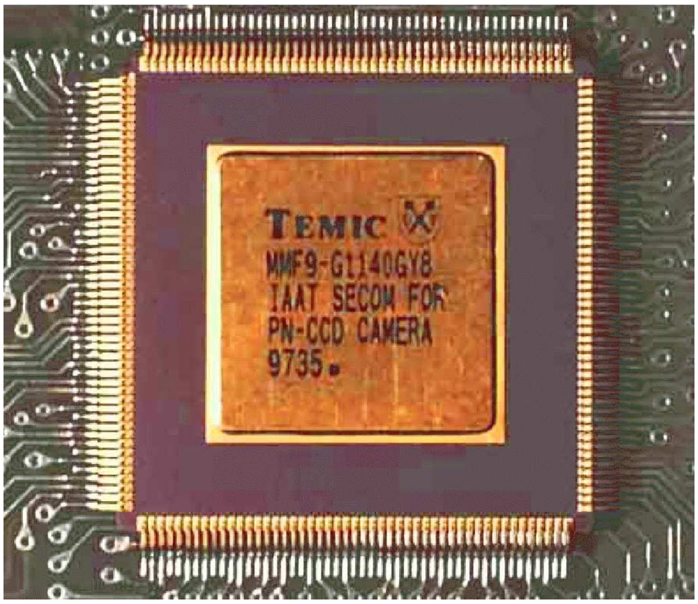
Entwicklung und Test eines Event-Pre-Prozessors für einen CdZnTe-Detektor

Diplomarbeit
von
Thomas Schanz



März 2001

Entwicklung und Test eines Event-Pre-Prozessors für einen CdZnTe-Detektor

Diplomarbeit
von
Thomas Schanz

Universität Tübingen
Institut für Astronomie und Astrophysik
Abteilung Astronomie

März 2001

Einleitung

Die Entwicklung von Weltraum-Hardware hat Tradition am Tübinger Institut für Astronomie und Astrophysik (IAAT). Vorallem die Röntgenastronomie-Gruppe um Professor Staubert beschäftigt sich schon seit vielen Jahren sehr erfolgreich mit der Entwicklung und dem Betrieb von Flug-Hardware für den Röntgen- und Gammabereich. In internationalen Kollaborationen hat das IAAT an zahlreichen Missionen mitgearbeitet. Populäre Beispiele in diesem Zusammenhang sind der deutsche Röntgensatellit ROSAT oder das erst kürzlich gestartete Röntgen-Observatorium XMM -Newton der ESA[1]. In Fortsetzung dieser Tradition entwickelt das Institut derzeit Komponenten für das neue *Gamma-Ray*-Observatorium INTEGRAL, das 2002 starten soll, und arbeitet in einem unabhängigen Projekt mit dem *Center of Astronomy and Space Science* (CASS) der *University of California San Diego* an der Entwicklung eines neuen, universell einsetzbaren *Gamma-Ray Burst*-Monitors. Ein *Burst*-Monitor dient der Aufspürung sogenannter *Gamma-Ray Bursts*, hochenergetischer kosmischer Energieausbrüche, für die die Röntgenastronomen sich schon seit langem interessieren. Die Entwicklung des *Burst*-Monitors ergab sich ursprünglich im Rahmen eines *Proposals* für das geplante *Gamma-Ray Large Area Space Telescope* der NASA. Obwohl sich die NASA mittlerweile für einen anderen *Burst*-Monitor entschieden hat, wird das Projekt von CASS und IAAT weiterverfolgt und zu gegebenem Zeitpunkt erneut für eine Röntgenmission vorgeschlagen.

Ziel dieser Diplomarbeit war die Entwicklung eines speziellen Ereignis-Prozessors für die schnelle An-Bord-Signalverarbeitung des geplanten *Gamma-Ray Burst*-Monitors. Die vorliegende Diplomarbeit stellt dabei den Beginn der Entwicklung an diesem Projekt in Tübingen dar. Zunächst war ein Gesamtkonzept für den Aufbau des Prozessors zu erstellen. Danach wurde mit der Realisierung der ersten konkreten Aufgaben begonnen, wobei diese nach Priorität geordnet wurden. Der Prozessor-Entwurf wurde deshalb mit der Einheit zur Signalkorrektur begonnen und anschließend stufenweise erweitert. Der bisherige Entwurf erfolgte ausschließlich in einer Hardware-Beschreibungssprache, um teure Hardware-Designzyklen zu vermeiden. Die bisher erstellten Designs sind aus diesem Grund rein formale Beschreibungen des Prozessors. Sie sind ähnlich einem Computerprogramm aufgebaut und ermöglichen auch den Test der Hardware durch Simulation direkt am Rechner.

Intention der vorliegenden Ausarbeitung ist neben der Dokumentation, auch die schrittweise Heranführung an die vielseitige Materie des *Burst*-Monitor Projektes. Die Arbeit gliedert sich in sechs Kapitel. Die beiden ersten Kapitel geben einen Überblick über die bisherigen Bemühungen der *Gamma-Ray Burst*-Forschung und über den Aufbau des geplanten *Burst*-Monitors. Kapitel 3 versucht eine knappe Einführung in die Hardware-Beschreibungssprache VHDL, die als wichtigstes Werkzeug für die derzeitige Entwicklung anzusehen ist. In Kapitel 4 werden die Aufgaben des Prozessors und der Prozessor selbst beschrieben. Schließlich wird in Kapitel 5 der Umgang mit der Testumgebung vorgestellt und in Kapitel 6 der derzeitige Stand des Prozessor-Projektes zusammengefasst. Im Anhang befinden sich Schaltpläne der im Rahmen dieser Diplomarbeit fertiggestellten Prozessor-Komponenten sowie die zugehörigen VHDL-*Listings* (CD-ROM).

Wie sich im Rahmen dieser Ausarbeitung gezeigt hat, ist die verbale Beschreibung von Hardware ein

[1] Das Titelblatt zeigt einen ASIC-Bausteins von XMM, an dessen Entwicklung das IAAT mit beteiligt war.

außerordentlich umständliches und schwieriges Unterfangen. Nicht zuletzt deshalb werden in der Elektronik Schaltpläne oder Beschreibungssprachen verwendet. Um die Darstellung dennoch einigermaßen verständlich zu halten, wurde der Text mit zahlreichen Abbildungen versehen. Diese stammen zum Teil aus der Designphase des Prozessors oder aus dem *Proposal* und sind größtenteils in Englisch beschriftet. Sie wurden zum Teil zu Dokumentationszwecken angefertigt und für die Ausarbeitung wiederverwendet. Abbildungen die speziell zur Illustration der Ausarbeitung angefertigt wurden, sind dagegen in deutscher Sprache beschriftet.

Inhaltsverzeichnis

Kapitel 1

Röntgen-Astronomie und *Gamma-Ray Bursts* (GRBs)

1.1 Röntgen- und Gamma-Astronomie

In der optischen Astronomie sind es vor allem Linsen- und Spiegelsysteme, die zur Beobachtung des Himmels verwendet werden. Beide Prinzipien basieren auf den Welleneigenschaften des Lichts und können noch mit den Methoden der Strahlenoptik beschrieben werden. Das Linsenfernrohr geht auf Johannes Kepler[1] zurück, das Spiegelteleskop auf Isaac Newton[2]. Obwohl das Prinzip dieser Systeme seit Jahrhunderten unverändert ist, konnte ihre Leistungsfähigkeit in zuvor kaum vorstellbarem Maße gesteigert werden. Mit der Zeit gelang es, immer größere und damit auch lichtstärkere Teleskope zu bauen, wobei heute vor allem Spiegelteleskope zu den leistungsfähigsten Fernrohren der Welt zählen. All diese Systeme arbeiten im Bereich des sichtbaren Lichts, oder zumindest in nahe angrenzenden Bereichen wie Ultra-Violett oder Infrarot. Strahlung anderer Wellenlängen kann mit diesen Systemen nicht detektiert werden. Dies hat vor allem zwei Gründe:

- Die Erdatmosphäre können nur ganz bestimmte Bereiche des elektromagnetischen Spektrums durchdringen.
- optische Standard-Abbildungsverfahren funktionieren dort nicht.

1.1.1 Die Beobachtungsfenster der Erdatmosphäre

Die Erdatmosphäre stellt ein wirksames Filter für jegliche Art elektromagnetischer Strahlung dar, mit zwei Ausnahmen: das sichtbare Licht bei Wellenlängen von 400 nm bis 800 nm und der Bereich der Radiowellen zwischen 20 Metern und wenigen Millimetern. Eine Übersicht des Durchlaßbereichs der Erdatmosphäre gibt Abbildung 1.1. In den meisten Energie-Bereichen wird die Strahlung durch Ionisation innerhalb des Wasserdampfs und der Gasmoleküle der Atmosphäre blockiert. Aus diesem Grund gelangt Röntgen- und Gamma-Strahlung nicht bis zum Erdboden. Astronomische Beobachtungen in diesem Bereich sind deshalb erst ab einer Höhe von ca. 30 Kilometern möglich. In der Tat bedient sich die Röntgen- und Gamma-Astronomie hochfliegender Atmosphären-Ballons oder satellitengestützter Beobachtungssysteme. Erst durch die Entwicklung der Raumfahrt in der zweiten Hälfte des letzten Jahrhunderts konnte sich die Beobachtung astronomischer Röntgenquellen zu einem eigenen, sehr erfolgreichen Zweig der Astronomie entwickeln.

[1] Deutscher Astronom und Mathematiker, gilt als Begründer der geometrischen Optik. Theorie des optischen Fernrohrs 1610-1611.
[2] Englischer Physiker, Mathematiker und Astronom. Erstes Spiegelfernrohr Newtonscher Bauart 1668.

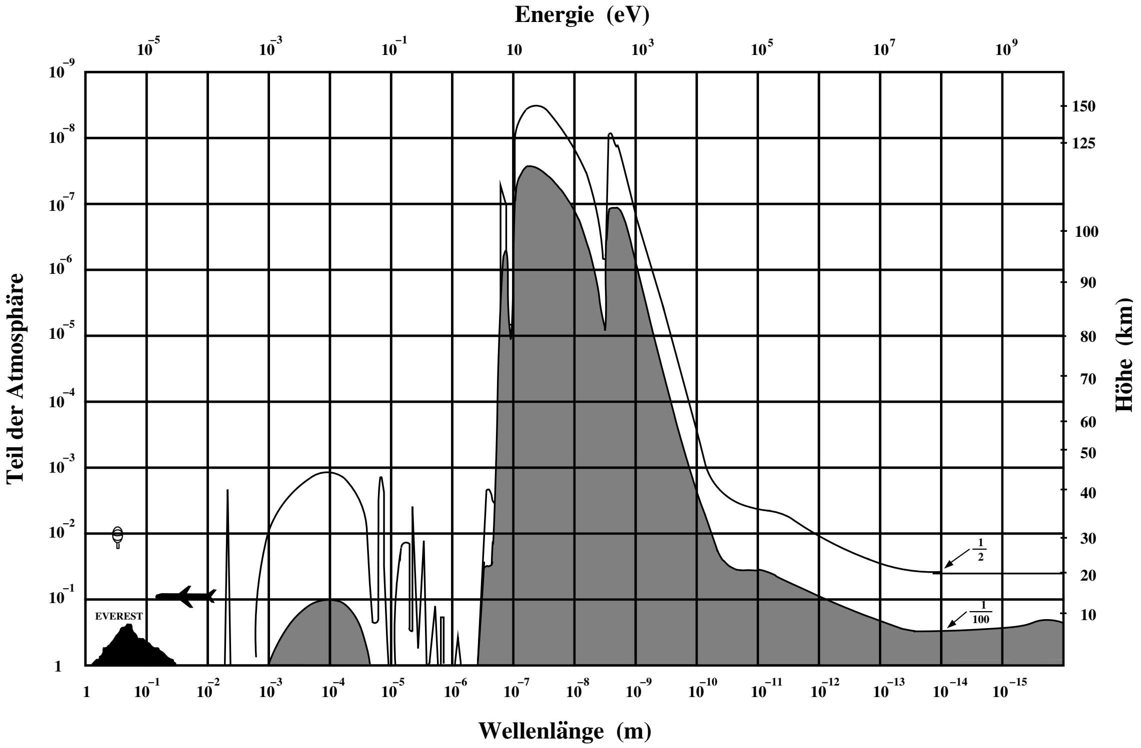


Abbildung 1.1: *Dämpfungsbereiche der Erdatmosphäre. Es ist der Rückgang der Intensitäten auf ein Hundertstel (grau unterlegt) und auf die Hälfte gezeigt. Deutlich erkennbar sind die beiden Fenster im Radio-Bereich (1 m bis 10^{-3} m) und im sichtbaren Bereich zwischen 10^{-6} und 10^{-7} Meter. Für die Röntgen- und Gamma-Astronomie ist der Bereich ab 10^{-11} Meter von Interesse. Die Abbildung stammt aus [Giacconi et al. 1968].*

1.1.2 Abbildungs-Verfahren

Das Prinzip optischer Linsen basiert auf Lichtbrechung, das von Spiegeln auf Resonanzerscheinungen in der Materie. Für elektromagnetische Strahlung außerhalb des sichtbaren Bereichs, insbesondere kurzwelligere, sind diese Phänomene nur teilweise oder überhaupt nicht ausnutzbar. So zeigt Röntgenstrahlung in keinem bekannten Material eine nennenswerte Brechung, was den Bau von Linsen bis heute verhindert hat. Spiegel besitzen nur für einen streifenden Einfall ausreichendes Reflexionsvermögen und stellen außerordentlich hohe Anforderungen an die Oberflächengüte. Der Bau großer satellitengestützter Röntgenteleskope[3] gelang deshalb erst innerhalb der letzten beiden Jahrzehnte. Für den Bereich harter Röntgen- und Gammastrahlung wird auch der Einsatz von Spiegelsystemen zunehmend problematischer. Die erforderliche Fokallänge der Spiegel nimmt mit zunehmender Energie zu. Das derzeitige Maximum erreicht das kürzlich gestartete europäische Röntgen-Observatorium XMM-Newton, mit einer Fokallänge von 7,5 Metern. XMM-Newton kann Röntgenstrahlen bis zu Energien von 15 keV abbilden. Für zukünftige Systeme mit noch höheren Energien, ist deshalb bereits die Trennung des Satelliten in zwei separat fliegende Einheiten geplant. Solche Missionen bestehen dann quasi aus zwei Satelliten: Einem großen Wolter-Teleskop, und einem kleinen Fokal-Satelliten. Der Fokal-Satellit fliegt separat, und wird durch Steuerelemente ständig im Fokus des Teleskops gehalten. Mit solch einem System, wie es sich derzeit in Planung befindet, könnten Röntgenstrahlen bis zu Energien von ca. 50 keV abgebildet werden.

[3] Hierbei handelt es sich in der Regel um Wolter-Teleskope. Ein Wolter-Teleskop besteht aus ringförmig angeordneten Paraboloid- und Hyperboloid-Spiegeln. Die Röntgenstrahlen reflektieren zunächst am Paraboloid, danach am Hyperboloid, sie müssen dabei fast streifend auf die Spiegeloberflächen einfallen damit eine Reflexion zustandekommt. Teleskope dieser Bauart sind z.B. auf ROSAT, Chandra oder XMM-Newton eingesetzt worden.

Für Röntgenstrahlen noch höherer Energien, oder für Gamma-Strahlen (ab 500 keV) sind auf absehbare Zeit keine direkt abbildenden Spiegelsysteme in Sicht. Die Gamma-Astronomie ist daher auf andere, indirekte Verfahren angewiesen.

Kollimator-Teleskope:

Eine Möglichkeit, auch bei hohen Röntgen-Energien Bilder des Himmels zu erhalten, besteht in der Kollimator Methode. Der Himmel wird hierbei mit einem nicht ortsauflösenden Detektor abgetastet. Das Gesichtsfeld des Detektors bestimmt dabei die Auflösung. In der Regel wird das Gesichtsfeld unter Verwendung eines geeigneten Kollimators eingeschränkt, dieser kann z.B. in Form eines Tubus realisiert werden. Ein Vorteil der Kollimator-Technik ist, daß kein ortsauflösender Detektor benötigt wird. Bei Verwendung von sogenannten Rotations-Kollimatoren lässt sich die Auflösung bis in den Bogenminuten-Bereich steigern. Ein Nachteil der Kollimator-Technik ist jedoch, daß das Teleskop ständig *Scan*-Bewegungen ausführen muß, was vor allem für eine *pointierte* Beobachtung außerordentlich hinderlich ist.

Compton-Teleskope:

Reagiert ein Gamma-Quant durch Compton-Streuung innerhalb eines Detektorvolumens, so kann dessen Einfallswinkel aus den Wechselwirkungsorten und -energien rekonstruiert werden. Im einfachsten Fall zweier Wechselwirkungen liegen die möglichen Einfallswinkel auf einem Konus. Der Öffnungswinkel des Konus hängt dabei von den beiden absorbierten Energiemengen ab, die Konusachse liegt auf der Trajektorie des gestreuten Quants. Anhand der Überschneidung solcher Konen lassen sich Gamma-Quellen am Himmel lokalisieren [Dean 1984].

Eine neuere Idee, dasselbe Prinzip anzuwenden, ist das *Liquid-Xenon-Telescope*. Bei diesem werden die Wechselwirkungen von Gamma-Strahlung in flüssigem Xenon gemessen, es funktioniert dabei ähnlich wie eine Blasenkammer [Aprile et al. 1994].

Luftschauer-Teleskope:

Eine andere Möglichkeit zur Beobachtung vor allem harter Gamma-Strahlung ($>10^{11}$ eV), ist die Verwendung von Luftschauer-Teleskopen. Bei dieser Art von Teleskop kann die Trajektorie der Photonen in der Atmosphäre anhand der längs ihrer Bahn freigesetzten Cerenkov-Strahlung festgestellt werden [Gerthsen et al. 1982]. Aus einer Schar solcher Trajektorien kann die Position der Quelle am Himmel abgeleitet werden. Der erste Nachweis einer Röntgen-Quelle (Crab) mittels eines Luftschauer-Teleskops gelang 1989 mit einem Experiment am Whipple Observatorium [Weekes et al. 1989].

***Coded-Mask*-Teleskope:**

Die bis heute vielleicht vielversprechendste Technik zur Untersuchung von Röntgen-Quellen ist die indirekte Abbildung durch sogenannte *Coded-Mask*-Teleskope (CM-Teleskope). Ein CM-Teleskop besteht aus einem ortsauflösenden Detektor und einer Apertur mit kodierter Maskengeometrie. Röntgen-Quellen innerhalb des Gesichtsfeldes des Teleskops projizieren die Geometrie der Maske auf den Detektor. Mit Hilfe mathematischer Verfahren kann aus der Detektorinformation die Position der Quelle am Himmel berechnet, und eine Abbildung rekonstruiert werden. Das erste CM-Teleskop mit kodierter Apertur war die *Multiplex-Pin-hole-Camera* (MPC) der *University of Chicago*, die 1972 an Bord einer Rakete unter Leitung von R.L. Blake [Blake et al. 1974] geflogen ist. Die Apertur bestand aus einem Edelstahlblech, in das - an zufällig ausgewählten Kreuzungspunkten eines quadratischen Rasters aus 61 x 61 Linien - runde Löcher gestanzt waren, so daß insgesamt etwa ein Achtel der Fläche durchlocht war.

Bis heute konnte die CM-Technik bei zahlreichen Ballon-Experimenten ihre Leistungsfähigkeit beweisen. Die erste Satelliten-Mission, die auf die CM-Technik in Verbindung mit Halbleiterdetektoren setzt, ist das ESA Satellitenprojekt INTEGRAL[4], das im Jahr 2002 starten soll, und an dessen Hardware-Entwicklung das IAAT[5] der Universität Tübingen ebenfalls beteiligt ist.

Die vorliegende Diplomarbeit beschäftigt sich mit Entwurf, Entwicklung und Test eines *Hardware Event Prozessors* im Rahmen des *Coded-Mask Imager* -Projektes der *University of California San Diego* (UCSD) und des IAAT. Der *Coded-Mask Imager* soll vor allem als *Gamma-Ray Burst*-Monitor zum Einsatz kommen.

1.2 *Gamma-Ray Bursts*

Gamma-Ray Bursts (GRBs) sind kurzzeitige Strahlungsausbrüche sehr hoher Intensität im hochenergetischen Röntgen-Bereich. Ihr Auftreten ist bis heute in Zeit und Ort unvorhersagbar. Die typische Lebensdauer der *Bursts* beträgt wenige Minuten, oft auch nur Sekunden [Fishman 1999]. Sie wird zum Teil von einem mehrtägigen Nachleuchten bei Wellenlängen niedererer Energien begleitet. Die stärksten bisher gemessenen *Bursts* zeigen Strahlungsflüsse in der Größenordnung von 10^{-4} erg cm^{-2}, genug, um meßbare Störungen in den oberen Atmosphärenschichten der Erde zu verursachen [Fishman et al. 1988]. Der Großteil der Strahlung wird im Bereich zwischen 100 keV und 1 MeV ausgesendet, das Spektrum zeigt in der Regel ein strukturloses, glattes Kontinuum. Die Verteilung der GRBs am Himmel ist annähernd sphärisch, was zu zahlreichen Spekulationen über den Ursprung Anlass gab.

Die präzise Lokalisierung der GRBs am Himmel ist seit der Entdeckung vor 30 Jahren ein ungelöstes Problem der Astronomie, und das obwohl bis heute mehrere Tausend GRBs registriert worden sind. Die Beobachtung ist sehr problematisch. Das Auftreten von GRBs ist zum einen unvorhersagbar, zum anderen verhindert die kurze Lebensdauer oft eine rechtzeitige Erfassung mit leistungsfähigen Teleskopen. Die Problematik ist in mancher Hinsicht mit der Fotografie von Blitzen inmitten eines Gewitters vergleichbar.

Die wenig zufriedenstellende Situation bei der Beobachtung führte zu einer Flut von Veröffentlichungen und wurde von mehr als 100 Theorien über den Ursprung der GRBs begleitet. Erst seit 1997 kam die Beobachtungstechnik so weit voran, daß erstmals die Positionsbestimmung von *Gamma-Ray Bursts* bis in den Bogenminutenbereich gelang, und die Aufzeichnung mancher *Bursts* von Beginn der Entstehung an verfolgt werden konnte. Als einer der größten Durchbrüche auf diesem Gebiet gilt die Untersuchung von GRB 990123, die erstmals gleichzeitig bei Röntgen-Energien und im optischen Bereich durchgeführt werden konnte. Solche Beobachtungen spielen für die Röntgen-Astronomie eine herausragende Rolle, denn erst mit der eindeutigen Identifizierung eines optischen Gegenstücks kann die Entfernung der *Burst*-Quellen präzise bestimmt (mittels Rotverschiebung optischer Spektrallinien) und der Entstehungsprozess der Röntgen-Strahlung vielleicht geklärt werden.

1.2.1 Evolution der GRB-Forschung

Die Entdeckung der GRBs geschah zufällig im Jahre 1969 bei der Auswertung geheimer Beobachtungsdaten der *Vela*-Satelliten-Reihe [Klebesadel et al. 1973]. Die Vela-Satelliten waren ein System des amerikanischen Militärs zur Überwachung von Atomwaffen-Teststop-Verträgen der damaligen Supermächte. Um die gesamte Erdoberfläche zu überwachen, wurden jeweils zwei gleiche Satelliten auf den gegenüberliegenden Seiten eines kreisrunden Orbits mit einem Durchmesser von 250000 km angeordnet. Die Vela-Satelliten waren mit Detektoren für Röntgen- und Gammastrahlung sowie mit Meßgeräten zur Aufspürung von Neutronen und EMPs[6] ausgerüstet.

[4]INTEGRAL: *International Gamma Ray Astrophysics Laboratory.* ist eine Kollaboration zwischen Europa, den USA und Rußland. Das IAAT beteiligt sich durch die Entwicklung des *Hardware-Event-Processor for IBIS (HEPI)* des hochauflösenden Gammastrahlen-Teleskops (IBIS) an INTEGRAL

[5]IAAT: Institut für Astronomie und Astrophysik der Universität Tübingen, Abteilung Astronomie.

[6]EMP: Electro Magnetic Pulse

Die Vela Serie wurde ständig verbessert und war von 1963 bis Mitte der 80'er Jahre im Einsatz. Die Meßgeräte sollten Vertragsverletzungen durch oberirdische Atomwaffentests überwachen. Im Laufe dieser geheimen Mission zeichneten die Satelliten jedoch gelegentlich auch Röntgen-Ereignisse kosmischen Ursprungs auf. Die genaue Auswertung der Satellitendaten brachte 1969 erste Hinweise auf eine kosmische, kurzzeitig sehr starke, Aktivität im harten Röntgen-Bereich. Das erste Ereignis ist auf den 2. Juli 1967 datiert und stellt vermutlich die erste Messung eines *Gamma-Ray Bursts* dar [Klebesadel et al. 1973]. Die Veröffentlichung der Daten geschah erst 1973 [Bonnell & Klebesadel 1996], siehe auch Abbildung 1.3.

Mitte der achtziger Jahre wurde eine zweite Klasse von GRBs entdeckt, die sich vom Großteil der *Bursts* deutlich unterschied [Laros et al. 1986]. Diese *Bursts* besitzen ein flaches, thermisches Bremsstrahlungs Spektrum (kT etwa 30 keV), eine kurze Lebensdauer, und stammen aus gut bestimmbaren Regionen des Himmels. Die Quellen, sogenannten *Soft Gamma-Ray Repeaters* (SGR) sind Objekte unserer eigenen Galaxis und der Großen Magellanschen Wolke, die gelegentlich Soft-GRBs aussenden [Hartmann 1995, Kouveliotou 1995, Hurley 1999].

Die Fortschritte der GRB-Forschung in den neunziger Jahren sind vor allem auf zwei Missionen zurückzuführen: dem Start des *Compton Gamma Ray Observatory* (CGRO) der NASA [Fishman 1981, Fishman et al. 1993] und dem italienischen Röntgen- und Gammastrahlen-Satelliten BeppoSAX [Frontera 1998, Boella et al. 1997, Costa et al. 1998]. Das CGRO wurde 1991 gestartet und war das erste große *Gamma-Ray* Observatorium der NASA. An Bord befanden sich vier Experimente zur Untersuchung des Röntgen- und Gamma-Himmels im Energiebereich zwischen 30 keV und 30 GeV. Für die GRB-Forschung von besonderer Bedeutung war das *Burst and Transient Source Experiment* (BATSE), ein *Burst* Monitor für die Suche nach *Gamma-Ray Bursts* und deren Untersuchung im Energiebereich zwischen 20-1000 keV. Tatsächlich registrierte BATSE in den 9 Jahren Betriebszeit von CGRO mehr als 2700 GRBs![7] Abbildung 1.2 illustriert das Ergebnis der *Burst*-Suche von BATSE. Sehr schön ist die gleichmäßige Verteilung der GRBs über den gesamten Himmel zu erkennen.

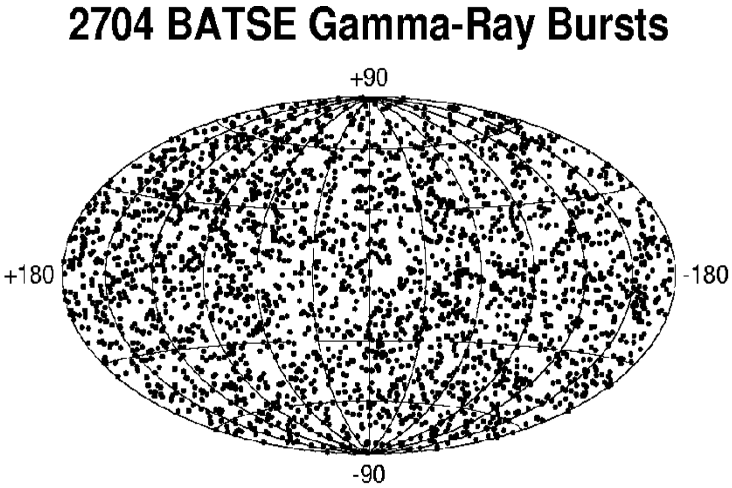


Abbildung 1.2: *Das Ergebnis der 9 jährigen Suche von BATSE/CGRO nach Gamma-Ray Bursts. Gut ist die relativ sphärische Verteilung der GRBs über den gesamten Röntgenhimmel zu erkennen [Meegan et al. 1992].*

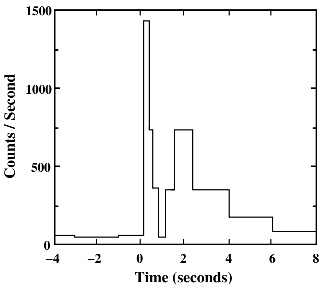


Abbildung 1.3: *Der vermutlich erste jemals gemessene GRB, ist das von den Satelliten Vela3, Vela4a,b registrierte Ereignis vom 2 Juli 1967. Aus den Messungen konnte der zeitliche Verlauf des Ereignisses rekonstruiert werden. Die Lichtkurve wurde im Scientific American, Oct.76 veröffentlicht.*

[7] Die Mission von CGRO endete am 4. Juni 2000 mit dessen geplanter Zerstörung beim kontrollierten Wiedereintritt in die Erdatmosphäre.

Erst durch die Beobachtungen mit BATSE in den frühen 90'ern wurde offensichtlich, daß die Quellen der GRBs entweder im galaktischen Halo oder sogar bei kosmologischen Entfernungen zu suchen waren [Meegan et al. 1992]. Verglichen mit den Fortschritten in den Jahren zuvor, war dies ein gewaltiger Durchbruch. Dennoch blieb die tatsächliche Entfernung der *Burst*-Quellen unbekannt. Sie lag entweder im Bereich von kpc im galaktischen Halo oder bei kosmologischen Entfernungen in der Größenordnung von Gpc [Mao & Paczynski 1992]. Aufgrund der unbekannten Entfernung blieb zunächst auch die tatsächliche Helligkeit der GRBs eine Unbekannte. Am vielversprechendsten für die Entscheidung dieser Frage schien entweder eine möglichst präzise Bestimmung der Isotropie der Winkelverteilung [Briggs et al. 1996, Lamb 1997, Hakkila et al. 1994, Tegmark et al. 1996] oder die Suche nach GRBs beziehungsweise deren Gegenstück im Optischen- und Röntgen-Bereich innerhalb des galaktischen Halo der Andromeda-Galaxie M31 [Klose 1995, Lamb 1995, Li et al. 1992]. Für die letztere ließ sich bislang kein Hinweis finden, und für das Erste lieferte die Auswertung der BATSE-Untersuchungen im Laufe der Zeit immer deutlicher zu Tage, daß GRBs ihren Ursprung wohl nicht im galaktischen Halo haben, sondern entferntere Phänomene sind.

Im Laufe der Zeit gab es zahlreiche Vorschläge zur Bestimmung der Entfernungsskala und zur Identifizierung der GRB-Quellen.

So könnte im Röntgen-Energieband z.B. die spektrale Untersuchung der Zusammenhänge zwischen zeitlicher Entwicklung der Lichtkurven, und der beobachteten Helligkeitsverteilung, kosmologische Merkmale zu Tage liefern [Brainerd 1994, Fenimore et al. 1995, Horack et al. 1996, Norris et al. 1995, Lee et al. 1197]. Andere Ansätze sind die Analyse der spektralen Energieverteilung der *Bursts* im Zusammenhang mit Photon-Photon-Paarbildung und der Strahlungsundurchlässigkeit innerhalb von GRB-Photon-Feldern [Baring et al. 1998] oder Abschätzungen der Energiemenge von Neutronensternen als potentielle GRB-Quellen [Hartmann 1996]. Bei Röntgen-Energien müßte sich ein galaktischer Ursprung der *Bursts* durch Streuung und Absorption des Röntgenlichts im interstellaren Mediums verraten [Klose 1994, Klose 1995, Owens et al. 1995]. Für Beobachtungen im Optischen- und Infrarot-Bereich, sowie bei Radio-Energien bietet sich die Suche nach *Burst*-Quellen innerhalb der *Error-Box* des *Burst*-Monitors[8] an. Hierzu werden Aufnahmen der gleichen Himmelsregion in diesen Bereichen untersucht und nach potentiellen *Burst*-Quellen durchmustert. Nach theoretischen Voraussagen sollte in der Nähe einer GRB-Quelle der Nachweis von Transienten gelingen, vorausgesetzt die Folgebeobachtungen geschehen unmittelbar nach einem *Burst-Trigger*.

Ein großes Problem für die Identifizierung der *Burst*-Quellen ist die Größe der *Error-Box* eines *Burst*-Monitors. Die 1σ *Error-Box* von BATSE hat beispielsweise einen Radius von wenigen Grad [Briggs et al. 1999]. Dies ist für ein *Gamma-Ray*-Teleskop kein schlechter Wert, für Folgeuntersuchungen im Optischen, Infrarot- oder Radio-Band aber ein gewaltiges Areal. Die Verwendung mehrerer Detektoren verschiedener Satelliten-Missionen verbesserte die Situation etwas. Trotzdem lag die Größenordnung der *Error-Box* selbst dann noch im Bereich von ein paar zehntel Grad. Für eine Beobachtung im optischen Bereich ist dies, aus astronomischer Sicht, eine unzureichende Positionsbestimmung. Betrachtet man die durchschnittliche Sterndichte unserer eigenen Galaxie und projiziert diese auf eine *Error-Box* von einem Grad Durchmesser, so erhält man ca. $10^{11}/40000 \sim 2 \text{ x } 10^6$ mögliche *Burst*-Quellen [Postnov 1999]. In etwa derselben Größenordnung liegt die Zahl für die beobachtbaren Galaxien. Aus diesem Grund ist wenig verwunderlich, daß die Identifizierung von *Burst*-Quellen auf diese Weise wenig erfolgreich war. In der Tat konnte damit bis Mitte der 90'er Jahre keine einzige Quelle eines *Gamma-Ray Bursts* zweifelsfrei identifiziert werden.

Am 30. April 1996 startete BeppoSAX, ein Projekt der *Italian Space Agency*. BeppoSAX ist ein Röntgenobservatorium für den Bereich von 0,1 keV bis 300 keV. An Bord befinden sich eine *Wide Field Camera* für den Bereich von 2-30 keV und mehrere *Narrow field* Instrumente, darunter zwei Spektrometer

[8]Das Auflösungsvermögen eines jeden Teleskops bzgl. der Genauigkeit, mit der die Position eines Objekts am Himmel bestimmt werden kann, ist begrenzt und wird durch die sogenannte *Error-Box* angegeben. Dabei handelt es sich um das Himmels-Segment, in dem sich die Quelle befindet, und das mit seinem Durchmesser die Unsicherheit bei der Winkelauflösung wiedergibt.

für den Bereich von 0,1 bis 10 keV, ein Szintillations-Zähler (4-120 keV) und das *Phoswich Detection System* (PDS) (15-300 keV). Die aktive Abschirmung des PDS dient als *Burst*-Monitor für den Bereich von 60 bis 600 keV. Der *Burst*-Monitor auf BeppoSAX bietet eine *Error-Box* in der Größenordnung von Bogenminuten, die Daten stehen binnen einer Stunde nach dem *Burst-Trigger* zur Verfügung. Leider ist das Gesichtsfeld des *Burst*-Monitors relativ klein, so daß nur mit 10 Ereignissen pro Jahr gerechnet wird.

Der erste *Burst*, den BeppoSAX registriert hat, war 1997 GRB 970111. Die *Error-Box* hatte einen Radius von 10 Bogenminuten und stand binnen einer Stunde via *Electronic-Mail* anderen Observatorien zur Verfügung [Costa et al. 1997]. Trotzdem gelang es nicht, irgend ein Nachleuchten des GRB im Optischen [Galama et al. 1997] oder im Radio-Band [Frail et al. 1997] nachzuweisen. Der Durchbruch gelang jedoch bald darauf mit GRB 970228 [Costa et al. 1997b]. Für diesen konnte zum ersten Mal bei Folgeuntersuchungen ein Nachleuchten im optischen Bereich zweifelsfrei identifiziert werden [Groot et al. 1997, van Paradijs et al. 1997], was eine Positionbestimmung mit sehr hoher Winkelpräzision ermöglicht hat. Die Multispektraluntersuchung von GRBs ist in der darauf folgenden Zeit mehr und mehr zu einer gängigen Praxis geworden. Sie hat die Kenntnis über Herkunft und Ablauf der *Gamma-Ray Bursts* deutlich vorangebracht.

Einen anderer Meilenstein für die GRB-Forschung stellt die Einrichtung eines hochentwickelten elektronischen Netzwerks für GRB-Nachrichten dar, das *GRB-Coordinated-Network* (GCN). Es koppelt alle wichtigen GRB-Observatorien und -Forschungseinrichtungen der Welt und stellt die Information über die Entdeckung neuer GRBs mittlerweile in Sekundenschnelle für Folgeuntersuchungen zur Verfügung[9].

1.2.2 GRB 990123

Eine wirkliche Sensation war die Entdeckung von GRB 990123 Ende Januar 1999. GRB 990123 war der erste GRB, der gleichzeitig im optischen Band und bei Röntgen-Energien untersucht werden konnte. Das Signal war so stark, daß alle Röntgen- und Gamma-Detektoren auf CGRO[10], und ebenfalls der *Gamma-Ray Burst*-Monitor von BeppoSAX ansprachen.

Der *Burst* hatte zwei deutliche Spitzen in seinem zeitlichen Verlauf und eine komplexe Struktur im Millisekunden-Bereich, seine Dauer betrug etwa 100 Sekunden. Während des *Bursts* wurde das *Gamma-Ray* Spektrum allmählich weicher. Die Flußdichte der Energie innerhalb des Bereichs von 20 keV - 300 MeV war etwa 3 x 10^{-4} erg cm^{-2} GRB 990123 gehörte damit zu den stärksten *Bursts*, die BATSE je registriert hat. Gleichzeitig beobachtete die *Wide-Field Camera* (WFC) auf BeppoSAX im Röntgenband (1,5 - 26 keV) eine Flußdichte von etwa 7 x 10^{-6} erg cm^{-2}, mehrere Prozent der Strahlungsstärke bei Gamma-Energien.

Wirklich spektakulär an GRB 990123 war jedoch die gleichzeitige Entdeckung und Beobachtung eines hellen Nachleuchtens im optischen Band mit einer stellaren Helligkeit von etwa 9^m, während des Maximums. Zum ersten Mal in der Geschichte der GRB-Forschung war es damit gelungen, gleichzeitig mit dem GRB eine optische Ausstrahlung zu beobachten (Abb. 1.4). Möglich wurde dies durch den schnellen Informationsfluss innerhalb des GCN-Netzwerks, das unmittelbar nach Beginn des GRBs andere Observatorien in Kenntnis setzte. Eines dieser Observatorien war das *Robotic Optical Transient Search Experiment* (ROTSE) des *Los-Alamos National Laboratory*. Es war unmittelbar in der Lage, das optische Nachleuchten aufzuzeichnen.

ROTSE besteht aus vier optischen Kameras mit einem Durchmesser von 11,1 cm. Das Gesichtsfeld beträgt dabei mehrere Grad, die Aufnahmen geschehen unter Verwendung von CCD Detektor-*Arrays*. Nach der Alarmierung durch das GCN fuhr ROTSE zu den angegebenen Himmelskoordinaten und konnte die Untersuchung 22,18 Sekunden nach dem GRB-*Trigger* aufnehmen [Postnov 1999]. Die Kameras von ROTSE nahmen alle 25 Sekunden ein Bild mit einer Belichtungszeit von 5 Sekunden auf und machten bis etwa zehn Minuten nach Beginn des GRB auch Aufnahmen mit 75 Sekunden Belichtungszeit. Bemerkenswert ist, daß die maximale Helligkeit von 8^m,95 ($\sim$ 1 Jy)[11] erst etwa 50 Sekunden nach Beginn des GRB er-

[9] Das GCN ist auch über das Internet unter *http://gcn.gsfc.nasa.gov/gcn/* erreichbar.

[10] Die Gamma-Detektoren von CGRO sind der Transienten Monitor BATSE (20-1000 keV), das Szintillations-Spektrometer OSSE (0,05-30 MeV) und das *Gamma-Ray*-Teleskop EGRET (30 MeV - 10 GeV)

[11] 1 Jy = 10^{-26} W m^{-2} Hz^{-1} = 10^{-19} erg m^{-2}.

reicht wurde, die Flußdichte im optischen betrug $\sim$ 2,5 x 10^{-7} erg cm^{-2}. Die von ROTSE aufgenommene Lichtkurve ist, zusammen mit der von BATSE im Röntgen-Energieband gemessenen, in Abbildung 1.4 dargestellt.

Die Beobachtung der Region von GRB 990123 mit großen optischen Teleskopen, die 3h 46 min nach dem *Burst-Trigger* begann, enthüllte eine punktförmige Lichtquelle 18. Größenklasse, die in der Helligkeit langsam abklang. Das optische Spektrum der Quelle zeigte deutliche Absorptionslinien mit einer Rotverschiebung von z $\approx$ 1,6. Zwei Wochen später, nachdem die Helligkeit bereits unter V $\simeq 25^m$ abgesunken war, fand das *Hubble Space Telescope* (HST) an dieser Stelle eine schwache Galaxie ungewöhnlicher Form [Fruchter et al. 1999]. Die Lichtquelle (vgl. Abb.1.6) befand sich 0,6 Bogensekunden außerhalb des galaktischen Zentrums; wahrscheinlich handelt es sich um zwei Galaxien, die miteinander in Wechselwirkung stehen.

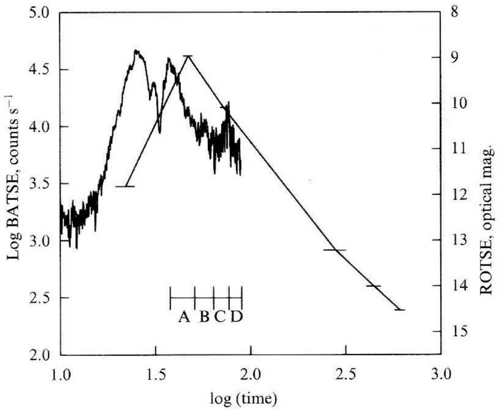


Abbildung 1.4: *Die Lichtkurven von BATSE (50-300 keV) und ROTSE (im sichtbaren Bereich, in stellaren Helligkeitsklassen), als Funktion der logarithmierten Zeit (in Sekunden), nach dem GRB-Trigger von BATSE #7343. Die horizontalen Linien in der Lichtkurve von ROTSE geben die Belichtungszeit wieder. Die Abbildung stammt aus [Fenimore et al. 1999].*

Das Nachleuchten im Röntgenbereich war ebenfalls außergewöhnlich stark. Sechs Stunden nach dem *Burst* zeigte die Röntgenuntersuchung eine Flußdichte von 1,1 x 10^{-11} erg cm^{-2} (ca. 0,8 μJy) und ein langsames Abklingen der Intensität. Infrarotmessungen der Quelle zeigten ein ähnliches Verhalten. Im Radio-Band wurde das Nachleuchten mit dem VLA[12] untersucht. Die Beobachtung etwa einen Tag nach dem *Burst*, ergab ein Signal von 260 $\pm$ 32 μJy bei 8,46 GHz. Abbildung 1.5 zeigt die Lichtkurven von GRB 990123, die CGRO aufgezeichnet hat, in unterschiedlichen Energie-Bändern. Die Lichtkurven zeigen, daß die niederenergetische Emission länger anhält als diejenige bei hohen Energien. Diese spektrale Entwicklung von harter Strahlung zu weicher ist typisch für den Verlauf von *Gamma-Ray Bursts*.

Bei einer Rotverschiebung von z = 1,6 und unter der Annahme isotroper Ausstrahlung, betrug die freigesetzte Energie im Röntgen- und Gamma-Bereich (20 - 700 keV) von GRB 990123 den unglaublichen Wert von $\sim$ 3 x 10^{54} erg. Die maximale Leuchtkraft im Röntgen-Energieband betrug $\sim$ 6 x 10^{53} erg s^{-1}. Selbst das innerhalb des optischen Bereichs gemessene Signal von 1 Jy bedeutet bei einer solchen Distanz eine isotrope Leuchtkraft von $\sim 10^{51}$ erg s^{-1}, wesentlich mehr als die höchste je gemessene optische Leuchtkraft bei Supernova-Explosionen ($\sim 10^{45}$ erg s^{-1}) [Postnov 1999].

Die Untersuchung von *Gamma-Ray Bursts* wie GRB 990123 kann vorübergehend wie folgt zusammengefasst werden:

- Kosmische GRBs sind *Bursts* im harten Röntgenbereich mit typischen Energien von 30 keV bis 100 MeV, ihre Lebensdauer beträgt wenige Sekunden (1 bis 100 s). Die Lichtkurve zeigt ein komplexes zeitliches Verhalten mit kleinen Schwankungen im Bereich von $\sim$ 1 ms. Das Spektrum von GRBs

[12] VLA: Very-Large-Array, Zusammenschluß von Radio-Teleskopen.

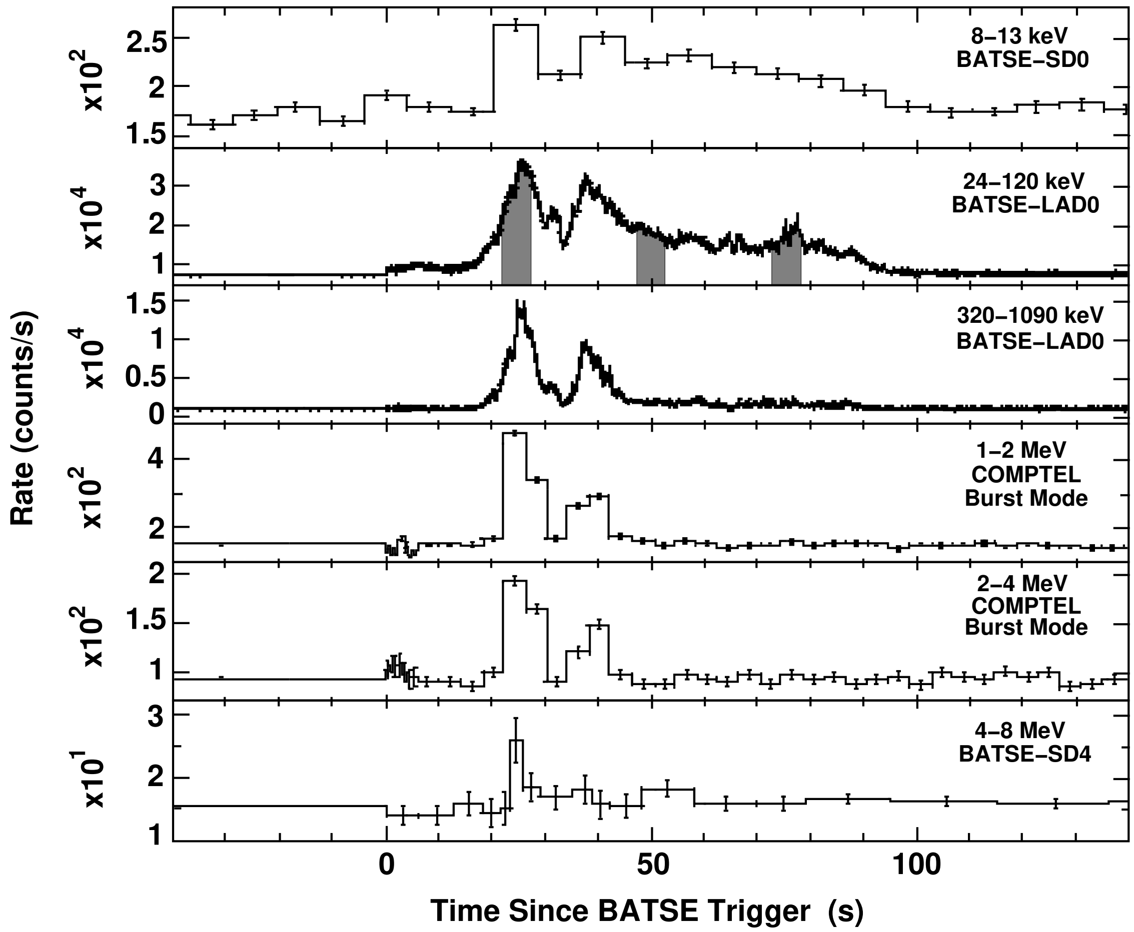


Abbildung 1.5: *Der zeitliche Verlauf von GRB 990123. Die Lichtkurven wurden von verschiedenen Meßgeräten auf CGRO nach dem BATSE-Trigger aufgezeichnet. Die Intervalle, in denen die ROTSE-Kameras aktiv waren, sind schattiert eingezeichnet. Gut erkennbar ist, daß die Lichtkurven bei hohen Energien schneller abfallen als bei niederen [Briggs et al. 1999].*

ist nicht-thermisch, zeitlich variabel und besitzt sein Emissionsmaximum bei Energien zwischen 0,1 - 1 MeV. Bei sehr hellen GRBs lassen sich selbst Photonen bis 1 GeV beobachten. Die Flußdichte der schwächsten meßbaren GRBs beträgt S $\sim 10^{-7}$ erg cm^{-2}, die der stärksten, S $\sim 10^{-4}$ erg cm^{-2}.

- Die Detektionsrate von GRBs beträgt $\sim$ 1 pro Tag, bei einer Nachweisgrenze von $\sim 10^{-7}$ erg cm^{-2}. Die Winkelverteilung über den Röntgenhimmel ist isotrop.

- Das Nachleuchten der GRBs im Röntgen-Band zeigt ein Abklingverhalten mit einer Gesetzmäßigkeit von $\sim t^{-1,3\pm0,1}$. In etwa der Hälfte der Fälle klingt die Strahlung im optischen Band bis etwa 200 Tage nach dem *Burst* mit einem konstanten Faktor ab. Manchmal werden auch variable Radio-Emissionen gemessen.

- Für manche GRBs ließ sich die Quelle in anderen Galaxien nachweisen. In fünf Fällen konnte die Rotverschiebung zwischen $0{,}7 < z < 3{,}14$ bestimmt werden. Die Rotverschiebung von GRB 990123 wurde mit Hilfe von Absorptionslinien innerhalb der optischen Emission gemessen. Bei einer Rotverschiebung von z = 1,6 und im Falle einer isotropen Ausstrahlung bedeutet dies für GRB 990123 eine totale Energie-Freisetzung von $\Delta E_\gamma \sim 3 \times 10^{54}$ erg. Dies liegt in der Größenordnung der Ruhemasse unserer Sonne ($M_\odot c^2 = 2 \times 10^{54}$ erg).

- In einem anderen Fall, bei GRB 980425 wurde an der Stelle des *Bursts* zeitgleich eine Supernova im Abstand von 40 Mpc entdeckt (SN 1998bw). Dies ist möglicherweise kein Zufall.
- Im Zusammenhang mit GRBs ließ sich bis heute keine Verstärkung durch Gravitationslinsen nachweisen.

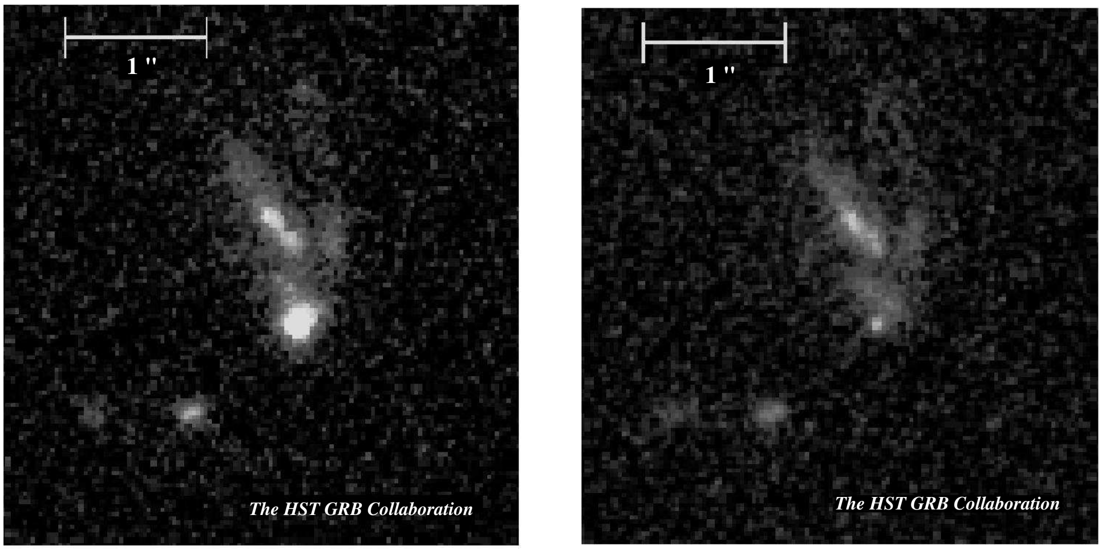


Abbildung 1.6: *Aufnahme des optischen Nachleuchtens und der Ursprungs-Galaxie von GRB 990123. Die linke Aufnahme entstand im Februar 1999, die rechte Aufnahme im März 1999. Deutlich erkennbar ist das Abklingen der Strahlungsintensität in der unteren Bildhälfte [Fruchter et al. 1999].*

1.2.3 Ursprung der *Gamma-Ray Bursts*

Die Natur der Energiequelle von *Gamma-Ray Bursts* ist bis heute eine offene Frage. Beobachtungen wie die von GRB 990123 implizieren Energien bis zu 10^{54} erg, eine isotrope Abstrahlung vorausgesetzt. Als Quellen existieren zwei mögliche Kandidaten. Eng benachbarte, miteinander verschmelzende Neutronensterne [Eichler et al. 1989] und Supernovae extrem massereicher Sterne [Wheeler 1999]. Trotzdem vermögen eigentlich selbst diese Quellen Energien von 10^{54} erg nicht zu erzeugen. Für einen Ausweg aus dieser '*Energie-Krise*' wurden zwei Möglichkeiten vorgeschlagen: Gravitationslinsen als Verstärker und die Idee einer nicht-isotropen Energieabstrahlung.

Die Gravitationslinsen-Hypothese geht davon aus, daß sich zumindest bei den stärksten GRBs, zwischen Quelle und Beobachter (also der Erde) ein massives Objekt (ein Stern oder ein Schwarzes Loch) befindet, das als Gravitationslinse wirkt und die scheinbare Helligkeit des GRB verstärkt. Eine solche Gravitationslinse würde die Ausstrahlung der GRB-Quelle auf der Erde also stärker erscheinen lassen, als sie in Wirklichkeit ist. Bis heute wurden allerdings bei keiner Beobachtung an GRBs jemals Anzeichen für die Wirkung von Gravitationslinsen gefunden.

Wesentlich wahrscheinlicher, wenn auch schwerer zu erklären, ist die Hypothese, daß die Ausstrahlung von GRBs nicht isotrop erfolgt, sondern nur in einem engen Winkelbereich, ähnlich einem *Jet*. Modelle sagen für einen solchen Fall das mögliche Auftreten von Pausen in der Lichtkurve der Nachleuchtphase voraus [Sari et al. 1999]. Dies geschieht, wenn die Intensität des *Jets* soweit abgenommen hat, daß der Beobachter auf den Rand des *Jets* sieht. Je kleiner der Öffnungswinkel des *Jet*, desto eher die Unterbrechung in der Lichtkurve. Den ersten Hinweis auf einen Effekt dieser Art ergab die Beobachtung von GRB 990123, als die Lichtkurve etwa zwei Tage nach dem *Burst* plötzlich abfiel [Castro-Tirado et al. 1999, Kulkarni et al. 1999]. Ein zweites Mal konnte das Verhalten bei GRB 990510 beobachtet werden. Eine bemerkenswerte Konsequenz dieser Entdeckung ist, daß möglicherweise sehr viel mehr GRBs existieren könnten als bisher angenommen wurde. Wenn die *Jet*-Theorie zutrifft, würden diese GRBs zwar ein Nachleuchten im Röntgen-, Optischen- oder Radio-Band erzeugen, wären aber aufgrund des für den Erdbeobachter unsichtbaren *Gamma-Ray Burst* nicht entdeckt worden. Mit der *Jet*-Theorie wird die Energie der GRB-Quelle zwar um den Faktor 10-1000 reduziert (abhängig vom Öffnungswinkel des *Jets*, der zugrundegelegt wird), dafür muß aber im gleichen Verhältnis die Anzahl der GRBs im Weltall steigen, da in diesem Fall eben auch nur für jeden 10'ten-1000'sten GRB ein *Burst* auf der Erde beobachtet wird.

Untersuchungen an Lichtkurven und Spektren von GRBs geben Hinweise auf den lokalen Entstehungsprozess möglicher *Jets*. So deuten schnelle Oszillationen im Millisekunden-Bereich, wie sie in den Lichtkurven vieler GRBs im Röntgenband beobachtet wurden (siehe z.B. Abb.1.5) darauf hin, daß die Strahlung der *Bursts* aus einer relativ kleinen Region, mit einem Radius von $R_e \sim 10^7$ cm [Postnov 1999], stammt. Ein dermaßen kleines Objekt wirft jedoch Probleme auf, denn Photonen könnten eine solch kompakte Region gar nicht ohne weiteres verlassen. Bei der erforderlichen hohen Dichte hochenergetischer Photonen würde die Region durch Photon-Photon-Paarbildung nämlich buchstäblich strahlungsundurchlässig. Dieses *Compactness*-Problem kann allerdings mit der Annahme umgangen werden, daß sich die GRB-Quelle zumindest entlang der Beobachtungslinie mit einer relativistischen Geschwindigkeit bewegt [Krolik et al. 1991]. Basierend auf dieser Überlegung sagt ein gängiges Modell relativistisch bewegte Feuerbälle, mit Lorentz-Faktoren[13] von $\Gamma \sim$ 100-1000, als GRB-Quellen voraus. Photon-Photon-Paarbildung hindert die Strahlung am Entweichen aus der Region und formt so den Feuerball. Das beobachtbare zeitabhängige Verhalten der Lichtkurve entsteht dann durch Kollisionen verschiedener Schockwellen mit unterschiedlichen Lorentz-Faktoren (internes Schockmodell). Diese emittieren Synchrotron-Strahlung und Strahlung durch inversen Compton-Effekt. Einige Wissenschaftler [Paczynski et al. 1993] vermuten hingegen, daß die Schockwelle innerhalb der umgebenden interstellaren Materie das Nachleuchten verursacht (externes Schockmodell). Die Feuerball-Theorie paßt bereits ganz gut zu den Beobachtungsdaten [Wijers et al. 1997, Waxman 1997], bedarf aber vermutlich noch einiger Modifikationen.

Eine Modifikation des Modells könnte wie folgt aussehen: Wenn ein massereicher Stern in einer Supernova explodiert, so kollabiert ein Teil seiner Masse zu einem sehr kompakten stellaren Objekt (einem Neutronenstern oder Schwarzen-Loch). Der andere Teil wird als Plasma in einer Schockwelle nach außen geschleudert. Einer neueren Idee zufolge, könnte ein kleiner Teil des Plasmas binnen Tagen auf das kompakte Objekt zurückstürzen und um dieses eine Akkretionsscheibe bilden. Falls dies zutrifft, könnten Instabilitäten innerhalb der Scheibe diese zum Kollabieren bringen und Plasmakugeln wie Geschosse nach außen schleudern (ähnlich wie bei Mikroquasaren). Diese Feuerbälle hätten relativistische Geschwindigkeiten und würden bei der Kollision mit der Schockwelle der Supernova eine stark kollimierte Strahlung aussenden, die durch den starken Dopplereffekt bis zu γ-Energien verschoben wäre. Die Feuerbälle würden sich daraufhin innerhalb der interstellaren Materie langsam abbremsen und diese bis zu Energien der kosmischen Strahlung beschleunigen. Die dabei erzeugte Synchrotronstrahlung könnte das Nachleuchten erklären, das auf der Erde registriert wird. Das Modell gibt eine gute Erklärung für das Energiespektrum, für die zeitliche Entwicklung der γ-Pulse, sowie für das Nachleuchten der GRBs [Dar 2001].

Die Frage nach der Energiequelle ist dennoch umstritten. Der Zusammenhang mit Supernova Explosionen liegt nahe, es existieren jedoch auch andere Möglichkeiten, wie z.B. die Verschmelzung zweier Neutronensterne oder eines Sterns mit einem Schwarzen-Loch. Sicher scheint nur, daß in jedem Fall als Folge der GRB-Explosion ein Schwarzes-Loch entsteht. Für *Gamma-Ray Burst* GRB 980425 konnte innerhalb der *Error-Box* von BeppoSAX eine Supernova (SN 1998bw) nachgewiesen werden. Bei SN 1998bw handelt es sich um eine anomale Supernova vom Typ 1c. Die Galaxie, in der sich SN 1998bw befindet, hat eine Rotverschiebung von 0,0085 [Tinney et al. 1998], was einer Entfernung von 38 Mpc (H_0 = 65 km s^{-1} Mpc^{-1}) entspricht. Der Kollaps eines massiven stellaren Eisenkerns zu einem Schwarzen Loch könn-

[13] Lorentz-Faktor: $\Gamma = [1 - (v/c)^2]^{-1/2}$

te die Beobachtung von GRB 980425 erklären [Iwamoto et al. 1998, Woosley et al. 1999]. Das stärkste Indiz für den Zusammenhang zwischen *Gamma-Ray Bursts* und Supernovae stammt aber aus der kürzlichen Entdeckung, daß die Lichtkurven von GRB 970228, GRB 980326 und möglicherweise GRB 990712 gegen Ende eine Beule haben, die bisher im Rahmen der GRB-Theorien nicht erklärt werden konnte. Dabei könnte es sich jedoch um die rotverschobenen Lichtkurven von Supernovae handeln, die den Lichtkurven der GRBs überlagert sind [Bloom et al. 1999, Dar 1999, Galama et al. 1999, Hjorth et al. 1999, Reichart 1999]. Sollte sich diese Vermutung bestätigen und *Gamma-Ray Bursts* eine direkte Folge von Supernova-Explosionen sein, dann hätte die GRB-Forschung bis heute bereits mehr Supernovae entdeckt, als alle anderen Supernova-Kampagnen zusammen.

Kapitel 2

Der *Coded-Mask Imager* (CMI)

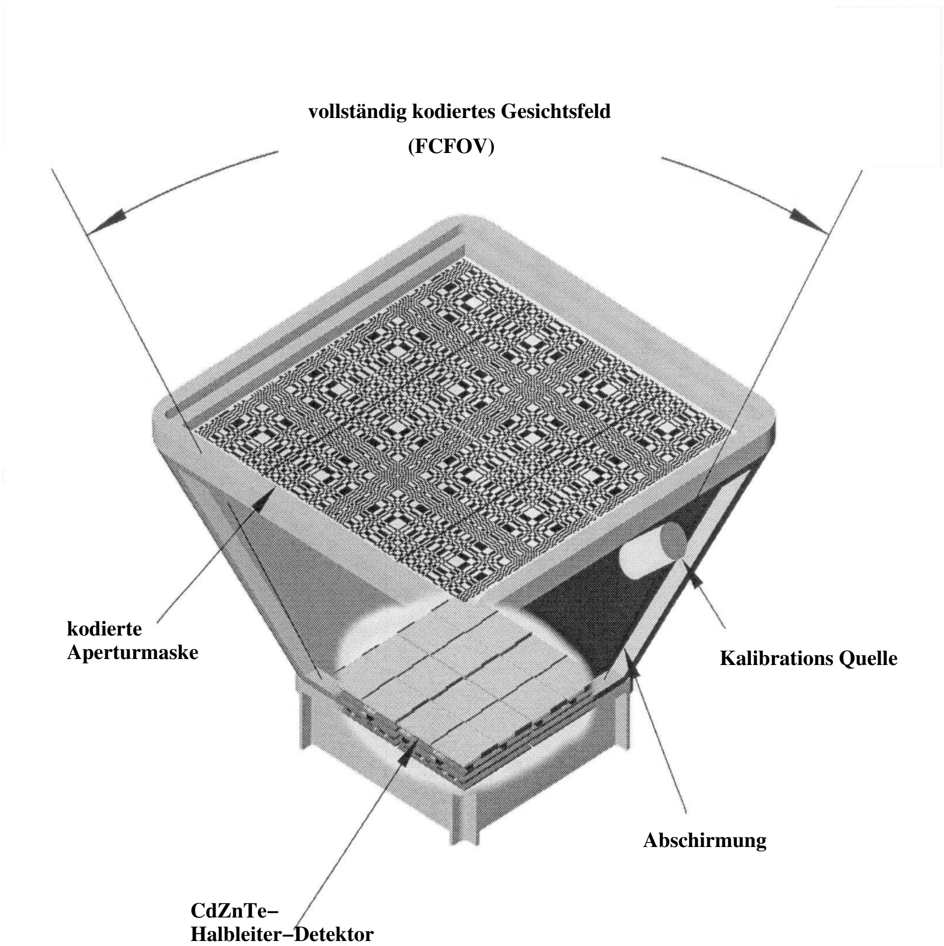


Abbildung 2.1: *Abbildung des CMI. Der CMI besteht aus einer kodierten Aperturmaske und einem ortsauflösenden Detektor. Gamma-Quellen innerhalb des FCFOV projizieren ein Bild der Aperturmaske auf den Detektor [Rothschild et al. 1999].*

Die Entwicklung des *Coded-Mask Imagers* (CMI) geht ursprünglich auf ein gemeinsames *Proposal* [Rothschild et al. 1999] der *University of California San Diego* und der Universität Tübingen an die NASA zurück. Inhalt des *Proposal* war die Entwicklung vier identischer CMI als Beitrag des von der NASA geplanten *Gamma-Ray Large Area Space Telescope* (GLAST). GLAST soll den Himmel nach Gamma-Quellen im Energiebereich zwischen 10 MeV und 100 GeV absuchen und die Arbeit des sehr erfolgreichen CGRO-Satelliten fortsetzen. Sein Start ist für das Jahr 2005 vorgesehen. Die vier CM-Kameras hätten GLAST als *Burst-Monitore* gedient und den Beobachtungsbereich um Energien zwischen 10 keV bis 200 keV ergänzt.

Da sich die NASA für einen anderen *Burst*-Monitor entschieden hat, soll CMI nun als eigenständiges Projekt zwischen den Röntgenastronomie-Gruppen der *University of California San Diego* (UCSD) und dem Institut für Astronomie und Astrophysik (IAAT) der Universität Tübingen realisiert werden. In dieser Zusammenarbeit wird die Hochenergie-Astronomie Gruppe von CASS[1] @ UCSD die CMI-Kamera, den Detektor und einen Teil der Elektronik beisteuern. Die Röntgen-Astronomie Gruppe des IAAT wird sich mit dem in dieser Arbeit beschriebenen *Event-Pre-Processor* (EPP) sowie einem Teil der *Data-Processing-Unit* (DPU) beteiligen.

Die vorliegende Arbeit stellt im Rahmen des CMI-Projektes den Anfang der Hardware Entwicklung in Tübingen dar.

2.1 Wissenschaftliche Ziele des Projektes

Bis zum heutigen Tag gibt es für das CMI-Projekt keine konkrete wissenschaftliche Mission. Aus diesem Grund existieren auch keine konkreten wissenschaftlichen Ziele. Dennoch gibt es eine klare Ausrichtung des CMI-Projektes in Richtung *Gamma-Ray Burst* Untersuchungen und für einen Einsatz als *Gamma-Ray Burst* Monitor.

2.1.1 Die Suche nach *Gamma-Ray Bursts*

Das Rätsel der *Gamma-Ray Bursts* (GRBs) gehört bestimmt zu den spannendsten Fragen in der modernen Astronomie. Obwohl schon vor über 30 Jahren entdeckt, konnte das Phänomen bis zum heutigen Tag nicht vollständig geklärt werden. Die jüngsten Fortschritte sind vor allem der Multispektralanalyse mehrerer *Bursts* zu verdanken. Erst die gleichzeitige Beobachtung in verschiedenen Energie-Bereichen, wie z.B. im Gamma-, Röntgen- und Optischen-Band ergeben die nötigen Daten, um den Ursprung der Strahlung zu verstehen und das Rätsel ihrer Entstehung aufzuklären. Der CMI wäre eben aus diesem Grund eine ideale Ergänzung zu GLAST gewesen.

Für zukünftige Missionen an denen CMI teilnehmen könnte, wäre deshalb die Kombination mit weiteren Instrumenten und die gleichzeitige Untersuchung anderer Energie-Bänder ein erstrebenswerter Vorsatz. Aber auch wenn sich dieser Wunsch nicht erfüllen lässt, und CMI einst auf eine '*standalone*' Mission geht, könnten seine Beobachtungen die Entdeckung und Untersuchung neuer GRBs ermöglichen. Die sofortige Alarmierung des GCN[2] im Falle eines *Burst-Triggers* kann andere Observatorien mobilisieren und so die angestrebte gleichzeitige Untersuchung der Quellen in anderen Energie-Bändern ermöglichen. Dies ist vor allem deshalb eine Option, da bisherige Untersuchungen gezeigt haben, daß die Emissionskomponente im Gamma-Energieband zeitlich am kurzlebigsten ist. Beobachtungen in niederenergetischeren Bändern können deshalb auch noch Minuten oder Stunden nach dem *Burst-Trigger* erfolgen.

Der große Öffnungsbereich mit einem Gesichtsfeld von 50° x 50° und das gute Auflösungsvermögen von unter 32 Bogenminuten, bei einer Zeitauflösung von ca. 250 μs, machen den *Coded-Mask Imager* zu einem idealen Monitor für die Suche nach GRBs. Insbesondere die Zusammenschaltung mehrerer CMI würde das Gesichtsfeld auf nahezu den ganzen Himmel ausdehnen und eine lückenlose Überwachung ermöglichen. Die Detektor-Empfindlichkeit ist mit 9 mCrab pro Tag, bei einer Bandbreite von 10-100 keV, außerordentlich gut und läßt die Verwendung des CMI auch für die Beobachtung anderer Röntgenquellen zu. Der nutzbare Spektralbereich reicht sogar von 10-200 keV mit einer spektralen Auflösung von bis zu

[1]CASS: *Center of Astrophysics and Space Science*
[2]GCN: GRB Coordinated Network, Netzwerk zur Koordinierung der GRB-Forschung.

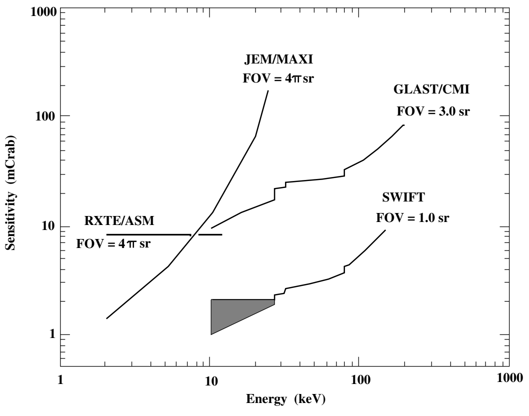


Abbildung 2.2: *Die spektrale Empfindlichkeit des CMI im Vergleich zu anderen Missionen. Es sind die Empfindlichkeit für eine Meßdauer von einem Tag für Quellen an der 5 σ-Grenze aufgetragen. Für das Energie-Intervall wurde ein Wert von $\Delta E/E$ = 0,5 angenommen [Rothschild et al. 1999]*

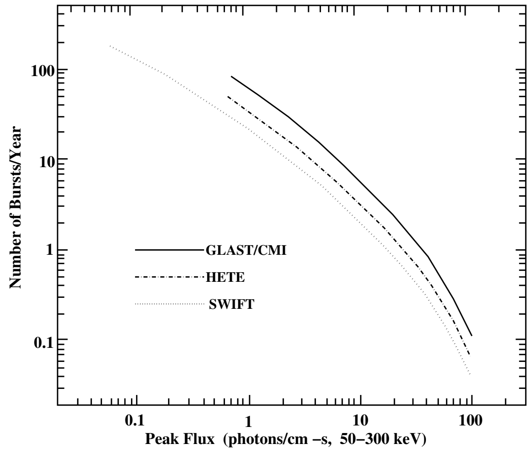


Abbildung 2.3: *Die Burst-Rate, die CMI pro Jahr entdecken würde, im Vergleich zu anderen Missionen. Die Grafik geht von einer Mission mit vier CMIs aus [Gehrels 1999].*

3 keV bei 60 keV. Die Empfindlichkeit von CMI im Vergleich mit konkurrierenden Missionen ist in Abbildung 2.2 dargestellt. CMI deckt vorallem den für die *Burst*-Suche wichtigen Energiebereich zwischen 10 und 200 keV gut ab. Für eine rasche Auswertung der Gamma-Ereignisse existiert ein schneller Ereignisprozessor, der binnen weniger Sekunden das Auftreten eines GRB registriert, und einen *Burst*-Alarm an die Bodenstation und das GCN aussendet. Um einen vollständigen Verlauf der Lichtkurve zu erhalten, existiert zusätzlich ein 2 MByte großer Ring-Speicher, der eine lückenlose Aufzeichnung des Ereignisses und der Vorgeschichte der GRBs garantiert.

In dieser Art ausgestattet, ist der CMI ein ideales Instrument für die Entdeckung neuer GRBs und die Unterstützung des *GRB Coordinated Network*.

2.1.2 Andere wissenschaftliche Einsatzgebiete

Obwohl für die Suche nach GRBs optimiert, kann der CMI auch zur Untersuchung anderer Röntgenquellen verwendet werden. Als mögliches Anwendungsgebiet bietet sich eine Himmelsdurchmusterung im harten Röntgenbereich an. Die Elektronik des CMI wird hierzu neben einem Modus zur pointierten Beobachtung auch die Möglichkeit des *Scan*-Betriebs bieten. Bei einer Durchmusterungs-Mission können Röntgen-Quellen die innerhalb des Energiebereichs von 10-200 keV und der Empfindlichkeit des CMI liegen vermessen und kartographiert werden. Aufgrund der großen Empfindlichkeit (bis zu 0.5 mCrab innerhalb von 5 Jahren) könnte eine derartige Mission 20 mal schwächere Quellen aufspüren, als dies bei der letzten Durchmusterung im harten Röntgenbereich (HEAO-1 A-4)[3] der Fall gewesen ist. Potentielle Untersuchungs-Objekte sind Supernovae, Schwarze Löcher, Galaxien mit röntgen-aktivem Kern (AGN)[4], oder binäre Sternsysteme mit einem starken Röntgen-Anteil aus nicht-thermischen Prozessen.

Der CMI bietet neben der Positionsbestimmung mit einer Genauigkeit von bis zu 12 Bogenminuten für starke Quellen (10 σ), die Möglichkeit einer detaillierten spektralen und zeitlichen Untersuchung der

[3] Die HEAO-1-Mission wurde 1977 gestartet und hatte die vollständige Kartographierung des Himmels im harten Röntgenbereich zum Ziel. Der Energiebereich von HEAO-1 lag zwischen 0,2 keV und 10 MeV.

[4] *AGN: Active Galactic Nuclei*; Galaxien mit Röntgen-aktivem Kernbereich.

Lichtkurven. Mit diesen Daten könnte der CMI für die Untersuchung hochenergetischer astronomischer Quellen und für ein besseres Verständnis der zugrundeliegenden Prozesse einen wichtigen Beitrag leisten.

2.2 Das *Coded-Mask* Teleskop

Ein *Coded-Mask Imager* (CMI)[5] besteht im wesentlichen aus einer Apertur und einem ortsauflösenden Detektorsystem (vergleiche hierzu Abbildung 2.1). Die Apertur besitzt eine Maske, die für Röntgenlicht teilweise opak und teilweise durchsichtig ist. Befindet sich eine Röntgenquelle innerhalb des Gesichtsfeldes der Kamera, so entsteht ein Schattenwurf der Maske auf dem Detektor. Die ortsauflösende Fähigkeit des Detektors kann dazu benutzt werden, ein Bild der Quelle und deren Position am Himmel zu berechnen. Zu diesem Zweck besitzt der CMI eine leistungsstarke Datenverarbeitungs-Elektronik. Die Elektronik übernimmt neben der Bilderzeugung auch zahlreiche andere Aufgaben. Sie ist für die Korrektur systematischer Detektorfehler genauso von Nutzen wie zur Unterdrückung von störendem Hintergrund, zur spektroskopischen Analyse der Quelle sowie zur Speicherung und Telemetrierung der Messdaten. In der Praxis stellt die Elektronik neben dem Detektorsystem den herausragenden Teil des Entwicklungsaufwandes dar. Ein Überblick über die Komponenten und ein Schnitt durch die Kamera ist in Abbildung 2.4 zu sehen.

Kameragehäuse und Apertursystem des CMI werden von CASS an der *University of California San Diego* entwickelt. Wichtige Parameter für eine CM-Kamera sind die Größe des Gesichtsfeldes, das Auflösungsvermögen und die Empfindlichkeit. Diese Größen werden maßgebend beeinflußt von der Geometrie des Aperturmusters, dessen Wiederholungsrate und Zellengröße, dem Abstand von Apertur zu Detektor, sowie von Pixelabstand und Pixelgröße des Detektors. Die Wahl der optimalen Parameter basiert auf langjährigen Erfahrungen mit der *Coded-Mask*-Technik und wurde unter Berücksichtigung des primären Projektzieles, der Suche nach GRBs, ausgesucht. Der CMI benötigt ein großes Gesichtsfeld, eine gute Hintergrundunterdrückung, sowie ausreichend hohe Detektorempfindlichkeit. Unter Verfolgung dieser Maßgaben wurden die einzelnen Komponenten des CMI gewählt und dimensioniert.

2.2.1 Die *Coded-Mask* Apertur

Die Aperturmaske besteht aus 0,6 mm dickem Wolfram und besitzt ein Quadrat aus sechzehn 101 x 101 zelligen MURAs[6]. Das verwendete MURA ist ein Muster mit einer Zellengröße von 1,3 mm. Es ist für Röntgenlicht zu 50% opak und zu 50% durchlässig. Das MURA wurde unter der Maßgabe minimaler Bildartefakte ausgewählt und im Abstand an die Detektorgröße angepasst. Der Detektor besitzt dementsprechend viele Segmente. Die Detektorgröße ist maßgebend für die Wahl der anderen Komponenten und bestimmt die Größe der gesamten CM-Kamera. Das Gesichtsfeld von 50° x 50° kann bei einem Abstand von 14 cm zwischen Detektor und Apertur erreicht werden. Die Wahl der Zellengröße der MURA mit 1,3 mm ist ein Kompromiss zwischen dem angestrebten Auflösungsvermögen von 10 Bogenminuten und einer ausreichenden Detektorempfindlichkeit. Abbildung 2.7 zeigt das Aussehen einer MURA, wie sie zum Test des Detektors verwendet wurde [Rothschild et al. 1999], und Abbildung 2.8 den Zusammenhang zwischen Empfindlichkeit und Auflösungsvermögen in Abhängigkeit von der Zellengröße.

Die *Coded-Mask*-Technik ist eine seit Jahren im Bereich der Astronomie harter Röntgenstrahlen etablierte Technologie. Ab Energien von etwa 15 keV ist die Verwendung von Spiegelsystemen mit streifendem Einfall nicht mehr möglich, wie sie im niederenergetischen Röntgenbereich sehr erfolgreich zum Einsatz kommen. Ab etwa 15 keV ist die Energie der Strahlung zu hoch, es kommt zu Wechselwirkungen mit der Materie der Spiegeloberfläche, und die Reflexionseigenschaften gehen verloren. Die CM-Technik kommt gänzlich ohne Spiegel oder reflektierende Schichten aus, sie benutzt als Effekt den Schattenwurf.

[5]Die Begriffe *Coded-Mask Imager* (CMI), CM-Kamera und CM-Teleskop werden hier synonym verwendet.

[6]MURA: Modified Uniformly Redundant Array

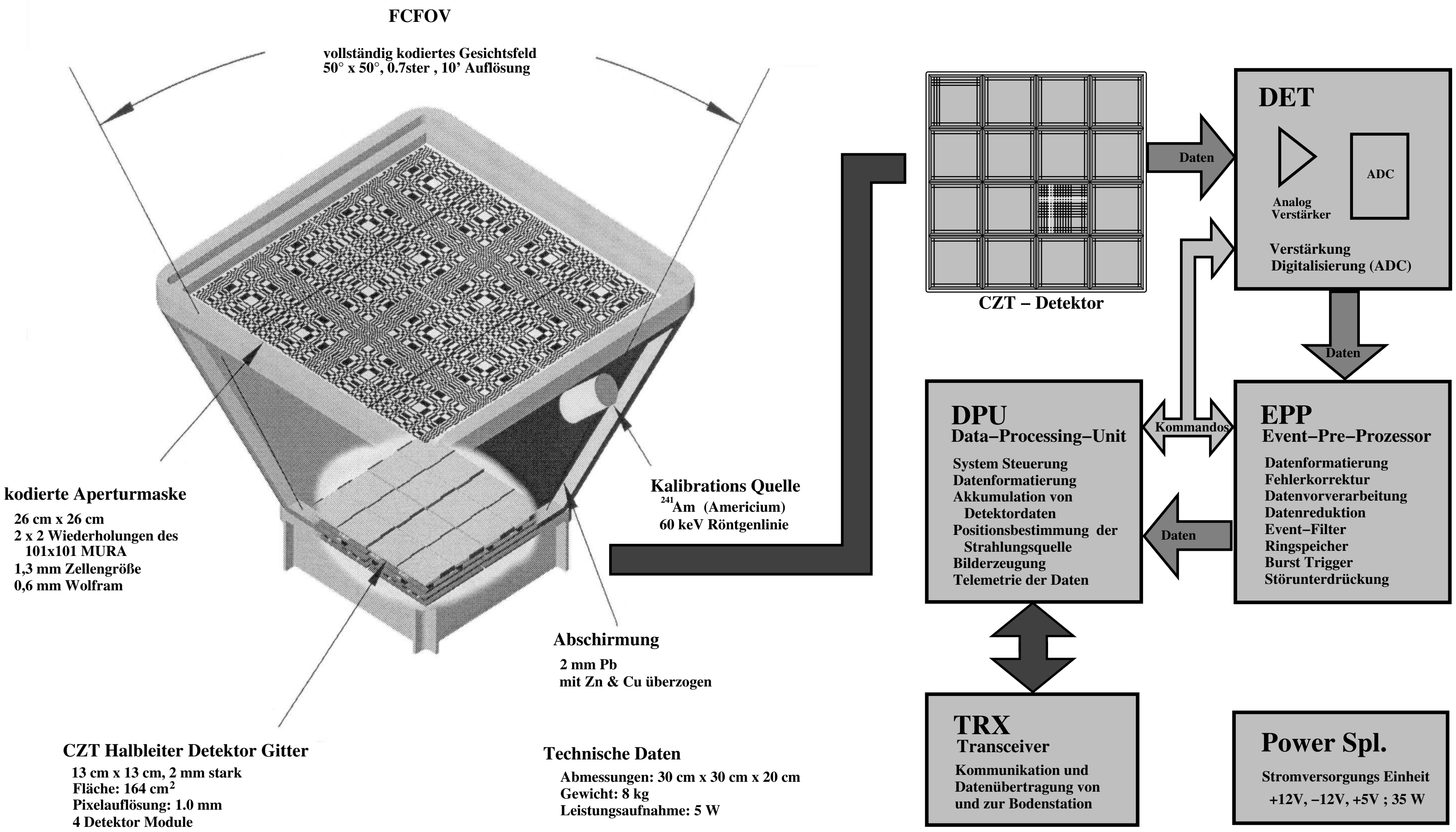


Abbildung 2.4: *Aufbau und Komponenten des CML.*

CMI Daten	
Energie Bereich	10-200(keV)
Energie Auflösung	3 keV bei 14 keV 4 keV bei 60 keV 6 keV bei 122 keV
vollständig kodiertes Gesichtsfeld (FCFOV)	50 ° x 50 °
Winkel Auflösung	32 Bogenminuten
Quellen Auflösung	12 Bogenminuten (10 Sigma Quelle)
Abmessungen	30 cm x 30 cm x 20 cm
Gewicht	8 kg
Leistungsaufnahme	5 W
Coded-Mask	
Zellen Größe	1,3 mm
Zellen Dicke	0,6 mm
Material	Wolfram
Maskenfläche	26 cm x 26 cm
Abstand Maske-Detektor	14 cm
Detektor	
Detektor Material	$Cd_{0.9}Zn_{0.1}Te$
Detektorsegmente	16 (4 x 4 Array) ; 4 Module
Segmentabmessungen	32 mm x 32 mm x 2 mm
Elektroden Geometrie	rechtwinklige Anordnung, 1 mm Streifenabstand
Pixel Größe	1 mm x 1 mm
Gesamtfläche	164 cm^2
Elektronik	16 32-Kanal RENA-ASICS, 14 Bit ADC
Detektorabmessungen	13,2 cm x 13,2 cm x 3 cm
Abschirmung	
Typ	passive Abschirmung, mit Z Abstufung
Abdeckung	11 sr
Material	Pb 2,0 mm Sn 0,5 mm Cu 0,1 mm

Tabelle 2.1: *Technische Daten des CMI*

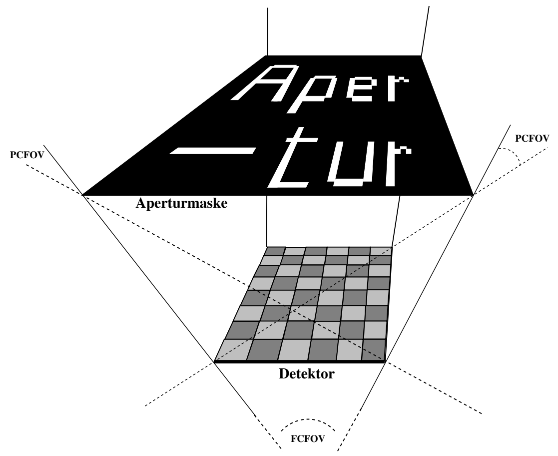


Abbildung 2.5: *Prinzip einer CM-Kamera. Quellen innerhalb des FCFOV werfen ein vollständiges Bild der Apertur auf den Detektor. Aus der Position des Schattens auf dem Detektor, kann die Richtung der Quelle rekonstruiert werden.*

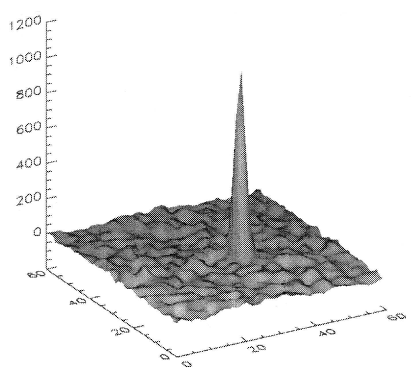


Abbildung 2.6: *Ideal korreliertes Bild einer Punktquelle. Liegt eine Punktquelle im FCFOV, so ergibt die Bildrekonstruktion einen Peak, der sich aus einem konstanten Hintergrund heraushebt [Rothschild et al. 1999].*

Die *coded mask* (CM) ist dazu für Röntgenstrahlung teilweise opak und teilweise transparent. Der Detektor ist speziell für die Detektion von Röntgenstrahlung gebaut und arbeitet ortsauflösend, d.h. er besteht aus einem *Array*, das mit jedem Ereignis auch den Ort der Wechselwirkungen auf dem Detektors ermittelt. Betrachtet die Kamera eine weit entfernte punktförmige Röntgenquelle, so projiziert diese Quelle ein Abbild der Aperturmaske auf den Detektor. Abbildung 2.5 zeigt das Prinzip der CM-Technik mit einer sehr vereinfachten Aperturmaske.

Das Urbild, die Punktquelle, ist durch Analyse der Intensitätsverteilung auf dem Detektor-*Array* und unter Verwendung mathematischer Verfahren rekonstruierbar, wenn die Maskengeometrie bekannt ist. Die einfachste Form, ein sogenanntes korreliertes Bild entsteht, indem auf einer Gitterstruktur für die jeweiligen Quellenpositionen die Zählratenverteilung des Hintergrundes mit dem Schatten der Maske für alle Richtungen in Beziehung gesetzt wird. Im Idealfall hat das korrelierte Bild genau einen *Peak* an der Stelle der Quellenposition, der sich aus einem konstanten Hintergrund hervorhebt (vgl. Abb. 2.6). Der Hintergrund ist unvermeidlich, er entsteht durch das Rekonstruktionverfahren und ist selbst bei idealisierten rauschfreien Quellen und vollkommener Abwesenheit von Detektorrauschen nicht zu vermeiden. Wenn auch unvermeidbar, so hängt das Kodierungsrauschen doch stark von der Geometrie der Maske ab. Bezüglich des Signal-Rausch Verhältnisses günstige Masken sind das *Uniformly-Redundant-Array* (URA) und das davon abgeleitete *Modified-Uniformly-Redundant-Array* (MURA), welches der CMI verwenden wird. Die URA bzw. MURA erzeugt bei idealen Punktquellen ein korreliertes Bild wie in Abbildung 2.6 zu sehen.

Das Gesichtsfeld eines CM-Teleskops wird durch die Größen *Fully-Coded-Field-Of-View* (FCFOV) und *Partially-Coded-Field-Of-View* (PCFOV) beschrieben. Die Begriffe sind in Abbildung 2.5 veranschaulicht. Die Verhältnisse für die Bildgewinnung sind ideal, wenn die Quelle innerhalb des FCFOV liegt. Alle Quellen innerhalb des FCFOV werfen ein vollständiges Bild der Aperturmaske auf den Detektor. Quellen die sich dagegen innerhalb des PCFOV, also außerhalb des FCFOV befinden, projizieren die Maske nur teilweise auf den Detektor. Als Folge entstehen bei der Bildrekonstruktion Artefakte, welche dem Kodierungsrauschen des Bildes äquivalent sind. Wünschenswert wäre demnach ein möglichst großes FCFOV. Leider nimmt damit in der Praxis der Rauschanteil des diffusen Röntgenhintergrundes zu. Dieser ist zu allem Überfluss bzgl. der Bildausdehnung nicht homogen, was die Rekonstruktion des Bildes erschwert

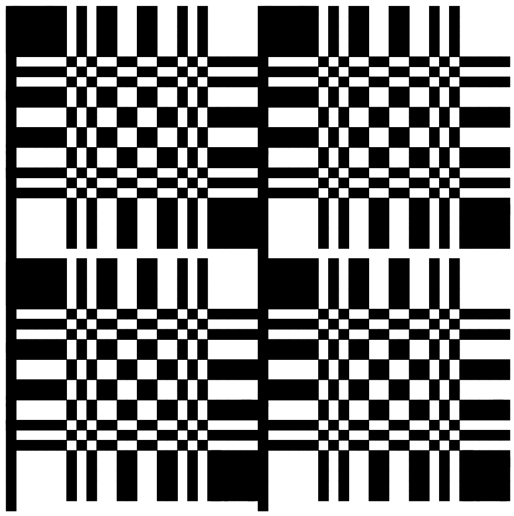


Abbildung 2.7: *Eine MURA wie sie beim Test des Detektors verwendet wurde. Für den CMI wird eine vergleichbare MURA, mit 101 x 101 Pixeln verwendet [Rothschild et al. 1999].*

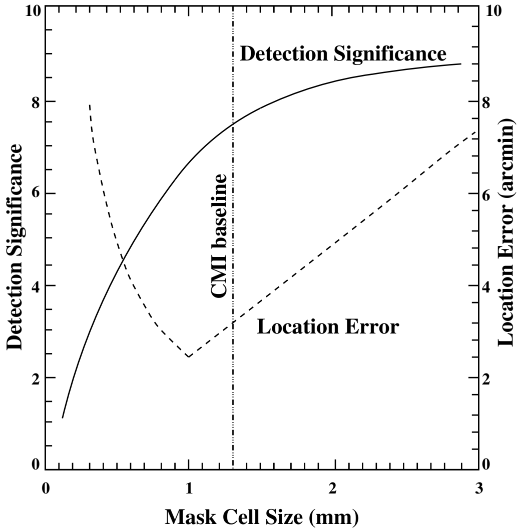


Abbildung 2.8: *Auflösungsvermögen und Empfindlichkeit in Abhängigkeit von der Zellengröße der MURA [Rothschild et al. 1999].*

und auch das Kodierungsrauschen zusätzlich erhöht. In der Praxis sind deshalb bei der Auswahl der Maske und der anderen Abbildungsparameter zahlreiche Kompromisse einzugehen. Die derzeitige Konfiguration des CMI stellt unter Berücksichtigung der Projekt-Ziele einen solchen Kompromiss dar.

2.2.2 Der CZT Detektor

Der Detektor des CMI ist ein ortsauflösender Halbleiterdetektor. Er besteht aus 16 Einzelsegmenten, die quadratisch zu einem 4 x 4 Gitter angeordnet sind. Elektrisch sind jeweils 4 benachbarte Detektorsegmente zu sogenannten Detektormodulen zusammengefasst. Jedes Modul besitzt einen eigenen Chip, mit dessen Hilfe Wechselwirkungsereignisse ausgelesen und verarbeitet werden, siehe hierzu Abbildung 2.4. Jedes Detektorsegment besitzt eine Fläche von 32 x 32 cm^2 und eine Dicke von 2 mm. Das Detektormaterial ist CdZnTe (CZT), eine Legierung von Cadmium-Tellurid und Zink-Tellurid, typischerweise im Verhältnis 9:1 ($Cd_{0.9}Zn_{0.1}Te$). Die große Bandlücke von 1,58 eV sorgt für eine geringe thermische Zufuhr von Elektronen in das Leitungsband, Eigenleitung und Rauschen von CZT bei Zimmertemperatur sind entsprechend gering. Die Folge ist ein hoher Materialwiderstand von größer 10^{11} Ω/cm und ein entsprechend kleiner Leckstrom bei typischen Feldstärken von ca. 1000 V/cm. CZT ist deshalb bei Zimmertemperatur ein Halbleiter. Das Shot-Rauschen, das von Fluktuationen des Leckstromes herrührt, ist klein und verspricht eine gute Energieauflösung (ca. 1 keV bei kleinen Detektoren). Die große durchschnittliche Massenzahl von 50 und die hohe Dichte von ca. 5,8 g/cm^3 machen CZT deshalb zu einem sehr geeigneten Material für den Bau hochenergetischer Röntgendetektoren. Die photoelektrische Absorption überwiegt bis zu Energien von 250 keV und der Absorptionsweg bei 60 keV beträgt nur 0,3 mm. Selbst nur wenige Millimeter dicke Detektoren haben deshalb bereits eine gute Ausbeute bis in den harten Röntgenbereich.

Der Detektor verwendet eine Technik spezieller Elektrodengeometrie bei der die Anschlüsse aus parallelen Streifen bestehen und so angeordnet sind, daß Kathoden- und Anodenanschlüsse von gegenüberliegenden Seiten an die Detektorplatte kontaktiert werden. Die Elektroden der Anode bilden mit den Elek-

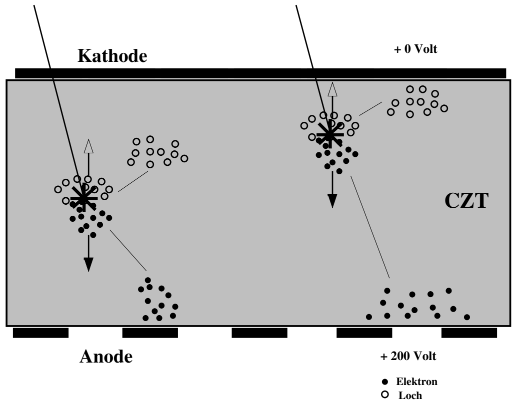


Abbildung 2.9: *Der Detektor im Schnitt. Die Röntgenphotonen fallen von der Kathodenseite ein. Dabei erzeugen sie durch Photoeffekt Elektronen, die ihrerseits Elektron-Loch-Paare ionisieren. Durch die angelegte Spannung driften die Ladungswolken zu den Streifenelektroden. Je weiter die Wechselwirkung von der Anode aus gesehen geschieht, desto größer ist die Ausdehnung der Ladungswolke und umso geringer die Ortsauflösung des Detektors.*

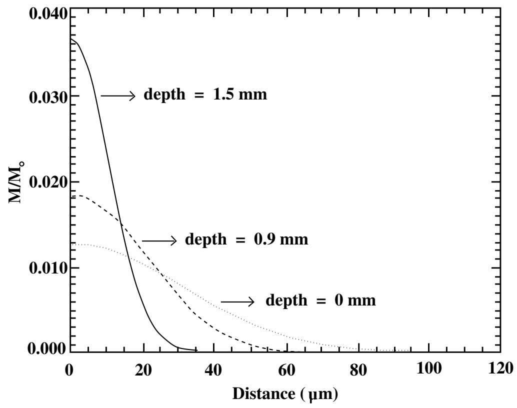


Abbildung 2.10: *Diffusion der Ladungswolke in Abhängigkeit der Wechselwirkungstiefe. Die Tiefe wird im Abstand von der Anode gemessen. Man erkennt deutlich wie die Ladungswolke für zunehmende Wechselwirkungs-Tiefen zerfließt. Der Effekt reduziert sowohl die Orts- als auch die Energie-Auflösung des Detektors [Kalemci et al. 2000].*

troden der Kathode einen Winkel von 90° und erzeugen so ein rechtwinkliges Koordinatengitter. Kommt es innerhalb des Detektors zur Wechselwirkung mit einem Photon, so wird die erzeugte Ladungswolke aufgrund der kleinen Absorptionslänge nur von den unmittelbar benachbarten Elektroden abgeleitet. Der Detektor erreicht so eine Ortsauflösung von einem Millimeter, wenn als Ort der Wechselwirkung die Elektrode mit dem maximalen Signal gilt. Die spezielle Elektrodentechnik wird als *Crossed-Strip-Readout* (CSR) bezeichnet. Die CSR-Technik besitzt gegenüber herkömmlichen Pixeldetektoren zahlreiche Vorteile. Ein CSR-Detektor ist einfacher im Aufbau, er kann aus einem Halbleiterstück gefertigt werden. In klassischen Pixeldetektoren entspricht sonst im Prinzip jedes Pixel einem einzelnen Detektor. Die Signale sind dort für jedes Pixel individuell zu korrigieren und es entstehen blinde Zonen an den Pixelrändern, was die Bildrekonstruktion erschwert. Ein Detektor in CSR-Technik hat diese Probleme nicht, die Pixel ergeben sich hier durch die spezielle Anordnung der Elektroden, und die *Readouts* können Streifen bezogen korrigiert werden. Zum Aufbau des CZT-Detektors in CSR-Technik, siehe Abbildung 2.11.

Entscheidend für die Energieauflösung des Detektors ist der Ladungstransport innerhalb des Halbleiters. Trifft ein Röntgenphoton den Detektor, so erzeugt es durch Photoeffekt ein Photoelektron. Dieses verliert seine Energie anschließend durch Ionisation, indem es auf seiner Bahn durch den Detektor solange Elektron-Loch-Paare erzeugt, bis es zum Stillstand kommt. Die Gesamtzahl, der auf diese Weise entstandenen Ladungsträger hängt von der Energie des Photoelektrons ab, also letztendlich von der Energie des Röntgen-Photons. Die Signalamplitude, die an den Elektroden registriert wird, ist deshalb ein direktes Maß für die Photonenenergie und kann zur spektroskopischen Untersuchung der Quelle verwendet werden. Wichtig für die Energieauflösung des Detektors ist der Umstand, daß möglichst alle Ladungsträger in die Auswertung gelangen. Dies ist unter realen Bedingungen nicht ganz erfüllbar. Zum einen rekombinieren manche Elektronen mit Kristall-Fehlern des Halbleiters, andere gehen in den Zwischenräumen der Elektroden verloren. Der Signalschwund ist von der Energie des Röntgenphotons und von der Wechselwirkungstiefe abhängig. Je höher die Photonen-Energie, und je tiefer die Wechselwirkung (Abstand von der Anode), desto größer ist die Ladungswolke bei ihrem Eintreffen auf der Anodenseite. Die Größe der Ladungswolke bestimmt, auf wieviele Anodenstreifen das Signal verteilt wird, und auch die Anzahl der

Elektronen die in den Zwischenräumen verloren geht. Für die Energie-Messung werden die Signale aller Anoden summiert. Der Signalschwund führt zu einer unerwünschten Verbreiterung des Signals der Energiemessung und hat letztendlich einen negativen Einfluss auf die Energieauflösung des Detektors. In Abbildung 2.14 ist der Zusammenhang zwischen der Energie des Röntgenphotons und der Energieauflösung dargestellt. Das Spektrum zeigt eine Überlagerung mehrerer Röntgen- und Gamma-Linien zwischen 14 und 184 keV, die mit einem Prototyp des CSR-Detektors aufgenommen wurde [Rothschild et al. 1999]. Die Linien haben alle eine Gaußförmige Verteilung, die Energieauflösung bei niederen Energien beträgt etwa 3 keV, bei hohen ca. 6 keV.

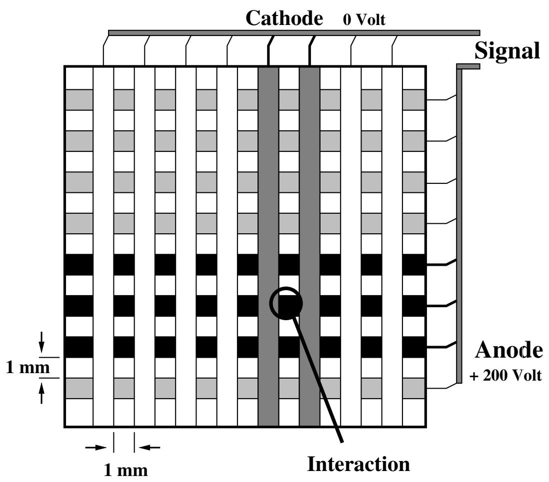


Abbildung 2.11: *Vereinfachte Darstellung eines CZT-Detektors in Crossed Strip Readout (CSR) -Technik. Die Pixel entstehen durch die Anordnung der Elektroden im Winkel von 90 Grad. Ein Problem ist, daß Ladungen in den Anodenzwischenraum driften und verloren gehen. Der Signalschwund steigt mit der Ausdehnung der Ladungswolke an.*

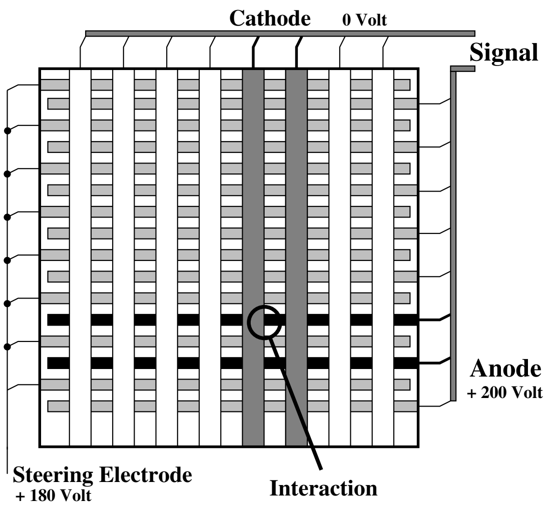


Abbildung 2.12: *Detektor in CSR-Technik mit zusätzlichen Steuerelektroden. Die Steuerelektroden fokussieren das elektrische Feld innerhalb des Halbleiters auf die Anodenstreifen, und erhöhen damit die Signalausbeute und die Energieauflösung des Detektors.*

Für die Ortsauflösung des Detektors ist die Größe der Ladungswolke ebenfalls von Bedeutung. Günstig für eine möglichst große Ortsauflösung wäre es, wenn die Ladungswolke gegenüber dem Streifenabstand der Elektroden so gering ist, daß jedes Ereignis auf den Bereich einer einzigen Anode und Kathode beschränkt bleibt. In der Praxis lässt sich dieser Fall leider nicht erreichen. Die Größe der Ladungswolke hängt wesentlich von der Energie des Photoelektrons ab. Je größer sie ist, desto größer ist in der Regel seine Absorptionslänge und schließlich die Ladungswolke. Für ein 40 keV Photoelektron beträgt die Absorptionslänge beispielsweise 10 μm, für ein 100 keV Photoelektron bereits 47 μm [Rothschild et al. 1999]. Die Ladungswolken sind nicht homogen bzgl. ihrer Ladungsdichte, die Elektron-Loch-Erzeugung nimmt gegen Ende der Absorptionsbahn deutlich zu. Für ein 100 keV Photoelektron entstehen beispielsweise 30% der Ladungen in den letzten 7 μm der Bahn [Rothschild et al. 1999]. Schließlich entsteht am Ort der primären Wechselwirkung ein Kα-Röntgenphoton mit einer Energie von ca. 25 keV. Dieses erzeugt ebenfalls innerhalb von ca. 85 μm eine Ladungswolke, die zusammen mit der ersten an den Elektroden registriert wird.

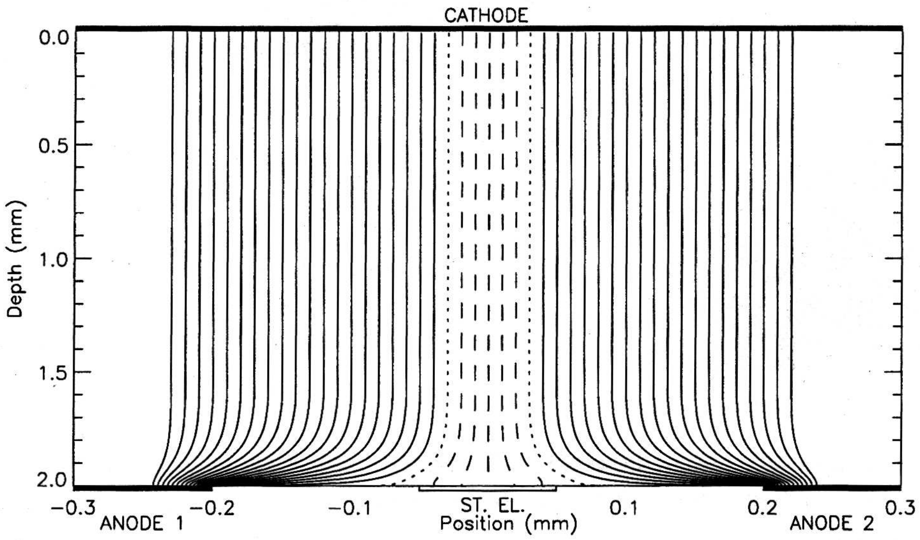


Abbildung 2.13: *Der Feldverlauf unter Verwendung von Steuerelektroden. Deutlich ist zu erkennen, wie das bzgl. der Anode negative Potential der Steuerelektroden den Feldverlauf verändert. Die Elektronen, die sich auf die Anode zu bewegen, driften entlang der Feldlinien [Kalemci et al. 2000].*

Zwischen Anode und Kathode liegt eine Gleichspannung von 200 Volt. Die freigesetzten Elektronen werden deshalb rasch in Richtung der Anode geleitet, die Löcher in Richtung Kathode. Die Driftgeschwindigkeit von Elektronen und Löchern ist sehr unterschiedlich. Die Elektronen erreichen die Anode von der Mitte des Halbleiters aus nach ca. 500 ns. Die Löcher benötigen dagegen für den gleich langen Weg zur Kathode etwa 100 μs [Rothschild et al. 1999]. Unglücklicherweise vergrößert sich die Ladungswolke auf dem Weg zur Anode. Durch Stöße mit dem Halbleitergitter und innere Abstoßung diffundieren die Elektronen auseinander und vergrößern so die Ausdehnung der Ladungswolke. Die Ausdehnung wird um so größer, je weiter der Weg zur Anode ist. Maximale Ausdehnung erreicht die Wolke in dem Fall, wenn die Wechselwirkung direkt in der Nähe der Kathode stattgefunden hat, der Weg zur Anode beträgt dann fast 2 mm. Abbildung 2.10 zeigt den Zusammenhang zwischen der Größe der Ladungswolke und der Tiefe der Wechselwirkung im Abstand von der Anode.

Der Signalschwund in den Anoden-Zwischenräumen kann durch Verwendung von sogenannten Steuerelektroden reduziert werden. Die Steuerelektroden liegen zwischen den Streifen der Anode des Detektors (siehe Abb. 2.12), und sind gegenüber dem Potential der Anode leicht negativ. In der Regel ist die Spannung um etwa 20 Volt geringer als an der Anode, sie liegt im Fall des CMI-Detektors [Rothschild et al. 1999] auf einem Potential von 180 Volt. Um die Steuerelektrode herum entsteht als Folge ein leicht negativer Potentialbereich. Elektronen die über dem Anoden-Zwischenraum entstehen, werden deshalb größtenteils in Richtung der Anoden gezogen. Der Feldverlauf innerhalb des CZT-Halbleiters ist in Abbildung 2.13 gezeigt. Es ist deutlich erkennbar, wie die Steuerelektroden das elektrische Feld auf die Anodenstreifen fokussieren. Die Elektronenbahnen folgen in etwa dem Feldverlauf. Als Folge gelangen deutlich weniger Elektronen in den Elektroden-Zwischenraum (gepunktete Linie), die Signalausbeute ist deshalb erhöht. Die Verwendung von Steuerelektroden verbessert die Energie-Auflösung des Detektors. Die Breite von Anodenstreifen und Steuerelektroden beträgt je 100 μm, die Breite der Kathodenstreifen 450 μm mit Abstände von 500 μm. Die Pixelgröße von ca. 1 mm und die Ortsauflösung bleibt auf diese Weise wie beim ursprünglichen CSR-Detektor bestehen.

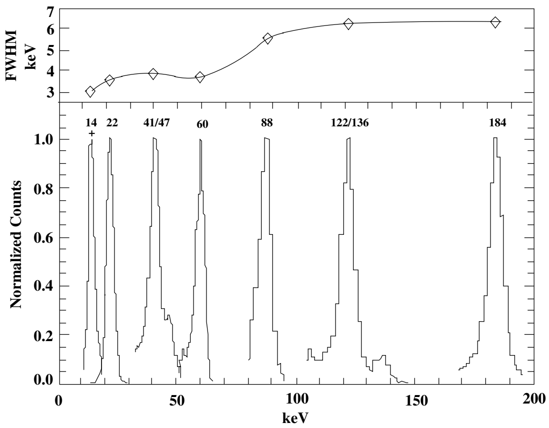


Abbildung 2.14: *Energieauflösung in Abhängigkeit von der Energie des Photons. Deutlich erkennt man, wie die Breite der Linien für hohe Energien zu-, und die Auflösung abnimmt. Der Effekt beruht auf der Tatsache, daß die Ausdehnung der Ladungswolken für hohe Energien zunimmt. Je höher die Energie des Röntgen-Photons, desto größer die Absorptionslänge des zugehörigen Photoelektrons. Die Ausdehnung der Ladungswolke bestimmt, auf wieviele Anodenleitungen das Signal verteilt wird, und damit auch die Größe des Signalschwunds [Rothschild et al. 1999].*

Der Detektor des CMI wird von CASS @ UCSD in Kalifornien entwickelt. Er profitiert aus langjährigen Erfahrungen mit dieser Detektortechnologie, die insbesondere bei Ballon Experimenten bereits ihre Leistungsfähigkeit erwiesen hat.

2.2.3 Die Elektronik-Komponenten

Anders als bei optischen Teleskopen möglich, kommt ein *Coded-Mask* Teleskop in keinem Fall ohne elektronische Komponenten aus. Das menschliche Auge vermag nur Wellenlängen im Bereich von 400 bis 800 Nanometer zu erkennen. Für die Astronomie bei längeren und kürzeren Wellenlängen sind deshalb die Bildinformationen in den für Menschen sichtbaren Bereich zu transformieren. Nicht genug damit, ist im Bereich harter Röntgenstrahlung auch die Abbildung mit bilderzeugenden Systemen nicht möglich. Ein Bild entsteht dort erst als Folge statistischer Untersuchungen zahlreicher Einzelereignisse eines elektronischen Teilchendetektors. Auch die Positionsbestimmung und die *Burst*-Suche sind ohne aufwendige elektronische Schaltungen nicht denkbar. Für den CMI, der dieser Problematik unterliegt, bedeutet dies einen enormen Entwicklungsaufwand für die elektronischen Komponenten. Der CMI wird neben ganz grundlegenden Komponenten wie Telemetrie und Stromversorgung mit mehreren Elektronik-Platinen für Datengewinnung, Datenverarbeitung und Bilderzeugung ausgestattet sein. Die Baugruppen sind in Abbildung 2.4 dargestellt. Die Elektronik zur Datenverarbeitung des CMI kann in zwei Bereiche unterteilt werden:

- Detektorelektronik (DET)
- *Central Electronic Unit* (CEU)

Die Detektorelektronik wird zusammen mit dem Detektor, bei CASS @ UCSD in Kalifornien, die CEU zum Großteil am IAAT der Universität Tübingen entwickelt. Die CEU besteht zum einen aus dem *Event-Pre-Processor* (EPP) und zum anderen aus der *Data-Processing-Unit* (DPU), einem Bord-Computer zur wissenschaftlichen Datenverarbeitung. Die Elektronik Komponenten des CMI werden in diesem Kapitel kurz vorgestellt.

Die Detektor-Elektronik:

Abbildung 2.15 zeigt das Block-Schaltbild der Detektor-Elektronik. Ihre Aufgabe ist die Signale der CZT-Detektoren auszulesen, und zu sogenannten *Hits* zu verarbeiten. Ein *Hit* ist ein 40 Bit breites Datenpaket, das alle Informationen enthält, die eine Streifenelektrode des Detektors liefert. Diese Informationen sind im einzelnen: Signalamplitude, Streifennummer, Segmentnummer und ein Zeitstempel, der den Zeitpunkt der Wechselwirkung festlegt.

Zur Erzeugung der *Hits* verfügt jedes Detektorsegment über einen eigenen Chip, einen speziellen *Application-Specific-Integrated-Circuit* (ASIC). ASICs sind hochintegrierte elektronische Schaltungen die, anders als Mikroprozessoren, in der Regel nicht frei programmierbar sind, sondern für eine ganz spezielle Aufgabe oder Anwendung entwickelt werden. Die für den CMI verwendeten RENAs (*Readout Electronics for Nuclear Applications*) der Firma Nova R&D sind ASICs in diesem Sinne [Rothschild et al. 1999].

Für jeden einzelnen Elektrodenstreifen eines Detektorsegments besitzt der zugehörige RENA-Chip einen eigenen Kanal. Jeder dieser Kanäle ist mit einem ladungsintegrierenden Vorverstärker sowie einem Signaldiskriminator bestückt, und verfügt außerdem über die Möglichkeit, Signalform und Spitzenamplituden zu verarbeiten. Jeweils vier RENAs (und damit Detektorsegmente) sind zu einem Modul zusammengefasst. Die RENAs sind so miteinander verschaltet, daß ein *Trigger* auf irgend einem Kanal das sofortige Einfrieren der Signalwerte aller anderen Kanäle und RENAs zur Folge hat. Die *Trigger*-Signale und Steuerleitungen sind mit einer schnellen Logik-Einheit (*prompt logic*) verbunden, welche im Bedarfsfall das sequentielle Auslesen der RENAs einleitet. Die RENA-Chips besitzen eine spezielle Signalübertragungs-Technik, bei der ein ge-*triggerter* Kanal das Lesen seiner zwei direkt benachbarten Kanäle auslöst. Auf diese Weise kann die gesamte, zur Charakterisierung der *Events* benötigte Information gespeichert werden. Andere Kanäle, die zum Signal nichts beitragen, werden auch nicht gelesen. Dies kommt der Geschwindigkeit der *Hit*-Erzeugung zugute, was sich in einer kleinen Totzeit des Detektors auszahlt. Zusätzlich wird das Datenaufkommen reduziert. Sind die Signale gelesen, gelangen sie in einen A/D-Wandler [7] und werden zu 14 Bit breiten Datenworten verarbeitet. Die 14 Bit breite Amplitude wird dann zusammen mit der Elektroden- (7 Bit) und Segment-Nummer (3 Bit), sowie dem Zeitstempel (16 Bit) als 40 Bit breiter *Hit* an die CEU übertragen.

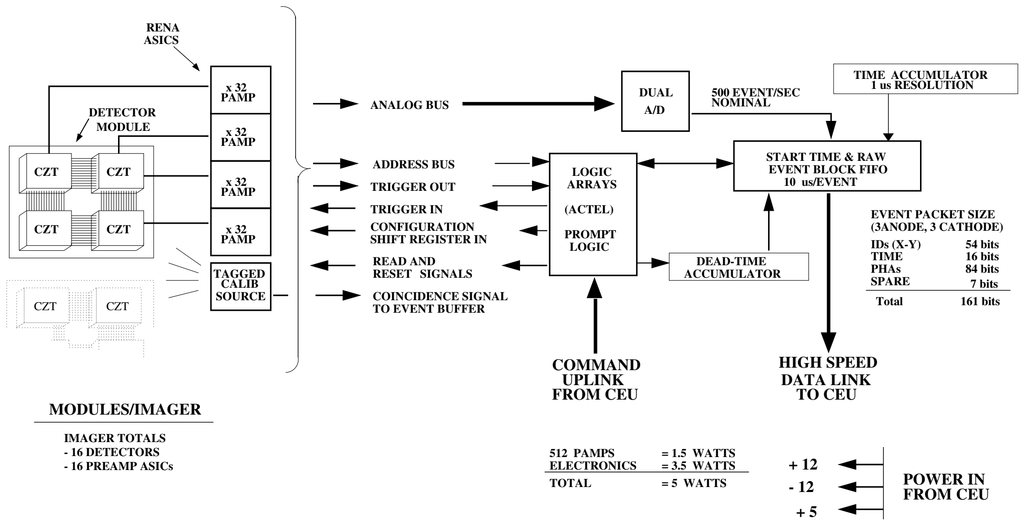


Abbildung 2.15: *Blockdiagramm der Detektorelektronik. Der Detektor ist segmentweise mit RENA-Chips verbunden, diese lesen die Signale der Elektrodenstreifen. Nach einem Trigger friert die Prompt-Logic den Zustand aller RENAs des Detektors ein. Anschließend wird die aktive Elektrode zusammen mit den beiden benachbarten gelesen, und in einem A/D-Wandler in ein 14 Bit breites Amplitudenwort umgewandelt. Dieses wird zusammen mit der Elektrodennummer und einem Zeitstempel als 40 Bit breites, sogenanntes Hit an die CEU gesendet [Rothschild et al. 1999].*

[7] A/D-Wandler: Analog zu Digital Wandler

Das Datenformat der *Hits* ist in Kapitel 4.3.1 HBR-*Interface-Unit* Detektor/EPP, Abbildung 4.13 dargestellt. Die *Hits* werden mit Hilfe eines 16 Bit breiten, PCI kompatiblen Busses an den EPP der CEU übertragen.

Die *Central Electronic Unit* (CEU):

Die CEU setzt sich aus dem *Event-Pre-Processor* (EPP) und der *Data-Processing-Unit* (DPU) zusammen. Der EPP wird derzeit am IAAT in Tübingen entwickelt.

Der *Event-Pre-Processor* (EPP):

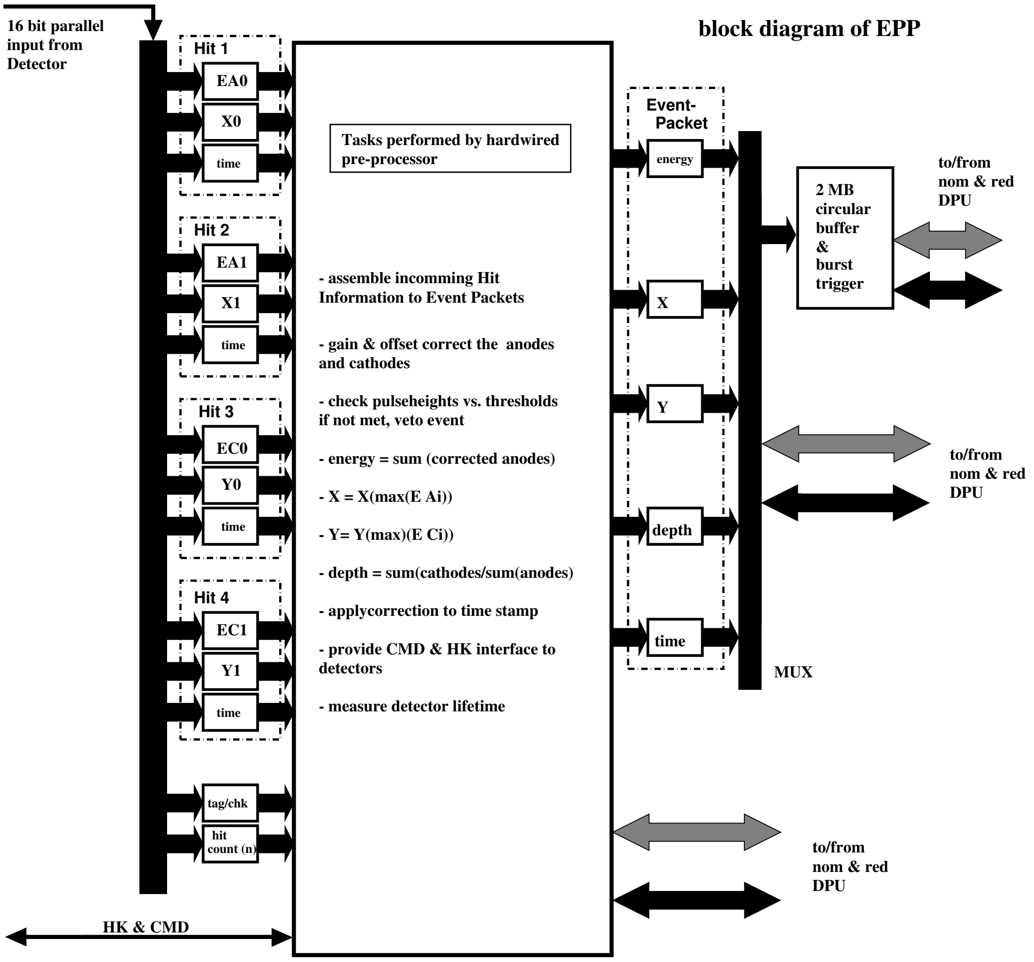


Abbildung 2.16: *Blockdiagramm des Event-Pre-Processors. Die Daten werden von links in Form von Hits empfangen und nach rechts in Form von Event-Packets gesendet. Dazwischen liegt der EPP, dessen Aufgaben innerhalb der Box in Stichpunkten beschrieben sind [Rothschild et al. 1999].*

Der EPP soll an dieser Stelle nur kurz beschrieben werden, da sich Kapitel 4 ohnehin sehr ausführlich damit beschäftigt. Die Ein- und Ausgangsdaten, sowie die Aufgaben sind in Abbildung 2.16 zusammengefasst. Die Daten werden von links in Form 40 Bit breiter *Hits* empfangen. Innerhalb des EPP durchlaufen sie diverse Korrekturen und werden am Ausgang in sogenannten *Event-Packets* an die Bilderzeugungs-Elektronik der DPU gesendet. Für zeitkritische Aufgaben besitzt der EPP auch Ausgänge, die direkt in die

Telemetrie geleitet werden. Der EPP wird vollständig in einer Hardware-Beschreibungssprache entwickelt, und in Form eines weltraumtauglichen FPGA[8] der Firmen ACTEL oder Xilinx realisiert. Auch die Verwendung eines ASIC[9] steht diesbezüglich zur Diskussion. Für eine detaillierte Beschreibung der Aufgaben und der Funktion des EPP, siehe Kapitel 4.

Die *Data Processing Unit* (DPU):

Die DPU ist eine Elektronik-Einheit zur wissenschaftlichen Datenverarbeitung. Sie wird aus Gründen der Redundanz aus zwei identischen Mikrocomputerbausteinen bestehen, und kann im Bedarfsfall die Daten von bis zu vier *Coded-Mask Imagern* verarbeiten [Rothschild et al. 1999]. Im Falle einer Mission mit mehreren CMI, würde jeder *Imager* über einen eigenen EPP verfügen, die DPU jedoch zentral für alle CMI gemeinsam arbeiten. Aufgabe der DPU ist die wissenschaftliche Analyse der Detektordaten. Dazu gehört vor allem die Bilderzeugung und die Winkelrekonstruktion aus den Positionsdaten der einzelnen *Events*. Desweiteren dient die DPU zur Steuerung des EPP und der Detektorelektronik. Sie vermittelt die Kommunikation dieser Baugruppen mit dem Satellitenbus und der Telemetrie-Einheit. Die *Interfaces* der DPU sind allesamt digital. Es handelt sich dabei um sogenannte *High-Bit-Rate-Interfaces* (HBRIs), die dem Austausch wissenschaftlicher Daten dienen, und um ein *Low-Bit-Rate-Interface* (LBRI) für betriebstechnische Daten und Steuerkommandos.

Die Notwendigkeit für die Datenverarbeitung an Bord, ergibt sich vor allem aus zwei Gründen:

- Detektion und Positionsbestimmung von *Gamma-Ray-Bursts*: Ziel ist die zuverlässige Aufzeichnung von *Bursts* innerhalb des Gesichtsfeldes des Imagers, und deren Positionsbestimmung mit einer Genauigkeit von ca. 10 Bogenminuten innerhalb weniger Sekunden.

- Das Verhältnis von wissenschaftlichen Daten (ca. 25 kbit/s pro *Imager*) zur erwarteten Telemetrierate (ca. 10 kbit/s) erreicht bei mehreren CMI schnell einen Faktor Zehn.

Es ist deshalb Aufgabe der DPU, die Daten in geeigneter Weise zu verarbeiten und die Telemetrie entsprechend zu entlasten. Zeitkritische Berechnungen benötigen eine sehr schnelle Verarbeitung und werden nicht mit Softwareprogrammen innerhalb der DPU gerechnet, sondern durch den festverdrahteten EPP.

Die Hardware der DPU erfordert einen Prozessor mit einer Leistung von etwa 20 MFLOPS[10]. Es sind dazu mehrere Prozessoren verschiedener Hersteller im Gespräch. Ein Prozessor höherer Leistung wäre im Prinzip wünschenswert, unterliegt jedoch der Problematik, daß das Design für einen Betrieb im Weltraum strahlungshart sein muß. Handelsübliche Prozessoren, die mittlerweile Leistungen bis zu einem GFLOP liefern können, genügen diesen Anforderungen leider nicht, ihre Strukturen würden unter Weltraumbedingungen schon nach kurzer Zeit zerstört. Der verfügbare Speicher der DPU beträgt ca. 6 MByte für temporäre Daten und 2 MByte für Programm-Software. Außerdem hat die DPU auch Zugriff auf den 2 MByte großen Ring-Speicher des EPP, der Adressraum beträgt somit mehr als 10 MByte, was einer großzügigen Dimensionierung entspricht. Der Speicher der Programmsoftware wird mit EDAC-RAMs[11] realisiert, der temporäre Daten-Speicher mit non-EDAC RAMs. Diese Maßnahme schützt die Programmsoftware vor Zerstörung und den CMI vor Prozessorabstürzen. Die Betriebs-Software des CMI benötigt, so die gegenwärtige Planung, nur einen Teil des Software-Speichers, der Rest steht späteren Programmergänzungen zur Verfügung. Auf diese Weise kann die Funktionalität der DPU entsprechend den jeweiligen Missionszielen auch nach dem Start der Mission noch variiert werden.

[8]FPGA: *Field-Programmable-Gate-Array*

[9]ASIC: Application Specific Integrated Circuit

[10]Die Bezeichnung FLOP steht für *Floating-Point Operations per Second* und ist ein gängiges Maß für die Rechengeschwindigkeit eines Prozessors. Entsprechend steht MFLOPS für Mega-FLOPS und GFLOPS für Giga-FLOPS.

[11]EDAC: *Error Detection and Correction*; bezeichnet einen speziellen Typ von Speicherbaustein, der über Mechanismen zur Fehlerkorrektur verfügt.

2.2.4 Die Kalibrationsquelle

Sowohl der Halbleiter-Detektor, als auch die Elektronik unterliegen diversen Alterungsprozessen. So führen zum Beispiel Strahlenschäden innerhalb des CZT-Detektors zu einer Erhöhung der Zahl der Kristallfehlstellen (*Traps*). Dies bewirkt im Laufe der Zeit einen größeren Ladungseinfang und eine Reduzierung der Transporteffizienz. Die Folge ist eine Verringerung der Energieauflösung des Detektors. Auch Halbleiter-Verstärker unterliegen im Weltraum ähnlichen Prozessen und reduzieren ihre Leistungsfähigkeit mit der Zeit. Für eine ständige Kontrolle dieser Defekte und ggf. eine Korrektur, benötigt der CMI eine Eichquelle.

Die Eichquelle besteht aus einer radioaktiven Substanz und hat die Aufgabe Röntgen-Photonen definierter Energie auszusenden, die eine Kallibration der Detektoreigenschaften ermöglichen. Im Fall des CMI ist diese Substanz das Isotop ^{241}Am von Americium, dessen Emissionslinien im Röntgenbereich bei 59,90 keV liegt, und für den Energie-Bereich des CMI besonders geeignet erscheint. Das Spektrum von ^{241}Am ist in Abbildung 2.17 zu sehen. Es wurde bei Messungen mit einem Prototyp des CZT-Detektors bei CASS @ UCSD aufgenommen [Rothschild et al. 1999].

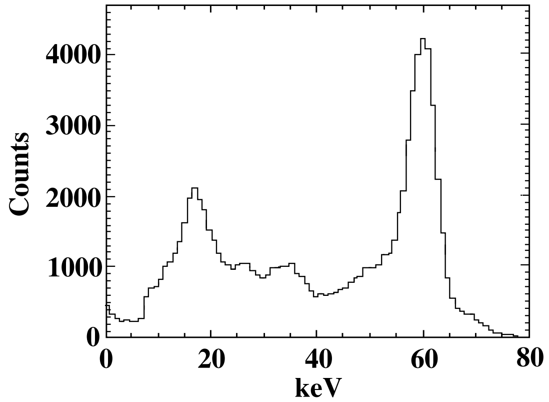


Abbildung 2.17: *Röntgen-Spektrum des Isotops ^{241}Am. Deutlich erkennbar ist die Emissionslinien bei 59,5 KeV. Die ^{241}Am -Quelle ist in einen Szintillator eingebettet, der Zerfall wird mit Hilfe eines Photomultipliers nachgewiesen. Röntgen-Ereignisse innerhalb des CZT-Detektors, die gleichzeitig mit einem Event im Szintillations-Detektor auftreten, werden als Kallibration-Ereignisse gekennzeichnet. Das Spektrum wurde mit einem Prototyp des CZT-Detektors unter Verwendung der RENA-Technik aufgenommen [Rothschild et al. 1999].*

Damit die Röntgen-Emissionen der ^{241}Am -Quelle nur zur Kalibrierung verwendet werden, und nicht versehentlich in die Auswertung geraten, ist die ^{241}Am-Quelle in einen Szintillations-Detektor eingebettet. Der Szintillations-Detektor registriert die Alpha-Emission der ^{241}Am Quelle, die bei jedem Zerfall zeitgleich mit der Röntgenemission stattfindet. *Event-Packets*, für die Koinzidenz mit dem α-Zerfall besteht, werden mit einem speziellen Bit als Kalibrationsereignisse markiert, und an die Bodenstation gesendet. Die Bodenstation kann auf Basis dieser Daten eine Nacheichung des Detektors vornehmen. Dazu werden speziell angepasste Korrekturwerte für den *Look-Up-Table* berechnet, die in der Amplituden-Korrektur den Alterungs-Effekt zu einem Teil wieder kompensieren können.

Die korrekte Auswertung von Kalibrations-Ereignissen ist bis zum heutigen Stand des EPP-Projektes nicht implementiert worden. Wie anderes auch, wurde sie zu Gunsten anderer Entwicklungen vorerst zurückgestellt. Das Protokoll der *Event-Packets* und die Funktion des EPP ist zu gegebener Zeit diesbezüglich zu erweitern.

Kapitel 3

Modernes Elektronik Design

3.1 Entwicklungstendenzen

Die Elektronik der letzten 20 Jahre ist vor allem durch zwei Veränderungen geprägt worden:

- Der weitgehenden Umstellung von der Elektronik analoger Schaltungen zur Digitaltechnik.
- Der stetig fortschreitenden Miniaturisierung der Bauteile und dem Übergang zur Mikroelektronik!

Diese Veränderungen haben auch deutliche Spuren für das Design und die Entwicklung elektronischer Schaltungen und Anwendung hinterlassen.

3.1.1 Der Übergang zur Digitaltechnik

Digitale Elektronik bietet für die industrielle Anwendung deutliche Vorteile gegenüber analoger Elektronik. Digitale Schaltungen sind wesentlich besser spezifizierbar, ihre Eigenschaften im Betrieb und während der Alterung sind voraussagbarer. Die Bauteile lassen sich im Wesentlichen auf einen Grundtyp zurückführen, der in großen Stückzahlen kostengünstig hergestellt werden kann. Die Funktion von digitalen Schaltungen kann mathematisch einfach beschrieben werden, und die Signale sind einer EDV bezogene Weiterverarbeitung besser angepasst. Aus diesen Gründen beschäftigt sich Elektronikdesign heutzutage fast ausschließlich mit dem Entwurf digitaler Schaltungen, was auch auf die vorliegende Arbeit zutrifft.

3.1.2 Die Miniaturisierung der Bauteile

Gewissermaßen war es die Miniaturisierung elektronischer Bauteile, die den Siegeszug der Elektronik im allgemeinen, und der digitalen Elektronik im speziellen erst ermöglicht hat. Die Miniaturisierung bewirkte eine drastische Reduktion der Herstellungskosten, eine platzsparende Unterbringung ständig komplexer werdender Aufgaben, und eine deutliche Verkürzung bei den Ausführungszeiten. Am Anfang dieser Entwicklung stand die Erfindung des *Integrierten Schaltkreises* (IC), mit dem es gelang, immer umfangreichere Transistorschaltungen auf einem einzigen Bauteil zu vereinen. Natürlich müßte die Aufgabe bzw. Funktion eines IC beim Entwurf festgelegt werden, und so entstanden mit der Zeit große Sortimente unterschiedlichster IC's für verschiedene Anwendungen. Insbesondere die TTL74 Baureihe erlangte große Bedeutung. Sie vereint in sich ein Sortiment digitaler Grundschaltungen[1], sogenannter logischer Gatter, die zu komplexeren digitalen Anwendungen verdrahtet werden können. Die Verdrahtung der ICs geschieht auf einer Leiterplatte.

Seit Beginn der 90'er Jahre ging die Miniaturisierung noch einen Schritt weiter, die sogenannten *Programmable-Logic-Arrays* (PLAs) begannen sich durchzusetzen. Die Verdrahtung vieler logischer Gatter auf einer Leiterplatte birgt gewisse Nachteile. Vor allem sehr komplexe digitale Aufgaben, können

[1] AND, NAND, OR, NOR, EXOR, INVERTER, FLIPFLOPs usw.

leicht mehrere hundert Gatter-ICs erfordern. Die Verdrahtung ist zeitaufwendig, und störanfällig. Etwaige nachträgliche Designänderungen sind nur mit großer Mühe möglich, oder benötigen gar einen kompletten Neuaufbau. Auch Stromverbrauch, Wärmeentwicklung und Platzbedarf steigen mit jedem weiteren IC der Leiterplatte. Insbesondere Schaltungen, die mit hohen Takt-Frequenzen arbeiten, sind enge Grenzen gesetzt. Probleme dieser Art könnten umgangen oder zumindest vermindert werden, wenn die komplette digitale Schaltung mit allen Gattern in einem einzigen Bauteil untergebracht werden könnte. Genau dies ist der Grundgedanke der PLA-Bausteine. Die Idee war zwar nicht neu, schließlich gab es 'Anwendungs Spezifische ICs' (ASICs) schon seit Anfang der 70'er Jahre, neu war aber, daß die PLAs frei programmierbar waren, und daß der Entwickler selbst diese Programmierung vornehmen konnte. Auch die Entwicklung der PLAs ging weiter, und hat in den letzten Jahren eine ganze Reihe verschiedener PLA-Familien hervorgebracht. Eine davon, die *Field-Programmable-Gate-Arrays* (FPGA), sind für Aufgaben, wie der hier beschriebene Entwicklung, besonders gut geeignet.

3.2 Entwicklungswerkzeuge

Auch die Design-Werkzeuge haben sich mit den Anforderungen und Möglichkeiten neuer Trends immer wieder gewandelt. Auch eine veränderte Methodik bei der Entwicklung trägt dem Rechnung. Während Schaltungen zu Beginn der Miniaturisierung am Reißbrett entworfen wurden, übernehmen heute leistungsfähige Computerprogramme diese Aufgabe. Mit dem Auftauchen preiswerter CAD[2]-Workstations Ende der 80'er Jahre wurde das Elektronik- und Chipdesign mehr und mehr auf den Rechner verlagert. Für die Elektronik Industrie bedeutete dies deutliche Einsparungen bei Entwicklungskosten und Entwicklungszeiten. Computer vermögen zudem noch mehr als das bloße Design. Mit den heutigen Programmen ist auch eine ausführliche Simulation der Entwürfe möglich, und zwar bevor irgend eine Komponente in Hardware realisiert wird. Mächtige Beschreibungssprachen bieten zudem die Möglichkeit zur Dokumentation und zum weltweiten Austausch von elektronischen Designs.

3.2.1 *Schematics*

Seit dem Auftauchen der ersten bezahlbaren Computersysteme haben sich nach und nach CAD-basierende Computerprogramme für das Elektronik- und Chip-Design durchgesetzt. Am Anfang standen dabei sogenannte Design-Editore (*schematics editor*). Ein Design-Editor besitzt eine Bibliothek graphischer Symbole, die den digitalen Bauteilen entsprechen, und die mit der Maus am Bildschirm zu mehr oder weniger komplexen Schaltungsdesigns verbunden werden können. Der Rechner kann anschließend die Schaltung auf Funktion und Laufzeit analysieren und die Reaktion auf Eingangs-Signal Stimuli graphisch darstellen. Abbildung 3.1 zeigt ein einfaches Programm dieser Art. Leistungsfähigere Programme besitzen zudem die Fähigkeit, Platinenlayouts zu synthetisieren und Leiterbahnwege zu entflechten (*routing*). Seit dem Auftauchen programmierbarer Logik-*Arrays* besitzen viele Programme nun auch die Fähigkeit, Netzlisten zur Programmierung dieser Bausteine zu synthetisieren. Das Design kann dann nach Entwurf und Simulationen mit dem Design-Editor, direkt in den PLA-Chip einprogrammiert werden. Die Industrie bietet dazu Programmiergeräte an, die zwischen Design-Editor und PLA-Chip als *Interface* dienen.

Mit der Zeit entstand eine große Vielfalt programmierbarer PLAs, entsprechend unüberschaubar war und ist die Zahl der Design- und Programmierwerkzeuge. Zudem ist die Programmierung mittels Design-Editor je nach Software-Hersteller und verwendeter PLA-Chiptechnologie uneinheitlich. Ständig komplexer werdende Designaufgaben machen zudem eine Modularisierung erforderlich und die Entwicklung in mehreren Teams, die oft international arbeiten. Diese Probleme sind ohne eine international standardisierte, portierbare Hardwarebeschreibung nur schlecht zu bewältigen. Aus diesem Grund begann das DOD[3] der USA bereits Anfang der 80'er Jahre mit der Entwicklung von Hardwarebeschreibungssprachen (HDLs) zur Struktur- und Funktionbeschreibung integrierter Schaltungen [Ashenden 1990].

[2]CAD: Computer-Aided-Design
[3]DOD: *Departement Of Defense*, amerikanisches Verteidigungs-Ministerium

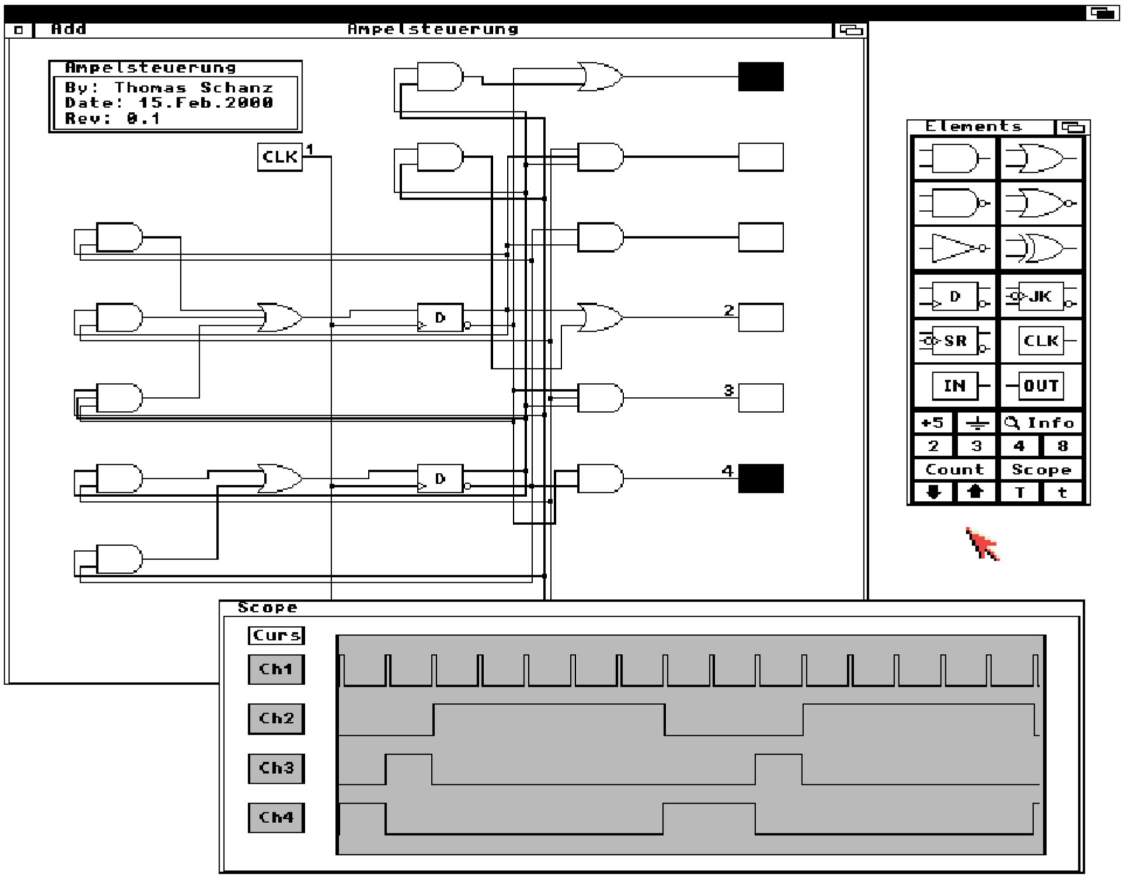


Abbildung 3.1: *Ein einfacher Design-Editor. Die digitalen Bauteile stehen als Symbole in einer Bibliothek zur Verfügung (Fenster rechts oben) und können mit der Maus innerhalb des Editors zu Schaltungen verbunden werden. Die Funktion des Designs kann anschließend mit allerlei Hilfsmitteln, hier z.B. mit einem* Scope *(Fenster rechts unten) überprüft werden.*

3.2.2 Hardware-Beschreibungssprachen

Eine Hardware-Beschreibungssprache (HDL *Hardware-Description-Language*) dient dem Entwurf und dem Test elektronischer Schaltungen, und stellt eine Alternative zu *Schematics* dar. HDLs können sowohl die Struktur, als auch die Funktion elektronischer Schaltungen abbilden. Die Syntax ähnelt i.a. der höherer Programmiersprachen, enthält aber zusätzliche Sprachelemente, z.B. für die Verschaltung von Funktionsblöcken. HDL-Designs lassen sich mit Hilfe eines sogenannten *Hardware-Compilers* in eine Schaltungsstruktur übersetzen (Synthese). Ein HDL-Modell bietet in der Regel auch die Basis für eine Simulation der spezifizierten Bausteine, sodaß der Entwickler Alternativen bewerten kann, ohne Prototypen herstellen zu müssen.

Die beiden weitverbreitetsten HDLs sind VERILOG [Golze 1995] und VHDL [Ashenden 1990, Lehmann et al. 1994]. Die Syntax von VHDL ähnelt sehr der Programmiersprache ADA. Sie enthält alle Elemente einer prozeduralen Programmiersprache, erweitert um Konstrukte für den Schaltungsentwurf. Die Abkürzung VHDL steht für *VHSIC Hardware-Discription-Language*, VHSIC für *Very-High-Speed-Integrated-Circuits*. VHDL wurde seit 1983 entwickelt und 1987 durch das IEEE[4] standardisiert. Die

[4]IEEE: Institute of Electrical and Electronic Engineers; Standardisierungs-Gremium der USA

Sprache hat unlängst begonnen, ihre Mitkonkurrenten (VERILOG und Andere) zu verdrängen und beste Aussichten, sich als weltweiter Standard zu etablieren. Durch die Standardisierung erreicht VHDL die Herstellerunabhängigkeit. Die Designs sind damit nicht nur zwischen verschiedenen Rechnerplattformen austauschbar, sondern auch zwischen Software verschiedener Hersteller. Der Entwickler ist bei der Wahl des Rechners und der Software somit vollkommen frei.

3.3 Die Sprache VHDL

Ein VHDL-Design bietet eine vollständige Beschreibung einer elektronischen Schaltung. Dazu ist es möglich, sowohl die Struktur (Aufbau in Funktionsblöcke und Verdrahtung dieser Blöcke), als auch die Funktion (funktionale Beschreibung mit Hilfe von Operatoren) einer Schaltung abzubilden. Jedes VHDL-Design ist untergliedert in ENTITY und ARCHITECTURE. Optional kann auch ein CONFIGURATION-Teil existieren. In Abbildung 3.2 ist der schematische Aufbau eines VHDL-Designs gezeigt.

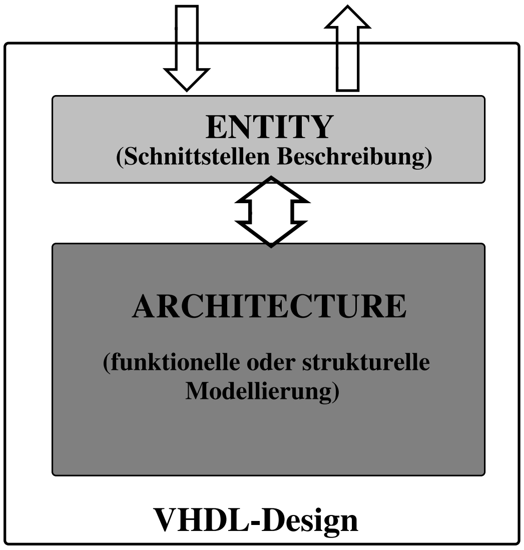


Abbildung 3.2: *Vereinfachte Form eines VHDL-Designs. Die Minimalkonfiguration eines VHDL-Designs besteht aus einer Schnittstellenbeschreibung (*ENTITY*) und der eigentlichen Designeinheit, der* ARCHITECTURE.

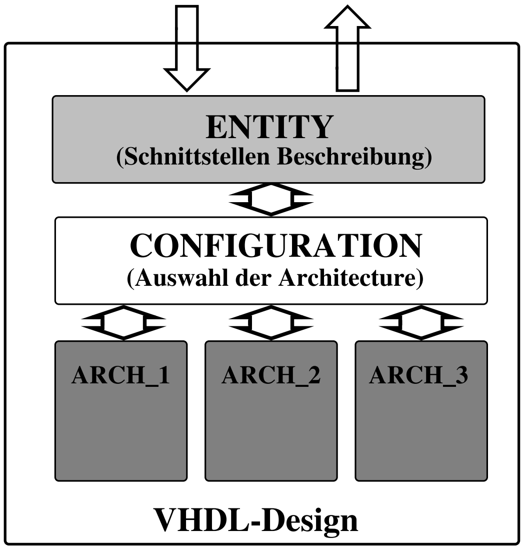


Abbildung 3.3: *Vollständiger Aufbau eines VHDL-Designs. VHDL spezifiziert zusätzlich zum* ENTITY- *und* ARCHITECTURE-*Teil, die* CONFIGURATION. *Die* CONFIGURATION *erlaubt die Auswahl der* ARCHITECTURE *während des Betriebs.*

Die ENTITY definiert die Schnittstelle des Designs. Hier werden die Signale außenstehender VHDL-Designs oder einer *Testbench*[5] übergeben. Die ENTITY ist mit der Anschlussbelegung eines IC vergleichbar, sie beschreibt die Ein- und Ausgänge der Einheit und spezifiziert die Signaltypen. Die ARCHITECTURE beinhaltet das eigentliche Design, z.B. die funktionalen Aufgaben der Applikation. Im Vergleich mit dem IC, entspricht dies in etwa dem internen Gatteraufbau der integrierten Schaltung. Abbildung 3.3 zeigt die vollständige Spezifikation von VHDL. Sie enthält auch die CONFIGURATION. Mit dieser können vom Entwickler verschiedene ARCHITECTURES ausgewählt und verglichen werden. Alle ARCHITECTURES benutzen in einem solchen Fall die gleiche ENTITY-Beschreibung. In vielen Fällen ist eine solche Auswahl

[5]Testbench: spezielle Testumgebung zum Test und zur Simulation von VHDL-Designs

nicht notwendig, das EPP-Design kommt bis jetzt ohne CONFIGURATION-Teil aus und verwendet in allen Instanzen die in Abbildung 3.2 gezeigte Form. Die nachfolgenden Ausführungen gehen deshalb auch im wesentlichen auf Designs ohne CONFIGURATION-Teil ein.

3.3.1 Die ENTITY

In der ENTITY werden die Ein- und Ausgänge eines VHDL-Designs beschrieben. Hauptbestandteil der ENTITY ist die *Port*-Liste.

```
entity D-FlipFlop is
        port    (CLK          : in std_logic;       -- Port Liste
                 D            : in std_logic;       --
                 RESET        : in std_logic;       --
                 Q            : out std_logic;      --
                );
end D-FlipFlop;
```

In der *Port*-Liste werden alle Schnittstellen vereinbart, mit denen Signale in das Design hinein oder heraus geführt werden. Solche *Port*-Signale verhalten sich innerhalb von VHDL wie globale Variablen, sie sind in allen Strukturen einer ARCHITECTURE bekannt und können nicht lokal überdefiniert werden. Neben dem *Port* selbst, muß auch der zugehörige Signal-Typ deklariert werden. Jedem *Port*-Eingang und -Ausgang wird innerhalb der *Port*-Liste ein Signaltyp zugeordnet. Solche Typen können neben dem sehr häufigen Bit-Typ *std_logic*, auch *Integer*-Typen oder sogar ganze Datenbusse in Form von Vektor-Typen (*std_logic_vector*, *signed*, *unsigned*) sein. In obigem Beispiel ist die ENTITY eines D-Flip Flop gezeigt. Die Ein- und Ausgänge sind wie bei der Hardware-Version eines solchen Flip Flops bezeichnet. Der Typ *std_logic* definiert ein *Port*-Signal mit den beiden möglichen Bit-Zuständen 0 und 1. Für einen detaillierten Überblick der *Port*- und Signal-Typen sei die Fachliteratur empfohlen z.B. Ashenden (*Students Guide to VHDL*) [Ashenden 1998].

Die Aufteilung in ENTITY und ARCHITECTURE entkoppelt die Schnittstelle vom operativen Teil des Designs. Dies ist besonders im Hinblick auf eine modulare Entwurfs-Strategie von Bedeutung. Veränderungen innerhalb der ARCHITECTURE lassen sich so relativ leicht herbeiführen. Die Auswirkungen sind vorwiegend lokal auf das geänderte Modul beschränkt.

3.3.2 Die ARCHITECTURE

Die ARCHITECTURE enthält den operativen Teil des VHDL-Designs. Es existiert eine große Ähnlichkeit zu Programmiersprachen, aber auch deutliche Unterschiede. Wesentlich für ein Verständnis von VHDL ist die Unterscheidung der Modellierungstechnik. Dabei sollte zumindest zwischen zwei Modellierungsarten unterschieden werden:

- funktionale Modellierung
- strukturale Modellierung

Unter funktionaler Modellierung versteht man die Beschreibung eines Designs unter Verwendung logischer oder mathematischer Operatoren. Die Addition oder der Vergleich zweier Signale ist z.B. eine Beschreibung in diesem Sinne. Alle Operationen die Signale manipulieren sind funktionale Beschreibungen. Dem gegenüber steht die strukturale Modellierung. Eine strukturale Modellierung beschreibt die Verknüpfung von Schnittstellen mit Hilfe von Signalen. Sie entspricht in etwa der Beschreibung einer Verdrahtung der funktionalen Einheiten.

Die Unterscheidung der Modellierungsart trägt den Erfordernissen Rechnung, wie sie im klassischen Elektronikdesign vorherrschen. Eine elektronische Schaltung ist meist in mehrere funktionale Komponenten untergliedert. Man denke zum Beispiel an integrierte Schaltungen mit spezifischen Aufgaben. Die Funktion dieser Bauteile kann mit VHDL beschrieben werden (funktionale Modellierung). Andererseits müssen die Bauteile oder Komponenten irgendwie miteinander verbunden werden. Dies geschieht in der Elektronik mit Signalleitungen in einem Chip oder auf einem *Board*. Diese Verdrahtung kann ebenfalls von VHDL beschrieben werden (strukturale Modellierung). In der Praxis vermischen VHDL-Designs beide Modellierungsarten. In einer modularen Design-Hierarchie kann eine möglichst weite Aufteilung in funktionale- und strukturale-Designs jedoch zweckmäßig sein und sehr zur Übersichtlichkeit beitragen. Die unterste Schicht besteht dann zumeist vollkommen aus Designs mit funktionalen Aufgaben, während höhere Schichten struktural angelegt sind, und die Verknüpfung der funktionalen Schicht liefern. Für das Design des EPP wurde eine solche Vorgehensweise gewählt.

funktionale Modellierung:

Nachfolgendes Beispiel zeigt die ARCHITECTURE des D-Flip Flops, dessen ENTITY bereits untersucht worden ist[6]:

```
architecture INSIDE of D-FlipFlop

begin
       ---
       -- Verhaltens-Beschreibung eines D FlipFlop
       ---

       d_flipflop:                              -- Prozess Name
       process (CLK,D,RESET)                    -- Sensity List
       begin                                    -- Verhaltens Beschreibung
              if RESET = '1' then               --
                    Q <= '0';                   --
              else                              --
                    if rising_edge(CLK) then    -- Funktion aus Library
                           Q <= D;              --
                    end if;                     --
              end if;                           --
       end process;                             --
end INSIDE;                                     --
```

Bei dieser ARCHITECTURE handelt es sich um die funktionale Modellierung eines D-Flip Flop. Wichtigster Bestandteil dieser ARCHITECTURE ist der PROCESS d_flipflop.

Prozesse spielen eine zentrale Rolle für die funktionale-Modellierung in VHDL. Sie beschreiben prozedurale Vorgänge, also eine Liste mit sequentiell, d.h. nacheinander ablaufenden Befehlen. Eine ARCHITECTURE kann beliebig viele Prozesse enthalten. Obwohl Prozesse intern nur sequentiell durchlaufen werden, kann ihr Aufruf gleichzeitig, also parallel erfolgen. Diese Eigenschaft bildet zum Teil die parallele Struktur der Elektronik ab. Auch in elektronischen Schaltungen können mehrere Prozesse zur Signalverarbeitung gleichzeitig aktiv sein. Die innere Struktur von Prozessen entspricht der höherer Programmiersprachen, der verwendbare Befehlsumfang ist vergleichbar. Eine Vielzahl mathematischer und logischer Operatoren steht zur Verfügung, ebenfalls die Möglichkeit zur Programmierung von Schleifen oder die bedingter Verzweigung. Die Abarbeitung der Befehlsliste geschieht zeilenweise von oben nach unten wie bei einer Programmiersprache, und wird mit jeder Aktivierung des Prozesses neu gestartet. Prozesse werden durch zwei verschiedene Mechanismen aktiviert, beide schließen sich gegenseitig aus. Die Aktivierung

[6] Die Funktion des D-FlipFlops selbst ist hier nicht weiter von Belang und soll nicht näher betrachtet werden.

erfolgt entweder durch eine WAIT-Anweisung innerhalb der Befehlsliste des Prozesses oder mit Hilfe der SENSITY-LIST im Kopf der Prozess-Anweisung.

Das obige Beispiel des D-Flip Flop benutzt eine SENSITY-LIST zur Aktivierung des d_flipflop-Prozesses. Bei jeder Änderung eines der Signale (CLK, D, RESET) wird der Prozess aktiv und beginnt von neuem mit der Abarbeitung der Befehlsliste. Die Befehlsliste des Prozesses (Zeilen zwischen BEGIN und END) enthält zwei bedingte Abfragen (*if*) und zwei Signalzuweisungen (<=). Ein guter Programmier-Stil in VHDL ist es, wenn jedes Signal beim *Reset* in einen definierten Zustand gebracht wird. Im Fall des D-Flip Flops wird das Signal des Ausgang-*Ports* Q deshalb bei aktivem RESET auf 0 gesetzt. Ist RESET inaktiv, so tritt die alternative Bedingung in Kraft. Die Funktion *rising_edge(CLK)* prüft das Signal CLK auf eine ansteigende Flanke. Ist die Bedingung erfüllt, wird das Signal des Ausgang-*Ports* Q auf den Wert des Eingang-*Ports* D gesetzt. Kurz gesagt, wird Q mit jeder ansteigenden Flanke von CLK auf den aktuellen Wert von D gesetzt, falls RESET inaktiv ist. Manche Funktionen, wie die hier verwendete *rising_edge*-Funktion, gehören nicht zum Standard-Sprachumfang von VHDL. Genau wie bei Programmiersprachen ist es aber auch in VHDL möglich, vordefinierte Funktionen aus externen Bibliotheken (*Libraries*) in das Design hinzu zu laden. Auf weitere Details kann hier im Rahmen dieser Ausarbeitung leider nicht eingegangen werden, es sei wieder auf die Fachliteratur verwiesen [Ashenden 1998].

strukturale Modellierung:

Strukturale Modellierung beschäftigt sich nicht mit der Funktion einer Komponente, sondern mit deren Verdrahtung in Form einer Netzliste. Bei den Komponenten handelt es sich selbst um VHDL-Designs, die funktional oder ebenfalls struktural modelliert sind. Im Grunde gleicht die strukturale Modellierung der Verdrahtung von ICs. Dazu werden zunächst die Sockeltypen festgelegt, dann die Verdrahtung der Sockel durchgeführt, und zum Schluss die IC-Typen den Sockel-Typen zugeordnet. VHDL benutzt genau diese Vorgehensweise bei der strukturalen Beschreibung. Die Festlegung des Sockeltyps der Komponenten steht im Deklarationsteil der ARCHITECTURE, die Verdrahtung der Komponenten im Anweisungsteil. Die Komponentenkonfiguration, also die Zuordnung der ICs zu den Sockeltypen, geschieht in der CONFIGURATION des VHDL-Designs. Wie bereits früher angemerkt, kann die CONFIGURATION in vielen Fällen entfallen. Bei der strukturalen Modellierung ist dies immer dann möglich, wenn die Zuordnung der Komponente in der COMPONENT-ENTITY einmalig ist und für alle Zeiten gleich bleibt, also jeder Sockeltyp genau einem IC-Typ gehört. Nachfolgendes Beispiel zeigt die strukturale Modellierung eines RS-Flip-Flop aus zwei NAND2-Gattern[7]. Das NAND2-Gatter existiert dabei bereits als funktionales Design innerhalb einer Bibliothek mit dem Namen *WORK*. Weil der Sockel-Typ für das 'NAND2-IC' in diesem Fall zweimal verwendet wird, ist eine Zuordnung des IC-Typ zum Sockel-Typ mittels CONFIGURATION erforderlich. Abbildung 3.4 zeigt das Schaltbild des RS-Flip Flop und dessen Aufbau mit Hilfe von NAND2-Gattern.

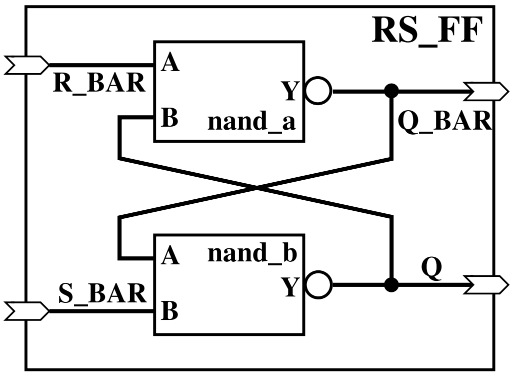


Abbildung 3.4: *Zur strukturalen Modellierung eines RS-FlipFlop mit NAND2-Gattern. Das NAND2-Gatter wird durch ein eigenes VHDL-Design beschrieben. Dieses kann zum Beispiel in einer Bibliothek vorliegen. Die strukturale Modellierung definiert einen Sockeltyp (Component) für das 'NAND-IC' und beschreibt die Verdrahtung der NAND-Sockel zu einem RS-FlipFlop. In der* CONFIGURATION *werden den Sockeln die NAND-ICs zugeordnet [Lehmann et al. 1994].*

[7] Ein NAND2-Gatter ist ein NAND-Baustein mit 2 Signaleingängen (A und B).

```
entity RS_FF is
        port    (R_BAR     : in bit :='0'; -- Port Liste
                 S_BAR     : in bit :='0';
                 Q         : inout bit;
                 Q_BAR     : inout bit
                );
end RS_FF;


architecture STRUCTURAL of RS_FF is
        component NAND2_SOCKET              -- Komponenten Deklaration
               port       (A     : in bit; -- Component Entity
                           B     : in bit;
                           Y     : out bit
                          );
        end component;
 begin
        nand_a: NAND2_SOCKET                -- Komponenten Instantiierung
               port map   (A    =>   R_BAR,
                           B    =>   Q,
                           Y    =>   Q_BAR
                          );
        nand_b: NAND2_SOCKET                -- Komponenten Instantiierung
               port map   (A    =>   Q_BAR,
                           B    =>   S_BAR,
                           Y    =>   Q
                          );
 end STRUCTURAL;


 configuration RS_FF_CONFIG of RS_FF is
        for STRUCTURAL
                for all : NAND2_SOCKET     -- Komponenten Konfiguration
                       use entity WORK.NAND2 (behavioral);
                end for;
        end for;
 end RS_FF_CONFIG;
```

Das Beispiel stammt aus dem Skript '*Schaltungsdesign mit VHDL*' von Lehmann et al. 1994. Die *Component* NAND2_SOCKET (Sockel-Typ) wird zweimal verwendet. In der CONFIGURATION wird allen *Components* (Sockeln) – for all: NAND2_SOCKETS – diesen Typs das gleiche Design (IC) NAND2 der Bibliothek *WORK* – use entity WORK.NAND2 – zugewiesen.

Die strukturale Modellierung eignet sich auch zum Aufbau einer Design-Hierarchie. Die Definition von *Components* bietet ein geeignetes Werkzeug zur Realisierung von Unter-Programmen bzw. Unter-Designs. Die hierarchisch tieferen Designs können dabei als *Component* in ein hierarchisch höheres Design integriert werden. Auf diese Weise kann ein Design in mehrere Ebenen gegliedert werden.

Der tatsächliche Sprachumfang, den VHDL bietet, ist bei weitem umfangreicher und vielseitiger als dies hier darstellbar ist. Für ein tieferes Studium sei nochmal auf die Fachliteratur verwiesen. Besonders empfohlen seien in diesem Zusammenhang die Skripte '*Schaltungsdesign mit VHDL*' [Lehmann et al. 1994] und '*VHDL Kurzbeschreibung*' [Maeder] die im Internet erhältlich sind, außerdem das Buch '*The Student's Guide to VHDL*' [Ashenden 1998].

3.3.3 Entwicklung und Test mit VHDL

Der Design-Vorgang unter VHDL geht wie folgt vor sich: Zuerst wird eine *Working Library* erzeugt (*WORK*-Verzeichnis). Diese ist zunächst leer, in ihr liegen später die neu entworfenen Designs des Projekts. Die Designarbeit selbst geschieht mit Hilfe eines Text-Editors freier Wahl. Manche VHDL-Systeme bieten auch eine vollständig integrierte Designumgebung, mit Editor, *Compiler*, *Debugger* und Simulator. Nachdem das Design entworfen ist, kann es durch Aufruf des *Compilers* übersetzt werden. Das Ergebnis des *Compiler*-Laufs liegt in der *Working-Library* und ist in der Regel eine nur noch maschinenlesbare Code-Liste. Diese Code-Liste kann vom Simulator, oder später von einem Synthese-Werkzeug, verarbeitet werden.

Ziel des Designvorgang ist ein möglichst fehlerfreier Entwurf. Aus diesem Grund besitzen alle VHDL-Systeme einen entsprechenden Simulator. Die Simulation erlaubt die Fehlerbehebung und Verfeinerung des Entwurfs ohne jemals Hardware realisieren zu müssen. Bis das endgültige Design-Resultat vorliegt, sind oft mehrere Design-Simulations-Zyklen erforderlich. Die vollständige Kontrolle über Design, *Debugging* und Simulation innerhalb einer einzigen Softwareumgebung macht den Entwurfsvorgang sehr effizient und kostengünstig. Abbildung 3.5 gibt eine Übersicht über eine VHDL-Designumgebung.

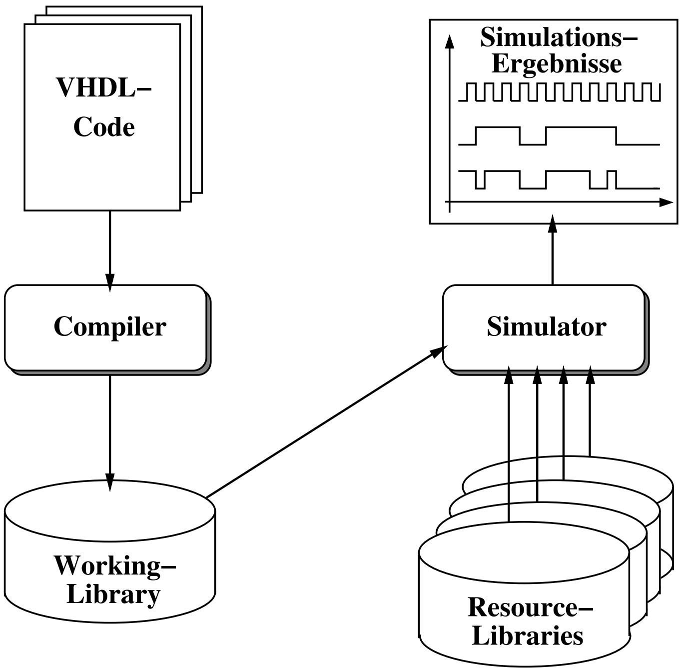


Abbildung 3.5: *Der Design-Vorgang unter VHDL. Zuerst werden die Beschreibungen mit einem Text-Editor freier Wahl erstellt (VHDL-Code). Anschließend wird der Compiler gestartet. Dieser übersetzt das Design in ein maschinenlesbares File, das er in die Working-Library legt. Die Files der Working-Library dienen später zusammen mit vorgefertigten Resource-Libraries dem Simulator als Eingangs-Daten. Mit dem Simulator können Signale der Schnittstellen und interne Signale gleichermaßen analysiert werden.*

Die meisten VHDL-Systeme bieten eine Reihe von vorgefertigten Komponenten und Funktionen innerhalb von Bibliotheken (*Resource Libraries*). Wichtige Bibliotheken sind zum Beispiel die *IEEE.std_logic-_1164* welche eine erweiterte Auswahl von Logik-Typen zur Verfügung stellt, oder die *IEEE.std_logic_arith* für zusätzliche arithmetische Funktionen.

Beim Aufruf des Simulators liest dieser die *Working*- und ggf. die *Resource-Libraries*. Die Signal-Ausgänge des Designs können in einem Signal-*Analyzer* dargestellt werden, in der Regel sogar auch ausschließlich intern verwendete Signale. Ein Signal-*Analyzer* zeichnet die ausgewählten Signale über einen gewünschten Zeitraum auf, und stellt sie graphisch dar. Eine genaue Analyse der Signale und deren zeitlicher Entwicklung gibt dem Entwickler Aufschluss darüber, ob das Design gemäß den Vorgaben funktioniert oder Änderungen erforderlich sind.

Ein Problem der Simulation mit *Analyzern* ist die Erzeugung sinnvoller Stimuli für die Eingangssignale. Oft sollen VHDL-Module, die Bestandteil eines größeren Designs sind, einzeln auf ihre Funktion getestet werden. Die Eingänge solcher Module benötigen dann ganz spezielle Stimuli, in einer strengen zeitlichen Abfolge. In diesem Fall sind die Standard-Stimulis die VHDL-Simulatoren von Haus aus anbieten (meistens periodische *Clock*-Signale, bzw. Teile und Vielfache davon) oft wenig zweckmäßig. Auch beim Test fertiger, sehr komplexer Designs ist dieses Problem gegenwärtig.

Einen Ausweg aus diesem Dilemma bietet meistens nur die Entwicklung einer speziell angepassten Testumgebung, einer sogenannten *Testbench*. Die *Testbench* ist selbst ein VHDL-Design mit der Aufgabe, erforderliche Signalstimuli für die Simulation des zu testenden VHDL-Designs bereitzustellen. Dazu emuliert diese die Signal-Stimuli, genauso wie sie später für die tatsächlichen Eingangs-Signale erwartet werden. Soll sogar mit echten Messdaten getestet werden, so muß die *Testbench* in der Lage sein, diese aus einer Festplattendatei des Test-Rechners zu lesen und entsprechend formatiert an das zu testende Design weiter zu geben. In solchen Fällen werden dann zumeist große Datenmengen durch das Test-Design geleitet. Eine Analyse mittels Signal-*Analyzer* ist dann oft wenig praktikabel. Die Ausgangs-Daten des Test-Designs werden in solchen Fällen von der *Testbench* direkt auf Festplatte zurückgeschrieben, und später mit getrennter Software analysiert.

Ein Problem der *Testbench* ist die schlechte Adaptionsfähigkeit. In den meisten Fällen muß sie für jedes neu zu testende VHDL-Design neu geschrieben, oder zumindest stark modifiziert werden. Für die Entwicklung des EPP wurden alle VHDL-Bestandteile einzeln, und am Schluss das Haupt-Design mit einer *Testbench* getestet.

Kapitel 4

Event Pre-Prozessor (EPP)

Wie andere Missionen der Röntgenastronomie auch, unterliegt der CMI starken Einschränkungen bezüglich der An-Bord-Datenauswertung. Eine optimale Situation entstünde, wenn Röntgen- und Gamma-Quellen von bodengebundenen Observatorien aus direkt beobachtbar wären. Weder die Datenübertragung, noch begrenzte Computerkapazitäten würden dann zu Problemen führen, und es wäre möglich alle Daten des CMI zu verarbeiten und zu analysieren. Leider ist dies für Röntgenmissionen ein unerfüllbarer Traum. Die Beobachtung von Röntgenquellen kann nur vom Weltraum aus, oder mit hochfliegenden Atmosphärenballons gelingen (vgl. Abb.1.1). In beiden Fällen entstehen Bedingungen, unter denen die Telemetrie der Daten zu starken Kompromissen zwingt. In der Regel verhindern mangelnde Übertragungskapazitäten die Übermittlung der vollständigen Messdaten. Für ein Satellitenprojekt wie CMI wird deshalb ein Teil der Daten bereits an Bord verarbeitet. Die Telemetrie überträgt dann lediglich Daten von wissenschaftlichem Interesse. Leider besitzen weltraumqualifizierte Computer noch nicht einmal annähernd die Leistung und Speicherkapazität, wie sie bei bodengebundenen Systemen heute erreichbar sind. Weltraum-Elektronik unterliegt starken Beanspruchungen durch extreme Temperaturen, das Weltraumvakuum und harte Strahlung[1]. Aus diesem Grund müssen Bauteile für einen Weltraum-Einsatz qualifiziert sein. Hochkomplexe Mikroprozessoren der neusten Generation, sind meist von den Anforderungen solcher Spezifikationen, weit entfernt. Sie würden unter Weltraumbedingungen nicht arbeiten, oder bereits nach kurzer Zeit zerstört werden. Die verfügbaren weltraumqualifizierten Computerbausteine entsprechen zwar den verschärften technischen Anforderungen, besitzen in der Regel jedoch nur Rechenleistungen, wie sie bei bodengebundenen Systemen vor etwa 10 Jahren erreichbar waren. Insbesondere für sehr schnelle, zeitkritische Berechnungen kommen daher bis zum heutigen Tag fest verdrahtete ASICs[2] zum Einsatz. Der in dieser Arbeit vorgestellte *Event-Pre-Processor* stellt ein ASIC-Design in diesem Sinne dar.

4.1 Aufgaben des EPP

Der EPP hat folgende Aufgaben, die anschließend im einzelnen besprochen werden:

- Fehlerkorrektur

- Datenvorverarbeitung, Berechnung grundlegender Daten

[1] Harte Strahlung kann zu Schäden an feinen Strukturen von Mikroprozessoren und Speichern führen. Durch Wechselwirkung, innerhalb der Isolierschichten des Halbleiters, entstehen gefährliche Leck-Ströme, die nach kurzer Zeit zur Zerstörung des Bauteils führen können.

[2] ASIC steht für *Application Specific Integrated Circuit*. Es handelt sich dabei um – für die Anwendung speziell entworfene – fest verdrahtete integrierte Schaltungen.

- Datenreduktion und Störungsunterdrückung

Der EPP ist das Verbindungsstück zwischen der Detektorelektronik auf der einen Seite und einer leistungsfähigen Datenverarbeitungselektronik (DPU) auf der anderen. Aufgabe des EPP ist nicht die ausführliche wissenschaftliche Analyse, sondern die schnelle Bereitstellung wichtiger Schlüsselinformationen.

4.1.1 Fehlerkorrektur

Verstärkerfehler:

Der Detektor des CMI besitzt insgesamt 1024 unabhängige Streifen-Elektroden und jede der Elektroden hat einen eigenen Signalverstärker. Erfahrungsgemäß produzieren die Verstärker Fehler in der Linearität und im *Offset* des Signals. Die Fehler treten für jeden Elektrodenstreifen individuell auf. Eine wichtige Aufgabe des EPP ist deshalb die Korrektur dieses Fehlers mit Hilfe einer programmierbaren Eichtabelle (*Look-Up-Table*).

Die Übertragungskennlinie eines Verstärkers sollte das Eingangssignal idealerweise linear auf das Ausgangssignal abbilden. In diesem Fall ist die Übertragungskennlinie eine Grade, die den Ursprungspunkt schneidet und deren Steigung den Grad der Verstärkung angibt. In Abbildung 4.1 ist eine ideale Verstärkerkennlinie für eine Verstärkung von Zwei gezeigt.

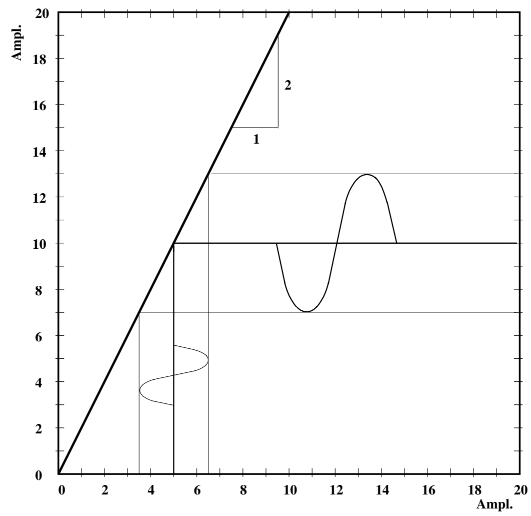


Abbildung 4.1: *Ideale Übertragungskennlinie. Das Signal wird in diesem Beispiel um Faktor zwei verstärkt. Leider ist die Übertragungskennlinie in der Realität niemals eine Grade.*

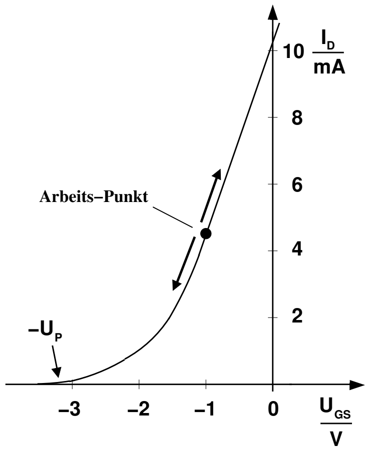


Abbildung 4.2: *Die Kennlinie eines FET hat einen annähernd parabelförmigen Verlauf. Eine lineare Signalübertragung ist nur dann möglich, wenn der Arbeitsbereich in den am wenigsten gekrümmten Bereich gelegt wird, und wenn die Signalamplituden sehr klein sind.*

Reale Verstärker besitzen jedoch keine idealen Eigenschaften, die Kennlinien weisen leichte Nichtlinearitäten auf. Ebenfalls unvermeidbar ist ein Signal-*Offset*. Die in modernen Verstärkerbausteinen eingesetzten Halbleiterverstärker basieren auf dem Transistoreffekt und sind im allgemeinen Verstärker die auf dem Prinzip des Feldeffekt-Transistors (FET) beruhen. Die Kennlinie eines Feldeffekt-Verstärkers ist in Abbildung 4.2 gezeigt, sie ist annähernd parabelförmig[3], ein linearer Betrieb ist nur näherungsweise und für kleine Signale möglich.

[3] zwischen Eingangsspannung U_{GS} und Ausgangsstrom I_D gilt der Zusammenhang:
$I_D = \text{const} * [1 + (U_{GS}/U_P)]^2$

Die Mängel der Verstärkung können nach der Digitalisierung erheblich besser bekämpft werden, als dies mit den Mitteln der Analogtechnik möglich wäre. Deshalb verfügt der EPP über eine Stufe zur Amplituden-Korrektur. Bei der Signalverstärkung sind grundsätzlich zwei Fehler zu unterscheiden:

- Offsetfehler: Eine Verschiebung des Signalnullpunktes um eine kleinen Betrag. Dieser Fehler kann durch eine entgegenwirkende *Offset*-Korrektur behoben werden.
- Linearitätsfehler: Der Linearitätsfehler wird durch die Transistorkennlinie des Verstärkers verursacht. Er wirkt auf die Steigung und die Krümmung der Übertragungskennlinie. Beide Effekte können durch Multiplikation mit geeigneten Korrekturwerten weitgehend ausgeglichen werden.

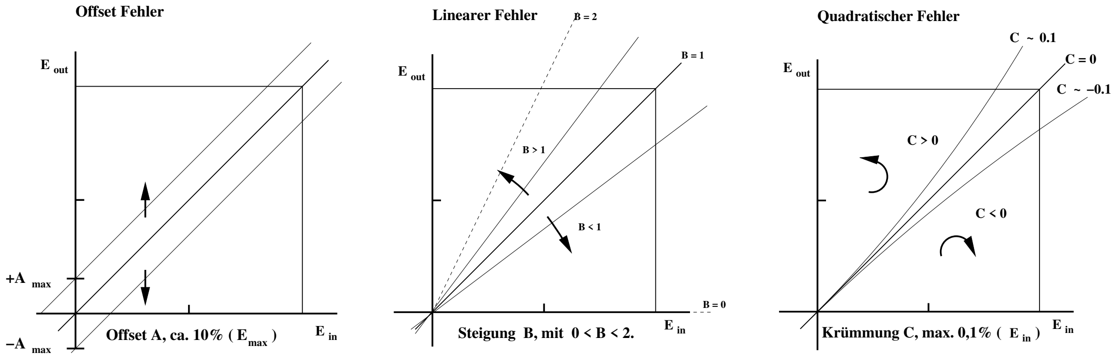


Abbildung 4.3: *Einfluss der Amplituden-Korrektur auf die Übertragungskennlinie. Signalverstärker verursachen neben Offset-Fehlern auch Verzerrungen auf dem Signal. Diese Verzerrungen Können mit einer geeigneten Korrektur neutralisiert werden. Dazu müssen neben dem Offset auch die Steigung und Krümmung des Signals verändert werden. Die Abbildung zeigt den Einfluss der drei Werte A,B und C auf die Kennlinie der Korrektur.*

Die Korrektur der Linearität und des *Offset*-Fehlers kann durch eine einfache quadratische Gleichung beschrieben werden:

$$E_{out} = A + BE_{in} + CE_{in}{}^2 \tag{4.1}$$

Gleichung 4.1 beschreibt die Abbildung des Eingangsignals E_{in} auf das Ausgangsignal E_{out}. Der Faktor A dient der *Offset* Korrektur, er legt den Signal-Nullpunkt fest. A kann sowohl positive als auch negative Werte annehmen, sein Wertebereich sollte ca. 10% des Eingangsignals E_{max} betragen. Der Multiplikator B wirkt auf die Steigung, Multiplikator C auf die Krümmung der Kennlinie. Beide können zur Korrektur der Nichtlinearitäten verwendet werden. Wert B nimmt dazu nur positive Werte zwischen 0 und 2 an, Wert C kann negative und positive Werte annehmen und muß zudem außerordentlich klein sein, da er mit dem Quadrat von E_{in} multipliziert wird. Die Wirkung der Korrekturen auf die Übertragungskennlinie ist in Abbildung 4.3 zu sehen.

Die Verzerrungen sind in der Regel sehr gering, der *Offset*-Fehler wird erwartungsgemäß im Prozentbereich liegen, der Linearitätsfehler kleiner ein Promille der Eingangs-Amplitude betragen. Die technischen Details der Korrektur, sowie die Bit-Darstellung der Korrektur-Werte A, B und C wird in Kapitel 4.3.3 beschrieben.

Tiefenfehler:

Trifft ein Röntgenphoton auf den CZT-Detektor, so erzeugt es dort ein Photo-Elektron. Das Photo-Elektron ionisiert das Halbleitermaterial und erzeugt in einer Kaskade Elektron-Loch-Paare. Die entstehenden freien Elektronen werden von der Anode, die Löcher von der Kathode abgesaugt. Die Ladungsmenge aller Elektronen ist der Energie des Photons proportional und wird als Gesamtenergie des Ereignisses ausgewertet, vergleiche hierzu Kapitel 2.2.2. Leider können nicht alle Elektronen abgeleitet werden, es existiert ein geringer Signal-Schwund. Dieser hängt mit der Elektrodenanordnung und der Reinheit des Detektors zusammen. Wie bereits in Kapitel 2.2.2 beschrieben, besitzt der Detektor Steuerelektroden, die das elektrische Feld auf die Anodenanschlüsse des Detektors fokussieren. Der Einsatz von Steuer-Elektroden erhöht die Signalausbeute, da sich die Ladungswolke auf weniger Elektroden verteilt, und der Ladungsabfluß in die Elektroden-Zwischenräume reduziert ist.

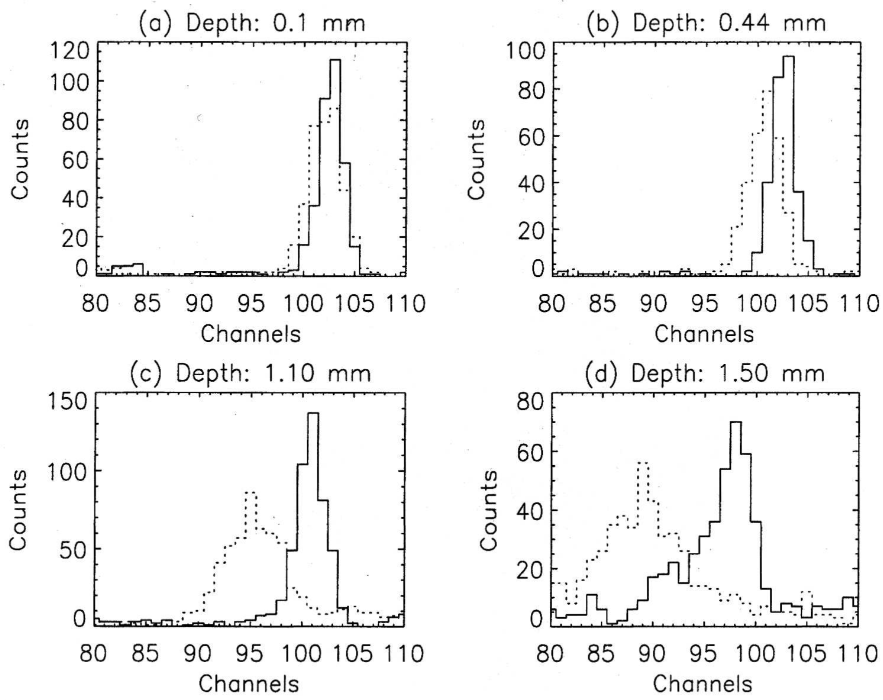


Abbildung 4.4: *Abhängigkeit des Signals vom DOI. Die Abbildung zeigt zwei Spektren einer* ^{57}*Co Quelle. Das durchgezogene Spektrum wurde mit dem Kollimator der Quelle direkt über der Anode aufgenommen, das gestrichelte mit dem Kollimator über der Steuerelektrode. Je tiefer die Wechselwirkung stattfindet, desto größer wird der Ladungseinfang der Steuerelektrode, und desto kleiner die Summe des Anodensignals [Kalemci et al. 2000].*

Selbst bei der Verwendung von Steuerelektroden bleibt jedoch ein geringer Signalschwund bestehen. Dieser hängt zum einen mit unvermeidbaren Rekombinationen im Halbleiter, und zum anderen mit dem Ladungseinfang der Steuerelektroden selbst zusammen. Beide Effekte sind von der Wechselwirkungstiefe, dem DOI (*Depth Of Interaction*) abhängig, und kompensieren sich etwas. Während die Rekombination mit der Weglänge zur Anode zunimmt, also mit der DOI abnimmt, steigt der Ladungseinfang der Steuerelektroden mit zunehmender DOI stark an. Abbildung 4.4 zeigt den Zusammenhang, zwischen DOI und Signalschwund [Kalemci et al. 2000]. Dazu ist die Anodensumme zweier Messungen mit einer ^{57}Co-Quelle dargestellt. Bei einer Messung (durchgezogene Linie) befindet sich der Kollimator der Quelle direkt oberhalb der Anoden-Elektrode, bei der anderen (gestrichelt) direkt oberhalb der Steuerelektrode. Gut erkennbar ist, wie der Ladungseinfang der Steuerelektrode mit steigendem DOI zunimmt. Die Verschiebung zu tieferen Energien und Verbreiterung der Anodensumme (gestrichelte Kurve) machen deutlich, wie negativ sich der Effekt auf die Energieauflösung des Detektors auswirkt. Je tiefer die Wechselwirkung stattfindet, desto größer wird der Ladungseinfang der Steuerelektrode, und desto kleiner die Summe des Anodensignals bei der Energieauswertung. Damit reduziert sich die Energieauflösung des Detektors.

Der EPP hat die Aufgabe den Tiefenfehler zu korrigieren. Dazu setzt er das Signal von Anode und Kathode zueinander in Beziehung. Der Quotient ergibt ein Maß für die Tiefe der Wechselwirkung bzw. den Abstand von der Anode, den DOI-Wert. Mit Hilfe des DOI-Wertes kann unter Verwendung eines *Look-Up-Tables*[4], der Signalschwund tiefenabhängig korrigiert werden.

4.1.2 Datenvorverarbeitung, Berechnung grundlegender Daten

Der EPP führt eine ganze Reihe von elementaren Berechnungen durch. Er stellt damit schnell wichtige Basis-Informationen für weitere Entscheidungen zur Verfügung.

Packet Assembly:

Das Protokoll der Datenübertragung zwischen Detektor und EPP ist zum gegenwärtigen Zeitpunkt noch nicht festgelegt, spezifiziert ist bisher nur der Inhalt der Daten-Pakete. Die Elektronik des Detektors sendet die Daten in Form sogenannte *Hits*. Jeder *Hit* enthält die vollständige Information einer Elektrode. Dies sind Signalamplitude (14 Bit ADC), Elektrodennummer (7 Bit *Subchannel*), Elektrodentyp (3 Bit *Type*), sowie ein Zeitstempel (16 Bit *Timestamp*), vergleiche Abbildung 4.5.

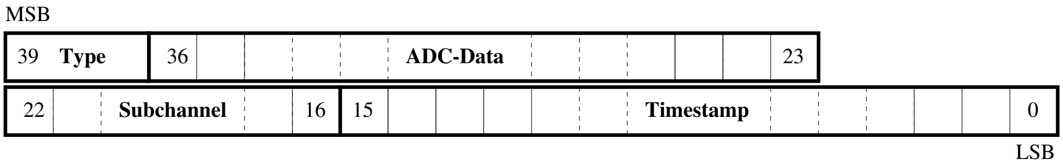


Abbildung 4.5: *Aufbau eines Hit wie er von der Detektorelektronik an den EPP übertragen wird. Jeder Hit enthält das Amplituden-Signal einer Elektrode, den Elektroden-Typ, die Elektroden-Koordinate und einen Zeitstempel .*

Der EPP erzeugt aus den *Hits* die sogenannten *Event-Packets*. Ein *Event-Packet* enthält die Information all derjenigen Elektroden, die im Zeitraum einer Wechselwirkung ein Signal ableiten. Weil der Detektor diese Informationen als Elektroden-*Hits* sendet, setzt sich ein *Event-Packet* zunächst aus diesen *Hit*-Informationen zusammen. Es wird erwartet, daß pro Ereignis maximal 4 Elektroden ein Signal ableiten können. Jedes *Event-Packet* enthält eine Zeitmarke, die sich aus den Zeitmarken der einzelnen *Hits* errechnet und den Zeitpunkt charakterisiert, an dem das Ereignis stattgefunden hat. Bei der Erzeugung der *Event-Packets* untersucht der EPP die Gültigkeit der *Hit*-Informationen. Nur wenn die Zeitstempel der einzelnen *Hits* innerhalb eines definierten Zeitfensters liegen, werden diese aus dem selben Ereignis stammen. Sind die Zeitstempel zu unterschiedlich, wird zwar ein *Event-Packet* erzeugt, dieses aber als ungültig gekennzeichnet. Ungültige *Event-Packets* werden in einer späteren Stufe des EPP verworfen. Neben der zeitlichen Analyse wird auch die Ortsinformation der *Hits* untersucht. Stammen die *Hits* von nicht direkt benachbarten Elektroden, so stammen sie aus unterschiedlichen Ereignissen. Auch in diesem Fall wird ein *Event-Packet* mit ungültigem Veto-*Flag* erzeugt.

[4] Ein *Look-Up-Table* ist eine Tabelle mit Korrekturwerten die eine Koordinaten- oder Tiefenabhängige Korrektur zulassen.

Die Maximalzahl der *Hits* für ein gültiges Ereignis ist derzeit mit vier *Hits* spezifiziert. Tabelle 4.1 gibt eine Auflistung der gültigen Kombinationen:

Anoden	Kathoden
1	1
1	2
2	1
2	2

Tabelle 4.1: *Gültige Hit-Kombinationen*

Kombinationen, die in Tabelle 4.1 nicht enthalten sind, erzeugen ein ungültiges *Event-Packet*. Die Maximalzahl der *Hits* hängt von der Detektor-Technologie ab. Liegen die Elektroden sehr nahe beieinander und ist die Ausdehnung der Ladungwolke innerhalb des Detektors relativ groß, so können auch leicht mehr als 4 Elektroden an der Signalableitung beteiligt sein. Detektor-Parameter, welche die Ausdehnung beeinflussen, sind die Absorptionslänge des Photons (Halbleitermaterial), die Dicke der Detektorplatte und der Streifenabstand der Elektroden.

Bestimmung der Gesamtenergie:

Ein Röntgen-Photon erzeugt im Detektor ein Photoelektron. Dieses ionisiert, um den Ort seiner Entstehung, eine Wolke aus Elektron-Loch-Paaren. Dabei freigesetzte Ladungsträger werden mit Hilfe einer Saug-Spannung von den Elektroden abgeleitet und erzeugen ein elektrisches Signal. Die Ladungsmenge, also die Signalstärke ist der Energie des Photons proportional. Durch Addition der Signale aller beteiligten Anodenleitungen kann deshalb die Gesamtenergie der Wechselwirkung, und damit die Energie des Röntgen-Photons, bestimmt werden.

Bestimmung der Detektor Koordinaten:

Von großer Bedeutung ist die Ortsbestimmung der Wechselwirkung auf der Detektorplatte. Aus den Detektor Koordinaten und der Geometrie der Aperturmaske kann die Position der Quelle am Himmel berechnet werden. Die Statistik liefert die Position umso genauer, je mehr Photonen der Quelle im Detektor gemessen werden. Je stärker die Quelle ist, desto genauer die Position.

Wie bereits in Kapitel 2.2.2 beschrieben, ist der Detektor segmentiert. Jedes der 16 Segmente ist auf beiden Seiten mit Elektrodenstreifen bestückt. Die Kathodenstreifen auf der Oberseite sind zu den Anodenstreifen der Unterseite um 90 Grad gedreht und bilden mit diesen 2-dimensionale Koordinatensysteme. Die Einteilung dieser Systeme ist durch die Elektrodenbreite und den Elektrodenabstand gegeben. Die bisherige Spezifikation sieht für alle Elektroden einen Abstand von 1 mm vor. Bei einer mittleren Wechselwirkungslänge der Röntgen-Photonen von 25 μm (für 60 keV) [Kalemci et al. 2000] und der Ausdehnung der Ladungswolken auf bis zu 0,25 mm, sind pro Wechselwirkung deshalb in keinem Fall mehr als drei Anoden- und Kathodenstreifen an der Signalableitung beteiligt.

Als Ereignisposition gilt der Kreuzungspunkt der Anoden- und Kathoden-Streifen mit dem stärksten Signal. Die Streifennummern der Elektroden ergeben die Koordinaten der Wechselwirkung auf dem Detektorsegment, die sogenannte *Hit*-Adresse. Jedes der 16 Detektorsegmente hat 1024 solcher *Hit*-Adressen. Die Angabe allein der *Hit*-Adresse reicht für die Ortsangabe auf dem Detektor nicht aus, zusätzlich muß noch das betreffende Segment adressiert werden. Der gesamte Detektor ist als Array aus 4 x 4 Segmenten aufgebaut. Es liegt daher nahe, auch die Segmente in einem System mit 4 x 4 Koordinaten zu beschreiben. Diese Segment-Adresse kann zusammen mit der *Hit*-Adresse jedes Ereignis auf dem Detektor eindeutig adressieren.

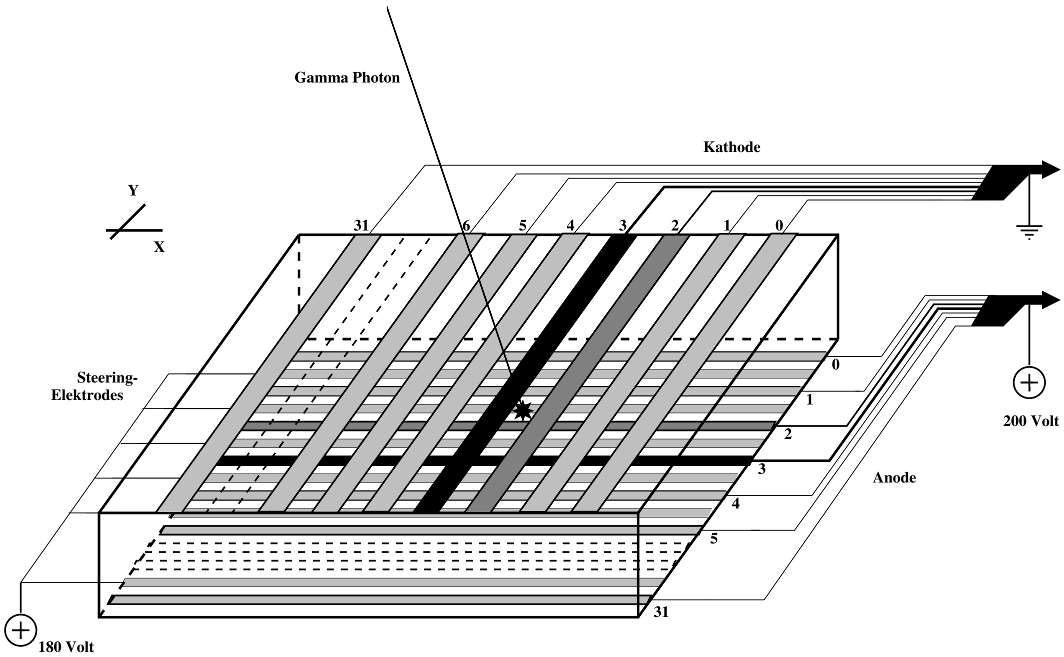


Abbildung 4.6: *Wechselwirkung eines Gamma-Quants innerhalb eines der 16 CZT-Detektorsegmente. Die beiden Elektroden von Anode und Kathode, die das Signal ableiten sind dunkelgrau und schwarz unterlegt. Auf jeweils einer Elektrode (schwarz) ist das Signal am stärksten. Diese Elektroden definieren die Position der Wechselwirkung auf dem Detektorsegment.*

Für die reine Ortsbestimmung ist ein einfaches Koordinatensystem, das sich über den gesamten Detektor erstreckt, besser geeignet. Hierzu kann der Detektor in Elektroden-Reihen und -Spalten unterteilt werden, die Segmentierung wird dabei vernachlässigt. Auf jeder Reihe, bzw. Spalte sind dann die Elektroden von 4 nebeneinander, bzw. 4 untereinander liegenden Segmenten zusammengefasst (siehe Abbildung 4.7). Es entsteht ein Koordinatensystem aus 128 Anoden und 128 Kathoden-Koordinaten, das mit 2 x 7 Bit adressierbar ist. Die Adresse eines Ereignisses, die *Event*-Adresse, ist einfach der Schnittpunkt der Anode und Kathode mit dem maximalen Signal. Die Adresse innerhalb dieses Systems reicht für die Berechnung der Position am Himmel und für die Bilderzeugung vollkommen aus.

Im Gegensatz zur Positionsbestimmung verlangt die Amplitudenkorrektur die Adressierung jeder einzelnen Elektrode (Adressierung mit Segment- und *Hit*-Adresse). Dies ist erforderlich, da jede der 1024 Elektroden einen eigenen Verstärker besitzt und der Fehler individuell korrigierbar sein muß. Das Ortsdatensegment (9Bit) zusammen mit dem Elektrodentyp (1Bit – Position des *Hit* im *Event-Packet*) ergibt die 10 Bit breite Adressinformation, die zur Adressierung des LUT erforderlich ist. Nach der Amplituden-Korrektur kann die nicht weiter benötigte Segment-Adresse vernachlässigt werden. Das Ortsdatensegment der *Hits* enthält dann nur noch die 7 Bit breite *Event*-Adresse.

Datensicherung:

Leider ist eine kontinuierliche und vorausplanbare Untersuchung der *Gamma-Ray Bursts* nicht möglich, da diese Phänomene nur sporadisch auftreten und innerhalb von Sekunden wieder vorbei sind. Die Position von GRBs am Himmel ist bis heute nicht vorhersagbar. Aus diesem Grund ist eine Aufzeichnung zwingend notwendig, die eine detaillierte nachträgliche Untersuchung der Daten ermöglicht. Wichtige Informationen für die Aufklärung des Phänomens erhofft man sich auch von der Untersuchung der unmittelbaren Vorgeschichte vor einem Ausbruch. Der EPP besitzt zu diesem Zweck eine Ring-Speicher-Einheit. Hauptbestandteil der *Ring-Buffer-Unit* ist ein Halbleiterspeicher der diejenigen Ereignis-Pakete aufnimmt, die im Sinne der Vorverarbeitung gültig sind. Der Ring-*Buffer* arbeitet ähnlich einem Endlosband, ein Überlaufen

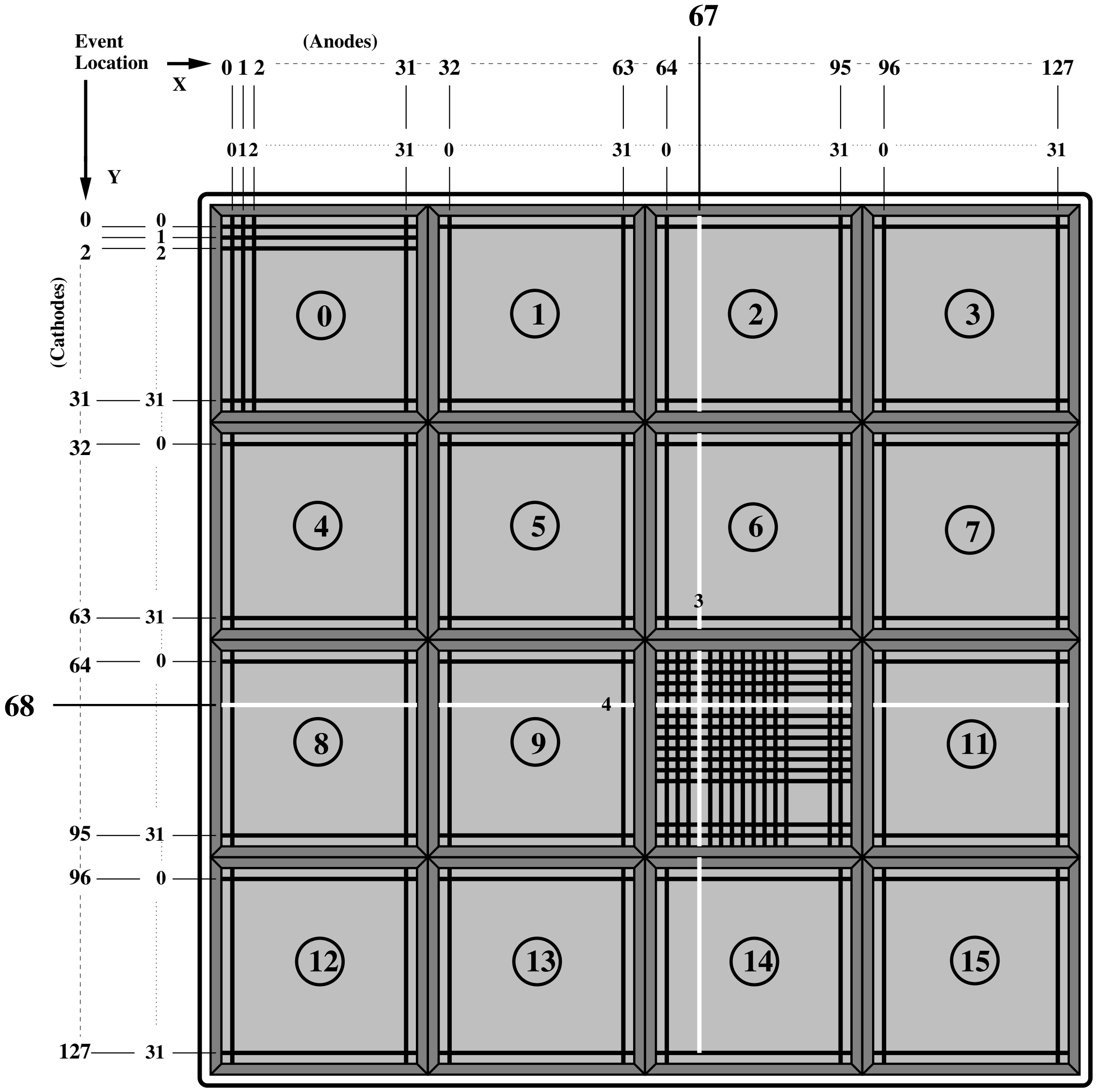


Abbildung 4.7: *Wechselwirkung eines Photons innerhalb des CZT-Detektors. Der Detektor besteht aus 16 Segmenten, von denen jedes ein 32 x 32 Koordinatensystem besitzt. Für die Adressierung eines Wechselwirkungs-Ereignisses wird ein Koordinatensystem verwendet, das sich über den gesamten Detektor erstreckt. Hierzu wird der Detektor in Elektroden-Reihen und -Spalten unterteilt. Auf jeder Reihe und Spalte sind dann die Elektroden von 4 nebeneinander, bzw. 4 untereinander liegenden Segmenten zusammengefasst. In der Abbildung, bildet z.B die Anode 3 von Segment 2, 6, 10 und 14 die X-Koordinate 67 des Detektors, oder Kathode 4 von Segment 8, 9, 10 und 11 die Y-Koordinate 68. Dieses 128 x 128 Koordinatensystem reicht für die Positions-Beschreibung auf dem Detektor aus. Für den Look-Up-Table wird die Adressierung jedoch aufwendiger, dort muß jede Elektrode einzeln adressierbar sein.*

ist nicht möglich! Wenn das Fassungsvermögen erreicht ist, werden die zuerst gespeicherten Daten der Reihe nach wieder überschrieben und gehen verloren. Die Ereignisdauer von *Gamma-Ray Bursts* umfaßt Zeiträume zwischen einigen Sekunden und Minuten. Der Speicher wird so dimensioniert, daß der Signalverlauf eines typischen *Bursts* vollständig in ihm Platz findet. Da die Aufzeichnung ständig mitläuft, ist im Falle eines *Burst-Triggers* auch die Vorgeschichte des *Burst* verfügbar. Der Ring-*Buffer* kann in diesem Fall angehalten und gesperrt werden, bis die Daten zu Boden übertragen und gesichert sind.

Erzeugung des *Burst Triggers*:

Die wichtigste Aufgabe des CMI ist die Suche nach *Gamma-Ray Bursts*. Zu diesem Zweck besitzt der EPP einen sogenannten *Burst*-Detektor. Der *Burst*-Detektor untersucht ständig die aktuelle Zählrate und vergleicht diese mit dem Mittelwert des Röntgenhintergrundes. Wird eine signifikante Abweichung von der Hintergrundzählrate festgestellt, löst der *Burst*-Detektor den *Burst-Trigger* aus. Der *Burst-Trigger* ist ein Alarmsignal, das der Bodenstation die Entdeckung eines neuen GRB mitteilt. Diese kann dann ihrerseits entscheiden (automatisch oder manuell), ob sie den Ring-*Buffer* sperrt und die Forschungsgruppen anderer Missionen und das GCN[5] informiert.

Für die Arbeit des *Burst*-Detektors ist die Tatsache entscheidend, daß die Zählrate während eines GRB signifikant über die Hintergrundzählrate ansteigt. Die Zählraten werden dafür innerhalb konstanter Zeitintervalle genommen, es stehen verschiedene Werte zur Verfügung. Die Bodenstation stellt das Intervall je nach gewünschter Empfindlichkeit ein.

Die Ereignisse, die der Ereigniszähler ermittelt, sind völlig unabhängig voneinander, ihr Eintreten hat eine zeitunabhängige Wahrscheinlichkeit. Wird für jedes Zeitintervall die Anzahl der Ereignisse notiert und der Vorgang hinreichend oft wiederholt, so erhält man den Durchschnittswert ν. Es kommt natürlich auch vor, daß in einem Zeitintervall deutlich weniger oder mehr Ereignisse gezählt werden, wenn auch um so seltener, je größer die Abweichung vom Mittelwert ist. Wenn wir die Anzahl der Ereignisse mit n bezeichnen und die Häufigkeit der Zählraten mit z_n, dann kann die Verteilung der Häufigkeit durch die Poisson-Gleichung 4.2 beschrieben werden.

$$z_n = Z\frac{\nu^n}{n!}e^{-\nu} \tag{4.2}$$

Betrachten wir einen längeren Zeitraum, dann ist Z die Anzahl aller *Events* des Zeitraumes und wir finden z_n Intervalle mit der Zählrate n. Eine Verteilung gemäß Gleichung 4.2 ist in Abbildung 4.8 gezeigt. Das Maximum stellt den Mittelwert und damit den Gamma- bzw. Röntgenhintergrund der Messung dar. Es sind Abweichungen von diesem Hintergrund die uns interessieren.

Die mittlere Schwankung oder Standard-Abweichung σ ist diejenige Abweichung, die nur bei einem Bruchteil $e^{-1} \approx 0,37$ der Intervalle überschritten wird. Sie entspricht ungefähr den Wendepunkten der Verteilungskurve in Abbildung 4.8. Für die Standard-Abweichung der Poisson-Verteilung gilt:

$$\sigma = \sqrt{\nu} \tag{4.3}$$

Ein *Gamma-Ray Burst* liegt im allgemeinen dann vor, wenn die Zählrate n, der gemessenen gültigen Gamma-Ereignisse, um mehr als 5 σ von der Hintergrund-Zählrate ν abweicht.

[5]GCN: GRB *Coordinated Network*

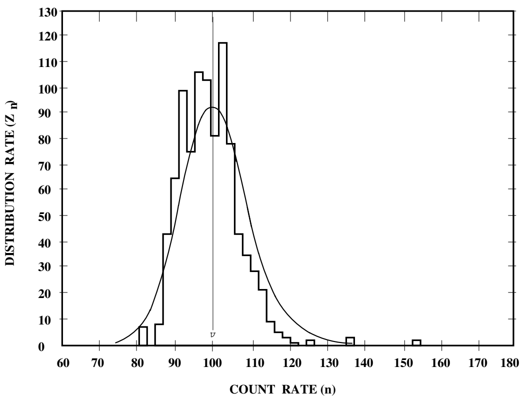


Abbildung 4.8: *Beispiel für eine Poisson-Verteilung. Wird in einem Histogramm, die Häufigkeit gegenüber der Zählrate aufgetragen, so entspricht diese einer Poisson-Verteilung. Das Maximum der Verteilung stellt den Mittelwert ν der Zählraten und somit den Hintergrund der Messung dar.*

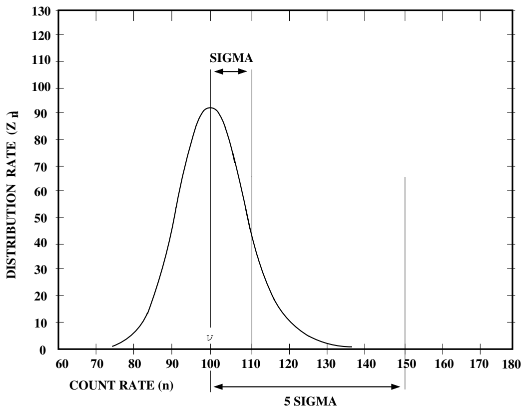


Abbildung 4.9: *Standardabweichung der Poisson-Verteilung. Die Standardabweichung (Sigma) ist diejenige Abweichung, die von 37% der Messungen überschritten wird. Der Burst-Detektor wertet Zählratenanstiege mit einer Abweichung größer 5 σ als potentielle Gamma-Ray Bursts.*

Zur Erzeugung des *Burst-Triggers* besitzt der EPP vier wählbare Zeitintervalle in denen die Ereignisse gezählt werden. Diese Zeitintervalle, werden als Tor-Zeit oder *Gate-Time* bezeichnet, die Auswahl der Tor-Zeit ist von der Bodenstation programmierbar. Es stehen Zeiten von 1/8 Sekunde, 1/2 Sekunde, 1 Sekunde und 2 Sekunden zur Verfügung. Der Ereigniszähler zählt alle eingehenden gültigen Gamma-Ereignisse innerhalb der Tor-Zeit und gibt das Ergebnis als Zahl aus. Jede dieser Zahlen wird dann auf eine 5 σ Abweichung zur Hintergrundzählrate untersucht und anschließend für eine neue Mittelwertbildung herangezogen. Der Mittelwert, also die Hintergrundzählrate wird ständig aus den letzten 64 Einzelzählraten ermittelt.

4.1.3 Datenreduktion und Störungsunterdrückung

Die Suche nach Röntgen-Ereignissen, die tatsächlich aus einem *Gamma-Ray-Burst* stammen, gleicht in gewissem Sinne der berüchtigten Suche nach der Nadel im Heuhaufen. Nicht nur das *Gamma-Ray-Bursts* seltene Erscheinungen sind und nur wenige Sekunden andauern, die gesuchten Signale befinden sich auch in einem Störhintergrund vielfältiger Herkunft.

Die Störquellen sind kosmischer Herkunft oder Quellen in unserem eigenen Sonnensystem, insbesondere die Sonne selbst[6]. Der Störhintergrund setzt sich deshalb aus folgenden Quellen zusammen:

- elektronischer Hintergrund:
  Zum elektronischen Hintergrund zählt das Rauschsignal der Aperturmaske (Kodierungs-Rauschen), sowie das Thermische- und das Shot-Rauschen der Signal-Verstärker.

- Detektor-Hintergrund:
  Der Detektor-Hintergrund wird durch hochenergetische Teilchen, die kosmischer Herkunft sind oder aus dem Sonnenwind stammen, hervorgerufen. Die Teilchen wechselwirken mit dem Satelliten- oder Detektor-Gehäuse und erzeugen dort Röntgen- und Gamma-Strahlung. Zum Teil wechselwirken die Partikel auch direkt im Detektor.

[6] Die Sonne emittiert neben Röntgenstrahlung den sogenannten Sonnenwind, einen Strom hochenergetischer geladener Teilchen, vor allem Protonen und Elektronen.

- kosmischer Röntgenhintergrund:
  Der kosmisch Röntgenhintergrund ist diffus, und kann vom CMI nicht in einzelne Quellen aufgelöst werden. Nach bisheriger Kenntnis sind die Quellen größtenteils sogenannte AGNs[7], deren Röntgen-Emissionen sich zu einem ununterscheidbaren Signal überlagern.

Der EPP besitzt die Fähigkeit, einen Großteil der Störquellen zu unterdrücken. So werden z.B. Röntgen-Ereignisse innerhalb des Detektors nur dann als gültige *Event-Packets* übertragen, wenn die Signale von direkt einander benachbarten Elektroden stammen. Ein Großteil des Detektor-Hintergrunds kann durch diese Maßnahme eliminiert werden. Für den elektronischen Hintergrund hält der EPP zwei programmierbare Energie-Filter bereit, mit deren Hilfe vor allem der niederenergetische Rauschanteil der Elektronik wirksam unterdrückt werden kann.

Leider sind der Telemetrie bei Satellitenmissionen enge Grenzen gesetzt. Die *Downlink*-Rate ist auf ca. 100 kBit pro Sekunde begrenzt. Nimmt der CMI als Zweit- oder Dritt-Experiment an einer Mission teil, wie es bei GLAST geplant war, so stehen nur noch *Downlink*-Raten von ca. 10 kBit pro Sekunde zur Verfügung. Um bei solch geringen Transfer-Raten noch einen vernünftigen Meßbetrieb zu erreichen, ist eine starke Reduktion der Daten angezeigt. Die Datenverarbeitung soll deshalb möglichst weit an Bord durchgeführt werden. Dazu sind zunächst alle Störereignisse herauszufiltern, die verbleibenden Messdaten sind wenn möglich vorzuverarbeiten oder vollkommen auszuwerten, bevor sie zur Bodenstation übertragen werden. Für die verbleibenden Daten ist die Telemetrierate dann gerade ausreichend.

Da CMI noch für keine Mission vorgeschlagen ist, kann auch über die verfügbaren Telemetrie-Raten keine Angabe gemacht werden. Es ist jedoch für die Verwendbarkeit bestimmt kein Schaden, wenn diese so dimensioniert ist, daß sie auch sparsamsten Ansprüchen noch gerecht wird. Aus diesem Grund besitzt der CMI in seinem *Event-Pre-Processor* ein leistungsfähiges Instrument zur Datenreduktion.

Die Datenreduktion innerhalb des EPP findet in mehreren Stufen statt. Ziel der Verarbeitung ist, die vom Detektor empfangenen Roh-Daten soweit wie möglich von Störungen zu befreien und für die Weiterverwendung in der DPU vorzuverarbeiten. Die Detektordaten werden dazu zunächst auf Koinzidenz untersucht. Nur Ereignisse die in Ort und Zeit entsprechend koinzidieren, sind potentielle Daten, alle anderen Ereignisse werden markiert und später verworfen. Die Ereignispakete durchlaufen anschließend den EPP und werden bereits in der ersten Stufe bezüglich der Paketlänge stark reduziert. Dies ist möglich, indem ein Großteil der Rohinformationen zu wissenschaftlich aussagekräftigen Daten vorverarbeitet wird. Nach der Vorverarbeitung werden die Daten gefiltert. Das Filter des EPP eliminiert dabei alle Pakete die zuvor als Störereignisse oder als defekte Pakete markiert worden sind und die mit hoher Wahrscheinlichkeit Störhintergrund darstellen. Die verbleibenden Pakete stellen nur noch einen Bruchteil der ursprünglichen Daten dar, und repräsentieren Gamma-Ereignisse, die nicht vom Störhintergrund stammen, sondern ihren Ursprung in tatsächlich lokalisierbaren Gamma-Quellen haben.

Für die Verwendung als *Burst*-Monitor ist der Datentransfer noch weiter reduzierbar. Zur Erfüllung dieser Aufgabe werden nämlich letztenendes nicht die *Event-Packets*, sondern nur deren Zählraten benötigt. Diese wiederum werden an Bord mit dem aktuellen Mittelwert verglichen und lösen nur bei starken Abweichungen einen *Burst*-Alarm aus. Für einen reinen Monitor-Betrieb ist der Datentransfer also verschwindend gering. Erst bei Ansprechen des *Burst-Triggers* steigt der Transferbedarf. Selbst dann aber, können relevanten Daten im eingebauten Ring-Speicher so lange gehalten werden, bis die Verbindung zur Bodenstation eine Übertragung erlaubt.

[7] AGN: Active Galaxy Nuclei

4.2 Aufbau und Funktion des EPP

Der Entwurf des EPP geschieht vollständig am Rechner in der Hardware Beschreibungssprache VHDL. Mit dieser Entscheidung geht das IAAT neue Wege in Sachen Hardware-Design. Frühere Projekte wurden zumeist in *Schematics* realisiert und stießen immer wieder an Grenzen der eingesetzten Software. Aus diesem Grund wird der EPP von Grund auf in VHDL geschrieben und getestet. In einer zweiten Stufe soll das Design in einen FPGA-Chip programmiert und die Hardware Realisierbarkeit getestet werden. Nach der endgültigen Fertigstellung des Entwurfs kann der EPP dann als weltraumqualifizierter ASIC-Baustein zur Herstellung gebracht werden.

4.2.1 Die Design-Hierarchie

Das Design des EPP ist modular angelegt. Die Design-Hierarchie unterscheidet dabei drei Ebenen: Module, Units und EPP-Top. Die Module sind funktionale VHDL-Designs, die Units und das EPP-Top strukturale. In dieser Weise nutzt das EPP-Design die Möglichkeiten zur Strukturierung die VHDL bietet.

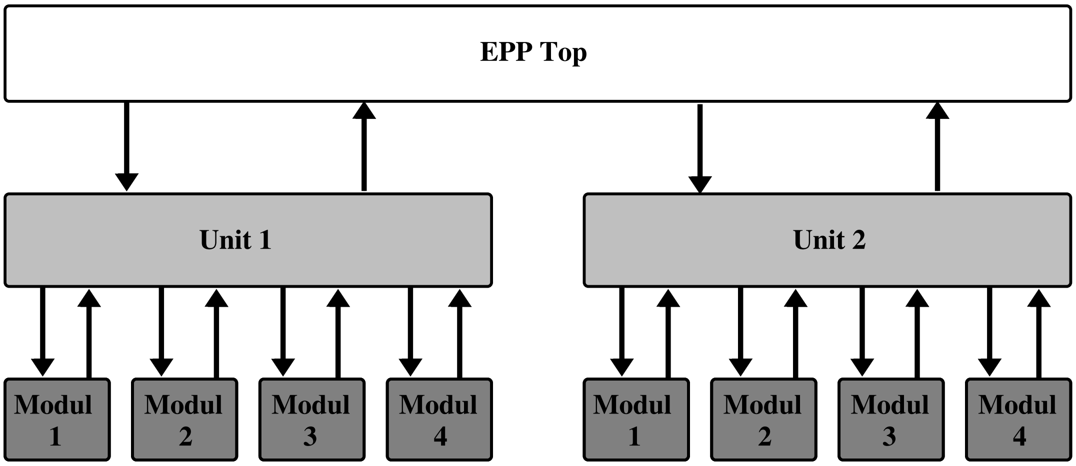


Abbildung 4.10: *Die drei Ebenen des EPP-Designs. In der untersten Ebene befinden sich die Module. Module sind funktionale VHDL-Designs und realisieren die operativen Aufgaben des EPP. In der zweiten Ebene stehen die Units. Jede Unit faßt Module mit ähnlichen Aufgaben zu einer Einheit zusammen. Units sind strukturale Designs und liefern die Verdrahtung der Modul-Ebene. Die oberste Ebene, EPP-Top ist ebenfalls struktural und verdrahtet die Unit-Ebene. Sie stellt außerdem die Schnittstelle zu peripheren Baugruppen und Interface-Bausteinen zur Verfügung.*

Die unterste Ebene des EPP-Designs sind die Module. Die Module sind die operativen Elemente des Prozessors. Jedes Modul erfüllt dabei eine ganz spezifische Aufgabe bezüglich Verarbeitung und Manipulation der Detektordaten. Im Sinne von VHDL sind die Module funktional modellierte Designs. Jedes Modul für sich ist ein vollständiges VHDL-Design mit ENTITY und ARCHITECTURE. Die Module konnten deshalb einzeln entwickelt und getestet werden, bevor sie zu größeren Designeinheiten zusammengefügt wurden. Die zweite Ebene sind die Units. Units fassen Module mit verwandten Aufgaben, wie z.B. Fehlerkorrektur oder Ereigniszählung, zu größeren Einheiten zusammen. Sie verknüpfen dabei die Ein- und Ausgänge der Module, beinhalten selbst also nur strukturale Elemente von VHDL. Auch im Fall der Units handelt es sich um vollständige VHDL-Designs die einzeln getestet und dann zum EPP-Top -Design zusammengefügt wurden. Das EPP-Top -Design ist die oberste Ebene der EPP-Hierarchie, es verbindet die Units untereinander und ist die Schnittstellen zu den Peripherie- und *Interface*-Bausteinen. Genauso wie die darunterliegenden Units, besitzt auch das EPP-Top ausschließlich strukturale Elemente. Abbildung 4.10 zeigt eine Darstellung der Hierarchie des EPP. Alle Baugruppen derselben Ebene kommunizieren über die nächst höhere Ebene miteinander. Signale zweier Module werden niemals direkt ausgetauscht, sondern immer über die Unit-Ebene geführt. Die Vorgehensweise isoliert die Module voneinander und verbessert so die Modifizierbarkeit. Etwaige Designänderungen betreffen deshalb zumeist nur das geänderte Modul selbst, und ggf. das aufrufende Unit-Design, aber keine anderen Design-Einheiten.

Das EPP-Top -Design erstreckt sich zum derzeitigen Stand der Arbeit nur auf den *Core*-Bereich des Prozessors (siehe Abbildung 4.11). Der *Core*-Bereich beinhaltet alle Units des EPP die Aufgaben der Datenverarbeitung übernehmen. Nach Fertigstellung und Test des *Core*-Bereichs wird das EPP-Top um die erforderlichen Peripherie-Designs ergänzt, und auf diese ausgedehnt. Die Gesamtheit aller VHDL-Designs bildet am Schluss das Top-Design des EPP-Projekts.

4.2.2 Funktions Beschreibung des EPP

Das Top-Design verbindet die hierarchisch tieferen Designeinheiten, die Units miteinander. Als Schnittstellen existieren zwei *High-Bit-Rate* Datenschnittstellen (HBR-*Interfaces*) zur Datenübertragung vom Detektor zum EPP und vom EPP zur *Data-Processing-Unit* (DPU). Des weiteren existiert ein *Low-Bit-Rate-Telecommand-Interface* (LBR-*Interface*) mit dessen Hilfe der EPP programmiert und gesteuert wird. Die Units des *Core*-Bereichs dienen der Datenverarbeitung, die Units des Peripheriebereiches der Kommunikation, Datenspeicherung und der Steuerung des EPP. Der schematische Aufbau des EPP-Designs ist in Abbildung 4.11 gezeigt. Das EPP-Top -Design beinhaltet insgesamt folgende Units:

Abkürzung	Bedeutung	Bereich
HBRI-UNIT Det/EPP	High-Bit-Rate-Interface Unit - Detektor/EPP	Peripherie
EPA-UNIT	Event-Packet Assembly Unit	Core
EC-UNIT	Error Correction Unit	Core
FD-UNIT	Filter & Decision Unit	Core
BD-UNIT	Burst Detection Unit	Core
RB-UNIT	Ring Buffer Unit	Peripherie
HBRI-UNIT EPP/DPU	High-Bit-Rate-Interface EPP/DPU - Unit	Peripherie
LBRI-UNIT	Low-Bit-Rate-Interface & Telecommand Unit	Peripherie

Tabelle 4.2: *Die Units des EPP*

Das Konzept der *Event-Packets*:

Grundlage für ein Verständnis der Arbeitsweise des EPP ist das Konzept der *Event-Packets*. Ein *Event-Packet* enthält alle Informationen derjenigen Elektroden des Detektors, die im Zeitraum einer Wechselwirkung ein Signal ableiten. Dies ist die Definition des *Event-Packets*[8]. Unabhängig davon ist das Datenformat und die Wortlänge. Auf seinem Weg durch den EPP erfährt das *Event-Packet* nämlich zahlreiche Modifikationen. Informationen werden korrigiert und umgeformt, zusammengefasst und mit Markierungen versehen. Dennoch repräsentiert das *Event-Packet* zu jedem Zeitpunkt den Inhalt seiner Definition. Ein *Event-Packet* enthält Daten über Energie, Position und Zeitpunkt eines Ereignisses, hinzu kommen *Flags*, welche die Gültigkeit kennzeichnen.

[8] Genau genommen gilt die Definition nur für ein gültiges *Event-Packet*. Der Begriff *Event-Packet* wird später für die Dateneinheit verwendet, auch wenn diese z.B. leer ist.

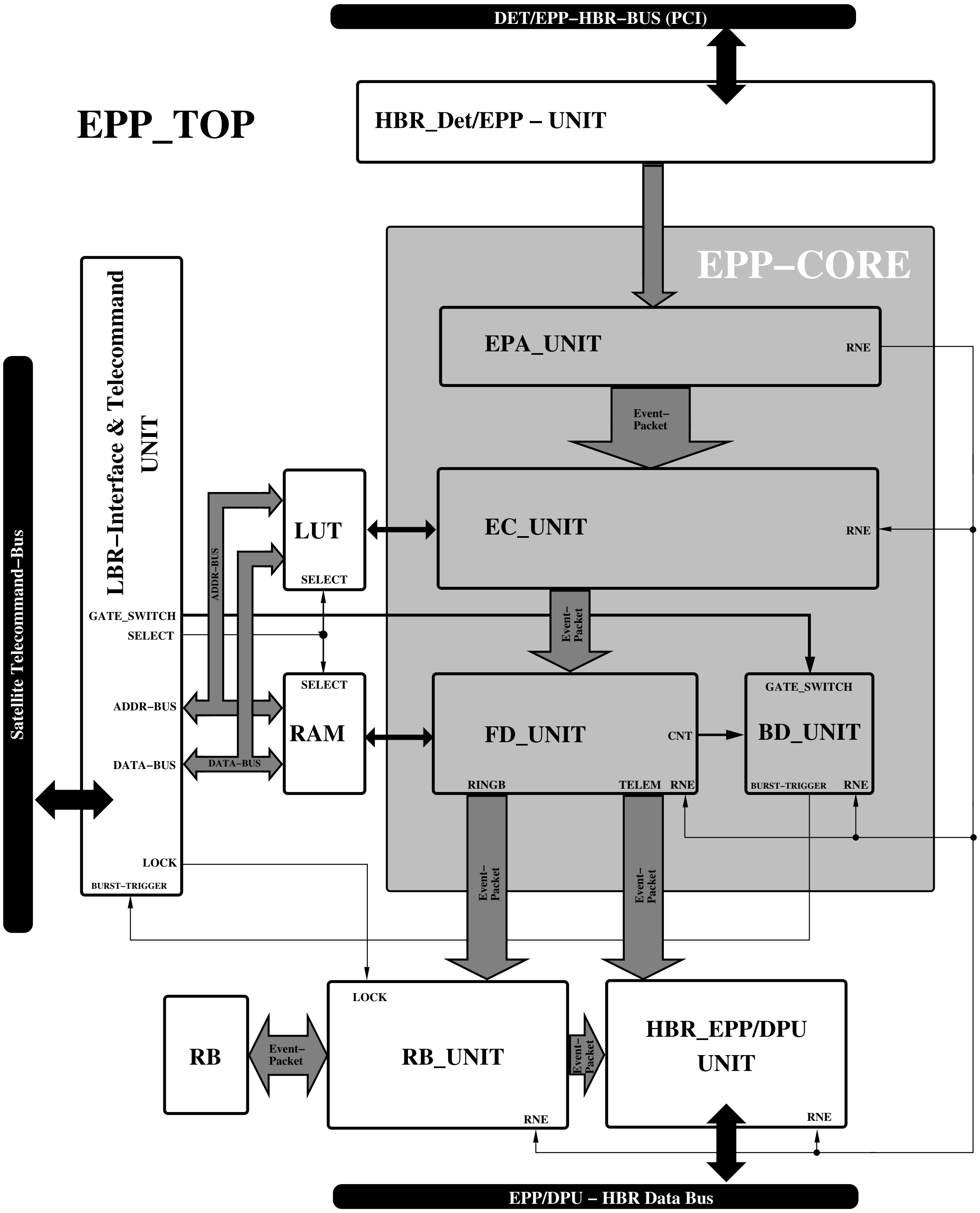


Abbildung 4.11: *Das Top-Design des EPP. Der Core-Bereich ist grau unterlegt. Die Daten werden in Form von Event-Packets von einer Unit zur nächsten gegeben. Die Übergabe der Pakete erfolgt mit Hilfe des* RNE *Pipeline-Signals. Länge und Aufbau der Event-Packets wird auf dem Weg durch die Units den Erfordernissen entsprechend ständig neu angepasst.*

Das *Event-Packet* wird innerhalb der EPA-Unit aus den Detektordaten erzeugt und durchläuft die einzelnen Units des EPP. Die EC-Unit ist das Herzstück des EPP, sie führt alle erforderlichen Korrekturen und eine Reihe von grundlegenden Berechnungen durch. Innerhalb der EC-Unit wird das *Event-Packet* zunächst in zwei Hälften geteilt. Der *Timestamp* wird isoliert und durchläuft getrennt, aber gleichzeitig das Modul zu Zeit-Korrektur. Der Rumpf des *Event-Packets*, mit den Orts- und Amplituden-Daten gelangt in die Amplituden-Korrektur, und wird danach zur Bearbeitung weiter zerlegt. Am Ende der EC-Unit werden alle Daten zu einem neuen *Event-Packet* zusammen geführt. Die Auswirkung der EC-Unit auf das *Event-Packet* ist eine Reduzierung der Paket-Breite. Beträgt dieses zu Beginn der EC-Unit noch 168 Bit, so schrumpft sie gegen Ende auf 45 Bit zusammen.

Die FD-Unit beeinflußt die Länge der *Event-Packets* nicht, sie reduziert die Anzahl. Die FD-Unit ist als Filter aufgebaut. Sie unterdrückt alle ungültigen *Event-Packets* und solche, deren Energie-Eintrag nicht den Vorgaben entspricht. Die FD-Unit filtert eine Vielzahl von Paketen heraus, und eliminiert damit einen Großteil des Störhintergrundes. Das Energiefenster des *Noise*-Filters ist von der Bodenstation programmierbar, ebenso ein zweites Energiefilter für die *Burst*-Detektion. Die gefilterten *Event-Packets* werden an das HBR-*Interface* EPP/DPU und in die *Ring-Buffer*-Einheit geleitet. Außerdem erzeugt jedes *Event-Packet* das den *Burst*-Detektor passiert einen Zählimpuls am Eingang des *Event*-Zählers der BD-Unit.

Die RB-Unit enthält einen *Ring-Buffer*. Dieser zeichnet die *Event-Packets* der Reihe nach auf, seine Funktion ist mit einem Endlosband vergleichbar. Ist der *Buffer* voll, werden die zuerst gespeicherten Daten wieder überschrieben. Soll ein Ereignis näher untersucht werden, kann die Aufzeichnung gestoppt und der Inhalt ausgelesen werden. Der *Ring-Buffer* kann von der Bodenstation gesperrt und gelesen werden, er erlaubt die Analyse der GRB-Lichtkurven zusammen mit deren Vorgeschichte.

Die BD-Unit führt Statistik über die Zählimpulse des *Burst*-Detektors. Für jedes Paket, das den Energie-Kriterien entspricht, erzeugt der *Burst*-Detektor einen Zählimpuls am Eingang der BD-Unit. Diese werden in festen Zeitintervallen gezählt und die Zählraten zu einem Mittelwert verarbeitet. Die BD-Unit vergleicht die aktuelle Zählrate mit dem Mittelwert und löst bei entsprechend großen Abweichungen den *Burst-Trigger* aus. Der *Burst-Trigger* benachrichtigt die Bodenstation, daß ein *Gamma-Ray Burst* entdeckt wurde, und alarmiert ggf. das GCN[9]. Die Zeitfenster der Zählratenermittlung sind von der Bodenstation programmierbar, ebenso die Ansprechschwelle des *Burst-Triggers*.

Die Verarbeitungs-*Pipeline*:

Die Signalverarbeitung des EPP ist dynamisch angelegt und wird durch einen Takt-Impuls (oder *Clock*-Impuls) in Gang gehalten. Der *Clock*-Impuls wird durch einen *Clock*-Generator außerhalb des EPP erzeugt und als Eingangsignal angelegt. *Clock*-Frequenzen sind üblicherweise im Mega-Hertz-Bereich um einen ausreichenden Datendurchsatz zu gewährleisten. Die Detektorelektronik wird mit 24 MHz betrieben, die Verwendung dieser Frequenz ist auch für den EPP naheliegend. Bei Bedarf kann der EPP auch mit einem Teil oder Vielfachen der angelegten Frequenz betrieben werden. Die genaue Festlegung der *Clock*-Frequenz ist zu gegebenem Zeitpunkt an die Erfordernisse anzupassen, dabei müssen zwei Kriterien beachtet werden:

- Die *Clock*-Frequenz muß groß genug sein um die Verarbeitung der Daten in Echtzeit sicherzustellen.
- Die *Clock*-Frequenz muß klein genug sein, sodaß für alle Komponenten des EPP ein stabiler Betrieb möglich ist.

Weitere Kriterien können Bauteil-Erwärmung und Stromverbrauch sein, die mit ansteigender Frequenz zunehmen werden. Im Rahmen dieser Bedingungen wird die endgültige Frequenz für das *Clock*-Signal festzulegen sein.

[9]GCN: GRB Coordinated Network

Die Signalverarbeitung innerhalb des EPP arbeitet synchron. Hierzu wird aus dem *Clock*-Impuls, das sogenannten RNE-Signal abgeleitet. RNE steht für *Read-Next-Event* und definiert den *Pipeline*-Takt des EPP. Das RNE-Signal wird von der EPA-Unit erzeugt und an die nachfolgenden Units verteilt (siehe Abb.4.11). Mit jedem RNE-Signal wird ein neues *Event-Packet* von der EPA-Unit an die EC-Unit übertragen, die Übergabe ist also RNE-synchron. Weitere Daten-Paket-Transfers die RNE-synchron arbeiten erfolgen zwischen EC- und FD-Unit, FD- und RB-Unit, sowie zwischen FD- und HBRI-Unit.

Manche RNE getriggerten Units benutzen das RNE Signal auch zur Erzeugung von SUB-RNE Signalen, die sie zur internen Ablaufsteuerung verwenden. Die SUB-RNES werden innerhalb der jeweiligen Unit mit Hilfe eines Zählers erzeugt. Die Länge kann den Erfordernissen innerhalb der Units angepasst werden.

Die RNE getriggerte Datenweitergabe zwischen den Units entspricht einem *Pipeline* Prinzip. Es befinden sich deshalb immer mehrere *Event-Packets* gleichzeitig in Bearbeitung. Während die FD-Unit ein Daten-Paket in das HBR-Telemetrie-*Interface* schickt, gelangt das nachfolgende Paket von der EC- in die FD-Unit und ein weiteres von der EPA- in die EC-Unit. Vergleiche hierzu Abbildung 4.12.

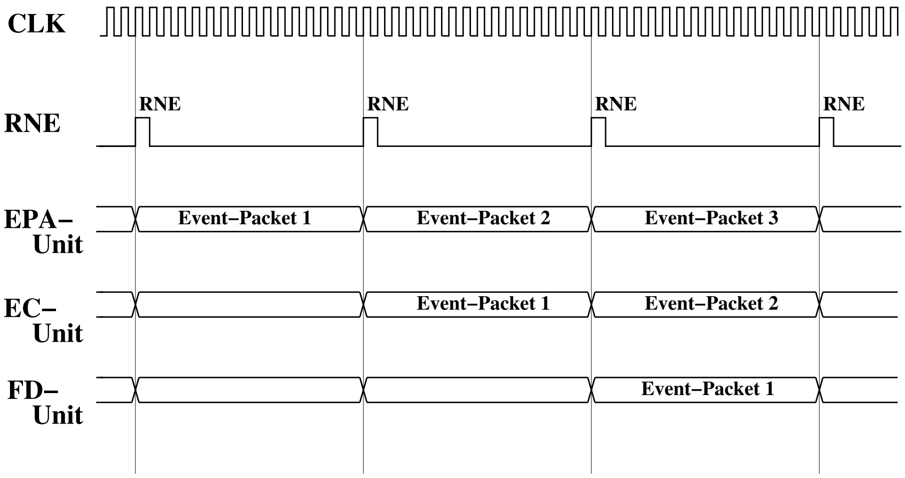


Abbildung 4.12: *Das RNE-Signal, Pipeline-Takt für den EPP. Das RNE-Signal steuert die Übergabe der Event-Packets zwischen den einzelnen Units des EPP. Mit jedem RNE übergibt die EPA-Unit ein neues Event-Packet an die EC-Unit. Gleichzeitig gelangt ein Event-Packet von der EC- in die FD-Unit und von der FD- in die RB- bzw. BD-Unit und in die Telemetrie.*

Die *Pipeline* des EPP kann nicht angehalten werden. Sollten einmal keine Detektordaten zur Verfügung stehen, müssen leere *Event-Packets* erzeugt werden. Leere *Event-Packets* sind im Sinne der Auswertung ungültige Daten und werden in der FD-Unit vernichtet. Sie gelangen weder in die Telemetrie noch in den *Ring-Buffer*.

4.3 Die Units im Detail

In diesem Unterkapitel soll auf Funktion und inneren Aufbau des EPP näher eingegangen werden. Dazu werden die Units im einzelnen vorgestellt und die Funktionsweise bis auf die Modul-Ebene erläutert. Eine genauere technische Beschreibung anhand der VHDL-*Listings*, oder mit Hilfe von Signal-*Schematics* würde jedoch den Rahmen dieser Ausarbeitung sprengen. Für ein Studium der VHDL-*Listings* sei auf den Anhang verwiesen. In Anhang A befinden sich auch detaillierte Signal-*Schematics* der drei *Core*-Units EC, FD und BD, die im Rahmen dieser Arbeit entstanden sind. Die CDROM in Anhang B enthält die zugehörigen VHDL-Listings.

4.3.1 HBR-*Interface-Unit* Detektor/EPP

Das HBR-*Interface* ist die Schnittstelle des EPP zur Detektorelektronik und der Dateneingang für den EPP. Seine Aufgabe ist die Kommunikation durch Übertragungsprotokolle mit dem Sendebaustein der Detektorelektronik. Die Detektorelektronik wird von CASS @ USCD entwickelt und hat die Aufgabe, die Rohdaten des Detektors zu 40 Bit breiten *Hits* zu verarbeiten.

Jeder dieser *Hits* enthält die Informationen eines einzelnen Elektrodenstreifens des Detektors. Diese sind Signalamplitude, Streifen Nummer (ID), *Hit* Typ und ein *Hit*-Zeitstempel (siehe auch Abb. 4.13). Der *Hit*-Zeitstempel dient später der Unterdrückung von Mehrfachereignissen und der Generierung des *Event-Packet*-Zeitstempels innerhalb der EPA-Unit. Die *Hits* werden über einen 16 Bit breiten, PCI kompatiblen, Datenbus an das HBR-*Interface* des EPP übertragen. Die vorläufigen Datenformate sind in Abbildung 4.13 dargestellt.

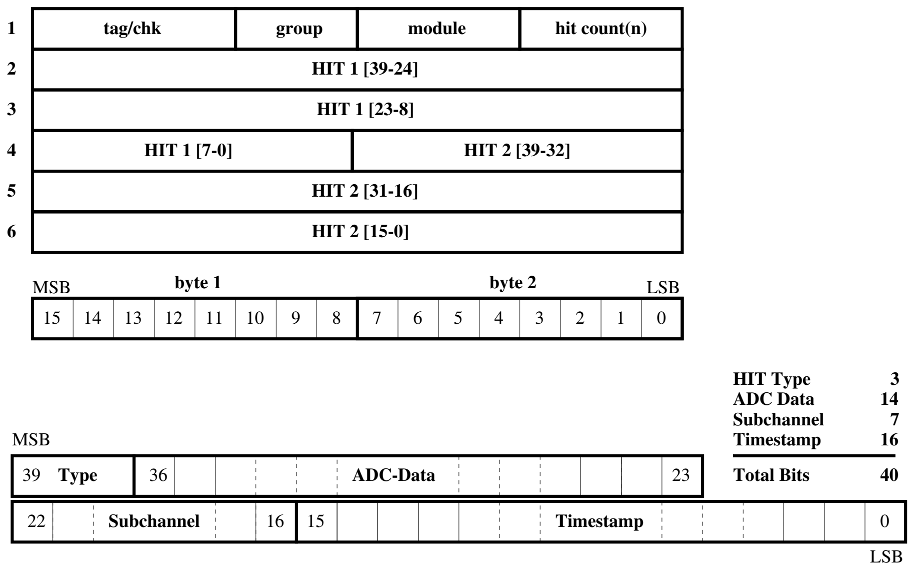


Abbildung 4.13: *Das Daten-Format der Detektorelektronik. Die Detektorelektronik erzeugt aus den Detektordaten Hits, die über einen 16 Bit breiten PCI Bus an den EPP übertragen werden. Jeder Hit ist 40 Bit breit und enthält den vollständigen Datensatz einer einzelnen Elektrode. Die Anzahl der Hits, die an einem Wechselwirkungs-Ereignis beteiligt sind kann zwischen 2 und 4 schwanken.*

Zu Beginn einer jeden *Hit*-Sequenz sendet die Detektorelektronik zwei Byte Status Informationen an das HBR-*Interface* des EPP. Diese Status-Bytes enthalten *Handshake*-Informationen für die Datenübertragung sowie Prüfsummen, und Informationen über die Anzahl der zu übertragenden *Hits*. Der genaue Inhalt dieser Status-Bytes ist bisher nicht spezifiziert. Nach der Initialisierung der Übertragung folgen die eigentlichen Detektordaten in Form mehrerer *Hits*. Die Anzahl der *Hits* pro *Event* kann zwischen zwei und vier schwanken. Die Erzeugung eines *Event-Packets* aus diesen Daten ist Aufgabe der EPA-Unit.

4.3.2 EPA-Unit (*Event-Packet Assembly Unit*)

Erzeugung der *Event-Packets*:

Die Aufgaben der EPA-Unit ist die Erzeugung der *Event-Packets*. Ein *Event-Packet* ist ein Datenwort variabler Breite. Es enthält die Informationen der Detektor-Elektroden, auf denen als Folge der Wechselwirkung ein Signal abgeleitet wurde. Jedes *Event-Packet* trägt die Informationen genau eines Detektor-Ereignisses durch den EPP. Es verändert dazu je nach Stand der Berechnungen seine Breite und sein Datenformat. Am Eingang des EPP-*Core* ist das *Event-Packet* 168 Bit breit, am Ausgang als Folge der Verarbeitung und Reduktion auf 41 Bit komprimiert.

Die EPA-Unit erzeugt das *Event-Packet* aus den Detektor-Daten. Diese werden in Form von *Hits* gesendet. Jeder *Hit* enthält die Information genau einer Detektor-Elektrode.

Wieviele *Hits* zu einem *Event-Packet* gehören hängt von Winkel und Richtung der Ladungswolke ab, die in Folge der Wechselwirkung im Detektor entsteht. Aufgrund bisheriger Messungen mit Detektor-Prototypen kann von maximal 2 *Hits* pro Anode und Kathode ausgegangen werden. Ein *Event-Packet* muß deshalb für die Übertragung von bis zu 4 *Hits* Platz bieten. Um für zukünftige Entwicklungen gewappnet zu sein, wurde das *Event-Packet* des EPP sogar für den Transport von 6 *Hits* ausgelegt (3 Anoden-*Hits* und 3 Kathoden-*Hits*). Das *Event-Packet* ist in der EPA-Unit dazu wie in Abbildung 4.14 aufgebaut. Neben der *Timestamp*-Information enthält es sechs Zeilen für die einzelnen *Hits*. Die *Hits* wurden zuvor um ihre Zeitstempel reduziert. Innerhalb des *Event-Packets* enthalten sie nur noch die Amplituden- und Ortsinformationen der zugehörigen Elektroden und ein VD-Bit, das die Gültigkeit kennzeichnet. Die Breite des EPA-*Event-Packets* ist nicht dynamisch, d.h. sie beträgt unabhängig von der Anzahl der erzeugten Detektor-*Hits* für jedes *Event* stets 168 Bit. Die nicht benötigten *Hits* (Zeile 3 und Zeile 6) des *Event-Packets* werden, solange nicht erforderlich, mit Null-*Hits* aufgefüllt. Dies gilt im übrigen auch für andere Zeilen, wenn vom Detektor weniger als 4 *Hits* erzeugt werden. Null-*Hits* enthalten mit Ausnahme des VD-Bits lauter Nullen, sie beeinflussen die Berechnung innerhalb des EPP nicht.

Wichtig für die Verarbeitung ist die Klassifizierung in gültige und ungültige *Event-Packets*. Die Information über die Gültigkeit wird zunächst in jedem *Hit* einzeln gespeichert. Dazu besitzt jeder *Hit* ein sogenanntes VD-Bit[10]. Ist das VD-Bit Null, so ist der *Hit* ungültig, bei Eins gültig. Die *Hit*-weise Erfassung der Gültigkeit erlaubt insbesondere im Hinblick auf die ebenfalls *Hit*-weise Verarbeitung der *Event-Packets* in der EC-Unit, eine detaillierte Diagnose von Fehler-Ursachen. Das *Event-Packet* als ganzes gilt als ungültig wenn bereits ein einziges seiner VD-Bits eine Null trägt.

Für die Erzeugung der *Event-Packets* untersucht die EPA-Unit die *Hits* auf Orts- und Zeit-Koinzidenzen. Für die Ortsuntersuchung gilt das Kriterium, daß nur *Hits* die zu direkt benachbarten Elektrodenstreifen gehören, gültige Ereignisse repräsentieren. Die Zeituntersuchung vergleicht die Zeitstempel der *Hits* auf Übereinstimmung, bevor diese in das *Event-Packet* eingebaut werden. Liegen die Zeiten zu weit auseinander stammen die Daten aus verschiedenen Ereignissen. Zu einem Doppelereignis kommt es z.B., wenn zwei Gamma-Quanten zufällig gleichzeitig in der Detektorplatte wechselwirken, oder wenn ein einzelnes Gamma-Photon über Compton-Streuung an verschiedenen Stellen der Detektorplatte Ladungsträger erzeugt. In beiden Fällen kann keine verlässliche Rekonstruktion der Herkunft des Photons gewährleistet werden. Ist ein *Hit* bei einer der Koinzidenz-Untersuchungen ungültig, so wird sein VD-Bit auf Null gesetzt. Damit ist sowohl der *Hit*, als auch das *Event-Packet* ungültig.

[10]VD-Bit: Valid Data Bit; Gültigkeits-Bit

Der derzeitige Detektor erlaubt für gültige *Event-Packet* die Beteiligung von minimal 2 und maximal 4 gültigen *Hits*. Empfängt die EPA-Unit mehr als 4 *Hits* auf einmal, und können diese nicht eindeutig zwei verschiedenen *Events* zugeordnet werden, so erzeugt die EPA-Unit daraus ein ungültiges *Event-Packet*. In diesem Fall werden die VD-Bits der betroffenen *Hits* auf Null gesetzt. Das Gleiche geschieht, wenn weniger als 2 *Hits* einem *Event* zugeordnet werden können.

Da der EPP von der EPA-Unit an synchron arbeitet, muß diese mit jedem RNE-Signal ein *Event-Packet* erzeugen. Der EPP verarbeitet deshalb sowohl gültige als auch ungültige *Event-Packets*. Stehen keine Detektordaten zur Verfügung, werden leere *Event-Packets* mit Nulleinträgen und ungültigen VD-Bits erzeugt. Eine Entscheidung über den Verbleib der *Event-Packets* wird erst am Ende der Verarbeitungs-*Pipeline*, in der FD-Unit, getroffen. Dort werden ungültige Pakete eliminiert. Diese Vorgehensweise ist sinnvoll, da VD-Bits nicht nur von der EPA-Unit geschrieben werden können, sondern auch von einer der anderen Units des EPP.

MSB

	167						Time Stamp 16 Bit									152	151			Spare 8 Bit				144
Anodes	143	Hit 1			Amplitude 14 Bit (rawdata)									130	129	128			Location 9 Bit				121	VD DET
Anodes	119	Hit 2												106	105	104			101				97	96
Anodes	95	Hit 3												82	81	80			77				73	72
Cathodes	71	Hit 4												58	57		55	54	53				49	48
Cathodes	47	Hit 5												34	33		31	30	29				25	24
Cathodes	23	Hit 6												10	9		6	5	4				1	0

LSB

Abbildung 4.14: *Das EPA-Event-Packet wie es die EPA-Unit in Richtung EC-Unit verlässt. Neben dem 16 Bit breiten Time-Stamp sind die sechs Hits mit den Detektordaten zu erkennen. Jeder Hit besteht aus 14 Bit Amplitudeninformation, einem 9 Bit breiten Ortsdatensegment und dem Valid-Data-Bit, das die Gültigkeit des jeweiligen Hit anzeigt.*

Zu beachten ist, daß die Reihenfolge der *Hits* innerhalb des *Event-Packets* nicht beliebig ist (vgl. Abb. 4.14)! Die ersten drei *Hits* (Bit 143 bis 72) sind für die Anode reserviert, die letzten drei *Hits* (Bit 71 bis 0) für die Kathode. Es ist Aufgabe der EPA-Unit die *Hits* an die richtige Stelle des *Event-Packets* zu platzieren, anderenfalls eine korrekte Verarbeitung in den nachfolgenden Units nicht möglich ist. Der Aufbau eines *EPA-Event-Packets* ist in Abbildung 4.14 gezeigt.

Das Ortsdatensegment:

Eine andere Aufgabe der EPA-Unit ist die Erzeugung eines EPP-kompatiblen, 9 Bit breiten Ortsdatensegments aus dem *Type* und *Subchannel* der Detektor-*Hits*. Das Ortsdatensegment transportiert die Detektorkoordinate der zugehörigen Streifenelektrode. Weil der Detektor in 16 Einzel-Elemente segmentiert ist, beinhaltet das Ortsdatensegment neben der *Hit*-Adresse auch die Segment-Adresse. Die Segment-Adresse entspricht den ersten 4 Bit des Ortsdatensegments, die *Hit*-Adresse den letzten 5 Bits. Der Aufbau des Ortsdatensegments ist in Abbildung 4.15 gezeigt.

Das Ortsdatensegment dient gleichzeitig der Positionsbestimmung der Röntgen-Quelle und der Amplitudenkorrektur als Adresse für den *Look-Up-Table*.

Erzeugung des RNE-Signals:

Neben den *Event-Packets* erzeugt die EPA-Unit das RNE-Signal. Das RNE-Signal definiert den *Pipeline*-Takt des EPP und wird von der EPA-Unit an alle nachfolgenden Units verteilt (Vgl. Abb. 4.11).

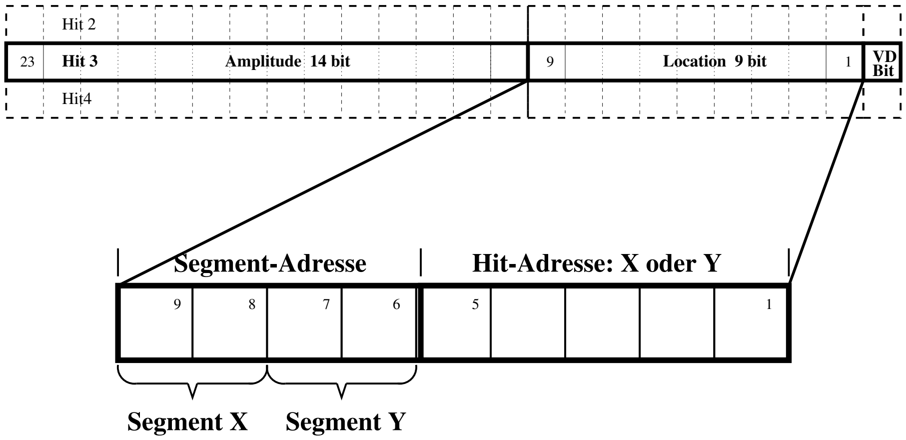


Abbildung 4.15: *Das Ortsdatensegment. Jeder Hit-Eintrag des Event-Packets enthält ein eigenes Ortsdatensegment. Das Ortsdatensegment ist 9 Bit breit und transportiert die Adresse der Elektrode, die das betreffende Signal (Amplitude) abgeleitet hat. Zusätzlich liefert das Ortsdatensegment die 10 Bit breite LUT-Adresse. Diese setzt sich aus dem 9 Bit breiten Ortsdatensegment und dem Hit-Typ (zehntes Bit) zusammen. Das zehnte Bit ist Null für Hits der Anode (Zeile 1 bis 3 des Event-Packets), und Eins für Hits der Kathode (Zeile 4 bis 6).*

Mit Hilfe des RNE-Signals werden die *Event-Packets* von einer Unit zur nächsten gegeben. Auf diese Weise befinden sich immer drei *Event-Packets* gleichzeitig innerhalb des EPP. Einige Units benutzen ihrerseits das RNE-Signal zur internen Ablaufsteuerung und erzeugen daraus sogenannte SUB-RNE-Signale. Die EPA-Unit besitzt zur Erzeugung des RNE einen Zähler der exakt alle 100 *Clock*-Impulse das Signal mit einer Länge von einem *Clock* erzeugt. Sollten zukünftige Designänderungen einen anderen *Pipeline*-Takt erfordern, so sind der RNE-Zähler der EPA-Unit und die SUB-RNE-Zähler der Units entsprechend zu modifizieren. Soll der *Pipeline*-Takt in kürzeren Abständen erfolgen, ist darauf zu achten, daß SUB-RNE gesteuerten Units genügend Zeit zur Datenverarbeitung bleibt. Bei welcher *Pipeline*-Taktlänge die untere Grenze für eine zuverlässige Verarbeitung liegt, ist bisher nicht untersucht worden, lediglich bei einer Verlängerung des *Pipeline*-Taktes sind keinerlei Beeinträchtigungen der Funktionalität zu erwarten.

4.3.3 EC-Unit (*Error Correction Unit*)

Die EC-Unit ist das Herzstück des EPP. Sie beinhaltet alle Komponenten zur Fehlerkorrektur und stellt elementare Berechnungen mit den Eingangsdaten an. Dazu stehen eine ganze Reihe von Modulen zur Verfügung. Ein Überblick über den Aufbau der EC-Unit ist in Abbildung 4.16 und in Tabelle 4.3 gegeben. Das Eingangsdatenformat des *Event-Packets* ist in Abbildung 4.14 zu sehen.

Korrektur des Zeit-Stempels:

Der exakte Aufbau des Zeitstempels (*Timestamp*) ist bisher nicht festgelegt, auch das Korrekturverfahren ist noch nicht geklärt. Der 16 Bit breite *Timestamp* passiert das TC-Modul deshalb unverändert und wird dem Ausgang-*Event-Packet* (Abb. 4.22) der EC-Unit wieder ohne Korrekturen angehängt. Zu gegebener Zeit wird das bisher noch leere Modul entsprechend modifiziert, und zu einer Korrekturstufe des *Timestamp* ausgebaut.

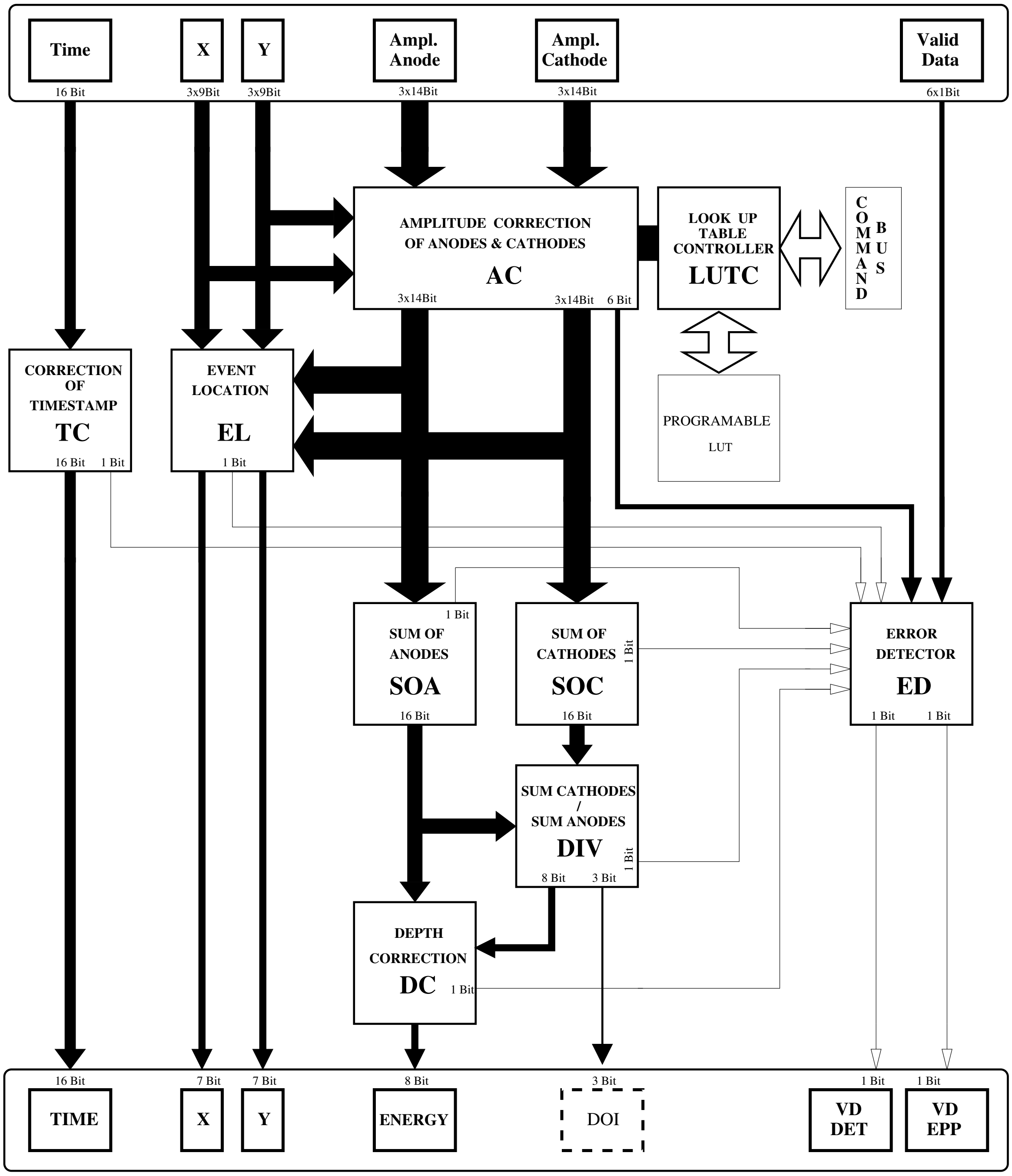


Abbildung 4.16: *Die EC-Unit im Blockschaltbild. An Ein- und Ausgang sind schematisch die Event-Packets zu sehen, dazwischen die Verarbeitungsstufen, die Module. Die beiden Module EC_IN und EC_OUT sind in dieser Darstellung nicht enthalten, sie dienen vor allem der Zerlegung bzw. Gruppierung der Event-Packets an Ein- und Ausgang der Unit. Mit einer Onboard-Korrektur des Tiefenfehlers könnte die Telemetrieübertragung der DOI ggf. ganz entfallen. Die Datenreduktion innerhalb der EC-Unit beruht auf der Verkürzung der Event-Packets. Während das Event-Packet am Eingang noch 168 Bit breit ist, reduziert die EC-Unit dessen Breite auf 45 Bit am Ausgang. Nicht alle Module der EC-Unit können gleichzeitig in Aktion treten. In vielen Fällen ist das Ergebnis vorheriger Module abzuwarten. Damit die Module in der richtigen Reihenfolge Aktiv werden, besitzt die EC-Unit eine interne Ablaufsteuerung, die mit dem Rne-Signal synchronisiert wird.*

Modul	Aufgabe
EC_IN (EC Input Register)	Paket-Übergabe - RNE synchron
AC (Amplitude Correction)	Korrektur der Amplitudenwerte der einzelnen *Hits* der *Event-Packets*
LUTC (Look-Up-Table Controller)	*Controller* des *Look-Up-Table* (LUT)
EL (Event Location)	Bestimmung der Ereignisposition auf dem Detektor
SOA (Sum Of Anode)	Summe der Anodensignale
SOC (Sum Of Cathode)	Summe der Kathodensignale
DIV (Divider)	Dividierer zur Bestimmung der Wechselwirkungstiefe DOI (Depth Of Interaction)
DC (Depth Correction)	Korrektur des Tiefenfehlers der Energiewerte (DOI)
TC (Time Correction)	Korrektur des Zeitstempels des *Event-Packets*
ED (Error Detector)	Generierung von Fehler-Bits
EC_OUT (EC Output Register)	Gruppierung der Ausgangs-*Event-Packets*

Tabelle 4.3: *Die Module der EC-Unit.*

Amplituden-Korrektur:

Die Notwendigkeit für eine Amplituden-Korrektur entsteht aus dem technischen Aufbau des Detektors. Wie beschrieben, besitzt der Detektor auf seiner Oberfläche Elektrodenstreifen, die ein rechtwinkliges Koordinatensystem definieren. Innerhalb dieses Systems kann die Position einer Wechselwirkung auf dem Detektor angegeben werden.

Zu jedem Elektrodenstreifen des Detektors gehört ein eigener Verstärker, der Detektor besitzt insgesamt 1024 davon. Linearitätsmängel der Verstärker erzeugen auf den Signalen leichte Verzerrungen, die sich ungünstig auf die Energiemessung auswirken. Aufgabe der EC-Unit ist deshalb die Korrektur dieser Verstärkerfehler. Hierzu gelangen die *Event-Packets* in die Amplituden-Korrektur (AC-Modul). Das innerhalb des AC-Moduls verwendete Daten-Format ist um den *Timestamp* reduziert. Das AC-Modul arbeitet *Hit* orientiert, d.h es korrigiert die Amplitudendaten in jedem einzelnen *Hit* des *Event-Packets* getrennt. Die Korrekturwerte werden dazu aus einer programmierbaren Eichtabelle, einem sogenannten *Look-Up-Table* (LUT) gelesen. Die Amplitudenkorrektur ist bereits in Kapitel 4.1.1 diskutiert worden. Grundlage ist die dort beschriebene Korrekturgleichung 4.1:

$$E_{out} = A + BE_{in} + CE_{in}{}^2$$

Die Parameter A, B und C sind die Korrekturwerte, die innerhalb des LUT gespeichert werden. Das LUT ist ein eindimensionales *Array*, das für jeden Elektroden-Streifen die drei 8-Bit breiten Werte bereithält, es faßt somit insgesamt 1024 Einträge mit je 24 Bit Breite. Die Korrekturwerte sind nicht einzeln adressierbar, sondern nur pro Elektrode, als Korrektur-Satz. Die Aufteilung der LUT-Einträge auf die Werte A,B und C ergibt sich aus Abbildung 4.17. Alle Korrekturwerte verwenden die gleiche Datenbreite von 8 Bit, beinhalten aber zum Teil sehr unterschiedliche Zahlen-Formate. Um die erforderlichen Wertebereiche dennoch zu speichern, werden die Zweier-Potenzen innerhalb der 8 Bit *Frames* einfach verschoben. Abbildung 4.18 zeigt die derzeit vereinbarten Formate.

Die Information über die Ereignisposition auf dem Detektor trägt das Ortsdatensegment (9 Bit) des jeweiligen *Hits*. Dieses enthält neben der Wechselwirkungs-Position auch gleichzeitig die Adresse für den *Look-Up-Table*. Der Aufbau des Ortsdatensegments ist in Abbildung 4.15 gezeigt. Die ersten 4 Bits (Bit 5 bis 8) charakterisiert das Detektorsegment der Wechselwirkung (Segment-Adresse). Die übrigen 5 Bits

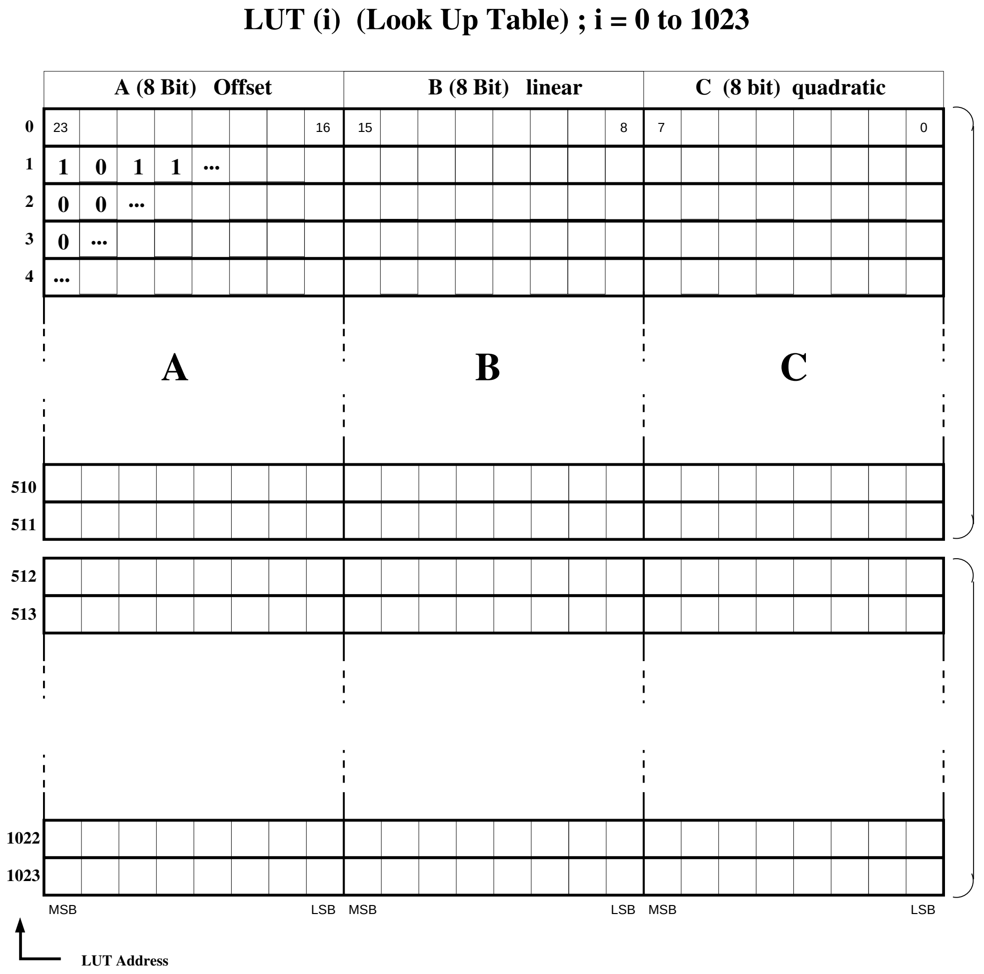


Abbildung 4.17: *Look-Up-Table. Aufteilung der LUT-Einträge auf die Korrekturwerte A, B und C. Die ersten 512 Einträge gehören zu den Anoden, die letzten 512 Einträge zu den Kathoden des Detektors.*

(Bit 0 bis 4) geben die Elektroden-Nummer auf dem Segment an[11] (*Hit*-Adresse).

Für die Lokalisierung eines Ereignisses auf dem Detektor reicht die Koordinateneinteilung der Elektroden in X- und Y-Achse aus. Es ist ein System mit 128 x 128 Koordinaten, was einer Bitdarstellung pro Achse von 7 Bit entspricht. Die Position der Wechselwirkung auf dem Detektor kann deshalb mit zwei 7 bit breiten Datenworten dargestellt werden. Mit Ausnahme der Amplitudenkorrektur arbeiten alle Module und Units des EPP mit dieser 2 x 7 Bit breiten Ereignis-Adresse.

Für die Amplituden-Korrektur reicht diese Adressierung aber nicht aus. Man bedenke, daß wegen der 4 x 4 Anordnung der Detektorsegmente, jede der 128 x 128 Koordinaten des Detektors durch 4 Elektrodenstreifen erzeugt wird. Im Prinzip besitzt eben jedes Detektorsegment sein eigenes Koordinatensystem aus 32 x 32 Koordinaten. Da zu jeder Elektrode ein eigener Verstärker gehört, sind für das LUT viermal mehr Einträge zu adressieren als für die Ereignis-Position. Dies entspricht einem Adressraum von 10 Bit oder einer Darstellung von 2 x 9 Bit, wenn wie zuvor X- und Y-Achse getrennt berücksichtigt werden. Jeder *Hit* enthält nur die Koordinate einer Achse, stellt also entweder ein Anoden- (X-Achse) oder ein Kathoden-

[11] Die *Hit*-Adresse ist die fortlaufende Nummer desjenigen der 32 Anoden- bzw. Kathodenstreifen des Segments, der die Amplitudeninformation des *Hit* abgeleitet hat.

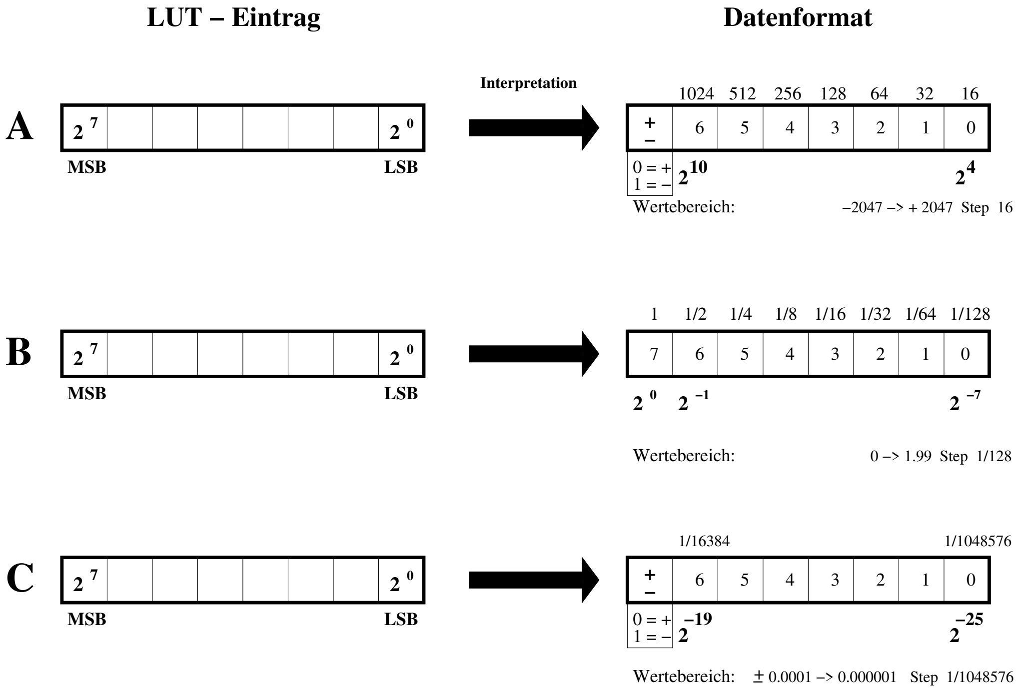


Abbildung 4.18: *Datenformate des LUT. Das LUT stellt für jeden Korrekturwert (A, B und C) nur eine Datenbreite von 8 Bit zur Verfügung. Um dennoch Komma-Typen oder negative Zahlen darin zu speichern, können z.B. die Kommastellen innerhalb der 8 Bit Frames verschoben werden. Dies erreicht man, indem die 2^0-Potenz entsprechend verschoben wird. Bei der Programmierung des LUT ist ansprechend darauf zu achten, daß die Zahlen im Betrieb anders interpretiert werden!*

Signal (Y-Achse) dar. Aus diesem Grund hat auch das Ortsdatensegment eines *Hit* nur eine Breite von 9 Bit, die Zuordnung zur X- oder Y-Achse ist durch den Elektroden-Typ und über die Anordnung der *Hits* im *Event-Packet* festgelegt. Vergleiche hierzu Abbildung 4.14. Die ersten drei *Hits* (Bit 72 bis 143) des *Event-Packets* gehören zur Anode. Sie entsprechen der X-Achse des Detektors und den ersten 512 Einträgen des LUT. Die letzten drei *Hits* gehören zur Kathode (Bit 0 bis 71) und entsprechen der Y-Achse des Detektors, also den zweiten 512 Einträgen des LUT. Der EPP ordnet die *Hits* entsprechend ihrer Position im *Event-Packet* dem richtigen Adressbereich des LUT zu. Auf diese Weise reicht das 9 Bit breite Ortsdatensegment jeden *Hits* für die direkte Adressierung des LUT aus. In Abbildung 4.19 ist ein Beispiel für die Adressierung gegeben.

Zur Gewinnung der 7 Bit breiten Ereignis-Adresse wird die nicht benötigte Segment-Adresse des Ortsdatensegments einfach weggelassen (vgl. Abb. 4.15). Für die Anodenstreifen (X-Achse) ist das die Y-Koordinate der Segment-Adresse (Bit 5 und 6 innerhalb des Ortsdatensegments), für die Kathodenstreifen (Y-Achse) entsprechend die X-Koordinate der Segment-Adresse (Bit 7 und 8). Die 7 Bit breite Adresse ist direkt die X- oder Y-Koordinate der Wechselwirkung innerhalb des 128 x 128 Koordinatensystems.

Der Inhalt des LUT kann vom *LUT-Controller* (LUTC) gelesen und geschrieben werden. Der Schreibzugriff erfolgt über das LBR-*Interface* und ist notwendig, da sich die Detektoreigenschaften, sowie die Eigenschaften der Verstärker im Laufe der Alterung ändern werden. Mit Hilfe der Kalibrationsquelle ist eine regelmäßige Kontrolle der Kennlinien und eine entsprechende Nacheichung möglich. Die jeweils aktuellen Korrekturwerte können von der Bodenstation ermittelt und in den LUT einprogrammiert werden. Da während dieses Schreibvorganges kein sinnvoller Korrekturbetrieb möglich ist, werden *Event-Packets* die in dieser Zeit den EPP durchlaufen, von der AC-Einheit als ungültig markiert und später verworfen.

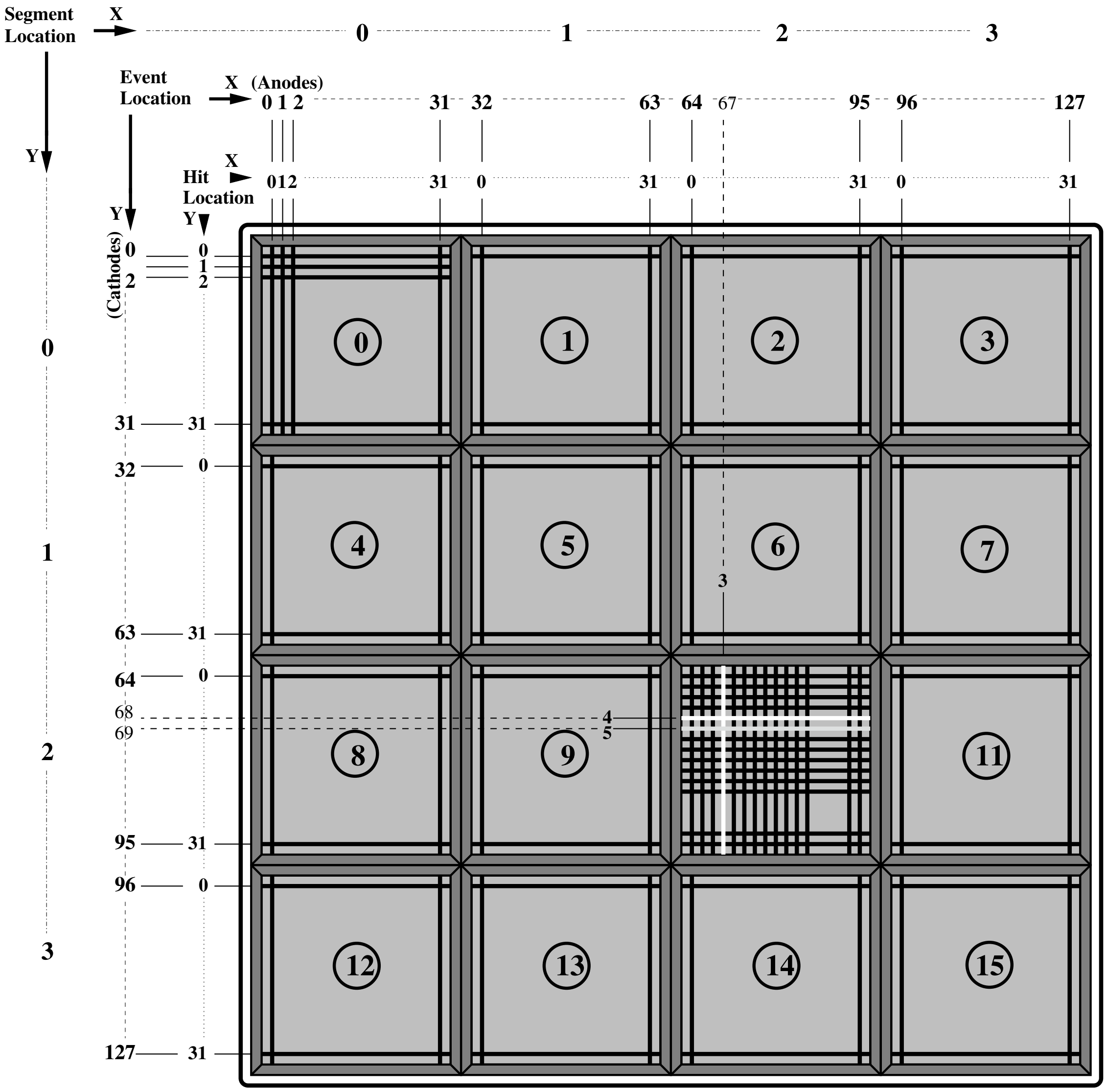


Abbildung 4.19: *Beispiel für die Hit-Erzeugung aus dem Signal einer Wechselwirkung. Die Wechselwirkung wurde in Segment 10 ausgelöst, Anodenstreifen 3 (X-Achse) und Kathodenstreifen 4&5 (Y-Achse) leiten das Signal ab. Die Detektorelektronik verarbeitet diese zu einem Anoden- und 2 Kathoden-Hits. Der Anoden-Hit soll im folgenden näher betrachtet werden. Sein 9 Bit breites Ortsdatensegmente enthält sowohl die Segment- als auch die Event-Adresse. Zusammen mit dem Elektroden-Typ (Position des Hit innerhalb des Event-Packets) ergibt sich so eine 10 Bit breite Adresse (1024 Elektroden). Dabei befinden sich Anoden-Hits immer in den Zeilen 1 bis 3 des Event-Packets (das zehnte Bit ist 0) und Kathoden-Hits in den Zeilen 4 bis 6 (das zehnte Bit ist 1). Der Anoden-Hit des Beispiels besitzt deshalb die Adresse [0 1010 00011], wobei die 9 Bit [1010 00011] aus seinem Ortsdatensegment stammen, und die 0 ganz links aus der Position im Event-Packet. Die 0 ganz links bedeutet, daß sich die Elektrode auf der X-Achse des Detektors befindet, also eine Anode ist. Der zweite Ziffernblock [1010] gibt die Segment-Adresse 2/2 an (Segment 10), der letzte Block [00011], daß es sich um Streifen Nummer 3 handelt. Damit ist die vollständige Adresse der Elektrode festgelegt. Es handelt sich um Anode 3 der X-Achse von Segment 2/2 (Nummer 10) des Detektors. Um lediglich die Position der Wechselwirkung auf dem Detektor zu beschreiben, ist eine 8 Bit Darstellung ausreichend. Diese Event-Koordinate ergibt sich, wenn die nicht benötigte Segment Koordinate weggelassen wird. Für das Beispiel ergibt sich dann die Darstellung [0 10 00011]. Dies entspricht Adresse 67 der X-Achse, wenn die letzten 7 Bit zusammen genommen werden. Wird das 8. Bit wie zuvor in der Position des Hit-Eintrags innerhalb des Event-Packets kodiert, so reicht nach der Adressierung des LUT eine 7 Bit breite Darstellung des Ortsdatensegments aus.*

Zum gegenwärtigen Stand der Entwicklung ist das LUT in Form eines VHDL-*Arrays* realisiert. Dieses liest seinen Inhalt aus einer Festplatten-Datei, sobald ein LOAD-Signal gegeben wird. Diese Simulation eines echten *Look-Up-Table* ist zu gegebenem Zeitpunkt durch eine geeignete VHDL-Realisierung zu ersetzen. Der LUT-Controller ist hierzu ebenfalls zu modifizieren.

Bestimmung der Ereignisposition:

Die Ereignisposition wird im EL-Moduls bestimmt. Hierzu vergleicht das EL-Modul die korrigierten Amplituden der *Hit*-Einträge des *Event-Packets* miteinander. Die ersten drei *Hits* des *Event-Packets* tragen die Koordinaten des Anodensignals, die letzten drei die des Kathodensignals. Das EL-Modul ermittelt aus den jeweils 3 Anode- und Kathoden-*Hits* die beiden mit der größten Amplitude, also die X- und Y-Elektrode, die das größte Signal abgeleitet hat. Die beiden 7 Bit-Adressen dieser *Hits* sind die Koordinaten der Wechselwirkung auf dem Detektor. Sie werden direkt an das EC_Out-Modul geleitet, und von diesem als Ortskoordinate in das Ausgangs-*Event-Packet* der EC-Unit eingefügt. Die Wechselwirkungs-Position auf dem Detektor wird später in der DPU verwendet, um die Himmelskoordinaten der Röntgen-Quelle zu berechnen.

Bestimmung der Gesamtenergie:

Die Gesamtenergie einer Wechselwirkung ist der Anzahl der erzeugten Elektron-Loch Paare proportional.

Jedes Röntgen-Quant das innerhalb des Detektors wechselwirkt, erzeugt dort mittels Photoeffekt Elektronen. Diese ionisieren das Halbleitermaterial, bis sie nach einer Strecke von wenigen Mikrometern zum Stillstand kommen. Bei dem Vorgang wird die gesamte Energie des Röntgen-Quants zur Erzeugung von Elektron-Loch Paaren umgewandelt. Die Elektronen und Löcher werden von den Elektroden abgeleitet. Dabei entsteht ein Signal dessen Amplitude der Energie des Röntgen-Photons proportional ist. Da sich die Ladungswolke durch Diffusion ausdehnt, werden die Ladungen in der Regel von mehr als einer Elektrode abgeleitet. Zur Bestimmung der Energie des Röntgen-Quants sind deshalb die Amplituden aller an der Signalableitung beteiligten Elektroden gleichen Typs (Anode oder Kathode) zu summieren.

Die Signale werden von den Modulen SOA (*Sum Of Anodes*) und SOC (*Sum Of Cathodes*) summiert. Dazu werden die Amplituden von den Ortsdaten getrennt und miteinander addiert. Die Summation erfolgt für die *Hits* von Anode und Kathode getrennt. Bei der Summation der drei 14 Bit breiten Amplitudenwerte erhöht sich der Wertebereich für die Amplitudensumme um jeweils 2 Bit. Das Ergebnis sind die Amplitudensummen SOA und SOC mit einer Breite von jeweils 16 Bit. Die Amplitudensumme der Anoden liefert das Maß für die Energie des Röntgen-Photons und wird an das DC- und DIV-Modul weitergeleitet. Die Kathodensumme wird indessen nur für die DOI-Korrektur herangezogen und gelangt ausschließlich in das DIV-Modul.

Bestimmung der Wechselwirkungstiefe:

Die Korrektur des Tiefenfehlers wurde bereits in Kapitel 4.1.1 besprochen. Die Verwendung von Steuerelektroden verbessert die Signalausbeute des Detektors und erhöht das spektrale Auflösungsvermögen. Allerdings fangen auch die Steuerelektroden einen Teil der Ladungswolke ein. Der Effekt ist besonders stark, wenn die Ladungswolke direkt oberhalb einer Steuerelektrode entsteht. Der hervorgerufene Signalschwund hängt von der Tiefe der Wechselwirkung ab, also davon, wie weit das Ereignis von der Anode aus stattgefunden hat. Aus dem Verhältnis von Kathodensumme (SOC) zu Anodensumme (SOA) laßt sich ein Korrekturverfahren gewinnen. Der Quotient von SOC zu SOA liefert das Signal DOI[12], daß für die Tiefe der Wechselwirkung charakteristisch ist. Das DOI-Signal wird deshalb innerhalb des DIV-Moduls gewonnen, indem die beiden 16 Bit breiten Datenworte SOC und SOA durcheinander dividiert werden. Mit Hilfe

[12]DOI: *Depth-Of-Interaction*; Wechselwirkungstiefe.

des DOI und einem speziellen *Look-Up-Table* kann die Anodensumme, welche der Gesamtenergie entspricht, korrigiert werden. Die Korrektur geschieht in der Bodenstation, das DOI wird deshalb zusammen mit den anderen Daten übermittelt.

Zur Übertragung des DOI sind innerhalb des Ausgang-*Event-Packets* lediglich 3 Bit an Information vorgesehen. Damit lassen sich nur 8 Tiefen übermitteln. Weil die Datenbreite innerhalb des EPP keinen solch starken Restriktionen unterliegt wie die Telemetrie wäre die Korrektur präziser, wenn sie innerhalb des EPP selbst vorgenommen werden könnte. In diesem Fall müßte das *Look-Up-Table* des EPP entsprechend erweitert werden, damit es auch tiefenabhängige Korrekturwerte aufnehmen kann. Das Ergebnis dieser Erweiterung wäre nicht nur eine deutlich feinere Abstufung bei der Tiefenkorrektur, auch der *Burst*-Detektor könnte dann mit bereits DOI-korrigierten Werten betrieben werden, was auch die Genauigkeit bei der *Burst*-Suche verbessern würde.

Tiefen-Korrektur und Amplituden Diskriminator:

Unter der Annahme, daß der Tiefenschwund des Signals in erster Näherung mit der Tiefe linear ist, wurde eine vereinfachte Stufe zur Tiefenkorrektur bereits innerhalb des EPP verwirklicht, das DC-Modul. Das DC-Modul ist an das DIV-Modul angeschlossen und erhält den DOI-Wert mit einer Genauigkeit von 8 Bit, anstatt von Drei. In erster Näherung ist das DOI gerade der Quotient um den das Energie-Signal durch den Tiefenschwund dezimiert wurde. Deshalb multipliziert das DC-Modul den DOI-Wert einfach mit der Energie und gleicht damit den Schwund wieder aus. Das DOI wird innerhalb des DC-Moduls mit 8 Bit verarbeitet. Die Multiplikation mit der 16 Bit breiten Anodensumme SOA führt zu einem 24 Bit breiten Daten-Wort, dem SOA-DOI-Produkt, oder kurz SD-Produkt.

Eine andere Aufgabe des DC-Moduls ist die Reduzierung der Energiebreite. Die Energie-Information liegt ohne DOI-Korrektur mit einer Breite von 16 Bit vor, mit DOI-Korrektur sind es sogar 24 Bit. Für das Ausgang-*Event-Packet* des EPP ist jedoch nur eine Breite von 8 Bit vorgesehen, vergleiche hierzu Abbildung 4.22. Die Reduktion der Energie-Information geschieht innerhalb des DC-Moduls. Der 8 Bit breite Energie-Ausschnitt aus dem 16, bzw. 24 Bit breiten SD-Produkt wird dabei so gewählt, daß die physikalisch nutzbare Information möglichst wenig beeinträchtigt wird.

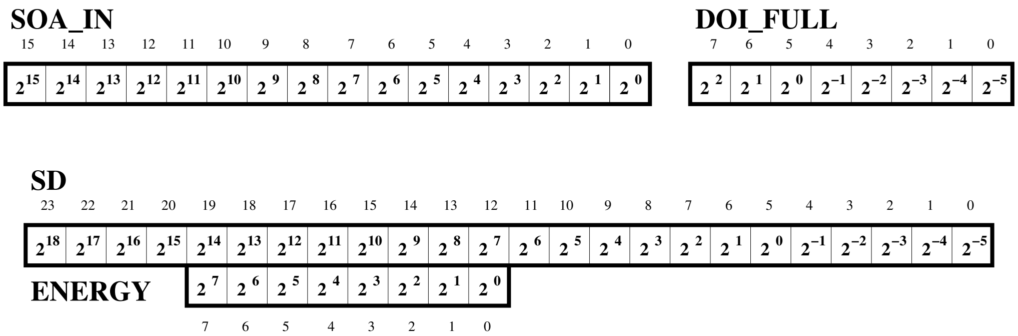


Abbildung 4.20: *Interne Datenformate des DC-Moduls. Das Eingang-Signal* SOA_IN *wird mit dem 8 Bit breiten DOI-Signal* DOI_FULL *multipliziert. Das Produkt ist ein 24 Bit breites Datenwort (SD-Produkt) aus dem die 8 Bit breite Energie-Information herausgeschnitten wird.*

Die derzeitige Version des EPP-*Core* arbeitet mit der DOI-Korrektur, d.h. der 8 Bit breite Energie-Ausgang wird aus dem 24 Bit breiten SD-Produkt ausgeschnitten. Sollte die DOI-Korrektur wie ursprünglich geplant am Boden erfolgen, muß entweder der DOI-Eingang des DC-Moduls konstant auf Eins gesetzt werden (dann wird die Energie immer mit der entsprechenden 8 Bit Darstellung von 1 multipliziert), oder die Korrektur-Stufe (DC-Modul) neu entworfen werden.

Abbildung 4.20 zeigt nacheinander die Datenformate innerhalb des DC-Moduls. Neben dem Eingangssignal SOA_IN in seiner 16 Bit Darstellung und dem 8 Bit breiten DOI_FULL-Signal, ist das SD-Produkt, und das 8 Bit breite Energie-Ausgangs-Signal ENERGY zu sehen. Die Auswahl des Energie-Ausschnitts aus dem SD-Produkt geschieht so, daß die Vor-Kommastellen des Energie-Wertes nicht beeinträchtigt werden. Die endgültige Auswahl des Energie-Fensters ist erst möglich, wenn das Format der Energie in seiner 14 Bit Darstellung am Eingang der EPA-Unit vollständig spezifiziert ist.

Die Erkennung von Verarbeitungsfehlern:

Der *Error*-Detektor (ED-Modul) ist ein wichtiger Bestandteil der EC-Unit. Das ED-Modul dient gleichzeitig der Reduktion und der Erhebung des Fehler-Status. Am Eingang des EC-Moduls verfügt das *Event-Packet* zunächst in jedem *Hit* über ein VD-Bit (VD-Det; vgl. Abb. 4.14). Dieses Bit trägt den Fehler-Status des jeweiligen *Hit*. Ist das VD-Det gleich Eins, so ist der *Hit* gültig, ist es Null, so wurde ein Fehler detektiert (z.B. ein Koinzidenz-Problem mit anderen *Hits*). Gleich zu Beginn der EC-Unit kommt für jeden *Hit* ein weiteres VD-Bit hinzu, das VD-AC (vgl. Abb. 4.21). Das VD-AC charakterisiert Fehler, in der Amplituden-Korrektur, und wird nur vom AC-Modul geschrieben. Wegen der *Hit*-weisen Bearbeitung innerhalb des AC-Moduls, wird auch der Fehler *Hit*-weise erhoben. Mit Hilfe der beiden VD-Bits ist die Fehlerquelle zwischen Detektor und Amplituden-Korrektur unterscheidbar. Vergleiche das Eingang-*Event-Packet* der EC-Unit, Abbildung 4.14 und das interne *Event-Packet* am Ausgang des AC-Moduls, Abbildung 4.21. Am Eingang des ED-Moduls können Fehler noch bis in den *Hit*-Eintrag des *Event-Packets* verfolgt werden. Diese Information geht innerhalb des ED-Moduls verloren. Im Zuge der Datenreduktion werden die 12 VD-Bit auf 2 pro *Event-Packet* reduziert. Am Ende der EC-Unit kann als Fehlerquelle nur noch zwischen Detektor und EPP unterschieden werden (vgl. Abb. 4.21).

Für die Reduktion der Fehler-Bits ist das ED-Modul zuständig. Es verarbeitet die VD-Det- und VD-AC-Bits der einzelnen *Hits* und steht über Signalleitungen mit allen Modulen der EC-Unit in Verbindung (außer dem AC-Modul; vergleiche hierzu Abbildung 4.16). Innerhalb des ED-Moduls werden die VD-Det-Bits der 6 *Hits* auf ein einziges VD-Det-Bit des Ausgang-*Event-Packets* abgebildet. Ist auch nur eines der Bits Null, so wird auch das VD-Det-Bit des Ausgang-*Event-Packets* Null. Es reicht ein einziger ungültiger *Hit* in den Eingangsdaten, und das gesamte *Event-Packet* wird ungültig. Das gleiche gilt für die Reduktion der VD-AC-Bits. Auch diese werden vom ED-Modul erfaßt und gehen zusammen mit den Fehler-Signalen anderer Module in das VD-EPP-Bit des Ausgang-*Event-Packets* ein. Hier gilt genauso: Ist auch nur ein einziges VD-AC-Bit Null, oder wird auf einer Signalleitungen ein Fehler signalisiert, so ist das VD-EPP-Bit des *Event-Packets* Null, und das *Event-Packet* ungültig.

Mit der Reduktion geht viel Information über die Fehlerquelle verloren. Das *Event-Packet* am Ausgang der EC-Unit ist in Abbildung 4.22 gezeigt. Es existieren insgesamt nur noch zwei VD-Bits für das gesamte *Event-Packet*. Während innerhalb des AC-Moduls die Fehler noch für jeden *Hit* einzeln erfaßt und gespeichert werden, erlauben die beiden VD-Bits des Ausgang-*Event-Packets* nur noch die Unterscheidung zwischen Detektor- und EPP-Fehlern. Um die Information über die genaue Fehlerursache dennoch zu bewahren, schlage ich die Erweiterung des Ausgangs-*Event-Packets* um einen 2 bis 3 Bit breiten Fehler-*Code* vor. Ein solcher Fehler-*Code* könnte bei Bedarf der Telemetrie angehängt werden und würde der Bodenstation eine detaillierte Diagnosemöglichkeit bieten. Der *Error-Code* würde im Normalbetrieb keine Erhöhung der Telemetrie oder der Datenbreite des *Ring-Buffers* erfordern, da ungültige *Events* im Normalbetrieb ohnehin eliminiert werden.

AC–Modul REG_OUT

	Amplitude 14 Bit (corrected)			Location 7 Bit		VD DET	VD AC
Anodes	137	HIT 1	124	123	117	116	115
Anodes	114	HIT 2	101	100	94	93	92
Anodes	91	HIT 3	78	77	71	70	69
Cathodes	68	HIT 4	55	54	48	47	46
Cathodes	45	HIT 5	32	31	25	24	23
Cathodes	22	HIT 6	9	8	2	1	0
	MSB		LSB	MSB	LSB		

Abbildung 4.21: *Internes Event-Packet des AC-Moduls. Das AC-Modul hängt an die Hit-Einträge des Event-Packets jeweils ein weiteres VD-Bit an (VD-AC). Dieses dient einer späteren Unterscheidbarkeit der Fehlerquelle. Am Ende der EC-Unit geht der Status aller 12 VD-Bits (VD-Det/VD-AC) auf die VD-Bits (VD-Det/VD-EPP) des Ausgang-Event-Packets über (vgl. Abb. 4.22). Der Error-Detektor (ED-Modul) sammelt dazu alle Fehler-Meldungen der EC-Unit und verarbeitet diese, zusammen mit den bereits vorhandenen, zu den beiden VD-Bits des Ausgang-Event-Packets (Abb. 4.22). Das Event-Packet in der internen Form enthält keine Zeitinformation (Timestamp) mehr. Diese wurde bereits abgezogen und wird vom TC-Modul getrennt verarbeitet. Zu beachten ist auch, daß das Ortsdatensegment bei Verlassen des AC-Moduls bereits 7 Bit breit ist, im Vergleich zu einer Breite von 9 Bit am Eingang der EC-Unit (vgl. Abb. 4.14). Nach der Adressierung des LUT wird die Segment-Adresse nicht weiter benötigt. Das AC-Modul reduziert die Adressierung deshalb auf die Event-Adresse, die fortan mit 7 Bit auskommt (siehe Text).*

Ausgangs *Event-Packet* der EC-Unit:

Nachdem alle Korrekturen und Berechnungen abgeschlossen sind, werden die Datensegmente im EC_OUT-Modul zu einem neuen *Event-Packet* zusammengeführt. Dieses Ausgangs-*Event-Packet* (Abb. 4.22) hat nur noch eine Bitbreite von 45 Bit. Im Vergleich mit dem *EPA-Event-Packet* am Eingang (Abb. 4.14) ist deutlich die Datenreduktion erkennbar. Das neue *Event-Packet* beinhaltet der Reihe nach folgende Daten:

- Zeitstempel des Ereignisses (16 Bit)
- X-Koordinate der Wechselwirkung (7 Bit)
- Y-Koordinate der Wechselwirkung (7 Bit)
- Energie des Röntgen- oder Gammaphotons (8 Bit)
- Wechselwirkungstiefe DOI (3 Bit).
- Fehler-Bits VD-Det und VD-EPP (2 Bit)
- Fehler-*Code* (2 Bit)

Event-Packets mit einem oder mehreren ungültigen VD-Bits werden in der nachfolgenden FD-Unit ausgefiltert. Auf diese Art wird bereits ein Teil des Störhintergrundes des Detektors unterdrückt, da vor allem Stör-Ereignisse, zu ungültigen *Event-Packets* führen.

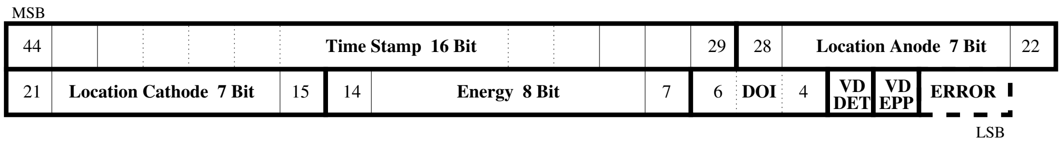


Abbildung 4.22: *Das Event-Packet am Ausgang der EC-Unit. Deutlich ist die Datenreduktion im Vergleich mit dem Event-Packet am Eingang (Abbildung 4.14) zu erkennen. Die beiden Fehler-Bits VD-Det und VD-EPP ermöglichen nur die Unterscheidung zwischen Detektor- und Verarbeitungsfehlern. Ein zusätzlicher Fehler-Code könnte im Bedarfsfall weitere Informationen übertragen.*

Die Übergabe des Ausgang-*Event-Packets* an die nachfolgende FD-Unit erfolgt mit dem RNE-Signal. Dies bedeutet, daß alle Korrekturen und Berechnungen der EC-Unit innerhalb eines *Pipeline*- oder RNE-Taktes abgeschlossen sein müssen. Der *Pipeline*-Takt ist ggf. auf die Bedürfnisse innerhalb der EC-Unit anzupassen.

4.3.4 FD-Unit (*Filter and Decision Unit*)

Die FD-Unit dient der Unterscheidung und der Auslese gültiger von ungültigen Ereignis-Paketen. Sie besitzt programmierbare Filter zur Störungsunterdrückung und zur Einengung der Energiebandbreite, für die *Burst*-Suche. Als Eingangsignal dient das *Event-Packet* wie in Abbildung 4.22 dargestellt. Die Module der FD-Unit verarbeiten, anders als diejenigen der EC-Unit, die *Event-Packets* als ganzes. Auch wird die Bitlänge, also die Informationsbreite, nicht verändert. Die Datenreduktion der FD-Unit liegt in einer Auswahl der Pakete, da bei weitem nicht jedes *Event-Packet* ein gültiges Ereignis darstellt.

Modul	Aufgabe
FD_IN (FD Input Register)	Buffer für Eingangs-*Event-Packets*
THN (Threshold Noise)	Energie Filter für Hintergrundstörungen
RAMC (RAM Controller)	Controller für Schwellwerteinstellungen
VD (Veto Detector)	Filter für ungültige *Event-Packets*
BS (Burst Snooper)	Energie Filter für Gamma Ray Burst Detektion
FD_OUT (FD Output Register)	Register und Weiche für Ausgangs-*Event-Packets*

Tabelle 4.4: *Die Module der FD-Unit*

Störungsunterdrückung:

Das THN-Modul dient der Störungsunterdrückung. Dies geschieht wie die Fehler-Handhabung in der EC-Unit durch die Verwendung von Markern. Zu diesem Zweck wird die Energie der *Event-Packets* mit Sollwerten verglichen, und das Ergebnis entsprechend seiner Gültigkeit markiert. Durch die Filterung kann ein Großteil der Stör-Ereignisse über den Energie-Werte eliminiert werden. An das *Event-Packet* wird dazu ein drittes VD-Bit angehängt, es trägt das Ergebnis der Energieuntersuchung. So wie die anderen VD-Bits, wird auch das VD-*Noise*-Bit erst innerhalb des Veto Detektors (VD-Modul) der FD-Unit ausgewertet. Ist das VD-*Noise*-Bit gleich Eins, so erfüllt die Energie die Anforderungen des Filters, liegt der Energiewert dagegen außerhalb, ist das VD-*Noise*-Bit gleich Null. Das THN ist wie frühere Module auch, mit voller Absicht nicht als Filter sondern als Marker konstruiert worden. Die Entscheidung über das Schicksal der *Event-Packets* kann so auf eine einzige Stufe am Schluss der Verarbeitungs-*Pipeline* übertragen werden. Diese Vorgehensweise ermöglicht maximale Flexibilität bei der Auswahl der zu telemetrierenden Daten. Das Datenformat des Ausgang-*Event-Packets* ist in Abbildung 4.24 dargestellt.

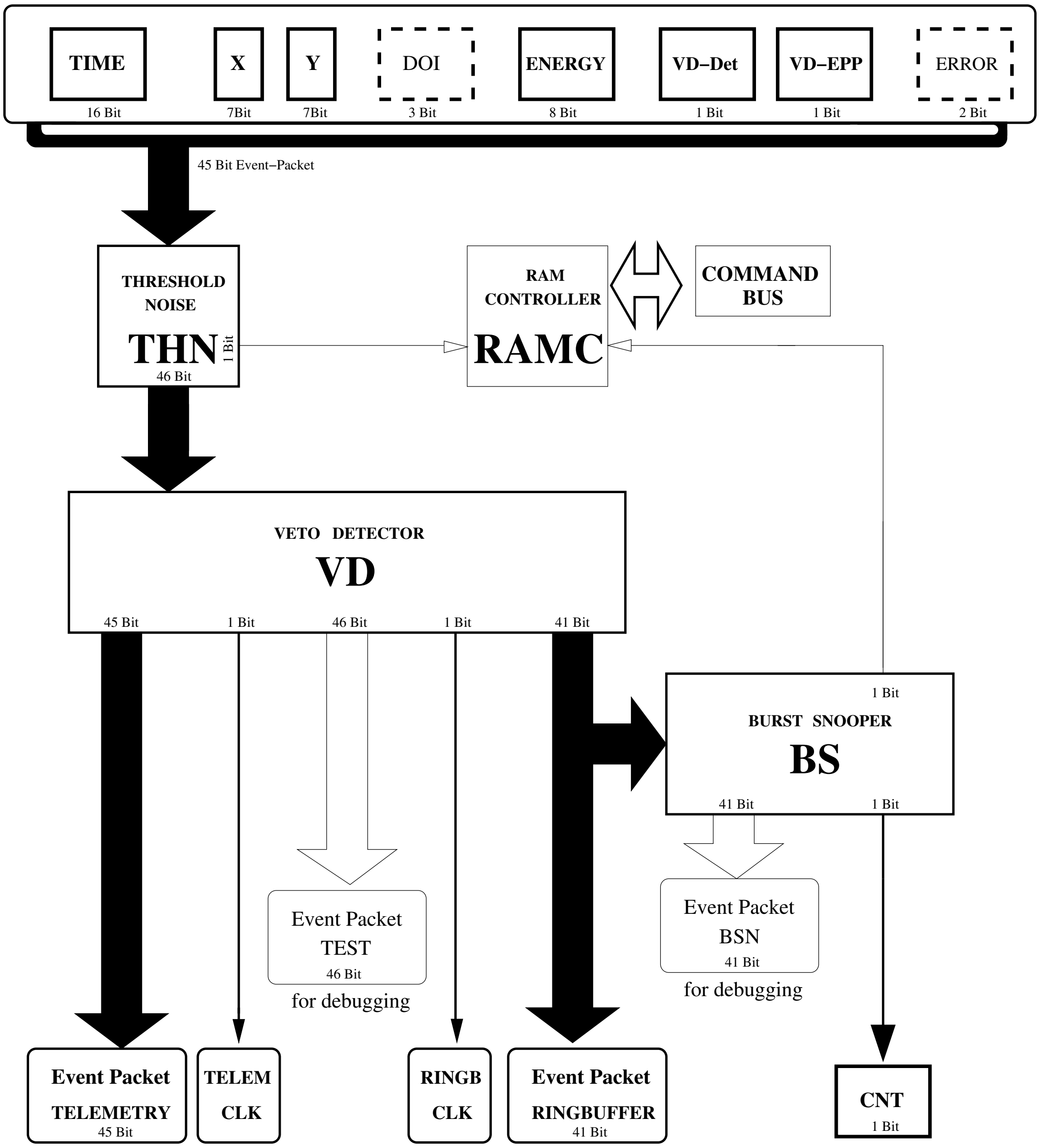


Abbildung 4.23: *Die FD-Unit im Blockschaltbild. Innerhalb dieser Unit werden die* Event-Packets *als ganzes verarbeitet. Die Datenreduktion besteht nicht darin, die Paketlänge zu verringern, sondern Pakete in gültige und ungültige Ereignisse zu klassifizieren, also zwischen den Paketen selbst eine Auswahl zu treffen. Die Entscheidung darüber, ob ein Paket vernichtet wird oder in die Auswertung gelangt, trifft der Veto-Detektor (VD-Modul). Alle vorhergehenden Module arbeiten nicht als Filter sondern als Marker. Dies ermöglicht, daß die Entscheidung erst am Ende der Verarbeitungskette innerhalb des VD-Moduls getroffen wird, und eröffnet maximale Flexibilität bei der Auswahl der zu telemetrierenden Daten. Die Module FD_IN und FD_OUT sind in dieser Darstellung nicht enthalten, sie regeln die Paketübergabe von der vorhergehenden und an die nachfolgenden Units synchron zum* Pipeline*-Takt.*

Die Sollwerte des THN-Moduls können von der Bodenstation programmiert werden. Anders als beim LUT ist während dessen kein Veto der *Event-Packets* notwendig. Die THN-Werte werden mit Hilfe des RAMC-Moduls vom *Telecommand*-Bus gelesen und in einem seiner Register gehalten.

Der Veto Detektor:

Das VD-Modul trifft die Entscheidung welche *Event-Packets* in die Auswertung gelangen, und welche verworfen werden. Als Kriterium werden die drei VD-Flags verwendet, die den Status des *Event-Packets* anzeigen (vgl. Abb. 4.24). Das VD-Modul übernimmt dabei nicht nur die Aufgabe eines Filters sondern auch die einer Weiche. So können z.B. Pakete, die zwar im Sinne des *Noise*- (VD-*Noise*-Bit) und Detektor-Flag (VD-Det-Bit) gültig sind, innerhalb des EPP einen Überlauf auslösen, und ein ungültiges EPP-Flag (VD-EPP) besitzen. In diesem Fall darf das Paket nicht in den Ring-Speicher oder den *Burst*-Zähler gelangen. Es könnte aber dennoch im Bedarfsfall in die Telemetrie geleitet werden, und der Bodenstation Informationen über den Status des Detektors oder des EPP geben. Ein Umschalten der Telemetrie zwischen Paketen mit unterschiedlichem VD-Status ist bis jetzt nicht implementiert worden, aber im Sinne einer Erweiterung auf einen Modus-Abhängigen Betrieb bereits vorgesehen.

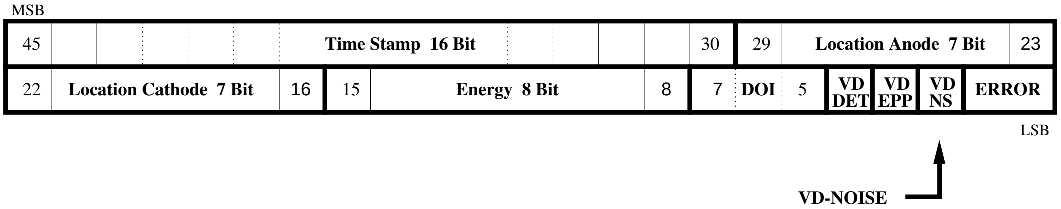


Abbildung 4.24: *Das Event-Packet am Schluss der Verarbeitungskette. Innerhalb des Veto Detektors können die Pakete entsprechend ihren VD-Flags zur Übertragung ausgewählt, gespeichert oder verworfen werden. Diese Entscheidung ist bisher fest implementiert, eine programmierbare Auswahl im Rahmen verschiedener Betriebsmodi des EPP ist jedoch auch denkbar.*

Zum derzeitigen Stand der Arbeit gelangen ausschließlich Pakete mit gültigen VD-Flags in die Telemetrie und den Ring-Speicher. Für beide existieren getrennte Ausgänge mit verschiedener Bitbreite (vgl. Abb. 4.23). Dies ist erforderlich, weil der Telemetrie-Ausgang im Bedarfsfall auch ungültige Pakete (mit VD-Flags, *Error-Code*) transportieren können soll. Der Ring-Speicher erhält in jedem Fall nur gültige Pakete. In der Entwicklungs- und Test-Phase des EPP-*Core* existiert noch kein Telemetrie-*Interface* (HBR-EPP/DPU Unit), die Daten werden von der VHDL Simulation auf Festplatte gespeichert.

Der *Burst Snooper*:

Zum Auffinden von *Gamma-Ray Bursts* (GRB) muß die Zählrate der gültigen Ereignisse innerhalb eines definierten Zeitintervalls ermittelt werden. Steigt die Zählrate gegenüber dem Mittelwert der Durchschnitts-Zählrate stark an, so liegt mit hoher Wahrscheinlichkeit ein GRB vor (vgl. Kapitel 4.1.2).

Als Ausgangspunkt für die *Burst*-Suche wird die Zahl aller gültigen *Event-Packets* herangezogen. Zu diesem Zweck liest das BS-Modul die Ereignis-Pakete, die vom Veto-Detektor in Richtung Ring-Speicher gehen. Diese stellen die Gesamtmenge aller gültigen Ereignisse dar. Für die *Burst*-Suche möchte man aber

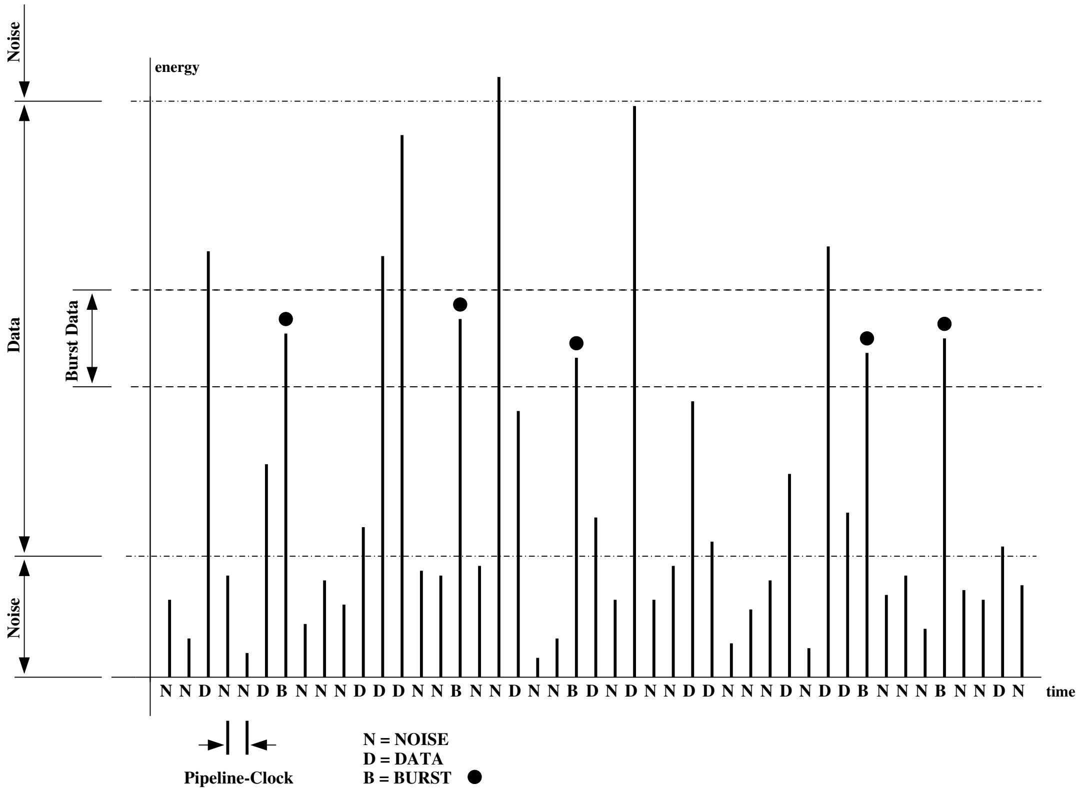


Abbildung 4.25: *Die Wirkung der Filter für Noise-Unterdrückung und Burstauszählung auf die Zahl der Event-Packets. Die meisten Event-Packets stellen Ereignisse des Störhintergrundes dar. Nur eine kleine Auswahl der Event-Packets schafft es bis in die Auswertung. Von diesen wählt ein weiteres Energie-Filter die für die Burst-Suche interessanten Event-Packets aus (Burst-Snooper).*

den Energiebereich der Ereignisse noch weiter einengen. Hierzu besitzt das BS-Modul ein zusätzliches Energiefilter. Die für den *Burst*-Detektor relevanten Pakete lassen sich damit weiter einschränken. In die Auszählung gelangen deshalb nicht alle gültigen Pakete, sondern nur solche im Energiebereich des *Burst*-Detektors. Dieser erzeugt am Eingang der BD-Unit Zähl-Impulse.

Die Wirkung der beiden Energiefilter des BS- und THN-Moduls auf die Zahl der *Event-Packets* ist in Abbildung 4.25 zu sehen. Das Energiefilter des *Burst-Snoopers* ist vom Energiefilter der Störungsunterdrückung (THN-Modul) entkoppelt. Die Einstellung des *Burst*-Fensters geschieht mit Hilfe des RAMC-Moduls, genau wie im Fall der THN-Stufe.

Ausgang *Event-Packets* der FD-Unit:

Die Ausgangs-Signale der FD-Unit stehen innerhalb eines *Pipeline*-Taktes zur Verfügung und werden von den nachfolgenden Units synchron mit dem RNE-Signal gelesen. Ebenso wie die EC-Unit ist auch die FD-Unit ein Teil der Verarbeitungs-*Pipeline*. Alle Berechnungen und Vergleiche müssen innerhalb eines *Pipeline*-Taktes ausgeführt und beendet sein. Der *Pipeline*-Takt hat die Länge von 100 *Clock*-Impulsen.

Die FD-Unit hat zwei *Event-Packet*-Ausgänge, und einen 1 Bit breiten Zählerausgang. Die *Event-Packet*-Ausgänge sind der Telemetrie- und der *Ringbuffer*-Ausgang. Der Telemetrie-Ausgang wird vom HBR-EPP/DPU-*Interface* gelesen, er ist 45 Bit breit und befördert die laufenden Messdaten, die zur Bo-

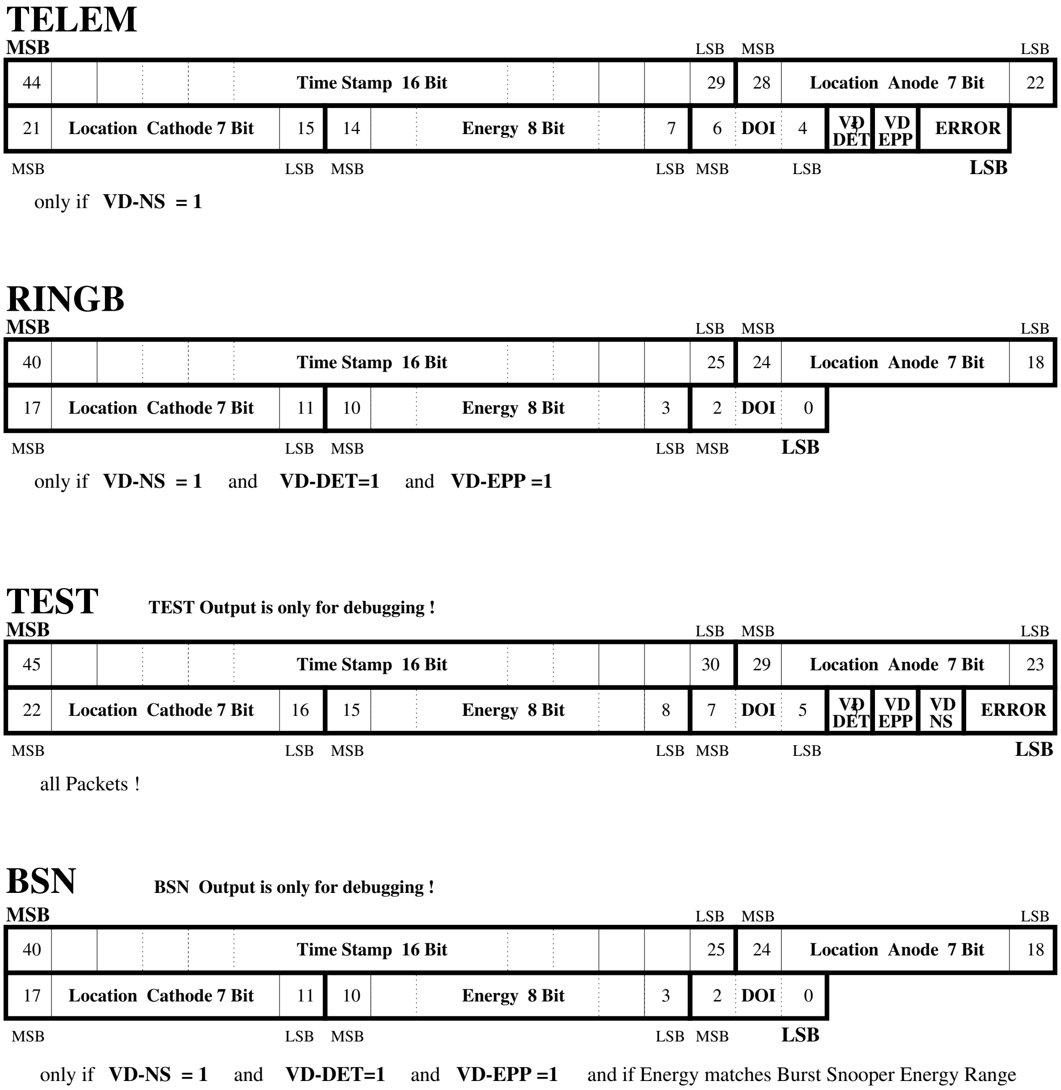


Abbildung 4.26: *Die Ausgang-Event-Packets der FD-Unit. Wichtig für den späteren Betrieb des EPP sind die Event-Packets* TELEM *und* RINGB. *Das Telemetrie-Event-Packet ist 45 Bit breit und enthält bisher neben den Event-Daten auch die VD-Bits und den Error-Code. Die Error-Bits könnten im Bedarfsfall zum Boden übertragen werden, im Normalbetrieb werden sie später unterdrückt. Das Event-Packet für den Ring-Buffer ist 41 Bit breit. Die beiden übrigen Event-Packets* TEST *und* BSN *dienen für Diagnose-Zwecke während der Entwicklungsphase des EPP. Der* TEST*-Ausgang enthält alle Event-Packets unzensiert (ohne VD-Bit Auswertung), der* BSN*-Ausgang diejenigen die der Burst-Snooper gefiltert hat.*

denstation gefunkt werden. Der *Ringbuffer*-Ausgang ist 41 Bit breit, er befördert *Event-Packets* zur RB-Unit. Da hier ausschließlich gültige Pakete transportiert werden, können die VD-Bits entfallen, was die Bus-Breite auf 41 Bit reduziert und Speicherplatz innerhalb des Ring-Speichers spart. Ein 1 Bit breiter Zählausgang versorgt schließlich die BD-Unit mit Zählimpulsen. Der Ausgang geht auf Eins, wenn innerhalb des zugehörigen *Pipeline*-Taktes, ein gültiges *Burst-Event-Packet* vorhanden ist. Die Abfrage erfolgt mit dem RNE-Impuls. Ist zu diesem Zeitpunkt der Ausgang auf Eins, so wird ein Zählimpuls an die BD-Unit durchgestellt, ist der Ausgang auf Null, erhält die BD-Unit bei RNE kein Signal.

Zum Zeitpunkt der Abgabe dieser Arbeit existieren noch zwei zusätzliche temporäre Ausgänge für *Debugging*-Aufgaben. Die *Debugging*-Ausgänge dienen während der Entwicklungs- und Testphase der Systemdiagnose im laufenden Betrieb, und können nach Fertigstellung des EPP stillgelegt werden. Als Test- und *Debugging*-Ausgang existiert ein 46 Bit breiter TEST-Ausgang, der alle *Event-Packets* unabhängig von der Gültigkeit ausgibt, sowie ein 41 Bit breiter BSN-Ausgang der *Event-Packets* ausgibt, deren Energie im Sinne des *Burst-Snoopers* gültig ist. Mit dem BSN-Ausgang kann die Funktion des *Burst-Snoopers* überwacht werden.

Eine Übersicht der Ausgänge ist in Abbildung 4.23 gezeigt, eine Darstellung der Daten-Formate der vier Ausgänge in Abbildung 4.26.

4.3.5 BD-Unit (*Burst Detection Unit*)

Die primäre Aufgabe des CMI ist die Suche nach *Gamma-Ray Bursts*. Dazu wird aus der Gesamtmenge aller gültigen *Event-Packets* eine Untermenge herausgefiltert, deren Energie innerhalb des Energiefensters des *Burst-Snoopers* liegt. Das Beobachtungsfenster kann damit auf einen für die *Burst*-Suche besonders günstigen Energiebereich eingestellt werden. Der Vorgang entspricht in etwa dem Zuschalten eines Farbfilters in der Optischen-Astronomie. Eine spektrale Einschränkung dieser Art erhöht in der Regel das Signal-Rausch-Verhältnis. Die gefilterten *Event-Packets* bilden die Basis für die Zählratenanalyse, und werden innerhalb des *Event*-Zähler ausgezählt. Für den *Event*-Zähler stehen 4 Torzeiten zur Verfügung. Die Torzeiten werden von der Bodenstation eingestellt und definieren die Länge der Zählintervalle (1/8s, 1/2s, 1s, 2s). Es wird für Intervalle stets gleicher Länge die Zählrate ermittelt. In Kapitel 4.1.2 wurde die statistische Auswertung der Zählrate untersucht, sie entspricht einer Poisson-Verteilung, wobei das Maximum der Verteilung den Röntgenhintergrund wiederspiegelt. Er definiert den statistischen Mittelwert aller echten Röntgen- und Gamma-Ereignisse und ist vom sogenannten Störhintergrund überlagert. Dieser setzt sich aus dem diffusen Röntgen- und Gamma-Hintergrund und aus diversen lokalen Störquellen zusammen. Der Störhintergrund wird von den vorherigen Units weitgehend unterdrückt.

Ein deutlicher Anstieg der Zählrate gegenüber dem Mittelwert kann mit hoher Wahrscheinlichkeit einem *Gamma-Ray Burst* zugeschrieben werden. Zur Entdeckung von GRBs ist die Zählrate deshalb nach starken Abweichungen von ihrem Mittelwert zu untersuchen. Die Bestimmung des Mittelwertes sowie der Größe der Abweichung ist Aufgabe der BD-Unit.

Der *Event*-Zähler:

Der *Event*-Zähler besteht aus vier VHDL-Modulen (vgl. Abb. 4.27). Die Zählimpulse, die vom BS-Modul der FD-Unit stammen, gelangen zunächst in das GATE-Modul. Das GATE hat zwei Aufgaben: Es legt die Zeitdauer fest, während der die Zähler offen sind, und es verteilt die Zählimpulse auf zwei identische Zähler die abwechselnd aktiv sind.

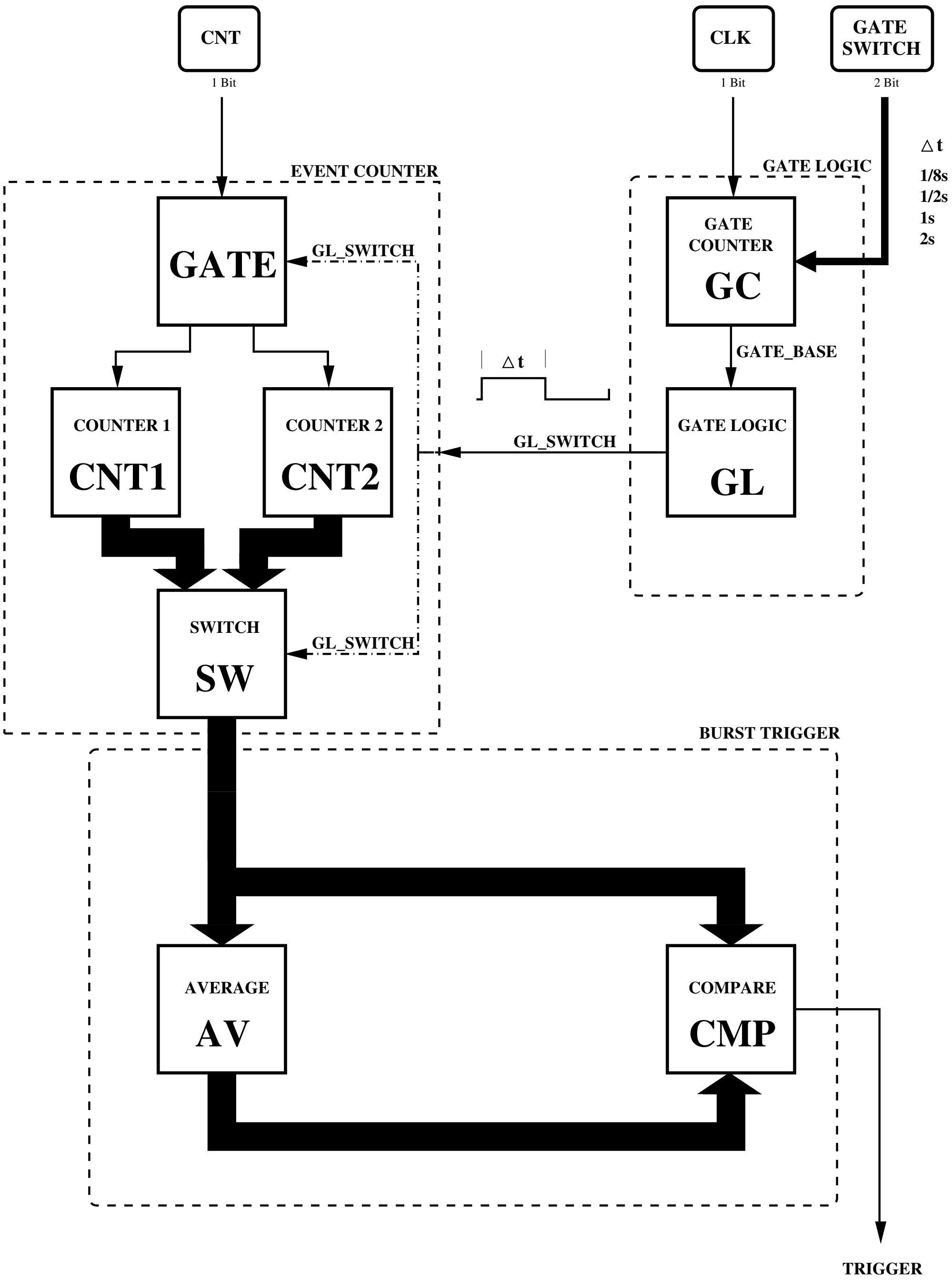


Abbildung 4.27: *Die BD-Unit im Blockschaltbild. Innerhalb dieser Unit werden die mittleren Zählraten berechnet und mit der aktuellen Zählrate verglichen. Übersteigt die Abweichung einen bestimmten Wert (5 Sigma), so wird der GRB-Alarm ausgelöst. Oben links ist der Event-Counter mit den beiden Zählern CNT1 und CNT2 dargestellt. Die Module GATE und SW schalten die Zähl-Impulse zwischen den beiden Zählern um. Rechts oben sind die Module zur Erzeugung der Steuersignale, und unten die eigentliche Burst-Trigger-Einheit, die die Zählraten analysiert, dargestellt. Das BD_IN-Modul ist aus Gründen der Vereinfachung nicht abgebildet. Es ist dem Event-Counter vorgeschaltet und synchronisiert die Zählimpuls-Übergabe mit der FD-Unit.*

Modul	Aufgabe
BD_IN	Synchronisation der Zählimpulsübergabe
GATE	Tor des Zählers, ist die aktive Zeit des Zählers offen
CNT (Counter)	Zähleinheit für *Event-Packets*
SW (Switch)	Umschaltung der Zähler
AV (Average)	Modul zur Ermittlung der Durchschnittszählrate
CMP (Compare)	Vergleicher, vergleicht die aktuelle Zählrate mit dem Mittelwert
GC (Gate Counter)	Modul zur Erzeugung der Torzeit aus dem Clock-Signal
GL (Gate Logic)	Modul zur Erzeugung der Steuersignale der BD-Unit

Tabelle 4.5: *Die Module der BD-Unit*

Das GATE-Modul ist den beiden Zählermodulen CNT1 und CNT2 vorgeschaltet. Es leitet die Zählimpulse direkt in einen der beiden Zähler. Welcher Zähler gerade aktiv ist, wird vom GL-SWITCH-Signal festgelegt, das innerhalb des GL-Moduls erzeugt wird. Das GL-SWITCH-Signal definiert die Torzeit des Zählers, es hat je nach Einstellung eine Länge von 1/8 bis 2 Sekunden. Die spezielle Betriebsweise mit zwei Zählern hat den Vorteil, daß nahezu keine Totzeit des *Event*-Zählers entsteht. Während der eine Zähler die *Events* zählt, kann der andere sein vorheriges Ergebnis sichern und den Zählerstand löschen. Die Arbeitsweise des *Event-Counters* ist in Abbildung 4.28 dargestellt.

Die Übergabe der Zähl-Impulse erfolgt RNE-synchron, d.h. mit jedem RNE-Signal kann ein Zählimpuls gelesen werden. Dies ist nicht weiter verwunderlich, da der EPP pro RNE-Takt genau ein *Event-Packet* verarbeitet. Ein *Event-Packet* ist entweder gültig oder nicht. Ist es gültig, wird der Zählerstand des aktiven Zählers mit RNE inkrementiert.

Die Zählermodule CNT1 und CNT2 sind skalierbare Hexadezimalzähler. Die Anzahl der maximal benötigten Zähler-Stellen, also die maximal zu erwartende Zählrate, hängt von der *Clock*-Frequenz des EPP und vom RNE-Takt der *Pipeline* ab. Da nur alle 100 *Clock*-Impulse (1 RNE-Takt) ein *Event-Packet* die *Pipeline* verlässt, ist die maximale Zählrate in diesem Fall auf ein hundertstel der *Clock*-Zahl pro Sekunde festgelegt. Natürlich wird die *Clock*-Frequenz dem Detektordurchsatz angepasst, sodaß jedes erzeugte Ereignis auch ausgewertet werden kann. Bei einer *Clock*-Frequenz von 24 MHz beträgt die maximal erzeugbare Zählrate 240000 *Events* pro Sekunde. Für die beiden Zähler bedeutet das bei einer Torzeit von 2 Sekunden einen Zählbereich von dezimal 480000 bzw. hexadezimal 75300. Weil die Zähler intern als kaskadierte 1 stellige Hexadezimalzähler aufgebaut sind, werden in diesem Fall 5 Hex-Zähler benötigt, die Breite der Busse ist mit 20 Bit entsprechend der Bitdarstellung dieser Zahl dimensioniert. Mit einer Kapazität von 240000 cts/s ist der Zählbereich großzügig dimensioniert. Zum Vergleich: Die stärksten bisher gemessenen GRBs erreichten einen maximalen Fluss von 50 Photonen $cm^{-2}s^{-1}$, was einer Zählrate von ca 4000 cts/s entspricht.

Das Ergebnis jeden Zählers wird über das SWITCH-Modul in das AV-Modul geschrieben. Dies geschieht sobald das GL_SWITCH-Signal das GATE auf den anderen Zähler umgeschaltet hat. Der Stand des ruhenden Zählers wird innerhalb dieses Zeitraums in das AV-Modul geschrieben und zurückgesetzt. Die Steuer-Signale des *Event*-Zähler stammen aus dem GL-Modul. Dies gilt nicht nur für das GL_SWITCH-Signal, sondern auch für das WRITE- und RESET-Signal der beiden Zähler. Die Signale und ihr Zusammenwirken sind in Abbildung 4.28 dargestellt.

Die GATE-Logik:

Die GATE-Logik besteht aus zwei VHDL-Modulen, dem GATE-COUNTER- (GC) und dem GATE-LOGIC Modul (GL) (vgl. Abb. 4.27). Das GC-Modul erzeugt aus dem *Clock*-Signal das GATE_BASE-Signal, daß die Grundlage für die GATE-Umschaltung und die Torzeit des Zählers darstellt. Weil die tatsächliche *Clock*-Frequenz noch nicht feststeht, müßte der GATE-*Counter* so entworfen werden, daß eine Anpassung leicht möglich ist. Nach Festlegung der *Clock*-Frequenz ist deshalb lediglich die Zeitdauer des GA-

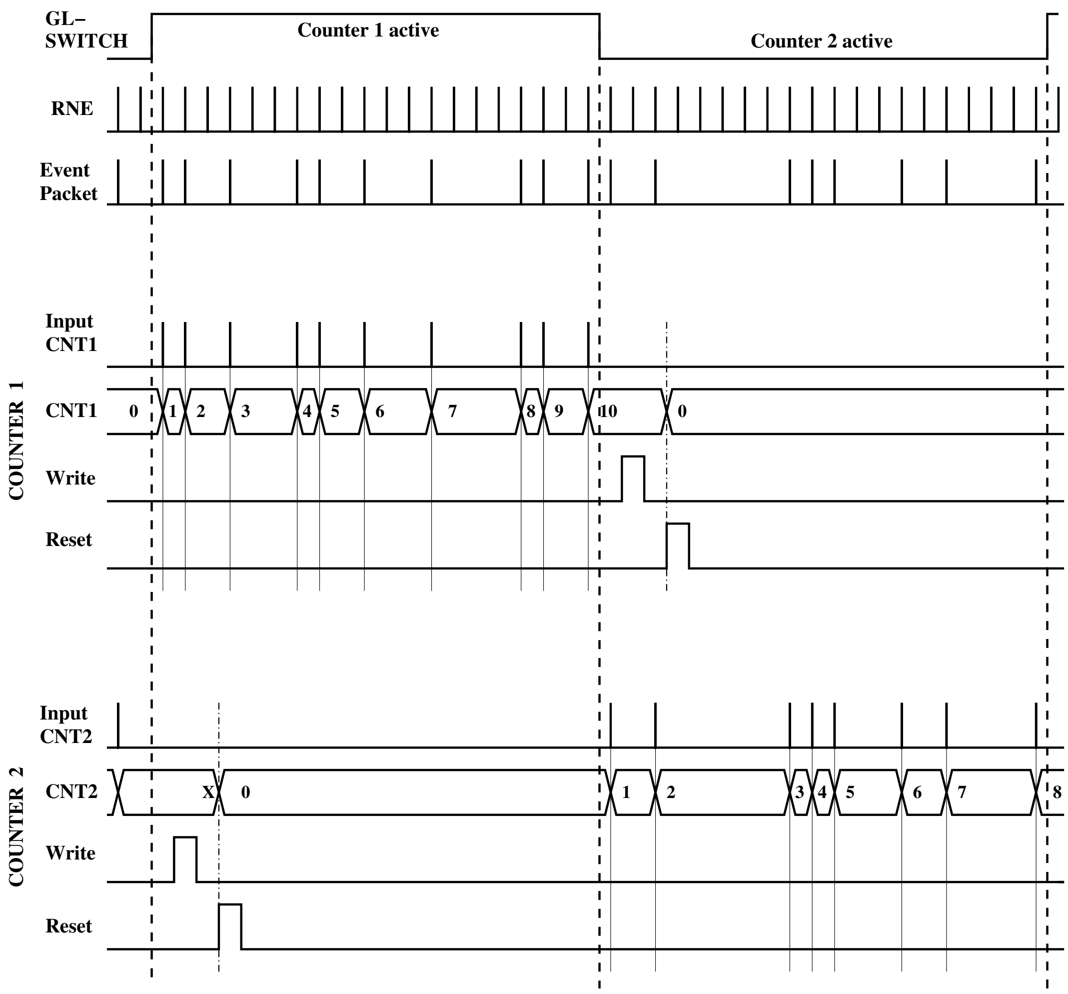


Abbildung 4.28: *Signale des Event-Zählers. Das* GL_SWITCH*-Signal steuert das GATE-Modul. Die Zählimpulse gelangen abwechselnd auf einen der beiden Zähler, während der andere Zähler ausgelesen und zurückgesetzt wird. Ist* GL_SWITCH *auf High, so ist Zähler 1 aktiv und Zähler 2 wird ausgewertet, ist* GL_SWITCH *auf Low, arbeitet Zähler 2 und Zähler 1 wird ausgewertet. Die Totzeit des Event-Zählers als Ganzes besteht also nur während der kurzen Umschaltzeit des GATE. Oben ist das* RNE*-Signal zu sehen. Darunter die Zählimpulse, die* RNE*-synchron anliegen und vom BS-Modul der FD-Unit ausgefiltert wurden. Nicht bei jedem* RNE*-Impuls liegt ein gültiges und von der Energie geeignetes Event-Packet an. Trotzdem muß der maximale Zählerstand in der Lage sein, auch bei jedem* RNE*-Impuls ein Event zu zählen.*

TE_BASE-Signals einzustellen, alle Steuersignale werden dann automatisch richtig erzeugt.

Der Hauptbestandteil des GATE-*Counters* ist ein Zähler, der mit jedem *Clock* inkrementiert wird, bis er seinen Maximalstand erreicht. Ist dieser erreicht, so wird der Zähler zurückgesetzt und das GATE_BASE-Signal kippt in den anderen Zustand. Der Maximalstand des GATE-*Counters* ist so eingestellt, daß das GATE_BASE-Signal die 1/8 Sekunden Basis des *Event*-Zählers definiert. Davon ausgehend sorgen Teiler für die Erzeugung der längeren Zeitbasen von 1/2, 1 und 2 Sekunden. Die Auswahl, welche Zeitbasis das GC-Modul als Torzeit verlässt (GL_SWITCH), kann von der Bodenstation über den Eingang GATE-SWITCH programmiert werden.

Das GATE-LOGIC-Modul verwendet das GATE_BASE-Signal auch zur Erzeugung der übrigen Steuer-Signale des *Event-Counters*. Diese sind mit der GATE_BASE synchron, sie steuern nacheinander das Auslesen und Rücksetzen des jeweilig inaktiven *Event*-Zählers. Das Signal, das die Module GATE und SWITCH des *Event*-Zählers umschaltet, heißt GL-SWITCH und seine Länge, die Torzeit, ist ein Multiplikator des GATE_BASE-Signals. Die Signale WRITE und RESET sind entsprechend mit dem GL-SWITCH gekoppelt.

Der *Event-Trigger*:

Jedesmal wenn das GL-SWITCH-Signal kippt, erhält der *Burst-Trigger*-Baustein die Zählrate des vorherigen Zeitintervalls. Der Zählerstand wird in das AV-Modul geschrieben, und der zugehörige Zähler gelöscht. Das AV-Modul berechnet aus den Zählraten der 64 vorangegangenen Zeitintervalle einen Zählraten Mittelwert. Dieser Zählraten-Mittelwert ist der Mittelwert des Röntgen-Hintergrundes innerhalb des eingestellten Energiebereichs. Er ist nicht zu verwechseln mit dem Mittelwert des Störhintergrundes, der auch Detektor spezifische Störungen enthält. Des Störhintergrund ist bereits in der FD-Unit weitgehend eliminiert worden. Zur Bestimmung des Mittelwerts verwendet das AV-Modul ein Register, in dem die letzten 64 Zählraten gepuffert werden. Der Puffer arbeitet nach dem Prinzip eines Schieberegisters. Wird ein neuer Wert gelesen, so verlässt der älteste Wert das Register und geht verloren. Das Ergebnis der Mittelwertberechnung sowie der aktuell ermittelte Zählratenwert gelangen gleichzeitig an den Eingang des CMP-Moduls. Das CMP-Modul vergleicht den Mittelwert mit der aktuellen Zählrate. Beträgt die Abweichung eine bestimmten Zahl, die sich aus dem 5σ Kriterium (siehe Kapitel 4.1.2) herleitet, so löst das CMP-Modul das BURST-TRIGGER-Signal aus. Dieser *Burst*-Alarm wird sofort vom LBR-*Interface* an die Bodenstation geleitet. Die 5 Sigma-Abweichung vom Mittelwert ist ein starkes Indiz dafür, daß das CMI einen *Gamma-Ray Burst* gefunden hat. Das 1 Bit breite BURST-TRIGGER-Signal ist der einzige Ausgang der BD-Unit.

In der Entwicklungs- und Testphase existieren auch in der BD-Unit zusätzliche Diagnoseausgänge, die für den späteren Betrieb stillgelegt werden. Dies sind Ausgänge für den Zählerstand der aktuellen Zählrate und für den Mittelwert des AV-Moduls. Mit ihnen kann die Funktion des *Event-Counters* und des *Burst-Trigger*-Bausteins überwacht werden (vergleiche Schaltplan der BD-Unit in Anhang A).

Im Zusammenhang mit der Mittelwertbildung sei noch auf zwei Eigenheiten des verwendeten Verfahrens hingewiesen:

Der Mittelwert erstreckt sich immer auf die zuletzt ermittelten 64 Zählraten. Im Falle eines GRB steigt die Zählrate gegenüber dem Mittelwert sprunghaft an. Dies bleibt jedoch auch auf den Mittelwert nicht ohne Wirkung. Je länger der GRB erhöhte Zählraten-Werte liefert, desto mehr steigt auch der Mittelwert an. Damit verschiebt sich jedoch das 5σ-Kriterium für den *Burst-Trigger* auf immer höhere Werte. Als Folge dieser Entwicklung erblindet der *Burst*-Detektor regelrecht für die Dauer des GRB. Sollte unmittelbar nach Beginn eines GRB ein zweiter innerhalb des Gesichtsfeldes des CMI auftreten, so wird dieser für den *Burst*-Detektor buchstäblich unsichtbar bleiben. Bei der erwarteten GRB-Häufigkeit von etwa einem GRB pro Tag, ist dies jedoch ein tollerierbares Risiko.

Ein zweiter Effekt betrifft das Einschalt-Verhalten des *Burst-Trigger*-Bausteins. Bei einem Neustart oder Reset des EPP wird auch der Zählraten-*Buffer* gelöscht. Der Mittelwert der Zählraten-Erfassung ist deshalb zu Beginn des Betriebs gleich Null, und jede aktuelle Zählrate größer Null wird unmittelbar das *Burst-Trigger*-Signal auslösen. Fehl-Alarme innerhalb der Einschaltphase des EPP sind also bisweilen konstruktionsbedingt. Erst nachdem der Zählraten-*Buffer* ausreichend gefüllt ist, erreicht die Zählrate einen stabilen Mittelwert. Eine Erweiterung des EPP zur Unterdrückung dieses Verhaltens ist zukünftigen Designänderungen vorbehalten.

4.3.6 RB-Unit (*Ring-Buffer-Unit*)

Da *Gamma-Ray Bursts* sehr dynamische Erscheinungen sind, die nur weniger Sekunden andauern, ist eine Aufzeichnung der Ereignisse für eine spätere wissenschaftliche Auswertung von herausragender Bedeutung. Würde die Aufzeichnung mit Auslösen des *Burst-Triggers* beginnen, so stünde nur ein unvollständiges Bild der Lichtkurven zur Verfügung. Von wissenschaftlichem Interesse ist insbesondere aber auch die Vorgeschichte eines GRBs. Aus diesem Grund besitzt der CMI einen Ringspeicher, der im Prinzip einer Überwachungskamera ständig alle *Event-Packets* aufzeichnet. Der Speicher ist so angelegt, daß nach dem voll laufen die ältesten Daten wieder überschrieben werden. Der *Ring-Buffer* des EPP besitzt eine Kapazität von 2 MByte. Bei einer Zählrate von 4000 cts/s und einer Paketlänge von 41 Bit entspricht dies einer Aufzeichnungsdauer von ca. 102 Sekunden.

Der Wert des *Ring-Buffers* besteht darin, daß nach Entdeckung eines *Bursts* der weitere Schreibzugriff gestoppt und der Inhalt des *Buffers* eingefroren werden kann. Auf diese Weise ist auch die Vorgeschichte der Lichtkurve bis zu mehrere hundert Sekunden vor dem *Burst-Triggers* enthalten. Legt man die bisherigen Erfahrungen mit GRB -Zählraten und -Zeitdauern zugrunde, so gestattet die Dimensionierung des *Buffers* die Aufzeichnung von Ereignissen bis zu einer Dauer von über fünf Minuten. Da GRBs im Normalfall eine Länge von zwei Minute nur selten überschreiten, ist daher die Aufzeichnung der Vorgeschichte mit einer ähnlichen Zeitdauer möglich. Nur im Falle außerordentlich starker und zusätzlich außerordentlich langer GRBs ist ein Überlaufen des *Ring-Buffers* zu erwarten.

Bei einer *Packet*-Breite von 41 Bit und einer Kapazität von 2 MByte finden 409200 *Event-Packets* innerhalb des *Ring-Buffers* Platz. Die Übertragung dieser Datenmenge dauert bei der bisher spezifizierten Übertragungsrate von 10 kBit/s fast eine halbe Stunde. Die Telemetrie stellt deshalb eine wesentliche Einschränkung beim Entwurf des EPP und des *Ring-Buffers* dar. Eine künftige Vergrößerung des *Buffers* erscheint deshalb kaum sinnvoll.

Der *Ring-Buffer* selbst ist nicht in den EPP integriert, sondern wird als externer Speicherbaustein realisiert. Innerhalb des EPP befindet sich jedoch der *Ring-Buffer-Controller*, der mit dem entsprechenden Ausgang der FD-Unit verbunden ist. Der *Controller* nimmt alle gültigen *Event-Packets* der FD-Unit in Empfang und schreibt sie kontinuierlich in den Speicher. Die Datenübergabe geschieht dabei RNE-Synchron. Synchron bedeutet allerdings nur, daß Daten mit anliegendem RNE-Signal transportiert werden können, aber nicht müssen. Die FD-Unit filtert die *Event-Packets* und deshalb wird nicht mit jedem RNE ein gültiges *Event-Packet* am Ausgang der FD-Unit liegen. Das *Packet* wird nur dann geschrieben, wenn eine separate Signal-Leitung die tatsächliche Gültigkeit signalisiert. Dies verhindert insbesondere, daß ein gültiges *Event-Packet* des vorherigen RNE noch einmal in den *Ring-Buffer* geschrieben wird. Die Signalleitung heißt RINGBCLK und ist in Abbildung 4.23 gezeigt. Sie signalisiert der RB-Unit, daß ein neues *Event-Packet* bereit liegt, der Zugriff erfolgt dann mit dem nächsten RNE-Impuls.

Das Datenformat der *Event-Packets* des *Ring-Buffers* ist in Abbildung 4.24 zu sehen. Die RB-Unit gehört zum Peripherie-Bereich des EPP. Sie konnte wie die anderen Peripherie-Units auch, im Rahmen dieser Arbeit nicht mehr realisiert werden.

4.3.7 LBR-*Interface* & *Telecommand Controller Unit*

Manche der EPP-Units bzw. Module benötigen externe Steuersignale. Diese Signale stammen aus der Detektorelektronik, der DPU, oder direkt von der Bodenstation. Sie steuern den Betrieb des EPP oder laden den LUT oder die Register mit neuen Werten. Eine Aufzählung aller bisher geplanten Signale ist in Tabelle 4.6 zusammengefasst. Die Signale des LBR-*Interface* sind außerdem auch in der Abbildung 4.11 des TOP-Designs zu sehen.

Signal	Aufgabe	Breite
Eingang:		
CLK	Clock Signal, f = 24.000000 MHz	1 Bit
RESET	Globales Reset Signal, bringt EPP in Einschaltzustand, löscht alle Register und Speicher innerhalb des EPP.	1 Bit
MODE	Schaltet den Betriebsmodus des EPP um.	2 Bit
GATE_TIMESET	Schaltet die Torzeit des *Event*-Zählers um, 1/8s, 1/2s, 1s, 2s.	2 Bit
LOCK	Sperrt den Ring-Buffer gegen schreiben.	1 Bit
RBREAD	Leitet das Auslesen der *Ring-Buffers* über das HBR-*Interface* EPP/DPU ein.	1 Bit
LOADLUT	Leitet den Ladevorgang des LUT ein.	1 Bit
SELECT	Selektiert das Register, das über den ADDR-BUS adressiert wird (LUT=1 oder RAM=0).	1 Bit
SIGMA	Legt die Ansprechschwelle des *Burst-Triggers* fest	3 Bit
ADDR-BUS	Legt eine Bus-Adresse an das LUT oder das RAM an.	10 Bit
DATA-BUS	Schreibt drei Datenworte in die unter ADDR-BUS angelegte Adresse.	8 Bit
Ausgang:		
TRIGGER	Gibt *Burst*-Alarm an die Bodenstation aus.	1 Bit

Tabelle 4.6: *Die Signale der LBR-Interface & Telecommand Unit*

Die Details sind derzeit noch nicht ausgearbeitet. Es ist jedoch sicher, daß der EPP neben den *High-Bit-Rate* Schnittstellen zum Transport der Messdaten, ein weiteres *Interface* benötigen wird. Mit diesem werden Steuerkommandos und Daten von der Bodenstation mit den EPP-Komponenten ausgetauscht. Das LBR-*Telecommand-Interface* wird dazu mit einem eigenen satellitenweiten Bus zur Kommandoübertragung verbunden sein (*Telecommand Bus*). Über diesen empfängt es Steuerkommandos und Daten, für die Register (RAM) und den *Look-Up-Table* (LUT). Details zu Kommunikationsprotokollen auf diesem Bus sind bisher nicht spezifiziert. Das LBR-*Telecommand-Interface* ist zu gegebenem Zeitpunkt zu entwerfen. Die Signal- und Datenleitungen, welche der EPP benötigt, werden im Folgenden näher besprochen.

CLK- und RESET-Signal:

Zwei sehr grundlegende Signale sind die CLK- und RESET-Signale. Der EPP arbeitet dynamisch, d.h. sämtliche Arbeitsschritte sind fest mit dem CLK-Signal synchronisiert. Fällt das CLK-Signal aus, so bricht der Betrieb des EPP augenblicklich zusammen.

Das CLK-Signal wird von der Detektorelektronik erzeugt. Zum derzeitigen Stand des EPP ist die Funktion des EPP noch vollkommen unkritisch auf die Wahl der *Clock*-Frequenz. Das bedeutet, daß alle funktionalen Komponenten mit jeder *Clock*-Frequenz fehlerfrei arbeiten werden. Natürlich ist die Wahl der Frequenz für die Verarbeitungsgeschwindigkeit ein entscheidender Parameter, sie wird zu gegebener Zeit auf den erforderlichen Datendurchsatz anzupassen sein. Es ist denkbar, daß der EPP mit der gleichen *Clock*-Frequenz wie der Detektor arbeiten wird, wahrscheinlich ist aber zumindest, daß der *Clock* von der Detektor-*Clock*-Frequenz abgeleitet und mit dieser synchronisiert wird. Für die Detektor-Elektronik ist bisher eine Frequenz von 24,0 MHz vorgesehen.

Das RESET-Signal versetzt den EPP-*Core* und alle angeschlossenen Register und Speichereinheiten zurück in den Ausgangszustand. Das bedeutet, daß alle Registerinhalte auf Null gesetzt werden, ebenfalls der Inhalt des *Look-Up-Tables* und des *Ring-Buffers*. Nach erfolgtem RESET sind deshalb die entsprechenden Register neu zu initialisieren, und das LUT neu zu laden. Der *Event-Buffer* des *Burst-Trigger*-Bausteins arbeitet sogar erst einwandfrei, nachdem die ersten 64 Zählraten zur Verfügung stehen. Das RESET-Signal kann von der Bodenstation im Falle einer Betriebsstörung oder zum vollständigen Löschen aller Speicherinhalte gegeben werden. Ein automatischer *Reset*, der an Bord des Satelliten aus entsprechendem Anlass ausgelöst wird, ist ebenfalls denkbar. Das RESET-Signal kann zu jedem beliebigen Zeitpunkt gegeben werden, und wird sofort wirksam. RESET ist *high*-aktiv, seine Länge muß mindestens die eines CLK-Signals betragen, längere Signale sind zulässig.

Das MODE-Signal:

Das MODE-Signal ist bisher nicht implementiert worden, könnte jedoch in Zukunft eine wichtige Rolle für den Betrieb des EPP spielen. Mit Hilfe des MODE-Signals kann der EPP in verschiedene Betriebszustände versetzt werden. Betriebszustände könnten neben dem bisher realisierten Arbeitsmodus z.B. ein Programmier-, Diagnose- und Transparent-Modus sein.

Der Programmiermodus könnte zum Schreiben der *Look-Up-Table* verwendet werden, während dessen kein sinnvoller Meßbetrieb möglich ist. Im Diagnose-Modus könnte eine Umschaltung der Telemetrie auf ungefilterte *Event-Packets* erfolgen, und so die Funktionalität des EPP geprüft werden. Ein Transparentmodus würde schließlich die Funktionalität des EPP ganz übergehen und die *Event-Packets* unkorrigiert und ungefiltert in die Telemetrie geben. Wenn ein Defekt innerhalb des EPP vorliegt, oder sich das Interesse auf Informationen richtet, die normalerweise innerhalb des EPP gefiltert werden, könnte ein solcher Modus von großem Nutzen sein. Für die Umschaltung zwischen den Betriebsmodi würde ein zusätzliches Signal mit einer Breite von 2 Bit ausreichen (4 Modi). Dies stellt nur einen unwesentlichen Mehraufwand für das Design dar, der Nutzen könnte aber beträchtlich sein.

Die *Trigger*-Schwelle:

Das Signal SIGMA ist bisher nicht implementiert worden. Es soll später die Ansprechschwelle des *Burst-Triggers* festlegen und eine größere Flexibilität bei der Suche nach GRBs ermöglichen. Bis zum derzeitigen Stand ist die Ansprechschwelle des *Burst-Trigger*-Bausteins auf 5 σ-Ereignisse festgelegt. Mit einem 3 Bit breiten SIGMA-Signal könnten 8 verschiedene Schwellwerte eingestellt werden, deren Werte sich z.B. im Bereich zwischen 1 σ und 10 σ bewegen könnten.

Torzeit Einstellung:

Die Einstellung der Torzeit des *Event*-Zählers geschieht mit Hilfe des LBR-*Telecommand-Interface*. Zu diesem Zweck existiert das Signal GATE_TIMESET mit einer Breite von 2 Bit.

Signal	Torzeit
00	1/8 s
01	1/2 s
10	1 s
11	2 s

Tabelle 4.7: *Das Signal* GATE_TIMESET *und die Torzeiten*

Die vier Zustände des Signals kann die Bodenstation mit Hilfe des LBR-*Interface* umschalten. Der Zusammenhang zwischen Torzeit und den Bitzuständen des GATE_TIMESET-Signals sind in Tabelle 4.7 dargestellt.

LOCK und RBREAD:

Im Falle, daß ein GRB gefunden und der *Burst-Trigger* ausgelöst wird, kann die Bodenstation (automatisch oder manuell) den *Ring-Buffer* vor weiteren Schreibzugriffen sperren. Dies ist sinnvoll, weil die Aufzeichnung sonst binnen weniger Minuten wieder überschrieben wird. Zum sperren des *Ring-Buffers* dient das LOCK-Signal. Geht LOCK auf *High*, so hält der *Ring-Buffer* augenblicklich an, und nimmt keine weiteren Daten mehr auf. Mit dem Signal RBREAD kann das Auslesen des *Buffers* eingeleitet werden. Die RB-Unit gibt dann Daten an das HBR-*Interface* EPP/DPU zur Telemetrierung weiter. Da *Event-Packets* der FD-Unit und der RB-Unit nicht gleichzeitig zum Boden übertragen werden können, gehen die *Event-Packets* der FD-Unit in dieser Zeit verloren. Die Datenaufnahme des HBR-*Interface* EPP/DPU ist zwischen FD- und RB-Unit umschaltbar. Die Umschaltung geschieht mit dem RBREAD-Signal.

Das HBR-*Interface*, die RB-Unit sowie die Signale LOCK und RBREAD sind zum gegenwärtigen Stand der Arbeit nicht implementiert.

Programmierung des LUT:

Die Signale LOADLUT, SELECT, ADDR-BUS, DATA-BUS dienen der Programmierung des *Look-Up-Tables* und der Register des EPP. Mit LOADLUT wird das LUTC-Modul in den Lesemodus versetzt. Der *Look-Up-Table* wird in Folge mit Daten geladen. Alle *Event-Packets* der betroffenen *Pipeline*-Takte werden während dieses Vorgangs ungültig, da in dieser Zeit kein Korrektur-Betrieb möglich ist. Die Adressen des LUT werden mit Hilfe des ADDR-BUS-Signals angelegt und die zugehörigen LUT-Daten über das DATA-BUS-Signal übertragen. Zu jeder LUT-Adresse gehören drei 8 Bit breite Datenworte, die Korrekturwerte A, B und C. Die Reihenfolge der Übertragung sorgt automatisch für die korrekte Einordnung dieser Werte in den *Look-Up-Table*. Mit dem SELECT-Signal wird die Adressierung zwischen LUT und RAM umgeschaltet. Das RAM ist eigentlich eher ein Register, und beinhaltet die vier *Threshold*-Werte für das THN- und das BS-Modul. Da nur ein Adressbus zur Verfügung steht und der Adressbereich vollständig für das LUT genutzt wird (1024-Einträge), ist SELECT als zusätzliches Signal erforderlich. Mit SELECT wird die Adressierung zwischen LUT und RAM umgeschaltet. So könnte z.B. bei SELECT=0 das Signal ADDR-BUS den LUT adressieren, während bei SELECT=1 die Speicherplätze des RAM angesprochen werden.

Zum derzeitigen Stand der Arbeit existiert nur das LOADLUT-Signal. Das LUT wird durch ein VHDL-*Array* simuliert. Geht LOADLUT für mindestens die Länge eines CLK-Signals auf *High*, so schaltet der EPP in den *Load*-Modus und das LUT wird geladen. Die Daten werden zur Zeit noch aus einem Festplatten-*File* des Test-Rechners in das LUT-*Array* geladen. Bei der fertigen Version des EPP werden diese Daten über den *Telecommand*-Bus des LBR-*Interface* übertragen.

4.3.8 HBR-*Interface-Unit* EPP/DPU

Das HBR-*Interface* EPP/DPU ist die Schnittstelle des EPP zur *Data-Processing-Unit* (DPU). Die DPU verarbeitet die EPP-*Event-Packets* zu Bildinformationen und berechnet daraus die Winkelposition der *Gamma*-Quellen am Himmel. Aus der Ortskoordinate des Detektors kann, mit Berücksichtigung der Aperturgeometrie und der Stellung des Satelliten, die Position der Quelle ermittelt werden. Die gewonnenen Daten werden an die Bodenstation übertragen und ggf. in das GCN eingespeist.

Das HBR-*Interface* EPP/DPU hat die Aufgabe mit der DPU zu kommunizieren, und die *Event-Packets* des EPP fehlerfrei zu übermitteln. Das Protokoll und die exakten Modalitäten dieser Datenübertragung sind bis zum derzeitigen Stand der Arbeit nicht spezifiziert.

4.4 Beschreibung des AC-Moduls

Am Beispiel des AC-Moduls der EC-Unit sollen einige Konstruktions-Prinzipien des EPP noch etwas näher besprochen werden. Das AC-Modul steht dabei in seiner Beschreibung exemplarisch für die anderen 25 VHDL-Module, die im Rahmen dieser Ausarbeitung nicht näher besprochen werden können.

4.4.1 Funktions Beschreibung

Das AC-Modul soll für jeden *Hit* des *Event-Packets* die Amplituden-Information korrigieren. Jeder *Hit* muß dazu einer der 1024 Detektor-Elektroden zugeordnet werden, wobei der *Look-Up-Table* die passenden Korrekturwerte (A,B und C) bereithält. Die Adressierung mit Hilfe des Ortsdatensegments (9 Bit + Position im *Event-Packet*) ist bereits in Kapitel 4.3.3 ausführlich besprochen worden. Als Eingangs-Signal für das AC-Modul dient der Rumpf (EPR)[13] des EPA-*Event-Packets*, der um den *Timestamp* reduziert wurde und nur noch die 6 *Hit*-Einträge beinhaltet.

Jeder *Hit*-Eintrag enthält nacheinander 14 Bit Amplituden-Information, das 9 Bit breite Ortsdatensegment und ein VD-Bit, das den Gültigkeits-Status des *Hits* kennzeichnen. Aufgabe des AC-Moduls ist, das 14 Bit breite Amplitudenwort zu korrigieren bevor die Daten in die weitere Auswertung (z.B Positions- oder Energie-Bestimmung) gelangen. Falls die Korrektur zu einem Überlauf oder einem Berechnungs-Fehler führt, ist der betreffende *Hit* ungültig. Um eine Unterscheidung der ungültigen *Event-Packets* bezüglich der Ursache zu ermöglichen, wird jedem *Hit*-Eintrag ein zusätzliches VD-Bit (VD-AC) angehängt. Das Ein- und Ausgangs-Format des AC-Moduls ist in Abbildung 4.29 gezeigt.

Der erste Schritt des AC-Moduls ist die Bestimmung der LUT-Adresse für jeden der 6 *Hit*-Einträge des EPR. Dazu wird das Ortsdatensegment eines jeden *Hits* gelesen, und als Adress-Information an das LUTC-Modul angelegt. Das LUTC-Modul adressiert mit diesem Signal den *Look-Up-Table* und liefert den Korrekturwert als 24 Bit breites Datenwort zurück. Die Signalübertragung geschieht dabei für alle *Hits* parallel, d.h. die Adress- und Daten-Busse zwischen AC- und LUTC-Modul existieren in sechsfacher Ausfertigung. Vergleiche hierzu Abbildung 4.30 und die ENTITY des AC-Moduls. Abbildung 4.30 ist ein Auszug aus dem Signal-Plan (EC-Unit) aus Anhang A, die ENTITY stammt aus dem VHDL-Listing AC aus Anhang B.

Nachdem die Korrektur-Werte am AC-Modul anliegen, werden sie in die drei 8 Bit breiten Worte A,B und C getrennt, und für jeden *Hit* mit der 14 Bit Amplituden-Information verrechnet. Dies geschieht ebenfalls parallel in sechs separaten VHDL-Prozessen. Die Korrektur geschieht nach Gleichung 4.1, Kapitel 4.1.1 in einem sequentiellen Prozess. Dabei werden nacheinander die Faktoren von Gleichung 4.1 berechnet und am Schluss miteinander addiert. Besonders zu beachten ist die korrekte Interpretation der 8 Bit breiten Korrekturwerte, vergleiche hierzu Kapitel 4.3.3 und Abbildung 4.18.

Am Ende des AC-Moduls werden die korrigierten Amplituden-Werte zusammen mit dem Ortsdatensegment und den beiden VD-Bits zu einem neuen EPR zusammengefügt. Das Ortsdatensegment wird zuvor noch auf die 7 Bit Darstellung reduziert. Das LUT ist die einzige Stufe des EPP, die eine 9 Bit breite Ortsadresse erfordert. Alle anderen Module kommen mit der auf 7 Bit reduzierten Ereignis-Adresse aus. Das AC-Modul reduziert deshalb am Ausgang das 9 Bit breite Ortsdatensegment auf 7 Bit, indem es den nicht mehr benötigten Teil der Segment-Adresse weglaßt (vgl. hierzu Kapitel 4.3.3).

Die EPA-Unit füllt alle nicht benötigten Zeilen des *Event-Packets* mit sogenannte Null-*Hits*. Die Null-*Hits* werden innerhalb des AC-Moduls als solche erkannt, und beeinflussen die Berechnungen nicht. Da-

[13]EPR: *Event-Packet*-Rumpf. Es ist korrekter an dieser Stelle von einem *Event-Packet*-Rumpf zu sprechen, um die Definition des *Event-Packets* wie in Kapitel 4.2 beschrieben nicht zu verletzen.

AC−Modul EPR−In Data Format (REG_IN)

Amplitude 14 Bit (rawdata)														Location 9 Bit									VD Det
143													130	129	128							121	120
119													106	105	104							97	96
95													82	81	80							73	72
71													58			55	54					49	48
47													34			31	30					25	24
23													10			7	6					1	0
MSB													LSB	MSB								LSB	

AC−Modul EPR−Out Data Format (REG_OUT)

Amplitude 14 Bit (corrected)														Location 7 Bit							VD Det	VD AC
137													124	123						117	116	115
114													101	100						94	93	92
91													78	77						71	70	69
68													55	54						48	47	46
45													32	31						25	24	23
22													9	8						2	1	0
MSB													LSB	MSB						LSB		

Abbildung 4.29: *Die beiden EPRs des AC-Moduls. Das Eingangs-EPR ist oben dargestellt. Es besitzt 9 Bit breite Ortsdatensegmente, die zur Adressierung des LUT notwendig sind. Die LUT-Adresse wird aus den 9 Bit des Ortsdatensegments plus der Stellung des Hit innerhalb des EPR gebildet. Dabei adressieren die ersten drei Hits (Anoden) LUT Adressen von 0 bis 511 und die letzten drei Hits (Kathoden) die LUT-Adressen von 512 bis 1023. Unten ist das Ausgangs-EPR dargestellt. Es enthält die korrigierten Amplituden-Daten und das auf 7 Bit reduzierte Ortsdatensegment. Für alle weiteren Aufgaben des EPP ist die 7 Bit breite Ereignis-Adresse ausreichend. Das Ausgangs-EPR enthält auch ein zusätzliche VD-Bit. Dieses kennzeichnet die Gültigkeit bezüglich Verarbeitungs-Fehlern innerhalb des AC-Moduls.*

zu wird ihr Inhalt (lauter Nullen außer den VD-Bits) unverändert in das Ausgangs-EPR des AC-Moduls kopiert. Die spezielle Erkennung der Null-*Hits* ist unbedingt erforderlich, da diese sonst mit dem LUT-Eintrag der Adresse Null korrigiert würden, und als Folge einen Amplitudenwert ungleich Null erhielten (z.B. durch *Offset*-Korrektur). Dieser würde in der weiteren Verarbeitung die Messwerte verfälschen. Die VD-Bits der Null-*Hits* müssen dagegen auf logisch 1 sein, da sonst das *Event-Packet* ungültig wird [14].

[14]Die Null-*Hits* sollen nur die nicht benötigten Zeilen des *Event-Packets* auffüllen, selbst aber weder das Signal, noch den Status des *Event-Packets* verändern! Dazu müssen alle Einträge des *Hits* logisch Null, und die VD-Bit logisch Eins sein.

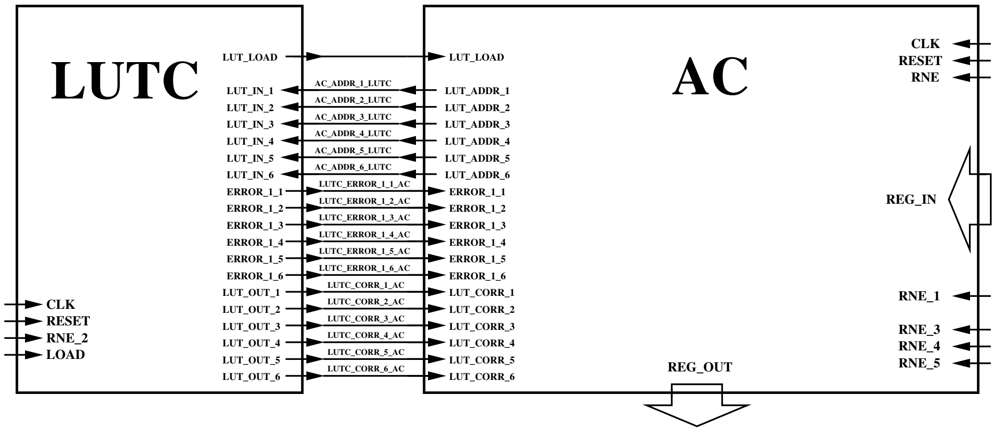


Abbildung 4.30: *AC- und LUTC-Modul. Das AC-Modul liest die LUT-Adressen (*LUT_ADDR*) aus dem Ortsdatensegment, und legt diese an den Eingang des LUTC-Modus. Das LUTC-Modul adressiert damit den* Look-Up-Table *und gibt die eingespeicherten Korrekturwerte als 24 Bit Datenwort (*LUT_CORR*) an das AC-Modul zurück. Vergleiche die Signale auch mit untenstehender* ENTITY *des AC-Moduls.*

<u>ENTITY des AC-Moduls:</u>

```
entity AC is
      port    (CLK         : in std_logic;                -- CLK signal.
               RESET       : in std_logic;                -- Reset.
               RNE         : in std_logic;                -- RNE signal.
               RNE_1       : in std_logic;                -- Sub RNEs
               RNE_3       : in std_logic;                --  signals.
               RNE_4       : in std_logic;                --
               RNE_5       : in std_logic;                --
               REG_IN      : in unsigned (143 downto 0);  -- AC input ERP.
               LUT_CORR1   : in unsigned ( 23 downto 0);  -- LUTC output
               LUT_CORR2   : in unsigned ( 23 downto 0);  --  Hit dependent
               LUT_CORR3   : in unsigned ( 23 downto 0);  --  corrections.
               LUT_CORR4   : in unsigned ( 23 downto 0);  --
               LUT_CORR5   : in unsigned ( 23 downto 0);  --
               LUT_CORR6   : in unsigned ( 23 downto 0);  --
               ERROR_1_1   : in std_logic;                -- LUTC output
               ERROR_1_2   : in std_logic;                --  Hit dependent
               ERROR_1_3   : in std_logic;                --  LUT errors.
               ERROR_1_4   : in std_logic;                --
               ERROR_1_5   : in std_logic;                --
               ERROR_1_6   : in std_logic;                --
               LUT_LOAD    : in std_logic;                -- LOAD Signal.
               LUT_ADDR_1  : out unsigned (8 downto 0);   -- AC output
               LUT_ADDR_2  : out unsigned (8 downto 0);   --  Hit dependent
               LUT_ADDR_3  : out unsigned (8 downto 0);   --  LUT Addresses.
               LUT_ADDR_4  : out unsigned (8 downto 0);   --
               LUT_ADDR_5  : out unsigned (8 downto 0);   --
               LUT_ADDR_6  : out unsigned (8 downto 0);   --
               REG_OUT     : out unsigned (137 downto 0); -- AC output ERP.
              );
 end AC;
```

4.4.2 RNE und Sub-RNE

Der interne Ablauf des AC-Moduls, wie auch der gesamten EC-Unit, ist RNE-gesteuert. RNE bedeutet *Read-Next-Event* und definiert den *Pipeline*-Takt, mit dem der EPP die *Event-Packets* von einer Unit zur nächsten schickt. Das RNE wird innerhalb der EC-Unit auch zur Ablaufsteuerung verwendet. Dazu erzeugt das EC_IN-Modul sogenannten Sub-RNE-Signale. Der Zeitraum zwischen zwei RNE-Signalen wird mit Hilfe der Sub-RNEs in 10 gleichlange Zeitintervalle geteilt. Diese haben jeweils eine Länge von 10 CLK-Impulsen. Die weiter oben beschriebene Funktion des AC-Moduls wird vollständig durch Verwendung der ersten 5 Sub-RNE-Signale (RNE_1 bis RNE_5, siehe ENTITY) gesteuert. Mit anliegen des RNE-Signals wird zunächst das EPA-*Event-Packet* durch das EC_IN-Modul der EC-Unit gelesen. Während der nächsten 10 CLK-Impulse trennt dieses die *Timestamp*-Information ab. Mit anliegen des ersten Sub-RNE (RNE_1) greift das AC-Modul auf das EPR zu. Das AC-Modul ließt die LUT-Adressen in den Ortsdatensegmenten und legt diese an das LUTC-Modul an. Das LUTC-Modul nimmt die Adressen mit Sub-RNE_2 entgegen und legt die Korrekturdaten auf den Daten-Bus. Diese gelangen mit Sub-RNE_3 in das AC-Modul zurück und werden in die Korrekturwerte A,B und C geteilt. Mit Sub-RNE_4 findet die Korrektur statt, und mit Sub-RNE_5 wird das neue EPR zusammengebaut. Bei anliegen des Sub-RNE_6 ist der gesamte Korrekturvorgang des AC-Moduls abgeschlossen und andere Module können auf das EPR am Ausgang zugreifen.

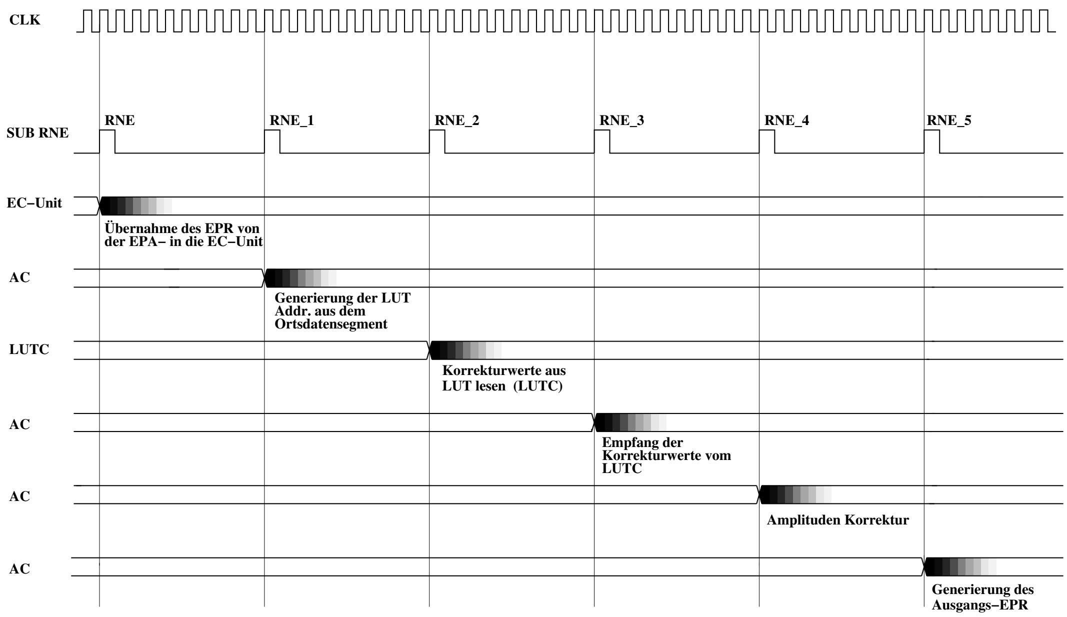


Abbildung 4.31: *Ablaufsteuerung des AC-Moduls. Die gezeigte Sequenz ist nur ein Teil der internen Steuerung der EC-Unit. Jeder neue Berechnungs-Schritt des AC-Moduls wird mit einem Sub-*RNE *eingeleitet (*RNE_1 *bis* RNE_5*). Für die Ausführung stehen jeweils 10* CLK*-Einheiten zur Verfügung.*

Zwischen jedem Sub-RNE- Schritt liegt der Zeitraum von 10 *Clock*-Impulsen. Dieser Zeitraum steht für jeden der Bearbeitungs-Schritte bzw. für die Adressierung des LUT zur Verfügung. Eine Ablaufsteuerung dieser Art beugt kritischen Laufzeit-Problemen vor. Sollte aus irgend einem Grund die Zeitspanne von 10 *Clock*-Impulsen nicht ausreichen, kann der Sub-RNE-Zähler des EC_IN-Moduls entsprechend modifiziert werden, indem z.B. die äquidistante Aufteilung des RNE-Zeitraums aufgegeben wird. Sollte dies immer noch nicht reichen, ist entweder der *Pipeline*-Takt (bisher 100 *Clocks*), oder besser das ganze Design zu ändern. Bei einem rein funktionalen Test mit dem VHDL-Simulator spielen Laufzeitprobleme keine Rolle. Erst bei einer Laufzeit-Simulation oder letztendlich bei der Hardware-Synthese wird sich zeigen, ob die bisherige Dimensionierungen ausreichend sind. Letztenendes wird das Problem vor allem von der später verwendeten *Clock*-Frequenz abhängig sein.

Abbildung 4.31 zeigt die Ablaufsteuerung des AC-Moduls. Das AC-Modul hat etwa den halben Zeitraum eines *Pipeline*-Taktes für die Amplituden-Korrektur zur Verfügung. Eine ähnliche Ablaufsteuerung wird auch innerhalb der FD-Unit verwendet. Ein Beispiel für die hier beschriebene Ablaufsteuerung des AC-Moduls findet sich auch in Abbildung 5.3 Kapitel 5.1.

4.4.3 *Load*-Modus

Für das AC-Modul von großer Bedeutung ist die korrekte Funktion des *Look-Up-Tables*. Dazu muß dieser mit Korrektur-Werten geladen werden. Auch im späteren Betrieb soll der Austausch dieser Korrekturwerte jederzeit möglich sein. Der EPP besitzt zu diesem Zweck das LOAD-Signal, daß den Ladevorgang einleitet, wenn LOAD für mindestens die Länge eines CLK-Signals auf *High* geht. Das Lesen der Korrekturwerte geschieht während der Testphase des EPP von Festplatte. Dort liegt eine Datei, die alle Korrekturwerte als Dezimalzahlen enthält. Später werden die Korrektur-Werte über den *Telecommand-Bus* des EPP gelesen.

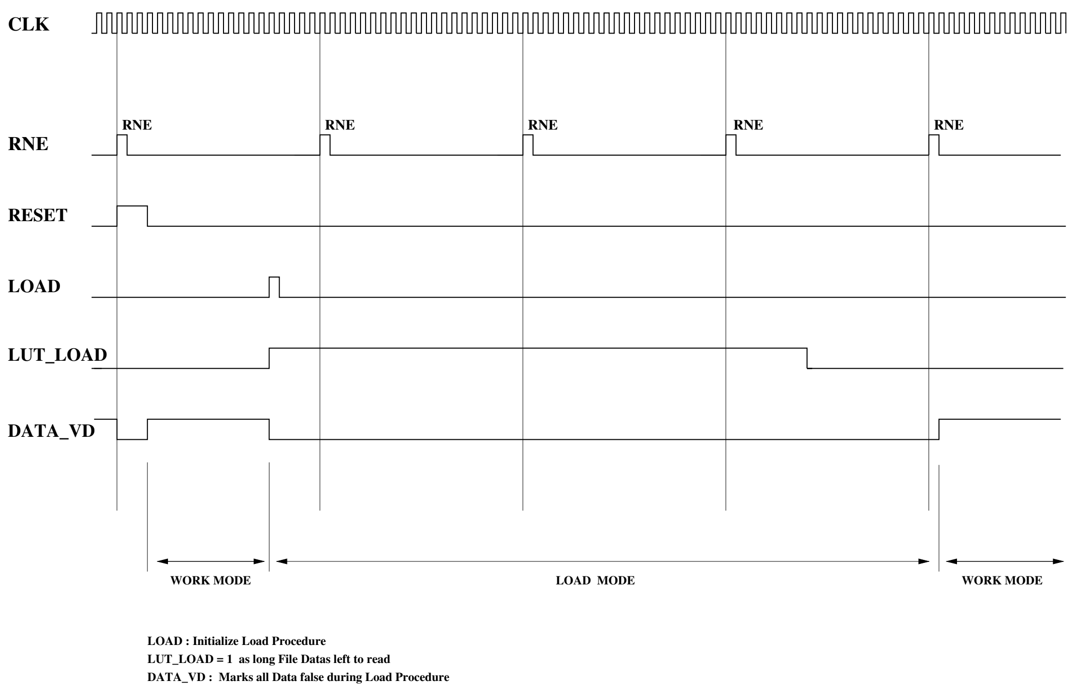


Abbildung 4.32: *Load-Vorgang des Look-Up-Tables. Sobald das* LOAD*-Signal mindestens ein* CLK *auf High ist, wird das Laden des LUT eingeleitet (*LUT_LOAD*). Das Signal* DATA_VD *(Data Valid) bleibt solange auf Null bis der Ladevorgang beendet ist und der nächste Pipeline-Takt beginnt. Solange* DATA_VD *Low ist, sind alle Event-Packets innerhalb der Pipeline ungültig.*

Während der *Load*-Vorgangs läuft ist kein Korrekturbetrieb möglich. Der EPP wird während dessen nicht angehalten, sondern erkennt den Vorgang und markiert alle *Event-Packets* als ungültig, die bis zum Ende des Ladezyklus die *Pipeline* durchlaufen. Dazu wird das VD-AC-Bit aller *Hits* solange auf Null gesetzt, bis der Ladevorgang zuende ist und ein neuer *Pipeline*-Takt beginnt. Die betroffenen *Event-Packets* gehen nicht in die Datenauswertung ein, sondern werden in der FD-Unit verworfen.

Der Ladevorgang ist in Abbildung 4.32 dargestellt. Für den vollständigen Vorgang sind 1024 LUT-Werte zu laden. Bei einer Lesegeschwindigkeit von einem Wert pro CLK benötigt das Laden genau 1024 CLK-Impulse, was etwas mehr als 10 *Pipeline*-Zyklen bei (100 *Clock* pro RNE) entspricht. Die ersten 11 *Event-Packets* nach Anlegen des LOAD-Signals (z.B. bei einem Testdurchlauf) werden aus diesem Grund in jedem Fall immer ungültig.

4.4.4 Fehlererkennung und *Error-Flags*

Innerhalb des AC-Moduls existieren zahlreiche Ursachen für das Auftreten von Verarbeitungs-Fehlern. Dies hängt damit zusammen, daß hier die Daten mit multiplikativen und additiven Operatoren verändert werden. Die Daten liegen dabei in festgelegten Datenformaten und Wertebereichen vor. Eine Verdopplung oder Negation kann sofort zu einem Überlauf oder unerlaubten Daten-Format führen, und einen Verarbeitungs-Fehler auslösen.

Natürlich darf ein Verarbeitungsfehler auf keinen Fall zu einer Betriebsstörung des EPP führen. Aus diesem Grund sind alle möglichen Fehlerquellen während der Testphase untersucht worden. Auch die Verarbeitung vollkommen unmöglicher Werte oder Fehleingaben innerhalb des *Look-Up-Tables* wurden in diesem Zusammenhang untersucht.

Ein Verarbeitungs-Fehler wird innerhalb des AC-Moduls erkannt und führt zu einem ungültigen *Hit*, und damit letztendlich auch zu einem ungültigen *Event-Packet*. Das AC-Modul schreibt dazu das VD-AC-Bit des betroffenen *Hit*-Eintrags und setzt dieses auf Null. Besitzt auch nur ein einziger *Hit* eines EPR ein ungültiges VD-Bit (VD-Det oder VD-AC) wird das ganze *Event-Packet* ungültig. Die Trennung von gültigen und ungültigen *Event-Packets* geschieht später in der FD-Unit.

Schon zu Beginn der Entwicklung des EPP wurde über die Einführung eines EPP-weiten *Error-Codes* nachgedacht. Der *Error-Code* ist zwar im *Event-Packet* Format vorgesehen, konnte aber bisher nicht realisiert werden. Mit Hilfe des *Error-Codes* könnten Verarbeitungs-Fehler auch noch während des Betriebs, bis zur genauen Ursache verfolgt werden. Die bisherige Handhabung mit den VD-Bits lässt diesbezüglich nur die Unterscheidung bis auf die Unit-Ebene zu.

4.4.5 Design-Prinzipien

Den VHDL-Designs des EPP liegen ein paar grundlegende Design-Prinzipien zugrunde. Diese sollen an dieser Stelle noch kurz besprochen werden:

- Prozesse werden wenn möglich CLK-synchron gestartet.

  Der Prozessor soll soweit möglich in seinen Komponenten synchron zum *Clock*-Signal betrieben werden. Dazu wird vorrangig mit der Bedingung

  ```
  if rising_edge (CLK) then
  ```

  innerhalb von Prozessen gearbeitet. Die Bedingung stellt sicher, daß der Prozess erst mit der ansteigenden Flanke des *Clock*-Signals gestartet wird. Damit wird die Befehlsverarbeitung mit dem *Clock*-Signal synchronisiert. Die Flanken-*getriggerte* Abfrage ist zudem ein wirksamer Schutz gegen *Spikes*. Die Signale der Prozess-Bedingungen werden nur mit Anstieg des *Clock*-Signals abgefragt, zwischenzeitliche Störungen auf diesem Signal bleiben ohne Auswirkungen.

- Alle Variable, Signale und *Port*-Ausgänge werden mit dem RESET-Signal in einen eindeutigen und definierten Zustand versetzt. Der Reset-Vorgang wird an Beispiel 1 verdeutlicht:

  Beispiel 1:

```
 1  architecture inside of AC is
 2  ...
 3  signal ERG_1   :  unsigned (13 downto 0);
 4  ...
 5  begin
 6      ...
 7      ac_hit1:
 8      process (CLK)
 9            variable    Q_ERG_1      : unsigned (27 downto 0);
10            variable    VAL_1_A      : unsigned ( 7 downto 0);
11            ...
12            variable    OVERFLOW_Q   : std_logic;
13            ...
14      begin
15            if rising_edge (CLK) then
16                   if RESET = '1' then
17                          Q_ERG_1    :=  (others => '0');
18                          VAL_1_A    :=  (others => '0');
19                          ...
20                          OVERFLOW_Q :=  '0';
21                          ...
22                          ERG_1      <=  (others => '0');
23                          ...
24                   else
25                          ...
26                          ...
27                          ...
28                   end if;
29            end if;
30      end process;
31      ...
32  end inside;
```

Beispiel 1 ist ein Ausschnitt der ARCHITECTURE des AC-Moduls. Im ARCHITECTURE-Kopf (Zeilen 2 bis 4) werden die Signale definiert. Diese sind in allen Prozessen des Designs bekannt, und verbinden die Ein- und Ausgänge verschiedener Prozesse miteinander. Im Anweisungsteil der ARCHITECTURE (Zeilen 5 bis 32) befinden sich die Prozess-Aufrufe. Einer der Prozesse (ac_hit1), ist für dieses Beispiel herausgegriffen. Er enthält die Anweisungen zur Amplituden-Korrektur des ersten EPR-*Hits*. Der Prozess-Aufruf geschieht mit Hilfe der *Sensity List* innerhalb der Kopfzeile (Zeile 8). Bei jeder Änderung der dort eingetragenen Signale (CLK) wird der Prozess gestartet. Der Prozess-Kopf (Zeilen 9 bis 13) enthält Variablen-Deklarationen. Variablen sind nur innerhalb des jeweiligen Prozesses bekannt, und dienen der Prozess-Bearbeitung. Prozesse selbst sind nebenläufige Anweisungen, d.h alle Prozesse einer ARCHITECTURE können gleichzeitig aktiv sein. Innerhalb der Prozesse werden die Befehle jedoch sequentiell, also Zeile für Zeile, verarbeitet. Der Anweisungsteil des Prozesses (Zeile 14 bis 30) enthält die funktionale Beschreibung. Die erste Zeile dieses Teils (Zeile 15) stellt sicher, daß die anschließende Befehlsliste immer nur mit der steigenden Flanke des CLK-Signals ausgeführt wird. In der folgenden Zeile (Zeile 16) wird der Status des RESET-Signals abgefragt. Ist RESET aktiv, werden sämtliche Variable, Signale und Port-Ausgänge, des Prozesses auf Null gesetzt. Dieses Vorgehen ist erforderlich, da nicht eindeutig definierte Variable oder Signale eine Gefahr für die Stabilität des Designs darstellen können. Ist RESET auf Null, wird die '*else*'-Anweisung (Zeile 24) aktiv, und führt die nachfolgenden Kommandos (Zeilen 25 bis 27) aus. Konstrukte mit '*if then else*' garantieren, daß immer mindestens ein Teil der Verzweigung aktiv ist, es sind keine Zustands-Lücken möglich, bei denen sich der Prozess weder in dem einen, noch in dem anderen Zustand befindet. Solche *if then else* -Konstrukte und der RESET-Mechanismus sind in allen Designs des EPP verwendet worden.

- Es werden keine *Tri-State* Ausgänge verwendet.

  Tri-State-Ausgänge sind Ausgänge, die sowohl die Zustände '0' und '1', als auch den Zustand 'Offen' kennen. Der 'Offen'-Zustand ist erforderlich, wenn ein Eingangssignal mehr als einen Treiber besitzt. Nur ein Treiber kann den Zustand des Signal steuern, alle anderen Treiber müssen 'Offen' sein um Kurzschlüsse zu verhindern. Ein Design mit mehreren Bus-Treibern birgt viele Risiken, und sollte wenn möglich vermieden werden.

- *Port*-Signale sind entweder Eingänge oder Ausgänge. Es werden keine Typen 'InOut' verwendet.

  In VHDL ist die Verwendung von *Port*-Signalen möglich, die gleichzeitig als Ein- und Ausgang fungieren (InOut-Typen). Die Verwendung solcher *Ports* wurde vermieden und Probleme dieser Art stattdessen durch Design-Interne Signale gelöst.

- Vermeidung von unbeabsichtigten Mehrfachaufrufen durch unsaubere *Clock*-Signale.

  Um möglichen Betriebsstörungen schon in der Design-Phase vorzubeugen, kann der Prozess-Aufruf gegen Signal-Überschwinger geschützt werden. Überschwinger entstehen, wenn digitale Signale sehr hoher Frequenz (MHz-Bereich) verarbeitet werden. Die Schaltzeiten von Transistorverstärkern sind nicht beliebig klein, zusätzlich führen kapazitive und induktive Lasten zu Auf- und Umlade-Effekten. Die Signalflanken von digitalen Signalen zeigen deshalb mit steigender Frequenz zunehmend das Verhalten von Einschwingvorgängen. Das Signal benötigt eine Weile um den jeweiligen Wert (0 oder 1) stabil zu erreichen.

  Das *Clock*-Signal ist besonders anfällig gegen Überschwinger, da es meistens die höchste Frequenz darstellt. Zudem arbeiten meist viele Stufen einer Schaltung *Clock*-Synchron, stellen also in gewissem Sinne ohmsche Verbraucher des *Clock*-Signals dar[15], was die Flankensteilheit reduziert und Einschwing-Effekte verstärkt.

  Das Auf- und Abschwingen des *Clock*-Signals stellt beim Aufruf von Prozessen via *Sensity-List* ein besonders Problem dar. Ab einer gewissen Amplitude der Signal-Verzerrung werden die Prozesse beim Einschwingen des CLK-Signals unmittelbar mehrmals hintereinander gestartet. Dies kann zu ernsthaften Betriebsstörungen führen. Um die Schaltung gegen Fehlinterpretationen zu sichern, kann die *if*-Bedingung bei '*if then else*'-Konstrukten mit einer zusätzlichen Hilfvariable überprüft werden. Das mehrfache starten des Prozesses durch CLK-Überschwinger führt dann trotzdem nur zu einem einmaligen Start der Befehlsausführung.

[15] In der Regel stellt ab einer gewissen Frequenz jedes reale Bauteil nicht nur eine rein Ohmsche, sondern auch eine Kapazitive und Induktive Last dar. Selbst die Signalleitungen besitzen bei hohen Frequenzen diese Eigenschaft.

Beispiel 2:

```
1    ...
2    process (CLK)
3           ...
4           last_RNE   := std_logic;
5           ...
6    begin
7           if rising_edge (CLK) then
8                  if RESET = '1' then
9                       ...
10                      last_RNE  := '0';
11                      ...
12                 else
13                      if RNE = '1' and last_RNE = '0' then
14                            ...
15                            ...
16                            ...
17                      end if;
18                      last_RNE := RNE;
19                  end if;
20          end if;
21   end process;
22   ...
```

Beispiel 2 zeigt den beschriebenen Konstrukt in Zeile 13 und Zeile 18. Der Befehlsteil (Zeilen 14 bis 16) wird erst gestartet, wenn der Prozess anläuft und das Signal RNE auf '1' liegt. Um ein mehrfaches Starten durch Überschwinger des CLK-Signals zu verhindern, wird die Prozess-Variable last_RNE definiert und als zusätzlichen Bedingung in die '*if*'-Abfrage (Zeile 13) eingebaut. Nach dem ersten Start des Prozesses nimmt last_RNE (Zeile 18) den Wert von RNE an. Jeder weitere Aufrufen via *Sensity-List*, der unmittelbar auf den ersten folgt, findet nun in der '*if*'-Abfrage die zweite Bedingung unerfüllt, da RNE = last_RNE. Der Prozess wird erst wieder 'scharf' wenn sich das RNE-Signal das nächste mal geändert hat.

Kapitel 5

Entwicklung und Test des EPP

Die Entwicklung des EPP begann im Frühjahr 2000 mit der vorliegenden Diplomarbeit. Sie geht auf ein *Proposal* zurück, daß CASS @ *University of California San Diego* und IAAT @ Universität Tübingen, 1999 gemeinsam an die NASA gerichtet haben. Das *Proposal* schlug die Entwicklung und den Bau eines *Coded-Mask-Imagers* als *Burst-Monitor* für die von der NASA geplante GLAST-Mission vor. Unter anderem ist die Entwicklung des EPP darin festgelegt worden. Das *Proposal* enthielt nur wenig technische Spezifikationen bzgl. des EPP, sodaß für die Realisierung der Funktionen vollkommen freie Hand blieb. Aufgrund ständiger Probleme mit *Schematic*-Designwerkzeugen bei früheren Projekten entschied man am IAAT, das EPP-Projekt vollständig in VHDL zu realisieren. Damit waren die Rahmenbedingungen vorgegeben und die Arbeit konnte mit dem Entwurf der ersten Design-Module beginnen. Das Augenmerk lag zunächst auf den operativen Aufgaben des EPP, wie z.B. der Fehlerkorrektur oder den *Event*-Zählern. Peripherie- und Schnittstellen-Designs wurden in Ermangelung ausreichender Spezifikationen vorerst zurückgestellt. Bis Herbst 2000 konnte eine erste VHDL-Version des EPP-*Core* mit 3 Design-Units fertig gestellt werden. Diese wurde seitdem immer wieder getestet, und um zahlreiche Details verbessert. Trotzdem sind bis zu einer endgültigen Fertigstellung noch unzählige Hürden zu überwinden. Fragen wie beispielsweise die Synthetisierbarkeit oder das *Timing*-Verhalten des VHDL-Designs sind noch vollkommen ungelöst. Auch die Schnittstellen-Designs und eine realistische Hardware-Beschreibung des *Look-Up-Tables* sind noch zu erarbeiten. Schließlich ist für die Fortsetzung der VHDL-Modellierung eine leistungsstarke und automatisierte Testumgebung zu entwickeln.

5.1 Test Methodologie

Zu jeder Entwicklungsphase des EPP-Projekts sind zahlreiche Tests erforderlich. Diese sollen nicht nur die einwandfreie Funktion des EPP gewährleisten, sondern sind auch für den Entwicklungvorgang selbst von grundlegender Bedeutung. Die Wechselwirkung von Innovation und Test stellt dabei einen wirkungsvollen Evolutions-Mechanismus für das Weiterkommen des Projektes dar. Dabei werden ständig neue Ansätze erdacht, die ihre Brauchbarkeit dann in den anschließenden Testphasen unter Beweis stellen. Auf diese Weise verbessert sich das Design solange, bis es die gestellten Anforderungen effizient und zuverlässig erfüllt.

Die Tests finden auf verschiedenen Abstraktionsebenen statt. So muß ein Elektronik-Projekt beispielsweise auf der Bauteilebene genauso getestet werden, wie auf der Logik- und der Funktions-Ebene. Für das EPP-Projekt, das sich bisher noch im Design-Stadium befindet, sind folgende Tests definierbar:

- Syntax-Test:
  Syntax-Tests beziehen sich hier auf die syntaktische Korrektheit der VHDL-Beschreibung. Neben manuellen Tests, stellt vor allem der eingesetzte VHDL-*Compiler* das wirksamste Werkzeug zum

Syntax-Test dar. In der Regel verweigert dieser die Übersetzung syntaktisch fehlerhafter Designs, und gibt detaillierte Anweisungen, an welcher Stelle des *Codes* der oder die Fehler aufgetreten sind.

- Logik-Test:
  Die logische Korrektheit eines VHDL-Designs ist vom *Compiler* nicht entdeckbar. Hierzu bieten die meisten VHDL-Systeme Simulatoren an. Ein Simulator stellt die Signale eines Designs in der Regel graphisch dar, und verfügt über eingebaute Signal-Generatoren zur Erzeugung von Signal-Stimuli. Die Stimuli können manuell an die Signal-Eingänge des zu testenden Designs angelegt werden. Ob ein Design logisch korrekt ist, obliegt dabei dem Urteil des Entwicklers. Der Simulator ist nur ein Werkzeug zur Visualisierung logischer Signal-Verknüpfungen, ein Urteil, ob ein Design logisch korrekt ist, kann dieser selbst nicht abgeben.

- Funktions-Test: Ein Funktions-Test stellt die nächst höhere Test-Ebene dar. Hier wird zum ersten mal überprüft, ob das Design seine Aufgaben bezüglich der Funktion auch wirklich erfüllt. In der Regel betrifft der Funktions-Test vor allem VHDL-Designs in funktionaler Modellierung. Die Korrektheit struktural modellierter Designs wird bereit innerhalb des Logik-Test verifiziert. Ob ein Design seine Funktion erfüllt, kann mit dem Simulator überprüft werden. Dazu wählt der Entwickler die Stimuli geschickt, und vergleicht die Reaktion des Designs, also die Ausgangssignale, mit den Erwartungen. In speziellen Fällen kann auch eine sogenannte *Testbench* konstruiert werden. Diese simuliert die spätere Umgebung des Designs und erzeugt genau die erforderlichen Stimuli. Zum Funktions-Test gehört nicht nur die Untersuchung des Entwurfs auf sein Verhalten bei Verarbeitung korrekter, sondern auch bei Verarbeitung fehlerhafter Eingangsdaten. Für jeden Entwurf ist sicherzustellen, daß korrupte Eingangsdaten das Design nicht in undefinierte Zustände versetzen, und damit einen Absturz der Signalverarbeitung auslösen können.

- Synthetisierbarkeit-Test:
  Nicht jedes VHDL-Design das syntaktisch und logisch in Ordnung ist, kann für den Einsatz in einem FPGA[1] synthetisiert werden. Die tatsächliche Synthetisierbarkeit hängt von den Eigenheiten elektronischer Bauteile ab. Einfache mathematische Funktionen innerhalb des VHDL-Designs können beispielsweise zu enormem Aufwand bei der Synthetisierung in reale Gatter führen. Die Anzahl der verfügbaren Gatter innerhalb eines FPGA ist vom Typ abhängig, aber in jedem Fall begrenzt. Ob ein VHDL-Design für einen PGA synthetisierbar ist oder nicht, wird erst bei der Synthese vom Synthesewerkzeug des FPGA festgestellt. VHDL ist eine universell einsetzbare Beschreibungssprache, und nicht speziell zur Entwicklung von FPGA-Chips entworfen worden. Es ist deshalb Aufgabe der Hersteller von FPGAs, die für ihre Chips synthetisierbaren VHDL-Konstrukte zu spezifizieren. In vielen Fällen bietet der Hersteller von FPGA-Chips ebenfalls ein für seine Chips geeignetes Synthesewerkzeuge an, in manchen Fällen sogar die vollständige VHDL-Entwicklungsumgebung. Die Synthetisierbarkeit eines VHDL-Designs für den FPGA eines speziellen Herstellers ist während der Entwicklung in regelmäßigen Abständen zu verifizieren. Andernfalls kann es vorkommen, daß ein fertiges Design am Ende an der Synthese scheitert, und aufwendige Änderungen daran notwendig werden. Dies kann die Fertigstellung des gesamten Projektes empfindlich verzögern.

- Laufzeit-Test:
  Nach erfolgreichen Funktions-Tests ist das VHDL-Design auf Laufzeit zu testen. Aktionen und Reaktionen, die sich im Stile einer Programmiersprache realisieren lassen und quasi ohne Zeit-Verzögerungen ablaufen, können in der Hardware-Realisierung erhebliche Probleme verursachen. Reale Bauteile besitzen keine unendliche Reaktionsgeschwindigkeit, in der Regel sind auf physikalischer Ebene Ladungen zu bewegen. Zahlreiche Effekte innerhalb realer Bauteile verzögern deren Reaktion- und Antwort-Zeiten. Eine Hardware-Beschreibung muß auf diesen Umstand Rücksicht nehmen. VHDL besitzt zu diesem Zweck Strukturen mit denen die Verzögerungszeiten für Signale festgelegt werden können [Lehmann et al. 1994]. In der Regel sind Signalverzögerungen auch von der Technologie abhängig. So kann eine Schaltungsrealisierung in HCMOS[2] andere Schaltzeiten

[1] FPGA:*Field Programable Gate Array*
[2] HCMOS: *High Speed* CMOS-Technologie (MOS: *Metall-Oxid-Silicon*)

erfordern als eine Realisierung auf TTL-Bipolar Technologie[3]. Ein VHDL-Projekt sollte vor der endgültigen Realisierung in seinem Zeitverhalten auf die jeweilige Technologie angepasst, und mit dem VHDL-Simulator auf Laufzeitprobleme getestet werden.

- integraler-Test:
  Der integrale Test steht am Ende der Entwicklung einer eigenständig arbeitenden Komponente und des gesamten Entwurfs. Für einen integralen Test bietet sich die Simulation, oder gar die Verwendung realer Daten, als Eingangs-Stimuli an. In der Regel ist zu diesem Test eine spezielle Testumgebung, eine *Testbench*, erforderlich. Diese sollte in der Lage sein, große Datenmengen aus einer Datei des Simulations-Rechners zu lesen und als Signaleingang in das Test-Design einzuspeisen. Die Ergebnisse, das Ausgangs-Signal des Test-Design, sind andererseits wieder in eine Datei des Simulations-Rechners zurück zu schreiben. Ein Vergleich der Datei mit den Dateien vorheriger Testläufe kann, vor allem bei Design-Änderungen, schnell Hinweise auf den Erfolg der Design-Modifikation geben.

- Umwelt-Test:
  Nach der Hardware-Synthetisierung des Entwurfs sind zahlreiche technische und physikalische Tests der Elektronik oder des PGA-Bauteils erforderlich. Dies umfaßt Bauteil-Tests genauso wie Tests der Schaltung auf Spannungsschwankungen, elektrische Signalspitzen, Erschütterungen, Wärme, etc. . Für den Einsatz im Weltraum sind noch zusätzliche Tests unter verschärften Anforderungen erforderlich. So wird zum Beispiel das Verhalten der Schaltung auf Temperaturschwankungen, das Weltraumvakuum oder starke Strahlung getestet.

Die Modularität des EPP-Entwurfs kommt der Testbarkeit stark entgegen. So konnte der Entwurf zunächst komponentenweise getestet werden. Jedes einzelne Modul wurde dazu mit dem Simulator, und wenn erforderlich mit einer eigens angepassten *Testbench*, ausgiebig getestet. Erst nach der erfolgreichen Absolvierung zahlreicher Einzeltests wurden die Module zu Units zusammengefügt. Der Test der Units erfolgte danach ausnahmslos mit Hilfe von *Testbenchs*. Jede Unit wurde dabei in mehreren Stufen auf Syntax, Logik und Funktion getestet. Bei Fehlern oder Problemen wurden die betreffenden Module aus der Unit isoliert, und solange einzeln weiter getestet, bis die Funktionstüchtigkeit der etwaigen Korrekturen nachgeprüft war.

Zu gegebenem Zeitpunkt konnten die fertig gestellten Units zum EPP-Top[4] zusammengefasst, und der Test des EPP-Top mit einer speziell entwickelten *Testbench* durchgeführt werden. Die Tests waren dabei selbst ständig ein wichtiger Bestandteil des Designvorganges und gaben fortwährend neue Impulse für die Verbesserung und Modifikation einzelner Komponenten. Nach einiger Zeit haben die Modifikationen das Design in einem Umfang verändert, daß der isolierte Test von Einzelkomponenten mit den zuvor erstellten Testumgebungen nicht länger praktikabel erschien. Die Testumgebungen hätten dazu ständig umgeschrieben und neu angepasst werden müssen, was von einem gewissen Punkt an mehr Aufwand erfordert hätte, als die Design-Tätigkeit und Fehlerkorrektur zusammen. An diesem Punkt wurde ganz auf den globalen Test mittels der EPP-Top *Testbench* umgestellt. Die Testdaten wurden von da an so geschrieben, daß mit ihnen ein isolierter Test ausgesuchter Komponenten erreicht werden konnte.

5.2 Die Entwicklungs- und Testumgebung

Während der Entwicklung kamen die VHDL-Systeme Xilinx Foundation 2.1i und Model*SIM* EE 5.2 von Model Technology zum Einsatz. Foundation 2.1i ist bisher nur auf WINDOWS basierenden Computersystemen verfügbar, während Model*SIM* sowohl für WINDOWS als auch für zahlreiche UNIX Betriebssysteme angeboten wird.

[3]TTL: *Transistor-Transistor-Logic*

[4]Das EPP-Top -Design besteht derzeit nur aus dem EPP-*Core*, es wird später um zusätzliche Units erweitert. Vgl. Abb. 4.11 in Kapitel 4.2.2

In der Anfangszeit der Design-Entwicklung des EPP wurde ausschließlich mit dem Foundation 2.1i VHDL-System gearbeitet, das eine vollintegrierte Entwicklungsumgebung mit Editor, *Compiler* und Simulator in einer einzigen Anwendung bietet. Vor allem Design und Test der ersten EPP-Module wurden mit diesem Werkzeug erarbeitet. Als die Anforderung an Design und Test wuchsen, erschien Foundation 2.1i dem VHDL-System von Model Technology jedoch deutlich unterlegen, und die Designarbeit wurde deshalb auf dem Model*SIM*-*Compiler*/Simulator 5.2 von Model Technology fortgesetzt. Die Stärke der Model*SIM*-Software liegt vor allem in der leichten Realisierbarkeit einer *Testbench*, die für eine Fortsetzung des Projektes unerlässlich war. Daneben besitzt auch die Model*SIM*-Software eine graphische Oberfläche und einen Simulator mit mächtigen Werkzeugen zur Signalverfolgung. Ihrer Herkunft von UNIX-Systemen ist es vermutlich zu verdanken, daß sowohl *Compiler* als auch Simulator aus einer *Shell* heraus betrieben werden können, was auch die Einbindung in selbst erstellte *Shell*-Skripte anbietet.

5.2.1 Der VHDL-*Compiler*

Der VHDL-*Compiler* hat die Aufgabe, das VHDL-Design in eine maschinenlesbare Form zu übersetzen. Diese dient dem Simulator und später dem Synthesewerkzeug als Eingangs-*File*. Die maschinenlesbaren *Files* liegen allesamt im Verzeichnis *work*. Dieses ist vor dem ersten *Compiler*-Aufruf im jeweiligen Arbeitsverzeichnis anzulegen[5]. Wird ein Entwurf *compiliert*, so testet der *Compiler* zunächst die Syntax des Designs. Bei jeder Verletzung der Syntax wird der Übersetzungsvorgang mit einem ERROR abgebrochen und eine Fehlermeldung angezeigt. Die Fehlermeldung enthält neben dem Fehlertyp in der Regel die Zeilennummer des Quell-Designs, wo der Fehler aufgetreten ist. Neben den ERRORS gibt der *Compiler* auch WARNINGS aus. Dies geschieht, wenn das Design zwar syntaktisch in Ordnung ist, aber andere mögliche Fehlerquellen verborgen sind. Dies kann beispielsweise bei einer Endlosschleife oder einer einseitigen Signalzuweisung (offene Signalleitung) der Fall sein. In der Regel ist das Design erst dann im Sinne von VHDL korrekt, wenn der *Compiler* es ohne Beanstandung bearbeitet hat.

5.2.2 Der VHDL-Simulator

Der Simulator stellt für die CAD-Entwicklung elektronischer Design-Einheiten das wichtigste Diagnosewerkzeug dar. Bei Computer-Programmen kann der Programm-Entwurf nach der Übersetzung mit dem *Compiler* einfach getestet werden. Dazu ist meist lediglich der Programmaufruf des Entwurfs notwendig. Für einen VHDL-Entwurf liegen die Dinge anders, VHDL-Entwürfe beschreiben Hardware-Strukturen. Diese könnten aber ohne einen Simulator nur durch den Aufbau der Hardware (Synthese) auf Korrektheit überprüft werden. Die Stärke des CAD-Designs ergibt sich aber gerade aus der Simulation der Entwürfe vor einer Hardware-Realisierung. Aus diesem Grund bieten alle Hersteller von VHDL-Systemen ausführliche Simulations-Programme zum Test der Entwürfe an.

Nach dem Aufruf des Simulators sind zuerst die Ein- und Ausgänge des Designs zu verbinden. Die Eingänge werden entweder auf konstante Werte gesetzt oder mit Signalquellen verbunden, die der Simulator bereithält. Bei den Signalquellen handelt es sich zumeist um programmierbare *Clock*-Generatoren. Diese erzeugen entweder periodische Signale, oder arbeiten mit Hilfe von Zuweisungstabellen und Zeitprogrammierungen. Ein wichtiger Betriebsmodus der Signalquelle ist auch die manuelle Eingabe. Mit dieser kann der Signal-Zustand von Signalzuweisungen von Hand getaktet werden. Die Reaktion des VHDL-Designs lässt sich dabei quasi in Zeitlupe verifizieren. Die Ausgangssignale des Designs werden mit einem Signal-Monitor, einem sogenannten *Analyzer* verbunden. Dieser zeigt die zeitliche Entwicklung der Signale graphisch an. Wichtig für eine detaillierte Diagnose ist, daß auch Signale die nicht zu den Ein- oder Ausgängen des Designs zählen, und rein interne Zustände repräsentieren, im *Analyzer* zur Anzeige

[5]Dies geschieht mit dem Kommando *vlib work* (Model*SIM*), oder bei Verwendung einer graphischen Oberfläche aus dem Menü heraus (Model*SIM*, Foundation 2.1i).

gebracht werden können. Damit bietet sich dem Entwickler die Möglichkeit, die Betriebszustände des gesamten Designs zu jedem Zeitpunkt und innerhalb variabler Zeitskalen im Detail zu studieren.

Der Simulator eignet sich vor allem für die frühe Entwicklungs-Phase, während der auf unterster Ebene die funktionalen Elemente des Projektes erstellt werden müssen. Die Ausstattungsmerkmale des Simulators eignen sich sowohl um logische, als auch funktionale Fehler auf dieser Ebene zu analysieren. Eine Schwäche der meisten Simulatoren ist jedoch die Erzeugung von Bus-Signalen[6] und die Simulation mit sehr großen Datenmengen. In ersterem Fall sind oft dutzende Signale gleichzeitig zu stimulieren, wobei die Einzel-Stimuli voneinander abhängig sein können. Der zweite Fall tritt meist im finalen Funktionstest ein, wenn das Design mit realen oder simulierten Messdaten getestet werden soll. Dazu sind oft in zeitlicher Abfolge viele tausend verschiedene Stimuli an die Bus-Signale anzulegen. In solchen Fällen ist dann oft die Entwicklung einer *Testbench* der einzig praktikable Ausweg.

5.2.3 Die *Testbench*

Eine *Testbench* ist eine spezielle, in VHDL realisierte, Testumgebung zur Verifizierung komplexer VHDL-Designs. Die *Testbench* kann nicht vom Hersteller des VHDL-Systems bereitgestellt werden, sondern muß während der Entwicklungsphase für das Projektes speziell entworfen werden. Ihr Vor- und Nachteil zugleich ist, daß sie auf den zu testenden Entwurf maßgeschneidert wird. Dies erlaubt einerseits einen hohen Grad der Anpassung an die Testerfordernisse, ist anderseits aber mit hohem Aufwand verbunden, insbesondere da jede Designänderung auch eine Änderung der *Testbench* erfordern kann.

Im Falle des EPP soll die *Testbench* eine Simulation mit Messdaten ermöglichen. Diese werden bisher künstlich erzeugt, sollen aber später durch echte Messdaten ersetzt werden. Für diese Aufgabe werden die Daten in einer Datei des Test-Rechners bereitgehalten und von der *Testbench* in das Test-Design (den EPP-*Core*) eingespeist. Die *Testbench* ist dafür wie in Abbildung 5.1 aufgebaut. Sie besteht aus den VHDL-Modulen LOAD und SAVE. Dazwischen befindet sich das zu testende VHDL-Design, z.B. das EPP-*Core*-Design.

Die genauen Aufgaben der *Testbench* sind vom zu testenden Design abhängig. In diesem Fall soll der EPP-*Core* im derzeitigen Entwicklungs-Stand getestet werden. Der EPP-*Core* besteht derzeit aus den Units EC-Unit, FD-Unit und BD-Unit, vgl Abb. 5.2 (grau unterlegter Bereich). Die *Testbench* wird in diesem Fall die Eingangs-Daten der EC-Unit, also die *Event-Packets* der noch nicht verfügbaren EPA-Unit, erzeugen. Auf der anderen Seite übernimmt sie das abspeichern der Ausgangs-Daten des EPP-*Core* und ersetzt damit vorerst die *Ring-Buffer*- und Telemetrie-Unit.

Der Betrieb der *Testbench* wird derzeit mit einem RESET- und LOAD-Signal eingeleitet. Beide Signale werden nacheinander von der *Testbench* erzeugt und an das *Core*-Design angelegt. Das RESET-Signal hat die Aufgabe, alle Signale des *Core-Designs* in einen eindeutig definierten Zustand zu versetzen. Nach dem RESET wird automatisch ein LOAD-Signal angelegt. Dieses leitet das Einlesen des *Look-Up-Tables* (LUT) der Amplituden-Korrektur ein. Erst nach vollständig beendetem LOAD-Zyklus[7] kann die eigentliche Datenverarbeitung innerhalb des EPP-*Core*-Designs beginnen.

Das LOAD-Modul der *Testbench* hat die Aufgabe, die simulierten Messdaten aus der Festplatten-Datei DET.dat zu lesen, und an das EPP-*Core*-Design zu übergeben. Die Übergabe geschieht mit Hilfe des RNE-*Pipeline*-Taktes, der ebenfalls vom LOAD-Modul der *Testbench* erzeugt wird[8]. Mit jedem RNE wird ein neuer Datensatz von der Festplatte des Test-Rechners gelesen und an die EC-Unit des EPP-*Core*-Designs

[6] Ein *Bus* besteht in der Elektronik aus einer Signalzuweisung mit sehr vielen parallelen Einzelsignalen. In der Computer-Technik ist der Begriff weit verbreitet.

[7] Der LOAD-Zyklus dauert 1024 *Clock*-Impulse, was etwas mehr als 10 RNE-Zyklen entspricht. Da während des LOAD-Vorgangs kein Korrektur-Betrieb möglich ist, werden die *Event-Packets* in dieser Zeitspanne ungültig. Die ersten 11 Datensätze, bzw. *Event-Packets* eines jeden Test-Zyklus werden somit in jedem Fall ungültig. Die Daten für den *Look-Up-Table* liegen in der Datei LUT.dat.

[8] In der fertigen Version des EPP wird das RNE-Signal von der EPA-Unit erzeugt. Da diese Unit bisher noch nicht existiert, wird das RNE-Signal innerhalb des LOAD-Moduls der *Testbench* erzeugt.

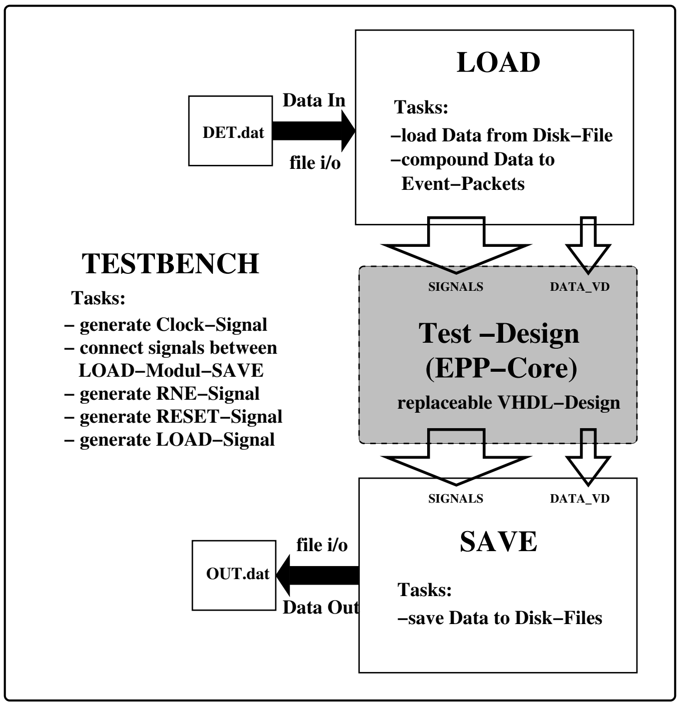


Abbildung 5.1: *Die Testbench. Für den EPP-Test wurde die hier dargestellte* Testbench *entworfen. Das aufrufende VHDL-Design (Testbench) verbindet die VHDL-Module LOAD und SAVE mit dem Test-Design (in diesem Fall dem EPP-Core). Das LOAD-Modul liest die* Event-Packets *als Testdaten aus einem Festplatten-*File *und übergibt sie an den* EPP-Core*. Die Ausgabe des* EPP-Core *wird vom SAVE-Modul wieder auf Platte zurückgeschrieben.*

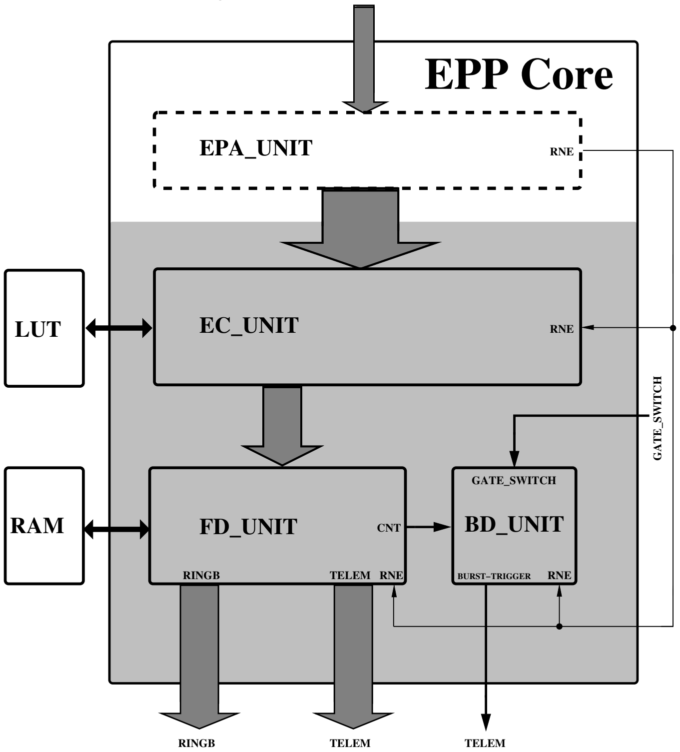


Abbildung 5.2: *EPP-Core. Der Kern des* Event-Pre Processors *enthält alle operativen Aufgaben des Designs. Der bisher fertiggestellte Teil ist grau unterlegt.*

gegeben. Jeder Datensatz entspricht dabei einem vollständigen *Event-Packet*, wie es später von der EPA-Unit erzeugt wird.

Auf diese Weise kann eine große Anzahl von *Event-Packets* simuliert werden. Dies ist auch erforderlich, da zum Test der BD-Unit die Zählraten der gültigen *Event-Packets* über Zeitintervalle von Sekunden zu akkumulieren sind. Für den Burst-Detektor der BD-Unit ist sogar der Mittelwert der Zählraten erforderlich. Der Test dieser Unit erfordert den Einsatz mehrerer tausend *Event-Packets* pro Testlauf. Das Ergebnis wird vom SAVE-Modul der *Testbench* in mehrere Dateien auf Festplatte geschrieben. Dieses Schreiben geschieht ebenfalls synchron zum RNE-Takt, genauso wie später für den Betrieb der wirklichen Ausgangs-Module vorgesehen. Es existieren derzeit Ausgabe-*Files* für Telemetrie (TELEM.dat), *Ring-Buffer* (RINGB.dat), *Burst-Detektor* (BSTDT.dat), *Burst-Snooper* (BSN.dat) und ein Test-Ausgang (TEST.dat) der alle Daten unzensiert abspeichert. Die letzten beiden Ausgänge dienen lediglich Diagnosezwecken und werden später still gelegt.

Die Verifikation des Test-Designs, in diesem Fall die Funktion des EPP-*Core*, geschieht sowohl mit dem graphischen *Analyzer* des Simulators, als auch durch gezielte Auswertung der Ausgabe-*Files* der *Testbench*. Die Eingangs-*Files* der *Testbench* werden derzeit von Hand erzeugt, die Ausgabe-*Files* von Hand analysiert. Die Eingangs-Daten müssen in einem speziellen Format erstellt werden, bevor das LOAD-Modul der *Testbench* sie in den EPP-*Core* einspeist. Ebenso liegen die Ausgabe-*Files* des SAVE-Moduls in einem Format vor, das einiger Konvertierung bedarf, bevor die Daten interpretierbar sind. Für einen dauerhaften, ins Detail führenden, Testbetrieb ist dieses Vorgehen ineffizient. Eine der nächsten Aufgaben im Zusammenhang mit dem EPP-Test wird deshalb die Entwicklung einer geeigneten Konvertierungs-*Software* sein müssen. Dies könnte zum Beispiel in Form eines PERL-Skripts geschehen, das neben den Konvertierungen auch die *Testbench* aufruft, und so den gesamten Test-Vorgang automatisiert.

Als Beispiel eines Testlaufs des EPP-*Core* mit der *Testbench*, ist in Abbildung 5.3 ein *Screenshot* des *Analyzers* gezeigt. In der obersten Zeile ist das CLK-Signal zu sehen. Die Betriebsfrequenz des EPP ist zum Test der Funktionalität stark reduziert worden. Dies ermöglicht einen besseren Überblick des Signalverlaufs[9]. Die zweite Zeile zeigt das RNE-Signal, es tritt exakt alle 100 CLK-Impulse auf und definiert den *Pipeline*-Takt der EPP-Units. Das Signal REG_IN ist ein Signal-Bus. Es hat eine Breite von 168 Bit und transportiert das Eingangs-*Event-Packet* des EC_IN-Moduls. Nachdem das REG_IN mit RNE gelesen wurde (Signal-Busse AMPL_OUT, TIME_OUT, REG_IN weiter unten) lädt die *Testbench* das nächste *Event-Packet* von der Festplatte. Das Signal ändert sich deshalb kurz nach RNE. Weiter unten sind die 9 Sub-RNE-Signale der EC-Unit gezeigt (RNE_1 bis RNE_9). Diese sind jeweils 10 CLK-Impulse lang, und steuern den internen Ablauf der EC-Unit. In den anschließenden Zeilen ist die Ablauf-Steuerung des AC-Moduls verdeutlicht. Dazu wurden die entsprechenden Signale jeweils unter das auslösende Sub-RNE-Signal gestellt. Die Signale existieren jeweils in sechsfacher Ausfertigung, da alle *Hits* gleichzeitig korrigiert werden. Ganz unten sind die Ausgangs-Signale des EPP-Core zu sehen (EPP_RINGB_SAVE, EPP_TESTD_SAVE,EPP_BSN_SAVE, EPP_TELEM_SAVE). Diese werden synchron mit RNE auf Festplatte geschrieben. Die letzte Zeile zeigt die Übergabe des *Event-Packets* der EC-Unit an die FD-Unit. Das FDIN-Modul übernimmt das *Event-Packet* synchron mit dem RNE-Signal, der Bus besitzt hier eine Breite von 45 Bit.

Abbildung 5.4 zeigt einen *Screenshot* mit anderer Zeitauflösung. Die Impulse des CLK-Signals liegen hier bereits so dicht, daß sie nicht mehr aufgelöst werden, das RNE-Signal erscheint als eine Reihe von Nadeln. Diese werden nach wie vor alle 100 CLK-Impulse ausgegeben und definieren den *Pipeline*-Takt des EPP. Die dritte und vierte Zeile verdeutlicht sehr schön den Lade-Vorgang des *Look-Up-Tables*. Es werden etwas mehr als 10 RNE-Zyklen zum lesen der Korrekturwerte benötigt. Das Signal LUT_LOAD bleibt dabei solange auf 1, wie das LUTC noch Werte empfängt. Während des Lade-Zeitraums werden alle *Event-Packets* der *Pipeline* ungültig. Dies bewirkt das Signal VD_LUT. Es bleibt solange auf 0 bis der Ladevorgang beendet ist, und der nächste *Pipeline*-Takt beginnt. Bis dahin sind alle *Event-Packets* ungültig. Die nachfolgenden Zeilen zeigen Bus-Signale der EPP Ein- und Ausgänge. Jede der kleinen Klammern symbolisiert die Änderung des Signal-Busses. Sehr schön erkennbar ist, wie der *Ring-Buffer*-Ausgang (Signal EPP_RINGB_SAVE) nur Werte für gültige *Event-Packets* auf den Ausgang schreibt. Das

[9] Die Zeitbemaßung des *Analyzers* ganz unten steht dazu übrigens in keinem Verhältnis. Die Frequenz des EPP wird bisher in der *Testbench* festgelegt.

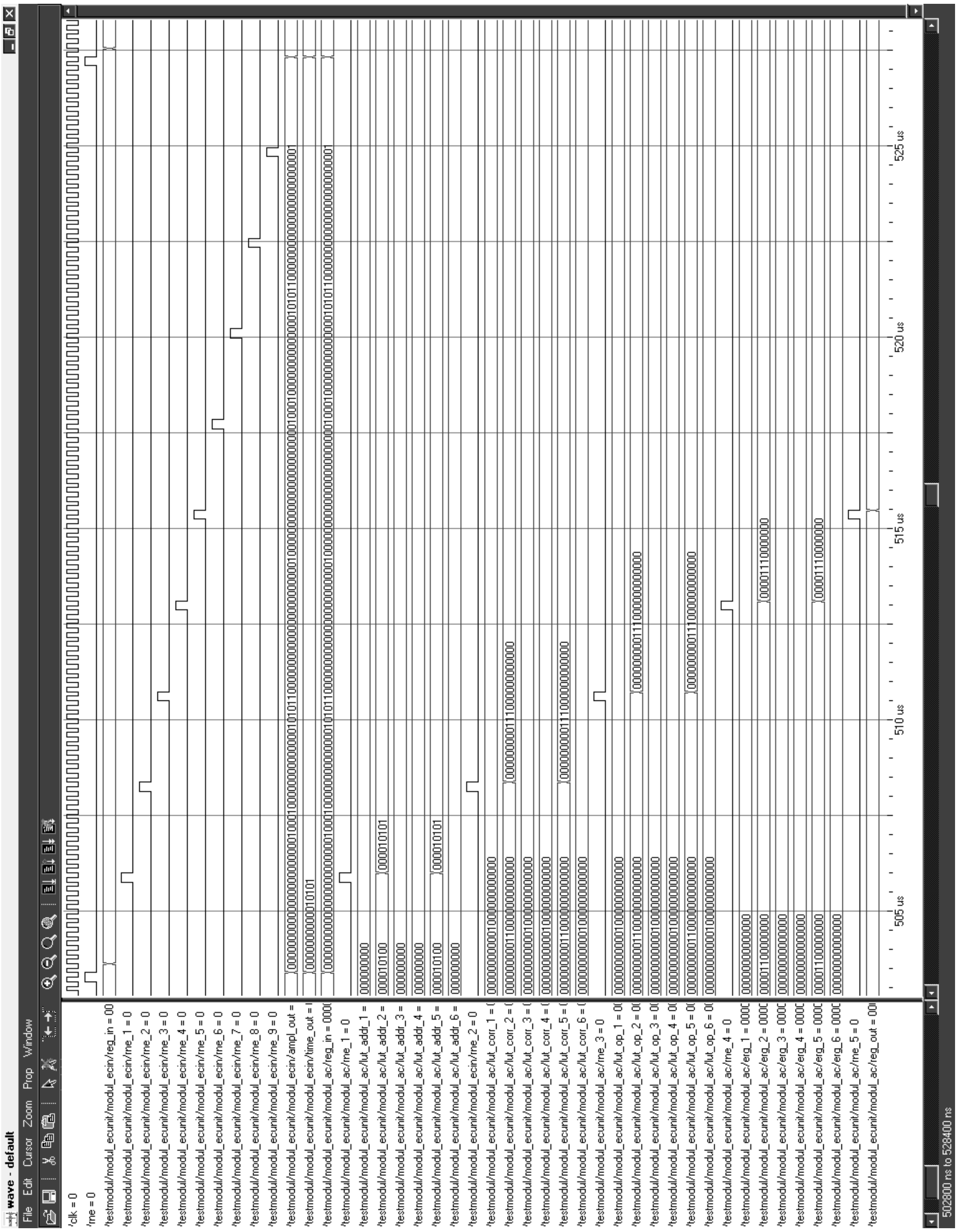


Abbildung 5.3: *Screenshot* eines Simulator-Laufs des EPP-Core Designs, siehe Text.

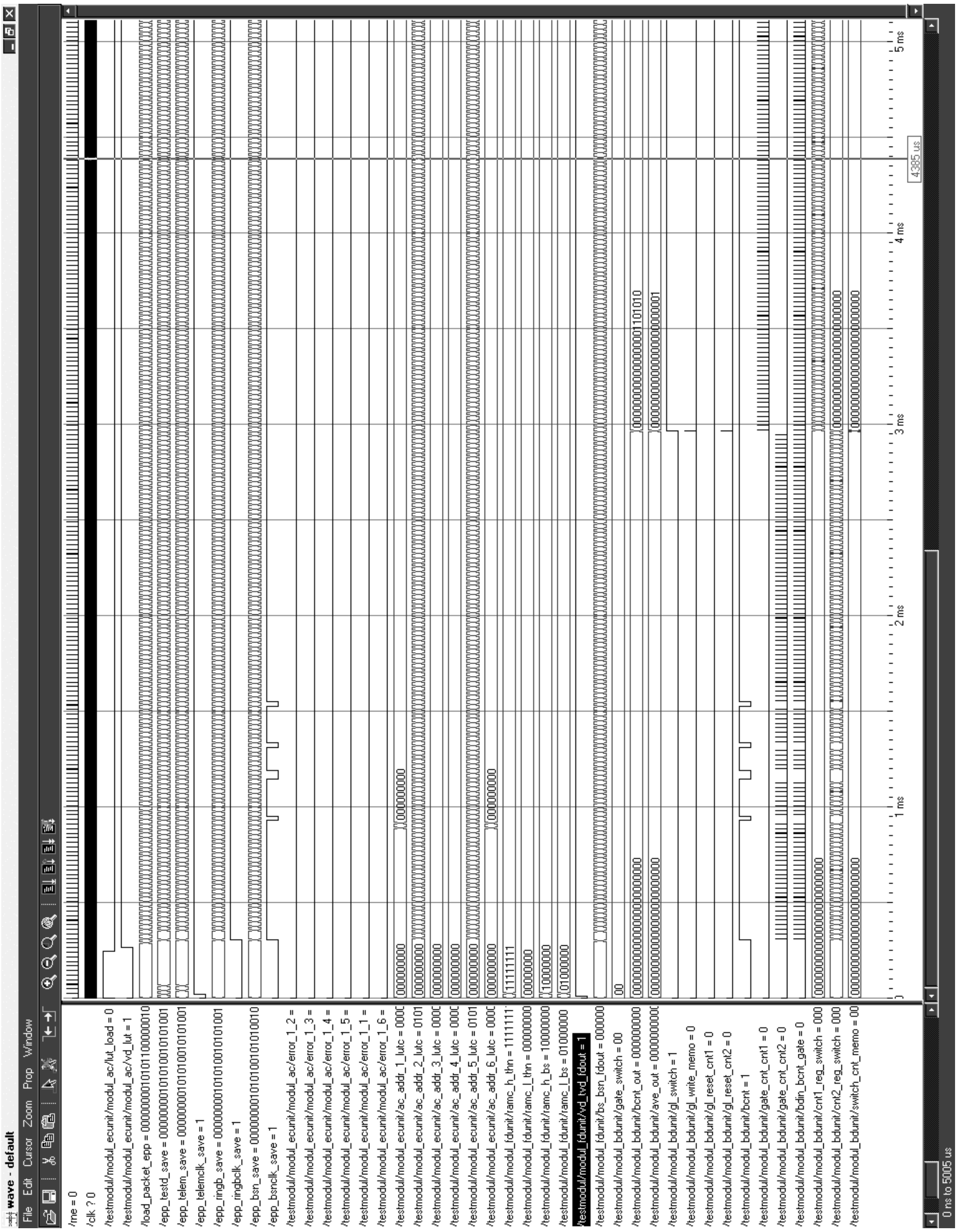


Abbildung 5.4: *Screenshot* eines Simulator-Laufs des EPP-Core Designs, siehe Text.

Signal EPP_RINGBCLK_SAVE bleibt dazu solange auf 0, bis das erste gültige *Event-Packet* am Ausgang des EPP erscheint. Es bleibt somit genau 2 RNE-Zyklen länger auf 0 als das VD_LUT-Signal, die Zeitspanne, die ein *Event-Packet* zum Durchlaufen des EPP-Core benötigt. Genau wie der *Ring-Buffer*-Ausgang schreibt auch der BSN-Ausgang (EPP_BSN_SAVE) nur gültige *Event-Packets*, während der Telemetrie- und Test-Ausgang (EPP_TELEM_SAVE, EPP_TESTD_SAVE) die *Event-Packets* unabhängig von der Gültigkeit speichert. Wie am EPP_RINGBCLK_SAVE-Signal erkennbar ist, verarbeitet die Simulation in diesem Testlauf nur potentiell gültige *Event-Packets*. Mit Ausnahme der *Load*-Phase des LUT ist das Signal nämlich konstant auf 1, alle *Event-Packets* sind somit gültig. Anders verhält sich das Signal EPP_BSNCLK_SAVE, es signalisiert die Gültigkeit der *Events* bezüglich des *Burst*-Energie-Filters. An vier Stellen fällt das Signal EPP_BSNCLK_SAVE auf 0 ab, d.h. die Energie der untersuchten *Event-Packets* entsprach dort nicht den eingestellten Kriterien des *Burst-Snoopers* (insgesamt 5 *Event-Packets*). Das gleiche Signal taucht weiter unten noch einmal auf (BCNT). Dort sind die Signale der BD-Unit gezeigt, sie steuern die Funktion der beiden *Event-Zähler*. Findet der *Burst-Snooper* Ereignisse, die nicht den Energie-Kriterien entsprechen, so werden die zugehörigen *Event-Packets* nicht in die Auszählung gelangen. Dieser Vorgang ist im Vergleich der Signale BCNT und GATE_CNT_CNT2 zu erkennen. Die Signale GATE_CNT_CNT1 und GATE_CNT_CNT2 transportieren den Zählerstand der beiden *Event*-Zähler der BD-Unit. Die Zähler sind abwechselnd aktiv. In der Ruhezeit wird der inaktive Zähler in das AV-Modul gelesen und zurück gesetzt. Der Vorgang wird mit den Signalen GL_SWITCH, GL_WRITE_MEMO und GL_RESET_CNT1/2 gesteuert. Der *Screenshot* zeigt sehr schön, wie die Zähler mit Hilfe des GL_SWITCH-Signals umschalten, und nacheinander die Steuersignale GL_WRITE_MEMO und GL_RESET_CNT1/2 angelegt werden. Die Signale erscheinen in der Abbildung nur als Nadel-Spitzen sind aber in Wahrheit genau ein CLK-Signal lang. Das GATE_SWITCH-Signal steht auf 00, was einer Torzeit von 1/8 Sekunde entspricht. Bei einer *Clock*-Frequenz von 24 MHz finden 30000 *Event-Packets* innerhalb eines solchen Zählerlaufs Platz. Für die hier gezeigten Simulation ist die Torzeit deshalb so stark reduziert worden, daß die Zählimpulse mit bloßem Auge sichtbar werden. Für die statistische Auswertung der Zählraten sind sehr viele 0/1-Wechsel des GL_SWITCH-Signals nötig, allein um den Zählraten-*Buffer* der *Burst-Trigger*-Einheit zu füllen werden 64 benötigt. Der Vorgang der *Burst*-Detektion ist deshalb auf dem Zeitrahmen dieses *Screenshot* nicht mehr darstellbar. Die Verwendung des *Analyzers* wird für große Zeiträume immer problematischer. Es ist dann sinnvoller für die Verifizierung, die Ausgabe-*Files* der *Testbench* zu verwenden, und diese zu analysieren.

Kapitel 6

Der Stand der Dinge

6.1 Ausblick

Das EPP-Projekt in seinem bisherigen Stand bietet einen guten Ausgangspunkt für die weitere Entwicklung und Realisierung eines *Event-Pre*-Prozessors. Freilich werden bis zur endgültigen Fertigstellung eines FPGA noch zahlreiche Ergänzungen und Modifikationen zu bewerkstelligen sein. Trotzdem kann auf der Basis dieser Arbeit bereits mit ersten Tests zur Hardware-Realisierung und mit dem Aufbau einer Testumgebung begonnen werden. Einige Modifikationen wurden bereits innerhalb dieser Arbeit vorgeschlagen. In der Regel ist es zu Beginn der Entwurfstätigkeit kaum möglich, alle Probleme und *Features* eines Projektes bis ins Detail vorauszusehen, bzw. vorauszuplanen. Zahlreiche Änderungen noch während der Design-Phase sind somit vorprogrammiert. Ein paar dieser Änderungen sollen in diesem Kapitel noch angeregt werden:

- Betriebsmodi
  Wie bereits früher erwähnt, erscheint die Einrichtung mehrerer Betriebsmodi sehr sinnvoll. Die Anzahl der Modi ist etwas vom Einsatzgebiet des CMI abhängig. Konkret könnten jedoch z.B. die folgenden Modi in Betracht gezogen werden.

  1.) Meßbetrieb: Der Modus zum Meßbetrieb entspricht schlicht dem Standard-Betriebsmodus wie er in dieser Arbeit beschrieben wurde. Er dient vor allem der Verwendung des CMI als *Burst-Monitor*, oder auch für astronomische Untersuchungen im Röntgen- und Gamma-Bereich.

  Bei einer Verwendung als *Gamma-Ray*-Observatorium ist der Standard-Modus vielleicht nochmals zu unterteilen. Je nachdem ob die Beobachtung im *Pointierten*- oder im *Scan*-Betrieb erfolgen soll, könnte auch eine Anpassung der Funktionsweise des EPP erforderlich werden.

  2.) *Boot*-Modus: Dieser Modus dient dem *booten* der CEU-Elektronik. Wie bereits besprochen, gehen nach jedem *Reset* eine Reihe von Messwerten verloren, und es verstreichen ein paar Sekunden bis die Elektronik betriebsbereit ist. In dieser Phase werden keine brauchbaren Messdaten erzeugt, und wahrscheinlich jede Menge Fehl-*Trigger* ausgelöst. Der Initialisierungs-Zeitraum sollte durch einen speziellen *Boot*-Modus realisiert werden, der ungültige *Trigger* maskieren könnte.

  3.) Transparent-Modus: Der Transparent-Modus sollte eine weitgehende Überbrückung der EPP-Funktionen bieten. Dies könnte im Bedarfsfall die Übertragung unkorrigierter Rohdaten ermöglichen, sei es zu Testzwecken, oder weil das Interesse auf einem Detail liegt, das normalerweise unerwünscht ist und vom EPP sonst verworfen wird. Wegen der starken Datenreduktion des EPP müßte sich die Übertragung allerdings auf einen Ausschnitt der Daten beschränken. Dieser Ausschnitt sollte von der Bodenstation programmierbar sein.

  4.) Diagnose-Modus: Der Diagnose-Modus sollte ermöglichen, daß ein Großteil der Daten ungefiltert und mit allen Fehler-*Bits* übertragen wird, soweit die Telemetrie dies zulässt. Im Falle von

Betriebsstörungen könnte die Ursache während der Mission diagnostiziert und möglicherweise beseitigt oder unterdrückt werden. Auch eine Überwachung der Detektoralterung mit Hilfe der Kalibrationsquelle könnte sich in diesem Modus abspielen.

Die Verwendung von Betriebsmodi, wie hier vorgeschlagen, erfordert einen EPP -internen *Controller*-Baustein notwendig. Dieser *Controller* muß während des Betriebs des EPP vom Boden aus programmierbar sein. Er benötigt direkten Zugriff auf alle HBR-*Interfaces* und auf den Veto-Detektor der FD-Unit.

- *Error-Codes*:
  Für eine detaillierte System-Diagnose ist das bisherige Konzept der VD-Bits um einen mehrstelligen *Error-Code* zu erweitern. Das *Event-Packet* könnte damit nur noch ein einziges VD-Bit verwenden, und die genauere Darstellung des Fehlers in den Fehler-*Code* verlagern. Der Fehler-*Code* würde auch eine Klassifizierung der Fehler zulassen. In manchen Fällen könnten damit vielleicht wertvolle Daten gerettet werden, die sonst wegen eines harmlosen Bereich-Überlaufs aus der Auswertung ausscheiden müßten.

- *Onboard* DOI-Korrektur:
  Durch eine geeignete Tiefenkorrektur kann die Energie-Auflösung des Detektors erhöht werden. Dazu ist der DOI-Wert als Quotient aus Anoden- und Kathoden-Summe zu berechnen. Nach der bisherigen Planung wird der Korrektur-Wert zwar an Bord ermittelt, die Korrektur selbst jedoch erst am Boden durchgeführt. Eine DOI-Korrektur bereits an Bord erscheint jedoch als denkbare Alternative. Dies würde die Notwendigkeit für eine Übertragung des DOI vermeiden, und der *Burst*-Detektor könnte bereits mit korrigierten Messwerten arbeiten. In dem Fall, daß der Tiefenfehler einer analytischen Beschreibung zugängig ist, könnte auch die Präzision der Korrektur deutlich erhöht werden. Der DOI-Wert kann EPP-intern deutlich genauer übertragbar werden, als dies über die Telemetrie hin möglich ist.

- Programmierung des *Burst-Detektors*:
  Der *Burst*-Detektor verwendet bisher einen fest eingestellten Wert für das 5 σ-Kriterium. Diese *Trigger*-Schwelle sollte wenn möglich von der Bodenstation frei programmierbar sein. Zu diesem Zweck ist das CMP-Modul des *Burst*-Detektors um die erforderlichen Signalleitungen zu erweitern. Ich schlage die Verwendung eines 3 Bit breiten Signals vor. Damit könnten 8 verschiedene *Trigger*-Werte im Bereich zwischen z.B. 1σ bis 10σ eingestellt werden.

6.2 Zusammenfassung

Die Aufgabe dieser Diplomarbeit war die Entwicklung und der Test eines *Event-Pre-Processors* für das CMI-Projekt. Dazu wurden neben dem EPP-Konzept (vgl. Abb. 4.11, Kapitel 4.2.2), zahlreiche VHDL-Entwürfe erarbeitet. Das EPP-Top -Design besteht derzeit ausschließlich aus dem *Core*-Bereich des EPP. Darin enthalten sind die drei Units EC-, FD- und BD-Unit, die den Kern des Prozessors darstellen und in sich alle operativen Aufgaben vereinen.

In der derzeitigen Entwicklungs-Phase umfaßt das Design insgesamt 26 VHDL-Module, die sich auf 3 Units verteilen. Hinzu kommt das EPP-Top -Design und die Designs der *Testbench*. Eine Zusammenstellung aller bisher vorhandenen Entwürfe steht in Tabelle 6.1. Die Design-*Listings* sind aus Platzgründen auf CD-ROM im Anhang dieser Arbeit zu finden. Ebenfalls im Anhang befinden sich die Signal-*Schematics* der drei *Core*-Units, die bisher realisiert wurden. Auf diesen ist die strukturale Verknüpfung der einzelnen VHDL-Designs im derzeitigen Entwicklungsstand gezeigt.

VHDL-Design	Aufgabe	Aufrufendes Design
EC_IN-Modul	RNE-synchrone Paket-Übergabe	EC-Unit
AC-Modul	Amplituden Korrektur	EC-Unit
LUTC-Modul	Controller Look-Up-Table	EC-Unit
EL-Modul	Position des Ereignisses	EC-Unit
SOA-Modul	Summe der Anoden	EC-Unit
SOC-Modul	Summe der Kathoden	EC-Unit
DIV-Modul	Erzeugung DOI	EC-Unit
DC-Modul	Tiefen Korrektur	EC-Unit
TC-Modul	Zeit Korrektur	EC-Unit
ED-Modul	Fehler Detektor	EC-Unit
EC_OUT-Modul	RNE-synchrone Paket-Übergabe	EC-Unit
FD_IN-Modul	RNE-synchrone Paket-Übergabe	FD-Unit
THN-Modul	Störunterdrückung	FD-Unit
BS-Modul	Burst Filter	FD-Unit
RAMC-Modul	Controller Register	FD-Unit
VD-Modul	Veto Detektor	FD-Unit
FD_OUT-Modul	RNE-synchrone Paket-Übergabe	FD-Unit
BD_IN-Modul	RNE-synchrone Zählimpuls-Übergabe	BD-Unit
GC-Modul	Erzeugung Gate-Signal	BD-Unit
GL-Modul	Erzeugung von Steuersignalen	BD-Unit
GATE-Modul	Zähler Eingang	BD-Unit
CNT1-Modul	Ereignis-Zähler für Zählraten	BD-Unit
CNT2-Modul	Ereignis-Zähler für Zählraten	BD-Unit
SWITCH-Modul	Zähler Auswahl	BD-Unit
AV-Modul	Mittelwert für Zählraten	BD-Unit
CMP-Modul	Zählraten Vergleich	BD-Unit
HC	Hexadezimal-Zähler	BD-Unit
DFF	D Flip Flop	BD-Unit
TFF	T Flip Flop	BD-Unit
EC-Unit	strukturale Verdrahtung der EC-Module	EPP_Top
FD-Unit	strukturale Verdrahtung der FD-Module	EPP_Top
BD-Unit	strukturale Verdrahtung der BD-Module	EPP_Top
EPP_Top	EPP Top Design (bisher Core-Design)	Testbench
LOAD	Laden und Formatieren von Testdaten	Testbench
SAVE_TM	Sichern von Telemetrie-Daten	Testbench
SAVE_RB	Sichern von Ring-Buffer-Daten	Testbench
SAVE_BD	Sichern von Burst-Daten	Testbench
SAVE_BS	Sichern von Daten des Burst Snoopers	Testbench
SAVE_TST	Sichern aller Ausgabe-Daten	Testbench
TESTBENCH	EPP Testbench	

Tabelle 6.1: *VHDL-Designs der bisherigen Entwicklungsphase*

Die VHDL-Designs wurden bisher sowohl auf der Modul- als auch auf der Unit-Ebene ausführlich getestet. Eine Fortsetzung dieser Tests ist insbesondere im Hinblick auf eine Ergänzung und Verfeinerung der Designs unbedingt erforderlich. Hierzu ist das bestehende Testprogramm durch zahlreiche zusätzliche Tests zu erweitern. Um ein vollständiges Testprogramm zu erstellen, und den Arbeitsaufwand pro Test zu reduzieren, wird die nächste Aufgabe im Zusammenhang mit der EPP-Entwicklung, vermutlich die Entwicklung einer automatisierten, skriptgesteuerten Testumgebung sein.

Anhang A

Signallaufpläne des EPP

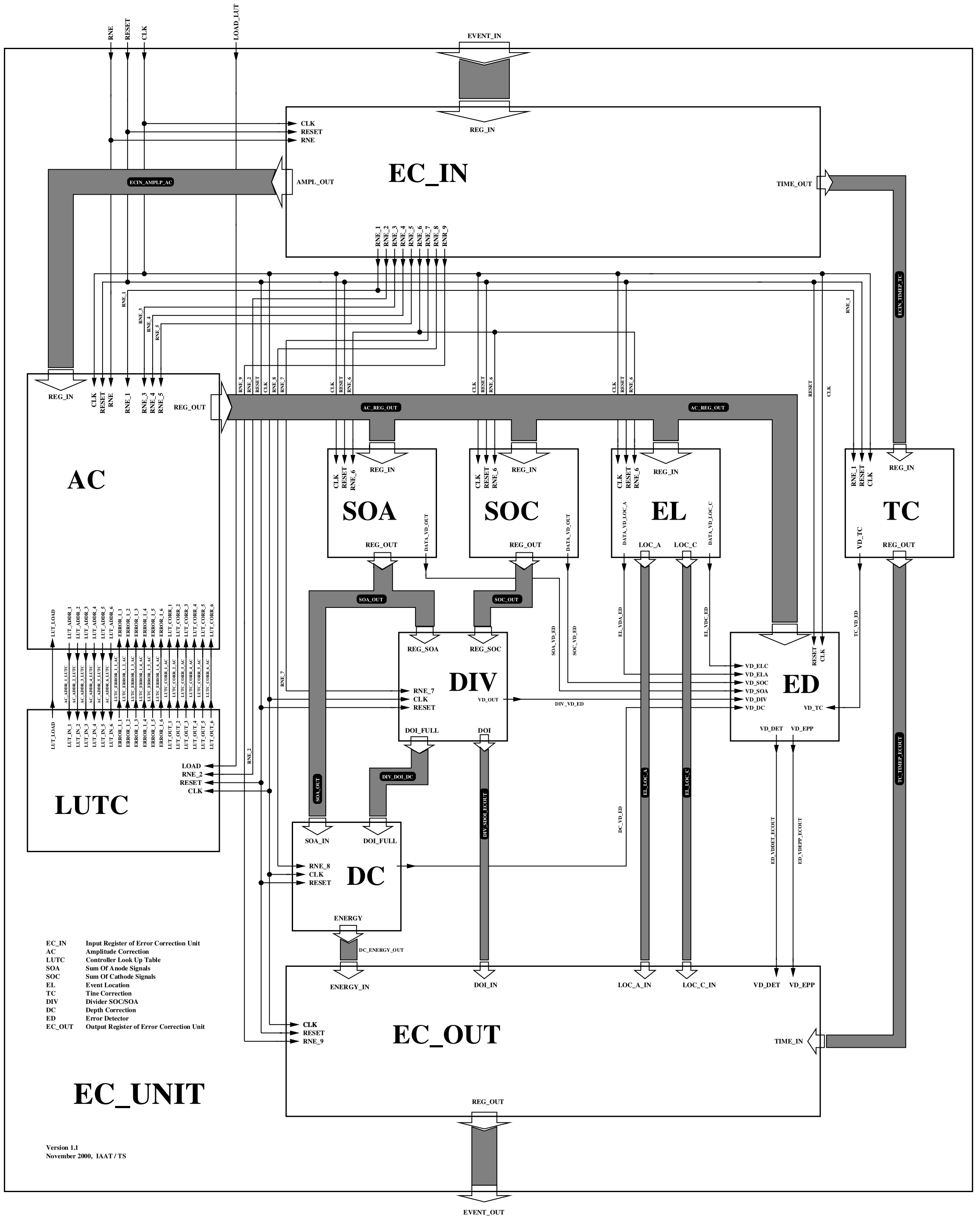
A.1 Anhang

Anhang A enthält die Signal-*Schematics* der in dieser Arbeit fertig gestellten VHDL-Designs. Die *Schematics* sind dabei keine Elektronik-Schaltpläne im klassischen Sinne, sondern geben die strukturale Verknüpfung der VHDL-Module wieder. Die einzelnen Kästchen repräsentieren die ENTITYS der Modul-Designs. Die Verbindungen zwischen den Kästchen entsprechen Signalen des hierarchisch höheren Unit-Designs.

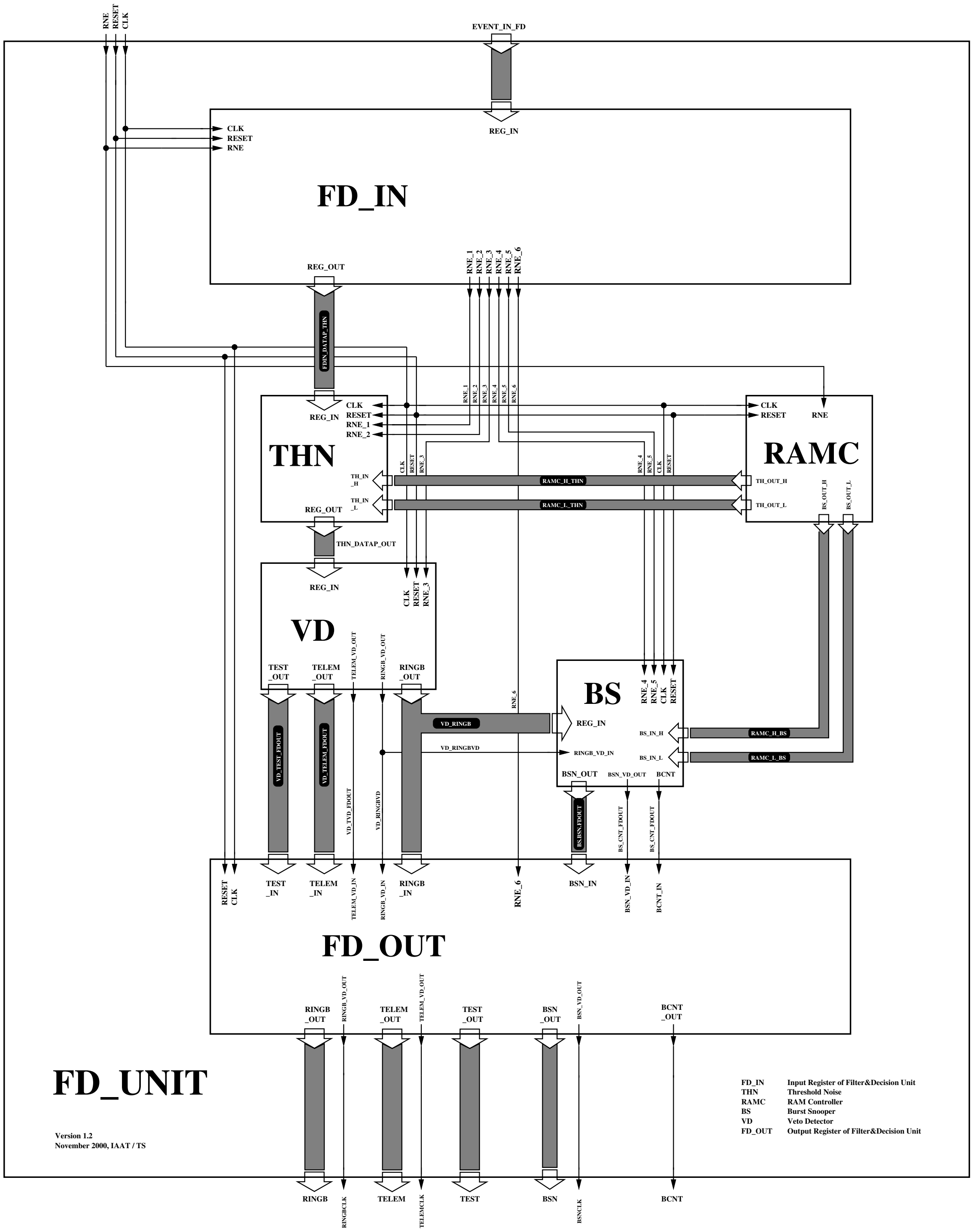
Schematics:

- EC-Unit: Fehler-Korrektur und Datenvorverarbeitung.
- FD-Unit: Filterung der *Event-Packets* und Störunterdrückung.
- BD-Unit: *Event*-Auszählung und *Burst-Trigger*.

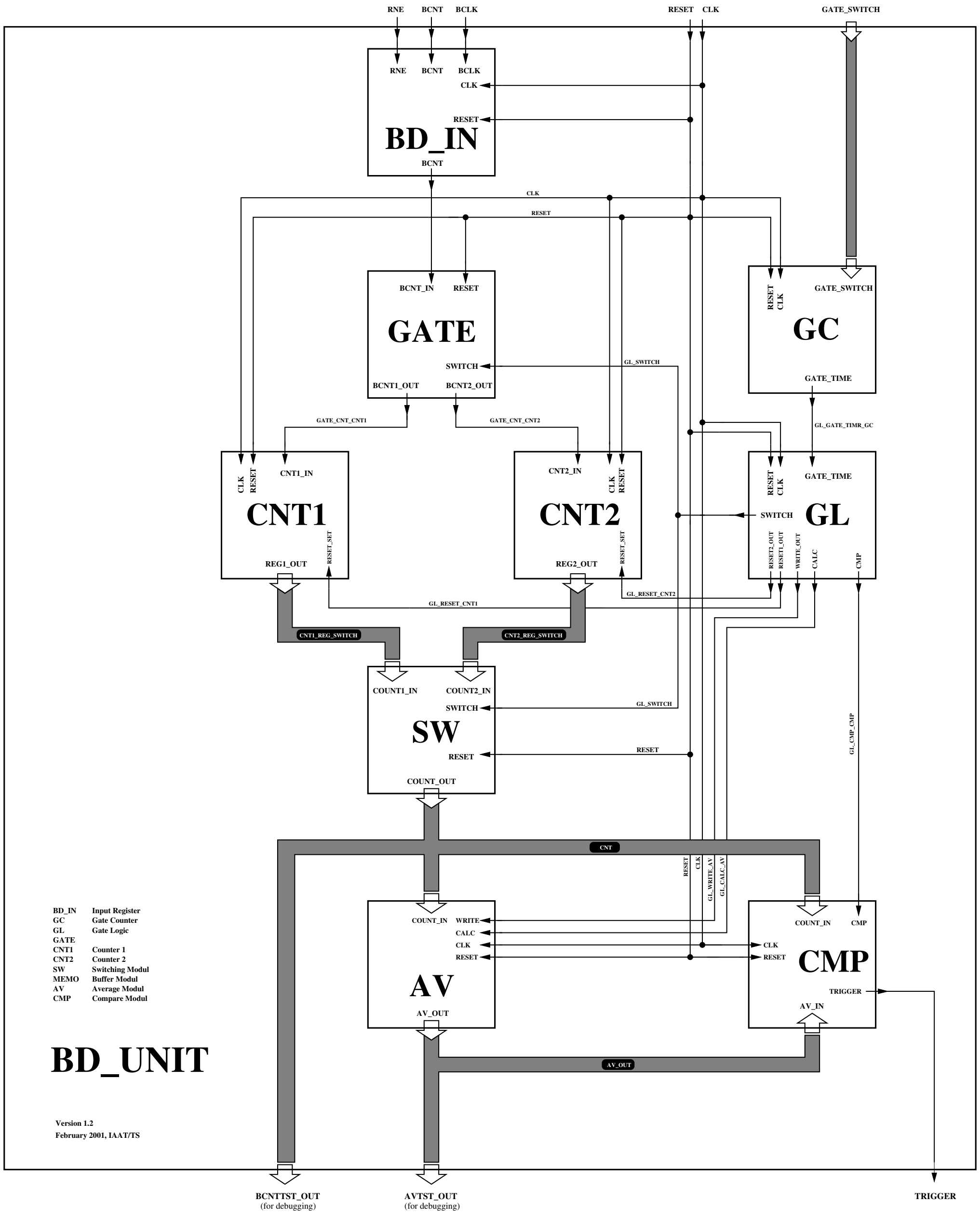
Signal - Schematic of EPP-EC_Unit:



Signal - Schematic of EPP-FD_Unit:



Signal – Schematic of EPP–BD_Unit:



Anhang B

VHDL Designs des EPP

B.1 EPP_TOP-Design

Anhang B enthält die VHDL-Designs, die im Rahmen dieser Arbeit entworfen wurden. Die Listings der insgesammt 41 VHDL-Designs sind aus Platzgründen auf beiliegender CD enthalten und nicht einzeln abgedruckt.

VHDL-Designs der CD:

```
-----------------------------------------------------------------------
--------- TOP Design of the EPP  , Rev 0.1 , 06.10.00, TS/IAAT --------
-----------------------------------------------------------------------


---
-- EPP_TOP  / Testbench
---


--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from Testbench
--
-- RNE           : Pipeline Tag from EPA_UNIT (Testbench)
--
-- LOADLUT       : starts LUT-load process if signal is high
--
-- MODE          : 2 Bit Mode-Set (not used yet) , must be set by hand before
--                 running Testbench.
--
-- GATETIMESET   : 2 Bit signal for setting Gate-Time. Must be set by hand
--                 before running Testbench.
--
-- IN_EVENT_EC   : 168 Bit Event-Packet from EPA_UNIT (Testbench)
--
-- IN_EVENT_EC:
--
--           [ 167       Timestamp      151 : 150   Spare   144 ]
-- Hit 1     [ 143          130 : 129       121 :     120      ]
-- Hit 2     [ 119          106 : 105        97 :      96      ]
-- Hit 3     [  95           82 :  81        73 :      72      ]
-- Hit 4     [  71           58 :  57        49 :      48      ]
-- Hit 5     [  47           34 :  33        25 :      24      ]
-- Hit 6     [  23           10 :  9          1 :       0      ]
--
------
--
-- Task: Structural Design to connect the units of the EPP Core
--       with each other, and the EPP-Core  with the Testbench
--       Designs: LOAD, TESTBENCH, SAVE_X
--
--
------
--
-- Output Signals:
--
-- RINGB         : 41 Bit Output Event-Packet for Ring-Buffer, currently
--                 connected with SAVE_RB-Module
--
-- RINGB:        [ 41   Time-Stamp   25 : 24  Location Anode 18
--               : 17 Location Cathode 11 : 10    Energy    3
--               : 2 DOI 0 ]
--
-- RINGBCLK      : Trigger Signal for writeout from SAVE_RB-Module to
--                 disk-file.
```

```
--
-- TELEM          : 45 Bit Output Event-Packet for Telemetry, currently
--                  connected with SAVE_TM-Module.
--
-- TELEM:         [ 44   Time-Stamp   29 : 28  Location Anode 22
--                : 21 Location Cathode 15 : 14    Energy    7
--                : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
-- TELEMCLK       : Trigger Signal for writeout from SAVE_TM-Module to
--                  disk-file.
--
-- BSN            : 41 Bit Output Event-Packet for diagnostic, currently
--                  connected with SAVE_BS-Module.
--
-- BSN:           [ 41   Time-Stamp   25 : 24  Location Anode 18
--                : 17 Location Cathode 11 : 10    Energy    3
--                : 2 DOI 0 ]
--
-- BSNCLK         : Trigger Signal for writeout from SAVE_BS-Module to
--                  disk-file.
--
-- TEST           : 46 Bit Output Event-Packet for diagnostic, currently
--                  connected with SAVE_TST-Module.
--
-- TEST:          [ 45   Time-Stamp   30 : 29  Location Anode 23
--                : 22 Location Cathode 16 : 15    Energy    8
--                : 7 DOI 5 : 4 VD-DET : 3 VD-EPP : 2 VD-Noise : 1 ERROR 0 ]
--
-- BURSTCOUNT     : Output for diagnostic, currently connected to
--                  SAVE_BD-Module.
--
-- AVECOUNT       : Count-Rate  Output to SAVE_BD-Module for diagnostics.
--
-- SQR_OUT        : SQR Output to SAVE_BD-Module for diagnostics.
--
-- SIG_OUT        : 5 Sigma Output to SAVE_BD-Module for diagnostics.
--
-- GATECLK        : Trigger Signal for writeout from SAVE_BD-Module to
--                  disk-file.
-- TRIGGER        : Burst Trigger Alert Output, currently connected to
--                  SAVE_BD-Module.
--

--------------------------------------------------------------------------



-- VHDL script starts here:


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


entity EPP_TOP is

        ---
```

```
        -- gfrequ = 1/8 Clock Frequency in Hz
        -- n = Number of HEX-Counters be needed to represent frequency number!
        ---
        -- gfreq = 3000000   -- for a 24 MHz System Clock
        -- n = 7
        ---
        --

        generic ( n             : in positive; -- := 4 --  minimum n=2 !
                  gfreq         : in integer   -- := 1000 -- Clock Frequency
                                                          -- for Testing
                );


        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE           : in std_logic;
                  LOADLUT       : in std_logic;
                  MODE          : in unsigned (1 downto 0);
                  GATETIMESET   : in unsigned (1 downto 0);
                  IN_EVENT_EC   : in unsigned (167 downto 0);
                  RINGB         : out unsigned (40 downto 0);
                  RINGBCLK      : out std_logic;
                  TELEM         : out unsigned (44 downto 0);
                  TELEMCLK      : out std_logic;
                  BSN           : out unsigned (40 downto 0);
                  BSNCLK        : out std_logic;
                  TEST          : out unsigned (45 downto 0);
                  BURSTCOUNT    : out unsigned ((4*n)-1 downto 0);
                  AVECOUNT      : out unsigned ((4*n)-1 downto 0);
                  SQR_OUT       : out unsigned ((4*n)-1 downto 0);
                  SIG_OUT       : out unsigned ((4*n)-1 downto 0);
                  GATECLK       : out std_logic;
                  TRIGGER       : out std_logic
                  );
end EPP_TOP;


architecture Structural of EPP_TOP is


        component EC_UNIT
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE           : in std_logic;
                          LOAD_LUT      : in std_logic;
                          MODE          : in unsigned (1 downto 0);
                          EVENT_IN      : in unsigned (167 downto 0);
                          EVENT_OUT     :out unsigned (44 downto 0)
                          );
        end component;




        component FD_UNIT
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
```

```
                          RNE            : in std_logic;
                          EVENT_IN_FD    : in unsigned (44 downto 0);
                          RINGB          : out unsigned (40 downto 0);
                          RINGBCLK       : out std_logic;
                          TELEM          : out unsigned (44 downto 0);
                          TELEMCLK       : out std_logic;
                          BSN            : out unsigned (40 downto 0);
                          BSNCLK         : out std_logic;
                          TEST           : out unsigned (45 downto 0);
                          BCNT           : out std_logic
                          );
        end component;



        component BD_UNIT
                generic   (n             : in positive;
                           gfreq         : in positive
                          );
                port      (CLK           : in std_logic;
                           RESET         : in std_logic;
                           RNE           : in std_logic;
                           GATE_SWITCH   : in unsigned (1 downto 0);
                           BCNT          : in std_logic;
                           BCNTTST_OUT   : out unsigned((4*n)-1 downto 0);
                           AVTST_OUT     : out unsigned((4*n)-1 downto 0);
                           SQR_OUT       : out unsigned((4*n)-1 downto 0);
                           SIG_OUT       : out unsigned((4*n)-1 downto 0);
                           GATE_CLK      : out std_logic;
                           TRIGGER       : out std_logic
                           );
        end component;


signal EC_EVENT_FD        : unsigned (44 downto 0);
signal FD_BCNT_BD         : std_logic;


begin


        ---
        -- invoking Unit: EC_UNIT (Error Correction Unit)
        ---
        module_ecunit: EC_UNIT
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  LOAD_LUT      => LOADLUT,
                                  MODE          => MODE,
                                  EVENT_IN      => IN_EVENT_EC,
                                  EVENT_OUT     => EC_EVENT_FD
                                  );


        ---
```

```
        -- invoking Unit: FD_UNIT (Filter & Decision Unit)
        ---
        module_fdunit: FD_UNIT
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  EVENT_IN_FD   => EC_EVENT_FD,
                                  RINGB         => RINGB,
                                  RINGBCLK      => RINGBCLK,
                                  TELEM         => TELEM,
                                  TELEMCLK      => TELEMCLK,
                                  BSN           => BSN,
                                  BSNCLK        => BSNCLK,
                                  TEST          => TEST,
                                  BCNT          => FD_BCNT_BD
                                  );



        ---
        -- invoking Unit: BD_UNIT (Burst Detection Unit)
        ---
        module_bdunit: BD_UNIT
                generic map     ( n             => n,
                                  gfreq         => gfreq
                                )
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  GATE_SWITCH   => GATETIMESET,
                                  BCNT          => FD_BCNT_BD,
                                  BCNTTST_OUT   => BURSTCOUNT,
                                  AVTST_OUT     => AVECOUNT,
                                  SQR_OUT       => SQR_OUT,
                                  SIG_OUT       => SIG_OUT,
                                  GATE_CLK      => GATECLK,
                                  TRIGGER       => TRIGGER
                                  );

end Structural;
```

B.2 EC-Unit

```
----------------------------------------------------------------------
--------- Error Correction Unit  , Rev 0.1 , 06.09.00, TS/IAAT --------
----------------------------------------------------------------------

---
-- EC_UNIT / EPP-Core
---

-------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
--
-- RNE            : Pipeline Signal from EPP-TOP. (Testbench, later EPA-Unit)
--
-- LOAD_LUT       : Input signal, to invoke the LUT-Load-Process, from EPP_TOP.
--
-- MODE           : 2 Bit Input signal to switch the processor in different
--                  working-modes. Not implemented yet.
--
-- EVENT_IN       : Main Event-Packet Input Signal (168 Bit) from EPP_TOP
--                  (Testbench, later from EPA-Unit).
--
------
-- Task:  Structural Design to connect the modules of the Error-Correction-Unit
--        with each other, and the Error-Correction-Unit with the TOP-Design
--        of the EPP.
--
--
------
-- Output Signals:
--
-- EVENT_OUT      : Event-Packet Output (45 Bit) after processing in EC-Unit,
--                  to FD-Unit.
--
-------------------------------------------------------------------------


-- VHDL script starts here:




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


entity EC_UNIT is
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE           : in std_logic;
                  LOAD_LUT      : in std_logic;
                  MODE          : in unsigned (1 downto 0);
```

```
                  EVENT_IN      : in unsigned (167 downto 0);
                  EVENT_OUT     : out unsigned (44 downto 0)
                  );
end EC_UNIT;


architecture Structural of EC_UNIT is


        component EC_IN
               port     ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE           : in std_logic;
                          REG_IN        : in unsigned (167 downto 0);
                          RNE_1         : out std_logic;
                          RNE_2         : out std_logic;
                          RNE_3         : out std_logic;
                          RNE_4         : out std_logic;
                          RNE_5         : out std_logic;
                          RNE_6         : out std_logic;
                          RNE_7         : out std_logic;
                          RNE_8         : out std_logic;
                          RNE_9         : out std_logic;
                          AMPL_OUT      : out unsigned (143 downto 0);
                          TIME_OUT      : out unsigned (15 downto 0);
                          SPAR_OUT      : out unsigned (7 downto 0)
                          );
        end component;


        component AC
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE           : in std_logic;
                          RNE_1         : in std_logic;
                          RNE_3         : in std_logic;
                          RNE_4         : in std_logic;
                          RNE_5         : in std_logic;
                          REG_IN        : in unsigned(143 downto 0);
                          LUT_CORR_1    : in unsigned (23 downto 0);
                          LUT_CORR_2    : in unsigned (23 downto 0);
                          LUT_CORR_3    : in unsigned (23 downto 0);
                          LUT_CORR_4    : in unsigned (23 downto 0);
                          LUT_CORR_5    : in unsigned (23 downto 0);
                          LUT_CORR_6    : in unsigned (23 downto 0);
                          ERROR_1_1     : in std_logic;
                          ERROR_1_2     : in std_logic;
                          ERROR_1_3     : in std_logic;
                          ERROR_1_4     : in std_logic;
                          ERROR_1_5     : in std_logic;
                          ERROR_1_6     : in std_logic;
                          LUT_LOAD      : in std_logic;
                          LUT_ADDR_1    : out unsigned (8 downto 0);
                          LUT_ADDR_2    : out unsigned (8 downto 0);
                          LUT_ADDR_3    : out unsigned (8 downto 0);
```

```
                  LUT_ADDR_4    : out unsigned (8 downto 0);
                  LUT_ADDR_5    : out unsigned (8 downto 0);
                  LUT_ADDR_6    : out unsigned (8 downto 0);
                  REG_OUT       : out unsigned(137 downto 0)
                );
end component;



component LUTC
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_2         : in std_logic;
                  LOAD          : in std_logic;
                  LUT_IN_1      : in unsigned (8 downto 0);
                  LUT_IN_2      : in unsigned (8 downto 0);
                  LUT_IN_3      : in unsigned (8 downto 0);
                  LUT_IN_4      : in unsigned (8 downto 0);
                  LUT_IN_5      : in unsigned (8 downto 0);
                  LUT_IN_6      : in unsigned (8 downto 0);
                  LUT_LOAD      : out std_logic;
                  ERROR_1_1     : out std_logic;
                  ERROR_1_2     : out std_logic;
                  ERROR_1_3     : out std_logic;
                  ERROR_1_4     : out std_logic;
                  ERROR_1_5     : out std_logic;
                  ERROR_1_6     : out std_logic;
                  LUT_OUT_1     : out unsigned (23 downto 0);
                  LUT_OUT_2     : out unsigned (23 downto 0);
                  LUT_OUT_3     : out unsigned (23 downto 0);
                  LUT_OUT_4     : out unsigned (23 downto 0);
                  LUT_OUT_5     : out unsigned (23 downto 0);
                  LUT_OUT_6     : out unsigned (23 downto 0)
                );
end component;


component EL
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_6         : in std_logic;
                  REG_IN        : in unsigned (137 downto 0);
                  LOC_A         : out unsigned (6 downto 0);
                  LOC_C         : out unsigned (6 downto 0);
                  VD_EL_A       : out std_logic;
                  VD_EL_C       : out std_logic
                );
end component;


component SOA
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_6         : in std_logic;
                  REG_IN        : in unsigned (137 downto 0);
                  REG_OUT       : out unsigned (15 downto 0);
                  VD_SOA        : out std_logic
                );
```

```
end component;


component SOC
        port    ( CLK: in std_logic;
                  RESET          : in std_logic;
                  RNE_6          : in std_logic;
                  REG_IN         : in unsigned (137 downto 0);
                  REG_OUT        : out unsigned (15 downto 0);
                  VD_SOC         : out std_logic
                );
end component;


component DIV
        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  RNE_7          : in std_logic;
                  REG_SOA        : in unsigned (15 downto 0);
                  REG_SOC        : in unsigned (15 downto 0);
                  DOI_FULL       : out unsigned (7 downto 0);
                  DOI            : out unsigned (2 downto 0);
                  VD_DIV         : out std_logic
                );
end component;


component DC
        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  RNE_8          : in std_logic;
                  SOA_IN         : in unsigned (15 downto 0);
                  DOI_FULL       : in unsigned (7 downto 0);
                  ENERGY         : out unsigned (7 downto 0);
                  VD_DC          : out std_logic
                );
end component;


component TC
        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  RNE_1          : in std_logic;
                  REG_IN         : in unsigned (15 downto 0);
                  REG_OUT        : out unsigned (15 downto 0);
                  VD_TC          : out std_logic
                  );
end component;


component ED
        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  AC_IN          : in unsigned (137 downto 0);
                  VD_ELA         : in std_logic;
                  VD_ELC         : in std_logic;
                  VD_SOA         : in std_logic;
                  VD_SOC         : in std_logic;
```

```
                  VD_DIV        : in std_logic;
                  VD_TC         : in std_logic;
                  VD_DC         : in std_logic;
                  VD_DET        : out std_logic;
                  VD_EPP        : out std_logic
                 );
end component;



component EC_OUT
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_9         : in std_logic;
                  TIME_IN       : in unsigned (15 downto 0);
                  LOC_A_IN      : in unsigned (6 downto 0);
                  LOC_C_IN      : in unsigned (6 downto 0);
                  ENERGY_IN     : in unsigned (7 downto 0);
                  DOI_IN        : in unsigned (2 downto 0);
                  VD_DET        : in std_logic;
                  VD_EPP        : in std_logic;
                  REG_OUT       : out unsigned (44 downto 0)
                );
end component;





signal RNE_1             :      std_logic;
signal RNE_2             :      std_logic;
signal RNE_3             :      std_logic;
signal RNE_4             :      std_logic;
signal RNE_5             :      std_logic;
signal RNE_6             :      std_logic;
signal RNE_7             :      std_logic;
signal RNE_8             :      std_logic;
signal RNE_9             :      std_logic;
signal LUTC_CORR_1_AC    :      unsigned (23 downto 0);
signal LUTC_CORR_2_AC    :      unsigned (23 downto 0);
signal LUTC_CORR_3_AC    :      unsigned (23 downto 0);
signal LUTC_CORR_4_AC    :      unsigned (23 downto 0);
signal LUTC_CORR_5_AC    :      unsigned (23 downto 0);
signal LUTC_CORR_6_AC    :      unsigned (23 downto 0);
signal AC_ADDR_1_LUTC    :      unsigned ( 8 downto 0);
signal AC_ADDR_2_LUTC    :      unsigned ( 8 downto 0);
signal AC_ADDR_3_LUTC    :      unsigned ( 8 downto 0);
signal AC_ADDR_4_LUTC    :      unsigned ( 8 downto 0);
signal AC_ADDR_5_LUTC    :      unsigned ( 8 downto 0);
signal AC_ADDR_6_LUTC    :      unsigned ( 8 downto 0);
signal LUTC_ERROR_1_1_AC :      std_logic;
signal LUTC_ERROR_1_2_AC :      std_logic;
signal LUTC_ERROR_1_3_AC :      std_logic;
signal LUTC_ERROR_1_4_AC :      std_logic;
signal LUTC_ERROR_1_5_AC :      std_logic;
signal LUTC_ERROR_1_6_AC :      std_logic;
signal LUTC_LUT_LOAD_AC  :      std_logic;
```

```
signal AC_REG_OUT        :      unsigned (137 downto 0);
signal EL_LOC_A          :      unsigned (6 downto 0);
signal EL_LOC_C          :      unsigned (6 downto 0);
signal SOA_OUT           :      unsigned (15 downto 0);
signal SOC_OUT           :      unsigned (15 downto 0);
signal DIV_DOI_DC        :      unsigned (7 downto 0);
signal DIV_SDOI_ECOUT    :      unsigned (2 downto 0);
signal DC_ENERGY_ECOUT   :      unsigned (7 downto 0);
signal TC_TIMEP_ECOUT    :      unsigned (15 downto 0);
signal ECIN_TIMEP_TC     :      unsigned (15 downto 0);
signal ECIN_AMPLP_AC     :      unsigned (143 downto 0);
signal ECIN_SPARP_ECOUT  :      unsigned (7 downto 0);
signal EL_VDA_ED         :      std_logic;
signal EL_VDC_ED         :      std_logic;
signal SOA_VD_ED         :      std_logic;
signal SOC_VD_ED         :      std_logic;
signal DIV_VD_ED         :      std_logic;
signal DC_VD_ED          :      std_logic;
signal TC_VD_ED          :      std_logic;
signal ED_VDDET_ECOUT    :      std_logic;
signal ED_VDEPP_ECOUT    :      std_logic;


begin



        ---
        -- invoking Module: EC_IN (Input Controller)
        ---

        module_ecin: EC_IN
                 port map       ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  REG_IN        => EVENT_IN,
                                  RNE_1         => RNE_1,
                                  RNE_2         => RNE_2,
                                  RNE_3         => RNE_3,
                                  RNE_4         => RNE_4,
                                  RNE_5         => RNE_5,
                                  RNE_6         => RNE_6,
                                  RNE_7         => RNE_7,
                                  RNE_8         => RNE_8,
                                  RNE_9         => RNE_9,
                                  AMPL_OUT      => ECIN_AMPLP_AC,
                                  TIME_OUT      => ECIN_TIMEP_TC,
                                  SPAR_OUT      => ECIN_SPARP_ECOUT
                                );



        ---
        -- invoking Module: AC (Amplitude Correction)
        ---

        module_ac: AC
```

```
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE           => RNE,
                          RNE_1         => RNE_1,
                          RNE_3         => RNE_3,
                          RNE_4         => RNE_4,
                          RNE_5         => RNE_5,
                          REG_IN        => ECIN_AMPLP_AC,
                          LUT_CORR_1    => LUTC_CORR_1_AC,
                          LUT_CORR_2    => LUTC_CORR_2_AC,
                          LUT_CORR_3    => LUTC_CORR_3_AC,
                          LUT_CORR_4    => LUTC_CORR_4_AC,
                          LUT_CORR_5    => LUTC_CORR_5_AC,
                          LUT_CORR_6    => LUTC_CORR_6_AC,
                          ERROR_1_1     => LUTC_ERROR_1_1_AC,
                          ERROR_1_2     => LUTC_ERROR_1_2_AC,
                          ERROR_1_3     => LUTC_ERROR_1_3_AC,
                          ERROR_1_4     => LUTC_ERROR_1_4_AC,
                          ERROR_1_5     => LUTC_ERROR_1_5_AC,
                          ERROR_1_6     => LUTC_ERROR_1_6_AC,
                          LUT_LOAD      => LUTC_LUT_LOAD_AC,
                          LUT_ADDR_1    => AC_ADDR_1_LUTC,
                          LUT_ADDR_2    => AC_ADDR_2_LUTC,
                          LUT_ADDR_3    => AC_ADDR_3_LUTC,
                          LUT_ADDR_4    => AC_ADDR_4_LUTC,
                          LUT_ADDR_5    => AC_ADDR_5_LUTC,
                          LUT_ADDR_6    => AC_ADDR_6_LUTC,
                          REG_OUT       => AC_REG_OUT
                        );


---
-- invoking Module: LUTC (Look Up Table Controller)
---

module_lutc: LUTC
         port map       ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_2         => RNE_2,
                          LOAD          => LOAD_LUT,
                          LUT_IN_1      => AC_ADDR_1_LUTC,
                          LUT_IN_2      => AC_ADDR_2_LUTC,
                          LUT_IN_3      => AC_ADDR_3_LUTC,
                          LUT_IN_4      => AC_ADDR_4_LUTC,
                          LUT_IN_5      => AC_ADDR_5_LUTC,
                          LUT_IN_6      => AC_ADDR_6_LUTC,
                          LUT_LOAD      => LUTC_LUT_LOAD_AC,
                          ERROR_1_1     => LUTC_ERROR_1_1_AC,
                          ERROR_1_2     => LUTC_ERROR_1_2_AC,
                          ERROR_1_3     => LUTC_ERROR_1_3_AC,
                          ERROR_1_4     => LUTC_ERROR_1_4_AC,
                          ERROR_1_5     => LUTC_ERROR_1_5_AC,
                          ERROR_1_6     => LUTC_ERROR_1_6_AC,
                          LUT_OUT_1     => LUTC_CORR_1_AC,
                          LUT_OUT_2     => LUTC_CORR_2_AC,
                          LUT_OUT_3     => LUTC_CORR_3_AC,
                          LUT_OUT_4     => LUTC_CORR_4_AC,
                          LUT_OUT_5     => LUTC_CORR_5_AC,
```

```
                        LUT_OUT_6      => LUTC_CORR_6_AC
                      );


--- 
-- invoking Module: EL (Event Location)
--- 

module_el: EL
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_6         => RNE_6,
                          REG_IN        => AC_REG_OUT,
                          LOC_A         => EL_LOC_A,
                          LOC_C         => EL_LOC_C,
                          VD_EL_A       => EL_VDA_ED,
                          VD_EL_C       => EL_VDC_ED
                        );



--- 
-- invoking Module: SOA (Sum Of Anode)
--- 

module_soa: SOA
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_6         => RNE_6,
                          REG_IN        => AC_REG_OUT,
                          REG_OUT       => SOA_OUT,
                          VD_SOA        => SOA_VD_ED
                        );


--- 
-- invoking Module: SOC (Sum Of Cathode)
--- 

module_soc: SOC
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_6         => RNE_6,
                          REG_IN        => AC_REG_OUT,
                          REG_OUT       => SOC_OUT,
                          VD_SOC        => SOC_VD_ED
                        );


--- 
-- invoking Module: DIV (Divider SOC/SOA)
--- 

module_div: DIV
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_7         => RNE_7,
                          REG_SOA       => SOA_OUT,
```

```
                          REG_SOC       => SOC_OUT,
                          DOI_FULL      => DIV_DOI_DC,
                          DOI           => DIV_SDOI_ECOUT,
                          VD_DIV        => DIV_VD_ED
                          );


---
-- invoking Module: DC (Depth Correction)
---

module_dc: DC
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_8         => RNE_8,
                          SOA_IN        => SOA_OUT,
                          DOI_FULL      => DIV_DOI_DC,
                          ENERGY        => DC_ENERGY_ECOUT,
                          VD_DC         => DC_VD_ED
                        );



---
-- invoking Module: TC (Time Correction)
---

module_tc: TC
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          RNE_1         => RNE_1,
                          REG_IN        => ECIN_TIMEP_TC,
                          REG_OUT       => TC_TIMEP_ECOUT,
                          VD_TC         => TC_VD_ED
                        );


---
-- invoking Module: ED (Error Detector)
---

module_ed: ED
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          AC_IN         => AC_REG_OUT,
                          VD_ELA        => EL_VDA_ED,
                          VD_ELC        => EL_VDC_ED,
                          VD_SOA        => SOA_VD_ED,
                          VD_SOC        => SOC_VD_ED,
                          VD_DIV        => DIV_VD_ED,
                          VD_TC         => TC_VD_ED,
                          VD_DC         => DC_VD_ED,
                          VD_DET        => ED_VDDET_ECOUT,
                          VD_EPP        => ED_VDEPP_ECOUT
                          );

---
-- invoking Module: EC_OUT (Output Controller)
```

```
        ---

        module_ecout: EC_OUT
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE_9         => RNE_9,
                                  TIME_IN       => TC_TIMEP_ECOUT,
                                  LOC_A_IN      => EL_LOC_A,
                                  LOC_C_IN      => EL_LOC_C,
                                  ENERGY_IN     => DC_ENERGY_ECOUT,
                                  DOI_IN        => DIV_SDOI_ECOUT,
                                  VD_DET        => ED_VDDET_ECOUT,
                                  VD_EPP        => ED_VDEPP_ECOUT,
                                  REG_OUT       => EVENT_OUT
                                  );


end Structural;
```

B.2.1 EC_IN-Module

```
---------------------------------------------------------------------------
-- Input Register & DataPacket Disassembler, Rel 0.1, 12.09.00, TS/IAAT --
---------------------------------------------------------------------------

---
-- EC_IN / EC-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
--
-- RNE           : Pipeline Tag from EPA_UNIT (Testbench)
--
-- REG_IN        : 168 Bit Event-Packet from EPA_UNIT (Testbench)
--
-- REG_IN:
--           [ 167       Timestamp      151 : 150   Spare   144 ]
-- Hit 1     [ 143          130 : 129        121 :     120      ]
-- Hit 2     [ 119          106 : 105         97 :      96      ]
-- Hit 3     [  95           82 :  81         73 :      72      ]
-- Hit 4     [  71           58 :  57         49 :      48      ]
-- Hit 5     [  47           34 :  33         25 :      24      ]
-- Hit 6     [  23           10 :  9           1 :       0      ]
--
------
-- Task: Split the Input Event-Packet into 3 parts: Timestamp, Spare and
--       the Event-Packet Remnant (AMPL_OUT), synchron with RNE-Signal.
--       Generate the Sub-RNE Signals for Sequential Control of the EC-Unit
--       from the RNE-Signal.
--
------
-- Output Signals:
--
-- AMPL_OUT       : Output Event-Packet Remnant
--
-- AMPL_OUT: [ 14 Bit Amplitude : 9 Bit Location : 1 Bit VD-Det ]
--
-- Hit 1     [ 143          130 : 129        121 :     120      ]
-- Hit 2     [ 119          106 : 105         97 :      96      ]
-- Hit 3     [  95           82 :  81         73 :      72      ]
-- Hit 4     [  71           58 :  57         49 :      48      ]
-- Hit 5     [  47           34 :  33         25 :      24      ]
-- Hit 6     [  23           10 :  9           1 :       0      ]
--
-- TIME_OUT       : Timestamp Output for TC-Module.
--
-- SPAR_OUT       : Spare Output (not used yet)
--
-- RNE_1 - RNE_9  : Sub-RNE Output for Sequential Control
--
---------------------------------------------------------------------------
```

```
-- VHDL script starts here:


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


entity EC_IN is

        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE           : in std_logic;
                  REG_IN        : in unsigned (167 downto 0);
                  RNE_1         : out std_logic;
                  RNE_2         : out std_logic;
                  RNE_3         : out std_logic;
                  RNE_4         : out std_logic;
                  RNE_5         : out std_logic;
                  RNE_6         : out std_logic;
                  RNE_7         : out std_logic;
                  RNE_8         : out std_logic;
                  RNE_9         : out std_logic;
                  AMPL_OUT      : out unsigned (143 downto 0);
                  TIME_OUT      : out unsigned (15 downto 0);
                  SPAR_OUT      : out unsigned (7 downto 0)
                );
end EC_IN;


architecture inside of EC_IN is


begin

         --------------------------
         -- Disassemble Data Packet
         --------------------------
         disassemble:
         process (CLK)

                variable last_RNE        : std_logic;


         -- begin of process
         begin
                if rising_edge (CLK) then
                        if RESET = '1' then
                                ---
```

```
                        -- Reset Signals and Variables
                        ---
                        last_RNE := '0';
                        TIME_OUT <= (others => '0');
                        SPAR_OUT <= (others => '0');
                        AMPL_OUT <= (others => '0');
                else
                        ---
                        -- Load Event-Packet from Input
                        ---
                        if RNE = '1' and last_RNE ='0' then
                                 TIME_OUT <= REG_IN(167 downto 152);
                                 SPAR_OUT <= REG_IN(151 downto 144);
                                 AMPL_OUT <= REG_IN(143 downto 0);
                        end if;
                        last_RNE := RNE;
                end if;
        end if;
end process;

--------------------------
-- Generating Sub-RNEs
--------------------------
sub_rne:
process (CLK)

        variable CNT            : integer range 0 to 127;
        variable last_RNE       : std_logic;
begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        RNE_1 <= '0';
                        RNE_2 <= '0';
                        RNE_3 <= '0';
                        RNE_4 <= '0';
                        RNE_5 <= '0';
                        RNE_6 <= '0';
                        RNE_7 <= '0';
                        RNE_8 <= '0';
                        RNE_9 <= '0';
                        CNT := 0;
                else
                        if RNE = '1' then
                                CNT := 0;
                                RNE_1 <= '0';
                                RNE_2 <= '0';
                                RNE_3 <= '0';
                                RNE_4 <= '0';
                                RNE_5 <= '0';
                                RNE_6 <= '0';
                                RNE_7 <= '0';
                                RNE_8 <= '0';
                                RNE_9 <= '0';
                        else
                                if CNT < 100 then
                                        CNT := CNT + 1;
                                        if CNT = 10 then
                                                RNE_1 <= '1';
```

```
                                                else
                                                        RNE_1<= '0';
                                                end if;
                                                if CNT = 20 then
                                                        RNE_2 <= '1';
                                                else
                                                        RNE_2 <= '0';
                                                end if;
                                                if CNT = 30 then
                                                        RNE_3 <= '1';
                                                else
                                                        RNE_3 <= '0';
                                                end if;
                                                if CNT = 40 then
                                                        RNE_4 <= '1';
                                                else
                                                        RNE_4 <= '0';
                                                end if;
                                                if CNT = 50 then
                                                        RNE_5 <= '1';
                                                else
                                                        RNE_5 <= '0';
                                                end if;
                                                if CNT = 60 then
                                                        RNE_6 <= '1';
                                                else
                                                        RNE_6 <= '0';
                                                end if;
                                                if CNT = 70 then
                                                        RNE_7 <= '1';
                                                else
                                                        RNE_7 <= '0';
                                                end if;
                                                if CNT = 80 then
                                                        RNE_8 <= '1';
                                                else
                                                        RNE_8 <= '0';
                                                end if;
                                                if CNT = 90 then
                                                        RNE_9 <= '1';
                                                else
                                                        RNE_9 <= '0';
                                                end if;
                                        end if;
                                end if;
                        end if;
                end if;
        end process;
end inside;
```

B.2.2 AC-Module

```
---------------------------------------------------------------------------
-------  Amplitude Correction  Module , Rev 0.6,  10.10.00,  TS/IAAT ------
---------------------------------------------------------------------------

---
-- AC (Amplitude Correction) / EC-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
--
-- RNE           : Pipeline Tag from EPA_UNIT (Testbench)
--
-- RNE_1 - RNE_5 : Sub-RNE-Signals for Sequence Control from EC_IN
--
-- REG_IN        : Input Event-Packet Remnant from EPA_UNIT (Testbench)
--
--
-- REG_IN:   [ 14 Bit Amplitude : 9 LUT Location : 1 Bit VD Det ]
--
-- Hit 1     [ 143          130 : 129        121 :     120      ]
-- Hit 2     [ 119          106 : 105         97 :      96      ]
-- Hit 3     [  95           82 :  81         73 :      72      ]
-- Hit 4     [  71           58 :  57         49 :      48      ]
-- Hit 5     [  47           34 :  33         25 :      24      ]
-- Hit 6     [  23           10 :  9           1 :       0      ]
--
--
-- LUT_CORR1/6   : LUT Entry for Amplitude Correction from LUTC
--
-- ERROR1/6      : Error Flags from LUTC
--
-- LUT_LOAD      : Flag from LUTC, Set to 1 during LUT loading
--
------
-- Task: Amplitude Correction of Energy-Entrys of the Event-Packet Remnant.
--       1.) RNE_1: Generate Hit dependent LUT-Address and apply to LUTC-Module.
--                  Divide Event-Packet Remnant into Hit dependent variables
--                  ERG_x_i, LOC_x, LUT_ADDR_x.
--       2.) RNE_3: Get the 24 Bit Correction-Word (LUT_CORR_x) from LUTC.
--       3.) RNE_4: Divide the 24 Bit Words in three 8 Bit Correction-Words
--                  (VAL_x_A, VAL_x_B, VAL_x_B) and correct the amplitude
--                  Information ERG_x_i to ERG_x for each of the 6 Hits.
--       4.) RNE_5: Compound the corrected Words to the Output-Event-Packet
--                  Remnant REG_OUT. Event-Packets with Energy Zero override
--                  the correction process.
--       During LUT-Load, Event-Packets were vetoed by using the VD_LUT
--       flag to set the VD_O_x-Flags to Zero.
--
------
-- Output Signals:
--
```

```
-- LUT_ADDR_1/6  : LUT Addresses to LUTC
--
-- REG_OUT       : Output Event-Packet Remnant
--
--
-- REG_OUT:  [ 14 Bit Amplitude : 7 Bit Location : 1 Bit VD-Det : AC-Det ]
--
-- Hit 1     [ 137          124 : 123        117 :     116      :   115  ]
-- Hit 2     [ 114          101 : 100         94 :      93      :    92  ]
-- Hit 3     [  91           78 :  77         71 :      70      :    69  ]
-- Hit 4     [  68           55 :  54         48 :      47      :    46  ]
-- Hit 5     [  45           32 :  31         25 :      24      :    23  ]
-- Hit 6     [  22            9 :   8          2 :       1      :     0  ]
--
---------------------------------------------------------------------------


-- VHDL script starts here:


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


library STD;
use STD.textio.all;
entity AC is


        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE           : in std_logic;
                  RNE_1         : in std_logic;
                  RNE_3         : in std_logic;
                  RNE_4         : in std_logic;
                  RNE_5         : in std_logic;
                  REG_IN        : in unsigned (143 downto 0);
                  LUT_CORR_1    : in unsigned (23 downto 0);
                  LUT_CORR_2    : in unsigned (23 downto 0);
                  LUT_CORR_3    : in unsigned (23 downto 0);
                  LUT_CORR_4    : in unsigned (23 downto 0);
                  LUT_CORR_5    : in unsigned (23 downto 0);
                  LUT_CORR_6    : in unsigned (23 downto 0);
                  ERROR_1_1     : in std_logic;
                  ERROR_1_2     : in std_logic;
                  ERROR_1_3     : in std_logic;
                  ERROR_1_4     : in std_logic;
                  ERROR_1_5     : in std_logic;
                  ERROR_1_6     : in std_logic;
                  LUT_LOAD      : in std_logic;
                  LUT_ADDR_1    : out unsigned (8 downto 0);
                  LUT_ADDR_2    : out unsigned (8 downto 0);
                  LUT_ADDR_3    : out unsigned (8 downto 0);
                  LUT_ADDR_4    : out unsigned (8 downto 0);
                  LUT_ADDR_5    : out unsigned (8 downto 0);
                  LUT_ADDR_6    : out unsigned (8 downto 0);
```

```
                  REG_OUT       : out unsigned (137 downto 0)
                );
end AC;



architecture inside of AC is



signal LUT_OP_1     : unsigned (23 downto 0);
signal LUT_OP_2     : unsigned (23 downto 0);
signal LUT_OP_3     : unsigned (23 downto 0);
signal LUT_OP_4     : unsigned (23 downto 0);
signal LUT_OP_5     : unsigned (23 downto 0);
signal LUT_OP_6     : unsigned (23 downto 0);
signal VD_O_1       : std_logic;
signal VD_O_2       : std_logic;
signal VD_O_3       : std_logic;
signal VD_O_4       : std_logic;
signal VD_O_5       : std_logic;
signal VD_O_6       : std_logic;
signal VD_LUT       : std_logic;
signal ERG_1_i      : unsigned (13 downto 0);
signal ERG_2_i      : unsigned (13 downto 0);
signal ERG_3_i      : unsigned (13 downto 0);
signal ERG_4_i      : unsigned (13 downto 0);
signal ERG_5_i      : unsigned (13 downto 0);
signal ERG_6_i      : unsigned (13 downto 0);
signal ERG_1        : unsigned (13 downto 0);
signal ERG_2        : unsigned (13 downto 0);
signal ERG_3        : unsigned (13 downto 0);
signal ERG_4        : unsigned (13 downto 0);
signal ERG_5        : unsigned (13 downto 0);
signal ERG_6        : unsigned (13 downto 0);


begin



        -----------------------
        -- read LUT Addresses
        -----------------------
        read_LUT_Addr:
        process (CLK)

                variable last_RNE_1     : std_logic;
                variable last_RNE_3     : std_logic;
                variable LOP_1          : unsigned (8 downto 0);
                variable LOP_2          : unsigned (8 downto 0);
                variable LOP_3          : unsigned (8 downto 0);
                variable LOP_4          : unsigned (8 downto 0);
                variable LOP_5          : unsigned (8 downto 0);
                variable LOP_6          : unsigned (8 downto 0);
                variable LA_1           : integer range 0 to 255;
                variable LA_2           : integer range 0 to 255;
                variable LA_3           : integer range 0 to 255;
```

```
        variable LA_4           : integer range 0 to 255;
        variable LA_5           : integer range 0 to 255;
        variable LA_6           : integer range 0 to 255;


-- begin of process
begin
        if rising_edge (CLK) then
                if RESET = '1' then

                        ---
                        -- Reset Signals and Variables
                        ---
                        LUT_OP_1        <= (others => '0');
                        LUT_OP_2        <= (others => '0');
                        LUT_OP_3        <= (others => '0');
                        LUT_OP_4        <= (others => '0');
                        LUT_OP_5        <= (others => '0');
                        LUT_OP_6        <= (others => '0');
                        LUT_ADDR_1      <= (others => '0');
                        LUT_ADDR_2      <= (others => '0');
                        LUT_ADDR_3      <= (others => '0');
                        LUT_ADDR_4      <= (others => '0');
                        LUT_ADDR_5      <= (others => '0');
                        LUT_ADDR_6      <= (others => '0');
                        ERG_1_i         <= (others => '0');
                        ERG_2_i         <= (others => '0');
                        ERG_3_i         <= (others => '0');
                        ERG_4_i         <= (others => '0');
                        ERG_5_i         <= (others => '0');
                        ERG_6_i         <= (others => '0');
                        LOP_1           := (others => '0');
                        LOP_2           := (others => '0');
                        LOP_3           := (others => '0');
                        LOP_4           := (others => '0');
                        LOP_5           := (others => '0');
                        LOP_6           := (others => '0');
                        last_RNE_1      := '0';
                        last_RNE_3      := '0';

                else


                        ---------------------------------------
                        -- Generate Hit-dependent LUT Addresses
                        ---------------------------------------

                        ---
                        -- ERG_1_i  :  Hit Energy
                        -- LOC_1    :  Hit Position ID
                        -- LUT_ADDR :  set LUT Address
                        ---

                        if RNE_1 = '1' and last_RNE_1 = '0' then


                               ERG_1_i <= REG_IN(143 downto 130);
                               LOP_1 := REG_IN(129 downto 121);
```

```
                                        LUT_ADDR_1 <= LOP_1;


                                        ERG_2_i <= REG_IN(119 downto 106);
                                        LOP_2 := REG_IN(105 downto 97);
                                        LUT_ADDR_2 <= LOP_2;


                                        ERG_3_i <= REG_IN(95 downto 82);
                                        LOP_3 := REG_IN(81 downto 73);
                                        LUT_ADDR_3 <= LOP_3;


                                        ERG_4_i <= REG_IN(71 downto 58);
                                        LOP_4 := REG_IN(57 downto 49);
                                        LUT_ADDR_4 <= LOP_4;


                                        ERG_5_i <= REG_IN(47 downto 34);
                                        LOP_5 := REG_IN(33 downto 25);
                                        LUT_ADDR_5 <= LOP_5;


                                        ERG_6_i <= REG_IN(23 downto 10);
                                        LOP_6 := REG_IN(9 downto 1);
                                        LUT_ADDR_6 <= LOP_6;



                                 end if;
                                 last_RNE_1 := RNE_1;



                                 -----------------------------------
                                 -- Load Hit dependent Data from LUT
                                 -----------------------------------
                                 if RNE_3 = '1' and last_RNE_3 = '0' then
                                         LUT_OP_1 <= LUT_CORR_1;
                                         LUT_OP_2 <= LUT_CORR_2;
                                         LUT_OP_3 <= LUT_CORR_3;
                                         LUT_OP_4 <= LUT_CORR_4;
                                         LUT_OP_5 <= LUT_CORR_5;
                                         LUT_OP_6 <= LUT_CORR_6;
                                 end if;
                                 last_RNE_3 := RNE_3;


                         end if;
                 end if;


         ---
         -- There are 6 Signals (LUT_OP-1 -> LUT_OP_6) that contain the
         -- 3 correction values for each of the 6 Hits!
         ---
```

```
end process;




---------------------------------
-- Data not valid during LUT Load
---------------------------------
data_not_vd:
process(RNE,CLK)
begin
        if RESET = '1' then
                VD_LUT <= '0';
        else
                if falling_edge(RNE) then
                        if LUT_LOAD = '0' then
                                VD_LUT <= '1';
                        else
                                VD_LUT <= '0';
                        end if;
                end if;
        end if;
end process;



-----------------------------
-- Energy Correction of Hit 1
-----------------------------
ac_hit1:
process(CLK)
   variable Q_ERG_1        : unsigned (27 downto 0);
   variable VAL_1_A        : unsigned (7 downto 0);
   variable VAL_1_B        : unsigned (7 downto 0);
   variable VAL_1_C        : unsigned (7 downto 0);
   variable C_1            : unsigned (10 downto 0);
   variable A_1            : unsigned (14 downto 0);
   variable REG_1_Q1       : unsigned (38 downto 0);
   variable REG_1_Q2       : unsigned (14 downto 0);
   variable REG_1_L        : unsigned (21 downto 0);
   variable REG_1_O        : unsigned (14 downto 0);
   variable COR_1_L        : unsigned (14 downto 0);
   variable COR_1_Q        : unsigned (14 downto 0);
   variable COR_1_LQ       : unsigned (14 downto 0);
   variable COR_1_LQO      : unsigned (13 downto 0);
   variable OVERFLOW_Q     : std_logic;
   variable OVERFLOW_L     : std_logic;
   variable OVERFLOW_O     : std_logic;
   variable last_RNE_4     : std_logic;
begin
   if rising_edge(CLK) then
       if RESET = '1' then
            Q_ERG_1    := (others => '0');
            VAL_1_A    := (others => '0');
            VAL_1_B    := (others => '0');
            VAL_1_C    := (others => '0');
            C_1        := (others => '0');
```

```
            A_1         := (others => '0');
            REG_1_Q1    := (others => '0');
            REG_1_Q2    := (others => '0');
            REG_1_L     := (others => '0');
            REG_1_O     := (others => '0');
            COR_1_L     := (others => '0');
            COR_1_LQ    := (others => '0');
            COR_1_LQO   := (others => '0');
            ERG_1       <= (others => '0');
            OVERFLOW_Q := '0';
            OVERFLOW_L := '0';
            OVERFLOW_O := '0';
            VD_O_1      <= '0';
            last_RNE_4 := '0';

     else

            if RNE_4 = '1' and last_RNE_4 = '0' then
                   VAL_1_A := LUT_OP_1(23 downto 16);
                   VAL_1_B := LUT_OP_1(15 downto 8);
                   VAL_1_C := LUT_OP_1(7  downto 0);



                   -----------------------
                   -- Linear correction --
                   -----------------------
                   REG_1_L              := VAL_1_B * ERG_1_i;
                   COR_1_L(13 downto 0) := REG_1_L(20 downto 7);

                   if REG_1_L(21) = '1' then
                           OVERFLOW_L := '1';
                   else
                           OVERFLOW_L := '0';
                   end if;
                   -----------------------



                   --------------------------
                   -- Quadratic correction --
                   --------------------------
                   Q_ERG_1          := ERG_1_i * ERG_1_i;
                   C_1              := (others => '0');
                   C_1(6 downto 0)  := VAL_1_C(6 downto 0);
                   REG_1_Q1         := (others => '0');
                   REG_1_Q2         := (others => '0');
                   REG_1_Q1         := C_1 * Q_ERG_1;

                   if VAL_1_C(7) = '0' then
                           REG_1_Q2 := COR_1_L + REG_1_Q1(38 downto 25);
                   else
                           REG_1_Q2 := COR_1_L - REG_1_Q1(38 downto 25);
                   end if;
                   COR_1_LQ := REG_1_Q2;

                   if REG_1_Q2(14) = '1' then
                           OVERFLOW_Q := '1';
```

```
                        else
                                OVERFLOW_Q := '0';
                        end if;
                        --------------------------



                        -----------------------
                        -- Offset correction --
                        -----------------------
                        REG_1_O    := (others => '0');
                        A_1        := (others => '0');

                        if  VAL_1_A(7) = '0' then
                                A_1(10 downto 4) := VAL_1_A(6 downto 0);
                                REG_1_O := COR_1_LQ + A_1;
                        else
                                A_1(10 downto 4) := VAL_1_A(6 downto 0);
                                REG_1_O := COR_1_LQ - A_1;
                        end if;

                        COR_1_LQO := REG_1_O(13 downto 0);

                        if REG_1_O(14) = '1' then
                                OVERFLOW_O := '1';
                        else
                                OVERFLOW_O := '0';
                        end if;
                        ------------------------



                        ------------
                        -- Result --
                        ------------
                        ERG_1 <= COR_1_LQO;
                        ------------



                        ----------------------
                        -- Error processing --
                        --- -----------------
                        VD_O_1 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_1) and VD_LUT;
                        ----------------------

                    end if;
                    last_RNE_4 := RNE_4;
                end if;
            end if;
        end process;



        -----------------------------
        -- Energy Correction of Hit 2
```

```
-----------------------------
ac_hit2:
process(CLK)
   variable Q_ERG_2        : unsigned (27 downto 0);
   variable VAL_2_A        : unsigned (7 downto 0);
   variable VAL_2_B        : unsigned (7 downto 0);
   variable VAL_2_C        : unsigned (7 downto 0);
   variable C_2            : unsigned (10 downto 0);
   variable A_2            : unsigned (14 downto 0);
   variable REG_2_Q1       : unsigned (38 downto 0);
   variable REG_2_Q2       : unsigned (14 downto 0);
   variable REG_2_L        : unsigned (21 downto 0);
   variable REG_2_O        : unsigned (14 downto 0);
   variable COR_2_L        : unsigned (14 downto 0);
   variable COR_2_Q        : unsigned (14 downto 0);
   variable COR_2_LQ       : unsigned (14 downto 0);
   variable COR_2_LQO      : unsigned (13 downto 0);
   variable OVERFLOW_Q     : std_logic;
   variable OVERFLOW_L     : std_logic;
   variable OVERFLOW_O     : std_logic;
   variable last_RNE_4     : std_logic;

begin

   if rising_edge(CLK) then
       if RESET = '1' then
           Q_ERG_2    := (others => '0');
           VAL_2_A    := (others => '0');
           VAL_2_B    := (others => '0');
           VAL_2_C    := (others => '0');
           C_2        := (others => '0');
           A_2        := (others => '0');
           REG_2_Q1   := (others => '0');
           REG_2_Q2   := (others => '0');
           REG_2_L    := (others => '0');
           REG_2_O    := (others => '0');
           COR_2_L    := (others => '0');
           COR_2_LQ   := (others => '0');
           COR_2_LQO  := (others => '0');
           ERG_2      <= (others => '0');
           OVERFLOW_Q := '0';
           OVERFLOW_L := '0';
           OVERFLOW_O := '0';
           VD_O_2     <= '0';
           last_RNE_4 := '0';

       else

           if RNE_4 = '1' and last_RNE_4 = '0' then
                   VAL_2_A := LUT_OP_2(23 downto 16);
                   VAL_2_B := LUT_OP_2(15 downto 8);
                   VAL_2_C := LUT_OP_2(7  downto 0);



                   -----------------------
                   -- Linear correction --
```

```
----------------------
REG_2_L              := VAL_2_B * ERG_2_i;
COR_2_L(13 downto 0) := REG_2_L(20 downto 7);

if REG_2_L(21) = '1' then
        OVERFLOW_L := '1';
else
        OVERFLOW_L := '0';
end if;
---------------------------------------
```

```
--------------------------
-- Quadratic correction --
--------------------------
Q_ERG_2          := ERG_2_i * ERG_2_i;
C_2              := (others => '0');
C_2(6 downto 0)  := VAL_2_C(6 downto 0);
REG_2_Q1         := (others => '0');
REG_2_Q2         := (others => '0');
REG_2_Q1         := C_2 * Q_ERG_2;

if VAL_2_C(7) = '0' then
        REG_2_Q2 := COR_2_L + REG_2_Q1(38 downto 25);
else
        REG_2_Q2 := COR_2_L - REG_2_Q1(38 downto 25);
end if;
COR_2_LQ := REG_2_Q2;

if REG_2_Q2(14) = '1' then
        OVERFLOW_Q := '1';
else
        OVERFLOW_Q := '0';
end if;
---------------------------------------
```

```
-----------------------
-- Offset correction --
-----------------------
REG_2_O   := (others => '0');
A_2       := (others => '0');

if  VAL_2_A(7) = '0' then
        A_2(10 downto 4) := VAL_2_A(6 downto 0);
        REG_2_O := COR_2_LQ + A_2;
else
        A_2(10 downto 4) := VAL_2_A(6 downto 0);
        REG_2_O := COR_2_LQ - A_2;
end if;

COR_2_LQO := REG_2_O(13 downto 0);
```

```
                        if REG_2_O(14) = '1' then
                                OVERFLOW_O := '1';
                        else
                                OVERFLOW_O := '0';
                        end if;
                        ---------------------------------------




                        ---
                        -- Result ----------------------------
                        ---
                        ERG_2 <= COR_2_LQO;
                        ---------------------------------------




                        ---
                        -- Error processing -------------------
                        ---
                        VD_O_2 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_2) and VD_LUT;
                        ---------------------------------------

                end if;
                last_RNE_4 := RNE_4;
            end if;
        end if;
    end process;




    -----------------------------
    -- Energy Correction of Hit 3
    -----------------------------
    ac_hit3:
    process(CLK)
        variable Q_ERG_3        : unsigned (27 downto 0);
        variable VAL_3_A        : unsigned (7 downto 0);
        variable VAL_3_B        : unsigned (7 downto 0);
        variable VAL_3_C        : unsigned (7 downto 0);
        variable C_3            : unsigned (10 downto 0);
        variable A_3            : unsigned (14 downto 0);
        variable REG_3_Q1       : unsigned (38 downto 0);
        variable REG_3_Q2       : unsigned (14 downto 0);
        variable REG_3_L        : unsigned (21 downto 0);
        variable REG_3_O        : unsigned (14 downto 0);
        variable COR_3_L        : unsigned (14 downto 0);
        variable COR_3_Q        : unsigned (14 downto 0);
        variable COR_3_LQ       : unsigned (14 downto 0);
        variable COR_3_LQO      : unsigned (13 downto 0);
        variable OVERFLOW_Q     : std_logic;
```

```
   variable OVERFLOW_L    : std_logic;
   variable OVERFLOW_O    : std_logic;
   variable last_RNE_4    : std_logic;

begin

   if rising_edge(CLK) then
       if RESET = '1' then
           Q_ERG_3     := (others => '0');
           VAL_3_A     := (others => '0');
           VAL_3_B     := (others => '0');
           VAL_3_C     := (others => '0');
           C_3         := (others => '0');
           A_3         := (others => '0');
           REG_3_Q1    := (others => '0');
           REG_3_Q2    := (others => '0');
           REG_3_L     := (others => '0');
           REG_3_O     := (others => '0');
           COR_3_L     := (others => '0');
           COR_3_LQ    := (others => '0');
           COR_3_LQO   := (others => '0');
           ERG_3       <= (others => '0');
           OVERFLOW_Q := '0';
           OVERFLOW_L := '0';
           OVERFLOW_O := '0';
           VD_O_3      <= '0';
           last_RNE_4 := '0';

       else

           if RNE_4 = '1' and last_RNE_4 = '0' then
                   VAL_3_A := LUT_OP_3(23 downto 16);
                   VAL_3_B := LUT_OP_3(15 downto 8);
                   VAL_3_C := LUT_OP_3(7  downto 0);




                   -----------------------
                   -- Linear correction --
                   -----------------------
                   REG_3_L              := VAL_3_B * ERG_3_i;
                   COR_3_L(13 downto 0) := REG_3_L(20 downto 7);

                   if REG_3_L(21) = '1' then
                           OVERFLOW_L := '1';
                   else
                           OVERFLOW_L := '0';
                   end if;
                   ---------------------------------------





                   --------------------------
                   -- Quadratic correction --
                   --------------------------
                   Q_ERG_3          := ERG_3_i * ERG_3_i;
```

```
C_3              := (others => '0');
C_3(6 downto 0)  := VAL_3_C(6 downto 0);
REG_3_Q1         := (others => '0');
REG_3_Q2         := (others => '0');
REG_3_Q1         := C_3 * Q_ERG_3;

if VAL_3_C(7) = '0' then
        REG_3_Q2 := COR_3_L + REG_3_Q1(38 downto 25);
else
        REG_3_Q2 := COR_3_L - REG_3_Q1(38 downto 25);
end if;
COR_3_LQ := REG_3_Q2;

if REG_3_Q2(14) = '1' then
        OVERFLOW_Q := '1';
else
        OVERFLOW_Q := '0';
end if;
---------------------------------------



-----------------------
-- Offset correction --
-----------------------
REG_3_O   := (others => '0');
A_3       := (others => '0');

if  VAL_3_A(7) = '0' then
        A_3(10 downto 4) := VAL_3_A(6 downto 0);
        REG_3_O := COR_3_LQ + A_3;
else
        A_3(10 downto 4) := VAL_3_A(6 downto 0);
        REG_3_O := COR_3_LQ - A_3;
end if;

COR_3_LQO := REG_3_O(13 downto 0);

if REG_3_O(14) = '1' then
        OVERFLOW_O := '1';
else
        OVERFLOW_O := '0';
end if;
---------------------------------------



---
-- Result -----------------------------
---
ERG_3 <= COR_3_LQO;
---------------------------------------
```

```
                    ---
                    -- Error processing -------------------
                    ---
                    VD_O_3 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_3) and VD_LUT;
                    ---------------------------------------

                end if;
                last_RNE_4 := RNE_4;
            end if;
        end if;
    end process;





    -----------------------------
    -- Energy Correction of Hit 4
    -----------------------------
    ac_hit4:
    process(CLK)
        variable Q_ERG_4        : unsigned (27 downto 0);
        variable VAL_4_A        : unsigned (7 downto 0);
        variable VAL_4_B        : unsigned (7 downto 0);
        variable VAL_4_C        : unsigned (7 downto 0);
        variable C_4            : unsigned (10 downto 0);
        variable A_4            : unsigned (14 downto 0);
        variable REG_4_Q1       : unsigned (38 downto 0);
        variable REG_4_Q2       : unsigned (14 downto 0);
        variable REG_4_L        : unsigned (21 downto 0);
        variable REG_4_O        : unsigned (14 downto 0);
        variable COR_4_L        : unsigned (14 downto 0);
        variable COR_4_Q        : unsigned (14 downto 0);
        variable COR_4_LQ       : unsigned (14 downto 0);
        variable COR_4_LQO      : unsigned (13 downto 0);
        variable OVERFLOW_Q     : std_logic;
        variable OVERFLOW_L     : std_logic;
        variable OVERFLOW_O     : std_logic;
        variable last_RNE_4     : std_logic;

    begin

        if rising_edge(CLK) then
            if RESET = '1' then
                Q_ERG_4    := (others => '0');
                VAL_4_A    := (others => '0');
                VAL_4_B    := (others => '0');
                VAL_4_C    := (others => '0');
                C_4        := (others => '0');
                A_4        := (others => '0');
                REG_4_Q1   := (others => '0');
                REG_4_Q2   := (others => '0');
                REG_4_L    := (others => '0');
                REG_4_O    := (others => '0');
                COR_4_L    := (others => '0');
                COR_4_LQ   := (others => '0');
```

```
        COR_4_LQO  := (others => '0');
        ERG_4      <= (others => '0');
        OVERFLOW_Q := '0';
        OVERFLOW_L := '0';
        OVERFLOW_O := '0';
        VD_O_4     <= '0';
        last_RNE_4 := '0';

    else

        if RNE_4 = '1' and last_RNE_4 = '0' then
                VAL_4_A := LUT_OP_4(23 downto 16);
                VAL_4_B := LUT_OP_4(15 downto 8);
                VAL_4_C := LUT_OP_4(7  downto 0);



                -----------------------
                -- Linear correction --
                -----------------------
                REG_4_L              := VAL_4_B * ERG_4_i;
                COR_4_L(13 downto 0) := REG_4_L(20 downto 7);

                if REG_4_L(21) = '1' then
                        OVERFLOW_L := '1';
                else
                        OVERFLOW_L := '0';
                end if;
                ---------------------------------------



                --------------------------
                -- Quadratic correction --
                --------------------------
                Q_ERG_4          := ERG_4_i * ERG_4_i;
                C_4              := (others => '0');
                C_4(6 downto 0)  := VAL_4_C(6 downto 0);
                REG_4_Q1         := (others => '0');
                REG_4_Q2         := (others => '0');
                REG_4_Q1         := C_4 * Q_ERG_4;

                if VAL_4_C(7) = '0' then
                        REG_4_Q2 := COR_4_L + REG_4_Q1(38 downto 25);
                else
                        REG_4_Q2 := COR_4_L - REG_4_Q1(38 downto 25);
                end if;
                COR_4_LQ := REG_4_Q2;

                if REG_4_Q2(14) = '1' then
                        OVERFLOW_Q := '1';
                else
                        OVERFLOW_Q := '0';
                end if;
                ---------------------------------------
```

```
                           -----------------------
                           -- Offset correction --
                           -----------------------
                           REG_4_O   := (others => '0');
                           A_4       := (others => '0');

                           if  VAL_4_A(7) = '0' then
                                   A_4(10 downto 4) := VAL_4_A(6 downto 0);
                                   REG_4_O := COR_4_LQ + A_4;
                           else
                                   A_4(10 downto 4) := VAL_4_A(6 downto 0);
                                   REG_4_O := COR_4_LQ - A_4;
                           end if;

                           COR_4_LQO := REG_4_O(13 downto 0);

                           if REG_4_O(14) = '1' then
                                   OVERFLOW_O := '1';
                           else
                                   OVERFLOW_O := '0';
                           end if;
                           ---------------------------------------


                           ---
                           -- Result -----------------------------
                           ---
                           ERG_4 <= COR_4_LQO;
                           ---------------------------------------


                           ---
                           -- Error processing -------------------
                           ---
                           VD_O_4 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_4) and VD_LUT;
                           ---------------------------------------

                   end if;
                   last_RNE_4 := RNE_4;
               end if;
           end if;
        end process;



        -----------------------------
        -- Energy Correction of Hit 5
        -----------------------------
        ac_hit5:
        process(CLK)
           variable Q_ERG_5         : unsigned (27 downto 0);
           variable VAL_5_A         : unsigned (7 downto 0);
           variable VAL_5_B         : unsigned (7 downto 0);
           variable VAL_5_C         : unsigned (7 downto 0);
           variable C_5             : unsigned (10 downto 0);
           variable A_5             : unsigned (14 downto 0);
```

```
        variable REG_5_Q1       : unsigned (38 downto 0);
        variable REG_5_Q2       : unsigned (14 downto 0);
        variable REG_5_L        : unsigned (21 downto 0);
        variable REG_5_O        : unsigned (14 downto 0);
        variable COR_5_L        : unsigned (14 downto 0);
        variable COR_5_Q        : unsigned (14 downto 0);
        variable COR_5_LQ       : unsigned (14 downto 0);
        variable COR_5_LQO      : unsigned (13 downto 0);
        variable OVERFLOW_Q     : std_logic;
        variable OVERFLOW_L     : std_logic;
        variable OVERFLOW_O     : std_logic;
        variable last_RNE_4     : std_logic;

     begin

        if rising_edge(CLK) then
            if RESET = '1' then
                Q_ERG_5    := (others => '0');
                VAL_5_A    := (others => '0');
                VAL_5_B    := (others => '0');
                VAL_5_C    := (others => '0');
                C_5        := (others => '0');
                A_5        := (others => '0');
                REG_5_Q1   := (others => '0');
                REG_5_Q2   := (others => '0');
                REG_5_L    := (others => '0');
                REG_5_O    := (others => '0');
                COR_5_L    := (others => '0');
                COR_5_LQ   := (others => '0');
                COR_5_LQO  := (others => '0');
                ERG_5      <= (others => '0');
                OVERFLOW_Q := '0';
                OVERFLOW_L := '0';
                OVERFLOW_O := '0';
                VD_O_5     <= '0';
                last_RNE_4 := '0';

            else

                if RNE_4 = '1' and last_RNE_4 = '0' then
                        VAL_5_A := LUT_OP_5(23 downto 16);
                        VAL_5_B := LUT_OP_5(15 downto 8);
                        VAL_5_C := LUT_OP_5(7  downto 0);



                        -----------------------
                        -- Linear correction --
                        -----------------------
                        REG_5_L              := VAL_5_B * ERG_5_i;
                        COR_5_L(13 downto 0) := REG_5_L(20 downto 7);

                        if REG_5_L(21) = '1' then
                                OVERFLOW_L := '1';
                        else
                                OVERFLOW_L := '0';
                        end if;
                        ---------------------------------------
```

```
--------------------------
-- Quadratic correction --
--------------------------
Q_ERG_5         := ERG_5_i * ERG_5_i;
C_5             := (others => '0');
C_5(6 downto 0) := VAL_5_C(6 downto 0);
REG_5_Q1        := (others => '0');
REG_5_Q2        := (others => '0');
REG_5_Q1        := C_5 * Q_ERG_5;

if VAL_5_C(7) = '0' then
        REG_5_Q2 := COR_5_L + REG_5_Q1(38 downto 25);
else
        REG_5_Q2 := COR_5_L - REG_5_Q1(38 downto 25);
end if;
COR_5_LQ := REG_5_Q2;

if REG_5_Q2(14) = '1' then
        OVERFLOW_Q := '1';
else
        OVERFLOW_Q := '0';
end if;
---------------------------------------



-----------------------
-- Offset correction --
-----------------------
REG_5_O   := (others => '0');
A_5       := (others => '0');

if  VAL_5_A(7) = '0' then
        A_5(10 downto 4) := VAL_5_A(6 downto 0);
        REG_5_O := COR_5_LQ + A_5;
else
        A_5(10 downto 4) := VAL_5_A(6 downto 0);
        REG_5_O := COR_5_LQ - A_5;
end if;

COR_5_LQO := REG_5_O(13 downto 0);

if REG_5_O(14) = '1' then
        OVERFLOW_O := '1';
else
        OVERFLOW_O := '0';
end if;
---------------------------------------



---
-- Result ----------------------------
---
ERG_5 <= COR_5_LQO;
```

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




                           ---
                           -- Error processing -------------------
                           ---
                           VD_O_5 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_5) and VD_LUT;
                           ---------------------------------------

                   end if;
                   last_RNE_4 := RNE_4;
               end if;
           end if;
        end process;




        -----------------------------
        -- Energy Correction of Hit 6
        -----------------------------
        ac_hit6:
        process(CLK)
           variable Q_ERG_6        : unsigned (27 downto 0);
           variable VAL_6_A        : unsigned (7 downto 0);
           variable VAL_6_B        : unsigned (7 downto 0);
           variable VAL_6_C        : unsigned (7 downto 0);
           variable C_6            : unsigned (10 downto 0);
           variable A_6            : unsigned (14 downto 0);
           variable REG_6_Q1       : unsigned (38 downto 0);
           variable REG_6_Q2       : unsigned (14 downto 0);
           variable REG_6_L        : unsigned (21 downto 0);
           variable REG_6_O        : unsigned (14 downto 0);
           variable COR_6_L        : unsigned (14 downto 0);
           variable COR_6_Q        : unsigned (14 downto 0);
           variable COR_6_LQ       : unsigned (14 downto 0);
           variable COR_6_LQO      : unsigned (13 downto 0);
           variable OVERFLOW_Q     : std_logic;
           variable OVERFLOW_L     : std_logic;
           variable OVERFLOW_O     : std_logic;
           variable last_RNE_4     : std_logic;

        begin

           if rising_edge(CLK) then
               if RESET = '1' then
                   Q_ERG_6    := (others => '0');
                   VAL_6_A    := (others => '0');
                   VAL_6_B    := (others => '0');
                   VAL_6_C    := (others => '0');
                   C_6        := (others => '0');
                   A_6        := (others => '0');
                   REG_6_Q1   := (others => '0');
                   REG_6_Q2   := (others => '0');
                   REG_6_L    := (others => '0');
                   REG_6_O    := (others => '0');
```

```
        COR_6_L    := (others => '0');
        COR_6_LQ   := (others => '0');
        COR_6_LQO  := (others => '0');
        ERG_6      <= (others => '0');
        OVERFLOW_Q := '0';
        OVERFLOW_L := '0';
        OVERFLOW_O := '0';
        VD_O_6     <= '0';
        last_RNE_4 := '0';

    else

        if RNE_4 = '1' and last_RNE_4 = '0' then
                VAL_6_A := LUT_OP_6(23 downto 16);
                VAL_6_B := LUT_OP_6(15 downto 8);
                VAL_6_C := LUT_OP_6(7  downto 0);



                -----------------------
                -- Linear correction --
                -----------------------
                REG_6_L              := VAL_6_B * ERG_6_i;
                COR_6_L(13 downto 0) := REG_6_L(20 downto 7);

                if REG_6_L(21) = '1' then
                        OVERFLOW_L := '1';
                else
                        OVERFLOW_L := '0';
                end if;
                ---------------------------------------



                --------------------------
                -- Quadratic correction --
                --------------------------
                Q_ERG_6          := ERG_6_i * ERG_6_i;
                C_6              := (others => '0');
                C_6(6 downto 0)  := VAL_6_C(6 downto 0);
                REG_6_Q1         := (others => '0');
                REG_6_Q2         := (others => '0');
                REG_6_Q1         := C_6 * Q_ERG_6;

                if VAL_6_C(7) = '0' then
                        REG_6_Q2 := COR_6_L + REG_6_Q1(38 downto 25);
                else
                        REG_6_Q2 := COR_6_L - REG_6_Q1(38 downto 25);
                end if;
                COR_6_LQ := REG_6_Q2;

                if REG_6_Q2(14) = '1' then
                        OVERFLOW_Q := '1';
                else
                        OVERFLOW_Q := '0';
                end if;
                ---------------------------------------
```

```
                        -----------------------
                        -- Offset correction --
                        -----------------------
                        REG_6_O   := (others => '0');
                        A_6       := (others => '0');

                        if  VAL_6_A(7) = '0' then
                                A_6(10 downto 4) := VAL_6_A(6 downto 0);
                                REG_6_O := COR_6_LQ + A_6;
                        else
                                A_6(10 downto 4) := VAL_6_A(6 downto 0);
                                REG_6_O := COR_6_LQ - A_6;
                        end if;

                        COR_6_LQO := REG_6_O(13 downto 0);

                        if REG_6_O(14) = '1' then
                                OVERFLOW_O := '1';
                        else
                                OVERFLOW_O := '0';
                        end if;
                        ---------------------------------------



                        ---
                        -- Result -----------------------------
                        ---
                        ERG_6 <= COR_6_LQO;
                        ---------------------------------------



                        ---
                        -- Error processing -------------------
                        ---
                        VD_O_6 <= NOT(OVERFLOW_Q) and NOT(OVERFLOW_L) and
NOT(OVERFLOW_O) and NOT(ERROR_1_6) and VD_LUT;
                        ---------------------------------------

                end if;
                last_RNE_4 := RNE_4;
            end if;
        end if;
    end process;



    ------------------------------------------
    -- Compound results to Output Event Packet
    ------------------------------------------
    results:
    process(CLK)
        variable last_RNE_5: std_logic;
        constant ERGNULL: unsigned (13 downto 0) := (others => '0');
    begin
```

```
if rising_edge(CLK) then
    if RESET = '1' then

        ---
        -- Reset Signals and variables
        ---
        REG_OUT <= (others => '0');
        last_RNE_5 := '0';
    else
        if RNE_5 = '1' and last_RNE_5 = '0' then

                REG_OUT(123 downto 122) <= REG_IN(129 downto 128);
                REG_OUT(121 downto 117) <= REG_IN(125 downto 121);
                REG_OUT(116) <= REG_IN(120);

                if ERG_1_i = ERGNULL then
                        REG_OUT(137 downto 124) <= (others => '0');
                        REG_OUT(115) <= '1';
                else
                        REG_OUT(137 downto 124) <= ERG_1;
                        REG_OUT(115) <= VD_O_1;
                end if;




                REG_OUT(100 downto  99) <= REG_IN(105 downto 104);
                REG_OUT(98 downto 94) <= REG_IN(101 downto 97);
                REG_OUT(93) <= REG_IN(96);

                if ERG_2_i = ERGNULL then
                        REG_OUT(114 downto 101) <= (others => '0');
                        REG_OUT(92) <= '1';
                else
                        REG_OUT(114 downto 101) <= ERG_2;
                        REG_OUT(92) <= VD_O_2;
                end if;




                REG_OUT(77  downto  76) <= REG_IN(81 downto 80);
                REG_OUT(75  downto  71) <= REG_IN(77 downto 73);
                REG_OUT(70) <= REG_IN(72);
                if ERG_3_i = ERGNULL then
                        REG_OUT(91  downto  78) <= (others => '0');
                        REG_OUT(69) <= '1';
                else
                        REG_OUT(91  downto  78) <= ERG_3;
                        REG_OUT(69) <= VD_O_3;
                end if;




                REG_OUT(54  downto  53) <= REG_IN(55 downto 54);
                REG_OUT(52  downto  48) <= REG_IN(53 downto 49);
                REG_OUT(47) <= REG_IN(48);
```

```
                            if ERG_4_i = ERGNULL then
                                    REG_OUT(68  downto  55) <= (others => '0');
                                    REG_OUT(46) <= '1';
                            else
                                    REG_OUT(68  downto  55) <= ERG_4;
                                    REG_OUT(46) <= VD_O_4;
                            end if;




                            REG_OUT(31  downto  30) <= REG_IN(31 downto 30);
                            REG_OUT(29  downto  25) <= REG_IN(29 downto 25);
                            REG_OUT(24) <= REG_IN(24);
                            if ERG_5_i = ERGNULL then
                                    REG_OUT(45  downto  32) <= (others => '0');
                                    REG_OUT(23) <= '1';
                            else
                                    REG_OUT(45  downto  32) <= ERG_5;
                                    REG_OUT(23) <= VD_O_5;
                            end if;




                            REG_OUT(8   downto   7) <= REG_IN(7 downto 6);
                            REG_OUT(6   downto   2) <= REG_IN(5 downto 1);
                            REG_OUT(1) <= REG_IN(0);
                            if ERG_6_i = ERGNULL then
                                    REG_OUT(22  downto  9) <= (others => '0');
                                    REG_OUT(0) <= '1';
                            else
                                    REG_OUT(22  downto  9) <= ERG_6;
                                    REG_OUT(0) <= VD_O_6;
                            end if;
                    end if;
                    last_RNE_5 := RNE_5;
                end if;
            end if;
        end process;
end inside;
```

B.2.3 LUTC-Module

```
---------------------------------------------------------------------------
----  Controller for Look Up Table  , Version 0.4, 06.09.00,  TS/IAAT  ----
---------------------------------------------------------------------------

---
-- LUTC (Look Up Table Controller) / EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE_2         : Sub-RNE-Signals for Sequence Control from EC_IN
-- LOAD          : invoke load-process for Look Up Table (from EPP_TOP)
-- LUT_IN_1/6    : LUT-Address, generated by AC-Module from the Event-Packet.
--
------
-- Task          : Controlling and loading of the Look-Up-Table.
--
------
-- Output Signals:
--
-- LUT_LOAD      : Flag that indicates that a LUT-Load process is in running.
--                 To AC-Module, to veto current Event-Packets.
-- ERROR_1_1     : Error Status of the LUT-Entry under specified address,
--                 to AC-Module.
-- LUT_OUT_1/6   : Correction words from inside the LUT, back to the AC-Module
--
--------------------------------------------------------------------------



-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity LUTC is

      port(   CLK          : in std_logic;
              RESET        : in std_logic;
              RNE_2        : in std_logic;
              LOAD         : in std_logic;
              LUT_IN_1     : in unsigned (8 downto 0);
              LUT_IN_2     : in unsigned (8 downto 0);
              LUT_IN_3     : in unsigned (8 downto 0);
              LUT_IN_4     : in unsigned (8 downto 0);
              LUT_IN_5     : in unsigned (8 downto 0);
              LUT_IN_6     : in unsigned (8 downto 0);
              LUT_LOAD     : out std_logic;
```

```
              ERROR_1_1    : out std_logic;
              ERROR_1_2    : out std_logic;
              ERROR_1_3    : out std_logic;
              ERROR_1_4    : out std_logic;
              ERROR_1_5    : out std_logic;
              ERROR_1_6    : out std_logic;
              LUT_OUT_1    : out unsigned (23 downto 0);
              LUT_OUT_2    : out unsigned (23 downto 0);
              LUT_OUT_3    : out unsigned (23 downto 0);
              LUT_OUT_4    : out unsigned (23 downto 0);
              LUT_OUT_5    : out unsigned (23 downto 0);
              LUT_OUT_6    : out unsigned (23 downto 0)
            );
end LUTC;


architecture inside of LUTC is

signal LOAD_DATA : std_logic;

begin


    -----------------------
    -- load LUT_Matrix Data
    -----------------------
    load_LUT_Data:
    process (CLK)

            type LUT_array is array (0 to 1023) of unsigned (23 downto 0);
            variable LUT            : LUT_array;

            variable LUT_REG        : unsigned(23 downto 0);
            variable d_lut_a        : integer;
            variable d_lut_b        : integer;
            variable d_lut_c        : integer;
            variable done           : boolean := FALSE;
            variable LA_1           : integer range 0 to 1024;
            variable LA_2           : integer range 0 to 1024;
            variable LA_3           : integer range 0 to 1024;
            variable LA_4           : integer range 0 to 1024;
            variable LA_5           : integer range 0 to 1024;
            variable LA_6           : integer range 0 to 1024;
            variable i              : integer range 0 to 1024;
            variable last_RNE_2     : std_logic;

            file data: text is in "stimuli/LUT.dat";
            -- Procedure, load Data from Lookup-Table Datafile
            procedure read_data_line is
                    variable L : line;
            begin
                    if (not endfile(data)) then
                            readline(data, L);
                            read(L, d_lut_a);
                            read(L, d_lut_b);
                            read(L, d_lut_c);
                    else
                            done := TRUE;
```

```
                end if;
        end;
-- begin of process
begin
    if rising_edge (CLK) then
        if RESET = '1' then

            ---
            -- Reset Signals and Variables
            ---
            LUT_REG         := (others => '0');
            d_lut_a         := 0;
            d_lut_b         := 0;
            d_lut_c         := 0;
            LA_1            := 0;
            LA_2            := 0;
            LA_3            := 0;
            LA_4            := 0;
            LA_5            := 0;
            LA_6            := 0;
            ERROR_1_1       <= '0';
            ERROR_1_2       <= '0';
            ERROR_1_3       <= '0';
            ERROR_1_4       <= '0';
            ERROR_1_5       <= '0';
            ERROR_1_6       <= '0';
            LUT_OUT_1       <= (others => '0');
            LUT_OUT_2       <= (others => '0');
            LUT_OUT_3       <= (others => '0');
            LUT_OUT_4       <= (others => '0');
            LUT_OUT_5       <= (others => '0');
            LUT_OUT_6       <= (others => '0');
            LOAD_DATA       <= '0';
            LUT_LOAD        <= '0';
            i               := 0;
            last_RNE_2      := '0';

            for j in 0 to 1023 loop
                    LUT(j) := (others => '0');
            end loop;
        else


            ---
            -- Load LUT Data from File to Array
            ---
            if LOAD = '1' or LOAD_DATA = '1' then
                    read_data_line;
                    if not done then
                            LOAD_DATA <= '1';
                            LUT_LOAD <= '1';

                            LUT_REG := (others => '0');
                            LUT_REG(23 downto 16) := CONV_UNSIGNED(d_lut_a, 8);
                            LUT_REG(15 downto  8) := CONV_UNSIGNED(d_lut_b, 8);
                            LUT_REG(7  downto  0) := CONV_UNSIGNED(d_lut_c, 8);
                            if i < 1024 then
                                    LUT(i) := LUT_REG; -- generate LUT
```

```
                        i := i + 1;        -- from Filedatas
                else
                        i := 0;
                end if;
        else
                LOAD_DATA <= '0';
                LUT_LOAD <= '0';
        end if;
end if;



---
-- Generate Hit-dependent Correction-Signals
---
-- LA_1    :  set LUT Address
---

if RNE_2 = '1' and last_RNE_2 = '0' then

        LA_1 := CONV_INTEGER(LUT_IN_1);
        if LA_1 > 511 then
                ERROR_1_1 <= '1';
                LUT_OUT_1 <= LUT(0);
        else
                ERROR_1_1 <= '0';
                LUT_OUT_1 <= LUT(LA_1);
        end if;


        LA_2 := CONV_INTEGER(LUT_IN_2);
        if LA_2 > 511 then
                ERROR_1_2 <= '1';
                LUT_OUT_2 <= LUT(0);
        else
                ERROR_1_2 <= '0';
                LUT_OUT_2 <= LUT(LA_2);
        end if;


        LA_3 := CONV_INTEGER(LUT_IN_3);
        if LA_3 > 511 then
                ERROR_1_3 <= '1';
                LUT_OUT_3 <= LUT(0);
        else
                ERROR_1_3 <= '0';
                LUT_OUT_3 <= LUT(LA_3);
        end if;


        LA_4 := CONV_INTEGER(LUT_IN_4);
        if LA_4 > 511 then
                ERROR_1_4 <= '1';
                LUT_OUT_4 <= LUT(0);
        else
                ERROR_1_4 <= '0';
                LUT_OUT_4 <= LUT(LA_4 + 512);
        end if;
```

```
                        LA_5 := CONV_INTEGER(LUT_IN_5);
                        if LA_5 > 511 then
                                ERROR_1_5 <= '1';
                                LUT_OUT_5 <= LUT(0);
                        else
                                ERROR_1_5 <= '0';
                                LUT_OUT_5 <= LUT(LA_5 + 512);
                        end if;


                        LA_6 := CONV_INTEGER(LUT_IN_6);
                        if LA_6 > 511 then
                                ERROR_1_6 <= '1';
                                LUT_OUT_6 <= LUT(0);
                        else
                                ERROR_1_6 <= '0';
                                LUT_OUT_6 <= LUT(LA_6 + 512);
                        end if;
                end if;
                last_RNE_2 := RNE_2;
            end if;
        end if;

    end process;
end inside;
```

B.2.4 EL-Module

```
--------------------------------------------------------------------------
---- Event Location - Energy Comparator Rel. 0.6 , 10.10.00 TS/IAAT ----
--------------------------------------------------------------------------

---
-- EL (Event Location) / EC-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- REG_IN        : Event-Packet Remnant from AC-Module.
--
-- Hit 1     [ 137          124 : 123        117 :     116      :   115  ]
-- Hit 2     [ 114          101 : 100         94 :      93      :    92  ]
-- Hit 3     [  91           78 :  77         71 :      70      :    69  ]
-- Hit 4     [  68           55 :  54         48 :      47      :    46  ]
-- Hit 5     [  45           32 :  31         25 :      24      :    23  ]
-- Hit 6     [  22            9 :   8          2 :       1      :     0  ]
--
-- RNE_6         : Sub-RNE for sequential control from EC_IN-Module.
--
------
-- Task:  Find anode (cathode) with maximum amplitude signal, synchron
--        with RNE_6. For that, compare the amplitudes inside the Event-Packet
--        Remnant.
--
--
------
-- Output Signals:
--
-- LOC_A         : 7 Bit Location of Anode with maximum signal.
--                 -> to EC_OUT
-- LOC_C         : 7 Bit Location of Cathode with maximum signal.
--                 -> to EC-OUT
-- VD_EL_A       : Valid Data Flag to ED-Module
-- VD_EL_C       : Valid Data Flag to ED-Module
--
---------------------------------------------------------------------------


-- VHDL script starts here:



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


entity EL is
        port    ( CLK         : in std_logic;
                  RESET       : in std_logic;
```

```
                  RNE_6        : in std_logic;
                  REG_IN       : in unsigned (137 downto 0);
                  LOC_A        : out unsigned (6 downto 0); -- Location Out
                  LOC_C        : out unsigned (6 downto 0); -- Location Out
                  VD_EL_A      : out std_logic;
                  VD_EL_C      : out std_logic
                );
end EL;


architecture inside of EL is


begin


       ---
       -- detect location of Anode
       ---
       process (CLK)
               variable VEC1           : unsigned( 20 downto 0);
               variable VEC2           : unsigned( 20 downto 0);
               variable VEC3           : unsigned( 20 downto 0);
               variable REG_OPA        : unsigned( 20 downto 0);
               variable last_RNE_6     : std_logic;
       begin
               if rising_edge(CLK) then
                       if RESET = '1' then
                               VEC1           := (others => '0');
                               VEC2           := (others => '0');
                               VEC3           := (others => '0');
                               VD_EL_A        <= '0';
                               REG_OPA        := (others => '0');
                               LOC_A          <= (others => '0');
                               last_RNE_6     := '0';
                       else
                               if RNE_6 = '1' and last_RNE_6 = '0' then

                                       ---
                                       -- No Error Code necessary
                                       ---
                                       VD_EL_A <= '1';


                                       ---
                                       -- Compound Energy and Location to one Bit-Vector.
                                       ---
                                       VEC1(20 downto 0) := REG_IN(137 downto 117);
                                       VEC2(20 downto 0) := REG_IN(114 downto  94);
                                       VEC3(20 downto 0) := REG_IN(91  downto  71);

                                       ---
                                       -- Compare Bit-Vectors
                                       ---
                                       REG_OPA := VEC1;
                                       if REG_OPA < VEC2 then
                                               REG_OPA := VEC2;
```

```
                                end if;
                                if REG_OPA < VEC3 then
                                        REG_OPA := VEC3;
                                end if;


                                ---
                                -- Discompound Location and Energy vector, get
                                -- Location Vector
                                ---
                                LOC_A(6 downto 0) <= REG_OPA(6 downto 0);
                        end if;
                end if;

                last_RNE_6 := RNE_6;
        end if;
end process;






---
-- detect location of Cathode
---
process (CLK)
        variable VEC4           : unsigned( 20 downto 0);
        variable VEC5           : unsigned( 20 downto 0);
        variable VEC6           : unsigned( 20 downto 0);
        variable REG_OPC        : unsigned( 20 downto 0);
        variable last_RNE_6     : std_logic;
begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        VEC4            := (others => '0');
                        VEC5            := (others => '0');
                        VEC6            := (others => '0');
                        VD_EL_C         <= '0';
                        REG_OPC         := (others => '0');
                        LOC_C           <= (others => '0');
                        last_RNE_6      := '0';
                else
                        if RNE_6 = '1' and last_RNE_6 = '0' then

                                ---
                                -- No Error Code necessary
                                ---
                                VD_EL_C <= '1';


                                ---
                                -- Compound Energy and Location to one Bit-Vector.
                                ---
                                VEC4(20 downto 0) := REG_IN(68 downto 48);
                                VEC5(20 downto 0) := REG_IN(45 downto 25);
                                VEC6(20 downto 0) := REG_IN(22 downto 2);

                                ---
```

```
                                                -- Compare Bit-Vectors
                                                ---
                                                REG_OPC := VEC4;
                                                if REG_OPC < VEC5 then
                                                        REG_OPC := VEC5;
                                                end if;
                                                if REG_OPC < VEC6 then
                                                        REG_OPC := VEC6;
                                                end if;

                                                ---
                                                -- Discompound Location and Energy vector, get
                                                -- Location Vector
                                                ---
                                                LOC_C(6 downto 0) <= REG_OPC(6 downto 0);
                                        end if;
                                end if;

                                last_RNE_6 := RNE_6;
                        end if;
                end process;
end inside;
```

B.2.5 SOA-Module

```
----------------------------------------------------------------------------
-- Sum of Energy ; Sum amplitude of all Hits  Rev 0.6 , 10.10.00 TS/IAAT --
----------------------------------------------------------------------------

---
-- SOA (Sum Of Anodes) /EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
-- REG_IN         : Event-Packet Remnant from AC-Module.
--
-- Hit 1     [ 137          124 : 123        117 :     116      :   115  ]
-- Hit 2     [ 114          101 : 100         94 :      93      :    92  ]
-- Hit 3     [  91           78 :  77         71 :      70      :    69  ]
-- Hit 4     [  68           55 :  54         48 :      47      :    46  ]
-- Hit 5     [  45           32 :  31         25 :      24      :    23  ]
-- Hit 6     [  22            9 :   8          2 :       1      :     0  ]
--
-- RNE_6          : Sub-RNE for sequential control from EC_IN-Module.
--
------
-- Task           :  Sum all Anode-Entries of the Event-Packet Remnant (REG_IN).
--                   Start task synchron with RNE_6.
--
------
-- Output Signals:
--
-- REG_OUT        : 16 Bit Output Signal -> to DIV-Module
-- VD_SOC         : Valid Data Flag to ED-Module
--
--------------------------------------------------------------------------


-- VHDL script starts here:



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


entity SOA is
        port( CLK              : in std_logic;
              RESET            : in std_logic;
              RNE_6            : in std_logic;
              REG_IN           : in unsigned (137 downto 0);
              REG_OUT          : out unsigned (15 downto 0); -- Location Out
              VD_SOA           : out std_logic
            );
end SOA;
```

```
architecture inside of SOA is

begin

        ---
        -- add energy vectors
        ---
        process (CLK)
                variable VEC1            : unsigned (15 downto 0);
                variable VEC2            : unsigned (15 downto 0);
                variable VEC3            : unsigned (15 downto 0);
                variable SUM             : unsigned (15 downto 0);
                variable last_RNE_6      : std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                VEC1            := (others => '0');
                                VEC2            := (others => '0');
                                VEC3            := (others => '0');
                                VD_SOA          <= '0';
                                SUM             := (others => '0');
                                REG_OUT         <= (others => '0');
                                last_RNE_6      := '0';
                        else
                                ---
                                -- Decompound energy contents of REG_IN to
                                -- separate vectors.
                                ---

                                if RNE_6 = '1' and last_RNE_6 = '0' then
                                        VEC1 := (others => '0');
                                        VEC1(13 downto 0) := REG_IN(137 downto 124);
                                        VEC2 := (others => '0');
                                        VEC2(13 downto 0) := REG_IN(114 downto 101);
                                        VEC3 := (others => '0');
                                        VEC3(13 downto 0) := REG_IN(91 downto 78);
                                        ---
                                        -- Add energy values
                                        ---
                                        SUM := VEC1 + VEC2 + VEC3;

                                        ---
                                        -- Convert Energy to Bit-Vector
                                        ---
                                        REG_OUT <= SUM;

                                        ---
                                        -- Generate Error Flag
                                        ---
                                        VD_SOA <= '1' ;
                                        -- no Error Detection necessary
                                end if;
                        end if;
                        last_RNE_6 := RNE_6;
```

```
                end if;
        end process;
end inside;
```

B.2.6 SOC-Module

```
---------------------------------------------------------------------------
-- Sum of Energy ; Sum amplitude of all Hits  Rev 0.6 , 10.10.00 TS/IAAT --
---------------------------------------------------------------------------

---
-- SOC (Sum Of Cathodes) /EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- REG_IN        : Event-Packet Remnant from AC-Module.
--
-- Hit 1     [ 137          124 : 123        117 :     116      :   115  ]
-- Hit 2     [ 114          101 : 100         94 :      93      :    92  ]
-- Hit 3     [  91           78 :  77         71 :      70      :    69  ]
-- Hit 4     [  68           55 :  54         48 :      47      :    46  ]
-- Hit 5     [  45           32 :  31         25 :      24      :    23  ]
-- Hit 6     [  22            9 :   8          2 :       1      :     0  ]
--
-- RNE_6         : Sub-RNE for sequential control from EC_IN-Module.
--
------
-- Task:  Sum all Cathode-Entries of the Event-Packet Remnant (REG_IN).
--        Start task synchron with RNE_6.
--
------
-- Output Signals:
--
-- REG_OUT       : 16 Bit Output Signal -> to DIV-Module
-- VD_SOC        : Valid Data Flag to ED-Module
--
--------------------------------------------------------------------------


-- VHDL script starts here:



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


entity SOC is
        port( CLK              : in std_logic;
              RESET            : in std_logic;
              RNE_6            : in std_logic;
              REG_IN           : in unsigned (137 downto 0);
              REG_OUT          : out unsigned (15 downto 0); -- Location Out
              VD_SOC           : out std_logic
);
end SOC;
```

```
architecture inside of SOC is

begin

        ---
        -- add energy vectors
        ---
        process (CLK)
                variable VEC4           : unsigned (15 downto 0);
                variable VEC5           : unsigned (15 downto 0);
                variable VEC6           : unsigned (15 downto 0);
                variable SUM            : unsigned (15 downto 0);
                variable last_RNE_6     : std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                VEC4            := (others => '0');
                                VEC5            := (others => '0');
                                VEC6            := (others => '0');
                                VD_SOC          <= '0';
                                SUM             := (others => '0');
                                REG_OUT         <= (others => '0');
                                last_RNE_6      := '0';
                        else
                                ---
                                -- Decompound energy contents of REG_IN to
                                -- separate vectors.
                                ---
                                if RNE_6 = '1' and last_RNE_6 = '0' then
                                        VEC4 := (others => '0');
                                        VEC4(13 downto 0) := REG_IN(68 downto 55);
                                        VEC5 := (others => '0');
                                        VEC5(13 downto 0) := REG_IN(45 downto 32);
                                        VEC6 := (others => '0');
                                        VEC6(13 downto 0) := REG_IN(22 downto 9);
                                        ---
                                        -- Add energy values
                                        ---
                                        SUM := VEC4 + VEC5 + VEC6;

                                        ---
                                        -- Convert Energy to Bit-Vector
                                        ---
                                        REG_OUT <= SUM;

                                        ---
                                        -- Generate Error Flag
                                        ---
                                        VD_SOC <= '1';
                                        -- no Error Detection necessary
                                end if;
                        end if;
                        last_RNE_6 := RNE_6;
                end if;
```

```
        end process;
end inside;
```

B.2.7 DIV-Module

```
-----------------------------------------------------------------------------
----- Divider Module for DOI-Signal  Rel. 0.7 , 10.10.00  TS/IAAT ----------
-----------------------------------------------------------------------------

---
-- DIV (Divider) / EC-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
-- RNE_7          : Sub-RNE for sequential control from FD_IN-Module.
-- REG_SOA        : Sum of Anodes 16 Bit Signal from SOA-Module.
-- REG_SOC        : Sum of Anodes 16 Bit Signal from SOC-Module.
--
------
-- Task:  Divide SOC by SOA, to get DOI-Signal as result. The division
--        operator is made by hand. Division by Zero returns the biggest
--        8 Bit Number possible (255), and set the Valid Data Flag to
--        Zero.
--        The DOI is generated in 8 Bit and drives the DOI_FULL-Output
--        Signal. The DOI-Output use reduced signal (3 Bit) as telemetry
--        output. The Operation starts synchron with RNE_8.
--
------
-- Output Signals:
--
-- DOI_FULL       : Full 8 Bit DOI Output Signal, to DC-Module.
-- DOI            : Reduced 3 Bit DOI Output Signal, to EC_OUT-Module.
-- VD_DIV         : Valid Data Flag about division to ED-Module.
--
---------------------------------------------------------------------------


-- VHDL script starts here:


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


entity DIV is
    port( CLK              : in std_logic;
          RESET            : in std_logic;
          RNE_7            : in std_logic;
          REG_SOA          : in unsigned (15 downto 0);
          REG_SOC          : in unsigned (15 downto 0);
          DOI_FULL         : out unsigned (7 downto 0);
          DOI              : out unsigned (2 downto 0);
          VD_DIV           : out std_logic
        );
end DIV;
```

```
architecture inside of DIV is

        ---
        -- function SHIFT -- shift a bit vector one bit to the left
        ---

        function SHIFT (INP : unsigned) return unsigned is

        variable OUTP : unsigned (20 downto 0);

        begin
                for i in 1 to 20 loop
                       OUTP(21-i) := INP(21-i-1);
                end loop;
                OUTP(0) := '0';
                return OUTP;
        end SHIFT;

signal REG_OUT : unsigned (20 downto 0);

begin

        ---
        -- divider
        ---
        process (CLK)
            variable REG_Z           : unsigned (20 downto 0);
            variable REG_N           : unsigned (20 downto 0);
            variable REG_OP          : unsigned (20 downto 0);
            variable REG_OP_2        : unsigned (20 downto 0);
            variable DIV_BY_ZERO     : std_logic;
            variable RATIO_ERROR     : std_logic;
            variable SIGNAL_LOSS     : std_logic;
            variable DIV_VD          : std_logic;
            variable ZERO            : unsigned (15 downto 0);
            variable last_RNE_7      : std_logic;
        begin
            if rising_edge(CLK) then
                  ZERO := (others => '0');
                  if RESET = '1' then
                        REG_OUT        <= (others => '0');
                        REG_OP         := (others => '0');
                        REG_OP_2       := (others => '0');
                        DIV_VD         := '0';
                        REG_Z          := (others => '0');
                        REG_N          := (others => '0');
                        VD_DIV         <= '0';
                        ZERO           := (others => '0');
                        last_RNE_7     := '0';
                  else

                        if RNE_7 = '1' and last_RNE_7 = '0' then

                             ---
```

```
-- Detect Error Possibilities
---
if REG_SOA = ZERO then
        DIV_BY_ZERO := '1';
else
        DIV_BY_ZERO := '0';
end if;

if REG_SOC < REG_SOA then
        RATIO_ERROR := '1';
else
        RATIO_ERROR := '0';
end if;

if REG_SOC = ZERO then
        SIGNAL_LOSS := '1';
else
        SIGNAL_LOSS := '0';
end if;


---
-- Generating Error Flag
---
DIV_VD := NOT(DIV_BY_ZERO) and NOT(RATIO_ERROR)
and NOT(SIGNAL_LOSS);
VD_DIV <= DIV_VD;


---
-- If VD = 1 process DOI, else set
-- DOI = 1
---
if DIV_VD = '1' then
        REG_OP   := (others => '0');
        REG_OP_2 := (others => '0');
        REG_Z    := (others => '0');
        REG_N    := (others => '0');

        REG_Z(20 downto 5) := REG_SOC;
        REG_N(15 downto 0) := REG_SOA;

        for i in 0 to 20 loop

            REG_OP(0) := REG_Z(20-i);

            if REG_OP < REG_N then
                    REG_OUT(20-i) <= '0';
                    REG_OP_2 := REG_OP;
            else
                    REG_OUT(20-i) <= '1';
                    REG_OP_2 := REG_OP - REG_N;
            end if;
            REG_OP := SHIFT(REG_OP_2);
        end loop;

  else
        REG_OUT    <= (others => '0');
```

```
                                REG_OUT(5) <= '1';
                          end if;
                     end if;
                     last_RNE_7 := RNE_7;
              end if;
          end if;
   end process;



   ---
   -- Grep Output Data Frame
   ---
   grep:
   process (CLK)
   begin
          if rising_edge(CLK) then
                 if RESET = '1' then
                         DOI_FULL  <= (others => '0');
                         DOI       <= (others => '0');
                 else
                         DOI_FULL  <= REG_OUT(7 downto 0);
                         DOI       <= REG_OUT(6 downto 4);
                 end if;
          end if;
   end process;
end inside;
```

B.2.8 DC-Module

```
--------------------------------------------------------------------------
------ Depth Correction Module, Version 0.1,  13.09.00,  TS/IAAT ----------
--------------------------------------------------------------------------

---
-- DC (Depth Correction) /EC-Unit
---


--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- RNE_8         : Sub-RNE for sequential control from FD_IN-Module.
-- SOA_IN        : Sum of Anodes 16 Bit Signal from SOA-Module.
-- DOI_FULL      : Full 8 Bit DOI-Signal from DIV-Module.
--
------
-- Task:  Multiply SOA with DOI_FULL to the so called SD-Product (24 Bit) and
--        cut out the 8 Bit for Energy Output. Synchonized with RNE_8.
--
------
-- Output Signals:
--
-- ENERGY       : 8 Bit Energy Output Signal. DOI-Corrected Energy of
--                the Event-Packet, to EC_OUT-Module.
-- VD_DC        : Dummy VD-Signal, not used yet. Flag shall signal Errors to
--                Error-Detector (ED-Module) in future designs.
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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


entity DC is

        port( CLK          : in std_logic;
              RESET        : in std_logic;
              RNE_8        : in std_logic;
              SOA_IN       : in unsigned (15 downto 0);
              DOI_FULL     : in unsigned (7 downto 0);
              ENERGY       : out unsigned (7 downto 0);
              VD_DC        : out std_logic
            );
end DC;


architecture inside of DC is
```

```
begin

        ----------------------
        -- Energy Correction
        ----------------------
        energy_correction:
        process(CLK)
                variable REG_OP : unsigned (23 downto 0);
                variable last_RNE_8 : std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                REG_OP     := (others => '0');
                                ENERGY     <= (others => '0');
                                last_RNE_8 := '0';
                                VD_DC      <= '0';
                        else
                                if RNE_8 = '1' and last_RNE_8 = '0' then

                                        ---
                                        -- Linear correction ------------------
                                        ---
                                        REG_OP := DOI_FULL * SOA_IN;
                                        ENERGY <= REG_OP(19 downto 12);

                                        ---
                                        -- Generate Error Flag
                                        ---
                                        VD_DC <= '1';
                                        -- no Error Detection yet

                                end if;
                                last_RNE_8 := RNE_8;
                        end if;
                end if;
        end process;
end inside;
```

B.2.9 ED-Module

```
--------------------------------------------------------------------------
------ Error Detector of EC-Unit , Rev 0.1,  10.10.00,  TS/IAAT  ---------
--------------------------------------------------------------------------

---
-- ED (Error Detector) / EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- AC_IN : 138 Bit Event-Packet Remnant with Error-Flag (VD_Det,
--                 VD_AC).
--
-- AC_IN:
-- Hit 1     [ 137          124 : 123        117 :     116      :   115  ]
-- Hit 2     [ 114          101 : 100         94 :      93      :    92  ]
-- Hit 3     [  91           78 :  77         71 :      70      :    69  ]
-- Hit 4     [  68           55 :  54         48 :      47      :    46  ]
-- Hit 5     [  45           32 :  31         25 :      24      :    23  ]
-- Hit 6     [  22            9 :   8          2 :       1      :     0  ]
--
--
-- VD_ELA        : Valid Data Flag from EL-Module (Anode compare).
-- VD_ELC        : Valid Data Flag from EL-Module (Cathode compare).
-- VD_SOA        : Valid Data Flag from SOA-Module.
-- VD_SOC        : Valid Data Flag from SOC-Module.
-- VD_DIV        : Valid Data Flag from DIV-Module.
-- VD_TC         : Valid Data Flag from TC-Module.
-- VD_DC         : Valid Data Flag from DC-Module.
--
------
-- Task: Collecting Error information from all modules of the EC_Unit and from
-- Event-Packet Remnant. Generate Valid Data Bits for output Event-Packet from
-- this informations and send it to EC_OUT-Module.
--
------
-- Output Signals:
--
-- VD_DET        : Output Valid Data Flag of Event-Packet, carrying Detector
--                 Errors.
-- VD_EPP        : Output Valid Data Flag of Event-Packet, carrying Errors
--                 occurred in module of EC-Unit.
--
--------------------------------------------------------------------------



-- VHDL script starts here:



library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

library STD;
use STD.textio.all;

entity ED is

    port( CLK           : in std_logic;
          RESET         : in std_logic;
          AC_IN         : in unsigned (137 downto 0);
          VD_ELA        : in std_logic;
          VD_ELC        : in std_logic;
          VD_SOA        : in std_logic;
          VD_SOC        : in std_logic;
          VD_DIV        : in std_logic;
          VD_TC         : in std_logic;
          VD_DC         : in std_logic;
          VD_DET        : out std_logic;
          VD_EPP        : out std_logic
        );
end ED;


architecture inside of ED is


begin


   -----------------------
   -- Assemble Error Flags
   -----------------------
   error:
   process (CLK)
   begin
      if rising_edge (CLK) then
         if RESET = '1' then

            ---
            -- Reset Signals and Variables
            ---
            VD_DET <= '0';
            VD_EPP <= '0';
         else
            ---
            -- Generate  Error Flags
            ---
            VD_DET <= AC_IN(116) and AC_IN(93) and AC_IN(70) and AC_IN(47)
            and AC_IN(24) and AC_IN(1);

            VD_EPP <= VD_ELA and VD_ELC and VD_SOA and VD_SOC and VD_DIV
            and VD_TC and VD_DC and AC_IN(115) and AC_IN(92) and AC_IN(69)
            and AC_IN(46) and AC_IN(23) and AC_IN(0);
         end if;
      end if;
   end process;
end inside;
```

B.2.10 TC-Module

```
-----------------------------------------------------------------------
---------- Time Stamp Correction  Rev 0.1 , 12.09.00 TS/IAAT ----------
-----------------------------------------------------------------------

---
-- TC (Time Correction) / EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
-- REG_IN         : 16 Bit Timestamp from EC_IN-Module.
-- RNE_1          : Sub-RNE for sequential control from EC_IN-Module.
--
------
-- Task: Correction of the Timestamp of the Event-Packet. This Module
--       is still empty, no specifications are known for the time being.
--       The operation will run synchron with RNE_1.
--
--
------
-- Output Signals:
--
-- REG_OUT        : 16 Bit Output Timestamp corrected
-- VD_TC          : Valid Data Flag to ED-Module
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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


entity TC is
        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_1         : in std_logic;
                  REG_IN        : in unsigned (15 downto 0);
                  REG_OUT       : out unsigned (15 downto 0);
                  VD_TC         : out std_logic
                );
end TC;



architecture inside of TC is

begin
```

```
        ---
        -- Correction Procedure, for now doing nothing.
        ---
        time_correction:
        process (CLK)

        variable last_RNE_1: std_logic;

        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                REG_OUT    <= (others => '0');
                                last_RNE_1 := '0';
                                VD_TC      <= '1';
                        else
                                if RNE_1 ='1' and last_RNE_1 = '0' then
                                        REG_OUT <= REG_IN;
                                        VD_TC   <= '1';
                                end if;
                        end if;

                        last_RNE_1 := RNE_1;
                end if;
        end process;
end inside;
```

B.2.11 EC_OUT-Module

```
--------------------------------------------------------------------------
------ Output Register of EC_Unit , Rev 0.1,  12.09.00,  TS/IAAT  --------
--------------------------------------------------------------------------

---
-- EC_OUT / EC-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET       : Signals from EPP-TOP (Testbench).
-- RNE_9           : Sub-RNE for sequential control from EC_IN-Module.
-- TIME_IN         : Timestamp Signal from TC-Module.
-- LOC_A_IN        : 7 Bit Location of strongest Anode Hit from EL-Module.
-- LOC_C_IN        : 7 Bit Location of strongest Cathode Hit from EL-Module.
-- ENERGY_IN       : Corrected 8 Bit Energy Signal from DC-Module.
-- DOI_IN          : Reduced 3 Bit DOI-Signal from DC-Module.
-- VD_DET          : Valid Data Flag of Detector Errors from EPA-Unit.
-- VD_EPP          : Valid Data Flag of Calculation Errors from ED-Module.
--
------
-- Task: Compound Results of EC-Unit to a 45 Bit Output Event-Packet.
--       Ready with RNE_9.
--
------
-- Output Signals:
--
-- REG_OUT         : Output Event-Packet of EC-Unit to FD-Unit.
-- REG_OUT:        [ 44   Time-Stamp   29 : 28  Location Anode 22
--                 : 21 Location Cathode 15 : 14    Energy    7
--                 : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
--------------------------------------------------------------------------



-- VHDL script starts here:



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

library STD;
use STD.textio.all;

entity EC_OUT is

        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  RNE_9          : in std_logic;
                  TIME_IN        : in unsigned (15 downto 0);
```

```
                  LOC_A_IN      : in unsigned (6 downto 0);
                  LOC_C_IN      : in unsigned (6 downto 0);
                  ENERGY_IN     : in unsigned (7 downto 0);
                  DOI_IN        : in unsigned (2 downto 0);
                  VD_DET        : in std_logic;
                  VD_EPP        : in std_logic;
                  REG_OUT       : out unsigned (44 downto 0)
                );
end EC_OUT;

-------------------------------------------------------------------------


architecture inside of EC_OUT is


signal ERROR: unsigned (1 downto 0);

begin


        -----------------------
        -- Assemble Data Packet
        -----------------------
        assemble:
        process (CLK)

                variable last_RNE_9 : std_logic;

        begin
                if rising_edge (CLK) then
                        if RESET = '1' then

                                ---
                                -- Reset Signals and Variables
                                ---
                                REG_OUT    <= (others => '0');
                                last_RNE_9 := '0';
                                ERROR      <= (others => '0');
                        else
                                ---
                                -- Generate Data Output Packet
                                ---

                                if RNE_9 = '1' and last_RNE_9 = '0' then
                                        REG_OUT <= ( others => '0');
                                        REG_OUT(44 downto 29) <= TIME_IN;
                                        REG_OUT(28 downto 22) <= LOC_A_IN;
                                        REG_OUT(21 downto 15) <= LOC_C_IN;
                                        REG_OUT(14 downto  7) <= ENERGY_IN;
                                        REG_OUT(6  downto  4) <= DOI_IN;
                                        REG_OUT(3)            <= VD_DET;
                                        REG_OUT(2)            <= VD_EPP;
                                        REG_OUT(1  downto  0) <= ERROR;
                                end if;
                                last_RNE_9 := RNE_9;
                        end if;
                end if;
```

```
        end process;
end inside;
```

B.3 FD-Unit

```
----------------------------------------------------------------------------
------- Filter & Decision Unit  , Rev 0.4 , 20.11.00, TS/IAAT -------------
----------------------------------------------------------------------------


---
-- FD-UNIT / EPP-Core
---


---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
-- RNE            : Pipeline Signal from EPP-TOP. (Testbench, later EPA-Unit)
-- EVENT_IN_FD    : 45 Bit Input Event-Packet from EC-Unit.
--
------
-- Task:  Structural Design to connect the modules of the Filter & Decision-Unit
--        with each other, and the Filter & Decision-Unit with the TOP-Design
--        of the EPP.
--
--
------
-- Output Signals:
--
-- RINGB          : 41 Bit Output Event-Packet to SAVE_RB-Module (later to
--                  Ring-Buffer Unit).
-- RINGBCLK       : Trigger Signal to write Data to Disk-file (SAVE_RB-Module)
-- TELEM          : 45 Bit Output Event-Packet to SAVE_TM-Module (later to
--                  HBRI-Unit).
-- TELEMCLK       : Trigger Signal to write Data to Disk-file (SAVE_TM-Module).
-- BSN            : 41 Bit Output Event-Packet for diagnostics to SAVE_BS-Module.
-- BSNCLK         : Trigger Signal to write Data to Disk-file (SAVE_BS-Module).
-- TEST           : 46 Bit Output Event-Packet for diagnostics to SAVE_TST-Module.
-- BCNT           : 1 Bit Count Signal to BD-Unit for Count Rate Statistics.
--
--
---------------------------------------------------------------------------


-- VHDL script starts here:



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


entity FD_UNIT is
        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
```

```
                  RNE             : in std_logic;
                  EVENT_IN_FD     : in unsigned (44 downto 0);
                  RINGB           : out unsigned (40 downto 0);
                  RINGBCLK        : out std_logic;
                  TELEM           : out unsigned (44 downto 0);
                  TELEMCLK        : out std_logic;
                  BSN             : out unsigned (40 downto 0);
                  BSNCLK          : out std_logic;
                  TEST            : out unsigned (45 downto 0);
                  BCNT            : out std_logic
                );
end FD_UNIT;


architecture Structural of FD_UNIT is

        component FD_IN
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE           : in std_logic;
                          REG_IN        : in unsigned (44 downto 0);
                          RNE_1         : out std_logic;
                          RNE_2         : out std_logic;
                          RNE_3         : out std_logic;
                          RNE_4         : out std_logic;
                          RNE_5         : out std_logic;
                          RNE_6         : out std_logic;
                          REG_OUT       : out unsigned (44 downto 0)
                        );
        end component;


        component THN
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE_1         : in std_logic;
                          RNE_2         : in std_logic;
                          TH_IN_H       : in unsigned (7 downto 0);
                          TH_IN_L       : in unsigned (7 downto 0);
                          REG_IN        : in unsigned (44 downto 0);
                          REG_OUT       : out unsigned (45 downto 0)
                        );
        end component;

        component RAMC
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
                          RNE           : in std_logic;
                          TH_OUT_H      : out unsigned (7 downto 0);
                          TH_OUT_L      : out unsigned (7 downto 0);
                          BS_OUT_H      : out unsigned (7 downto 0);
                          BS_OUT_L      : out unsigned (7 downto 0)
                        );
        end component;

        component BS
                port    ( CLK           : in std_logic;
                          RESET         : in std_logic;
```

```
                  RNE_4         : in std_logic;
                  RNE_5         : in std_logic;
                  BS_IN_H       : in unsigned (7 downto 0);
                  BS_IN_L       : in unsigned (7 downto 0);
                  REG_IN        : in unsigned (40 downto 0);
                  RINGB_VD_IN   : in std_logic;
                  BSN_OUT       : out unsigned (40 downto 0);
                  BSN_VD_OUT    : out std_logic;
                  BCNT          : out std_logic
                );
        end component;

        component VD
                port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_3         : in std_logic;
                  REG_IN        : in unsigned (45 downto 0);
                  RINGB_OUT     : out unsigned (40 downto 0);
                  TELEM_OUT     : out unsigned (44 downto 0);
                  TEST_OUT      : out unsigned (45 downto 0);
                  RINGB_VD_OUT  : out std_logic;
                  TELEM_VD_OUT  : out std_logic
                );
        end component;

        component FD_OUT
                port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_6         : in std_logic;
                  RINGB_IN      : in unsigned (40 downto 0);
                  RINGB_VD_IN   : in std_logic;
                  TELEM_VD_IN   : in std_logic;
                  BSN_IN        : in unsigned (40 downto 0);
                  BSN_VD_IN     : in std_logic;
                  BCNT_IN       : in std_logic;
                  TELEM_IN      : in unsigned (44 downto 0);
                  TEST_IN       : in unsigned (45 downto 0);
                  RINGB_OUT     : out unsigned (40 downto 0);
                  RINGB_VD_OUT  : out std_logic;
                  TELEM_VD_OUT  : out std_logic;
                  TELEM_OUT     : out unsigned (44 downto 0);
                  TEST_OUT      : out unsigned (45 downto 0);
                  BSN_OUT       : out unsigned (40 downto 0);
                  BSN_VD_OUT    : out std_logic;
                  BCNT_OUT      : out std_logic
                );
        end component;




signal RNE_1            :        std_logic;
signal RNE_2            :        std_logic;
signal RNE_3            :        std_logic;
signal RNE_4            :        std_logic;
signal RNE_5            :        std_logic;
signal RNE_6            :        std_logic;
```

```
signal FDIN_DATAP_THN   :        unsigned (44 downto 0);
signal THN_DATAP_OUT    :        unsigned (45 downto 0);
signal RAMC_H_THN       :        unsigned (7 downto 0);
signal RAMC_L_THN       :        unsigned (7 downto 0);
signal RAMC_H_BS        :        unsigned (7 downto 0);
signal RAMC_L_BS        :        unsigned (7 downto 0);
signal VD_RINGB         :        unsigned (40 downto 0);
signal VD_TELEM_FDOUT   :        unsigned (44 downto 0);
signal VD_TEST_FDOUT    :        unsigned (45 downto 0);
signal VD_RINGBVD       :        std_logic;
signal VD_TVD_FDOUT     :        std_logic;
signal BS_BSN_FDOUT     :        unsigned (40 downto 0);
signal BS_BSNCLK_FDOUT  :        std_logic;
signal BS_BCNT_FDOUT    :        std_logic;

begin

        ---
        -- invoking Module: FD_IN (Input Controller)
        ---
        module_fdin: FD_IN
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  REG_IN        => EVENT_IN_FD,
                                  RNE_1         => RNE_1,
                                  RNE_2         => RNE_2,
                                  RNE_3         => RNE_3,
                                  RNE_4         => RNE_4,
                                  RNE_5         => RNE_5,
                                  RNE_6         => RNE_6,
                                  REG_OUT       => FDIN_DATAP_THN
                                );



        ---
        -- invoking Module: THN (Threshold Noise)
        ---
        module_thn: THN
        port map                ( CLK          => CLK,
                                  RESET        => RESET,
                                  RNE_1        => RNE_1,
                                  RNE_2        => RNE_2,
                                  TH_IN_H      => RAMC_H_THN,
                                  TH_IN_L      => RAMC_L_THN,
                                  REG_IN       => FDIN_DATAP_THN,
                                  REG_OUT      => THN_DATAP_OUT
                                );


        ---
        -- invoking Module: RAMC (RAM Controller)
        ---
        module_ramc: RAMC
        port map                ( CLK          => CLK,
                                  RESET        => RESET,
                                  RNE          => RNE,
```

```
                          TH_OUT_H     => RAMC_H_THN,
                          TH_OUT_L     => RAMC_L_THN,
                          BS_OUT_H     => RAMC_H_BS,
                          BS_OUT_L     => RAMC_L_BS
                        );

---
-- invoking Module: BS (Burst Snooper)
---
module_bs: BS
port map                ( CLK          => CLK,
                          RESET        => RESET,
                          RNE_4        => RNE_4,
                          RNE_5        => RNE_5,
                          BS_IN_H      => RAMC_H_BS,
                          BS_IN_L      => RAMC_L_BS,
                          REG_IN       => VD_RINGB,
                          RINGB_VD_IN  => VD_RINGBVD,
                          BSN_OUT      => BS_BSN_FDOUT,
                          BSN_VD_OUT   => BS_BSNCLK_FDOUT,
                          BCNT         => BS_BCNT_FDOUT
                        );


---
-- invoking Module: VD (Veto Detector)
---
module_vd: VD
port map                ( CLK          => CLK,
                          RESET        => RESET,
                          RNE_3        => RNE_3,
                          REG_IN       => THN_DATAP_OUT,
                          RINGB_OUT    => VD_RINGB,
                          TELEM_OUT    => VD_TELEM_FDOUT,
                          TEST_OUT     => VD_TEST_FDOUT,
                          RINGB_VD_OUT => VD_RINGBVD,
                          TELEM_VD_OUT => VD_TVD_FDOUT
                        );


---
-- invoking Module: FD_OUT (Output Controller)
---
module_fdout: FD_OUT
port map                ( CLK          => CLK,
                          RESET        => RESET,
                          RNE_6        => RNE_6,
                          RINGB_IN     => VD_RINGB,
                          RINGB_VD_IN  => VD_RINGBVD,
                          TELEM_VD_IN  => VD_TVD_FDOUT,
                          BSN_IN       => BS_BSN_FDOUT,
                          BSN_VD_IN    => BS_BSNCLK_FDOUT,
                          BCNT_IN      => BS_BCNT_FDOUT,
                          TELEM_IN     => VD_TELEM_FDOUT,
                          TEST_IN      => VD_TEST_FDOUT,
                          RINGB_OUT    => RINGB,
                          RINGB_VD_OUT => RINGBCLK,
                          TELEM_VD_OUT => TELEMCLK,
```

```
                                  TELEM_OUT    => TELEM,
                                  TEST_OUT     => TEST,
                                  BSN_OUT      => BSN,
                                  BSN_VD_OUT   => BSNCLK,
                                  BCNT_OUT     => BCNT
                                );

end Structural;
```

B.3.1 FD_IN-Module

```
------------------------------------------------------------------------
--------  Input Register of FD-UNIT , Rel 0.2, 21.09.00,  TS/IAAT --------
------------------------------------------------------------------------

---
-- FD_IN / FD-Unit
---

------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE           : Pipeline Tag from EPA_UNIT (Testbench)
-- REG_IN        : Input Event-Packet of the FD-Unit.
--
-- REG_IN:       [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14    Energy    7
--               : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
------
-- Task:     Copy and hold Event-Packet from EC-Unit synchron with RNE-Signal.
--           Processing of Sub-RNE Signals for internal sequential control of
--           the FD-Unit.
--
------
-- Output Signals:
--
-- REG_OUT       : Event-Packed Output for use in Modules inside the FD-Unit.
--
-- REG_OUT:      [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14    Energy    7
--               : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
-- RNE_1/RNE_6   : Sub-RNE Signals for sequential control inside the
--                 FD-Unit.
--

------------------------------------------------------------------------


-- VHDL script starts here:



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



entity FD_IN is

        port( CLK            : in std_logic;
              RESET          : in std_logic;
```

```
              RNE            : in std_logic;
              REG_IN         : in unsigned (44 downto 0);
              RNE_1          : out std_logic;
              RNE_2          : out std_logic;
              RNE_3          : out std_logic;
              RNE_4          : out std_logic;
              RNE_5          : out std_logic;
              RNE_6          : out std_logic;
              REG_OUT        : out unsigned (44 downto 0)
            );
end FD_IN;


architecture inside of FD_IN is


begin



        ----------------------------------
        -- Copy Input Data Packet with RNE
        ----------------------------------
        copy:
        process (CLK)

                variable last_RNE : std_logic;


        -- begin of process

        begin

                if rising_edge (CLK) then
                        if RESET = '1' then
                                ---
                                -- Reset Signals and Variables
                                ---
                                last_RNE := '0';
                                REG_OUT <= (others => '0');
                        else
                                ---
                                -- Load Event-Packet from Input
                                ---
                                if RNE = '1' and last_RNE ='0' then
                                        REG_OUT <= REG_IN;
                                end if;
                                last_RNE := RNE;
                        end if;
                end if;
        end process;
```

```
--------------------------
-- Generating Sub-RNEs
--------------------------
sub_rne:
process (CLK)

        variable CNT: integer range 0 to 127;
        variable last_RNE: std_logic;

begin

        if rising_edge(CLK) then
                if RESET = '1' then
                        RNE_1 <= '0';
                        RNE_2 <= '0';
                        RNE_3 <= '0';
                        RNE_4 <= '0';
                        RNE_5 <= '0';
                        RNE_6 <= '0';
                        CNT := 0;
                else
                        if RNE = '1' then
                                CNT := 0;
                                RNE_1 <= '0';
                                RNE_2 <= '0';
                                RNE_3 <= '0';
                                RNE_4 <= '0';
                                RNE_5 <= '0';
                                RNE_6 <= '0';
                        else
                                if CNT < 100 then
                                        CNT := CNT + 1;
                                        if CNT = 12 then
                                                RNE_1 <= '1';
                                        else
                                                RNE_1 <= '0';
                                        end if;
                                        if CNT = 24 then
                                                RNE_2 <= '1';
                                        else
                                                RNE_2 <= '0';
                                        end if;
                                        if CNT = 36 then
                                                RNE_3 <= '1';
                                        else
                                                RNE_3 <= '0';
                                        end if;
                                        if CNT = 48 then
                                                RNE_4 <= '1';
                                        else
                                                RNE_4 <= '0';
                                        end if;
                                        if CNT = 60 then
                                                RNE_5 <= '1';
                                        else
                                                RNE_5 <= '0';
                                        end if;
                                        if CNT = 72 then
```

```
                                                        RNE_6 <= '1';
                                                else
                                                        RNE_6 <= '0';
                                                end if;
                                        end if;
                                end if;
                        end if;
                end if;
        end process;
end inside;
```

B.3.2 THN-Module

```
--------------------------------------------------------------------------
-- Threshold Energy for Noise rejection  Rel. 0.5 , 19.09.00  TS/IAAT ----
--------------------------------------------------------------------------

---
-- THN (Threshold Noise) / FD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE_1/RNE_2   : Sub-RNE for sequential control from FD_IN-Module
-- TH_IN_H/L     : Upper and lower Threshold for Noise-Energy-Filter from
--                 LOAD-Module (Testbench). The LOAD-Module reads this numbers
--                 from disk file RAM.dat
-- REG_IN        : Event-Packet input from EC-Unit
--
-- REG_IN:       [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14  Energy  7
--               : 6 DOI 4 : 3 VD-Det : 2 VD-EPP : 1 Error 0 ]
--
------
-- Task:         : Filter for Energy-Threshold for Noise Signals.
--        RNE_1 : Compare Energy-Entry of Input-Event-Packet with
--                 Noise-Threshold. If Energy is outside, VET1/2 will set
--                 to 1, otherwise to 0.
--        RNE_2 : Set Valid Data Flag (Noise) according to result of
--                 VET1 and VET2 values. If VD-Noise is 0, the
--                 Output Event-Packet will be deleted in Veto Detector
--                 (VD-Module).
--
------
-- Output Signals:
--
-- REG_OUT       : 45 Bit Output Event-Packet with additional VD-Flag for
--                 result of noise-filter-operation. -> to Veto Detector.
--
-- REG_OUT:      [ 45   Time-Stamp   30 : 29  Location Anode 23
--               : 22 Location Cathode 16 : 15  Energy  8 : 7 DOI 5
--               : 4 VD-Det : 3 VD-EPP : 2 VD-Noise : 1 Error 0 ]
--
--------------------------------------------------------------------------



-- VHDL script starts here:



library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
```

```
entity THN is
        port( CLK            : in std_logic;
              RESET          : in std_logic;
              RNE_1          : in std_logic;
              RNE_2          : in std_logic;
              TH_IN_H        : in unsigned (7 downto 0);
              TH_IN_L        : in unsigned (7 downto 0);
              REG_IN         : in unsigned (44 downto 0);
              REG_OUT        : out unsigned (45 downto 0)
            );
end THN;

architecture inside of THN is

signal VETO_TH : std_logic;

begin
       ---
       -- threshold noise
       ---
       threshold:
       process (CLK)
              variable VET1       : std_logic;
              variable VET2       : std_logic;
              variable last_RNE_1 : std_logic;
       begin
              if rising_edge(CLK) then
                     if RESET = '1' then
                            VET1        := '0';
                            VET2        := '0';
                            VETO_TH     <= '0';
                            last_RNE_1  := '0';
                     else
                            if RNE_1 = '1' and last_RNE_1 = '0' then
                                   if REG_IN(14 downto 7) < TH_IN_L then
                                          VET1 := '1';
                                   else
                                          VET1 := '0';
                                   end if;

                                   if REG_IN(14 downto 7) > TH_IN_H then
                                          VET2 := '1';
                                   else
                                          VET2 := '0';
                                   end if;

                                   VETO_TH <= VET1 or VET2;
                            end if;
                            last_RNE_1 := RNE_1;
                     end if;
              end if;
       end process;

       ---
       -- Set Threshold Flag
       ---
       setflag:
       process (CLK)
```

```
        variable last_RNE_2 : std_logic;
    begin
        if rising_edge(CLK) then
            if RESET = '1' then
                last_RNE_2           := '0';
                REG_OUT(0)           <= '1';
                REG_OUT(45 downto 1) <= REG_IN(44 downto 0);
            else
                if RNE_2 = '1' and last_RNE_2 = '0' then
                    REG_OUT(45 downto 3) <= REG_IN(44 downto 2);
                    REG_OUT(1 downto 0)  <= REG_IN(1 downto 0);
                    if VETO_TH = '1' then
                        REG_OUT(2) <= '0';
                    else
                        REG_OUT(2) <= '1';
                    end if;
                end if;
                last_RNE_2 := RNE_2;
            end if;
        end if;
    end process;
end inside;
```

B.3.3 VD-Module

```
-------------------------------------------------------------------------------
-- Veto Event Packets for VD=0 or Noise Flag set, Rev 0.5, 19.09.00, TS/IAAT -
-------------------------------------------------------------------------------

---
-- VD (Veto Detector) / FD-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE_3         : Sub-RNE for sequential control from FD_IN-Module
-- REG_IN        : 45 Bit Event-Packet from Veto-Detector (VD-Module)
--                 with full Error Flags.
--
-- REG_IN:       [ 45   Time-Stamp   30 : 29  Location Anode 23
--               : 22 Location Cathode 16 : 15    Energy    8
--               : 7 DOI 5 : 4 VD-DET : 3 VD-EPP : 2 VD-Noise : 1 ERROR 0 ]
--
------
-- Task:         : Vetos all Event-Packets from THN-Module with any Valid-Data
--                 Bits set to Zero. Works synchron with RNE_3.
--
------
-- Output Signals:
--
-- RINGB_OUT     : 41 Bit Event-Packet to FD_OUT-Module.
--
-- RINGB_OUT:    [ 41   Time-Stamp   25 : 24  Location Anode 18
--               : 17 Location Cathode 11 : 10    Energy    3
--               : 2 DOI 0 ]
--
-- TELEM_OUT     : 45 Bit Event-Packet to FD_OUT-Module.
--
-- TELEM_OUT:    [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14    Energy    7
--               : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
-- TEST_OUT      : 46 Bit Event-Packet to FD_OUT-Module (for diagnostics)
--
-- TEST_OUT:     [ 45   Time-Stamp   30 : 29  Location Anode 23
--               : 22 Location Cathode 16 : 15    Energy    8
--               : 7 DOI 5 : 4 VD-DET : 3 VD-EPP : 2 VD-Noise : 1 ERROR 0 ]
--
-- RINGB_VD_OUT  : Valid Data Signal to FD_OUT-Module.
-- TELEM_VD_OUT  : Valid Data Signal to FD_OUT-Module.
--
---------------------------------------------------------------------------



-- VHDL script starts here:
```

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


entity VD is
    port    ( CLK            : in std_logic;
              RESET          : in std_logic;
              RNE_3          : in std_logic;
              REG_IN         : in unsigned (45 downto 0);
              RINGB_OUT      : out unsigned (40 downto 0);
              TELEM_OUT      : out unsigned (44 downto 0);
              TEST_OUT       : out unsigned (45 downto 0);
              RINGB_VD_OUT   : out std_logic;
              TELEM_VD_OUT   : out std_logic
            );
end VD;

architecture inside of VD is

begin

   ---
   -- Filter Veto Signals out
   ---
   veto_filter:
   process (CLK)
          variable last_RNE_3 : std_logic;
   begin
        if rising_edge(CLK) then
             if RESET = '1' then
                    TELEM_OUT      <= (others => '0');
                    RINGB_OUT      <= (others => '0');
                    TEST_OUT       <= (others => '0');
                    RINGB_VD_OUT   <= '0';
                    TELEM_VD_OUT   <= '0';
                    last_RNE_3     := '0';
             else
                    if RNE_3 = '1' and last_RNE_3 = '0' then
                         TEST_OUT <= REG_IN;
                         if REG_IN(2) = '1' then
                              TELEM_OUT(44 downto 2) <= REG_IN(45 downto 3);
                              TELEM_OUT(1  downto 0) <= REG_IN(1  downto 0);
                              TELEM_VD_OUT           <= '1';

                              if REG_IN(3) = '1' and REG_IN(4) = '1' then
                                   RINGB_OUT(40 downto 0) <= REG_IN(45 downto 5);
                                   RINGB_VD_OUT           <= '1';
                              else
                                   RINGB_VD_OUT           <= '0';
                              end if;
                         else
                              TELEM_VD_OUT <= '0';
                         end if;
                    end if;
                    last_RNE_3 := RNE_3;
             end if;
        end if;
```

```
   end process;
end inside;
```

B.3.4 BS-Module

```
------------------------------------------------------------------------
------ Set a Burst observation window  , Rev 0.5 , 19.09.00  TS/IAAT -----
------------------------------------------------------------------------

---
-- BS (Burst Snooper) / FD-Unit
---

------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench)
-- RNE_4/RNE_5    : Sub-RNE for sequential control from FD_IN-Module
-- BS_IN_H/L      : Upper and lower Threshold for Energy-Filter from LOAD-Module
--                  (Testbench). The LOAD-Module reads this numbers from disk
--                  file RAM.dat
-- REG_IN         : Reduced Event-Packet from Veto-Detector (VD-Module) without
--                  Error Flags.
--
-- REG_IN:        [ 40   Time-Stamp   25 : 24  Location Anode 18
--                : 17 Location Cathode 11 : 10  Energy  3 : 2 DOI 1]
--
-- RINGB_VD_IN    : If Event-Packet matches Veto-Detector criteria (is valid)
--                  RINGB_VD_IN is 1. From VD-Module.
--
------
-- Task:          : Filter (Energy-Threshold) of Burst-Counter.
--         RNE_4 : Compare Energy-Entry of Input-Event-Packet with
--                  Energy Threshold. If Energy is outside, VET3 will set
--                  to 1, otherwise to 0.
--         RNE_5 : Look for Valid-Data-Flag of Veto-Detector and for VET3
--                  and decide if current Event-Packet is valid and should
--                  be counted in BD-Unit.
--
------
-- Output Signals:
--
-- BSN_OUT       :  Output Event-Packet for diagnostics. All Packets that
--                  matches Energy criteria will be written to SAVE_BSN-Module.
-- BSN_VD_OUT    :  Trigger the SAVE_BSN-Module for Output.
-- BCNT          :  1 Bit Output-Signal for BD-Unit / BD_IN-Module. Indicate
--                  if a Count Impulse should be generated in BD_IN-Module
--                  for the current Event-Packet.
--
------------------------------------------------------------------------



-- VHDL script starts here:



library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;


entity BS is
        port( CLK            : in std_logic;
              RESET          : in std_logic;
              RNE_4          : in std_logic;
              RNE_5          : in std_logic;
              BS_IN_H        : in unsigned (7 downto 0);
              BS_IN_L        : in unsigned (7 downto 0);
              REG_IN         : in unsigned (40 downto 0);
              RINGB_VD_IN    : in std_logic;
              BSN_OUT        : out unsigned (40 downto 0);
              BSN_VD_OUT     : out std_logic;
              BCNT           : out std_logic
            );
end BS;

architecture inside of BS is

signal VETO_BS : std_logic;

begin
        ---
        -- threshold burst
        ---
        threshold:
        process (CLK)
                variable VET3       : std_logic;
                variable VET4       : std_logic;
                variable last_RNE_4 : std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                VET3       := '0';
                                VET4       := '0';
                                VETO_BS    <= '0';
                                last_RNE_4 := '0';
                        else
                                if RNE_4 = '1' and last_RNE_4 = '0' then
                                        if REG_IN(10 downto 3) < BS_IN_L then
                                                VET3 := '1';
                                        else
                                                VET3 := '0';
                                        end if;

                                        if REG_IN(10 downto 3) > BS_IN_H then
                                                VET4 := '1';
                                        else
                                                VET4 := '0';
                                        end if;

                                        VETO_BS <= VET3 or VET4;
                                end if;
                                last_RNE_4 := RNE_4;
                        end if;
                end if;
```

```
        end process;

        ---
        -- Set  Flag
        ---
        setflag:
        process (CLK)
                variable last_RNE_5 : std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                last_RNE_5  := '0';
                                BSN_VD_OUT  <= '0';
                                BSN_OUT     <= (others => '0');
                                BCNT        <= '0';
                        else
                                if RNE_5 = '1' and last_RNE_5 = '0' then
                                        if VETO_BS = '0' and RINGB_VD_IN ='1' then
                                                BSN_VD_OUT  <= '1';
                                                BSN_OUT     <= REG_IN;
                                                BCNT        <= '1';
                                        else
                                                BSN_VD_OUT  <= '0';
                                                BSN_OUT     <= REG_IN;
                                                BCNT        <= '0';
                                        end if;
                                end if;
                                last_RNE_5 := RNE_5;
                        end if;
                end if;
        end process;
end inside;
```

B.3.5 RAMC-Module

```
--------------------------------------------------------------------------
---  RAM Controller for Threshold  Data , Rel 0.5, 19.09.00, TS/IAAT -----
--------------------------------------------------------------------------

---
-- RAMC / FD-Unit
---


--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench)
-- RNE            : Pipeline Tag from EPA_UNIT (Testbench)
-- REG_IN         : Input Event-Packet of the FD-Unit.
--
--
------
-- Task:     Read Energy Thresholds from Disk-File RAM.dat and distribute
--           it to VD- and BS-Module.
--
------
-- Output Signals:
--
-- TH_OUT_H       : Upper Energy value of the Noise-Filter, to VD-Module.
-- TH_OUT_L       : Lower Energy value of the Noise-Filter, to VD-Module.
-- BS_OUT_H       : Upper Energy value of the Burst-Filter, to BS-Module.
-- BS_OUT_L       : Lower Energy value of the Burst-Filter, to BS-Module.
--

--------------------------------------------------------------------------



-- VHDL script starts here:



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

library STD;
use STD.textio.all;

entity RAMC is

        port    ( CLK            : in std_logic;
                  RESET          : in std_logic;
                  RNE            : in std_logic;
                  TH_OUT_H       : out unsigned (7 downto 0);
                  TH_OUT_L       : out unsigned (7 downto 0);
                  BS_OUT_H       : out unsigned (7 downto 0);
                  BS_OUT_L       : out unsigned (7 downto 0)
```

```
                );
end RAMC;


architecture inside of RAMC is


begin


        --------------------------------
        -- load -Threshold Data from RAM
        --------------------------------
        load_TH_Data:
        process (CLK)
                variable d_th_h   : integer;
                variable d_th_l   : integer;
                variable d_bs_h   : integer;
                variable d_bs_l   : integer;
                variable last_RNE : std_logic;
                variable done     : boolean := FALSE;

                file data: text is in "stimuli/RAM.dat";
                -- Procedure, load Data from LookUpTable Datafile
                procedure read_data_line is
                        variable L : line;
                begin
                        if (not endfile(data)) then
                                readline(data, L);
                                read(L, d_th_h);
                                read(L, d_th_l);
                                read(L, d_bs_h);
                                read(L, d_bs_l);
                        else
                                done := TRUE;
                        end if;
                end;
        -- begin of process
        begin
                if rising_edge (CLK) then
                        if RESET = '1' then
                                ---
                                -- Reset Signals and Variables
                                ---
                                d_th_h   := 0;
                                d_th_l   := 0;
                                d_bs_h   := 0;
                                d_bs_l   := 0;
                                TH_OUT_H <= (others => '0');
                                TH_OUT_L <= (others => '0');
                                BS_OUT_H <= (others => '0');
                                BS_OUT_L <= (others => '0');
                                last_RNE := '0';
                        else
                                ---
                                -- Load RAM Data from File to Register
                                ---
                                if RNE = '1' and last_RNE = '0' then
```

```
                                        read_data_line;
                                        if not done then
                                                TH_OUT_H <= CONV_UNSIGNED(d_th_h, 8);
                                                TH_OUT_L <= CONV_UNSIGNED(d_th_l, 8);
                                                BS_OUT_H <= CONV_UNSIGNED(d_bs_h, 8);
                                                BS_OUT_L <= CONV_UNSIGNED(d_bs_l, 8);
                                        end if;
                                end if;
                                last_RNE := RNE;
                        end if;
                end if;
        end process;
end inside;
```

B.3.6 FD_OUT-Module

```
--------------------------------------------------------------------------
----  Output Register of FD-Unit , Version 0.2,  21.09.00,  TS/IAAT ------
--------------------------------------------------------------------------

---
-- FD_OUT / FD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE_6         : Sub_RNE Signal from FD_IN-Module.
-- RINGB_IN      : 41 Bit Input Event-Packet from Veto-Detector (VD-Module).
--
-- RINGB_IN:     [ 41   Time-Stamp   25 : 24  Location Anode 18
--               : 17 Location Cathode 11 : 10    Energy    3
--               : 2 DOI 0 ]
--
-- RINGB_VD_IN   : Ringbuffer Valid-Data Signal from VD-Module.
-- TELEM_VD_IN   : Telemetry Valid-Data Signal from VD-Module.
-- BSN_IN        : 41 Bit Input Event-Packet from BS-Module (for diagnostics).
--
-- BSN_IN:       [ 41   Time-Stamp   25 : 24  Location Anode 18
--               : 17 Location Cathode 11 : 10    Energy    3
--               : 2 DOI 0 ]
--
-- BSN_VD_IN     : Valid-Data Bit that signals if Event-Packet BSN_IN has
--                 passed the energy criteria of the Burst-Snooper.
-- BCNT_IN       : Count Signal for each Event-Packet that passed the energy
--                 criteria from Burst-Snooper.
-- TELEM_IN      : 45 Bit full Event-Packet from VD-Module.
--
-- TELEM_IN:     [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14    Energy    7
--               : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
-- TEST_IN       : 46 Bit Event-Packet with Noise-Flag for diagnostics from
--                 VD-Module.
--
-- TEST_IN:      [ 45   Time-Stamp   30 : 29  Location Anode 23
--               : 22 Location Cathode 16 : 15    Energy    8
--               : 7 DOI 5 : 4 VD-DET : 3 VD-EPP : 2 VD-Noise : 1 ERROR 0 ]
--
--
------
-- Task:     Interface between internal Signals of FD-Unit and Port-Signals
--           of the FD-Unit Output.
--
------
-- Output Signals:
--
-- RINGB_OUT     : Event-Packet Output to SAVE_RB-Module. Same format as
--                 RINGB_IN.
```

```
-- RINGB_VD_OUT  : Valid Data Bit, trigger for SAVE_RB-Module.
-- TELEM_VD_OUT  : Valid Data Bit, trigger for SAVE_TM-Module.
-- TELEM_OUT     : Event-Packet Output to SAVE_TM-Module. Same format as
--                 TELEM_IN.
-- TEST_OUT      : Event-Packet Output to SAVE_TST-Module. Same format as
--                 TEST_IN.
-- BSN_OUT       : Event-Packet Output to SAVE_BS-Module. Same format as
--                 BSN_IN.
-- BSN_VD_OUT    : Valid Data Bit, trigger for SAVE_BS-Module.
-- BCNT_OUT      : Count signal output to SAVE_BD-Module.
--
--

---------------------------------------------------------------------------



-- VHDL script starts here:



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


entity FD_OUT is

        port    ( CLK           : in std_logic;
                  RESET         : in std_logic;
                  RNE_6         : in std_logic;
                  RINGB_IN      : in unsigned (40 downto 0);
                  RINGB_VD_IN   : in std_logic;
                  TELEM_VD_IN   : in std_logic;
                  BSN_IN        : in unsigned (40 downto 0);
                  BSN_VD_IN     : in std_logic;
                  BCNT_IN       : in std_logic;
                  TELEM_IN      : in unsigned (44 downto 0);
                  TEST_IN       : in unsigned (45 downto 0);
                  RINGB_OUT     : out unsigned (40 downto 0);
                  RINGB_VD_OUT  : out std_logic;
                  TELEM_VD_OUT  : out std_logic;
                  TELEM_OUT     : out unsigned (44 downto 0);
                  TEST_OUT      : out unsigned (45 downto 0);
                  BSN_OUT       : out unsigned (40 downto 0);
                  BSN_VD_OUT    : out std_logic;
                  BCNT_OUT      : out std_logic
                );
end FD_OUT;


architecture inside of FD_OUT is

begin

        -----------------------
        -- DATA OUT
        -----------------------
```

```
        data_out:
        process (CLK)

                variable last_RNE_6     : std_logic;

        begin
                if rising_edge (CLK) then
                        if RESET = '1' then

                                ---
                                -- Reset Signals and Variables
                                ---
                                RINGB_OUT       <= (others => '0');
                                RINGB_VD_OUT    <= '0';
                                TELEM_OUT       <= (others => '0');
                                TELEM_VD_OUT    <= '0';
                                TEST_OUT        <= (others => '0');
                                BSN_OUT         <= (others => '0');
                                BSN_VD_OUT      <= '0';
                                BCNT_OUT        <= '0';
                                last_RNE_6      := '0';
                        else
                                ---
                                -- Data Output
                                ---

                                if RNE_6 = '1' and last_RNE_6 = '0' then
                                        RINGB_OUT       <= RINGB_IN;
                                        RINGB_VD_OUT    <= RINGB_VD_IN;
                                        TELEM_VD_OUT    <= TELEM_VD_IN;
                                        TELEM_OUT       <= TELEM_IN;
                                        TEST_OUT        <= TEST_IN;
                                        BSN_OUT         <= BSN_IN;
                                        BSN_VD_OUT      <= BSN_VD_IN;
                                        BCNT_OUT        <= BCNT_IN;
                                end if;
                                last_RNE_6 := RNE_6;
                        end if;
                end if;
        end process;
end inside;
```

B.4 BD-Unit

```
---------------------------------------------------------------
---- Burst Detection Unit  , Rev 0.1 , 06.10.00, TS/IAAT ------
---------------------------------------------------------------

---
-- BD-UNIT / EPP-Core
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
-- RNE          : Pipeline Signal from EPP-TOP. (Testbench, later EPA-Unit)
-- GATE_SWITCH    : 2 Bit input signal, to set GATE Timeframe (00 = 1/8s,
--                  01 = 1/2s, 10=1s, 11= 2s.
-- BCNT           : Count Signal Input from Burst-Snooper-Module (FD-Unit).
--
------
-- Task:  Structural Design to connect the modules of the Burst-Detection-Unit
--        with each other, and the Burst-Detection-Unit with the TOP-Design
--        of the EPP.
--
--
------
-- Output Signals:
--
-- BCNTTST_OUT    : Test Output for diagnostic, to SAVE_BD-Module.
-- AVTST_OUT      : Test Output for diagnostic, to SAVE_BD-Module.
-- SQR_OUT        : Test Output for diagnostic, to SAVE_BD-Module.
-- SIG_OUT        : Test Output for diagnostic, to SAVE_BD-Module.
-- GATE_CLK       : Clock signal to control writing data in Save_BD-Modules.
-- TRIGGER        : Burst-Trigger-Out to SAVE_BD-Module.
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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


entity BD_UNIT is

         generic ( n              : in positive;
                   gfreq          : in integer
                  );
         port    ( CLK            : in std_logic;
                   RESET          : in std_logic;
                   RNE            : in std_logic;
```

```
                   GATE_SWITCH    : in unsigned (1 downto 0);
                   BCNT           : in std_logic;
                   BCNTTST_OUT    : out unsigned((4*n)-1 downto 0);
                   AVTST_OUT      : out unsigned((4*n)-1 downto 0);
                   SQR_OUT        : out unsigned((4*n)-1 downto 0);
                   SIG_OUT        : out unsigned((4*n)-1 downto 0);
                   GATE_CLK       : out std_logic;
                   TRIGGER        : out std_logic
                 );
end BD_UNIT;


architecture Structural of BD_UNIT is


        component BD_IN
               port    ( CLK            : in std_logic;
                         RESET          : in std_logic;
                         RNE            : in std_logic;
                         BCNT_IN        : in std_logic;
                         BCNT_OUT       : out std_logic
                       );
        end component;

        component GC
               generic ( n              : in positive;
                         gfreq          : in positive
                       );
               port    ( CLK            : in std_logic;
                         RESET          : in std_logic;
                         GATE_SWITCH    : in unsigned (1 downto 0);
                         GATE_TIME      : out std_logic
                       );
        end component;

        component GL
               port    ( CLK            : in std_logic;
                         RNE            : in std_logic;
                         RESET          : in std_logic;
                         GATE_TIME      : in std_logic;
                         SWITCH         : out std_logic;
                         WRITE_OUT      : out std_logic;
                         CALC_OUT       : out std_logic;
                         CALC2_OUT      : out std_logic;
                         CALC3_OUT      : out std_logic;
                         CALC4_OUT      : out std_logic;
                         CALC5_OUT      : out std_logic;
                         CALC6_OUT      : out std_logic;
                         CALC7_OUT      : out std_logic;
                         CALC8_OUT      : out std_logic;
                         RESET1_OUT     : out std_logic;
                         RESET2_OUT     : out std_logic;
                         GATE_CLK       : out std_logic
                       );
        end component;

        component GATE
               port    ( RESET          : in std_logic;
```

```
                 SWITCH         : in std_logic;
                 BCNT_IN        : in std_logic;
                 BCNT1_OUT      : out std_logic;
                 BCNT2_OUT      : out std_logic
               );
end component;

component CNT1
       generic ( n              : in positive);
       port    ( CLK            : in std_logic;
                 CNT1_IN        : in std_logic;
                 RESET          : in std_logic;
                 RESET_SET      : in std_logic;
                 REG1_OUT       : out unsigned ((4*n)-1 downto 0)
               );
end component;

component CNT2
       generic ( n              : in positive);
       port    ( CLK            : in std_logic;
                 CNT2_IN        : in std_logic;
                 RESET          : in std_logic;
                 RESET_SET      : in std_logic;
                 REG2_OUT       : out unsigned ((4*n)-1 downto 0)
               );
end component;

component SWITCH
       generic ( n              : in positive);
       port    ( RESET          : in std_logic;
                 SWITCH         : in std_logic;
                 COUNT1_IN      : in unsigned ((4*n)-1 downto 0);
                 COUNT2_IN      : in unsigned ((4*n)-1 downto 0);
                 COUNT_OUT      : out unsigned ((4*n)-1 downto 0)
               );
end component;

component AV
       generic ( n              : in positive);
       port    ( CLK            : in std_logic;
                 WRITE          : in std_logic;
                 RESET          : in std_logic;
                 CALC           : in std_logic;
                 COUNT_IN       : in unsigned ((4*n)-1 downto 0);
                 COUNT_OUT      : out unsigned ((4*n)-1 downto 0);
                 TEST_OUT       : out unsigned ((4*n)-1 downto 0);
                 AV_OUT         : out unsigned ((4*n)-1 downto 0);
                 AVTST_OUT      : out unsigned ((4*n)-1 downto 0)
               );
end component;

component CMP
       generic ( n              : in positive);
       port    ( CLK            : in std_logic;
                 RESET          : in std_logic;
                 COUNT_IN       : in unsigned ((4*n)-1 downto 0);
                 AV_IN          : in unsigned ((4*n)-1 downto 0);
                 CALC2          : in std_logic;
```

```
                        CALC3          : in std_logic;
                        CALC4          : in std_logic;
                        CALC5          : in std_logic;
                        CALC6          : in std_logic;
                        CALC7          : in std_logic;
                        CALC8          : in std_logic;
                        SQR_OUT        : out unsigned ((4*n)-1 downto 0);
                        SIG_OUT        : out unsigned ((4*n)-1 downto 0);
                        TRIGGER        : out std_logic
                      );
        end component;



signal GC_GATE_TIME_GL   : std_logic;
signal GL_SWITCH         : std_logic;
signal GL_WRITE_AV       : std_logic;
signal GL_CALC_AV        : std_logic;
signal GL_CALC2_CMP      : std_logic;
signal GL_CALC3_CMP      : std_logic;
signal GL_CALC4_CMP      : std_logic;
signal GL_CALC5_CMP      : std_logic;
signal GL_CALC6_CMP      : std_logic;
signal GL_CALC7_CMP      : std_logic;
signal GL_CALC8_CMP      : std_logic;
signal GL_RESET_CNT1     : std_logic;
signal GL_RESET_CNT2     : std_logic;
signal GATE_CNT_CNT1     : std_logic;
signal GATE_CNT_CNT2     : std_logic;
signal BDIN_BCNT_GATE    : std_logic;
signal CNT1_REG_SWITCH   : unsigned ((4*n)-1 downto 0);
signal CNT2_REG_SWITCH   : unsigned ((4*n)-1 downto 0);
signal SWITCH_CNT        : unsigned ((4*n)-1 downto 0);
signal AV_BCNT_CMP       : unsigned ((4*n)-1 downto 0);
signal AV_OUT            : unsigned ((4*n)-1 downto 0);

begin
        ---
        -- invoking Module: BD_IN (Input Module)
        ---
        module_bdin: BD_IN
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE           => RNE,
                                  BCNT_IN       => BCNT,
                                  BCNT_OUT      => BDIN_BCNT_GATE
                                );

        ---
        -- invoking Module: GC (Gate Counter)
        ---
        module_gc: GC
                generic map     ( n             => n,
                                  gfreq         => gfreq
                                )
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
```

```
                          GATE_SWITCH   => GATE_SWITCH,
                          GATE_TIME     => GC_GATE_TIME_GL
                        );



---
-- invoking Module: GL (Gate Logic)
---
module_gl: GL
        port map        ( CLK           => CLK,
                          RNE           => RNE,
                          RESET         => RESET,
                          GATE_TIME     => GC_GATE_TIME_GL,
                          SWITCH        => GL_SWITCH,
                          WRITE_OUT     => GL_WRITE_AV,
                          CALC_OUT      => GL_CALC_AV,
                          CALC2_OUT     => GL_CALC2_CMP,
                          CALC3_OUT     => GL_CALC3_CMP,
                          CALC4_OUT     => GL_CALC4_CMP,
                          CALC5_OUT     => GL_CALC5_CMP,
                          CALC6_OUT     => GL_CALC6_CMP,
                          CALC7_OUT     => GL_CALC7_CMP,
                          CALC8_OUT     => GL_CALC8_CMP,
                          RESET1_OUT    => GL_RESET_CNT1,
                          RESET2_OUT    => GL_RESET_CNT2,
                          GATE_CLK      => GATE_CLK
                        );


---
-- invoking Module: GATE (Gate Switch)
---
module_gate: GATE
        port map        ( RESET         => RESET,
                          SWITCH        => GL_SWITCH,
                          BCNT_IN       => BDIN_BCNT_GATE,
                          BCNT1_OU      => GATE_CNT_CNT1,
                          BCNT2_OU      => GATE_CNT_CNT2
                        );


---
-- invoking Module: CNT1 (Counter 1)
---
module_cnt1: CNT1
        generic map     ( n             => n)
        port map(         CLK           => CLK,
                          CNT1_IN       => GATE_CNT_CNT1,
                          RESET         => RESET,
                          RESET_SET     => GL_RESET_CNT1,
                          REG1_OUT      => CNT1_REG_SWITCH
                        );


---
-- invoking Module: CNT2 (Counter 2)
---
```

```
module_cnt2: CNT2
        generic map     ( n              => n)
        port map        ( CLK            => CLK,
                          CNT2_IN        => GATE_CNT_CNT2,
                          RESET          => RESET,
                          RESET_SET      => GL_RESET_CNT2,
                          REG2_OUT       => CNT2_REG_SWITCH
                        );



---
-- invoking Module: SWITCH (Signal Switch)
---
module_switch: SWITCH
        generic map     ( n              => n)
        port map        ( RESET          => RESET,
                          SWITCH         => GL_SWITCH,
                          COUNT1_IN      => CNT1_REG_SWITCH,
                          COUNT2_IN      => CNT2_REG_SWITCH,
                          COUNT_OUT      => SWITCH_CNT
                        );


---
-- invoking Module: AV (Average Module)
---
module_av: AV
        generic map     ( n             => n)
        port map        ( CLK           => CLK,
                          WRITE         => GL_WRITE_AV,
                          RESET         => RESET,
                          CALC          => GL_CALC_AV,
                          COUNT_IN      => SWITCH_CNT,
                          COUNT_OUT     => AV_BCNT_CMP,
                          TEST_OUT      => BCNTTST_OUT,
                          AV_OUT        => AV_OUT,
                          AVTST_OUT     => AVTST_OUT
                        );

---
-- invoking Module: CMP (Compare Module)
---
module_cmp: CMP
        generic map     ( n             => n)
        port map        ( CLK           => CLK,
                          RESET         => RESET,
                          COUNT_IN      => AV_BCNT_CMP,
                          AV_IN         => AV_OUT,
                          CALC2         => GL_CALC2_CMP,
                          CALC3         => GL_CALC3_CMP,
                          CALC4         => GL_CALC4_CMP,
                          CALC5         => GL_CALC5_CMP,
                          CALC6         => GL_CALC6_CMP,
                          CALC7         => GL_CALC7_CMP,
                          CALC8         => GL_CALC8_CMP,
                          SQR_OUT       => SQR_OUT,
                          SIG_OUT       => SIG_OUT,
```

```
                                  TRIGGER       => TRIGGER
                                );
end Structural;
```

B.4.1 BD_IN-Module

```
--------------------------------------------------------------------------
-------------  Burst-Detection-Unit Input , 27.12.00  TS/IAAT -------------
--------------------------------------------------------------------------


---
-- BD_IN  (Burst Detection Input) / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RNE           : Pipeline Tag from EPA_UNIT (Testbench)
-- BCNT_IN       : Flag-Signal from FD-Unit/BS-Module. If the energy entry
--                 of an Event-Packet matches the Energy Threshold of the
--                 Burst Snooper it will make BCNT_IN = 1.
--
------
-- Task:     Generate Count-Signal for Event-Counter for each valid Event.
--           RNE:  Process starts with every RNE. If BCNT_IN is 1, a valid
--                 Event-Packet has passed the Burst-Snooper. In this case
--                 the BD_IN will generate a Count-Signal (BCNT_OUT),
--                 otherwise not.
--
------
-- Output Signals:
--
-- BCNT_OUT      : Count-Signal for the following Burst Counter. There is
--                 a Count-Impulse every RNE if the BCNT_IN was 1.
--
--------------------------------------------------------------------------


-- VHDL script starts here:



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


entity BD_IN is

        port     ( CLK         : in std_logic;
                   RESET       : in std_logic;
                   RNE         : in std_logic;
                   BCNT_IN     : in std_logic;
                   BCNT_OUT    : out std_logic
                 );
end BD_IN;

architecture inside of BD_IN is
```

```
begin
        ---
        -- Read Event
        ---
        read:
        process (CLK, RESET)
                variable last_RNE: std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                BCNT_OUT <= '0';
                                last_RNE := '0';
                        else
                                if RNE = '1' and last_RNE = '0' then
                                        BCNT_OUT <= BCNT_IN;
                                else
                                        BCNT_OUT <= '0';
                                end if;
                                last_RNE := '0';
                        end if;
                end if;
        end process;
end inside;
```

B.4.2 GATE-Module

```
--------------------------------------------------------------------------
--  Gate for Burst Counts, 04.10.00  TS/IAAT ----------------------------
--------------------------------------------------------------------------

---
-- GATE / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- RESET          : Signals from EPP-TOP (Testbench)
-- SWITCH         : Input signal to toggle the Count-Input to one of the
--                  two Counter Outputs. Signal is generated from GC-Module
-- BCNT_IN        : Count-Signal Input from BD_IN-Module
--
------
-- Task:     Switch the Count-Signal to one of the two counters.
--           Terminate Gate-Time during the counters are provided
--           with count-signals.
------
-- Output Signals:
--
-- BCNT1_OUT      : Count-Signal Output to Counter 1
-- BCNT2_OUT      : Count-Signal Output to Counter 2
--
--------------------------------------------------------------------------



-- VHDL script starts here:



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


entity GATE is
        port     ( RESET          : in std_logic;
                   SWITCH         : in std_logic;
                   BCNT_IN        : in std_logic;
                   BCNT1_OUT      : out std_logic;
                   BCNT2_OUT      : out std_logic
                 );
end GATE;

architecture inside of GATE is

begin
        ---
        -- Switching
```

```
        ---
        gate_bcnt:
        process (BCNT_IN, SWITCH, RESET)
        begin
                if RESET = '1' then
                        BCNT1_OUT <= '0';
                        BCNT2_OUT <= '0';
                else
                        if SWITCH = '1' then
                                BCNT1_OUT <= BCNT_IN;
                                BCNT2_OUT <= '0';
                        else
                                BCNT2_OUT <= BCNT_IN;
                                BCNT1_OUT <= '0';
                        end if;
                end if;
        end process;
end inside;
```

B.4.3 CNT1-Module

```
--------------------------------------------------------------------------
--  Burst Counter 1, 03.10.00  TS/IAAT ----------------------------------
--------------------------------------------------------------------------


---
-- CNT1 (Counter 1) / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RESET_SET     : Local Reset, from GC-Module.
-- CNT1_IN       : Input Count-Signal from GATE-Module.
--
------
-- Task:     Pulse Counter. Counts all pulses from BD_IN-Module, caused
--           by valid Event-Packets of the Burst-Snooper (FD-Unit).
--           Counters-Gate is controlled by GATE-Module.
------
-- Output Signals:
--
-- REG1_OUT      : Count-Number to SWITCH-Module
--
--------------------------------------------------------------------------


-- VHDL script starts here:




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


entity CNT1 is
   generic  (n               : in positive);
   port     (CLK             : in std_logic;
             CNT1_IN         : in std_logic;
             RESET           : in std_logic;
             RESET_SET       : in std_logic;
             REG1_OUT        : out unsigned ((4*n)-1 downto 0)
            );
end CNT1;


architecture inside of CNT1 is

   component HC
       port     ( CNT        : in std_logic;
```

```
                  RESET      : in std_logic;
                  CARRY      : out std_logic;
                  Q          : out unsigned (3 downto 0)
                );
   end component;

signal C                  : unsigned ((n-1) downto 0);
signal RESET_LOCAL        : std_logic;

begin
   ---
   --  Generate Counter for GATE Time
   ---
   counter: for i in 0 to (n-1) generate
      counter_in: if i = 0 generate
         cnt_in: HC
             port map (CNT1_IN,RESET_LOCAL,C(0),REG1_OUT(3 downto 0));
      end generate;

      counter_mid: if i > 0 and i < (n-1) generate
         cnt_mid: HC
             port map (C(i-1),RESET_LOCAL,C(i),REG1_OUT(((4*i)+3) downto (4*i)));
      end generate;

      counter_out: if i = (n-1) generate
         cnt_out: HC
             port map (C(i-1),RESET_LOCAL,C(i),REG1_OUT(((4*i)+3) downto (4*i)));
      end generate;
   end generate;


   ---
   -- Generate Local Reset Signal
   ---
   gc_reset:
   process (CLK)
   begin
        if RESET = '1' or RESET_SET = '1' then
             RESET_LOCAL <= '1';
        else
             RESET_LOCAL <= '0';
        end if;
   end process;
end inside;
```

B.4.4 CNT2-Module

```
--------------------------------------------------------------------------
--  Burst Counter 2, 03.10.00  TS/IAAT ----------------------------------
--------------------------------------------------------------------------

---
-- CNT2 (Counter 2) / BD-Unit
---


--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench)
-- RESET_SET     : Local Reset, from GC-Module.
-- CNT2_IN       : Input Count-Signal from GATE-Module.
--
------
-- Task:     Pulse Counter. Counts all pulses from BD_IN-Module, caused
--           by valid Event-Packets of the Burst-Snooper (FD-Unit).
--           Counters-Gate is controlled by GATE-Module.
------
-- Output Signals:
--
-- REG2_OUT      : Count-Number to SWITCH-Module
--
--------------------------------------------------------------------------


-- VHDL script starts here:




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


entity CNT2 is
   generic  (n              : in positive);
   port     (CLK            : in std_logic;
             CNT2_IN        : in std_logic;
             RESET          : in std_logic;
             RESET_SET      : in std_logic;
             REG2_OUT       : out unsigned ((4*n)-1 downto 0)
            );
end CNT2;


architecture inside of CNT2 is

   component HC
       port    ( CNT          : in std_logic;
                 RESET        : in std_logic;
```

```
                 CARRY         : out std_logic;
                 Q             : out unsigned (3 downto 0)
               );
   end component;

signal C: unsigned ((n-1) downto 0);
signal RESET_LOCAL: std_logic;

begin
   ---
   --  Generate Counter for GATE Time
   ---
   counter: for i in 0 to (n-1) generate
       counter_in: if i = 0 generate
            cnt_in: HC
                port map (CNT2_IN,RESET_LOCAL,C(0),REG2_OUT(3 downto 0));
       end generate;

       counter_mid: if i > 0 and i < (n-1) generate
            cnt_mid: HC
                port map (C(i-1),RESET_LOCAL,C(i),REG2_OUT(((4*i)+3) downto (4*i)));
       end generate;

       counter_out: if i = (n-1) generate
            cnt_out: HC
                port map (C(i-1),RESET_LOCAL,C(i),REG2_OUT(((4*i)+3) downto (4*i)));
       end generate;
   end generate;


   ---
   -- Generate Local Reset Signal
   ---
   gc_reset:
   process (CLK)
   begin
        if RESET = '1' or RESET_SET = '1' then
             RESET_LOCAL <= '1';
        else
             RESET_LOCAL <= '0';
        end if;
   end process;
end inside;
```

B.4.5 SWITCH-Module

```
--------------------------------------------------------------------------
--  Switch for Burst Counts, 04.10.00  TS/IAAT --------------------------
--------------------------------------------------------------------------

---
-- SWITCH / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- RESET          : Signals from EPP-TOP (Testbench)
-- SWITCH         : Input signal to toggle the Count-Input to the Output.
--                  Signal is generated from GC-Module
-- COUNT1_IN      : Count Signal from Counter 1
-- COUNT2_IN      : Count Signal from Counter 2
--
------
-- Task:      Switch the Counter Output of the inactive Counter to the
--            following AV-Module and save last counter state.
--
------
-- Output Signals:
--
-- COUNT_OUT       : Count Signal from Counter 1 or Counter 2, depending
--                   on SWITCH Signal.
--
--------------------------------------------------------------------------



-- VHDL script starts here:



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


entity SWITCH is
        generic  ( n             : in positive);
        port     ( RESET         : in std_logic;
                   SWITCH        : in std_logic;
                   COUNT1_IN     : in unsigned ((4*n)-1 downto 0);
                   COUNT2_IN     : in unsigned ((4*n)-1 downto 0);
                   COUNT_OUT     : out unsigned ((4*n)-1 downto 0)
                 );
end SWITCH;

architecture inside of SWITCH is

begin
        ---
```

```
        -- Switching
        ---
        switch_bcnt:
        process (COUNT1_IN, COUNT2_IN, SWITCH, RESET)
        begin
                if RESET = '1' then
                        COUNT_OUT <= (others => '0');
                else
                        if SWITCH = '1' then
                                COUNT_OUT <= COUNT2_IN;
                        else
                                COUNT_OUT <= COUNT1_IN;
                        end if;
                end if;
        end process;
end inside;
```

B.4.6 AV-Module

```
--------------------------------------------------------------------------
--  Average Counts of Gamma Events, 03.10.00  TS/IAAT --------------------
--------------------------------------------------------------------------

---
-- AV (Average Module) / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET      : Signals from EPP-TOP (Testbench).
--
-- WRITE          : Signal from GL-Module.
--
-- CALC           : Signal from GL-Module.
--
-- COUNT_IN       : Actual Count from Switch -Module.
--
------
-- Task:  Copy the actual counts for later processing in CMP-Module synchron
--        with WRITE-Signal. Calculate average count over the last 64 actual
--        counts synchron with CALC-Signal. The last 64 counts are stored in
--        an array. The division by 64 is easy made by arithmetic shifting.
--        Output (stored actual counts and average counts) goes to CMP-Module
--        for comparing.
--
------
-- Output Signals:
--
-- COUNT_OUT      : Actual Count to CMP-Module.
--
-- TEST_OUT       : Actual Count for diagnostic, to SAVE_BD-Module.
--
-- AV_OUT         : Average Count over last 64 Count Rates, to CMP-Module.
--
-- AVTST_OUT      : Average Count for diagnostic, to SAVE_BD-Module.
--
---------------------------------------------------------------------------


-- VHDL script starts here:



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


entity AV is
   generic  (n              : in positive);
   port     (CLK            : in std_logic;
```

```
            WRITE           : in std_logic;
            RESET           : in std_logic;
            CALC            : in std_logic;
            COUNT_IN        : in unsigned ((4*n)-1 downto 0);
            COUNT_OUT       : out unsigned ((4*n)-1 downto 0);
            TEST_OUT        : out unsigned ((4*n)-1 downto 0);
            AV_OUT          : out unsigned ((4*n)-1 downto 0);
            AVTST_OUT       : out unsigned ((4*n)-1 downto 0)
           );
end AV;


architecture inside of AV is


begin

   ---
   --  Copy Input Register to entry-array if WRITE = 1
   ---
   cnt_buffer:
   process (WRITE,CALC,RESET)
       type AV_array is array (0 to 64) of unsigned ((4*n)+5 downto 0);
       variable AV      : AV_array;
       variable SUM     : unsigned ((4*n)+5 downto 0);
       variable COUNT   : unsigned ((4*n)+5 downto 0);
       variable i       : integer range 0 to 64;
   begin
       if RESET = '1' then
             for j in 0 to 63 loop
                     AV(j) := (others => '0');
             end loop;
             SUM           := (others => '0');
             COUNT         := (others => '0');
             AV_OUT        <= (others => '0');
             AVTST_OUT     <= (others => '0');
             COUNT_OUT     <= (others => '0');
             i             := 0;
       else
             if rising_edge(WRITE) then
                   COUNT_OUT                   <= COUNT_IN;
                   TEST_OUT                    <= COUNT_IN;
                   COUNT                       := (others => '0');
                   COUNT((4*n)-1 downto 0)     := COUNT_IN;
                   AV(i)                       := COUNT;
                   i                           := i + 1;
                   if i = 64 then
                         i := 0;
                   end if;
             end if;
             if rising_edge(CALC) then
                   SUM := (others => '0');
                   for j in 0 to 63 loop
                         SUM := SUM + AV(j);
                   end loop;
                   AV_OUT((4*n)-1 downto 0)    <= SUM((4*n)+5 downto 6);  -- / 64
                   AVTST_OUT((4*n)-1 downto 0) <= SUM((4*n)+5 downto 6);  -- / 64
             end if;
```

```
      end if;
   end process;
end inside;
```

B.4.7 CMP-Module

```
------------------------------------------------------------------------
-- Compare actual count to average count rate, 07.03.01  TS/IAAT ---------
------------------------------------------------------------------------

---
-- CMP (Compare Module) / BD-Unit
---


------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- WRITE         : Signal from GL-Module.
-- COUNT_IN      : Actual count from AV-Module.
-- AV_IN         : Average count from AV-Module.
-- CALC2-8       : Signals for sequential control
--
------
-- Task:  Processing Square Rout from AV_IN signal, and compare standard
--        deviation (s = 5 * Sigma) with actual count. If actual count is
--        outside the 5 Sigma frame (larger), set Trigger-Output-Signal
--        to high -> BURST-ALERT!
--
------
-- Output Signals:
--
-- SQR_OUT       : Square Rout Out for diagnostic, to SAVE_BD-Module.
-- SIG_OUT       : 5 Sigma Out for diagnostic, to SAVE_BD-Module.
-- TRIGGER       : Burst-Trigger-Out to SAVE_BD-Module.
--
------------------------------------------------------------------------


-- VHDL script starts here:




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


entity CMP is
        generic  (n             : in positive);
        port     (CLK           : in std_logic;
                  RESET         : in std_logic;
                  COUNT_IN      : in unsigned ((4*n)-1 downto 0);
                  AV_IN         : in unsigned ((4*n)-1 downto 0);
                  CALC2         : in std_logic;
                  CALC3         : in std_logic;
                  CALC4         : in std_logic;
                  CALC5         : in std_logic;
```

```
                  CALC6         : in std_logic;
                  CALC7         : in std_logic;
                  CALC8         : in std_logic;
                  SQR_OUT       : out unsigned ((4*n)-1 downto 0);
                  SIG_OUT       : out unsigned ((4*n)-1 downto 0);
                  TRIGGER       : out std_logic
                 );
end CMP;


architecture inside of CMP is

   ---
   -- function SHIFTL -- shift a bit vector one bit to the left
   ---

   function SHIFTL (INP : unsigned) return unsigned is
           variable OUTP1 : unsigned ((4*n) downto 0);
   begin
           for i in 1 to (4*n) loop
                   OUTP1((4*n)+1-i) := INP((4*n)+1-i-1);
           end loop;
           OUTP1(0) := '0';
           return OUTP1;
   end SHIFTL;

   ---
   -- function SHIFTR -- shift a bit vector one bit to the right
   ---

   function SHIFTR (INP : unsigned) return unsigned is
           variable OUTP2 : unsigned ((4*n) downto 0);
   begin
           for i in 0 to (4*n)-1 loop
                   OUTP2(i)     := INP(i+1);
           end loop;
                   OUTP2((4*n)) := '0';
           return OUTP2;
   end SHIFTR;


signal  EP   : integer range 0 to (4*n);      -- Exponent of AV_IN
signal  X    : unsigned ((4*n)-1 downto 0);   -- sqr of AV_IN out
signal  XC   : unsigned ((4*n)-1 downto 0);   -- Count-In variable
signal  XOut : unsigned ((4*n)-1 downto 0);   -- sqr of mantissa out

begin
   ---
   -- find out exponent entry
   ---
   get_exponent:
   process (CLK,CALC2,CALC3,CALC4,CALC5,CALC6,CALC7,CALC8,RESET)
           variable  i          : integer range 0 to (4*n)-1;
           variable  j          : integer range 0 to (8*n)-1;
           variable  q          : integer range 0 to (8*n)-1;
           variable  XP         : unsigned ((4*n) downto 0);
           variable  AV         : unsigned ((4*n)-1 downto 0);
           variable  XN         : unsigned ((4*n) downto 0);
```

```
        variable  BN         : unsigned ((4*n) downto 0);
        variable  MN         : unsigned ((4*n) downto 0);
        variable  OP1        : unsigned ((4*n) downto 0);
        variable  OP2        : unsigned ((4*n) downto 0);
        variable  OP3        : unsigned ((4*n) downto 0);
        variable  XO         : unsigned ((4*n) downto 0);
        variable  k          : integer range 0 to (4*n)-1;
        variable  E          : integer range 0 to (4*n)-1;
        variable  XO1        : unsigned ((4*n)+3 downto 0);
        variable  XO2        : unsigned ((4*n)+3 downto 0);
        variable  XO3        : unsigned ((4*n)+3 downto 0);
        variable  last_CALC2 : std_logic;
        variable  last_CALC3 : std_logic;
        variable  last_CALC4 : std_logic;
        variable  last_CALC5 : std_logic;
        variable  last_CALC6 : std_logic;
        variable  last_CALC7 : std_logic;
        variable  last_CALC8 : std_logic;

begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        i          := 0;
                        j          := 0;
                        k          := 0;
                        q          := 0;
                        EP         <= 0;
                        X          <= (others => '0');
                        MN         := (others => '0');
                        AV         := (others => '0');
                        XC         <= (others => '0');
                        XP         := (others => '0');
                        XN         := (others => '0');
                        BN         := (others => '0');
                        OP1        := (others => '0');
                        OP2        := (others => '0');
                        OP3        := (others => '0');
                        E          := 0;
                        XO         := (others => '0');
                        XOut       <= (others => '0');
                        TRIGGER    <= '0';
                        XO1        := (others => '0');
                        XO2        := (others => '0');
                        XO3        := (others => '0');
                        SQR_OUT    <= (others => '0');
                        SIG_OUT    <= (others => '0');
                        last_CALC2 := '0';
                        last_CALC3 := '0';
                        last_CALC4 := '0';
                        last_CALC5 := '0';
                        last_CALC6 := '0';
                        last_CALC7 := '0';
                        last_CALC8 := '0';
                else
                        -- set initial values
                        if CALC2 = '1' and last_CALC2 = '0' then
                              AV := AV_IN;
                              XC <= COUNT_IN;
```

```
end if;
last_CALC2 := CALC2;

-- find the first 1 in bit-vector AV from left
if CALC3 = '1' and last_CALC3 = '0' then

      first:
      for i in (4*n)-1 downto 0 loop
            if AV(i) = '1' then
                    j := i + 1;
                    EXIT first;
            end if;
      end loop first;
end if;
last_CALC3 := CALC3;

-- find out if Number is odd or even
if CALC4 = '1' and last_CALC4 = '0' then
      evenodd:
      for i in 0 to ((4*n)-1) loop
            q := 2*i + 1;
            if j = q then
                    EP <= j + 1;
                    EXIT evenodd;
            else
                    EP <= j;
            end if;
      end loop evenodd;
end if;
last_CALC4 := CALC4;


---
-- calculate mantissa
---

-- initialise variables
if CALC5 = '1' and last_CALC5 = '0' then
      BN                   := (others => '0');
      BN(EP)               := '1';
      XN                   := (others => '0');
      MN((4*n)-1 downto 0) := AV;
      XP                   := (others => '0');
      OP1                  := (others => '0');
      OP2                  := (others => '0');
      OP3                  := (others => '0');
end if;
last_CALC5 := CALC5;


-- calculate SQR of mantissa
if CALC6 = '1' and last_CALC6 = '0' then
      iteration:
      for i in 1 to EP+1 loop
              BN  := SHIFTR(BN);
              XN  := XP;
              if MN = 0 then
                      EXIT iteration;
```

```
                    end if;

                    XN  := SHIFTL(XN);
                    OP1 := XN + BN;

                    for k in 1 to i loop
                            OP1 := SHIFTR(OP1);
                    end loop;

                    if OP1 > MN then
                            OP2 := MN;
                            OP3 := XP;
                    else
                            OP2 := MN - OP1;
                            OP3 := XP + BN;
                    end if;

                    MN  := OP2;
                    XP  := OP3;
                    OP1 := (others => '0');
                    OP2 := (others => '0');
            end loop iteration;
            XO := XP;
      end if;
      last_CALC6 := CALC6;

      -- processing SQR of Number AV_IN
      if CALC7 = '1' and last_CALC7 = '0' then
            E := EP/2;
            for i in 1 to E loop
                    XO := SHIFTR(XO);
            end loop;
            XOut       <= XO((4*n)-1 downto 0);
            SQR_OUT    <= XO((4*n)-1 downto 0);
            XO1        := (others => '0');
            XO2        := (others => '0');
            XO3        := (others => '0');
      end if;
      last_CALC7 := CALC7;


      ---
      -- compare and burst trigger
      ---
      if CALC8 = '1' and last_CALC8 = '0' then
            XO1((4*n)-1 downto 0) := XOut;
            XO3((4*n)-1 downto 0) := XC;

            for i in 1 to 5 loop      -- 5 SIGMA criteria
                    XO2 := XO2 + XO1;  -- multiply with 5
            end loop;

            SIG_OUT <= XO2((4*n)-1 downto 0);

            if XO3 > AV+XO2 then     -- cmp count-rate XO3
                    TRIGGER <= '1';   -- with Average
            else
                    TRIGGER <= '0';
```

```
                                end if;
                        end if;
                        last_CALC8 := CALC8;
                  end if;
          end if;
   end process;
end inside;
```

B.4.8 GC-Module

```
--------------------------------------------------------------------------
--  GATE COUNTER for Gate Time, Rev 0.2, 08.10.00  TS/IAAT ---------------
--------------------------------------------------------------------------

---
-- GC (Gate Counter) / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- GATE_SWITCH   : 2 Bit signal to set the Gate-Time of the Burst Counter.
--                 00 = 1/8s, 01 = 1/2s, 10 = 1s, 11 = 2s.
--
------
-- Task:  Processing the time base of the Gate from Clock-Signal, by using
--        a counter (cascades of hex-counters).
--
------
-- Output Signals:
--
-- GATE_TIME     : Time-Base for the Gate, to GC-Module for processing
--                 further signals.
--
--------------------------------------------------------------------------


-- VHDL script starts here:



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


entity GC is
        ---
        -- f = Clock Frequency, n = Number of Hex-Counters to be needed to
        -- represent the frequency number! For initial values see BD_UNIT !
        ---
        generic  (n              : in positive;
                  gfreq          : in integer
                 );

        port     (CLK            : in std_logic;
                  RESET          : in std_logic;
                  GATE_SWITCH    : in unsigned (1 downto 0);
                  GATE_TIME      : out std_logic
                 );
end GC;
```

```
architecture inside of GC is

        component TFF
                port    ( T      : in std_logic;
                          RESET  : in std_logic;
                          OUTPUT : out std_logic
                        );
        end component;

        component HC
                port    ( CNT    : in std_logic;
                          RESET  : in std_logic;
                          CARRY  : out std_logic;
                          Q      : out unsigned (3 downto 0)
                        );
        end component;


signal C                : unsigned ((n-1) downto 0);
signal REG_OP           : unsigned ((4*n)-1 downto 0);
signal RESET_SET        : std_logic;
signal RESET_LOCAL      : std_logic;
signal BASE_CLK         : std_logic;
signal BASE_0250        : std_logic;
signal BASE_0500        : std_logic;
signal BASE_1000        : std_logic;
signal BASE_2000        : std_logic;
signal GATE_0125        : std_logic;
signal GATE_0500        : std_logic;
signal GATE_1000        : std_logic;
signal GATE_2000        : std_logic;

begin
    ---
    --  Generate Counter for GATE Time
    ---
    counter: for i in 0 to (n-1) generate
        counter_in: if i = 0 generate
           cnt_in: HC
               port map (CLK,RESET_LOCAL,C(0),REG_OP(3 downto 0));
        end generate;

        counter_mid: if i > 0 and i < (n-1) generate
           cnt_mid: HC
               port map (C(i-1),RESET_LOCAL,C(i),REG_OP(((4*i)+3) downto (4*i)));
        end generate;

        counter_out: if i = (n-1) generate
           cnt_out: HC
               port map (C(i-1),RESET_LOCAL,C(i),REG_OP(((4*i)+3) downto (4*i)));
        end generate;
    end generate;


    ---
    -- Generate Time Base
    -- TIMECONST set the Time Base to generate the Gate Times:
    -- 1/8sec, 1/2sec, 1sec and 2sec.
```

```
---
-- TIMECONST depend on Clock Frequency and define the 1/8 second
-- Gate Time Base (BASE_CLK) . The explicit GATE times (BASE_0125,
-- BASE_0500, BASE_1000, BASE_2000) will be generated by using a
-- T-FlipFlop as Frequency Divider.
---
-- The Counter starts always with Zero! The last Number that is
-- counted is the Number in TIMECONST!
---

gc_timebase:
process (CLK)
      variable TIMECONST : unsigned ((4*n)-1 downto 0);
begin
      --TIMECONST := CONV_UNSIGNED(2000,(4*n));
      TIMECONST := CONV_UNSIGNED(gfreq,(4*n));
      if CLK = '1' then
            if REG_OP = TIMECONST then
                  RESET_SET <= '1';
            else
                  RESET_SET <= '0';
            end if;
      else
            RESET_SET <= '0';
      end if;
end process;


---
-- Generate Local Reset Signal
---
gc_reset:
process (CLK,RESET,RESET_SET)
begin
      if RESET_SET = '1' or RESET = '1' then
            RESET_LOCAL <= '1';
      else
            RESET_LOCAL <= '0';
      end if;
end process;





---
-- Generate BASE_CLK
---
base: TFF
      port map (RESET_SET,RESET,BASE_CLK);
GATE_0125 <= BASE_CLK;

---
-- Generate multiple Gate Times
---
quarter: TFF
      port map (BASE_CLK,RESET,BASE_0250);
```

```
    ---
    -- Generate multiple Gate Times
    ---
    half: TFF
          port map (BASE_0250,RESET,BASE_0500);
    GATE_0500 <= BASE_0500;

    ---
    -- Generate multiple Gate Times
    ---
    full: TFF
          port map (BASE_0500,RESET,BASE_1000);
    GATE_1000 <= BASE_1000;

    ---
    -- Generate multiple Gate Times
    ---
    double: TFF
          port map (BASE_1000,RESET,BASE_2000);
    GATE_2000 <= BASE_2000;


    ---
    -- GATE Switching
    ---
    g_switch:
    process (GATE_0125,GATE_0500,GATE_1000,GATE_2000)
          variable ADDR : integer range 0 to 3;
    begin
          ADDR := CONV_INTEGER(GATE_SWITCH);
          case ADDR is
                when 0 => GATE_TIME <= GATE_0125;
                when 1 => GATE_TIME <= GATE_0500;
                when 2 => GATE_TIME <= GATE_1000;
                when 3 => GATE_TIME <= GATE_2000;
          end case;
    end process;
end inside;
```

B.4.9 GL-Module

```
---------------------------------------------------------------------------
--  Gate Logic for Burst Counters, 29.09.00  TS/IAAT ---------------------
---------------------------------------------------------------------------

---
-- GL (Gate Logic) / BD-Unit
---


---------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
--
-- RNE           : RNE  Pipeline Signal.
--
-- GATE_TIME     : Input Raw-Time Base Signal from Gate Counter Module.
--
------
-- Task:  Processing signals for sequential control of the BD-Unit modules,
--        especially the counters CNT1 and CNT2, and the CMP-Module.
--
------
-- Output Signals:
--
-- SWITCH        : Gate-Switch Signal for GATE and SWITCH-Module.
--
-- WRITE_OUT     : Signal to trigger the writeout from Counters to AV-Module.
--
-- CALC_OUT      : Signal to trigger the processing inside the AV-Module.
-- CALC2_OUT/
-- CALC8_OUT     : Sequential Control Signals for processing inside the
--                 CMP-Module.
--
-- RESET1_OUT    : local reset signal to reset counter 1 after writeout.
--
-- RESET2_OUT    : local reset signal to reset counter 2 after writeout.
--
-- GATE_CLK      : Clock signal to control writing data in SAVE_BD-Modules.
--
---------------------------------------------------------------------------


-- VHDL script starts here:



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


entity GL is
        port    (CLK              : in std_logic;
```

```
                 RNE              : in std_logic;
                 RESET            : in std_logic;
                 GATE_TIME        : in std_logic;
                 SWITCH           : out std_logic;
                 WRITE_OUT        : out std_logic;
                 CALC_OUT         : out std_logic;
                 CALC2_OUT        : out std_logic;
                 CALC3_OUT        : out std_logic;
                 CALC4_OUT        : out std_logic;
                 CALC5_OUT        : out std_logic;
                 CALC6_OUT        : out std_logic;
                 CALC7_OUT        : out std_logic;
                 CALC8_OUT        : out std_logic;
                 CALC9_OUT        : out std_logic;
                 RESET1_OUT       : out std_logic;
                 RESET2_OUT       : out std_logic;
                 GATE_CLK         : out std_logic
                );
end GL;


architecture inside of GL is

        ---
        -- D FlipFlop
        ---
        component DFF
                port    ( CLK          : in std_logic;
                          D            : in std_logic;
                          RESET        : in std_logic;
                          OUTPUT       : out std_logic
                        );
end component;

signal SH1              : std_logic;
signal SH2              : std_logic;
signal SH3              : std_logic;
signal SH4              : std_logic;
signal SH5              : std_logic;
signal SH6              : std_logic;
signal SH7              : std_logic;
signal SH8              : std_logic;
signal SH9              : std_logic;
signal SH10             : std_logic;
signal SH11             : std_logic;
signal SH12             : std_logic;
signal SH13             : std_logic;
signal SH14             : std_logic;
signal SH15             : std_logic;
signal SH16             : std_logic;
signal SH17             : std_logic;
signal SH18             : std_logic;
signal SH19             : std_logic;
signal SH20             : std_logic;
signal SL1              : std_logic;
signal SL2              : std_logic;
signal SL3              : std_logic;
signal SL4              : std_logic;
```

```
signal SL5              : std_logic;
signal SL6              : std_logic;
signal SL7              : std_logic;
signal SL8              : std_logic;
signal SL9              : std_logic;
signal SL10             : std_logic;
signal SL11             : std_logic;
signal SL12             : std_logic;
signal SL13             : std_logic;
signal SL14             : std_logic;
signal SL15             : std_logic;
signal SL16             : std_logic;
signal SL17             : std_logic;
signal SL18             : std_logic;
signal SL19             : std_logic;
signal SL20             : std_logic;
signal GATE_TIME_BAR    : std_logic;

begin

        -- should be replaced by a generate some day. --


        write1_sh1: DFF
                port map (RNE,GATE_TIME,RESET,SH1);

        write1_sh2: DFF
                port map (RNE,SH2,RESET,SH3);

        write1_sh3: DFF
                port map (RNE,SH3,RESET,SH4);

        write1_sh4: DFF
                port map (RNE,SH4,RESET,SH5);

        write1_sh5: DFF
                port map (RNE,SH5,RESET,SH6);

        write1_sh6: DFF
                port map (RNE,SH6,RESET,SH7);

        write1_sh7: DFF
                port map (RNE,SH7,RESET,SH8);

        write1_sh8: DFF
                port map (RNE,SH8,RESET,SH9);

        write1_sh9: DFF
                port map (RNE,SH9,RESET,SH10);

        write1_sh10: DFF
                port map (RNE,SH10,RESET,SH11);

        write1_sh11: DFF
                port map (RNE,SH11,RESET,SH12);

        write1_sh12: DFF
                port map (RNE,SH12,RESET,SH13);
```

```
write1_sh13: DFF
        port map (RNE,SH13,RESET,SH14);

write1_sh14: DFF
        port map (RNE,SH14,RESET,SH15);

write1_sh15: DFF
        port map (RNE,SH15,RESET,SH16);

write1_sh16: DFF
        port map (RNE,SH16,RESET,SH17);

write1_sh17: DFF
        port map (RNE,SH17,RESET,SH18);

write1_sh18: DFF
        port map (RNE,SH18,RESET,SH19);

write1_sh19: DFF
        port map (RNE,SH19,RESET,SH20);



write1_sl1: DFF
        port map (RNE,GATE_TIME_BAR,RESET,SL1);

write1_sl2: DFF
        port map (RNE,SL2,RESET,SL3);

write1_sl3: DFF
        port map (RNE,SL3,RESET,SL4);

write1_sl4: DFF
        port map (RNE,SL4,RESET,SL5);

write1_sl5: DFF
        port map (RNE,SL5,RESET,SL6);

write1_sl6: DFF
        port map (RNE,SL6,RESET,SL7);

write1_sl7: DFF
        port map (RNE,SL7,RESET,SL8);

write1_sl8: DFF
        port map (RNE,SL8,RESET,SL9);

write1_sl9: DFF
        port map (RNE,SL9,RESET,SL10);

write1_sl10: DFF
        port map (RNE,SL10,RESET,SL11);

write1_sl11: DFF
        port map (RNE,SL11,RESET,SL12);
```

```
write1_sl12: DFF
        port map (RNE,SL12,RESET,SL13);

write1_sl13: DFF
        port map (RNE,SL13,RESET,SL14);

write1_sl14: DFF
        port map (RNE,SL14,RESET,SL15);

write1_sl15: DFF
        port map (RNE,SL15,RESET,SL16);

write1_sl16: DFF
        port map (RNE,SL16,RESET,SL17);

write1_sl17: DFF
        port map (RNE,SL17,RESET,SL18);

write1_sl18: DFF
        port map (RNE,SL18,RESET,SL19);

write1_sl19: DFF
        port map (RNE,SL19,RESET,SL20);




---
-- Generate SH Delay
---
delay_sh:
process (CLK,RNE)
begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        SH2 <= '0';
                else
                        if GATE_TIME = '1' and SH1 = '0' then
                                SH2 <= '1';
                        else
                                SH2 <= '0';
                        end if;
                end if;
        end if;
end process;


---
-- Generate SL Delay
---
delay_sl:
process (CLK,RNE)
begin
        if rising_edge(CLK) then
                if RESET = '1' then
```

```
                        SL2 <= '0';
                        GATE_TIME_BAR <= '0';
                else
                        GATE_TIME_BAR <= NOT(GATE_TIME);
                        if GATE_TIME_BAR = '1' and SL1 = '0' then
                                SL2 <= '1';
                        else
                                SL2 <= '0';
                        end if;
                end if;
        end if;
end process;


---
-- Generate Sequence Control for Counter1 & Counter2 & AV & CMP
---
counter_sf:
process (CLK)
begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        SWITCH     <= '0';
                        WRITE_OUT  <= '0';
                        CALC_OUT   <= '0';
                        CALC2_OUT  <= '0';
                        CALC3_OUT  <= '0';
                        CALC4_OUT  <= '0';
                        CALC5_OUT  <= '0';
                        CALC6_OUT  <= '0';
                        CALC7_OUT  <= '0';
                        CALC8_OUT  <= '0';
                        CALC9_OUT  <= '0';
                        RESET1_OUT <= '0';
                        RESET2_OUT <= '0';
                        GATE_CLK   <= '0';
                else
                        if GATE_TIME = '1' then
                                SWITCH     <= '1';
                                WRITE_OUT  <= SH2;
                                RESET2_OUT <= SH4;
                                GATE_CLK   <= SH5;
                        elsif GATE_TIME = '0' then
                                SWITCH     <= '0';
                                WRITE_OUT  <= SL2;
                                RESET1_OUT <= SL4;
                                GATE_CLK   <= SL5;
                        end if;

                        CALC_OUT  <= SH4 or SL4;
                        CALC2_OUT <= SH6 or SL6;
                        CALC3_OUT <= SH8 or SL8;
                        CALC4_OUT <= SH10 or SL10;
                        CALC5_OUT <= SH12 or SL12;
                        CALC6_OUT <= SH14 or SL14;
                        CALC7_OUT <= SH16 or SL16;
                        CALC8_OUT <= SH18 or SL18;
                        CALC9_OUT <= SH20 or SL20;
```

```
                        end if;
                end if;
        end process;
end inside;
```

B.4.10 HC-Module

```
-------------------------------------------------------------------------
-----------  HexaDecimal Counter, Rev 1.1 , 06.10.00  TS/IAAT ------------
-------------------------------------------------------------------------


---
-- HC (Hex Counter) / BD-Unit
---


-------------------------------------------------------------------------
------
-- Input Signals:
--
-- RESET         : Signals from EPP-TOP (Testbench).
-- CNT           : Counter Input.
--
------
-- Task:  Hexadecimal Counter
--
------
-- Output Signals:
--
-- CARRY         : Carry Output for Counter Overflow.
-- Q             : Counter Output.
--
-------------------------------------------------------------------------


-- VHDL script starts here:




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


entity HC is
        port    ( CNT           : in std_logic;
                  RESET         : in std_logic;
                  CARRY         : out std_logic;
                  Q             : out unsigned (3 downto 0)
                );
end HC;

architecture inside of HC is

signal STATE     : integer range 0 to 15;

begin
        ---
        -- Count Process
        ---
```

```
        count:
        process (CNT, RESET)
                variable ICARRY : std_logic;
        begin
                if RESET = '1' then
                        STATE  <= 0;
                        ICARRY := '0';
                        CARRY  <= '0';
                else
                        if rising_edge(CNT) then
                                if STATE < 15 then
                                        STATE  <= STATE + 1;
                                        ICARRY := '0';
                                else
                                        STATE  <= 0;
                                        ICARRY := '1';
                                end if;
                        end if;
                        if ICARRY = '1' and CNT = '1' then
                                CARRY <= '1';
                        else
                                CARRY <= '0';
                        end if;
                end if;


        end process;

        Q <=    ('0','0','0','0') WHEN STATE = 0 ELSE
                ('0','0','0','1') WHEN STATE = 1 ELSE
                ('0','0','1','0') WHEN STATE = 2 ELSE
                ('0','0','1','1') WHEN STATE = 3 ELSE
                ('0','1','0','0') WHEN STATE = 4 ELSE
                ('0','1','0','1') WHEN STATE = 5 ELSE
                ('0','1','1','0') WHEN STATE = 6 ELSE
                ('0','1','1','1') WHEN STATE = 7 ELSE
                ('1','0','0','0') WHEN STATE = 8 ELSE
                ('1','0','0','1') WHEN STATE = 9 ELSE
                ('1','0','1','0') WHEN STATE = 10 ELSE
                ('1','0','1','1') WHEN STATE = 11 ELSE
                ('1','1','0','0') WHEN STATE = 12 ELSE
                ('1','1','0','1') WHEN STATE = 13 ELSE
                ('1','1','1','0') WHEN STATE = 14 ELSE
                ('1','1','1','1');
end inside;
```

B.4.11 DFF-Module

```
--------------------------------------------------------------------------
-- D FlipFlop, 26.09.00  TS/IAAT ---------------------------------------
--------------------------------------------------------------------------

---
-- DFF (D-Flip Flop) / BD-Unit
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
-- D             : D - Input (see Flip Flops)
--
------
-- Task:  Simulate a D - Flip Flop. Signal at D goes to OUTPUT with
--        rising edge of CLK-Signal.
--
------
-- Output Signals:
--
-- OUTPUT        : Output Signal
--
--------------------------------------------------------------------------


-- VHDL script starts here:




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


entity DFF is
        port     ( CLK          : in std_logic;
                   D            : in std_logic;
                   RESET        : in std_logic;
                   OUTPUT       : out std_logic
                 );
end DFF;

architecture inside of DFF is

begin
        ---
        --  Behavioural of D FlipFlop
        ---
        d_flipflop:
        process (CLK,D,RESET)
        begin
```

```
                if RESET = '1' then
                        OUTPUT <= '0';
                else
                        if rising_edge(CLK) then
                                OUTPUT <= D;
                        end if;
                end if;
        end process;
end inside;
```

B.4.12 TFF-Module

```
-------------------------------------------------------------------------
-- T  FlipFlop, 26.09.00  TS/IAAT ---------------------------------------
-------------------------------------------------------------------------

---
-- TFF / BD-Unit
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- RESET         : Signal from EPP-TOP (Testbench).
-- T             : Toggle Input
--
------
-- Task:  Simulate a Toggle - Flip Flop.
--
------
-- Output Signals:
--
-- OUTPUT        : Output Signal
--
---------------------------------------------------------------------------


-- VHDL script starts here:




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


entity TFF is
        port    ( T             : in std_logic;
                  RESET         : in std_logic;
                  OUTPUT        : out std_logic
                );
end TFF;

architecture inside of TFF is

begin
        ---
        --  Behavioural of T FlipFlop
        ---
        t_flipflop:
        process (T,RESET)
                variable last_Q : std_logic;
                variable Q      : std_logic;
        begin
                if RESET = '1' then
```

```
                        OUTPUT <= '0';
                        last_Q := '0';
                        Q      := '0';
                else
                        if rising_edge(T) then
                                if last_Q = '1' then
                                        Q      := '0';
                                        OUTPUT <= '0';
                                else
                                        Q      := '1';
                                        OUTPUT <= '1';
                                end if;
                                last_Q := Q;
                        end if;
                end if;
        end process;
end inside;
```

B.4.13 EPP Testbench

```
-----------------------------------
-- EPP Testbench, TS/IAAT, 2000 ---
-----------------------------------

---
-- Testbench / Test Environment
---


----------------------------------------------------------------------------
------
-- Input Signals:
--
-- GATETIMESET   : Signal to determine Gate-Time. Must be set by hand before
--                 running Testbench (Simulator). (00 = 1/8s, 01 = 1/2s,
--                 10 = 1s, 11 = 2s).
-- MODESET       : Invoke Processor Mode. Not implemented now! Signal must
--                 be set by hand before running Testbench (Simulator)
--                 (any number)
--
--
------
-- Task: Connect Testbench Moduls with EPP-Core. Generate Clock-Signal,
--       and RESET/LUTLOAD- Signal.
--
--
------
-- Output Signals:
--
--
----------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity TESTBENCH is
        generic  ( cycle         : time := 1000 ns;
                   n             : in positive := 6;
                   gfreq         : in integer  := 30000 -- according to 240000 Hz CLK
                 );
        port     ( GATETIMESET   : in unsigned (1 downto 0);
                   MODESET       : in unsigned (1 downto 0)
                 );
end TESTBENCH;


architecture Ait of TESTBENCH is
```

```
component LOAD
        port    ( CLK          : in std_logic;
                  RESET        : in std_logic;
                  RNE_OUT      : out std_logic;
                  REG_OUT      : out unsigned (167 downto 0);
                  FILE_DATA_VD : out std_logic
                );
end component;

component EPP_TOP
       generic  ( n            : in positive;
                  gfreq        : in positive
                );
       port     ( CLK          : in std_logic;
                  RESET        : in std_logic;
                  RNE          : in std_logic;
                  LOADLUT      : in std_logic;
                  MODE         : in unsigned (1 downto 0);
                  GATETIMESET  : in unsigned (1 downto 0);
                  IN_EVENT_EC  : in unsigned (167 downto 0);
                  RINGB        : out unsigned (40 downto 0);
                  RINGBCLK     : out std_logic;
                  TELEM        : out unsigned (44 downto 0);
                  TELEMCLK     : out std_logic;
                  BSN          : out unsigned (40 downto 0);
                  BSNCLK       : out std_logic;
                  TEST         : out unsigned (45 downto 0);
                  BURSTCOUNT   : out unsigned ((4*n)-1 downto 0);
                  AVECOUNT     : out unsigned ((4*n)-1 downto 0);
                  SQR_OUT      : out unsigned ((4*n)-1 downto 0);
                  SIG_OUT      : out unsigned ((4*n)-1 downto 0);
                  GATECLK      : out std_logic;
                  TRIGGER      : out std_logic
               );
end component;

component SAVE_TM
        port    ( RNE          : in std_logic;
                  TMCLK        : in std_logic;
                  DATA_IN      : in unsigned (44 downto 0);
                  FILE_DATA_VD : in std_logic
                );
end component;

component SAVE_BS
        port    ( RNE          : in std_logic;
                  BSCLK        : in std_logic;
                  DATA_IN      : in unsigned (40 downto 0);
                  FILE_DATA_VD : in std_logic
                );
end component;


component SAVE_RB
        port    ( RNE          : in std_logic;
                  RBCLK        : in std_logic;
                  TMCLK        : in std_logic;
                  DATA_IN      : in unsigned (40 downto 0)
```

```
                );
        end component;

        component SAVE_TST
                port    ( RNE          : in std_logic;
                          DATA_IN      : in unsigned (45 downto 0)
                        );
        end component;

        component SAVE_BD
                generic ( n            : in positive );
                port    ( GATECLK      : in std_logic;
                          DATA_IN      : in unsigned ((4*n)-1 downto 0);
                          AVE_IN       : in unsigned ((4*n)-1 downto 0);
                          SQR_IN       : in unsigned ((4*n)-1 downto 0);
                          SIG_IN       : in unsigned ((4*n)-1 downto 0);
                          TRIGGER_IN   : in std_logic
                        );
        end component;



signal CLK                      : std_logic;
signal RESET                    : std_logic;
signal LOADLUT                  : std_logic;
signal RNE                      : std_logic;
signal LOAD_PACKET_EPP          : unsigned (167 downto 0);
signal LOAD_FDVD_SAVE           : std_logic;
signal EPP_RINGB_SAVE           : unsigned (40 downto 0);
signal EPP_RINGBCLK_SAVE        : std_logic;
signal EPP_TELEM_SAVE           : unsigned (44 downto 0);
signal EPP_TELEMCLK_SAVE        : std_logic;
signal EPP_TESTD_SAVE           : unsigned (45 downto 0);
signal EPP_BCNT_SAVE            : unsigned ((4*n)-1 downto 0);
signal EPP_ACNT_SAVE            : unsigned ((4*n)-1 downto 0);
signal EPP_SQR_SAVE             : unsigned ((4*n)-1 downto 0);
signal EPP_SIG_SAVE             : unsigned ((4*n)-1 downto 0);
signal EPP_GCLK_SAVE            : std_logic;
signal EPP_BSN_SAVE             : unsigned (40 downto 0);
signal EPP_BSNCLK_SAVE          : std_logic;
signal EPP_TRIGGER_SAVE         : std_logic;

begin

        ---
        -- invoking subdesign LOAD
        ---
        loaddata: LOAD
                port map        ( CLK           => CLK,
                                  RESET         => RESET,
                                  RNE_OUT       => RNE,
                                  REG_OUT       => LOAD_PACKET_EPP,
                                  FILE_DATA_VD  => LOAD_FDVD_SAVE
                                );


        ---
        -- invoking subdesign Test-Modul:
```

```
---
testmodul: EPP_TOP
        generic map    ( n              => n,
                         gfreq          => gfreq
                       )
        port map       ( CLK            => CLK,
                         RESET          => RESET,
                         RNE            => RNE,
                         LOADLUT        => LOADLUT,
                         MODE           => MODESET,
                         GATETIMESET    => GATETIMESET,
                         IN_EVENT_EC    => LOAD_PACKET_EPP,
                         RINGB          => EPP_RINGB_SAVE,
                         RINGBCLK       => EPP_RINGBCLK_SAVE,
                         TELEM          => EPP_TELEM_SAVE,
                         TELEMCLK       => EPP_TELEMCLK_SAVE,
                         BSN            => EPP_BSN_SAVE,
                         BSNCLK         => EPP_BSNCLK_SAVE,
                         TEST           => EPP_TESTD_SAVE,
                         BURSTCOUNT     => EPP_BCNT_SAVE,
                         AVECOUNT       => EPP_ACNT_SAVE,
                         SQR_OUT        => EPP_SQR_SAVE,
                         SIG_OUT        => EPP_SIG_SAVE,
                         GATECLK        => EPP_GCLK_SAVE,
                         TRIGGER        => EPP_TRIGGER_SAVE
                       );

---
-- invoking subdesign SAVE_TM ( Save Telemetrie Data )
---
save_data_tm: SAVE_TM
        port map       ( RNE            => RNE,
                         TMCLK          => EPP_TELEMCLK_SAVE,
                         DATA_IN        => EPP_TELEM_SAVE,
                         FILE_DATA_VD   => LOAD_FDVD_SAVE
                       );

---
-- invoking subdesign SAVE_BS ( Save Data from BurstSnooper )
---
save_data_bs: SAVE_BS
        port map       ( RNE            => RNE,
                         BSCLK          => EPP_BSNCLK_SAVE,
                         DATA_IN        => EPP_BSN_SAVE,
                         FILE_DATA_VD   => LOAD_FDVD_SAVE
                       );

---
-- invoking subdesign SAVE_TST ( Save Test Data )
---
save_data_tst: SAVE_TST
        port map       ( RNE            => RNE,
                         DATA_IN        => EPP_TESTD_SAVE
                       );

---
-- invoking subdesign SAVE_RB ( Save Ringbuffer Data )
---
```

```
        save_data_rb: SAVE_RB
                port map       ( RNE            => RNE,
                                 RBCLK          => EPP_RINGBCLK_SAVE,
                                 TMCLK          => EPP_TELEMCLK_SAVE,
                                 DATA_IN        => EPP_RINGB_SAVE
                               );

        ---
        -- invoking subdesign SAVE_BD ( Save Burst Count Data and Average )
        ---
        save_data_bd: SAVE_BD
                genericmap     ( n              => n
                               )
                port map       ( GATECLK        => EPP_GCLK_SAVE,
                                 DATA_IN        => EPP_BCNT_SAVE,
                                 AVE_IN         => EPP_ACNT_SAVE,
                                 SQR_IN         => EPP_SQR_SAVE,
                                 SIG_IN         => EPP_SIG_SAVE,
                                 TRIGGER_IN     => EPP_TRIGGER_SAVE
                               );



-- generate  4Mhz clock
CLK <= '1' after 118 ns when CLK = '0'
    else '1' when CLK = 'U'
    else '0' after 119 ns;

RESET <= '1', '0' after 500 ns;
LOADLUT <= '1' , '0' after 1000 ns;


end Ait;
```

B.4.14 LOAD-Module

```
--------------------------------------------------------------------------
-- LOAD: Load Data from File  10.10.00 TS/IAAT --------------------------
--------------------------------------------------------------------------

---
-- LOAD Data / Test Environment
---



--------------------------------------------------------------------------
------
-- Input Signals:
--
-- CLK/RESET     : Signals from EPP-TOP (Testbench).
--
------
-- Task: Read Data from Disk-File DET.dat and send it to EPP_TOP as
--       Event-Packets.
--
--
------
-- Output Signals:
--
-- RNE_OUT       : RNE-Pipeline Signal to EPP_TOP. This Signal will be
--                 produced later from the EPA-Unit.
-- REG_OUT       : 168 Bit Event-Packet to EPP_TOP.
--
-- REG_OUT       [ 167       Timestamp      151 : 150   Spare   144 ]
--     Hit 1     [ 143          130 : 129        121 :     120      ]
--     Hit 2     [ 119          106 : 105         97 :      96      ]
--     Hit 3     [  95           82 :  81         73 :      72      ]
--     Hit 4     [  71           58 :  57         49 :      48      ]
--     Hit 5     [  47           34 :  33         25 :      24      ]
--     Hit 6     [  23           10 :  9           1 :       0      ]
--
-- FILE_DATA_VD  : Signal that indicate valid file-datas
--                 1 = data found , 0 no further datas in file.
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity LOAD is
        generic ( cycle           : time := 1000 ns;
                  n               : in positive := 4;
```

```
                  gfreq           : in positive := 125 -- 1/8 Clk-Frequency
                );
        port    ( CLK             : in std_logic;
                  RESET           : in std_logic;
                  RNE_OUT         : out std_logic;
                  REG_OUT         : out unsigned (167 downto 0);
                  FILE_DATA_VD    : out std_logic
                );
end LOAD;

architecture AIT of LOAD is

signal RNE      : std_logic;
signal RNE_1    : std_logic;
signal RNE_2    : std_logic;
signal REG      : unsigned (167 downto 0);

begin

        ---
        -- detector_data:
        ---
        load_data:
        process

        variable d_next_event       : integer := 1;
        variable d_time             : integer;
        variable d_spare            : integer;

        variable d_erg_1            : integer; -- Hit 1
        variable d_SX_1             : integer; -- Anode X -- Segment X Addr
        variable d_SY_1             : integer; -- Anode X -- Segment Y Addr
        variable d_E_1              : integer; -- Electrode Addr
        variable d_vd_1             : integer;

        variable d_erg_2            : integer; -- Hit 2
        variable d_SX_2             : integer; -- Anode X -- Segment X Addr
        variable d_SY_2             : integer; -- Anode X -- Segment Y Addr
        variable d_E_2              : integer; -- Electrode Addr
        variable d_vd_2             : integer;

        variable d_erg_3            : integer; -- Hit 3
        variable d_SX_3             : integer; -- Anode X -- Segment X Addr
        variable d_SY_3             : integer; -- Anode X -- Segment Y Addr
        variable d_E_3              : integer; -- Electrode Addr
        variable d_vd_3             : integer;

        variable d_erg_4            : integer; -- Hit 4
        variable d_SX_4             : integer; -- Cathode Y -- Segment X Addr
        variable d_SY_4             : integer; -- Cathode Y -- Segment Y Addr
        variable d_E_4              : integer; -- Electrode Addr
        variable d_vd_4             : integer;

        variable d_erg_5            : integer; -- Hit 5
        variable d_SX_5             : integer; -- Cathode Y -- Segment X Addr
        variable d_SY_5             : integer; -- Cathode Y -- Segment Y Addr
        variable d_E_5              : integer; -- Electrode Addr
        variable d_vd_5             : integer;
```

```
variable d_erg_6            : integer; -- Hit 6
variable d_SX_6             : integer; -- Cathode Y -- Segment X Addr
variable d_SY_6             : integer; -- Cathode Y -- Segment Y Addr
variable d_E_6              : integer; -- Electrode Addr
variable d_vd_6             : integer;

variable done: boolean      := FALSE;

---
-- file data : Read data from file "stimuli/DET.dat";
---
file data : text is in "stimuli/DET.dat";

procedure read_data_line is

      variable L     : line;

      begin
               if (not endfile(data)) then
                       readline(data, L);
                       read(L, d_next_event);  -- number of cycles till next event
                       if d_next_event = 0 then
                               read(L, d_time);  -- detector data
                               read(L, d_spare); -- detector data

                               read(L, d_erg_1); -- detector data
                               read(L, d_SX_1);  -- detector data
                               read(L, d_SY_1);  -- detector data
                               read(L, d_E_1);   -- detector data
                               read(L, d_vd_1);  -- detector data

                               read(L, d_erg_2); -- detector data
                               read(L, d_SX_2);  -- detector data
                               read(L, d_SY_2);  -- detector data
                               read(L, d_E_2);   -- detector data
                               read(L, d_vd_2);  -- detector data

                               read(L, d_erg_3); -- detector data
                               read(L, d_SX_3);  -- detector data
                               read(L, d_SY_3);  -- detector data
                               read(L, d_E_3);   -- detector data
                               read(L, d_vd_3);  -- detector data

                               read(L, d_erg_4); -- detector data
                               read(L, d_SX_4);  -- detector data
                               read(L, d_SY_4);  -- detector data
                               read(L, d_E_4);   -- detector data
                               read(L, d_vd_4);  -- detector data

                               read(L, d_erg_5); -- detector data
                               read(L, d_SX_5);  -- detector data
                               read(L, d_SY_5);  -- detector data
                               read(L, d_E_5);   -- detector data
                               read(L, d_vd_5);  -- detector data

                               read(L, d_erg_6); -- detector data
                               read(L, d_SX_6);  -- detector data
```

```
                                read(L, d_SY_6);  -- detector data
                                read(L, d_E_6);   -- detector data
                                read(L, d_vd_6);  -- detector data
                        end if;
                else
                        done := TRUE;
                end if;
        end;
begin

---
-- if rising_edge(RNE) then
-- convert detector data to Bit_Vectors
---
        read_data_line;

        if d_next_event > 0 then
                FILE_DATA_VD <= '0';
                wait for d_next_event * cycle;
        else
                if not done then
                        FILE_DATA_VD <= '1';
                else
                        FILE_DATA_VD <= '0';
                end if;

                REG <= (others => '0');
                REG(167 downto 152)  <= CONV_UNSIGNED(d_time,16);
                REG(151 downto 144)  <= CONV_UNSIGNED(d_spare,8);

                REG(143 downto 130)  <= CONV_UNSIGNED(d_erg_1,14);
                REG(129 downto 128)  <= CONV_UNSIGNED(d_SX_1,2);
                REG(127 downto 126)  <= CONV_UNSIGNED(d_SY_1,2);
                REG(125 downto 121)  <= CONV_UNSIGNED(d_E_1,5);
                if d_vd_1 = 1 then
                        REG(120) <= '1';
                else
                        REG(120) <= '0';
                end if;

                REG(119 downto 106)  <= CONV_UNSIGNED(d_erg_2,14);
                REG(105 downto 104)  <= CONV_UNSIGNED(d_SX_2,2);
                REG(103 downto 102)  <= CONV_UNSIGNED(d_SY_2,2);
                REG(101 downto  97)  <= CONV_UNSIGNED(d_E_2,5);
                if d_vd_2 = 1 then
                        REG(96) <= '1';
                else
                        REG(96) <= '0';
                end if;

                REG(95  downto  82)  <= CONV_UNSIGNED(d_erg_3,14);
                REG(81  downto  80)  <= CONV_UNSIGNED(d_SX_3,2);
                REG(79  downto  78)  <= CONV_UNSIGNED(d_SY_3,2);
                REG(77  downto  73)  <= CONV_UNSIGNED(d_E_3,5);
                if d_vd_3 = 1 then
                        REG(72) <= '1';
                else
                        REG(72) <= '0';
```

```
                end if;

                REG(71  downto  58)  <= CONV_UNSIGNED(d_erg_4,14);
                REG(57  downto  56)  <= CONV_UNSIGNED(d_SX_4,2);
                REG(55  downto  54)  <= CONV_UNSIGNED(d_SY_4,2);
                REG(53  downto  49)  <= CONV_UNSIGNED(d_E_4,5);
                if d_vd_4 = 1 then
                        REG(48) <= '1';
                else
                        REG(48) <= '0';
                end if;

                REG(47  downto  34)  <= CONV_UNSIGNED(d_erg_5,14);
                REG(33  downto  32)  <= CONV_UNSIGNED(d_SX_5,2);
                REG(31  downto  30)  <= CONV_UNSIGNED(d_SY_5,2);
                REG(29  downto  25)  <= CONV_UNSIGNED(d_E_5,5);
                if d_vd_5 = 1 then
                        REG(24) <= '1';
                else
                        REG(24) <= '0';
                end if;

                REG(23  downto  10)  <= CONV_UNSIGNED(d_erg_6,14);
                REG(9   downto   8)  <= CONV_UNSIGNED(d_SX_6,2);
                REG(7   downto   6)  <= CONV_UNSIGNED(d_SY_6,2);
                REG(5   downto   1)  <= CONV_UNSIGNED(d_E_6,5);
                if d_vd_6 = 1 then
                        REG(0) <= '1';
                else
                        REG(0) <= '0';
                end if;

                wait until rising_edge(RNE);
        end if;
end process;




---
-- Data Out (Write Output-Register with each RNE).
---
data_out:
process (CLK)
        variable last_RNE : std_logic;
begin
        if rising_edge(CLK) then
                if RESET = '1' then
                        last_RNE  := '0';
                        REG_OUT   <= (others => '0');
                else
                        if RNE = '1' and last_RNE ='0' then
                                REG_OUT <= REG;
                        end if;
                        last_RNE := RNE;
                end if;
        end if;
end process;
```

```
        ---
        -- RNE Generator (Read Next Event pulses for intern use only).
        ---

        process (CLK)
                variable CNT : integer range 0 to 127;
                variable last_RESET: std_logic;
        begin
                if rising_edge(CLK) then
                        if RESET = '1' then
                                RNE_1    <= '0';
                                RNE_2    <= '0';
                                RNE      <= '0';
                                RNE_OUT  <= '0';
                                CNT      := 0;
                        else
                                if CNT < 100 then
                                        CNT     := CNT + 1;
                                        RNE_1   <= '0';
                                        RNE_OUT <= '0';
                                else
                                        CNT     := 0;
                                        RNE_1   <= '1';
                                        RNE_OUT <= '1';
                                end if;

                                if last_RESET = '1' then
                                        RNE_2   <= '1';
                                else
                                        RNE_2   <= '0';
                                end if;

                                RNE <= RNE_1 or RNE_2;
                        end if;
                        last_RESET := RESET;
                end if;
        end process;
end AIT;
```

B.4.15 SAVE_TM-Module

```
--------------------------------------------------------------------------
-- Save Datas to disk file, 19.09.00, TS/IAAT ---------------------------
--------------------------------------------------------------------------

---
-- SAVE_TM (Save Telemetry Data) / Test Environment
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- RNE           : RNE Signal Input, synchronize save of Event-Packets.
-- TMCLK         : special signal that indicate a valid Event-Packet to save.
-- DATA_IN       : Input Event-Packet to be saved.
--
-- DATA_IN:      [ 44   Time-Stamp   29 : 28  Location Anode 22
--               : 21 Location Cathode 15 : 14    Energy    7
--               : 6 DOI 4 : 3 VD-DET : 2 VD-EPP : 1 ERROR 0 ]
--
--
------
-- Task:  Save Event-Packet (DATA_IN) synchron with RNE, if TMCLK and
--        indicate that the Event-Packet is valid.
--
------
-- Output Signals:
-- Data disk-file:  TELEM.dat
--
--------------------------------------------------------------------------


-- VHDL script starts here:



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

library STD;
use STD.textio.all;

entity SAVE_TM is
        port    ( RNE             : in std_logic;
                  TMCLK           : in std_logic;
                  DATA_IN         : in unsigned(44 downto 0);
                  FILE_DATA_VD    : in std_logic
                );
end SAVE_TM;


architecture AIT of SAVE_TM is
```

```
FILE out_file     : text is out "results/TELEM.dat";

begin

        ---
        -- write results into file
        ---
        process (RNE)

        variable l          : line;

        begin
                if  rising_edge(RNE)  then
                        if TMCLK = '1' then
                                write(l, now);
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(44 downto 29)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(28 downto 22)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(21 downto 15)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(14 downto 7)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(6 downto 4)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(3)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(2)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(1 downto 0)));
                                writeline(out_file, l);
                        end if;
                end if;
        end process;
end AIT;
```

B.4.16 SAVE_RB-Module

```
---------------------------------------------------------------------------
-- Save Datas to disk file,  19.09.00, TS/IAAT --------------------------
---------------------------------------------------------------------------

---
-- SAVE_RB (Save Ring Buffer Data) / Test Environment
---

---------------------------------------------------------------------------
------
-- Input Signals:
--
-- RNE            : RNE Signal Input, synchronize save of Event-Packets.
-- RBCLK          : special signal that indicate a valid Event-Packet to save.
-- TMCLK          : special signal that indicate a valid Event-Packet to save.
-- DATA_IN        : Input Event-Packet.
--
-- DATA_IN:       [ 40   Time-Stamp   25 : 24  Location Anode 18
--                : 17 Location Cathode 11 : 10  Energy  3 : 2 DOI 1]
--
------
-- Task:  Save Event-Packet (DATA_IN) synchron with RNE, if RBCLK and
--        TMCLK indicate that the Event-Packet is valid.
--
------
-- Output Signals:
-- Data disk-file:  RINGB.dat
--
---------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity SAVE_RB is
        port    ( RNE           : in std_logic;
                  RBCLK         : in std_logic;
                  TMCLK         : in std_logic;
                  DATA_IN       : in unsigned(40 downto 0)
                 );
end SAVE_RB;


architecture AIT of SAVE_RB is

FILE out_file     : text is out "results/RINGB.dat";

begin
```

```
        ---
        -- write results into file
        ---
        process (RNE)

        variable l         : line;

        begin
                if  rising_edge(RNE)  then
                        if RBCLK = '1' and TMCLK ='1' then
                                write(l, now);
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(40 downto 25)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(24 downto 18)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(17 downto 11)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(10 downto 3)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(2  downto 0)));
                                writeline(out_file, l);
                        end if;
                end if;
        end process;
end AIT;
```

B.4.17 SAVE_BS-Module

```
---------------------------------------------------------------------------
-- Save Datas to disk file,  19.09.00, TS/IAAT ---------------------------
---------------------------------------------------------------------------

---
-- SAVE_BS (Save Burst Snooper Data) / Test Environment
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- RNE           : RNE Signal Input, synchronize save of Event-Packets.
-- BSCLK         : special signal that indicate a valid Event-Packet to save.
-- DATA_IN       : Input Event-Packet.
--
-- DATA_IN:      [ 40   Time-Stamp   25 : 24  Location Anode 18
--               : 17 Location Cathode 11 : 10  Energy  3 : 2 DOI 1]
--
------
-- Task:  Save Event-Packet (DATA_IN) synchron with RNE, if BSCLK indicate
--        that the Event-Packet is valid.  (for diagnostics)
--
------
-- Output Signals:
-- Data disk-file:  BSN.dat
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity SAVE_BS is
        port    ( RNE           : in std_logic;
                  BSCLK         : in std_logic;
                  DATA_IN       : in unsigned(40 downto 0);
                  FILE_DATA_VD  : in std_logic
                 );
end SAVE_BS;


architecture AIT of SAVE_BS is

FILE out_file     : text is out "results/BSN.dat";

begin
```

```
        ---
        -- write results into file
        ---
        process (RNE)

        variable l         : line;

        begin
                if  rising_edge(RNE)  then
                        if BSCLK = '1' then
                                write(l, now);
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(40 downto 25)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(24 downto 18)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(17 downto 11)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(10 downto 3)));
                                write(l, ' ');
                                write(l, CONV_INTEGER(DATA_IN(2 downto 0)));
                                writeline(out_file, l);
                        end if;
                end if;
        end process;
end AIT;
```

B.4.18 SAVE_BD-Module

```
--------------------------------------------------------------------------
-- Save Datas to disk file ,  23.03.01, TS/IAAT -------------------------
--------------------------------------------------------------------------

---
-- SAVE_BD (Save Burst Detector Data) / Test Environment
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- GATECLK       : special clock signal that indicate a Count
--                 ready to save.
-- DATA_IN       : Input Signal for actual Count.
-- AVE_IN        : Input Signal for average Count.
--
------
-- Task:  Save the actual count and the average count to disk file,
--        triggerd by GATECLK. (for diagnostics)
--
------
-- Output Signals:
-- Data disk-file: BSTDT.dat
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity SAVE_BD is
        generic  ( n             : in positive );
        port     ( GATECLK       : in std_logic;
                   DATA_IN       : in unsigned ((4*n)-1 downto 0);
                   AVE_IN        : in unsigned ((4*n)-1 downto 0);
                   SQR_IN        : in unsigned ((4*n)-1 downto 0);
                   SIG_IN        : in unsigned ((4*n)-1 downto 0);
                   TRIGGER_IN    : in std_logic
                 );
end SAVE_BD;


architecture AIT of SAVE_BD is

        FILE out_file     : text is out "results/BSTDT.dat";

begin
```

```
        ---
        -- write results into file at every Gate Clock
        ---
        process (GATECLK)

        variable l       : line;

        begin
               if rising_edge(GATECLK) then
                       write(l, now);
                       write(l, ' ');
                       write(l, CONV_INTEGER(DATA_IN));
                       write(l, ' ');
                       write(l, CONV_INTEGER(AVE_IN));
                       write(l, ' ');
                       write(l, CONV_INTEGER(SQR_IN));
                       write(l, ' ');
                       write(l, CONV_INTEGER(SIG_IN));
                       write(l, ' ');
                       write(l, CONV_INTEGER(TRIGGER_IN));
                       writeline(out_file, l);
               end if;
        end process;
end AIT;
```

B.4.19 SAVE_TST-Module

```
--------------------------------------------------------------------------
-- Save Datas to disk file, 19.09.00, TS/IAAT ----------------------------
--------------------------------------------------------------------------

---
-- SAVE_TST (Save all Event-Packets) / Test Environment
---

--------------------------------------------------------------------------
------
-- Input Signals:
--
-- RNE           : RNE Signal Input, synchronize save of Event-Packets.
-- DATA_IN       : Input Event-Packet.
--
-- DATA_IN:      [ 45   Time-Stamp   30 : 29  Location Anode 23
--               : 22 Location Cathode 16 : 15  Energy  8 : 7 DOI 5
--               : 4 VD-Det : 3 VD-EPP : 2 VD-Noise : 1 Error 0 ]
--
------
-- Task:  Save all Event-Packets synchron with RNE to disk file,
--        (for diagnostics)
------
-- Output Signals:
-- Data disk-file: TEST.dat
--
--------------------------------------------------------------------------


-- VHDL script starts here:


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

library STD;
use STD.textio.all;

entity SAVE_TST is
        port    ( RNE           : in std_logic;
                  DATA_IN       : in unsigned(45 downto 0)
                );
end SAVE_TST;


architecture AIT of SAVE_TST is

        FILE out_file     : text is out "results/TEST.dat";

begin

        ---
        -- write results into file
        ---
```

```
        process (RNE)

                variable l        : line;

        begin
                if  rising_edge(RNE)  then
                        write(l, now);
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(45 downto 30)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(29 downto 23)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(22 downto 16)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(15 downto 8)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(7 downto 5)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(4)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(3)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(2)));
                        write(l, ' ');
                        write(l, CONV_INTEGER(DATA_IN(1 downto 0)));
                        writeline(out_file, l);
                end if;
        end process;
end AIT;
```

Literaturverzeichnis

[Aprile et al. 1994] Aprile, E., Bolotnikov, A., Chen, D., Tawara, H., Xu, F. et al., 1994, The polarization sensitivity of the liquid xenon imaging telescope., The Astrophysical Journal Supplement Series, 92, 689-692.

[Ashenden 1990] Ashenden, P.J., 1990, The VHDL Cookbook, Internet: http://tech-www.informatik.uni-hamburg.de/vhdl/ oder ftp://ftp.cs.adelaide.edu.au/pub/VHDL-Cookbook/ .

[Ashenden 1998] Ashenden, P.J., 1998, The Student's Guide to VHDL, Morgan Kaufmann Publishers, Inc., ISBN 1-55860-520-7.

[Baring et al. 1998] Baring, M.G. & Harding, A.K., 1998, Adv. Space Res. 22(7), 1115.

[Blake et al. 1974] Blake, R.L., Burek, A.J., Fenimore, E.E., Puetter, R., 1974, Solar X-Ray photography with multiplex pin-hole camera., Rev. Sci. Instrum., 45 (4), 513-516.

[Bloom et al. 1999] Bloom, J.S., et al., 1999, Nature 401, 453.

[Boella et al. 1997] Boella, G. et al., 1997, A&A Suppl. Ser, 122, 299.

[Bonnell & Klebesadel 1996] Bonnell, J.T. & Klebesadel, R.W., 1996, in: C. Kouveliotou, M.F. Briggs, & G.J. Fishman (eds.), 3rd Huntsville Symposium on Gamma-Ray Bursts (AIP: New York), AIP Conf. Proc. 384, 977.

[Brainerd 1994] Brainerd, J.J., 1994, ApJ 428, L1.

[Briggs et al. 1996] Briggs, M. et al., 1996, ApJ 459, 40.

[Briggs et al. 1999] Briggs, M. et al., 1999, ApJ Suppl. Ser. 122, 503.

[Castro-Tirado et al. 1999] Castro-Tirado, A.J., 1999, Astroph. Sp. Sci. 263, 15, astro-ph/9903187

[Costa et al. 1997] Costa, E. et al., 1997, IAU Circ. 6533.

[Costa et al. 1997b] Coste, E. et al., 1997, IAU Circ. 6572.

[Costa et al. 1998] Costa, E. et al., 1998, Adv. Space Res. 22 (7), 1129.

[Dar 1999] Dar, A., 1999, GCN 346.

[Dar 2001] Dar, A., 2001, astro-ph/0101007.

[Dean 1984] Dean, A.J., 1984, Imaging systems for low energy γ-ray astronomy., Nuclear Instruments and Methods in Physics Research, 221, 265-277.

[Eichler et al. 1989] Eichler, D., et al., 1989, Nature, 340, 126.

[Fenimore et al. 1995] Fenimore, E.E. & Bloom, J.S., 1995, ApJ 453, 25.

[Fenimore et al. 1999] Fenimore, E.E., Ramirez-Ruiz, E., Wu, B., 1999, Astrophys. J. Lett , preprint astro-ph/9902007, LA-UR-99-529.

[Fishman 1981] Fishman, G.J., 1981, Astroph. Sp. Sci. 75, 125.

[Fishman et al. 1988] Fishman, G.J., Inan, U.S., 1988, Nature 331, 418.

[Fishman et al. 1993] Fishman G.L. et al., 1993, A&A Suppl. Ser, 97, 17.

[Fishman 1999] Fishman, G.J., 1999, A&A, Suppl. Ser. 138, 395.

[Frail et al. 1997] Frail, D.A. et al., 1997, ApJ 483, L91.

[Frontera 1998] Frontera, F., 1998, in: M.S. Potgieter, B.C. Raubenheimer, & D.J. van der Wait (eds.), 25th International Cosmic Ray Conference, Vol. 8, (World Scientific: Singapore), p. 307; astro-ph/9802157.

[Fruchter et al. 1999] Fruchter, A., et al., 1999, GCN 255.

[Fryer et al. 1999] Fryer, C.L., Woosley, S.E., Hartmann, D.H., 1999, ApJ 526, 152.

[Galama et al. 1999] Galama, T.J. et al., 1999, preprint astro-ph/9907264.

[Galama et al. 1997] Galama, T.J. et al., 1997, A&A 321, 229.

[Gehrels 1999] Gehrels, N. 1999, in Swift, a Panchromatic Gamma Ray Burst Mission, GSFC reprint 2/5/99, proposal submitted in response to AO-98-OSS-03.

[Gerthsen et al. 1982] Gerthsen, C., Kneser, H.O., Vogel, H., 1982, Physik, Springer-Verlag, (14. Auflage), ISBN 3-540-11369-X.

[Giacconi et al. 1968] Giacconi, R., Gursky, H., Van Speybroeck, L.P., 1968, Observational techniques in X-ray astronomy, Annual Reviews of Astronomy and Astrophysics, 6, 373-416.

[Golze 1995] Golze, U., 1995, VLSI-Entwurf eines RISC-Prozessors, Vieweg.

[Groot et al. 1997] Groot, P.J. et al., 1997, IAU Circ. 6584.

[Hakkila et al. 1994] Hakkila, J. et al., 1994, ApJ 422, 659.

[Hartmann 1995] Hartmann, D.H., 1995, A&A Rev. 6, 225.

[Hartmann 1996] Hartmann, D.H., 1996, A&A Suppl. Ser. 120,31.

[Hjorth et al. 1999] Hjorth, J. et al., 1999, GCN 403.

[Horack et al. 1996] Horack, J.M., Mallozzi, R.S., & Koshut, T.M., 1996, ApJ 466, 21.

[Hurley 1999] Hurley, K., 1999, preprint astro-ph/9912061.

[Iwamoto et al. 1998] Iwamoto, K. et al, 1998, Nature 395,672.

[Kalemci et al. 2000] Kalemci, E., Matteson, J.L., 2000, Investigation of charge sharing among electrode strips for a CdZnTe detector, CASS @ USCD.

[Klebesadel et al. 1973] Klebesadel, R.W., Strong, I.B., & Olson, R.A., 1973, Observations of Gamma-Ray Bursts of Cosmic Origin, ApJ. 182, L85.

[Klose 1994] Klose, S., 1994, ApJ 423, L23.

[Klose 1995] Klose, S., 1995, A&A 303, 75.

[Klose 2000] Klose, S., 2000, astro-ph/0001008.

[Kouveliotou 1995] Kouveliotou, C., 1995, Astroph. Sp. Sci. 231, 49.

[Krolik et al. 1991] Krolik, J., Pier, E., 1991, ApJ, 373, 277.

[Kulkarni et al. 1999] Kulkarni, S.R. et al., 1999, Nature 398, 389.

[Lamb 1995] Lamb, D.Q., 1995, PASP 107, 1152.

[Lamb 1997] Lamb, D.Q., 1997, in: Gravitation, R.E.Schielicke (ed.), Rev. Mod. Astron. 10, 101.

[Lamb et al. 1999] Lamb, D.Q., et al., 1999, A&A Suppl. Ser. 138, 479.

[Laros et al. 1986] Laros, J.G. et al., 1986, Nature 322, 152.

[Lee et al. 1997] Lee, T.T. & Petrosian, V., 1997, ApJ 474, 37.

[Lehmann et al. 1994] Lehmann, G., Wunder, B., Selz, M., 1994, Schaltungsdesign mit VHDL. Fanzis Verlag, ISBN 3-7723-6163-3, oder im Internet als PDF-Dokument: unter http://tech-www.informatik.uni-hamburg.de/vhdl/ oder http://www.eda.ei.tum.de/forschung/vhdl/

[Li et al. 1992] Li, H. & Liangm E.P., 1992, ApJ 400, L59.

[Maeder] Mäder, A., VHDL Kurzbeschreibung, Universität Hamburg - Fachbereich Informatik, Internet: http://www.eda.ei.tum.de/forschung/vhdl.

[Mao & Paczynski 1992] Mao, S. & Paczynsk, B., 1992, ApJ 388, L45.

[Meegan et al. 1992] Meegan, C.A. et al., 1992, Nature 355, 143.

[Norris et al. 1995] Norris, J.P. et al., 1995, ApJ 439, 542.

[Owens et al. 1995] Owens, A., Schaefer, B.E., & Sembay, S., 1995, ApJ 447, 279.

[Paczynski et al. 1993] Paczynski, B., Rhoads, J., 1993, ApJ 418,L5.

[Postnov 1999] Postnov, K.A., 1999, Cosmic gamma-ray bursts, Physics Of Our Days, Physics-Uspekhi, 42(5) 469-480

[Reichart 1999] Reichart, D., 1999, ApJ 521, L111.

[Rothschild et al. 1999] Rothschild, E.R., Matteson J.L., Heindl, W.A., Peterson, L.E., Staubert, R., Kendziorra, E., Wilms, J., 1999, Technical Proposal: The Coded Mask Imager for Gamma-Ray Burst and Hard X-Ray Source Monitoring on GLAST, Center for Astrophysics and Space Science (CASS) University of California San Diego La Jolla, CA/USA; Institut für Astronomie und Astrophysik (IAAT) Universität Tübingen, Deutschland.

[Sari et al. 1999] Sari, R., Piran, T., Halpern, J., 1999, ApJ, 519, L17.

[Tegmark et al. 1996] Tegmark, M. et al., 1996, ApJ 468, 214.

[Tinney et al. 1998] Tinney, C. et al., 1998, IAU Circ. 6896.

[Waxman 1997] Waxman, E., 1997, ApJ, 485, L5.

[Weekes et al. 1989] Weekes, T.C. et al., 1989, Observation of TeV Gamma-Rays from Crab Nebula using the Atmospheric Cerencov Imaging Technique., The Astrophysical Journal, 342, 379-395.

[Wheeler 1999] Wheeler, J., 1999, astro-ph/9909096.

[Wijers et al. 1997] Wijers, R.A.M.J., Rees, M.J., Meszaros, P., 1997, MNRAS 288, L51.

[Woosley et al. 1999] Woosley, S.E., MacFadney, A.I., Heger, A., 1999, in: M. Livio, K. Sahu, & N. Panagia (eds.), The Largest Explosions Since The Big Bang: Supernovae and Gamma-Ray Bursts (Cambridge University Press: Cambridge), preprint astro-ph/9909034.

[van Paradijs et al. 1997] van Paradijs, J. et al., 1997, Nature 386, 686.

Danksagung

Mein herzlicher Dank gilt Herrn Prof. Rüdiger Staubert und Herrn Dr. Eckhard Kendziorra, die mir diese Diplomarbeit ermöglicht haben. Herrn Staubert danke ich für die Vergabe des Themas. Ich habe mir immer eine Diplomarbeit dieser Art gewünscht, die mein technisches und mein naturwissenschaftliches Interesse miteinander verbindet. Ganz besonders danke ich Herrn Kendziorra für die nette Betreuung während meiner Zeit am Institut und für die erste Korrektur dieser Ausarbeitung. Auch für die Tolleranz gegenüber meiner Nebenberufstätigkeit möchte ich mich nocheinmal ausdrücklich bedanken.

Bedanken möchte ich mich ferner bei Claus Dreischer für viele wertvolle Tips zum Thema VHDL und für die hilfsbereite Betreuung bei anderen Computer-Problemen. Ebenso bei Reiner Volkmer und den Leuten von der Sternwarte, die bei Fragen und Problemen stets eine offene Tür für mich hatten. Ausserdem bei meinem Tischnachbarn Wolfgang Wettlaufer von der AVT, für viele interessante Gespräche im Umfeld der Astronomie, und das er es ein Jahr lang mit mir und meinen surrenden Rechnern ausgehalten hat.

Zum Schluss möchte ich mich bei meinen Eltern bedanken, für ihr Vertrauen und das sie mir das Studium ermöglicht haben. Ich danke auch meinen Freunden und Kollegen, die mich während meiner Studienzeit so tatkräftig unterstützt haben. Insbesondere seien in diesem Zusammenhang Andreas Nagel, Jürgen Österle und Jan-Peer Schneider genannt. Ich danke Rudi Scholze von der Firma HP (OVBU) für die ständigen Vertragsverlängerungen, ohne die ich mein Studium sicher niemals fortgesetzt hätte.

Danke!