Martin Padeffke

Entwurfsverfahren für asynchrone Schaltungen unter Verwendung von Standardsoftware

Cuvillier Verlag Göttingen

Entwurfsverfahren für asynchrone Schaltungen unter Verwendung von Standardsoftware

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Martin Padeffke

Erlangen - 2004

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2005
Zugl.: Erlangen-Nürnberg, Univ., Diss., 2004
ISBN 3-86537-471-9

Als Dissertation genehmigt von der Technischen
Fakultät der Universität Erlangen-Nürnberg

Tag der Einreichung:	03. November 2004
Tag der Promotion:	6. April 2005
Dekan:	Prof. Dr. Albrecht Winnacker
Berichterstatter:	Prof. Dr. Wolfram Glauert, Prof. Dr. Sorin Huss

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2005
Gedruckt auf säurefreiem Papier

ISBN 3-86537-471-9

Die vorliegende Arbeit ist im Rahmen einer Doktorarbeit am Lehrstuhl für Rechnergestützten Schaltungsentwurf an der Friedrich-Alexander-Universität Erlangen-Nürnberg entstanden.

Ich bedanke mich bei Prof. Dr.-Ing. Wolfram H. Glauert für die Betreuung und Begutachtung dieser Dissertation.

Für die Übernahme der Anfertigung des Zweitgutachtens und sein Interesse an dieser Arbeit danke ich Prof. Dr.-Ing. Sorin A. Huss. Bei Prof. Erich Wehrhahn bedanke ich mich für die intensive Zusammenarbeit, sein Engagement und Anregungen. Weiterer Dank gilt den Studierenden, die mit Studien- und Diplomarbeiten an Lehrstuhl für Rechnergestützten Schaltungsentwurf im Rahmen einer Projektgruppe zu asynchronen Schaltungen wichtige Beiträge geleistet haben.

Den Kolleginnen und Kollegen am Lehrstuhl möchte ich herzlich für die sehr gute Zusammenarbeit und Hilfsbereitschaft danken. Ein besonderer Dank geht dabei an Oliver Kraus, der mit seinem Wissen, seiner eigenen Arbeit und Bereitschaft zum offenen Meinungsaustausch zum Gelingen dieser Arbeit beigetragen hat.

Dank auch meinen Eltern für die stetige Unterstützung.

Der größte Dank gilt jedoch meiner geliebten Frau für Ihre tatkräftige Motivation und aktive Unterstützung sowie meinen Kindern, die mich täglich an die wichtigen Dinge im Leben erinnern.

Inhalt

Abstract

This work describes the systematic (semi-) automatic design of asynchronous circuits in a design environment for synchronous circuits. The methodology developed in this work allows to implement any behavioural model in stages of an asynchronous pipeline. Models can therefore be described with standard hardware description languages (VHDL, Verilog, EDIF) and are allowed to have storing elements. As a result a gate netlist is obtained which timing behaviour is validated by a pre-layout timing simulation. Controlling of each stage can be done with standardised or new defined control automata. DGC [45] which can generate hazard free circuits is used to synthesise the new defined control automata. Other synthesis steps and simulation are done with the common tool synopsys [94]. Integrating the asynchronous moduls into an synchronous design is possible without problems. Further an extensive catalogue of controllers for the four phase protocol was created. Additional interface modules are also part of the catalogue. These interface modules are for example connections to synchronous modules. The capacity of the software ASMOGEN as a part of this work is shown in a comparison between a synchronous implementation and an asynchronous one of a Reed Solomon decoder (symbolwidth is 12 bits; wordlength is 2720 symbols).

Kurzzusammenfassung

Diese Arbeit beschreibt den systematischen (semi-) automatischen Entwurf asynchroner Schaltungen in einer Entwurfsumgebung für synchrone Schaltungen. Die erarbeitete Methodik erlaubt es, beliebige Verhaltensmodelle in asynchrone Piplinestufen zu realisieren. Die Modelle können hierzu in einer standardmäßigen Hardware-Beschreibungssprache (VHDL, Verilog, EDIF) verfasst und nach Wunsch auch mit Speichern versehen sein. Als Ergebnis erhält man eine Gatternetzliste, die auf ihr Zeitverhalten mittels einer Pre-Layout-Simulation überprüft ist. Die Steuerung der einzelnen Stufen kann über standardisierte oder selbst vorgegeben Kontrollautomaten erfolgen. Zur Synthese der selbst vorgegeben Kontrollstrukturen wird die frei verfügbare Synthesesoftware DGC [45] verwendet, die hazardfreie Schaltungen erzeugen kann. Für die restlichen Syntheseschritte und Simulationen wird die Standardsoftware Synopsys [94] verwendet. Eine Einbindung der asynchronen Module in eine synchrone Einsatzumgebung ist ohne Probleme möglich. Im Rahmen dieser Arbeit ist ein umfangreicher Katalog an Kontrollstrukturen für das Vierphasenprotokoll entstanden. Zu diesem Katalog zählen auch weitere Schnittstellenmodule, wie z.B. für die Anbindung an synchrone Module. Die Leistungsfähigkeit der erstellten Software ASMOGEN wurde anhand des Vergleiches der asynchronen Implementierung mit der synchronen Implementierung eines Read Solomon Dekoders (Symbolbreite ist 12 Bits, Wortlänge ist 2720 Symbole) nachgewiesen.

1 Einführung

Die minimal mögliche Strukturgröße innerhalb einer integrierten Schaltung sinkt über die Zeit exponentiell. In [36] wird die Entwicklung von Mikroprozessoren von 1997 bis 1999 und eine Prognose bis 2002 gezeigt. Die Anzahl der Transistoren pro Mikroprozessoren nimmt stetig zu. Bis 2002 führt dieser Anstieg zu geschätzten 130 Millionen Transistoren auf einem einzigen Chip. Diese Entwicklung führt zu Problemen, da die Entwickler nicht in dem gleichen Maß ihre individuelle Produktivität steigern können. Hier müssen entsprechend neue Softwarewerkzeuge mit höherem Automatisierungsgrad sowie neue Methoden und Wege der Entwicklung helfen, diese Lücke der Entwicklerproduktivität zu der Komplexität der zu entwerfenden Bausteine zu verringern.

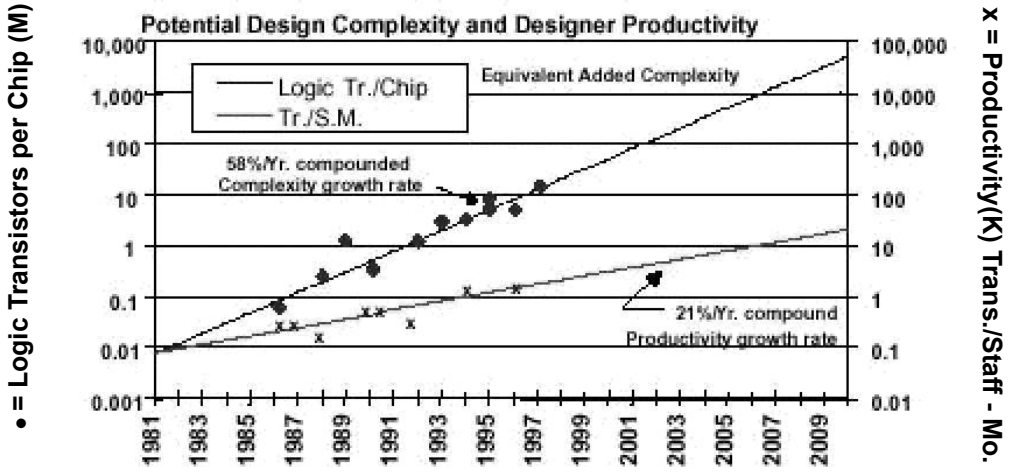


Bild 1.1: Lücke in der Entwurfsproduktivität (Figure5 ITRS)

Um den Sachverhalt und die Implikationen dieser Steigerung der Komplexität zu verdeutlichen, gibt es einen anschaulichen Vergleich mit Straßennetzen. Während die Komplexität einer integrierten Schaltung des Jahres 1963 in einer 25 µm CMOS-Technologie dem Straßennetz einer mittelgroßen Stadt mit ca. 4 km^2 Fläche entsprach, müsste man die Komplexität einer integrierten Schaltung des Jahres 1993 in einer 0,5 µm CMOS-Technologie mit einem Straßennetz vergleichen, welches mit ca. 9 Millionen km^2 schon die Größe Europas überbietet. Mit einer 0,13 µm CMOS-Technologie im Jahr 2001 umspannt das vergleichbare Straßennetz mit etwa 2,3 Milliarden km^2 schon ein Vielfaches der Erdoberfläche (~500 Millionen km^2).

Um den Vergleich zu Ende zu führen, muss man sich jetzt vorstellen, dass letzteres Straßensystem mit nur einem einzigen zentralen Signal gesteuert wird. Für ein Straßennetz erscheint dies unmöglich, in integrierten Schaltungen wird aber genau dieses gemacht. Nahezu immer steuert ein einziges globales Takt-Signal den gesamten Datenverkehr auf einem Baustein. Dies ist aus elektrischen Gründen auch hier nicht ohne erhebliche Probleme und Entwurfsaufwand zu erreichen.

Auch in [36] werden diese Probleme festgehalten und eine mögliche Lösung angegeben:

„*The fastest buffered interconnect for a long wire can be over 100x the switching time of an individual gate. Given expected global clock rates approaching 3 GHz, it will be difficult to achieve synchronous, on-chip operation without introducing multi-cycle latencies in the interconnect. System design must comprehend the timing issue more fully*“

„*The large currents being introduced with increasing power densities and lower voltages all lead to larger supply rail inductive noise. Synchronous systems worsen the problem by scheduling the power switch surges around regular time periods. Thus, supply rail design to reduce effects such as voltage drop or current surges are required early in the design process.*“

„*Process technology support for increased switching is not keeping up with the trend in clock rate increase. Thus, design techniques such as reducing the number of logic levels between clocked registers allow the current trend for clock rate increase. Careful consideration of single versus multi-cycle paths and circuits will become more important early in the design process. Specifically, locally synchronous but globally asynchronous design techniques will need to be supported by the tools.*“

Ein vorgeschlagener möglicher Lösungsweg aus den oben beschriebenen Problemen ist also die Nutzung asynchroner Schaltungskonzepte. Das heißt, dass nicht mehr ein einzelnes globales Signal den Datentransfer steuert, vielmehr wird es lokale beziehungsweise lokal erzeugte Steuersignale geben, die jeweils ein Modul steuern. Letztendlich ist der integrierte Baustein aus einzelnen Modulen aufgebaut, die ihrerseits in sich synchron arbeiten. Die Kommunikation zwischen den Modulen findet aber asynchron statt. Dabei kann die Größe der einzelnen Module unterschiedlich sein und auch die Module selbst können asynchron arbeiten.

1.1 Motivation für den Entwurf asynchroner Schaltungen

Der Entwurf asynchroner Schaltungen wird im Moment nicht in der Weise unterstützt, wie der Entwurf synchroner Schaltungen. Dies liegt zum einen an der großen Menge an Erfahrungen, die sich in den Softwarefirmen und Hardwareentwicklungsfirmen gesammelt haben. Zum anderen reichten bisher synchrone Schaltungen weitestgehend aus. Mit den zunehmenden Problemen im Entwurf synchroner Schaltungen wird aber gerade die Nutzung asynchroner Schaltungen interessanter.

Als Hauptursache der Probleme synchroner Schaltungen ist der Takt selbst zu betrachten. Da die Schaltungen immer größer werden, müssen immer größere Taktnetze erzeugt werden. Dieses bedingt ein immer größer werdendes Taktnetz, welches auszubalancieren ist. Das wieder bedeutet einen erhöhten Energieverbrauch und auch Platzbedarf. Neben den aufkommenden Designschwierigkeiten sind noch andere Aspekte eines globalen Taktes zu betrachten. Das sind

zum Beispiel die Elektromagnetische Verträglichkeit oder auch der Schutz der Informationen vor unerwünschtem Zugriff.

Im Einzelnen erhofft man sich durch den Wegfall eines globalen Taktes mehrere Vorteile.

1.1.1 Vereinfachung globaler Zeitaspekte

Im synchronen Entwurf ist üblicherweise eine bestimmte Taktrate vorgegeben, die durch die spätere Einsatzumgebung bedingt ist. Innerhalb der synchronen Schaltung muss man alle Pfade so trimmen, dass die Propagierungszeit durch diese kürzer als die Taktperiode minus der Setup-Zeit des nachfolgenden Speicherelements ist. Im asynchronen Entwurf interessieren hier nur die absoluten Zeiten („bis wann muss der Ausgang reagiert haben?“), so dass hier weniger Pfade zeitlich zu optimieren sind.

1.1.2 Bessere Schnittstellen zur Umwelt

Ein weiteres Problem der synchronen Schaltung stellt die Einsynchronisation externer (asynchroner) Signale dar. Eine Bedingung, die dabei einzuhalten ist, ist der sichere gegenseitige Ausschluss, d.h. es darf nicht gleichzeitig von außen (Daten) und innen (Takt) ein Zugriff auf die Einsynchronisierungsflipflops erfolgen (Einhaltung der Setup- und Hold- Zeiten).

Bei einer asynchronen Schaltung entfällt dieses Problem im Allgemeinen, auch wenn hier trotzdem Zeitbedingungen auftreten, die eingehalten werden müssen.

1.1.3 Geschwindigkeit

Die in [36] angesprochenen Probleme der Laufzeiten auf Leitungen werden mit zunehmender Größe der integrierten Bausteine immer dramatischer. Die Gatterverzögerungszeiten nehmen durch Verbesserungen in der Prozesstechnologie im größeren Maße ab, als dies für die Laufzeiten auf den Verbindungsleitungen der Fall ist. Da somit das Verhältnis Leitungslaufzeit zu Gatterverzögerungszeit signifikant steigt, muss den Leitungslaufzeiten beim Entwurf mehr Aufmerksamkeit gewidmet werden.

Vorrangiges Ziel sind kurze Verbindungen zwischen den Gattern einer Schaltung. Die für diesen Optimierungsschritt relevanten Platzierungs- und Verdrahtungswerkzeuge benutzen über die Jahrzehnte verbesserte und ausgereifte Algorithmen. Allerdings sind diese Algorithmen bei der Verkürzung der Leitungslänge globaler Signale nutzlos, da diese globalen Signale meist auf dem gesamten Baustein verfügbar sein müssen. Bei einer synchronen Schaltung ist es vor allem die Taktleitung mit den entsprechenden Signallaufzeiten, die den maximal möglichen Datendurchsatz beschränken.

Asynchrone Schaltungen werden nicht zentral über einen globalen Takt gesteuert, sondern dezentral über lokale Steuerelemente. Somit kommen asynchrone Schaltungen ohne einen globalen Takt aus. Das Fehlen globaler Signale in asynchronen Schaltungen macht den Datendurchsatz unabhängig von den Laufzeiten globaler Signale. Auf dieser Tatsache beruht das Potential asynchroner Schaltungen, eine größere durchschnittliche Performanz zu erreichen, als die entsprechenden synchronen Implementierungen.

1.1.4 Keine Clock Skew Probleme

Eine synchrone Schaltung basiert auf der „zeitgleichen“ Abarbeitung der Daten, zum Beispiel gesteuert durch eine steigende Flanke eines Taktsignals. Der hierfür verwendete Takt muss aber im gesamten Entwurf gleichzeitig aktiv sein. Der tatsächlich vorhandene Zeitunterschied des Auftretens dieser Taktflanke an unterschiedlichen Orten bezeichnet man mit *Clock Skew*, beziehungsweise *Taktversatz*. Bei der zunehmenden Fläche heutiger integrierter Schaltungen bei gleichzeitiger Miniaturisierung der Schaltelemente nehmen die relativen Leitungslängen und damit die Signallaufzeiten zu. Somit wird es auch immer schwieriger, die Bedingung der Gleichzeitigkeit zu erfüllen.

Während hier bei synchronen Schaltungen ein nicht unerheblicher Entwurfsaufwand entsteht, fällt dieser bei asynchronen Schaltungen weg, da kein globaler Takt zum Einsatz kommt. Die lokal erzeugten Takte und die für einzelne (kleine) Module benötigten (globalen) Takte kann man wesentlich einfacher implementieren.

1.1.5 Durchschnittliche Performanz

Mit der Entscheidung, alles mit einem globalen Takt zu steuern, steht beim synchronen Entwurf ein weiterer Nachteil in Verbindung. Die Taktfrequenz kann nicht höher sein, als es der längste Pfad der Schaltung zulässt. Erst wenn Daten durch diesen propagiert sind und am Ende dieses Pfades für die Zeit T_{setup} stabil anliegen, darf die nächste Taktflanke auftreten. Damit bestimmt die größte Propagierungszeit die Performanz der Schaltung.

Viele asynchrone Schaltungen besitzen daher einen Mechanismus, der feststellt, wann ein Datum vollständig verarbeitet ist und als Ergebnis in der nächsten Stufe weiterverarbeitet werden kann. Da die Propagierungszeit datenabhängig ist, zeigt die asynchrone Schaltung eine von den Daten abhängige Performanz. Es ergibt sich daher für asynchrone Schaltungen eine bessere durchschnittliche Performanz.

1.1.6 Geringere Abhängigkeit von physikalischen Eigenschaften

Die Laufzeiten durch einen Pfad variieren auf Grund unterschiedlicher Umgebungsparameter (Temperatur, Versorgungsspannung, ...). In einer synchronen Schaltung muss man immer die schlechtesten Parameterkombinationen zur Bestimmung der maximalen Taktfrequenz und damit der Performanz der Schaltung betrachten.

Im Gegensatz zum synchronen Entwurf macht dies im asynchronen Entwurf keine größeren Probleme. Durch die verwendeten Mechanismen zur Detektion der vollständigen Abarbeitung von Daten ist eine asynchrone Schaltung unempfindlicher gegen globale Parameterschwankungen. Sie arbeitet weiterhin korrekt, wobei sich die Performanz der asynchronen Schaltung entsprechend den Parameterwerten verändert.

1.1.7 Unabhängiger von Technologieänderungen

Ein Problem des synchronen Entwurfs ist die Anpassung an eine neue Technologie beziehungsweise an den Herstellungsprozess eines anderen Anbieters. Es ändern sich die Laufzeiten durch die verschiedenen Pfade und dementsprechend muss man die Schaltung den neuen Bedingungen anpassen. Unter Umständen ist entweder eine niedrigere Taktfrequenz zu wählen (unrealistisch), oder die nun neu entstandenen längsten Laufzeiten müssen gekürzt werden. Es ist auf jeden Fall ein Eingriff in eine bestehende Schaltung notwendig.

Auch hier hat eine asynchrone Schaltung mit Komplettierungsdetektionsmechanismus den Vorteil, weniger abhängig von solchen Änderungen der Technologie oder des Herstellungsprozesses zu sein. Es sind hier keine oder nur minimale Änderungen notwendig.

1.1.8 Sicherheit

In einer synchronen Schaltung wird die Datenverarbeitung gleichzeitig mit der aktiven Taktflanke gestartet. Das kann einerseits zu Störungen umliegender Schaltungen führen. Andererseits können damit auch Informationen über die Funktionsweise des Bausteins gewonnen werden. Bei einer asynchronen Schaltung lässt sich das Spektrum verschmieren, was das Gewinnen von Informationen aus dem Betrieb des Bausteins wesentlich erschwert.

1.1.9 Bessere Elektromagnetische Verträglichkeit

In synchronen Schaltungen ist die Datenverarbeitung, gesteuert durch den globalen Takt, schrittweise durchzuführen. Mit jeder aktiven Taktflanke startet in der gesamten Schaltung gleichzeitig eine neue Berechnung. Dies führt zu hohen Versorgungsstromspitzen, die bei einer asynchronen Schaltung vermeidbar sind. Hier hat man die Möglichkeit die Abarbeitung der Daten so zu steuern, dass keine Störspitzen auftreten.

1.1.10 Low Power

Bei einer synchronen Schaltung entsteht durch kontinuierliche Pegelwechsel auf der Taktleitung Verlustleistung, auch wenn sich der Rest der Schaltung in „Ruhe“ befindet. Somit wird in einer synchronen Schaltung immer dann unnötig Energie verbraucht, wenn diese Schaltung nichts zu tun hat.

Allerdings kann der Energieaufwand des Taktnetzwerkes auch in einer „aktiven“ Schaltung bis zu 50% des Gesamtverbrauchs der Schaltung ausmachen. Also selbst wenn die Schaltung Daten verarbeitet, ist das Taktnetz immer noch der Hauptverbraucher von Energie.

Asynchrone Schaltungen besitzen keinen globalen Takt. Damit fällt ein wichtiger Leistungsverbraucher weg. Hat eine asynchrone Schaltung nichts zu tun, so befindet sie sich vollständig in Ruhe und es wird kaum Energie verbraucht (vergleiche Kapitel 2.6.2). Andererseits benötigen Daten verarbeitende asynchrone Schaltungen oftmals auch einen selbstgenerierten Takt, der also nur dann aktiv ist, wenn neue Daten anliegen. Da dieser selbstgenerierte Takt aber nur lokal benötigt wird, ist dessen Taktnetz mit weniger Treibern versehen, als ein Taktnetz in einer synchronen Schaltung. So verbrauchen diese lokalen Taktnetze in der Summer weniger Energie als ein vergleichbarer globaler Takt in einer synchronen Schaltung.

Somit sind asynchrone Schaltungen eine attraktive Alternative für Low Power Anwendungen.

1.2 Einordnung dieser Arbeit

Ziel dieser Arbeit ist es, den Entwurf asynchroner Schaltungen in einen Standard- Entwurfsablauf einzubinden, der für den Entwurf synchroner Schaltungen ausgearbeitet ist. Der Vorteil dieser Vorgehensweise liegt darin, dass der Hardwareentwickler in der gleichen Entwurfs- und Softwareumgebung sein gesamtes System entwickeln kann. Somit ergibt sich die Möglichkeit, einzelne asynchrone Module in ein System einzubauen, um zum Beispiel den Verbrauch des verbrauchsstärksten Moduls zu senken. Damit können die zu erwartenden Vorteile einer asynchronen Schaltung genutzt und gezielt eingesetzt werden. Eine konsequente Anwendung des dargestellten Weges erlaubt den Entwurf eines asynchronen Systems mit Hilfe der Standard-Entwurfswerkzeuge für synchrone Schaltungen mit vergleichsweise geringen Änderungen.

Um zu gewährleisten, in einer Entwurfsumgebung bleiben zu können, werden die Architekturen synchroner Schaltungen angepasst und zusätzlich zu entwickelnde Steuerlogik interaktiv generiert. Der Entwurf asynchroner Module wird mit Hilfe einer standardisierten Hardware-

Beschreibungssprache (VHDL[1] [35]) und einer gängigen Simulations- sowie Synthesesoftware bewerkstelligt.

1.3 Übersicht

In Kapitel 1.1 wurden die Vorteile asynchroner Schaltungen dargelegt. Diese sind als zu erreichende Ziele anzusehen. Das heißt man kann nicht erwarten, eine Variante asynchroner Schaltungen beziehungsweise eine Methodik zu finden, die all diese Vorteile aufweist. Vielmehr geht es darum, einzelne Vorteile auszunutzen und überhaupt erst zu erzielen. Wie dies erfolgen kann, wird in dieser Arbeit gezeigt.

Die **Grundlagen** zum Thema **asynchroner Schaltungen** werden in Kapitel 2 erläutert. Hier werden Definitionen zur Asynchronität und Klassifizierungsmerkmale vorgestellt. Des Weiteren werden einzelne Aspekte aufgedeckt, die bei asynchronen Schaltungen zu beachten sind. Der globale Takt selbst wurde als größtes Hindernis synchroner Schaltungen in Kapitel 1.1 dargestellt. Das Taktnetz ist vor allem auch ein großer Energieverbraucher. Die **Entstehung der Verlustleistung** wird in Kapitel 2.6.2 beschrieben.

Für das Aufstellen einer effektiven Methode und einer entsprechenden Software, ist der Überblick über **existierende Werkzeuge** und **Implementierungsvarianten asynchroner Schaltungen** wichtig. Dieser wird in Kapitel 3 gegeben.

Der Haupteil der Arbeit ist in Kapitel 4 beschrieben. Kapitel 4.1 beschreibt die erarbeitete **Methodik** und die Arbeitsschritte der entwickelten Software. Der **Entwurf des Datenpfades** ist sehr eng an den Standardentwurfsablauf synchroner Schaltungen angelehnt. Der **Entwurf des Kontrollpfades** hingegen benötigt zusätzliche Schritte und erfolgt mit Hilfe einer speziellen Synthesesoftware (DGC [45]).

Die für die gewählte Implementierungsvarianten relevanten Schnittstellen werden in Kapitel 4.2 vorgestellt. Es wird eine systematische **Klassifizierung** möglicher (Standard-) **Kontrollstrukturen** und die einzelnen **Implementierungen** dieser Kontrollstrukturen vorgestellt. Zudem werden spezielle - aber häufig vorkommende - Schnittstellenmodule vorgestellt.

In Kapitel 5 werden zwei **Beispiele** für realisierte asynchrone Schaltungen gezeigt. Das erste Beispiel zeigt, dass die gewählte Methode nicht nur für rein datenverarbeitende Schaltungen einsetzbar ist. Bei dem Beispiel handelt es sich um einen fiktiven Controler für eine **digitale Fotokamera**. Das zweite Beispiel ist ein größeres Beispiel aus der Praxis: Ein **Read-Solomon Decoder** mit einer Symbolbreite von 12 Bit, 170 Symbolen pro Frame und der Fähigkeit bis zu 85 Fehler korrigieren zu können. Der Decoder ist für eine Anbindung an einen 40Gbit Daten-

1. **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage

strom ausgelegt, wobei die Parallelisierung der Daten in einem externen Modul zu erfolgen hat. Sowohl die synchrone als auch die asynchrone Realisierung sind eigene Entwürfe.

2 Theoretische Grundlagen des Asynchronen Entwurfs

2.1 Isochronität und Anisochronität

Ein einzelnes binäres Signal besitzt eine Phase und eine Frequenz, die als Phase und Frequenz eines dem Signal zu Grunde liegenden (virtuellen) Taktes definiert (Bild 2.1) wird.

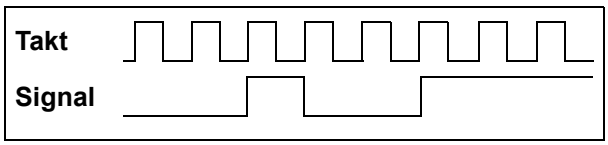


Bild 2.1: Signal und assoziierter Takt

Der mit dem Signal assoziierte (symmetrische) Takt wird mathematisch folgendermaßen definiert:

$$clk(t) = p(((f+\Delta f)t + \phi(t))modulo\,1) \qquad 2.1$$

Hierbei ist clk(t) der von der Zeit t abhängige Wert des Taktes. f ist die *nominale Frequenz*, Δf ein möglicher *Offset* der nominalen Frequenz und ϕ(t) ($0 \le \phi < 1$) die zeitabhängige *Phasenvariation* des Taktsignals. Die Phasenvariation beschreibt das zeitliche Abweichen gegenüber dem Takt. Diese Abweichung wird auch *Jitter* (Zittern) gennant. Für p gilt:

$$p(x) = \begin{cases} 1 & 0 \le x < 0,5 \\ 0 & 0,5 \le x \le 1 \end{cases} \qquad (2.2)$$

Wenn also kein Offset ($\Delta f = 0$) vorhanden ist und die Phasenvariation im Mittel konstant ist ($\frac{d\phi(t)}{dt} = 0$), dann liegt ein symmetrischer Takt vor. Das heißt, die Transitionen (Wertewechsel) des Taktsignals erfolgen über die gesamte Zeit hinweg in äquidistanten Abständen.

Einen solchen Takt bezeichnet man auch als *isochrones* Signal. Allgemein definiert sich ein isochrones Signal folgendermaßen:

Definition 1:

Ein Signal ist isochron, wenn die durchschnittliche nominale Frequenz konstant ist.

Definition 2:

Ein Signal ist anisochron, wenn die durchschnittliche nominale Frequenz nicht konstant ist.

Bei anisochronen Signalen ist zudem die Phasenvariation nicht begrenzt. Das heißt, nur für isochrone Signale gilt:

$$|\phi(t)| \le |\phi_{max}| \qquad (2.3)$$

2.2 Definition der Asynchronität

Asynchron lässt sich am einfachsten als Gegenteil des Synchronen erklären. Beide Begriffe stammen aus der griechischen Sprache, wobei synchron die Eigenschaft beschreibt, dass etwas gleichzeitig, beziehungsweise gleichlaufend erfolgt.

Beim Entwurf digitaler Schaltungen kam man schnell auf die Idee, die Abarbeitung von Daten durch ein einziges globales Steuersignal zu kontrollieren. Dies erfolgt derart, dass an bestimmten Stellen, den sogenannten speichernden Elementen, nur dann Werte aktualisiert werden, wenn dieses Steuersignal (der Takt) es veranlasst. Da dies an allen speichernden Stellen gleichzeitig erfolgt, spricht man hier von synchronen Schaltungen.

Im Speziellen definiert man Synchronität durch folgenden Sachverhalt:

Definition 3: Synchronität
Zwei isochrone Signale sind synchron zueinander, wenn sie den gleichen Offset der nominalen Frequenz besitzen und die Phasendifferenz der beiden Signale gleich Null ist.

Eine synchrone Schaltung besitzt also zum Beispiel einen globalen Takt und (Ausgangs-) Signale, die synchron zu diesem Takt und damit synchron untereinander sind.

Gemäß Definition 3 gilt folgende Definition für die Asynchronität:

Definition 4: Asynchronität
Zwei Signale sind asynchron, wenn sie nicht synchron sind.

Ein anisochrones Signal kann also zum Beispiel nicht synchron zu irgendeinem anderen Signal sein, bis auf den Spezialfall, dass zwei anisochrone Signale identische Transitionen besitzen.

Neben der allgemeinen Asynchronität wird zwischen weiteren Eigenschaften unterschieden. So gibt es Definitionen für die *Mesochronität*, *Plesiochronität* und *Heterochronität*:

Definition 5: Mesochronität
Zwei isochrone Signale sind mesochron zueinander, wenn sie die gleiche durchschnittliche Frequenz $f+\Delta f$ besitzen.

Eine Schaltung ist zum Beispiel dann mesochron, wenn sie Signale besitzt, die zwar vom selben Takt erzeugt werden, aber untereinander unterschiedliche unbekannte Verzögerungen in ihren Pfaden besitzen. Im Speziellen können zwei Taktsignale aus der selben Taktquelle mesochron zueinander sein, wenn unterschiedliche Verzögerungen in den Taktpfaden (*clocktree*) besitzen.

Definition 6:Plesiochronität

Zwei isochrone Signale sind plesiochron zueinander, wenn sie die gleiche nominelle Frequenz besitzen, aber unterschiedliche Offsets zu dieser nominellen Frequenz.

Ein Beispiel für plesiochrone Signale sind zwei Taktsignale mit fast der gleichen Frequenz, welche aber aus unterschiedlichen Taktquellen stammen.

Definition 7: Heterochronität

Zwei isochrone Signale sind heterochron zu einander, wenn sie unterschiedliche nominelle Frequenzen besitzen.

2.3 Asynchronität und Synchronität in der Praxis

Im Allgemeinen Sprachgebrauch versteht man unter asynchronen Schaltungen solche, die mehr oder weniger nur anisochrone Signale enthalten. Als synchrone Systeme hingegen werden entgegen der Definition 3 zumeist alle Schaltungen bezeichnet, die einen oder mehrere globale Takte besitzen.

Dies liegt daran, dass synchrone Systeme nach Definition 3 streng genommen selten vorkommen. Der Entwickler alleine entwirft zwar in seiner Sichtweise durchaus eine synchrone Schaltung, ein synchrones Modul, doch dieses Modul wird zumeist mit anderen Modulen zusammengeschaltet, die nicht immer den gleichen Takt besitzen. Somit wäre das Endergebnis als asynchron, genauer heterochron, zu bezeichnen. Allerdings beschäftigt sich der einzelne Entwickler während der gesamten Entwicklungsphase nur mit den jeweiligen synchronen Modulen, so dass hier die Vereinfachungen und Verfahren vom synchronen Schaltungsentwurf zum Einsatz kommen können. Erst wenn aus mehreren synchronen Modulen ein Gesamtsystem, dass mehrere Takte kennt, entstehen soll, ist unter Umständen auch hier der Sachverhalt der Asynchronität zu berücksichtigen.

Schaltungen sind im Allgemeinen asynchron (heterochron), wenn sie mehrere sogenannte *Taktwelten* umfassen.

Definition 8:

Eine Taktdomäne (clock domain) bezeichnet den Bereich einer Schaltung, der von einem einzigen globalen Signal getaktet wird.

Definition 5 besagt, dass zum Beispiel Schaltungen, bei denen sich Logik im Taktpfad (*gated clock*) befindet, asynchron sind. Hier blendet man die Steuerimpulse des Taktsignals teilweise aus, so dass sich hier nicht immer alle Daten zur gleichen Zeit aktualisieren.

Definition 9:

Ein Takt (clock) bezeichnet man als „gated clock“, wenn sich im Taktpfad zwischen Taktquelle und Taktsenke funktionale Elemente (Logik) befinden.

Ein Spezialfall liegt vor, wenn sich die unterschiedlichen Takte, die in einer Schaltung auftauchen, von einem globalen Taktsignal ableiten, und untereinander ein ganzzahliges Teilungsverhältnis sowie eine starre Phasenlage besitzen. Man entwickelt hier jede Taktwelt alleine wieder mit synchronen Verfahren. Danach schaltet man die Module zusammen, wobei durch ein ganzzahliges Taktteilerverhältnis eine sichere Übernahme der Daten gewährleistet ist. Aus der Praxis betrachtet sind also alle Schaltungen asynchron,

- die mehrere Takte besitzen, welche nicht aus einem globalen Takt abgeleitet sind und untereinander kein ganzzahliges Teilerverhältnis sowie keine starre Phasenlage besitzen
- die Logik im Taktpfad enthalten
- die lokal erzeugte Takte enthalten (selbstgetaktet) und
- die keinen Takt besitzen

Asynchrone Schaltungen der ersten beiden Varianten kann man zumeist noch durch geringfügige Änderung und Erweiterung der synchronen Entwurfsmethoden entwickeln. Bei den *gated clocks* muss das Zeitverhalten der Logik im Taktpfad für den Entwurf des Taktverteilersystems mit berücksichtigt werden; dieses Zeitverhalten muss sehr stabil sein. Verschiedene Entwicklungswerkzeuge unterstützen das Benutzen von gated clocks.

Auch Schaltungen mit mehreren Takten, die nicht von einer Taktquelle abgeleitet sind, oder untereinander kein ganzzahliges Teilerverhältnis besitzen, kann man unter Umständen mit synchronen Entwurfsmethoden entwickeln. Allerdings nehmen dabei die Probleme beim Entwurf zu und es sind oft sehr viele Iterationsschritte nötig, um letztendlich für das korrekte Funktionieren einer solchen Schaltung Gewähr leisten zu können. Allgemein kann man sagen, je mehr Taktwelten eine Schaltung besitzt, desto schwieriger ist es, mit synchronen Entwurfsmethoden eine funktionierende Schaltung zu entwerfen.

Die letzen beiden Varianten von asynchronen Schaltungen lassen sich mit den gängigen synchronen Entwurfsmethoden nicht mehr entwickeln. Somit muss man hier veränderte, angepasste beziehungsweise ganz neue Entwurfsverfahren verwenden.

2.4 Unterscheidungskriterien asynchroner Implementierungen

Asynchrone Schaltungen kann man anhand der verwendeten Verzögerungsmodelle unterscheiden. Hierbei unterscheidet man nicht nur allgemein zwischen begrenzten (bekannten) und unbegrenzten (unbekannten) Verzögerungen, sondern auch zwischen dem örtlichen Auftreten in Gattern beziehungsweise der Verdrahtung. Entsprechend den verwendeten Verzögerungsmodellen haben sich auch unterschiedliche Kodierungsvarianten bei asynchronen Schaltungen

durchgesetzt. Hauptsächlich unterscheidet man zwischen den beiden Varianten „Dual-Rail“-Kodierung und „Bundled-Data“-Kodierung.

Zuletzt kann man asynchrone Schaltungen auch nach den verwendeten Datenübergabeprotokollen (Handshake) unterscheiden. Man ordnet die verschiedenen Varianten den zwei Gruppen des Zwei-Phasen und des Vier-Phasen Protokolls zu.

2.4.1 Verzögerungsmodelle

Verzögerungsmodelle lassen sich prinzipiell nach drei Kriterien unterscheiden. Bei dem zeitlichen Verhalten der Verzögerungsmodelle betrachtet man die Begrenztheit und Bekanntheit der Verzögerung. Bei dem Ort der Verzögerung unterscheidet man zwischen der reinen Gatterverzögerung, der reinen Leitungsverzögerung oder einer gleichzeitigen Betrachtung beider Verzögerungen. Schließlich unterscheidet man noch, ob die Verzögerungsglieder jeden Signalimpuls übertragen, oder kurze Signalimpulse herausfiltern und ihren alten Wert behalten (speichern).

2.4.1.1 Art des Verzögerungsmodells

Bei den Verzögerungsmodellen geht man im Prinzip von zwei Möglichkeiten aus: Die Verzögerungen sind bekannt und können damit in der Schaltung explizit berücksichtigt und auch ausgenutzt werden. Oder die Verzögerungen sind unbekannt und es müssen geeignete Maßnahmen ergriffen werden, damit dies nicht zu einem Fehlverhalten der Schaltung führt.

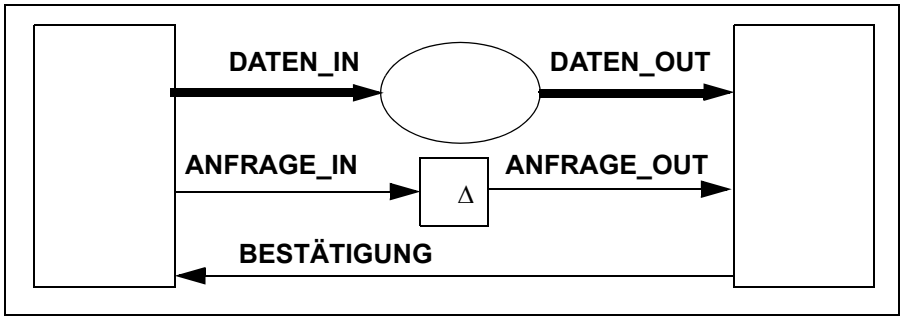


Bild 2.2: Datentransfer im bounded delay Modell

Begrenztes Verzögerungsmodell (bounded delay):

Bei dem begrenzten Verzögerungsmodell geht man davon aus, dass alle in der betrachteten Schaltung auftretenden Verzögerungen endlich und bekannt sind. Eine Folgerung daraus ist, dass man gezielt die Laufzeit einzelner Signale zum Beispiel durch Verzögerungselemente verlängern kann, um so das korrekte Verhalten der Schaltung sicherzustellen.

Bild 2.2 zeigt ein kleines Beispiel für den Einsatz des begrenzten Verzögerungsmodells. Das Signal „Anfrage“ soll das Ankommen eines neuen Datums signalisieren. Dies darf erst erfolgen,

wenn das Datum am Ausgang der Logikwolke stabil anliegt. Da aufgrund des begrenzten Verzögerungsmodells alle Laufzeiten bekannt sind, ist in Bild 2.2 ein Verzögerungselement eingefügt worden, welches das Steuersignal um mindestens den Wert verzögert, den die langsamste Signalausbreitung durch die Logik vorgibt.

Es sind hier auch gleich die Probleme zu sehen. Zum einen bedingt das begrenzte Verzögerungsmodells, dass man alle Verzögerungen kennt. Dies impliziert eine Extraktion aller relevanten Verzögerungszeiten aus der synthetisierten Schaltung. Dies muss man bei jedem Synthesedurchgang und auch nach Fertigstellung des Layouts einer Schaltung durchführen. Somit entsteht ein nicht unerheblicher Entwurfsaufwand.

Zum anderen handelt man sich hier eine lokale „worst case“ Performanz ein, da der längste Datenpfad den Wert der zusätzlichen Verzögerung in Bild 2.2 vorgibt. Dadurch verliert man etwas von dem unter Kapitel 2.6 angesprochenen Vorteil einer „average case“ Performanz.

Ein weiterer Punkt ist der Test auf Verzögerungsfehler. Dies ist keine triviale Aufgabe, da diese von vielen Parametern abhängen und die Kontrolle und Beobachtbarkeit schwer zu gewährleisten ist.

Unbegrenztes Verzögerungsmodell (unbounded delay)

Im Gegensatz zu den begrenzten Verzögerungsmodellen geht man bei unbegrenzten Verzögerungsmodellen von unbekannten aber endlichen Verzögerungen aus. Macht man hierbei keine Unterscheidung bezüglich des Ortes (siehe Kapitel 2.4.1.2), handelt es sich um so genannte **verzögerungsunabhängige** (**delay insensitive**) Schaltungen. Das heißt der Entwickler oder das Entwurfswerkzeug muss davon ausgehen, dass die Laufzeiten durch einen kombinatorischen Pfad nicht bekannt sind. Folgerichtig muss im Entwurf ein zusätzlicher Mechanismus das Funktionieren der Schaltung sicherstellen.

Diesen Mechanismus bezeichnet man allgemein als „Handshaking“. Der als Sender bezeichnete Datengenerator signalisiert, wann ein neues Datum zum Abholen bereitsteht. Der Empfänger wiederum signalisiert das erfolgreiche Abholen dieses Datums. Siehe hierzu auch 2.4.4 Datenübergabeprotokolle.

Ein Problem ist hier das Generieren der entsprechenden Handshake-Signale. Eine Möglichkeit hierfür ist eine sogenannte Komplettierungsdetektion (completion detection). Eine zusätzliche Schaltung erkennt ein neu anliegendes Datum und signalisiert dies durch Setzen eines Anfragesignals. Siehe hierzu auch 2.4.2 Datenkodierungen.

Eine andere Möglichkeit ist, im Entwurf selbst sicherzustellen, dass zum Beispiel der Sender das Anfragesignal immer erst nach dem Anliegen eines neuen Datums setzt. Da man aber von unbegrenzten Verzögerungen ausgeht, kann man dies nur durch zusätzliche Logik erwirken.

Man erkennt schnell, dass Modelle mit unbegrenzten Verzögerungen zu robusteren Schaltungen führen, da der Handshake-Mechanismus Parameterschwankungen und damit sich ändernde Verzögerungen ohne Probleme abfängt. Nachteilig sind aber der zusätzliche Entwurfsaufwand und Hardwareaufwand, der sich nicht nur in der zusätzlich benötigten Fläche, sondern auch im Verbrauch (mehr Gatter, die zu schalten sind) und in der Performanz (Laufzeit des Datums + Laufzeit der Komplettierungsdetektion) bemerkbar macht.

2.4.1.2 Ort der Verzögerung

Prinzipiell gibt es die Möglichkeit, bei den auftretenden Verzögerungen zwischen den Gatter- und den Leitungsverzögerungen zu unterscheiden. Beim synchronen Entwurf betrachtete man lange nur die Gatterverzögerungen. Dies lag daran, dass diese gegenüber den Leitungsverzögerungen erheblich größer waren. Heutzutage berücksichtigt man im synchronen, als auch im asynchronen Entwurf beide Anteile der Verzögerung.

Im asynchronen Entwurf kann man je nach verwendetem Verzögerungsmodell weiterhin auf das Modell mit reinen Gatterverzögerungen zurückgreifen oder nicht. Nimmt man als Ort der auftretenden Verzögerungen nur die Gatter an, so sind die Leitungsverzögerungen geeignet in die Gatterverzögerungen einzurechnen. Das folgende Bild soll zeigen, dass dies allgemein aber nur bei dem Modell der begrenzten Verzögerungen erlaubt ist.

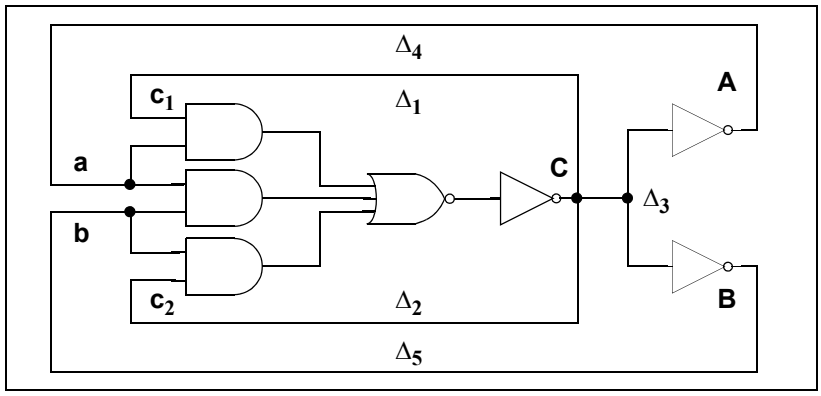


Bild 2.3: Verzögerungen in einer Schaltung (C-Element)

Die Auswirkungen der unterschiedlichen Verzögerungsmodelle stellt hier das Beispiel eines rückgekoppelten Muller C-Elements dar. Die logische Darstellung zeigt, dass das eigentliche Ausgangssignal C ähnlich einem Takt zwischen den Werten ‘0’ und ‘1’ alterniert. Dabei müssen für einen Wertewechsel beide Eingänge a und b jeweils den gleichen Wert einnehmen.

Verwendet man ein Modell mit **unbegrenzten Gatterverzögerungen** spricht man auch von **geschwindigkeitsunabhängigen** (**speed independent**) Schaltungen. Für diesen Fall erfüllt die Schaltung in Bild 2.3 anscheinend die Spezifikation. Der Ausgang C ist sofort als Rückkopplung durch c_1 und c_2 wirksam und das korrekte Verhalten ist dadurch gewährleistet.

Verwendet man ein Modell mit **unbegrenzten Leitungsverzögerungen**, dann ist dies nicht mehr gewährleistet. Ist zum Beispiel die Verzögerung Δ_1 von C nach c_1 größer als ($\Delta_3 + \Delta_4$) von C nach A, dann kann sich ein Wechsel von A am Ausgang C bemerkbar machen, ohne dass der zweite Eingang B wie verlangt, den gleichen Wert von A einnimmt. Dies liegt daran, dass die Rückkopplung von C in c_1 noch nicht wirksam ist, während A schon seinen Wert wechselt.

Bei der Verwendung von begrenzten Verzögerungsmodellen hängt die Korrektheit von den tatsächlich vorliegenden Verzögerungen ab. Dementsprechend fügt man hier dann zusätzliche Verzögerungsglieder für die Justierung der Verzögerungen Δ_1 und Δ_2 ein.

Bild 2.4: Isochronische Verzweigung im Vergleich zu laufzeitunabhängiger Verzweigung

Im Spezialfall, dass unbegrenzte Leitungsverzögerungen angenommen werden, aber nur unter der Bedingung der *isochronic forks*, liegen **quasi-verzögerungsunabhängige** (**quasi delay insensitive**) Schaltungen vor. Isochronic Forks stellen ein spezielles Modell der Gabelung von Leitungen dar, wobei der Unterschied zwischen den Verzögerungen der gegabelten Leitung vernachlässigbar ist, das heißt, ähnlich wie in Taktnetzen sind die Verzögerungen im gesamten Netz gleich.

2.4.1.3 Speichereigenschaften

Verzögerungen entstehen, wenn sich eine Wertänderung durch ein Gatter oder über eine Leitung ausbreiten soll. Je nach Eigenschaft des Mediums über das sich die Wertänderung ausbreiten soll, werden nur kurz anliegende Wertänderungen beziehungsweise Pulse, die man auch Spikes nennt, unverändert übertragen oder herausgefiltert. Die beiden Verzögerungsarten, die man an dieser Stelle unterscheidet, sind die Trägheits- und die Transportverzögerung.

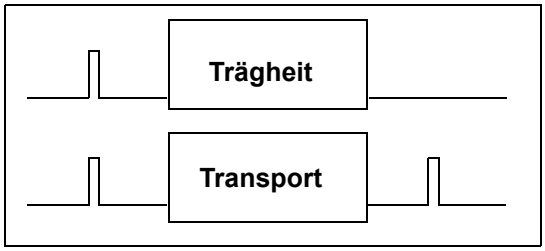


Bild 2.5: Trägheits- (inertial) und Transportverzögerung

Trägheit

Ein Trägheitsverzögerungsglied überträgt nur Signalimpulse, die eine Mindestbreite besitzen. In Bild 2.5 ist dieser Sachverhalt auf der linken Seite dargestellt. Impulse, die nicht lange genug anliegen, sind nicht in der Lage, den Ausgang auf den neuen Wert zu treiben; somit bleibt der Ausgang auf seinem alten Wert. Allgemein kann man sagen, dass Gatter eine Trägheitsverzögerung besitzen. Allerdings ist bei einem Grundgatter die Mindestbreite für Pulse sehr klein, so dass hier selten eine effektive Filterung von Spikes erfolgt.

Transport

Eine Transportverzögerung überträgt jede Wertänderung, unabhängig von der Zeit, die diese Änderung anliegt. Auf der rechten Seite in Bild 2.5 ist dies dargestellt. Im Allgemeinen betrachtet man Leitungsverzögerungen als transport delays.

2.4.2 Datenkodierungen

Asynchrone Automaten sind ereignisgesteuert. Während in einer synchronen Schaltung eher die Signalpegel (zum Zeitpunkt der auftretenden aktiven Taktflanke) zu betrachten sind, führt in einer asynchronen Schaltung allgemein jede Änderung eines Signals zu einer Reaktion der Schaltungen. Um einen gesicherten Datentransfer durchführen zu können, muss man in einer asynchronen Schaltung zusätzliche Kommunikationsinformationen implementieren. Wie dies prinzipiell aussieht, hängt nicht zuletzt von der Kodierung der Daten ab. Generell unterscheidet man zwischen den gebündelten Daten, also eigentlich unkodierten Daten, und den kodierten Daten. Bild 2.6 zeigt die beiden Varianten, wobei für das Beispiel der kodierten Daten die häufig verwendete Variante „dual rail“ gewählt worden ist.

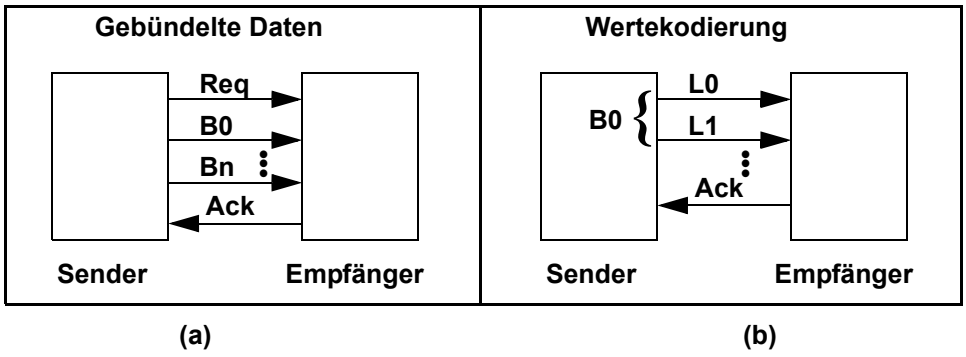


Bild 2.6: Datentransfer mit gebündelten Daten (a) und mit „dual rail“ kodierten Daten (b)

2.4.2.1 Gebündelte Daten

Bei gebündelten Daten repräsentiert eine Leitung (Draht, wire) genau ein Bit. In Bild 2.6 sind dies die Leitungen B_0 bis B_n für n+1 Bits. Um ein neues Datum anzuzeigen, benötigt man ein zusätzliches Steuersignal (Req). Die Verzögerung der Steuersignalleitung muss auf jeden

Fall größer als die längste Verzögerung in einer der Bitleitungen sein. Nur dann kann der Empfänger die ankommenden Daten sicher übernehmen. Die erfolgreiche Übernahme eines neuen Datums signalisiert der Empfänger über das Acknowledge Signal (Ack).

2.4.2.2 Kodierte Werte

Bei dem Datentransfer mittels kodierter Daten spricht man auch vom „transitions signalling“. Es werden also Übergänge oder allgemeiner Ereignisse signalisiert. Um diese detektieren zu können, sind dann aber mehr Leitungen als Datenbits notwendig, da zu den gültigen Daten noch ungültige (auch “neutrale”) Werte hinzukommen.

Bei kodierten Werten kann der Empfänger feststellen, wann ein neues Datum anliegt. Voraussetzung hierfür ist, dass zwischen gültigen Daten ein neutrales Datum anliegt. Ansonsten kann die Empfängerseite nicht zwischen zwei aufeinanderfolgenden gleichen Datensätzen unterscheiden. Auch bei kodierten Werten signalisiert das Setzen des Ack Signals den Empfang eines neuen Datums.

Allgemein erhält bei der Wertekodierung jeder zu kodierende Wert ein eigenes Kodewort. Oft ist das ein „k-aus-n“ Kodewort, wobei k die jeweilige Anzahl der Einsen in einem Kodewort und n die gesamte Wortbreite des Kodes ist. Folgende Fälle nutzt man häufiger:

„k-aus-n“ Kodierung

Bei der allgemeinen „k-aus-n“ Kodierung muss man für jeden zu kodierenden Wert ein eindeutiges Kodewort wählen. Hierbei bezeichnet k die Anzahl der ‘1’ Werte in dem n Bit breiten Kodevektor (L=n). Die Bitbreite und somit die Anzahl der benötigten Leitungen des Kodewortes berechnet sich einerseits nach der Anzahl m der Werte, die darzustellen sind, sowie der Anzahl k der ‘1’ Werte pro Kodewort.

$$\frac{L!}{k!(L-k)!} \geq m \qquad (2.4)$$

Gleichung *(2.4)* zeigt wie die drei Größen zusammenspielen. Beispiel: Für eine Wertemenge der Mächtigkeit von m= 12 und k=2 Einsen pro Kodewort benötigt man 6 Leitungen. Mit diesen 6 Leitungen könnte man bei k=2 15 Werte darstellen. Für den Spezialfall k=1 erhält man wiederum die 1-aus-n Kodierung. Für k=3 benötigt man ebenfalls 6 Leitungen, könnte dann aber 20 Werte darstellen. Zumeist ist es aber vorteilhaft, möglichst wenige ungenutzte Kodewörter zu haben.

Man kann für die zu übertragenden Daten eine optimale „k-aus-n“ Kodierung finden. Allerdings unterscheiden sich die zusätzlichen Detektierungsschaltungen, die ein gültiges Kodewort erkennen sollen, von Fall zu Fall. Es ist deswegen allgemein einfacher, einen der beiden Spezialfälle der Dual-Rail oder 1-aus-n Kodierung zu wählen.

Dual Rail („1-aus-2“) Kodierung

Bild 2.6 zeigt, dass bei der Dual Rail Kodierung jedes einzelne Bit kodiert wird. Sind beide Leitungen auf ‘0’, so liegt kein gültiger Wert an. Liegt zum Beispiel L_0 auf ‘1’ und L_1 auf ‘0’ so hat das betreffende Bit den Wert ‘1’ und umgekehrt den Wert ‘0’. Der Fall, dass beide Leitungen auf ‘1’ liegen ist nicht erlaubt und stellt einen ungültigen Wert dar.

Ist ein ganzer Datenvektor zu übertragen, so muss der Empfänger feststellen, ob für jedes Bit ein gültiger Wert anliegt. Nur dann darf er den aktuellen Wert als neues Datum übernehmen. Zwischen den gültigen Daten müssen alle Leitungen auf ‘0’ sein, um dem Empfänger eindeutig mitzuteilen, dass demnächst ein neues Datum ankommt.

Bei einer Dual Rail Kodierung benötigt man für jedes Bit zwei Leitungen. Somit ergibt sich für die Anzahl der benötigten Leitungen L für m Datenwörter:

$$L = 2 \cdot ENTIER[ld(m)] \qquad (2.5)$$

Der ENTIER Operator erzeugt dabei aus einer reellen Zahl die nächst größere natürliche Zahl. Beispiel: Für eine Wertemenge der Mächtigkeit von m=12 benötigt man 4 Bits ($2^4 = 16 \geq 12$). Nach Gleichung *(2.5)* sind also 8 Leitungen erforderlich, um diese 12 Werte in einer Dual Rail Kodierung darstellen zu können.

One-Hot („1-aus-n“) Kodierung

Dies ist eine Variante der Wertekodierung, die man auch bei Automaten in synchronen Schaltungen verwendet. Man geht von einem zu übertragendem Datenvektor (L=n) aus. Für jeden möglichen Wert dieses Vektors verwendet man eine eigene Leitung. Liegt nur eine Leitung auf ‘1’ so liegt der entsprechende Wert an, den diese Leitung repräsentiert. Liegen alle Leitungen auf ‘0’, wird damit angezeigt, dass demnächst ein neues Datum kommt. Bei asynchronen Schaltungen ist das gleichzeitige Anliegen einer ‘1’ auf mehreren Leitungen verboten, da dies einen ungültigen Wert darstellt.

$$L = m \qquad (2.6)$$

Gleichung *(2.6)* zeigt, dass man bei der 1-aus-n Kodierung gleichviel Leitungen wie zu kodierende Werte benötigt. Es zeigt sich, dass bei größeren Wertemengen die Anzahl der benötigten Leitungen erheblich stärker zunimmt als dies für die Dual Rail Kodierung der Fall ist.

2.4.3 Komplettierungsdetektierung

Ein Vorteil von kodierten Werten ist, dass die Empfängerseite feststellen kann, ob ein neues Datum anliegt. Im Gegensatz zu Bild 2.2, bei dem der Sender dem Empfänger ein neues Datum ankündigt, überwacht hier der Empfänger die Datenleitungen und übernimmt selbständig das anliegende Datum, sobald er ein gültiges Kodewort erkannt hat (siehe Bild 2.7). Zwei Voraus-

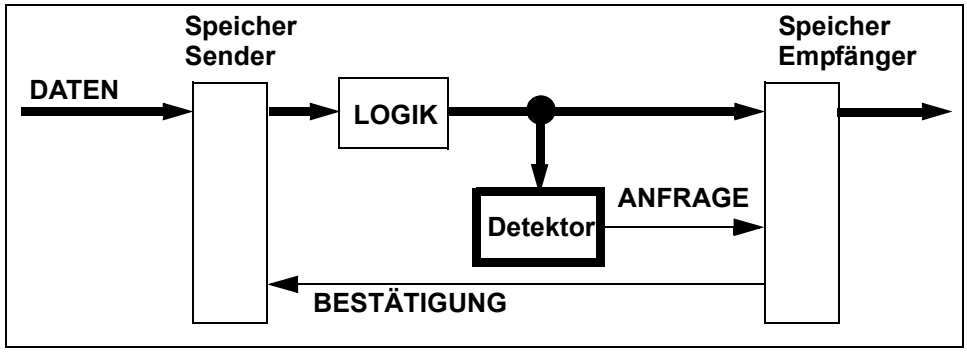


Bild 2.7: Datentransfer mit Komplettierungsdetektion

setzungen müssen erfüllt sein. Zum einen muss jedes Kodewort eindeutig und damit detektierbar sein; zum anderen muss erkennbar sein, wann ein neues Datum gesendet wird, selbst wenn dieses mit dem vorherigen Datum übereinstimmt. Um letztere Voraussetzung zu erfüllen, legt man allgemein zwischen den gültigen Kodewörtern ein vereinbartes neutrales Wort („Nullwort“) an. Mit diesem soll der Sender die vollständige Rücknahme eines Datums angezeigen.

Bitweise Komplettierungsdetektion

In Bild 2.8 sind zwei Schaltungsvarianten gezeigt, die es erlauben den eingehenden Wert eines einzelnen Bits in Dual Rail Kodierung zu detektieren. Das Komplettierungssignal dient dann als Bestätigung (ACK) an die sendende Seite, dass das Datum stabil anliegt.

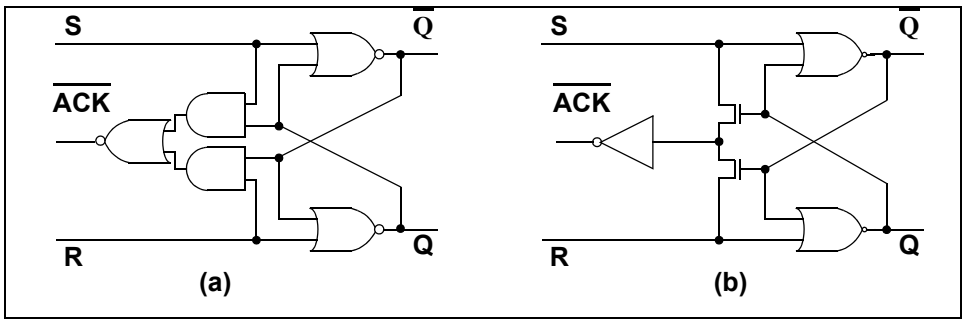


Bild 2.8: Bitweise Komplettierungsdetektion [65] [99]

In Variante (a) befindet sich die Bestätigungsleitung ($\overline{\text{Ack}}$) im Ruhezustand (S=’0’; R=’0’) auf ’1’. Geht einer der beiden Eingänge auf ’1’, dann geht das entsprechende UND-Gatter auf ’1’ (der passende Ausgang muss hierzu ebenfalls auf ’1’ liegen). Diese wiederum zieht das Bestätigungssignal auf ’0’, welches die Übernahme des Datums anzeigt. Nun kann die sendende Seite die Eingänge wieder in den Ruhezustand bringen.

Bei der zweiten Variante (b) befindet sich das Bestätigungssignal ($\overline{\text{Ack}}$) im Ruhezustand auch auf ’1’, da es sich über einen der beiden Transistoren entladen kann. Geht eine der beiden Eingänge auf ’1’, geht der entsprechende Ausgang auf ‘1’, was den passenden Transistor öffnet. Über diesen lädt sich das Anfragesignal auf ‘1’, was eine Anfrage an den Empfänger darstellt.

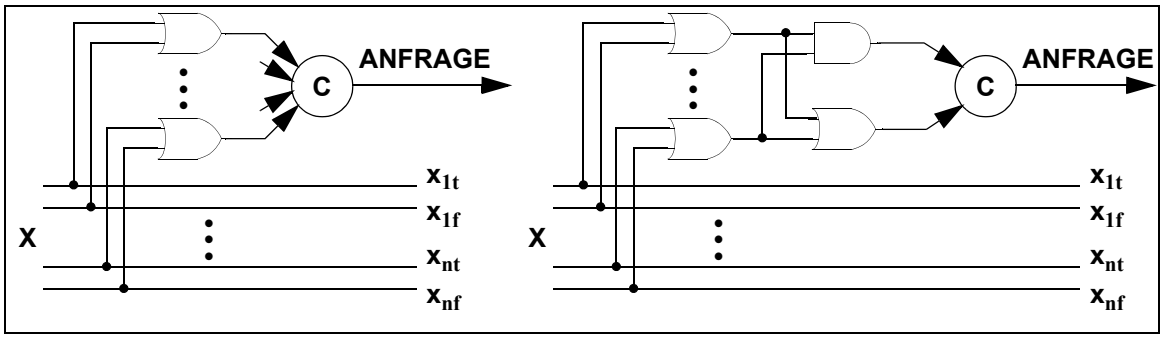


Bild 2.9: Vektorweise Komplettierungsdetektion [5] [101]

Vektorenweise Komplettierungsdetektion

Die oben vorgestellte bitweise Komplettierungsdetektion kann man für einen Vektor benutzen, indem man die Bestätigungssignale über ein ODER beziehungsweise ein UND Gatter zusammenfasst. Bild 2.9 zeigt eine andere Variante, einen neuen Wert eines dual-rail kodierten Vektors zu detektieren. Man benutzt an dieser Stelle ein C-Element (siehe Bild 3.15). Dieses C-Element übernimmt nur dann einen Wert, wenn alle Eingänge den gleichen Wert besitzen. Somit geht das Anfragesignal erst dann auf '1', wenn alle Leitungspaare einen Wert liefern. Letzteres stellt man mittels der ODER Gatter fest. Das Anfragesignal geht wiederum erst dann auf '0' zurück, wenn wieder alle Leitungspaare im neutralen Zustand sind; d.h. alle Leitungen müssen auf '0' sein. Da C-Elemente mit einer höheren Zahl von Eingängen schwerer zu realisieren sind, greift man oftmals auf die rechts in Bild 2.9 dargestellte Variante zurück. Die UND und ODER Gatter fassen wiederum die bitweisen Detektionen zusammen.

2.4.4 Datenübergabeprotokolle

Im vorigen Abschnitt wurde die Möglichkeit der Datenübergabe mit gebündelten Daten gezeigt. Damit die Übergabe geregelt stattfindet, wird ein vorher festgelegtes Übergabeprotokoll (Handshake) abgearbeitet. Der Sender signalisiert dabei mit dem Req Signal in Bild 2.6, dass er ein neues Datum sendet und der Empfänger bestätigt den Empfang mit dem Ack Signal. Bei kodierten Werten ist die Anfrage implizit mit den Daten verknüpft. Erkennt der Empfänger ein gültiges Kodewort, was einer Anfrage bei den gebündelten Daten entspricht, so bestätigt er den Empfang.

Man unterscheidet zwischen folgenden zwei Varianten des Handshakes:

2.4.4.1 Zweiphasenprotokoll

Der Sender signalisiert ein gültiges neues Datum mit einer Wertänderung des Anfragesignals Req (jeweils Phase I). Der Empfänger quittiert den erfolgreichen Empfang des Datums mit einer entsprechenden Wertänderung des Bestätigungssignals Ack (jeweils Phase II). Wie Bild

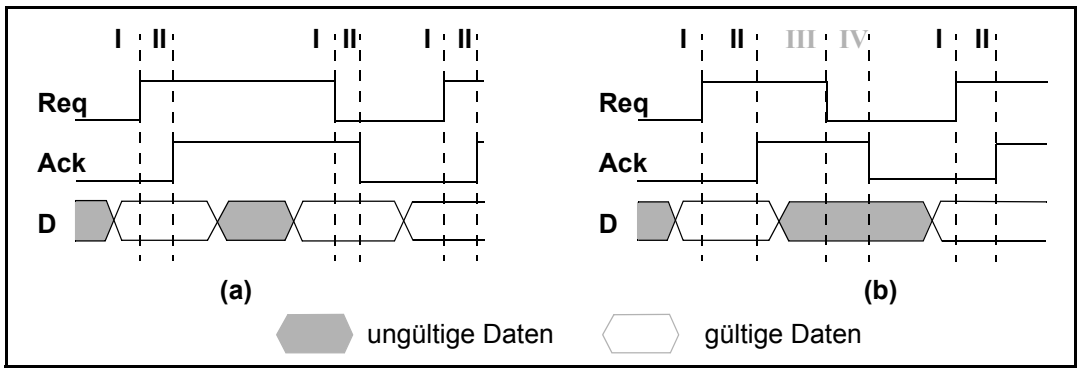


Bild 2.10: Datenübertragungsprotokolle mit (a) 2 Phasen und (b) 4 Phasen

2.10 (a) zeigt, sind steigende und fallende Flanken der Kontrollsignale äquivalent. Somit besteht ein kompletter Zyklus zur Datenübergabe aus den zwei aufeinanderfolgenden Transitionen der Kontrollsignale von Sender und Empfänger.

2.4.4.2 Vierphasenprotokoll

Bei dem Vierphasenprotokoll benutzt man nur eine Art von Flanken, um die Daten auszutauschen. Im Allgemeinen benutzt man die steigenden Flanken, wie in Bild 2.10 (b) dargestellt. Ein gültiges Datum zeigt der Sender mit dem Setzen des Req Signals auf ‘1’ (Phase I) und der Empfänger bestätigt den Empfang durch das Setzen des Ack Signals (Phase II). In Phase III und IV setzen die Partner dann die beiden Kontrollsignale nacheinander wieder auf ‘0’ zurück.

2.5 Graphische Spezifikationsmethoden

Es gibt eine Vielzahl von Möglichkeiten asynchrone Schaltungen zu spezifizieren. Eine Gruppe von Möglichkeiten sind die unterschiedlichen Hochsprachen aus dem synchronen Entwurf (VHDL, Verilog) und aus dem asynchronen Entwurf (Tangram, Occam, LARD, Balsa, und andere). Interessant sind aber auch die graphischen Möglichkeiten, da es oftmals einfacher ist, die Zusammenhänge in einer graphischen Präsentation zu erkennen.

Im Folgenden werden drei sehr interessante und für diese Arbeit wichtige Formen der graphischen Spezifikation von asynchronen Automaten vorgestellt. Dies sind die von den Petri Netzen abgeleiteten Signal Transition Graphen (STG), die Burst Mode Graphen (BMS) und die Extended Burst Mode Graphen (XBM).

2.5.1 Signal Transition Graphen

Vor etwa 10 Jahren führten Rosenblum [80] und Chu [19] unabhängig voneinander den Signalübergangsgraphen ein, auch Signalflankengraph oder - in englischer Sprache - Signal Transition Graph genannt (STG). Signalübergangsgraphen basieren wie die I-Nets [64][79] auf

Petri-Netzen [73], [74] und [66]. Die Transitionen der Petri-Netze werden zu diesem Zweck mit Signalnamen (kleine Buchstaben) und der Art des Wertewechsels gekennzeichnet. a+ zum Beispiel kennzeichnet den Wechsel 0→1 des Signals a. Dagegen besagt a*, dass an a entweder eine positive (0→1) oder eine negative (1→0) Transition erfolgen kann. Ist ein Signalname unterstrichen, so handelt es sich dabei um Eingangssignale.

Wie Petri-Netze auch, besitzen STGs mehrere Tokens. Dies ist eine Besonderheit, da zum Beispiel synchrone und asynchrone (Zustands-) Automaten nur ein „Token“ besitzen, das heißt, sich nur in genau einem Zustand befinden können; anders ausgedrückt, kann nur ein Ort im Graphen eingenommen werden.

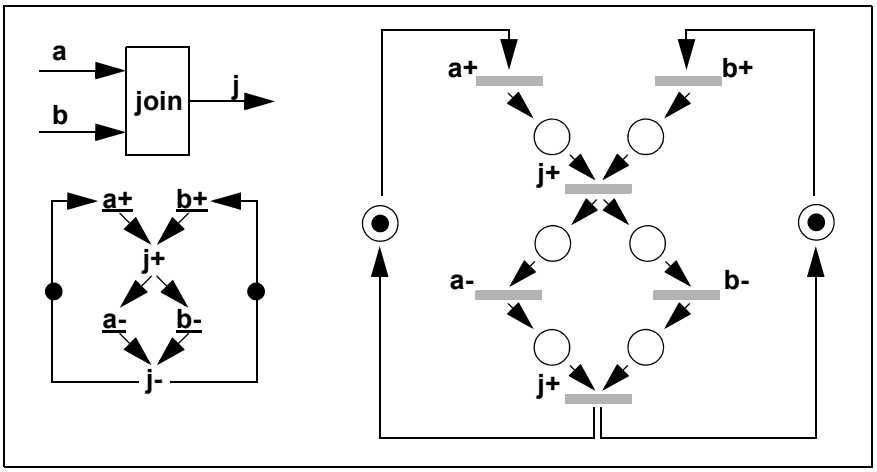


Bild 2.11: STG (links) und Petri-Netz (rechts) eines Join Elementes

Im Bild 2.11 ist ein Beispiel für einen Signalübergangsgraphen und das zugehörige Petri-Netz dargestellt. Es handelt sich hierbei um die Beschreibungen eines Join-Elements, das heißt, nur wenn beide Eingänge mit dem gleichen Wertewechsel reagiert haben, ändert sich auch der Ausgang.

Findet in dem Petri-Netz eine Transition statt („feuert“ sie also), dann ist dies gleichbedeutend mit dem Wertewechsel des entsprechenden Signals. Im Signalübergangsgraphen werden die Plätze eines Petri-Netzes als gerichtete Kanten dargestellt. Die eingezeichneten Marker können also nur in die entsprechende Richtung wandern. Wie auch bei Petri-Netzen kann ein Signalwechsel nur dann stattfinden, wenn alle Eingangsplätze (Pfeile) einen Marker besitzen. So kann das Signal j nur seinen Wert ändern, wenn beide Eingänge des Join-Elements ihren Wert gewechselt haben.

Die Signalübergangsgraphen werden im Gegensatz zu den I-Nets für die automatische Synthese optimiert und es werden nur bestimmte Typen von Konstrukten eines Petri-Netzes erlaubt. Je nach Einschränkung der erlaubten Konstrukte lassen sich die Petri-Netze in bestimmte Klassen gruppieren. Diese sind in dem Übersichtsartikel von Tadao Murata [66] nachzulesen. Der Typ des Petri-Netzes in Bild 2.11 zum Beispiel ist ein Marked Graph (MG), das heißt, jeder

Platz besitzt höchstens eine Eingangstransition und nicht mehr als eine Ausgangstransition. Ist dies nicht der Fall, handelt es sich um sogenannte Free Choice Nets (FCN).

2.5.2 Burst Mode Graphen

Burst Mode Graphen sind spezielle Signalübergangsgraphen. Entgegen der Einschränkung des *Fundamentalmodus* (siehe Kapitel 3.1.1.1), sind beim Burst Mode mehrere Signalwechsel "gleichzeitig" erlaubt. Vorausgesetzt wird allerdings, dass die zu einem Burst gehörenden Signalwechsel innerhalb eines festgelegten Zeitraums erfolgen. Dies wird kurz am folgenden Beispiel gezeigt:

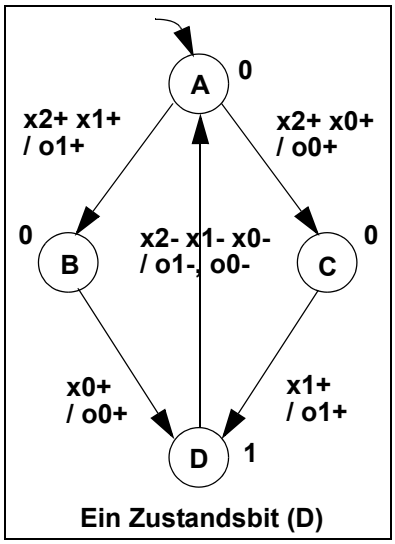


Bild 2.12: Beispiel eines Burst Mode Graphen

Wie beim ursprünglichen STG werden Wertewechsel in einem Burst Mode Graphen mit einem "-" oder einem "+" gekennzeichnet. In Bild 2.12 führen zum Beispiel aus dem Zustand A zwei unterschiedliche Bursts heraus. Es wird in den Zustand B gewechselt, wenn x2+ und x1+ auftreten, oder in den Zustand C, wenn x2+ und x0+ auftreten. Man erkennt, dass x2 in beiden Bursts auftritt, die anderen Eingänge aber jeweils nur in einem der Bursts. Damit der jeweilige Burst von der Schaltung korrekt erkannt und verarbeitet wird, darf sich zum Beispiel beim Wechsel von A nach B der zunächst unbeteiligte Eingang x0 erst ändern, wenn der Zustand B fest eingenommen worden ist.

Außer dieser Bedingung sind für eine korrekte Implementierung noch andere Bedingungen einzuhalten. Dies wird an anderer Stelle genauer beschrieben (siehe Kapitel 3.1.1.3 und Kapitel 3.2.3.3).

2.5.3 Extended Burst Mode Graphen

Beim Extended Burst Mode (XBM) sind nicht nur Bursts von Signalwechseln erlaubt, sondern es können auch Signalpegel berücksichtigt werden. Durch eckige Klammern wird dabei eine Abfrage auf einen Pegelwert dargestellt. Ein '*' besagt, dass das Verhalten oder der Pegelwert eines Signals bei dem aktuellen Übergang keine Auswirkungen hat. Somit erweitert sich die Anwendungsmöglichkeit der Burst Mode Schaltungen. Letztendlich kann man auch synchrone Automaten als XBMs modellieren.

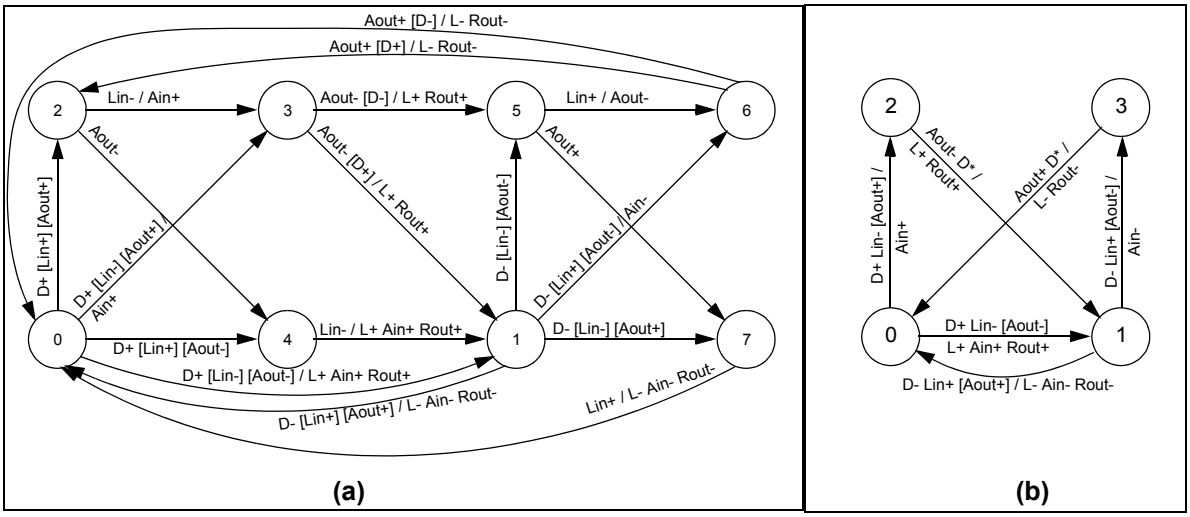


Bild 2.13: Extended Burst Mode Automaten mit Pegelabfrage (a,b) und expliziten Ignorieren eines Signals (b).

Bild 2.13 zeigt eine Spezifikation nach den Extended Burst Mode Regeln. Einzelne Zustandsübergänge, bei denen die entsprechenden Signalwechsel zu keiner Ausgangspegeländerung führen, können samt dem „überflüssigen" Zustand entfernt und in andere Übergänge als Pegelwertabfrage aufgenommen werden. Bild 2.13 (a) zeigt die Spezifikation eines Kontrollers bei denen Pegelwerte von Signalen abgeprüft werden. Der Anzahl der überprüften Pegelwerte pro Übergang ist prinzipiell keine Grenze gesetzt. Allerdings ist generell zu erwarten, dass sehr vielen Pegelwertabfragen pro Übergang zu entsprechend aufwendigen Implementierungen und somit eventuell zu keiner Verbesserung gegenüber der Burst Mode Spezifikation führen. Die für die Synthese verwendete Software und die Leistungsfähigkeit der darin verwendeten Optimierungsalgorithmen spielen dabei eine große Rolle.

Bild 2.13 (b) zeigt die Verwendung von „directed dont cares". Das heißt die durch ein '*' gekennzeichneten Signale werden bei dem aktuellen Übergang ignoriert, da sie hier keinen Einfluss auf die Ausgangsbelegung haben. Bedingungen hierbei sind aber, dass das Signal seinen Wert nur einmal innerhalb des Übergangs ändern darf und dass das Verhalten dieses Signals explizit beim Verlassen des betretenen Zustandes abgeprüft oder wiederum als „directed dont care" gekennzeichnet werden muss.

Zum Beispiel wird für das Signal D im Übergang 3 auf 0 ein „directed dont care“ benutzt. In allen den Zustand 0 verlassenden Transitionen wird dann explizit die Signaländerung D+ abgeprüft. Somit bieten die „directed dont cares“ eine gute Möglichkeit eine Kontrollerspezifikation kompakter darzustellen. Inwieweit die Benutzung der „directed dont cares“ sich auf die resultierende Implementierung auswirken, hängt wiederum von der Leistungsfähigkeit der verwendeten Synthesesoftware ab.

2.6 Probleme synchroner Schaltungen

Beim Entwurf digitaler integrierter Schaltungen stellt man einige vereinfachende Annahmen auf. Zu den wichtigsten hierbei zählen die Annahmen, dass Signale nur diskrete Werte annehmen und dass die Zeit sich nur diskret ändert. Die letztere Annahme gilt streng genommen nur für synchrone Schaltungen und kann deswegen zur Unterscheidung von asynchronen Schaltungen dienen.

2.6.1 Synchronität

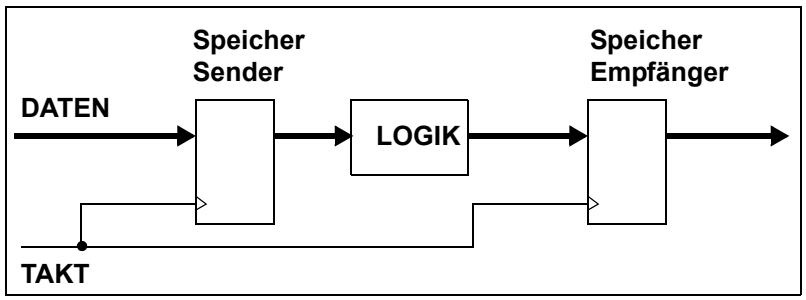


Bild 2.14: Synchroner Datentransfer

In einer synchronen Schaltung sind Speicher (Flipflops, Register) enthalten, die mittels eines Taktes zeitgleich Daten übernehmen. So wechseln die Werte der Speicher nur zu diskreten Zeitpunkten und durch den Takt zum gleichen Zeitpunkt. Dies ist das Hauptmerkmal synchroner Schaltungen. Aus Bild 2.14 ist ersichtlich, warum man bei Modellen von synchronen Schaltungen oft von einer Register Transfer Beschreibung spricht. Man unterscheidet auf dieser Ebene zwischen den speichernden Elementen, den getakteten Registern, und der dazwischen befindlichen Logik zum Transferieren der Daten. Es ist dabei möglich, die Vorgängerstufe dieser Register immer als Sender und die Nachfolgestufe als Empfänger zu betrachten.

Aus dem Signalverlauf in Bild 2.15 kann man die grundlegenden Zeitbedingungen für synchrone Schaltungen erkennen. Die Annahme, dass sich die Werte nur zu diskreten Zeitpunkten ändern, spiegelt das Verhalten der Ausgänge der speichernden Elemente wider. Die Werte dieser Ausgänge ändern sich zeitgleich mit einer aktiven Flanke des gemeinsamen Taktes. Damit diese Sichtweise gültig bleibt, müssen die Daten am Eingang eines Registers lange genug vor

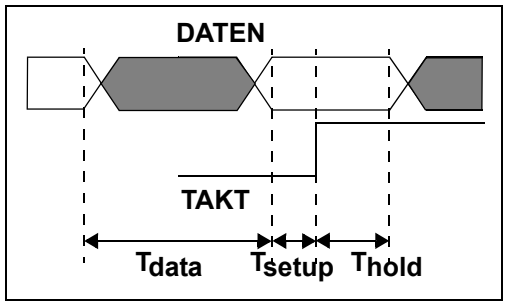


Bild 2.15: Zeitbedingung für den synchronen Datentransfer

der aktiven Taktflanke (T_{setup}) und lange genug nach der aktiven Taktflanke (T_{hold}) stabil anliegen. Ist der längste Datenpfad (T_{data}) im gesamten Entwurf bekannt, dann ist durch diese Bedingung die maximal mögliche Taktfrequenz vorgegeben. Ist diese Taktfrequenz zu gering ist der längste Datenpfad (kritischer Pfad) entsprechend zu verkürzen.

Für synchrone Schaltungen gilt also folgende Zeitbedingungen:

$$T_{crit} < T_{clock} - T_{setup} - T_{uncertain} \qquad (2.7)$$

$$T_{min} > T_{hold} \qquad (2.8)$$

Dabei ist T_{crit} die maximale Laufzeit eines Einzelpfades zwischen zwei Flipflops einer synchronen Schaltung, welche von einem Takt mit der Taktperiode T_{clock} getaktet wird. T_{setup} bezeichnet die Zeit vor der aktiven Taktflanke (Vorbereitungszeit), innerhalb dessen sich die Flipflop Eingangswerte nicht ändern dürfen. $T_{uncertain}$ ist ein Zeitbetrag, den man aus Sicherheitsaspekten subtrahiert. Er umfasst Faktoren, wie zum Beispiel den Taktversatz.

Definition 10:

Der Taktversatz (clock skew) bezeichnet die Differenz der unterschiedlichen Ankunftszeitpunkte des Taktes an unterschiedlichen Stellen in einer Schaltung.

Gleichung *(2.8)* besagt, dass die kürzeste Laufzeit T_{min} eines Einzelpfades zwischen zwei Flipflops einer synchronen Schaltung größer als die Haltezeit (auch Übergabezeit) T_{hold} nach einer aktiven Taktflanke sein muss.

Gleichung *(2.7)* lässt erkennen, dass die Performanz einer synchronen Schaltung durch den langsamsten Pfad (T_{crit}) vorgegeben ist. Die Laufzeit dieses Pfades hängt nicht zuletzt von der verwendeten Technologie ab. Auch die Setup- und Hold Zeit (T_{setup} und T_{hold}) hängen von der verwendeten Technologie ab. Als Folge dieser Abhängigkeit muss nun der Entwickler bei vorgegebener Taktfrequenz sicherstellen, dass die Gleichungen *(2.7)* und *(2.8)* für alle vorgegebenen Einsatzbedingungen der Schaltung erfüllt sind.

Ein besonderes Problem synchroner Schaltungen ist die Sicherstellung der Synchronität, das heißt hier, des technisch hinreichend gleichzeitigen Schaltens aller Flipflops einer Schaltung. Dies kann nur der Fall sein, wenn die aktive Taktflanke zeitgleich an jedem Flipflop in der gesamten Schaltung wirksam ist. Dies sicherzustellen, ist in Anbetracht vieler Abhängigkeiten (Lage der Schaltungsteile auf dem späteren elektronischen Baustein, Verdrahtung der einzelnen Schaltungselemente, unterschiedliche Einsatzbedingungen, etc.) keine triviale Aufgabe. Die Synchronität sicherzustellen ist mit zunehmender Schaltungsgröße immer schwieriger und wird somit in Zukunft zu noch größerem Schaltungs- und Entwurfsaufwand führen.

Diese und andere hier noch nicht erwähnte Probleme können asynchrone Schaltungen theoretisch vermeiden beziehungsweise lösen. Allerdings ist auch der Entwurf asynchroner Schaltungen nicht unproblematisch.

2.6.2 Verlustleistung

Es ist leicht ersichtlich, dass eine synchrone Schaltung auch im „Ruhezustand“, das heißt, wenn sich die Daten nicht ändern, Energie durch den Takt verbraucht. Der Takt lädt und entlädt ständig Kapazitäten, wobei die Umladungsenergie als Wärme verlorengeht.

Allgemein gibt es drei verschiedene Ursachen für Verlustleistung in digitalen CMOS-Schaltungen [77] [56]:

$$P_{Total} = P_S + P_{KS} + P_{Le} \qquad 2.1$$

- $\mathbf{P_S}$ repräsentiert die Schaltverluste, die durch das Auf- und Entladen von Kapazitäten entstehen.
- $\mathbf{P_{KS}}$ sind Verluste, die durch direkte Kurzschlussströme entstehen, die beim Schalten der CMOS-Schaltung auftreten, weil der NMOS- und PMOS-Transistor gleichzeitig leiten.
- $\mathbf{P_{Le}}$ stellen Leckverluste dar, die durch das Sperrverhalten von Dioden und Transistoren hervorgerufen werden.

Die verschiedenen Mechanismen tragen mit unterschiedlichen Anteilen zum gesamten Leistungsverbrauch P_{Total} bei. In [56] wird die in Bild 2.1 abgebildete Aufteilung vorgestellt, die für konventionelle CMOS-Schaltungen gültig ist. Bei integrierten Schaltungen der neuesten Generationen kann der Anteil der Leckverluste allerdings bis zu 20% betragen.

P_S verursacht den Hauptanteil der gesamten Verlustleistung. Genau an diesem Anteil ist wiederum der Takt einer synchronen Schaltung in erheblichen Umfang beteiligt:

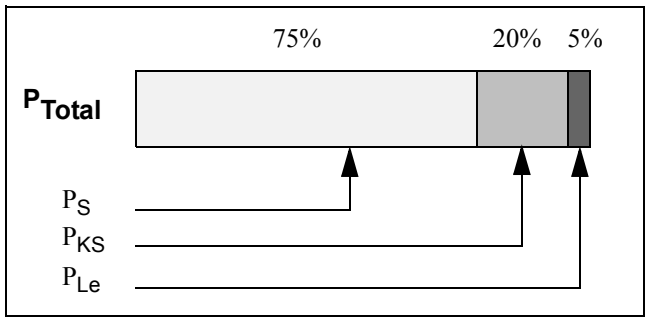


Bild 2.1: Aufteilung der Verlustleistungen

Entstehung der Schaltverluste

Zur Verdeutlichung, wie die Schaltverlustkomponente bei der Verlustleistung entsteht, wird in Bild 2.2 ein konventioneller CMOS-Inverter betrachtet [56].

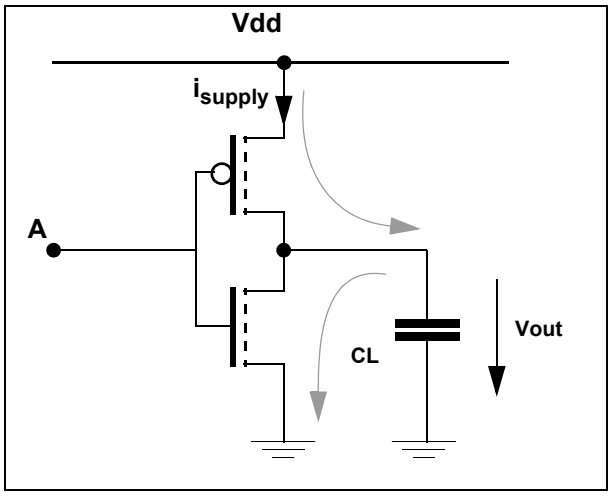


Bild 2.2: Modell eines CMOS-Inverters

C_L am Ausgangsknoten steht für die Summe aller externen und internen (parasitären) Lastkapazitäten des Inverters. V_{out} stellt den Spannungsabfall über C_L dar. Findet am Eingang des Inverters ein Signalwechsel statt, der zu einem 0→1 Wechsel am Ausgang führt, so fließt über den PMOS-Transistor der Ladestrom i_{supply}. Dieser ist:

$$i_{supply} = C_L \frac{dV_{out}}{dt} \qquad (2.2)$$

Weiter gilt für die Leistung P(t):

$$P(t) = \frac{dE}{dt} = i_{supply} \cdot V_{dd} \qquad (2.3)$$

Mit Gleichung *(2.2)* und Gleichung *(2.3)* kann die Energie berechnet werden, die bei diesem Ladevorgang von der Versorgungsquelle geliefert wird:

$$E_{0\to 1} = \int_0^T P(t)dt = V_{dd}\int_0^T i(t)_{supply}dt = V_{dd}\int_0^{V_{dd}} C_L dV_{out} = C_L \cdot V_{dd}^2 \qquad (2.4)$$

Der Anteil der Energie, die in C_L gespeichert wird, ist nun die Hälfte davon:

$$E_{cap} = \int_0^T P_{cap}(t)dt = \int_0^T V_{out} i_{cap}(t)dt = \int_0^{V_{dd}} C_L V_{out} dV_{out} = \frac{1}{2} C_L \cdot V_{dd}^2 \qquad (2.5)$$

Beim Übergang am Ausgang des Inverters (V_{out}), tritt somit folgende Energieverteilung auf:

- Energie, die aus der Spannungsversorgung abfließt: $C_L V_{dd}^2$
- Energie, die in C_L gespeichert wird: $\frac{1}{2} C_L V_{dd}^2$
- Energieverlust am PMOS-Transistor, der als Verlustwärme abgegeben wird: $\frac{1}{2} C_L V_{dd}^2$

Beim Übergang von $V_{dd} \to 0$ am Ausgang (V_{out}) wird die ganze in C_L gespeicherte Energie $\frac{1}{2} C_L V_{dd}^2$ über den NMOS-Transistor ebenfalls als Verlustwärme abgegeben. Weitere Energie wird bei diesem Entladevorgang nicht aus der Versorgungsquelle bezogen. Die beim Aufladeprozess aufgenommene Leistung wird somit bei konventionellen CMOS-Strukturen bei jedem Schaltzyklus, der aus Auf- und Entladen von C_L besteht, zu 100% als Wärme abgegeben.

Wenn ein Paar von steigenden und fallenden Übergängen an der Last-Kapazität mit einer Rate von f_{Clock} auftreten, dann berechnet sich die gesamte Leistung, welche von der Energiequelle abfließt, folgendermaßen:

$$P_{Drawn} = C_L V_{dd}^2 f_{Clock} \qquad (2.6)$$

Gleichung *(2.6)* gilt aber nur für den Takt selbst. Im Allgemeinen treten Schaltaktivitäten am Inverter nicht mit der Rate f_{Clock} auf. Um dies zu berücksichtigen, wird die Schaltwahrscheinlichkeit $\alpha_{0\to 1}$ eingeführt. $\alpha_{0\to 1}$ steht für die Wahrscheinlichkeit, dass innerhalb einer Taktperiode der Ausgang vom logischen Wert ‘0’ auf den logischen Wert ‘1’ wechselt.

Somit kann man allgemein die durchschnittliche Schaltverlustleistung eines CMOS-Gatters wie folgt definieren:

$$P_S = \alpha_{0\to 1} C_L V_{dd}^2 f_{Clock}; \quad (0 \le \alpha_{0\to 1} \le 0,5) \qquad (2.7)$$

In der Literatur ist auch der allgemeine Faktor α für die generelle Schaltwahrscheinlichkeit eines Gatters geläufig. Bei Verwendung dieses Faktors muss man beachten, dass hier pro betrachteten Schaltvorgang (Laden oder Entladen) nur die halbe Energie verbraucht wird. Andererseits werden alle zwei möglichen Wertewechsel berücksichtigt. Durch die Beziehung $\alpha=2\alpha_{0\rightarrow1}$ sind beide Darstellungsvarianten (2.7) und (2.8) gleichwertig.

$$P_S = \frac{\alpha}{2} C_L V_{dd}^2 f_{Clock}; \qquad (0 \le \alpha \le 1) \tag{2.8}$$

C_L und $\alpha_{0\rightarrow1}$ wird in der Literatur häufig zu der effektiven Kapazität C_{eff} zusammengefasst:

$$C_{eff} = \alpha_{0\rightarrow1} C_L \tag{2.9}$$

Wie aus Gleichung *(2.6)*, Gleichung *(2.7)* und Gleichung *(2.9)* ersichtlich, kann der Einfluss von P_S durch folgende Maßnahmen verringert werden:

- Reduzierung der Taktrate, beziehungsweise Ausblenden oder Beseitigen des Taktes
- Verkleinerung der tatsächlichen Kapazität C_L
- Verringerung der Schaltaktivität α
- Reduzierung der Betriebsspannung V_{dd}

Der ursprüngliche Anstoß, asynchrone Schaltungen und deren Entwurf erneut zu untersuchen, wurde durch das Problem der Reduzierung der Verlustleistung gegeben. Der interessierte Leser kann im Anhang A bis Anhang D mehr zum Thema „*Low Power*“ - Entwurf nachlesen.

2.7 Probleme asynchroner Schaltungen

Die in der Einführung aufgeführten Punkte lassen den Eindruck erwecken, dass asynchrone Schaltungen die Lösung für die Probleme beim heutigen Entwurf integrierter Schaltungen sind. Sie versprechen schnellere Schaltungen, die unempfindlicher sind gegenüber Parameterschwankungen und Umwelteinflüssen, selbst die Umwelt dabei weniger stören und dabei weniger Energie verbrauchen als ihre synchronen Pendants.

Die Praxis zeigt aber, dass auch einige schwerwiegende negative Aspekte beim Entwurf asynchroner Schaltungen zu beachten sind. Diese sind der Grund, warum man in der Allgemeinheit synchrone Realisierungen bevorzugt. Asynchrone Schaltungen lassen sich zum Beispiel nicht in der Weise vorausschauend entwerfen, wie synchrone Schaltungen.

Bei synchronen Schaltungen muss der Entwickler nur darauf achten, dass die größte Laufzeit in der Schaltung nicht größer als das durch den Takt vorgegebene Maximum ist. Ist diese

Bedingung erfüllt, hat man schon weitestgehend sichergestellt, dass die Schaltung später korrekt arbeitet.

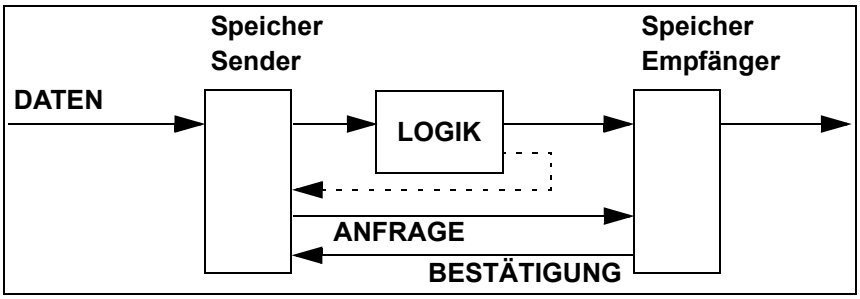


Bild 2.3: Asynchroner Datentransfer

Wie in Bild 2.3 ersichtlich ist, tauschen asynchrone Schaltungen Daten zumeist über ein sogenanntes Handshake (Datenübergabe-) Protokoll aus. Eine Bedingung für das Funktionieren des Datenaustausches ist, dass das Anfragesignal erst dann ein neues Datum signalisiert, wenn dieses auch tatsächlich stabil an den Eingängen des Empfängers anliegt. Das heißt, es liegt hier eine lokale Zeitbedingung vor, die unabhängig von den gerade verarbeiteten Daten und den Umgebungseinflüssen einzuhalten sind. Überall in einer asynchronen Schaltung, die Daten über solch ein Protokoll austauscht, entstehen lokale Zeitbedingungen, deren Erfüllung gesondert zu behandeln sind.

Bei asynchronen Schaltungen ist also nach der Synthese immer wieder das dynamische Verhalten der Schaltung zu untersuchen. Der Datenaustausch erfolgt aufgrund festgelegter Protokolle, die strenge funktionale und zeitliche Vorgaben machen. Es ist zu überprüfen, ob diese Vorgaben nach der Synthese immer noch eingehalten werden. Dabei muss der Entwickler ausreichende Erfahrungen haben, um im Voraus schon Problemstellen identifizieren und eventuell vermeiden zu können. Ansonsten läuft er Gefahr, in einer Endlosschleife (Korrigieren - Überprüfen) steckenzubleiben.

Im Einzelnen muss man sich beim Entwurf asynchroner Schaltungen mit folgenden Problemen auseinandersetzen:

Hazards:

Betrachtet man eine technische Realisierung eines Schaltnetzes, bestehend aus Elementebausteinen, so reagiert diese auf eine Eingangswertänderung mit der Bildung des entsprechenden Ausgangswert. Allerdings ergeben sich aufgrund der parallel laufenden Pfade innerhalb des Logikblocks unterschiedliche Ankunftszeiten von Zwischenergebnissen an den Ausgangsgattern. Somit kann es zu ungültigen Zwischenwerten am Ausgang des betrachteten Logikblocks kommen. Die Möglichkeit, dass an einer Stelle ungültige Zwischenwerte auftreten können, bezeichnet man als Hazard.

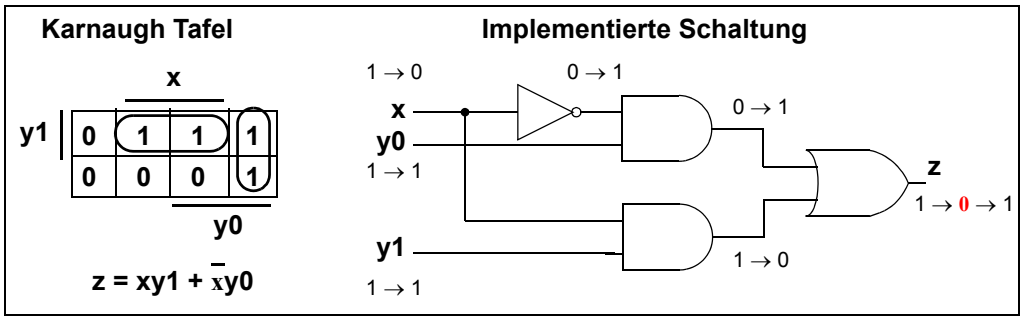


Bild 2.4: Beispielschaltung mit einem statischen 1-Hazard

In einer synchronen Schaltung macht dies keine Probleme, da ein speicherndes Element den (endgültigen) Ausgangswert erst dann übernimmt, wenn das Taktsignal eintrifft. Ungültige Zwischenwerte werden einfach ignoriert und können sich nicht über die speichernden Elemente hinaus fortpflanzen.

In asynchronen Schaltungen können sich ungültige Zwischenwerte leicht fortpflanzen, da hier der zentrale Steuermechanismus einer synchronen Schaltung fehlt. Beim Entwurf ist also sicherzustellen, dass keine ungültigen Zwischenwerte auftreten können, oder dass diese geeignet gefiltert werden, so dass keine Fehlfunktion der Schaltung auftreten kann. Dies ist aber nicht so einfach zu gewährleisten und erfordert zumeist andere und komplexere Synthesemechanismen als bei synchronen Schaltungen. So lässt sich der statische 1-Hazard in Bild 2.4 dadurch vermeiden, dass man den Term (Primimplikant) y_1y_0 ebenfalls in der endgültigen Schaltung implementiert. Gängige Synthesewerkzeuge für synchrone Schaltungen kennen die Nebenbedingung nicht und betrachten diesen Term als redundant und verwerfen ihn.

Timing:

Um das korrekte Funktionieren einer asynchronen Schaltung gewährleisten zu können ist es oftmals notwendig, das zeitliche Verhalten einzelner Signale nach der Synthese anzupassen. So ist zum Beispiel die Propagierung einzelner Steuersignale um den Betrag zu verzögern, der sicherstellt, dass das Signal erst nach den zu verarbeitenden Daten an der nachfolgenden Verarbeitungsstufe ankommt. Oftmals kann man auch eine hazardfreie Schaltung nur dann erzeugen, wenn diese bestimmte zeitliche Bedingungen einhält.

In asynchronen Schaltungen ist das dynamische Verhalten zu berücksichtigen und notfalls, zum Beispiel durch Einfügen von Verzögerungsgliedern (delay elements), anzupassen. Dies hat weitreichende Auswirkungen. So muss entweder das Entwurfswerkzeug oder der Entwickler dafür sorgen, dass die zeitlichen Bedingungen eingehalten werden. Dies bedarf eigener Synthesealgorithmen beziehungsweise Erfahrungen des Entwicklers. Zumeist muss man nach der Synthese nochmals in das Design eingreifen, um zum Beispiel passende Verzögerungselemente in die Schaltung einzufügen.

In synchronen Schaltungen muss man nur darauf achten, dass die maximale Laufzeit zwischen den Speichern nicht größer als das vorgegebene Maximum ist. Diese Bedingung gilt für alle Pfade, so dass bei den kombinatorischen Schaltungen letztendlich nur eine Zeitbedingung zu berücksichtigen ist. Bei asynchronen Schaltungen sind es dagegen mehrere unterschiedliche Zeitbedingungen, die zu einem zusätzlichen Entwurfsaufwand führen.

Verifikation:

Bei asynchronen Schaltungen vermehrt sich der Verifikationsaufwand. Gegenüber synchronen Schaltungen erschwert sich die Verifikation der korrekten Initialisierung und der logischen Funktionalität. Zusätzlich muss man asynchrone Schaltungen bezüglich dynamischer und statischer Hazards überprüfen.

Die Verifikation asynchroner Schaltungen erschwert sich zum Beispiel durch die eingefügten Verzögerungselemente oder auch durch notwendige redundante Hardware. Bei den Verzögerungselementen führen Parameterschwankungen zu veränderten Laufzeiten und können somit unter Umständen zu einer Fehlfunktion führen. Die sind außerdem auf Einhaltung der Toleranzen zu testen. Redundante Hardware erschwert die funktionale Verifikation, da durch sie die Auswirkungen bestimmter Wechsel der Eingangswerte nicht mehr an den Ausgängen der Schaltung zu beobachten sind.

Fläche:

Allgemein sind komplexere asynchrone Schaltungen größer als ihre synchronen Varianten. Dies liegt an den zusätzlichen Schaltungselementen (Verzögerungselemente, Komplettierungsdetektionsmechanismen, Schaltungen zur Vermeidung von Hazards), die in einer asynchronen Schaltung einzufügen sind. Je nach Realisierungsvariante kann dieser Flächenzuwachs ein Vielfaches gegenüber der synchronen Realisierung betragen. Ob dies akzeptabel ist, hängt von den geforderten Eigenschaften des Endprodukts ab und ist vorab zu überprüfen.

Performanz:

Trotz der oben erwähnten vielversprechenden Vorteile, die eine asynchrone Realisierung einer Schaltung verspricht, hat sich bisher gezeigt, dass diese in der Praxis nicht zu erreichen sind. Viele der bisherigen Prototypenentwürfe haben gezeigt, dass die asynchronen Versionen gegenüber den synchronen Varianten nicht nur größer sind, sondern oftmals auch langsamer. Somit ist der allgemeine Nachweis bisher nicht erbracht, dass asynchrone Schaltungen das Potential für bessere, schnellere Schaltungen besitzen.

Man erhält also trotz eines erheblichen Mehraufwands beim Entwurf mit einer asynchronen Schaltung nicht zwingend eine schnellere Schaltung. Dies liegt vor allem an den zusätzlichen Schaltungselementen, die in eine asynchrone Schaltung einzufügen sind, wie zum Beispiel Verzögerungselemente oder Komplettierungsdetektionsmechanismen.

Andere Aspekte wie Störsicherheit, niedrigerer Leistungsverbrauch und das Vermeiden des Taktverteilungsproblems sind zur Zeit also die vorwiegenden Gründe, die einen Einsatz asynchroner Schaltungen vorteilhaft machen könnten. Allerdings erzielt man auch auf dem Gebiet der Performanz Fortschritte, wie dies die Implementierung eines Instruction Length Decoders [81] zeigt.

Automatisierung:

Für asynchrone Schaltungen existieren bisher wenig kommerzielle Entwurfswerkzeuge, und noch keine etablierte Entwurfsmethodik. Es gibt viele Ansätze und Lösungen für Teilschritte, aber keinen durchgehenden Entwurfsablauf. Verantwortlich hierfür sind

a. die Notwendigkeit von zusätzlichen, dem synchronen Entwurf widersprechenden Synthesealgorithmen
b. zusätzlich notwendige Schaltungselemente (Muller C Element, Call, Toggle, ...), die in keiner Standardzellenbibliothek für synchrone Schaltungen vorhanden sind
c. Zusätzliche Verifikationsalgorithmen und -werkzeuge
d. die Unterschiede der bestehenden vorgeschlagenen Entwurfsmethodiken
e. fehlende Testkonzepte und Testvektorgeneratoren.

Diese fehlende Automatisierung zusammen mit den zusätzlich auftretenden Verifikationsschwierigkeiten sind Gründe, warum die Verwendung von asynchronen Schaltungen immer noch auf Anwendungen begrenzt ist, bei denen eine synchrone Realisierung schlicht nicht möglich ist.

3 Stand der Technik

3.1 Entwurfsverfahren

Bei dem Entwurf asynchroner Schaltungen gibt es unterschiedliche Zielsetzungen. Häufig ist nur ein Modul einer Schaltung asynchron aufzubauen; hier steht die Einbindung eines asynchronen Moduls, das eventuell per Hand entworfen wird, in die synchrone Umgebung im Vordergrund. Soll ein ganzer Entwurf asynchron aufgebaut sein, können unterschiedliche Ziele, wie minimale Fläche oder maximaler Durchsatz, eine Rolle spielen. Abhängig von diesen Zielen und dem Grad der Automatisierbarkeit haben sich im Laufe der Jahrzehnte unterschiedliche Entwurfsverfahren entwickelt, die hier kurz vorgestellt werden.

Zunächst werden die Konzepte von Huffman-Schaltungen, Mikropipelines, Signalübergangsgraphen basierten Schaltungen sowie der NULL Convention Logic vorgestellt und den entsprechenden Verzögerungsmodellen (Kapitel 2.4.1) zugeordnet. Es sei darauf hingewiesen, dass es natürlich noch einige andere Verfahren gibt. So gibt es die Ansätze über I-Nets [64][79], Trace basierte Schaltungen [21][29], Kommunizierende Prozesse (CSP) [31][13] und Asynchronous Wave Pipelines [27][28].

3.1.1 Bounded Delay Schaltungen

Das Bounded Delay Modell (Kapitel 2.4.1.1) geht von der Annahme aus, dass die Verzögerungszeiten aller Schaltungselemente bekannt oder zumindest begrenzt sind. Schaltungen, die mit diesem Modell entworfen werden, bezeichnet man allgemein als ’Huffman Schaltungen’, benannt nach D.A. Huffman, der viele der grundlegenden Konzepte entwickelt hat.

3.1.1.1 Huffman-Schaltungen

Ein naheliegendes Ziel beim asynchronen Entwurf ist es, möglichst nahe am synchronen Entwurfsvorgehen zu bleiben. Somit sollen die gleichen oder nur leicht modifizierten Entwurfswerkzeuge einsetzbar und auch die leichte Einbindung der asynchronen Module in eine synchrone Umgebung möglich sein. Eines der ersten Konzepte ist das der Huffman-Schaltungen [33].

Wie in Bild 3.1 zu sehen ist, ähneln sich Huffman-Schaltungen und synchrone Schaltungen. Im synchronen Design aktualisiert sich der Zustand (Signalwerte) der Schaltung mit jeder neuen aktiven Taktflanke. Das heißt, Signaländerungen haben zumeist eine Taktperiode T lang Zeit, um sich durch die Schaltung auszubreiten. Es spielt dabei keine Rolle, welches Zustandssignal sich wann innerhalb dieser Taktperiode ändert.

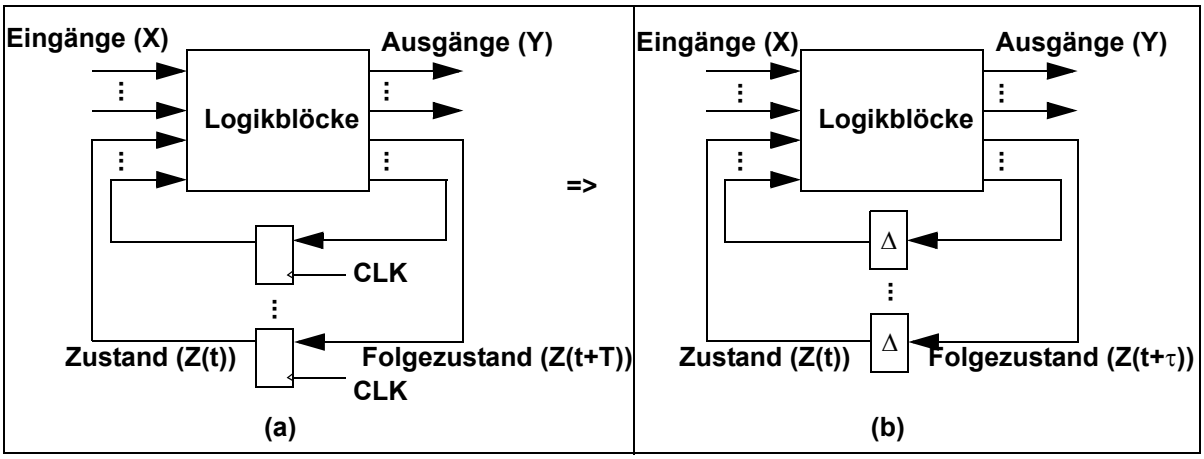


Bild 3.1: Datenspeicherprinzip bei (a) synchronen und (b) asynchronen Schaltungen

Im asynchronen Design sind die Flipflops durch Verzögerungselemente ersetzt. Auch hier werden neue Zustandssignale nach einer Zeit τ als aktuelle Zustandssignale übernommen. Diese Übernahme erfolgt allerdings nicht gemeinsam, da sich jedes einzelne Signal unabhängig von den anderen verzögern darf. Deswegen dürfen sich weder die Eingangssignale noch die Zustandssignale beliebig ändern, sondern nur nach bestimmten Regeln, um unter Anderem sogenannte *critical races* [99] zu vermeiden.

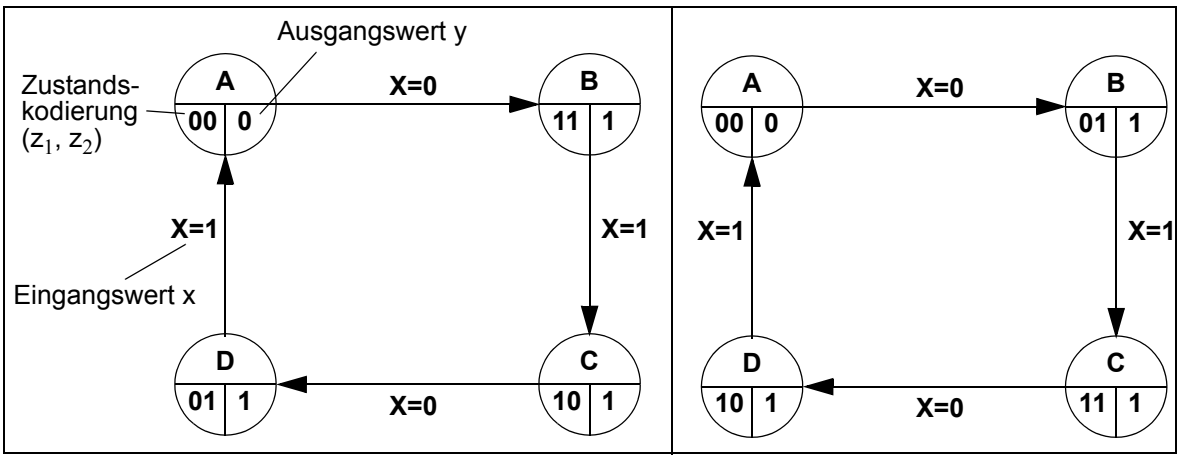


Bild 3.2: Beispiel für einen Automat mit einem critical race (links) und ohne critical race (rechts)

Ein critical race kann zum Beispiel bei einem Zustandswechsel in Bild 3.2 links vom Zustand A nach B auftreten. Für diesen Zustandswechsel müssen beide Zustandsbits von ’0’ auf ’1’ wechseln. Wechselt hierbei das erste Zustandsbit als erstes (“01“), so kann es passieren, dass der Automat in den Zustand D wechselt und nicht in den Zustand B.

Durch das Festlegen von Regeln für das Wechseln von den Zustands- sowie Ein- und Ausgangsbit und einer speziellen Zustandskodierung kann man critical races verhindern. Bild 3.2 rechts zeigt das Beispiel aus Bild 3.2 links mit veränderter Zustandskodierung. Vorausgesetzt

ist, dass sich zuerst das Eingangsbit x ändert, danach die Zustandsbits und das Ausgangsbit. Erst danach darf sich wieder ein Eingangsbit ändern. Durch diese Regeln und die ausgewählte Kodierung ist sichergestellt, dass sich immer nur ein Bit während eines Wechsels ändert. Diese Betriebsart bezeichnet man als *Fundamental Mode*.

Definition 11: Fundamental Mode Huffman-Schaltungen

Fundamental Mode Huffman-Schaltungen sind asynchrone Schaltungen, bei denen als speichernde Elemente Verzögerungselemente zum Einsatz kommen. Es gilt die Regel, dass sich der Eingangswert nur dann ändern darf, wenn die Schaltung eingeschwungen ist.

Entwurfsverfahren

Das Entwurfsverfahren von Huffman-Schaltungen unterscheidet sich nur geringfügig von dem Entwurf synchroner Schaltungen. Ausgehend von einer Spezifikation kann man eine Flusstafel erzeugen. Diese ist in ein Karnaugh Diagramm umzusetzen. Aus diesem leitet man wiederum die Boole'schen Gleichungen ab. Um Hazards zu vermeiden, muss man aber im Gegensatz zu synchronen Schaltungen alle Primimplikanten verwenden. Das zu verwendende Synthesewerkzeug muss also in der Lage sein, hazardfreie Logik zu erzeugen. Die meisten Synthesewerkzeuge können dies aber nicht.

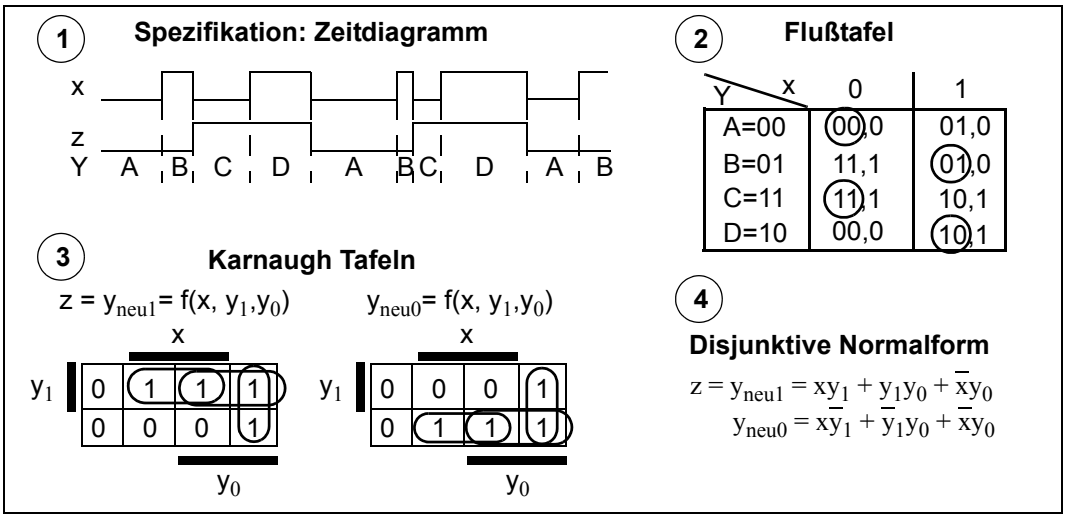


Bild 3.3: Beispiel für den Entwurf einer Huffman-Schaltung

Des Weiteren sind immer die Bedingungen des Fundamental Modus zu erfüllen. Das heißt, um mehrere Huffman-Schaltungen miteinander zu verknüpfen, dürfen sich auch die Ausgänge nur jeweils an einer Bitstelle ändern. Dies ist so, da die Ausgänge eines Moduls gleich die Eingänge des nächsten Moduls sind. Letztendlich behandelt man die Ausgänge gleichberechtigt zu dem internen Zustand der Schaltung.

Bild 3.3 zeigt exemplarisch die Einzelschritte des Entwurfs einer Huffman-Schaltung [29]. Aus der Spezifikation, hier ein Zeitdiagramm, ergibt sich eine Flusstafel. In dieser sind die stabilen Zustände mit Kreisen markiert. Aus der Flusstafel leiten sich die Karnaugh Tafeln für die

Zustands- und Ausgangssignale ab. Aus diesen bildet man die Disjunktiven Normalenformen. Um Hazards zu vermeiden, kann man zum Beispiel alle Primimplikanten verwenden, wie dies in diesem Beispiel getan worden ist. Aus den Boole'schen Gleichungen des vierten Schrittes lässt sich die Schaltung aus Grundgattern nachbauen.

3.1.1.2 Hollaar-Schaltungen

Eine Erweiterung des Fundamentalmodus sind die Schaltungen nach Hollaar [32]. Ziel ist, dass sich die Eingangswerte früher ändern dürfen. Man geht davon aus, dass die Zustände eines Automaten 1-aus-n kodiert sind (siehe Kapitel 2.4.2). Das heißt, jedem Zustand wird ein eigenes Bit zugewiesen; in dem sich ergebenden Zustandsvektor ist also immer nur ein Bit auf '1' gesetzt.

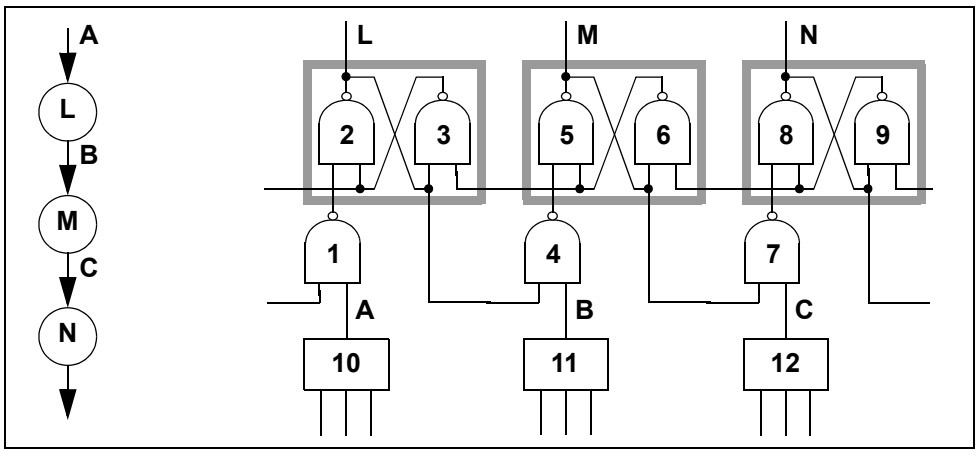


Bild 3.4: Prinzip von Hollaar-Schaltungen

Bild 3.4 zeigt auf der linken Seite einen Ausschnitt aus einem Automaten und auf der rechten Seite einen Ausschnitt aus der entsprechenden Realisierung. Man erkennt, dass die Ausgänge der SR-Latches die jeweiligen Zustände repräsentieren. Die Übergänge in dem Automaten sind als entsprechende Aktivierungsfunktion in der Realisierung zu erkennen. Das entsprechende Zustandsbit beziehungsweise SR-Latch (zum Beispiel M, Gatter 5) ist gesetzt, wenn die entsprechende Aktivierungsfunktion (B) und das vorherige Zustandsbit (L, Rückkopplung des Gatters 2) den Wert '1' besitzen. Das Zustandsbit (M) ist dann wieder zurückgesetzt, wenn das nachfolgende Zustandsbit (N, Rückführung von Gatter 9 nach Gatter 6) auf den Wert '1' gegangen ist.

Der Vorteil bei den Hollaar Modellen ist, dass eine Eingangsänderung nicht warten muss, bis der Folgezustand stabil anliegt: Wenn L den Wert '1' besitzt und nun B den Wert '1' annimmt, dann ist M sicher gesetzt, nachdem die Gatter 4, 5 und 6 geschaltet haben. Nun könnten sich die Eingangssignale wieder ändern, auch wenn diese zu einer Zustandsänderung führen, obwohl die Schaltung noch nicht stabil ist (L ist immer noch gesetzt!). Im Fundamentalmodus

müsste man noch warten, bis auch die Gatter 3, 2 und 4 geschaltet haben. Man müsste also fast doppelt so lange mit einer Eingangsänderung warten.

Die Hollaar-Schaltung ist aber leider nicht fehlerfrei: Gegeben ist zum Beispiel der aktuelle Zustand = L, die Übergangsfunktionen B=x1 und $C=\overline{x}_1 * f(x_2, x_3, x_4)$; weiterhin ist der Wert von f='1' und der Wert von x_1 ändert sich gerade von '0' nach '1'. C braucht auf Grund der größeren Komplexität länger, seinen Wert zu ändern ('1'→'0') als B ('0'→'1'). Ist dieser Laufzeitunterschied (D_C - D_B) größer als die Laufzeit durch die Gatter 4 und 5, dann produziert Gatter 7 einen Hazard ('0'→'1'→'0'). Dadurch wird ungewollt das Zustandsbit N gesetzt oder dieses oszilliert. In beiden Fällen ergibt sich ein ungewolltes Verhalten.

Weiterhin gilt für den allgemeinen Fall, dass ein Zustand mehrere Folgezustände hat, dass man leider 5 Gatterverzögerungen (4, 5, 6, 3, 2) mit einer erneuten Eingangsänderung warten muss, da ansonsten ein "paralleles" Zustandsbit unerwünscht gesetzt werden kann. Ein weiteres Problem ist, dass sich in einer Hollaar-Schaltung die Zustandsbits schnell hintereinander ändern können. Somit steigt die Fehlerwahrscheinlichkeit der nachgeschalteten Logik, welche die Zustandsbits auswertet.

3.1.1.3 Lokal getaktete Schaltungen

Die Bedingung „Single Input Change" (SIC), dass sich nur ein Eingangsbit ändern darf, ist zumeist zu streng. Bei Burst Mode Schaltungen (Kapitel 2.5.2) dürfen sich im Gegensatz zum SIC mehrere Eingangswerte ändern, so lange dies in einem bestimmten Zeitfenster passiert. Im Allgemeinen gibt es Gruppen von Eingangsbits, die sich ohne größere Probleme ändern dürfen. Eine solche Wertänderung von mehreren Eingangsbits nennt man *input burst*. Man geht davon aus, dass jeder input burst einen entsprechenden output burst (Änderung der Ausgangswerte) und eventuell eine Zustandsänderung hervorruft.

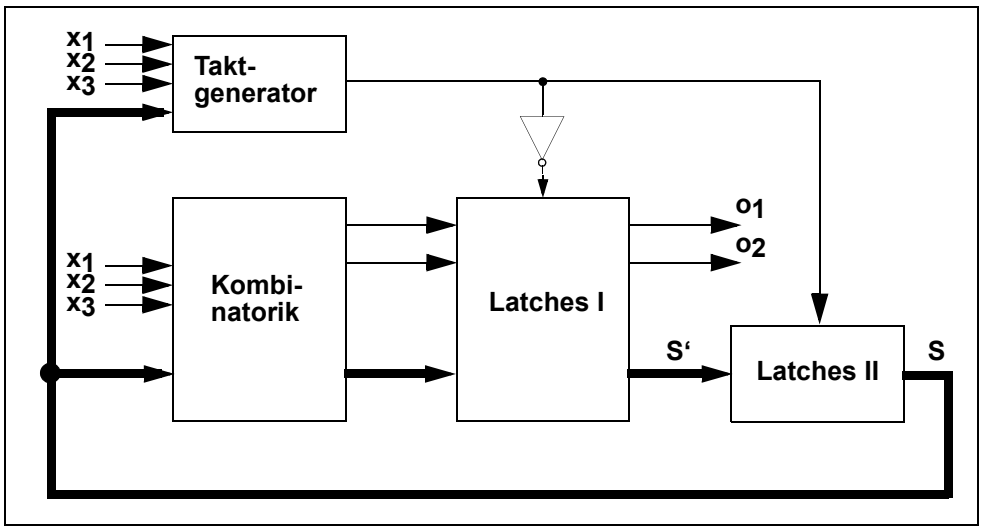


Bild 3.5: Implementierung einer Burst Mode Schaltung
als selbstgetaktete Schaltung

Realisiert werden Burst Mode Schaltungen unter anderem durch selbstgetaktete Schaltungen [70] [71], wie dies in Bild 3.5 dargestellt ist. Soll sich der Zustandsvektor ändern, generiert die Taktlogik einen Wertewechsel des Taktes, der die Latch Bank 1 (dynamische Latches) sperrt und die Latch Bank 2 (statische Latches) transparent schaltet. Hiermit wird ein neuer Zustandsvektor übernommen, der seinerseits nun durch die Logik propagiert und letztendlich für ein Zurücksetzen des Taktes sorgt.

Für die Burst gilt allgemein, dass kein Burst eine Untermenge eines anderen Burst sein darf und ein Eingangs-Burst nur dann erfolgen darf, wenn die Schaltung eingeschwungen ist.

Entwurfsverfahren

Nowick und Dill haben in [70] und [71] gezeigt, dass es möglich ist, Burst Mode Schaltungen zu synthetisieren. Die Schaltung wird als Signalübergangsgraph (siehe auch Kapitel 3.1.4.1) spezifiziert. Aus dem Signalübergangsgraph leiten sie sowohl für die Ausgangslogik als auch für das Erzeugen des Taktes eine Flusstafel ab, welche sie minimieren. Aus den Flusstafeln generieren Nowick und Dill die logische und zeitliche Implementierung. Hierbei setzen sie aber voraus, dass das entsprechende Synthesewerkzeug 5 Bedingungen einhält:

1. Die Taktlogik und Ausgangslogik darf für jeden erlaubten Eingangs-Burst keine logischen Hazards haben.
2. Die minimale Ausbreitungszeit durch die Taktlogik darf nicht größer sein, als die maximale Ausbreitungszeit durch die Ausgangslogik plus Setup-, Hold- und Ausbreitungszeit der entsprechenden Latches.
3. Die Taktlogik muss stabil sein, bevor sie zurückgesetzt wird.
4. Jeder 1-0 Übergang des Taktsignals muss frei von sämtlichen Hazards sein.
5. Die Zeitverzögerung zwischen dem Öffnen der zweiten Latchbank und dem Sperren der ersten Latchbank muss kleiner sein, als die kleinste Verzögerung im Rückführungspfad.

3.1.1.4 Asynchronous Wave Pipelines

Die Technik der asynchronen Wave Pipelines ([27], [28]) verknüpft Eigenschaften konventioneller Pipelines (Bild 3.6a), synchroner Wave Pipelines (Bild 3.6b) und asynchroner Mikropipelines [89].

Wie bei konventionellen Pipelines ist auch bei den asynchronen Wave Pipelines die Logik in mehrere „Stufen“ unterteilt. Allerdings benötigt das konventionelle Pipelining zusätzliche Speicher, welche die Schaltung vergrößern und verlangsamen. Zur ursprünglichen Laufzeit durch die einzelnen Logikblöcke A, B und C kommen noch die Laufzeiten der Flipflops. Somit steigt die Latency merkbar an. Die minimal mögliche Taktperiode wird zudem durch die Setup- und Hold-Zeiten der Flipflops vergrößert. Nicht zuletzt nimmt die Verbrauchsleistung zu.

Ein Möglichkeit, die eben beschriebenen Nachteile zu mindern, ist, synchrone Wave Pipelines (Bild 3.6b) zu benutzen. Hier werden weniger Flipflops benutzt. Dafür müssen die Pfade

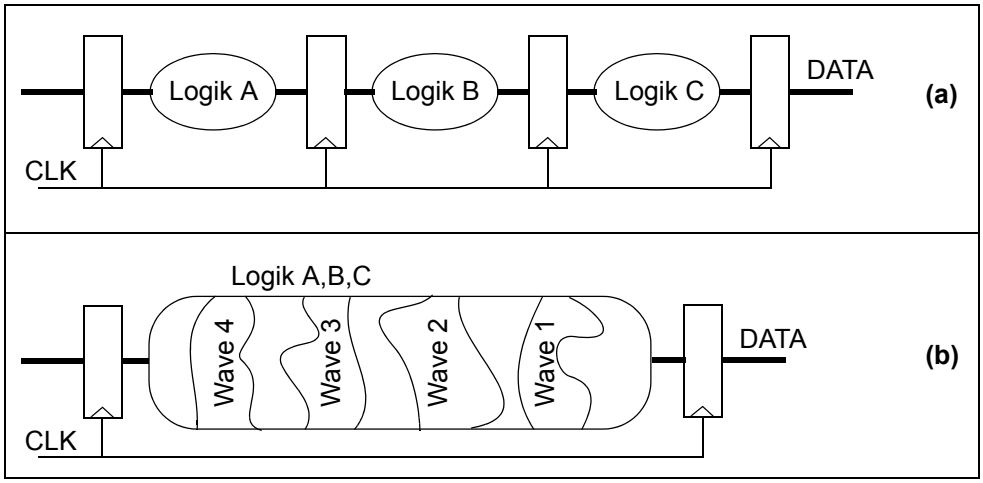


Bild 3.6: (a) Konvenionelle Pipeline; (b) Synchrone Wave Pipeline

bezüglich der Laufzeit der Daten hinreichend ausbalanciert werden. Wenn dies der Fall ist, überschneiden sich die Datenwörter nicht, die sich gleichzeitig in der Logik zwischen zwei Speichern befinden. Allerdings ist das Ausbalancieren der Datenpfade nicht einfach und es müssen zumeist Verzögerungselemente eingefügt werden. Asynchrone Wave Pipelines übernehmen das Konzept der wandernden Wellen, benutzen aber, wie oben beschrieben, auch Techniken konventioneller Pipelines.

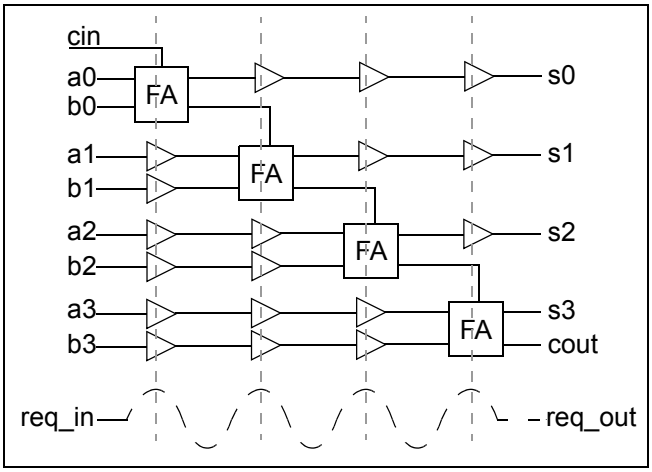


Bild 3.7: 4-Bit Addierer als asynchrone Wave Pipeline [28]

Mit den Mikropipelines gemein ist die asynchrone Datenverarbeitung, gesteuert durch Handshake Signale. Allerdings verzichten asynchrone Wave Pipelines auf die Elastizität der Mikropipelines. Somit reicht ein einzelnes Request Signal aus und es wird kein rückführendes Acknowledge Signal benötigt. Mit nur einem vorwärtsgerichteten Request Signal beschränkt man sich auch auf Schaltungen ohne Rückführungen. Bei den asynchronen Wave Pipelines wird die Logiktiefe zwischen den Speichern anders als bei den Mikropipelines in einzelne Stufen unterteilt.

Bild 3.7 zeigt das Funktionsprinzip einer asynchronen Wave Pipeline anhand einer 4-Bit Ripple Addierer Implementierung. Entsprechend der Ripple Architektur werden die einzelnen Bitsummen nacheinander gebildet. Annähernd gleiche Laufzeiten erreicht man letzten Endes durch Einfügen von Verzögerungselementen in den entsprechenden Pfaden. Zusammen mit dem Datensatz bewegt sich das Request Signal von der Eingabeseite zur Ausgabeseite. Somit wandern die Datenwellen zusammen mit der Steuerwelle durch die Logik und können an der Ausgabeseite mit Hilfe des Request Signals abgegriffen werden.

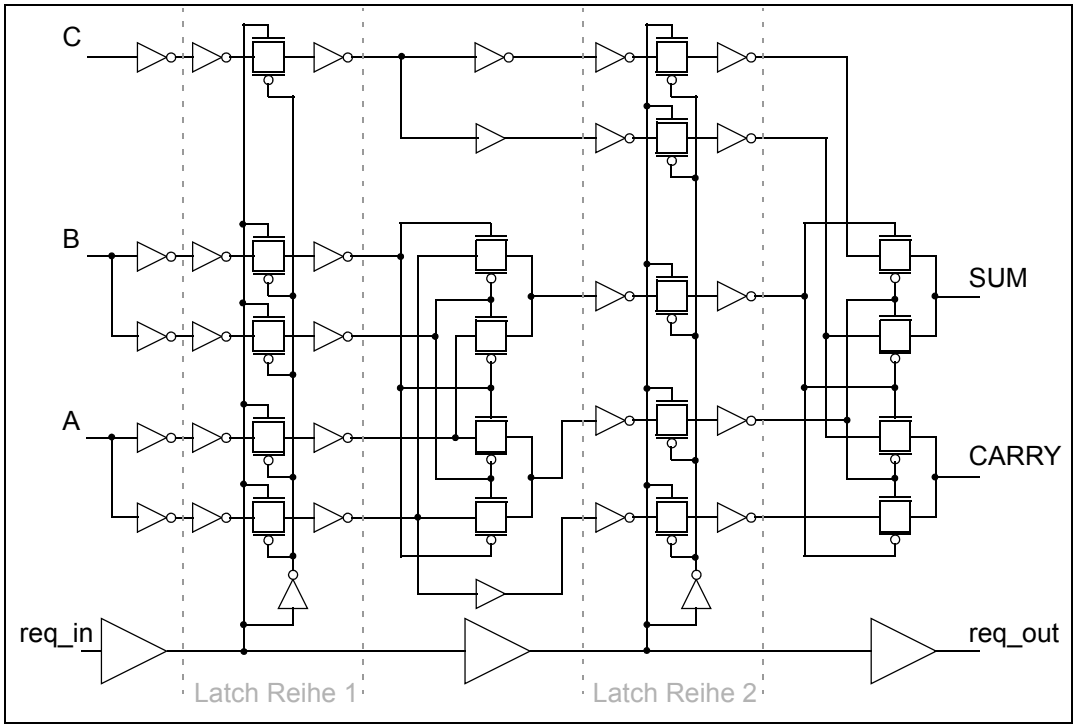


Bild 3.8: Implementierung eines Volladdierers als asynchrone Wave Pipeline [28]

Die Verzögerung des Request Signals erfolgt über mehrere kleinere Verzögerungselemente, da das Signal innerhalb der Volladdierer (FA) zur korrekten Ausführung der Addition benötigt wird. Dies ist in Bild 3.8 zu sehen. Als Zwischenspeicher werden dynamische Latches (Inverterpaare mit dazwischengeschalteten Transmissionsgattern) verwendet. Die Schaltung ist lokal synchron, global aber asynchron. Hier ist also die direkte Parallele zu den Mikropipelines zu erkennen. Die Beschränkung auf eine geringe Logiktiefe zusammen mit der Verwendung von dynamischen Latches führt das Konzept der konventionellen Pipeline in das einer Wave Pipeline über.

3.1.2 Delay Insensitive Schaltungen

Das Modell der delay insensitiven Schaltungen verhält sich genau umgekehrt zum ’Bounded Delay’ Modell. Man nimmt hier an, dass die Verzögerungen der Elemente und der Leitungen unbegrenzt (*unbounded delays*, siehe Kapitel 2.4.1.1) sind. Daraus ergibt sich die notwendige

Forderung, dass beim Datenaustausch zwischen zwei Blöcken der Empfänger dem Sender ein Bestätigungssignal zukommen lassen muss. Der Sender hat mit dem Versenden des nächsten Datenpakets so lange zu warten, bis er dieses Signal erhalten hat. Weiterhin müssen die Daten an sich kodiert werden, da zu keinem Zeitpunkt klar ist, ob die Daten den korrekten Wert enthalten. Dies betrifft sowohl den Kontroll- als auch den Datenpfad. Eine interessante Variante von delay insensitiven Schaltungen wird in den folgenden Kapiteln vorgestellt.

3.1.2.1 NULL Convention Logic

NULL Convention Logic^{TM} (NCL^{TM}) [22][54][85][100] ist eine taktfreie, delay-insensitive Design Methode für digitale Systeme. Es wird eine vollständige symbolische Logik verwendet, die Prozesse im Ganzen beschreiben kann. Im Gegensatz dazu steht die traditionelle Boole'sche Logik, die mit einem Taktsignal gekoppelt werden muss, um die gleichen Prozesse erfassen zu können. Die Design Methode und die neue symbolische Logik stellen die NULL Convention dar.

Grundbausteine für die NCL^{TM} Schaltungen sind die Dual Rail Kodierung (siehe Kapitel 2.4.2) und die Verwendung von Threshold-Gattern. Bei der Dual Rail Kodierung erfolgt der Datenaustausch, in dem sich Datenfronten und Leerfronten (kein Signal führt ein Datum) abwechseln. Bei der NCL^{TM} wird die Kontrolle dieser Vereinbarung bis auf die Gatter, die Threshold Gatter, abgebildet. In Bild 3.9 sind zwei verschiedene Arten von Threshold Gattern dargestellt. Teilbild (a) zeigt ein einfaches 5-Input / Threshold-3 Gatter. Somit besitzt dieses Gatter 5 Eingänge und das Gatter schaltet, wenn 3 oder mehr Eingänge ein Datum (V_{dd}) führen.

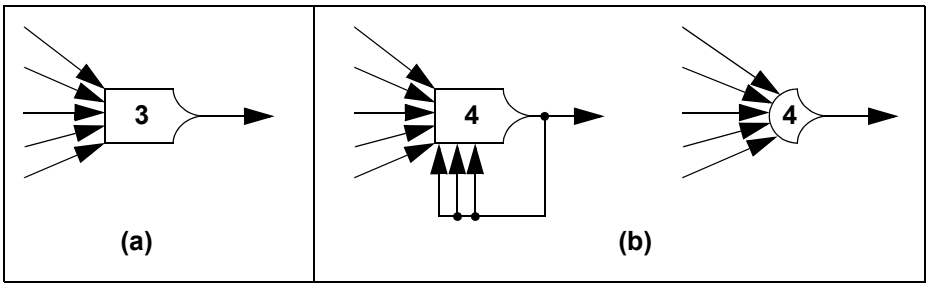


Bild 3.9: Einfaches Threshold Gatter (a) und ein speicherndes Threshold Gatter (b)

Während bei dem linken Gatter der Ausgang sofort auf NULL (= kein Datum = V_{ss}) zurückgeht, wenn die Threshold-Grenze unterschritten wird, ist dies bei dem Gatter des Teilbildes (b) nicht der Fall. Dieses besitzt N-1 Rückführungen, wenn N die Threshold-Grenze ist. Somit müssen alle puren Eingänge (hier 5) auf NULL (V_{ss}) liegen, damit auch der Ausgang kein Datum liefert. Somit ist ein speicherndes Threshold Gatter funktional die Verallgemeinerung eines Muller C-Elements (Bild 3.15 auf Seite 51). Ein zum Muller C-Element identisches Verhalten erhält man für Bild 3.9 (b), wenn hier die Threshold-Grenze auf 5 gesetzt wird; somit müssen

wie beim C-Element alle Eingänge den gleichen Wert besitzen, damit dieser auch am Ausgang übernommen wird. Die geforderte Vereinbarung dass sich Daten- und Leerfronten abwechseln ist hiermit erfüllt.

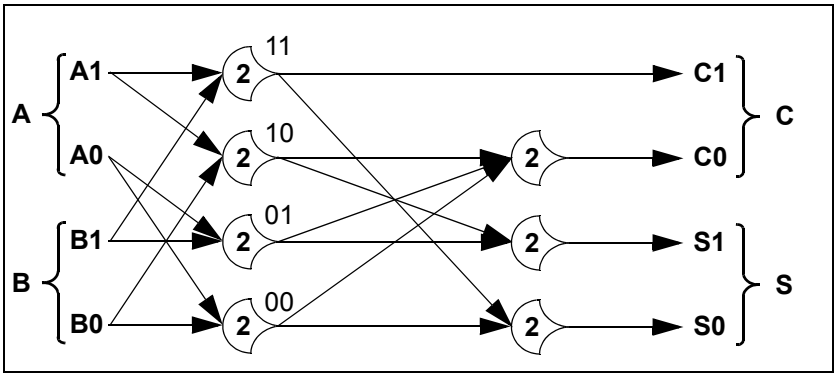


Bild 3.10: NULL Convention Halbaddierer

Bild 3.10 zeigt einen Halbaddierer, aufgebaut aus Threshold Gattern. Die Signale sind Dual Rail kodiert, das heißt pro Signal werden zwei Leitungen benötigt. An den Ausgängen der ersten Gatterreihe sind die entsprechenden Eingangsbelegungen als Symbole angeführt. Es ist zu sehen, dass nur dann ein Carry Bit erzeugt wird (C1=1), wenn A und B den logischen Wert 1 besitzen (A1=1 und B1=1). Alle anderen Eingangsbelegungen erzeugen eine logische 0 für das Carry Bit (C0=1).

Andererseits wechseln die Ausgänge der Hysterese behafteten Gatter erst von einem Datum auf NULL, wenn alle Eingänge eine logische 0 besitzen.

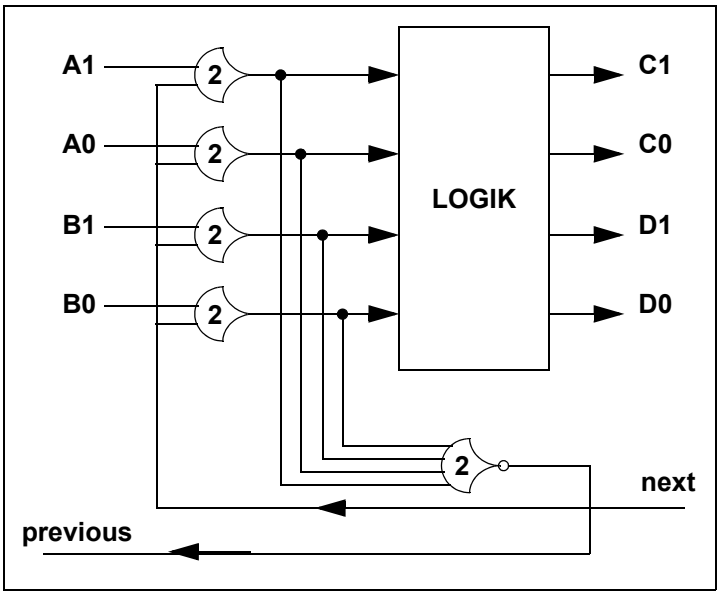


Bild 3.11: NULL Convention Register mit Steuerung und nachgeschalteter Logik

In Bild 3.11 ist die Speicherung der Daten und die Kommunikation mit der Umwelt gezeigt. Verwendet werden die speichernden Threshold Gatter nach Bild 3.9 (b). Die Steuerung der Speicher gleicht der Steuerung der Mikropipelines (Kapitel 3.1.3). Nur wenn die nächste Stufe ein neues Datum erlaubt (next=1), können die Werte der Eingänge in die Register übernommen werden. Danach müssen diese wieder auf NULL beziehungsweise 0 gesetzt werden. Dies geht wiederum nur, wenn next seinerseits gleich NULL ist. Gleichzeitig wird über das zusätzliche Threshold Gatter am unteren Rand von Bild 3.11 das Steuersignal für die vorhergehende Stufe erzeugt. Es gilt mit n = Anzahl der Signalleitungen:

$$Threshold = \frac{n}{2} \qquad (3.1)$$

Neben diesen Grundbausteinen sind in [22][54][85][100] noch die notwendige Verzweigung und Zusammenführung dargestellt. Somit sind alle notwendigen Bausteine vorhanden, um einen Datenpfad in NCL^{TM} zu erstellen.

Entwurfsverfahren

Für den Entwurf können kommerzielle Entwicklungswerkzeuge und die Hochsprache VHDL verwendet werden [54][85]. Die NCL baut vor allem auf der Unterscheidung zwischen einem gültigen Datum (DATA) und keinem Datum (NULL) auf. Dieser Sachverhalt kann in einer Erweiterung der boole'schen Logik aufgenommen und innerhalb VHDL durch Festlegen eines Mehrwerte-Typs berücksichtigt werden:

```
TYPE ncl_ulogic IS ('U','X','0','1','N','Z','-');
SUBTYPE ncl_logic IS resolved ncl_ulogic;
```

Neben diesen Erweiterungen und den Überladungen der Standardoperatoren ist zusätzlich noch eine Prozedur hysteresis notwendig, welche die Speichereigenschaft der NCL-Gatter berücksichtigt. Mit all diesen Erweiterungen ist eine Modellierung und Simulation einer Schaltung mit einer ideellen 3-Werte Logik (0,1,N) möglich. In diesem Zusammenhang wird von einer 3NCL gesprochen, was widerspiegelt, dass noch Signale mit einer Leitung aber 3 Werten benutzt werden.

In der Synthese werden sowohl der Wert 'N' als auch die Prozedur hysteresis ignoriert. Wie in Bild 3.12 dargestellt, wird die VHDL Beschreibung auf Register-Transfer-Ebene (RTL) mit einer generischen Zellenbibliothek unter Einbindung einer Soft-Makro Bibliothek analysiert und auf logischer (Boole) Ebene optimiert. Ergebnis dieses Schrittes ist eine sogenannte 3NCL Netzliste.

Diese Netzliste besteht aus „einfachen" Signalen sowie UND und NICHT Verknüpfungen. Im nächsten Schritt werden die Signale durch Dual Rail kodierte Signale ersetzt. Die Definitionen hierfür sind in einem VHDL Package abgelegt:

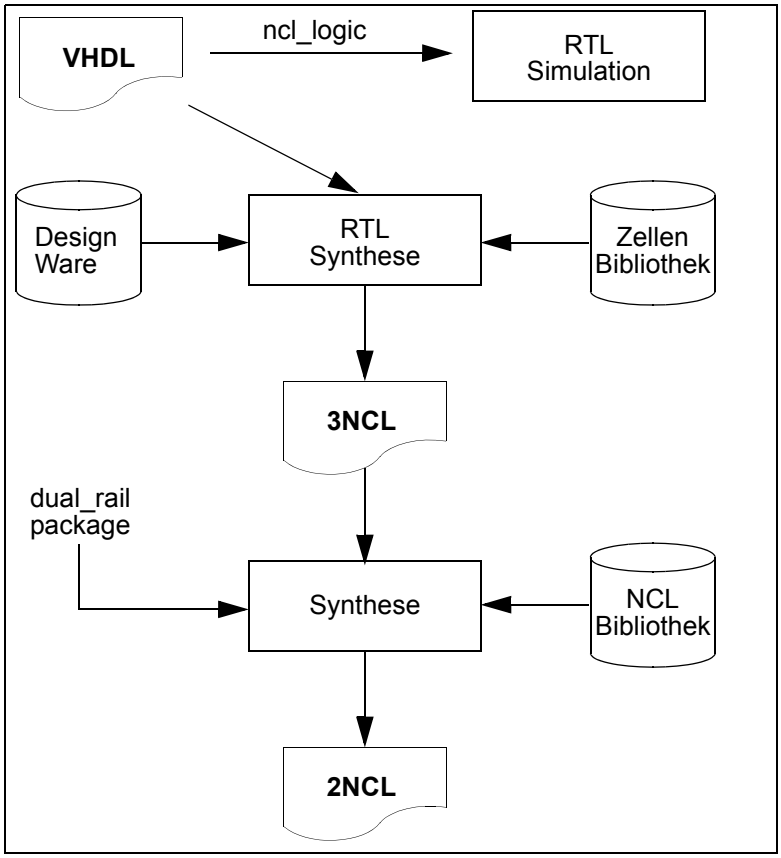


Bild 3.12: RTL Entwurfsablauf für NCL Schaltungen

```
TYPE dual_rail_logic IS RECORD
   rail1 : std_logic;
   rail2 : std_logic;
END RECORD;

FUNCTION "not" (a: dual_rail_logic) RETURN r: dual_rail_logic IS
   r.rail1 <= a.rail0;
   r.rail0 <= a.rail1;
END "not";

FUNCTION "and" (a,b: dual_rail_logic) RETURN r: dual_rail_logic IS
   r.rail1 <= a.rail1 and b.rail1;
   r.rail0 <= (a.rail0 and b.rail0) or (a.rail0 and b.rail1) or
              (a.rail1 and b.rail0);
END "and";
```

Nach dem Ersetzen der Signaldefinitionen wird die Schaltung noch auf die NCL Bibliothek abgebildet. In dieser befinden sich die notwendigen Gatter und Register, welche aus Hysterese behafteten Threshold-Gattern aufgebaut sind.

3.1.2.2 Abgeleitete Verfahren

NCL_D

Basierend auf der Grundidee von NCL wurden weitere Verfahren abgeleitet. Hierzu zählt das von Lighart vorgeschlagene NCL_D [54], welches auf einer verzögerungsinsensitive Minterm Synthese (DIMS [87]) beruht. Hierzu bildet Lighart optimierte Netzwerke auf NAND, NOR und XOR Gatter mit jeweils zwei Eingängen ab. Anschließend wird für jeden Draht eine Dual Rail (Bild 2.6 (b)) Darstellung gewählt und jedes Zwei-Eingänge Gatter in Threshold Gatter überführt. Man kann hierbei noch eine eingeschränkte Optimierung für Threshold Netzwerke durchführen.

Somit liefert dieses Überführungsschema für Gatter und Drähte vor Verzweigungen (vergleiche Verzögerungsunabhängigkeit in 2.4.1) Schaltungen, die automatisch verzögerungsunabhängig sind. Für Drähte nach Verzweigungen muss die Korrektheit anhand von Betrachtungen der Annahmen bezüglich der Verzögerungen überprüft werden.

Die Hauptvorteile von NCL_D sind die Einfachheit des Überführungsschemas und die automatische Überprüfung der Verzögerungsunabhängigkeit während der Implementierung. Allerdings entsteht ein erheblicher Überschuss an Hardware. Dies liegt einerseits an der lokal sehr eingeschränkten Sicherstellung der Verzögerungsunabhängigkeit; es ist kein gemeinsames Benutzen von Komplettierungssignalen erlaubt. Andererseits liegt es auch an der Tatsache, dass fast keine Optimierungen durchgeführt werden können.

NCL_X

Kondratyev und Lwin stellen in [40] eine weitere Variante des NCL Prinzips vor. Ausgehend von einem vorliegenden Netzwerk (zum Beispiel nach der Synthese eines RTL Modells) werden dessen Eingänge (x) durch Paare für die normalen und negierten Eingänge (x.0, x.1) ersetzt. Dadurch erhält man ein so genanntes *unates*[1] Netzwerk, welches keine negierenden Gatter besitzt und den Eins-Zweig der Ausgangsfunktion darstellt. Anschließend wird eine Dual Rail Implementierung erarbeitet, in dem das Netzwerk des Eins-Zweiges dupliziert und die Gatter durch duale Gatter ersetzt werden (OR -> AND, AND -> OR, ...). Anschließend fügt man Komplettierungsdetektoren (Oder-Gatter) für jedes Paar von dualen Gattern ein, deren Ausgänge in ein weiteres Komplettierungsdetektionsnetz eingehen.

Bei NCL_X trennt man also strikt den Funktionspart vom Komplettierungspart. Somit soll eine bessere Optimierung erreicht werden. Der Vergleich mit NCL_D (Tabelle 1 in [40]) zeigt tatsächlich eine deutliche Verbesserung bei dem Flächenbedarf. Der Energieverbrauch ist bei beiden Verfahren in etwa gleich. Über die Performanz werden leider keine Aussagen getroffen.

1. Der Wert einer unaten Funktion (Netzwerk) hängt nur von einem Typ Literal der jeweiligen Variablen (Eingängen) ab, aber nicht von beiden (binate). $f_{unate} = \bar{x}y + yz$; $f_{binate} = \bar{x}y + \bar{y}z$

NCL_X benötigt wie alle anderen NCL Varianten eine eigene Zellenbibliothek. Die ist ein deutlicher Nachteil bei der Integration in den synchronen Entwurfsablauf. Leider werden in [40] keine Vergleiche mit dem ursprünglichen NCL Verfahren gemacht. Somit lässt sich NCL_X nicht komplett bewerten.

Zusammenfassend ist zu sagen, dass bei asynchronen Wave Pipelines ebenso wie bei NCL die Prinzipien asynchroner Schaltungennachnachd auf die jeweils niedrigste mögliche Ebene transferiert wurden. Bei NCL ist das bis auf das einzelne Gatter abgebildet und bei asynchronen Wave Pipelines auf die Transistorebene.

3.1.3 Mikropipelines

Mikropipelines [89] nehmen eine Sonderstellung innerhalb der Verzögerungsmodelle ein. Das Prinzip von Mikropipelines basiert auf einer Verknüpfung von bounded delay-Schaltungen, dem Datenpfad, und delayinsensitiven Schaltungen, den Kontrollpfaden. Die Daten werden wie bei einem FIFO (First In First Out) durchgeschoben. Dies passiert unter der Kontrolle von Handshake-Signalen.

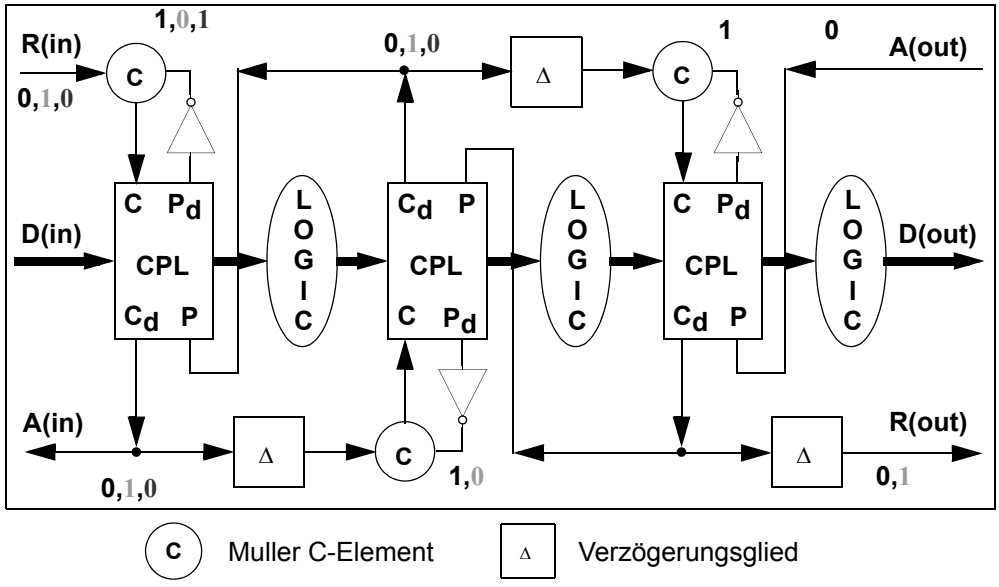


Bild 3.13: Prinzipschaltung einer Mikropipeline

Der Datenpfad ist eine bounded delay-Schaltung und damit sind hier die Standardverfahren des synchronen Entwurfes anwendbar. Bild 3.13 zeigt die Prinzipschaltung einer Mikropipeline. Der Datenpfad ist in der Mitte eingezeichnet. Wird dieser Datenpfad mit einem Standardsynthesewerkzeug erzeugt, so tauscht man anschließend nur mehr die erzeugten Flipflops durch die in Bild 3.13 gezeigten Capture-Pass-Latches aus.

Ein Capture-Pass-Latch wird durch die beiden Flanken zweier Steuersignale kontrolliert (Bild 3.14). Erscheint am Captureeingang eine Flanke, so wird der Latchausgang gehalten und damit das Datum eingefroren (keine Schattierung). Mit einer nachfolgenden Wertänderung des

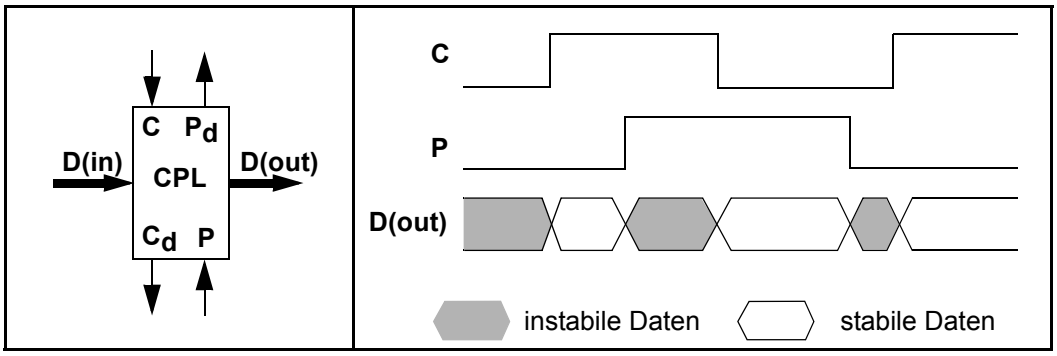
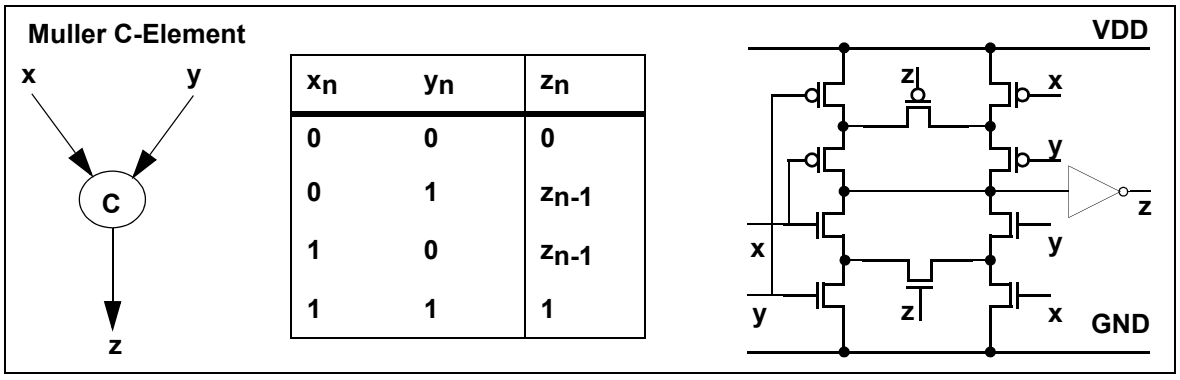


Bild 3.14: Funktionsprinzip eines Capture-Pass-Latches (CPL)

Passsignals wird das Latch wieder transparent und die Daten können durch das Latch propagieren. In dem Timingdiagramm von Bild 3.14 wird deutlich, dass beide Sorten von Flanken zu dem entsprechenden Einfrieren beziehungsweise Durchlassen der Daten führen. Beide Kontrollsignale werden verzögert (C_d, P_d) wieder nach außen geführt, um der vorhergehenden beziehungsweise nachgeschalteten Stufe die entsprechend durchgeführte Aktion anzuzeigen.



x_n	y_n	z_n
0	0	0
0	1	z_{n-1}
1	0	z_{n-1}
1	1	1

Bild 3.15: Funktionsprinzip und statische Realisierung eines Muller C-Elements

Der Kontrollpfad ist eine delayinsensitive Schaltung, die den Transport der Daten steuert. Benötigt werden hierfür Muller C-Elemente (Bild 3.15). Der Ausgang (z) eines Muller C-Elements nimmt nur dann ein neues Datum an, wenn beide Eingänge (x,y) den gleichen Wert einnehmen. In Bild 3.13 erkennt man das Prinzip der Datenflusskontrolle: Zu einem Zeitpunkt sind alle Leitungen, bis auf die Inverterausgänge mit ‘0‘ belegt. Der Sender zeigt durch einen Wertewechsel von R(in) einen Request an. Der Ausgang des ersten C-Elements wechselt auf den Wert ‘1‘. Der Sender bekommt über die Rückführung (A(in)) mitgeteilt, dass sein Request akzeptiert (acknowledged) worden ist. Gleichzeitig wandert die Wertänderung bis zu R(out) weiter. Eine weiter Wertänderung des Senders auf der Requestleitung (R(in)) kann nun nur bis zum letzten C-Element durchkommen; die nächste nur bis zum vorletzten C-Element usw. Kommt ein Request gar nicht mehr durch, so wird der Sender dadurch angehalten. Gibt der Empfänger über A(out) bekannt, dass es weitergehen kann, dann wandern alle Wertänderungen um eine Stufe weiter. Mit jeder Wertänderung von A(out) können also die Requests eine Stufe weiter wandern.

Somit stellt die dargestellte Prinzipschaltung einen dynamischen Puffer dar, der zur generellen Berechnung von Daten in einer asynchronen Schaltung genutzt werden kann. Da der Datenpfad eine bounded delay-Schaltung ist, müssen in den Kontrollpfad Verzögerungselemente eingefügt werden. Der Wert der Verzögerung muss nun mindestens so groß sein, wie die längste Laufzeit durch die entsprechende Logik zwischen den CP-Latches. Somit ist klar, dass die maximale Datenrate durch diese Mikropipeline durch die maximalen Laufzeiten durch die Logik bestimmt werden. Damit sind mit dieser Realisierungsvariante nur in besonderen Fällen Verbesserungen gegenüber der Worst Case Performanz von synchronen Schaltungen zu erwarten.

Entwurfsverfahren

Bei Mikropipelines können Daten- und Kontrollpfad getrennt entworfen werden. Benutzt man Verzögerungsglieder im Kontrollpfad, um die Laufzeit durch die Logik zu kompensieren, kann der Entwurf des Logikblocks in einem gewohnten synchronen Ablauf erfolgen:

Das Modul kann in einer gängigen Hardwarebeschreibungssprache erstellt und mit einer gängigen Software synthetisiert werden. In der Synthese können beliebig Constraints gesetzt werden, um die Laufzeit durch den Logikblock oder die benötigte Fläche zu minimieren. Es muss im Speziellen keine Rücksicht auf Hazardfreiheit genommen werden.

Steht das Syntheseergebnis des Logikblocks fest, so kann die längste Laufzeit durch diesen Block und damit auch die benötigte Verzögerung des Anfragesignals (vergleiche Bild 2.2) bestimmt werden. Diese Verzögerungsglieder müssen separat synthetisiert und für spätere Syntheseschritte gesperrt werden. Ansonsten werden die Verzögerungsglieder von der Standardsynthesesoftware wegoptimiert, da diese keine logischen Operationen durchführen und aus der Sicht der Optimierungsalgorithmen damit überflüssig sind.

Wurden bei der Modellierung des Datenpfades Speicher implementiert, (zumeist Flipflops), so sind diese durch den geeignetsten Speichertyp (zum Beispiel transparente Latches) zu tauschen.

Die Kontrolllogik kann dann auf unterschiedliche Weise entworfen werden. Neben dem Entwurf per Hand, der für größere Designs zu aufwendig ist, bietet sich die Benutzung existierender (universitärer) Synthesesoftware für asynchrone Schaltungen an. Eine mögliche Software ist hier Petrify [20] (vergleiche Kapitel 3.2.3.1). Die Modellierung erfolgt mit STGs, die über mehrere Arbeitsschritte synthetisiert und als VHDL Code ausgegeben werden (Bild 3.28). Das Ergebnis kann anschließend mit Versify (vergleiche 3.2.2.1) überprüft werden.

Abschließend werden Kontroll- und Datenpfad zusammengefügt und können auf Gatterebene durch eine Simulation unter Berücksichtigung der in der Synthese errechneten Verzögerungszeiten (Pre-Layout) überprüft werden.

3.1.4 Speed Independent und Quasi Delay Insensitive Schaltungen

Bei den geschwindigkeitsunabhängigen (speed independent, siehe Kapitel 2.4.1.2) Schaltungen geht man von der Annahme aus, dass die Gatterverzögerungen unbegrenzt, die Leitungsverzögerungen hingegen vernachlässigbar sind. Im Gegensatz dazu wird bei den quasi delay insensitiven Schaltungen (siehe Kapitel 2.4.1.2) die Annahme gemacht, dass sowohl die Gatterverzögerungen als auch die Leitungsverzögerungen unbegrenzt sind. Jedoch kommen *isochronic forks* zum Einsatz, die ein spezielles Modell der Gabelung von Leitungen darstellen. In diesem Modell werden die Unterschiede zwischen den Verzögerungen der gegabelten Leitungen als vernachlässigbar betrachtet, d.h. ähnlich wie in Taktnetzen sind die Verzögerungen im gesamten Netz gleich. Im Vergleich zu den rein delay insensitiven Schaltungen ergeben sich bei den beiden genannten Modellen mehr Realisierungsmöglichkeiten, jedoch unter Annahmen zu Verzögerungszeiten, die im praktischen Einsatz schwierig implementierbar sein können. Im folgenden Kapitel wird als Beispiel der Entwurfablauf von Petrify [20] gezeigt.

3.1.4.1 STG basierter Entwurf (Petrify)

Wie in Kapitel 2.5.1 erwähnt lassen sich STGs in bestimmte Gruppen mit unterschiedlichen Eigenschaften aufteilen:

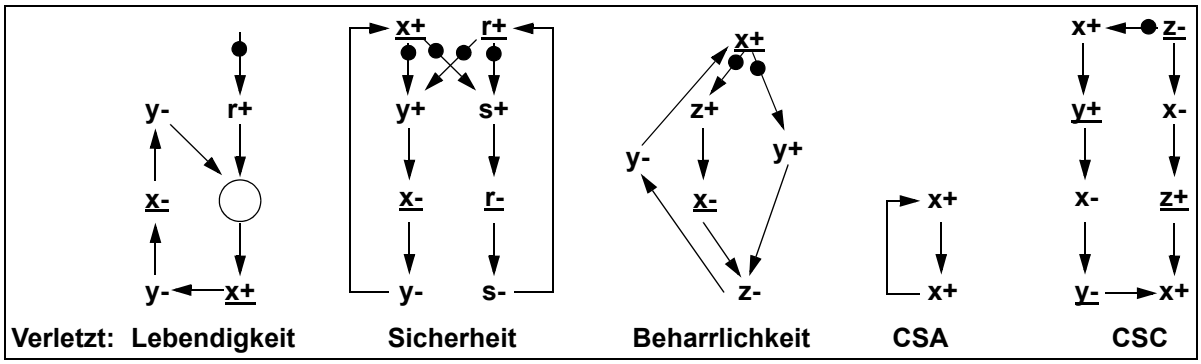


Bild 3.16: Verletzung geforderter Eigenschaften von STGs

Definition 12: ***Lebendigkeit (liveness)***

Jede Transition kann von einer beliebigen erreichbaren Markierung aus unter bestimmten Umständen feuern.

Das erste Beispiel (Bild 3.16) ist nicht lebendig, da die Transition r+ nur einmal feuern kann. Die Lebendigkeit garantiert eine deadlock-freie Schaltung.

Definition 13: ***Sicherheit (Beschränktheit)***

Jeder Platz (gerichtete Kante) kann immer nur einen Marker besitzen.

Das zweite Beispiel ist nicht sicher, da zum Beispiel die Kante von r+ zu y+ zwei Markierungen haben kann, wenn y+ nicht feuert. Die Sicherheit garantiert ein endliches Signalübergangsdiagramm (siehe Entwurfsverfahren).

Definition 14: ***Beharrlichkeit (Persistency)***

Existiert eine Kante von a nach b* (*= + | -), so muss sichergestellt sein, dass ¬a* erst feuert, wenn b* gefeuert hat.*

Das dritte Beispiel ist nicht beharrlich, da eine Kante x+ → y+ existiert und x- feuern kann, bevor y+ gefeuert hat. In [62][63] wird gesagt, dass die Beharrlichkeit eine hazardfreie Schaltung garantiert. In [52] wird aber nachgewiesen, dass es auch nicht-beharrliche STGs gibt, die eine hazardfreie Implementierung besitzen. Des Weiteren wird in [52] belegt, dass die Beharrlichkeit nur bei Verwenden des unbegrenzten Verzögerungsmodells hinreichend für eine hazardfreie Implementierung ist.

Definition 15: ***Konsistente Zustandszuweisung (Consistent State Assignment, CSA)***

Die Transitionen eines Signals müssen zwischen + und - alternieren.

Das vorletzte Beispiel besitzt keine vollständige Zustandskodierung. Solche Graphen können nicht implementiert werden, da es nicht möglich ist, ein bereits angestiegenes Signal erneut ansteigen zu lassen, ohne dass es davor einmal gesunken ist.

Definition 16: ***Vollständige Zustandskodierung (Complete State Coding, CSC)***

Niemals darf bei zwei Markierungen der Wert aller Signale gleich sein.

Im letzten Beispiel ist dies nicht gegeben, da in den Plätzen z-→x+ und y-→x+ die Signale jeweils alle den Wert ‘0‘ besitzen.

Definition 17: ***Einfache Zyklen***

Jedes Signal hat genau einen steigenden und einen fallenden Signalwechsel in dem STG.

Im letzten Beispiel ist dies nicht der Fall, da sowohl x+ als auch x- zweimal enthalten sind.

Um korrekte Schaltungen zu erhalten, müssen die STGs die Eigenschaften nach Definition 12 bis Definition 16 besitzen. Definition 17 (Einfache Zyklen) vereinfacht zwar die Implementierung ist aber nicht zwingend notwendig. Damit entsprechend effektive Algorithmen für die Synthese verwendet werden können, müssen die STGs eine oder mehrere der anderen Eigenschaften besitzen. Es gilt zum Beispiel, dass ein lebendiger und beharrlicher STG in einer laufzeitunabhängigen Schaltung implementiert werden kann.

Entwurfsverfahren

In Bild 3.17 ist ein Beispiel für einen Entwurfsablauf in Anlehnung an Petrify [20] dargestellt. Ausgangspunkt ist entweder ein Signalübergangsgraph (STG) oder ein asynchroner end-

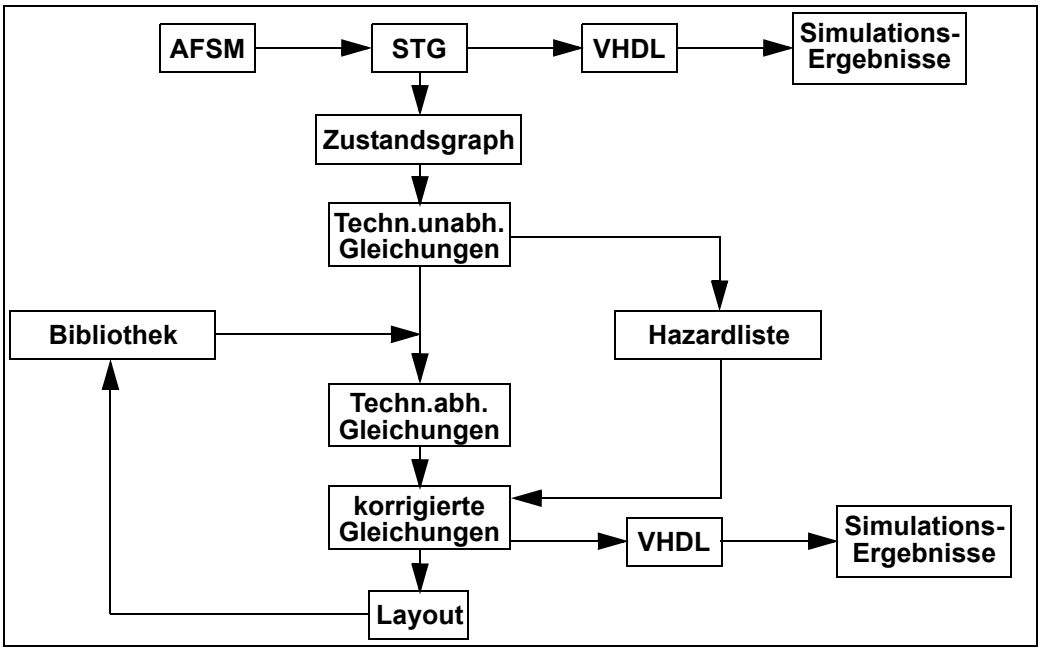


Bild 3.17: Entwurfsablauf mit STGs

licher Automat (AFSM), der in einen STG umgewandelt wird. Der STG kann in ein VHDL Modell übersetzt werden, welches mittels Simulation zur Verifikation des STG verwendet wird.

Der STG wird dann in einen Zustandsgraphen umgesetzt, der durch Optimierungsschritte in einen „Race"-freien Zustandsgraphen gebracht wird. Ein Race entsteht, wenn mehrere Zustandsbits sich ändern. Eines dieser Bits kann sich dabei schneller als die anderen ändern und somit zu einer Fehlfunktion führen.

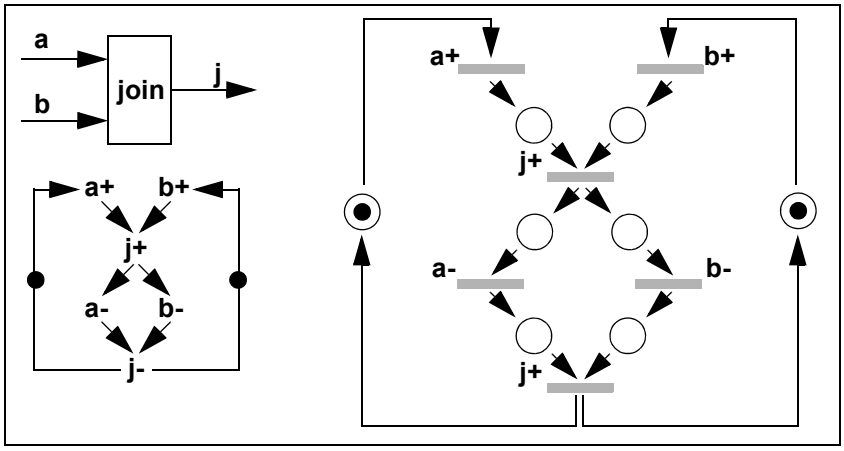


Bild 3.18: STG (links) und Petri-Netz (rechts) eines Join Elementes

Der nächste Schritt ist die Erstellung von Boole'schen Gleichungen (Normalform), wobei hier schon erste Hazardbeseitigungen stattfinden. Es wird eine Liste der noch vorhandenen Hazards erstellt. Nach der eigentlichen Synthese erhält man nun Boole'sche Gleichungen, welche die vorhandenen Bibliothekselemente berücksichtigen. Mit Hilfe der Hazardliste können nun die noch verbleibenden Hazards eliminiert werden. Auch hier kann wieder ein VHDL Modell zur Verifikation der Schaltung verwendet werden.

Letztendlich wird das Layout generiert, aus dem Informationen zum Erstellen neuer Bibliothekselemente oder Verfeinerung der Parameter vorhandener Bibliothekselemente extrahiert werden können.

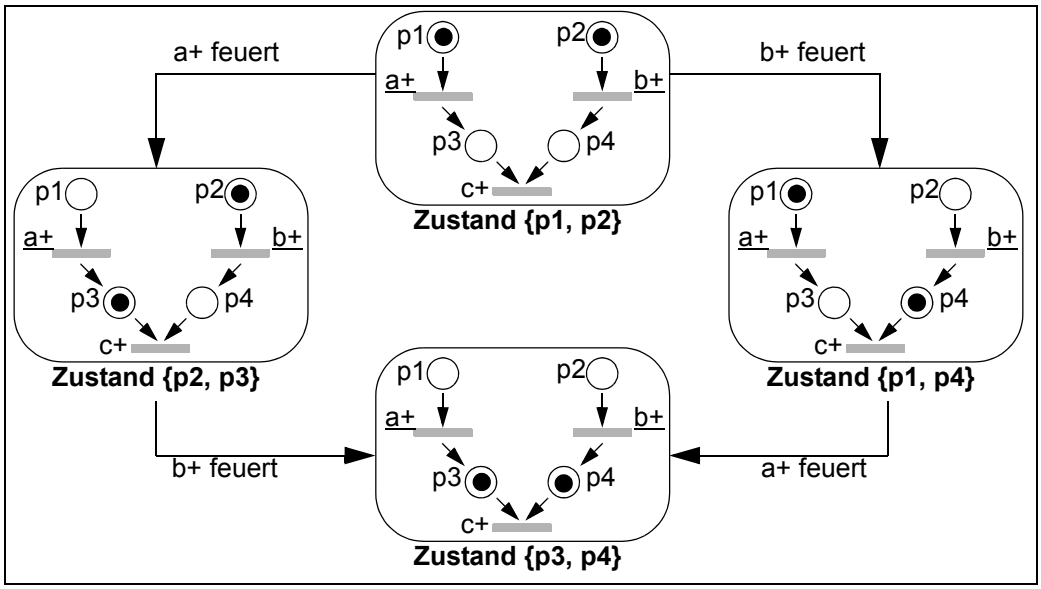


Bild 3.19: Teil des Erreichbarkeitsgraphen des STG aus Bild 3.18.

Der Schritt von einem STG zu den Boole'schen Gleichungen teilt sich in weitere Unterschritte. Aus dem STG wird zunächst ein Petri-Netz erstellt (Bild 3.18). Aus dem Petri-Netz wird mittels Durchlaufen des Graphen der Erreichbarkeitsgraph aufgestellt, wie in Bild 3.19 gezeigt. Ergebnis dieses Schrittes ist der Erreichbarkeitsgraph in Bild 3.20.

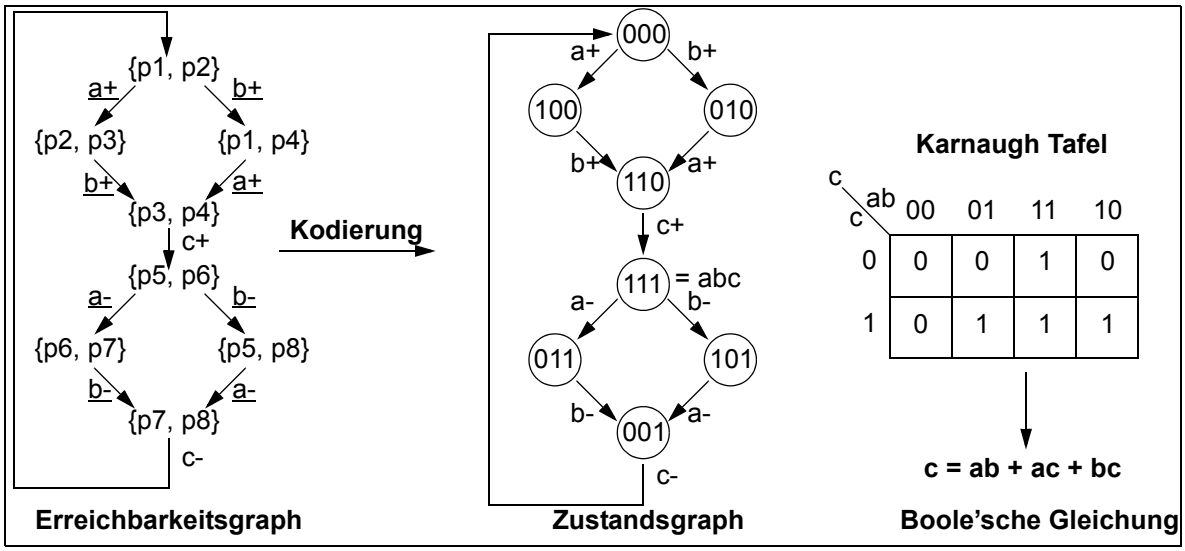


Bild 3.20: Vom Erreichbarkeitsgraphen zur Booleschen Gleichung

Der Erreichbarkeitsgraph wird kodiert, womit man den Zustandsgraph nach Bild 3.20 erhält. Aus dem Zustandsgraph lassen sich sofort die Karnaugh Tafeln aufstellen, aus denen wiederum die Boole'schen Gleichungen hergeleitet werden können, die dann aber hazardfrei sein müssen.

3.2 Werkzeuge für den Entwurf asynchroner Schaltungen

3.2.1 High Level-Beschreibungssprachen

3.2.1.1 Tangram

Tangram [9] ist eine Beschreibungssprache für die Modellierung von VLSI - Schaltungen. Sie wurde von den Philips Forschungslabors entwickelt. Da sie nicht öffentlich zugänglich ist, können keine Aussagen über die Leistungsfähigkeit dieser Sprache gemacht werden.

Die Sprache selbst hat laut den Entwicklern Ähnlichkeit mit der Beschreibungssprache Balsa, welche hier in Kapitel 3.2.1.2 beschrieben wird. Beide setzen auf den Eigenschaften der Communicating Sequential Processes (CSP, [31][13]) auf. Neben den dort festgelegten Mechanismen und Möglichkeiten wurden in Tangram weitere Sprachkonstrukte aufgenommen. Insbesondere ist auch das Einbinden von RAMs, ROMs und Registerfiles möglich.

3.2.1.2 Balsa

Balsa ([7], [8]) wurde im Rahmen des AMULETT3i Projekts entwickelt und ist eine Hochsprache zur Beschreibung asynchroner Schaltungen, speziell von Handshake Komponenten. Zudem wurden Compiler zur Verfügung gestellt (vergleiche Bild 3.22), die es erlauben, ein Balsa Modell zu simulieren und in eine Gatterebenen-Beschreibung zu überführen.

Balsa orientiert sich einerseits an VHDL, andererseits an Tangram. Wie bei Tangram können parametrisierbare Module beschrieben werden. Die Anzahl und Größe der Terminals lassen sich also pro Modul frei festlegen. Allen Modulen ist ein festes Handshake-Protokoll zu Grunde gelegt. Ziel von Balsa ist eine syntaxorientierte Kompilation, die ohne einen Optimierungsschritt auf Gatterebene auskommt.

Bild 3.21 zeigt als Beispiel einen Modulo 10 Zähler. Das Steuersignal aclk lässt den Zähler inkrementieren. Der Zählerstand count wird als Datum nach außen sichtbar gemacht. Der Zähler ist als Prozedur beschrieben. Das Modell besitzt ein ähnliches Verhalten, wie ein Prozess in VHDL. Die Prozedur wird durch die Anfrage an einem Eingang angeworfen und verarbeitet die am Eingang anliegenden Daten (hier der „Kanal“ aclk). Ähnlich wie in VHDL können auch in Balsa Kommandos parallel abgearbeitet werden, solange diese zu keinem Ressourcen-Konflikt führen (VHDL: ungültige Mehrfachzuweisung). Die bedingungslose Schleife sorgt für das gleiche Wiederholungsverhalten in der Simulation wie ein VHDL Prozess.

Die Zählbeschreibung ist umhüllt durch die Abfrage von aclk. Der Ausgang count erhält den Wert von count_reg durch die Anweisung count <- count_reg. Allerdings sind sowohl aclk als auch count Kanäle. Das heißt hier verbergen sich neben den Datenleitungen auch noch die Kon-

```
-- mod10.balsa: async
-- decade counter

import [balsa.types.basic]
public

type C_size is nibble
constant max_count = 9

procedure    mod10    (sync
aclk;
  output count
```

```
local
  variable    count_reg    :
C_size
  variable tmp : C_size
begin
  loop
    select aclk then
      if    count_reg    /=
max_count then
        tmp := (count_reg+1
as C_size)
      else
```

Bild 3.21: Modulo 10 Zähler beschrieben in Balsa [7].

trollsignale, die für ein Handshake notwendig sind. Dies ist eine Eigenheit, die man auch bei Communicating Sequential Processes (CSP, [31][13]) sieht. Allerdings kann in Balsa ein Kanal mit dem Befehl select „offen“ gehalten werden, während das nachgeschaltete Module arbeitet.

Bild 3.22 zeigt, dass der Entwurfsablauf eng an dem synchroner Schaltungen gehalten wird. Der in Balsa geschriebene Quelltext wird mit einem eigens dafür entwickelten Compiler in eine Beschreibung transformiert, die sich Breeze nennt. Diese Beschreibung kann von back-end Werkzeugen gelesen werden. Sie enthält aber auch Prozeduren und Typendefinitionen, welche vom Balsa-Quelltext übernommen werden. Somit kann diese Beschreibung auch als Package für weitere Balsa-Quelltexte verwendet werden.

breeze2lard erzeugt eine Beschreibung, welche von dem Verifikationswerkzeug LARD (siehe 3.2.2.2) verwendet werden kann. Dieser Schritt ist optional. Eine Verifikation rein auf Gatterebene ist ebenfalls denkbar.

Die Breeze-Beschreibung wird mit balsa-netlist in eine Gatterebenen-Beschreibung transferiert. Diese Beschreibung kann dann, wie im gewohnten Entwurfsablauf synchroner Schaltungen simuliert und in ein Layout überführt werden.

Die Kommunikation über die Kanäle erfolgt mit gebündelten Daten, welche also Anfrage- und Bestätigungssignale benötigen. Diese werden mit Hilfe von Verzögerungselementen an die Laufzeit der Daten angeglichen. Somit wird eine Verifikation des Zeitverhaltens der Schaltung notwendig, welche nicht in Bild 3.22 eingezeichnet ist.

3.2.1.3 VHDL

Das ursprüngliche Anwendungsziel von VHDL [35], [30] war die Modellierung und Dokumentation von digitalen Hardware-Entwürfen. Im Laufe der Entwicklung der Sprache wurde aber die Simulierbarkeit und die Möglichkeit der automatisierten Weiterentwicklung (Synthese) berücksichtigt. Die Sprache hat sich in Europa und Amerika neben Verilog [34] zu der

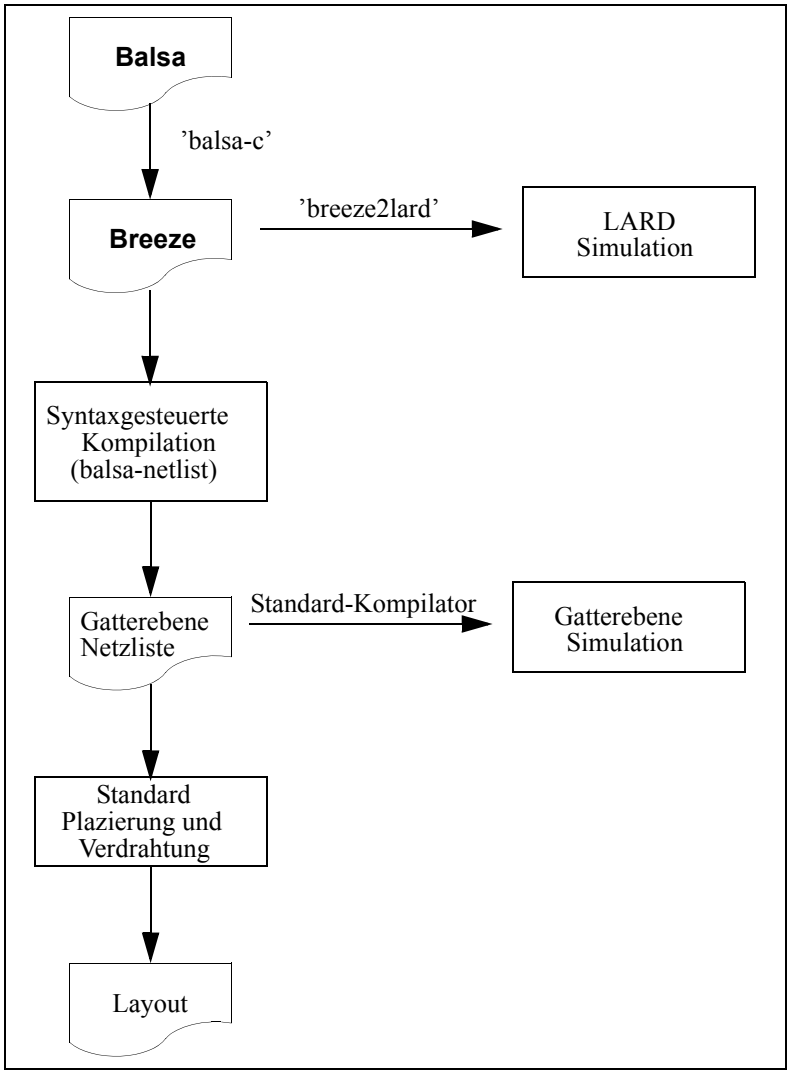


Bild 3.22: Balsa Entwurfsablauf.

am häufigsten verwendeten Hardwarebeschreibungssprache durchgesetzt. Nicht zuletzt aus diesem Grund wird diese Sprache für die Integration des asynchronen in den synchronen Entwurf ausgewählt und hier kurz beschrieben. Auch andere Ansätze ([88], [95], [40]) benutzen die Hochsprache VHDL und die zur Verfügung stehenden kommerziellen Entwurfswerkzeuge.

In VHDL existieren sowohl nebenläufige als auch sequentielle Anweisungen. Beide sind ereignisgesteuert. Das heißt, nur wenn sich die entsprechenden „Eingangs“-Signale ändern, werden die Anweisungen ausgeführt. Eine spezielle nebenläufige Anweisung ist der Prozess (siehe Bild 3.23). Der Prozess als Ganzes ist eine nebenläufige Anweisung, enthält aber selbst eine oder mehrere sequentielle Anweisungen. Zudem kann der Prozess in der Simulation über eine Sensitivitäts-Liste gesteuert werden. Nur wenn ein Signal, das in der Sensitivitäts-Liste aufgeführt ist, seinen Wert ändert, wird der Prozess tatsächlich abgearbeitet.

Der Prozess ist wegen der ihm zu Grunde liegenden Funktionsmechanismen für die Modellierung asynchroner Module geeignet. Weitere Mechanismen von VHDL unterstützen diesen Vorteil. So modelliert man in VHDL hauptsächlich mit Signalen. Ein Signal in einem VHDL

Modell entspricht immer einer oder mehreren Leitungen in dem späteren, nicht optimierten Netzwerk. Neben den Signalen gibt es aber auch noch Variablen. Der Wert einer Variable wird mit einer Zuweisung aktualisiert. Bei einem Signal ist das nicht der Fall. Hier wird der Wert erst dann aktualisiert, wenn die Abarbeitung des Prozesses unterbrochen ist. Somit ist die Assoziation eines Signals mit der erzeugten Hardware leicht nachzuvollziehen, der Aktualisierungsmechanismus erschwert aber bei umfangreicher Funktionalität die Verständlichkeit des Prozesses. Bei gegebenen Fall kann man also auf Variablen zurückgreifen.

Ein weiterer Vorteil von VHDL ist die Möglichkeit, zeitliche Verzögerungen für die anfängliche Simulation mit zu modellieren. Da eine Realisierung einer ganz bestimmten Verzögerung nicht synthetisierbar ist, muss diese modellierte Verzögerung in der Synthese ignoriert werden. Dies ist bei den meisten Synthese-Werkzeugen der Fall. In der Simulation kann man aber schon von Anfang an mit „geschätzten“ Lauf- und Verzögerungszeiten arbeiten und somit die prinzipielle Funktion der Schaltung überprüfen.

```
entity ADDER is
  port(A,B : in   bit;
       CARRY,SUM : out bit);
end ADDER;

architecture RTL of ADDER is
begin

  ADD: process (A,B)
  begin
    SUM   <= A xor B;
    CARRY <= A and B;
  end process ADD;
```

```
entity TB_ADDER is
end TB_ADDER;

architecture TEST of TB_ADDER is

  component ADDER
    port(A, B        : in  bit;
         CARRY, SUM  : out bit);
  end component;

  signal A_I, B_I : bit;
  signal CARRY_I, SUM_I : bit;

begin

  DUT: ADDER port map(A_I, B_I,
                      CARRY_I, SUM_I);

  STIMULUS: process
  begin
    A_I <= ´1´; B_I <= ´0´;
    wait for 10 ns;

    A_I <= ´1´; B_I <= ´1´;
    wait for 10 ns;

    -- and so on ...
```

Bild 3.23: Beispiel eines VHDL Modells plus zugehöriger Testbench [30]

In Bild 3.23 ist auf der linken Seite ein Beispiel für ein VHDL Modell gezeigt. Die ersten vier Zeilen beschreiben die so genannte „entity“. Hier wird die Schnittstelle des Moduls definiert, also alle Eingänge und Ausgänge sowie deren Typ. In den darauffolgenden Zeilen wird

der Funktionskörper, die „architecture“ beschrieben. Hier besteht diese aus nur einem Prozess, in dem die Werte für die Addiererausgänge CARRY und SUM bestimmt werden.

Auf der rechten Seite von Bild 3.23 ist eine mögliche Testbench für das Beispiel der linken Seite gegeben. Hier ist die entity (TB_ADDER) „leer“, das heißt sie besitzt keine Schnittstellensignale. Die Architektur bindet das zu verifizierende Modul über eine so genannte „component“ ein. Diese Komponente wird mit den entsprechend definierten Zwischensignalen (*_I) verknüpft und in der Simulation durch das passende entity/architecture Paar ersetzt. Der Prozess der Testbencharchitektur sorgt einerseits für immer neue Eingangswerte des Prüflings (Signalzuweisungen); andererseits treibt dieser Prozess durch die „wait-for“ Anweisungen die Simulationszeit voran.

Ein wichtiger Vorteil von VHDL ist also, dass sowohl die Modellierung als auch die Simulation in einer Entwicklungsumgebung bewerkstelligt werden. Somit sind auch die Zusammenhänge zwischen Sprachkonstrukten und Simulationsergebnissen einfach zu erkennen. Auch die Synthesewerkzeuge halten sich an die Semantik und Abarbeitungsregeln, die für die Simulation gelten. Somit ist auch hier ein eindeutiger Zusammenhang zwischen Modell, Simulation und Syntheseergebnis gegeben.

Der ASIC-Entwurfsablauf mit VHDL ist in Bild 3.24 gezeigt. Ausgangspunkt ist das in VHDL erstellte Modell auf Register-Transfer-Ebene. Mit einem Standard-Analysierer kann dieses Modell in ein ausführbares und damit simulierbares Objekt kompiliert werden. Sind die Simulationsergebnisse zufriedenstellend, so wird das VHDL Modell mit einem Standard Synthesewerkzeug in eine detailliertere Beschreibung (Modell auf Gatterebene) transferiert. Faktoren bei dieser Umformung sind neben dem VHDL Modell selbst die verwendete Technologiebibliothek, die zur Verfügung stehenden Makrozellen und die gesetzten Optimierungs- und Einsatzbedingungen (constraints).

Das resultierende Modell auf Gatterebene kann wiederum mit der schon existierenden VHDL Testbench auf Korrektheit überprüft werden. Wenn dies der Fall ist, kann das Modell weiterentwickelt werden; mit einem Platzierungs- und Verdrahtungswerkzeug wird dann das zur Herstellung der für die IC-Fertigung benötigten Masken notwendige Layout erzeugt.

3.2.2 Verifikationswerkzeuge

3.2.2.1 Versify

Versify [78] umfasst eine Menge von Programmen, die an der Universität von Katalonien entwickelt wurden. Versify benutzt die gleichen Daten- und Beschreibungsformate wie das ebenfalls an der Universität von Katalonien entwickelte Synthesetool Petrify (siehe Kapitel 3.2.3.1). Es ist in der Lage sowohl die Eigenschaften einer Spezifikation als auch deren Imple-

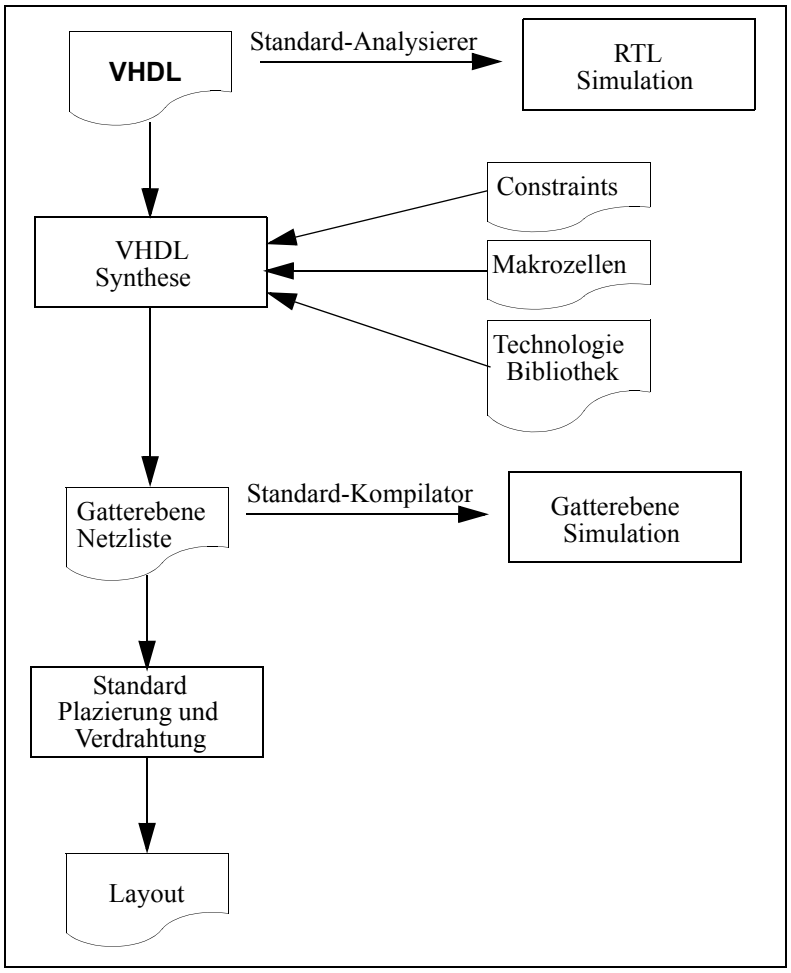


Bild 3.24: VHDL ASIC-Entwurfsablauf.

mentierung zu überprüfen. Im Speziellen überprüft Versify, ob eine Implementierung der Spezifikation entspricht und ob die Implementierung verzögerungsunabhängig ist.

Eine Implementierung wird von Versify als korrekt betrachtet, wenn sie kein unerwartetes Verhalten auf vorgeschriebene Stimuli der späteren Einsatzumgebung und das durch die Spezifikation (STG) vorgegebene Verhalten zeigt.

3.2.2.2 LARD

Bei LARD (Language for Asynchronous Research and Development) [50] handelt es sich um eine Hardware Beschreibungssprache und Simulationsumgebung, die in dem Rahmen des AMULET3 Projekts entwickelt und angewendet worden ist. LARD wurde extra für den Einsatz im asynchronen Schaltungsentwurf entwickelt. Es handelt sich, genauso wie bei VHDL, um eine Hochsprache, die eine Vereinfachung bei der Entwicklung von asynchronen Schaltungen bringen soll.

Bei der Entwicklung des AMULET3 Prozessors wollte man eine Möglichkeit schaffen, den Prozessor auf Verhaltensebene zu modellieren, damit Simulationsexperimente mit verschiede-

nen Architekturen sehr leicht durchgeführt werden konnten. Aus dieser Problemstellung heraus wurde die Hardware Beschreibungssprache LARD mit folgenden Eigenschaften entwickelt:

- Ein einfacher Aufbau der Sprachsyntax, um kurze Einarbeitungszeiten zu erhalten. Damit können auch sehr schnell Ergebnisse erzielt werden.

`/* ... */`	Kommentar, vergleichbar mit der Syntax in C.
`func_1 ; func_2`	Sequentielles Abarbeiten, zuerst *func_1* dann *func_2*.
`func_1 | func_2`	Paralleles Abarbeiten von *func_1* und *func_2*.
`n:var(int).`	Deklaration einer Variable; hier vom Typ *integer*.
`array:var(int^10).`	Felder werden aus 'Typ^Breite' gebildet.
`byte:type=(bit^8).`	Deklaration eines Typen (Bitvektor der Breite 8).
`coord:type=record (x::int,y::int).`	Deklaration eines eigenen, zusammengesetzten Typs.
`n := 27;`	Zuweisung an eine Variable.
`array[5] := 42;`	Zuweisung an ein Element eines Feldes.
`AR := [0,7,43,b+1];`	Vollständige Zuweisung an ein Feld.
`coord -> x := 53;`	Zuweisung an ein Element eines Records.
`forever(...);`	Endlosschleife.
`wait_until(expr);`	Unterbrechung des Prozesses, bis *expr* wahr ist.
`wait_for(t);`	Unterbrechung des Prozesses für *t* Zeiteinheiten.
`chan, init_chan(input), chan_peek(input) chan_ready(input), select(in1, in2, ...)`	Vordefinierte Typen und Funktionen für die Kanalkommunikation.

Bild 3.25: Beispiele für die Syntax von LARD.

- Die 'atomare' Kanalkommunikation, vergleichbar der in Tangram (Kapitel 3.2.1.1) und Occam, ist in der Sprache implementiert. Die Request-Acknowledge Signale und Signalprotokolle nicht mehr explizit modellieren zu müssen, stellt eine wesentliche Vereinfachung bei der Modellbildung dar (Bild 3.26).

	VHDL	LARD

Sendesequenz	`data <= x;`
	`req <= '1';`
	`wait until ack='1';`
	`req <= '0';`
	`wait until ack='0';`
		`data ! x;`

Empfangssequenz	`wait until req='1';`
	`y := data`
	`ack <= '1';`
	`wait until req='0';`
	`ack <= '0';`
		`data ? (`
		`y := ?data`
		`);`

Bild 3.26: Sende- und Empfangssequenz in VHDL und LARD modelliert.

- Programmiertechnische Eigenschaften von Hochsprachen werden unterstützt.

- Hierarchischer Aufbau des Modells ist möglich.
- Die notwendigen Typen werden unterstützt: arrays, records, aufzählende Typen, polymorphische und überladene Funktionen
- Ein Front End zur Bearbeitung des Codes, wie zum Beispiel Debugger, Visualisieren und Ablaufverfolgung, sind verfügbar.
- Das Einrichten von Bibliotheken und ein separates Kompilieren sind möglich.
- Es existiert eine Standardbibliothek für arithmetische und logische Operationen.

Um die Verhaltensmodellierung von asynchronen Systemen zu ermöglichen, unterstützt LARD weiterhin:

- Kanalkommunikation
- Paralleles Abarbeiten von Prozessen
- Modellieren von Zeit durch ein Steuerprogramm

Bei der Entwicklung von LARD wurde besonders viel Wert auf Flexibilität gelegt, um Erweiterungen so schnell wie möglich einbauen zu können. Daraus ergab sich, dass die Implementierung viel mehr wie die einer 'normalen' Programmiersprache, als die eines speziellen Simulationssystems verlief. Der Aufbau der einzelnen Blöcke der LARD Werkzeuge ist in Bild 3.27 zu sehen.

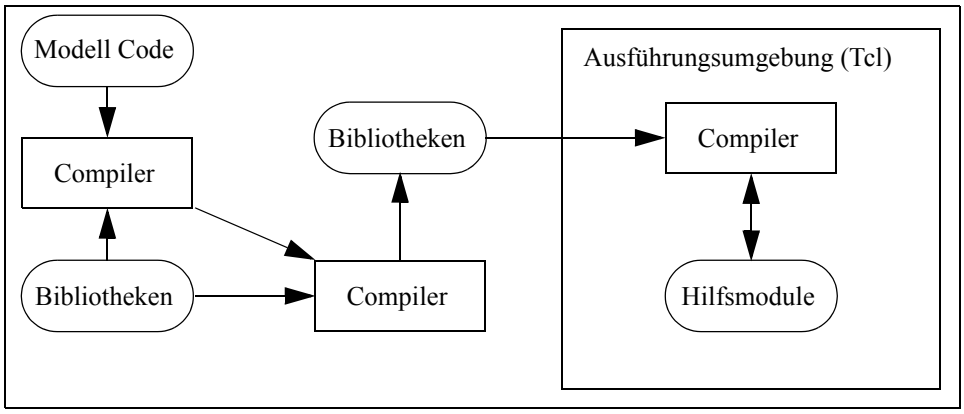


Bild 3.27: Organisation der LARD Werkzeuge.

Die Quellcodedateien werden zusammen mit den eingebundenen Bibliotheken mit Hilfe des Linkers und des Compilers in einen maschinenunabhängigen Bytecode übersetzt. Der Bytecode ist auf einer einfachen Virtual Machine lauffähig, die in den LARD Werkzeugen enthalten ist. Der Übersetzer für die Virtual Machine ist in die Tcl Umgebung eingebettet. Die Simulationen können im Batch-Betrieb, im interaktiven Kommandozeilenmodus oder mit dem graphischen Benutzer-Interface durchgeführt werden.

3.2.3 Synthesewerkzeuge

3.2.3.1 Petrify

Petrify ist ein Entwurfswerkzeug zum Bearbeiten von Petri-Netzen und zur Synthese asynchroner Schaltungen, basierend auf dem STG-Entwurf. Das Programm Petrify wurde von Jordi Cortadella, Michael Kishinevski, Luciano Lavagno, Alex Kondratyev und Alexandre Yakovlev entwickelt [20].

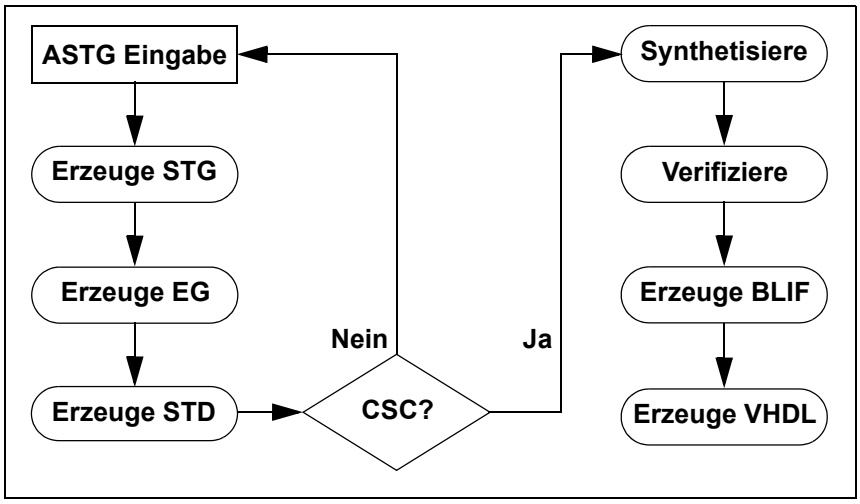


Bild 3.28: Entwurfsprozess mit Petrify

Bild 3.28 zeigt den Entwurfsablauf mit dem Werkzeug Petrify. Zunächst wird ein STG (Signalübergangsgraph, siehe Kapitel 2.5.1) als Textdatei in dem ASTG Format spezifiziert (siehe Bild 3.29 (a)). Bei dem ASTG handelt es sich um ein Text Format das für beliebige markierte Petri-Netze verwendet werden kann.

Aus der ASTG Spezifikation kann eine graphische Darstellung eines STG erzeugt werden (Bild 3.29 (b)). Aus diesem wird in dem nächsten Schritt ein Erreichbarkeitsgraph (EG) erzeugt (Bild 3.29 (c)), welcher anschließend binär kodiert wird. Als Ergebnis erhält man den Zustandübergangsgraphen (STD). Hierbei wird versucht, eine vollständige Kodierung (CSC: Complete State Coding) zu erreichen. Vergleiche hierzu Bild 3.30 (b), bei dem das Signal csc0 hinzugefügt wurde, um einen CSC Konflikt zu beseitigen. Gelingt es nicht, eine vollständige Kodierung zu erreichen, muss der Entwickler per Hand die ursprüngliche ASTG Spezifikation ändern. Ansonsten erzeugt Petrify in einem Syntheseschritt eine Schaltung, die als BLIF-Datei (Berkley Logic Interchange Format) ausgegeben werden kann. Die Synthesealgorithmen sind Weiterentwicklungen aus dem SIS Synthesewerkzeug [51], welche für synchrone Schaltungen entwickelt wurde. Das Syntheseergebnis wird mit Versify (Kapitel 3.2.2.1) verifiziert. Versify [78] ist eng mit Petrify verknüpft, wurde aber an der Technischen Universität von Katalonien entwickelt. Das nach der Synthese vorliegende Modell kann als BLIF Modell ausgegeben werden und seinerseits in ein VHDL Modell oder auch Verilog Modell konvertiert werden.

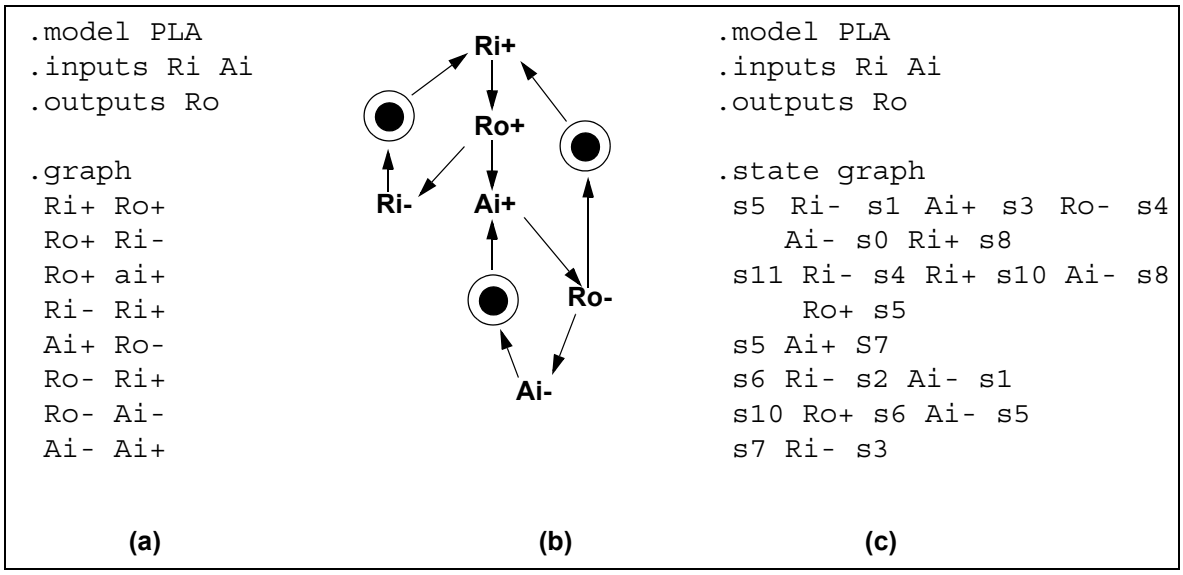


Bild 3.29: ASTG (a), STG (b) und Erreichbarkeitsgraph (c) eines Beispielentwurfes mit Petrify

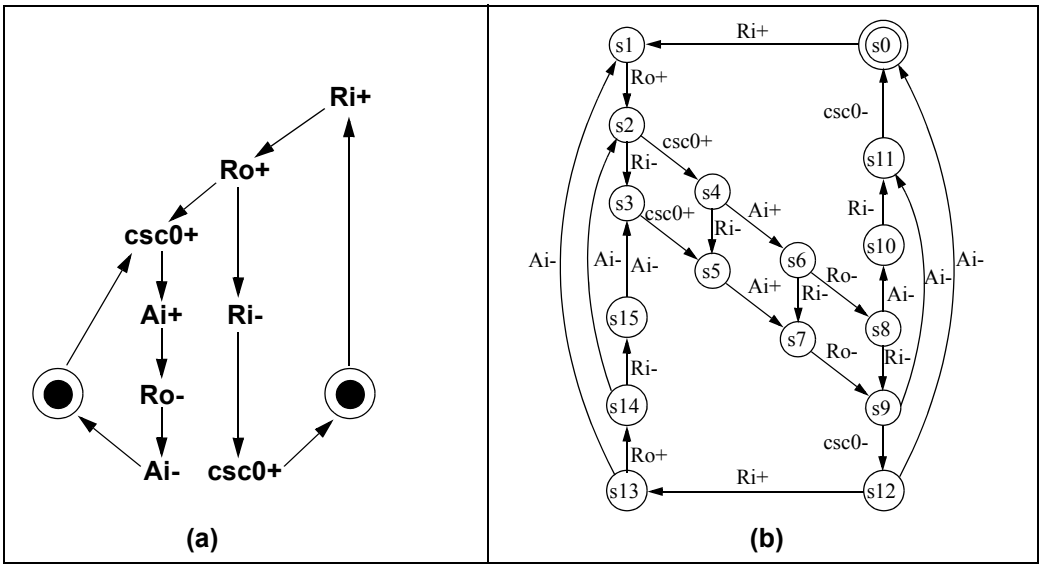


Bild 3.30: STG (a) und STD (b) mit beseitigtem CSC Konflikt

Ist die CSC-Eigenschaft eines STG in Petrify erfüllt, so können entweder logische Gleichungen (EQN-Format) oder eine Netzliste (BLIF-Format), basierend auf einer Standard-Bibliothek, generiert werden. Dabei ist zu beachten, dass in Petrify zahlreiche Synthese-Optionen zur Wahl stehen, wobei die meisten von ihnen das „technology mapping“, d.h. das Einbinden von Gatterbibliotheken, vermeiden und komplexe Gatter und generalisierte C-Elemente bevorzugt verwenden. Man hat jedoch die Möglichkeit, eine Synthese ohne oder mit „technology mapping“ durchzuführen.

Erfolgt keine Abbildung auf eine Gatterbibliothek, so wird als Ergebnis eine Menge an logischen Gleichungen ausgegeben. Anschließend müssen die Gleichungen mit einem herkömmli-

chen (synchronen) Synthesewerkzeug auf eine Technologiebibliothek abgebildet werden. Damit dies erfolgreich durchgeführt werden kann, muss darauf geachtet werden, dass die verwendeten Komplexgatter eventuell in der letztendlich verwendeten Technologiebibliothek fehlen oder vom Synthesewerkzeug nicht ausgewählt werden; in diesen Fällen werden die Komplexgatter durch einfache Gatter nachgebildet, wodurch aber Hazards entstehen können. Der Entwickler muss dafür sorgen, dass keine Hazards erzeugt werden.

Für eine Synthese in Petrify mit „technology mapping“ ist das Vorhandensein einer für asynchrone Schaltung geeigneten Technologiebibliothek in dem GENLIB Format Voraussetzung. Meist muss eine vorhandene Technologiebibliothek erst in dieses Format konvertiert werden. Allerdings stehen Petrify dann mit der „asynchronen“ Technologiebibliothek zusätzliche Informationen zur Verfügung, die zur Optimierung der asynchronen Schaltung und Gewährleistung der Hazardfreiheit herangezogen werden und somit zu einem besseren Ergebnis führen.

3.2.3.2 ATACS

Traditionelle Designmethoden zur Erzeugung asynchroner Schaltungen benutzen oftmals Annahmen über die Verzögerungszeiten der Schaltungen, die zu unnötig ’strengen’ Designs führen. Diese Schaltungen sind zwar verifizierbar, aber vernachlässigen der Einfachheit halber genauere Zeitinformationen. Es werden sehr große Grenzen für die Delays in der Schaltung angegeben; dabei kommt oftmals das ’unbounded delay’ Modell zum Einsatz. In der Industrie handelt es sich aber bei vielen asynchronen Designs um zeitabhängige (timed) Schaltungen, d.h. die korrekte Funktionsweise ist abhängig von der Einhaltung gewisser Zeitvorgaben. Allerdings müssen diese Schaltungen ausgiebig simuliert werden, um die korrekte Arbeitsweise sicherzustellen, da es sich bei den Synthesemethoden oftmals um ad hoc Methoden handelt.

Die *Timed Circuits* werden nun mit diesen Zeitinformationen bei der Spezifikation versehen, die dann bei der Syntheseprozedur zur Optimierung des Designs herangezogen werden. Dadurch können effizientere und zuverlässigere Schaltungen erzeugt werden. Die Timing Analyse wird dadurch sehr komplex. In [67] wurden Algorithmen entwickelt, die dieses Problem minimieren. Weiterhin wurde ein Softwarewerkzeug mit Namen ATACS entworfen, das die Verifikation und Synthese von zeitabhängigen asynchronen Schaltungen vornimmt. Es werden hazardfreie Timed Circuits erzeugt, deren Implementierungen dann auf Semi Custom Gate Libraries abgebildet werden können.

Die Methode basiert auf timed Event Rule (timed ER) Strukturen [67], die aus high level Sprachen wie timed HSE (timed handshaking expansion, Bild 3.31), ähnlich CSP (Communicating Sequential Processes), oder VHDL automatisch generiert werden können. Timed ER Strukturen und Petri-Netze benutzen eine ähnliche Semantik, wobei die timed ER Strukturen eine genauere Syntax haben. Zuerst wird die Spezifikation in einer der Hochsprachen (timed

`/* ... */`	Kommentar, vergleichbar mit der Syntax in C.
`delay DELAYNAME = <10, 20>`	Deklaration eines Verzögerungs-/ Zeitraumes
`input NAME = {true, DELAYNAME}`	Signaldeklaration Es gibt zwei Arten von Übergängen pro Signal.
`s+ , s-`	Sequentielles Abarbeiten
`s+; r-`	Paralleles, nebenläufiges Abarbeiten
`s+ || r-`	Auswählendes Abarbeiten
`s+ | r-`	Auswahlanweisung mit bedingten Anweisungen
`[G1->S1 | G2->S2 | ... | Gn->Sn]`	Wiederholungsanweisung
`*[...]`	

Bild 3.31: Beispiele für die Syntax von HSE.

HSE, VHDL) beschrieben. Es folgt eine Übersetzung von der Hochsprache in die timed Event Rule Strukturen (siehe [67]).

Danach erfolgt eine Timinganalyse, die auf zwei Arten durchgeführt werden kann. Welche Analyse genommen wird, hängt von der timed ER Struktur ab. Das Ziel beider Methoden ist das Auffinden redundanter Regeln beziehungsweise die Reduktion des Zustandsraumes. Eine Regel ist genau dann redundant, wenn das Weglassen dieser Regel keine Änderung des spezifizierten Verhaltens nach sich zieht. Das Resultat ist in beiden Fällen ein reduzierter Zustandsgraph (RSG, Reduced State Graph). Der reduzierte Zustandsgraph ist der Ausgangspunkt für die Synthese.

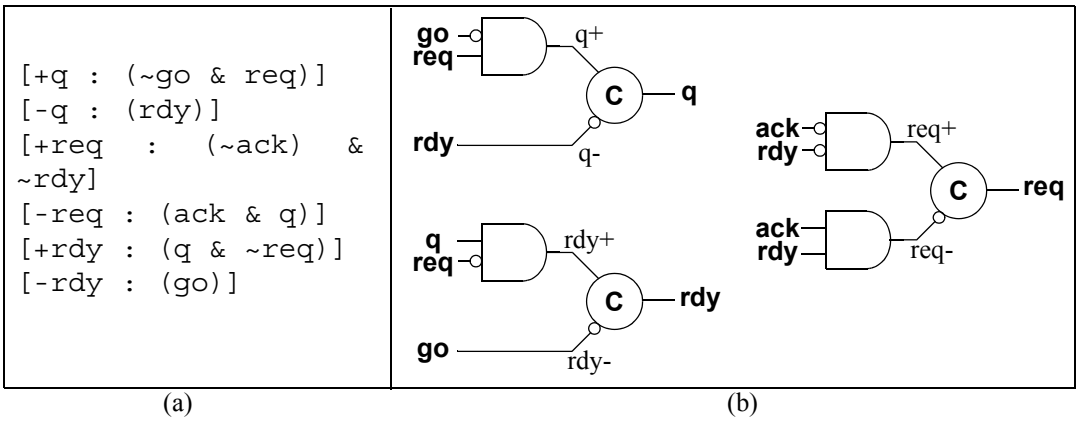


Bild 3.32: Produktionsregeln (a) für den SCSI Controller und deren Implementierungen (b) mit Standard C-Elementen

In der Synthese werden Standard C-Elemente benutzt, wobei wiederum eine Aufteilung in steigende und fallende Übergänge vorgenommen wird. Die graphische Darstellung Bild 3.32(b) auf Gatterebene gleicht der Implementierung mit den verallgemeinerten C-Elementen. Der Unterschied liegt in den Zeitbedingungen, welche die Gatter in den Logikblöcken erfüllen müssen. Beim verallgemeinerten C-Element wurden keine Zeitbedingungen für die Logikblöcke aufgestellt, womit es möglich ist Schaltungen zu implementieren, die auf Gatterebene hazard-

frei sind. Das Ergebnis der Synthese sind die Produktionsregeln (Bild 3.32(a)). Somit steht eine technologieunabhängige Realisierung zur Verfügung.

Zum Schluss muss die Gatterrealisierung noch auf die Gatterbibliothek abgebildet werden. Die Logikgatter haben bis zu diesem Zeitpunkt keine Einschränkung, was die Anzahl der Eingänge betrifft. Um die synthetisierte Schaltung auf eine Semi Custom Gate Library abbilden zu können, ist eine weitere Zerlegung der Logikgatter auf Gatter mit beschränkter Eingangsanzahl notwendig. Die Zerlegung darf keinesfalls Hazards in die Schaltung einbringen, was das Hauptproblem bei diesem Schritt darstellt.

Die Verifikation wird mit Hilfe des Algorithmus zur Timing Analyse durchgeführt. Es wird der Algorithmus für nicht deterministische Strukturen verwendet. Die Grundlage bilden die Spezifikation und das Syntheseergebnis, welche auf Grund des Algorithmus als orbitale Netze vorliegen. Es werden bei der Spezifikation die Ein- und Ausgänge vertauscht und mit dem Syntheseergebnis verbunden. Dann folgt eine Untersuchung des Zustandsraumes mit dem Timing Algorithmus, wobei die Schaltung die Spezifikation erfüllt, wenn keine Fehler auftreten.

3.2.3.3 3D

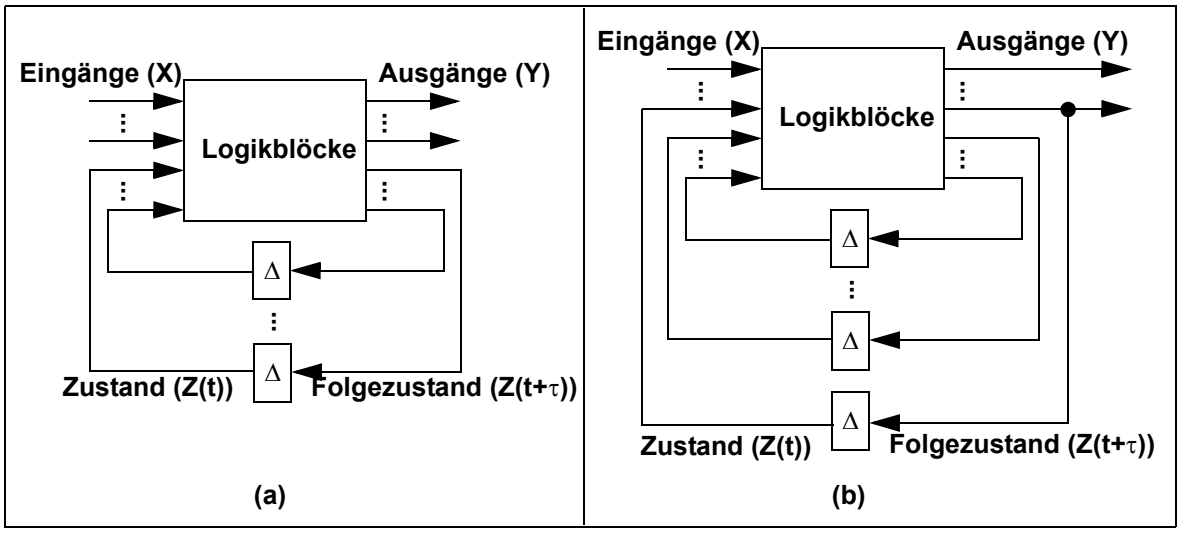


Bild 3.33: Huffman Struktur (a) und 3D Struktur (b) asynchroner Automaten

3D [103], [104] versteht sich als ein Implementierungsstil. Als Ausgangspunkt steht eine Spezifikation eines Kontrollers im „Extended Burst Mode“ (siehe Kapitel 2.5.3). Das besondere an Extended Burst Mode Spezifikationen ist, das mit ihnen auch synchrone (Moore) Automaten beschrieben werden können. Somit bietet sich diese Spezifikationsweise für heterogene Systeme an; als heterogene Systeme werden hier Systeme verstanden, die sowohl synchrone als auch asynchrone Module beinhalten.

Implementierungen von 3D sind ähnlich der der Huffman Schaltungen (siehe Kapitel 3.1.1.1). Nur werden bei den Huffman Schaltungen nur interne Zustandssignale verwendet,

während bei 3D die Ausgangssignale so weit wie möglich als Zustandssignale genutzt werden. Ansonsten besitzt eine 3D Schaltung ebenfalls keine explizite speichernde Elemente und die Logikblöcke müssen frei von Hazards sein.

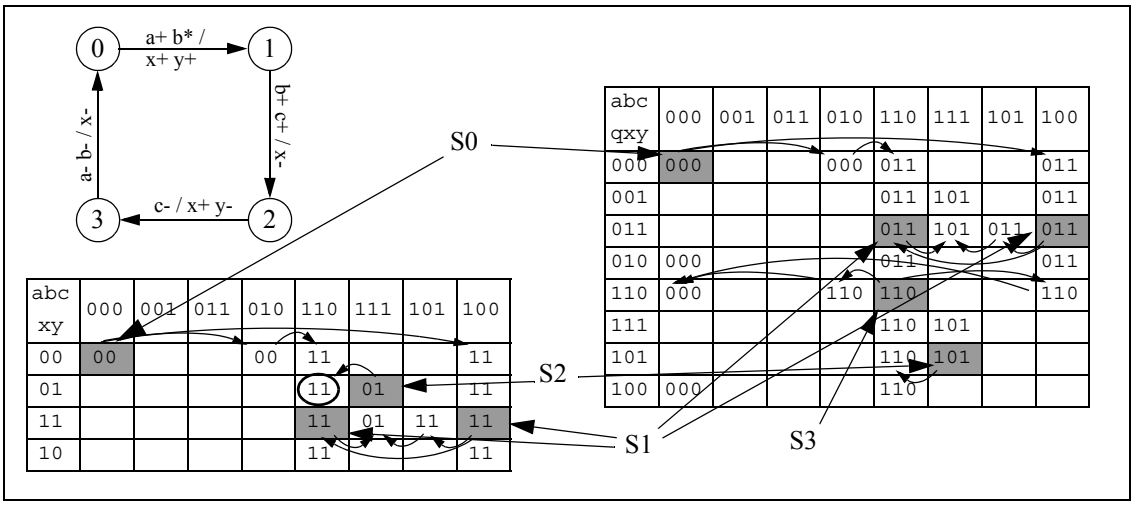


Bild 3.34: Beispiel für Syntheseschritte für eine 3D Struktur

Bild 3.34 zeigt ein Spezifikationsbeispiel und den Vorgang beim Bilden der Folgezustandstabelle. Die linke Tabelle in Bild 3.34 zeigt den ersten Ansatz beim Bilden der Folgezustandstabelle. Vom Startzustand S_0 aus ist der Wechsel des Einganges a entscheidend. Erst wenn dieser erfolgt ist, werden die Ausgänge x und y jeweils auf '1' gesetzt. Falls Eingang b sich vor a geändert hat, erfolgt der Übergang über 010 zu 110. Innerhalb der erreichten Spalte bleibt die Ausgangsbelegung gleich. Haben die Ausgänge ihren neuen Wert eingenommen, befindet man sich im Zustand S_1.

Für den nächsten Zustandswechsel ist der Wechsel b+ und c+ entscheidend. Die Ausgangsbelegungen bleiben wiederum stabil, bis beide Wechsel stattgefunden haben (Spalte 111). Nachdem die Ausgänge dann ihre neuen Werte angenommen haben, befindet man sich im Zustand S_2. Als Nächstes wird auf den Wechsel c- gewartet. Erfolgt dieser, sollte eine Ausgangsbelegung xy=10 erwirkt werden. In der Tabelle ist die entscheidende Stelle aber schon mit der Ausgangsbelegung 11 belegt und es existiert damit ein Konflikt.

Dieser Konflikt wird durch das Hinzufügen eines zusätzlichen Zustandssignals gelöst. Das endgültige Ergebnis ist in der rechten Tabelle in Bild 3.34 gezeigt. Das zusätzliche Zustandssignal wird wie ein Ausgangssignal behandelt, das heißt, es darf sich gleichzeitig zu den Ausgängen ändern. Erst wenn sowohl Ausgangssignale als auch Zustandssignale ihre neuen Werte eingenommen haben, ist der Folgezustand erreicht. So liegen die Zustände S_2 und S_3 nun in der unteren Tabellenhälfte.

Aus der erstellten Folgezustandstabelle kann die entsprechende Schaltung synthetisiert werden. Dabei werden für die einzelnen Ausgangs- und Zustandssignale Karnaugh Tafeln gebildet

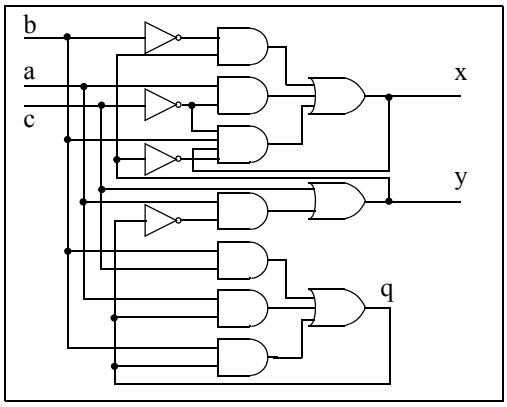


Bild 3.35: Implementierung des Beispiels von Bild 3.34

und aus diesen die passenden (es dürfen keine Hazards enthalten sein) Terme extrahiert werden. Die resultierende Implementierung ist in Bild 3.35 dargestellt.

3.2.3.4 Minimalist

In dem Werkzeug minimalist (Columbia University, New York) wird, ebenso wie bei 3D auf die Vermeidung von Hazards Wert gelegt. Das Tool ermöglicht eine sehr gute Minimierung der Boole'schen Funktionen des Automaten unter Beibehaltung der Hazardfreiheit [24]. Auch mit minimalist können keine Gatter-Netzlisten und damit auch keine elektrischen Schaltungen erzeugt werden. minimalist ist hier auf zusätzliche Werkzeuge angewiesen. Dabei verarbeitet minimalist nur „burst-mode" Automaten. Diese Spezifikationsart ist weniger leistungsfähig als die „extended burst-mode" Spezifikation (Synthesesystem 3D), siehe auch 2.5.3.

4 Entwurfsverfahren für asynchrone Module

4.1 Entwurf von Pipelines

4.1.1 Entwurfsablauf

Der Entwurf der asynchronen Pipelines in einer synchronen Entwurfsumgebung ist in Bild 4.1 grob dargestellt. Ziel ist es, einen gleichartigen Ablauf wie beim rein synchronen Entwurf zu erhalten (vergleiche Bild 4.3).

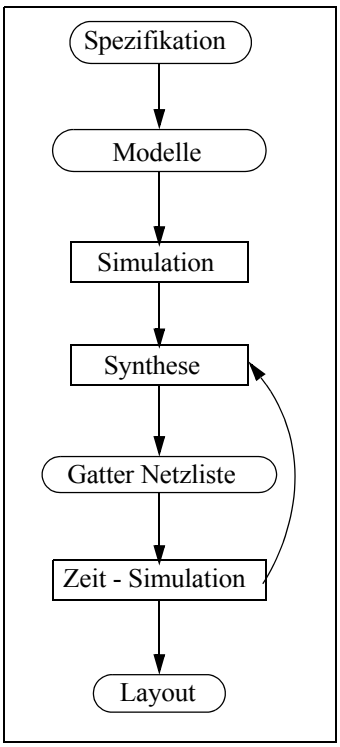


Bild 4.1: Entwurfsfluss von asynchronen Pipelines

Dies ist insoweit erfolgt, als dass prinzipiell die gleichen Arbeitsschritte durchgeführt werden und ebenfalls kommerzielle Software für den Entwurf für synchrone Schaltungen integriert ist. Allerdings sind die Modelle in Bild 4.1 nicht auf (V)HDL Modelle beschränkt und damit die Simulation und Synthese nicht allein auf Standardsoftware beschränkt.

Da das Bounded Delay Modell zu Grunde gelegt wird, muss auf der Gatterebene explizit eine Timing-Simulation durchgeführt werden. Dies ist wegen der asynchronen Kontrollpfade und den darin enthaltenen Verzögerungsketten notwendig. Entsprechend den Simulationsergebnissen sind eventuell Korrekturen (in den Verzögerungspfaden) notwendig, so dass erneut eine Synthese durchgeführt werden muss.

Liefert die Zeit-Simulation zufriedenstellende Ergebnisse, kann die Erzeugung eines Layouts mit geeigneter Standardsoftware erfolgen.

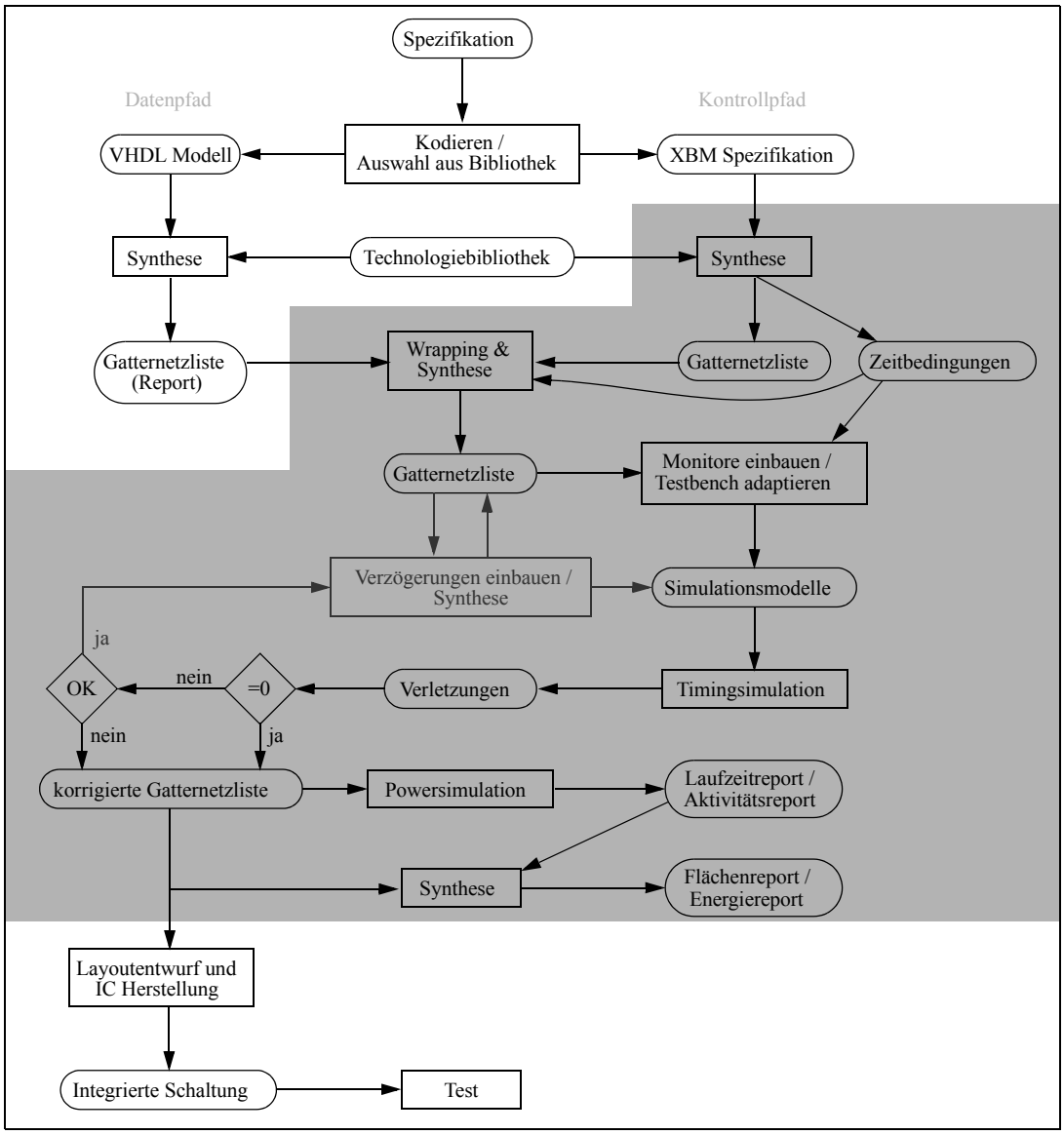


Bild 4.2: Detaillierter Entwurfsfluss von asynchronen Pipelines

In Bild 4.2 ist der erarbeitete Entwurfsfluss genauer dargestellt. Der grau hervorgehobene Bereich wird von der in dieser Arbeit entwickelten Software ASMOGEN (**AS**ynchronous **MO**dul **GEN**erator) abgedeckt.

Ausgehend von einer Spezifikation werden die Modelle geschrieben. Für die in dieser Arbeit vorgestellten Beispiele wurden für den Datenpfad die Hochsprache VHDL (vergleiche Kapitel 3.2.1.3) und für den Kontrollpfad das XBM Format (vergleiche Kapitel 2.5.3) gewählt.

Der erste Schritt des Erstellens der Entwurfsmodelle ist der wichtigste und liegt in der Hand der Entwickler. Deren Erfahrung und Fachwissen spielt bei der Berücksichtigung und Umsetzung der Anforderungen und Randbedingungen eine große Rolle. Gegenüber der sonst den Ent-

wicklern gegebenen Freiheiten beim Entwurf des Datenpfades ist nun allerdings zu berücksichtigen, dass eine strenge Zeitoptimierung zu entsprechend kleineren Verzögerungspfaden und damit zu geringerem Flächenbedarf und Energiebedarf der Kontrollschaltung führt. Somit ist eine Zeitoptimierung vor anderen Optimierungszielen zu bevorzugen. Die Synthese des Datenpfades wurde mit der kommerziellen Synthesesoftware Synopsys [94] durchgeführt und das prinzipielle Verfahren ist in Kapitel 4.1.3 vorgestellt.

In Rahmen dieser Arbeit wurde ein großer Katalog an Kontrollstrukturen angelegt (siehe Kapitel 4.2.3), aus dem die passende ausgewählt werden kann. Allerdings bedarf es auch hier eines entsprechenden Erfahrungsschatzes, um eine geschickte Auswahl zu treffen. Zudem enthält der Katalog nicht alle denkbaren Kontrollstrukturen für jedweden möglichen Einsatzfall. Entsprechend kann und muss der Katalog erweitert werden (ähnlich den sogenannten Softmakro-Bibliotheken).

Für die Synthese des Kontrollpfades wurde das Programm DGC [45] benutzt und in Zusammenarbeit mit dem Autor der Software erweitert. DGC führt folgende Schritte durch:

Überprüfung der Eingabedaten

Der eingegebene Automat muss bestimmten Kriterien genügen, die von DGC überprüft werden. So muss *Eindeutigkeit* vorliegen, das heißt, dass die Schnittmenge von zwei verschiedenen Übergängen, die aus dem selben Zustand herausführen, leer sein muss. Der Automat muss aus *stabilen Zuständen* aufgebaut sein; so darf zum Beispiel ein Zustand nicht mit der Eingangsbelegung als Ausgangsbelegung wieder verlassen werden. Des Weiteren wird von DGC geprüft, dass keine *maskierten Zustände*[1] vorliegen und dass das *Ausgabeverhalten hazardfrei* ist.

Bestimmung einer Zustandskodierung

Damit eine hazardfreie Automatenfunktion bestimmbar ist, muss eine spezielle Zustandskodierung verwendet werden. Die von DGC verwendete *unicode single-transition-time* Kodierung geht auf Arbeiten von James Tracey [96] zurück. Letztendlich werden Bedingungen für die einzelnen Zustandscodes aufgestellt, welche Aussagen darüber machen, welche Zustandscodes von anderen Zustandscodes durch ein Bit in ihrer Kodierung getrennt werden müssen. Die von DGC erstellten Zustandskodierungen bauen auf diesen Grundsätzen auf und in einem weiteren Schritt minimiert DGC die Kodierung.

1. Ein Zustand ist maskiert, wenn ein direkter Übergang zu diesem Zustand durch andere Übergänge asugeschlossen wird.

Aufstellung der Automatenfunktion

Durch die Einschränkung der Spezifikation auf Burst Mode Automaten[1], beziehungsweise der mittlerweile erweiterten Möglichkeit der Extended Burst Mode Automaten, werden Eigenschaften garantiert, die sich zum Aufstellen einer Automatenfunktion benutzen lassen. Es werden Boole'sche Ausdrücke extrahiert, die hinsichtlich Ausgangsverhalten und Berechnung des Folgezustands eine hazardfreie Realisierung ermöglichen.

Technische Umsetzung der Automaten

Der Automat wird prinzipiell in einer Huffmanstruktur implementiert. Da die Schaltung prinzipiell hazardfrei ist, *essentielle Hazards* aber nicht durch reine Logikgatter ausgeschlossen werden können, werden zusätzliche Verzögerungsglieder eingebaut. Diese eleminieren vorhandene essentielle Hazards.

Der Einsatz von DGC und dessen Funktionsweise ist in Kapitel 4.1.4 genauer vorgestellt. Im folgenden Kapitel wird die Funktionsweise von ASMOGEN näher beschrieben.

4.1.2 Funktionsweise von ASMOGEN

ASMOGEN[2] steuert den in Bild 4.2 hervorgehobenen Bereich. Die anfängliche Synthese des Datenpfades ist optional über ASMOGEN steuerbar. Es empfiehlt sich aber, die Synthese des Datenpfades komplett dem Entwickler zu überlassen. Dies hat zwei Gründe: Zum einen hat der Entwickler die direkte Kontrolle über die Synthese; dies kann vor allem bei komplexeren Schaltungen von Vorteil sein. Zum anderen besteht die Möglichkeit, ein bestehendes synchrones Design in ein asynchrones Design umgestalten zu lassen. Auch wenn dies nicht immer zu einer optimalen asynchronen Schaltung führen wird, ist dies eine attraktive Option, um möglichst schnell synchrone und asynchrone Implementierungsalternativen miteinander zu vergleichen.

Wird die Synthese des Datenpfades durch den Entwickler durchgeführt, sollte zusätzlich ein Laufzeitreport erstellt werden, damit dies nicht in einem späteren Schritt im Entwurfsfluss gemacht werden muss.

Der Aufruf des Programms DGC zur Synthese des Kontrollpfades wird von ASMOGEN gesteuert, wobei die Parameter zum Aufruf von ASMOGEN durch den Entwickler festgelegt werden. Hierzu wählt man durch Angabe der Klassenmerkmale der Kontrollschaltung (vergleiche Kapitel 4.2.1) eine bestimmte Klasse aus und durch die Angabe weiterer Optionen die Art (Basisschaltung, erweiterte Kontrollschaltungen). Als Ergebnis der Synthese erhält man die

1. Burst Mode Automaten werden auch als generalized-fundamental-mode Automaten bezeichnet, da sie eine Verallgemeinerung von in fundamental mode arbeitenden „Single Input / Single Output"- Automaten sind.
2. ASynchronous MOdul GENerator

Gatternetzliste der Kontrollschaltung (EDIF Format), sowie Zeitbedingungen für die Eingangssignale. Die Kontrollschaltung wird nur dann neu synthetisiert, wenn nicht eine entsprechende Reportdatei und EDIF-Datei (eingestelltes Ausgabeformat von DGC) vorliegen.

Aus den Gatternetzlisten der ersten Syntheseläufe (Daten- und Kontrollpfad) wird unter Berücksichtigung der Laufzeitreports eine neue Ebene des aktuell bearbeiteten Moduls erstellt. Neben der Verknüpfung des Daten- mit dem Kontrollpfad muss an dieser Stelle noch die Komplettierungsdetektionsschaltung eingebaut werden. In den in dieser Arbeit gezeigten Beispielen wurde hierfür eine Verzögerungskette im Kontrollpfad benutzt. Neben dieser Verzögerungskette müssen eventuell noch andere Verzögerungen eingebaut werden. Vergleichen Sie hierzu Kapitel 4.1.5. Der Einbau zusätzlicher Verzögerungen kann unter Umständen erst in einem zweiten Syntheselauf erfolgen, wenn die vorläufigen Laufzeiten durch die Schaltungen festliegen.

Für die Überprüfung des Zeitverhaltens einer Pipeline mit einer festgelegten Klasse von Kontrollschaltungen sind Templates von Testbenches erstellt worden. Diese Templates werden automatisch angepasst. Hierzu wird ein Platzhalter in der Testbench durch das zu prüfende Design ersetzt und die Stimuli entsprechend den von DGC vorgegebenen Zeitbedingungen angepasst.

Ein weiterer Punkt ist das Einsetzen von sogenannten Monitoren in die Gatternetzliste des zu prüfenden Designs. Hierzu werden die in der Synthese mit DGC erzeugten Zeitbedingungen herangezogen und die entsprechenden Signale im Design identifiziert. Auf die gefundenen Signale werden assertions aufgesetzt. Sie überprüfen jeweils, ob zwischen einer Änderung eines Zustands- oder Ausgangssignals und dem entsprechenden Eingangssignal der geforderte Zeitabstand liegt. Wenn nicht, wird eine entsprechende Fehlermeldung ausgegeben. Diese wird wiederum mit anderen Meldungen in einer Simulationsreport-Datei für eine weitere Bearbeitung gespeichert.

Im nachfolgenden Schritt wird überprüft, ob Verletzungen der vorgegebenen Zeitbedingungen vorliegen. Ist dies der Fall und können diese von ASMOGEN automatisiert korrigiert werden, so werden in den entsprechenden Pfad Verzögerungsglieder eingebaut. Das geänderte Design wird wieder synthetisiert und eine neue Gatternetzliste sowie ein Simulationsmodell erstellt.

Liegen keine beziehungsweise nicht automatisch korrigierbare Verletzungen der Zeitbedingungen vor, so wird eine entsprechende Meldung ausgegeben und das Design für die Powersimulation abgespeichert.

Anschließend wird eine Simulation durchgeführt, bei der die Aktivitätsraten jedes Netzes mitprotokolliert werden. Hierfür sind für den späteren Betrieb typische Stimuli anzulegen. Für die Untersuchung der unterschiedlichen Kontrollstrukturen wurden im Rahmen dieser Arbeit

immer die gleichen Stimuli verwendet. Ein Ergebnis neben den Aktivitätsraten ist die Gesamtlaufzeit, die das untersuchte Design benötigt hat. Da bei asynchronen Schaltungen keine entsprechende Vergleichsgröße, wie der Takt in synchronen Schaltungen vorliegt, kann diese Gesamtlaufzeit für Vergleiche herangezogen werden.

Mit Hilfe der in der Powersimulation gewonnenen Aktivitätsraten kann mit einem Standardprogramm für Verbrauchsabschätzung der voraussichtliche Energiebedarf ermittelt werden. Siehe “Leistungsabschätzung auf Gatter-Ebene” auf Seite 178 im Anhang B.3.

Die Erzeugung eines Layouts ist mit der zuletzt erzeugten Netzliste möglich. Somit ist der komplette Entwurf einer rein asynchronen Schaltung oder einer gemischten (asynchron/synchron) mit dem vorgestellten Verfahren durchführbar.

4.1.3 Entwurf des Datenpfades

Die Entwicklung des Datenpfades beginnt, wie der Standardentwurfsablauf in Bild 4.3 zeigt, mit einer Spezifikation (Funktion und Zeitverhalten). Aus dieser entwickelt man ein synthesegerechtes VHDL Modell. Dieses kann nun simuliert und damit auf Richtigkeit der Funktion geprüft werden.

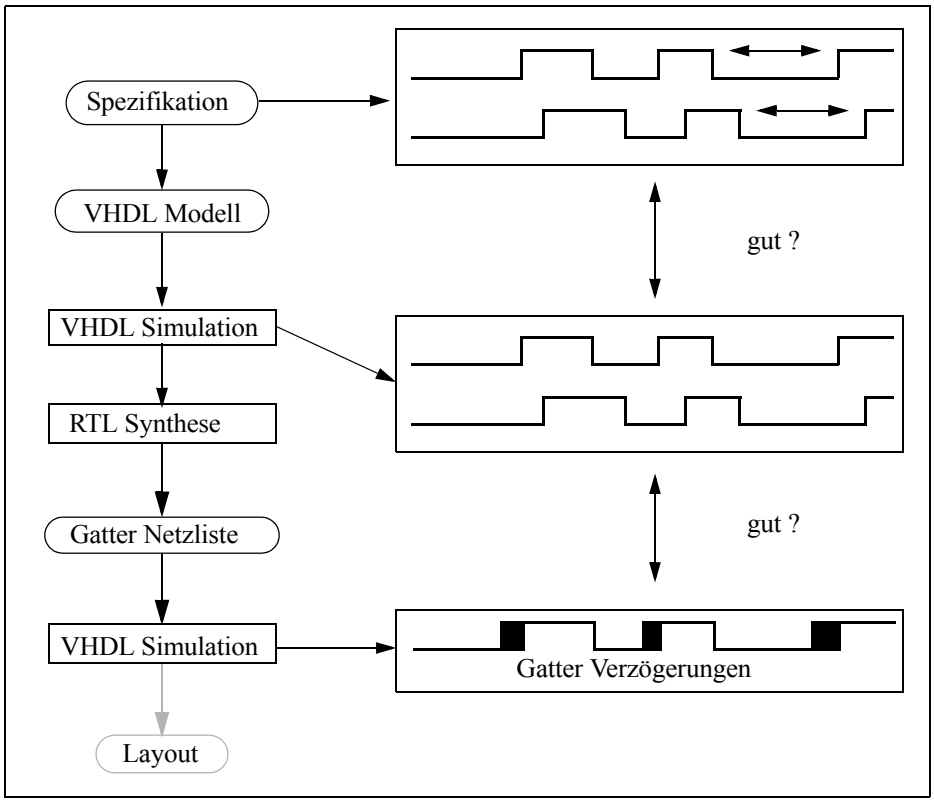


Bild 4.3: Entwurfsablauf einer synchronen digitalen Schaltung

Zeigt das Modell das gewünschte Verhalten, so wird die VHDL Beschreibung synthetisiert (Bild 4.4). Dabei ersetzt ein Synthese-Werkzeug in einem ersten Schritt einzelne Module (Ope-

ratoren, Standardschaltungen) aus der VHDL Beschreibung durch sogenannte Softmakros aus einer entsprechenden Bibliothek. Bei der Auswahl der entsprechenden Softmakros werden die vom Entwickler gesetzten Constraints berücksichtigt. Aus dem so vorverarbeiteten VHDL Code werden Boole'sche Gleichungen extrahiert. Diese werden unter Berücksichtigung der Constraints optimiert und anschließend durch Gatter aus der Technologiebibliothek nachgebildet. Auch hier werden bei der Auswahl der Gatter die gesetzten Constraints berücksichtigt.

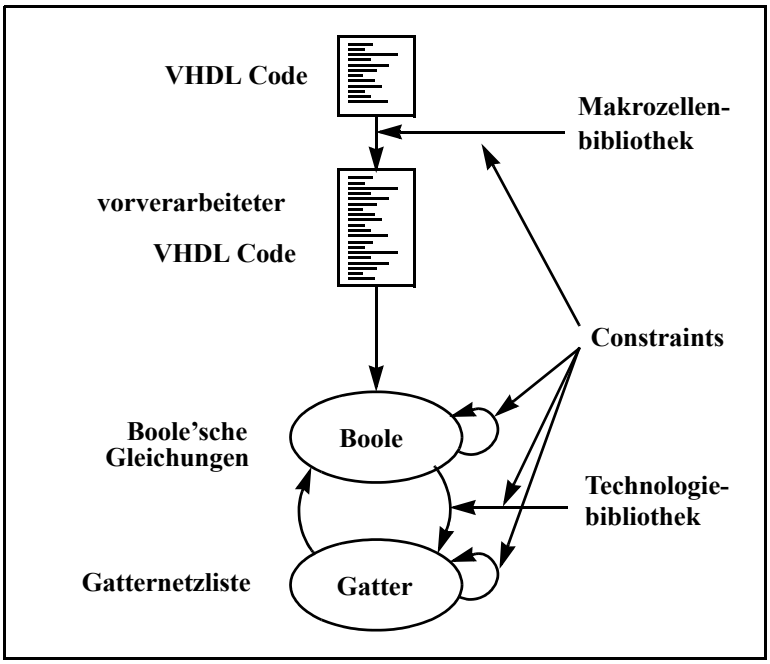


Bild 4.4: Iterative Synthese eines Datenpfades

Auf der Gatterebene findet dann die Technologieoptimierung statt. Aus der so gewonnenen, optimierten Gatternetzliste können wieder Boole'sche Gleichung extrahiert und entsprechend optimiert werden. Das Syntheseprogramm führt die Schritte auf Boole'scher Ebene und Gatterebene so lange aus, bis die Kostenfunktion (erstellt aus den Constraints) nicht mehr zu verbessern ist, oder eine maximale Grenze der Anzahl der „erlaubten" Iterationsläufe erreicht ist. Die optimierte Schaltung speichert man dann in einem für den nächsten Schritt passenden Format (zum Beispiel in VHDL, um eine Simulation durch zu führen).

Wesentliches Constraint bei dem Synthesevorgang ist, dass die Summe der Gatterlaufzeiten der gewählten Gatter auf den längsten Pfaden (von den Eingängen zu den Ausgängen beziehungsweise zu den Eingängen der Speicherelemente) möglichst klein ist. Im Fall eines synchronen Entwurfs ist hierbei die Taktperiode die maßgebende Größe. Ist die Laufzeit über den längsten Pfad deutlich kleiner als die vorgegebene Taktperiode, kann auf andere Kriterien (Fläche, Verbrauch) verstärkt Rücksicht genommen werden. Im Fall eines asynchronen Entwurfs können andere Kriterien als die Laufzeit zum obersten Optimierungsziel gesetzt werden. Benutzt man später allerdings Verzögerungsglieder in den Kontrollpfaden, um die Laufzeit durch die Logik auszugleichen, empfiehlt es sich ebenfalls, die Schaltung auf die Laufzeit hin

zu optimieren. Je weniger Verzögerungsglieder notwendig sind, desto weniger Fläche und Energie benötigt der Kontrollpfad. Somit wird durch die Laufzeitoptimierung des Datenpfades die Schaltung nicht nur schneller, sondern der Flächen- und Energiebedarf der Kontrollschaltung wächst nicht unnötig.

Sobald das VHDL-Modell durch bestimmte Elemente einer Technologiebibliothek dargestellt ist, kann eine Simulation auf Gatterebene durchgeführt werden. Erst jetzt treten Gatter- und Leitungsverzögerungen bei den einzelnen Signalen auf. Die einzelnen Gatter können als verzögerungsbehaftete VHDL-Modelle beschrieben werden. Für die Leitungsverzögerungen werden Schätzwerte berechnet.

Jetzt muss nochmals geprüft werden, ob die Schaltung das vorgegebene Zeitverhalten der Spezifikation einhält. Neben der Laufzeit der Schaltung erhält man nach der Synthese eine Schätzung bezüglich der Größe, Stromaufnahme und anderer Bewertungsgrößen der Schaltung, was wieder mit der Spezifikation verglichen werden kann.

4.1.4 Entwurf des Kontrollpfades mit DGC

Nachdem der Datenpfad eher im klassischen synchronen Verfahren entworfen wird, ist der Entwurf des Kontrollpfades nicht mit herkömmlicher kommerzieller Software zu bewerkstelligen. Am Lehrstuhl für Rechnergestützten Schaltungsentwurf wurde die Synthesesoftware DGC (Digital Gate Compiler [45]) entwickelt, mit deren Hilfe hazardfreie Schaltungen aus einer gegebenen Spezifikation erzeugt werden können. DGC wird im Rahmen dieser Arbeit für den Entwurf des Kontrollpfades benutzt.

Bild 4.5 zeigt den Entwurfsfluss für asynchrone Zustandsautomaten, wie er in DGC durchgeführt wird. Der markierte Bereich darin kennzeichnet den Funktionsumfang von DGC. Aus der Spezifikation werden zunächst die Boole'schen Ausdrücke bestimmt. Die Korrektheit dieser Gleichungen wird durch eine Hazardanalyse sichergestellt.

Daran schließt sich die Umsetzung in eine technologiespezifische Gatternetzliste an. Bestandteil von DGC ist auch eine formale Verifikation dieser Netzliste, also die Berechnung Boole'scher Ausdrücke aus der Gatternetzliste und eine Äquivalenzprüfung mit den ursprünglich vorgegebenen Ausdrücken.

Die Timing-Analyse der Gatternetzliste erzwingt meist das Einfügen von Verzögerungselementen in die Zustandsrückführungsleitungen. Die daraus entstehende korrigierte Netzliste kann mit Hilfe eines externen Gattersimulators überprüft werden. Diese Simulation wird durch die Exportmöglichkeit einer VHDL-Netzliste und einer VHDL-Testbench unterstützt. Verglichen mit einem Entwurfsfluss für synchrone Schaltungen wird hier also die Timing-Analyse durch die vorangegangene Hazardanalyse ergänzt.

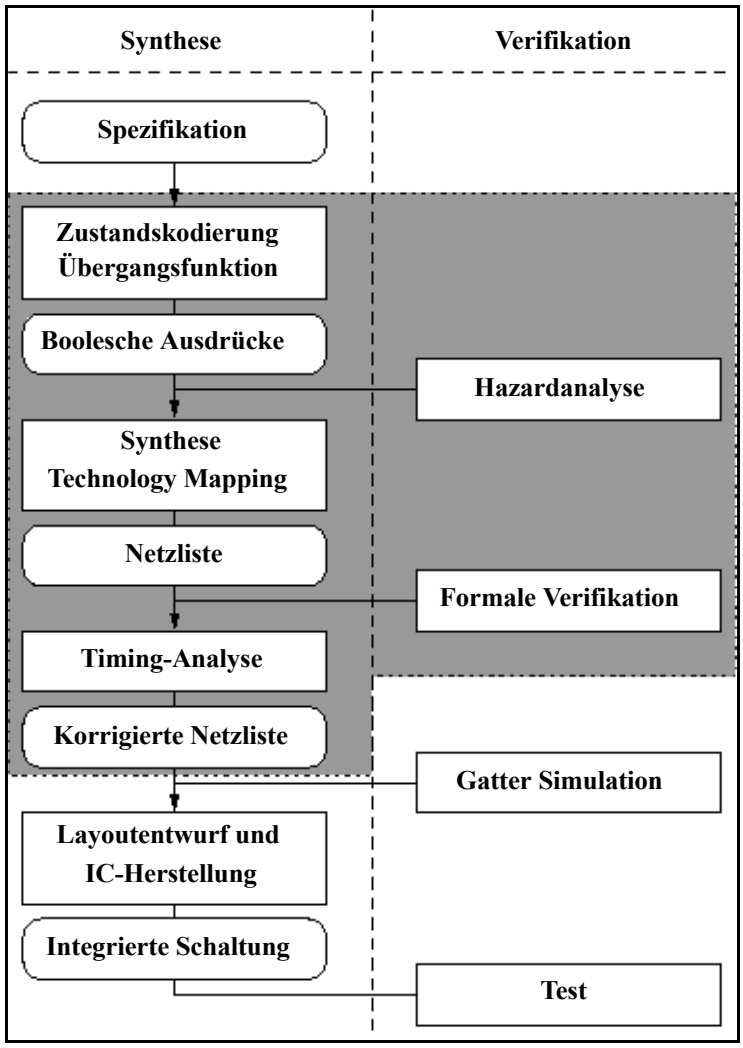


Bild 4.5: Entwurfsfluss DGC

Die Erzeugung eines Layouts ist mit Hilfe geeigneter Designumgebungen möglich. Damit fügt sich das hier vorgestellte Synthesewerkzeug problemlos in einen existierenden Entwurfsfluss für integrierte Schaltungen ein.

DGC grenzt sich von anderen existierenden Synthesesoftware asynchroner Schaltungen folgendermaßen ab (vergleiche Kapitel 3.2.3):

- Bessere Zustandskodierung als 3D und minimalist
- hazardfreies Technology Mapping
- Berechnung und Umsetzung der Verzögerungskette in den Zustandsrückführungsleitungen (inklusive der Ermittlung der essentiellen Hazards)

Die Fähigkeit eine hazardfreie auf die Zieltechnologie abgebildete Implementierung erzeugen zu können, macht DGC für den Einsatz im Entwurf Asynchroner Schaltungen in synchroner Entwurfsumgebung geeignet. Die in DGC verwendeten Algorithmen zur Erzeugung der

Zustandskodierung sind leistungsfähiger als bei den anderen zum Vergleich herangezogenen Programmen 3D und minimalist.

Während der Entstehung dieser Arbeit wurde DGC erweitert, so dass neben den in [45] angegebenen Formaten nun auch das XBM Format als Eingabe verarbeitet werden kann. XBM steht dabei für das „eXtnded Burst Mode“, in dem nicht jeder Signalwechsel streng einem Burst zugeordnet sein muss (vergleiche Kapitel 2.5.3).

Eine weitere Erweiterung ist die Ausgabe von zusätzlichen Zeitbedingungen, abhängig von Ausgangssignaländerungen. Im Normalfall wird die „Fundamental-Zeit“ als Ergebnis ausgegeben, also die maximale Zeit, die die erzeugte Schaltung benötigt, um in einen stabilen Zustand zu kommen. Nun werden die minimal notwendigen „Ruhe“-Zeiten von Eingangssignalen infolge einer Änderung eines Ausgangssignals ausgegeben (vergleiche Kapitel 4.1.5). ASMOGEN verwertet diese Zeitbedingungen einerseits, um die Timing Verifikation in der Gesamtsimulation durch Einfügen von Monitoren zu unterstützen; andererseits werden bei Bedarf (Verletzung der Zeitbedingungen) automatisiert Verzögerungsglieder eingefügt, um die Zeitbedingungen zu erfüllen.

4.1.5 Berücksichtigung von Zeitbedingungen

Ausgleich der Logiklaufzeit

Die Erzeugung des Fertigsignals (**D**one) kann auf unterschiedliche Weise erfolgen. Eine einfache Weise, die von ASMOGEN standardmäßig benutzt wird, ist es, Verzögerungsglieder zu benutzen. Die auftretenden Laufzeiten müssen folgender Gleichung genügen:

$$D_{reg} + D_{\log ic} + T_{setup} \leq T_{Req \to L} \qquad (4.2)$$

Das heißt, die Laufzeit der Anfrage (Request, Req) durch die Kontrolllogik bis zum Speicher (Register) muss mindestens so groß sein, wie die tatsächliche Laufzeit eines neuen Datums durch den Logikblock. Die Laufzeit der Logik ergibt sich als Summe der Laufzeit durch die Speicher D_{reg} (vorgeschaltete Stufe), der Laufzeit durch die Logik D_{logic} und der einzuhaltenden Setup-Zeit T_{setup}.

Die Laufzeit durch die Kontrolllogik unterscheidet sich für request-activated und acknowledge-activated Schaltungen. Bei request-activated Schaltungen ist nur ein Pfad durch die Kontrolllogik zu berücksichtigen, nämlich der vom Fertigsignal zum Speichersignal ($T_{D \to L}$). Bei acknowledge-activated Schaltungen muss zusätzlich neben der Laufzeit zum Speichersignal ($T_{Ain \to L}$) der Pfad vom eingangsseitigen Request-Signal zum eingangsseitigen Acknowledge-Signal berücksichtigt werden ($T_{Rin \to Ain}$). Des Weiteren muss die Berechnung mit schlechtest möglichen Werten erfolgen, um eine größtmögliche Sicherheit zu erhalten. Somit ergeben sich folgende Gleichungen (vergleiche hierzu Bild 4.6):

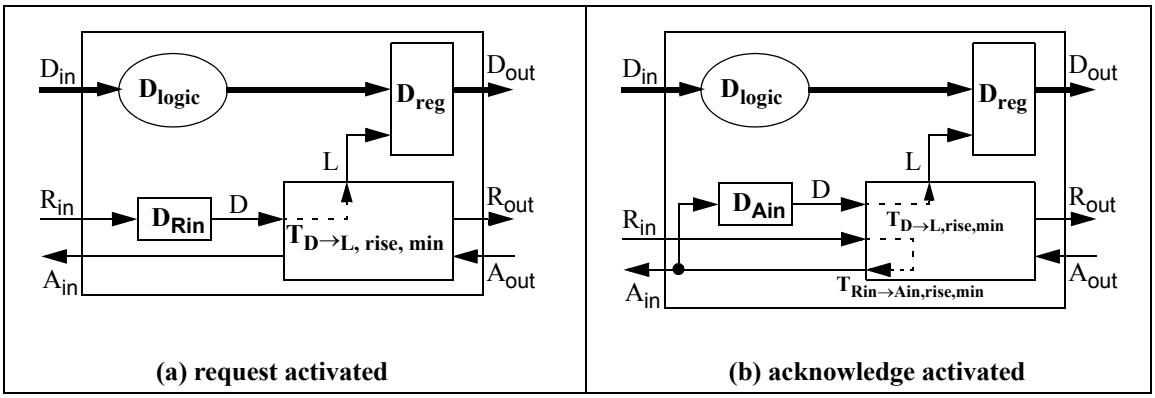


Bild 4.6: Zeitbedingungen bei einer asynchronen Pipeline-Stufe
bei Benutzen von Verzögerungsgliedern

Request-activated:

$$D_{Rin} \geq D_{reg,\,rise} + D_{\log ic,\,max} + T_{setup} - T_{D \to L,\,rise,\,min} \qquad (4.3)$$

Acknowledge-activated:

$$D_{Ain} \geq D_{reg,\,rise} + D_{\log ic,\,max} + T_{setup} - T_{Rin \to Ain,\,rise,\,min} - T_{D \to L,\,rise,\,min} \qquad (4.4)$$

Die Verzögerung (D_{Rin}, D_{Ain}) zum Ausgleich der Logiklaufzeit muss also größer oder gleich der maximalen Ausbreitungszeit des Datums durch die Logik und Speicher minus der minimalen Laufzeiten durch die Kontrolllogik sein.

Die Anzahl der benötigten Verzögerungsglieder ergibt sich dann aus der Division der benötigten Verzögerung durch die Laufzeit durch ein Verzögerungsglied und Aufrundung auf eine ganze Zahl.

Zeitbedingungen des Automaten

DGC gibt generell die Fundamental-Zeit an, das heißt die maximale Zeit, die der Automat nach einer Eingangsänderung benötigt, um in Ruhe zu kommen. Diese Zeit als Bedingung für jeden Eingangswechsel zu verwenden, ist zu restriktiv. Deswegen wurde DGC so erweitert, dass auch einzelne Zeitbedingungen abhängig von stattfindenden Zustandswechseln angegeben werden. Diese Zeitbedingungen werden von ASMOGEN in eigene Dateien gespeichert. In Bild 4.7 werden die von DGC gelieferten Zeitbedingungen für die Kontrollschaltung RSBFF (Kapitel 4.2.3.21) gezeigt. Darin sind zo0 und zo1 interne Zustandssignale, DIN das Fertigsignal, AOUT das ausgangsseitige Acknowledge-Signal und LAT_ROUT die zu einem Signal zusammengefassten Speicher- und ausgangsseitige Request-Signale.

Um ein korrektes Funktionieren auf jeden Fall sicherzustellen, müssen diese Zeitbedingungen eingehalten werden. Das heißt, dass sich zum Beispiel nach einer Änderung des Signals AIN das Signal DIN erst nach einer Zeit von 189,7 ps ändern darf. Für Aout ergibt sich ein Wert

```
Required external delay for      zo0 -->      DIN is       0
Required external delay for      zo0 -->     AOUT is       0
Required external delay for      zo1 -->      DIN is       0
Required external delay for      zo1 -->     AOUT is       0
Required external delay for      AIN -->      DIN is  0.1897
Required external delay for      AIN -->     AOUT is   0.437
Required external delay for LAT_ROUT -->      DIN is  0.2613
Required external delay for LAT_ROUT -->     AOUT is  0.5373
```

Bild 4.7: Zeitbedingungen für die Kontrollschaltung RSBFF

von 437 ps. Man muss in der Synthese also überprüfen, ob diese Zeitbedingungen erfüllt sind oder nicht. Wenn nein, müssen geeignete Änderungen vorgenommen werden.

Es reicht im Normalfall aus, bei Bedarf zusätzliche Verzögerungen in den Pfad einzubauen. Das heißt, sollte zum Beispiel die Schaltung RSBFF im Pfad von AIN nach DIN nicht die in Bild 4.7 geforderte Verzögerung vorhanden sein, muss man zusätzliche Verzögerungsglieder oder Inverterpaare in diesen Pfad einfügen. Dabei muss folgende Gleichung erfüllt werden:

$$D_{Din} \geq T_{DGC,\,Ain \rightarrow Din} - D_{Rin} \qquad (4.5)$$

Die einzufügende Verzögerung ist also der (positive) Wert der Subtraktion der Verzögerung des Request-Signals nach Gleichung *(4.3)* von der geforderten Laufzeit.

Ähnliches gilt für die Verzögerung von LAT_ROUT nach AOUT. Hier muss allerdings die Laufzeit durch die nachfolgende Stufe verwendet werden. Somit kann diese Verzögerung erst angepasst werden, wenn die nachfolgende Stufe einbezogen werden kann, oder die Laufzeit der nachfolgenden Stufe bekannt ist. Dann ergibt sich mit den Laufzeiten der nachfolgenden Stufe als *next* gekennzeichnet:

$$D_{Aout} \geq T_{DGC,\,Rout \rightarrow Aout} - D_{Rin \rightarrow Ain,\,next} \qquad (4.6)$$

Interessant ist hier, dass es bei einer nachfolgenden acknowledge-activated Stufe passieren kann, dass in der aktuellen Stufe entsprechend dem schnelleren Acknowledge Verzögerungsglieder eingefügt werden müssen. Somit verliert man dann hier einen Teil der durch das schnellere Acknowledge gewonnenen Laufzeit (siehe Bild 4.8).

Die verbleibenden von Null verschiedenen Zeitbedingungen sind nicht von vornherein korrigierbar. Diese können erst im Zusammenspiel mit der Vorgängerstufe oder der Nachfolgerstufe überprüft und dann korrigiert werden. Dabei wird die Korrigierbarkeit mit dem Grad der Entkopplung (vergleiche Kapitel 4.2.3) niedriger und die Anzahl der Verzögerungsglieder unter Umständen größer. Sind die Stufen eng gekoppelt, so besteht über die Kopplungsbedingungen zum Beispiel eine indirekte Abhängigkeit zwischen dem Ausgangssignal ROUT und dem Ein-

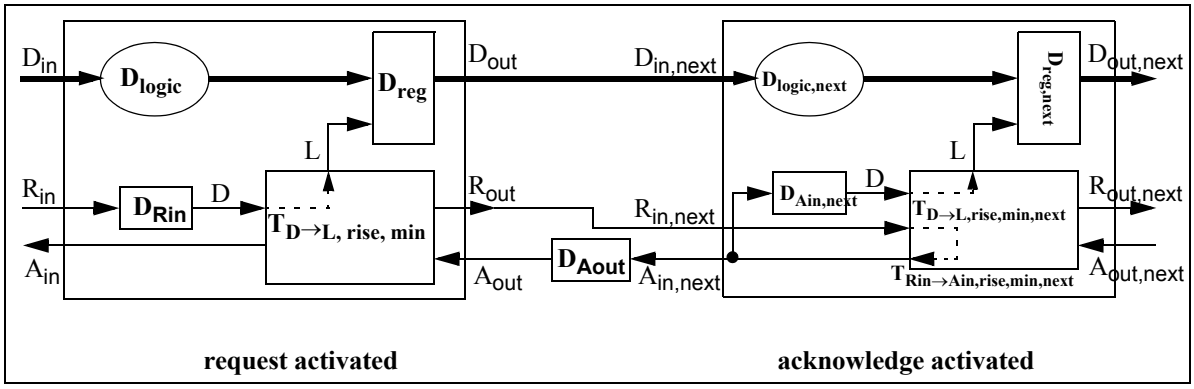


Bild 4.8: Verzögerungsglied in der Acknowledge-Leitung zwischen einer request activated und einer acknowledge activated Schaltung

gangssignal DIN: AIN und somit auch DIN darf sich erst nach einer Änderung von AOUT (abhängig von ROUT) ändern.

Bei gekoppelten Stufen kann eine Zeitbedingung $T_{DGC, Rout \rightarrow Din}$ durch einfügen von Verzögerungsgliedern erfüllt werden. Hierbei wird die Verzögerung in den Pfad des betreffeden Eingangssignals (hier DIN) eingebaut, um eine vorliegende Zeitspanne zwischen den betreffenden Signalflanken zu vergrößern.

Bei ungekoppelten Stufen muss in einer Simulation überprüft werden, ob die Zeitbedingungen eingehalten werden. Ist dies nicht der Fall, so ist eine Korrektur sehr schwierig, da die Signale der Eingangsseite unabhängig von den Signalen der Ausgangsseite schalten und somit zu beliebigen Zeitpunkten auftreten können. Ein Einfügen von Verzögerungsgliedern behebt somit das Problem nur für die aktuell vorliegende Simulation.

Inhärente Zeitbedingungen der Speicherelemente

Eine letzte Zeitbedingung, die eingehalten werden muss, ist die Pulsbreite des Speichersignals. Bei kleinen und damit schnellen Schaltungen im Einsatz einer Schieberegister-Pipeline oder bei Verwenden von standardmäßig geschlossenen Latches (das Speichersignal wird auf ’1’ und anschließend gleich wieder auf ’0’ gesetzt) kann es vorkommen, dass die geforderte Pulsbreite für das Speichersignal nicht erreicht wird. Hier müssen ebenfalls an geeigneter Stelle, zum Beispiel den Pfad des Speichersignals selbst, Verzögerungsglieder eingefügt werden.

4.2 Schnittstellen Module

4.2.1 Datenprotokolle

4.2.1.1 Zweiphasenprotokoll

Beim Zweiphasenprotokoll (vergleiche 2.4.4.1) signalisiert jede Wertänderung des Requestsignals ein neues Datum. Bild 4.9 zeigt den dafür gültigen Signalverlauf. Ein Wertewechsel des Request-Signals (R_{in}) zeigt das Anliegen eines neuen gültigen Datums an. Die erfolgreiche Übernahme wird durch einen Wertewechsel des Acknowledge-Signals (A_{in}) angezeigt. Somit spielen alle vier Flanken der beiden Steuersignale eine aktive Rolle in diesem Protokoll.

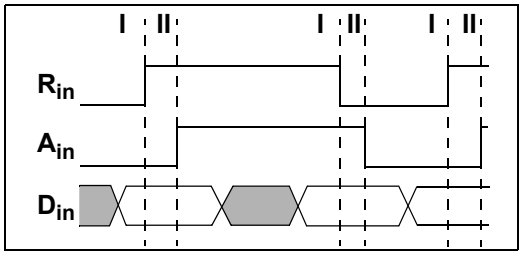


Bild 4.9: Signalverlauf beim Zweiphasenprotokoll

Eine einfache Realisierungsmöglichkeit [105] ist in Bild 4.10 gezeigt. Bei dem hierbei benutzten Speicher handelt es sich um ein zweiflanken-gesteuertes D-Flipflop (DETDFF). Es wird also mit jeder Flanke ein neues Datum übernommen. Somit ist der Kontrollaufwand gering, verglichen zum Beispiel mit der Realisierung in Bild 4.12. Allerdings benötigen die DETDFFs eine erheblich größere Fläche; laut [105] ist dies zweimal die Fläche eines einfachen flankengesteuerten D-Flipflops.

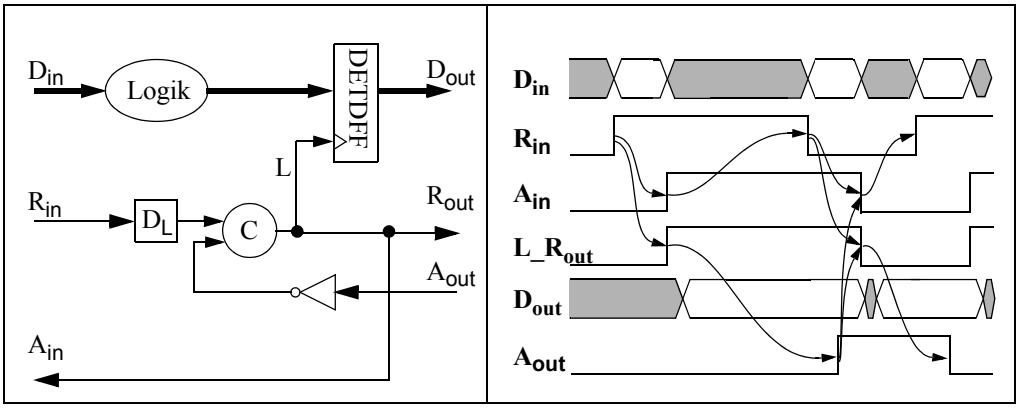


Bild 4.10: Alternative Realisierung und Signalverlauf eines Zweiphasenprotokolls

Ein Beispiel einer Realisierung eines DETDFFs ist in Bild 4.11 gezeigt. Die Kapazität, die umgeladen werden muss, ist ebenfalls erhöht (laut [105] um den Faktor 4). Eine Aussage über

den Anstieg des Gesamtverbrauchs ist aber schwierig, da er unterschiedlich gewichtet werden muss (Menge der Logik zwischen den Speichern, Häufigkeit des Schaltens gegenüber pegelsensitiven Realisierungen, und anderes).

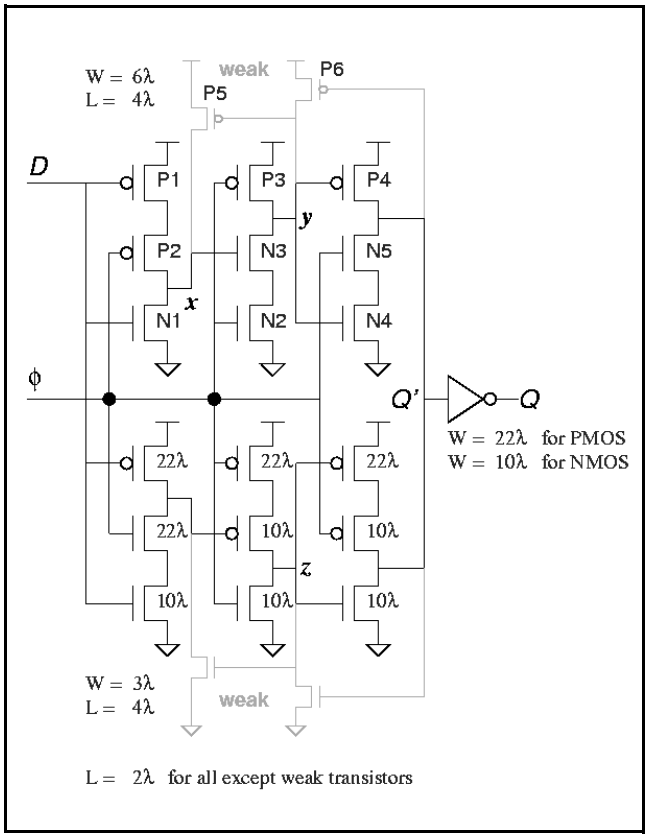


Bild 4.11: Realisierung eines zweiflanken-gesteuerten D-Flipflops [105]

Eine weitere Realisierung eines Zweiphasenprotokolls ist in Bild 4.12 gezeigt. Diese Realisierung orientiert sich nach der in [89] von Sutherland vorgeschlagenen Struktur. Das von Sutherland benutzte „Capture-Pass-Latch“ ist hier durch ein Exklusiv-Oder-Gatter, ein Latch und ein Toggle-Gatter nachgebaut. Damit ein Wertewechsel an die richtige Ausgangsleitung weitergegeben wird, benötigt man das Toggle-Gatter. Beim Toggle-Gatter wird eine Wertänderung am Eingang abwechselnd als Wertänderung an die Ausgänge weitergegeben. Hier führt zum Beispiel jeder "0" -> "1" Wechsel zum Invertieren des C_d-Ausgangs und jeder "1" -> "0" Wechsel zum Invertieren des P_d-Ausgangs.

Im neutralen Zustand der Pipeline besitzen die Steuersignale den gleichen Wert, zum Beispiel “0“. Ein neues Datum von D_{in} wird dann von einer “1“ des eingehenden Request-Signals begleitet. Die Verzögerung D_L passt die Laufzeit von R_{in} der längsten Laufzeit des Datums durch die Logik an. Der invertierte Wert von P_d und die “0“ von A_{out} sorgen für das Durchkommen der “1“ von R_{in}. Somit wird das Latch geschlossen (es wird ein normal geöffnetes Latch vorausgesetzt) und das neu errechnete Datum gehalten. R_{out} wird über das Toggle-Gatter invertiert, also auf “1“ gesetzt und daraufhin wird A_{out} letztlich auf “1“ gehen. Somit geht der Aus-

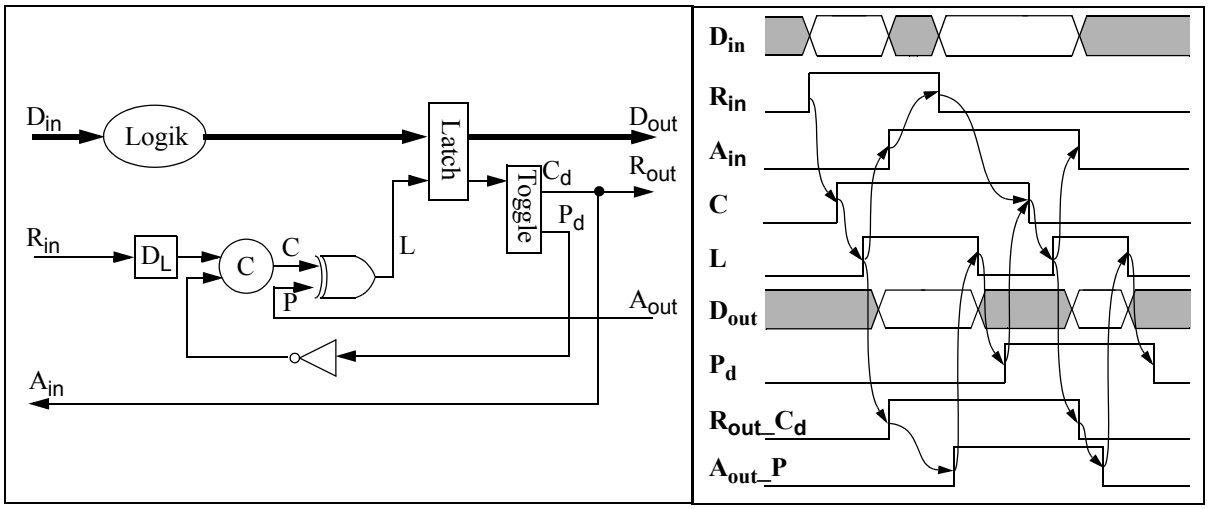


Bild 4.12: Realisierung und Signalverlauf eines Zweiphasenprotokolls

gang des Exklusiv-Oder-Gatters auf „0“, das Latch wird wieder geöffnet und nun P_d über das Toggle-Gatter invertiert, also auf „1“ gesetzt. Somit kann eine Wertänderung von R_{in} sich wiederum ausbreiten und für eine Speicherung neuer Daten sorgen.

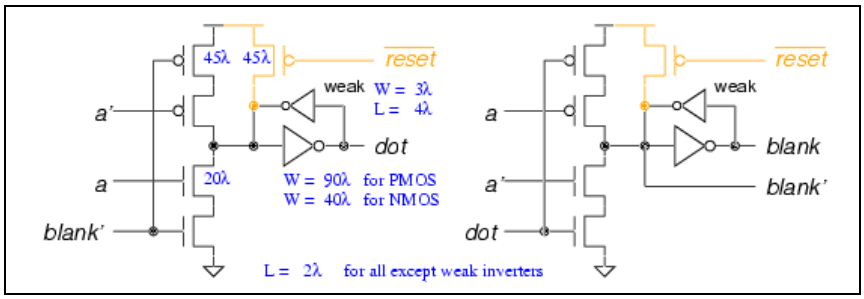


Bild 4.13: Zweiphasen-Toggle Schaltung [106]

Das in Bild 4.12 eingezeichnete Toggle-Gatter setzt eine Flanke am Eingang abwechselnd als Flankenwechsel auf die beiden Ausgänge; vergleiche hierzu den Einfluss von Signal L auf die Signale $R_{out}\_C_d$ und P_d in Bild 4.12. In Bild 4.13 ist eine Realisierung eines Zweiphasen-Toggle „Gatters“ nach [106] als Beispiel gezeigt. Das Signal *a* ist hierbei der Eingang und die Signale *blank* und *dot* sind die Ausgänge.

4.2.1.2 Vierphasenprotokoll

Beim Vierphasenprotokoll (vergleiche 2.4.4.2) wird nicht jede Flanke des Request Signals berücksichtigt. Vielmehr beschränkt man sich auf eine Art von Flanke als auslösendes Ereignis. Die andere Arte der Flanke wird in der sogenannten Rücksetzphase nicht zur Datenauswertung oder Übernahme verwendet.

Bild 4.14 zeigt beispielhaft den Signalverlauf eines Vierphasenprotokolls. Hier wird die steigende Flanke des Request-Signals (R_{in}) zur Einleitung des Datenaustausches benutzt. Die

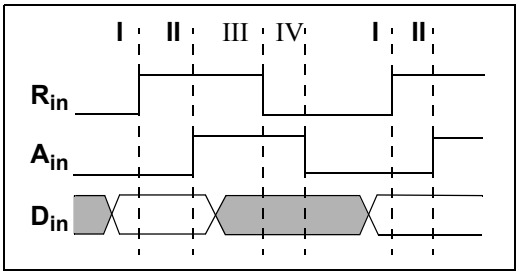


Bild 4.14: Signalverlauf beim Vierphasenprotokoll

aktuell betrachtete Stufe übernimmt das neue Datum (D_{in}) und signalisiert dies mit einer steigenden Flanke des Acknowledge-Signals (A_{in}). Die Rücknahme der jeweiligen Handshake-Signale hat in diesem Beispiel keine Auswirkung auf den Datenaustausch. Die beiden letzten Phasen werden auch als Rücksetzphase bezeichnet.

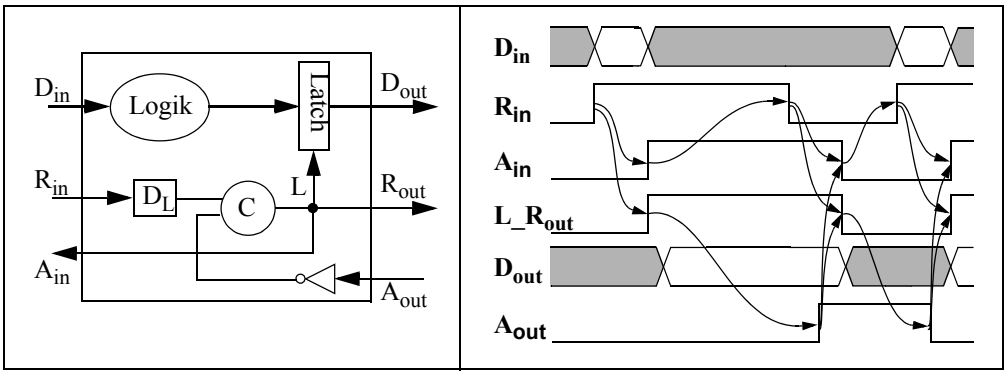


Bild 4.15: Realisierung und Signalverlauf eines Vierphasenprotokoll

Eine einfache Realisierung eines Vierphasenprotokolls ist in Bild 4.15 gezeigt. Wieder wird ein standardmäßig geöffnetes Latch vorausgesetzt. Eine neues Datum (D_{in}) wird mit dem Setzen des Request-Signals (R_{in}+) angezeigt. Dies führt zum Schließen des Latches (L+), so dass das Datum für die nachfolgende Stufe stabil gehalten wird. Gleichzeitig wird das Request-Signal für die nachfolgende Stufe (R_{out}+) und das Acknowledge-Signal für die vorausgehende Stufe (A_{in}+) gesetzt. Die Signale L, R_{out} und A_{in} werden erst dann zurückgesetzt, wenn das Request-Signal (R_{in}-) zurückgenommen und das Acknowledge-Signal (A_{out}+) gesetzt worden ist. Umgekehrt muss A_{out} wieder zurückgenommen werden, bevor über einen neuen Request (Rin+) ein neuer Datenaustausch erfolgen kann. Dieses Verhalten wird von dem Muller-C Element sichergestellt.

4.2.1.3 Bewertung von Zwei- und Vierphasenprotokollen

Bild 4.10 und Bild 4.12 zeigen mögliche Implementierungen eines Zweiphasenprotokolls. Gegenüber den Vierphasenprotokollen erhofft man sich hiermit schnellere Schaltungen. Dies

wird damit begründet, dass beim Zweiphasenprotokoll jede Flanke berücksichtigt wird und somit keine Zeit durch eine Rücksetzphase wie beim Vierphasenprotokoll benötigt wird.

Man muss allerdings aber beachten, dass Vierphasenprotokolle im Prinzip einfacher zu realisieren sind. Das heißt, dass zum Beispiel weniger Hardware für die Realisierung der Kontrollstruktur notwendig ist. In Bild 4.12 werden zum Beispiel gegenüber Bild 4.15 das Exklusiv-ODER-Gatter und das Toggle-Gatter benötigt. Diese sorgen für zusätzliche Laufzeit und damit für eine niedrigere Performanz.

Auch in der Realisierung nach Bild 4.10 ist ein zusätzlicher Flächenaufwand für die zweiflanken-gesteuerten D-Flipflops zu berücksichtigen. Diese sind größer als einfache Latches oder auch als ein „normales“ D-Flipflop.

Tendenziell resultieren Zweiphasenprotokolle also in größeren und zumeist auch langsameren Kontrollschaltungen. Somit kann der Vorteil, dass jede Flanke der Steuersignale berücksichtigt wird, nicht effizient für eine Steigerung der Performanz ausgenützt werden.

Ein anderer Gesichtspunkt ist die Tatsache, dass nur dann schnelle Kontrollschaltungen notwendig sind, wenn die Menge der Daten verarbeitenden Logik zwischen den Speichern klein ist. Ansonsten benötigt diese von sich aus mehr Zeit, als der Ablauf in den Kontrollschaltungen. Somit ist auch nicht immer die Notwendigkeit einer schnellen Kontrollschaltung gegeben.

Die Verwendung von Vierphasenprotokollen bietet sich also aus den oben genannten Gründen an. Außerdem wünscht man oft Pegel-sensitive Schaltungen, um eine optimale Implementierung zu bekommen. Somit benötigt man ein Vierphasenprotokoll, und es ist oft einfacher, dann die gesamte Schaltung im Vierphasenprotokoll zu realisieren. Somit spart man sich zusätzliche Schnittstellenmodule.

4.2.2 Klassifizierung der Vierphasenprotokolle

Beim Vierphasenprotokoll kann man zwischen verschiedenen Varianten des Datenaustauschs unter den Pipeline-Stufen unterscheiden. Ideen dafür sind in [25],[26],[53] angegeben; dort wird eine Einteilung der Vierphasenprotokolle nach dem Datengültigkeitsschema sowie nach dem Entkopplungsgrad gemacht. Zudem wird in einer Nebenbemerkung die Unterscheidung zwischen dem Start durch den Sender und dem Empfänger angesprochen. Es gibt aber noch weitere Unterscheidungskriterien, so dass sich insgesamt folgende Kriterien heranziehen lassen:

- Datengültigkeitsschema: Früh, Breit und Weit, Spät
- Start: durch den Sender oder den Empfänger
- Logikauswertung: durch die Anfrage oder die Bestätigung

- Entkopplungsgrad: voll-entkoppelt, halb-entkoppelt oder gekoppelt

- Speicher: Normal offene oder normal geschlossene Latches, Flipflops

In der Literatur werden diese verschiedenen Varianten und deren Kombination zum Teil beschrieben. Zumeist wird zur Verdeutlichung des verwendeten Protokolls zum Datenaustausch ein graphischer Signalverlauf benutzt. Ein Problem dabei ist aber, dass nicht immer die gleiche Struktur der späteren Hardware für den Datenpfad gewählt wird, auf die sich dieser zeitlich orientierte Signalverlauf bezieht. Da es ein Ziel dieser Arbeit ist, synchrone und asynchrone Module mit einer möglichst gleichartigen Methodik zu entwerfen und miteinander zu koppeln, wird die in Bild 4.16 gezeigte Struktur gewählt[1]. Diese ist die für synchrone Schaltungen gängige Struktur, in der die speichernden Elemente die Modulgrenze für die Ausgänge bilden. In Bild 4.16 sind zudem die für den asynchronen Datenaustausch notwendigen Steuersignale eingezeichnet.

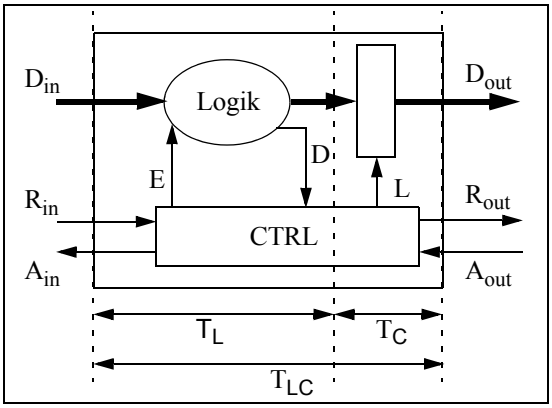


Bild 4.16: Asynchrones Modul mit allgemeiner Schnittstelle und gewählter Struktur des Datenpfads

4.2.2.1 Datengültigkeitsschema

Bei Handshakes mit gebündelten Daten signalisiert das Request-Signal ein neues Datum der vorhergehenden Stufe. Das Acknowledge-Signal zeigt eine Übernahme dieses Datums an. Entweder ist es verarbeitet und gespeichert worden (Auswertung der Logik gestartet durch den Request) oder wird noch gespeichert werden (Auswertung der Logik gestartet durch das Acknowledge). Diese Bestätigung sorgt dafür, dass die vorhergehende Stufe einen Schritt weiter im vorgesehenen Protokoll geht. Im Vierphasenprotokoll ist dies das Zurücksetzen des Request-Signals, welches ebenfalls wieder von der aktuellen Stufe bestätigt werden muss.

1. Bei NCL ist zum Beispiel die umgekehrte Anordnung von Logik und Speicher gewählt worden.

Ein Teil der Protokollspezifikation ist es festzulegen, wann das anliegende Datum innerhalb der Protokollabwicklung gültig sein muss; das heißt, wann die vorhergehende Stufe das Datum hält. Nun sind im Vierphasenprotokoll die vier Flanken der Handshake Signale als Grenzen für die Gültigkeit des anliegenden Datums verwendbar. In Bild 4.17 sind die sich ergebenden unterschiedlichen Protokollspezifikationen bezüglich der Datengültigkeit gezeigt.

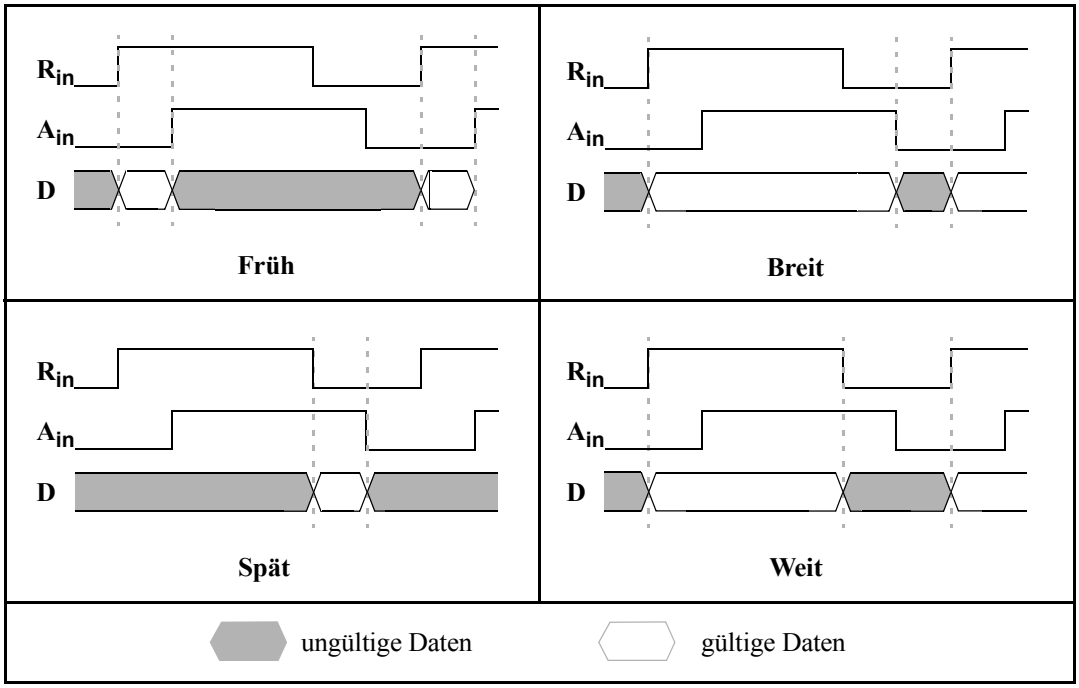


Bild 4.17: Signalverläufe für die verschiedenen Handshake-Kontrollvarianten

Anfangs sind sowohl R_{in} als auch A_{in} auf low, das heißt dem Wert "0". Beim **„Früh"-Protokoll** liegt ein neues Datum an, wenn die steigende Flanke des Request-Signals kommt. Mit der steigenden Flanke von A_{in} als Bestätigung der Übernahme des Datums darf dieses sich auf der Senderseite wieder ändern. Hier wird der Bereich von der steigenden Flanke von R_{in} bis zu der steigenden Flanke von A_{in} als „Bearbeitungsphase" bezeichnet. In diesem Bereich muss das Datum stabil bleiben. Im restlichen Bereich zwischen der steigenden Flanke von A_{in} bis zur nächsten steigenden Flanke von R_{in}, der Erholungs- oder Rücksetzphase, darf sich das anliegende Datum ändern.

Beim **„Breit"-Protokoll** zeigt ebenfalls die steigende Flanke von R_{in} ein neues Datum an. Allerdings wird die Übernahme des Datums erst mit der fallenden Flanke von A_{in} signalisiert. Somit muss das Datum über den gesamten Bereich des Handshakes stabil anliegen. Es wird nicht zwischen einer Bearbeitungsphase und Erholungsphase unterschieden.

Beim **„Weit“-Protokoll** muss das Datum nur im Bereich des gültigen Requests stabil sein, das heißt zwischen der steigenden und fallenden Flanke von R_{in}. Hier existiert wiederum eine kurze Erholungsphase zwischen der fallenden Flanke von R_{in} und der fallenden Flanke von A_{in}.

Das **„Spät“-Protokoll** benutzt die komplementären Flanken gegenüber dem „Früh“-Protokoll. Das Datum muss also zwischen der fallenden Flanke von R_{in} und der fallenden Flanke von A_{in} stabil anliegen. Der Bereich zwischen der steigenden Flanke von R_{in} bis zu der steigenden Flanke von A_{in} wird als „Setzphase“ bezeichnet, während die darauffolgende Phase wieder Berechnungsphase“ heißt.

Die verbleibenden Varianten (Datengültigkeit zwischen A_{in} steigend und R_{in} fallend, beziehungsweise zwischen A_{in} steigend und A_{in} fallend) werden selten benutzt und besitzen daher keinen Namen. Das „Spät“-Protokoll wird ebenfalls selten verwendet. Vom Verhalten her kann das „Spät“-Protokoll als „Früh“-Protokoll mit invertierten Kontrollpegeln interpretiert werden. Somit wird dieses Protokoll im Weiteren ebenfalls außer Betracht gelassen.

4.2.2.2 Start des Datenaustausches

Die erste der oben angegebenen Unterscheidungsmöglichkeiten ist der Initiator des Datenaustausches. Im Folgenden wird kurz gezeigt, dass diese Unterscheidung eine reine Betrachtungsweise ist und keine relevante Auswirkung auf die spätere Realisierung der Schaltung besitzt.

In Bild 4.18 ist die Ausbreitung von Daten entlang einer asynchronen Pipeline gezeigt. Die Module sollen alle eine Struktur gemäß Bild 4.16 besitzen. Im Fall (a) wird angenommen, dass der Sender den Datenaustausch anstößt, während im Fall (b) der Empfänger dafür verantwortlich ist. Entsprechend dieser Sichtweise bedeutet eine ’1’ eine aktiver Request beziehungsweise Anforderung. Für den Fall (a) müssen die Steuersignale der Pipeline alle mit ’0’ initialisiert werden (Schritt 0), da ja keine gültigen Daten (repräsentiert durch das ’X’) anliegen. Zu einem Zeitpunkt wird der externe Sender die Request-Leitung (R) auf ’1’ setzen, um anzuzeigen, dass ein gültiges Datum (D1) anliegt (Schritt 1), welches der Empfänger empfangen soll. Wie in Bild 4.16 gezeigt, dauert es eine gewisse Zeit T_{LC}, bis das Request-Signal durch den Logikblock (Pfad E -> D) sowie durch den Kontrollblock läuft und dann gegebenenfalls (abhängig von der vorhergehenden Stufe) für die Speicherung (Übernahme) der neu errechneten Daten sorgt. Diese Laufzeit T_{LC} wird in Bild 4.18 für alle Module als gleich groß angenommen und für die zeitliche Trennung der einzelnen Momentaufnahmen verwendet. Des Weiteren wird angenommen, dass für das Antwortsignal (A) die Laufzeit T_C (ebenfalls für alle Module gleich) durch den Kontrollblock deutlich geringer ist als T_{LC}. Es gilt also:

$$T_C < T_{LC} < 4T_C \qquad (4.7)$$

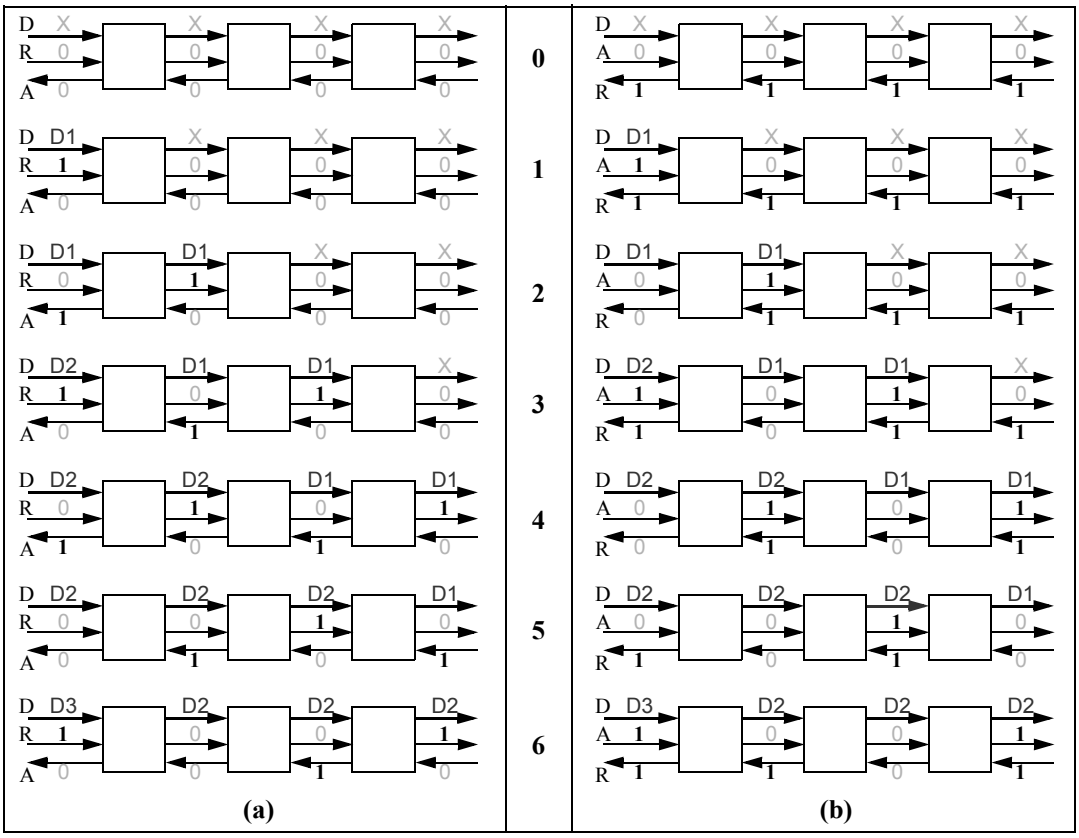


Bild 4.18: Ausbreitung der Daten über eine asynchrone Pipeline, initiiert vom Sender (a) oder Empfänger (b).

Nachdem also der erste Request durch das erste Modul gelaufen ist (Schritt 2), liegt an dessem Ausgang ein neues aus D1 hervorgegangenes Datum an, begleitet durch einen Request an das nächste Modul. Gleichzeitig wird die erfolgreiche Übernahme von D1 an das vorhergehende Modul bestätigt. Dies führt zur Zurücknahme des Request-Signals noch im selben Schritt (vergleiche Zeitbedingung von Gleichung *(4.7)*). In Schritt 3 breitet sich die Datenfront von D1 über das nächste Modul aus, während der externe Sender ein neues Datum D2 mit entsprechendem Request schickt. Der Request-Ausgang des ersten Moduls ist hier wegen der obigen Zeitbedingung schon auf ’0’. Im nächsten Schritt (4) erreicht die Datenfront von D1 das letzte Modul und die Datenfront von D2 den Ausgang des zweiten Moduls. Im Schritt 5 wandert die Datenfront von D2 ein Modul weiter; allerdings kann der externe Sender noch kein neues Datum anlegen, weil die Zeitbedingung aus Gleichung *(4.7)* angenommen wird. Der externe Empfänger bestätigt die Übernahme der Datenfront D1. Dies führt zur Bestätigung der Übernahme der „Nullfront“ im letzten Modul (1 x T_C). Dadurch kann das vorletzte Modul die Datenfront von D2 übernehmen und quittieren (2 x T_C). Daraufhin kann das erste Modul dem externen Sender die „Nullfront“ bestätigen (3 x T_C). Ein mögliches neues Datum des externen Senders wird aber noch nicht angezeigt, da hier ein viertes mal die Zeit T_C benötigt wird.

Vergleicht man jetzt den Daten- und Kontrollfluss von Bild 4.18 (a) mit dem von Bild 4.18 (b), so erkennt man, dass sich die Daten in beiden Fällen gleich ausbreiten. Hierbei werden wieder gleiche Laufzeiten für die Daten, als auch die Kontrollsignale angenommen. Im Speziellen soll hier ebenfalls Gleichung *(4.7)* gelten. Die unterschiedliche Betrachtungsweise äußerst sich nur in der Übernahme des invertierten Pegels des Acknowledge-Signals als Request-Signal im zweiten Fall (b). Das ursprüngliche Request-Signal vom ersten Fall (a) kann direkt als Acknowledge-Signal im zweiten Fall (b) übernommen werden.

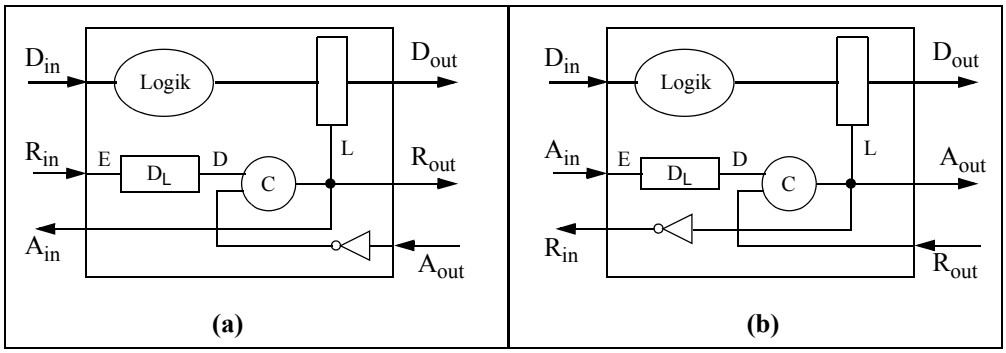


Bild 4.19: Asynchrone Module mit Datenaustausch, initiiert vom Sender (a) oder vom Empfänger (b)

Für eine spätere Realisierung bedeutet die Unterscheidung der Initiierung des Datenaustausches eigentlich nur ein unterschiedliches Setzen der Modulgrenzen bezüglich des Kontrollblockes. Bild 4.19 zeigt zwei mögliche Realisierungen für asynchrone Module die ihre Daten nach einem Vierphasen-Protokoll austauschen. Im Fall (a) initiiert der Sender den Datenaustausch und im Fall (b) der Empfänger. Außer dem Vertauschen der Namen der Steuersignale ändert sich nur der Ort des Inverters. Im Fall (a) wird das eingehende Acknowledge-Signal invertiert; im Fall (b) das ausgehende Request-Signal.

In einer zusammengeschalteten Pipeline ist aber ein ausgehendes Steuersignal verknüpft mit dem eingehenden Steuersignal des vorhergehenden oder des nachfolgenden Moduls. Schaltet man also jeweils zwei Module von Fall (a) und Fall (b) zusammen und lässt die Signalnamen sowie die Modulgrenzen weg, kann man in den Schnittstellen zwischen den jeweiligen Modulen keine Unterschiede mehr feststellen. Aus diesem Grund wird im Weiteren nur mehr der Fall (a) angenommen, dass der Sender den Datenaustausch startet.

4.2.2.3 Logikauswertung

In Bild 4.16 ist zu sehen, dass die Auswertung des eingehenden Datums mit der Steuerung gekoppelt ist. Es muss mindestens eine zeitliche Kopplung vorhanden sein, wie es zum Beispiel in Bild 4.19 gezeigt ist. Hier wird das entsprechende Steuersignal um die Zeit T_L verzögert. T_L repräsentiert dabei eine Zeit die im schlimmsten Fall benötigt wird, um ein gültiges und stabiles Datum am Ende der Logikwolke vorliegen zu haben. Es ist auch ein Mechanismus denkbar, wie

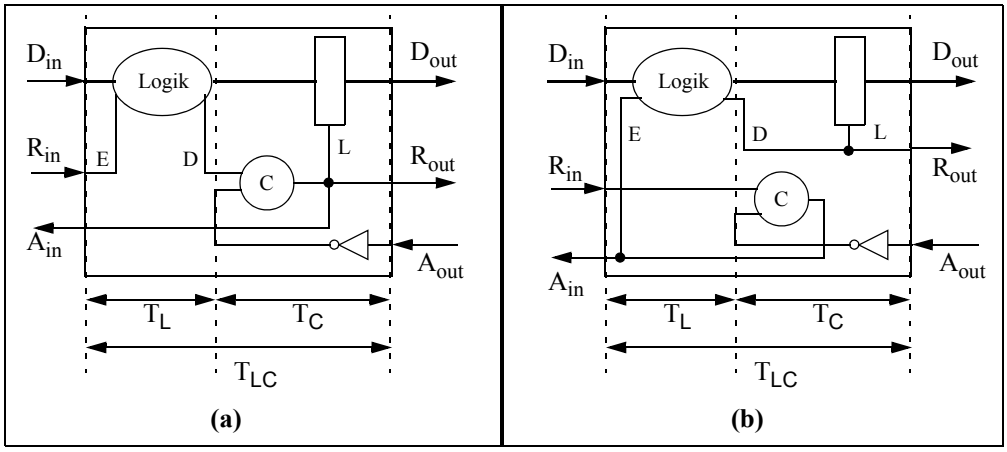


Bild 4.20: Asynchrone Module mit Logikauswertung, initiiert vom Request (a) oder Acknowledge (b)

er in Bild 4.16 angedeutet ist. Der Kontrollblock erzeugt ein Enable-Signal mit dem die Auswertung des anliegenden Datums ermöglicht wird. Ist die Logik mit der Auswertung fertig, erzeugt sie über einen Komplettierungsmechanismus ein Done-Signal, welches von dem Kontrollblock ausgewertet wird, um zum Beispiel das neue Datum in den Speichern festzuhalten.

Das in beiden Fällen erzeugte Enable-Signal kann in den einfachsten Fällen direkt aus dem Request-Signal oder dem Acknowledge-Signal abgeleitet werden (Bild 4.20). Im normalen Betrieb ohne Konflikte sind keine Unterschiede in der Datenausbreitung auszumachen.

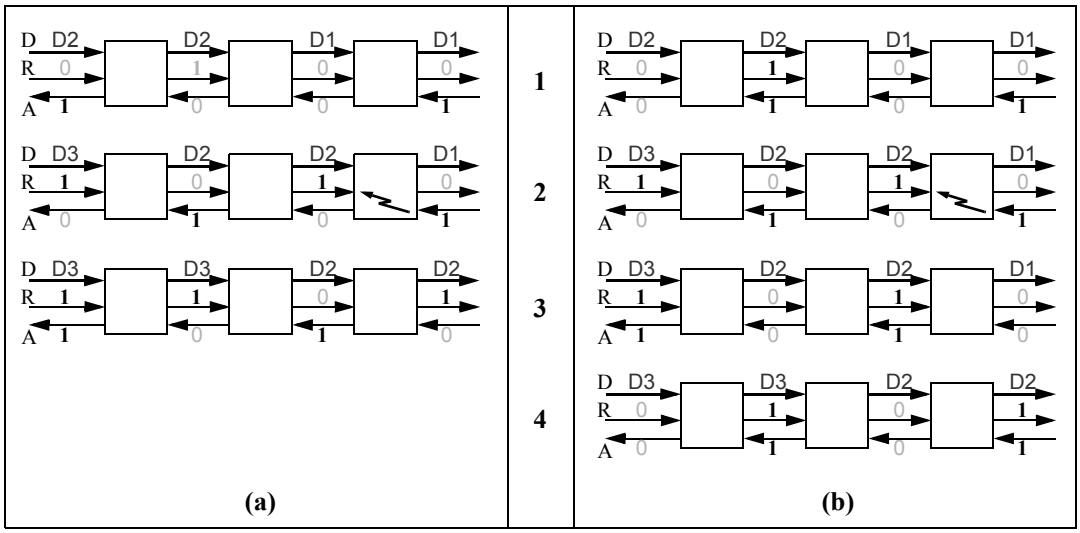


Bild 4.21: Konfliktauflösung bei einer asynchrone Pipeline; Logikauswertung über den Request (a) oder das Acknowledge (b).

Wenn allerdings an einer Stelle ein Konflikt auftritt, so dass ein Datum nicht weiterlaufen kann, so ergibt sich für die Variante der Logikauswertung durch das Acknowledge-Signal (Bild 4.20 (b)) eine größere Reaktionszeit. In Bild 4.21 ist für beide Varianten der Konfliktfall gegeben, dass der externe Empfänger die Pipeline blockiert (Die Nullfront wird nicht bestätigt; Schritt 1). In Schritt 2 sind beide Pipelines komplett blockiert; das heißt es kann von Außen kein

weiteres Datum angelegt werden. Im nächsten Schritt 3 bestätigt der externe Sender schließlich die Nullfront. Im Fall (a) wird die Datenfront D2 zirka nach der Zeit T_C im letzten Modul übernommen und die Datenfront D3 zirka nach der Zeit $3T_C$.

In der zweiten Variante (Bild 4.21 (b)) wird durch das Bestätigen des externen Empfängers der Nullfront erst einmal nur die Rückwärtsausbreitung der jeweiligen Acknowledge's ermöglicht. Die noch notwendige Auswertung der Daten nimmt aber zusätzlich ungefähr die Zeit T_L in Anspruch. Somit wird hier die Datenfront D2 im letzten Modul zirka erst nach der Zeit $T_{LC} = T_C + T_L$ übernommen (Schritt 4). Die Datenfront D3 wird entsprechend erst nach der Zeit $3T_C + T_L$ übernommen.

Somit ergibt sich für die Reaktionszeiten auf eine Konfliktaufhebung für die Variante der Datenauswertung mittels des Acknowledge-Signals (TReactDA) gegenüber der Datenauswertung mittels des Request-Signals (TReactDR):

$$T_{ReactDA} = T_{ReactDR} + T_L \qquad (4.8)$$

Für eine praxisnahe Bewertung muss aber das jeweilige Konzept im Einsatz innerhalb einer Mikropipeline betrachtet werden. Bild 4.22 zeigt, dass die längere Reaktionszeit aber im Allgemeinen zu keiner Einbuße der Performanz führt. Es werden zur Vereinfachung folgende Laufzeiten für die Module A, B und C angenommen:

$$T_{CA} = T_{CB} = T_{CC} = T_C \;\; ; \;\; T_{LA} = T_{LB} = T_{LC} = 3 \cdot T_C \qquad (4.9)$$

Für den externen Empfänger (rechte Seite) gilt:

$$T_{Cextern} = T_C \;\; ; \;\; T_{Lextern} = 9 \cdot T_C \qquad (4.10)$$

Da der Empfänger langsamer reagiert, wird sich die Pipeline auf jeden Fall füllen und für einen Konfliktfall sorgen. Man sieht in Bild 4.22, dass sich die Datenfronten von D1 und D2 bis zur Zeit $17T_C$ in beiden Varianten der Mikropipeline gleich ausbreiten. Allerdings fällt hier schon auf, dass bei Variante (a) ein Konflikt früher auftritt, als dies bei der Variante (b) der Fall ist. Dies liegt an den unterschiedlichen Bestätigungsprinzipien.

Bei der Datenevaluierung auf Grund eines Requests (Fall a), wird ein Acknowledge-Signal erst dann generiert, wenn die neu berechneten Daten tatsächlich gespeichert wurden. Somit tritt sofort dann ein Konflikt auf, wenn neue Daten nicht übernommen werden.

In Variante (b) wird das Acknowledge „im Voraus" gegeben. Somit tritt hier erst dann ein Konflikt auf, wenn das nachgeschaltete Modul die Daten doch nicht übernimmt. In diesem Fall wird kein (vorausschauendes) Acknowledge für eine Nullfront geliefert und dadurch die Pipeline blockiert.

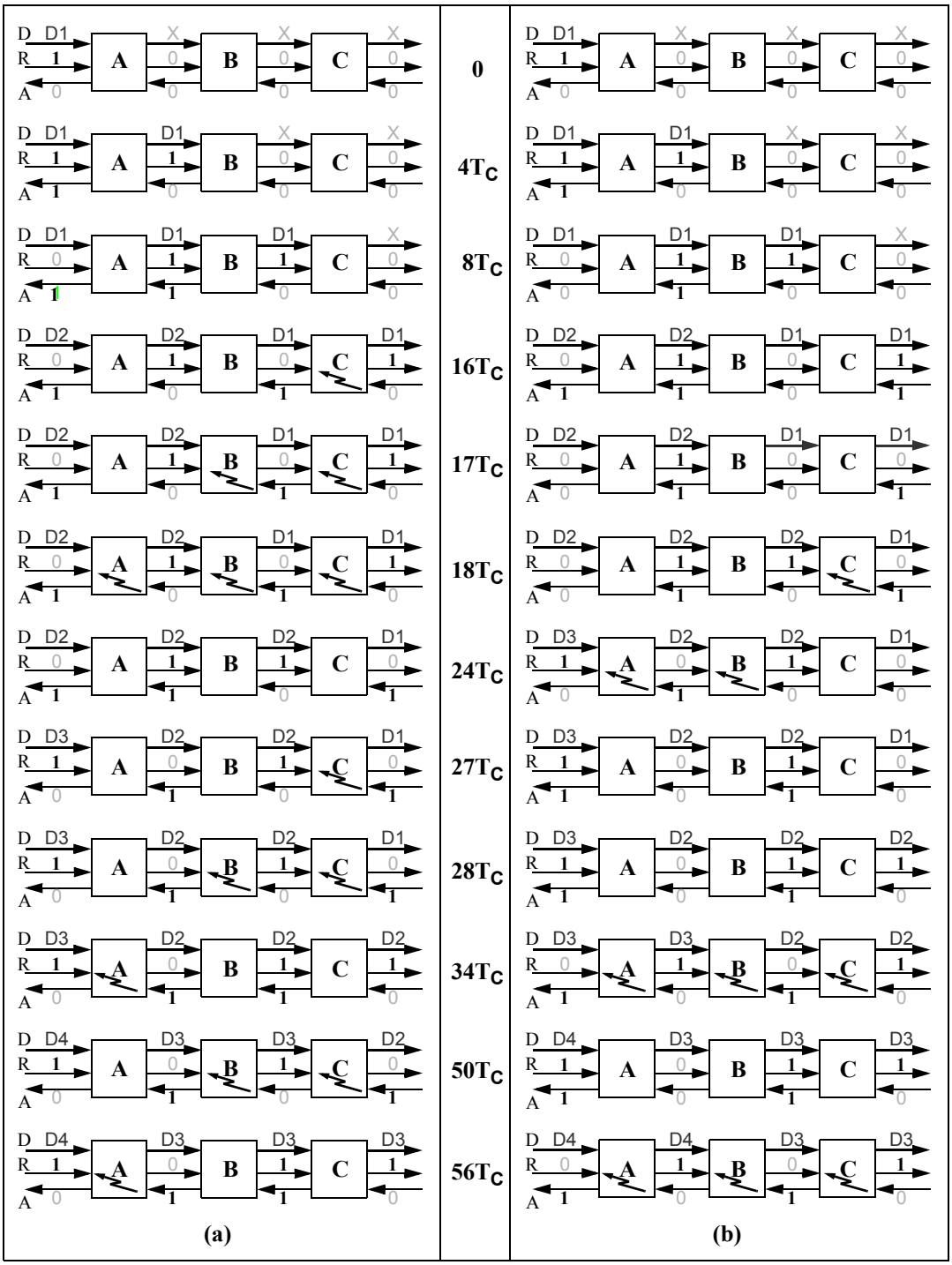


Bild 4.22: Konflikt bei Ausbreitung von Daten über eine asynchrone Pipeline; Logikauswertung über den Request (a) oder das Acknowledge (b).

Des Weiteren führt das vorausschauende Prinzip von Variante (b) dazu, dass die Datenfront sich in gleicher Zeit gegenüber Variante (a) um ein Modul weiter ausbreitet (Zeit $18T_C$). Ist der Konflikt beseitigt, reagiert Variante (b) zwar prinzipiell langsamer, allerdings entsteht der Konflikt in einer anderen Phase als bei der ersten Variante (a), nämlich bei dem Acknowledge der Nullfront. Somit ist diese Variante gegenüber der anderen hier schon einen halben Schritt voraus.

Insgesamt führt die zweite Variante zu einer kleineren Latenzzeit; das heißt der vorausschauende Charakter sorgt hier dafür, dass die Pipeline sich schneller füllt und somit die jeweilige Datenfront um die Zeit $6T_C$ eher am Ausgang anliegt. Es ergibt sich aber kein höherer Datendurchsatz, da der langsame Empfänger in beiden Varianten der entscheidende Faktor für den Datendurchsatz ist. Befinden sich nach dem langsamen Empfänger weitere Module, so werden aus deren Sicht die Daten tatsächlich mit der gleichen Datenrate und Latenzzeit ankommen (siehe Anhang A Bild A.35).

4.2.2.4 Entkopplungsgrad

Der Entkopplungsgrad ist eine Angabe darüber, wie die Eingabeseite mit der Ausgabeseite verknüpft ist. Für die weitere Diskussion sollen hier kurz drei Begriffe vorgestellt werden:

Definition 18 Initiierte Kommunikation:
Eine Kommunikation wird als initiiert bezeichnet, wenn ein neuer Request nicht abgeblockt wird, sie sich also fortpflanzen kann.

Definition 19 Komplettierte Kommunikation:
Eine Kommunikation wird als komplettiert bezeichnet, wenn das Rücksetzen des Request-Signals erfolgreich zu einem Rücksetzen des Acknowledge-Signals geführt hat.

Definition 20 Unterbrochene Kommunikation:
Eine Kommunikation wird als unterbrochen bezeichnet, wenn das Rücksetzen des Request-Signals sich nicht fortpflanzen kann.

Diese Zustände von Handshakes werden herangezogen, um den Entkopplungsgrad einer Mikropipeline-Stufe zu definieren (siehe auch [55]). Es kommt hierbei darauf an, welche Auswirkungen der Zustand der Handshakes auf der Ausgangsseite auf den der Eingangsseite hat.

Definition 21 Gekoppelt (coupled)
Die Stufe einer Mikropipeline wird als gekoppelt bezeichnet, wenn
- *eine neue Kommunikation auf der Eingangsseite nicht initiiert werden kann, bis die aktuelle Kommunikation auf der Ausgangsseite komplettiert worden ist.*
- *eine neue Kommunikation auf der Eingangsseite unterbrochen wird, falls eine neue Kommunikation auf der Ausgangsseite nicht initiiert wurde.*

Definition 22 Halb-Entkoppelt (semi-decoupled)
Die Stufe einer Mikropipeline wird als halb-entkoppelt bezeichnet, wenn die erste Bedingung aus Definition 21 wegfällt.

Definition 23 Voll-Entkoppelt (decoupled)
Die Stufe einer Mikropipeline wird als entkoppelt bezeichnet, wenn keine der beiden Bedingungen aus Definition 21 gelten.

Bild 4.23 zeigt die beiden Bedingungen aus der Definition 21. Bei der ersten Bedingung wird der Request (R_{in}) auf der Eingangsseite so lange blockiert, bis das Acknowledge (A_{out}) auf der Ausgangsseite auf "0" liegt, die aktuelle Kommunikation also beendet beziehungsweise komplettiert ist. Somit muss also das nachfolgende Latch offen (leer) sein, damit das betrachtete Latch sich schließen (füllen) kann. Wenn eine solche Pipeline gefüllt ist, dann besitzen die Latches nur alternierend Daten. Jedes zweite Latch ist offen, also leer.

Auch bei der zweiten Bedingung (Bild 4.23) wird die Eingangsseite blockiert; diesesmal kann das Zurücksetzen des Requests nicht bis zum Latch durchkommen, da die Initiierung auf der Ausgangsseite noch nicht bestätigt wurde. Bild 4.20 beziehungsweise Bild 4.19 zeigen also Implementierungen von gekoppelten Mikropipeline-Stufen.

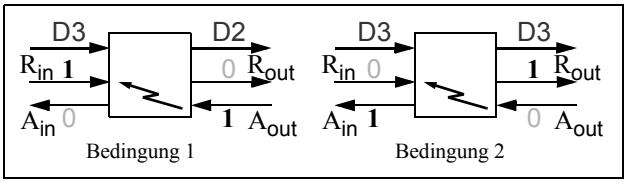


Bild 4.23: Zustände und gegenseitige Behinderung der Handshakes als Kopplungskriterium

Bei halb entkoppelten Stufen darf sich das aktuelle Latch also füllen, obwohl das nachfolgende Latch noch nicht geöffnet ist. Allerdings darf das aktuelle Latch nicht geöffnet werden, bevor das nachfolgende Latch nicht bestätigterweise geschlossen worden ist. Bei voll entkoppelten Stufen darf sich die aktuelle Stufe füllen und leeren ohne das entsprechende Acknowledge der nachfolgenden Stufe erhalten zu haben.

4.2.2.5 Speicher

Bei den verwendeten Speichern kann man zwischen Flipflops und den transparenten Latches unterscheiden. Es soll hier zunächst auf die transparenten Latches eingegangen werden, da diese im Allgemeinen in asynchronen Schaltungen verwendet werden.

Es gibt zwei verschiedene Betriebsarten von transparenten Latches in Mikropipelines:

- Normal geöffnet: Das Latch ist standardmäßig geöffnet. Ein neuer Handshake schließt das Latch und hält das neu errechnete Datum, gesteuert durch das Aktivierungssignal; dieses ist entweder das Request- oder das Acknowledge-Signal (siehe 4.2.2.3 "Logikauswertung" auf Seite 95).

- Normal geschlossen: Das Latch ist standardmäßig geschlossen. Ein neuer Handshake öffnet das Latch und das neu errechnete Datum wird übernommen.

Der Vorteil der normal geöffneten Latches liegt in der Tatsache, dass sich ein Datum schneller über die Pipeline ausbreiten kann. Darauf basierende Pipelines verhalten sich im Anfangszustand wie eine kombinatorische Schaltung. Damit kann sie einfacher getestet werden. Allerdings pflanzen sich Spikes über die gesamte Pipeline fort und sorgen somit für eine erhebliche Verlustleistung.

Verwendet man indessen normal geschlossene Latches so werden die Daten erst übernommen werden, wenn sie stabil an den Eingängen des Latches anliegen. Somit können sich die Glitches und Spikes nicht mehr fortpflanzen. Die damit verbundene Minderung der Verlustleistung erkauft man sich aber mit einer Verschlechterung der Performanz. Es muss an jeder Stufe gewartet werden, bis das entsprechende Datum stabil anliegt, um es dann zu übernehmen.

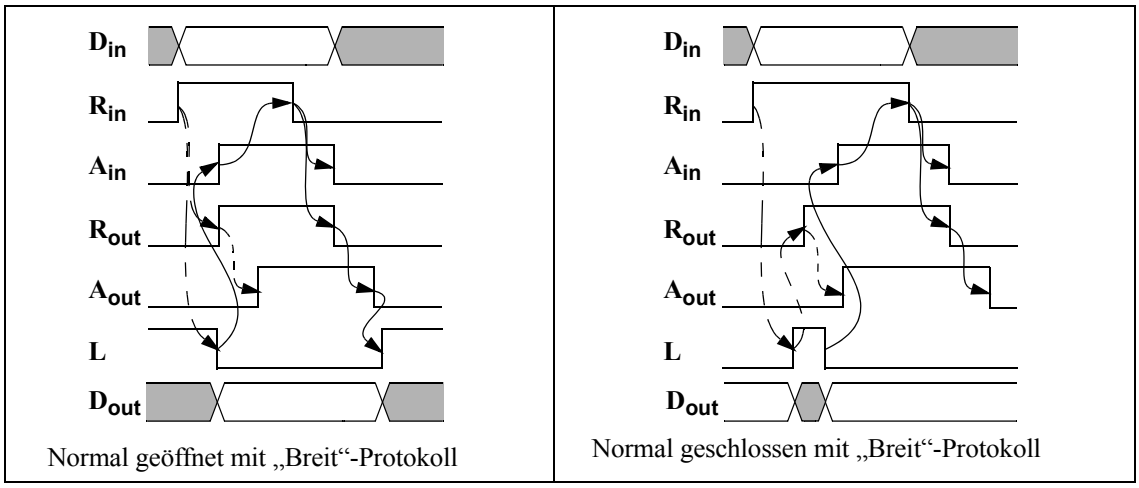


Bild 4.24: Signalverläufe für die zwei Betriebsarten von Latches in einer Mikropipeline

Bild 4.24 zeigt typische Signalverläufe für die zwei unterschiedlichen Betriebsarten von transparenten Latches. Die verwendeten Signalnamen basieren dabei auf den in Bild 4.16 gezeigten. Links in Bild 4.24 werden normal geöffnete Latches verwendet. Ein Request an der Eingangsseite (R_{in}+) führt nach einer gewissen Zeit (Durchlaufzeit der Kombinatorik beziehungsweise Verzögerungen im Kontrollpfad eines FIFOs) zu einem Sperren der Latches (L-) und einem damit einhergehenden Acknowledge des Requests (A_{in}+) sowie eines Request an der Ausgangsseite (R_{out}+). Die Latches bleiben nun solange geschlossen, bis auf der Ausgangsseite das Rücksetzen des Requests bestätigt worden ist (A_{out}-).

In der rechten Grafik von Bild 4.24 werden normal geschlossene Latches verwendet. Ein Request an der Eingangsseite (R_{in}+) führt nach einer gewissen Zeit zu einem Öffnen der Latches (L+). Die interne vorliegende Kontrolllogik muss nun für ein Setzen des ausgangsseitigen Requests (R_{out}+) als auch für ein erneutes Schließen der Latches (L-) sorgen.

Die zu generierende Flanke zum Schließen der Latches in der rechten Grafik in Bild 4.24 sorgt für eine Verlangsamung, da die Bestätigung des eingangseitigen Requests erst nach dem

Zurücknehmen von L gesetzt werden kann. Somit werden normal offene Latches verwendet, wenn es auf die Performanz ankommt; während normal geschlossene Latches bei gefordertem niedrigeren Verbrauch benutzt werden.

Eine weitere Möglichkeit ist es, wie in synchronen Entwürfen Flipflops zu verwenden. Somit würden Daten nur bei einer Art von Flanke (oftmals die steigende) übernommen werden. Wie bei den normal geschlossenen Latches pflanzen sich also Spikes nicht fort. Andererseits wird das Protokoll nicht verlangsamt, da die inverse Flanke des Latchsignals keine Auswirkungen hat. Das Latchsignal muss somit nicht intern, sondern kann durch externe Kontrollsignale zurückgesetzt werden.

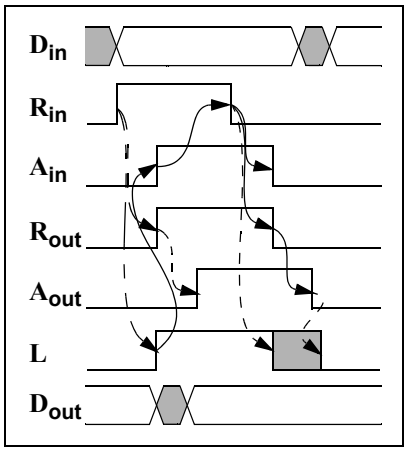


Bild 4.25: Signalverlauf für die Verwendung von Flipflops in einer Mikropipeline

Dieser Sachverhalt wird in Bild 4.25 dargestellt. Die Schraffierung zeigt hierbei einen möglichen Bereich für die fallende Flanke des Latchsignals L an. Da das hier angenommene Flipflop Daten nur mit der steigenden Flanke des Latchsignals übernimmt, liegt das Datum automatisch für den Rest des Protokollablaufs stabil am Ausgang an. Somit ergibt sich hier automatisch das „Breit"-Protokoll.

Mit den gezeigten Unterscheidungsmerkmalen lassen sich die unterschiedlichsten Kontrollstrukturen aufbauen und entsprechende unterschiedlich reagierende Pipelines realisieren. Die in Bild 4.19 (a) gezeigte Kontrollstruktur implementiert zum Beispiel ein Vierphasenprotokoll welches gekoppelt ist, eine Request-aktivierte Logikauswertung besitzt, ein „Früh"-Protokoll benutzt und mit normal geöffneten Latches arbeitet.

4.2.3 Systematik der Kontrollstrukturen

Die in dem vorigen Kapitel aufgeführten Unterscheidungsmerkmale ergeben eine Vielzahl an Kombinationsmöglichkeiten. Lässt man die Unterscheidung der Initialisierung des Datenprotokolls weg (Sender oder Empfänger) und beschränkt sich bei den Datengültigkeitsschemata

auf die drei vorgestellten Varianten „früh“, „weit“ und „breit“, dann ergeben sich als ein theoretisches Maximum 54 Möglichkeiten (Logikauswertung 2; Kopplungsgrad 3; Datengültigkeit 3; Speicher 3).

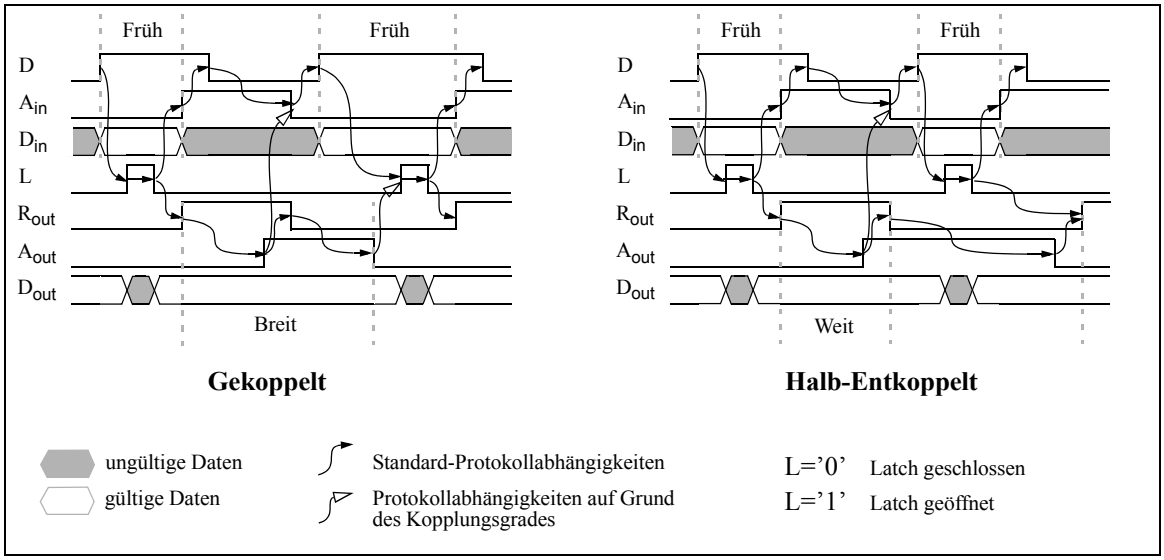


Bild 4.26: Signalverläufe für request-activated Datenprotokolle für die Verwendung mit normal geschlossenen Latches

Allerdings ergeben sich ein paar Einschränkungen. So machen einige Kombinationsmöglichkeiten keinen Sinn oder sind praktisch nicht realisierbar. Betrachtet man zum Beispiel die normal geschlossenen Latches, so erkennt man, dass diese mit der steigenden Flanke des Signals L ein neues Datum übernehmen. Danach wird das Latch geschlossen und das Datum dadurch stabil gehalten. Der Kopplungsgrad entscheidet hier letztendlich, wann ein Latch wieder geöffnet werden darf, um ein neues Datum zu übernehmen.

Bild 4.26 zeigt Signalverläufe für zwei verschiedene Kopplungsfälle bei eingangsseitig anliegendem „Früh“-Protokoll. Das Signal D ist dabei das direkt aus R_{in} hervorgehende Done-Signal, welches das Ende der Datumsauswertung kennzeichnet. In dem Fall einer vollständigen Kopplung ergibt sich am Ausgang der Mikropipelinestufe ein „breites“ Protokoll. In dem Fall einer halben Entkopplung ergibt sich am Ausgang ein „weites“ Protokoll. Es lassen sich also mit normal geschlossenen Latches nur dann „Früh“-Protokolle realisieren, wenn entkoppelte Pipelinestufen vorliegen. Ein „Weit“-Protokoll lässt sich mit entkoppelten und halb-entkoppelten Pipelinestufen realisieren.

Bei der Verwendung von Flipflops ergibt sich genau der gleiche Sachverhalt wie bei normal geschlossenen Latches. Auf Grund dieser Tatsachen fallen für die request-activated Protokolle 6 von den oben genannten 54 Kombinationsmöglichkeiten weg.

Des Weiteren fallen für den Fall der acknowledge-activated Logik das „Früh“-Protokoll und das „Weit“-Protokoll und damit weitere 18 Kombinationsmöglichkeiten weg (Bild 4.27). Da die Logikauswertung erst mit dem eingangsseitigen Acknowledge startet, darf das anliegende

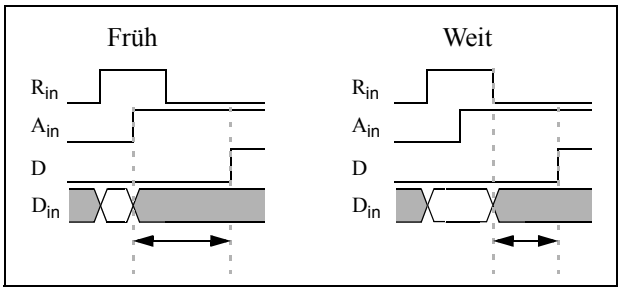


Bild 4.27: Verletzungen des Datenprotokolls bei Verwendung von acknowledge-activated Protokollen

Datum natürlich nicht mit der steigenden Flanke von A_{in} („Früh"-Protokoll) oder der fallenden Flanke von R_{in} („Weit"-Protokoll) ungültig werden. Bei beiden Protokollen kann die Verarbeitung eines Datums zu lange dauern, so dass ein neues (ungültiges) Datum in die aktuelle Verarbeitung einfließen und somit zu einem falschen Ergebnis führen kann.

Unter Berücksichtigung der eben genannten Einschränkungen verbleiben noch 30 Möglichkeiten. Zur Kennzeichnung der Kontrolllogik werden folgende Buchstaben verwendet:

- Aktivierung: R = **R**equest activated
  A = **A**cknowledge activated
- Entkopplungsgrad: U = **U**ndecoupled
  S = **S**emidecoupled
  D = **D**ecoupled
- Datengültigkeit: E = **E**arly
  W = **W**ide
  B = **B**road
- Speicher: LO = **L**atch **O**pened
  LC = **L**atch **C**losed
  FF = **F**lip**f**lop

In Tabelle 1 sind die sich ergebenden Kombinationsmöglichkeiten für die verschiedenen Kontrollstrukturen zusammengefasst.

Tabelle 1: Mögliche Kontrollstrukturen für Mikropipelines

		request activated			acknowledge activated		
		Gekoppelt	Halbent-koppelt	Entkoppelt	Gekoppelt	Halbent-koppelt	Entkoppelt
Früh	Latch Opened	RUELO	RSELO	RDELO			
	Latch Closed			RDELC			
	Flipflop			RDEFF			

Tabelle 1: Mögliche Kontrollstrukturen für Mikropipelines

		request activated			acknowledge activated		
		Gekoppelt	Halbent-koppelt	Entkoppelt	Gekoppelt	Halbent-koppelt	Entkoppelt
Weit	Latch Opened	RUWLO	RSWLO	RDWLO			
	Latch Closed		RSWLC	RDWLC			
	Flipflop		RSWFF	RDWFF			
Breit	Latch Opened	RUBLO	RSBLO	RDBLO	AUBLO	ASBLO	ADBLO
	Latch Closed	RUBLC	RSBLC	RDBLC	AUBLC	ASBLC	ADBLC
	Flipflop	RUBFF	RSBFF	RDBFF	AUBFF	ASBFF	ADBFF

Im Folgenden werden die verschiedenen Kontrollstrukturen vorgestellt. Dabei wird jeweils als Spezifikation ein Burst Mode Graph [45] gezeigt, da dieser als Eingabe für das verwendete Synthesewerkzeug DGC [45] verwendet wird. Die Syntax für die Zustandsübergänge ist hierbei: Eingangsänderungen / Ausgangsänderungen. Zum Beispiel besagt A+ B+ | C-, dass der betreffende Übergang stattfindet, wenn die Werte sowohl von A als auch von B auf ’1’ wechseln (Eingangs-Burst). Mit diesem Wechsel wird der Wert von C aus ’0’ gesetzt (Ausgangs-Burst). Neben Wertänderungen kann man auch Wertepegel von „nicht relevanten“ Eingangssignalen abprüfen. Hierzu verwendet man eckige Klammern. Mit [A+] prüft man also die Bedingung A=’1’ ab.

4.2.3.1 RUELO

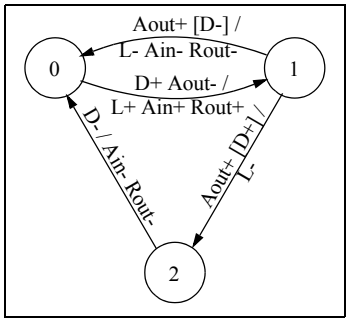


Bild 4.28: Zustandsgraph für die Kontrollstruktur RUELO

Bild 4.28 zeigt den Automaten für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Früh“-Protokoll. Zur Vereinfachung des Automaten ist der Anfangszustand mit Aout=’1’ gewählt. Tatsächlich werden durch einen Reset aber alle Ausgangssignale der Vorgängerstufe auf ‘0’ gesetzt, also besitzt auch Aout den Wert ‘0’ nach einem

Reset. Das hat hier und in anderen Automaten (siehe folgende Kapitel) aber keine negativen Auswirkungen. Die zur Implementierung verwendeten Grundgatter sind nur pegelsensitiv. Somit wird in der vorliegenden Schaltung bei einem durchgeführten Reset implizit gleichzeitig ein Aout- registriert.

Da die Kontrollstruktur RUELO request-activated (**R**) ist, erscheint in dem Automaten das Signal D (für **D**one) statt dem Request-Signal Rin (vergleiche auch Bild 4.26). Das Signal D ergibt sich entweder durch eine Verzögerung von Rin oder durch eine explizite Generierung in der Logik als Folge von Rin.

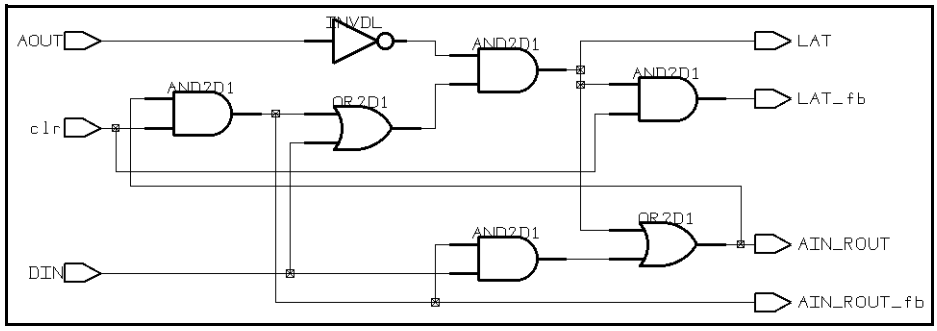


Bild 4.29: Implementierung der Schaltung RUELO

Bild 4.29 zeigt eine Implementierung der RUELO Schaltung. Da die Ausgangssignale Ain und Rout in dem Automaten nach Bild 4.28 immer gleichzeitig gesetzt werden, wurden sie zu einem Signal AIN_ROUT zusammengefasst. Wählt man in DGC [45] die Feedback-Option, dass also die Ausgangssignale als interne Zustandssignale berücksichtigt werden sollen, erhält man für diese Signale neue Signale mit der Erweiterung „_fb“. Hiermit wird gekennzeichnet, dass es sich hierbei um zurückgeführte Leitungen handelt. Diese sind dann die neuen, in der äußeren Beschaltung zu verwendenden Ausgangssignale. clr ist das von DGC eingesetzte „low“-aktive Resetsignal, welches jeweils über ein UND-Gatter die Ausgangssignale zurücksetzt.

4.2.3.2 RSELO

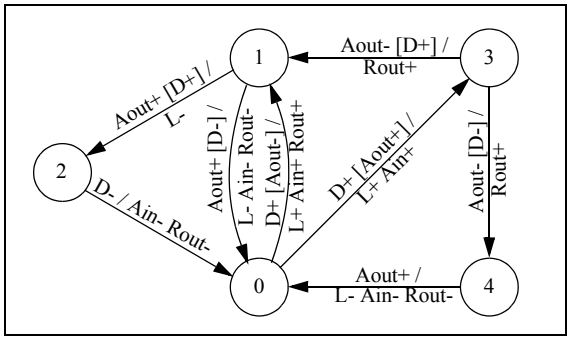


Bild 4.30: Zustandsgraph für die Kontrollstruktur RSELO

In Bild 4.30 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Früh“-Protokoll gezeigt. Gegenüber Bild 4.29 kommt auf Grund der Lockerung der Kopplung ein Freiheitsgrad hinzu. Eingangsseitig darf ein neues Protokoll gestartet werden, ohne dass das ausgangsseitige Protokoll abgeschlossen ist. Somit darf Ain+ vor Aout- auftreten. Dies führt zu dem zusätzlichen Zustand 3; aus diesem Zustand führt ein weiterer Weg über Zustand 4 wieder in den Anfangszustand zurück.

4.2.3.3 RDELO

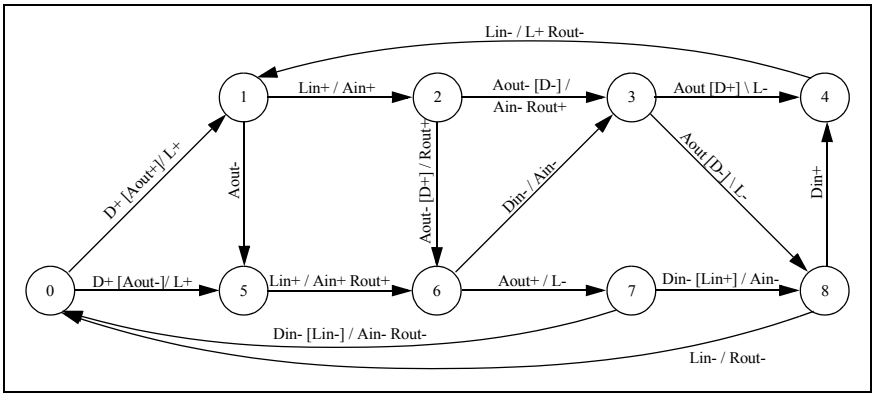


Bild 4.31: Zustandsgraph für die Kontrollstruktur RDELO

In Bild 4.31 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Früh“-Protokoll gezeigt. Als weiterer Freiheitsgrad darf auf Grund der Entkopplung Ain- vor einem Aout+ auftreten. Man muss allerdings beachten, dass eingangsseitig nur genau ein weiteres Protokoll gestartet werden darf. Das heißt ein „zweites“ Ain+ darf erst dann gesetzt werden, wenn das „erste“ Aout+ detektiert wurde.

Wegen der Entkopplung und der hier benutzten standardmäßig geöffneten Latches sowie dem „Früh“-Protokoll muss man zusätzlich den Zustand des Latchsignals (**L**) beachten. Speziell darf ein Ain+ erst erfolgen, wenn die Latches im geschlossenen Zustand sind, also ein Lin+ detektiert wurde.

Somit ergeben sich auf Grund der Entkopplung und der Berücksichtigung des Latchsignals weitere Zustände gegenüber dem halbentkoppelten, äquivalenten Automaten in Bild 4.30. Dies sagt zwar nicht grundlegend etwas über die spätere Performanz der Schaltung aus; allerdings erhöht sich mit steigender Komplexität der Schaltung die Wahrscheinlichkeit von „mutual exclusions“ und damit die Wahrscheinlichkeit von nicht behebbaren Fehlfunktionen beim Einsatz der Schaltung.

4.2.3.4 RDELC

In Bild 4.32 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Früh“-Protokoll gezeigt. Da die Latches standardmä-

ßig geöffnet sind, muss bei einem neuen Datum das Latch immer erst geöffnet und dann wieder geschlossen werden, bevor ein Ain+ gesendet werden darf. Somit muss jeweils das Latchsignal L beachtet werden. Im Speziellen existieren Zustandsübergänge Lin+/L-, die das Schließen der Latches repräsentieren.

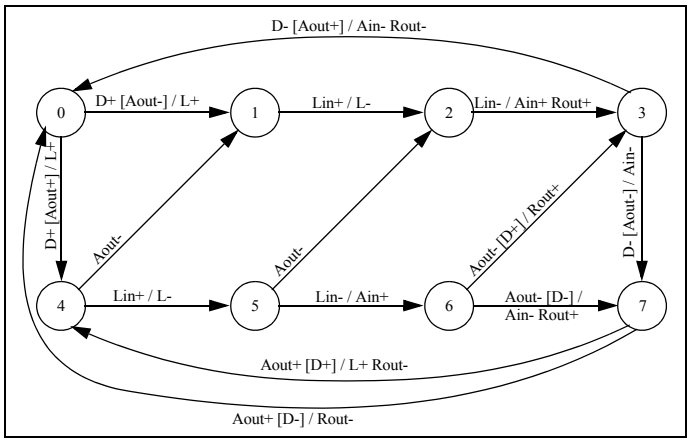


Bild 4.32: Zustandsgraph für die Kontrollstruktur RDELC

Speziell für die hier gewählte Realisierung eines RDELC Automaten gibt es eine mögliche Optimierungsmöglichkeit: Wird mit Verzögerungselementen für die Erzeugung des Done-Signals gearbeitet, kann man die Verzögerung statt in den Pfad von Rin nach D in den Pfad von L nach Lin einsetzen. Da Ain und Rout von Lin abhängen, werden diese auf keinen Fall zu früh gesetzt. Allerdings können sich insgesamt kürzere Wege durch die Kontrolllogik ergeben. Nämlich genau dann, wenn für die korrekte Funktion des Automaten sowieso Verzögerungen in den Pfad L -> Lin eingebaut werden müssen. Verschiebt man die Verzögerung Rin -> D nach L -> Lin fällt der kleinere der beiden Verzögerungswerte weg.

4.2.3.5 RDEFF

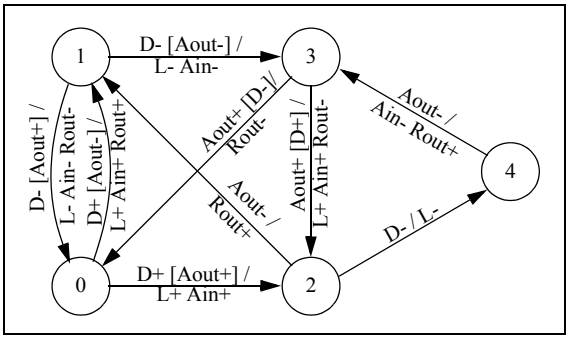


Bild 4.33: Zustandsgraph für die Kontrollstruktur RDEFF

In Bild 4.33 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für Flipflops im „Früh“-Protokoll gezeigt. Auf Grund des Aufbaus eines Flipflops aus einem Master- und einem Slave-Latch wird ein Datum mit einer Flanke (hier der steigenden) des Latchsignals gespeichert. Beruhend auf dieser Tatsache vereinfacht sich der Automat gegenüber den

äquivalenten Automaten für standardmäßig geschlossene oder standardmäßig geöffnete Latches (vergleiche Bild 4.32 und Bild 4.31).

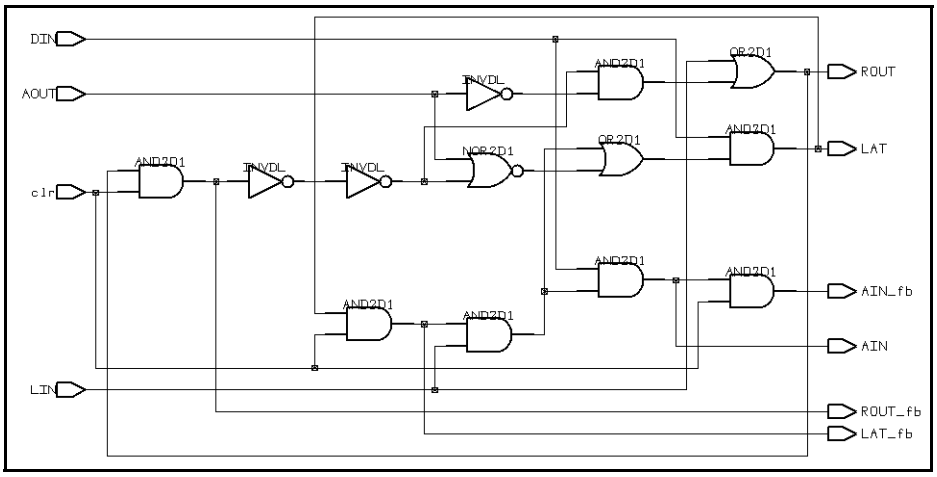


Bild 4.34: Implementierung der Schaltung RDEFF

In der Vereinfachung der Kontrollschaltung liegt somit das Potential der Verwendung von Flipflops. Diese besitzen eine größere Fläche und die Durchlaufzeit der Daten ist ebenfalls größer; allerdings kann der vereinfachte Kontrollautomat in Einzelfällen zu einer besseren Performanz der Gesamtschaltung führen.

Die Implementierung der RDEFF Kontrollstruktur ist in Bild 4.34 gezeigt. Es sind ein „low"-aktives Resetsignal und die Feedback Option realisiert worden. Man sieht im Speziellen die von DGC automatisch eingefügten Verzögerungen (hier ein Inverterpaar) für das Zustandssignal ROUT_fb. Ansonsten ist auch diese Implementierung relativ kompakt, da die drei Ausgangssignale als Zustandssignale völlig ausreichen.

4.2.3.6 RUWLO

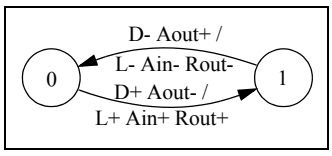


Bild 4.35: Zustandsgraph für die Kontrollstrukturen RUWLO und RUBFF

In Bild 4.35 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Weit"-Protokoll gezeigt. Dieser ist identisch für eine request-activated, gekoppelte Kontrollschaltung für Flipflops im „Breit"-Protokoll. Auf Grund der Kopplung muss immer auf die Beendigung/Start des laufenden Protokolls am Ausgang (Aout-/Aout+) gewartet werden. Durch diese Bedingung ist die Schaltung im Einsatz prinzipiell langsamer (maximal enthält nur jede zweite Stufe gültige Daten), aber der Automat und damit

auch die Schaltung sind einfach. Damit sind die Laufzeiten durch diese Schaltung und der Flächenbedarf gering.

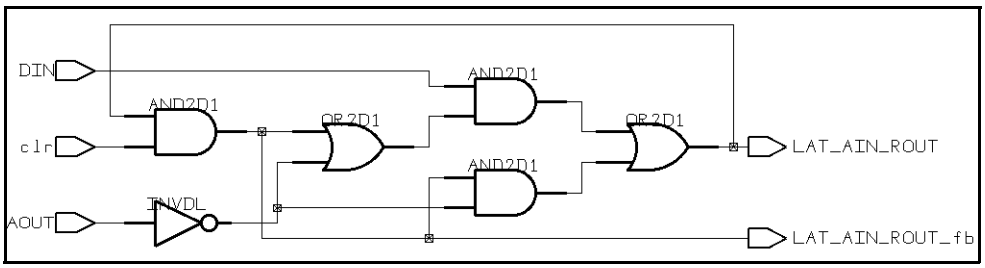


Bild 4.36: Implementierung der Schaltungen RUWLO und RUBFF

Bild 4.36 zeigt die Implementierung für den Automaten nach Bild 4.33. Da die Signale L, Ain und Rout in dem Automaten immer gleichzeitig gesetzt werden, sind sie in der Realisierung zu einem Signal LAT_AIN_ROUT zusammengefasst worden. Dieses Signal reicht als einziges Zustandssignal (LAT_AIN_ROUT_fb) aus. Somit ist auch die technologische Implementierung sehr einfach, klein und schnell.

4.2.3.7 RSWLO

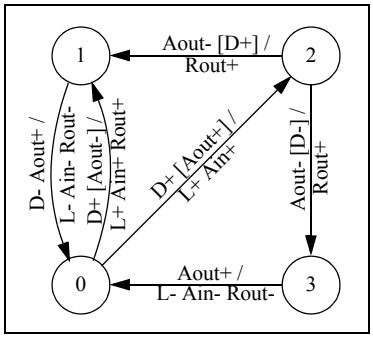


Bild 4.37: Zustandsgraph für die Kontrollstruktur RSWLO

In Bild 4.37 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Weit“-Protokoll gezeigt. Durch den zusätzlichen Freiheitsgrad der Halb-Entkopplung (Ain+ vor Aout- erlaubt) erweitert sich der Automat von Bild 4.35 um zwei Zustände.

Eine Konsequenz dieser Erweiterung ist, dass die drei Ausgangssignale nicht mehr immer gleichzeitig gesetzt werden. Dies gilt nun nur mehr für L und Ain, die in der Implementierung nach Bild 4.38 zu einem Signal (LAT_AIN) zusammengefasst worden sind. Die Implementierung vergrößert sich gegenüber der in Bild 4.36 um zwei Zustandssignale (Aufteilung der Ausgangssignale und ein zusätzliches Zustandssignal). Entsprechend wächst der Flächenbedarf und die Abhängigkeiten zwischen den Signalen.

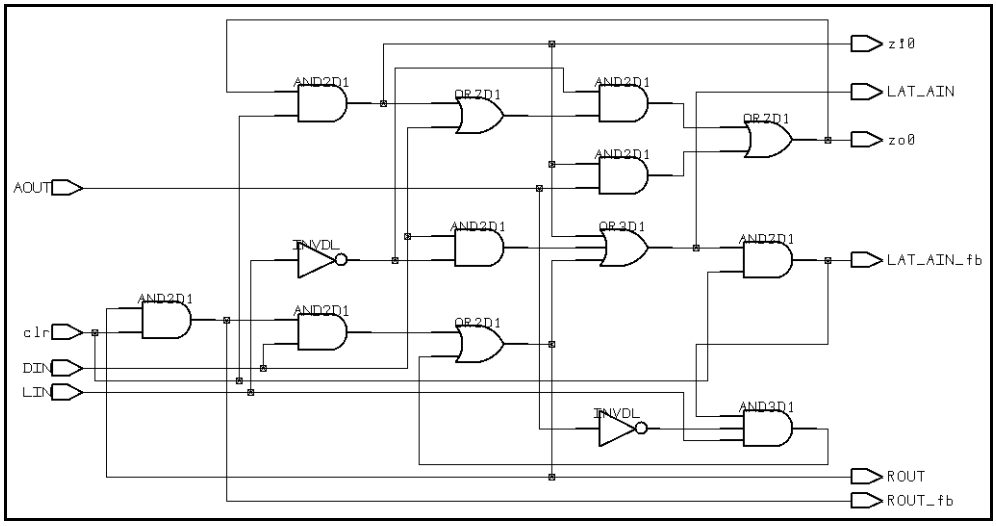


Bild 4.38: Implementierung der Schaltungen RSWLO

4.2.3.8 RDWLO

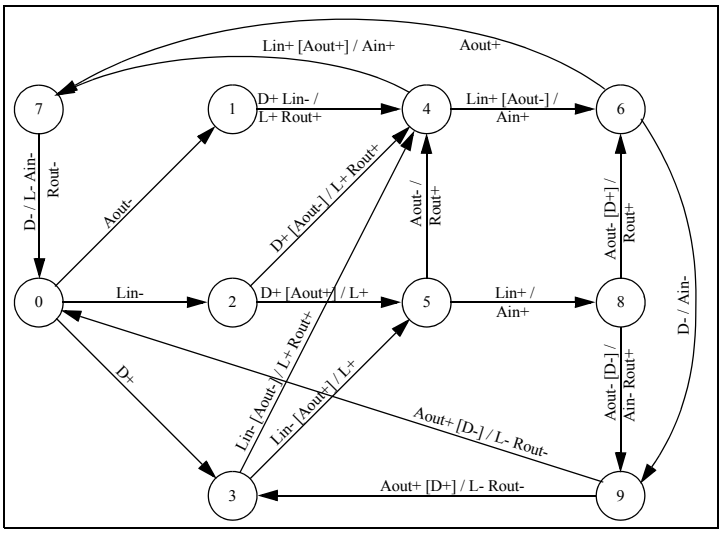


Bild 4.39: Zustandsgraph für die Kontrollstruktur RDWLO

In Bild 4.39 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Weit“-Protokoll gezeigt. Durch die Entkopplung muss das Latchsignal L wieder explizit detektiert werden. Der Automat vergrößert sich gegenüber dem nach Bild 4.37 mehr als um den Faktor 2, so dass hier kein wesentlicher Vorteil in der späteren Performanz zu erwarten ist.

4.2.3.9 RSWLC

In Bild 4.40 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Weit“-Protokoll gezeigt. Gegenüber der Variante mit standardmäßig geöffneten Latches (Bild 4.37) muss hier wiederum explizit das Öffnen

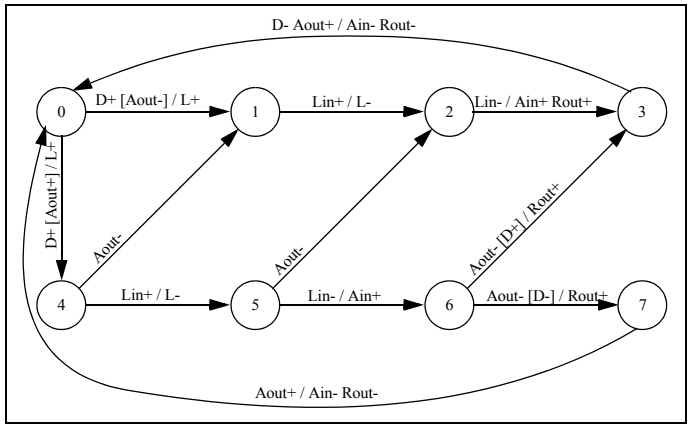


Bild 4.40: Zustandsgraph für die Kontrollstruktur RSWLC

(Lin+) und Schließen (Lin-) der Latches detektiert werden. Dadurch vergrößert sich der Automat auf 8 Zustände.

4.2.3.10 RDWLC

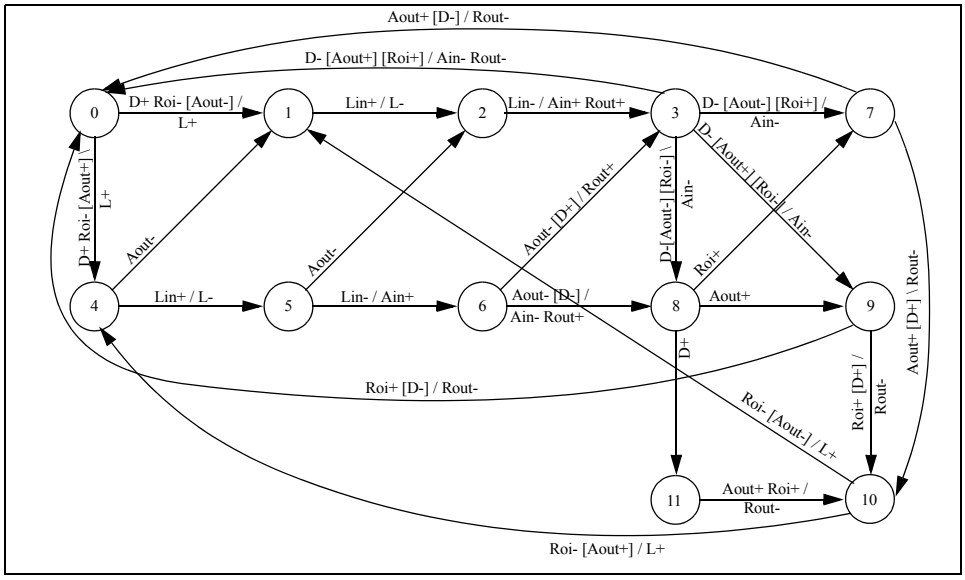


Bild 4.41: Zustandsgraph für die Kontrollstruktur RDWLC

In Bild 4.41 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Weit“-Protokoll gezeigt. Der zusätzliche Freiheitsgrad wirkt sich in einer Vergrößerung des Automaten gegenüber Bild 4.40 aus. Dadurch wird die Implementierung signifikant größer und die Wahrscheinlichkeit von „mutual exclusions“ steigt.

4.2.3.11 RSWFF

In Bild 4.42 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für Flipflops im „Weit“-Protokoll gezeigt. Gegenüber Bild 4.37 ergibt sich hier nur ein zusätz-

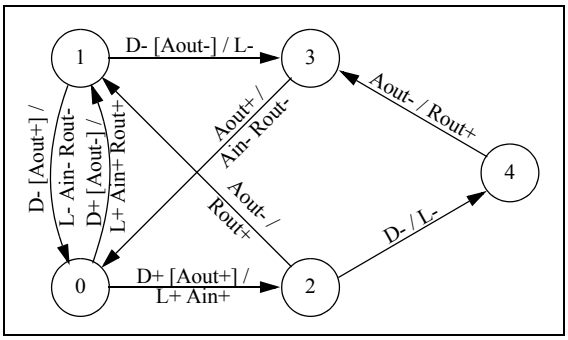


Bild 4.42: Zustandsgraph für die Kontrollstruktur RSWFF

licher Zustand. Dieser ergibt sich aus der Tatsache, dass bei positiv-flanken-getriggerten Flipflops die fallende Flanke zu einem selbst gewähltem Zeitpunkt erfolgen kann. Zur Vereinfachung wurde das Signal L direkt von D abhängig gemacht, welches zu dem zusätzlichen Zustand 4 führt.

4.2.3.12 RDWFF

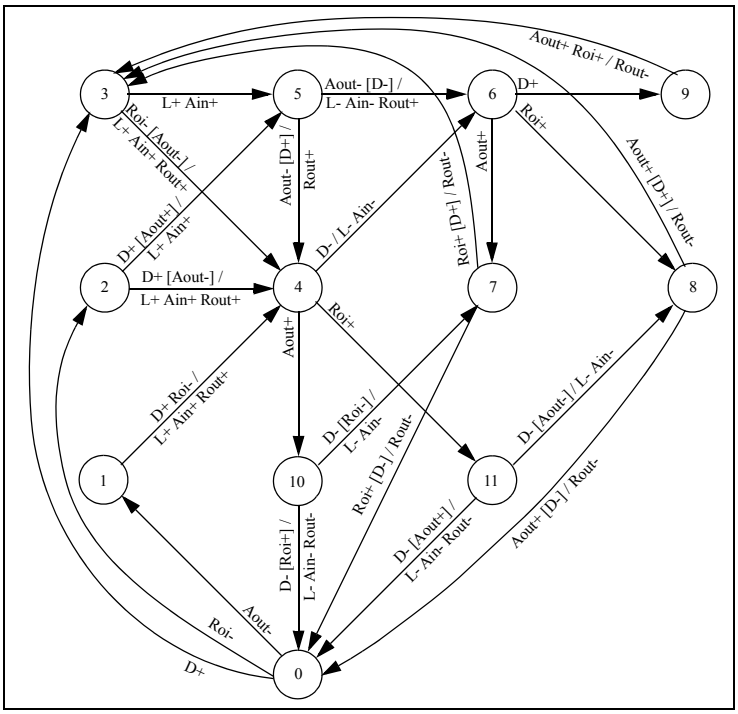


Bild 4.43: Zustandsgraph für die Kontrollstruktur RDWFF

In Bild 4.43 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für Flipflops im „Weit“-Protokoll gezeigt. Die Entkopplung führt wiederum zur Vergrößerung des Automaten und entsprechenden Gefahren von „mutual exclusions“.

4.2.3.13 RUBLO

In Bild 4.44 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit“-Protokoll gezeigt. In dem „Breit“-Protokoll muss

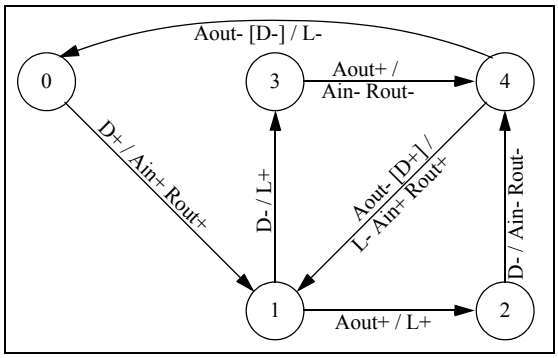


Bild 4.44: Zustandsgraph für die Kontrollstruktur RUBLO

das Datum bis zum Ende des jeweiligen Protokolls stabil anliegen. Somit ergeben sich gegenüber Bild 4.35 zusätzliche Bedingungen, die zu einer Vergrößerung des Automaten führen. So darf das Latchsignal nicht zurückgenommen werden, bevor nicht Aout- detektiert wird.

In Bild 4.44 wird ausgenützt, dass die Vorgängerstufe die Daten lange stabil hält. Somit kann das Datum zu einem späteren, für eine kleine Implementierung günstigeren Zeitpunkt übernommen werden. Das heißt L+ erfolgt nicht direkt mit einem detektiertem D+.

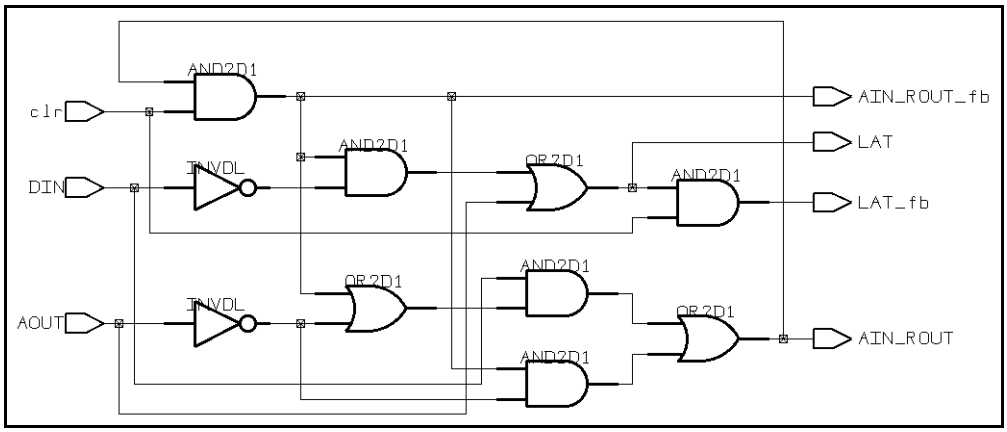


Bild 4.45: Implementierung der Schaltungen RUBLO

Wie in Bild 4.45 ersichtlich ist, sind für die Implementierung die Ausgangssignale als Zustandssignale ausreichend. Somit ergibt sich eine kompakte und schnelle Realisierung für diese Kontrollschaltung.

4.2.3.14 RSBLO

In Bild 4.46 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit“-Protokoll gezeigt. Dieser Automat benötigt lediglich einen Zustand mehr als der eingeschränktere gekoppelte Automat nach Bild 4.44.

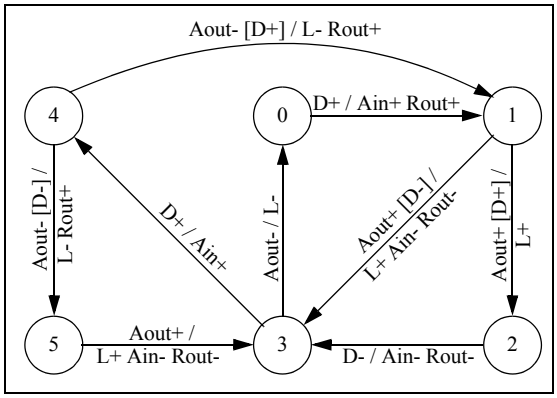


Bild 4.46: Zustandsgraph für die Kontrollstruktur RSBLO

4.2.3.15 RDBLO

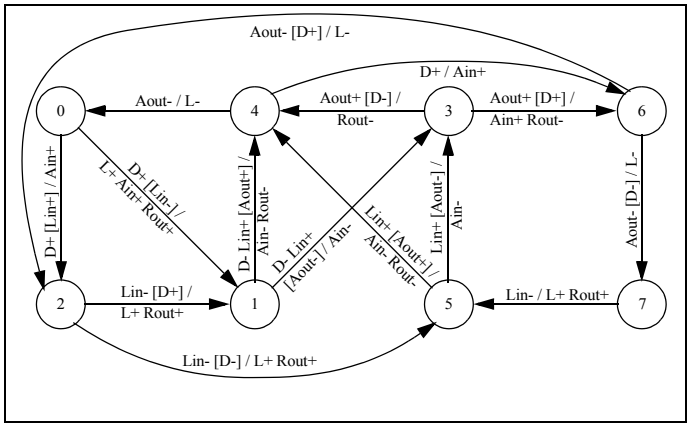


Bild 4.47: Zustandsgraph für die Kontrollstruktur RDBLO

In Bild 4.47 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit“-Protokoll gezeigt. Die Entkopplung führt wiederum dazu, dass das Latchsignal L explizit detektiert werden muss. Der größere Automat führt wieder zu einer größeren Wahrscheinlichkeit von „mutual exclusions“.

4.2.3.16 RUBLC

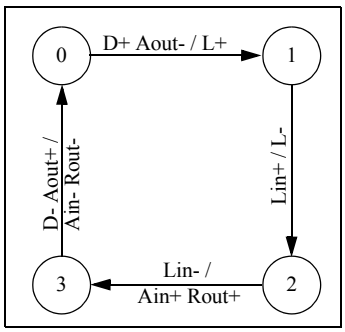


Bild 4.48: Zustandsgraph für die Kontrollstruktur RUBLC

In Bild 4.48 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Gegenüber der Variante mit standardmäßig geöffneten Latches (RUBLO, vergleiche Bild 4.44) vereinfacht sich der Automat um einen Zustand. Dies liegt daran, dass hier strikt auf das Latchsignal gewartet werden muss und somit weniger Freiheitsgrade vorliegen.

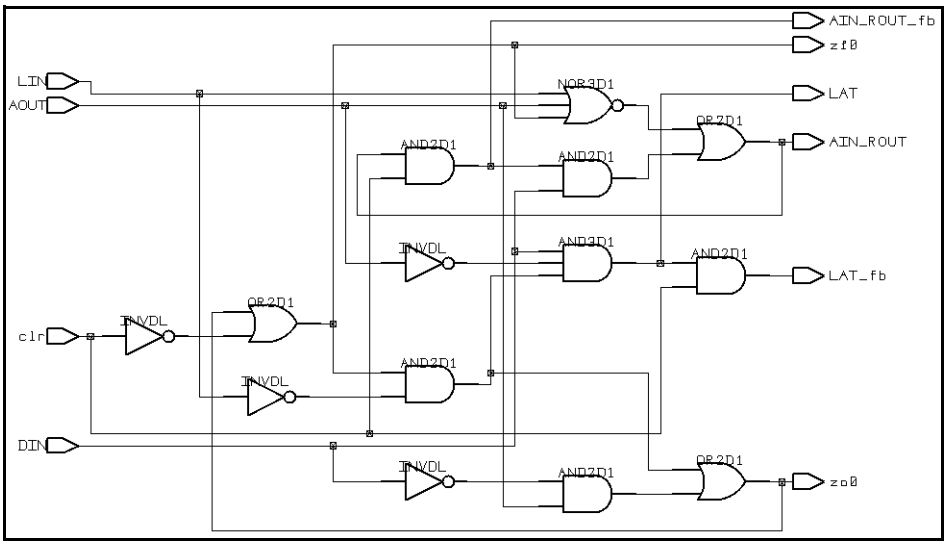


Bild 4.49: Implementierung der Schaltungen RUBLC

Bild 4.49 zeigt die Implementierung für den Automaten nach Bild 4.48. Interessant ist hierbei, dass die Implementierung zwei Gatter mehr benötigt, als die der RUBLO Schaltung (vergleiche Bild 4.45). Man erkennt hieran, dass man nicht direkt von der Komplexität des Automaten auf die Größe und entsprechende Performanz der Implementierung des Automaten schließen kann. Tendenziell wird ein größerer Automat auch in einer größeren und vermutlich „langsameren“ Schaltung resultieren. Es wird aber immer Ausnahmen geben.

4.2.3.17 RSBLC

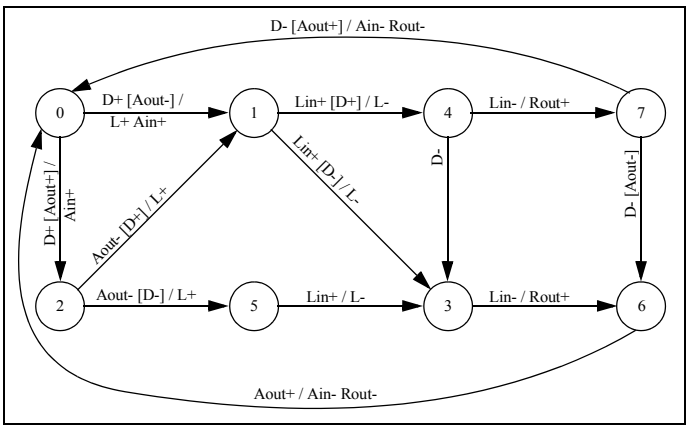


Bild 4.50: Zustandsgraph für die Kontrollstruktur RSBLC

In Bild 4.50 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Durch den zusätzlichen Freiheitsgrad (Ain+ ohne Aout- erlaubt) ergibt sich gegenüber Bild 4.48 ein entsprechend größerer Automat.

4.2.3.18 RDBLC

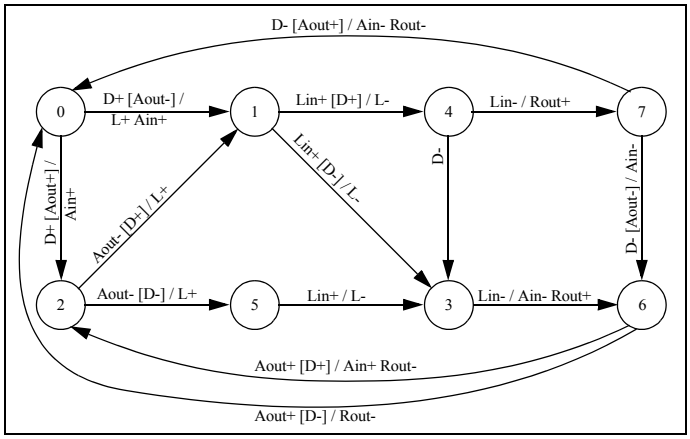


Bild 4.51: Zustandsgraph für die Kontrollstruktur RDBLC

In Bild 4.51 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Interessant ist hier, dass der Automat nicht wesentlich komplexer wird als der Automat der RSBLC Schaltung nach Bild 4.50. Es kommt nur ein weiterer Zustandsübergang vom Zustand 6 zum Zustand 2 hinzu.

4.2.3.19 RUBFF

Der Automat der einer request-activated, gekoppelten Kontrollschaltung für Flipflops im „Breit“-Protokoll gleicht der einer request-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Weit“-Protokoll (siehe Bild 4.35). Durch die Tatsache, dass Flipflops nur Daten mit der (hier positiven) Flanke des Latch-Signals übernehmen, ergibt sich durch den Automaten nach Bild 4.35 automatisch das Breitprotokoll. Somit kann dieser Automat sowohl für die Realisierung einer RUWLO als auch einer RUBFF Schaltung verwendet werden.

4.2.3.20 RSBFF

In Bild 4.51 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für Flipflops im „Breit“-Protokoll gezeigt. Der Automat ist im Vergleich zu der Variante mit standardmäßig geschlossenen Latches halb so groß. Dies liegt daran, dass hier das Latch-Signal nicht explizit abgeprüft werden muss. Allerdings sollte das Latch-Signal in der Realisierung mit dem Rout ein gemeinsames Signal bilden. Ansonsten läuft man Gefahr, dass das Rout Signal schneller als das Latch-Signal generiert wird und somit zu Fehlfunktionen führt.

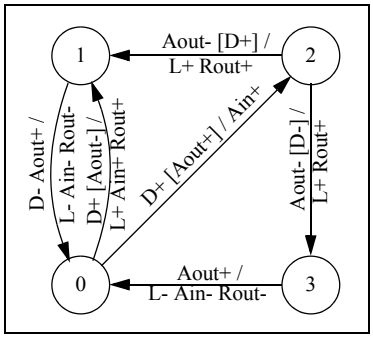


Bild 4.52: Zustandsgraph für die Kontrollstruktur RSBFF

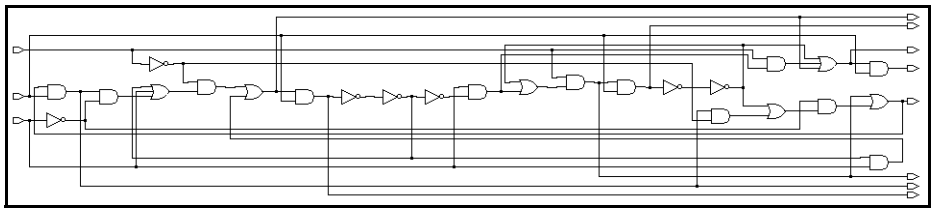


Bild 4.53: Implementierung der Schaltungen RSBFF

Die Implementierung der RSBFF Schaltung ist in Bild 4.53 gezeigt. Man erkennt, dass die Implementierung trotz des einfachen Automaten nicht laufzeitoptimal ist. Die Inverterpaare in der Schaltung zum Beispiel sind notwendig, um auftretende Zeitbedingungen zu erfüllen. Die transversale Ausdehnung lässt erkennen, dass die maximale Laufzeit über der anderer - als Automat - ähnlich kompakten Lösungen liegen wird.

4.2.3.21 RDBFF

In Bild 4.54 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für Flipflops im „Breit“-Protokoll gezeigt. Der Automat nimmt nur wenig an Komplexität gegenüber der halbentkoppelten Variante zu (Bild 4.52).

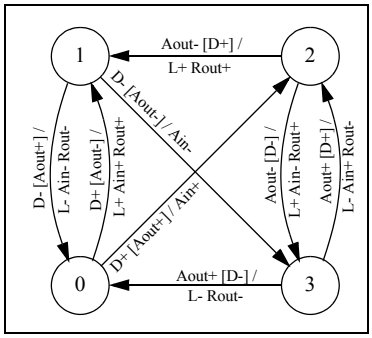


Bild 4.54: Zustandsgraph für die Kontrollstruktur RDBFF

Allerdings ist die Verwendung dieser Realisierung einer RDBFF Schaltung wiederum an Bedingungen geknüpft. Um das Latch-Signal nicht explizit überprüfen zu müssen, muss diese zusammen mit Rout ein Signal bilden. Somit ist sichergestellt, dass Rout nicht wesentlich vor

dem Latch-Signal generiert wird. Eine zweite Bedingung ergibt sich aus dem zeitkritischen Übergang von Zustand 2 zu 3. Hier darf Ain- nicht wesentlich vor L+ erzeugt werden. Liegt nämlich in der Vorgängerstufe schon das nächste Datum bereit, kann es passieren, dass sich dieses neue Datum so schnell in der aktuellen Stufe ausbreitet, dass es vor dem Wirksamwerden des L+ Wechsels schon die Eingänge der Flipflops erreicht. Somit würden falsche Werte übernommen werden. Notfalls müssen dem Signal Ain also Verzögerungsglieder nachgeschaltet werden.

4.2.3.22 AUBLO

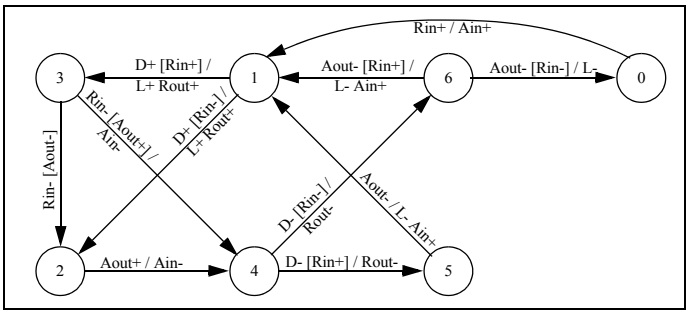


Bild 4.55: Zustandsgraph für die Kontrollstruktur AUBLO

In Bild 4.55 ist der Automat für eine acknowledge-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit"-Protokoll gezeigt. Acknowledge-activated Schaltungen haben generell den Vorteil, dass an die Vorgängerstufe eine schnelle Antwort erfolgt. Wie man aber im Vergleich zu der request-activated Variante (Bild 4.44) erkennt, sind die Automaten generell etwas komplexer. Dies liegt daran, dass nun nicht nur das „Erledigt"-Signal (D) sondern auch das Request-Signal Rin berücksichtigt werden muss. Somit erhält man ein paar zusätzliche Freiheitsgrade, die in einem entsprechenden Zuwachs des Automaten resultieren.

4.2.3.23 ASBLO

In Bild 4.56 ist der Automat für eine acknowledge-activated, halbentkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit"-Protokoll gezeigt. Auch hier ist zu erkennen, dass der Automat deutlich komplexer ist, als die request-activated Variante (Bild 4.46).

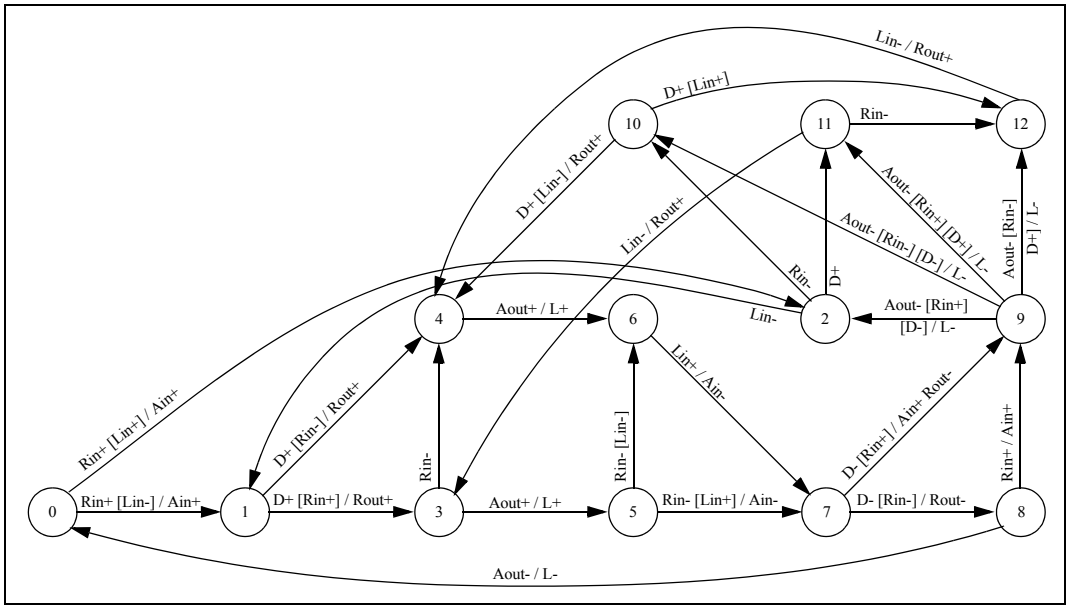


Bild 4.56: Zustandsgraph für die Kontrollstruktur ASBLO

4.2.3.24 ADBLO

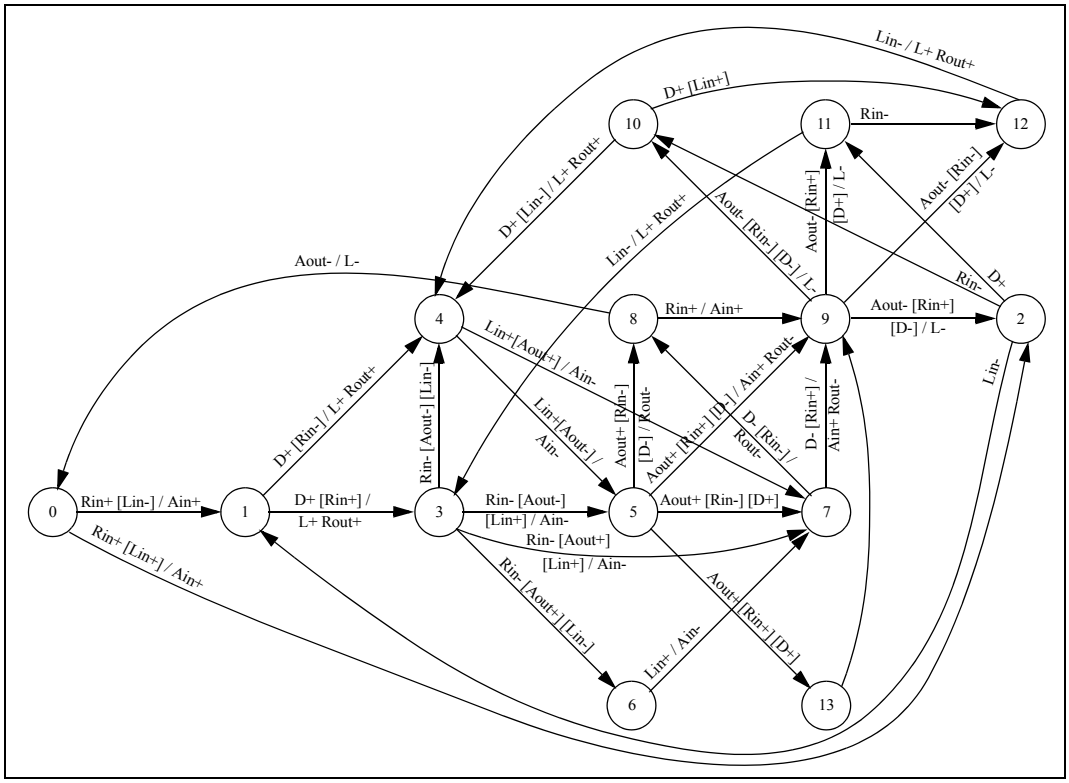


Bild 4.57: Zustandsgraph für die Kontrollstruktur ADBLO

In Bild 4.57 ist der Automat für eine acknowledge-activated, entkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit“-Protokoll gezeigt. Auch hier ist zu erkennen, dass der Automat deutlich komplexer ist, als die request-activated Variante (Bild 4.47).

4.2.3.25 AUBLC

In Bild 4.58 ist der Automat für eine acknowledge-activated, gekoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Dieser Automat ist für die acknowledge-activated Automaten einer der kompaktesten und somit interessanter als die restlichen acknowledge-activated Varianten.

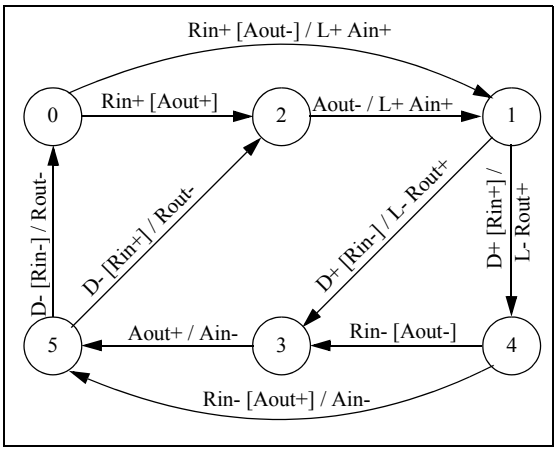


Bild 4.58: Zustandsgraph für die Kontrollstruktur AUBLC

Die Implementierung in Bild 4.59 zeigt, dass die Ausgangssignale als Zustandssignale ausreichen. Somit ist auch die Implementierung relativ kompakt. Die durch die Grafik verdeutlichte Parallelität der Schaltung lässt erkennen, dass die Laufzeit durch die Schaltung nicht allzu schlecht im Vergleich zu den request-activated Schaltungen sein wird.

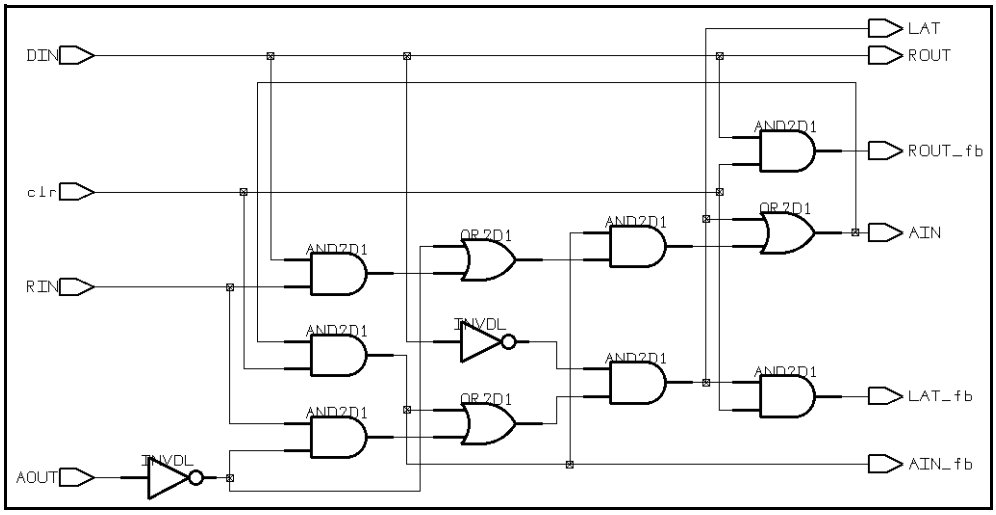


Bild 4.59: Implementierung der Schaltungen AUBLC

4.2.3.26 ASBLC

In Bild 4.60 ist der Automat für eine acknowledge-activated, halbentkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Wiederum ist der Automat komplexer, als die entsprechende request-activated Variante (Bild 4.50), auch wenn

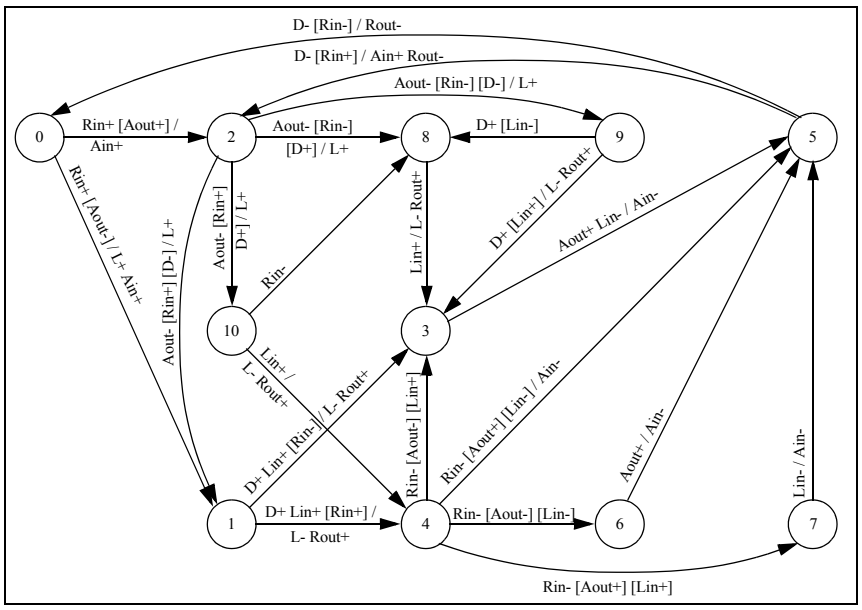


Bild 4.60: Zustandsgraph für die Kontrollstruktur ASBLC

hier der Zuwachs nicht so deutlich ist, wie bei den Varianten mit standardmäßig geöffneten Latches.

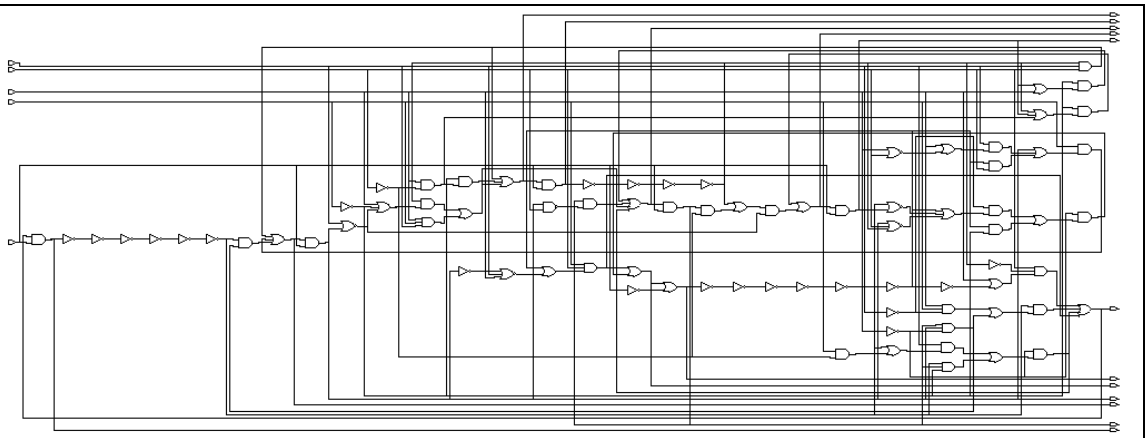


Bild 4.61: Implementierung der Schaltung ASBLC

Die Implementierung der ASBLC Schaltung ist in Bild 4.61 gezeigt. Sie dient zur Verdeutlichung der Zunahme der Schaltungskomplexität. Die größere Anzahl an Gattern und speziell der notwendigen Verzögerungsglieder (Inverterpaare) verdeutlicht schon, dass die Schaltung mehr Fläche und Laufzeit benötigen wird, als andere Kontrollschaltungen, speziell die der request-activated Schaltungen.

4.2.3.27 ADBLC

In Bild 4.62 ist der Automat für eine acknowledge-activated, entkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit“-Protokoll gezeigt. Durch geschicktes Zusammenfassen, von Übergangsbedingungen ist dieser Automat nicht wesentlich komplexer

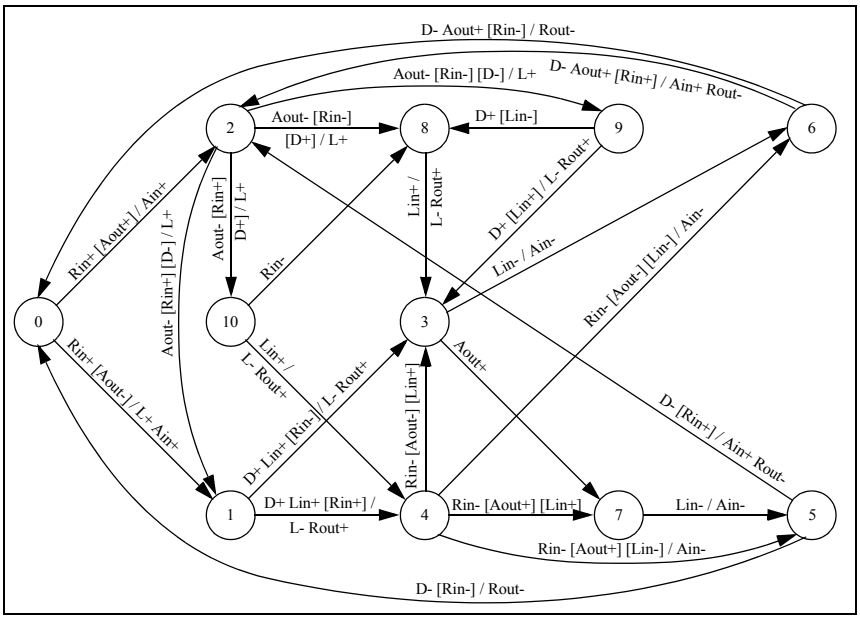


Bild 4.62: Zustandsgraph für die Kontrollstruktur ADBLC

als die halbentkoppelte Variante nach Bild 4.60. Trotzdem wird die Implementierung schlechter sein, da der größere Freiheitsgrad zu entsprechend größeren Implementierung führen wird.

4.2.3.28 AUBFF

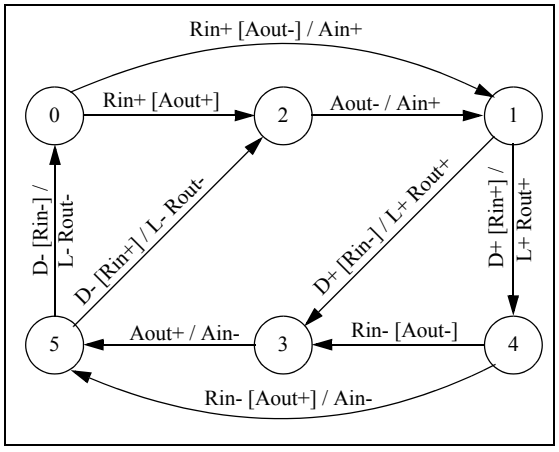


Bild 4.63: Zustandsgraph für die Kontrollstruktur AUBFF

In Bild 4.63 ist der Automat für eine acknowledge-activated, gekoppelte Kontrollschaltung für Flipflops im „Breit“-Protokoll gezeigt. Dies ist der kompakteste Automat der acknowledge-activated Kontrollschaltungen. Sein Vorteil liegt darin, dass die Signale L (Latch-Signal) und Rout zu einem Signal zusammengefasst werden können. Die Implementierung vereinfacht sich dadurch, da zwischen L und Rout keine besonderen Zeitbedingungen auftreten.

Die Implementierung in Bild 4.64 zeigt, dass die Ausgangssignale als Zustandssignale ausreichen. Somit erhält man eine kompakte Realisierung einer AUBFF Schaltung. Inwieweit der Vorteil der früheren eingangsseitigen Antwort der acknowledge-activated Schaltungen durch

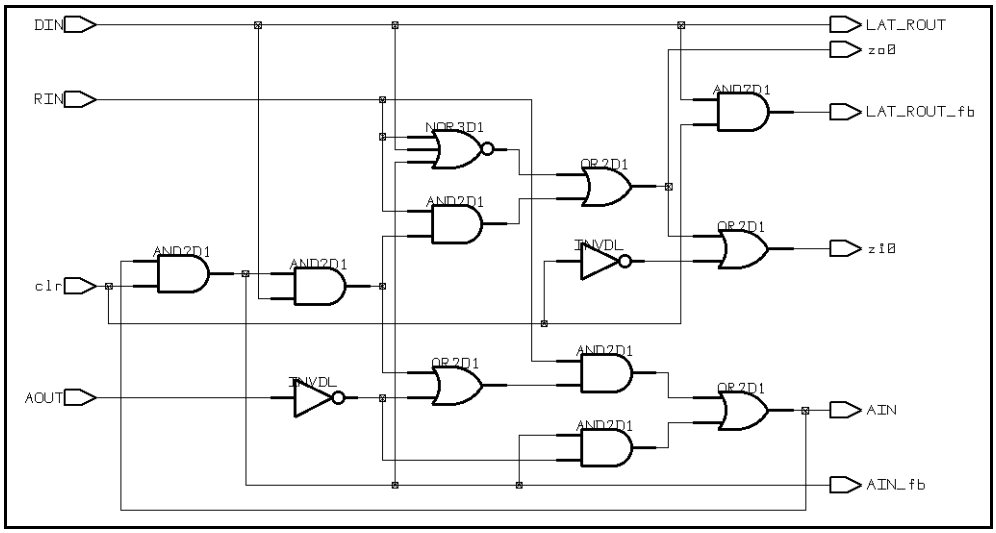


Bild 4.64: Implementierung der Schaltungen AUBFF

die größere Schaltung und damit größeren Laufzeiten als die request-activated Varianten (Kapitel 4.2.3.19) wieder aufgehoben werden, muss in Simulationen gezeigt werden.

4.2.3.29 ASBFF

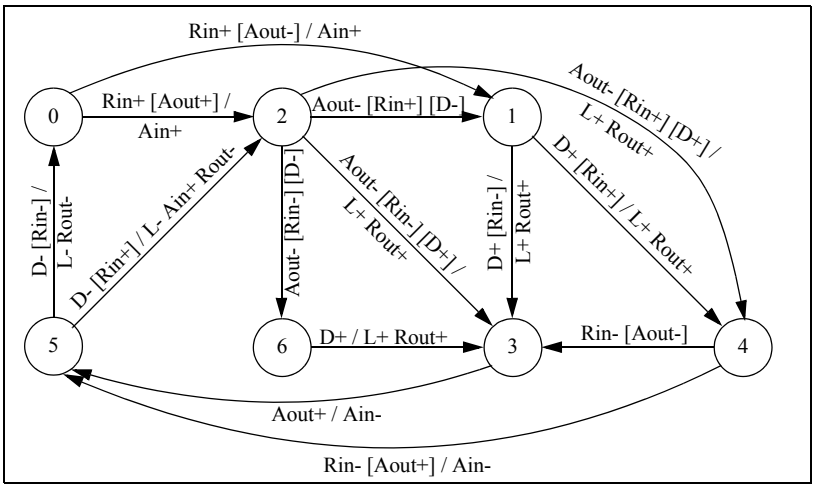


Bild 4.65: Zustandsgraph für die Kontrollstruktur ASBFF

In Bild 4.65 ist der Automat für eine acknowledge-activated, halbentkoppelte Kontrollschaltung für Flipflops im „Breit“-Protokoll gezeigt. Wiederum ist der Automat dieser Schaltung kleiner als der für die anderen Speicherarten, allerdings deutlich größer als der Automat der request-activated Variante. Somit ist der Gewinn durch die acknowledge-activation nicht zwingend.

4.2.3.30 ADBFF

In Bild 4.66 ist der Automat für eine acknowledge-activated, entkoppelte Kontrollschaltung für Flipflops im „Breit“-Protokoll gezeigt. Dieser Automat ist der komplexeste aller bisher vor-

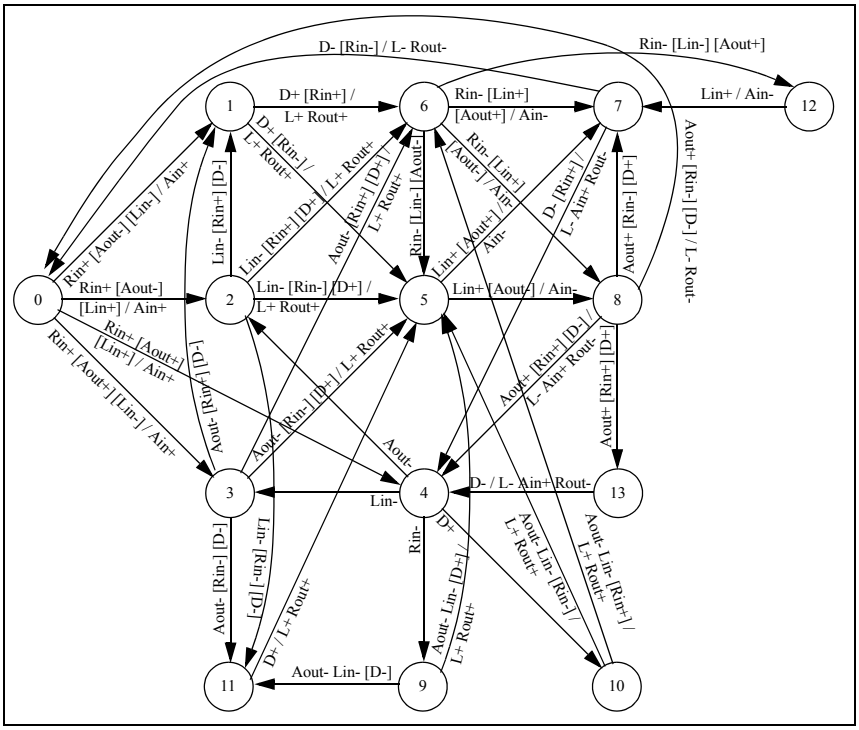


Bild 4.66: Zustandsgraph für die Kontrollstruktur ADBFF

gestellten Automaten. Somit ist kein Gewinn in der späteren Performanz dieser Schaltung zu erwarten. Auch eine Einsparung bei der Verlustleistung ist nicht zu erwarten.

4.2.4 Bewertung der Kontrollstrukturen

Bei der Vorstellung der möglichen Kontrollstrukturen wurden Unterscheidungskriterien zur Klassifizierung angeführt. An diese Kriterien sind verschiedene Erwartungen geknüpft. So erhofft man sich von einer acknowledge-activated Schaltung eine bessere Laufzeit als die vergleichbare request-activated Variante. Das gleiche erwartet man von den entkoppelten Schaltungen gegenüber den gekoppelten Schaltungen, sowie den standardmäßig geöffneten Latches gegenüber den anderen Speichervarianten. Auch bei dem Datengültigkeitsschema erwartet man beim Format „Früh“ einen höheren Datendurchsatz als beim Format „Breit“.

Die Simulationsergebnisse zeigen, dass sich diese Hoffnungen nicht immer erfüllen. Dies liegt zum einen daran, dass sich ein Kriterium allein nicht immer direkt als Vorteil bemerkbar macht, weil zum Beispiel der Aktivierungsmechanismus schwerwiegender als der Kopplungsgrad ist. Zum anderen führt das Einhalten eines Kriteriums zu komplexeren Automaten und damit zwangsläufig zu größeren und langsameren Kontrollschaltungen. Eine vollständige Entkopplung zweier Stufen führt zum Beispiel generell zu größeren Automaten. Die Simulationsergebnisse zeigen, dass die entkoppelten Pipelines dementsprechend häufig langsamer sind als die halbentkoppelten oder gekoppelten Varianten.

Ein weiterer schwerwiegender Faktor ist die sich ergebende Spezifikation des Zustandsgraphen. Lässt sich dieser in DGC geschickt optimieren, so erhält man eine kleinere und schnellere Realisierung. Als bestes Beispiel ist hierfür die Schaltung RDBFF (siehe 4.2.3.21) aufzuführen. In Kapitel 4.2.7 wird gezeigt, welchen Einfluss die Spezifikation auf das Endergebnis haben kann.

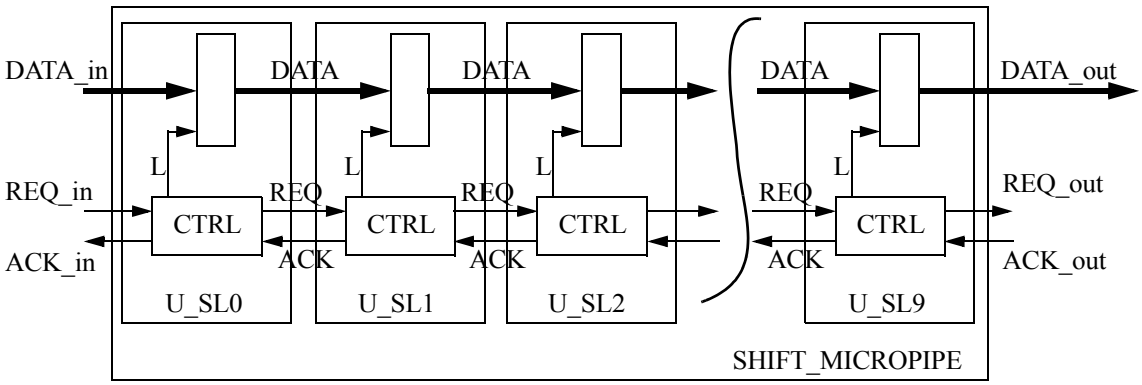


Bild 4.67: Mikropipeline mit Schieberegistern

Um die unterschiedlichen Kontrollstrukturen bewerten zu können wurden zwei verschiedene Pipelines erstellt. Die erste soll den Extremfall erfassen, dass jede Vorgängerstufe und auch jede Nachfolgerstufe in kürzestmöglicher Zeit reagiert. Zu diesem Zwecke wurde eine Pipeline augebaut, welche eingehende Daten über 10 Stufen weiterschiebt (Schieberegister). Dieser Fall ist in Bild 4.67 dargestellt.

Die zweite Mikropipeline wurde aus einzelnen Stufen mit unterschiedlicher durchschnittlicher Reaktionszeit aufgebaut. Die Stufen an sich führen eine Addition oder eine Substraktion aus. Um das Zusammenspiel besser zu prüfen, wurde am Anfang sowie am Ende eine langsame Stufe (U2_ADD127 beziehungsweise U11_ADD_IF) direkt einer schnellen Stufe (Schieberegister) nachgeschaltet. Dieser Fall ist in Bild 4.68 dargestellt.

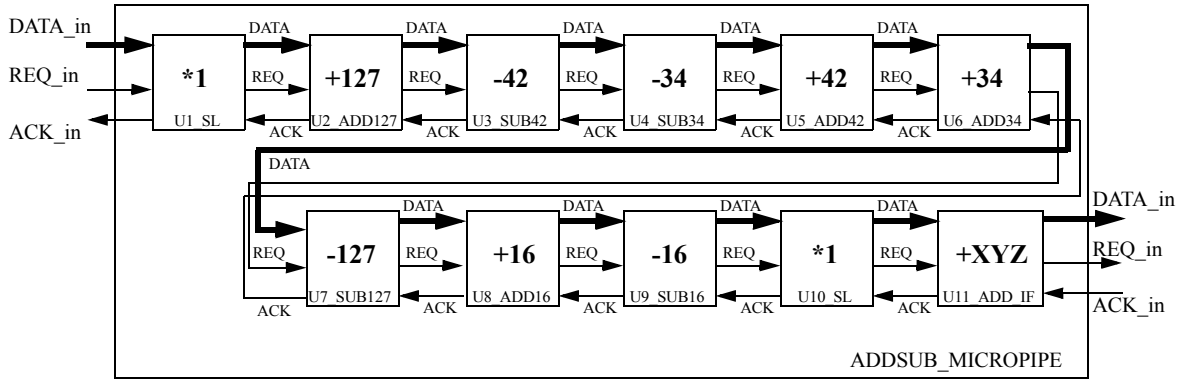


Bild 4.68: Mikropipeline mit Addierern und Subtrahierern

Um die unterschiedlichen Kontrollschaltungen untereinander und mit einer synchronen Realisierung vergleichen zu können, wurden jeweils drei verschiedene Simulationsläufe durchgeführt:

- FAST: In diesem Modus wird mit einem maximalen Datendurchsatz gearbeitet. Das heißt die Daten werden so schnell wie möglich an der Eingangsseite nachgeliefert und auch so schnell wie möglich an der Ausgangsseite abgenommen.

- 100MHZ: In diesem Simulationslauf werden die Daten mit einer Frequenz von 100MHz angelegt. Dadurch werden die Übergaben zwischen den schnellen Stufen verlangsamt. Somit verschieben sich die Flanken der eingangsseitigen Kontrollsignale zu denen der Ausgangsseite bei jedem Modul. Neben anderen Performanzwerten können in der FAST - Simulation noch nicht aufgedeckte Fehlerquellen aufgespürt werden.

- THIRD: Dieser Simulationslauf dient einerseits, um weitere Vergleichswerte für den Vergleich der asynchronen Schaltungen untereinander zu liefern. Andererseits dient es zum Vergleich mit der synchronen Realisierung. Die synchrone Schaltung wird in diesem Lauf mit einer für die asynchronen Schaltungen erreichbaren Frequenz getaktet, aber nur in jeder dritten Periode werden neue Daten angelegt. Hier wird der Fall gezeigt, dass in einer synchronen Schaltung im „Leerlauf“ durch den Takt ein nicht unerheblicher Energieverbrauch entsteht.

Als Kriterien für eine Bewertung der Güte der jeweiligen Kontrollschaltung werden die (absolute) Durchlaufzeit der Stimulationsvektoren, die auf eine Gesamtlaufzeit von 200 ns relativierte verbrauchte Energie[1] und die Fläche der Pipeline herangezogen. Um einen direkten Vergleich zu erleichtern, wird aus der Durchlaufzeit und der normierten verbrauchten Energie ein hier „Performanz“ benannter Bewertungsfaktor nach folgender Gleichung gebildet:

$$PERFORMANCE = (Time \cdot Energy) / 100 \qquad (4.11)$$

Des Weiteren werden von ASMOGEN automatisch so genannte „Wachhunde“ in die Simulationsmodule eingebaut, die das Einhalten der Zeitbedingungen (vergleiche Kapitel 4.1.5) überprüfen. Eine Verletzung einer der Zeitbedingungen führt aber nicht zwangsläufig zu einem Fehlverhalten. Alle in diesem Vergleich durchgeführten Simulationen führten zu einem korrekten Ergebnis. Allerdings sagen Verletzungen der Zeitbedingungen etwas über die Sicherheit der Schaltung aus. Nur wenn keine Zeitbedingungen verletzt werden, kann ein korrektes Funktionieren der Schaltung garantiert werden. Ansonsten bleibt ein Unsicherheitsfaktor bestehen.

Die vollständigen Ergebnistabellen sind in dem Anhang F einzusehen. In diesem Kapitel werden nur zur Bewertung relevante Auszüge dargestellt. In Tabelle 2 sind die jeweils 10 besten „FAST“-Simulationsergebnisse asynchronen Schiebeketten für die unterschiedlichen Bewertungsgrößen aufgelistet. Ist nur die Laufzeit in Zusammenhang mit dem Energiebedarf

1. Die verwendete Synthesesoftware Synopsys gibt einen Energieverbauch pro Zeiteinheit heraus (Power). Zur besseren Vergleichbarkeit wird das gewonnene Ergebnis auf eine feste Simulationslaufzeit normiert.

Tabelle 2: Beste „FAST“-Simulationsergebnisse der asynchronen Schiebeketten Implementierungen

Laufzeit (ns)		Power (µW)		Performanz		Fläche (μm^2)	
RUBLO	19.783	RUELO	1.0338	RUBLO	0.219	RUWLO	6585.70
AUBLO	20.115	AUBLC	1.0370	AUBLO	0.220	AUBLC	6667.00
ASBFF	21.357	RUWLO	1.0587	RUELO	0.222	RUELO	6748.29
AUBLC	21.513	AUBLO	1.0986	AUBLC	0.223	AUBLO	6910.89
RUELO	21.545	RUBLO	1.1109	AUBFF	0.279	ASBLC	6992.20
ADBLO	22.518	AUBFF	1.1596	RUWLO	0.279	RUBLO	6992.20
ADBFF	23.904	RUBFF	1.1597	RUBFF	0.304	RUBLC	7480.00
AUBFF	24.075	ASBLC	1.1931	ASBFF	0.343	RSWLO	7886.60
RSWFF	24.877	RUBLC	1.2640	ASBLC	0.356	RSBLC	8049.10
ASBLO	25.651	RSWLO	1.4295	ADBLO	0.367	ASBLO	8171.10

von Bedeutung, dann gibt die Spalte „Performanz“ die geeignetesten Kontrollschaltungen für asynchrone Pipelines mit wenig Logik zwischen den Speichern wider. Ist zusätzlich die Fläche von Bedeutung, so kehrt sich die Reihenfolge der ersten vier Kontrollschaltungen um (RUBLO bis AUBLC). Interessant ist, dass die acknowledge-activated Schaltungen im Mittel etwas schneller sind als die request-activated Schaltungen. Allerdings ist der Energiebedarf der request-activated Schaltungen im Mittel etwas geringer als bei den acknowledge-activated Schaltungen.

Tabelle 3: Beste „FAST“-Simulationsergebnisse der asynchronen Addieren-Subtrahieren-Ketten Implementierungen

Laufzeit (ns)		Power (µW)		Performanz		Fläche (μm^2)	
AUBLC	46.518	RUELO	1.6543	AUBLC	0.775	AUBLC	11537.25
ADBLO	52.512	AUBLC	1.6661	AUBLO	0.907	RUWLO	11724.25
AUBLO	52.843	RUWLO	1.6828	RUELO	0.931	RUELO	11903.11
ASBFF	53.670	AUBLO	1.7171	RUBLO	0.970	AUBLO	12049.45
AUBFF	54.069	RUBLO	1.7569	AUBFF	1.000	ASBLC	12138.87
RUBLO	55.243	AUBFF	1.8506	RUWLO	1.038	RUBLO	12236.44
ADBFF	56.122	RUBFF	1.8592	ASBLC	1.064	RUBLC	12724.24
RUELO	56.333	ASBLC	1.8817	RUBFF	1.143	RSWLO	13155.24
ASBLC	56.575	RUBLC	1.9897	ADBLO	1.214	RSBLC	13382.77
ASBLO	59.236	RSWLO	2.0941	ASBFF	1.274	ASBLO	13484.44

In Tabelle 3 sind die jeweils 10 besten „FAST“-Simulationsergebnisse asynchroner Addieren-Subtrahieren-Ketten für die unterschiedlichen Bewertungsgrößen aufgelistet. Ist nur die Laufzeit in Zusammenhang mit dem Energiebedarf von Bedeutung, dann gibt die Spalte „Performanz“ die geeignetesten Kontrollschaltungen für Mikropipelines mit viel Logik zwischen den Speichern wider. Die Einbeziehung der Flächenwerte ändert hier nichts an der Reihenfolge. Auch in diesen Beispielen sind die acknowledge-activated Schaltungen im Mittel schneller, während der Energiebedarf der request-activated Schaltungen im Mittel ein klein wenig geringer ist.

Betrachtet man Tabelle 3 und Tabelle 2 zusammen, so erkennt man, dass die Kontrollschaltung AUBLC am geeignetesten ist, wenn man im gesamten Design nur eine Variante einer Kontrollschaltung benutzen möchte. Damit ist aber nicht gesagt, das die anderen Varianten nutzlos sind. Speziell wenn die Laufzeit durch die Logik sehr groß ist, kann eine „langsame“ Kontrollschaltung von Vorteil sein, da hier weniger zusätzliche Verzögerungselemente eingebaut werden müssen.

Tabelle 4: Simulationsergebnisse der synchronen Implementierung

(Fläche)	Schieben (6504.79)			Addieren/Subtrahieren (9842.66)		
Simulation	Zeit (ns)	Power (µW)	Performanz	Zeit (ns)	Power (µW)	Performanz
Fast	8.876	1.1322	0.10	44.001	1.5701	0.69
100MHz	175.126	1.1607	2.03	185.126	1.7300	3.20
Third	82.426	2.0542	1.69	107.276	2.5642	2.75

Die Verhältnisse der Simulationsergebnisse der asynchronen Schaltungen untereinander in den verschiedenen Simulationen „FAST“, „100MHZ“ und „THIRD“ ändern sich nicht. Allerdings kann man anhand dieser unterschiedlichen Simulationen abschätzen, ab wann die asynchronen Implementierungen Vorteile bieten.

Tabelle 4 zeigt die Simulationsergebnisse der synchronen Implementierungen der zwei verschiedenen Pipelines an. In der Simulation „FAST“ mit maximalen Datendurchsatz ist die synchrone Variante wie zu erwarten schneller. Die synchrone Schiebekette ist gut doppelt so schnell wie die schnellste asynchrone Implementierung (RUBLO) und fast 2,5 mal so schnell wie die favorisierte Kontrollschaltung AUBLC. Die Addieren-Subtrahieren-Pipeline ist allerdings nur 2,5 ns schneller als die asynchrone Variante mit der Kontrollschaltung AUBLC. Das sind gerade mal 6% Vorsprung.

In Tabelle 5 sind nun die Leistungswerte der synchronen Implementierungen mit denen der asynchronen AUBLC-Implementierungen gegenübergestellt. Nicht angezeigt ist der Flächenzuwachs der asynchronen Varianten, der bei 2,5% (Schiebekette) beziehungsweise 14,7% (Addieren-Subtrahieren) gegenüber den synchronen Varianten liegt. Des Weiteren besitzt die

Tabelle 5: Vergleich der Leistungswerte der synchronen Implementierungen mit der asynchronen AUBLC Implementierungen

		Schieben			Addieren/Subtrahieren		
		Fast	100Mhz	Third	Fast	100Mhz	Third
Zeit (ns)	SYNCHRON	8.876	175.126	82.426	44.001	185.126	107.276
	AUBLC	21.513	77.909	62.209	45.481	85.224	84.524
	Gewinn	**-142,4%**	**55,5%**	**24,5%**	**-5,7%**	**53,8%**	**21,2%**
Power (µW)	SYNCHRON	1.1322	1.1607	2.0542	1.5701	1.7300	2.5642
	AUBLC	1.0370	1.0415	1.0415	1.6543	1.6630	1.6458
	Gewinn	**8,4%**	**10,3%**	**49,3%**	**-5,4%**	**3,9%**	**35,8%**
Performanz	SYNCHRON	0.10	2.03	1.69	0.69	3.20	2.75
	AUBLC	0.223	0.811	0.647	0.771	1.421	1.391
	Gewinn	**-123%**	**60%**	**61,7%**	**-11,7%**	**55,6%**	**49,4%**

synchrone Schiebekette einen erheblich größeren maximalen Datendurchsatz. Interessanterweise ergibt sich aber für die AUBLC Schaltung in der reinen Schiebekette ein geringerer Energiebedarf. Dies liegt an der Tatsache, dass in der asynchronen Variante nur die tatsächlich aktiven Stufen Energie verbrauchen. In der synchronen Variante werden von Anfang an (Wegnahme des Reset-Signals) alle Speicher getaktet. Somit verbrauchen hier auch die inaktiven Stufen Energie. Dieser kleine Vorteil geht bei der Addieren/Subtrahieren Pipeline unter, da hier für die in der Logik entstehenden Laufzeit Verzögerungsglieder in den Pfaden der Kontrolllogik eingefügt werden müssen; diese sorgen für einen entsprechend höheren Energiebedarf.

Liegt der Systemtakt fest zum Beispiel bei 100MHz, so erhält man bei den asynchronen Realisierungen einen deutlichen Laufzeitvorteil. Dies ist naheliegend, da die eigentliche Laufzeit durch die jeweilige Logik unter 10 ns liegt. Das heißt man kann diesen Vorteil nutzen, wenn man zum Beispiel eine asynchrone Pipeline in eine synchrone Schaltung einbaut, welche mit einem relativ langsamen Systemtakt arbeitet. Der in der Tabelle 5 gezeigt Vorteil beim Energiebedarf wird sich bei der Einbettung in einen synchronen Entwurf nicht halten können, da hier noch Schnittstellenschaltungen (vergleiche Kapitel 4.2.9) eingebaut werden müssen, deren Energiebedarf etwas höher ist.

Der interessanteste Einsatzfall einer asynchronen Pipeline ist der hier „Third“ genannte Simulationsfall: Die Daten kommen nicht mit einer konstanten (hohen) Rate. Der synchrone Systemtakt wird im Normalfall so hoch wie nötig gewählt. Liegt aber im realen Einsatz der Schaltung nicht immer der maximale Datendurchsatz vor, so verbraucht der Takt in der synchronen Realisierung unnötig Energie. Liegen zum Beispiel in der synchronen Variante nur mit jedem dritten Takt neue Daten an, so erzielt die AUBLC Schaltung bei gleicher Eingangsdatenrate einen niedrigeren Energieverbrauch von über 49% (Schiebekette) beziehungsweise 35%

(Addieren/Subtrahieren). Auf Grund der Taktunabhängigkeit ist die asynchronen Pipeline auch schneller (24,5% beziehungsweise 21,2%). Da hier aber am Eingang die Daten mit der gleichen Rate anliegen, ist dies vor allem dann ein Vorteil, wenn das Endergebnis möglichst schnell vorliegen soll. So spart man sich entsprechende Verzögerungsschaltungen in einem eventuell parallel zur Pipeline laufenden Pfad.

Insgesamt ist festzuhalten, dass es nicht eine „beste“ asynchrone Kontrollschaltung gibt. Die unterschiedlichen Anforderungen, die sich aus der entsprechenden Anwendung ergeben, führen dazu, dass verschiedene Kontrollschaltungen die jeweils beste Performanz leisten. Ein weiterer Punkt ist der Einfluss der Art der Automatenspezifikation und die Leistungsfähigkeit der verwendeten Optimierungsalgorithmen beim Umsetzen der Automatenspezifikation in eine Schaltung. Nichtsdestotrotz erscheint im Rahmen dieser Arbeit die Kontrollschaltung AUBLC am geeignetesten, als Standardkontrollschaltung verwendet zu werden; sie liefert unabhängig von der Einsatzart immer gute Leistungswerte. Bis auf die absolute maximale Datenrate kann eine asynchrone Mikropipeline mit AUBLC Kontrollschaltungen mit einer synchronen Realisierung mithalten. Im Bereich des Energiebedarfs ist die asynchrone Realisierung vor allem dann von Vorteil, wenn die Pipeline nicht mit einer konstanten Rate mit neuen Daten gespeist wird.

4.2.5 Optimierte Automaten für das Breite Datenprotokoll

In dem „Breit“-Protokoll liegen die Daten über den gesamten Datenaustausch hinweg stabil an, das heißt vom Setzen des Request-Signals bis zur Rücknahme des Acknowledge-Signals (vergleiche Bild 4.17). Verwendet man Verzögerungsglieder, um das Fertig-Signal zu erzeugen, so bietet sich hier die Möglichkeit die Verzögerungskette zu halbieren. Das Setzen des Request-Signals wird zur Initialisierung des Protokolls verwendet. Das Acknowledge des Requests erfolgt frühestens nach der Hälfte der maximalen Durchlaufzeit durch die Logikwolke („Logikverzögerung“). Anschließend wird das Request-Signal zurückgenommen. Wiederum wird dieses Signal um die halbe „Logikzeit“ verzögert, so dass sich eine Gesamtverzögerung größer gleich der maximalen Logikverzögerung ergibt.

Allerdings müssen die Protokolle angepasst werden. Das Datum liegt erst nach der Rücknahme der Request-Signals stabil an den Eingängen der Speicher an. Somit darf das Signal zum Speichern des Datums auch erst nach der Rücknahme gesetzt werden.

Ein weiterer Vorteil ist, dass solche Schaltungen - mit dann wiederum angepasster Verzögerung - auch als Konverter eines „Spät“-Protokolls in ein „Breit“-Protokoll verwendet werden können. Durch die Halbierung der Anfrageverzögerung liegt das Datum am Eingang nur von der Rücknahme des Requests bis zur Rücknahme des Acknowledge stabil an. Dies entspricht der Vereinbarung eines „Spät“-Protokolls. Am Ausgang liegen die Daten aber weiterhin entsprechend dem „Breit“-Protokoll an.

4.2.5.1 RUBLO_HALF

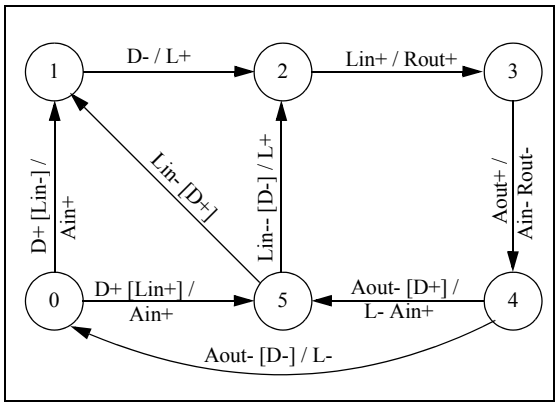


Bild 4.69: Zustandsgraph für die Kontrollstruktur RUBLO_HALF

In Bild 4.69 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Vergleicht man diesen mit dem ursprünglichen Automaten für diese Schaltung (Bild 4.44) so erkennt man, dass hier nun das rückgekoppelte Latch-Signal Lin verwendet wird, um Rout+ zu setzen. Das heißt, erst wenn eingangsseitig das Zurücknehmen des Requests im verzögerten Signal D detektiert wird, wird das Latch geschlossen, um das aktuelle Datum zu halten. Rout darf folgerichtig erst nach dem Schließen der Latches gesetzt werden.

Dieses explizite Berücksichtigen des Latch-Signals führt wieder zu einem, wenn auch gering, größeren Automaten. Inwieweit ein Gewinn in der Performanz zu erreichen ist, muss überprüft werden.

4.2.5.2 RSBLO_HALF

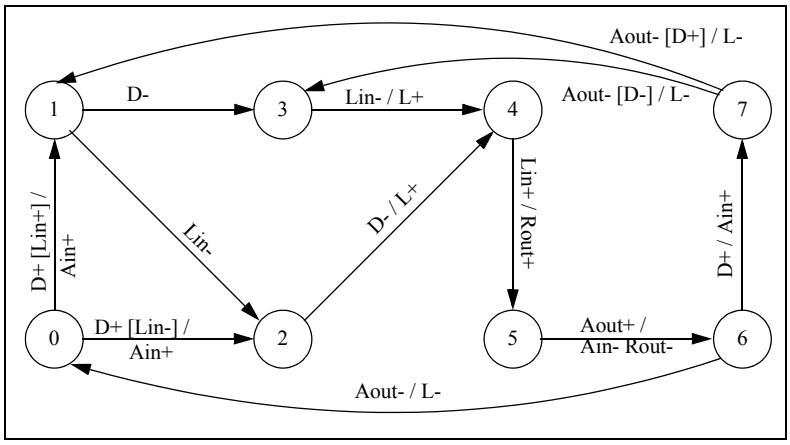


Bild 4.70: Zustandsgraph für die Kontrollstruktur RSBLO_HALF

In Bild 4.70 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Auch hier wird das Latch-Signal explizit detektiert, um abhängig davon das Request-

signal für die nachfolgende Stufe zu erzeugen. Hierdurch ergeben sich zusätzliche Freiheitsgrade, die in einen größeren Automaten resultieren. Somit ist wieder zu überprüfen, ob man einen Gewinn in der Performanz erreicht.

4.2.5.3 RDBLO_HALF

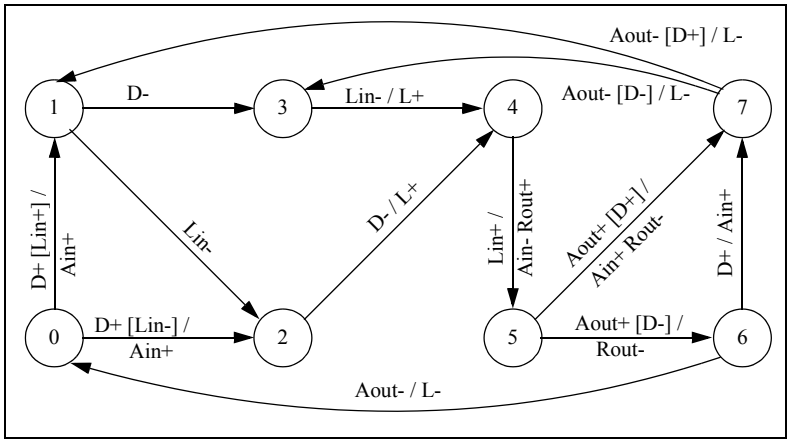


Bild 4.71: Zustandsgraph für die Kontrollstruktur RDBLO_HALF

In Bild 4.71 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geöffnete Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Wiederum wird das Latch-Signal explizit detektiert, um abhängig davon das Request-Signal für die nachfolgende Stufe zu erzeugen. Hierdurch ergeben sich zusätzliche Freiheitsgrade, die in einen größeren Automaten resultieren. Allerdings ist der Zuwachs gegenüber der halbentkoppelten Schaltung minimal. Ob sich tatsächlich ein Gewinn in der Performanz erreichen lässt, muss überprüft werden.

4.2.5.4 RUBLC_HALF

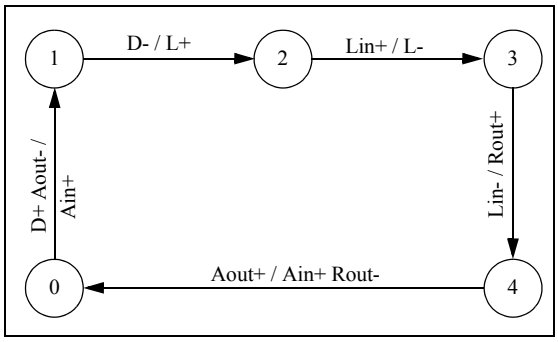


Bild 4.72: Zustandsgraph für die Kontrollstruktur RUBLC_HALF

In Bild 4.72 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. In dem Vergleich zur ursprünglichen Realisierung (Bild 4.48) kommt lediglich ein

Zustand hinzu. Der ergibt sich durch das Erzeugen des Signals Rout aus dem rückgekoppelten Latchsignal (Lin).

4.2.5.5 RSBLC_HALF

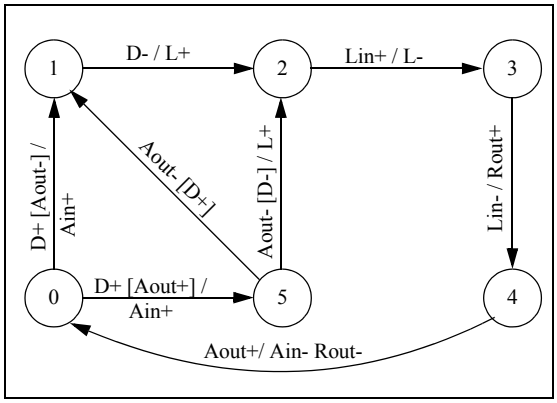


Bild 4.73: Zustandsgraph für die Kontrollstruktur RSBLC_HALF

In Bild 4.73 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. In diesem Fall vereinfacht sich der Automat gegenüber der ursprünglichen Realisierung (Bild 4.50). Dies liegt daran, dass das Latch erst bei Auftreten von D- geschlossen wird. Somit wird das Protokoll gegenüber der ursprünglichen Spezifikation eingeschränkt.

4.2.5.6 RDBLC_HALF

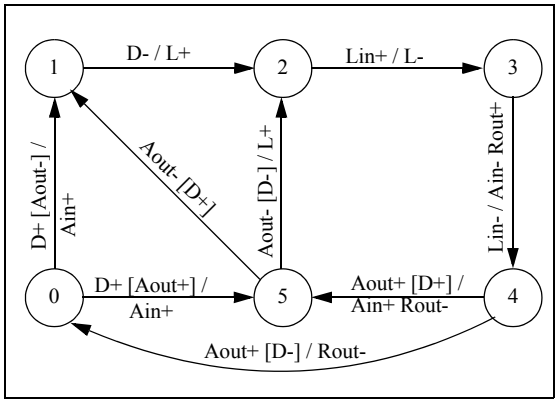


Bild 4.74: Zustandsgraph für die Kontrollstruktur RDBLC_HALF

In Bild 4.74 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für standardmäßig geschlossene Latches im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Auch in diesem Fall vereinfacht sich der Automat gegenüber der ursprünglichen Realisierung (Bild 4.51), weil das Protokoll gegenüber der ursprünglichen Spezifikation eingeschränkter ist.

4.2.5.7 RUBFF_HALF

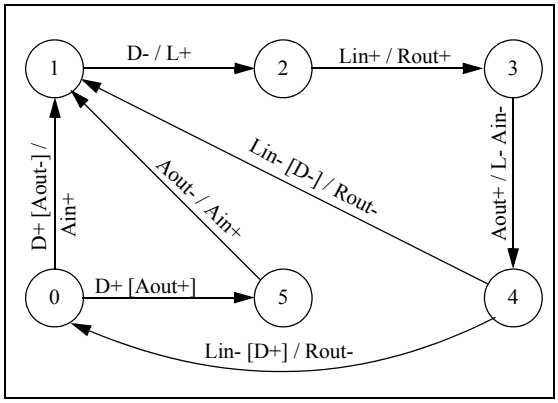


Bild 4.75: Zustandsgraph für die Kontrollstruktur RUBFF_HALF

In Bild 4.75 ist der Automat für eine request-activated, gekoppelte Kontrollschaltung für Flipflops im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Dieser Automat ist mehr als doppelt so groß, als der ursprüngliche Automat nach Bild 4.35. Es ist damit zu rechnen, dass hier kaum eine Performanzsteigerung erzielt werden kann.

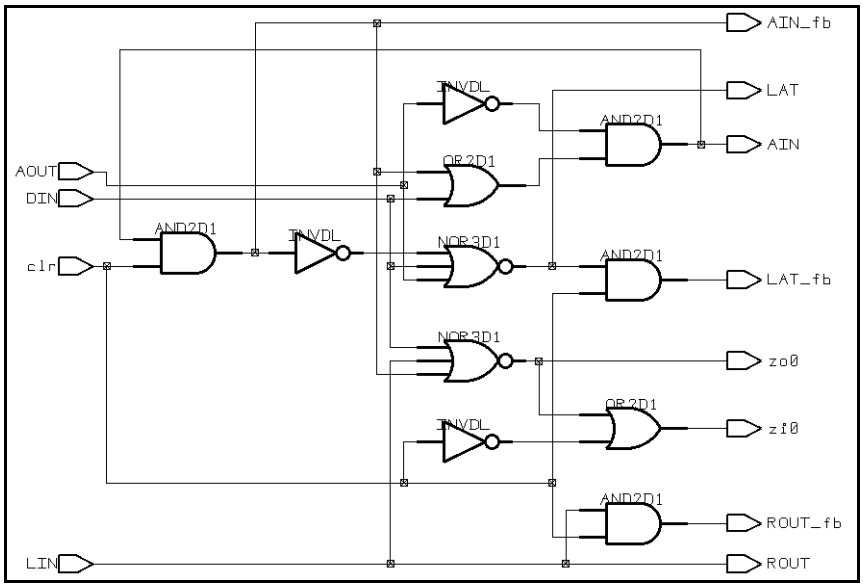


Bild 4.76: Implementierung der Schaltungen RUBFF_HALF

Die Implementierung des Automaten nach Bild 4.75 ist in Bild 4.76 gezeigt. Man sieht, dass diese Implementierung deutlich größer ist, als die des ursprünglichen Automaten (siehe Bild 4.36). Der Hauptgrund liegt darin, dass sich hier die Ausgangssignale nicht zu einem Signal zusammenfassen lassen. Somit sind mehr Zustände in dem resultierenden Automaten zu berücksichtigen.

4.2.5.8 RSBFF_HALF

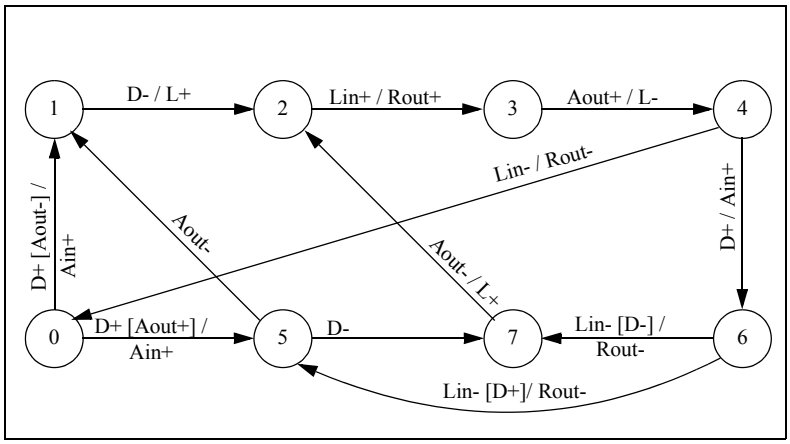


Bild 4.77: Zustandsgraph für die Kontrollstruktur RSBFF_HALF

In Bild 4.77 ist der Automat für eine request-activated, halbentkoppelte Kontrollschaltung für Flipflops im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Auch hier vergrößert sich der Automat deutlich gegenüber der ursprünglichen Beschreibung (Bild 4.52).

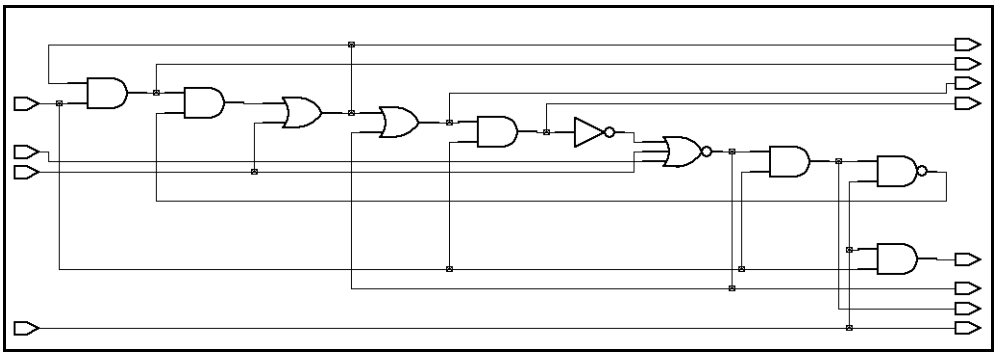


Bild 4.78: Implementierung der Schaltungen RSBFF_HALF

Die Implementierung in Bild 4.78 zeigt aber, dass das Programm DGC hier eine kompaktere Lösung ermittelt, als für die ursprüngliche RSBFF Schaltung. Deren Implementierung (Bild 4.53) ist fast doppelt so groß, wie die hier betrachtete Implementierung. Somit kann hier eventuell ein Performanzgewinn erzielt werden.

4.2.5.9 RDBFF_HALF

In Bild 4.79 ist der Automat für eine request-activated, entkoppelte Kontrollschaltung für Flipflops im „Breit"-Protokoll mit halber „Logikverzögerung" gezeigt. Wie in manchen Fällen vorher auch, vergrößert sich hier der Automat deutlich gegenüber dem ursprünglichen Automaten nach Bild 4.54.

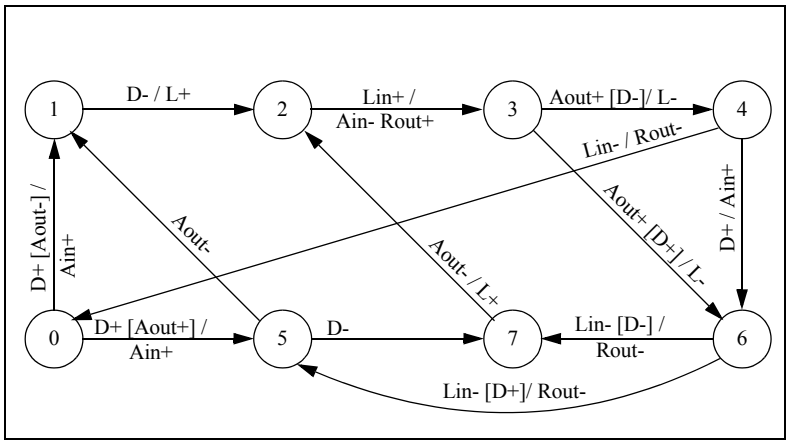


Bild 4.79: Zustandsgraph für die Kontrollstruktur RDBFF_HALF

4.2.6 Optimierung der Rücksetzphase

Bei der Abarbeitung eines Vier-Phasen-Protokolls (Kapitel 4.2.1.2) werden eine Setz- und eine Rücksetzphase abgearbeitet. In der Setzphase wird unter anderem darauf gewartet, dass neue Daten verarbeitet werden. Ist dies erfolgt, werden die neu errechneten Daten in lokale Speicher übernommen. Anschließend erfolgt die Rücksetzphase, in der das Protokoll vervollständigt wird. In dieser Phase wird letztendlich nur gewartet.

Um diese Wartezeit zu verkürzen, kann man versuchen, direkt das Request-Signal (Rin) auszuwerten. Somit muss nicht mehr auf ein Zurücksetzen des Fertig-Signals (D) gewartet werden. Speziell wenn man mit Verzögerungsgliedern zur Erzeugung des Fertig-Signals arbeitet, kann dieses Vorgehen Vorteile beinhalten.

Um ein korrektes Funktionieren zu gewährleisten, muss die Verzögerung im Allgemeinen der maximalen Laufzeit durch die Logik entsprechen. Somit ergibt sich eine Gesamtzeit zur Abarbeitung des Protokolls von circa zweimal der maximalen Logiklaufzeit. Ist die Laufzeit durch die Logik und damit durch die Verzögerungsglieder groß, verlangsamt dies unnötig die Pipeline. Benutzt man nun das unverzögerte Request-Signal Rin statt das verzögerte Signal D, so kann sich diese Abarbeitungszeit signifikant verringern.

Allerdings wird sich dies nur bei großen Schaltungen ergeben. Der Automat wird mit der Berücksichtigung eines zusätzlichen Eingangssignals komplexer und benötigit somit selbst mehr Zeit zur Abarbeitung. Im Extremfall eines Schieberegisters (keine Logik und damit keine Laufzeit durch diese) wird sich sogar eine Verschlechterung bemerkbar machen.

In Bild 4.80(a) ist als Beispiel der Automat für die Kontrollstruktur RUWLO mit einer verkürzten Rücksetzphase gezeigt. Dies ist an den Transitionen zu erkennen, die den Zustand 5 verlassen. Entscheidend ist hier neben dem Setzen von Aout (ergibt sich aus der Kopplung, siehe Kapitel 4.2.3) das Zurücksetzen von Rin und nicht mehr von D, wie im ursprünglichen Auto-

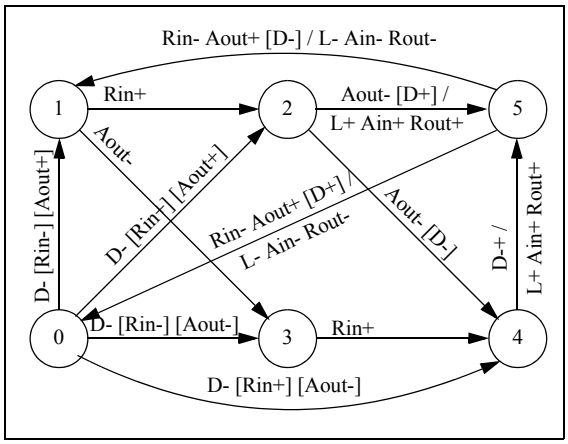


Bild 4.80: Zustandsgraph für die Kontrollstruktur RUWLO
mit verkürzter Rücksetzphase

maten (Bild 4.35). Somit kann ausgangsseitig das Request-Signal Rout zurückgesetzt werden, ohne dass D zurückgesetzt worden ist.

Andererseits erkennt man, dass der Automat komplexer geworden ist. Er besitzt dreimal so viel Zustände wie der ursprüngliche Automat. Allerdings lässt sich der Automat verkleinern, in dem Zustände zusammengefasst werden. Dies wird von DGC erkannt und auch erledigt.

Tabelle 6: Vergleich der Leistungswerte zweier RUWLO Implementierungen

		Schieben			Addieren/Subtrahieren		
		Fast	100mhz	Third	Fast	100mhz	Third
Zeit (ns)	RUWLO	26.446	79.629	59.761	61.686	88.782	71.282
	RUWLO_S	27.417	79.683	60.083	48.581	87.148	69.648
	Gewinn	**-3,7%**	**-0,1%**	**-0.5%**	**21,2%**	**1,8%**	**2,3%**
Power (µW)	RUWLO	1.0587	1.0681	1.0681	1.6828	1.7008	1.6946
	RUWLO_S	1.1746	1.1710	1.1710	1.8208	1.8258	1.8201
	Gewinn	**-10,9%**	**-9,6%**	**-9,6%**	**-8,2%**	**-7,3%**	**-7,4%**
Performanz	RUWLO	0.279	0.850	0.638	1.038	1.510	1.207
	RUWLO_S	0.322	0.933	0.703	0.884	1.591	1.267
	Gewinn	**-15,4%**	**-9,8%**	**-10,2%**	**14,8%**	**-5,4%**	**-5,0%**

Die Gegenüberstellung der Leistungswerte der beiden Realisierungen der Kontrollschaltung RUWLO ist in Tabelle 6 dargestellt. Wie zu erwarten, sind die Leistungswerte bei dem Beispiel mit Schieberegistern durchwegs schlechter. Dies liegt an der Vergrößerung der Kontrollschaltung, die direkt für eine Verlangsamung, Vergrößerung und höheren Verbrauch der Gesamtschaltung sorgt.

Bei dem Beispiel mit Addierern und Subtrahierern ist durchwegs eine Verbesserung der Laufzeit zu beobachten. Genau dieses sollte durch die Verkürzung der Rücksetzphase erreicht werden. Die vergrößerte Kontrollschaltung sorgt allerdings für einen größeren Energiebedarf. Der Gewinn bei der Laufzeit muss also gegen die Zunahme des Energieverbrauchs abgewägt werden. Im Fall des schnellstmöglichen Betriebs der Pipeline ist der Geschwindigkeitsgewinn aber so deutlich (21,2%), dass der Zuwachs des Energiebedarfs (8,2%) in Kauf genommen werden kann.

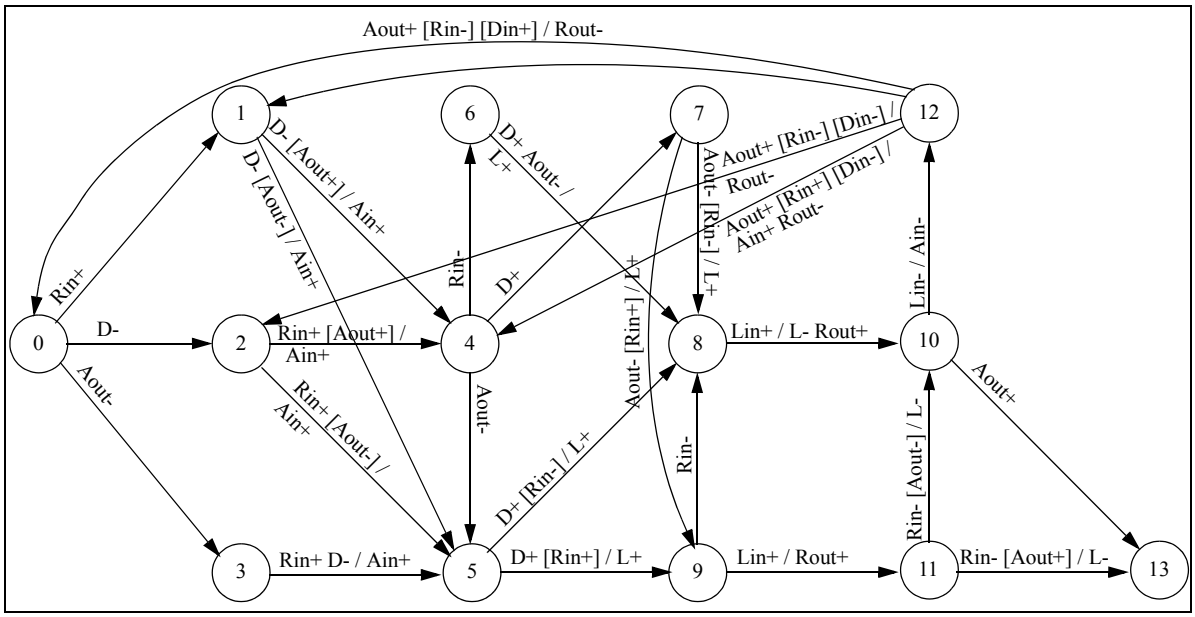


Bild 4.81: Zustandsgraph für die Kontrollstruktur ADBLC mit verkürzter Rücksetzphase

Anders als bei den request-activated Schaltungen wird bei den acknowledge-activated Kontrollschaltungen schon das Request-Signal im Zustandsgraph ausgewertet. Somit ist bei dieser Gruppe von Kontrollschaltungen generell ein höheres Potential für Verbesserungen der Laufzeiten zu erwarten.

In Bild 4.81 ist der Automat für die Kontrollstruktur ADBLC mit verkürzter Rücksetzphase gezeigt. Im Vergleich zum ursprünglichen Automaten (Bild 4.62) hat sich die Anzahl der Zustände um 3 vermehrt. DGC kann den Automaten aber ähnlich wie den ursprünglichen zusammenfassen, so dass eine entsprechende Verbesserung zu erwarten ist.

Tabelle 7: Vergleich der Leistungswerte zweier ADBLC Implementierungen

		Schieben			Addieren/Subtrahieren		
		Fast	100mhz	Third	Fast	100mhz	Third
Zeit (ns)	ADBLC	32.670	86.642	67.042	60.268	94.832	78.122
	ADBLC_S	28.729	82.423	62.823	56.391	88.742	71.242
	Gewinn	**12,1%**	**4,9%**	**6,3%**	**6,4%**	**6,4%**	**8,8%**
Power (µW)	ADBLC	1.5920	1.6023	1.6023	2.3834	2.3592	2.3809

Tabelle 7: Vergleich der Leistungswerte zweier ADBLC Implementierungen

		Schieben			Addieren/Subtrahieren		
		Fast	100mhz	Third	Fast	100mhz	Third
	ADBLC_S	1.4906	1.4996	1.4996	2.1755	2.1728	2.1681
	Gewinn	**6,4%**	**6,4%**	**6,4%**	**8,7%**	**7,9%**	**8,9%**
Performanz	ADBLC	0.520	1.388	1.074	1.436	2.237	1.860
	ADBLC_S	0.428	1.236	0.942	1.226	1.928	1.544
	Gewinn	**17,7%**	**11,0%**	**12,3%**	**14,6%**	**13,8%**	**17,0%**

Die Gegenüberstellung der Leistungswerte der beiden Realisierungen der Kontrollschaltung ADBLC ist in Tabelle 7 dargestellt. Wie prognostiziert erhält man deutliche Verbesserungen. Am deutlichsten ist der Gewinn, wenn mit schnellstmöglicher Datenrate gearbeitet wird (modus "Fast"). Dies ist klar, da bei entsprechenden Wartezeiten auf das nächste Datum der Vorteil einer verkürzten Rücksetzphase zunichte gemacht wird.

Auffällig ist allerdings, dass in allen Simulationen ebenfalls ein niedriger Energieverbrauch zu verzeichnen ist. Das kann sich nur aus einer kleineren Realisierung der Kontrollschaltung ergeben. Somit ergibt sich eine durchwegs bessere Realisierung. Der Grund hierfür ist in der Optimierung des Zustandsgraphen durch das Programm DGC zu suchen. Mit der hier gegebenen Spezifikation schafft DGC es offensichtlich eine kleinere Lösung zu finden.

Die Qualität des Ergebnisses hängt also nicht nur von der Wahl der theoretisch besten Klasse einer Kontrollschaltung, sonderen auch von der „geschickten" Spezifikation des Graphen selbst ab.

4.2.7 Spezifikationsoptimierung

Eine weitere Rolle bei der Entwicklung einer optimalen asynchronen Schaltung mit Hilfe von DGC spielt die Spezifikation der Kontrollautomaten. DGC versucht mittels einer Automatenminimisierung eine kleinstmögliche Lösung zu finden. Entsprechend kann eine bessere Ausgangsposition (Spezifikation des Automaten) zu einer signifikaten Verbesserung des Endergebnisses führen.

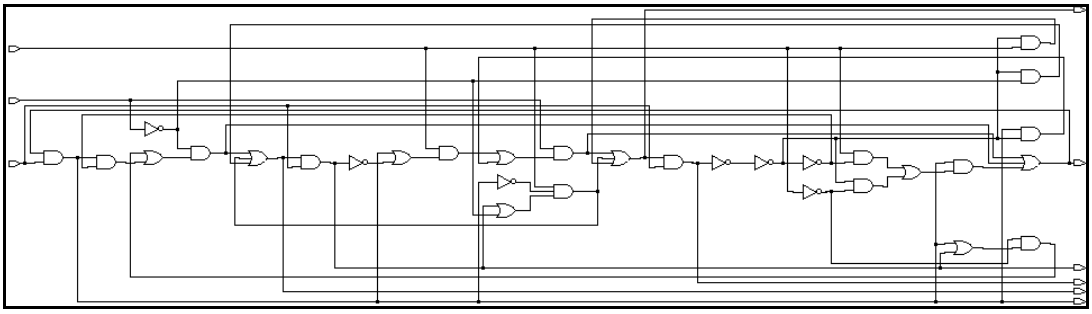


Bild 4.82: Implementierung der Schaltungen RDBFF

In Bild 4.54 ist zum Beispiel der Zustandsgraph für die Kontrollstruktur RDBFF gezeigt. Diese Spezifikation ergab sich intuitiv aus den Vorgaben für diese spezielle Kontrollstruktur. Sie hat aber zwei Nachteile. Zum Ersten muß eine Laufzeitüberprüfung für Ain- erfolgen. In einem Schieberegister kann das Zurücksetzen von Ain zu einer schnellen Änderung des Datums führen. Geschieht dies schneller als das Speichersignal L gesetzt wird, so wird die Setup-Zeit für den Speicher verletzt, beziehungsweise ein falsches Datum übernommen. Hier sind notfalls Verzögerungsglieder einzufügen.

Zum Zweiten kann theoretisch bei Schieberegistern die Reaktion der vor- und nachgeschalteten Stufen so schnell erfolgen, dass die Mindestbreite für eine Eins- oder Null-Phase des Speichersignals nicht erreicht wird. Auch dieses muss gegebenenfalls durch Verzögerungsglieder korrigiert werden.

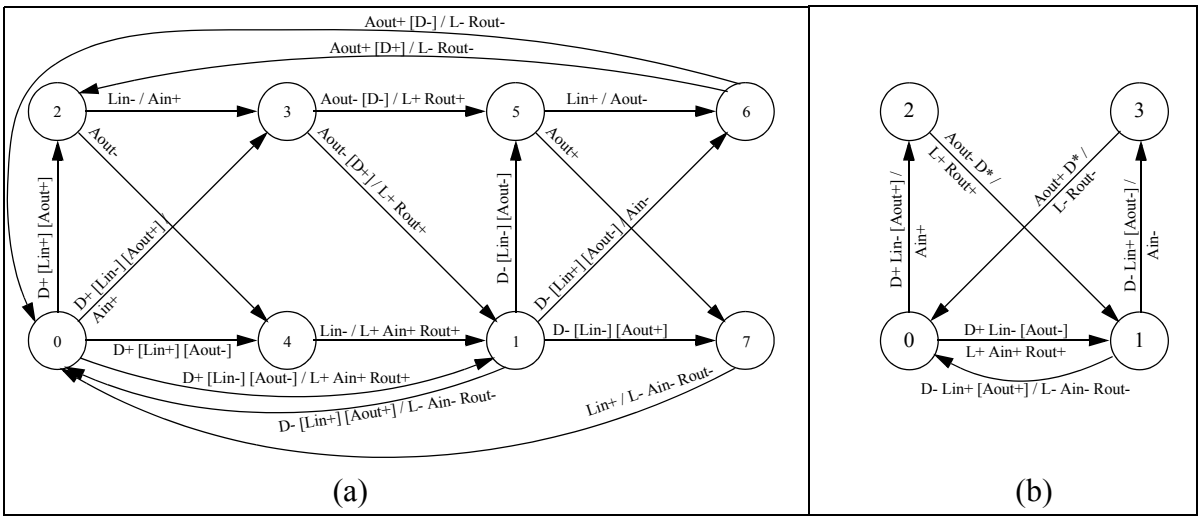


Bild 4.83: Verbesserter Zustandsgraph für die Kontrollstruktur RDBFF

Der Vorteil der gewählten Spezifikation scheint der an sich kleine Automat zu sein. Berücksichtigt man aber explizit den Zustand des Speichersignal L, dann vermeidet man die oben beschriebenen Probleme („zu schnelles Ain“), beziehungsweise kann diese leichter korrigieren (Eins- und Null- Phase des Speichersignals). Dadurch scheint der Automat auf die doppelte Größe zu wachsen, wie dies in Bild 4.83 (a) zu sehen ist. Allerdings lässt sich dieser Automat hinreichend minimieren. Das Ergebnis ist in Bild 4.83 (b) dargestellt. In dieser Darstellung wird die Eigenschaft ausgenützt, dass in bestimmten Zuständen und Übergängen der Pegel eines Signals keinen Einfluss hat. Dieses kann in einem „extended burst mode“ - Graphen mit einem “*“ gekennzeichnet werden. Auch DGC kann diese Spezifikation verarbeiten; somit kann der Graph Bild 4.83 (b) direkt als Eingabe für DGC dienen.

Der Graph nach Bild 4.83 (b) ist jetzt wieder gleich groß wie der der ursprünglichen Spezifiaktion. Allerdings sind hier jetzt ein paar Übergänge verboten (2 nach 3 und umgekehrt). Dies führt letztendlich dazu, dass die Ausgangssignale (Signal Ain und das kombinierte Signal

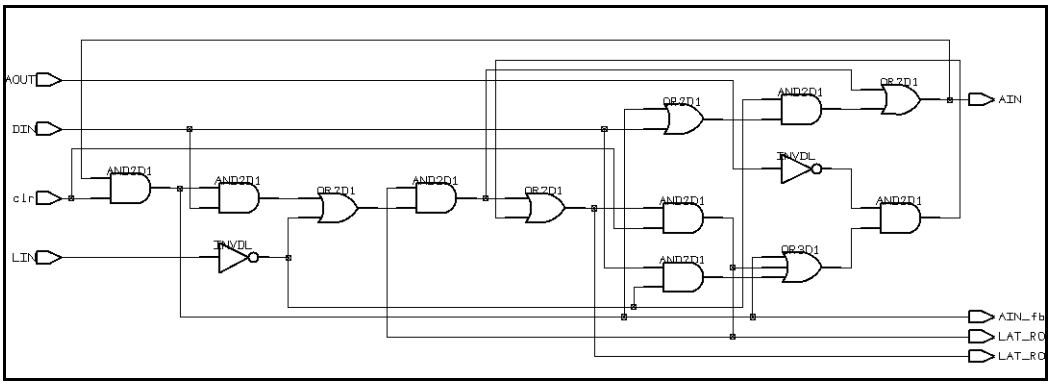


Bild 4.84: Verbesserte Implementierung der Schaltungen RDBFF

L_Rout) als Zustandssignale ausreichen und sich somit die Implementierung (Bild 4.84) gegenüber Bild 4.82 verkleinert. Dies hat signifikante Auswirkungen auf die Laufzeiten durch die Kontrollschaltung und damit auf die Performanz der Pipeline.

Tabelle 8: Vergleich der Leistungswerte zweier RDBFF Implementierungen

		Schieben			Addieren/Subtrahieren		
		Fast	100mhz	Third	Fast	100mhz	Third
Zeit (ns)	RDBFF1	36.457	80.966	61.091	70.594	89.965	74.440
	RDBFF2	21.593	79.152	59.252	57.362	88.334	70.834
	Gewinn	**40,8 %**	**2,2 %**	**3,0 %**	**18,7 %**	**1,8 %**	**4,8 %**
Power (µW)	RDBFF1	2.5144	2.4548	2.4553	3.3961	3.3345	3.3808
	RDBFF2	1.4012	1.3996	1.4000	2.1421	2.1399	2.1406
	Gewinn	**44,3 %**	**43,0 %**	**43,0 %**	**36,9 %**	**35,8 %**	**36,7 %**
Performanz	RDBFF1	0.916	1.987	1.499	2.397	2.999	2.516
	RDBFF2	0.302	1.107	0.829	1.228	1.890	1.516
	Gewinn	**67,0 %**	**44,3 %**	**44,7**	**48,8 %**	**37,0 %**	**39,7 %**

Die Gegenüberstellung der Leistungswerte der beiden Realisierungen der Kontrollschaltung ist in Tabelle 8 dargestellt. Auffallend ist die generelle Verbesserung bei der Verlustleistung. Diese ergibt sich vor allem durch die kleinere Kontrollschaltung, bei der weniger Gatter umgeschaltet werden. Die kleinere Kontrollschaltung besitzt auch eine kleinere Durchlaufzeit beziehungsweise Einschwingzeit (settle time für den Fundamentalmodus). Dies wird bei der „Fast“ Simulation mit größtmöglicher Durchsatzrate deutlich. Bei den Schieberegistern wird die Gesamtschaltung zum Beispiel um gute 40% schneller. Der für eine Auswahl entscheidende Performanzfaktor ist ebenfalls bei allen Simulationen besser.

An dem gezeigten Beispiel der Kontrollschaltung RDBFF wird klar, dass die Aufstellung des asynchronen Kontrollautomaten eine wesentliche Auswirkung auf die spätere Schaltung haben kann. Die resultierende Implementierung hängt stark von dem vorgegebenen Graphen ab.

Die in ASMOGEN zur Verfügung gestellten Kontrollschaltungen sind empfohlene Schaltungen. Bei einer entsprechenden Weiterentwicklung von DGC oder einem Auffinden eines besseren Graphen für die entsprechende Kontrollschaltung, sollten diese passend ersetzt werden.

4.2.8 Protokollkonverter

Um eine breite Anwendbarkeit sicher zu stellen, braucht man noch zusätzlich die Bindeglieder zwischen den unterschiedlichen Protokollen. Schaut man sich die Zeitbedingungen für das „Breit“-Protokoll an, so ist zu sehen, dass nach diesem sowohl „Weit“- als auch „Früh“-Protokolle angeschlossen werden können. Das Gleiche gilt für den Anschluss eines „Früh“-Protokolls in Folge eines „Weit“-Protokolls. Somit müssen die Bindeglieder für die Konvertierung „Weit“ -> „Breit“ sowie „Früh“ -> „Weit“ und „Früh“ -> „Breit“ entworfen werden.

4.2.8.1 „Früh“ zu „Weit“ Konvertierung

Hier kommt es darauf an, die Daten länger zu halten. Die eingangsseitige Übernahme ist unkritisch, da beim „Weit“-Protokoll genauso wie beim „Früh“-Protokoll das Datum vor dem Acknowledge übernommen worden sein muss. Ausgangsseitig muss das Datum länger stabil anliegen, bis die Anfragebestätigung erfolgt ist.

Somit kommen für die Konvertierung die jeweils für das „Weit“-Protokoll ausgelegten Kontrollstrukturen in Frage. Das heißt alle Kontrollstrukturen des „Weit“-Protokolls sind als „Früh“ - „Weit“ Konverter einsetzbar.

4.2.8.2 „Früh“ zu „Breit“ Konvertierung und „Weit“ zu „Breit“ Konvertierung

Hier muss das Datum am Ausgang deutlich länger stabil anliegen, als dies eingangsseitig der Fall ist. Somit kann man nicht alle „Breit“-Protokolle der verschiedenen Speichertypen auch für die Konvertierung benutzen.

Im Speziellen kann man hier keine vollständige oder halbe Entkopplung der einzelnen Stufen erreichen. Eingangsseitig darf eine Anfragebestätigung immer erst dann erfolgen, wenn das Datum erfolgreich übernommen worden ist. Ausgangsseitig muss das Datum allerdings bis zum Ende des Datenaustauschs stabil anliegen („Breit“-Protokoll). Somit wird eingangsseitig erst ein Request bestätigt, wenn der vergangene Datenaustausch am Ausgang beendet ist. Dies entspricht einer Kopplung zwischen den Stufen einer synchronen Pipeline.

Es verbleiben also noch die gekoppelten Protokollvarianten. Im Einzelnen sieht das folgendermaßen für die verschiedenen Speichertypen aus:

- Normal geschlossene Latches:
  Da hier bei einem eingangsseitigen Request erst das Latch geöffnet und anschließend

gleich wieder geschlossen werden muss, bleibt das Datum ausgangsseitig stabil bis zum Ende des Austausches. Wegen dieser Gegebenheit ergibt sich die Möglichkeit, die gekoppelte Schaltung für das „Breit“-Protokoll von normal geschlossenen Latches zu verwenden, wie es unter “RUBLC” auf Seite 115 beschrieben ist.

- Flipflops:
  Auch hier wird durch einen eingangsseitigen Request ein neues Datum übernommen und wegen der Kopplung und dem Verhalten eines Flipflops bis zum Ende des ausgangsseitigen Datenaustauschs stabil gehalten. Allerdings kann man die unter “RUBFF” auf Seite 117 beschriebene Schaltung nur bei „langsamen“ Schaltungen auf der Eingangsseite verwenden.

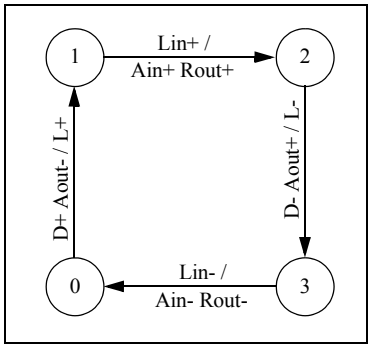


Bild 4.85: Zustandsgraph für die Kontrollstruktur RUE2BFF

Der Grund: Das Speichersignal LAT wird zeitgleich mit AIN gesetzt. Das heißt, wenn die Eingangsseite ein „Früh“-Protokoll (statt eines „Breit“-Protokolls, wie es in “RUBFF” auf Seite 117 vorausgesetzt ist) abarbeitet und die aktuelle Stufe keine Logik enthält, kann es unter Umständen passieren, dass neue Daten bis zu den Eingängen der Flipflops durchkommen, bevor die Übernahme der aktuellen Daten sicher erfolgt ist.
Eine sichere Variante ist in Bild 4.85 dargestellt. Hier wird das Latch-Signal explizit überprüft. Erst wenn durch die Rückkopplung ein Lin+ detektiert wird, wird ein Acknowledge auf der Eingangsseite gesetzt (Ain+). Somit ist dieser Automat auch für eingangsseitig schnellere Schaltungen nutzbar.

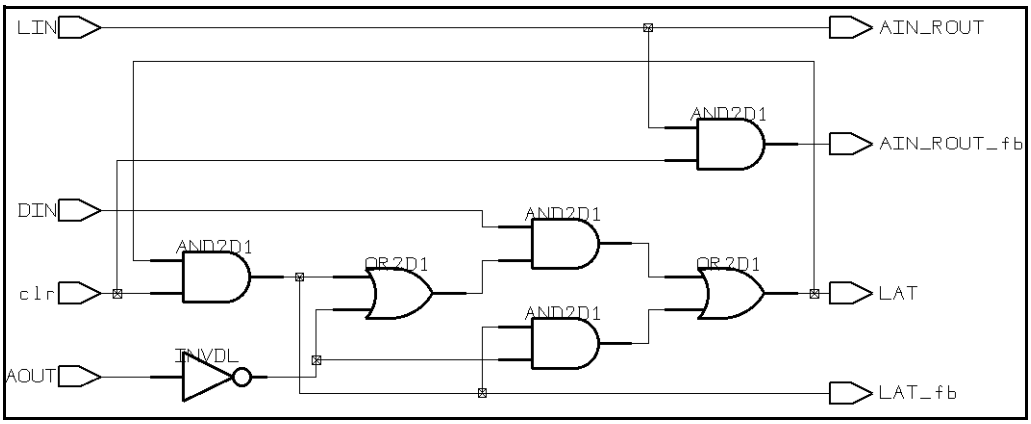


Bild 4.86: Implementierung der Schaltungen RUE2BFF

Die Implementierung in Bild 4.86 zeigt, dass die resultierende Schaltung nicht viel größer ist, als die der ursprünglichen RUBFF Schaltung (Bild 4.36). Somit stellt die hier gezeigte Variante eine gute Alternative zur angestrebten Kopplung dar.

- Normal geöffnete Latches:
  Hier muss eine eigene Schaltung implementiert werden. Die Daten liegen am Eingang zu kurz an, um eine der anderen Schaltungen für geöffnete Latches zu benutzen. Speziell muss das Speichersignal während des gesamten Datenaustauschs am Ausgang gesetzt bleiben.

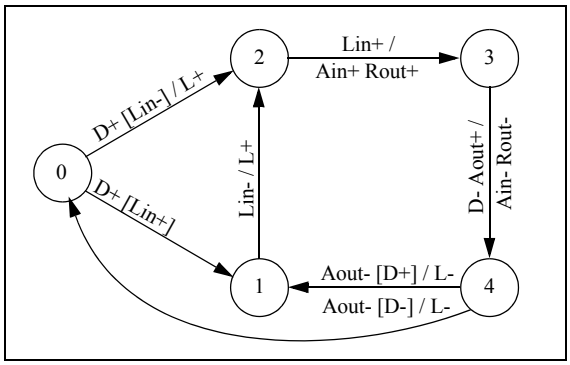


Bild 4.87: Zustandsgraph für die Kontrollstruktur RUE2BLO

In Bild 4.87 ist der Automat für die Konvertierung gezeigt. Er ist nicht größer als der Automat der RUBLO Schaltung. Somit dürfte diese Kopplungsschaltung keinen negativen Einfluss auf die Performanz der späteren Gesamtschaltung haben.

4.2.8.3 Asynchron zu Synchron Konvertierung

Der Übergang von einer asynchronen Pipeline zu einer synchronen Schaltung kann auf unterschiedliche Arten erfolgen. Eine einfache Methode ist in Bild 4.88 (a) dargestellt. Diese Methode synchronisiert auf einfachste Weise sowohl die ankommenden Daten als auch das Request-Signale ein. Das synchronisierte Request-Signal dient hierbei sowohl als Acknowledge-Signal für die asynchrone Seite als auch als Enable-Signal (Zeichen dafür, dass ein neues Datum anliegt) für die nachfolgende synchrone Seite. Diese Variante (a) hat aber mehrere Nachteile:

- Die Daten und auch das Request-Signal können sich in der Setup-Zeit des Taktsignals ändern und somit zu unvorhersehbaren Ergebnissen führen.

- Um das Risiko der Verletzung der Setup-Zeit zu verringern, schaltet man oftmals zwei Einsynchronisierungsstufen hintereinander. Das kann aber unter Umständen einen erheblichen Flächenbedarf und Energieverbrauch bedeuten. Zudem kann man dann auch keine

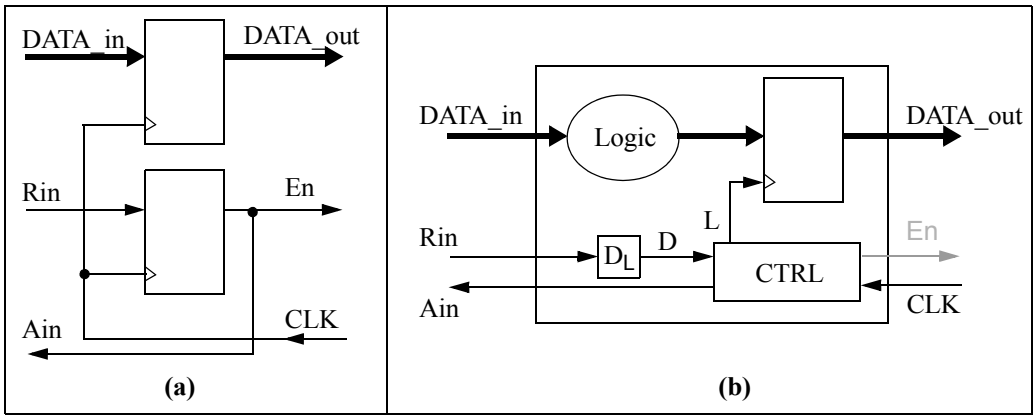


Bild 4.88: Schnittstellen von asynchron nach synchron

Datenverarbeitung zwischen den Einsynchronisierungsstufen durchführen und muss somit deutliche Performanzverluste hinnehmen.

- Es werden immer zwei Taktzyklen für den Austausch eines Datums benötigt. Das kann bei einer schnellen asynchronen Pipeline zum einen bedeuten, dass die synchrone Seite eine sehr hohe Taktrate benötigt. Im Extremfall müsste die Frequenz doppelt so hoch sein wie der Datendurchsatz auf der asynchronen Seiten. Zum anderen wird die asynchrone Pipeline unnötig ausgebremst, wenn nämlich die synchrone Seite eine relativ niedrige Taktfrequenz besitzt.

In Bild 4.88 (b) ist eine bessere Alternative gezeigt. Hier wird die Schnittstelle zur synchronen Seite wie eine normale asynchrone Pipelinestufe behandelt. Das heißt das Taktsignal wird als einfaches Steuersignal in der Kontrollschaltung ausgewertet. Das Fertig-Signal (D) wird zusätzlich als Enable-Signal für die synchrone Seite abgetaktet.

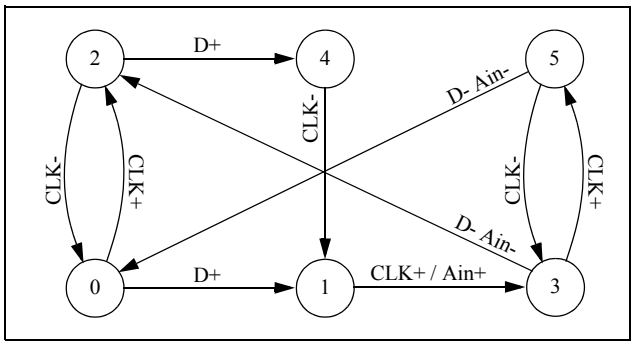


Bild 4.89: Zustandsgraph für die Kontrollstruktur AS2SY

Die Kontrollschaltung für die Schaltung in Bild 4.88 (b) ist in Bild 4.89 dargestellt. Man sieht, dass die Daten nur dann eingetaktet werden, wenn nach einem erfolgten Request eine aktive Taktflanke (CLK+, Übergang 1 auf 3) kommt. Liegt das Taktsignal schon (länger) auf ’1’ in dem Moment, in dem D auf ’1’ geht, so wird das Datum erst mit der nächsten Taktflanke übernommen.

Diese Variante ist sicherer als die erste, da das gesamte Zeitverhalten von DGC und ASMOGEN angepasst wird. Ein neues Datum liegt, ähnlich wie auf der synchronen Seite, nach einer aktiven Taktflanke an. Hierbei ist nur zu beachten, dass etwas weniger Zeit für die nachfolgende synchrone Stufe zur Verfügung steht, als für die restlichen synchronen Stufen, da das Latch-Signal erst eine kurze Zeit nach der aktiven Taktflanke generiert wird.

Ist die asynchrone Pipeline schnell genug, so wird in jedem Taktzyklus ein neues Datum übernommen. Somit führt diese Variante zu keinem unnötigen Performanzverlust. Ist sie „langsamer“ als die synchrone Seite, passt sich die Schnittstelle automatisch an. In diesem Fall müsste der Kontrollautomat noch ein Enable-Signal erzeugen, um der synchronen Seite jeweils ein neues Datum bekannt zu geben. Nicht zuletzt kann in dieser zweiten Variante immer ein Logikblock enthalten sein, somit erhält man keine unnötigen Zwischenstufen, wie bei der Serienschaltung mehrerer Einsynchronisierungsstufen der ersten Variante.

4.2.8.4 Synchron zu Asynchron Konvertierung

Der Übergang von einer synchronen Schaltung auf eine asynchrone Pipeline kann ebenfalls auf unterschiedliche Arten erfolgen. Wiederum kann man versuchen, den Übergang komplett in der synchronen Seite zu bewerkstelligen (vergleiche auch Bild 4.88 (a)). Allerdings muss man hierfür die asynchronen Handshake Signale eintakten (Gefahr der Verletzung der Setup Zeit des Taktes) und einen Automaten entwerfen, der den Datenaustausch abarbeitet. Dieses wiederum bedeutet aber, dass man mehrere Takte (mindestens 4 Takte) für den Austausch eines einzigen Datums benötigt. Das heißt, das bei dieser Variante eine Unsicherheit der Funktion (Verletzung der Setup Zeit) vorliegt und die Taktfrequenz mindestens viermal so hoch wie die mögliche Datenrate der asynchronen Pipelinestufe sein muss. Somit ist diese Möglichkeit der Anbindung nicht optimal.

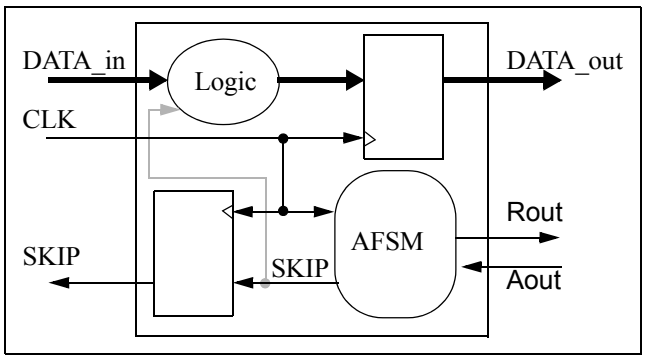


Bild 4.90: Schnittstellen von synchron nach asynchron

Ein weitere Möglichkeit liegt wieder in einer Lösung mit Hilfe einer asynchronen Schaltung. Bild 4.90 zeigt eine mögliche Struktur für die Anbindung. Der Datenpfad ist synchron aufgebaut, wobei die Logik Steuersignale aus dem asynchronen Kontrollautomaten berücksichtigen kann. Der asynchrone Kontrollautomat verarbeitet den Takt als ein Steuersig-

nal, wie auch die asynchronen Handshake-Signale. Er erzeugt ein Signal (SKIP), das verwendet wird, um anderen synchronen Modulen mitzuteilen, wenn eine Pause (jeweils für die nächste Taktperiode gültig) im Datenaustausch stattfindet.

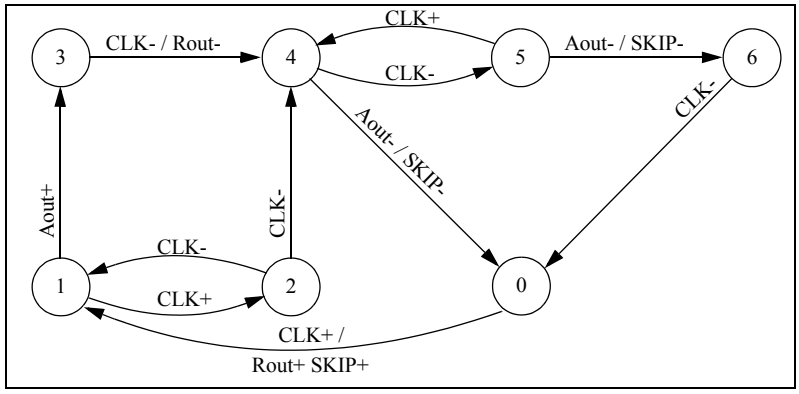


Bild 4.91: Zustandsgraph für eine einfache Kontrollstruktur SY2AS

Ein möglicher Zustandsgraph für eine Struktur nach Bild 4.90 ist in Bild 4.91 gezeigt. Mit dem nächsten gültigen Datum (CLK+) wird ein Request (Rout+) und das SKIP Signal gesetzt. Liegt dann das Acknowledge (Aout+) zusammen mit einer ‘0’-Phase (CLK-) des Taktsignals an, wird der Request zurückgenommen. Erfolgt die Rücknahme des Acknowledge noch vor der nächsten aktiven Taktflanke, kann das SKIP Signal zurückgenommen werden, ansonsten nicht. Hier liegt der Schwachpunkt dieser Implementierung. Eine Veränderung des SKIP Signals kann in der Setup Zeit des Taktsignals auftreten und somit zu Fehlfunktionen führen.

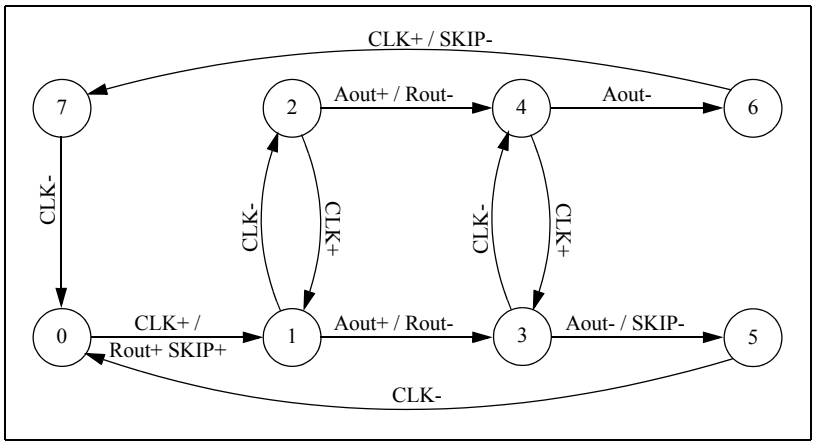


Bild 4.92: Zustandsgraph für eine sichere Kontrollstruktur SY2AS

Eine Methode, um die Anbindung nach Bild 4.90 sicherer durchzuführen, ist es, die Taktphasen mit einem Enable-Signal zu maskieren: Nur in der 1-Phase des Enable-Signals darf sich das SKIP Signal ändern. Bild 4.92 zeigt eine mögliche Variante. Hier dient das Taktsignal selbst als Enable-Signal. das SKIP Signal darf sich also nur in der 1-Phase des Taktsignals ändern. Somit bleibt der asynchronen Seite streng genommen nur die halbe Taktperiode für einen

schnellen Datenaustausch. Man kann diese Variante also in den Fällen einsetzen, in denen die asynchrone Seite schnell gegenüber der synchronen Seite arbeitet.

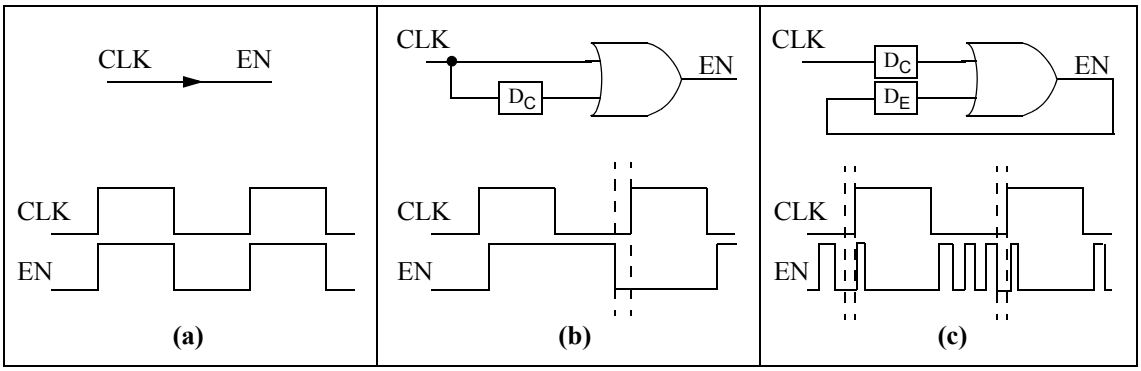


Bild 4.93: Maskieren des Taktsignals

Eine kleine Zusammenstellung der Möglichkeiten, ein Enable-Signal zur Maskierung des Taktsignals zu erzeugen, sind in Bild 4.93 dargestellt. Fall (a) ist der vorher beschriebene Fall, in dem das Taktsignal gleich dem Enable-Signal ist. Fall (b) ist für einen schnellen Systemtakt geeignet. Hier setzt das Taktsignal das Enable-Signal, welches durch das verzögerte Taktsignal gehalten wird. Das Zeitverhalten lässt sich relativ einfach einstellen, da nur ein Pfad Verzögerungsglieder enthält und einzig eine ‘0’-Phase des Enable-Signals während der Setup Zeit des Taktsignals gewährleistet werden muss. Für die Verzögerung ergibt sich folgende Bedingung:

$$D_C \leq \frac{T}{2} - t_{setup} - D_{OR} \qquad (4.12)$$

Variante (c) hingegen ist schwieriger zu handhaben. Es müssen zwei Pfade abgeglichen werden. Zu dem können Spikes auftreten, wenn am Beginn der 1-Phase des Taktsignals das Enable-Signal auf ‘1’ geht und durch das Taktsignal wieder auf ‘0’ gesetzt wird. Der Energiebedarf ist ebenfalls größer als bei den anderen beiden Varianten.

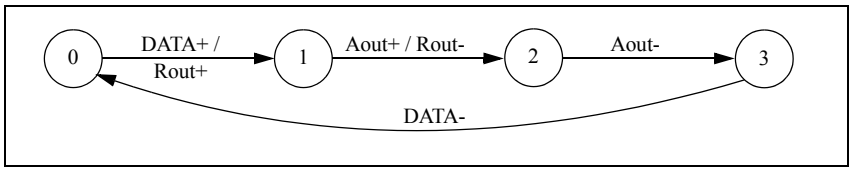


Bild 4.94: Zustandsgraph für eine Kontrollstruktur SY2AS,
bei sehr langsamen synchronen Takt

Bild 4.94 zeigt eine weitere Speziallösung für den Fall, dass der synchrone Systemtakt sehr langsam ist. Hier ist sichergestellt, dass die asynchrone Seite das Datum immer abnimmt, bevor ein neues Datum anliegen kann. Ist das Datum zum Beispiel ein skalares Signal, bei dem allein dessen ‘1’ Wert in der nachfolgenden Stufe ausgewertet wird, so vereinfacht sich der Zustandsgraph wie in Bild 4.94. Eine steigende Flanke des Datums wirft den Datenaustausch an. Nach

dessen Abschluss (Zustand 3) wird auf die Rücknahme des Datums gewartet, welches seinerseits zu keinem Datenaustausch führt.

4.2.9 Erweiterte Kontrollstrukturen

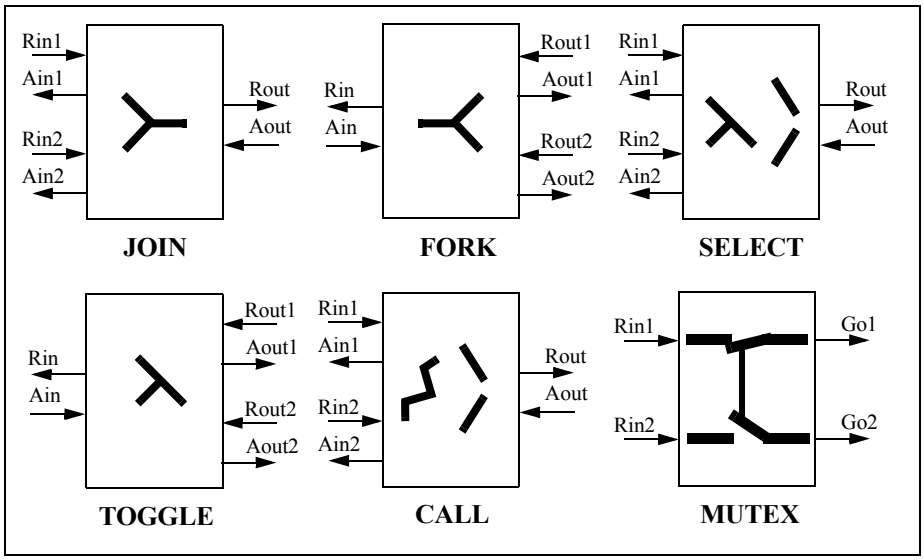


Bild 4.95: Erweiterte Kontrollstrukturen für asynchrone Pipelines

In diesem Kapitel werden zusätzliche Kontrollstrukturen vorgestellt, die notwendig sind, um größere und komplexere Schaltungen aufzubauen. Speziell die Verzweigung (FORK) und Zusammenführung (JOIN) mehrerer Pipelines ist eine wichtige Voraussetzung, um größere leistungsstarke asynchrone Systeme aufbauen zu können. Die wichtigsten Kontrollstrukturen sind in Bild 4.95 gezeigt. Sie werden nachfolgend näher erklärt. Da die Klasse der RUWLO Schaltungen am einfachsten zu beschreiben sind, werden die erweiterten Kontrollstrukturen für diese Klasse beschrieben.

JOIN_RUWLO

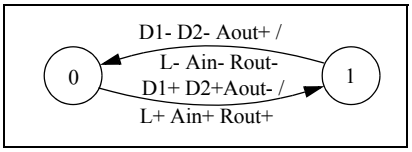


Bild 4.96: Zustandsgraph für die Kontrollstruktur JOIN_RUWLO

Bei einer Zusammenführung von Requests an eine Pipeline (Bild 4.96) müssen alle eingehenden Requests anliegen, damit in der aktuellen Stufe das Datum übernommen wird (L+). Dies wird dann der Vorgängerstufe bestätigt (Ain+) und der nachfolgenden Stufe eine neuer Request gesendet (Rout+). Wegen der Kopplung zwischen den Stufen muss der Ausgangsseitige Datenaustausch bereits beendet sein (Aout-). Der gleiche Vorgang wiederholt sich in der Rücksetzphase, nur mit umgekehrten Vorzeichen.

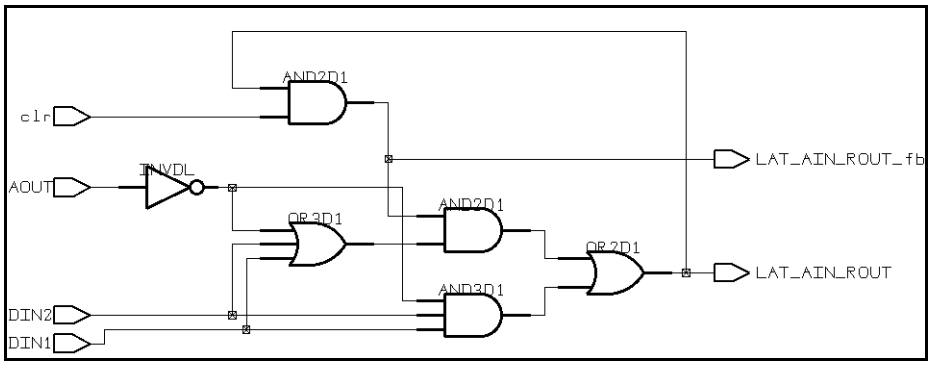


Bild 4.97: Implementierung der Schaltungen JOIN_RUWLO

Die resultierende Implementierung ist in Bild 4.97 gezeigt. Sie ist unwesentlich größer als die einfache RUWLO Schaltung (vergleiche Bild 4.36). Je mehr vorangehende Stufen zusammengefasst werden sollen, desto größer wird die Schaltung aber werden. Ab einer bestimmten Anzahl zusammen zu führender Stufen, empfiehlt es sich, mehrere JOIN_RUWLO Schaltungen zu schachteln, anstatt alle Stufen in einer einzigen Kontrollschaltung zusammenzuführen.

FORK_RUWLO

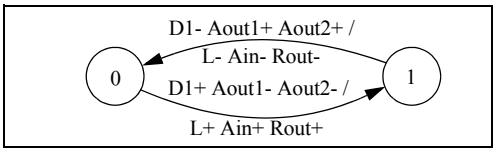


Bild 4.98: Zustandsgraph für die Kontrollstruktur FORK_RUWLO

Bei einer Verzweigung eines Requests an mehrere nachfolgende Stufen ergibt sich eine ähnliche Spezifikation, wie für die Zusammenführung. Nur muss hier auf mehrere nachfolgende Stufen gewartet werden. Die Spezifikation (Bild 4.98), als auch die Implementierung (Bild 4.99), sind wieder ähnlich einfach, wie die einfache RUWLO Schaltung. Auch hier empfiehlt es sich, bei vielen nachfolgenden Stufen eine Schachtelung von Verzweigungen zu benutzen.

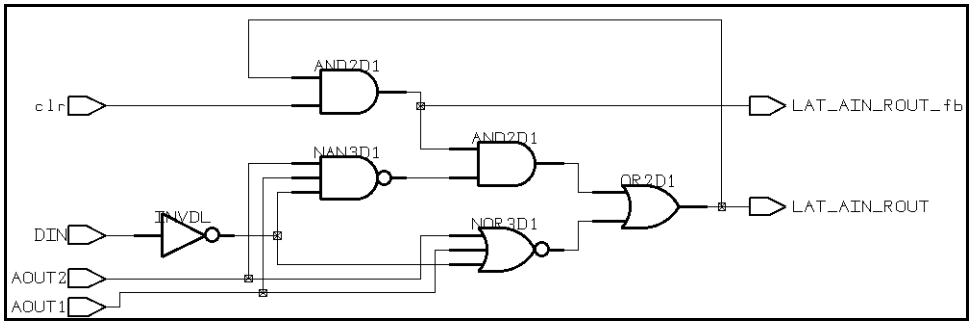


Bild 4.99: Implementierung der Schaltungen FORK_RUWLO

TOGGLE_RUWLO

Eine Toggle Schaltung ist eine vom Prinzip her einfache Schaltung. Die eingehenden Requests sollen abwechselnd an die nachgeschalteten Stufen weitergegeben werden. Bei zwei

nachfolgenden Stufen verdoppelt sich entsprechend die Anzahl der Zustände in der Automatenspezifikation (Bild 4.100). Über die Zustände 1 und 2 wird das Protokoll mit der ersten nachgeschalteten Stufe abgehandelt; mit den Zuständen 3 und 4 das Protokol mit der zweiten nachgeschalteten Stufe.

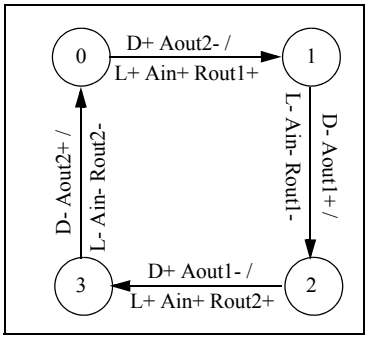


Bild 4.100: Zustandsgraph für die Kontrollstruktur TOGGLE_RUWLO

Die RUWLO Variante ist die von der Spezifikation her einfachste Kontrollschaltung. Somit bleibt der Automat klein. Bei anderen Kontrollschaltungen kann der Automat schnell sehr groß werden und damit entsprechend auch die Implementierung. Die Laufzeiten können unter Umständen erheblich größer werden. Anhand von Bild 4.101 erkennt man, dass selbst für die RUWLO Variante die Implementierung mehr als doppelt so groß ist, als die einfache RUWLO Schaltung (Bild 4.36).

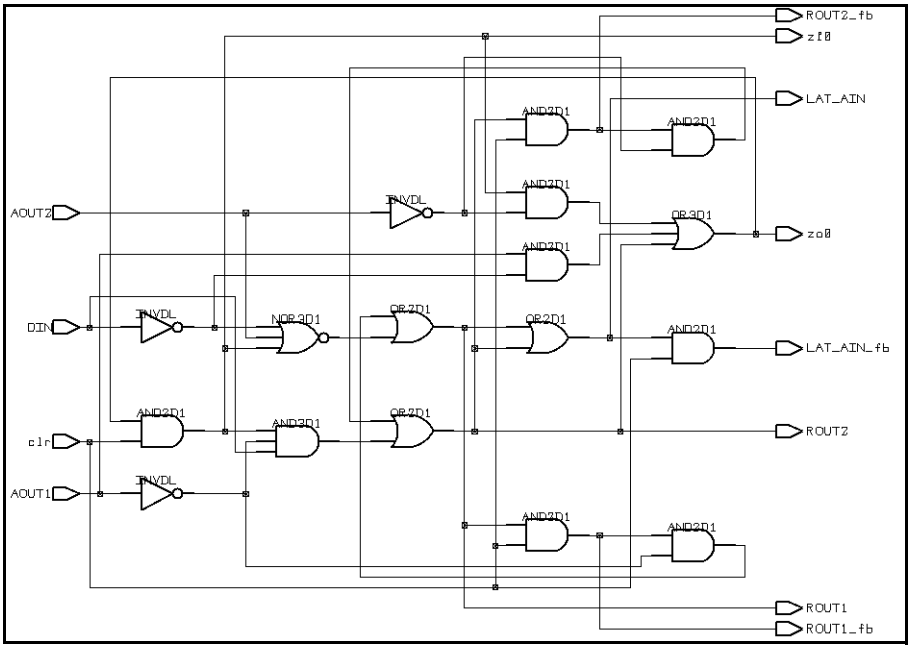


Bild 4.101: Implementierung der Schaltungen TOGGLE_RUWLO

Somit ist klar, dass ein TOGGLE Gatter mit mehr als 2 Folgestufen überproportional groß und langsam wird. Vor allem bei mehr werdenden Eingangssignalen die Notwendigkeit

zunimmt, auftretende interne Zeitbedingungen durch Verzögerungsglieder zu erfüllen. In diesem Fall ist zu überlegen, ob man nicht eine alternative Architektur wählt.

SELECT_RUWLO

Eine SELECT Schaltung besitzt die umgekehrte Funktion eines TOGGLE Gatters. Die aktuelle Stufe besitzt mehrere Vorgängerstufen, die abwechselnd Requests stellen. Dabei wird jeweils eine Vorgängerstufe durch das Nicht-Zurücksetzen des Acknowledge-Signals „blockiert". Die blockierte Stufe wird erst nach Beendigung des aktuellen Datenaustausches freigegeben.

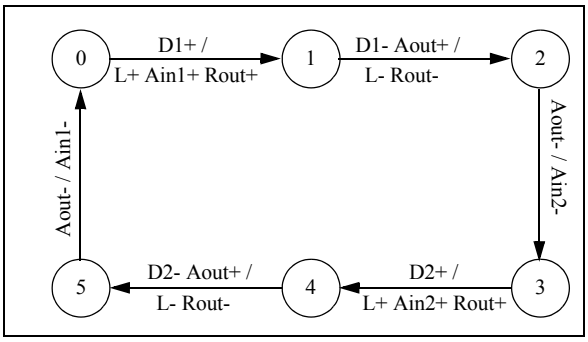


Bild 4.102: Zustandsgraph für die Kontrollstruktur SELECT_RUWLO

Durch die Einschränkung, dass die Eingänge abwechselnd Requests stellen dürfen, bleibt sowohl die Spezifikation als auch die Implementierung (Bild 4.103) überschaubar. Auch hier führen weitere Eingangsstufen zu einer raschen Ausweitung der Spezifikation sowie zu einer Vergrößerung der Schaltung. Eventuell muss man eine alternative Implementierung in Betracht ziehen.

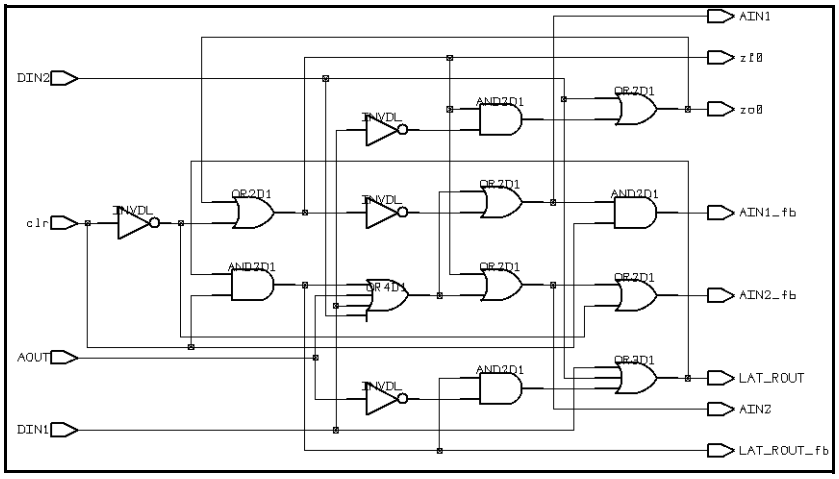


Bild 4.103: Implementierung der Schaltungen SELECT_RUWLO

CALL_RUWLO

Eine CALL Schaltung erlaubt den Request an die aktuelle Stufe von einer, zwei oder gegebenenfalls mehreren Vorgängerstufen.Wichtig ist hierbei, dass die Requests getrennt auftreten

müssen. Das heißt, es darf nur genau ein Request vorliegen. Somit müssen die Vorgängerstufen koordiniert werden, oder ein gegenseitiger Ausschluss (mutual exclusion) der Requests erzwungen werden. Ist dies der Fall, kann die in Bild 4.104 gezeigte Schaltung benutzt werden.

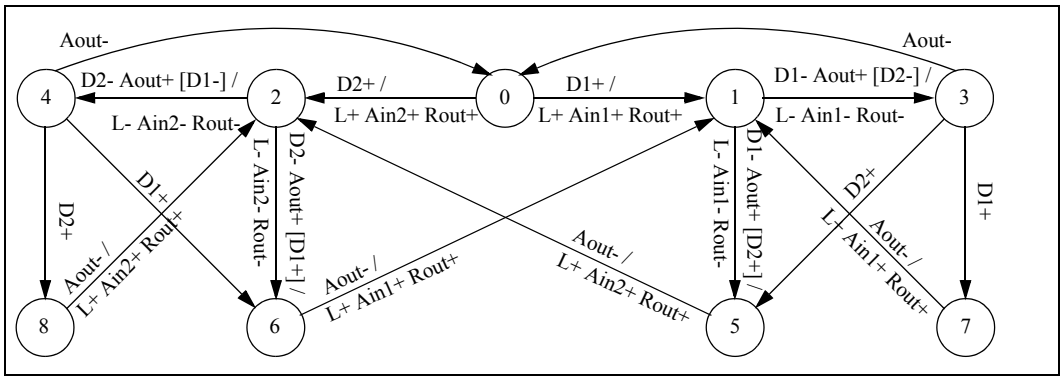


Bild 4.104: Zustandsgraph für die Kontrollstruktur CALL_RUWLO

Da nur eine Vorgängerstufe aktiv sein darf, erfolgt entweder ein Request von der Vorgängerstufe 1 oder der Vorgängerstufe 2. Nach Beendigung des Datenaustauschs kann wieder ein Request von einer der beiden Vorgängerstufen erfolgen.

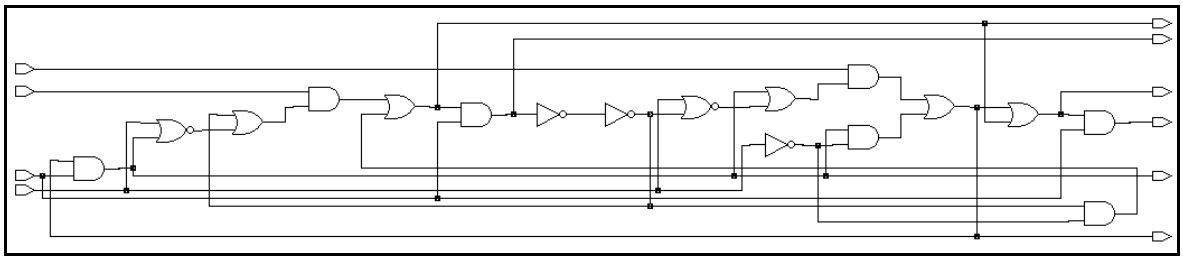


Bild 4.105: Implementierung der Schaltungen CALL_RUWLO

Bild 4.105 zeigt die Implementierung der CALL Schaltung. Sie ist sehr kompakt. Das liegt zum einen an der Einfachheit der RUWLO Schaltungen und zum anderen daran, dass hier das Problem des gegenseitigen Ausschlusses nach Außen verlagert wurde. Um diesen zu erreichen benötigt man zum Beispiel eine MUTEX Schaltung.

MUTEX

MUTEX steht für „mutual exclusion“, das heißt also einen gegenseitigen Ausschluss. Diese Schaltung wird für eine Arbitrierung benötigt, wenn also zum Beispiel zwei Requests gleichzeitig an eine CALL Schaltung getätigt werden. MUTEX Schaltungen sind nicht einfach zu entwickeln, da sie ein inhärentes Zeitproblem besitzen. Im Prinzip benötigt man eine Schaltung, die intern die vorkommenden metastabilen Zustände auflöst und nach außen hin nur definierte Zustände und Zustandsübergänge einnimmt. Somit wirkt sich der metastabile Zustand nur auf die Performanz aber nicht auf die Funktionalität aus.

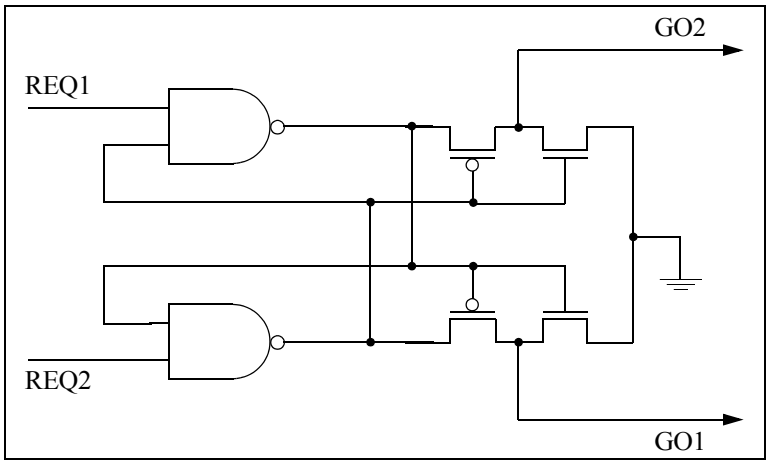


Bild 4.106: Analoge MUTEX Schaltung [59]

Eine analoge Realisierung einer MUTEX Schaltung ist in Bild 4.106 dargestellt. Der gegenseitige Ausschluss wird prinzipiell über zwei kreuzgekoppelte NAND Gatter erwirkt. Treten aber beide Requests einigermaßen zeitgleich auf, entsteht ein metastabiler Zustand an den Ausgängen der NAND Gatter. Durch den nachgeschalteten Filter wird sichergestellt, dass der metastabile Zustand nicht nach außen sichtbar wird.

Die analoge Realisierung hat den Nachteil, dass bei jedem Technologiewechsel diese Schaltung (per Hand) neu entworfen werden muss. Ziel dieser Arbeit ist aber ein automatisiertes Verfahren zur Verfügung zu stellen. Somit wäre es wünschenswert, wenn eine MUTEX Schaltung aus vorliegenden Zellen der verwendeten Technologiebibliothek erstellt werden kann.

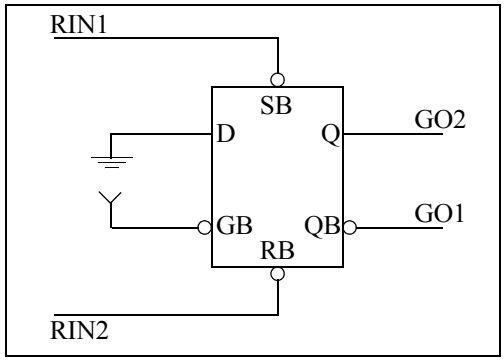


Bild 4.107: MUTEX Schaltung mit Hilfe eines RS-Latches

In Bild 4.107 ist eine mögliche Realisierung einer MUTEX Schaltung mit Hilfe von Standardzellen - hier ein RS-Latch - gezeigt. Letztlich ist es nicht möglich, eine MUTEX Schaltung aus logischen Grundgattern zu bauen, da immer eine Race - Bedingung vorliegen wird. Somit ist die Verwendung eines RS-Latches aus der für das aktuelle Design verwendeten Technologiebibliothek ein Kompromiss. Dieser Kompromiss gewährleistet den halbautomatisierten Entwurf von asynchronen Pipelines. Es muss nur zu Anfang das passende RS-Latch ausgewählt

und als MUTEX Schaltung verdrahtet werden. Nachteil hierbei ist eine eventuell vorliegende Zeitbedingung des RS-Latches zwischen dem Auftreten der beider Requests. Inwieweit eine Verletzung dieser Zeitbedingung die Funktionalität stört, muss im Datenbuch nachgelesen werden.

5 Beispiele

5.1 Digitale Fotosteuerung

Als ein erstes „komplexeres“ Beispiel ist hier die digitale Steuerung eines Fotoapparats. Es handelt sich hierbei um eine Schaltung, die Kontrollaufgaben erfüllt und somit erst einmal nicht der vorher beschriebenen Zielgruppe von datenverarbeitenden Schaltungen entspricht. Aber zum einen zeigt dieses Beispiel, dass mit ASMOGEN nicht nur Pipelines entworfen werden können, sondern auch komplexere Kontrollschaltungen. Zum anderen ist eine Eigenschaft der vorliegenden Schaltung, dass über einen längeren Zeitraum nur ein Modul arbeitet, nämlich der Zähler für die Belichtungszeit und den Weitertransport des Films.

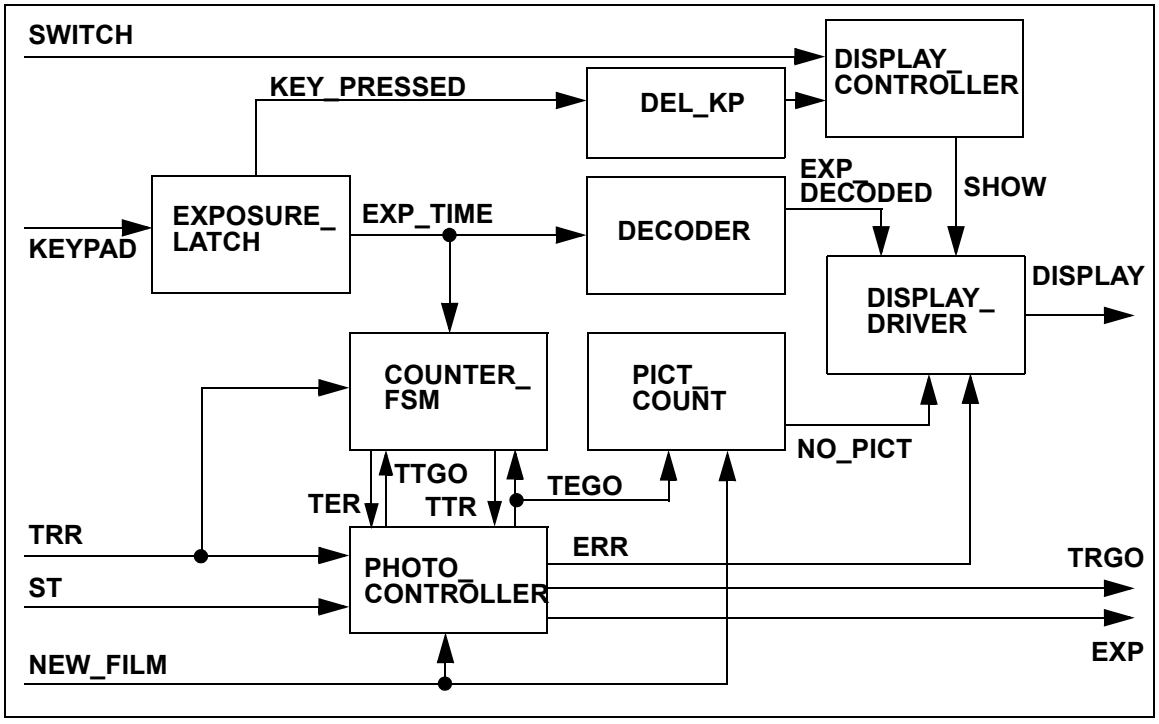


Bild 5.1: Blockdiagramm der synchronen Photosteuerung

In Bild 5.1 ist das Blockdiagramm für den synchronen Entwurf dargestellt. Über eine Tastatur kann eine Belichtungszeit eingestellt werden. Der über die Tastatur eingegebene Wert wird in EXPOSURE_LATCH gespeichert. Der gespeicherte Tastaturwert wird in DECODER für die spätere Anzeige einer Belichtungszeit kodiert und in COUNTER_FSM als Grenze für das Zählen der Belichtungszeit benutzt.

Das Drücken der Tastatur wird ferner einen Takt lang zwischengespeichert, um anschließend in DISPLAY_CONTROLLER zusammen mit SWITCH (Umschalten der Anzeige) ausgewertet zu werden. PICT_COUNT zählt die Anzahl der Bilder und PHOTO_CONTROLLER koordiniert den gesamten Ablauf.

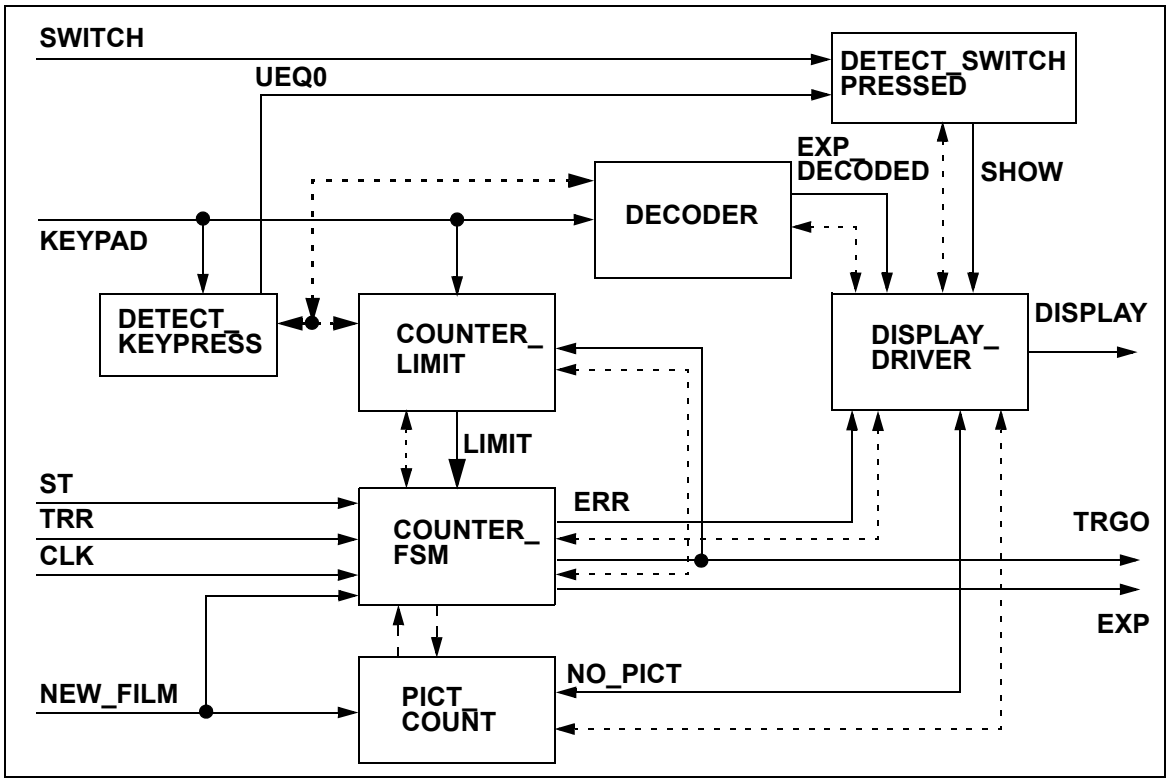


Bild 5.2: Blockdiagramm der asynchronen Photosteuerung

Das Blockdiagramm der asynchronen Realisierung der Fotosteuerung ist in Bild 5.2 dargestellt. Die gestrichelten Verbindungen repräsentieren die zusätzlich notwendigen Handshake-Signale. Der zentrale Controller aus der synchronen Realisierung ist nicht mehr notwendig und geht in den entsprechenden Kontrollschaltungen der einzelnen Module auf.

Eine besondere Eigenschaft der Fotosteuerung ist, dass die Eingangssignale KEYPAD und SWITCH als sehr langsam betrachtet werden können, da sie vom Benutzer der Fotokamera betätigt werden. Somit vereinfachen sich die Kontrollautomaten. Sie beschränken sich darauf, relevante Änderungen (zum Beispiel nur die steigende Flanke) der Eingangssignale zu detektieren und den entsprechenden Datenaustausch der nachfolgenden Module zu bewerkstelligen.

In der asynchronen Variante ist ein zusätzliches Modul notwendig (COUNTER_LIMIT), um die Zählergrenze festzulegen. Diese Aufgabe ist im synchronen Entwurf im Modul COUNTER_FSM selbst integriert. Ansonsten sind die Aufgaben der Module in der asynchronen Realisierung gleich der der synchronen Realisierung.

Die hier betrachtete Schaltung ist keine datenverarbeitende Pipeline und die entsprechenden Werte (zum Beispiel EXP_DECODED oder NO_PICS) werden benötigt, auch wenn sie sich gerade nicht geändert haben. Ein Beispiel ist das Umschalten der Anzeige (über SWITCH) von der Belichtungszeit (EXP_DECODED) auf die Anzahl der Bilder (NO_PICS). Somit wurden als Standardspeicher Flipflops und als Standardkontrollstruktur RUBFF (siehe Kapitel 4.2.3.19) gewählt.

Tabelle 9: Vergleich der Leistungswerte der synchronen Fotosteuerung mit der asynchronen Fotosteuerung

	Fläche (μm^2)	Energie (nWs)
SYNCHRON	13928	4,298
ASYNCHRON	20835	1,616
Gewinn	**-50%**	**62%**

Die gewonnenen Werte aus der Synthese und Simulation der beiden Realisierungen sind in Tabelle 9 gezeigt. Die entsprechende Laufzeit spielt bei diesem Beispiel keine Rolle, da die Umgebung um Größenordnungen langsamer und damit der die Zeit bestimmende Faktor ist. Der Flächenzuwachs durch die zusätzlich notwendigen Kontrollschaltungen beträgt 50% und ist damit nicht zu groß. Dahingegen beträgt der Gewinn beim Energiebedarf über 60%. Damit beträgt der Energiebedarf der asynchronen Realisation weniger als 40% gegenüber der synchronen Realisierung.

5.2 Reed Solomon Dekoder

Reed-Solomon Codes [76] [102] sind nach ihren Entdeckern Irvind S. Reed und Gustave Solomon benannt. Sie werden benutzt, um digitale Daten, die über einen Kanal versendet und damit gestört werden können, zu kodieren und auf der Empfängerseite zu dekodieren. Bei der Dekodierung können bis zu einem gewissen Grad Fehler korrigiert werden.

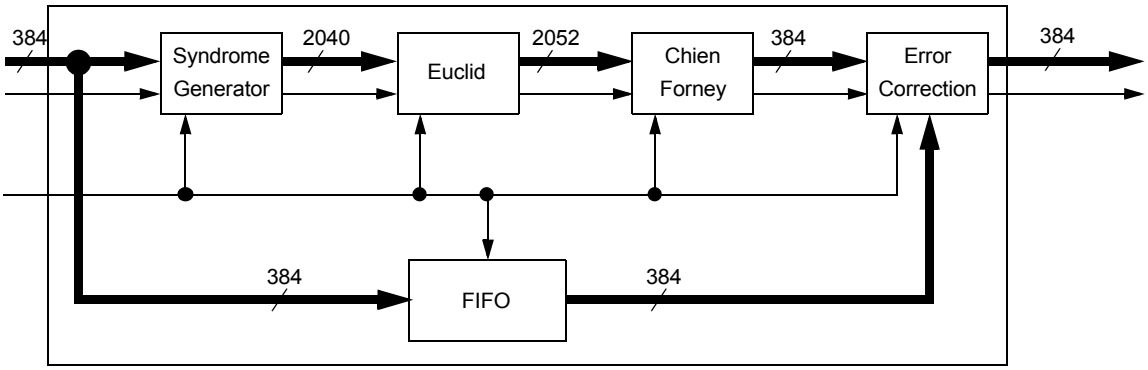


Bild 5.3: Blockdiagramm des Reed Solomon Dekoders

Die grobe Struktur eines Reed Solomon Dekoders ist in Bild 5.3 dargestellt. Das empfangene Datum wird in Schieberegistern (beziehungsweise Fifos) zwischengespeichert, um am Ende der Verarbeitung mit dem Korrekturwert verknüpft zu werden. Der Korrekturwert wird in mehreren Schritten ermittelt.

In dem Syndrome Generator werden aus den eingehenden Daten iterativ die sogenannten Syndrome generiert. Dies erfolgt in der Weise, dass die 2*t (t ist dabei eine Richtgröße für die Zahl der Fehler, die korrigiert werden können) Nullstellen des für die Kodierung benutzten Generatorpolynoms in das empfangen Polynom eingesetzt werden. Da 32 Symbole von je 12

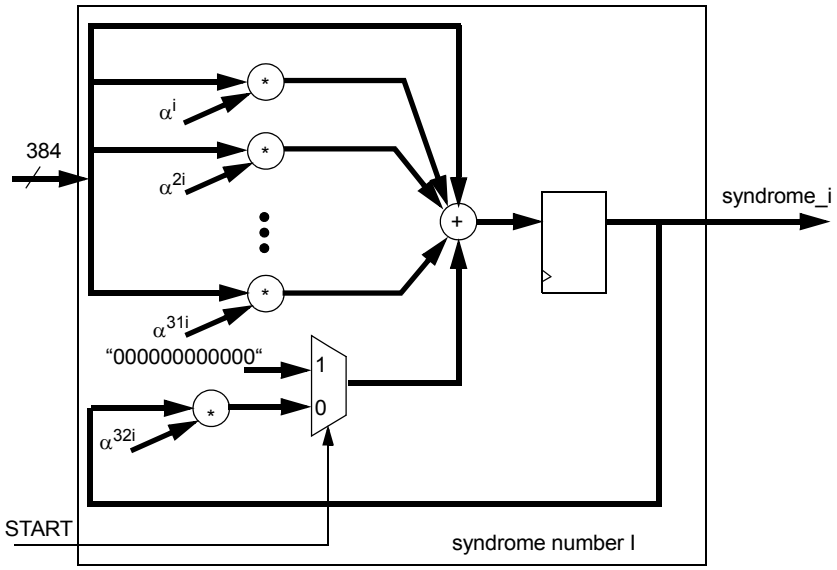


Bild 5.4: Blockdiagramm einer Syndrom Komponente

Bit parallel empfangen werden, müssen immer 32 Stellen des Polynoms gleichzeitig behandelt werden. Nach t Iterationen erhält man 2*t Syndromkomponenten. Ist das empfangene Polynom fehlerfrei, sind alle Syndromkomponenten gleich Null. Andernfalls liegen Fehler vor und die einzelnen Syndromkomponenten werden als Werte des Fehlerpolynoms an den Nullstellen betrachtet. So erhält man 2*t Gleichungen, um bis zu t Fehler zu berechnen.

Aus den im Syndromblock gewonnenen Werten für das beschriebene Gleichungssystem lassen sich auf unterschiedliche Weisen ein Fehlerortpolynom (Lamda) und ein Fehlerwertpolynom (Omega) erzeugen. Ein häufig verwendetes Verfahren ist der Berlekamp-Massey Algorithmus [10] [58]. Dieser Algorithmus enthält aber Divisionen, die bei einer Wortbreite von 12 Bit zu entsprechend langen Laufzeiten führen. Zusätzlich benötigt der Algorithmus im Allgemeinen 2*t Bearbeitungsschritte, muss aber nach t Schritten schon wieder ein neues Syndrompolynom verarbeiten können. Dieses Problem ist in einer synchronen Realisierung durch erheblichen Mehraufwand (zum Beispiel zwei hintereinander geschaltete Berlekamp-Massey Blöcke) lösbar. Aus diesen Gründen wurde die Alternative gewählt: der Euklidische Algorithmus.

Das Prinzip des Euklidischen Algorithmus wird auch gegenseitige Wechselwegnahme genannt. Eingangsgrößen sind zwei natürliche Zahlen a und b. Bei der Berechnung verfährt man nach Euklid wie folgt:

1. setze m = a; n = b
2. ist m < n, so vertausche m und n
3. berechne r = m - n
4. setze m = n, n = r
5. ist r ≠ 0 fahre fort mit Schritt 2

Als Ergebnis erhält man den größten gemeinsamen Teiler von a und b. Man kann den Algorithmus für das vorliegende Problem adaptieren, wobei a und b Polynome sind; die Abbruchbedingung ist, dass der Grad des Polynoms r nicht kleiner als t sein darf. Die dann vorliegenden Restpolynome repräsentieren das Fehlerort- und Fehlerwertpolynom.

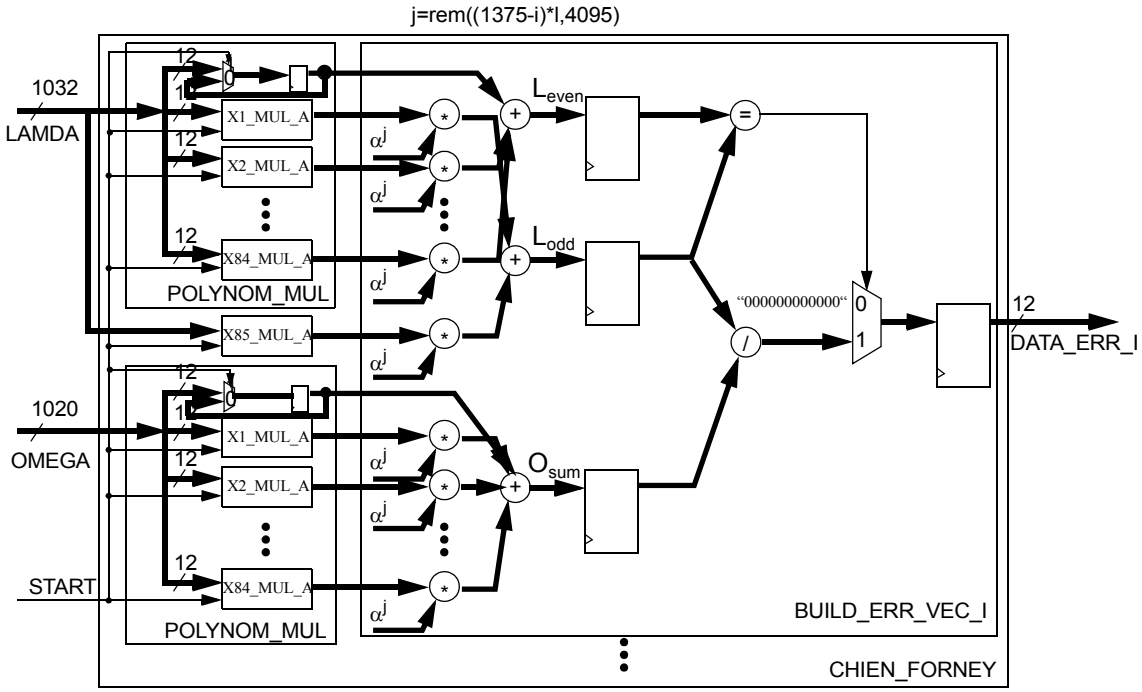


Bild 5.5: Blockdiagramm des Chien/Forney Blocks

In einem der letzten Schritte werden die tatsächlichen Fehlerorte und -werte bestimmt. Um die Fehlerorte zu berechnen bestimmt man die Nullstellen des Fehlerortpolynoms (Lamda) [18]. Es werden hierzu der inverse Wert der gerade betrachteten Stelle in das Fehlerortpolynom eingesetzt. Ist der Polynomwert Null ($L_{even} = L_{odd}$), liegt an dieser Stelle ein Fehler vor. Der aktuelle Fehlerwert (wieder wird der inverse Wert eingesetzt) ergibt sich aus der Division des Fehlerwertpolynoms (O_{sum}) mit dem Wert des abgeleiteten Fehlerortpolynoms (ist gleich dem L_{odd}) [23]. Nur im Fehlerfall wird ein Korrekturwert ausgegeben.

Als letzter Schritt wird der aktuelle Korrekturwert mit dem entsprechenden Symbol verknüpft und als korrigierter Wert ausgegeben.

Ergebnisse

Zur Vereinfachung wird für beide Realisierungen angenommen, dass die vorgeschaltete Stufe die Daten synchron anlegt. Somit unterscheidet sich der Aufbau der beiden Realisierungen kaum. Aufgrund der iterativen Arbeitsweise des Syndromegenerators werden auch für die asynchrone Realisierung Flipflops verwendet, die von dem globalen (synchronen) Takt gesteuert werden. Im Gegensatz zur synchronen Realisierung fällt jedoch die Erzeugung eines „Fertig“ Signals weg, da die nachfolgende Stufe (EUKLID-Block) das „START“-Signal der dem Syndromgenerator vorgeschalteten Stufe verwenden kann.

Tabelle 10: Vergleich der Leistungswerte des synchronen Syndromgenerators mit denen des asynchronen Syndromgenerators

	Fläche (μm^2)	Power (mW)	Synthesezeit	Simulationszeit (Gatterebene, Toggelraten)
SYNCHRON	6255917	8.24	~2 Tage	~6 Tage
ASYNCHRON	6256630	5.23	~2 Tage	~6 Tage
Gewinn	**0%**	**37%**		

Die aus der Synthese gelieferten Flächen- und Verbrauchswerte für den **Syndromgenerator** sind in Tabelle 10 dargestellt. Da beide Realisierungen mit dem globalen Takt gesteuert werden, ist die Laufzeit bei beiden gleich und hier nicht aufgelistet. Die Zeiten[1] für die Synthese und die Simulation sind für beide Entwürfe gleich. Beide Zeiten zeigen aber, in welcher Größenordnung das Modul einzuordnen ist. Der Verbrauch der asynchronen Realisierung ist deutlich niedriger als erwartet. Schließlich wurden nur wenige Schaltungsteile gegenüber der synchronen Schaltung weggelassen. Als Erklärung bleibt nur übrig, dass die Default-Einstellungen für die Synthese der asynchronen Schaltungen sich von den Einstellungen der synchronen Schaltungen unterscheidet und die Synthesesoftware in der asynchronen Realisierung eine für den Verbrauch günstigere Lösung gefunden hat.

Tabelle 11: Vergleich der Leistungswerte des synchronen Euklid-Blocks mit denen des asynchronen Euklid Blocks

	Fläche (μm^2)	Power (mW)	Synthesezeit	Simulationszeit (Gatterebene, Toggelraten)
SYNCHRON	5905646	2.51	~5 Tage	~8 Tage
ASYNCHRON	5767533	1.89	~5 Tage	~8 Tage
Gewinn	**2%**	**25%**		

In Tabelle 11 sind die Vergleichswerte für den jeweiligen **Euklid**-Block gezeigt. Auch hier sind wieder zur Einordnung der Komplexität des Designs die Synthese- und Simulationszeiten angegeben. Obwohl der Euklid Block nicht so groß ist, wie zum Beispiel der Syndromgenerator, benötigt es doch auf Grund der vielen verwendeten Arithmetik einige Zeit und vor allem viel Arbeitsspeicher[2], ihn zu synthetisieren und zu simulieren. Die asynchrone Realisierung nutzt in der ersten Stufe das Frame-Signal, um ein neues Datum zu übernehmen. Dieses Frame-Signal erfolgt jedesmal, wenn ein neuer Frame angefangen wird und dient in beiden Realisierungen zur Synchronisation bezüglich des Frameformats. In der zweiten Stufe, der eigentlichen

1. Gemessen auf einer Sun-Fire-280R mit 8GB physikalischen Arbeitsspeicher und zwei 2x Ultra-SPARC-III+ Prozessoren.
2. Ein einzelner Prozeß darf auf obiger Sun Workstation nur bis zu 2GB Arbeitsspeicher belegen.

Realisierung des Euklid-Algorithmus erfolgen wiederum iterative Berechnungen, so dass für die asynchrone Realisierung wiederum Flipflops als Speicher und als Kontrollstruktur RUBFF gewählt wurden. Die asynchrone Implementierung befindet sich in Ruhe, sobald die Schlüsselgleichungen gefunden wurden. Erst wenn die nachfolgende Stufe (Chien-Forney Block) die Daten übernommen hat und die vorgehende Stufe neue Daten meldet, fängt der Block wieder an zu „arbeiten“. Somit besitzt dieser Block einen deutlichen Vorteil beim Energieverbrauch. Dies spiegelt sich auch in den ermittelten Werten wider. Die asynchrone Lösung verbraucht rund 25% weniger Energie.

Die Fläche der asynchronen Realisierung ist geringer, da das Treibernetzwerk kleiner als in der synchronen Lösung ist. Während die synchrone Variante alle 8284 Flipflops mit einem Netz treiben muss, teilt sich dies bei der asynchronen Variante auf 2078 und 6206 Flipflops auf. Außerdem benötigt die asynchrone Lösung weniger Logik im Datenpfad, da sie ein paar Zähler weniger benötigt.

Tabelle 12: Vergleich der Leistungswerte des synchronen Chien-Forney Blocks mit denen des asynchronen Chien-Forney Blocks

	Fläche (μm^2)	Power (mW)	Synthesezeit	Simulationszeit (Gatterebene, Toggelraten)
SYNCHRON	13045800	9,89	~5 Tage	~8 Tage
ASYNCHRON	8734152	7,14	~5 Tage	~8 Tage
Gewinn	**33%**	**28%**		

Die Vergleichswerte für den **Chien-Forney** Block zusammen mit dem **Error Correcting** Block (**ECU**) sind in Tabelle 12 gezeigt. Auch dieser Block besitzt eine große Komplexität, was auch die Synthese- und Simulationszeiten zeigen. Die asynchrone Realisierung benutzt in den ersten Stufen wegen den iterativen Arbeitsschritten wiederum die RUBFF Kontrollstruktur und in der letzten Stufe (ECU) die AS2SY Struktur, also einer Schnittstelle zur nachfolgenden synchronen Seite. Ansonsten wurden hier keine wesentlichen Änderungen gegenüber der synchronen Realisierung vorgenommen. Der Flächenunterschied ist hier sehr deutlich; neben den unterschiedlichen Syntheseparametern, kommen hier wieder das Weglassen von nicht mehr notwendiger Logik im Datenpfad der asynchronen Lösung zum Tragen und vor allem wieder die kleineren einzelnen Treibernetzwerke in der asynchronen Lösung. In der synchronen Lösung muss das Taktnetzwerk 3976 Flipflops treiben. Die asynchrone Variante muss Flipflops in vier getrennten Gruppen zu 2053, 2 mal 385 und 1153 Flipflops treiben. Im Endeffekt sind in der synchronen Schaltung mehr als doppelt so viele Treiber vorhanden als in der asynchronen Schaltung. Die asynchrone Schaltung hat noch den Vorteil, dass das Treibernetzwerk im gleichen Syntheselauf wie der Rest der Schaltung erzeugt wird und damit ebenfalls optimiert wird. In der synchronen Schaltung wird standardmäßig zuerst die gesamte Schaltung synthetisiert und anschließend erst das Taktnetz zumeist „von Hand“ generiert.

Beim Verbrauch macht sich erneut das große Treibernetzwerk in der synchronen Realisierung bemerkbar. Der ermittelte Wert liegt um 28% über dem der asynchronen Implementierung.

Die Ergebnisse in den drei Tabellen zeigen, dass die asynchrone Variante klare Vorteile gegenüber der synchronen Variante besitzt. Neben des zu erwartenden niedrigeren Verbrauchs besitzt die asynchrone Implementierung eine noch deutlich kleinere Fläche. Letzteres liegt vor allem am Taktnetz. Da in der synchronen Realisierung ein einziges Treibernetzwerk alle im Design vorkommenden Speicher treiben muss, ist dieses entsprechend mit sehr vielen Buffern zu versehen. Die im Rahmen dieser Untersuchung erzeugten Treibernetzwerke wurden automatisiert durch die Synthesesoftware Synopsys eingefügt. Das heißt diese sind nicht optimal und würden auf jeden Fall anders ausfallen. Gleiches gilt allerdings auch für die asynchrone Realisierung. Somit galt für beide Varianten die gleiche Voraussetzung. Trotzdem würde sich durch vernünftige Treibernetzwerke der Vorteil der asynchronen Schaltungen schmälern. Allerdings ist nicht zu erwarten, dass er gänzlich verschwindet.

6 Zusammenfassung und Ausblick

6.1 Umfang der Arbeit

Ausgehend vom derzeitigen Stand der Technik wurde in der vorliegenden Arbeit eine **Methodik** für den systematischen (semi-) automatischen Entwurf asynchroner Schaltungen in einer Entwurfsumgebung für synchrone Schaltungen erarbeitet. Die erzielten Ergebnisse dieses Vorgehens sind, dass

- durchgehend Standardzellen für den gesamten Entwurf verwendet werden können. Speziell müssen keine neuen Gatter entwickelt werden, wie zum Beispiel Muller-C-Elemente (Mikropipelines) oder Mehrheitsgatter (NCL).

- auf etablierte und erprobte Entwicklungssoftware zurückgegriffen werden kann. Ausnahme ist hierbei die benutzte Software DGC zur Synthese der asynchronen Kontrollschaltung. Das Benutzen altbekannter Entwicklungssoftware hat mehrere Vorteile. Zum einen sind keine größeren Umstellungen eines etablierten Entwurfsablaufs synchroner Schaltungen erforderlich (Zeit und Kosten). Zum anderen steigert dies die Chance der Akzeptanz bei den Designern, da diese sich ebenfalls nicht komplett umstellen müssen.

- der Entwurf gemischt synchroner und asynchroner Systeme in einer Entwicklungsumgebung erfolgen kann. Somit ist hier keine harte Grenze beim Entwurf der beiden Varianten gesetzt. Es ist kein Wechsel der Software beziehungsweise der Entwicklungsumgebung notwendig, und die einzelnen Module können problemlos zusammen entworfen und verifiziert werden.

Die Eingliederung in eine synchrone Entwurfsumgebung wurde nicht zuletzt dadurch erreicht, dass der Entwurf des Datenpfades weitestgehend genauso erfolgt wie bei einer synchronen Schaltung. Der Entwurf der asynchronen Kontrolllogik kann per Hand unter Einhaltung bestimmter Regeln (Hazardfreiheit, Complete Stat Coding und andere) erfolgen. Allerdings ist dies aufwendig und das Hauptziel war ein möglichst automatisierter Entwurf. Deshalb erfolgt der Entwurf der Kontrolllogik mit der frei verfügbaren Synthesesoftware DGC (http://sourceforge.net/projects/dgc/). DGC erzeugt hazardfreie asynchrone (extended) burst mode Automaten. Die Eingliederung des Entwurfes der asynchronen Kontrolllogik ist so erfolgt, dass möglichst wenig Änderungen des Entwurfsablaufs synchroner Schaltungen notwendig sind.

Ein weiterer Schwerpunkt dieser Arbeit ist die **Untersuchung der Vierphasenprotokolle**. Zu diesem Zweck wurden Klassifizierungsmerkmale festgelegt und alle sinnvollen Kombinationsmöglichkeiten aufgestellt. Die klassifizierten Vierphasenprotokollstrukturen wurden einge-

hend untersucht und bewertet. Es wurden Optimierungsmöglichkeiten herausgearbeitet und die optimierten Schaltungen ebenfalls untersucht. Der durch die Untersuchung entstandene Katalog an Vierphasenprotokollstrukturen wurde um etliche weitere Schaltungen erweitert. Zu diesen zählen Konverter für die Zusammenschaltungen der unterschiedlichen Kontrollstrukturen, Kontrollstrukturen für das Zweiphasenprotokoll, Schnittstellenschaltungen zur Anbindung an synchrone Module und verschiedene häufig benötigte Sonderstrukturen, wie Join, Fork, Call und andere. Dieser **Katalog an Kontrollstrukturen** ist in einer erweiterbaren Bibliothek zusammengefasst worden.

Die oben beschriebene Bibliothek an Kontrollstrukturen ist in die, in dieser Arbeit ebenfalls geschriebene **Software ASMOGEN** eingebunden. Zu den Aufgaben von ASMOGEN zählt das Einrichten einer Projektumgebung, sprich das Erzeugen notwendiger Verzeichnisse, Setzen von Umgebungsvariablen, Einbinden der Bibliothek an Kontrollstrukturen. Sie steuert den Gesamtablauf des Entwurfes, inklusive der Laufzeituntersuchung und entsprechender Laufzeitkorrektur durch Einfügen von Verzögerungsgliedern. Werden in einer Setup Datei (.async_setup) alle notwendigen Einstellungen getätigt und auf in der Bibliothek vorhandene Kontrollstrukturen zurückgegriffen, läuft alles automatisch ab. Sollen besondere Kontrollstrukturen verwendet werden, sind diese im entsprechenden Pfad als extended burst mode Spezifikationen abzulegen. Danach erfolgt der Entwurf wiederum automatisch, es sei denn, der Entwickler möchte dies nicht und gibt entsprechende Argumente beim Aufruf von ASMOGEN an.

Die Methodik, die Bibliothek und die Entwicklungssoftware ASMOGEN wurden im Rahmen dieser Arbeit auch erprobt und erfolgreich eingesetzt. Die mögliche volle Automatisierung kam bei der Untersuchung der Vierphasenprotokolle zum Tragen. Die Leistungsfähigkeit bei großen Schaltungen, die ihrerseits auch besondere Kontrollschaltungen verlangen, wurde anhand des Beispiels eines **Read Solomon Dekoders** gezeigt.

6.2 Ausblick

Die vorgestellte Methodik hatte zum Ziel, den Entwurf asynchroner Schaltungen in einer Entwurfsumgebung für synchrone Schaltungen zu ermöglichen. Somit orientiert sich der Ablauf an dem Entwurf synchroner Schaltungen. Dies hat zur Folge, dass aber bestimmte Realisierungsvarianten für den Datenpfad ausgeschlossen wurden. Speziell wurden „unbounded delay“ - Schaltungsvarianten, die eine einfache Komplettierungsdetektion zulassen, ausgeschlossen, wie es zum Beispiel *dual rail* Implementierungen erlauben. Statt dessen wurde das bounded delay Modell für den Datenpfad festgelegt und entsprechend Verzögerungen in den passenden Kontrollpfaden eingefügt. Aus diesen Erkenntnissen lassen sich einige Punkte für Erweiterungen und Verbesserungen ableiten:

Optimierung der Bestimmung der notwendigen Verzögerungsglieder

Die Verwendung von Verzögerungsgliedern ergibt sich aus der Tatsache, dass für den Datenpfad das bounded delay Modell verwendet und kein Komplettierungsmechanismus verwendet wird. Weitere einzubauende Verzögerungen ergeben sich aus den Zeitbedingungen, die von DGC vorgegeben werden, also durch die Implementierung der Kontrolllogik. Die Größe der einzufügenden Verzögerung wird aus den Vorgaben der Syntheseergebnisse (Daten- und Kontrollpfad) vorab berechnet und später in der Simulation nur auf zu kurze Verzögerungen überprüft. Im Moment findet noch keine automatische Untersuchung und Aufdeckung von zu großen Verzögerungen statt, so dass diese vom Designer selbst verkleinert werden müssen. Es ist zu überlegen, ob die Abschätzung der benötigten Verzögerung verbessert werden kann; zum Beispiel wird momentan eine *worst case* Betrachtung zur Berechnung der Verzögerung benutzt, was etwas zu pessimistisch sein dürfte.

Einfügen eines Treibernetzes für das Latch-Signal

Das Beispiel „Reed Solomon Dekoder“ hat gezeigt, dass die in dieser Arbeit entwickelte Software auch für große Schaltungen anwendbar ist. Allerdings müssen in großen Schaltungen oftmals in einer Pipeline-Stufe sehr viele Speicher durch ein Latch-Signal gesteuert werden. Die dadurch entstehende Last sorgt für eine zu große Anstiegszeit des Latch-Signals. So betrug die Anstiegszeit eines Latch-Signals im obigen Beispiel 139 ns. Dies ist natürlich um Faktoren zu groß.

Eine Möglichkeit, dieses Problem zu beheben, ist es, statt einer Kontrollstruktur mehrere parallel zu verwenden. Dafür benötigt man dann noch voraus- und nachgeschaltet Kontrollschaltungen, die zusätzliche Fläche, Energie und vor allem Laufzeit benötigen.

Eine andere und wohl geschicktere Möglichkeit ist es, ähnlich wie im synchronen Entwurf das Taktnetz hier das Netz des Latch-Signals mit Treibern zu versehen. Es ist möglich, die durch das Einfügen des Treibernetzes entstehende Verzögerung in die Berechnung der notwendigen Verzögerung einfließen zu lassen. Somit entsteht im optimalen Fall gegenüber der ursprünglichen Schaltung keine zusätzliche Laufzeit durch die Kontrollschaltung. Es wird lediglich mehr Fläche und mehr Energie benötigt.

Erweiterung der Methodik

Wie vorher schon angedeutet, beschränkt sich die erarbeitete Methodik, wie auch die erstellte Software, auf ein bounded delay Modell für den Datenpfad. Es ist denkbar, diese Einschränkung aufzuheben, und zum Beispiel unbounded dealy Modelle zuzulassen. Speziell delay-insensitive Schaltungen wären hier eine interessante Möglichkeit.

Der Vorteil wäre, dass das Fertig-Signal nicht durch Verzögerung des Request-Signals der Vorgängerstufe erzeugt werden müsste, sondern ein Komplettierungsmechanismus benutzt werden kann. Somit würde die durchschnittliche Performanz gegenüber der in dieser Arbeit gewählten Implementierungsvariante sich eventuell verbessern.

Nachteile wären der zusätzliche Flächenbedarf und der vermutlich im Mittel zu erwartende höhere Energiebedarf für den Datenpfad. Ein weiterer Nachteil ist im Zusammenhang mit dem Latch-Signal zu sehen. Das in großen Schaltungen teilweise notwendige Treibernetzwerk für das Latch-Signal (siehe oben) erzeugt zusätzliche Verzögerungen. Benutzt man einen Komplettierungsmechanismus, kann diese zusätzliche Verzögerung an keiner Stelle der Kontrollschaltung mehr gegengerechnet werden. Somit wirkt sich diese zusätzliche Verzögerung direkt auf die Performanz der Schaltung aus. Eventuell verschlechtert sich hierdurch die Performanz, anstatt, wie erhofft besser zu werden.

Anhang A
Entstehung von Verlustleistung

Es gibt drei verschiedene Ursachen für Verlustleistung in digitalen CMOS-Schaltungen [77] [56]:

$$P_{Total} = P_S + P_{KS} + P_{Le} \qquad (A.1)$$

- $\mathbf{P_S}$ repräsentiert die Schaltverluste, die durch das Auf- und Entladen von Kapazitäten entstehen.
- $\mathbf{P_{KS}}$ sind Verluste, die durch direkte Kurzschlussströme entstehen, die beim Schalten der CMOS-Schaltung auftreten, weil der NMOS- und PMOS-Transistor gleichzeitig leiten.
- $\mathbf{P_{Le}}$ stellen Leckverluste dar, die durch das Sperrverhalten von Dioden und Transistoren hervorgerufen werden.

Die verschiedenen Mechanismen tragen mit unterschiedlichen Anteilen zum gesamten Leistungsverbrauch P_{Total} bei. In [56] wird die in Bild A.1 abgebildete Aufteilung vorgestellt, die für konventionelle CMOS-Schaltungen gültig ist. Bei integrierten Schaltungen der neuesten Generationen kann der Anteil der Leckverluste allerdings bis zu 20% betragen.

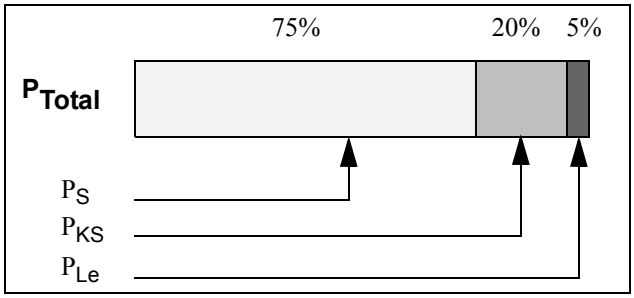


Bild A.1: Aufteilung der Verlustleistungen

P_S verursacht den Hauptanteil der gesamten Verlustleistung, weshalb sich viel Methoden in der Literatur auf die REduzierung dieses Anteil beschränken.

A.1 Verlustleistung durch Umladen der Lastkapazität

Zur Verdeutlichung, wie die Schaltverlustkomponente bei der Verlustleistung entsteht, wird in Bild A.2 ein konventioneller CMOS-Inverter betrachtet [56].

C_L am Ausgangsknoten steht für die Summe aller externen und internen (parasitären) Lastkapazitäten des Inverters. V_{out} stellt den Spannungsabfall über C_L dar. Findet am Eingang des

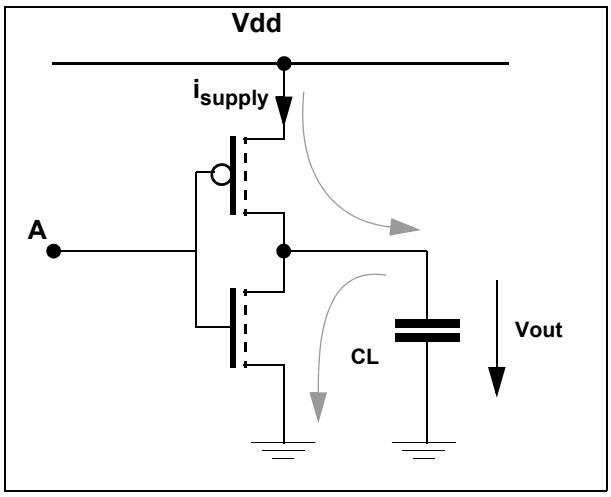


Bild A.2: Modell eines CMOS-Inverters

Inverters ein Signalwechsel statt, der zu einem 0→1 Wechsel am Ausgang führt, so fließt über den PMOS-Transistor der Ladestrom i_{supply}. Dieser ist:

$$i_{supply} = C_L \frac{dV_{out}}{dt} \qquad A.2$$

Weiter gilt für die Leistung P(t):

$$P(t) = \frac{dE}{dt} = i_{supply} \cdot V_{dd} \qquad A.3$$

Mit Gleichung *A.2* und Gleichung *A.3* kann die Energie berechnet werden, die bei diesem Ladevorgang von der Versorgungsquelle geliefert wird:

$$E_{0\to 1} = \int_0^T P(t)dt = V_{dd}\int_0^T i(t)_{supply}dt = V_{dd}\int_0^{V_{dd}} C_L dV_{out} = C_L \cdot V_{dd}^2 \qquad A.4$$

Der Anteil der Energie, die in C_L gespeichert wird, ist nun die Hälfte davon:

$$E_{cap} = \int_0^T P_{cap}(t)dt = \int_0^T V_{out} i_{cap}(t)dt = \int_0^{V_{dd}} C_L V_{out} dV_{out} = \frac{1}{2} C_L \cdot V_{dd}^2 \qquad A.5$$

Beim Übergang am Ausgang des Inverters (V_{out}), tritt somit folgende Energieverteilung auf:

- Energie, die aus der Spannungsversorgung abfließt: $C_L V_{dd}^2$
- Energie, die in C_L gespeichert wird: $\frac{1}{2} C_L V_{dd}^2$

• Energieverlust am PMOS-Transistor, der als Verlustwärme abgegeben wird: $\frac{1}{2}C_L V_{dd}^2$

Beim Übergang von $V_{dd} \rightarrow 0$ am Ausgang (V_{out}) wird die ganze in C_L gespeicherte Energie $\frac{1}{2}C_L V_{dd}^2$ über den NMOS-Transistor ebenfalls als Verlustwärme abgegeben. Weitere Energie wird bei diesem Entladevorgang nicht aus der Versorgungsquelle bezogen. Die beim Aufladeprozess aufgenommene Leistung wird somit bei konventionellen CMOS-Strukturen bei jedem Schaltzyklus, der aus Auf- und Entladen von C_L besteht, zu 100% als Wärme abgegeben.

Wenn ein Paar von steigenden und fallenden Übergängen an der Last-Kapazität mit einer Rate von f_{Clock} auftreten, dann berechnet sich die gesamte Leistung, welche von der Energiequelle abfließt, folgendermaßen:

$$P_{Drawn} = C_L V_{dd}^2 f_{Clock} \qquad A.6$$

Gleichung *A.6* gilt aber nur für den Takt selbst. Im Allgemeinen treten Schaltaktivitäten am Inverter nicht mit der Rate f_{Clock} auf. Um dies zu berücksichtigen, wird die Schaltwahrscheinlichkeit $\alpha_{0\rightarrow 1}$ eingeführt. $\alpha_{0\rightarrow 1}$ steht für die Wahrscheinlichkeit, dass innerhalb einer Taktperiode der Ausgang vom logischen Wert ‘0’ auf den logischen Wert ‘1’ wechselt.

Somit kann man allgemein die durchschnittliche Schaltverlustleistung eines CMOS-Gatters wie folgt definieren:

$$P_S = \alpha_{0\rightarrow 1} C_L V_{dd}^2 f_{Clock}; \quad (0 \le \alpha_{0\rightarrow 1} \le 0,5) \qquad A.7$$

In der Literatur ist auch der allgemeine Faktor α für die generelle Schaltwahrscheinlichkeit eines Gatters geläufig. Bei Verwendung dieses Faktors muss man beachten, dass hier pro betrachteten Schaltvorgang (Laden oder Entladen) nur die halbe Energie verbraucht wird. Andererseits werden alle zwei möglichen Wertewechsel berücksichtigt. Durch die Beziehung $\alpha = 2\alpha_{0\rightarrow 1}$ sind beide Darstellungsvarianten A.7 und A.8 gleichwertig.

$$P_S = \frac{\alpha}{2} C_L V_{dd}^2 f_{Clock}; \quad (0 \le \alpha \le 1) \qquad A.8$$

C_L und $\alpha_{0\rightarrow 1}$ wird in der Literatur häufig zu der effektiven Kapazität C_{eff} zusammengefasst:

$$C_{eff} = \alpha_{0\rightarrow 1} C_L \qquad A.9$$

Wie aus Gleichung *A.7* und Gleichung *A.9* ersichtlich, kann der Einfluss von P_S durch folgende Maßnahmen verringert werden:

• Verkleinerung der tatsächlichen Kapazität C_L

• Verringerung der Schaltaktivität α

• Reduzierung der Betriebsspannung V_{dd}

A.2 Verlustleistung durch Leckströme

Es gibt zwei Arten von Leckströmen: Dioden-Sperrstrom I_{Le} an den Anschlüssen der Transistoren und Sub-Threshold-Leckstrom I_{ds} des ausgeschalteten Transistors.

$$P_{Le} = (I_{Le} + I_{ds}) \cdot V_{dd} \qquad A.10$$

Die Größe der Ströme wird durch die Prozesstechnologie und die Arbeitstemperatur bestimmt. Es gibt jedoch einige Möglichkeiten für den Entwickler, den Einfluss dieser Ströme zu minimieren [14][56].

Dioden-Sperrstrom. Dieser Leckstrom I_{Le} tritt auf, wenn ein Transistor ausgeschaltet ist und entsprechende Sperrspannung am pn-Übergang anliegt [56].

$$I_{Le} = A_D \cdot J_S \qquad A.11$$

A_D ist hierbei die Fläche des Diffusionsgebietes am Drain-Anschluss und J_S ist die technologie- und temperaturabhängige Stromdichte des Leckstromes.

Sub-Threshold Leckstrom. Sub-Threshold-Ströme I_{ds} entstehen durch Ladungsträgerdiffusion zwischen Source und Drain wenn die Gate-Source-Spannung V_{gs} den Flachband-Inversionspunkt überschritten hat, aber noch unterhalb der Schwellspannung V_{Th} liegt. Unter diesen Umständen verhält sich der MOS-Transistor wie ein Bipolartransistor und I_{ds} ist gegeben durch:

$$I_{ds} = \kappa \cdot e^{\frac{V_{gs} - V_{Th}}{n V_T}} \cdot \left[1 - e^{\frac{V_{ds}}{V_T}} \right] \qquad A.12$$

κ ist ein Faktor, der durch die Prozesstechnologie bestimmt wird. $V_T = (kT)/q$ entspricht der Temperatur-Spannung, V_{ds} ist die Drain-Source-Spannung und V_{Th} die Schwellspannung [56].

A.3 Verlustleistung durch Kurzschlussströme

Es wird angenommen, dass V_{ThN} und V_{ThP} die Schwellspannungen des NMOS- und PMOS-Transistors sind und V_{In} die am Eingang anliegende Spannung ist.

Ist die Ungleichung $V_{ThN} < V_{In} < V_{dd} - |V_{ThP}|$ erfüllt, dann gibt es einen leitenden Pfad zwischen der Versorgungsspannung V_{dd} und dem Grundpotential (GND), da sowohl der NMOS- wie auch der PMOS-Transistor gleichzeitig leiten. Der Strom, der zwischen V_{dd} und GND

fließt, wird als Kurzschlussstrom I_{KS} bezeichnet. Ein deutlicher Anteil der Verlustleistung bei CMOS Schaltungen wird durch I_{KS} verursacht und wird wie folgt definiert [77]:

$$P_{KS} = \alpha_{0\to 1} \cdot I_{KS} \cdot V_{dd} \cdot \frac{\tau_{in}}{4} 2 f_{Clock} \approx \alpha_{0\to 1} \frac{C_L}{10} V_{dd}^2 f_{Clock} \qquad A.13$$

Hierbei ist τ_{in} die angenommene Anstiegs- und Abfallszeit des Eingangssignals.

Der durchschnittliche Kurzschlussstrom I_{mean} ist ausgeprägter, wenn die Anstiegs- beziehungsweise Abfallzeit des Eingangs sehr viel größer als die des Ausgangs ist. Das kommt daher, dass in diesem Fall der leitende Pfad länger existiert.

Die Anstiegs- und Abfallzeiten am Eingang beziehungsweise Ausgang einer CMOS-Schaltung werden unter anderem durch die Ausgangskapazität bestimmt, was anhand Bild A.3 näher erläutert werden soll.

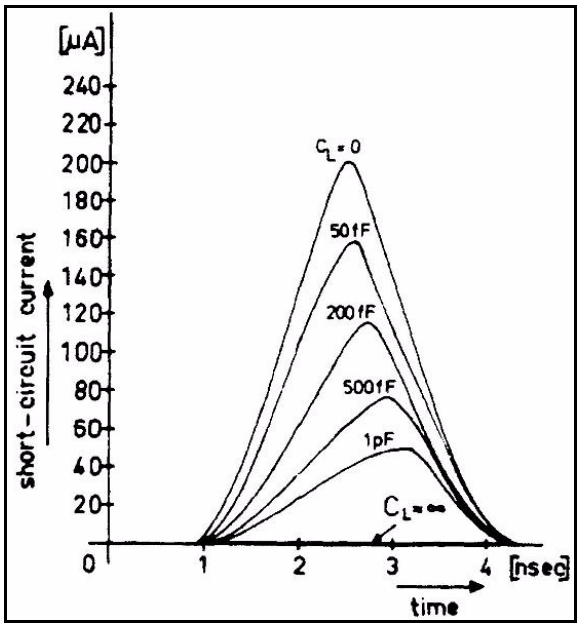


Bild A.3: Kurzschlussströme für unterschiedliche Ausgangskapazität C_L bei einer Inverterschaltung Bild A.2

Ist die Abfallzeit am Ausgang deutlich größer als die Anstiegszeit am Eingang, dann ist die Lastkapazität C_L sehr groß. In diesem Fall wird der Eingang durch die „transient region“ gehen, bevor der Ausgang sich zu ändern beginnt. Da die Source-Drain-Spannung des PMOS-Transistors notwendigerweise während dieser Zeit gleich 0 ist, schaltet dieser Transistor aus, ohne einen Strom zu liefern. Der Kurzschlussstrom ist somit gleich null.

Wird C_L hingegen sehr klein gewählt und die Abfallzeit am Ausgang ist kleiner als die Anstiegszeit am Eingang, so ist die Drain-Source-Spannung am PMOS-Transistor während des Übergangs durch die DC-Transfer Charakteristik bestimmt, was einen höheren Kurzschlussstrom zur Folge hat [14].

Anhang B
Leistungsabschätzung (Power Estimation)

Um ein optimales Schaltungsdesign zu erhalten, muss man Möglichkeiten haben, den Leistungsverbrauch einer CMOS-Schaltung schon in den höheren Ebenen der Entwurfshierarchie vorhersagen zu können. Nur dadurch wird es möglich, Entscheidungen zwischen verschiedenen Optionen zu treffen, um ein aufwendiges Re-Design zu vermeiden.

Es werden also Methoden benötigt, mit denen es möglich ist, die Faktoren zu berechnen beziehungsweise abzuschätzen, welche hauptsächlich an der Entstehung der Verluste beteiligt sind. Zumeist beschränkt man sich darauf, P_S festzulegen.

B.1 Abschätzungen auf der Architekturebene

In folgenden Abschnitten werden Methoden für die Abschätzung des Leistungsbedarfs eines Schaltungssystems auf der Architekturebene vorgestellt.

B.1.1 Activity-Based-Control (ABC)-Modell

Im Kontroll-Pfad wird zur Vorhersage des Leistungsverbrauchs häufig das Activity-Based-Control Modell, kurz ABC-Modell, benutzt. Im Gegensatz zu den Datenwörtern im Datenpfad, die eine definierte Struktur besitzen, sind die Signalwörter im Kontrollpfad eine unabhängige Aneinanderreihung von Datenfeldern und Steuersignalen. Aus diesem Grund kann a priori kein Modell für die Datenaktivität des Kontrollpfades entwickelt werden. Deshalb benutzt man beim ABC-Modell konventionelle Aktivitätsparameter, wie die Schaltwahrscheinlichkeit und die Signalwahrscheinlichkeit [14]. Diese werden mit Größen wie der Anzahl der Eingänge (N_I), Anzahl der Ausgänge (N_O) und Anzahl der Min-Terme (N_M) einer Finite State Machine entsprechend kombiniert. Somit kann man Leistungsmodelle für eine Vielzahl von Controllerrealisierungen erstellen [46].

Die gesamte Kapazität (C_{Total}) eines Controllers und somit sein Leistungsverbrauch kann beispielsweise folgendermaßen berechnet werden:

$$C_{Total} = C_I \alpha_I N_I N_M + C_O \alpha_O N_O N_M \qquad A.14$$

$$P = C_{Total} V_{dd}^2 f \qquad A.15$$

α_I und α_O sind die Schaltwahrscheinlichkeiten am Eingangs- und Ausgangsbus des Controllers, C_I und C_O sind empirisch ermittelte Kapazitätskoeffizienten [77][56][46].

In [77] wurden Verbrauchsabschätzungen für Controller, deren Implementierungen auf ROM- oder PLA-Techniken basieren, mittels des ABC-Modells durchgeführt. Dabei wurde eine Abweichung von ca. 10% von den Ergebnissen der Switch-Level-Simulation (IRSIM) festgestellt.

B.1.2 Dual-Bit-Type (DBT)-Modell

Das DBT-Modell [47] geht von einer Korrelation der Eingangsdaten aus und ist somit für Modelle geeignet, die Daten im Zweierkomplement darstellen. Auf Grund dieser Korrelation werden die Bits der Eingangsvektoren in UWN-Bits (Universal White Noise) und Vorzeichen Bits gruppiert. Für die UWN Bits gilt eine Schaltwahrscheinlichkeit P(0→1)=0,25. Ein Wechsel der Vorzeichen Bits ist unwahrscheinlicher, weswegen sie eine deutlich geringere Schaltwahrscheinlichkeit zugewiesen bekommen (Bild A.4). Anhand der Schaltwahrscheinlichkeiten werden die effektiven Kapazitäten für die Bitgruppen ermittelt [56] und darauf aufbauend die Verlustleistung.

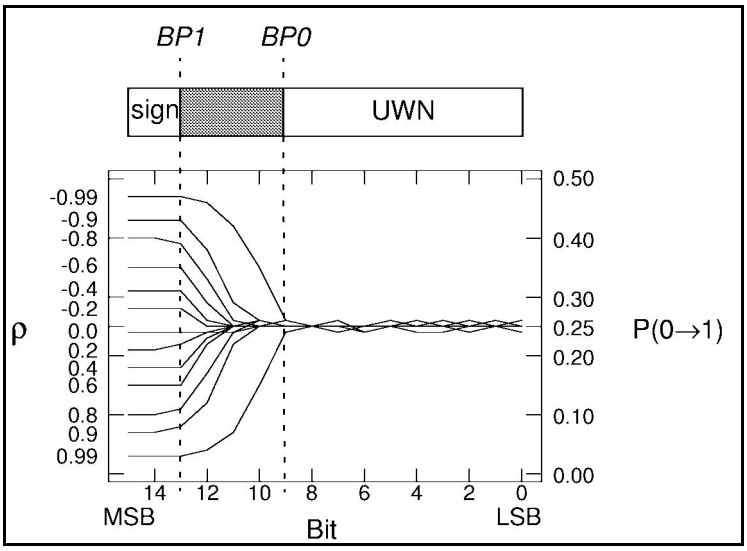


Bild A.4: Darstellung der Schaltwahrscheinlichkeiten der einzelnen Bitpositionen bei unterschiedlich korrelierten Datensignalen

Bild A.5 zeigt die Genauigkeit der Vorhersagen der Verlustleistung beim Verwenden des reinen UWN Modells [84] und der DBT Modells im Vergleich zu einer Low Level Simulation. Das DBT Modell liefert genauere Ergebnisse und ist damit geeigneter.

Bei dem verwendeten Beispiel (Bild A.5) handelt es sich um einen 16x16 Multiplizierer. Hier liegen nur zwei korrelierte Eingangsvektoren vor. Für mehrere unterschiedlich korrelierte Eingänge müssen zusätzliche Überlegungen gemacht werden (siehe [47]). Das DBT Modell ist allerdings auf Daten in Zweierkomplement-Darstellung beschränkt.

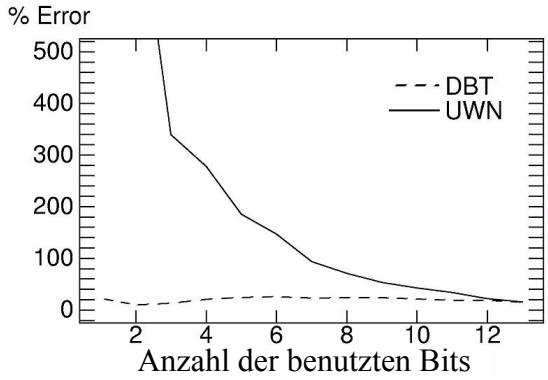


Bild A.5: Fehlerquote bei der Verwendung von UWN- beziehungsweise DBT-Modellierung

B.2 Abschätzung auf der Algorithmenebene:

Ausgehend von der Gleichung $P_{Switching} = \frac{1}{2} C_{total} V^2 f_{Clock}$ (vergleiche Gleichung *A.6* und Gleichung *A.7*), mit f_{Clock} als Taktfrequenz und C_{total} als durchschnittliche Kapazität pro Sample Periode, wird auf dieser Ebene die Verlustleistung durch Bestimmung von C_{total} ermittelt.

Die durchschnittlich geschaltete Kapazität C_{total} setzt sich folgendermaßen zusammen:

$$C_{total} = C_{exu} + C_{register} + C_{interconnect} + C_{control} \qquad A.16$$

C_{exu}, $C_{register}$, $C_{interconnect}$ und $C_{control}$ sind die durchschnittlichen Schaltkapazitäten der entsprechenden Einheiten: Execution Units (C_{exu}), Register ($C_{register}$), Interconnect ($C_{interconnect}$) und Controller ($C_{control}$). Modelle zur Bestimmung dieser Kapazitäten werden in den folgenden Absätzen vorgestellt [14].

B.2.1 Ausführungseinheiten (Execution Units)

Die Schaltkapazität (C_{exu}) der unterschiedlichen Ausführungseinheiten wird abgeschätzt, indem die Anzahl der Zugriffe (N_i) der Module auf die verschiedenen Operationsarten (numtypes) pro Periode mit der durchschnittlichen Kapazität (C_i) der Operationen multipliziert wird und alle Ergebnisse des Moduls aufsummiert werden:

$$C_{exu} = \sum_{i=1}^{numtypes} N_i \cdot C_i \qquad A.17$$

Die typischen Werte C_i für die unterschiedlichen Module werden mittels zum Beispiel SPICE oder IRSIM bestimmt, indem ein gleichverteiltes Eingangssignal auf das Modul gegeben wird. Im Allgemeinen sind die Eingangssignale jedoch nicht gleichverteilt, wie bereits in

Anhang B.1 festgestellt wurde. Diese Annahme wird aber getroffen, um den Rechenaufwand der Schätzverfahren zu reduzieren.

Laut [61] weist dieses Modell einen durchschnittlichen Fehler von 16.7% auf.

B.2.2 Speicher

Bei der Bestimmung der Kapazität der Register wird prinzipiell genauso verfahren wie bei den Ausführungseinheiten. Der Unterschied besteht nur darin, dass die Anzahl der physikalisch vorhandenen Register nicht benötigt wird, sondern nur die durchschnittliche Schaltkapazität der Register $C_{registers}$ und die Anzahl der Zugriffe $N_{Register}$ innerhalb einer Periode [14].

Der durchschnittliche Fehler dieses Modells liegt nach [61] bei 12%.

B.2.3 Verbindungen

Die Interconnect-Schaltkapazität hängt wesentlich vom endgültigen Chip-Layout (Bus-Länge) ab und ist somit nur sehr schwer vorhersagbar, da sowohl die physikalische Buskapazität als auch die Anzahl der Buszugriffe auf der Algorithmus-Ebene noch nicht bekannt sind. Statistische Vorhersagemethoden sind notwendig, um die Interconnect-Schaltkapazität zu bestimmen.

Die Schaltkapazität $C_{interconnect}$ wird folgendermaßen bestimmt:

$$C_{interconnect} = \alpha \cdot \frac{C_{Bus}}{N} \qquad A.18$$

α ist die durchschnittliche Aktivität (Produkt aus der totalen Anzahl der Interconnect Zugriffe und der Schaltwahrscheinlichkeit), C_{Bus} die physikalische Kapazität einer durchschnittlich langen Leitung und N ist ein geschätzter Faktor, der physikalische Verbindungen der Busse untereinander berücksichtigt [14][56].

Modelle, die auf dem oben erläuterten Prinzip beruhen, weisen einen durchschnittlichen Fehler von ca. 15% auf [61].

B.2.4 Kontroll-Logik

Die Abschätzung der benötigten Leistung der Kontroll-Einheit ist, wie auch beim Interconnect, problematisch, da in dieser Ebene noch keine definitiven Informationen über das Endlayout vorliegen. Wie beim Interconnect, sind statistische Vorhersagemethoden notwendig, um den Energieverbrauch, und somit auch die Verlustleistung dieser Einheit bestimmen zu können.

Man unterscheidet zwischen den globalen Controllern und den lokalen Controllern.

Für **globale Controller** gilt nach [14][17][61] die Näherung:

$$C_{FSM} = \alpha_1 \cdot N_{states} + \alpha_2 \qquad A.19$$

N_{states} ist dabei die Anzahl der Zustände und α_1, α_2 sind experimentell gefundene Faktoren.

Für **lokale Controller** gilt nach [14][61] entsprechend:

$$C_{lc} = \beta_0 + \beta_1 N_{trans} + \beta_2 N_{states} + \beta_3 B_f \qquad A.20$$

C_{lc} ist die effektive Schaltkapazität eines lokalen Controllers, N_{trans} ist die Anzahl der Transitionen (aus einem statistischem Modell ermittelt), N_{states} ist die Anzahl der Zustände im Controller. β_0, β_1, β_2, β_3 stellen Korrekturfaktoren dar. Für eine 1.2µm Technologie sind folgende Faktoren ermittelt worden: β_0= 72fF, β_1=0,15fF, β_2=8,3fF, β_3=0,55fF [14]. B_f ist der Busfaktor, der durch den Quotienten aus Zugriffen auf den Bus (B_Z) und Anzahl der Busse (B_A) definiert wird.

Der durchschnittliche Fehler dieses Prinzips zur Berechnung der Schaltkapazität von Controllern liegt nach [61] bei 11.52%.

B.3 Leistungsabschätzung auf Gatter-Ebene

Wie auf den anderen Abstraktionsebenen geht man hier ebenfalls von der Gleichung *A.7* für P_S aus. Die Gleichung wird aber angepasst:

$$P_{Switching} = \frac{V_{dd}^2}{2T_{Zyklus}} \cdot \sum_{g=1}^{k} C_g E_{sw}(g) \qquad A.21$$

Dabei sind hier C_g die physikalische Kapazität, die durch das Grundgatter geschaltet wird und k ist die Anzahl der Gatter. $E_{sw}(g)$ ist vergleichbar mit der Schaltwahrscheinlichkeit α, mit dem Unterschied, dass $E_{sw}(g)$ sowohl $0\rightarrow1$, wie auch $1\rightarrow0$ Übergänge am Gatterausgang berücksichtigt, wodurch der Faktor 1/2 in Gleichung *A.21* begründet wird [56][77].

Zur Vorhersage des Leistungsverbrauchs werden unterschiedliche Techniken benutzt. Man kann diese prinzipiell in die zwei Kategorien Simulationsgestützte Methoden und Stochastische Methoden aufteilen.

B.3.1 Simulationsgestützte Methoden

Das Verhalten einer Schaltung wird anhand von repräsentativen Testmustern auf Schaltungsebene simuliert. Das Festlegen der Testmuster beeinflusst wesentlich die Genauigkeit dieser Methode. Entsprechende Testmuster vorausgesetzt, ist diese Analyse sehr genau und kann

für die unterschiedlichsten Schaltungsmodelle eingesetzt werden. Allerdings kann man diese Simulatoren nur für relativ kleine Entwürfe benutzen, da sie sehr speicher- und rechenzeitaufwendig sind. Verwendete Simulationsprogramme sind zum Beispiel SPICE, PowerMill und Entice-Aspen [56][77][84]. Die Ungenauigkeit der erzielten Ergebnisse werden zum Beispiel für PowerMill mit 10% angegeben [56].

B.3.2 Stochastische Methoden

Unter der Annahme, dass V_{dd} und T_{Zyklus} konstant sind, kann man eine Aussage über den Energieverbrauch, bedingt durch die Schaltaktivität der Gatter, nur dann treffen, wenn die physikalische Kapazität C_n und die Schaltwahrscheinlichkeit $E_{sw}(n)$ eines Knotens n bekannt sind. C_n wird wiederum durch Extraktion bestimmt. $E_{sw}(n)$ können mittels der Signalwahrscheinlichkeit prob(n) des Knotens n realisiert werden. Bei diesen Berechnungen wird im einfachsten Fall ein Zero-Delay-Modell zugrundegelegt und es wird angenommen, dass die Eingangssignale unabhängig voneinander sind:

$$E_{sw}(n) = 2prob(n)(1 - prob(n)) \qquad A.22$$

prob(n) gibt die Wahrscheinlichkeit an, dass der Knoten n auf „1“ liegt.

Liegen die Signalwahrscheinlichkeiten der Eingänge fest, können mit Algorithmen wie dem *Tree-Algorithm* [77] alle weiteren Signalwahrscheinlichkeiten bestimmt werden.

Aufgrund des einfachen und unspezifischen Aufbaus dieser Methode ist sie für Worst-Case-Betrachtungen geeignet.

Anhang C
Low Power Entwurfsmethoden

Dieser Anhang stammt aus den Anfängen der Arbeit, als der Low Power Aspekt eine zentrale Rolle spielte. Sowohl auf der Architektur- wie auf der Algorithmenebene gibt es eine Vielzahl von Möglichkeiten, den Leistungsverbrauch eines Systems zu optimieren.

C.1 Verbesserung des Datendurchsatzes für Voltage Schaltungen

Die Herabsetzung der Versorgungsspannung sorgt dafür, dass der Datendurchsatz einer Schaltung geringer wird. Um diesen Datendurchsatz anzuheben gibt es folgende Möglichkeiten.

C.1.1 Veränderungen auf der Architekturebene

Um die Durchsatzverluste einer CMOS-Schaltung durch die Spannungsreduktion zu mindern beziehungsweise ganz zu kompensieren, werden auf der Architekturebene Parallel- und Pipelinestrukturen verwendet. Zur Demonstration der Arbeitsweise dieser zwei Techniken wird ein Addierer-Komparatorpfad betrachtet. Dieser addiert die Eingangssignale A und B und vergleicht das Ergebnis mit dem Signal C (Bild A.6).

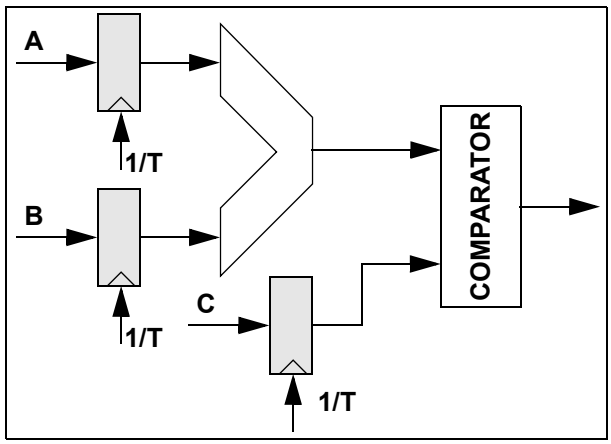


Bild A.6: Addierer-Komparator-Schaltung

Die Worst-Case Verzögerung, bedingt durch Flipflops, Addierer und Komparator, beträgt 25ns, bei einer Versorgungsspannung von $V_{ref} = 5V$. Somit ist die bestmögliche Takt-Periodendauer $T_{ref} = 25ns$. Mit diesen Werten ergibt sich die Referenz-Leistung $P_{Ref} = C_{Ref} V_{Ref}^2 f_{Ref}$.

Die totale effektive Kapazität C_{Ref} wurde bestimmt, indem über einen gewissen Zeitraum gleichverteilte Signale auf die Eingänge gegeben und dabei die durchschnittliche Leistung gemessen wurden.

Dieser Datenpfad wird in den folgenden Untersuchungen als Referenz-Datenpfad benutzt, um Verbesserungen beziehungsweise Verschlechterungen des Leistungsverbrauches durch Veränderungen der Architekturstruktur bewerten zu können. Es soll ein maximaler Datendurchsatz stattfinden, d.h. der Durchsatzverlust durch eine Spannungsreduktion muss durch die entsprechende Architekturveränderung behoben werden [56][14].

C.1.1.1 Parallel-Architektur

Durch den Einsatz einer parallelen Architekturstruktur kann der Durchsatzverlust, der durch eine Reduktion der Versorgungsspannung entsteht, ausgeglichen werden. Wie aus Bild A.7 ersichtlich, besteht diese Architektur aus zwei identischen Addierer-Komparatorpfaden, deren Elemente mit der halben Referenzfrequenz arbeiten. Der bei diesem Aufbau notwendige Multiplexer wird mit dem Referenztakt betrieben [56][14].

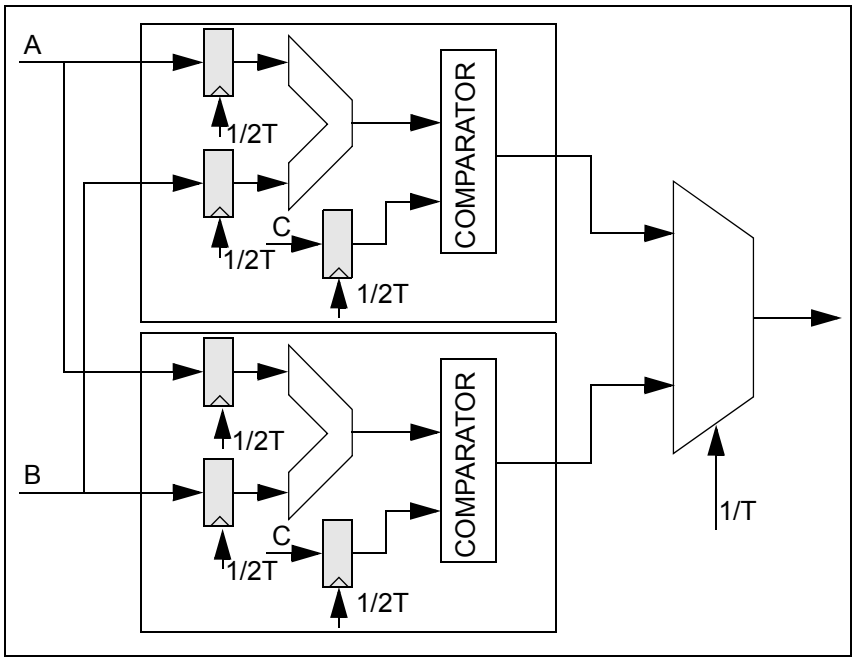


Bild A.7: Parallel-Architektur für einen Addierer-Komparator-Datenpfad

Die Worst-Case Verzögerung ist von 25ns auf 50ns angestiegen. Die minimal mögliche Taktperiodendauer ist somit $T_{Par}=2T_{Ref}=50ns$ [56]. Mittels dem V_{dd}-Delay Diagramm (Bild A.8) kann nun die mögliche Spannungsabsenkung bestimmt werden:

T_d ist die normierte Verzögerungszeit, V_{dd} ist die Betriebsspannung. Man kann ablesen: $V_{Par}=0,58*V_{Ref}=2.9V$. Die totale effektive Kapazität ergibt sich aus: $C_{Par}=2,15C_{Ref}$. Der Faktor 2,15 anstatt der erwarteten 2 kommt hauptsächlich durch die zusätzlichen Verbindungsleitungen zustande. Mit einer Taktfrequenz von $f_{Par}=0,5f_{Ref}$ kann der entsprechende Leistungsverbrauch bestimmt werden:

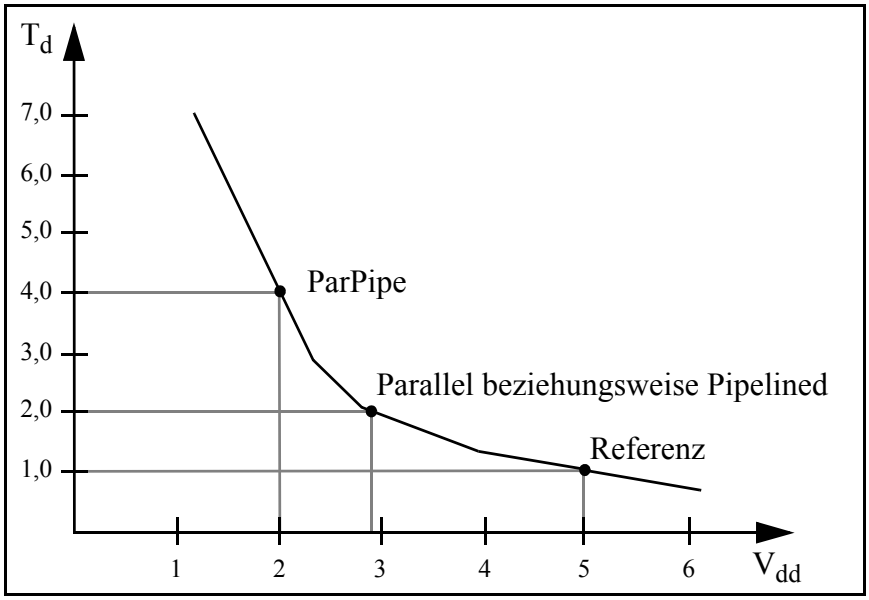


Bild A.8: V_{dd}-Delay-Diagramm

$$P_{Par} = C_{Par} V_{Par}^2 f_{Par} = (2{,}15\, C_{Ref})(0{,}58\, V_{Ref})^2 \left(\frac{f_{Ref}}{2}\right) \approx 0{,}36\, P_{Ref} \qquad A.23$$

Um die Versorgungsspannung noch weiter absenken zu können und somit auch den Leistungsverbrauch zu reduzieren, können weitere Parallel-Pfade hinzugefügt werden.

Die Grenze der Parallelstruktur ist aber dann erreicht, wenn die Versorgungsspannung in die unmittelbare Nähe der Threshold Spannung V_{Th} der CMOS-Transistoren kommt. Dann steigt die Verzögerungszeit rapide an.

C.1.1.2 Pipeline-Architektur

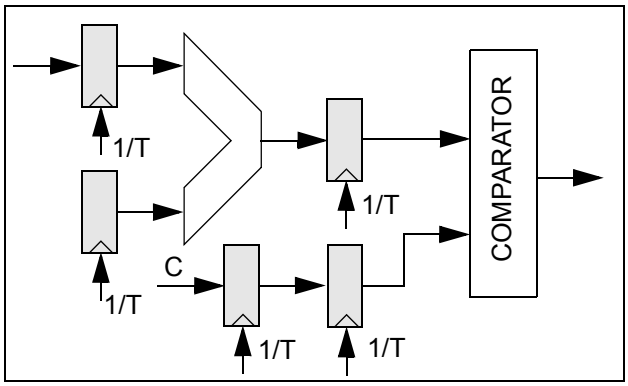


Bild A.9: Pipeline-Architektur

Bei der Pipeline-Architektur werden zusätzliche Flipflops in den Datenpfad eingesetzt. Mit diesen zusätzlichen Speichern wird die Länge des kritischen Pfades durch die maximale Verzögerung T_{max}=max(T_{adder}, $T_{comparator}$) bestimmt. Für dieses Beispiel wird angenommen, dass

T_{adder} und $T_{comparator}$ identisch sind. Werden die Flipflops mit dem Referenztakt von T_{Ref}=25ns betrieben, so stehen sowohl dem Addierer wie auch dem Komparator eine ganze Taktperiode zur Berechnung zur Verfügung. Die Verzögerungszeit kann somit doppelt so groß sein, ohne dass Durchsatzverluste entstehen, sofern die größere Latenzzeit akzeptiert wird. Wie man aus dem V_{dd}-Delay-Diagramm (Bild A.8) ersehen kann, ist für diese Verzögerungszeit $V_{Pipe} = 0,58 \cdot V_{Ref} = 2,9V$. Die totale effektive Kapazität beträgt: $C_{Pipe} = 1,15C_{Ref}$. Sie ist deutlich geringer als bei der Parallel-Architektur, da die Pipeline-Struktur weit weniger Fläche benötigt. Mit der gleichen Taktfrequenz $f_{Pipe}=f_{Ref}$ ergibt sich folgender Leistungsverbrauch:

$$P_{Pipe} = C_{Pipe}V_{Pipe}^2 f_{Pipe} = (1,15C_{Ref})(0,58V_{Ref})^2 f_{Ref} \approx 0,39P_{Ref} \qquad A.24$$

Die Pipeline-Struktur reduziert die Logiktiefe in einem Schaltkreis, wodurch die Verlustleistung durch Hazards verringert wird [56][14].

C.1.1.3 Kombination aus Pipeline- und Parallel-Architektur

Eine weitere Möglichkeit, den Durchsatzverlust durch eine Spannungsreduktion zu beheben, ist die Kombination der Pipeline- und Parallel-Architektur (Bild A.10).

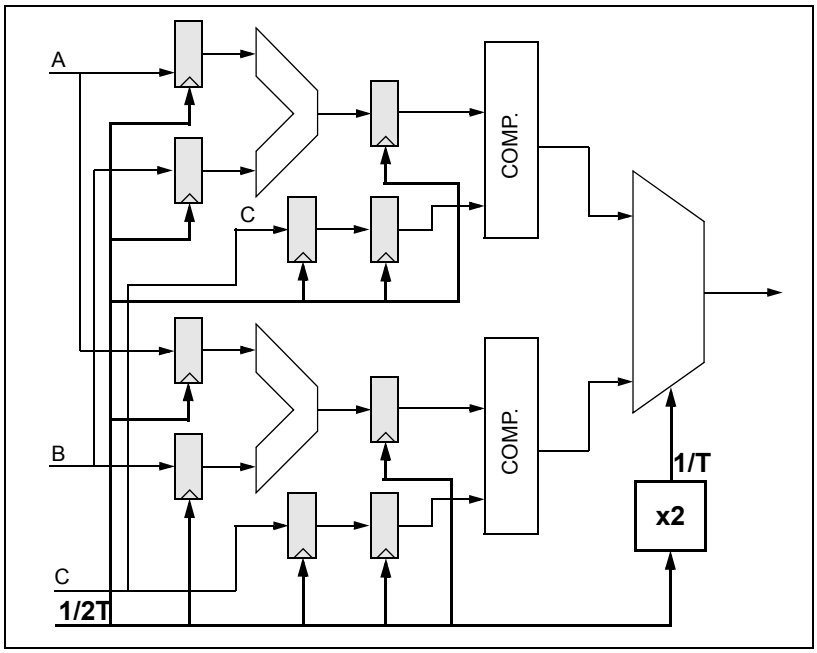


Bild A.10: Pipeline/Parallel-Architektur

Der kritische Pfad reduziert sich dadurch um den Faktor 4, die minimale Taktperiode ist: $T_{ParPipe} = 2T_{Ref} = 50ns$. Aus dem V_{dd}-Delay-Diagramm kann die mögliche Versorgungsspannung entnommen werden: $V_{ParPipe} = 0,4V_{Ref} = 2,0V$. Die effektive Kapazität beträgt

$C_{ParPipe} = 2,5C_{Ref}$. Mit einer Taktfrequenz von $f_{ParPipe} = 0,5f_{Ref}$ ergibt sich folgender Leistungsverbrauch:

$$P_{ParPipe} = C_{ParPipe}V_{ParPipe}^2 f_{ParPipe} = \\ = (2,5C_{Ref})(0,4V_{Ref})^2\left(\frac{f_{Ref}}{2}\right) \approx 0,2P_{Ref} \qquad A.25$$

Durch die Kombination beider Architekturen kann nochmals eine Reduzierung des Leistungsverbrauches erzielt werden. Allerdings steigt bei dieser Methode der Platzbedarf auf das 3,7-fache der ursprünglichen Schaltung [56][14].

In Tabelle 13 sind alle Daten der vier Architekturen im Überblick dargestellt.

Tabelle 13: Vergleich der 4 Architekturen

Architektur	V_{dd}	C_{eff}	Fläche	Leistung
Referenz	5 V	1	1	1
Parallel	2,9 V	2,15	3,4	0,36
Pipeline	2,9 V	1,15	1,3	0,39
Parallel/Pipeline	2,0 V	2,50	3,7	0,2

C.1.2 Veränderungen auf der Algorithmenebene

Speed up. Die Speed-up Transformation wird sehr häufig zur Verringerung des Energieverbrauchs eines Systems in der CMOS-Technik eingesetzt [14]. Mit ihr sollen Durchsatzverluste eines Systems ausgeglichen werden, die durch eine Reduktion der Betriebsspannung, und somit des Leistungsverbrauchs, entstehen. In Anhang C.1.1 wurden für diesen Zweck Parallel- und Pipeline-Architekturen verwendet. Diese können bei einfachen Beispielen, wie der Addierer-Komparator -Schaltung, problemlos eingesetzt werden. Enthalten Systeme aber Rückführungen, so können Parallel- und Pipeline-Strukturen allein gar nicht beziehungsweise nur noch bis zu einer gewissen Grenze zur Durchsatzverbesserung eingesetzt werden.

Wird beispielsweise ein „First Order IIR-Filter“ betrachtet, wie in Bild A.11 Fig.a dargestellt, so besteht keine Möglichkeit, den Durchsatz des Filters mittels einer Pipelinestruktur oder Re-timing zu verbessern. Den Grund hierfür kann man aus der Systembeschreibung $Y_N=X_N+AY_{N-1}$ (Bild A.11 Fig.a) erkennen. Für eine Pipelinestruktur müsste man Y_{N-2} bilden können, was ohne eine Verfälschung der Funktionsweise des Filters nicht möglich ist

Wird diese einfache Struktur mit Loop Unrolling in die in Bild A.11 (Fig.b) abgebildete Darstellung transformiert, so können zwei Samples parallel abgearbeitet werden. Der kritische Pfad hat sich durch diese Transformation verdoppelt. Dadurch, dass nun aber zwei Samples gleichzeitig verarbeitet werden können, wird der kritische Pfad effektiv nicht verändert, ebenso

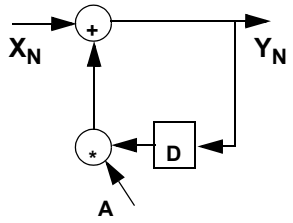
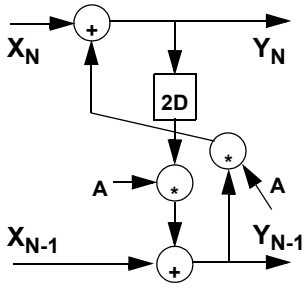
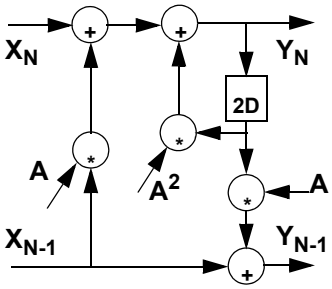
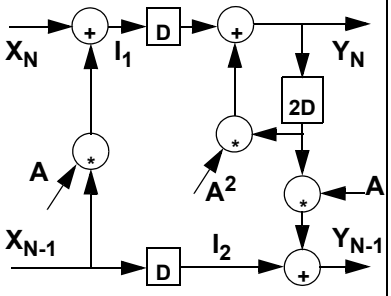
Fig.a $Y_n = X_n + A \cdot Y_{n-1}$		C_{eff}=1 Spannung=5V Durchsatz=1 Leistung=25
Fig.b $Y_{n-1} = X_{n-1} + AY_{n-2}$ ↓ $Y_n = X_n + A(X_{n-1} + AY_{n-2})$		C_{eff}=1 Spannung=5V Durchsatz=1 Leistung=25
Fig.c $Y_n = X_n + AX_{n-1} + A^2 Y_{n-2}$		C_{eff}=1,5 Spannung=3,7V Durchsatz=1 Leistung=20 (20% Leistungs-reduktion)
Fig.d $I_1 = X_n + AX_{n-1}$ $Y_n = I_1 + A^2 Y_{n-2}$ $I_2 = X_{n-2}$ $Y_{n-1} = I_2 + AY_{n-2}$		C_{eff}=1,5 Spannung=2,9V Durchsatz=1 Leistung=12,5 (50% Leistungs-reduktion)

Bild A.11: Leistungsreduktion mittels Speed-up

bleibt auch die effektive Kapazität gleich. Durch Loop Unrolling alleine kann also der Energieverbrauch des Filters nicht verringert werden, da weder die effektive Kapazität noch der kritische Pfad verringert werden, wodurch eine Leistungsreduktion ermöglicht würde [14].

Loop Unrolling ermöglicht aber, dass die neue Filterstruktur mittels weiterer Transformationen, wie Ausnutzung der Distributivität und Pipelining, weiter verändert werden kann, was zu einer deutlichen Leistungsverminderung der Schaltung führt. In Bild A.11 sind die entsprechend veränderten Graphen (Fig.b,c) abgebildet. Durch die Ausnutzung des Distributivgesetzes konnte Darstellung Fig.b in Fig.c transformiert werden. Dadurch konnte der effektive kritische

Pfad verkleinert werden, wodurch sich die Möglichkeit ergibt, die Versorgungsspannung auf V_{dd}=3,7V bei gleichbleibendem Datendurchsatz zu verringern. Da die effektive Kapazität ansteigt, ist lediglich eine Leistungsreduktion von 20% zu erreichen [14].

Man kann jedoch den effektiven kritischen Pfad der neuen Filterstruktur mittels Pipelining weiter reduzieren (Fig.d), wodurch eine Absenkung der Betriebsspannung auf V_{dd}=2,9V ermöglicht wird. Es kann somit eine gesamte Energieeinsparung von 50% erreicht werden [14].

C.2 Absenkung der Versorgungsspannung (Voltage Scaling)

Ausgehend von der Gleichung $P = C_{eff} V_{dd}^2 f_{Clock}$ kann man erkennen, dass eine Reduktion der Versorgungsspannung V_{dd} erheblich zur Verminderung der Verlustleistung beitragen kann, da V_{dd} quadratisch in diese Gleichung eingeht. Veränderungen der Versorgungsspannung beeinflussen aber die Geschwindigkeit der CMOS-Schaltung. Im Fall der Spannungsreduktion vergrößert sich die Verzögerungszeit T_d der Schaltung, wodurch es zu einer Verringerung des Datendurchsatzes kommt.

Dies kann primär, in gewissen Grenzen, mit einer Veränderung der Threshold-Spannung (Schwellspannung) V_{Th} verhindert werden [14]. Leider kann die Threshold-Spannung V_{Th} nicht beliebig klein angenommen werden. Die Einhaltung gewisser Störabstände und die Vergrößerung von Subthreshold-Strömen setzen der Skalierung von V_{Th} Grenzen.

Möchte man den Energieverbrauch einer Schaltung, unter Beibehaltung des Datendurchsatzes, mit einer Verringerung der Versorgungsspannung V_{dd} reduzieren, so muss man einen Kompromiss zwischen dem Energiegewinn durch kleinere Schaltungsverluste und dem Auftreten von Leckverlusten durch Subthreshold-Ströme finden. In Bild A.12 ist diese Problematik anhand eines Diagramms dargestellt.

Ist eine Veränderung der Schwellspannung nicht möglich, da beispielsweise gewisse Störabstände nicht unterschritten werden dürfen, so gibt es sowohl auf der Architektur- wie auch auf der Algorithmenebene Möglichkeiten, die durch eine Reduzierung der Versorgungsspannung verursachten Durchsatzveränderungen auszugleichen (Siehe Anhang).

C.3 Verringerung der effektiven Kapazität

Ist eine Skalierung der Betriebsspannung nicht möglich, so kann man aus Formel $P = \frac{1}{2} C_{eff} V_{dd}^2 f_{Clock}$ mit $C_{eff} = \frac{\alpha}{2} C_L$ (siehe Anhang A.1) ersehen, dass eine weitere Möglichkeit, die Verlustleistung eines Systems zu verringern darin besteht, die Schaltwahrscheinlichkeit α zu verkleinern und dadurch die effektive Kapazität C_{eff} zu reduzieren.

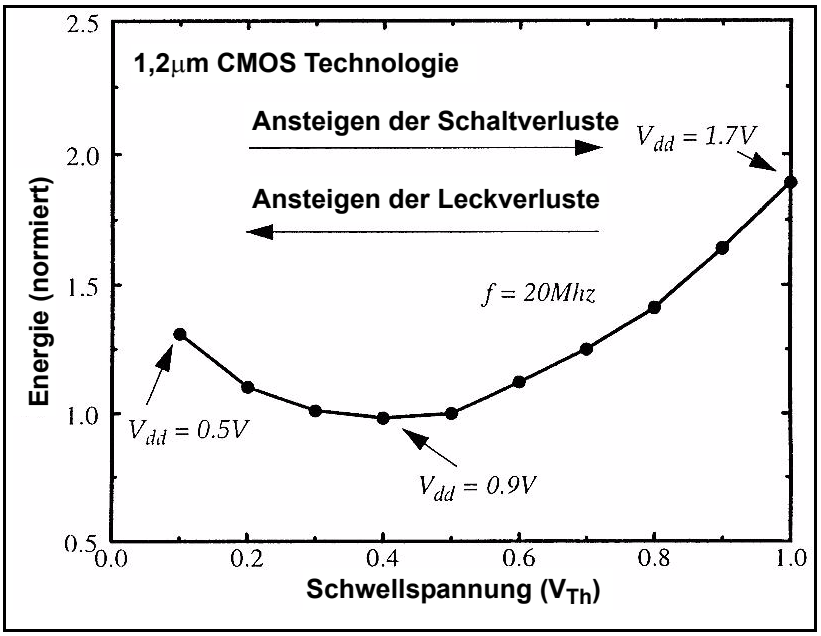


Bild A.12: Kompromiss zwischen Schalt- und Leckverluste durch die Variation der Schwellspannung V_{Th}

C.3.1 Optimierung auf der Algorithmenebene (Vektorquantisierung)

Auf der Algorithmenebene besteht die Möglichkeit, die Schaltaktivität einer Anordnung durch geeignete Wahl der Berechnungsverfahren zu reduzieren, was anhand von Vektorquantisierungsalgorithmen verdeutlicht werden soll. Vektorquantisierung (VQ) ist eine Kompressionsmethode, die beispielsweise in Videoübertragungssystemen eingesetzt wird. Das hier betrachtete 16 zu 1 Video Kompressionsverfahren arbeitet folgendermaßen:

Ein Bildausschnitt wird in eine 4*4-Pixel-Matrix unterteilt, wie in Bild A.13 dargestellt.

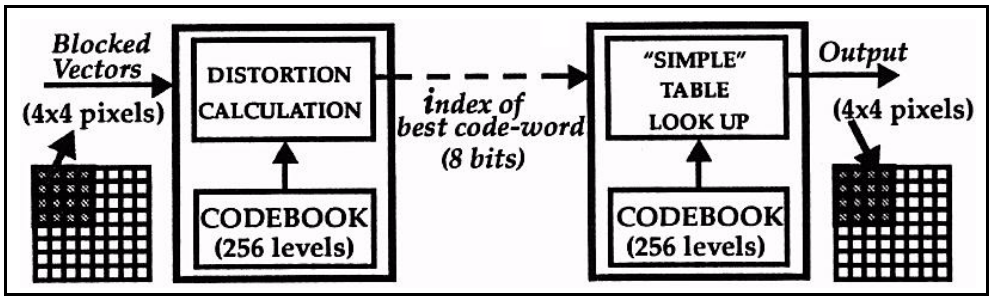


Bild A.13: Video-Kompression/Dekompression mittels Vektorquantisierung

Die Farbe jedes Pixels wird durch eine 8-Bit-Zahl repräsentiert. Die Pixel-Matrix kann somit durch einen 16-zeiligen Vektor beschrieben werden, der in jeder Zeile die 8-Bit Information für den entsprechenden Pixel enthält. Dieser Vektor wird mit vorher erstellten Code-Book-Vektoren verglichen. Diese Code-Book-Vektoren wurden a priori so erstellt, dass sie in möglichst vielen Zeilen den real vorkommenden Werten der Bildvektoren möglichst nahe kommen

können, um somit die Möglichkeit zu bekommen, mit einer gewissen Anzahl von Code-Vektoren alle möglichen Bild-Vektoren möglichst fehlerfrei darzustellen. In diesem Beispiel gibt es 2^8=256 Code-Book-Vektoren. Nachdem ein entsprechender Code-Book-Vektor nach ganz bestimmten Algorithmen einem realen Bildvektor zugeordnet wurde, wird nur die 8-Bit lange Identifikationsnummer des Code-Book-Vektors vom Sender an den Empfänger übertragen. Dort wird in dem ebenfalls vorhandenen Code-Book mittels der übertragenen Information der Code-Book-Vektor herausgesucht und zu der entsprechenden Bilddarstellung umgewandelt. Bei diesem Kompressionsverfahren wurde also aus sechzehn 8-Bit-Wörtern, ein 8-Bit-Wort generiert und übertragen, wodurch eine Datenreduktion von 16 zu 1 erreicht werden konnte.

Die Zuordnung eines Code-Book-Vektors zu einem Bildvektor kann nach unterschiedlichen Kriterien erfolgen. Nach [77] und [14] wird als Maß für die Übereinstimmung von Bildvektor und Code-Book-Vektor der mittlere quadratische Fehler der entsprechenden Vektoren benutzt, der allgemein nach Gleichung *A.26* berechnet wird.

$$D_i = \sum_{i=0}^{15} (X_j - C_{ij})^2 \qquad \text{für alle i=0..255} \qquad A.26$$

In dieser Darstellung sind X_j die Elemente des Bildvektors, C_{ij} entsprechen den Elementen der Code-Book-Vektoren.

Ausgehend von Formel 4.5 kann man unterschiedliche Zuordnungs-Algorithmen entwickeln. In [77] und [14] werden 3 Zuordnungsverfahren schematisch vorgestellt, welche durch ihre unterschiedliche Arbeitsweise die gesamte Schaltaktivität eines Schaltungssystems beeinflussen.

C.3.1.1 Full-Search Vektorquantisierung

Bei dieser sehr aufwendigen Methode wird die Übereinstimmung genau nach Gleichung A.26 bestimmt. Dies bedeutet, dass für jeden Code-Book-Vektor der mittlere quadratische Fehler bezüglich des Bildvektors berechnet wird. Der Code-Book-Vektor, der die geringste Abweichung vom Bildvektor aufweist, wird ausgewählt und seine entsprechende Identifikationsnummer übertragen. Für das obige Beispiel bedeutet dies, dass die Gleichung A.26 256 mal berechnet werden muss. Für jeden Bildvektor müssen daher 16 Subtraktionen, 16 Multiplikationen und 15 Additionen durchgeführt werden. Anschließend muss aus den 256 berechneten Fehler-Werten der minimale ausgewählt werden, wofür 255 Vergleichsoperationen notwendig sind [77][14].

C.3.1.2 Tree-Structured Vektorquantisierung

Um den enormen Rechenaufwand zu reduzieren, wie er bei der Full-Search VQ notwendig ist, wird üblicherweise eine Tree-Search-VQ-Methode verwendet. Die Grundidee ist, die Suche nach dem ähnlichsten Code-Book-Vektor mittels einer Baumstruktur durchzuführen.

Somit steigt der Rechenaufwand mit logN anstatt mit N, wobei N die Anzahl der Vektoren im Code-Book ist. Bei der Tree-Search-VQ muss somit, für das obige Beispiel, nur 2*log(256)=16 mal der Mittlere-Quadratische-Fehler berechnet werden anstatt 256 mal, und es werden anstatt 255 Vergleichsoperationen nur 8 benötigt, wodurch sich die Schaltaktivität und dadurch der Verbrauch des Systems drastisch reduziert [77][14].

C.3.1.3 Differential Tree-Structured Vektorquantisierung

Die Differential Tree-Structured Vektorquantisierung wird durch mathematische Umformungen der Tree-Structured Vektorquantisierung erstellt. Die Anzahl der Rechenoperationen kann dadurch nochmals reduziert werden (siehe Tabelle 3) [77][14].

Die Anzahl der Operationen, welche die unterschiedlichen Zuordnungsverfahren für das vorgestellte 16 zu 1 Video Kompressionsverfahren benötigen, ist in Tabelle 14 zusammenfassend dargestellt [14].

Tabelle 14: Anzahl der Operationen der verschiedenen Zuordnungsverfahren für eine 16 zu 1 Video Kompression

Algorithmus	Anzahl der Speicherzugriffe	Anzahl der Multiplikationen	Anzahl der Additionen	Anzahl der Subtraktionen
Full Search	4096	4096	3840	4096
Tree Search	256	256	240	264
Differential Tree Search	136	128	128	0

Aus diesem Beispiel kann man sehr gut erkennen, dass durch Optimierungen auf der Algorithmenebene der Rechenaufwand - und somit auch die Schaltaktivität - eines Schaltungssystems deutlich reduziert werden kann.

C.3.2 Wahl des Datenformats

Durch die Änderung des Datenformats ist es möglich, in bestimmten Fällen den Leistungsverbrauch von Schaltungssystemen zu reduzieren, was in den folgenden Kapiteln erläutert wird.

C.3.2.1 1-aus-n Kodierung:

Eine der vielen Kodierungsmöglichkeiten, die zur Verlustleistungsminimierung benutzt werden können, ist die 1-aus-n Kodierung (One-Hot-Coding). Diese wird gelegentlich bei der Zustandskodierung von endlichen Automaten eingesetzt. Es wird von der Grundidee ausgegangen, dass zur Übertragung einer n-Bit Dualzahl $m=2^n$ Signalleitungen benötigt werden (siehe Bild A.14). Jedem Wert der n-Bit langen Dualzahl wird eine Übertragungsleitung zugeordnet. Möchte man beispielsweise den Zahlenwert "2" einer 2-Bit-langen Dualzahl mit diesem Prinzip übertragen, so wird die dritte von vier Übertragungsleitungen auf HIGH und alle anderen Leitungen auf LOW gelegt, indem der Encoder aus der Dualzahl den entsprechenden Übertragungsvektor generiert. Der Empfänger- Decoder wandelt diesen Übertragungsvektor wieder in die entsprechende Dualzahl um.

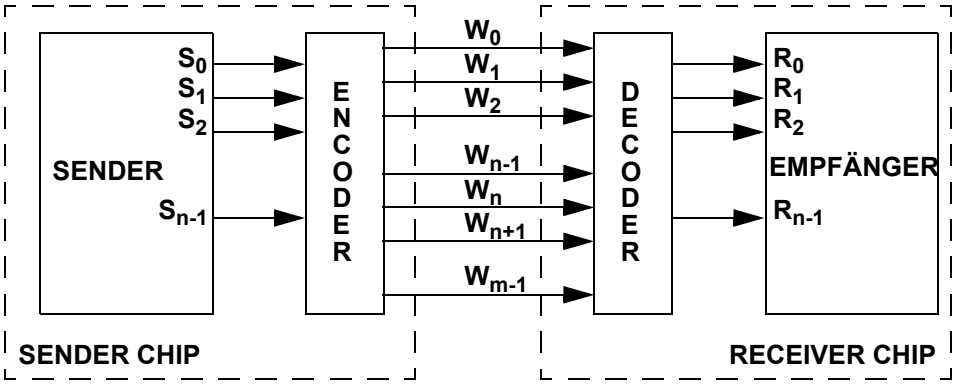


Bild A.14: Busmodell für One-Hot-Coding

Bei diesem One-Hot-Coding werden zwar $m=2^n$ Signalleitungen benötigt, man hat aber auch die Garantie, dass genau ein $0\rightarrow1$ und ein $1\rightarrow0$ Wechsel stattfindet. Unter der Annahme eines gleichverteilten Signals kann man das Verhältnis des Energieverbrauchs zur unkodierten Übertragung wie folgt berechnen:

$$\frac{P_{one-hot}}{P_{nocoding}} = \frac{\alpha_{one-hot} C V^2 f}{\alpha_{nocoding} C V^2 f} = \frac{\alpha_{one-hot}}{\alpha_{nocoding}} = \begin{cases} 1 & n = 1 \\ \dfrac{4(1-2^{-n})}{n} & n \geq 2 \end{cases} \qquad A.27$$

Obwohl mit der 1-aus-n Kodierung eine hohe Reduktion der Schaltaktivität erreicht werden kann, ist sie aufgrund der exponentiell mit n steigenden Anzahl der Übertragungsleitungen sehr unpraktisch. Bei einer Bitzahl von zum Beispiel n=8 erreicht man mit dieser Methode eine Energieeinsparung von 75%, man benötigt dafür aber auch 256 Signalleitungen [14].

C.3.2.2 Bus Inversions Kodierung

Zur Übertragung eines n-Bit langen Datenwortes werden m=n+1 Datenleitungen benötigt. Wird ein Datenwort D_i über die Datenleitungen übertragen, so kann bei dem darauffolgenden

Datenwort D_{i+1} entschieden werden, ob D_{i+1} oder das inverse Datenwort $\overline{D}_{i+1}$übertragen wird. Das Entscheidungskriterium ist die Anzahl der Signalwechsel auf den Datenleitungen, wenn das Datenwort D_i auf D_{i+1}beziehungsweise $\overline{D}_{i+1}$ wechselt. Sind bei D_i auf D_{i+1}weniger Signalwechsel vorhanden als bei D_i auf $\overline{D}_{i+1}$, dann wird D_{i+1}genommen, ansonsten $\overline{D}_{i+1}$. Um dem Empfänger nun mitteilen zu können, ob D_{i+1}oder $\overline{D}_{i+1}$ gesendet wurde, wird die zusätzliche Signalleitung benötigt. Ist diese Leitung zum Beispiel "1" so wird dem Empfänger mitgeteilt, dass das ankommende Datenwort D_{i+1} darstellt; bei "0" ist das ankommende Datensignal das inverse Datenwort $\overline{D}_{i+1}$, was der Empfänger entsprechend umwandeln muss.

In dem in Bild A.15 abgebildeten Diagramm ist der Quotient aus dem Energieverbrauch mit Kodierung und dem Energieverbrauch ohne Kodierung ($P_{kodiert}/P_{nicht\ kodiert}$) über die Anzahl der Datenbits aufgetragen. Diese Darstellung wurde für gleichverteilte Datensignale berechnet. Daraus kann man erkennen, dass bei n=2 die Energieeinsparung maximal 25% betragen kann. Sogar bei einem 32-Bit-Bus kann mit dem Mehraufwand von nur einer Signalleitung eine Verringerung des Verbrauchs von 11,3% erreicht werden. Wie aus Bild A.15 ersichtlich, arbeitet dieses einfache Kodierungsverfahren am besten bei möglichst kleinen Werten für n, d.h. für möglichst schmale Datenbusse. Für große Werte von n ist es also sinnvoll, den Datenbus in kleinere Teilbereiche zu zerlegen, auch wenn dies zusätzliche Verbindungsleitungen erfordert. Unterteilt man beispielsweise einen 32-Bit-Bus in 4 Gruppen zu je 8 Bits, werden zwar 4 zusätzliche Signalleitungen benötigt, man bekommt bei der Datenübertragung aber eine Energieeinsparung von 18,3% anstatt der oben erhaltenen 11,3% [14].

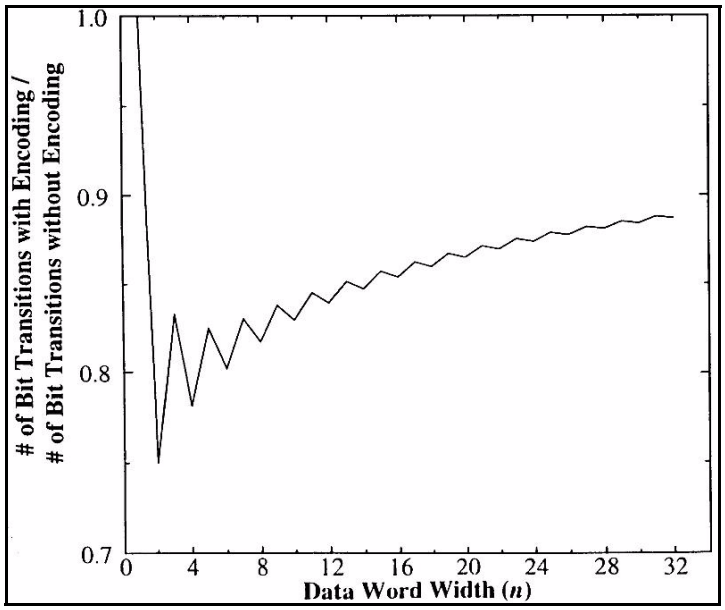


Bild A.15: Darstellung des Verhältnisses $P_{kodiert}/P_{nicht\ kodiert}$ für Datenwörter mit unterschiedlicher Bitbreite

C.3.2.3 Vorzeichen-Betrags-Kodierung

In vielen Signal- und Übertragungssystemen wird zur Dualzahldarstellung das Zweier-Komplement benutzt, da sich damit arithmetische Operationen wie zum Beispiel Addition und Subtraktion sehr einfach bewerkstelligen lassen. Ein großer Nachteil des Zweier-Komplements ist aber, dass bei Übergängen zwischen den positiven und den negativen Zahlenbereichen eine hohe Schaltaktivität der Bits im Datenwort entsteht. Beispielsweise wechseln beim Übergang von -1 auf 0 alle Bits des Datenwortes. Wird nun auf ein System mit Zweier-Komplementdarstellung ein Datensignal gelegt, dessen Datenworte hauptsächlich Werte um den Nullpunkt annehmen, kann eine erhebliche Schaltaktivität und somit Energieverbrauch im Übertragungssystem entstehen [14].

Eine Möglichkeit, die Schaltaktivität zu reduzieren ist der Einsatz der Vorzeichen-Betrags-Kodierung. Im Gegensatz zum Zweier-Komplement unterscheidet diese Darstellung den positiven und negativen Zahlenbereich nur in einem Vorzeichen-Bit. Dieses eine Bit repräsentiert das Vorzeichen des Datenwortes, wobei die anderen Bits den Zahlenwert angeben. Um die Vorzüge der Vorzeichen-Betrags-Kodierung zu verdeutlichen, werden zwei Beispiele betrachtet. Das erste zeigt die Schaltaktivität der Bits beim Wechsel von $(+1)_{10}$ nach $(-1)_{10}$ in der Zweier-Komplement-Darstellung und Vorzeichen-Betrags-Kodierung.

1. Fall: 8-Bit-Zahl in der Zweier-Komplement Darstellung
- $(+1)_{10}=(00000001)_2$
- $(-1)_{10}=(1111111)_2$

2. Fall: 8-Bit-Zahl in der Vorzeichen-Betrags-Kodierung
- $(+1)_{10}=(00000001)_2$
- $(-1)_{10}=(10000001)_2$

Im 1. Fall müssen wegen des Vorzeichenwechsels 7 Bits geschaltet werden, im 2. Fall hingegegen nur 1 Bit. Bei der Vorzeichen-Betrags-Realisierung sind also bei einem Vorzeichenwechsel deutlich weniger Schaltaktivitäten notwendig.

Im nächsten Beispiel [14] wird ein 16-Bit breiter Datenbus betrachtet: Auf diesen Datenbus wird ein Gaußverteiltes Datensignal gegeben, das den Mittelwert $\mu=0$ hat und die Varianz $3\sigma=2^{11}$ aufweist. In Bild A.16 ist die Schaltwahrscheinlichkeit der Bits, die in Abhängigkeit der Signal-Korrelation $\rho = (cov(X_n,X_{n+1}))/\sigma^2$ des Signals X angegeben wird, zur Bit-Position aufgetragen.

Für $\rho=0$ sind die Datenworte unkorreliert, d.h die Schaltwahrscheinlichkeit der MSBs ist 50%. Dies bedeutet, die Wahrscheinlichkeit, dass die auf den Bus geschickten Datenwörter ihr Vorzeichen wechseln, liegt bei 50%. Bei einer großen positiven Korrelation der Datenwörter ($\rho=0,99$) ändern die Datenwörter nur selten ihr Vorzeichen, wodurch eine geringere Schalt-

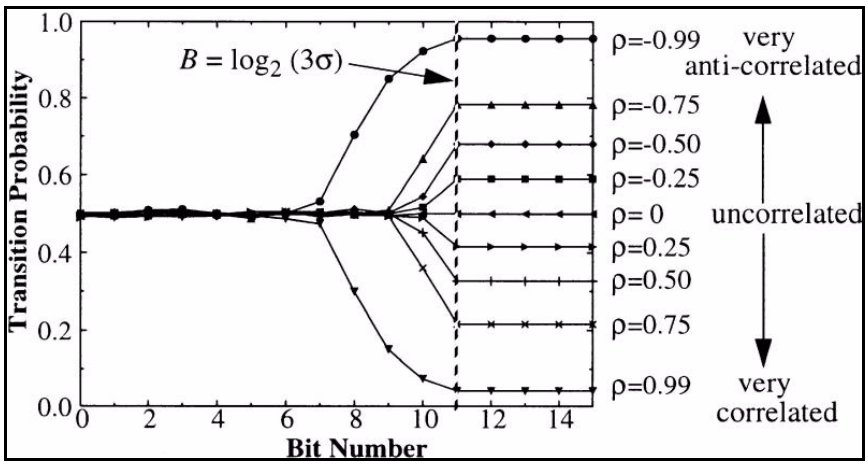


Bild A.16: Schaltwahrscheinlichkeiten der einzelnen Bitspositionen bei der Zweier-Komplement-Darstellung

wahrscheinlichkeit für die MSBs entsteht. Im Gegensatz dazu bedeutet eine große negative Korrelation der Datenwörter (ρ=-0,99) einen häufigen Wechsel des Vorzeichens und somit eine hohe Schaltwahrscheinlichkeit der MSBs. Der obere Knickpunkt, der den dynamischen Bereich des Datensignals darstellt, liegt in diesem Beispiel bei $\log_2(3\sigma) = 11$ Bits. Das bedeutet für die Zweier-Komplementdarstellung, dass die Bits 11 bis 15 die Schaltaktivität von Vorzeichen-Bits aufweisen. Vergleicht man dies mit der Vorzeichen-Betrags-Darstellung in Bild A.17, so kann man erkennen, dass Bit 11-14 eine geringere Schaltwahrscheinlichkeit aufweisen als beim Zweier-Komplement, die von Bit 15 hingegen gleich bleibt. Auch der Übergangsbereich (Bit 7-10) weist in der Vorzeichen-Betrags-Darstellung eine geringere Schaltaktivität auf, wodurch der Energieverbrauch des entsprechenden Übertragungssystems gegenüber der Zweier-Komplement-Darstellung reduziert wird.

Zusammenfassend kann man folgendes festhalten: Mit der Vorzeichen-Betrags-Darstellung kann man die Schaltaktivität in einem Bussystem reduzieren. Dies ist am effektivsten, wenn

a) der dynamische Bereich des Datensignals schmal und b) die verarbeiteten Daten nahezu unkorreliert sind. Das Zweier-Komplement hingegen bringt Vorteile bei arithmetischen Operationen, da nur einfache Algorithmen zur Berechnung notwendig sind. Die beste Lösung für Übertragungssysteme ist die Kombination beider Darstellungen in der Form, dass das Zweier-Komplement innerhalb der Arithmetikschaltungen benutzt wird und die Vorzeichen-Betrags-Darstellung bei den Bussystemen. Vorwiegend bei Systemen mit breitem Datenbus ist die Leistungseinsparung bei der Datenübertragung durch den Einsatz dieser kombinierten Technik recht hoch, da die Datenwörter meist nicht die ganze Busbreite ausnutzen. Der zusätzliche Energieverbrauch der Decoder/Encoder Hardware, welche für die Kombination der Darstellungen notwendig ist, kann bei solchen Systemen meistens toleriert werden [14][56].

Datendarstellung. Die Vorzeichen-Betrags-Darstellung kann die Schaltaktivitäten auf Datenbussen entscheidend reduzieren, hat aber das Problem, dass Operationen wie Addition und Sub-

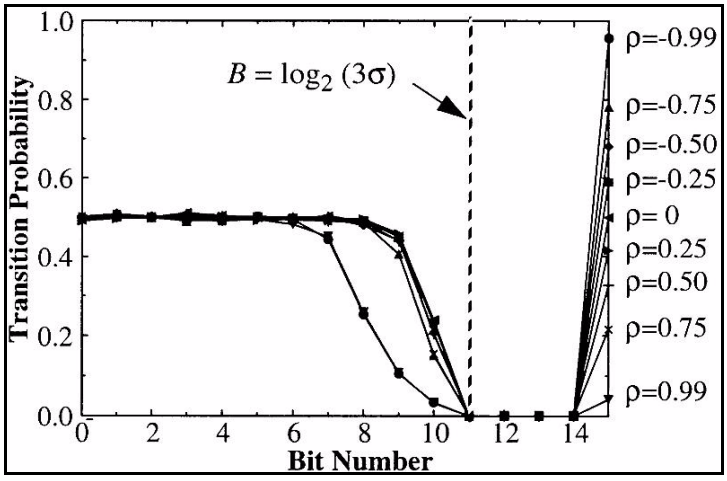


Bild A.17: Schaltwahrscheinlichkeiten der einzelnen Bitspositionen bei der Vorzeichen-Betrags-Darstellung

traktion schwieriger zu realisieren sind als beim Zweier-Komplement. Trotzdem kann in Systemen, bei denen mehrere Additionen in einer Periode durchgeführt werden müssen, die Vorzeichen-Betrags-Darstellung zur Schaltleistungsreduktion eingesetzt werden. Diese erfordert zwar durch die komplexere Architektur einen erhöhten Platzbedarf auf einem Chip, der Energieverbrauch kann aber bei gleichbleibendem Datendurchsatz bei bestimmten Datensignalen vermindert werden. Als Beispiel wird ein Akkumulator betrachtet (Bild A.18), der die Zweier-Komplement Darstellung benutzt [14].

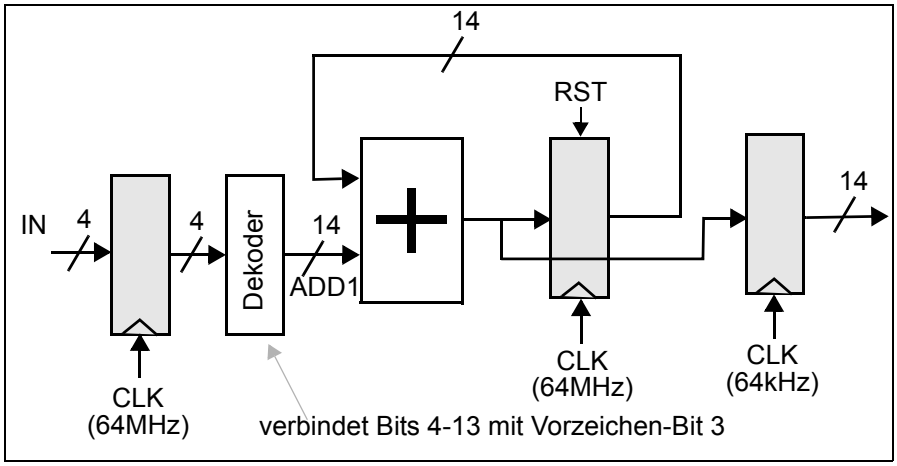


Bild A.18: Akkumulator in Zweier-Komplementdarstellung

Er akkumuliert 1024 4-Bit-Wörter mit 64MHz. Das Ergebnis wird zum Akkumulator-Register mit 64 kHz übertragen. Die Eingangsdaten haben einen Wertebereich von -7 bis +7. Bei der Verwendung der Zweier-Komplement Darstellung muss in diesem Aufbau das Vorzeichen-Bit des 4-Bit breiten Eingangsbusses, Bit 3 (wenn das Least Significant Bit (LSB) Bit 0 ist), dem MSB des 14 Bit breiten Signal ADD1 zugewiesen werden.

Wird ein gleichverteiltes Eingangssignal IN angenommen, so ist die Wahrscheinlichkeit einer positiven beziehungsweise negativen Eingangsgröße gleich groß. Die Schaltwahrscheinlichkeit der Bits 0-13 des Addiereingangs ADD1 ist somit gleich 50%. Da die unbenutzten Bits 4-13 des Eingangs ADD1 ebenfalls bei jedem Vorzeichenwechsel schalten müssen, entsteht eine hohe Schaltaktivität am Eingang des Addierers.

Eine andere Architektur eines Akkumulators ist in Bild A.19 zu sehen, der die Vorzeichen-Betrags-Darstellung benutzt [14]. Hierbei werden zwei Datenpfade verwendet. Einer der Pfade wird benutzt, um die positiven, der andere, um die negativen Zahlen aufzusummieren. Die entsprechende Zuweisung der Kanäle wird über das Vorzeichen-Bit des Eingangssignals gesteuert. Am Ende des 1024 Datenwort langen Zyklus werden die jeweils aufsummierten Signale an ein separates Register weitergeleitet, um anschließend in einem Subtrahierer-Glied entsprechend verrechnet zu werden.

Im Gegensatz zur Zweier-Komplement Methode weist diese Architektur insgesamt weniger Schaltaktivität auf. Der Grund hierfür ist, dass die Darstellung des Vorzeichens und somit der Vorzeichenwechsel mit weniger Schaltaktivität verbunden ist. Im Gegensatz zur Realisierung mit der Zweier-Komplement Darstellung tritt beim Vorzeichenwechsel keine unnötige Schaltaktivität am Addiereingang auf, da nur Bits 0 bis 2 schalten müssen um die Werte (0 bis 7) der Eingangswörter darzustellen. Die Bits 3-12 können den Wert „0“ beibehalten. Das Vorzeichen wird durch die Aufteilung in zwei Datenpfade berücksichtigt und verursacht bei dem eigentlichen Akkumulationsprozess keine zusätzlichen Schaltvorgänge.

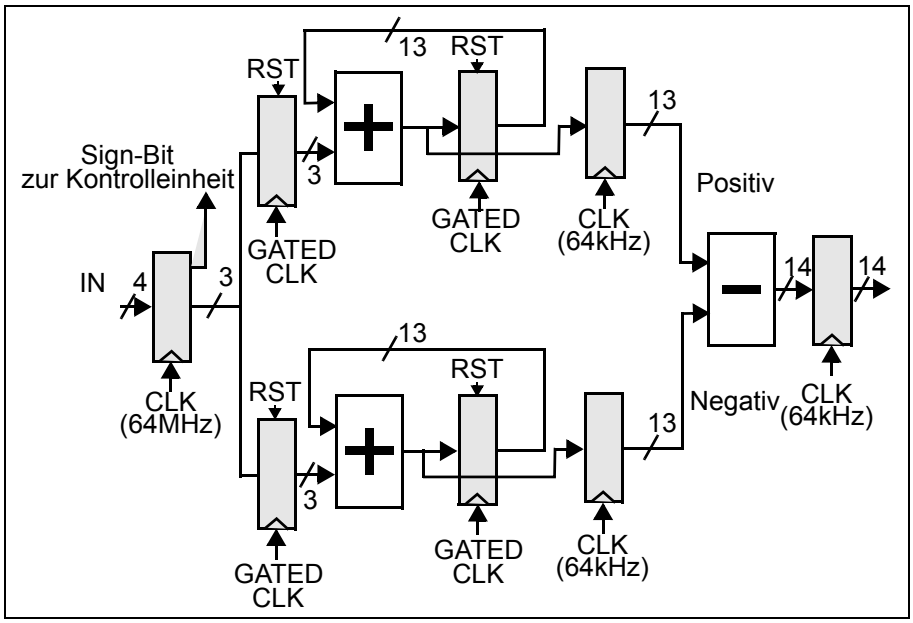


Bild A.19: Vorzeichen-Betrags-Darstellung des Akkumulators

Durch die getrennte Addition der positiven und negativen Datenwörter ist die Schaltaktivität des Akkumulators weitgehend unabhängig von der Beschaffenheit des Datensignals. Der größte

Leistungsgewinn gegenüber dem Zweier-Komplement wird erzielt, wenn das Datensignal eine hohe Varianz aufweist, wodurch Vorzeichenwechsel sehr häufig vorkommen. Liegt dagegen ein Signal mit wenig Vorzeichenwechsel vor, so ist der Leistungsbedarf des Vorzeichen-Betrags-Akkumulators, bedingt durch den höheren Schaltungsaufwand, größer. In Tabelle 4 ist der Energieverbrauch beim Einsatz der Zweier-Komplement-Realisierung und Vorzeichen-Betrags-Realisierung für unterschiedliche Eingangssignale abgebildet. Bei Testsignalen mit einer zufälligen Abfolge von Datenwörtern bringt die Verwendung der Vorzeichen-Betrags-Realisierung bei dem Akkumulatorsystem beispielsweise eine Energieeinsparung von 27% [14].

Tabelle 15: Energieverbrauch von Zweier-Komplement-Darstellung und Vorzeichen-Betrags-Darstellung

Eingangssignale (1024 Zyklen)	Zweierkomplement Leistungsverbrauch (mW), V_{dd}=3V	Vorzeichen-Betrags-Leistungsverbrauch (mW), V_{dd}=3V
Konstant (IN=7)	1,97	2,25
Rampe (-7,-6..6,7)	2,13	2,43
Zufall	3,42	2,51
Min→Max→Min	5,28	2,46

C.3.2.4 Gray-Code

Müssen über einen Datenbus Signale übertragen werden, die stark korreliert sind, so kann der Einsatz des Gray-Codes die Anzahl der Bit-Umschaltungen drastisch reduzieren. Wird beispielsweise ein Zähler betrachtet, der schrittweise hochzählt, so kann man bei der Übertragung die Schaltaktivität enorm vermindern, indem man den Gray-Code anstatt des normalen Binär-Codes benutzt wie aus Tabelle 16 ersichtlich ist. Der Vorteil des Gray-Codes und Einschritt-Codes allgemein ist, dass aufeinanderfolgende Zahlen sich nur in einem Bit unterscheiden. Aus Tabelle 5 kann man erkennen, dass bereits beim Hochzählen von 1 bis 8 der Gray-Code nur halb soviel Schaltaktivitäten aufweist wie der Binär-Code.

Tabelle 16: Anzahl der Schaltvorgänge beim Einsatz von Binär-Code und Gray-Code

Abfolge von Dezimalzahlen	Binär-Code	Gray-Code
1	0001	0001
2	0010	0011

Tabelle 16: Anzahl der Schaltvorgänge beim Einsatz von Binär-Code und Gray-Code

3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
Anzahl Schaltvorgänge	14	7

Bei entsprechenden Datensignalen können somit zwischen 24% und 58% der Schaltleistung eingespart werden [14].

C.3.3 Reduktion der Glitch-Aktivitäten

Aufgrund der Verzögerungszeiten von Gattern und der daraus resultierenden Verzögerungen der Signaländerungen, kann es in CMOS-Schaltungen zu unerwünschten, mehrmaligen Änderungen des Ausgangs während eines Übergangs kommen (Glitch). Um die Anzahl dieser zusätzlichen Übergänge zu minimieren, müssen alle Signalpfade ausbalanciert und die Logiktiefe reduziert werden.

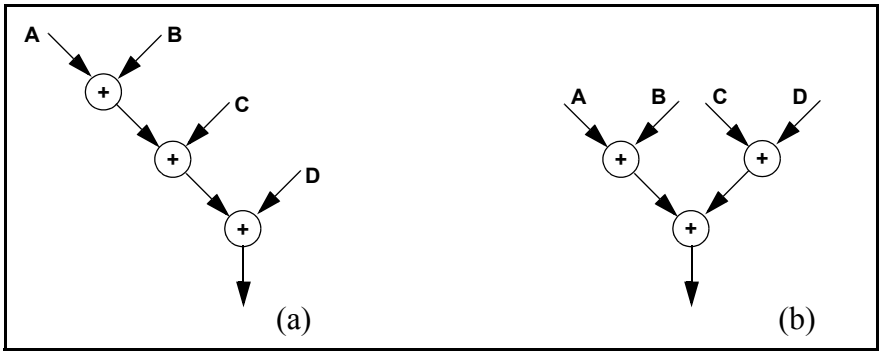


Bild A.20: Darstellung eines Datenpfades in Kettenschaltung (a) und Baumstruktur (b)

Angenommen, alle Eingänge der in Bild A.20 dargestellten Funktion ändern sich gleichzeitig, so ändert sich das Ergebnis der Kettenschaltung (Bild A.20(a)), bedingt durch die Verzögerungszeiten der Gatter, am Ausgang bis zu dreimal, bis der endgültige Wert anliegt. In der Baumanordnung (Bild A.20(b)) hingegen treten weniger bis keine Glitches auf, da hier die Signalpfade ausgeglichen sind. Wird angenommen, dass jede Addition einen Kontrollzyklus benötigt, so liegt das richtige Ergebnis bei der Baumstruktur schon nach dem zweiten Zyklus vor, bei der Kettenstruktur hingegen erst beim dritten Kontrollzyklus.

Man kann zusammenfassend sagen: Zunehmende Logiktiefe durch Kaskadierung lässt die Schaltkapazität wegen der höheren Glitch-Wahrscheinlichkeit ansteigen, während eine parallele Anordnung mit weniger Logiktiefe eine höhere Registerleistung ermöglicht. Es ist zu bemerken, dass allein durch die Reduktion der Logiktiefe Durchsatzverluste, bedingt zum Beispiel durch die Reduzierung der Versorgungsspannung, in gewissen Grenzen ausgeglichen werden können [14].

C.3.4 Reduktion von Operationen

Eine Methode, die Schaltkapazität zu reduzieren, ist die Verringerung der Anzahl von Operationen im CDFG (Control-Data-Flow Graph), die eine Reduktion der Schaltereignisse zur Folge hat. Durch die Umwandlung eines Polynoms zweiter Ordnung mit dem Horner-Schema kann beispielsweise die Anzahl der Multiplikationen von 2 auf 1 verkleinert werden (siehe Bild A.21)

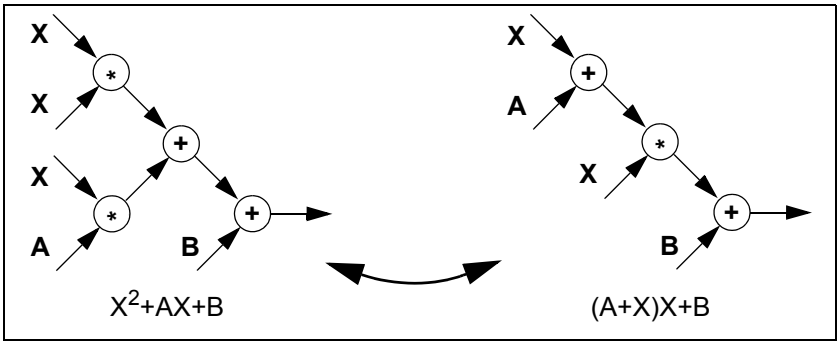


Bild A.21: Reduktion der Operationen (1)

Der transformierte CDFG hat in diesem Fall den gleichen kritischen Pfad wie der Ausgangsgraph und somit auch den gleichen Datendurchsatz bei gleicher Spannung, verbraucht aber weniger Energie als die Ausgangsanordnung.

Durch Umwandlung des CDFG eines Polynoms 3.Grades in Bild A.22 mit Hilfe des Horner-Schema konnte die Anzahl der Multiplikationen von 4 auf 2 gesenkt werden. Die Gesamtzahl der Operationen reduziert sich dadurch von 7 auf 5, wodurch sich auch die effektive Schaltkapazität vermindert. Der kritische Pfad ist jedoch in diesem Beispiel von 4 auf 5 gestiegen. Das bedeutet, man braucht eine höhere Betriebsspannung um den gleichen Datendurchsatz wie der Ausgangsgraph zu erhalten [14]. Die Erhöhung der Betriebsspannung hat allerdings ein Ansteigen der Verlustleistung zur Folge (siehe Kapitel 4.1). Soll der Datendurchsatz gleich bleiben, so muss die Reduktion der Operationen mehr Energie einsparen als mit der notwendigen Spannungserhöhung zusätzlich verbraucht wird, oder es muss eine andere Technik zur Erhöhung des Datendurchsatzes, wie zum Beispiel Pipelining, eingesetzt werden.

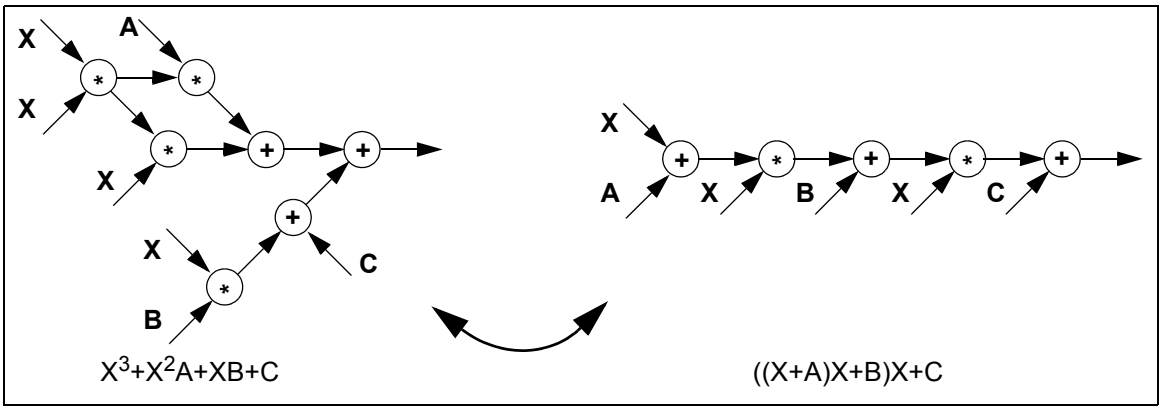


Bild A.22: Reduktion der Operationen (2)

C.3.5 Ersetzen von Operationen

In bestimmten Fällen ist es möglich, Operationen mit hohem Leistungsbedarf durch Operationen mit niedrigerem Leistungsbedarf zu ersetzen.

In Software-Compilern beispielsweise werden oft Multiplikationen durch Additionen ersetzt. In Bild A.23 wird diese Methode anhand einfacher CDFGs verdeutlicht.

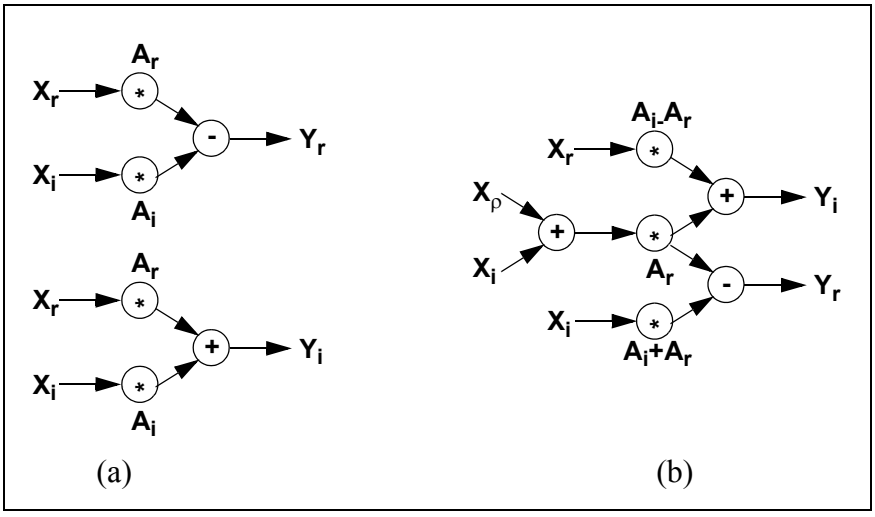


Bild A.23: Ersetzen einer Multiplikation durch eine Addition

Die in Bild A.23(a) abgebildete Darstellung $Y_r=X_rA_r-X_iA_i$ und $Y_i=X_rA_i+X_iA_r$ (A_r und A_i sind Konstanten) wird in $Y_r=(X_r+X_i)A_r-X_i(A_i+A_r)$ und $Y_i=(X_r+X_i)A_r+X_r(A_i-A_r)$ (Bild A.23(b)) transformiert, wobei durch entsprechende Umformung eine Multiplikation durch eine Addition ersetzt werden kann. Die Anordnung in Bild A.23(b) besitzt die gleiche Funktion wie Bild A.23(a), weist aber eine geringere effektive Kapazität auf. Allerdings hat sich bei dieser Transformation der kritische Pfad um einen Addierer verlängert. Um bei Bild A.23(b) den gleichen Datendurchsatz zu erreichen wie bei Bild A.23(a), müssen entweder die Betriebspannung

erhöht oder Techniken wie Pipelining, Parallelisierung etc. (siehe Anhang C.1.1) eingesetzt werden [14][17].

Nach [14] wird diese Methode in der Praxis nicht so häufig eingesetzt wie beispielsweise das im Anhang C.3.4 vorgestellten Verfahren. Die Energieeinsparung durch das Ersetzen von Operationen kann bei entsprechenden Systemen erheblich sein, wobei in der vorhanden Literatur keine genauen Angaben über die Beschaffenheit solcher Systeme und den Umfang der Einsparung gemacht werden.

C.3.6 Veränderung der Eingangssignalanordnung von Operationen

Die Schaltkapazität in einem System kann durch eine verbesserte Anordnung der Eingangssignale von Operationen reduziert werden. Um diese Methode zu verdeutlichen, wird das Problem betrachtet, drei 16-Bit breite Eingangssignale (IN1, IN2, IN3) zu addieren, die unterschiedliche dynamische Bereiche aufweisen.

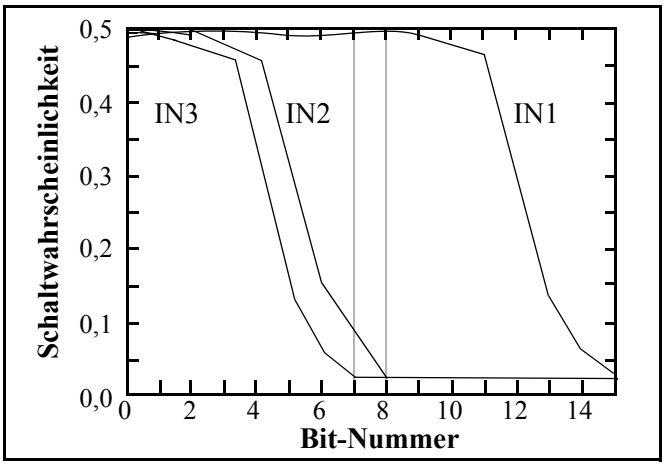


Bild A.24: Eingangssignale IN1, IN2 und IN3 mit unterschiedlichen dynamischen Bereichen

In Bild A.24 ist die Schaltwahrscheinlichkeit der einzelnen Bits für die drei Eingangssignale IN1, IN2 und IN3 aufgetragen. Das Eingangssignal IN1 weist eine hohe Varianz auf und benutzt die ganze Bit-Breite. Die Signale IN2 und IN3 besitzen einen kleineren dynamischen Bereich.

Es wird nun angenommen, dass die gewünschte Addition durch folgende Darstellung realisiert werden kann (Bild A.25(a)): Die Signale IN1 und IN2 werden im ersten Addierer aufsummiert, zu dem Ergebnis SUM1 wird im zweiten Addierer IN3 aufaddiert. Das Ergebnis der zweiten Addition ist SUM2. In diesem Fall ist die Schaltcharakteristik von SUM1 und dem Eingangssignal IN1 sehr ähnlich, da der dynamische Bereich von IN2 kleiner ist als der von IN1. SUM2 ist ebenfalls IN1 sehr ähnlich, da SUM1 einen größeren Bitbereich benutzt als IN3.

Eine zweite Darstellung (Bild A.25(b)) erhält man mittels Umordnung der Eingangssignale. Die Signale IN2 und IN3 werden dabei in der ersten Stufe addiert. Zum Ergebnis SUM1 wird in der zweiten Stufe IN1 aufsummiert. In diesem Fall hat das Ergebnis SUM1 aus dem ersten Addierer einen kleineren dynamischen Bereich als IN1 und somit eine geringere Schaltaktivität, da zwei Signale (IN2 und IN3) mit reduziertem dynamischen Bereich angelegt wurden.

In [14] wird von Systemen berichtet, die durch diese Methode eine Verringerung der Schaltaktivität und somit des entsprechenden Leistungsbedarfs von 30% erreichen.

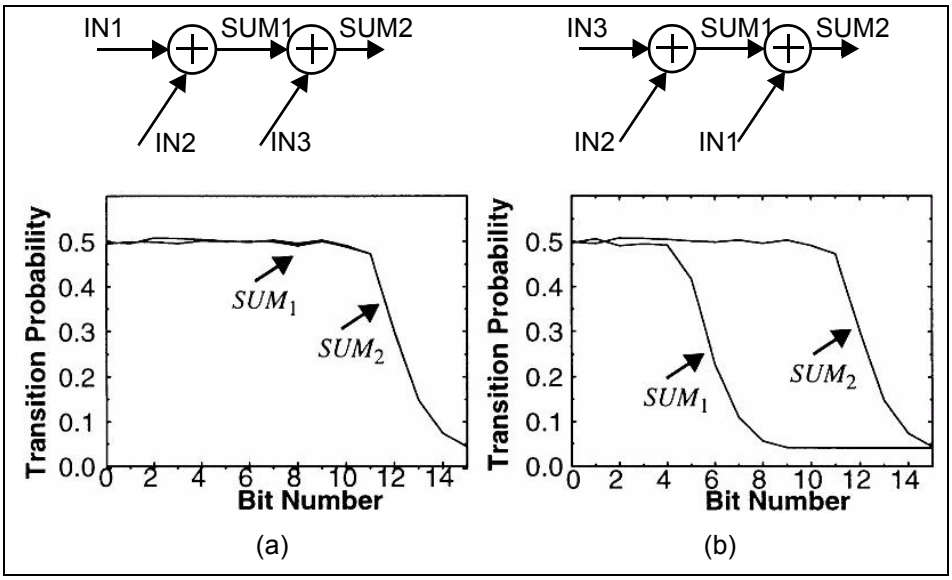


Bild A.25: Verringerung der Schaltaktivität durch eine bessere Anordnung der Eingangssignale

C.3.7 Resource Sharing

In DSP-Chips müssen eine Reihe von Operationen innerhalb einer Abtast-Periode abgearbeitet werden. Um diesen hohen Datendurchsatz zu ermöglichen, werden Parallelstrukturen eingesetzt, die leider sehr platzaufwendig sind. Im Gegensatz dazu werden in Schaltungen, bei denen der Datendurchsatz nicht so wichtig ist, häufig Time-Multiplexed-Architekturen verwendet. Diese ermöglichen eine mehrmalige Nutzung einer physikalisch nur einmal vorhandenen Operatoren-Einheit, wodurch weniger Hardware benötigt wird. Somit kann ein platzsparendes Design entwickelt werden. Dies kann zu der falschen Annahme führen, dass sich durch diese Einsparungen auch der Leistungsbedarf der Schaltung reduziert. Resource Sharing kann jedoch die effektive Schaltkapazität eines Systems erhöhen, da durch die Mehrfachnutzung die Schaltaktivität ansteigt, und somit auch der Energieverbrauch der Anordnung. Am Beispiel eines Datenbusses soll dieses Problem erläutert werden.

Die Ergebnisse von 2 Zählern sollen übertragen werden. In Bild A.26 wird diese Übertragung a) mit zwei parallelen Bussen und b) mit einer Time-Multiplexed Schaltung realisiert.

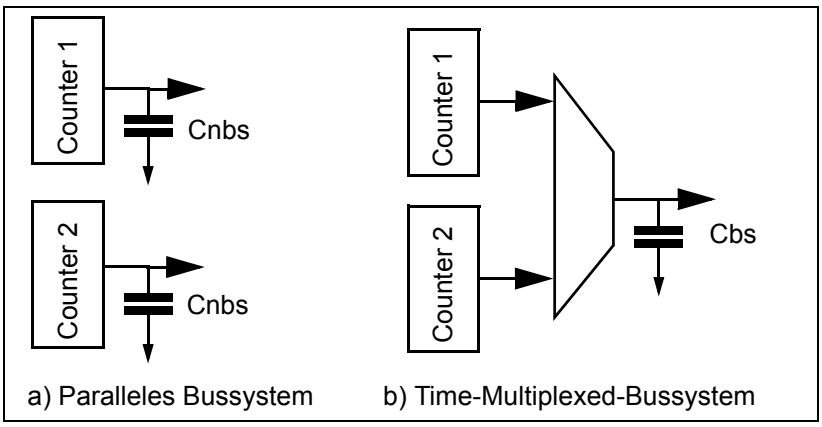


Bild A.26: Darstellung unterschiedlicher Bussysteme

Im Fall a) arbeiten die zwei separaten Busse jeweils mit der Taktfrequenz f, der Bus im Fall b) wird mit der doppelten Taktfrequenz 2*f betrieben, um den gleichen Datendurchsatz wie im Fall a) zu ermöglichen. Der Energieverbrauch der Paralleldarstellung im Fall a) wird folgendermaßen berechnet:

$$P_{no-bussharing} = \frac{1}{2}(\sum \alpha_i) C_{nbs} V^2 f + \frac{1}{2}(\sum \alpha_i) C_{nbs} V^2 f = (\sum \alpha_i) C_{nbs} V^2 f \quad A.28$$

C_{nbs} ist die physikalische Kapazität eines Bits von BUS1 und BUS2, unter der Annahme, dass jedes Bit die gleiche physikalische Kapazität besitzt. α_i ist die Schaltaktivität des i-ten Bits. Es ist zu beachten, dass α_i in diesem Beispiel sowohl $0 \rightarrow 1$ als auch $1 \rightarrow 0$ Übergänge berücksichtigt, wodurch der Faktor 1/2 in Formel 4.7 begründet wird. Die Schaltaktivität kann in diesem Fall sehr einfach bestimmt werden, da die Datensignale von einem Zähler kommen. Das LSB schaltet jede Taktperiode, das zweite LSB schaltet jede zweite Taktperiode etc., woraus die Formel 4.8 entwickelt werden kann [14].

$$\sum \alpha_i = 1 + \frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{128} \approx 2 \quad A.29$$

Dies bedeutet, dass jeder Bus durchschnittlich ca. zwei Transitionen pro Taktperiode hat, die Gesamtzahl der Übergänge pro Taktperiode bei dieser Paralleldarstellung also konstant bei vier liegt (Bild A.27).

Bei der "Time-Multiplexed"-Darstellung im Fall b) existiert nur ein Bus, der mit der doppelten Taktfrequenz 2*f betrieben wird. C_{bs} ist die physikalische Kapazität eines Bits vom Datenbus. Der Energieverbrauch dieses Systems wird durch Gleichung 4.9 bestimmt.

$$P_{bussharing} = \frac{1}{2}(\sum \alpha_v) C_{bs} V^2 2f = (\sum \alpha_v) C_{bs} V^2 f \quad A.30$$

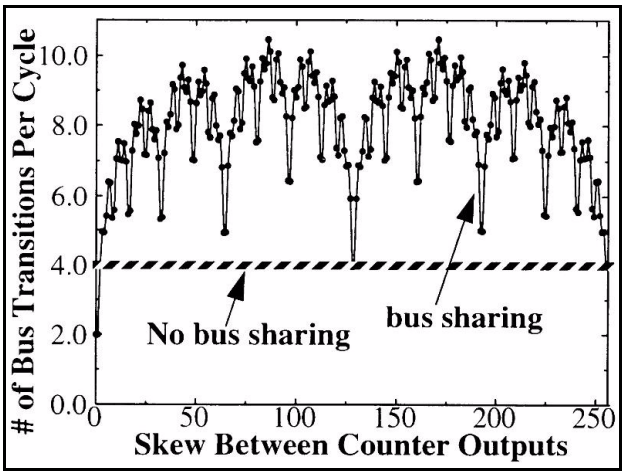


Bild A.27: Darstellung der Schaltaktivität von Bussystemen mit und ohne Resource Sharing

Die Summe der Transitionen pro Taktperiode ist in Bild A.27 abgebildet.

Im Gegensatz zu Fall a) ist hier die Schaltaktivität abhängig von Unterschieden in den Bitstellen der zwei Zähler.

Bis auf den Fall, dass beide Zähler jeweils den gleichen Zahlenwert ausgeben, die Verschiebung also gleich "0" ist, liegt die Schaltaktivität der Paralleldarstellung stets unter der Schaltaktivität des Time-Multiplexed-Systems.

Dieses Beispiel zeigt, dass Resource Sharing die Signalcharakteristik verändert und eine erhebliche Erhöhung der Schaltaktivität verursachen kann. Auch wenn angenommen wird, dass C_{bs} geringfügig kleiner ist als C_{nbs}, muss mit einem höheren Energieverbrauch gerechnet werden [14][56]. Zudem muss die Laufzeit durch den Multiplexer kompensiert werden. Somit sind kürzere Schaltzeiten notwendig, die eine kräftigere Dimensionierung und damit eine größere Kapazität zur Konsequenz haben.

C.4 Clock Gating

Clock Gating beruht auf der Idee, den Takt eines endlichen Automaten (FSM) vollständig abzuschalten, wenn sich weder Zustand noch Ausgabe dieses Systems ändern. Dadurch wird es möglich, die Schaltaktivität eines Moduls beispielsweise in Warteschleifen drastisch zu reduzieren.

Die Ausgabewerte eines Moore Automaten ändern sich nur dann, wenn sich der Zustand ändert. Befindet sich ein Moore Automat in einer sogenannten Self-Loop (Eigenschleife), so ändert sich weder der Zustand noch die Ausgabe der FSM, womit es sinnvoll sein kann, den Takt für die FSM abzuschalten. Diese Abschaltung erfolgt über eine Aktivierungsfunktion F_a. Diese gibt den logischen Wert "1" aus, wenn sich der Automat in einer Eigenschleife befindet.

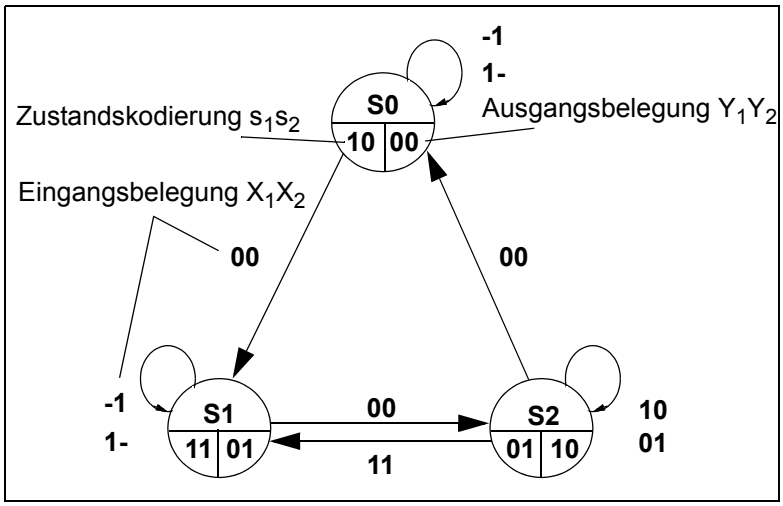


Bild A.28: Zustandsdiagramm eines Moore Automaten

Anhand der Tabelle 17 erhält man nach der Minimierung für den in Bild A.28 abgebildeten Automatengraphen folgende Aktivierungsfunktion F_a:

$$F_a = X_1\overline{X_2} + \overline{X_1}X_2 + X_1S_1$$

Tabelle 17: Self-Loops

X_1	X_2	S_t		S_{t+1}		Y	Self-Loops
-	1	S0	10	S0	10	00	-110
1	-	S0	10	S0	10	00	1-10
-	1	S1	11	S1	11	01	-111
1	-	S1	11	S1	11	01	1-11
1	0	S2	01	S2	01	10	1001
0	1	S2	01	S2	01	10	0101

Es ist zu beachten, dass diese Funktion bei komplexeren Automaten recht aufwendig werden kann. Man muss somit prüfen, ob der zusätzliche Schaltungsaufwand durch den Einbau der realisierten Aktivierungsfunktion nicht mehr Energie benötigt als durch die Abschaltung im Falle von Eigenschleifen eingespart wird. Durchschnittlich kann mit Clock Gating bis zu 25% des Leistungsverbrauches eingespart werden. Der zusätzliche Flächenbedarf beträgt dabei durchschnittlich 5%. Weiter muss darauf geachtet werden, dass F_a hazardfrei ist, da sonst Fehlfunktionen des Automaten auftreten können.

Die hier vorgestellte Idee des Clock Gating ist in dem Low-Power Toolpaket "Pie" (Stanford University) implementiert [83]. Mittels dieses Toolpakets kann man aus einem Zustandsdiagramm die erforderlichen Self-Loop Informationen extrahieren und daraus eine Aktivierungsfunktion F_a generieren. Die Abschätzung des Energieverbrauchs des originalen und des

modifizierten Automaten ist mit "Pie" ebenfalls möglich, wodurch die Effektivität der Clock-Gating Technik überprüft werden kann [83].

Ein Mealy-Automat muss zu einem entsprechenden Moore Automat transformiert werden, um die hier vorgestellte Clock-Gating Methode anwenden zu können. Bei einem Medvedev-Automat hingegen kann genauso verfahren werden wie bei einem Moore-System, da die Ausgangsbelegungen mit der Zustandskodierung identisch sind ($Y^t=Z^t$) und sich in einer Eigenschleife nicht ändern können.

C.5 Adiabate Schaltungstechnik

In einem CMOS-Schaltkreis entsteht normalerweise durch das Laden und Entladen der Lastkapazität der größte Teil der Verlustleistung. In konventionellen CMOS-Strukturen ist dieser Energieverlust nicht zu vermeiden und kann effektiv nur reduziert werden, indem man die Schaltaktivität der Gatter verringert beziehungsweise die Versorgungsspannung senkt. In den letzten Jahren gab es erhebliche Forschungsaktivitäten auf dem Gebiet des Adiabaten Schaltens. Ziel dieser Schaltungstechnik ist es, die in Kapazitäten gespeicherte Energie zurückzugewinnen und nicht zu 100% als Wärme zu verlieren. Wie bereits in Anhang A erläutert, wird die von der Spannungsversorgung gelieferte Energie $E = C_L V_{dd}^2$ beim Laden und Entladen in einer konventionellen CMOS-Struktur zu 100% in Verlustwärme umgesetzt [2].

C.5.1 Prinzip des adiabaten Schaltens

Durch die Technik des adiabaten Schaltens kann dieser Energieverlust deutlich verringert werden, ohne dass die Versorgungsspannung reduziert werden muss. Adiabates Schalten beruht auf der Theorie des adiabaten Ladens [83]. Diese wiederum beruht auf der Idee, dass die Verlustleistung beim Aufladen einer Kapazität über einen Widerstand R durch Verlängerung der Ladezeit T verringert werden kann (Bild A.29).

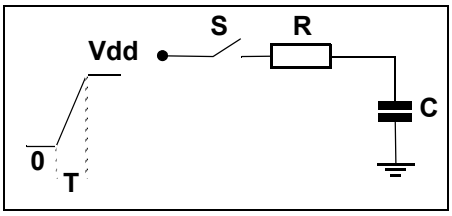


Bild A.29: Prinzip des „adiabaten Ladens“

Nach dem Modell in Bild A.29 kann die Verlustleistung E_{diss} in R nach Gleichung *A.31* berechnet werden. Dabei soll gelten T>>RC.

$$E_{diss} = P \cdot T = I^2 R \cdot T = \left(\frac{CV_{dd}}{T}\right)^2 R \cdot T = \left(\frac{RC}{T}\right) CV_{dd}^2 \qquad A.31$$

Durch Verlängerung der Ladezeit T kann man somit theoretisch den Strom $I = (CV_{dd})/T$ beliebig verkleinern, wodurch sich die entsprechende Verlustleistung beliebig reduzieren lässt. Im Vergleich zu der konventionellen Ladetechnik reduziert sich der Energieverbrauch um den Faktor 2RC/T [46][56].

C.5.2 Laden und Entladen mittels einer Induktivität

Um den Energieverlust beim Entladen der Lastkapazität zu kompensieren, kann man eine Induktivität zum Laden und Entladen benutzen. Dadurch wird es möglich, die Ladung in sinusförmigen Signalverläufen zu transportieren. Da der Ladestrom nicht mehr konstant, sondern sinusförmig ist, verändert sich E_{diss}. Diese Veränderung wird durch den Shape-Faktor ξ berücksichtigt (Gleichung *A.32*).

$$E_{diss} = \xi \frac{RC}{T} CV_{dd}^2 \qquad \xi = \frac{\pi^2}{8} \qquad A.32$$

Um den Datendurchsatz eines Systems mit adiabater Schaltungslogik erhöhen zu können, werden entsprechend entwickelte, adiabate Pipelinestrukturen eingesetzt [38][77].

C.5.3 Schrittweise Lade- Entladetechnik

Ein anderer Weg eine Kapazität über einen Widerstand mit möglichst wenig Verlustleistung zu laden, ist der, mittels mehrerer Spannungsquellen den Ladevorgang schrittweise durchzuführen [27]. Statt einer Induktivität werden mehrere Spannungsquellen benutzt, die durch eine FSM gesteuert zugeschaltet werden.

Die Spannungswerte $V_1 ... V_N$ der einzelnen Quellen steigen bis V_N=V stufenweise an. Die als Wärme abgegebene Energie beträgt bei dieser Technik:

$$E_{diss} = CV\left(\frac{V}{2N}\right) = CV^2 \frac{1}{2N} \qquad A.33$$

Die Verwendung von mehreren Spannungsquellen ist für den realen Einsatz dieser Technik sehr ungeeignet. Die Spannungsquellen werden deswegen bis auf eine durch sogenannte „Tank-Kapazitäten“ C_T ersetzt [92][91]. Diese Tank-Kapazitäten müssen deutlich größer sein als die Lastkapazität C_L.

Bedingt durch zusätzliche Schaltungstechnik wie Spulen, FSM und anderes ist der Platzbedarf der adiabaten Schaltungstechniken größer als beim Einsatz konventioneller CMOS-Strukturen. Generelle Aussagen über die zusätzlich benötigte Chipfläche sind recht schwierig zu treffen, da diese von der Komplexität der betrachteten Schaltung abhängt. Der zusätzliche Flä-

chenbedarf realisierter Schaltungen liegt aber zwischen 6% (Pad Driver [93]) bis 250% [56][93].

Anhang D
Low Power Synthesemethoden

Dieser Anhang stammt aus den Anfängen der Arbeit, als der Low Power Aspekt eine zentrale Rolle spielte. Bei der Synthese gibt es eine Vielzahl unterschiedlicher Techniken, um den Leistungsbedarf einer Schaltung zu vermindern.

D.1 Extraktion von Ausdrücken

Komplexe Schaltungssysteme werden meist modular aufgebaut. Mittels algebraischer Division kann man versuchen, bei diesen Modulen gemeinsame Untereinheiten zu finden und diese aus den Modulen herauszuziehen, wie in Bild A.30 schematisch dargestellt.

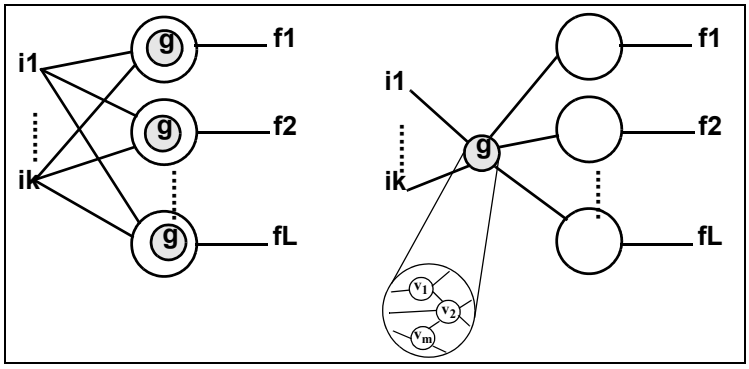


Bild A.30: Zerlegung von Modulen

Die Einheit g ist aus den Modulen $f_1...f_L$ extrahiert worden, wodurch sich die Kapazitäten, die von $i_1...i_k$ geschaltet werden müssen, verringern. Genauer: Jeder Knoten i_j muss L-1 weniger Eingänge (von g) schalten, wobei L die Anzahl der faktorisierten Module ist. In [56] wird zur Berechnung der Energieeinsparung folgende Methode angegeben: Es wird angenommen, dass $g=g(i_1...i_k)$ gilt. $v_1,...,v_m$ stehen für die internen Knoten von g. Die Leistungsreduktion ΔP ist dann:

$$\Delta P = \frac{C_0 V_{dd}^2}{2T_{cycle}} \cdot (L-1) \cdot \left(\sum_{k=1}^{K} n_{ik} E_{ik} + \sum_{m=1}^{M} n_{vm} E_{vm} \right) \qquad A.34$$

Dabei ist C_0 die Unit-Kapazität; n_{ik} ist die Anzahl der Gatter in g, die von i_k angesteuert werden; n_{vm} ist die Anzahl der Gatter in g, die vom Signal v_m angesteuert werden; $E_x = 2p_x(1-p_x)$ und V_{dd} ist die Versorgungsspannung.

Der erste Term berücksichtigt die Energieeinsparung durch die Verminderung der Verzweigungen. Die Leistungsreduktion, die durch eine geringere Anzahl der Einheiten g zustande kommt, wird mit Hilfe des zweiten Terms berechnet.

Im Allgemeinen kann mit dieser Methode eine Leistungsreduktion von 10% erzielt werden [77].

D.2 Re-Timing

Beim Re-Timing werden heuristisch Gatter gesucht, deren Ausgänge eine hohe Glitch-Wahrscheinlichkeit und eine große Last-Kapazität besitzen. An deren Ausgängen wird ein Flip-flop hinzugefügt, das unnütze Schaltaktivitäten des angeschlossenen Netzwerkes, das durch C_L repräsentiert wird, verhindert (siehe Bild A.31) und somit eine Reduktion des Energieverbrauches ermöglicht.

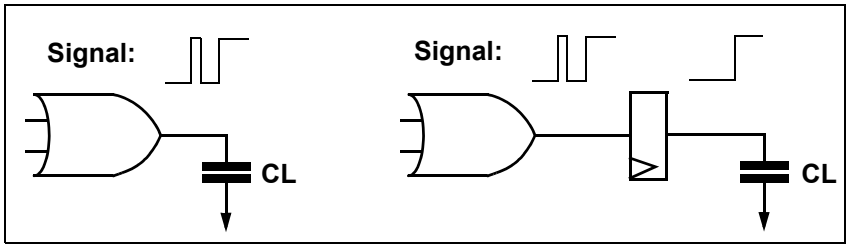


Bild A.31: Vermeidung von Glitches mittels Re-Timing

Dieses Verfahren arbeitet nicht optimal, da das Re-Timing eines einzelnen Gate-Ausgangs meist keine großen Verbesserungen bringt, und die Auswirkungen der Veränderungen auf den Energieverbrauch auch nur schwer vorhersagbar sind. Re-Timing ist besonders effektiv in Schaltungen, die eine Pipelinestruktur besitzen. In einer 3-stufigen Pipeline Schaltung ist eine Energieeinsparung von bis zu 8% zu erreichen [77][56][84].

D.3 Decomposition

Dieses Prinzip beruht auf der Beobachtung, dass man den Energieverbrauch einer Schaltung verringern kann, indem man komplexe Gatter gezielt in kleine Untergatterstrukturen zerlegt und somit die interne Schaltaktivität eines Gatternetzwerkes reduziert. Die Grundidee ist, Eingangssignale mit einer hohen Schaltwahrscheinlichkeit durch diese Zerlegung so spät wie möglich in eine Gatterstruktur einspeisen zu können, um dadurch die Schaltaktionen so gering wie möglich zu halten.

In Bild A.32 ist ein UND-Gatter mit vier Eingängen dargestellt. Jeder Eingang besitzt eine gewisse Schaltwahrscheinlichkeit p(x). Diese komplexe Gatterstruktur kann man in unterschiedliche Unterstrukturen zerlegen, die nur noch UND-Gatter mit zwei Eingängen benutzen. Mittels eines Zero-Delay-Modells können die Strukturen nach ihrem unterschiedlichen Energieverbrauch bewertet werden. Die in Bild A.32(b) dargestellte Baumanordnung, bei der jedes Signal die gleiche Anzahl von Gatter durchlaufen muss, hat einen höheren Energieverbrauch als

die Kettenanordnung(Bild A.32(a)), bei der das Signal mit der höchsten Schaltwahrscheinlichkeit sehr spät in das Netzwerk eingeführt wird und dadurch weniger Gatter durchlaufen muss als die restlichen Signale.

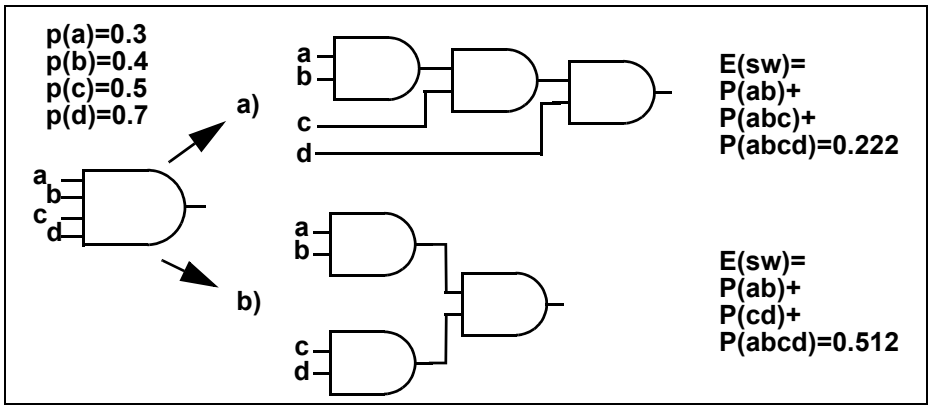


Bild A.32: Technologie Decomposition für Low Power

Wenn die entsprechenden Schaltwahrscheinlichkeiten vorliegen, kann mit der Kettenstruktur eine Reduzierung des Energieverbrauchs von bis zu 5% gegenüber der - ausbalancierten und somit für viele Anwendungen meist besseren - Baumstruktur erreicht werden.

Wie auch dieses Beispiel zeigt, hängt die Möglichkeit einer Leistungsreduktion in einem Schaltkreis extrem von den Eigenschaften der Eingangssignale ab [77][56]. Um diese Methode gewinnbringend einsetzen zu können, müssen die Schaltwahrscheinlichkeiten p(x) der Gattereingänge fest definiert sein. Nur so kann sichergestellt werden, dass Signale, die hohe Schaltaktivität zur Folge haben, recht spät in die Gatterstruktur eingespeist werden. Diese Voraussetzung ist in real aufgebauten Systemen selten gegeben, wodurch die Einsatzmöglichkeit dieser Technik sehr begrenzt ist.

D.4 Technologie Mapping

Die Idee, den Leistungsverbrauch eines Systems während des "Mapping" zu verbessern, wird in der Literatur häufig diskutiert [77][56]. Das Grundprinzip beruht auf der Tatsache, dass die Last-Kapazität einzelner Knoten innerhalb eines definierten Gatters deutlich geringer ist als außerhalb, da dort weitere Kapazitäten, zum Beispiel durch Verbindungsleitungen, hinzukommen. Aus diesem Grund wird versucht, Knoten, die eine hohe Schaltaktivität besitzen, ins Innere eines Komplexgatters zu legen, um so die effektive Schaltkapazität zu senken.

Im Vergleich zu einem "Delay-Target Technologie Mapper" kann man mit dieser Methode bis zu 18% Energie einsparen [77].

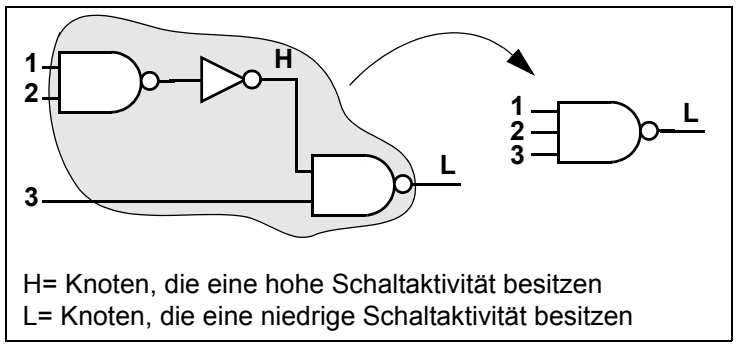


Bild A.33: Verlagerung von Knoten mit hoher Schaltaktivität in Komplexgatter

D.5 Pre-Computation

Pre-Computation beruht auf der Idee, selektiv Ausgabewerte eines Systems einen Taktzyklus vorherzubestimmen, um somit in der darauffolgenden Taktperiode Schaltaktivität einsparen zu können. Dieses Prinzip soll anhand einer Komparatorschaltung (Bild A.34) näher erläutert werden.

Mit diesem Komparator soll das n-Bit lange Datenwort C mit dem n-Bit langen Datenwort D verglichen werden. Wenn C>D vorliegt, geht der Ausgang f auf "1", für C≤D ist der Ausgang gleich "0". Die Eingangswerte C und D werden in 2 Registern, R1 und R2, zwischengespeichert. R2 besitzt einen Enable-Eingang LE. Ist dieser "1", so wird der Registerinhalt in den eigentlichen Komparator weitergeleitet, ist er "0", so behalten die Registerausgänge ihren vorhergehenden Wert, d.h es treten keine Schaltvorgänge an den entsprechenden Komparatoreingängen auf.

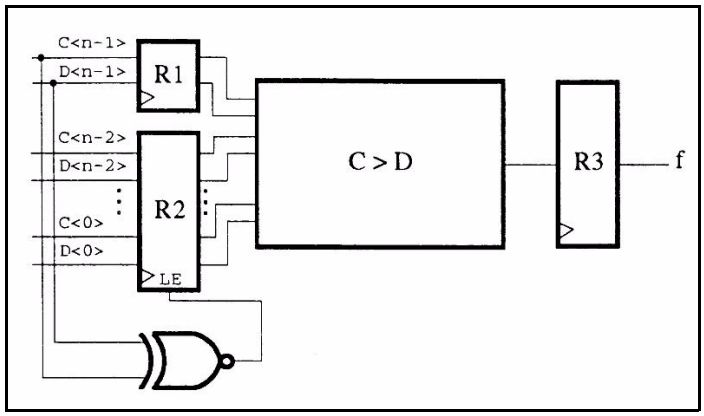


Bild A.34:

In dieser Schaltung beruht das Prinzip der Pre-Computation auf der Überlegung, dass man allein mit den MSBs von C und D vorherberechnen kann, ob C>D vorliegen kann, ohne die Bits von Register 2 mit in den Vergleich einbeziehen zu müssen. Ist beispielsweise das MSB von C gleich "1" und das MSB von D gleich "0" so wird immer C>D gelten, egal welchen Wert die

Bits von Register 2 haben. Das heißt, für diesen Fall ist LE von R2 auf "0", die vorgehenden Werte werden beibehalten. Es treten somit keine Schaltaktivitäten an den entsprechenden Eingängen des Komparators auf. Die Schaltung verbraucht weniger Energie.

Lediglich wenn beide MSBs gleich sind, werden die Bits von R2 zum Vergleich benötigt. Zur Steuerung des LE-Eingangs von R2 wird also ein XNOR-Gatter benötigt, an dessen Eingänge die MSBs von C und D gelegt werden.

Unter der Annahme von gleichwahrscheinlichen Eingangssignalen liegt die Schaltwahrscheinlichkeit des XNOR-Gatters bei P(XNOR)=50% unabhängig von n. Dadurch kann eine Reduzierung des Energieverbrauchs von 50% [56] erreicht werden, wobei in diesem Beispiel der zusätzliche Leistungsbedarf der Pre-Computation-Logik vernachlässigt wurde.

Bei Schaltungen, die eine komplexere Pre-Computation-Logik zur Vorhersage benötigen, muss natürlich vorher abgeklärt werden, ob der höhere Schaltungsaufwand die gewonnene Leistungsreduktion nicht wieder zunichte macht [77].

Anhang E
Anfrageaktivierte Datenauswertung

In Bild A.35 wird ein vollständigerer Ablauf der Datenausbreitung über eine Mikropipeline mit „Anfrage-aktivierter Datenauswertung“ und über eine Mikropipeline mit „Bestätigung-aktivierter Datenauswertung“ gezeigt.

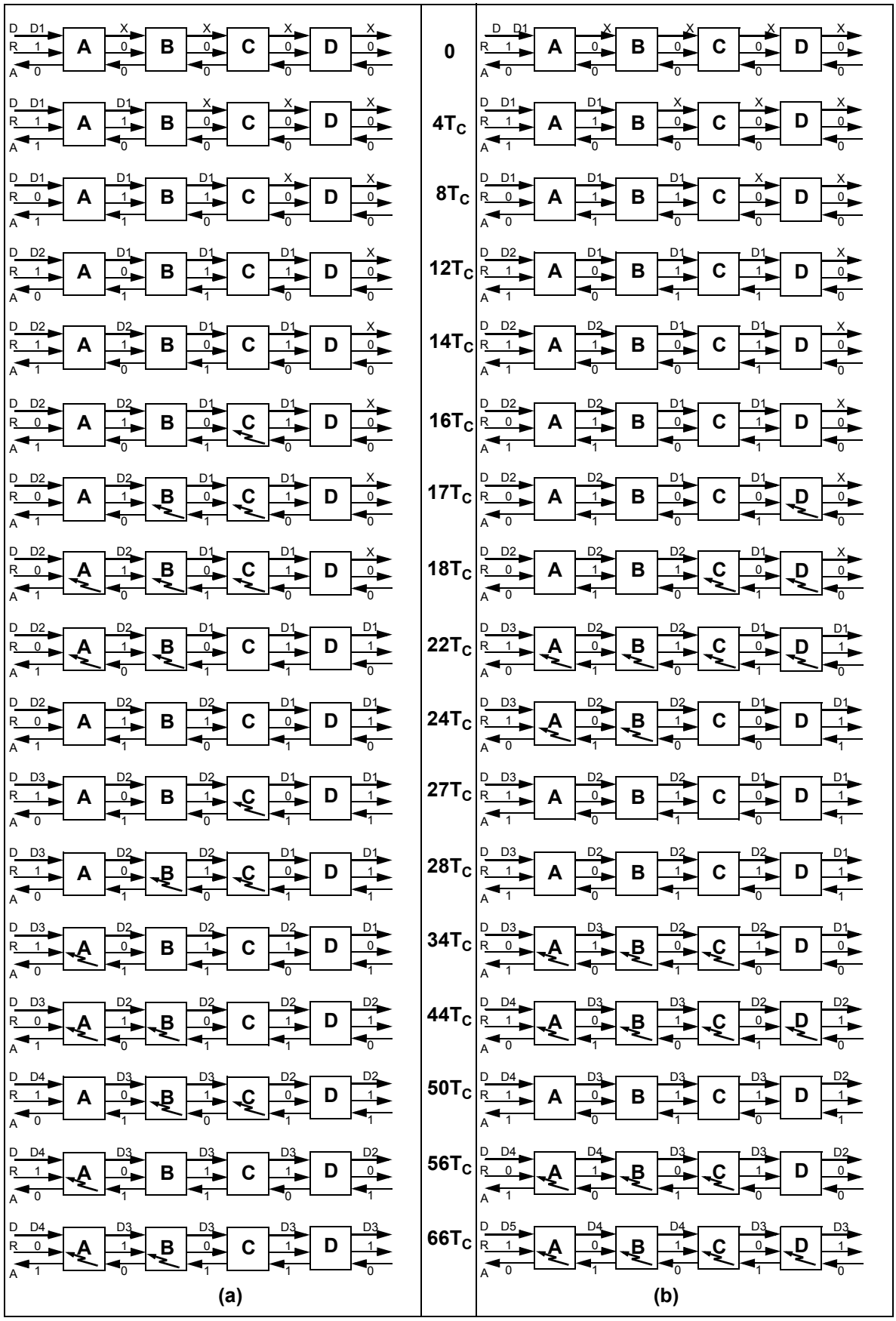


Bild A.35: Ausbreitung von Daten über eine asynchrone Pipeline; Logikauswertung über die Anfrage (a) oder die Bestätigung (b).

Anhang F
Simulationsergebnisse der Kontrollstrukturen

Nachfolgend sind die Leistungswerte für die unterschiedlichen Pipeline-Implementierungen gezeigt. Nachfolgend Leistungswerte für verschiedene asynchrone Implementierungen; Fehler bezeichnet dabei die Verletzungen von Zeitbedingungen - Es wurden trotzdem über alle Stufen hinweg korrekte Werte geliefert

Tabelle 18: Leistungswerte für die Pipeline SCHIEBEN mit maximaler Datenrate

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
23.904	1.5764	0.376	9269.09	0	adbff
32.670	1.5920	0.520	8724.04	0	adblc
22.518	1.6319	0.367	8537.00	0	adblo
21.357	1.6062	0.343	9879.00	0	asbff
29.915	1.1931	0.356	6992.20	0	asblc
25.651	1.5372	0.394	8171.10	0	asblo
24.075	1.1596	0.279	8374.70	0	aubff
21.513	1.0370	0.223	6667.00	0	aublc
20.115	1.0986	0.220	6910.89	0	aublo
36.457	2.5144	0.916	13740.70	0	rdbff
35.942	2.1341	0.767	10162.90	0	rdblc
46.557	3.3342	1.552	13683.43	0	rdblo
26.667	1.4404	0.384	9309.69	0	rdeff
55.689	3.0230	1.683	12439.70	0	rdelc
45.617	2.9082	1.326	11821.51	0	rdelo
27.442	1.9521	0.535	11464.30	0	rdwff
45.370	2.6828	1.217	12358.19	0	rdwlc
34.396	2.3289	0.801	11098.00	0	rdwlo
25.757	1.4547	0.374	10732.50	7	rsbff
29.131	1.4614	0.425	8049.10	0	rsblc
32.996	1.7689	0.583	10203.59	0	rsblo
26.602	1.6798	0.446	8903.00	0	rselo
24.877	1.5706	0.390	10610.50	0	rswff
47.098	1.6704	0.786	9268.59	0	rswlc
31.944	1.4295	0.456	7886.60	0	rswlo
26.228	1.1597	0.304	8212.09	0	rubff
34.167	1.2640	0.431	7480.00	0	rublc
19.783	1.1109	0.219	6992.20	0	rublo
21.545	1.0338	0.222	6748.29	0	ruelo
26.446	1.0587	0.279	6585.70	0	ruwlo
37.063	1.5092	0.559	9106.40	0	rdbff_half
48.755	1.4956	0.729	8171.10	0	rdblc_half
46.585	2.1567	1.004	9959.90	0	rdblo_half
37.806	1.2810	0.484	8659.29	0	rsbff_half
50.549	1.3006	0.657	7642.59	0	rsblc_half
41.757	1.5086	0.629	7886.70	0	rsblo_half
40.917	1.1179	0.457	8293.39	0	rubff_half
62.523	1.6356	1.022	8455.70	0	rublc_half
55.536	2.0426	1.134	9512.59	0	rublo_half

Tabelle 19: Leistungswerte für die Pipeline ADDIEREN_SUBTRAHIEREN mit maximaler Datenrate

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
56.122	2.3176	1.300	14724.76	0	adbff
60.868	2.3748	1.445	14045.52	0	adblc
52.512	2.3124	1.214	13919.45	0	adblo
53.670	2.3746	1.274	15411.91	0	asbff
56.575	1.8817	1.064	12138.87	0	asblc
59.236	2.1943	1.299	13484.44	0	asblo
54.069	1.8506	1.000	13659.63	0	aubff
45.481	1.6543	0.751	11488.48	0	aublc
52.843	1.7171	0.907	12049.45	0	aublo
70.594	3.3961	2.397	19692.30	0	rdbff
64.711	2.7783	1.797	15098.19	0	rdblc
70.184	3.5187	2.469	18200.06	0	rdblo
62.634	2.1859	1.369	14785.68	0	rdeff
86.183	3.8698	3.335	18228.68	0	rdelc
75.478	3.4324	2.590	17171.45	0	rdelo
61.581	2.7701	1.705	17090.70	0	rdwff
75.842	3.5535	2.695	18090.25	0	rdwlc
66.542	3.1060	2.066	16801.59	0	rdwlo
61.238	2.3689	1.450	16399.54	0	rsbff
63.626	2.2049	1.402	13382.77	0	rsblc
64.377	2.5172	1.620	15817.75	0	rsblo
62.537	2.3839	1.490	14289.54	0	rselo
59.403	2.4114	1.432	16330.38	0	rswff
75.081	2.4973	1.874	14935.59	0	rswlc
64.440	2.0941	1.349	13155.24	0	rswlo
61.490	1.8592	1.143	13513.29	0	rubff
67.866	1.9897	1.350	12724.24	0	rublc
55.243	1.7569	0.970	12236.44	0	rublo
56.333	1.6543	0.931	11903.11	0	ruelo
61.686	1.6828	1.038	11724.25	0	ruwlo
59.116	2.0155	1.191	13602.72	0	rdbff_half
71.199	2.0032	1.426	12557.62	0	rdblc_half
70.971	2.7504	1.951	14785.46	0	rdblo_half
56.295	1.7570	0.989	13094.65	0	rsbff_half
68.835	1.7884	1.231	11976.28	0	rsblc_half
62.081	1.9600	1.216	12293.57	0	rsblo_half
58.768	1.5774	0.927	12692.16	0	rubff_half
81.447	2.1671	1.765	12886.95	0	rublc_half
77.763	2.5500	1.982	14130.83	0	rublo_half

Tabelle 20:Leistungswerte für die Pipeline SHIFTEN mit einer Datenrate von 100 MHz

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
81.004	1.5687	1.270	9269.09	0	adbff
86.642	1.6023	1.388	8724.04	0	adblc
79.123	1.6263	1.286	8537.00	0	adblo
79.724	1.5855	1.264	9879.00	0	asbff
82.119	1.1974	0.983	6992.20	0	asblc
81.163	1.5330	1.244	8171.10	0	asblo
79.604	1.1595	0.923	8374.70	0	aubff
77.909	1.0415	0.811	6667.00	0	aublc
77.693	1.0986	0.853	6910.89	0	aublo
80.966	2.4548	1.987	13740.70	0	rdbff
81.533	1.9927	1.624	9593.79	0	rdblc
82.164	2.7253	2.239	12545.24	0	rdblo
81.286	1.4437	1.173	9309.69	0	rdeff
95.347	2.9903	2.851	12439.70	0	rdelc
84.815	2.6219	2.223	11301.20	0	rdelo
79.976	1.9326	1.545	11464.30	0	rdwff
88.603	2.7157	2.406	12358.19	0	rdwlc
83.071	2.3289	1.934	11098.00	0	rdwlo
79.631	1.4267	1.136	10732.50	0	rsbff
80.515	1.4650	1.179	8049.10	0	rsblc
81.517	1.7549	1.430	10203.59	0	rsblo
82.497	1.6756	1.382	8903.00	0	rselo
80.342	1.5696	1.261	10610.50	0	rswff
87.870	1.5561	1.367	9268.59	0	rswlc
82.516	1.4293	1.179	7886.60	0	rswlo
79.593	1.1686	0.930	8212.09	0	rubff
82.365	1.2640	1.041	7480.00	0	rublc
77.730	1.1198	0.870	6992.20	0	rublo
78.426	1.0338	0.810	6748.29	0	ruelo
79.629	1.0681	0.850	6585.70	0	ruwlo
90.136	1.5379	1.386	9106.40	0	rdbff_half
95.294	1.4956	1.425	8171.10	0	rdblc_half
92.737	2.1486	1.992	9959.90	0	rdblo_half
89.074	1.2809	1.140	8659.29	0	rsbff_half
93.893	1.3006	1.221	7642.59	0	rsblc_half
88.188	1.5082	1.330	7886.70	0	rsblo_half
87.782	1.1179	0.981	8293.39	0	rubff_half
95.856	1.5901	1.524	8455.70	0	rublc_half
92.582	2.0231	1.873	9512.59	0	rublo_half

Tabelle 21: Leistungswerte für die Pipeline ADDIEREN_SUBTRAHIEREN mit einer Datenrate von 100 MHz

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
90.467	2.3096	2.089	14724.76	0	adbff
95.694	2.3847	2.282	14045.52	0	adblc
87.392	2.3153	2.023	13919.45	0	adblo
89.410	2.3473	2.098	15411.91	0	asbff
91.055	1.8994	1.729	12138.87	0	asblc
90.763	2.2099	2.005	13484.44	0	asblo
88.849	1.8499	1.643	13659.63	0	aubff
85.468	1.6630	1.421	11537.25	0	aublc
86.760	1.7291	1.500	12049.45	0	aublo
89.965	3.3345	2.999	19562.22	0	rdbff
90.803	2.7996	2.542	15081.93	0	rdblc
92.035	3.5512	3.268	18200.06	0	rdblo
90.591	2.1884	1.982	14785.68	0	rdeff
106.447	3.8916	4.142	18179.90	0	rdelc
94.769	3.4104	3.232	16927.55	0	rdelo
88.865	2.7541	2.447	17090.70	0	rdwff
98.996	3.5861	3.550	18090.25	0	rdwlc
93.060	3.0953	2.880	16736.55	0	rdwlo
89.334	2.1692	1.937	16350.76	0	rsbff
89.762	2.1980	1.972	13301.47	0	rsblc
90.630	2.4680	2.236	15768.97	0	rsblo
92.019	2.3666	2.177	14289.54	0	rselo
89.205	2.3101	2.060	16151.52	0	rswff
98.336	2.3191	2.280	14724.22	0	rswlc
92.005	2.0918	1.924	13155.24	0	rswlo
88.711	1.8686	1.657	13513.29	0	rubff
92.149	1.9895	1.833	12724.24	0	rublc
86.624	1.7762	1.538	12236.44	0	rublo
87.769	1.6663	1.462	11903.11	0	ruelo
88.782	1.7008	1.510	11724.25	0	ruwlo
98.525	2.0456	2.015	13602.72	0	rdbff_half
104.074	2.0027	2.084	12557.62	0	rdblc_half
101.995	2.6678	2.721	14622.86	0	rdblo_half
97.231	1.7552	1.706	13094.65	0	rsbff_half
102.533	1.7881	1.833	11976.28	0	rsblc_half
96.626	1.9512	1.885	12293.57	0	rsblo_half
95.812	1.5759	1.509	12692.16	0	rubff_half
104.819	2.1109	2.212	12886.95	0	rublc_half
101.824	2.5303	2.576	14130.83	0	rublo_half

Tabelle 22: Leistungswerte für die Pipeline SHIFTEN mit einer Datenrate von 7,8 MHz

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
65.604	1.5690	1.029	9269.09	0	adbff
71.372	1.5213	1.085	8415.09	0	adblc
63.423	1.6263	1.031	8537.00	0	adblo
64.324	1.5858	1.020	9879.00	0	asbff
66.717	1.1974	0.798	6992.20	0	asblc
65.763	1.5330	1.008	8171.10	0	asblo
64.204	1.1596	0.744	8374.70	0	aubff
62.209	1.0415	0.647	6667.00	0	aublc
62.293	1.0986	0.684	6910.89	0	aublo
65.291	2.4553	1.603	13740.70	0	rdbff
65.833	1.9926	1.311	9593.79	0	rdblc
66.764	2.7549	1.839	12545.24	0	rdblo
65.586	1.4442	0.947	9309.69	0	rdeff
79.947	2.9903	2.390	12439.70	0	rdelc
69.415	2.6219	1.819	11301.20	0	rdelo
63.605	1.8908	1.202	11301.70	0	rdwff
73.203	2.7157	1.987	12358.19	0	rdwlc
67.671	2.3289	1.575	11098.00	0	rdwlo
64.231	1.4269	0.916	10732.50	0	rsbff
65.007	1.4413	0.936	7967.90	0	rsblc
65.817	1.7549	1.155	10203.59	0	rsblo
66.894	1.6756	1.120	8903.00	0	rselo
64.642	1.5701	1.014	10610.50	0	rswff
72.470	1.5561	1.127	9268.59	0	rswlc
67.116	1.4293	0.959	7886.60	0	rswlo
63.893	1.1690	0.746	8212.09	0	rubff
66.965	1.2640	0.846	7480.00	0	rublc
62.030	1.1198	0.694	6992.20	0	rublo
63.015	1.0338	0.651	6748.29	0	ruelo
63.961	1.0681	0.683	6585.70	0	ruwlo
69.026	1.4956	1.032	8943.79	0	rdbff_half
75.394	1.4956	1.127	8171.10	0	rdblc_half
72.837	2.1486	1.564	9959.90	0	rdblo_half
67.954	1.2386	0.841	8496.70	0	rsbff_half
72.773	1.2583	0.915	7480.00	0	rsblc_half
68.288	1.5082	1.029	7886.70	0	rsblo_half
66.662	1.0756	0.717	8130.79	0	rubff_half
74.737	1.5477	1.156	8293.09	0	rublc_half
72.682	2.0231	1.470	9512.59	0	rublo_half

Tabelle 23: Leistungswerte für die Pipeline ADDIEREN_SUBTRAHIEREN mit einer Datenrate von 9,9 MHz

Zeit (ns)	Energie (µW)	Performanz	Fläche	Fehler	Name
89.767	2.3097	2.073	14724.76	0	adbff
94.739	2.3558	2.231	13947.96	0	adblc
86.692	2.3209	2.012	13919.45	0	adblo
88.649	2.3432	2.077	15395.66	0	asbff
90.355	1.8951	1.712	12138.87	0	asblc
90.063	2.2065	1.987	13484.44	0	asblo
88.149	1.8499	1.630	13659.63	0	aubff
84.524	1.6458	1.391	11488.48	0	aublc
86.060	1.7291	1.488	12049.45	0	aublo
89.265	3.3345	2.976	19562.22	0	rdbff
90.103	2.7996	2.522	15081.93	0	rdblc
91.335	3.5512	3.243	18200.06	0	rdblo
89.830	2.1843	1.962	14769.42	0	rdeff
105.503	3.8789	4.092	18131.12	0	rdelc
94.069	3.4104	3.208	16927.55	0	rdelo
87.616	2.7203	2.383	16960.63	0	rdwff
98.235	3.5819	3.518	18073.99	0	rdwlc
92.360	3.0953	2.858	16736.55	0	rdwlo
88.634	2.1693	1.922	16350.76	0	rsbff
88.949	2.1719	1.931	13212.15	0	rsblc
89.930	2.4680	2.219	15768.97	0	rsblo
91.319	2.3666	2.161	14289.54	0	rselo
88.505	2.3102	2.044	16151.52	0	rswff
97.514	2.3107	2.253	14691.69	0	rswlc
91.305	2.0918	1.909	13155.24	0	rswlo
88.011	1.8686	1.644	13513.29	0	rubff
91.205	1.9768	1.802	12675.45	0	rublc
85.924	1.7744	1.524	12236.44	0	rublo
87.069	1.6663	1.450	11903.11	0	ruelo
88.082	1.7008	1.498	11724.25	0	ruwlo
79.795	2.0211	1.612	13505.16	0	rdbff_half
85.952	1.9987	1.717	12541.36	0	rdblc_half
83.995	2.6680	2.240	14622.86	0	rdblo_half
77.889	1.7095	1.331	12915.79	0	rsbff_half
83.923	1.7671	1.483	11894.98	0	rsblc_half
78.382	1.9429	1.522	12261.04	0	rsblo_half
77.202	1.5555	1.200	12610.86	0	rubff_half
85.722	2.0730	1.777	12740.61	0	rublc_half
83.824	2.5305	2.121	14130.83	0	rublo_half

Für den synchronen Fall sind alle Ergebnisse in einer Tabelle zusammengefasst.

Tabelle 24: Simulationsergebnisse der synchronen Implementierung

(Fläche)	Schieben (6504.79)			Addieren/Subtrahieren (9842.66)		
Simulation	Zeit (ns)	Energie (µW)	Performanz	Zeit (ns)	Energie (µW)	Performanz
Fast	8.876	1.1322	0.10	44.001	1.5701	0.69
100MHz	175.126	1.1607	2.03	185.126	1.7300	3.20
Third	82.426	2.0542	1.69	107.276	2.5642	2.75

Literaturverzeichnis

[1] M. Afghahi und J. Yuan: Double edge-triggered D-flip-flops for high-speed CMOS circuits, IEEE Journal of Solid-State Circuits, Ausgabe 26, Nummer 8, IEEE Verlag, August 1991.

[2] W. C. Athas u.a.: An Energy-Efficient CMOS Line Driver Using Adiabatic Switching AC-MOS-TR-2, Proceedings of the Fourth Great Lakes Symposium on VLSI Design, Seiten 159-164, IEEE Press, März 1994.
http://www.isi.edu/acmos/papers/94-03.GreatLakes.ps

[3] W. C. Athas u.a.: An Energy-Efficient CMOS Line Driver Using Adiabatic Switching, Technical Report USC Information Sciences Institute California, ACMOS-TR-2, Juli 1993.
http://www.isi.edu/acmos/papers/93-08.ACMOS-TR-2.ps

[4] W. C. Athas u.a.: Energy Recovery CMOS for Highly Pipelined DSP-Design, Proceedings of the International Symposium on Low-Power Electronics and Design, Monterey, August 1996.
http://www.isi.edu/acmos/papers/96-08.MontereyResclock.ps

[5] D. B. Armstrong u.a.: Design of asynchronous circuits assuming unbounded gate delays, IEEE Transactions on Computers, Seiten 1110 - 1120, 1969.

[6] U. Baake: Interaktive Methoden zur rechnergestützten Synthese asynchroner Kommunikationsschaltungen mittels einer objekt-orientierten Entwurfsumgebung, Dissertation, TH Darmstadt, August 1995.

[7] A. Bardsley und D.A. Edwards: Compiling the Language Balsa to Delay Insensitive Hardware, Proceedings of CHDL'97, Toledo, veröffentlicht in: Hardware Descriptions Languages and their Applications, IFIP & Chapman Hall, Seiten 89 - 91, 1997.

[8] A. Bardsley und D.A. Edwards: The Balsa Asynchronous Circuit Synthesis System, Proceedings of Forum on Design Languages, September 2000.

[9] K. van Berkel: Handshake Circuits - An Asynchronous Architecture for VLSI Programming, Cambrigde University Press, März 1994.

[10] E. R. Berlekamp: On Decoding Binery BCH Codes, IEEE Transactions in Information Theory, Nr. 11, Seite 393 ff, Oktober 1965.

[11] M. Borah, R. M. Owens: Accurate Estimation of Combinational Circuit Activity, 32nd ACM/IEEE Design Automation Conference, 1995.

[12] R.W. Brodersen: Low Power Digital CMOS Design, Kluwer Academic Publishers, 1996

[13] E. Brunvand: Designing self-timed systems using concurrent programs, Journal of VLSI Signal Processing, Seiten 47 - 59, Februar 1994.

[14] A. P. Chandrakasan und R. W. Brodersen: LOW POWER CMOS DIGITAL DESIGN, Kluwer Academic Publishers, Boston, 1995.

[15] A. P. Chandrakasan u.a.: Low-Power CMOS Digital Design, IEEE Journal of Solid-State Circuits, Ausgabe 27, Nr. 4, Seiten 473 - 484, April 1992.

[16] A. P. Chandrakasan und R. W. Brodersen: Minimizing Power Consumption in Digital CMOS Circuits, Proceedings of the IEEE, Ausgabe 83, Nr. 4, Seiten 498 - 523, April 1995.

[17] A.P. Chandrakasan, M.Potkonjak: Optimizing Power Using Transformations, IEEE Transaction on COMPUTER-AIDED-DESIGN, Ausgabe 14, Nr. 1, Seiten 12 - 31, Januar 1995.

[18] R. T. Chien: Cyclic Decoding Procedures for BCH Codes, IEEE Transactions on Information Theory, Ausgabe 27, Seite 254 ff., Oktober 1981.

[19] T. A. Chu: Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specifications, PhD, MIT, 1987.

[20] J. Cortadella: Petrify: A Tool for Synthesis of Petri Nets and Asynchronous Circuits, 1998. http://www.lsi.upc.es/~jordi/petrify/distrib

[21] J. C. Ebergen.: Translating Programs into Dealy-Insensitive Circuits,Center for Mathematics and Computer Science, UNI Amsterdam, 1987.

[22] K. M. Fant und S. A. Brandt: NULL Convention Logic, Theseus Logic Inc., 1997. http://www.theseus.com/

[23] G. D. Forney: On Decoding BCH Codes, IEEE Transactions on Information Theory, Ausgabe 11, Seite 393 ff., Oktober 1965.

[24] Fuhrer und Nowick: Sequential Otimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools, Kluwer Academic Publishers, 2001.

[25] S. Furber und J. Liu: Dynamic logic in four-phase micropipelines, IEE Proceedings of Async 96, IEEE Computer Society Press, March 1996.

[26] S. Furber und P. Day: Four-Phase Micropipeline Latch Control Circuits, IEEE Transactions on VLSI Systems, Ausgabe 4, Nr. 2, Seiten 247-253, June 1996.

[27] O. Hauck u.a.: VLSI System Design Using Asynchronous Wave Pipelines: A 0.35 μm CMOS 1.5 GHz Elliptic Curve Public Key Cryptosystem Chip. In 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems. Eilat, Israel. IEEE Press, 2000.

[28] O. Hauck und S. A. Huss: Asynchronous Wave Pipelines for High Throughput Datapaths, IEEE International Conference on Electronics, Circuits and Systems, Seiten 1283 - 1286, Lissabon, Portugal, September 1998.

[29] S. Hauck: Asynchronous Design Methodologies: An Overview, Proceedings of the IEEE, Volume 83, Nr. 1, Seiten 69-93, Januar 1995.

[30] U. Heinkel, M. Padeffke u.a.: The VHDL Reference, John Wiley & Sons, LTD, Chichester, 2000.

[31] C.A.R. Hoare: Communicating Sequential Processes, Prentice-Hall International, 1985.

[32] L. A. Hollaar: Direct implementation of asynchronous control units, IEEE Transactions on Computers, Ausgabe C-31, Nr.12, Seiten 1133-1141, December 1982.

[33] D. A. Huffman: The synthesis of sequential switching circuits, in E. F. Moore, Editor, Sequential Machines: Selected Papers. Addison-Wesley, 1964.

[34] IEEE Standard Hardware Description Language Based on the Verilog® Hardware Description Language, IEEE Press, New York, 1995.

[35] IEEE Standard VHDL Language Reference Manual, IEEE Press, New York, 1994.

[36] International Technology Roadmap for Semiconductors - 1999 Edition - Design http://public.itrs.net/Files/1999_SIA_Roadmap/Home.htm

[37] M. Kishinevsky u.a.: The systematic design of asznchronous circuits and why it can be useful for low power design, Technical report, Proceedings of the International Conference of Computer-Aided Design (ICCAD), 1995.

[38] J. G. Koller u.a.: Thermal Logic Circuits, Proceedings of the IEEE 1994 Workshop on Physics and Computing, Dallas, IEEE Press, 1994.
http://www.isi.edu/acmos/papers/94-10.DallasTherm.ps

[39] J. G. Koller und W. C. Athas: Adiabatic Switching, Low Energy Computing and the Physics of Storing and Erasing Information, Proceedings of the Workshop on Physics and Computation 1992, IEEE Press, 1993.
http://www.isi.edu/acmos/papers/92-10.Dallas.ps

[40] A. Kondratyev und K. Lwin, Design of Asynchronous Circuits Using Synchronous CAD Tools, Design & Test of Computers, Ausgabe 19, Nr. 4, Seiten 107 - 117, IEEE Verlag, New York, 2002.

[41] O. Kraus und M. Padeffke: Programming Environment for Manipulation of State Machines, International Workshop an Logic Synthesis, Granlibakken, CA USE, June 27-30, 1999.

[42] O. Kraus und M. Padeffke:Entwurfsumgebung für asynchrone Burst-mode Automaten, GI/ ITG/GMM Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen Tübingen, Februar 2002.

[43] O. Kraus und M. Padeffke:XBM2PLA: A flexible Synthesis Tool for Extended Burst Mode Machines, Design, Automation and Test in Europe (DATE) Conference, Seiten 1092,1093 Muenchen, 2003.

[44] O. Kraus und M. Padeffke: Synthese von asynchronen "Burst-mode" Automaten, 11. GMM/GI/ITG-Fachtagung - Entwurf Integrierter Schaltungen und Systeme, Erlangen, 2003

[45] O. Kraus, Synthese von digitalen asynchronen Zustandsautomaten, Fortschritt-Berichte VDI, Reihe 20, Nr. 372, VDI-Verlag, 2003.

[46] P. Landman: High-Level Power Estimation, International Symposium On Low Power Design, Seiten 29 - 35, August 1996.

[47] P. Landman und J.Rabaey: Architectural Power Analysis: The Dual Bit Type Method, IEEE Transactions on VLSI Systems, Ausgabe 3, Nr. 2, June 1995.

[48] P.Landman und J.Rabaey: Power Estimation for High Level Synthesis, Proceedings of EDAC-EUROASIC, Seiten 361-366, Februar 1993.

[49] P. Landman, J.Rabaey, Black-Box Capacitance Models for Architectural Power Analysis, Proceedings of the 1994 International Workshop on Low Power Design, Napa Valley, CA, Seiten 165 - 170, April 1994.

[50] LARD Documentation Home Page: Language for Asynchronous Research and Development,
http://www.cs.man.ac.uk/apt/projects/tools/lard/

[51] L. Lavagno: SIS: A System for Sequential Circuit Synthesis, Mai 1992.
ftp://ic.eecs.berkeley.edu/pub/sis/

[52] L. Lavagno und A. Sangiovanni-Vincentelli: Algorithms for Synthesis and Testing of Asynchronous Circuits, Kluwer Academic Publishers, 1993.

[53] M. Lewis u.a.: Reconfigurable Latch Controllers for Low Power Asynchronous Circuits, Proceedings of the International Symposium of Advanced Research in Asynchronous Circuits and Systems (Async 99), Seite 27 - 35 , IEEE Verlag, April 1999.

[54] M. Lighart u.a.: Asynchronous Design Using Commercial HDL Synthesis Tools, Proceedings of the International Symposium of Advanced Research in Asynchronous Circuits and Systems (ASYNC 00), Seiten 114-125, IEEE Verlag, Los Alamitos, Californien, 2000. http://www.theseus.com/

[55] J.Liu : Arithmetic and Control Components for an Asynchronous System, PhD Thesis, Dept. of Computer Science, University of Manchester, 1998. http://www.cs.man.ac.uk/apt/publications/thesis/liu98_phd.html.

[56] E. Macii und M. Poncino: Predicting the Functional Complexity of Combinational Circuits by Symbolic Spectral Analysis of Boolean Functions, EuroDac'95, IEEE Press, Seiten 294 - 299, September 1995.

[57] A. J. Martin: The Design of a Self-timed Circuit for Destributed Mutual Exclusion, Proceedings of the 1985 Chapel Hill Conference on VLSI, Seiten 245 - 260, 1985

[58] J. L. Massey: Shift Register Synthesis and BCH Decoding, IEEE Transactions in Information Theory, Nr. 15, Seite 122 ff, Oktober 1969.

[59] C. Mead und L. Conway: Introduction to VLSI systems, Addison-Wesley, London, 1980.

[60] H. Mehta u.a.s: Accurate Estimation of Combinational Circuit Activity, Proceedings of the 32nd Design Automation Conference, IEEE Press, 1995.

[61] R. Mehra und J. Rabaey: Behavioral Level Power Estimation and Exploration, Proc. First International Workshop on Low Power Design, Napa Valley, CA, Seiten 197 - 202, April 1994.

[62] T. H. Meng u.a: Automatic synthesis of asynchronous circuits from high level specifications, Transactions on Computer Aided Design, Ausgabe 8, Nr. 11, Seiten 1185 - 1205, IEEE Press, November 1989.

[63] T. H. Meng: Synchronization Design for Digital Systems, Kluwer Academic Publishers, 1990.

[64] C. E. Molnar u.a.: Synthesis of Delay-Insensitive Modules, Proceedings of the 1985 Chapel Hill Conference on Advanced Research in VLSI, Seiten 67 - 86, 1985.

[65] D. E. Muller und W. S. Bartky: A theory of asynchronous circuits, Proceedings of an International Symposium of Theory of Switching, Seiten 204-243, Harvard University Press, 1959.

[66] T. Murata: Petri Nets: properties, analysis, and applications. In Proceedings of the IEEE, Ausgabe 77, Nr. 4, Seiten 541 - 580, IEEE Press, 1989.

[67] C. J. Myers: Compuer-Aided Synthesis and Verification of Gate-Level Timed Circuits, PhD Thesis, Stanford University, 1995. http://www.async.elen.utah.edu/publications.html

[68] F. N. Najm: A Survey of POWER ESTIMATION TECHNIQUES in VLSI-Design, IEEE Transactions on VLSI-System, Ausgabe 2, Nr. 4, Dezember 1994. http://www.eecg.toronto.edu/~najm/papers/tvlsi94-survey.ps

[69] S. M. Nowick and B. Coates: Automated design of high-performance asynchronous state machines, in Proceedings of the International Conference of Computer Design (ICCD), Seiten 434 - 441, IEEE Press, Oktober 1994.

[70] S. M. Nowick and David L. Dill: Automatic synthesis of locally-clocked asynchronous state machines, in Proceedings of the International Conference of Computer-Aided Design (ICCAD), Seiten 318-321. IEEE Press, November 1991.

[71] S. M. Nowick and David L. Dill: Synthesis of asynchronous state machines using a local clock, in Proceedings of the International Conference of Computer Design (ICCD), Seiten 192-197. IEEE Press, Oktober 1991.

[72] M. Pedram: Power Minimization in IC-Design: Principles and Applications, Transactions on Design Automation of Electronic systems, Ausgabe 1, Nr. 1, Seiten 3 - 56, 1996.

[73] C. A. Petri: Kommunikation mit Automaten, Schriften des Institut für Instrumentelle Mathematik Nr. 2 , Bonn, 1962.

[74] C. A. Petri: Fundamentals of a Theory of Asynchronous Information Flow, in Proceedings of IFIP Congress 62, Seiten 386 - 390. North Holland Publication Company, Amsterdam , 1963.

[75] F. Prosser u.a.: A comparison of modular self-timed design styles, Internes Papier der Universität Utah, 1995.

[76] I. S. Reed und G. Solomon: Polynomial Codes over Certain Finite Fields, Journal of the Society for Industrial and Applied Mathematics, Seiten 300 - 304, Juni 1960.

[77] J. Rabaey und M. Pedram: Low Power Design Methodologies, Kluwer Academic Publishers, Boston, 1996.

[78] O. Roig: Versify: Verification of Speed-Independent Circuits, November 1998.
http://www.ac.upc.es/recerca/VLSI/versify

[79] F. U. Rosenberger u.a.: Q-Moduls: Internally Clocked Delay-Insensitive Modules, Transactions on Computers, Ausgabe 37, Nr. 9, Seiten 1005 - 1018, IEEE Press, 1988.

[80] L. Rosenblum und A. Yakovlev: Signal Graphs: from self-timed to timed ones, International Workshop on Timed Petri Nets, Torino, Seiten 199 - 206 1985.

[81] Shai Rotem u.a.: RAPPID: An Asynchronous Instruction Length Decoder, The Fifth International Symposium on Advanced Research in Asynchronous Circuits and Systems, April 1999.
http://www.async.elen.utah.edu/publications.html

[82] C. L. Seitz: Hot-Clock nMos, Proceedings of the 1985 Chapel Hill Conference on VLSI, Seiten 1 - 17, 1985.

[83] P. Siegel u.a.: Saving Power by Synthesizing Gated Clocks for sequential Circuits, IEEE Design & Test of Computers, IEEE Press, 1994.

[84] D. Singh: Power Conscious CAD Tools and Methodologies: A Perspective, Proceedings of the IEEE, Ausgabe 83, Nr. 4, Seiten 570 - 594, April 1995.

[85] R. Smith and M. Ligthart: Asynchronous logic design with commercial High Level Design tools, März 2001.
http://www.theseus.com/

[86] J. Sparsø u.a.: Design of Self-timed Multipliers: A Comparison, Technical University of Denmark, Department of Computer Science Technical Report, 1992.

[87] J. Sparsø und J. Staunstrup: Delay-insensitive multi-ring structures, Journal of Integration, Ausgabe 15, Nr. 3, seiten 313 - 340, Oktober 1993.

[88] N. A. Starodoubtsev u.a.: Use of VHDL Environment for Interactive Sythesis of Asynchronous Circuits, Technischer Report 540, Seiten 1 - 12, Department of Computing Science, University of Newcastle upon Tyne, 1995.

[89] I.E. Sutherland: Micropipelines, Communications of the ACM, Ausgabe 32, Nr.6, Seiten 720-738, Juni 1989.

[90] I. Sutherland und L. Lexau: Designing Fast Asynchronous Circuits, Proceedings of the 7'th International Symposium of Advanced Research in Asynchronous Circuits and Systems (ASYNC 01), Seiten 184-193, IEEE Verlag, Los Alamitos, Californien, 2001.

[91] L. J. Svensson und J. G. Koller: Adiabatic charging without inductors, Technical Report USC Information Sciences Institute California, ACMOS-TR-3a, Februar 1994. http://www.isi.edu/acmos/papers/94-02.ACMOS-TR-3a.ps

[92] L. J. Svensson und J. G. Koller: Driving a capacitive load without dissipating fCV2, Proceedings of the IEEE 1994 Symposium on Low-Power Electronics, IEEE Press, Oktober 1994.
http://www.isi.edu/acmos/papers/94-10.SanDiegoStepwise.ps

[93] L. J. Svensson u.a.: A sub-CV2 driver with 10 ns transition time, Proceedings of the International Symposium on Low-Power Electronics and Design, Monterey, CA, August 1996. http://www.isi.edu/acmos/papers/96-08.MontereyStepwise.ps

[94] Synopsys Inc.: http://europe.synopsys.com/

[95] S.Y. Tan u.a.: The Design of an Asynchronous VHDL Synthesizer, Proceedings of the 35th ACM/IEEE Design Automation Conference, Paris, February 1998

[96] J. H. Tracey: Internal state assignment for asynchronous sequential machines, IEEE transactions on electronic computers, Volume 5, Nr. 4, Seiten 551 - 560, August 1966.

[97] G. K. Theodoropoulos u.a.: Occam: An Asynchronous Hardware Description Language?, Proceedings of the 23rd IEEE Euromicro Conference on New Frontiers of Information Technology, Budapest, Hungary, Seiten 249 - 256, IEEE Press, September 1997.

[98] N. Tzartzanis, W. C. Athas: Energy Recovery for the Design of High-Speed, Low-Power Static RAMs, Proceedings of the International Symposium on Low-Power Electronics and Design, Monterey, CA, Seiten 55 - 60, August 1996.

[99] S. H. Unger: Asynchronous Sequential Switching Circuits, John Wiley & Sons, Inc., New York, 1969.

[100]Various publications form Theseus Logic Inc. anout NULL Convention Logic: http://www.theseus.com/

[101]V. Varshavsky u.a.: Totally self-checking asynchronous combinational circuits and the indication property, Automation and Remote Control, 1982.

[102]S. B. Wicker, V. K. Bhargava: Reed-Solomon Codes and Their Applications, John Wiley & Sons, LTD, Chichester, September 1999.

[103]K. Y. Yun u.a.: Synthesis of 3D Asynchronous State Machines, Proceedings of the International Conference on Computer Design (ICCD), Seiten 346-350, 1992.

[104]K. Y.Yun: Synthesis of asynchronous controllers for heterogeneous systems, Dissertation Standford University, 1994.

[105]K. Y. Yun u.a.: High-Performance Asynchronous Pipeline Circuits, Proceedings of the International Symposium of Advanced Research in Asynchronous Circuits and Systems (ASYNC 96), Seiten 17 - 28, IEEE Verlag, Los Alamitos, Californien, 1996.

[106]K. Y. Yun u.a.: High-Performance Two-Phase Micropipeline Building Blocks: Double Edge Triggered Latches and Burst-Mode Select abd Toggle Circuits, IEEEProceedings of Circuits Devices Systems, Ausgabe 143, Seiten 282 - 288, October 1996.
http://jungfrau.usc.edu/pub/det.ps

[107]C. Ykman u.a.: ASSASSIN: A Synthesis System for Asynchronous Control Circuits, 1995.
ftp://imec.be/pub/vsdm/SW_distrib/assassin/

[108]H. Zheng: Specification and Compilation of Timed Systems, MS Thesis, University of Utah, 1998.
http://www.async.elen.utah.edu/publications.html

Lebenslauf

05.11.1968	geboren in Karlsruhe
1975 - 1977	Grundschule in Karlsruhe
1977 - 1979	Grundschule in Passau
1979 - 1988	Auersperg-Gymnasium Passau - Freudenhain
1988 - 1989	Grundwehrdienst
1989 - 1995	Studium der Elektrotechnik an der Universität Erlangen-Nürnberg
1995 - 2004	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Rechnergestützten Schaltungsentwurf an der Universität Erlangen-Nürnberg
Seit 2004	Mitarbeiter bei IBM Deutschland GmbH