Ruprecht-Karls-Universität Heidelberg
Fakultät für Physik und Astronomie
Max - Planck - Institut für Kernphysik

IHEP 98-02
HD-ASIC-39-0298

Datenkompression für die Auslese des ATLAS Level-1 Triggers

Diplomarbeit

von

Bernhard Niemann

ASIC-Labor http://www.ihep.uni-heidelberg.de/Asic/
Schröderstraße 90 D-69120 Heidelberg

Fakultät für Physik und Astronomie

Ruprecht-Karls-Universität Heidelberg

Diplomarbeit
im Studiengang Physik

vorgelegt von
Bernhard Niemann
aus Karlsruhe

Dezember 1997

Datenkompression für die Auslese des ATLAS Level-1 Triggers

Die Diplomarbeit wurde von Bernhard Niemann ausgeführt am
Institut für Hochenergiephysik der Universität Heidelberg
unter der Betreuung von
Herrn Prof. Dr. K. Meier

In dieser Arbeit werden Möglichkeiten zur Datenkompression für die Auslese des ATLAS Level-1 Kalorimeter-Triggers untersucht. Die Ergebnisse von Simulationen mit Huffman Coding und Run-Length Coding, sowie für zwei speziell für die Nullunterdrückung von verrauschten Daten mit festen Wortgrenzen entwickelten Algorithmen (Huffman-Inspired und Difference Coding), werden vorgestellt. Die Parameter der verschiedenen Algorithmen wurden dabei an die zu erwartenden Daten angepaßt. Demnach können die Daten um einen Faktor 2-3 komprimiert werden, wenn alle Daten von 0 GeV bis 255 GeV, inklusive des Rauschuntergrundes von 0.5 GeV, mit 10 bit Auflösung ausgelesen werden sollen.

Die getesteten Algorithmen wurden auf einem ASIC, dem Readout Merger ASIC (RemASIC) implementiert. Die Parameter der Verfahren wurden entsprechend den Ergebnissen der Simulation gewählt. Im Huffman und Run-Length Modus können sowohl 8 bit, als auch 10 bit Daten verarbeitet werden, während die anderen beiden Modi nur mit 10 bit Daten betrieben werden können. Für das Run-Length Encoding kann ein Energie-Cut gewählt werden, der zu einer drastischen Erhöhung des Kompressionsfaktors führt. Die komprimierten Daten werden in 32 bit Datenworte formatiert. Ein erster Test des RemASIC wurde erfolgreich durchgeführt.

Data Compression for the ATLAS Level-1 Calorimeter Trigger:

The feasibility of data compression for the readout of the ATLAS level-1 calorimeter trigger has been investigated. Simulations are presented using Huffman and run-length coding as well as two compression schemes developed for zero suppression of noisy data with codes of fixed size (Huffman-inspired and difference coding). Parameters of these algorithms have been tuned to deliver best results with expected data. Compression factors of 2-3 may be obtained, if all data between 0 GeV and 255 GeV including the noise background of 0.5 GeV are to be read out with a resolution of 10 bit.

The tested algorithms have been implemented on an ASIC, the Readout Merger ASIC (RemASIC). Parameters were chosen according to the results obtained from the simulation. The Huffman and run-length coders are able to process 8 bit or 10 bit data, the other algorithms expect 10 bit data as input. When compressing in run-length mode an energy cut may be applied, which impressively increases compression factors. Compressed output data are formatted in 32 bit words. A first test of the RemASIC has been carried out successfully.

Inhaltsverzeichnis

Einleitung

Moderne Hochenergiephysik-Experimente produzieren Daten in der Größenordnung von einigen 10-100 GByte/s. Um solche Datenmengen für eine spätere Analyse zu speichern werden immer aufwendigere Datenübertragungssysteme benötigt, die in der Lage sind solche Datenmengen vom Detektor bis zu den Massenspeichern zu befördern.

Die Entwicklung von Bussystemen mit der entsprechenden Bandbreite ist die eine Möglichkeit, eine Reduktion des anfallenden Datenvolumens ohne Informationsverlust die andere. In dieser Arbeit wird die zweite Methode, die verlustfreie Datenkompression, näher untersucht. Die mathematisch-theoretische Grundlage der Datenkompression bildet die Informationstheorie. Sie wurde vor etwa 50 Jahren, mit der Verknüpfung von Entropie und Kommunikation durch Claude Shannon, Gegenstand der Forschung. Eine kurze Einführung der wichtigsten Begriffe findet sich in Anhang A.

Diese Arbeit beschäftigt sich in zwei Teilen mit der Datenkompression. Im ersten Teil (Kapitel 3 und 4) werden verschiedene Verfahren und die Ergebnisse ihrer Simulation mit Testdaten vorgestellt. Im zweiten Teil (Kapitel 5 und 6) wird die Hardwareimplementation einer Kompressionseinheit auf einem ASIC (Application Specific Integrated Circuit) und deren Test beschrieben. Die ersten beiden Kapitel beschäftigen sich mit den Gegebenheiten des ATLAS Experimentes und dem daraus resultierenden Aussehen der zu komprimierenden Daten.

Kapitel 1

Physikalischer Hintergrund

Im Rahmen der Diplomarbeit wurde eine Datenkompressionseinheit für die Auslese des ATLAS[1]Level-1 Triggers entwickelt und auf einem ASIC[2], dem *Readout Merger ASIC* (RemASIC) implementiert. Dieses Kapitel gibt zunächst einen kurzen Überblick über das ATLAS Experiment am LHC[3], um dann auf die von Physik und Technik gegebenen Randbedingungen für die Datenkompression einzugehen.

1.1 ATLAS und LHC

Die Hochenergiephysik beschäftigt sich mit der Suche nach den fundamentalen Bausteinen der Materie. Hierzu ist es nötig, immer kleinere Strukturen auflösen zu können. Dies geschieht mit Hilfe von Streuexperimenten, die wegen der Heisenbergschen Unschärferelation bei ständig wachsenden Energien durchgeführt werden müssen. Je kleiner nämlich die Strukturen sind, die man auflösen möchte, desto größer muß der Impulsübertrag bei der Streuung werden. Für eine Darstellung der Hochenergiephysik siehe zum Beispiel [1].

Zur Durchführung der Streuexperimente werden Teilchenbeschleuniger mit möglichst hohen Strahlenergien, und Detektoren benötigt, die in der Lage sind, die interessanten Streuereignisse zu registrieren. Für das Jahr 2005 wird am europäischen Kernforschungszentrum CERN in Genf die Inbetriebnahme eines neuen Beschleunigers, des LHC, geplant. Mit ihm erhofft man sich unter anderem Klarheit über folgende Aspekte der modernen Teilchenphysik, die mit den bisher existierenden Experimenten nicht geklärt werden konnten:

- Existenz des Higgs Bosons
- Suche nach supersymmetrischen (SUSY) Teilchen
- Substruktur der Quarks

[1] ATLAS: *A Toroidal LHC Apparatus*
[2] ASIC: *Application Specific Integrated Circuit*
[3] LHC: *Large Hadron Collider*

Um die bei der Streuung erzeugten Sekundärteilchen beobachten zu können, gibt es am LHC insgesamt vier Detektoren, einer davon ist ATLAS.

1.1.1 Der LHC

Der LHC ist ein Collider, das heißt es werden zwei gegenläufig beschleunigte Teilchenstrahlen an bestimmten Punkten, an denen ein Detektor steht, zur Wechselwirkung gebracht. Am LHC werden zwei Protonenstrahlen in getrennten Röhren auf jeweils 7 TeV beschleunigt, womit im pp-System bei der Wechselwirkung eine Energie von

$$\sqrt{s} = 2 * 7 \text{ TeV} = 14 \text{ TeV} \tag{1.1}$$

zur Verfügung steht. Beide Strahlen bestehen aus Teilchenpaketen, den *Bunches*. Alle 25 ns treffen in den Wechselwirkungspunkten zwei Bunches aufeinander und erzeugen Sekundärteilchen. Dies bezeichnet man als *Bunch-Crossing*.

1.1.2 Aufbau des ATLAS Detektors

ATLAS ist ein Universaldetektor, mit dem möglichst viele verschiedene Produktions- und Zerfallskanäle beobachtet werden sollen. Hierzu werden in den einzelnen Detektorkomponenten, die im folgenden kurz beschrieben werden, Teilchenbahn, Impuls und Energie gemessen. In Abbildung 1.1 ist der prinzipielle Aufbau des Detektors zu sehen. Das Streupotential ist invariant gegenüber Drehungen um die Strahlachse und Spiegelungen am Wechselwirkungspunkt. Deshalb ist der Detektor zylindersymmetrisch um den Wechselwirkungspunkt aufgebaut. Zur Beschreibung des Detektors verwendet man ein Koordinatensystem mit z-Achse in Strahlrichtung. Meistens gibt man den Azimuthwinkel ϕ und anstelle des Polarwinkels θ die Pseudorapidität η an. Differenzen in dieser sind für schnelle Teilchen mit $\beta := \frac{v}{c} \to 1$ lorentzinvariant. Sie ist definiert durch

$$\eta = -\ln \tan \frac{\theta}{2} \tag{1.2}$$

Innerer Detektor

Der innere Detektor, direkt um den Wechselwirkungspunkt, besteht aus Siliziumpixeldetektoren, Silizumstreifenzählern und Drahtkammern [2]. Sie dienen alle dem Zweck die Bahnen geladener Teilchen unmittelbar nach der Reaktion möglichst genau zu rekonstruieren. Da sich der innere Detektor in einem Magnetfeld parallel zur Strahlachse (solenoidales Feld) befindet, läßt sich aus der Bahnkrümmung auch der Teilchenimpuls rekonstruieren.

Kalorimeter

Die Bestimmung der Teilchenenergie erfolgt in den um den inneren Detektor gelegenen Kalorimetern. Im inneren elektromagnetischen Teil werden die Energien von Elektronen und Photonen gemessen. Stark wechselwirkende Teilchen (Hadronen) deponieren hier nur einen kleinen Teil ihrer Energie. Der Rest wird im äußeren hadronischen Teil nachgewiesen. Das Prinzip der Energiemessung ist in beiden Fällen dasselbe. Hochenergetische

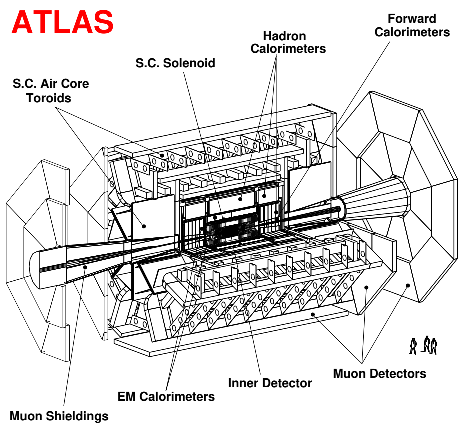

Abbildung 1.1: Der ATLAS Detektor [3]

Material	$X_0[g/cm^2]$	$\lambda_{had}[g/cm^2]$
Ar	18.9	119.7
Fe	13.8	131.9
Pb	6.3	193.7

Tabelle 1.1: Eigenschaften einiger Absorbermaterialien (nach [4])

Primärteilchen erzeugen in einem Absorbermaterial eine Kaskade von Sekundärteilchen. Die Energie der Teilchen in der Kaskade nimmt mit wachsender Eindringtiefe ab bis zum Stillstand. In einem geeigneten sensitiven Material werden die Sekundärteilchen nachgewiesen. Man erhält dann ein elektrisches Signal, das proportional zur Energie des Primärteilchens ist.

Im elektromagnetischen Kalorimeter bildet sich über Bremsstrahlung und Paarbildung die Kaskade von Sekundärteilchen aus. Die Energie der Teilchen in der Kaskade wird durch Ionisation im sensitiven Material gemessen. Im hadronischen Teil werden durch inelastische Stöße mit den Atomkernen des Absorbermaterials Hadronen gebildet. Für die Absorber wird man Materialien verwenden, die für die jeweiligen Prozesse einen besonders guten Wirkungsquerschnitt besitzen. Für den elektromagnetischen Teil ist dies Blei, für den hadronischen Eisen, Kupfer oder Wolfram. Da die hadronische Absorptionslänge um einiges größer ist, als die Strahlungslänge für elektromagnetische Prozesse (siehe Tabelle 1.1), nehmen hadronische Kalorimeter wesentlich mehr Platz ein als elektromagnetische.

Als sensitive Materialien kommen flüssiges Argon und Plastikszintillatoren zum Einsatz. Das Edelgas wird im inneren und in Strahlrichtung gelegenen Bereich verwendet, da es weniger anfällig gegenüber radioaktiver Strahlung ist. Durch eine räumliche Segmentierung der Auslese kann auch der Punkt der Energiedeposition bestimmt werden. Um im elektromagnetischen Kalorimeter einzelne Elektronen von Jets[4] unterscheiden zu können

[4]Bündel von Teilchenstrahlen

	η -Bereich	Granularität ($\eta * \phi$)	Zahl d. Kanäle
elektromagnetisch	0.. $\pm$ 3.2	≤ 0.05	214000
hadronisch	0.. $\pm$ 4.9	0.1..0.2	20100

Tabelle 1.2: Wichtige Kenngrößen des Kalorimeters (nach [2])

ist dieses feiner segmentiert als das hadronische. Eine Übersicht ist in Tabelle 1.2 gegeben.

Die Myonkammern

An das Kalorimeter schließen sich die Myonkammern an. Es sind Driftkammern unterschiedlicher Bauart, in denen die Myonen gekrümmte Spuren hinterlassen, da sie von einem zylindersymmetrischen (toroidalen) Feld mit Symmetrieachse in Strahlrichtung abgelenkt werden. Erzeugt wird das Feld von supraleitenden Magneten. Wie schon im inneren Detektor, wird der Impuls aus der Bahnkrümmung rekonstruiert.

Myonen werden zweckmäßigerweise weit außen detektiert, da sie das Kalorimeter nahezu ungehindert durchdringen, während die meisten anderen Teilchen bereits im Kalorimeter zum Stillstand kommen. Ein Teilchen, das in einer der Myonkammern eine Spur hinterläßt, ist also mit großer Sicherheit ein Myon.

1.1.3 Trigger

Wollte man alle 25 ns den gesamten Detektor auslesen, so erhielte man eine Datenrate von etwa 50 TByte/s [2]. Es ist technisch unmöglich, solch eine Datenmenge zu speichern und hinterher sinnvoll zu verarbeiten. Da zudem die interessanten Streuereignisse extrem selten sind, ist es auch sinnlos, Daten von allen Bunch-Crossings auszulesen. Aufgabe des Triggers ist es, in Echtzeit die interessanten Ereignisse zu identifizieren, und nur diese werden dann für die spätere *Offline-Analyse* gespeichert. Die Selektion der Ereignisse, und damit die Reduktion der Ereignisrate, erfolgt in drei Schritten (Level 1-3). Eine Darstellung des Datenflusses durch den Trigger ist in Abbildung 1.2 zu sehen. Die kompletten Datensätze von der Auslese eines Bunch-Crossings werden zunächst in einer Pipeline gespeichert. Der Level-1 Trigger hat nun solange Zeit für eine Entscheidung, wie der Datensatz durch die Pipeline wandert. Diese Zeit bezeichnet man als *Latency*. Wurde ein Datensatz akzeptiert, so wird er in den *Readout Buffer* kopiert und wandert im Falle einer positiven Entscheidung des Level-2 Triggers in den *Event Builder*. Von hier aus wird er dann, wenn er vom Level-3 Trigger akzeptiert wurde, für die spätere Offline-Analyse gespeichert.

Level-1 Trigger

Der Level-1 Trigger reduziert die Ereignisrate von 40 MHz auf maximal 100 kHz. Hierfür werden Daten aus dem Kalorimeter und dem Myonsystem getrennt verarbeitet. Der zentrale Triggerprozessor fällt dann aufgrund dieser Daten die Entscheidung, ob ein Ereignis interessant ist oder nicht (siehe Abbildung 1.3). Akzeptiert die erste Triggerstufe ein Ereignis, so produziert sie das Startsignal für die zweite Stufe (*Level-1 Accept*). Außerdem

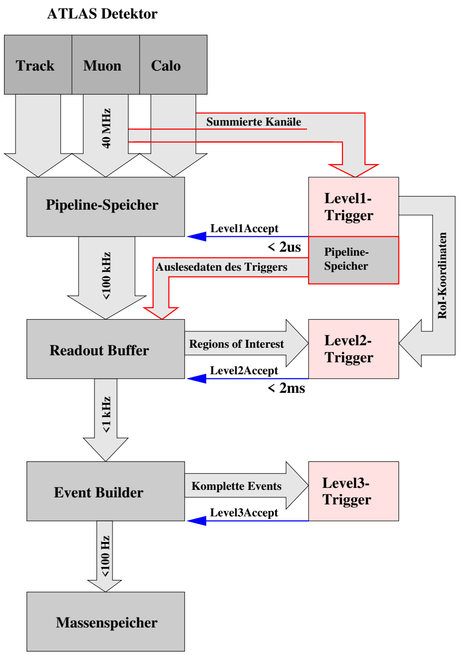

Abbildung 1.2: Datenpfad im Trigger und bei der Auslese (nach [5])

werden die Orte interessanter Energiedepositionen an den Level-2 Trigger übergeben. Man bezeichnet sie als *Regions of Interest* (RoI).

Der Level-1 Trigger hat eine Latency von maximal 2 μs und muß die Ereignisse mit der vollen Bunch-Crossing-Rate von 40 MHz verarbeiten. Daher verarbeitet er die Kalorimeterdaten nicht in voller räumlicher Auflösung, sondern er erhält sowohl vom elektromagnetischen als auch vom hadronischen Kalorimeter 4096 Kanäle. Sie entstehen durch analoge Summation der Kalorimetersignale in einem Bereich von $0.1 * 0.1$ in der η-ϕ-Ebene. Ein solcher Kanal wird als *Trigger Tower* bezeichnet.

Seine Entscheidungen trifft der Level-1 Trigger, indem er nach bestimmten physikalisch interessanten Objekten sucht und überprüft, ob ihre Energie vorher definierte Schwellen überschreitet. Liegt ihre Energie unterhalb dieser Schwellen, werden die Objekte als uninteressant angesehen. Der Myon-Trigger ist auf die Suche nach Myonen spezialisiert, während im Kalorimeter-Trigger nach folgenden Objekten gesucht wird:

- Elektronen und Photonen
- Jets
- transversale Energie $\mathrm{E_T}$
- fehlende transversale Energie $\mathrm{E_T^{miss}}$.

Unter der *transversalen Energie* versteht man

$$E_T = E \sin\theta. \tag{1.3}$$

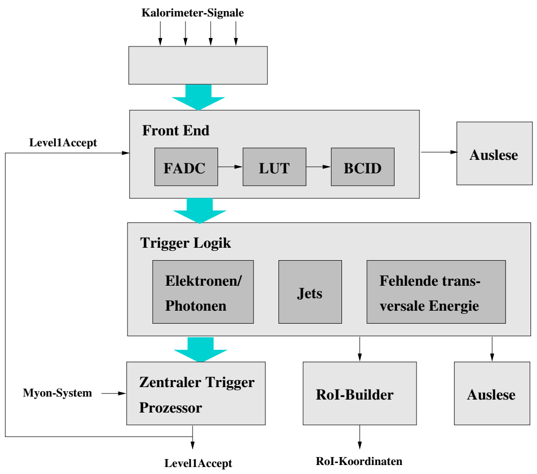

Abbildung 1.3: Der Level-1 Trigger (nach [6])

Die fehlende transversale Energie bezeichnet die Vektorsumme der transversalen Teilchenimpulse und deutet auf unidentifizierte Teilchen, wie Neutrinos, hin.

Der Level-1 Trigger muß eine große Menge von Daten möglichst schnell verarbeiten und dabei möglichst kompakt bleiben. Deswegen wird dieser Teil des Triggers mit ASICs aufgebaut. Da diese Chips speziell auf eine bestimmte Aufgabe zugeschnitten sind, lassen sich hohe Verarbeitungsgeschwindigkeiten erreichen.

Level-2 und Level-3 Trigger

In den beiden nachfolgenden Triggerstufen erfolgt eine weitere Reduktion der Ereignisrate auf weniger als 100 Hz. Beide Triggerstufen verarbeiten die Kalorimeterdaten in voller Granularität. Der Level-2 Trigger liest allerdings nur die durch die RoI-Koordinaten vorgegebenen Bereiche aus, während der Level-3 Trigger den gesamten Datensatz bearbeitet. Am Ende müssen dann nur noch 10-100 MByte/s für die spätere Offline-Analyse gespeichert werden.

Die Triggerstufen zwei und drei werden mit kommerziell erhältlichen Komponenten und parallel arbeitenden Prozessoren realisiert. Hierdurch wird die nötige Flexibilität bei der Wahl der Triggeralgorithmen erreicht.

1.2 Triggerauslese und Datenkompression

Der Kalorimeter-Trigger benutzt für seine Entscheidung nicht die volle Granularität, sondern etwa 8000 Trigger Tower, die bereits auf dem Detektor gebildet und anschließend zum Triggersystem übertragen werden. Die Trigger Tower finden ausschließlich im Level-1 Trigger Verwendung. Es ist daher notwendig, eine Möglichkeit zu schaffen, den kompletten Level-1 Trigger auszulesen. Nur so lassen sich die korrekte Funktion der Summation und Übertragung der Trigger Tower, sowie die Entscheidungen der Triggerlogik überprüfen.

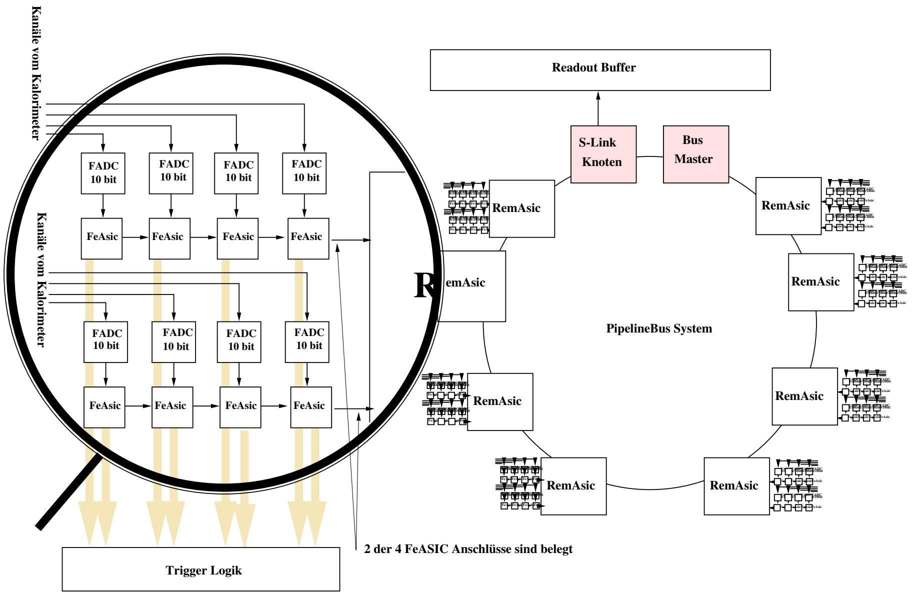

Abbildung 1.4: Auslese des Front-Ends mit PipelineBus

1.2.1 Front-End

In Abbildung 1.3 ist der Aufbau des Level-1 Triggers zu sehen. Vor der eigentlichen Triggerlogik, die in Abschnitt 1.1.3 beschrieben wurde, befindet sich das Front-End, auf dem die Auslese der rohen Triggerdaten realisiert ist. Die Aufgaben des Front-End sind im einzelnen (siehe [7]):

- Digitalisieren der analogen Kalorimeterpulse mit 10 bit
- Energiekalibration
- Durchführung des *Bunch-Crossing Identification* (BCID)
- Auslese der rohen Triggerdaten
- Serielle Übertragung der Daten zur Triggerlogik
- Bereitstellen einer Datenquelle für Testzwecke
- Zwischenspeicherung der Daten bis zum Level-1 Accept (Pipeline)

Durch das Front-End gibt es zwei Datenpfade, einen schnellen für die Übertragung der Kalorimeterdaten an die eigentliche Triggerlogik und einen langsamen für die Auslese des Front-Ends (siehe Abbildung 1.4). Auf dem Detektor erfolgt die analoge Summation der Signale aus dem Kalorimeter zu den Trigger Towern. Außerdem wird von einem Pulsformer (*Shaper*) ein bipolares Signal erzeugt, das sich über mehrere Bunch-Crossings erstreckt. Die Pulshöhe im Maximum ist proportional zur deponierten Energie.

Digitalisierung

Die summierten Analogsignale aus den Trigger Towern werden von einem FADC (*Flash Analog Digital Converter*) digitalisiert. In der bisher existierenden Prototyp Version des Front-Ends erfolgt die Digitalisierung mit 8 bit, im finalen System kommen 10 bit FADCs zum Einsatz. Über einen DAC (*Digital Analog Converter*) kann außerdem für jeden Kanal ein konstanter Offset (*Pedestal*) eingestellt werden. Er dient dazu, die Analogsignale in den Arbeitsbereich des FADCs zu bringen und die Nullinie aller Kanäle möglichst anzugleichen.

Der dynamische Bereich des FADC beträgt sowohl in der derzeitigen Version mit 8 bit als auch im finalen System mit 10 bit 256 GeV. Ein LSB[5] entspricht also ungefähr 1 GeV, beziehungsweise 0.25 GeV. Im weiteren wird der Einfachheit halber von einem dynamischen Bereich von exakt 256 GeV ausgegangen.

Front-End ASIC

Der Front-End ASIC (FeASIC) ist verantwortlich für Energiekalibration, BCID und die Auslese der Daten, für welche die Triggerlogik ein Level-1 Accept gibt. Die Energiekalibration erfolgt über eine Look-Up Table (LUT). Die digitalisierten Energiewerte werden als Adresse eines Speicherbausteins benutzt, der daraufhin einen an dieser Adresse gespeicherten Wert, die kalibrierte Energie, abgibt. Da jede Energiedeposition im Kalorimeter einen Puls erzeugt, der sich über mehrere Bunch-Crossings erstreckt, muß jedem Puls eindeutig ein Bunch-Crossing zugeordnet werden, in dem das entsprechende Teilchen erzeugt wurde. Dies geschieht über die BCID Logik, die das Pulsmaximum findet. Um unerwünschte zeitliche Verschiebungen des Pulsmaximums zu detektieren, ist es sinnvoll, von jedem Puls nicht nur das Maximum sondern auch die Werte für einige Bunch-Crossings davor und danach auszulesen. Die Zahl der ausgelesenen Werte wird als Bunch-Crossings pro Event bezeichnet. Die Auslese der Daten kann nach jedem Bearbeitungsschritt erfolgen, also vor LUT und BCID, nach LUT aber vor BCID oder nach LUT und BCID.

Der am Heidelberger ASIC-Labor entwickelte Prototyp besitzt eine LUT mit 8 bit Adresse und 10 bit Datenausgang und ermöglicht eine Auslese von ein, vier oder acht Bunch-Crossings pro Event [8]. Der Nachfolgechip wird eine LUT mit 10 bit Adressen und 8 bit Datenausgang besitzen und auch die Auslese von fünf Bunch-Crossings pro Event unterstützen [9]. Die Zahl Fünf wird favorisiert, da man so zwei Werte vor und zwei nach dem Maximum erhält, was eine optimale Kontrolle ermöglicht.

Schneller Trigger-Datenpfad

Die im FeASIC vorverarbeiteten Daten werden von einem speziellen Baustein, dem *HP Gigabit-Link* (HP G-Link) [10], serialisiert und mit einer Geschwindigkeit von 800 Mbit/s an die Triggerlogik übertragen. Dies ist der für den Betrieb von ATLAS wichtige Datenpfad. Über ihn werden die Daten übertragen, aufgrund derer der Level-1 Trigger seine Entscheidung fällt. Da in diesem Pfad keine Datenkompression stattfindet, wird nicht mehr weiter auf ihn eingegangen. Für eine genauere Darstellung siehe [6].

---

[5]LSB: <u>L</u>east <u>S</u>ignificant <u>B</u>it: Niederwertigstes bit einer Binärzahl. Im Gegensatz dazu MSB: <u>M</u>ost <u>S</u>ignificant <u>B</u>it: Höchstwertiges bit einer Binärzahl.

Langsamer Auslese-Datenpfad

Die Auslese des Kalorimeter-Triggers erfolgt über den langsamen Datenpfad. Der FeASIC verfügt über eine serielle Schnittstelle, über die nach erfolgtem Level-1 Accept verschiedene Daten aus der Pipeline ausgelesen werden können, die für die Kontrolle des Triggers wichtig sind. Im finalen System sollen die Daten von jeweils 64 Kanälen auf einem *Readout Merger ASIC* (RemASIC) gesammelt, sortiert, komprimiert und über ein Bussystem, den PipelineBus, zum Readout Buffer übertragen werden. Die derzeit vorhandene Prototypversion verfügt über vier Anschlüsse (*Ports*) für FeASICs. An jedem der Anschlüsse können bis zu vier FeASICs in Reihe geschaltet (*Daisy Chain*) betrieben werden, so daß eine Auslese von 16 Kanälen möglich ist. Vor der Kompression werden die Daten so sortiert, daß für jeden Kanal die zu einem Event ausgelesenen Bunch-Crossings in einem Block stehen. Für eine genauere Beschreibung des RemASIC siehe Abschnitt 5.1.

1.2.2 Warum Datenkompression?

Im vorangegangenen Abschnitt wurde gezeigt, daß die Auslese der rohen Triggerdaten für Test- und Kontrollzwecke unentbehrlich ist, und es wurde ein Konzept für diese Auslese, die auf dem Front-End realisiert ist, vorgestellt. Für den Transport der Daten vom Front-End zum Readout Buffer wurde das PipelineBus-Konzept entwickelt.

Die Datenein- und Ausgänge von jeweils 8 RemASICs, einem Busmaster und einem S-Link[6] Knoten werden zu einem Ring geschlossen (siehe Abbildung 1.4) . Sowohl Daten als auch Befehle (*Tokens*) propagieren auf dem Bus von einem Knoten zum nächsten. Die RemASICs geben ihre Daten auf den Bus, wenn der Busmaster ihnen dies durch ein Token erlaubt. Das Ende eines Datensatzes markiert der schreibende RemASIC ebenfalls über ein Token. Der S-Link Knoten sorgt dafür, daß die Daten wieder vom Bus genommen werden und über S-Link an den Readout Buffer übertragen werden.

Am LHC findet alle 25 ns ein Bunch-Crossing statt, was einer Rate von 40 MHz entspricht. Um Synchronisation zwischen Datenproduktion und Verarbeitung zu gewährleisten, wird die Ausleseelektronik, und damit auch RemASIC und PipelineBus mit demselben Takt betrieben. Da ein PipelineBus-Wort 32 bit breit ist, ergibt sich eine maximale Bandbreite von 152 MByte/s. In jedem Bus Ring befinden sich 8 RemASICs, die im finalen System jeweils 64 Kanäle auslesen. Für jedes Event sollen bis zu fünf Bunch-Crossings ausgelesen werden, um zusätzliche Kontrollmöglichkeiten zu besitzen. Bei 100 kHz Triggerrate muß im Schnitt alle 10 μs ein Event ausgelesen werden. Damit ergibt sich eine Datenrate von 250-300 MByte/s, je nachdem ob 8 bit oder 10 bit Daten ausgelesen werden sollen.

Die Bandbreite des PipelineBusses reicht also bei 32 bit Datenleitungen ganz offensichtlich nicht aus, um den kompletten Trigger bei maximaler Triggerrate und maximaler Anzahl der Bunch-Crossings auszulesen. Es wurde daher die Möglichkeit diskutiert, nur jedes fünfte oder zehnte Bunch-Crossing auszulesen. Sie wurde jedoch wieder verworfen, da hierbei für seltene Ereignisse wichtige Kontrollinformationen verloren gehen können. Eine andere Möglichkeit wäre ein leistungsfähigeres Bussystem zu konzipieren. Es ist allerdings

[6] Simple Link (siehe [11]).

technisch nicht unproblematisch Daten mit dieser Bandbreite über größere Entfernungen zu transportieren.

Da die Daten nicht sofort weiterverarbeitet, sondern für eine spätere Offline-Analyse gespeichert werden, bietet sich die Möglichkeit der Datenkompression an. Sie bietet zwei Vorteile gegenüber einer Auslese der unkomprimierten Daten. Erstens wird die benötigte Bandbreite des Auslesebusses verringert und zweitens benötigt man weniger Speicherplatz auf dem Massenspeicher, der die Daten aufnimmt. Die Bandbreite des PipelineBusses wird reduziert durch Tokens, die auch über den Bus laufen. Man benötigt daher Kompression um mehr als den Faktor $250/152 = 1.6$ bis $300/152 = 2$, der durch die maximale Netto-Bandbreite des PipelineBusses gegeben ist. Angestrebt wurde eine Reduktion der Daten um etwa einen Faktor 2-3. Die Datenkompression wurde realisiert auf dem RemASIC, zwischen FeASIC Interface und PipelineBus Interface.

Kapitel 2

Erzeugung von Testdaten

Vor der Implementation der Kompressionseinheit auf dem RemASIC (siehe Kapitel 5), wurden verschiedene Kompressionsalgorithmen mit Testdaten auf dem Rechner simuliert und ihre Leistungsfähigkeit bestimmt. In diesem Kapitel wird die Erzeugung der Testdaten beschrieben. Hierzu wurde das Programm *DATASIM* in C++[1] entwickelt, das den Einfluß der wichtigsten Komponenten zwischen Detektor und Kompressionseinheit auf das Aussehen der Daten simuliert.

2.1 Modell der Datenproduktion

Für die Simulation eines Problems benötigt man ein Modell, das dieses Problem in geeigneter Weise beschreibt. Bei der Erzeugung von Testdaten sind dies einige relativ einfache Funktionen und Wahrscheinlichkeitsverteilungen. Sie sollen im folgenden beschrieben und begründet werden.

2.1.1 Energieverteilung

Den Startpunkt der Simulation bildet ein mit Energie gefüllter Trigger Tower. Die Physik der Streuung und die Geometrie des Detektors verschwinden in einer Wahrscheinlichkeitsverteilung für die Energie eines Trigger Towers. Dieses Modell reicht vollkommen aus, da nicht die Herkunft eines Teilchens, oder die Teilchensorte interessiert, sondern lediglich die Daten die später komprimiert werden sollen und diese bestehen aus Energiewerten. Ein Blick auf Abbildung 2.1 zeigt, daß die Dichte der sekundär produzierten Teilchen glücklicherweise über den η-Bereich, den das Kalorimeter abdeckt, weitgehend konstant ist. Es ist daher nicht notwendig verschiedene Teile des Kalorimeters getrennt zu untersuchen.

Betrachtet man bereits gewonnene Daten von Experimenten bei niedrigen Schwerpunktsenergien ($\sqrt{s} = 546$ GeV), so zeigt sich, daß die Verteilung des transversalen Impulses $\mathrm{p_t}$ für geladene Teilchen einem Potenzgesetz gehorcht, das für kleine $\mathrm{p_t}$ in einen exponentiellen Abfall übergeht [13]. Setzt man dieses Verhalten für die Kurven in Ab-

[1] C++ ist eine objektorientierte Programmiersprache [12]

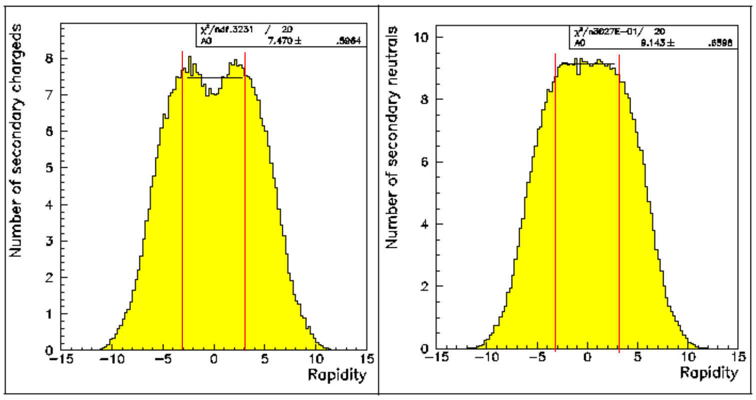

Abbildung 2.1: Teilchendichte in Abhängigkeit von η für geladene und ungeladene Teilchen (nach [14]). Die senkrechten Balken geben den vom Kalorimeter abgedeckten η-Bereich an.

bildung 2.2 an, so erhält man eine Abschätzung für die Verteilung von *Minimum Bias* Ereignissen[2].

Betrachtet man als typische physikalisch relevante Ereignisse 2 Jets mit einem Radius in der $\eta - \phi -$ Ebene von 0.4, so wird ein Bereich von

$$A_{Jet} = 2 * 0.4^2 \pi = 1.0 \tag{2.1}$$

von diesen Jets überdeckt. Da jeder Trigger Tower eine Fläche von $0.1 * 0.1$ in der η-ϕ-Ebene einnimmt werden ungefähr 100 Trigger Tower durch diese Jets mit Energie gefüllt. Die Anzahl der Zellen mit einer bestimmten Energie sollte in etwa quadratisch mit der Energie abnehmen, da die Jetenergie mit dem Radius abnimmt, die Zahl der zur Verfügung stehenden Zellen jedoch quadratisch mit dem Radius zunimmt [15]. Dieses Verhalten ist auch in einer am Institut für Hochenergiephysik in Heidelberg simulierten Verteilung in Abbildung 2.3 zu sehen. Da diese Verteilung nicht normierbar ist, mußte für Simulationszwecke ein Abschneideparameter gewählt werden, der auf 1 GeV gelegt wurde.

Das Bild, das man bis jetzt erhalten hat, besagt also, daß bei jedem Bunch-Crossing im Mittel 100 der 4000 hadronischen oder elektromagnetischen Trigger Tower mit Energien, die ungefähr proportional zu $\mathrm{E_T^{-2}}$ verteilt sind, gefüllt sein werden. Zur Berücksichtigung von Minimum Bias Ereignissen, wurde das Verhältnis von Trigger Towern mit Energien über 1 GeV zu Trigger Towern mit weniger als 1 GeV abgeschätzt. Diesem Verhältnis entsprechend bleiben in der Simulation dann Zellen leer, oder werden mit einer Energie aus der Verteilung gefüllt. Für eine Abschätzung dieses Verhältnisses (im folgenden mit Bin0/Rest-Ratio bezeichnet), wurde in Abbildung 2.2 die Flächen unter den Kurven bis 1 GeV mit denjenigen ab 1 GeV verglichen. Eine andere Abschätzung beruht auf der mittleren Energie der Minimum Bias Teilchen, die bei 400 MeV liegt. Dies bedeutet, daß im Mittel mindestens drei Teilchen in einem Trigger Tower ihre Energie deponieren müssen,

[2] Minimum Bias Ereignisse: Untergrund der uninteressanten Ereignisse

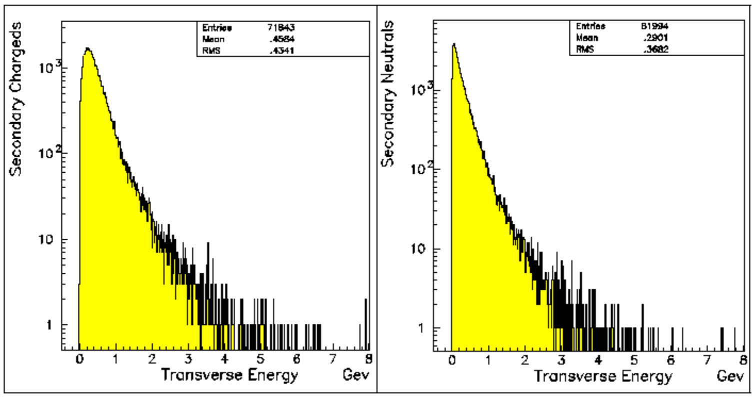

Abbildung 2.2: Teilchendichte in Abhängigkeit von E_T für geladene und ungeladene Teilchen [14]

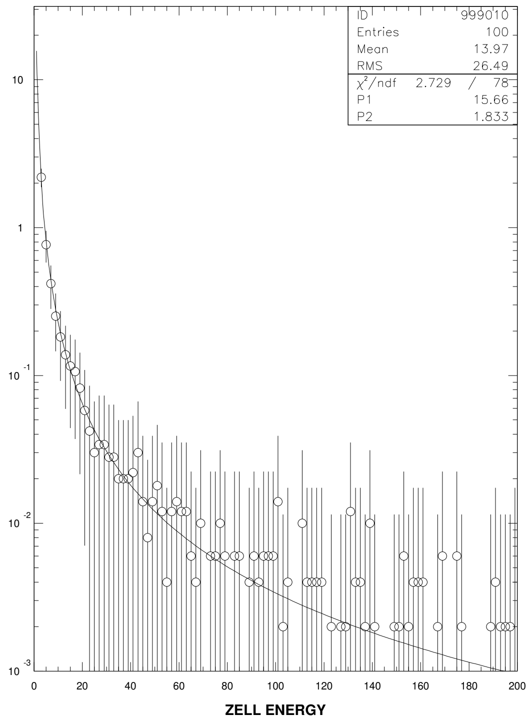

Abbildung 2.3: Simulierte Teilchendichte in Abhängigkeit von E_T [16]

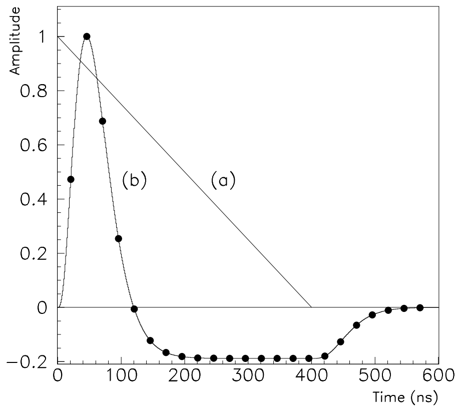

Abbildung 2.4: Shapingfunktion [2]

um über 1 GeV zu kommen. Bei beiden Abschätzungen erhält man ein Bin0/Rest-Ratio von ungefähr Neun.

2.1.2 Komponenten der Elektronik

Die Energieverteilung im Kalorimeter ist nun bekannt. Es ist also möglich eine Simulation zu schreiben, die einem Trigger Tower eine bestimmte Energie zuweist. Allerdings erhält man durch die Energiedeposition eines Teilchens in einem Trigger Tower keinen Energiewert, sondern ein analoges elektrisches Signal, das erst mit Hilfe entsprechender Elektronik in einen oder mehrere Energiewerte umgewandelt wird, die dann von der Kompressionseinheit verarbeitet werden. Es ist also notwendig auch die nachfolgende Elektronik zu modellieren. Der anfänglich für den Trigger Tower erhaltene Energiewert spielt die Rolle eines Normierungsfaktors, mit dem die Pulshöhe des analogen Signals festgelegt wird.

Pulsformung

In den einzelnen Trigger Towern wird eine Ionisationsladung gemessen, die proportional zur darin deponierten Energie ist. Ein typisches Signal zeigt Kurve (a) in Abbildung 2.4. Es ist ein Dreiecksimpuls

$$i_\delta(t) = I_0 \left(1 - \frac{t}{t_{dr}}\right) \quad t \leq t_{dr}, \tag{2.2}$$

dessen Abklingszeit t_{dr} der Driftzeit in flüssigem Argon (ca. 400 ns) entspricht. Aus diesem Puls wird durch den Pulsformer die in Abbildung 2.4 (b) gezeigte Shapingkurve erzeugt. Das Maximum des bipolaren Signals ist proportional zur deponierten Energie. Die Länge des Unterschwingers ist abhängig von der Driftzeit t_{dr}. Die schwarzen Punkte kennzeichnen die Bunch-Crossings. Dieses Signal hat gegenüber dem Dreieckspuls den Vorteil, daß es die Summe aus elektronischem und Pile-Up[3] Rauschen durch sein scharfes

[3] Überlagerung zweier um wenige Bunch-Crossings gegeneinander verschobener Signale.

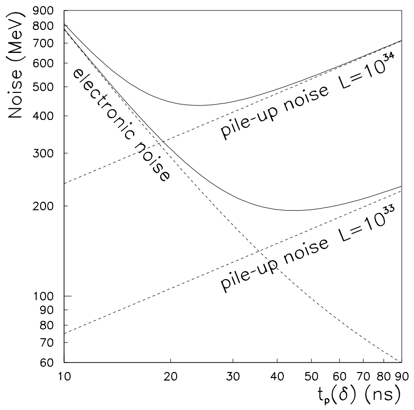

Abbildung 2.5: Rauschen [2]

Maximum verringert. Dadurch, daß das zeitliche Integral über diese Kurve verschwindet, wird auch eine Verschiebung der Nullinie verhindert.

Die Shapingfunktion läßt sich für verschiedene Eingangspulsformen geschlossen angeben [17]. Für die Simulation wurde als Eingangssignal der durch t_{dr} parametrisierte Dreieckspuls angenommen. Das Shaping wird dann beschrieben durch einen Satz von Funktionen, die in Anhang C aufgelistet sind. Parametrisiert wird die Kurve durch die Driftzeit t_{dr}, die Zeitkonstante des Shapers τ und das Verhältnis der Zeitkonstanten von Vorverstärker und Shaper λ.

Rauschen

Eng mit dem Shaping verbunden ist das Rauschen. In Abbildung 2.5 ist die Änderung der Summe aus elektronischem und Pile-Up Rauschen mit der Lage des Maximums (Anstiegszeit) der Shapingkurve zu sehen. Da sich mit wachsender Anstiegszeit die Bandbreite verringert fällt das elektronische Rauschen. Das Pile-Up Rauschen wächst hingegen mit der Anstiegszeit, da die Wahrscheinlichkeit zur Überlagerung zweier Pulse mit der Pulsbreite zunimmt. Legt man das Maximum der Shapingkurve auf circa 50 ns, so wird das Rauschen minimal und läßt sich durch eine Gaußkurve mit $\sigma \approx 0.5$ GeV beschreiben [2].

Im Simulationsprogramm wurde ein Zufallszahlengenerator eingebaut, der gaußverteilte Zufallszahlen mit beliebigem σ und Mittelwert produziert. Damit läßt sich die Breite des Rauschens variieren. Eine Variation des Mittelwertes ist zwar theoretisch möglich, jedoch nicht besonders sinnvoll.

FADC und Pedestal

Die Einstellung des Pedestals wird realisiert über die Addition eines einstellbaren konstanten Wertes zu jeder Energie. Am Schluß erfolgt die Digitalisierung. Hier läßt sich der Arbeitsbereich des FADC und die Auflösung in bits wählen. Die eigentliche Digitalisierung erfolgt durch Auf- beziehungsweise Abrunden der Energien. Liegt ein Wert außerhalb des Arbeitsbereiches, so wird diejenige Grenze des Arbeitsbereiches als Ausgabe gewählt, die überschritten wurde. Am Ende der Verarbeitungskette erhält man schließlich eine ganze Zahl, die eine bestimmte Energie repräsentiert.

2.2 Das Programm

Aufgabe von DATASIM ist es, ein einfaches Werkzeug für die Produktion von Testdaten zur Verfügung zu stellen. Die Variation möglichst vieler Parameter ist dabei besonders wichtig. Das Kalorimeter wird simuliert durch die in Abschnitt 2.1.1 motivierte Potenzfunktion, bei der sich der Exponent beliebig wählen läßt. Das Verhältnis von leeren zu gefüllten Trigger Towern wird eingestellt über das Bin0/Rest-Ratio. Auch für Rauschen und Shaping lassen sich alle wichtigen Parameter frei wählen. Es ist möglich die Anzahl der ausgelesenen Bunch-Crossings und die Anzahl der Bunch-Crossings vor dem Maximum der Shapingkurve zu bestimmen. Eine Phasenverschiebung zwischen Digitalisierung und Shapermaximum ist ebenfalls einstellbar. Für die FADCs ist schließlich die Auflösung in bits und der abgedeckte Energiebereich wählbar. Diese Einstellungen werden in einem Parameterfile vorgenommen, der von der Simulation eingelesen wird. Ein typisches Beispiel ist in Abbildung 2.6 zu sehen.

Die einzelnen Schritte der Berechnung eines Events für einen Kanal sind in Abbildung 2.7 dargestellt. Ein Zufallszahlengenerator bestimmt entsprechend dem Bin0/Rest-Ratio, ob ein Teilchen mehr als 1 GeV in einem Trigger Tower deponiert hat. Ist dies der Fall, wird eine Energie aus der Verteilung gewürfelt, ansonsten wird sie auf Null gesetzt. Nachdem die Shapingfunktion berechnet wurde, wird eine Schleife solange durchlaufen, bis die vom Benutzer eingestellte Anzahl von Werten auf der Kurve berechnet wurde. Zu jedem Wert werden Rauschen und Pedestal addiert und zum Schluß wird digitalisiert.

Es wird ein File angelegt, das in binärer Form die simulierten Daten enthält. Sie sind so angeordnet, daß jeweils die für einen Kanal zu einem Event ausgelesenen Bunch-Crossings in einem Block hintereinander stehen, und die im Parameterfile konfigurierten Kanäle nacheinander durchlaufen werden. Die Daten haben somit dasselbe Format, wie im RemASIC vor der Kompression (siehe 5.1). Außerdem ist eine Variation einzelner Parameter mit wählbarem Variationsbereich und beliebiger Schrittweite möglich. In diesem Fall kann zusätzlich zum Datenfile eine Tabelle erzeugt werden, die den aktuellen Parameterwert und die zugehörige Entropie enthält.

Nov 18 1997 18:05 **ParameterFile** Page 1

```
#Number of Events
1000
#Bin 0 / rest ratio
9
#Energy Cut
0
#Number of parameters for noise
4
#Parameters for noise
-100        # Minimum rnd number
100          # Maximum rnd number
0           # Mean
0.5         # Standard deviation
#Number of parameters for pow
4
#Four fitparameters follow
1           # Minimum rnd number
255         # Maximum rnd number
15.67       # Fit-parameter 1
1.833       # Fit-parameter 2
#Shaper Data
15          # Shaper time constant
400         # Drift time
2           # Lambda
#Sampling data
5           # Number of BunchCrossings per event
25          # Time between two samples
3           # Number of the sample in shaper maximum
0           # Phase-shift
# The following values are only of interest if you use an FADC object
# to digitize the obtained energies
#Number of bits
10
#Maximum energy
255
#Output Mode 0=dec 1=bin
0
#Number of FADCs
1
#Pedestal values for all the FADCs follow
1.5
```

Abbildung 2.6: Typischer Parameterfile für *DATASIM*

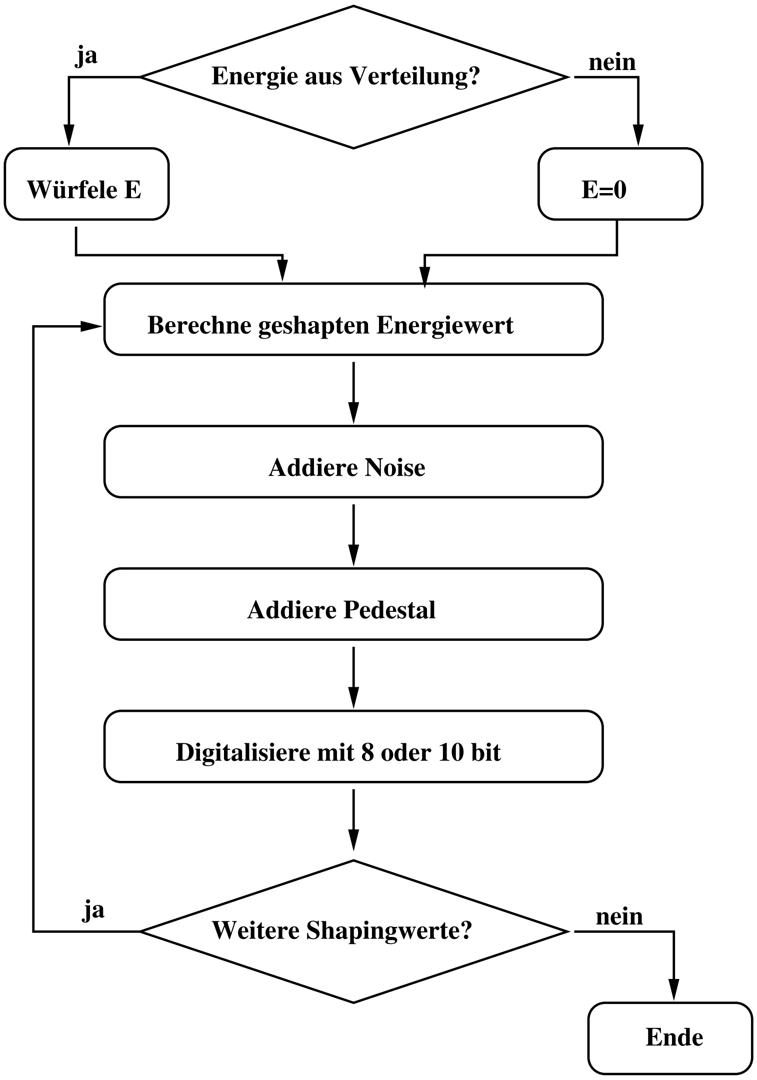

Abbildung 2.7: Flußdiagramm für die Berechnung eines Events in einer Zelle

Kapitel 3

Datenkompression

Für einen Test von verschiedenen Möglichkeiten zur Datenkompression wurde Software entwickelt, die Datensätze von DATASIM einlesen und mit verschiedenen Algorithmen komprimieren kann. Die getesteten Algorithmen wurden dann auf dem RemASIC implementiert. Dieses Kapitel widmet sich einer Darstellung der verwendeten Verfahren, die zum Teil allgemein bekannt sind (universelle Algorithmen) und zum Teil speziell für die Auslese des Level-1 Triggers entwickelt wurden (spezielle Algorithmen). Eine kurze Beschreibung der Kompressionssoftware bildet den Abschluß dieses Kapitels.

3.1 Eine kurze Einführung

Zwei Begriffe werden im Laufe der weiteren Diskussion immer wieder auftauchen, nämlich die Entropie H und die mittlere Codewortlänge $\bar{l}$ eines Datensatzes. Sie werden hier nur kurz eingeführt. Für eine Einführung in die Informations- und Codierungstheorie sei der Leser an Anhang A verwiesen.

Den Ausgangspunkt der Betrachtungen bildet eine Folge von Energiewerten, die zum Beispiel mit DATASIM erzeugt wurde, im weiteren als Datensatz bezeichnet. Ein Maß für die in diesem Datensatz enthaltene Information ist die Entropie

$$H(S) = -\frac{1}{\ln 2} \sum_{k=1}^{N} p_k \ln p_k; \qquad [H] = bit \tag{3.1}$$

N: Anzahl der verschiedenen Energiewerte
p_k : Wahrscheinlichkeit des k. Energiewertes.

In der Informationstheorie ist es üblich, die Entropie in der Einheit bit zu messen. Hat man zum Beispiel die 256 möglichen Energiewerte eines 8 bit FADC in einem Datensatz, so ist die darin enthaltene Information, und damit die Entropie, nur dann 8 bit, wenn alle Werte gleichwahrscheinlich sind. Ansonsten ist die Entropie kleiner.

Mit geeigneten Codierungsverfahren (siehe Anhang A) läßt sich dieser Umstand ausnutzen. Indem man häufigen Energiewerten kurze Codes, weniger häufigen dafür längere zuweist, wird der Umfang des Datensatzes reduziert. Da nun nicht mehr jeder Energiewert mit der gleichen Anzahl an bits (zum Beispiel 8 bit bei 256 Werten) codiert wird, kann man nicht mehr von einer Codewortlänge sprechen, sondern es muß die mittlere Codewortlänge verwendet werden. Die mittlere Codewortlänge $\bar{l}$ eines Datensatzes ist gegeben durch:

$$\bar{l} := \sum_{k=1}^{N} h_k l_k. \tag{3.2}$$

l_k: Länge des k. Codewortes
h_k: Relative Häufigkeit des k. Energiewertes im Datensatz

Ein Code ist umso besser, je näher die mittlere Codewortlänge an die Entropie herankommt. Es ist nicht möglich einen Datensatz mit einer mittleren Codewortlänge zu codieren, die kleiner ist als seine Entropie, ohne Information zu verlieren.

Zum Schluß seien noch kurz die verschiedenen prinzipiellen Möglichkeiten zur Datenkompression aufgeführt. Zunächst einmal lassen sich die Kompressionsalgorithmen einteilen in verlustfreie und verlustbehaftete. Verlustbehaftete Verfahren bieten sich vor allem bei der Übertragung von Audio- und Videodaten an. Für die Auslese der Triggerdaten kommen sie jedoch nicht in Frage, da man keine Information verlieren möchte. Deswegen sollen nur Möglichkeiten zur verlustfreien Kompression näher diskutiert werden.

Die Codierung der Ausgangsnachricht erfolgt im einfachsten Fall zeichenweise. Das bedeutet, jedem Zeichen der Ausgangsnachricht wird ein Codewort zugewiesen, das dann übertragen wird. Beispiele hierfür sind:

- Huffman (siehe 3.3.1)
- Huffman-Inspired (siehe 3.4.1)
- Shannon-Fano (nachzulesen in [18])

Eine andere Möglichkeit besteht darin, bestimmten Zeichenfolgen, die häufig auftreten, kurze Codes zuzuweisen. Bekannte Algorithmen sind:

- Run-Length (siehe 3.3.2)
- Lempel-Ziv (nachzulesen in [18])

Eine dritte Variante ist, die Nachricht an sich zu verändern. Dies setzt allerdings eine genauere Kenntnis der Datenstruktur voraus. Weiß man zum Beispiel, daß die Differenzen aufeinanderfolgender Zahlenwerte stets gering sind, die Zahlenwerte selbst sich jedoch über einen großen dynamischen Bereich erstrecken, so kann man statt der Zahlenwerte selbst die Differenzen codieren. Je mehr Wissen man über das Aussehen der Daten besitzt, desto effektiver lassen sich solche Methoden gestalten. Der Nachteil liegt allerdings in der Spezialisierung auf bestimmte Datenmuster. Das auf dem RemASIC realisierte *Difference Coding* (siehe 3.4.2) ist ein Beispiel für einen solchen Algorithmus.

3.2 Forderungen an die Verfahren

Nicht jedes beliebige Kompressionsverfahren kam für eine Implementation auf dem RemASIC in Frage. Da die Datenkompression lediglich ein Werkzeug für die Auslese des

Level-1 Triggers darstellt, mußten bestimmte Forderungen berücksichtigt werden. Das ideale Kompressionsverfahren müßte über folgende Eigenschaften verfügen

- Verlustfreie Kompression
- Erhalt von festen Wortgrenzen
- Rekonstruktion des Datensatzes ohne zusätzliche Informationen
- Robust gegenüber Schwankungen verschiedener Parameter
- Hoher Kompressionsfaktor

Die Forderung nach verlustfreier Kompression ist strikt einzuhalten. Kompressionsverfahren, bei denen ein Teil der Daten verloren geht, kamen von vornherein nicht in Frage. Allerdings wird diskutiert, einen *Energie-Cut* auf die Daten anzuwenden. Das bedeutet, daß alle Energien, die kleiner sind als ein bestimmter Wert, auf Null gesetzt werden. Daher wurde auch eine für diese Art von Daten optimale Kompressionsmöglichkeit untersucht.

Die vier folgenden Eigenschaften sind nicht strikt gefordert, es wäre jedoch wünschenswert, ein Verfahren zu finden, das möglichst viele dieser Eigenschaften besitzt. Gibt es in einem Code keine festen Wortgrenzen, also Blöcke mit wohl definierter Länge, so führt schon ein einziges bei der Übertragung des Codes verändertes bit dazu, daß der gesamte restliche Code unlesbar wird, oder falsch decodiert wird. Zusätzliche bits für eine Fehlerkorrektur würden hier Abhilfe schaffen, allerdings auch die Kompression verschlechtern, da die Codes länger werden. Im Rahmen dieser Arbeit wurde eine Fehlerkorrektur nicht implementiert. Um den Aufwand bei der Verwaltung der gespeicherten Daten gering zu halten, möchte man möglichst komprimierte Datensätze erhalten, die ohne zusätzliche Informationen oder Parameter decodierbar sind. Ein Algorithmus ist oft nur für Datensätze mit speziellen Eigenschaften geeignet (siehe Abschnitt 4.2). Wird er für die Bearbeitung von Datensätzen verwendet, die nicht über diese Eigenschaft verfügen, so kann er schlimmstenfalls zu einer Aufblähung des Codes führen. Da zum Beispiel die Pedestals von Kanal zu Kanal schwanken oder sich während des Betriebs verschieben können, ist es wichtig ein Verfahren zu finden, das möglichst unabhängig ist von solchen Einflüssen.

3.3 Universelle Algorithmen

Huffman und Run-Length Coding sind allgemein bekannte und häufig verwendete Verfahren zur Datenkompression. Während mit Hilfe des Huffman Coding beinahe jeder beliebige Datensatz (mehr oder minder gut) komprimiert werden kann, läßt sich das Run-Length Encoding nur auf solche Daten anwenden, bei denen dasselbe Zeichen möglichst oft hintereinander übertragen wird.

3.3.1 Huffman Coding

Es wird erwartet, daß häufig niedrige Energiewerte übertragen werden und die Wahrscheinlichkeit für ein Auftreten zu hohen Energien rasch abnimmt. Dies bedeutet jedoch (siehe

Formel 3.1), daß die Entropie deutlich unter ihrem Maximalwert (8/10bit bei 256/1024 Energiewerten) liegen wird.

Es ist natürlich wünschenswert, diesen Umstand auszunutzen, ohne allzu sehr von der genauen Form der zu verarbeitenden Daten abhängig zu sein. Dies ist auch der Grund, warum Huffman Coding auf dem RemASIC implementiert wurde. Solange die Energiewerte nicht gleichverteilt sind, sondern einige häufiger auftreten und andere dafür seltener, wird man eine Kompression erreichen.

Prinzip und Algorithmus

Beim Huffman Coding werden häufig auftretenden Zeichen kurze Codes zugewiesen und seltenen dafür längere Codes (siehe Anhang A.2.2). Das Problem besteht darin, zum einen einen eindeutig decodierbaren Code zu erzeugen, zum anderen mit der mittleren Codewortlänge $\bar{l}(C, S)$ möglichst nah an die Entropie $H(S)$ zu kommen. Solch einen Code bezeichnet man nach A.2.2 als optimalen Prefix Code. Sie lassen sich sehr gut durch Baumstrukturen darstellen. Es wird also im folgenden die Konstruktion eines binären Baumes, der einen optimalen Prefix Code repräsentiert, vorgestellt.

Das Vorgehen ist in Abbildung 3.1 dargestellt. Zunächst werden die Zeichen nach fallender Wahrscheinlichkeit in einer Liste eingetragen. Die letzten beiden (unwahrscheinlichsten) Zeichen werden als Blätter an einen Knoten gehängt. Statt den beiden Blättern wird nun der Knoten in die Liste eingetragen und erhält die Summe der Wahrscheinlichkeiten der beiden Blätter. Die Liste wird neu sortiert, und das ganze solange wiederholt, bis sie nur noch ein einziges Element enthält, die Wurzel des Codebaumes. Mit dem dargestellten Algorithmus läßt sich für ein gegebenes Ensemble der Huffman Code erzeugen.

Um mit dem erzeugten Code einen Datensatz zu codieren, wird eine Art Codebuch verwendet. Die Zeichen des Ensembles werden durchnummeriert von 1 bis N. Diese Zahlenwerte dienen als Index für eine Tabelle, die in irgendeiner Form sowohl den eigentlichen Code als auch die Codewortlänge enthalten muß, da die Codes verschieden lang sind. Im Falle der Software ist die Tabelle einfach ein zweidimensionales Array. Mit dem ersten Index wird das Zeichen selektiert. Der zweite Index kann nur die Werte Null oder Eins annehmen. Eine Variable enthält den Code als Integer Variable, die andere die Codewortlänge. Auf dem RemASIC wurde die Tabelle als Look-Up Table realisiert (siehe 5.4.1).

Das Decodieren eines Datensatzes, der bereits Huffman codiert ist, kann zum Beispiel über den Codebaum erfolgen. Man wandert, beginnend an der Wurzel, solange entsprechend den eingelesenen bits den Baum entlang, bis man auf ein Blatt stößt. Das Zeichen, das dieses Blatt repräsentiert, wird dann geschrieben. Auf diese Weise erkennt man automatisch die Wortgrenzen.

Beispiel

Gegeben sei ein Datensatz mit vier Energiewerten zwischen 1 GeV und 4 GeV. Die Wahrscheinlichkeiten für das Auftreten der Energien und die Codes, die sich mit dem beschriebenen Verfahren ergeben sind in Tabelle 3.1 zu sehen. Der Codebaum für dieses Beispiel

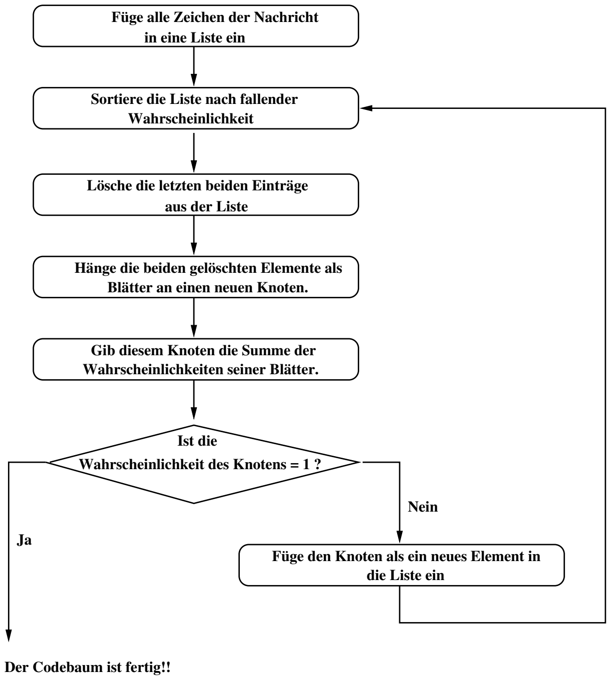

Abbildung 3.1: Der Huffman Algorithmus

Energie (GeV)	Code	Wskt. $p(E)$
1	0	0.55
2	10	0.20
3	111	0.15
4	110	0.10

Tabelle 3.1: Huffman Codes für einen Beispieldatensatz

ist in Anhang A abgebildet. Die Entropie dieses Datensatzes beträgt

$$H(S) = -\frac{1}{\ln 2}(0.55 \ln 0.55 + 0.20 \ln 0.20 + 0.15 \ln 0.15 + 0.10 \ln 0.10) \text{ bit} = 1.68 \text{ bit}. \quad (3.3)$$

Die mittlere Codewortlänge beträgt

$$\bar{l} = (1 * 0.55 + 2 * 0.20 + 3 * 0.15 + 3 * 0.10) \text{ bit} = 1.7 \text{ bit}. \quad (3.4)$$

Bei der Codierung mit normalen Binärzahlen hätte man 2 bit benötigt. An diesem Beispiel läßt sich auch eine Gefahr des Huffman Coding veranschaulichen, die Aufblähung des Codes. Wird mit obigem Code ein Datensatz codiert, der nur aus den unwahrscheinlichen Energien 3 GeV und 4 GeV besteht, so wächst die mittlere Codewortlänge $\bar{l}$ auf 3 bit.

Vor- und Nachteile

Mit dem Huffman-Verfahren hat man die Möglichkeit, für ein unveränderliches Ensemble einen optimalen Prefix Code zu finden. Der Algorithmus zur Erstellung des Codebaumes läßt sich relativ einfach auf einem Rechner implementieren. Das Codieren selbst erfordert keine zeitaufwendigen Berechnungen, sondern läßt sich durch Abfrage der Werte einer Tabelle realisieren.

Trotz seiner großen Beliebtheit und seiner vielen Vorzüge hat diese Methode auch einige Nachteile. Sollen die codierten Daten übertragen werden, so ist man auf die absolute Zuverlässigkeit des Übertragungskanals angewiesen, oder muß zusätzliche bits für eine Fehlerkorrektur senden, da es keine festen Wortgrenzen gibt. Verändern sich die Wahrscheinlichkeiten für das Auftreten von Zeichen mit der Zeit, so wird sich die Kompression verschlechtern, da der erzeugte Code nur für ein ganz bestimmtes Ensemble optimal ist. Die Speicherung des Codebaumes ist zwingend erforderlich, da nur so die Daten wieder rekonstruiert werden können.

3.3.2 Run-Length Encoding

Run-Length Encoding wurde implementiert, um Daten nach BCID, Pedestalsubtraktion und eventuell einem Energie-Cut effizient zu komprimieren. Subtrahiert man nämlich die Pedestal und wendet einen moderaten Energie-Cut (ca. 1 GeV) an, so verschwindet der Rauschuntergrund der Trigger Tower, in denen keine Energie deponiert ist. Für diese wird dann immer der Wert Null übertragen. Dies bedeutet, daß es lange Folgen von Nullen geben wird.

Prinzip

Es ist ineffizient, dasselbe Zeichen oft hintereinander zu übertragen. Das Run-Length Encoding bietet eine einfache Möglichkeit, das Datenvolumen zu reduzieren. Anstatt ein Zeichen n-mal hintereinander zu schreiben, wird das Zeichen und danach die Anzahl der Wiederholungen geschrieben. Die einzige Schwierigkeit ist, beim Decodieren zu erkennen, ob ein Code nun ein Zeichen oder die Zahl der Wiederholungen repräsentiert. Es gibt hier eine große Vielfalt von Möglichkeiten, hier soll jedoch nur die Variante besprochen werden, die auf dem RemASIC realisiert ist.

Da der einzige Energiewert, der oft hintereinander auftreten wird, Null ist, codiert man lediglich Folgen von Nullen und überträgt alle anderen Energien unverändert, das heißt als 8 oder 10bit Zahl. Der Wert Null wird als Signal dafür benutzt, daß als nächstes die Zahl der Wiederholungen eingelesen werden muß. Auch bei einem einmaligen Auftreten einer Null wird zusätzlich die Anzahl der Wiederholungen übertragen, das Datenvolumen wird größer.

Die Lauflänge wird auch als 8 oder 10bit Zahl übertragen, so daß es bei diesem Verfahren definierte Wortgrenzen gibt.

Nachricht	Code
27-12-31	27-12-31
255-0-13	255-0-1-13
255-0-0-0-0-13	255-0-4-13
12-..(300 mal 0)..-18	12-0-255-0-45-18

Tabelle 3.2: Run-Length Encoding für Beispieldaten

Beispiel

Als Beispiel mögen einige ausgewählte Folgen von 8 bit Energiewerten dienen. In Tabelle 3.2 ist ihre Codierung, wie sie auf dem RemASIC stattfindet, angegeben. Der erste Datensatz bleibt unverändert, da er keine Null enthält, der zweite wird aufgebläht, denn es taucht nur eine einzige Null gefolgt von einem anderen Wert auf. Im dritten findet eine Kompression statt, aus vier Nullen wird eine Null und eine Vier. Auch der letzte Datensatz wird komprimiert. Hier ist gezeigt, was passiert, wenn die Zahl der Nullen zu groß wird, um mit einer 8 bit Zahl codiert zu werden. Es wird einfach bis 255 gezählt und dann erneut eine Null übertragen und die Zählung beginnt von vorne.

Vor- und Nachteile

Da es beim Run-Length Encoding feste Wortgrenzen gibt, ist die Übertragung des Codes relativ fehlertolerant. Sollte ein 1bit-Fehler auftreten, ist zwar ein Energiewert falsch, oder es werden zu viele Nullen rekonstruiert, aber der restliche Datensatz bleibt davon unberührt.

Dieses Verfahren ist nur sinnvoll, falls bereits eine Pedestalsubtraktion stattgefunden hat, und ein Energie-Cut angewendet wird. Andernfalls ist es sehr wahrscheinlich, daß in den Datensätzen einzelne Nullen erscheinen. Dies führt bei dem auf dem RemASIC implementierten Verfahren dann zu einer Vergrößerung des Datenvolumens.

3.4 Spezielle Algorithmen

Zwei weitere Kompressionsverfahren wurden auf dem RemASIC implementiert. Sie sind speziell für die erwartete Struktur der Kalorimeterdaten entwickelt worden. Beide Verfahren machen sich den Umstand zu nutze, daß die Daten sehr häufig aus einem engen Bereich um einen bestimmten Wert stammen werden.

3.4.1 Huffman-Inspired Coding

Nimmt man an, daß das Pedestal P (siehe Abschnitt 1.2.1 und 2.1.2) für alle Kanäle gleich ist, so hat die Wahrscheinlichkeitsverteilung für die Energiewerte ein scharfes Maximum bei P. Dieses Maximum wird umso mehr verschmiert, je mehr die Pedestals der einzelnen Kanäle schwanken. Sind diese Schwankungen nicht allzu groß, so liegen die Energiewerte, die bei weitem am häufigsten auftreten, in einem Intervall um eine *Referenzenergie* E_R.

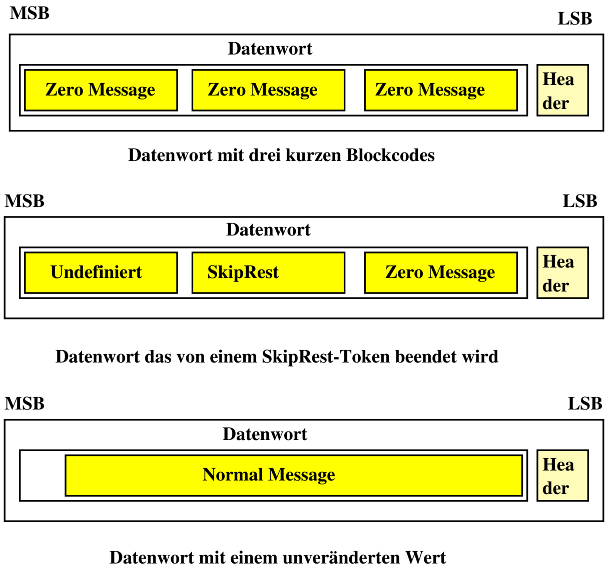

Abbildung 3.2: Datenwörter beim Huffman-Inspired Coding

Prinzip

Codiert man nun die Energien, die in diesem Intervall um E_R liegen mit kurzen Blockcodes (siehe Anhang A) und überträgt alle anderen Energien unverändert, so wird sich eine Reduktion des Datenvolumens ergeben, ohne daß auf Wortgrenzen verzichtet werden muß [19]. Allerdings muß nun noch eine Möglichkeit geschaffen werden, zwischen kurzen und langen Blockcodes zu unterscheiden.

Es werden immer Datenwörter gleicher Länge l_{DW} übertragen. Der Index steht für *Data Word*. Jedes Datenwort beginnt mit einem Header. Er ist ein bit lang und zeigt an, wie der Rest des Datenwortes zu interpretieren ist. Die verschiedenen Möglichkeiten zum Aufbau eines Datenwortes zeigt Abbildung 3.2.

Die kurzen Blockcodes werden im weiteren als Zero-Message bezeichnet, ihre Länge sei l_{ZM}. Die unveränderten Energien haben die Länge l_{NM}, dabei bedeutet NM *Normal Message*. Es lassen sich also maximal

$$\Delta = \pm\frac{1}{2}(2^{l_{ZM}} - 2) = \pm 2^{l_{ZM}-1} - 1 \tag{3.5}$$

Energiewerte um E_R durch kurze Codes ersetzen. Mit l_{ZM} bits können nämlich maximal $2^{l_{ZM}}$ Nachrichten codiert werden. Einen Code braucht man für die Referenzenergie selbst und einer wird als *SkipRest*-Token reserviert. Das *SkipRest*-Token wird benötigt, weil es möglich ist, daß weniger kurze Codes hintereinander auftreten, als in ein Datenwort passen. Tritt also das *SkipRest*-Token auf, so wird der Rest des Datenwortes ignoriert und der Decoder liest das nächste Datenwort.

Bisher wurde die Frage offengelassen, wie die kurzen Blockcodes zustande kommen. Dazu wird die Differenz zwischen dem aktuellen Energiewert und E_R berechnet. Liegt diese innerhalb von Δ, so läßt sie sich durch eine l_{ZM}bit-Zahl ausdrücken. Das MSB wird

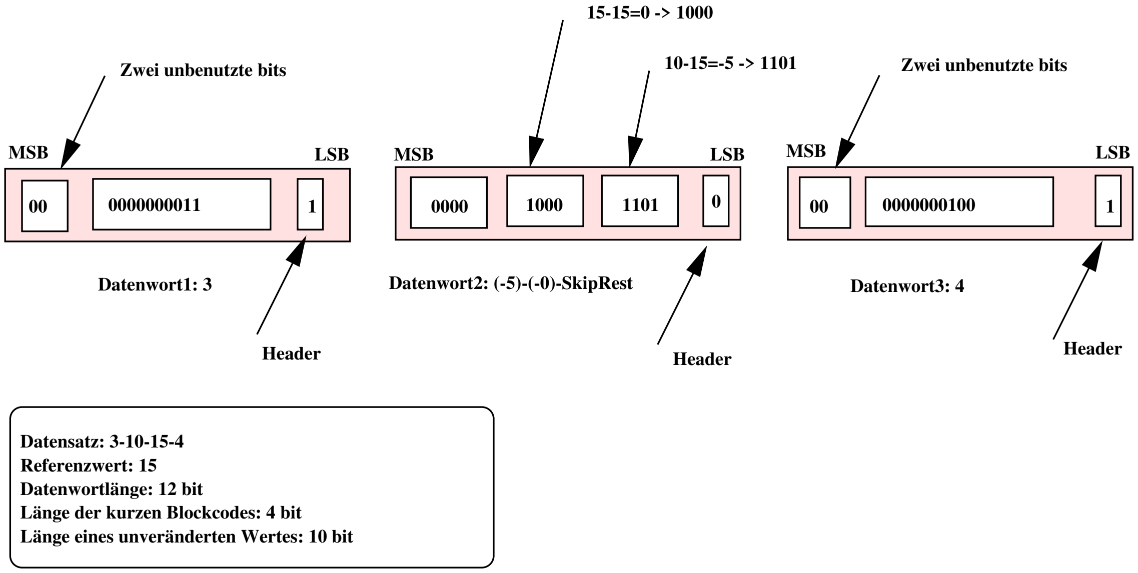

Abbildung 3.3: Huffman-Inspired Coding für einen Beispieldatensatz

als Vorzeichen-bit genutzt, der Rest stellt, in binärer Form, den Wert dar, um den E_R zu vergrößern, oder zu verkleinern ist. Der Wert Null mit Vorzeichenbit auf Null, also b0000[1], wird als SkipRest-Token verwendet, für die Referenzenergie wird b1000 reserviert.

Auf dem RemASIC wurde das Huffman-Inspired Verfahren mit festen Werten für l_{DW}, l_{NM} und l_{ZM} realisiert. Der Wert für E_R läßt sich über ein Register wählen (siehe Kapitel 5).

Beispiel

In Abbildung 3.3 ist die Codierung nach dem Huffman-Inspired Verfahren für eine kurze Folge von Energiewerten zu sehen. Die Energieauflösung betrage 10 bit, der Wertebereich 0 GeV bis 255 GeV. Die verwendeten Parameter sind dem Kasten darunter zu entnehmen. Die Wortlängen entsprechen den auf dem RemASIC realisierten Werten. Man erkennt, daß bei 3 LSBs, was bei 10 bit Auflösung 0.75 GeV entspricht, die Energie unverändert in das Datenwort eingesetzt wird. Die Werte 2.5 GeV (10 LSBs) und 3.75 GeV (15 LSBs) liegen im Bereich Δ um $E_R = 3.75$ GeV (15 LSBs) und werden mit kurzen Codes codiert. Dann folgt ein Wert, der nicht mehr in Δ liegt, das angefangene Datenwort wird mit dem SkipRest-Token (b0000) beendet und die Energie unverändert in ein neues Datenwort eingefügt.

Vor- und Nachteile

Beim Huffman-Inspired Coding gibt es, im Gegensatz zum Huffman Coding, feste Wortgrenzen. Damit ist es fehlertoleranter. Weiterhin ist es nicht erforderlich einen Codebaum

[1] Das führende b zeigt an, daß die nachfolgende Zahl als Binärzahl dargestellt ist.

zur Rekonstruktion der Daten zu speichern.

Es ist jedoch immer noch ein Parameter, der Referenzwert E_R nötig, um die Daten vollständig zu decodieren. Ist er verlorengegangen, ist eine Rekonstruktion zwar möglich, jedoch nur bis auf eine additive Konstante. Ein weiterer Nachteil ist die Anfälligkeit gegenüber Pedestalschwankungen (siehe 4.2). Für die 16 Kanäle, die von einem RemASIC ausgelesen werden können, läßt sich ein gemeinsamer Referenzwert E_R einstellen. Sind die Pedestals der Kanäle weit voneinander entfernt, oder verändern sie sich während des Betriebs, so wird das Huffman-Inspired Coding keine guten Ergebnisse mehr liefern. Theoretisch wäre es auch möglich gewesen, E_R für jeden Kanal einzeln einzustellen, dies würde jedoch der Forderung nach einer möglichst einfachen Rekonstruktion der Ausgangsdaten widersprechen.

3.4.2 Difference Coding

Das Difference Coding ist eine Weiterentwicklung des Huffman-Inspired Coding. Es werden nach denselben Prinzipien Datenwörter aus einem Header und mehreren kurzen Blockcodes oder einem unveränderten Energiewert gebildet. Bei ersterem Verfahren mußte jedoch ein Referenzenergiewert E_R angegeben werden, um den herum die Energiewerte mit kurzen Codes liegen.

Prinzip

Die Nachteile des Huffman-Inspired Verfahrens werden beim Difference Coding beseitigt, indem nicht die Differenz des aktuellen Energiewertes zu einem Referenzwert E_R gebildet wird. Statt dessen zieht man immer zwei aufeinanderfolgende Energiewerte voneinander ab, und überträgt diese Differenz, wenn sie im Intervall Δ liegt. Ansonsten wird die aktuelle Energie übertragen.

Beispiel

Es wird dasselbe Beispiel verwendet, wie für das Huffman-Inspired Coding. In Abbildung 3.4 ist das Ergebnis zu sehen. Während das Huffman-Inspired Verfahren jedoch nur in einem bestimmten Bereich Δ um die Referenzenergie E_R gute Ergebnisse liefert, funktioniert das Difference Coding in jedem Energiebereich, vorausgesetzt, daß die aufeinanderfolgenden Energiewerte nur nahe genug beieinander liegen.

Vor- und Nachteile

Das Difference Coding ist das beste der im Rahmen dieser Arbeit untersuchten Verfahren, wenn es darum geht Kalorimeterdaten ohne Pedestalsubtraktion und Energie-Cut zu komprimieren. Es gibt feste Wortgrenzen, es ist unempfindlich gegenüber Pedestalschwankungen und liefert gute Kompressionsraten. Die Rekonstruktion der Ausgangsdaten ist vollkommen ohne zusätzliche Informationen möglich. Dadurch, daß nicht ausschließlich Differenzen übertragen werden, sondern nur wenn dies einen Platzvorteil bei der Codierung bedeutet, muß nicht eine Verschiebung aller nachfolgenden Energiewerte befürchtet

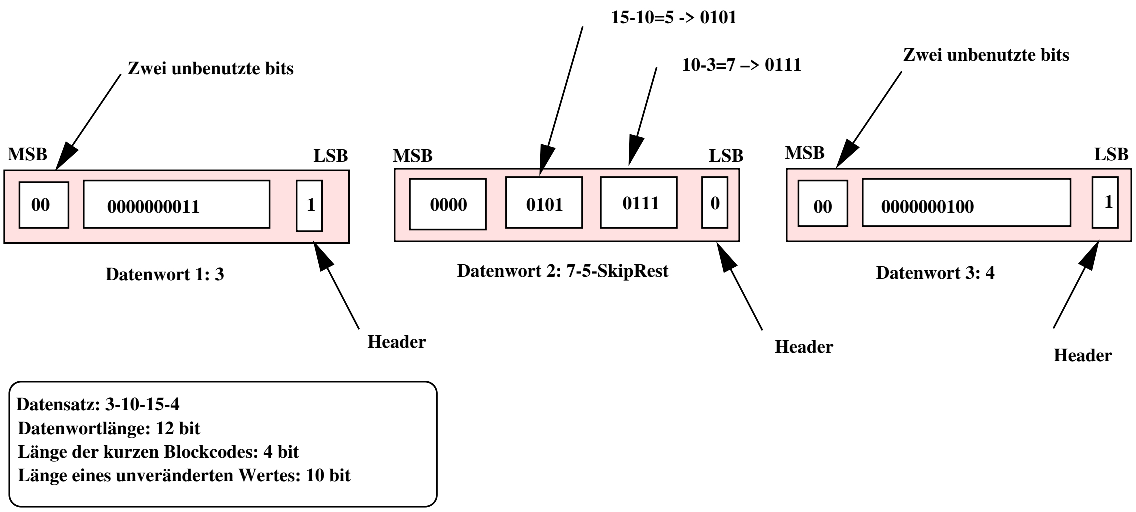

Abbildung 3.4: Difference Coding für einen Beispieldatensatz

werden, falls ein Übertragungsfehler auftritt. Lediglich die Folge der differenzcodierten Werte, in der der Übertragungsfehler aufgetreten ist, ist davon betroffen.

3.5 Software zur Datenkompression

Auch die Kompressionssoftware *XCompress* wurde in C++ entwickelt. Im Gegensatz zu DATASIM wurde sie noch mit einer graphischen Benutzeroberfläche versehen. Hierzu wurde die Qt Klassenbibliothek von Troll Tech verwendet, die für die Entwicklung nicht kommerzieller Software frei erhältlich ist [20]. Bei den Überlegungen zum Design der Software standen folgende Punkte im Vordergrund:

- Bereitstellung der Routinen zur Kompression und Dekompression
- Einstellbarkeit von möglichst vielen Parametern für die Simulation
- Lesen und Schreiben von Daten in möglichst vielen Formaten
- Einfache Bedienung und Fehlertoleranz

Die Kompressionssoftware wurde nicht nur für einen Test der verschiedenen Algorithmen konzipiert, sondern sie soll auch für die Dekompression von Datensätzen verwendet werden, die mit Hilfe des RemASIC erzeugt wurden.

Der Aufbau der Software ist in Abbildung 3.5 zu sehen. Die Benutzeroberfläche (Abbildung 3.6) dient zum Einstellen der Parameter für die Kompressionsalgorithmen. In der Zwischenschicht erfolgt der Aufruf der gewünschten Routinen mit den entsprechenden Parametern. Sie benutzt zwar den Signal/Slot Mechanismus von Qt[2] (siehe [20]), ist aber

[2] Verfahren zur Reaktion auf Ereignisse in der Benutzeroberfläche

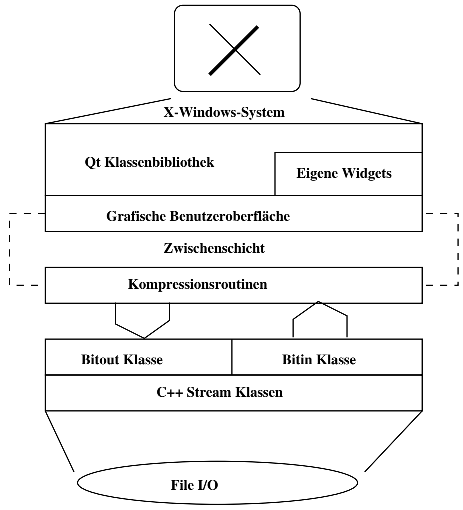

Abbildung 3.5: Aufbau der Kompressionssoftware

ansonsten so ausgelegt, daß sie mit leichten Modifikationen auch ohne Qt mit einer textorientierten Schnittstelle eingesetzt werden kann. Die Kompressionsroutinen benutzen ihrerseits Objekte der Klassen *Bitin* und *Bitout* als Schnittstelle zu den Standard Ein- und Ausgabeklassen von C++.

Ohne auf weitere Implementationsdetails eingehen zu wollen, soll im folgenden eine Übersicht über die erreichte Funktionalität gegeben werden

3.5.1 Ein- und Ausgabe

Um Datenkompression betreiben zu können, wird eine Möglichkeit zum Schreiben und Lesen einzelner bits oder Bitfolgen benötigt. Es wurde eine Schnittstelle für die Benutzung solcher *Bitstreams* zur Verfügung gestellt, die sich in der Handhabung an die Standard C++-Streams anlehnt [21]. Der Benutzer kann zwischen verschiedenen Formaten (binäre Darstellung, Dezimalzahlen oder Hexadezimalzahlen variabler Länge) auswählen. Im Binärformat wird ein File direkt als Folge von Nullen und Einsen interpretiert. In diesem Modus können zum Beispiel Textfiles bearbeitet werden. Im Dezimalzahlen-Modus werden Files verarbeitet, die aus einer Folge von Dezimalzahlen bestehen. Die einzelnen Zahlen werden durch Leerzeichen voneinander getrennt. Intern werden diese Zahlen dann in Binärzahlen umgewandelt, deren Länge der Benutzer angeben kann. Hiermit lassen sich zum Beispiel Files verarbeiten, die eine Folge von Energiewerten in Zahlendarstellung beinhalten. Der Hexadezimalzahlen-Modus ist äquivalent zum Dezimalzahlen-Modus, mit dem Unterschied, daß die Zahlen in hexadezimaler Darstellung erwartet werden.

3.5.2 Erzeugung von Codebäumen

Codebäume lassen sich auf verschiedene Arten erzeugen, zum einen rekursiv von der Wurzel bis zu den Blättern (Top Down) oder, angefangen mit einem Blatt, durch immer neues

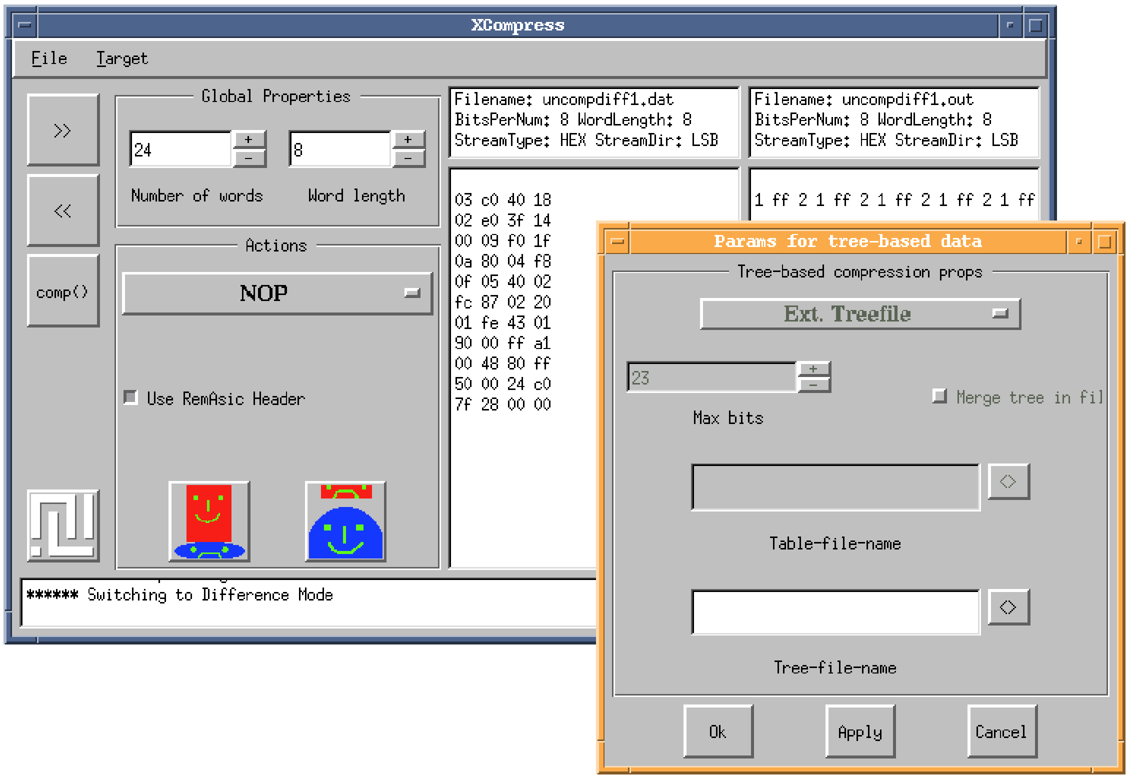

Abbildung 3.6: Benutzeroberfläche der Kompressionssoftware

Zusammenfügen von Blättern zu Knoten bis hin zur Wurzel (Bottom Up). Das Top Down Verfahren wird zur Rekonstruktion des Baumes im Speicher aus einem File verwendet, während das Bottom Up Verfahren zur Konstruktion eines neuen Codebaumes aus einer Tabelle benötigt wird. Diese Tabelle kann entweder als File zur Verfügung stehen und eingelesen werden, oder sie wird durch Zählen der einzelnen Zeichen im zu komprimierenden File erstellt.

Es stehen somit drei Möglichkeiten zur Verfügung einen Codebaum zu erzeugen. Er kann direkt aus der Häufigkeitsverteilung der Zeichen im zu komprimierenden File erzeugt werden. Der Codebaum ist dann optimal an den zu komprimierenden Datensatz angepaßt. Eine andere Möglichkeit besteht darin, den Baum aus einer vorhandenen Tabelle mit den Häufigkeiten der einzelnen Zeichen zu erstellen, oder aber einen bereits erzeugten und gespeicherten Baum direkt in den Speicher einzulesen.

3.5.3 Abschneiden von Codebäumen

Beim Huffman Verfahren läßt sich im Gegensatz zu den üblichen Kompressionsprogrammen die maximale Anzahl der bits für ein Codewort einstellen. Hat man einen bestimmten Datensatz gegeben, so erzeugt der Huffman Algorithmus einen Codebaum, bei dem das unwahrscheinlichste Zeichen unter Umständen mit sehr vielen bits codiert wird. Es ist nun wünschenswert (siehe 4.2) die maximale Länge der Huffman Codes zu limitieren. Da der Huffman Code ein optimaler Code ist in dem Sinne, daß die mittlere Codewortlänge minimal ist (siehe A.2.2), wird sofort klar, daß diese Limitierung mit einer wachsenden mittleren Codewortlänge erkauft wird.

Realisiert wurde das Abschneiden über eine Änderung der Wahrscheinlichkeiten in der Zeichenliste (Abschnitt 3.3.1). Zunächst wird der Baum mit der unveränderten Liste gebildet. Ist er zu lang, so werden die Wahrscheinlichkeiten der hinteren, unwahrscheinlicheren Hälfte alle gleichgesetzt. Dadurch wird erreicht, daß in diesem Teil die Codewortlänge minimal wird (siehe Anhang A). Nun wird erneut der Codebaum gebildet und seine Länge ermittelt. Ist der Baum immer noch zu lang, so werden die Wahrscheinlichkeiten der unteren Hälfte des unveränderten Teils der Liste gleichgesetzt. Ist er zu kurz, so wird nur noch das untere viertel der gesamten Liste auf dieselbe Wahrscheinlichkeit gesetzt. Es wird also über einen Intervallteilungsalgorithmus die Stelle in der Liste gesucht, ab der die restlichen Wahrscheinlichkeiten gleichgesetzt werden müssen, um die gewünschte Länge zu erreichen.

Das Ergebnis, das man mit dieser Methode erzielt ist nicht unbedingt die Variante, die die minimale mittlere Codewortlänge bei gegebener maximaler Baumgröße liefert. Da ein Abschneiden des Codebaumes bei der Auslese des Level-1 Triggers mit dem RemASIC selten notwendig sein wird (siehe Kapitel 4.2.1), ist es auch nicht nötig, diesen Algorithmus entsprechend zu optimieren.

Kapitel 4

Ergebnisse der Simulationen

In diesem Kapitel werden die Ergebnisse der Simulationen vorgestellt und diskutiert, die mit DATASIM und XCompress erhalten wurden. Es werden zwei verschiedene Typen von Simulationen behandelt. Zum einen wird die Entropie der simulierten Datensätze in Abhängigkeit von verschiedenen Parametern untersucht, zum anderen die Güte der verschiedenen Kompressionsalgorithmen für typische Energieverteilungen. Als Ausgangsbasis für die Parametervariation werden die in Tabelle 4.1 zusammengestellten Grundeinstellungen verwendet. Es werden also pro Event fünf Bunch Crossings ausgelesen, von denen das dritte genau im Pulsmaximum liegt. Die Form der Shapingkurve wird so gewählt (siehe 2.1.2), daß das Rauschen minimal wird. In den folgenden Diskussionen werden nur Parameter angegeben, die von diesen Standardwerten abweichen.

4.1 Entropie der Daten

Eine erste Untersuchung widmete sich der Entropie der Kalorimeterdaten. Hierzu wurden mit der Datensatzsimulation *DATASIM* Datensätze mit bis zu 10^5 Events erzeugt und verschiedene Parameter variiert. Anschließend wurden mit der Kompressionssoftware die Häufigkeiten der Energiewerte gezählt und daraus die Entropie in diesem Datensatz bestimmt.

4.1.1 Die Energieverteilung

In Anhang A wird gezeigt, daß die Entropie eines Datensatzes bestimmt wird durch die Wahrscheinlichkeitsverteilung der Zeichen. Es ist daher nicht verwunderlich, daß die Entropie von der Wahl des Exponenten der Verteilung und des Bin0/Rest-Ratio abhängig ist. Je größer die Exponenten werden, desto steiler wird die Energieverteilung. Sehr wenige kleine Energien werden also immer wahrscheinlicher, die Entropie sinkt. Abbildung 4.1 zeigt den Verlauf der Entropie von $H = 6.6$ bit für einen Exponenten von 1.25 bis zu $H = 3.6$ bit bei einem Exponenten von 5 für den Fall, daß nur Energien größer 1 GeV auftreten (Bin0/Rest-Ratio = 0). Variiert man den Exponenten dagegen bei einem Bin0/Rest-Ratio von 9, so bleibt die Entropie nahezu konstant. Die Zahl der leeren (E = 0 GeV) Zellen nimmt mit steigendem Bin0/Rest-Ratio zu. Die Energieverteilung und

Simulation von	mit	Parameter
Energiebereich		0 GeV-255 GeV
Energiedeposition	Potenzfunktion Bin0/Rest-Ratio	Exponent = 1.8 Bin0/Rest = 9
Rauschen	Gaußverteilung	$< x >= 0$ GeV $\sigma = 0.5$ GeV
Shaping	$f(\tau, t_{dr}, \lambda)$ siehe Anhang C	$\tau = 15$ $t_{dr} = 400$ ns $\lambda = 2$
Sampling	Bunch Crossings pro Event Sample im Pulsmaximum	5 3. Sample
Pedestal	$E = E + Ped$	$Ped = 1.5$ GeV

Tabelle 4.1: Grundeinstellungen für die Simulation

damit auch die Entropie wird also mehr und mehr unabhängig von der Potenzfunktion und ihrem Exponenten.

Die Abbildungen 4.2 und 4.3 zeigen den Verlauf der Entropie mit steigendem Bin0/Rest-Ratio für 8 und 10 bit. Im Grenzfall Bin0/Rest-Ratio $\rightarrow \infty$ gibt es nur noch einen einzigen Energiewert, E = 0 GeV. Das heißt die Entropie wird nur noch bestimmt durch das Detektorrauschen.

4.1.2 Entropie und Pedestal

Durch eine Variation des Pedestals läßt sich die Energieverteilung im Fenster[1] des FADC hin und herschieben. Wird das Pedestal erhöht, so wird der relativ unwahrscheinliche Schwanz abgeschnitten (siehe Abbildung 4.5), wird es erniedrigt, so schneidet man die Kurve nahe am Maximum ab (Abbildung 4.4). Man erwartet also, daß sich die Entropie mit wachsendem Pedestal solange erhöht, bis das gesamte Maximum im Arbeitsbereich des FADC liegt. Sie sollte dann für längere Zeit einen konstanten Wert einnehmen, bis zuviel vom Schwanz abgeschnitten wurde und die Entropie wieder sinkt. Der Anstieg der Entropie mit wachsendem Pedestal ist in den Abbildungen 4.6 und 4.7 für 8 und 10 bit Werte zu sehen. Das Pedestal wurde nicht genügend erhöht, um auch den Abfall bei höheren Werten zu beobachten. Es sei an dieser Stelle explizit darauf hingewiesen, daß der Anstieg der Entropie bis zu einem Pedestal von 1.5 GeV daher rührt, daß bei einem niedrigeren Pedestal ein Teil des Rauschens abgeschnitten wird. Man führt faktisch einen Energie-Cut durch, der allerdings nur das Rauschen abschneidet, da bei $\sigma = 0$ GeV, also ohne Rauschen, keine Energien auftauchen, die kleiner sind als das Pedestal. In Abbildung 4.6 ist eine Schwankung der Entropie mit Maximum bei halbzahligen und Minimum bei ganzzahligen Pedestalwerten zu beobachten. Ihre Ursache wird im nächsten Abschnitt erklärt. 4.7).

[1]Bereich in dem der FADC arbeitet (0-255 GeV).

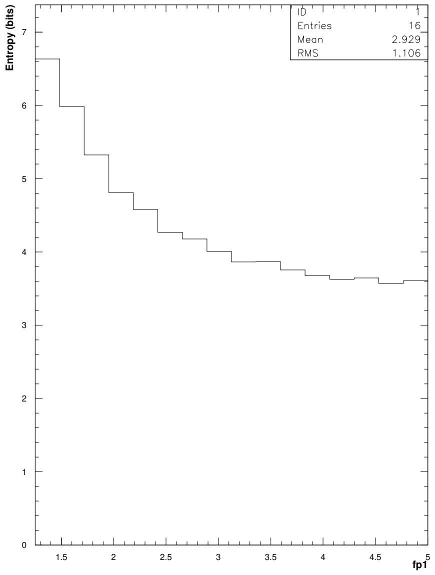

Abbildung 4.1: Entropie vs Exponent für 10 bit und Bin0/Rest-Ratio = 0

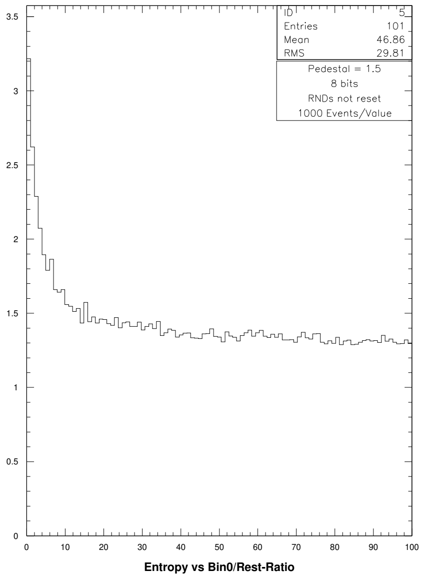

Abbildung 4.2: Entropie vs Bin0/Rest-Ratio bei 8bit

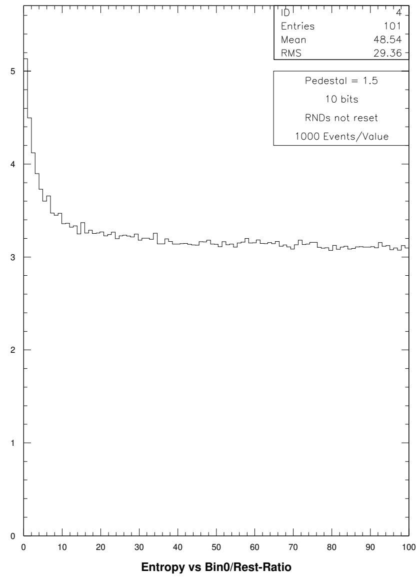

Abbildung 4.3: Entropie vs Bin0/Rest-Ratio bei 10bit

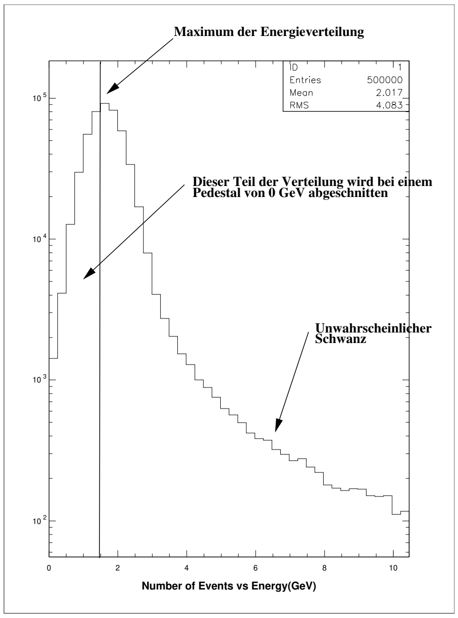

Abbildung 4.4: Energieverteilung bei einem Pedestal von 1.5 GeV

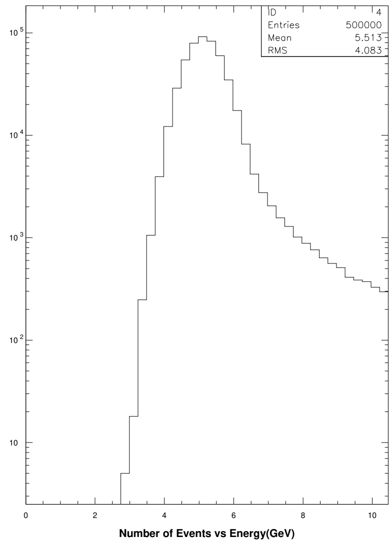

Abbildung 4.5: Energieverteilung bei einem Pedestal von 5.0 GeV

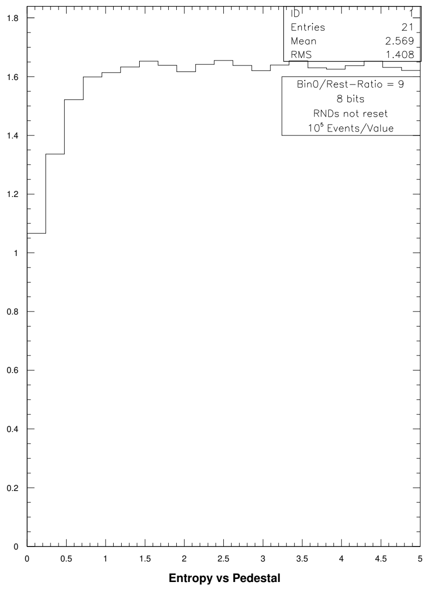

Abbildung 4.6: Entropie vs Pedestal bei 8bit

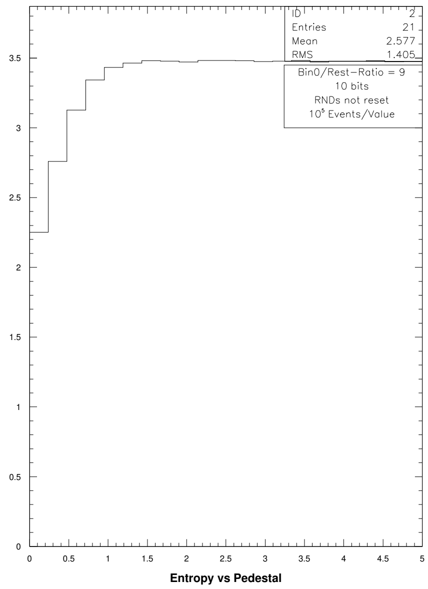

Abbildung 4.7: Entropie vs Pedestal bei 10bit

# Samples	# Sample im Maximum	Entropie (bits)
5	3	3.42
3	2	3.54
1	1	3.19

Tabelle 4.2: Entropie und Anzahl der Samples auf der Shapingkurve für 10 bit

4.1.3 Entropie und Energieauflösung

Vergleicht man Abbildung 4.2 mit 4.3 oder 4.6 mit 4.7, so fällt auf, daß die Entropie beim Übergang von 8 bit auf 10 bit Auflösung ebenfalls um fast 2 bit zunimmt. Diese Zunahme ist praktisch ausschließlich auf das vorhandene Rauschen zurückzuführen. Bestimmt man nämlich die Entropie der Energieverteilung ohne Rauschen ($\sigma = 0$ GeV) für 8 bit und 10 bit Auflösung so ergibt sich eine Zunahme der Entropie um lediglich 0.3 bit. Das bedeutet, daß beim Übergang von 8 bit auf 10 bit beinahe keine physikalisch wichtige Information dazugewonnen, sondern hauptsächlich das elektronisches und Pile-Up Rauschen digitalisiert wird.

Bei der Digitalisierung treten Rundungsfehler auf. Sie liegen in einem Bereich von ± 0.5 LSBs. Bei einer Auflösung von 8 bit entspricht dies gerade der Breite der Gaußverteilung für das Rauschen. Die Digitalisierung wird modelliert durch Aufrunden, falls die Differenz zum nächstgrößeren Wert kleiner oder gleich einem halben LSB ist, ansonsten wird abgerundet. Liegt also bei 8 bit Auflösung das Maximum der Gaußverteilung bei einem halbzahligen Wert, ergeben sich bei der Digitalisierung in der Nähe des Maximums zwei Werte, einer links vom Maximum, einer rechts davon. Liegt das Maximum hingegen bei einem ganzzahligen Wert, so wird nur ein Wert digitalisiert. Dieser Effekt verschwindet bei 10 bit Auflösung. Zu beobachten ist die dadurch verursachte Schwankung der Entropie bei 8 bit in Abbildung 4.6.

4.1.4 Anzahl der Samples

Für die Änderung der Entropie mit der Zahl der ausgelesenen Bunch-Crossings gibt es keine einheitliche Tendenz. Es ist jedoch offensichtlich, daß die Entropie mit der Anzahl der Samples in der Nähe des Maximums zunimmt. Werden jedoch viele Werte hinter dem Peak, also im flachen Unterschwinger digitalisiert, so nimmt die Entropie ab. Die Stärke der Änderung hängt außerdem von der Steilheit der Energieverteilung ab.Verwendet man die Parameter aus Tabelle 4.1, so spielt sie keine wesentliche Rolle. Tabelle 4.2 zeigt ein paar Beispielwerte.

4.2 Kompressionsfaktoren

Die bisher vorgestellten Ergebnisse gaben einen Überblick über den Informationsgehalt der Kalorimeterdaten. In diesem Abschnitt wird die Güte der Kompressionsalgorithmen getestet. Hierzu wurden zunächst verschiedene Datensätze mit *DATASIM* erzeugt und anschließend mit der Kompressionssoftware komprimiert. Als Maß für die Leistungsfähigkeit der Algorithmen wird im folgenden der Kompressionsfaktor benutzt.

$$CF = \frac{\text{Bytes vor der Kompression}}{\text{Bytes nach der Kompression}}. \tag{4.1}$$

Da zu jedem File eine Angabe über seine Länge existiert, ist er sehr einfach zu bestimmen. Beim Huffman Coding kann etwas anders vorgegangen werden, da hier die mittlere Wortlänge $\bar{l}$ für diesen Datensatz einfach zu bestimmen ist. Da jedem Zeichen genau ein Codewort zugeordnet wird, ergibt sich für das Huffman Coding mit

$$CF = \frac{\text{Energieauflösung (bits)}}{\bar{l}\ \text{(bits)}}, \qquad \text{nur für Huffman Coding} \tag{4.2}$$

derselbe Wert für CF, wie in Gleichung 4.1.

Für die Diskussion und Darstellung der Ergebnisse werden häufig Begriffe und Abkürzungen verwendet, die in Anhang A und Kapitel 3 genauer erläutert sind. Der Übersicht halber wurden sie in Tabelle 4.3 nocheinmal zusammengestellt.

DW	Data Word: Ein Datenwort
ZM	Zero Message: kurzer Blockcode
RLM	Run-Length Message: Länge einer Folge von gleichen Energien
NM	Normal Message: unveränderter Energiewert
E_R	Referenzenergie
Δ	Bereich kurzer Codes um die Referenzenergie
$H(S)$	Entropie eines Ensembles von Zeichen
$\bar{l}$	mittlere Codewortlänge

Tabelle 4.3: Abkürzungen für nullunterdrückende Algorithmen

4.2.1 Huffman Coding

Für die Realisation eines Huffmancoders wird eine Codetabelle benötigt. Diese wurde auf dem RemASIC als Look-Up Table realisiert. Das bedeutet, die Huffman Codes sind in einem RAM[2] gespeichert und werden, durch Anlegen des zu codierenden Zeichens als Adresse, ausgelesen (genaueres siehe 5.2). Die Huffman Codes sind also in ihrer Länge limitiert auf die Breite der LUT. Macht man die LUT zu klein, muß der Codebaum unter Umständen abgeschnitten werden (siehe Abschnitt 3.5). Damit sinkt jedoch die Güte der Kompression. Wird die LUT andererseits größer als benötigt, verschwendet man unnötig Platz auf dem RemASIC und steigert die Produktionskosten.

Eine erste Untersuchung widmete sich daher der Länge der Huffman Codes für typische Energieverteilungen. Außerdem war von Interesse, wie sich $\bar{l}$ und damit CF ändert, wenn die Länge der Huffman Codes auf einen Wert l_{max} limitiert wird. Die Ergebnisse sind in Tabelle 4.4 zu sehen. Sie wurden für einen Datensatz erhalten, der nur einen Kanal mit einem Pedestal von 1.5 GeV und 10 bit Auflösung simuliert. Der längste Huffman

[2] Random Access Memory: Speicher mit wahlfreiem Zugriff

Code hat, wenn der Codebaum nicht abgeschnitten wird, eine Länge von 19 bits. Der Anstieg von $\bar{l}$ mit abnehmendem l_{max} verwundert nicht. Erstaunlich ist hingegen auf den ersten Blick der Sprung von $\bar{l}$ auf 10 bit bei $l_{max} = 10$ bits. Der Grund dafür wird jedoch sofort klar, wenn man bedenkt, daß die Auflösung der Energie in der Simulation 10 bit beträgt. Erlaubt man keine Codewörter, die länger sind als 10 bit, so muß zwangsläufig jedes Codewort 10 bit lang sein, wenn keine Information verloren gehen soll.

l_{max} (bits)	10	11	12	13	14	15	16	17	18	19
$\bar{l}$ (bits)	10	5.72	4.98	4.20	3.76	3.60	3.55	3.53	3.52	3.51
CF	1.0	1.7	2.0	2.4	2.7	2.8	2.8	2.8	2.8	2.9
$H(S)$ (bits)	3.47									

Tabelle 4.4: Abschneiden des Codebaumes

Bei einem Datensatz, der 8 Kanäle mit Pedestals von 1 GeV bis 3 GeV enthält, ändert sich lediglich die minimal erreichbare mittlere Codewortlänge $\bar{l}$. Für die Darstellung des längsten Huffman Codes werden auch hier 19 bits benötigt. Die LUT auf dem RemASIC sollte also groß genug sein, um 19 bit Codes aufnehmen zu können. Um über eine gewisse Sicherheitsreserve zu verfügen, wurde eine LUT realisiert, die 23 bit lange Codes aufnehmen kann (siehe Kapitel 5.4.1).

Aus Tabelle 4.4 kann man außerdem entnehmen, daß die Entropie $H(S)$ des Datensatzes und $\bar{l}$ sehr nahe beieinander liegen, falls der Codebaum nicht abgeschnitten wird (Tabelle 4.4 bei $l_{max} = 19$ bits). In der Tat sollte man dieses Verhalten auch erwarten, da der Codebaum direkt aus den Häufigkeiten der Energiewerte im Datensatz gebildet wurde. In der Realität passiert das natürlich nicht. Der Codebaum wird vielmehr einmal aufgrund einer möglichst genauen Energieverteilung festgelegt, und dann auf sämtliche auszulesende Daten angewandt. Das bedeutet jedoch, daß die Kompression umso schlechter wird, je weniger die Wahrscheinlichkeitsverteilung der Daten in einem bestimmten Zeitraum der angenommenen Energieverteilung entspricht.

Für den Betrieb der Kompressionseinheit bei der Detektorauslese ist es natürlich interessant zu untersuchen, wie sich der Kompressionsfaktor quantitativ verschlechtert, wenn die Energieverteilung eines Datensatzes nicht der Verteilung entspricht, mit der der Codebaum konstruiert wurde. Hierzu wurde zunächst ein Datensatz mit einem Pedestal von $P = 0$ GeV erzeugt und gefittet (Abbildung 4.8). Dieser Fit ist notwendig, da ein von der Simulation erzeugter Datensatz nicht unbedingt jeden möglichen Energiewert enthalten muß. Würde man den Codebaum einfach durch Zählen der Häufigkeiten der einzelnen Energien in diesem Datensatz erzeugen, so gäbe es nicht zu jedem Energiewert einen Huffman Code.

Aus der gefitteten Verteilung wurde eine Tabelle mit den Häufigkeiten aller vorkommenden Energien erzeugt, aus der dann schließlich der Codebaum erhalten wurde. Komprimiert man mit dem Baum für $P = 0$ GeV einen Datensatz mit $P = 1.5$ GeV, so sinkt der Kompressionsfaktor auf 1.5 (Tabelle 4.5). Auch eine Verteilung mit einem Pedestal von 5 GeV wurde auf diese Weise komprimiert. Der Kompressionsfaktor liegt nun nahe bei eins. Es ist jedoch bemerkenswert, daß selbst bei dieser Verschiebung noch keine Auf-

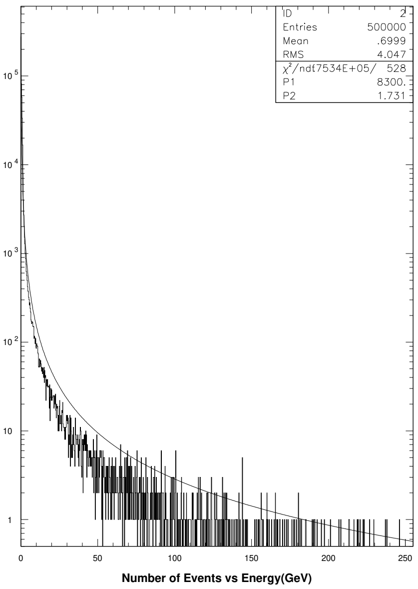

Abbildung 4.8: Energieverteilung und Fit für $P = 0$ GeV und 10 bit Auflösung

blähung bei der Codierung auftritt. Ein vielfach geäußertes Argument gegen die Huffman Codierung war die Befürchtung, daß sich schon bei kleinen Pedestalschwankungen der Code stark aufblähen könnte, was ganz offensichtlich nicht der Fall ist.

Baum für $P = 0$ GeV			
Pedestal (GeV)	0	1.5	5
Entropie (bits)	2.25	3.47	3.47
$\bar{l}$	2.36	6.36	9.33
CF	4.24	1.51	1.07

Tabelle 4.5: Kompression mit falschem Codebaum

Der Kompressionsfaktor wurde außerdem bestimmt für eine Verteilung mit einem Pedestal von 0 GeV und verschiedenen Energie-Cuts. Diese Untersuchung ist interessant, um die Leistungsfähigkeit von Run-Length Encoding und Huffman Coding vergleichen zu können (siehe dort). In Tabelle 4.6 sind die Ergebnisse zusammengestellt. Auf dem RemASIC werden die Daten aus bis zu 16 Kanälen komprimiert. Jeder dieser Kanäle hat ein separat einstellbares Pedestal, so daß eine Energieverteilung für nur einen Kanal mit konstantem Pedestal zu gute Werte für CF liefert. Eine realistische Abschätzung des Kompressionsfaktors gibt eine Verteilung für acht Kanäle mit Pedestals zwischen 1 GeV und 3 GeV. Man erreicht damit einen Kompressionsfaktor von 2.5 bei 10 bit Auflösung. Dieser Wert würde auch ausreichen, um alle Level-1 Daten auszulesen (siehe 1.2).

Energie-Cut (GeV)	CF
0	4.4
1	7.6
2	8.4
4	9.0

Tabelle 4.6: Kompression mit Energie-Cut

Sämtliche Untersuchungen zum Huffman Coding wurden für 10 bit Energiewerte durchgeführt. Wie bereits erläutert (4.1) erhöht sich beim Übergang von 8 bit auf 10 bit die Entropie um fast 2 bit. Der Kompressionsfaktor wird also bei 8 bit Daten um einiges besser.

4.2.2 Run-Length Encoding

Wie bereits bei der Diskussion der Kompressionsalgorithmen (Kapitel 3) erwähnt, eignet sich das Run-Length Encoding besonders für Datensätze, bei denen oft hintereinander die Null übertragen wird. Für Datensätze ohne Energie-Cut sollte daher der Kompressionsfaktor nahe bei eins oder sogar kleiner sein, was eine Aufblähung statt eine Kompression der Daten bedeuten würde. Deswegen wurde für Daten ohne Energie-Cut zunächst die Leistungsfähigkeit eines modifizierten Run-Length Encodings untersucht. Dabei wurde angenommen, daß bereits eine Pedestalsubtraktion stattgefunden hat, ohne die das Run-Length Encoding in der auf dem RemASIC realisierten Variante nutzlos ist.

Wird kein Energie-Cut angewandt, so sind die niedrigen Energiewerte von Rauschen dominiert und die zusammenhängenden Folgen mit $E = 0\ GeV$ werden sehr kurz sein. Man könnte sich nun vorstellen die Lauflänge anstatt als 8 bit oder 10 bit Zahl einfach mit entsprechend weniger bits zu kodieren. Auf diese Weise läßt sich selbst für kurze Nullfolgen eine Kompression erreichen. Für längere Folgen muß dann unter Umständen mehrmals hintereinander eine Null und die Lauflänge übertragen werden. Die Ergebnisse der Simulation sind in Tabelle 4.7 zu sehen. Die Wortlänge für den Lauflängenzähler l_{RLM} wurde solange erhöht, bis der Kompressionsfaktor wieder kleiner wird, so daß die optimale Länge bestimmt werden konnte. Bei einer Auflösung von 8 bits beträgt sie 4 bit und liegt damit höher als bei einer Auflösung von 10 bit, wo das Optimum bei 3 bit liegt. Dies ist mit dem geringeren Einfluß des Rauschens auf die Energiewerte bei niedrigerer Auflösung zu erklären.

Da die Kompressionsfaktoren nicht besonders gut sind, und vor allem da keine festen Wortgrenzen existieren, wurde diese Möglichkeit der Codierung verworfen. Tritt zum Beispiel bei der Übertragung einer Null ein Fehler auf, so wird das nachfolgende Wort als normale Energie interpretiert und der gesamte restliche Datensatz wird falsch rekonstruiert. Dieses Problem gibt es beim ursprünglichen Run-Length Encoding, bei dem die Lauflänge mit der gleichen Anzahl an bits codiert wird, wie die Energiewerte ($l_{RLM} = l_{NM}$), nicht. Es wurde daher genauer untersucht und auf dem RemASIC implementiert. In Tabelle 4.8 sind die erreichbaren Kompressionsfaktoren für verschiedene Energie-Cuts dargestellt. Auch hier wurde angenommen, daß eine Pedestalsubtraktion stattgefunden hat.

Auflösung (bits)	l_{RLM} (bits)	CF
10	2	1.27
10	3	1.33
10	4	1.30
8	2	1.64
8	3	2.12
8	4	2.29
8	5	2.18

Tabelle 4.7: Run-Length Encoding mit kurzen Codes für die Lauflänge

Auflösung (bits)	Energie-Cut (GeV)	CF
10	0	1
10	1	8
10	2	18
10	4	34
8	0	2
8	1	13
8	2	27
8	4	33

Tabelle 4.8: Run-Length Encoding bei verschiedenen Energie-Cuts

Vergleicht man diese Werte mit denen aus Tabelle 4.6 für das Huffman Coding, so zeigt sich die klare Überlegenheit des Run-Length Encoding für diese Art von Daten. Die erreichbaren Kompressionsfaktoren sind unschlagbar. Es muß jedoch nocheinmal betont werden, daß dieses Verfahren nur für Daten nach Pedestalsubtraktion und mit einem Energie-Cut von mindestens 1 GeV sinnvoll ist. Unter diesen Voraussetzungen ist es allerdings das Mittel der Wahl.

4.2.3 Huffman-Inspired und Difference Coding

In Kapitel 3 wurde erklärt, daß Huffman-Inspired und Difference Coding auf demselben Prinzip beruhen. Beim Huffman-Inspired Coding wird die Differenz zwischen dem aktuellen Energiewert und einem festen Referenzwert E_R betrachtet. Das Difference Coding benutzt die Differenz zweier aufeinanderfolgender Energiewerte. Aufgrund der Ähnlichkeit der beiden Algorithmen, sind auch die Untersuchungen beinahe identisch und werden daher zusammen diskutiert.

Zunächst einmal wird man daran denken, Datenwörter mit einer Länge von $l_{DW} = 8$ bit oder $l_{DW} = 10$ bit zu verwenden und nach der optimalen Länge der kurzen Blockcodes l_{ZM} zu suchen. Damit ist die Länge der Datenwörter an die Länge der *Normal Messages* angepaßt und es wird bei der Übertragung eines unveränderten Energiewertes kein Platz verschenkt. Benutzt man jedoch bei einer Datenwortlänge von 10 bit Blockcodes mit einer Länge von beispielsweise 3 bit, so verschenkt man bei der Übertragung der Blockcodes mit jedem Datenwort 1 bit Platz. Um die optimalen Parameter für die Codierung zu finden, müssen daher sowohl l_{DW}, als auch l_{ZM} variiert werden.

Die Ergebnisse der Variation von l_{ZM} und l_{DW} sind in Tabelle 4.9 für das Difference Coding mit 10 bit dargestellt. Die Tabelle 4.10 für 8 bit Energiewerte wurde nur der Vollständigkeit halber angeführt. Da auf dem RemASIC Huffman-Inspired und Difference Coding nur für 10 bit durchgeführt wird, widmet sich die Diskussion der Kompressionsfaktoren ausschließlich den Ergebnissen für 10 bit. Man sieht, daß der Kompressionsfaktor bei einer Datenwortlänge von 12 bit und einer Länge der kurzen Codes von 4 bit am besten ist. Diese Werte wurden mit einem festen Pedestal von 1.5 GeV ermittelt. Für das Huffman-Inspired Coding erhält man dieselben Ergebnisse. Allerdings ist dabei zu beachten, daß die Kompression nur solange gut ist, wie Referenzenergie E_R und Pedestal möglichst genau übereinstimmen. Bei der Verwendung von 12 bit Datenwörtern werden keine bits verschenkt, wenn mit kurzen Codes codiert wird, da gerade drei der 4 bit Codes in ein 12 bit Wort passen. Andererseits bläht man den Code um 2 bits pro Datenwort auf, falls Energiewerte unverändert übertragen werden.

l_{DW} (bits)	l_{ZM} (bits)	CF
10	2	1.1
10	3	1.6
10	4	1.7
10	5	1.8
12	3	1.5
12	4	2.0
12	5	1.8

Tabelle 4.9: CF in Abhängigkeit von Datenwortlänge und Länge der kurzen Codes für 10 bit Daten

l_{DW} (bits)	l_{ZM} (bits)	CF
8	2	2.7
8	3	1.7
8	4	1.8
9	3	2.3
10	2	2.6

Tabelle 4.10: CF in Abhängigkeit von Datenwortlänge und Länge der kurzen Codes für 8 bit Daten

Der Effekt einer Verschiebung von Pedestal und Referenzenergie ist in Tabelle 4.11 dargestellt. Schon bei einer Verschiebung um 2.5 GeV wird der Datensatz nicht mehr komprimiert, sondern aufgebläht. In Abbildung 4.9 ist die verwendete Energieverteilung zu sehen. Man kann sehr gut erkennen, daß die Referenzenergie ziemlich genau im Maximum der Verteilung liegen muß, um eine Kompression zu erreichen. Diese Probleme treten beim Difference Coding nicht auf, da es unabhängig vom Pedestal ist.

Sowohl Difference, als auch Huffman-Inspired Coding wurden mit einem Datensatz, der 8 Kanäle mit Pedestals zwischen 1 GeV und 3 GeV enthält getestet. Er wurde auch schon beim Huffman Coding verwendet. Man erzielt Kompressionsfaktoren von 2.0, was noch genügen sollte, um den Trigger auszulesen (Tabelle 4.12).

E_R (GeV)	$\mid P - E_R \mid$ (GeV)	Δ (GeV)	CF
2.5	2.5	0.75 .. 4.25	0.78
4.5	0.5	3.75 .. 7.25	2.2
5.0	0.0	3.25 .. 6.75	2.1
5.5	0.5	4 .. 7.5	2.2
7.5	2.5	5.75 .. 9.25	0.8

Tabelle 4.11: Kompressionsfaktor mit verschobener Referenzenergie für 10 bit Daten, $l_{DW} = 12$ und $l_{ZM} = 4$

	Pedestals (GeV)	CF
Difference	1 .. 3	2.0
Huffman-Inspired	1 .. 3	2.0

Tabelle 4.12: Kompressionsfaktoren für Datensatz mit acht verschiedenen Pedestals

Um das Bild von der Leistungsfähigkeit abzurunden, wurde der CF für das Difference Coding noch in Abhängigkeit vom Bin0/Rest-Ratio bestimmt. Je niedriger dieses Verhältnis wird, desto öfter befindet sich die betrachtete Kalorimeterzelle im Radius eines Jets. In Tabelle 4.13 kann man erkennen, daß der Kompressionsfaktor zwar abnimmt, aber selbst für die doppelte Anzahl von Jet-Ereignissen (Bin0/Rest-Ratio = 5) noch bei 2.0 liegt. Eine Aufblähung des Datensatzes ist nicht zu befürchten. Durch entsprechende Anpassung von l_{DW} und l_{ZM}, läßt sich der Algorithmus optimal an verschiedene Energieverteilungen anpassen (letzte Zeile in Tabelle 4.13).

Bin0/Rest-Ratio	l_{DW} (bits)	l_{ZM} (bits)	CF
0	12	4	1.3
5	12	4	2.0
9	12	4	2.1
0	10	5	1.6

Tabelle 4.13: CF in Abhängigkeit vom Bin0/Rest-Ratio

Auch die Kompressionsfaktoren für verschiedene Energie-Cuts wurden untersucht (Ta-

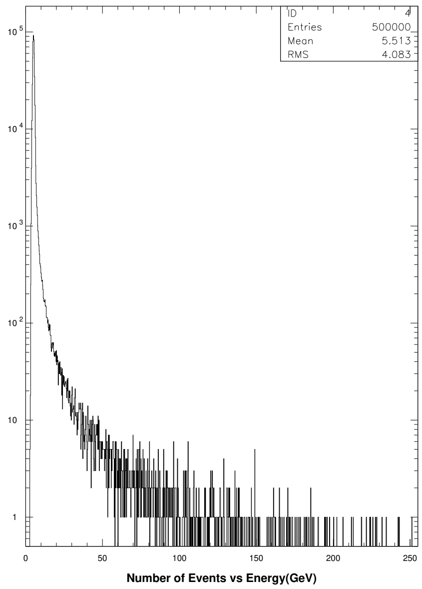

Abbildung 4.9: Energieverteilung zur Demonstration des Effektes der Pedestalverschiebung

belle 4.14). Wie zu erwarten ergeben sich nicht die sehr hohen Kompressionsfaktoren, wie beim Run-Length Encoding. Das Difference Coding ist jedoch, wie gezeigt wurde, unempfindlich gegenüber Pedestalschwankungen. Auch eine Vergrößerung der Anzahl getroffener Kalorimeterzellen verkraftet dieser Algorithmus recht gut. Auch unter schlechten Bedingungen ist noch ein Kompressionsfaktor von 2 erreichbar. Beim Huffman-Inspired Coding ist auf eine gute Übereinstimmung von Referenzenergie und Maximum der Energieverteilung zu achten, da sonst eine Codeaufblähung zu befürchten ist. Auch bei optimaler Lage von E_R lassen sich mit dem Huffman-Inspired Coding keine besseren Resultate erzielen, als mit dem Difference Coding.

Energie-Cut	CF
0	2.1
1	2.1
2	2.1
4	2.2

Tabelle 4.14: CF in Abhängigkeit vom Bin0/Rest-Ratio mit $l_{DW} = 12$ und $l_{ZM} = 4$

4.2.4 Zusammenfassung

Die Leistungsfähigkeit der Kompressionsalgorithmen in Abhängigkeit verschiedener Parameter wurde untersucht und diskutiert. Damit die Fülle von Daten und Tabellen nicht den Blick auf das Wesentliche verstellt, sind in Tabelle 4.15 typische Kompressionsfaktoren zusammengestellt. Es wurde gezeigt, daß alle vorgestellten Algorithmen in der Lage sind die Bandbreite bei der Auslese so weit zu verringern, daß der komplette Level-1 Trigger über den PipelineBus ausgelesen werden kann.

Algorithmus	Parameter	typischer CF
Huffman	8 Pedestal von 1 .. 3 GeV	2.5
Huffman	Energie-Cut 1 GeV	7.6
Huffman Inspired Difference	8 Pedestal von 1 .. 3 GeV	2
Huffman Inspired Difference	Energie-Cut 1 GeV	2
Run-Length	Energie-Cut 1 GeV	13

Tabelle 4.15: Typische Kompressionsfaktoren für 10 bit Daten

Die Hintereinanderschaltung verschiedener Algorithmen, wie zum Beispiel das Huffman Coding von Run-Length codierten Daten, wurde nicht untersucht. Die Implementation der Kompressionseinheit wäre damit um einiges aufwendiger geworden und hätte den eng gesteckten Zeitrahmen für die Implementation gesprengt. Außerdem war eine solche Doppelcodierung nicht erwünscht, da die Rekonstruktion der Ausgangsdaten aufwendiger geworden wäre. Für ein finales System sollte jedoch die Möglichkeit des Huffman Coding von bereits codierten Datensätzen in Betracht gezogen werden.

Kapitel 5

Die Kompressionseinheit

Wesentlicher Bestandteil der Diplomarbeit war die Implementation der getesteten Algorithmen auf dem RemASIC. Der Aufbau der Kompressionseinheit und die dabei verwendeten Methoden werden in diesem Kapitel erläutert. Die Kompressionseinheit ist sehr stark eingebunden in die Infrastruktur, die von FeASIC Interface und PipelineBus Interface zur Verfügung gestellt wird. Sie wird über den PipelineBus konfiguriert und verfügt über keine eigenen I/O-Pins[1]. Daher wird zunächst der Aufbau des RemASICs vorgestellt, um dann die Architektur der Kompressionseinheit im Detail diskutieren zu können.

5.1 Der RemASIC - ein Überblick

5.1.1 Die Idee

Der Readout Merger ASIC wurde entwickelt, um folgende Funktionalität nach außen hin zur Verfügung zu stellen:

- Auslese und Konfiguration von bis zu 16 FeASICs über 4 Anschlüsse (Ports)
- Kompression der Daten mit verschiedenen Algorithmen
- PipelineBus Interface für Konfiguration und Auslese.

Darüber hinaus, sollte der Chip möglichst einfach zu testen sein. Es wurde daher beschlossen auf dem RemASIC drei Testmöglichkeiten zu integrieren, nämlich:

- Überprüfung interner Signale
- Unterbrechung der Auslese
- Datenquelle zum Test ohne FeASIC

In Abbildung 5.1 ist ein Blockdiagramm des RemASIC zu sehen. Er ist in drei Funktionsblöcke aufgeteilt, das FeASIC Interface, die Kompressionseinheit und das PipelineBus Interface. Die Blöcke sind in einer pipelineartigen Struktur angeordnet, wodurch sich eine große Flexibilität für den Test ergibt.

[1] Input-/Output-Pins, die Ein- und Ausgangsanschlüsse eines Chips

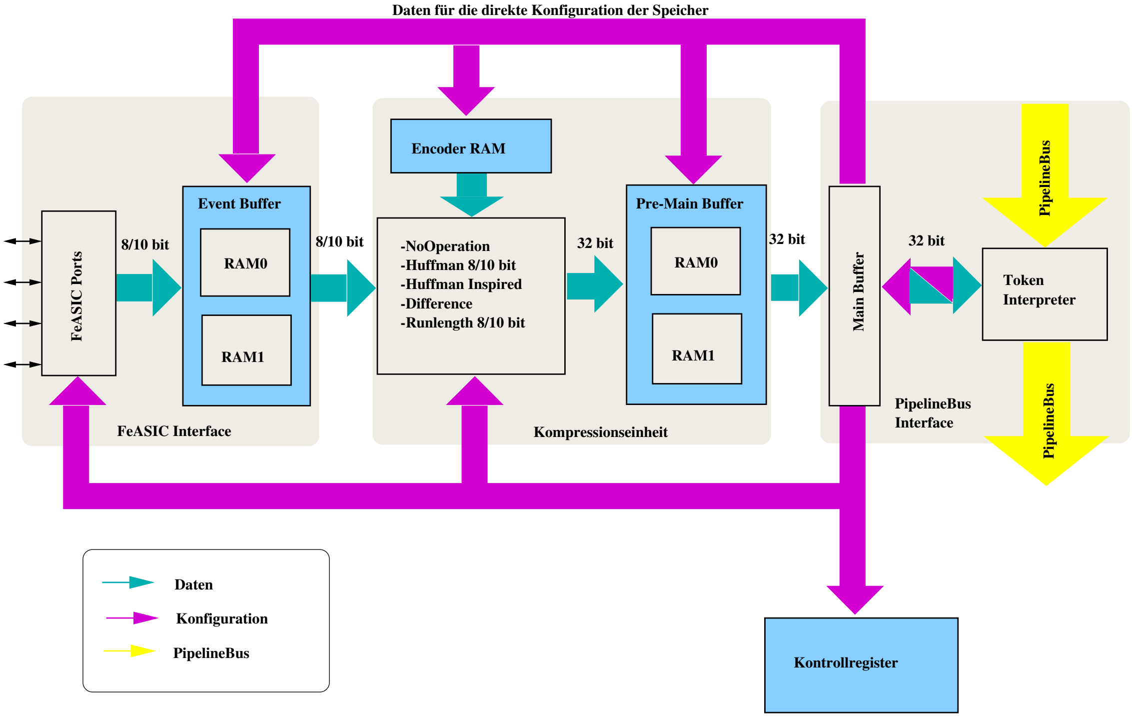

Abbildung 5.1: Blockdiagramm des RemASIC

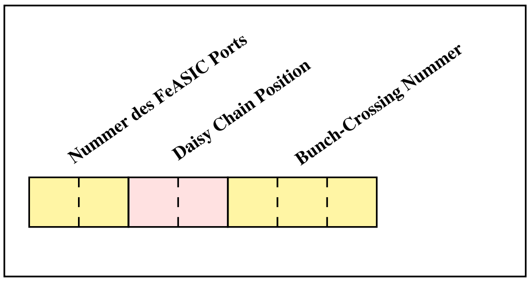

Abbildung 5.2: Struktur der Adressen im Event Buffer

FeASIC Interface

Die Daten von bis zu 16 FeASICs werden über die vier *FeASIC Ports* ausgelesen. Im FeASIC Interface erfolgt dann die Berechnung eines Headerwortes zu jedem Event. Anschließend werden die Daten so in einem der beiden Event Buffer abgelegt, daß die zu einem Event ausgelesenen Bunch-Crossings in einem Block direkt hintereinander stehen (*Event Ordering*). Das Adressschema ist in Abbildung 5.2 zu sehen. Die untersten 3 bit der Adresse geben die Nummer des Bunch-Crossings an, dann folgen 2 bits, in denen die Position in der Daisy Chain steht, und die obersten 2 bits geben die Nummer des FeASIC Ports an. Durch die Verwendung von zwei RAMs kann das FeASIC Interface ein Event in einem RAM ablegen, während die Kompressionseinheit aus dem anderen RAM das vorhergehende Event ausliest.

Kompressionseinheit

Auf die Daten aus dem Event Buffer wendet die Kompressionseinheit den vorher eingestellten Algorithmus an. Für das Huffman Coding wird der Encoder RAM, der die Tabelle mit den Huffman Codes enthält, als LUT benötigt. Die komprimierten Daten werden in 32 bit Worte formatiert und im Pre-Main Buffer abgelegt. Er ist aus zwei RAMs mit jeweils 48 Datenworten aufgebaut. Auf diese Weise ist es möglich zugleich ein fertig komprimiertes Event in den Main Buffer zu kopieren und mit der Kompression eines neuen Events zu beginnen.

PipelineBus Interface

Der Main Buffer ist ein FIFO[2] , der in zwei Richtungen betrieben werden kann. Dadurch ist es möglich den RemASIC über den PipelineBus auszulesen und zu konfigurieren. Der TokenInterpreter erkennt Tokens auf dem PipelineBus und reagiert entsprechend darauf. Außer der 32 bit breiten Schnittstelle, die insgesamt 70 Leitungen, jeweils 32 Daten- und 3 Kontrolleitungen für Ein- und Ausgabe, benötigt, steht auch ein serielles Interface zur Verfügung. Dadurch wird ein Betrieb mit nur 2 Datenleitung ermöglicht, was vor allem für Testzwecke sinnvoll ist (siehe Kapitel 6).

[2] First in first out: Daten, die zuerst in einem FIFO gespeichert werden, werden auch als erste ausgelesen.

Devices

Der RemASIC enthält vier Blöcke für die Speicherung von Daten, den Event Buffer, Encoder RAM, Pre-Main Buffer und die Kontrollregister für die Konfiguration des Chips. Sie werden im folgenden als *Devices* bezeichnet. Jedes Device kann über den Pipeline-Bus konfiguriert werden, wenn der Main Buffer Daten über den Bus aufnimmt. Damit ist es zum Beispiel möglich den Event Buffer mit Testdaten zu beschreiben und ihn als Datenquelle für die Kompression zu benutzen. Auf diese Weise kann der RemASIC ohne FeASICs betrieben werden.

Umgekehrt kann auch jedes Device in den Main Buffer kopiert und von dort auf den PipelineBus gegeben werden. Über den PipelineBus läßt sich auch die Auslese anhalten. FeASIC Interface und Kompressionseinheit arbeiten dann solange weiter, bis sie ein eventuell angefangenes Event fertig bearbeitet haben. Auf diese Weise ist die Auslese der Daten nach jeder Verarbeitungsstufe möglich.

SpyBus

Die bisher dargestellte Funktionalität ermöglicht den Auslesebetrieb und Test mit vielfältigen Konfigurationsmöglichkeiten. Es wird jedoch vorausgesetzt, daß der Chip elektrisch fehlerfrei funktioniert. Zum Auffinden von Design- oder Produktionsfehlern wurde zusätzlich ein sogenannter *SpyBus* integriert. Dies ist ein 8 bit breiter Bus auf den über einen Multiplexer 64 Signale gegeben werden können. So ist zum Beispiel die Kontrolle von internen Registern möglich. Außerdem können über den SpyBus die Signale der Selbsttestfunktion der verschiedenen Speicher ausgelesen werden. Diese Funktion wird als BIST (*Built In Self Test*) bezeichnet.

5.1.2 Das Design

Der RemASIC besteht aus drei Blöcken (siehe Abschnitt 5.1.1), die jeweils einen Teil der Funktionalität zur Verfügung stellen. Jeder dieser Blöcke wurde von einem Designer implementiert, so daß insgesamt drei Leute an der Entwicklung des Chips gearbeitet haben. Damit die unabhängig entwickelten und getesteten Blöcke am Schluß zu einem funktionsfähigen Chip zusammengesetzt werden können, wurden zunächst die Schnittstellen der einzelnen Blöcke untereinander und nach außen festgelegt.

Die Verilog-Beschreibung

Für die Implementation des RemASIC wurde die *Verilog Hardware Description Language* (Verilog HDL) verwendet [22]. Sie bietet die Möglichkeit das Verhalten einer Digitalschaltung inklusive Zeitverhalten (*Timing*), zu modellieren. Für eine Darstellung der Sprache siehe [23].

Das zentrale Element von Verilog ist das Modul. Module haben Verbindungen nach außen (Input und Output) und bilden das Verhalten eines kleinen Teils der Gesamtschaltung nach. In einem Modul können wiederum Module enthalten (instanziiert) sein. Die Module werden jedoch nicht nacheinander abgearbeitet wie in einer sequentiellen Programmiersprache, sondern parallel. Diese Parallelität ist zwingend erforderlich für die Modellierung einer Schaltung und stellt den Hauptunterschied zwischen einer sequentiellen Sprache wie C und einer Hardwarebeschreibungssprache dar.

Der RemASIC wurde zunächst entsprechend den Funktionsblöcken in drei Module unterteilt, von denen die Ein- und Ausganssignale und deren Timing festgelegt wurden. Die Implementation der gewünschten Funktionalität innerhalb der Blöcke blieb jedem Entwickler selbst überlassen. Um das Timing möglichst unkritisch zu gestalten werden bis auf wenige Ausnahmen im FeASIC und PipelineBus Interface alle Flipflops mit demselben System-Takt betrieben. Dieses Vorgehen bezeichnet man als synchrones Design. Der Sourcecode ohne Kommentare umfaßt 22 kByte.

Synthese und Layout

Um aus der Verilog Beschreibung einen ASIC zu erstellen, wird eine Netzliste benötigt, die die verwendeten Bauelemente und deren Verbindungen untereinander beinhaltet. Sie läßt sich graphisch als Schaltplan darstellen. Man erhält die Netzliste mit Hilfe von sogenannten Synthese-Tools. Für die Synthese des RemASIC wurde das Programm *Synergy* verwendet. Es ist beschrieben in [24]. Bei der Modellierung müssen bestimmte Regeln beachtet werden, um ein synthetisierbares Design zu erhalten [25]. So darf zum Beispiel nicht der gesamte Verilog Sprachumfang verwendet werden und es muß eine strenge Trennung von Blöcken mit sequentieller (getakteter) und rein kombinatorischer Logik eingehalten werden.

Der Synthesizer verwendet Bibliotheken mit Standardzellen, die von den Halbleiterproduzenten zur Verfügung gestellt werden. Standardzellen sind vorgegebene häufig benutzte Bauelemente für Digitalschaltungen, wie zum Beispiel Gatter, Flipflops und Multiplexer. Mit diesen Elementen konstruiert der Synthesizer einen Schaltplan, der dieselbe Funktionalität wie die Verilog Beschreibung hat. Für den RemASIC wurde die ES2 ECPD-07 Bibliothek verwendet, die Standardzellen für den 0.7 μm Prozeß[3] der Firma ES2 enthält.

Die Speicherblöcke stehen nicht als Standardzellen, sondern als *Compiled Megacells* von ES2 zur Verfügung. Der Benutzer kann mit einem Programm Speicherblöcke beliebiger Größe generieren und in seinem Design verwenden. Es wird automatisch eine Verilog Beschreibung generiert, die für die Simulation verwendet werden kann, jedoch nicht synthetisierbar ist. Erst beim Produzenten werden diese Blöcke dann durch die entsprechende Schaltung ersetzt. Auf diese Weise sind Event Buffer, Encoder RAM, Pre-Main Buffer und Main Buffer realisiert worden.

Hat man einen vollständigen, funktionsfähigen Schaltplan des Chips, folgt das Layout. Es enthält alle Informationen, die für die Produktion des Chips notwendig sind. Die Bibliothekselemente der Netzliste werden durch die Standardzellen und deren Verbindungen ersetzt, es entsteht ein Standardzellblock. Diese Blöcke müssen zusammen mit den Speichern und den Anschlüssen für Eingänge, Ausgänge und Betriebsspannung (*Pads*) so verteilt werden, daß die Leitungen zwischen den Transistoren möglichst kurz und der verbrauchte Platz möglichst gering ist. Die Generierung des Layouts erfolgt weitgehend automatisch.

[3] Die kleinste Struktur auf dem ASIC ist 0.7 μm breit

Simulation

Ein Standardzellen Design auf Verilog Basis läßt sich auf verschiedenen Stufen der Entwicklung simulieren. Hierfür wird der *Verilog-XL Simulator* verwendet. Über eine spezielle Sprache, die *Simulation and Test Language* (STL) [26], können Testvektoren auf die Eingänge gegeben und die Ausgänge beobachtet werden.

In einer ersten Stufe wird der Verilog Source Code vor der Synthese simuliert und auf korrekte Funktion geprüft. Ist dies erfolgreich, wird der Verilog Code synthetisiert. Der Synthesizer erzeugt eine neue Verilog Beschreibung, in der auch die Durchlaufzeiten durch die Gatter, die in der Netzliste benutzt werden, enthalten sind. In einer erneuten Simulation muß nun überprüft werden, ob das synthetisierte Design dasselbe Verhalten zeigt, wie die Verilog Beschreibung. In einem letzten Schritt läßt sich ein aus dem Layout gewonnenes *Standard Delay File* (SDF) mit in die Simulation einbinden. Es enthält Informationen über die Verzögerungen (*Delays*), die durch die Leitungskapazitäten verursacht werden. Dieser Schritt war für den RemASIC leider nicht möglich, da das SDF-File zu groß war, um von der Software eingelesen werden zu können.

Zusätzlich gibt es die Möglichkeit, sich über ein Programm namens *Veritime* die Laufzeiten von verschiedenen Pfaden in der Schaltung anzeigen zu lassen. Pfade lassen sich gezielt über Angabe von Anfangs- und Endpunkt auswählen. Für jeden dieser Pfade werden maximale oder minimale Laufzeiten berechnet und angezeigt. Auch die Laufzeiten zwischen getakteten Bausteinen (Flipflops) können untersucht werden. Da der LHC mit einer Bunch-Crossing Frequenz von 40 MHz läuft (siehe Kapitel 1), und damit auch der Level-1 Trigger Ereignisse mit 40 MHz verarbeiten muß, wurde auch der RemASIC für einen Betrieb mit 40 MHz ausgelegt. Die Simulationen mit Verilog und Veritime zeigen, daß dies mit dem Design erreichbar ist.

5.1.3 Der Chip

In diesem Abschnitt wird das Resultat des Designprozesses, der fertige Chip, vorgestellt. Eine Fotografie des Dies unter dem Mikroskop ist in Abbildung 5.3 zu sehen. Man erkennt in der Mitte einen großen Standardzellblock. Die kleineren Blöcke oben und unten sind die RAM Zellen. Am äußeren Rand liegt ringsherum der Padkranz. Auch die Bonddrähte, die Verbindungen der Pads zu den Anschlüssen im Gehäuse (*Package*), sind zu erkennen.

Für die Verteilung der Clock wurde ein einziger großer Clockbuffer gewählt. Von ihm aus läuft jeweils eine Clock-Leitung links und rechts am Standardzellblock vorbei und verzweigt sich dann senkrecht in die einzelnen Standardzellreihen.

Auf dem Chip befinden sich etwa 160000 Transistoren, 800 Flipflops und 2.5 kByte Speicher. Er hat eine Fläche von etwa 50 mm^2 und 129 Ein- und Ausgangsleitungen. Er wurde für eine Taktfrequenz von 40 MHz ausgelegt. Eine Zusammenfassung der wichtigsten Daten des RemASIC ist in Tabelle 5.1 zu finden.

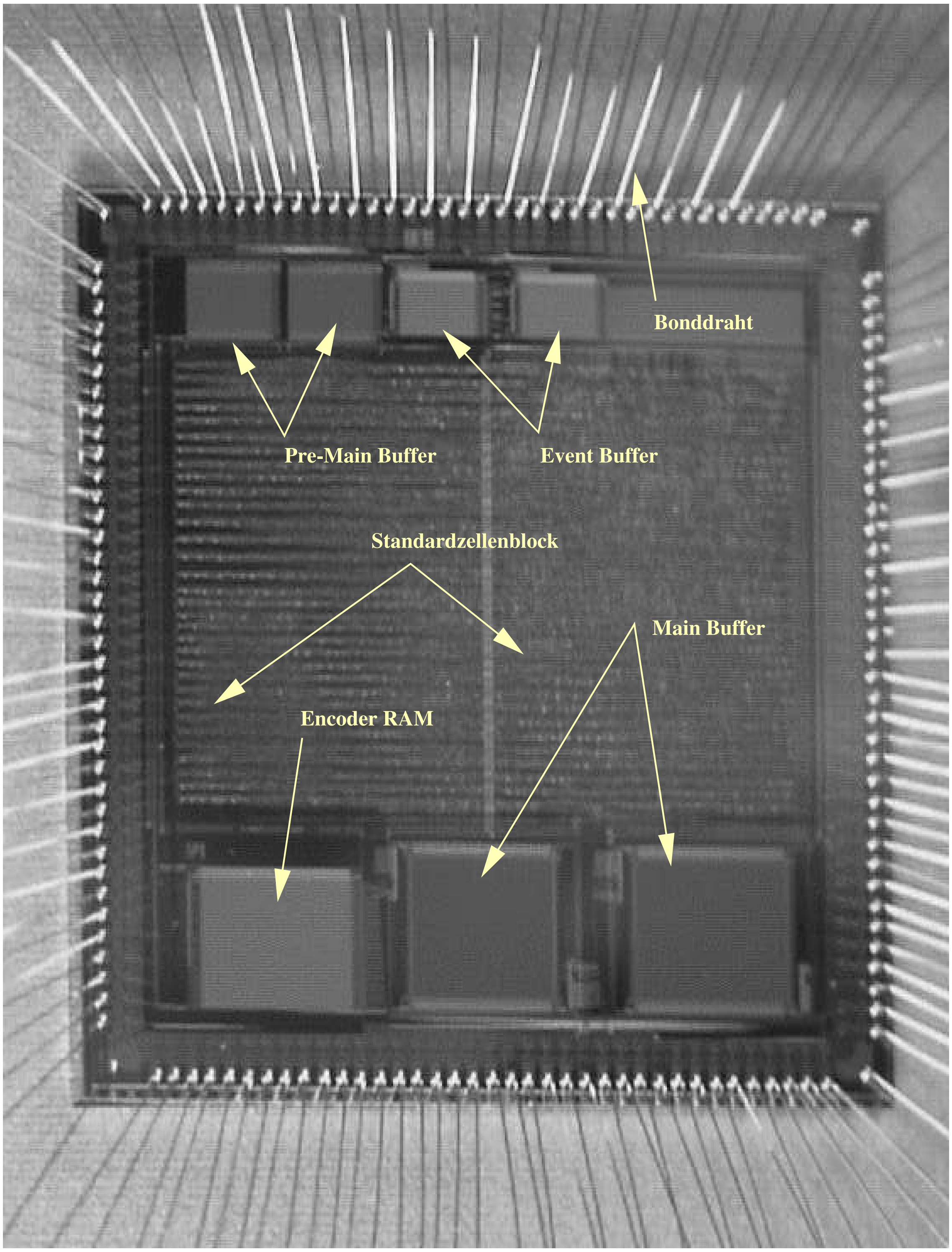

Abbildung 5.3: Vergrößerte Aufnahme eines RemASICs

Design	Verilog HDL, Standardzellen
Prozeß	0.7 μm CMOS von ES2
Die Fläche	7.8 mm $*$ 7.15 mm
Transistoren	ca. 160000
Flipflops	ca. 800
Speicher	2.5 kByte
Taktfrequenz	$\geq$ 40 MHz (simuliert)
I/O-Pins	129

Tabelle 5.1: Design Ergebnis auf einen Blick

5.2 Architektur der Kompressionseinheit

Die Kompressionseinheit wurde als hierarchisches Verilog Design ausgeführt. Entsprechend den einzelnen Funktionsblöcken, die in 5.2.2 beschrieben werden, wurde sie in Module unterteilt, die ihrerseits auch wieder aus Modulen bestehen können. Jede State Machine[4] belegt ein eigenes Modul.

5.2.1 Anforderungen

Am Anfang der Entwicklung stand eine Liste von Anforderungen an die Kompressionseinheit, die Implementation war hingegen noch weitgehend unklar. Folgende Kompressionsmodi sollten zur Verfügung stehen:

- NoOperation (Nop): keine Kompression
- Huffman
- Huffman Inspired
- Difference
- Run-Length

Der NoOperation Modus ist hier auch als Kompressionsmodus aufgeführt, da er sich der gesamten Infrastruktur der Kompressionseinheit bedient und lediglich als ein spezieller Algorithmus implementiert ist, der die Daten unverändert läßt und nur umformatiert. Jeder Kompressionsmodus sollte, soweit möglich, sowohl 8 bit als auch 10 bit Daten verarbeiten. An dieser Stelle wurden jedoch einige Kompromisse gemacht (siehe Abschnitt 5.4), um die Kompression nicht zu komplex werden zu lassen. Weitere Anforderungen sind

- Auslese der Daten aus dem Event Buffer unter Berücksichtigung des Event Ordering
- Alle zwei Taktzyklen muß ein Datum aus dem Event Buffer ausgelesen werden
- Formatierung der komprimierten Daten zu 32 bit Wörtern

[4]Zustandsmaschine, siehe 5.5 und [27].

- Konfiguration und Auslese jedes Speichers über den PipelineBus

Da im Schnitt alle 10 μs ein Level-1 Accpet erfolgt, das heißt ein Datensatz ausgelesen werden muß, ist es erforderlich, die Kompression auch im ungünstigsten Fall, das bedeutet Kompression von 128 Datenwörtern, nach höchstens 10 μs zu beenden. Daher muß alle zwei Taktzyklen ein Energiewert bearbeitet werden.

Eine Formatierung der Daten ist notwendig, da die Kompressionseinheit Datenwörter mit ganz unterschiedlicher Länge produziert, der PipelineBus jedoch eine feste Breite besitzt. Würde man jedes Datenwort einzeln auf den Bus geben, so wäre die bei der Kompression gewonnene Reduktion der erforderlichen Bandbreite wieder verschenkt.

Eine Konfiguration des Encoder RAMs ist in jedem Fall notwendig, um die Codetabelle für das Huffman Coding laden zu können. Die Konfiguration des Pre-Main Buffers, sowie eine Auslesemöglichkeit für beide Speicher ist für Testzwecke wünschenswert.

5.2.2 Implementation

Ein Blockdiagramm der Kompressionseinheit ist in Abbildung 5.4 zu sehen. Sie besteht aus vier Funktionsblöcken (Event Readout, Compression Modules, Shifting Unit und Pre-Main Buffer), drei Speicherblöcken (RAM0, RAM1 und Encoder RAM) und 4 Kontrollblöcken (CUN Control, Dataflow Control, Memory Control und ein Multiplexer).

Event Readout

Der Event Readout ist zuständig für die Auslese der Daten aus dem Event Buffer. Alle zwei Taktzyklen wird ein Datenwort ausgelesen und in einem Register gespeichert, damit es unabhängig von den Speicherzugriffszeiten volle zwei Taktzyklen für die weitere Verarbeitung in den Compression Modules zur Verfügung steht. Außerdem wird ein Signal generiert, das anzeigt, wann das letzte Datenwort aus dem Event Buffer ausgelesen wurde.

Die Auslese erfolgt nicht immer konsekutiv, da sich die Adresse eines Energiewertes zusammensetzt aus Portnummer, Daisy Chain Position und Bunch-Crossing Nummer (siehe Abschnitt 5.1.1). Hängen zum Beispiel am FeASIC Port 0 und 3 jeweils ein FeASIC, die pro Event fünf Bunch-Crossings auslesen, so müssen zuerst die Adressen 0x0-0x5[5], und dann 0x60-0x65 ausgelesen werden. Die Register *RegPortMask* und *RegDaisyChainCount* enthalten die Konfiguration der FeASIC Ports. Die Anzahl der Bunch-Crossings steht in *RegBcPerEvent*. Über diese Register wird festgelegt, an welchen Adressen sich gültige Daten befinden, die auszulesen sind.

Compression Modules

In den Compression Modules erfolgt die eigentliche Kompression der Daten. Der Kompressionsmodus wird über das Register *RegComprMode* festgelegt. In *RegComprParameter* wird für das Huffman-Inspired Coding die Referenzenergie festegelegt, beim Run-Length Encoding läßt sich so ein Energie-Cut wählen (siehe auch Anhang B).

Jeder Algorithmus ist als eigenständiges Modul realisiert. Aus dem Register *RegComprMode* wird ein Signal generiert, das das entsprechende Modul aktiviert, so daß nur einer

[5]Das vorangestellte 0x bedeutet, daß die folgende Zahl hexadezimal dargestellt ist.

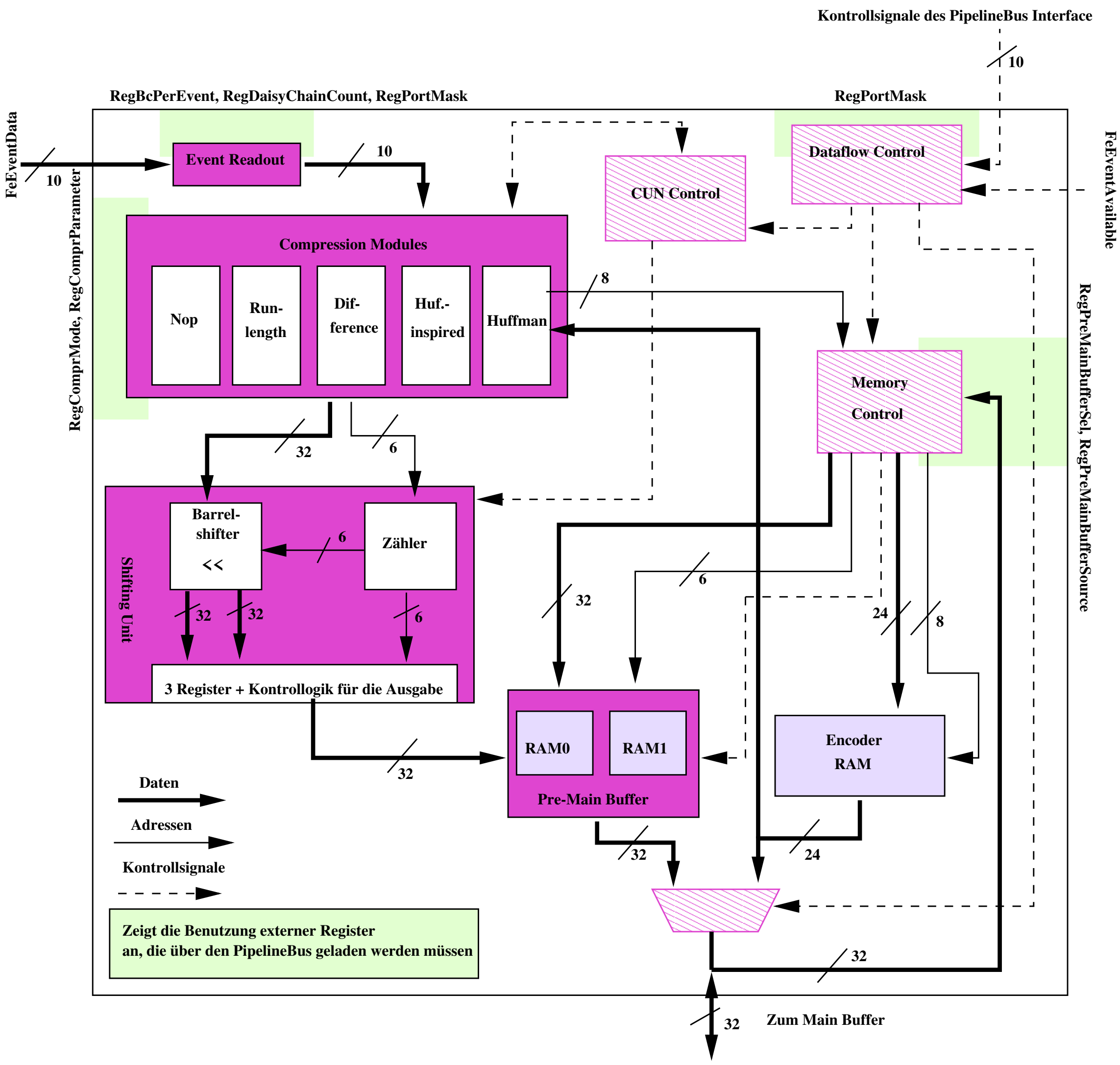

Abbildung 5.4: Blockdiagramm der Kompressionseinheit

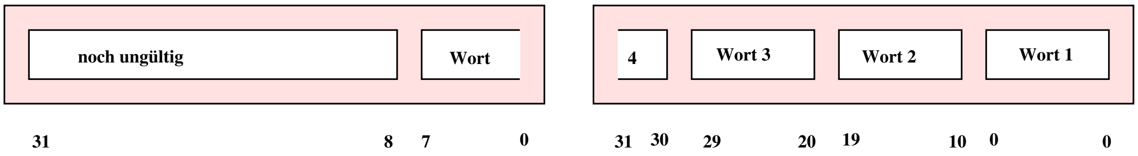

Abbildung 5.5: Verdeutlichung der Formatierung eines 32 bit Wortes

der Algorithmen benutzt wird. Die einzelnen Module erhalten die ausgelesenen Datenworte aus dem Event Buffer zur Verarbeitung. Die Ausgangssignale des aktiven Moduls werden über einen Multiplexer an die Shifting Unit und die CUN Control gegeben. Je nach Komplexität des Algorithmus erfolgt die Codierung mit keiner (Nop-Modus) bis zwei (Huffman-Inspired und Difference Modus) Pipelinestufen. Dadurch ist garantiert, daß in jedem Fall nach zwei Taktzyklen das nächste Datenwort aus dem Event Readout verarbeitet werden kann.

Der Compression Modules Block enthält außerdem noch einen 10 bit Subtrahierer, der für verschiedene Funktionen genutzt wird. Zum einen kann im Run-Length Modus ein Energie-Cut gewählt werden. Ist der aktuelle Energiewert kleiner als der Abschneidewert, so wird die Energie auf Null gesetzt. Diese Vergleichsoperation wird über das Vorzeichenbit des Subtrahierers realisiert. Beim Huffman-Inspired Coding wird der Subtrahierer genutzt, um die Differenz von aktuellem Wert und der Referenzenergie zu berechnen. Für das Difference Coding berechnet er die Differenz zweier aufeinanderfolgender Energiewerte.

Beim Huffman Coding wird zusätzlich aus dem ausgelesenen Energiewert eine 8 bit Adresse und das Memory Enable[6] für den Encoder RAM generiert. Aus dem zurückgelieferten 24 bit Wort läßt sich der Huffman Code und seine Länge berechnen (genaueres siehe 5.4).

Shifting Unit

Die Shifting Unit ist dafür zuständig, den von den Compression Modules erzeugten Code an die richtige Stelle in einem 32 bit Wort zu setzen, das, sobald es gültig ist, in den Pre-Main Buffer geschrieben wird. Werden zum Beispiel vier 10 bit lange Codewörter erzeugt, so bilden die ersten drei zusammen mit den 2 LSBs des vierten Codes ein gültiges Datenwort. Die oberen acht bit des vierten Codes sind dann die 8 LSBs eines neuen Datenwortes, das, bevor es in den Pre-Main Buffer geschrieben werden kann, erst noch vollständig aufgefüllt werden muß (siehe auch 5.5).

Für ihre Arbeit benötigt die Shifting Unit eine Reihe von Informationen und Daten, die von den Compression Modules zur Verfügung gestellt werden. Einige Signale gelangen jedoch erst nach einem Umweg über die CUN Control in die Shifting Unit. Hier wird zum Beispiel auf den Sonderfall des letzten Energiewertes eines Events reagiert. Die Shifting Unit hat einen 32 bit breiten Eingang für die Codewörter. Da diese jedoch verschieden lang sind, muß zusätzlich die Länge des Wortes bekannt sein. Sie gibt an, wieviele LSBs

[6]Mit dem Memory Enable wird die Auslese des Datenwortes gestartet, das sich an der Position befindet, die am Adresseingang spezifiziert ist.

des 32 bit Wortes gültig sind. Die restlichen bits sind auf Null gesetzt. Außerdem wird noch ein Signal benötigt, das anzeigt, wann ein gültiges Codewort am Eingang der Shifting Unit anliegt.

Für die Formatierung der Daten werden ein Barrel Shifter, ein Zähler und drei 32 bit Register verwendet. Die Bedeutung der einzelnen Komponenten und die Arbeitsweise der Shifting Unit werden in Abschnitt 5.3 erläutert. Die Shifting Unit stellt an ihrem Ausgang das fertig formatierte 32 bit Datenwort, wie es in den Pre-Main Buffer geschrieben wird, zur Verfügung. Außerdem wird signalisiert, wann es in den Speicher geschrieben werden kann.

Pre-Main Buffer

Der Pre-Main Buffer besteht aus den zwei RAMs (RAM0 und RAM1), zwei Adressregistern und einem Kontrollblock. Es kann gewählt werden, welcher der beiden RAMs beschrieben werden soll, der andere befindet sich dann automatisch im Lesemodus. Das eine Register speichert die aktuelle Schreibposition, das andere die Leseposition. Dadurch ist es möglich einen Speicher zu beschreiben und den anderen auszulesen.

Für die Adressierung des Pre-Main Buffers gibt es zwei Modi. Im normalen Betrieb wird bei jedem Schreib- oder Lesezugriff die entsprechende Adresse automatisch inkrementiert. Wird die oberste Adresse erreicht, bleibt der Adresszähler stehen. Dieser Modus hat den Vorteil, daß er das Beschreiben und Auslesen des Pre-Main Buffers bei der Kompression wesentlich erleichtert. Für den Testbetrieb ist es auch möglich in einen wahlfreien Modus umzuschalten, in dem an beliebigen Adressen gelesen und geschrieben werden kann.

Die Kontrollblöcke

Die Kontrollblöcke regeln den Datenfluß durch die Kompressionseinheit für die verschiedenen Kompressions- und Testmodi. Ihre genaue Funktionsweise wird nicht erörtert, da dies langweilig ist und einer Erklärung des gesamten Verilog Sourcecodes gleich käme. Statt dessen wird kurz ihre Bedeutung für die Kompressionseinheit vorgestellt.

Die Data Flow Control regelt den globalen Datenfluß und reagiert auf Signale des PipelineBus Interface. Sie ist sozusagen die Oberkontrollinstanz und steuert sämtliche anderen Kontrollmodule. Für einen korrekten Ablauf der Kompression, deren Beginn und Ende, ist die CUN Control zuständig, während mit der Memory Control der Zugriff auf die verschiedenen Speicher geregelt wird. Der Multiplexer schließlich sorgt dafür, daß sowohl der Pre-Main Buffer, als auch der Encoder RAM, über denselben Ausgang der Kompressionseinheit zum PipelineBus ausgelesen werden kann.

Die Speicher

RAM0 und RAM1 sind zwei identische Speicherblöcke, die jeweils 48 Datenworte mit einer Länge von 32 bit aufnehmen können. Sie verfügen über einen eingebauten Selbsttest (BIST) und *Latches*[7] als Ausgänge. Dadurch bleibt das ausgelesene Datum bis zum nächsten Lesezugriff stabil. Sie belegen eine Fläche von jeweils 0.57 mm^2 und haben eine

[7] Pegelgesteuerte Speicherbausteine (siehe [27].

mittlere Zugriffszeit von 4.4 ns. Der Encoder RAM umfaßt 256 Datenworte zu 24 bit und verfügt ebenfalls über BIST und Latches am Datenausgang. Er benötigt eine Fläche von 1.84 mm^2. Die Zugriffszeit beträgt typischerweise 5.8 ns. Diese Zeiten sind dem Datenfile entnommen, das das Programm zur Erzeugung der Speicherblöcke ausgibt.

5.3 Formatierung der Daten

Die Formatierung der Codewörter in einem 32 bit Wort wird in diesem Abschnitt beschrieben. Bewerkstelligt wird dies von der Shifting Unit. Sie ist das Herzstück der gesamten Datenkompression, da alle Algorithmen auf die Formatierung angewiesen sind. Die Schnittstelle der Kompressionsmodule zur Shifting Unit besteht aus einem 32 bit breiten Wort, das den Code enthält, der 6 bit breiten Codewortlänge und einigen Steuerleitungen.

Wichtigste Funktionseinheit der Shifting Unit ist der *Barrel Shifter*. Ein Barrel Shifter dient dazu, ein Datenwort um eine beliebige Anzahl von bits zu verschieben. Der Unterschied zu einem Schieberegister besteht darin, daß der Barrel Shifter aus rein kombinatorischer Logik besteht. Das bedeutet, daß die Zeit, die für die Schiebeoperation benötigt wird, nur von den Gatterlaufzeizten abhängt. Die Simulation hat typische Durchlaufzeiten von einem viertel Taktzyklus, also etwa 7 ns ergeben. Beim Schieberegister wird hingegen pro Schiebeoperation um ein bit ein Taktzyklus benötigt. Der hier implementierte Barrel Shifter hat einen 32 bit breiten Eingang und einen 64 bit breiten Ausgang, der in zwei jeweils 32 bit breite Ausgänge unterteilt ist (*DataHigh* und *DataLow*). Über eine 6 bit breite Steuerleitung (*ShiftBits*) wird festgelegt, um wieviele bits das Eingangswort nach links verschoben werden soll. Für jedes geschobene bit rückt eine Null nach. Es wird also nicht zyklisch geschoben. Eine Schiebeoperation mit einem 7 bit breiten Datenwort, das um 28 bits nach links geschoben wird, ist in Abbildung 5.6 zu sehen.

Um aus dem Output des Barrel Shifters ein Datenwort zusammenzusetzen werden drei 32 bit breite Register verwendet (siehe Abbildung 5.7). Eines ist das aktive, eines das Überlaufregister und eines wird auf Null gesetzt (Resetregister). Der Ausgang jedes Registers ist über eine ODER-Verknüpfung mit der Datenleitung auf den Eingang zurückgekoppelt. Auf der Datenleitung des aktiven Registers liegt der DataLow Ausgang des Barrel Shifters. Hier wird also das Datenwort für den Pre-Main Buffer aus den Codewörtern zusammengesetzt. Das Überlaufregister ist auf Null gesetzt, seine Datenleitung ist mit DataHigh des Barrel Shifters verbunden. In dieses Register wird der Überlauf geschrieben, der entsteht, wenn ein Datenwort so weit geschoben wird, daß ein Teil über der 32 bit Grenze liegt. Ist dies der Fall, so sind alle 32 bit des aktiven Registers mit Daten belegt, das Wort kann in den Pre-Main Buffer geschrieben werden. Nun wird das Überlaufregister zum aktiven Register, das Resetregister wird das neue Überlaufregister und das aktive Register wird zum Resetregister. Das Resetregister ist nötig, damit das Überlaufregister stets auf Null gesetzt ist.

Die Ansteuerung des Barrel Shifters und der Register geschieht über den sogenannten *Shift Counter*. Er enthält zwei Register. In einem werden die Codewortlängen der aktuellen Codes aufaddiert. Wird der Wert des Registers größer als 32, ist also ein 32 bit Wort fertig formatiert, so wird das Signal *Overflow* (Überlauf) gesetzt. Im anderen Register wird die Anzahl der bits gespeichert, um die der Barrel Shifter nach links schieben muß

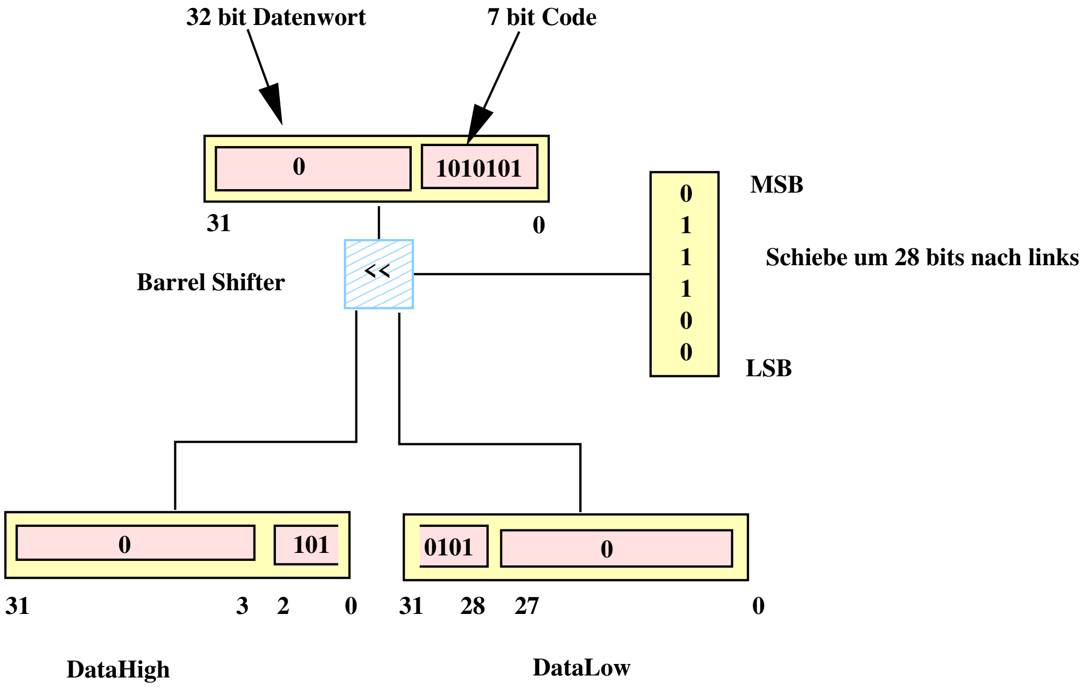

Abbildung 5.6: Veranschaulichung einer Schiebeoperation

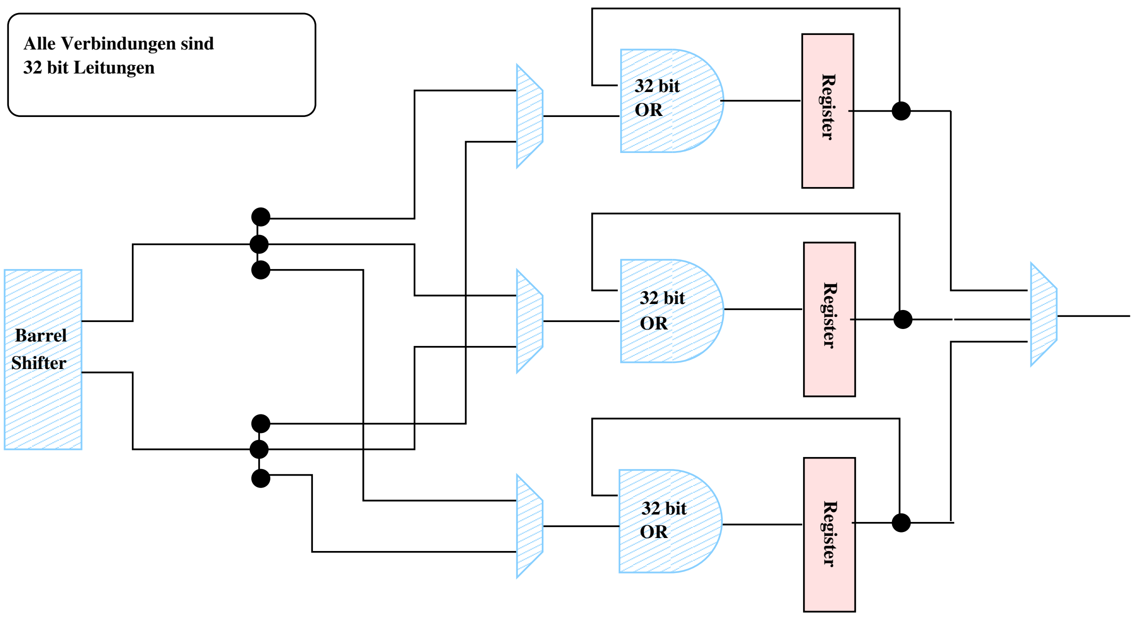

Abbildung 5.7: Organisation der Barrel Shifter Auslese

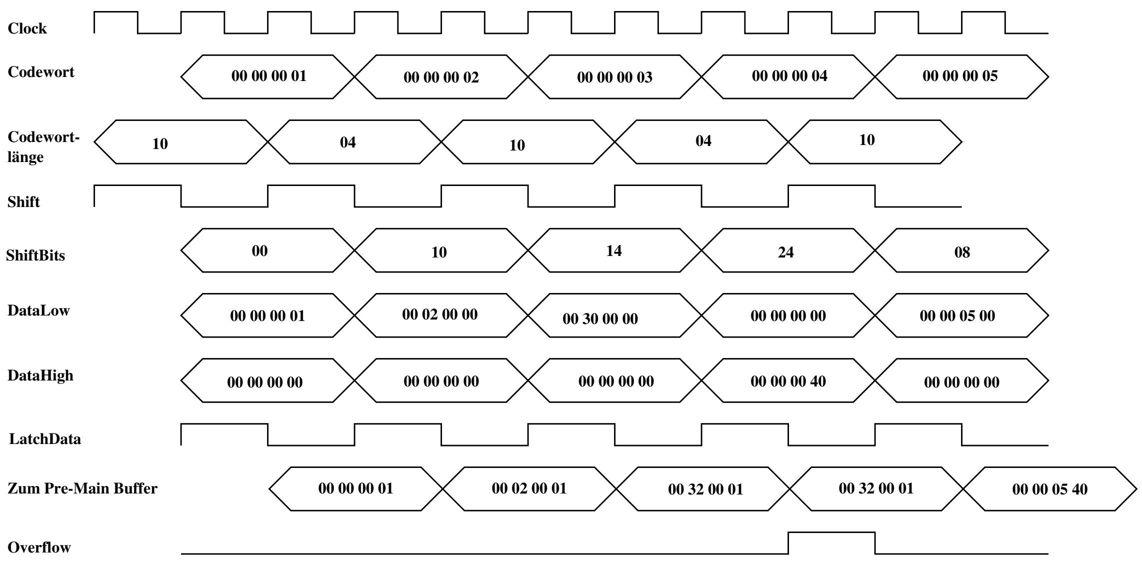

Abbildung 5.8: Timing beim Formatieren der Daten. Alle Werte der Datenbusse sind als Hexadezimalzahlen zu verstehen. Damit entspricht eine Ziffer 4 bit.

(*Shift Bits*). Vor jeder neuen Addition von Codewortlängen im ersten Register wird einfach die alte Summe im zweiten Register gespeichert. Diese entspricht gerade der Anzahl der zu Schiebenden bits.

Der zeitliche Ablauf (*Timing*) beim Formatieren eines Datenwortes ist in Abbildung 5.8 zu sehen. Sowohl Codewort als auch Codewortlänge müssen für zwei Taktzyklen gültig sein. Das Signal *Shift* betätigt den Shift Counter, der ShiftBits berechnet. Im Barrel Shifter wird das Codewort um ShiftBits nach links geschoben, was höchstens einen Taktzyklus beanspruchen darf, da mit *LatchData* die Ausgänge des Barrel Shifters in das aktive und das Überlaufregister gespeichert werden. Ist ein Datenwort voll, so wird durch setzen von Overflow seine Speicherung im Pre-Main Buffer veranlaßt (siehe auch Abschnitt 5.5).

Die Formatierung der Daten wurde auf diese Weise realisiert, um absolute Synchronizität der Schaltung zu gewährleisten (siehe Abschnitt 5.1.2). Ohne Forderung nach Synchronizität hätte die Shifting Unit auch mit zwei 32 bit Registern realisiert werden können. Das Resetregister ist bei der gewählten Vorgehensweise notwendig, da in einer synchronen Schaltung für ein Zurücksetzen des Registers ein Taktzyklus benötigt wird. Ohne das Reset Register müßte das aktive Register zurückgesetzt werden, während sein Inhalt noch in den Pre-Main Buffer geschrieben wird.

5.4 Die Kompressionsmodule

Alle Kompressionsmodule haben nach außen hin das gleiche Interface und stellen Codewort, Codewortlänge und Steuersignale für die Shifting Unit zur Verfügung. Dabei müssen sie sich an das in Abschnitt 5.3 dargestellte Timing halten. Die der Implementation zu-

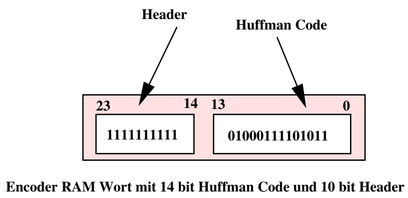

Abbildung 5.9: Aufbau eines Encoder RAM Wortes

grundeliegenden Ideen werden in diesem Abschnitt erläutert.

Im NoOperation (Nop) Modus passiert mit den ausgelesenen Daten nichts. Es wird lediglich die Datenwortlänge über einen Multiplexer zwischen 8 bit und 10 bit umgeschaltet, je nachdem welche Wortlänge gewählt wurde. Die anderen Kompressionsmodule sind etwas komplexer.

5.4.1 Huffman Coding

Wichtigster Bestandteil des Huffman Coding ist der Encoder RAM. Zu jedem möglichen Energiewert ist ein Pseudocode gespeichert, der den eigentlichen Huffman Code und seine Länge enthält. Die Energiewerte dienen als Adresse für den Speicher. Man erhält dann am Datenausgang den Pseudocode. Der Encoder RAM wird also als Look-Up Table benutzt. Er umfaßt lediglich 256 Datenworte. Das bedeutet, daß ein echtes Huffman Coding für 10 bit Daten nicht möglich ist. Es ist jedoch auch nicht nötig, da beim Übergang von 8 bit auf 10 bit die Entropie der Kalorimeterdaten um fast zwei bit ansteigt (siehe 4.1). Man erhält also beinahe dasselbe Ergebnis wie beim echten Huffman Coding, wenn nur die 8 MSBs eines 10 bit Wortes mit den Codes aus dem Encoder RAM codiert werden, und die zwei LSBs einfach unverändert an den Huffman Code angehängt werden. Damit ist auch kein Wechsel der Codetabelle beim Übergang von 8 bit auf 10 bit Daten notwendig. Wie man sich leicht klar macht, entsteht auch durch diese Art der Codierung ein Prefix Code (siehe auch Anhang A).

Im Encoder RAM werden die Huffman Codes so gespeichert, daß man sowohl den eigentlichen Code, als auch die Codelänge erhält. Dazu werden die vom Codewort nicht belegten bits des Encoder RAM Wortes mit dem invertierten MSB des Codewortes aufgefüllt (siehe Abbildung 5.9). Es können also maximal Codes gespeichert werden, die ein bit weniger benötigen, als in ein Encoder RAM Wort passen. Die maximale Länge eines Huffman Codes beträgt somit $24 - 1 = 23$ bit. Um Huffman Code und Codelänge zu bestimmen ist es nur noch nötig, die Stelle zu bestimmen, an der sich, beginnend vom MSB, zum ersten Mal ein bit ändert. Realisiert wird dies über eine XOR-Schaltung, die alle MSBs bis zum ersten geänderten bit auf logisch eins setzt, dann folgt eine Null und der Rest ist unbestimmt. Mit dieser Information kann die Codelänge bestimmt werden. Hat man die Codelänge, lassen sich auch die führenden ungültigen bits auf Null setzten, man hat den Huffman Code zurückerhalten.

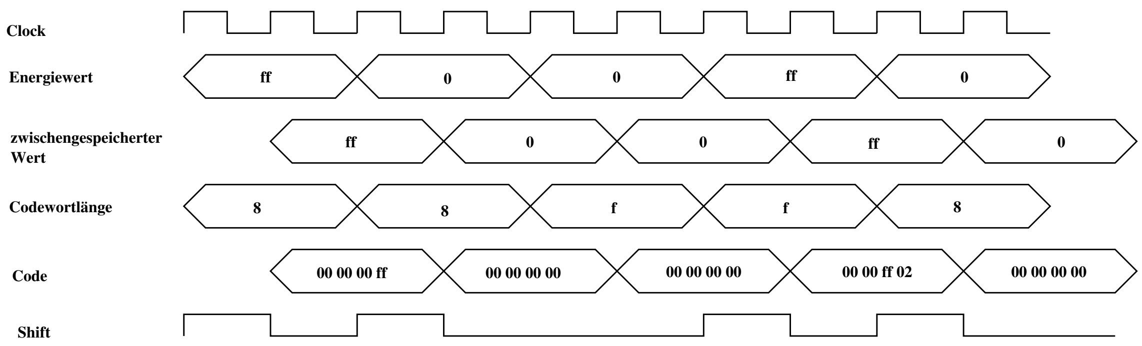

Abbildung 5.10: Timing beim Run-Length Encoding. Alle Zahlenwerte sind Hexadezimal

5.4.2 Run-Length Encoding

Der zeitliche Ablauf des Run-Length Encoding ist in Abbildung 5.10 zu sehen. Durch Einführung einer Pipelinestufe wird der Energiewert solange gespeichert, bis der nachfolgende Wert bekannt ist. Es gibt einen Überlapp der beiden Werte von einem Taktzyklus. Solange keine Null erkannt wird, werden einfach die Energiewerte und deren Länge (8 bit oder 10 bit) an die Shifting Unit übergeben. Auch beim ersten Auftreten einer Null wird so verfahren. Die nun folgenden Nullen werden nur gezählt, es werden keine Datenwörter an die Shifting Unit gegeben. Daher wird auch das Shiftsignal nicht mehr gesetzt. Sobald nun ein anderer Wert als Null folgt, wird an die Shifting Unit ein Wort mit doppelter Länge übergeben (16 bit oder 20 bit), in dem die Anzahl der Nullen und der neue Energiewert stehen. Dadurch wird erreicht, daß beim Schreiben des Zählers keine zusätzliche Zeit benötigt wird. Auch wenn mehr Nullen gezählt werden, als mit den zur Verfügung stehenden bits codiert werden können (also 256 oder 1024), werden Anzahl der Nullen und nachfolgend eine neue Null in einem Doppelwort an die Shifting Unit übergeben, so daß die Zählung wieder bei eins anfangen kann.

5.4.3 Huffman-Inspired und Difference Coding

Um eine freie Wahl von Datenwortlänge l_{DW} und Länge der kurzen Codes l_{ZM} zu implementieren, würde ein zusätzlicher Barrel Shifter benötigt. Da die besten Werte für diese Parameter aus der Simulation bekannt sind, und um Chipfläche und damit Kosten zu sparen, wurden feste Werte für l_{DW} und l_{ZM} vorgegeben. Außerdem wurde die Energieauflösung für beide Modi auf 10 bit festgelegt. Bei 10 bit erhält man mit $l_{DW} = 12$ bit und $l_{ZM} = 4$ bit die besten Kompressionsfaktoren (siehe 4.2). Mit diesen Parametern wurden die beiden Algorithmen daher auch auf dem RemASIC realisiert.

Huffman-Inspired und Difference Coding unterscheiden sich kaum. Für das Huffman-Inspired Coding wird die Differenz von Referenzenergie E_R und Pedestal P benötigt, für das Difference Coding die Differenz zweier aufeinanderfolgender Energiewerte. Daher wird dasselbe Modul für das Huffman-Inspired und Difference Coding benutzt. Das Difference Coding braucht zusätzlich nur noch ein Register, in dem ein Datenwort aus dem Event Buffer zwischengespeichert wird, bis ein neues ausgelesen ist, und die Differenz gebildet

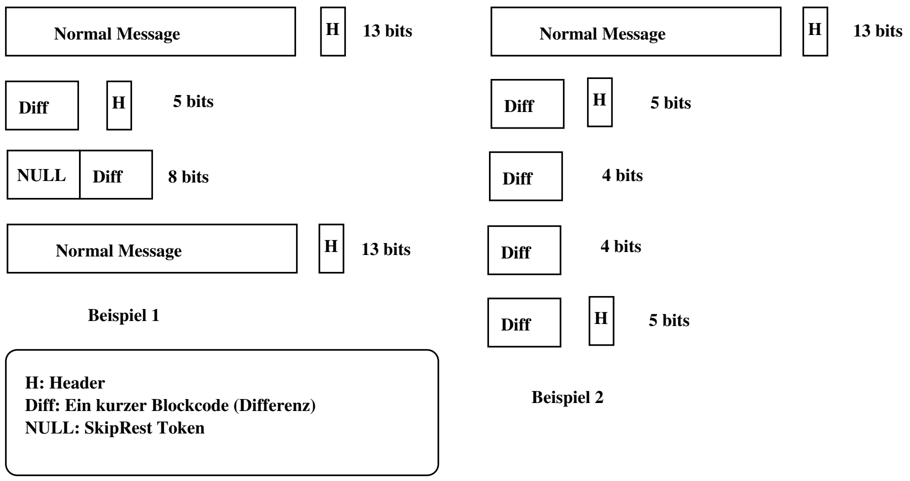

Abbildung 5.11: Codewörter und -längen, wie sie an die Shifting Unit übergeben werden. Die Codewortlänge läßt sich erst bestimmen, wenn der folgende Energiewert bekannt ist.

werden kann. Damit die Kompression des ersten ausgelesenen Energiewertes nicht zu einem Spezialfall wird, wird hier die Differenz mit Null gebildet. Auch der erste Energiewert kann also mit einem kurzen Code codiert werden.

Das Modul, das die eigentliche Codierung übernimmt, erhält den Energiewert aus dem Event Buffer und zusätzlich eine Differenz. Wie diese zustande kommt, interessiert für die Codierung nicht weiter. Für die Realisierung der Codierung kann man sich zwei prinzipielle Möglichkeiten vorstellen. Zum einen könnte ein Datenwort zunächst komplett im Kompressionsmodul zusammengesetzt werden und dann an die Shifting Unit übergeben werden. Diese Methode hat jedoch den Nachteil, daß das Kompressionsmodul Arbeit erledigt, die eigentlich die Shifting Unit übernehmen sollte, nämlich die Formatierung der Daten. Eine andere Methode ergibt sich, wenn man Abbildung 5.11 betrachtet. Jede Zeile entspricht dem Code für einen Energiewert, Header und abschließende Nullen (SkipRest) werden gegebenenfalls mit in den Code einbezogen. Das bedeutet jedoch, daß sich die Codewortlänge erst angeben läßt, wenn auch der nächste Energiewert bekannt ist.

In Abbildung 5.12 ist ein Blockdiagramm zu sehen, das den Ablauf bei der Codierung verdeutlicht. Energiewert und Differenz werden solange zwischengespeichert, bis auch der neue Energiewert und die neue Differenz bekannt sind. Die zwischengespeicherten Werte sind in der Abbildung als *alte Werte* bezeichnet. Aus der alten und der neuen Differenz wird über einen Block mit kombinatorischer Logik ein Signal berechnet, das den Multiplexer MUX1 bedient. Über ihn kann entweder die Differenz (der kurze Code) oder der unveränderte alte Energiewert selektiert werden. Ein zweiter Multiplexer MUX2 sorgt dafür, daß das ausgewählte Datenwort um ein bit nach links verschoben wird, falls ein Headerbit gebraucht wird. Dies ist der Fall, für den ersten kurzen Code eines Datenwortes oder für einen unveränderten Energiewert. Wird von MUX1 ein kurzer Code selektiert, so sind nur die 4 LSBs des 10 bit Wortes belegt, die anderen 6 bits enthalten den Wert Null. An dieser Stelle sollte klar werden, warum gerade die Null signalisiert, daß der Rest des

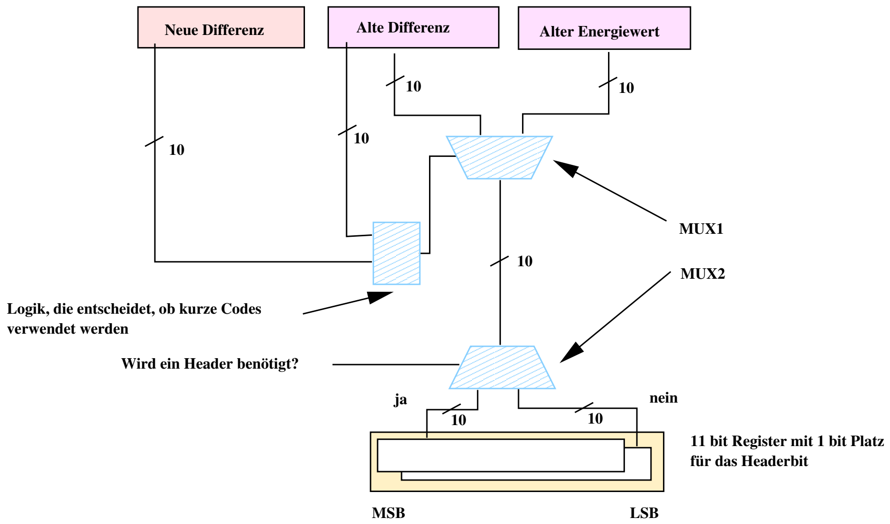

Abbildung 5.12: Blockdiagramm des gemeinsamen Teils von Huffman-Inspired und Difference Coder

Datenwortes übersprungen werden soll. Um eine Null vor ein vorhandenes Codewort zu stellen muß man garnichts tun, außer die Codewortlänge entsprechend zu erhöhen. Den Rest erledigt die Shifting Unit.

5.5 Betrieb der Kompressionseinheit

Bevor die Kompressionseinheit für die Auslese von Daten betrieben wird, muß sichergestellt sein, daß die Register für die Konfiguration mit den gewünschten Werten geladen sind (für die Konfiguration siehe Anhang B). Sollen die Daten mit dem Huffman Algorithmus codiert werden, so muß zusätzlich der Encoder RAM mit der gewünschten Codetabelle geladen werden. Eine Änderung der eingestellten Konfiguration während der Kompression eines Datensatzes ist nicht vorgesehen. Nach der vollständigen Verarbeitung eines Datensatzes können hingegen ohne Probleme neue Konfigurationsdaten geladen werden.

Der Ablauf der Kompression ist vereinfacht dargestellt in Abbildung 5.13 als Zustandsdiagramm zu sehen. Dieses Zustandsdiagramm veranschaulicht die Arbeit von sogenannten *State Machines* (FSMs). Eine State Machine, oder Zustandsmaschine, wechselt ihren Zustand in Abhängigkeit vom augenblicklichen Zustand und externen Signalen. Der aktuelle Zustand wird im *State Register* gespeichert. Dargestellt sind die State Machines von CUN Control und Event Readout. In der Mitte ist eine State Machine mit nur zwei Zuständen zu sehen, die die Kompression symbolisiert.

Gestartet werden CUN Control und Event Readout über das Signal *FeEventAvailable*, das vom FeASIC Interface gesetzt wird, sobald ein Event Buffer mit einem Datensatz gefüllt ist. In der CUN Control wird als erstes das Schreiben des Headerwortes in den Pre-Main Buffer erledigt. Im Event Readout wird zur gleichen Zeit die Auslese aus den Event Buffern gestartet und mit dem ersten ausgelesenen Datum das entsprechende Kompressi-

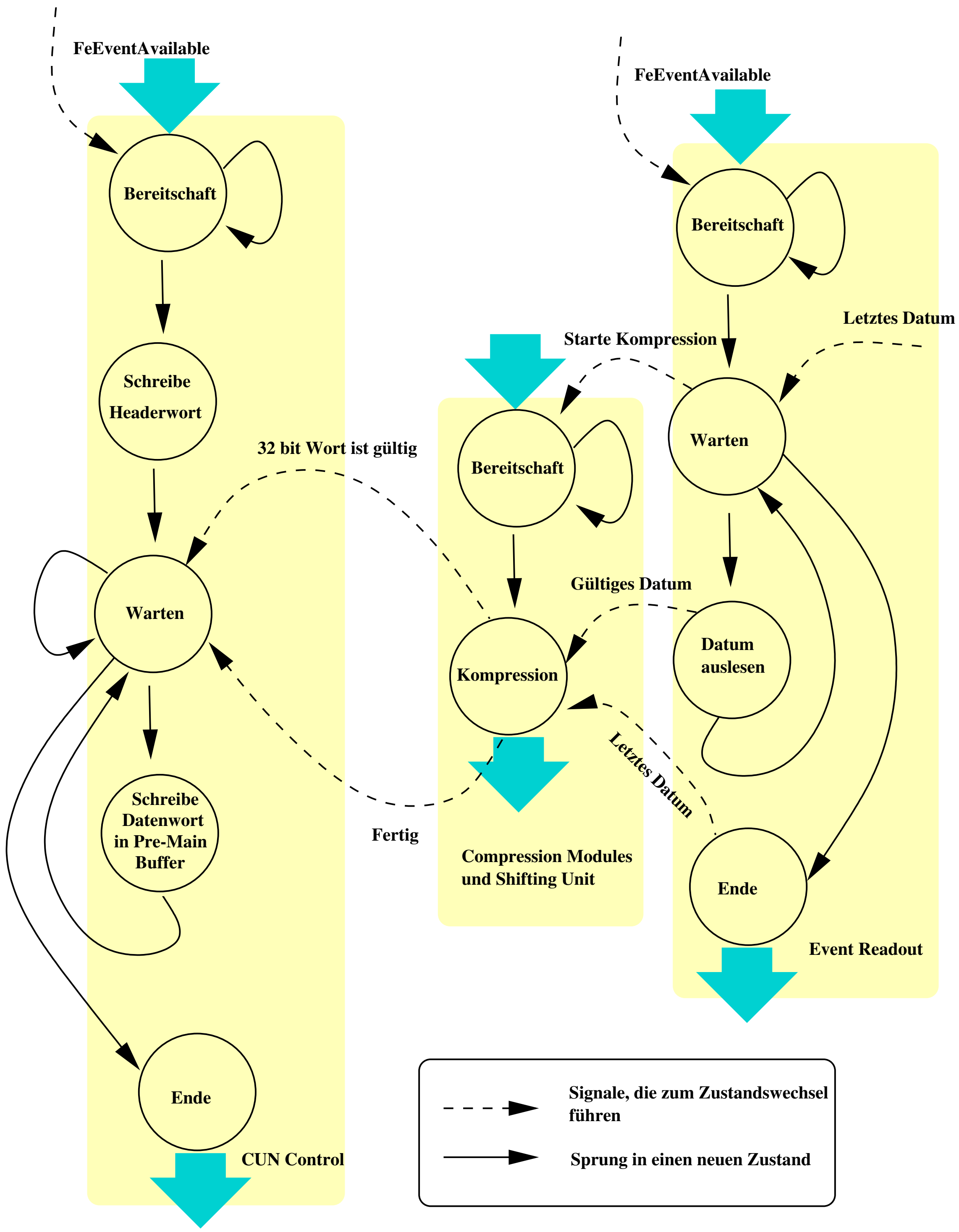

Abbildung 5.13: Zustandsdiagramm für die Auslese und Kompression eines Datensatzes

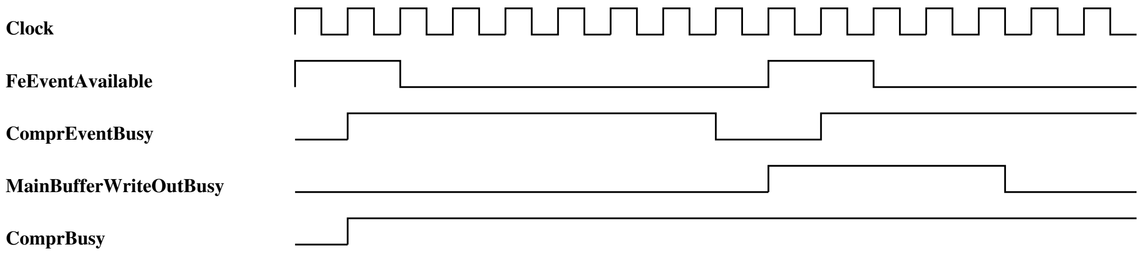

Abbildung 5.14: Signale beim Betrieb der Kompressionseinheit

onsmodul aktiviert. Während der Kompression werden gültige 32 bit Wörter signalisiert, worauf die CUN Control mit einem Zustandswechsel reagiert und das Datenwort in den Pre-Main Buffer geschrieben wird. Die Datenwörter können nicht direkt von der Shifting Unit in den Pre-Main Buffer geschrieben werden, da nur über diesen Zwischenzustand auf den Sonderfall des letzten Datenwortes reagiert werden kann. Tritt nämlich beim letzten Wort ein Überlauf in der Shifting Unit auf, so muß gewartet werden, bis der Überlauf im aktiven Register steht und in den Pre-Main Buffer geschrieben werden kann. Ansonsten ginge der eventuell vorhandene Überlauf verloren. Ist der Event Readout fertig mit der Auslese, so wird dies an das aktive Kompressionsmodul signalisiert, das daraufhin die Verarbeitung beendet und der CUN Control das Ende der Kompression signalisiert. Die dicken Pfeile deuten an, daß die State Machines nach erreichen des Endzustandes wieder in den Ausgangszustand zurückkehren.

Während ihrer Arbeit werden von der Kompressionseinheit zwei Signale generiert, die die Aktivität nach außen signalisieren. Für das FeASIC Interface ist dies *ComprEventBusy*, für das PipelineBus Interface *ComprBusy* (siehe Abbildung 5.14). Solange ComprEventBusy gesetzt ist, wird vom FeASIC Interface kein FeEventAvailable gegeben. ComprEventBusy bleibt auf logisch Eins, solange ein Datensatz komprimiert wird, geht jedoch auf logisch Null bevor der Pre-Main Buffer in den Main Buffer geschrieben wird. Das Schreiben des Datensatzes in den Main Buffer wird in Abbildung 5.14 symbolisiert durch das interne Signal *MainBufferWriteOutBusy*. Dem PipelineBus Interface wird über das setzen von ComprBusy verboten die Speicher der Kompressionseinheit zu konfigurieren oder eine Auslese derselben zu starten. Dieses Signal bleibt solange gesetzt, bis ein Datensatz komplett im Main Buffer abgelegt ist.

Es kann passieren, daß die Kompressionseinheit nicht auf ein gesetztes FeEventAvailable reagiert. Um unvorhersehbares Verhalten zu verhindern geschieht dies in folgenden Fällen:

- Es wurde kein FeASIC Port konfiguriert, also kann auch nicht komprimiert werden.

- Die Kompressionseinheit ist noch beschäftigt.

Ist die Kompressioneinheit noch mit dem Schreiben eines Datensatzes aus dem Pre-Main Buffer in den Main Buffer beschäftigt, während der nächste bereits fertig komprimiert wurde, so bleibt ComprEventBusy weiterhin gesetzt, um einen Datenverlust zu vermeiden.

Diese Situation tritt vor allem dann auf, wenn das Schreiben in den Main Buffer unterbrochen werden muß, weil er voll ist. Das bedeutet allerdings, daß die Auslese stockt. Man verliert dann eventuell einen kompletten Datensatz, für den inzwischen ein Level-1 Accept gegeben wurde. Dafür ist sichergestellt, daß ein ausgelesener Datensatz stets komplett im Main Buffer steht. Es werden also keine bruchstückhaften Datensätze ausgelesen, statt dessen verzichtet man in so einem Fall lieber auf einen anderen Datensatz.

Für den seltenen Fall, daß ein Datensatz sich bei der Kompression so stark aufbläht, daß er nicht mehr in den Pre-Main Buffer paßt, wird die Kompression im NoOperation-Modus wiederholt, und der Datensatz unkomprimiert übertragen. Angezeigt wird dies durch setzen eines bits im Headerwort (siehe auch Anhang B).

Kapitel 6

Erster Test des RemASIC

Ein erster Test des RemASIC sollte die prinzipielle Funktionsfähigkeit des Chips unter Beweis stellen. Hierzu wurde ein einfaches Testboard entwickelt. Die Leistungsfähigkeit des PipelineBus-Systems zusammen mit der Datenkompression läßt sich jedoch erst mit einem Testsystem messen, das zur Zeit von der Elektronikwerkstatt des Instituts für Hochenergiephysik gebaut wird.

6.1 Testaufbau

Für den Test wurde ein HP82000 Chip-Tester verwendet. Es standen 48 Kanäle für die Bedienung von Eingangs- und die Aufnahme von Ausgangssignalen zur Verfügung. Da für einen parallelen Betrieb des PipelineBus Interface alleine 70 Leitungen benötigt würden, mußte für diesen Test auf den seriellen Betrieb ausgewichen werden, in dem über ein Schieberegister die 35 bit eines PipelineBus-Wortes seriell eingelesen werden. Das bedeutet, daß die Taktfrequenz des seriellen Interface das 35-fache der Bus Clock[1] betragen muß.

Der HP82000 bietet 40 Kanäle für einen Betrieb mit 100 MHz und 8 Kanäle, die mit bis zu 400 MHz betrieben werden können. Die verwendete Software zum Erzeugen von Testvektoren für den HP82000 läßt eine parallele Nutzung von 100 MHz und 400 MHz Kanälen nur eingeschränkt zu. Für die 400 MHz Kanäle werden automatisch pro Taktzyklus vier identische Testvektoren erzeugt, so daß auch die 400 MHz Kanäle nur bis maximal 100 MHz betrieben werden können [28]. Damit stehen für den Test also effektiv 48 100 MHz Kanäle zur Verfügung. Daraus ergibt sich eine maximale Taktfrequenz des RemASIC zu:

$$f = \frac{100 \text{ MHz}}{35} = 2.86 \text{ MHz}. \tag{6.1}$$

Ein Test des RemASIC bei der für den Betrieb vorgesehenen Taktfrequenz von 40 MHz ist also erst mit dem in Planung befindlichen Testsystem möglich. Ein Test darf maximal 256000 Testvektoren pro Kanal umfassen. Da dies auch für die 400 MHz Kanäle gilt und für ein PipelineBus-Wort 35 Vektoren benötigt werden, lassen sich Tests mit maximal 1800 Taktzyklen der Bus Clock aufsetzen.

[1] Bus Clock: Systemtakt von RemASIC und PipelineBus.

Eine Skizze des Testaufbaus ist in Abbildung 6.1 zu sehen. Es gibt einen Sockel für einen RemASIC und einen für die Aufnahme eines FeASIC (graue Blöcke). Der FeASIC ist an Port 1 angeschlossen. Port 0 läßt sich über den Tester bedienen. Für Port 2 und 3 sind ebenfalls Verbindungen zum Tester vorgesehen, die jedoch erst genutzt werden können, wenn 16 zusätzliche Kanäle für den Tester zur Verfügung stehen.

6.2 Teststrategie

Für den HP82000 existiert eine Software zum direkten Eingeben und Editieren von Testvektoren. Für den Test des RemASIC ist diese allerdings untauglich, da jedes PipelineBus-Wort serialisiert und bit für bit eingegeben werden müßte. Es wurde statt dessen ein Programm verwendet, das aus den für die Simulation verwendeten STL-Programmen automatisch Testvektoren für den HP82000 erzeugt [28].

Das Verilog-Modul für den RemASIC wird in ein Modul eingebunden, dessen Ein- und Ausgänge genau denen des Testaufbaus entsprechen. Für dieses Modul wird ein Testprogramm in STL entwickelt und mit der Netzliste des RemASIC simuliert. Im nächsten Schritt wird das STL Programm in HP82000 Testercode übersetzt. Dabei können auch die simulierten Ausgangssignale als sogenannter *Expected Output* miteinbezogen werden. Läßt man anschließend den Test auf dem Chip-Tester ablaufen, so kann man sich alle Abweichungen der Ausgänge vom Expected Output anzeigen lassen. Dadurch wird ein komfortables Testen möglich, denn für die Simulation können in Verilog Zusatzmodule geschrieben werden, die zum Beispiel die Geschehnisse auf dem Bus mitprotokollieren. Mit Hilfe dieser Module ist eine einfache Verifikation der Ergebnisse der Simulation möglich. Anschließend muß nur noch die Äquivalenz von Test und Simulation überprüft werden, was Dank des Expected Output aus der Simulation auf Knopfdruck erfolgt.

In STL wird jeder Testvektor mit einem *xv* Befehl erzeugt. Er hat das Format xv(input1 input2 ...). Es werden also einfach die Werte sämtlicher Eingangssignale hintereinandergereiht. Auf diese Weise lassen sich noch keine aufwendigen Tests programmieren. In STL kann jedoch der gesamte Sprachumfang von Skill, einer Lisp ähnlichen Sprache, verwendet werden. Damit lassen sich dann ganze Folgen von xv Befehlen zu einem Unterprogramm zusammenfassen und mit einem Befehl aufrufen. So konnte für den RemASIC eine Testumgebung entwickelt werden, die eine einfache Formulierung der Tests ermöglicht. Ein Beispielprogramm für einen Test der Kompressionseinheit ist in Abbildung 6.2 zu sehen.

Ein Test kann nicht mehr als 1800 Taktzyklen der Bus Clock umfassen (siehe 6.1). Alleine zum Konfigurieren des gesamten Encoder RAMs würden jedoch über 3000 Taktzyklen benötigt. Die Tests mußten sich daher darauf beschränken jeweils ein wohl definiertes Teilstück der gesamten Funktionalität zu überprüfen. Das Huffman Coding beispielsweise kann auch mit einem teilweise konfigurierten Encoder RAM getestet werden, wenn bei der Auslese nur Energiewerte verwendet werden, für die ein Code gespeichert ist. Ein Test, der die Gesamtfunktionalität auf einmal verifiziert wäre zwar wünschenswert, ist jedoch mit der eingeschränkten Zahl der Testvektoren nicht zu realisieren.

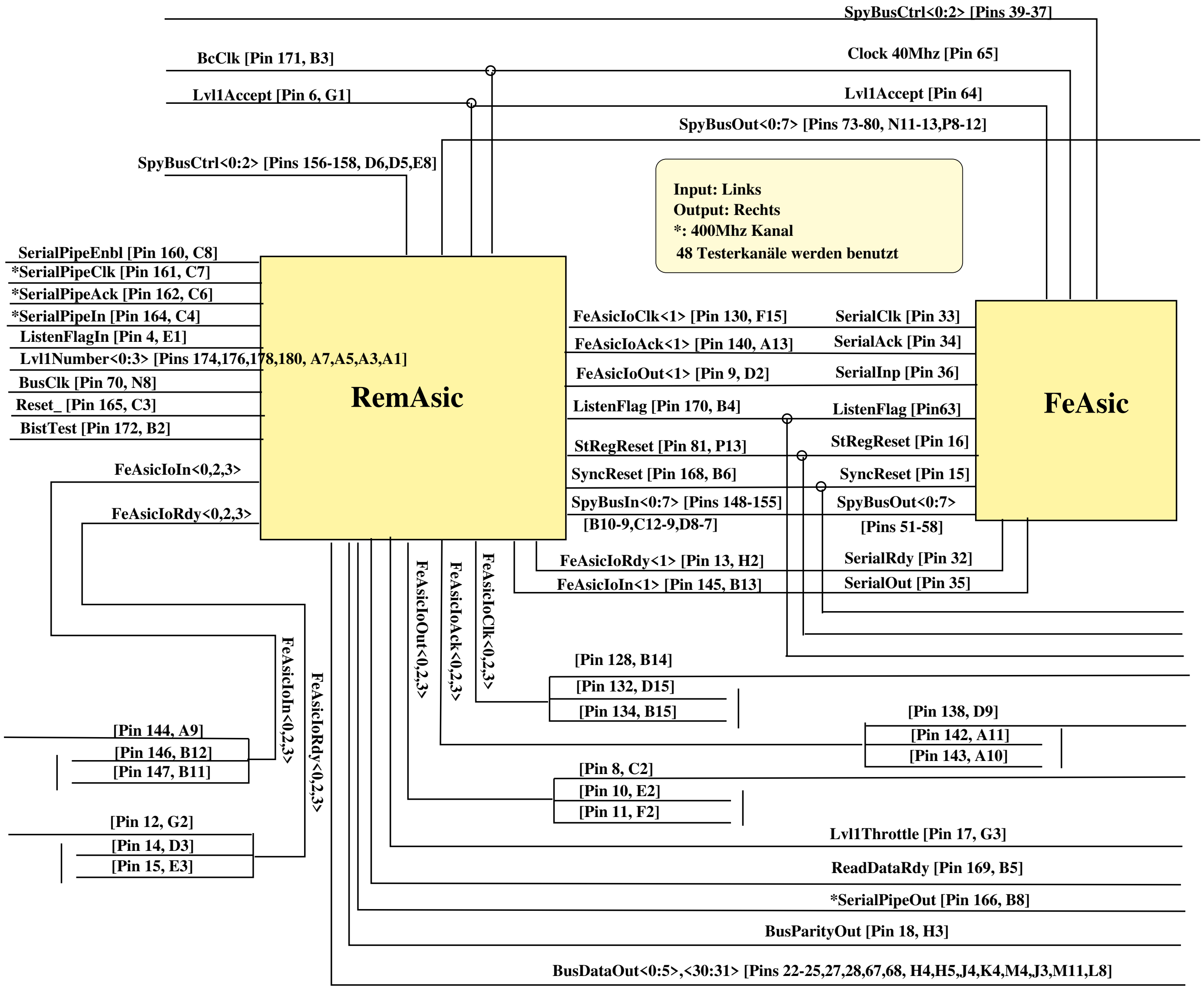

Abbildung 6.1: Verschaltung von RemASIC und FeASIC im Testaufbau

```
;--------------------Konfiguration des RemASIC---------------
SpyBusCtrl = 3

DisableSerAck    ;---Reset--
Reset 1 1 1
Reset 1 1 1
EnableSerAck

Address = 0         ;--Laden der PipelineBus Adresse
LoadNodeAddress 4
Address = 4

StartInput          ;--Laden des Event Buffers mit Daten
ConfigureEventBufferFileCons 3 "./ramdata/rem_evram.dat.old2"

;-----------Konfiguration der Kontrollregister für------------
;-------------die Auslese im Difference Modus-----------------
StartInput
BeginOfData RD_ControlRegister
  PortMask            = 0x01
  DaisyChainCount     = 0x00
  BcPerEvent          = 0x07
  ComprMode           = 0x03
  ReadoutMode         = 0x3
  SelectFeWord        = 0
  EventBufferSource   = 1
  PreMainBufferSource = 0
  PreMainBufferSelect = 0
  StopModeEvents      = 0x0
  ComprParameter      = 0x2
  ListenFlag          = 0
  EnableExtListen     = 0
  ConfigureCtrlRegs
EndOfData
BusIdle 4*9+20

StartReadout     ;--Start der Auslese
BusIdle 50
StopReadout      ;--Stop der Auslese

GetEventCounter  ;--Auslesen des komprimierten Datensatzes über
FlushMainBuffer  ;--den PipelineBus
BusIdle 15
```

Abbildung 6.2: Ein einfaches STL-Programm für den Test des Difference Modus

6.3 Ergebnisse

Acht RemASICs standen für einen Test zur Verfügung. Drei von ihnen wiesen Fehler auf. Im einzelnen sind dies:

- Kurzschluß der Stromversorgung
- Fehler in den Compiled Megacells (Main Buffer)
- Fehler in den Standardzellen (Event Readout der Kompressionseinheit)

Der Kurzschluß wurde festgestellt, da der Chip bei einem Test keine Reaktion zeigte und bei einer anschließenden Messung einen stark erhöhten Stromverbrauch aufwies. Der Fehler im Main Buffer wurde über den BIST-Test gefunden, der keine korrekten Ergebnisse lieferte. Ein nachträglich durchgeführter Test zeigte dann auch, daß Daten, die durch den Main Buffer liefen, verändert wurden. Der Fehler in den Standardzellen äußerte sich über ein falsches Ergebnis bei der Kompression von Testdaten. Er konnte über den SpyBus als Standardzellenfehler im Event Readout identifiziert werden. Ein Flipflop eines Adresszählers schaltet von logisch Null auf logisch Eins und bleibt in diesem Zustand bis zum nächsten Reset. Eine Änderung des Zustandes über das Speichern einer Null ist nicht möglich. Dadurch werden Daten an falschen Adressen aus dem Event Buffer ausgelesen.

Die restlichen Chips bestanden sämtliche durchgeführten Tests. Ein Test gilt als bestanden, wenn keine Abweichungen zwischen den tatsächlich gemessenen und den von der Simulation her erwarteten Ausgangssignalen bestehen. Ein Bild der Testsoftware mit einem bestandenen Test ist in Abbildung 6.3 zu sehen. Neben einem Test von PipelineBus und FeASIC Interface wurden für sämtliche Speicher die BIST-Test durchgeführt.

Für die Kompressionseinheit wurden sämtliche Kompressionsmodi und die Funktionsfähigkeit des Event Readout exemplarisch getestet. Für das Huffman Coding wurden die Zahlen von von 0 bis 15 in aufsteigender Reihenfolge komprimiert. Dadurch war es möglich nur die ersten 16 Adressen des Encoder RAMs mit Codewörtern zu belegen und die Anzahl der Testvektoren klein zu halten. Beim Huffman Coding treten keine Sonderfälle auf, die vermuten lassen könnten, daß dieser Test zur Verifikation der Funktionsfähigkeit nicht ausreicht.

Bei den anderen Codierungsverfahren mußte anders vorgegangen werden. Je nach Aussehen und Reihenfolge der Daten wird die Funktionalität der Module mehr oder weniger genutzt. Wird beispielsweise Run-Length Encoding mit einem Datensatz ohne Nullen durchgeführt, so bedeutet eine korrekte Codierung noch nicht, daß auch ein Run-Length Encoding mit Nullen erfolgreich ist. Daher wurde hier der Datensatz 1-255-2-1-255-2 verwendet. Indem man die Referenzenergie E_R für das Huffman-Inspired Coding und den Energie-Cut für das Run-Length Encoding auf zwei einstellt, erhält man einen Test sämtlicher Sonderfälle diese Codierungsarten. Auch beim Difference Coding ist mit diesem Datensatz ein Test der gesamten Funktionalität sichergestellt.

Für den Event Readout wurde durch die Verwendung einer Konfiguration mit Adressen in nicht konsekutiver Reihenfolge die korrekte Funktion von Adresssprüngen getestet. Ein Fehler in der Kompressionseinheit wurde bei den Tests allerdings entdeckt. Er ist jedoch nicht auf eine Abweichung zwischen Simulation und echtem Chip zurückzuführen, sondern

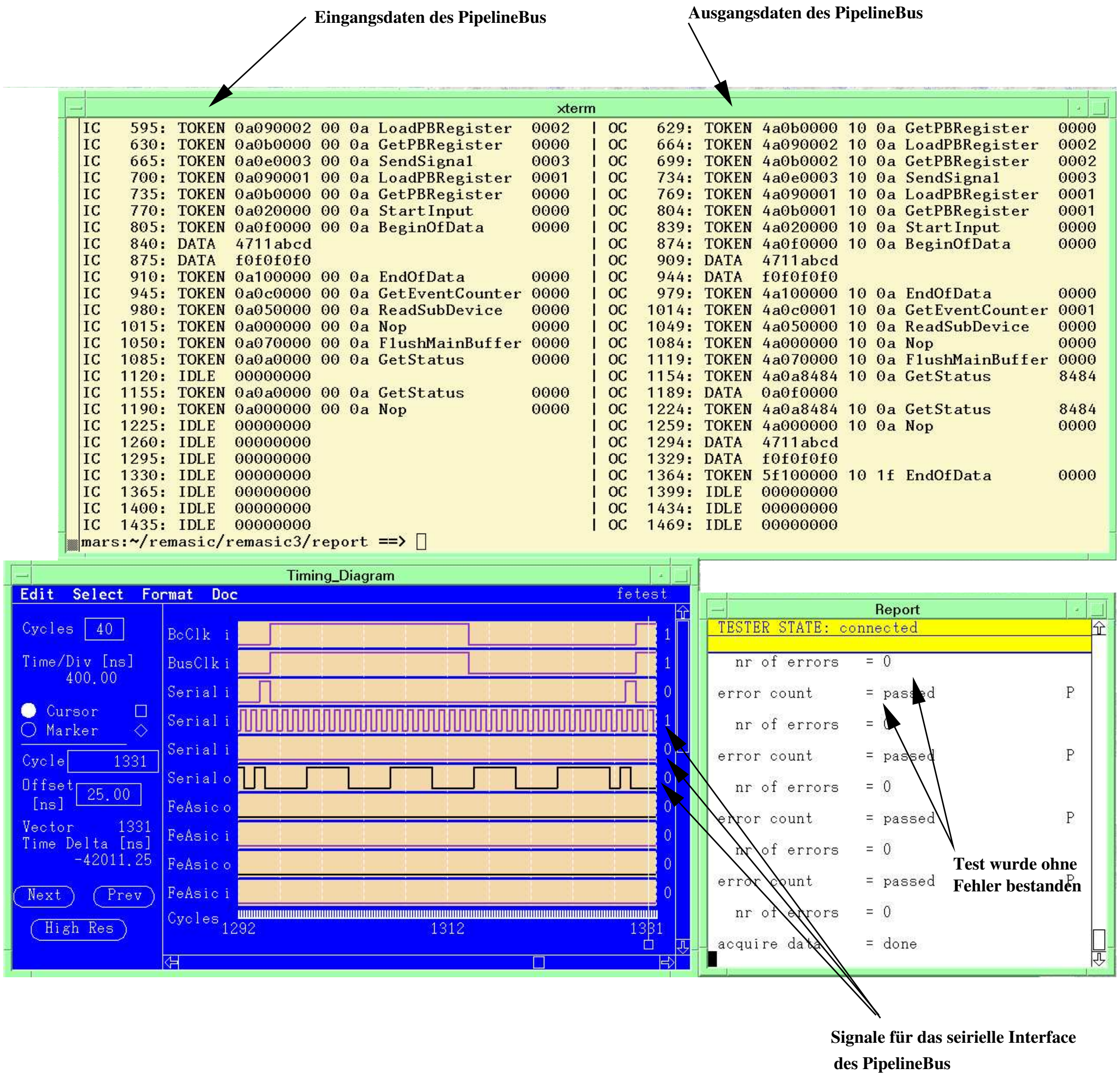

Abbildung 6.3: Abbildung der Testsoftware mit bestandenem Test

ist in einem Fehler im Verilogcode begründet. Durch diesen Fehler versagt die erneute Kompression im NoOperation-Modus, falls der Datensatz nicht in den Pre-Main Buffer paßt. Dieser Fehler ist ärgerlich, stellt aber kein Hindernis beim Einsatz des RemASIC dar, da selbst im ungünstigsten Fall (es werden nur 23 bit Huffmancodes produziert) 64 Datenwörter bearbeitet werden können. Die Auslese von acht Kanälen bei acht Bunch-Crossings pro Event ist also selbst dann noch möglich.

Zusammenfassung und Ausblick

Datenkompression ist eine Möglichkeit die benötigte Bandbreite für die Auslese eines Hochenergiephysik-Experimentes zu reduzieren. Dadurch wird die Verwendung kostengünstiger Bussysteme möglich. Außerdem werden Kosten für Massenspeicher-Medien gespart. Mit der Implementation einer Datenkompressionseinheit auf dem Readout Merger ASIC wurde die prinzipielle Machbarkeit der Kombination aus Datenkompression und PipelineBus gezeigt.

Vor der Implementation wurden alle Algorithmen mit Testdaten simuliert. Demnach können Kompressionsfaktoren von 2-3 erreicht werden, wenn die Daten in vollem Umfang samt Rauschen ausgelesen werden sollen. Akzeptiert man einen Energie-Cut von 1 GeV, so läßt sich das Datenvolumen um einen Faktor 13 reduzieren. Durch die Implementation von vier verschiedenen Algorithmen besteht die Möglichkeit die Vor- und Nachteile der einzelnen Verfahren in einem Praxistest zu untersuchen.

In einem ersten Test, der allerdings nur mit knapp 3 MHz durchgeführt werden konnte, wurde gezeigt, daß der RemASIC dieselbe Funktionalität aufweist, wie in der Simulation. Die Konfiguration der Kontrollregister und eine Kompression einiger ausgewählter Beispieldaten verlief erfolgreich. Ein Test mit der geplanten Taktfrequenz von 40 MHz und eine Verifikation der vorhergesagten Kompressionsraten mit Datensätzen aus Teststrahl-Experimenten steht noch aus. Auch eine stabile Funktion bei der Verarbeitung umfangreicher Datensätze sollte untersucht werden.

Für einen Einsatz im finalen System sind sicherlich noch eine ganze Reihe von Verbesserungen des Kompressionskonzeptes möglich. So sollte man sich für die Verwendung eines Algorithmus entscheiden und die Hardware daraufhin optimieren. Die implementierten Verfahren stellen eine Auswahl verschiedener Möglichkeiten dar, es sollte jedoch nach weiteren Alternativen gesucht werden. Zuerst sollte die Entropie der Daten mit einem geeigneten Verfahren möglichst minimiert und der Datenumfang reduziert werden. Hier bietet sich neben dem Difference Coding auch eine Codierung der Differenzen zu einem fest gespeicherten Referenzpuls an. Anschließend könnten diese Daten dann mit einem adaptiven Verfahren (zum Beispiel adaptives Huffman Coding) weiter komprimiert werden. Der Vorteil eines adaptiven Verfahrens liegt in der automatischen Anpassung an wechselnde Wahrscheinlichkeiten. Bei diesen Verfahren kann außerdem auf die Speicherung eines Codebaumes verzichtet werden, da er im Datensatz integriert ist. Dieses Verfahren wurde im Rahmen der Diplomarbeit nicht untersucht, da seine Implementation für den knapp gesteckten Zeitrahmen zu aufwendig ist. Auch über eine Fehlerkorrektur (zum Beispiel *Cyclic Redundancy Check*) sollte nachgedacht werden.

Anhang A

Grundlagen der Informationstheorie

Die Informationstheorie beschäftigt sich mit der mathematischen Beschreibung von Information und Kommunikation im weitesten Sinne. Als Begründer der Informationstheorie gilt *Claude Shannon*, der in seinem 1948 veröffentlichten Artikel "A mathematical theory of communication" [29] die Wahrscheinlichkeitstheorie zur Beschreibung von Kommunikation verwendete. Im folgenden sollen die für das weitere Verständnis wichtigen Begriffe erläutert werden.

A.1 Entropie und Information

Der Begriff der Entropie stammt ursprünglich aus der Physik. Er wurde 1854 von Rudolph Clausius als Zustandsgröße in die Thermodynamik eingeführt. Mit der Definition der Entropie als

$$H = -k_B \sum_k p_k \ln p_k \tag{A.1}$$

k_B: Boltzmann Konstante
p_k : Wahrscheinlichkeit des Zustandes mit der Nummer k

durch Boltzmann übernahm die Entropie eine zentrale Rolle in der statistischen Physik. Wie sich herausstellen wird, sind die Entropie in der Physik und in der Informationstheorie bis auf einen Faktor identisch. Während jedoch in der Physik Ensembles von Teilchen in verschiedenen (Quanten-) Zuständen betrachtet werden, interessiert man sich in der Informationstheorie für Ensembles von Zeichen.

Um die weiteren Betrachtungen zu vereinfachen, soll davon ausgegangen werden, daß die Zeichen aus einem endlichen Vorrat stammen. Diesen bezeichnet man als Alphabet. Mathematisch bedeutet das folgendes.

Definition A.1 *Ein Alphabet $A = \{a_1, ..., a_N\}$ ist die Menge aller Zeichen, die zur Bildung einer Nachricht zur Verfügung stehen. Mit $|A|$ wird die Zahl der Elemente von A bezeichnet: $|A| = N$.*

Definition A.2 *Ein Ensemble S ist ein Alphabet A mit zugeordneten Wahrscheinlichkeiten für das Auftreten der Zeichen $p_k := p(a_k)$. Es gelte außerdem: $\sum_{k=1}^{N} p_k = 1$.*

Man möchte nun gerne ein Maß für die Information haben, die mit der Übertragung eines Zeichens aus einem bestimmten Ensemble verknüpft ist. Oder, um es von der anderen Seite zu betrachten, ein Maß für die Unsicherheit vor der Übertragung dieses Zeichens. Dieses Maß ist die Entropie.

Definition A.3 *Die Entropie eines Ensembles S ist gegeben durch*

$$H(S) = -\sum_{k=1}^{N} p_k \log p_k. \tag{A.2}$$

Die Boltzmann-Konstante, die in der Physik für Identität von statistischer und thermodynamischer Entropie makroskopischer Systeme sorgt, ist hier natürlich verschwunden. Außerdem ist die Basis des Logarithmus noch nicht näher definiert.

Die wichtigsten, auch aus der Physik bekannten, Eigenschaften der Entropie werden im folgenden angeführt. Auf einen Beweis wurde hier, wie auch bei allen folgenden Sätzen verzichtet.

Satz A.1 $H(S) \geq 0; H(S) = 0 \Leftrightarrow \exists_1 k : p_k = 1, p_{i \neq k} = 0.$

Satz A.2 $H(S) \leq \log N$. *Das Gleichheitszeichen gilt genau dann, wenn* $p_1 = p_2 = ... = p_N = \frac{1}{N}$.

Satz A.3 *Für zwei unabhängige Ensembles S_1, S_2 gilt* $H(S_1, S_2) = H(S_1) + H(S_2)$.

Es stellt sich nun noch die Frage nach einer zweckmäßigen Wahl für die Basis der Logarithmen. Da Informationsverarbeitung praktisch immer eine Darstellung der Daten in binärer Form erfordert, ist die Wahl des Zweierlogarithmus sinnvoll:

$$H(S) = -\frac{1}{\ln 2} \sum_{k=1}^{N} p_k \ln p_k; \qquad [H] = bit. \tag{A.3}$$

Betrachtet man ein Alphabet, das nur aus zwei Zeichen besteht, zum Beispiel $A = \{0, 1\}$, so sieht man sofort, was durch diese Normierung erreicht wird. Es gilt nämlich: $H \leq 1bit$. Vor einer 0/1-Entscheidung herrscht also ein Informationsmangel über das System von höchstens einem bit. Das bedeutet also, daß man nach der (störungsfreien) Übertragung der Entscheidung eine Information über das System von höchstens einem bit besitzt. In diesem Sinne dient also die Entropie als Maß für die in einem System enthaltene Information.

A.2 Codierungstheorie

Die Frage im vorangegangenen Abschnitt lautete, wie groß ist der Informationsgehalt eines Ensembles, also eines Alphabetes mit einer bestimmten Wahrscheinlichkeitsverteilung der einzelnen Zeichen. Gegenstand dieses Abschnitts ist nun die Codierung einer Nachricht.

Definition A.4 *Eine Nachricht (*Message*) M ist eine Folge von Zeichen aus einem Ensemble.*

Definition A.5 *Unter einem Code soll eine eineindeutige Abbildung der Zeichen eines Quellalphabets A auf Zeichen(folgen) des Codealphabets C verstanden werden. Diese Zeichenfolgen bezeichnet man als Codewörter. Dabei ist im allgemeinen $|A| \neq |C|$.*

Um die weiteren Betrachtungen zu vereinfachen, soll davon ausgegangen werden, daß die Zeichen einer Nachricht unabhängig voneinander gesendet werden. Eine Nachricht mit n Zeichen ist also nichts anderes, als ein mögliches Ergebnis eines n-mal hintereinander ausgeführten Zufallsexperiments. Es gilt $p(a_i a_j) = p_i p_j$. Diese Einschränkung trifft natürlich in der Realität nicht immer zu, so wird zum Beispiel ein u nach einem q in der deutschen Sprache viel wahrscheinlicher sein als ein x.

Besonders interessant ist die Frage, wieviele Zeichen man zur Codierung eines Zeichens aus einem Ensemble S braucht, wenn $C = \{0, 1\}$, also wieviele bits benötigt werden, um ein Zeichen aus S zu übertragen. Deswegen wird im folgenden der Schwerpunkt bei der Binärcodierung liegen.

A.2.1 Blockcodes

Unter einem Blockcode versteht man die Codierung aller Zeichen eines Ensembles mit Codewörtern gleicher Länge. Der ASCII-Code zur Darstellung von Tastaturzeichen im Computer ist ein Beispiel für einen Blockcode.

Bezeichnet man die Zahl der Zeichen im Quellalphabet mit N und im Codealphabet mit K und stellt die Forderung, daß für jedes Quellzeichen ein eigenes Codewort vorhanden ist, gelangt man zu der Ungleichung: $N^n \leq K^l$, wobei n die Länge der Nachricht und l die Länge des Codes bezeichnet. Eine untere Grenze für die Codelänge ist also:

$$l \geq n \frac{\log N}{\log K}. \tag{A.4}$$

Diese Ungleichung ist vollkommen unabhängig von der Entropie der Quelle, also von den Wahrscheinlichkeiten der einzelnen Zeichen. Betrachtet man hingegen den Limes $n \to \infty$, so läßt sich folgender Satz von Shannon beweisen:

Satz A.4 *Im Limes $n \to \infty$ gilt für die Codelänge $l \geq n \frac{H(S)}{\log K}$ mit beliebig kleinem Verlust an Information.*

Der Beweis läßt sich nachlesen in [30] oder [31]. Dieser Satz stellt eine Verbindung zwischen der Länge eines Codewortes und der Entropie eines Datensatzes her.

A.2.2 Codes mit variabler Länge

Es sollen nun Codes betrachtet werden, deren Länge von Zeichen zu Zeichen variiert. Man bezeichnet solche Codes auch als *Variable Length Codes.* Der große Vorteil von Variable Length Codes gegenüber Blockcodes liegt in der Möglichkeit häufig auftretenden Zeichen kurze Codes zuzuweisen und seltenen dafür längere, so läßt sich im Mittel die Anzahl der zu übertragenden bits pro Zeichen reduzieren. Da man nicht mehr von *der* Codewortlänge sprechen kann, soll das Symbol l_i eingeführt werden, das die Länge des Codewortes für a_i bezeichnet. Die Betrachtungen in diesem Abschnitt beschränken sich auf binäre Codes. Entropie und Wortlänge werden in bits gemessen.

Wie man sich leicht klar macht, läßt sich nicht aus jedem Variable Length Code die Ausgangsnachricht eindeutig reproduzieren. Man stellt deswegen die Forderung der eindeutigen Decodierbarkeit.

Definition A.6 *Ein Code wird als eindeutig decodierbar bezeichnet, wenn jede beliebige Nachricht ihren eigenen Code hat.*

Eine für die praktische Anwendung wichtige Klasse von Codes, die diese Forderung erfüllt, sind die *Prefix Codes.*

Definition A.7 *Ein Code heißt Prefix Code, wenn kein Codewort Präfix eines anderen ist.*

Es läßt sich, zum Beispiel durch explizite Konstruktion, zeigen, daß nicht jeder eindeutig decodierbare Code ein Prefix Code sein muß. Andererseit gilt folgender Satz, dessen Beweis sich zum Beispiel in [30] findet.

Satz A.5 *Jeder eindeutig decodierbare Code läßt sich durch einen Prefix Code ersetzen, ohne die Längen der Codewörter zu verändern.*

Prefix Codes lassen sich sehr gut über binäre Bäume konstruieren. Ein Blatt bezeichnet das zu codierende Quellzeichen. Der Code wird durch den Weg zu diesem Blatt repräsentiert, wobei eine Verzweigung nach links zum Beispiel eine Null bedeutet, eine nach rechts eine Eins. Dadurch, daß nach einem Blatt keine Zweige mehr kommen können, hat man automatisch Bedingung A.7 erfüllt. Ein Beispielbaum ist in Abbildung A.1 zu sehen. Als Zeichen wurden Energiewerte verwendet, wie man sie bei der Auslese des ATLAS Kalorimeter Level-1 Triggers erwartet.

Bis jetzt wurde noch keine Aussage darüber gemacht, wie gut sich eine Nachricht mit einem Prefix Code codieren läßt. Ein Maß für die Güte der Kompression ist die mittlere Wortlänge

$$\bar{l}(C,S) := \sum_{k=1}^{N} p_k l_k. \tag{A.5}$$

Die Argumente C und S sollen anzeigen, daß $\bar{l}$ von der gewählten Codierung C abhängig ist und sich die Mittelwertbildung über das Ensemble S erstrecken soll. Davon zu unterscheiden ist $\bar{l}(C,M)$, die mittlere Codewortlänge für eine bestimme Nachricht. Die

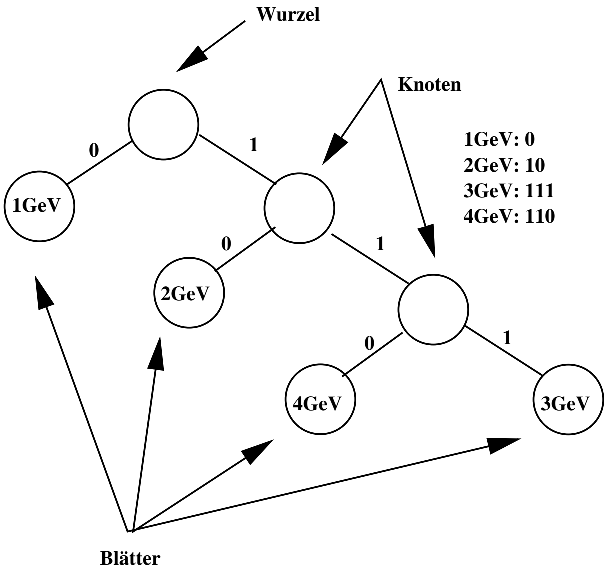

Abbildung A.1: Codebaum für einen Prefix Code

Mittelwertbildung läuft hier über die relativen Häufigkeiten h_k der Zeichen in der Nachricht

$$\bar{l}(C, M) := \sum_{k=1}^{N} h_k l_k. \tag{A.6}$$

Es läßt sich nun folgender Satz beweisen:

Satz A.6 *Für jedes Ensemble S existiert ein Prefix Code, für den gilt*

$$H(S) \leq \bar{l}(C, S) \leq H(S) + 1. \tag{A.7}$$

Man bezeichnet ihn als optimalen Code.

Den Beweis findet man wieder in [30]. Dieser Satz bedeutet, daß man bei der Codierung mit einem optimalen Prefix Code maximal ein bit über der theoretisch erreichbaren Minimallänge liegt. Würde $\bar{l}$ nämlich unter $H(S)$ absinken, wäre damit ein Informationsverlust verbunden, die Nachricht ließe sich nicht mehr vollständig rekonstruieren.

Nachdem nun bekannt ist, daß es einen optimalen Code gibt, interessiert man sich als Anwender dafür, wie sich ein solcher Code explizit konstruieren läßt. In der Praxis verwendet man hierfür den *Huffman Algorithmus.* Er wird bei der Diskussion der Kompressionsalgorithmen genauer erläutert.

Um keine Mißverständnisse aufkommen zu lassen, sei zum Abschluß noch darauf hingewiesen, daß die mittlere Codewortlänge für eine spezielle Nachricht $\bar{l}(C, M)$ natürlich deutlich über $\bar{l}(C, S)$ liegen kann. Dies wird immer dann der Fall sein, wenn Zeichen, die in S eine geringe Wahrscheinlichkeit haben, in der Nachricht M mit großer Häufigkeit auftreten. Im Grenzfall unendlich langer Nachrichten gilt:

$$\bar{l}(C, M) = \bar{l}(C, S) \tag{A.8}$$

Anhang B

Operations- und Testmodi der Kompressionseinheit

Die Bedienung der Kompressionseinheit ist Gegenstand dieses Anhangs. Zunächst wird die Konfiguration über verschiedene Register des RemASIC beschrieben und dann die Kompressions- und Testmodi. Am Schluß wird noch die Belegung der beiden SpyBus-Seiten, die von der Kompressionseinheit genutzt werden vorgestellt.

B.1 Konfiguration

Es gibt vier Register, die speziell der Konfiguration der Kompressionseinheit dienen. Sie werden, wie alle Register des RemASIC, über den PipelineBus konfiguriert. Darüberhinaus werden drei weitere Register für den Event Readout benötigt, die gleichzeitig der Konfiguration des FeASIC Interface dienen. Wird Huffman Coding gewählt, so muß außerdem sichergestellt sein, daß der Encoder RAM entsprechend beschrieben wurde. Die einzelnen Register haben folgende Bedeutung:

RegComprMode[3:0] Dieses Register stellt den Kompressionsmodus ein. Die untersten 3 bit geben die Nummer des Algorithmus an, während bit 4 als 8BitSelect dient. Das bedeutet, daß bei gesetztem 8BitSelect 8 bit Daten im gewählten Kompressionsmodus verarbeitet werden. Für das Difference und Huffman-Inspired Coding hat dieses bit keine Bedeutung, es wird ignoriert. Die Numerierung der Algorithmen ist im folgenden aufgeführt:

- **0** NoOperation
- **1** Huffman Coding
- **2** Huffman-Inspired Coding
- **3** Difference Coding
- **4** Run-Length Encoding

RegComprParameter[9:0] Wird mit dem Huffman-Inspired Verfahren komprimiert, so wird in diesem Register der Wert der Referenzenergie E_R erwartet. Im Run-Length

Modus werden alle Energien, die kleiner sind als der Wert in RegComprParam auf Null gesetzt. Auf diese Weise läßt sich ein Energie-Cut wählen.

RegPreMainBufferSource Wird dieses Flag gesetzt, so wird das Schreiben eines komprimierten Events in den Pre-Main Buffer unterdrückt. Das bedeutet, die Auslese läuft ganz normal, die vorhandenen Daten im Pre-Main Buffer werden jedoch nicht mit denen der Auslese überschrieben. Man kann auf diese Weise die Funktionsfähigkeit des Interfaces zwischen Pre-Main Buffer und Main Buffer testen, indem der Pre-Main Buffer vor der Auslese mit bekannten Daten beschrieben wird.

RegPreMainBufferSel Dieses Register gibt die Nummer des RAMs an, der im PreMainBufferSource-Modus in den Main Buffer geschrieben werden soll.

RegPortMask[3:0] Dieses Register gibt direkt die Konfiguration der FeASIC Ports an. Ist zum Beispiel bit drei gesetzt, so hängt an Port drei mindestens ein FeASIC.

RegDaisyChainCount[7:0] RegDaisyChainCount hat zwei bits für jeden FeASIC Port, die die Anzahl der FeASICs in der Daisy Chain an diesem Port angeben. Die zwei LSBs sind für Port 0, die zwei MSBs für Port 3.

RegBcPerEvent[2:0] In diesem Register steht die Anzahl der pro Event ausgelesenen Bunch-Crossings.

B.2 Headerwort

Ein fertig komprimierter Datensatz enthält einen Header, in dem neben Informationen des FeASIC Interface auch der Kompressionsmodus kodiert ist. Das Headerwort ist das erste 32 bit Datenwort eines Datensatzes. Die von der Kompressionseinheit benutzten bits werden im folgenden aufgelistet:

Headerwort[31]: Repeat Flag Das Repeat Flag ist gesetzt, wenn die Kompression im gewählten Modus abgebrochen wurde, und im NoOperation Modus wiederholt wurde. Dies geschieht, wenn sich das Event wider Erwarten so stark aufblähen sollte, daß es nicht mehr in den Pre-Main Buffer paßt.

Headerwort[30:27]: Kompressionsmodus In diesen Teil des Headers wird das Register RegComprMode kopiert.

B.3 Beschreiben der Speicher

Das Beschreiben der Speicher erfolgt über den PipelineBus. In welcher Form die Daten dabei vorliegen müssen, um die Speicher korrekt zu konfigurieren, wird in diesem Abschnitt erklärt.

Encoder RAM Da der Encoder RAM 24 bit Datenwörter und 8 bit Adressen hat, ist seine Konfiguration besonders einfach. Adresse und Datenwort können zusammen in ein 32 bit Wort des PipelineBusses gepackt werden.

```
PipelineBusData[23: 0] -> Datenwort
PipelineBusData[24:31] -> Adresse
```

Pre-Main Buffer Zum Beschreiben des Pre-Main Buffers können beide Modi, der konsekutive und der wahlfreie verwendet werden. Das erste Datenwort hat bei beiden Modi dasselbe Format:

```
PipelineBusData[ 5: 0] -> Adresse des ersten Datenwortes
PipelineBusData[    6] -> Nummer des RAMs der beschrieben werden soll
PipelineBusData[    7] -> 0: wahlfrei, 1: konsekutiv
PipelineBusData[31: 8] -> unbenutzt
```

Im konsekutiven Modus folgen nun nur noch 32 bit Datenwörter, die aufsteigend ab der spezifizierten Adresse abgelegt werden:

```
PipelineBusData[31: 0] -> Datenwort
```

Im wahlfreien Modus folgt jetzt abwechselnd ein Daten- und ein Adresswort. Bei jeder Adresse kann auch der Speicher gewählt werden:

```
PipelineBusData[ 5: 0] -> Adresse des Datenwortes
PipelineBusData[    6] -> Nummer des RAMs der beschrieben werden soll
PipelineBusData[31: 7] -> unbenutzt
```

Die Auswahl eines Schreibmodus ist nur einmal zu Anfang des Schreibvorganges möglich.

B.4 Auslese der Speicher

Bei der Auslese des Encoder RAMs werden, angefangen mit Adresse Null konsekutiv alle 256 Datenwörter in den Main Buffer geschrieben. Dabei werden jeweils die 24 LSBs eines 32 bit Wortes belegt. Die Adressen werden bei der Auslese nicht angegeben.

Wird die Auslese des Pre-Main Buffers aktiviert, so wird zunächst RAM0, dann RAM1 in den Main Buffer kopiert. Angefangen wird wieder mit Adresse Null. Die Auslese des Pre-Main Buffers erfolgt immer komplett, eine Auslese nur eines RAMs ist nicht möglich.

B.5 SpyBus

Auf den SpyBus können über einen Multiplexer acht *Seiten* mit jeweils acht internen Signalen gegeben werden. Die Kompressionseinheit belegt zwei dieser Seiten komplett. Eine Seite ist für die BIST Signale der Speicher reserviert, von der die Kompressionseinheit drei Leitungen belegt. Die Signale der Kompressionseinheit, die über den SpyBus angeschaut werden können, sind im folgenden aufgeführt.

SEITE 0:

```
0 BIST RAM0
1 BIST RAM1
2 BIST Encoder RAM
```

SEITE 6:

```
4:0 ShiftBits[4:0]
7:5 State Register aus CUN Control
```

SEITE 7:

```
0:  LSB des Bunch-Crossing Counters
1:  LSB des Daisy-Chain Counters
2:  LSB des Port Counters
4:3 State Register der Registersteuerung in der Shifting Unit
7:5 State Register aus Data Flow Control
```

Bunch-Crossing, Daisy Chain und Port Counter werden beim Event Readout verwendet, um die Adressen für die Auslese der Daten aus dem Event Buffer hochzuzählen. Dadurch, daß das LSB herausgeführt ist, läßt sich beobachten, ob sie an der richtigen Stelle inkrementiert werden, denn durch Addition einer eins wird das LSB invertiert.

Anhang C

Die Shapingfunktion

Für einen Dreieckspuls

$$i_\delta(t) = I_0 \left(1 - \frac{t}{t_{dr}}\right) \quad t \leq t_{dr} \tag{C.1}$$

mit der Abklingzeit t_{dr}, berechnet sich die Shapingfunktion nach [17] mit den Formeln:

$$\begin{aligned}
V(x) &\propto h_1(x) - \frac{1}{x_{dr}} h_2(x) \quad \text{für } x \leq x_{dr} && \text{(C.2)}\\
V(x) &\propto h_1(x) - \frac{1}{x_{dr}} (h_2(x) - h_2(x - x_{dr})) \quad \text{für } x \leq x_{dr} && \text{(C.3)}\\
h_1(x) &= \frac{\lambda^2 e^{-x/\lambda}}{(\lambda-1)^3} - \left[\frac{x^2}{2} + \frac{\lambda x}{\lambda - 1} + \frac{\lambda^2}{(\lambda-1)^2}\right] \frac{e^{-x}}{\lambda - 1} && \text{(C.4)}\\
h_2(x) &= 1 - \frac{\lambda^3 e^{-x/\lambda}}{(\lambda-1)^3} + \left[\frac{x^2}{2} + \frac{2\lambda - 1}{\lambda - 1} x + \frac{3\lambda^2 - 3\lambda + 1}{(\lambda-1)^2}\right] \frac{e^{-x}}{\lambda - 1}. && \text{(C.5)}
\end{aligned}$$

Dabei wurden folgende Substitutionen verwendet

$$x = \frac{t}{\tau}, \quad x_{dr} = \frac{t_{dr}}{\tau}, \quad \lambda = \frac{\tau_{pa}}{\tau}. \tag{C.6}$$

Mit τ und τ_{pa} werden die Zeitkonstanten von Pulsformer und Vorverstärker bezeichnet [17]. In der Simulation wird die Shapingfunktion $V(x)$ dann so normiert, daß ihr Maximum der Energie im Trigger Tower entspricht.

Literaturverzeichnis

[1] Perkins, Donald H.
Introduction to High Energy Physics
Addison-Wesley, Menlo Park, 1986

[2] ATLAS Collaboration
Technical Proposal
CERN/LHCC/94-43,LHCC/P2

[3] ATLAS Homepage
ATLAS Figures
`http://atlasinfo.cern.ch/Atlas/ATLASFIGS/atlasfigures.html`

[4] Kleinknecht, K.
Detektoren für Teilchenstrahlung
B.G. Teubner, Stuttgart, 1992

[5] Homepage der ATLAS-Gruppe des ASIC-Labors
Transparents for ATLAS
`http://wwwasic.ihep.uni-heidelberg.de/atlas/`

[6] Schumacher, C.
Der ATLAS Level-1 Trigger, Auslese des Frontends
Diplomarbeit, Universität Heidelberg, 1997, HD-ASIC-31-0197

[7] Mass, A. et al.
A Front-End Digitisation and Readout System for the ATLAS Level-1 Calorimeter Trigger
CERN/LHCC/96-39

[8] Mass, A.
Front-End and Fast Readout ASIC: FeAsic. User Manual
HD-ASIC-19-0896

[9] Pfeiffer, U.
Doktorarbeit in Vorbereitung

[10] Hewlett Packard
Low Cost Gigabit Rate Transmission/Receive Chipset, Technical Data

[11] CERN HSI Group
S-Link Homepage
`http://www.cern.ch/HSI/s-link/`

[12] Stroustrup, B.
The C++ Programming Language
Addison-Wesley, 2nd Ed., 1995

[13] Altarelli, G. et al.
Proton-Antiproton Collider Physics
World Scientific, 1989

[14] ATLAS Collaboration
Technichal Design Report, Calorimeter Performance
CERN/LHCC 96-40

[15] Wunsch, M.
Persönliche Mitteilung

[16] Müller, C.
Persönliche Mitteilung

[17] Chase, R. L.
A Fast Monolthic Shaper for the ATLAS E.M. Calorimeter
ATLAS Internal Note LARG-NO-10

[18] Lelewer, D. A. et al.
Data Compression
`http://www.ics.uci.edu./~dan/pubs/DataCompression.html`

[19] Mass, A.
Persönliche Mitteilung

[20] Troll Tech AS
Online Qt Documentation
`http://www.troll.no/qt/index.html`

[21] Josuttis, N.
Die C++ Standardbibliothek
Addison-Wesley, 1996

[22] Cadence Design Systems Inc.
Verilog XL Reference
Online-Dokumentation, 1997

[23] Thomas, D. E. et al.
The Verilog Hardware Description Language
Kluwer, 1996

[24] Cadence Design Systems Inc.
Synergy HDL Command Reference Help System
Online-Dokumentation, 1997

[25] Cadence Design Systems Inc.
Synergy HDL Syntheizer and Optimizer Modeling Style Guide
Online-Dokumentation, 1997

[26] Cadence Design Systems Inc.
Simulation and Test Language User Guide
Online-Dokumentation, 1997

[27] Mano, M. Morris
Digital Design
Prentice-Hall, 1984

[28] Grill, S.
Documentation to the HP82000 Test Pattern Generator
`http://www.ihep.uni-heidelberg.de/~keller/cds/chipTester/Documentation.html`

[29] Shannon, C. E.
A mathematical theory of communication
Bell Sys. Tech. J., vol.27, pp. 379-423, 623-656, 1948

[30] Jones, D. S.
Elementary Information Theory
Oxford University Press, 1989

[31] MacKay, David
A Short Course in Information Theory - Outline
`http://131.111.48.24/pub/mackay/info-theory/course.html`

Danksagung

Danken möchte ich allen, die, auf die eine oder andere Weise, zum Gelingen dieser Arbeit beigetragen haben.

- Prof. Meier für die Möglichkeit meine Diplomarbeit am ASIC-Labor durchzuführen.
- Prof. Straumann für die Übernahme der Zweitkorrektur.
- Alexander Mass für die ausgezeichnete Betreuung und die Möglichkeit eigene Ideen zu verwirklichen.
- Cornelius Schumacher für viele Anregungen und die Hilfe bei der Einarbeitung in Cadence und Verilog. Ganz besonders für eine genaue Durchsicht meiner Diplomarbeit.
- Ullrich Pfeiffer für die genaue Durchsicht des Abstracts und die Klärung vieler Fragen.
- Michael Keller, der eine unerschöpfliche Quelle von Tips und Ratschlägen bei Problemen mit UNIX und Cadence war.
- Johannes Schemmel für die überaus gründliche Durchsicht dieser Arbeit und seine stete Bereitschaft meine Fragen zu C++ zu diskutieren.
- Martin Wunsch für seine geduldige Hilfe bei Fragen der Hochenergiephysik.
- Allen Mitgliedern des ASIC-Labors für das gute Arbeitsklima. Ganz besonders A. Mass, J. Schemmel und P. Schneider, die meine Begeisterung für das Kaffeetrinken teilten.
- Annette Wenig, die mir stets zuhörte und mit einer Menge Kaffee und guter Einfälle dafür sorgte, daß mein Studium nie langweilig wurde.
- Karl-Heinz Apelt und allen Mitgliedern und Gästen des J.C. Scheune für ihre moralische Unterstützung während des Studiums.
- Meinen Eltern, ohne deren Unterstützung mein Studium so nicht möglich gewesen wäre.