RUPRECHT-KARLS-UNIVERSITÄT HEIDELBERG

Volker Schatz

Test of a Readout and Compression ASIC for the ATLAS Level-1 Calorimeter Trigger

Diploma Thesis

HD-KIP-00-13
HD-ASIC-58-0600

KIRCHHOFF-INSTITUT FÜR PHYSIK

Faculty of Physics and Astronomy

University of Heidelberg

Diploma Thesis
in Physics

submitted by
Volker Schatz
born in Karlsruhe

June 2000

Test of a Readout and Compression ASIC for the ATLAS Level-1 Calorimeter Trigger

This diploma thesis has been carried out by Volker Schatz at the

Kirchhoff-Institut für Physik of the University of Heidelberg

under the supervision of

Prof. Dr. K. Meier

Test of a Readout and Compression ASIC for the ATLAS Level-1 Calorimeter Trigger:

The subject of this thesis is a prototype readout ASIC for the first-level trigger of the ATLAS experiment. This so-called readout merger ASIC (RemAsic) reads out and compresses raw calorimeter data.
Using a test system based on VME boards, an operating environment for the RemAsic similar to the Level-1 Trigger system was built up. Readout and access of internal RemAsic memories was tested. The tests were largely successful.
For that purpose, a controller was developed and implemented in an FPGA. For certain tests, another FPGA design acting as a data source for the RemAsic was developed. Software was written to control test hardware, the RemAsic and other prototype components. This software package will also be used for monitoring and diagnostics in the ATLAS experiment.
For evaluating compression algorithms for the ATLAS Level-1 Calorimeter Trigger, a software program was written. It generates calorimeter data and calculates the compression rates for different algorithms. Improved compression algorithms based on those of the RemAsic were developed.

Test eines Auslese- und Kompressions-ASICs für den ATLAS Level-1 Kalorimetertrigger:

Das Thema dieser Arbeit ist der Prototyp eines Auslese-ASICs für die erste Triggerstufe des ATLAS-Experiments. Dieser sogenannte Readout Merger ASIC (RemAsic) liest Kalorimeterdaten aus und komprimiert sie.
Mit einem auf VME-Karten basierenden Testsystem wurde ein dem Level-1 Trigger ähnliches System aufgebaut, in dem der RemAsic betrieben wurde. Die Auslese und der Zugriff auf interne Speicher des RemAsics wurde getestet. Die Tests waren zum größten Teil erfolgreich.
Zu diesem Zweck wurde ein Controller entwickelt und in einem FPGA implementiert. Für spezielle Tests wurde ein weiteres FPGA-Design entwickelt, das als Datenquelle für den RemAsic diente. Um die Testhardware, den RemAsic und andere Prototypen zu kontrollieren, wurde Software geschrieben. Dieses Softwarepaket wird auch im ATLAS-Experiment zur Überwachung und Diagnose der Hardware verwendet werden.
Um Kompressionsalgorithmen für den ATLAS Level-1 Kalorimetertrigger zu evaluieren, wurde ein Softwareprogramm geschrieben. Es erzeugt Kalorimeterdaten und berechnet Kompressionsraten für verschiedene Algorithmen. Aus den Kompressionsalgorithmen des RemAsics wurden verbesserte Algorithmen entwickelt.

Contents

Introduction

The ATLAS experiment at the Large Hadron Collider (LHC) at CERN will search for the Higgs boson and supersymmetric particles and investigate other physics questions. Since physics events not yet observed by other experiments are rare, a highly sophisticated trigger system is required. The Level-1 Trigger, the first level of the ATLAS trigger system, reduces the event rate from the LHC bunch-crossing rate of 40 MHz to a maximum of 100 kHz. This is done by searching for isolated particles and jets above a certain energy threshold and by requiring the global energy sum to clear another threshold.

The Level-1 Trigger is composed of the Level-1 Calorimeter Trigger, the Level-1 Muon Trigger and the Central Trigger Processor. The first two parts process data from specific parts of the detector. The Central Trigger Processor takes the final trigger decision. The Calorimeter Trigger contains a preprocessor system which digitises data and prepares them for the trigger.

Data are read out from the trigger for every accepted event. This serves for monitoring trigger function and complementing other readout data. In the case of the Preprocessor of the Level-1 Calorimeter Trigger, the readout is to be done by a Field Programmable Gate Array (FPGA). Its prototype is an Application-Specific Integrated Circuit (ASIC), the Readout Merger ASIC (RemAsic). The amount of data read out is very large. To simplify data transmission downstream of the RemAsic, it compresses readout data. Since the RemAsic is a prototype intended for evaluation of this approach, it implements a variety of compression algorithms. Tests of the RemAsic are the subject of this thesis.

The first chapter of this thesis describes the aims of the ATLAS experiment and the ATLAS detector. The ATLAS trigger system and the Level-1 Calorimeter Trigger Preprocessor, the operating environment of the RemAsic's successor, are also treated in that chapter. Chapter 2 explains the RemAsic and other prototype components with which it works together. In the following chapter, the hardware used for the tests and a controller developed for it is described. Chapter 4 presents software written to control all the hardware. Chapter 5 decribes the RemAsic test setups and the test results. In the last chapter, the compression algorithms of the RemAsic are discussed. A software program developed for calculating compression rates is presented and its results are given. Finally, improvements of the compression algorithms of the RemAsic are described.

Chapter 1

Physical Background and Trigger System

The ATLAS experiment is designed to investigate the existence of the Higgs boson and of supersymmetric particles and other physics questions. It is located at the Large Hadron Collider accelerator which will achieve unprecedented center-of-mass energies.

The high event rate at LHC requires high trigger selectivity. The ATLAS trigger has three trigger levels of different complexity. The first-level trigger contains a preprocessor system for signal preparation. The ASIC which is the subject of this thesis is a prototype of one of its components.

1.1 The ATLAS Experiment

1.1.1 The Large Hadron Collider at CERN

The Large Hadron Collider (LHC) is designed to achieve energies never reached before in accelerator physics. It will provide proton-proton collisions with a center-of-mass energy of 14 TeV and a luminosity[1] of 10^{34} $\mathrm{cm}^{-2}\mathrm{s}^{-1}$. The LHC accelerates bunches of protons or heavy ions in two separate synchroton rings in opposite directions. The beams intersect at four interaction points where experiments are located. At those points, bunches collide every 25 ns, which is called a bunch-crossing. For that reason, 40 MHz is the characteristic frequency for LHC experiments and their electronics. The steady series of bunch-crossings is occasionally interrupted by 'bunch gaps' needed for beam insertion and extraction (dump).

The LHC is being installed in the tunnel of an existing accelerator, the Large Electron Positron Collider (LEP). It is scheduled to go into operation in the year 2005. Its circular tunnel has a radius of 4.3 km and crosses the border between France and Switzerland.

At the LHC, two general-purpose experiments are located: ATLAS (A Toroidal LHC Apparatus) and CMS (Compact Muon Solenoid). They have roughly the same aims, described in the next section. The ALICE experiment (A Large Ion Collider Experiment) is dedicated to the physics of strongly interacting matter at extreme energy densities. The purpose of a fourth experiment, LHCb, is studying CP violation in decays of B mesons.

[1] The luminosity is the number of particle collisions per geometrical cross-section per time interval.

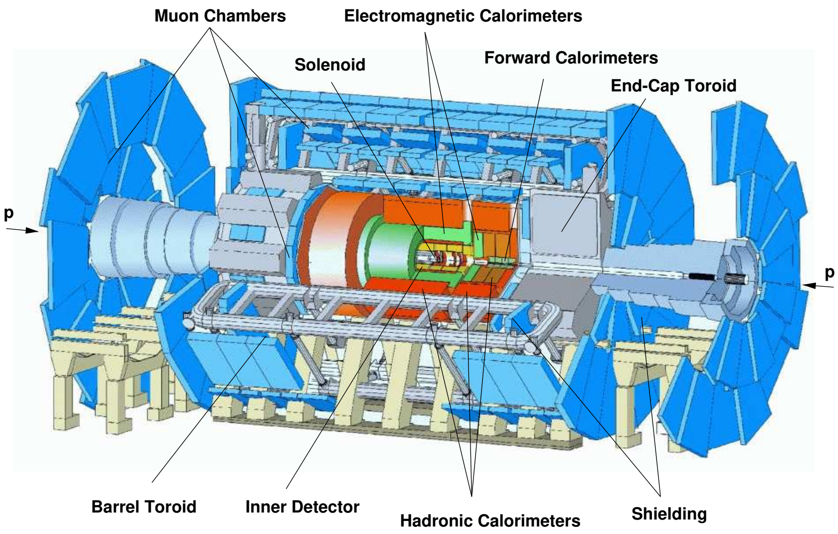

Figure 1.1: The ATLAS detector

1.1.2 Physics Issues at ATLAS

The Standard Model is widely regarded as a comprehensive description of the known phenomena concerning particles and their interactions (excluding gravity). However, not all of it has yet been confirmed by experiments. The Higgs boson which is instrumental in explaining particle masses has never been created in an experiment because of its high mass. The LHC's energy range will allow this, and verifying the Higgs's existence is one of the primary aims of the ATLAS experiment.

An extension of the Standard Model, Supersymmetry, introduces a symmetry between fermions and bosons. It postulates for each Standard Model particle a supersymmetric partner whose spin differs by one half. The supersymmetric particles have never been observed so far and are to be searched for by ATLAS.

Another aim of the ATLAS experiment is to ascertain that quarks are truly elementary or to find their substructure if they are not. The high energies available at the LHC may reveal substructures not visible to earlier experiments. Besides, ATLAS will measure decays of B mesons. On the basis of these measurements, CP violations will be searched for and the elements of the Cabibbo-Kobayashi-Maskawa matrix will be determined more precisely.

1.1.3 The ATLAS Detector

1.1.3.1 The Inner Detector

The inner detector is for tracking particles on their way from the interaction point. From its measurements, particle tracks and their vertices can be reconstructed. Since the inner detector is located in a magnetic field, the particles' momentum can be worked out from their tracks and their energy measured by the calorimeter.

The inner part of this detector is made up of high-resolution semiconductor pixel and strip detectors. Farther from the beam tube, less accurate wire chambers are located.

1.1.3.2 The Calorimeters

The task of calorimeters is to measure the energy of particles. This is done with alternating layers of detector and absorber material. The particles decay in the absorber regions, creating showers of particles. This process is repeated in the following absorber layers until the particles are slow enough to be absorbed without creating others. Only a small fraction of the energy is deposited in the detector material, but it is proportional to the energy of the shower and hence of the primary particle.

The ATLAS detector contains several different calorimeters, as can be seen in Figure 1.1. The one closest to the interaction point is the electromagnetic calorimeter. Its absorber material is lead. Liquid argon is used as detector material because it is tolerant to intensive radiation. The electromagnetic calorimeter is suited only for detecting electromagnetic showers of electrons and photons. Hadrons pass through it without depositing much energy.

Hadrons are detected by the hadron calorimeter behind the electromagnetic one. Its barrel region is composed of staggered tiles of plastic scintillator embedded in iron which acts as absorber. The tiles are perpendicular, not parallel to the beam axis, but their staggered arrangement ensures good coverage.

For particles with trajectories very close to the beam axis, there is the forward calorimeter. It consists of liquid argon as sensitive material and copper and tungsten as absorber.

1.1.3.3 The Muon Detector

Muons are the most penetrating particles of all. For that reason, the muon detector is the outermost layer of the ATLAS detector. It consists of different types of drift chambers. The barrel of the muon detector is located in the field of the air core toroid magnets that gave ATLAS its name (A Toroidal LHC Apparatus). Thus, the muons' momenta can be reconstructed from the bending radii of their tracks.

1.2 The ATLAS Trigger System

1.2.1 Overview

The ATLAS detector generates about 50 terabytes of data each second. Storing this amount of data for off-line analysis is techically impossible. Furthermore, most of this information is unrelated to the physics processes ATLAS is intended to investigate. It is the task of the trigger system to decide whether such a process happened at any given bunch-crossing and to trigger readout of the detector only in that case.

Deciding whether detector data represent an interesting physics event takes in the order of seconds. Storing all data even for that time is still prohibitive. For that reason, the trigger decision is taken in several stages. These stages, called trigger levels, are the Level-1 Trigger, the Level-2 Trigger and the Event Filter (see Figure 1.2). Lower levels of the trigger are less selective than higher ones. They exist to reduce the amount of data to be stored while the trigger decision in the next level is being taken. Thus higher levels of the trigger can take more time without requiring unacceptable amounts of storage space.

The Level-1 Trigger is the first level of the trigger. Since its decision must be made within 2 μs, its algorithm is implemented in hardware. The time limit applies to the sum of signal delays (latency) through electronics and cables. The Level-1 Trigger comprises the Calorimeter Trigger and a separate Muon Trigger. The Muon Trigger's decision is based on special trigger chambers in the muon detector. The Calorimeter Trigger uses analogue sums of the signals of several calorimeter cells. See next section for more detail. The third component of the Level-1 Trigger is the Central Trigger Processor which combines the results of the other two subsystems to make the final trigger decision.

The Level-1 Trigger has to reduce the data rate by a factor of 500. To do this, it searches for isolated electrons, photons, muons and hadrons and for jets of particles. It also calculates

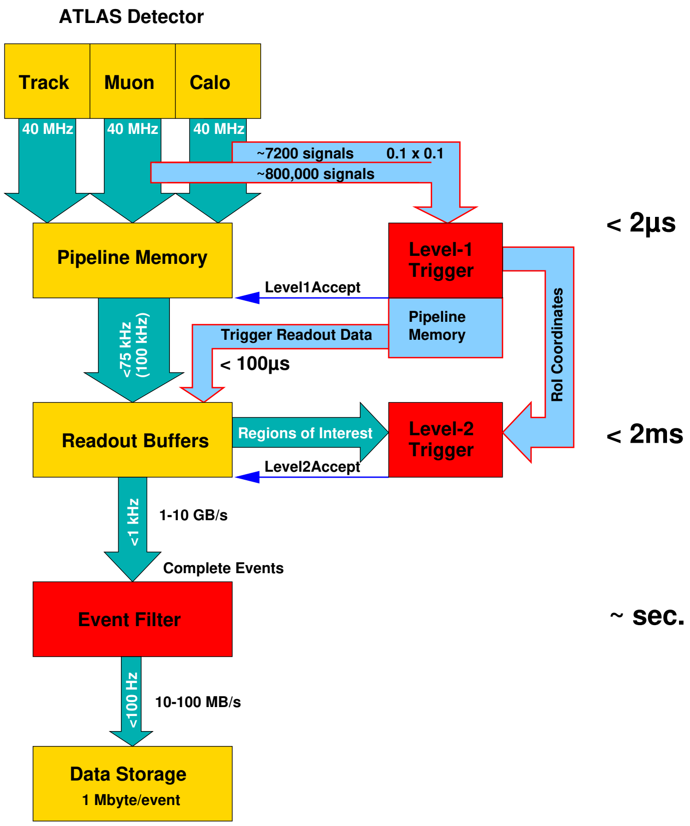

Figure 1.2: ATLAS data acquisition and trigger data paths [Pfei]

global quantities like the transverse energy sum and the missing transverse energy, which must pass tresholds for the event to be accepted. It determines the coordinats of regions of interest to be investigated by the Level-2 Trigger.

The Level-2 Trigger, unlike the Level-1 Trigger, has access to the same data which are later passed on to the readout. It uses the calorimeter data in their full granularity, the regular muon detector chambers and the inner tracking detector. For the regions of interest defined by the Level-1 Trigger, it converts data from several subdetectors to physical quantities. The Level-2 Trigger algorithms are implemented in software running on dedicated multi-processor systems.

The last level of the trigger is called the Event Filter. It is functionally part of the data acquisition (DAQ) data flow (see Figure 1.2). Unlike the Level-2 Trigger, it is not confined to regions of interest but considers data from the whole detector. It builds physics events from the fragments provided by the Level-2 Trigger and selects interesting events for storage.

1.2.2 The Level-1 Calorimeter Trigger

The ATLAS Level-1 Calorimeter Trigger is the part of the first-level trigger which operates exclusively on calorimeter data. Its architecture is shown in Figure 1.3. The Level-1 Trigger does not use the full granularity that single calorimeter cells provide. It processes trigger tower data which are sums of the energies of several calorimeter cells. Trigger towers comprise the cells in an interval

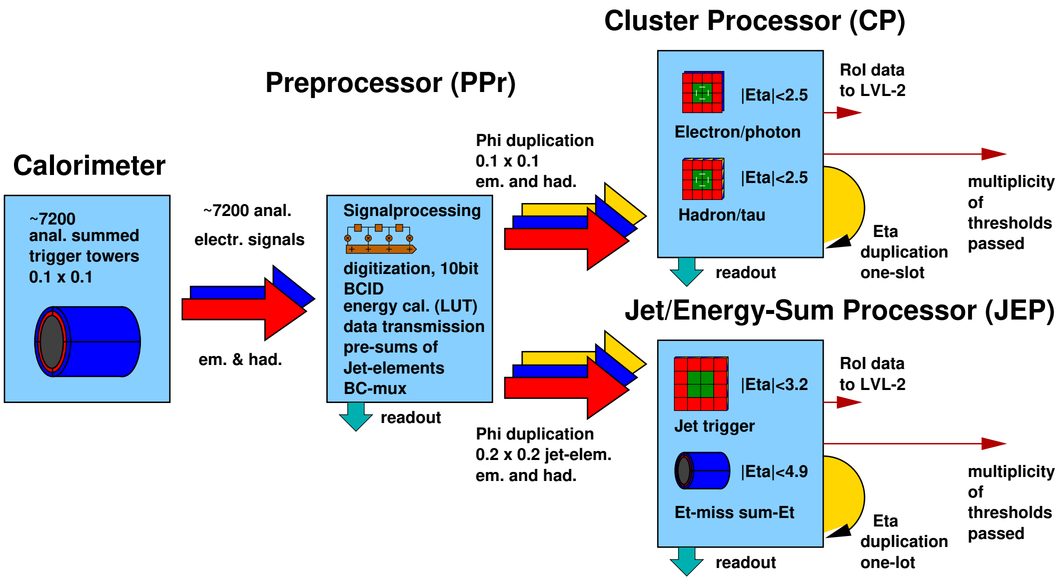

Figure 1.3: ATLAS Level-1 Calorimeter Trigger architecture [Pfei]

of 0.1×0.1 in pseudorapidity[2] and azimuth angle.

The signals first pass through the Preprocessor. It digitises them and calibrates the energies using lookup-tables. It also performs bunch-crossing identification, ie it correlates energy pulses with the bunch-crossing of the event from which they resulted. These data are passed on to the Cluster Processor (CP) and the Jet/Energy-Sum Processor (JEP) which work in parallel. The Cluster Processor's task is to detect isolated particles (electrons, photons or hadrons) above a certain energy threshold. The Jet/Energy-Sum Processor looks for jets and computes the sum of absolute transverse energies and the missing transverse energy. Both transmit the multiplicity of passed thresholds to the Central Trigger Processor which takes the final trigger decision. If the event is accepted, the regions of interest (RoIs) are passed on to the Level-2 Trigger. That are the locations of the cells causing the Level-1 Accept.

As has been said before, the Level-1 Trigger algorithms are implemented in hardware. This is necessary because stringent latency requirements demand fast signal processing. The Preprocessor relies on an application-specific integrated circuit (ASIC) for customised signal processing. Besides, it uses commercial ADCs and serial transmitters. The Cluster Processor also contains commercial components. Its algorithm will probably be implemented in an FPGA[3], though an ASIC is still being considered. The Jet/Energy-Sum Processor will also be implemented in FPGAs.

All three subsystems of the Calorimeter Trigger provide data for readout. These data will be used for monitoring the trigger and trigger-tower summation. The Preprocessor can read out raw or processed data. The CP and JEP provide the data they receive from the Preprocessor (for checking data transmission) and the results of their own processing.

[2]Pseudorapidity (η) is used instead of the polar angle θ. The fact that pseudorapidity differences are Lorentz invariant simplifies trigger algorithms. Its definition is: $\eta = -\ln \tan \frac{\theta}{2}$

[3]An FPGA (Field Programmable Gate Array) is a piece of configurable hardware. It contains logical gates and flip-flops that can be connected to perform all sorts of functions, in the most general sense. In many cases, an FPGA is an alternative to a digital ASIC (Application-Specific Integrated Circuit).

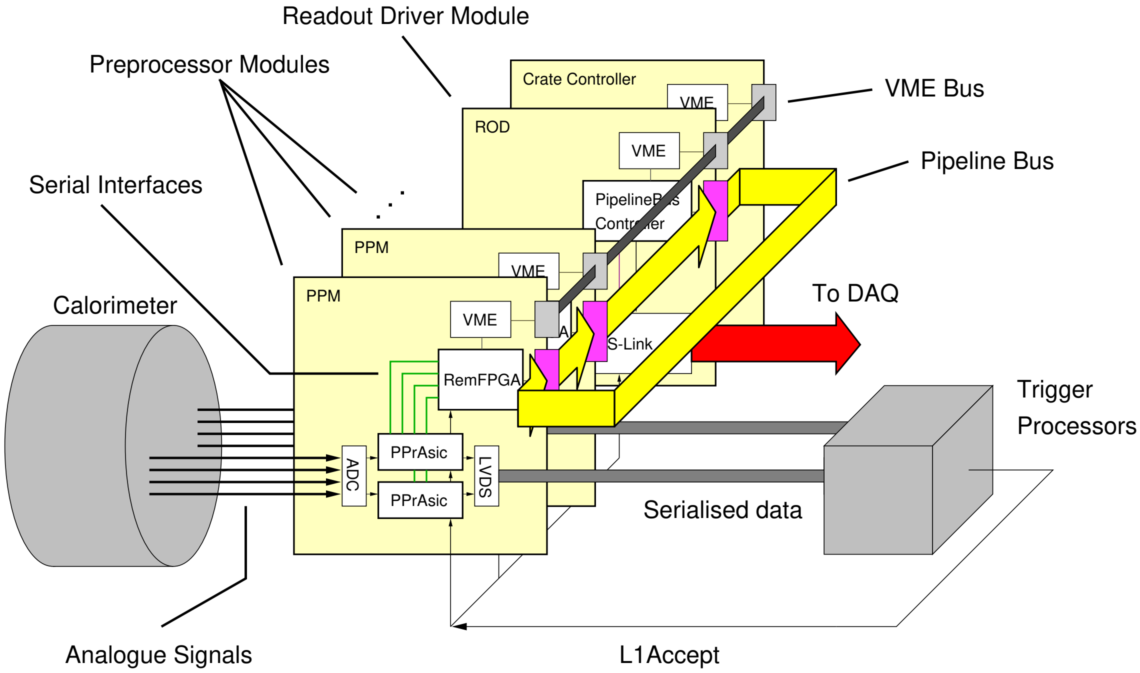

Figure 1.4: Calorimeter Trigger Preprocessor system [Schu]

1.2.3 The Level-1 Calorimeter Trigger Preprocessor

The Preprocessor of the Calorimeter Trigger receives about 7 200 channels of analogue trigger-tower data. It digitises and preprocesses them for the Cluster Processor (CP) and the Jet/Energy Sum Processor (JEP). The Preprocessor will physically consist of eight 9U VME crates with 16 Preprocessor Modules each. For high integration density and easy substitutability of defect parts, data processing will be done by Multi-Chip Modules (MCMs). One MCM carries four ADCs, an ASIC for customised signal processing and three serial transmitters. It can handle four channels.

The Preprocessor processes data on two data paths, as can be seen in Figure 1.4. On the trigger data path, it digitises trigger-tower energies, calibrates them and performs bunch-crossing identification (BCID). This means locating energy pulses in time. The latter two tasks are performed by the Preprocessor ASIC (PPrAsic). The results are serialised and transmitted to the two trigger processors, the CP and the JEP. This is done using LVDS technology.

The other data path serves for reading out data of interest for monitoring the Preprocessor. Readout data are provided by the Preprocessor ASIC. The so-called Readout Merger FPGA (RemFPGA) collects data from all the Preprocessor ASICs on a module via serial interfaces. All RemFPGAs transmit their readout data to one of two readout driver modules on a dedicated parallel bus, the Pipeline Bus. The Pipeline Bus connection will be integrated in the customised backplane of the VME crate. The readout driver transmits the data from the whole crate to the data acquisition (DAQ) readout buffers via fast serial links.

The RemFPGA will compress the data before passing them on. This is done for various reasons: If the Pipeline Bus is operated at 40 MHz, its bandwidth is not sufficient for all the data (though it could be operated at a higher frequency). If the data were not compressed, more serial links and inter-crate cables would be necessary for transmitting them to the DAQ. Moreover, implementing simple compression algiritims in hardware imposes little extra cost, in design effort or performance. The DAQ would in any case compress the data before storing them finally. Pre-compressing them at an early stage in the data stream makes transmission easier and adds to the total compression rate.

1.2.4 Prototypes for Components of the Preprocessor

To establish experience in design techniques and to demonstrate feasibility, prototypes of all the main components of the Preprocessor were developed. The prototype of the Preprocessor ASIC, called Front-End ASIC, was successfully operated on a 'demonstrator front-end module'. An MCM carrying ADCs, Front-End ASICs and serial transmitters was developed, and its trigger data path was tested successfully (see [Pfei]). The prototype performing the function of the RemFPGA is an ASIC since FPGAs were too small and too expensive when it was designed. It is called the Readout Merger ASIC (RemAsic) and is the subject of this thesis.

The RemAsic, the Preprocessor Demonstrator MCM and the Front-End ASIC are described in detail in the next chapter. The VME-based test system used to build up a prototype Preprocessor system is treated in Section 3.1.

Chapter 2

The RemAsic and Associated Hardware

The Readout Merger ASIC (RemAsic) is a prototype designed for compression and readout of calorimeter data. It implements four different compression algorithms for evaluation. It also contains some features specifically intended for testing and debugging.

The RemAsic receives its data from four other ASICs on a Multi-Chip Module. These so-called Front-End ASICs are prototypes of ASICs that do most of the preprocessing for the Level-1 Trigger.

2.1 The RemAsic

The RemAsic consists of about 160 000 transistors and 2.5 kBytes of memory. It was designed using the hardware description language Verilog, synthesized into a schematic circuit represention with the tool Synergy and transformed into a layout. The manufacturing process was a 0.7 μm-CMOS process from Atmel ES2. The Verilog source code is still in existence and was used for finding the root cause of any misbehaviour the ASIC exhibited.

A detailed account of how the RemAsic works can be found in its user manual [RemA]; for a thorough description of its compression unit see [Nie].

2.1.1 Functional Overview

A block diagram of the RemAsic is shown in Figure 2.1. Data flow is indicated by arrows. During normal operation, the readout data path is active. Up to four Front-End ASICs (or FeAsics, see 2.2.2 below) can be connected to each of the four ports on the left. Whenever the Level-1 Accept signal is activated, up to eight 23-bit words are read from each of them. Those data words contain ten-bit values representing the energies of trigger towers at successive bunch-crossings, which are written to one of the two Event Buffers.

When the Event Buffer is filled, the compression unit compresses the data and writes the result (plus one header word) to one of the 32 bit wide Pre-Main Buffers. The different compression modules produce output words of different widths, none of which is 32 bits. The output words are mapped to 32 bits with a barrel shifter, placing each to the left of its predecessor and breaking the resulting bit stream up into 32 bit portions. Data are coded *en bloc* for each FeAsic channel, ie first all values from the first channel, then all from the second and so on.

The compressed 32-bit data are transmitted on a parallel bus, the Pipeline Bus (see below for more detail). It is used for controlling the RemAsic with command words as well as the transmission

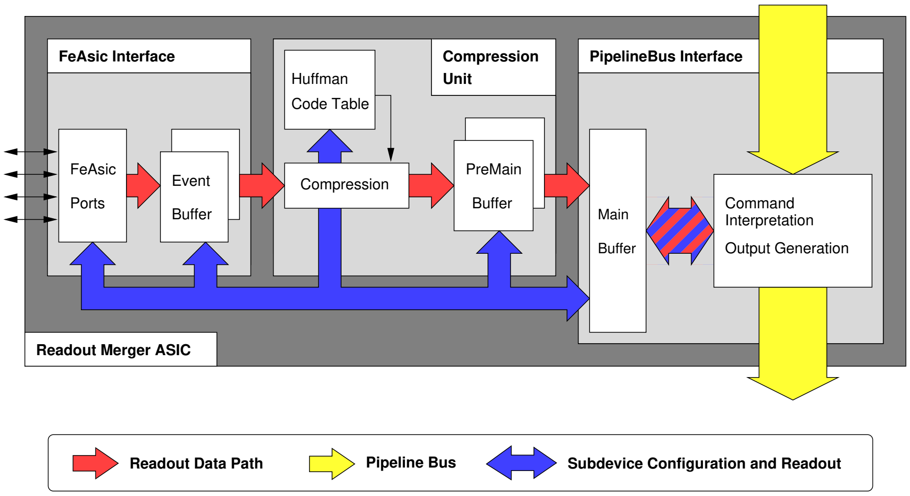

Figure 2.1: Block diagram of the RemAsic [RemA]

of readout data. When a command requesting readout data is received on the Pipeline Bus, it is written from the Pre-Main Buffer to the bus via the general-purpose Main Buffer.

Since there are two Event Buffers and two Pre-Main Buffers, all three stages of processing, ie serial input, compression and parallel output, can be performed simultaneously for data corresponding to successive Level-1 Accept signals. This makes the RemAsic a pipelined system.

When the rate of Level-1 Accept signals exceeds the rate of readout, the RemAsic stores a flag and the corresponding level-1 event number in a FIFO buffer, the level-1 queue. As the corresponding data are stored in a FIFO on the FeAsic, readout can catch up when the accept rate slows down. If the level-1 queue overflows, the RemAsic asserts a signal telling the Central Trigger Processor not to generate any more Level-1 Accept signals.

2.1.2 Subdevices

Subdevices are parts of the RemAsic which can be read and/or configured via the Pipeline Bus in a common way. Subdevices include the Event Buffers, the Pre-Main Buffers, the code table for Huffman compression and the level-1 queue. The set of four control registers that determine the RemAsic's behaviour also represents one subdevice. The serial FeAsic interface is implemented as a write-only subdevice. Certain configuration data words trigger actions by the interface, eg loading the shift register or activating the shift clock.

2.1.3 The Pipeline Bus

The Pipeline Bus is the main interface for controlling the RemAsic as well as reading out event data. Devices connected to this bus are called bus 'nodes'. The Pipeline Bus is so called because data propagate from one bus node to the next at each bus clock cycle, as in a FIFO[1] buffer or pipeline. In the final Preprocessor system, the readout driver module (1.2.3) will both control the bus and receive readout data. Then the Pipeline Bus will have ring topology, ie the last bus node will pass bus data back to the first. In the test system (3.1), the bus is controlled by a master

[1] First In, First Out. This is a buffer that stores data for a certain time before passing them on in the same order. The opposite is a LIFO (Last In, First Out) buffer or stack, from which the last word written to it is read first.

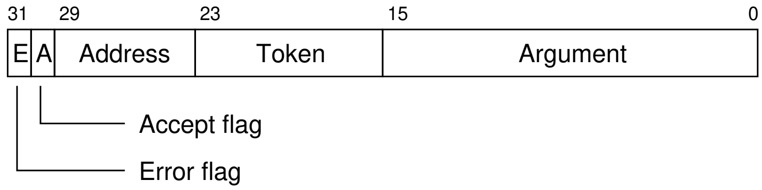

Figure 2.2: Format of the Pipeline Bus command words

node and monitored by a spectator node. As the topology of the bus is of no importance for the RemAsic, the ring was left open for the tests.

The Pipeline Bus has 35 bits. Of them, 32 are data bits, two control bits and one the parity bit. The control lines indicate the meaning of bus data. According to their value, they may be RemAsic commands, user-defined data or 'idle' words. Idle words are transmitted when the bus is unused or as placeholders to be overwritten with readout data. Blocks of user-defined data are enclosed by two special commands called *BeginOfData* and *EndOfData*.

Command words contain an eight-bit command token, a six-bit address, two flags and a 16-bit argument (see Figure 2.2). Each RemAsic has a unique node address and a group address it may share with others. There is also a broadcast address for addressing all RemAsics on the bus. They react to commands containing its node, group or the broadcast address. The two flags are written by the RemAsic in response to the command. When it is addressed by a command word, it sets either the accept flag or the error flag. The accept flag is set if the command could be executed, the error flag otherwise.

The argument field of a command word contains configuration data or status information written by the RemAsic, depending on the command. Commands that require larger amounts of data are followed by a data block, enclosed by *BeginOfData* and *EndOfData* commands. For a description of the command tokens recognised by the RemAsic and their meaning, see [RemA].

2.1.4 Compression Modes

Four different compression algorithms, plus a no-operation mode, are implemented in the RemAsic. Some of them use a special control register, the compression parameter register. Most have the option of using only eight bits of the ten-bit input words. They work as follows:

Run-Length Encoding: Unlike the others, this is an algorithm which entails loss of information. Data values which are less than the compression parameter are suppressed. Instead, a zero is output (a value that is always suppressed), followed by the number of successive values under the threshold.

Huffman Encoding: Data words are coded according to a dedicated memory, the Huffman code table, which contains codes of arbitrary lengths in a certain format. The upper eight bits of the input word are used as an index in the table, the lower two are appended to the code from the table. The table must be configured by the user, for instance with codes generated with the Huffman algorithm (see [Le] for details).

Huffman-Inspired Encoding: This algorithm outputs 13-bit words. If the input words differ from the compression parameter by less than eight, up to three of those differences are stored in one word, otherwise one unchanged input value is contained. The thirteenth bit is used for flagging between those two formats. This mode cannot process eight-bit input data.

Difference Encoding: Same as Huffman-inspired encoding, but the differences between successive data words are used instead of the difference between data word and parameter.

A more detailed description of the compression modes and the compression unit can be found in [Nie].

2.1.5 Features for Testing the RemAsic

The RemAsic contains a number of functions which are not intended for use in the running of the Preprocessor system but for putting the ASIC into operation and testing it. They are described in the following sections.

2.1.5.1 The Serial Pipeline Bus

Running the parallel Pipeline Bus requires several modules, which makes simple functional tests complicated and introduces additional sources of errors. Therefore, the RemAsic contains a serial interface that can receive and transmit Pipeline Bus words. It consists of one serial data line for each direction plus a common serial clock line. On the test system Preprocessor Module (see 3.1), they are connected to the motherboard FPGA, which makes standalone tests of this module possible. The Pipeline Bus clock, which is also the system clock of the RemAsic chip, acts as a frame clock and must be a factor of 35 slower than the serial clock.

The serial Pipeline Bus is not intended for full-speed tests. The RemAsic, including the serial interface, is guaranteed to work only at clock frequency of 40 MHz. The RemAsic's clock is restricted to 40 MHz/35 =1.14 MHz when the serial bus is used, since it is the same as the serial frame clock. Because the FPGA design controlling the Preprocessor Module (see 3.2) runs at only 20 MHz internally, the frequency actually used is half that, 0.57 MHz.

2.1.5.2 Data Buffer Subdevices

All buffers in the data path can be directly accessed via the Pipeline Bus, as subdevices (see 2.1.2 above). When readout does not provide the expected data, intermediate buffers can be read to find out which stage of processing produced the error. Another application of this feature is to test the compression unit without reading data through the serial interface, which was a frequent source of errors. This is done by writing the Event Buffer, triggering compression, and reading the Pre-Main Buffer which stores compression results (Figure 2.1).

2.1.5.3 The Spy Bus

The Spy Bus is the most important debugging feature of the RemAsic. It is the only way to look inside the black box the ASIC represents. When the RemAsic doesn't react to Pipeline Bus input at all, it provides a way of finding out why.

The Spy Bus connects one of eight 8-bit status words to eight output ports. Those status words contain the RemAsic's Pipeline Bus address, state machine[2] registers and other internal registers. The desired Spy Bus word is selected by three address inputs.

2.2 The Preprocessor Demonstrator Multi-Chip Module

2.2.1 Overview

The Preprocessor Demonstrator Multi-Chip Module (MCM) is the predecessor of the Preprocessor MCM to be used in the final Preprocessor system, just as the RemAsic is the preliminary version

[2] A state machine is a piece of digital logic that writes a succession of different values to a state register. Its next value depends on the current value and a number of inputs. Depending on the value of the state register, other actions can be performed by separate logic. The RemAsic contains 32 such state machines. Not all state registers are on the Spy Bus.

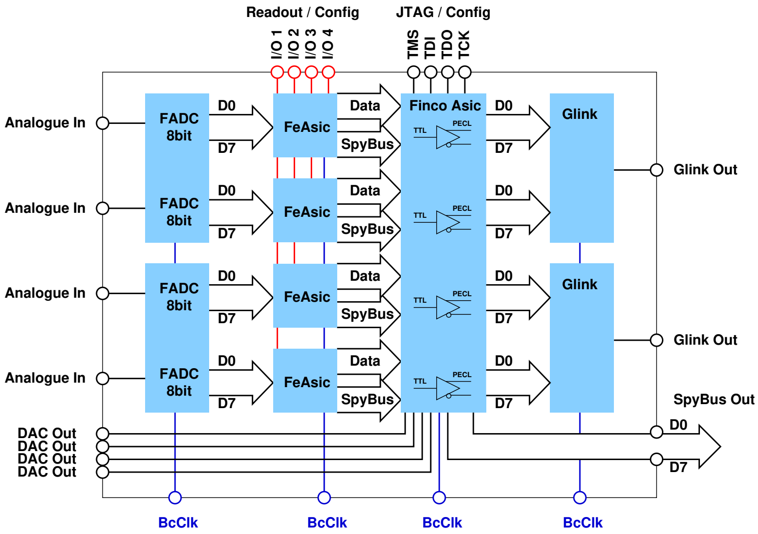

Figure 2.3: Block diagram of the Preprocessor Demonstrator MCM [Pfei]

of a readout chip for that system. It represents the core of the Preprocessor system. It digitises trigger-tower energies, processes those data and transmits the result to other components of the Level-1 Calorimeter Trigger. In addition to this trigger data path, readout data are transmitted to the RemAsic on demand.

The demonstrator MCM can handle four channels of trigger-tower data. Figure 2.3 shows a block diagram of it. It contains two dual-channel flash ADCs, four Front-End ASICs (described below), an interconnection chip (Finco) and two G-Link serial transmitters. The Finco ASIC converts voltage levels and multiplexes the four FeAsics' Spy Buses. The trigger data path runs from the left to the right of the diagram; the readout interfaces are shown above the FeAsics.

The MCM has four different clock inputs so the timing between successive components in the trigger data path can be adjusted. The clock frequencies must be identical, but their relative phase should be optimised to achive maximum reliability on that data path. For most of the RemAsic tests, only the Front-End ASICs were needed, so that timing was irrelevant. The timing is fine-tuned with the help of a delay chip described very briefly below (2.2.3).

For more information on the MCM and the Finco ASIC, see [Pfei].

2.2.2 The Front-End ASIC

The Front-End ASICs (FeAsics) do most of the processing on the demonstrator MCM. They calibrate energy values using a look-up table and locate the maximum of energy peaks, a task known as bunch-crossing identification (BCID). Figure 2.4 shows a block diagram of the FeAsic. The trigger data path is at the bottom, passing through the look-up table (LUT) and the BCID logic. Most of the rest of the ASIC is dedicated to the readout.

Readout data can originate from three different positions in the trigger data path: before the look-up table, before and after BCID. They are copied continuously to the Pipeline Memory. Whenever the FeAsic receives a Level-1 Accept signal, a number of successive values are copied from the Pipeline Memory to the so-called Derandomiser Buffer. This process is in synch with the RemAsic's level-1 accept queue. The RemAsic's handshake with the Central Trigger Processor (see 2.1.1 above) ensures the Derandomiser Buffer does not overflow. When readout data are requested by the RemAsic, the content of the Derandomiser Buffer is transmitted on the serial interface.

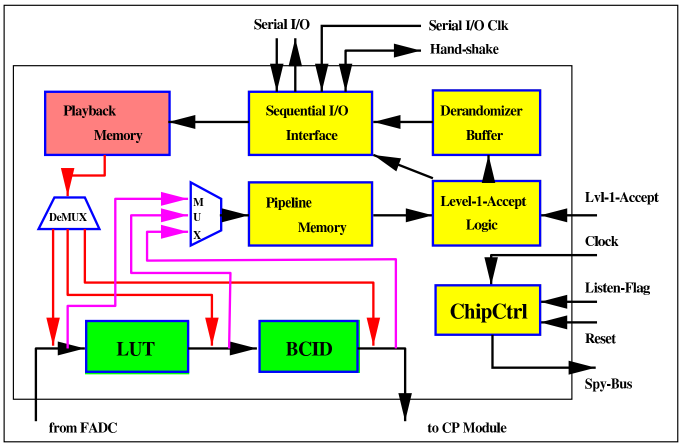

Figure 2.4: Block diagram of the Front-End ASIC [FeA]

The serial interface transmits a 23-bit word, containing a ten-bit energy value, the address of that value in the Pipeline Memory and some flags.

The serial interface between the RemAsic and the FeAsics serves two purposes: transmitting readout data to the RemAsic and configuration data to the FeAsics. It consists of five signals: a serial clock, one data line for each direction, a ready signal and a so-called 'acknowledge' signal. The clock, the acknowledge and one of the data lines are driven by the RemAsic which acts as a master. The acknowledge signal is activated by the RemAsic as a data request for readout or as a load flag to complete a configuration transmission. The ready signal is normally high. It is pulled down by the FeAsic while it is busy transferring data to or from the serial interface internally. This happens after the RemAsic has asserted the acknowledge signal. This procedure is not a handshake since the duration of those signals is fixed by the implementation of the two ASICs.

As a test feature, data from an internal memory, the Playback Memory, can be injected into the trigger data path at any point (see Figure 2.4). Subsequent readout by the RemAsic must produce the pre-defined data. The Playback Memory is written via the serial interface. As a debug feature for checking FeAsic function, the RemAsic can read out the Pipeline Memory address and two mode bits instead of the energy value. They are also contained in the 23-bit word received from the FeAsic. The resulting ten bits are then compressed and put on the Pipeline Bus like an energy value.

2.2.3 The PHOS4 Delay Chip

The PHOS4 chip is not a component of the MCM, but it is needed to operate it. It can impose defined delays on up to four signals. When it is used with the MCM, these signals are the clocks for the different MCM components. The delays are adjusted so that the timing between successive elements in the MCM's trigger data path is right. When only the Front-End ASICs are used (as was the case in nearly all RemAsic tests) that timing is irrelevant. But the PHOS4 still has to be reset and configured to let the signals through at all.

The PHOS4 is controlled via an I^2C bus, a two-line open-collector bus developed by Philips Semiconductors (see [Phi]).

Chapter 3

Test Hardware and FPGA Designs

An existing test, evaluation and demonstrator system based on a multi-purpose VME motherboard was used for all tests. Its purpose is testing prototype components of the Preprocessor system and building a prototype Preprocessor.

An FPGA on that motherboard served as controller of the RemAsic and other hardware. Two FPGA designs were written to do different kinds of tests.

3.1 The VME-Based Test System

3.1.1 General

The VME test system was designed as a flexible platform for doing a variety of hardware tests. Its main purpose is testing the RemAsic and the Preprocessor Demonstrator Multi-Chip Module (2.2.1) and building a demonstrator system with them. This system has the functionality of the Level-1 Trigger Preprocessor System but cannot handle the number of channels. It has also been used for testing serial transmission from the Preprocessor to the components downstream in the trigger data path.

The core of the system is a 6 U VME motherboard (Figure 3.1). It carries an FPGA[1] made by Xilinx Inc. and 32 kBytes of dual-ported RAM. The memory can be read and written by the FPGA or via the VME bus. It contains two special addresses that assert an interrupt signal to one side when being written by the other. The VME interface for configuring the FPGA and accessing the memory is implemented in an EPLD[2] also made by Xilinx. Besides, the motherboard carries a programmable clock generator that clocks the FPGA (and through it all other hardware). It has two different clock outputs with frequencies adjustable from 0.4 to 120 MHz. It is controlled via the VME bus, by writing a register decoded by the EPLD.

Most of the space on the motherboard is taken up by two slots for CMC daughterboards. Five connectors link them to the motherboard FPGA and to each other. Four of them are conform to the standard for common mezzanine cards (CMCs). The fifth is compatible with the CERN specification of a high-speed serial link ('S-Link') daughtercard [SLi].

For each set of daughtercards, there is a corresponding FPGA design for controlling them. Designing a configuration for an FPGA is very similar to designing a digital ASIC (though FPGA designs tend to be smaller). The design's functionality is described in a hardware description

[1] An FPGA (Field Programmable Gate Array) is a piece of configurable hardware. It contains logical gates and flip-flops that can be connected to perform all sorts of functions, in the most general sense. In many cases, an FPGA is an alternative to a digital ASIC (Application-Specific Integrated Circuit).

[2] An EPLD (Electrically Programmable Logical Device) is a device similar to an FPGA but less complex.

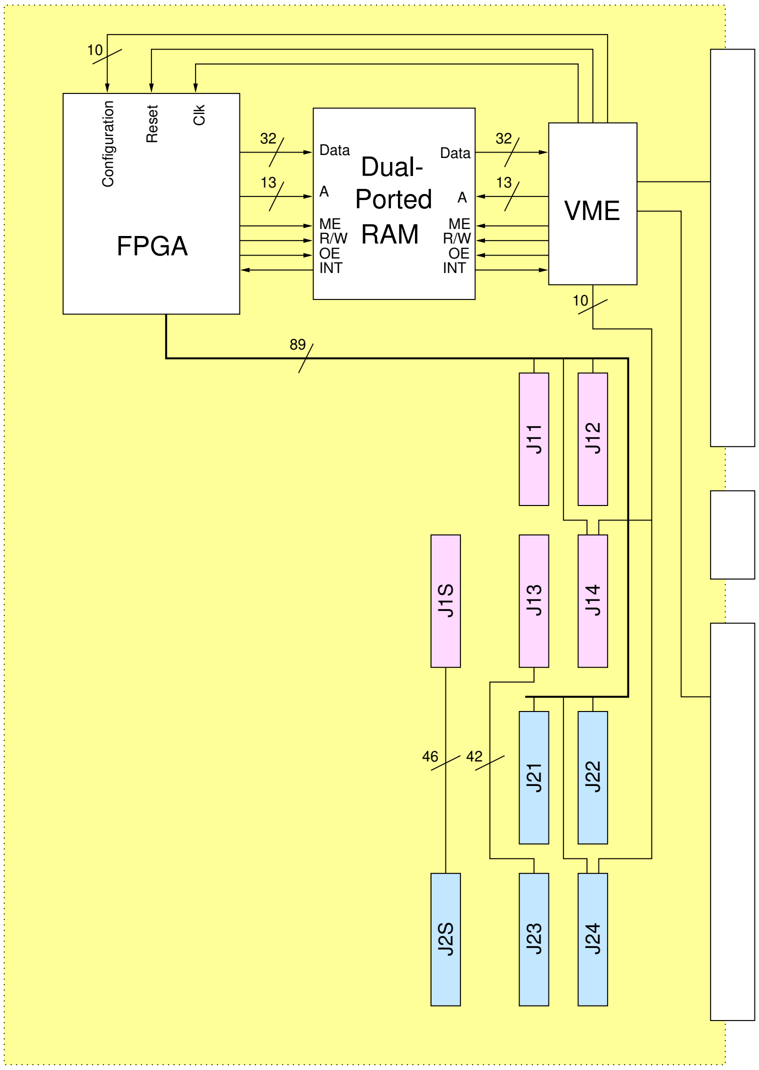

Figure 3.1: The test system motherboard (after [Schu2])

language (in this case Verilog). This description is transformed to a schematic circuit representation and optimised with a tool program, in our case Synopsys. A different tool (made by Xilinx) maps the circuit onto the cells of the FPGA and generates a configuration file that can be loaded into the FPGA.

A motherboard with one or two daughtercards and the right FPGA design forms a VME module that performs a specific function. The three types of modules needed for RemAsic tests are described in the following sections.

3.1.2 The Pipeline Bus Master and Control Signal Source Module

This module has a dual function: It acts as a master controlling the Pipeline Bus, and it provides global signals to all modules in the system. Both the Pipeline Bus signals and the control signals are generated in the motherboard FPGA. The module has two daughtercards shown in Figure

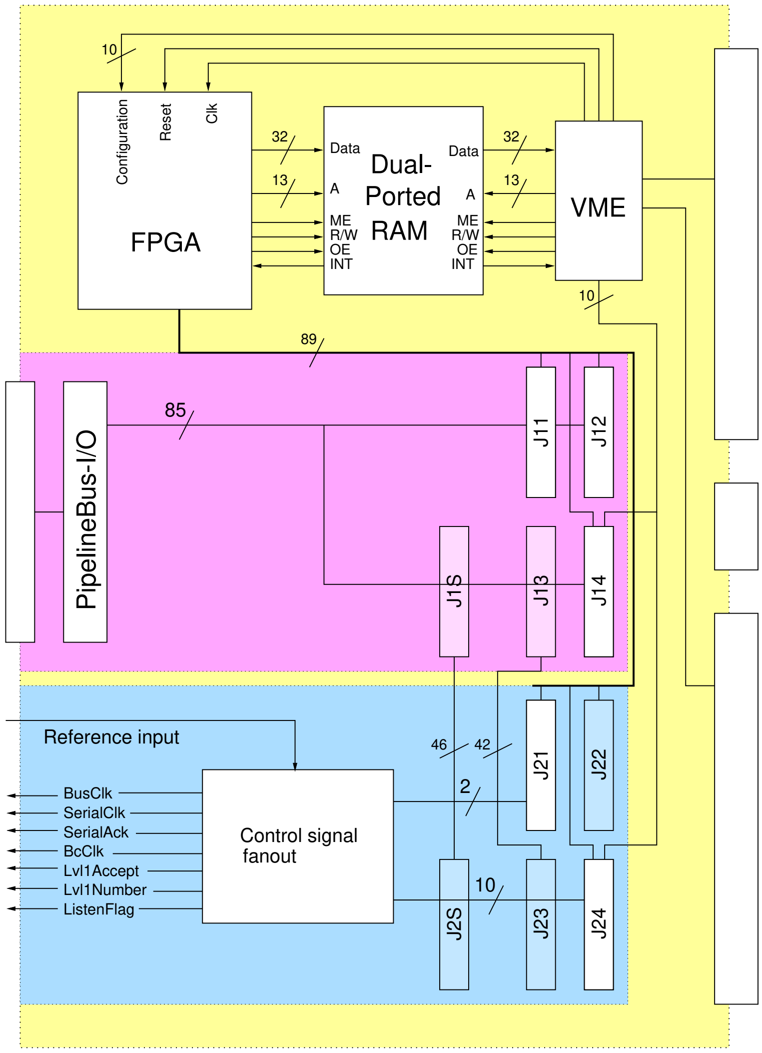

Figure 3.2: The Pipeline Bus master and control signal source module (after [Schu2])

3.2. The upper one links the motherboard FPGA to the Pipeline Bus. As all other daughtercards using the Pipeline Bus, it has two Compact PCI connectors at its front (left side of the figure). Successive Pipeline Bus nodes are linked by plugging small PCB bridges with pins on them into the connectors.

The second daughtercard carries an EPLD which is programmed to provide a number of control signals. In the final Preprocessor system these signals would originate from the Trigger, Timing and Control system. Each of them is fanned out to several pins of a front connector. From there, they are transmitted via cables to all the modules in the system. To make transmission safer, the signals are differential. The two most important signals, the Level-1 Accept signal and the bunch-crossing clock which is also the Pipeline Bus clock, are provided by the motherboard FPGA.

The Pipeline Bus master design with which the FPGA is configured contains all the functionality of this module. The daughtercards only serve as conduit to the Pipeline Bus and fanout, respectively. Acting on a program in the memory, the master design can send a string of Pipeline Bus commands and activate the Level-1 Accept. For a more detailed description of the Pipeline

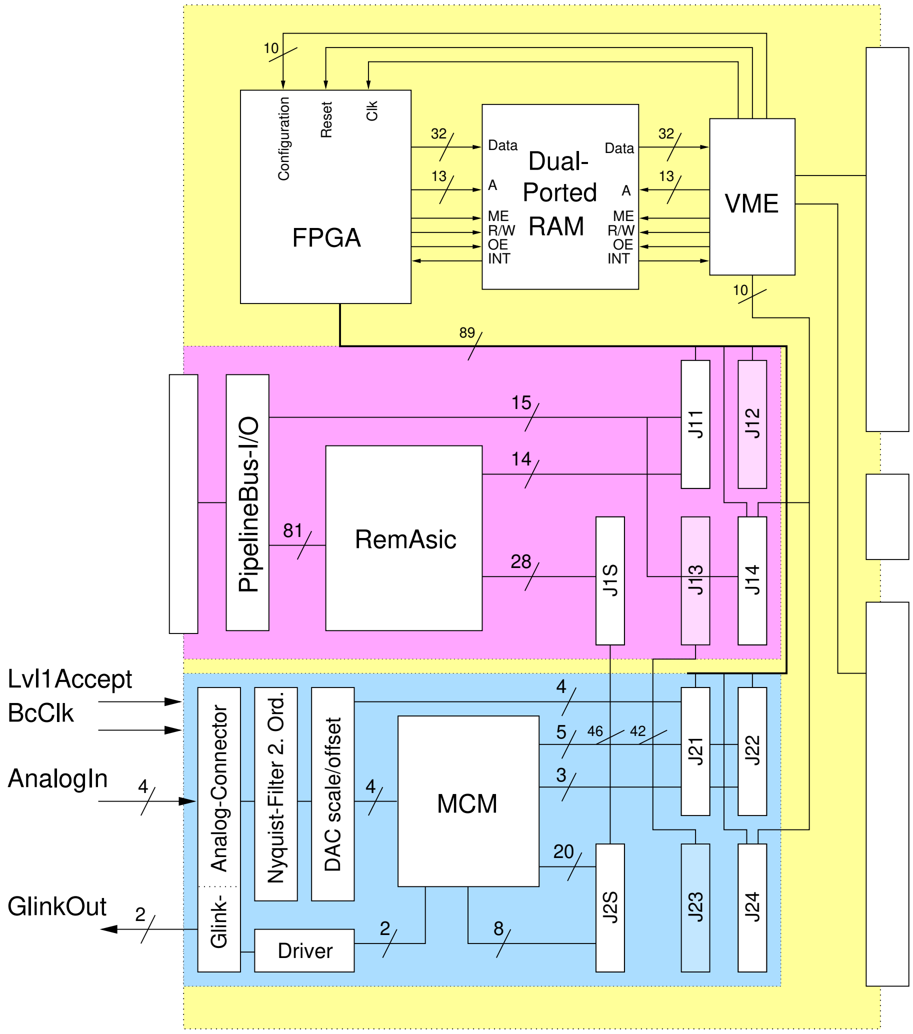

Figure 3.3: The Preprocessor Module

Bus master design see a forthcoming diploma thesis [Ste]. It works very similarly in principle to the Preprocessor Controller design described below, which has some different functions but is largely compatible.

3.1.3 The Pipeline Bus Spectator Module

This module serves as a monitor for the Pipeline Bus. The motherboard carries just one daughtercard in the upper slot, the same as the Pipeline Bus master module. It connects the Pipeline Bus lines to the motherboard FPGA which is configured to write any Pipeline Bus data to the memory. From there, it can be read via the VME bus.

3.1.4 The Preprocessor Module (after [Schu2])

The Preprocessor Module of the test system performs the same tasks as the Preprocessor Module in the final Preprocessor system (1.2.3). Shown in Figure 3.3, it carries two daughtercards, one

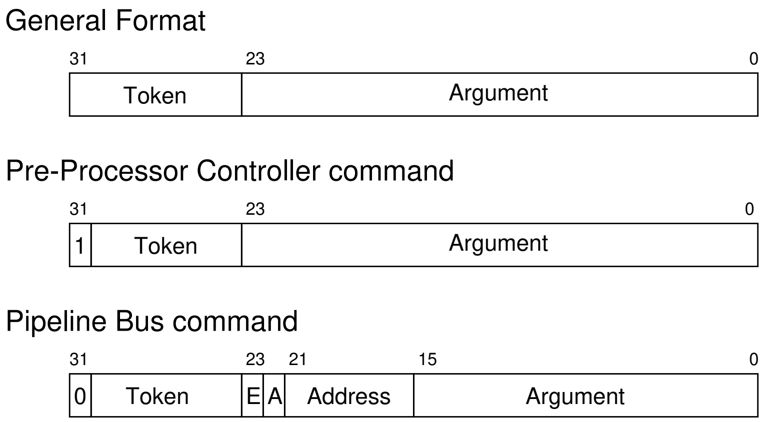

Figure 3.4: Format of the Preprocessor Controller commands

with the RemAsic on it and one with the demonstrator Multi-Chip Module (MCM, see 2.2.1) mounted on top.

The RemAsic daughtercard has at its front (left-hand side of the figure) two Compact PCI connectors for the Pipeline Bus, as all other cards using that bus. It carries a number of multiplexers that allow feeding in control signals generated by this module's FPGA, rather than from the master/fanout module. These control signals include the Pipeline Bus clock, bunch-crossing clock and the Level-1 Accept. The serial Pipeline Bus lines (2.1.5) are also multiplexed. When they are connected to the FPGA, the Preprocessor Module can be operated stand-alone, albeit at a low clock frequency.

The lower daughtercard carries the demonstrator MCM and a signal delay chip (2.2.3) needed to operate it. Separately from the other daughtercard, it has inputs for the bunch-crossing clock and the Level-1 Accept signal. As above, those signals can come from the master module (via front-panel connectors) or from this module's FPGA. The MCM daughtercard also has front-panel connectors for the analogue input signals to the MCM and its serial digital output.

The FPGA design associated with this module is the Preprocessor Controller design described below. It controls the multiplexers on the upper daughtercard and generates control signals when the Preprocessor Module is operated stand-alone. It has functions for accessing all hardware on the daughtercards, including the RemAsic, the delay chip and the MCM.

3.2 The Preprocessor Controller

3.2.1 Introduction

This FPGA design contains functionality for accessing all the hardware on the Preprocessor Module. It is itself controlled via the VME bus. Command words of a certain format can be written to one of the dual-ported memory's interrupt registers from the VME side. On receiving the interrupt signal from the RAM, the Preprocessor Controller reads the register and executes the command. In this context, that interrupt register is called the 'command register'.

The command format is shown in Figure 3.4. The most significant bit distinguishes between commands for the Preprocessor Controller proper and commands to be transmitted on the serial Pipeline Bus. The seven next most significant bits contain a command token that specifies the action to be taken. Valid command tokens for the Preprocessor Controller are listed in table 3.1. The actions they trigger are described below. In the Pipeline Bus commands, the token and

Name	Binary Value	Hex. Value	Brief Description	See Section
Nop	1000 0000	80	No operation	
SetI2C	1000 0001	81	Set I^2C bus lines	3.2.5
LoadFinco	1000 0010	82	Send data to Finco	3.2.1
Reset	1000 0011	83	RemAsic and other resets	3.2.4
Execute	1000 0100	84	Execute RAM program	3.2.1
Stop	1000 0101	85	Stop program execution	3.2.1
Wait	1000 0110	86	Wait for some clock cycles	
CtrlReg	1000 0111	87	Set control register	3.2.2
StartPipeOut	1000 1000	88	Start Pipeline Bus readback	3.2.3
StopPipeOut	1000 1001	89	Stop Pipeline Bus readback	3.2.3
PipeWait	1000 1010	8A	Wait for some PipelineBus slots	3.2.3
PipeIdle	1000 1011	8B	Send some idle words	3.2.3
PipeData	1000 1100	8C	Send some data words	3.2.3
JtagCtrl	1000 1101	8D	Load JTAG controller commands	3.2.1
Lvl1Accept	1001 0000	90	Generate one Level-1 Accept	3.2.4
GetSpyBus	1001 0001	91	Get RemAsic spy bus word	3.2.4
(miscellaneous)	0xxx xxxx	0-7F	Pipeline Bus commands	

Table 3.1: List of Preprocessor Controller command tokens

address-and-flags bytes are exchanged relative to the Pipeline Bus format (Figure 2.2) so the token is at the same position as in Controller commands.

Besides executing commands written to the command register, the Preprocessor Controller can execute a sequence of command words at any location in the RAM. When a string of commands have to be excuted, the command register should never be used, as commands may be lost if the frequency of writing is too high. The command *Execute* starts execution of a program at the address given by its argument. The program must end with a *Stop* command which stops program execution. It may contain further *Execute* commands for jumping to different memory locations. If the command register is written during program execution, the command from the register is executed in between program commands. A program can be aborted by writing *Stop* to the command register.

The main function of the Preprocessor Controller is controlling the serial Pipeline Bus that the RemAsic is connected to when the Preprocessor Module is operated stand-alone (see 2.1.5). It acts as a bus master and generates both the serial and the 'bus' clock, which is both the frame clock[3] for the serial transmission and the system clock for the RemAsic chip. It transmits commands and data to the RemAsic and receives its Pipeline Bus output. It recognises the Pipeline Bus command *BeginOfData* which initiates a user data block. For the Controller, this command must contain in its argument field the number of following data words. This is necessary for it to set the Pipeline Bus control lines correctly for data words and to continue with command words at the right point. Data words are output on the Pipeline Bus as data and not interpreted as commands. When no Pipeline Bus command or data word is given by the program, idle words are transmitted.

The Preprocessor Controller also contains functionality for accessing the interconnection chip (Finco) on the Multi-Chip Module (MCM, see 2.2.1). The command *LoadFinco* transmits a 13-bit word to it via its serial interface. However, due to problems with flip-chip bonding, on all MCMs the reset pin for that interface was not connected, activating it permanently. Therefore, the interface could not be used. The other command concerning the Finco is *JtagCtrl* which permits

[3]The frame clock of a serial interface indicates when the transmission of a data word is complete. Its rising edge causes the data from the shift register (which receives them) to be passed on to the next stage of processing.

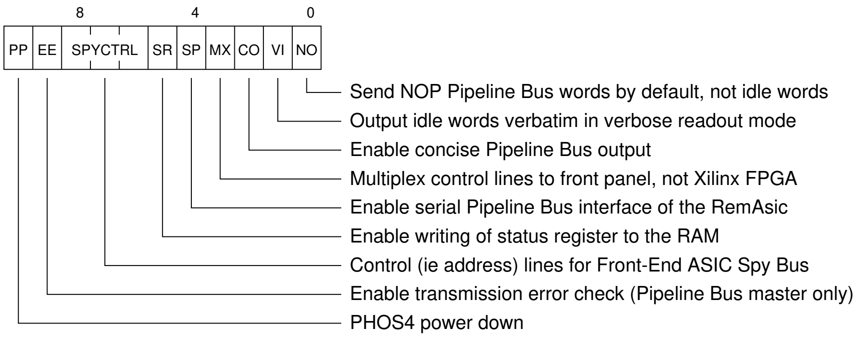

Figure 3.5: The Preprocessor Controller control register

accessing the Finco's JTAG interface[4]. As it is of no importance for the RemAsic tests, it will not be described here.

For executing tests of the RemAsic and controlling auxiliary hardware, many new features had to be added to the Preprocessor Controller design. The design became too large to be fitted into the FPGA. Even after removing the JTAG controller which needs a lot of space, the design could not run at a clock frequency of 40 MHz.[5] The problem was solved by dividing the clock by two internally. This makes the RemAsic clock another factor of two slower when the serial Pipeline Bus is used (0.52 instead of 1.14 MHz), but in any case it is only intended for preliminary tests. The new features are described in the following.

3.2.2 Control and Status Register

The Preprocessor Controller now has a control register which modifies the Controller's behaviour. In addition, some of its bits are directly connected to outputs. The register is shown in Figure 3.5. It is written with the command *CtrlReg* and can be read as part of the status register.

The PP bit of the control register was implemented to get round an inconvenience inherent in the design of the PHOS4 delay chip (see 2.2.3). It can be reset only once after power-up. if it hangs up during operation, as happened occasionally during testing, the whole crate has to be powered down. Therefore, a transistor was added to the CMC board carrying the PHOS4 which switches all its power lines. It is controlled by the control register bit in the FPGA. As we discovered later, the fix is useless since the power is pulled up by the PHOS4's inputs such as clocks and I^2C bus lines. (All input pads contain diodes that prevent the input voltage from rising above the power voltage. When power pins are not connected, current will flow through them and pull the power lines up.)

The EE bit is only used by the Pipeline Bus master design which has the same control register format except that some other bits are unused.

The three address bits for the Front-End ASIC Spy Bus select one of the eight Spy Bus words provided for debugging. They are controlled by the FPGA even though the Spy Bus data lines are read by the RemAsic.

Setting bit five of the control register (SR) enables the writing of a status register to the RAM.

[4]The JTAG interface is a serial interface that permits accessing a chip's input and output pads and internal registers for testing purposes (or those of several daisy-chained chips). In the Finco, only the input pads are included. JTAG stands for the Joint Test Action Group of companies that initiated the creation of this test interface. The acronym is now synonymous with the IEEE standard for it. See [Pa] for more information.

[5]In FPGAs, routing resources (the wires connecting logical gates and flip-flops) are limited. Therefore, fitting a large design in an FPGA can involve roundabout routing which increases delays and lowers the maximum clock frequency.

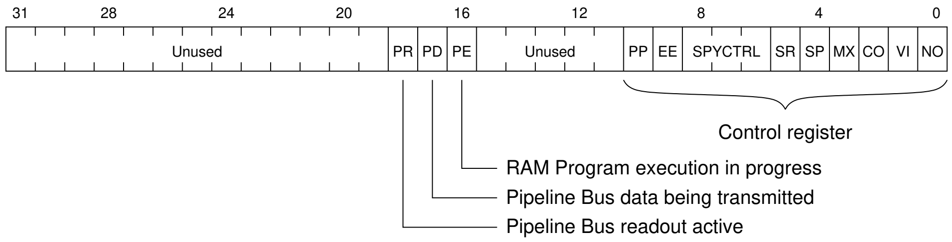

Figure 3.6: The Preprocessor Controller status register

The SP bit is directly connected to the RemAsic input enabling its serial Pipeline Bus mode. Bit three (MX) controls the multiplexers for the control signals and serial Pipeline Bus lines (see description of the Preprocessor Module in 3.1.4). The other three bits modify the behaviour of the Pipeline Bus interface and are described in the next section.

To make it easier to check what the Preprocessor Controller is doing, a status register was implemented. It is written to the RAM whenever a bit in it changes, providing this procedure is enabled. Its memory location is the second interrupt register of the dual-ported RAM so a VME bus interrupt can be raised every time it is written. The status register contains the control register in its lower eleven bits plus three additional status bits. The most important of them is the PE bit. It is set while a RAM program is being executed, so it is possible to find out whether a program has terminated. The PD bit indicates whether user data are being output on the Pipeline Bus. The PR bit is set when received Pipeline Bus words are written to the memory. This can be switched on and off with special commands, see next section.

3.2.3 Extensions of the Pipeline Bus Interface

The NO bit of the control register (Figure 3.5) affects the Preprocessor Controller's Pipeline Bus output: When it is set, no-operation Pipeline Bus command words are transmitted unless a command says otherwise. Normally, idle words are sent in that case, but they may be overwritten with readout data. In some cases, that is undesirable.

There are some special commands concerning Pipeline Bus output. *PipeIdle* transmits the number of idle words given in its argument, even when the control register setting enables no-operation words. *PipeData* transmits a number of data words that follow it, without sending a *BeginOfData* first. That makes it possible to send larger amounts of data or send command words in between a data block. The *PipeWait* command waits for the time it takes to transmit a number of Pipeline Bus words. This is useful because the RemAsic takes some time transferring data to and from its Pipeline Bus interface. Since the RemAsic's system clock is the Pipeline Bus clock, that time is measured in Pipeline Bus words transmitted.

In addition to Pipeline Bus transmissions, Pipeline Bus readout was implemented. This refers to receiving Pipeline Bus words and writing them to the memory. Since the RAM is also used for other status information and for programs, it is divided up into dedicated regions. Figure 3.7 shows a memory map. The memory region for received Pipeline Bus words starts at address 0 and extends up to address FFF (hexadecimal).

The readout is switched on with the command *StartPipeOut* and deactivated with *StopPipeOut*. When the end of the readout memory region is reached, it stops automatically. By default, two 32-bit memory words are written for each received Pipeline Bus word so all 35 bits from the Pipeline Bus are present. If control bit one (VI) is set, idle words are written in the same way. If not, they are just counted and their number is written in a special pair of readout words.

When the CO bit in the control register is set, just the 32 Pipeline Bus data bits are written, one memory word per Pipeline Bus word. The upper two bytes of command words are swapped,

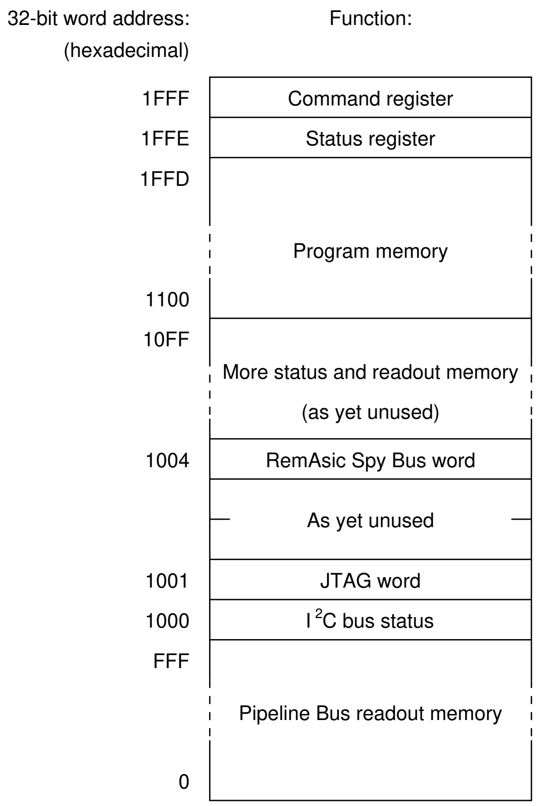

Figure 3.7: Memory regions used by the Preprocessor Controller

resulting in the same format as in Controller programs. This makes the format more readable. Idle words are ignored in this concise readout mode.

3.2.4 Additional Features Concerning the RemAsic

The command *GetSpyBus* makes the Controller read the RemAsic Spy Bus lines. They are written to the RAM at the location shown in Figure 3.7. One of the eight Spy Bus words is selected by the command's argument.

The *Reset* command can trigger many kinds of reset depending on the bits set in its argument field. The only one relevant in this context is the RemAsic reset corresponding to bit 0. The RemAsic reset signal is activated for 64 cycles of the internal Preprocessor Controller clock. That makes sure it is active for more than one RemAsic clock cycle even in the serial Pipeline Bus mode.

The command *Lvl1Accept* generates a Level-1 Accept signal which triggers a readout operation of the system. This affects both the RemAsic and the Front-End ASICs on the MCM. The duration of the signal is one cycle of the bunch-crossing clock. Since the level of the signal is sampled at the positive edge of the clock, it has to be activated and deactivated at the negative edge. This command will have any effect only if the multiplexers are set to local standalone operation of the Preprocessor Module. Otherwise, Level-1 Accept signals are generated by the Pipeline Bus master design and transmitted to the Preprocessor Module via differential cables.

3.2.5 I^2C Bus Interface

The Controller design contains a very simple I^2C bus interface. It is used to access the PHOS4 delay chip on the lower CMC daughtercard of the Preprocessor Module.

The Controller has two open-collector ports connected to the two bus lines. The command *SetI2C* tells it whether to pull them down or not. Immediately after updating the state of the output drivers, the state of the lines is read and written to a special address in the RAM (see memory map, Figure 3.7). The I^2C bus protocol has to be handled by software (see 4.3.5).

3.3 The Front-End Data Source

Because of problems with the Front-End ASICs on the Multi-Chip Module, it was not possible to read out digitised data from them at a 40 MHz clock rate. To get round that problem, a new FPGA design was created. It feeds data into the RemAsic's serial interfaces which are usually connected to the Front-End ASICs (FeAsics). When it is used, the Preprocessor Module has only the upper daughtercard with the RemAsic. A soldered adaptor replaces the lower daughtercard. It links the 'S-Link' connector (J2S in the figures) to the connector number two (J22). The adaptor is required because the serial interface lines are on the S-link connector which is not linked to the FPGA.

Every time the RemAsic requests readout data, the Data Source FPGA design reads four values from the memory. The upper 22 bits of the 32-bit values are discarded. The remaining ten-bit words are transmitted to the RemAsic as energy values, one for each serial interface. The design statically sets the multiplexers on the RemAsic daughtercard so that the parallel front-panel Pipeline Bus is enabled. It resets the RemAsic whenever it is reset itself. Writing to a special register on the VME bus resets the RAM read address to the beginning.

Chapter 4

Software Development

Both the test system and the final Preprocessor system have to be controlled by software. An existing hardware diagnostic software framework was greatly extended, regarding both features of the core program and functionality for accessing certain pieces of hardware. The immediate purpose for this was the RemAsic tests, but the software will also be used for setting up the Preprocessor in the ATLAS experiment. Parts of it will be integrated into the experiment's data acquisition and run control software.

4.1 Object-Oriented Programming for Hardware Access

The test software in question is called HDMC, for Hardware Diagnostic, Monitoring and Control software. Figure 4.1 shows a screenshot of some of its windows. It was written in C++, an object-oriented programming language which was developed from a non-object-oriented programming language, C, by Bjarne Stroustrup at AT&T [Strou]. This section will give an introduction to both C++ and the test software.

The central concept of C++ is the *class*. A class is a variable type, usually a composite type like a *struct* in C or a *record* in Pascal. The notion of a class also encompasses a set of subroutines that use that type of variable and perform functions that are closely related to the 'meaning' (in the mind of the programmer) of the type. When a variable of a certain class is created in memory, it is called an *instance* or *object* of the class, hence 'object-oriented'.

What makes the class concept powerful is *inheritance*. A class can 'inherit' the sub-variables (called *member* variables) and subroutines (*member* functions) of another class. It is then called a *subclass* of the other class (the *superclass*). The subclass can add member variables and functions of its own; it can also replace member functions of the superclass by its own special implementation. It is possible for a superclass to offer no implementation of its member functions at all, requiring the subclasses to implement them. It is then called an *abstract* class. Objects of a subclass can be used as objects of their superclass since they have all the functionality; the part of the program using them does not even have to know about their exact type.

Let us consider a piece of software that accesses hardware via the VME bus, on different systems. There is a variety of VME bus computers with different interfaces to the VME bus, but they all provide the same functionality, ie reading from and writing to an address on the VME bus. That problem lends itself well to an object-oriented approach: A VME bus superclass defines read and write funtions. Their implementation is done in subclasses that correspond to the different VME bus interfaces. The class diagram of VME bus classes in HDMC is shown in Figure 4.2. The arrows point from the subclass to the superclass. The class hierarchy is larger than was said above: The VME bus class is derived from a bus class. It is the superclass of classes that do

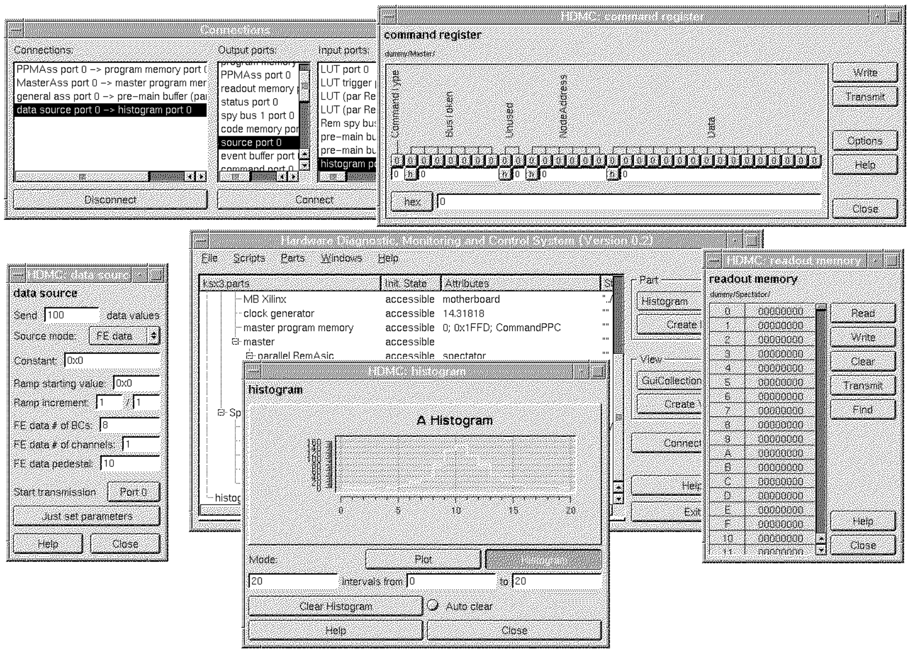

Figure 4.1: Screenshot of the HDMC software

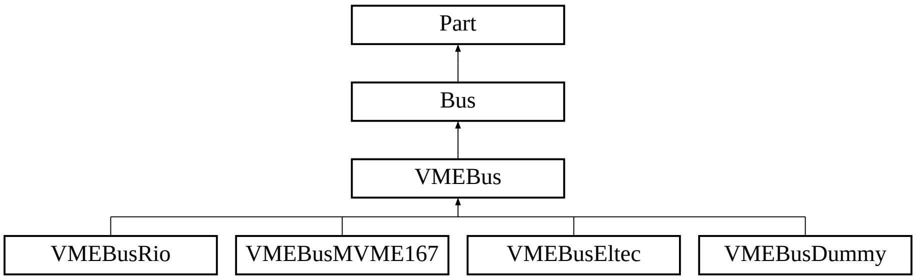

Figure 4.2: Class diagram of VME bus access classes

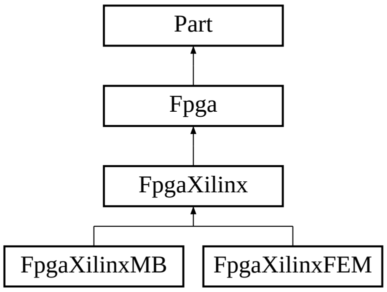

Figure 4.3: Class diagram of Xilinx FPGA classes

not necessarily access VME bus interfaces but offer the same functionality. The `Part` class is the superclass of all hardware access classes in HDMC.

The situation is similar for FPGAs[1]. All FPGAs can be configured, cleared and reset. When a certain type of FPGA is mounted on different boards, it is configured in the same way in principle: A configuration file of a certain format has to be read and its contents have to be written to the FPGA. Only the address the data are written to and the way of starting a configuration process may differ. That is the case for the Xilinx FPGA which is both on the test system motherboard (3.1) and on an older demonstrator front-end module. The hierarchy of the classes for accessing FPGAs is shown in Figure 4.3. The class `Fpga` is the abstract superclass of all classes that control FPGAs. The class `FpgaXilinx` implements a function for configuring Xilinx FPGAs. It calls functions of its subclasses for handling the details.

Figure 4.4 shows all Part classes and I/O frame classes (which are special Parts, see 4.2.6 below). The two most important ones are the `Register` and `Memory` classes. VME bus registers and memories are derived from them, but also on-chip registers and memories which are more complicated to access.

4.2 Features of the HDMC Main Program

4.2.1 `Part` Classes and the Concept of HDMC

In the HDMC software there exists a C++ class derived from `Part` for each hardware component. The Parts are organised in hierarchies of dependencies. These hierarchies have nothing to do with the class hierarchy but illustrate the fact that some `Part` objects need others for working properly. A Part representing a register on the VME bus will need a VME bus Part for accessing its hardware register. Part hierarchies model the structure of the hardware. For example, a memory on an ASIC on a module connected to the VME bus is represented by an object of a `Memory` subclass depending on a class for that ASIC depending on a `Module` class which in turn depends on a VME bus class. The Parts needed by a specific Part type are called its *Part dependencies*.

In addition to Part dependencies, a Part can have *Attribute dependencies*. That are strings containing parameters like addresses and formats of registers or memory sizes. Attributes describing more complex properties refer to the HDMC configuration file. In it properties like register formats are defined.

Part hierarchies are built up by the user of the HDMC software. It offers menu options for creating and deleting Parts and modifying their dependencies (see next section). Therefore it

[1] An FPGA (Field Programmable Gate Array) is a piece of configurable hardware. It contains logical gates and flip-flops that can be connected to perform all sorts of functions, in the most general sense. In many cases, an FPGA is an alternative to a digital ASIC (Application-Specific Integrated Circuit).

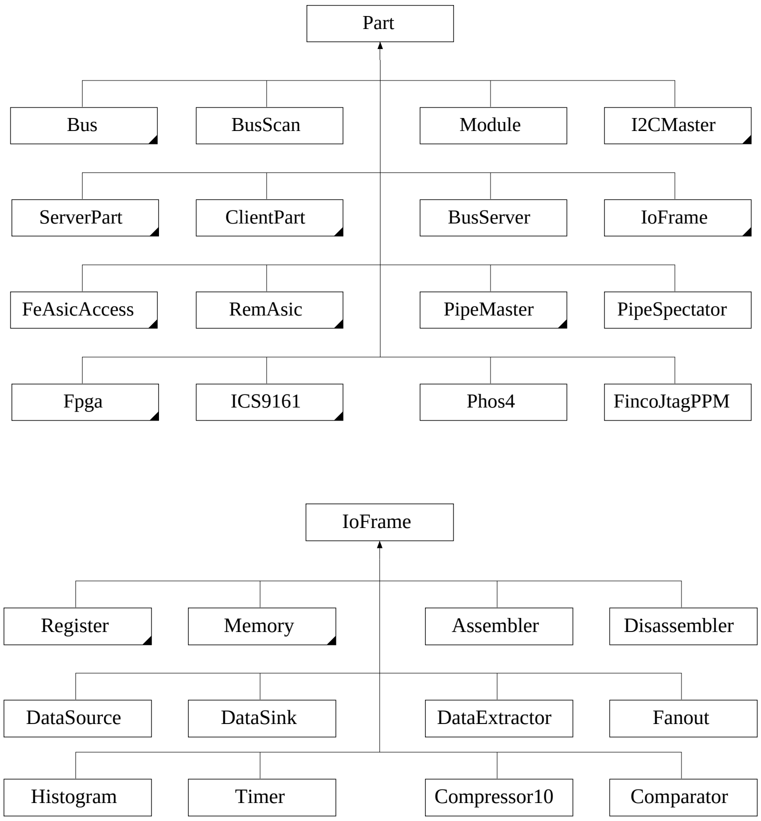

Figure 4.4: All Part and I/O frame classes in HDMC. The black triangle in the lower right corner indicates that a Part type has subclasses of its own.

can be used for all kinds of hardware configurations by persons without programming skills. All existing Parts are displayed in HDMC's main window. Hierarchies can be saved to ASCII files (*Parts files* in HDMC parlance) and restored later. Saved hierarchies can be merged to existing ones.

Most Parts have graphical user interfaces (GUIs), ie windows with buttons and text inputs with which the user can trigger actions. Here again, the object-oriented approach is useful: The GUI classes use only functions of the most high-level superclasses. So the same GUI class can be used for all the subclasses of `Register`, `Memory`, `Fpga` etc. The GUI classes were written using a C++ class library developed by Troll Tech of Norway (see [Qt]).

Outside HDMC, fixed hierarchies of Part classes can be used to access certain hardware. This is done in a server program for accessing hardware via a network connection (see 4.2.10). It is also the planned way to integrate the Preprocessor into the ATLAS data acquisition software. Its run control part will load the Preprocessor with configuration data. It will probably use fixed hierarchies of Parts as hardware drivers.

4.2.2 Handling of Part Hierarchies

A number of user options were added to make building and modifying Part hierarchies more convenient. The user can now create Parts without giving its Part dependencies at once. This is called leaving the dependency 'open'. Such Parts cannot be used for hardware access, but the

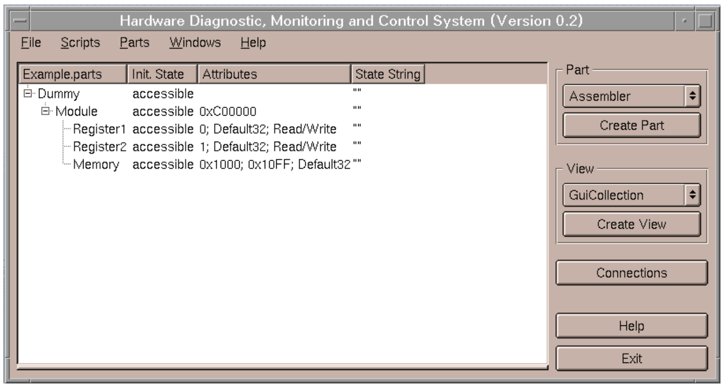

Figure 4.5: HDMC's Main window with the Part display

feature is useful when editing Part hierarchies. The Part dependencies can be created later. When deleting a Part on which others depend, the others can be deleted along with it. Otherwise their Part dependencies are left open.

Some changes have been made to the way Attribute dependencies may be given when creating a Part. Part types can give a default value for Attributes. It is written to the relevant line editor in the Part creation window. A selection box giving a limited number of choices for an Attribute is also possible.

In HDMC's main window, all Parts are represented in a hierarchical display, similar to the way file hierarchies are displayed in some file browsers (see Figure 4.5). Each Part is immediately followed by the Parts that depend on it. They can be hidden by clicking on the field to the left of the Part. Parts that have more than one Part dependency are attached to their first one. Parts with open dependencies are displayed as top-level (like Parts without any Part dependencies). The headline of the first column of the display contains the name of the hierarchy's Parts file.

The other three columns give more detailed information on the Parts. The second column displays a Part's Attribute dependencies. Additional Part dependencies are also listed there. The third shows its *initialisation state* (see next section). Different words are displayed depending on whether the Part and its dependencies are properly initialised. Open dependencies are also noted in this column. The last column of the Part display shows the *state string* of a Part, which is used for saving and restoring the state of the hardware (see next but one section).

4.2.3 Initialisation of Part Objects

Part objects of some types need to initialise themselves by reading or writing the hardware. They do this to synchronise information stored in Parts with the state of the hardware. Originally, this was done when installing a Part's dependencies. This had the disadvantage that Parts files could not be edited if the hardware they were designed for was not available.

Therefore, two functions were added to the `Part` class. They perform or undo any kind of initialisation for which the Part has to use the hardware. Part hierarchies can be loaded and modified without initialising the Parts. The user can initialise single Parts and their dependencies, or the whole Part hierarchy, or a sub-hierarchy. While a Part is not initialised, its GUI cannot be opened and it cannot parse a state string (see below).

The hardware initialisation procedure is used by VME bus Parts for activating the VME bus interface. Parts handling a network connection establish it on initialisation. Some Parts for on-chip memories use it in a slightly different way that does not entail a hardware access. They initialise themselves by allocating buffer memory.

4.2.4 State Strings

State strings are strings containing the state of a piece of hardware. State strings of registers contain their value, those of FPGAs the name of the configuration file for loading the device. State strings can be parsed by the corresponding Part. Because of the object-oriented approach the functions for parsing have to be implemented only in the superclasses. A state string for each Part is saved to Parts files along with the Part hierarchy. That makes it possible to save the state of the hardware with the hierarchy. It can be restored at the next HDMC session.

By way of a context menu, the user can make a Part parse its state string and configure the hardware accordingly. He or she can also update the string from the current state of the hardware. A state string can also be edited directly. It is possible to parse or update the state strings of all Parts in one go. When parsing all strings, a Part is preceded by its dependencies in case a Part can only be used if its dependency is in the right state. The information contained in a state string can also be written to the input elements of the Part's GUI window. The user can then modify the settings before configuring the hardware.

4.2.5 Scripts

In most cases state strings provide a satisfactory way for restoring the state of hardware components. They fail, however, when there are dependencies between pieces of hardware that are not represented in the Part hierarchy. Even when it is possible to represent hardware dependencies accurately in the Part hierarchy, it is sometimes inconvenient because it may lead to very deep hierarchies (ie ones with quite many levels). Scripts offer a way to load hardware in a specific order. They are also a great help for developing Part classes that perform complex operations (like accessing the RemAsic via the Pipeline Bus). Unlike code for the functions of Part classes, scripts can be changed at runtime to try out different approaches.

HDMC scripts can make a Part or its GUI take a certain action. Special script commands permit waiting for some time between actions or repeating the whole script for a number of times. Script lines accessing Parts start with the Part's *path* in inverted commas. In HDMC Part hierarchies, all Part objects have names. If a hierarchy contains several modules of the same type, however, equivalent registers on those modules usually have the same name. A Part's path consists of the names of the Part in question and all Parts it depends on, down from the top level of the hierarchy. The names are separated by slashes. Part paths are unambiguous provided there are no Parts with the same name in the same branch of the hierarchy. This restriction is enforced when creating new Parts and when merging additional Parts to a hierarchy.

For a hardware access, the path is followed by a state string in double quotes. A Part can also be initialised by script, by putting the word `Initialise` in place of the state string. If not the Part itself but its GUI is to take action, the Part path is followed by the word `GUI`. If it is followed by a state string, the information contained in it is written to the GUI's input elements (editors, selection boxes etc.). Otherwise a command word follows. The GUI can be opened and positioned with script commands. It can be told to update the data it displays by reading from the hardware.

The following script uses the register `reg` on the module `mod` on the VME bus `bus`. First, a value is written to it, then its GUI is opened and made to display its actual value.

```
'bus/mod/reg' "0x12345678"
'bus/mod/reg' GUI Open
'bus/mod/reg' GUI Read
```

4.2.6 Data Interchange

For transmitting data between Part objects, HDMC provides *I/O frames*[2]. Part types derived from the class `IoFrame` have ports that can be connected to other I/O frames. Through the ports, they can exchange chunks of 32-bit words. I/O frames can have input, output or trigger ports. Input and output ports are for receiving and transmitting data. A trigger port triggers the transmission of the Part's data (eg the content of a memory) every time it receives a data word.

There are also 'pure' I/O frame classes that do not access hardware themselves. They exist just to generate, display or process data. There is a `Histogram` Part which can plot or histogram data, a `DataSource` for generating defined data (see 4.3.6.1) and a `Timer` which sends single data at regular intervals. Since registers and memories write the hardware on reception of I/O frame data, transmissions from any I/O frame may trigger hardware access.

4.2.7 Register and Assembler Formats

On program startup, HDMC reads a configuration file that contains definitions of data word formats. These formats divide a data word up into several bit fields that can have names. A register's GUI offers separate inputs for the bit fields and displays the names. The `Assembler` class, a pure I/O frame, also uses the formats (see next section). Special values for certain bit fields can also be named. These constants appear in a context menu of the register GUI and can be used in assembler sources.

The data format may be conditional on the value of one special bit field. This is the case for the command format of the Preprocessor Controller (see 3.2). The number and size of parameter bit fields depends on the value of a command token. Both the register GUI and the Assembler can handle such formats. They are also defined in the configuration file.

4.2.8 The `Assembler`

The `Assembler` Part creates binary data from ASCII source files and transmits them to the connected I/O frame. Every line of a source file gives the values of all bit fields in the selected format (see previous section).

The `Assembler` accepts two different source formats which may be used in the same source file. In the more elaborate format, the bit fields are given by assigning a value (or a constant) to the name of the bit field. In the concise notation, just their values are given, from left to right in the data word. If constants are used for the first bit field, this makes the source files similar in appearance to 'real' assembler sources for processors. This mode does not work for all formats.[3] It is distinguished from the long format by a number sign (#) after the first bit field.

The assembler also understands some commands that do not refer to the format it is using. They start with a dollar sign to distinguish them from ordinary source lines. The command "`$Inline`" is followed by a list of values to be transmitted without change, regardless of the format. This was used for generating user data blocks within Pipeline Bus programs. The other two commands, "`$Loop` n" and "`$EndLoop`", define a loop. n is the number of repetitions. To date, there may be only one loop in a source file.

[2]These classes were so called because they originally were wrapper classes that used the Parts, not Parts themselves. We found that making I/O frames special Parts simplified things, but the name has stuck.

[3]There may be a problem when the word format depends on the value of a bit field. If the number of bit fields in the left part of the word depends on the value of a bit field farther to the right, it is impossible to match bit fields with values. Then, the more verbose format has to be used.

4.2.9 Command Line and Program Startup

When starting HDMC from a UNIX shell, a Parts file may be given as a command line parameter. HDMC will then load the Part hierarchy from the file on startup. As a second parameter, the name of a configuration file (the file that contains register formats) can be given. If it is not specified, the name "`default.conf`" is used. It is searched for first in the current directory, then in the user's home directory and finally at the path where HDMC is located. For that last resort, the environment variable `HDMCDIR` has to be set correctly.

Two options can be used to modify HDMC's startup behaviour. The "-i" option prevents the loaded Parts from being initialised automatically. The option "-n" tells HDMC that no Parts file is given in the command line. This is necessary when passing a configuration file but no Parts file.

4.2.10 Remote Hardware Access

HDMC offers a way of accessing hardware on a different computer from the one running the main program. This is necessary if the VME bus computer is too small for HDMC to compile and run on it. It also makes it possible to control hardware in several VME crates from one HDMC window.

There is a small 'bus server' program that runs on the VME bus system. It translates instruction packets it receives via the network into VME bus accesses and sends back data from read operations. In the HDMC program running on a remote computer, a `NetBus` Part replaces the usual `VMEBus`. It uses another Part which is responsible for the network connection.

The bus server program contains a fixed hierarchy of three Parts, one of which is the appropriate VME bus Part for the system. When it is compiled, the right VME bus subclass is used automatically.

When reading a register on a different computer via the network, a message requesting data is first transmitted. Then, the computer running the bus server program will respond with the register's value. Reading a memory word by word in this way is very slow, since for every memory cell network packets have to be exchanged.

To improve this, a block read operation was created. The request packet contains the length of the data block along with the address. Only one packet containing the whole block is sent back. For the sake of completeness, a block write operation was also implemented. Both block accesses can be performed without incrementing the address, ie reading or writing several times to the same address. This is useful for loading FPGAs, where configuration data have to be written to the same address one after the other.

These changes entailed extensions of the `Module` class and all VME bus classes. Since each Part only calls functions of its Part dependencies, block accesses have to propagate up in the hierarchy, eg from a `Memory` Part via a `Module` and a `Bus` to the Part handling the network connection. New functions for block read and write were implemented in those classes.

4.2.11 Online Help

HDMC offers an extensive online help of more than 16 000 words in total. It describes features of the main window and of all Part GUIs. Documentation of all Part classes and the formats of their state strings is included. There are two tutorials: One describes how to build and use an exemplary Part hierarchy. The other one is about remote access of hardware. The register/assembler formats and the assembler source syntax are also treated in the online help.

The online help is written in HTML format. It can be viewed with the HDMC help window which contains a small browser. It can also be read at:

`http://wwwasic.kip.uni-heidelberg.de/atlas/DATA/hdmc/onlinehelp/`

4.3 Part Classes Developed for RemAsic Tests

4.3.1 Components of the Motherboard

A subclass of the class `FpgaXilinx` was created to configure the Xilinx FPGA on the test system motherboard. Since most of the functionality is contained in the superclass, it was sufficient to write some low-level functions. They clear the device or start a configuration process. Separately, the configuring function in the superclass was reimplemented to make use of the new block write functions (see 4.2.10 above).

Another important component on the motherboard is the clock generator made by Integrated Circuit Systems (ICS) Inc. It contains two phase-locked loops (PLLs) that are controlled with a set of registers. One of them has three associated registers, the other only one. Each of the registers contains settings for the divider in the PLL feedback and for an output divider.

Two classes were written for the ICS clock generator: An abstract superclass and a subclass specific for the motherboard clock generator. Since the register settings of the chip allow only a limited number of frequencies, the best match has to be found for each desired clock rate. This is done in the superclass `ICS9161` (named after the number designating the chip). It also implements functions for parsing and updating state strings. The state string for this Part type contains a register number and a frequency. On parsing, the specified register is loaded according to the frequency. If the register is one of those associated with the first PLL, it is also selected for use. Because the registers cannot be read, they are buffered. That way, a state string can be updated from the current state of the device.

The subclass `ICS9161MB` contains low-level functions specific for the motherboard. They can load a register value into the chip and detect transmission errors. On initialisation, a `ICS9161MB` Part sets the clock chip's control register and determines the selected PLL register. Its Attribute dependency is the frequency of the clock quartz used as a reference input to the clock generator. This information is necessary for calculating the right divider factors. The quartz frequency usually used on the motherboard is provided as a default value to help the user.

4.3.2 Pipeline Bus

The RemAsic can be accessed in two ways: with the parallel Pipeline Bus, or via the serial Pipeline Bus that exists locally on the Preprocessor Module. In the former case, the Pipeline Bus is controlled by the master FPGA design on the master module and monitored by a spectator FPGA. There are two Part types called `PipeMaster` and `PipeSpectator` which control those FPGAs. They have no GUIs since they exist exclusively as dependencies for other Parts.

The master Part offers functions for writing a program to the memory or a command to the command register. The binary values of specific command words are returned by functions in the `PipeMaster` class. That way Parts that have a `PipeMaster` as a dependency (such as the `RemAsic` class) do not have to know about the command format. In the future they could work equally well with a different Pipeline Bus master using a different format.

The spectator Part class controls the FPGA which monitors the Pipeline Bus. It can obtain the contents of its readout memory and reset its write address counter. The most important task of this class, however, is comparing the received data with those transmitted by the master. It can check whether commands were accepted by a Pipeline Bus node (RemAsic) by looking at the accept flag of those command words (see 2.1.3).

When the serial Pipeline Bus is used, the Preprocessor Controller performs the functions of Pipeline Bus master and spectator combined. It would be logical for the corresponding Part class to be a subclass of both `PipeMaster` and `PipeSpectator`. Though possible in principle, this leads to problems when *casting* that type, ie transforming it into one of its superclasses and

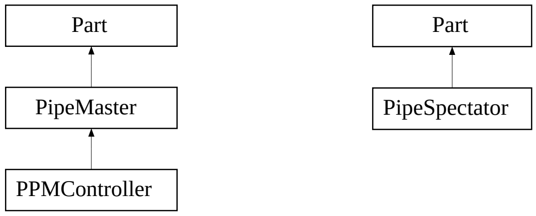

Figure 4.6: Class diagram of Pipeline Bus master and spectator classes

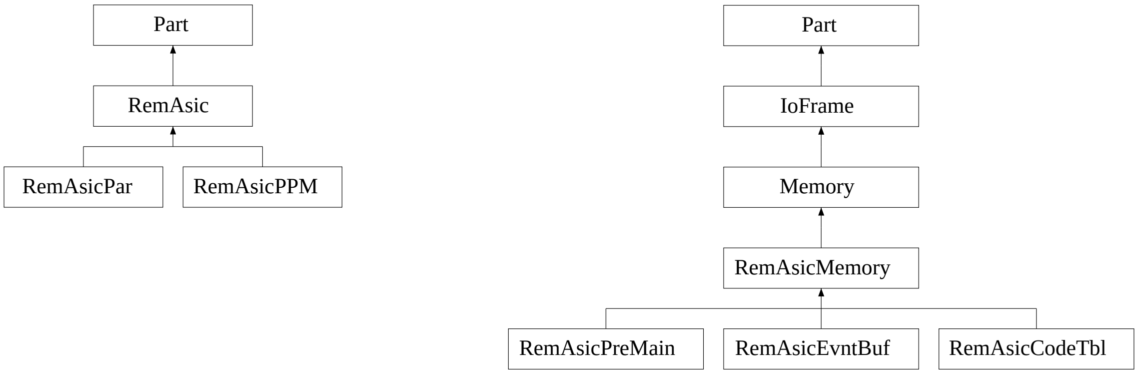

Figure 4.7: Class diagram of RemAsic and RemAsic memory classes

using it as such. So the Part type `PPMController`[4] is derived only from `PipeMaster`. Figure 4.6 shows a diagram of classes related to the Pipeline Bus. The class `PPMController` implements functions for some additional binary commands that the (parallel) master does not support. Just as the `PipeSpectator` class, it can check the accept flags of received Pipeline Bus commands. On initialisation, `PPMController` objects enable the status register write in the Preprocessor Controller (3.2.2, page 23). The status bit indicating program execution is read before a new program is started. This prevents the previous program from being interrupted.

4.3.3 RemAsic

There are two classes for the RemAsic which share a common superclass. One is for accessing it locally on the Preprocessor Module via the serial Pipeline Bus. It is called `RemAsicPPM` and has a Preprocessor Controller Part as a dependency. The other one, `RemAsicPar`, uses the parallel Pipeline Bus and depends on a master and a spectator Part. The class diagram is shown in Figure 4.7.

The RemAsic Parts have a GUI that allows the setting of operation modes and configuring of the level-1 queue in a user-friendly way (see Figure 4.8). For reading the control and level-1 queue registers, there is a separate class derived from `Memory`. It contains the four control registers and the three level-1 queue registers as its memory cells. They are displayed in the memory GUI, which is not so user-friendly but is hardly ever necessary except for debugging the software.

The RemAsic contains three on-chip memories that are implemented as subdevices: the Event Buffers, the Pre-Main Buffers and the Huffman code table. These are represented in HDMC by three classes derived from `RemAsicMemory`. They are shown in the right part of Figure 4.7. As the way of accessing them is common to all subdevices, the superclass contains the relevant functions.

[4] `PPMController` stands for 'Preprocessor Module Controller'

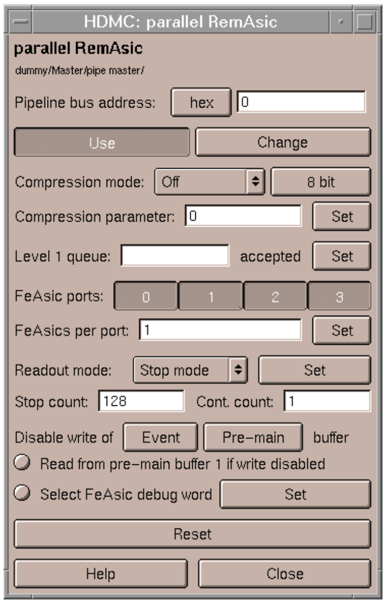

Figure 4.8: The RemAsic GUI

The subclasses just differ in the subdevice number and the size of the memory.

The Part corresponding to the RemAsic Spy Bus is a read-only memory containing the eight Spy Bus words. Its dependency has to be a `RemAsicPPM` since the Spy Bus can be read only with the Preprocessor Controller.

4.3.4 Front-End ASIC

Four Front-End ASICs (FeAsics) are located on the Preprocessor Demonstrator MCM. They are accessed by way of the RemAsic's four serial interfaces. An older module, the Demonstrator Front-End Module, also carries four FeAsics. The HDMC software can access FeAsics in both hardware configurations. This is done by the classes `FeAsicAccessPPM` and `FeAsicAccessFEM`, respectively. Both are derived from the class `FeAsicAccess`.

As usual, the superclass implements high-level functions. These functions set operation modes and configure internal memories. They call functions from the subclasses that handle the transmission of configuration words to the FeAsics. On the Preprocessor Module of the test system, that is done by way of the RemAsic. Its FeAsic interface is implemented as a subdevice and transmits data to the FeAsics when a subdevice configuration is performed. On the Front-End Module, the FeAsics are configured more directly by a controller FPGA.

As in the case of the RemAsic, these FeAsic classes have a GUI that allows the comfortable configuration of control registers. The on-chip memories are represented by classes derived from the `Memory` class. The look-up table and the Playback Memory, can only be configured, not read. The pipeline memory can only be read. The FeAsic Spy Bus was implemented as a read-only memory containing all the Spy Bus words. Accessing it on the Preprocessor Module is rather complex. Both the RemAsic and the Preprocessor Controller are involved. The Controller controls the address lines which have to be set first. Then the Pipeline Bus command *GetSpyBus* is sent to the RemAsic. It reads the FeAsic Spy Bus data lines and writes the result into the argument field of the command word. The word is read back by the Preprocessor Controller, written to the RAM. It is obtained from there by the `PPMController` Part which checks it for errors. If no error occurred, the `FeAsicSpyBus` Part writes it to an internal buffer.

4.3.5 I^{2}C Bus and PHOS4

To make the Preprocessor Demonstrator MCM work, the clock delay chip, PHOS4, has to be initialised. This is done via the I^{2}C bus (see [Phi] for detailed information on it). To save space on the FPGA, only the rudiments of an I^{2}C interface were implemented in the Preprocessor Controller. The protocol for controlling the bus as a bus master was written in software, in the class `I2CMaster`. It supports the I^{2}C clock synchonising mechanism and bus arbitration. When there are multiple masters on the bus and several of them try to start a transmission at the same time, bus arbitration decides which of them gets its way. The clock synchronising mechanism allows a slow receiver to cope with a transmission from a faster device. The state of the I^{2}C bus lines is the logical *and* of all output driver states, since those are open-collector outputs. A slow device can extend the clock cycle by pulling the clock line down after the master has released it. Both features can be disabled in the `I2CMaster` Part because they involve read operations which may be time-consuming when doing a remote access over the network.

A subclass, `I2CMasterPPM`, interfaces the high-level functions of `I2CMaster` with the functionality provided by the Preprocessor Controller. It has a `PPMController` Part as a dependency. With its help it writes Controller programs setting the state of the I^{2}C lines and obtains the I^{2}C status word (Figure 3.7). It disables both clock synchronisation and bus arbitration in its superclass. There is no other master on the bus, and the PHOS4 is fast enough to cope with a transfer as fast as allowed by the I^{2}C specification. That is still much slower than the Preprocessor Controller. Therefore, a *Wait* command is inserted after each command changing the state of the I^{2}C lines.

Naturally, there is also a Part class for the PHOS4. It uses an `I2CMaster` Part for tranmitting configuration bytes to the PHOS4. It has as an Attribute dependency the I^{2}C bus address of the chip. Four bits of that address are (constant) inputs to the PHOS4 so several of them can be on the same bus. The `Phos4` Part has a GUI with which the user can set the delay or activate special output modes for each channel. The Part can also parse a state string that contains the delays of all channels.

4.3.6 Pure I/O Frames

4.3.6.1 The `DataSource`

The Part type `DataSource` is an I/O frame that does not itself access hardware. As its name suggests, its task is generating data and transmitting it via the I/O frame mechanism. The user can select the number of words and its operation mode in its GUI. It can be used to trigger other I/O frames or to configure hardware with pre-defined values.

The data source has four different operation modes. It can transmit constant or random data. It can also generate ramps with a defined initial value and slope. The slope is input as a fraction. For instance, a value of one half will result in values that are incremented by one after each second word.

The fourth mode generates calorimeter pulses with noise and a pedestal added. This is the sort of data the RemAsic would receive in the ATLAS experiment. The heights of the generated pulses have a realistic probability distribution and are generated with the technique described in appendix C. The data model is rather simpler than the one used for calculating compression modes (chapter 6). It does not take into account the superposition of pulses from different bunch-crossings (pile-up).

This operation mode is called the 'front-end data mode' and was used to generate realistic data for the Front-End Data Source FPGA. This FPGA reads data from the dual-ported RAM and feeds them into the RemAsic (see 3.3). The `DataSource` can create multiple channels of calorimeter data. A word from each channel is transmitted in turn. This is the way the FPGA design reads the data from memory.

The `DataSource` can be operated with a state string. It contains the number of values to be transmitted, a character giving the operation mode and some more parameters depending on the operation mode. Parsing a state string triggers a transmission of data in the desired mode. This makes it possible to use the data source in a script. An example can be found in Section 5.1.3, especially Figure 5.4.

4.3.6.2 I/O Frames for Automatic Tests

An automatic test was set up to check the correct coding of data by the RemAsic. For that purpose, two I/O frame Part classes were written. The first, `Compressor10`, emulates the compression of ten-bit words done by the RemAsic. It receives data in the same order as the Front-End Data Source FPGA and the `DataSource`: one word from each of the four channels in turn. It can operate in three of the compression modes the RemAsic implements, namely no operation, Huffman-inspired coding and difference coding. The latter are the most promising modes (see 6.3). Like the RemAsic, the `Compressor10` class has a bit mask to disable some channels. Member variables contain the number of channels and the number of bunch-crossings read out. All parameters can be set with a state string. A GUI does not yet exist.

The other I/O frame needed for automatic tests is the `Comparator`. It has two input ports. Whenever data from the ports differ, it writes both values and the data count to a log file. It can buffer data from either or both ports until transmissions to the other port catch up. Buffering only one port makes sense when the other one may receive irrelevant surplus data. The length of the data block to be compared is determined by the buffered port. The `Comparator` can stop the execution of a script when it finds a mismatch. This is done with *exceptions*, a C++ mechanism that allows aborting nested function calls until the exception is 'caught' by the ultimate caller. The `Comparator` is controlled with state strings; a GUI does not exist.

Chapter 5

Test Setups and Results

As a preliminary test, the Preprocessor Module with the RemAsic was operated stand-alone at a low clock frequency. This was followed by tests within a larger system at the full clock frequency. Difficulties due to defective Front-End ASICs were overcome by using a new FPGA design as a data source (described in Section 4.3.6.1). To check reliability over a period of hours, an automated readout test was set up.

For the largest part, the RemAsics worked well as intended. The exceptions were some minor design flaws in all RemAsics and manufacturing defects in one of the chips. The Front-End ASIC (FeAsic) interface is impractical in some ways, which has already been taken into account for the FeAsic's successor, the Preprocessor ASIC.

5.1 Executed Tests

5.1.1 Standalone Tests of the Preprocessor Module

5.1.1.1 Setup and Test Procedure

For some initial tests, only the Preprocessor Module was used. The RemAsic was accessed via the serial Pipeline Bus controlled by the Preprocessor Controller. That limited the RemAsic's clock frequency to 0.57 MHz. The Front-End ASICs on the Multi-Chip Module had to be clocked with the same frequency for the serial interface to the RemAsic to work.

The hardware setup for the standalone tests is shown in a symbolic representation in Figure 5.1. The VME crate contains the test system Preprocessor Module and a computer which controls the VME bus. On that VME computer, the bus server program is running. It is part of the HDMC package and translates network packets sent by HDMC into VME bus accesses. HDMC runs on a workstation or a PC which is linked to the VME computer by a network (TCP/IP) connection.

All standalone tests follow the same pattern. A program for the Preprocessor Controller is written to the dual-ported RAM on the Preprocessor Module. Then, it is started by writing an *Execute* command to the Controller's command register. Finally, the readout memory region that by now contains the RemAsic's Pipeline Bus reply is read. The Controller program is given as a source code which can be processed by the `Assembler` Part. The `Assembler` transmits the binary values to the program memory Part, which writes it to the right region of the dual-ported RAM.

To make this procedure more convenient, a short script was written. It executes the three steps described in the last paragraph, after clearing the readout memory for clarity. It looks as follows:

```
'NetClient/LANtoVME/PPM/readout memory' GUI Clear
'PPMAss' "../prg/RemAsic.prg"
```

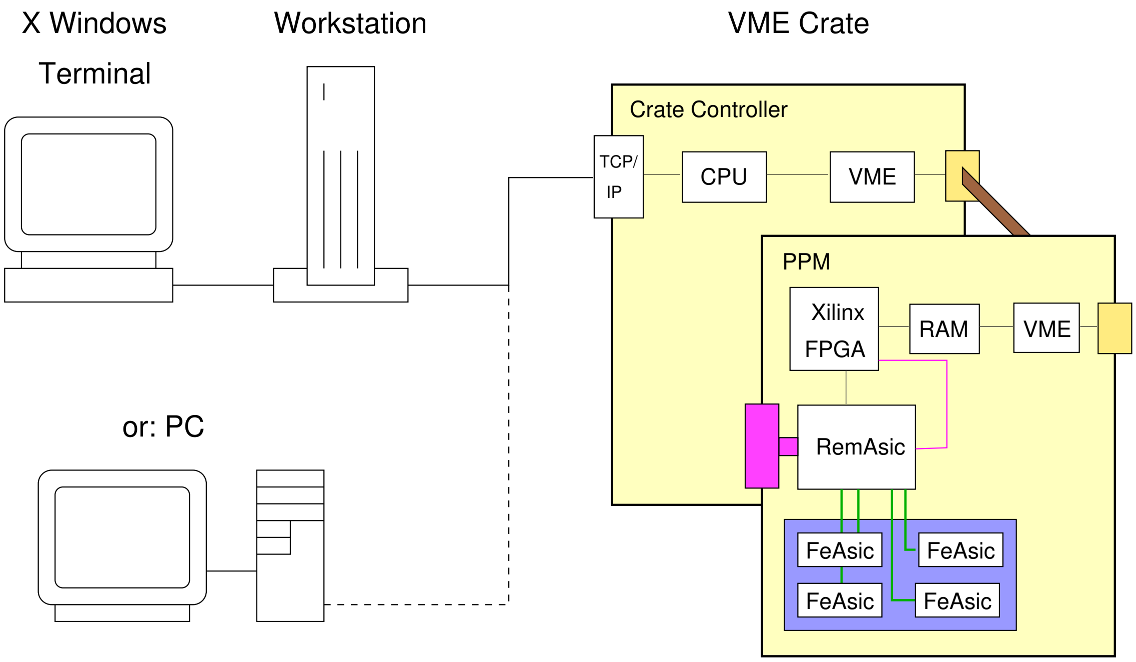

Figure 5.1: Test setup for standalone tests of the Preprocessor Module

```
'NetClient/LANtoVME/PPM/command' "0x84001100"
'NetClient/LANtoVME/PPM/readout memory' GUI Read
```

The readout memory and command register are on the Preprocessor Module called `PPM`, which depends on the remote VME bus `LANtoVME`, which in turn depends on the Part called `NetClient` for network access. The value written to the command register starts program execution at the beginning of the program memory. The assembler `PPMAss` has no dependencies, but its output port is connected to the input port of the program memory. The source file "RemAsic.prg" contains the Preprocessor Controller program to be executed. Different file names can be substitued for different tests. The source files first load the Controller's control register, reset the RemAsic and switch on the Controller's Pipeline Bus readout. Then some Pipeline Bus commands for transmission to the RemAsic follow.

5.1.1.2 Results

All the tests using only the Preprocessor Module were functional, 'in principle' tests. The RemAsic clock was 0.57 MHz, much slower than its usual operating frequency. All the same, these tests were very important for putting the RemAsic into operation and verifying its functionality. Besides, in the course of these tests the RemAsic Part classes for HDMC were developed (see 4.3.3, page 36).

At first, the module was operated without the lower CMC card carrying the Preprocessor Multi-Chip Module (MCM). RemAsic subdevices were configured and read via the serial Pipeline Bus. This allowed verifying the function of the serial Pipeline Bus interface and of the internal subdevice interface. In one of the three RemAsics tested in this way, an error was encountered: Bit 15 of the level-1 queue registers was sometimes set regardless of what had been written to it and even after a reset. This is probably due to a manufacturing defect in that ASIC. The other two RemAsics and the other subdevices worked fine.

As a next step, the compression unit was put into operation. The data to be compressed were

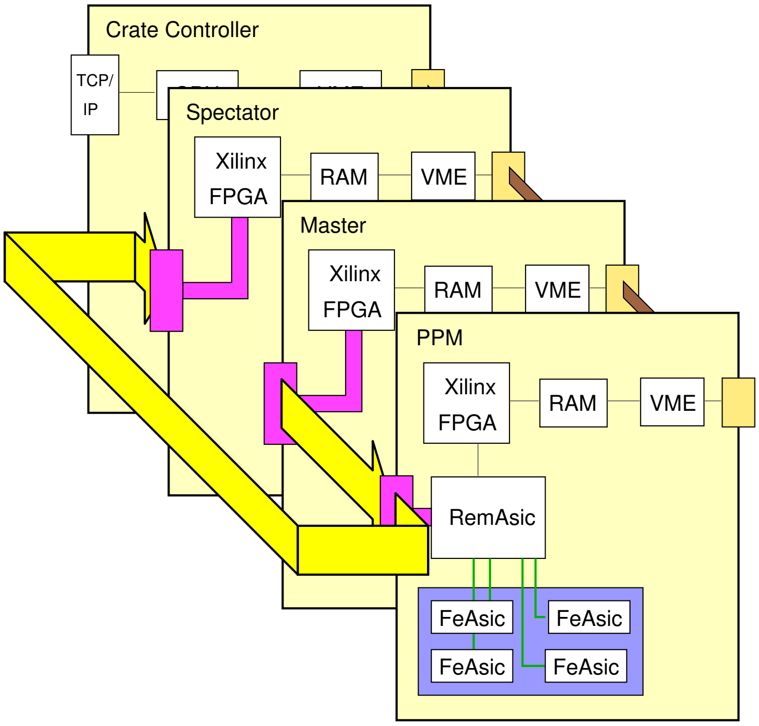

Figure 5.2: Test setup for tests with the parallel Pipeline Bus

written to one of the Event Buffers. This makes the compression unit process the data and write the result to one of the Pre-Main Buffers. They can be read by a subdevice read or by an ordinary readout operation that will be the rule during the running of the ATLAS experiment. Both possibilities were tried and found to work. All compression modes implemented in the RemAsic were tested. Since the results were checked by hand, only a small number of data sets could be processed. No errors occurred in these compressed data, even though later tests revealed small errors in the implementation of two compression modes (see 5.2.1 and appendix A).

Finally, the lower CMC card was added to the Preprocessor Module. The RemAsic was operated together with the Front-End ASICs (FeAsics), using the serial interface between them. The FeAsics were configured via that interface, and the result was checked with the FeAsic Spy Bus. The Spy Bus words contain, among others, the values of internal registers that can be set by a configuration process.

The most complex test done in this configuration was a complete readout operation. To make the readout data predictable, the FeAsics' Playback Memories were loaded with pre-defined data. The multiplexers inside the FeAsics were set to make the Playback Memory data directly available for readout. Then a readout operation was initiated by asserting the Level-1 Accept signal with the Preprocessor Controller. The readout data were obtained from the RemAsic via the Pipeline Bus. At this low clock frequency, the readout worked correctly without problems.

5.1.2 Tests using the Parallel Pipeline Bus

5.1.2.1 Setup and Test Procedure

The same tests done previously with the serial Pipeline Bus were now carried out with the parallel bus. The test setup is shown in Figure 5.2. The wide arrows on the left represent the parallel

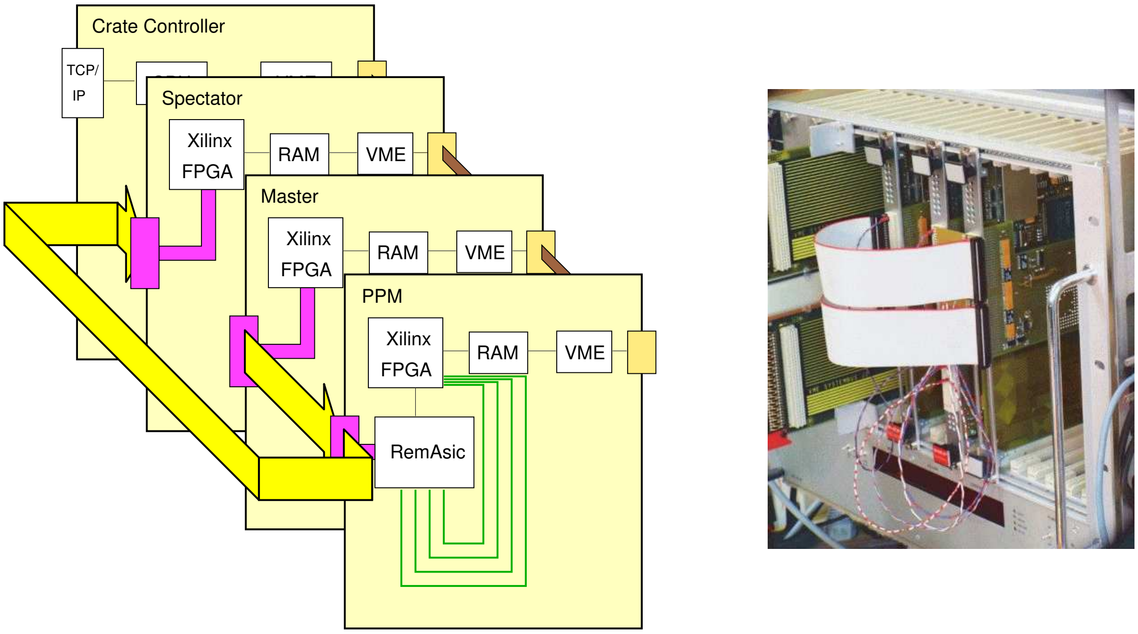

Figure 5.3: Test setup using the Front-End Data Source design

Pipeline Bus. The Preprocessor Module was placed to the right of the other boards so the RemAsic pins were accessible with an oscilloscope. The Pipeline Bus connection to the spectator module was made with a flat ribbon cable. The VME modules were accessed from a remote computer as described in the last section (see 5.1.1.1 and Figure 5.1).

Contrary to the situation in the standalone tests, the RemAsic is now clocked with 40 MHz. The Pipeline Bus clock, which is also the clock of the RemAsic chip, is generated by the Pipeline Bus master FPGA. It is fanned out by the lower CMC card on the master module and transmitted to all modules via wires. The Level-1 Accept signals that initiate readout are also generated by the master and fanned out like the clock.

Only details of the test procedure differ from the standalone tests: The program and readout memories were located on different modules. The program containing the Pipeline Bus commands for the RemAsic were executed by the Pipeline Bus master instead of the Preprocessor Controller. The Controller program source files from the standalone tests could be reused for these tests since the Pipeline Bus master has the same command format. It ignores commands which only the Controller supports. There is a script for the parallel tests which is very similar to the one shown in 5.1.1.1. It performs a few additional tasks before assembling and starting the Pipeline Bus master program. First, it makes the Preprocessor Controller execute a program resetting the RemAsic and adjusting some settings. The parallel Pipeline Bus and the front-panel clock and Level-1 Accept inputs have to be enabled. Besides, the script resets the Pipeline Bus spectator's address counter so it starts writing the Pipeline Bus data to the beginning of the memory.

5.1.2.2 Results

Reading and writing subdevices worked correctly with the parallel Pipeline Bus. The compression of data loaded into the Event Buffers also worked properly. As in the standalone tests of the Preprocessor Module, the checks were done by hand, so only small amounts of data could be processed.

Finally, the Front-End ASICs were used as a source of readout data. This led to two problems. For one thing, the serial interface between RemAsic and FeAsics did not work without transmission errors at 40 MHz. This affected both configuring the FeAsics and reading data from them. (Configuration was necessary to put the FeAsics into the right readout mode for the tests.) That the FeAsic configuration failed could be seen on the FeAsic's Spy Bus which provides internal registers set by the configuration process. The errors were due to the relative timing of the clock and data lines (see 5.2.2 for more details). To be able to continue tests with the FeAsics, the data signal was delayed by soldering a capacitance between the signal line and ground. This improved the timing.

When the interface was working, the Playback Memory of the FeAsics was configured with defined data so the result of a readout was predictable. However, the readout data from three of the four channels were incorrect. Those channels always provided the same data value, regardless of what was written to the Playback Memory. This was not due to transmission errors or processing errors in the RemAsic: The serial interfaces were monitored with an oscilloscope and showed the same wrong data. An incorrect configuration of the Playback Memory could not have been the reason either. The resulting data values were the same when the FeAsics were configured to use the ADC input instead. Save for the one intact FeAsic, the data were independent of the voltage applied to the analogue inputs of the ADCs. It was concluded that the cause was a manufacturing defect in the FeAsics concerned. That defect was slight enough for the ASICs to work at the low clock frequency applied in the standalone tests, but not at 40 MHz.

To continue testing the readout data path as far as the RemAsic was concerned, data was fed into the serial interfaces from the FPGA on the motherboard. For that purpose, a new FPGA design was created which is described in Section 3.3. The lower CMC card on the Preprocessor Module is replaced by a soldered adaptor that connects the FPGA to the RemAsic's interface pins. The modified setup is shown in Figure 5.3. Using the FPGA as a data source has the disadvantage that not all the functionality of the Preprocessor Controller is available any more. For instance, the RemAsic's Spy Bus cannot be read with the Data Source design.

Readout operations with the Front-End Data Source design did not produce any errors.

5.1.3 Automated Test

5.1.3.1 Motivation, Setup and Test Procedure

The disadvantage of the readout tests described in the last two chapters is that error checks are done manually. This implies that only small amounts of data can be processed and rare errors are missed. It is also error-prone since the 32-bit Pipeline Bus words containing code words of different lengths are hard to read. But the Preprocessor system in the ATLAS experiment has to run continually for months. A test involving large amounts of data was necessary to prove reliability of the RemAsic.

To check the integrity of readout data over a period of hours, an automated test was set up. The compression performed by the RemAsic was done in parallel by an I/O frame Part in the HDMC software (4.3.6.2). After reading out the compressed data, they were compared with the result of the software compression by another I/O frame. The hardware setup was the same as for the last parallel tests, shown in Figure 5.3. The one difference is that a modified version of the Pipeline Bus spectator FPGA design was used. It only writes user data blocks to the memory and discards the first data word of each block. In readout data blocks, the first word is a status word not containing any readout data.

The test is controlled by the HDMC software. The sript in Figure 5.4 executes the five steps necessary for one run of the test. Their effect is shown in Figures 5.5 and 5.6. Besides Parts corresponding to the hardware components on the VME modules, the HDMC Part hierarchy contains I/O frame Parts for handling the data flow. The connections between those I/O frames are represented in Figure 5.5.

```
    Repeat 1000

1   // Reset Pipeline Bus spectator:
    'NetClient/LANtoVME/Spectator/Xildat0' "0"

    // Generate new data and write it to front-end data memory:
2   'Raw Data Source' "0x1000 F 8 4 10"
    Wait 10

3   // Reset Front-End Data Source read address counter:
    'NetClient/LANtoVME/PPM/Xildat0' "0"

    // Execute Pipeline Bus master program:
4   'NetClient/LANtoVME/Master/command' "0x84000000"
    Wait 300

5   // Trigger transmission readout memory -> comparator:
    'Trigger Data Source' "1 C 0"
```

Figure 5.4: HDMC script for the automatic test. The `Wait` commands make sure that the current operation is finished before the next one starts. The `Repeat` command repeats script execution for the given number of times. The `DataSource` state string in step two and five contains the number of data words to be transmitted, a character specifying the operation mode and some additional parameters. The mode character 'C' stands for constant mode, 'F' for 'front-end' data.

Step one of the script resets the Pipeline Bus spectator's address counter. This makes it write Pipeline Bus data to the beginning of the memory. The second step generates the raw data to be compressed and read out. They are generated by a `DataSource` Part and passed on along two data paths (see Figure 5.5): One leads to the `Memory` Part which writes them to the dual-ported RAM on the Preprocessor Module. On the other path, the data are compressed by an I/O frame and fed into one port of the comparator. Step three of the script resets the address counter of the Front-End Data Source on the Preprocessor Module. This is necessary to make it restart reading data at the beginning of the RAM.

Step four is central to the test: An *Execute* command is written to the Pipeline Bus master's command register. This starts a program in the dual-ported RAM which triggers 128 readout operations. For each operation, the master asserts the Level-1 Accept signal once and transmits a *BeginOfData* command on the Pipeline Bus. This command requests readout data from the RemAsic. Since the master program is the same for each run of the test, it does not have to be rewritten every time. Instead, it is written by an initialisation script which has to be executed before the test. This script also loads the FPGAs and sets compression modes in the RemAsic and the software.

The last, fifth step of the script uses a `DataSource` Part to trigger an I/O frame transmission from the readout memory (Figure 5.5). When receiving a value on its trigger port, the `Memory` Part reads the dual-ported RAM on the spectator module. The data written there by the Pipeline Bus spectator are transmitted to the second port of the comparator. When the `Comparator` Part detects mismatches between the data from software and hardware compression, it writes them to a log file. It is configured to ignore surplus data from the second port. The whole memory content is transmitted to it regardless how much of it is readout data. The number of data words to be compared is determined by the result of the software compression arriving on the first port.

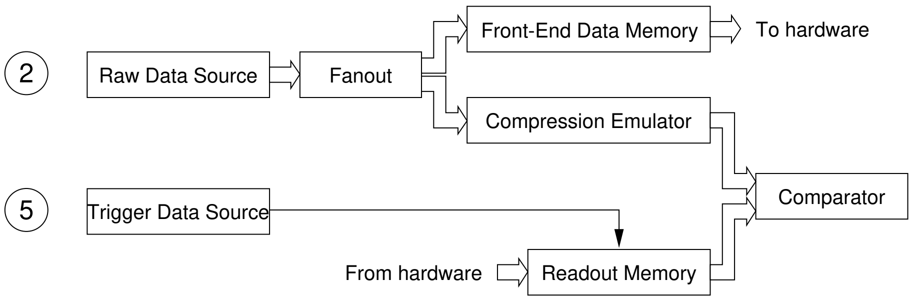

Figure 5.5: HDMC I/O frames and data paths in the automatic test. The 'Fanout' I/O frame is necessary only because the Data Source cannot be connected to more than one input port.

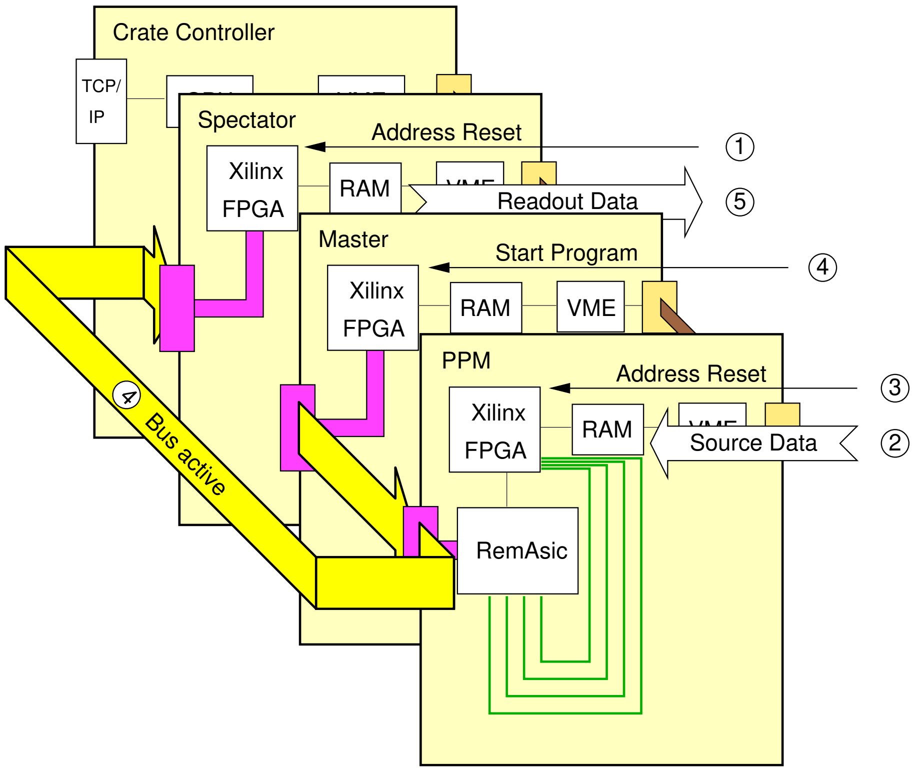

Figure 5.6: Test procedure for automatic tests (see text and Figures 5.4 and 5.5)

5.1.3.2 Results

In the longest automatic tests, the script was repeated 50 000 times. This means 204.8 million data words were compressed and read out. The test lasted about ten hours. It was performed for both the Huffman-inspired and the difference compression mode, with two different RemAsics. The `DataSource` I/O frame generating the raw data was in 'front-end data' mode, producing calorimeter trigger tower data. No errors were found. It can be concluded that the RemAsic works reliably. The word error rate is below $2.59 \cdot 10^{-8}$ with a confidence level of 99% (see appendix B).

Shorter tests were carried out with slightly different parameters. Random data as well as realistic data were generated for compression. All three available RemAsics were used. On a hot afternoon, one of them produced wrong data: In about one in every thousand Pipeline Bus words, bit five was not set when it should have been. The following morning, when the air was cooler, it worked correctly. It was the same ASIC which had already shown a manufacturing defect in the subdevice readout (see 5.1.1.2 above). The observed malfunction was probably caused by a defect in the Pipeline Bus interface which makes it extremely sensitive to heat.

5.2 Further Results

5.2.1 Discovered Design Errors

On some occasions during the RemAsic tests, all three RemAsics misbehaved in the same way. This was due to flaws in the RemAsic's source code. Some small points have been implemented incorrectly in the hardware description language code.

The first error was discovered during problems with the serial Front-End ASIC interface. The Spy Bus does not contain the state register of the FeAsic interface as specified in the RemAsic manual [RemA]. Instead, the three bits concerned are constant. The second design error concerns the subdevice configuration process. Because the reinitialisation of an internal register is missing, it is impossible to configure both Pre-Main Buffers in sequence.

The other two design errors found are in the compression unit. As was found during tests of the Front-End Data Source, the eight-bit no-operation mode does not work properly. It should discard the upper two bits of each ten-bit energy value. Instead, these bits are logical *or* combined with the lower bits of the next datum. The last design flaw was found only during the automated tests, since it occurs rather rarely. It concerns the Huffman-inspired and difference modes. Both output 13-bit data which may contain three short codes. For some combinations of data, data words are not completed at the end of a data set. When an output word is partly filled by the last values of one data set, it is continued at the beginning of the next.

These design errors are described in more detail in appendix A. Their deeper reasons found in the RemAsic's source code are also treated there.

5.2.2 The Front-End ASIC Interface

The serial interface between the RemAsic and the Front-End ASICs (FeAsics, see 2.2.2) was a frequent source of problems. By design, the bits on the serial data line are latched at the same clock edge which operates the shift register. It must have been assumed by the designers that because of signal delays the data would be latched just before the data line changes. That is not the case. Instead, the bits are read just at the time it changes, which often results in undefined data. This problem was exacerbated by the fact that the output drivers for the serial interface could drive a current of only 4 mA. That made the slopes of the signals slow and extended the time intervals during which their level was undefined.

These results have already been taken into account for the successor of the Front-End ASIC, the Preprocessor ASIC. Its serial configuration and readout interface is a totally different, synchronous interface.

A second point about the serial interface concerns the RemAsic. To request readout data, the RemAsic activates the acknowledge signal. It then waits for the ready signal from the FeAsic to be toggled before activating the serial clock. When no FeAsics are connected to any port, the RemAsic gets stuck and has to be reset. This is the case even when no port is enabled in the RemAsic's port mask.

Even when FeAsics are connected, there may be problems. During setup and testing it may happen that the Level-1 Accept is not delivered to the FeAsics. In that case, they keep the ready line down because they have no readout data. The RemAsic's hang-up is no sensible way to react to this and makes diagnosis much harder. A timeout on the interface would have been desirable.

The same problem occurs during configuration transmissions from the RemAsic to the FeAsics. Configuring the FeAsics through the serial interface is handled in the RemAsic as a subdevice configuration process (2.1.2). The configuration is not complete until they have responded on the ready line. While a subdevice configuration is in progress, the RemAsic does not accept most Pipeline Bus commands. Therefore it can only be reset if the FeAsics do not answer correctly.

Chapter 6

Evaluation of Compression Modes

The RemFPGA, the RemAsic's successor, has to compress data provided by the Preprocessor ASICs (see 1.2.3). For every Level-1 Accept, a number of energy values corresponding to adjacent bunch-crossings are read out.

The RemAsic implements a number of compression modes suitable for trigger-tower data. A piece of software was written to calculate compression rates for these and some new compression modes. The new compression algorithms retain the advantages of the best RemAsic algorithms but achieve better compression.

6.1 Introduction

The RemAsic is the prototype of a readout chip in the Level-1 Calorimeter Trigger Preprocessor. As has been described in Section 1.2.3, readout data are compressed for several reasons. Though the data would be compressed in any case before being stored, it makes sense to precompress them early in the data stream. For one thing, this increases the total compression rate achieved. For another, it makes transmitting the data easier. The RemAsic outputs data on the Pipeline Bus. Data from eight RemAsics on that bus are collected by a readout driver module and transferred to the data acquisition (DAQ) via a serial link. Reducing the amount of data to be transmitted lowers the bandwidth requirements for the Pipeline Bus and cuts the number of serial links required.

The Pipeline Bus is planned to operate at a clock frequency of 40 MHz. At this frequency it can transfer 153 MBytes per second. This is much less than the amount of data output by the RemAsics. Eight RemAsics process data from 64 channels on their Preprocessor Module. Typically, for each Level-1 Accept five raw energy values and one BCID value[1] are to be read out. The envisaged maximum Level-1 Accept rate is 75 kHz, with the option to raise it to 100 kHz. This amounts to a data rate of 275 or 366 MBytes, respectively. A compression rate of 1.8 is necessary for the smaller figure, 2.4 for the higher one.

The compression rate aimed at is not a strict requirement. The bandwidth of the Pipeline Bus could be increased by operating it at a higher clock frequency. The number of serial links transmitting data from the readout driver modules could also be increased. Since the final Preprocessor system is still being designed, it could be adapted without problems. Alternatively, the number of values to be read out per Level-1 Accept could be reduced. It can be adjusted in the Preprocessor ASIC. Reading out one or two values less would be sufficient for monitoring the trigger. Furthermore, compression performance could be improved by compressing the BCID values separately

[1]That value is a result of the bunch-crossing identification done by the Preprocessor ASIC. It is 0 if no peak is identified or the height of the peak otherwise. These are the data transmitted to the trigger processors on the trigger data path, see 1.2.3.

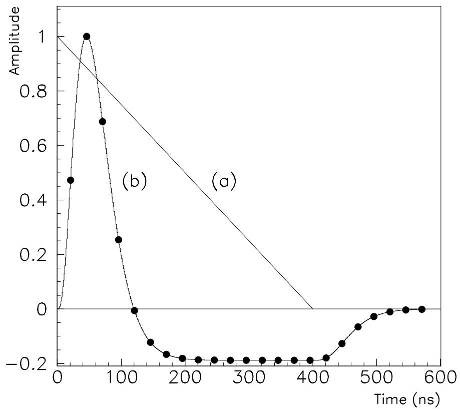

Figure 6.1: Liquid argon calorimeter drift current (a) and shaped pulse (b) [At]. The dots are the points in time at which the shaped pulse is sampled by the ADC at 40 MHz, once per bunch-crossing. Signal delays are adjusted so that one sample is always taken at the peak.

from the raw data with a different algorithm. This is beyond the scope of this thesis, which treats only the compression of raw trigger-tower data.

6.2 Calculating Compression Rates

6.2.1 Overview

A software program called COMCALC was written for evaluating the performance of different compression algorithms. It generates realistic calorimeter data and calculates the length of the code words which result from the compression. The number of bits for each data set is rounded up to a multiple of 32. That is done because the code words will be mapped onto Pipeline Bus words and any unused bits at the end of a data set will be wasted. It is the number of Pipeline Bus words that is relevant for the compression rate because the bandwidth of the bus is the limiting element. The program outputs the number of Pipeline Bus words and the compression rate for all compression algorithms. The compression rate is defined in terms of Pipeline Bus words, ie it is the factor by which the number of words is less than without compression.

Special care was taken to make the program efficient. No floating-point operations are used for data generation. Small inaccuracies are acceptable since the size of code words does not depend critically on the data for any of the algorithms. Further gains in efficiency were achieved by not computing the actual code words or Pipeline Bus words, just the amount of data. Forming of the codes and mapping them onto 32-bit words involves logical and shifting operations which can be very time-consuming.

COMCALC is started from a UNIX shell and provides text output. Its behaviour can be modified by giving some parameters in the command line. The number of data sets to process and the number of channels and bunch-crossings per data set can be adjusted. The energy pedestal and the intensity of noise can be given. These parameters influence data generation. The compression parameter used by some compression modes can also be passed in the command line.

As a special feature, a histogram of all generated data is recorded. The program prints their entropy (see 6.3.1 below) and the theoretical maximum compression rate derived from it.

6.2.2 Generation of Test Data

The part of the program generating data produces calorimeter pulses and adds noise and a pedestal value to them. The shape of a calorimeter pulse is shown in Figure 6.1. The calorimeter cell provides a triangular drift current which is then shaped to reduce the noise. Since the width of the pulse is always the same, pulses of different energies can be generated by scaling them.

The probability distribution of trigger tower energies is the same as that of particle transverse energies. It was simulated before the RemAsic was designed (see [Nie]) and is proportional to the energy to the power of -1.8. Since this function has a singularity at 0, it must be cut off for low energies. Pulses with heights below 1 GeV or four counts are not generated by COMCALC. Smaller signals are treated as noise. The fraction of particles with transverse energies under the threshold can be estimated from simulations of minimum-bias (ie low-energy) particals published in [Calo]. It is one tenth [Nie]. This is the fraction of bunch-crossings for which a pulse must be generated.

Since a pulse extends over more than 20 bunch-crossings, a given energy value may be affected by pulses that occurred far earlier. When generating trigger-tower data, these pulses have to be taken into account. The probability distribution discussed in the last paragraph applies to each bunch-crossing separately. All pulses contributing to one bunch-crossing have to be superimposed.

The data model used by the designers of the RemAsic for compression rate calculations is rather simpler. It generates at most one pulse for each data channel, no matter how many bunch-crossings are to be read out. For this reason the compression rates from [Nie] are much higher than the ones determined with the more accurate data model.

To the calorimeter pulses, Gaussian-distributed noise and a pedestal value is added. The noise comprises electronic noise and pile-up noise[2] and typically has a mean deviation of slightly less than 500 MeV or two counts [Cha]. To investigate compression performance in situations when the noise is more intense, its mean deviation is a parameter of COMCALC. The pedestal value models the ADC's reference voltage and is zero when it is adjusted properly. Since the Preprocessor readout is of special interest in debugging situations, the reference voltage cannot be assumed to be correct in this context. Therefore the pedestal is a free parameter of COMCALC.

Numbers with a certain probability distribution were generated by using homogenously distributed pseudo-random numbers as an index to a look-up table. The technique is described in detail in appendix C. It was used for the heights of pulses as well as for Gaussian noise.

6.2.3 Usage of the Compression Rate Software

When calculating compression rates with COMCALC, certain default parameters were used as a rule. Unless otherwise stated below, the program's parameters were as follows.

Typically, six values are to be read out per channel for each Level-1 Accept. Five of them are raw calorimeter data and one is the result of the bunch-crossing identification (BCID) algorithm of the Preprocessor ASIC. COMCALC was made to use six raw data instead (since it cannot emulate the BCID algorithm). As the BCID value is often zero and contains no noise, compression performance can only be underestimated that way. The error is on the side of caution.

A data set read from each Preprocessor Module consists of data from the 64 channels on the module. This number was always passed to COMCALC. One thousand data sets were generated for most compression rate calculations, corresponding to 1000 Level-1 Accept signals. Occasionally, 10 000 or 50 000 were used, but the compression rates obtained did not differ significantly. A number of 1000 was found to be sufficient.

The typical mean deviation of the noise on calorimeter signals is given in [Cha] as 420 MeV,

[2]The superimposed calorimeter pulses of low-energy particles arriving at a high rate can be treated as noise. This is called the pile-up noise of minimum-bias events.

which amounts to two counts[3]. A higher noise intensity can be expected when the trigger system is being put into operation and was also simulated. The pedestal value was usually either zero or ten. It should be zero when the ADC's reference voltage is adjusted correctly. The other value was often used to simulate a small maladjustment which can be expected in debugging and setup situations.

6.3 Compression Algorithms and Their Performance

6.3.1 A Word about Information Theory

Information theory is concerned with the information content of data, their compression and transmission. This section cannot be an introduction to information theory, but it will give an overview of some of its concepts which were used for evaluating compression modes.

One central concept of information theory is entropy. A probability distribution p_i is given which states how likely a data word is to take each possible value i. Then entropy is defined as follows:

$$H = -\sum_i p_i \log p_i$$

This is the same definition as for thermodynamic entropy, just without the Boltzmann constant. The information-theoretic entropy is a measure of the information content of a stream of data words. It may be given in the unit 'bits' by choosing two as the base of the logarithm. (In that sense, 1 bit $= 1 \cdot \log 2$.) If the data words are wider than the entropy in bits, space is wasted and the data can be compressed. It cannot be compressed to fewer bits than the entropy without loss of information. (This is a rough formulation of Shannon's source coding theorem.)

The Huffman algorithm provides a way of generating a suitable code from the probability distribution p_i. It is asymptotically optimal, ie for large amounts of data the average code word length approaches entropy, its theoretical minimum. The main disadvantage of Huffman codes is that they are inflexible: If the data turn out to have a somewhat different probability distribution, it yields bad results.

Entropy values have to be taken with a pinch of salt. They depend on the definition of what constitutes a data word. Consider the following string of bytes: (1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4). If you choose a byte as the size of a data word, the entropy is two bits (since four different values are transmitted with equal frequency of occurrence). If one data word contains four successive bytes, the entropy is zero because only one value occurs. The probability distribution and the entropy value contain no information about correlations between data words. What was said above about entropy and Huffman codes holds only if data words are totally uncorrelated, which they rarely are. Compression algorithms which exploit such correlations or code sequences of data at a time can get round the maximal compression rate derived from the entropy of single data.

For a brief introduction to information theory, see [Ma].

6.3.2 Algorithms Involving Loss of Information

One of the algorithms implemented in the RemAsic involves loss of information. The so-called run-length algorithm discards all energies below a certain threshold. In their stead, an indicator value and the number of values below the threshold are transmitted. Values over the threshold are transmitted without change.

Though it is highly desirable to read out the data without any loss of information, the run-length algorithm should be retained for the RemFPGA. The readout in the trigger is intended primarily for monitoring and debugging the trigger. Therefore it will often be used in exceptional

[3] One count represent 250 MeV.

Threshold parameter → Noise mean deviation ↓	10	10+2	10+4	10+6	10+8
2	1.30	1.86	2.82	3.67	4.44
4	1.22	1.49	1.97	2.71	3.61
6	1.17	1.35	1.62	2.03	2.62

Table 6.1: Compression rates for run-length encoding for different noise intensities and compression parameters. The pedestal of the data was always 10.

situations. The noise may be higher than usual, increasing the entropy of the data. Achieving a reasonable compression rate with an information-preserving algorithm may then be impossible. By adjusting the threshold of the run-length algorithm, by contrast, the amount of data it outputs can be reduced to fit the bandwidth of the Pipeline Bus. Since the energy peaks resulting from physics events are higher than the noise, they will be preserved even when strong noise is discarded.

There is no single compression rate for the run-length method. It can be increased indefinitely by raising the energy threshold. This is limited only by the amount of information one is prepared to lose. Table 6.1 shows compression rates for different noise intensity and different thresholds. As a rule, a threshold of the noise's mean deviation plus the pedestal yields good enough compression for a Level-1 Accept rate of 75 kHz. To achieve the better compression required for an Accept rate of 100 kHz, the threshold should exceed the pedestal by one and a half times to twice the noise RMS.

6.3.3 Information-Preserving Algorithms of the RemAsic

Of the RemAsic's information-preserving compression modes, the Huffman mode was dropped for practical reasons. It requires for operation a code table which can be generated with the Huffman algorithm[4]. Since Huffman compression yields bad results when data deviate from the foreseen probability distribution, the code table would have to be changed occasionally. This makes this compression mode difficult to handle since data sets would have to be matched with the code tables with which they were compressed.

The two remaining compression modes, the Huffman-inspired[5] and the difference method, are closely related. Both output 13-bit words and achieve compression by transmitting up to three short (4-bit) codes in one word. These short codes contain values between -7 and 7 and are intended for the majority of the data which consist only of noise. Larger values are transferred unchanged, one per output word. The difference method writes differences between successive data words in short codes, the Huffman-inspired mode the difference of data words relative to the parameter.

Table 6.2 compares compression rates for the Huffman-inspired and the differnce mode. There are several things to be learned from it: The Huffman-inspired method becomes ever less effective when the pedestal increases. The difference method is more tolerant in that respect but its compression rate is less than that of a well-adjusted Huffman-inspired compression. That fact can be understood from the way it works: Since the small codes in both compression modes are suitable mainly for noise, the probability distribution of the data derived from noise matter. In the case of Huffman-inspired compression the energy values are stripped of their pedestal by subtraction of the well-adjusted compression parameter, which shifts but does not widen their distribution. For the difference mode, the difference of successive data is formed, which increases their mean

[4]The Huffman algorithm that gives this compression mode its name is really an algorithm for obtaining an optimal code from a probability distribution. The Huffman mode is so called because it is intended to be used with Huffman-generated code tables.

[5]This is *not* an automatically adapting Huffman algorithm. It is named after Huffman because it occurred to the RemAsic's designers while experimenting with Huffman compression.

Pedestal value	0	2	5	7	10
Entropy of data [bits]	2.6	3.4	4.1	4.4	4.6
Compression rate of ...					
Huffman-inspired mode	1.79	1.79	1.79	1.79	1.37
Difference mode	1.74	1.70	1.65	1.60	1.54
Huffman-inspired, parameter 7	1.94	1.91	1.84	1.79	1.37

Table 6.2: Huffman-inspired versus difference compression mode. The compression parameter used for the Huffman-inspired mode was either set to the pedestal or kept constant at 7. The decrease in the compression rates is natural since the data's entropy rises with the pedestal. For small pedestal values the negative tail of calorimeter pulses (Figure 6.1) and part of the noise is cut off.

deviation by a factor of $\sqrt{2}$. The share of data values which can be represented in short codes is therefore lower. For high pedestal values, this does not matter any more because calorimeter pulses and their undershoot predominate. This favours difference coding since it allows for short-coding of data series which deviate from the pedestal.

The second lesson from the rates in table 6.2 is that both algorithms fall short of the rate of 1.8 required for a Level-1 Accept rate of 75 kHz. However, using the Huffman-inspired mode with a fixed parameter solves the problem of the low compression rate and at the same time saves the parameter. Having a variable parameter is undesirable for the practical reasons that speak against the Huffman mode. Fixing the parameter of the Huffman-inspired mode at 7 allows a maximal interval of data (from 0 to 14) to be represented in short codes. This works well for a pedestal of up to 6. When the noise is intensified by a factor of two (to a mean deviation of 1 GeV / four counts), the compression rate is still 1.80 for a pedestal of 5.

The Huffman-inspired compression mode is therefore suitable for day-to-day operation when the system has been correctly configured and the ADC reference voltage is near its optimum. However, the difference method has the edge when the reference voltage is wide off the mark, resulting in a high pedestal value of the data. Its compression is not good enough for it to be retained in its present form, but as will be explained in the next section, it can be improved.

6.3.4 Improvements of the RemAsic's Algorithms

The Huffman-inspired and difference compression modes produce output words of a fixed width. This was done because fixed-length codes are less susceptible to transmission errors than variable-length codes. In variable-length codes, the length of the data words is indicated by some of their bits. If these are transmitted wrongly, word boundaries will be lost and the whole rest of the data set may be misunderstood.

As could be seen in Section 5.1.3.2, the rate of transmission errors during readout is very low. The serial links to data acquisition are also very reliable. Therefore, codes producing words with different lengths are an option.

The two RemAsic modes can be improved by doing without fixed word lengths. For large energy values, their code word contains an unchanged input value. Energy values are ten bits wide. This means that two bits of the 13-bit code word remain unused (one is needed for a flag bit; see upper part of Figure 6.2). If one gives up fixed word lengths, they can be saved.

If input data are small enough to be compressed, up to three short codes are contained in one code word. They are four bits wide and can therefore fill the code word completely. If the series of short codes ends in the middle of a code word, however, this has to be indicated with a 'skip' code. Since the series of short codes tend to be long, it makes sense not to repeat the flag bit in each code word. This implies that a short 'end' code for signalling its end is needed in all cases.

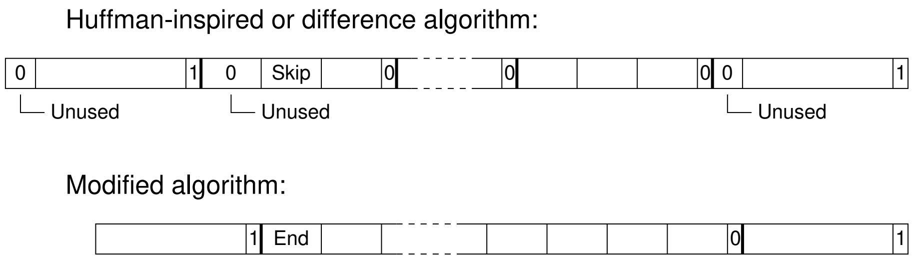

Figure 6.2: Modification of the Huffman-inspired/difference method output format. The data have to be read from right to left. In the modified version, the unused bit fields are no longer there. Besides, the '0' bits indicating short codes are not repeated.

Pedestal value	0		5		10	
Noise mean deviation	2	4	2	4	2	4
Huffman-inspired mode	1.94	1.93	1.84	1.80	1.65	1.49
Difference mode	1.74	1.65	1.65	1.42	1.54	1.29
Modified Huffman-insp.	2.30	2.29	2.24	2.23	2.11	2.06
Modified Difference	2.15	2.12	2.08	1.98	1.99	1.88

Table 6.3: Compression rates of modified compression modes. The compression parameter of the Huffman-inspired modes is fixed at 7. The improved algorithms perform well even for intensive noise and relatively high pedestals.

But this disadvantage is compensated by the fact that in some cases an additional unused short code is saved (see Figure 6.2).

As can be seen from table 6.3, these modifications increase the compression rates significantly. Even for more intense noise than is expected, they are well above the requirement of 1.8. But further improvements can be achieved. Small energy values occur more often than large ones. This holds also for those which cannot be represented in short codes. Therefore it makes sense to introduce absolute values of 'medium' width which are longer than short codes but shorter than ten bits. This is worthwhile even though an extra bit is needed to flag between the medium and the long words. A width of five bits was found to be optimal for medium codes.

It is worth reconsidering the width of the short codes, too. They are intended to contain mostly noise. Since the expected noise has a mean deviation of only two counts, it is sufficient to allow an interval of seven values for short codes. This reduces their width from four to three bits. (One

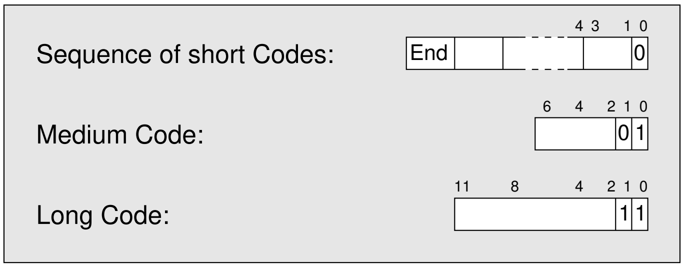

Figure 6.3: The output format of the new algorithms. The data words have to be read from right to left. The size of the words is indicated by the preceding flag bits. The end of a series of short codes is marked with an 'end' code.

Absoute algorithm				Difference algorithm			
Pedestal →	0	5	10	Pedestal →	0	5	10
Noise mean deviation ↓				Noise mean deviation ↓			
2	2.84	2.52	1.74	2	2.62	2.34	2.22
4	2.80	2.43	1.86	4	2.52	2.15	2.01
6	2.74	2.39	1.94	6	2.41	2.05	1.88

Table 6.4: Compression rates of new algorithms. The absolute method achieves better compression but the difference method is more tolerant to high pedestals.

code value is reserved as an 'end' code.) Figure 6.3 shows the resulting format. For the difference algorithm, the short codes contain the signed difference of successive data. In the case of the 'absolute' method, they are small unsigned data values. The sequence of short codes is initiated by the flag bit and ended by the 'end' code. Each medium and long code is preceded by flag bits.

Table 6.4 shows compression rates for the new compression algorithms. The absolute algorithm achieves better compression on the whole. For a low pedestal its rates are higher than 2.4. This is the requirement if the maximum Level-1 Accept rate is raised to 100 kHz. The absolute mode is also tolerant to high levels of noise. Curiously, for a high pedestal (10) the compression rate rises when the noise intensifies. This is because intensive noise contains large negative values. After the pedestal is added, they result in small values which are easy to compress.

The difference method is less effective but more tolerant towards high pedestals. Even for a pedestal of 35 the compression rate is above 1.8, sufficient for an Accept rate of 75 kHz. If the pedestal is 0, the compression rate exceeds 2.4 even for intensive noise.

6.3.5 Summary

The favourites of the RemAsic compression modes are the Huffman-inspired mode and the difference mode. They were considered for day-to-day operation of a correctly configured Preprocessor system. Only the Huffman-inspired mode achieves a compression mode sufficient for the expected Level-1 Accept rate of 75 kHz. New compression algorithms developed on the basis of the two RemAsic algorithms are suitable for the envisaged upgrade Accept rate of 100 kHz.

In debug situations and during setup of the system, data may be harder to compress. Noise may be intense and the ADC reference voltage maladjusted, resulting in a high pedestal value. The RemAsic's runlength algorithm is useful in that case. It involves loss of information, but the compression rate is the higher the more information is discarded.

The no-operation mode of the RemAsic compression unit has not been mentioned above. It should be retained as it is quite useful for debugging the RemAsic itself.

Conclusions and Outlook

On the whole, the RemAsic tests were very successful. The ASIC was found to work as intended at the desired clock rate of 40 MHz. Writing and reading of internal memories and registers was tested. Readout was found to work reliably for ten hours. The word error rate during readout is below $2.59 \cdot 10^{-8}$ with a confidence level of 99%. Four minor design errors discovered in the RemAsic can be easily fixed and would not have rendered it useless.

In one of the three RemAsics tested, manufacturing defects prevented proper functioning of some parts. This was although it had passed some less rigorous tests on a chip tester. This will not pose problems in the future since the RemAsic's successor will be an FPGA. Yield problems will therefore be no issue.

Only one of the RemAsic's information-preserving compression modes reaches the required compression rate. The new algorithms developed from it are much better. They are even suitable for the Level-1 Accept rate envisaged for upgrading, 100 kHz. One compression mode of the RemAsic which is not information-preserving is useful when data are unfavourable for the other algorithms. This may be the case when the Preprocessor system is not yet properly configured.

The successor of the RemAsic will be an FPGA, the RemFPGA. Large parts of the RemAsic design can be reused for it. This will involve porting the source code for a different synthesis tool. Some parts of the RemAsic will have to be changed: The serial interface receiving readout data will have to be rewritten completely. The compression unit will provide a different set of compression modes, including the new algorithms described in this thesis.

The RemFPGA will first be operated within the test system. There exists a daughtercard with a large enough FPGA and Pipeline Bus connectors. The RemFPGA will then be employed on the final Preprocessor Module to be presented at the beginning of the year 2001.

Much work remains to be done on the Hardware Diagnostic, Monitoring and Control software (HDMC). It is beginning to be used by collaboration members outside Heidelberg. Code for controlling the Preprocessor System and other components of the ATLAS Level-1 Calorimeter Trigger has to be written. HDMC's hardware drivers will have to be integrated into the ATLAS data acquisition software.

Appendix A

Design Errors of the RemAsic

Contrary to expectations, the tests revealed some design flaws in the RemAsic whose deeper causes could be traced in its hardware description language source code. They are listed here, in order of declining importance. None of them would have seriously impaired the usefulness of the RemAsic, had it been the final chip design for its purpose. The readout data could still have been decoded, so the RemAsic could have been used.

Huffman-Inspired and Difference Modes of Compression Unit

These two modes are very similar and in fact share a submodule in the RemAsic design. That is why this error occurs in both of them. Both codes achieve compression by writing to a 13-bit output word three short differences instead of one unchanged input word. The string of short codes is not interrupted by the end of a data set. This cannot have been intended by the RemAsic's designers since it complicates decoding and is not mentioned in the user manual. When an output word is partially filled by the last values of one data set and the first value of the next can also be short-coded, no new output word is started. The remaining one or two short codes for which there would have been room in the last output word are placed at its beginning without a flag to indicate that. The situation is illustrated in Figure A.1. The readout data could still be decoded, but one would have to remember the number of short codes, if any, that the last word of the previous data set contained.

The reason for this behaviour is that the counter for the remaining number of bits in an output word is not reset at the start of a data set. Because it has to be reset whenever an input value is too large for a short code, it has a synchronous reset signal[1] of its own. It is independent of the synchronous reset that initialises all other registers before starting on a new data set. Activating the bit counter's synchronous reset at the start of each data set was simply forgotten. Since the effect does not show for most combinations of data, it probably did not occur in simulations of the RemAsic design.

Eight-Bit No-Operation Mode of Compression Unit

This operation mode was intended to output the lower eight bits of each ten-bit data word. The resulting bytes are mapped to 32 bits as usual by placing each to the left of its predecessor. The output data bus of a compression module is always 32 bits wide. Successive output words are logical *or* combined after being shifted to the right position. The error in this operation mode

[1] A synchronous reset is not to be confused with the (asynchronous) reset of all flip-flops in a chip triggered by asserting the external reset pin. Any signal that initialises a register to a defined constant value may be called a synchronous reset.

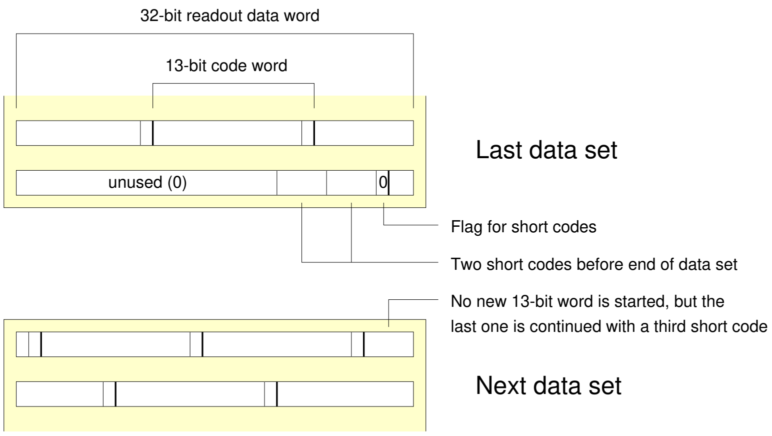

Figure A.1: Design flaw in the Huffman-inspired and Difference compression modes (see text)

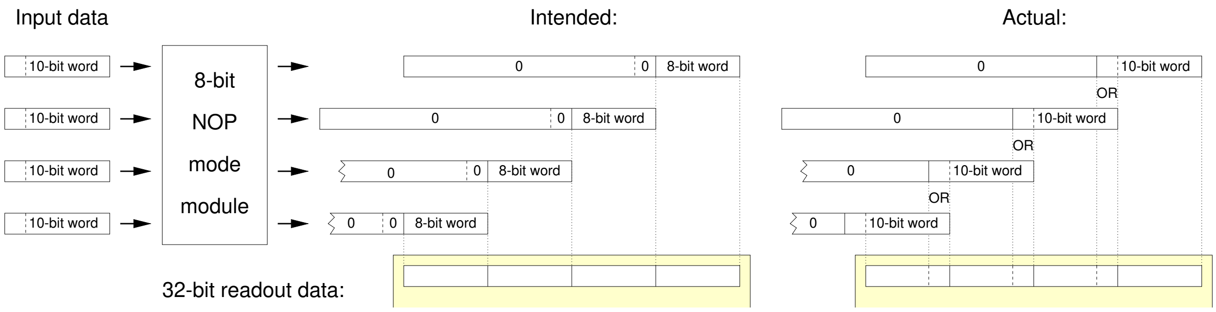

Figure A.2: Design flaw in the 8-bit no-operation compression mode (see text)

consists in that the two most significant bits of input data are not cleared. This has the effect that the lowest two bits of each word are logical *or* combined with the upper two bits of the previous datum, as shown in Figure A.2. This yields wrong output data whenever the most significant bits are not zero.

Configuration of Pre-Main Buffers

Immediately after configuring one Pre-Main Buffer in continuous mode, no other Pre-Main Buffer configuration can be carried out. For instance, configuring both Pre-Main Buffers in sequence in continuous mode is impossible. This is inconvenient since it is the obvious way to configure both buffers.

The Pre-Main Buffers are the only subdevices that contain 32-bit data. This makes it impossible to include an address in each configuration data word. The RemAsic provides two ways to configure Pre-Main Buffers: In random access modes, address and data words alternate. In continuous mode, one address word is followed by data written to successive addresses. The address word also selects which Pre-Main Buffer to configure.

After a configuration in continuous mode, the address words of following configurations are not interpreted. Instead, they are taken for data words belonging to the first configuration process. Only after a different subdevice has been configured or read a renewed Pre-Main Buffer configuration is possible.

In continuous mode configuration the RemAsic sets a flag indicating that all following words are data. The error occurs because this flag bit is not reset before a repeated Pre-Main Buffer configuration. Only subdevice read operations or configuration of a different subdevice clear it.

This error represents more of an inconvenience than a serious problem. One of the ways round it is to initiate the configuration of a different subdevice between the two Pre-main Buffer configurations. It is possible to do this without actually sending any data to the other subdevice.

Spy Bus

The last error concerns the Spy Bus. Since the Spy Bus is a feature only used for testing, it is of little importance. It just makes testing less easy. Due to three flawed assignments in the source code, three bits of one Spy Bus word are always set. They were intended to give the state of a state machine.

To avoid glitches[2], the eight states were coded as eight-bit values with only one bit set. To save space on the Spy Bus, the eight-bit state register was to be mapped to three bits. In the right-hand side of the assignments, only the state constants are present. Comparing the state register with them was forgotten. The bits are therefore assigned to non-zero values, setting them permanently.

[2]When a register containing several bits changes, not all the bits change at precisely the same time. Intermediately, the register may have a value that differs from both its initial and final state. This may cause spikes on signals resulting from comparisons of that register value. To avoid this, state machines are sometimes written in a way that the state register takes only values with just one bit set.

Appendix B

Calculating Upper Limits for Error Rates

In the long-time compression and readout tests with the RemAsic, no errors occurred. All the same, the error rate to be expected is clearly not 0. The error rate can be plausibly said to be below a certain limit with a certain level of confidence. That limit can be calculated from the desired confidence level using probability theory.

In textbooks (for instance [Kri]), the following problem is treated: A probability experiment yielding a result of either 1 or 0 is performed n times. The result 1 has the probability p, 0 the probability $1-p$. $k \in \{0, 1, \ldots, n\}$ is the sum of all results. Then the confidence interval $C(k)$ for p with a confidence level of $1-\alpha$ is

$$\begin{aligned} C(0) &= [0, p''(0)[\\ C(k) &=]p'(k), p''(k)[\qquad \text{for} \quad k = 1, \ldots, n-1 \\ C(n) &=]p'(n), 1] \end{aligned} \tag{B.1}$$

where p' and p'' are defined as follows:

$$p'(k) = max\{p : 1 - B(k-1; n, p) \leq \frac{\alpha}{2}\} \tag{B.2}$$

$$p''(k) = min\{p : B(k; n, p) \leq \frac{\alpha}{2}\} \tag{B.3}$$

$B(k; n, p)$ is the cumulative binomial distribution with parameter p, defined as follows:

$$B(k; n, p) = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i} \tag{B.4}$$

It gives the probability of the above experiment resulting in 1 not at all, once, twice or up to k times.

In our case, the probability experiment is the serial transmission of one data word to the RemAsic, its compression and transmission on the Pipeline Bus. A result of 1 means the word was not transmitted and compressed correctly, a result of 0 that it was. p is the rate of wrongly transmitted words. Since no errors were observed, k is 0, which simplifies things significantly. Formula B.4 defaults to

$$B(0; n, p) = (1-p)^n \tag{B.5}$$

Since the mapping $p \mapsto (1-p)^n$ is monotonous, the confidence interval becomes

$$C(0) = [0, p''[= [0, 1 - \left(\frac{\alpha}{2}\right)^{\frac{1}{n}} [\tag{B.6}$$

For large n, computing p'' with this formula is beyond any floating-point package. But it can be approximated very well in those cases:

$$p'' = 1 - \left(\frac{\alpha}{2}\right)^{\frac{1}{n}} = 1 - \exp\left(\frac{1}{n} \ln \frac{\alpha}{2}\right) \approx -\frac{1}{n} \ln \frac{\alpha}{2} \tag{B.7}$$

For a confidence level of 99%, this is

$$p'' \approx -\frac{1}{n} \ln 0.005 = \frac{1}{n} \cdot 5.3 \tag{B.8}$$

Appendix C

Generation of Pseudo-Random Numbers with an Arbitrary Probability Distribution

For calculating compression rates, realistic data had to be generated. That required the generation of random numbers with certain, non-homogenous distributions for energy values and Gaussian-distributed noise. For reasons of performance, it was desirable to be able to calculate these numbers with exclusively fixed-point operations.

That was done using a table in which the more probable results occur more frequently. A homogenously distributed pseudo-random number was used as an index to that table. This approach involves a discretisation of the desired probability distribution, in two respects. First, the generated numbers are integer (or fixed-point) numbers. This corresponds to the analogue-to-digital conversion also done by the electronics.[1] Second, the probabilities for each value are multiples of the inverse of the table length, as the index is homogenously distributed and the number of equal table entries is an integer. That has the effect that continuous distribution functions that extend to infinity, like the Gaussian distribution, are cut off at some point. To model the continuous distribution reasonably well, the table should be significantly larger than the number of different values to be generated. If the pseudo-random numbers are to be scaled by a factor, this has to be taken into account when estimating this number.

How to generate such a table? It can be seen as the look-up table of a function $f(x)$ for equidistant values of x. The larger the table has to be, the finer the steps for x and the more accurately $f(x)$ is scanned. f is assumed to be continuous. In that sense, f is the limit of a table of infinite size and precision, defined in an interval in which the infinite set of x values used for sampling f is dense. As it depends on the probability distribution to be modelled, it must be possible to calculate it analytically from the probability distribution function $p(y)$. Without loss of generality, f can be assumed to be monotonic and increasing; this results in look-up tables whose entries are also increasing. The number of equal entries in the table is 'roughly' proportional to their value's probability. This number is an approximation of the inverse of the derivative of f. Since f is the idealisation of a table, this means that

$$\frac{dx}{df(x)} \propto p(y) = \frac{dP(y)}{dy}. \tag{C.1}$$

[1] There is a small error of up to one point if two such values are added up, as an energy pulse and a noise value often are in our case. Since the compression rates do not depend noticeably on such small inaccuracies, that is acceptable.

That makes f proportional to the inverse of the cumulative distribution function $P(y) = \int p(y)\, dy$. f can be chosen as P^{-1}, which is defined in the interval $]0,1[$ if p is normalised.

Bibliography

[At] ATLAS Collaboration
Techical Proposal
CERN/LHCC/94-43, LHCC/P2

[Calo] ATLAS Collaboration
ATLAS Calorimeter Technical Design Report
CERN/LHCC 96-40

[Cha] R. L. Chase et al.
A Fast Monolithic Shaper for the ATLAS E. M. Calorimeter
ATLAS Internal Note LARG-NO-10

[FeA] Alexander Mass
ATLAS Level-1 Trigger: Front End ASIC; Users Manual
HD-ASIC-19-0896
http://wwwasic.kip.uni-heidelberg.de/atlas/docs/

[Kri] Klaus Krickeberg, Herbert Ziezold
Stochastische Methoden
Springer, 4.Auflage 1995
ISBN 3-540-57792-0

[Le] Debra A. Lelewer and Daniel S. Hirschberg
Data Compression
http://www.ics.uci.edu./ dan/pubs/DataCompression.html

[Ma] David MacKay
A Short Course in Information Theory
Cavendish Laboratory, Cambridge, Great Britain, February 1995
http://131.111.48.24/pub/mackay/info-theory/course.html

[Nie] Bernhard Niemann
Datenkompression für die Auslese des ATLAS Level-1 Triggers
Diplomarbeit, Universität Heidelberg, IHEP 98-02 or HD-ASIC 39-0298
http://wwwasic.kip.uni-heidelberg.de/atlas/publications.html

[Pa] Kenneth P. Parker
The Boundary-Scan Handbook
Kluwer Academic Publishers, 1994
ISBN 0-7923-9270-1

[Pfei] Ullrich Pfeiffer
A Compact Pre-Processor System for the ATLAS Level-1 Calorimeter Trigger
Dissertation, IHEP 99-11 or HD-ASIC 50-0899
http://wwwasic.kip.uni-heidelberg.de/atlas/publications.html

[Phi] Philips Semiconductors
The I^2C bus and how to use it (including specifications)
Search for "I2C_BUS_SPECIFICATION_1995" at:
http://www.semiconductors.com/cgi-bin/search/search.pl

[Qt] Troll Tech A.S.
Qt On-Line Reference Documentation
http://doc.trolltech.com/index.html

[RemA] A. Mass, B. Niemann, C. Schumacher
DRAFT RemAsic User and Reference Manual
http://wwwasic.kip.uni-heidelberg.de/atlas/docs/

[Schu] C. Schumacher
The Readout Bus of the ATLAS Level-1 Calorimeter Trigger Pre-Processor Talk at the fifth Workshop on Electronics for LHC Experiments, Snowmass, 20-24 September 1999.
http://wwwasic.ihep.uni-heidelberg.de/atlas/publications.html

[Schu2] C. Schumacher
Ein flexibles Test- und Entwicklungssystem fr das ATLAS Level-1 Kalorimeter-Trigger Pre-Processor System
http://wwwasic.ihep.uni-heidelberg.de/atlas/docs/index.html

[SLi] CERN High Speed Interconnect Group
S-Link Homepage
http://www.cern.ch/HSI/s-link/

[Ste] Bernd Stelzer
Diploma Thesis
to be available at http://wwwasic.kip.uni-heidelberg.de/atlas/publications.html

[Strou] Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley, 3rd Edition, 1997
ISBN 0-201-88954-4