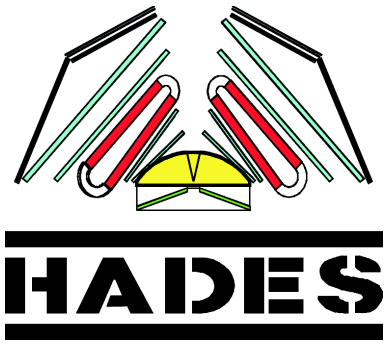


Entwicklung einer Test-Umgebung für den HADES Second-Level Trigger

DIPLOMARBEIT VON RENÉ BECKER

aus Laubach, OT. Münster

II. Physikalisches Institut

der Justus-Liebig-Universität Gießen

28. Dezember 1998

Zusammenfassung

Das derzeit an der GSI (**G**esellschaft für **S**chwer**i**onenforschung) im Aufbau befindliche Spektrometer HADES (**H**igh **A**cceptance **D**i-**E**lectron **S**pectrometer) dient zur Untersuchung der Dileptonenproduktion in hadronen- und schwerioneninduzierten Reaktionen. Um die zu erwartenden Datenraten von ca. 3GByte/Sekunde mit einem vertretbaren Aufwand verarbeiten zu können, verfügt HADES über ein dreistufiges Triggersystem, durch dessen Hilfe nur physikalisch interessante Daten für die spätere Auswertung gespeichert werden. Die zweite und effektivste Triggerstufe, der *Second Level Trigger*, der eine Datenreduktion um den Faktor 100 erreichen soll, wird komplett am II.Physikalischen Institut der JLU-Gießen entwickelt. Um die Funktion und Effizienz dieser Triggerstufe unabhängig vom restlichen Detektorsystem Testen zu können, wurde im Rahmen dieser Diplomarbeit ein Konzept entwickelt und teilweise umgesetzt. Dieses Testsystem besteht im wesentlichen aus einer PC-Einsteckkarte für einen PCI-Steckplatz (**P**eripheral **C**omponents **I**nterconnect) mit einer Schnittstelle zum Second-Level Trigger.

Mit diesem Testsystem soll voraussichtlich im Frühjahr 1999 das komplette Datenaufnahmesystem getestet werden.

Inhaltsverzeichnis

Kapitel 1

Einführung und Motivation

Während die fundamentalen Eigenschaften (wie Masse, Lebensdauer, etc.) freier Hadronen weitgehend bekannt sind, stellt das Verhalten derselben innerhalb hadronischer Materie ein aktuelles Forschungsgebiet dar. Neben dem Zustand in kalter Kernmaterie interessiert man sich auch für Veränderungen in heißer, verdichteter hadronischer Materie.

Als Obversable bei diesen Untersuchungen bieten sich Leptonenpaare an: da sie nicht an der starken Wechselwirkung teilnehmen, können sie Informationen aus dem Inneren der Materie weitgehend ungestört hinaustragen.

Hinweise auf eine mögliche Veränderung der Eigenschaften von Hadronen bei großen Baryonendichten wurden schon am DLS (**D**i-**L**epton **S**pectrometer) am BEVALAC (Berkley,USA) gefunden: Bei Schwerionenstößen stimmte das Spektrum der invarianten Masse nicht mehr mit den theoretischen Vorhersagen ohne Beachtung von In-Medium Effekten überein [1]. Ähnliche Ergebnisse wurden kurze Zeit später auch bei Messungen mit CERES am CERN in Genf gefunden [2]: Wie man in Abbildung 1.1 (links) sieht, lassen sich die Stöße von Protonen mit schweren Kernen, in diesem Beispiel Gold, bei denen keine große Baryonendichte erreicht wird, recht gut theoretisch beschreiben. Im Gegensatz dazu beobachtet man bei Stößen zweier schwerer Ionen wie z.B. Schwefel und Gold (Abbildung 1.1, rechts) eine deutliche Zunahme der Zählrate bei kleineren invarianten Massen.

Als Reaktion auf diese Messungen enstanden eine Vielzahl von theoretischen Modellen, die diesen Dileptonenüberschuß zu erklären versuchen, sich jedoch zum Teil signifikant unterscheiden. Eine mit einem feldtheoretischen Baryonen-Mesonen-Modell durchgeführte Rechnung [4] sagt voraus, daß die Kopplung von Austausch-Pionen an Δ-Loch Zustände der ausschlaggebende Faktor für die zu erwarteten In-Medium Modifikationen

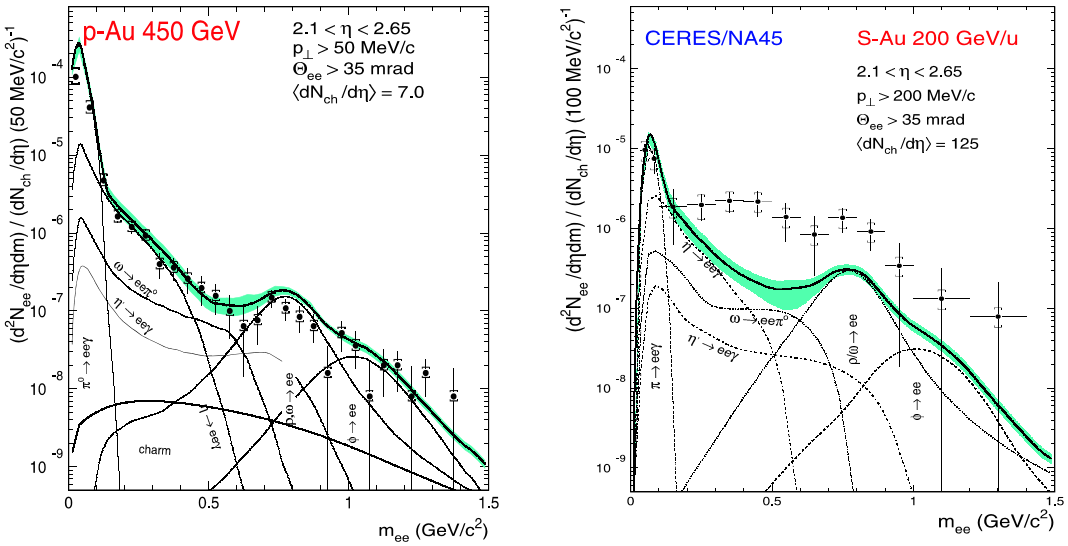


Abbildung 1.1: *Ergebnisse von Experimenten am CERN. Man erkennt deutlich, wie die p-Au-Stöße recht gut durch die Theorie beschrieben werden, bei schweren Systemen sind jedoch deutliche Abweichungen von der theoretischen Vorhersage zu erkennen[2]*

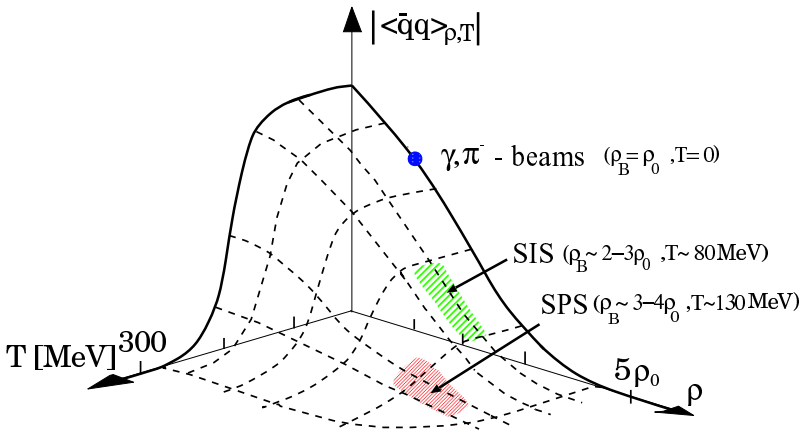


Abbildung 1.2: *Eine auf dem Nambu-Jona-Lasinio Modell [3] beruhende Rechnung zum Erwartungswert des chiralen Kondensates als Funktion von Temperatur und Baryonendichte zu sehen. Hervorgehoben sind die am SIS bzw. SPS zugänglichen Regionen.*

Meson	Masse $[MeV/c^2]$	Breite $[MeV/c^2]$	$c \cdot t\ [fm]$	Dominanter Zerfall	Verzweigungsverhältnis $e^+e^-/hadr.$ Zerfälle
ρ	768	152	1.3	$\pi\pi$	$4.4 \cdot 10^{-5}$
ω	782	8.43	23.4	$\pi^+\pi^-\pi^0$	$7.2 \cdot 10^{-5}$
ϕ	1019	4.43	44.4	K^+K^-	$3.1 \cdot 10^{-4}$

Tabelle 1.1: Fundamentale Eigenschaften der leichtesten Vektormesonen

des ρ-Mesons sind. Das ρ-Meson wird dabei als $\pi^+\pi^-$-Resonanz beschrieben, dessen Breite sich in Folge der Kopplung der Pionen an die Δ-Lochzustände verändert. Allerdings wird in diesem Modell keine Verschiebung der Masse erwartet. Im Kontrast dazu steht die Aussage eines weiteren Modells, das auf QCD-Regeln beruht[5]: Bei zunehmender Temperatur und Dichte wird hier eine Abnahme des chiralen Kondensats bis hin zu $\langle q\bar{q}\rangle = 0$ (Restauration der chiralen Symetrie) vorausgesagt (Abbildung 1.2). Da im Rahmen der QCD die Masse von Konstituenten-Quarks mit dem Konzept der chiralen Symmetriebrechung erklärt wird, sollte man eine Verschiebung der ρ-Masse hin zu kleineren Werten beobachten. Als wichtigsten Beitrag zum Dileptonenspektrum wird bei den meisten aktuelleren Rechnungen allerdings die paarweise Anihilation von Pionen ($\pi^+\pi^- \rightarrow e^+e^-$) angesehen. Einige der neueren Rechnungen, die auch den Effekt der Pionen-Anihilation berücksichtigen, sind in Abbildung 1.3 dargestellt.

Die experimentelle Untersuchung solcher *In Medium Modifikationen* bei $\rho > \rho_0$ ($\rho_0 \approx 0.17 fm^{-3}$, Baryonendichte im Atomkern) ist momentan nur mit Hilfe von Schwerionenkollisionen möglich. Am SIS (**S**chwer**i**onen-**S**ynchrotron) der GSI (**G**esellschaft für **S**chwer**i**onenforschung) in Darmstadt erreicht man bei solchen Kollisionen für kurze Zeit (ca. 10 fm/c) Baryonendichten von bis zu $3\rho_0$ und Temperaturen zwischen 70-100 MeV [6].

Bei diesen Kollisionen werden unter anderem auch die leichten Vektormesonen ρ, ω und ϕ produziert, von denen sich insbesondere das ρ aufgrund der in Tabelle 1.1 aufgeführten Eigenschaften hervoragend als Sonde eignet: Wegen seiner kurzen Lebensdauer wird der Zerfall dieses Teilchens mit hoher Wahrscheinlichkeit noch in der heißen und dichten Reaktionszone stattfinden.

Durch den leptonischen Zerfallskanal der Vektormesonen können anschließend Informationen über die Verhältnisse während der dichten Phase weitestgehend ungestört die

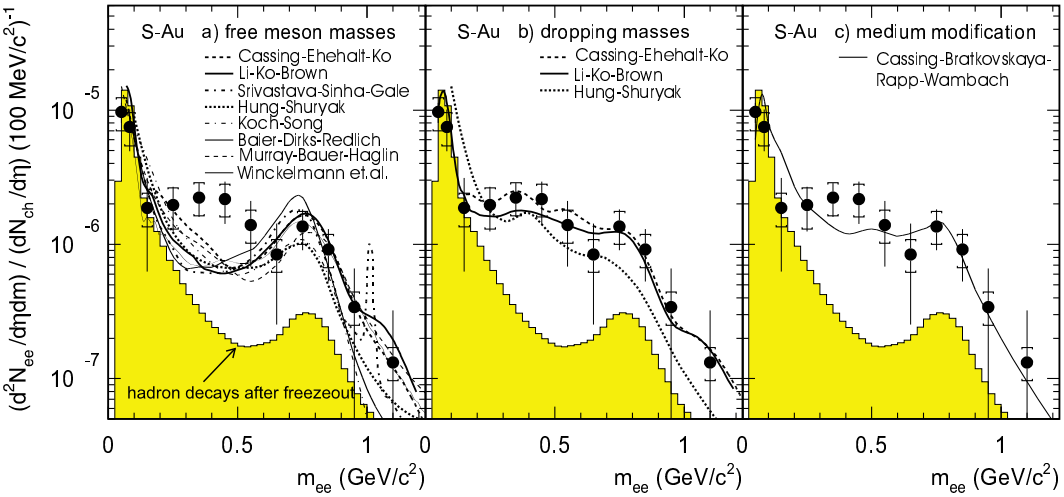


Abbildung 1.3: *Vergleich von einigen theoretischen Modellen mit den Daten von CERES. Bei allen Modellen wurde die Pionen-Anihilation ($\pi^+\pi^- \rightarrow e^+e^-$) mit berücksichtigt. [2]*

Reaktionszone verlassen, und eventuell auftretende Modifikationen des ρ-Mesons sollten sich in der invarianten Masse der nachgewiesenen e^+e^--Paare niederschlagen.

Auch um diese Phänomene näher zu untersuchen, befindet sich derzeit an der GSI ein neues Spektrometer mit dem Namen HADES (**H**igh **A**cceptance **D**i **E**lektron **S**pectrometer) im Aufbau, welches im folgenden Kapitel vorgestellt wird.

Das primäre Ziel des HADES-Spektrometers ist die Überprüfung der vielen theoretischen Modelle auf ihre Gültigkeit, vor allem in Hinsicht auf außergewöhnliche äußere Umstände wie z.B. großen Dichten. Zu einem späteren Zeitpunkt soll auch mit dem in der Planung befindlichen Pionenstrahl an der GSI der Zerfall rückstoßfrei erzeugter ω-Mesonen untersucht werden [7].Auf diese Weise wird es möglich, auch In-Medium Effekte in kalter Kernmaterie zu untersuchen. Die Produktion der ω-Mesonen muß dazu allerdings rückstoßfrei geschehen, da es sonst aufgrund seiner langen Lebensdauer von 23 fm/c (siehe Tabelle 1.1) außerhalb der Reaktionszone zerfallen würde.

Ein weiteres physikalisches Ziel von HADES wird sein, den zeitartigen elektromagnetischen Formfaktor von Hadronen, insbesondere des ω-Mesons, zu vermessen. Hierzu müssen speziell deren Dalitz-Zerfälle des ω-Mesons nachgewiesen werden. Das Massenspektrum der Dileptonen sollte dann die q^2-Abhängigkeit des ω-Formfaktors wiederspiegeln. Um das bei diesem Zerfallskanal auftretende π^0-Meson durch seinen Zerfall in zwei Photonen nachweisen zu können, benötigt man allerdings noch ein elektromagneti-

sches Kalorimeter. Dafür würde sich prinzipiell das Two Arm Photon Spektrometer TAPS hervorragend eignen, das allerdings ist in seiner derzeitigen Konfiguration keinen ausreichenden Raumwinkel abdeckt.

Ein großes Problem bei der Dileptonenspektroskopie sind die sehr kleinen Produktionswahrscheinlichkeiten für Dileptonen aus Mesonenzerfällen im Vergleich zu denen der Hadronen (vgl. Tabelle 1.1). Um eine hinreichende Statistik zu erhalten, muß man folglich mit sehr hohen Strahlintensitäten arbeiten: für den späteren Betrieb ist eine Strahlintensität von ca. 10^8 pps (particles per second) erforderlich. Da die dabei anfallenden Datenmengen nur mit sehr hohem finanziellen und technischem Aufwand vollständig auf Band zu speichern wären, benötigen solche Experimente einen selektiven Trigger, der dafür sorgt, daß nur physikalisch interessante Daten für die spätere Analyse gespeichert werden.

Für HADES wird ein 3-stufiges Triggersystem konstruiert, die größte Leistung, eine Reduktion der Datenraten um den Faktor 100, soll dabei die zweite Triggerstufe erbringen, die in Kapitel 3 ausführlich beschrieben wird.

Dieser sogenannte *Second-Level Trigger* wird vollständig in Gießen konzipiert und produziert. Um die Funktion und Performance desselben überprüfen zu können, wurde ein System benötigt, mit dem möglichst experimentnahe Umstände geschaffen werden können. Diese *Trigger Test Umgebung* wurde im Rahmen der vorliegenden Diplomarbeit entwickelt.

Kapitel 2

Das HADES Spektrometer

2.1 Überblick

Ein wesentliches Problem bei der Dileptonenspektroskopie sind die sehr kleinen Produktionswahrscheinlichkeiten für Dileptonen aus Mesonenzerfällen im Vergleich zu denen der Hadronen (vgl. Tabelle 1.1). Um eine hinreichende Statistik zu erhalten, muß man demzufolge mit sehr hohen Strahlintensitäten arbeiten: Bei Verwendung eines Targets mit 1% Wechselwirkungslänge muß die Intensität des Schwerionenstrahls ungefähr 10^8 Teilchen pro Sekunde betragen, um auf eine Ausbeute von 0.1 e^+e^--Paaren mit einer invarianten Masse $m_{inv} > 500 MeV/c^2$ pro Sekunde zu kommen. Daher benötigt man einen Detektor mit sehr hoher geometrischer Akzeptanz und Zählratenfestigkeit. Zusätzlich muß die Massenauflösung des Detektorsystems hinreichend groß sein, um die Beiträge der einzelnen Mesonen wirklich trennen zu können.

Bei der Entwicklung von HADES wurde daher in erster Linie eine Massenauflösung von 1% (0.8% in der ρ, ω-Region ($\Delta m < 10 MeV/c^2$)) bei gleichzeitig hoher geometrischer Akzeptanz von ca. 40% angestrebt. HADES zeichnet sich durch eine extreme Verbesserung von Akzeptanz und Auflösungsvermögen gegenüber Vorgänger-Experimenten wie DLS aus und wird mitunter auch als Experiment der zweiten Generation bezeichnet.

Um die oben genannten Anforderungen zu erfüllen, ist HADES aus mehreren Detektortypen zusammengesetzt, die jeweils rotationsymetrisch um den Strahl angeordnet sind. Jede Detektor-Komponente ist dabei in 6 Segmente unterteilt, wobei jedes Segment 60° abdeckt. Durch eine konusförmige Anordnung der einzelnen Komponenten wird der Polarwinkel zwischen 18° und 85° abgedeckt.

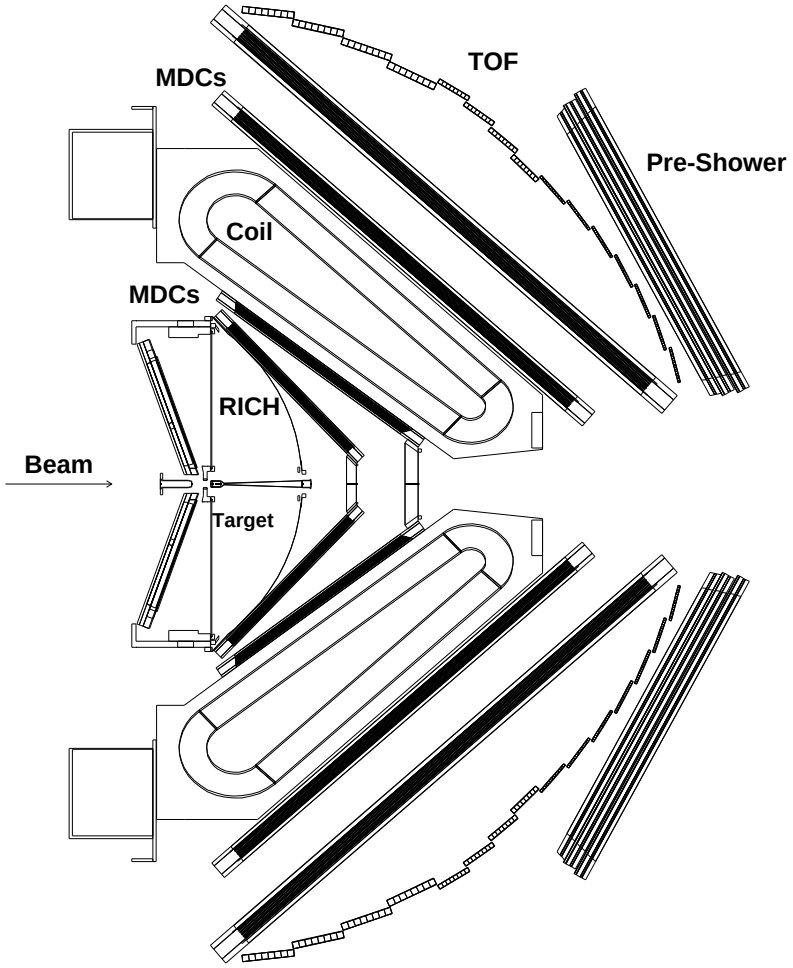


Abbildung 2.1: *An diesem schematischen Querschnitt durch den HADES Detektor sind die wesentlichen Komponenten dargestellt: Im Zentrum der ringabbildende Čerenkov-Detektor (RICH) mit einem gasförmigen Radiator (C_4F_{10}) und einer festen Photokathode aus CsI. Er dient zu einer ersten Leptonenidentifikation. Weiter außen befindet sich zur Impulsmessung das eigentliche Magnet-Spektrometer, das sich aus supraleitenden Spulen und jeweils zwei Lagen Mini-Drift-Kammern (MDCs) vor und hinter den Feldspulen zusammensetzt. Ganz außen befinden sich zur weiteren Teilchenidentifizierung eine Flugzeitwand (TOF) aus schnellen Plastikszintillatoren. Bei kleinen Polarwinkeln wird der TOF durch den SHOWER-Detektor ergänzt, der Leptonen anhand von elekromagnetischen Schauern erkennt.*

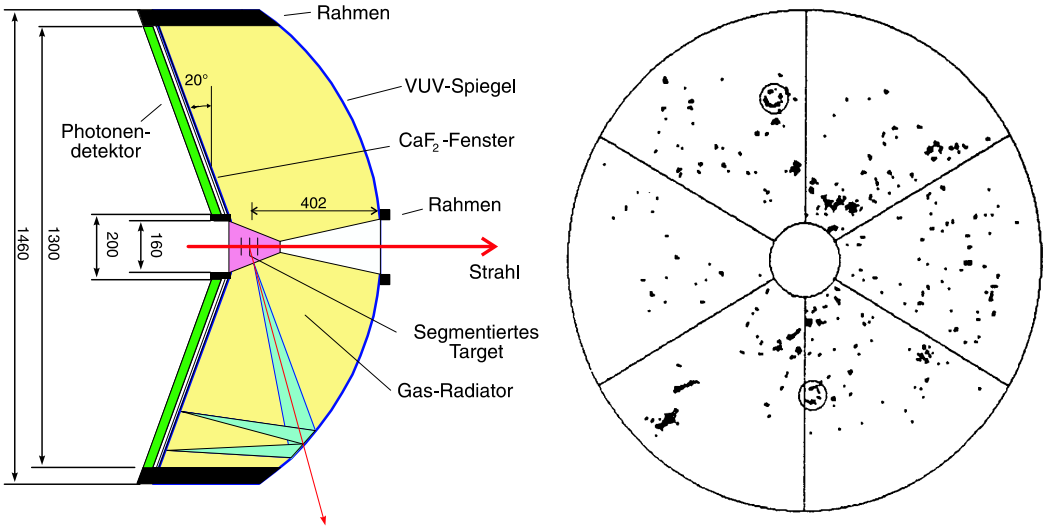


Abbildung 2.2: *Auf der linken Seite ist ein Querschnitt durch den RICH-Detektor zu sehen, auf der rechten ist die Pad-Belegung bei einem simulierten Ereignis gezeigt. Zwei Ringe sind durch Kreise markiert.*

2.2 Der RICH

Der Ring Imaging Čerenkov Detektor RICH ist das Herz des Detektorsystems und die wichtigste Komponente zur Diskriminierung des hadronischen Untergrundes. Er besteht im wesentlichen aus einem gasförmigen Radiator (C_4F_{10}) der das Target umschließt, einem sphärischen Spiegel und einem UV-Photonendetektor. Wenn ein geladenes Teilchen das Radiatorgas durchquert, erzeugt es bei Überschreitung einer Grenzgeschwindigkeit

$$\beta_{th} = \frac{1}{n(\omega)}$$

(n: Brechungsindex des Radiatorgases; w: Frequenz des Lichts)
das sogenannte *Čerenkov-Licht*, das unter einem geschwindigkeitsabhängigen Winkel

$$\cos\theta_C = \frac{1}{n(\omega)\cdot\beta} \quad , \qquad \beta = \frac{v}{c}$$

(v: Geschwindigkeit des Teilchens; c: Lichtgeschwindigkeit)
rotationssymetrisch zur Bewegungsrichtung des Teilchens emittiert wird[8]. Dieses Licht wird durch einen Spiegel auf den UV-Photonendetektor reflektiert und gleichzeitig fokussiert. Ein angenehmer Nebeneffekt bei der Verwendung des Spiegels ist, daß unerwüschte Signale von gestreuten Partikeln, die weitgehend alle in Vorwärtsrichtung emittiert werden, vermieden werden. Abbildungsfehler, die der Spiegel verursacht, werden durch eine

positionsabhängige Dimensionierung der einzelnen Pads kompensiert, wodurch die Ringsuche erheblich vereinfacht wird. Das Radiatorgas ist so gewählt, das alle Protonen und nahezu alle Pionen zu langsam sind um Čerenkov Licht zu erzeugen. Dadurch wird der RICH nahezu hadronenblind und eignet sich somit hervoragend für eine erste Leptonenidentifizierung. Die erzeugten Elektronen und Positronen bewegen sich mit ultrarelativistischen Geschwindigkeiten ($v \approx c$), so daß der Čerenkov-Winkel θ_C sich nach obiger Gleichung einem asymptotischen Grenzwert nähert. Bei der Suche nach Ringen kann man sich daher auf einen konstanten Durchmesser beschränken.

2.3 Der Magnet

Der Magnet stellt in Verbindung mit den MDCs das eigentliche Spektrometer dar: Durch das Feld des Magneten erfahren geladene Teilchen durch die Lorentz-Kraft eine Ablenkung, welche eine Impulsbestimmung ermöglicht.

In Abbildung 2.3 ist ein Bild des supraleitenden Toroids zu sehen. Um eine hinreichend genaue Impuls- und Massenauflösung zu erreichen, muß eine möglicht große transversale Ablenkung des betreffenden Teilchens erreicht werden, sie darf aber nicht so groß sein, daß das Teilchen nicht mehr die Detektoren durchquert. Für HADES ist zu diesem Zweck ein Magnetfeld von ca 0.5 T angestrebt, welches durch supraleitende Spulen erzeugt werden wird. Durch die Wahl einer toroidalen Geometrie soll die Einschränkung des Magnetfeldes auf den Bereich zwischen den Driftkammern erreicht werden, insbesondere darf die Teilchenbahn innerhalb des RICH nicht beeinträchtigt werden.

2.4 Die Driftkammern

Vor und hinter den supraleitenden toroidialen Magnetspulen befinden sich jeweils zwei Ebenen von Mini Drift Kammern (MDC :**M**ini **D**rift **C**hamber) mit deren Hilfe die Teilchenbahn durch das Magnetfeld hindurch rekonstruiert werden kann, was wiederum eine Impulsbestimmung des betreffenden Teilchens ermöglicht. Mit Hilfe der Impulsinformationen erfolgt dann die Bestimmung der invarianten Masse eines Dileptons, und damit auch der invarianten Masse des zerfallenen Mesons.

Jede dieser MDC-Ebenen ist eine Kombination aus 6 hintereinanderliegenden Driftkammern, damit auch bei hohen Teilchenmultiplizitäten eine eindeutige Zuordnung der

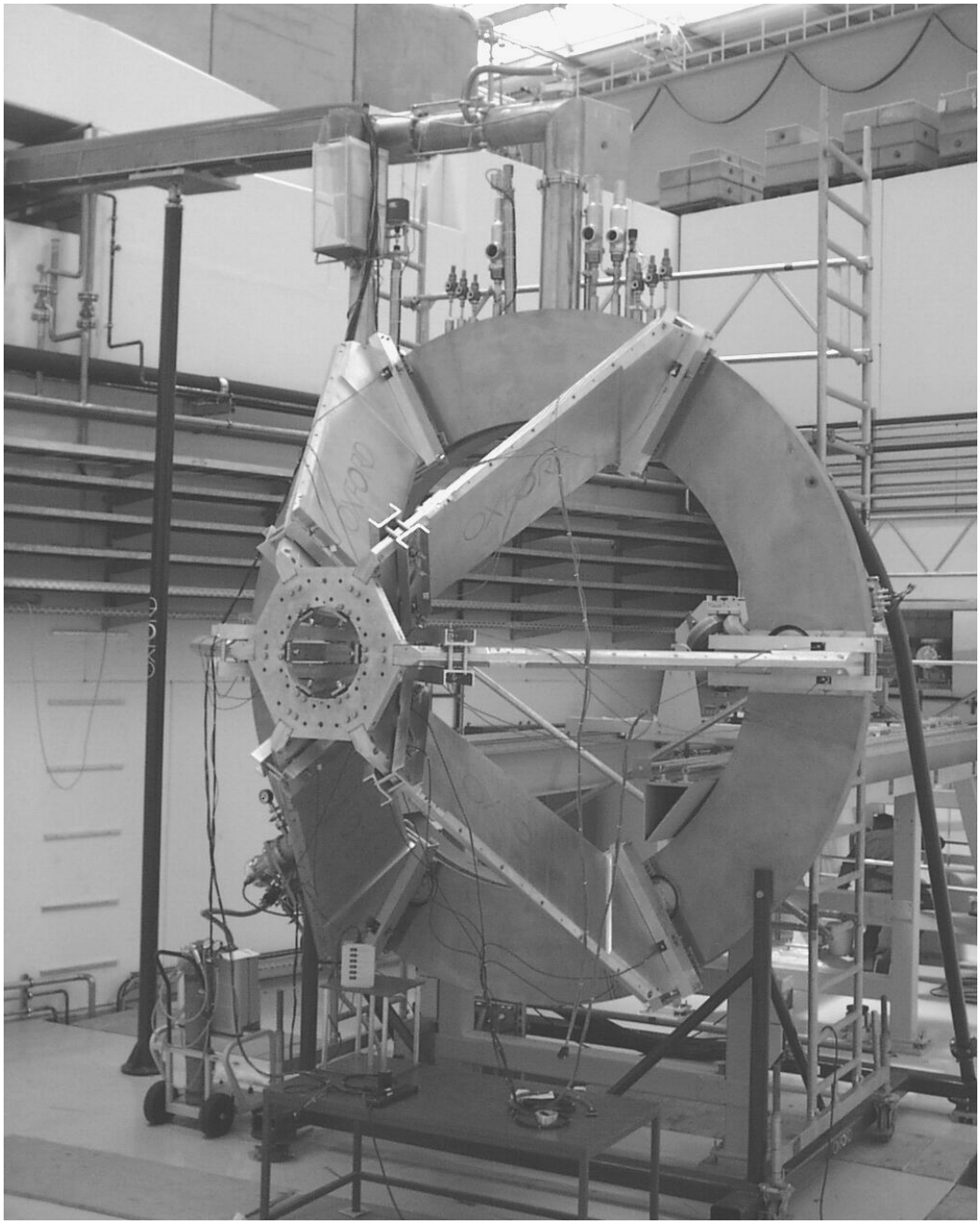


Abbildung 2.3: *Der supraleitende Magnet des HADES-Spektrometers. Klar erkennbar ist die toroidale Geometrie, durch die das Magnetfeld weitgehend auf den Raum zwischen den MDC beschränkt bleiben soll.*

einzelnen Driftzeitinformationen zu den einzelnen Treffern gewährleistet ist. Darüber hinaus sind die sensitiven Ebenen um Winkel von $-20°, 40°, 0°, 0°, -40°, 20°$ gegeneinander verdreht, um zu verhindern, daß Informationen durch Doppeltreffer in einer Zelle verlorengehen. Der Abstand der Anodendrähte beträgt bei der innersten Driftkammer 5mm, bei der äußersten 1,2 cm. Die maximale Ortsauflösung der MDCs soll $80\mu m$ betragen, wodurch eine Impulsbestimmung mit 1% Genauigkeit ermöglicht wird. Damit kann auch die geforderte Massenauflösung von 1% gewährleistet werden.

2.5 META

Als META (**M**ultiplicity **E**lectron **T**rigger **A**rray) wird die Kombination aus Flugzeitwand (TOF) und Schauerdetektor bezeichnet. Diese Detektorkomponente ermöglicht eine weitere Leptonenidentifizierung durch die Flugzeitmessung bei großen Polarwinkeln und zusätzlicher Schaueridentifizierung bei Polarwinkeln unter $45°$, wo aufgrund des Lorenz-Boosts die Flugzeitbestimmung allein keine Trennung von Hadronen und Elektronen möglich ist. Außerdem wird anhand der Teilchenmultiplizität im TOF der *First Level Trigger* generiert (Siehe Kapitel 3).

Der SHOWER Detektor, der bei Polarwinkeln unter $45°$ zur Leptonenidentifizierung beitragen soll, besteht aus drei Vieldrahtkammern, die zwischen denen sich jeweils ein Bleikonverter mit der Dicke von zwei Strahlungslängen befindet. Die erste Drahtkammer wird dabei als *Preshower*-Detektor, die beiden folgenden werden als *Postshower*-Detektoren bezeichnet (Abbildung 2.4). Um eine Ortsinformation zu erhalten, sind die Kathodenoberflächen segmentiert: Ein Ebene des SHOWER Detektors besteht aus 32×32 Pads. Bei drei Ebenen und 6 Segmenten kommt man folglich auf eine Gesamtzahl von über 18000 Pads. Die Drahtkammern werden im sogenannten Self-Quenching Streamer Mode (SQS-Mode) betrieben [9]. In diesem Modus ist die durch ein hindurchgehendes Teilchen erzeugte Ladung nahezu unabhängig von dessen Energieverlust. In einer im SQS-Mode betriebenen Drahtkammer ist demnach die erzeugte Ladungsmenge in erster Näherung nur von der Zahl der hindurchgehenden Teilchen abhängig.

Der TOF (**T**ime **o**f **f**light) Detektor besteht aus Plastik-Szintillator-Stäben, die wie in Abbildung 2.5 angeordnet sind. Zur Auslese eines solchen Szintillators werden an beiden Enden Photomultiplier angebracht, wodurch eine genaue Ortsbestimmung durchgeführt werden kann. Diese Detektorkomponente ermöglicht haupsächlich eine Teilchenidenti-

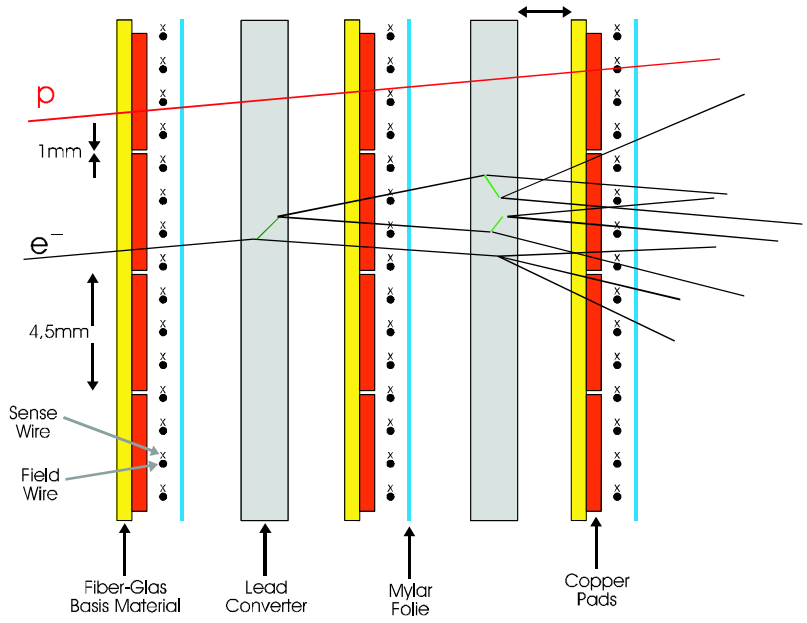


Abbildung 2.4: *Querschnitt durch den SHOWER-Detektor. Zu sehen sind die einzelnen Drahtkammern, die durch den Bleikonverter getrennt sind. Außerdem ist die Entwicklung eines elektromagnetischen Schauers durch ein Lepton skizziert. Die schwereren Protonen und Pionen dagegen lösen keinen Schauer aus.*

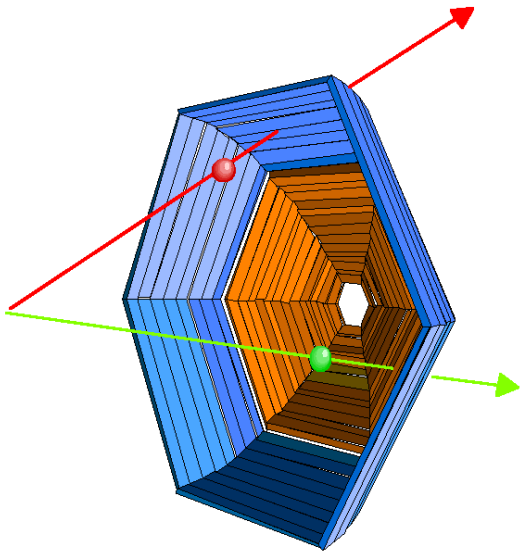


Abbildung 2.5: *Anordnung der Szintillatoren des TOF-Detektors. Die Szintillatoren werden an beiden Enden mit Photomultipliern versehen, wodurch eine Ortsbestimmung eines Hits ermöglicht wird.*

fizierung über Flugzeitmessung. Zusätzliche Informationen können gewonnen werden, wenn man die im Szintillator deponierte Energie bestimmt (über die Pulshöhen der Photomultiplier).

Kapitel 3

Der Second Level Trigger

3.1 Übersicht und Triggerkonzept

Um eine hinreichende Statistik zu erhalten wird HADES mit Strahlintensitäten von bis zu $10^8 pps$ arbeiten müssen. Bei Einsatz eines Targets mit 1% Wechselwirkungslänge werden dabei im Mittel 10^6 Reaktionen pro Sekunde stattfinden. Würde man alle Daten, die die einzelnen Detektoren dabei produzieren, speichern wollen, müsste man Datenraten von ca. 3GByte/s verarbeiten. Dieses ist zwar technisch möglich, wäre allerdings mit einem erheblicheb finanziellen Aufwand verbunden. Da aber nur etwa 0.1 Dileptonen pro Sekunde mit $m_{inv} > 500 MeV/c^2$ erzeugt werden, drängt sich die Verwendung eines effizienten Triggers zum Filtern der Rohdaten geradezu auf.

In Abbildung 3.1 findet man eine Übersicht über das für HADES geplante Triggersystem. Die Entscheidung, ob die Daten eines Events physikalisch relevante Informationen enthalten könnten, und somit gespeichert werden sollen, wird in 3 Stufen gefällt:

1. Eine erste Datenreduktion soll durch die Selektion zentraler Stöße geschehen, denn nur bei diesen werden hinreichen hohe Baryonendichten erzeugt. Erkannt werden die zentralen Stöße dadurch, daß viele der Nukleonen an Reaktionen teilnehmen und dabei geladene Teilchen in den ganzen Raumwinkel emittiert werden. Bei peripheren Stößen nehmen im Gegesatz dazu nur wenige Nukleonen an einer Reaktion teil, die meisten bewegen sich unbehelligt weiter. Als Bedingung für diesen First Level Trigger nimmt man deshalb die Multiplizität geladener Teilchen im TOF. Dadurch soll die zu verarbeitende Datenmenge bereits um den Faktor 10 reduziert werden.

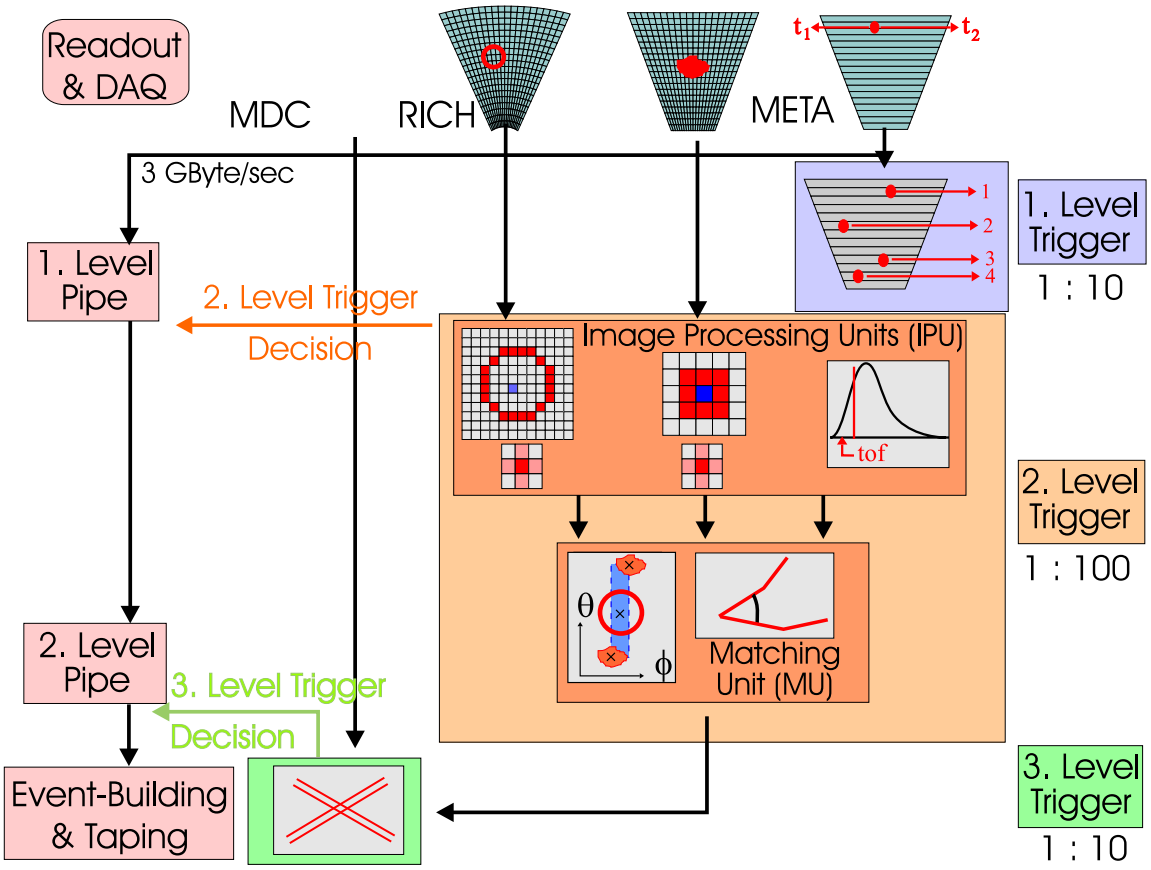


Abbildung 3.1: Übersicht über das HADES Trigger System: Die Datenreduktion erfolgt in drei Stufen, wobei die zweite Trigger-Stufe, bestehend aus den IPUs (**I**mage **P**rocessing **U**nit) und der Matching Unit, den größten Anteil verrichtet.

Nach einer positiven First Level Trigger werden die Daten der einzelnen Detektoren in der *First Level Pipe* zwischengespeichert, bis der Second Level Trigger eine Entscheidung trifft.

2. In der zweiten Triggerstufe, dem Second Level Trigger, benutzt man zusätzlich die Informationen von RICH und SHOWER, um Dileptonenkandidaten zu identifizieren. Dabei wird zum einen die segmentierte Photokathode des RICH nach Ringen durchsucht, zum anderen wird anhand der Flugzeitinformation vom TOF entschieden, ob es sich um Elektronen handeln könnte. Bei Polarwinkeln unter 45° werden zusätzlich die SHOWER Informationen ausgewertet, da hier keine klare Trennung von Hadronen und Leptonen aufgrund der Flugzeit möglich ist. (Lorentz-Boost). Die Signale jedes Detektors werden dabei in einer getrennten IPU (**I**mage **P**rocessing **U**nit) verarbeitet. Die Ergebnisse dieser IPUs werden dann an die Matching Unit weitergereicht, die zuerst kontrolliert ob die Winkelinformation aus den gefundenen Ringen mit der Position der TOF-Hits bzw. der e-m-Shower konsistent sind, und dann nach einem Di-Leptonenpaar mit adäquater invarienter Masse sucht. Falls alle Bedingungen erfüllt wurden, wandern die Daten aus der First in die Second Level Pipe. Andernfalls werden sie verworfen. Simulationen sagen eine Datenreduktion um den Faktor 100 durch diese Triggerstufe voraus. Der Second-Level-Trigger wird im folgenden Kapitel näher beschrieben.

3. Bevor eine endgültige Entscheidung getroffen wird, ob ein Event gespeichert werden soll, werden die MDC-Daten vom Third Level Trigger auf das Vorhandensein von Teilchenspuren, die von Leptonen stammen könnten, untersucht. Dadurch sollen falsche Zuordnungen zwischen Drahtkammertreffern und Teilchenspuren eliminiert werden. Durch diese Triggerstufe wird eine weitere Datenreduktion um den Faktor 10 erwartet.

Um den hohen Datendurchsatz von ca 3GB/s, der durch die hohe Strahlintensität bedingt ist, bewältigen zu können, basiert das HADES Triggersystem auf einem Konzept mit massiv parallel arbeitenden Prozessoren. Zusätzlich sollen die in den Triggerprozessoren implementierten Algorithmen eine pipeline-artige Struktur aufweisen. Das bedeutet, das langwierige Prozesse in kleinere Unterprozesse unterteilt werden, die dann sequentiell abgearbeitet werden. Die Daten werden dabei von Prozeß zu Prozeß weitergereicht, so

daß langwierige Speicherzugriffe auf Daten und eventuelle Zwischenergebnisse vermieden werden können. Ein Nachteil dieser Technik ist die relativ große Propagationszeit der Daten vom Detektor bis zum Bandlaufwerk und der damit verbundenen Speicherplatzbedarf: Für den HADES 2.Level Trigger z.B. erwartet man eine Laufzeit (Latency) von ca $200\mu s$, was bedeutet, daß dessen Speicher eine Kapazität für etwa 20 Events aufweisen muß.

Im folgenden Abschnitt werden die einzelnen Komponenten des Second Level Triggers etwas näher beschrieben. Besonderes Gewicht liegt hierbei auf den drei Image Processing Units, für die im Rahmen dieser Diplomarbeit ein Testsystem entwickelt werden sollte.

3.2 Der Triggerbus

Für die Kommunikation zwischen den einzelnen Triggerstufen steht der *Triggerbus* zur Verfügung. In Abbildung 3.2 ist dessen Bedeutung dargestellt. Falls ein First Level Trigger ausgelöst wird, verteilt die CTU (**C**entral **T**rigger **U**nit) ein Signal über den First Level Triggerbus an die DTUs (**D**etector **T**rigger **U**nits) welche in unmittelbarer Nähe der Readout-Systeme plaziert sind. Eine detektorspezifische Add-On Karte ermöglicht eine Kommunikation zwischen selbigen. Über den Triggerbus wird ein *Trigger Code* und der *Trigger Tag* übermittelt. Mit dem Trigger Code wird den Readoutsystemen mitgeteilt, um was für eine Art Trigger es sich handelt: neben einem „echten" Trigger können so auch verschiedene Test-Trigger erzeugt werden, auf die die Readout-Systeme unterschiedlich reagieren (z.B. mit der Ausgabe spezieller Test-Daten anstelle der echten Eventdaten). Der Trigger Tag wird den übertragenen Daten angehängt und dient dazu, zusammengehörige Datenpakete der einzelnen Detektoren zu kennzeichnen. Das soll verhindern, daß die Daten zweier verschiedener Events vermischt werden(z.B. durch einem Fehler in den Pipes). Im normalen Betrieb veranlassen die DTUs, daß die Detektoren ausgelesen werden, und die entsprechenden Daten in die Fist-Level-Pipe geschrieben werden.

Falls der Second-Level Trigger eine Entscheidung fällt (d.h. die Matching Unit), übermittelt diese ein Signal an die CTU, welche dann je nach Fall einen positiven oder negativen Second Level Trigger (=Trigger Code) über den Second Level Triggerbus verteilt. Ein negativer Second-Level Trigger hat zur Folge, daß die entsprechenden Daten verworfen werden, bei einer positiven Entscheidung werden sie jedoch in die Second Level Pipe

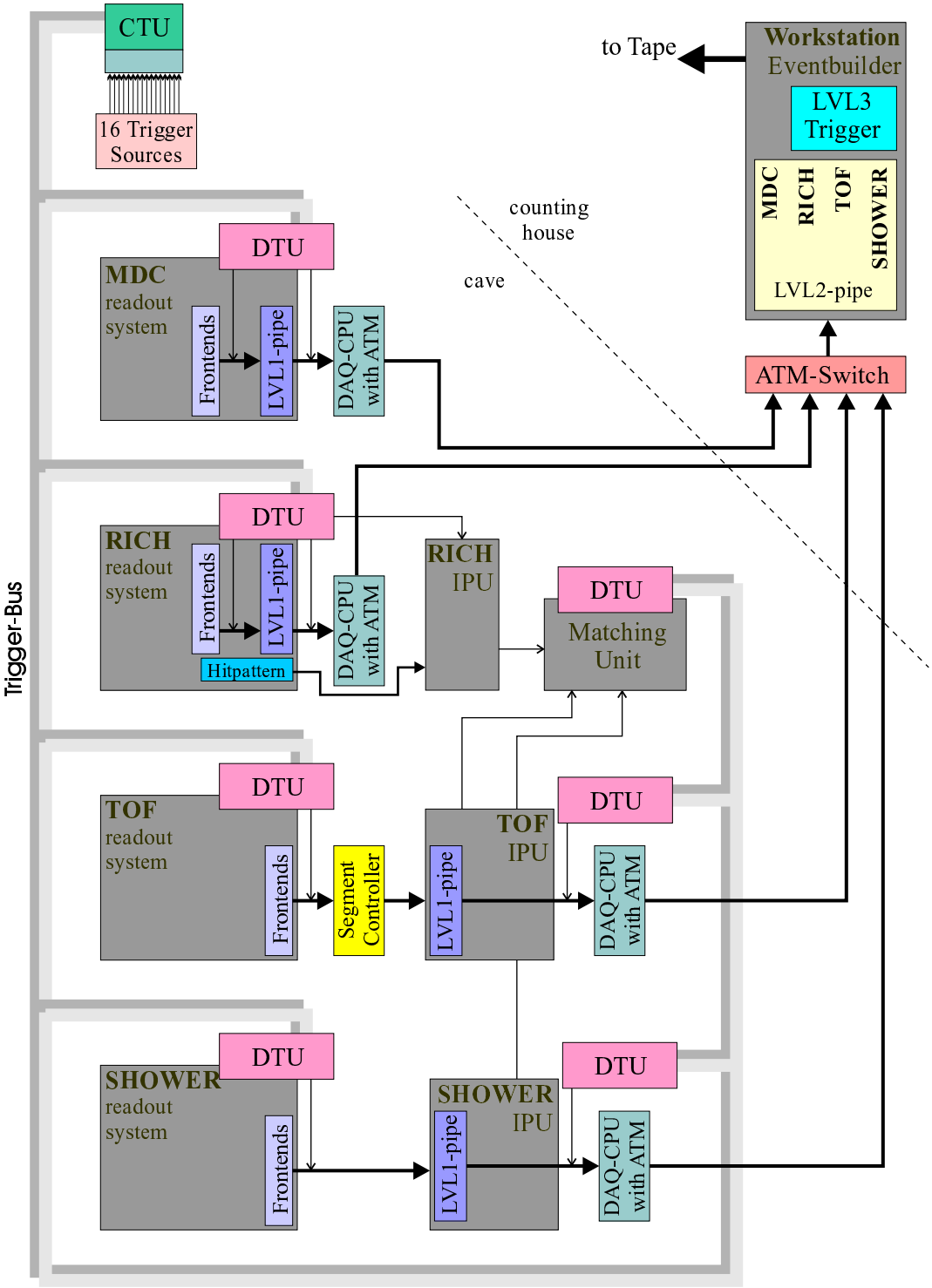


Abbildung 3.2: *Hier wird die Bedeutung des Triggerbus für die einzelnen Komponenten des HADES Triggersystems veranschaulicht. Die CTU verteilt die Trigger-Entscheidung über den Triggerbus an die DTUs, die die Information für die Readout-Systeme, bzw. die Trigger-Komponenten bereitstellt.*

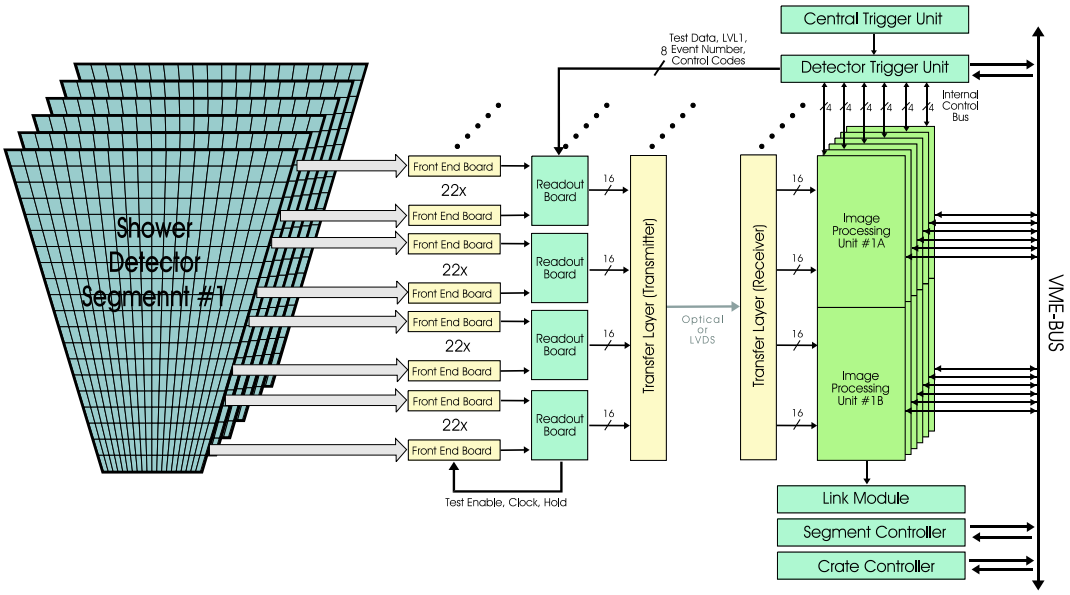


Abbildung 3.3: *Darstellung des Datenflusses von den SHOWER-Detektor-Segementen zu den zugehörigen Image Processing Units (IPUs)*

transportiert.

3.3 Die Triggerprozessoren

3.3.1 Die SHOWER IPU

Die einzelnen Pads des SHOWER-Detektors werden mittels twisted Pair Kabel zeilenweise mit einem „Front End Board" verbunden. Dieses Board enthält einen speziell entwickelten ASIC (**A**pplication **S**pecific **I**ntegrated **C**ircuit) welcher die Detektor-Daten verstärkt und im Verhältnis 32:1 multiplext. Von diesem gehen die noch analogen Daten weiter an den *Readout Controller* wo sie nochmals im Verhältnis 12:1 gemultiplext werden, bevor sie digitalisiert werden. Die digitalisierten Daten werden weitergereicht zur IPU, wo die Shower Erkennung stattfindet. Der Datenfluß von den Detektoren zu den IPUs ist in Abbildung 3.3 dargestellt.

Die eigentliche Elektronenidentifizierung geschieht durch einen Vergleich der erzeugten Ladung im Pre- und in den Postshower Detektoren. Elektronen sollten im den Bleikonvertern einen elektromagnetischen Schauer erzeugen, der sich durch einen Zuwachs der Ladungsmenge in den Postshower-Detektoren bemerkbar macht. Die Padgröße der

SHOWER-Detektoren ist so bemessen, das im Falle eines Treffers ein Feld von 3×3 Pads betroffen ist.

Um einen elektromagnetischen Schauer zu finden geht die IPU deshalb wie folgt vor: Falls die Ladungsmenge $q_{i,j}$ eines Pads (i, j) einen gewissen Schwellwert überschreitet, werden sowohl im Pre- als auch im Postshower-Detektor die Ladungssummen dieses Pads mit seinen nächsten Nachbarn gebildet:

$$Q^P(i,j) = \sum_{k=i-1}^{i+1} \sum_{l=j-1}^{j+1} q^P_{k,l}$$

wobei $P = Pre, Post_1, Post_2$. Sollte nun die Ladungssumme in einem der Postshower Detektoren größer sein als im Preshower Detektor zuzüglich eines Schwellwertes T:

$$Q^{Post_1} > Q^{Pre} + T \qquad \vee \qquad Q^{Post_2} > Q^{Pre} + T$$

so wird dieses Pad als Zentrum eines elektromagnetischen Schauers identifiziert. Zusätzlich wird noch verlangt, daß im Preshower Detektor die Ladungsmenge dieses Pads größer ist als die jeweilige Ladungsmenge seiner 4 nächsten Nachbarn (lokales Maximum).

Aufgrund der hohen Eventrate ($10^5/s$) und der großen Anzahl der zu analysierenden Pads (> 18000) muß die Analyse der SHOWER-Daten in massiv parallel arbeitenden Prozessoren mit *Pipeline-Architektur* vorgenommen werden. Diese Pipeline-Architektur läßt sich hervorragend in FPGAs (**F**ield **P**rogrammable **G**ate **A**rray) implementieren. Wie diese Implementierung im einzelnen aussieht, ist in [10] nachzulesen.

3.3.2 Die RICH IPU

Jedes der 6 Segmente des RICH beinhaltet 6560 Pads. Jeweils 64 dieser Pads werden im Falle eines positiven First-Level Triggers von einem PFM (**P**rogrammable **F**rontend **M**odule) ausgelesen. Diese Frontend Module bestehen im wesentlichen aus einem GASSIPLEX-Chip [1], der die analogen Daten der Photokathode digitalisiert, einem FPGA, der den Auslesevorgang kontrolliert und zwei FiFos (**F**irst **I**n **F**irst **O**ut: Speichertyp). Einer dieser FiFos stellt die First-Level-Pipe dar, in der sowohl Koordinaten als auch Pulshöheninformationen der einzelnen Pads nach einem positiven First-Level Trigger gespeichert werden. In dem anderen FiFo werden nur die Koordinaten der getroffenen Pads für die Weiterverarbeitung durch die RICH-IPU gepuffert. In beiden FiFos werden nur die Pads gespeichert, deren Pulshöhe einen programmierbaren Schwellwert überschreitet. Dieser Wert ist

[1] Ein ladungs-sensitiver Vorverstärker [11]

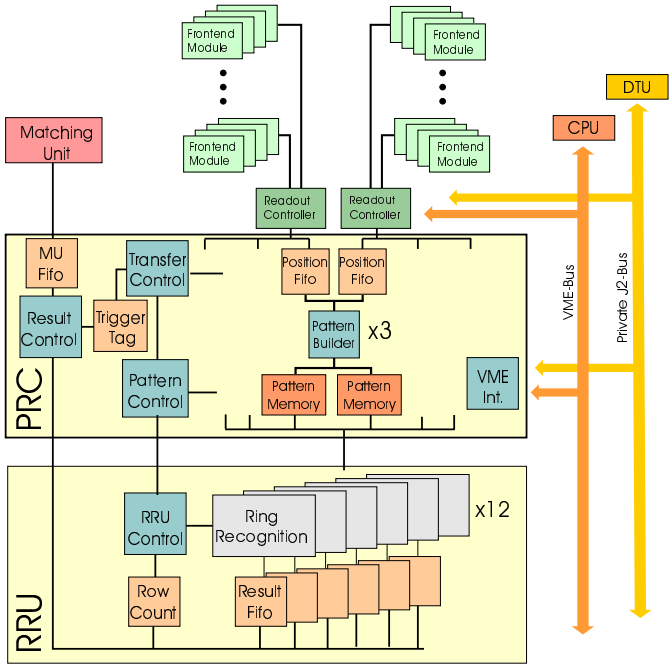


Abbildung 3.4: *Schematischer Überblick über die RICH-IPU. Die ankommenden Hit-Koordinaten werden durch die PRC (Pattern Reconstruction Card) in ein Hit-Pattern transformiert. Die Ringerkennung findet in der RRU (Ring Recognition Unit) statt.*

für beide Kanäle getrennt einstellbar. Bis zu 5 dieser Frontend-Module werden zu einer Daisy-Chain [2] zusammengefaßt, 8 dieser Daisy-Chains werden dann von einem Readout Controller ausgelesen. Dieser liefert letztendlich die Hit-Koordinaten an die IPU.

Die RICH-IPU setzt sich aus zwei Teilen zusammen: eine PRC (**P**attern **R**econstruction **C**ard) und eine RRU (**R**ing **R**ecognition **U**nit). Die PRC dient dazu,um aus den Hit-Koordinaten, die sie von den Readout Controllern erhält das Hit-Pattern zu rekonstruieren. Dieses recht aufwendige Verfahren wurde gewählt, weil hier die Übertragung bei einer mittleren Belegung von 47 Pads/Event/Segment wesentlich schneller vonstatten geht als die Übertragung des kompletten Hitpatterns [12]. Außerdem ermöglicht es eine flexiblere Konfiguration des Readout-Schemas.

Nachdem das Hitpattern rekonstruiert worden ist, erfolgt die Ringsuche durch die RRU. Ähnlich wie der Algorithmus zur Showersuche ist auch der hier verwendete Algorithmus sehr gut in FPGAs implementierbar. In der vorliegenden Version der RRU wer-

[2] sequentielles Beschreiben oder Auslese einer Kette von verschalteten Bauelementen, wobei ein *activity strobe* von einem Baustein zum Nächsten durchgereicht wird.

den für ein Detektorsegment 8 FPGAs vom Typ XC4028EX der Firma Xilinx verwendet. Jeder dieser Bausteine bearbeitet dabei 12 Spalten der Detektorebene.

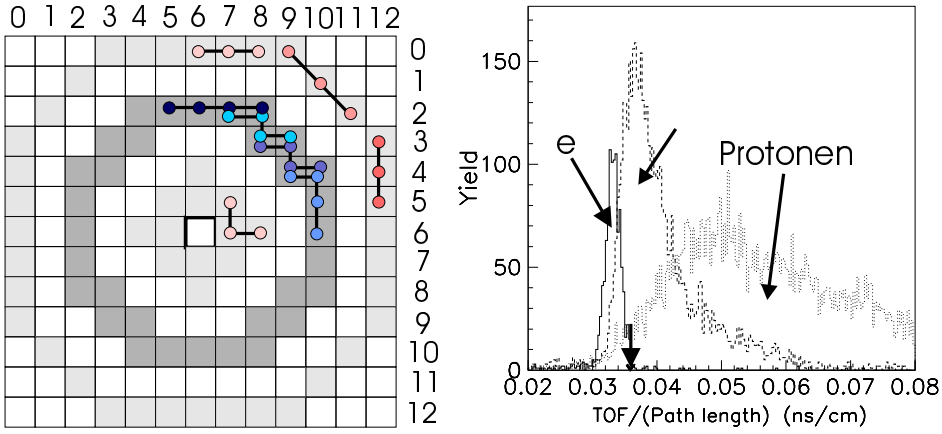
In Abbildung 3.5(a) ist der zur Ringsuche verwendete Algorithmus dargestellt. Die Ringsuche findet auf einer (12×12) Pads umfassenden Fläche statt, wobei die gesuchten Ringe einen Durchmesser von 8 Pads haben sollten. Diese Fläche wird unterteilt in eine *Veto-Region* (hellgrau) und in ein *Ring-Region* (dunkelgrau). In der ersten Stufe werden dann 3-4 Pads innehalb dieser Regionen durch eine Oder-Verknüpfung miteinander korreliert (In Abbildung 3.5(a) für $1/4$ des Rings durch eine Verbindungslinie dargestellt). Anschliessend werden die Ergebnisse dieser Verknüpfung (0 oder 1) für beide Regionen gertrennt aufsummiert (Für die Ring-Region erhält man so z.B. einen Wert zwischen 0 und 16). Danach werden die beiden Summen noch mit programmierbaren Schwellwerten verglichen: es wird eine minimale Trefferzahl in der Ringregion verlangt, während gleichzeitig nicht zu viele Pads in der Veto-Region getroffen sein dürfen. Da bei diesem Algorithmus in der Regel mehrere benachbarte Pads als Zentrum eines Ringes in Frage kommen, wird anschliessend noch eine lokale Maximumsuche durchgeführt, um den besten Kandidaten zu finden.

Auf Simulationsdaten angewandt wurden mit diesem Algorithmus mehr als 90% der Ringe korrekt erkannt, pro simuliertem Event wurden dabei im Mittel 0.3 Nicht-Ringe fälschlicherweise als Čerenkov-Ringe identifiziert [12].

3.3.3 Die TOF IPU

Der TOF wird nicht nur zur Generierung des First Level Triggers verwendet, sondern hilft auch durch Messung der Flugzeit bei der Erzeugung des Second Level Triggers. Zu diesem Zweck müssen die Photomultiplier der rund 1000 Szintillatoren ausgelesen werden. In Verbindung mit einem Startzähler wird aus ihnen dann als erstes die Flugzeit bestimmt. Aufgrund von Laufzeitunterschieden in der Verkabelung müssen diese Werte in einem ersten Schritt erst einmal kalibriert werden. Da jeder Szintillator an beiden Enden ausgelesen wird, kann nun als nächstes die genaue Position des Hits festgestellt werden, und damit auch die Länge des Weges, die das Teilchen zurückgelegt hat. Dabei wird allerdings die Ablenkung im Magnetfeld nicht berücksichtigt. Aus diesen Informationen kann dann die Geschwindigkeit des Partikels ermittelt werden.

In Abbildung 3.5(b) ist eine Geschwindigkeitsverteilung dargestellt, wie sie aufgrund



(a) Der Ring-Such Algorithmus (b) Ein Flugzeit-Spektrum

Abbildung 3.5: *In (a) ist der zur Ringsuche verwendete Algorithmus dargestellt. Eine Fläche von* 12×12 *Pads wird in eine Ringregion (dunkelgrau) und eine Vetoregion (hellgrau) unterteilt. Ein simuliertes Flugzeitspektrum ist in (b) zu sehen. Durch einen Cut (bei* $\approx 0.035 ns/cm$*) kann der hadronische Teil recht gut isoliert werden.*

von Simulationen erwartet wird: links der Elektronenpeak, daneben die Pionen und ganz rechts die Protonen-Beiträge. Wie man erkennen kann, läßt sich der hadronische Untergrund durch eine entsprechende Bedingung ($TOF \leq 0.035 ns/cm$) recht gut diskriminieren.

Ihre Daten bekommt die TOF-IPU von den *Segment-Controllern*, die für die Auslese der ADC- und TDC-Boards verantwortlich sind.

3.4 Die Matching Unit

Die Matching Unit (MU) ist die Komponente der zweiten Triggerstufe welche letztendlich für die Erzeugung eines Second Level Triggers verantwortlich ist. Für diese Aufgabe werden die Ergebnisse der 3 IPUs herangezogen. Der geplante *Matching-Algorithmus* [13] sieht wie folgt aus (vgl. Abbildung 3.6):

1. Als erstes versucht die MU zu jedem gefundenen RICH-Ring einen passenden Hit im META zu finden und daraus die Flugbahn des Teilchens zu rekonstruieren.

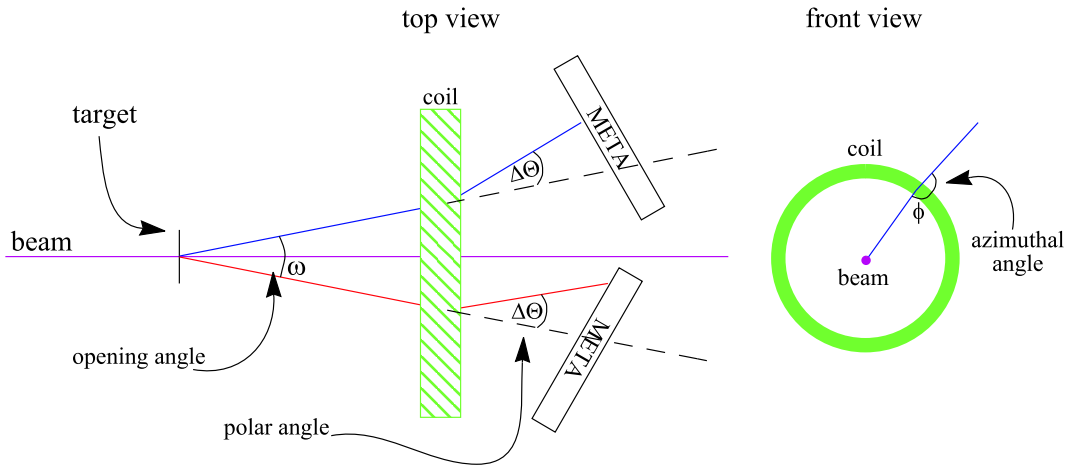


Abbildung 3.6: *Schematische Darstellung der Aufgaben der Matching Unit (MU). Sie sucht nach konsistenten Treffern in RICH und META und prüft, ob die invariante Masse der Leptonenpaares hinreichend groß ist.*

2. Da nach e^+e^--Paaren gesucht wird, müssen in den gefundenen Bahnen welche mit entgegengesetzter Krümmung ($\sim \Delta\theta$) zu finden sein.

3. Den durch *Dalitz-Zerfälle* ($\pi^0 \rightarrow \gamma e^+e^-$) erzeugten Untergrund kann man dadurch reduzieren, daß man eine untere Schwelle für den Öffnungswinkel ω fordert.

4. Für die bis jetzt übriggebliebenen Leptonenpaare wird die invariante Masse bestimmt. Dazu wird Impuls jedes Leptons aus seinem Ablenkwinkel ermittelt. Dann kann man die invariante Masse des Dileptons mit Hilfe des Öffnungswinkels ω bestimmen:

   $$m_{inv} \approx 2 \sin\left(\frac{\omega}{2}\right) \sqrt{p_{e^+} p_{e^-}}$$

   Für einen positiven Trigger muß die invariante Masse innerhalb eines festzulegenden Fensters liegen.

Falls die MU einen einen möglichen Dileptonen-Kandidaten entdeckt, wird ein Second Level Trigger ausgelöst und die Daten des betreffenden Event-Daten propagieren in die Second-Level Pipe.

Kapitel 4

Das Testsystem für den Second Level Trigger

4.1 Aufgabenstellung

Um den vollständig in Gießen entwickelten Second Level Trigger unabhängig vom restlichen Detektorsystem testen zu können, sollte im Rahmen dieser Arbeit ein System entwickelt werden, mit dem ein solcher Test unter nahezu realistischen Bedingungen möglich ist. Dieses System sollte ein Interface zum First Level Triggerbus haben, und in der Lage sein sämtliche für den Second Level Trigger relevanten Readout-Systeme zu emulieren, d.h. den einzelnen IPUs sollten Daten aus Simulationen oder Testexperimenten mit vergleichbaren Raten wie im späteren Experiment zur Verfügung gestellt werden.

4.2 Realisierung

4.2.1 Hardware-Konzept

Als Basis für dieses Testsystem wurde ein handelsüblicher PC gewählt. Um die Anbindung an den Second Level Trigger zu ermöglichen, mußte für diesen PC eine zusätzliche Hardware entwickelt und produziert werden. In Abbildung 4.1 ist diese Idee noch einmal anschaulich dargestellt: Die Readout-Systeme der einzelnen Detektoren können durch einen PC mit spezieller Hard- und Software ersetzt werden (vgl. Abbildung 3.2). Die Hardware in Form einer PCI-Erweiterungskarte ist mit einem Anschluß an den Trigger-

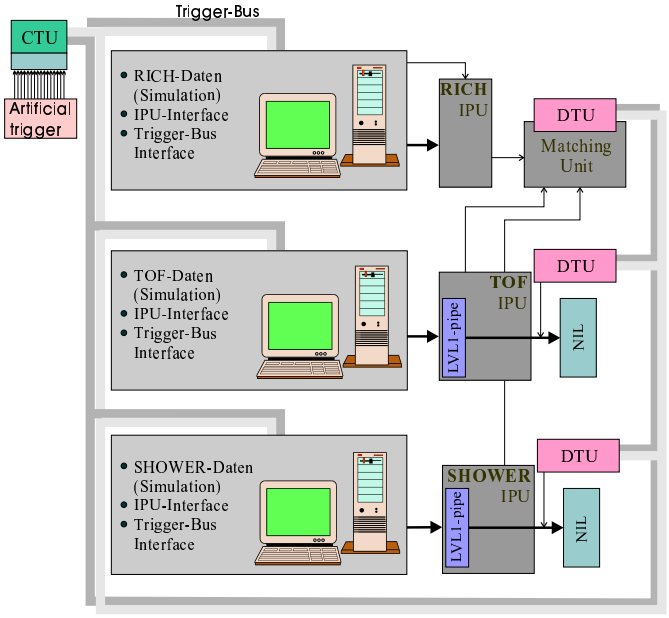


Abbildung 4.1: *Grundgedanke eines Trigger Test Systems: Die einzelnen Readout-Systeme können durch einen PC mit entsprechender Hard- und Software ersetzt werden (vgl. Abbildung 3.2).*

Bus versehen und kann, falls sie einen First-Level Trigger erhält, die IPUs über IPU-spezifische Add-On Karten mit Daten versorgen. In Abbildung 4.2 ist eine schematische Übersicht über die PCI-Karte zu sehen. Die Event-Daten werden in vier Memory-Modulen gespeichert, die von einem *Board Controller*, der in einem programmierbaren Baustein implementiert ist, angesteuert werden. Dieser ist auch für die Kontrolle aller anderen Komponenten zuständig. Wenn die Add-On Karte einen First Level Trigger empfängt, holt sie sich die Daten aus den FiFos. Falls diese halb leer sind, setzen sie ein Flag, welches den Board Controller veranlaßt wieder neue Daten nachzuschieben. Auf diese Weise wird der Datentrom zu den IPUs vom den Speicher-Modulen entkoppelt, wodurch die erforderlichen Steuerungszugriffe auf den Speicher wirkungsvoll versteckt werden können. In den folgenden Abschnitten folgt eine nähere Beschreibung der verwendeten Hardware-Komponenten.

4.2.1.1 Der PCI Bus

Der PCI-Bus (**P**eripheral **C**omponent **I**nterconnect) ist ein non-proprietärer Bus Standart, dessen Entwicklung von Intel initialisiert wurde. Heute ist die PCISIG (**PCI** **S**pecial

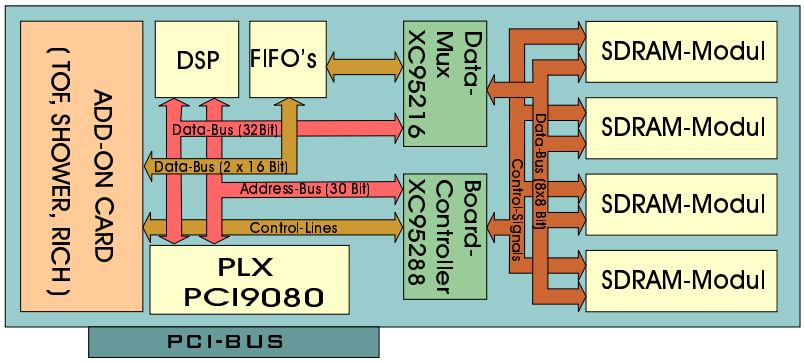


Abbildung 4.2: *Hardware Blockdiagramm. Die PCI-Karte enthält vier SDRAM-Module (**S**ynchronous **D**ynamic **R**andom **A**ccess **Memory**), zwei CPLDs (**C**omplex **P**rogrammable **L**ogic **D**evices) (XC9500 von Xilinx) zur Steuerung, einen DSP (**D**igital **S**ignal **P**rocessor), FiFos und ein Interface zu einer IPU-spezifischen Add-On-Karte.*

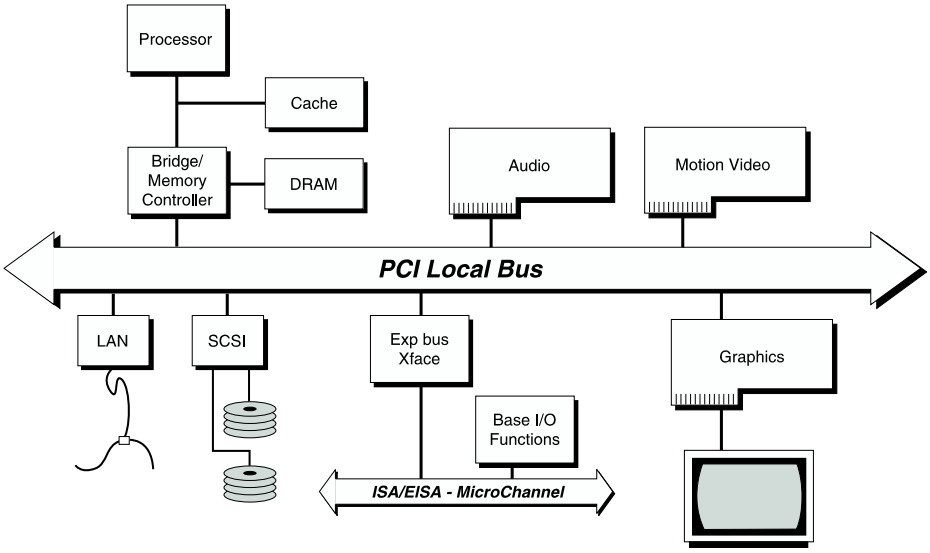


Abbildung 4.3: *Die PCI-Bus Architektur in modernen Computer-Systemen. Eines der wichtigsten Merkmale ist die klare Trennung des Hauptspeichers von der Peripherie.*

Interest **G**roup), ein Konsortium der führenden Computer-Hersteller, für die Weiterentwicklung dieses Bus-Systemes zuständig. Abbildung 4.3 veranschaulicht die Rolle des PCI-Bus in modernen Computersystemen. Einsteckplätze für PCI-Karten sind heute in fast jedem Computer vorhanden, deshalb bot sich die Entwicklung einer PCI-Karte geradezu an.

Aufbau Der PCI-Bus ist ein 32 Bit breiter Bus, der nach der derzeitig aktuellen Spezifikation [14] mit einer beliebigen Frequenz zwischen 0 und 33 MHz betrieben werden darf. Diese Spezifikation ist sowohl für 3.3V (LVTTL) als auch für 5V (TTL) Signalpegel gültig. Für die Spannungsversorgung sollten Anschlüsse mit 3.3V,5V und $\pm$12V zur Verfügung stehen, die maximale Leistungsaufnahme einer Karte ist dabei auf 25W begrenzt. Um die erforderliche Leistung der Bus-Treiber zu minimieren werden die Bus-Leitungen nicht terminiert, sondern man benutzt das sog. *reflected wave switching*. Aus diesem Grund mußte die maximale Zahl von Karten an einem Bus auf 4 beschränkt werden, die Benutzung einer größeren Anzahl von Karten erfordert die Verwendung einer *PCI-to-PCI Bridge*. Darüberhinaus kann jede einzelne PCI-Karte bis zu 4 unabhängige Funktionsgruppen besitzen, die getrennt ansprechbar sind.

In Abbildung 4.4 sind die verwendeten Signale dargestellt. Für jede Karte erreichbar sind die Address und Datenleitungen und die Interface Kontroll Leitungen. Die Interuptleitungen sind für jedes Gerät getrennt vorhanden, auch die Arbitrierungsleitungen, die allerdings nur implementiert werden müssen, wenn die Karte die Kontrolle über den Bus übernehmen möchte (Master-Funktion).

Bus-Betrieb Auf jeder PCI-Karte muß ein Satz von definierten Registern (256 Byte) implementiert werden, die es dem Host-System ermöglichen den Bus nach einem Reset selbst zu konfigurieren. Diese Register enthalten im wesentlichen die Größe der benötigten IO- bzw. Memory-Bereiche, aber auch Informationen über den Hersteller (Vendor-ID) und die Karte selbst (Device-ID), die von der Software benutzt werden können. Zum Konfigurieren wird die betreffende Karte durch das setzen der IDSEL-Leitung ausgewählt. Dann werden die Größen der benötigten Speicherbereiche ausgelesen. Nun kann der Host jedem Agenten aus dem 4 GigaByte grossen Addressraum einen Bereich zuweisen, ohne daß es Überschneidungen gibt. Nach der Konfiguration muß dann jede Karte selbst entscheiden, ob sie angesprochen wird, die IDSEL-Leitung ist während des normalen Betriebes nicht

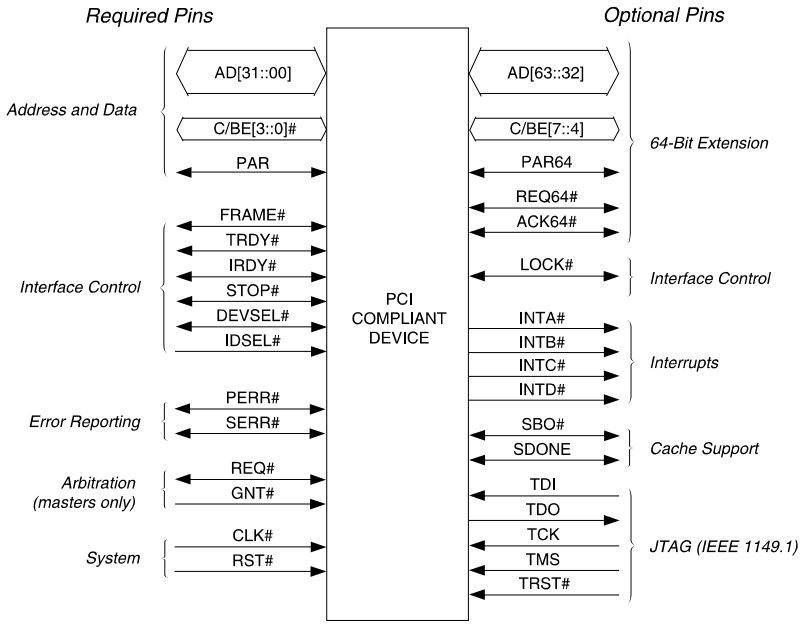


Abbildung 4.4: *Die Signale des PCI-Bus. Auf der linken Seite befinden sich die Signale, die jeder PCI-Agent haben muß, auf der rechten Seite sind die optionalen Leiungen aufgeführt.*

aktiv.

Der PCI-Bus ist ein *Burst*-orientierter Bus: Um den Geschwindigkeitsverlust der durch das *Multiplexen* von Daten und Addressen entsteht zu kompensieren, verzichtet man bei größeren Daten-Mengen auf die abwechselnde Übergabe von Addressen und Daten . Statt dessen wird nur einmal am Anfang eines Bursts die Startaddresse angegeben, in den folgenden Zyklen werden dann bis zum Ende des Bursts nur noch Daten geschickt. Dieses funktioniert allerdings nur wenn sowohl Target als auch Initiator [1] ihre Addressen in gleicher Weise inkrementieren (in der Regel linear).

In Abbildung 4.5 ist ein Beispiel für einen PCI Schreibzugriff dargestellt: Nachdem der Initiator den Bus zugeteilt bekommen hat, startet er einen Zugriff auf ein Target durch das herunterziehen der FRAME# [2] -Leitung. Bei der nächste steigenden Taktflanke(2) müssen die Addressen (AD) und das Kommando (C/BE#) anliegen. Falls ein Target erkennt das

[1] In PCI-Terminologie wird der aktuelle Bus-Master als *Initiator* bezeichnet, während mit *Target* die angesprochene Karte gemeint ist.

[2] Das Zeichen # kennzeichnet *low-active* Signale, die logische 1 wird durch Herunterziehen der entsprechenden Leitung auf 0V dargestellt.

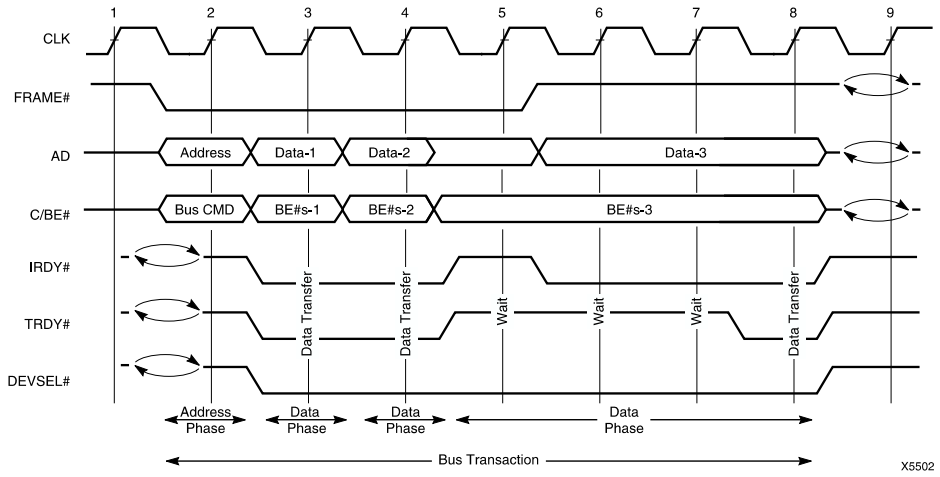


Abbildung 4.5: *Ein PCI-Schreibzugriff. Die Datenübertragugung erfolgt in Bursts: einer Addressphase folgen ein oder mehrere Datenphasen.*

auf ihn zugegriffen wird, muß er dies durch das aktivieren der DEVSEL#-Leitung anzeigen. Dafür hat das Target bis zu 3 Taktzyklen Zeit, ansonsten muß der Initiator den Zugriff abbrechen. Falls nun die beiden Signale IRDY# (Initiator Ready) und TRDY# (Target Ready) auf 0 gezogen sind, wird mit jedem Taktzyklus ein Datenwort übermittelt. Durch C/BE# werden die gültigen Bytes gekennzeichnet. Sowohl Target als auch Initiator können den Transfer für maximal n (konfigurierbar) Taktzyklen unterbrechen (*Wait-States*), ohne das eine neue Addresse übermittelt werden muß. Das Ende einer Übertragung wird angezeigt, indem der Initiator die FRAME#-Leitung wieder auf 1 setzt, was vor dem letzten übertragenen Datenwort geschehen muß. Detailliertere Beschreibung findet sich in [15] und [16].

4.2.1.2 PCI-Interface

Da Implementierung der PCI-Spezifikationen in Hardware, z.B. einem FPGA, aufgrund der sehr strengen zeitlichen und kapazitiven Kriterien mit hohem Aufwand verbunden wäre, wurde für diese Arbeit eine fertige Lösung gewählt: der PCI9080 von PLX-Technologie. Altenativ zu dieser Lösung bieten aber auch die Hersteller programmierbarer Logik mittlerweise Programmmodule an, mit denen die Erfüllung sämtlicher Anforderungen möglich wäre. In Abbildung 4.6 ist der Aufbau dieses Chips zu sehen. Er enthält ein komplettes Master-fähiges PCI-Interface sowie ein Interface zu einem lokalen Bus, für welchen 3 Betriebsmodi vorgesehen sind. Durch den Einsatz von FiFos ist der lokale Bus vollständig

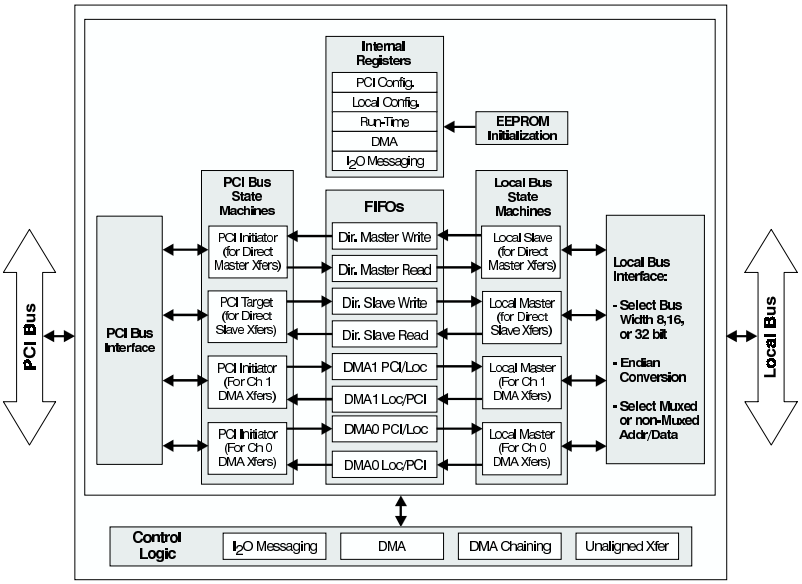


Abbildung 4.6: *Interner Aufbau des PCI-Interface Chips PCI9080 von PLX-Technology. Die wesentlichen Bestanteile sind ein PCI-Interface, ein konfigurierbares Interface zu einem lokalen Bus und die FIFOs, die beidseitige Burstzugriffe ermöglichen. Konfiguriert wird der Chip über ein EEPROM*

vom PCI-Bus entkoppelt, er kann dadurch insbesondere mit einer vom PCI-Takt abweichenden Frequenz betrieben werden, ohne daß die Möglichkeit von PCI-Bursts unterbunden wird. Weiterhin übernimmt dieser Chip das Remapping von PCI auf lokale Addressen, was problematisch selbst zu implementieren wäre, da die die PCI-Basis-Addressen erst während der Konfiguration zugewiesen werden. Die zur Konfiguration des PCI-Bus notwendigen anwendungsspezifischen Daten wie z.B. Größe der IO- und Memory-Bereiche können in einem externen EEPROM (**E**lectrical **E**rasable **P**rogrammable **R**ead-**O**nly **M**emory) gespeichert werden. Weiteregehende Informationen sind unter [17] zu finden.

4.2.1.3 SDRAM

In neueren Computersystemen kommt heute vorwiegend SDRAMs (**S**ynchronous **D**ynamic **R**andom **A**ccess **M**emory) zum Einsatz. Der Vorteil dieser SDRAMs gegenüber herkömmlichem DRAM (**D**ynamic **R**andom **A**ccess **M**emory) ist, daß alle Ein- und Ausgangssignale vom Systemtakt abhängen (d.h. alle Daten werden bei steigender Taktflanke

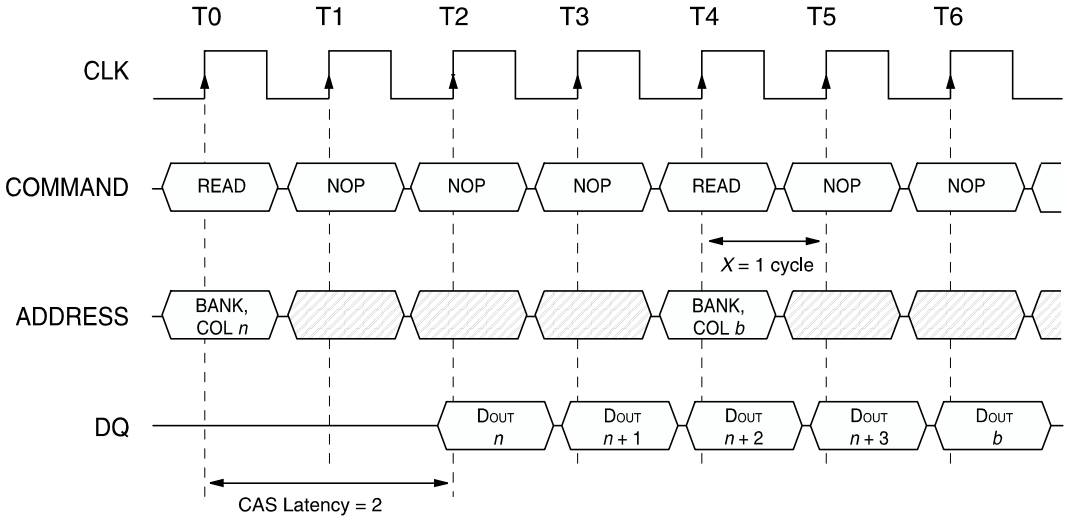


Abbildung 4.7: *Ein Lesezugriff auf ein SDRAM. Wie beim PCI-Bus werden die Daten in Bursts übermittelt.*

ubernommen) und nicht mehr Level-getrieben (Signale müssen mit Hilfe eines dedizierten Signals übergeben werden) sind. Durch zusätzliches internes Pipelining können die meisten Speicherbausteine mit einer Frequenz von bis zu 100 MHz betrieben werden. Ähnlich wie der PCI-Bus ist auch das SDRAM Burst-orientiert, wodurch sich eine weitere erhebliche Steigerung der Transferrate ergibt, falls größere, zusammenhängende Bereiche übertragen werden. Allerdings müssen wie beim herkömmlichen DRAM die Kondensatoren, in denen die Daten gespeichert werden, in regelmäßigen Abständen wieder aufgeladen werden. Darin besteht ein wesentlicher Nachteil gegenüber den SRAM (**S**tatic **R**andom **A**ccess **M**emory) da deswegen ein spezieller Controller benötigt wird. Der Nachteil des SRAMs liegt zum einen im relativ hohen Preis, zum anderen in der relativ geringen Packungsdichte gegenüber DRAMs (bzw. SDRAMs).

Die SDRAM-Speicherchips werden auf DIMM-Module montiert, die strengen mechanischen und elektrischen Spezifikationen genügen müssen [18]. Heute sind Module mit einer Kapazität von 512 MB erhältlich, d.h. bei Verwendung von 4 Modulen können auf der PCI-Karte bis zu 2GByte Daten für die IPUs gespeichert werden.

In Abbildung 4.7 ist ein Beispiel für einen Lesezugriff auf ein SDRAM zu sehen. Bemerkenswert ist die Ähnlichkeit zum PCI-Zugriff (Abbildung 4.5).

4.2.1.4 CPLDs

Um komplexere logische Funktionen zu implementieren, ist ein diskreter Aufbau mit Standart-Komponenten der TTL-, bzw.CMOS-Familie nicht sehr sinnvoll. Zum einen schränkt die begrentzte Funktionalität dieser Bausteine den Anwender sehr stark ein, zum anderen werden die logischen Verknüpfungen zwischen den einzelnen Modulen hardwaremäßig realisiert, wobei leicht nicht zu kompensierende Fehler entstehen können. Diese würden dann ein kostspieligen Neu-Entwurf der Hardware bedingen.

Eine Alternative dazu bietet der Einsatz von programmierbaren Logikbausteinen, von denen heute im wesentlichen zwei Familien benutzt werden: Zum einen die FPGAs (**F**ield **P**rogrammable **G**ate **A**rray) und die CPLDs (**C**omplex **P**rogrammable **L**ogic **D**evice), die sich durch ihre Architektur unterscheiden: CPLD-Bausteine basieren im wesentlichen auf EECMOS-Technologie, während FPGAs in der Regel auf SRAM-Strukturen basieren. Daher behalten CPLDs ihr Programm auch nach Abschalten der Spannung, wohingegen FPGAs stets neu programmiert werden müssen.

Die Programmierung dieser Bausteine kann entweder über eine schematische, graphische Beschreibung erfolgen oder aber durch eine *Hardware Description Language* wie z.B. ABEL oder VHDL erfolgen. Näheres zu programmierbarer Logik ist in [19] zu finden.

Für diese Arbeit wurden CPLDs der Firma XILINX gewählt, um einen Controller für das PCI-Board zu implementieren. Da abgesehen vom DSP alle Komponenten auf diesem Board synchron arbeiten, können die aufwendigsten Elemente wie z.B. der SDRAM-Controller als synchrone State-Machine implementiert werden, für die CPLDs besser als FPGAs geeignet sind, da mehr Signale pro Flip-Flop kombinatorisch miteinander verknüpft werden können. Außerdem bieten diese Bausteine die Möglichkeit die Signalpegel der Ausgänge durch eine getrennte Spannungsversorgung für die I/O-Blocks zu regeln (TTL oder LVTTL-Pegel). Dieses ist hilfreich, da das SDRAM nur mit LVTTL-Signalpegeln betrieben werden kann, die restlichen Komponenten aber TTL-Pegel ausgeben.

4.2.1.5 DSP

Optional wurde ein ADSP21060 von Analog Devices vorgesehen. DSPs (**D**igital **S**ignal **P**rocessor) wurden für schnelle Verarbeitung von Bild und Ton in der Multimediatechnik

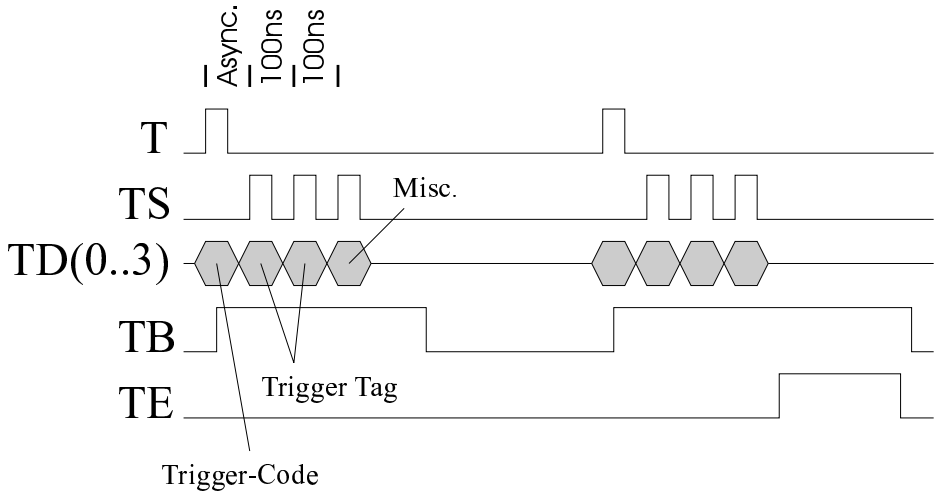


Abbildung 4.8: *Timing Diagramm für den Triggerbus. Nach dem asynchronen Trigger Signal (T) mit dem der Trigger Code übertragen wird, wird noch ein Trigger Tag, welches den entsprechenden Datenpaketen zur Identifizierung angehängt wird, weitergegeben.*

entwickelt, und zeichnen sich durch numerische Leistungsfähigkeit aus. Mit diesem Prozessor könnten später einmal schnelle, einfache Berechnungen durchgeführt werden, ohne den Hauptprozessor zu sehr zu belasten.

4.2.2 Beschreibung der Schnittstellen zu den IPUs

4.2.2.1 Interface zum Triggerbus

Die Entscheidungen der ersten und zweiten Triggerstufe werden über den Triggerbus verteilt, der aus folgenden Signalen besteht:

T: Asynchrones Trigger Signal, mit dem gleichzeitig der Trigger Code übertragen wird.

TS: Trigger Strobe, dieses Signal zeigt einen gültigen Trigger Tag an

TD(0..3): Datenleitungen für Trigger Code und Trigger Tag

TB: Trigger Busy. Verhindert einen weiteren Trigger während die Readout-Systeme mit der Auslese beschäftigt sind.

TE: Trigger Error

TS(0..1): Momentan unbenutzte Leitungen.

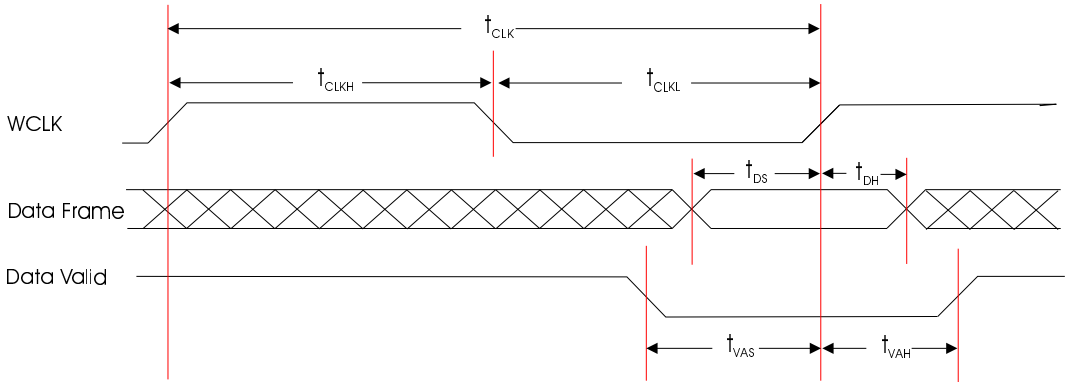


Abbildung 4.9: *Timing Diagramm für den Datanaustausch zwischen SHOWER Readout Boards und IPU. Die zeitlichen Vorgaben für eine Übertragung sind:* $t_{CLK} \geq 20ns$, $t_{CLKH}, t_{CLKL} \geq 10ns$, $t_{DS}, t_{DH} \geq 5ns$ *und* $t_{VAS}, t_{VAH} \geq 10ns$

Um Störungen zu unterbinden, werden diese Signale differentiell übertragen.

4.2.2.2 Interface zur SHOWER-IPU

Für den Datenaustausch zwischen den SHOWER Readout Boards und den entsprechenden IPUs wurden 20 Signale definiert:

D(15..0): Auf diesen Leitungen wird entweder die digitalisierte Ladungsmenge eines Pads oder ein Kontroll-Wort übertragen. (abhängig von EV-CTR(1..0))

EV-CTR(1..0): Diese beiden Bits charakterisieren D(15..0) als Datenwort (EV-CTR(1..0) = 00) oder als Kontrollwort (EV-CTR(1..0) = 01). Die Kombination EV-CTR(1..0) = 10 ist nicht definiert und kann für zukünftige Erweiterungen benutzt werden.

Data Valid: Dieses Low-Aktive Signal zeigt die Gültigkeit der Daten an.

Clk: Taktsignal. Falls Data Valid = 0 ist, werden die Daten von der IPU bei steigender Flanke dieses Signals übernommen.

Aufgrund des langen Signalweges von ca. 10 Metern wurden der LVDS [3]-Standart für die Datenübermittelung gewählt. Die zeitlichen Spezifikationen dieses Protokolls sind Abbildung 4.9 zu entnehmen.

[3] **L**ow **V**oltage **D**ifferential **S**ignal

Taktzyklus	D(15..8)	D(7..0)
i	Trigger Tag	RB Status Word
$i+1$	$Pre(m,n)$	$Pre(m,n+4)$
$i+2$	$Pre(m,n+1)$	$Pre(m,n+5)$
$i+3$	$Pre(m,n+2)$	$Pre(m,n+6)$
$i+4$	$Pre(m,n+3)$	$Pre(m,n+7)$
$i+5$	$Post1(m,n)$	$Post1(m,n+4)$
...	...	...
$i+8$	$Post1(m,n+3)$	$Post1(m,n+7)$
...	...	...
$i+383$	$Post2(m+31,n+2)$	$Post2(m+31,n+6)$
$i+384$	$Post2(m+31,n+3)$	$Post2(m+31,n+7)$
$i+385$	Contr. Wort	Contr. Wort
$i+386$	Prüfsumme	Prüfsumme

Tabelle 4.1: Struktur der Datenpakete für die SHOWER-IPU

In Tabelle 4.1 ist die Struktur der zu übertragenden Daten dargestellt: Ein Daten-Paket besteht aus 384 16-bit Datenworten. Zusätzlich wird am Anfang jedes Packetes ein Trigger Tag und Readout Board Status-Wort übermittelt, am Ende befinden sich noch ein Kontroll-Wort sowie eine Prüfsumme. Es werden immer die Daten von 8 Reihen eines Detektorsegmentes in einem Paket übermittelt.

4.2.2.3 Interface zur RICH-IPU

Die Übertragung der Daten vom Readout-Controller an die PRC erfolgt über einen 20 Bit breiten Bus, der mit TTL-Pegeln [4] betrieben wird. Die einzelnen Signale haben dabei folgende Bedeutung:

D(15..0): Auf diesen Leitungen werden Zeilen- und Spaltenaddesse der Pads, welche den programmierten Schwellwert überschreiten, übermittelt.

Write: Das auf D(15..0) anliegende Datenwort wird bei steigender Flanke dieses Signals von der PRC übernommen.

[4] **T**ransistor to **T**ransistor **L**ogic, Low = 0V, High = 5V

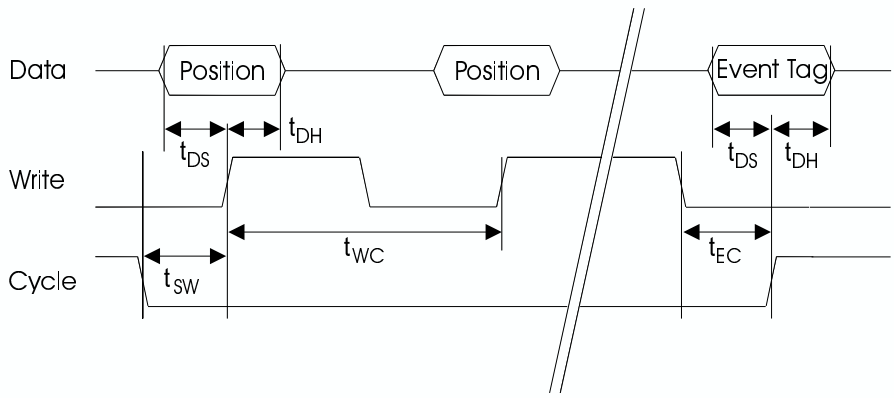


Abbildung 4.10: *Timing Diagramm für den Daten-Transfer Zwischen Readout Controller und PRC. Die wichtigsten Timing-Parameter:* $t_{DS} \geq 15ns$,$t_{DH} \geq 3ns$,$t_{SW} \geq 15ns$, $t_{WC} \geq 32ns$ *und* $t_{EC} \geq 20ns$

Cycle: Mit diesem Signal zeigt der Readout Controller der PRC die Gültigkeit der Daten D(15..0) an. Bei steigender Flanke dieses Signals am Ende einer Datenübermittelung wird außerdem das *Trigger-Tag* übertragen.

Clear: Dieses Signal wird vom Readout-Controller aktiviert um einen Datentransfer vorzeitig zu beenden. Dieses kann geschehen, wenn in einem Event zu viele Koordinaten enthalten sind.

4.2.2.4 Interface zur TOF-IPU

Die TOF-IPU erhält ihre Daten von einem Segment-Controller mittels folgender Leitungen:

D(0..15): Datenleitungen

WCLK: Taktsignal

Data Valid: dieses Signal zeigt an, wann gültige Daten anliegen.

Busy: Dieses Signal ist während der Initialisierungsphase der Segment-Controller aktiv.

Das Protokoll ist weitgehend identisch mit dem der SHOWER-IPU, nur die zeitlichen Parameter sind genau doppelt so groß. Die Datenübertragung erfolgt also mit der halben Frequenz der SHOWER-IPU.

Kapitel 5

Inbetriebnahme des Systems

5.1 Bau des Prototypen

Für die ersten Tests wurde zunächst eine PCI-Basis-Karte und eine Add-On Karte für die SHOWER-IPU, deren Entwicklung am weitesten fortgeschritten ist, gebaut. Die Platinen dafür wurden außer Haus bei der Firma ILFA in Hannover gefertigt, die Bestückung konnte jedoch im institutseigenen SMD-Labor [1] erfolgen: Hier standen ein Dispenser der Firma Martin, ein Fimeplacer von Fritsch, eine Lötstraße der Firma Rehm, eine Reparaturstation der Firma PACE, ein 3D-Mikroskop zur optischen Kontrolle der Lötstellen sowie die alle notwendigen Geräte für elektrische Tests (Oszilloskop, Logik-Analysator,...) zur Verfügung.

Die PCI-Karte wurde als erstes gebaut. Die Platine wurde in 6-Lagen-Technik entworfen: 3 Lagen wurden für die Spannungsversorgung der Komponenten (5V, 3.3V, Masse) benötigt, die restlichen Lagen standen für Signal-Leitungen zur Verfügung. Aufgrund der ungeraden Zahl an Kupferebenen für die Spannungsversorgung konnte die Platine nicht mit der Lötstraße gelötet werden. Da diese in Reflow-Technik[2] arbeitet, hätte die Platine sich dabei verziehen können. Die Bauteile mußten alle manuell gelötet werden, wobei der Fine-Placer mit dem integrierten Mikroskop gute Dienste leistete. Die Lötstellen konnten dann mit Hilfe des 3D-Mikroskops visuell auf kalte Lötstellen und Kurzschlüssen hin untersucht werden. In Abbildung 5.1 ist ein Bild der teilweise bestückten Karte zu sehen.

Die ersten Funktionstests der Basiskarte wurden mit einem Pentium90, der unter dem

[1] SMD: **S**urface **M**ounted **D**evice.

[2] Bei der Reflow-Löttechnik wird die gesamte bestückte Platine stufenweise erhitzt und abgekühlt

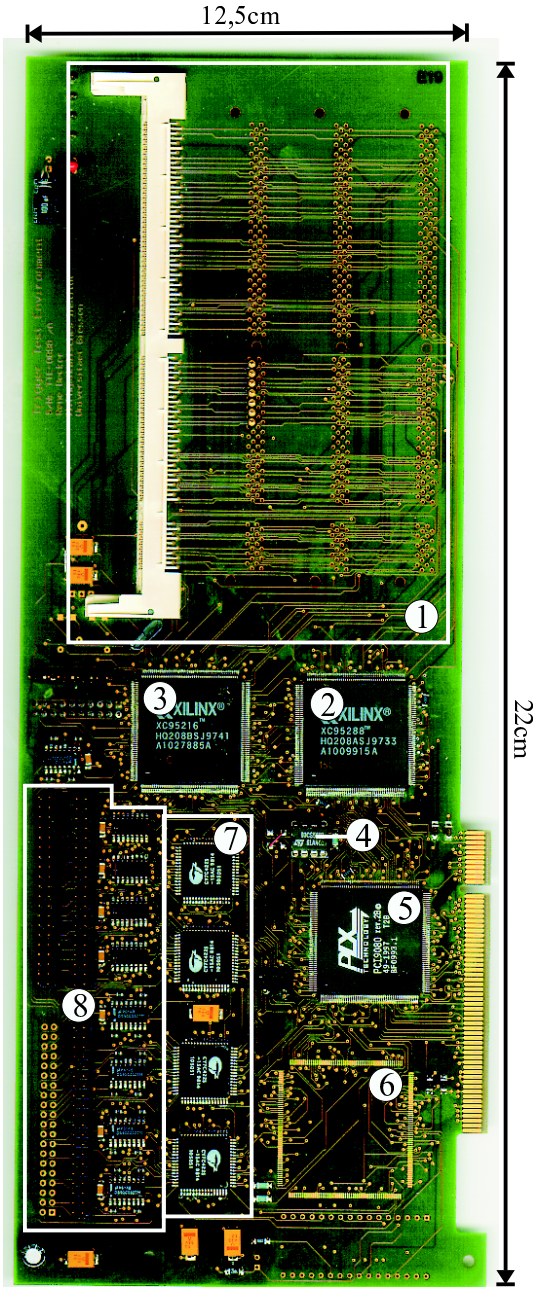


Abbildung 5.1: *Die PCI-Einsteckkarte: 1: DIMM-Sockel für die SDRAM-Module; 2: XC95288 CPLD als Board-Controller (enthält insbesondere die SDRAM Steuerung); 3: XC95216 CPLD zum entkoppeln der TTL- und LVTTL-Komponenten; 4: EEPROM zur Konfiguration des PCI-Chips; 5: PCI-Interface PCI9080 von PLX-Technology; 6: Optionaler DSP (noch nicht eingelötet); 7: FiFos ; 8: LVDS-Treiber und -Empfänger für eine IPU-spezifische Add-On-Karte.*

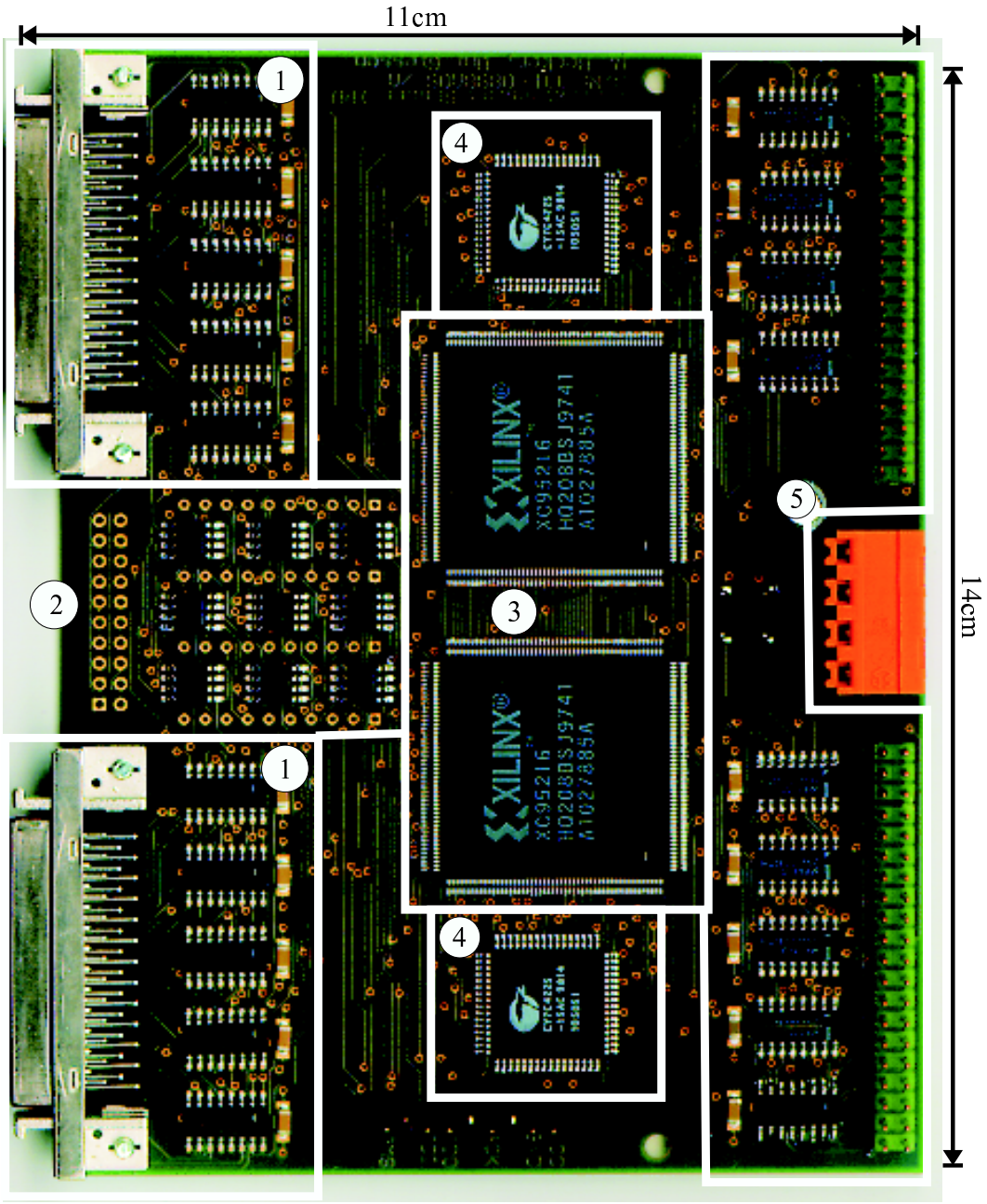


Abbildung 5.2: *Die Add-On Karte für die SHOWER-IPU: 1: LVDS-Treiber und Stecker für die SHOWER-IPU; 2: Differentielle Treiber für den Triggerbus. Der Stecker is auf der Rückseite eingelötet.; 3: Xilinx XC95216 CPLDs zur Kontrolle des Datenflusses; 4: FiFos; 5: LVDS-Receiver und Stecker für den Anschluß an die PCI-Karte*

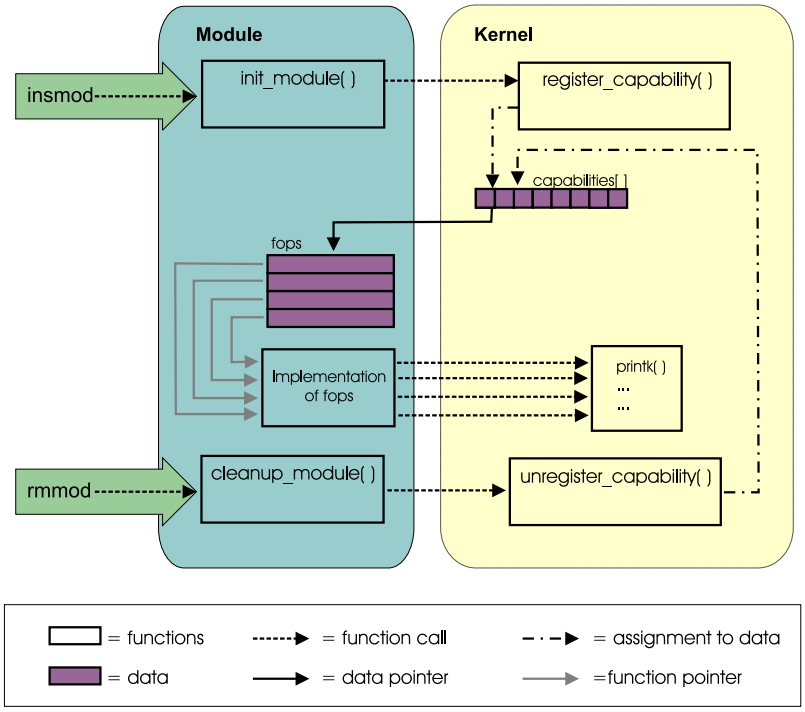


Abbildung 5.3: *Funktionsweise eines Modularen Treibers unter Linux*

Betriebssystem LINUX[3] lief, vorgenommen. Die einzelnen Schritte werden in den folgenden Abschnitten erklärt.

Für die Add-On-Karte wurden lediglich 4 Lagen benötigt: Zwei Lagen für die Spannungsversorgung (5V, Masse) sowie zwei Signallagen. Sie wurde mit Hilfe der Lötstraße angefertigt: Dazu mußte als erstes mit dem Dispenser eine Lötpaste auf die SMD-Pads aufgebracht werden, bevor die Komponenten mit Hilfe des Fine-Placers aufgebracht werden konnten. Die Add-On Karte konnte bislang noch nicht in Betrieb genommen werden. Lediglich die beiden CPLDs konnten durch Programmierung auf ihre Funktionstüchtigkeit überprüft werden. Ein Bild der gelöteten Add-On Karte ist in Abbildung 5.2 zu finden.

5.2 Der PCI-Bus

5.2.1 Software

Das Betriebssystem LINUX erlaubt es einem normalen Anwender nicht, direkt auf die Hardware zuzugreifen. Statt dessen mußte für die Kommunikation mit dem PCI-Board ein *Treiber* entwickelt werden, der eine definierte Schnittstelle zwischen dem PC-Nutzer und der Peripherie darstellt.

Als Vorbild wurde ein Treiber für eine andere PCI-Karte genommen [20], der mit minimalen Veränderungen angepaßt werden konnte. Es handelt sich dabei um einen *modularen* Treiber, der zur Laufzeit des Betriebsystems eingebunden und auch entfernt werden kann.

Die Funktionsweise eines solchen Treibers ist in Abbildung 5.3 dargestellt. Mit dem Befehl `insmod <driver>` wird zunächst die Funktion `init_module()` ausgeführt. Diese hat dafür zu sorgen, daß das betreffende Gerät korrekt initialisiert wird und fordert gleichzeitig den Kernel auf, die Möglichkeiten des Gerätes zu registrieren (`register_capability()`). Welche Fähigkeiten (schreiben,lesen,...) die Hardware besitzt, ist in der fops(file-operations)-Struktur deklariert. Wenn nun ein Anwender auf die Hardware zugreifen will, muß er dieses dem Kernel über einen Systemruf mitteilen, dieser holt sich dann die Daten aus dem User-Raum und überträgt sie gemäß der Anweisungen des Treibers (Implementation of fops) an die Hardware.

Für diese Arbeit wurde im neben den Funktionen `open()` und `close()` , die für jedes Gerät zwingend vorhanden sein müssen, nur die `memmap()`-Funktion implementiert. Diese Funktion blendet den Memory-Addressraum des Gerätes in den User-Addressraum ein, so das Lese- und Schreibzugriffe auf die Karte durch einfache memcopy()-Befehle durchgeführt werden können. Tiefergehende Informationen zum Thema Linux-Treiber finden sich unter anderem in [21, 22]

5.2.2 Hardware

Der Hardware-Teil des PCI-Bus wurde mit Hilfe eines speziellen Diagnosesystems getestet, das aus einer *Extender-Karte*, einem Logik-Analysator und einem Software-Paket bestand. Die Extender-Karte wird zwischen den PCI-Slot und die zu untersuchende Karte

[3]Ein frei verfügbares UNIX-ähnliches Betriebssystem. Insbesondere ist der Source-Code für jederman zugänglich

```
Dec 18 1998 15:17                                                  Page 1

PCI64_ST    -  State Listing

Label    >             PCI 64-Bit Local Bus 2.1                   Time
Base     > Address  Command/Data           Status             Absolute

     -3    000FBC48? - Idle -                                   -96 ns
     -2    00000CFC? - Idle -                                   -64 ns
     -1    00000CFC? - Idle -                                   -32 ns
      0    00000000  Configuration Read                           0  s
                       Type=0 Register=0x00 Function=0
      1    00000000      Initiator Wait                       22.95 us
                         Target Wait
      2    00000000   Data = 0x901010B5 Word #0               23.09 us
      3    00000008  Configuration Read                       43.63 us
                       Type=0 Register=0x02 Function=0
      4    00000008   Data = 0x06800001 Word #0               43.77 us
      5    00000000  Configuration Read                       65.65 us
                       Type=0 Register=0x00 Function=0
      6    00000000   Data = 0x901010B5 Word #0               65.78 us
```

Abbildung 5.4: *Ausgabe des Disassemblers auf dem Logic-Analyzer*

gesteckt, und dient dazu die PCI-Signale für den Logic-Analysator abzugreifen, ohne dabei den Bus im Betrieb zu beeinträchtigen. Auf dem Logik-Analysator können dann die Aktionen auf dem Bus dargestellt werden, wobei die aufzuzeichnenden Daten durch einen sehr flexiblen Trigger selektiert werden können. Sehr hilfreich war auch der mit dem Extender mitgelieferte *Disassembler*, der die digitalen Signale decodiert auf dem Bildschirm wiedergibt. In Abbildung 5.4 ist die Ausgabe des Disassemblers während der Konfigurationsphase des PCI-Bus zu sehen.

5.3 Programmierung der CPLDs

Neben der graphischen Programmierung der Funktionen eines CPLDs oder FPGAs haben sich in neuerer Zeit auch *Hardware Description Languages* etabliert. Der Vorteil bei der Verwendung solcher (abstrakter) Programmiersprachen ist, daß sie zunächst einmal unabhängig von der zugrundeliegenden Hardware sind, d.h. ein existierendes Programm kann prinzipiell auch in verschiedenen Architekturen implementiert werden, wohingegen man sich bei einem *schematic Entry* auf einen Chiptypus festlegt.

Die zur Zeit am weitesten verbreiteten Sprachen sind *Verilog* und *VHDL*. Von den Möglichkeiten her sind sich beide Sprachen sehr ähnlich, wobei Verilog allerdings von der geschichtlichen Entwicklung etwas mehr an der Hardware orientiert ist. Dafür wird in Verilog das Bibliothekskonzept nicht unterstützt, wodurch VHDL bei sehr komplexen

Abbildung 5.5: *Blocksymbol für einen 4-Bit Komparator: Er hat 2 Eingänge* `a` *und* `b` *die miteinander verglichen werden. Wenn die beiden gleich sind wird das Ausgangssignal* `equals` *auf 1 gesetzt, ansonsten auf 0.*

Designs leichter zu handhaben ist.

Die im Institut vorhanden Werkzeuge erlauben es, zwischen beiden Sprachen zu wählen.

5.3.1 VHDL

VHDL (**V**HSIC **H**ardware **D**escription **L**anguage) ist während des VHSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) Programmes, daß Anfang 1980 vom amerikanischen Department of Defense gefördert wurde, kreiert worden. Ziel dieses Programmes war es, die Entwicklung von Integrierten Schaltkreisen voranzutreiben. Die Entwicklung der Sprache VHDL hatte im wesentlichen zwei Gründe:

- Eine Gate-Level Beschreibung war bei den immer komplexer werdenden Schaltungen nicht mehr überschaubar
- Es sollte ein Standart entwickelt werden, die den Informations-Austausch zwischen den Teilnehmern des VHSIC-Programmes erleichtern.

Die Intention der Entwickler war es vornehmlich, einen Standart zur Dokumentation der Schaltungen zu etablieren. Erst später wurden Programme entwickelt, die logische Schaltungen aus VHDL extrahieren. Mit dem Programm-Paket von *SYNOPSYS* stand zur Programmierung der Logik-Bausteine die Software des Marktführers auf diesem Gebiet zur Verfügung.

5.3.2 Design-Entry mit VHDL

Anhand eines kurzen Beispiels soll der Design-Prozeß bei der Verwendung von VHDL illustriert werden. Es soll ein 4Bit-Komparator implementiert werden: dieser soll als

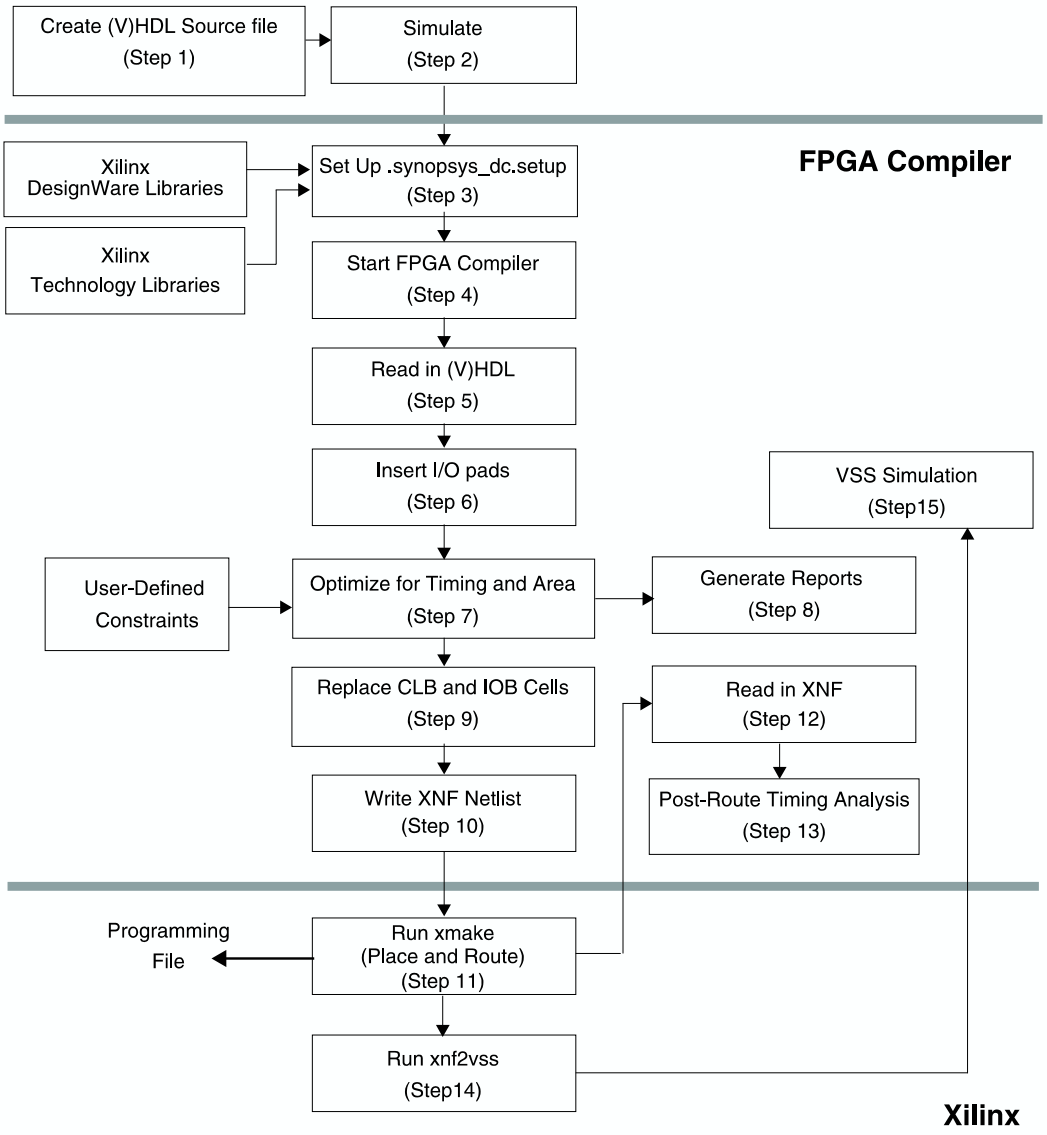


Abbildung 5.6: *Flußdiagramm für die Programmierung eines FPGA's mit VHDL-Entry. Abgesehen von Schritt 9 ist die Programmierung von CPLD's identisch*

Eingangssignale zwei 4-Bit Vektoren `a` und `b` haben die miteinender verglichen werden. Wenn die beiden Vektoren gleich sind, soll das Ausgangssignal `equals` den Wert 1 zurückgegeben, ansonsten den Wert 0. In Abbildung 5.6 ist der Design-Prozeß graphisch dargestellt. Er läßt sich in drei große Abschnitte unterteilen: Zunächst wird die Schaltung entworfen und funktional verifiziert. Als nächstes wird der VHDL-Code compiliert, bevor letztendlich schließlich die Daten zur Programmierung erzeugt werden. Als ein etwas komplexeres Beispiel ist im Anhang A der VHDL-Code für den SDRAM-Controller zu finden.

5.3.2.1 Entwurf und funktionale Verifikation

Der VHDL-Code für den Komparator ist in Abbildung 5.7 aufgelistet. In den Zeilen 1 und 2 wird zunächst angekündigt, auf welche Bibliotheken (bzw. Teile einer Bibliothek) der Designer zurückgreifen möchte. In diesem Fall wird das gesammte Paket `std_logic_1164` aus der Bibliothek `ieee` verwendet, ein Standart-Paket, das in nahezu jedem Design verwendet werden muß (ähnlich wie `stdio.h` in C). In den Zeilen 4 bis 7 erfolgt dann die Deklaration der Komponente als `entity`: Hier wird ihr ein eindeutiger Name (`comp4`) zugewiesen und die Ein- und Ausgangssignale (`ports`) werden definiert. Alle Signale, Variablen und Konstanten müßen einem Datentyp zugewiesen werden. In diesem Fall werden die Ein- und Ausgänge als `std_logic` bzw. als `std_logic_vector` deklariert. Andere mögliche Datentypen wären z.B. Integer, Boolean, Bit(-vector).

Als nächstes kann die Funktion der Komponente beschrieben werden, was in einer `architecture` geschieht. Um eine Schaltung in VHDL zu charakterisieren, stehen dem Programmierer prinzipiell 3 Möglichkeiten zur Verfügung. In Abbildung 5.7 ist für jede dieser Möglichkeiten eine gesonderte architecture vorhanden:

- Mit den Zeilen 9 bis 19 wird die Verhaltensweise des Komparators beschrieben. Dies geschieht in sogenannten Prozeßen, innerhalb derer die Anweisungen sequentiell abgearbeitet werden. Innerhalb von Prozessen können Konstrukte wie *if* und *case* verwendet werden. Eine architecture kann meherere Prozesse enthalten, die dann allerdings parallel ablaufen.

- In der folgenden architecture (Zeilen 21 bis 27) wird der Komparator durch Bool'sche Gleichungen beschrieben.

comp.vhd **1/1**
d:/ Dec 01 1998

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity comp4 is
5      port (a,b : in std_logic_vector(3 downto 0);
6            equals : out std_logic);
7  end comp4;
8
9  architecture behavioral of comp4 is
10 begin  -- behavioral
11     comp : process (a,b)
12     begin  -- process comp
13         if a = b then
14             equals <= '1';
15         else
16             equals <= '0';
17         end if;
18     end process comp;
19 end behavioral;
20
21 architecture bool of comp4 is
22 begin  -- bool
23     equals <=       not(a(0) xor b(0))
24                 and not(a(0) xor b(0))
25                 and not(a(0) xor b(0))
26                 and not(a(0) xor b(0))
27 end bool;
28
29 architecture struct of comp4 is
30     signal x : std_logic_vector(0 to 3);
31 begin  -- struct
32     u0 : xnor2 port map (a(0),b(0),x(0));
33     u1 : xnor2 port map (a(1),b(1),x(1));
34     u2 : xnor2 port map (a(2),b(2),x(2));
35     u3 : xnor2 port map (a(3),b(3),x(3));
36     u4 : and4 port map (x(0),x(1),x(2),x(3),equals);
37 end struct;
38
end
```

Abbildung 5.7: *Der VHDL-Quell-Code für den 4-Bit Komparator. Die Deklaration der Komponente findet in den Zeilen 4 bis 7 statt. In den folgenden Zeilen 9 bis 19 wird der Komparator über sein Verhalten definiert, in den Zeilen hingegen 21 bis 27 durch Bool'sche Gleichungen. Die Zeilen 29 bis 37 zeigen eine VHDL-Netzliste. Alle 3 Beschreibungen liefern das gleiche Ergebnis bei der Synthese*

- Der Komparator kann auch als Netzliste mit einfachen Gattern (`xnor` und `and`) geschrieben werden (Zeilen 31 bis 37). Die einzelnen Gatter werden dann mit den Signalen x(0),x(1),x(2) und x(3) verdrahtet.

Von diesen Möglichkeiten ist insbesondere die erste interessant, da die sequentielle Beschreibung eines Bauteiles in der Regel besser zu verstehen ist. Innerhalb einer architecture ist es dabei aber durchaus möglich, alle 3 Methoden gleichzeitig anzuwenden.

Nachdem der Code erstellt worden ist, kann eine funktionale Simulation erfolgen (Schritt 2). Zu diesem Zweck wurde der SYNOPSYS **V**hdl-**S**ystem-**S**imulator (VSS) benutzt dessen graphische Frond-Ends, ein *Debugger* und ein *Waveform-Viewer* in Abbildung 5.8 zu sehen sind.

5.3.2.2 Synthese und Optimierung

Im nächsten Abschnitt wird der VHDL-Quell-Code mit dem FPGA-Compiler von SYNOPSYS *synthetisiert* und optimiert. Unter *Synthese* versteht man dabei die Transformation einer abstrakten Schaltungsbeschreibung auf eine konkretere Abbildungsebene. Je nach Zielebene unterscheidet man zwischen der Architektursynthese und der Logiksynthese. Die Architektursynthese bezeichnet die Transformation einer Systembeschreibung (VHDL-Code) in eine Darstellung auf Register-Transfer-Ebene (RTL), während man mit Logiksynthese die Abbildung von der RTL-Ebene in eine hardwarenahe Beschreibung auf Gatterebene meint.

Bevor der Compiler gestartet wird, teilt man ihm über eine Set-Up Datei mit, für welchen Baustein man das Design entworfen hat. Der Compiler liest dann die *Technology Library* ein, eine Bibliothek in der alle elemtaren Gatter, die in einem Baustein zur Verfügung stehen, definiert sind (Schritt 3). Nachdem der Compiler aufgerufen wurde, wird der VHDL-Code eingelesen (Schritt 5). Dabei findet die Architektursynthese statt. Die Logiksynthese findet in den darauf folgenden Schritten (6 bis 10) statt. Als erstes werden die IO-Buffer für die Ports eingefügt. Danach wird die Logik anhand vom Designer gemachten Vorgaben (*User-Defined Constraints*) optimiert und auf die in der *Technology Library* enthaltenen Elemente abgebildet. Das Ergebnis (Schritt 10) ist eine reine bausteinspezifische Netzliste (entweder in einem herstellerspezifischen Format wie *.xnf oder als EDIF-Netzliste). In Abbildung 5.9 ist das Ergebnis der Synthese des Beispiel-Designs zu sehen.

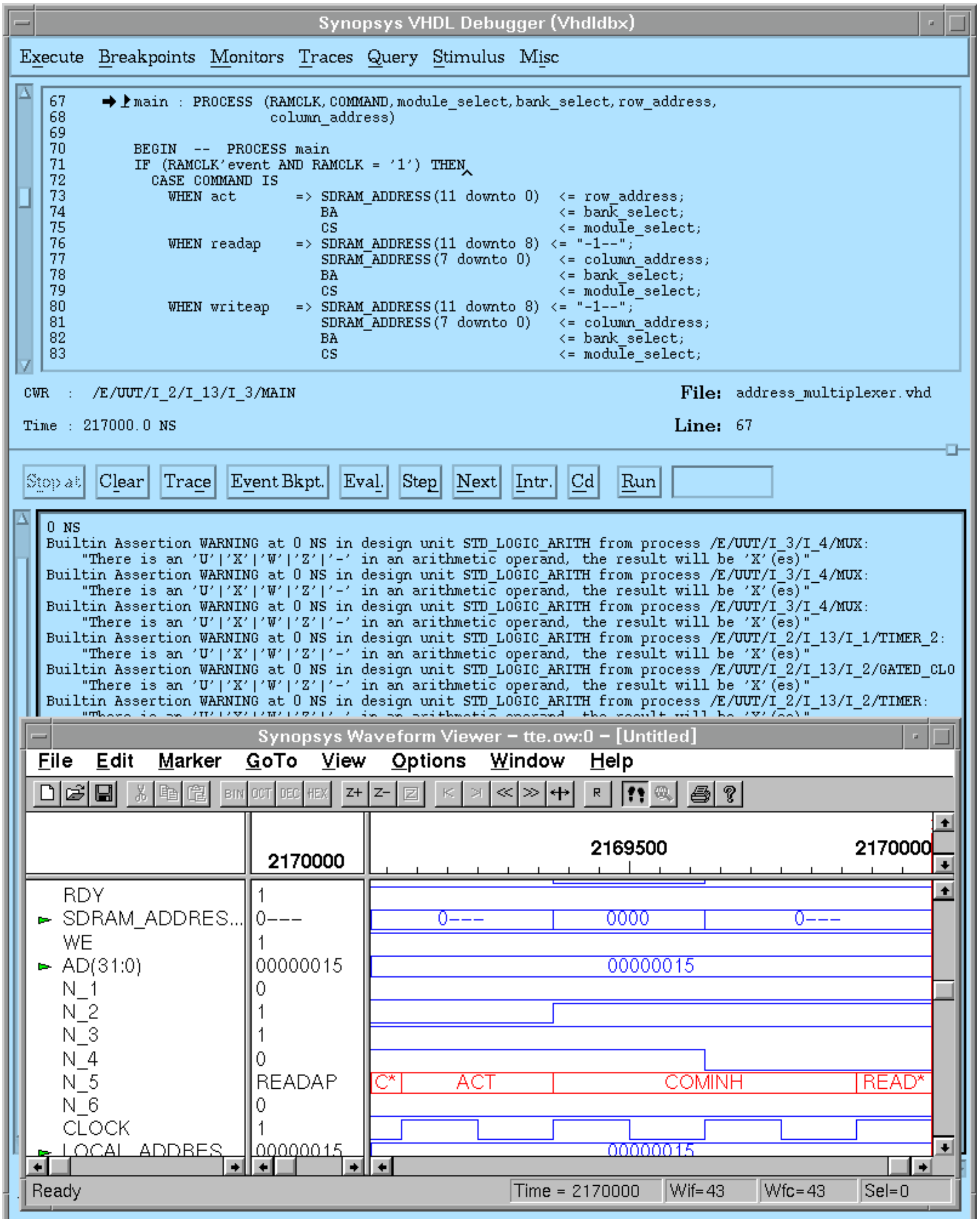


Abbildung 5.8: *Graphische Oberfläche des VHDL Simulators VSS von SYNOPSYS. Im Hintergrund sieht man das Debugger-Fenster, im Vordergrund der Waveform-Viewer, mit dem das zeitliche Verhalten der Signale dargestellt wird.*

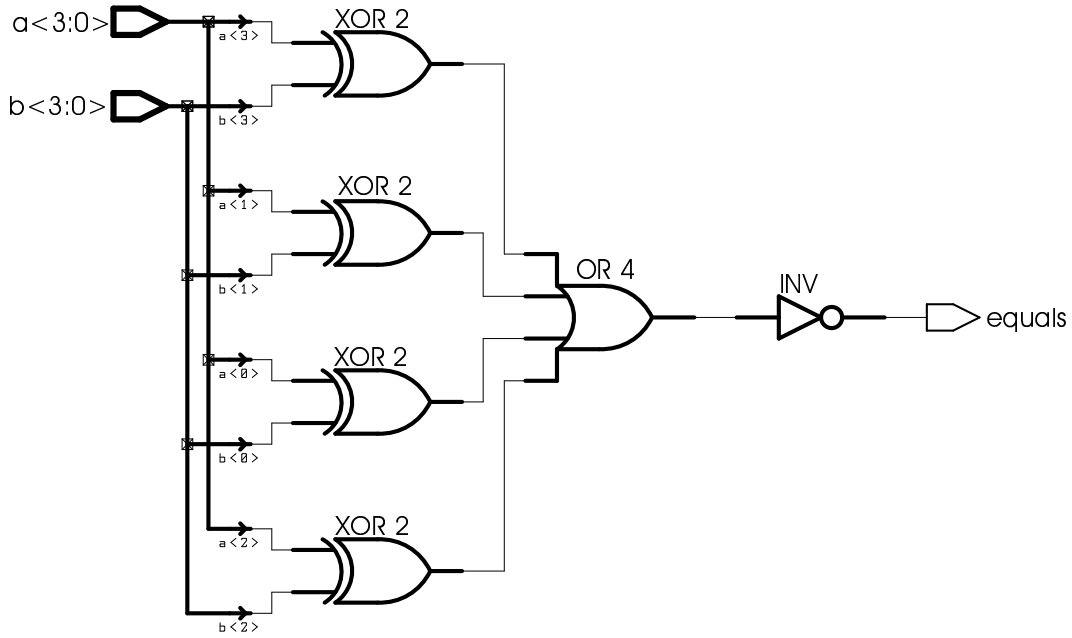


Abbildung 5.9: *Der synthetisierter 4Bit Komparator. Er wurde für die XC9500 CPLDs von Xilinx optimiert.*

5.3.2.3 Mapping und Timing-Simululation

In Schritt 11 wird die optimierte Netzliste mit den Software-Tools des Chip-Herstellers, in diesem Fall Xilinx, *gefittet*, dh. die Logik wird plaziert und geroutet. Das Ergebnis ist eine Datei, die direkt zur Programmierung der Chips benutzt werden kann. Als letztes (Schritt 15) kann das fertige Design einer Timing-Simulation unterzogen werden. Dazu schreibt der Fitter das fertige Design im VHDL-Format (im Netzlisten-Stil) inclusive Timing-Informationen hinaus. Da die so erzeugten VHDL-Dateien sehr groß sind, ist die Timing-Analyse allerdings sehr zeitaufwendig.

Nähere Informationen zum Thema VHDL gibt es in [23, 24, 25]. Als Beispiel für ein komplexeres Programm befindet sich im Anhang A der Quell-Code für den SDRAM-Controller.

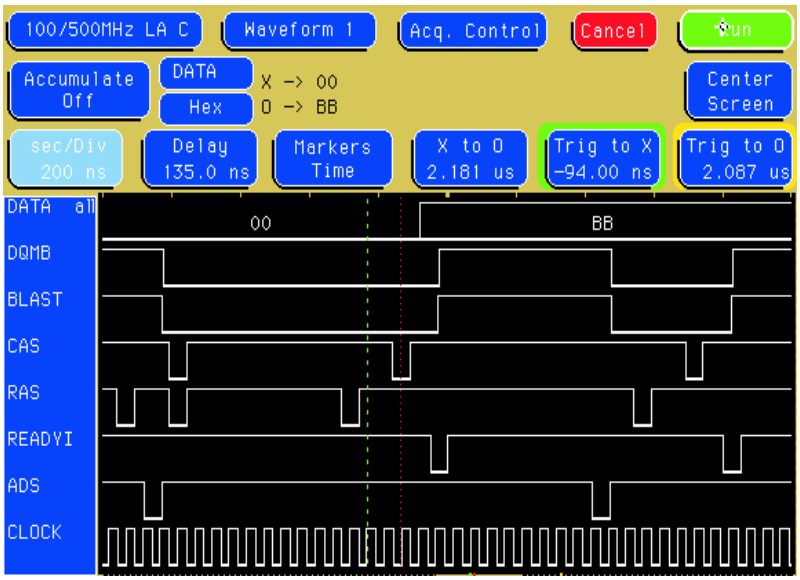


Abbildung 5.10: *Screenshot der Logik-Analysators. Zu sehen ist ein Zugriff auf das SDRAM-Modul*

5.4 Das SDRAM

Für die Inbetriebnahme wurde ein PC100-kompatibles[4] SDRAM-Modul mit einer Speicherkapazität von 64 MByte verwendet. In Abbildung 5.10 ist eine Aufnahme des Logik-Analysators während eines Zugriffs zu sehen. Aufgezeichnet wurden ein *Refresh*-Befehl (RAS und CAS gleichzeitig aktiv, ganz links), sowie 2 Lesezugriffe. Das SDRAM wurde zunächst nur für Einzelzugriffe konfiguriert, da Burst-Zugriffe einen erheblichen Mehraufwand bei der Programmierung des Controllers erfordern.

[4]Diese Speicher können mit einer Frequenz von bis zu 100MHz betrieben werden

Kapitel 6

Ausblick

Im Rahmen dieser Diplomarbeit wurde ein Konzept für ein Trigger-Test System erarbeitet, mit dem es möglich sein sollte, den Second-Level Trigger des HADES-Spektrometers mit realistischen Datenraten zu testen, ohne dabei auf die übrigen Komponenten angewiesen zu sein. In Hardware umgesetzt wurden von diesem Konzept bisher eine PCI-Einsteckkarte sowie eine Add-On Karte für die SHOWER-IPU. Mit den Hilfsmitteln des Elektronik-Labors konnten bereits große Teile dieses Systems in Betrieb genommen werden: die Anbindung über den PCI-Bus an einen PC funktioniert problemlos, es wurde ein Treiber für die Steuerung der Hardware geschrieben, die CPLDs der PCI-Karte wurden programmiert und das SDRAM konnte beschrieben und ausgelesen werden.

Im nächsten Schritt muß die Add-On Karte für die SHOWER-IPU, deren Entwicklung am weitesten fortgeschritten ist, programmiert werden, mit ihr können dann die ersten Funktionstests der IPU durchgeführt werden, allerdings noch nicht mit realistischen Datenraten. Für die Tests mit voller Geschwindigkeit muß die Burst-Fähigkeit der SDRAMs ausgenutzt werden, d.h. der im Anhang aufgelistste Quell-Code muß modifiziert werden.

Darauf folgend müssen dann die Add-On Karten für RICH- und TOF-IPU gebaut und programmiert werden. Mit deren Hilfe können dann auch die übrigen IPUs, sobald sie betriebsbereit sind, zunächst unabhängig voneinander geprüft werden.

Falls die ersten Tests erfolgreich bestanden wurden kann der komplette Second-Level Trigger einem Funktionstest unterworfen werden. Ein solcher Test ist im Frühjahr 1999 geplant.

Anhang A

VHDL-Code

Als Beispiel für ein kompleseres VHDL-Programm ist auf den folgenden Seiten der Quell-Code für den in einem Xilinx XC95288 implementierten SDRAM-Controller zu finden. Der Code ist wie folgt aufgebaut: Die Entity `sdram_controller` stellt die oberste Hierarchieeben dar. Diese Entity ist komplett im *Netzlisten-Stil* geschrieben und enthält die Komponenten `refresh_timer`,`reg_d`,`reg_ss`, `state_machine` und `address_multiplexer`, welche alle durch eine vehaltensmäßige Beschreibung spezifiziert werden. Der Refresh-Timer und der Adress-Multiplexer greifen auf das Package `conversions` zurück, in dem einige Fuktionen zur Typen-Umwandlung definiert sind. Zur Simulation ist eine Test-Bench `E` enthalten. In ihr ist spezifieziert, wie der Controller angesteuert werden muß.

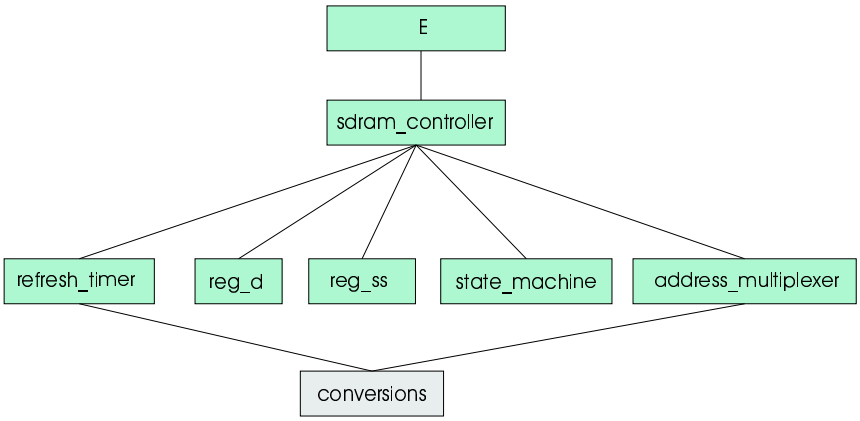


Abbildung A.1: Struktur des VHDL-Code für den SDRAM-Controller

```
------------------------------------------------------------------------------
-- ?? Conversions ??
------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE conversions IS
-- synopsys synthesis_off
    FUNCTION r_log_2 (arg:integer) RETURN integer;  -- returns Xyz
    FUNCTION dmr (a,b,c,d,e : integer) RETURN std_logic_vector;
-- synopsys synthesis_on
END conversions;

PACKAGE BODY conversions IS
    -- purpose: always returns log to determine length of std_logic_vectors
    FUNCTION r_log_2 (arg : integer) RETURN integer IS
        VARIABLE temp : integer;        -- auxiliary variable
    BEGIN  -- r_log_2
        IF (arg < 2) THEN
            RETURN 0;
        ELSE
            temp := 2;
            FOR i IN 1 TO (arg/2) LOOP
                temp := 2 * temp;
                IF (temp = arg) THEN
                    RETURN (i + 1);
                ELSIF (temp > arg) THEN
                    RETURN i;
                END IF;
            END LOOP;  -- i
        END IF;
    END r_log_2;

    -- purpose: generates the value to be written in the SDRAM's mode register
    FUNCTION dmr (a,b,c,d,e : integer) RETURN std_logic_vector IS
        CONSTANT default_val : std_logic_vector(11 DOWNTO 0) := "000000100000";
        VARIABLE temp : std_logic_vector(11 DOWNTO 0) := "000000000000";
        VARIABLE err : boolean := false;
    BEGIN  -- dmr
        CASE a IS
            WHEN 0      => temp(9) := '0';
            WHEN 1      => temp(9) := '1';
            WHEN OTHERS => err := true;
        END CASE;
        CASE b IS
            WHEN 0      => temp(8 DOWNTO 7) := "00";
            WHEN OTHERS => err := true;
        END CASE;
        CASE c IS
            WHEN 1      => temp(6 DOWNTO 4) := "001";
            WHEN 2      => temp(6 DOWNTO 4) := "010";
            WHEN 3      => temp(6 DOWNTO 4) := "011";
            WHEN OTHERS => err := true;
        END CASE;
        CASE d IS
            WHEN 0      => temp(3) := '0';
            WHEN 1      => temp(3) := '1';
            WHEN OTHERS => err := true;
        END CASE;
        CASE e IS
            WHEN 1      => temp(2 DOWNTO 0) := "000";
            WHEN 2      => temp(2 DOWNTO 0) := "001";
            WHEN 4      => temp(2 DOWNTO 0) := "010";
            WHEN 8      => temp(2 DOWNTO 0) := "011";
            WHEN 0      => temp(2 DOWNTO 0) := "111";
            WHEN OTHERS => err := true;
        END CASE;
```



```
        IF (err = true) THEN
            RETURN default_val;
        ELSE
            RETURN temp;
        END IF;
    END dmr;


END conversions;

------------------------------------------------------------------------------
--  Package SDRAM-Modul
------------------------------------------------------------------------------


PACKAGE sdram_module IS
    TYPE sdram_command IS (dsel,            --   Device deselect
                           nop,             --   No operation
                           read,            --   Read
                           readap,          --   Read with Auto Precharge
                           write,           --   Write
                           writeap,         --   Write with Auto Precharge
                           act,             --   Bank Activate
                           pre,             --   Precharge select bank
                           pall,            --   Precharge all banks
                           cbr,             --   Auto refresh
                           slfrsh,          --   Self refresh entry from IDLE
                           slfrshx,         --   Self refresh exit
                           pwrdn,           --   Power down entry from IDLE
                           pwrdnx,          --   Power down exit
                           mrs,             --   Mode register set
                           cominh           --   Command inhibit
                          );
END sdram_module;

------------------------------------------------------------------------------
--  tte-types - nur fr plx-chip?
------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

PACKAGE tte_types IS
  CONSTANT plx_read_high_word_from_ram : std_logic_vector(3 DOWNTO 0):="0001";
  CONSTANT plx_write_high_word_to_ram  : std_logic_vector(3 DOWNTO 0):="0010";
  CONSTANT plx_read_low_word_from_ram  : std_logic_vector(3 DOWNTO 0):="0011";
  CONSTANT plx_write_low_word_to_ram   : std_logic_vector(3 DOWNTO 0):="0100";
  CONSTANT ipu_read_high_word_from_ram : std_logic_vector(3 DOWNTO 0):="0101";
  CONSTANT ipu_write_high_word_to_ram  : std_logic_vector(3 DOWNTO 0):="0110";
  CONSTANT ipu_read_low_word_from_ram  : std_logic_vector(3 DOWNTO 0):="0111";
  CONSTANT ipu_write_low_word_to_ram   : std_logic_vector(3 DOWNTO 0):="1000";
  CONSTANT plx_write_to_ipu            : std_logic_vector(3 DOWNTO 0):="1001";
  CONSTANT plx_read_from_ipu           : std_logic_vector(3 DOWNTO 0):="1010";
  CONSTANT no_command                  : std_logic_vector(3 DOWNTO 0):="0000";
END tte_types;

------------------------------------------------------------------------------
-- D-FlipFlop
------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg_d IS
```

```
    GENERIC (width : integer := 1);

    PORT (clk : IN std_logic;                                   -- Clock input
          d : IN std_logic_vector((width -1) DOWNTO 0) ;        -- D input
          e : IN std_logic;                                     -- Clock enable
          reset : IN std_logic;                                 -- resets
                                                                -- the register
                                                                -- (low active)
          q : OUT std_logic_vector((width -1) DOWNTO 0));       -- result

END reg_d;

-- purpose: behavioral description of a d-type register with clock enable.
--          synthesizable architecture.
ARCHITECTURE behavioral OF reg_d IS

BEGIN  -- behavioral

    -- purpose : a simple register implementation
    -- type    : sequential
    -- inputs  : clk, reset, d,e
    -- outputs : q
    main : PROCESS (clk, reset)

    BEGIN  -- PROCESS main
        -- activities triggered by asynchronous reset (active low)
        IF reset = '0' THEN
            q <= (OTHERS => '0');
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            IF (e = '0') THEN
                q <= d;
            END IF;
        END IF;
    END PROCESS main;

END behavioral;

CONFIGURATION cfg_reg_d_behavioral OF reg_d IS
    FOR behavioral

    END FOR;
END cfg_reg_d_behavioral;

-------------------------------------------------------------------------------
--   SS Register
-------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg_ss IS

    GENERIC (width : integer := 1);                         -- register width

    PORT (clk : IN std_logic;                               -- clock input
          reset : IN std_logic;                     -- async. reset
                                                            -- (low-active)
          a : IN std_logic;                                 -- syncronous set 1
          b : IN std_logic;                                 -- syncronous set 0
          q : OUT std_logic_vector((width -1) DOWNTO 0)); -- result

END reg_ss;

-- purpose: behavioral description
```



```
ARCHITECTURE behavioral OF reg_ss IS

BEGIN  -- behavioral

    -- purpose : x
    -- type    : sequential
    -- inputs  : clk, reset, a,b
    -- outputs : q
    main : PROCESS (clk, reset)

    BEGIN  -- PROCESS main
        -- activities triggered by asynchronous reset (active low)
        IF reset = '0' THEN
            q <= (OTHERS => '0');
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            IF (a = '0') THEN
                q <= (OTHERS => '1');
            ELSIF (b = '1') THEN
                q <= (OTHERS => '0');
            END IF;
        END IF;
    END PROCESS main;

END behavioral;

CONFIGURATION cfg_reg_ss_behavioral OF reg_ss IS
    FOR behavioral

    END FOR;
END cfg_reg_ss_behavioral;

-------------------------------------------------------------------------------
-- Refresh Timer
-------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
LIBRARY work;
USE work.conversions.ALL;

ENTITY refresh_timer IS

    GENERIC (refcycle : integer := 300); -- clock cycles between two
                                         -- refresh signals

    PORT (clk : IN std_logic;            -- clock input
          reset : IN std_logic;          -- asyncronous reset
          rfc : IN boolean;              -- refresh complete
          ref : OUT boolean);            -- refresh signal

END refresh_timer;

-- purpose: syntheszable behavioral description of the refresh counter
ARCHITECTURE behavioral OF refresh_timer IS
    SIGNAL count : std_logic_vector(r_log_2(refcycle) DOWNTO 0); -- clock cycle
                                                                 -- counter

BEGIN  -- behavioral

    -- purpose : counts the clock cycles after the last
    --           refresh complete(rfc) pulse
    -- type    : sequential
    -- inputs  : clk, reset, rfc
    -- outputs : ref
    count_clock : PROCESS (clk, reset,rfc)
```

```
    BEGIN  -- PROCESS count_clock
        -- activities triggered by asynchronous reset (active low)
        IF reset = '0' THEN
            count <= (OTHERS => '0');
        -- the refresh complete (rfc)-signal behaves like an asyncronous
        -- reset too:
        ELSIF (rfc = true) THEN
            count <= (OTHERS => '0');
        -- if refcycle is reached, the counter should lock itself:
        ELSIF (count = refcycle) THEN
            count <= count;
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            count <= count + 1;
        END IF;
    END PROCESS count_clock;

    -- purpose : the refresh signal (ref) is generated here
    -- type    : combinational
    -- inputs  : count
    -- outputs : ref
    gen_ref : PROCESS (count)

    BEGIN  -- PROCESS gen_ref
        IF (count = refcycle) THEN
            ref <= true;
        ELSE
            ref <= false;
        END IF;
    END PROCESS gen_ref;

END behavioral;

CONFIGURATION cfg_refresh_timer_behavioral OF refresh_timer IS
    FOR behavioral

    END FOR;
END cfg_refresh_timer_behavioral;

--------------------------------------------------------------------------------
-- State machine
--------------------------------------------------------------------------------


LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
LIBRARY work;
USE work.sdram_module.ALL;


ENTITY state_machine IS

    GENERIC (cas_latency : integer := 2); -- cas latency of SDRAM

    PORT (clk : IN std_logic;           -- clock input
          reset : IN std_logic;         -- asyncronous reset
          ale : IN std_logic;           -- address latch enable
          r_w : IN std_logic;           -- read/write signal
          ref : IN boolean;             -- refresh signal
          rfc : OUT boolean;            -- refresh complete
          ack : OUT std_logic;          -- access acknoledge
          rdy : OUT std_logic;          -- indicates valid data
          com : OUT sdram_command;      -- commands for address_multiplexer
          cas : OUT std_logic;          -- column address strobe for SDRAM
          ras : OUT std_logic;          -- row address_starobe for SDRAM
```



```
          we : OUT std_logic);          -- write enable for SDRAM

END state_machine;

-- purpose: synthesizable behavioral description of state machine
ARCHITECTURE behavioral OF state_machine IS
    TYPE statetype IS (rst,init,idle,refresh,read,write); -- possible states
    SIGNAL present_state,next_state : statetype;          -- self explaining
    SIGNAL t : std_logic_vector(12 DOWNTO 0);             -- timer
    TYPE access_type IS (read_access,write_access);       -- access types
    SIGNAL rw_flag : access_type;                         -- read/write flag
    SIGNAL tmp : std_logic_vector(2 DOWNTO 0);            -- register
                                                          -- cas,ras,we
    SIGNAL rdy_t : std_logic_vector(3 DOWNTO 0);          -- timer for rdy
    SIGNAL reset_rdy_t : std_logic;                       -- resets rdy_t
    SIGNAL command : sdram_command;                       -- sdram command
BEGIN  -- behavioral

    -- purpose : describes state transitions
    -- type    : combinational
    -- inputs  : present_state,ref,ale,r_w,t
    -- outputs : next_state,com,tmp,reset_rdy_t,rfc,rw_flag
    state_tr : PROCESS (present_state,ref,ale,r_w,t)

    BEGIN  -- PROCESS state_tr
        CASE present_state IS
            WHEN rst  => command     <= cominh;
                         next_state  <= init;
                         reset_rdy_t <= '1';
                         rfc         <= true;
                         rw_flag     <= read_access;
            WHEN init => rw_flag     <= read_access;
                         reset_rdy_t <= '0';
                         rfc         <= true;
                         CASE t IS
                             WHEN "0000000000001" => command    <= nop;
                                                     next_state <= init;
                             WHEN "1111101000000" => command    <= pall;
                                                     next_state <= init;
                             WHEN "1111101000011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101001011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101010011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101011011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101100011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101101011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101110011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111101111011" => command    <= cbr;
                                                     next_state <= init;
                             WHEN "1111110000011" => command    <= mrs;
                                                     next_state <= init;
                             WHEN "1111110000110" => command    <= cominh;
                                                     next_state <= idle;
                             WHEN OTHERS          => command    <= cominh;
                                                     next_state <= init;
                         END CASE;
            WHEN idle => command     <= cominh;
                         reset_rdy_t <= '0';
                         rfc         <= false;
                         rw_flag     <= read_access;
                         IF (ref = true) THEN
```

```
                    next_state <= refresh;
                ELSIF (ale = '0') THEN
                    next_state <= idle;
                ELSIF (r_w = '0') THEN
                    next_state <= read;
                ELSE
                    next_state <= write;
                END IF;
        WHEN refresh => reset_rdy_t <= '0';
                    rw_flag     <= read_access;
                    CASE t(4 DOWNTO 0) IS
                        WHEN "01011" => rfc        <= false;
                                        command    <= pall;
                                        next_state <= refresh;
                        WHEN "01110" => rfc        <= false;
                                        command    <= cbr;
                                        next_state <= refresh;
                        WHEN "10101" => rfc        <= true;
                                        command    <= pall;
                                        next_state <= refresh;
                        WHEN "10110" => rfc        <= false;
                                        command    <= cominh;
                                        next_state <= idle;
                        WHEN OTHERS  => rfc        <= false;
                                        command    <= cominh;
                                        next_state <= refresh;
                    END CASE;
        WHEN read => rw_flag <= read_access;
                    rfc     <= false;
                    CASE t(2 DOWNTO 0) IS
                        WHEN "000"  => next_state  <= read;
                                       command     <= act;
                                       reset_rdy_t <= '1';
                        WHEN "011"  => next_state  <= read;
                                       command     <= readap;
                                       reset_rdy_t <= '0';
                        WHEN "110"  => next_state  <= idle;
                                       command     <= cominh;
                                       reset_rdy_t <= '0';
                        WHEN OTHERS => next_state  <= read;
                                       command     <= cominh;
                                       reset_rdy_t <= '0';
                    END CASE;
        WHEN write => rw_flag <= write_access;
                     rfc     <= false;
                     CASE t(2 DOWNTO 0) IS
                        WHEN "000"  => next_state  <= write;
                                       command     <= act;
                                       reset_rdy_t <= '1';
                        WHEN "011"  => next_state  <= write;
                                       command     <= writeap;
                                       reset_rdy_t <= '0';
                        WHEN "110"  => next_state  <= idle;
                                       command     <= cominh;
                                       reset_rdy_t <= '0';
                        WHEN OTHERS => next_state  <= write;
                                       command     <= cominh;
                                       reset_rdy_t <= '0';
                    END CASE;
    END CASE;
END PROCESS state_tr;

com <= command;

-- purpose : timer
-- type    : sequential
-- inputs  : clk, reset, t
```



```
-- outputs : t
timer_1 : PROCESS (clk, reset,present_state)

BEGIN  -- PROCESS timer_1
    -- activities triggered by asynchronous reset (active low)
    IF reset = '0' THEN
        t <= (OTHERS => '0');
    -- present_state = idle also resets t
    ELSIF (present_state = idle) THEN
        t <= (OTHERS => '0');
    -- activities triggered by rising edge of clock
    ELSIF clk'event AND clk = '1' THEN
        t <= t + 1;
    END IF;
END PROCESS timer_1;

-- purpose : state transitions
-- type    : sequential
-- inputs  : clk, reset, next_state
-- outputs : present_state
clocked_state : PROCESS (clk, reset)

BEGIN  -- PROCESS clocked_state
    -- activities triggered by asynchronous reset (active low)
    IF reset = '0' THEN
        present_state <= rst;
    -- activities triggered by rising edge of clock
    ELSIF clk'event AND clk = '1' THEN
        present_state <= next_state;
    END IF;
END PROCESS clocked_state;

-- purpose : register Outputs ras,cas,we
-- type    : sequential
-- inputs  : clk, reset, command
-- outputs : ras,cas,we
com_reg : PROCESS (clk, reset)

BEGIN  -- PROCESS com_reg
    -- activities triggered by asynchronous reset (active low)
    IF reset = '0' THEN
        (ras,cas,we) <= std_logic_vector'("000");
    -- activities triggered by rising edge of clock
    ELSIF clk'event AND clk = '1' THEN
        CASE command IS
            WHEN nop     => (ras,cas,we) <= std_logic_vector'("111");
            WHEN pall    => (ras,cas,we) <= std_logic_vector'("010");
            WHEN cbr     => (ras,cas,we) <= std_logic_vector'("001");
            WHEN mrs     => (ras,cas,we) <= std_logic_vector'("000");
            WHEN act     => (ras,cas,we) <= std_logic_vector'("011");
            WHEN readap  => (ras,cas,we) <= std_logic_vector'("101");
            WHEN writeap => (ras,cas,we) <= std_logic_vector'("100");
            WHEN OTHERS  => (ras,cas,we) <= std_logic_vector'("111");
        END CASE;
    END IF;
END PROCESS com_reg;

-- purpose : timer
-- type    : sequential
-- inputs  : clk, reset_rdy_t, rdy_t
-- outputs : rdy_t
timer_2 : PROCESS (clk, reset_rdy_t)

BEGIN  -- PROCESS timer_2
    -- activities triggered by asynchronous reset (active low)
    IF reset = '0' OR reset_rdy_t = '1' THEN
        rdy_t <= (OTHERS => '0');
```

```
        ELSIF (rdy_t = 12) THEN
            rdy_t <= rdy_t;
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            rdy_t <= rdy_t + 1;
        END IF;
    END PROCESS timer_2;

    -- purpose : generate rdy signal
    -- type    : sequential
    -- inputs  : clk, reset, rw_flag,rdy_t
    -- outputs : rdy
    gen_rdy : PROCESS (clk, reset)

    BEGIN  -- PROCESS gen_rdy
        -- activities triggered by asynchronous reset (active low)
        IF reset = '0' THEN
            rdy <= '1';
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            IF (rw_flag = read_access) THEN
                IF (rdy_t = (2 + cas_latency)) THEN
                    rdy <= '0';
                ELSE
                    rdy <= '1';
                END IF;
            ELSIF (rw_flag = write_access) THEN
                IF (rdy_t = 2) THEN
                    rdy <= '0';
                ELSE
                    rdy <= '1';
                END IF;
            END IF;
        END IF;
    END PROCESS gen_rdy;

    -- purpose : generate ack signal
    -- type    : sequential
    -- inputs  : clk, reset, present_state
    -- outputs : ack
    gen_ack : PROCESS (clk, reset)

    BEGIN  -- PROCESS gen_ack
        -- activities triggered by asynchronous reset (active low)
        IF reset = '0' THEN
            ack <= '0';
        -- activities triggered by rising edge of clock
        ELSIF clk'event AND clk = '1' THEN
            IF (present_state = read OR present_state = write) THEN
                ack <= '1';
            ELSE
                ack <= '0';
            END IF;
        END IF;
    END PROCESS gen_ack;


END behavioral;

CONFIGURATION cfg_state_machine_behavioral OF state_machine IS
    FOR behavioral

    END FOR;
END cfg_state_machine_behavioral;


--------------------------------------------------------------------------------
```



```
-- Adress Multiplexer
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
LIBRARY work;
USE work.sdram_module.ALL;
USE work.conversions.ALL;


ENTITY address_multiplexer IS

    GENERIC (write_burst_mode : integer := 0;
             op_mode : integer := 0;
             cas_latency : integer := 2;
             burst_type : integer := 0;
             burst_length : integer := 4);

    PORT (clk : IN std_logic;                          -- clock input
          reset : IN std_logic;                        -- asyncronous reset
          blast : IN std_logic;                        -- burst's last word
          be : IN std_logic_vector(3 DOWNTO 0);        -- byte enables
          com : IN sdram_command;                      -- current command
          addr : IN std_logic_vector(31 DOWNTO 0);     -- local address
          ba : OUT std_logic_vector(1 DOWNTO 0);       -- SDRAM bank address
          cs : OUT std_logic_vector(3 DOWNTO 0);       -- SDRAM Module select
          dqmb : OUT std_logic_vector(7 DOWNTO 0);     -- SDRAM Data Mask
          raddr : OUT std_logic_vector(13 DOWNTO 0)); -- SDRAM Address

END address_multiplexer;

-- purpose: synthesizable behavioral description of address multiplexer
ARCHITECTURE behavioral OF address_multiplexer IS

    CONSTANT mode_reg : std_logic_vector := dmr(write_burst_mode,
                                                op_mode,cas_latency,
                                                burst_type,burst_length
                                                );  -- mode register
    CONSTANT all_mods : std_logic_vector(3 DOWNTO 0) := "0000"; -- select all
                                                                -- modules
    CONSTANT no_mod : std_logic_vector(3 DOWNTO 0) := "1111";  -- select no
                                                               -- module
    SIGNAL mod_sel : std_logic_vector(3 DOWNTO 0);  -- selects SDRAM module
    SIGNAL b_sel : std_logic_vector(1 DOWNTO 0);  -- selects bank on module
    SIGNAL row_addr : std_logic_vector(11 DOWNTO  0);  -- row address
    SIGNAL column_addr : std_logic_vector(7 DOWNTO 0);  -- column address

BEGIN  -- behavioral
 -- purpose : generate commands for SDRAM Module
 -- type    : sequential
 -- inputs  : clk, reset, com
 -- outputs : raddr,cs,ba
 gen_com : PROCESS (clk, reset)

 BEGIN  -- PROCESS gen_com
     -- activities triggered by asynchronous reset (active low)
     IF reset = '0' THEN
         raddr <= (OTHERS => '0');
         ba    <= (OTHERS => '0');
         cs    <= (OTHERS => '1');
     -- activities triggered by rising edge of clock
     ELSIF clk'event AND clk = '1' THEN
         CASE com IS
             WHEN act     => raddr(11 DOWNTO 0) <= row_addr;
                             ba                 <= b_sel;
                             cs                 <= mod_sel;
```

```
            WHEN readap  => raddr(11 DOWNTO 8) <= "-1--";
                            raddr(7 DOWNTO 0)  <= column_addr;
                            ba                 <= b_sel;
                            cs                 <= mod_sel;
            WHEN writeap => raddr(11 DOWNTO 8) <= "-1--";
                            raddr(7 DOWNTO 0)  <= column_addr;
                            ba                 <= b_sel;
                            cs                 <= mod_sel;
            WHEN pall    => raddr(11 DOWNTO 0) <= "-1----------";
                            ba                 <= b_sel;
                            cs                 <= mod_sel;
            WHEN mrs     => raddr(11 DOWNTO 0) <= mode_reg;
                            ba                 <= "00";
                            cs                 <= all_mods;
            WHEN cbr     => raddr(11 DOWNTO 0) <= "------------";
                            ba                 <= "00";
                            cs                 <= all_mods;
            WHEN nop     => raddr(11 DOWNTO 0) <= "------------";
                            ba                 <= "00";
                            cs                 <= all_mods;
            WHEN cominh  => raddr(11 DOWNTO 0) <= "------------";
                            ba                 <= "00";
                            cs                 <= no_mod;
            WHEN OTHERS  => raddr(11 DOWNTO 0) <= "------------";
                            ba                 <= "00";
                            cs                 <= no_mod;
        END CASE;
    END IF;
END PROCESS gen_com;

raddr(13 DOWNTO 12) <= "00";

-- purpose : local address to sdram address mapping
-- type    : combinational
-- inputs  : reset,addr,be,com,blast
-- outputs : row_addr,column_addr,mod_sel,b_sel,dqmb
addr_dec : PROCESS (com,reset,addr,be,blast,com)

BEGIN  -- PROCESS addr_dec
    row_addr                <= addr(21 DOWNTO 10);
    column_addr(1 DOWNTO 0) <= addr(1 DOWNTO 0);
    column_addr(7 DOWNTO 2) <= addr(9 DOWNTO 4);
    b_sel                   <= addr(3 DOWNTO 2);

    CASE addr(24 DOWNTO 23) IS
        WHEN "00"   => mod_sel <= "1110";
        WHEN "01"   => mod_sel <= "1101";
        WHEN "10"   => mod_sel <= "1011";
        WHEN "11"   => mod_sel <= "0111";
        WHEN OTHERS => mod_sel <= "1111";
    END CASE;

    IF (reset = '0') THEN
        dqmb <= "11111111";
    ELSIF (com = writeap OR com  = readap) THEN
        IF (addr(22) = '1') THEN
            dqmb(7 DOWNTO 4) <= be;
            dqmb(3 DOWNTO 0) <= "1111";
        ELSE
            dqmb(7 DOWNTO 4) <= "1111";
            dqmb(3 DOWNTO 0) <= be;
        END IF;
    ELSIF (blast = '1') THEN
        dqmb <= "11111111";
    END IF;

END PROCESS addr_dec;
```



```
END behavioral;

CONFIGURATION cfg_address_multiplexer_behavioral OF address_multiplexer IS
    FOR behavioral

    END FOR;
END cfg_address_multiplexer_behavioral;


-------------------------------------------------------------------------------
-- Schematic top level
-------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
LIBRARY work;
USE work.sdram_module.ALL;


ENTITY sdram_controller IS

    GENERIC (write_burst_mode : integer := 0; -- write burst mode:
                                              -- 0 = programmed burst length
                                              -- 1 = single location access
             op_mode : integer := 0;          -- Operation mode:
                                              -- 0 = standart
                                              -- all others = undefined
             cas_latency : integer := 2;      -- CAS latency:
                                              -- Number of clock cycles after
                                              -- CAS when read data is valid.
                                              -- possible values: 1,2,3
             burst_type : integer := 0;       -- Burst type:
                                              -- 0 = sequential
                                              -- 1 = interleaved
             burst_length : integer := 1;     -- bust_length. Possible values:
                                              -- 1,2,4,8,0(=full page, only
                                              -- defined for burst type = 0).
             refcycle : integer := 300);      -- Number of clock cycles between
                                              -- two refresh commands.

    PORT (clk : IN std_logic;                          -- clock input
          reset : IN std_logic;                        -- asyncronous reset
          ale : IN std_logic;                          -- addess latch enable
          rdy : OUT std_logic;                  -- data valid signal
          addr : IN std_logic_vector(31 DOWNTO 0);     -- address
          rw : IN std_logic;                           -- read/write signal
          be : IN std_logic_vector(3 DOWNTO 0);        -- byte enables
          blast : IN std_logic;                        -- burst's last word
          ba : OUT std_logic_vector(1 DOWNTO 0);       -- SDRAM bank address
          ras : OUT std_logic;                         -- SDRAM row address
                                                       -- strobe
          cas : OUT std_logic;                         -- SDRAM column address
                                                       -- strobe
          cs : OUT std_logic_vector(3 DOWNTO 0);       -- SDRAM Module select
          dqmb : OUT std_logic_vector(7 DOWNTO 0);     -- SDRAM data mask
          raddr : OUT std_logic_vector(13 DOWNTO 0);   -- SDRAM address
          we : OUT std_logic);                         -- SDRAM write enable

END sdram_controller;

-- purpose: xxx
ARCHITECTURE schematic OF sdram_controller IS

    COMPONENT reg_d
        GENERIC (width : integer);
        PORT (clk : IN std_logic;
```

```
        d : IN std_logic_vector((width -1) DOWNTO 0);
        e : IN std_logic;
        reset : IN std_logic;
        q : OUT std_logic_vector((width -1) DOWNTO 0));
END COMPONENT;

COMPONENT reg_ss
    GENERIC (width : integer);
    PORT (clk : IN std_logic;
          reset : IN std_logic;
          a : IN std_logic;
          b : IN std_logic;
          q : OUT std_logic_vector((width - 1) DOWNTO 0));
END COMPONENT;

COMPONENT refresh_timer
    GENERIC (refcycle : integer);
    PORT (clk : IN std_logic;
          reset : IN std_logic;
          rfc : IN boolean;
          ref : OUT boolean);
END COMPONENT;

COMPONENT state_machine
    GENERIC (cas_latency : integer);
    PORT (clk : IN std_logic;
          reset : IN std_logic;
          ale : IN std_logic;
          r_w : IN std_logic;
          ref : IN boolean;
          rfc : OUT boolean;
          ack : OUT std_logic;
          rdy : OUT std_logic;
          com : OUT sdram_command;
          cas : OUT std_logic;
          ras : OUT std_logic;
          we : OUT std_logic);
END COMPONENT;

COMPONENT address_multiplexer
    GENERIC (write_burst_mode : integer;
             op_mode : integer;
             cas_latency : integer;
             burst_type : integer;
             burst_length : integer);
    PORT (clk : IN std_logic;
          reset : IN std_logic;
          blast : IN std_logic;
          be : IN std_logic_vector(3 DOWNTO 0);
          com : IN sdram_command;
          addr : IN std_logic_vector(31 DOWNTO 0);
          ba : OUT std_logic_vector(1 DOWNTO 0);
          cs : OUT std_logic_vector(3 DOWNTO 0);
          dqmb : OUT std_logic_vector(7 DOWNTO 0);
          raddr : OUT std_logic_vector(13 DOWNTO 0));
END COMPONENT;

SIGNAL n_1 : boolean;
SIGNAL n_2 : std_logic;
SIGNAL n_3 : boolean;
SIGNAL n_4 : std_logic_vector(0 DOWNTO 0); -- must be vector due to
                                           -- error code HDL-206
SIGNAL n_5 : sdram_command;
SIGNAL n_6 : std_logic_vector(0 DOWNTO 0);
SIGNAL n_7 : std_logic_vector(0 DOWNTO 0);
SIGNAL ad : std_logic_vector(31 DOWNTO 0);
```



```
BEGIN  -- schematic

    n_7(0) <= rw;   -- converts std_logic signal rw to std_logic_vector n_7

    I_1 : reg_ss
        GENERIC MAP (width => 1)
        PORT MAP (a => ale,
                  b => n_2,
                  clk => clk,
                  reset => reset,
                  q => n_4);
    I_2 : reg_d
        GENERIC MAP (width => 1)
        PORT MAP (clk => clk,
                  d => n_7,
                  e =>ale,
                  reset => reset,
                  q => n_6);
    I_3 : reg_d
        GENERIC MAP (width => 32)
        PORT MAP (clk => clk,
                  d => addr,
                  e => ale,
                  reset => reset,
                  q => ad);
    I_4 : state_machine
        GENERIC MAP (cas_latency => cas_latency)
        PORT MAP (clk => clk,
                  reset => reset,
                  ale => n_4(0),
                  r_w => n_6(0),
                  ref => n_1,
                  rfc => n_3,
                  ack => n_2,
                  rdy => rdy,
                  com => n_5,
                  cas => cas,
                  ras => ras,
                  we => we);
    I_5 : address_multiplexer
        GENERIC MAP (write_burst_mode => write_burst_mode,
                     op_mode => op_mode,
                     cas_latency => cas_latency,
                     burst_type => burst_type,
                     burst_length => burst_length)
        PORT MAP (clk => clk,
                  reset => reset,
                  blast => blast,
                  be => be,
                  com => n_5,
                  addr => ad,
                  ba => ba,
                  cs => cs,
                  dqmb => dqmb,
                  raddr => raddr);
    I_6 : refresh_timer
        GENERIC MAP (refcycle => refcycle)
        PORT MAP (clk => clk,
                  reset => reset,
                  rfc => n_3,
                  ref => n_1);

END schematic;

CONFIGURATION cfg_sdram_controller_schematic OF sdram_controller IS
    FOR schematic
       FOR I_1 : reg_ss
```

```
        USE ENTITY work.reg_ss(behavioral);
      END FOR;
      FOR I_2 : reg_d
        USE ENTITY work.reg_d(behavioral);
      END FOR;
      FOR I_3 : reg_d
        USE ENTITY work.reg_d(behavioral);
      END FOR;
      FOR I_4 : state_machine
        USE ENTITY work.state_machine(behavioral);
      END FOR;
      FOR I_5 : address_multiplexer
        USE ENTITY work.address_multiplexer(behavioral);
      END FOR;
      FOR I_6 : refresh_timer
        USE ENTITY work.refresh_timer(behavioral);
      END FOR;
    END FOR;
END cfg_sdram_controller_schematic;


-- pragma translate_off

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_unsigned.all;

entity E is
end E;

Architecture A of E is

   signal  addr : std_logic_vector (31 downto 0);
   signal  ale  : std_logic;
   signal  be   : std_logic_vector (3 downto 0);
   signal  blast: std_logic;
   signal  clk  : std_logic;
   signal  reset: std_logic;
   signal    rw : std_logic;
   signal   ba  : std_logic_vector (1 downto 0);
   signal  cas  : std_logic;
   signal    cs : std_logic_vector (3 downto 0);
   signal  dqmb : std_logic_vector (7 downto 0);
   signal   ras : std_logic;
   signal   rdy : std_logic;
   signal raddr : std_logic_vector (13 downto 0);
   signal    we : std_logic;

   component sdram_controller
      Port (clk : IN std_logic;                          -- clock input
            reset : IN std_logic;                        -- asyncronous reset
            ale : IN std_logic;                          -- addess latch enable
            rdy : OUT std_logic;                         -- data valid signal
            addr : IN std_logic_vector(31 DOWNTO 0);     -- address
            rw : IN std_logic;                           -- read/write signal
            be : IN std_logic_vector(3 DOWNTO 0);        -- byte enables
            blast : IN std_logic;                        -- burst's last word
            ba : OUT std_logic_vector(1 DOWNTO 0);       -- SDRAM bank address
            ras : OUT std_logic;                         -- SDRAM row address
                                                         -- strobe
            cas : OUT std_logic;                         -- SDRAM column address
                                                         -- strobe
            cs : OUT std_logic_vector(3 DOWNTO 0);       -- SDRAM Module select
            dqmb : OUT std_logic_vector(7 DOWNTO 0);     -- SDRAM data mask
            raddr : OUT std_logic_vector(13 DOWNTO 0);   -- SDRAM address
```



```
            we : OUT std_logic );
   end component;



begin
   UUT : sdram_controller
      Port Map (clk,reset,ale,rdy,addr,rw,be,blast,ba,ras,cas,
                cs,dqmb,raddr,we);

-- *** Test Bench - User Defined Section ***
   TB : block
   begin
   ------------------------------------------------------------------------
   --  Process:   clock
   --  Purpose:   40 MHz System Clock
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   clock : PROCESS

     VARIABLE   clktmp           : std_ulogic:= '0';
     CONSTANT   Period           : time := 25 ns;
     VARIABLE   simtime          : integer := 0;
   BEGIN  --  PROCESS clock
     WAIT FOR Period/2;
       clktmp := NOT clktmp;
       clk <= clktmp;
       simtime := simtime + 1;
   END PROCESS clock;

   ------------------------------------------------------------------------
   --  Process:   startup
   --  Purpose:   Initialize the System...
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   startup : PROCESS

   BEGIN  --  PROCESS startup
     reset <= '0';
     WAIT FOR 100 ns;
     reset <= '1';
     WAIT;
   END PROCESS startup;

   ------------------------------------------------------------------------
   --  Process:   stimulus
   --  Purpose:   Stimulate the other Inputs...
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   stimulus : PROCESS

   BEGIN  --  PROCESS stimulus
     ale <= '0';
     addr <= (OTHERS => '0');
     blast <= '1';
     rw <= '0';
     WAIT;
   END PROCESS stimulus;
   end block;
-- *** End Test Bench - User Defined Section ***

end A;

configuration CFG_TB_SDRAM_CONTROLLER_BEHAVIORAL of E is
```

```
         USE ENTITY work.reg_ss(behavioral);
      END FOR;
      FOR I_2 : reg_d
         USE ENTITY work.reg_d(behavioral);
      END FOR;
      FOR I_3 : reg_d
         USE ENTITY work.reg_d(behavioral);
      END FOR;
      FOR I_4 : state_machine
         USE ENTITY work.state_machine(behavioral);
      END FOR;
      FOR I_5 : address_multiplexer
         USE ENTITY work.address_multiplexer(behavioral);
      END FOR;
      FOR I_6 : refresh_timer
         USE ENTITY work.refresh_timer(behavioral);
      END FOR;
   END FOR;
END cfg_sdram_controller_schematic;


-- pragma translate_off

library IEEE;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_misc.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_unsigned.all;

entity E is
end E;

Architecture A of E is

   signal  addr : std_logic_vector (31 downto 0);
   signal  ale  : std_logic;
   signal  be   : std_logic_vector (3 downto 0);
   signal  blast: std_logic;
   signal  clk  : std_logic;
   signal  reset: std_logic;
   signal    rw : std_logic;
   signal   ba  : std_logic_vector (1 downto 0);
   signal  cas  : std_logic;
   signal    cs : std_logic_vector (3 downto 0);
   signal  dqmb : std_logic_vector (7 downto 0);
   signal   ras : std_logic;
   signal   rdy : std_logic;
   signal raddr : std_logic_vector (13 downto 0);
   signal    we : std_logic;

   component sdram_controller
      Port (clk : IN std_logic;                          -- clock input
            reset : IN std_logic;                        -- asyncronous reset
            ale : IN std_logic;                          -- addess latch enable
            rdy : OUT std_logic;                         -- data valid signal
            addr : IN std_logic_vector(31 DOWNTO 0);     -- address
            rw : IN std_logic;                           -- read/write signal
            be : IN std_logic_vector(3 DOWNTO 0);        -- byte enables
            blast : IN std_logic;                        -- burst's last word
            ba : OUT std_logic_vector(1 DOWNTO 0);       -- SDRAM bank address
            ras : OUT std_logic;                         -- SDRAM row address
                                                         -- strobe
            cas : OUT std_logic;                         -- SDRAM column address
                                                         -- strobe
            cs : OUT std_logic_vector(3 DOWNTO 0);       -- SDRAM Module select
            dqmb : OUT std_logic_vector(7 DOWNTO 0);     -- SDRAM data mask
            raddr : OUT std_logic_vector(13 DOWNTO 0);   -- SDRAM address
```



```
            we : OUT std_logic );
   end component;



begin
   UUT : sdram_controller
      Port Map (clk,reset,ale,rdy,addr,rw,be,blast,ba,ras,cas,
                cs,dqmb,raddr,we);

-- *** Test Bench - User Defined Section ***
   TB : block
   begin
   ------------------------------------------------------------------------
   --  Process:   clock
   --  Purpose:   40 MHz System Clock
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   clock : PROCESS

     VARIABLE   clktmp           : std_ulogic:= '0';
     CONSTANT   Period           : time := 25 ns;
     VARIABLE   simtime          : integer := 0;
   BEGIN  --  PROCESS clock
     WAIT FOR Period/2;
       clktmp := NOT clktmp;
       clk <= clktmp;
       simtime := simtime + 1;
   END PROCESS clock;

   ------------------------------------------------------------------------
   --  Process:   startup
   --  Purpose:   Initialize the System...
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   startup : PROCESS

   BEGIN  --  PROCESS startup
     reset <= '0';
     WAIT FOR 100 ns;
     reset <= '1';
     WAIT;
   END PROCESS startup;

   ------------------------------------------------------------------------
   --  Process:   stimulus
   --  Purpose:   Stimulate the other Inputs...
   --  Inputs:
   --  Outputs:
   ------------------------------------------------------------------------
   stimulus : PROCESS

   BEGIN  --  PROCESS stimulus
     ale <= '0';
     addr <= (OTHERS => '0');
     blast <= '1';
     rw <= '0';
     WAIT;
   END PROCESS stimulus;
   end block;
-- *** End Test Bench - User Defined Section ***

end A;

configuration CFG_TB_SDRAM_CONTROLLER_BEHAVIORAL of E is
```

Anhang B

Danksagung

An dieser Stelle möchte ich all jenen danken, die mir bei der Durchführung dieser Arbeit unterstützt haben.
Als erstes danke ich Herrn Prof. Dr. Wolfgang Kühn für die Aufnahme in seine Arbeitsgruppe und die interessante Aufgabenstellung. Sie war die Grundlage für eine (lange) lehrreiche Zeit.
Weiterhin möchte ich den übrigen Mitgliedern der Arbeitsgruppe, bestehend aus Markus, Jörg, Hans, Erik, Michael, Carsten, Ingo, Marc-André und Berengar für die freundliche Aufnahme, das angenehme Arbeitsklima und die fachliche wie moralische Unterstützung danken. Auch die *Ehemaligen* Torsten, Arndt und die beiden Stefans sollten nicht unerwähnt bleiben.
Bei den logistischen und organisatorischen Dingen konnte ich mich voll auf Anita, Marianne, Jürgen, Werner und den Mitarbeitern der Elektronik- und Mechanik-Werkstatt verlassen.
Ein besonderer Dank geht natürlich auch an meine Eltern, ohne deren Unterstützung und Förderung das Studium gar nicht möglich gewesen wäre.
Für die schöne Zeit außerhalb des Studiums möchte ich letztendlich bei all meinen Freunden, insbesondere Carlo, Günter, Nina und Susanne bedanken.

Literaturverzeichnis

[1] R.Porter *et al. Phys. Rev. Lett.*, 79(1229), 1997. und alle darin enthaltenen Referenzen.

[2] Björn Lenkheit. *Elektron-Positron-Paar Emissionen in Pb-Au-Kollisionen bei 158AGeV*. Doktorarbeit, Ruprecht-Karls-Universität,Heidelberg, 1998. und alle darin enthaltenen Referenzen.

[3] W.Weise. Workshop on Dilepton Production in Heavy Ion Reactions. 1994.

[4] M.Herrman et al. *Nucl.Phys.*, 560:411, 1993.

[5] Asawaka et al. . *Phys.Rev.*, 46, 1992.

[6] W.Cassing. BUU Modellrechnung. *Phys.Rep.*, 188:363, 1990.

[7] W.Schön et al. Untersuchung rückstoßfrei produzierter ω-Mesonen am Pionenstrahl der GSI. 1996.

[8] P.A.Cerenkov. The invisible glow of pure liquids under the action of γ -rays. 2, 1934.

[9] G.D.Alekseev et al. Investigation of Self-Quenching Streamer Discharge in a Wire Chamber. *NIM*, 177:385, 1980.

[10] M.Petri. Schnelle Ladungsmuster Erkennung mit feldprogrammierbaren Gate Arrays. Diplomarbeit, II.Physikalisches Institut, JLU-Giessen, 1995.

[11] Mathias Münch. Ein Datenaufnahmesystem mit Echtzeit Bildverarbeitung für Ringabbildende Cerenkov-Detektoren. Diplomarbeit, Technische Universität München, 1995.

[12] J.Lehnert. private Mitteilung.

[13] Michael Traxler. The Matching Unit for the HADES Trigger System. Technical report, II.Physikalisches Institut, JLU-Giessen, 1998.

[14] PCISIG. PCI Specifications, Rev. 2.1.

[15] D.Anderson T.Shanley. *PCI System Architecture*. MindShare, Inc, 1995.

[16] G.Willse E.Solari. *PCI Hardware und Software*. Annabooks, 3 edition, 1994.

[17] PLX Technology. *PCI9080 Datasheet*.

[18] Intel. PC SDRAM unbuffered DIMM Specification.

[19] H.-J. Blank. *Logikbausteine - Grundlagen Programmierung und Anwendung*. Markt und Technik Verlag AG, 1992.

[20] Peter Bellows Micheal J. Wirthlin. A Driver for the 6216-based HotWorks board.

[21] Alessandro Rubini. *Linux Device Drivers*. O'Reilly & Associates, Inc., 1 edition, 1998.

[22] Michael Beck. *Linux Kernel Programmierung*. Addison-Wesley, Inc, 4 edition, 1997.

[23] Douglas L. Perry. *VHDL*. McGraw-Hill,Inc, 2 edition, 1993.

[24] Kevin Skahill. *VHDL for programmable Logic*. Addison-Wesley Publishing Company,Inc, 1996.

[25] SYNOPSYS. *Online-Manual*, 1998.

Ich habe diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

René Becker Gießen, Dezember 1998