Bachelorarbeit in Physik von Oliver Freyermuth

Aufbau eines TDC auf einem FPGA–Board

angefertigt im Physikalischen Institut

vorgelegt der Mathematisch–Naturwissenschaftlichen Fakultät der Universität Bonn

Juli 2010

Betreuung: PD Dr. Jörg Pretz

1. Gutachter: PD Dr. Jörg Pretz
2. Gutachter: Prof. Dr. Kai–Thomas Brinkmann

Hiermit bedanke ich mich für die tatkräftige Unterstützung durch meinen Betreuer PD Dr. Jörg Pretz, den Gruppenleiter Prof. F. Klein und Dr. Jürgen Hannappel, ohne welche die Umsetzung einer solchen Bachelorarbeit in diesem Umfang nicht möglich gewesen wäre.

Zusammenfassung

In der Teilchenphysik ist es von essentieller Bedeutung, Zeitabstände zwischen einem Start– und einem Stop–Signal mit hoher Auflösung (im Sub-Nanosekundenbereich) bestimmen zu können. Durch die Entwicklung Feldprogrammierbarer Logikgatteranordnungen (FPGA) bietet sich eine Möglichkeit, kostengünstig und flexibel mit einem rein digitalen Aufbau diesen Anforderungen gerecht zu werden. Im Rahmen der vorliegenden Bachelorarbeit wird ein solcher Aufbau mit notwendigen Zusatzfunktionen beschrieben und auf seine Präzision untersucht. Weiterhin wurde eine Datenauslese über ein Bussystem implementiert, womit es ermöglicht wird, die gemessenen Zeitabstände bis zu einer Auflösung von 1,79 ns zu erfassen und zur Weiterverarbeitung durch das Datenanalysepaket „ROOT“ asynchron auf einem PC abzulegen.

Zentrale Punkte dieser Implementation sind:

- Kostengünstiger Aufbau mit für viele Anwendungszwecke ausreichender Zeitauflösung
- Vorwahl eines Zeitfensters um ein Triggersignal, in dem mehrfach Pulsabstände erfasst werden können (Multi-Hit–Fähigkeit)
- Direkte Anbindung an das Datenanalysepaket „ROOT“, um einen problemlosen Einsatz in teilchenphysikalischen Experimenten zu ermöglichen
- Sicherstellen weitgehender Portabilität (Veränderung von Hardware oder Software soll möglich bleiben)

Im Rahmen der vorliegenden Arbeit konnten diese Punkte erfüllt werden, sodass für einen Einsatz des Aufbaus bei einem Experiment nur noch geringe Anpassungen nötig sind.

Inhaltsverzeichnis

1. Einleitung

Die exakte Messung von Zeitintervallen spielt in vielen Gebieten der Physik und insbesondere in der Teilchenphysik eine große Rolle.

Hier lassen sich beispielsweise über die Messung von Flugzeiten (TOF, „time of flight“) von Teilchen in magnetischen Feldern Eigenschaften wie Impuls oder Ladung eines Teilchens bestimmen (siehe Abbildung 1.1 für einen exemplarischen Aufbau). Dazu ist eine hohe Auflösung nötig, die Messung findet im unteren Nanosekundenbereich oder in der Größenordnung von hunderten Picosekunden statt. Bei einem Detektor kann eine so hohe Auflösung etwa auch dafür genutzt werden, die „ToT“ (Time over Threshold) eines ladungsempfindlichen Verstärkers mit nachgeschaltetem Diskriminator zu messen und so die deponierte Ladung (innerhalb gewisser Genauigkeitsgrenzen) zu bestimmen.

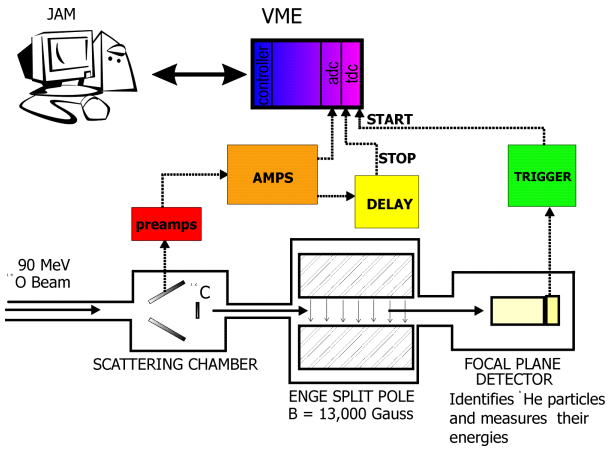

Abbildung 1.1.: Das „Yale Lamp Shade Array“ zur Untersuchung von angeregten ^{22}Mg–Zuständen mit TDC–TOF–Messung (Quelle: [WNS05])

Zur Messung werden sogenannte TIM („time interval meter“) verwendet, welche zur schnelleren Speicherung und Verarbeitung der Messdaten in der Regel digital als TDC („time-to-digital–converter“) ausgeführt sind.

Die Zielsetzung dieser Bachelorarbeit ist es, einen modernen TDC auf Basis eines FPGA aufzubauen. Wünschenswert sind eine Auflösung im einstelligen Nanosekundenbereich, die Fähigkeit, mehrere Hits zu einem Triggersignal zu verarbeiten sowie die Möglichkeit, mehrere Kanäle gleichzeitig aufzunehmen und über einen Datenbus mit einem PC in einer Auslesesoftware für die spätere Verwendung vorzubereiten.

2. Aufbau eines TDC

2.1. Klassischer Aufbau

Der vom Grundprinzip her einfachste Aufbau besteht aus einem analogen Messteil mit anschließender Digitalisierung. Ein sogenannter TAC („time-to-amplitude–converter“) nutzt einen Kondensator, der mit einem festgelegten Strom für das zu messende Zeitintervall – je nach Variante – ge- oder entladen wird. Mit einem ADC („analog-digital–converter“) lässt sich danach die gesammelte Ladung (bzw. die fehlende Ladung über den nötigen Ladestromimpuls) bestimmen und digital ausgeben. Diese ist proportional zur Ladezeit.

Nach der Messung muss der Kondensator also wieder ent– / geladen werden, woraus sich eine nicht unerhebliche Totzeit ergibt. Um dieses Problem zu minimieren, ist es bereits vor der Messung nötig, die Signale zu filtern. Dazu wird die Messung nur in einem Zeitfenster um ein Triggersignal freigegeben, zudem ist über die Kondensatorkapazität die maximale Messzeit, in der sich die Differenz zwischen „Start“– und „Stop“–Signal bewegen darf, begrenzt. Durch eine Logik und eine Verzögerung der Eingangssignale über lange Delay–Kabel (Verzögerungsleitungen) lässt sich so bereits vor der Ladung / Entladung des Kondensators bestimmen, ob die Messung physikalisch interessant ist.

Ein „Dual slope / ramp TDC“ verbessert dieses Verfahren, indem der Kondensator nach dem Ladevorgang mit einem niedrigen, festgelegten Strom entladen wird (beispielsweise $1/100$ oder $1/1000$ des Ladestromes).

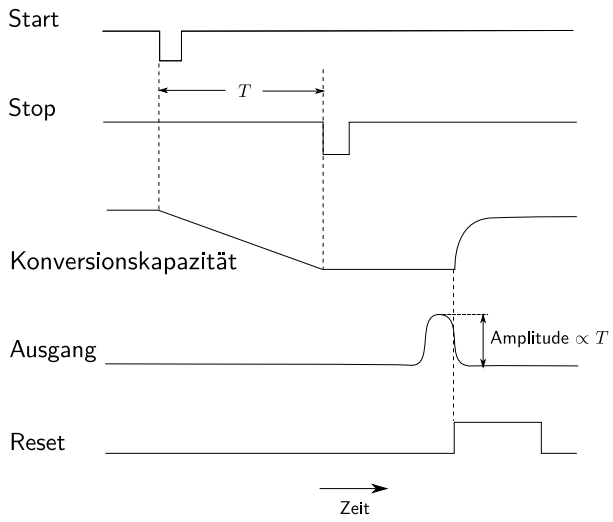

Abbildung 2.1.: Signalschema in einem klassischen TAC

Somit wird effektiv das zu messende Zeitintervall in eine andere Größenordnung gestreckt und konventionell (über Digitalzähler / einen einfacheren TDC–Aufbau) messbar, die Totzeit allerdings auch deutlich höher. Hier kann man beispielsweise durch die wechselweise Nutzung mehrerer TDC–Aufbauten für einen Kanal gegensteuern.

Eine andere Alternative ist es, den Kondensator nur während der Koinzidenz zweier Signale zu entladen, wobei eines der Signale einem bekannten internen Takt entspricht. Damit ist aber beispielsweise keine Unterscheidung möglich, ob die Überschneidung am Anfang oder am Ende des Taktpulses stattfand.

Mit diesen analogen Methoden sind Auflösungen bis zu 1 ps erreichbar, für eine einfache, automatisierte Auslese der Daten mehrerer Kanäle ist es allerdings sinnvoll, Vielkanal–TDCs direkt in Digitaltechnik zu entwickeln. Dabei werden weiterhin Verzögerungsleitungen erheblicher Länge für die Signale eingespart, da diese über einen Trigger, welcher in der Regel erst nach den interessanten Ereignissen anliegt, ausgewählt werden. Insgesamt können so Kosten und Fehleranfälligkeit des Aufbaus reduziert werden.

2.2. Aufbau mit Logikelementen

Ein TDC lässt sich ebenso mit Logikelementen aufbauen. Meist wird eine „Delay Line“ verwendet, also eine Verzögerungsleitung, die das eingeführte Signal trägt. In regelmäßigen Abständen sind an dieser Leitung Flipflops angebracht, welche mit einem gemeinsamen Takt geschaltet werden. Damit wird der innerhalb eines Taktzyklus zurückgelegte Signalweg gespeichert, was direkte Rückschlüsse auf die Laufzeit zulässt. Alternativ können im Aufbau auch Takt und Signal vertauscht werden. Der Nachteil bei dieser Bauart ist, dass die Signallaufzeiten genau bekannt sein und die Flipflops exakt synchron schalten müssen. Auf einem FPGA ist dies ohne feste Platzierung und mehrfache Messreihen nicht realisierbar.

Alternativ dazu kann das Vernier–Verfahren angewendet werden: Ein vom Signal getriggerter Oszillator wird auf Koinzidenzen mit einem internen Referenzoszillator geprüft. Die Auflösung wird hier über die Abweichung der beiden Oszillatorfrequenzen voneinander definiert, bei 1 % Abweichung beträgt die Auflösung in guter Näherung $1/100$ des Produktes der Taktzyklen (vergleiche dazu [Leo87]). Um die Oszillatorfrequenzen stabil halten zu können, wird etwa ein PLL („Phase-locked loop“) verwendet, welcher die Frequenz mit einem langsameren Referenztakt stabilisiert (siehe auch Abschnitt 3.4.1). Auf einem FPGA lassen sich nur begrenzt hohe Takte mit ebenfalls begrenzter Präzision erzeugen, sodass bei diesem Verfahren durch das Jittern des Taktes keine hohe Genauigkeit erreichbar wäre.

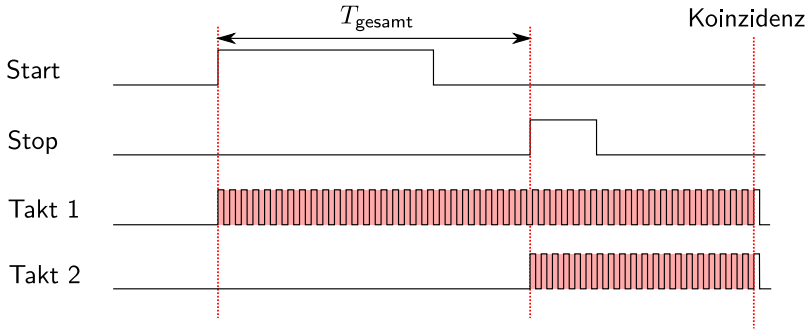

Abbildung 2.2.: Signalschema des Vernier–Verfahrens

Eine andere offensichtlich erscheinende Möglichkeit, welche auch in dieser Bachelorarbeit zum Einsatz kommen wird, ist es, das Signal mit einem hohen Takt abzutasten. Dazu wird die erste Taktperiode, bei der das Eingangssignal eine logische „1“ widerspiegelt, zur Messung herangezogen und startet einen Zähler, der dann bei jedem Takt hochgezählt und beim nächsten erkannten Eingangssignal ausgelesen wird. In der Praxis bereitet es hier Schwierigkeiten, die Schaltzeiten der Flipflops möglichst kurz, den Takt hoch und die Leitungslängen so auszuführen, dass synchrones Schalten möglich ist.

Eine Möglichkeit, diese Problematiken mit einem FPGA zu lösen, wird im Rahmen dieser Bachelorarbeit vorgestellt werden.

3. Vorgaben

Im Folgenden werden die vorgegebenen Randbedingungen (also die zur Verfügung stehende Hardware und Software sowie deren Aufbau und Funktionsweise) vorgestellt.

3.1. Aufbau eines FPGA

Ein FPGA besteht aus Logikgattern, welche über eine Schaltmatrix quasi frei miteinander verbunden werden können. Ein solcher Aufbau wurde früher in den Anfängen der Digitaltechnik mit einzelnen Logikbausteinen verwirklicht, welche auf Basis von Transistoren oder Dioden als Gatter mit bestimmten logischen Funktionen (AND, OR, NOT) angefertigt wurden. Eine Integration in einen integrierten Schaltkreis (IC, „Integrated Circuit“) konnte danach bei Anforderung einer hohen Stückzahl durchgeführt werden.

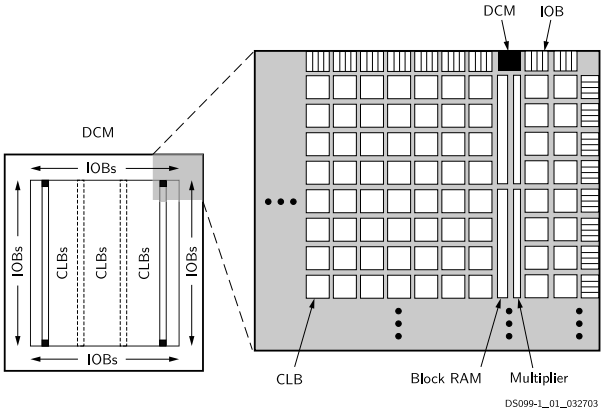

Abbildung 3.1.: Architekturübersicht der Spartan 3–Serie von XILINX® (aus [XILINX®09, S. 4])

Eine Lösung für individuelle Anwendungen bietet bereits ein CPLD („Complex Programmable Logic Device“), welches allerdings eine viel geringere Anzahl von Logikbausteinen im Verhältnis zu den I/O–Ports als ein FPGA besitzt. Damit eignet es sich nicht so sehr für komplexe Datenverarbeitung, sondern eher für ein komplexes Mapping der Eingänge auf die Ausgänge. Ein FPGA wiederum ist der komplexen und parallelen Datenverarbeitung gewachsen. Im Gegensatz zu einem Prozessor, bei welchen die Zahl der Logikgatter gering ist und die Funktionsvielfalt durch eine sequentielle Abarbeitung erreicht wird, ist beim FPGA die Aufgabe vorher festgelegt und wird in eine komplexe, parallel arbeitende Logikschaltung abgebildet. Neuere Ansätze gehen dahin, bei den immer größer werdenden FPGA–Bausteinen einzelne Segmente während des Betriebes umzuprogrammieren (also den Schaltplan zu verändern) und so die Aufgabe dynamisch anzupassen.

In dieser Arbeit wurde ein Spartan 3–FPGA der Baugröße „1500“ mit dem Speed-Grade 5 verwendet. Der Speed-Grade gibt dabei eine Klasse der Mindestgeschwindigkeit der Logikelemente an, die Baugröße bestimmt die Zahl der Blöcke aus Logikgattern, der sogenannten CLBs („Configurable Logic Blocks“). Weiterhin ist je nach Baugröße auch die Bestückung mit Block–RAM und insbesondere auch Ein– und Ausgängen anders (siehe dazu Abbildung 3.1). Bei diesem Modell existieren die Speed-Grades 4 und 5, wobei 5 der schnelleren Version entspricht. Diese ist allerdings nur im Temperaturbereich „Commercial“ erhältlich, was bedeutet, dass die Betriebstemperatur im Bereich von $(0 \dots 85)\,°C$ liegen darf (im anderen Bereich „Industrial“ wären $(-40 \dots 100)\,°C$ möglich).

Entscheidend für die vorliegende Arbeit war die Anordnung der Logikgatter. Diese sind in den CLBs noch einmal in Slices unterteilt, im Falle des Spartan 3 in vier Slices, welche sich aus zwei verschiedenen Typen zusammensetzen, dem Slicetyp „M“ und „L“ (siehe dazu Abbildung 3.2). Diese unterscheiden sich darin, dass mit dem Slice „M“ nicht nur benachbarte Logik, sondern

auch Block–RAM angesprochen werden kann. Dies könnte bei der Untersuchung der Laufzeiten in Abschnitt 6.1 interessant werden.

Der vereinfachte, allerdings seitenfüllende Aufbau eines solchen Slices findet sich in [XILINX®09, S. 23] wieder. Entscheidend ist die Zusammensetzung aus zwei sogenannten LUTs („Look–Up Tables“), welche logische Signale an den Eingängen nach einem programmierbaren Muster auf den Ausgang abbilden und zwei Flipflops, welche ebenso als Latches (Flipflops, welche taktunabhängig angelegte Signale durchleiten oder halten) verwendet werden können. Weiterhin finden sich in einem Slice noch zwei UND– sowie zwei XOR–Gatter und mehrere programmierbare MUX–Bausteine, welche eines von mehreren Eingangssignalen auf einen Ausgang legen können. Zusätzlich sind Inverter vorhanden, um verschiedene Signale einzeln invertieren zu können. Alle diese Bausteine sind dynamisch verschaltbar, eine schnelle Anbindung zu benachbarter Logik (etwa zum Zähleraufbau) wird über eine Carry–Chain ermöglicht.

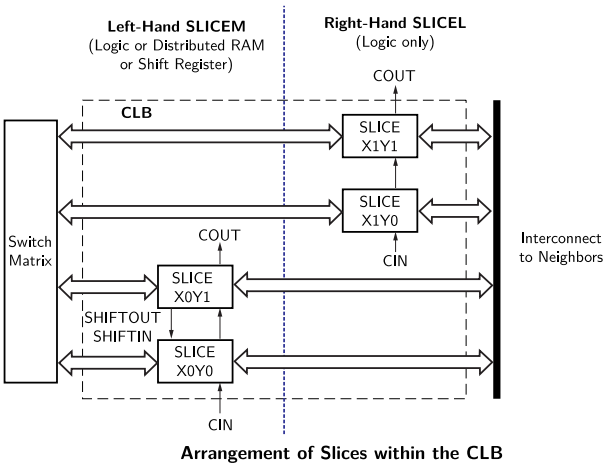

Abbildung 3.2.: Aufbau eines CLB aus 4 Slices (entnommen aus [XILINX®09, S. 22])

3.2. Aufbau des FPGA-Boards

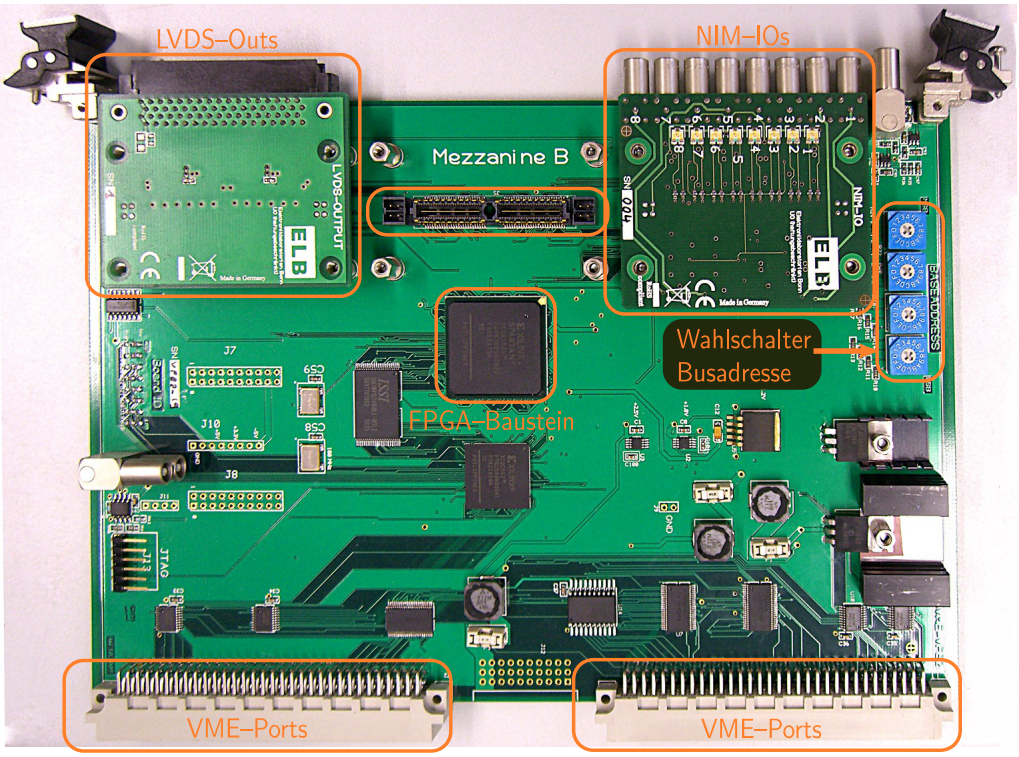

Abbildung 3.3.: Foto des verwendeten Spartan® 3 VME–Boards

Das verwendete Board (siehe Abbildung 3.3) enthält ein FPGA des Typs „Spartan® 3“ der Firma XILINX® und wird über einen VME–Bus mit dem auslesenden PC verbunden. Es bietet Anschluss für bis zu drei Tochterboards (Mezzanines), die für IO–Ports genutzt werden und im

Falle eines LVDS–Mezzanines pro Board 32 Anschlüsse bieten, welche je nach Ausführung als Eingänge, Ausgänge oder wahlweise verwendet werden können. Für den Versuchsaufbau bietet sich die NIM-IO–Karte an, welche 8 NIM–Anschlüsse mit zusätzlich je einer roten und einer grünen LED pro Anschluss bietet. Damit ist zumindest die Funktion des Aufbaus am Board selbst kontrollierbar.

Über die vier auf dem Board vorhandenen Wahlschalter lassen sich die ersten 16 Bit der 32-bittigen Busadresse einstellen, über welche das Board nachher abgefragt werden kann. Die letzten 16 Bit der Adresse stehen im FPGA zur Verfügung und erlauben es, verschiedene Funktionen direkt einer Adressanfrage zuzuordnen. Auf dem Board befinden sich weiterhin noch Quartzgeneratoren, ein externer Speicherbaustein, Spannungsregler sowie weitere zur Funktion wichtige Bauteile, auf die in dieser Arbeit nicht näher eingegangen wird.

3.3. Die Entwicklungsumgebung

Zur Programmierung des FPGA wird von XILINX® eine Entwicklungsumgebung kostenlos zur Verfügung gestellt, das „ISE WebPack™“. Hiermit ist es nicht nur möglich, Programmcode in den verbreitetsten Hardwarebeschreibungssprachen (siehe Abschnitt 3.3.1) über in Abschnitt 3.3.2 genauer beschriebene Schritte in einen auf das FPGA ladbaren Datenstrom zu wandeln, welcher die Logikgatterverschaltung beschreibt, sondern es werden auch viele Fehler frühzeitig abgefangen und Probleme mit Signallaufzeiten gemeldet. Zudem ist eine Simulation des Aufbaus möglich, wenn — ebenfalls in einer Hardwarebeschreibungssprache — die Signale, welche an die Eingänge gelegt werden sollen, beschrieben werden.

3.3.1. Grundlagen der FPGA–Programmierung

Zur Programmierung eines Hardwarebausteins ist eine deutlich andere Sprache als bei herkömmlichem Programmieren nötig. Diese formale Sprache wird als HDL (Hardware Description Language) bezeichnet. Als Industriestandards und weltweit am häufigsten verwendete Sprachen haben sich dabei VHDL und Verilog etabliert, in der vorliegenden Arbeit wird VHDL Verwendung finden.

Das Wichtige bei der Benutzung einer HDL ist es, den nicht mehr sequentiellen Programmablauf nachzuvollziehen. Dazu gibt es bei VHDL zum Einen die Möglichkeit, sehr schaltungsnah einzelne „Bauteile“ zu definieren, Ein– und Ausgänge festzulegen und diese dann zu verbinden. Dies hat den großen Vorteil, dass mehrere identische Funktionen durch eine mehrfache Instantiierung eines einmal erstellten Bausteins gelöst werden können und dieser sehr hardwarenah abgebildet werden kann. Durch diese Kapselung von Funktionen in einzelne Bausteine wird auch eine hohe Modularität erreicht, bei der Programmierung kann das Modul zudem ohne genaue Kenntnis der exakten inneren Funktionsweise weiterverwendet werden, ähnlich wie ein Bauteil beim Aufbau einer Schaltung.

Bei der vorliegenden Problemstellung wird allerdings in der Hauptsache eine weitere Möglichkeit in VHDL genutzt, welche der sequentiellen Programmierung ähnlicher ist, aber im Detail klare Unterschiede aufweist. Hier werden sogenannte Prozesse verwendet, welche bei Pegeländerung definierter Signale (in der Regel wird der Takt benutzt) „ausgeführt“ werden. Die Ausführung geschieht dabei natürlich prinzipbedingt nicht sequentiell, sondern findet parallel statt. Dabei muss gewährleistet sein, dass bis zur nächsten Veränderung des benutzten Taktsignals alle „Befehle“ abgearbeitet sind. Dazu gibt es mehrere Einschränkungen, beispielsweise darf ein Signal nicht in mehreren Prozessen gesetzt, wohl aber in mehreren Prozessen gelesen werden. Eine Kommunikation zwischen asynchronen Prozessen findet also in der Regel, wie auch bei asynchronen Bussystemen, über Signale zur Datenanforderung und Bestätigung des Datenerhalts statt. Dieses Programmierkonzept wird in dieser Arbeit verwendet werden, da so Totzeiten minimiert oder in fast allen Fällen völlig vermieden werden können.

Ein einfacherer und übersichtlicherer Ansatz bildet eine sequentielle Programmierung ab, indem über ein Signal verschiedene Zustände definiert werden, welche dann in einem Prozess zu verschiedenen Aktionen und zur Veränderung des Zustands führen. Vergleichbar ist dies mit dem Aufbau eines Prozessors, wo der „Program–Counter“ auf den durchzuführenden Befehl zeigt und so zur sequentiellen oder teilparallelisierten Abarbeitung führt. Damit wird aber häufig nicht das gesamte FPGA gleichzeitig benutzt, ein Teil des eigentlichen Vorteils der dauerhaften parallelen Abarbeitung geht auf Kosten eines geringeren Entwicklungsaufwands verloren. Bei noch größeren Projekten oder bei von den Eingangssignalen abhängigen Verarbeitungsschritten ist dieser Aufbau einer „Finite–state Machine“ aber durchaus sinnvoll.

3.3.2. Umwandlung des Programmcode in einen Bitstream

Dieser Programmcode in einer HDL nach Wahl muss in einen Schaltplan umgesetzt werden. Dies wird im Wesentlichen durch eine Schaltungssynthese erreicht, welche die durch den Programmcode vorgegebenen Anforderungen in eine Schaltung aus Logikbausteinen umsetzt und je nach Einstellung verschiedene Optimierungen durchführen kann. Hier ist bereits über die Schaltzeiten der verwendbaren Gatter eine Schätzung der maximal erreichbaren Frequenz möglich. Im Detail liegt hier eine NGC–Netzliste vor, welche noch nicht die FPGA–Komponenten selbst, sondern generalisierte Komponenten aus der UNISIM–Bibliothek verwendet (vergleiche dazu [1-C09]). Eine Simulation des Verhaltens des Aufbaus (eine sogenannte „Behavioural–Simulation“) ist aber bereits hier möglich.

Im nächsten Schritt wird ein „Translate“–Prozess durchgeführt. Hierbei wird im Wesentlichen die Komponentenbibliothek gewechselt, nun sind auch genauere Schätzungen zu den Schaltzeiten möglich. Diese Netzliste wird nun im „Map“–Prozess in reale Komponenten des FPGA umgesetzt, also etwa in Flipflops oder Block-RAM–Bausteine.

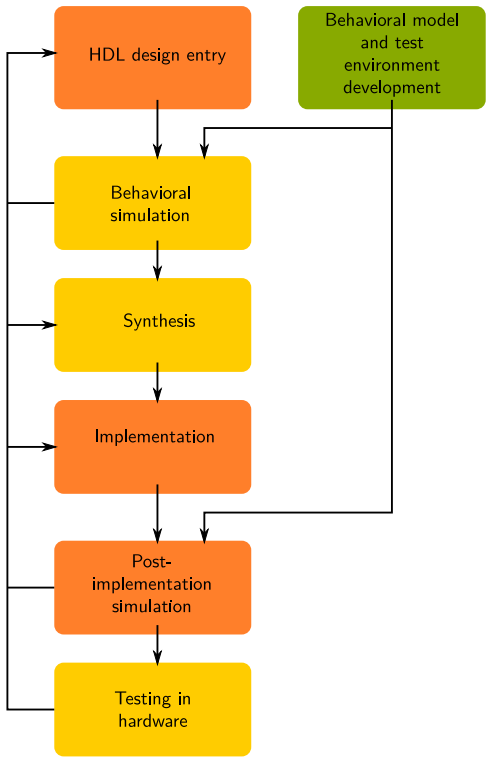

Abbildung 3.4.: Design–Flow für ein FPGA (basierend auf einer Grafik von [1-C09])

Nach diesem Prozess müssen die Bauteile sowie ihre Verbindungen noch örtlich auf dem Chip untergebracht werden (also die gewünschten Verbindungen zwischen ihnen aktiviert werden), hierbei sind Schaltzeiten und Signalwege von besonderer Bedeutung. Zunächst wird in einem Schritt das „Placing“ durchgeführt, welches die Bauteile auf dem Chip platziert, so dass die vorgegebenen Anforderungen an das Timing (entweder durch die Programmierung oder manuell über sogenannte Constraints) nach Möglichkeit eingehalten werden können. Eng damit verbunden ist der nächste Schritt, das sogenannte „Routing“, daher werden beide Schritt auch in der Regel als „Place and Route“ oder kurz „PAR“ zusammengefasst. Beim Routing werden die Signalwege selbst optimiert, das Routing ist als insbesondere von einem guten Placing abhängig. Zur Lösung dieser Prozesse mit einem Rechner gibt es viele verschiedene Ansätze, sodass durch Wahl anderer Strategien unter Umständen einige Laufzeitprobleme vermieden werden können oder erst entstehen.

Im Rahmen dieser Bachelorarbeit wurde durch Einstellungen, welche das Verhalten dieser Prozesse beeinflussen, das Timing optimiert. So wurde es vermieden, bestimmte Bauteile fest zu platzieren und stattdessen ein Satz von Beschränkungen (sogenannte „Constraints“) der erlaubten

Signallaufzeiten verwendet, um vom Chiptyp möglichst unabhängig zu bleiben und einfache Umbauten zu ermöglichen. Weiterhin wurden einige automatische, für den Anwendungsfall kontraproduktive Optimierungen global oder lokal deaktiviert.

Nach dem Routing liegt ein Layout (sogenannter „Floorplan“) für den Chip vor, welches über einen abschließenden Schritt noch mit exakteren Schätzungen der Signallaufzeiten versehen wird und dann begutachtet werden kann. Hier lassen sich bereits viele Details erkennen und es ist möglich, kontraproduktiven Optimierungen gegenzusteuern. Nun ist auch eine Simulation mit dem endgültigen Layout möglich.

Im allerletzten notwendigen Schritt (auf weitere Tools wie etwa eine Schätzung der Leistungsaufnahme wird im Rahmen dieser Arbeit nicht zurückgegriffen) wird dieser Floorplan in einen Bitstream verwandelt. Mit diesem, welcher in das FPGA geladen wird, werden letztendlich die Verbindungen zwischen den Bauteilen geschaltet, es handelt sich also um eine binäre Schaltmatrix.

Diese Schritte sind nach jeder Veränderung des Programmcodes oder der Optimierungen erneut durchzuführen, ein realer Funktionstest ist weiterhin erst nach Übertragung des Bitstreams in das FPGA mit einem zusätzlichen Tool möglich. Im Rahmen der Entwicklungsumgebung werden jedoch alle Einzelschritte automatisch ausgeführt und geben auch (oft hilfreiche) Fehlermeldungen und Warnungen aus.

3.4. Grundlagen für einen FPGA–TDC

Ziel ist es, zunächst eine Zeitauflösung im einstelligen Nanosekundenbereich zu ermöglichen. Die auf dem Board vorhandenen Quartzgeneratoren liefern allerdings nur 1 MHz bzw. 100 MHz. Um höhere Taktraten zu erreichen, bieten viele FPGAs sogenannte DCM („digital clock manager“) an. Diese erlauben es, den Takt über Multiplikatoren und Divisoren zu verändern oder auch Phasenverschiebungen durchzuführen. Ihre grundlegende Funktionsweise ist in Abschnitt 3.4.1 beschrieben.

Die so erreichbare Frequenz ist allerdings ebenfalls begrenzt (detaillierte Informationen in Abschnitt 3.4.1), sodass noch weitere Methoden eingesetzt werden müssen, um die Abtastrate zu erhöhen. Hier gibt es wieder mehrere mögliche Ansätze: Zum Einen ist es möglich, eine Kette aus einer ungeraden Anzahl gekoppelter Inverter aufzubauen, welche mit einer hohen Frequenz oszilliert. Das Problem hierbei ist die Regelung dieser Frequenz: Aufgrund von Veränderungen in Temperatur und Versorgungsspannung bleibt diese nicht konstant. In der Praxis würde man eine Regelung über eine kontrollierte Veränderung der Versorgungsspannung ausführen oder über mehrere einstellbare Verzögerungsketten einen DLL („Delay–locked loop“, siehe Abschnitt 3.4.1) aufbauen.

Im FPGA besteht diese Möglichkeit intern jedoch nur (innerhalb bestimmter Grenzen) mit den DCM (siehe dazu Abschnitt 3.4.1), daher muss also ein anderes Verfahren angewandt werden, um noch höhere Taktraten zu erreichen. Im Rahmen dieser Bachelorarbeit wurde die Nutt–Methode angewendet, welche durch Phasenverschiebung mit einer einzelnen Frequenz eine schnellere Abtastung ermöglicht (siehe Abschnitt 3.4.2).

3.4.1. Funktionsweise eines DCM

Der zentrale Baustein, um Taktsignale für das FPGA zu erzeugen und zu verändern, ist der „Digital Clock Manager“. Dieser nutzt einen „Delay–Locked Loop“ (DLL), also eine steuerbare Kette von Verzögerungen. Neben der Möglichkeit, eine solche Kontrolle über die Veränderung der Betriebsspannung der Verzögerungen auszuführen, ist es auch möglich, diskret Verzögerungselemente einzuschalten oder zu entfernen. Nach diesem Prinzip arbeitet auch ein DCM (siehe Abbildung 3.5 mit einer Implementation von XILINX®). Dieses verwendet mehrere solche

DLL (Detailansicht in Abbildung 3.6), um damit mehrere Phasenverschiebungen sowie eine Multiplikation und / oder eine Division der Frequenz zu ermöglichen.

Bei der Phasenverschiebung eines Taktes wird in der Regel aufgrund der Funktionsweise mit DLL der sogenannte „Duty–Cycle“ verändert, also das Verhältnis zwischen „High“ und „Low“ innerhalb einer Taktperiode. Auch zur Korrektur dieses Verhaltens bietet der DCM eine Möglichkeit.

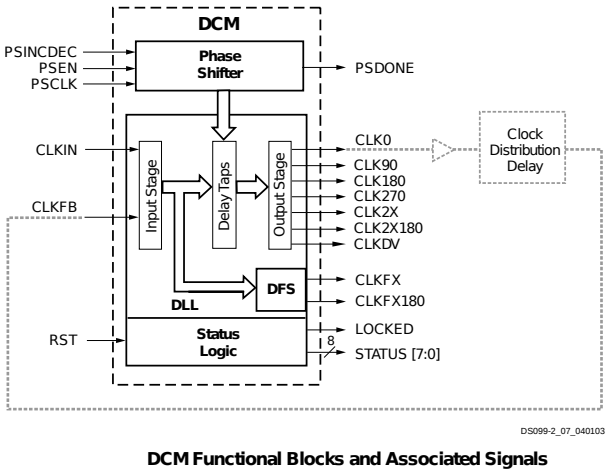

Abbildung 3.5.: Vereinfachter DCM–Aufbau nach XILINX® (entnommen aus [XILINX®09, S. 32])

Interessant für diese Arbeit ist die maximal mit einem DCM erreichbare Taktrate, da dieser Takt durch den Aufbau als DLL bereits stabilisiert ist und den maximal mit Quartzgeneratoren erreichbaren Takt übertrifft. Ein Nachschlagen im Datenblatt (siehe [XILINX®09, S. 87]) bestätigt die Angaben der Software, dass am Ausgang eines DCM im Hochfrequenzmodus maximal eine Frequenz von 280 MHz ausgegeben werden kann. Bei Verwendung des Niederfrequenzmodus und des „2X“–Ausgangs (an dem die Frequenz verdoppelt anliegt) lassen sich bis zu 334 MHz erreichen. Hierbei ist auch ein Zugriff auf den verdoppelten Takt sowie den um 180° phasenverschobenen Takt möglich.

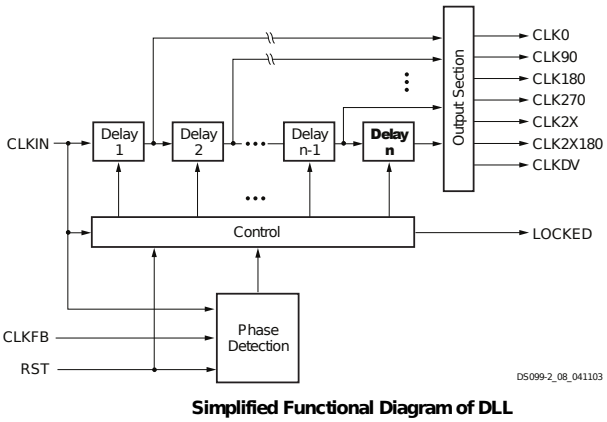

Abbildung 3.6.: Vereinfachter DLL–Aufbau nach XILINX® (entnommen aus [XILINX®09, S. 32])

Aufgrund der deutlich längeren Flipflopschaltzeiten wird in der hier vorgestellten Umsetzung allerdings auf die Möglichkeit zurückgegriffen, auf 4 Phasen eines Taktes zuzugreifen. Auch hier kann nur der Niederfrequenzmodus benutzt werden, es lassen sich bis zu 167 MHz an den Ausgängen erreichen. Da nun allerdings 4 Phasen zur Verfügung stehen, ist dies für den späteren Aufbau vorteilhafter (vergleiche Abschnitt 3.4.2). In der Praxis wurde der Aufbau zunächst mit einer Frequenz von 150 MHz begonnen, da dann aber die Phasensynchronizität nicht gewährleistet zu sein schien (die an den Takt angeschlossenen Flipflops schalteten nicht synchron zu den 4 Phasen) wurde der Takt für den Aufbau später auf 140 MHz reduziert.

3.4.2. Nutt–Methode

Die Nutt–Methode ermöglicht es, bei Begrenzung der maximalen Taktrate ein Signal dennoch mit einem höheren Takt abzutasten. Dieses Verfahren wird beispielsweise in [Kal04] und [KPP87] detailliert beschrieben und untersucht.

Im vorliegenden Fall wird im FPGA die Phasenschiebemöglichkeit eines DCM (siehe Abschnitt 3.4.1) genutzt. Ein zunächst mit einem weiteren DCM möglichst hoch multiplizierter Takt aus einem Quartzgenerator wird in vier Phasen mit gleichem Abstand geschoben (0°, 90°, 180° und 270°), sodass bei reiner Betrachtung aller „Leading Edges“ bzw. „Trailing Edges“ (also der steigenden bzw. fallenden Flanken des Signals) effektiv der vierfache Takt vorliegt. Dabei entspricht die Dauer der Taktpulse allerdings immer noch dem ursprünglichen Takt. Noch weitere Phasen einzusetzen würde den Aufbau aber deutlich verkomplizieren, da ein einzelnes

DCM maximal auf 4 Phasen schieben kann und beim Einsatz weiterer, auf dem Chip örtlich getrennter DCM große Synchronisationsprobleme (für alle Kanäle einzeln) zu bewältigen wären.

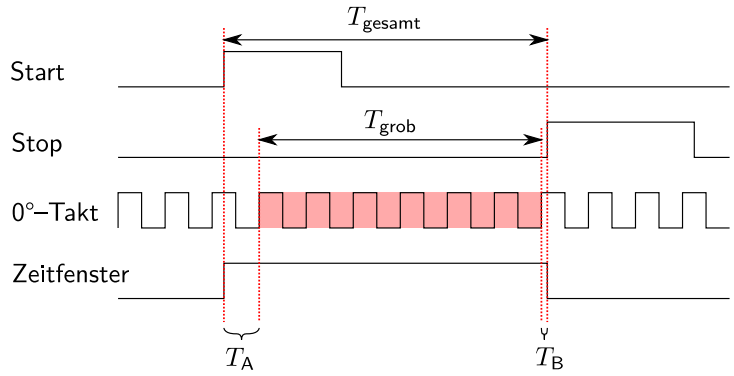

Abbildung 3.7.: Schematische Darstellung des Nutt–Verfahrens

Für die Digitaltechnik, etwa zur Ansteuerung von Flipflops, sind nur die Flanken des Taktes interessant, weshalb es somit möglich ist, den vierfachen Takt voll auszuschöpfen. Wichtig ist allerdings, dass die Taktrate nun durch den maximalen Eingangstakt des DCM im Vierphasenschubbetrieb begrenzt wird. Ein weiteres Problem ist, dass die Flipflops nicht direkt mit diesem hohen Takt verwendet werden können, die Abtastung muss also asynchron erfolgen.

Somit ist es nicht möglich, mit den „Leading Edges“ dieses vierfachen Taktes einfach einen Zähler anzusteuern, welcher über das Eingangssignal gestartet und gestoppt wird. Daher wird zusätzlich zum Zähler, der nur mit dem ursprünglichen, langsamen Takt läuft, für die verschiedenen Phasen eine Signalerkennung eingeschoben, die jeweils eine Korrektur auf den Zählerwert ermöglicht. Dazu betrachten wir das Schema in Abbildung 3.7.

Wir erkennen hier, dass eine hochauflösende Messung nur für die angegebenen Zeiten T_A und T_B benötigt wird, da T_{grob} direkt über den langsamen Intervallzähler bestimmt werden kann. Diese hochauflösende Messung kann für alle Signalflanken analog durchgeführt werden. Dazu wird das Signal von einzelnen Flipflops abgetastet, welche getrennt voneinander mit den 4 Phasen getaktet sind. Somit ergibt sich beispielsweise Abbildung 3.8.

Hier ist markiert, wann die Flipflops in den 4 Phasen das anliegende Signal erkennen können. Die erkannten Werte werden dann bis zum nächsten Taktzyklus in der entsprechenden Phase im zugehörigen Flipflop gespeichert.

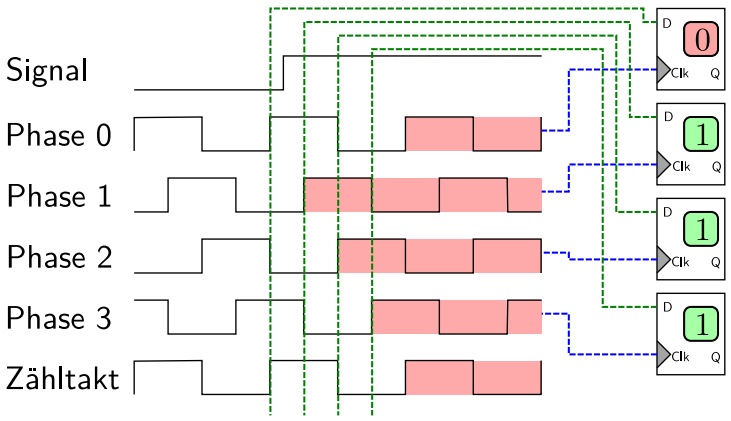

Abbildung 3.8.: Hochauflösende Messung eines Pulses

Damit kann die Datenauswertung im ursprünglichen, langsamen Takt stattfinden, das Nutt–Verfahren ermöglicht es also, bei Vervierfachung des Taktes dennoch mit der langsamen Frequenz zu schalten. Hierbei wurde natürlich ein Problem nicht berücksichtigt: Die Flipflops, welche in den vier einzelnen Phasen geschaltet werden, müssen im ursprünglichen Takt gleichzeitig zur Verfügung stehen. Diese Problematik wird im Folgenden diskutiert und durch Verwendung mehrerer Taktzyklen umgangen werden.

4. Umsetzung

Die Umsetzung des Projektes geschah in mehreren Schritten:

1. „Blinder“ Aufbau eines TDC nach dem Nutt–Verfahren auf einem Standalone–FPGA–Board, Signal manuell über Taster, Ausgabe über LEDs
2. Transfer des Aufbaus auf ein FPGA–Board mit VME-Bus–Anbindung
3. Verwendung eines getrennten FPGA–Boards als Taktgenerator, simple Direktauslese per VME–Bus
4. Zwischenschalten eines FIFO („First-in – First-Out“) als Zwischenspeicher
5. komplette Umstrukturierung in einen geschlossenen Block für einen Kanal, dieser mit Schieberegister und FIFO für Zeitfenstervorwahl und Zwischenspeicherung

Aus Platzgründen werden nur der endgültige Aufbau des TDC–Bausteins sowie die aus den vorher gefundenen Problemen gewonnenen Erfahrungen beschrieben.

4.1. Aufbau des TDC

Wie in Abbildung 3.8 bereits erkennbar, wird das Eingangssignal zunächst auf 4 Flipflops gelegt, welche mit den 4 Taktphasen geschaltet werden, um das Nutt–Verfahren umzusetzen. Von entscheidender Bedeutung ist hierbei (dies war im „blinden“ Aufbau ohne echte Signale nicht erkennbar), dass die Laufzeiten zu den Eingängen der vier Flipflops identisch sein müssen.

Problem: Platzierung der Flipflops

Der einfachste Weg ist hier, die 4 Flipflops am Eingang fest zu platzieren. Damit wäre aber gleichzeitig der Aufbau fest auf einen Chiptyp beschränkt, außerdem müsste dieser Aufwand für jeden Kanal einzeln betrieben werden. Um diese Probleme zu umgehen, wird die Möglichkeit der XILINX®–Software benutzt, eine sogenannte „Constraint“ einzurichten, welche allerdings nur eine maximale Signallaufzeit festlegen kann. Wird diese aber auf den kleinstmöglichen Wert gesetzt, sind Placer und Router gezwungen, die Flipflops so nah wie möglich am Eingang zu platzieren. Damit sind immer noch Laufzeitunterschiede möglich, diese bewegen sich jedoch im Bereich von 10 ps (vergleiche dazu Abschnitt 6.1) und sind damit zunächst kaum relevant. Eine leichte Asymmetrie der Ergebnisse ist aus diesem Grund dennoch zu erwarten, Vorschläge zur Korrektur (sowie eine Vermessung der Asymmetrie) werden in Abschnitt 6.2 diskutiert. Wichtig ist auch, hier die Optimierung des Aufbaus durch die XILINX®–Software einzuschränken: Um die Laufzeiten zu minimieren, wird im Normalfall ein sogenannter IO–Buffer im Eingangsbaustein platziert, welcher das Signal (schneller als die Schaltzeit eines Flipflops auf dem Chip) zwischenpuffert, allerdings — je nach Einstellung — auch die Flipflops ersetzen kann. Da maximal 3 Flipflops auf diese Weise in den Eingang gelegt werden können, würde dies in jedem Fall zu einer großen Asymmetrie führen — daher ist es sinnvoll, die Flipflops direkt an den Eingang anzuschließen oder alternativ nur einen IO–Buffer zu verwenden, an den alle Flipflops angeschlossen sind.

An dieser Stelle ist auch sehr gut der Unterschied zum Aufbau mit Delay–Line zu erkennen: In diesem Fall wären die Laufzeiten zu den Flipflops unterschiedlich, während alle mit dem gleichen Takt angesteuert würden. Dies lässt sich bei neueren FPGAs durch das Setzen von IO–Delays verwirklichen, welche das Eingangssignal um eine einstellbare Zeitdauer verzögern. Damit wäre auch eine höhere Auflösung möglich, da der Aufbau nicht mehr an 4 Phasen gebunden ist.

Bei der höheren Auflösung ist eine Kalibration der Laufzeiten für jeden Kanal allerdings noch entscheidender, sodass hier ein größerer Aufwand betrieben werden müsste.

Wurde das Eingangssignal zunächst in den 4 einzelnen Phasen abgegriffen, steht dieser Wert (bei entsprechend kurzem Signalweg zum nächsten Flipflop) weniger als 1/2–Taktzyklus des ursprünglichen Taktes (also der nullten Phase) später am Eingang des nachfolgenden Flipflops zur Verfügung (genaueres im Datenblatt, siehe [XILINX®09, S. 82], 1,90 ns für Setup und Clock-to-Output–Time).

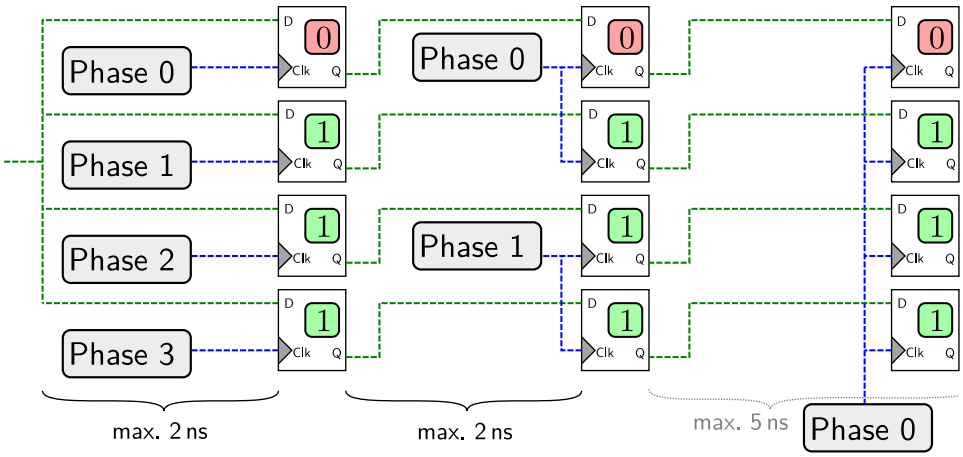

Abbildung 4.1.: Synchronisieren des Bitmusters

Daher ist es möglich, die Inhalte der Flipflops in einen anderen Takt „umzukopieren“. Wie in Abbildung 4.1 zu sehen, wird der 1-Bit–Wert von Flipflop 1 in den nächsten Takt mit Phase 0 kopiert, der 1-Bit–Wert aus Phase 1 wird gleichzeitig ebenfalls in Phase 0 in ein anderes Flipflop geschrieben. Analog dazu werden die Inhalte aus Phase 2 und 3 in Phase 1 umkopiert, sodass die Regel, dass mindestens ein halber Takt zum Durchschalten des Signals benötigt wird, nicht verletzt ist. Dies wird auch durch die XILINX®–Software kontrolliert. Gleichzeitig muss aber noch beachtet werden, dass die Signale selbst auch tatsächlich in dieser Zeit an den Zwischenspeicherflipflops anliegen. Daher muss auch hier wieder ein „Constraint“ gesetzt werden, welches die Laufzeiten zwischen den beiden Flipflopgruppen synchron hält.

Aufgabe: Synchronisieren der Phasen

Im auf diesen Kopierprozess folgenden Takt in Phase 0 können nun alle vier Flipflops gleichzeitig abgefragt werden. Während des gesamten Ablaufs waren die ersten 4 Flipflops direkt wieder für eine neue Messung bereit, sodass es durch den Kopierprozess zunächst keine Totzeit gibt.

Eine Totzeit entsteht allerdings im nächsten Schritt: Hier findet die Auswertung der „getroffenen“ Phasen statt, es wird also erkannt, in welcher Phase das Signal zuerst angelegen hat. Dabei werden folgende Bitmuster zugeordnet:

Bitmuster	Zuordnung
0000	kein Signal
1000	Signal in Phase 4
1100	Signal in Phase 3
1110	Signal in Phase 2
1111	Signal in Phase 1 (oder: Signal liegt noch an)

Die Muster sind hier genau spiegelverkehrt zur eigentlichen Erwartung, was daran liegt, dass das MSB („most significant bit“), also die 4. Phase, vorne liegt: Das Bitmuster ist also in der Phasenreihenfolge 4321 angeordnet. Aus diesen vorgegebenen Mustern ergibt sich, dass das zu

vermessende Signal mindestens einen Zyklus des Referenztaktes lang sein muss, also mindestens 7,14 ns. Dies ist für physikalische Ereignisse schon durch die Totzeiten der Detektoren bzw. den Einsatz von Diskriminatoren, die ein Ausgangssignal fester und meist auch einstellbarer Länge erzeugen, gewährleistet.

Wichtig für den weiteren Teil des Aufbaus ist, dass aus diesem Kernbaustein, dem eigentlichen TDC, im Falle eines erkannten Musters ein „OK“ ausgegeben wird, im Falle eines leeren Signals kein „OK“ und im Falle eines abweichenden Musters ein „ERROR“, mit welchem sich beispielsweise die Trailing–Edge (oder aber starke Asynchronitäten der Phasen / Flipflops am Eingang) erkennen lassen.

4.2. Aufbau eines Auslesekanals

Die gemessenen Zeiten müssen nun online, also auf dem FPGA, weiterverarbeitet und gepuffert werden, da eine direkte Auslese in Echtzeit nicht möglich ist (sonst könnte die Messung auch direkt mit dem PC durchgeführt werden). Wünschenswert sind daher folgende Funktionen und Eigenschaften:

Trigger: Verarbeitung des Triggersignals.

Zeitfenster: um den Trigger. Es werden nur Hits innerhalb des Zeitfensters verwendet (Filterung)

Pufferung: Die Daten müssen zur asynchronen Auslese zwischengespeichert werden.

Aufgabe: Datenpufferung

Zur Zwischenspeicherung vor der Auslese kann ein FIFO verwendet werden. Diese Abkürzung steht für „First-In – First-Out“ und beschreibt damit direkt die Funktion: Die synchron zu einem Schreibtakt eingegebenen Werte können sofort nach dem Einschreiben in identischer Reihenfolge synchron zu einem Lesetakt ausgegeben werden. Diese getrennte Taktung ist allerdings eine Besonderheit der Implementation von XILINX®, die in der Entwicklungsumgebung direkt eingebunden werden kann. Im Normalfall gibt es nur Gate–Leitungen, welche den Schreib– bzw. Leseprozess aktivieren. Dies ist hier zusätzlich möglich. Zu beachten ist allerdings, dass bei Überfüllen des FIFO keine weiteren Daten angenommen werden.

Eine weitere benutzte Funktion der XILINX®–Implementation ist die „First–Word Fall–Through“–Funktion. Dies bedeutet, dass stets der bei der nächsten Leseanfrage auszugebende Wert abrufbar ist, sodass nicht erst die Gateleitung zum Lesen aktiviert und einen Taktzyklus auf Daten gewartet werden muss. Dies funktioniert natürlich beim Schreiben von Daten nicht, hier müssen also Gate und Daten einen Taktzyklus lang anliegen.

Weiterhin wird das Feature benutzt, mit asymmetrischen Datenbreiten in das FIFO zu schreiben und aus diesem zu lesen, da der VME–Bus, über den später die Datenauslese läuft, 32 Bit Breite bietet, die Messdaten aus dem TDC allerdings (wie in Abschnitt 5.1 beschrieben wird) nur 16 Bit breit sind. Damit wird automatisch eine Ausnutzung der vollen Busbreite möglich.

Mit Einsatz des FIFO ist also das Problem der asynchronen Auslese durch den PC lösbar. Um nun allerdings noch zeitlich vor dem Triggersignal liegende Daten erhalten zu können, müssen dauerhaft Messungen stattfinden und in einen Speicher geschrieben werden, welcher die ältesten Messungen automatisch verwirft. Diese Funktion, die im Grunde einer Verzögerung der Daten entspricht, ist mit einem Schieberegister umgesetzt worden. Dieses ist in der Datenbreite des TDC ausgelegt und schiebt mit jedem Taktzyklus die enthaltenen Daten eine Ebene tiefer, bis diese hinten aus dem Schieberegister „herausfallen“. Über ein Gate, welches dem „OK“–Ausgang des TDC–Bausteins entspricht, wird sichergestellt, dass nur gültige Ereignisse in das Schieberegister geschrieben werden.

Aufgabe: Datenverzögerung / Zeitfenster

Das Triggersignal löst nun ein Umschreiben der gewünschten Events in das FIFO aus. Dazu müssen mehrere Probleme umgangen werden: Zum Einen muss das Schieberegister nun mit jedem Taktzyklus weitergeschoben werden, da ansonsten für das Herausschreiben der Daten auf

genügend gültige Events gewartet werden müsste. Zudem ist es zur Wahl des Zeitfensters nötig, den Zeitinhalt des Schieberegisters zu kennen. Dazu wird synchron zum Takt des Registers ein Differenzierer / Addierer (im Folgenden „Time–Accounter“ genannt) angebracht, welcher stets den auslaufenden Wert von seinem Zustand abzieht und den einlaufenden Wert addiert. Berücksichtigt wird hier nur der Stand des Intervallzählers, welcher mit dem Referenztakt läuft (da die Phasensegmente erst bei der Auslese differenziert werden und diese Auflösung für das Zeitfenster genügt). Dies bedeutet, dass bei Einschieben einer (16 Bit breiten) 0, die in der Realität kein Messergebnis sein kann, auf den Time–Accounter keine Wirkung zeigt und somit zum aktiven „Auffüllen“ des Schieberegisters benutzt werden kann.

Liegt ein Triggersignal an, so wird der Inhalt des Time–Accounters in einen Subtrahierer kopiert. Dieser zieht jeden aus dem Schieberegister auslaufenden Wert von diesem Startwert ab und vergleicht den aktuellen Wert mit dem gewünschten Zeitfenster vor dem Triggersignal. Wird dieser Zeitfensterwert unterschritten, so müssen die aus dem Schieberegister erhaltenen Daten in den Ausgabe–FIFO gespeichert werden, das „write–enable“ des FIFO wird also mit dem Komparatorausgang angesteuert.

Bis zum Zeitpunkt T_0, wenn also der Trigger aus dem Schieberegister ausläuft, ist garantiert, dass aus dem Schieberegister nur gültige Messdaten (oder „0“, da das Schieberegister zu Beginn auf einen leeren Zustand gesetzt wird) in den FIFO gespeichert werden. Dabei werden alle Werte aus dem Schieberegister, welche „0“ entsprechen, vor dem Speichern in den FIFO verworfen.

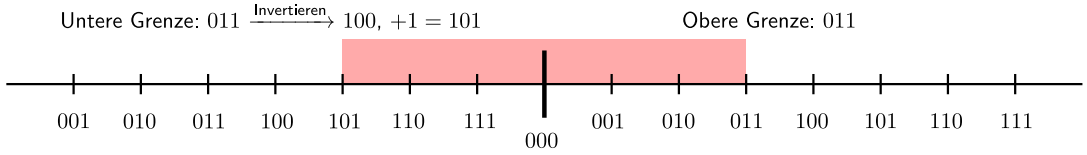

Abbildung 4.2.: Das Zweierkomplement im Zahlenstrahl (Beispiel mit 3 Bit)

Weiterhin muss berücksichtigt werden, dass zum Zeitpunkt T_0 im Subtrahierer ein „Underflow“ entsteht, da bei einer Differenzierung mit Werten hinter dem Trigger der Wert im Subtrahierer negativ würde. Hier handelt es sich allerdings um einen binären Wert, weshalb der Wert nach dem „Underflow“ als Zweierkomplement interpretiert werden kann. Der Komparator vergleicht nun mit einem anderen Wert, nämlich mit dem invertierten Binärwert des Zeitfensters (zu dem 1 addiert wurde) nach dem Trigger. Veranschaulichen lässt sich dies bei Betrachtung des Zahlenstrahls in Abbildung 4.2.

Hier wurden zwei Grenzen vorgegeben, in diesem Fall symmetrisch. Die untere Grenze wird in eine negative Zweierkomplementzahl umgewandelt und bildet damit direkt die nötige untere Grenze. Am Zahlenstrahl wird noch ein Problem deutlich, welches hier aber nicht zum Tragen kommt: Bei Verwendung des Zweierkomplements verringert sich natürlich der Maximalwert der positiven und negativen Zahlen, was im Zahlenstrahl an den identischen Zahlen im „negativen“ und „positiven“ Bereich erkennbar ist. Da allerdings zusätzlich das Vorzeichen im FPGA als „Vor-Trigger“– und „Nach-Trigger“–Zustand abgelegt wird, ist das Vorzeichen stets klar definiert.

Es bestehen allerdings noch zwei weitere Probleme. Zum Einen werden nur gültige Daten im Schieberegister abgelegt, was bedeuten würde, dass das aktuelle Event erst nach dem nächsten Event (oder sogar mehreren Events) vollständig im FIFO vorhanden wäre. Um dies zu umgehen, wird das Gate des Schieberegisters während des Zeitfensters dauerhaft aktiviert, und falls der TDC keine gültigen Messdaten ausgibt, wird eine „0“ (mit 16 Bit Breite) in das Schieberegister eingeschoben (welche die Zeitmessung nicht beeinträchtigt und nicht im FIFO abgelegt wird).

Das zweite Problem besteht darin, dass unbegrenzt lange im Zeitfenster nach dem Trigger auf die nächste gültige Messung gewartet wird. Um dies zu umgehen, wird im Subtrahierer zusätzlich eine „1“ abgezogen, was genau einem Taktzyklus des Referenztaktes entspricht. Dies bedeutet, dass das Nach-Trigger–Zeitfenster doppelt so groß wie gewünscht eingestellt werden

muss und im schlimmsten Fall der Zeitpunkt bis zum Abschließen des Umschreibens in den FIFO ebenfalls der doppelten Zeitfensterlänge entspricht (wenn gar keine Hits im Fenster vorliegen). Weiterhin wird damit aber automatisch das Fenster bis zur doppelten Länge vergrößert, falls keine Hits gefunden werden.Dieser gesamte Aufbau ist in Abbildung 4.3 schematisch dargestellt.

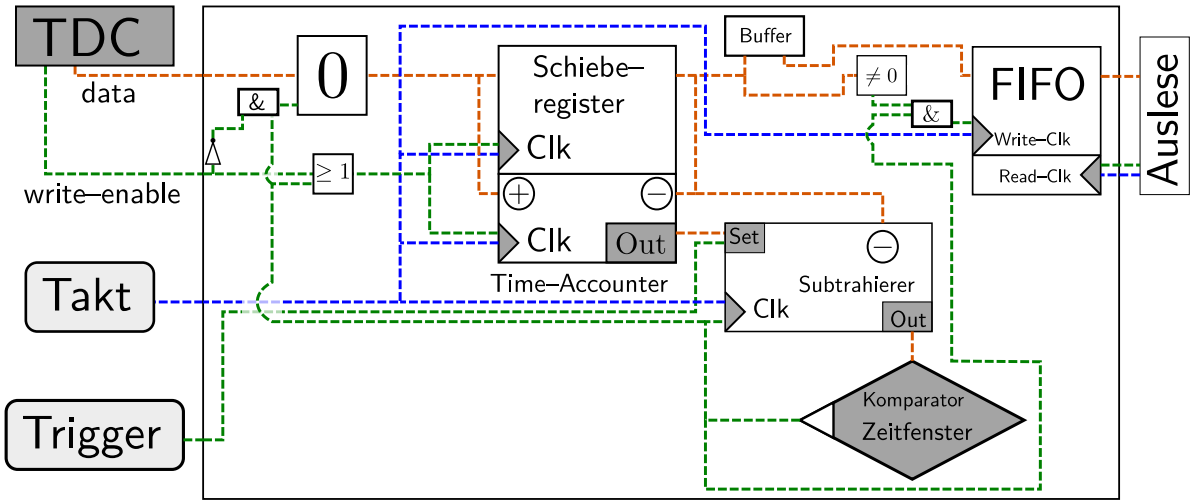

Abbildung 4.3.: Gesamtaufbau der Online–Datenverarbeitung

Hierbei fällt noch als weiteres Detail der Puffer hinter dem Schieberegister auf. Dieser erweitert das Schieberegister um eine Ebene und ermöglicht es gleichzeitig, auf die letzten beiden Werte zuzugreifen. Dies ist nötig, da „write–enable“ und die gültigen Daten mindestens einen Taktzyklus lang am Eingang des FIFO anliegen müssen, „write–enable“ allerdings erst gesetzt werden kann, wenn der Wert am Ausgang des Schieberegisters auf die Bedingung „$\neq 0$“ geprüft wurde.

Insgesamt ist es mit diesem Aufbau nun möglich, die aus dem kontinuierlich messenden TDC kommenden Daten zu filtern, den Trigger zu markieren und eine asynchrone Auslese bei gleichzeitiger Nutzung der vollen Busbreite zu ermöglichen.

4.3. Zusammenfassung mehrerer Kanäle

Zum Einsatz mehrerer Kanäle ist es besonders wichtig, dass der Aufbau möglichst wenig Platz auf dem Chip belegt. Dazu wurde bei Verwendung der XILINX®–Implementationen des Schieberegisters und des FIFO darauf geachtet, diese im Block–RAM des FPGA ablegen zu lassen. Somit werden Slices auf dem Chip sowie insbesondere Flipflops eingespart und das ansonsten ungenutzte Block–RAM kann sinnvoll verwendet werden.

Zum Aufbau einer Auslese mehrerer Kanäle wurde bisher nur ein Konzept erarbeitet, welches allerdings aus Zeitgründen noch nicht umgesetzt wurde (eine Umsetzung wird allerdings nach Fertigstellung dieser schriftlichen Arbeit angestrebt). Geplant ist, den bisherigen Aufbau komplett beizubehalten und hinter den FIFOs der einzelnen Kanäle ein großes Ausgabe–FIFO einzusetzen, welches die Daten der Kanäle wechselweise abfragt und zusammenstellt. Damit können die einzelnen Kanal–FIFOs deutlich kleiner ausgeführt werden, der Auslesesoftware kann bei der Leseanfrage als Antwort direkt der als Nächstes folgende Kanal mitgeteilt werden und dann schließlich der Inhalt des großen FIFO, wieder mit der Breite von zwei Hits. Die Synchronisierung der Kanäle selbst muss beim Aufbau bzw. danach in der Software durchgeführt werden, da weder im Board–Layout noch bei den angeschlossenen Leitungen von synchronen Timings ausgegangen werden kann.

5. Datenauslese

Im Rahmen dieser Arbeit wird auch die Auslese der Daten behandelt. Hier werden die noch fehlenden Bestandteile der Datenverarbeitung, welche nicht online geschehen müssen (Differenzierung der getroffenen Segmente, Berechnung der absoluten Zeiten) sowie die Speicherung der Daten selbst durchgeführt.

5.1. Anbindung an den VME–Bus

Als Grundlage für die Anbindung an den VME–Bus wird die schon vorhandene Implementation eines VME–Interface des Synchronisationsclients des B1–Experimentes benutzt, der bereits in VHDL auf einem FPGA umgesetzt wurde.

Diese Implementation ermöglicht es, über vorher im FPGA festgelegte Register auf Anfragen des PC zu antworten oder auch Daten entgegenzunehmen. Zur Übertragung der Messwerte wird folgende die Datenstruktur in Abbildung 5.1 verwendet.

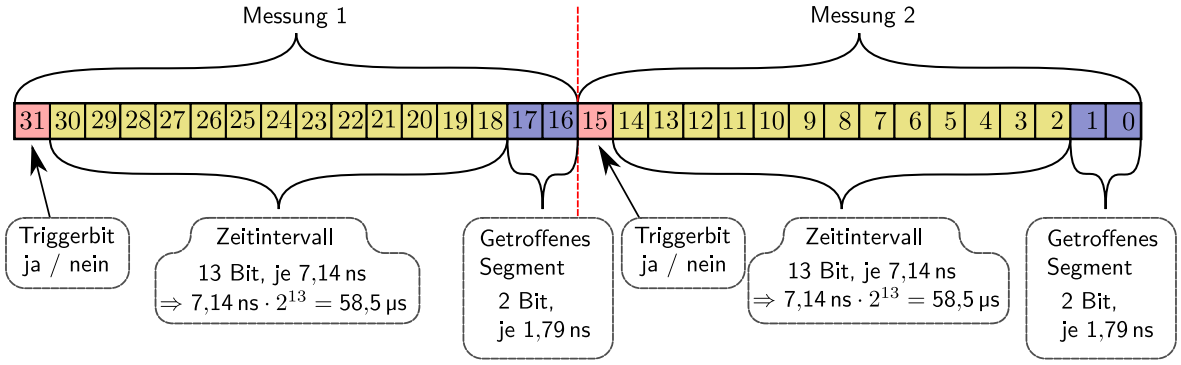

Abbildung 5.1.: Verwendetes Datenformat für die Übertragung

Dort ist ebenso zu erkennen, wie groß der zeitliche Abstand zwischen zwei benachbarten Hits maximal sein darf, ohne dass es zu einem Überlauf kommt. Dies ist hier nur durch die Datenbreite der Messdaten beschränkt, ließe sich also für den Fall, dass ein größerer Maximalabstand benötigt wird, mit wenig Aufwand erweitern, für teilchenphysikalische Experimente reicht eine Maximalzeit in der Größenordnung von 10 µs jedoch in der Regel völlig aus.

Wichtig ist weiterhin, sich klarzumachen, dass der VME–Bus ein asynchrones Bussystem ist. Dies ist für den vorliegenden Fall sehr günstig, da das Timing des Übertragungsprotokolls damit passend gestaltet werden kann und letztendlich durch die maximale Geschwindigkeit der Datenanfragen / Empfangsbestätigungen des PC bestimmt wird, da das FPGA bei vorhandenen Daten im Ausgabepuffer deutlich schneller reagieren kann.

5.2. Auslesesoftware

Die Auslesesoftware muss mit den ausgelesenen Daten in einer Offline–Datenverarbeitung nun Folgendes leisten:

1. Auftrennen der einzelnen Events

2. Differenzieren der Messdaten (die letzten 2 Bit enthalten bisher nur eine Trefferinformation und beschreiben keine Relativzeit)

3. Ausrichten der Daten am Trigger und Errechnen von Absolutzeiten

4. Ausgabe in einem für die Weiterverarbeitung günstigen Datenformat (hier: ROOT–Tree)

Zunächst werden die Messdaten einzelnen Events zugeordnet. Dazu ist es bereits im FPGA nötig, nach dem Event eine Art „Stoppsignatur“ einzufügen, ähnlich dem „Stoppbit“ bei vielen seriellen Protokollen, denn auch hier werden die Daten letztendlich seriell übertragen, während die Datenrate gleichzeitig durch parallele Übertragung zweier Messwerte gesteigert wird. Dazu erscheint es zunächst logisch, einfach einen „0“–Wert mit 16 Bit Breite in den Ausgabe–FIFO bei Schließung des Zeitfensters einzuschieben. Das Problem dabei ist, dass eine 32-Bit–Auslese des FIFO nur möglich ist, wenn zwei 16-Bit–Werte eingeschoben wurden, in etwa 50 % der Fälle kommt es also dazu, dass das 16 Bit breite Stoppsignal in den ersten 16 Bit des FIFO landet und damit erst eine Ausgabe des Stoppsignals möglich wäre, wenn ein neues gültiges Event gemessen wurde. Um das zu umgehen, werden zwei Stoppsignale eingeschoben, welche von der Auslesesoftware dann zur Auftrennung der Events genutzt werden können.

Events trennen

Innerhalb eines solchen Blocks aus mehreren Hits befindet sich nun ein überflüssiger Wert, da bei der Differenzierung Relativzeiten berechnet werden, aber bei jedem Hit ein einzelner Eintrag erzeugt wird. Aus dem ersten Messwert im Zeitfenster wird also nur das getroffene Segment verwendet, welches vom zweiten Wert abgezogen wird. Dieser enthält bereits die Differenzzeit zum vorhergehenden Hit, damit also genau die gewünschte Relativzeit, und muss nur um den Segmentfaktor korrigiert werden, was durch diese Differenzierung geschieht. Dieses Vorgehen ist für alle folgenden Hits identisch, sodass die gewünschten Relativzeiten nun sehr leicht bestimmt werden können.

Differenzierung

Die Ergebnisse (für ein Event) werden dabei zunächst in einem Array abgelegt. Zur Bestimmung des absoluten Zeitabstands zum Triggersignal, welches durch das oberste Bit eindeutig identifiziert werden kann, wird nun dem ersten Wert im Array die mit (-1) multiplizierte Summe der Relativzeiten bis zum Trigger zugeordnet. Auf diese Absolutzeit wird bei der Ausgabe der Daten nun nur noch die Relativzeit der einzelnen Hits addiert.

Absolutzeiten

Die Ausgabe geschieht direkt in einen ROOT–Tree, welcher in den in Tabelle 5.1 angegebenen Branches befüllt wird.

Tree	TDC	tdc measured data	
Br	0	event	event/i
Br	1	hit	hit/i
Br	2	trigger	trigger/i
Br	3	reltime	reltime/d
Br	4	abstime	abstime/D
Br	5	raw	rawdata/i

Tabelle 5.1.: Branches des ROOT–Trees

Damit lassen sich die Rohdaten noch kontrollieren, um etwa eine Betrachtung der Asymmetrie durchzuführen (durch Abtrennen der zwei LSB („Least Significant Bit“)). Weiterhin stehen die differenzierten Relativzeiten sowie die Absolutzeiten, ausgerichtet am Triggersignal, zur Verfügung.

6. Analyse

Nach Vervollständigung des Aufbaus müssen dessen Funktion sowie die erreichte Präzision überprüft werden. Dazu werden im Folgenden Messungen durchgeführt und Verbesserungsmöglichkeiten diskutiert. Abschließend erfolgt eine Bewertung der Ergebnisse.

6.1. Erreichte Präzision

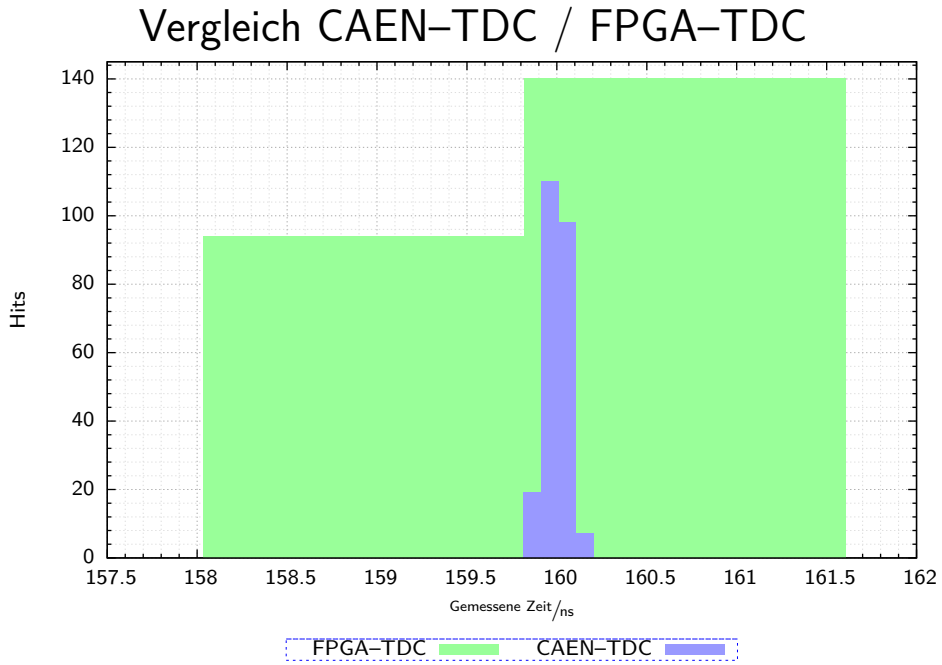

Abbildung 6.1.: Analyse der Präzision

Um die mit dem Aufbau erreichbare Präzision zu bestimmen, wird zunächst ein einfacher Vergleich bei der Messung konstanter Periodendauern mit den Ergebnissen eines kommerziellen TDC der Firma CAEN, Modell V1190A, verglichen. Dazu wird das gleiche Triggersignal und ein identisch großes Zeitfenster benutzt, zur Kontrolle wird die volle Auflösung des V1190A von 100 ps herangezogen.

Als Quelle für die Signale und den Trigger wird ein weiteres FPGA–Board der gleichen Bauart verwendet, sodass bei der Messung mit dem CAEN–TDC ebenfalls eine qualitative Aussage über die Taktstabilität eines generierten Taktes möglich sein könnte. Betrachten wir dazu eine histogrammierte Darstellung der Messergebnisse in Abbildung 6.1.

Im Rahmen der möglichen Auflösung des eigenen Aufbaus ist also der Idealfall erreicht: Es werden nur die beiden Bins um den erwarteten Pulsabstand von 160 ns befüllt, während der Schwerpunkt sich bei der erwarteten Zeit befindet.

Bei Betrachtung der Daten aus dem CAEN–TDC fällt zunächst die deutlich höhere Zeitauflösung auf. Weiterhin ist hier eine Streuung der Periodenlängen sichtbar, was höchstwahrscheinlich auf einen Jitter des takterzeugenden FPGAs zurückzuführen ist.

In diesem wurde der Takt mit einem DCM aus einem Quartzgenerator mit 100 MHz erzeugt, so wie dies auch beim Referenztakt im TDC–Aufbau des FPGA umgesetzt wurde. Hier ist also ein Jittern des Taktes zu erkennen, welches sich allerdings deutlich unterhalb der Zeitauflösung des eigenen Aufbaus befindet. Dies kann damit aber sehr wohl zu einer Asymmetrie der vier verschobenen Takte zueinander führen, welche durch eine Messung mit zufälligen Zeitperioden bestimmt werden kann. Um den Aufbau gleichzeitig einfach zu halten und weiterhin ein Triggersignal zu nutzen, wird der Ausgang eines Diskriminators an einem Szintillator (welcher zeitlich zufällig verteilte Höhenstrahlung misst) als Trigger verwendet, während als Signal weiterhin die 160 ns–periodischen Pulse des anderen FPGA verwendet werden. Sinnvollerweise werden zur Auswertung der Asymmetrie dann nur die Werte verwendet, die als Trigger markiert sind. Um die Phasenzuordnung zu erreichen, werden die unteren beiden Bits der Rohdaten abgetrennt und die Hits danach histogrammiert. Damit ergibt sich Abbildung 6.3.

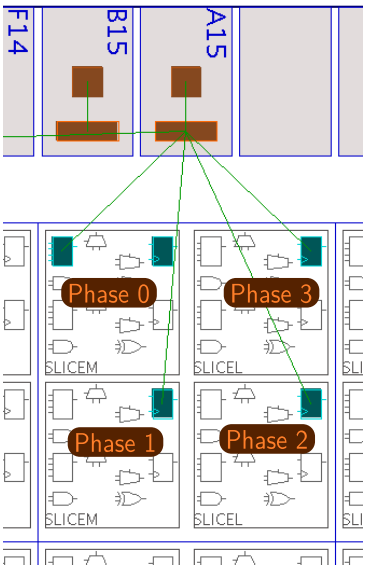

Abbildung 6.2.: Anordnung der Flipflops am Signaleingang. Screenshot aus PlanAhead™.

Hier ist erkennbar, dass die Phasen 1 und 2 gegenüber 0 und 3 benachteiligt sind, was im Umkehrschluss bedeutet, dass diese im Verhältnis kürzer sind als die anderen beiden Phasen, wodurch ihre Akzeptanz verringert wird. Dies ist durch die Prozentangaben an den 4 Phasen dargestellt, welche dem Zeitanteil des Referenztaktes entsprechen den die 4 Phasen real abdecken. Somit liegt also durch die Messung eine direkte Beschreibung der Asymmetrie vor, wobei so aber keine Aussage über einen möglichen Jitter, also zeitliche Veränderungen der Asymmetrie, getroffen werden kann.

Um eine mögliche Erklärung zu finden betrachten wir dazu einmal den von der Placing–Software gewählten Aufbau in Abbildung 6.2.

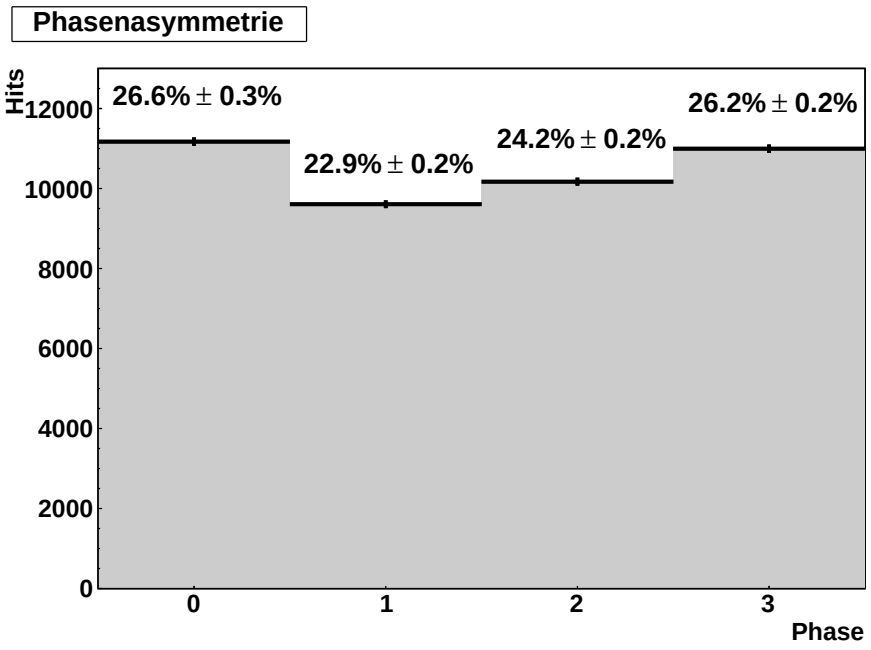

Abbildung 6.3.: Histogramm der Phasenasymmetrie

Unter der Annahme, dass die Takte selbst mit der exakten Phasenverschiebung zu den Flipflops geführt werden (sie laufen alle aus einem DCM aus und starten so mit innerhalb des

Jitters korrekten Phasenverschiebungen), sollte sich die Asymmetrie über diese Platzierung und die damit verbundenen Signalwege erklären lassen.

Zunächst scheint sich die Asymmetrie direkt durch die Entfernung der Flipflops zum Eingang, also durch den Signalweg, zu ergeben.

Zur Kontrolle dieser Vermutung können noch die errechneten Signallaufzeiten im „FPGA Editor“ von XILINX® überprüft werden. Dieser zeigt die realen Signalwege im fertig gerouteten Design und erzeugt bei Vergrößerung des Aufbaus am Eingang die Grafik in Abbildung 6.4. Hier scheint zunächst Phase 3 den längsten Signalweg aufzuweisen, während alle anderen Phasen nahezu identisch angeschlossen sind.

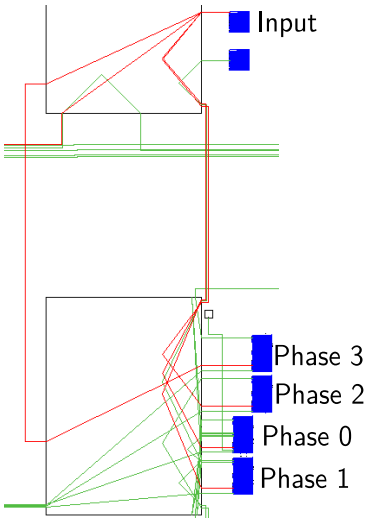

Abbildung 6.4.: Anordnung der Flipflops am Signaleingang. Screenshot aus FPGA Editor.

Als weitere Information stehen die vom FPGA–Editor bestimmten Zeiten der Signallaufzeiten in Tabelle 6.1 zur Verfügung. Bei Betrachtung dieser Zeiten fällt auf, dass sich die Unterschiede tatsächlich nur im Picosekundenbereich bewegen sollten. Es wurden auch die Laufzeiten der Taktsignale selbst untersucht, diese waren nach Berechnungen des FPGA Editor exakt identisch. Bemerkenswert jedoch ist der angegebene maximale Jitter im Datenblatt, der im Vierphasenbetrieb bei 150 ps pro Phase liegt, also durchaus die Abweichungen des generierten Pulses bei der Vergleichsmessung mit dem CAEN–TDC (siehe Abbildung 6.1) erklärt, allerdings bei der hier vorgenommenen Asymmetriemessung durch die statistische Mittelung verschwinden sollte.

Input–Flipflop (Phase)	Delay/ns (Signallaufzeit)	
0	0,533	ns
1	0,533	ns
2	0,541	ns
3	0,550	ns

Tabelle 6.1.: Signallaufzeiten laut FPGA Editor

Die beobachteten Unterschiede scheinen also (unter der Annahme, dass der FPGA Editor die Signallaufzeiten korrekt berechnet) an den Schaltzeiten der Flipflops selbst zu liegen. Auf diese kann kein Einfluss genommen werden, eine Korrektur wäre unter Umständen über eine feste Platzierung möglich, was allerdings eine Kalibrierung jedes einzelnen Kanals und damit für die spätere Nutzung einen nicht mehr vertretbaren Aufwand nach sich ziehen würde. Das Resultat erscheint dennoch sehr zufriedenstellend, da die Abweichungen nur gering sind (die Asymmetrie macht im Extremfall einen Messfehler von 300 ps aus).

6.2. Mögliche Verbesserungen

Um mögliche Verbesserungen untersuchen zu können, betrachten wir zunächst die Ergebnisse des vorigen Abschnitts 6.1. Möglichkeiten bestehen hier zum Einen in einer Korrektur der Asymmetrie, aber auch in einer Erhöhung der Abtastrate.

Als Vergleich sei die Umsetzung eines FPGA–TDC in [SKJ09] erwähnt. Hier wurde auf einem Spartan 3–FPGA unter Verwendung zweier Interpolationsstufen und durch Online–Korrektur von Nichtlinearitäten mit einem Hardware–Prozessor auf dem FPGA eine Auflösung von 45 ps erreicht, allerdings mit maximal zwei Kanälen (vergleiche dazu die Anwendung als PCI–Karte in [SKJR07]).

6.2.1. Kalibration

Eine Korrektur der Asymmetrie in der Hardware würde, wie bereits in Abschnitt 6.1 beschrieben, einen nicht vertretbaren Aufwand für jeden einzelnen Kanal bedeuten. Es ist allerdings möglich, nach einer Rauschmessung, wie sie auch im Rahmen der Arbeit durchgeführt wurde, automatisiert die Asymmetrie in Software zu korrigieren. Dies ist natürlich nur statistisch möglich und damit ist der Effekt im Idealfall auf die Auflösung von 1,79 ns begrenzt.

6.2.2. Migration

Im Gegensatz zu einer Korrektur der Asymmetrie, womit sich nur ein geringer Gewinn erreichen ließe, ist bei einer Migration auf einen neueren FPGA–Typ der Gewinn deutlich höher. Da die technische Entwicklung stets fortschreitet und somit neue FPGA–Typen auch dem Preisverfall unterworfen sind, ist es also sinnvoll, einen Aufbau direkt so anzulegen, dass eine Migration möglichst erleichtert wird. Im Rahmen von XILINX®–FPGAs ist dies mit der vorliegenden Umsetzung weitgehend gewährleistet, da außer den XILINX®–Implementationen von DCM, Schieberegister und FIFO sowie den definierten Constraints keine weiteren spezifischen Elemente verwendet werden.

Im Rahmen einer Migration bietet sich bei Verwendung einer neueren Version der Entwicklungsumgebung der integrierte Migrationsassistent an, beim Wechsel des FPGA selbst können teilweise identische Komponenten verwendet werden, zum Teil muss jedoch der sogenannte IP–Core („intellectual property core“) des Bauteils neu erzeugt werden. Da mit DCM, Schieberegister und FIFO nur grundlegende Bauteile verwendet wurden, ist auch eine zukünftige Unterstützung durch XILINX® gewährleistet.

Vorteile kann ein Nachfolge–FPGA hier insbesondere bei der maximalen Taktrate, die sich im vierphasigen Betrieb mit dem DCM erzielen lässt, sowie bei der maximalen Kanalanzahl (durch mehr Slices) bieten. Alternativ ließe sich dann auch (da bei neueren FPGAs einstellbare IO–Delays zur Verfügung stehen) ein anderer Ansatz verwirklichen: Anstatt eine Verschiebung der Phasen für die abtastenden Flipflops auszunutzen, könnte das Signal über IO–Delays zu synchron getakteten Flipflops verzögert werden (dies wurde bereits in Abschnitt 4.1 angedacht). So ließe sich eine deutlich höhere Auflösung erreichen, da so die Beschränkung auf 4 Phasen aufgehoben wäre. Der große Nachteil dabei ist allerdings, dass nun die IO–Delays jedes Kanals einzeln abgestimmt werden müssen.

7. Zusammenfassung

Im Rahmen dieser Bachelorarbeit ist es gelungen, durch Programmierung eines FPGA–Bausteins auf einem VME–Board einen TDC mit nebenstehenden Eigenschaften (siehe Tabelle 7.1) zu konstruieren. Dabei wurde eine für viele Experimente ausreichende Genauigkeit bei gleichzeitig kostengünstiger Umsetzung erreicht, während durch die Umsetzung auf FPGA–Basis noch weitere Möglichkeiten zur Datenverarbeitung direkt zur Verfügung stehen.

Auflösung	bis zu	1,79 ns
Maximale Eventzahl vor dem Trigger		64 events
Maximale Eventzahl nach dem Trigger		bis FIFO voll
Tiefe des FIFO		64 events
Maximaler Hit–Abstand		58,5 µs

Tabelle 7.1.: Kenndaten des Aufbaus

A. Quelltexte

A.1. TDC–Aufbau

A.1.1. Verwendete Datenstruktur (Auszug)

```
1  type sig_buf is
2    record
3      -- Shifting, Teil 1 (Clipping über einen Takt):
4      clk_fire_cl : std_logic_vector (3 downto 0);
5      -- Shifting in Synchrontakt, Phase 0:
6      clk_fire_ro : std_logic_vector (3 downto 0);
7      -- Intervallzähler:
8      timentp : std_logic_vector (timeres-1 downto 0);
9      -- Signal-schon-vorhanden--Erkennung
10     lastsig : std_logic;
11   end record;
12 -- Für alle Kanäle (Datentyp):
13 type sig_bufmulti is array (channels-1 downto 0) of sig_buf;
14 -- "Variable":
15 signal s2m : sig_bufmulti;
16 -- Vierphasige Clock:
17 signal clk_200_0, clk_200_90, clk_200_180, clk_200_270 : std_logic;
18 -- Asynchrone Triggerverarbeitung:
19 signal trigger_cl, trigger_ack : std_logic;
```

A.1.2. Aufbau des Kernbausteins (TDC)

```
-- 0. Phase (mit Mustererkennung):
my_clock_0 : process (clk_200_0)
variable
  data_ok : boolean;
begin
  if rising_edge(clk_200_0) then
    -- Mustererkennung der getroffenen Phasen aus Readout-Speicher
    -- (vom letzten Takt) für 'channels' Kanäle
    -- Die Muster sind 'falschrum', da Phasen b"4321" sind.
    for i in 0 to channels-1 loop
      data_error(i) <= '0';
      data_ok := true;
      case s2m(i).clk_fire_ro is
        when "1111" =>
          if (s2m(i).lastsig = '0') then
            data_out((i*(timeres+2))+1 downto i*(timeres+2)) <= "00";
          else
            data_ok := false;
          end if;
        when "1110" => data_out((i*(timeres+2))+1 downto i*(timeres+2)) <= "01";
        when "1100" => data_out((i*(timeres+2))+1 downto i*(timeres+2)) <= "10";
        when "1000" => data_out((i*(timeres+2))+1 downto i*(timeres+2)) <= "11";
        when "0000" => data_ok:=false; s2m(i).lastsig <= '0';
        when others => data_error(i) <= '1'; data_ok:=false;
      end case;
      -- Acknowledge-Signal für Trigger (asynchron!)
      if (trigger_cl = '1') then
        trigger_ack <= '1';
        data_ok:=true;
      else
        trigger_ack <= '0';
      end if;
      -- Datenausgabe
      if (data_ok) then
        data_out(((i+1)*(timeres+2))-1 downto (i*(timeres+2)+2)) <= s2m(i).timentp;
        data_ready(i) <= '1';
        -- Datenausgabe aktiv.
        -- Also Intervallzähler auf 0.
        s2m(i).timentp <= (others => '0');
        s2m(i).lastsig <= '1';
      else
        data_ready(i) <= '0';
        s2m(i).timentp <= s2m(i).timentp + 1;
      end if;
```

```
45      end loop;
46
47      -- Signalsynchronisierung Phase 0:
48      for i in 0 to channels-1 loop
49        s2m(i).clk_fire_cl(0) <= sig(i);
50        s2m(i).clk_fire_ro(0) <= s2m(i).clk_fire_cl(0);
51        s2m(i).clk_fire_ro(1) <= s2m(i).clk_fire_cl(1);
52      end loop;
53    end if;
54  end process;
55
56  -- 1. Phase:
57  my_clock_90 : process (clk_200_90)
58  begin
59    if rising_edge(clk_200_90) then
60      for i in 0 to channels-1 loop
61        s2m(i).clk_fire_cl(1) <= sig(i);
62        s2m(i).clk_fire_ro(2) <= s2m(i).clk_fire_cl(2);
63        s2m(i).clk_fire_ro(3) <= s2m(i).clk_fire_cl(3);
64      end loop;
65    end if;
66  end process;
67
68  -- 2. Phase:
69  my_clock_180 : process (clk_200_180)
70  begin
71    if rising_edge(clk_200_180) then
72      for i in 0 to channels-1 loop
73        s2m(i).clk_fire_cl(2) <= sig(i);
74      end loop;
75    end if;
76  end process;
77
78  -- 3. Phase:
79  my_clock_270 : process (clk_200_270)
80  begin
81    if rising_edge(clk_200_270) then
82      for i in 0 to channels-1 loop
83        s2m(i).clk_fire_cl(3) <= sig(i);
84      end loop;
85    end if;
86  end process;
```

A.2. Auslesesoftware

```
/*
  Auslese eines FPGA-TDC
*/

// Includes, VME-Anfrageroutine und SIGHANDLER aus Platzgründen eingespart

int main(int argc, char **argv)
{
  int imageask, imagedat;
  unsigned int val, addr, tval=0;
  unsigned int hitcount=0, i=0, eventid=0;
  unsigned int trigindex=0;
  unsigned int fifostat=0;
  unsigned short int segment=0, segmentbefore=0;
  double timeunit, subtimeunit;
  double abstime;

  map<int, double> dataset;
  map<int, double> triggerflag;
  map<int, int> rawdata;
  bool output_now=false;

  signal(SIGINT, sighandler);
  signal(SIGQUIT, sighandler);

  struct timestruct {
    Int_t event;
    Int_t hit;
    Int_t trigger;
    double reltime;
    double abstime;
    Int_t raw;
  };
  timestruct timedata;

  // Zeiteinheiten zur Umrechnung:
  timeunit = 1./140*1000.;
  subtimeunit = timeunit / 4.;

  // ROOT-File anlegen:
  f = new TFile("tdc.root","RECREATE");

  // TTree anlegen:
  tree = new TTree("TDC","tdc measured data");

  // Branches anlegen:
  tree->Branch("event",&timedata.event,"event/i");
```

```
48  tree->Branch("hit",&timedata.hit,"hit/i");
49  tree->Branch("trigger",&timedata.trigger,"trigger/i");
50  tree->Branch("reltime",&timedata.reltime,"reltime/d");
51  tree->Branch("abstime",&timedata.abstime,"abstime/D");
52  tree->Branch("raw",&timedata.raw,"rawdata/i");
53
54  // TDC-Reset (zum Test auskommentiert):
55  //sscanf("0x55550050", "%x", &addr);
56  //imagedat = allocImage(&addr, D32);
57  //vme.rl(imagedat, addr, &val);
58  //vme.releaseImage(imagedat);
59  // Reset beendet
60
61  // Auslesesschleife:
62  while (true) {
63    // Ist das FPGA zur Auslese bereit?
64    sscanf("0x55550030", "%x", &addr);
65    imageask = allocImage(&addr, D32);
66    if (!vme.rl(imageask, addr, &val)) {
67      fifostat = val;
68      //cout << fifostat;
69    }
70    vme.releaseImage(imageask);
71
72    // Wenn ja, dann lesen wir einen Datensatz:
73    if (fifostat == 1)  {
74      sscanf("0x55550020", "%x", &addr);
75      imagedat = allocImage(&addr, D32);
76      if (!vme.rl(imagedat, addr, &val)) {
77        // Erster Wert liegt in den 16 MSB:
78        tval = val >> 16;
79        // Coarse Timer beginnt beim dritten Bit von unten:
80        rawdata[hitcount] = tval;
81        // Sonderfall für ersten Wert: Ist dieser 0 und der ↘
             ↪allererste,
82        // dann ist es ein Füll-Wert.
83        // Das heißt, er wird ignoriert!
84        // Ansonsten: Füll-Null ist Abschluss, ebenfalls ↘
             ↪ignorieren, aber Ausgabe starten.
85        // Event zuende? Dann kann Ausgabe laufen:
86        if (hitcount != 0) {
87          if ((tval == 0)) {
88            output_now = true;
89          } else {
90            hitcount++;
91          }
92        }
93
94        // Zweiter Wert liegt in den 16 LSB:
95        tval = val & 0xFFFF;
96        // Coarse Timer beginnt beim dritten Bit von unten:
```

```
97          rawdata[hitcount] = tval;
98          // Event zuende? Dann kann Ausgabe laufen:
99          if (tval == 0) {
100           output_now = true;
101         } else {
102           hitcount++;
103         }
104       }
105       vme.releaseImage(imagedat);
106     }
107
108     // Event zuende? Ausgabe:
109     if (output_now) {
110       cout << "Event Nr. "<< eventid <<" aufgenommen.\n";
111       cout << "Hits: " << hitcount << "\n";
112
113       // Schritt 1: Segmentdifferenzierung.
114       // a) 0. Element gibt stets das erste Segment an, Zeitdaten ↘
            ↪hier uninteressant, denn:
115       // __|___|___|___ hat bei den 3 Hits nur 2 Zwischenräume.
116       segmentbefore = rawdata[0] & 3;
117       // b) echte Relativzeiten berechnen:
118       for (i=1; i < hitcount; i++) {
119         // b.1) Achtung bei Triggern:
120         // Auflösung ist geringer. Dies wurde aber erreicht, indem ↘
              ↪der nächste Hit nach Triggerpuls der Trigger ist. Also:
121         // o segmentbefore wie normal speichern
122         // o Triggerflag in "triggerflag" setzen
123         // o den ersten Trigger im event in trigindex hinterlegen
124         dataset[i-1] = rawdata[i] & 0x7FFF; // Nimmt das ↘
              ↪Triggerbit weg und lässt alles andere da.
125         if ((triggerflag[i-1] = (rawdata[i] >> 15)) == 1) {
126           if (trigindex == 0) trigindex = i-1;
127           cout << "Event enthaelt Trigger bei Hit Nr. " << i-1 << ↘
                ↪". \n";
128         }
129         segment = rawdata[i] & 3; // Segment zwischenspeichern
130         dataset[i-1] -= segmentbefore; // differenzieren
131         segmentbefore = segment; // neues Bezugssegment sichern
132
133         dataset[i-1] *= subtimeunit; // Zeit berechnen.
134         // Der Zähler muss generell um 1 erhöht werden. Damit ↘
              ↪ergibt sich:
135         dataset[i-1] += timeunit;
136       }
137       // Ergebnis: Dataset enthält nun alle Relativzeiten zwischen ↘
            ↪Index 0 und Index hitcount-2.
138       // Zum einfacheren Schleifenaufbau:
139       hitcount--;
140
141       // Schritt 2: Absolutzeiten berechnen.
```

```
142         // a) Minimalabsolutzeit bestimmen. Ausrichtung am ersten Trigger.
143         abstime = 0;
144         for (i = 0; i <= trigindex; i++) {
145           abstime -= dataset[i];
146         }
147         cout << "Minimale Absolutzeit (Fenster vor Trigger) liegt bei: " << abstime << " ns. " << "\n";
148         // b) Berechnung erfolgt direkt beim Ausgeben.
149
150         cout << "Ausgabe startet." << "\n";
151         cout << "************************************************************\n";
152         cout << "Hit" << "\t" << "Raw" << "\t" << "Trg"<< "\t" << "Rel. T" << "\t\t" << "Abs. T" << "\n";
153         cout << "************************************************************\n";
154
155         // Schritt 3: Ausgabe. Dazu timedata füllen.
156         timedata.event = eventid;
157         for (i=0; i < hitcount; i++) {
158           timedata.hit = i;
159           timedata.trigger = triggerflag[i];
160           timedata.reltime = dataset[i];
161           abstime += dataset[i];
162           timedata.abstime = abstime;
163           timedata.raw = rawdata[i+1];
164
165           // ROOT-Tree füllen:
166           tree->Fill();
167           // Debug-Ausgabe:
168           cout << i << "\t" << rawdata[i+1] << "\t" << triggerflag[i] << "\t" << dataset[i] << "\t\t" << abstime << "\t" << "\n";
169         }
170         cout << "************************************************************\n";
171         char timestr[128];
172         time_t tnow;
173         struct tm *now;
174         tnow = time(NULL);
175         now = localtime(&tnow);
176         strftime(timestr,sizeof(timestr),"%H:%M:%S",now);
177         cout << "Ausgabe beendet um " << timestr << "\n";
178         cout << "Naechstes Event abwarten." << "\n";
179
180         // Eventid erhöhen.
181         eventid++;
182
183         dataset.clear();
184         rawdata.clear();
```

```
185         triggerflag.clear();
186         trigindex = 0;
187         hitcount = 0;
188         output_now = false;
189 
190         fflush(stdout);
191       }
192     }
193 
194     tree->Write();
195     f->Close();
196 
197     return 0;
198 }
```

Glossar

CLB

„Configurable Logic Block“. Ein mehrfach auf dem FPGA vorhandener, jeweils identischer Block, welcher verschiedene Slices enthält. Der Name resultiert aus der dynamischen Rekonfigurierbarkeit der logischen Verknüpfungen über ein Bitmuster. Details zum Aufbau werden in Abschnitt 3.1 beschrieben.

Constraint

Eine manuelle oder technische Beschränkung / Festlegung von Laufzeiten, Platzierung oder Schaltzeiten zwischen zwei vorher definierten Punkten. Diese Constraints werden beim Erzeugen des Schaltplans berücksichtigt und erlauben so eine beliebig feine Kontrolle der Optimierungsprozesse.

CPLD

„Complex Programmable Logic Device“, ähnlich einem FPGA, allerdings mit weniger Flipflops und (ursprünglich) mit der Zielsetzung, eine logische Zuordnung zwischen Eingängen und Ausgängen abzubilden (also nicht zur komplexen Datenverarbeitung).

DCM

„Digital Clock Manager“, ein Baustein, welcher innerhalb fester Grenzen unter Verwendung mehrerer DLL einen Takt multiplizieren, dividieren oder in der Phase verschieben kann.

DLL

„Delay–locked loop“, eine mit einem Regelkreis veränderbare Kette von Verzögerungen (meist Inverter, über die Betriebsspannung oder diskret über ihre Anzahl kontrolliert) zur Erzeugung einer definierten Verzögerung eines eingeführten Signals.

FIFO

„First-In – First-Out“. Ein Datenspeicher, der zuerst eingeschriebene Daten auch zuerst wieder ausgibt. Die Größe ist begrenzt, ist ein FIFO voll, werden keine weiteren Daten angenommen.

FPGA

„Field Programmable Gate Array“, ein integrierter Schaltkreis aus Blöcken mit Logikbausteinen, die mit einem Bitmuster verschaltet werden können. Damit ist die Funktion dynamisch programmierbar. Details zum Aufbau werden in Abschnitt 3.1 beschrieben.

Gate

Ein logisches Signal, welches mit „Low“ einen Takteingang blockiert und somit etwa ein dauerhaftes Schreiben bei jedem Taktzyklus einer Write–Clock verhindert.

LSB

„Least Significant Bit“. Das Bit mit dem niedrigsten Wert im Binärsystem, also jenes, für das 2^n minimal ist. Dies entspricht nicht notwendigerweise der Reihenfolge der Bits im Speicher, je nach Prozessor kann diese anders sein, was mit dem Begriff der „Endianness“ bezeichnet wird, die es ebenfalls in verschiedensten Mischungen gibt.

MSB

„Most Significant Bit“. Das Bit mit dem höchsten Wert im Binärsystem, also jenes, für das 2^n maximal ist. Dies entspricht nicht notwendigerweise der Reihenfolge der Bits im Speicher, je nach Prozessor kann diese anders sein, was mit dem Begriff der „Endianness“ bezeichnet wird, die es ebenfalls in verschiedensten Mischungen gibt.

NGC

Ein XILINX®–spezifisches Dateiformat, welches die Formate `EDIF` und `NCF` ersetzt. Dabei enthält `EDIF` in einem neutralen Format Schaltplan und Netzliste, während `NCF` die festgelegten Bedingungen (Timings, Platzierung etc.) enthält. NGC kombiniert beide Formate.

NIM

„Nuclear Instrumentation Standard“, welcher die Logikpegel „1“ und „0“ über negative Ströme (bzw. negative Spannungen bei 50 Ω) definiert.

PLL

„Phase–locked loop“, ein mit einem Regelkreis stabilisierter Oszillator, meist ein Ring von Verzögerungen (etwa eine ungerade Anzahl von Invertern, über Betriebsspannung oder Anzahl regelbar) zur Erzeugung eines Taktsignals, welches über Koinzidenzen mit einem Referenztakt stabilisiert wird.

Schieberegister

Ein Datenspeicher, welcher die eingeschobenen Daten um seine Tiefe verzögert. Dazu werden die Daten mit jedem Taktpuls eine Ebene tiefer geschoben und „fallen“ am Ende automatisch heraus.

Slice

Ein Block aus Logikgattern. Im Spartan 3 befinden sich pro CLB vier solche Slices, zwei des Typs „M“ und zwei des Typs „L“. Details zum Aufbau werden in Abschnitt 3.1 beschrieben.

VHDL

„Very High Speed (Integrated Circuit) Hardware Description Language“, eine Sprache zur hardwarenahen Beschreibung des Verhaltens einer digitalen Schaltung. In Abwandlungen auch rein funktional (zur Simulation) oder für analoge Schaltungen verwendbar (Bezeichung: VHDL–AMS).

VME

VMEbus („Versa Module Eurocard“-bus). Ein asynchrones Bussystem mit anfangs 16 Bit Datenbus und 24 Bit Adressbus, als VME64 inzwischen auch mit 64 Bit Busbreite, in der vorliegenden Arbeit wird VME32 verwendet. Dieser Mehrteilnehmerbus bietet durch ein vergleichsweise einfaches Protokoll und eine Unterstützung durch viele Rechnerarchitekturen eine kostengünstige Möglichkeit zur Umsetzung einer Datenerfassung.

Literaturverzeichnis

[1-C09] 1-CORE Technologies: *FPGA synthesis and implementation (Xilinx design flow).* http://www.1-core.com/library/digital/fpga-design-tutorial/implementation_xilinx.shtml, April 2009. abgerufen am 2.7.2010.

[Gri] Griffiths, Bruce: *Precision Time and Frequency Systems.* http://www.ko4bb.com/~bruce/. abgerufen am 6.7.2010.

[Kal04] Kalisz, Józef: *Review of methods for time interval measurements with picosecond resolution.* Metrologia, 41:17–23, 2004.

[KPP85] Kalisz, J., M. Pawlowski und R. Pelka: *A method for autocalibration of the interpolation time interval digitiser with picosecond resolution.* Journal of Physics E: Scientific Instruments, 18(5):444, 1985. http://stacks.iop.org/0022-3735/18/i=5/a=018.

[KPP87] Kalisz, J., M. Pawlowski und R. Pelka: *Error analysis and design of the Nutt time-interval digitiser with picosecond resolution.* Journal of Physics E: Scientific Instruments, 20(11):1330, 1987. http://stacks.iop.org/0022-3735/20/i=11/a=005.

[Leo87] Leo, W. R.: *Techniques for Nuclear and Particle Physics Experiments.* Springer–Verlag, 1987.

[SAL06] Song, Jian, Qi An und Shubin Liu: *A High–Resolution Time–to–Digital Converter Implemented in Field–Programmable–Gate–Arrays.* IEEE Transactions on Nuclear Science, 53(1):236–241, Februar 2006.

[SKJ09] Szplet, R., J. Kalisz und Z. Jachna: *A 45 ps time digitizer with a two-phase clock and dual-edge two-stage interpolation in a field programmable gate array device.* Measurement Science and Technology, 20(2):025108, 2009. http://stacks.iop.org/0957-0233/20/i=2/a=025108.

[SKJR07] Szplet, R., J. Kalisz, Z. Jachna und K. Różyc: *A method for autocalibration of the interpolation time interval digitiser with picosecond resolution.* Proc. 39th Ann. Precise Time and Time Interval Meeting (PTTI), Seiten 531–540, 2007.

[WNS05] WNSL Nuclear Astrophysics Group: *Explosive Nucleosynthesis.* http://wnsl.physics.yale.edu/astro/research/, März 2005. abgerufen am 01.07.2010.

[XILINX®09] XILINX®: *Spartan–3 FPGA Family Data Sheet.* http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf, 2009. Version 2.5 vom 4.12.2009.

B. Selbstständigkeitserklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate kenntlich gemacht habe.

Bonn, den ..... Unterschrift .....