Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thorben J. Wübbenhorst

Synthesegerechte Konvertierung synchroner in asynchrone Schaltungen für FPGAs

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thorben J. Wübbenhorst

Synthesegerechte Konvertierung synchroner in asynchrone Schaltungen für FPGAs

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Canzler
Zweitgutachter: Prof. Dr. Schäfers

Eingereicht am: 14. November 2014

Thorben J. Wübbenhorst

Thema der Arbeit

Synthesegerechte Konvertierung synchroner in asynchrone Schaltungen für FPGAs

Stichworte

Asynchrone Schaltung, Bundled-Data, FPGA, STG

Kurzzusammenfassung

Asynchrone Schaltungen haben Vorteile im Bereich der EMV und des Energieverbrauchs. Die vorliegende Arbeit beschreibt einen Ansatz, synchrone in asynchrone Schaltungen zu konvertieren und für einen FPGA zu synthetisieren. Es wird gezeigt wie spezielle Bauelemente für asynchrone Schaltungen auf FPGAs realisiert werden können, dabei wird insbesondere auch die Möglichkeit diskutiert, die üblichen Delay Chains durch „Delay Shift"-Elemente zu ersetzen. Anschließend wird ein Konvertierungsablauf für VHDL beschrieben und ein Beispiel vorgestellt. Im Ergebnis wird die Funktionsfähigkeit des vorgestellten Konvertierungsansatzes demonstriert; es zeigt sich dabei, dass die Performanceerwartungen nur bedingt erfüllt werden. Sämtliche Software die verwendet wurde ist frei verfügbar.

Thorben J. Wübbenhorst

Title of the paper

Conversion of synchronous in asynchronus circuits synthesizable for FPGAs

Keywords

Asynchronous circuit, Bundled-Data, FPGA, STG

Abstract

Asynchronous circuits have advantages in the field of EMC and energy consumption. The present thesis describes how to convert a synchronous into an asynchronous circuit and how to synthesize it for an FPGA. It is shown how specific components for asynchronous circuits can be realized on FPGAs; in the process, it is specifically discussed to replace the usual Delay Chains by "Delay Shift" components. The conversion sequence is described for VHDL and an example is presented. It is demonstrated that the conversion approach is operational, but that prior performance expectations are only partly fulfilled. All the software used is free.

Inhaltsverzeichnis

Inhaltsverzeichnis v

1. Einführung 1
1.1. Synchrone und asynchrone Schaltungen 1
1.2. Vorteile asynchroner Schaltungen 3
1.3. Nachteile asynchroner Schaltungen 4
1.4. Bisheriger Konvertierungsansatz 5
1.5. Ziel der Arbeit 5

2. Grundkonzepte asynchroner Schaltungen 7
2.1. Handshake Protokolle 7
2.2. Muller C-Element 10
2.3. Pipelinestrukturen 10
2.3.1. Muller Pipeline 10
2.3.2. 4 Phasen Bundled-Data Pipeline 11
2.3.3. Micropipelines 12
2.3.4. Entkopplungsgrad 15
2.3.5. Fork und Join Elemente 16
2.4. Signal Transition Graphs 17
2.4.1. Petrinetze 18
2.4.2. STGs 18
2.4.3. STG Synthese 21
2.4.4. STG Synthese mit Petrify 22

3. Implementierung asynchroner Schaltungen auf FPGAs 27
3.1. FPGAs 27
3.2. Muller C-Element 28
3.2.1. Muller C-Element mit 2 Eingängen 28
3.2.2. Muller C-Element mit 3 Eingängen 31
3.2.3. Muller C-Element mit 4 Eingängen 33
3.3. Delay Chain 34
3.4. Delay Shift 36
3.5. Pipelines 39
3.5.1. Nicht entkoppelte Pipeline 39
3.5.2. Entkoppelte Pipeline 40

3.5.3. Doubly-Latched Pipeline . . . 43
3.5.4. Entkoppeln von Abhängigkeiten . . . 46

4. VHDL Konvertierung asynchroner Schaltungen 49
4.1. Konvertierungsablauf . . . 49
4.1.1. Kontrollpfad . . . 49
4.1.2. Datenpfad . . . 50
4.1.3. Delay Matching . . . 52
4.2. MIPS Prozessor . . . 52
4.2.1. Ressourcenverbrauch . . . 55
4.2.2. Performancevergleich . . . 55
4.2.3. Bewertung . . . 55

5. Fazit 57
5.1. Zusammenfassung . . . 57
5.2. Resümee . . . 58
5.3. Ausblick . . . 58

Anhang 59

A. Literatur 61

B. Abkürzungsverzeichnis 65

C. Glossar 67

D. Glossar verwendeter Programme 69

E. Abbildungsverzeichnis 71

F. Quellcodecode 73
F.1. Grundlegende Elemente asynchroner Systeme . . . 75
F.1.1. Muller C-Element . . . 75
F.1.2. Muller C-Element mit 3 Eingängen . . . 78
F.1.3. Muller C-Element mit 4 Eingängen . . . 79
F.1.4. Delayelemente . . . 81
F.2. Technologiebibliothek . . . 84
F.3. Registercontroller . . . 85
F.3.1. Nicht entkoppelter Registercontroller . . . 85
F.3.2. Entkoppelter Registercontroller . . . 87
F.3.3. Registercontroller für eine Doubly-Latched Asynchronous Pipeline (DLAP) . . . 89
F.3.4. Join für DLAP . . . 91

1. Einführung

Die Unterscheidung von synchronen und asynchronen Schaltungsentwürfen erfolgte bereits, als die ersten digitalen Schaltungen in den 1950er Jahren implementiert wurden. Der weitaus überwiegende Teil digitaler Schaltungen ist heutzutage synchron aufgebaut. Asynchrone Schaltungen hingegen finden nur bei sehr speziellen Problemstellungen Anwendung. Beispiele hierfür sind Schaltungen, die eine gute elektromagnetische Verträglichkeit (EMV) voraussetzen oder nur einen sehr geringen Energieverbrauch haben dürfen.
Synchrone Schaltungen sind für die unterschiedlichsten Problemstellungen häufig schon implementiert und auch funktional getestet. Damit diese Implementierungen Anforderungen im Bezug auf EMV und Energieverbrauch erfüllen, bietet es sich an, eine Konvertierung in eine asynchrone Schaltung vorzunehmen. Die Schaltung muss dadurch nicht ein weiteres mal implementiert werden, sondern wird lediglich angepasst.

1.1. Synchrone und asynchrone Schaltungen

Abbildung 1.1 zeigt eine einfache synchrone Schaltung. Es wird unterschieden zwischen den speichernden Elementen und der dazwischen liegenden kombinatorischen Logik.

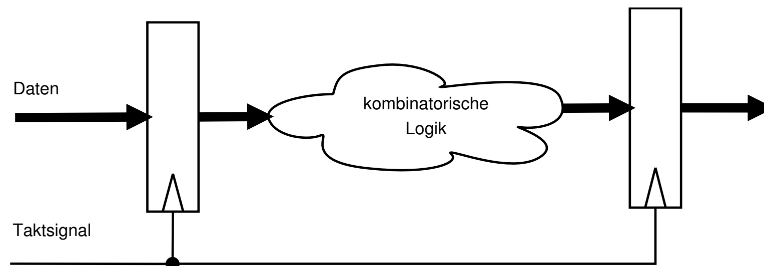


Abbildung 1.1.: Synchroner Datentransfer

Bei einem Signalwechsel der Eingänge einer kombinatorischen Schaltung kann kurzzeitig der Ausgang dieser Schaltung einen falschen Wert besitzen. Dieses Verhalten wird verursacht aufgrund unterschiedlicher Signallaufzeiten in den Pfaden einer Schaltung und ist bekannt als

„Glitch“. Es ist mit einem hohen Aufwand verbunden Glitches zu beseitigen. Die Auswirkungen können jedoch verhindert werden, indem flankengesteuerte D-Flipflops hinter die logische Schaltung platziert werden [Wak05].
Das zentrale Prinzip synchroner Schaltungen besteht darin, dass die Werte der kombinatorischen Logik immer dann von den Flipflops übernommen werden, wenn ein sicheres Datum anliegt, also keine Glitches mehr auftreten können. Die maximale Laufzeit der kombinatorischen Logik, bestimmt die minimale Periode in der eine Taktflanke an den Flipflops auftreten darf. Die Periode ist abgebildet in dem Taktsignal.
Aus den Flipflops können Register verschiedener Wortbreiten zusammengesetzt werden, um beispielsweise das Ergebnis einer arithmetischen Operation zwischenzuspeichern.
Die Flipflops einer synchronen Schaltung übernehmen ihre Daten in der Regel mit der steigenden Flanke des Taktsignals. Das geschieht bei allen Flipflops zum gleichen Zeitpunkt. Die Taktgeschwindigkeit einer synchronen Schaltung richtet sich daher an den Pfad, der am längsten braucht um ein korrekten Ausgangswert zu produzieren.[1]
Asynchrone Schaltungen werden im Gegensatz zu synchronen Schaltungen lokal gesteuert. Das heißt, dass ein lokal erzeugtes Signal die Übernahme der gültigen Daten in die speichernden Elemente steuert und kein globales Taktsignal.

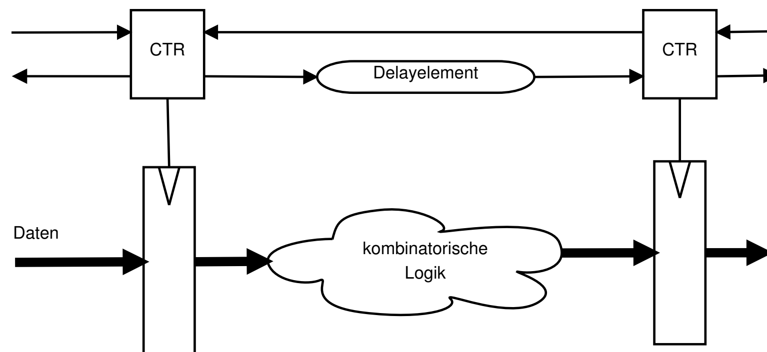


Abbildung 1.2.: Asynchroner Datentransfer

Abbildung 1.2 zeigt das asynchrone Äquivalent der Schaltung aus Abbildung 1.1.
Die Kontrolleinheiten erzeugen für jedes Register ein Steuersignal. Der korrekte Zeitpunkt zu dem ein Steuersignal sich ändert wird durch die Kommunikation zwischen den Kontrolleinheiten bestimmt. Die Auswirkungen von Glitches werden bei asynchronen Systemen verhindert, indem in der Kommunikation der Kontrolleinheiten Delayelemente eingefügt werden. Diese Delayelemente sind an die Laufzeit der kombinatorische Logik angepasst.

[1]Dieser langsamste Pfad wird auch “worst-case” genannt

1.2. Vorteile asynchroner Schaltungen

Keine Clock Skew Probleme

Gordon Moore hat 1965 und 1975 in seinen Aufsätzen [Moo65] [Moo75] vorausgesagt, dass sich die minimale Strukturgöße integrierter Schaltungen über die Zeit exponentiell verkleinern wird. Es können daher immer mehr Transistoren auf einem Halbleiterchip untergebracht werden. Mit der Verkleinerung der Strukturgröße und der damit einhergehenden Erhöhung der maximalen Schaltfrequenz wird es immer schwieriger, das Taktsignal bei synchronen Schaltungen so zu verteilen, dass überall gleichzeitig die Taktflanke anliegt. Die Zeitdifferenz zwischen dem Eintreffen einer Taktflanke an zwei Flipflops wird als Taktversatz oder „Clock Skew" bezeichnet. Um zu vermeiden, dass Probleme durch Clock Skews entstehen, werden immer größere Taktnetzwerke notwendig [Fri01]. Asynchrone Schaltungskonzepte bieten eine Lösung, weil anstelle eines globalen Taktes lokal erzeugte Signale einen korrekten Datenfluss gewährleisten.

Globale Zeitaspekte

Bei synchronen Schaltungen ist eine bestimmte Taktrate vorgegeben, die sich nach dem langsamsten Pfad richtet. Innerhalb der synchronen Schaltung müssen alle Pfade so optimiert sein, dass die Daten für die Taktflanke sicher anliegen. In asynchronen Schaltungen müssen weniger Pfade optimiert werden, weil hier die Laufzeiten der Pfade zwischen den Speicherelementen individuell abgebildet werden.

Geschwindigkeit

Die logischen Gatter einer integrierten Schaltung werden durch die Verkleinerung der Strukturgröße immer schneller. Die Laufzeiten der Leitungen werden hingegen anteilig zur Gesamtlaufzeit immer länger. Dadurch muss bei dem Entwurf synchroner Schaltungen den Leitungen heute mehr Aufmerksamkeit gewidmet werden. Die Länge zwischen zwei Elementen sollte also so kurz wie möglich sein. Die Algorithmen für die Platzierung der Bauelemente werden immer ausgefeilter[US13]. Bei globalen Signalen sind diese Algorithmen aber nutzlos, weil die Signale auf der gesamten Schaltung verfügbar sein müssen. Bei einer synchronen Schaltung bestimmt vor allem die Signallaufzeit der Taktleitung den maximal möglichen Datendurchsatz. Asynchrone Schaltungen hingegen besitzen keine globalen Signale und haben dadurch das Potenzial eine größere durchschnittliche Performance zu erreichen.

Verlustleistung

Bei einer synchronen CMOS Schaltung werden mit jedem Schaltvorgang Kapazitäten geladen und entladen. Die Umladungsenergie dieses Vorgangs geht dabei als Wärme verloren und stellt den größte Anteil der Verlustleistung. Eine weitere Ursache für Verluste bei Schaltvorgängen ist auf direkte Kurzschlussströme zurückzuführen. Diese treten auf, wenn der PMOS und NMOS Transistor gleichzeitig leitet [MC80, S. 340].
Das Taktsignal einer synchronen Schaltung verursacht permanent Schaltvorgänge, durch die Energie verbraucht wird. Das geschieht auch wenn die Schaltung im Leerlauf ist, also die Daten der Schaltung sich nicht ändern. Asynchrone Schaltungen können so entwickelt werden, dass nur dann Schaltvorgänge stattfinden, wenn es nötig ist. Dadurch beschränkt sich der Energieverbrauch asynchroner Schaltungen im Leerlauf auf die Leckverluste der gesperrten Dioden und Transistoren.

EMV

In synchronen Schaltungen wird mit jeder Taktflanke auf dem gesamten Chip eine neue Berechnung begonnen. Das führt dazu, dass der Chip zum Zeitpunkt der Taktflanke besonders viel Energie verbraucht und das elektromagnetische Rauschen dadurch auf diesen Zeitpunkt konzentriert ist. Bei asynchronen Schaltungen ist das elektromagnetische Rauschen besser verteilt, weil nicht alle Berechnungen gleichzeitig beginnen. Ein Beispiel für eine Verbesserung der EMV durch asynchronen Schaltungsentwurf ist der Amulet 2e Prozessor [Fur+97].

1.3. Nachteile asynchroner Schaltungen

Die aufgeführten Vorteile asynchroner Schaltungen erwecken den Eindruck, dass sie die Lösung für die Probleme beim heutigen Entwurf integrierter Schaltungen sind. Bei der Implementierung asynchroner Schaltungen haben sich jedoch auch einige Nachteile ergeben.

Delay Matching

Bei synchronen Schaltungen ist die Laufzeit des langsamsten Pfades kleiner als die Taktgeschwindigkeit. Damit ist weitestgehend sichergestellt, dass die Schaltung ein korrektes Verhalten hat. Es ist deutlich aufwendiger ein korrektes Verhalten bei asynchronen Schaltungen zu gewährleisten, da die Laufzeit jedes Pfades individuell in Delayelemente abgebildet wird.

Fläche

Asynchrone Schaltungen sind größer als ihre entsprechende synchrone Implementierung. Das liegt an den zusätzlichen Schaltungselementen wie Delayelemente und Kontrolleinheiten.

Performance

Einige der bisherigen asynchronen Realisierungen haben die Erkenntnis gebracht, dass die Performance im Vergleich zu dem synchronen Gegenstück geringer [Pav94; Fur+97] oder gleich ist [FGG98]. Trotz des Mehraufwandes für den Entwurf einer asynchronen Schaltung erhält man also nicht zwingend eine bessere Performance.

1.4. Bisheriger Konvertierungsansatz

Das in [FEH10] vorgestellte Verfahren konvertiert eine synchrone Schaltung in eine asynchrone auf Netzlistenebene für Field Programmable Gate Arrays (FPGAs). Netzlisten beschreiben die Verbindungen zwischen Bauelementen[2] in Textform. Das übliche Format für Netzlisten ist das Electronic Design Interchange Format (EDIF)[Edi]. Das Verfahren wendet einen Suchalgorithmus auf eine EDIF Netzliste an, um zusammengehörige Flipflops eines Registers zu identifizieren. Die Taktleitung dieser Flipflops wird durch Signale ersetzt, die mit Kontrolleinheiten erzeugt werden. Als nächstes werden die Kontrolleinheiten miteinander verbunden und bilden dadurch einen asynchronen Kontrollpfad für die Register. Das Verfahren benötigt Software die ein Hardware Description Language (HDL) Design in das EDIF synthetisiert. Die Standardsynthesesoftware der Marktführer für FPGAs leistet das nicht. Es ist zwar möglich Designs die im EDIF vorliegen auf einen FPGA zu platzieren; bei der HDL Synthese jedoch wird ein herstellerspezifisches Format verwendet[Xil; Alt].

1.5. Ziel der Arbeit

Ziel dieser Arbeit ist es, eine Konvertierung für die HDL Ebene zu beschreiben. Das Resultat soll synthetisiert und auf einen FPGA geladen werden können. Die allgemeinen Vorteile eines HDL Entwurfs gelten auch für die Konvertierung auf HDL Ebene. So ist eine gewisse Technologieunabhängigkeit gegeben. Ein Entwurf ist somit übertragbar auf andere Bausteinfamilien oder sogar auf Bausteine anderer Hersteller. Das führt dazu, dass ein Entwurf für zukünftige oder andere Technologien wiederverwendbar wird. Des Weiteren kann ein konvertiertes

[2]Bauelemente sind zum Beispiel Logikgatter oder speichernde Elemente wie Flipflops.

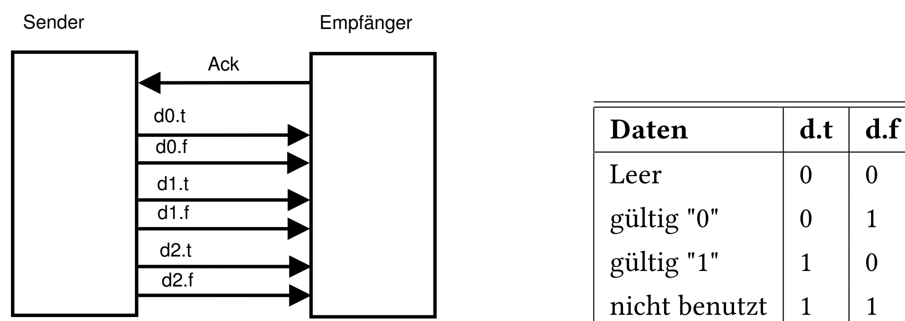
Design mit kostenloser[3] Standardsoftware synthetisiert werden.
Asynchrone Schaltungen benötigen Komponenten die nicht in synchronen Entwicklungsumgebungen enthalten sind. Damit eine synthesegerechte Konvertierung erfolgen kann, müssen zunächst spezielle Komponenten für die Zieltechnologie entwickelt werden. In dieser Arbeit ist die Zieltechnologie ein FPGA. FPGAs wurden für synchrone Entwürfe entwickelt. Die Komponenten für asynchrone Schaltungen werden auf FPGAs in dem Bereich implementiert, der sonst für die kombinatorische Logik verwendet wird. Die Synthesesoftware für FPGAs ist nicht in der Lage, die Komponenten für asynchrone Schaltungen korrekt zu synthetisieren. Daher müssen alle asynchronen Komponenten manuell implementiert werden, indem die Bauelemente des FPGA mit Hilfe einer dafür vorgesehenen HDL Bibliothek konfiguriert werden. In dieser Arbeit werden sämtliche Komponenten vorgestellt die nötig sind um auf einem FPGA asynchrone Schaltungen zu realisieren.

[3]Xilinx und Altera bieten für ihre FPGAs kostenlose Synthesesoftware an.

2. Grundkonzepte asynchroner Schaltungen

2.1. Handshake Protokolle

Die Kommunikation zwischen zwei benachbarten Modulen wird durch Handshake-Protokolle abgewickelt. Es gibt zwei Hauptprotokolle: Dual-Rail und Bundled-Data. Das Dual-Rail Protokoll arbeitet mit zwei Leitungen für jedes Bit. Die Redundanz ermöglicht es mit drei der vier Signalzuständen die Synchronisation vorzunehmen. Ein gültiges Datum wird übertragen, wenn die Leitungen unterschiedliche Werte haben. Sind beide Signale '0', ist die Berechnung des neuen Datums noch nicht abgeschlossen. Abbildung 2.1 zeigt die Verbindungen für die Übertragung von drei Bits mit dem Dual-Rail Protokoll. In der Tabelle daneben, sind die Zustände der beiden Datenleitungen für ein Bit aufgelistet.



Daten	d.t	d.f
Leer	0	0
gültig "0"	0	1
gültig "1"	1	0
nicht benutzt	1	1

Abbildung 2.1.: Struktur eines Datenpfades mit dem Dual-Rail Protokoll

Das Dual-Rail Protokoll hat vier Phasen:

1. Der Sender gibt ein gültiges Datum
2. Der Empfänger übernimmt das Datum und setzt ACK auf '1'
3. Der Sender antwortet indem er alle Datenleitungen auf '0' setzt

4. Der Empfänger setzt ACK auf '0'

Aufgrund der zwei Leitungen für jedes Bit, erfordern Schaltungen mit dem Dual-Rail Protokoll einen sehr hohen Verdrahtungsaufwand. Darüber hinaus sind für die Implementierung von Dual-Rail Logikgattern sehr viele Transistoren erforderlich: Für ein einfaches CMOS AND2 Gatter sind 6 Transistoren nötig, im Vergleich dazu sind für das Dual-Rail Pendant 30 Transistoren erforderlich [SF01, pp 67]. Schaltungen mit dem Dual-Rail Protokoll sind sehr robust gegenüber äußeren Einflüssen wie Temperaturänderungen oder Fluktuationen in der Spannungsversorgung.
Ein weiteres und häufig eingesetztes Protokoll ist das Bundled Data Protokoll. Es teilt die Steuer- und die Dateninformationen der Schaltungsmodule. Die Steuersignale bestehen aus einem Request (REQ), um ein Datum anzufordern und einem Acknowledge (ACK), um den Empfang eines Datums zu bestätigen. Parallel zu den Steuersignalen überträgt ein Datenbus die Operanden.
Schaltungen mit dem Bundled-Data Protokoll haben einen ähnlichen Aufbau wie getaktete Datenpfade. Anstelle eines globalen Taktes wird ein lokales Signal erzeugt und zur Synchronisation verwendet. Abbildung 2.2 zeigt die Verbindung zwischen Sender und Empfänger einer Bundled Data Struktur nach Seitz [MC80].

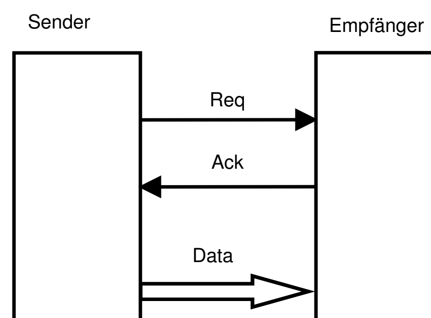


Abbildung 2.2.: Bundled Data nach Seitz

Seitz beschreibt zwei Protokolle für seine Bundled Data Struktur: die „Two-Phase Bundled Data Convention“ (Abbildung 2.3) und die „Four-Phase Bundled Data Convention“ (Abbildung 2.4). Bei der Two Phase Bundled-Data Convention ist die Synchronisierung zwischen zwei Schaltungsmodulen ereignisgesteuert. Das heißt, dass die Signalwechsel anstelle der Signalpegel zur Synchronisierung verwendet werden. Darauf aufbauend beschreibt Sutherland die Micropipelines als eine Art elastische Pipelinestruktur [Sut89].
Die Signalverarbeitung in integrierten Schaltungen war damals aber in aller Regel pegelgesteuert und nicht ereignisgesteuert; um dem Rechnung zu tragen, hat Seitz das Four-Phase

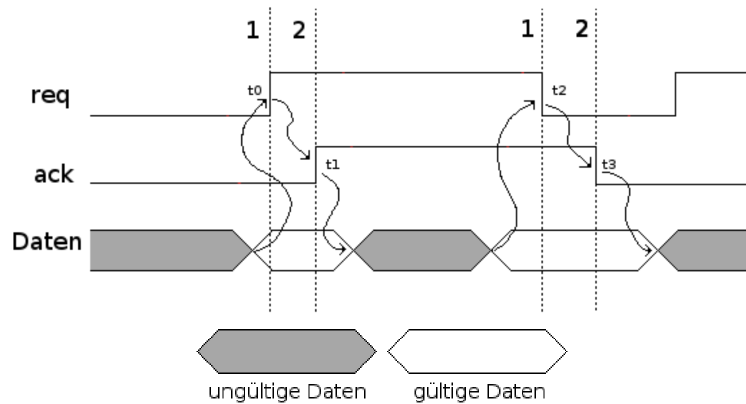


Abbildung 2.3.: Two-Phase Bundled Data Convention

Bundled Data Protokoll vorgeschlagen. Bei dieser Konvention wird die pegelgesteuerte Synchronisation durch zwei zusätzliche Signalwechsel erkauft. Die Request und Acknowledge Signale erreichen ihren Initialwert, bevor das nächste Datum verarbeitet wird. Dies vereinfacht die Signaldekodierung in den Schaltungsmodulen [MC80].
Bei der Konvertierung synchroner Schaltkreise in asynchrone Schaltkreise auf Basis des Bund-

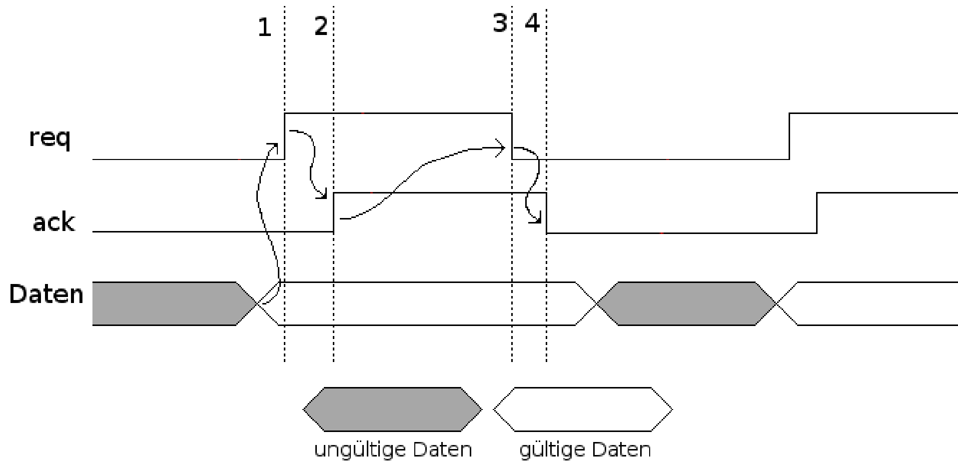


Abbildung 2.4.: Four-Phase Bundled Data Convention

led Data Protokolls muss die kombinatorische Logik nicht verändert werden. Dagegen ist die Konvertierung für das Dual-Rail Protokoll deutlich aufwendiger, weil sämtliche Logik berücksichtigt werden muss. Zudem ist der Ressourcenverbrauch für Schaltungen mit dem

Dual-Rail Protokoll sehr hoch [SF01, S. 23]. Bei der Implementierung auf einen FPGA würde man bei größeren Entwürfen schnell an Kapazitätsgrenzen stoßen. Aus diesen Gründen wird in dieser Arbeit das Bundled-Data Protokoll verwendet.

2.2. Muller C-Element

Um hazardfreie asynchrone Schaltungen entwickeln zu können, wird eine neue Komponente benötigt: das Muller C-Element. Das Muller C-Element ist eine zustandspeichernde Komponente und ist essentiell für den Aufbau von Schaltungen, die ein lokales Taktsignal für flankengesteuerte Flipflops erzeugen. Sein Ausgang wird auf 0 gesetzt, wenn alle Eingänge 0 sind und wird auf 1 gesetzt, wenn alle Eingänge 1 sind. Haben die Eingänge einen unterschiedlich Wert, behält das C-Element seinen vorherigen Ausgangswert [MB59]. Tabelle 2.1 stellt alle Zustände des Muller C-Elements dar. Abbildung 2.5 zeigt das Symbol für das Logikgatter des Muller C-Elements.

A	B	Y	Erklärung
0	0	0	Der Wechsel auf 0
0	1	Halten	Der Ausgangszustand wird gehalten
1	0	Halten	Der Ausgangszustand wird gehalten
1	1	1	Der Wechsel auf 1

Tabelle 2.1.: Muller C-Element Funktionstabelle

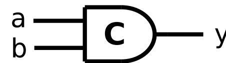


Abbildung 2.5.: Muller C-Element

2.3. Pipelinestrukturen

2.3.1. Muller Pipeline

Die Pipeline in Abbildung 2.6 realisiert einen Mechanismus, der Handshakes weiterleitet. Sie ist eine Schaltung, die aus Invertern und C-Elementen besteht und ist das Grundgerüst für die meisten asynchronen Pipelines; sie ist bekannt als Muller Pipeline oder auch als Muller Distributer [Mul63]. Wenn alle C-Elemente mit '0' initialisiert sind, wird das Handshaking gestartet, indem der linke REQ Eingang auf '1' gesetzt wird. Um zu verdeutlichen wie diese Pipeline funktioniert, bietet es sich an, das mittlere C-Element C[i] zu betrachten. C[i] übernimmt

nur dann eine ’1’ von seinem Vorgänger C[i-1], wenn der Ausgang von seinem Nachfolger C[i+1] ’0’ ist. Auf ähnliche Weise übernimmt C[i] nur eine ’0’ von C[i-1], wenn der Ausgang von C[i+1] ’1’ ist. Gibt der rechte ACK Eingang keine Rückmeldung, ist die Pipeline voll und

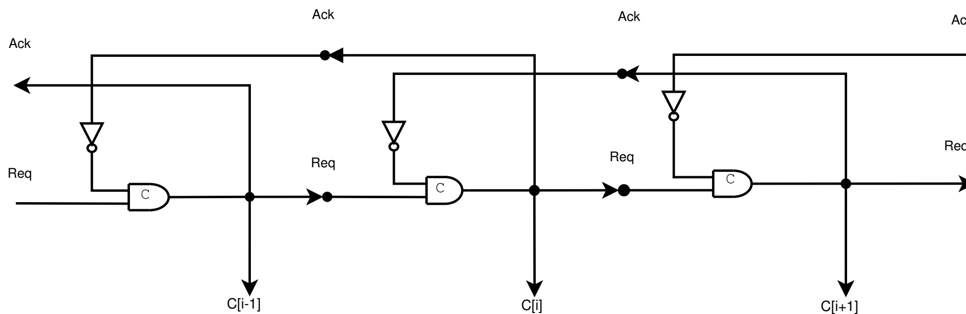


Abbildung 2.6.: Muller Pipeline[SF01]

stoppt das Handshaking. Unabhängig von den Laufzeiten der C-Elemente der Inverter oder der Leitungen arbeitet diese Schaltung hazardfrei.[1]

2.3.2. 4 Phasen Bundled-Data Pipeline

Die Organisation der zu verarbeitenden Logik im Datenpfad ist bei Schaltungen mit einem Bundled-Data Protokoll ähnlich dem eines synchronen Schaltungsentwurfs.[2] Die Register werden gesteuert durch Signale, die mit einer Muller Pipeline erzeugt werden. Diese Signale werden auch als lokale Taktsignale bezeichnet [SF01]. Eine Pipelinestufe ist mit ihrem Nachbarn so verzahnt, dass die lokalen Taktsignale einen kontrollierten Signalfluss des Datenpfades gewährleisten. Dadurch ist es möglich, pegelgesteuerte Register anstelle von flankengesteuerten Registern zu verwenden [Kom+88]. Pegelgesteuerte Register werden aus D-Latches aufgebaut. D-Latches speichern für gewöhnlich ihren Eingangswert wenn das Steuersignal ’0’ ist und sind transparent wenn das Steuersignal ’1’ ist [Wak05]. Die Latches der Register einer 4-Phasen Bundled-Data Pipeline reagieren auf das Steuersignal invertiert. Das Register ist transparent, wenn das Steuersignal ’0’ ist und es speichert seine Daten wenn das Steuersignal ’1’ ist.
Die Verzögerung der kombinatorischen Logik zwischen zwei Pipelinestufen muss in “Delay-Elementen” abgebildet werden. Delay-Elemente verzögern ein Signal in definierter Länge. Das Delay-Element muss so abgestimmt werden, dass ein gültiges Datum am Register anliegt. Die Verzögerung entspricht also der worst-case Laufzeit der kombinatorischen Logik zwischen

[1]Diese Eigenschaft wird als "delay-insensitiv"bezeichnet (DI) [Mar90].
[2]Die folgenden Ausführungen folgen im Wesentlichen der Darstellung bei [Kom+88].

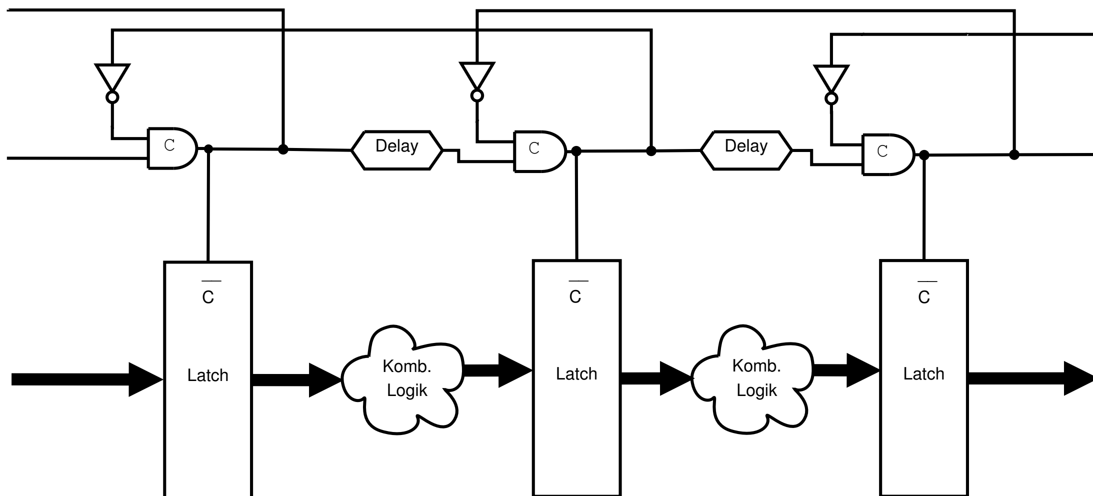


Abbildung 2.7.: 4-Phasen Bundled-Data Pipeline

zwei Pipelinestufen. Ein Nachteil dieser in Abbildung 2.7 dargestellten Pipeline liegt in der Durchsatzgeschwindigkeit. Der Durchsatz ist abhängig von der Zeit, die für einen kompletten Handshakezyklus benötigt wird. Bei der gezeigten Implementierung sind beide Nachbarn in die Kommunikation involviert. Es muss also der Nachfolger ein ACK senden und der Vorgänger ein REQ, bevor das Muller C-Element der Pipelinestufe auf '1' wechselt und somit das lokale Taktsignal gesetzt wird. Ein weiterer Nachteil der gezeigten Pipeline ist: Wenn sie gefüllt ist, besitzt nur jedes zweite Latch gültige Daten. Jedes andere Latch ist transparent.
Im Rahmen des Amulet 2 Projektes [Fur+97] der University of Manchester entstanden Control Circuits [FD96]. Diese Schaltungen entkoppeln die Pipelinestufen voneinander und bieten damit eine Möglichkeit, die genannten Nachteile zu neutralisieren. Die Implementierrung von Latch Control Ciruits wird im laufe der Arbeit beschrieben.

2.3.3. Micropipelines

Ian Sutherland stellt in [Sut89] eine 2-Pasen Bundled Data Pipelinestruktur vor (Micropipeline). Wie auch bei den 4-Phase Bundled Data Pipelines wird eine Muller Pipeline verwendet, um lokale Taktsignale zu erzeugen. Der Unterschied besteht in der Interpretation der Signale. Abbildung 2.8 zeigt eine dreistufige Pipeline nach Sutherland. Damit die Pipeline funktioniert sind spezielle Capture-Pass Latches notwendig. Der C und P Eingang eines Latches muss so geschaltet werden, dass Capture Modus und Pass Modus sich abwechseln. Das wird erreicht indem der Capture Eingang an REQ und der Pass Eingang an ACK angelegt wird.
Abbildung 2.9 zeigt ein Registerlatch nach Sutherland. Es sind zwei Latches integriert die nicht

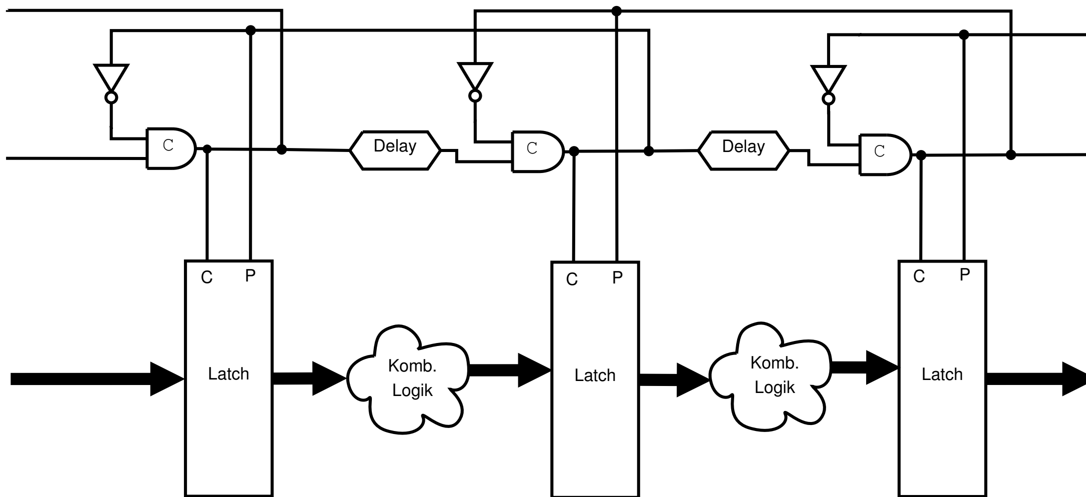


Abbildung 2.8.: Micropipeline mit "Capture-PassLatches, siehe Text

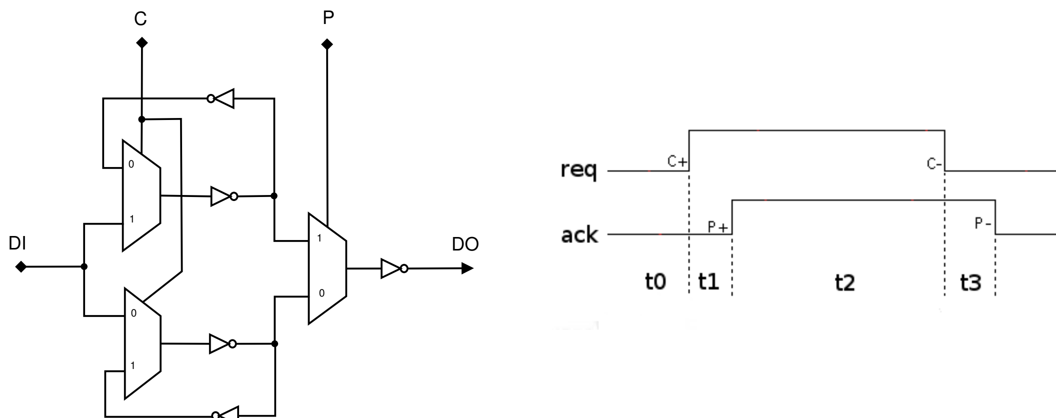


Abbildung 2.9.: Realisierung eines Capture-Pass Latch

hintereinander, sondern parallel angeordnet sind. Hieraus ergibt sich eine schnelle Verarbeitung der Signale. Tabelle 2.2 zeigt die Operationen des Capture-Pass Latches. Zum Zeitpunkt

Zeitpunkt	C	P	Mudus
t0	0	0	pass
t1	1	0	capture
t2	1	1	pass
t3	0	1	capture

Tabelle 2.2.: Operationen des Capture-Pass Latch

t0 sind beide Eingänge '0' und das Latch ist im Pass-Modus, also transparent. Das Signal DI fließt durch den unteren Multiplexer zum Multiplexer mit dem P Eingang als Steuersignal und anschließend zum Ausgang DO. Ein Signalwechsel am C Eingang schaltet das untere Latch in den Capture-Modus. Zum Zeitpunkt t2 gelangt das Latch wieder in den Pass-Modus, jedoch fließt das Signal DI jetzt durch den oberen Multiplexer. Der Signalwechsel vom Eingang C zum Zeitpunkt t3 hat zur Folge, dass das Latch wieder in den Capture-Modus wechselt.
Durch die zwei eingesparten Phasen hat eine Micropipline einen Performancevorteil gegenüber Pipelines mit 4 Phasen. Bei der heute überwiegend eingesetzten CMOS Technologie ist die Verlustleistung abhängig von den Schaltvorgängen. Die Muller Pipeline muss nur die Hälfte der Schaltungen vornehmen um einen Pipelinezyklus abzuschließen. Darum ist der Energieverbrauch für den Kontrollpfad zwar geringer als bei 4 Phasen Bundled Data Pipelines, der Verbrauch der Capture-Pass Latches hingegen ist aber erheblich höher als bei RS-Latches [BOF10, chapter 9.1]. Somit ist der Energiebedarf einer Micropipeline ähnlich wie bei einem synchronen Entwurf [Pav94]. Darüber hinaus benötigen Capture-Pass Latches viel Platz.
Bei einer Pipeline kommt es häufig vor, dass Berechnungen Werte benötigen, die in unterschiedlichen Pipelinestufen gespeichert sind. Bei einem synchronen Entwurf ist das kein Problem, weil jedes Datum taktsynchron übernommen wird und dadurch jedes Datum gleichzeitig gültig ist. Bei asynchronen Systemen ist das nicht so. Die worst-case Laufzeit der kombinatorischen Logik jeder Piplinestufe ist abgebildet in dem Delay-Element. Jede Pipelinestufe schaltet daher individuell ihre Latches. Aus diesen Grund müssen bei dem Entwurf des Kontrollpfades Abhängigkeiten berücksichtigt und gegebenenfalls aufgelöst werden. Bei den Amulet Projekten [Pav94; Fur+97], hat sich herausgestellt, dass es schwieriger ist diese Abhängigkeiten bei Pipelines aufzulösen, die das 2-Phase Bundled-Data Protokoll benutzen. Dieser Nachteil ist der Grund warum bei großen Projekten vorwiegend der 4-Phase Bundled Data Ansatz angewendet wird. [Gag+98; Ren+98]. Bei Systemen mit unabhängigen Datenfluss und hohen Performance Anforderungen ist das 2-Phase Bundled-Data Protokoll zu bevorzugen [SF01,

chapter 2.4.2]. In dieser Arbeit wird das 4 Phasen Bundled Data Protokoll verwendet, weil keine Capture-Pass Latches auf Field Programmable Gate Arrays (FPGAs) vorhanden und zudem Entwicklungen von Komponenten für das 4 Phasen Bundled-Data Protokoll zahlreicher und besser dokumentiert sind.

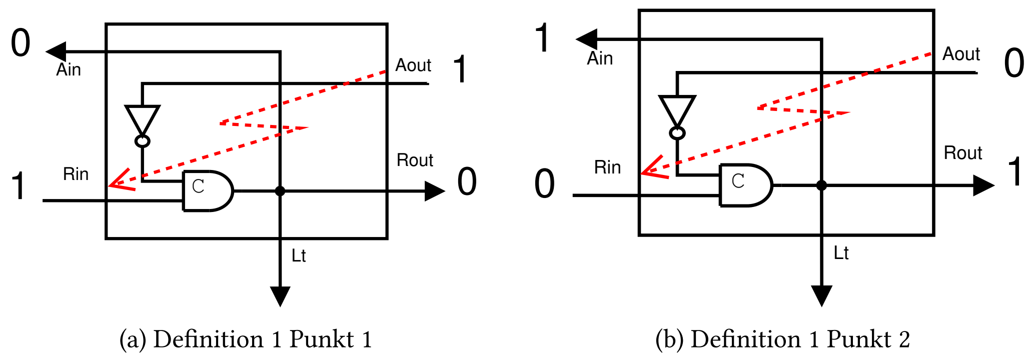
2.3.4. Entkopplungsgrad

Der Entkopplungsgrad gibt an, wie die Handshakesignale einer Pipelinestufe mit ihren Nachbarn verknüpft sind und interagieren. Um den Grad der Entkopplung zu bestimmen, werden in [Liu97, S. 101] folgende Begriffe definiert:

- Initiierter Handshake: Ein Handshake ist initiiert, wenn ein neuer Request nicht blockiert wird. Der Handshake kann sich also fortpflanzen.
- Komplettierter Handshake: Wenn das REQ zurück auf '0' geht und dadurch das ACK ebenfalls auf '0' wechselt, ist ein Handshake komplettiert.
- Unterbrochener Handshake: Ein Handshake ist unterbrochen, wenn sich das zurücksetzen des REQ nicht fortpflanzen kann.

In der folgenden Aufzählung wird der Grad der Entkopplung definiert. Die Definitionen stammen ebenfalls aus [Liu97]

1. Eine Pipeline gilt als nicht entkoppelt (un-decoupled),
    - wenn ein neuer Handshake am Eingang nicht beginnen kann, bevor der alte Handshake komplettiert wurde, oder
    - wenn ein neuer Handshake am Eingang unterbrochen wird, weil ein neuer Handshake am Ausgang nicht initiiert wurde.
2. Ein Handshake gilt halb-entkoppelt (semi-decoupled), wenn der erste Punkt aus der vorigen Definition nicht gilt.
3. Ein Handshake gilt als entkoppelt (decoupled) wenn beide Punkte aus Definition 1 nicht gelten.



(a) Definition 1 Punkt 1

(b) Definition 1 Punkt 2

Abbildung 2.10.: Szenarien einer gekoppelten Pipelinestufe

Abbildung 2.10 zeigt den Teil aus Abbildung 2.7, der für die Kontrolle des Latches zuständig ist. Es sind die beiden Situationen der gekoppelten Pipelinestufe abgebildet. Bei dem ersten Punkt 2.10a wird der Request am Eingang (Rin) so lange blockiert, bis das ACK am Ausgang (Aout) auf '0' wechselt. Durch einen Wechsel ist der Handshake komplettiert. Das bedeutet, dass das nachfolgende Latch offen sein muss, damit das Latch der betrachteten Pipelinestufe schließen kann. Wenn eine gekoppelte Pipeline gefüllt ist, dann besitzt nur jedes zweite Latch Daten. Jedes andere Latch ist leer, also offen. Auch bei dem zweiten Punkt 2.10b wird der Handshake am Eingang blockiert; diesmal kann das Request Signal (Rout) nicht auf '0' zurückgesetzt werden, weil die Bestätigung durch das Aout Signal noch ausbleibt.
Bei einer halb entkoppelten Pipeline, darf sich das betrachtete Latch füllen, auch wenn bei der nachfolgenden Pipelinestufe das ACK noch nicht wieder auf '0' gesetzt wurde. Allerdings darf das aktuelle Latch nicht geöffnet werden, bevor die nachfolgende Pipelinestufe ein ACK gesendet hat. Bei entkoppelten Pipelines darf sich die aktuelle Stufe füllen und leeren, unabhängig von dem entsprechenden ACK der nachfolgenden Pipelinestufe.

2.3.5. Fork und Join Elemente

Fork und Join Komponenten werden verwendet, um parallele Berechnungen zu kontrollieren. Ein Fork wird benötigt, wenn der Ausgang einer Komponente von mehr als einer Komponente als Input verwendet wird.

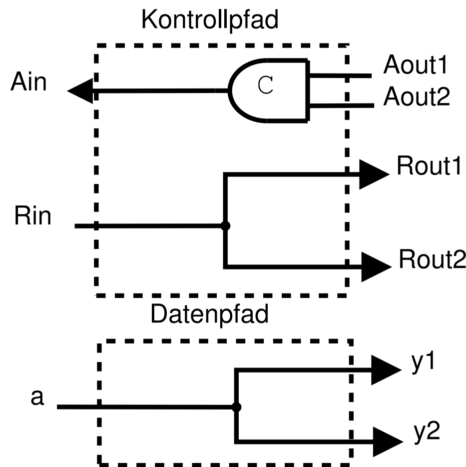


Abbildung 2.11.: 4 Phasen Bundled Data Fork

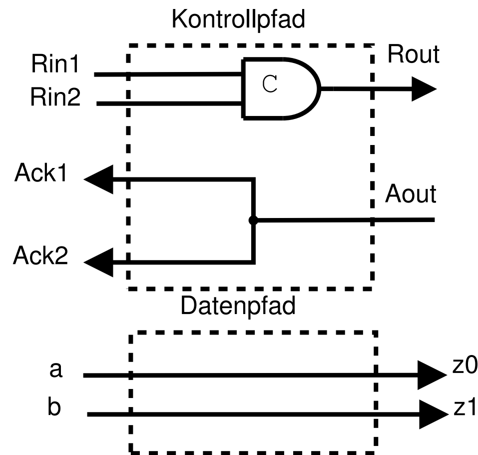


Abbildung 2.12.: 4 Phasen Bundled Data Join

Joins werden verwendet, wenn die Eingänge mehrerer unabhängiger Komponenten synchronisiert werden müssen. Der Fork für 4 Phasen Bundled Data besitzt ein Muller C-Element, um die ACK Leitungen der nachfolgenden Komponenten miteinander zu vereinen. Bei einem Join werden die REQs Leitungen der Vorgänger durch ein Muller C-Element vereint. In Abbildung 2.11 und 2.12 sind die Implementierungen der Fork und Join Komponenten zu sehen.

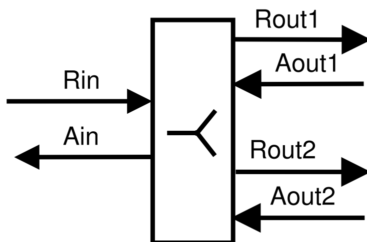


Abbildung 2.13.: Schaltsymbol Fork

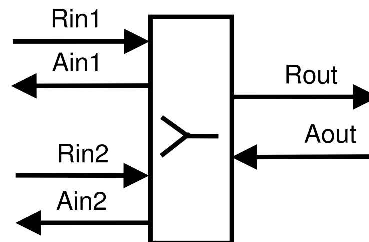


Abbildung 2.14.: Schaltsymbol Join

2.4. Signal Transition Graphs

Eine verbreitete Methode asynchrone Schaltungen graphisch zu spezifizieren sind Signal Transition Graphs (STGs). Die nachfolgenden Erläuterungen der STGs basieren auf der Dissertation von Chu [Chu87]. Petrinetze wurden entwickelt von Carl Adam Petri [Pet62]. Petrinetze werden für viele unterschiedliche Problemlösungen verwendet. Es gibt zahlreiche Varianten und Verfeinerungen von Petrinetzen. Eine Variante sind STGs, sie können als eine Art formalisiertes Timing-Diagramm angesehen werden.

2.4.1. Petrinetze

Ein Petrinetz ist ein gerichteter Graph, bestehend aus einer endlichen Anzahl von Transitionen T, einer endlichen Anzahl von Plätzen[3] P und Flussrelationen $F \subseteq (P \times T) \cup (T \times P)$ die eine binäre Relation zwischen Transition und Plätzen definiert [Rei10, S. 22]. Besonders gut eignen sich Petrinetze zur Modellierung von diskreten verteilten Systemen, sowie zur Modellierung von Nebenläufigkeit und Parallelität. Nebenläufigkeit bedeutet, dass Prozesse voneinander unabhängig ablaufen, also sich gegenseitig nicht beeinflussen. Um dynamische Vorgänge beschreiben zu können, werden die Plätze mit Marken belegt, die mit schwarzen Punkten in den Plätzen symbolisiert werden. Jeder Platz kann mehrere Marken aufnehmen. Beim Schalten einer Transition werden Marken an Inputplätzen verbraucht und an Outputplätzen erzeugt, wobei die Gesamtanzahl der Marken sich ändern kann. Eine Transition kann nur schalten, wenn alle ihrer Inputplätze mindestens eine Marke besitzen.
Abbildung 2.15 zeigt ein einfaches Petrinetz vor dem Schalten einer Transition. Abbildung 2.16 zeigt den Zustand des Petrinetzes, wenn die Transition t0 geschaltet hat. Das Petrinetz hat auf den Plätzen P0, P1 und P2 jeweils eine Marke verbraucht, wohingegen die Plätze P3 und P4 eine Marke dazu bekommen haben. Die Anzahl der Marken hat sich von 5 auf 4 verringert. Wenn in einem Schritt mehrere Transitionen schalten können, wird eine Transition nichtdeterministisch ausgewählt. Auch wenn das Petrinetz parallele Abläufe modelliert, schaltet in jedem Schritt nur genau eine Transition. Zwei Transitionen mit gemeinsamen Stellen im Vorbereich stehen bei passender Markierung im Konflikt. Jede der beiden Transitionen kann schalten, aber nicht beide nacheinander.

2.4.2. STGs

Petrinetze sind eine Gruppe von verwandten Modellen und kein einzelnes eindeutig definiertes Modell. Die Motivation von Chu [Chu87] war es, ein Modell zu schaffen, um möglichst einfach Self-Timed Schaltkreise zu beschreiben und sie effizient zu synthetisieren. STGs sind eine Untergruppe der Petrinetze und haben folgende Eigenschaften und Einschränkungen:

- Plätze dürfen nie mehr als eine Marke besitzen.
- Die Auswahl der Alternativen darf nur von Eingängen mit gegenseitigem Ausschluss kontrolliert werden.
- Der STG muss frei von Deadlocks sein.

[3] In der Standardliteratur zu Petrinetzen variiert die Bezeichnung für einen Platz. Reisig verwendet den Begriff „Platz“[Rei10], andere Veröffentlichungen verwenden den Begriff „Stelle“.

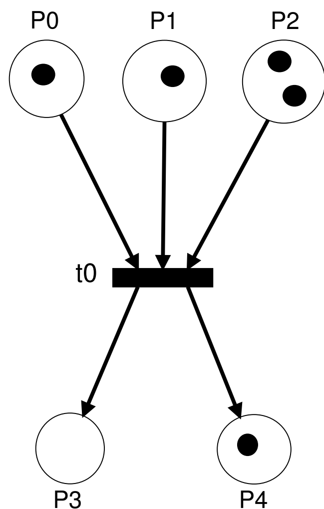


Abbildung 2.15.: Einfaches Petrinetz

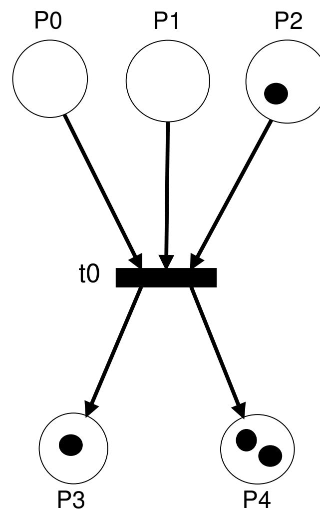


Abbildung 2.16.: Petrinetz nach dem Schalten der Transition t0

Ein STG der eine Schaltung beschreibt die Speed Independent ist, sollte folgende Charakteristika haben:

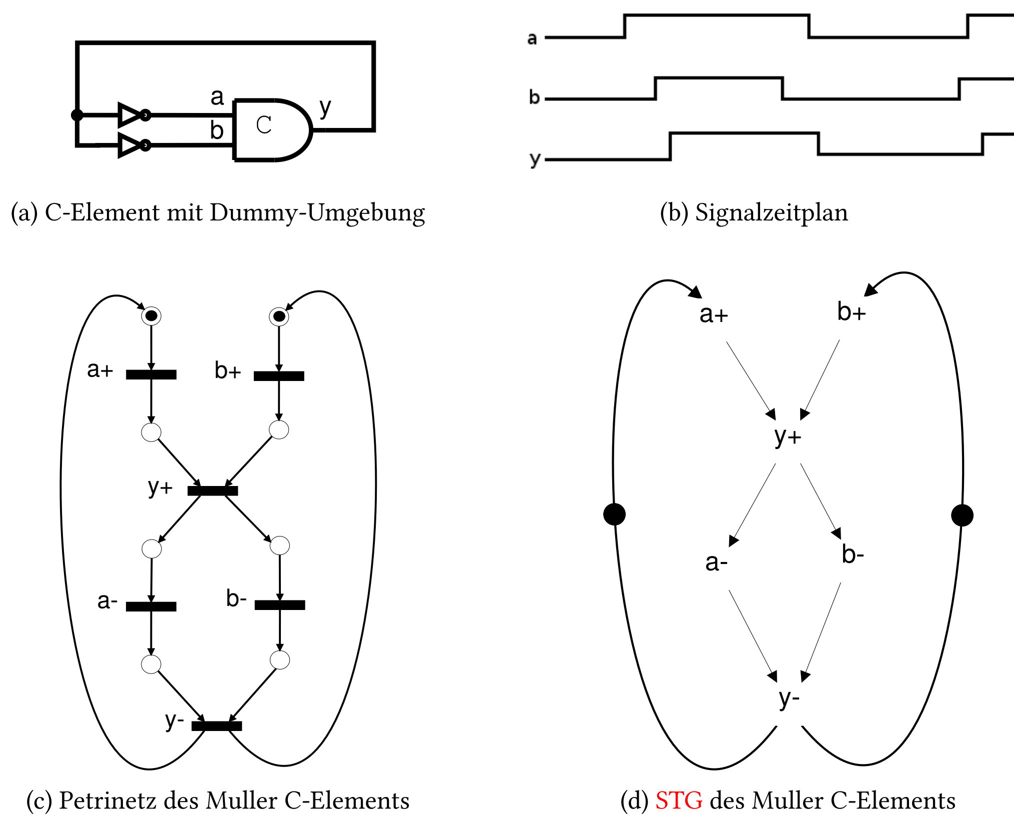
- Die Transitionen eines Signals müssen zwischen steigender und fallender Flanke alternieren.
- „Wenn eine Signaltransition aktiv ist, muss sie auch stattfinden. Sie darf nicht von einer anderen Signaltransition deaktiviert werden. Die STG Spezifikation einer Schaltung muss garantieren, dass die internen Signale und die Output-Signale persistent sind. Dafür ist es die Aufgabe der Umgebung, für die Persistenz der Input-Signale zu sorgen.“ [SF01]

Um ein STG synthetisieren zu können, ist die folgende Eigenschaft erforderlich:

- „Zwei oder mehr unterschiedliche Markierungen eines STG dürfen nicht die gleichen Signalwerte haben (entsprechend zu dem gleichen Zustand). Wenn das nicht der Fall ist, ist es möglich zusätzliche Zustandsvariablen einzufügen, die entsprechend zu unterschiedlichen Zuständen führen.“ [SF01]

Ein STG kann als eine formale Spezifikation der Signalwechsel eines Prozesses angesehen werden, der zum Beispiel aus einem Signalablaufdiagramm gewonnen wird. Die Idee von Chu [Chu87] besteht darin, dass durch kausale Relationen die beschriebenen Signalwechsel in eine Schaltung mit gleichem Verhalten umgewandelt werden kann. Es werden im Gegensatz zu

den Zustandsgraphen nicht die Zustände auf die Knoten des Graphen abgebildet, sondern die Signalwechsel. Das hat bei der Umsetzung der Schaltungsfunktion zur Folge, dass die Methodik zur Kodierung der Zustände frei ist. Des Weiteren können lokale Prozesse beschrieben werden, ohne das Verhalten und die Kodierung von anderen Schaltungsfunktionen zu berücksichtigen. Abbildung 2.17(a) zeigt das Muller C-Element mit einer Dummy Umgebung, die dafür sorgt, dass die Eingänge aktualisiert werden, wenn der Ausgang sich ändert. Darunter ist in Abbildung 2.17(c) die Petrinetzdarstellung des Muller C-Elements. Neben dem Petrinetz ist der äquivalente STG in Abbildung 2.17(d) dargestellt.



(a) C-Element mit Dummy-Umgebung

(b) Signalzeitplan

(c) Petrinetz des Muller C-Elements

(d) STG des Muller C-Elements

Abbildung 2.17.: Formale Spezifikation mit einem Petrinetz und einem STG des Muller C-Elements aus einem Signalzeitplan [Chu87; SF01]

STGs werden im Vergleich zu Petrinetzen etwas vereinfachter dargestellt, indem die meisten Plätze weggelassen werden. Jeder Pfeil, der zwei Transitionen miteinander verbindet, stellt somit einen Platz dar. Ein „+“ hinter einem Signalnamen bedeutet einen Wechsel des Signals von ’0’ auf ’1’. Ein „-“ bedeutet einen Wechsel von ’1’ auf ’0’.

Der STG ist an seinen Eingängen zu den Transitionen a+ und b+ markiert, der zugehörige Zustand ist: $(a, b, y) = (0, 0, 0)$. Die Transitionen a+ und b+ können in jeder Reihenfolge auslösen. Wenn beide ausgelöst haben $(1, 1, 0)$, ist y+ bereit ebenfalls auszulösen $(1, 1, 1)$ etc.

2.4.3. STG Synthese

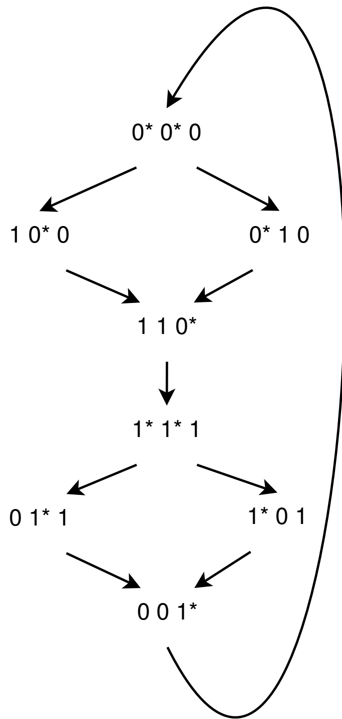


Abbildung 2.18.: Zustandsgraph des Muller C-Elements

Abbildung 2.19.: KV-Diagramm für y

	y		$\bar{y}$	
b	1	1	0*	0
$\bar{b}$	1*	1	0	0
	$\bar{a}$	a	a	$\bar{a}$

$$y' = by \vee ab \vee ay$$

Der erste Schritt der Synthese ist das Sicherstellen der Eigenschaften, die im Unterkapitel 2.4.2 aufgelistet sind. Der zweite Schritt ist das Ableiten des entsprechenden Zustandsgraphen aus dem STG. Der Zustandsgraph wird erstellt, indem alle möglichen Markierungen identifiziert werden, die von der initialen Markierung erreichbar sind. Der letzte Schritt des Syntheseprozesses ist das Entwickeln der Booleschen Gleichungen für die Zustandsvariablen und Ausgangsvariablen.

Um den Syntheseprozess zu verdeutlichen, wird an dieser Stelle ein Beispiel näher betrachtet. Es wird das Muller C-Element aus Abbildung 2.17 synthetisiert. Die Abbildung 2.18 zeigt den entsprechenden Zustandsgraphen des Muller C-Elements. Die Anordnung der Variablen ist: $a\ b\ y$. Wenn eine Variable sich durch die nächste Transition ändern kann, ist sie mit einem Stern gekennzeichnet. Die Boolesche Gleichung wird mit Hilfe eines Karnaugh-Veitch-

Diagramms (KV-Diagramms) ermittelt. Das KV-Diagramm ist in Abbildung 2.19 dargestellt. Um die Boolesche Gleichung für den Ausgang y zu erstellen, müssen die Felder mit $y = 1$ und $y = 0*$ zusammengefasst werden.

2.4.4. STG Synthese mit Petrify

Im folgenden etwas komplexeren Beispiel wird das Programm Petrify angewendet. [Cor+97]. Der STG in Abbildung 2.20 beschreibt einen einfachen Latch-Controller für das 4 Phasen Bundled Data Protokoll. Das Signal „Lt“ ist das Steuersignal für das Latch. Es öffnet das Latch mit '0' und schließt es mit '1'. Die Signale Rin und Ain interagieren mit dem Vorgänger und die Signale Rout und Aout mit dem Nachfolger einer Pipelinestufe. Der STG spezifiziert diese Interaktionen. Das Programm Petrify erwartet eine Textdatei, in der ein STG im Asynchronous

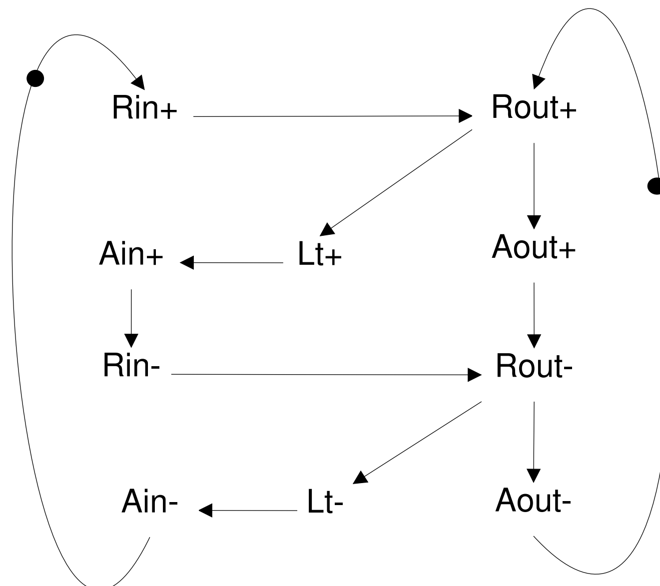


Abbildung 2.20.: Einfacher Latch Controller STG

Signal Transition Graph (ASTG) Format beschrieben wird. Das ASTG Format ist in [Pas04] spezifiziert. In der Datei werden als erstes die Ein- und Ausgangs- sowie die internen Signale deklariert. Als nächstes werden alle Verbindungen des STG definiert. Dabei wird der Name der Transition oder des Platzes, gefolgt von einer oder mehreren Transitionen oder Plätzen, aufgelistet an die die Markierung weitergereicht wird. Als letztes werden die Anfangsmarkierungen definiert. Listing 2.1 zeigt die Definition des STG aus Abbildung 2.20 in dem ASTG Format. Bevor der STG synthetisiert wird, kontrolliert Petrify, dass alle Eigenschaften, die zur

Synthese erforderlich sind, eingehalten wurden. Wenn es nötig ist, fügt Petrify zusätzliche Zustände ein, damit keine Markierung den gleichen Signalwert hat.

```
.model Einfacher
.inputs Aout Rin
.outputs Ain Rout Lt
.graph
Rin+ Rout+
Aout+ Rout-
Aout- Rout+
Ain- Rin+
Lt- Ain-
Rin- Rout-
Rout- Aout- Lt-
Ain+ Rin-
Lt+ Ain+
Rout+ Aout+ Lt+
.marking { <Aout- ,Rout+ >  <Ain- ,Rin+ > }
.end
```

Listing 2.1: Einfacher Latch-Controller

Die Option -cg beim Aufruf bewirkt, dass Petrify eine Schaltung mit komplexen Gattern produziert. Der Aufruf mit der Option -gc produziert eine Schaltung mit Muller C-Elementen. Die Ausgabe hat Boolesche Funktionen, von denen man die Set und Reset Funktionen für Muller C-Elemente ablesen kann. Es ist ebenfalls möglich eine Technologiebibliothek anzugeben, damit Petrify diese bei der Synthese verwendet. Die Technologiebibliothek muss in dem GENLIB Format sein. Das GENLIB Format wird in [Rud87] definiert.
Synthetisiert man das obere Beispiel mit dem Befehl: *petrify -tm -lib petrify.lib controller.g -eqn out.eqn*, erhält man die Ausgabe in Listing 3.2. In dem Befehl gibt *petrify.lib* die Datei an, in der die Technologiebibliothek definiert ist; *controller.g* ist die Datei mit dem Inhalt aus 2.1.

```
INORDER = Aout Rin Ain Rout Lt;
OUTORDER = [Ain] [Rout] [Lt];
[Ain] = Lt;
[Lt] = Rout;
[0] = Aout';        # gate inv
[Rout] = Rin ([0] + Rout) + Rout [0];       # gate cgate

# Set/reset pins: reset(Rout)
```

Listing 2.2: Ausgabe von Petrify

Es ist ein Inverter und ein Muller C-Element aus der Technologiebibliothek verwendet worden. Das Syntheseprogramm hat erkannt, dass die Signale „Ain", „Lt" und „Rout" die gleichen sind. Die Boolesche Gleichung in Zeile 6 des Listing 3.2 kann auf einem Muller C-Element abgebildet werden. Formt man die Gleichung um, erhält man die Gleichung für das Muller C-Element aus Abbildung 2.19.

$$[Rout] = Rin \wedge ([0] \vee Rout) \vee Rout \wedge [0]$$

$$[Rout] = Rin \wedge Rout \vee Rin \wedge [0] \vee Rout \wedge [0]$$

Die Eingänge des Muller C-Elements sind damit Rin und das invertierte Signal von Aout also „0". Werden alle Signale auf ihre Bauelemente verteilt und verbunden, erhält man das Resultat in Abbildung 2.21.

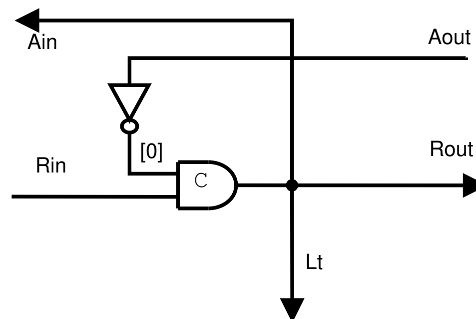


Abbildung 2.21.: Schaltbild eines einfachen Latch-Controllers

Eine weitere Fähigkeit von Petrify ist es, Verilog-Code zu erzeugen. In Abbildung 2.22 ist der Workflow für das Erzeugen von Komponenten mit Petrify dargestellt. Wenn eine Verilog Komponentenimplementation erstellt werden soll, muss eine Technologiebibliothek bei dem Aufruf von Petrify angegeben werden. In dieser Datei sind die elementaren Komponenten definiert, wie das Muller C-Element und die grundlegenden Logikgatter (AND, OR, XOR...). Nachdem Petrify den Verilog-Code erzeugt hat, kann dieser in einem HDL Entwurf instantiiert werden. Dabei spielt es keine Rolle ob die Instantiierung in VHDL oder in Verilog erfolgt.[4] Die Logiksynthese benötigt eine konkrete Implementierung jeder Komponente, die in der Technologiebibliothek definiert ist. Im Anhang F.2 ist die Technologiebibliothek für Petrify abgedruckt. Die Implementierung der Komponenten wird im nächsten Kapitel für Xilinx FPGAs näher betrachtet.
Es gibt mehrere Programme um Petrinetze und STGs zu zeichnen. Ein Programm ist VSTGL.

[4]Bei den Programmen Xilinx ISE und Altera Quartus 2 können innerhalb eines VHDL Entwurfs Verilog-Module instantiiert werden.

Es ermöglicht neben dem Zeichnen und Simulieren eines STGs auch das Exportieren als .g Datei für die Synthese mit Petrify.

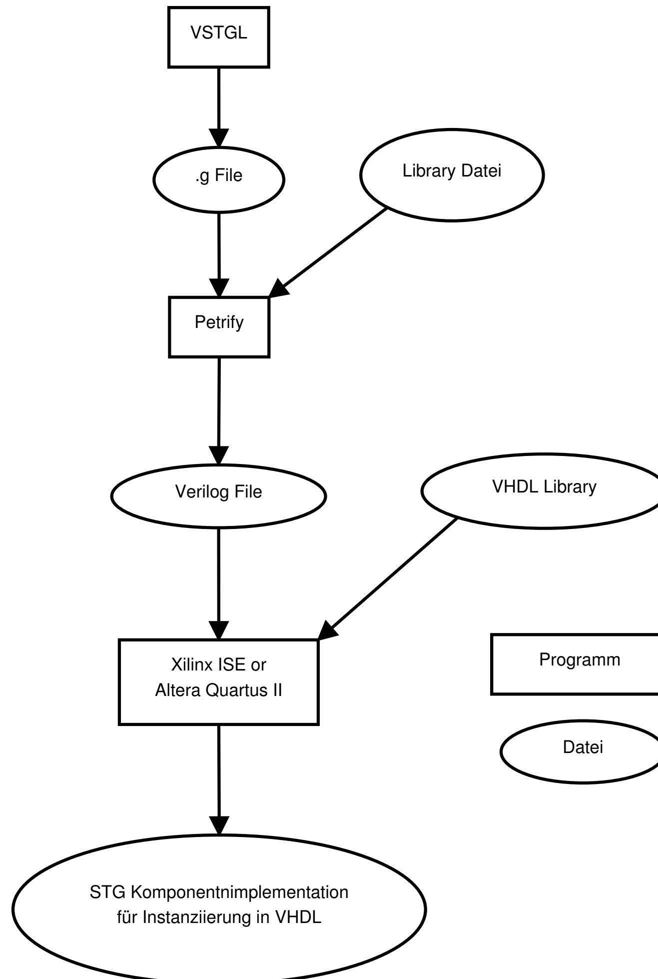


Abbildung 2.22.: Workflow für Petrify

3. Implementierung asynchroner Schaltungen auf FPGAs

Damit eine Konvertierung einer synchronen Schaltung in eine asynchrone Schaltung möglich wird, werden grundlegende Komponenten benötigt, die bei einer Entwicklungsumgebung für den synchronen Entwurf nicht enthalten sind. In diesem Kapitel werden alle Komponenten vorgestellt, die nötig sind, um asynchrone Schaltungen auf FPGAs zu implementieren.

3.1. FPGAs

Im folgenden wird grob der interne Aufbau eines Spartan 3e FPGA der Firma Xilinx beschrieben. Ein FPGA besteht aus einer großen Anzahl von Configurable Logic Blocks (CLBs). Eine Schaltmatrix organisiert das Routing zwischen den CLBs. Jeder CLB beinhaltet vier miteinander verbundene Slices. Die Slices sind gruppiert in Paaren. Jedes Paar ist als Spalte angeordnet mit einer unabhängigen „Carry Chain“. Das linke Paar ist für Logik- sowie für Speicheraufgaben, das rechte Paar ist nur für Logikaufgaben vorgesehen. Abbildung 3.1 zeigt die Gruppierung der Slices eines Spartan 3e FPGAs. Jeder Slice verfügt über zwei Look-Up Tables (LUTs) mit vier Eingängen und einem Ausgang. Eine LUT dient zur Realisierung kombinatorischer Logik. Sie arbeitet nach dem Prinzip eines programmierbaren Speichers mit wahlfreiem Zugriff. Die vier Eingänge bilden hierbei die Adresse, unter der die Logikzustände für den Ausgang abgespeichert sind. Dadurch kann für eine beliebige Eingangskombination sofort auf den Wert für den Ausgang zugegriffen werden. Des Weiteren verfügt ein Slice über zwei Speicherelemente, die entweder als Latch oder als Flip-Flop konfiguriert werden können. Die beiden linken Slices eines CLB können darüber hinaus als 16x1 Distributed RAM funktionieren.

Abbildung 3.2 zeigt das Nummerierungsschema der Slices innerhalb des FPGAs. Wenn bei einem Entwurf das Timing beeinflusst werden muss, ist die Nummerierung wichtig, um zu bestimmen welche Slices wie benutzt werden. Xilinx bietet hierfür die LOC Constraints an. LOC Constraints werden im User Constraints File (UCF) definiert oder direkt in der HDL Datei [Xil08, S. 160]. Die Xilinx FPGAs der nächsten Generation haben LUTs mit 6 Eingängen und 2

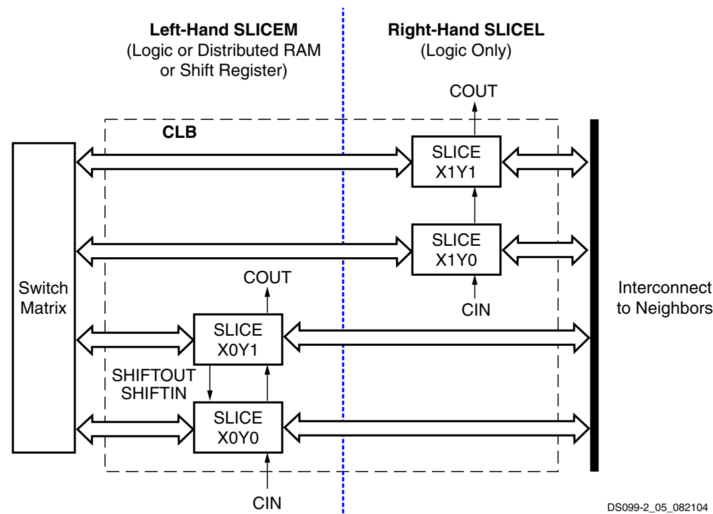


Abbildung 3.1.: Gruppierung der Slices in einem CLB [Xil13]

Ausgängen, dadurch passt mehr Logik in die LUTs. Für die Logiksynthese- und Logiksimulation bietet Xilinx das Xilinx ISE Design Software Paket an. In dem Glossar ist eine Auflistung der wichtigsten Bestandteile der Software.

3.2. Muller C-Element

3.2.1. Muller C-Element mit 2 Eingängen

Bei der Implementierung asynchroner Schaltungen auf FPGAs sind die Verzögerungen der Leitungen bedeutender als bei Fullcustom Integrated Circuits (ICs). Die Verbindung zweier CLBs geht durch eine oder mehrere programmierbare Switch-Boxes, die eine deutlich größere Verzögerung haben als normale Leitungen. Um Hazards zu vermeiden, muss daher bei dem Routing sehr auf die Balance geachtet werden. Eine lange Verzögerung in der Rückkopplung oder ein schneller Wechsel der Inputsignale eines Muller C-Elements kann Hazards hervorrufen. Um zu vermeiden, dass die Rückkopplung Hazards verursacht, sollte ein Muller C-Element so geroutet sein, dass folgende Ungleichung zutrifft:

$$\delta_{max}(FB) < min\{\delta_{min}(A), \delta_{min}(B)\} + \delta_{min}(Y) + \delta_{min}(OCIC)$$

Dabei ist δ_{min} die minimale und δ_{max} die maximale Verzögerung. $OCIC$ ist die kombinatorische Logik, die Änderungen am Ausgang entdeckt und einen neuen Eingangswert produziert.

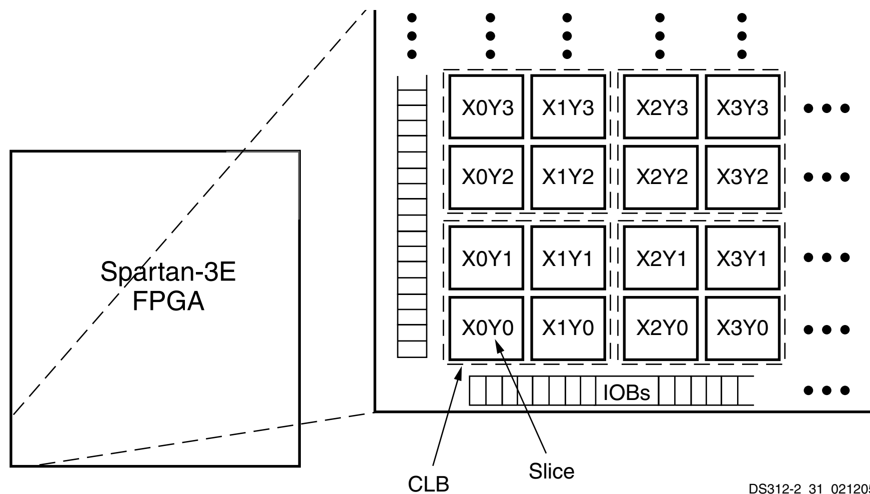


Abbildung 3.2.: Nummerierung der Slices im FPGA [Xil13]

FB ist die Rückkopplungsleitung. Diese Formel wurde vorgestellt und bewiesen in [Mah95]. Ein Muller C-Element kann mit nur einer LUT realisiert werden, indem der Ausgang durch eine lokale Rückkopplung zu einem Eingang geleitet wird. Die Rückkopplung erfolgt also innerhalb eines Slices und verläuft nicht über Switch-Boxes. Die oben angegebene Formel ist damit erfüllt, weil $\delta_{max}(FB)$ sehr klein ist. Die Rückkopplung kann daher keine Hazards mehr verursachen.
Mit der UNISIM Bibliothek von Xilinx ist es möglich, einzelne LUTs zu konfigurieren. Mit dem VHDL Code aus Listing 3.1 wird eine LUT als Muller C-Element konfiguriert.

```
c_element: lut4_l
generic map (
  init => "11101000" & reset_vector
)
port map (
  i0 => a,
  i1 => b,
  i2 => y_out,
  i3 => reset,
  lo => y_out
);
```

Listing 3.1: VHDL Implementierung des Muller C-Elements

Die lokale Rückkopplung wird erreicht, indem der Ausgang lo auf den Eingang i2 gemappt wird. Der Initialisierungsvektor, der bei der Instantiierung einer LUT angegeben wird, kon-

figuriert das Verhalten. Wie im letzten Abschnitt schon erwähnt, kann eine LUT als ein programmierbarer Speicher angesehen werden, der an einer Adresse lediglich ein Bit speichert. In der Tabelle 3.1 sind alle Kombinationen für die Eingänge i3 bis i0 aufgelistet. Die ersten vier Spalten der Tabelle 3.1 bilden für jede Zeile eine Adresse. In der letzten Spalte ist für jede Eingangskombination der Wert des Ausgangs verzeichnet. Der Initialisierungsvektor für eine LUT ist in der letzten Spalte ablesbar. Der Resetvektor, der bei der Instantiierung der LUT im

i3 = Reset	**i2 = cout**	**i1 = b**	**i0 = a**	**init Vektor = cout**
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	RV(7)
1	0	0	1	RV(6)
1	0	1	0	RV(5)
1	0	1	1	RV(4)
1	1	0	0	RV(3)
1	1	0	1	RV(2)
1	1	1	0	RV(1)
1	1	1	1	RV(0)

Tabelle 3.1.: Tabelle für den Initialisierungsvektor des Muller C-Elements

Listing 3.1 mit angegeben wird, bestimmt, ob der Ausgang des Muller C-Elements nach einem Reset ’0’ oder ’1’ ist. Abbildung 3.3 zeigt das Technologieschaltbild nach der Synthese mit dem Xilinx Synthetisierer.

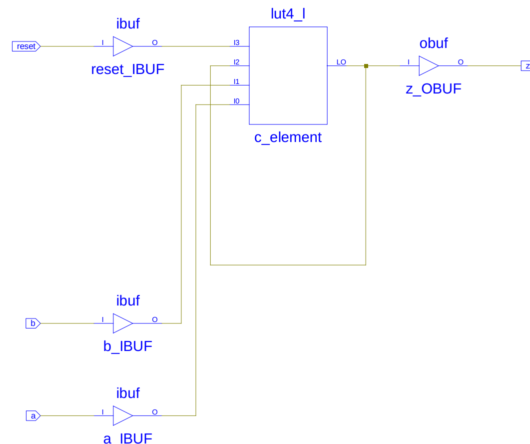


Abbildung 3.3.: Schaltbild des Muller C-Elements

Die komplette VHDL Implementierung des Muller C-Elements befindet sich in Anhang F.1.1.

3.2.2. Muller C-Element mit 3 Eingängen

Tabelle 3.2 beschreibt das Verhalten eines Muller C-Elements mit 3 Eingängen. Muller C-Elemente mit 3 Eingängen ohne Reseteingang können aus einer LUT erzeugt werden. Wie im letzten Abschnitt beschrieben, wird ein Eingang der LUT für eine lokale Rückkopplung verwendet. Die restlichen 3 Eingänge werden auf ähnliche Weise konfiguriert wie bei einem Muller C-Element mit 2 Eingängen. In den meisten Fällen wird jedoch ein definierter Zustand

A	B	C	Y	Erklärung
0	0	0	0	Der Wechsel auf 0 ist erfolgt
0	0	1	Halten	Der Ausgangszustand wird gehalten
0	1	0	Halten	Der Ausgangszustand wird gehalten
0	1	1	Halten	Der Ausgangszustand wird gehalten
1	0	0	Halten	Der Ausgangszustand wird gehalten
1	0	1	Halten	Der Ausgangszustand wird gehalten
1	1	0	Halten	Der Ausgangszustand wird gehalten
1	1	1	1	Der Wechsel auf 1 ist erfolgt

Tabelle 3.2.: Muller C-Element Funktionstabelle

nach einem Reset vorausgesetzt, um kein willkürliches Verhalten zu provozieren. In [PQDD10] wird beschrieben, wie ein Muller C-Element mit 3 Eingängen und Reset auf einem Xilinx FPGA realisiert wird. In der Veröffentlichung ist ein Verilog-Code enthalten, der zwei LUT

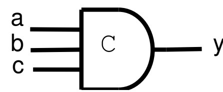


Abbildung 3.4.: Muller C-Element mit 3 Eingängen

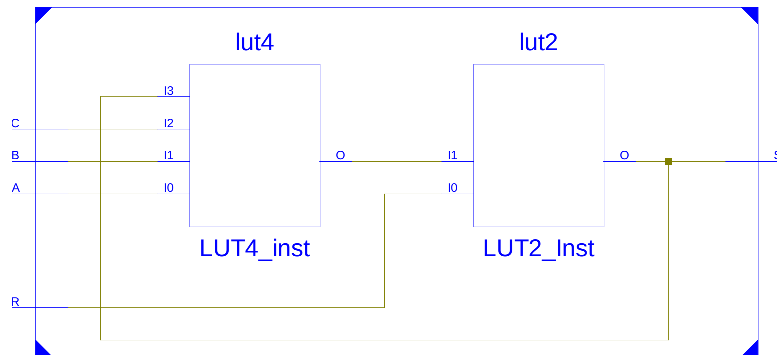


Abbildung 3.5.: Schaltbild des Muller C-Elements mit 3 Eingängen

wie in Abbildung 3.5 miteinander verbindet. Die linke LUT mit 4 Eingängen wird als Muller C-Element mit 3 Eingängen konfiguriert. Das Rückkopplungssignal ist das Ausgangssignal einer zweiten LUT. Die LUT2 wird als UND konfiguriert. Wenn also das Resetsignal R auf '0' ist, dann ist der Ausgang der LUT2 auch auf '0' und die Rückkopplung zu der LUT4 ebenfalls '0'. Das Muller C-Element hat den definierten Zustand '0'. Wenn das Resetsignal auf '1' wechselt, schaltet die LUT2 das Ausgangssignal von der LUT4 durch, und es entsteht die Rückkopplung. Das Muller C-Element kann so konfiguriert werden, dass nach einem Reset der Ausgang '1' ist. Dafür muss die LUT2 nicht als UND sondern als Implikation (siehe Tabelle 3.3) konfiguriert werden. Bei dieser Methode muss darauf geachtet werden, dass die beiden LUTs innerhalb

I1	I0	o
0	0	1
0	1	0
1	0	1
1	1	1

Tabelle 3.3.: Wahrheitstabelle der Implikation

eines Slice platziert sind. Die Rückkopplung würde sonst über die Switchmatrix geroutet werden, was Hazards verursachen könnte. Die Platzierung der LUT kann mit RLOC Constraint beeinflusst werden. RLOC Constraints werden im UCF definiert, indem der Name der LUT Komponente, gefolgt von der Nummer eines Slices, angegeben wird.

```
1 INST "LUT4_inst" RLOC=X0Y0;
2 INST "LUT2_Inst" RLOC=X0Y0;
```

Listing 3.2: UCF des Muller C-Elements mit 3 Eingängen

Wenn zwei LUTs die gleiche Nummer zugewiesen bekommen, werden sie gemeinsam in einem Slice platziert. Welcher Slice verwendet wird, entscheidet der „Place and Route“-Algorithmus.

3.2.3. Muller C-Element mit 4 Eingängen

Für die Implementierung von Muller C-Elementen mit 4 Eingängen und Resetsignal werden 3 LUTs benötigt. Die LUTs werden wie in Abbildung 3.6 verdrahtet. Zwei LUTs sind mit den

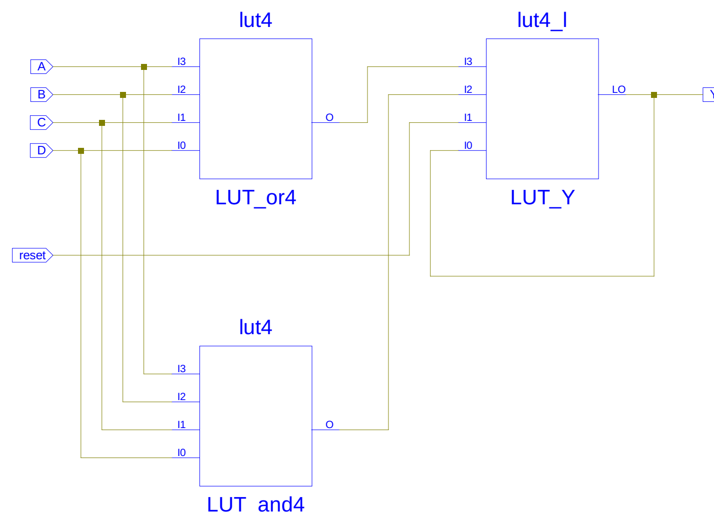


Abbildung 3.6.: Schaltbild des Muller C-Elements mit 4 Eingängen

vier Eingängen des Muller C-Elements verbunden. Eine davon ist als Oder-Verknüpfung, die andere als Und-Verknüpfung mit jeweils 4 Eingängen konfiguriert. Wenn alle Eingänge ’0’ sind, erscheint am Ausgang der LUT_or4 eine ’0’, sonst ist der Ausgang ’1’. Sind alle Eingänge auf ’1’, schaltet die LUT_and4 ihren Ausgang auf ’1’, der sonst ’0’ ist. LUT_or4 detektiert also einen Wechsel auf ’0’ und LUT_and4 einen Wechsel auf ’1’. Der Initialisierungsvektor für die LUT_Y ist aus Tabelle 3.4 ablesbar. Die LUT_Y implementiert mit einer lokalen Rückkopplung das speichernde Verhalten.
Wenn in dem VHDL Code des Muller C-Elements die Funktion(3.1) für den Ausgang angegeben wird, liefert die Synthese nahezu die hier vorgestellte Konfiguration.

$$Y = (A \wedge B \wedge C \wedge D) \vee ((A \vee B \vee C \vee D) \wedge Y) \tag{3.1}$$

i3 = OR4	i2 = AND4	i1 = Reset	i0 = Y	Y = init Vektor	Erklärung
0	0	0	0	0	Reset
0	0	0	1	0	Reset
0	0	1	0	0	Auf '0' bleiben
0	0	1	1	0	Wechsel auf '0'
0	1	0	0	0	Reset
0	1	0	1	0	Reset
0	1	1	0	0	dont't care
0	1	1	1	1	dont't care
1	0	0	0	0	Reset
1	0	0	1	0	Reset
1	0	1	0	0	'0' speichern
1	0	1	1	1	'1' speichern
1	1	0	0	0	Reset
1	1	0	1	0	Reset
1	1	1	0	1	Wechsel auf '1'
1	1	1	1	1	Auf '1' bleiben

Tabelle 3.4.: Tabelle für den Initialisierungsvektor für LUT_Y

Der Unterschied besteht darin, dass die mit „don't care" gekennzeichneten Zeilen aus Tabelle 3.4 andere Werte haben. Bei der Synthese der Booleschen Funktion sind die Werte vertauscht. Speichert LUT_Y eine '1' (i0 = 1) wird der Ausgang auf '0' gesetzt. Speichert LUT_Y eine '0' (i0 = 0) verändert es den Ausgang auf '1'. Wenn eine LUT schneller schalten würde als die andere und die Eingangskombination i3 = 0 und i2 = 0 kurzzeitig anliegen würde, könnte das zu einem Fehlverhalten führen.
Der VHDL Code für ein Muller C-Element mit 4 Eingängen befindet sich im Anhang F.3.

3.3. Delay Chain

Bei asynchronen Schaltungen mit Bundled Data Protokollen haben Delayelemente eine entscheidende Funktion. Sie sorgen dafür, dass die Latches nur Werte der kombinatorischen Logik übernehmen, die sicher anliegen. Darüber hinaus stellen die Delayelemente sicher, dass keine Hazards des Muller C-Elements durch schnelle Eingangswechsel entstehen.
Die Laufzeit der kombinatorischen Logik bestimmt in herkömmlichen synchronen Schaltkreisen die Taktgeschwindigkeit. Dabei wird der langsamste Pfad zwischen zwei getakteten Registern betrachtet und die Verzögerung dieses Pfades als globaler Takt genommen. Bei Bundled Data Pipelines hingegen wird jede Verzögerung zwischen zwei Registerstufen indivi-

duell betrachtet und in dem Delayelement abgebildet. In Abbildung 2.7 ist zu sehen, dass ein Delayelement in die Request Leitungen integriert ist. Dadurch verzögert sich der Request an die nachfolgende Pipelinestufe, und die kombinatorische Logik im Datenpfad hat Zeit, ihre Berechnung zu beenden. Die minimale Verzögerung des Delayelements muss die maximale Verzögerung des Datenpfades abdecken.
In integrierten Schaltungen bestehen Delayelemente aus hintereinander geschalteter kombinatorischer Logik. Die Verzögerung dieser Delay-Lines ist angepasst an die worst-case Verzögerung des zugehörigen Datenpfades. Durch die Hinzunahme oder Reduzierung von Logikgattern, passt man die Verzögerung der Delay-Lines des Datenpfades an. Bei FPGAs wird eine Delay-Line realisiert, indem LUTs aneinandergereiht werden. Die Verzögerung ergibt sich aus der Laufzeit der LUTs und der Verbindungen zwischen den LUTs [Las08; Asp].
Beim 4 Phasen Bundled Data Protokoll muss die Verzögerung auf einer steigenden Signalflanke der Requestleitung erfolgen. Wenn die Verzögerung auch auf der fallenden Signalflanke erfolgen würde, würde das System unnötig verlangsamt werden. Die LUTs der Delay-Line sind als UND konfiguriert und haben an einem Eingang alle das gleiche Signal. Am zweiten Eingang wird das Verzögerungssignal durchgeschaltet. Durch diese Konfiguration hat die Delay-Line die gewünschte Eigenschaft, dass die steigende Flanke verzögert wird. Die fallende Flanke dagegen hat nur die Verzögerung der letzten LUT. Elemente die diese Eigenschaft besitzen werden als asymmetrische Delay Lines bezeichnet [BOF10].
Ein Synthesetool das eine solche Aneinanderreihung von LUTs nicht im Zuge der Optimierung entfernen würde, wäre heute sicher nicht mehr konkurrenzfähig. Bei Xilinx ISE und Alteras Quartus 2 gibt es die Option, entweder auf Fläche oder auf Geschwindigkeit zu optimiert. Es ist aber nicht möglich die Optimierung komplett zu unterbinden. Damit die Delay-Line nicht wegoptimiert wird, muss im HDL Entwurf das KEEP Constraint [Xil08, s. 153] angewendet werden. Das KEEP Constraint bewirkt, dass bei der Synthese keine Komponenten zusammengefasst werden. Dadurch wird die Optimierung umgangen und es werden alle UND Gatter in LUTs abgebildet und miteinander verbunden. Abbildung 3.7 zeigt das Resultat der Synthese. Die UNISIM Bibliothek enthält auch LUTs mit 2 Eingängen, diese werden auf dem FPGA später auf LUTs mit 4 Eingängen abgebildet. Die LUTs sind wie oben schon erwähnt als UND konfiguriert. Die Verzögerung der Delay-Line ist auch abhängig von der Laufzeit der Verdrahtung zwischen den LUTs. Für gewöhnlich hat man keinen Einfluss auf die Platzierung der LUTs innerhalb eines FPGAs. Selbst eine kleine Änderung am Design kann bei einer erneuten Synthese eine völlig andere Platzierung durch den Place and Route Algorithmus hervorrufen. Daraus resultiert, dass die Verzögerungen schwer vorhersagbar sind.
Eine Möglichkeit dieses Problem zu lösen ist die Verwendung von LOC Constraints. LOC

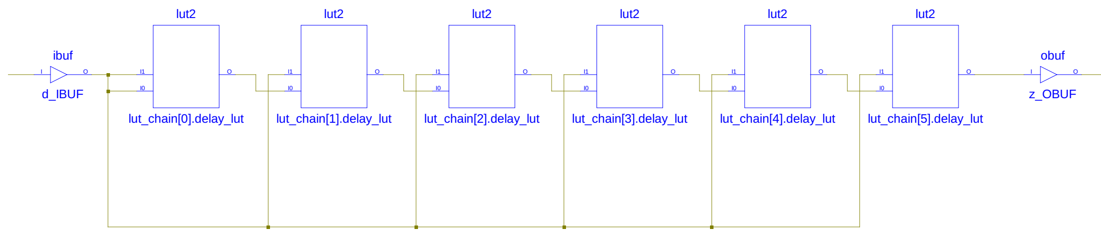


Abbildung 3.7.: Asymmetrisches Delayelement mit LUTs

Constraints bewirken, dass Grundbauelemente des FPGAs, wie LUTs oder Flipflops, fest platziert werden. Je mehr LOC Constraints jedoch verwendet werden, desto schlechter ist die Optimierung des Datenpfades. Des Weiteren benötigt die Synthese mehr Zeit. Bei dem Aspida Projekt [Asp] wird dieses Problem gelindert, indem definiert wird, in welchen Bereich des FPGA die LUTs einer Delay-Line platziert werden sollen. Erreicht wird das durch ein „areaGroup“-Constraint. Je kleiner der Bereich ist, desto genauer ist die Verzögerung vorhersagbar. Ein Vorteil dieser Methode ist, dass keine LOC Constraints verwendet werden.
Ein anderer Ansatz wird in [Las08] beschrieben. Die Delay-Line wird in dieser Arbeit mit dem RLOC Constraint relativ platziert. Dabei ist der Abstand der Verdrahtungen der LUTs fest vorgegeben. Wo die LUTs der Delay-Line platziert werden, entscheidet der Place and Route Algorithmus. RLOC Constaints können im HDL Entwurf gesetzt werden. Durch die Platzierung der gesamten Delay Line muss der Entwurf nicht an den verwendeten FPGA angepasst werden. Die Timingsimulation einer Schaltung geht von der maximalen Laufzeit der Grundbauelemente aus. Die maximale Laufzeit eines Spartan 3e ist damit definiert. Die minimale Laufzeit hingegen wird weder in der Timingsimulation berücksichtigt noch in dem Datasheet definiert. Bei den Implementierungen, die in dieser Arbeit noch vorgestellt werden, korrespondiert die Timingsimulation immer mit dem Verhalten auf dem FPGA. Es ist jedoch nicht auszuschließen, dass bei einem anderen FPGA, auch der gleichen Bauart, Fehler durch die Delay-Line verursacht werden könnten. In diesen Fall muss die Anzahl der LUTs angepasst werden.

3.4. Delay Shift

Die im letzten Abschnitt vorgestellten Methoden haben den Nachteil, dass die Verdrahtungen zwischen den LUTs einen bedeutenden Anteil der Gesamtverzögerung ausmachen. RLOC Constraints können das Problem lindern aber nicht lösen. Die Timing Simulation zeigt, dass die Verzögerung einer Delay Line variiert, je nachdem wo sie platziert wurde [Las08]. Darüber

hinaus stoppt der Implementationsprozess[1], wenn es nicht mehr möglich ist, die Delay Line als Ganzes zu platzieren. Das geschieht häufiger, je größer das Design. Ein weiterer Nachteil ist der erhöhte Bedarf an Ressourcen. Für eine Verzögerung von 10 ns werden 11 bis 12 LUTs benötigt.
Um diesen Problemen zu begegnen, wird in dieser Arbeit eine neue Methode vorgestellt, Signale zu verzögern. Es wird eine LUT zum Oszillieren gebracht und das entstandene Signal als Takt an ein Schieberegister angelegt. Eingang und Ausgang des Schieberegisters sind zugleich Eingang und Ausgang des Delayelements. Die Anzahl der Bits des Schieberegisters bestimmt die Länge der Verzögerung. Ein einfacher Oszillator lässt sich mit einer ungeraden Anzahl von Invertern aufbauen. Dabei werden alle Inverter in Reihe geschaltet und der Ausgang des letzten mit dem Eingang des ersten Inverters verbunden. Die Frequenz, mit der dieser Oszillator schwingt, ist abhängig von der Gatterlaufzeit der einzelnen Inverter. Um einen solchen Oszillator auf einen FPGA zu übertragen, muss eine LUT als Inverter konfiguriert werden. Es wird ein Eingang einer LUT invertiert und der Ausgang an diesen Eingang rückgekoppelt. Die Rückkopplung geschieht wie in Abschnitt 3.2.1 beschrieben. Damit die LUT nicht ständig oszilliert, wird ein weiterer Eingang genutzt, um den Ausgang auf '0' zu setzten. Wenn das Delayelement nicht arbeitet, also kein Signal verzögert werden muss, sorgt der zweite Eingang dafür, dass der Ausgang auf '0' ist und die LUT nicht oszilliert. Dadurch wird sichergestellt, dass der Zeitpunkt der ersten Flanke des Taktsignals definiert ist. Darüber hinaus wird durch das Stoppen des Oszillators und der damit verbundenen Schaltvorgänge nicht unnötig Energie verbraucht.

i1 = Stopp	**i0 = Rückkopplung**	**init Vektor = Ausgang**
0	0	0
0	1	0
1	0	1
1	1	0

Tabelle 3.5.: Tabelle für den Initialisierungsvektor des Oszillators

Das Schieberegister ist mit Flipflops aufgebaut die in den Slices enthalten sind. Der Reset der Flipflops ist mit dem invertierten Eingangssignal des Delayelements verbunden. Dadurch hat das Element ein asymmetrisches Verhalten (Abschnitt 3.3). Der Eingang des Schieberegisters ist fest mit '1' verbunden, weil die Flipflops nur Werte speichern, wenn an dem Delayelement eine '1' anliegt. Das Schaltbild des Delay Shift-Elements ist in Abbildung 3.8 dargestellt. Die logische Funktion für den Oszillator ist die Inhibition. In der Wahrheitstabelle 3.5 ist der

[1]Der Implementationsprozess besteht aus: Synthetisieren, Übersetzen, Abbilden und Platzieren & Verbinden.

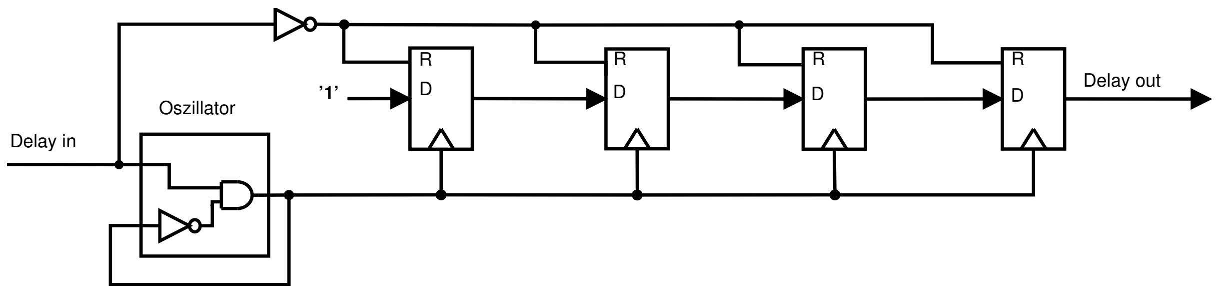


Abbildung 3.8.: Delayelement aus einem Schieberegister

Initialisierungsvektor für die Konfiguration der LUT ablesbar. In Anhang F.6 ist der VHDL Code für das Delay Shift-Element enthalten, sowie das Resultat der Synthese.
Die „Post-Place and Route“-Simulation in Abbildung 3.9 zeigt, dass der Oszillator mit einer Frequenz von 475,7 MHz schwingt. Die Verzögerung beginnt, wenn das Reset Signal den Wert '0' annimmt. Der Ausgang des letzten Flipflops ist zugleich der Ausgang des Delay Shift-Elements. Die Verzögerung wird nicht zwischen dem Ein- und Ausgang abgelesen, da die Simulation die Latenzzeit der Input und Output Buffer des FPGAs berücksichtigt.

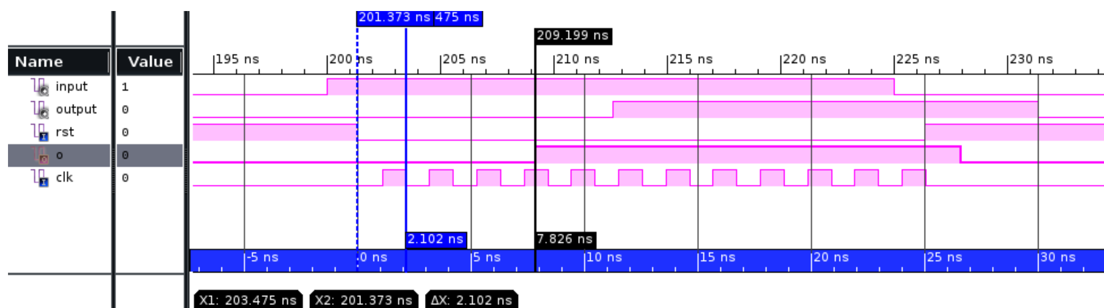


Abbildung 3.9.: Timing Simulation des Delay Shift Elements

Die Simulation zeigt auch das asymmetrische Verzögerungsverhalten. Wenn das Reset Signal auf '1' wechselt, geht das Ausgangssignal des letzten Flipflops innerhalb von 1,6 ns auf '0'. In dem in Abbildung 3.10 dargestellten Diagramm sind die Simulationsergebnisse der Verzögerung des Delay Shift-Elements dargestellt. Die Simulationen wurden mit Schieberegistern mit einer Länge von 1 bis 12 Bit durchgeführt. Im Vergleich zu der in Abschnitt 3.3 vorgestellten Methode ist die Verzögerung des Delay Shift-Elements linearer im Verhältnis zu der Anzahl der Bits.

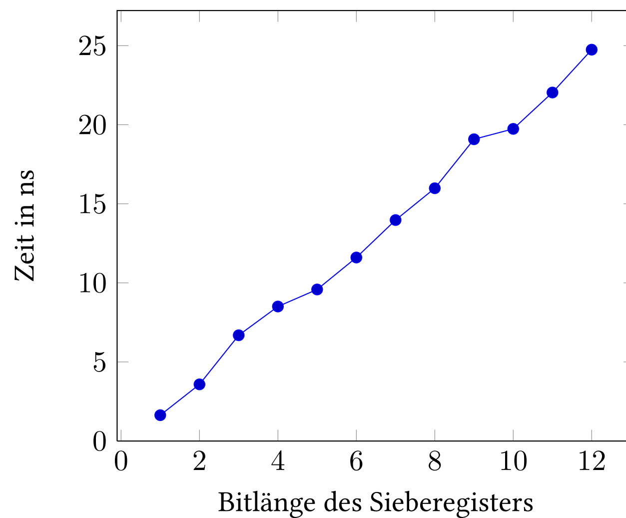


Abbildung 3.10.: Diagramm Schiebeoperationen Zeit

3.5. Pipelines

Mit den Komponenten, die in den letzten Abschnitten vorgestellten wurden, ist es bereits möglich, einen Kontrollpfad für eine asynchrone Pipeline auf einem FPGA zu realisieren. Grundsätzlich gilt, dass Registerstufen auf FPGAs mit den Flipflops aufgebaut werden, die in den Slices enthalten sind. Diese Flipflops können entweder als flankengesteuerte D-Flipflops oder als einfache RS-Latches konfiguriert werden. Die Registerstufen einer einfachen, nicht entkoppelten 4-Phasen Bundled-Data Pipeline bestehen aus RS-Latches. Das wirkt sich bei einer Implementierung als IC positiv auf den Platzbedarf aus. Bei der Realisierung einer asynchronen Pipeline auf einem FPGA hat es hingegen keine Auswirkung auf den Ressourcenverbrauch, Latches oder Flipflops zu verwenden.

3.5.1. Nicht entkoppelte Pipeline

Die einfache Pipeline in Abbildung 3.11 ist eine dreistufige Pipeline. Der Datenpfad ist trivial und dient lediglich zur Veranschaulichung des Datentransports. Das erste Register lädt den inkrementierten Wert des letzten Registers, und das zweite und dritte Register lädt den Wert ihres jeweiligen Vorgängers. Die Registercontroller[2] haben einen internen Aufbau wie er in der Abbildung 2.10 dargestellt ist. Wie in Abschnitt 2.3.4 beschrieben, sind die Registercontroller nicht entkoppelt. Wenn bei einer Pipeline die erste Stufe das Datum der letzten Stufe benötigt,

[2]In der Literatur wird meistens der Begriff „Latch Controller“ benutzt. In dieser Arbeit werden die Register aus flankengesteuerten Flipflops aufgebaut. Im Folgenden wird daher der Begriff Registercontroller verwendet.

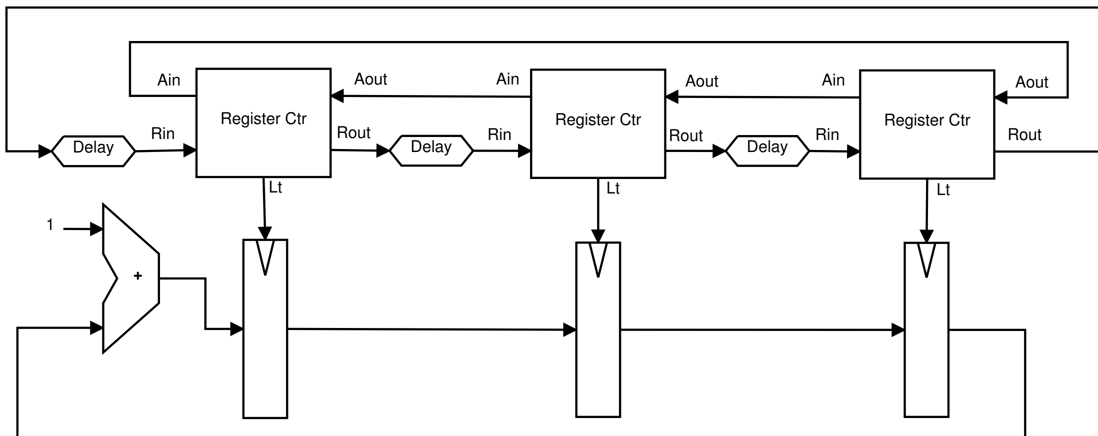


Abbildung 3.11.: Einfache Pipeline

die Pipeline also als Ring angeordnet ist, müssen bei nicht entkoppelten Pipelines mindestens 3 Stufen vorhanden sein. Bei weniger als 3 Stufen wird die Pipeline ihre Handshakes nicht fortsetzen (siehe Abschnitt 2.3.4). Nicht entkoppelte Registercontroller bestehen aus einem Muller C-Element und einem Inverter und können daher mit einer LUT realisiert werden. Der Initialisierungsvektor des Muller C-Elements aus Abschnitt 3.2.1 wird so angepasst, dass der Eingang für Aout invertiert ist. Der Sourcecode und die Tabelle für den Initialisierungsvektor sind in Anhang F.3.1 enthalten.
Die Simulation in Abbildung 3.12 zeigt die Handshakekommunikation der Registercontroller. Das Muller C-Element der letzten Pipelinestufe ist nach einem Reset mit '1' initialisiert. Dadurch ist der Request Ausgang auch '1' und es liegt am Request Eingang Rin des ersten Registercontrollers ebenfalls eine '1' an. Der Handshake setzt sich fort bis der letzte Controller erneut einen Request an den ersten Controller sendet. Die Signale für die Steuerung der Register sind in dem Vektor „register_ctr“ zusammengefasst. Es ist zu erkennen, dass die Register ihre Werte nach einer nacheinander übernehmen und nicht gleichzeitig, wie bei getakteten synchronen Pipelines. Die Reihenfolge lautet reg1, reg2, reg3, reg1 usw. Die Simulation zeigt auch das Verhalten einer nicht entkoppelten Pipeline.

3.5.2. Entkoppelte Pipeline

Im Folgenden wird eine Pipeline mit entkoppelten Registercontrollern vorgestellt. Der STG in Abbildung 3.13 definiert das Kommunikationsverhalten eines Registercontrollers, der voll entkoppelt ist. Der STG wurde in [FD96] sowie in [Liu97] vorgestellt. Es wird der Workflow für Petrify angewendet der in Abbildung 2.22 dargestellt ist, um aus dem STG ein Verilog

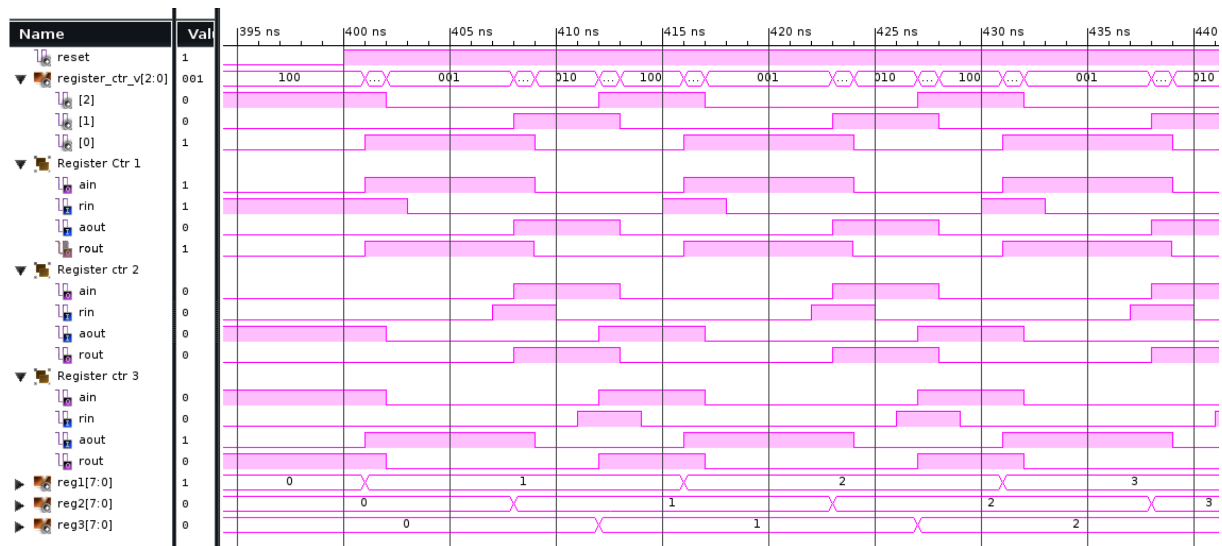


Abbildung 3.12.: Simulation der einfachen Pipeline

Modul zu erhalten.
Die Muller C-Elemente können nach einem Reset entweder mit '1' oder mit '0' initialisiert werden. Damit ein Registercontroller anfängt mit seinen Nachbarn zu kommunizieren, müssen bei den Komponenteninstantiierungen der Muller C-Elemente die Anfangswerte angegeben werden. Die Anfangswerte der Signale sind am Ende der Verilog Datei als Kommentar aufgelistet und entsprechen der Anfangsmarkierung des STGs.
Abbildung 3.14 zeigt eine zweistufige Pipeline. Um ein neues Datum zu erhalten addiert der Datenpfad der Pipeline ein altes Datum zu seinen Vorgänger $f_n = f_{n-1} + f_{n-2}$. Das erste Register ist mit 1 initialisiert. Wenn der Datenpfad synchron implementiert wäre, könnte man in dem ersten Register die Fibonacci Folge ablesen $(1, 1, 2, 3, 5, 8, 13, 21, ...)$. Die nacheinander erfolgende Übernahme der Werte (siehe Abschnitt 3.5.1) stellt kein Problem dar, solange die Register keine Abhängigkeiten untereinander haben. In dem vorliegenden Beispiel aus Abbildung 3.14 sind jedoch Abhängigkeiten vorhanden. Damit ein neues Datum für Register 1 berechnet werden kann, benötigt die kombinatorische Logik das Datum aus Register 1 und Register 2. Werden diese Abhängigkeiten nicht berücksichtigt, liefert diese Pipeline nicht die Fibonacci Folge, sondern quadriert den Wert des ersten Registers. In der Simulation in Abbildung 3.15 ist dieses Verhalten zu erkennen. Wenn Register 1 ein neues Datum übernimmt, hat Register 2 bereits dieses übernommen.
Die Simulation zeigt auch ein entkoppeltes Verhalten der Registercontroller. Es beginnt eine

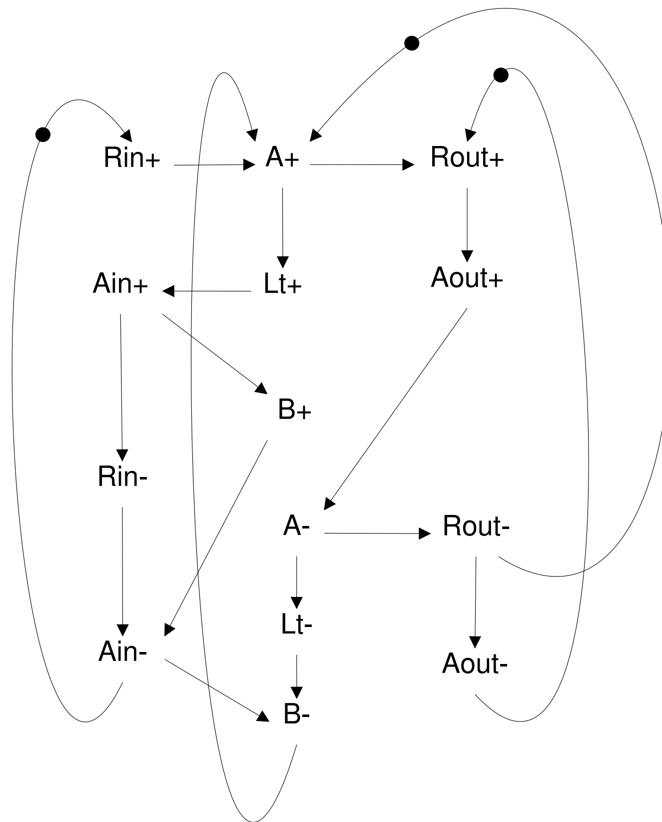


Abbildung 3.13.: STG eines Entkoppelten Controllers [FD96]

neue Kommunikation, bevor die vorherige Kommunikation mit dem Setzen des Acknowledge auf '0' auf der Ausgangsseite (Aout) komplettiert wurde.

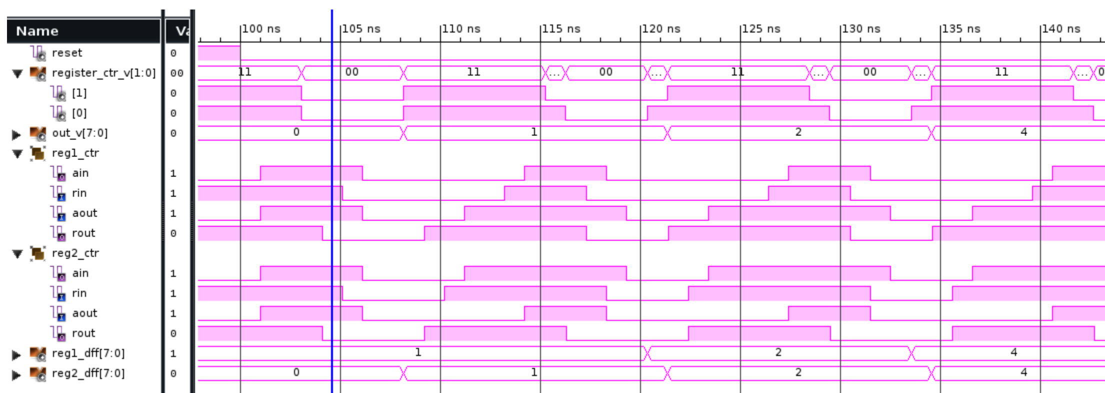


Abbildung 3.15.: Simulation der Pipeline aus Abbildung 3.14

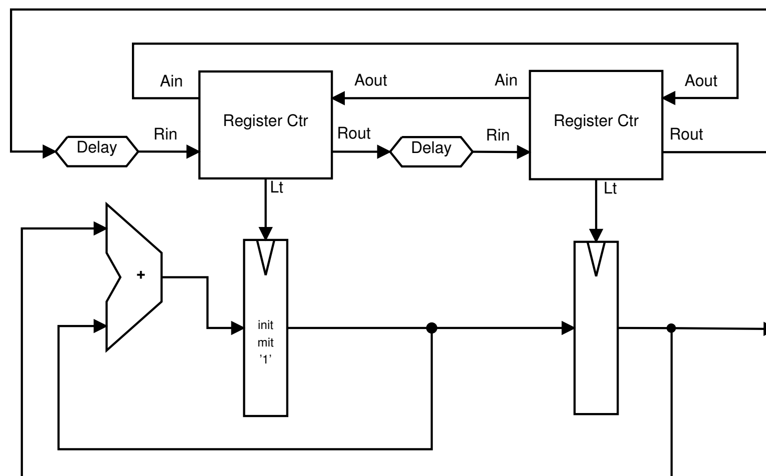


Abbildung 3.14.: Pipeline mit Registercontrollern die entkoppelten sind

3.5.3. Doubly-Latched Pipeline

Die Probleme die durch Abhängigkeiten unter den Registern entstehen, wurden in [KG97] gelöst. In der Arbeit wird ein entkoppelter Registercontroller vorgestellt, mit dem Pipelines aufgebaut werden können, die nahezu wie synchrone Pipelines arbeiten. Wenn alle Delays zwischen den Pipelinestufen ähnlich lang sind, sind alle Abhängigkeiten aufgelöst. Das wird erreicht, indem jedes Register einer Pipeline in ein Master und in ein Slave Register aufgeteilt wird. Hat eine Pipelinestufe ihre Berechnungen abgeschlossen, werden die neuen Werte im Master Register übernommen, während in dem Slave Register die alten Werte gespeichert bleiben. Der Inhalt des Master Registers wird gleichzeitig von allen Pipelinestufen in die Slave Register übernommen. Diese Pipelines werden DLAPs genannt. Eine DLAP Pipelinestufe ist in Abbildung 3.16 dargestellt. Der STG in Abbildung 3.17 beschreibt das Verhalten des DLAP Registercontrollers.
In dem ursprünglichen Entwurf in [KG97] sind „Done"-Signale für die Register vorhanden (D_m, D_s). Diese Signale zeigen dem Registercontroller, dass die Register ihre Daten gespeichert haben. Eine Timingsimulation zeigt, dass die Done-Signale nicht nötig sind, weil die Laufzeiten der Muller C-Elemente des Registercontrollers ausreichen, um die Daten zu übernehmen. Für die Synthese mit Petrify ist daher der ursprüngliche Entwurf aus [KG97] angepasst worden. Die Signale D_m und D_s sind als intern deklariert und werden dadurch bei der Synthese wegoptimiert.
Abbildung 3.18 zeigt das Resultat der Synthese. Der DLAP Registercontroller besteht aus 3 Muller C-Elementen und 3 Invertern. Der Controller kann auf einem FPGA mit 3 LUTs

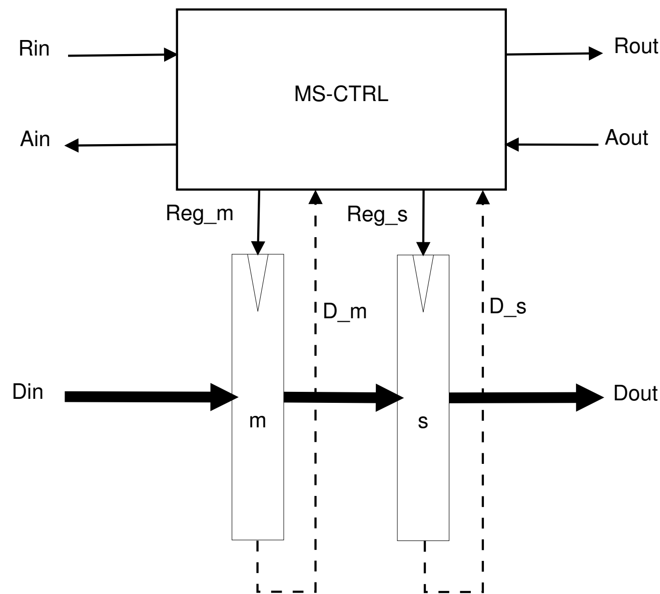


Abbildung 3.16.: DLAP Pipelinestufe [KG97]

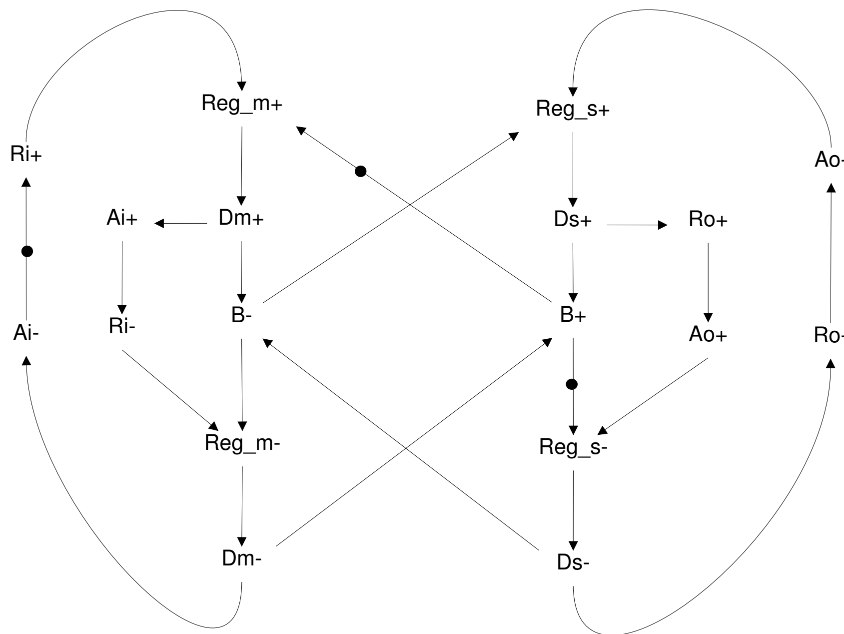


Abbildung 3.17.: STG eines Master-Slave Registercontrollers [KG97]

realisiert werden. Es müssen die Initialisierungsvektoren der Muller C-Elemente angepasst werden (siehe Abschnitt 3.5.1). Wenn die Pipeline aus 3.14 als DLAP aufgebaut ist, verhält sie sich wie eine synchrone Pipeline. Die Simulation in Abbildung 3.19 zeigt, dass die Fi-

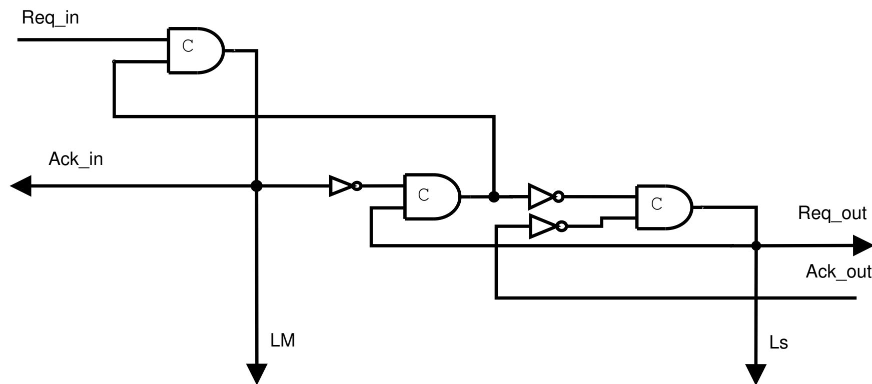


Abbildung 3.18.: Schaltbild des Master-Slave Registercontrollers

bonacci Folge im ersten Register abgelesen werden kann. Wie eingangs schon erwähnt, ist

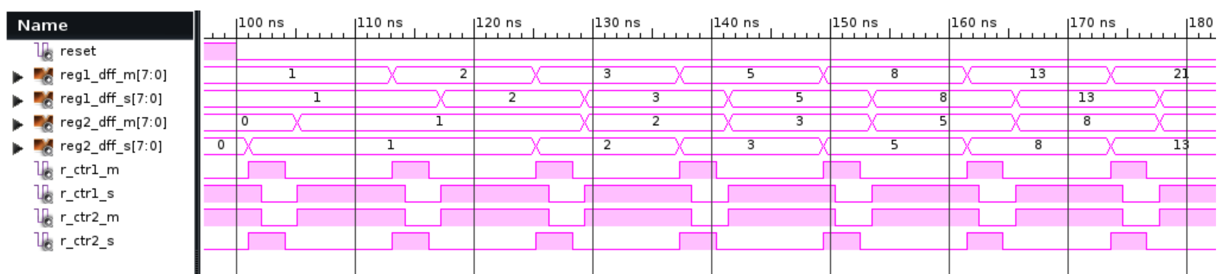


Abbildung 3.19.: Simulation der DLAP Implementierung des Beispiels aus Abbildung 3.14

eine Voraussetzung für die korrekte Funktion, dass die Delay-Elemente ähnlich lang sind. Um diese Einschränkung zu vermeiden wird in [FEH10] eine Lösung präsentiert, die alle Abhängigkeiten der Pipelinestufen aufhebt. Für jeden Registercontroller wird vor dem Req_in und vor dem Ack_out ein Muller C-Element wie in Abbildung 3.20 platziert. Dadurch werden die Pipelinestufen synchronisiert. Jedes neue Datum eines Registers kann aus jedem Datum der vorhandenen Register berechnet werden, unabhängig von der Länge der Delay-Elemente. Mit dieser Lösung ist es möglich, synchrone in asynchrone Schaltungen umzuwandeln, ohne die Abhängigkeiten der Register zu berücksichtigen. Ein großer Nachteil dabei ist, dass auch Abhängigkeiten berücksichtigt werden, die nicht vorhanden sind. Die Vorteile einer Bundled Data-Pipeline, die durch die Elastizität entstehen, werden bei diesem Ansatz aufgehoben. Alle Pipelinestufen synchronisieren sich mit dem längsten Delay-Element. Die Vorteile des geringeren Energieverbrauchs im Leerlauf sowie die bessere EMV bleiben hingegen bestehen.

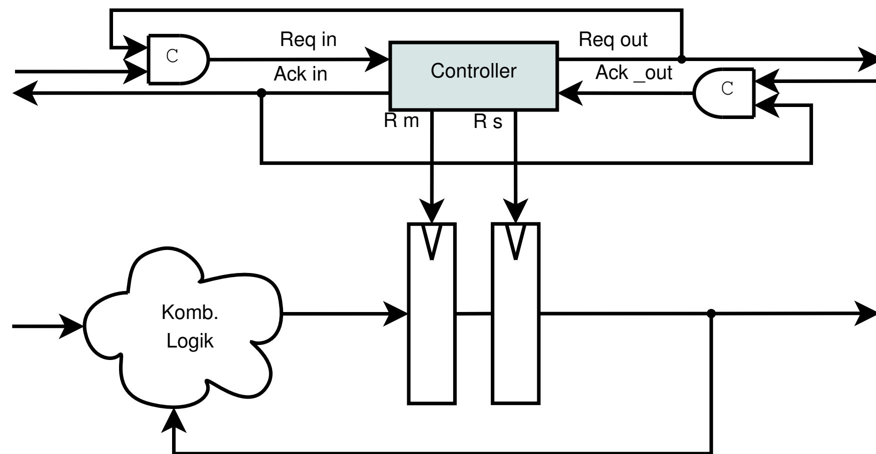


Abbildung 3.20.: Synchronisation einer DLAP nach [FEH10]

3.5.4. Entkoppeln von Abhängigkeiten

Um bei Schaltungen mit dem Boundled Data-Protokoll Abhängigkeiten unter den Registern individuell aufzulösen, werden Fork- und Join-Elemente verwendet (vgl. Abschnitt 2.3.5). In [KG97] werden diese Elemente in die Registercontroller integriert. In Abbildung 3.21 und 3.22 sind die Implementierungen dieser Registercontroller zu sehen. Die Implementierung ähnelt der des Registercontrollers aus Abbildung 3.18. Bei dem Fork wird die Synchronisierung der beiden Ack_out Eingänge durch ein Muller C-Element mit 3 Eingängen realisiert. Bei dem Join werden die beiden Req_in Eingänge ebenfalls mit einem Muller C-Element mit 3 Eingängen synchronisiert.

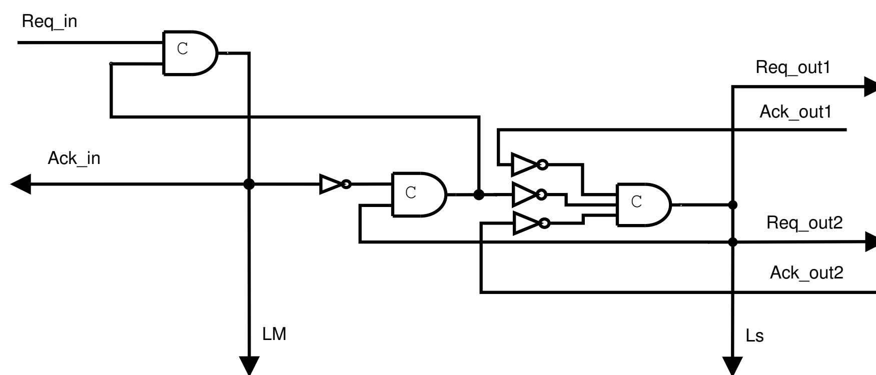


Abbildung 3.21.: Implementierung eines DLAP Registercontrollers als Fork

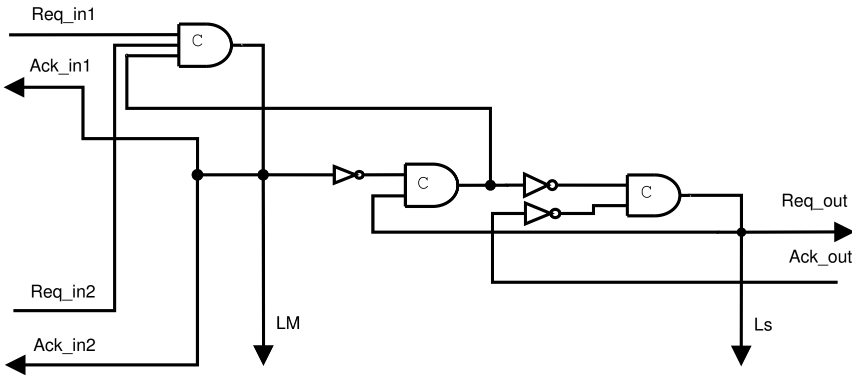


Abbildung 3.22.: Implementierung eines DLAP Registercontrollers als Join

4. VHDL Konvertierung asynchroner Schaltungen

Die Herausforderung bei der Konvertierung einer synchrone in eine asynchrone Schaltung mit dem Bundled Data-Protokoll ist das Erstellen des Kontrollpfades. In diesem Kapitel wird beschrieben, wie eine synchrone VHDL Beschreibung in eine asynchrone konvertiert werden kann. Anschließend wird eine konkrete Konvertierung eines MIPS Prozessors vorgestellt und das Resultat mit dem ursprünglichen synchronen Entwurf verglichen.

4.1. Konvertierungsablauf

Der Konvertierungsablauf gliedert sich wie folgt:

- Erstellen des Kontrollpfades
- Komponenteninstanziierung des Kontrollpfades in der Top-level Entity
- Anpassen der synchronen Prozesse im Datenpfad und Austausch des Taktsignals mit den Steuersignalen des Kontrollpfades
- Feinabstimmung der Delay-Elemente

Die Top-level Entity eines hierarchischen VHDL Entwurfs ist die Entity, in der alle Komponenteninstantiierungen vorgenommen werden.

4.1.1. Kontrollpfad

Der Kontrollpfad besteht aus Registercontrollern und Delay-Elementen die miteinander verbunden sind. Jedes Register oder jede Registergruppe erhält einen Registercontroller. Um bestimmen zu können, welcher Typ von Registercontroller geeigneter ist und wo Forks und Joins einzufügen sind, ist es notwendig, die Verbindungen der Register zu analysieren. Hierzu muss der kombinatorische Pfad zwischen den Registern verfolgt werden. Bei komplexeren Schaltungen bietet es sich an, einen Abhängigkeitsgraphen zu erstellen, um eine Übersicht zu

erhalten, wo Forks und Joins einzufügen sind. In diesem Graphen wird jedes Register als Kreis dargestellt, und der kombinatorische Pfad zwischen den Registern wird als Pfeil eingezeichnet. Wenn die kombinatorische Logik vor einem Register einen neuen Wert aus zwei oder mehreren

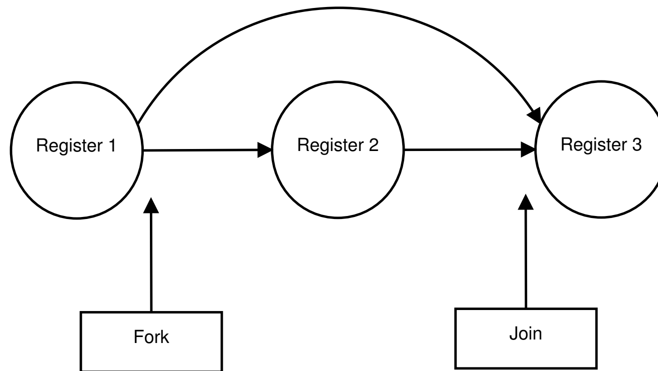


Abbildung 4.1.: Abhängigkeitsgraph mit einem Fork und Join

Registern berechnet, so muss ein Join vor dem Registercontroller eingefügt werden. Wenn hingegen der gespeicherte Wert eines Registers von mehreren Registern benötigt wird, ist ein Fork hinter dem Registercontroller einzufügen. Abbildung 4.1 zeigt einen Graphen aus dem ersichtlich wird, wo ein Fork und wo ein Join in den Kontrollpfad zu integrieren ist. Wenn vor einem Registercontroller ein Join oder nach einem Registercontroller ein Fork platziert ist, müssen DLAP Registercontroller verwendet werden. Ist ein Register frei von einem Fork oder einem Join, können einfache entkoppelte Registercontroller verwendet werden.
Zwischen den Requestleitungen der einzelnen Registercontroller werden Delay-Elemente eingefügt. Die Delay-Elemente werden vorerst mit einer langen Verzögerung konfiguriert. Beim ersten Synthesedurchgang kann man dadurch die korrekte Funktion der Schaltung verifizieren. Die Konfiguration der Delay-Elemente wird im letzten Schritt der Konvertierung an die Laufzeit des jeweiligen Datenpfades angepasst. In der Top-Level Entity werden die Steuersignale deklariert und bei der Instantiierung des fertigen Kontrollpfades mit diesen verbunden.

4.1.2. Datenpfad

Die ursprüngliche Konvertierungsidee des Datenpfades ist, dass lediglich das Taktsignal der Register mit den Steuersignalen der Registercontroller ausgetauscht wird. Wenn Joins und Forks vorhanden sind, müssen DLAP Registercontroller verwendet werden. Die Register im Datenpfad müssen dann verdoppelt und in ein Masterregister und in ein Slaveregister aufgeteilt werden. Der Eingang des Masterregisters erhält den Eingang des ursprünglichen Registers. Der

Eingang des Slaveregisters wird mit dem Ausgang des Masterregisters verbunden. Die beiden Register werden mit dem zugehörigen Steuersignal des DLAP Registerkontrollers verbunden.

```
RegisterName : process(Clk,Reset)
begin
   if reset = '0' then
      Ausgangssignal <= (others => '0');
   elsif rising_edge(Clk) then
      Ausgangssignal <= Eingangssignal
   end if;
end process;
```

Listing 4.1: VHDL Implementierung eines Registers

```
RegisterName_m : process(Reg_m,Reset)
begin
   if reset = '0' then
      Signal_Master <= (others => '0');
   elsif rising_edge(Reg_m) then
      Signal_Master <= Eingangssignal
   end if;
end process;

RegisterName_s : process(Reg_s,Reset)
begin
   if reset = '0' then
      Ausgangssignal <= (others => '0');
   elsif rising_edge(Reg_s) then
      Ausgangssignal <= Signal_Master
   end if;
end process;
```

Listing 4.2: DLAP Registers

Der Code in Listing 4.1 zeigt die Implementierung eines einfachen Registers in VHDL. Der angepasste VHDL Code eines Register für einen DLAP Registercontroller ist in Listing 4.2 abgebildet. Bei einem hierarchischen VHDL Entwurf müssen die Entities angepasst werden, in denen der taktsynchrone Prozess beschrieben ist. Anstelle des Taktsignals müssen die Steuersignale in der Entity deklariert werden. In der Top-Level Entity müssen bei der Instantiierung entsprechend die Steuersignale angegeben werden.

4.1.3. Delay Matching

Damit die konvertierte Schaltung effizient arbeitet, müssen die Delay-Elemente an die maximale Laufzeit des kombinatorischen Pfades angepasst werden. Die Laufzeiten aller kombinatorischen Pfade sind bei Xilinx in der *.dly Datei zusammengefasst. Die Dimensionierung der Delay-Elemente erfolgt indem der langsamste Pfad abgebildet wird. Die Anzahl der Flipflops für das Delay-Shift können aus den in Abbildung 3.10 dargestellten Diagramm entnommen werden.

4.2. MIPS Prozessor

An dieser Stelle wird die Konvertierung eines MIPS Prozessors beschrieben und das Ergebnis der Konvertierung mit dem Original verglichen. Der Aufbau und Befehlssatz des MIPS Prozessors ist beschrieben in [PH05]. Die VHDL Implementierung des MIPS ist entnommen aus einem Projekt der Luleå University of Technology [Wal00]. Es handelt sich um einen MIPS Modell R2000 mit folgenden Eigenschaften:

- 5 Pipelinestufen
- Eine Operation pro Taktzyklus
- Alle Flipflops sind mit der steigenden Taktflanke gesteuert.
- Unterstütze Befehle: Alle R2000 Befehle außer Multiplikation, Division und Gleitkommaoperationen. Die Formate sind:
  - R-Format: Arithmetische Operation
  - I-Format: bedingter Sprung, Immediate-Format, Datentransport
  - J-Format: unbedingter Sprung
- 32 Register mit 32 Bit
- Forwarding ist implementiert.

Der ursprüngliche Prozessor wird modifiziert, damit das Block RAM auf dem Xilinx FPGA genutzt werden kann. Die Assemblerprogrammierung des MIPS erfolgt mit Hilfe des Programms „Mars“, mit dem es möglich ist, die MIPS Befehle eines Programms als Hex-Datei auszugeben. Die Hex-Datei wird mit einem Java Programm (HEX2VHDL) in ein VHDL Array umgewandelt und kann anschließend in die Instantiierung des Block RAM-Datenspeichers eingefügt werden. Abbildung 4.2 zeigt den Datenpfad des MIPS Prozessors. Die 5 Pipelinestufen sind:

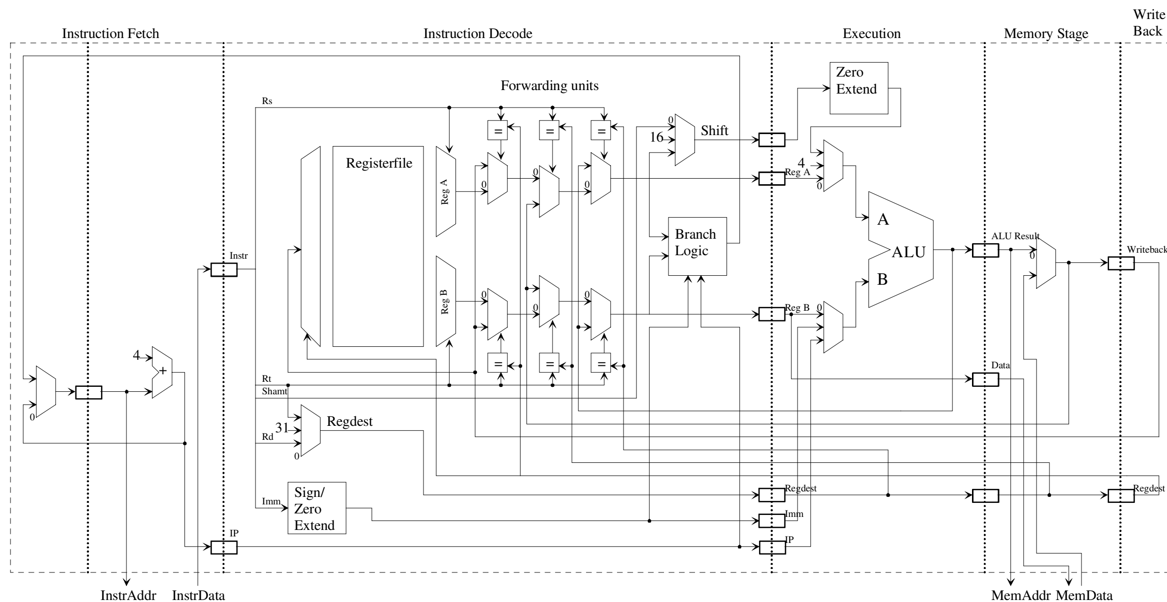


Abbildung 4.2.: Übersicht des MIPS mit den 5 Pipelinestufen [Wal00]

- Instruction Fetch (IF): Instruktion wird aus dem Programmspeicher geladen.
- Instruction Decode (ID): Instruktion wird dekodiert und die Steuersignale werden generiert.
- Execute (EX): Arithmetische und logische Instruktionen werden ausgeführt.
- Memory access (MEM): Speicherzugriff auf Datenspeicher bei Lade- und Speicherinstruktionen.
- Write back (WB): Zurückschreiben des Resultats in das Registerfile.

Die Verbindungen der kombinatorischen Logik zwischen den Pipelineregistern sind in dem Abhängigkeitsgraphen in Abbildung 4.3 verdeutlicht. Die Verbindungen der MEM und WB auf die EX Pipelinestufe resultieren aus der Optimierung durch Forwarding. Mit Hilfe des Abhängigkeitsgraphen wird der Kontrollpfad für den MIPS Prozessor entwickelt. Wenn ein Register für eine Berechnung den eigenen Inhalt benötigt[1], so muss diese Abhängigkeit nicht aufgelöst werden. Der Abhängigkeitsgraph zeigt, dass jede Registerstufe Abhängigkeiten besitzt. Daher werden im Kontrollpfad ausschließlich DLAP Registercontroller verwendet.

[1]Wie zum Beispiel beim “Program Counter“.

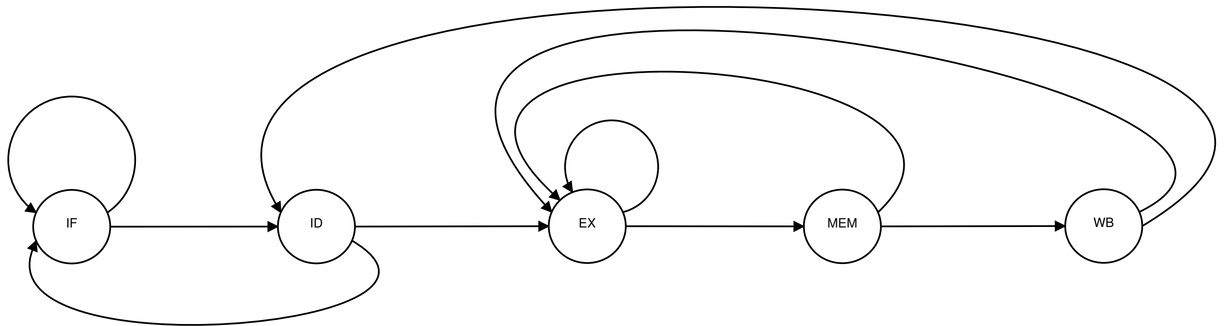


Abbildung 4.3.: Abhängigkeiten der Pipelinestufen des MIPS Prozessors

Die Abhängigkeiten der 5 Registercontroller werden aufgelöst, indem folgende Elemente eingefügt werden: 3 Forks mit 2 Eingängen, ein Join mit 2 Eingängen und ein Join mit 3 Eingängen. Die Verbindungen der Elemente sind in Abbildung 4.4 dargestellt. Der Datenpfad wird so an-

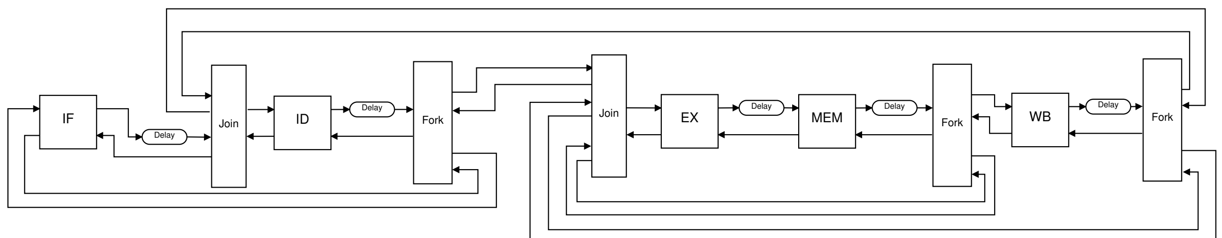


Abbildung 4.4.: Kontrollpfad für den MIPS Prozessor

gepasst wie in Abschnitt 4.1.2 beschrieben. Es werden alle Pipelineregister verdoppelt und mit den entsprechenden Signalen des Kontrollpfades verbunden. Das Taktsignal des Registerfiles wird mit dem Mastersignal des WB-Registerkontrollers ausgetauscht. Der Block RAM für den Progammspeicher wird mit dem Mastersignal des IF-Registercontrollers verbunden und der Block RAM für den Datenspeicher mit dem Mastersignal des MEM-Registercontrollers. Die Analyse der kombinatorischen Logik zwischen den Pipelineregistern ergab folgende worst-case Laufzeiten:

- IF: 7 ns
- ID: 10,5 ns
- EX: 8 ns
- MEM: 7 ns
- WB: 6 ns

4.2.1. Ressourcenverbrauch

Logic Utilization	synchron	asynchron	Mehraufwand
Slice Flipflops	318	594	87%
4-Input LUTs	1706	2418	42%
besetzte Slices	961	1474	53%

Tabelle 4.1.: Ressourcenverbrauch

Durch Tabelle 4.1 wird verdeutlicht, dass die konvertierte asynchrone Version des MIPS Prozessors beträchtlich mehr Logik benötigt als die synchrone Version. Die Verdoppelung der Register hat daran den größten Anteil. Der gesamte Kontrollpfad benötigt 51 LUTs.

4.2.2. Performancevergleich

Wie Abbildung 4.3 zeigt, haben komplexe Schaltungen wie der MIPS Prozessor viele Abhängigkeiten. Insbesondere der Forwarding-Mechanismus koppelt die Pipelineregister stark aneinander. Durch die Forks und Joins führt das dazu, dass der Durchsatz der Pipeline sich an die Verzögerung des längsten Delayelements richtet. Die Performance ist also abhängig von langsamsten Pfad. Die Setup-Zeit des zusätzlichen Registers führt dazu, dass die asynchrone Version langsamer ist als ihr synchrones Pendant.

4.2.3. Bewertung

Es zeigt sich, dass der konvertierte Prozessor langsamer ist als der synchrone Prozessor. Die gleichmäßigere Verteilung der Registeraktivität über die Zeit führt dazu, dass der asynchrone Prozessor wahrscheinlich eine bessere EMV aufweist. Bei dem hier betrachteten Prozessor sind keine Interrupts und auch keine Peripherie wie zum Beispiel Timer implementiert. Wenn der Prozessor im Leerlauf ist, werden Bubbles eingefügt. Bubbles sind Operationen, die zu keiner Zustandsänderung führen. Der Prozessor verbraucht aber trotzdem Energie, weil der Kontrollpfad nach wie vor die Signale für die Register generiert. Würde man den Prozessor durch eine Operation ergänzen, die dafür sorgt, dass der Handshake zwischen der IF- und der ID-Phase nicht weitergeleitet wird, könnte man potenziell bewirken, dass im Leerlauf weniger Energie verbraucht wird.

5. Fazit

5.1. Zusammenfassung

Die vorliegende Arbeit beschreibt einen Ansatz, synchrone in asynchrone Schaltungen zu konvertieren und für einen FPGA zu synthetisieren. Als erstes werden Grundkonzepte asynchroner Schaltungen eingeführt, dabei werden die unterschiedlichen Handshake-Protokolle für die Kommunikation erläutert. Das Bundled-Data Handshake-Protokoll wird näher betrachtet, weil es sich gut für eine Konvertierung eignet. Als Nächstes wird darauf eingegangen wie die Kommunikationsteilnehmer miteinander verknüpft sind und interagieren. Um eine asynchrone Schaltung zu spezifizieren, ist es gebräuchlich STGs zu verwenden. Ein Abschnitt dieser Arbeit spezifiziert STGs und schildert wie man aus einem STG einen HDL-Entwurf synthetisiert.
Basierend auf diesen Grundkonzepten wird beschrieben, wie sich asynchrone Schaltungen auf FPGAs realisieren lassen. Hierzu wird zunächst der Aufbau eines FPGA der Firma Xilinx vorgestellt. Im Anschluss wird erklärt wie spezielle Bauelemente für asynchrone Schaltungen implementiert werden. Ein wichtiges Bauelement ist das Muller C-Element. Es wird dargestellt, wie Muller C-Elemente mit zwei, drei und vier Eingängen auf FPGAs implementiert werden. Ein weiterer Bestandteil asynchroner Schaltungen mit dem Bundled Data-Protokoll sind Delay-Elemente. Es wird beschrieben wie Delay-Elemente bei bisherigen Projekten auf FPGAs implementiert werden.
In Abschnitt 3.8 wird eine neue Methode vorgestellt, Delay-Elemente mit Schieberegistern zu realisieren. Die Idee besteht darin die Schieberegister mit einem lokal erzeugten Takt zu steuern, um zum einen die Verzögerung vorhersagbarer zu machen und zum anderen Ressourcen einzusparen.
Aus den vorgestellten Bauelementen werden Registercontroller aufgebaut. Um die Funktionsweise der Registercontroller zu demonstrieren, werden einfache Pipelines aufgebaut. Dabei wird auch auf Probleme mit Abhängigkeiten unter den Registern eingegangen und darauf, wie diese Probleme gelöst werden können.
Darauf aufbauend werden die einzelnen Schritte des Konvertierungsablaufs beschrieben. Ab-

schließend wird ein MIPS Prozessor konvertiert und das Ergebnis mit der ursprünglichen synchronen Implementierung verglichen.

5.2. Resümee

Ein zentrales Ergebnis dieser Arbeit besteht in der Erkenntnis, dass es möglich ist, komplexe synchrone Schaltungen in asynchrone Schaltungen zu konvertieren und auf FPGAs zu laden. Ein weiteres Ergebnis besteht in der praktischen Erkenntnis, dass dies zumindest bei der hier vorgestellten Konvertierung mit einem erheblich größeren Logikverbrauch einhergeht. Es bieten sich für dieses Ergebnis eine mögliche Erklärungen an: Die Abhängigkeiten zwischen Registern führen zu einem Mehrverbrauch an Logik und verursachen auch eine schlechtere Performance.
Bei der Konvertierung einer einfachen Pipeline ohne klare Abhängigkeiten brauchen keine DLAP-Registercontroller verwendet werden, und es müssen auch keine Joins und Forks eingefügt werden. Solche Schaltungen hätten das Potenzial schneller zu sein als ihre synchronen Versionen.

5.3. Ausblick

Um die Vorteile asynchroner Schaltungen auch auf FPGAs zu verdeutlichen, wäre es nötig Untersuchungen der EMV und des Energieverbrauchs vorzunehmen. Diese Untersuchungen könnten die Akzeptanz asynchroner Systeme vergrößern oder sogar in manchen fällen die Notwendigkeit unterstreichen asynchronen Systeme zu verwenden.

Anhang

A. Literatur

[Alt] *Altera Design Flow for Xilinx Users*. Altera. 2013.

[Asp] *Asynchronous Open-Source Processor IP of the DLX Architecture*. abgerufen am 10.08.2014. 2004. URL: http://opencores.org/project,aspida.

[BOF10] Peter A. Beerel, Recep O. Ozdag und Marcos Ferretti. *A Designer's Guide to Asynchronous VLSI*. Cambridge University Press, 2010.

[Chu87] Tam-Anh Chu. "Synthesis of Self-timed VLSI Circuits from Graph-theoretic Specification". Diss. Massachusetts Institute of Technology, 1987.

[Col] *asynch.genlib*. 2006. URL: http://www.cs.columbia.edu/~cs6861/sis/sis1.3/sis/sis_lib/asynch.genlib.

[Cor+97] Jordi Cortadella u. a. *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers*. Techn. Ber. Univ. Politecnica de Catalunya, Barcelona, Spain u. a., 1997.

[Edi] *The Electronic Design Interchange Format*. 2005. URL: https://web.archive.org/web/20051218205705/http://www.edif.org/guide.html.

[FD96] Stephen B. Furber und Paul Day. "Four-phase micropipeline latch control circuits". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (1996), S. 247–253.

[FEH10] Phillip David Ferguson, Tughrul Arslan Aristides Efthymiou und Danny Hume. "Optimising Self-Timed FPGA Circuits". In: *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools* (2010), S. 563–570.

[FGG98] S. B. Furber, J. D. Garside und D. A. Gilbert. "AMULET3: A High-Performance Self-Timed ARM Microprocessor". In: *Computer Design: International Conference on VLSI in Computers and Processors* (1998). Proceedings.

[Fri01] Eby G. Friedman. "Clock Distribution Networks in Synchronous Digital Integrated Circuits". In: *Proceedings of the IEEE* 89.5 (Mai 2001), S. 665–692.

[Fur+97] Stephen B. Furber u. a. “AMULET2e: an asynchronous embedded controller”. In: *Third International Symposium on Advances Research in Asynchronous Circuits and Systems* (1997), S. 290–299.

[Gag+98] Hans van Gageldonk u. a. “An Asynchronous Low-Power 80C51 Microcontroller”. In: *1998 Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 1998, S. 96–107.

[KG97] Rakefet Kol und Ran Ginosar. “A doubly-latched asynchronous pipeline”. In: *Proceedings International Conference on Computer Design- VLSI in Computers and Processors*. 1997, S. 706–711.

[Kom+88] Shinji Komori u. a. “An Elastic Pipeline Mechanism by Self-Timed Circuits”. In: *IEEE Journal of Solid-State Circuits* 23.1 (1988).

[Las08] Jon Neerup Lassen. “FPGA Prototyping of Asynchronous Networks-on-Chip”. Masterthesis. Danmarks Tekniske Universitet, 2008.

[Liu97] Jianwei Liu. “Arithmetic and Control Components for an Asynchronous System”. Diss. University of Manchester, 1997.

[Mah95] K. Maheswaran. “Implementing Self-timed Circuits in Field Programmable Gate Arrays”. Masterthesis. University of California, Davis, 1995.

[Mar90] Alain J. Martin. “The Limitations to Delay-Insensitivity in Asynchronous Circuits”. In: *Proceedings of the 6th MIT Conference on Advances Research in VLSI*. MIT Press, 1990.

[MB59] D.E. Muller und W.S. Bartky. “A theory of asynchronous circuits”. In: (1959).

[MC80] Carver Mead und Lynn Conway. *Introduction to VLSI Systems*. Addison-Wesley Publishing Co, 1980.

[Moo65] Gordon E. Moore. “Cramming more components onto integrated circuits”. In: *Electronics* 38.8 (Mai 1965), S. 114.

[Moo75] Gordon E. Moore. “Progress In Digital Integrated Electronics”. In: *IEEE International Electron Devices Meeting* (1975), S. 11–13.

[Mul63] D.E. Muller. “Asynchronous Logics and Application to 20. A. Sheibanyrad, A. Greiner, and I. Miro Panades, Multi- Information Processing”. In: (1963).

[Pas04] Enric Pastor. *ASTG - Asynchronous Signal Transition Graphs*. Universitat Politecnica de Catalunya Barcelona, Spain. 2004.

[Pav94] Nigel Charles Paver. “The Design and Implementation of an Asynchronous Microprocessor”. Diss. University of Manchester, 1994.

[Pet62] Carl Adam Petri. “Kommunikation mit Automaten”. Diss. Technische Hochschule Darmstadt, 1962.

[PH05] David A. Patterson und John L. Hennessy. *Computer Organization and Design*. 3. Aufl. Morgan Kaufmann, 2005.

[PQDD10] Cuong Pham-Quoc und Anh-Vu Dinh-Duc. “Hazard-free Muller Gate for Implementing Asynchronous Circuits on Xilinx FPGA”. In: *2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications*. 2010, S. 289–292.

[Rei10] Prof Dr. Wolfgang Reisig. *Petrinetze*. Hrsg. von Ulrich Sandten und Kerstin Hoffmann. Vieweg + Teubner Verlag, 2010.

[Ren+98] M. Renaudin u. a. “ASPRO-216: a standard-cell Q.D.I. 16-bit RISC asynchronous microprocessor”. In: *1998 Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems*. 1998.

[Rud87] Richard Rudell. *File Format for GENLIB Library File*. Techn. Ber. 1987. URL: http://www.ece.cmu.edu/~ee760/760docs/genlib.pdf.

[SF01] Jens Sparsø und Steve Furber. *Principles of Asynchronous Circuit Design*. European Low-Power Initiative for Electric System Design. Kluwer Academic Publishers, 2001.

[Sut89] Ian Sutherland. “Micropipelines”. In: *Communications of the ACM, Volume 32, No. 6* (1989), S. 720–738.

[US13] Vaishali Udar und Sanjeev Sharma. “Analysis of Place and Route Algorithm for Field Programmable Gate Array (FPGA)”. In: *IEEE Conference on Information & Communication Technologies (ICT)* (2013), S. 116–119.

[UW04] Klaus Urbanski und Roland Woitowitz. *Digitaltechnik: Ein Lehr- und Übungsbuch*. 4. Aufl. Springer, 2004.

[Wak05] John F. Wakerly. *Digital Design: Principles and Practices Package*. 4. Aufl. Prentice Hall, 2005.

[Wal00] Anders Wallander. *A VHDL Implementation of a MIPS*. Techn. Ber. Luleå Tekniska Universitet, 2000.

[Xil] *ISE Source File Types*. 2009. URL: http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/ise_r_source_types.htm.

[Xil08] Xilinx. *Constraints Guide.* 10.1. Xilinx. 2008.

[Xil13] Xilinx. *Spartan-3E FPGA Family Data Sheet.* v4.1. Xilinx. Juli 2013.

B. Abkürzungsverzeichnis

EMV elektromagnetische Verträglichkeit . . . 1

EDIF Electronic Design Interchange Format . . . 5

ACK Acknowledge . . . 8

REQ Request . . . 8

IC Integrated Circuit . . . 28

STG Signal Transition Graph . . . 17

KV-Diagramm Karnaugh-Veitch-Diagramm . . . 21

FPGA Field Programmable Gate Array . . . 5

CLB Configurable Logic Block . . . 27

LUT Look-Up Table . . . 27

UCF User Constraints File . . . 27

HDL Hardware Description Language . . . 5

VHDL Very High Speed Integrated Circuit Hardware Description Language

DLAP Doubly-Latched Asynchronous Pipeline . . . vi

ASTG Asynchronous Signal Transition Graph . . . 22

C. Glossar

Block RAM Block RAM ist SRAM der auf Xilinx FPGAs mit auf den Chip Platziert ist. Bei der Verwendung von Block RAM werden keine Ressourcen der Slices benötigt.. 52, 54

CMOS „Halbleiterschaltungen, die auf einem Subtrat sowohl N-Kanal- als auch P-Kanal-Transistoren enthalten werden als (...) Complementary MOS = CMOS bezeichnet. Wegen des komplementären Aufbaus benötigen CMOS-Schaltungen einen extrem niedrigen Ruhestrom.(...) Die in der CMOS-Schaltung auftretende Verlustleistung ist der Frequenz direkt proportional und wird daher auch in der Einheit W/Hz angegeben“[UW04]. 4, 14

Forwarding „Forwarding bezeichnet eine Methode zum Lösen eines Datenkonflikts, bei der das fehlende Datenelement aus internen Pufferspeichern abgerufen und nicht darauf gewartet wird, bis dieses aus den für den Programmierer sichtbaren Register oder dem Speicher kommt“[PH05]. 53

Hazard Die Gatterlaufzeiten in den kombinatorischen Pfaden können temporäre Zustandsänderungen hervorrufen. Eine Schaltung, die solch ein Verhalten aufweist, hat einen Hazard.. 28, 29, 32, 34

Pipeline Pipelines sind ein effektives Mittel um Schaltkreise performanter zu machen. Die Abarbeitung wird dabei in Teilaufgaben zerlegt und kann dadurch parallel durchgeführt werden. Weil die Teilaufgaben einfacher und dadurch die Länge des kombinatorischen Pfades kleiner ist, kann das gesamte System schneller getaktet werden.. 10

Speed Independent Geht man davon aus, dass die Laufzeiten der Logikgatter unbegrenzt und die Laufzeiten der Leitungen vernachlässigbar sind, spricht man von geschwindigkeitsunabhängigen (speed-independent) Schaltungen.. 19

Verilog Verilog ist neben VHDL die meistgenutzte Hardware Description Language. Sie ist stark verbreitet in Nordamerika und Japan, weniger in Europa. 24

D. Glossar verwendeter Programme

HEX2VHDL HEX2VHDL ist ein kleines Java Programm für die Umwandlung einer HEX Datei in ein VHDL Array.. 52

Mars Mars (MIPS Assembler and Runtime Simulator) ist eine interaktive Entwicklungsumgebung für die MIPS Assemblerprogrammierung. Das Programm wurde an der Missouri State University Entwickelt und kann unter folgenden Link heruntergeladen werden: `http://courses.missouristate.edu/KenVollmar/MARS/`. 52

Petrify Petrify ist ein Werkzeug, um Petrinetze und STGs zu manipulieren und zu synthetisieren. Es ist ein typisches UNIX Programm mit vielen Optionen und Schaltern. Die Syntax für die Input-Datei und die möglichen Optionen sind in der Man Page [Pas04] beschrieben. Benutzte Version ist: 4.2. Das Programm ist zu bekommen unter folgenden Link:
`http://www.lsi.upc.edu/~jordicf/petrify/`
Seiten: . v, 22–25, 40

VSTGL Visual STG Lab wurde an der DTU (Danmarks Tekniske Universitet) entwickelt um STGs zu zeichnen und zu simulieren. Das Programm wurde mit der C++ Klassenbibliothek Qt 1.4x entwickelt und sieht daher, für heutige Verhältnisse, etwas antiquiert aus. Ein Nachteil ist, dass aktuelle Linux Distributionen nicht mehr die alte Version von Qt unterstützen. Das erneute kompilieren des Programms mit einer neueren Qt Version setzt voraus, das der Sourcecode massiv verändert werden müsste, weil keine Absatzkompatibilität besteht. Eine Lösung: Eine ältere Linux Distribution in einer Virtuellen Umgebung wie z.B. VMware installieren, um in dieser dann VSTGL zu starten.
Download Link:
`http://sourceforge.net/projects/vstgl/files/`
Seite:. 24

Xilinx ISE Design Software Die wichtigsten Werkzeuge von ISE sind:

- **Project Navigator** ist die zentrale Benutzerschnittstelle. Die meisten anderen Werkzeuge können hieraus gestartet werden.
- **XST** ist der Logiksynthetisierer von Xilinx. Er wandelt HDL (VHDL/Verilog) in Xilinxspezifische Netzlisten um.
- **MAP** macht die Zuordnung der Netzliste zu FPGA Primitiven.
- **PAR** platziert und verbindet die FPGA Primitiven.
- **Floorplanner** kann benutzt werden vor dem MAP und nach dem PAR. Vor dem MAP kann der Floorplanner benutzt werden um Einschränkungen am Design festzulegen. nach dem PAR können manuell änderungen am Floorplan vorgenommen werden.

http://www.xilinx.com/products/design-tools/ise-design-suite/ise-webpack.htm
Seite. 28

E Abbildungsverzeichnis

1.1. Synchroner Datentransfer . . . 1

1.2. Asynchroner Datentransfer . . . 2

2.1. Struktur eines Datenpfades mit dem Dual-Rail Protokoll . . . 7

2.2. Bundled Data nach Seitz . . . 8

2.3. Two-Phase Bundled Data Convention . . . 9

2.4. Four-Phase Bundled Data Convention . . . 9

2.5. Muller C-Element . . . 10

2.6. Muller Pipeline[SF01] . . . 11

2.7. 4-Phasen Bundled-Data Pipeline . . . 12

2.8. Micropipeline mit "Capture-PassLatches, siehe Text . . . 13

2.9. Realisierung eines Capture-Pass Latch . . . 13

2.10. Szenarien einer gekoppelten Pipelinestufe . . . 16

2.11. 4 Phasen Bundled Data Fork . . . 17

2.12. 4 Phasen Bundled Data Join . . . 17

2.13. Schaltsymbol Fork . . . 17

2.14. Schaltsymbol Join . . . 17

2.15. Einfaches Petrinetz . . . 19

2.16. Petrinetz nach dem Schalten der Transition t0 . . . 19

2.17. Formale Spezifikation mit einem Petrinetz und einem STG des Muller C-Elements aus einem Signalzeitplan [Chu87; SF01] . . . 20

2.18. Zustandsgraph des Muller C-Elements . . . 21

2.19. KV-Diagramm für y . . . 21

2.20. Einfacher Latch Controller STG . . . 22

2.21. Schaltbild eines einfachen Latch-Controllers . . . 24

2.22. Workflow für Petrify . . . 25

3.1. Gruppierung der Slices in einem CLB [Xil13] 28
3.2. Nummerierung der Slices im FPGA [Xil13] 29
3.3. Schaltbild des Muller C-Elements 31
3.4. Muller C-Element mit 3 Eingängen 32
3.5. Schaltbild des Muller C-Elements mit 3 Eingängen 32
3.6. Schaltbild des Muller C-Elements mit 4 Eingängen 33
3.7. Asymmetrisches Delayelement mit LUTs 36
3.8. Delayelement aus einem Schieberegister 38
3.9. Timing Simulation des Delay Shift Elements 38
3.10. Diagramm Schiebeoperationen Zeit 39
3.11. Einfache Pipeline 40
3.12. Simulation der einfachen Pipeline 41
3.13. STG eines Entkoppelten Controllers [FD96] 42
3.15. Simulation der Pipeline aus Abbildung 3.14 42
3.14. Pipeline mit Registercontrollern die entkoppelten sind 43
3.16. DLAP Pipelinestufe [KG97] 44
3.17. STG eines Master-Slave Registercontrollers [KG97] 44
3.18. Schaltbild des Master-Slave Registercontrollers 45
3.19. Simulation der DLAP Implementierung des Beispiels aus Abbildung 3.14 . . . 45
3.20. Synchronisation einer DLAP nach [FEH10] 46
3.21. Implementierung eines DLAP Registercontrollers als Fork 46
3.22. Implementierung eines DLAP Registercontrollers als Join 47

4.1. Abhängigkeitsgraph mit einem Fork und Join 50
4.2. Übersicht des MIPS mit den 5 Pipelinestufen [Wal00] 53
4.3. Abhängigkeiten der Pipelinestufen des MIPS Prozessors 54
4.4. Kontrollpfad für den MIPS Prozessor 54

F.1. Nicht entkoppelter Registercontroller 85

F. Quellcodecode

Listings

2.1. Einfacher Latch-Controller 23
2.2. Ausgabe von Petrify 23

3.1. VHDL Implementierung des Muller C-Elements 29
3.2. UCF des Muller C-Elements mit 3 Eingängen 32

4.1. VHDL Implementierung eines Registers 51
4.2. DLAP Registers 51

F.1. VHDL Implementierung eines Muller C-Elements mit Reset 75
F.2. Verilog Implementierung eines Muller C-Elements mit 3 Eingängen und Reset [PQDD10] 78
F.3. VHDL Implementierung eines Muller C-Elements mit 4 Eingängen und Reset . 79
F.4. VHDL Implementierung einer asymmetrischen Delay Chain [Las08] 81
F.5. VHDL Implementierung des Shift Delayelements 83
F.6. Technologiebibliothek für Petrify im GENLIB Format [Col] 84
F.7. VHDL Implementierung eines Registercontrollers, der nicht entkoppelt ist . . 86
F.8. ASTG Datei für einen Entkoppelten Registercontrollers 87
F.9. Verilog Implementierung des Entkoppelten Registercontrollers 88
F.10. ASTG Datei für einen DLAP Registercontrollers 89
F.11. Verilog Implementierung eines DLAP Registercontrollers 90
F.12. Verilog Implementierung eines Join für DLAPs 91

F.1. Grundlegende Elemente asynchroner Systeme

F.1.1. Muller C-Element

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
```

```
4  library unisim;
5  use unisim.vcomponents.ALL;
6
7  entity muller_c is
8          generic(
9                  reset_value : bit := '0'
10         );
11     Port ( reset : in  STD_LOGIC;
12            a : in  STD_LOGIC;
13            b : in  STD_LOGIC;
14            z : out  STD_LOGIC);
15 end muller_c;
16
17 architecture Behavioral of muller_c is
18         constant rv : bit := reset_value;
19         constant reset_vector : bit_vector(7 downto 0)
20                 := rv&rv&rv&rv&rv&rv&rv&rv;
21         -- Internal signals
22         signal s_out : std_logic;
23         attribute keep : string;
24         attribute keep of s_out : signal is "true";
25 begin
26
27         c_element: lut4_l
28         generic map (
29                 init => "11101000" & reset_vector
30         )
31         port map (
32                 i0 => a,
33                 i1 => b,
34                 i2 => s_out,
35                 i3 => reset,
36                 lo => s_out
37         );
38         z <= s_out after 1 ns;
39
40 end Behavioral;
41
42 library IEEE;
43 use IEEE.STD_LOGIC_1164.ALL;
```

```
44
45 library unisim;
46 use unisim.vcomponents.ALL;
47
48
49 entity muller_c_1 is
50         generic(
51                 reset_value : bit := '1'
52         );
53    Port ( reset : in  STD_LOGIC;
54           a : in  STD_LOGIC;
55           b : in  STD_LOGIC;
56           z : out  STD_LOGIC);
57 end muller_c_1;
58
59 architecture Behavioral of muller_c_1 is
60         constant rv : bit := reset_value;
61         constant reset_vector : bit_vector(7 downto 0)
62                                 := rv&rv&rv&rv&rv&rv&rv&rv;
63         -- Internal signals
64         signal s_out : std_logic;
65         attribute keep : string;
66         attribute keep of s_out : signal is "true";
67 begin
68
69         c_element: lut4_l
70         generic map (
71                 init => "11101000" & reset_vector
72         )
73         port map (
74                 i0 => a,
75                 i1 => b,
76                 i2 => s_out,
77                 i3 => reset,
78                 lo => s_out
79         );
80         z <= s_out after 1 ns;
81
```

```
82 end Behavioral;
```

Listing F.1: VHDL Implementierung eines Muller C-Elements mit Reset

F.1.2. Muller C-Element mit 3 Eingängen

```
module C3_element(
    input  A,
    input  B,
    input  C,
    input  R,
    output S
    );
    LUT4 #(.INIT(16'hFE80))
          LUT4_inst(.I0(A),
                    .I1(B),
                    .I2(C),
                    .I3(S),
                    .O(w1));
    LUT2 #(.INIT(4'h8)) //nach Reset Ausgang S = '1': .INIT(4'hD)
          LUT2_Inst(.I0(R),
                    .I1(w1),
                    .O(S));
endmodule
```

Listing F.2: Verilog Implementierung eines Muller C-Elements mit 3 Eingängen und Reset [PQDD10]

i3 = S	i2 = C	i1 = B	i0 = A	init Vektor = S'
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

Tabelle F.1.: Tabelle für den Initialisierungsvektor der LUT4_inst

F.1.3. Muller C-Element mit 4 Eingängen

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

library unisim;
use unisim.vcomponents.ALL;

entity c4_element is
   Port ( reset : in  STD_LOGIC;
          A : in  STD_LOGIC;
          B : in  STD_LOGIC;
          C : in  STD_LOGIC;
          D : in  STD_LOGIC;
          Y : out  STD_LOGIC
   );
end c4_element;

```

```
17 architecture Behavioral of c4_element is
18     signal LUT_0_s, LUT_1_s, Y_s: STD_LOGIC;
19 begin
20    LUT_all_0: lut4_l
21    generic map(
22       init => "1111111111111110"
23    )
24    port map (
25       i0 => D,
26       i1 => C,
27       i2 => B,
28       i3 => A,
29       lo => LUT_0_s
30    );
31
32    LUT_all_1: lut4_l
33    generic map(
34       init => "1000000000000000"
35    )
36    port map (
37       i0 => D,
38       i1 => C,
39       i2 => B,
40       i3 => A,
41       lo => LUT_1_s
42    );
43
44    LUT_Y: lut4_l
45    generic map(
46       init => "1100100010000000"
47    )
48    port map (
49       i0 => Y_s,
50       i1 => reset,
51       i2 => LUT_1_s,
52       i3 => LUT_0_s,
53       lo => Y_s
54    );
55    Y <= Y_s;
```

```
56 end Behavioral;
```

Listing F.3: VHDL Implementierung eines Muller C-Elements mit 4 Eingängen und Reset

F.1.4. Delayelemente

Delay Chain

```
1
2  library ieee ;
3  use ieee.std_logic_1164.all ;
4
5  library unisim ;
6  use unisim.vcomponents.lut2 ;
7  use unisim.vcomponents.lut1 ;
8
9  entity as_bd_4p_delay is
10   generic (
11     size : natural range 1 to 30 := 6 -- Delay size
12   );
13   port (
14     d : in std_logic ; -- Data in
15     z : out std_logic -- Data out
16   );
17 end as_bd_4p_delay ;
18
19 architecture lut of as_bd_4p_delay is
20
21   component lut2
22   generic (
23     init : bit_vector := X"4"
24   );
25   port (
26     o : out std_ulogic ;
27     i0 : in std_ulogic ;
28     i1 : in std_ulogic
29   );
30   end component ;
31
32  -- ---------------------------------------------------
33  -- Internal signals .
```

```
34 -- ---------------------------------------------------
35   signal s_connect : std_logic_vector ( size downto 0);
36 -- signal d_inv , o_first : std_logic ;
37 
38 -- ---------------------------------------------------
39 -- Synthesis attributes - we don't want the
40 -- synthesizer to optimize the delay - chain .
41 -- ---------------------------------------------------
42   attribute keep : string ;
43   attribute keep of s_connect : signal is "true"; -- d_inv
44 
45   attribute rloc : string ;
46 
47   begin
48     s_connect (0) <= d;
49 
50 -- ---------------------------------------------------
51 -- Create a riple - chain of luts ( and gates ).
52 -- ---------------------------------------------------
53    lut_chain : for index in 0 to (size -1) generate
54 
55    signal o : std_logic ;
56   type y_placement is array ( integer range 0 to 29) of integer;
57 --constant y_val : y_placement := (0,1,0,1,0,1,0,1,2,3,2,3,2,3,2,
58 --                                 3,4,5,4,5,4,5,4,5,6,7,6,7,6,7);
59 --                                   location Slices Vertex5
60   constant y_val : y_placement := (0,1,2,3,4,5,6,7,8,9,10,11,12,
61                                    13,14,15,16,17,18,19,20,21,22,
62                                    23,24,25,26,27,28,29);
63 --                              location slices Spartan3!
64   begin
65     delay_lut : lut2
66     generic map (
67        init => X"8" -- And truth - table .
68     )
69     port map (
70        I1 => d,
71        I0 => s_connect ( index ),
72        O => o
73     );
```

```
74       s_connect ( index +1) <= o after 1 ns;
75                        --Simulate delay of 1 ns.
76    end generate lut_chain ;
77
78  -- ------------------------------------------------
79  -- Connect the output of delay element
80  -- ------------------------------------------------
81    z <= s_connect (size -1);
82 end lut ;
```

Listing F.4: VHDL Implementierung einer asymmetrischen Delay Chain [Las08]

Shift Delay

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  Library UNISIM;
4  use UNISIM.vcomponents.all;
5  entity DelaySLR is
6  generic (
7  WIDTH : natural := 5
8  );
9  Port ( a : in  STD_LOGIC;
10        z : out  STD_LOGIC);
11 end DelaySLR;
12
13 architecture Behavioral of DelaySLR is
14    signal i0_s : std_logic;
15    signal lc_s : std_logic;
16    signal vektor : std_logic_vector(WIDTH downto 0) :=
17                                                (others => '0');
18 begin
19    oszillator: lut2_l
20    generic map (
21       init => "0100"
22    )
23    port map (
24       i0 => i0_s,
25       i1 => a,
26       lo => i0_s
27    );
```

```
28     lc_s <= i0_s;
29 
30     process(a, lc_s)
31     begin
32         if a = '0' then
33             vektor <= (others => '0');
34         elsif lc_s'event and lc_s = '1' then
35             vektor(0) <= '1';
36             for i in 1 to WIDTH loop
37                 vektor(i) <= vektor(i-1);
38             end loop;
39         end if;
40     end process;
41     z <= vektor(WIDTH);
42 end Behavioral;
```

Listing F.5: VHDL Implementierung des Shift Delayelements

F.2. Technologiebibliothek

```
1  # --- Kombinatorische Schaltungen
2 
3  GATE    "inv"          16      O=!A;
4  PIN     * INV 1 999 1 .2 1 .2
5 
6  GATE    "and2"         32      O=A*B;
7  PIN     * NONINV 1 999 1 .2 1 .2
8 
9  GATE    "nand2"         24      O=!(A*B);
10 PIN     * INV 1 999 1 .2 1 .2
11 
12 GATE    "and3"         40      O=A*B*C;
13 PIN     * NONINV 1 999 1 .2 1 .2
14 
15 GATE    "nand3"         32      O=!(A*B*C);
16 PIN     * INV 1 999 1 .2 1 .2
17 
18 GATE    "or2"          32      O=A+B;
19 PIN     * NONINV 1 999 1 .2 1 .2
20 
21 GATE    "nor2"          24      O=!(A+B);
```

```
22 PIN      * INV 1 999 1 .2 1 .2
23
24 GATE     "or3"           40      O=A+B+C;
25 PIN      * NONINV 1 999 1 .2 1 .2
26
27 GATE     "nor3"            32      O=!(A+B+C);
28 PIN      * INV 1 999 1 .2 1 .2
29
30 # Muller C-Element
31 LATCH    "muller_c"      40      C = A*B+(A+B)*C_NEXT;
32 PIN      A       NONINV 1 999 1 .2 1 .2
33 PIN      B       NONINV 1 999 1 .2 1 .2
34 SEQ      C C_NEXT ASYNCH
35
36 # Muller C3-Element
37 LATCH  "muller_c3"       40       D = (A*B*C)+((A+B+C)*D_NEXT);
38 PIN      A       NONINV 1 999 1 .2 1 .2
39 PIN      B       NONINV 1 999 1 .2 1 .2
40 PIN      C       NONINV 1 999 1 .2 1 .2
41 SEQ      D D_NEXT ASYNCH
```

Listing F.6: Technologiebibliothek für Petrify im GENLIB Format [Col]

F.3. Registercontroller

F.3.1. Nicht entkoppelter Registercontroller

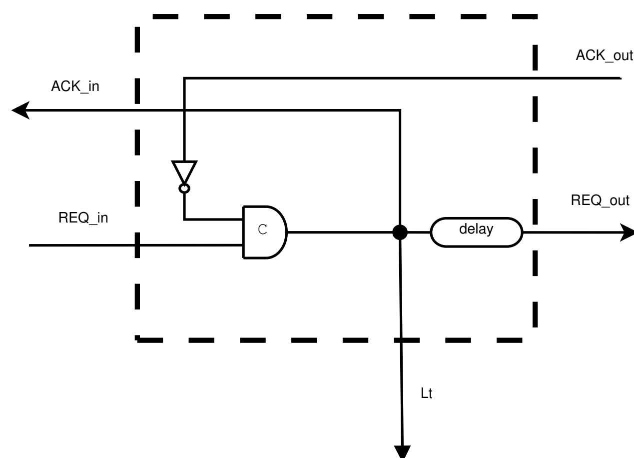


Abbildung F.1.: Nicht entkoppelter Registercontroller

i3 = Reset	i2 = s_out	i1 = Aout	i0 = Rin	init Vektor = s_out
0	0	0	0	RV(7)
0	0	0	1	RV(6)
0	0	1	0	RV(5)
0	0	1	1	RV(4)
0	1	0	0	RV(3)
0	1	0	1	RV(2)
0	1	1	0	RV(1)
0	1	1	1	RV(0)
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

Tabelle F.2.: Tabelle für den Initialisierungsvektor eines Registercontrollers, der nicht entkoppelt ist

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  library unisim;
5  use unisim.vcomponents.ALL;
6
7  entity Register_CTR_undecoupled is
8          generic(
9                  reset_value : bit
10         );
11     Port ( Ain : out  STD_LOGIC;
12            Rin : in  STD_LOGIC;
13            Aout : in  STD_LOGIC;
14            Rout : out  STD_LOGIC;
15            R_ctr : out  STD_LOGIC;
16                        reset : in STD_LOGIC
```

```
17                                  );
18 end Register_CTR_undecoupled;
19
20 architecture Behavioral of Register_CTR_undecoupled is
21         constant rv : bit := reset_value;
22         constant reset_vector : bit_vector(7 downto 0)
23                               := rv&rv&rv&rv&rv&rv&rv&rv;
24         signal s_out : std_logic;
25         attribute keep : string;
26         attribute keep of s_out : signal is "true";
27 begin
28    c_element: lut4_l
29         generic map (
30                 init => "10110010" & reset_vector
31         )
32         port map (
33                 i0 => Rin,
34                 i1 => Aout,
35                 i2 => s_out,
36                 i3 => reset,
37                 lo => s_out
38         );
39         Rout <= s_out after 1 ns;
40         Ain <= s_out after 1 ns;
41         R_ctr <= s_out after 1 ns;
42 end Behavioral;
```

Listing F.7: VHDL Implementierung eines Registercontrollers, der nicht entkoppelt ist

F.3.2. Entkoppelter Registercontroller

```
1 .model FullyDecoupl
2 .inputs Aout Rin
3 .outputs Lt Rout Ain
4 .internal B A
5 .graph
6 Rin+ Ain+
7 A+ Lt+ Rout+
8 Lt+ Ain+
9 Ain+ B+ Rin-
10 B+ A- Ain-
```

```
A- Lt- Rout-
Lt- B-
B- A+
Rin- Ain-
Ain- B- Rin+
Rout+ Aout+
Aout+ A-
Rout- Aout-
Aout- Rout+
.marking { <Lt+ ,Ain+ >  <Ain- ,Rin+ >  <Rout+ ,Aout+ > }
.end
```

Listing F.8: ASTG Datei für einen Entkoppelten Registercontrollers

```
module FullyDecoupl_net (
    reset,
    Aout,
    Rin,
    Lt,
    Rout,
    Ain
);
input reset;
input Aout;
input Rin;

output Lt;
output Rout;
output Ain;


// Functions mapped into library gates:
// -----------------------------------

buf _U0 (Lt,A);
// This inverter should have a short delay
inv _U1 (.A(csc0), .O(Rout));
// This inverter should have a short delay
inv _U2 (.A(Lt), .O(_1_));
nor2 _U3 (.A(_1_), .B(B), .O(_2_));
muller_c _U4 (.reset(reset), .A(Rin), .B(_2_), .C(Ain));
```

```
28 // This inverter should have a short delay
29 inv _U5 (.A(Lt), .O(_4_));
30 nand2 _U6 (.A(csc0), .B(_4_), .O(_5_));
31 muller_c _U7 (.reset(reset), .A(Ain), .B(_5_), .C(B));
32 // This inverter should have a short delay
33 inv _U8 (.A(csc0), .O(_7_));
34 nand2 _U9 (.A(_7_), .B(Aout), .O(_8_));
35 // This inverter should have a short delay
36 inv _U10 (.A(B), .O(_9_));
37 muller_c_1 _U11 (.reset(reset), .A(_9_), .B(_8_), .C(A));
38 // This inverter should have a short delay
39 inv _U12 (.A(A), .O(_11_));
40 muller_c _U13 (.reset(reset), .A(_11_), .B(Aout), .C(csc0));
41
42 // signal values at the initial state:
43 //  Lt !Aout !Rin Rout !_1_ _2_ !Ain !_4_ _5_
44 //  !B _7_ _8_ _9_ A !_11_ !csc0
45 endmodule
```

Listing F.9: Verilog Implementierung des Entkoppelten Registercontrollers

F.3.3. Registercontroller für eine DLAP

```
1 .model controller
2 # Declaration of signals
3 .inputs r_in a_out
4 .outputs r_out a_in rm rs
5 .internal a
6 # Petri net
7 .graph
8 a_in- r_in+
9 r_in+ rm+
10 rm+ a-
11 a- rm-
12 rm- a_in-
13 rm+ a_in+
14 a_in+ r_in-
15 r_in- rm-
16 r_out- a_out-
17 a_out- rs+
18 rs+ a+
```

```
a+ rs-
rs- r_out-
rs+ r_out+
r_out+ a_out+
a_out+ rs-
a+ rm+
a- rs+
rm- a+
rs- a-

# Initial marking
.marking { <r_out+,a_out+> <r_in+,rm+> <a+,rm+> <a+,rs-> }
.end
```

Listing F.10: ASTG Datei für einen DLAP Registercontrollers

```
module controller_net (
    r_in,
    a_out,
    r_out,
    a_in,
    rm,
    rs
);

input r_in;
input a_out;

output r_out;
output a_in;
output rm;
output rs;


// Functions mapped into library gates:
// ------------------------------------

buf _U0 (r_out,rs);
buf _U1 (a_in,rm);
cgate _U2 (.A(a), .B(r_in), .C(rm));
inv _U3 (.A(a_out), .O(_1_));
```

```
26 // This inverter should have a short delay
27 inv _U4 (.A(a), .O(_2_));
28 cgate _U5 (.A(_1_), .B(_2_), .C(rs));
29 // This inverter should have a short delay
30 inv _U6 (.A(rm), .O(_4_));
31 cgate _U7 (.A(_4_), .B(rs), .C(a));
32
33 // signal values at the initial state:
34 //      r_out !a_in r_in !a_out !rm _1_ !_2_ rs _4_ a
35 endmodule
```

Listing F.11: Verilog Implementierung eines DLAP Registercontrollers

F.3.4. Join für DLAP

```
1 module join_net(
2    reset,
3    Ri1,
4    Ai1,
5    Ri2,
6    Ai2,
7    Ro,
8    Ao,
9    Lm,
10   Ls
11 );
12  input reset;
13  input Ri1;
14  output Ai1;
15  input Ri2;
16  output Ai2;
17  output Ro;
18  input Ao;
19  output Lm;
20  output Ls;
21
22  buf _U0 (Ro,Ls);
23  buf _U1 (Ai1,Lm);
24  buf _U8 (Ai2,Lm);
25  inv _U2 (.A(Ao), .O(_0_));
26  // This inverter should have a short delay
```

```
27  inv _U3 (.A(B), .O(_1_));
28  cgate_reset1 _U4 (.reset(reset),.A(_0_), .B(_1_), .C(Ls));
29  //cgate_reset0 _U5 (.reset(reset),.A(B), .B(Ri), .C(Lm));
30  C3_element_0 _U5 (.R(reset),.A(B), .B(Ri1), .C(Ri2), .S(Lm));
31  // This inverter should have a short delay
32  inv _U6 (.A(Lm), .O(_4_));
33  cgate_reset1 _U7 (.reset(reset),.A(Ls), .B(_4_), .C(B));
34 // signal values at the initial state:
35 // Ro !Ai !Ao !Ri _0_ !_1_ Ls !Lm _4_ B
36 endmodule
```

Listing F.12: Verilog Implementierung eines Join für DLAPs

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 14. November 2014 Thorben J. Wübbenhorst