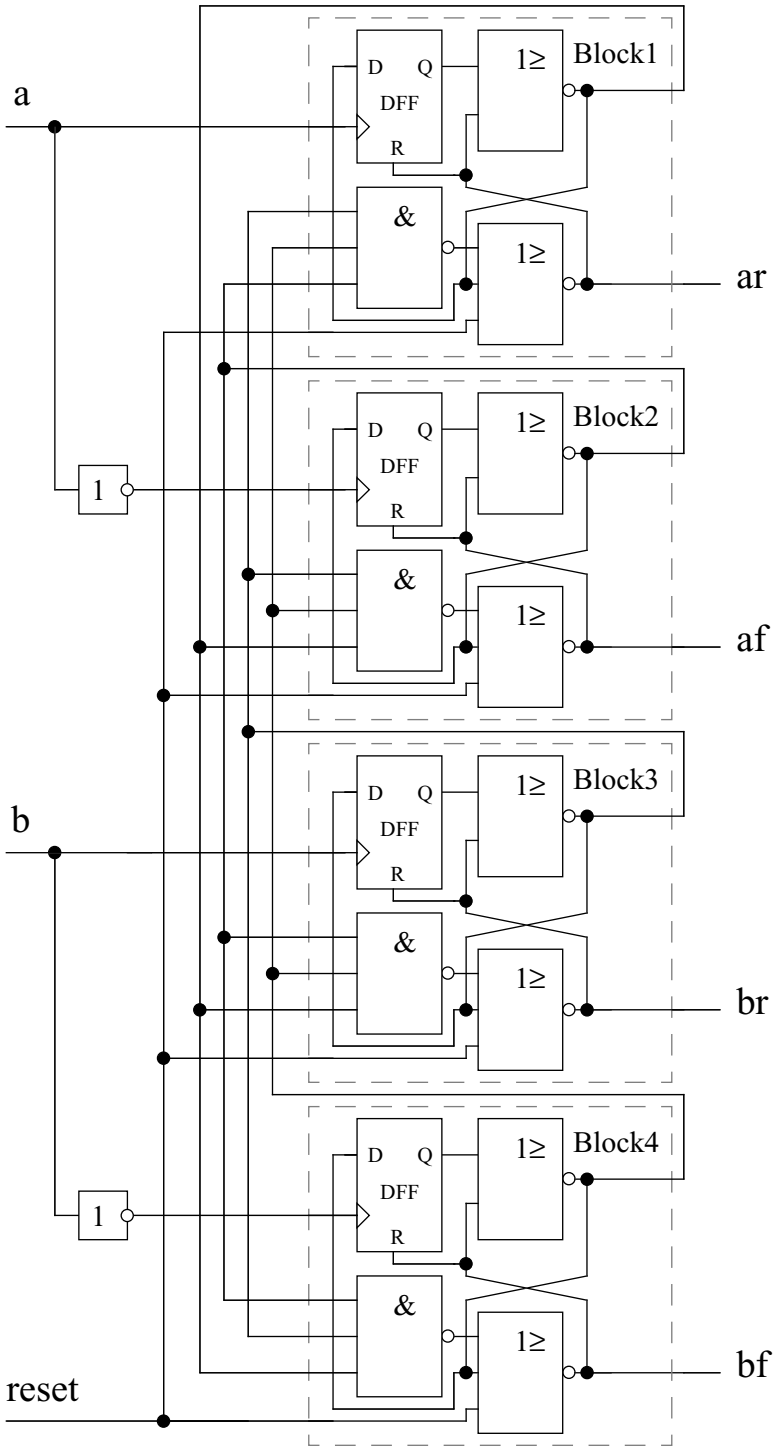


Abbildung 4.10 *Flankendecoder für den im Listing 4.11 dargestellten Prozess.*

Die Ausgangsstufe C erfüllt die Funktion des Multiplexers C und des nachgeschalteten D-FF im Abbildung 4.6. Die Aufgabe ist es, in Abhängigkeit der aktuell erfüllten Bedingungen die entsprechenden Zuweisungen auszuführen, also den Ausgang des der Zuweisung entsprechenden Schaltnetzes auf den Ausgang der Gesamtschaltung durchzuschalten. Bis zur nächsten Änderung der Bedingungen muss der Ausgangspegel konstant gehalten werden. Eine diese Funktionalität erfüllende Schaltung kann sehr einfach realisiert werden, da

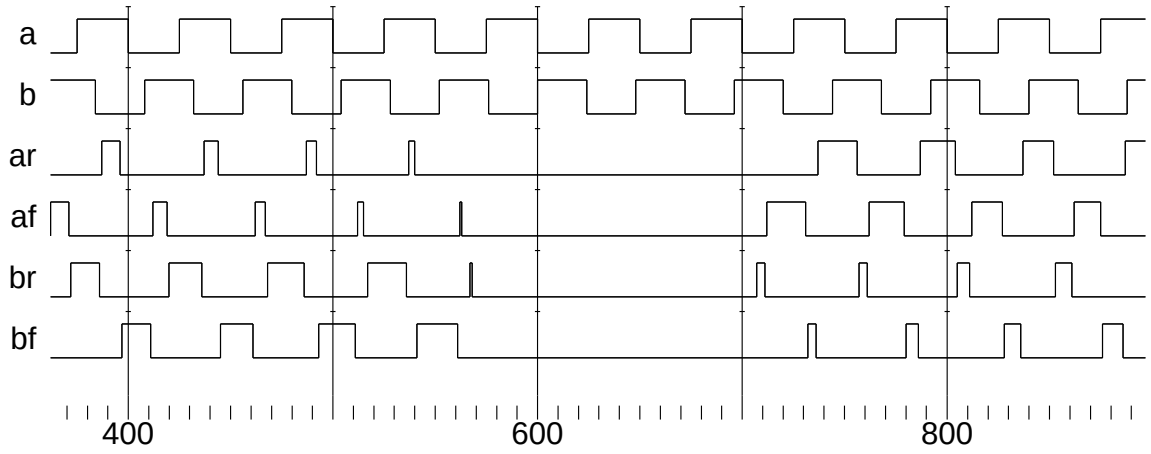


Abbildung 4.11 *Timingsimulation der Realisierung des in Abbildung 4.10 dargestellten Flankendecoders für den ce81-Fujitsu-Prozess.*

die vom Flankendecoder bestimmte gültige Bedingung im 1-aus-n-Code zur Verfügung gestellt wird und es schaltungsbedingt nicht vorkommen kann, dass an mehr als einem Ausgang des Flankendecoders ein High-Pegel anliegt.

In der in Abbildung 4.12 dargestellten Realisierung ist für jede Bedingung bzw. Zuweisung ein eigener Schaltungsblock instantiiert. Die Schaltungsblöcke sind ausgangsseitig über einen Bus verbunden, über welchen das endgültige Ausgangsdatum übertragen wird. Innerhalb eines Blockes wird über ein Latch sichergestellt, dass bei gültiger Bedingung das Zuweisungsdatum am Ausgang konstant gehalten wird. Ist die Bedingung gültig, wird das Zuweisungsdatum über den Tristatetreiber auf den Ausgangsbus übertragen.

Das generalisierte Syntheseverfahren wurde beispielhaft für kleinere Schaltungen getestet (siehe Anhang B.4 auf Seite 133). Insbesondere bei der Realisierung der Ausgangsstufe bieten sich Modifikationen an, da unter bestimmten Bedingungen Hazards am Ausgang auftreten können. Bedingt durch die Eigenschaften des Flankendecoders kann auch der Bus am Ausgang zeitweise ungetrieben sein, was zu entsprechenden Effekten führt. Eine Erhöhung des Auflösungsvermögens ist mit zusätzlichem Hardwareaufwand ebenfalls denkbar.

Im Gegensatz zu den generierten Schaltungen konventioneller Werkzeuge, stimmen die durch das generalisierte Verfahren generierten Schaltungen funktional mit den Ausgangsbeschreibungen überein. Aufgrund des sehr hohen Ressourcenbedarfs in Bezug auf die zu erfüllende Funktionalität ist es jedoch nur für wenige Spezialfälle interessant. An dieser Stelle sollte grundsätzlich auf die genaue Synthese verzichtet werden und die Ausgangsbeschreibung überarbeitet werden.

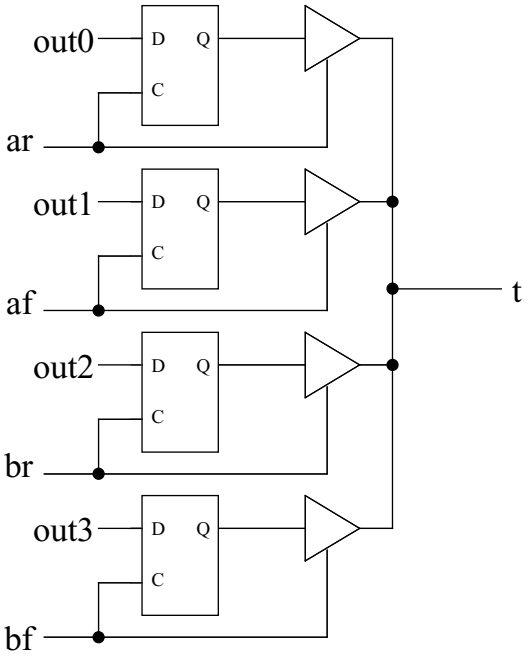


Abbildung 4.12 *Beispiel für eine Realisierung der Ausgangsstufe.*

4.4 Synthese paralleler VHDL-Beschreibungen

Die Synthese paralleler VHDL-Beschreibungen ist für die Synthese der nichtsequentiell beschriebenen Schaltungsteile zuständig. Alle VHDL-Anweisungen innerhalb einer Architektur, die nicht innerhalb eines Prozedur- Funktions- oder Prozessblockes stehen, werden innerhalb einer VHDL-Simulation als nebenläufige Anweisungen behandelt, sie sind also in jeweils parallel laufende Schaltungsteile zu transformieren. Im Gegensatz zur Synthese sequentieller VHDL-Beschreibungen ist die Synthese paralleler VHDL-Beschreibungen unproblematisch. Die Reihenfolge von VHDL-Anweisungen hat keine Bedeutung. Die Abhängigkeiten der einzelnen Anweisungen voneinander werden allein durch die Verwendung desselben Signals in diesen realisiert.

4.4.1 Synthese logischer Strukturen

Die Synthese logischer Strukturen erfolgt im Logiksyntheseverfahren SibaS linear zur gegebenen Hardwarebeschreibung. Für jede abgeschlossene Zeile der Beschreibung wird die verwendete Sprachkonstruktion ermittelt und abhängig von dieser eine entsprechende Schaltung instantiiert. Abhängigkeiten der Zeilen untereinander, die durch die Verwendung gleicher Signalnamen realisiert sind, werden in der generierten Schaltung durch entsprechende Signalleitungen hergestellt.

Für jeden Signalnamen werden abhängig vom Signaltyp eine oder mehrere Signalleitungen vorgesehen. Hierzu wird der Signaltyp so weit zergliedert, dass nur noch duale Typen übrig bleiben, welchen für jedes Signal des Typs eine eigene Signalleitung zugewiesen wird.

Beschreibungen, die komplexe Signale verwenden, werden entsprechend der Anzahl der benötigten Signalleitungen für die Realisierung des komplexen Signals mehrfach instantiiert. Die Verarbeitung des komplexen Signals erfolgt dementsprechend parallel. Im Listing 4.12 ist beispielsweise in den Zeilen 15 bis 16 eine `with-select`-Anweisung gegeben, welche von der Synthese als Beschreibung für einen Multiplexer erkannt wird. Die zu ver-

```
1.  type states is ( s0, s1, s2 );
2.  type com_ty is
3.    record
4.      state: states;
5.      count: integer range 0 to 3;
6.    end record;
7.
8.       [...]
9.
10. signal a, b, x: com_ty;
11. signal sel: bit;
12.
13.      [...]
14.
15. with sel select x <= a when '0',b when others;
```

Listing 4.12 *Beispiel für die Verwendung komplexer Signaltypen.*

schaltenden Signale `a` und `b` sind dabei komplexe Signale, deren Aufbau als Typdefinition in den Zeilen 1 bis 6 zu finden ist. Dieser Typ lässt sich rekursiv bis auf vier Bit auflösen. Im Listing 4.13, Zeilen 1 bis 3, sind dementsprechend für jedes Signal des Typs vier Signalleitungen vorgesehen. In den Zeilen 6 bis 13 von Listing 4.13 wird dann für die gewünschte Verschaltung der zweimal vier Bit für jeweils ein Bit die Ausgangsanweisung wiederholt. In der dann gegebenen Form ist keine weitere Auflösung möglich und es werden entsprechend den Anweisungen, wie in Abbildung 4.13 dargestellt, vier Multiplexer instantiiert.

Viele Optimierungsschritte, die in der Regel bei verfügbaren Synthesewerkzeugen erst im Rahmen der Technologieabbildung durchgeführt werden, sind beim Square-Dance-Entwicklungssystem Teil der Logiksynthese. Hintergrund dieses Vorgehens ist die Überlegung, dass ein Debugging auf Technologieebene vergleichsweise schwierig durchzuführen ist. Schaltungsteile, die im Rahmen einer Optimierung umgestellt oder weggelassen wurden, sind innerhalb einer zumeist flachen technologieabhängigen Netzliste praktisch nicht mehr zu identifizieren. Werden die Optimierungen im Gegensatz dazu schon auf Logikebene durchgeführt, sind diese durch die dann noch vorhandene hierarchische Struktur leichter zu identifizieren und nachzuvollziehen. Das durch die Optimierung ermöglichte automatische Weglassen nicht benötigter Schaltungsteile macht die generierten Schaltungen zusätzlich besser lesbar.

Zunächst wird die gegebene Schaltung auf besondere Schaltungselemente überprüft. So ist es beispielsweise möglich, ein Latch in Form eines Multiplexers zu beschreiben, welches dann zunächst von der Synthese auch, wie in Abbildung 4.14 dargestellt, in Form eines

```
1.  signal a/state/0, a/state/1, a/count/0, a/count/1: bit;
2.  signal b/state/0, b/state/1, b/count/0, b/count/1: bit;
3.  signal x/state/0, x/state/1, x/count/0, x/count/1: bit;
4.  signal sel: bit;
5.
6.       [...]
7.
8.  with sel select x/state/0 <= a/state/0 when '0', \
                                  b/state/0 when others;
9.  with sel select x/state/1 <= a/state/1 when '0', \
                                  b/state/1 when others;
10. with sel select x/count/0 <= a/count/0 when '0', \
                                  b/count/0 when others;
11. with sel select x/count/1 <= a/count/1 when '0', \
                                 b/count/1 when others;
```

Listing 4.13 *Aufsplittung des komplexen Typs von Listing 4.12 in seine elementaren Bestandteile.*

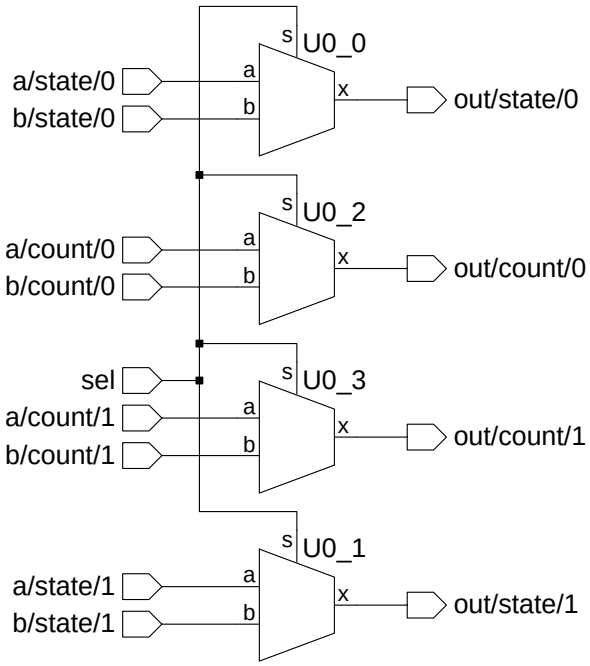


Abbildung 4.13 *Aus Listing 4.12 generierte Schaltung.*

Multiplexers umgesetzt wird. Vor der eigentlichen Ausgabe werden deshalb alle Multiplexer auf Rückkopplungen überprüft und ggf., wie in Abbildung 4.15 dargestellt, durch ein entsprechendes Latch ersetzt.

Abschließend wird das Schaltwerk reduziert, d. h., es werden die Schaltungsteile gelöscht, die für die Berechnung der Ausgangssignale irrelevant sind. Weiterhin werden alle Treiberinstanzen zwischen Signalnetzen mit unterschiedlichen Namen (aber gleicher Funktionali-

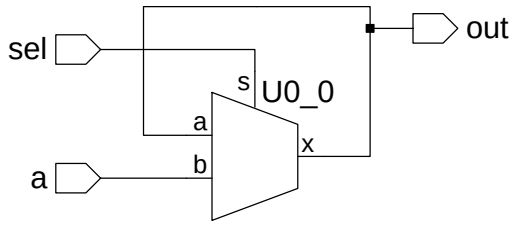


Abbildung 4.14 *Multiplexer mit rückgekoppeltem Ausgang.*

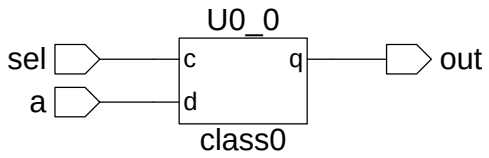


Abbildung 4.15 *Entsprechend Abbildung 4.14 generiertes Latch.*

tät) gelöscht und die betreffenden Netze zusammengelegt. Handelt es sich bei den zusammengelegten Signalnetzen um Signale, denen in der Ausgangsbeschreibung ein fester Name zugewiesen wurde, erhält das zusammengelegte Netz einen aus den Namen der Einzelnetze zusammengesetzten Namen, um ein späteres Debugging zu erleichtern. Soweit vorhanden, werden Inverterkaskaden eliminiert. Alle Schaltnetze zwischen Speicherelementen werden auf anliegende konstante Signale überprüft und solange entsprechend reduziert, bis keine konstanten Signale mehr vorliegen. Die Anwendung des gleichen Schrittes auf Speicherelemente, die am Dateneingang mit konstanten Signalen beschaltet sind, wird nicht durchgeführt, da hierzu Annahmen über die später verwendete Technologie getroffen werden müssten. Durch diesen Schritt wäre es zum Beispiel möglich, dass ein in der Ausgangsbeschreibung aufgeführter, einmal beschreibbarer Speicher (WORM) entfernt wird.

4.5 Synthese arithmetischer Strukturen

Im Gegensatz zum Stand der Technik werden bei der Logiksynthese des Square-Dance-Entwicklungssystems keine arithmetischen Strukturen erzeugt. Es wird ausgenutzt, dass die für die VHDL-Beschreibung arithmetischer Strukturen zu verwendende Schnittstelle innerhalb von IEEE-VHDL-Packages standardisiert wurde. Diese standardisierten Pakete werden im Gegensatz zu anderen Synthesewerkzeugen von der Logiksynthese SibaS nicht mitübersetzt. Die im Design verwendeten Typen und Funktionen der Packages werden von der Synthese direkt interpretiert, so dass diese direkten Zugriff auf abstrakte Informationen des Designs haben. Im Unterschied zu den im Kapitel „Logiksynthese aus VHDL-Beschreibungen“ auf Seite 12 in Abbildung 2.2 dargestellten Abhängigkeiten bei der herkömmlichen Synthese ergibt sich der in Abbildung 4.16 dargestellte geänderte Aufbau. Die genaue Abstimmung von Logiksynthese und Technologieabbildung erlaubt es, bestimmte Schaltungsstrukturen aus der Logiksynthese herauszunehmen und der Technologieabbildung zur endgültigen Synthese zu überlassen. Zum gegenwärtigen Zeitpunkt betrifft dies vor allem elementare arithmetische Funktionen wie Addierer und Vergleicher. Es ist jedoch denkbar, dieses auf komplexere Funktionen wie beispielsweise Multiplizierer auszudehnen, die gegenwärtig noch nicht synthetisiert werden können.

Die eigentliche Synthese der arithmetischen Strukturen ist beim Square-Dance-Entwicklungssystem Teil der Technologieabbildung, welche anhand der vorgegebenen Fertigungstechnologie und Randparameter eine jeweils optimale Realisierung für die abzubildende arithmetische Einheit auswählen kann.

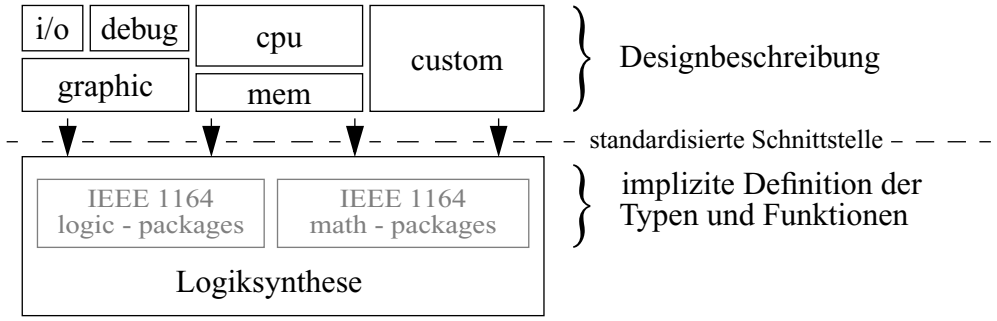


Abbildung 4.16 *Abhängigkeit eines Designs von standardisierten Packages und Anbindung an die Synthese.*

Der Einsatz von Coregeneratoren ist auch nach dieser mächtigeren Synthese weiterhin sinnvoll, da sie viele Schaltungen wie beispielsweise Speicher, Schaltungen für Signaltransformationen, Bus-Interfaces oder Microcontroller generieren können, die sich nicht oder nur mit hohem Aufwand in VHDL beschreiben lassen. Der Wirkungsbereich der Coregeneratoren verschiebt sich also hin zu komplexeren Einheiten.

5 Technologieabbildung ALTeM

Ziel der Technologieabbildung ist im Allgemeinen die Umsetzung einer generischen in eine technologiespezifische Netzliste. Neben der möglichst ressourcenschonenden Umsetzung gilt es außerdem, vorgegebene Randbedingungen einzuhalten und die ggf. durch die Technologie gegebenen Besonderheiten zu erkennen und zu nutzen. Im Gegensatz zur Technologieabbildung konventioneller Synthesewerkzeuge erfolgt bei der Technologieabbildung ALTeM (Arithmetic-Logic-Technology-Mapping) neben der reinen Logikabbildung zusätzlich eine spezielle Arithmetikabbildung. Da die hierzu nötigen zusätzlichen Informationen innerhalb einer zu anderen Werkzeugen kompatiblen Schnittstelle nicht mit verschlüsselt werden können, baut die Technologieabbildung ALTeM direkt auf der A-RTL-Datenbank auf. Der Einsatz von ALTeM, losgelöst von der Logiksynthese SibaS, ist nicht vorgesehen.

Die Arithmetikabbildung von ALTeM demonstriert, dass es grundsätzlich möglich ist, auf Coregeneratoren in Hinblick auf die Erzeugung einfacher arithmetischer Strukturen zu verzichten. Das Verfahren in schon bestehende Werkzeuge zu integrieren, gestaltet sich jedoch sehr schwierig, da benötigte Zusatzinformationen bei der konventionellen Logiksynthese verloren gehen. Dementsprechend sind neben den durchzuführenden Modifikationen der Werkzeuge selbst, zusätzlich die Entwicklungen neuer Schnittstellen zwischen Logiksynthese und Technologieabbildung notwendig.

5.1 Übersicht

Ausgehend von der generischen Netzliste erfolgt die Technologieabbildung entsprechend dem in Abbildung 5.1 dargestellten Ablauf: Die von der Logiksynthese SibaS erstellte generische Netzliste wird zuerst einer Vorsynthese unterworfen. Ziel der Vorsynthese ist es, eine erste Abschätzung zum Ressourcenbedarf (Größe und Leistungsbedarf) der einzelnen Teilschaltungen zu ermitteln. Anhand dieser Abschätzung wird in der nächsten Stufe eine Aufteilung der tatsächlich vorhandenen Ressourcen auf die Teilschaltungen vorgenommen. Für jede Komponente wird dann, unter Berücksichtigung zugeteilter Ressourcen und ggf. vorhandener Optimierungsrichtlinien, getrennt die technologieabhängige Logik- und Arithmetiksynthese durchgeführt. Ausgehend von dem sich aus der Technologieabbildung ergebenden tatsächlichen Ressourcenbedarf wird in der Rückkopplung die Verteilung der Ressourcen angepasst. War eine Synthese mit den gegebenen Vorgaben möglich, wird der Zyklus verlassen. Konnten die Vorgaben nicht eingehalten werden, wird eine Neuverteilung der Ressourcen vorgenommen. In jedem Durchlauf werden dabei Teilschaltungen als unveränderlich markiert. Der Zyklus wird auch dann verlassen, wenn alle Teilschaltungen als unveränderlich markiert wurden. Im letzten Schritt werden die Teilschaltungen zusammengesetzt und, ausgehend von der dann flachen Struktur, alle Schaltnetze einer weiteren Optimierung unterzogen.

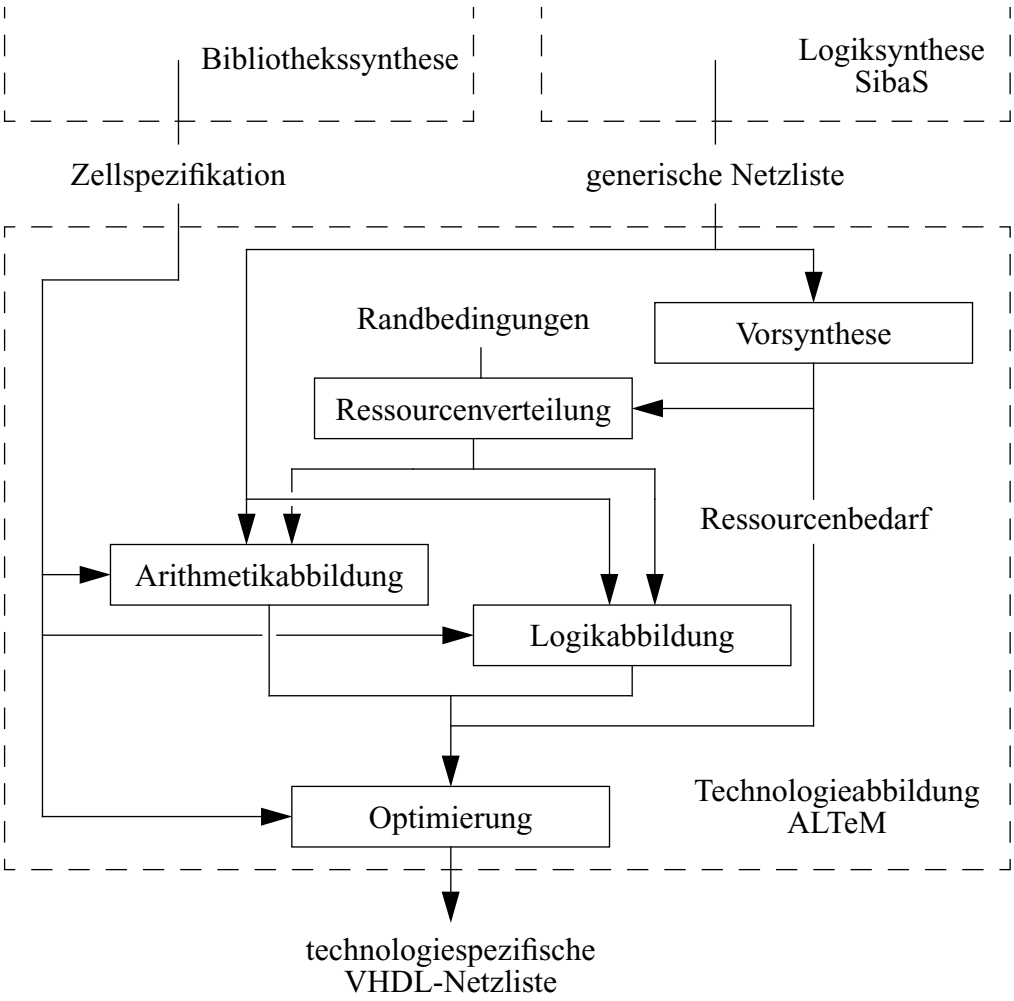


Abbildung 5.1 *Aufbau der Technologieabbildung und Einbettung in den Schaltungssyntheseprozess*.

Sind keine absoluten Randbedingungen gegeben, wird der Zyklus nur einmal durchlaufen. Für jede Teilschaltung wird dann entsprechend der relativen Ressourcenvorgaben in der Logik- bzw. Arithmetiksynthese lokal nach einer optimalen Abbildung gesucht.

5.2 Vorsynthese

5.2.1 Motivation

Gängige Verfahren der Technologieabbildung nehmen eine Clusterung [16, 54] der Gesamtschaltung vor, um den durch die Nichtlinearität der verwendeten Mapping-Algorithmen gegebenen Rechenaufwand zu reduzieren. Äquivalente Aspekte gelten auch für das vorliegende Verfahren. Eine erste Clusterung ist durch die vom Designer vorgegebene Struktur des Designs gegeben. In der Regel sind dann funktionell zusammengehörige Teilschaltungen in einzelnen Modulen gekapselt. Zusätzlich erlaubt es die Technologieabbildung ALTeM dem Benutzer, einzelnen Teilschaltungen individuelle Ressourcen zuzuordnen.

So können Teile einer insgesamt auf Verlustleistung zu optimierenden Schaltung beispielsweise auf Geschwindigkeit optimiert werden. Mit dem so optional eingebrachten Hintergrundwissen des Designers können sich global ausschließende Randbedingungen gezielt derart verteilt werden, dass auf eine wiederholte Technologieabbildung mit jeweils geänderten globalen Randbedingungen seitens des Benutzers verzichtet werden kann. Dies kann insbesondere bei Designs mit sehr engen einzuhaltenden Vorgaben helfen, die effektiv vom Benutzer benötigte Entwurfszeit zu reduzieren.

Aufgabe der Vorsynthese ist es, den Zyklus der Ressourcenvergabe an die Teilschaltungen mit Ausgangsdaten unter Berücksichtigung der vom Benutzer vorgegebenen Ausnahmen zu initialisieren. Die Güte der Vorsynthese bestimmt maßgeblich die Zahl der später tatsächlich zu durchlaufenden Zyklen, was sich in der absoluten Rechenzeit für den Syntheseprozess niederschlägt. Grundsätzlich ist eine Vorsynthese nur dann sinnvoll, wenn eine absolute Grenze der verfügbaren Ressourcen (Platz- und Leistungsbedarf) gegeben ist. Sind keine absoluten Ressourcen gegeben, müssen sie auch nicht aufgeteilt werden, und es wird dementsprechend für diese Schaltungsteile keine Vorsynthese durchgeführt. Da kein Einfluss der Güte der Vorsynthese auf das Endergebnis vorhanden ist, ist die Vorsynthese primär laufzeitorientiert implementiert.

5.2.2 Verfahren

Für die initiale Aufteilung der tatsächlich verfügbaren Ressourcen sind relative Größenangaben ausreichend. Es wird davon ausgegangen, dass die Verhältnisse von Ressourcen und Funktion in allen Technologien ähnlich sind, so dass direkt auf der bereits vorhandenen generischen Netzliste gearbeitet werden kann. Als weitere Vereinfachung wird angenommen, dass die später durchzuführenden Optimierungen auf alle Teilschaltungen einen ähnlichen Effekt haben und sich die Verhältnisse (wenn überhaupt) nur wenig ändern. Zusätzlich kann der relative Platz- und Leistungsbedarf in einer Größe zusammengefasst werden, da im gegenwärtigen Zustand der Schaltung keine Bauelemente zum Einsatz kommen, bei denen das Verhältnis Platz-/Leistungsbedarf auffällig abweicht (wie z. B. bei RAM oder I/O-Zellen). Der sich nach diesen Annahmen ergebende Ressourcenbedarf für die einzelnen Bauelemente ist in Tabelle 5.1 abgebildet.

Tabelle 5.1: *Relativer Ressourcenbedarf für generische Technologie*

Zelltyp	Ressourcenbedarf
Bustreiber	5
Latch	5
Flipflop	10
Logik; n Bit	n
Addition/Subtraktion; (0 < n <= 3)	3n
Addition/Subtraktion ; (3 < n)	15n
Multiplikation; (n; m)	15nm

Der Ressourcenbedarf einer Teilschaltung setzt sich zusammen aus dem Ressourcenbedarf der in ihr instantiierten Bauelemente und Teilschaltungen. Dieser wird dann als zusätzliches Attribut an die Teilschaltung gebunden. Der relative Gesamtressourcenbedarf wird dementsprechend der obersten Instanz der Schaltung zugeordnet.

5.3 Ressourcenverteilung

Sind für die Technologieabbildung hinsichtlich Platz und Leistung nur begrenzte Ressourcen verfügbar, werden diese in diesem Schritt entsprechend dem Bedarf auf die einzelnen Teilschaltungen verteilt. Sind keine absoluten Vorgaben hinsichtlich der Ressourcen gegeben, wird dieser Schritt übersprungen.

5.3.1 Verfahren

Nach der Vorsynthese wird die Ressourcenverteilung zunächst mit den dort berechneten relativen Aufwandsabschätzungen initialisiert. Innerhalb der obersten Instanz werden die tatsächlich verfügbaren Ressourcen im Verhältnis der Abschätzungen auf die Teilschaltungen verteilt. Für jede Teilschaltung läuft das gleiche Verfahren rekursiv bis zu den elementaren Unterschaltungen weiter. Im in Abbildung 5.2 dargestellten Beispiel werden der

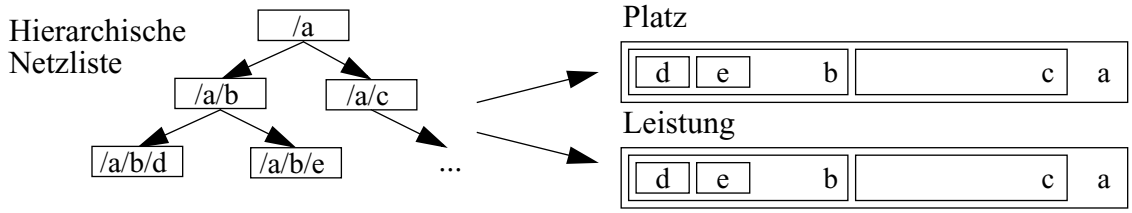


Abbildung 5.2 *Beispiel einer Schaltungshierarchie und die initale Verteilung auf absolute Ressourcen.*

obersten Schaltungsinstanz **/a** alle Ressourcen zugewiesen. Diese werden auf die in **/a** instantiierten Schaltungen **/a/b** und **/a/c** entsprechend den Abschätzungen aufgeteilt. Das gleiche Verfahren wird dann innerhalb der Instanzen durchgeführt.

Können bei der nachfolgenden Logik- und Arithmetik-Synthese die Vorgaben innerhalb einer Instanz nicht eingehalten werden, wird die Schaltungsvariante der Instanz, die am wenigsten Ressourcen benötigt, für den weiteren Prozess als unveränderlich markiert. Die Differenz von tatsächlich benötigter und vorgegebener Ressource wird in diesem Fall an die nächst höhere Schaltungsinstanz weitergegeben. Diese teilt die erhaltene negative Ressource auf alle noch nicht als unveränderlich markierten Instanzen auf. Im Abbildung 5.3 reicht der von der Ressourcenvergabe vorgegebene Platz für die Instanz **d** nicht aus. Das Syntheseergebnis mit minimalem Platzbedarf unter Einhaltung aller weiteren Ressourcen **d'** wird als unveränderlich markiert (im Bild durch den schwarzen Balken symbolisiert) und die Differenz des Platzes von **d** und **d'** auf **e** und **b** entsprechend dem bekannten Ressourcenverhältnisses verteilt. Reicht, wie in Abbildung 5.4 dargestellt, beispielsweise der

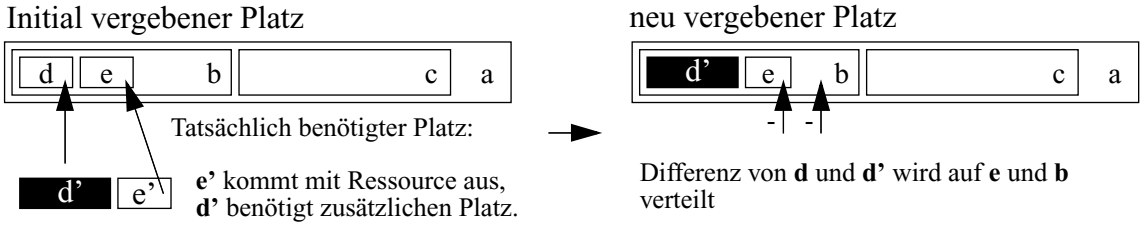


Abbildung 5.3 *Verteilung negativer Ressourcen innerhalb einer Instanz bei Nichteinhaltung der Vorgaben einer Unterinstanz.*

dadurch reduzierte Platz für **e** nicht mehr aus, wird das minimale Syntheseergebnis **e'** ebenfalls als unveränderlich markiert. Kann der zusätzliche Platzbedarf innerhalb der Instanz **b** nicht mehr anderweitig ausgeglichen werden, wird das minimal mögliche **b'** als unveränderlich markiert (siehe Alternative 1) und die negative Ressource weiter an die nächsthöhere Instanz **a** zur weiteren Verteilung weitergegeben. Ist ein Ressourcenausgleich innerhalb der Instanz möglich (siehe Alternative 2), muss auch keine Neuverteilung der Ressourcen erfolgen. Die Instanz **b'** wird nicht als unveränderlich markiert, da die Instanz möglicherweise weitere negative Ressourcen aufnehmen kann, die z. B. bei der Synthese von **c** anfallen.

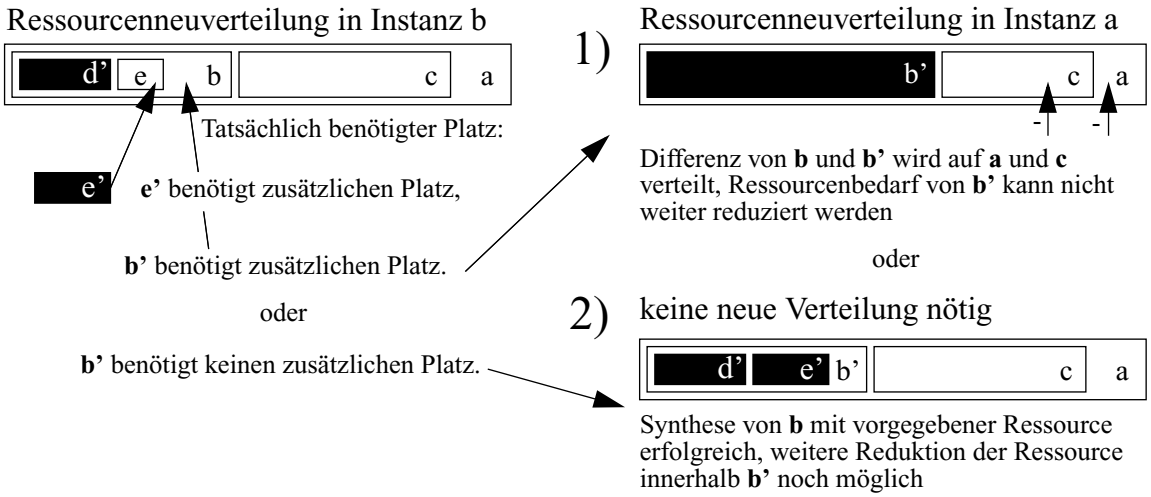


Abbildung 5.4 1) *zusätzlicher Ressourcenbedarf von* **e'** *kann innerhalb Instanz* **b** *nicht ausgeglichen werden und wird durchgereicht;* **2)** *zusätzlicher Ressourcenbedarf kann ausgeglichen werden.*

Sind zwei absolute Ressourcen vorgegeben, läuft das Verfahren parallel für beide Ressourcen. Wird dabei eine Schaltungsinstanz als unveränderlich markiert, wirkt dieses auch auf die jeweils andere Ressource, auch wenn hier die Vorgaben nicht verletzt wurden. Durch dieses Vorgehen wird zwar möglicherweise die optimale Abbildung nicht gefunden, jedoch werden so aufwändig zu behandelnde Rückkopplungen vermieden.

5.3.2 Geschwindigkeitsoptimierung

Die Optimierung hinsichtlich Geschwindigkeit muss gesondert betrachtet werden, da sie als Ressource im Gegensatz zu Platz- und Strombedarf nicht verteilt werden kann, sondern als Konstante global auf alle Teilschaltungen einer Geschwindigkeitsdomain wirkt. Drei Fälle sind zu unterscheiden:

1. Keine Vorgabe: Die Geschwindigkeit bleibt vollständig unberücksichtigt. Die in der Arithmetik- und Logikabbildung verwendete Gütefunktion ist geschwindigkeitsunabhängig.

2. Absolute Vorgabe: Die gegebene Geschwindigkeit ist in jeder Teilschaltung einzuhalten. Da eine Verletzung dieser Vorgabe nicht anderweitig ausgeglichen werden kann, hat diese Vorgabe immer höhere Priorität als alle aufteilbaren Ressourcen. Die Gütefunktion wird dementsprechend soweit angepasst, dass die Geschwindigkeitsvorgabe in jedem Fall erfüllt wird.

3. Relative Vorgabe: Nach der vollständigen Technologieabbildung mit geschwindigkeitsabhängiger Gütefunktion limitiert die langsamste Teilschaltung die Geschwindigkeit des Gesamtsystems. Mit dieser festen Geschwindigkeit kann die Technologieabbildung komplett mit absoluter Geschwindigkeitsvorgabe wiederholt werden. Diese nach Fall 2 durchgeführte Abbildung kann zu einem ressourcenschonenderem Ergebnis führen, da die Geschwindigkeit dann auf die Gütefunktion keinen Einfluss mehr hat.

5.3.3 Bewertung

Das Verfahren führt bei ausreichend kalkulierten Vorgaben vergleichsweise schnell zu einer guten Aufteilung der Ressourcen. Je knapper die Ressourcen vom Benutzer kalkuliert wurden, desto mehr steigt die Zahl der neu zu übersetzenden Teilschaltungen zwecks Ressourcenausgleichs an. Im Extremfall kann eine unzureichende Ressource in der letzten zu übersetzenden Unterinstanz dazu führen, dass die komplette Schaltung neu abgebildet werden muss, wenn die anfallenden negativen Ressourcen bis in die oberste Instanz durchgereicht werden.

Die beschriebene Vorgehensweise bei der parallelen Verteilung von zwei Ressourcen ist im gegebenen Umfeld nur aufgrund der Abhängigkeiten der Ressourcen voneinander praktikabel (in der Regel ist das Verhältnis von Stromverbrauch zu Platzbedarf bei gleicher Technologie und Geschwindigkeit konstant). Für den Einsatz unterschiedlicher Technologien in einem Projekt oder von mehreren Taktdomains mit jeweils deutlich unterschiedlichen Geschwindigkeiten ist das Verfahren im gegenwärtigen Stadium nur eingeschränkt geeignet, da in diesem Fall andere Abhängigkeiten der Ressourcen voneinander in den jeweiligen Schaltungen vorhanden sind. Für eine entsprechende Verbesserung des Verfahrens müsste u. A. eine automatische Erkennung unterschiedlicher Taktdomains innerhalb der Technologieabbildung durchgeführt werden, um die jeweiligen Abhängigkeiten der Ressourcen zu ermitteln und sie für jede entsprechende Unterschaltung separat zu sichern.

5.4 Gütefunktion

Die neu eingeführte Möglichkeit der ALTeM-Technologieabbildung, relative Ressourcenvorgaben zu verarbeiten, erfordert auch einen neuen Ansatz bei der Bewertung der generierten Schaltung. Dokumentierte Technologieabbildungsverfahren optimieren mittels spezieller Algorithmen[67,10] in der Regel hinsichtlich klar definierter Randbedingungen oder spezieller Technologien [35] und meist auch nur hinsichtlich eines Kriteriums (Verlustleistung, Geschwindigkeit oder Platzbedarf). Im vorliegenden Ansatz können die Optimierungskriterien und die Technologie frei gewählt werden.

Jeder auf eine Technologie abgebildeten Schaltung **x**, charakterisiert durch ihre Eigenschaften $\mathbf{a_x}$ (Platzbedarf), $\mathbf{p_x}$ (Strombedarf) und $\mathbf{t_x}$ (maximale Signallaufzeit), wird ein Wert $\mathbf{g_x}$ (siehe Gl. 6-1) zugeordnet, der die Güte der Schaltung hinsichtlich der vorgegebenen relativen Randbedingungen $\mathbf{r_a}$ (relativer Platzbedarf), $\mathbf{r_p}$ (relativer Strombedarf) und $\mathbf{r_t}$ (relative Geschwindigkeit) beschreibt.

$$g_x = a_x^{r_a} \cdot p_x^{r_p} \cdot t_x^{r_t} \qquad \text{(Gl. 6-1)}$$

Aufgabe der Arithmetik- und Logikabbildung ist es, für eine gegebene Schaltung Abbildungen **x** zu finden, für die der Wert $\mathbf{g_x}$ minimal ist und alle ggf. vorhandenen absoluten Ressourcenbeschränkungen $\mathbf{m_a}$ (maximaler Platzbedarf), $\mathbf{m_p}$ (maximaler Strombedarf) und $\mathbf{m_t}$ (maximale Signallaufzeit) nicht verletzt sind. Die relativen und absoluten Randbedingungen werden von der Ressourcenverteilung vorgegeben. Reichen die absoluten Ressourcen für die Technologieabbildung nicht aus, werden die relativen Ressourcen, und damit auch die Gütefunktion, modifiziert. Die Änderung einer relativen Randbedingungen **x** geschieht in der Form:

$$r_{(x;n+1)} = \begin{cases} r_{(x;n)} + \dfrac{1 - r_{(x;n)}}{2^n} & \text{für} \quad a_{(x;n)} < m_{(x;n)} \\ r_{(x;n)} - \dfrac{1 - r_{(x;n)}}{2^n} & \text{für} \quad a_{(x;n)} > m_{(x;n)} \end{cases} \qquad \text{(Gl. 6-2)}$$

Die Änderung, welche die Ressource $\mathbf{r_x}$ durch diesen Vorgang erfährt, wird entsprechend der initialen Verteilung ($\mathbf{r_{(y;0)}}$ und $\mathbf{r_{(z;0)}}$) mit der aktuellen Verteilung verrechnet:

$$r_{(y;n+1)} = r_{(y;n)} + \frac{r_{(y;0)}\left(r_{(x;n)} - r_{(x;n+1)}\right)}{r_{(y;0)} + r_{(z;0)}} \qquad \text{(Gl. 6-3)}$$

$$r_{(z;n+1)} = r_{(z;n)} + \frac{r_{(z;0)}\left(r_{(x;n)} - r_{(x;n+1)}\right)}{r_{(y;0)} + r_{(z;0)}} \qquad \text{(Gl. 6-4)}$$

Werden in einer Abbildung zwei voneinander unabhängige absolute Ressourcen überschritten, können die zugehörigen relativen Ressourcen parallel entsprechend Gleichung 6-2 modifiziert werden. Die Gleichung zur Berechnung der verbleibenden Ressource vereinfacht sich:

$$r_{(z;n+1)} = 1 - r_{(x;n+1)} - r_{(y;n+1)} \qquad \text{(Gl. 6-5)}$$

Fällt der Wert der verbleibenden Ressource dabei auf einen Wert unter 0, so können die vorgegebenen absoluten Ressourcen nicht eingehalten werden. In diesem Fall ist eine Priorisierung der Ressourcen notwendig, um zumindest einen Teil der Vorgaben einzuhalten:

1. Ist eine absolute Geschwindigkeitsgrenze gegeben, hat diese Vorrang. Die Ressourcenanpassung mit Gleichung 6-2 wird ausschließlich für die relative Geschwindigkeit $\mathbf{r_t}$ durchgeführt, auch wenn dabei andere absolute Ressourcen verletzt werden. Kann die absolute Geschwindigkeitsgrenze auch dann nicht eingehalten werden, wird die Abbildung mit der Gütefunktion $g_x = t_x$ zurückgegeben.
2. Ist keine absolute Geschwindigkeitsgrenze gegeben, wird die Ressource mit dem höchsten initialen relativen Ressourcenbedarf analog zu 1. bevorzugt.

Der Zyklus wird abgebrochen, wenn sich nach zwei Durchgängen keine Änderungen mehr im tatsächlichen Ressourcenbedarf der jeweils generierten Abbildung ergeben. Zur Verdeutlichung ist in Listing 5.1 das Verfahren als Pseudocode gegeben.

```
1.  start:
2.  res_x-1 = res_x;
3.  res_x = abbilden( schaltung(x), g_x );
4.  while( (res_x > absolute_ressourcen_x) und (res_x-1 != res_x) )
5.    if( eine absolute ressource x überschritten )
6.      r_x = gl6.2( r_x );
7.      r_y = gl6.3( r_y );
8.      r_z = gl6.4( r_z );
9.    elsif( zwei absolute ressourcen x, y überschritten )
10.     r_x = gl6.2( r_x );
11.     r_y = gl6.2( r_y );
12.     r_z = gl6.4( r_x, r_y );
13.     if( r_z < 0 )
14.
15.       if( t_x < ∞ )
16.         a_x = ∞;
17.         p_x = ∞;
18.         goto start;
19.       else
20.         if( r_a,0 < r_p,0 )
21.           a_x = ∞;
```

Listing 5.1 *Programm zur Modifikation der Gütefunktion bei der Technologieabbildung.*

```
22.          else
23.            p_x = ∞;
24.          end if;
25.          goto start;
26.        end if;
27.      end if;
28.    elsif( drei absolute ressourcen ueberschritten )
29.      goto end;
30.    end if;
31.    res_x-1 = res_x;
32.    res_x = abbilden( schaltung(x), g_x );
33. end while;
34. end:
```

Listing 5.1 *Programm zur Modifikation der Gütefunktion bei der Technologieabbildung.*

5.5 Logikabbildung

Die Logikabbildung besteht aus folgenden Schritten:

- Abbildung aller Speicherelemente,
- Auftrennung von Rückkopplungen innerhalb der Schaltnetze,
- Abbildung der Schaltnetze.

5.5.1 Abbildung der Speicherelemente

Die Abbildung der Speicherelemente gehört streng genommen nicht zur Logikabbildung, wurde aus Effizienzgründen jedoch mit in diese integriert. Wurden von der Logiksynthese Gruppen von Flipflops oder Latches als zusammengehörig markiert (siehe „Abschließende Überprüfungen und Optimierungen“ auf Seite 46), werden diese, je nach vorhandenen Breiten der entsprechenden Zellen der Technologiebibliothek, abgebildet. Stehen die Zellen in mehreren Treiberstärken zur Verfügung, wird die Zweitschwächste genommen, da aufgrund von zu diesem Zeitpunkt noch fehlenden tatsächlich auftretenden Lasten keine korrekte Dimensionierung möglich ist. Die Treiberstärken werden im Anschluss an die Schaltnetzabbildung angepasst. Flipflops, deren Taktsignal exklusiv für maximal noch ein weiteres Flipflop bestimmt ist, werden als Teil des einbettenden Schaltnetzes betrachtet und aus der Speicherabbildung an dieser Stelle ausgenommen.

5.5.2 Zyklenerkennung

Die im nachfolgenden Schritt durchzuführende Abbildung der Schaltnetze funktioniert nur mit azyklischen Schaltungen. Vorhandene Rückkopplungen müssen also vorher erkannt und aufgebrochen werden. Das Problem der Zyklenerkennung und -beseitigung ist ein grundlegendes Problem im Rahmen des Hardwaredesigns [34, 74], entsprechende Algorithmen sind unter [13] dokumentiert. Im vorliegenden Fall wird zur Zyklenerkennung ein in [52]

beschriebener, angepasster Ansatz verwendet. Für die Zyklenbeseitigung wurde ein eigener Ansatz gewählt, um die Zahl der nötigen Schnitte diese aufzulösen zu minimieren.

Die Schaltung wird als gerichteter Graph (Gatter werden als Knoten, Signalleitungen als gerichtete Kanten interpretiert) behandelt und topologisch sortiert [38], um die ggf. vorhandenen Zyklen zu erkennen und aufzuteilen. Als Beispiel kann die in Abbildung 5.5 gegebene Ausgangsschaltung als gerichteter Graph (wie in Abbildung 5.6 dargestellt) interpretiert und in der in Tabelle 5.2 dargestellten Form als Adjazenzmatrix gespeichert werden. Der Graph wird aus Effizienzgründen so weit wie möglich reduziert. Knoten, die ausschließlich einlaufende bzw. auslaufende Kanten haben, können nicht Teil eines Zyklusses sein und deswegen aus dem Graph entfernt werden.

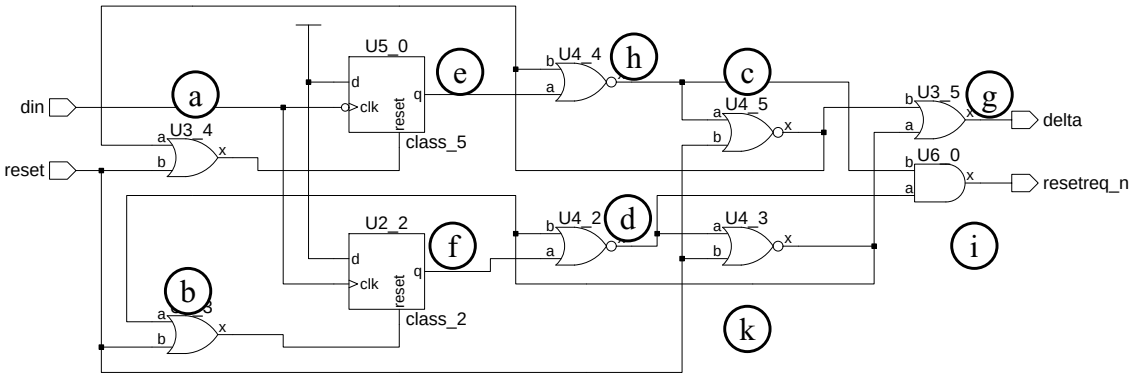


Abbildung 5.5 *Ausgangsnetz für die Zyklenerkennung.*

Im Beispiel (siehe Abbildung 5.6) haben die Knoten **i** und **g** nur einlaufende Kanten. In der Adjazenzmatrix (siehe Tabelle 5.2) sind bei genau diesen Knoten alle Einträge in den entsprechenden Zeilen 0. Es werden aus der Tabelle alle zu einem Knoten zugehörigen Zeilen und Spalten gelöscht, für die eine Zeile oder Spalte existiert, die nur Nullwerte enthält (in Tabelle 5.2 grau hervorgehoben). Dieser Vorgang wird mit der jeweils entstehenden neuen Tabelle solange fortgesetzt, bis in jeder Zeile oder Spalte mindestens ein von 0 verschiedener Wert aufgeführt wird. Ist nach diesem Prozess die Tabelle leer, ist der Graph zyklenfrei, und das entsprechende Schaltnetz enthält keine Rückkopplungen, kann also unmodifiziert direkt weiter verarbeitet werden. Ist die Tabelle nach der Reduktion nicht leer, enthält der Graph mindestens einen Zyklus, und alle noch vorhandenen Knoten liegen in mindestens einem Zyklus. Im gegebenen Beispiel ist das Ergebnis der Reduktion in Abbildung 5.7 bzw. Tabelle 5.3 gegeben.

Überschneiden sich mehrere Zyklen, kann man diese mit genau einem Schnitt an der Überschneidungsstelle auftrennen. Diese Überschneidungsstücke müssen, wenn vorhanden, identifiziert werden, um die Anzahl der aufzutrennenden Kanten so gering wie möglich zu halten. Hierzu werden zunächst alle Zyklen identifiziert. Der Algorithmus ist in Listing 5.2 als Pseudocode gegeben. An einem beliebigen Knoten (`knoten`) des gegebenen Graphen (`Graph`) wird hierbei eine Tiefensuche (`finde_zyklen`) gestartet. Trifft der Pfad (`pfad`) durch den als Baum interpretierten Graphen auf einen bereits besuchten Knoten, wird der dann im Pfad protokollierte Zyklus (`zyklusstack`) separat gesichert

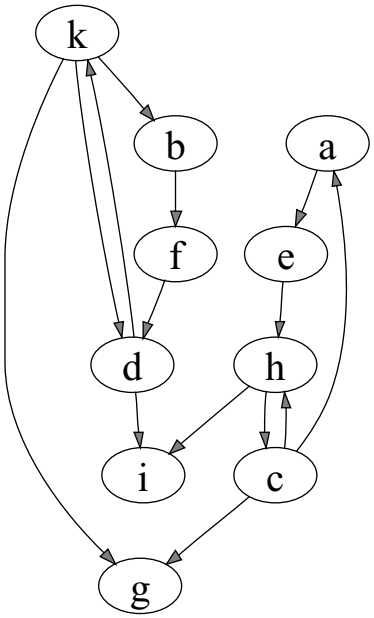


Abbildung 5.6 *Gerichteter Graph der in Abbildung 5.5 dargestellten Schaltung.*

Tabelle 5.2: *Adjazenzmatrix des in Abbildung 5.6 dargestellten Graphen*

	a	b	c	d	e	f	g	h	i	k
a	0	0	0	0	1	0	0	0	0	0
b	0	0	0	0	0	1	0	0	0	0
c	1	0	0	0	0	0	1	1	0	0
d	0	0	0	0	0	0	0	0	1	1
e	0	0	0	0	0	0	0	1	0	0
f	0	0	0	1	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0	0
h	0	0	1	0	0	0	0	0	1	0
i	0	0	0	0	0	0	0	0	0	0
k	0	1	0	1	0	0	1	0	0	0

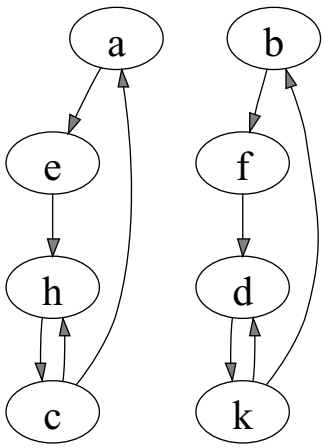


Abbildung 5.7 *Zyklen des in Abbildung 5.6 dargestellten Graphen.*

Tabelle 5.3: *Adjazenzmatrix des in Abbildung 5.7 dargestellten Graphen*

	a	b	c	d	e	f	h	k
a	0	0	0	0	1	0	0	0
b	0	0	0	0	0	1	0	0
c	1	0	0	0	0	0	1	0
d	0	0	0	0	0	0	0	1
e	0	0	0	0	0	0	1	0
f	0	0	0	1	0	0	0	0
h	0	0	1	0	0	0	0	0
k	0	1	0	1	0	0	0	0

(`zyklenlist`) und es kommt zum Backtracking. Ist die Tiefensuche beendet, und es existiert noch mindestens ein nicht besuchter Knoten (`knotenlist`), wird an diesem eine neue Suche gestartet. Sind keine unbesuchten Knoten mehr vorhanden, wurden alle Zyklen gefunden.

Die Kanten zwischen den Knoten sind durch die Ordnung der Knoten in den ermittelten Zyklenlisten implizit gegeben. Jede Kante kann in einem Zyklus nur einmal vorhanden sein, jedoch Teil von mehreren Zyklen sein. Alle in den protokollierten Zyklen implizit gegebenen Kanten werden nach Häufigkeit sortiert. Die am häufigsten vorkommende Kante wird

```
1.  list knotenlist, zykluslist;
2.  stack pfad;
3.  foreach( knoten ∈ Graph )
4.    knotenlist.insert( knoten );
5.  end foreach;
6.  while( knotenlist.not_empty() )
7.    knoten = knotenlist.front();
8.    finde_zyklen( knoten );
9.  end while;

10. function finde_zyklen( knoten )
11.   knotenlist.erase( knoten );
12.   if( knoten ∉ pfad )
13.     pfad.push( knoten );
14.     foreach( kante( quelle->ziel )∈ Graph| quelle=knoten)
15.       finde_zyklen( ziel );
16.     end foreach;
17.   else
18.     stack tempstack = pfad.copy();
19.     stack zyklusstack.push( knoten );
20.     while( tempstack.top() ≠ knoten )
21.       zyklusstack.push(tempstack.pop());
22.     end while;
23.     zyklenlist.add(zyklusstack);
24.   end if;
25.   pfad.pop();
26. end function;
```

Listing 5.2 *Algorithmus für eine Zyklenidentifikation innerhalb von Graphen.*

zerschnitten und alle Zyklen, welche diese Kante beinhalten, aus der Liste der Zyklen gelöscht. Sind mehrere Kanten mit gleicher Häufigkeit vorhanden, wird eine beliebige ausgewählt. Dieser Vorgang wird solange wiederholt, bis alle Zyklen aufgelöst sind. Im Beispiel Abbildung 5.7 werden folgende Zyklen gefunden:

$$\{\{\mathbf{a}\rightarrow\mathbf{e}\rightarrow\mathbf{h}\rightarrow\mathbf{c}\rightarrow\mathbf{a}\}, \{\mathbf{f}\rightarrow\mathbf{d}\rightarrow\mathbf{k}\rightarrow\mathbf{b}\rightarrow\mathbf{f}\}, \{\mathbf{h}\rightarrow\mathbf{c}\rightarrow\mathbf{h}\}, \{\mathbf{d}\rightarrow\mathbf{k}\rightarrow\mathbf{d}\}\}$$

Die Kanten **h**→**c** und **d**→**k** sind dabei doppelt vorhanden. Nach der Zerschneidung der Kante **h**→**c** und dem Löschen der entsprechenden Zyklen verbleiben die Zyklen {**d**→**k**→**d**} und {**f**→**d**→**k**→**b**→**f**}. Die Kante **d**→**k** ist in diesen Zyklen als einzige doppelt vorhanden und wird zerschnitten. Nach diesem Schritt sind keine Zyklen mehr vorhanden. Übertragen auf die Ausgangsschaltung ergibt sich die in Abbildung 5.8 dargestellte zyklenfreie Schaltung.

Es ist möglich, dass durch dieses Verfahren nicht die minimale Anzahl der Schnitte ermittelt werden kann. Beispielsweise ist es beim in Abbildung 5.9 dargestellten Graphen möglich, dass bei der Identifikation der zu schneidenden Kanten zuerst die Kante **d**→**e** geschnitten wird. Es entstehen dann zwei getrennte Zyklen, die jeder für sich an beliebiger Stelle noch

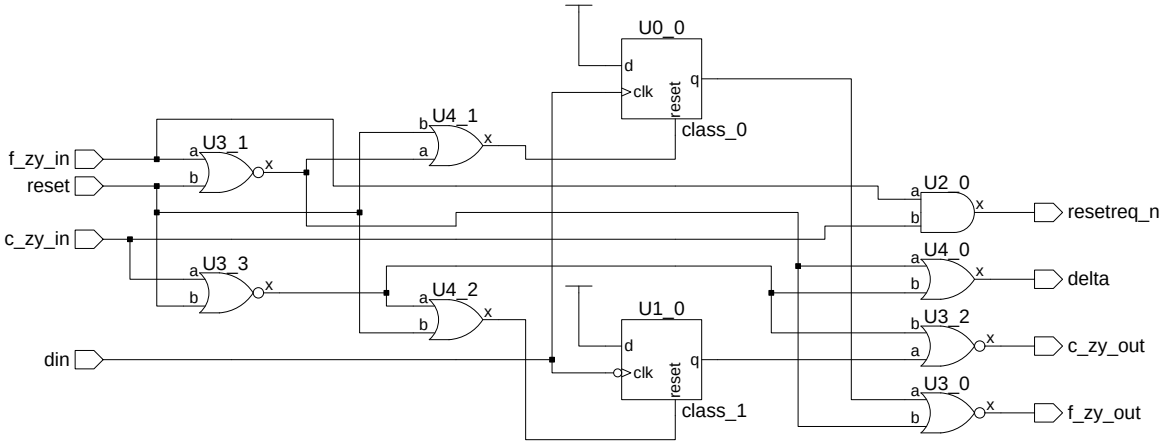


Abbildung 5.8 *Zu Abbildung 5.5 entsprechende zyklenfreie Schaltung.*

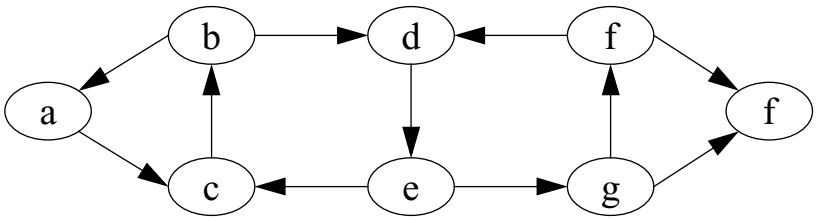


Abbildung 5.9 *Graph mit vier Zyklen; die Kanten* **b** → **c, d** → **e** *und* **g** → **f** *sind Teil von jeweils zwei Zyklen.*

einmal geschnitten werden müssten. Wird stattdessen zuerst **c**→**b** bzw. **g**→**f** geschnitten, reicht nur ein weiterer Schnitt aus, um alle Zyklen zu lösen. Dieses Problem ließe sich beispielsweise durch eine weitere Tiefensuche im Baum der möglichen Schnitte lösen. Solcherart verschachtelte Schaltungen kommen jedoch praktisch nicht vor, deshalb wird auf diesen Schritt aus Effizienzgründen verzichtet.

5.5.3 Abbildung der Schaltnetze

Das Schaltnetz wird als zyklenfreier gerichteter Graph behandelt. Jedem Gatter und allen Ein- und Ausgangssignalen der Schaltung wird jeweils ein eigener Knoten zugeordnet. Ausgehend von den Ausgangsknoten werden analog zur Breitensuche für jeden Knoten die Funktion des Knoten selbst sowie alle Funktionen, die sich aus dem Knoten und den im Graphen vorhergehenden Knoten ergeben, ermittelt. Das Limit für die dabei ermittelten Funktionen wird durch die in der Technologiebibliothek vorhandenen Komplexgatter bestimmt. Unabhängig davon ist die absolute Obergrenze auf neun Eingangsvariablen festgesetzt. Zum einen ist dies ausreichend, um damit die umfangreichsten Komplexgatter gebräuchlicher Technologien zu erfassen, zum anderen entspricht dies der mit der Rechenleistung heute gebräuchlicher Hardware sinnvoll zu verarbeitenden Komplexität.

Anhand der Funktion wird der Hashwert bestimmt und die entsprechenden Zellen aus der Zellbibliothek ermittelt. Aufgrund der Eigenschaften des Hashwertes werden dabei auch die Zellen ermittelt, die das am Knoten zu berechnende Literal in negierter Form liefern bzw. ein negiertes Signal am Eingang erwarten. Gatter, die direkt ein Ausgangssignal treiben (in Abbildung 5.10 grau hinterlegt), sind von der Zusammenfassung von nachfolgenden Gattern zur Bildung von Komplexgattern ausgenommen (ausschließlich Inverter, welche implizit immer mit berücksichtigt werden). Gibt es nach diesem Prozess Knoten, die nicht beachtet wurden, stehen diese für Schaltungsteile, die für die Funktionalität der Gesamtschaltung irrelevant sind und somit gelöscht werden können. In Abbildung 5.10 ist eine Auswahl der ermittelten Zellen für den dargestellten Graphen gegeben. In diesem Stadium enthält der Graph implizit alle möglichen Schaltungsabbildungen, die durch dieses Verfahren erzeugbar sind.

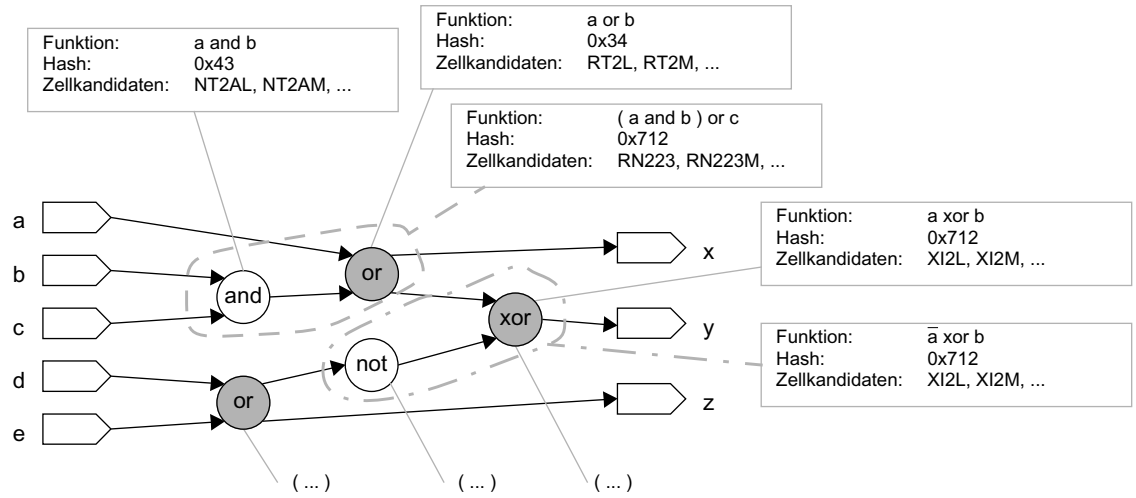


Abbildung 5.10 *Ermitteln der für die Abbildung der Schaltung geeigneten Zellen (ce81-Fujitsu-Prozess).*

Die Ergebnisse aller nachfolgenden Schritte hängen von der verwendeten Gütefunktion ab (siehe „Gütefunktion“ auf Seite 65). Da sich die Gütefunktion aufgrund veränderter Randbedingungen im Laufe des gesamten Prozesses mehrfach ändern kann, wird an diesem Punkt eine Kopie des Graphen erstellt, um die Berechnung weiterer Abbildungen mit geänderter Gütefunktion zu beschleunigen. Das feste Binden der Knoten an ein bestimmtes Gatter erfolgt in zwei Schritten:

1. Ausgehend von den Eingängen werden für jeden Knoten, für jede mögliche im Knoten abbildbare Zelle Platzbedarf, Strombedarf und Geschwindigkeit des gesamten Pfades bestimmt und daraus die Güte berechnet. Von den jeweils vorgelagerten Gattern werden die Werte des Gatters mit der besten Güte genommen.

In Abbildung 5.11 sind für zwei Knoten beispielhaft mögliche Zellen, ihre Eigenschaften und die sich daraus ergebende Güte (mit $g_x = a_x^{0,6} \cdot p_x^{0,3} \cdot t_x^{0,1}$) gegeben. Beim ersten Knoten (and) gibt es noch keine vorgelagerten Knoten. Dementsprechend unterscheiden

sich die Zellen nicht in ihrer Funktionalität, sondern nur in der Treiberstärke, was sich in diesem Fall nur in der Verzögerungszeit (t(ps)) niederschlägt. Für den oberen `or`-Knoten gibt es sowohl Einzelzellen (RT2L) als auch Komplexzellen (RN223L). Für die Zusammenfassung der Eigenschaften der Einzelzelle werden die Parameter der jeweils besten vorgelagerten Zelle, im Beispiel NT2AL, hinzuaddiert. Die Komplexzelle RN223L hat keine vorgelagerten Zellen, so dass die Eigenschaften der Zelle direkt übernommen werden können.

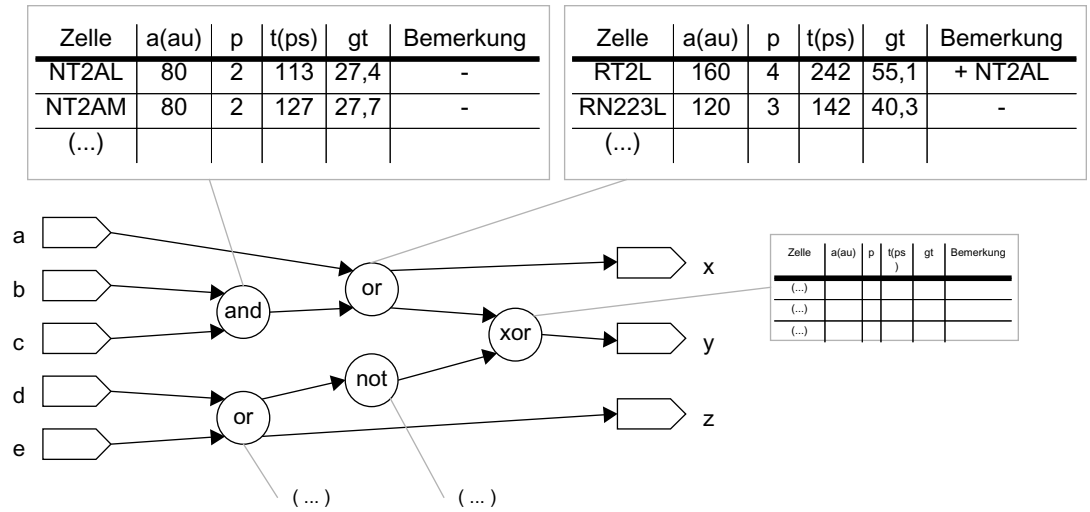


Abbildung 5.11 *Ermitteln der Güte jedes Knotens (ce81-Fujitsu-Prozess).*

Die so gefundenen Pfade durch den Graphen zu den Ausgängen sind in der Regel nicht miteinander vereinbar. Das bedeutet, einzelne Knoten können in unterschiedlichen Pfaden mit jeweils anderen Knoten zu Komplexgattern zusammengefasst worden sein. Die einzige Ressource, die zu diesem Zeitpunkt bestimmbar ist, ist die Verzögerungszeit.

2. Es werden die jeweils niedrigsten Verzögerungszeiten der möglichen Pfade jedes Signalausgangs verglichen. Der Pfad des Ausgangs mit der höchsten Verzögerungszeit wird realisiert, d.h. alle Knoten auf dem Pfad an das zugehörige Gatter fest gebunden.

Nach diesem Schritt wird zurück zum Punkt 1 gesprungen, wobei die bereits abgebildeten Knoten nicht erneut belegt werden. Der Zyklus wird solange durchlaufen, bis alle Knoten abgebildet sind. In Abbildung 5.12 ist das Ergebnis der Logikabbildung für das in Abbildung 5.10 gegebene Beispiel dargestellt. Es wurde nachbearbeitet, da die Logiksynthese keine Information über die Symbole der Zellen hat und diese insbesondere mit Blick auf umfangreiche Komplexgatter nicht ohne höheren zusätzlichen Aufwand automatisch erzeugt werden können.

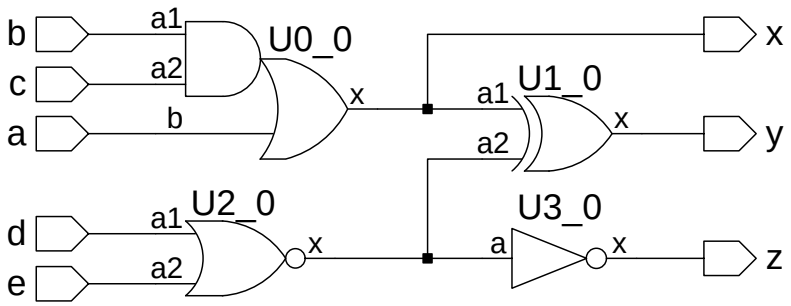


Abbildung 5.12 *Resultierende Schaltung nach erfolgter Logikabbildung.*

5.6 Arithmetikabbildung

In der Beschreibung der Vorgehensweise bei der Arithmetikabbildung wird stellvertretend für alle abbildbaren Funktionen (Subtraktion, Multiplikation, Vergleicher) nur auf die Addition eingegangen. Das Verfahren kann ausgehend von der Addition auf diese nicht explizit aufgeführten Funktionen übertragen werden.

5.6.1 Synchron

Im Gegensatz zur Logikabbildung ist bei der Arithmetikabbildung das testweise Abbilden der zur Verfügung stehenden Zellen auf die boolesche Funktion nicht effektiv. In Tabelle 5.4 ist beispielsweise eine Abschätzung für den Ressourcenbedarf der Implementierung eines Addierers (ausgehend von der Normalform) für die ce81-Technologie gegeben. In der vorgegebenen Umgebung kann für die 1-, 2- und 4-Bit-Addierer innerhalb der Logikabbildung noch eine aus jeder Sichtweise optimale Form gefunden werden. Die ce81-Technologie bietet 1-, 2- und 4-Bit-Addierer als eigene Zellen, deren Ressourcenbedarf diskret aufgebaut nicht unterschritten werden kann. Wird die Implementierung eines breiteren Addierers gesucht, ist das Ergebnis einer normalen Logiksynthese aufgrund der hohen Komplexität der generierten Schaltung nicht zu verwerten.

Die Arithmetikabbildung besteht aus einer Bibliothek von Spezialschaltungen, unter denen, ausgehend von den Anforderungen, die geeignete Schaltung ausgesucht wird. Die Entscheidung des Typs einer Schaltung ausschließlich auf Basis der gegebenen Randbedingungen ist möglich, aber nicht sinnvoll, da technologiespezifische Besonderheiten den tatsächlichen Ressourcenbedarf zugunsten einer anderen, eigentlich weniger geeigneten Schaltungstechnik verschieben können. Je nach vorgegebenen Parametern der Gütefunktion kann es auch innerhalb einer Schaltungstechnik zu unterschiedlichen Resultaten kommen, da funktional gleiche Zellen in der Regel mehrfach, d. h. mit unterschiedlichem Ressourcenbedarf, implementiert sind.

In Tabelle 5.4 ist als Beispiel der Ressourcenbedarf für Addierer unterschiedlicher Breite und Schaltungstechnik für die ce81-Technologie von Fujitsu gegeben. Die Werte für den 8-Bit-Addierer in Normalform sind dabei Abschätzungen. Die jeweils besten Ergebnisse einer bestimmten Ressource bei gleicher Breite des Addierers sind grau hinterlegt. In der letzten Spalte der Tabelle 5.4 ist die Güte der entsprechenden Schaltung für die willkürlich gewählte Optimierungsfunktion

$$g_x = a_x^{0,6} \cdot p_x^{0,3} \cdot t_x^{0,1} \qquad \text{(Gl. 6-6)}$$

gegeben. Entsprechend der vorgegebenen relativen Randbedingungen ist das primäre Optimierungskriterium der Platzbedarf. Strombedarf und Geschwindigkeit werden mit berücksichtigt, gehen jedoch nur zu einem geringen Anteil mit in die Gleichung ein. Wie aus Tabelle 5.4 ersichtlich, sind für unterschiedliche Addiererbreiten jeweils andere Typen optimal. Auch wenn mit Hinsicht auf den Platzbedarf, unabhängig von der Breite, immer der Carry-Ripple-Addierer die beste Lösung darstellt, sind, bedingt durch seine bis zu 5fach geringere Geschwindigkeit im Vergleich zu den anderen Addierertypen (abhängig von der Breite), andere Addierertypen besser geeignet. Bedingt durch die hohe Abhängigkeit der Ressourcenverhältnisse von der gewählten Technologie ist eine generische Entscheidung, bei welcher Gütefunktion welcher Schaltungstyp geeignet ist, praktisch nicht durchführbar. Einzige Ausnahme sind dual-rail-Addierer, welche aufgrund anderer Randbedingungen nur als Carry-Ripple sinnvoll einsetzbar sind (siehe „Arithmetikabbildung“ auf Seite 79). Auf die Berechnung der Gütefunktion für diesen Schaltungstyp kann dementsprechend verzichtet werden.

Tabelle 5.4: *Verschiedene technologieabhängige (ce81-) Addierertypen und ihre Charakteristika*

Typ	Breite	Geschwindigkeit (ps) [a]	Platzbedarf (au) [b]	Leistungsaufnahme [c]	Güte [d]
Normalform	1 [e]	163	160	4	53
	2 [c]	367	480	12	154
	4 [c]	683	2000	50	594
	8 [f]	~ 4500	~ $57 \cdot 10^6$	~ $1{,}5 \cdot 10^6$	~ $7{,}44 \cdot 10^6$
Carry-Ripple	8	1366	4000	100	1188
	16	2732	8000	200	2376
	32	5464	16000	400	4752
	64	10928	32000	800	9503
Carry-Skip/ Carry-Bypass	8	1810	4160	110	1287
	16	2070	8320	220	2434
	32	2330	16640	440	4597
	64	2850	33280	880	8753

Tabelle 5.4: *Verschiedene technologieabhängige (ce81-) Addierertypen und ihre Charakteristika*

Typ	Breite	Geschwindigkeit (ps) [a]	Platzbedarf (au) [b]	Leistungsaufnahme [c]	Güte [d]
Carry-Select (linear)[g]	8	943	8560	214	2270
	16	1203	17120	428	4341
	32	1723	34240	856	8396
	64	2763	68480	1712	16430
Carry-Select (Wurzel)	8	830	7000	175	1870
	16	960	13440	336	3413
	32	1350	22560	753	6139
	64	1750	36980	1057	9382
Look-Ahead [e]	8	1082	4160	104	1202
	16	1248	8400	210	2295
	32	1294	18240	459	4638
	64	1565	52600	1318	12250
Brent-Kung [e] [8]	8	970	4160	104	1189
	16	1100	8640	216	2325
	32	1230	17920	448	4533
	64	1360	37120	928	8818
dual-rail [h]	1	163	320	8	-
	2	367	960	24	-
	4	731	2240	56	-
	8	1001	4800	120	-
	16	1334	9920	248	-
	32	1661	20160	504	-
	64	1918	40640	1016	-

a. Summe der maximalen Verzögerungen der Einzelgatter auf dem kritischen Pfad für Low-Power Zellen ohne Last
b. area unit
c. Summe der durchschnittlichen Leistungsaufnahme aller Zellen im Netz relativ zur Leistungsaufnahme eines unbelasteten Low-Power-Inverters
d. Normiert; r_t=0,1; r_a=0,6; r_p=0,3
e. Addierer mit 1, 2 und 4 Eingängen sind für die ce81-Technologie als Zellen gegeben und werden nur einmal aufgeführt
f. unoptimiert
g. 4 Bit / Stufe
h. Die Leistungsaufnahmen sind nicht direkt mit denen der synchronen Addierer vergleichbar, da im Unterschied zu diesen nur bei tatsächlich durchzuführenden Additionen Schaltvorgänge stattfinden. Die Verzögerungszeit entspricht dem Durchschnitt über alle Additionen. Die tatsächlichen durchschnittlichen Werte liegen in der Regel darunter.

Sind für die Abbildung absolute Ressourcenbeschränkungen gegeben, werden die Implementierungen, welche diese verletzen, bei der Auswahl nicht mit berücksichtigt. Erst wenn keine Implementierung die Ressourcenbeschränkungen einhält, wird die Gütefunktion entsprechend modifiziert (siehe „Gütefunktion“ auf Seite 65). In der Regel sind bei geänderter Gütefunktion im Vergleich zur Logikabbildung nur geringe Veränderungen zu erwarten, da sich die ergebenden Änderungen nur aus dem Austausch von Gattertypen ergeben.

5.7 Dual-rail-encoding

Die Grundidee beim dual-rail-encoding ist, dass durch eine geeignete Verschlüsselung der zu verarbeitenden Daten, diese ihre eigene Gültigkeit signalisieren können. Die Analyse einer synchronen Netzliste kann keine Aussage treffen, wann welche Teile des Netzes reale Daten verarbeiten oder sich im Leerlauf befinden. Bei der direkten Abbildung wird deshalb davon ausgegangen, dass alle Signalleitungen innerhalb des synchronen Schaltwerks immer gültige Daten weiterleiten. Diese Annahme vereinfacht die direkte Abbildung, verhindert jedoch eine automatische Synthese von Schnittstellen. Dementsprechend werden diese durch die Synthese nicht generiert und müssen bei Bedarf anderweitig erzeugt werden.

Für die fehlerfreie Synthese von dual-rail-Schaltungen müssen im Vergleich zur synchronen Synthese zusätzliche Randbedingungen erfüllt sein:

- Synchronisationsschaltungen von unterschiedlichen Taktdomains können aufgrund deren Vielfältigkeit nicht sicher identifiziert werden. Dementsprechend ist nur eine Taktdomain innerhalb des Designs zulässig.
- Die Abbildung von Flipflops als Teil kombinatorischer Logik ist nicht möglich, da der entsprechende Kontext von der Synthese nicht erkannt und nachgebildet werden kann. Der Einsatz von Flipflops ist dementsprechend nur als Teil von Registern zulässig, die durch dasselbe Taktsignal synchronisiert sein müssen. Aus gleichen Überlegungen sind die dual-rail-Abbildungen von Latches, egal in welchem Kontext sie im Ausgangsnetz eingesetzt sind, nicht zulässig.
- Ausgehend von der Annahme, dass in einem abzubildenden synchronen Schaltwerk alle Daten gültig sind, befinden sich vorhandene asynchrone Rückkopplungen immer in einem Zustand, in dem gültige Daten verarbeitet werden. Ein solches Netz, als dual-rail-Schaltung realisiert, kann nicht initialisiert oder angehalten werden. Entsprechend ist die dual-rail-Abbildung von Zyklen innerhalb kombinatorischer Logik nicht möglich.

5.7.1 Logikabbildung

In der einfachsten Variante wird das von der Logiksynthese generierte Schaltnetz gatterweise in die entsprechende dual-rail-Schaltung übersetzt. Durch diesen Vorgang steigt die Komplexität der Schaltung in der Regel mit dem Faktor zwei. Als Beispiel ist in Abbildung 5.13 ein Volladdierer, in Abbildung 5.14 die zugehörige dual-rail-Schaltung nach der gatterweisen Übersetzung dargestellt. Einfache Grundgatter (and, or, nor, nand) sind in ihrer dual-rail-Form mit jeweils zwei Grundgattern realisierbar. Äquivalenz und Antivalenz benötigen

jeweils sechs Grundgatter, was ungefähr dem vierfachen Platz- bzw. Energiebedarf entspricht. Inverter (in der gegebenen Schaltung nicht vorhanden) werden durch das Vertauschen der beiden Signalleitungen realisiert und benötigen deshalb keine zusätzlichen Ressourcen.

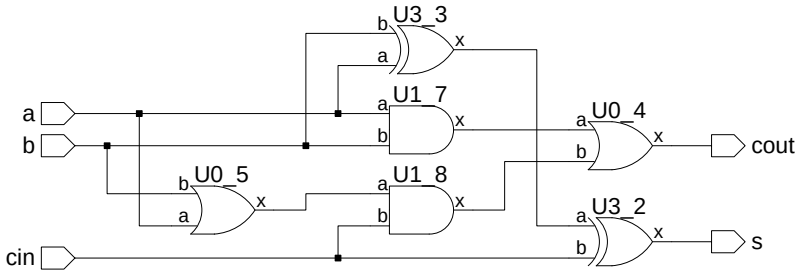


Abbildung 5.13 *Technologieunabhängiger Volladdierer.*

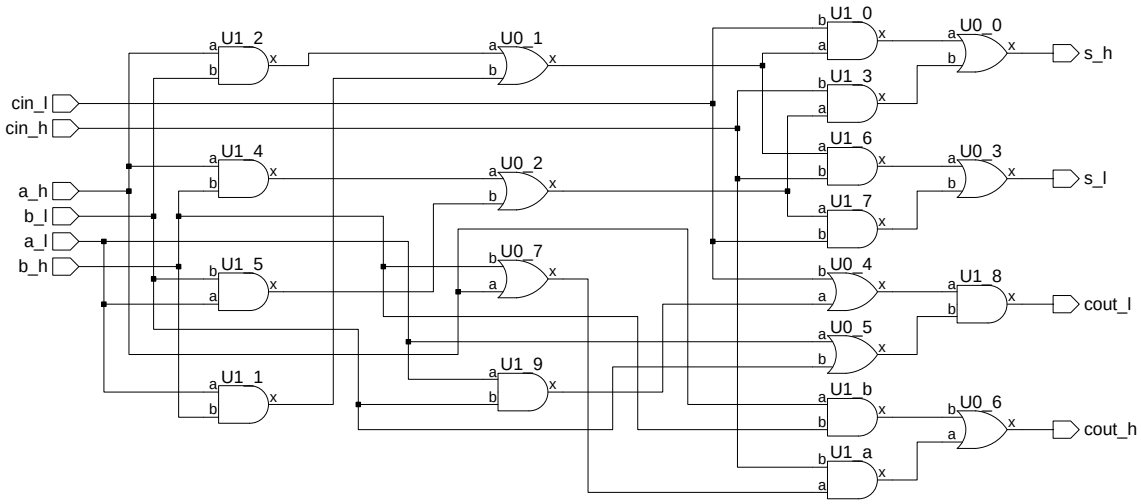


Abbildung 5.14 *Dual-rail-Volladdierer nach 1:1 Übertragung der Schaltung von Abbildung 5.13.*

Bedingt durch die Abbildung mit ausschließlich nichtnegierenden Gattern ist die resultierende Schaltung prinzipbedingt hazardfrei und kann ohne weitere Schritte übernommen werden.

Testweise durchgeführte Optimierungen, bei denen die vom gegebenen Schaltnetz implizit gegebenen Logikgleichungen zusammen mit ihren Negationen minimiert wurden [56,53], führten kaum zu besseren Ergebnissen. In Abbildung 5.15 ist das Ergebnis nach einer Optimierung mit misII aus dem Octtools-Paket [99] nach der Technologieabbildung dargestellt. Es unterscheidet sich hinsichtlich des Ressourcenbedarfs nicht von dem in Abbildung 5.14 dargestellten Netz.

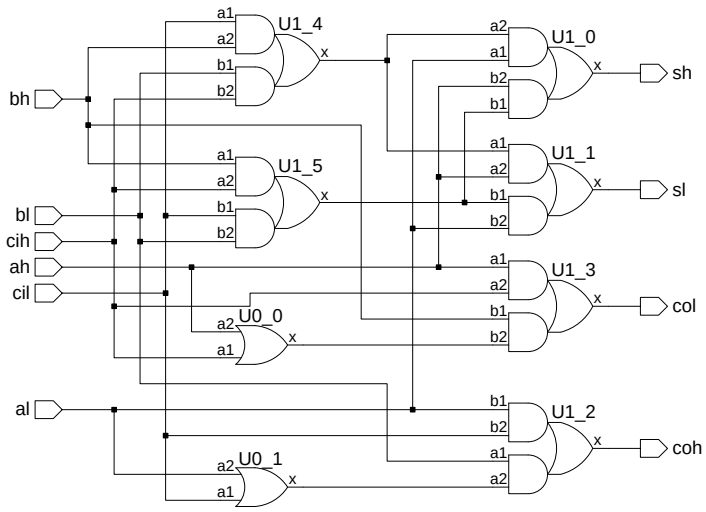


Abbildung 5.15 *Dual-rail-Volladdierer nach Minimierung durch externes Werkzeug, nach Technologieabbildung.*

5.7.2 Arithmetikabbildung

Im Unterschied zu synchronen Schaltungen, bei denen der worst-case-Fall die maximale Taktgeschwindigkeit eines Schaltnetzes bestimmt, zeigt eine dual-rail-Schaltung selbst das Ende einer Berechnung an. Dementsprechend muss für einen sinnvollen Vergleich der Schaltungstechniken die durchschnittliche Berechnungszeit herangezogen werden. Aus dem gleichen Grund ist es bei dual-rail-Schaltungen nicht effektiv, eine Optimierung hinsichtlich des worst-case-Falls durchzuführen, da dann in der Regel die durchschnittliche Berechnungszeit ansteigt.

Bei den in Schaltwerken am häufigsten benötigten arithmetischen Operationen (Addition, Subtraktion, Vergleich) wird beim Einsatz von Carry-Ripple-Schaltungen der kritische Pfad durch das zwischen den Einzeloperationen weiterzuleitende Carry-Signal bestimmt. Er geht nicht zwingend durch alle Einzeloperationen. Im Beispiel der Abbildung 5.16 ist der kriti-

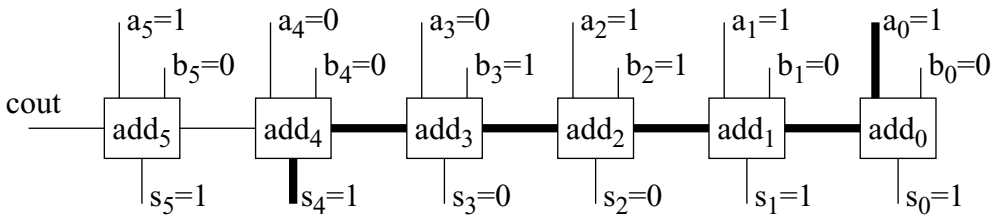


Abbildung 5.16 *Aufbau eines Carry-Ripple-Addierers mit kritischem Pfad für die aktuell dargestellte Operation (fett).*

sche Pfad für eine konkrete Addition fett eingezeichnet. Bei **add$_4$** wird für die Berechnung des Carry-out, der Carry-in am Eingang des Schaltnetzes nicht benötigt, da beide Summandenbits am Eingang null sind. Der theoretisch kritische Pfad wird also an dieser Stelle unterbrochen. Die durchschnittlichen Längen des kritischen Pfades $\mathbf{t_{adv}}$ über alle Additionen der Länge n lässt sich mit der Gleichung 6-7 berechnen. $\mathbf{t_{cc}}$ ist dabei die Zeit vom Carry-in zum Carry-out, $\mathbf{t_{ac}}$ der kritische Pfad von einem Summandenbit zum Carry-out und $\mathbf{t_{cs}}$ der kritische Pfad vom Carry-in zur Summe des verwendeten 1-Bit-Addierers.

$$t_{adv} = \left(\frac{12,72 \cdot n}{15,08 + n} - 1\right) \cdot t_{cc} + t_{ac} + t_{cs} \qquad \text{(Gl. 6-7)}$$

Die Funktion des Durchschnitts der längsten Carrys **cl** für **n** = [1 bis 64] ist in Abbildung 5.17, ausgewählte Funktionswerte sind in Tabelle 5.5 abgebildet. Die Funktion wurde empirisch durch das Aufstellen einer entsprechenden Statistik über alle möglichen Additionen der entsprechenden Breiten ermittelt.

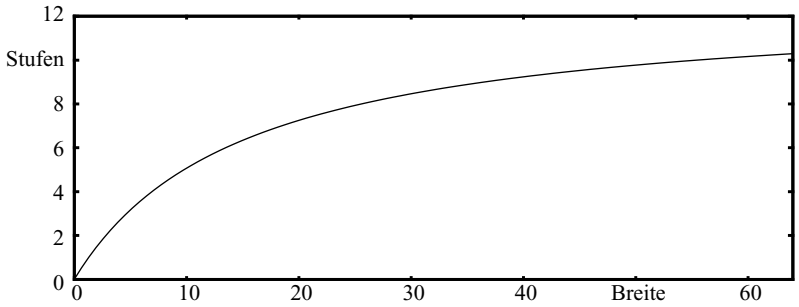


Abbildung 5.17 *durchschnittlich zu durchlaufende Addiererstufen des kritischen Pfades für unterschiedliche Addiererbreiten.*

Tabelle 5.5: *Ausgewählte Funktionswerte von Abbildung 5.17*

n	Stufen
1	0,75
2	1,5
4	2,66
8	4,4
16	6,55
32	8,65
64	10,29

Diese Werte sind jedoch eher als obere Schranke zu sehen, da in der Regel die durchzuführenden Additionen nicht gleichverteilt sind. Insbesondere wenn „kleine Zahlen“ zu addieren sind (z. B. bei einer Schleifenabarbeitung [62]), sinkt die tatsächlich benötigte Rechenzeit deutlich.

Technologieabhängige Implementierungen von dual-rail-Addierern unterschiedlicher Länge sind zum direkten Vergleich mit synchronen Addierern in Tabelle 5.4 auf Seite 75 angegeben. Dabei ist zu bemerken, dass die dual-rail-Implementierungen ohne Spezialzellen für die Addition auskommen müssen. Die Verzögerungszeiten der dual-rail-Addierer wurden mit der Gleichung 6-7 berechnet und sind dementsprechend durchschnittliche Zeiten.

Die Betrachtungen am Addierer lassen sich auf die Subtraktion und alle Vergleicher übertragen. Entsprechend kommen bei der dual-rail-Synthese keine anderen Schaltungstypen zum Einsatz.

5.7.3 Registerabbildung

Zusätzlich zur Speicherung von Daten haben die Register bei dual-rail-Schaltungen die Aufgabe, die Datenflüsse entsprechend Tabelle 3.10 auf Seite 32 zu synchronisieren. Grundlage für die verwendete Struktur ist die in Abbildung 5.18 dargestellte Registerstruk-

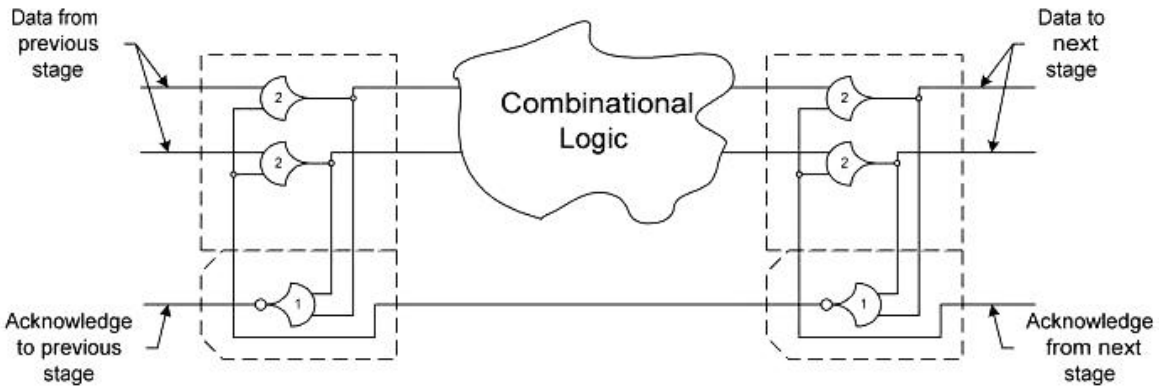


Abbildung 5.18 *NCL-Logik Register- und Synchronisationsstruktur (Quelle: [104]).*

tur der NCL-Logik [19]. In der Regel stehen die dargestellten Muller-C-Gates [47] in der verwendeten Zieltechnologie nicht zur Verfügung und müssen deshalb mit konventionellen Gattern nachgebildet werden. Eine von vielen Varianten ist in Abbildung 5.19 abgebildet. Die Speicherfunktion wird in dieser Variante von einem RS-Flipflop übernommen. Im Gegensatz zum Muller-C-Gate befindet sich dieses nach einem power-on nicht in einem definierten Zustand, weshalb zusätzlich eine Resetleitung vorzusehen ist. Die Resetleitung wird beim verwendeten Symbol bzw. dann auch in den weiteren Schaltungen aus Übersichtlichkeitsgründen nicht mit aufgeführt, da sie außer dem Setzen des initialen Zustands keine weitere Funktion hat und auch nicht sinnvoll in die eigentliche Logik mit eingebunden werden kann.

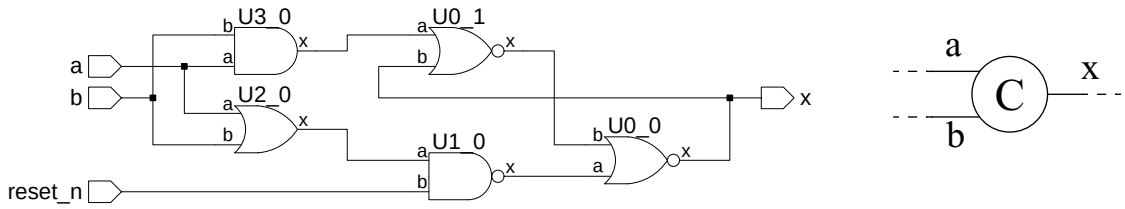


Abbildung 5.19 *Realisierung eines Muller-C-Gates mit konventionellen Gattern und zugehöriges Gattersymbol.*

Da für die Speicherung von einem dual-rail-Datum jeweils zwei Muller-C-Gates benötigt werden, wird für eine vereinfachte Darstellung im Weiteren ein sog. dual-rail-Flipflop (im folgenden DR-FF) eingeführt. Die Schaltung des DR-FF ist in Abbildung 5.20, das entsprechende Symbol in Abbildung 5.21 abgebildet. In Tabelle 5.6 ist die Übergangstabelle für das DR-FF gegeben.

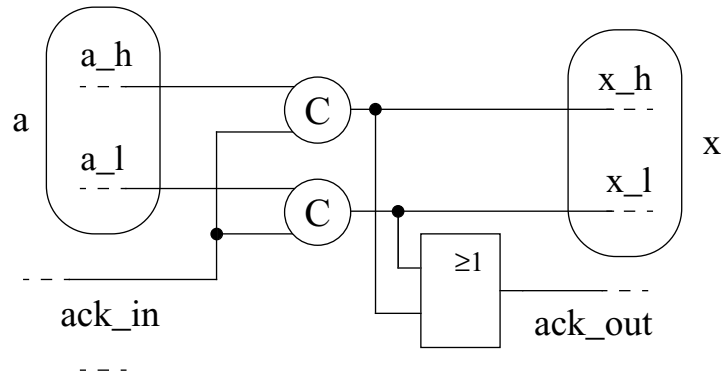


Abbildung 5.20 *Aufbau des dual-rail-Flipflops (DR-FF).*

.

Tabelle 5.6: *Übergangstabelle des DR-FF*

a	ack_in	x^{+1}	ack_out^{+1}
u	0	u	0
0	0	x	ack_out
1	0	x	ack_out
u	1	x^a	ack_out^a
0	1	0	1
1	1	1	1

a. i. d. R.: $x^{+1} = u$; $ack\_out^{+1} = 0$.

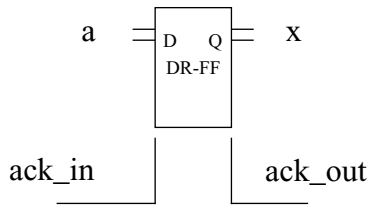


Abbildung 5.21 *Schaltsymbol des DR-FF.*

Die technologieunabhängige Struktur der Register mit eingebetteter kombinatorischer Logik ist in Abbildung 5.22 gegeben. Das acknowledge-Signal (low aktiv) für die vorhergehende Stufe wird nur dann gesetzt, wenn alle Daten am Eingang des Registers gültig sind. Dies bewirkt das Löschen der Register in der vorhergehenden Stufe, um diese für die nächste Datenwelle zu initialisieren. Ist das gesamte Register gelöscht, wird auch das acknowledge-Signal zurückgenommen, wodurch die vorhergehende Stufe für die Aufnahme von neuen Daten freigeschaltet wird. Das acknowledge-Signal wird in einem RS-Flipflop gesichert, da dieses, auch wenn es aus mehreren Gattern zusammengesetzt werden muss, zumeist günstiger im Ressourcenbedarf ist als ein vergleichbares D-Flipflop. Eine eigene Resetleitung ist nicht notwendig, da der Initialzustand der Muller-C-Gates ggf. das Löschen des Flipflops bewirkt.

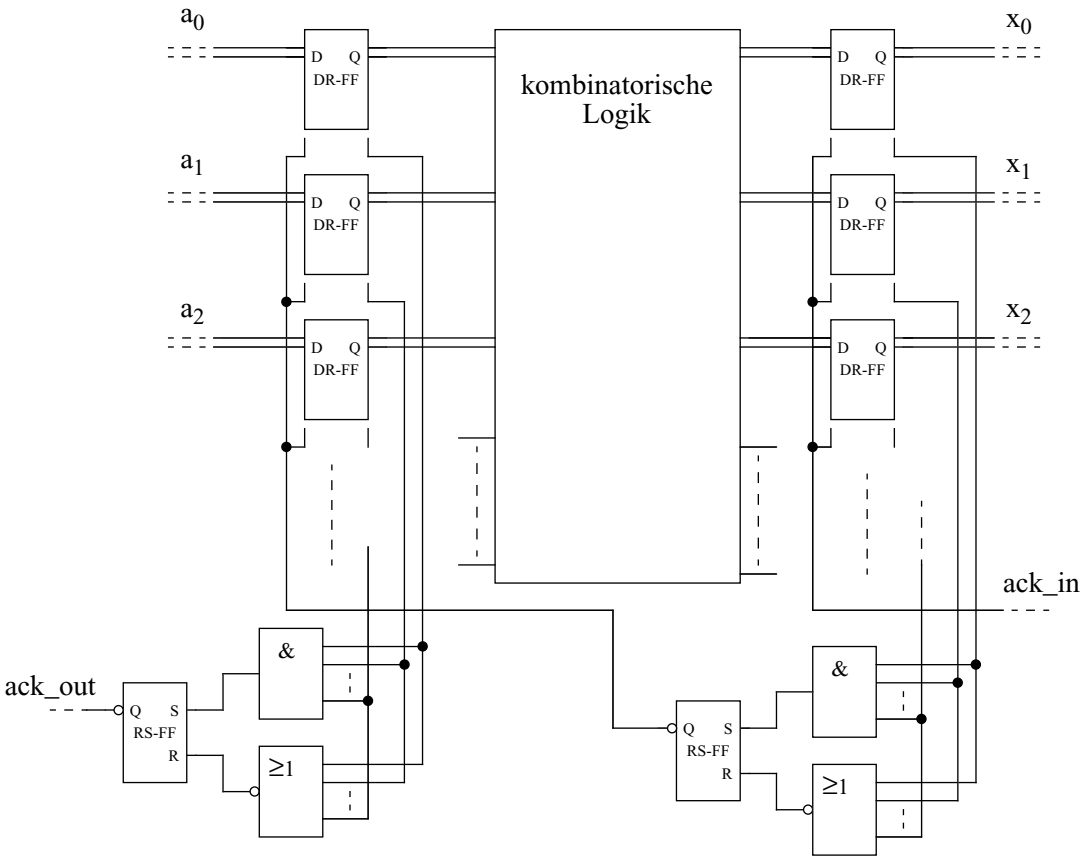


Abbildung 5.22 *Technologieunabhängige dual-rail-Registerstruktur entsprechend Abbildung 5.18.*

6 Datenbank A-RTL

Die Datenbank A-RTL (Arithmetic-RTL) stellt die grundlegende Datenorganisationsstruktur sowohl für die Logiksynthese SibaS als auch für die Technologieabbildung ALTeM zur Verfügung und stellt gleichzeitig die Schnittstelle zwischen den beiden Werkzeugen da. Die Datenbank ist als relationale Datenbank in C++ implementiert, wobei die in C++ gegebenen Klassen- und Vererbungsmechanismen voll ausgeschöpft werden. Neben den reinen Synthesedaten werden auch weitere Strukturen beispielsweise für die Schaltungsvisualisierung und Speicherverwaltung zur Verfügung gestellt. Die Datenbank besteht aus 43 Grundklassen, wobei im Folgenden hauptsächlich auf die für den Datenaustausch zwischen Logiksynthese und Technologieabbildung relevanten Klassen bzw. Objekte näher eingegangen wird.

6.1 Übersicht

Aus Synthesesicht werden die Daten innerhalb der A-RTL-Datenbank (wie in Abbildung 6.1 dargestellt) implizit in drei Schichten gegliedert. Die VHDL-Ebene wird ausschließlich von der Logiksynthese genutzt und enthält alle im Kontext der VHDL-Synthese benötigten Daten. Die in der generischen Ebene von der Logiksynthese generierten Objekte bilden die erzeugte generische Netzliste ab, die als Eingangsdaten von der Technologieabbildung benötigt werden. Innerhalb der Technologie-Ebene wird die generische Netzliste in eine technologieabhängige Liste umgewandelt.

Im Folgenden werden die einzelnen Schichten aus Sicht der jeweiligen Werkzeuge beschrieben. Es wird dabei ausschließlich auf die Struktur eingegangen. Die zugrunde liegenden Algorithmen sind in den jeweiligen Kapiteln aufgeführt.

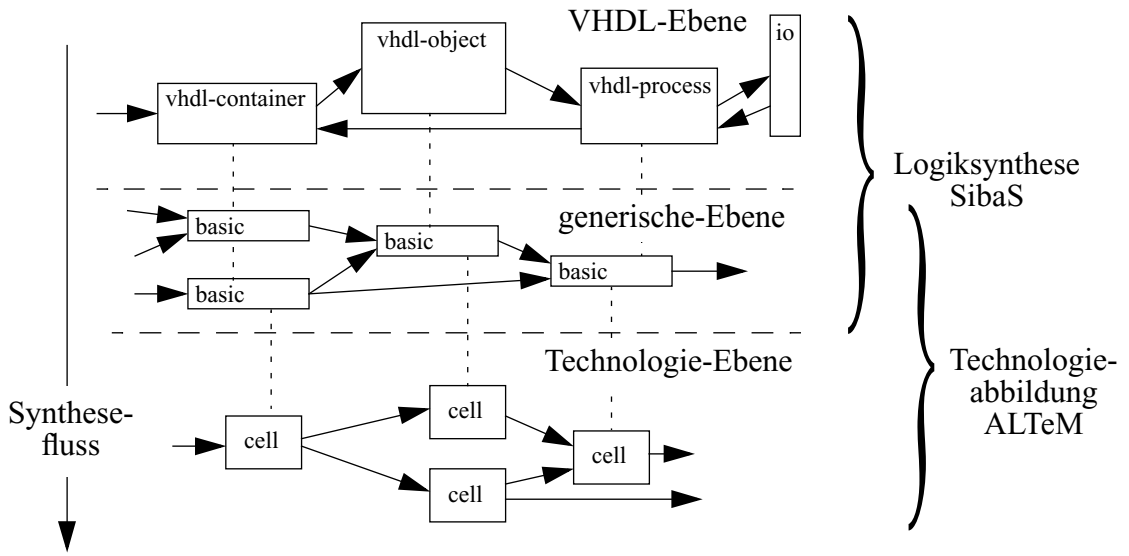


Abbildung 6.1 *Syntheseebenen innerhalb der A-RTL-Datenbank.*

6.2 Logiksynthese

Ausgehend von den gegebenen VHDL-Beschreibungen werden innerhalb der A-RTL-Datenbank die durch den VHDL-Parser generierten Übersetzungsbäume gespeichert. Bei dem Entwurf der Datenbank wurde darauf geachtet, Strukturen so, wie sie in der Sprache VHDL verankert sind, zu erhalten. So gibt es für die unterschiedlichen Beschreibungsformen (wie beispielsweise VHDL-Entities, Komponenten und Signale) jeweils eigene vorgegebene Datenstrukturen in der A-RTL-Datenbank, die im Folgenden genauer beschrieben werden. Dieses Vorgehen vereinfacht zum einen den Datenbankentwurf an sich, da implizit vorhandene Strukturen nur übernommen werden müssen, wirkt sich zusätzlich auch positiv auf ggf. durchzuführende Analysen (z. B. im Kontext einer Fehlersuche) aus, da eine Zuordnung von Objekt und zugehörigem Sprachelement der Ausgangsbeschreibung sehr einfach durchzuführen ist. Auf eine vollständige Abbildung der Namen von VHDL-Sprachelementen wurde verzichtet, da diese zum Teil stark kontextbezogen sind oder völlig fehlen. So steht beispielsweise für die Instanzbildung einer VHDL-Entity kein eigenes VHDL-Schlüsselwort zur Verfügung.

Für jede Instanz einer VHDL-Entity wird innerhalb des Syntheseprozesses ein A-RTL-Objekt `vhdl_container` generiert, welches den Ausgangspunkt für die zu generierende Netzliste darstellt. Dieses enthält entsprechend dem in Abbildung 6.2 dargestellten Aufbau u. a. folgende Elemente:

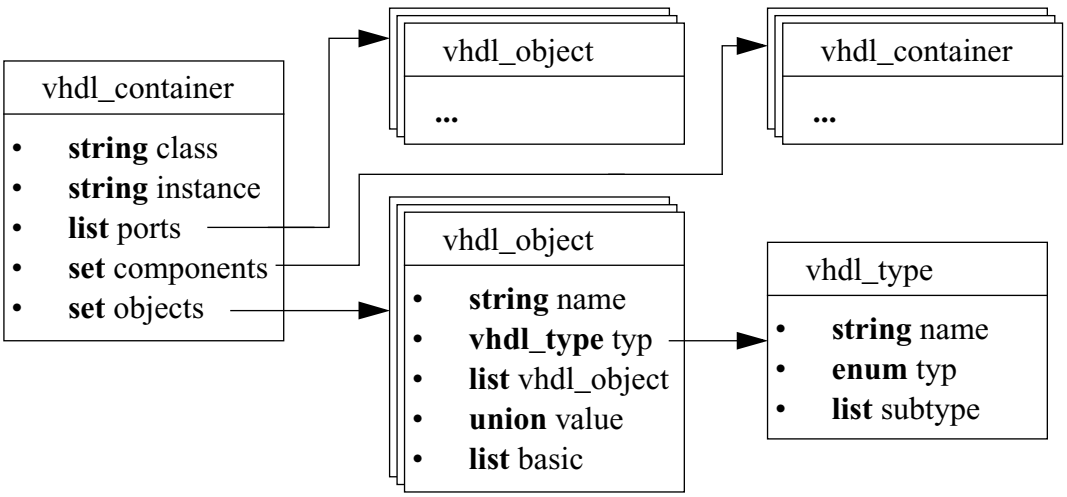


Abbildung 6.2 *Repräsentationsstruktur von VHDL-Entities als* `vhdl_container` *innerhalb der A-RTL-Datenbank.*

- Der Klassenname `class` entspricht dem Typnamen der zugehörigen VHDL-Entity.
- Der Instanzname `instance` entspricht dem Instanznamen der aktuellen Entity zuzüglich eines Präfixes, der gleich dem Instanznamen der Elterninstanz ist. Durch dieses Vorgehen ist zum einen jeder Instanzname eines `vhdl_container` innerhalb der A-RTL-Datenbank eindeutig, zum anderen kann so das Elternobjekt direkt über den Instanznamen identifiziert werden.

- Die ungeordnete Liste `components` enthält alle zu den in der VHDL-Entity instantiierten VHDL-Entities zugehörigen `vhdl_container`.
- Zu jedem in der VHDL-Entity deklarierten VHDL-Signal wird ein eigenes `vhdl_object` in der ungeordneten Liste `objects` gesichert. Der Begriff VHDL-Signal steht im Folgenden stellvertretend für alle innerhalb einer VHDL-Entity deklarierbaren Signale, Variablen, Records oder Arrays.
- Die geordnete Liste `ports` enthält (entsprechend der Reihenfolge der VHDL-Signale der Port-Deklaration der VHDL-Entity) die jeweils zugehörigen Instanzen aus der Liste `objects`. Sie erfüllt so die gleiche Funktion wie die in einer VHDL-Entity-Deklaration aufgeführte Portliste.

Die aus den Deklarationen der VHDL-Entity generierten `vhdl_object`-Objekte stellen die Schnittstelle zu der durch die Logiksynthese SibaS zu generierenden generischen Netzliste dar. Sie besteht u. a. aus folgenden Teilen:

- Der Instanzname `name` entspricht dem Namen des VHDL-Signals zuzüglich des Instanznamens des zugehörigen Elternobjektes `vhdl_container`.
- Anhand des Objektes `vhdl_type` wird der Typ des zugehörigen VHDL-Signals näher spezifiziert. `vhdl_type` selbst ist ebenfalls ein Objekt, welches sich bei komplexen Typen, wie z. B. VHDL-Records, aus weiteren Typspezifikationen zusammensetzen kann.
- Analog zu `vhdl_type` wird auch das `vhdl_object` selbst aus weiteren `vhdl_object`-Objekten für die Repräsentation von komplexen VHDL-Signalen aufgebaut. Die zugehörigen Objekte werden in einer geordneten Liste gesichert.
- Der Eintrag `value` ist als union (Struktur die ein Objekt unbekannten Typs aufnehmen kann) organisiert und kann einen in VHDL für das zugehörige VHDL-Signal spezifizierten Defaultwert oder eine Konstante aufnehmen. Der tatsächliche Typ des in der union gehaltenen Objektes wird durch den `vhdl_type` spezifiziert.
- Die geordnete Liste der `basic`-Objekte stellt die Schnittstelle zur generischen Netzliste dar. Jedes `basic` steht für eine Signalleitung der generischen Netzliste. Jedes `vhdl_object` enthält genau so viele `basic`-Objekte wie Bits notwendig sind, um alle möglichen Zustände des zugehörigen VHDL-Signals zu verschlüsseln.

Nach dem vollständigen Aufbau aller `vhdl_container` innerhalb der A-RTL-Datenbank werden die `basic`-Objekte entsprechend den Anweisungen der zugrunde liegenden VHDL-Entity untereinander verknüpft, so dass nach dem Auswerten aller VHDL-Anweisungen die generische Netzliste als Verknüpfung der `basic`-Objekte vorliegt.

Der Aufbau eines `basic`-Objektes und die grundlegende Struktur zur Repräsentation der generischen Netzliste innerhalb der A-RTL Datenbank ist in Abbildung 6.3 dargestellt.

- Der `name` leitet sich aus dem Instanznamen des Elternobjekts `vhdl_object` ab. In Hinsicht auf den Syntheseprozess hat er keine Funktion. Er vereinfacht jedoch die Fehlersuche, da eine aus dem `basic` erzeugte Signalleitung diesen Namen übernimmt, welcher dann bei der Schaltungsvisualisierung das Identifizieren der einzelnen Signalleitungen erheblich vereinfacht. Muss ein `basic`-Objekt unabhängig von der VHDL-

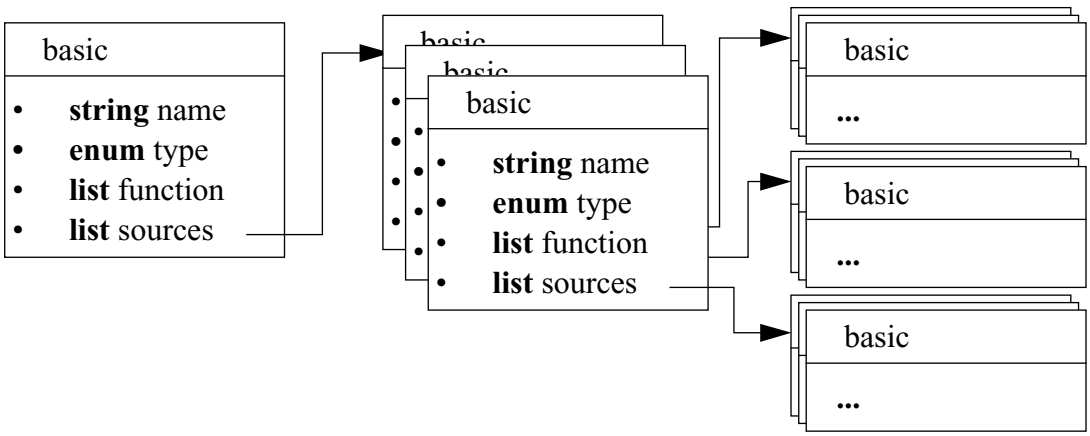


Abbildung 6.3 *Repräsentationsstruktur von* `basic`*-Objekten und die Vernetzung von* `basic`*-Objekten untereinander zur Bildung von generischen Netzlisten innerhalb der A-RTL-Datenbank.*

Beschreibung bei z. B. der Schaltungsoptimierung erzeugt werden, erhält es einen allgemein eindeutigen Namen.

- Der Eintrag `type` spezifiziert den Typ des `basic`-Objekts. Dieser wird beim Syntheseprozess gesetzt und ist zum Initialisierungszeitpunkt als „unbekannt“ markiert. Er spezifiziert den Gattertyp (Funktion/Speicher/IO) und beeinflusst so die weiteren Einträge des Objektes hinsichtlich ihrer Interpretation.
- Der Eintrag `function` besteht aus einer Liste boolescher Werte und spezifiziert je nach `type` die durch das `basic` repräsentierte Gatter. Im Kontext eines einfachen Funktionsgatters sind es die Funktionswerte der dem Gatter zugehörigen Funktionswerte. Im Kontext eines Speicherelements werden durch die `function` die Art des Speichers und die optional vorhandenen Steuersignale genauer spezifiziert.
- Die geordnete Liste `sources` hält die Referenzen der `basic`-Objekte, die dem aktuell repräsentierten Gatter vorgeschaltet sind.

Nach erfolgter Logiksynthese bildet die Gesamtheit aller `basic`-Objekte die generische Netzliste, die innerhalb der Square-Dance-Entwurfsumgebung visualisiert werden kann bzw. Ausgangspunkt für die Technologieabbildung ist.

6.3 Technologieabbildung

Innerhalb der Technologieabbildung wird ausgehend von der generischen Netzliste innerhalb der A-RTL-Datenbank ein Netz erzeugt, welches alle möglichen durch die Technologieabbildung ALTeM erzeugbaren Netze enthält. Wie in Abbildung 6.4 dargestellt, wird ausgehend von den `basic`-Objekten mindestens je ein `extended`-Objekt erzeugt. Die Klasse der `extended`-Objekte ist von `basic` abgeleitet und kann somit die gleichen Informationen abbilden. Zusätzlich enthält jedes `extended` folgende Informationen:

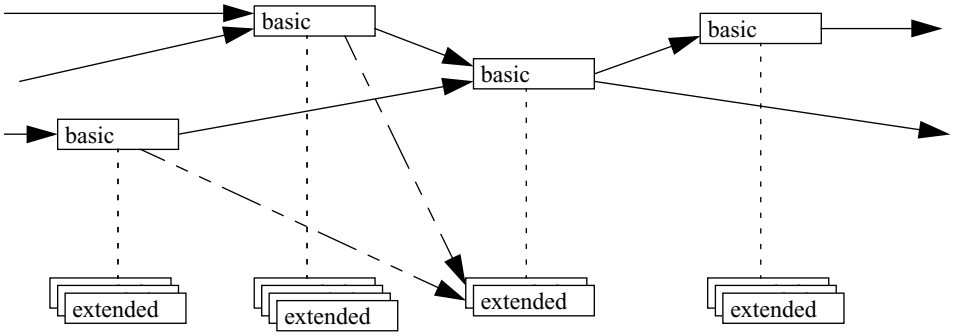


Abbildung 6.4 *Ableitung der* `extended`*-Objekte aus der generischen Netzliste.*

- Der Eintrag `cell_name` verweist auf eine technologiespezifische Zelle, die durch das Objekt repräsentiert wird.
- Der `quality`-Eintrag wird für die Bewertung des Netzes benötigt und entspricht der erreichbaren Güte des aktuellen Schaltnetzes bis zu diesem Punkt.
- Gatter mit mehreren Ausgängen werden durch entsprechend viele `extended`-Objekte abgebildet. Da der Ressourcenbedarf des Gatters jedoch nicht teilbar ist, kennen alle `extended`-Objekte diejenigen Objekte, welche dieselbe Ressource belegen. Die Liste wird in jedem Objekt im Eintrag `brothers` gesichert.

Die `extended`-Objekte sind untereinander nicht vernetzt, sondern verweisen bei den Eingangssignalen (siehe Eintrag `sources` der `basic`-Objekte) auf die jeweiligen Objekte der generischen Liste (in Abbildung 6.4 aus Übersichtlichkeitsgründen nur für ein `extended`-Objekt dargestellt).

7 Implementierung

7.1 Überblick

Das Synthesepaket Square-Dance besteht aus zwei separaten Werkzeugen, deren Aufbau in Abbildung 7.1 dargestellt ist. Aufgabe des library_compilers ist es, die gegebenen Zellbeschreibungen der Zieltechnologie zu analysieren und sie in ein internes Format (sdlc) umzuwandeln. Der design_compiler übernimmt die Logiksynthese und die Technologieabbildung der gegebenen Projektdaten. Beide Werkzeuge benutzen denselben VHDL-Compiler, der als separate Programmbibliothek vorliegt.

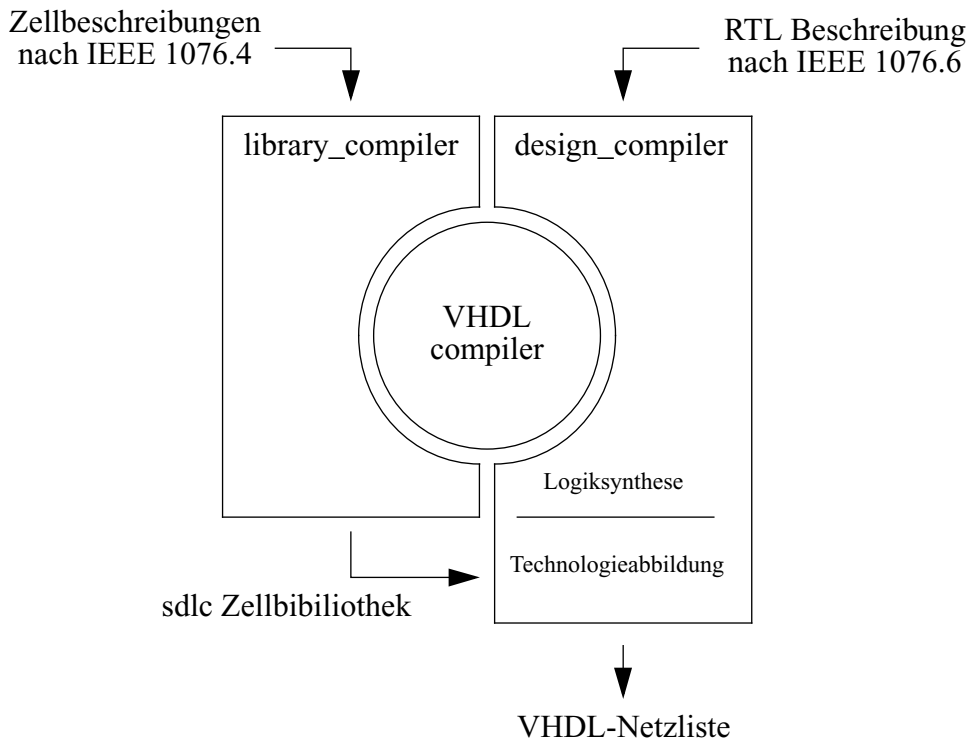


Abbildung 7.1 *Schematischer Aufbau des Synthesepaketes Square-Dance.*

Square-Dance ist in der objektorientierten Sprache C++ [70,82] implementiert und umfasst in der aktuellen Version 52.593 Zeilen Quelltext. Der VHDL-Parser wurde mit Hilfe von für dieses Projekt modifizierten Versionen der Lexer- und Parsergeneratoren bison++ und flex++ [90] aus der 4.223 Zeilen umfassenden VHDL-Grammatik automatisch generiert. Die Benutzeroberflächen wurden mit dem QT-Toolkit der Firma Trolltech [105] entwickelt. Durch die weitgehende Plattformunabhängigkeit von QT kann das Entwicklungspaket auf den gängigsten Plattformen (Unix/X11, Windows, MacOS X) übersetzt werden. Für die Schaltungsvisualisierung wurde die NlviewQT-Bibliothek der Firma Concept Engineering GmbH [85] eingesetzt. Diese ist ausschließlich vorübersetzt für ausgewählte Plattformen

(Windows, Solaris, Linux) verfügbar und kann auf Kosten der Visualisierung aus den Quellen herausgelöst werden, um beispielsweise den Einsatz von Square-Dance unter MacOS X zu ermöglichen.

7.2 library_compiler

Für den effizienten Einsatz der Technologieabbildung werden die Zellen nach Funktion geordnet und in einem proprietären Datenbankformat verschlüsselt, welches direkt in der A-RTL-Datenbank abgebildet werden kann und so einen schnellen Zugriff auf bestimmte Zellen erlaubt. Für jede gegebene Zieltechnologie muss die Berechnung der zugehörigen Datenstruktur prinzipiell nur einmal durchgeführt werden, da diese in der Regel keinen Änderungen unterworfen ist. Dementsprechend bietet es sich an, diese Berechnung in einem eigenen Programm, dem library_compiler, zu implementieren, welches bei einer neuen gegebenen Zieltechnologie nur einmal auszuführen ist, um die notwendigen Berechnungen durchzuführen und die Datenstruktur zu erstellen.

7.2.1 Funktionalität

Die Übertragung der Zellen in das Datenbankformat erfolgt linear in folgenden Schritten:

1. Die nach dem Standard IEEE 1076.4 spezifizierten Zellbeschreibungen werden eingelesen und dabei auf Syntax und zu erfüllende Randbedingungen überprüft. So sind beispielsweise generische Beschreibungen oder algorithmische Funktionsspezifikationen für Zellbeschreibungen nach Standard IEEE 1076.4 nicht zulässig.

2. Aus der Entityspezifikation werden die verwendeten Ports bestimmt und die in der Zellbeschreibung verwendeten Zeiten ermittelt.

3. Es werden die Funktionen ermittelt und in Funktionstabellen umgewandelt.

4. Die Funktionstabellen werden nach dem Entropieverfahren in eine eindeutige Form gebracht (siehe „Eindeutige Repräsentation boolescher Funktionen“ auf Seite 24) und der dazugehörige Hashwert bestimmt.

5. Anhand des Hashwertes und optional vom Benutzer vorgegebener Eigenschaften wird der Typ der Zelle ermittelt und entsprechend in die A-RTL-Datenbank eingetragen.

Nach diesem Schritt sind alle Zellen entsprechend der in Abbildung 7.2 dargestellten Struktur organisiert. Die Zellen werden nach Funktionalität in unterschiedliche Gruppen geordnet:

- debug: Spezialzellen für den Funktionstest des Bausteins nach der Fertigung, wie z. B. JTAG, Boundary scan [75],
- buffer: Spezielle Treiberzellen für Signale mit hoher Last bzw. für Taktbäume,

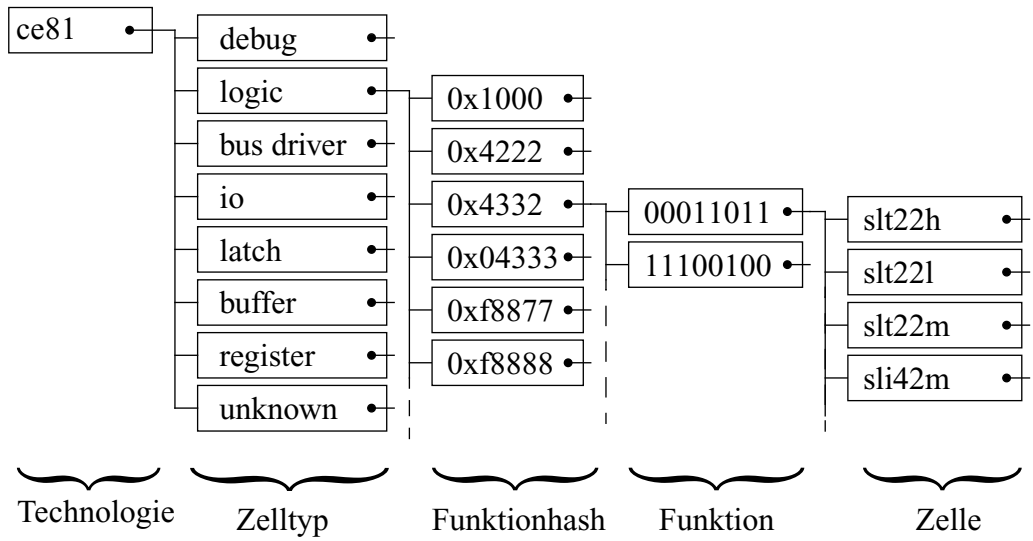


Abbildung 7.2 *Anordnung der Zellen in der Zelldatenbank.*

- logic: Zellen für die logische Verknüpfung von Signalen einschließlich arithmetischer Funktionen,
- bus driver: Zellen für den Aufbau von Busstrukturen auf dem Baustein, wie z. B. Tristatetreiber,
- io: Zellen für die Datenschnittstelle nach außen (Pads),
- latch: Pegelgetriggerte Zellen zur Speicherung von Daten (Latches),
- register: flankengetriggerte Zellen zur Speicherung von Daten (Flipflops, RAM)
- unknown: Zellen, die in keine andere Kategorie eingeordnet werden konnten, wie z. B. Zellen für die Stromversorgung, Takterzeugung, Synchronisation, Strukturen zum Test bei der Fertigung, Dummyzellen oder Zellen, die aus anderen Gründen nicht korrekt zugeordnet werden konnten.

Diese Einteilung reduziert die Suche nach einer passenden Zelle innerhalb der Technologieabbildung, da entsprechend der Anforderungen nur in der entsprechenden Kategorie gesucht werden muss. Ein Auslassen dieser Stufe verschiebt das Problem der Klassifizierung der Zelle in die Technologieabbildung. Die Klassifizierung ist jedoch in jedem Fall notwendig, da beispielsweise io-Zellen trotz gleicher Funktionalität nicht als Buffer eingesetzt werden können. Die nächste Stufe, die Ordnung der Zellen nach ihrem Hashwert, wird ausschließlich aus Effizienzgründen eingesetzt. Der Hashwert einer Funktion erlaubt eine schnellere Suche innerhalb der Bibliothek. Das Berechnen des Hashwertes einer gegebenen Funktion hat einen konstanten und, im Vergleich zur direkten Suche nach einer bestimmten Zelle, vernachlässigbaren Aufwand. In der vorletzten Ebene sind alle Zellen gleicher Funktion der aktuellen Kategorie gegeben. In der Regel handelt es sich hier um Zellen, die in unterschiedlichen Treiberstärken vorliegen. Die Wahl, welche Zelle schließlich zum Einsatz kommt, wird entsprechend der gegebenen Randbedingungen, wie zum Beispiel Optimierungskriterien, getroffen.

7.2.2 Bedienung

Die Steuerung des library_compilers erfolgt primär über die zugehörige Oberfläche. Der nach dem Starten des Programms angezeigte Dialog (siehe Abbildung 7.3) ermöglicht direkt das Öffnen einer vorhandenen Zelldatenbank bzw. das Erzeugen einer neuen. Beim Anlegen einer neuen Zelldatenbank muss in dem dafür zuständigen Fenster (siehe Abbildung 7.4) der Technologiename angegeben werden. Optional kann eine beliebige zusätzliche Beschreibung hinzugefügt werden. Als Letztes wird die Angabe eines Dateiverzeichnisses benötigt, in der sich die Zellbeschreibungen im VHDL-Format nach Standard IEEE 1076.4 befinden. Nach der Bestätigung der Angaben wird das angegebene Verzeichnis rekursiv durchsucht und alle vorkommenden VHDL-Beschreibungen werden automatisch übersetzt. Für jede in einer Datei gefundenen VHDL-Entity wird in der Datenbank ein neuer Eintrag einer Zelle erzeugt und entsprechend eingeordnet.

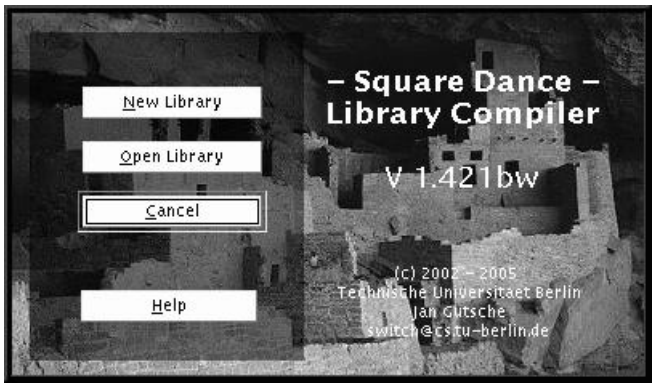


Abbildung 7.3 *Splash-Screen des library_compilers.*

Abbildung 7.4 *Fenster zum Erzeugen einer neuen Bibliothek innerhalb des library_compilers.*

Insbesondere bei sehr umfangreichen Zellbibliotheken kann der Übersetzungsvorgang einige Minuten dauern. Ein in dieser Zeit angezeigtes Fenster informiert über den Fortschritt der Verarbeitung und erlaubt es auch, diese zu unterbrechen.

Nach dem Übersetzungsvorgang bzw. nach dem Einlesen einer bereits existierenden Zelldatenbank wird das in Abbildung 7.5 dargestellte Hauptfenster angezeigt. Es unterteilt sich in drei Bereiche:

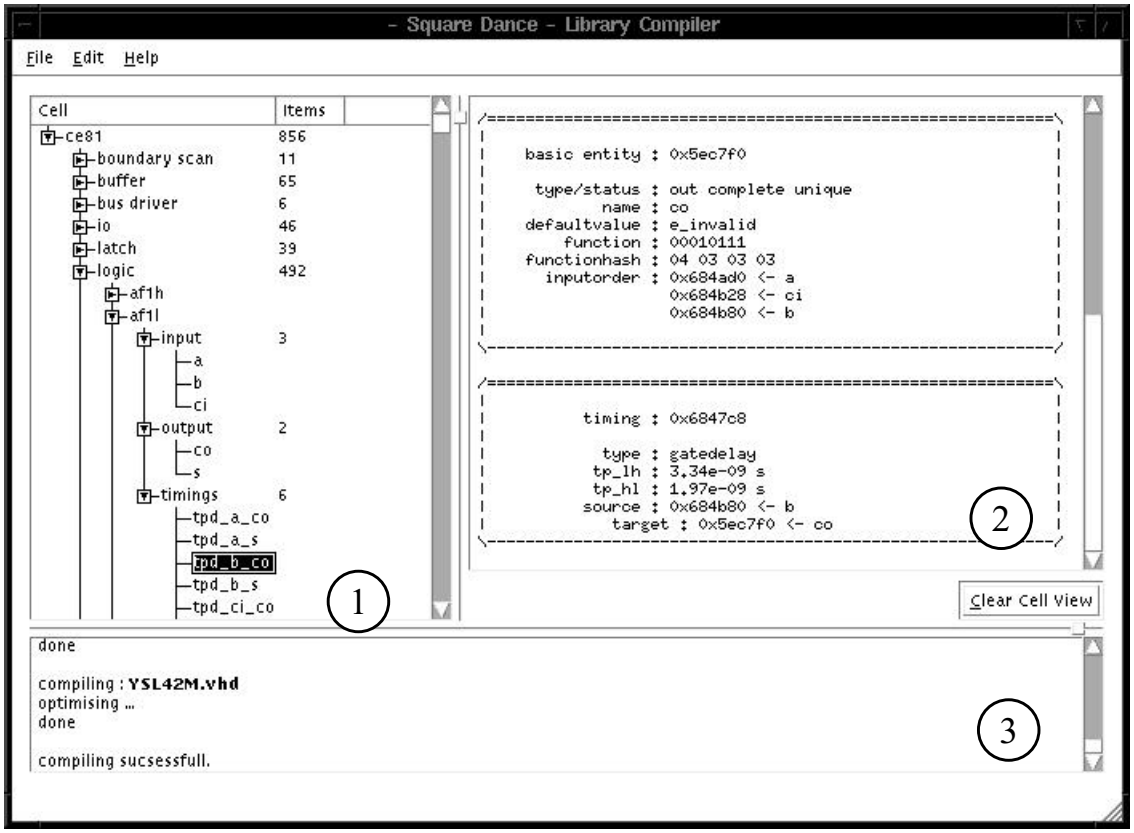


Abbildung 7.5 *Hauptfenster des library_compilers.*

- Im ersten Bereich liegt der sog. Cellbrowser. In diesem sind die Verwaltungsstruktur der Datenbank und die in ihr eingeordneten Zellen in einer Baumstruktur dargestellt. Es können mehrere Zellbibliotheken gleichzeitig geladen und dargestellt werden, um einen direkten Vergleich zu ermöglichen. Die Zellen sind nach Typen geordnet. Die Auswahl einer Zelle ermöglicht den Zugang zu zusätzlichen Informationen, die dann im zweiten Bereich, dem Cellview-Bereich dargestellt werden können. Die Schnittstellen werden nach Ein- und Ausgängen geordnet und zusammen mit allen verwendeten Zeitparametern angegeben.

- Der zweite Bereich ist der sog. Cellview-Bereich. In diesem werden weitergehende Informationen zum jeweils angewählten Eintrag im ersten Bereich des Fensters dargestellt.
  Bei der Auswahl eines Ausgangsports einer Zelle im Cellbrowser werden neben Debuginformationen die Abhängigkeiten von den ebenfalls dargestellten Eingangsports der Zelle (Funktion) mit zugehörigem Hashwert aufgeführt. Die angegebene Funktion kann als Funktionsspalte einer Wertetabelle interpretiert werden. Die angegebenen Eingangssignale sind aufsteigend nach Wertigkeit sortiert und entsprechen somit den Ein-

gangssignalen der impliziten Wertetabelle. Übersteigt die Anzahl der Eingangssignale einen bestimmten Wert, wird die Funktion aus Übersichtsgründen als Hexadezimalzahl dargestellt.
Die Auswahl eines Zeitparameters zeigt Start und Endpunkt des Signals mit den zugehörigen Zeiten (tp_{hl}; tp_{lh}) durch die jeweilige Zelle an.

- Im dritten Bereich, dem sog. Message-Bereich, werden Meldungen des library_compilers an den Benutzer ausgegeben. Hierzu gehören hauptsächlich Protokollinformationen über beispielsweise die durch das Programm ausgeführten Aktionen und Fehlermeldungen.

Als letztes muss die fertig übersetzte und ggf. verifizierte Zelldatenbank für die Verwendung im design_compiler in einer Datei gespeichert werden. Die Organisationsstruktur bleibt dabei vollständig erhalten, so dass die Datenbank nach dem Einlesen durch den design_compiler sofort betriebsbereit ist.

7.3 design_compiler

Der design_compiler vereinigt die Logiksynthese und Technologieabbildung zu einem einzigen Werkzeug. Aufgabe ist die Umsetzung der vorgegebenen abstrakten VHDL-Beschreibung in eine technologieabhängige Netzliste unter Berücksichtigung von vorgebbaren Randbedingungen für Platzbedarf, Geschwindigkeit und Verlustleistung. Die Oberfläche hat einen ähnlichen Aufbau wie die des library_compilers und erlaubt die Überwachung und Einflussnahme auf alle Teilschritte der Synthese.

7.3.1 Funktionalität

Die Logiksynthese erfolgt linear in folgenden Schritten:

- Die gegebenen VHDL-Designbeschreibungen werden syntaktisch überprüft. Für jede in einer Datei gefundenen VHDL-Entity wird in der programminternen Datenbank ein neuer Eintrag als neue bekannte Objektklasse generiert und die Quelldatei der zugehörigen Verwaltungsstruktur bekannt gegeben.

- Nach der Angabe der Top-Level-Entity durch den Benutzer wird die sog. Kopfsynthese im Präprozessor (siehe „Präprozessor“ auf Seite 34) durchgeführt. Dabei wird festgestellt, ob sich alle Abhängigkeiten mit den gegebenen Entities erfüllen lassen. Insbesondere werden je nach Bedarf generische Teile der Entities konkretisiert und die prinzipielle Übersetzbarkeit unter diesen Randbedingungen wird geprüft.

- Ausgehend von der gegebenen Top-Level-Entity wird rekursiv die eigentliche Logiksynthese (siehe „Prozesssynthese“ auf Seite 35 bzw. „Synthese paralleler VHDL-Beschreibungen“ auf Seite 54) durchgeführt. Für jede in einer Entity benötigten Instanz einer anderen Entity wird eine eigene Netzliste, ausgehend von der zugehörigen Quell-

datei der Klasse, erzeugt. Es entsteht ein generischer Netzlistenbaum, wobei jede Netzlisteninstanz einer Entity nur einmal verwendet wird. Dieses Vorgehen ist in dieser Form Stand der Technik in den vergleichbaren Werkzeugen und vereinfacht die im nächsten Schritt durchgeführte Technologieabbildung, da auf der implizit gegebenen flachen Struktur die Technologie direkt abgebildet werden kann.

- Als Letztes wird eine elementare Technologieabbildung der generischen Netzliste durchgeführt. Die Zieltechnologie ist dabei die Menge der graphisch darstellbaren Gatter der Schaltungsvisualisierung. Die Abbildung erfolgt dabei 1:1 zu der internen Repräsentation, d. h. ohne zwischengeschaltete Optimierungsschritte.

Nach diesen Schritten wird das Ergebnis als Netzliste graphisch dargestellt. Das Ergebnis kann vom Benutzer vor der Technologieabbildung geprüft werden. Änderungen an den Quelldateien des Designs können vorgenommen werden, ohne dass unveränderte Teile erneut mit übersetzt werden. Nach erfolgter Logiksynthese kann die Technologieabbildung ALTeM (siehe „Technologieabbildung ALTeM“ auf Seite 59) durchgeführt werden, die linear in folgenden Schritten erfolgt:

- Die vom library_compiler erzeugte Zelldatenbank wird geladen, geprüft und in die A-RTL-Datenbank eingebunden.

- Anhand der vorgegebenen Optimierungskriterien und der verfügbaren Zellen der Zieltechnologie werden die arithmetischen Strukturen abgebildet.

- Es werden die Speicherelemente (Flipflops, Latches) ermittelt, ggf. zu Registern zusammengefasst (siehe „Abschließende Überprüfungen und Optimierungen“ auf Seite 46) und abgebildet.

- Die zwischen den Registern liegenden Schaltwerke werden auf Rückkopplungen geprüft und diese ggf. aufgespalten.

- Die resultierenden Schaltnetze werden unter Berücksichtigung der Optimierungskriterien in die Zieltechnologie übertragen.

- Es wird eine Statistik mit den Eigenschaften der erzeugten Schaltung dargestellt, welche als flache, technologieabhängige Netzliste zur weiteren Verarbeitung gespeichert werden kann.

Die erzeugte VHDL-Netzliste kann unverändert zur weiteren Verarbeitung an einen beliebigen Platzierer und Verdrahter weitergegeben werden.

7.3.2 Bedienung

Der Aufbau der Oberfläche ist sehr ähnlich zu der des library_compilers. Nach dem Start des Programms wird der in Abbildung 7.6 dargestellte Dialog geöffnet. Er erlaubt das Erstellen eines neuen Projektes bzw. das Öffnen eines bereits vorhandenen.

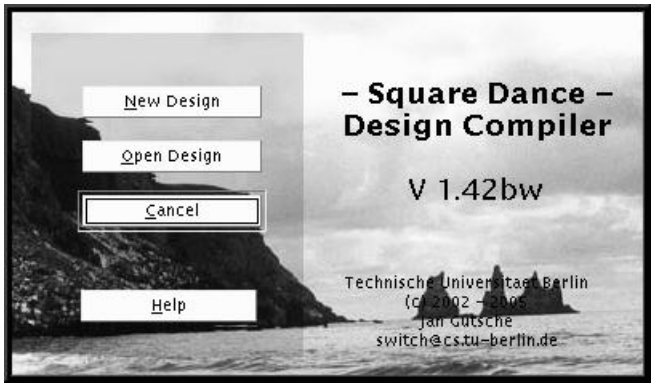


Abbildung 7.6 *Splash-Screen des design_compilers.*

Analog zum library_compiler werden beim Erstellen eines neuen Projektes in einem weiteren Dialog (siehe Abbildung 7.7) alle benötigten Informationen abgefragt. Dem Projekt kann hier ein eigener Name und optional eine freie Beschreibung zugeordnet werden. Notwendig ist die Angabe der Zieltechnologie, die im sdlc-Format, welches vom library_compiler generiert wird, vorliegen muss. Als Quelldateien können im Anschluss beliebig viele VHDL-Beschreibungen angegeben werden. Ein Hinzufügen, Löschen oder Modifizieren der Quelldateien ist auch nach Beendigung des Dialogs jederzeit möglich.

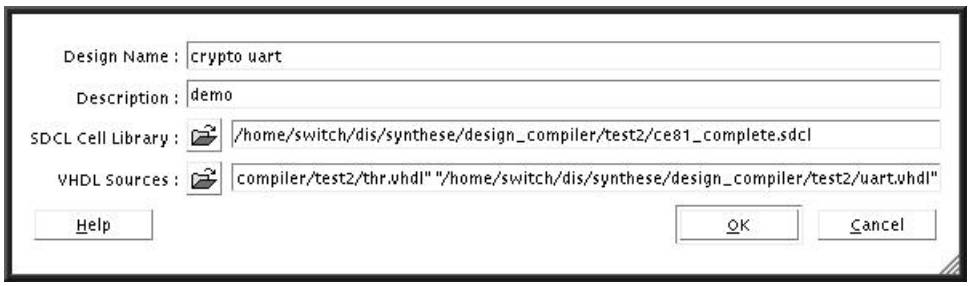


Abbildung 7.7 *Fenster zum Erzeugen eines neuen Designs innerhalb des design_compilers.*

Nach der Bestätigung der Angaben werden die Designbeschreibungen geladen und die Ergebnisse der Analyse im Hauptfenster, wie in Abbildung 7.8 exemplarisch dargestellt, dem Benutzer angezeigt. Es unterteilt sich in vier Bereiche:

- Über die Katalogreiter im ersten Bereich ist der Wechsel der durch das Werkzeug behandelten Abstraktionsebenen möglich. Unter dem Punkt „Files“, der auch in der Abbildung 7.8 aktiv ist, ist der Zugriff auf die Ausgangsbeschreibungen möglich. Der Punkt „Entity Hierarchy“ zeigt das Ergebnis der Logiksynthese und der Punkt „Statistics“ die Ergebnisse der Technologieabbildung sowie weitere Informationen.

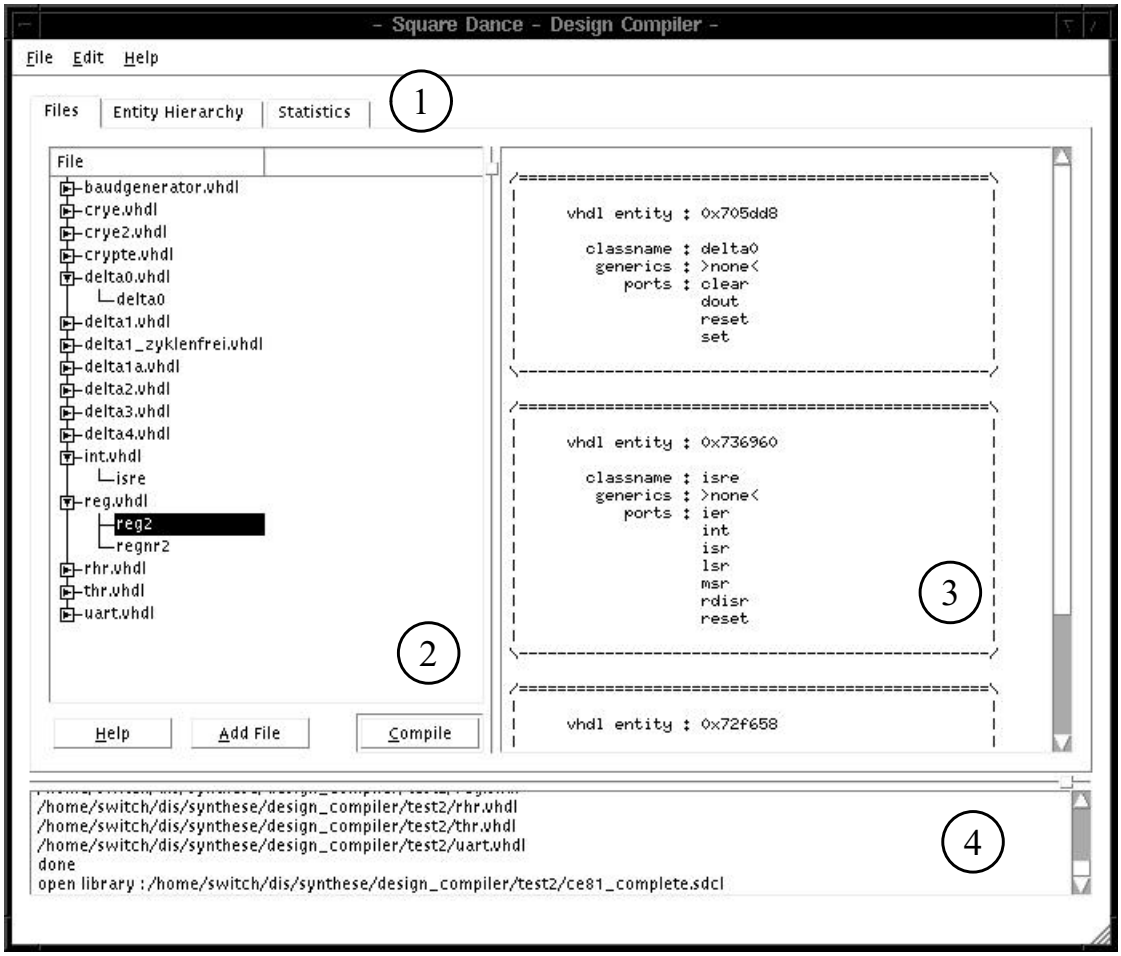


Abbildung 7.8 *Darstellung des Dateibrowsers im Hauptfenster des design _compilers*.

- Im zweiten Bereich ist der Filebrowser abgebildet. Hier werden alle zum Design zugehörigen Quellbeschreibungen aufgeführt. Zusätzlich werden alle in einer Datei definierten Entities angezeigt. Für die Logiksynthese ist die Auswahl einer Entity durch den Benutzer für die Definition der Top-Level-Entity notwendig. Die Logiksynthese kann dann über den Punkt „Compile“ im unteren Teil des zweiten Bereichs durchgeführt werden.

- Der dritte Bereich wird als Entityview bezeichnet. An dieser Stelle werden zusätzliche Informationen über die im ersten Bereich anwählbaren Entities ausgegeben. Neben allgemeinen Debugging Informationen gehören dazu die Ports und generischen Variablen. Zusätzliche Informationen wie beispielsweise Typ und Breite von Ports werden nicht angegeben, da diese von den generischen Variablen abhängig sein können, die zu diesem Zeitpunkt noch undefiniert sind.

- Im vierten Bereich, dem Message-Bereich werden analog zum Message-Bereich des library_compilers Protokollinformationen sowie Fehlermeldungen ausgegeben. Dieser Bereich steht in allen Sichtweisen auf das Projekt zur Verfügung.

Nach dem Start der Logiksynthese über den Punkt „Compile“ wird, ausgehend von der angewählten Top-Level-Entity, das Design rekursiv übersetzt. Bei Fehlern im Design wird eine entsprechende Fehlermeldung mit Dateinamen und Zeilennummer ausgegeben und der Übersetzungsvorgang unterbrochen. Ein erneutes Einlesen der Designbeschreibung nach einer Korrektur ist nicht notwendig, da Änderungen an den vorliegenden Dateien automatisch erkannt werden. Nach erfolgreicher Logiksynthese wird automatisch die angezeigte Sichtweise auf das Projekt (wie in Abbildung 7.9 dargestellt) geändert:

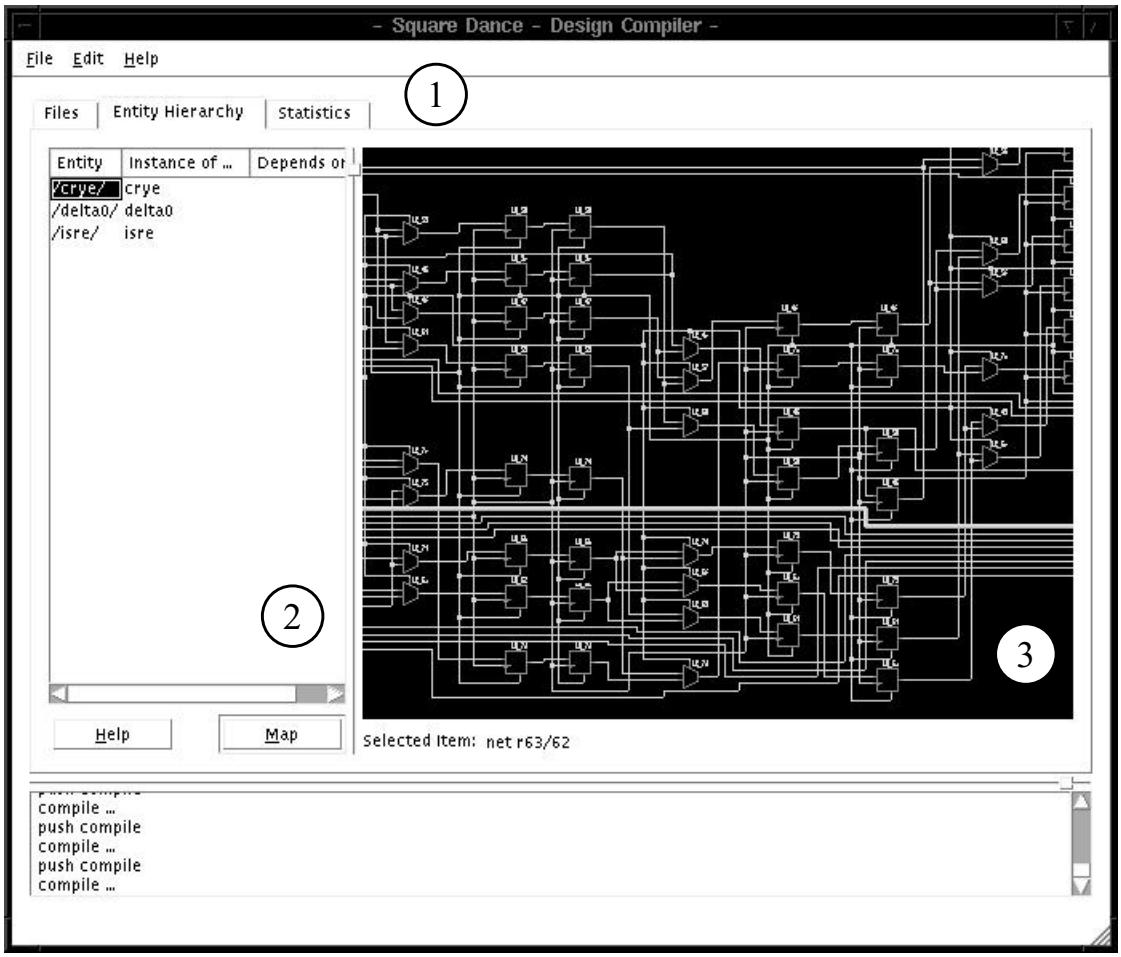


Abbildung 7.9 *Darstellung des Designbrowsers im Hauptfenster des design_compilers*.

- Über die Katalogreiter ist auch nach der Logiksynthese weiterhin ein Zugriff auf die vorhergehende Sichtweise möglich. Bei einem ungenügenden Syntheseergebnis kann so die entsprechende Komponente ausgetauscht und erneut synthetisiert werden, ohne die komplette Synthese für die erfolgreich abgebildeten Komponenten erneut durchführen zu müssen.

- Im zweiten Bereich wird der Instancebrowser dargestellt. Es werden in diesem alle synthetisierten Schaltungsinstanzen mit zugehöriger Schaltungsklasse und bestehenden Abhängigkeiten von anderen Schaltungsinstanzen tabellarisch aufgelistet. Bei der Auswahl einer Instanz wird die entsprechende Schaltung im dritten Bereich, dem Schematicview, angezeigt.

- Der dritte Bereich wird als Schematicview bezeichnet. In diesem wird die jeweils zugehörige Schaltung der im Instancebrowser ausgewählten Schaltungsinstanz angezeigt. Die Schaltung kann gezoomt und der dargestellte Ausschnitt beliebig festgelegt werden. Bei der Auswahl einer Leitung wird das zugehörige Leitungsnetz hervorgehoben und der Netzname unter der Schaltung angegeben.

Beim Wechsel der dargestellten Schaltung über den Instancebrowser bleiben die Darstellungseinstellungen der zuvor dargestellten Schaltung erhalten.

Zur weiteren Verbesserung der Übersichtlichkeit werden Schaltungen, die eine bestimmte Anzahl von Bauelementen überschreiten, in mehrere separate Schaltpläne aufgespalten.

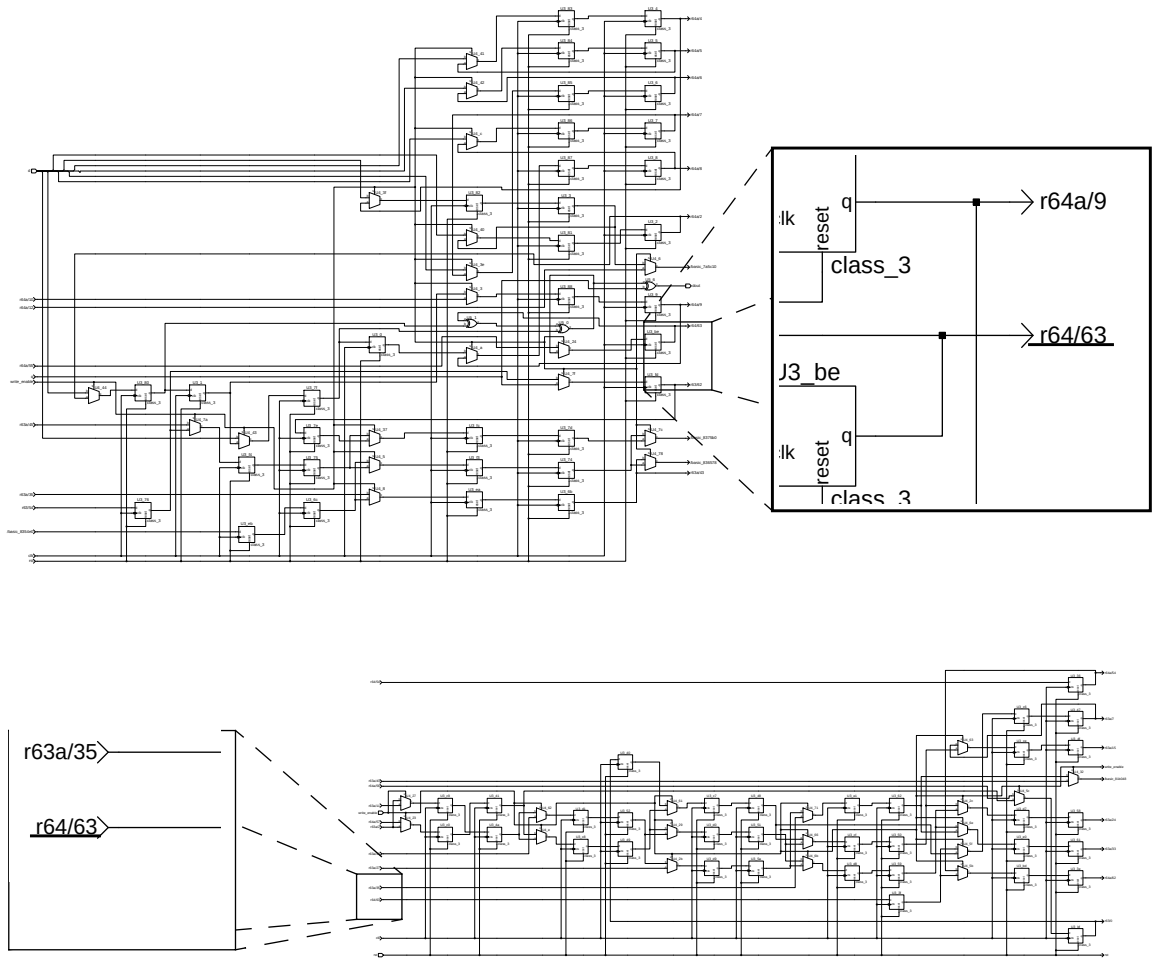


Abbildung 7.10 *Schaltungsausschnitte eines komplexen Schaltplans. Die Ausschnitte sind über sog. Page-Connectoren miteinander verbunden.*

Der Signalfluss innerhalb einer Unterschaltung wird soweit möglich von links nach rechts dargestellt. Als Beispiel sind in Abbildung 7.10 Ausschnitte eines komplexen Schaltwerks abgebildet. Sie sind untereinander über sog. Page-Connectoren verbunden. Die Anwahl eines Page-Connectors stellt das jeweils dort angeschlossene Netz dar. Die strenge Beibehaltung des Signalflusses von links nach rechts führt, wie in Abbildung 7.11 dargestellt, zu einer Ringstruktur, d. h. die ausgangsseitigen Page-Connectoren des letzten Schaltungsaus-

schnittes sind mit den eingangsseitigen Page-Connectoren des ersten Schaltungsausschnittes verbunden. Je nach Aufbau des Netzes variiert die Anzahl der Segmente des Schaltungsrings, wobei ein Segment bei Bedarf mehrere Schaltungsausschnitte aufnehmen kann, welche dann parallel vom gleichen Segment ihre Eingangssignale beziehen.

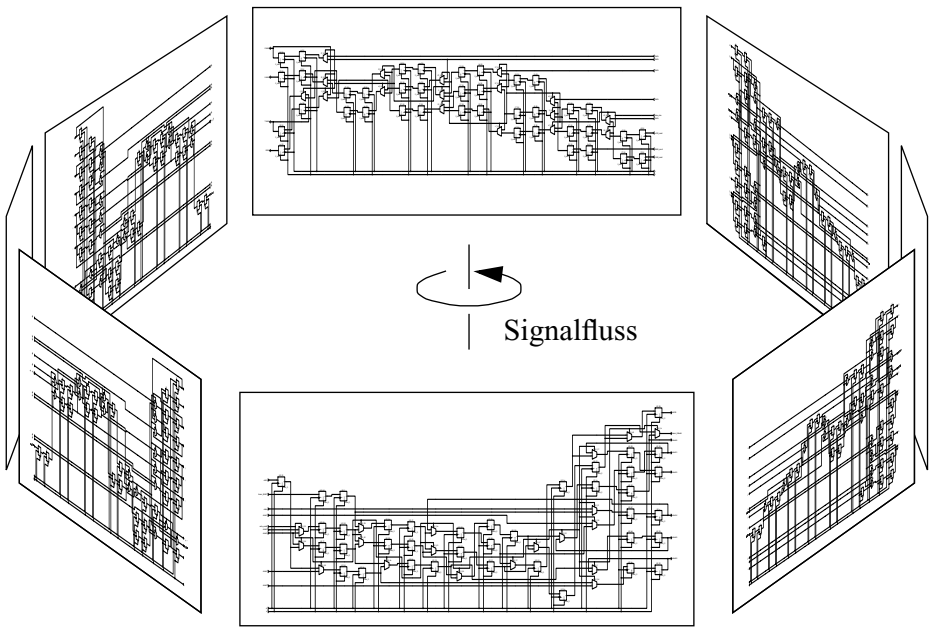


Abbildung 7.11 *Ringstruktur bei der Aufteilung einer umfangreichen Schaltung in mehrere Schaltungsseiten. Der Signalfluss ist entgegengesetzt dem Uhrzeigersinn gerichtet.*

Nach diesem Schritt kann die Technologieabbildung vorgenommen werden. Den einzelnen Schaltungsinstanzen können für diesen Schritt zusätzliche Randbedingungen zugeordnet werden. Der entsprechende in Abbildung 7.12 dargestellte Dialog wird nach der Auswahl der Instanz durch Anwahl der Funktion „Map“ (siehe auch Abbildung 7.9, zweiter Bereich unten) aufgerufen. Er ermöglicht die Angabe wahlweise absoluter oder relativer Optimierungskriterien für Geschwindigkeit, Platzbedarf und elektrische Leistung. Absolute Werte können direkt eingetragen werden, relative jeweils über Schieberegler eingestellt werden. Wird mehr als ein Optimierungskriterium relativ angegeben, wird das Verhältnis der Position der Schieberegler, wie dann auch im Bereich „Summary“ angegeben, übernommen. Befindet sich ein Schieberegler am linken Anschlag, bleibt das entsprechende Optimierungskriterium in den weiteren Schritten unberücksichtigt. Nach Einstellung der gewünschten Randbedingungen können diese über „Apply“ an die aktuelle Schaltungsinstanz gebunden werden. Diese gelten dann auch entsprechend für alle in der aktuellen Schaltungsinstanz verwendeten weiteren Instanzen, so dass für diese individuelle Optimierungskriterien nicht explizit angegeben werden müssen. So ist es möglich, ausgewählten Komponenten der Gesamtschaltung speziell Ressourcen zuzuordnen. Beispielsweise können auf diese Weise Schaltungen für mobile Anwendungen insgesamt auf geringe Verlustleistung hin entwickelt werden, wobei zeitkritische Komponenten des Systems ungeachtet des höheren Leistungsbedarfs selektiv auf maximale Verarbeitungsgeschwindigkeit optimiert werden.

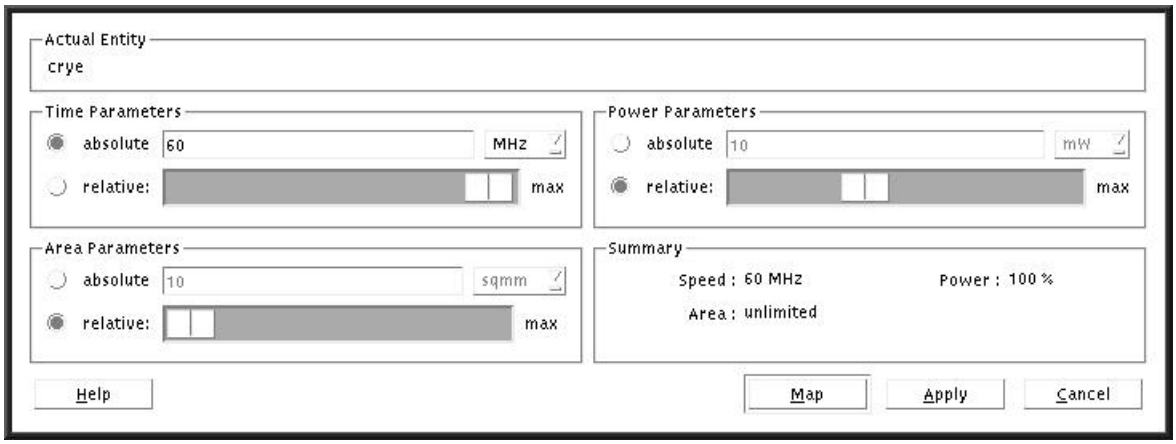


Abbildung 7.12 *Dialog zum Einstellen der zur Technologieabbildung der aktuellen Schaltungsinstanz einzuhaltenden Randbedingungen.*

Nach dem Einstellen der Gütekriterien für die oberste Schaltungsinstanz kann die Technologieabbildung über den Punkt „Map“ im Dialogfenster gestartet werden. Nach dieser werden die Ergebnisse im Hauptfenster des design_compilers wie in Abbildung 7.13 dargestellt angezeigt. Neben den durch die Optimierung erreichten Randparametern werden auch die für das Hauptmodul geforderten Randparameter vergleichend aufgeführt. Verletzungen der geforderten Parameter durch die Technologieabbildung werden farblich gekennzeichnet. Zusätzlich zu den Ergebnissen der Synthese wird der Ressourcenverbrauch des Synthesewerkzeugs selbst auch mit aufgeführt. Das Syntheseergebnis kann nach der Technologieabbildung über „Save Result“ als technologieabhängige Netzliste gespeichert werden.

7.4 Ergebnisse

Ziel der Arbeit war die Entlastung des Entwicklers durch eine weitere Automatisierung des Designprozesses mit Hilfe einer robusteren und erweiterten Logiksynthese und einer mächtigeren Technologieabbildung. Die zu diesem Zweck entwickelten Verfahren sind weitgehend unabhängig und wurden deshalb getrennt betrachtet.

7.4.1 Logiksynthese SibaS

Mit Ausnahme der Prozesssynthese und Arithmetiksynthese sind die verwendeten Verfahren in der Logiksynthese SibaS weitgehend denen gleich, die in konventionellen Synthesewerkzeugen zur Anwendung kommen.

Die SibaS-Prozesssynthese liefert in der Regel ebenfalls die gleichen Ergebnisse wie die Prozesssynthese vergleichbarer Logiksynthesewerkzeuge. Der neue Ansatz, anstelle einer Strukturanalyse von Prozessen eine Simulation durchzuführen, führt jedoch zu einer wesentlich robusteren Logiksynthese, die zum einen weniger Prozessbeschreibungen

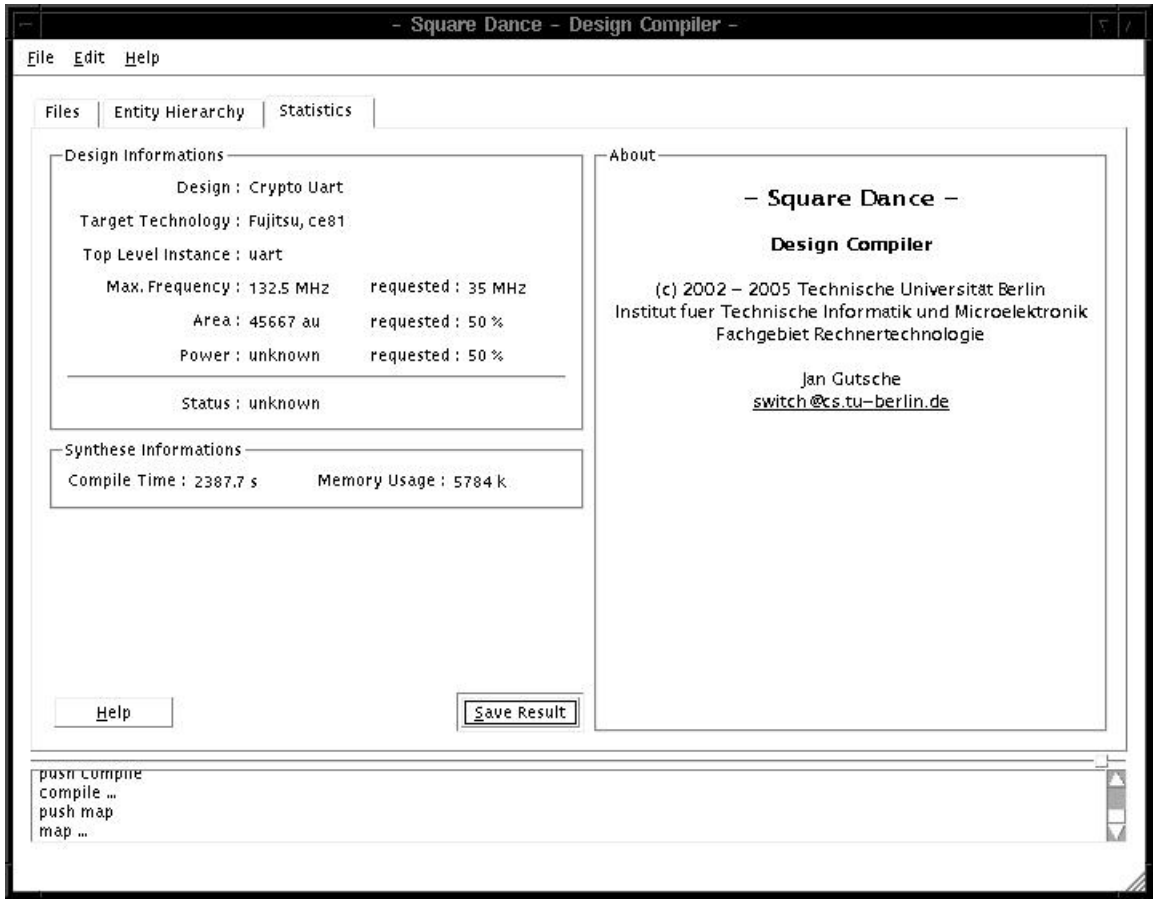


Abbildung 7.13 *Zusammenfassung der Ergebnisse der Technologieabbildung im Hauptfenster des design_compilers.*

ablehnt und zum anderen auch angenommene Prozessbeschreibungen fehlerfrei in eine entsprechende Schaltung überträgt. Durch dieses Verhalten wird die durch den Entwickler zu leistende Arbeit bei der Fehlersuche bzw. das entsprechende Umformulieren der Ausgangsbeschreibung reduziert.

Bei der Arithmetikabbildung wurde ebenfalls ein abweichendes Verfahren implementiert. In den verfügbaren Werkzeugen wird an sich keine gesonderte Arithmetikabbildung durchgeführt, da entsprechende Abbildungsvorschriften in den zu verwendenden VHDL-Arithmetik-Packages vorhanden sind. So wird das Problem auf das Abbilden von Logikstrukturen transformiert. Bedingt durch zumeist jeweils nur einer vorgegebenen Struktur innerhalb der VHDL-Arithmetik-Packages ist auch nur jeweils eine Schaltungsvariante durch die automatische Logiksynthese erzeugbar. Dies kann je nach Aufgabe ausreichend sein. In der Regel werden jedoch die entsprechenden Arithmetikstrukturen durch Spezialschaltungen ersetzt, die mit Hilfe zusätzlicher Werkzeuge durch den Entwickler generiert werden müssen. Innerhalb der Logiksynthese SibaS wird keine Arithmetikabbildung durchgeführt, sondern die abzubildende Funktion wird an die Technologieabbildung weitergereicht. Durch diesen Schritt geht die Austauschbarkeit der Technologieabbildung bzw. Logiksynthese teilweise verloren. Die Vorteile der verschobenen Arithmetiksynthese können nur beim kombinierten Einsatz beider Werkzeuge genutzt werden, auch wenn eine

prinzipielle Austauschbarkeit weiterhin gegeben ist. Als zusätzliche Modifikation im Vergleich zu konventionellen Synthesewerkzeugen sind zahlreiche Optimierungsschritte innerhalb der Logiksynthese implementiert, die normalerweise erst als Teil der Technologieabbildung durchgeführt werden. Hierzu gehört u. a. das Auflösen von Konstanten, die sowohl explizit (VHDL-Konstanten) als auch implizit (durch Logik erzeugte konstante Signale) gegeben sein können, oder die Generierung konkreter Speicherelemente. Aus Sicht des Entwurfsflusses hat die Verschiebung der Optimierungsschritte aus der Technologieabbildung in die Logiksynthese keinen Einfluss auf die Qualität der schlussendlich erzeugten technologieabhängigen Netzliste. Ein erheblicher Gewinn ergibt sich jedoch wieder aus Sicht des Entwicklers, da die erzeugten optimierten generischen Netzlisten wesentlich leichter zu lesen sind, was eine ggf. durchzuführende Fehlersuche erheblich vereinfacht. Fehler der Optimierung, die im konventionellen Synthesefluss erst in der technologiespezifischen, vergleichsweise schwer zu lesenden Netzliste auftauchen, sind in der generischen Netzliste leichter nachzuvollziehen.

Aufgrund des verschiedenen Aufbaus und der Funktionalität der Logiksynthese SibaS ist ein direkter Schaltungsvergleich von Logiksyntheseergebnissen anderer Logiksynthesewerkzeuge schwierig. Durch die Verschiebung verschiedener Optimierungsschritte von der Technologieabbildung in die Logiksynthese wird bei der Logiksynthese SibaS grundsätzlich immer ein in jeder Hinsicht vorteilhafteres Ergebnis geliefert. Die Verschiebung der Arithmetiksynthese in die Technologieabbildung verstärkt diesen Effekt zusätzlich, da Gatter für die Realisierung von Arithmetik bei der Logiksynthese SibaS nicht vorgesehen sind. Ein direkter Vergleich von Logiksyntheseergebnissen hinsichtlich des Ressourcenbedarfs usw. ist dementsprechend nicht sinnvoll.

In Hinblick auf die Robustheit des Verfahrens wurden im Rahmen der Arbeit bereits unterschiedliche Beispiele aufgeführt, die mit konventionellen Werkzeugen nur sehr fehlerhaft zu übersetzen sind, mit der Logiksynthese SibaS jedoch ein fehlerfreies Ergebnis liefern. (Siehe beispielsweise Listing 2.6 und Abbildung 2.1 auf Seite 13 bzw. Listing 4.9 auf Seite 40 und die dazugehörigen Logiksyntheseergebnisse in Abbildung 4.4 und Abbildung 4.3 auf Seite 45). Als Beispiel für eine mit konventionellen Werkzeugen nicht zu übersetzende Schaltung ist im Anhang B.3 auf Seite 131 eine typische Lösung einer im Rahmen der Veranstaltung „Komponenten digitaler Systeme“ an der TU Berlin von Studenten entwickelten VHDL-Beschreibung abgebildet. Die Beschreibung ist, wie in Abbildung 8.11 auf Seite 132 dargestellt, mit Square-Dance problemlos zu übersetzen.

Als Beispiel für die durch den Benutzer einfachere Nachvollziehbarkeit der durch die Logiksynthese SibaS generierten Schaltung ist im Anhang B.1 auf Seite 119 die Beschreibung eines Interruptcontrollers mit den generierten Schaltungen der Logiksynthese SibaS und des Synopsys design_analyzer als Vergleich gegeben. Beide Schaltungen sind funktional identisch. Durch die Verschiebung von Optimierungsschritten von der Technologieabbildung in die Logiksynthese kommt die Lösung der Logiksynthese SibaS jedoch u. a. mit deutlich weniger Bauelementen aus. Zusätzlich sind in diesem Stadium der Synthese ggf. vorhandene Optimierungsfehler in der Schaltung leichter zu identifizieren.

Im Anhang B.2 auf Seite 124 ist die Beschreibung einer vergleichsweise komplexen Verschlüsselungseinheit mit zugehörigem SibaS-Logiksyntheseergebnis gegeben. Trotz vergleichsweise hoher Komplexität der Schaltung ist durch die Optimierung eine Nachvollziehbarkeit der Schaltung noch gegeben, da funktionslose Schaltungsteile eliminiert und einfachere Symbole insbesondere für Speicherelemente verwendet wurden. Das Syntheseergebnis des Synopsys-design-analyzers ist trotz funktionaler Korrektheit praktisch nicht nachzuvollziehen. Auf eine entsprechende Abbildung wurde verzichtet.

7.4.2 Technologieabbildung ALTeM

Im gegenwärtigen Zustand ist die Technologieabbildung nur bedingt einsatzfähig. Insbesondere die Arithmetikabbildung ist aufgrund des sehr hohen Implementierungsaufwandes nur teilweise implementiert und getestet. In Tabelle 7.1 sind Platzbedarf und Zeit des längsten Pfades für unterschiedliche Schaltungen nach einer Synthese mit Square-Dance und Leonardo Spectrum 2004 aufgeführt. Alle Schaltungen wurden hinsichtlich maximaler Geschwindigkeit optimiert. Unabhängig von der unvollständigen Arithmetikabbildung ist ein schwerwiegendes Problem bei der Technologieabbildung ALTeM, wie auch bei der Interpretation der Ergebnisse, in der Qualität der Technologiedaten zu sehen. Ausgehend von den verwendeten reinen Simulationsdaten sind die dort gegebenen Verzögerungszeiten statisch, d. h. unabhängig von der in der Schaltung zu betrachtenden Last. Da es sich bei den Simulationszeiten in der Regel um den worst-case-Fall handelt, liegen die Zeiten des kritischen Pfades grundsätzlich höher als bei der zum Vergleich herangezogenen Technologieabbildung. Elektrische Randbedingungen konnten nicht geprüft werden, auch wenn die von ALTeM generierten Schaltungen funktional korrekt sind. Eine Auswirkung dieser fehlenden Prüfung ist bei der Technologieabbildung ALTeM die grundsätzliche Wahl des Gatters mit der schwächsten Treiberleistung (in der Regel stehen für jede Funktion drei oder mehr Gatter unterschiedlicher Treiberleistung zur Verfügung). Dieses wirkt sich auf den Platzbedarf und die Verzögerungszeiten aus. So ist der berechnete Platzbedarf bei den durch Square-Dance generierten Schaltungen günstiger. Die reduzierte Verzögerungszeit bei Gattern mit wenig Treiberleistung kompensiert zum Teil die durch die Simulationsbeschreibung gegebenen zu hoch liegenden Werte.

Tabelle 7.1: *Ergebnisse der Technologieabbildung von Leonardo Spectrum 2004 und Square-Dance-ALTeM für die SCL05u Testtechnologie*

Schaltung	Square-Dance-ALTeM		Leonardo Spectrum 2004	
	Area	kritischer Pfad	Area	kritischer Pfad
baudgen	6734	7,23	6913	7,65
delta4	1098	2,01	1119	1,90
isre	1020	2,58	1026	2,50
rhre	11077	5,57	10485	5,10
thr	8735	5,25	8268	4,86
uart	106100	7,24	106307	7,66
crypte	67387	3,27	67411	3,24

Eine direkte Auswirkung der Arithmetikabbildung ist nur in der Schaltung `baudgen` zu sehen. Innerhalb dieser Schaltung sind mehrere Addiererschaltungen instantiiert, welches sich auf den reduzierten Platzbedarf im Vergleich zum Abbildungsergebnis von Leonardo Spectrum 2004 auswirkt. Der längste Pfad in dieser Schaltung wird dann durch einen Vergleicher bestimmt, welcher aufgrund noch fehlender Implementierung als Carry-Ripple instantiiert wird.

Ein weiteres Problem ist der vergleichsweise hohe Rechenbedarf der Technologieabbildung ALTeM. Dieser liegt je nach Schaltung um den Faktor 100 bis 200 höher als bei vergleichbaren Werkzeugen. Als hauptsächlicher Schwachpunkt ist hier die verwendete Gütefunktion zu sehen, welche zum einen sehr häufig berechnet zum anderen sehr aufwändige Rechenschritte erfordert.

8 Ausblick

8.1 Logiksynthese

Potential für eine Weiterentwicklung der Verfahren bzw. auch des Werkzeugs selbst ist in unterschiedlichen Richtungen gegeben. Im gegenwärtigen Entwicklungsstadium können bei der Schaltungsvisualisierung Signale nur an den Schnittstellen zu anderen Schaltungsteilen zu Bussen zusammengefasst werden. Das Zusammenfassen von Signalen auch innerhalb einer Schaltung, bzw. auch das Zusammenfassen von z. B. einzelnen Flipflops zu Registern, kann die zu visualisierende Schaltungsstruktur weiter vereinfachen und die Übersichtlichkeit weiter erhöhen.

Da ein kompletter Austausch der Logiksyntheseverfahren in bestehenden Synthesewerkzeugen mit hohem Aufwand verbunden ist, ist als Zwischenlösung eine Implementierung der Verfahren im Rahmen eines Linters (Lint: Prozessor zur lexikalischen, syntaktischen und semantischen Analyse von Programmen [68]) vorstellbar. Dieser könnte entwurfsflussunabhängig parallel zum eigentlichen Synthesewerkzeug eine eigene Synthese durchführen, und die Syntheseergebnisse vergleichen. Diskrepanzen im Syntheseergebnis könnten dann relativ schnell und früh auf mögliche Synthesefehler hinweisen.

Neben den in der aktuellen Implementierung durch die neue Prozesssynthese behobenen Ursachen für Synthesefehler gibt es noch weitere Quellen für fehlerhafte Syntheseergebnisse. Als Beispiel ist in Listing 8.1 eine Beschreibung für eine Datenverzögerung um zwei Takte gegeben. In Abbildung 8.1 ist das entsprechende Square-Dance-Syntheseergebnis abgebildet, welches funktional mit den Syntheseergebnissen vergleichbarer Werkzeuge übereinstimmt. Innerhalb einer VHDL-Simulation der in Listing 8.1 beschriebenen Schaltung wird das Taktsignal, bedingt durch die Zuweisung in Zeile 8, um einen Deltawert verzögert, bevor es den zweiten Prozess auslöst. Innerhalb dieser Verzögerung wird die im ersten Prozess `ff1` beschriebene Zuweisung ausgeführt, d. h., das Datenwort `d` wird im ersten Flipflop durchgeschaltet und liegt am Dateneingang des nachgeschalteten Flipflops an. Löst das verzögerte Taktsignal also den zweiten Prozess `ff2` aus, wird das gleiche Datum im entsprechenden Flipflop gespeichert, ebenso wie im vorgeschalteten Flipflop. Innerhalb der Simulation wird das Datum also nur um einen Takt verzögert. Die durch die Synthese realisierte Schaltung verzögert das Datum hingegen um zwei Takte. Allein das Identifizieren eines solchen Falles mit einer entsprechenden Rückmeldung an den Benutzer auch ohne konkrete Auflösung würde schon helfen, entsprechende Beschreibungen zu vermeiden.

```
1.  ff1: process(clk1)
2.  begin  -- process ff1
3.    if clk1'event and clk1 = '1' then
4.      s1 <= d;
5.    end if;
6.  end process ff1;
7.
8.  clk2 <= clk1;
9.
10. ff2: process( clk2 )
11. begin -- process ff2
12.   if clk2'event and clk2 = '1' then
13.     out2 <= s1;
14.   end if;
15. end process ff2;
```

Listing 8.1 *VHDL-Beschreibung eines Registers zur Datenverzögerung.*

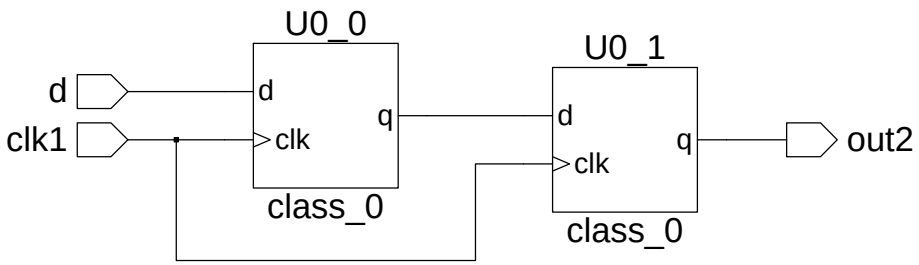


Abbildung 8.1 *Square-Dance-Syntheseergebnis von Listing 8.1.*

8.2 Technologieabbildung

In Hinblick auf einen erfolgreichen Einsatz der Technologieabbildung sind insbesondere zwei Probleme zu lösen:

Da die Qualität der Ergebnisse maßgeblich von der Qualität der Eingangsdaten abhängt, diese mit den verwendeten Simulationsdaten jedoch nicht gegeben ist, muss der Eingangsfilter der Bibliothekssynthese auf gängige Technologiebeschreibungsformate erweitert werden. Daran geknüpft ist die A-RTL Datenbank entsprechend zu erweitern und die Technologieabbildungsverfahren für die korrekte Verarbeitung der komplexeren Zellbeschreibungen anzupassen.

Hinsichtlich der benötigten Rechenzeit ist vor allem die Gütefunktion zu modifizieren. Zum einen ist es denkbar, mehrere Funktionen unterschiedlicher Genauigkeit und dementsprechend unterschiedlichen Zeitbedarfs einzusetzen, die innerhalb der durchzuführenden Itera-

tionen bei der Verfeinerung der Schaltung ausgetauscht werden. Zusätzlich sollten Möglichkeiten gefunden werden, die Funktion mit möglichst wenigen und einfachen Operationen (z. B. ausschließlich Integerarithmetik) abzubilden. Innerhalb der Technologieabbildung könnte eine Gegenüberstellung der in der Vorsynthese berechneten und später tatsächlich benötigten Ressourcen helfen, die Abschätzung bei zukünftigen Syntheseprozessen zu verbessern. Entsprechende Statistiken könnten in der jeweiligen Technologiebibliothek mit abgelegt werden und so über die Zahl der durchgeführten Syntheseprozesse die relative Synthesezeit jedes neuen Projektes reduzieren. Es wäre auch interessant zu untersuchen, inwieweit die Technologie dann überhaupt einen Einfluss auf die Abschätzungen hat. Entsprechende Statistiken könnten sich auch als wertvoll für eine grundsätzliche Klassifikation der verschiedenen Technologien erweisen. Diese dann synthesewerkzeugspezifische Klassifikation könnte helfen, die generelle Tauglichkeit einer bestimmten Technologie, verbunden mit dem speziellen Werkzeug für ein bestimmtes Projekt besser zu beurteilen.

Eine weitere Zeitersparnis würde sich bei der Verschiebung der Arithmetiksynthese in die Bibliothekssynthese ergeben. Der für eine Technologie und Schaltungstechnik charakteristische Ressourcenbedarf wird dann jeweils nur einmal ermittelt und in der Technologiebibliothek mit gespeichert. Die entsprechenden Rechenvorgänge bei der Technologieabbildung selbst würden entfallen. Der nächste Schritt wäre es, auf die Implementierung einer Arithmetiksynthese ganz zu verzichten und sich die in den Coregeneratoren schon vorhanden Implementierungen zu Nutze zu machen. In Hinblick auf einen weitgehend automatisierten Syntheseprozess müssten diese herstellerunabhängig mit einer einheitlichen Programmierschnittstelle ausgerüstet werden.

Eine Weiterentwicklung der in dem Programmsystem selbst verwirklichten Idee wäre beispielsweise die Integration von Place and Route. Die Rückführung der Ergebnisse an die Logiksynthese (Back Annotation) wäre dann ebenfalls automatisch durchführbar, wodurch eine weitere Interaktion mit dem Benutzer entfallen könnte.

Anhang A Dateiformate

Dieser Abschnitt enthält einige typische Beispiele für die Datenrepräsentation von Ein- und Ausgangsdatensätzen, auf die in dieser Arbeit Bezug genommen wird. Aus Anschaulichkeitsgründen wurde auf die Aufführung der entsprechenden Grammatiken verzichtet.

A.1 Synopsys Technologiebeschreibung

Die Synopsys-Technologiebeschreibung ist das Ausgangsformat für den Synopsys library compiler, der diese in ein für die Synthese verwertbares Datenbankformat überträgt. Die Technologiebeschreibung besteht aus einem Dateikopf mit den technologiespezifischen Parametern und der sequentiellen Aufführung aller Zellen. Die Zellbeschreibung selbst enthält alle für die Synthese notwendigen Informationen. Im Beispiel ist ein Abschnitt der Technologiebeschreibung abgebildet, der die Zelle ACCSIHCONX2 des SMC 0.13µm-Prozess von Artisan in stark gekürzter Form beschreibt.

```
1.  cell (ACCSIHCONX2) {
2.    cell_footprint : accsihcon;
3.    area : 10.1844;
4.    pin(A) {
5.      direction : input;
6.      capacitance : 0.007219;
7.    }
8.    pin(B) {
9.      direction : input;
10.     capacitance : 0.008079;
11.   }
12.   pin(CO0N) {
13.     direction : output;
14.     capacitance : 0.0;
15.     function : "(!(A B))";
16.     internal_power() {
17.       related_pin : "A";
18.       rise_power(energy_template_7x7) {
19.         index_1 ("0.02,0.03,0.05, 0.10, 0.19,0.37, 0.7");
20.         index_2("0.00316, 0.008216, 0.01896, 0.040448,
21.                  0.083424, 0.168744, 0.34128");
22.         values( "0.021516, 0.020526, 0.021996, 0.021915,
23.                  0.021457, 0.020210, 0.017362","0.021980,
24.                  0.022606, 0.022401, 0.021928, 0.020725,
25.                  0.017926", "0.023206, 0.022173,0.020956,
26.                  "0.025347, 0.026141, 0.025749,0.024583,
27.                  0.032047, 0.030860, 0.029482, 0.027108,
28.                  0.024078, 0.019885", "0.046434,0.045473,
```

```
29.                  0.043789, 0.040939, 0.037159, 0.032098,
30.                   [...],
31.                   0.025548");
32.       }
33.       fall_power(energy_template_7x7) { [...] }
34.     }
35.     timing() {
36.       related_pin : "A";
37.       timing_sense : negative_unate;
38.       cell_rise(delay_template_7x7) { [...] }
39.       rise_transition(delay_template_7x7) { [...] }
40.       cell_fall(delay_template_7x7) { [...] }
41.       fall_transition(delay_template_7x7) { [...] }
42.     }
43.     internal_power() {
44.       related_pin : "B";
45.       rise_power(energy_template_7x7) { [...] }
46.       fall_power(energy_template_7x7) { [...] }
47.     }
48.     timing() {
49.       related_pin : "B";
50.       timing_sense : negative_unate;
51.       cell_rise(delay_template_7x7) { [...] }
52.       rise_transition(delay_template_7x7) { [...] }
53.       cell_fall(delay_template_7x7) { [...] }
54.       fall_transition(delay_template_7x7) { [...] }
55.     }
56.     max_capacitance : 0.170640;
57.   }
58.   pin(CO1N) {
59.     direction : output;
60.     capacitance : 0.0;
61.     function : "(!(A | B))";
62.     internal_power() {
63.       related_pin : "A";
64.       rise_power(energy_template_7x7) { [...] }
65.       fall_power(energy_template_7x7) { [...] }
66.     }
67.     timing() {
68.       related_pin : "A";
69.       timing_sense : negative_unate;
70.       cell_rise(delay_template_7x7) { [...] }
71.       rise_transition(delay_template_7x7) { [...] }
72.       cell_fall(delay_template_7x7) { [...] }
73.       fall_transition(delay_template_7x7) { [...] }
74.     }
75.     max_capacitance : 0.170640;
76.   }
77.   cell_leakage_power : 1.000000;
78. }
```

A.2 VHDL Zellbeschreibung nach Standard IEEE 1076.6

Zellbeschreibungen nach dem Standard IEEE 1076.6 sind primär für die post-Synthese-Simulation gedacht. Unabhängig vom verwendeten Entwurfswerkzeug und dem zugehörigen Quellformat der Zellbeschreibungen ist die Erzeugung oder Bereitstellung der Zellbeschreibungen in diesem eindeutig spezifizierten Format notwendig. Im Vergleich zur Synopsys Technologiebeschreibung fehlen alle nicht zur Simulation notwendigen Informationen. Als Beispiel ist die komplette Beschreibung der Zelle RT3V des ce81-Prozesses von Fujitsu gegeben.

```
1.  ------------------------------------------------------------
2.  -- Created by the Fujitsu SCM2VITAL (V2.10 1999.06.11)
3.  -- TECHNOLOGY    :    CE81
4.  -- FILENAME      :    RT3V.vhd
5.  -- FILE CONTENTS :    Entity, Structural Architecture
6.  -- DATE CREATED  :    99 6/25 11:54:18
7.  -- LIBRARY       :    VITAL3.0/SDF2.1
8.  -- SCM REVISION  :    1.2
9.  -- TIME SCALE    :    1 ps
10. -- LOGIC SYSTEM  :    IEEE-1164
11. ------------------------------------------------------------
12. library IEEE;
13. use IEEE.STD_LOGIC_1164.all;
14. use IEEE.VITAL_Timing.all;
15. use IEEE.VITAL_Primitives.all;
16. library FJ_ASIC_PKG;
17. use FJ_ASIC_PKG.FJ_ASIC_VTables.all;
18.
19. entity RT3V is
20.
21.   generic(
22.     tipd_A1    : VitalDelayType01  := (0 ps, 0 ps);
23.     tipd_A3    : VitalDelayType01  := (0 ps, 0 ps);
24.     tipd_A2    : VitalDelayType01  := (0 ps, 0 ps);
25.     tpd_A1_X   : VitalDelayType01  := (263 ps , 535 ps);
26.     tpd_A2_X   : VitalDelayType01  := (292 ps , 472 ps);
27.     tpd_A3_X   : VitalDelayType01  := (314 ps , 392 ps) );
28.
29.   port(
30.     A1         : in    STD_ULOGIC;
31.     A3         : in    STD_ULOGIC;
32.     A2         : in    STD_ULOGIC;
33.     X          : out   STD_ULOGIC
34.   );
35.
36.   attribute VITAL_LEVEL0 of RT3V : entity is TRUE;
37. end RT3V;
```

```
38. architecture VITAL of RT3V is
39.   attribute VITAL_LEVEL1 of VITAL : architecture is TRUE;
40.
41.   signal A3_ipd     : STD_ULOGIC := 'U';
42.   signal A2_ipd     : STD_ULOGIC := 'U';
43.   signal A1_ipd     : STD_ULOGIC := 'U';
44.
45.   begin
46.
47.   -- Input Path Delays
48.   WireDelay : block
49.   begin
50.     VitalWireDelay(A1_ipd, A1, tipd_A1);
51.     VitalWireDelay(A3_ipd, A3, tipd_A3);
52.     VitalWireDelay(A2_ipd, A2, tipd_A2);
53.   end block;
54.
55.   -- Behaivioural Section
56.   VitalBehavior : process (A1_ipd, A2_ipd, A3_ipd)
57.
58.     variable X_GlitchData   : VitalGlitchDataType;
59.     variable X_temp   : STD_ULOGIC := 'U';
60.
61.   begin
62.
63.     -- Functionality Section
64.     X_temp := ((A1_ipd) OR (A2_ipd) OR (A3_ipd));
65.
66.     -- Path Delay Section
67.     VitalPathDelay01  (X, X_GlitchData, "X", X_temp,
68.      paths => ( 0 => (A1_ipd'LAST_EVENT, tpd_A1_X, TRUE ),
69.                 1 => (A2_ipd'LAST_EVENT, tpd_A2_X, TRUE ),
70.                  2 => (A3_ipd'LAST_EVENT, tpd_A3_X, TRUE )
71.                    ),
72.      DefaultDelay => VitalZeroDelay01,
73.              Mode => DefGlitchMode,
74.               XOn => DefGlitchXOn,
75.             MsgOn => DefGlitchMsgOn,
76.       MsgSeverity => DefGlitchMsgSeverity
77.     );
78.
79.   end process;
80.
81. end VITAL;
```

A.3 Strukturelles VHDL

Strukturelles VHDL ist ein Netzlistenformat, welches von praktisch allen gängigen Synthesewerkzeugen erzeugt und entsprechend von den meisten Layoutwerkzeugen auch gelesen werden kann. Es handelt sich dann auch um das primäre Ausgangsformat des Square-Dance design_compilers. Das Format ist vergleichsweise einfach, so dass sich entsprechende parser mit wenig Aufwand entwickeln lassen. So sind in diesem Format ausschließlich Komponentendeklarationen, Komponenteninstantiierungen und deren Verknüpfung über einfache Signale (std_logic bzw. std_logic_vector) zulässig.

```
1.  -- created by square dance 0.42b;
2.  -- Wed Nov 24 11:13:22 CET 2004
3.
1.  library IEEE;
2.  use IEEE.std_logic_1164.all;
3.
4.  entity ncl_fulladder_1 is
5.    port( a, b, ci: in  std_logic_vector (1 downto 0);
6.          s, co: out std_logic_vector (1 downto 0));
7.  end ncl_fulladder_1;
8.
9.  architecture SYN_verhalten_1 of ncl_fulladder_1 is
10.   component NI1L
11.     port( A: in  std_logic;
12.           X: out std_logic);
13.   end component;
14.
15.   component RN234L
16.     port( A1, A2, A3, B: in std_logic;
17.           X: out std_logic);
18.   end component;
19.
20.   component RN4L
21.     port( A1, A2, B, C: in std_logic;
22.           X: out std_logic);
23.   end component;
24.
25.   component NI2L
26.     port( A1, A2: in std_logic;
27.           X: out std_logic);
28.   end component;
29.
30.    signal basic_22, basic_23, basic_24, basic_25,
31.           basic_26, basic_27, basic_28, basic_29,
32.           basic_30, basic_31: std_logic;
```

```
33. begin
34.
35.   U0_0: RN234L port map( basic_22, basic_23, basic_24,
36.     basic_25, s(1));
37.   U0_1: RN234L port map( basic_26, basic_27, basic_28,
38.     basic_29, s(0));
39.   U0_2: RN234L port map( ci(1), b(1), a(1), co(0),
40.     basic_25);
41.   U0_3: RN234L port map( b(0), ci(0), a(0), co(1),
42.     basic_29);
43.   U1_0: RN4L port map( basic_26, basic_28, basic_27,
44.     basic_30, co(0));
45.   U1_1: RN4L port map( basic_24, basic_23, basic_22,
46.     basic_31, co(1));
47.   U2_0: NI1L port map( ci(1), basic_23);
48.   U2_1: NI1L port map( a(1), basic_24);
49.   U2_2: NI1L port map( b(1), basic_22);
50.   U2_3: NI1L port map( ci(0), basic_27);
51.   U2_4: NI1L port map( b(0), basic_28);
52.   U2_5: NI1L port map( a(0), basic_26);
53.   U3_0: NI2L port map( a(0), b(0), basic_30);
54.   U3_1: NI2L port map( a(1), ci(1), basic_31);
55.
56. end SYN_verhalten_1;
```

Anhang B Beispielschaltungen

Dieser Abschnitt enthält einige Beispielergebnisse der Logiksynthese. Eine Visualisierung der Schaltungen nach der Technologieabbildung ist nicht vorgesehen, da sinnvolle Symbole für die einzelnen Zellen nicht vorliegen bzw. auch nicht erzeugt werden können.

B.1 Interruptcontroller des Crypto-UART

Der in Listing 8.2 beschriebene Interruptcontroller generiert das Interruptsignal des Crypto-UART und stellt verschiedene Interruptbedingungen in einem Interrupt-Status-Register zur Verfügung. In Abbildung 8.2 ist das Synopsys design-analyzer Logiksyntheseergebnis abgebildet, in Abbildung 8.3 das entsprechende Logiksyntheseergebnis des Square-Dance design_compilers und in Abbildung 8.4 eine von Leonardo Spectrum 2004 generierte technologiespezifische (SCL05u) Realisierung. Alle Schaltungen sind korrekt und funktional gleich.

```
1.  ------------------------------------------------------
2.  -- crypto uart interrupt statusregister
3.  ------------------------------------------------------
4.  -- steigende flanke von rdisr setzt lv3reset. geloescht
5.  -- wird es mit fallender flanke von lsr5; wenn gesetzt
6.  -- gibt es keinen lv3 int. lv3reset wird nur gesetzt,
7.  -- wenn kein hoeherer interrupt anliegt.
8.  ------------------------------------------------------
9.  library ieee;
10. use ieee.std_logic_1164.all;
11.
12. entity isre is
13.
14. port(
15.     msr   : in  std_logic_vector(3 downto 0);
16.     lsr   : in  std_logic_vector(5 downto 0);
17.     rdisr : in  std_logic;
18.     ier   : in  std_logic_vector(3 downto 0);
19.     reset : in  std_logic;
20.     int   : out std_logic;   -- interruptleitung
21.     isr   : out std_logic_vector(2 downto 0)
22.   );
23. end isre;
```

Listing 8.2 *Interruptcontroller des Crypto-UART.*

```
24. architecture verhalten of isre is
25.
26.   signal lv3reset    : std_logic;
27.   signal setlv3reset : std_logic;
28.   signal lsr5        : std_logic;
29.   signal isri        : std_logic_vector( 4 downto 1 );
30.   signal isrc, isro  : std_logic_vector( 2 downto 0 );
31.
32. begin
33.
34.   lsr5 <=  lsr(5); -- sonst spinnt die synthese :)
35.
36.   lv3 : process (setlv3reset, lsr5)
37.   begin  -- process lv3
38.     if setlv3reset = '1' then  -- rising clock edge
39.       lv3reset <= '1';
40.     elsif falling_edge( lsr5 ) then --asynchronous reset
41.       lv3reset <= '0';
42.     end if;
43.   end process lv3;
44.
45.   setlv3reset <= '1' when ( (isri( 3 downto 1 ) = "100")
46.          and ( rdisr = '1' ) ) or reset = '1' else '0';
47.
48.   isri(1)<= '0' when lsr(4 downto 1) = "0000"
49.                 else ier(2);
50.   isri(2)<= '0' when lsr(0) = '0'
51.                 else ier(0);
52.   isri(3)<= '0' when ( lsr(5) = '0' ) or ( lv3reset = '1')
53.                 else ier(1);
54.   isri(4)<= '0' when msr(3 downto 0) = "0000"
55.                 else ier(3);
56.
57.   isrc <= "110" when isri(1) = '1' else
58.           "100" when isri(2) = '1' else
59.           "010" when isri(3) = '1' else
60.           "000" when isri(4) = '1' else
61.           "001";
62.
63.   int <= not isrc(0);                  -- ungelatcht(!);
64.
65.   with rdisr select
66.     isro <= isro when '1',
67.     isrc when others;
68.
69.   isr <= isro;
70.
71. end verhalten;
```

Listing 8.2 *Interruptcontroller des Crypto-UART.*

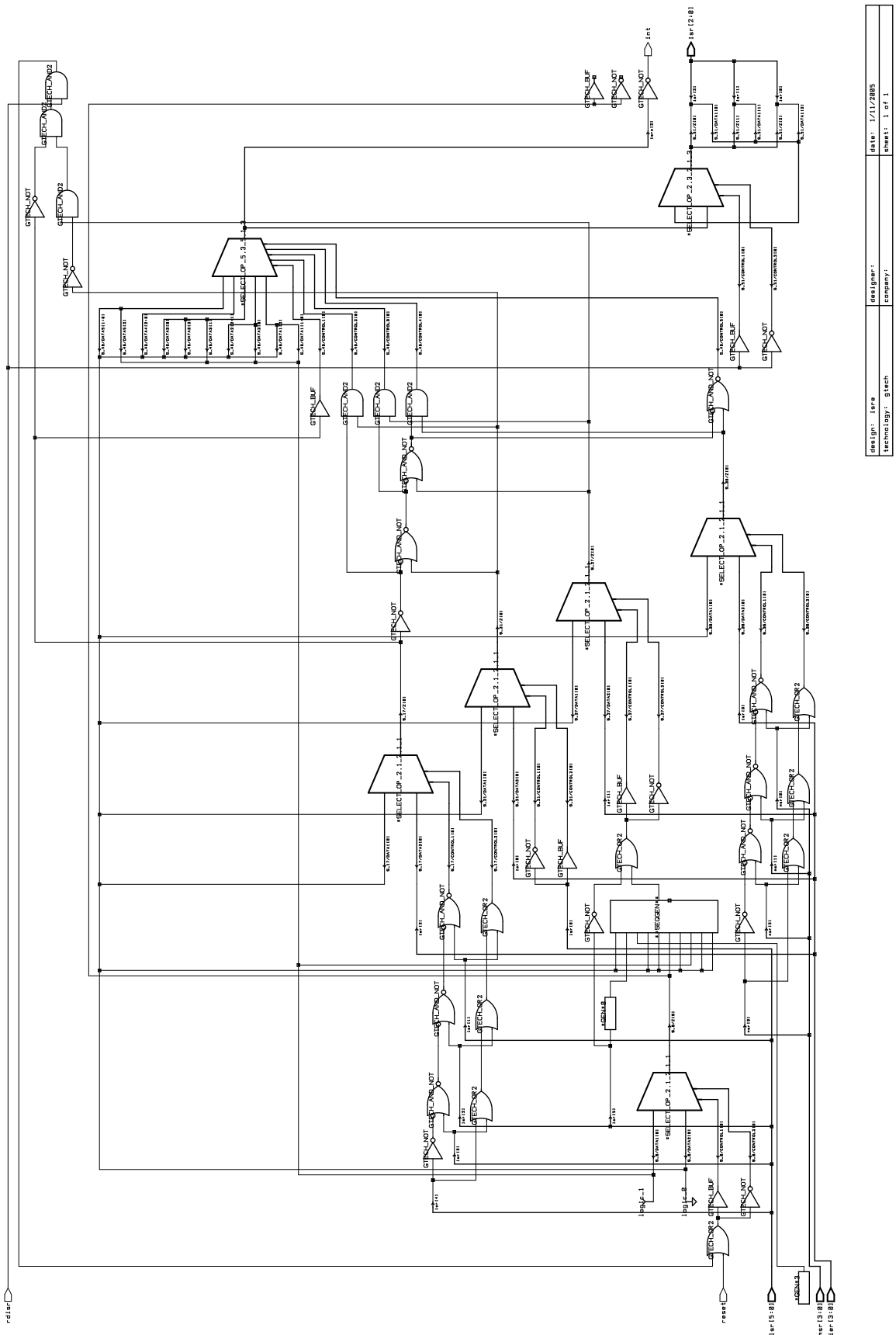


Abbildung 8.2 *Synopsys design_analyzer Logiksyntheseergebnis des Crypto-UART-Interruptcontrollers.*

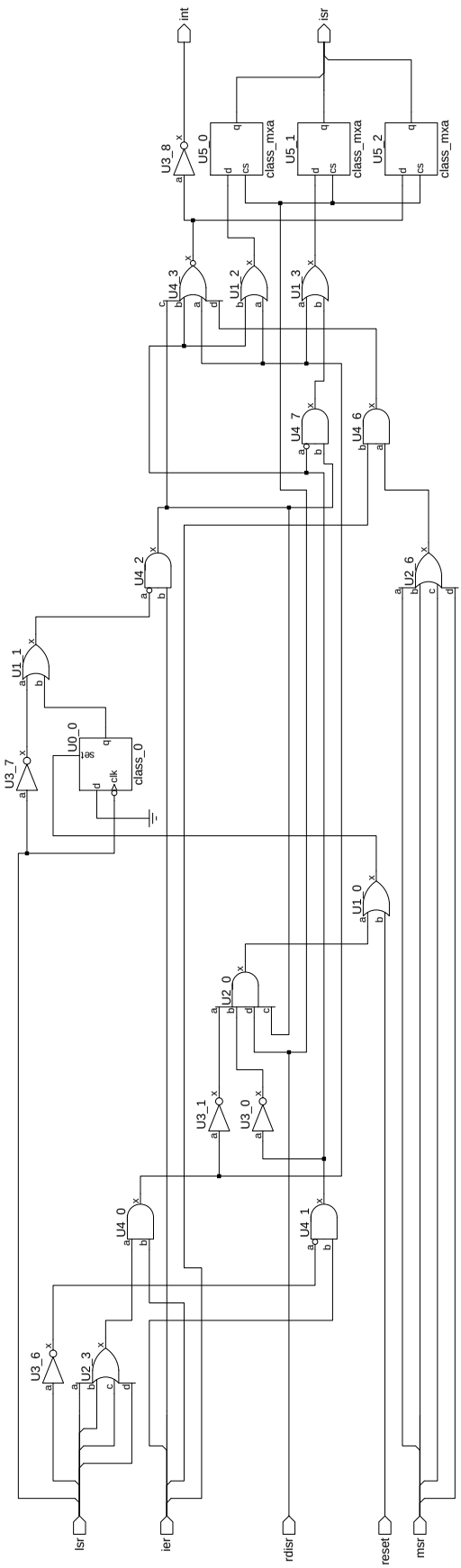


Abbildung 8.3 *Square-Dance design-compiler Logiksyntheseergebnis des Crypto-UART-Interruptcontrollers.*

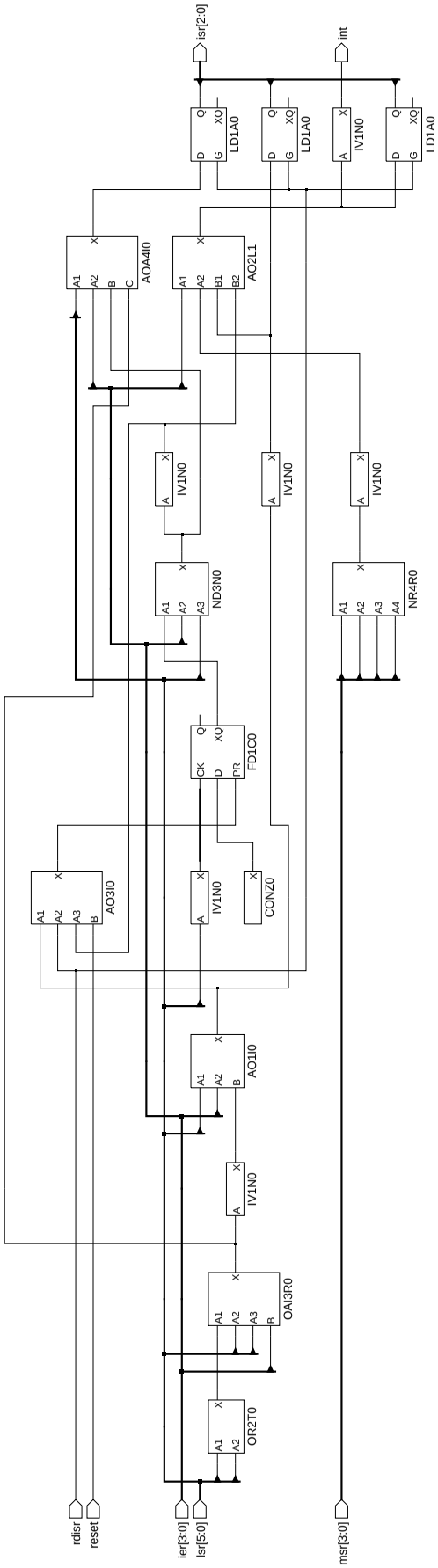


Abbildung 8.4 *Leonardo Spectrum Realisierung des Crypto-UART-Interruptcontrollers in der SCL05u Technologie.*

B.2 Verschlüsselungseinheit des Crypto-UART

Die in Listing 8.3 aufgeführte Verschlüsselungseinheit erzeugt aus zwei linear rückgekoppelten Shiftregistern einen Datenstrom von Pseudozufallszahlen, der zur Ver- bzw. Entschlüsselung von Daten verwendet werden kann. Es ist ein einfaches Interface vorgesehen, um den 127 Bit langen Schlüssel in das Register parallel zu übertragen.

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.  entity crye is
4.    port(rst, weckr, cclk, write_enable:   in  std_logic;
5.            d: in  std_logic_vector(7 downto 0);
6.         dout: out std_logic);
7.    end crye;
8.  architecture verhalten of crye is
9.    signal clk, P, Q : std_logic;
10.   signal R64, R64A: std_logic_vector(63 downto 0);
11.   signal R63, R63A: std_logic_vector(62 downto 0);
12. begin
13.   clk <= weckr  when write_enable = '1' else cclk;
14.   process (rst, clk) begin
15.     if rst='1' then
16.       R64A <= (others => '0');
17.       R63A <= (others => '0');
18.     elsif rising_edge ( clk ) then
19.       R64A <= R64;   -- shift operation
20.       R63A <= R63;   -- shift operation
21.     end if;
22.   end process;
23.   crp : process ( clk )
24.   begin
25.     if rst='1' then
26.       R64 <= (others => '0');
27.       R63 <= (others => '0');
28.     elsif falling_edge ( clk ) then
29.       if write_enable = '1' then
30.         R64 <= R64A( 55 downto 0 ) & d;
31.         R63 <= R63A( 54 downto 0 ) & R64A( 63 downto 56 );
32.       else
33.         R64 <= P & R64A(63 downto 1);   -- shift operation
34.         R63 <= Q & R63A(62 downto 1);   -- shift operation
35.       end if;
36.     end if;
37.   end process crp;
38.   P <= R64(63) xor R64(1) xor R64(0); -- P(X)
39.   Q <= R63(62) xor R63(6) xor R63(5) xor R63(3)xor R63(0);
40.   dout <= P xor Q;
41. end verhalten;
```

Listing 8.3 *Verschlüsselungseinheit des Crypto-UART.*

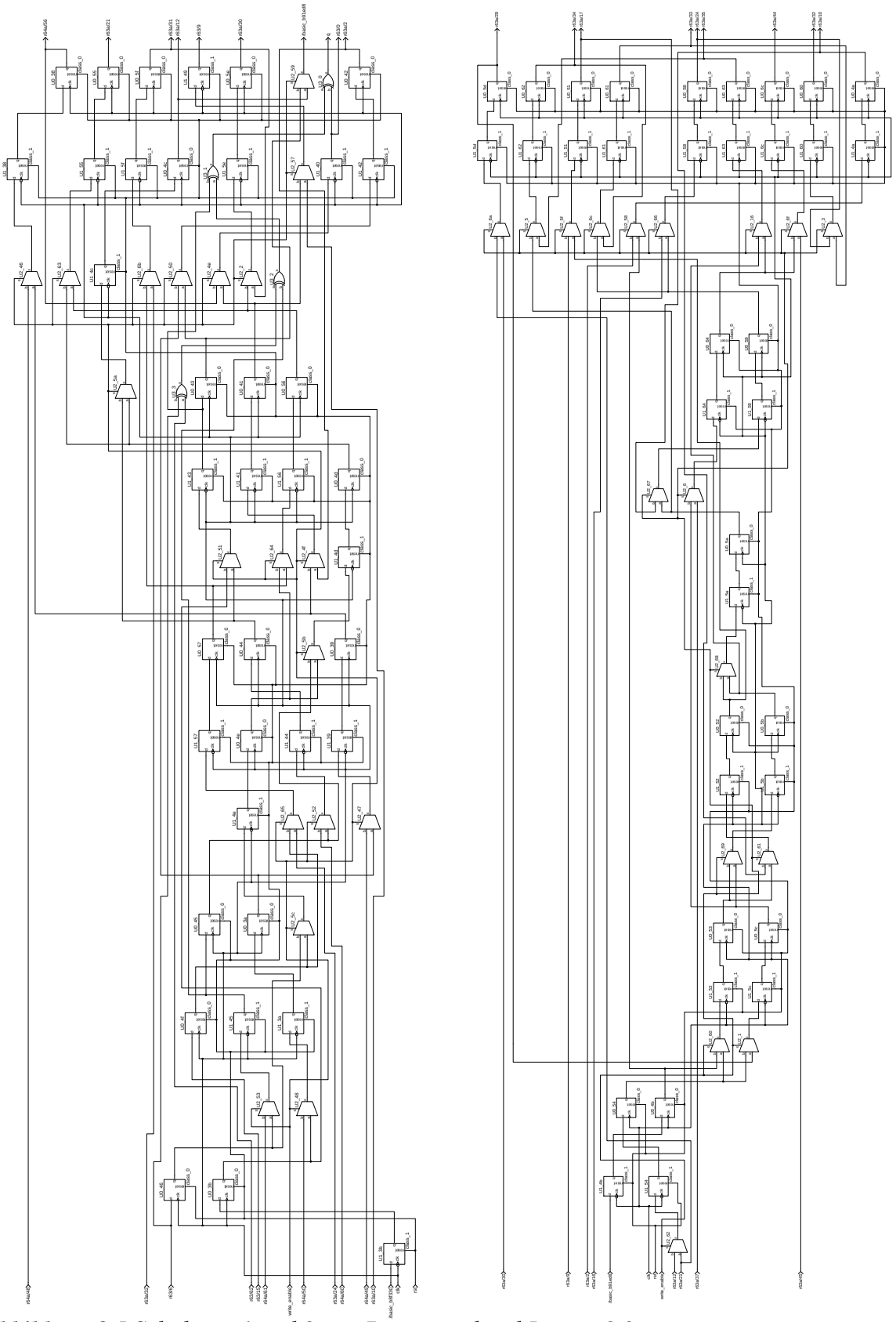


Abbildung 8.5 *Schaltung 1 und 2 von 7, entsprechend Listing 8.2.*

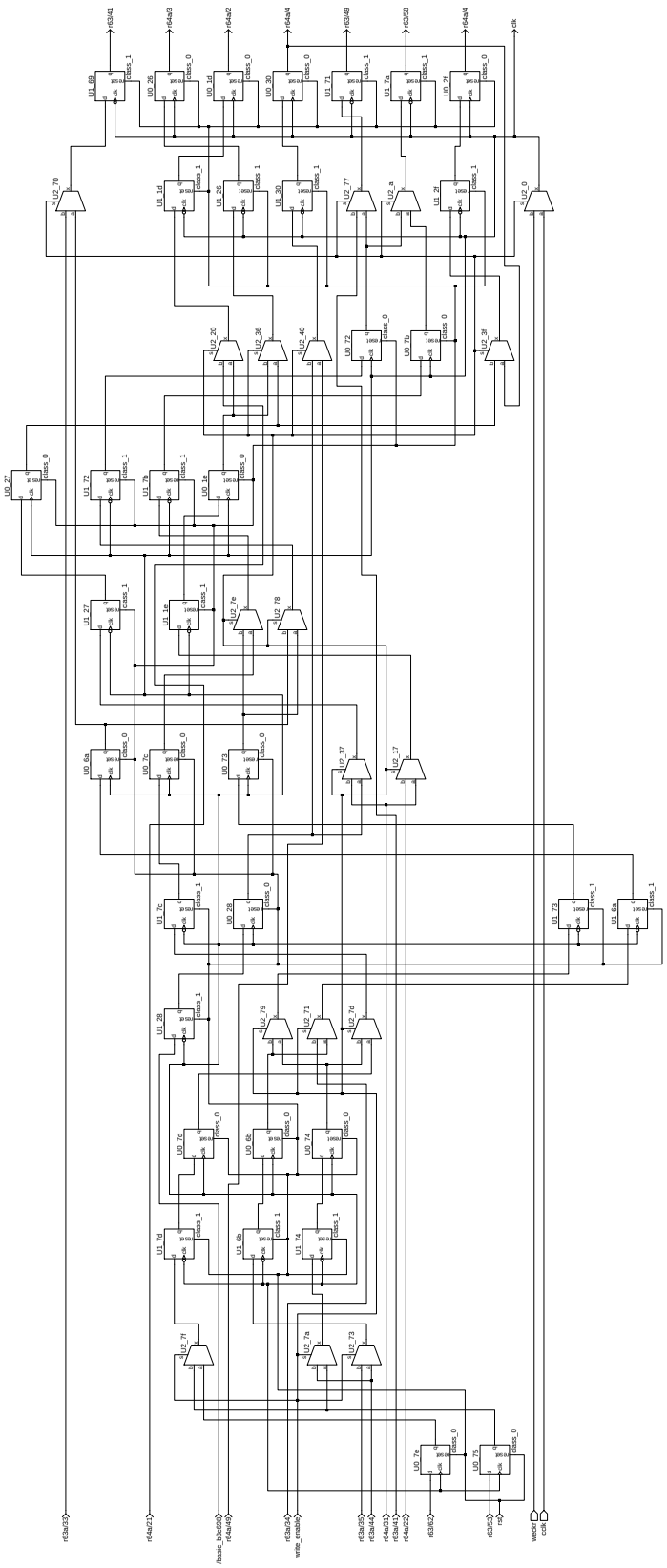


Abbildung 8.6 *Schaltung 3 von 7, entsprechend Listing 8.2.*

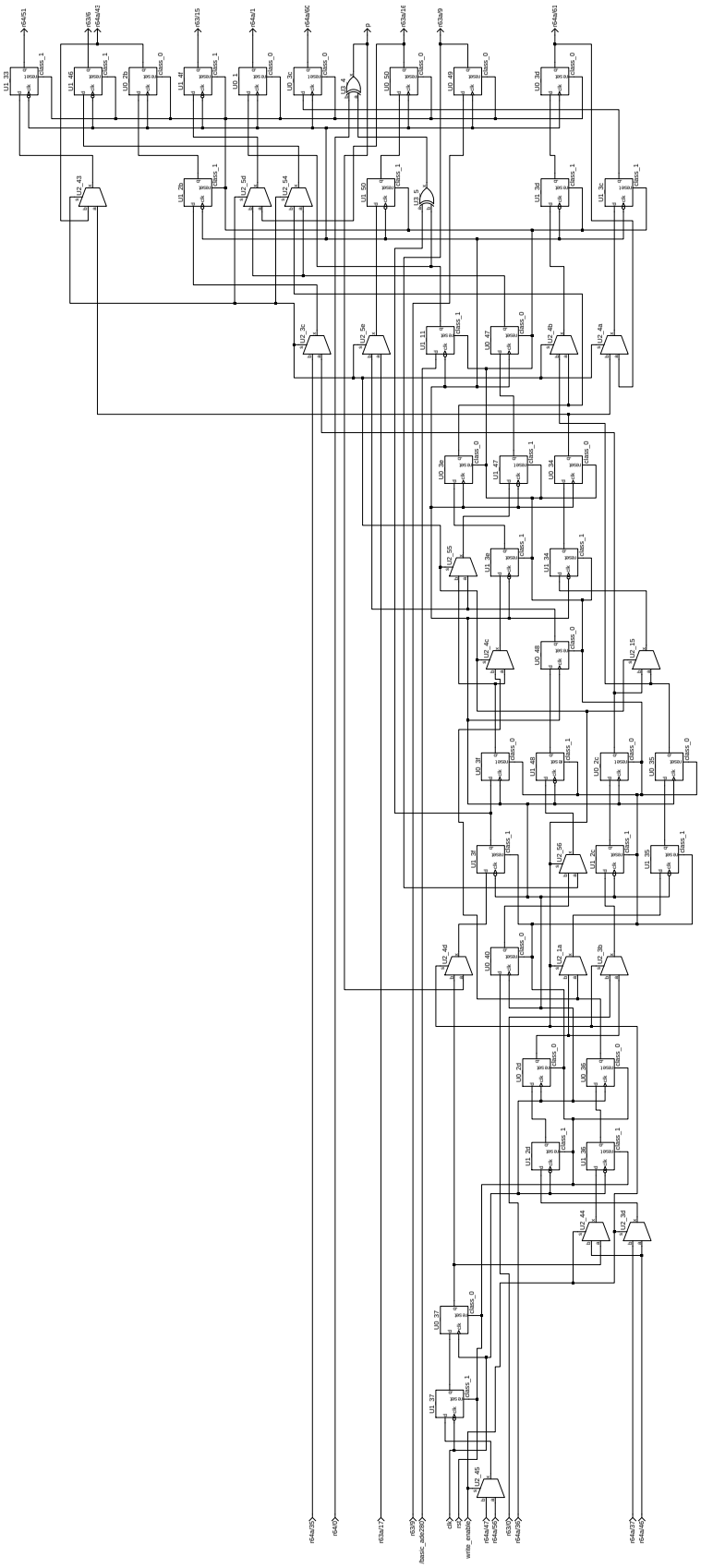


Abbildung 8.7 *Schaltung 4 von 7, entsprechend Listing 8.2.*

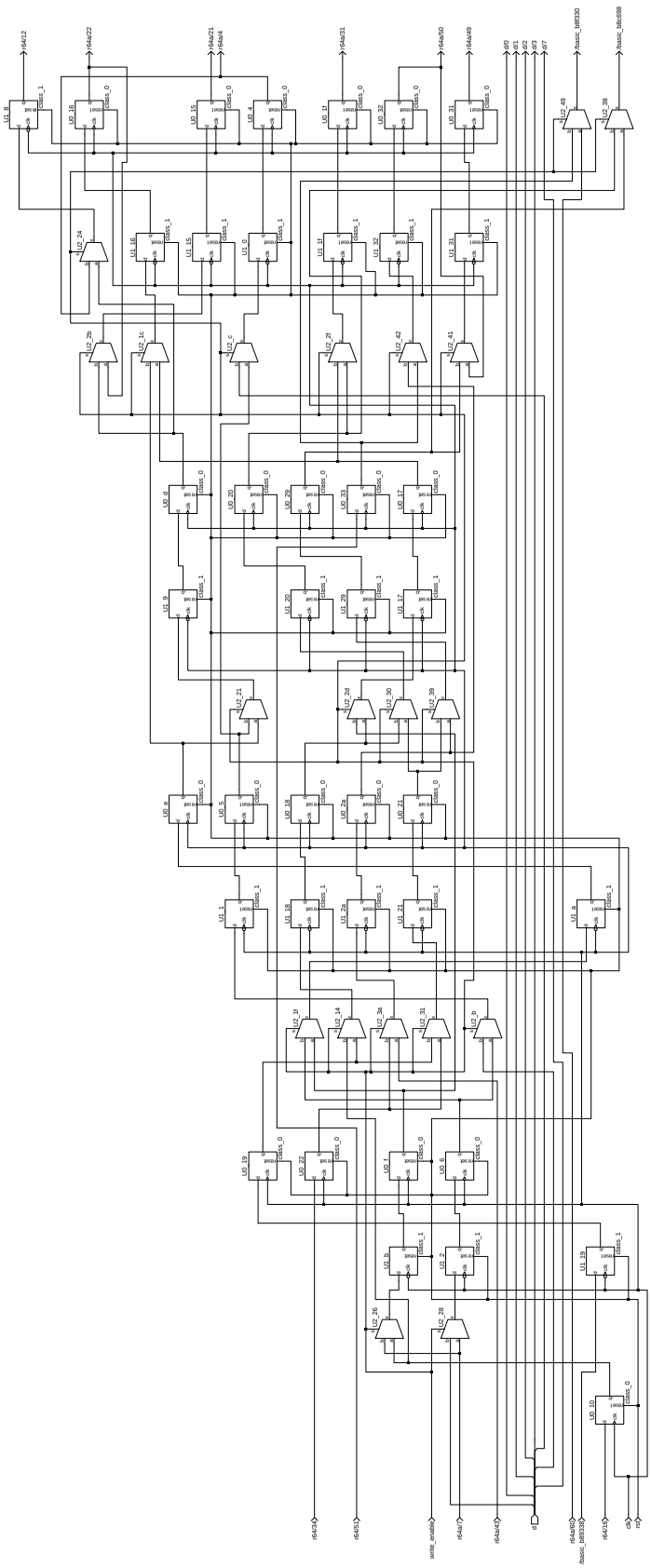


Abbildung 8.8 *Schaltung 5 von 7, entsprechend Listing 8.2.*

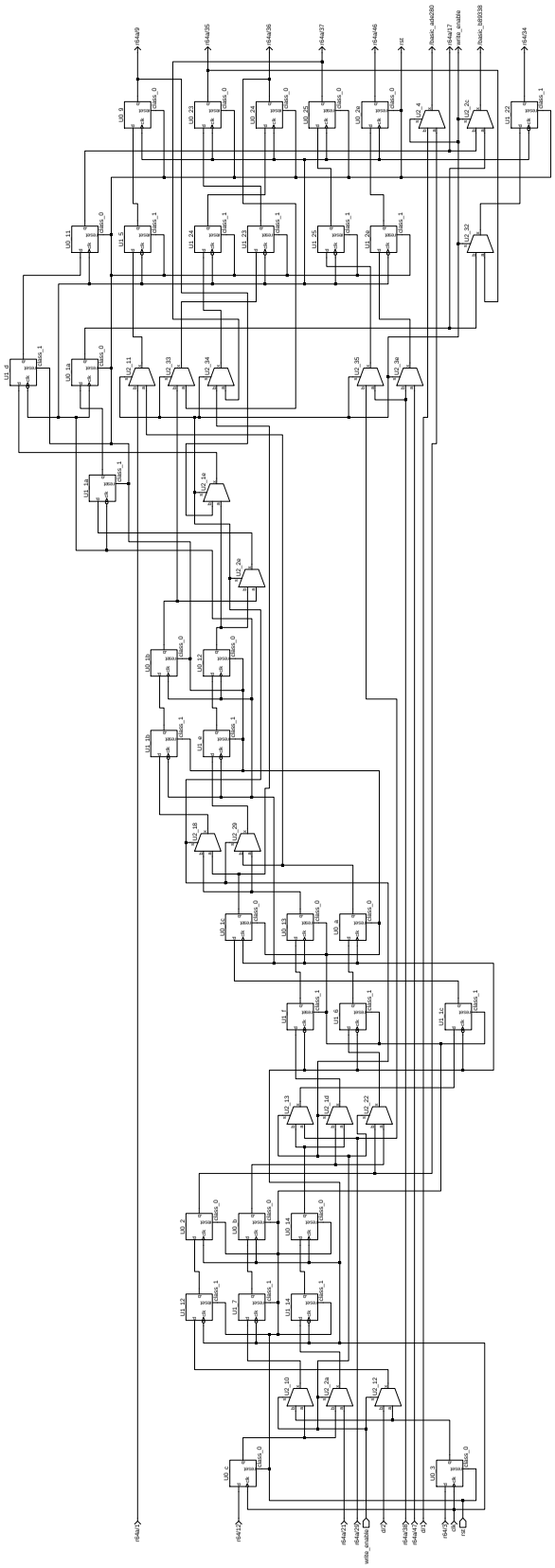


Abbildung 8.9 *Schaltung 6 von 7, entsprechend Listing 8.2.*

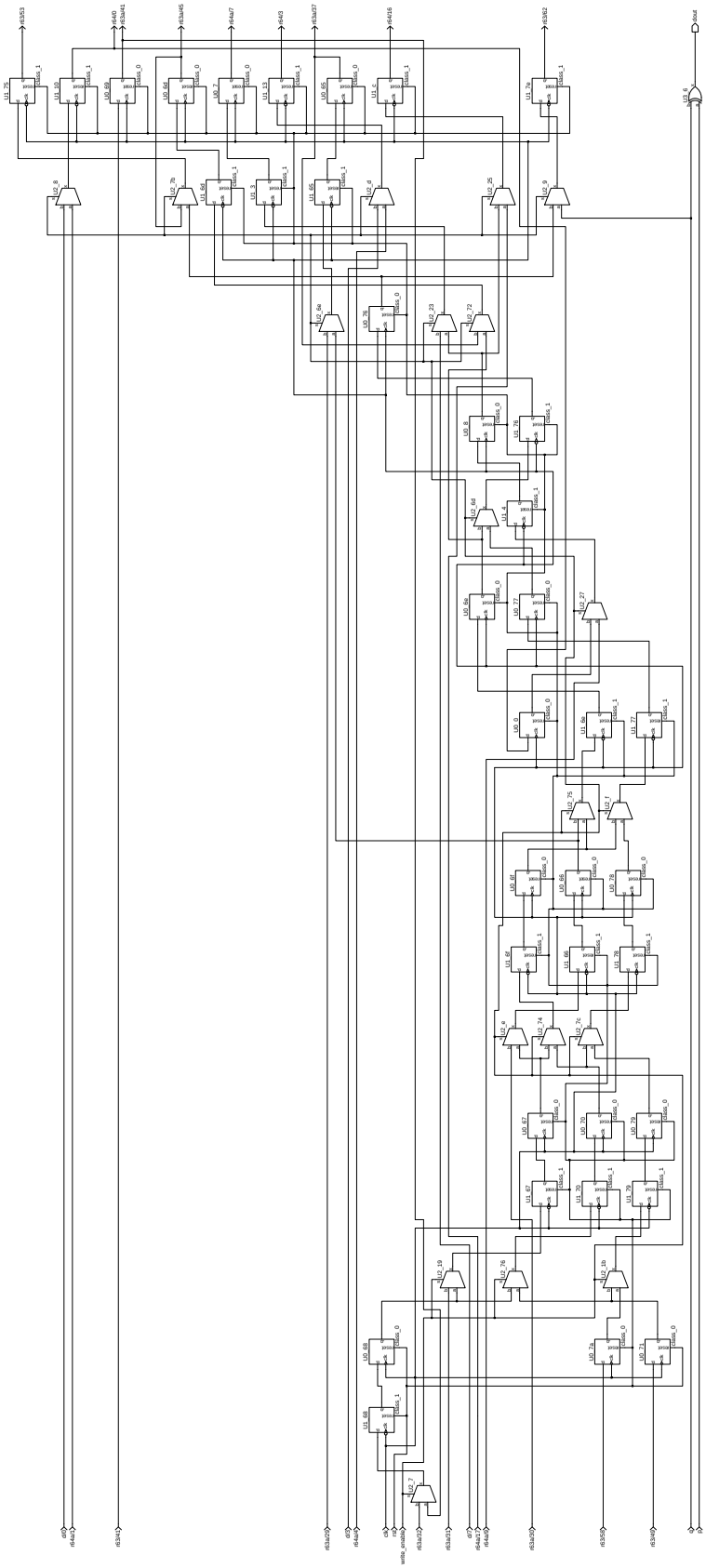


Abbildung 8.10 *Schaltung 7 von 7, entsprechend Listing 8.2.*

B.3 KDS Double-Data-Rate-Register

In Listing 8.4 ist eine typische Beschreibung einer mit konventionellen Synthesewerkzeugen nicht zu übersetzenden Schaltung eines Double-Data-Rate-Registers, die im Rahmen der Veranstaltung „Komponenten digitaler Systeme“ von den Teilnehmern erarbeitet wurde gegeben. In Abbildung 8.11 ist das Ergebnis der Square-Dance-Logiksynthese gegeben.

```
1.  library ieee;
2.  use ieee.std_logic_1164.all;
3.
4.  entity kds_reg_ddin is
5.
6.    generic (
7.      width : integer := 8);
8.
9.    port (
10.     clk    : in  std_logic;
11.     reset  : in  std_logic;
12.     din    : in  std_logic_vector(width-1 downto 0);
13.     dout_h : out std_logic_vector(width-1 downto 0);
14.     dout_l : out std_logic_vector(width-1 downto 0)
15.    );
16.
17. end kds_reg_ddin;
18.
19. architecture behaviour of kds_reg_ddin is
20.
21.   signal dout_l_in : std_logic_vector(width-1 downto 0);
22.
23. begin  -- behaviour
24.
25.   reg_l1 : process( clk, reset )
26.   begin  -- process reg
27.     if reset = '0' then  -- asynchronous reset
28.       dout_l_in <= (others => '0');
29.       dout_h <= (others => '0');
30.       dout_l <= (others => '0');
31.     elsif falling_edge(clk) then  -- rising clock edge
32.       dout_l_in <= din;
33.     elsif rising_edge(clk) then  -- rising clock edge
34.       dout_l <= dout_l_in;
35.       dout_h <= din;
36.     end if;
37.   end process reg_l1;
38.
39. end behaviour;
```

Listing 8.4 *Double-Data-Rate-Register.*

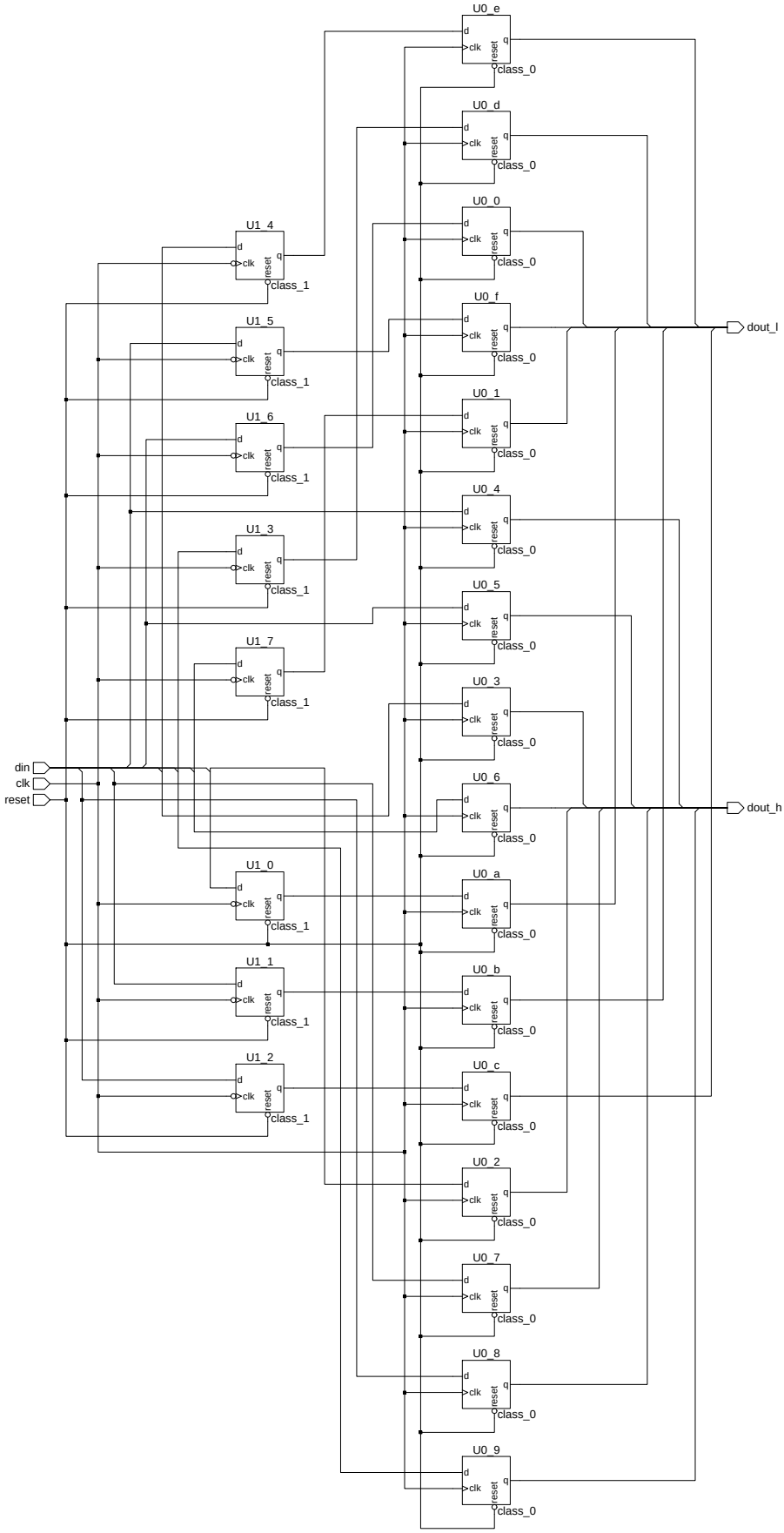


Abbildung 8.11 *Square-Dance-Logiksyntheseergebnis des in Listing 8.4 beschriebenen Double-Data-Rate-Registers.*

B.4 Logiksynthese mit generalisiertem Verfahren

Die in Listing 8.5 gegebene VHDL-Beschreibung ist mit konventionellen Synthesewerkzeugen nicht in eine entsprechende Schaltung übertragbar. In Abbildung 8.12 ist das entsprechende Square-Dance-Logiksyntheseergebnis abgebildet. In Hinsicht auf die hohe Komplexität der generierten Schaltung im Vergleich zur Ausgangsbeschreibung ist eine Überarbeitung der Beschreibung in Betracht zu ziehen.

```
1.  entity generic_demo is
2.    port (
3.      a, b, c, d : in std_logic;
4.      xout       : out std_logic
5.    );
6.  end generic_demo;
7.
8.  architecture behaviour of generic_demo is
9.
10. begin  -- behaviour
11.
12.   ff1: process ( a, b )
13.   begin
14.     if a'event then
15.       if a = '1' then
16.         xout <= c;
17.       elsif a = '0' then
18.         xout <= d;
19.       end if;
20.     end if;
21.     if b'event and b = '1' then
22.       xout <= c nand d;
23.     end if;
24.   end process ff1;
25.
26. end behaviour;
```

Listing 8.5 *VHDL-Beschreibung einer mit dem generalisierten Syntheseverfahren übersetzbaren Schaltung.*

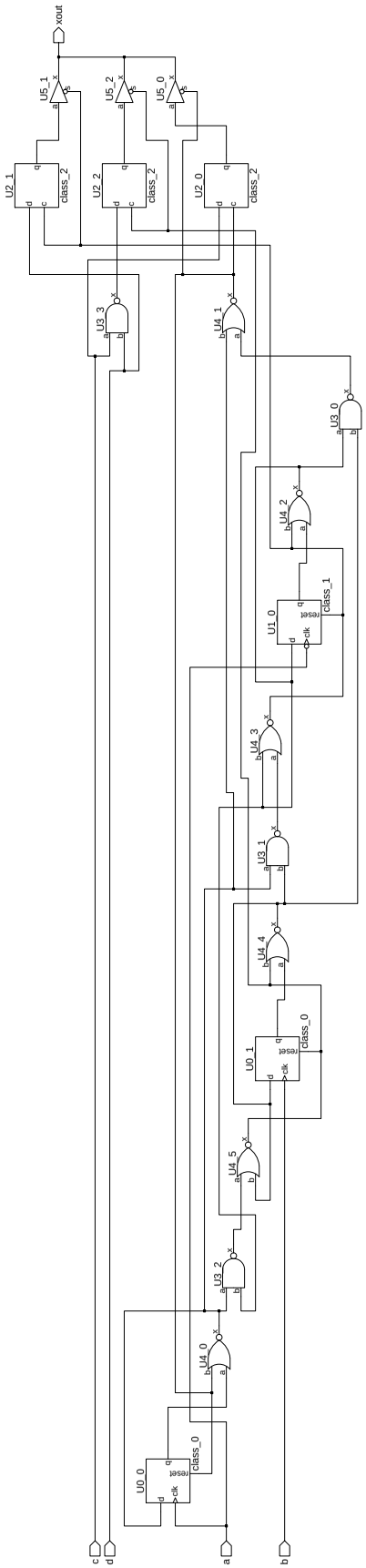


Abbildung 8.12 *Square-Dance-Logiksyntheseergebnis der in Listing 8.5 beschriebenen Schaltung.*

Literaturverzeichnis

Veröffentlichungen

[1] *Amnell T., David A., Fersman E.*:Tools for Real-Time UML: Formal Verification and Code Synthesis, Department of Information Technology, Uppsala University, August 2001.

[2] *Ananian C. S.*: Turning Java into Hardware: Caffinated Compiler Construction, http://cscott.net/Publications/, März 2002

[3] *Andersen H. R.*: An Introduction to Binary Decision Diagrams, Lecture notes for 49285 Advanced Algorithms E97, Oktober 1997.

[4] *Beierlein T., Fröhlich D., Seinbach B.*:Model-Driven Compilation of UML-Models for Reconfigurable Architectures, Technische Universität Bergakademie Freiberg, April 2004.

[5] *Bhasker J.*: Die VHDL-Syntax, 1. Auflage, Prentice Hall, München, 1996.

[6] *Björklund D., Lilius J.*:From UML Behavioral Descriptions to Efficient Synthesizable VHDL, TUCS Turku Centre for Computer Science, August 2002.

[7] *Bolton Institute and Northumbria University*: Advanced Electronic Design Automation, Unit 8 - VHDL for Synthesis, August 2003.

[8] *Brent R. P., Kung H. T.*:A regular layout for parallel adders, IEEE Trans. Comput., Band C-31, Nr. 3, S. 260-264, März 1982.

[9] *Center for Electronic Design, Communications, & Computing*: Synthesis Using VHDL, RASSP Education & Facilitation, Module 60, Version 1.2, 1998.

[10] *Chang-woo K., Massoud P.*: Technology Mapping for Low Leakage Power and High Speed with Hot-Carrier Effect Consideration, Dept. of Electrical Engineering Systems, University of Southern California, Februar 2004.

[11] *Clarke E. M., Fujita M., Rajan S. P., Reps T., Shankar S., Teitelbaum T.*: Program Slicing for VHDL, Software Tools for Technology Transfer manuscript, September 2004.

[12] *Cook S.A.*: The complexity of theorem-proving procedures, Proceedings of the Third Annual ACM Symposium on the Theory of Computing, New York, 1971.

[13] *Dantzig G., Rao M., Blattner W.*: All Shortest Routes from a Fixed Origin in a Graph, Proceedings of the International Symposium Rome, Dunod, Paris, 1966.

[14] *Davis D.*: Java for Digital Circuit Design - Using the Java Language and Environment for High-Level Circuit Design, Xilinx Forge Java Whitepaper.

[15] *De MicheliG., Brayton R.*: KISS: A Program for Optimal State Assignment of Finite State Machines, ICCAD Proceedings, Seiten 209-211, November 1984.

[16] *Department of Electrical Engineering and Computer Science University of California*: SIS: A System for Sequential Circuit Synthesis, Mai 1992.

[17] *Ehrenfest P.*: Review of L. Couturat, 'The Algebra of Logic', Journal Russian Physical & Chemical Society, 1910.

[18] *Elmasri R., Navathe S. B.*: Grundlagen von Datenbanksystemen, Pearson Studium, 2002.

[19] *Fant K. M., Brandt S. A.*: NULL Convention Logic, Theseus Logic Inc. 1997.

[20] *Fuhrer R., Nowick S.* M.: Symbolic hazardfree minimization and encoding of asynchronous finite state machines, Proc. International Conf. Computer-Aided Design, 1995.

[21] *Fujitsu*: New SoC Design Methodology based on UML and C Programing Languages, Dezember 2002.

[22] *Furber S. B., Garside J. D., Gilbert D. A.*:AMULET3: A High-Performance Self-Timed ARM Microprocessor Proceedings, ICCD, Oktober 1998.

[23] *Goossens G., Bolsens I., Lin B., Catthoor F.*: Design of heterogeneous ICs for mobile and personal communication systems, Proceedings of the 1994 IEEE/ACM international conference on Computer-Aided Design, S. 524 - 531, 1994.

[24] *Gordon M.*: Linking higher order logic to binary decision diagrams, Proceedings of the Symposium in Celebration of the work of C.A.R. Hoare, September 1999.

[25] *Günter W., Höreth S.*: Some Common Synthesis-Simulation-Mismatches, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Shaker Verlag, Februar 2004.

[26] *Gupta R. K.*: Simulation and Synthesis using VHDL, Department of Computer Science and Engineering, University of California, Frühjahr 2003.

[27] *Gutsche J. F.*: Synthese eines asynchronen seriellen Interface-Bausteins mit einem Ver-/Entschlüsselungsalgorithmus, Diplomarbeit an der TU Berlin, November 1999.

[28] *Gremzow C.*: High-Level-Synthese aus flachen Kontroll-/Datenflussgraphen, Dissertation an der TU Berlin, Februar 2004.

[29] *Hansen E.*: Generierung einer synthesefähigen VHDL-Beschreibung aus abstrakten Kommunikationsmodellen, Diplomarbeit an der TU Berlin, Januar 2003.

[30] *Hauck S.*: Asynchronous Design Methodologies: An Overview, Proceedings of the IEEE Vol. 1, Januar 1995.

[31] *Hutle M.*: Asynchrone Prozessoren, Seminarausarbeitung, TU Wien, Februar 2000.

[32] *Johnson S.D.*: Synthesis of Digital Designs from Recursion Equations, The MIT Press, 1983.

[33] *Katz R. H., Cummings B.*: Contemporary Logic Design, Addison Wesley Publishing Company, 1993.

[34] *Kessler R.*: Ein System zur kompakten und rechenzeiteffizienten VLSI-Layouttranslation unter Berücksichtigung von Regelwissen, Dissertation an der TU Berlin, Juni 1989.

[35] *Kommu V., Pomerang I.:* GAFPGA Generic Algorithm for FPGA Technology Mapping, Department of Electrical and Computer Engineering, University of Iowa, Februar 1998.

[36] *Kraus O., Padeffke M.*: Entwurfsumgebung für asynchrone Burst-Mode Automaten, Lehrstuhl für rechnergestützten Schaltungsentwurf, Universität Erlangen, Januar 2002.

[37] *Kraus O., Padeffke M.*:Synthese von asynchronen Burst-Mode Automaten, Lehrstuhl für rechnergestützten Schaltungsentwurf, Universität Erlangen, April 2003.

[38] *Lasser D. J.*: Topological Ordering of a List of Randomly-Numbered Elements of a Network, Communications of the ACM 4, S. 167-168, 1961.

[39] *Lehmann G.*: Schaltungsdesign mit VHDL, Franzis'-Verlag, Poing, 1994

[40] *Lewis J., Vinaya K.*: Extension to the VHDL Synthesis Standard, SynthWorks Design Inc. / Cadence Design Systems, 2003.

[41] *Lewis J.*: Dual-Edge triggered FF, 1076.6 RTL Style, SynthWorks Design Inc., Mailingliste, EDA Industry Working Groups, April 2003.

[42] *Liebig H., Thome S.*: Logischer Entwurf digitaler Systeme, 3. Auflage, Springer-Verlag, Berlin, 1996.

[43] *Lis S. L., Gajski D. D.*: Synthesis from VHDL, Proceedings of the 1988 IEEE International Conference, Oktober 1988.

[44] *Meinel C., Theobald T.*: Algorithms and Data Structures in VLSI Design, Springer-Verlag Berlin Heidelberg, 1998.

[45] *Mentor Graphics*: SystemC Verification with ModelSim, Oktober 2003.

[46] *Mills D., Cummings C. E.*: RTL Coding Styles That Yield Simulation and Synthesis Mismatches, 1999.

[47] *Muller D. E., Bartky W. S.*: A Theory of Asynchronous Circuits, Proceedings of an International Symposium on the Theory of Switching, Harvard University Press. S. 204-243, 1959.

[48] *Myers C. J.*: Asynchronous Circuit Design, Wiley US, Juli 2001.

[49] *Neumann I.*: Eine Methode zur laufzeitorientierten Schaltungsoptimierung während des Layout-Entwurfs, Dissertation an der TU Berlin, Dezember 1998.

[50] *Nguyen K. D., Sun Z., Thiagarajan P. S.*: Model-driven SoC Design Via Executable UML to SystemC, School of Computing National University of Singapore, August 2004.

[51] *Nowick S. M.*: Automatic Synthesis of Burst-Mode Asynchronous Controllers, überarbeitete Dissertation an der Stanford University, 1995.

[52] Perl J.: Graphentheorie - Grundlagen und Anwendungen, Akademie Verlagsgesellschaft, 1981.

[53] *Post H.-U.*:Rechnertechnologie, Vorlesungsskript TU Berlin, SS 2002.

[54] *Rajaraman R., Wong D. F.*: Optimum Clustering for Delay Minimization, IEEE Trans. On Computer-Aided Design of Integrated Circuits and Systems, Band 14, Nr. 12, Seiten 1490-1495, December 1995.

[55] *Randal E. Bryant*: Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers, 8(C-35), 1992.

[56] *Reifschneider N.* : CAE- gestützte IC-Entwurfsmethoden, Prentice Hall, 1998.

[57] *Schallenberg A., Nebel W., von Ossietzky C., Oppenheimer F.*: Designing for dynamic partially reconfigurable FPGAs with SystemC and OSSS, OFFIS Institute Oldenburg, Juni 2004.

[58] *Schiele W.*: Ansätze zur zweidimensionalen Kompaktierung und Knickgenerierung, E.I.S. Workshop, März 1986.

[59] *Schoppa I.*: Synthese von Registertransferstrukturen auf Basis konventioneller Programmiersprachen, Dissertation an der TU Berlin, Oktober 1998.

[60] *Siepmann E. Zimmermann G.*: An Object-Oriented Datamodel for the VLSI Design System PLAYOUT, Proceedings 26. Design Automation Conference, Seiten 814-817, 1989.

[61] *Sutherland I.E.*: micropipelines, Communications of the ACM, Vol. 32, No. 6, Juni 1989.

[62] *Villarreal J., Lysecky R.*: A Study on the Loop Behavior of Embedded Programs, University of California, Technical Report UCR-CSE-01-03, Dezember 2001.

[63] *Wang C.-Y., Parker D., Jorgenson R., Fant K.*: Technology Independent Design Using NULL Convention Logic, Theseus Logic Inc., 1998.

[64] *Weiser M.*: Program slicing, IEEE Transactions on Software Engineering, Vol. SE-10, No. 4. S. 352-357, Juli 1984.

[65] *Wuu T.-Y., Vrudhula S. B. K.*: Synthesis of Asynchronous Systems from Data Flow Specifications, ISI Research Report, 1993.

[66] *Wuttke H.-D., Henke K.*: Schaltsysteme, Eine automatenorientierte Einführung, Pearson Studium, 2003.

[67] *Yeh C., Chang C. C., Wang J. S.*: Power-driven technology mapping using pattern oriented power modelling, IEE Proceedings online no. 19990199, Januar 1999.

[68] *Zeller A., Krinke J.*: Open-Source-Programmierwerkzeuge, dpubkt verlag, Dezember 2003.

Standards

[69] *ANSI*: Database Language SQL,X3.135-1992, ISO/IEC 9075:1992

[70] *ANSI*: Programing languages - C++, ISO/IEC 14882:2003

[71] *IEEE Computer Society*: IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language
IEEE Std 1364-1995, Dezember 2002.

[72] *IEEE Computer Society*: IEEE Standard VHDL Language Reference Manual,
IEEE Std 1076-2002, März 2002.

[73] *IEEE Computer Society*: IEEE Standard for VHDL Register Transfer Level (RTL) Synthesis, IEEE Std 1076.6-2004, Juli 2004.

[74] *IEEE Computer Society*: IEEE Standard for VITAL ASIC Modeling Specification, IEEE Std 1076.4-2000, September 2001.

[75] *IEEE Computer Society*: IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE Std 1149.1-2001, Juni 2001.

Anbieter und Produktbeschreibungen

[76] *Alliance*: http://www-asim.lip6.fr/recherche/alliance/

[77] *AMULET*: http://www.cs.man.ac.uk/apt/projects/processors/amulet/

[78] *ARM*: http://www.arm.com/

[79] *Artisan Components*: TSMC 0.13µm (CL013G) Process 1.2-Volt SAGE-XTM Standard Cell Library Databook, August 2001.

[80] *Blast Fusion*:
http://www.magma-da.com/c/@7BqBr3VinnLQU/Pages/blastfusion.html

[81] *Blast RTL*: http://www.magma-da.com/c/@7BqBr3VinnLQU/Pages/blastrtl.html

[82] *C++ Reference*: http://www.cppreference.com/

[83] *CodeModeller*: http://www.arubadev.com/

[84] *CoCentric*: http://www.synopsys.com/C-level.html

[85] *Concept Engineering GmbH*: http://www.concept.de/

[86] *Cyntesizer*: http://www.forteds.com/products/cynthesizer.asp

[87] *DC Expert*: http://www.synopsys.com/products/logic/design_compiler.html

[88] *Electric:* http://www.staticfreesoft.com/

[89] *Encounter digital IC design platform*:
http://www.cadence.com/products/digital_ic/index.aspx?lid=dic

[90] *flex++ - bison++*: ftp://ftp.th-darmstadt.de/pub/programming/languages/C++/tools/flex++bison++/LATEST/

[91] *Fujitsu*: CE81 Series Embedded Array, 0.18 um CMOS-Technology, Fujitsu Microelectronics America Inc., 1999.

[92] *FPGAExpress*: http://www.xilinx.com/prs_rls/xpress2.htm

[93] *Free Model Foundry*: http://www.eda.org/fmf

[94] *Icarus Verilog*: http://www.icarus.com/eda/verilog/

[95] *ISE Alliance/Foundation*: http://www.xilinx.com/products/design_resources/design_tool/index.htm

[96] *LeonardoSpectrum*: http://www.mentor.com/products/fpga_pld/synthesis/leonardo_spectrum/

[97] *libqt*: http://www.trolltech.com/

[98] *MySQL*: http://www.mysql.com/

[99] *Octtools 5.2*: http://www.eecs.berkeley.edu/IPRO/Software/Catalog/Description/octtools5.2.html

[100] *Oracle*: http://www.oracle.com/

[101] *SAP DB*: http://www.sapdb.org/

[102] *Synopsys Inc.*: Guide to HDL Coding Styles for Synthesis, Juni 2002.

[103] *Synopsys Inc.*: VHDL Compiler Reference Manual, 1999.

[104] *Theseus Logic*: http://www.theseus.com/

[105] *Trolltech*: http://www.trolltech.com/

[106] *Unimodeler*: http://www.unimodeler.com/

[107] *Verilog*: http://www.verilog.org

[108] *PtolemyII*: http://ptolemy.eecs.berkeley.edu/ptolemyII/

[109] *RASP*: http://ballade.cs.ucla.edu/software_release/rasp/htdocs/

[110] *STL*: http://www.sgi.com/tech/stl/

[111] *SystemC Community*: https://www.systemc.org/

[112] *Wikipedia*: http://de.wikipedia.org

[113] *Xilinx*: The Programmable Logic Data Book, Xilinx Inc. San Jose, 1994.

[114] *Xilinx*: XC3000 Series Field Programmable Gate Arrays, Xilinx Inc. Dezember 1999.