Fakultät für Physik und Astronomie
Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von
Christian Jung
aus Heidelberg

1996

Das Pipeline-Auslese-Konzept des CERES-Experimentes und dessen Realisation bei der Auslese der Triggerdaten

Die Diplomarbeit wurde von C. Jung ausgeführt am
Physikalischen Institut der Universität Heidelberg
unter der Betreuung von
Herrn Prof. Dr. P. Glässel

Vorwort

Die vorliegende Arbeit entstand in der Zeit von August 1995 bis Oktober 1996 am Physikalischen Institut der Universität Heidelberg innerhalb der CERES-Arbeitsgruppe.

Allen Mitgliedern dieser Gruppe, insbesondere meinem Betreuer, Dr. Roland Baur sowie Wolfram Seipp, Michael Richter und Andreas Köpfle gilt mein besonderer Dank, sie haben den Rahmen geschaffen, der mir diese Arbeit ermöglichte.

Auf den Editor emacs, der Compiler gcc, das Zeichenprogramm xfig, das Textsystem LaTeX und der Postscript-Viewer ghostview konnte ich mich bei der Erstellung der Programme und Texte uneingeschränkt verlassen. Ich kann nur ahnen, wieviel Arbeit zahlreiche Programmierer freiwillig und kostenlos in diese frei verfügbaren Programme eingebracht haben.

Die Elektronikwerkstatt und die Galvanik des Physikalischen Institutes haben einen nicht unerheblichen Beitrag zum Gelingen dieser Diplomarbeit geleistet, insbesondere haben die Herren Grendus, Rausch, Rusniak und Schumacher mir bei Entwurf und Herstellung der Platinen mit Rat und Tat zur Seite gestanden.

Eine große Zahl von Personen hat, sei es durch Diskussion oder Durchsicht meiner Manuskripte dafür gesorgt, daß diese Arbeit in der jetzigen Form vorliegt. Dies waren (in alphabetischer Reihenfolge): A. Drees, P. Glässel U. Kaps, O. Nix, M. Osterholt, A. Pfeiffer, W. Rau, M. Schulz, und W. Seipp.

Völlig unmöglich wäre die Erstellung dieser Diplomarbeit gewesen ohne die Unterstützung durch Familie und Freunde, die um mein seelisches und körperliches Wohl besorgt waren.

Allen auf dieser Seite Erwähnten gilt mein innigster Dank.

Heidelberg, 12. November 1996 Christian Jung

Zusammenfassung

Die vorliegende Arbeit beschreibt ein neues Auslesekonzept des CERES-Experimentes, das die Totzeit von 1ms auf 300μs senken und dadurch die Zahl der pro *spill* auf Band geschriebenen Ereignisse erhöhen wird.

Zur Verwirklichung dieses Konzeptes bei einem Teil des Detektors wurden zwei VME-Module entworfen, gebaut, getestet und installiert, die die Daten nach einem vorzugebenden Programm selbständig sammeln, über eine Strecke von ca. 25 m übertragen und zur Weiterverarbeitung bereitstellen. Dazu wurden die Vorgaben analysiert, Schaltpläne entworfen und die Ablaufsteuerung der Module in der Hardwarebeschreibungssprache Abel programmiert.

Neben der Test- und Bedienungssoftware für beide Module wurde zur einfacheren Handhabung eine Assemblersprache entworfen, die den Auslesevorgang beschreibt, und ein entsprechender Compiler erstellt. In dieser Sprache wurde dann ein Großteil der zum Betrieb notwendigen Programme entworfen.

Die beiden Module wurden während der Protonstrahlzeit im September 1996 und der der Bleistrahlzeit im Oktober/November 1996 erfolgreich eingesetzt und trugen wesentlich dazu bei, daß die Zahl der pro *burst* aufgenommenen Ereignisse effektiv um einen Faktor 2.5 vergrößert werden konnte.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit befaßt sich mit dem neuen Pipeline-Auslesekonzept des CERES-Experimentes [1, 2, 3, 4, 5, 6], das gegenüber der alten Auslese (Stand 1995) den Vorteil einer deutlich geringeren Totzeit besitzt. Durch die geringere Totzeit kann ein größerer Anteil der aufgetretenen Ereignisse erfaßt werden, bzw. schon bei kleinerer Ereignisrate – und damit geringerer Belastung der Zähler – die Datenaufnahmerate gesättigt werden.

Das Pipeline-Konzept soll die Benutzung eines mehrstufigen Triggersystems ermöglichen, dessen höhere Stufen aus den erfaßten Ereignissen die uninteressanten herausfiltern und nur die interessanten zur Speicherung auf Band freigeben. Dadurch werden in den auf Band gespeicherten Daten Ereignisse angereichert, welche die bei der Analyse geforderten Kriterien erfüllen.

Die Totzeit der Auslese nach dem Pipeline-Konzept wird durch den langsamsten Detektorteil bestimmt. Dies war im Jahr 1995 die Auslese der Triggerdetektoren, die im Gegensatz zu den anderen Subdetektoren nicht aus speziell entworfener Hardware, sondern ausschließlich aus käuflichen Modulen verschiedener Hersteller aufgebaut war und softwaregesteuert ausgelesen wurde. Diese Vorgehensweise ist für das Pipeline-Konzept zu langsam.

Deshalb wurden im zuge der vorliegenden Diplomarbeit zwei neue VME-Module entworfen, gebaut, getestet und installiert, von denen das erste, DACOX (*data collecor and transmitter*, Datensammler und -Sender), die Daten der Triggerdetektoren selbständig nach einem vorgegebenen Programm sammelt und über eine Hardwareverbindung von ca. 25 Metern überträgt. Das zweite Modul, DARX (*data receiver*, Datenempfänger), empfängt sie am anderen Ende dieser Verbindung und stellt sie dort zur weiteren Auslese zur Verfügung.

Zusätzlich wurde die für die Programmierung und den Betrieb der Module notwendige Software geschaffen, wozu vor allem ein Compilerprogramm gehört, das die Anweisungen für die Datenauslese in für die Hardware verständliche Binärcodes übersetzt.

In der Strahlzeit 1996 wurden die neu gebauten Module zusammen mit dem restlichen Detektor nach dem Pipelining-Konzept betrieben. Durch den Übergang zu diesem Konzept, der durch die beiden hier vorgestellten Module überhaupt erst möglich wurde, konnte die Zahl der aufgenomenen verwendbaren Ereignisse um einen Faktor 2.5 erhöht werden.

In Kapitel 2 wird ein Überblick über das CERES-Experiment und die verschiedenen Detektorkomponenten gegeben und das alte Datenauslesesystem dem Pipelining-Konzept gegenübergestellt. In den darauffolgenden Kapiteln wird die neu entwickelte VME-Elektronik beschrieben, deren Spezifikationen sich aus den Bedürfnissen der Pipeline-Auslese ergeben. Dabei werden Aspekte der Hardware und der programmierbaren Logik erläutert und es wird auf die zugehörige Software, speziell das Compilerprogramm, eingegangen. Kapitel 5 erläutert die Vorgehensweise bei Bau, Test und Installation der Module und gibt an, in wieweit die gesteckten Ziele erreicht werden konnten. Der Anhang gibt die im Zuge dieser Diplomarbeit geschaffene Dokumentation wieder.

Die gesamte Arbeit soll einerseits den Leser in die Lage versetzen, die beiden neuen Module zu verstehen und effektiv zu nutzen und andererseits die künftige Weiterentwicklung der Hard- und Software ermöglichen.

2 Das CERES-Experiment

2.1 Physikalischer Hintergrund

Das CERES-Experiment untersucht Elektronenpaare aus ultrarelativistischen Proton-Kern- und Kern-Kern-Kollisionen. Bei solchen Kollisionen sagen theoretische Überlegungen die Bildung eines Quark-Gluon-Plasmas (QGP) voraus, in dem sich die Quarks und Gluonen frei bewegen können.

Zum Nachweis der Bildung des QGP und zur Untersuchung seiner Eigenschaften werden mehrere Signaturen erwogen, nämlich die Dileptonenproduktion, die Produktion direkter Photonen, die J/ψ-Unterdrückung und eine vermehrte Strangeness-Produktion.

Die Untersuchung von Dileptonen und Photonen hat dabei den entscheidenden Vorteil, daß diese das Innerste des 'Feuerballs', wo sie erzeugt wurden, viel leichter ohne Endzustandswechselwirkungen verlassen können als Hadronen, da sie nur schwach und elektromagnetisch wechselwirken. So erhält man direkten Aufschluß über die frühe, heiße Phase der Kollision.

Die im Verlauf der Kollision produzierten Hadronen (vornehmlich π, η, η', ω, ρ) zerfallen ebenfalls zu einem gewissen Prozentsatz in Elektronenpaare und/oder Photonen und stellen deshalb den größten Untergrundbeitrag dar.

2.2 Bisheriges Meßprogramm

Der CERES-Detektor besteht, wie im nächsten Abschnitt nochmals genauer erläutert wird, im wesentlichen aus zwei ringabbildenden Čerenkov-Ring-Detektoren, die im wesentlichen nur auf Elektronen reagieren. Durch Ablenkung in einem Magnetfeld können Ladung und Impuls der Elektronen bestimmt werden. Zwei radiale Siliziumdriftdetektoren in der Nähe des Target dienen der Vertexrekonstruktion und der Rejektion von Konversionspaaren.

Da die gesuchten Ereignisse sehr selten sind (auf 10^5 geladene Teilchen kommt ein Elektronenpaar) wurde während der p-Be und p-Au Strahlzeiten ein schneller Triggerprozessor verwendet, der in den von den RICH-Detektoren gelieferten Daten nach Ringen suchte (vgl. dazu [7, 8]). Es wurden dann ausschliesslich diejenigen Ereignisse aufgezeichnet, in denen der Prozessor mindestens ein Elektronenpaar erkannt hatte.

In S-Au-Kollisionen war die Teilchenmultiplizität gegenüber Proton-Kern-Kollisionen stark erhöht. Die wesentlich höhere Rate an Ringen aus Untergrundprozessen (Konversionen und sog. Dalitzzerfälle) bewirkte ein sehr häufiges Ansprechen des Triggerprozessors. Ausserdem erkannte der Prozessor bei der höheren Photonendichte im UV-Detektor mehr falsche (kombinatorische) Ringe (*fake rings*). Dadurch fiel der Anreichnerungfaktor von 160 bzw. 75 in p-Be bzw. p-Au auf nur 3 in S-Au. Es wurden also deutlich mehr Ereignisse auf Band geschrieben, sodaß ihre Zahl durch die Geschwindigkeit der Datenerfassung limitiert war. Detaillierte differentielle Untersuchungen, z.B. als Funktion der geladenen Teilchenmultiplizität, waren wegen der geringen Statistik nicht möglich.

Bei Pb-Au Reaktionen steigt die Teilchenmultiplizität gegenueber S-Au nochmals stark an, so daß das bisherige Triggerkonzept wirkungslos wird. Stattdessen wurde die Datenerfassung so weit verbessert, daß nahezu alle angebotenen Ereignisse auch auf Band geschrieben werden können. Dies ermöglicht auch die Erhöhung der Statistik, die für differentielle Untersuchungen nötig ist.

2.3 Experimenteller Aufbau

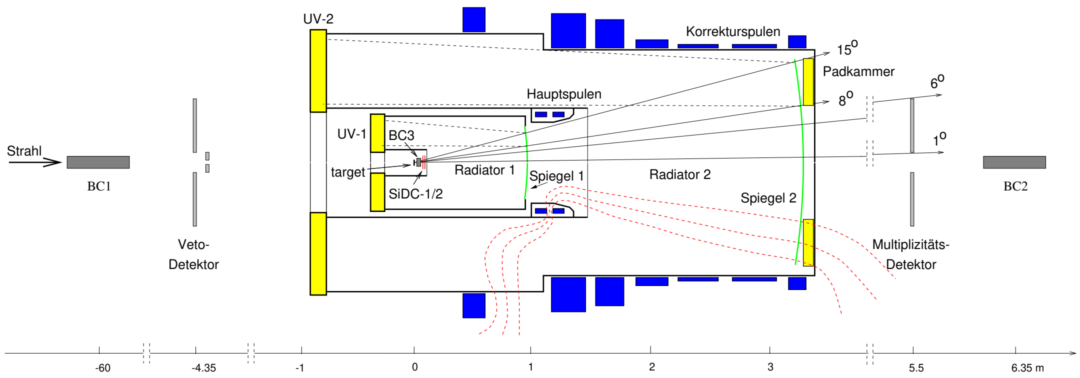


Abbildung 1: Der CERES-Detektor

Das CERES-Spektrometer besteht aus zwei ringabbildenden Čerenkovdetektoren (RICH-1/2), zwei radialen Siliziumdriftdetektoren (SDD-1/2), der sogenannten Pad-Kammer (PADC), und mehreren zusätzlichen Detektoren, die es erlauben, Ereignisse zu erkennen und zu klassifizieren (BC1-3, MD und VD). Der gesamte Aufbau ist in Abbildung 1 dargestellt.

Die beiden RICH-Detektoren dienen der Selektion von Elektronen und zur Impulsmessung. Sie bestehen jeweils aus einem zylindrischen Radiatorvolumen (Radiator 1/2), einem sphärischen Spiegel (Spiegel 1/2) und einem UV-Detektor (UV-1/2). Durch die Wahl von Methan bei Normaldruck als Füllgas des Radiatorvolumens liegt die Čerenkovschwelle bei $\gamma = 32$, sodaß die meisten entstandenen Hadronen im Radiator keine oder deutlich weniger Čerenkov-Photonen erzeugen als die erzeugten Elektronen. Die Čerenkov-Photonen werden durch den sphärischen Spiegel auf einen Ring innerhalb des UV-Detektors fokussiert, wo sie TMAE[1]-Moleküle ionisieren. Die entstandenen Elektronen werden durch Verstärkungsebenen zu einem meßbaren Signal vervielfacht, das mit einer Pad-Ebene von 53800 Pixeln in einem Raster von 2.74 mm (UV-1) bzw. 48400 Pixeln in einem Raster von 7.62 mm (UV-2) nachgewiesen wird.

Zur Impulsmessung werden die Teilchen nach dem Durchgang durch den Spiegel des RICH-1 durch ein Magnetfeld abgelenkt, das von zwei supraleitenden Spulen erzeugt wird. Mehrere (normalleitende) Korrekturspulen sorgen dafür, daß das Feld innerhalb von Radiator-2 parallel zur Teilchenbahn verläuft und RICH-1 feldfrei bleibt. Aus der Winkelablenkung können dann der Impuls und die Ladung des Teilchens bestimmt werden.

[1]Tetrakis-di-Methyl-Amino-Ethylen, Ionisationsenergie 5.4 eV.

Die beiden Siliziumdriftdetektoren (SDD-1/2) dienen zur Vertexrekonstruktion und zur Erkennung von Konversionspaaren. Die durch Ionisation beim Durchgang eines geladenen Teilchens im Halbleiter freigesetzten Ladungen driften radial zum äußeren Rand der Kammern und werden dort an jeweils 360 Anoden als Ladungssignal ausgelesen. Aus der Driftzeit kann die radiale und aus der Anodenposition bzw. dem Schwerpunkt benachbarter Anoden die azimutale Position des Teilchens bestimmt werden. Konversionspaare können nicht immer als zwei disjunkte Spuren aufgelöst werden, sind aber daran zu erkennen, daß sie im Vergleich zu einzelnen Elektronen im Mittel doppelt so viele Ladungen freisetzen.

Zusätzlich liefern die SDDs auch Kandidaten für mögliche Ringmittelpunkte, was bei der Auswertung der von UV-1 aufgenommenen Daten das Auffinden von Ringen erleichtert. Die Daten der SDDs können nicht für UV-2 verwendet werden, da bei der Ablenkung durch das Magnetfeld nur der Polarwinkel θ unverändert bleibt. Deshalb weist eine Vieldrahtproportionalkammer (*multiwire-proportional-chamber*, MWPC) mit Pad-Auslese, die als Pad-Kammer (*pad chamber*, PADC) bezeichnet wird, die Teilchen nach dem Durchgang durch den Spiegel von RICH-2 nochmals nach.

Um Interaktionen zu erkennen und daraufhin die Auslese starten zu können (Trigger) ist das Experiment mit mehreren weiteren Detektoren ausgestattet. Dies sind die drei Strahldetektoren (BC1, BC2 und BC3), der Multiplizitätsdetektor (MD) und der Veto-Detektor (VD), die in ihrer Gesamtheit als Triggerdetektoren bezeichnet werden.

Die drei Strahldetektoren sind Čerenkovdetektoren, die Luft als Radiatorgas benutzen. Sie sind 60 m vor dem Target (BC1), direkt hinter dem Target (BC3) und nach dem Detektor (BC2) installiert (vgl. Abbildung 1) und geben Informationen über durchgegeangene Blei-Ionen. Das entstandene Čerenkovlicht wird mit Sekundärelektronenvervielfachern (SEV) nachgewiesen und dann einer Auswerteelektronik zugeführt. Dabei werden vier Fälle unterschieden:

1. (*interaction trigger*) Ein Blei-Ion durchquert BC1, aber nicht BC3; es wechselwirkte also innerhalb des Target.

2. (*beam trigger*) Ein Blei-Ion durchquert BC1, es tritt also ein Strahleilchen in den Detektor ein.

3. (*clear passage*) Ein Blei-Ion durchquert alle drei Strahldetektoren; es erfährt also keine Ablenkung.

4. (*downstream interaction*) Ein Blei-Ion durchquert BC1 und BC3, jedoch nicht BC2, wechselwirkt also erst nach dem Target.

Anhand der Signale von BC1, BC2 und BC3 kann also festgestellt werden, wann ein Blei-Ion in den Detektor eingetreten ist und ob und wo es zu einer Wechselwirkung gekommen ist.

Der Multiplizitätsdetektor (MD) ist eine in 24 Sektoren eingeteilte Wand von Plastik-Szintillatoren, die hinter dem Detektor aufgebaut ist und einen Winkelbereich von 1° bis 6° abdeckt (vgl. Abbildung 1). Das im Szintillator entstandene Licht wird für jeden Sektor getrennt mit einem SEV nachgewiesen. Die Summe der am Ausgang der SEV gemessenen Ladungssignale gibt Aufschluß über die Multiplizität des Ereignisses und erlaubt es, selektiv Ereignisse hoher Multiplizität aufzuzeichnen.

Der Veto-Detektor (VD) ist vor dem Experiment aufgebaut und besteht aus insgesamt 11 Plastikszintillatoren, die von SEV ausgelesen werden und es erlauben, Ereignisse auszusondern, bei denen ein Teilchen außerhalb des Strahls in den Detektor eintritt, etwa bei Wechselwirkungen innerhalb des Strahlrohres oder in BC1 (*upstream interaction*).

Die Messungen dieser Triggerdetektoren erlauben es, Ereignisse zu erkennen und zu klassifizieren. Durch eine zusätzliche Logik kann ein ganzzahliger Bruchteil jeder Ereignisklasse aufgezeichnet werden, etwa 1/3 aller *interaction trigger* mit niedriger Multiplizität, und alle *interaction trigger* mit hoher Multiplizität.

2.4 Die Auslese (Stand 1995)

Die Ausleseelektronik von RICH-1, RICH-2 und PADC sind fast identisch, wobei sich jedoch aus dem unterschiedlichen Pad-Abstand Unterschiede im mechanischen Aufbau ergeben.

Direkt auf die Pad-Ebene von RICH-1 sind Module aufgesteckt, die ladungsempfindliche Vorverstärker und Analog-Digital-Wandler (AD-Wandler, ADCs) enthalten. Von einer im Kontrollraum befindlichen Elektronik (dem sogenannten Kontrollmodul) gesteuert kann jedes dieser Module die Ladungen von 8×32 Pads speichern und dann nacheinander in einen digitalen Wert wandeln. Je 16 Module werden zu einer sogenannten Auslesekette zusammegefaßt und über gemeinsame Datenleitungen ausgelesen. Die Datenleitungen der insgesamt 16 Ketten führen in den Kontrollraum, wo jede mit einem sogenannten Receiver verbunden ist, der die Daten weiterverarbeitet.

RICH-2 und die Pad-Kammer haben ein identisches Auslesesystem, im Gegensatz zu RICH-1 werden hier jedoch Module verwendet, die die Ladungen von 11×11 Pads speichern können und noch keinen AD-Wandler enthalten. Bis zu 16 dieser Module werden über ein gemeinsames Flachbandkabel von einem AD-Wandler-Modul ausgelesen. Vier am Detektor angebrachte Mutterplatinen, auf die jeweils maximal 16 solche ADC-Module aufgesteckt sind, fassen die Daten von je vier ADC-Modulen zusammen und leiten sie an einen im Kontrollraum befindlichen Receiver weiter.

Das Ladungssignal an den Anoden der SDD wird verstärkt, von Pulsformern (*Shaper*n) in annähernd gaußförmige Pulse umgewandelt und nochmalig verstärkt. Die Digitalisierung erfolgt durch schnelle sogenannte Flash-ADCs (FADCs) mit einer Taktfrequenz von 50 MHz. Mehrere Scanner-Module, welche die digitalisierten Werte nach Treffern (*hits*) durchsuchen und Werte unterhalb einer vorgegebenen Schwelle entfernen, leiten die Daten in den Kontrollraum weiter.

Im Kontrollraum gibt es für jeden Subdetektor ein eigenes VME-Crate, das ein Kontrollmodul (CTRL), bis zu 16 der bereits angesprochenen Receiver und ein Interface-Modul (MM-IF) enthält (siehe Abb. 2). Das Kontrollmodul steuert auf ein Triggersignal hin die gesamte Ausleseelektronik am Detektor, die die Daten erfaßt und an die Receiver weiterleitet. Dort werden sie reduziert, komprimiert und über den FDCS (*Fast Data Collection System*)-

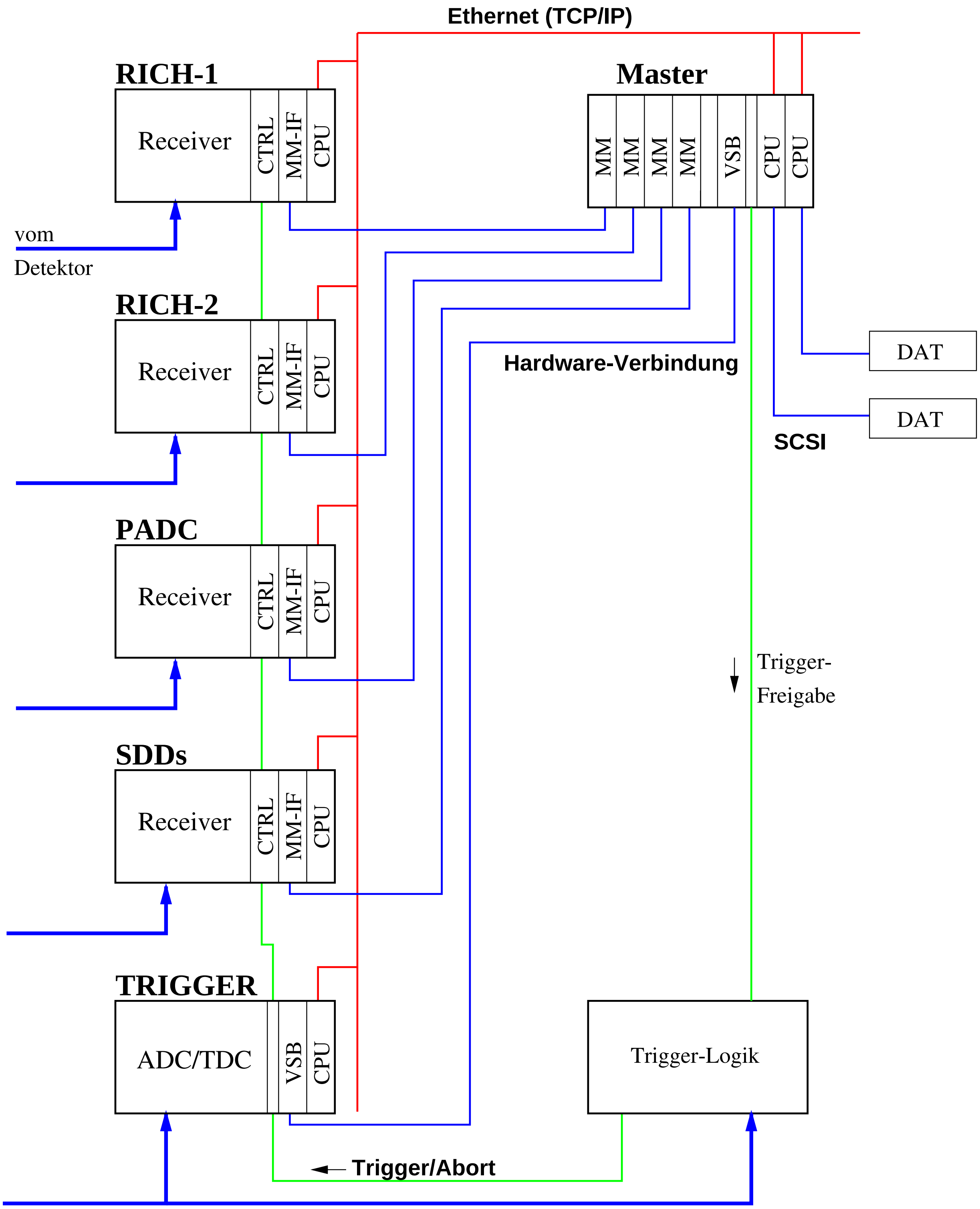


Abbildung 2: Die Auslese-Hardware (Stand 1995)
Die direkt am Detektor befindliche Ausleseelektronik ist nicht dargestellt.

Bus, der die Receiver auf der Rückseite miteinander verkettet, bis zum Interface-Modul durchgegeben. Dieses sendet sie über ein *twisted-pair*-Kabel an ein Memory-Modul weiter, das zusammen mit den Memory-Modulen der anderen Subdetektoren im Mastercrate sitzt (siehe Abbildung 2 rechts oben). Die beiden dort vorhandenen CPUs lesen abwechselnd ein komplettes Ereignis aus den Memorymodulen aus und speichern es in ihrem internen Speicher. Erst wenn dies abgeschlossen ist, ist das Datenerfassungssystem bereit, ein neues Ereignis aufzunehmen, und die Triggerlogik wird wieder freigegeben. Da der Strahl nur für jeweils 4.8 Sekunden aktiv ist (*burst*) und dann eine Pause von 15 Sekunden eintritt (*out-of-burst*), haben die CPUs in dieser Pause ausreichend Zeit, die gesammelten Daten über ihre SCSI-Schnittstelle auf DAT-Bänder zu schreiben.

Auch die von den Triggerdetektoren erfaßten Daten und die Einteilung des Ereignisses in eine bestimmte Klasse müssen auf Band geschrieben werden, damit sie für eine spätere Analyse zur Verfügung stehen. Dazu werden die an den Ausgängen der SEV der einzelnen Triggerdetektoren anliegenden Ladungssignale in ADC-Modulen vom Typ LeCroy 1182 digitalisiert. Außerdem werden verschiedene Logiksignale, wie etwa die Ausgänge der in Abschnitt 2.3 genannten Auswerteelektronik der Strahldetektoren (*interaction trigger*, *beam trigger*, *clear passage* und *downstream interaction*) von einem TDC-Modul (*Time-to-Digital-Converter*) vom Typ LeCroy 1172 überwacht. (Das verwendete Modul kann pro Kanal die Zeitpunkte von bis zu 16 Pegeländerungen in einem Zeitfenster von 64 μs festhalten. Anhand dieser Informationen können bei der Analyse Ereignisse ausgesondert werden, bei denen kurz vor oder nach der Interaktion ein weiteres Strahlteilchen den Detektor durchquert hat (*before-after-protection*). Die dabei erzeugten δ-Elektronen verursachen einen hohen Untergrund und erschweren die Mustererkennung.) Diese ADC- und TDC-Module befinden sich im sogenannten Triggercrate (vgl. Abbildung 2 links unten), das mit der restlichen Triggerlogik ca. 25 Meter vom Kontrollraum entfernt aufgebaut ist. Auf ein Startsignal hin liest eine ebenfalls dort befindliche CPU die in den TDC- und ADC-Modulen gespeicherten Daten aus und schreibt sie über eine VSB-Verbindung in einen Speicher im Mastercrate, wo sie zusammen mit den Daten der übrigen Detektoren von den CPUs des Mastercrates ausgelesen und auf Band geschrieben werden.

Der zeitliche Verlauf eines solchen Auslesezyklus kann wie folgt skizziert werden (vgl. dazu Abbildung 3):

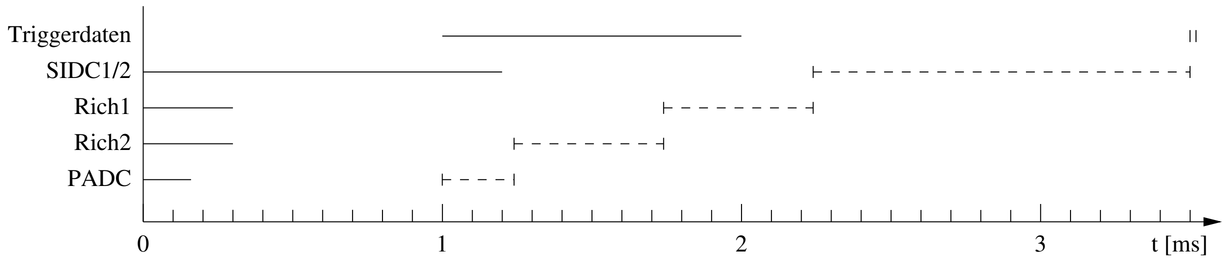


Abbildung 3: Der zeitliche Ablauf der Auslese (Stand 1995)
Die Zeit für die Erfassung der Daten ist mit durchgezogenen Linien gekennzeichnet; die gestrichelten Linien geben die Zeit für die Übertragung über den VME-Bus an.

Etwa 200 ns nach der Interaktion hat die Triggerlogik entschieden, ob dieses Ereignis of Band geschrieben werden soll. Daraufhin starten die Kontrollmodule die Auslese von RICH-1, RICH-2, PADC sowie der beiden SDD und die CPU des Triggercrates beginnt, die in den ADC- und TDC-Modulen erfaßten Werte auszulesen.

Nach etwa 300 μs hat die schnelle Ausleseelektronik von RICH-1, RICH-2 und PADC die durchschnittlich etwa 5-10 kByte pro Ereignis und Detektor bis in die Memory-Module transferiert, wo sie zur Auslese bereitstehen.

Bei den SDD dauert dieser Prozess etwa die vierfache Zeit, 1.2 ms, da zum einen eine etwas größere Datenmenge von insgesamt 25 kByte pro Ereignis verarbeitet werden muß, zum anderen die von den recht langsamen Scannern (15 MHz) vorgenommene Datenreduktion zusätzliche Zeit kostet.

Bei den Triggerdetektoren nimmt diese Phase trotz der geringen Datenmenge von nur 0.2 kByte 2 ms in Anspruch, da die CPU gegenüber der Spezial-Hardware der anderen Detektoren den Nachteil einer deutlich langsameren softwaremäßigen Ablaufsteuerung hat und auch nur mit einer Verzögerung von etwa einer Millisekunde auf das Triggersignal reagieren kann.

Ähnlich wie im Triggercrate beginnen auch die CPUs des Mastercrates nach etwa einer Millisekunde damit, die Daten des gesamten Ereignisses, durchschnittlich 50 kByte, mit einer Rate von etwa 20 MByte/s auszulesen. Dies dauert etwa 2.5 ms, sodaß ein gesamter Auslesezyklus, vom Zeitpunkt des Ereignisses bis zur erneuten Freigabe des Triggers, etwa 3.5 ms dauert.

Diese Totzeit des Auslesesystems begrenzt die Zahl der pro *burst* aufgenommenen Ereignisse. Theoretisch könnten bei einer Totzeit von 3.5 ms in jedem Burst von 4.8 Sekunden Länge 1370 Ereignisse ausgelesen werden. Dieser Wert kann jedoch nicht erreicht werden, da nach dem Ende der Auslese gewartet werden muß, bis die Triggerelektronik die Auslese neu startet. Die Triggerrate kann jedoch nicht beliebig erhöht werden, da dadurch die Wahrscheinlichkeit steigt, daß kurz vor oder nach einem Ereignis weitere Strahlteilchen den Detektor passieren. Außerdem würden dadurch UV1/2 zu stark belastet. Legt man eine Rate von 1000 Triggern pro *burst* zugrunde, so können davon nur 530 erfaßt werden. Eine Verkürzung der Totzeit (etwa auf 300 μs) würde es ermöglichen, daß selbst bei einer höheren Triggerrate von 1250 pro *burst* noch 1150 Ereignisse auf Band geschrieben werden können. (Alle Zahlen dieses Abschnitts stammen aus [6].)

2.5 Das Pipeline-Konzept

Dem neuen Auslesekonzept, das als Pipeline-Auslese bezeichnet wird, liegt die Idee zugrunde, neue Trigger nicht erst dann zuzulassen, wenn das letzte Ereignis komplett im Speicher der CPU steht, sondern bereits wenn alle Kontrollmodule signalisieren, daß die Daten komplett über den FDCS-Bus in das Memorymodul-Interface übertragen worden sind. Dann würde die Totzeit der gesamten Datenerfassung nur durch die Totzeit des langsamsten Subdetektors bestimmt.

Dies funktioniert jedoch nur, wenn die alten Memorymodul-Interfaces gegen neue, sogenannte Pipeline-Module ausgetauscht werden (vgl. Abbildung 4), die mehrere in schneller

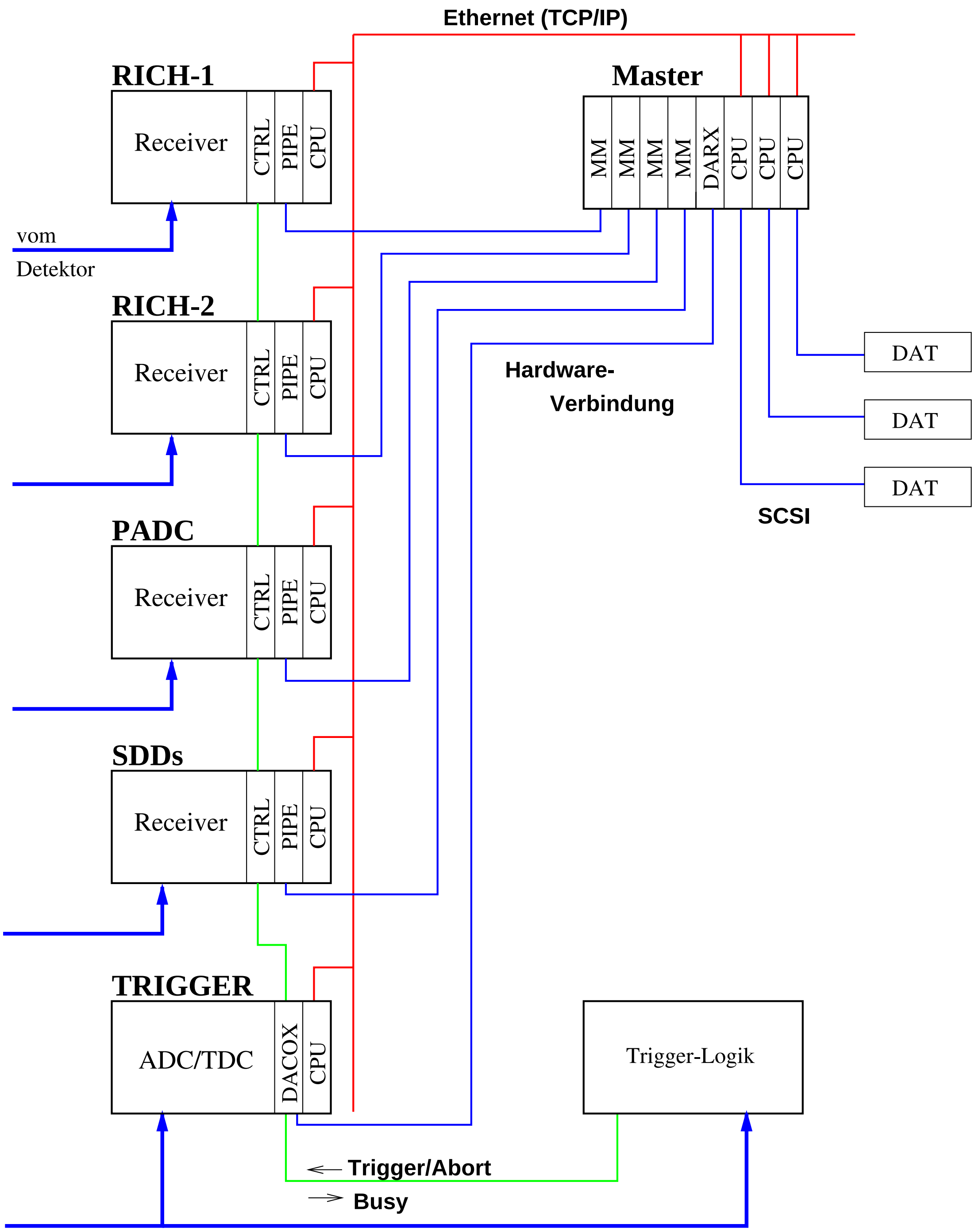


Abbildung 4: Die Auslese-Hardware (Pipelining)
Im Vergleich zur alten Konfiguration (Abbildung 2) sind hier die Memorymodul-Interfaces durch Pipeline-Module ersetzt, die Auslese der Triggerdetektoren wird von den DACOX/DARX-Modulen vorgenommen und die Freigabe des Triggers geschieht nicht mehr durch die CPUs des Mastercrates, sondern durch die Kontrollmodule.

Folge erfaßte Ereignisse so lange speichern, bis die CPUs sie ausgelesen haben.

In diesem Falle begrenzt die Übertragungsrate über den VME-Bus nicht den minimalen Abstand zwischen zwei aufeinanderfolgend erfaßten Ereignissen, sondern nur die durchschnittliche Anzahl von Ereignissen pro Sekunde. Eine Synchronisation der Ausleseprozesse mit der Triggerlogik entfällt, da beides durch die Pipeline-Module entkoppelt wird. Dies vermeidet die dabei auftretende Totzeit von 1 ms.

Um diese Totzeit auch bei der Auslese der Triggerdetektoren zu vermeiden und die Totzeit dieses Detektorteils in die selbe Größenordnung wie bei den anderen Detektoren zu bringen, muß auch hier eine Spezial-Elektronik verwendet werden, die die langsame Softwaresteuerung umgeht und sich schnell mit der Triggerlogik synchronisieren läßt.

Ein neu entworfenes Scannersystem soll die Totzeit der beiden SDD um einen Faktor vier verkleinern. Dies soll erreicht werden, indem einerseits ein neues Scannermodul verwendet wird, das die digitalisierten Daten mit der doppelten Geschwindigkeit nach *hits* durchsucht, und andererseits die Zahl der Scannermodule verdoppelt wird, wodurch die Totzeit nochmals halbiert wird.

Der Engpaß der Übertragung des gesamten Ereignisses über den VME-Bus soll durch einen Neuentwurf der Memory-Module weiter entschärft werden. Trotzdem wird man auf lange Sicht dort an Grenzen stoßen. Deshalb wird einerseits erwogen, den VME-Bus durch ein anderes Bussystem zu ersetzen oder aber aus den erfaßten Ereignissen bereits diejenigen herauszufiltern, die physikalisch nicht interessant sind. Es sind zwei Mechanismen vorgesehen um diese Ereignisse aus den Daten zu entfernen:

Beim *Abort*-Vorgang, der innerhalb von $20\mu s$ nach dem Start der Erfassung eines Ereignisses ausgelöst werden kann, wird die am Detektor befindliche Elektronik zurückgesetzt und werden die bereits in den Receivern gespeicherten Daten gelöscht. Dadurch gelangen die Daten gar nicht erst in das Pipeline-Modul. Mit Hilfe des Abort-Vorgangs konnte 1996 die *before-after-protection* (vgl. Seite 7) bereits während der Datennahme durchgeführt werden.

Ist das Ereignis bereits im Pipeline-Modul gespeichert, so kann es dort aus dem Speicher entfernt werden, und der belegte Speicher zur Speicherung anderer Ereignisse freigegeben werden. Dieses Verfahren wird als *event removal* bezeichnet; die zur Entscheidung zur Verfügung stehende Zeit hängt von der durchschnittlichen Zahl erfaßter Ereignissen ab und beträgt etwa 3.8 Millisekunden.

Da es für jeden Subdetektor eine eigene Pipeline gibt, muß stets sichergestellt werden, daß die CPUs immer Daten desselben Ereignisses zusammenfassen. Dies geschieht, indem in jedem Subdetektor die Ereignisse seit Beginn des *bursts* gezählt werden und diese Zahl (die sogenannte Ereignis-Nummer oder *event number*) den Daten beigefügt wird. Am anderen Ende der Pipeline werden die Daten in die jeweiligen Memory-Module transferiert und von den CPUs ausgelesen. Dabei wird auch die Ereignis-Nummer auf Gleichheit geprüft.

In der Strahlzeit 1996 wird die Datenauslese auf der Basis der alten Hardware (also ohne Pipeline-Modul) als sogenannte Ein-Ereignis-Pipeline (*1-event pipeline*) betrieben. Dabei wird der Trigger bereits dann freigegeben, wenn das letzte Ereignis von den Receivern bis zum Memorymodul-Interface weitergegeben wurde. Das nächste Ereignis verbleibt dann so lange in den Receivern, bis die Memorymodule zur Aufnahme neuer Daten bereit sind. Dadurch können zwei schnell aufeinanderfolgende Ereignisse aufgenommen werden und wegen der fehlenden Synchronisation mit der Triggerlogik entfällt die Totzeit von 1 ms bis die CPUs mit der Auslese beginnen. Die für das Pipelining nötigen Signale zur Triggerfreigabe werden von einem neu gebauten Kontrollmodul erzeugt, das auch in der voll ausgebauten Pipeline-Auslese verwendet werden wird.

Zur Auslese der Triggerdetektoren werden zwei neue Module verwendet, die im Rahmen der vorliegenden Diplomarbeit entworfen, gebaut, getestet und installiert wurden. Ohne diese neuen Module wäre eine Einführung des Pipeline-Systems kaum möglich gewesen, und selbst wenn, so hätte sich dadurch die Totzeit des Gesamtsystems vergrößert. Mit den neuen Modulen konnte die Totzeit im Bereich der Triggerdetektoren entscheidend verkürzt werden, sodaß diese nicht mehr zur Totzeit des Detektors beitragen. Sie können außerdem mit geringen oder gar keinen Änderungen auch im voll ausgebauten Pipeline-System verwendet werden.

3 Das DACOX-Modul

3.1 Vorgaben

Aus dem in Kapitel 2.5 Gesagten ergeben sich die Rahmenbedingungen für das neu zu bauende DACOX-Modul, welches im Trigger-Crate die Daten ausliest und in das Mastercrate weitersendet.

Dieses soll auf externe Signale hin mehrere verschiedene Programme selbständig ausführen: Zu Beginn jedes *bursts* soll ein Programm abgearbeitet werden, das alle auszulesenden Module zurücksetzt und für das erste Ereignis vorbereitet. Dann wird bei jedem von der Triggerlogik ausgewählten Ereignis ein zweites Programm abgerufen, das alle benötigten Daten ausliest und die Module auf das nächste Ereignis vorbereitet. Das dritte Programm soll am Ende jedes *bursts* *burst*-spezifische Daten wie etwa die Zahl der insgesamt in BC1 nachgewiesenen Strahlteilchen auslesen. Ein zusätzliches viertes Programm könnte genutzt werden, um Testdaten zu generieren. Die vier Programme werden im folgenden als `TRIG0`, `TRIG1`, `TRIG2` und `TRIG3` bezeichnet.

Um im Rahmen des Pipelining (siehe 2.5) einen reibungslosen Ablauf zu gewährleisten, muß DACOX wie die Ausleseelektronik aller anderen Subdetektoren ein BUSY-Signal generieren, damit der Trigger freigegeben werden kann, sobald alle Daten erfaßt sind. Bei der Verwendung eines mehrstufigen Triggersystems soll es durch ein ABORT-Signal angewiesen werden können, die Auslese abzubrechen.

Die Daten sollen über eine noch zu definierende Hardware-Verbindung in das ca. 25 Meter entfernte Mastercrate zum DARX-Modul übertragen werden. Dabei ist eine Fehlerprüfung von Vorteil, um größtmögliche Datensicherheit zu gewährleisten.

Zusammen mit dem DARX-Modul am anderen Ende dieser Verbindung soll DACOX pipeline-fähig sein. Das bedeutet, daß auf der Eingangsseite wieder neue Daten eingelesen werden können, selbst wenn noch nicht alle Daten von den CPUs ausgelesen wurden. Um aufeinanderfolgende Ereignisse voneinander zu trennen muß das Ende jedes Ereignisses besonders markiert werden. Auch bei der Einführung des *event removal* soll sich das Modul in dieses System einfügen.

Bei der Abarbeitung der Programme sollen `D8EO`-, `D16`- und `D32`-VME Zyklen[2] durchgeführt werden. Dabei sollen alle Adressierungsarten möglich sein, nämlich *short* (`A16`), *standard* (`A24`) und *extended* (`A32`).Das Modul selbst soll mit `D8EO`- oder `D16`-Zyklen über den VME-Bus anzusprechen sein.

Es sollen bedingte Verzweigungen möglich sein. Dies wird benutzt um auf ABORT-Signale und Zugriffsfehler (*bus error*) zu reagieren.

Da bei einigen ADC-Modulen zunächst gewartet werden muß, bis ein Bit des Statusregisters anzeigt, daß die Wandlung in digitale Werte abgeschlossen ist, muß der Programmablauf auch durch die Daten beeinflußbar sein.

Die Abarbeitung eines Programms zur Auslese von ca 100 Datenworten soll dabei eine Zeit von maximal etwa 300μs in Anspruch nehmen, damit die Totzeit der Triggerdaten-

[2]Zur Erklärung dieser und anderer VME-Fachbegriffe siehe [9] oder [10]

auslese in etwa der der anderen Detektoren entspricht.

Zusätzlich sollten möglichst viele Möglichkeiten zur Fehlersuche und zum Testen des Moduls vorgesehen werden, um im späteren Aufbau auftretende Fehler schnell lokalisieren zu können.

3.2 Die Hardware

Ein interner Bus (IB, vgl. Abbildung 5) verbindet die verschiedenen Datenquellen und -senken auf dem Modul, da zwischen ihnen viele verschiedene Datenverbindungen möglich sein sollen. Da auch die VME-Zugriffe von außen (mit einer maximalen Breite von 16 Bit) über diesen Bus laufen sollen, hat er zweckmäßigerweise ebenfalls eine Breite von 16 Bit.

Um den Aufbau und die Fehlersuche des Moduls zu erleichtern, wurde ein IB-Anschluß vorgesehen, an dem die wichtigsten Signale mit einem Oszilloskop oder Logik-Analysator abgegriffen werden können. Zwei zusätzliche Kontakten können nachträglich mit Steuersignalen verbunden werden, sodaß hier eventuell auch eine zusätzliche Datenquelle oder -senke angeschlossen werden kann.

3.2.1 Der Programmspeicher (RAM)

Für den auszuführenden Programmcode muß auf dem Modul ein Speicher vorhanden sein, der über den VME-Bus beschreibbar sein soll. Um der Anbindung an den VME-Bus gerecht zu werden, wird ein 16 Bit breiter RAM-Bereich einer Größe von 16K-Worten verwendet, der aus vier SRAM-chips vom Typ 6264 aufgebaut ist. Diese Bausteine haben zum einen gegenüber dynamischen RAMs den Vorteil eines geringeren Schaltungsaufwandes, zum anderen gibt es zahlreiche kleinere und größere Speicherchips mit ähnlicher Pinbelegung. Dies ermöglicht eine Anpassung der Speichergröße an die jeweiligen Bedürfnisse oder die Verwendung von EPROMs, um den Programmcode nicht jedesmal neu laden zu müssen.

3.2.2 Der Datenspeicher (FIFO)

Ein 16 Bit breiter FIFO(*First-In-First-Out*)-Speicher speichert die über den VME-Bus eingelesenen Daten zwischen, falls sie nicht sofort in das Mastercrate weitergereicht werden können. Es werden zwei jeweils 9 Bit breite Chips aus der IDT7200-er Familie von IDT oder der CY7C420-er Familie von Cypress verwendet, die in verschiedenen Größen zwischen 512 und 4096 Speicherplätzen erhältlich und zueinander pinkompatibel sind, sodaß durch ihren Austausch die Speichergröße variiert werden kann.

Da der Datenspeicher somit 18 statt 16 Bit breit ist, kann eines der beiden zusätzlichen Bits (EOE, *End Of Event*) im Rahmen des Pipelining dazu verwendet werden kann, aufeinanderfolgende Ereignisse im FIFO voneinander zu trennen. Das andere Bit, (EOL, *End Of Label*), kann benutzt werden, um die Ereignisse in Unterabschnitte für die verschiedenen Module zu zerlegen. Dadurch können solche Abschnitte ohne Kenntnis des jeweiligen Datenformats schnell übersprungen werden. Die Bedeutung dieser Zusatzbits kann aber

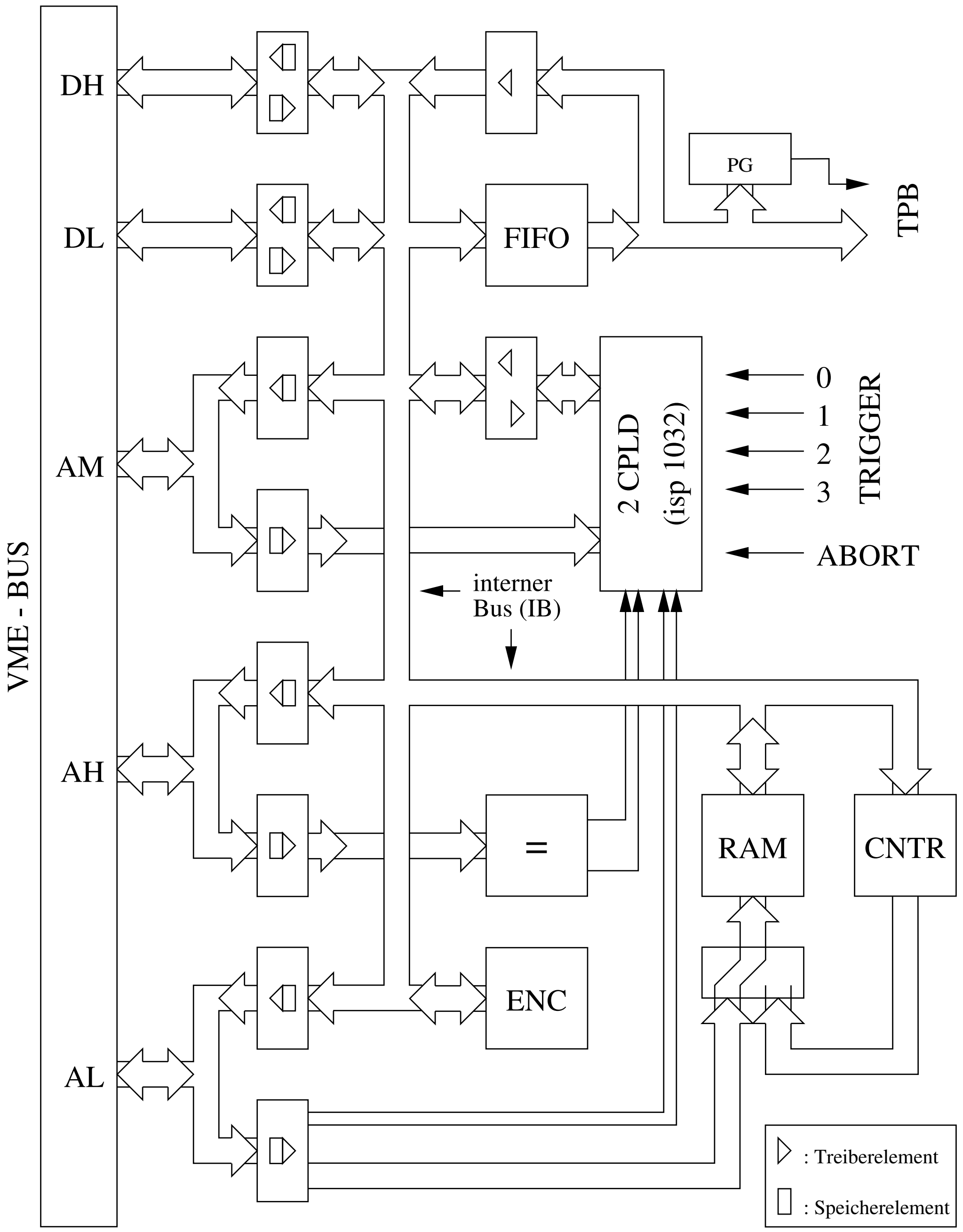


Abbildung 5: Blockdiagramm des DACOX-Moduls
Auf ein Triggersignal an den Trigger-Eingängen (rechts im Bild) hin beginnen die CPLDs mit der Ausführung des zuvor im RAM gespeicherten Programms, welches VME-Zyklen ausführt und die dabei gelesenen Daten ins FIFO transferiert. Die im FIFO vorhandenen Daten werden automatisch über den *twisted-pair-bus* (TPB) ins Mastercrate weitergeleitet, wobei der Paritätsgenerator (PG) der Fehlererkennung dient.

von der Software frei gewählt werden; aus der Sicht der Hardware handelt es sich einfach um zwei zusätzliche Datenbits.

Der an den FIFO-Chips vohandene ReTransmit- (RT-) Eingang eröffnet eine einfache Möglichkeit zur Fehlersuche: Wird das Modul ohne Verbindung zum Mastercrate getestet, so hätte man normalerweise keinen Zugriff auf die Daten im FIFO, da diese bereits während der Programmausführung automatisch weitergeleitet werden und somit verloren sind. Durch die RT-Funktion können diese Daten nun ein zweites Mal, diesmal über den VME-Bus, ausgelesen und überprüft werden. Zu diesem Zweck wird die Ausgangsseite des FIFO mit einem Treiberbaustein (siehe Abbildung 5, oberhalb des FIFO) auf den IB zurückgeführt.

3.2.3 Datenübertragung und externe Steuerung

Für die Verbindung mit dem Mastercrate bietet sich wegen der großen Entfernung die Verwendung von 'twisted-pair'-Kabeln und differentiellen Treibern/Empfängern (AM26LS31 / 32) an ('Twisted-Pair-Bus', TPB). Neben den 16+2 Datenbits wird zur Fehlererkennung ein Paritätsbit übertragen, das von zwei Paritätsgeneratoren vom Typ 74ALS280 erzeugt wird. Mit einer Kurzschlußbrücke kann zwischen gerader und ungerader Parität gewählt werden. Zwei Steuersignale (DS_TPB und FULL_TPB) sorgen für einen reibungslosen Ablauf der Übertragung. Insgesamt werden also mindestens

$$2 \times (16 + 2 + 1 + 2) = 42$$

Adern benötigt. Deshalb werden die nächstgrößeren Flachkabelstecker mit 50 Kontakten verwendet.

Zur Verbindung der BUSY-, ABORT- und TRIG0-3-Signale mit der aus NIM-Modulen aufgebauten Triggerlogik muß zwischen den auf dem Modul verwendeten TTL-Logikpegeln von 0V (logisch 0) und +5V (logisch 1) und den NIM-Logikpegeln von 0V (logisch 0) und -0.8V (logisch 1) gewandelt werden. Dies wird mit wenigen ECL-Bausteinen realisiert.

3.2.4 Ablaufsteuerung

Zwei CPLDs vom Typ isp1032 der Firma Lattice[3] realisieren die Ablaufsteuerung des Moduls und zahlreiche weitere Logikfunktionen, auf die im Kapitel 3.3 näher eingegangen wird. Diese Bausteine haben jeweils 64 frei konfigurierbare I/O-Pins und sind über speziell dafür vorgesehene Pins programmierbar, ohne daß sie dazu aus der Schaltung ausgebaut werden müßten.

Das Modul kann sich in zwei verschiedenen Modi befinden:

- Im Grundzustand (`SLAVE`-Modus) können RAM und FIFO über VME ausgelesen werden. Das Modul reagiert jedoch nicht auf Triggersignale.

[3]Mehr Information zu diesen Bausteinen findet sich in [11].

- Im `MASTER`-Modus ist der Zugriff auf RAM und FIFO verboten; das Modul erwartet Triggersignale oder arbeitet das geladene Programm ab.

Diese Vorgehensweise verhindert, daß während der Programmausführung der Inhalt des RAM-Speichers verändert oder Daten aus dem FIFO ausgelesen werden. Der `MASTER`-Modus kann nochmals unterteilt werden, nämlich in den Zustand, in dem auf Trigger gewartet wird, und den Zustand, in dem das Modul mit der Abarbeitung des Programmes beschäftigt ist. Nur im Wartezustand ist das BUSY-Signal nicht aktiv, sodaß die Triggerlogik nur dann freigegeben wird, wenn das Modul auch auf Trigger reagiert.

3.2.5 Die Anbindung an den VME-Bus

Zur Ausgabe der Adresse auf den VME-Bus werden vier 74AS574 verwendet, die während der Programmabarbeitung mit Daten aus dem RAM geladen werden können und dann bei VME-Zugriffen die Adreßleitungen treiben (vgl. Abbildung 5 links unten). Bei einem Zugriff von außen wird die am VME-Bus anliegende Adresse bei der fallenden Flanke des $\overline{\mathrm{AS}}$-Signals von weiteren vier 74AS574 gespeichert. Deren Ausgänge werden dann entweder von den Adreßkomparatoren mit der Basis-Adresse des Moduls verglichen (AH, A16-31), oder den Adreßleitungen des RAMs zugeführt (AL, A01-14). A15 und einige der unteren Adreßleitungen sind mit dem in den CPLDs befindlichen Adreßdekoder (AD) verbunden, der feststellt, ob es sich um eine RAM-, FIFO-, ENC- oder CSR- Adresse handelt. Die Adreßmodifizierer (AM) werden genau wie die Adreßleitungen von je einem 74AS574 empfangen und getrieben.

Der Anschluß des Moduls an den VME-Datenbus ist mit vier 74ALS646-1-Bausteinen realisiert (siehe Abbildung 5 links oben). Diese Bausteine besitzen ein Eingangs- und ein Ausgangsregister und können zusätzlich auch transparent geschaltet werden. Für VME-Schreibzugriffe wird zunächst das Ausgangsregister mit den zu schreibenden Daten geladen und diese dann auf den VME-Bus gegeben. Bei Lesezugriffen werden die Daten vom VME-Bus in das Eingangsregister gespeichert, von wo sie z.B. in das FIFO gebracht werden können. Die Treiberbausteine werden in zwei Register zu je 16 Bit eingeteilt, die getrennt angesteuert werden können; dadurch können bei `D32`-Zyklen oberes und unteres Wort getrennt über den nur 16 Bit breiten internen Bus übertragen werden. Bei Lese- oder Schreibzugriffen von außen (etwa auf das Statusregister CSR) wird der Transparentmodus benutzt und die Daten direkt vom internen Bus zum VME-Bus (oder umgekehrt) weitergegeben.

3.2.6 Die Programmausführung

Der Programmzähler (CNTR), der mit zwei Bausteinen vom Typ 74AS867 realisiert ist, gibt die Speicherzelle an, bei der sich das Programm zum jeweiligen Zeitpunkt befindet (CNTR, in Abbildung 5 unten rechts). Sein Eingang ist mit dem internen Bus IB verbunden, damit er bei Sprungbefehlen mit einem neuen Wert geladen werden kann. Sein Ausgang ist mit dem Adreßselektor verbunden, der aus vier 74ALS241 besteht und entweder die am VME-Bus anliegende oder die vom Zähler generierte Adresse mit den Adreßleitungen des RAMs verbindet.

Ein Instruktionsregister (IR) speichert den aktuell bearbeiteten Befehl, damit er der Logik während des gesamten Befehlszyklus zur Verfügung steht. Dieses Register wurde (auch zur Einsparung von I/O-Pins) in die beiden CPLDs verlegt, da wegen des Statusregisters CSR der interne Bus IB in jedem Falle in ganzer Breite an sie herangeführt werden muß (siehe Kapitel 3.3). Da die Ausgänge der CPLDs nicht den internen Bus treiben können, ist ein Treiberbaustein zwischengeschaltet.

Bei jedem Datentransfer über den internen Bus IB können zwei Datenbits in den Testregistern TH und TL gespeichert werden. In ähnlicher Weise wird bei jedem VME-Bus-Transfer der Wert der $\overline{\text{BERR}}$-Leitung, die einen Zugriffsfehler anzeigt, gespeichert. Bei bedingten Sprungbefehlen können diese drei Werte und eventuell aufgetretene ABORT-Signale geprüft und dadurch der Programmablauf beeinflußt werden.

3.2.7 Das *event removal* und der Ereignisnummernzähler

Im Bereich der Triggerdetektoren lohnt sich das Entfernen von Ereignissen aus der Pipeline (*event removal*) aufgrund der geringen Datenmenge nicht. Es müßten dafür nämlich anstatt des FIFOs mehrere unabhängige Speicherchips vorgesehen werden, die jeweils ein Ereignis halten und zum erneuten Beschreiben freigegeben werden, wenn das in ihnen enthaltene Ereignis vom Trigger verworfen wurde. Stattdessen können die Daten beim Auslesen übersprungen werden. Dies wird nochmals im Kapitel 4 besprochen.

Der Ereignisnummernzähler (ENC, siehe Seite 10), der es ermöglicht, beim *event removal* aufgetretene Fehler beim Zusammenfassen der Daten in den CPUs zu erkennen, besteht aus zwei 74F579 und hat eine Breite von 16 Bit. Er kann durch das abgearbeitete Programm oder über VME geschrieben und gelesen werden.

3.3 Die Programmierung der CPLDs

Bei der Programmierung wurde zunächst versucht, einzelne eigenständige Blöcke zu identifizieren und festzulegen. Diese wurden zunächst getrennt programmiert und getestet und später zu einem Ganzen zusammengefügt und gemeinsam getestet. Diese Vorgehensweise wird im Fachjargon als hierarchisches Design bezeichnet. Die hier gewählten Blöcke, von denen einige bereits in Kapitel 3.2 genannt wurden, sollen im folgenden vorgestellt werden (siehe dazu Abb. 6).

Der Adreßdekoder (AD) stellt anhand der VME-Bus Adreßleitungen A01, A02 und A15, sowie der Ausgänge der beiden Adresskomparatoren fest, ob eine Adresse anliegt, die einen VME-Zugriff auf den RAM-Bereich, das FIFO, den Ereignisnummernzähler ENC oder das Statusregister CSR anzeigt. Hierbei werden auch die momentanen Werte der Adreßmodifizierer AM und der Zustand des Moduls berücksichtigt. Beispielsweise sind Zugriffe auf FIFO und RAM nur dann zulässig, wenn sich das Modul im `SLAVE`-Modus befindet (vgl. dazu Seite 15). Die Belegung des vom Modul abgedeckten VME-Bus

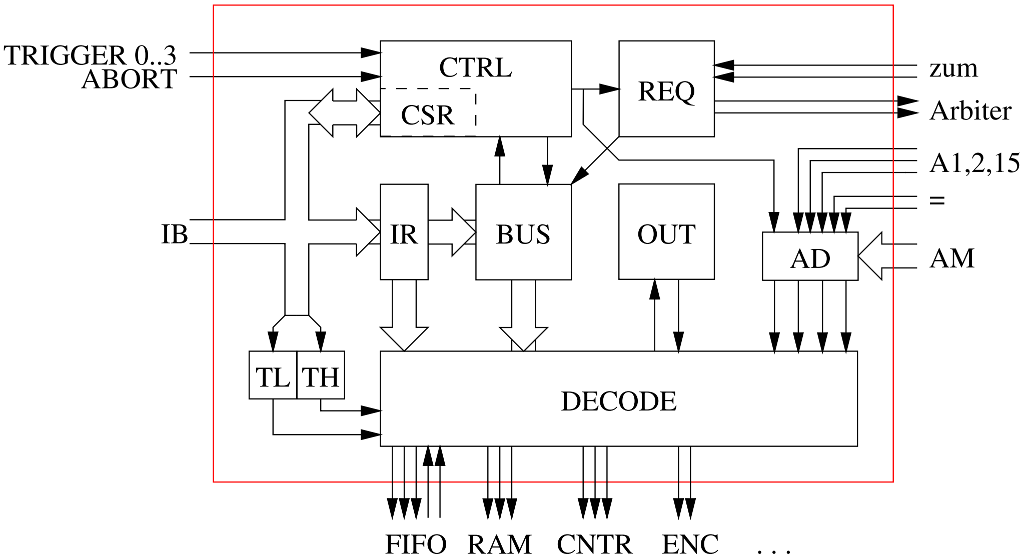


Abbildung 6: Schematische Darstellung der programmierbaren Logik

Adreßbereichs von 64 kByte und die erkannten Adreßmodifizierer (AM) sind in Tabelle 1 angegeben.

Drei zusätzliche IB-Quellen und -Senken sind innerhalb der programmierbaren Logik realisiert. Dies sind die Testregister TL und TH, das Instruktionsregister IR und das Kontroll- und Statusregister CSR. Die zwei Test-Register ermöglichen es, bestimmte Bits aus den übertragenen Daten zu speichern. Bei einem Sprungbefehl kann dann ihr Wert geprüft werden. Das Instruktionsregister IR speichert den gerade auszuführenden Befehl, und das Statusregister dient der Steuerung des Moduls über den VME-Bus. Seine Funktion geht allerdings voll im Block CTRL auf.

Der Requester (REQ) ist ein Logikblock, der bei Bedarf den VME-Bus bei dem dafür zuständigen VME-Bus-Modul (dem sogenannten Arbiter) anfordert und der restlichen Logik meldet, wann er benutzt werden darf. Da es jederzeit möglich sein soll, auf das Statusregister CSR zuzugreifen und so ein eventuell laufendes Programm abzubrechen, muß entweder der Requester den VME-Bus für jeden Zugriff neu anfordern und hinterher freigeben (`RWD`), oder ihn halten solange das Programm läuft aber sofort freigeben, sobald ein anderes Modul den Bus anfordert (`ROR`[5]). Um nicht bei jedem VME-Zugriff eventuell noch auf den Requester warten zu müssen, wurde hier die zweite Variante gewählt.

Die Ausgangssteuerung, die in Abbildung 6 mit OUT bezeichnet ist, sorgt dafür, daß

[4]Zahlen, die mit `0x` beginnen sind Hexadezimalzahhlen, ein großes `X` soll andeuten, daß es entweder gleichgültig ist, welche Ziffer an dieser Stelle steht oder daß nicht bekannt ist, was dort zu stehen hat. Beispielsweise ist `0x11` gleich siebzehn und `0x1X` bezeichnet Zahlen zwischen 16 und 31. Im Falle von Tabelle 1 steht `XXXX` für die Basisadresse des Moduls.

[5]Eine genauere Erklärung dieser Begriffe findet sich in [9] und [10].

Adresse [4]	Bedeutung	
	`Slave`	`Master`
`0xXXXX0000` ⋮ `0xXXXX7FFF`	RAM	—
`0xXXXX8000`	FIFO	—
`0xXXXX8002`	ENC	ENC
`0xXXXX8004`	CSR	CSR
`0xXXXX8006`	CSR	CSR
⋮	⋮	⋮
`0xXXXXFFF8`	FIFO	—
`0xXXXXFFFA`	ENC	ENC
`0xXXXXFFFC`	CSR	CSR
`0xXXXXFFFE`	CSR	CSR

Als Adreß-modifizierer sind zulässig:	
0x09	0x0A
0x0D	0x0E
0x29	0x2E
0x39	0x3A
0x3D	0x3E

Tabelle 1: Adreßbelegung des DACOX-Moduls

der Inhalt des FIFOs über den TPB weitergegeben wird, wenn das Modul am anderen Ende zum Empfang bereit ist. Ist DACOX jedoch im `SLAVE`-Modus, hat der VME-Bus die Kontrolle über die Ausgangsseite des FIFOs.

Es gibt 5 verschiedene Arten von Buszyklen. Dies sind:

FETCH-Zyklen: wenn ein neuer Befehl vom RAM ins Instruktionsregister (IR) geladen wird,

XFER-Zyklen: wenn aufgrund einer entsprechenden Instruktion Daten transferiert werden, etwa vom RAM ins FIFO,

JMP-Zyklen: wenn aufgrund einer entsprechenden Instruktion der Befehlszähler (CNTR) neu geladen wird, oder

MVME-Zyklen: wenn aufgrund einer entsprechenden Instruktion selbständig ein VME-Zyklus initiiert wird.

SVME-Zyklen: wenn über VME auf RAM, FIFO, ENC oder CSR zugegriffen wird,

Die ersten vier treten bei der Abarbeitung des geladenen Programms auf, SVME-Zyklen müssen aber jederzeit möglich sein und haben stets Priorität. Deshalb wird folgendermaßen vorgegangen:

Ein Logikblock (CTRL) gibt zusammen mit dem aktuellen Inhalt des Instruktionsregisters IR an, was für ein Zyklus als nächstes ausgeführt werden soll. Ein zweiter Logikblock

(BUS) führt ihn aus und meldet die erfolgreiche Ausführung zurück. Wird aber gleichzeitig von außen auf das Modul zugegriffen, stellt BUS den angeforderten Befehl zurück und führt zuerst diesen Zugriff aus.

Der CTRL-Block ist darüber hinaus auch für die Zustandssteuerung des gesamten Moduls zuständig. Er legt fest, ob sich das Modul im `MASTER`- oder `SLAVE`-Modus befindet und ob gerade auf Trigger gewartet wird. Er reagiert auf die vier Trigger-Eingänge und Statusregisterzugriffe, die das Modul auf Trigger vorbereiten oder Trigger auslösen. Die Bedeutung der einzelnen Bits des Statusregisters CSR bei Lese- und Schreibzugriffen ist im Abschnitt 3.6 angegeben.

Die Steuersignale für alle auf dem Modul vorhandenen Bauteile können aus dem Zustand der Blöcke AD, OUT, BUS, IR und CTRL erzeugt werden[6]. Dies geschieht im Block DECODE. Welche Bedeutung dabei die einzelnen Bits des Instruktionsregisters IR haben, wird ausführlich im Kapitel 3.4 besprochen.

3.4 Die Programmierung des Moduls

Im folgenden Abschnitt wird auf die verschiedenen Instruktionen eingegangen, die vom DACOX-Board ausgeführt werden können. Dabei werden jeweils die im RAM zu speichernden Binärcodes und die dadurch hervorgerufene Aktion vorgestellt. Nur der Vollständigkeit halber werden auch stets die mnemonischen Kürzel genannt, die bei der Programmierung mit dem im Zuge dieser Diplomarbeit geschaffenen DACOX-Assembler (siehe Abschnitt 3.5) verwendet werden. Die Programmierung mit dem DACOX-Assembler wird genauer im Anhang C erläutert.

Die Instruktionen lassen sich in vier Kategorien einteilen: interne Datentransfers (*IB transfers*), bedingte und unbedingte Sprungbefehle (*jumps*), Datentransfers auf dem VME-Bus (*VME transfers*) und den `STOP`-Befehl.

3.4.1 Interne Datentransfers (*IB transfers*, `MOV`)

Die Binärcodes `0b 10XX XXXX XXXX XXXX`[7] veranlassen das Modul, einen internen Transfer durchzuführen. Im Assembler wird ein solcher Zyklus durch das Schlüsselwort `MOV` eingeleitet.

Das Ziel des Transfers wird in den Bits 0 bis 2 des Binärcodes festgelegt. Mögliche

[6] Beispielsweise müssen die auf dem Bus liegenden Daten in den ENC gespeichert werden, wenn erstens BUS gerade einen VME Schreibzugriff abarbeitet und der Adreßdekoder (AD) die Adresse des ENC erkannt hat, oder wenn zweitens BUS gerade einen internen Transfer abarbeitet und das Instruktionsregister IR anzeigt, daß Daten in den ENC geschrieben werden sollen.

[7] Zahlen, die mit `0b` beginnen, sind Binärzahlen; ein großes `X` soll andeuten, daß es entweder gleichgültig ist, welche Ziffer an dieser Stelle steht oder daß nicht bekannt ist, was dort zu stehen hat. Beispielsweise ist `0b11` gleich drei und `0bXXX0` bezeichnet gerade Zahlen zwischen 0 und 15.

Werte sind:

Bitkombination	Ziel	Mnemonic
`0b000`	D00-15	`DL`
`0b001`	D16-31	`DH`
`0b010`	Ereignisnummernzähler	`ENC`
`0b011`	FIFO	`FIFO`
`0b101`	Nur Testbits speichern	—
`0b110`	A01-15	`AL`
`0b111`	A16-31	`AH`

Die Datenquelle für den Transfer wird in Bit 3 bis 5 angegeben. Mögliche Werte sind hier:

Bitkombination	Ziel	Mnemonic
`0b000`	D00-15	`DL`
`0b001`	D16-31	`DH`
`0b010`	Ereignisnummernzähler	`ENC`
`0b100`	Programmspeicher	`RAM`

Das FIFO kann hier nicht als Quelle dienen, da seine Ausgabeseite während der Programmausführung unter der Kontrolle der OUT-Logik steht (vgl. 3.3). Die Ausgangsregister für die Adressen (AL und AH) können nicht zurückgelesen werden, und die Adreßmodifizierer (AM) werden erst beim Aufruf des VME-Transfers geladen (siehe unten).

Bei Transfers zum FIFO können noch die beiden Zusatz-Bits EOE und EOL mit angegeben werden. Sie sind als Bit 7 (EOE) und Bit 6 (EOL) im Binärcode gespeichert. Wird im Assembler zusätzlich das Schlüsselwort `EOE` oder `EOL` angegeben, so wird das entsprechende Bit gesetzt, andernfalls bleibt es gelöscht.

Bei jedem internen Datentransfer werden die beiden Test-Register TH und TL geladen. Die Bits 8 bis 10 des Binärcodes geben an, welches der Bits 0 bis 7 des Datenwortes im Flag TL gespeichert wird, während die Bits 11 bis 13 angeben, welches der Bits 8 bis 15 in TH übernommen wird. Im Assembler gibt man einfach die Bitnummern an, sie werden dann entsprechend in TH oder TL geladen.

3.4.2 Bedingte und unbedingte Sprünge (*jumps*, `JMP`)

Die Binärcodes `0b 01XX XXXX XXXX XXXX` veranlassen das Modul, die Ausführung des Programms an einer anderen Stelle im Programmspeicher fortzusetzen, wenn gewisse Bedingungen erfüllt sind. Im Assembler wird ein solcher Zyklus durch das Schlüsselwort `JMP` eingeleitet.

Die Adresse, zu der gesprungen werden soll, wird in der nächsten Programmspeicherstelle abgelegt.[8] Im Assembler gibt man einfach die entsprechende Adresse oder ein Label an (Genaueres über Labels in Abschnitt 3.5 und Anhang C).

[8] Es ist zu beachten, daß es sich dabei um eine Wort-Adresse handelt, sie ist also halb so groß wie die Adresse, unter der sie vom VME-Bus aus angesprochen würde.

Es können verschiedene Bedingungen geprüft werden, bevor der Sprung ausgeführt wird:

Bit	Daten	Mnemonic	Bedeutung
2	1,0	`PRGM`	aktuell bearbeitetes Programm
3	–	`BERR`	Buserror beim letzten VME Zugriff
5	4	`TL`	TESTL hat den angegebenen Wert
7	6	`TH`	TESTH hat den angegebenen Wert
8	–	`ABOR`	ABORT ist aktiv

Ist Bit 2 des Binärcodes gesetzt, so wird geprüft, ob die Nummer des aktuell bearbeiteten Programm mit der in den Bits 0 und 1 angegebenen Zahl übereinstimmt. Ist Bit 3 gesetzt, so wird nur verzweigt, wenn beim letzten VME-Zugriff ein *bus error* aufgetreten ist. Ist Bit 5 gesetzt, so wird Bit 4 des Binärcodes mit dem momentan im Testregister TESTL gespeicherten Wert verglichen, ähnliches gilt für die Bits 7 und 6 und das Testregister TESTH. Ist schliesslich Bit 8 des Binärcodes gesetzt, so wird nur verzweigt, wenn im Laufe der Programmausführung ein ABORT-Signal auftrat. Sind mehrere der Bits 2, 3, 5, 7 und 8 gesetzt, so müssen alle zugehörigen Bedingungen erfüllt sein, um eine Verzweigung zu bewirken. Im Assembler werden die Mnemonics für die zu prüfenden Bedingungen (eventuell mit dem zugehörigen Zahlenwert für die Programmnummer oder den Wert des Testbits) nach der Sprungadresse angegeben.

3.4.3 VME Transfers (*VME transfers*, `VMER`/`VMEW`)

Die Binärcodes `0b 11XX XXXX XXXX XXXX` veranlassen das Modul, Daten über den VME-Bus zu lesen oder zu schreiben. Im Assembler wird ein solcher Zyklus durch die Schlüsselwörter `VMER` und `VMEW` eingeleitet.

Im Binärcode bedeutet ein gesetztes Bit 3, daß es sich um einen Schreibzugriff handelt, ein gelöschtes Bit signalisiert einen Lesezugriff.

Das VME-Signal LWORD* wird in Bit 2 des Binärcodes angegeben, DS0 und DS1 in Bit 6 und 7, und die Adreßmodifizierer AM belegen die Bits 8 bis 13. Dabei ist zu beachten, daß DS0 und DS1 wie WRITE '*active-high*' angegeben werden, die tatsächlichen Pegel auf dem VME-Bus sind also genau anders herum. LWORD* und die Adreßmodifizierer AM werden dagegen genau so angegeben, wie sie nachher erscheinen.

Im Assembler werden diese potentiellen Fehlerquellen vermieden, indem stattdessen die üblichen VME-Mnemonics `D80`, `D8E`, `D16`, `D32` und der Wert der Adreßmodifizierer (AM) angegeben werden. Das Programm ermittelt daraus dann den richtigen Binärcode.

Bei Schreibzyklen werden die Daten geschrieben, die zuvor in den Datenbustreibern abgelegt wurden (etwa mit dem Befehl '`MOV RAM DL;`'), bei Lesezyklen werden sie dorthin gespeichert und können (etwa mit '`MOV DL FIFO;`') von dort gelesen werden.

3.4.4 Beendigung des Programms (*stop*, `STOP`)

Die Binärcodes `0b 00XX XXXX XXXX XXXX` veranlassen das Modul, die Ausführung des Programms zu beenden. Im Assembler wird ein solcher Zyklus durch das Schlüsselwort `STOP` eingeleitet. Das Modul geht in den Wartezustand, und das BUSY-Signal wird inaktiv.

3.5 Der DACOX-Assembler

Die Programmierung des Moduls wird deutlich einfacher, wenn der Programmierer anstatt der komplizierten Binärcodes einfache, aussagekräftige Kürzel (sogenannte mnemonics) eingibt, die das auszuführende Programm für ihn verständlich beschreiben. Das in diesem Abschnitt vorgestellte Programm, der DACOX-Assembler, erzeugt daraus die maschinenlesbaren Binärcodes und umgeht so die fehlerträchtige Kodierung 'von Hand'.

Bei der manuellen Programmierung müßten auch alle Sprungbefehle nochmals auf Korrektheit geprüft werden, sobald irgendwo ein Befehl eingefügt wird, da sich durch den zusätzlichen Befehl eventuell die absoluten Adressen der Sprungziele ändern. Beim DACOX-Assembler kann der Programmierer die Sprungziele mit Namen (sog. Labels) versehen und bei Sprungbefehlen dieses Label anstatt der Zieladresse angeben. Die Speicherstelle, zu der dann tatsächlich gesprungen wird, wird automatisch festgestellt und braucht dem Programmierer nicht einmal bekannt zu sein.

Der DACOX-Assembler erlaubt die Verwendung von Konstanten und Variablen, was die Verwendung von aussagekräftigen Namen (z.B. `TDC_BASE` und `TDC_CSR` anstelle von `0x110000` und `0x118000`) ermöglicht. Dadurch wird der Quelltext übersichtlicher und auch flexibler, da so zum Beispiel Basisadressen von Modulen durch Änderung einer einzigen Zeile im gesamtem Programm geändert werden können.

Damit mit der Änderung der Basisadresse auch alle anderen von ihr abhängenden Adressen angepaßt werden können, kann der Programmierer deren Wert durch einen arithmetischen Ausdruck (*expression*) festlegen, der die Basisadresse enthält. Der DACOX-Assembler berechnet dann automatisch den aktuellen Wert.

Neben den Konstanten kennt der DACOX- Assembler auch Variablen, deren Wert sich innerhalb der Quelldatei ändern kann. Im Gegensatz dazu haben Konstanten überall denselben Wert (selbst an Stellen, die im Quelltext vor ihrer Definition stehen).

Die vom DACOX-Assembler verstandenen Instruktionen entsprechen genau denen, die im Kapitel über die Hardware genannt sind (siehe 3.4), nämlich den internen Datentransfers (`MOV`), den bedingten und unbedingten Sprüngen (`JMP`), VME-Zugriffen (`VMER`, `VMEW`) und dem `STOP`-Befehl, der die Programmausführung beendet. Sie werden ergänzt durch Anweisungen an den Assembler (*Compiler commands*), die Variablen (`VAR`) und Konstanten (`CONST`) zuweisen, Labels definieren oder den Speicherzeiger beeinflussen. (Der Speicherzeiger gibt stets an, an welche Speicheradresse die nächste Instruktion geschrieben wird.)

Der DACOX-Assembler erzeugt eine einzige Ausgabedatei, die Binärcode, Quelltext und Fehlermeldungen in einem enthält. Dies hat mehrere Vorteile:

- Da die Fehlermeldungen direkt im Quelltext stehen, ist immer klar, in welcher Zeile der Fehler aufgetreten ist.

- Wurde der Quellcode durch einen Präprozessor vorverarbeitet, so steht in der Ausgabedatei der präprozessierte Quelltext. Dies erleichtert die Fehlersuche von Präprozessorkommandos.

- Da der Binärcode direkt beim Quelltext steht, kann auf der Suche nach Hardwarefehlern leicht die richtige Stelle im Quelltext gefunden werden. Dadurch weiß man sofort, was der momentane Befehl tut und braucht nicht erst den Binärcode wieder zurück zu übersetzen. Da auch die Kommentare wiedergegeben sind erhält man sogar noch zusätzliche Informationen, die im Binärcode nicht enthalten sind.

Außerdem ist es mit diesem Ausdruck leichter, den Assembler selbst auf Fehler zu untersuchen.

Um die Benutzung des Moduls auch für Nicht-Experten zugänglich zu machen, wurde zum Assembler ein 14-seitiges Handbuch (in englischer Sprache) verfaßt, das dieser Diplomarbeit als Anhang C beigefügt ist. Es gibt zunächst eine sehr knappe Einführung in die Hardware des Moduls, die aber zur Programmierung ausreichen sollte. Dann wird der Aufruf des Programms erläutert und die verwendete Sprache erklärt. Es schließt mit einigen Beispielen, die die Benutzung der Konstanten und Variablen demonstrieren.

Wie schon bei der Programmierung der Hardware wurde zur Programmierung des Assemblers zunächst versucht, das Problem in kleinere, einfache Module zu zerlegen. Diese Module sollen im folgenden besprochen werden.

3.5.1 `fileio`

Das Modul `fileio` erledigt alles, was mit Ein- und Ausgabe zu tun hat. Nur über die Funktionen aus diesem Modul wird der Quelltext gelesen, oder Binärcode, Fehlermeldungen, Quelltext und Informationen zur Fehlersuche ausgegeben. Durch Änderungen an diesem Modul konnte beispielsweise nachträglich der *m4*-Präprozessor eingebaut werden, ohne daß die anderen Module dafür geändert werden müßten. Ebenso könnte die Ausgabe umgestaltet oder auf mehrere Dateien verteilt werden.

3.5.2 `node`

Im Assembler wird der Quelltext zunächst in sogenannte Knoten (*nodes*) zerlegt. In einem Knoten kann eine Zahl (z.B. `12`, `0xFF`), ein Operator (`+`, `*`, `)`), ein Schlüsselwort (`MOV`, `CONST`), ein Name (`i`, `BASE_ADDR`) oder ein Semikolon (`;`)[9] gespeichert werden. Die Bezeichnung Knoten hängt mit seiner Verwendung bei arithmetischen Ausdrücken zusammen, auf die später eingegangen wird. Dort werden auch noch einige weitere Eigenschaften von *node*s beschrieben, die hier der Einfachheit halber weggelassen wurden. Da *node*s in fast allen Modulen benutzt werden, wurde die Definition dieses Datentyps zusammen mit einigen nützlichen Funktionen zur Erzeugung, Vernichtung und Ausgabe von *node*s in ein eigenes Modul, `node`, verlegt

[9] Alle Befehle werden durch ein Semikolon abgeschlossen.

3.5.3 `parse`

Das Modul `parse` nimmt die oben erwähnte Zerlegung der Eingabedatei vor. Dabei werden Kommentare und Leerzeichen im Quelltext ignoriert und der Rest in *nodes* umgewandelt. Die weiter unten beschriebenen Module holen sich den Quelltext dann 'knotenweise' vom Modul `parse` ab. Zusätzlich besteht die Möglichkeit, bereits abgeholte *node*s wieder zurückzugeben. Werden dann wieder Knoten angefordert, so werden erst die zurückgegebenen Knoten weitergegeben und dann neue aus der Eingabedatei geholt.

Wie schon im letzten Abschnitt angedeutet, können *node*s in verschiedene Klassen eingeteilt werden: Zahlen (*number*), Operatoren (*operator*), Namen (*identifier*), Schlüsselwörter (*keyword*) und Semikolon. Die genau in die einzelnen Klassen fällt wird im Anhang C beschrieben.

3.5.4 `variable`

Das Modul `variable` ermöglicht es, Variablen abzuspeichern. Dies geschieht mit Hilfe von `VarTables` – Variablen-Tabellen. Ein Eintrag in einer solchen Tabelle besteht aus einem Variablennamen und den dazugehörigen Daten. Hat man zuvor gewisse Daten unter einem bestimmten Namen in einer `VarTable` abgespeichert (indem man den Namen und die Daten der entsprechenden Funktion übergeben hat), so ermöglicht eine andere Funktion den Zugriff auf die Daten, wenn man ihr den entsprechenden Namen übergibt.

Der DACOX-Assembler benutzt zwei solche `VarTables`, eine für die Konstanten und Labels, und eine andere für die Variablen. Dieses Modul stellt dazu Funktionen zum Erzeugen und Vernichten von `VarTables`, zum Speichern und Lesen von Daten, und für jede `VarTable` einen sogenannten Iterator zur Verfügung. Dieser ermöglicht es, die gesamte `VarTable` einmal durchzugehen.

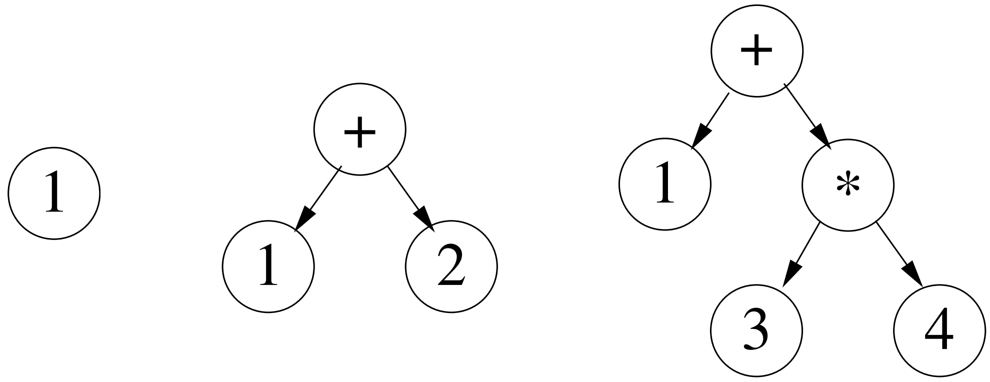
3.5.5 `aritex`

Das Modul `aritex` enthält eine Routine zum Einlesen von arithmetischen Bäumen und eine weitere zur Vereinfachung (Berechnung) solcher Bäume.

Diese beiden Funktionen ermöglichen es dem Assembler, Ausdrücke wie `12 + 5 * ( 1 << 4 )` unter Beachtung der Rangfolge der Operatoren einzulesen und zu berechnen. Enthält der Ausdruck auch Variablen oder Konstanten (etwa wie in `12 * a + 1`), so kann der Vereinfachungsalgorithmus deren Werte einsetzen, falls es gewünscht wird, und zwar bis zu einer gewissen Schachtelungstiefe. Diese Beschränkung ist nötig, da sonst eine rekursive Konstantendefinition wie $\mathtt{A} = \mathtt{A} + 1$ eine Endlosschleife hervorrufen würde.

Hier jetzt Genaueres über Knoten und deren besondere Eigenschaften: Jedem Knoten sind bis zu zwei Tochterknoten zugeordnet. Dadurch kann man Knoten zu sogenannten Bäumen verknüpfen, die benutzt werden, um arithmetische Ausdrücke zu speichern.

Der einfachste Ausdruck (eine einzelne Zahl) besteht lediglich aus einem einzigen Knoten, einem Zahlknoten. Ein etwas komplizierterer Ausdruck ist zum Beispiel `1 + 2`, der aus einem $\oplus$-Operatorknoten mit zwei Tochterknoten ① und ② besteht. Richtig interessant wird die Sache erst bei mehreren Ebenen, etwa `1 + 3 * 4`, was aus dem letzten



(a) `1` (b) `1 + 2` (c) `1 + 3 * 4`

Abbildung 7: Baumdarstellung arithmetischer Ausdrücke

Beispiel dadurch hervorgeht, daß der zweite Tochterknoten des +-Knotens nicht eine 2 ist, sondern ein *-Operatorknoten, der selbst zwei Tochterknoten (3 und 4) hat.

Die Routine zum Einlesen von Ausdrücken muß nun immer einen neuen Knoten vom Modul `parse` abholen und ihn an der richtigen Stelle im bisher gelesenen Baum einbauen. Dabei kommt natürlich die Rangfolge der Operatoren ins Spiel. Läßt sich der Knoten nicht einbauen, so ist der Ausdruck zuende und die Routine gibt den fertigen Baum zurück.

Die Routine zum Vereinfachen von Bäumen versucht, jeden Operatorknoten durch einen Zahlknoten zu ersetzen, indem sie zunächst die Tochterknoten zu Zahlen macht und dann deren Summe, deren Produkt, oder was auch immer der jeweilige Operator bedeutet, berechnet. Diese Zahl ersetzt dann den Operatorknoten. Wenn alles klappt, bleibt am Ende nur ein einziger (Zahl-)Knoten übrig.

Genauere Informationen über die implementierten Operatoren und deren Rangfolge finden sich im Anhang C, wo die Benutzung des Assemblers erläutert wird.[10]

3.5.6 pass_one

Das Modul `pass_one` geht den Quellcode ein erstes Mal durch und liest alle Labels und die Definitionen aller Konstanten ein. Am Ende werden die Werte aller Konstanten berechnet.

Um die Labels richtig zuweisen zu können, muss `pass_one` immer wissen, an welche Speicherstelle die nächste Instruktion geschrieben wird. Deshalb bei jeder Instruktion festgestellt, wieviel Speicher sie belegt und der Speicherzeiger entsprechend erhöht. Tritt im Quelltext eine Labeldefinition auf, so wird der Speicherzeiger als Wert des Labels gespeichert. Bei Konstantendefinitionen wird zunächst der zugehörige Ausdruck gespeichert, der

[10] Normalerweise geht die Berechnung von arithmetischen Ausdrücken deutlich schneller, wenn der Baum zuvor in eine UPN-Liste gewandelt wird. Da hier aber die meisten Bäume nur einmal abgearbeitet werden, bringt die Wandlung keinen Geschwindigkeitsvorteil. Deshalb wurde von einer Wandlung in eine UPN-Liste abgesehen.

dann am Ende berechnet wird, wenn alle Konstanten definiert und alle Labels bekannt sind.

3.5.7 `pass_two`

Das Modul `pass_two` führt die eigentliche Assemblierung durch. Es geht den Quelltext ein zweites Mal durch und wandelt jede Instruktion in den zugehörigen Binärcode um.

Die Eingabedatei wird knotenweise eingelesen, und jede Instruktion für sich bearbeitet: Bei Variablendefinitionen wird die entsprechende Variable angelegt, bei Instruktionen werden die Daten zusammengestellt, die später in das DACOX-Modul geladen werden sollen. Nach jeder Instruktion wird der Speicherzeiger erhöht. Konstantendefinitionen werden ignoriert, da deren Wert bereits feststeht. Aus diesem Grund kann auch das Modul `aritex` alle Ausdrücke berechnen, sofern der Programmierer keine undefinierten Konstanten verwendet und keine rekursiven Definitionen gemacht hat.

3.6 Weitere Software

Neben dem Assembler wurde weitere Software geschaffen, die einen Test der Hardware erlaubt und die Programmierung und Installation des Moduls vereinfacht. Dies sind eine Unterprogrammbibliothek, die zugehörige Kommandozeilen-Schnittstelle und ein graphisch orientiertes Kontroll-Panel, das eine einfache und intuitive Benutzung ermöglicht.

3.6.1 Die Unterprogrammbibliothek

Die Unterprogrammbibliothek ist in C geschrieben und enthält Routinen zur Initialisierung, zum Test und zur Steuerung des Moduls. Eine Übersicht über die einzelnen Funktionen ist in Abbildung 2 gegeben. Es soll hier kurz erläutert werden, wozu die Funktionen im einzelnen dienen. Für weitergehende Fragen muß leider auf die Quelltexte verwiesen werden, die jedoch ausführlich kommentiert sind. Auch die Quelltexte der Kommandozeilenschnittstelle, die im nächsten Abschnitt vorgestellt wird, können als Beispiel herangezogen werden.

Die beiden Funktionen `dcxopen()` und `dcxclose()` funktionieren analog zu den Funktionen `fopen()` und `fclose()` aus den Standard-Bibliotheken. `dcxopen()` liest alle das Modul betreffenden Informationen aus einer Konfigurationsdatei ein, speichert sie in einer speziellen Datenstruktur ab und übergibt diese an das aufrufende Programm. Beim Aufruf der übrigen Routinen wird sie mit übergeben, sodaß die Information allen zur Verfügung steht. Die Funktion `dcxclose()` gibt den von `dcxcopen()` reservierten Speicher wieder frei.

Die drei Funktionen `dcx_test_RAM()`, `dcx_test_FIFO()` und `dcx_test_ENC()` testen den ENC, das RAM und das FIFO und stellen deren Größe fest. (Diese wird auch in der obigen Struktur abgespeichert.) Jeder dieser Tests erfordert nur die Funktionstüchtigkeit der jeweiligen Komponente.

Im Gegensatz dazu testet die Funktion `dcx_test_all()` das Zusammenspiel aller Komponenten. Dieser Test gliedert sich in 3 Teile:

Funktion	Funktion
`DACOX *dcxopen(char *scname, char *cname);` `int dcxclose(DACOX *dcx);`	analog zu fopen() und fclose()

`int dcx_test_RAM(DACOX *dcx);`
`int dcx_test_FIFO(DACOX *dcx);`
`int dcx_test_ENC(DACOX *dcx);`
	Low-Level Tests

`int dcx_test_all(DACOX *dcx);`	Gesamt-Test
`int dcx_get_sreg(DACOX *dcx);` `int dcx_set_sreg(DACOX *dcx,int val);`	Statusregister lesen Statusregister schreiben

`int dcx_reset(DACOX *dcx);`
`int dcx_go_master(DACOX *dcx);`
`int dcx_go_slave(DACOX *dcx);`
`int dcx_trigger(DACOX *dcx,int trigno);`
`int dcx_retrans(DACOX *dcx);`
`int dcx_wait_prgm_stop(DACOX *dcx);`
	zurücksetzen
	auf Trigger vorbereiten
	
	Auslese starten
	FIFO 'zurückspulen'
	Ende abwarten

`int dcx_is_master(DCX);`
`int dcx_is_busy(DCX);`
`int dcx_is_slave(DCX);`
`int dcx_is_running(DCX);`
	Diese Makros
	überprüfen den
	Status des Moduls

`int dcx_download(DACOX *dcx,char *fn);`	Programm laden

Tabelle 2: Die Funktionen der Unterprogrammbibliothek

1. **jump test:** Test des Programmzählers CNTR und der beiden Test-Register durch Ausführen mehrerer bedingter und unbedingter Sprungbefehle.

2. **`D16` test:** Test von VME `D16` Lese- und Schreibzugriffen

3. **`D32` test:** Test von VME `D32` Lese- und Schreibzugriffen

Die letzten beiden Tests erfordern auch die fehlerfreie Übertragung vom und zum ENC und in das FIFO.

Die folgenden Funktionen dienen vor allem der Lesbarkeit und Portabilität des Quelltextes. Es wird dringend empfohlen, alle Programme so zu gestalten, daß sie das DACOX-Modul ausschließlich über diese Unterprogramme ansprechen. So müssen bei einer Veränderung des Moduls nur die Bibliotheksfunktionen umgearbeitet werden.

Die Funktionen **`dcx_get_sreg`**() und **`dcx_set_sreg`**() greifen auf das Statusregister zu. Die Bedeutung der einzelnen Bits ist in Abbildung 8 erläutert. Es sei aber betont, daß der Programmierer zur Steuerung des Moduls wenn möglich stattdessen lieber eine der folgenden Funktionen benutzen sollte:

`dcx_reset()` versetzt das Modul in den `SLAVE`-Modus[11]. Das FIFO ist leer, es wird nicht auf Trigger reagiert.

[11]Zu den verschiedenen Modi s. Seite 15

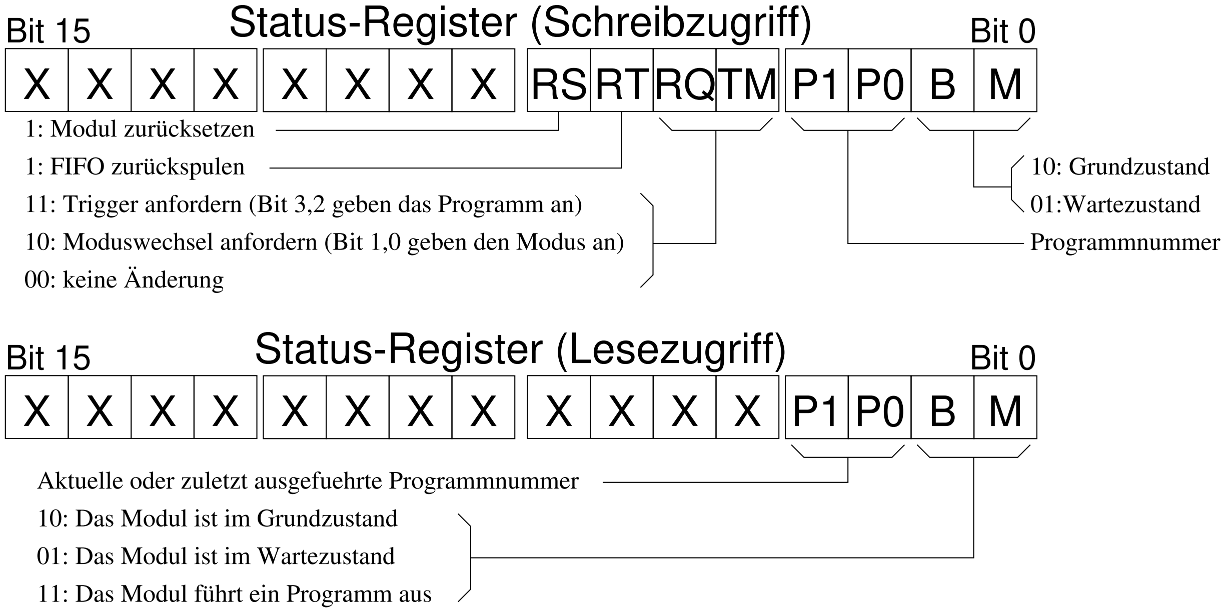


Abbildung 8: Die Belegung des Kontroll- und Statusregisters CSR

`dcx_go_master()` versetzt das Modul in den Wartezustand. Ankommende Trigger veranlassen die Abarbeitung des geladenen Programms.

`dcx_go_slave()` versetzt das Modul in den Ausgangszustand, ohne jedoch das FIFO zu leeren. Auf Trigger wird nicht reagiert.

`dcx_trigger()` startet die Abarbeitung des geladenen Programms, wenn sich das Modul zuvor im Wartezustand befunden hat.

`dcx_retrans()` setzt den Lesezeiger des FIFOs zurück, sodaß die Daten ein zweites Mal ausgelesen werden können.

`dcx_wait_prgm_stop()` wartet, bis das laufende Programm beendet und das Modul wieder im Wartezustand ist.

`dcx_is_master()` liefert TRUE, wenn das Modul gerade im Wartezustand ist oder ein Programm ausführt.

`dcx_is_busy()` liefert TRUE, wenn das Modul gerade nicht im Wartezustand ist und demzufolge nicht auf Trigger reagiert.

`dcx_is_slave()` liefert TRUE, wenn das Modul gerade im Grundzustand ist.

`dcx_is_running()` liefert TRUE, wenn das Modul gerade ein Programm abarbeitet.

Die letzte Funktion, `dcx_download()`, lädt das angegebene Programm, das zuvor mit dem DACOX-Assembler (siehe Kap. 3.5 und Anhang C) erzeugt wurde, in den RAM-Bereich des Moduls.

3.6.2 Die Kommandozeilen-Schnittstelle und das Kontroll-Panel

Die Kommandozeilen-Schnittstelle erlaubt einen Aufruf der meisten Bibliotheksroutinen von der Kommandozeile aus. Dadurch kann das Modul von einem einfachen Terminal aus gesteuert werden, ohne daß dazu jedesmal ein neues Programm geschrieben und compiliert werden müßte. Den meisten Bibliotheksroutinen ist eine Kommandozeilenoption zugeordnet, durch deren Angabe man sie aufrufen kann. Sind mehrere Optionen angegeben, so werden auch mehrere Routinen in der angegebenen Reihenfolge aufgerufen. Es besteht sogar die Möglichkeit Schleifen, auszuführen, also eine Gruppe von Routinen mehrmals hintereinander ausführen zu lassen.

Um die Sache noch einfacher zu gestalten und die Bedienung des Moduls auch dem Nicht-Experten zugänglich zu machen, wurde ein in Tcl/Tk programmiertes Kontroll-Panel geschaffen, das auf einen Mausklick hin die entsprechenden Kommandos ausführt. Durch Deaktivierung einzelner Knöpfe wird der Benutzer auf die beim jeweiligen Zustand des Moduls sinnvollen Befehle eingeschränkt.

4 Das DARX-Modul

4.1 Vorgaben

Das DARX-Modul soll die vom DACOX-Modul über den TPB gesendeten Daten entgegennehmen und anhand des Paritätsbits auf Fehler prüfen. Dabei soll es in der Lage sein, mehrere Ereignisse gleichzeitig zu halten, wenn die CPUs noch nicht zur Aufnahme von Daten bereit sind (Pipelining). Die Schnittstelle zur Triggerlogik (BUSY- und ABORT-Signal) ist dabei bereits im DACOX-Modul implementiert; die Möglichkeit zum Entfernen von Ereignissen aus der Pipeline (*event removal*, siehe Abschnitt 2.5) muß auf dem DARX-Modul vorgesehen werden.

Ausleseprogramme sollen an einem Statusregister erkennen können, ob Daten zur Auslese anstehen und ob bei der Datenübertragung irgendwelche Fehler aufgetreten sind. Ebenso muß klar sein, wo ein Ereignis zuende ist und das nächste Ereignis beginnt.

Die Schnittstelle zum TPB muß mit der auf dem DACOX-Modul befindlichen OUT-Logik zusammenarbeiten, es müssen also auf einen Datentastimpuls (DS_TPB) hin die Daten vom Bus übernommen werden und ein FULL-Signal erzeugt werden, das angibt, wann das Modul keine Daten mehr aufnehmen kann. Gegebenenfalls kann jedoch auch das DACOX-Modul an die Bedürfnisse von DARX angepaßt und das TPB-Protokoll entsprechend verändert werden.

4.2 Hardware

Das Herzstück des Moduls sind zwei FIFOs vom Typ IDT7206, die 4096 Worte $\times$ 18 Bits speichern können. Die Eingänge sind direkt mit den Datenempfängern des TPB verbunden, die Ausgänge können von den Datenbustreibern auf den VME-Bus gegeben werden (vgl. dazu Abb 9).

Da die Schnittstelle zum Zweitebenen-Trigger noch nicht spezifiziert ist, wurde bei der ersten Version dieses Moduls auf die Implementierung einer Entfernung von Ereignissen aus der Pipeline (*event removal*) verzichtet. Diese ist aber prinzipiell möglich und es sollen zwei Möglichkeiten diskutiert werden:

1. Die erste mögliche Realisierung einer Entfernung von Ereignissen erfordert keine hardwareseitigen Veränderungen des in dieser Arbeit beschriebenen Moduls. Es muß lediglich die Auslesesoftware der CPUs so geschrieben werden, daß das nächste gültige Ereignis durch Vergleich der Ereignisnummern mit den von den anderen Detektoren gelieferten Ereignisdaten erkannt wird.

2. Die zweite mögliche Realisierung sieht vor, daß ein Ereignis erst dann in das FIFO des DARX-Moduls übernommen wird, wenn der Mehrstufen-Trigger es akzeptiert hat. Vom Mehrstufen-Trigger ausgesonderte Ereignisse werden zwar vom DACOX-Modul übertragen, aber nicht in das FIFO des DARX-Moduls eingespeichert.

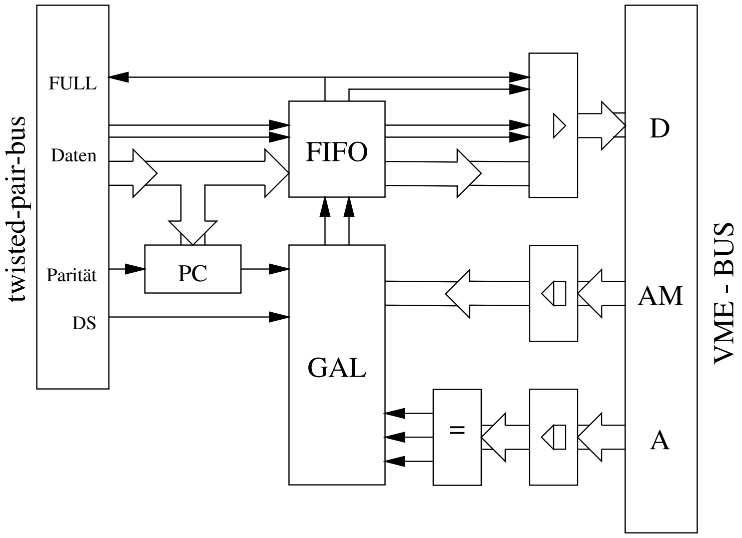


Abbildung 9: Blockdiagramm des DARX-Moduls
Die über den *twisted-pair-bus* (TPB) ankommenden und vom Paritätsprüfer (PC) auf Korrektheit geprüften Daten werden ins FIFO gespeichert, wo sie verbleiben, bis eine der CPUs sie über den VME-Bus ausliest.

Da einerseits die FIFOs nur 18 Bit breit sind, es insgesamt aber 19 Datenbits zu speichern gibt (16 Bit Daten, EOE, EOL und das Paritätsbit), muß auf die Speicherung eines dieser Bits verzichtet werden. Dazu gibt es zwei Möglichkeiten:

- Ist eine Paritätsprüfung nicht von entscheidender Wichtigkeit (etwa weil bereits bekannt ist, daß Datenfehler äußerst selten auftreten), so kann die Parität vor der Speicherung der Daten in das FIFO geprüft und bei einem Fehler ein Paritätsfehler-Flipflop gesetzt werden. Dieses wird bei einem Schreibzugriff auf die Adresse des Moduls wieder zurückgesetzt. Bei diesem Verfahren ist erfährt man nicht wo genau der Fehler aufgetreten ist.

- Will man genau wissen, bei welchem Datenwort der Fehler auftrat, so kann das Ergebnis der Paritätsprüfung statt des EOL Bits mit den Daten in das FIFO gespeichert werden. Die Unterteilung der Ereignisse in Unterabschnitte kann auch mit dem EOE-Bit alleine vorgenommen werden, indem die mit dem EOE-Bit übertragenen Daten angeben, ob es sich um eine EOE oder EOL-Markierung handelt.

Um den Transfer über den TPB schneller ablaufen lassen zu können, wird nicht der $\overline{\mathrm{FF}}$(*full flag*)-Ausgang des FIFOs als FULL_TPB-Signal an das DACOX-Modul zurückgegeben, sondern der $\overline{\mathrm{HF}}$(*half full*)-Ausgang. Dies hat zwar zur Folge, daß bereits bei halb gefülltem FIFO der Transfer angehalten wird[12], dafür ist aber sichergestellt, daß selbst im

[12]Dies ist unproblematisch, da das FIFO mit seinen 4096 Speicherplätzen ca. 40 Ereignisse halten kann, eine Zahl, die die zu bauenden Pipeline-Module wohl kaum erreichen können.

Fall, daß die Übertragungsstrecke das FULL_TPB-Signal stark verzögert, kein Datenwort durch einen Überlauf des FIFOs verlorengeht.

Da auf eine Übertragung der Zusatzbits EOE und EOL bzw. Parität in die CPUs nicht verzichtet werden darf (sonst könnten aufeinanderfolgende Ereignisse nicht voneinander getrennt und Fehler nicht erkannt werden), können die Daten nicht mit `D16` VME-Zugriffen ausgelesen werden. Stattdessen wird mit einem `D32` Zugriff jeweils ein Datenwort, die beiden Zusatzbits und der Zustand des Moduls ausgelesen. Dies beinhaltet den Wert des Paritätsfehler-Flipflops und der $\overline{\mathrm{EF}}$(*empty flag*)- und $\overline{\mathrm{HF}}$-Leitungen des FIFOs.

Adresse	lesen	schreiben
0xXXXX0000 ⋮ 0xXXXX7FFC	Datenregister	Paritätsfehler-Flipflop und FIFO löschen
0xXXXX8000 ⋮ 0xXXXXFFFC	Statusregister	

Tabelle 3: **Adreßbelegung des DARX-Moduls**
Alle Zugriffe sind `D32`-Zugriffe; `D16` und `D8EO` verursachen einen Zugriffsfehler (*bus error*)

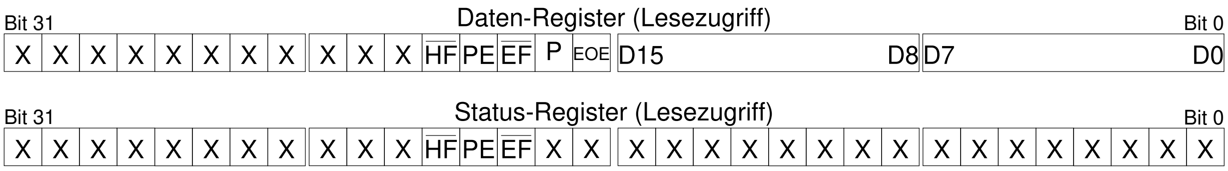


Abbildung 10: **Belegung von Daten- und Statusregister des DARX-Moduls**
Die beiden Register können nur mit `D32`-Zugriffen angesprochen werden. $\overline{\mathrm{HF}}$ und $\overline{\mathrm{EF}}$ geben den Zustand der entsprechenden Leitungen des FIFOs wieder; PE zeigt einen Paritätsfehler an. D0..15, EOE und P geben die über den TPB empfangenen Daten, das EOE-Bit und das Paritätsbit wieder.

Um den Zustand des Moduls testen zu können ohne eventuell ein Datenwort aus dem FIFO zu entfernen, ist ein weiteres Register, das Statusregister, vorhanden, welches nur den Status des Moduls wiederspiegelt, aber keine Daten liefert (vgl. dazu Abbildung 3). Der Einfachheit halber hat es die selbe Bit-Belegung wie das Datenregister (siehe Abbildung 10).

5 Bau und Test der Module

Die Module wurden zunächst einzeln aufgebaut und getestet. Dabei wurde durch sukzessives Bestücken und anschließenden Test des Boards im teilbestückten Zustand jeweils die Fehlerquelle auf wenige Komponenten beschränkt. In dieser Phase waren die auf beiden Platinen vorgesehenen Anschlüsse für den Logikanalysator von großem Nutzen, da sie ein schnelles Auf- und Abkabeln ermöglichten, welches für jede größere Modifikation vorgenommen werden mußte.

Nach dem Aufbau des DACOX-Boards und ersten erfolgreichen Datentransfers wurde die Fehlerfreiheit geprüft. Dazu wurde jeweils ein 32 Worte großer Speicherbereich mit Zufallsdaten gefüllt. Dann wurde DACOX programmiert, diesen über VME auszulesen und die Daten einerseits in das auf dem Board befindliche FIFO zu laden und andererseits in einen anderen Speicherbereich zu kopieren. Nach Ausführung des Programms wurde dann der Inhalt des FIFOs und des Speichers mit den geschriebenen Daten verglichen und eventuelle Fehler ausgegeben. Dieses Programm lief ohne Fehler während eines Wochenendes und führte dabei etwa 3,5 Millionen Zyklen durch, was einer Übertragung von 450 Megabyte oder 2,2 Millionen Ereignissen mit durchschnittlich 100 Worten entspricht. (= 8 Stunden Datennahme).

Vor der Einbindung der Module in das Experiment wurde ein gemeinsamer Test beider Module durchgeführt. Dabei wurden beide in verschiedenen Crates installiert und DACOX programmiert, einen festen, mit beliebigen Daten gefüllten Speicherbereich von 100 Langworten auszulesen. Dieses Programm wurde mit einem frei laufenden Triggersignal gestartet. Die übertragenen Daten wurden mit den erwarteten Werten verglichen und alle Abweichungen gezählt. Auf diese Art und Weise wurden die Daten 5.4 Millionen mal übertragen, ohne daß dabei ein einziger Datenfehler oder Paritätsfehler auftrat. Dies entspricht einer fehlerfreien Übertragung von 110 MByte oder etwa 1 Woche Datennahme.

Für die Verwendung im Experiment mußten noch die ins DACOX-Modul zu ladenden Programme geschrieben werden. Durch Einbau des `m4`-Präprozessors in den in Kapitel 3.5 und Anhang C beschriebenen DACOX-Assembler konnte die Programmierung nochmals deutlich besser strukturiert werden. Es wurde eine Makro-Bibliothek geschaffen, die die Programmierung von Schleifen und VME-Zugriffen erleichtert und die Auslese von Modulen, die über CAMAC- oder VMV-Bridges angesprochen werden müssen, automatisiert. Zur einfachen Konfiguration der Triggerauslese wurden für jeden auszulesenden Modultyp Auslesemakros bereitgestellt, die dann für jeden *run* zu einem Programm zusammengestellt werden können.

Nach Bereitstellung der vollständigen DACOX-Programme und Umstellung der Auslese-Software des Experimentes auf Pipeline-Auslese wurde in mehreren Durchgängen gemessen, wie lange das DACOX-Modul zur Auslese der Daten benötigt. Dabei wurde jeweils die Auslese verschiedener Teile der Hardware programmiert. Die dabei gemessenen Zeiten sind in Tabelle 4 zusammengestellt.

Die Spalte 'Datenmenge' enthält nicht die zusätzlich transferierten 8 Datenworte, die die jeweiligen Daten ankündigen (den sogenannten *header*) und das Wort, das das Ende der Daten markiert. Außerdem sind je nach Modul zusätzliche Statusregisterzugriffe nötig,

Module	Datenmenge	Auslese über	benötigte Zeit
4 Downscaler 2 Downscaler	4 Worte 2 Worte	VMV VME	50 μs
4 ADC 1182	32 Worte	VME	170 μs
1 TDC 1172	0 Langworte	VME	30 μs
1 CTDC	1 Wort	CAMAC	130 μs

Tabelle 4: Zeitaufwand zur Auslese verschiedener Module

um die Verfügbarkeit von Daten prüfen und/oder das Modul auf das nächste Ereignis vorbereiten.

Das TDC-Modul 1172 lieferte keine Daten, da dieser Test ohne Strahl durchgeführt wurde und deshalb die Strahldetektoren BC1-3 keine Teilchen nachwiesen. Die 30 μs wurden benötigt um den *header* zu schreiben, den Status des Moduls abzufragen, und es zurückzusetzen.

Der CAMAC-TDC benötigt mit 130 μs einen relativ großen Bruchteil der Auslesezeit, obwohl nur ein einziges Wort ausgelesen wird. Es sollte versucht werden ihn durch einen schnelleren VME-TDC zu ersetzen, da er die Totzeit der Triggerdatenauslese überdurchschnittlich vergrößert.

Die Beiträge zur Totzeit, die nicht von der variablen Datenmenge des TDC-Moduls abhängen sind oben angegeben und summieren sich zu 380 μs. Rechnet man bei der Auslese des TDC mit 30 Langworten, die je 4 μs beanspruchen, so dauert die Auslese von insgesamt 100 Worten 500 μs.

Dieser Wert übersteigt zwar die für die endgültige Datenauslese angestrebte Zeit von 300 μs, fällt aber in der Strahlzeit 1996 nicht ins Gewicht, da die SDD eine mehr als doppelt so große Totzeit haben.

Die nötige Geschwindigkeitssteigerung kann jedoch relativ einfach erreicht werden: Im Zuge des Aufbaus des Prototyps mußte die Taktfrequez des Moduls mehrmals gesenkt werden, da wegen einiger Modifikationen an der Programmierung der CPLDs die Hardware nicht mehr optimal an die Bedürfnisse der programmierbaren Logik angepasst war. Durch Vertauschen von I/O-Pins oder Verwendung eines großen CPLDs anstatt der zwei kleinen könnte bei einem Neudesign der Platine die Taktfrequenz ohne weiteres verdoppelt werden.

6 Leistungsfähigkeit des *1-event pipelining*

Während der Vorbereitung der Bleistrahlzeit im Oktober 1996 wurden Messungen zur Leistungsfähigkeit der aufgebauten Ausleseelektronik vorgenommen. Dabei wurde die Zahl der aufgenommenen Ereignisse (pro *burst*) als Funktion der angebotenen Trigger (pro *burst*) bestimmt. Die Zahl der angebotenen Trigger wurde durch Änderung der Strahlintensität variiert und enthält schon die *before-protection*, nicht jedoch die *after-protection*.

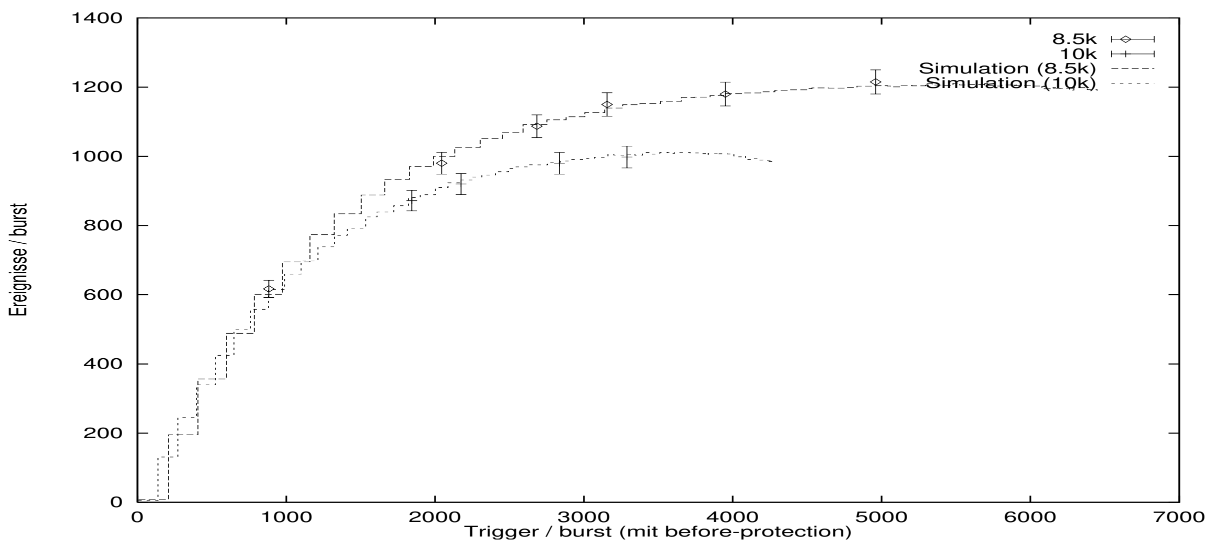


Abbildung 11: Ereignisrate als Funktion der Triggerrate
Diese Grafik zeigt die Zahl der pro *burst* aufgenommenen Ereignisse als Funktion der Zahl der pro *burst* angebotenen Trigger. Es wurden zwei Datensätze mit unterschiedlicher Multiplizitätsschwelle aufgenommen, sodaß die durchschnittliche Datenmenge pro Ereignis einmal 8.5 k Langworte (niedrige Schwelle) und einmal 10 k Langworte (hohe Schwelle) beträgt. Die gemessenen Daten (Punkte) werden gut durch die Ergebnisse einer Simulation beschrieben (Stufen).

Abbildung 11 zeigt zwei aufgenommene Datensätze, denen Ereignisse mit durchschnittlichen Datenmengen von 8.5 k Langworten[13] bzw. 10 k Langworten zugrunde liegen. Der Datensatz mit längeren Ereignissen dient dem Vergleich mit der Datenerfassung von 1995, die noch nicht mit der 1996 eingeführten Datenkompression ausgestattet war.

Ebenfalls in Abbildung 11 eingezeichnet sind Simulationsergebnisse, die die Daten beschreiben. Für die Simulation wurde angenommen, daß jeweils eine fest vorgegebene Zeit benötigt wird, um ein Ereignis vom Detektor in die Receiver, von den Receivern in die Memorymodule und von den Memorymodulen in den Speicher der CPU zu bringen. Die verschiedenen Detektoren wurden wie ein einzelner Detektor behandelt, da im wesentlichen der langsamste Detektor die Auslesezeit bestimmt. Nimmt man an, daß sich die Datenmenge immer gleich auf die einzelnen Detektoren verteilt, so skalieren die drei Zeitparameter auch proportional zur Gesamtdatenmenge. Die Simulation protokollierte genau mit, wo

[13]Ein Langwort entspricht 32 Bit oder 4 Bytes.

gerade Daten übertragen wurden und reagierte entsprechend auf die von einem Zufallsgenerator erzeugten Trigger. Nach diesem Muster wurden Simulationen der herkömmlichen Datenerfassung, des *1-event pipelining* und einer Pipeline-Auslese mit variabler Tiefe programmiert.

Durch Anpassung der Simulation an die Daten wurden die Zeitparameter der verschiedenen Datentransfers (Detektor $\Rightarrow$ Receiver, Receiver $\Rightarrow$ Memorymodul und Memorymodul $\Rightarrow$ CPU) bestimmt. Diese sind in Tabelle 5 wiedergegeben und stimmen mit den Kenndaten der einzelnen Komponenten gut überein.

von	nach	Zeit	
		8.5 k	10 k
Detektor	Receiver	1.014 ms	1.193 ms
Receiver	Memorymodul	0.394 ms	0.463 ms
Memorymodul	CPU	2.705 ms	2.986 ms

Tabelle 5: Datenübertragungszeiten im *1-event pipelining*

Bei der Anpassung an die Datensätze erwartet man, daß sich die Übertragungszeiten proportional zur Datenmenge erhöhen; die Übertragungzeit in die CPUs ergibt sich aus einem Sockelbetrag von 1.1 ms und einem Betrag, der entsprechend der VME-Übertragungsrate von 21 MB/s mit der Datenmenge ansteigt.

Allein durch Erhöhung der Strahlintensität können nicht beliebig hohe Triggerraten erreicht werden, da dann immer mehr Ereignisse durch *before-protection* und *after-protection* verworfen werden. Wegen der *before-protection* brechen in in Abbildung 11 die Kurven bei der maximal erreichbaren Triggerrate ab.[14] Die *after-protection* führt dazu, daß in Abbildung 11 die Zahl der aufgenommenen Ereignisse ab einem gewissen Punkt (oder genauer: einer gewissen Strahlintensität) wieder abnimmt.

Für die Diskussion der Leistungsfähigkeit des Auslesesystems sind die durch die *before-after-protection* der Triggerlogik hervorgerufenen Limitierungen nicht von Interesse. Deshalb wird bei den folgenden Überlegungen die *before-after-protection* nicht berücksichtigt.

In Abbildung 12 sind Simulationen einer herkömmlichen Datenerfassung, des *1-event pipelining* und einer Auslese mit vierstufiger Pipeline dargestellt.

Für die herkömmliche Datenauslese wurde eine Totzeit von 3.5 ms angenommen, was dem 1995 erreichten Wert entspricht. Den Simulationen des *1-event pipelining* liegen die Werte aus Tabelle 5 zugrunde. Für die vierstufige Pipeline wurden die Daten des 8.5 k Datensatzes mit einer verbesserten Zeit von 0.3 ms für den Transfer in die Receiver kombiniert.

Man erkennt in Abbildung 12, daß die Zahl der pro *burst* aufgenommenen Ereignisse für große Triggerraten in einen Sättigungswert übergeht, der gleich dem Quotienten aus der

[14] Daß die beiden Kurven bei unterschiedlichen Triggerraten abbrechen rührt daher, daß die unterschiedlichen Ereignislängen durch Änderung der Multiplizitätsschwelle erreicht werden. Deshalb werden bei der selben Strahlintensität einmal 6400 und einmal 4200 Trigger erreicht.

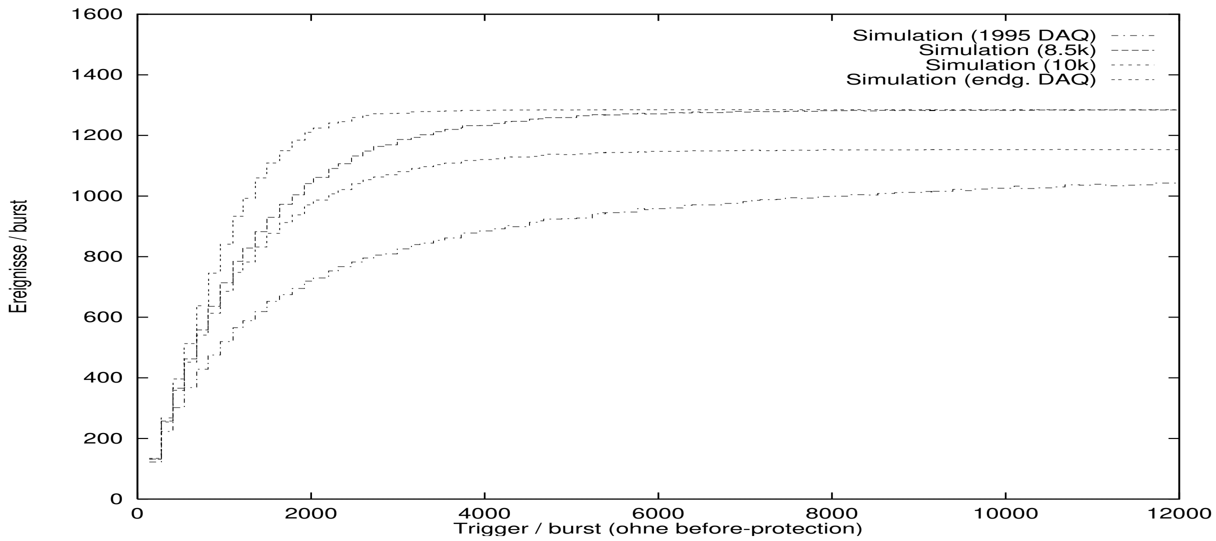


Abbildung 12: Simulierte Ereignisrate als Funktion der Triggerrate

effektiven *burst*-Länge $t_{\mathrm{B,eff}}$[15] und der Totzeit des Systems bei ständig gefüllter Pipeline ist. Dies sind im Falle der alten Datenerfassung natürlich 3.5 ms und für die restlichen Kurven die Summe der Übertragungszeiten von den Receivern zum Memorymodul und vom Memorymodul in die CPU. Die Sättigung wird mit abnehmender Pipeline-Tiefe bei immer größeren Triggerraten erreicht, sodaß sich die Kurve der herkömmlichen Datenerfassung erst weit ausserhalb der Abbildung waagrecht wird.

Für kleine Triggerraten ist die Totzeit des Systems bei geleerter Pipeline entscheidend. Dies kann folgendermaßen veranschaulicht werden: Ist t_{B} die Länge des *bursts*, T die Zahl der pro *burst* angebotenen Trigger und E die Zahl der pro *burst* aufgenommenen Ereignisse, so kann man mit der Formel

$$\overline{t_{\mathrm{tot}}} = t_{\mathrm{B}} \cdot \left(\frac{1}{E} - \frac{1}{T}\right) \tag{1}$$

eine über den gesamten *burst* gemittelte Totzeit $\overline{t_{\mathrm{tot}}}$ berechnen. Diese gemittelte Totzeit ist in Abbildung 13 gegen die Zahl der angebotenen Trigger aufgetragen.

In dieser Auftragung erkennt man, welche Zeit für die Erfassung eines Ereignisses benötigt wird. Dies sind im Falle der alten Datenerfassung unabhängig von der Triggerrate immer 3.5 ms, bei den übrigen Systemen geht die Kurve im Grenzfall kleiner Triggerraten gegen die Zeit, die für die Übertragung in die Receiver benötigt wird. Mit zunehmender Triggerrate steigt sie allmählich an. Dieser Anstieg vollzieht sich umso langsamer, je tiefer die Pipeline ist.

Für sehr große Triggerraten erwartet man, daß die Totzeit gegen die Zeit geht, die für den Datentransfer vom Receiver über die Memorymodule in die CPUs benötigt wird. Mit zunehmender Pipeline-Tiefe wird der Sättigungswert der Totzeit bei immer höheren

[15] Die effektive *burst*-Länge $t_{\mathrm{B,eff}}$ ist etwas kleiner als die tatsächliche Länge t_{B}, und berücksichtigt die Tatsache, daß sich die Strahlintensität nicht homogen auf den gesamten *burst* verteilt.

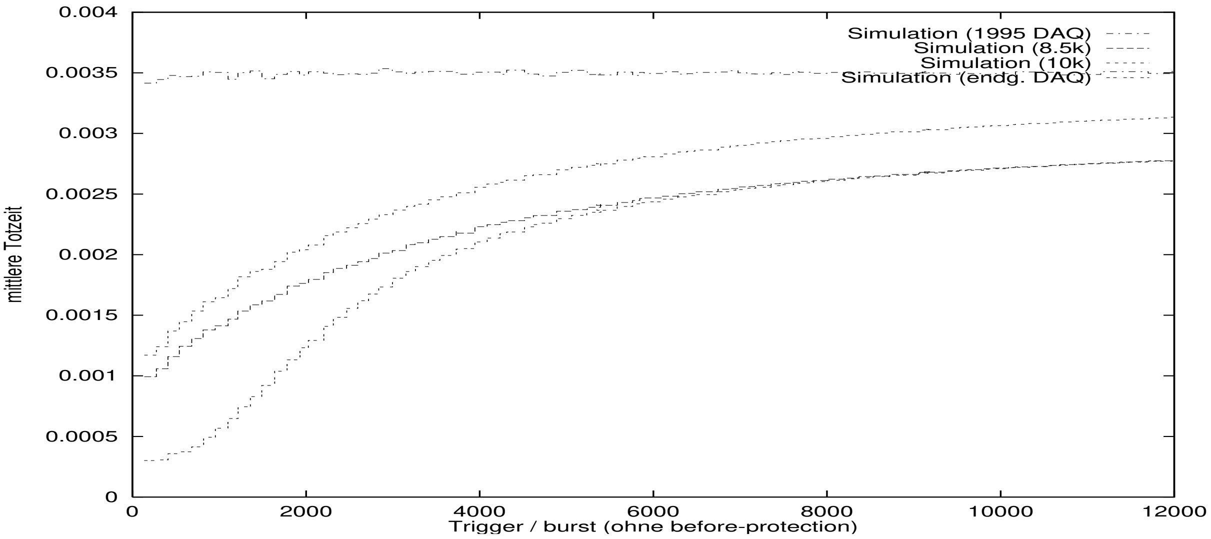


Abbildung 13: Die mittlere Totzeit als Funktion der Triggerrate

Triggerraten erreicht, sodaß man diese Zeitparameter besser aus Abbildung 12 ablesen kann.

Während der Strahlzeit 1995 wurden bei etwa 1000 Triggern pro Burst ca. 530 Ereignisse aufgenommen, von denen bei der späteren Analyse aber etwa 20-25% durch die *after-protection* ausgesondert wurden, da ein weiteres Strahlteilchen den Detektor innerhalb von 1.5 μs passierte. Solche Ereignisse wurden 1996 bereits während der Datennahme durch den *Abort*-Mechanismus entfernt, sodaß alle aufgenommenen Ereignisse verwendet werden können.

Durch Optimierung der Detektoren konnte außerdem die Strahlintensität auf 1.1 Millionen Strahlteilchen pro *burst* erhöht werden, sodaß durchschnittlich 1050 verwendbare Ereignisse pro *burst* aufgezeichnet werden konnten. Im Vergleich zu den 400 verwendbaren Ereignissen von 1995 ist dies eine Steigerung um den Faktor 2.5. Nimmt man an, daß das vollständig ausgebaute Auslesesystem bei der gleichen Triggerrate von etwa 2000 Triggern pro *burst* betrieben wird, so liest man an Abbildung 12 ab, daß mit dem vollständigen Pipeline-System etwa 1200 Ereignisse aufgezeichnet werden können. Dies entspricht einer Steigerung um den Faktor 3.

Da in in Abbildung 12 die Kurve der vierstufigen Pipeline bei 2000 Triggern schon fast die Sättigung erreicht hat, bringt eine Erhöhung der Pipeline-Tiefe keinen entscheidenden Vorteil mehr. Weitere Verbesserungen können aber noch durch eine höhere Datentransferrate der Memorymodule zu den CPUs erreicht werden, und sind derzeit in Vorbereitung.

7 Zusammenfassung

Im Rahmen der vorliegenden Diplomarbeit wurde diskutiert, welche Vorteile man von Umstellung des Auslesesystems des CERES-Detektors auf das Pipeline-Konzept erwartet.

Es wurde erörtert, welche Veränderungen an der bisher vorhandenen Hardware dazu nötig sind und über welche Ausbaustufen (*1-event pipelining*, Pipelining ohne *event removal* und Pipelining mit *event removal*) das endgültige System erreicht werden soll.

Es wurden zwei VME-Module entworfen, gebaut, getestet und in Betrieb genommen, die in allen drei Ausbaustufen benötigt werden und das Pipelining-Konzept im Bereich der Triggerdetektoren verwirklichen.

Für beide Module wurden im vorliegenden Text die Vorgaben analysiert, die Hardware beschrieben und der Aufbau der programmierbaren Logik dargestellt. Der Anhang dieser Arbeit gibt die ebenfalls geschaffene Dokumentation wieder, die dem Benutzer der Module an die Hand gegeben wird.

Es wurde eine Assemblersprache entworfen, mit der der Benutzer den Datenauslesevorgang beschreiben kann. Für diese Sprache wurde ein Compiler entwickelt, der die Binärcodes erzeugt, die den Auslesevorgang steuern. In dieser Sprache wurden Routinen geschrieben, die die Auslese aller im Experiment verwendeten Module ausführen.

Prototypen der entworfenen VME-Module werden zum Zeitpunkt der Abfassung dieser Arbeit erfolgreich im Experiment eingesetzt und tragen nicht zur Totzeit des Gesamtsystems bei. Die endgültigen Module, für die im wesentlichen ein Neuentwurf der Platine notwendig sein wird, werden ohne Probleme die im voll ausgebauten Auslesesystem geforderten Spezifikationen einhalten.

Die in der Blei-Strahlzeit 1996 aufgebaute erste Ausbaustufe, das *1-event pipelining*, hatte wie erwartet am Front-Ende eine Totzeit von ca. 1.2ms, gegenüber der 1995 verwendeten herkömmlichen Auslese konnte die Zahl der pro *burst* aufgenommenen verwendbaren Ereignisse um einen Faktor 2.5 erhöht werden.

A DACOX Hardware Configuration

A.1 Base Address Switches

The base address switches for (A16-23) are situated on the upper edge of the board (`SW1`, see fig. 15(a)). The leftmost one is for A16 and the rightmost one for A23. If a switch is turned on, the corresponding address bit has to be low for the address to match. Note that these bits are ignored in `A16` cycles.

The base address switches for (A24-31) are situated in the lower right corner of the board (`SW2`, see fig. 15(b)). The lowest one is for A24 and the upmost one for A31. If a switch is turned on, the corresponding address bit has to be low for the address to match. Note that these bits are ignored in `A24` and `A16` cycles.

A.2 Bus Request Jumpers

The requester of the module can operate on any of the four bus request levels `BR0-3`. Configuration of the bus request level is done with jumpers that can be found on the right hand side of the board next to the P1 connector (see fig. 16(a)). The upper 2×8 pins configure the bus grant daisy chain, while the lower 2×4 pins configure the bus request level. The settings for each level can be determined from figure 16(b).

A.3 LEDs

The board has 6 LEDs in a row whose meaning (from top to bottom) is given in table 6.

<table>
<tr><th>No</th><th>NAME</th><th>OFF</th><th>ON</th></tr>
<tr><td>1</td><td>MODE</td><td>slave mode</td><td>master mode</td></tr>
<tr><td>2</td><td>BUSY</td><td>waiting for trigger</td><td>busy or slave mode</td></tr>
<tr><td>3</td><td>PRGM0</td><td colspan="2" rowspan="2">current or last
program</td></tr>
<tr><td>4</td><td>PRGM1</td></tr>
<tr><td>5</td><td>$\overline{\text{EF}}$</td><td>FIFO is empty</td><td>FIFO contains data</td></tr>
<tr><td>6</td><td>DS_TPB</td><td colspan="2">shows TPB activity</td></tr>
</table>

Table 6: DACOX LEDs

A.4 LEMO Connectors

The board has 7 LEMO connectors whose meaning (from top to bottom) is given in table 7.

A.5 Twisted Pair Bus Pinout

The twisted pair bus connector is a 50-pin ribbon cable connector. Two neighbouring pins are dedicated to each signal. The pinout is given in table 8.

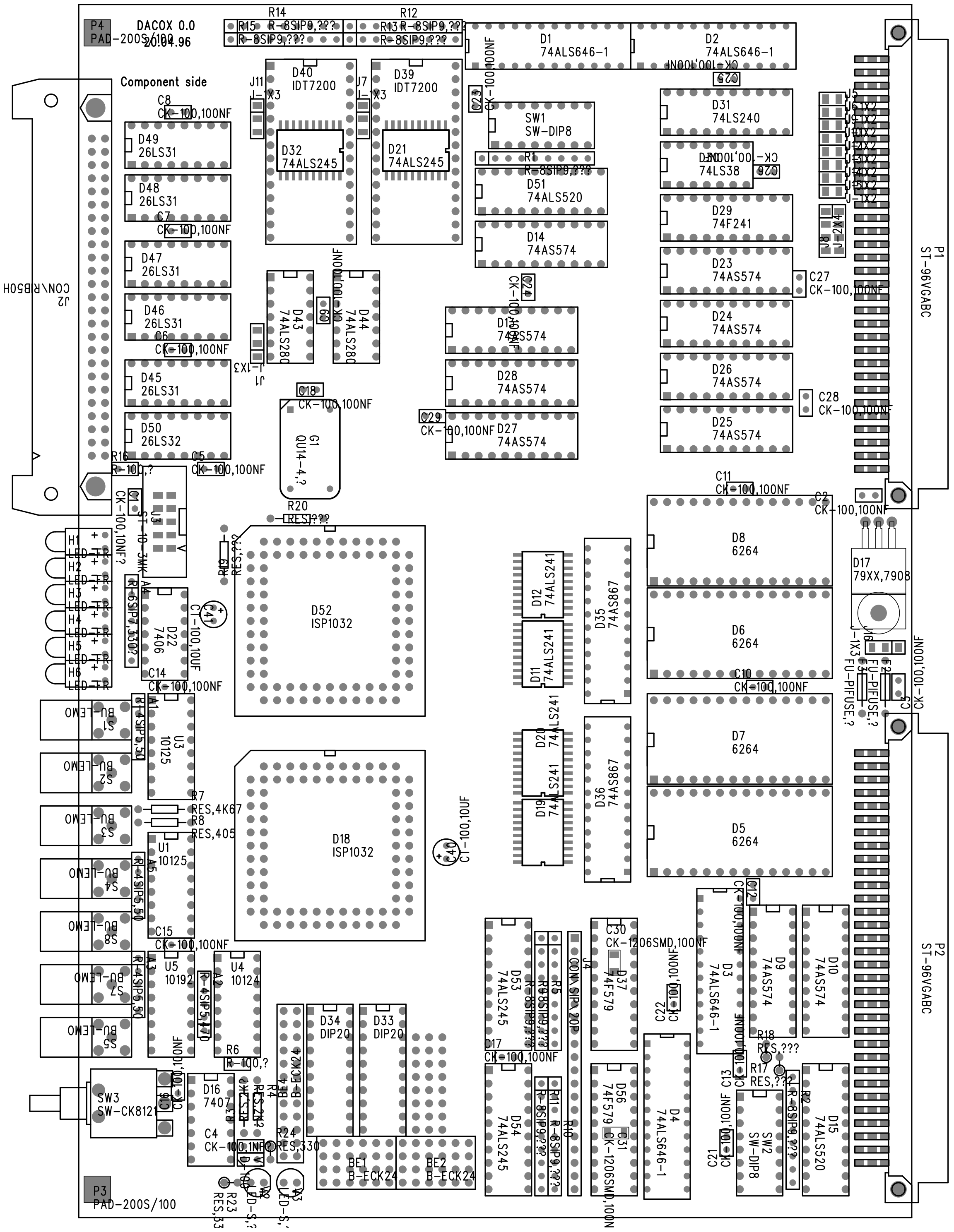


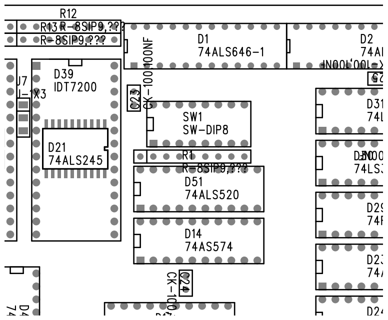
Figure 14: DACOX Board overview

No	NAME	in/out	meaning:
1	TRIG0	in	Trigger inputs
2	TRIG1	in	
3	TRIG2	in	
4	TRIG3	in	
5	ABORT	in	Abort input
6	RESET	in	Resets the whole board
7	BUSY	out	Busy output for trigger logic

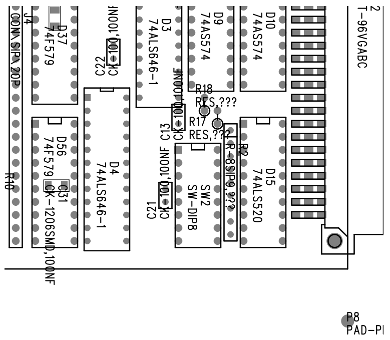
Table 7: DACOX LEMO connectors

Pin	in/out	description
50,49	out	$\overline{\mathrm{EOL}}$, EOL
48,47	out	$\overline{\mathrm{EOE}}$, EOE
46,45	out	$\overline{\mathrm{DS\_TPB}}$, DS_TPB
44,43	out	$\overline{\mathrm{PARITY}}$, PARITY
42,41	out	$\overline{\mathrm{D15}}$, D15
40,39	out	$\overline{\mathrm{D14}}$, D14
⋮	out	⋮
14,13	out	$\overline{\mathrm{D01}}$, D01
12,11	out	$\overline{\mathrm{D00}}$, D00
10,9	out	No Connection
⋮	out	
4,3	out	
2,1	in	$\overline{\mathrm{FULL\_TPB}}$, FULL_TPB

Table 8: DACOX twisted-pair-bus pinout



(a) A16-23



(b) A24-31

Figure 15: DACOX Address switches

A.6 FIFO jumpers

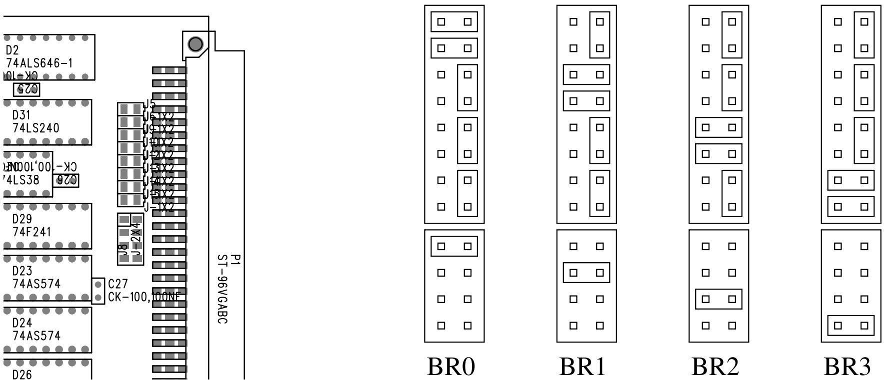
In order to make debugging easier, the bits 6 and 14 of the FIFO can be configured to store the two extra bits EOE and EOL instead of bits 6 and 14 of the data. This way EOL and EOE can be monitored from VME. To configure this behaviour, a three-pin jumper is located on the left hand side of each FIFO (see fig. 14). The jumper next to the right hand side FIFO controls IB06 and EOE while the one next to the left FIFO controls IB14 and EOL. If you connect the upper two pins, the FIFO stores the normal data bit (bit 06/14). If the lower two pins are connected, the FIFO will store EOE/EOL.

A.7 Parity jumper

The parity bit that is sent to the output bus along with the data can be chosen to be either even or odd parity. This can be configured with the parity jumper next to the parity generator chips. They are located between the clock generator and the FIFOs (see 14). If the middle pin is connected to the upper one, even parity is transmitted, e.g. if the data is `0x0000` the parity bit will be `1`. If it is connected to the lower pin, odd parity is transmitted, e.g. if the data is `0x0000` the parity bit will be `0`.

A.8 RAM/EPROM sockets

The RAM/EPROM chips are situated on the right edge of the board. There are 4 28-pin DIP sockets which can accept up to 16 k $\times$ 16 bit RAM or EPROM. Initially, the sockets were designed to contain 4 8 k $\times$ 8 bit RAM chips, where the upper two sockets (D6 and D8) are connected to the bits 0..7 of the internal bus IB, while the lower two ones (D5 and D7) are connected to the bits 8..15. In each of these pairs, the lower sockets (D6/D5) cover the lower half of the RAM space (VME addresses `0xXXXX0000` .. `0xXXXX1FFF`) while



(a) Location (b) Settings

Figure 16: DACOX Address switches

the upper sockets (D8/D7) cover the rest (VME addresses `0xXXXX0000` .. `0xXXXX1FFF`). However, also the following are examples of use:

D8	D6	D7	D5	Remarks
6264 (8k x 8)	6264 (8k x 8)	6264 (8k x 8)	6264 (8k x 8)	Standard setup, 16k x16
– 0k	6264 (8k x 8)	– 0k	6264 (8k x 8)	upper bank empty, 8k x16
62256 (32k x 8)	– 0k	62256 (32k x 8)	– 0k	only half of the chips used, 16k x16

Other configurations with smaller or EPROM chips should be possible with little or no change to the board.

A.9 IB terminators

The internal bus IB runs once across the board on the right hand edge near the VME connectors. On both ends it can be terminated to obtain clear signals. The sockets for the terminating resistors are designed to accept 4 9-pin resistor networks on each end of the bus, two pulling up and two pulling down.

The upper bus termination sockets can be found on the upper edge of the board near the FIFOs. The upper ones have a GND connection while the lower ones have a +5V connection on their leftmost pins.

The lower bus termination sockets can be found in the middle of the lower edge of the board next to the test connector for the logic analyzer. Here the right ones have +5V connection while the left ones have GND connection on their topmost pins.

A.10 ISP Programming

The ISP programming socket is situated on the left side of the board near the LEDs. The pinout is the following:

$\overline{\text{ISPEN}}$	10	9	SDI
NC	8	7	MODE
+5V	6	5	SDO
GND	4	3	CLK
NC	2	1	NC

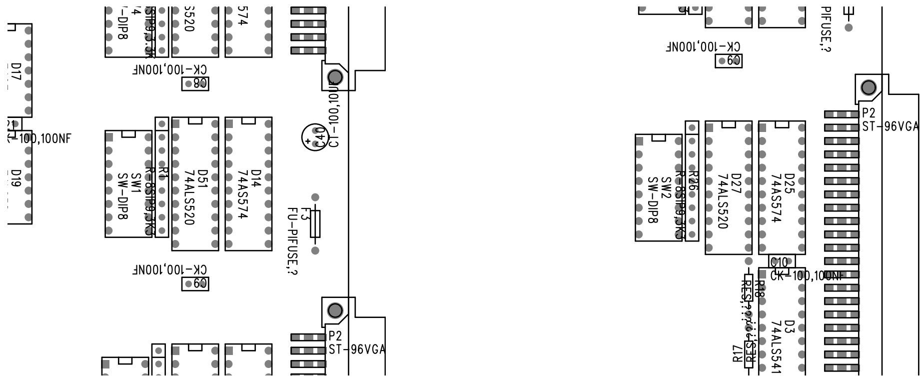
There are two ispLSI1032 chips on the board of which the first one (lower, D18) is to be loaded with the file `Lattice1.jed`, which contains the bus control logic. The second one (upper, D52) must be loaded with the file `Lattice2.jed` that contains CSR, address decoder, requester and more.

A.11 Power Supply

The board is equipped with a -5V DC converter in order to generate the -5V power supply needed for the ECL-chips that interface the LEMO connecters. However, in some VME-crates, a -5V supply is available on pin C4 on VME-connector P2. The user can select one of these sources with a 3-pin jumper between the two VME connectors. If the middle pin is connected to the right pin, the -5V is generated from the VME bus -12V. Otherwise, if the middle pin is connected to the left pin, the -5V is supplied from P2-C4. The presence of the -5V supply is indicated by a red LED at the lower left edge of the board. The green LED to the right of it indicates that +5V is OK. If either of these is not on, the fuses, situated between the two VME connectors, should be checked. The left fuse is for +5V, the right one for -5V.

B DARX Hardware Configuration

B.1 Base Address Switches



(a) A16-23 (b) A24-31

Figure 17: DARX Address switches

The base address switches for (A16-23) are situated on the right edge of the board between the VME connectors (see fig. 17(a)). The top one is for A23 and the bottom one for A16. If a switch is turned on, the corresponding address bit has to be low for the address to match.

The base address switches for (A24-31) are situated below the switches for A16-23 (see fig. 15(b)). The top one is for A24 and the bottom one for A31. If a switch is turned on, the corresponding address bit has to be low for the address to match.

B.2 LEDs

The board has 6 LEDs in a row which indicate (from top to bottom):

NAME	OFF	ON
VMER	—	VME read in progress
VMEW	—	VME write in progress
full	FIFO not full	FIFO full
empty	FIFO not empty	FIFO empty
Error	no parity error	a parity error occurred
DS_TPB	data transfer in progress	no data transfer in progress

B.3 Twisted Pair Bus Pinout

The twisted pair bus connector is a 50-pin ribbon cable connector. Two neighbouring pins are dedicated to each signal. The following table shows the pin assignment from top to

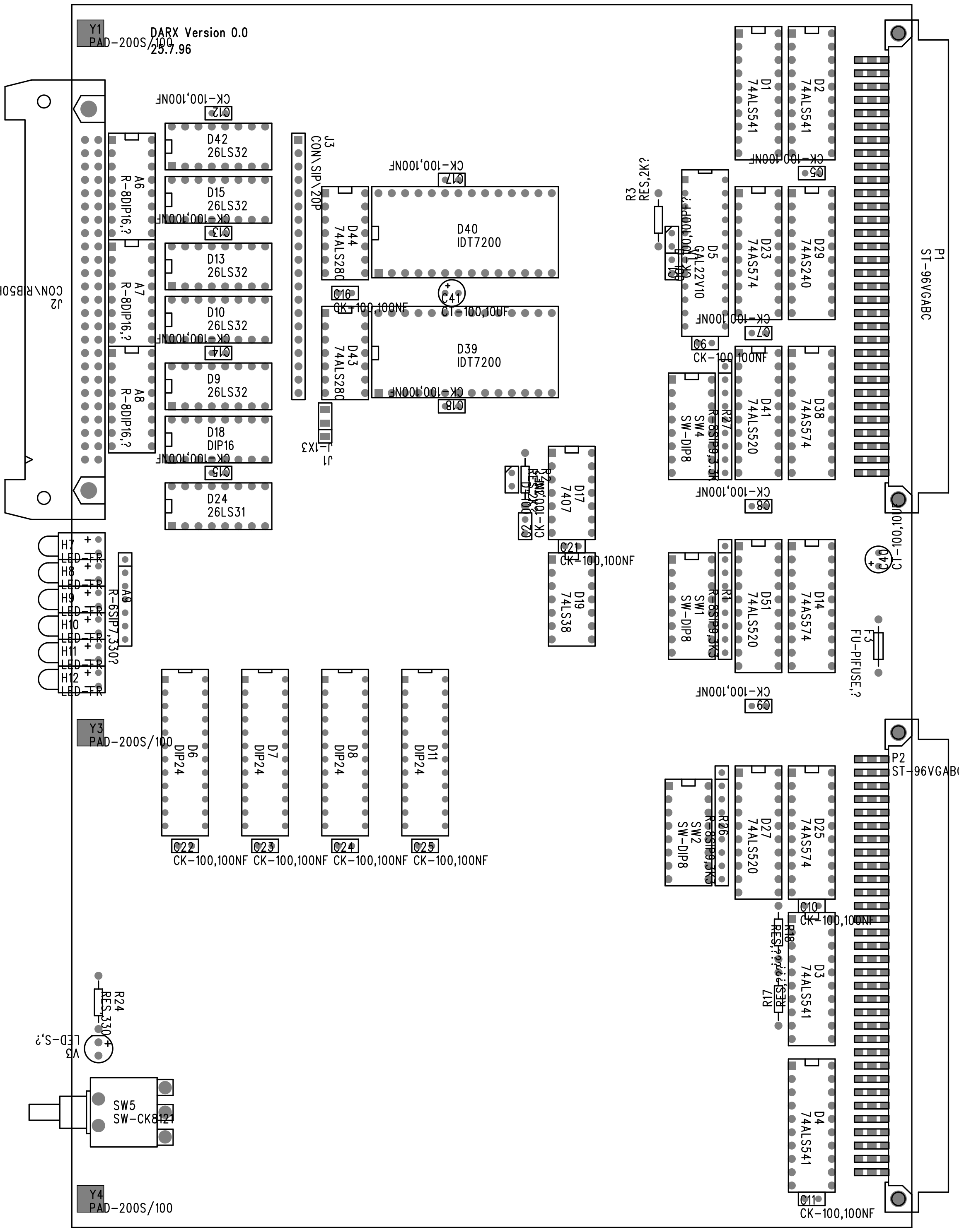


Figure 18: DARX Board overview

bottom:

Pin	in/out	description
50,49	out	$\overline{\text{EOL}}$, EOL
48,47	out	$\overline{\text{EOE}}$, EOE
46,45	out	$\overline{\text{DS\_TPB}}$, DS_TPB
44,43	out	$\overline{\text{PARITY}}$, PARITY
42,41	out	$\overline{\text{D15}}$, D15
40,39	out	$\overline{\text{D14}}$, D14
⋮	out	⋮
14,13	out	$\overline{\text{D01}}$, D01
12,11	out	$\overline{\text{D00}}$, D00
10,9	out	
⋮	out	No Connection
4,3	out	
2,1	in	$\overline{\text{FULL\_TPB}}$, FULL_TPB

B.4 Twisted Pair Bus Test connector

An oscilloscope or logic-analyzer can be connected to the TPB test connector, which interfaces the TTL outputs of the TPB line receivers. It is a 20-pin connector situated between the TPB line receiver and the parity generator chips (D43/D44). The pins have the following meaning (from top to bottom):

Pin	description
1	EOL
2	EOE
3	DS_TPB
4	PARITY
5	D15
6	D14
⋮	⋮
19	D01
20	D00

B.5 Parity jumper

The parity of the data received from the twisted-pair-bus (TPB) is checked against the parity bit that came with it. The parity checker can be configured to expect either even or odd parity using the parity jumper, which is situated just below the parity checker chip D43. If the middle pin is connected to the upper one, even parity is expected, e.g. if the data is `0x0000` the parity bit must be `1`. If it is connected to the lower pin, odd parity is expected, e.g. if the data is `0x0000` the parity bit must be `0`.

B.6 GALs

There are two GALs of type `P22v10` on the board. The first one, D5 is situated next to the P1 VME-bus connector and shall be loaded with the file 'Darxgal0.tex', which contains the decoding of VME signals and the parity flipflop. The second one, D11 is situated near the center of the board and shall be loaded with the file 'Darxgal1.tex', which contains logic for the $\overline{\text{OE\_FIFO}}$ signal.

B.7 +5V LED and fuse

A green LED near in the bottom left corner of the board indicates the presence of the +5V power supply voltage. If this LED is not on, the fuse situated between the two VME connectors should be checked.

C The Assembler

C.1 Introduction

This compiler manual describes the `dcxcomp` utility, a program that translates source code written for the DACOX board into the binary code that can be downloaded to the board's RAM space. `dcxcomp` has compiled successfully under HP/UX and Linux. It has also been ported to OS-9.

A very simplified overview over the DACOX board is given in figure 19.

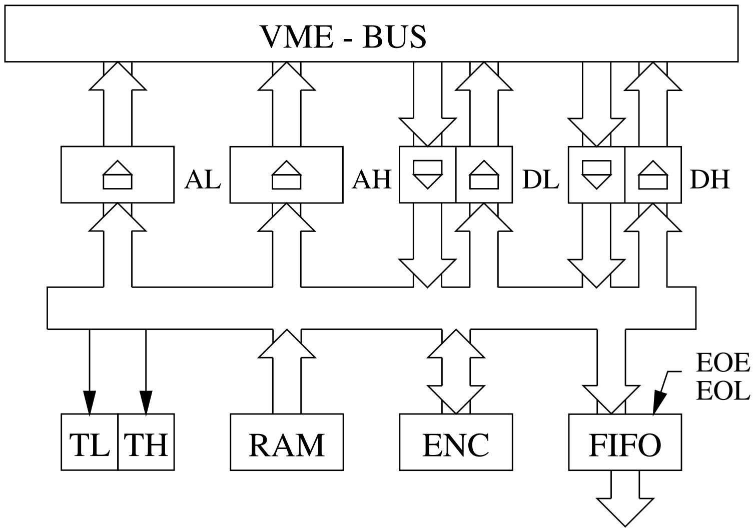


Figure 19: The DACOX Board — simplified overview

All data sources and destinations on the board are connected to an internal bus.

The data registers DL and DH are used to store the data that shall be written to VME and data that was read from there.[16] The address registers AL and AH store the address for VME cycles. They can only be written. The FIFO accepts the data that shall be transmitted via the twisted-pair cable. It is removed from the FIFO automatically. The ENC shall be used to count events. It is increased when its value is transferred to the FIFO. It can also be written to. The RAM can send data to the internal bus as well. All these bus components have a width of 16 bits. The only exception is the FIFO which can store 18 bits. The two extra Bits are used for storing end_of_label and end_of_event marks. The two test registers have a width of 1 bit each. The TL(TH) Bit can be loaded with one of Bits 0..7 (8..15) of the transferred data during every bus cycle.

C.2 Command line options

The `dcxcomp` utility understands the command line options shown in figure 20.

[16] Input and output registers are not the same, i.e. the output data cannot be read back.

If a filename is given on the command line, this name is taken as the input file. Note that here standard input ("-") is not allowed. [17] From this filename, the filenames of the output file and the symbol table file are created by appending the extensions ".dld" and ".sym".

The option `-p` allows you to specify, which filename shall be used for the preprocessors output. If you don't give this option, a name will be created by appending ".pre" to the body of the input file name.

Option `-P` specifies the preprocessor command that the compiler invokes using a **`system()`** call. If you don't want to use any preprocessor, you might fill in something like "**`copy &s &p`**". The default is "**`m4 &s > &p`**".

With option `-o <outfile>` you specify to which file the compiler's output shall be directed. The filename must be present, otherwise an error message will be displayed and the program will stop. When the output filename is "-", the output will be directed to standard output. If no `-o` option is given, the output filename will be determined from the input filename by appending a ".dld" extension.

If a `-s` option is specified, a symbol table will be generated and written to a file. If a filename follows, this file is used, otherwise, a name is created by appending ".sym" to the filename of the input file.

For debugging purposes only, the option `-d` turns on debugging printouts in the given routine. Up to 10 routines may be specified by giving more than one `-d` option.

If you invoke the program with wrong arguments, a command line help will be displayed and no compilation will be done.

C.3 Language description

In order to describe the language in this subsection, a Backus-Naur-like language is used, which is explained in figure 21. The syntax of mathematical expressions is designed to be similar to C. In the following paragraphs, some differences to C will be pointed out in footnotes, so C-programmers should read them to avoid funny results.

Numbers The compiler handles all numbers as integer numbers that are at least 16 bit wide. Do not rely on the width of the numbers, on some platforms the arithmetics might be done with 24, 32 or even more bits. Hexadecimal numbers must start with `0x`, decimal numbers with a non-zero decimal digit, octal numbers with a zero and binary numbers with `0b`.[18] As an example, `123`, `0x7B`, `0173` and `0b1111011` all represent the same number. Note that something like `0190` will result in two numbers, namely an octal `1` and a decimal `90`. The compiler will display a warning in these cases.

[17] The compiler scans the input file twice. It can't work as a filter because it can't rewind the standard input.

[18] This is the normal way, C-programmers are used to type numbers except for binary numbers that are not supported in C.

```
dcxcomp [option [option [ ...  ]]]
Valid options are:

<srcfile>     : read input from <srcfile>.  If not specified,
                'a.dcx' is assumed.
-p <prefile> : store preprocessed input in <prefile>.  If
                not specified, a name is derived from the
                souce file name.
-P <prepcmd> : use <prepcmd> to run the preprocessor.
                In <prepcmd>, '&s' will be replaced by the
                <srcfile>, '&p' by <prefile> and '&&' by '&'.
-o <outfile> : write output to <outfile>.  If not specified,
                the name will be determined from <infile>.
-s <symbfile>: print table of constants to <symbfile>.  If
                <symbfile> is not specified, make a name.
-d <rout>     : print debugging info from routine <rout> to
                the output file.
```

Figure 20: Command line options of the `dcxcomp` utility

```
 ( and )   are used for grouping
 [ and ]   enclose optional objects that can be left out
   ...     mark repetitions
    |      means that either the object before of after the
           | sign may stand here
 ' and '   delimit plain text, e.g.  keywords
    =      says the right side replaces the left side
```

Figure 21: Backus-Naur-like language conventions

```
   number  =  hex_number | oct_number
                | dec_number | bin_number
hex_number =  '0x' hex_digit ...
dec_number =  dec_exc_0 dec_digit ...
oct_number =  '0' oct_digit ...
bin_number =  '0b' bin_digit ...
 hex_digit =  dec_digit | 'A' | 'B'
                | 'C' | 'D' | 'E' | 'F'
 dec_exc_0 =  '1' | '2' | '3' | '4' | '5'
                | '6' | '7' | '8' | '9'
 dec_digit =  oct_digit | '8' | '9'
 oct_digit =  '0' | '1' | '2' | '3' | '4'
                | '5' | '6' | '7'
 bin_digit =  '0' | '1'
```

Figure 22: Numbers

Operators The operators that `dcxcomp` understands are listed in table 9, according to their precedence level. For example, `2 + 3 * 4` will be evaluated as `2 + ( 3 * 4 )` because the multiplication operator `*` comes first in table 9. All operators are evaluated from left to right, i.e. `1 << 2 << 2` is `( 1 << 2 ) << 2 = ( 4 ) << 2 = 16` (and not `1 << ( 2 << 2 ) = 1 << ( 8 ) = 256`). [19]

<table>
<tr><th>Operators</th><th>Explanation</th></tr>
<tr><td><code>! + - (</code></td><td>unary not, plus, minus</td></tr>
<tr><td><code>* / %</code></td><td>times, divide, modulus</td></tr>
<tr><td><code>+ -</code></td><td>plus, minus</td></tr>
<tr><td><code><< >></code></td><td>bit shift operators</td></tr>
<tr><td><code>< <= > >=</code></td><td>comparison</td></tr>
<tr><td><code>== !=</code></td><td>is equal, is not equal</td></tr>
<tr><td><code>&</code></td><td>arithmetic AND</td></tr>
<tr><td><code>^</code></td><td>exclusive-OR</td></tr>
<tr><td><code>|</code></td><td>arithmetic OR</td></tr>
<tr><td><code>&&</code></td><td>logical AND</td></tr>
<tr><td><code>||</code></td><td>logical OR</td></tr>
<tr><td><code>)</code></td><td>closing brace</td></tr>
</table>

Table 9: Operators according to their precedence level

[19] Additionally, the function for simplifying arithmetic expressions is not yet optimized to stop when the outcome is already known. For operators that are on their own precendence level (such as the logical OR) this makes no difference to C, because the assignment operator is not in our set.

Identifiers Identifiers are used as names for variables and constants. Identifiers are case sensitive. Programmers are encouraged to follow the convention, that constants are upper case while variables are lower case. Every identifier must start with a letter. This letter may be followed by up to 31 other characters, which may be letters, numbers, or underscores ('_'). [20] As an example, `Address`, `D_a_t_a` and `TDC1176` are valid identifiers, while `1176`, `_Data_` and `LWORD*` are not.

```
identifier  =  letter ( letter | dec_digit | '_' )...
   keyword  =  ccmd | instr | dwidth | IB_dest | jmpcond
      ccmd  =  'CONST' | 'VAR' | 'PTR'
     instr  =  'VMER' | 'VMEW' | 'MOV' | 'JMP' | 'STOP'
    dwidth  =  'D80' | 'D8E' | 'D16' | 'D32'
   IB_dest  =  IB_src | 'FIFO' | 'RAM' | 'AL' | 'AH'
    IB_src  =  'DL' | 'DH' | 'ENC' | expression
   jmpcond  =  'ABOR' | 'TH' | 'TL' | 'BERR' | 'PRGM'
```

Figure 23: Identifiers and Keywords

Keywords Keywords follow the same conventions as identifiers. In fact, the parser first reads an identifier and then determines, if it is a keyword or identifier. The recognised keywords are listed in table 10. Note that all keywords are uppercase (and case sensitive). If you misspell a keyword by using lowercase letters instead of uppercase ones, the compiler will take it as an identifier and output a warning.[21] You can't have variables that have exactly the same name as a keyword. As an example, `D80` and `VMER` are keywords, while `D8e` and `abort` will be recognised as identifiers.

Keywords	Meaning
`CONST VAR PTR`	compiler commands
`VMER VMEW MOV JMP STOP`	DACOX instructions
`D80 D8E D16 D32`	data widths
`DH DL ENC RAM FIFO AL AH`	IB sources and destinations
`TH TL BERR ABOR PRGM`	jumping conditions

Table 10: Keywords

Expressions Expressions are composed of operators, numbers and identifiers and are evaluated according to the operator levels indicated in Table 9.

[20] A character is a letter if `isletter()` returns true. A character is a number if isdigit() returns true. This may vary from system to system.

[21] This feature does not work on the OS-9 platform, beacause `stcasecmp()` is not supported there.

Compiler commands There are three compiler commands `CONST`, `VAR` and `PTR`, which will be described in the next subparagraphs:

`CONST` The `CONST` compiler command sets the values of constants, which can be used in mathematical expressions. The value of a constant is determined after the first compilation pass and can not be changed later. Thus, referencing a constant anywhere in the code (even before the place where it is defined) will always yield the same result. If you assign a constant more than once, the compiler will print a warning message and the last value will be taken. The syntax of the `CONST` compiler command is shown in figure 24. As an example,

```
CONST A 1;
CONST A 2;
```

will produce an error and after assigning

```
CONST B A+1;
CONST A 1;
```

`A` will have the value 1 and `B` will be 2.

```
CONSTcmd  =  'CONST' identifier expression ';'
  VARcmd  =  'VAR' identifier expression ';'
  PTRcmd  =  'PTR' expression ';'
```

Figure 24: Compiler commands

`VAR` The `VAR` compiler command sets the values of variables, which can be used in mathematical expressions. The value of a variable is determined in pass two of the compiler at the point where the assignment is made. The used expression can only depend on constants and the variables that have been assigned before. Referencing a variable before its first definition will result in an error. A variable can be assigned more than once and its value will depend on the last assignment in the source code before the point where it is referenced. Note that you cannot create variables that have the same name as a constant. The syntax of the `VAR` compiler command is shown in figure 24. As an example,

```
VAR a b+1;
VAR b 1;
```

and

```
CONST A 1;
VAR A 2;
```

will both result in an error, while

```
VAR a 1;
VAR a a+1;
```

will not produce any error and `a` will be 2 afterwards.

`PTR` The `PTR` compiler command changes the memory pointer where the created binary code shall be written to. It is evaluated both in the first and second compilation pass at the point where it stands. For this reason, expressions defining the new value may only depend on constants that have been defined in the source code before this point. The `PTR` command can be used more than once, the memory pointer will be changed each time. The syntax of the `PTR` compiler command is shown in figure 24. As an example,

```
CONST B A;
PTR B;
CONST A 1;
```

will result in an error, because the value of `B` cannot be determined when the PTR command occurrs. Similarly,

```
VAR a 5;
PTR a+3;
```

will produce an error because `a+3` depends on a variable.

Labels When the compiler encounters an identifier at a place where it looks for an instruction or compiler command (i.e. between a semicolon and the next `CONST`, `MOV` ...), it is taken as a label definition. The current value of the memory pointer is assigned to a newly created constant with this name. Except of their creation, labels are identical to constants: They are assigned once during pass one and can not be changed later. `PTR` instructions can use them. Label definitions need not to be terminated by a semicolon. As an example,

```
label   JMP label;
```

will be an infinite loop, while

```
        PTR 0;
label   CONST label 2;
```

will result in a warning, because the value of 'label' is assigned twice.

Instructions The DACOX board understands four types of instructions, which will be explained in the following subparagraphs. These are the internal data transfers (`MOV`), the VME instructions (`VMER`, `VMEW`), the conditional and unconditional jumps (`JMP`) and the `STOP` instruction. All these instructions must be terminated with a semicolon.

```
MOV_instr  =  'MOV' src [dest] ['EOL'] ['EOE']
                 [expression [expression]] ';'
      src  =  'DL' | 'DH' | 'ENC' | expression
     dest  =  'DL' | 'DH' | 'ENC' | 'FIFO'
                 | 'RAM' | 'AL' | 'AH'
```

Figure 25: The `MOV` instruction

`MOV` The internal data transfers bring data from an IB source to an IB destination. The source is given just after the `MOV` keyword. Valid sources are `DL`, `DH` (the data bus low and high word input register), `ENC` (the current value of the event number counter), or an expression, whose value will be stored in the RAM.

Immediately after the source, the destination may be specified, which is one of `DL`, `DH` (the data bus low and high word output register), `ENC` (the event number counter), `FIFO` (the FIFO), `RAM` (the following RAM location) or `AL`, `AH` (the address bus low and high word output register). If no destination is given, only the test bit registers will be loaded. If the destination is equal to the source, a warning will be displayed.

The optional destination may be followed by none, one or both of the keywords `EOE` and `EOL`. These indicate to add an end_of_event or end_of_label mark to the data transferred. This has an effect only if the destination of the transfer is the FIFO. Otherwise, a warning will be printed.

Next, up to two expressions may be specified, that determine, which bits of the data shall be loaded into the bit test registers. Note that only one of bits 0 to 7 and one of bits 8 to 15 can be specified. Otherwise, the compiler will give a warning.

The instruction must be terminated with a semicolon.

As an example,

```
MOV DL FIFO EOE 0 8;
```

will transfer the low word of the data bus input register to the FIFO, adding an end_of_event mark and loading the bits 0 and 8 into the bit test registers.

```
MOV 12 ENC
```

will set the event number counter to 12. On the other hand side,

```
MOV DL DL EOE 0 0;
```

will produce 3 warnings.

`JMP` The `JMP` instruction tells DACOX to continue execution at another memory location. Its address is given just after the `JMP` keyword. Programmers are encouraged to use labels (see C.3) for specifying these addresses[22]. After the address, certain conditions

[22] Addresses are word addresses, i.e. the numbers that you see in the compiler output must be divided by 2.

```
JMP_instr  =  'JMP' expression [cond[cond...]]  ';'
     cond  =  ('PRGM' expression)
              | ('BERR')
              | ('TH' expression)
              | ('TL' expression)
              | ('ABOR')
```

Figure 26: the JMP instruction

may be specified. Only if all given conditions are met, the jump is executed, otherwise execution continues on the next memory location. Valid conditions are:

`PRGM` **expression** It is tested, if the specified expression is equal to the current program (i.e. the number of the last trigger).

`BERR` It is tested if the last VME cycle terminated with a bus error.

`TL` **expression** It is tested, if the lower test bit has the same value as expression.

`TH` **expression** Same for the higher test bit.

`ABOR` Test if the abort flag is set.

The instruction must be terminated with a semicolon.
As an example,

```
JMP 100;
```

will always jump to (word-)address 100, while

```
JMP 100 PRGM 0;
```

will only jump, if the board was triggerd via the TRIG0 lemo cable or its CSR equivalent.

```
 VME_instr  =  ('VMER'|'VMEW') [dwidth] [expression] ';'
    dwidth  =  ('D8') | ('D16') | ('D32')
STOP_instr  =  'STOP' ';'
```

Figure 27: the VMER, VMEW and STOP instructions

`VMER` and `VMEW` The `VMER` and `VMEW` instructions execute VME cycles. An optional data width can be given just after the `VMER` or `VMEW` keyword. Valid data widths are `D8E`, `D8O`, `D16` and `D32`. If you specify no data width, the compiler will use the width of the

VME cycle, that came just before this one *in the source code.* [23] [24] After the optional data width, the address modifier (AM) may be specified. If it is left out, the address modifier of the VME cycle that came just before this one *in the source code* is used.

The instruction must be terminated with a semicolon.

As an example,

```
VMEW D16 0x39;
VMER;
```

will first do a `D16` write with address modifier 0x39 and then read back again with the same address modifier and data width.

`STOP` The `STOP` instruction does nothing except stopping program execution. The board is ready to accept new triggers. Don't forget to terminate the instruction with a semicolon.

C.4 Output file

The output file shows the binary code to be downloaded on the left hand side of the page. Separated from it by a semicolon, on the right hand side a copy of the source file is given. This helps finding binary code that belongs to a specific source line. Error messages are printed in the output file on a line of their own. This way the origin of an error can be found quickly. A sample output file is shown below:

C.5 Error messages

Error messages are printed in the output file where they ocurred and can be divided into four groups:

error group	meaning	effect
critical error	compilation can't continue	compilation stops immediately
syntax error	error in input file syntax	compilation stops after current pass
unknown error	error of unknown origin	
internal error invalid error	errors in the compiler program	
warning	warn the user	compilation doesn't stop

Below, we give possible reasons for the diffrent messages:

[23] Be careful with this feature when you jump to another place. Better always specify data widths and address modifiers (AM).

[24] Keep in mind, that the compiler can't know if you wrote an even or odd address to the address bus output registers. So, if you make several `D8EO` cycles, always specify `D8E` or `D8O`.

Critical errors can be caused by invalid command line options, missing files, insufficient write permissions, lack of memory, errors during the different compilation stages and unresolvable constants.

Syntax errors can be caused by violations of the language syntax. Most of the time, the compiler tells you what it expects, e.g. a semicolon, a brace, an identifier, a value and so on.

Unknown errors are errors where the reason was not clear when the message was printed. They may be caused by earlier errors that explain the reason more clearly. Recursive variable definitions cause unknown error messages as well.

Warnings are issued if the compiler suspects that what you typed is not what you expect. An example might be the one given in the paragraph **Numbers** on page 52 or if you specify two testbits within the low data word.

C.6 Examples

The following examples shall illuminate the use of some compiler features.

Constants Consider the source that is given in figure 28. Here a module is to be read

```
                                        # definitions
        CONST BASE_ADDR 0x110000 ;      # the modules' base
        CONST CSR BASE_ADDR+0x800 ;     # the ctrl & status reg
        CONST DATA BASE_ADDR+0 ;        # the data address
                                        # readout
loop    MOV (CSR>>16) & 0xFFFF AH ;     # CSR address
        MOV CSR & 0xFFFF AL ;           #
        VMER D16 0x39 ;                 # read
        MOV DL 15 7 ;                   # Test bit 15
        JMP loop TH 1 ;                 # set ?
        MOV ((DATA>>16) & 0xFFFF) AH ;# no, DATA address
        MOV (DATA & 0xFFFF) AL ;        #
        VMER D16 0x39 ;                 # read
        MOV DL FIFO ;                   # store
        STOP ;                          # done
```

Figure 28: Example: constants

out, that has a status register and a data register, both at a given offset to the module's base address. By defining BASE_ADDR to be the base address of the module and defining CSR and DATA relative to it, the source becomes more readable. Additionally, the program can be reconfigured quickly if the module is moved to another base address. By changing the definition of BASE_ADDR, all the lines where BASE_ADDR, CSR or DATA are used will be updated to the new base address.

Note that the loop label is also handled by the compiler. The jump instruction will always jump to the correct place, even if the address in memory changed because you added some code.

Variables For the board has no internal registers or counters that can be used for programming, things like 'for'-loops are impossible. A crude way of workaround for a loop that writes the numbers 0 to 3 into the FIFO is shown in figure 29. First, the loop's body was written, and then copied several times. It doesn't have to be changed because we use the (compile-time) variable i.

```
VAR i 0 ;
MOV i FIFO ;                # 0
VAR i i+1 ;
MOV i FIFO ;                # 1
VAR i i+1 ;
MOV i FIFO ;                # 2
VAR i i+1 ;
MOV i FIFO ;                # 3
STOP ;
```

Figure 29: Example: variables

D Literaturverzeichnis

Literatur

[1] U. Faschingbauer et al.: Study of Electron Pair Production in Hadron and Nuclear Collisions at the CERN SPS, Proposal to the SPSC, CERN SPSC/88-25/P237 and SPSC/88-40/P237/Add 1

[2] P. Rehak et al.: Study of Electron Pair Production in Hadron and Nuclear Collisions at the CERN SPS, Addendum II, CERN/SPSLC 92-48 SPSLC/P237 Add 2

[3] H. Kraner et al.: Study of Electron Pair and Photon Production in Lead-Lead Collisions at the CERN SPS, Proposal to the SPSLC, CERN/SPSLC 94-01 SPSLC/P280

[4] H. Kraner et al.: CERES/NA45 Status Report to the SPSLC, CERN/SPSLC 94-02 SPSLC/M529

[5] P. Holl et al.: CERES/NA45 Report to Cogne 1995, CERN/SPSLC 95-20, SPSLC/M542

[6] P. Holl et al.: Study of Electron Pair and Photon Production in Lead-Lead Collisions at the CERN SPS, Addendum, CERN SPSLC/96–35/P280/Add 1

[7] R. Baur: Dissertation; Universität Heidelberg (1994)

[8] R. Baur, J. Gläß and R. Männer: First Experiences with a Systolic Trigger Processor for RICH Detectors; Computing in High Energy Physics, Tokyo 1991, Eds. I. Wanatabe and F. Abe, Frontier Science Series 3 (1991) 211–218

[9] VITA: The VMEbus specification; (1987)

[10] Wade Peterson: The VMEbus Handbook; 3rd Edition, (1993)

[11] Lattice Semiconductor Corp.: Lattice Data Book, (1994)

Index der Signale

Index

Erklärung:

Ich versichere, daß ich diese Arbeit selbständig verfaßt habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 12. November 1996

....

Christian Jung