Bachelorarbeit

Konzeption und Entwicklung eines Logic Analyzers auf Hardwarebeschreibungsebene

von

Manuel Penschuck

Betreuer:
Prof. Dr. rer. nat. Uwe Brinkschulte
M.Sc. Benjamin Betting

Professur für Eingebettete Systeme
Institut für Informatik
Johann Wolfgang Goethe-Universität

4. Dezember 2011

Zusammenfassung

Logic Analyzer sind Messgeräte, die primär zur Untersuchung von komplexen digitalen Systemen eingesetzt werden und eine Vielzahl von Signalen parallel erfassen können. Ausgehend von einer Untersuchung verfügbarer Lösungen, wurde im Rahmen dieser Arbeit ein leistungsfähiger und flexibler integrierter Logic Analyzer mit geringem Logikbedarf konzipiert und mittels der Hardwarebeschreibungssprache *VHDL* implementiert.

Das Projekt ist dabei als portables Framework ausgelegt, das die Kernfunktionalität des Logic Analyzers und das Triggersystem enthält. Technologiespezifische Treiber werden außerhalb der Struktur zur Verfügung gestellt. Die Anbindung erfolgt mittels freiem *WISHBONE Bus*, über den das System konfiguriert werden kann. Umsetzer für *RS232* und *JTag* wurden implementiert. Neben einer verlustfreien Datenkompression ist die Reduzierung des Speicherbedarfs zur Laufzeit durch das Deaktivieren einzelner Eingänge möglich. Durch eine zentrale Konfiguration kann der Entwurf anhand diverser Optionen vor der Synthese auf den Anwendungsfall angepasst werden, insbesondere lässt sich die Anzahl der Eingänge bestimmen.

Als Referenzsystem kommt ein *low-cost* FPGA vom Typ *Xilinx Spartan-3* zum Einsatz, mit dem bei vollem Funktionsumfang eine Abtastrate von 99 MHz erreicht werden konnte. Hierbei benötigt ein Logic Analyzer mit 32 Eingängen, Hardwarekompression und *RS232*-Schnittstelle unter Nutzung aller FPGA-internen RAM-Blöcken nur rund 360 Makrozellen.

Selbständigkeitserklärung

Hiermit versichere ich die vorliegende Arbeit selbständig und nur unter Verwendung angegebener Hilfsmittel und Quellen angefertigt zu haben.

Manuel Penschuck

Hattersheim, den 4. Dezember 2011

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Motivation

1.1 Motivation

Die Entwicklung eines komplexen elektronischen Systems erfordert neben einer systematischen Planung auch diverse Verifikationsstufen. Bei einer nicht-linearen Projektplanung – beispielsweise mittels Rapid-Prototyping – wirken sich diese Zwischenschritte sogar wieder direkt auf Details des Designs aus wodurch das Testen nicht mehr ausschließlich der Fehlerbeseitigung bzw. Qualitätssicherung dient. Werkzeuge, die quantitative Untersuchungen der Systeme ermöglichen, sind daher unverzichtbar. Dies gilt für Hardware- wie Software-Engineering gleichermaßen [LL07].

In der technischen Informatik stehen zum Korrektheitsnachweis eine Vielzahl an Methoden zur Verfügung. So ist ein formaler Korrektheitsbeweis zwar im Allgemeinen wünschenswert, jedoch meist nur für kleinste Teilprobleme möglich. Daher gewinnt mit steigender Komplexität die Simulation anhand von Verhaltensmodellen an Bedeutung.

Ein solches Modell besteht dabei aus einer Verhaltensbeschreibung des Entwurfs (z.B. in *VHDL*, *Verilog* oder *SystemC*) und Stimuli, deren Effekte bewertet werden. Dieser Ansatz erlaubt das genaue Inspizieren von Signalen sowie deren Wirkung auf das Design. Da es sich um ein isoliertes System handelt, eignet sich die Simulation besonders für *Modultests (Unit Tests)*.

Die für einen *Integrationstest* nötige Interaktion mit der Umgebung kann jedoch meist nur durch fest programmierte Stimuli approximiert werden. Es ist im Allgemeinen nicht möglich, korrekte und vollständige Verhaltensmodelle der Kommunikationspartner zu erstellen. Daher kommt bereits in frühen Entwicklungsstadien programmierbare Logik (z.B. in FPGAs) zum Einsatz, um das Zusammenspiel eines Entwurfs mit seiner Zielumgebung zu untersuchen. Verglichen mit einer Simulation können Testfälle dabei meist flexibler und praxisbezogener gestaltet werden. Jedoch ist es so nur noch indirekt möglich, die Interna eines Entwurfs zu überwachen.

Ein Interface zum Auslesen dieser verborgenen Signale stellt beispielsweise ein integrierter Logic Analyzer zur Verfügung. Ein solches Messgerät wird – nach einer ausführlichen Analyse vorhandener Lösungen – im Rahmen dieser Arbeit konzipiert und implementiert. Es soll hiermit ein freies und offenes Werkzeug zum einfacheren Debuggen von Schaltungen in *FPGA*s geschaffen werden.

1.2 Struktur dieser Arbeit

Im folgenden Kapitel sollen Grundlagen eingeführt werden, die für das Verständnis dieser Ausarbeitung hilfreich sind. Hierzu gehören eine Beschreibung und Abgrenzung von Logic Analyzern, sowie eine kurze Erläuterung der Herausforderungen beim Sampling (d.h. Aufzeichnen von Signalen). Weiterhin werden verwendete Techniken, wie *VHDL*, und eine Auswahl der später implementierten Kommunikations-Schnittstellen, erläutert.

Ziel des dritten Kapitels, *State of the Art*, ist es, vorhandene Ansätze vorzustellen und deren Eigenschaften zu beleuchten. Hierbei wird besonders auf das von *Xilinx* vertriebenen

ChipScope™ Pro und den freien *Sump* [Pop07] Logic Analyzer eingegangen. Aus dieser Analyse heraus wird im vierten Kapitel ein Konzept entwickelt, das in einem Pflichtenheft des zu entwickelnden Analyzers mündet.

Das fünfte Kapitel beschreibt die Implementierung. Dafür wird – der Entwicklung folgend – in einem Top-Down-Ansatz zuerst das Gesamtsystem präsentiert und dann die Funktion und Struktur der Submodule beschrieben.

Abschließend werden im letzten Kapitel Stärken und Schwächen der Konzeption sowie der Implementation diskutiert, und Ansatzpunkte für mögliche Verbesserungen erarbeitet. Die ausführliche Dokumentation des Produktes ist im Appendix beigefügt.

2 Grundlagen

Dieses Kapitel erläutert einige Grundlagen, die in der weiteren Ausarbeitung benötigt werden. So wird etwa die Struktur der verwendeten Hardwarebeschreibungssprache *VHDL* vorgestellt, da diese die Implementierung stark beeinflusst. Weiterhin werden die systematischen Schwierigkeiten beim Aufzeichnen von Signalen, dem sog. Abtasten, aufgezeigt. Im Anschluss folgt ein kurzer Überblick über den *WISHBONE Bus*, der zur internen Kommunikation im Logic Analyzer genutzt wird, sowie über die *JTag*-Schnittstelle, die die externe Anbindung ermöglicht.

2.1 Logic Analyzer

Logic Analyzer sind Instrumente der Digitaltechnik. Sie zeichnen den zeitlichen[1] Verlauf von Signalen auf, und stellen diesen, meist graphisch aufbereitet, dar.

Ihre Funktion ist daher sehr stark mit der von Oszilloskopen verwandt. Während diese jedoch für Messungen analoger Signale auf typischerweise einem bis vier Kanälen ausgelegt sind, verfügen Logic Analyzer über eine höhere Anzahl an Eingängen, die aber in der Regel nur logische Zustände wie `low`, `high` und `undefined` erfassen. Aufgrund ihrer engen funktionalen Verwandtschaft und den zum Teil sehr ähnlichen Anforderungen an die Bedienung existieren sog. *Mixed-Signal-Oszilloskope*. Diese sind sowohl mit analogen als auch digitalen Eingängen ausgestattet.

Historisch gesehen handelt es sich bei Logic Analyzern um dedizierte Geräte, die über vielpolige Kontakte mit der zu untersuchenden Hardware verbunden werden. Dabei werden z.B. Busse, die verschiedene Bausteine auf einer Leiterplatte verbinden, abgetastet, um den Nachrichtenfluss zu protokollieren. So können etwa durch die Überwachung des Datenbusses eines Prozessors in vielen Fällen Rückschlüsse auf den Programmverlauf gezogen werden. Logic Analyzer agieren dabei rein passiv, benötigen also meist – bis auf die Kontaktmöglichkeit – keine Vorkehrungen auf dem zu untersuchenden Objekt. Anders als bei Emulatoren wird das überwachte System nicht direkt beeinflusst. Assistiert das Gerät beim Dekodieren von Busprotokollen spricht man häufig von *Protocol Analyzern*.

[1] Wie in Kapitel 2.2 beschrieben, wird synchron zu einem Takt aufgezeichnet. Wenn nicht explizit anders ausgewiesen, wird in dieser Arbeit angenommen, dass dieser Takt äquidistant ist und somit einer Aufzeichnung eine konstante Zeitbasis zugeordnet werden kann.

Das Leistungsspektrum verschiedener Logic Analyzer ist immens. Am unteren Ende sind einfache Lösungen zu nennen, die nur in Verbindung mit einem PC betrieben werden können, und über wenige Eingänge sowie geringe Speichertiefe verfügen[2]. Für komplexere Messungen existieren modulare Systeme, die tausende Eingänge mit mehr als 10 GSamples/s auflösen können[3].

2.1.1 Trigger

Da bei typischen Messungen nur relativ kurze Ereignisse erfasst werden müssen, ist die Speichergröße von Logic Analyzern stark begrenzt. Relevante Daten müssen folglich erkannt werden, um diese dann gezielt aufzeichnen zu können. Hierzu werden sog. *Trigger*[4] verwendet.

Die durchzuführende Messung entscheidet maßgeblich über die Art des einzusetzenden Triggers. Einfache Modelle lösen z.B. bei Flankenübergängen aus, oder erkennen, wenn Busse gewünschte Werte annehmen. Komplexere Ausführungen können zusätzlich noch den zeitlichen Verlauf der Signale berücksichtigen oder verschiedene Events durch Bedingungen verknüpfen. In diesem Zusammenhang spricht man von einer *Trigger Sequence*, wobei einzelne Schritte der Abfolge als *Trigger Level* bezeichnet werden.

Auch ermöglichen einige Geräte das Interpretieren der Messungen, sodass ein Trigger Event auf einer höheren Abstraktionsebene bestimmt werden kann. Wird beispielsweise das Protokoll des aufgezeichneten Busses vom Analyzer unterstützt, müssen Ereignisse nicht mehr durch Signalformen definiert werden, sondern können allgemein gültig anhand der übermittelten Daten beschrieben werden[5]. Dies ist besonders bei *Protocol Analyern* der Fall.

Die meisten Geräte unterstützen es, die Aufzeichnung zeitlich zu dem Trigger Event zu versetzen. Wird mit dem Ereignis gestoppt, enthält der Speicher den Verlauf, der zu einem Auslösen führte. Man spricht in diesem Fall von *Pretriggering*. Mit *Posttriggering* bezeichnet man analog eine Aufzeichnung, die mit dem auslösenden Ereignis startet.

Natürlich können auch Mischformen zwischen beiden Modi bestehen. Als besondere ist hier das *Center Triggering* zu nennen, bei dem die Hälfte der Daten den Zustand vor dem Ereignis beschreibt, während die andere Hälfte die Entwicklung danach dokumentiert.

Durch das Einfügen von Timern lassen sich *Posttriggered Events* auch „beliebig“ verzögern. Dies ist bei Aufzeichnungen vor dem Trigger Event prinzipbedingt nicht möglich. In diesem Modus müssen so lange kontinuierlich Werte erfasst und vorgehalten werden, bis das Ereignis auftritt. Ist der Speicher des Analyzers vollständig belegt, werden in der Regel beim Einfügen neuer Daten die ältesten überschrieben (vgl. Kapitel 4.3.2). Somit begrenzt die Speichertiefe das verwendbare Zeitfenster.

---

[2] z.B. LeCroy Logic Studio mit 16 Kanälen, 500 MSamples/s, Speicher für 20000 Messwerte [lec11]

[3] z.B. Tektronix TLA7000 Series [tek11b]

[4] engl. für Auslöser

[5] z.B. Tektronix SPI-4.2/SPI-3 Support Package [tek11a]

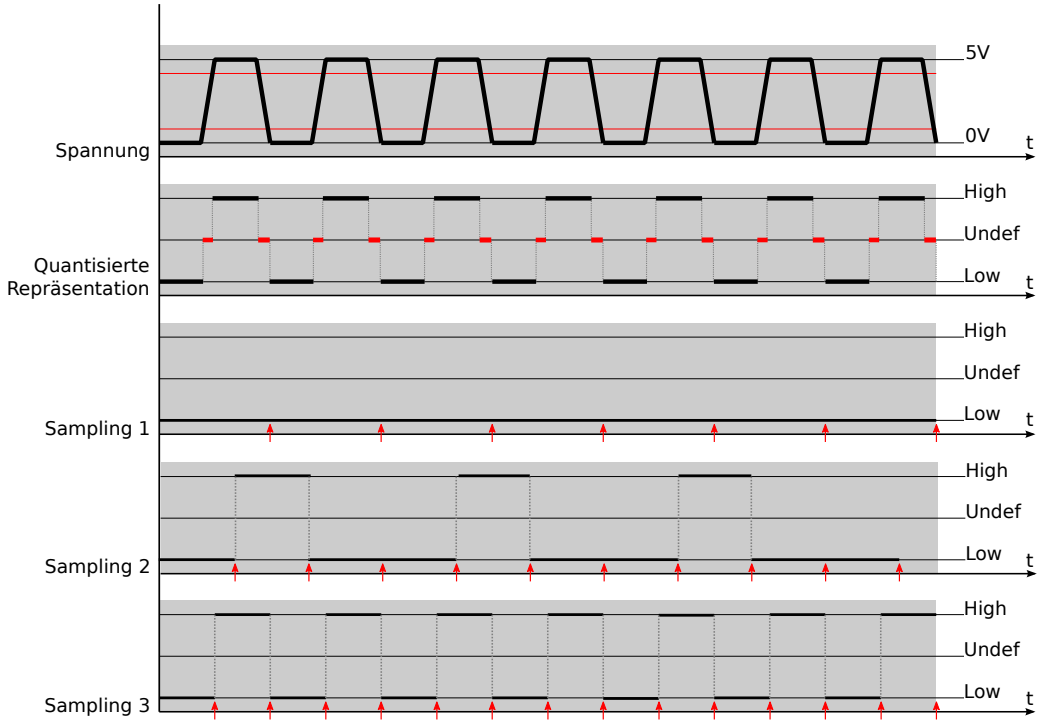


Abbildung 1: Abtastung eines Rechteck-Signals mit konstanter Frequenz F_R unter Verwendung unterschiedlicher Abtastraten F_S: $F_{\text{Sampling 1}} = F_R$, $F_{\text{Sampling 2}} = \frac{2}{3}F_R$, $F_{\text{Sampling 3}} = 2F_R$
Das zweite Diagramm zeigt eine mögliche Quantisierung der kontinuierlichen Spannung mittels der Zustände `low`, `undefined` und `high`. Die Pfeile auf den Zeitachsen der folgenden Plots markieren die Zeitpunkte der Abtastungen. Bei den ersten beiden Samplings kommt es zu Alias-Effekten. Im dritten Beispiel wird die korrekte Frequenz ermittelt.

2.2 Abtastung

In elektronischen Digitalcomputern werden logische Zustände durch Spannungen, bzw. durch die aus ihnen resultierenden Ströme, repräsentiert. Vernachlässigt man quantenmechanische Phänomene, ist die Spannung eine wert- und zeit-kontinuierliche Größe. Sie kann als stetige Funktion der Zeit aufgefasst werden.

Um diese Größe in digitalen Schaltungen verarbeiten zu können, müssen sie in eine wert- und zeit-diskrete Approximation überführt werden. Man spricht von *abtasten* oder *samplen*.

Wie im vorangehenden Kapitel ausgeführt, können Logic Analyzer in der Regel pro Eingang nur sehr wenige Zustände (z.B. *low* und *high*) erkennen. Hierzu muss der erfasste Pegel quantisiert, d.h. einem Spannungsbereich ein eindeutiger logischer Wert zugewiesen werden. Dies geschieht z.B. mit einem Schmitt-Trigger, der Pegel nach dem CMOS- oder den (L)TTL-Standards interpretiert. Durch diese Quantisierung entstehen automatisch wert-diskrete Größen.

Eingänge von externen Logic Analyzern (und auch von FPGAs) lassen sich meist auf verschiedene Spannungsbereiche programmieren, um den verschiedenen Spannungsstandards Rechnung zu tragen. Da diese Arbeit jedoch auf einer höheren Abstraktionsebene ansetzt und FPGA-intern arbeitet, wird vorausgesetzt, dass eine korrekte Quantisierung von der Hardware durchgeführt wird.

Von größerer Bedeutung für diese Ausarbeitung ist die zeitliche Diskretisierung. Hierbei wird die Entwicklung einer Größe innerhalb eines Zeitintervalls durch eine endliche Anzahl von Mess-

punkten angenähert. Bei analogen Eingängen kommen hierzu meist *sample-and-hold* Module zum Einsatz. Ein digitales Signal kann u.a. durch ein flankengesteuertes Flip-Flop in Verbindung mit einem Abtasttakt diskretisiert werden. Mathematisch lässt sich das Ergebnis der beiden oberen Diskretisierungen als endliche Folge über einer endlichen Menge interpretieren.

Beim Abtasten kann ein beliebiger Takt verwendet werden, jedoch ist der Fall einer konstanten Frequenz mathematisch genau untersucht. Für diese Betrachtung eignet sich der Fourierraum, d.h. die Frequenz-Domäne, besser als die Zeit-Domäne. Das Nyquist-Shannon-Abtasttheorem[6] stellt die Verbindung zwischen der Abtastfrequenz f_S und der maximalen zeitlichen Auflösung, der sog. *Nyquist-Frequenz* $f_N = \frac{1}{2} f_S$, her.

Shannon und Nyquist zeigten, dass jede Frequenz oberhalb f_N bei der Abtastung fälscherlicherweise als Frequenz unterhalb f_N erkannt wird. Dieser Effekt ist als *Aliasing* bekannt. Prinzipbedingt ist es im Allgemeinen nicht möglich, diese falschen Messungen rechnerisch zu korrigieren. Um Aliasing zu verhindern, muss entweder der Sampling-Stufe ein Tiefpass vorgeschaltet werden, der zu hohe Anteile des Frequenzspektrums unterdrückt, oder genügend schnell gesampelt werden.

Da viele interne Signale nur in Verbindung mit einem Takt interpretiert werden, liegt es nahe, die Signale mit demselben Takt abzutasten. Dabei ist es zwar kaum möglich kurze Glitches zu detektieren, jedoch wird *Aliasing* effektiv vermieden.

2.3 Hardwareentwurf mit *VHDL*

Wie bereits im ersten Kapitel erwähnt, erfolgt die Hardwarebeschreibung des Logic Analyzers in *VHDL*[7]. Hierbei handelt es sich neben Verilog um die Standardsprache zur synthetisierbaren Hardwarebeschreibung. Anfang der 1980er vom amerikanischen Verteidigungsministerium initiiert, wurde der Sprachkern ab 1987 durch die IEEE[8] normiert [IEEa] und seither durch Revisionen ([IEEb], [IEEc]) und Erweiterungen (z.B. [IEEe]) gepflegt. Zusätzlich sind *VHDL* sowie damit verbundene Techniken und Bibliotheken durch das IEC[9] standardisiert.

Ursprünglich wurde die Sprache als Werkzeug zur Dokumentation und Simulation digitaler Systeme konzipiert. Dabei können einfache Schaltungen bis hin zu komplexen Boards abgebildet werden. Erst später wurde die Möglichkeit geschaffen, Modelle von integrierten Schaltkreisen automatisch zu synthetisieren, d.h. auf Hardware abzubilden. Dazu kann jedoch nicht der gesamte Sprachumfang genutzt werden. Man unterscheidet daher zwischen synthetisierbaren Konstrukten und solchen, die nur simuliert werden können. In letztere Kategorie gehören beispielsweise fast alle Statements, die bei der Verhaltensbeschreibung eine absolute Wartezeit einfügen.

---

[6] If a function contains no frequencies higher than W cps *[cycles per second, entspricht hier* Hz*]*, it is completely determined by giving its ordinates at a series of points spaced $\frac{1}{2}W$ seconds apart. [Sha49]

[7] Very High Speed Integrated Circuit Hardware Description Language

[8] Institute of Electrical and Electronics Engineers, Inc.

[9] International Electrotechnical Commission

Heute existiert eine Vielzahl an Simulatoren und Synthesewerkzeugen unterschiedlicher Hersteller, die ob des standardisierten Sprachkerns relativ kompatibel zueinander arbeiten. Zu beachten ist jedoch, dass nur eine verhältnismäßig kleine Untermenge der gängigen Bibliotheken genormt ist.

Die von *VHDL* unterstützen Paradigmen fördern eine systematische Beschreibung. So besteht ein komplexer Entwurf meist aus hierarchisch angeordneten Komponenten, die im Sinne einer *Top-Down*-Entwicklung Teilaspekte der Schaltung immer feiner abgrenzen und beschreiben. Die Struktur wird primär durch drei Schlüsselwörter abgebildet: `entity`, `architecture` und `configuration`.

Jede Design-Einheit der Hierarchie wird von einer Entität (`entity`) beschrieben. Da diese nur ein Interface definiert, ist sie mit der Beschreibung der Ein- und Ausgänge einer integrierten Schaltung vergleichbar. Die Funktionalität wird durch eine Architektur (`architecture`) modelliert. Zu jeder Entität können mehrere Architekturen angeben werden, wobei für jede Instanz maximal eine gewählt werden kann. Das Schlüsselwort `configuration` erlaubt u.a. diese Selektion.

Durch die Verwendung mehrerer Architekturen können alternative Ansätze zur Lösung eines Problems angegeben werden. In der Praxis wird dies etwa genutzt, um Modelle auf die Zieltechnologie (Simulation, FPGA, *ASIC*[10], etc.) zu optimieren. Weiterhin ermöglicht dieses Konzept, verschiedene Designs und Algorithmen einfach zu vergleichen.

In *VHDL* kann die Verhaltensbeschreibung auf unterschiedliche Arten erfolgen. Grundsätzlich ist zwischen drei Möglichkeiten zu unterscheiden, wobei Mischformen möglich sind und Konzepte auf höherer Abstraktionsebene (z.B. die *Two Process Method* [Gai]) existieren:

- Die Beschreibung mit der größten Hardware-Nähe ist die **strukturelle Beschreibung** (siehe Architektur `STRUCT` in Listing 1). Hierbei wird die Funktionalität ausschließlich durch die gezielte Vernetzung existierender Modelle beschrieben. Dieses explizite Angeben einer hierarchischen Gatternetzliste ist also mit dem Zeichnen eines Schaltplans verwandt, und daher ab einer gewissen Komplexität nur bedingt effizient.

- Die **Modellierung des Datenflusses** ermöglicht eine kompaktere Implementierung von kombinatorischen Schaltungen. Hierbei werden ausschließlich nebenläufige Funktionen, wie beispielsweise arithmetische Operationen auf Signalen, durchgeführt. Signale sind Erweiterungen von Variablen aus klassischen Programmiersprachen. Sie speichern zusätzlich zu ihrem aktuellen Wert ihren letzten und erlauben das Planen von zukünftigen Änderungen. Durch das Verwenden von Funktionen findet das Design mittels Datenfluss im Vergleich zur strukturellen Beschreibung auf einer höheren Abstraktionsebene statt, ist kompakter und erhöht in der Regel die Lesbarkeit.

[10] Application-Specific Integrated Circuit, eine integrierte Schaltung, die meist von Auftragsherstellern mittels lithographischer Verfahren hergestellt wird

- Ähnlich wie nebenläufige Operationen können auch beliebig viele Prozesse eingefügt werden. Die Anweisungen innerhalb eines Prozess werden quasi sequentiell abgearbeitet, wobei die Zuweisung von Signalen erst am Ende des Prozesses, bzw. an `wait`-Statements synchron erfolgt. Die Steuerung des Kontrollflusses ist wie bei anderen Hochsprachen u.a. durch `if`-Bedingungen und Schleifen möglich. Somit lassen sich – mit einigen Einschränkungen – **algorithmische Verhaltensbeschreibungen** definieren.

2.3.1 Synthese für FPGAs

Die Generierung einer Schaltung aus der Hardwarebeschreibung läuft in der Regel in mehreren Stufen ab. Üblicherweise wird in einem ersten Schritt die Struktur des abstrakten Modells aufgelöst und Formeln vereinfacht sowie Werte berechnet (man spricht von *elaborieren*). Darüber hinaus werden algorithmische Beschreibungen auf Register-Transfer-Ebene in endliche Zustandsautomaten überführt und dann optimiert.

Hieraus wird eine hardwarenahe, aber technologieunabhängige Netzliste berechnet, die das System durch die Vernetzung elementarer Schaltungen repräsentiert. Es werden dabei große Anstrengungen unternommen auch indirekt beschriebene Strukturen (etwa Speicher) durch ein spezialisiertes Primitiv abzubilden.

Der Prozess, ausgehend von einem Verhaltensmodell oder der Register-Transfer-Ebene hin zu einer Gatterliste, wird als Synthese bezeichnet. Der Begriff beschreibt häufig jedoch auch den Gesamtprozess bis hin zur „lauffähigen“ Hardwarebeschreibung und wird in dieser Ausarbeitung meist mit der zweiten Bedeutung genutzt.

Die folgenden Schritte sind nun technologieabhängig, laufen aber zumeist vollständig automatisiert. Für ASIC enden sie etwa in Lithographie-Masken, mit denen die Chips gefertigt werden können. Dies ist aber für Forschung, Entwicklung, wie auch für Kleinserien meist zu teuer und unflexibel. Als *Meet-In-The-Middle*-Ansatz stellt die programmierbare Logik von FPGAs in den meisten Fällen eine akzeptable Lösung dar.

Die Struktur dieser Chips wird durch eine Vielzahl an Logikblöcken (Hunderte[11] bis einige Millionen[12]), die wiederum zu Makrozellen zusammengefasst werden, und einem konfigurierbarem Netzwerk zwischen diesen, geprägt. Ein Logikblock bildet eine – je nach Chip – 4- bis 6-stellige Binärfunktion mittels Look-Up-Table ab und verfügt zur Sequentialisierung über mindestens ein Speicherglied (meist ein D-Flip-Flop). Desweiteren enthalten FPGAs häufig genutzte Hardwarestrukturen, wie *Clock Manager*, Speicherblöcke oder Multiplizierer.

Die Synthesewerkzeuge müssen daher vorhandene Netzlisten zunächst auf die vom Chip angebotenen Features anpassen (man spricht vom *Mapping*), um sie dann während des *Place-and-Route*-Schrittes konkreten Strukturen auf dem Chip zuzuordnen. Die Anordnung kann dabei u.a. durch Bedingungen an die Laufzeit (*Timing-Constraints*) oder die feste Zuordnung von Signalen zu Pins beeinflusst werden.

---

[11] *Xilinx Spartan II - XC2S15* mit 432 Logikblöcken

[12] *Xilinx Virtex 7 T* mit 1955K Logikblöcken

```
1  library IEEE;
2      use IEEE.STD_LOGIC_1164.ALL;
3
4  entity NAND_GATTER is
5      port (
6          a,b : in STD_LOGIC;
7          c : out STD_LOGIC
8      );
9  end entity;
10
11 architecture STRUCT of NAND_GATTER is
12     component AND_GATTER
13         port (
14             a,b : in STD_LOGIC;
15             c : out STD_LOGIC
16         );
17     end component;
18     component INV_GATTER
19         port (
20             a: in STD_LOGIC;
21             b : out STD_LOGIC
22         );
23     end component;
24
25     signal d : STD_LOGIC;
26 begin
27     my_and : AND_GATTER port map (a => a, b => b, c => tmp);
28     my_inv : INV_GATTER port map (a => tmp, b => c);
29 end architecture;
30
31 architecture DATAFLOW of NAND_GATTER is
32 begin
33     c <= not (a and b);
34 end architecture;
35
36 architecture ALGO of NAND_GATTER is
37 begin
38     process(a,b) is
39     begin
40         if a='1' and b='1' then
41             c <= '0';
42         else
43             c <= '1';
44         end if;
45     end process;
46 end architecture;
```

Listing 1: Exemplarische Implementation eines NAND-Gatters mit zwei Eingängen als Beispiel für die strukturelle und algorithmische Beschreibung sowie per Datenfluss-Modell

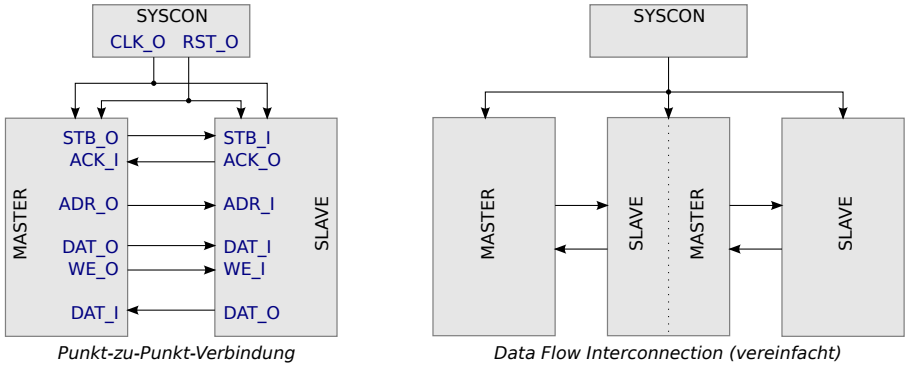


Abbildung 2: Zwei exemplarische Topologien des *WISHBONE Bus*

2.4 WISHBONE

Der *WISHBONE Bus* [Ope10] ist ein offener und freier Datenbus zur Vernetzung einzelner Komponenten innerhalb eines SoC[13]. Er ist explizit für chip-interne Kommunikation konzipiert und sieht daher eine Implementierung auf hoher Abstraktionsebene (z.B. in *VHDL* oder *Verilog*) vor. So beschreibt die Spezifikation logische Signale und verschiedene Protokolle zur Kommunikation, lässt jedoch physikalische Eigenschaften der Verbindungen offen. Daten und Adressen werden parallel übertragen, wobei die Breite beider auf 64 Bit beschränkt ist. Die Semantik von Daten und Adressen kann mit sog. *Tags* erweitert werden (s.u.).

Über einen *WISHBONE Bus* können theoretisch beliebig viele Komponenten verbunden werden. Dabei ist ein Teilnehmer entweder ein *Master* oder ein *Slave*. Der erste Typ kann aktiv Adressen auf den Bus legen und Lese- und Schreibzyklen initiieren. Ein *Slave* hingegen reagiert nur, wenn er von einem *Master* angesprochen wird. Jeder aktive Teilnehmer kann mittels eines *Handshake*-Protokolls die Kommunikation vorübergehend pausieren.

Der Bus unterstützt diverse Topologien, wobei immer mindestens ein *Master* und ein *Slave* angeschlossen sein müssen. Die einfachste Konfiguration ist daher eine Punkt-zu-Punkt-Verbindung, die exakt zwei Bus-Teilnehmer besitzt. Werden mehrere solcher Architekturen verbunden, wobei der *Slave* eines Busses als *Master* eines anderen agiert, entsteht eine *Pipeline* (vgl. *Data Flow Interconnection* in [Ope10]). Abbildung 2 illustriert beide Topologien.

Zusätzlich zu den o.g. Bus-Teilnehmern ist das *SYSCON*-Modul vorgeschrieben. Dieses stellt einen Bustakt und ein Reset-Signal, das nach dem Einschalten des Systems mindestens einen Takt lang aktiv sein muss, zur Verfügung. Sämtliche Kommunikation mit Ausnahme der Datenfluss-Steuerung (s.u.) findet synchron zur steigenden Flanke des Taktes statt.

Der einfachste Buszyklus zum Übertragen eines Datenwortes funktioniert wie in Abbildung 3 beschrieben: Im ersten Takt legt der *Master* eine Adresse auf den Bus und initiiert mit dem sog. *Strobe*-Signal den Beginn einer Übertragung. Handelt es sich um einen Schreibvorgang, wird zeitgleich auch das Datum sowie ein *write-enable*-Bit auf den Bus gelegt. Sobald der angesprochene *Slave* die geforderte Aktion durchgeführt hat, bestätigt er durch ein *acknowledge*-Bit.

[13] *System on Chip*, bezeichnet die Bündelung aller Funktionseinheiten eines Systems in einem einzigen Baustein

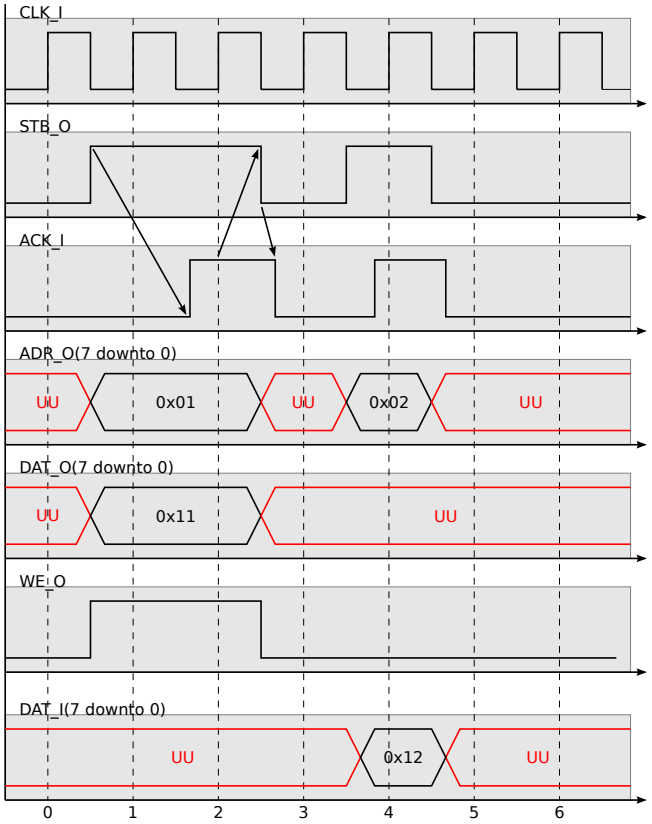


Abbildung 3: Timing-Diagramm eines Schreib- und Lesezugriffs über den *WISHBONE Bus* aus der Sicht des *Masters*. Der erste Zugriff (Takt 0 bis 2) schreibt das Wort 0x11 auf Adresse 0x01. Es demonstriert, wie der *Slave* durch Verzögern des `ACK_I` die Übertragung pausieren kann. Der zweite Zugriff (Takt 4) liest das Datum 0x12 von der Adresse 0x02. Durch die asynchrone Bestätigung kann die Übertragung in einem Taktzyklus abgeschlossen werden.

Der Master erkennt die Bestätigung mit der nächsten steigenden Flanke des Taktes, liest ggf. das Datenwort vom Bus und beendet die Übertragung durch Absenken des *Strobe*-Signals.

Für simple Varianten erlaubt der *WISHBONE Bus* eine kombinatorische Bestätigung durch den *Slave*. Hierdurch kann jedoch die maximale Übertragungsfrequenz reduziert und somit der Datendurchsatz geschmälert werden. Daher existiert mit den *WISHBONE Registered Feedback Bus Cycles* eine Erweiterung des Protokolls, bei der die Bestätigung nur takt-synchron geschieht und vom *Master* mittels Adress-Tags zusätzliche Informationen zur nächsten Adresse zur Verfügung gestellt werden. Durch diesen Ansatz kann die Übertragung von Datenblöcken mit zusammenhängenden Adressen beschleunigt werden, der randomisierte Zugriff wird jedoch verzögert.

2.5 JTag

JTag[14] bezeichnet ein durch die IEEE als 1149.1 [IEEd] standardisiertes Verfahren zum Überprüfen von ICs und deren Verbindung zur Peripherie auf Leiterplatten. Dieser sog. *Boundary Scan* ermöglicht beispielsweise eine vollständig automatisierte Verifikation von Boards

[14] Joint Test Action Group

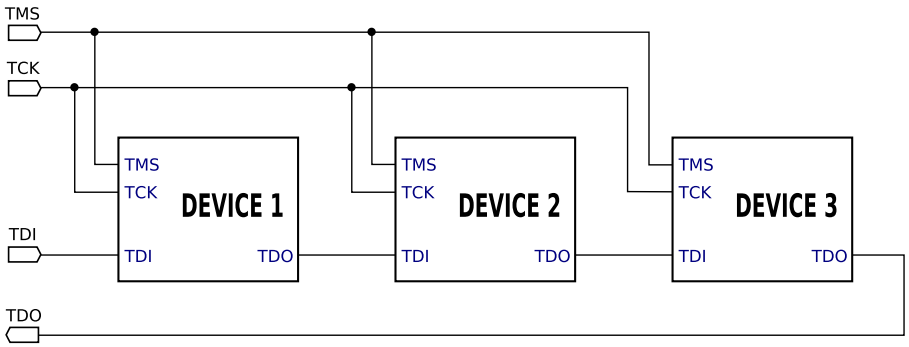


Abbildung 4: Beispiel einer *JTag*-Chain nach [Wik11a]. Im Regelbetrieb befinden sich alle Geräte im gleichen *TAP-Controller*-Zustand, da `TMS` und `TCK` parallel angeschlossen sind. Die aktiven *Schieberegister* aller Teilnehmer werden durch die Reihenschaltung von `TDI` und `TDO` konkateniert.

während des Fertigungsprozesses. Da der Standard nur einen minimalen Funktionsumfang definiert und die Schnittstelle modular konzipiert ist, wird der *JTag*-Port heutzutage auch für zusätzliche Aufgaben verwendet. So können über *JTag* Speicherbausteine und FPGA programmiert oder Mikroprozessoren mit Debuggern verbunden werden.

Bei *JTag* handelt sich um eine asymmetrische Schnittstelle, die einen Host (z.B. ein Prüfgerät) und beliebig viele Clients verbindet. Jegliche Kommunikation geht vom Host aus; Clients ist es nicht möglich eine Übertragung zu initiieren. Alle *JTag*-kompatiblen Clients müssen einen TAP[15] besitzen, der über mindestens vier dedizierte Pins angesprochen wird. Da der Datenausgang eines Bausteins gemäß des in Abbildung 4 gezeigten Schemas an den Eingang eines weiteren angeschlossen werden kann, ist die Verbindung zu einer Leiterplatte unabhängig von der Anzahl der darauf befindlichen *JTag*-Geräte.

Die verschiedenen Prüffunktionen eines Chips können über das *Instruktionsregister* ausgewählt werden. Hierbei handelt es sich um ein Schieberegister, auf das bitweise zugegriffen wird. Jeder Instruktion ist ein Daten-Schieberegister zugeordnet. Die Länge des Registers ist ausschließlich von dem ausgewählten Befehl abhängig und darf sich während des Betriebs nicht ändern. Durch die Reihenschaltung aller Bausteine zu einer sog. *JTag-Kette* werden effektiv die Schieberegister aller Bausteine zu einem größeren konkateniert.

Die Länge aller Register sowie die Kodierung der einzelnen Befehle ist bauteil-spezifisch und wird dem Prüfgerät über das BDSL Dateiformat[16] vom IC-Hersteller bekannt gegeben. In der Spezifikation werden die vier Befehle `BYPASS`, `EXTEST`, `PRELOAD` und `SAMPLE` gefordert ([IEEd], Kap. 7). Die `BYPASS`-Instruktion selektiert ein 1 Bit-langes Daten-Schieberegister, das keine weitere Funktionalität hat. Hierdurch kann mit anderen Geräten am Bus kommuniziert werden, ohne den normalen Betrieb des im `BYPASS`-befindlichen ICs zu stören. Die weiteren drei Instruktionen dienen dem *Boundary Scan* und sind für die weitere Ausarbeitung nicht von Bedeutung.

[15] Test Access Port

[16] Boundary Scan Description Language, erstmals standisiert durch IEEE 1149.1 [IEEd]

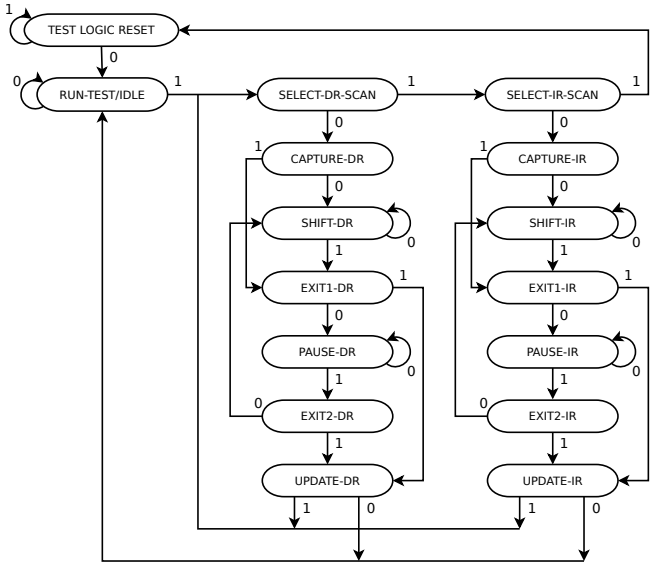


Abbildung 5: Zustandsübergange des *JTag* TAP-Controllers. Die Ziffern repräsentieren den logischen Zustand des `TMS` Singals. [Wik11b]

Weiter schlägt der Standard optionale Identifikationsregister vor, die es dem Host erlauben, automatisiert die *JTag*-Kette zu berechnen. Jedem Hersteller steht es frei, weitere Befehle zu unterstützen. So besitzen beispielsweise *Xilinx Spartan-3* FPGAs zwei Instruktionen, deren Datenregister durch programmierbare Logik implementiert werden können und somit der Anwendung begrenzten Zugriff auf den Bus ermöglichen.

Jeder angeschlossene Baustein verfügt über einen *TAP Controller*, der mittels `TMS`-Signal[17] gesteuert wird. Beim Controller handelt es sich um einen endlichen Automaten, der synchron zum Bustakt `TCK` arbeitet und die Zustände gemäß Abbildung 5 wechselt. Da die beiden relevanten Pins parallel an alle Bus-Teilnehmer angeschlossen sind, sollten sich alle ICs im gleichen Zustand befinden.

Die Übergänge sind dabei so definiert, dass durch Halten eines `high`s am `TMS` binnen maximal fünf Taktzyklen die *TAP Controller* in den Reset-Zustand wechseln. In diesem Zustand ist das gesamte *JTag*-Interface deaktiviert und beeinflusst die Funktion des Chips nicht.

Befindet sich ein Gerät im *Boundary Scan*-Modus sind in der Regel alle Ausgänge des ICs von der eigentlichen Logik getrennt und werden ausschließlich über *JTag* gesteuert. Ein Testprogramm setzt daher typischerweise für jeden Baustein verschiedene Testmuster und liest bei angebundenen ICs die Werte zurück. Somit lassen sich fehlerhafte Verbindungen aufspüren.

3 State of the Art

Logic Analyzer sind teils stark spezialisierte Messgeräte und zeigen daher große Unterschiede in Funktion und Ausstattung. An einer kleinen Auswahl an Logic Analyzern speziell für FPGAs gibt dieses Kapitel eine vertiefende Übersicht über bereits existierenden Lösungen.

[17] Test Mode Select

Die beiden größten Hersteller von FPGAs bieten mit *Xilinx ChipScope™ Pro* und *Altera SignalTap™* nur proprietäre und kostenpflichtige Verifikationswerkzeuge an, die in den FPGA integriert werden und – neben Schnittstellen – ohne externe Hardware auskommen. Da der für diese Arbeit relevante Funktionsumfang ähnlich ist, reicht es aus, nur eines dieser Produkte zu betrachten; in diesem Fall *ChipScope Pro*.

Als freie und kostenlose Alternativen werden die ebenfalls integrierten Logic Analyzer *Sump*[18] und *OpenVeriFla* vorgestellt.

Die in diesem Kapitel genannten Fakten wurden bewusst hauptsächlich den offiziellen Homepages und Dokumentationen entnommen. Diese werden daher nicht mehrfach aufgeführt. Unabhängige Quellen sind explizit kenntlich gemacht.

3.1 ChipScope Pro Tools

Bei den *ChipScope Pro™ Tools*[19] von *Xilinx Inc.* handelt es sich um eine Sammlung von IP-Cores und Werkzeugen, um diese zu konfigurieren und zu steuern. Die Software ist dabei eng mit der *Xilinx*-eigenen Entwicklungsumgebung *ISE* verzahnt und ermöglicht somit ein komfortables Arbeiten. Sie kann jedoch auch separat genutzt werden. Als Zielhardware werden nur die gängigen *Xilinx* FPGAs unterstützt, Chips anderer Anbieter können nicht überwacht werden. Der Einsatz von *ChipScope Pro* ist nur in Verbindung mit einem steuerndem Host (meist ein PC oder ein kompatibler diskreter Logic Analyzer) möglich.

Für den Einsatz im FPGA stehen vier Module zur Verfügung:

- `ICON` regelt die Kommunikation zwischen dem PC und den anderen *ChipScope Pro* Modulen. Als Schnittstelle zur Außenwelt wird der JTag-Port verwendet, wodurch keine externen Pins belegt werden, jedoch mindestens ein *JTag-Primitiv* in Anspruch genommen wird und somit der zu überwachenden Logik nicht mehr zur Verfügung steht.
- `ILA` stellt den eigentlichen integrierten Logic Analyzer zur Verfügung und ist das Modul, dessen Funktionalität im Folgenden genauer betrachtet wird.
- `VIO` bildet eine Ein-/Ausgabe-Schicht. Es können Signale synchron und asynchron geschrieben und gelesen werden. Da jedoch kein Zwischenspeicher verwendet wird, ist die Bandbreite sehr begrenzt.
- `ATC2`[20] ermöglicht es, während des Betriebs interne Signale auszuwählen und auf FPGA-Pins auszugeben, um sie dann mittels eines externen Logic Analyzers zu protokollieren.

[18] vom Autor als „FPGA basierter Logikanalysator“ bezeichnet, jedoch zur Unterscheidung von anderen Lösungen allgemeinhin nach der Domain `http://sump.org` benannt

[19] die Ausführungen in diesem Kapitel beziehen sich auf Version 13.2. [Xil] [Xil11b]

[20] Agilent Trace Core 2

ChipScope Pro kann auf zwei Arten in einen Hardwareentwurf eingebunden werden. Der klassische Ansatz besteht in der expliziten Instanziierung der Komponenten in der Hardwarebeschreibungssprache. Hierdurch können bereits während der Entwicklung explizite Überwachungspunkte vorgesehen werden, was sich positiv auf die Struktur des Entwurfs auswirken kann, jedoch auch in der Regel zu einer relativ unflexiblen Überwachung führt.

Die zweite Möglichkeit resultiert aus der hohen Integration mit den Synthesewerkzeugen. So können mittels entsprechender Werkzeuge aus der Gatternetzliste der zu untersuchenden Schaltung beliebige Signale ausgewählt und mit dem Logic Analyzer verbunden werden. Im Gegensatz zur erst genannten Methode können mit diesem Ansatz einfacher Signale mehrerer Entitäten überwacht werden, da die strenge strukturelle Kapselung nicht auf Netzlisten abgebildet wird.

3.1.1 Integrierter Logic Analyzer

Der integrierte Logic Analyzer verfügt über getrennte Daten- und Triggereingänge, wobei bis zu 4096 Bit parallel aufgezeichnet werden können. Abhängig von der Datenbreite können bis zu 131 072 Datensätze zwischengespeichert werden. Diese werden in den BRAM-Zellen des FPGAs abgelegt, wodurch der überwachten Anwendung weniger integriertes RAM zur Verfügung steht. Das Sampling findet dabei synchron zu einem zugeführten Takt statt, der von der zu überwachenden Logik zur Verfügung gestellt wird. Da diese Logik häufig nahe der maximal möglichen Frequenz betrieben wird, eignen sich die erfassten Daten nur bedingt, um das *Timing* zu bewerten oder *Hazards* zu untersuchen (vgl. Kapitel 2.2).

Der Analyzer erlaubt das Triggern auf bis zu 16 Ports, wobei die Breite jedes Ports maximal 256 Bit beträgt. Einem Port können bis zu 16 boolesche Funktionen zugewiesen werden, um kombinatorisch Ereignisse zu definieren. Alternativ können die Formeln zu einer Triggersequenz, die sequentiell erfüllt werden muss, verbunden werden. Weiter ist es möglich, die Triggerlogik während der Aufzeichnung darüber entscheiden zu lassen, ob ein Datensatz gespeichert oder verworfen werden soll.

Sowohl Daten- als auch Triggereingänge werden bei der Synthese mit dem zu untersuchenden System verbunden. Eine Anpassung oder das Abschalten einiger Eingänge, um die Datensatzgröße zu reduzieren, ist nicht möglich.

Für die Aufzeichnung stehen einige Betriebsmodi zur Verfügung. So kann beispielsweise der Speicher in mehrere Bereiche gleicher Größe unterteilt werden. Ein Trigger-Ereignis stößt dann das Schreiben bis zur Bereichsgrenze an (vgl. *Window Capture Mode* [Xil11b]). Somit lassen sich mehrere kurze Ereignisse protokollieren. Alternativ kann die Länge einer Aufzeichnung angegeben werden, die eine flexiblere Fragmentierung ermöglicht (vgl. *N Samples Capture Mode*).

3.1.2 Externer Logic Analyzer

Die beiden Marktführer im Bereich diskreter Logic Analyzer *Agilent Technologies, Inc* und *Tektronix, Inc.* bieten beide Hilfsmittel, die es ermöglichen, interne Signale mittels externer Messgeräte zu erfassen. Da die von *Tektronix* angebotene Lösung *FPGAView* dem Interface von *Agilent*

grundsätzlich recht ähnlich ist, wird hier nur die zweite Variante, die auf *Xilinx ChipScope Pro* aufbaut, vorgestellt.

Hierzu werden wie bei der internen Lösung vor der Synthese alle relevanten Signale ausgewählt und mit dem Analysemodul (in diesem Fall `ATC2`) verbunden. Zur Laufzeit kann nun eine Untermenge dieser Signale direkt an dafür vorgesehene und mit einem externen Logic Analyzer verbundene FPGA-Pins umleitet werden.

Dieser Ansatz ist natürlich nur möglich, wenn genügend freie Pins zur Verfügung stehen, bzw. wenn zur Entwicklung ein spezielles Board mit entsprechenden Anschlüssen verwendet wird. Durch den Einsatz externer Messgeräte, die in der Regel eine höhere Aufzeichnungsgeschwindigkeit ermöglichen, können auch o.g. Probleme untersucht werden, für die sich integrierte Logic Analyzer nicht eignen. Zusätzlich verfügen diese diskreten Geräte zumeist über eine leistungsfähigere Software, die das Interpretieren der Messdaten vereinfacht.

3.2 Sump

Michael Poppitz, der Entwickler von *Sump*, beschreibt sein Projekte als einen „FPGA basierten Logikanalysator [..] für den Heimbedarf“. Es wurde auf Basis des *Xilinx Spartan-3 Starter Kit* umgesetzt und ist als diskreter Logic Analyzer zum Untersuchen externer Schaltungen ausgelegt. Da der Analyzer jedoch in *VHDL* implementiert wurde und unter der freien GNU GPL lizenziert ist, lässt er sich auch einfach kapseln und als integriertes Messgerät zum Protokollieren interner Logik nutzen. Ein *Fork* von *Sump* wird auch in speziell dafür konzipierter Hardware als Hobby Logic Analyzer vertrieben [Whe11].

Die Bedienung sowie die Auswertung der Messungen wird an einem PC vorgenommen, der über die *RS232*-Schnittstelle (ugs. „serielle Schnittstelle“) mit dem Analyzer verbunden wird. Die Quellen der hierfür nötigen Software liegen der Hardwarebeschreibung bei und sind in Java implementiert. Das Programm verfügt über eine graphische Oberfläche und erlaubt das Konfigurieren des Messgerätes, sowie die direkte Visualisierung der Messung. Darüber hinaus können die erfassten Daten mittels eines SPI- und I^2C-Bus Protocol Analyzer interpretiert werden. Alternativ ist die Steuerung auch mit der freien Logic Analyzer Software *Sigrok* (siehe Kapitel 3.4) oder dem Eclipse-basierten *LogicAnalyzer* [Wei11] möglich.

Der Analyzer besitzt 32 Eingänge. Durch die Konzeption als diskretes Messgerät kann die Anzahl nicht trivial geändert werden. Es stehen vier Trigger zur Verfügung, die jeweils entweder einen Datenwert über alle Eingänge oder eine 32 Bit lange Sequenz auf einem Pin erkennen. Zwischen einer *Pre-* und *Posttrigger*-Operation lässt sich auf vier Datensätze genau variieren.

Mit einem *Spartan-3* FPGA lässt sich eine maximale Aufzeichnungsgeschwindigkeit von 100 MHz realisieren, wobei der Takt durch eine externe Quelle zugeführt wird. Weiterhin besteht die Möglichkeit, nur die Hälfte der Eingänge, dafür aber mit doppelter Geschwindigkeit abzutasten. Hierzu werden die Werte sowohl bei der steigenden wie auch bei der fallenden Flanke des Taktes erfasst. Sämtliche o.g. Einstellungen inklusive der Trigger lassen sich zur Laufzeit über die RS232-Schnittstelle konfigurieren.

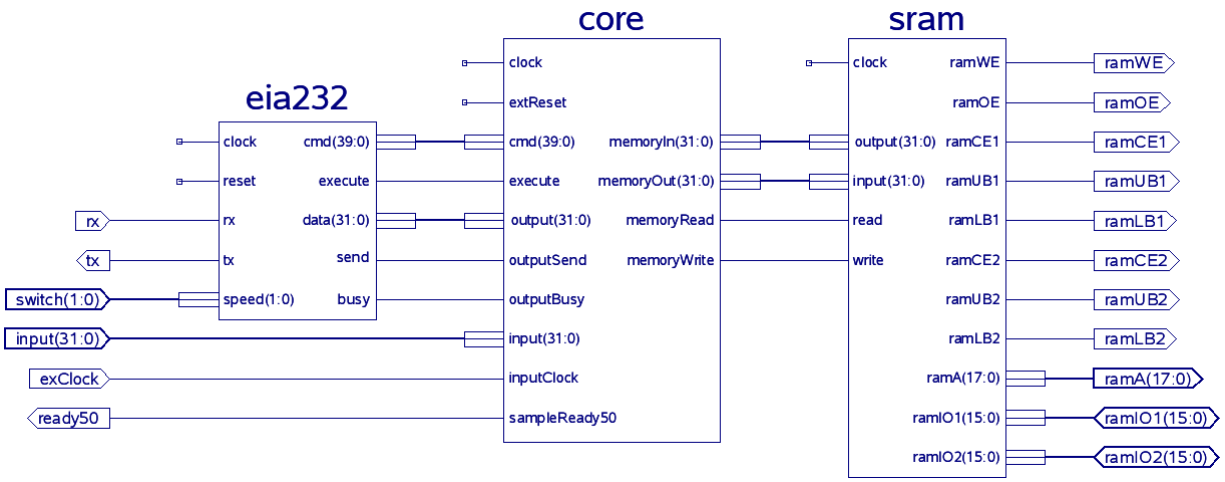


Abbildung 6: Top-Level Schema des *Sump* Logic Analyers [Pop07]. Der Hardwareentwurf ist in drei Hauptkomponenten geteilt, wobei die Interaktion mit dem PC über die RS232-Schnittstelle und der Speicherzugriff vom Kern getrennt sind

Die Datensätze werden in einem externen SRAM abgelegt und benötigen pro Frame 4 Byte. Da das Referenzboard über einen 1MB großen RAM-Chip verfügt, können rund 250 000 Sätze erfasst werden. Der externe Speicher schränkt die Verwendbarkeit von *Sump* als integrierter Logic Analyzer ein, da – falls vorhanden – diese externen Bausteine meist von der zu überwachenden Logik genutzt werden und das Nachrüsten eines weiteren Speicherbusses häufig nicht möglich ist. Wie in Abbildung 6 gezeigt, ist jedoch die Speicherverwaltung gekapselt und nicht Teil des Logic Analyzer-Kerns. Durch Anpassen dieses Moduls könnte die Abhängigkeit von externem Speicher gelöst werden.

3.3 OpenVeriFla

OpenVeriFla ist ein an der Universität Bukarest entwickelter integrierter Logic Analyzer, der auf die Verwendung im akademischen Rahmen, besonders im Bereich der Lehre zielt. Sein Funktionsumfang ist sehr beschränkt und soll daher als Beispiel für ein minimalistisches Konzept dienen.

Auch dieser Analyzer besteht aus zwei Teilen, wobei nur das Sampling und ein rudimentärer Trigger in Hardware gelöst wurden, während sämtliche höheren Funktionen durch die Steuersoftware am PC übernommen werden. Die Hardwarebeschreibung wurde in Verilog implementiert, die PC-Software in Java.

Bemerkenswert an der Software ist der Export der aufgezeichneten Daten als Verhaltensmodell in Verilog. Werden beispielsweise die Werte aller Eingänge einer zu testenden Schaltung überwacht, lassen sich auf diese Weise schnell realistische Test-Stimuli erzeugen. Nachteil dieses Ansatzes ist jedoch, dass im Gegensatz zu primitiveren Export-Formaten ein Verilog-Simulator zur Visualisierung der Messungen erforderlich ist.

Der Großteil der Konfiguration des Analyzers wird vor der Synthese durchgeführt und lässt sich während der Laufzeit nicht mehr beeinflussen. Über die Bediensoftware kann lediglich der Trigger scharf geschaltet werden, sowie die erfassten Daten ausgelesen werden. Die Kommunikation findet über eine RS232-Schnittstelle statt. Insbesondere sind die Triggerwerte fest

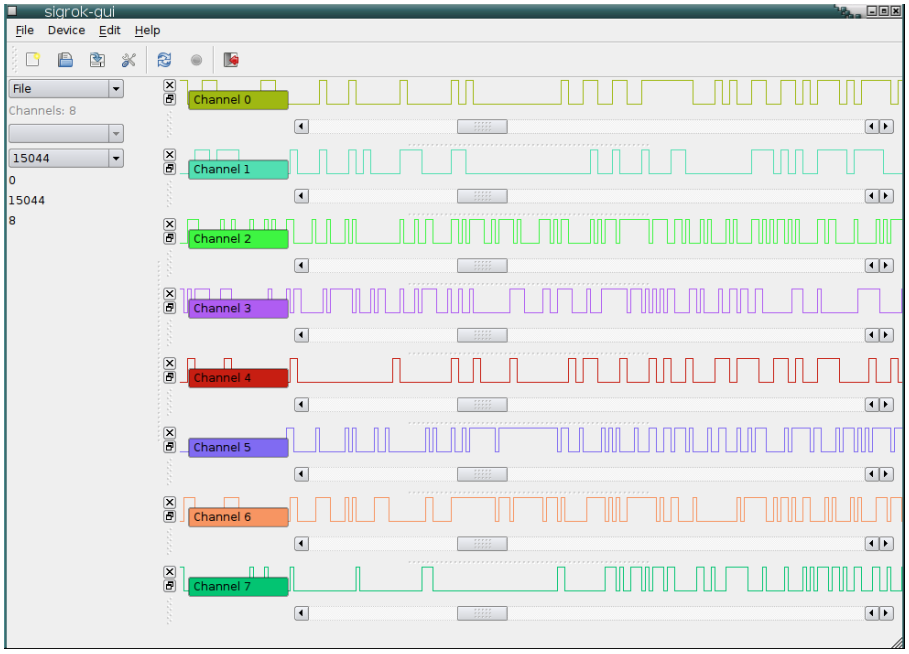


Abbildung 7: Screenshot [sig11] des Logic Analyzers unter Linux

kodiert, wodurch ein effektives Debuggen erschwert wird. So kann während der Ausführung einer Anwendung nur ein Ereignistyp untersucht werden: Das Ändern des Fokusses erfordert eine Rekonfiguration des FPGAs und somit im Allgemeinen einen Neustart der Anwendung.

3.4 Sigrok

Sigrok stellt im Gegensatz zu vorherigen Logic Analyzern keine eigene Hardware zur Verfügung. Es ist vielmehr als einheitliches Software-Interface für eine Vielzahl an freien und kommerziellen Logic Analyzern konzipiert. Die Software ist in *C++* geschrieben und verfügt sowohl über ein Kommandozeilen-Interface wie auch eine graphische Benutzerschnittstelle, die mit *QT4*[21] realisiert wurde. Die Software lässt sich unter *FreeBSD*, *Linux*, *Mac OSX* und *Windows* verwenden. [sig11]

Durch eine dynamisch-linkende PlugIn-Struktur lässt sich das System um Datei-Import und -Export-Formate, Protocol Analyzer und Hardwaretreiber erweitern. Treiber greifen dabei auf eine betriebssystemunabhängige *USB*- oder *RS232*- Schnittstellenabstraktion zu und können sich daher meist auf die Implementierung des Protokolls beschränken.

Sigrok erlaubt das Konfigurieren von Trigger und weiterer hardwarespezifischer Eigenschaften, sowie das Herunterladen und Visualisieren aufgezeichneter Daten. Da sich das Programm derzeit noch in einer frühen Entwicklungsphase befindet, stehen noch nicht alle geplanten Funktionen zur Verfügung.

3.5 Zusammenfassung

Alle untersuchten integrierten Logic Analyzer zeichnen sich durch eine ähnliche Architektur aus. So sind möglichst viele der Features durch Software außerhalb des FPGAs realisiert, um

[21] plattformübergreifende GUI-Bibliothek

nur ein Minimum an Hardware zu benötigen. Die Kommunikation zwischen beiden Modulen wird dabei von seriellen Schnittstellen übernommen, die im Vergleich zu parallelen Pendants weniger Pins verbrauchen. In den gewählten Beispielen kamen die RS232-Schnittstelle, die sehr einfach zu implementieren ist und lediglich zwei Pins benötigt, sowie das JTag-Interface, das häufig schon zur Konfiguration des FPGAs genutzt wird, zum Einsatz.

Da der Datendurchsatz der verwendeten Schnittstellen sehr begrenzt ist, verwenden alle Produkte einen Zwischenspeicher. Eine kontinuierliche Aufzeichnung ist somit nur begrenzt möglich. Daher kommen Trigger zum Einsatz, die es ermöglichen, relevante Ereignisse zu erkennen und nur diese aufzuzeichnen. Die Komplexität der Trigger unterschiedet sich dabei signifikant, wobei *ChipScope Pro* den mit Abstand Leistungsfähigsten bietet. Dieses Messgerät ermöglicht zusätzlich den Trigger während der Aufzeichnung als Filter für nicht benötigte Datensätze zu verwenden und somit den Speicherplatz effektiver zu nutzen.

Als Puffer wurden je nach Produkt entweder FPGA-eigene Speicherbereiche oder externe Bausteine verwendet. Externe Bausteine verfügen zwar über eine höhere Kapazität, benötigen aber zusätzliche Pins und Randbeschaltung. Nur durch die Verwendung interner RAM-Blöcke kann die Portabilität des Analyzers maximiert werden.

Große Unterschiede konnten im Bereich der Portabilität und Bediensoftware ausgemacht werden. Bei *ChipScope Pro* handelt es sich um ein auf *Xilinx*-Produkte zugeschnittenes Werkzeug, das eng mit der dazugehörigen *Toolchain* verknüpft ist und hierdurch sehr komfortabel konfiguriert und eingesetzt werden kann.

OpenVeriFla und *Sump* sind im Gegensatz dazu offene Projekte, die versuchen möglichst einfach an die eingesetzte Hardware angepasst werden zu können. Diese Portabilität bedingt eine niedrige Integration in vorhandene Strukturen und erfordert beim Wechsel der Hardware eventuell Anpassungen (z.B. am Speichermanagement). Außerdem handelt es sich um Entwicklungen, die primär von einer Person gepflegt wurden[22], und somit nur rudimentäre Funktionen implementieren konnten.

Mit *Sigrok* wurde eine freie Bediensoftware präsentiert, die bei aktiver Weiterentwicklung das Potential hat, eine einheitliche Schnittstelle für Logic Analyzer aller Art zu werden, und somit als Langzeitziel die mangelnde Integration der freien Messgeräte ausgleichen könnte.

4 Konzeption

Eine Untersuchung der im vorherigen Kapitel vorgestellen Logic Analyzer zeigte, dass alle Projekte unterschiedliche Schwerpunkte setzen. Von diesen sind nur *ChipScope Pro*, das jedoch unfrei und kostenpflichtig ist, und *OpenVeriFla*, das zu wenig Features bietet, als integrierte Analyzer konzipiert. Der als diskreter Analyzer ausgelegte *Sump* verfügt zwar über die benötigten Features, eignet sich durch seine unflexiblen Eingänge, die hierauf ausgelegte Hardwarestruktur und das daran angepasste Kommunikationsprotokoll, nur bedingt als ein integrierter Analyzer.

[22] letzte Änderungen in der offiziellen Version von *Sump* im Jahr 2007, von *openVeriFla* im Jahr 2009

Daher fiel die Entscheidung, keines der offenen Projekte zu erweitern, sondern ein neues Konzept aufzustellen. Dabei wird ebenfalls eine Teilung des Projektes in eine reagierende Hardware und die steuernde Software vollzogen. Die weitere Ausarbeitung konzentriert sich auf die Hardware-Seite. Eine grobe Beschreibung der Softwarestruktur kann dem Appendix B entnommen werden.

Zunächst werden die wichtigsten Anforderungen formuliert, um im Folgenden eine Gewichtung vorzunehmen und daraus ein konkretes Design zu erarbeiten.

4.1 Anforderungen

Zu entwickeln ist ein **lizenzfreier** und **quelloffener** Logic Analyzer in *VHDL*. Das Produkt soll in einen zu überwachenden Entwurf (künftig als *Anwendung* bezeichnet) integriert werden können. Die Anwender des Analyzers sind somit vor allem Entwickler, denen die Instantiierung des Analyzers sowie dessen Konfiguration mittels *VHDL* ohne zusätzliche Automatisierung zuzutrauen ist. Der Entwurf soll synthetisierbar sein und die folgende Eigenschaften erfüllen:

- **Variable Eingänge** Da die Anzahl der zu überwachenden Signale von Anwendung zu Anwendung variiert, muss diese vor der Synthese konfigurierbar sein.
- **Synchrone Aufzeichnung** Die Aufzeichnung soll synchron zu einem externen[23] Takt erfolgen, weil sich hierdurch – wie in Kapitel 2.2 ausgeführt – *Aliasing*-Effekte auf das für die Anwendung geltende Maß reduzieren. Eine Taktanpassung (z.B. Frequenzteilung) sowie eine verlustfreie Datenkompression sind wünschenswert.
- **Asynchrones Lesen** Die am Eingang anliegenden Signale sollen jederzeit durch den Host lesbar sein. Insbesondere muss dies auch funktionieren, wenn kein externer Takt anliegt.
- **Universeller Trigger** Der Analyzer soll über einen universellen Trigger verfügen, der möglichst viele Anwendungsfälle abdeckt. Insbesondere sollen verschiedene Ereignisarten erkannt werden. Zusätzlich soll das Produkt mindestens *Pre-* und *Posttriggering* unterstützen.
- **Konfiguration zur Laufzeit** Wo sinnvoll sollen Einstellungen zur Laufzeit möglich sein. Die Reduktion des Speicherverbrauchs während der Aufzeichnung durch das Deaktivieren einzelner Eingänge soll ermöglicht werden.
- **Erweiterbarkeit** Das Design sowie die Struktur der Implementierung sollen das nachträgliche Hinzufügen neuer Funktionen unterstützen.
- **Portabilität** Der zu entwickelnde Analyzer soll möglichst hardwareunabhängig sein. Insbesondere muss der Speicherzugriff und die Kommunikations-Schnittstelle zum Host anpassbar sein.

[23] in der Regel durch die Anwendung zur Verfügung gestellt

- **Automatisierte Verifikation** Das zu entwickelnde Produkt soll Möglichkeiten bieten die Funktion möglichst aller Module automatisiert zu testen. Da dieses Feature jedoch während des regulären Betriebs nicht benötigt wird, soll es sich nicht auf die benötigte Fläche im FPGA auswirken.

- **Performance und Hardwarebedarf** Es soll eine möglichst hohe Sampling-Frequenz erreicht werden. Eine Frequenz von etwa 100 MHz auf einem *Xilinx Spartan-3* (Speedgrade -4) ist wünschenswert. Der Flächenbedarf soll dabei minimal sein.

Im Folgenden wird nun aus diesen losen Forderungen ein Pflichtenkatalog erstellt. Dabei als verpflichtend betrachtete Punkte werden durch das Wort „*soll*“ verdeutlicht.

4.2 Struktur des Designs

Das spezifizierte Produkt ist eine allgemeine Beschreibung eines Logic Analyzers ohne spezielle Schwerpunkte. Dies wiederum muss als Anforderung aufgefasst werden: Das zu entwickelnde Produkt *soll* möglichst flexibel sein, dabei jedoch kaum Hardware-Anforderungen stellen. In der Software-Entwicklung handelt es sich hierbei um ein klassisches Spannungsfeld, das sich aber für dieses Projekt aufgrund der Natur der Synthese von Hardware aus *VHDL*-Code entschärfen lässt. So können in *VHDL* mehrere Alternativen angegeben werden, die in der Elaborations- sowie der anschließenden Optimierungsphase ausgewählt, bzw. verworfen werden. Dies erhöht zwar die Synthesezeit, wirkt sich jedoch im Allgemeinen nicht negativ auf die Performance des Produktes aus.

Im Gegensatz zu wenigen Entitäten, die jeweils eine Vielzahl an Funktionen realisieren, bietet sich daher ein modulares Framework an, dessen Basisfunktionen in dieser Arbeit entwickelt werden *sollen*. Dabei *soll* möglichst jede Funktionsgruppe durch eine eigene Komponente implementiert werden. Im einfachsten Fall besteht dann eine Anpassung darin, eine alternative `architecture` anzugeben, und diese entsprechend in das Projekt zu konfigurieren. Dies kann durch das Hinzufügen neuer Dateien bewerkstelligt werden, ohne dass Änderungen am Kern des Frameworks durchgeführt werden müssen.

Die einzelnen Entitäten *sollen* dabei in einer baumartigen Struktur instanziiert werden. Im Unterschied zu einem planaren Modell, in dem die Wurzel (Top-Modul des Frameworks) alle anderen Funktionen direkt einbindet, verursacht dieser Ansatz zwar mehr Code-Overhead, lässt sich aber flexibler anpassen.

Alle zur Synthese getroffenen Konfigurationen *sollen* einheitlich definiert und zentral abgelegt werden können. Dazu bietet *VHDL* grundsätzlich zwei Möglichkeiten: Durch `generics` lassen sich einer Entität Konfigurationen übergeben. Diese sind dann für die aktive Architektur sichtbar, müssen jedoch explizit an Komponenten, die tiefer in der Hierarchie liegen, weitergereicht werden. Durch dieses Konzept können mehrere Instanzen der gleichen Entität mit unterschiedlichen Konfigurationen erzeugt werden. Der Nachteil besteht aber darin, dass das Hinzufügen einer neuen Einstellung in einem Modul dazu führt, dass diese Eigenschaft ebenfalls in allen Modulen, die in der Hierarchie darüber liegen, ergänzt werden muss.

Über das Schlüsselwort `constant` erzeugte Konstanten weisen diese Probleme nicht auf: Ihre Sichtbarkeit hängt vom Ort ihrer Definition ab. Ein Ansatz besteht darin, ein Paket zu erzeugen, dass alle Konfigurationen mittels Konstanten bekannt macht. Durch das Importieren des Pakets in jeder Entität des Frameworks stehen potentiell jedem Modul alle Konfigurationen zur Verfügung. Weiterhin lassen sich hierdurch von der Konfiguration abhängige Datentypen zum strukturierten Austausch von Informationen zwischen einzelnen Modulen definieren. Der primäre Nachteil ist, dass zwei unterschiedlich konfigurierte Analyzer nicht mehr komfortabel gleichzeitig erzeugt werden können. Da jedoch die Vorteile überwiegen, das zweite Problem kaum praxis-relevant ist und sich im Zweifelsfall durch die getrennte Generierung von Netzlisten umgehen lässt, *soll* ein Konfigurationspaket verwendet werden.

Um die Portabilität zu garantieren, *sollen* bei der Implementation des Framework nur durch die *IEEE* spezifizierte Sprachkonstrukte und Bibliotheken zum Einsatz kommen. Für Hardwaretreiber (vgl. Kapitel 4.3 und 4.4) darf diese Anforderung zum Instantiieren von FPGA-spezifischen Primitiven und Makros ignoriert werden.

Die Freiheit des Frameworks *soll* durch Verwendung der *GNU General Public License v3.0* [Fre07] unterstrichen werden. Das Framework sowie die Treiber *sollen* weiterhin nur Komponenten verwenden, die unter einer äquivalenten Lizenz stehen. Eine Ausnahme besteht ebenfalls in der Instantiierung von herstellerspezifischen Komponenten.

4.3 Speicher

4.3.1 Technologie

Eine zentrale Komponente eines jeden Logic Analyzers ist der Speicher zum Erfassen und Puffern der Messwerte. Gängige FPGAs bieten hierzu meist drei Ansätze:

- Die Anbindung von **externem Speicher** wurde bereits kurz im vorherigen Kapitel erörtert. Dieser Ansatz kann zwar – verglichen mit den anderen Alternativen – um Größenordnungen mehr Speichertiefe ($\gg$ 1 GBit) zur Verfügung stellen, benötigt aber auch eine große Anzahl von Daten- und Adressleitungen, die aus dem FPGA heraus geführt werden müssen. Da eine möglichst hohe Sampling-Frequenz gefordert wurde, muss weiterhin davon ausgegangen werden, dass der Speichertakt nur wenig höher liegt als der Takt der Abstastung. Die Anzahl der Eingänge skaliert folglich linear mit der Datenbreite des Speicherbus und limitiert diese stark.

- Alternativ bieten die meisten FPGAs dedizierten **internen Speicher**. Dieser ist in Blöcke unterteilt, die nahezu beliebig kombiniert werden können. Die maximale Speichertiefe ist vom eingesetzten Chip abhängig, bewegt sich aber zwischen einigen 10 KBit[24] bis hinzu etwa 50 MBit[25]. Abhängig von der Anzahl der Eingänge des Analyzers lassen sich diese

[24] z.B. Altera Cyclone [Alt11]

[25] z.B. Altera Stratix [Alt11]

Blöcke entweder massiv parallelisieren oder konkatenieren. Dabei werden keinerlei Pins benötigt.

- **Verteilter Speicher** [Xil11a] nutzt die eigentlich zur Implementierung der Logik gedachten *look-up-tables* der elementaren Blöcke des FPGAs als Speicherelemente. Dieser Ansatz eignet sich zwar, um kleinere Register zu realisieren, für welche die Verwendung der dedizierten Speicherblöcke nicht in Frage kommt, belegt jedoch immens viel Fläche im Chip, die der Anwendung damit nicht mehr zur Verfügung steht. Sollten die zuvor genannten Möglichkeiten jedoch nicht verfügbar sein, stellt dies einen Ausweg dar.

Jede der präsentierten Methoden hat ihren Anwendungsfall; keine sollte von vornherein ausgeschlossen werden. Da ein Schreibzugriff auf alle der o.g. Speicher sowohl die Adressierung der aktuellen Speicherzelle sowie das parallele Anlegen der Daten voraussetzt, kann ohne großen Hardwarebedarf eine Abstraktionsschicht eingepflegt werden, die eine einheitliche Schnittstelle zur Verfügung stellt. Das Framework *soll* daher nur ein Speicher-Interface exportieren und keine Technologie bevorzugen. Die Anbindung des Speichers bleibt dem Anwender überlassen.

4.3.2 Zugriffsmodell

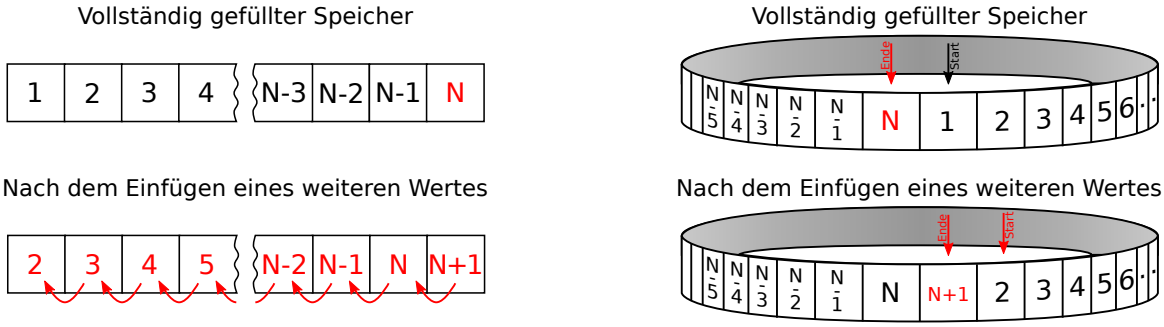
Anhand der Anforderungen wird nun ein Zugriffsmodell auf den Speicher erarbeitet. Dabei werden schreibender und lesender Zugriff zunächst getrennt behandelt.

Um den Datendurchsatz nicht einzuschränken und keine weiteren Caches zu benötigen, muss das Schreiben eines Datensatz in einem Zyklus des Samplingtaktes geschehen. Es liegt daher nahe, diesen Takt auch als Steuertakt für das Beschreiben des Speichers zu nutzen. Das Lesen der aufgezeichneten Daten ist weniger zeitkritisch. Zwar ist ein hoher Datendurchsatz erwünscht, jedoch sind die Auswirkungen einer Verzögerung weniger gravierend als beim Schreiben, bei dem es im *worst case* zum Datenverlust kommt. Zu beachten ist aber, dass das Lesen auch funktionieren soll, wenn der Abtasttakt ausbleibt.

Aus diesem Grund muss für das Lesen der Daten ein interner Takt verwendet werden. Der Speicher und dessen Verwaltung werden somit zur Schnittstelle zwischen zwei Taktbereichen (*clock domains*), wodurch eine Synchronisation aller relevanten Daten erforderlich wird. Als besondere Schwierigkeit stellt sich dabei heraus, dass keine Aussage darüber getroffen werden kann, welcher der Takte schneller läuft, bzw. ob der externe Takt überhaupt anliegt.

Die Option *Pretriggering* durchführen zu können hat große Auswirkungen auf das Speichermodell. Da bei dieser Art der Auslösung die Daten von Interesse sind, die *vor* dem Ereignis erfasst wurden, ist es bereits vor dem Event nötig zu samplen. Sobald die Speichergrenze N erreicht ist, müssen die ältesten Daten überschrieben werden. Auf diese Weise steht – bei genügend langer Aufzeichnung – zu jedem Zeitpunkt ein N Sätze umfassendes, lückenloses Protokoll der am Eingang anliegenden Daten zur Verfügung.

Abbildung 8a illustriert einen naiven Ansatz das o.g. Problem zu lösen. Hierbei wird das Alter eines Datensatzes direkt auf dessen Adresse abgebildet: Auf der ersten Position 0 liegt der älteste Wert, der neuste wird jeweils auf die oberste Adresse $(N-1)$ geschrieben. Ist der Speicher



(a) Linearer Speicher **(b)** Ringpuffer

Abbildung 8: Visueller Vergleich von linearem Speicher und einem Ringpuffer im FIFO Betrieb. Jeweils dargestellt ist der Zustand nach dem Einfügen auf der letzten freien Position, sowie nach dem ersten Überschreiben. Änderungen zum vorherigen Zustand sind rot hervorgehoben.

voll, werden sämtliche Daten an eine niedrigere Adresse verschoben, wobei das älteste Element verloren geht. Diese kommt jedoch nicht in Frage, da pro Takt nur auf ein Speicherelement zugegriffen werden kann, das Reorganisieren jedoch N Zugriffe benötigt.

Ein *Ring-Puffer* stellt die Lösung dieses Problems dar. Dieser simuliert einen zyklischen Adressenraum. Die Abbildungsvorschrift zur Berechnung der physikalischen Adresse des n-ten Datensatz lautet $f(n) = n \mod N$. Im konkreten Fall würde beim Speichern des $(N+1)$-ten Datensatzes auf die logische Adresse $n = N$ das Naheliegende geschehen: Durch die Kongruenz erfolgt der Zugriff tatsächlich auf die Hardwareadresse $f(N) = 0$. Der älteste Wert wird also überschrieben, alle anderen bleiben unverändert (siehe Abbildung 8b).

Löst der Trigger aus, muss abhängig vom eingestellten Zeitversatz die Aufzeichnung entweder sofort (im Falle von *Pretriggering*) oder nach maximal N Schritten (*Posttriggering*) gestoppt werden. Hierzu kann entweder ein Zähler, der bei jedem weiteren Zugriff dekrementiert wird, benutzt oder eine statische Stop-Adresse, bei deren Erreichen angehalten wird, verwendet werden.

Der Vorteil der Stop-Adresse liegt darin, dass sie nur einmal berechnet werden muss und nicht während jedes Taktes verändert wird. Wird der Puffer nebenläufig gefüllt und gelesen, kann trotz des Taktbereich-Übergangs durch das Nachführen der Stop-Adresse die effektive Aufzeichnungslänge erhöht werden. Dieses Feature *soll* jedoch nicht in der Grundversion implementiert werden.

Da das Berechnen der Stop-Adresse anhand der aktuellen Schreibadresse und einem Offset geschieht, bedarf eine datensatzgenaue Zeitverschiebung zwischen *Pre-* und *Posttriggering* nur wenig zusätzlicher Ressourcen und *soll* daher ermöglicht werden.

Komfortabel lässt sich der Übergang zwischen den Taktdomänen über sog. *dual-port RAM* lösen, das über zwei getrennte Schnittstellen verfügt, die auf die gleichen Daten abbildet. Da der Speicherzugriff aber unabhängig von der Technologie geschehen soll und besonders bei externem Speicher nicht immer zwei getrennte Kanäle verfügbar sind, kann dies nicht vorausgesetzt werden.

Jedoch kann ausgenutzt werden, dass aus dem einem Taktbereich nur geschrieben, und mit dem anderen nur gelesen wird, wobei zusätzlich das Lesen theoretisch beliebig verzögert werden darf. Daher *soll* das Framework neben getrennten Lese- und Schreib-Ports, die einen eigenen Takt, eigene Adresse-, Daten- sowie Steuerleitungen besitzen, auch Informationen darüber exportieren, ob gerade aufgezeichnet wird. Somit kann der Speichertreiber *single-* und *dual-port RAM* implementieren.

Für den Zugriff des Hosts auf den Speicher stehen mindestens zwei Möglichkeiten zur Verfügung: Die eine besteht darin, den Logic Analyzer als erweiterten *FIFO* anzusehen. Die Datensätze können bei diesem Modell nur in der Reihenfolge ausgelesen werden, in der sie auch in den Speicher gelangten. Hierzu wird eine Logik benötigt, die das sequentielle Lesen ermöglicht und nur auf gültige Speicherbereiche zugreift.

Alternativ kann dem Host direkter Zugriff auf den Speicher sowie auf Informationen gewährt werden, die benötigt werden um den Ringpuffer zu rekonstruieren. Dieser führt dann die Adressierung selbst durch. Das Auslagern dieser Funktionalität in die Software des Hosts ist erstrebenswert. Weiterhin kann ein Wert, der etwa bei der Übertragung über die externe Schnittstelle verloren ging, ohne zusätzlichen Zwischenspeicher beliebig oft gelesen werden. Die beiden zuletzt genannten Argumente sind dabei zugunsten des zweiten Ansatzes ausschlaggebend.

4.4 Kommunikations-Schnittstellen

4.4.1 Schnittstellentypen - Eine Übersicht

Die Schnittstelle zwischen dem Framework und dem Host dient zur Konfiguration diverser Einstellungen zur Laufzeit sowie der Übermittlung der erfassten Daten. Explizit gefordert wurde die Möglichkeit, verschiedene dieser Schnittstellen zu unterstützen. Um die benötigten Hardware-Ressourcen zu minimieren, *soll* die Auswahl des zu verwendenden Interface vor der Synthese geschehen. Im Folgenden werden nun die Eigenschaften potentieller Kommunikationskanäle beleuchtet, um daraus ein gemeinsames Zugriffskonzept zu erarbeiten.

- Die RS232-Schnittstelle, umgangssprachlich auch *serielle Schnittstelle* genannt, kommt in vielen eingebetteten Systemen zum Einsatz, da ihre Implementation im Vergleich zu moderneren Alternativen sehr simpel ist, und sie über keine High-Level-Protokolle verfügt. Die Datenübertragung findet seriell statt, wobei pro Rahmen 7 oder 8 Bit Nutzdaten üblich sind. Ein optionales Handshaking kann per Hardware oder Software gelöst werden, wobei im zweiten Fall die Sonderzeichen `XON` und `XOFF` genutzt werden, um die Kommunikation zu pausieren. Es obliegt der Anwendung dafür zu Sorge zu tragen, dass bei aktiviertem Software-Handshaking, diese Steuerzeichen nicht in den Nutzdaten vorkommen. Mittels Parity-Check wird eine simple Datenkonsistenzprüfung ermöglicht. Da es sich um eine asynchrone Übertragung handelt, können beide Partner eine Kommunikation initiieren.

- Die *JTag*-Schnittstelle wird in Kapitel 2.5 detailliert beschrieben. Sie eignet sich für dieses Projekt besonders, da FPGAs während der Entwicklung meist über die *JTag*-Schnittstelle

konfiguriert werden. Somit kann der Analyzer ohne zusätzliche Hardware betrieben werden. Es handelt sich ebenfalls um eine serielle Schnittstelle, die allerdings keine standardisierte Paketgröße erwartet. Die Kommunikation wird vom Master, z.B. dem PC, gesteuert. Clients können keine Kommunikation initiieren. Eine Datensicherung existiert nicht. Das Protokoll unterstützt durch verschiedene Register die semantische Trennung diverser Daten. Wird aber derselbe Port ebenfalls zum Konfigurieren des FPGAs verwendet, ist die Anzahl der Register im Allgemeinen stark beschränkt[26].

- Bei *Ethernet* handelt es sich um eine in Computernetzwerken verbreitete Schnittstelle. Nach dem ISO/OSI-Standard-Modell definiert *Ethernet* die beiden unteren Schichten. Die Übertragung findet paketorientiert statt, wobei pro Rahmen maximal 1 500 Byte Nutzdaten übertragen werden können. Die minimale Paketgröße beträgt 64 Byte. Die Schnittstelle eigenet sich folglich nur bedingt für kurze Mitteilungen. *Ethernet* kommt fast ausschließlich in Verbindung mit höheren Netzwerkschichten wie TCP/IP zum Einsatz, deren Handhabung in reiner Hardware sehr aufwendig ist. Es wird daher meist in Verbindung mit (Mikro-)Prozessoren verwendet.

- Mikroprozessoren stellen ebenfalls einen interessanten Kommunikationspartner dar. Dabei ist ein Einsatz als Vermittler zwischen PC und dem Framework genauso denkbar, wie ein Szenario, in dem der Prozessor die Steuerung selbst übernimmt. Von besonderem Interesse ist ein *Soft-Core*, also ein ebenfalls im FPGA realisierter Rechenkern, da dieser keine weiteren Pins belegt. Üblicherweise verfügen diese Kerne über einen Datenbus, der aus Adresse und Datenleitungen sowie einigen Handshake-Signalen besteht. Im Unterschied zu den o.g. Interface sind diese meist parallel ausgeführt.

4.4.2 Abstraktionsschicht

Alle vorgestellten Schnittstellen weisen zum Teil erhebliche Differenzen auf. Es ist kaum möglich ein effizientes Protokoll zu finden, dass die Eigenheiten aller vereint. Daher *soll* eine Abstraktionsschicht eingefügt werden. Diese *soll* ein kompaktes Interface aufweisen, das sich bei Anpassungen des Frameworks nach außen höchstens minimal ändert. Das individuelle Exportieren aller Einstellungen kommt deshalb nicht in Frage.

Die Verbindung zwischen dem Framework und dem Schnittstellentreiber *soll* aus diesem Grund über einen adressierbaren Datenbus geschehen. Es wird somit ein mit *ChipScope Pro* vergleichbarer Ansatz gewählt. Alle Eigenschaften werden in Form von Registern auf den Datenbus gelegt. Dabei *sollen read-only* Register, die z.B. Informationen zu Synthese-Einstellungen enthalten, *write-only* Bereiche, für Daten die nur gesetzt, nicht aber ausgelesen werden müssen, sowie Register mit wahlweisem Zugriff unterstützt werden.

Hierzu stehen eine Vielzahl an sog. Bustypen zur Verfügung, wobei der in Kapitel 2.4 beschriebene *WISHBONE Bus* die populärste freie Schnittstelle darstellt. Diese schreibt weder die

[26] typischerweise zwischen 2 (z.B. *Xilinx Spartan-3* [Xil11d]) bis 4 (etwa *Xilinx Virtex 6*)

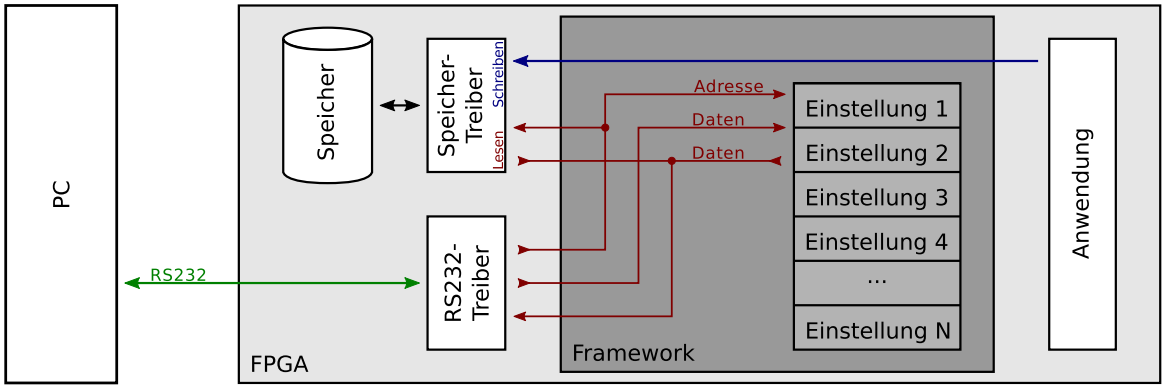


Abbildung 9: Schematische Darstellung des bisherigen Konzeptes. Exemplarisch ist eine *RS232* Schnittstelle und interner Speicher dargestellt. Datenfluss gleicher Farbe gehört zur gleichen Taktdomäne. Der *WISHBONE Bus* ist rot hervorgehoben.

Breite der Datensignale noch die des Adressbusses vor. Um eine möglichst einfache Integration mit einem vorhanden integrierten Prozessor(system) zu ermöglichen, *soll* sich das Framework an dessen Vorgaben anpassen. Register, deren Breite die der Datenleitungen übersteigt, müssen dafür automatisch über mehrere Adressen verteilt werden. Der Datenspeicher wird in den Adressbereich integriert.

Im Zuge dieser Arbeit werden Treiber für die *RS232*- und die *JTag*-Schnittstelle erstellt. Da für die erste Schnittstelle eine Vielzahl an Terminal-Anwendungen existiert, *soll* zu Entwicklungszwecken für diese ein menschenlesbares Protokoll definiert werden.

4.4.3 Integration von Erweiterungen

Das Spannungsfeld zwischen der geforderten Erweiterbarkeit und einem minimalem Hardwarebedarf tritt an dieser Stelle besonders zu Tage. Da die Grundidee für eine flexible Struktur darin besteht, möglichst alle Anpassungen auf das Einsatzfeld vor der Synthese durchzuführen, ist es wünschenswert diese Konfiguration an das steuernde System mitzuteilen, damit dieses die korrekten Treiber laden kann.

Maximale Flexibilität wird durch ein PlugIn-System möglich. Dabei müssen zwei Aspekte beachtet werden: Zum einen muss eine Schnittstelle für PlugIns auf *VHDL*-Ebene geschaffen werden, die eine Verbindung der Erweiterung mit dem Kommunikationsbus erlaubt. Zum anderen muss auf logischer Ebene eine Methode entwickelt werden, mit der das steuernde System automatisiert Informationen über die im Framework vorhandenen Komponenten sammeln kann.

Auf der Hardware-Ebene sind keine großen Probleme zu erwarten. Der *WISHBONE Bus* unterstützt implizit die *point-to-many* Topologie. So kann jedes PlugIn als *Slave* an den Bus angeschlossen werden. Auf *VHDL*-Ebene kann dann von jedem Modul gefordert werden, seinen Adressbedarf mittels einer Konstante bekannt zu machen, und seine Basisadresse über eine generische Konstante zu erhalten. Hierdurch lässt sich automatisiert eine Kette aufbauen. Die Adressberechnung erfolgt in diesem Szenario während der Elaboration und hat daher keine Auswirkung auf den Hardwarebedarf.

Die logische Ebene hingegen muss unter Verwendung von zusätzlicher Hardware implementiert werden. Für deren Entwurf können erweiterbare Bussysteme, etwa *PCI* und *USB* (um

zwei physisch sehr unterschiedliche Vertreter zu nennen), ebenso als Inspiration dienen, wie Container-Dateiformate aus beispielsweise dem Multimediabereich. Ein gebräuchlicher Ansatz, der mit der o.g. Modulkette kompatibel ist, besteht darin, den Adressbereich jedes PlugIns mit einem Header zu versehen.

Das Format des Headers muss dabei standardisiert sein und z.B. eine Identifikationsnummer vorschreiben, anhand derer der Treiber die Funktion und Konfigurations-Möglichkeiten des Moduls erkennt. Zusätzlich ist ein Zeiger, der auf das nächste Plugin zeigt, erforderlich. Dieser Pointer kann genutzt werden, wenn das steuernde Gerät das Modul nicht unterstützt und somit keine Informationen über seinen Adressbedarf hat.

Da jedoch viele Module des Analyzers teilweise nur über ein Bit (etwa um die entsprechende Funktion zu de-/aktivieren) an Laufzeiteinstellung verfügen, verursacht die skizzierte Methode einen erheblichen Overhead, der in Hardware gelöst werden muss.

Daher wurde ein weniger eleganter, jedoch pragmatischer Ansatz gewählt. Anstatt viele kleine Header zu verwenden *soll* ein zentraler Informationsbereich definiert werden, der Auskunft über diverse vor der Synthese getroffenen Konfigurationen der Basismodule gibt. Die vorhandenen Laufzeiteinstellungen *sollen* manuell in eine Reihenfolge gebracht werden. Kurze Werte werden in Multifunktionsregistern gesammelt und müssen vom Gerätetreiber bitweise manipuliert werden. Desweiteren *sollen* zwei Register geschaffen werden, die eine Hauptversionsnummer und eine Nebenversionsnummer speichern. Die Hauptversionsnummer gibt dabei Auskunft über das Adress-Schema aller Basisfunktionen, während die Nebenversionsnummer verwendet werden kann, um Erweiterungen bekannt zu geben. Natürlich lässt sich hiermit keine eindeutige Identifizierung vieler Module durchführen, jedoch ist auch zunächst kein hohes Aufkommen an neuen Erweiterungen zu erwarten.

4.5 Trigger

4.5.1 Eingänge

Bei der vorangegangenen Analyse verschiedener Produkte konnten zwei Anbindungtypen der Triggereingänge beobachtet werden. Die Mehrheit aller diskreten Analyzer verwendet kombinierte Daten- und Triggereingänge und bietet darüber hinaus meist noch wenige Triggereingänge zum Anschluss externer Auslöserlogik an. Dieses Konzept ist primär dem Umstand geschuldet, dass das Verbinden vieler Signale zwischen der Anwendung und einem diskreten Messgerät aufwendig ist.

Es ist aber keineswegs immer sinnvoll signifikante Ereignisse auf Basis der aufzuzeichnenden Daten zu identifizieren. Soll beispielsweise die Aktivität auf einem Datenbus nach Auftreten eines Interrupts protokolliert werden, muss der Trigger nur Zugriff auf die Interrupt-Leitung haben. Umgekehrt können auch Szenarien konstruiert werden, in denen deutlich mehr Triggereingänge benötigt werden als Daten aufgezeichnet werden sollen.

Für integrierte Logic Analyzer ist die Trennung der beiden Eingänge nicht mit den Nachteilen von diskreten Messgeräten verbunden. Für den Fall, dass an beiden die selben Daten anliegen, wird erwartet, dass diese strukturelle Redundanz während der Synthese erkannt und durch

nachträgliches Verschmelzen vermieden wird. Werden hingegen getrennte Bereiche überwacht, bietet dieses Design enormes Potential zur Optimierung des Hardwarebedarfs.

Dem Beispiel von *ChipScope Pro* folgend *soll* das Framework aus diesem Grund einen von den Daten unabhängigen Eingang erhalten, dessen Breite vor der Synthese konfiguriert werden kann. Dedizierte Triggereingänge, die an der internen Logik vorbei agieren, *sollen* nicht vorgesehen werden.

4.5.2 Triggertypen

Schwieriger fällt die Wahl der Triggertypen (siehe Kapitel 2.1.1 und 3 für einige Bespiele). Für dieses Projekt ist ein universeller Trigger gefordert, der möglichst viele Anwendungsfälle abdeckt. Daher *soll* das Framework mehrere Trigger gleichzeitig einbinden können, wobei zur Laufzeit konfigurierbar sein *soll*, welche Trigger aktiv sind. Erkennt mindestens einer der aktiven Trigger ein Ereignis, *soll* dies zum restlichen System propagiert werden.

Protokollsensitive Auslöser kommen nicht in Frage, weil sie sehr anwendungsspezifisch sind und zum Teil einen hohen *Logic Footprint* aufweisen. Sie müssen daher bei Bedarf nachgerüstet werden. Dies kann entweder durch Integration in das Framework geschehen, oder als ein extern angeschlossenes Modul implementiert werden. Im zweiten Fall wird das Trigger Event an einen der Triggereingänge des Frameworks gelegt. Da der *WISHBONE Bus-Master* nicht Teil des Frameworks ist, kann durch Ergänzen eines Multiplexers die vorhandene Infrastruktur genutzt werden, um das externe Modul ebenfalls zur Laufzeit zu konfigurieren.

Somit bleiben nur noch Low-Level-Trigger, die auf Werte oder Signalwechsel reagieren. Der erste Typ kann rein kombinatorisch gelöst werden, für den zweiten Typ muss ein Vergleich zwischen dem aktuellen und mindestens einem vorherigen Wert angestellt werden. Daher ist eine sequentielle Schaltung mit mindestens einem Speicherglied pro Eingang erforderlich. Die Detektierung der relevanten Wechsel geschieht jedoch auch hier kombinatorisch.

Alle in der Grundversion implementierten Trigger *sollen* zur Laufzeit konfiguriert werden können. Daher sind die Parameter der kombinatorischen Formeln, die für die Detektion der Ereignisse verwendet werden, zur Synthese nicht bekannt, wodurch sich die Situation ergibt, dass mit der programmierbaren Logik eines FPGAs, erneut programmierbare Logik implementiert wird. Dieses recht ineffiziente Modell lässt sich z.B. durch eine *partielle Rekonfiguration* des Chips platzsparender umsetzen. Weil diese aber im Allgemeinen nicht zur Verfügung steht, wird der Ansatz nicht weiter verfolgt.

Die Grundversion *soll* die beiden o.g. einfachen Triggertypen enthalten. Ihre Anzahl *soll* vor der Synthese ausgewählt werden können. Für jede Instanz, die auf Werte triggert, *soll* individuell eine Bitmaske an relevanten Signalen sowie deren Werte angeben werden können. Es *soll* ausgelöst werden, sobald alle selektierten Eingänge den geforderten Wert annehmen. Die aus einer Kombination verschiedener Trigger resultierende Struktur ähnelt daher der Matrix eines PLA. Für die Flankenerkennung *soll* für jedes Signal definiert werden können, ob auf eine steigende oder fallende Flanke reagiert wird, oder ob das Signal nicht relevant ist.

Durch dieses Modell können einfach einzelne Werte auf einem Adress- oder Datenbus erkannt werden. Zur Detektierung von Wertebereichen eignet sich das Verfahren aber nicht, weil im

worst case für jeden Wert eine Triggerinstanz vorgesehen werden muss. Weiterhin lassen sich Sequenzen nur durch zusätzliche Randbeschaltung erkennen. Dieser Kompromiss zwischen Funktion und Platzbedarf wurde jedoch bewusst eingegangen. Eine Anpassung des Triggerkonzeptes in späteren Versionen ist jedoch wahrscheinlich.

4.6 Variable Dateneingänge

Die Anzahl der Eingänge hat große Auswirkungen auf die Einsatzmöglichkeit des Analyzers sowie auf dessen Struktur. Wie in den Kapitel 2.1 und 3 gezeigt, bewegt sich diese je nach Anwendungsfall in einem Bereich von eins bis einigen tausend.

Um die Werte während eines Taktzyklus zu erfassen, muss der interne Datenbus die entsprechende Datenmenge verarbeiten können. Insbesondere betrifft dies den Speicherzugriff. Der Hardwarebedarf skaliert daher mindestens linear mit der Anzahl der Eingänge. Daher *soll* diese vor der Synthese bestimmt werden können.

Da gerade beim Protokollieren großer Datenmengen die Verwendung von externem Speicher wahrscheinlicher wird, dessen Breite aber gleichzeitig auch aufgrund der begrenzten Anzahl an FPGA-Pins limitiert ist, *soll* der Analyzer eine sequentielle Datenverarbeitung erlauben. Zusätzlich wird erwartet mit dieser Betriebsart den Hardwarebedarf des Systems zu reduzieren. Natürlich muss diese durch eine niedrigere Abtastungsgeschwindigkeit erkauft werden.

Zur Serialisierung des Datenstroms *sollen* die Eingänge in eine vor der Synthese definierte Anzahl an Bänken partitioniert werden. In jedem Taktzyklus des Samplingtaktes wird eine Bank bearbeitet. Zur Laufzeit *soll* es zusätzlich möglich sein, verschiedene Bänke zu aktivieren, bzw. von der Datenerfassung auszuschließen. Um die zeitliche Korrelation der Bänke garantieren zu können, *sollen* deren Werte zeitgleich erfasst werden, bevor diese seriell verarbeitet werden.

Die geforderten Features können mittels eines Schieberegisters realisiert werden, das die Daten mit jeder steigenden Flanke des Takes zur nächsten aktiven Bank verschiebt. Wie im folgenden Kapitel gezeigt werden wird, ist der Logikbedarf dieses Modules enorm.

Um eine sehr platzsparende Alternative zu demonstrieren, *soll* zusätzlich ein einfacher Multiplexer implementiert werden, der nur die Wahl einer Bank ermöglicht.

4.7 Mechanismen zur automatisierten Verifikation

Um die Grundfunktion der Komponenten des Projektes sicherzustellen, *sollen* an geeigneter Stelle *Unit Tests* der Architekturen durchgeführt werden. Die Testszenarien *sollen* dabei die Grundfunktionen überprüfen und Grenzwerttests einschließen. Die Testbenches müssen anhand von Verifikationsroutinen das Verhalten der untersuchten Komponente automatisch bewerten. Der Erfolg oder Misserfolg eines Tests *soll* dabei über einheitliche Statusmeldungen ausgegeben werden, sodass alle *Unit Tests* im *Batch*-Betrieb ausgeführt und mittels einfacher Skripts verarbeitet werden können.

Die Integretationstests sind primär auf der Hardware durchzuführen. Sie müssen dabei das korrekte Zusammenspiel einer Vielzahl von logischen Einheiten sicherstellen. Zu diesen gehören:

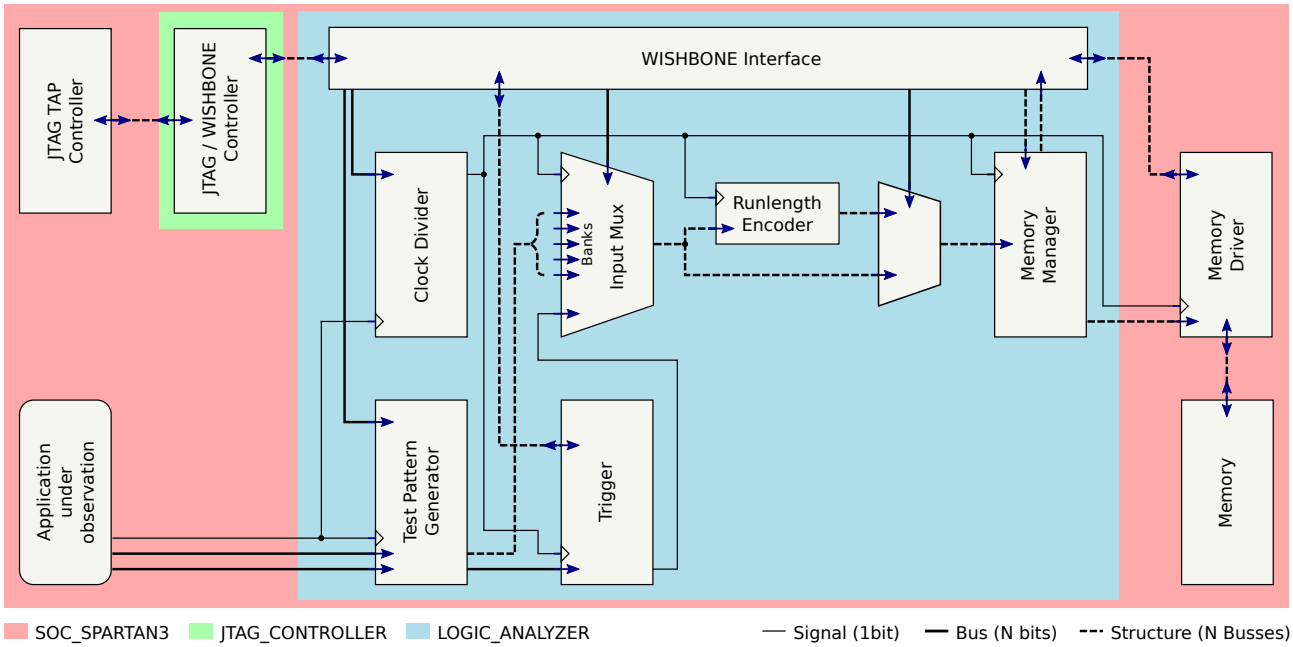


Abbildung 10: Struktur der Implementierung und schematische Darstellung des Datenflusses

Der Software-Treiber auf dem PC, die Kommunikation über die gewählte Schnittstelle, der Schnittstellentreiber im FPGA, Trigger, Eingangsmultiplexer sowie das Speichermanagement.

Hierzu *soll* vor der Synthese ein optionaler Funktionsgenerator eingebunden werden können. Dieser *soll* den Betrieb des Systems im Normalfall nicht beeinflussen. Durch das Auswählen eines Prüfmodus zur Laufzeit *sollen* jedoch die an den Daten- und Triggereingängen anliegenden Daten durch ein rekonstruierbares sequentielles Testmuster überschrieben werden. Da die Testmuster den Testbenches bekannt sind, kann auf diese Weise das gesamte System verifiziert werden. Die Pattern *sollen* so gewählt sein, dass die Funktion möglichst vieler Einheiten überprüft werden kann.

5 Implementierung

Dieses Kapitel beschreibt die Implementierung des Logic Analyzers auf Basis des bereits erarbeiteten Konzeptes. Dabei wird auf die Grundstruktur des Entwurfs eingegangen und wichtige Designentscheidungen anhand einer Auswahl von zentralen Funktionen beleuchtet. Für kleinere Einheiten, die bereits durch *VHDL* vollständig beschrieben sind, wird auf den ausführlich kommentierten Code verwiesen.

Das Kapitel teilt sich in drei Bereiche. Der erste Teil gibt eine strukturelle Übersicht des Projekts auf Hardwarebeschreibungsebene und beleuchtet globale Konventionen. Hierauf folgten eine Beschreibung des Frameworks und Erläuterungen der Schnittstellentreiber.

5.1 Grundlegende Designentscheidungen

Zur Implementierung des Projektes werden neben *VHDL* auch die Programmiersprachen *Python* und *C* eingesetzt, wobei diese Sprachen nur zur Programmierung der Steuerungssoftware dienen. Da – wie in der Konzeption gefordert – sämtliche Software frei sein und öffentlich gemacht werden soll, sind Quellcodes und Dokumentation in Englisch gehalten. Für *Python*

wird weitgehend der anerkannte *Code-Style-Guide* [RW01] und ein objektorientiertes Design verwendet. Ähnlich populäre Konventionen konnten für VHDL nicht gefunden werden, so wird für dieses Projekt ein eigenes Regelwerk angewandt, das dem Quellcode beiliegt und Richtlinien zur Benennung, zur Kommentierung und bzgl. der Groß- und Kleinschreibung enthält, und dazu anregt auch bei Erweiterungen durch Dritte ein konsistentes Codebild zu bewahren.

In der Konzeption werden bereits wichtige Entscheidungen zur Struktur des Projektes getroffen. So besteht die Forderung nach einer strikten Kapselung von Funktionseinheiten, insbesondere zwischen Treibermodulen und technologieunabhängigen Beschreibungen. Diese Separation wird mit der Verwendung von getrennten Bibliotheken auf die Implementierung übertragen. Es kommen drei portable Libraries zum Einsatz: `LOGIC_ANALYZER` enthält das Framework, `RS232_CONTROLLER` und `JTAG_CONTROLLER` beinhalten die Schnittstellentreiber. Jede Bibliothek verfügt dabei über ein Paket, das alle zur Instanziierung der öffentlichen Entitäten nötigen Komponentenbeschreibungen sowie Datentypen definiert.

Weiterhin beinhaltet das Referenzdesign die Library `SOC_SPARTAN3`, welche das Top-Modul zur Synthese für einen *Xilinx Spartan-3* FPGA sowie Treiber für den Low-Level-Speicher- und *JTag*-Zugriff enthält (siehe Abbildung 10). Näheres hierzu kann dem Kapitel *Inbetriebnahme* (Appendix A) entnommen werden.

Alle Parameter, die bei der Synthese erforderlich sind, werden zentral in dem Paket `LOGIC_ANALYZER.CONFIG_PKG` als öffentliche Konstanten definiert. Das Paket muss vom Anwender beim Einbinden des Logic Analyzers zur Verfügung gestellt werden. Eine Beispielkonfiguration liegt dem Referenzdesign bei.

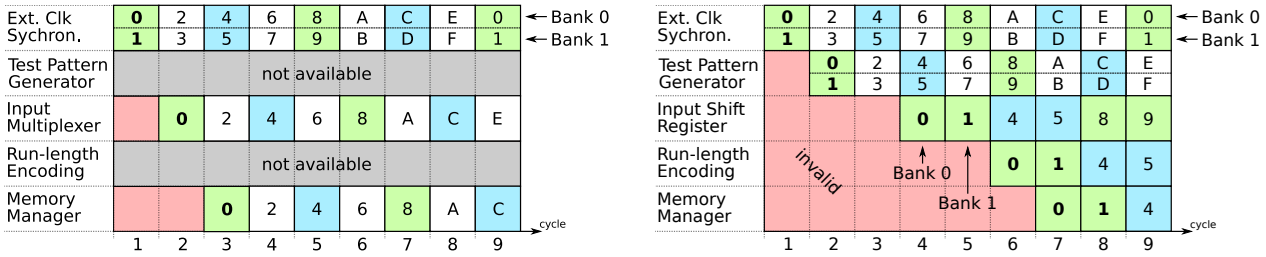
In Anlehnung an die *Two Process* [Gai] Designmethodik werden verwandte Ein- und Ausgänge von Entitäten durch Verbund-Datentypen (`record`-Typ) zusammengefasst. Hierbei werden besonders die Endpunkte der Signale innerhalb des Projekts beachtet, sodass im Idealfall die Vernetzung zweier Entitäten in *VHDL* durch ein einziges zusammengesetztes Signal implementiert werden kann.

Das Design unterstreicht somit im Quellcode den Datenfluss und ermöglicht es, auch durch eine nachträgliche Anpassung der Typ-Definition zusätzliche Informationen austauschen zu können, ohne die Implementierung der transportierenden Zwischenschichten ändern zu müssen.

Eine vollständige Anwendung der *Two Process Method* erscheint für dieses Projekt jedoch nicht sinnvoll, da es eine Vielzahl von kompakten Architekturen gibt, bei denen die Fixierung auf einen Implementierungsansatz die Ausdrucksmöglichkeit von *VHDL* unnötig einschränkt und in einigen Fällen zu schlechter lesbaren Beschreibungen führt.

5.2 Die Bibliothek `LOGIC_ANALYZER`

Die Bibliothek `LOGIC_ANALYZER` vereinigt die Kernfunktionen des Analyzers. Hierzu zählen vor allem die technologieunabhängigen Funktionen, die zur Datenaufbereitung und -verarbeitung benötigt werden. Die Library besitzt nur eine öffentliche Entität (`LOGIC_ANALYZER_CORE`), welche die in der Konfiguration definierten Funktionen instanziiert und vernetzt. Die Schnittstelle der Entität verwendet die bereits angesprochenen Verbund-Datentypen. Dies fördert die



(a) Kurze Pipeline **(b)** Lange Pipeline

Abbildung 11: Schematische Darstellung der Sampling-Pipeline in zwei Konfigurationen.
(a) repräsentiert eine kurze Pipeline, in der sowohl der Testmustergenerator wie auch Kompressionsstufe fehlen. Es werden insgesamt 2 Registerbänke zum Verzögern der Daten benötigt.
(b) stellt die komplexeste Variante dar. Es wird ein Input Shift Register (2 Bänke aktiv) und Run-Length Encoding verwendet. Weiterhin ist der Testmustergenerator vorhanden, aber deaktiviert.

Lesbarkeit und erhöht die Portabilität, da sich die Komponentendefintion bei Anpassung der Konfiguration nur implizit ändert.

Sämtliche Parameter, auf die zur Laufzeit zugegriffen werden kann, werden durch die Kern-Entität verwaltet und an den *WISHBONE Bus* angebunden. Hierdurch lässt sich das Adress-Schema zentral verwalten und sprachlich unkompliziert mehrere Einstellungen zu einem Register zusammenfassen (vgl. Kapitel 5.2.7). Der Trigger nimmt diesbezüglich eine Sonderstellung ein, da dieser über eine eigene Busanbindung verfügt.

5.2.1 Sampling-Pipeline

Konfigurationsabhängig besteht die Datenverarbeitung vom Eingang des Logic Analyzers bis hin zum Speicher aus unterschiedlich vielen und komplexen Schritten. Um diese einzelnen Stufen zu synchronisieren und unabhängig von der Ausstattung und den Einstellungen des Systems eine konstant hohe Abtastrate zu ermöglichen, werden die einzelnen Funktionsblöcke zu einer Pipeline zusammengefasst. Die einzelnen Module arbeiten also parallel auf unterschiedlichen Daten. Je näher sich eine Einheit am Beginn der Pipeline befindet, desto neuere Daten analysiert sie. Die Ergebnisse werden dann synchron zu einem gemeinsamen Takt an die jeweils nächste Stufe übergeben.

Zwar erfordert eine Pipeline zwischen jeder Stufe Speicherglieder, die durch eine Vielzahl an *Flip-Flops* ($\geq$ Datenbreite $\times$ Pipelinestufen) realisiert werden müssen, jedoch enthält jede Logikzelle des FPGAs mindestens ein Zwischenspeicher, der in den meisten Fällen sonst ungenutzt bliebe. Im konkreten Fall bewirkte das experimentelle Verschmelzen von Stufen, eine Verlängerung des kritischen Pfades, die sich wiederum negativ auf die Arbeitsgeschwindigkeit des Gesamtsystems ausübte. Der Hardwarebedarf wurde hingegen nicht signifikant reduziert.

Die Timing-Diagramme in Abbildung 11 demonstrieren anhand zweier Beispiele verschieden lange Pipelines und führen dabei auch die einzelnen Module auf. Deren Funktion sowie Architektur wird im Folgenden genauer diskutiert.

5.2.2 Taktvorteiler `CLOCK_DIVIDER`

Der Taktvorteiler versorgt alle Pipeline-Module ab der Eingangsstufe mit einem gemeinsamen Takt. Hierzu wird die Frequenz des Quelltaktes durch eine Ganzzahl geteilt. Das Verhältnis kann zur Laufzeit mittels des Parameters `CLK_THRESHOLD` eingestellt werden. Dabei gilt das Frequenzverhältnis $f_{\text{out}} = \frac{f_{\text{in}}}{\texttt{CLK\_THRESHOLD} + 1}$. Folglich ändert der Standardwert 0 die Samplingrate nicht.

Die Nutzung interner *Clock Manager* (etwa *Xilinx Spartan-3 DCM* [Xil11d]) wird ausgeschlossen, da sich diese nicht technologieunabhängig instanziieren lassen und darüber hinaus teils nicht zur Laufzeit konfiguriert werden können. Daher ist die Frequenzteilung mittels eines Counters, der zyklisch bis `CLK_THRESHOLD` zählt, realisiert. Bei dessen Überlauf wird der Ausgang für eine Taktperiode invertiert.

Weil hierdurch die Frequenz in jedem Fall um mindestens den Faktor zwei geteilt wird, muss zur Weiterleitung der Originalfrequenz ein kombinatorischer Multiplexer verwendet werden, der zwischen dem externen und dem generierten Takt umschaltet. Dies erzeugt wiederum einen Phasenversatz des Samplingtaktes zum externen Signal. Um trotzdem sicherzustellen, dass die Daten während der steigenden Flanke des externen Taktes erfasst werden, wird der Sampling-Pipeline ein Speicherglied vorgeschaltet, das synchron zum externen Takt arbeitet. Dieses Modul ist direkt in der Kern-Entität realisiert und in Abbildung 11 als *Ext. Clk. Synchron.* bezeichnet.

Es lassen sich einige Szenarien konstruieren, in denen sich die Phasenverschiebung des Taktes mit der Latenz der Eingangslogik aufheben sollte und somit das Speicherglied nicht benötigt wird. Dies ist jedoch stark vom endgültigen *Routing* im FPGA abhängig und somit im Allgemeinen spekulativ.

Die Implementierung behandelt die Fälle daher nicht automatisiert. Der Anwender kann das Eingangs-Speicherglied jedoch per Konfiguration entfernen, sollte dies im konkreten Fall durch *Timing Constraints* in Verbindung mit einer *Static Timing Analysis* verifiziert werden.

Wird der Parameter `CLK_THRESHOLD` verringert, kann es dazu kommen, dass der erste neue Takt zu lang ausfällt. Es ist möglich den Effekt durch das Zwischenspeichern des Threshold-Wertes zu vermeiden, jedoch erscheint es ohnehin nicht sinnvoll, den Takt während einer Aufzeichnung zu ändern. Daher wird zur Reduzierung der benötigten Fläche hierauf verzichtet und empfohlen nach dem Ändern des Taktes die Aufzeichnung neu zu starten.

Die Breite des Zählers lässt sich vor der Synthese definieren. Darüber hinaus kann vollständig auf den Taktvorteiler verzichtet werden.

5.2.3 Testmustergenerator `TEST_PATTERN_GENERATOR`

Der Testmustergenerator dient ausschließlich der Verifikation des Projekts und lässt sich für den Produktiveinsatz vor der Synthese deaktivieren. Ist der Generator aktiv, werden die Werte des Daten- und des Trigger-Eingangs mit Testmustern überschrieben. Jedes Testmuster enthält einen Zähler, der abhängig vom Testmodus unterschiedlich schnell, jedoch immer synchron zum externen Takt erhöht wird.

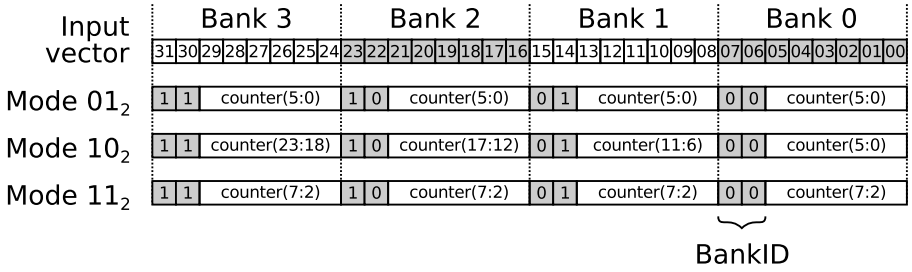


Abbildung 12: Betriebsmodi des Testmustergenerators

Wenn der Dateneingang des Analyzers in mehrere Bänke unterteilt ist, passt sich das Testmuster dieser Struktur an. Den oberen Bits einer Bank wird in diesem Fall ein konstanter Wert, der die Banknummer kodiert, zugewiesen. Somit kann jeder erfasste Datensatz einer Bank zugeordnet werden, wodurch sich etwa das *Input Shift Register* (siehe Kapitel 5.2.4), aber auch der Dekompressionsalgorithmus auf dem PC automatisiert überprüfen lässt.

Es gibt insgesamt vier Betriebsmodi, die durch zwei Bits kodiert werden (siehe Abbildung 12):

- Modus **00** ist der Standardwert und lässt die Daten unverändert. Der Generator ist also inaktiv.
- Modus **01** legt an alle Bänke den selben Zählerwert an und inkrementiert diesen mit jeder steigenden Taktflanke. Dieser Modus eignet sich daher als Stresstest, da in jedem Zyklus auf allen Bänken Änderungen auftreten.
- Modus **10** interpretiert den Datenbereich aller Bänke als einen großen Zähler. Der Wert der untersten Bank wird somit in jedem Taktzyklus erhöht. Läuft dieser über, wird der Wert der nächsten Bank inkrementiert. Mit höheren Bänken wird analog verfahren. Weil Änderungen höherer Bänke seltener erfasst werden müssen, eignet sich dieses Muster, um eine Bank-basierte Kompression zu testen.
- Modus **11** arbeitet wie Modus 00 mit dem Unterschied, dass der Wert in jedem vierten Takt erhöht wird. Hierdurch können einfachere Kompressionsalgorithmen getestet werden.

Neben dem synchronisierenden Speicherglied zu Beginn der Pipeline, ist der Generator das einzige Modul, das mit dem externen Takt betrieben wird. Daher kann über den Abstand zweier Zählerwerte in der Aufzeichnung die Funktion des Taktteilers überprüft werden.

5.2.4 Eingangsstufe

Die Eingangsstufe befindet sich innerhalb der Pipeline direkt nach dem Testmustergenerator. Ihre Funktion ist es, aus allen Eingangsbänken die aktiven auszuwählen und sie in der korrekten Reihenfolge in die Pipeline einzufügen, sowie Änderungen der relevanten Daten zu ermitteln. Die Selektion wird daher nur benötigt, wenn mehr als eine Bank konfiguriert ist. Anderenfalls wird die Struktur bereits zur Synthese entfernt.

Es wurden zwei unterschiedliche Implementierungen für dieses Modul erstellt. Da diese aber unterschiedliche Einstellungen zur Laufzeit benötigen, können nicht zwei Architekturen zur selben Entität angeben werden. Vielmehr existieren zwei unabhängige Einheiten, von denen eine vor der Synthese durch eine Konstante im Konfigurationspaket ausgewählt werden muss.

- Die Entität `INPUT_MULTIPLEXER` ist im wesentlichen ein einfacher Multiplexer, der die Wahl einer aktiven Bank unterstützt. Zur Selektion wird ein Register verwendet, das den Index der ausgewählten Bank speichert. Es ist nicht vorgesehen (jedoch trotzdem möglich), den Wert während einer Aufzeichnung zu ändern.

  Weiterhin überwacht das Modul die ausgewählten Daten und signalisiert der nächsten Pipelinestufe, ob es seit dem letzten Takt zu einer Änderung kam. Diese Information wird beispielsweise von der *Lauflängenkompression* verwendet. Findet die Synthese ohne Kompressionseinheit statt, wird die Vergleichslogik durch Optimierungen automatisch entfernt.

- Das Modul `INPUT_SHIFT_REGISTER` erlaubt im Vergleich zur o.g. Variante die Auswahl mehrerer Bänke. Die Kodierung geschieht über eine Bitmaske, in der jedes Bit genau einer Bank entspricht. Wie der Name bereits andeutet, ist die zentrale Komponente ein Schieberegister, das zu Beginn eines *shift cycles* alle am Eingang anliegenden Daten parallel lädt. Mit jedem Abtasttakt wird auf logischer Ebene der Inhalt des Registers genau so weit geschoben, dass die nächste eingeschaltete Bank am Ausgang des Registers anliegt. Inaktive Bänke werden dabei übersprungen.

  Die konkrete Implementierung nutzt nur für die Steuerlogik ein Schieberegister, um die Bitmaske zur Auswahl der aktiven Bänke zu verwalten. Die Auswahl der Daten ist jedoch effizienter über ein einfaches Speicherglied mit nachgeschaltetem Multiplexer zu lösen.

  Auch dieses Modul detektiert Änderungen an den Werten der aktiven Bänke und leitet diese Information an die nachfolgenden Stufen weiter. Das Signal bleibt dabei während eines Zyklus konstant.

Mit dem zweiten Ansatz lässt sich folglich das Verhalten des `INPUT_MULTIPLEXERS` simulieren. Jedoch ist der Hardwarebedarf des Schieberegisters im Vergleich mehr als doppelt so hoch. So muss für jedes Eingangsbit eine Logikzelle als Speicher verwendet werden, ohne zusätzliche Funktionen zu implementieren. Weiterhin fällt das Steuerwerk deutlich komplexer aus.

5.2.5 Kompressionsstufe

Die Kompressionsstufe ist eine optionale Schicht innerhalb der Sampling-Pipeline, welche direkt auf die Eingangsstufe folgt und es erlaubt, während der Aufzeichnung den Datenstrom zu komprimieren. Dies kann verlustbehaftet (vgl. *Xilinx ChipScope Pro*, Kapitel 3.1) oder verlustfrei mittels generischer Kompressionsalgorithmen geschehen.

Dabei können durch einfache Anpassung des Codes beliebig viele Filtermodule angegeben und zur Laufzeit ausgewählt werden. Insbesondere ist es möglich, sämtliche Filter zu deaktivieren.

Für Kompressions- und Filtermodule existiert eine einheitliche Infrastruktur zur eindeutigen Kennzeichnung von Nutzdaten und Zusatzinformationen. Hierzu enthält der Ausgang der Pipelinestufe neben den Daten zusätzlich ein Signal `meta`, das per Definition aktiv wird, wenn es sich bei den Daten um Metainformationen handelt.

Es ist den folgenden Einheiten überlassen, wie diese Information abgelegt wird, bzw. ob sie ignoriert wird, wenn vor der Synthese alle Filter deaktiviert wurden. Im Referenzdesign wird die Speicherbreite um ein Bit erhöht, und auf diese Weise zu jedem Eintrag explizit abgespeichert, ob es sich bei diesem um Metadaten handelt. Dabei kann ausgenutzt werden, dass die internen *RAM*-Blöcke des *Xilinx Spartan-3* FPGA pro 8 Datenbits ein Bit zum Speichern der Parität bereithalten. Da die Implementierung keine Prüfsummen berechnet, bliebe dieser Bereich ungenutzt. Der Hardwarebedarf ist daher sehr gering und beschränkt sich auf zwei Speicherglieder zwischen den folgenden Pipelinestufen.

Kann die Speicherbandbreite nicht erweitert werden, steht ein alternativer Ansatz zur Verfügung, der häufig in Softwarelösungen Anwendung findet. Grundsätzlich muss davon ausgegangen werden, dass die Daten jeden darstellbaren Wert annehmen können, also kein Wert existiert, der niemals in den Nutzdaten vorkommt. Daher können sog. *Escape Sequenzen* definiert werden, in denen ein Symbol A den Beginn von Zusatzinformationen markiert.

Darüber hinaus ist es notwendig, eine Folge von Zeichen A' (z.B. $A' = AA$) zu bestimmen, die wiederum als Nutzdaten-Wert A interpretiert wird. Es kann jedoch zu Problemen kommen, wenn z.B. eine Sequenz, die nur aus Nutzdaten A besteht, geschrieben werden soll. Naiv wird jedes A mittels A' im Speicher repräsentiert. Da aber $|A'| > |A|$, kann A' nicht mit der Geschwindigkeit geschrieben werden, die für normale Zeichen erwartet wird. Dies kann entweder durch Senken der Abtastfrequenz oder mit zusätzlichen Verwaltungsstrukturen gelöst werden.

Exemplarisch für ein Modul der Kompressionsstufe wird die *Lauflängenkodierung* (meist als *Run-Length Encoding*, kurz RLE, bezeichnet) betrachtet. Es handelt sich hierbei um einen verlustfreien Kompressionsalgorithmus, der mit wenig Hardwarebedarf umgesetzt werden kann. Ziel ist zu vermeiden, dass unveränderte Signale mehrfach in den Speicher geschrieben werden. Statt diese Signale redundant abzulegen, wird der Wert und die Dauer seiner Gültigkeit gespeichert.

In der Implementierung wird die Taktfrequenz geteilt durch die Anzahl der aktiven Bänke als Zeitbasis verwendet. Kommt ein *Input Shift Register* zum Einsatz, entspricht dies also einem Schiebezyklus, dessen Beginn über ein dediziertes Signal angezeigt wird. In jedem anderen Fall sind Abtasttakt und Zeitbasis identisch. Weiter werden alle aktiven Bänke als ein Wert interpretiert. Ändert sich nur ein Element der Gruppe, wird der gesamte Zyklus neu geschrieben. Offensichtlich arbeitet dieser Ansatz in einigen Fällen suboptimal (etwa wenn sich immer nur eine Bank verändert während der Rest konstant bliebt). Er reduziert jedoch deutlich den Hardwarebedarf und erzeugt eine einfachere Datenstruktur.

Die Gültigkeitsdauer eines Datums wird zu Beginn eines Abtastzyklus erhöht, wenn die Eingangsstufe keine Änderung der Daten signalisiert. Um eine möglichst einfache Datenstruktur zu schaffen, entspricht die Breite dieses Registers der Bankbreite. Es handelt sich somit um die größte Zahl, die in einem Abtasttakt in den Speicher geschrieben werden kann. Da die zum Inkrementieren des Zählers benötigte Zeit mit zunehmender Bankbreite wächst und somit die maximale Samplingrate negativ beeinflussen kann, lässt sich die Registerbreite manuell begrenzen.

Wird eine Änderung festgestellt oder ist ein Zählerüberlauf eminent, wird der Wert des Counters gefolgt von dem neuen Datensatz in den Speicher kopiert. Auf das Speichern der Gültigkeitsdauer wird verzichtet, wenn der Zähler den Wert 1 hat, die Daten also nicht für mindestens zwei Zyklen stabil waren.

Da die Entscheidung, ob der Zähler geschrieben werden muss, erst in dem Takt getroffen werden kann, in dem auch die neuen Daten eintreffen, müssen diese eventuell zunächst gepuffert werden. Der Zwischenspeicher ist als Schieberegister mit zwei Eingängen, die parallel geladen werden können, implementiert. Da Metainformationen nur dann geschrieben werden müssen, wenn zuvor das Speichern von redundanten Nutzdaten verhindert wurde, kann der ggf. belegte Puffer in diesem Zeitraum geleert werden. Ein Zwischenspeicher für zwei Elemente ist daher ausreichend. Die Logik des Zwischenspeichers wurde als eigene Pipelinestufe ausgeführt. Daten benötigen daher – wie in Abbildung 11 dargestellt – zwischen zwei und drei Taktperioden um das RLE Modul zu queren.

5.2.6 Speicherverwaltung

Die Speicherverwaltung stellt die letzte Stufe der Sampling-Pipeline innerhalb des Frameworks dar. Neben den eigentlichen Daten und dem bereits diskutierten Meta-Bit, besteht die Eingabe des Moduls aus dem Zustand der Triggerlogik sowie einem Signal, das anzeigt, ob die anliegenden Daten gespeichert werden sollen. Anhand dieser Informationen steuert die Speicherverwaltung den Schreibzugriff auf die in der Konzeption beschriebene Speicher-Abstraktionsschicht. Die hierfür nötige Logik wird durch die Entität `MEMORY_MANAGER` beschrieben.

Diese befindet sich zu jedem Zeitpunkt in einem der drei Zustände `EMPTY`, `CAPTURING` und `FULL`, wobei nach einem Reset der erste Zustand angenommen wird. In diesem Betriebsmodus verhält sich das Modul wie der in Kapitel 4.3.2 beschriebene Ringpuffer: Mit jedem neuen Datensatz wird die Speicheradresse inkrementiert. Beim Überschreiten der höchsten Adresse wird erneut bei der untersten begonnen. Dabei wird der erste Überlauf festgehalten, um entscheiden zu können, ob die Werte oberhalb der aktuellen Schreibadresse gültig sind.

Ein *Trigger Event* lässt die Speicherverwaltung in den Aufzeichnungszustand wechseln. Hierzu wird zunächst der Laufzeitparameter `BASE_ADDRESS_OFFSET` von der aktuellen Adresse subtrahiert und als Basisadresse definiert. Durch diesen Versatz lässt sich zwischen *Pre-* und *Posttriggering* variieren. Wenn weniger Daten aufgezeichnet wurden als für das *Pretriggering* gefordert, wird die Basisadresse auf den ältesten Datensatz umgeschrieben und der geschützte Rahmen somit verkürzt.

Es ist alternativ auch denkbar, das Triggerereignis solange zu ignorieren, bis der Rahmen für das *Pretriggering* ausreichend gefüllt ist. Hierbei handelt es sich offensichtlich um eine wichtige Designentscheidung, deren Für und Wider primär vom Anwendungsfall abhängen. Es konnte kein allgemeines Argument zugunsten einer Methode gefunden werden. Weiter ist der Hardwarebedarf identisch. Der Wechsel zwischen beiden Verhaltensmodellen lässt sich durch eine fast triviale Anpassung der Architektur bewerkstelligen.

Zusätzlich zur Basisadresse wird beim Übergang in den Zustand `CAPTURING` auch die derzeit gültige Schreibadresse gespeichert. Diese sog. `TRIGGER_ADDRESS` lässt sich über den

WISHBONE Bus abfragen und wird von der Steuerungssoftware verwendet, um den Datenstrom zu rekonstruieren. Da ein *Trigger Event* bis zum Start des nächsten Schiebezyklus verzögert wird, zeigt die Triggeradresse folglich immer auf den Beginn eines Abtastzyklus. Hierdurch lassen sich beispielsweise beim Einsatz des *Input Shift Registers* die Datensätze den korrekten Bänken zuordnen.

Die Speicherverwaltung zeichnet solange weiter auf, bis die Schreibadresse der Basisadresse entspricht, und wechselt dann in den Zustand `FULL`. Dieser Betriebsmodus deaktiviert das Modul und verhindert insbesondere weitere Schreibzugriffe. Der Modus lässt sich nur durch ein Reset-Signal wieder verlassen. Der Reset muss dabei die gesamte Pipeline passieren und synchronisiert damit alle Module. Dieser Ansatz ist möglich, da alle Register, die zur Laufzeit über den *WISHBONE Bus* gesetzt werden können, außerhalb der Pipeline verwaltet werden, und somit auch nach dem Reset erhalten bleiben.

5.2.7 *WISHBONE Bus*

Wie bereits diskutiert, ist der Logic Analyzer intern über den *WISHBONE Bus* angebunden. Das durch die Entität `LOGIC_ANALYZER_CORE` repräsentierte Framework implementiert dabei logisch ein Bus-Multiplexer, der die Anbindung von drei *Slaves* erlaubt und die Schnittstellentreiber mit dem Trigger, dem Speichertreiber sowie den Laufzeit-Parametern des Analyzers vernetzt. Der Multiplexer wird mittels der in den Grundlagen eingeführten *Data Flow Interconnection*-Topologie umgesetzt.

Die Selektierung des aktiven Slaves erfolgt eindeutig über die am Bus anliegende Adresse. Jedem Teilnehmer ist ein zusammenhängender Adressbereich zugeordnet, wobei sich Framework und Trigger die untere Hälfte des Adressraums teilen und die obere Hälfte dem Speichertreiber zugeordnet wird. Daher ist die Adressbereite des Busses ein Bit höher als die des Speichers.

Das Adress-Schema wird automatisch zur Synthese generiert und kann dem Syntheseprotokoll entnommen werden. Die ersten 16 Adressen sind statisch jeweils 8 Bit-breiten Registern zugewiesen, anhand derer die Steuersoftware sowohl die Ausstattung des Analyzers ableiten als auch das dynamische Adress-Schema rekonstruieren kann.

Es gibt dabei zwei Ursachen für Adressverschiebungen. Zum einen werden Parameter von Modulen, die bereits vor der Synthese deaktiviert wurden, aus dem Adressraum entfernt. Zum anderen wurde in der Konzeption gefordert, dass die Datenbreite des Busses flexibel festgelegt werden soll, damit das Framework einfach in eine bereits vorhandene Infrastruktur integriert werden kann. Es ist daher notwendig, große Register auf die Granularität des Busses anzupassen und sie über mehrere Adressen zu verteilen. Als minimale Datenbreite werden jedoch 8 Bit gefordert.

Der *WISHBONE Bus Slave* ist algorithmisch implementiert. Dies wird durch eine Reihe an Prozeduren und Funktionen des `LOGIC_ANALYZER.WISHBONE` Paketes unterstützt, die z.B. Bitvektoren, Basisadressen und die Bussignale erhalten und die vollständige Buslogik inklusive der Verteilung von Registern auf mehrere Adressen kapseln.

Die Basisadressen von Registern werden zentral in einem internen Paket berechnet und mittels Konstanten allen Module des Frameworks zur Verfügung gestellt. Diese werden auch von den Testbenches intensiv genutzt.

Je nach Konfiguration existieren zwei Register, die sich gegenüber anderen Registern abweichend verhalten. Immer vorhanden ist das Statusregister auf der Adresse $0x10$, dessen obere zwei Bits genutzt werden können, um einen Reset auszulösen. Dazu wird zwischen einem *Soft Reset*, der nur die Pipeline zurücksetzt, und einem *Cold Reset*, der zusätzlich alle Parameter auf die Standardwerte setzt, unterschieden. Der erste Typ sollte dabei nach dem Ändern von Parametern genutzt werden, um sicherzustellen, dass die erfassten Daten konsistent mit den neuen Einstellungen aufgezeichnet wurden. Wie bereits ausgeführt, muss die Funktion desweiteren benutzt werden, um die Speicherverwaltung aus dem Zustand `FULL` zu holen. Der Wert der Bits wird in jedem Zyklus auf 0 gesetzt. Ein Reset muss daher nicht explizit zurückgenommen werden.

Das zweite atypische Register ist optional und kann genutzt werden, um die am Dateneingang des Logic Analyzers anliegenden Werte zu lesen. Die Erfassung geschieht synchron zum Systemtakt und funktioniert daher auch, wenn kein Abtasttakt vorhanden ist (vgl. Kapitel 4.1, *asynchrone Eingänge*). Weiterhin arbeitet dieses Register vollständig an der Sampling-Pipeline, insbesondere an der Eingangsstufe, vorbei und deckt daher nicht nur die gerade aktive Bank ab. Da sich das Register im Allgemeinen über mehrere Adressen verteilt, werden die Daten gelatcht, wenn die Basisadresse am Bus anliegt. Eine Lesevorgang sollte daher an der untersten Adresse beginnen, um die zeitliche Korrelation aller Teilwerte zu garantieren.

5.2.8 Trigger

Die Triggerlogik agiert weitgehend losgelöst vom Rest der Kernkomponenten. Diese Abweichung vom bisherigen Vorgehen erklärt sich dadurch, dass – wie in Kapitel 4.5 diskutiert – davon auszugehend ist, dass der Trigger für einige Anwendungsfälle angepasst werden muss. Es erscheint daher sinnvoll, alle Komponenten inklusive der Busanbindung zu kapseln, sodass ein neues Triggermodell durch Austauschen der bisherigen Architektur in den Entwurf konfiguriert werden kann. Da hierfür u.a. auf die zuvor beschriebenen Hilfsfunktionen zur Implementierung des Busses zurückgegriffen werden, erhöht dieser Ansatz die Codekomplexität nur minimal.

Da der Trigger über viele Register verfügen kann, deren Werte nicht ausgelesen werden müssen, kann zur Senkung des Hardwarebedarfs der Lesezugriff auf alle nicht relevanten Register vor der Synthese deaktiviert werden. Dies lässt sich zentral in der Konfiguration einstellen und hat ebenfalls Auswirkungen auf das Kernmodul des Analyzers.

Die logische Struktur der Triggerlogik wurde in der Konzeption bereits ausführlich diskutiert. Es kommen Trigger zum Erkennen von Werten und Flanken zum Einsatz. Sie werden jeweils durch eine Entität `TRIGGER_VALUE` bzw. `TRIGGER_EDGE` repräsentiert und von dem Modul `TRIGGER` instanziiert. Die Anzahl der Instanzen lässt sich durch die zentrale Konfiguration vor der Synthese bestimmen, wobei mindestens ein Objekt ausgewählt werden muss.

Jede Instanz der Entität `TRIGGER_VALUE` verfügt über die zwei Register `VALUE_MASK` und `SENSITIVITY_MASK`. Diese seien im Folgenden mit $\underline{v} = (v_0, ..., v_{n-1})$ und $\underline{s} = (s_0, ..., s_{n-1})$ be-

zeichnet. Das Ergebnis $t(\underline{x})$ wird dann als Funktion der Triggereingänge $\underline{x} = (x_0, ..., x_{n-1})$ kombinatorisch als $t(\underline{x}) = \bigwedge_i [(x_i \leftrightarrow v_i) \vee \neg s_i]$ berechnet.

Die Trigger zum Erfassen von Flanken sind analog konstruiert, wobei zusätzlich ein Register benötigt wird, um den Wert $\underline{x'}$ des letzten Taktes zwischen zu speichern. Die kombinatorische Logik vergleicht dann die beiden Werte unter Berücksichtigung der Parameter `REQUIRE_RISING_EDGE`, hier als Vektor $\underline{r}$ notiert, und `REQUIRE_FALLING_EDGE`, kurz $\underline{f}$. Die Funktion lautet $t(\underline{x'}, \underline{x}) = \bigwedge_i [(f_i \wedge x'_i \wedge \neg x_i) \vee (r_i \wedge \neg x'_i \wedge x_i) \vee (\neg f_i \wedge \neg r_i)]$.

Während sich also bei beiden Triggertypen der Beitrag jedes Bits durch eine Logikzelle mit drei bzw. vier Eingängen bestimmen lässt, kann die äußere Konjunktion für breite Triggereingänge nicht einstufig bestimmt werden. Die Laufzeit skaliert daher mit $\log_a n$, wobei a der Anzahl der Eingänge einer Look-Up-Table (typischerweise 4 bis 5) entspricht.

Aus diesem Grund enthält das Triggermodul eine zweistufige Pipeline, in deren ersten Schritt sämtliche Trigger-Instanzen parallel arbeiten, um dann in der zweiten Stufe zu einem gemeinsamen Ergebnis verbunden zu werden. Für die zweite Stufe lassen sich durch das Register `TRIGGER_ENABLE_MASK` die einzelnen Instanzen mit einer Bitmaske auswählen. Der endgültige Triggerwert berechnet sich dann als Disjunktion aller aktiven Instanzen. Nach einem Reset sind alle Trigger zunächst deaktiviert, um ein ungewolltes Triggern zu verhindern.

Zwischen dem Auftreten eines Triggerereignisses und dessen Erkennen liegen konstant zwei Zyklen des Abtasttaktes. Der Ausgang der Triggerlogik wird an der Eingangsstufe der Sampling-Pipeline übergeben. Unabhängig von deren Gestalt bleibt der Versatz zwischen Daten und der Detektierung des Events konstant zwei Taktzyklen lang. Dies sollte beim Berechnen des *Pretriggering*-Rahmens durch die Steuersoftware berücksichtigt werden.

5.2.9 Zusammenfassung

Abschließend soll das Design anhand einer Pipeline maximaler Länge in Hinblick auf den Datenfluss innerhalb der Sampling-Pipeline zusammengefasst werden. Bei dem Beispiel handelt es sich um die in den Abbildungen 11b und 13 skizzierte Konfiguration. Wie bereits ausgeführt, sind die ersten beiden Module (synchronisierendes Speicherglied und Testmustergenerator) optional und haben im Normalbetrieb – bis auf eine Verzögerung von einem Takt pro Entität – keinen Einfluss auf die Daten. Daher umfasst die pipeline-relevante Signatur beider Einheiten nur die abzutastenden Daten sowie die Triggereingänge.

Anders als die ersten beiden Stufen arbeiten alle folgenden Module aktiv auf den Daten und berechnen teils neue Werte. Ihre Signaturen ändern sich daher in jedem Schritt. Die Eingangsstufe und der Trigger arbeiten parallel, wobei die aus einem Bit bestehende Ausgabe der Ereigniserkennung rückgeführt wird und der Eingangsstufe als Eingabe dient. Hierdurch ist der zeitliche Versatz zwischen Daten und dem Erkennen eines Ereignisses unabhängig von der Eingangsstufe, oder den folgenden Modulen. Das Ereignis durchläuft nun die gesamte Pipeline und wird u.a. von der Lauflängenkompression und dem Speichermanagers berücksichtigt.

Je nach konfigurierter Bankanzahl reduziert die Eingangsstufe die Breite des Eingangssignals. Ist mehr als eine Bank aktiv, werden diese zu einem Abtastzyklus zusammengefasst und von

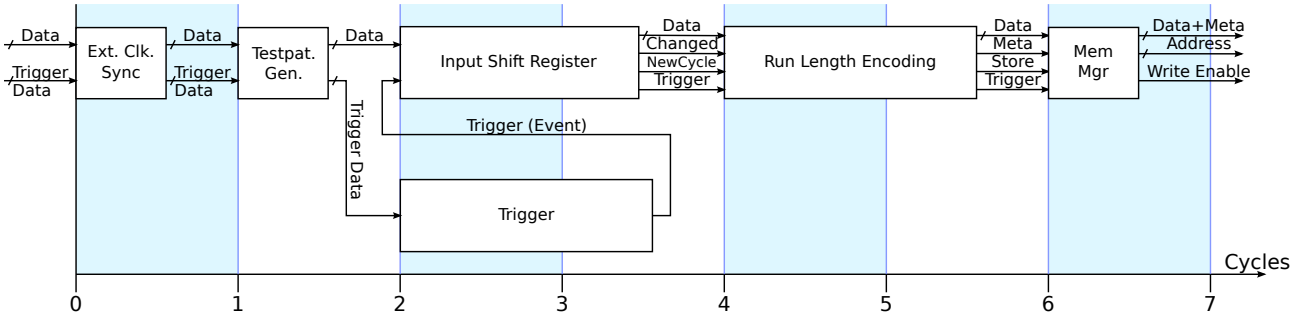


Abbildung 13: Datenfluss innerhalb der Sampling-Pipeline am Beispiel einer Konfiguration maximaler Länge

den folgenden Modulen als eine sequentiell verarbeitete Einheit betrachtet. Zur Synchronisation dient das `new_cycle` Signal, das jeweils bei der ersten Bank eines Zyklus aktiv wird. Kann aufgrund der Konfiguration des Systems nur eine Bank ausgewählt werden, ist die Leitung kontinuierlich `high`. Zusätzlich signalisiert das `changed` Bit, das jeweils für einen Abtastzyklus gültig ist, ob an den aktiven Bänken Änderungen auftraten. Das *Input Shift Register* verzögert ein Triggerereignis bis zum Beginn des nächsten Zyklus. Ein einmalig am Ausgang der Eingangstufe signalisiertes Trigger Event wird bis zu einem Reset nicht mehr zurückgenommen.

Die Kompressionsstufe – hier repräsentiert durch die Lauflängenkompression – kontrolliert, welche Daten in den Speicher gelangen. Hierfür wird das Signal `store` verwendet. Darüber hinaus kann die Stufe Zusatzinformationen ablegen. Diese werden mittels des Meta-Bits gekennzeichnet. Erreicht ein aktives Triggersignal die *RLE* während die Eingänge konstant sind, d.h. kein Speicherzugriff stattfindet, wird dieses Signal als Änderung gewertet und solange verzögert, bis der Zähler in den Speicher geschrieben wurde und die erste Bank zu Beginn eines neuen Abtastzyklus dem Speichermanager übergeben wird.

Die Implementierung der RLE sieht eine Laufzeit von zwei Taktzyklen vor, wobei im ersten Schritt die Informationen verarbeitet werden und das Ergebnis berechnet wird, während die zweite Stufe für den Zwischenspeicher (vgl. Kapitel 5.2.5) benötigt wird.

Die letzte Stufe steuert den Speicherzugriff. Hierzu werden die Daten um eine Speicheradresse ergänzt und zusammen mit dem `write_enable`-Bit dem Speichertreiber übergeben. Dieser ist nicht mehr Teil des Frameworks.

5.3 Bibliothek `RS232_CONTROLLER`

Die Library `RS232_CONTROLLER` implementiert die Schnittstellenumsetzung zwischen dem externen RS232-Interface und dem internen *WISHBONE Bus*. Das Design des Adapters ist durch die Asymmetrie der Übertragungsraten der beiden Schnittstellen geprägt. Zwar legen die Standards beider Interfaces keine Geschwindigkeiten fest, jedoch sind bei modernen RS232-Schnittstellen maximale Bitraten zwischen 115.2 KBit/s und 1 MBit/s üblich. [Wik11c] Die Datenrate auf dem *WISHBONE Bus* wird primär durch dessen Datenbreite und die Taktfrequenz bestimmt. So ergibt sich als realistische Mindestabschätzung für einen 8 Bit breiten Bus, der mit 50 MHz getaktet ist und zwei Zyklen pro Übertragung benötigt, eine Datenrate von 200 MBit.

Befehl	Anforderungsformat	Antwort bei Erfolg
Setze Adressregister	`'x' adresse crc ';'`	`'+'`
Setze Blocklänge	`'z' laenge crc ';'`	`'+'`
Schreibe Datenwort	`'y' wert crc ';'`	`'+'`
Lese ein Wort	`'r'`	`wert crc '+'`
Lese Block	`'m'`	`wert {wert} crc '+'`

Tabelle 1: Befehlsformat der RS232-Schnittstelle in EBNF. `adresse`, `laenge`, `wert` und `crc` sind Nichtterminale und repräsentieren Werte in Hexadezimal-Darstellung.

Folglich ist die Übertragung auf dem internen Bus ($< 1\%$ der Zeit) gegenüber der externen Kommunikation vernachlässigbar. Daher wird die Bearbeitung der Schnittstellen nur an Stellen parallelisiert, an denen hierdurch kein zusätzlicher Hardwarebedarf zu erwarten ist.

Zur Steuerung der seriellen Schnittstelle wird der freie UART *MiniUART IP Core* [Car] verwendet, der selbst über den *WISHBONE Bus* angeschlossen wird und den Übergang zwischen den Taktdomänen der RS232-Schnittstelle und dem Systemtakt übernimmt. Der Schnittstellen-Umsetzer implementiert daher nur zwei *WISHBONE Bus* Master, die beide synchron arbeiten. Die Verhaltensbeschreibung ist als Moore-Automat mit Datenpfad modelliert. Dies erscheint insbesondere deshalb sinnvoll, weil die durch das Protokoll induzierte formale Sprache regulär ist und daher durch einen Automaten erkannt werden kann [Win02]. Strukturell ist der Automat in eine eigene Entität gekapselt, da sich hierdurch *Unit Tests* einfacher durchführen lassen.

Wie in Kapitel 2.4 beschrieben, legt der *WISHBONE Bus* Standard bereits ein Kommunikationsprotokoll fest. Für die RS232-Schnittstelle existieren dagegen nur rudimentäre Konventionen für die untersten beiden Schichten des ISO/OSI-Modells (Bitübertragungs- und Sicherungsschicht). Daher muss für diese Anwendung ein eigenes Protokoll (siehe Tabelle 1) entwickelt werden.

Dieses verwendet ausschließlich druckbare Zeichen des ASCII-Zeichensatzes und ist daher mit nahezu allen modernen Rechnersystemen direkt kompatibel. Es ist in erster Linie auf Erweiterbarkeit, Lesbarkeit und Sicherheit ausgelegt, wodurch an einigen Stellen Performance-Nachteile auftreten. Aufgrund der sehr begrenzten Datenmenge, die für die Steuerung des Logic Analyzers übertragen werden muss, sind die Geschwindigkeitseinbußen jedoch hinnehmbar.

Die größten Performance-Verluste entstehen dadurch, dass alle Datenwerte hexadezimal repräsentiert werden. Dies verursacht zwar eine Verdoppelung der zu transportierenden Daten (ein Zeichen umfasst 8 Bits, kann aber nur 4 Bits kodieren), bringt jedoch eine Reihe an Vorteilen: Durch die Beschränkung des Protokolls auf druckbare Zeichen, kann nachträglich eine *Datenfluss-Steuerung* mittels `XON/XOFF` eingefügt werden. Da nur wenige Symbole zur Darstellung von Zahlenwerten benötigt werden, können eindeutige Metazeichen, z.B. zur Beschreibung von Befehlen, genutzt werden.

Die Kommunikation findet auf logischer Ebene paketorientiert statt und besteht immer aus einer Anforderung, die vom Host gesendet wird, und einer Antwort diese Moduls. Der Adapter initiiert keine Übermittlungen. Jede Anforderung enthält genau einen Befehl, der durch das

erste Zeichen kodiert wird. Handelt es sich um eine Leseanforderung, besteht das Paket nur aus diesem Zeichen. Es gibt zwei Befehle zum Lesen, wobei einer nur von der aktuellen Adresse des *WISHBONE Bus* liest, während der zweite einen Block einstellbarer Länge ermittelt. Nach jedem Lesevorgang wird die Adresse des Busses automatisch erhöht, sodass durch das Senden zweier Leseanforderungen die Werte zweier aufeinander folgender Adressen erfasst werden. Die Antwort enthält – unabhängig von deren Umfang – eine Prüfsumme konstanter Länge.

Befehle zum Beschreiben der Register des Adapters enthalten zusätzlich den neuen Wert des Registers, eine Prüfsumme sowie ein Symbol um das Ende der Übertragung zu kodieren. Insgesamt existieren drei Register. Diese speichern die aktuelle *WISHBONE Bus* Adresse, das Datenwort sowie die Anzahl der Adressen, die beim Lesen eines Datenblocks erfasst werden sollen. Wird das Datenwort verändert, startet automatisch ein Schreibvorgang auf dem *WISHBONE Bus*, der das neue Wort auf die aktuelle Adresse schreibt und diese danach inkrementiert.

Unabhängig von der Befehlsart wird jede Antwort durch ein Symbol, das das Ende der Nachricht markiert, abgeschlossen. Es kommen zwei Zeichen zum Einsatz, wobei eines eine erfolgreiche Ausführung kodiert und das andere im Fehlerfall gesendet wird. Zwei Fehlerquellen sind beispielsweise eine falsche Prüfsumme oder ein ungültiges Anforderungsformat. Wird beim Empfang eines Befehls ein Fehler erkannt, wechselt der Automat nach dem Senden des Fehlersymbols in den Ruhezustand.

Da die Zeichen zum Kodieren eines Befehls nicht anderweitig verwendet werden, kann sich der Adapter nun auf den Beginn der nächsten Anforderung synchronisieren.

Das Protokoll unterscheidet nicht zwischen Groß- und Kleinschreibung. Da sich im ASCII-Zeichensatz die zwei Vertreter eines Buchstabens nur im 6. Bit unterscheiden, erfordert diese Freiheit keiner zusätzlichen Hardware und kann den Bedarf sogar reduzieren.

Die Prüfsumme wird mit Hilfe des CRC-Algorithmus [PB71] und einem Generatorpolynom vom Grad 4 berechnet. Die Prüfsumme ist daher 4 Bit lang und lässt sich durch ein Zeichen hexadezimal kodieren. Zur hardwareseitigen Berechnung und Verifikation des Codes wird der freie *Ultimate CRC* Core [Dra09] verwendet.

Die Datenwerte werden über ein Schieberegister, das diese um 4 Bit pro Takt versetzt, erfasst und ausgegeben. Dabei wird *big-endian* verwendet. Da jedes Bit des Speichergliedes zu Beginn eines Befehls auf 0 gesetzt wird, müssen beim Schreiben von Datenwerten führende Nullen nicht übermittelt werden.

5.4 Bibliothek `JTAG_CONTROLLER`

Die Bibliothek `JTAG_CONTROLLER` ermöglicht das Steuern des Logic Analyzers über die *JTag*-Schnittstelle. Hierbei ist es grundsätzlich wünschenswert, den im FPGA vorhandenen *TAP Controller* zu verwenden, da nur auf diese Art keine zusätzlichen IO-Pins benötigt werden. Besteht kein Zugriff auf den integrierten Controller, kann ein zweiter Controller durch zusätzliche Logik erzeugt werden. Dies erscheint aber nicht sinnvoll, da in üblichen Konfigurationen die

effektive Datenrate[27] des *JTag*-Ports geringer ist als beim RS232-Interface und der *JTag*-Port mehr Chip-Pins benötigt. Um sich trotzdem vor dieser Möglichkeit nicht zu verschließen und eine möglichst große Abstraktationsebene zu erreichen, wurde der Interface-Umsetzer gekapselt und erwartet – anders als der RS232-Adapter – die Zuführung bereits *dekodierter* Signale.

Hierfür wird das Interface des *Xilinx Spartan-3 BSCAN*-Primitives übernommen, das gemäß der entsprechenden Datenblätter in kompatibler Form auch in anderen Chips des gleichen Herstellers vorhanden ist (z.B. *Spartan 2*, *Spartan-3A/E*). Durch simple Glue-Logic ist es ebenfalls möglich *Virtual JTAG*, das von vielen *Altera* FPGAs angeboten wird, [Alt08] zu nutzen.

Das Interface erlaubt den Zugriff auf (mindestens) zwei logische Datenregister, wobei diese durch den Host mittels unterschiedlicher *Instruction Codes* adressiert werden. Einige der Zustände des *TAP-Controllers* stehen mittels *One-Hot-Kodierung* der Logik zur Verfügung. Dies sind der `Reset`-, `Capture`-, `Shift`- und `Update`-Zustand (vgl. Kapitel 2.5), wobei alle – mit Ausnahme des Reset-Zustands – nur in Verbindung mit einem aktiven Datenregister gültig sind. Die beiden letzten Zustände sind weiterhin synchron zum *JTag*-Takt aktiv. [Xil11d]

Im vorangehenden Kapitel wurde die Verhaltensbeschreibung anhand eines zentralen Automaten modelliert. Für dieses Design erschien dieser Ansatz nicht sinnvoll, da – anders als beim RS232-UART – hier die Synchronisation zwischen drei Taktdomänen (Systemtakt, JTag-Takt und der `Capture`-Latch) explizit durchgeführt werden muss. Die Teilfunktionalitäten sind daher algorithmisch in der jeweils gültigen Domäne angegeben und kommunizieren mittels Semaphoren.

Da die *JTag*-Schnittstelle über keine Datenkonsistenzprüfung verfügt, wird diese durch das Protokoll anhand eines *Paritätsbits* realisiert. Das Protokoll selbst kann als *JTag-over-JTag* betrachtet werden. Eines der Datenregister wird als Instruktionsregister verwendet, das beispielsweise zwischen adressieren, lesen und schreiben umschaltet. Das zweite Register dient als reguläres Datenregister.

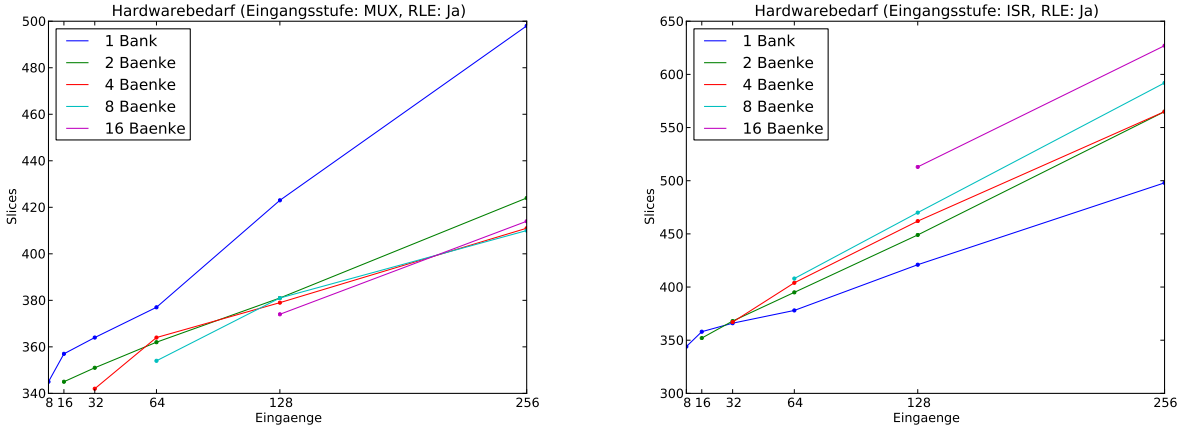
Da der *JTag*-Standard für jede Instruktion ein Datenregister fester Größe vorschreibt, existieren zwei Betriebsmodi. Der voreingestellte Modus arbeitet standardkonform und nutzt ein Datenregister, dessen Länge sich an der des größten Datenregisters[28] orientiert. Der adaptive Modus passt die Größe des Schieberegisters an die ausgewählte Instruktion an. Insbesondere ermöglicht dies das sequentielle Lesen mehrerer Werte.

Auf dem PC kommt im Referenzdesign die freie Software *urJTag* [urj11] zum Einsatz, die als *JTag*-Treiber fungiert und den Zugriff auf eine Vielzahl an Hardware-Adaptern ermöglicht. Mit diesem Tool kann der adaptive Modus durch die Mehrfachbelegung von Instruktionen genutzt werden.

---

[27] Der *JTag* Standard spezifiziert keine Übertragungsrate, jedoch arbeiten typische Geräte unterhalb 30 MHz (z.B. [Xil11c]). Drosselnd wirkt häufig die verwendete Software-Emulation des *JTag* Interfaces via *LPT*. Zusätzlich sinkt der Datendurchsatz aufgrund der Zustandswechsel des TAP-Controllers signifikant.

[28] je nach Konfiguration des Analyzers haben Daten- und Adressleitung des *WISHBONE Bus* eine unterschiedliche Breite. Es wird daher der größere Wert gewählt.



(a) *Input Multiplexer* **(b)** *Input Shift Register*

Abbildung 14: Flächenbedarf als Funktion der Breite der Eingänge sowie der Bankanzahl. Werte sind *Post-Synthesis*-Zwischenergebnisse und beziehen sich auf ein komplettes `SoC` inklusive *RS232*- und Speichertreiber. Der interne Datenbus hat eine Breite von 8 Bit. Der verwendete *Low cost FPGA* verfügt über 2000 Slices (Makrozellen). In realistischen Konfigurationen werden demnach rund 20% für den Logic Analyzer benötigt.

Das Protokoll bietet zusätzliche Instruktionen, mit denen Informationen über die Konfiguration des *WISHBONE Bus* abgefragt werden können. Dies ermöglicht dem PC-Treiber die automatische Anpassung an die Hardware.

5.5 Kenngrößen der Implementierung

Die Hardwarebeschreibung wurde für einen *Xilinx Spartan-3 XC3S200 -4* synthetisiert und mit diesem ausführlich erprobt. Unter Nutzung aller *BRAM-Primitive* und mit voll ausgestattetem Framework lässt sich ein Entwurf für eine Abtastfrequenz bis zu 99 MHz synthetisieren. Der *WISHBONE Bus* Takt bleibt auf etwa 85 MHz begrenzt.

Abbildung 14 stellt den Hardwarebedarf in Abhängigkeit der Eingänge dar. Hierbei ergibt sich erwartungsgemäß, dass der *Input Multiplexer* zur Reduzierung des Ressourcenverbauchs verwendet werden kann. Da bei dieser Konfiguration kaum Logik vor dem Multiplexer angesiedelt ist, wirkt sich eine Verdoppelung der Anzahl der Eingänge bei gleichzeitiger Verdoppelung der Bankanzahl nicht signifikant aus. Somit können etwa mehrere Busse innerhalb eines *ManyCore*-Systems angeschlossen werden, um dann zur Laufzeit zu bestimmen, welcher überwacht werden soll.

Aufgrund der komplexen Struktur trägt das *Input Shift Register* in der gegenwärtigen Implementation nicht zur Senkung der benötigten Fläche bei. Es eignet sich daher nur in Fällen, bei denen eine sequentielle Verarbeitung der Daten nötig ist, z.B. wenn ein externer Speicher mit begrenzter Datenbreite zum Einsatz kommt.

Der Hardwarebedarf steigt annähernd linear mit der Breite des internen Datenbusses. Daher wird konsequent eine Busbreite von 8 Bit empfohlen. In Verbindung mit einer *RS232*-

Schnittstelle wirkt sich dies nur bei Schreibzugriffen, die in Fallstudien[29] weniger als 1% der Operationen ausmachten, negativ auf den Datendurchsatz aus. Im Vergleich zu einer vollparallelen Anbindung mit 256 Eingängen können mit einem schmalen Datenbus mehr als 300 Makrozellen (rund 60 %) eingespart werden.

Während der Entwicklung wurde die RS232-Schnittstelle mit 230 KBit/s betrieben, wodurch sich beim Lesen von zusammenhängenden Speicherbereichen eine Nutzdatenrate von rund 14 KiB/s ergibt. Bei der Benutzung aller BRAM-Zellen lässt sich ein Speicherabbild folglich in rund 1.5s übertragen. Die Datenrate wurde bei den Tests durch den UART des PCs begrenzt. Eine Übertragungsrate von 1 MBit/s ist seitens des Analyzers vorgesehen, konnte jedoch nicht erprobt werden.

Die *JTag*-Schnittstelle konnte bisher nur über eine LPT-Emulation, die über einen ineffizienten Software-Stack gesteuert wurde, betrieben werden. Hierbei ergab sich eine durchschnittliche Brutto-Datenrate von rund 6 KiB/s.

6 Fazit

Dieses abschließende Kapitel reflektiert über die Arbeit und überprüft dabei kritisch, wo Stärken und Schwächen des Konzeptes sowie der Implementierung liegen. Im Sinne eines evolutionären Entwicklungsmodells ist dabei das Ziel, Ansatzpunkte für etwaige Verbesserungen zu identifizieren, und Vorschläge zu unterbreiten, in welche Richtung das Projekt weiterentwickelt werden könnte.

Bereits in Kapitel 2 wurde deutlich, dass Logic Analyzer eine sehr große Geräteklasse stellen, deren konkrete Instanzen teils sehr unterschiedlich ausfallen. In der Konzeption wurde daher die Entscheidung getroffen, sich nicht an Extremen zu orientieren, sondern ein möglichst modulares Framework zu kreieren, das sich auf die konkreten Bedürfnisse des Anwendungsfalls anpassen lässt. Im *best-case* können alle Anpassungen durch Änderung der Konfiguration vorgenommen werden. Der Code wurde jedoch so implementiert, dass einzelne Einheiten möglichst einfach ausgetauscht werden können.

Die Entwicklung neuer oder angepasster Komponenten wird dabei durch die Infrastruktur (inklusive der entwickelten Softwarelösungen, die hier nicht behandelt werden) unterstützt, die vom Framework zur Verfügung gestellt wird. Diese wird als eine der Hauptleistungen des Projektes gewertet.

So ist das Projekt zwar auf den Einsatz als integrierter Logic Analyzer, d.h. zur Überwachung interner Signale eines FPGAs, zugeschnitten. Die Implementierung kann jedoch ebenfalls ohne Änderung als dedizierter Analyzer betrieben werden.

Der flexible Designansatz bringt jedoch mit sich, dass der Anwender gezwungen wird, sich mit den strukturellen Details des Designs auseinanderzusetzen, bevor er das Projekt nutzen kann.

[29] statistische Auswertung der Zugriffsmuster beim Abarbeiten von Testbenches sowie praxisnahen Anwendungsszenarien

Dieses Problem ließe sich zukünftig beispielsweise durch einen Assistenten nach dem Vorbild des *Xilinx IP Core Generator and Architecture Wizard*, der bei der Instanziierung des Cores behilflich ist, entschärfen.

Alle in der Konzeption aufgestellten Anforderungen konnten in der Implementierung umgesetzt werden. Die Kerndisziplin, nämlich das parallele Aufzeichnen vieler Signale, konnte in diversen Testszenarien und Einsätzen verifiziert werden. Dabei wurde mit einem *Xilinx Spartan-3* FPGA eine Abtastfrequenz von 99 MHz erreicht und durch das *Static Timing Analysis*-Werkzeug des Herstellers abgesichert.

Während der Entwicklung des *JTag*-Schnittstellentreibers konnte der Analyzer über die *RS232*-Schnittstelle aktiv beim Debugging unterstützen. Hierbei war die Lauflängenkompression von großem Nutzen, da die Signale mit mehr als einer Größenordnung *oversampled* werden konnten, ohne dabei den Speicher im gleichen Maße zu belasten. Somit konnten rudimentäre, aber hilfreiche Aussagen bzgl. des *Timings* getroffen werden.

Der generische, verlustfreie Kompressionsalgorithmus ist gegenüber den betrachteten existierenden integrierten Analyzern ein Alleinstellungsmerkmal. Ähnliches gilt für das sequentielle Multiplexing der Eingänge, das in keinem anderen Produkt der Klasse gefunden werden konnte.

Wie in Kapitel 5.5 beschrieben, kann der *Input Multiplexer* für die Auswahl von bis zu 4 Bänken ohne signifikant gesteigerten Hardwarebedarf verwendet werden. Für eine höhere Anzahl steigt die benötigte Fläche in einem angemessenen Rahmen.

Anders als in der Konzeption geplant, muss die zusätzliche Funktionalität des *Input Shift Registers* in der bisherigen Implementierung „teuer“ erkauft werden. Es eignet sich daher primär in Fällen, in denen der zusätzliche Hardwarebedarf hinnehmbar ist, oder wenn Datenströme erfasst werden sollen, welche die Datenbreite des Speichers übersteigen. Hier wäre die Untersuchung alternativer Ansätze zur Implementierung interessant.

Die Anbindung des Triggermoduls an den Analyzer wurde durch spezielle Testfälle untersucht und erwies sich als sehr zuverlässig und insbesondere zeitlich vorhersagbar. Das Triggern auf Flanken hingegen stellt im Vergleich zu den Konzepten anderer untersuchter Geräte eine schwächere Lösung dar. Die Einsetzbarkeit des Projektes kann etwa durch die Implementierung von Triggersequenzen deutlich erhöht werden. Auch denkbar ist ein minimaler Prozessorkern, der im Vergleich zu den existierenden Strukturen sehr frei programmiert werden könnte. Aufgrund einer Priorisierung der Features wurden diese Ansätze nicht weiter verfolgt, könnten aber in zukünftigen Entwicklungen untersucht werden.

Während der Konzeptionsphase war ein Hauptgrund für die Wahl der *RS232*- und *JTag*-Schnittstellen, dass diese verhältnismäßig platzsparend in Hardware implementiert werden können. Nichtsdestotrotz wird derzeit ein großer Teil der Logik[30] für das Handling der externen Kommunikation, insbesondere der Protokolle, aufgewendet. Daher bietet es sich für zukünftige

[30] stark konfigurationsabhängig; in einer typischen Anwendung rund 100 bis 150 Makrozellen

Weiterentwicklungen an, die Möglichkeit zu untersuchen einen Mikrocontroller als *WISHBONE Bus* Master einzusetzen.

Im Gegensatz zu reinen Hardwarelösungen könnten dadurch einfache Steuerungsfunktionen durch den integrierten Prozessor übernommen werden, wodurch die Kommunikation zwischen PC und Analyzer auf einer höheren Abstraktionsebene möglich wäre. Es ist zu erwarten, dass somit das Datenaufkommen über die Schnittstelle reduziert und das System – vor allem durch Senkung der Latenzzeiten – beschleunigt werden kann. Darüber hinaus ließen sich leistungsfähigere Schnittstellen, wie etwa *Ethernet*, implementieren.

Das *JTag*-Protokoll verfügt, aufgrund des in Kapitel 5.4 beschriebenen Tunnelings, über keine optimale Performance. Weitere Bemühungen, die Anzahl der benötigten Registerwechsel und den damit verbundenen Protokoll-Overhead zu reduzieren, sind erstrebenswert.

Nach Meinung des Autors wurde im Zuge dieser Arbeit ein Produkt geschaffen, welches bei der Entwicklung und Verifikation von Entwürfen den Anwender aktiv unterstützt. Konkret ist der Einsatz in einem Forschungsprojekt aus dem Bereich *SoC*s geplant.

Nahziele sollten insbesondere den *Software-Stack* betreffen. Da der Fokus dieser Ausarbeitung auf der Hardwarebeschreibung liegt, assistiert die bisher entwickelte Software besonders bei Aufgaben, die während der Entwicklung anfallen. Zwar wird die in Appedix B beschriebene Struktur als sehr flexible Lösung, die sich auch für den Produktiveinsatz eignet, betrachtet, jedoch könnte die Ausführungsgeschwindigkeit, gerade in Verbindung mit der *JTag*-Schnittstelle optimiert werden. Darüber hinaus wäre ein robustes *User Interface* wünschenswert, wodurch das Projekt deutlich komfortabler nutzbar wäre. Eine Integration in vorhandene Software, etwa dem *Sigrok*-Projekt, ist denkbar.

Letztlich erscheint die Portierung des Projektes, z.B. auf eine *Altera* Architektur, sinnvoll, da nur so der Beweis der weitgehenden Hardware-Unabhängigkeit erbracht werden kann.

Zusammenfassend wird das Projekt als Erfolg gewertet. Es wurde ein freies und leistungsfähiges Werkzeug geschaffen, das flexibel dem Einsatzgebiet angepasst werden kann. Hierdurch lässt sich die Hardwarenutzungsdichte maximieren, d.h. durch Deaktivieren nicht benötigter Funktionen ein kompaktes Gesamtsystem erzeugen. Wie in Kapitel 5.5 gezeigt, skaliert das Gesamtsystem darüber hinaus gut mit der Anzahl der Eingänge. Die gefundenen Schwächen sind nicht konzeptioneller Natur und lassen sich durch die Weiterentwicklung des Projektes überwinden. Um diese zu fördern ist die Veröffentlichung der Quellen inklusive der verwendeten Software auf *OpenCores.org* geplant.

7 Literaturverzeichnis

Literatur

[Gai] Gaisler, Jiri: *A structured VHDL design method.* www.gaisler.com/doc/vhdl2proc.pdf

[LL07] Ludewig, Jochen ; Lichter, Horst: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.verlag, 2007. – ISBN 3–89864–268–2

[PB71] Peterson, W.W. ; Brown, D.T: *Cyclic Codes for Error Detection.* 1971

[Sha49] Shannon, C.E.: Communication in the presence of noise. In: *Proc. Institute of Radio Engineers* Bd. 37, 1949, S. 10–21

[Win02] Winter R.: *Theoretische Informatik.* 2002

Handbücher und Standards

[Fre07] Free Software Foundation: *General Public License Version 3.* http://www.gnu.org/licenses/gpl-3.0.html. Version: 2007

[IEEa] IEEE: *IEEE Standards Interpretations: IEEE Std. 1076-1987, IEEE Standard VHDL Language Reference Manual*

[IEEb] IEEE: *IEEE Standards Interpretations: IEEE Std. 1076-1993, IEEE Standard VHDL Language Reference Manual*

[IEEc] IEEE: *IEEE Standards Interpretations: IEEE Std. 1076-2002, IEEE Standard VHDL Language Refernce Manual*

[IEEd] IEEE: *IEEE Standards Interpretations: IEEE Std. 1149.1-1990, IEEE Standard VHDL Test Access Port and Boundary-Scan Architecture Manual*

[IEEe] IEEE: *IEEE Standards Interpretations: IEEE Std. 1164-1993, IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Stdlogic1164)*

[Ope10] OpenCores: *Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecturefor Portable IP Cores*, 2010. http://opencores.org/opencores,wishbone

[Xil] Xilinx Inc.: *ChipScope Pro and the Serial I/O Toolkit.* http://www.xilinx.com/tools/cspro.htm

[Xil11a] Xilinx Inc.: *DS322: LogiCORE IP Distributed Memory Generator v5.1.* http://www.xilinx.com/support/documentation/ip_documentation/dist_mem_gen_ds322.pdf. Version: 2011

[Xil11b] Xilinx Inc.: *UG029: ChipScope Pro Software and Cores User Guide v13.2.* http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/chipscope_pro_sw_cores_ug029.pdf. Version: 2011

[Xil11c] Xilinx Inc.: *UG332: Spartan-3 Generation Configuration User Guide.* www.xilinx.com/support/documentation/user_guides/ug332.pdf. Version: 2011

[Xil11d] Xilinx Inc.: *UG607: Spartan-3 Libraries Guide for HDL Designs.* http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/spartan3_hdl.pdf. Version: 2011

Verweise und URLs

[Alt08] Altera Corporation: *Megafunction User Guide – Virtual JTAG (sld_virtual_jtag)*. www.altera.com/literature/ug/ug_virtualjtag.pdf. Version: 2008

[Alt11] Altera Corporation: *Altera Product Selector v1.0*. http://www.altera.com/products/selector/psg-index.html. Version: 2011

[Car] Carton, Philippe: *MiniUART IP Core*. http://opencores.org/project,miniuart2

[Dra09] http://opencores.org/project,ultimate_crc

[lec11] *LeCroy Logic Studio - Datasheet*. http://cdn.lecroy.com/files/pdf/lecroy_logicstudio_datasheet.pdf. Version: 2011

[Pop07] Poppitz, Michael: *FPGA Based Logic Analyzer*. http://www.sump.org/projects/analyzer/. Version: 2007

[RW01] Rossum, Guido van ; Warsaw, Barry: *PEP 8 – Style Guide for Python Code*. http://www.python.org/dev/peps/pep-0008/. Version: 2001

[sig11] http://sigrok.org/wiki/Main_Page

[tek11a] *Tektronix SPI-4.2/SPI-3 Support Package*. http://www2.tek.com/cmswpt/psdetails.lotr?ct=PS&cs=psu&ci=14908&lc=EN. Version: 2011

[tek11b] *Tektronix TLA7000 Series*. http://www.tek.com/products/logic-analyzer/tla7000/. Version: 2011

[urj11] *UrJTAG - Universal JTAG library, server and tools*. http://urjtag.org/. Version: 2011

[Wei11] http://logicanalyzer.sourceforge.net/

[Whe11] Where Labs, LLC: *Dangerous Prototypes - Open Bench Logic Sniffer*. http://dangerousprototypes.com/docs/Open_Bench_Logic_Sniffer. Version: 2011

[Wik11a] Wikipedia, The Free Encyclopedia: *Datei:Jtag chain.svg*. http://de.wikipedia.org/w/index.php?title=Datei:Jtag_chain.svg&filetimestamp=20060604110530. Version: 2011

[Wik11b] Wikipedia, The Free Encyclopedia: *Datei:JTAG TAP Controller State Diagram.svg*. http://de.wikipedia.org/w/index.php?title=Datei:JTAG_TAP_Controller_State_Diagram.svg&filetimestamp=20090725153043. Version: 2011

[Wik11c] Wikipedia, The Free Encyclopedia: *RS-232*. http://de.wikipedia.org/wiki/RS232. Version: 2011

A Inbetriebnahme

Dieses Kapitel beschreibt die zur Inbetriebnahme des Frameworks nötigen Schritte anhand eines konkreten Beispiels, das für einen Xilinx Spartan-3 FPGA synthetisiert wird. Das Howto ist Teil der offiziellen Dokumentation und daher in Englisch verfasst.

The logic analyzer can be used to observe the value of any logic vector of your design that lies within the scope of one entity which can be located anywhere in the hierarchy. However you have to keep in mind that the framework needs to communicate with the outside of the chip. If you use external memory or a communication port other than *JTag*, you probably have to route some signals to the top module of your design. To avoid unnecessary complexity, this example uses the top module. Please note that any code listed here is included in the *SOC_SPARTAN3* library in the source archive.

Embedding of the Framework

Consider the following code, that implements a simple 8 bit counter which is incremented every clock cycle given the reset bit is not asserted. The logic analyzer allows you to observe the evolution of the signal and therefore to verify the counter's behaviour. Of course this is a trivial example, but you can apply the steps to any other case.

```
1  library IEEE;
2      use IEEE.STD_LOGIC_1164.ALL;
3      use IEEE.NUMERIC_STD.ALL;
4
5  entity COUNTER is
6      port (
7          clk_i : STD_LOGIC;
8          rst_i : STD_LOGIC;
9          data_o : out STD_LOGIC_VECTOR(7 downto 0)
10     );
11 end entity;
12
13 architecture RTL of
14     signal data_l : STD_LOGIC_VECTOR(7 downto 0);
15 begin
16     counter_process: process(clk_i) is begin
17         if rising_edge(clk_i) then
18             if rst_i = '1' then
19                 data_l <= (others => '0');
20             else
21                 data_l <= STD_LOGIC_VECTOR(UNSIGNED(data_l) +
22                     TO_UNSIGNED(1, data_l'LENGTH));
23             end if;
24         end if;
25     end process;
26     data_o <= data_l;
27 end architecture;
```

First you have to add the necessary libraries (namely `LOGIC_ANALYZER` and either `JTAG_CONTROLLER` or `RS232_CONTROLLER`) to your project. Let's use a *RS232* port for communications and one *BRAM* primitive as storage. Thus you have to add the two *i/o ports* *rx_i* and *tx_o* to the entity's port. Furthermore the following additional import statements are required:

```
1  library LOGIC_ANALYZER;
2      use LOGIC_ANALYZER.LOGIC_ANALYZER_PKG.ALL;
3      use LOGIC_ANALYZER.CONFIG_PKG.ALL;
4
5  library RS232_CONTROLLER;
6      use RS232_CONTROLLER.RS232_CONTROLLER_PKG.ALL;
7
8  library UNISIM; -- memory
9      use UNISIM.VCOMPONENTS.ALL;
```

The next step is to instantiate the logic analyzer in the architecture's body and connect it to the *RS232* controller. The declaration of the local signals is omitted here:

```
1      my_la : LOGIC_ANALYZER_CORE
2      port map(
3          sa_clk_i => clk_i,
4          sa_data_i => data_l,    -- connect the signal to be observed
5          trg_data_i => data_l,   -- here
6
7          wb_clk_i => clk_i,
8          wb_rst_i => rst_i,
9          wb_o => wb_miso_l,
10         wb_i => wb_mosi_l,
11
12         mem_state_o => la_mem_state_l,
13         mem_clk_o => mem_sa_clk_l,
14         mem_o => mem_mosi_l,
15         mem_wb_i => mem_wb_miso_l,
16         mem_wb_o => mem_wb_mosi_l
17     );
18
19     my_rs232 : RS232_CONTROLLER_CORE
20     generic map (uart_br_divisor => 109) -- 110 KBps at 50 MHz
21     port map (
22         txd_pad_o => tx_o,
23         rxd_pad_i => rx_i,
24
25         wb_clk_i => clk_i,
26         wb_rst_i => rst_i,
27         wb_i => wb_miso_l,
28         wb_o => wb_mosi_l
29     );
```

You might need to alter the `uart_br_divisor`-generic to match your hardware requirements. The divisor is given by $\texttt{uart\_br\_divisor} = \frac{\text{frequency(clk\_i)}}{4 \cdot \text{baudrate}}$.

The logic analyzer is now connected to your design and able to communicate with the host. There are only two tasks remaining to embed the framework into your project: First you have to provide memory. As the interface is directly compatible to most two port memory interfaces, you just have to do the "wiring". As the read access is implemented via a *WISHBONE Bus* interconnection, you need to generate the acknowledge signal. The is done in the last line of the sample code without any hardware usage:

```
1   my_mem: RAMB16_S9_S9 port map (
2       -- write port
3       clka  => mem_sa_clk_l,
4       wea   => mem_mosi_l.we,
5       addra => mem_mosi_l.address,
6       dia   => mem_mosi_l.data,
7       ena => '1',
8       ssra => '0',
9       dipa => "0",
10
11      -- read port
12      clkb  => clk_i,
13      addrb => mem_wb_mosi_l.address,
14      dob   => mem_wb_miso_l.data
15      web => '0',
16      enb => '1',
17      ssrb => '0',
18      dib => X"00",
19      dipb => "0",
20  );
21  mem_wb_miso_l.ack <= mem_wb_mosi_l.stb;
```

The listing above utilises only one *BRAM* primitive. Several memory banks can be easily connected, either to increase the data or the address width. The reference design provided with the framework's sources includes the entity `SAMPLER_MEMORY`, which automatically produces both types depending on the configuration. Furthermore the module generates a multiplexer if the width of the *WISHBONE Bus* is smaller than the memory's data width. This however increases the address width by $\lceil log_2 \frac{\text{width(WISHBONE)}}{\text{width(memory data)}} \rceil$ bits.

The last step is to provide the framework with a valid configuration. A template suitable for this example is included in the source archive and attached at the end of this chapter.

Connect to the Analyzer

The software consists of a driver process and the application. Both require Python 3.0+ and can be executed on separate machines. In order to access the serial port, the `python-serial` module is needed for the driver. The *JTag* access is handled by *urJTag 0.1*. If this console application is started without *TTY* pipes, e.g. if they are routed to another application, the tool disables the interactive mode. This behaviour is undocumented and as it makes it impossible for the logic analyzer driver to interact with the tool, the software needs to be patched. Instructions are provided with the sources.

The main program of the driver module is the `server.py` file. Before the first start please open the file with a text editor and specify, whether the tool shall use the *RS232* or the *JTag* port. Further configurations such as the baudrate can be set there.

Similarly to the driver module, the header of the application's main file `client.py` contains two parameters that specify the *host name* and port number of the server. The default values point at the localhost. If both programs are running on the same machine, you do not have to alter those values.

The program enables you to setup the trigger, and download the captured data. More features can be accessed via Python's interactive console. Just navigate to the software's directory and start Python by typing `python3 -i interative.py`. The script connects to the server specified in its header section and registers the public object `la`. Use the help command or have a look into the documenation to inspect its functions.

```
1  package config_pkg is
2  -- SAMPLING DATA CONFIG
3      type INPUT_MUX_T is (ISR, MUX);
4      constant LA_CFG_BANK_WIDTH_C : INTEGER := 8;
5      constant LA_CFG_BANK_COUNT_C : INTEGER := 1;
6      constant LA_CFG_MUX_METHOD_C : INPUT_MUX_T := MUX;
7
8  -- WISHBONE CONFIG I (continued at bottom)
9      -- wishbone standard defines 8, 16, 32, 64
10     -- the logic analyzer core support any value bigger than 8
11     -- however only the standard values have been tested
12     constant LA_CFG_WB_DATA_WIDTH_C : POSITIVE := 8;
13     constant LA_CFG_WB_ADDR_WIDTH_C : POSITIVE := 12;
14
15     -- allow reading rw registers. disable to reduce hardware usage
16     constant LA_CFG_NO_ESSENTIAL_WB_READ_C : BOOLEAN := FALSE;
17
18 -- ENCODING CONFIG
19     -- include run-length encoding support
20     constant LA_CFG_RLE_C : BOOLEAN := FALSE;
21
22 -- TRIGGER
23     -- width of the trigger input port
24     constant LA_CFG_TRG_WIDTH_C : INTEGER:= 8;
25     -- number of value and edge triggers
26     constant LA_CFG_EDGE_TRIGGERS_C : INTEGER := 2;
27     constant LA_CFG_VALUE_TRIGGERS_C : INTEGER  := 2;
28
29 -- MISC
30     constant LA_CFG_INPUT_SYNC_WITH_EXT_CLOCK : BOOLEAN := FALSE;
31
32     -- enable reading of current data via wishbone (no external clock required)
33     constant LA_CFG_READ_CURRENT_DATA : BOOLEAN := FALSE;
34
35     -- if enabled input can be overwritten by test patterns
36     constant LA_CFG_TEST_PATTERN_C : BOOLEAN := FALSE;
37
38     -- set to any positive number to enable the clock divider
39     constant LA_CFG_CLKDIV_WIDTH_C : INTEGER  := 0;
40
41
42 -- JTAG INTERFACE
43     -- length of the block read shift register. set to any multiple of four
44     constant LA_CFG_JTAG_MAX_BLOCK_C : INTEGER  := 64;
45
46
47 -- MEM CONFIG
48     -- MEM DATA WIDTH (if rle is in use this constant has to be ¿= LA_CFG_BANK_WIDTH_C + 1)
49     constant LA_CFG_MEM_DATA_WIDTH_C : INTEGER := 8;
50     constant LA_CFG_MEM_MAX_ADDR_C : INTEGER := 2047;
51     constant LA_CFG_MEM_ADDR_WIDTH_C : INTEGER := 11;
52 end config_pkg;
```

B Software

Die Implementierung der Softwarelösung besteht aus einer Treiberschicht und dem Anwendungslayer. Beide Komponenten sind dabei als eigenständige Prozesse ausgelegt, die über das *Posix socket*-Modell, das unter allen populären Betriebssystemen verfügbar ist, kommunizieren. Dieses Design bietet intrinsisch die Möglichkeit, den Logic Analyzer über das Netzwerk anzusteuern.

Treiber

Die Treiberkomponente abstrahiert den Zugriff auf den *WISHBONE Bus* und bietet Funktionen zum Lesen und Schreiben von Registern. Für beide Operationen steht jeweils nur eine Funktion zur Verfügung, die sowohl einzelne Adressen als auch ganze Adressblocks bedienen kann. Es ist dabei dem Schnittstellentreiber überlassen, effiziente Algorithmen für den Zugriff auf die Hardware bereitzustellen.

Der Treiber implementiert eine Sicherungsschicht. Wird ein Fehler bei der Kommunikation zwischen PC und Logic Analyzer erkannt, versucht er diesen zu beheben. Bei Lesezugriffen kann dies etwa durch wiederholen des Vorgangs geschehen. Die Verbindung zwischen dem Treiber und der Anwendung wird automatisch durch die Verwendung von *sockets* gesichert. Das Design sieht somit vor, dass jeder Fehler, welcher der Anwendung gemeldet wird, als nicht automatisch behebbar eingestuft wird.

Das RFC[31]-Protokoll ist auf einer höheren Abstraktionsebene angesiedelt als das in Kapitel 5.3 beschriebene Konzept, und erlaubt die partielle Verarbeitung der Befehle ohne das gesamte Paket zu kennen. Hierdurch lässt sich dieses in möglichen Weiterentwicklungen durch einen Mikroprozessor direkt auf der Hardware interpretieren, ohne große Zwischenspeicher vorhalten zu müssen.

Das Treibermodul ist in Python geschrieben. Die Sprache wurde aus mehreren Gründen gewählt. So sind die Grundpakete betriebssystemunabhängig ausgelegt. Insbesondere existiert mit `python-serial` ein Modul, das den einheitlichen Zugriff auf die serielle Schnittstelle unter *Windows*, *Linux* und *MacOS X* erlaubt. Im Gegensatz zu etwa *rxtx* in *Java* wird dieses aktiv gepflegt und konnte problemlos eingesetzt werden.

Zusätzlich beinhaltet die Kernsprache einen Datentyp zur Verwaltung von Ganzzahlen „beliebiger Länge“. Von diesem wird aufgrund der theoretisch unbeschränkten Breite des *WISHBONE Bus* intensiver Gebrauch gemacht.

Der Zugriff auf den *JTag*-Port wird mit der freien Software *urJTag* [urj11] realisiert. Hierbei handelt es sich um eine plattformübergreifende Konsolenanwendung, die eine Vielzahl an *USB-JTag*-Umsetzern und *LPT-JTag*-Kabeln unterstützt.

[31] Remote Function Call

Im Gegensatz zu verbreiteten *SVF-Playern*[32] erlaubt diese Anwendung die interaktive Manipulation des *Instruction-* und *Data Register*s.

Wenn das Treiberprogramm im *JTag*-Modus betrieben wird, startet es *urJTag* und kommuniziert mit diesem über die `stdin`/`stdout`-Pipes. Aufgrund eines undokumentierten Verhalten des Tools, das diese Interaktion unmöglich macht, ist ein einzeiliger Patch der Software notwendig. Offensichtlich handelt es sich bei dem Umweg über eine menschenlesbare Repräsentation der Befehle um kein effizientes Interface. Eine Optimierung an dieser Stelle – etwa mittels zu entwickelnden Python-*Bindings* – ist erstrebenswert.

Anwendung

Die Anwendung wurde ebenfalls in Python implementiert. Hierfür wurde ein objektorientiertes Abbild aller Funktionen des Logic Analyzers, das einen sehr abstrakten Zugriff gestattet, kreiert. Die Kernkomponente ermöglicht das Auslesen der für die Synthese verwendeten Konfiguration des Analyzers sowie aller Laufzeitparameter. Einzelne Register werden dabei nicht durch ihre Adresse identifiziert, sondern sind entsprechend ihrer Semantik benannt. Die Software setzt die Bezeichner automatisch auf das dynamische Adress-Schema um. Die Bestandteile kombinierter Register lassen sich darüber hinaus getrennt ansprechen.

Zusätzlich zu diesem immer noch recht hardwarenahen Zugriff stehen Methoden zum Herunterladen der aufgezeichneten Daten zur Verfügung. Dabei werden komprimierte Datenströme entpackt und im Falle mehrerer aktiver Bänke den einzelnen Bänken zugeordnet. Weitere Funktionen ermöglichen beispielsweise das Ablegen eines Datenstroms in einer VCD-Datei[33]. Diese Dateien können direkt mittels freier *Waveform-Viewer* visualisiert werden.

Auf Basis dessen wurde eine Vielzahl an Testfällen entwickelt. Diese sind jeweils eigenständig lauffähig und verifizieren u.a. das Triggern auf Werte und Flanken, die Taktteilung, die Kompression und die Eingangsstufe. Dabei wird ein möglichst breites Spektrum an zur Laufzeit veränderbaren Parametern abgedeckt. Dies geschieht teils deterministisch, teils randomisiert.

Ähnlich der eingesetzten *VHDL Unit Tests* kommt dafür ein einheitliches Report-Format zum Einsatz, welches das vollautomatisierte Testen einer Hardware begünstigt. Fast alle Testfälle benötigen den *Testmustergenerator*.

Für den Produktiveinsatz steht eine auf *Tcl/Tk* aufbauende graphische Oberfläche zum Konfigurieren der Trigger und weniger zusätzlicher Parameter zur Verfügung. Nach dem derzeitigen Stand ist jedoch die Nutzung der interaktiven Python-Konsole in Verbindung mit dem o.g. Treibergerüst die leistungsfähigste Lösung.

[32] Serial Vector Format, ein Dateiformat zur Beschreibung von JTag Zugriffsmustern

[33] Value change dump, ein Dateiformat zum Beschreiben zeitlicher Änderungen von Signalen. Die Syntax orientiert sich an Verilog