UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

DESIGN OF READOUT DRIVERS FOR ATLAS PIXEL DETECTORS USING FIELD PROGRAMMABLE GATE ARRAYS

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

In partial fulfillment of the requirement for the

degree of

MASTER OF SCIENCE

(Electrical Engineering)

By

Sriram Sivasubramaniyan

Norman, Oklahoma

2001

UNIVERSITY OF OKLAHOMA
LIBRARIES

OU
THESIS
SIV
cop 2

DESIGN OF READOUT DRIVERS FOR ATLAS PIXEL DETECTORS USING FIELD PROGRAMMABLE GATE ARRAYS

A Thesis
Approved for the School of Electrical and Computer Engineering

By

[redacted]
Dr.Monte Tull, Chair

[redacted]
Dr.Patrick Skubic, Co-Chair

[redacted]
Dr.Zhisheng Shi, Member

©Copyright by Sriram Sivasubramaniyan,2001

All Rights Reserved

Acknowledgement

I offer sincere thanks to Dr. Monte Tull for accepting to chair my thesis committee and for his continuing encouragement, guidance and help during my graduate program.

I express my profound gratitude to my Guru, Dr. Patrick Skubic for his academic, moral and financial help during the course of this research. Above all I thank him for giving me an opportunity to work at Lawrence Berkeley National Laboratory, one of the finest Research facilities in the entire contemporary world. I would cherish all my life, the work I have done with Dr.Skubic.

I extend a special thanks to Dr.Zhisheng Shi for his advice and for accepting me to be on my Master's thesis committee. I thank Richard Jared, Damon Fasching, John Joseph, Mark Nagel, Valerie Risk and Lukas Tomasek for guiding me at LBNL. I also thank all my friends who made my stay at Norman really memorable.

I really appreciate the help from Rusty Boyd, which made my thesis possible.

Above all, I would like to thank my mom and dad, without their support this work would not have been possible. I also thank Almighty for having blessed me with such good parents. I dedicate this thesis to my parents. My achievements are as much theirs as mine.

Table of contents

List of Figures and Tables

Table

Abstract

Microstrip detectors are an integral part of high energy physics research. Special protocols are used to transmit the data from these detectors. To readout the data from such detectors specialized instrumentation have to be designed. To achieve this task, creative and innovative high speed algorithms were designed simulated and implemented in Field Programmable gate arrays, using CAD/CAE tools. The simulation results indicated that these algorithms would be able to perform all the required tasks quickly and efficiently. This thesis describes the design of data acquisition system called the Readout Drivers (ROD). It focuses on the ROD data path for ATLAS Pixel detectors. The data path will be an integrated part of Readout Drivers setup to decode the data from the silicon micro strip detectors and pixel detectors.

This research also includes the design of Readout Driver controller. This Module is used to control the operation of the ROD. This module is responsible for the operation of the Pixel decoders based on the trigger and command inputs from the module control chip for Pixel detectors respectively. Comprehensive tests were performed to verify the functionality of the ROD data path and ROD controller.

CHAPTER 1

Introduction

Semiconductor tracking detectors (SCT) and Pixel detectors are an integrated part of high energy physics research. These detectors when placed in the path of high energy (GeV) particles traveling in space, determine the spatial coordinates of the passing particles. These particles are capable of producing small electronic signals when they pass through such detectors. This thesis focuses on the instrumentation for reading out the data collected from such particle detectors (ROD). The objective is to design a data acquisition system for the Pixel detectors. It will present the modules designed for this specific purpose. It covers the OU-HEP[1] collaboration with Lawrence Berkeley National Laboratory (LBNL) and University of Wisconsin for the development of readout drivers using FPGAs for ATLAS Pixel detectors.

1.1 The ATLAS Detector

To receive information about particles generated in a Large Hadron Collider, which are short lived, physicists take advantage of a detector to look at the particle's decay products, which exist long enough to leave a path that is possible to detect. Following each event, computers collect and interpret the vast quantity of data from the detectors and present the extrapolated results to the physicist. In order to achieve this, three different systems have been developed, the trigger system, the data acquisition system

[1] University of Oklahoma, High Energy Physics

and the computing system. These systems will be further introduced after the composition of the ATLAS detector has been described.

The ATLAS detector is composed of four major components, which test for different aspects of an event, the Inner Detector, Calorimeters, Muon Spectrometer and Magnet system. Each component is sensitive to various particle properties and detects a particle with reference to their properties. The components are arranged in order so that all particles will go through the four layers sequentially, in order to obtain maximum amount of data about the particles produced by an event. However the region of interest in this thesis is Inner Detector and is explained in detail.

1.1.1 Inner Detector

The Inner Detector reconstructs the tracks and vertices of each event, contributing to particle recognition and additional information about short-lived particles, by studying their decay products. It consists of three tightly integrated, radiation-hardened systems of sensors all immersed in a magnetic field parallel to the beam axis. The innermost sensors closest to the collision point are the semiconducting pixel detectors. Further out are the semiconductor tracking (SCT) detector, and the outermost sensors are the Transition Radiation Tracker (TRT). This thesis will feature only Pixel detectors.

1.1.1.1 Pixel Detector

The pixels are made from silicon wafers etched with conducting strips and divided into pixels of 50 by 300 μm. Altogether there are approximately 140 million pixels, which are

placed in a cylindrical array closest to the interaction point and placed on disks further away. A charged particle traversing through such a layer produces a signal that identifies which pixel has been traversed, thereby providing gives a precise measure of the position of the particle. This position is precise enough to determine whether the particle originated at the proton-proton collision point, or a few millimeters from it as a decay product of another particle.

1.1.2 SCT and Pixel Off-Detector Electronics

The off-detector electronics are housed off the detector, in contrast to the front-end electronics that are located on the ATLAS detector. The ATLAS Pixel and SCT ROD groups at LBNL and University of Wisconsin will provide the design, tailored for the specific needs of the SCT and pixel detectors, whereas the responsibility of the off-detector electronics for the TRT belongs to another group. This thesis will only consider the pixel specific electronics. The SCT and pixel off-detector electronics consists basically of the Read-Out Driver (ROD), Timing and Trigger control (TTC), Trigger Interface Module (TIM), ROD Crate Controller (RCC) and Back Of Crate (BOC) module as well as some power supplies. The interconnection between the SCT/pixel off-detector electronics is provided by a custom backplane. However this thesis will focus only on pixel off-detector electronics.

1.1.2.1 Readout Driver Electronics

The Readout Driver Module is the sole communication path between detector specific Front-end electronics and the outside world. The ATLAS SCT and Pixel Readout Driver

(ROD) Group, Lawrence Berkeley National Laboratory, California, presently works on a part of the readout system for the ATLAS detector, the ROD. The ROD will be specifically designed to suit the particular needs of the pixel detector housed in the inner detector electronics. In this thesis the device responsible for decoding the data from the Module Control Chip has been designed, which is located on each Pixel module. The Pixel ROD will have the same basic design as the SCT ROD and is implemented using FPGAs.

1.2 Background of Field Programmable Gate arrays

Recently, Field Programmable Gate Arrays (FPGA) have played an important role in the design of logic circuits. Standard off-the-shelf ICs have fixed functions defined by chip manufacturers. On the other hand, functions are defined for both ASICs and FPGAs for each application. An FPGA is a completely manufactured, design-independent device. Unlike an ASIC, an FPGA does not require a final manufacturing process to customize its operation. Each FPGA vendor manufactures devices with a proprietary architecture. However, the architecture includes a number of programmable logic blocks connected to programmable switching matrices. To configure a device for a particular functional operation, switching matrices to route signals between the individual logic blocks are programmed by the designer.

Field Programmable Gate Arrays consist of programmable cells across the silicon. A basic cell consists of a number of transistors, and there may be multiple rows of cells. The Channels between the rows of cells is used for interconnecting the basic cells during

final customization. Gate arrays contain from a few thousand to several million equivalent gates. The library of cells provided in a gate array contains primitive logic gates, registers, and hard and soft macros. Manufacturers define hard macros in terms of cell primitives. In comparison, the designer characterizes soft macros.

Hard and soft macro libraries contain elements of Large Scale Integration (LSI) and Very Large Scale Integration (VLSI) complexity, such as controllers, Arithmetic Logic Units (ALUs), and Microprocessors. Additionally, soft-macro libraries contain RAM functions that cannot be efficiently implemented in gate-array devices. On the other hand, ROM functions can be implemented more efficiently in cell Primitives.

The advantages of FPGAs are derived from its general-purpose array architecture. It has an interior matrix of logic blocks and a surrounding ring of I/O interface blocks. User programmable interconnection resources are used to create logic network from these elements. The functions of the logic and I/O blocks and the routing of interconnect networks are defined by a configuration program stored in an internal memory. Until configured by the user, each device is identical. A processor programs the device at power-up, or upon request while the system is running. Apart from that the advantage of FPGAs is that they are quick and easy to program, or customize. Also, FPGAs allow pc-board CAD layout to begin while internal FPGA design is being completed. This procedure lets the designer perform early hardware and software integration testing. If system testing fails, the designer can modify the design and immediately program another

FPGA at a relatively low cost. For these reasons, designers often initially target hardware designs to FPGA devices for system testing and small production runs.

1.3 HDL

An Hardware Description Language (HDL) is a software-programming language used to model the intended operation of a piece of hardware. There are two aspects to the description of hardware facilitated by an HDL, true abstract behavioral modeling and hardware structure modeling. An HDL is declarative to facilitate the abstract description of hardware behavior for specification. Structural and design aspects of hardware do not affect the behavior. One can model hardware structure in an HDL irrespective of the design's behavior. One can also model and represent hardware behavior at various levels of abstraction during design. Higher-level models describe the operation of hardware abstractly and lower level models include more detail, such as inferred hardware structure.

1.3.1 VHDL Modeling and Synthesis

The choice of an HDL involves the analysis of its characteristics: the hardware platform independence, Schematic capture tools integration, the simulation, the logic synthesis, the models diversity, the standardization and accessible cost. A digital circuit can be modeled in Very high speed integrated circuits Hardware Description Language (VHDL) in three description styles: structural, data-flow and behavioral. Structural descriptions present a larger detail and are hierarchical arrangements of components. In the data-flow descriptions, the digital circuit control and data flows are represented. The instructions

and the concurrent signal statements determine the relationship between the input and the output. Behavioral descriptions have less detail. They are used for digital circuit verification and functional simulation. Input to Output relationship is specified by the VHDL processor.

1.31.1 Top-down Modeling

A true top-down, system-level design methodology would enable complete systems to be described at an abstract level using an HDL and automated tools, such as partitioners and synthesizers, for implementation. This approach drives the abstract-level description to implementation on pc boards or multichip modules, which can contain standard ICs, ASICs, FPGAs, and PLD and full-custom ICs. Provided the budget is available for simulation and synthesis tools, a top-down design approach using an HDL is, by far, the best design methodology. There are many advantages of adopting a top-down design methodology. In using schematics to design an ASIC with large number of gates, a small design change can result in time-consuming changes to the schematics. Using an HDL to develop electronic hardware is similar to doing a software-development project using a high-level programming language, such as C.

A top-down design methodology takes the HDL model of hardware, written at a high level of behavioral abstraction (system or algorithmic), down through intermediate levels, to a low (gate or transistor) level. The term "behavioral" means the behavior of intended hardware, independent of the level of abstraction at which we model it. A design represented at the gate level still represents the behavior of hardware intent. As hardware

models are translated progressively lower levels, they become more complex and contain more structural detail. This modeling methodology reduces design complexity. Hardware structure is ignored when modeling hardware at the two highest behavioral levels.

1.4 Organization of the thesis

The thesis will first present the Pixel Detector Electronics, readout protocols and data formats along with the design specifications in Chapter 2, where the reader will be introduced to the functionality and composition of each of them. It also explains about the proposed implementation model and introduces to the reader the various functional blocks of the ROD. Next in Chapter 3, an in-depth analysis of the Formatter FPGA is presented along with its design and implementation. Chapter 4 explains the design of two more modules in the data path, the Event Fragment Builder and Router FPGAs. The design and implementation of the ROD Resources Interface FPGA is explained in Chapter 5. This Module is present in the ROD controller and is responsible for the operation of the ROD. Chapter 6 contains a summary, conclusion and directions for future work on the ROD.

CHAPTER 2

Protocols and Readout Driver Electronics

This chapter explains in detail the data format and protocols of the data from the Module Control Chip (MCC). A brief description of the MCC is given in order to help understand the detector on the whole. The ROD designed will decode the data transmitted by these chips. This chapter will also explain a whole range of specifications needed for the design of the ROD.

2.1 Module Control Chip

The Module Control Chip (MCC) is a digital IC located on top of each pixel module that coordinates the 16 FE ICs that are placed on the same Module. It collects the data from the End of Column (EoC) logic circuits, builds the events and handles error conditions. It also manages the data input/output toward the Read Out Drivers (RODs), including the initialization and storage of parameters in the FE ICs. It has 3 main basic functions.

- FE chip readout and event building.
- Loading of parameter and configuration data into chips and into the MCC itself.
- Distribution of timing signals like bunch crossing clock (The unit of time between consecutive bunch crossings of the beam in the LHC machine (25 ns) and LV1 trigger (The first-level trigger will take its signals from calorimeters and muon detectors and must accept a new set of data every 25 nanoseconds (the time between bunch collisions)).

2.1.1 Output Data format of the data from Module Control Chip to ROD

The format of the event generated by the MCC has been defined considering that:

- **Events are ordered by LV1 arrival time.**

  Event building is done by grouping all hits (when the voltage generated by a pixel crosses a threshold voltage its a hit) belonging to the same LV1 number. The first event, which has been triggered, is the first to be sent out. Every event is entirely transmitted before the next event is considered for transmission: no event interleaving is allowed.

- **Data are sorted and grouped by FE chips.**

  The event builder sorts pixel hits by FE chip order. This permits data compression by "clustering" for either the case of a non-uniform hit distribution (jets) or a large pixel occupancy (B-layer) since every hit does not need to extend the address information to include the FE chip number with its row and column address.

- **Event length is not known until the whole event is transmitted.**

  The event builder does not know until the event is completely built up what the total number of hits is, or what the number of hits per FE chip is. This makes the event builder simpler, but forbids the usage of forward word counters in the event frame. Event data fields must be recognized by their content.

- **Data must be compressed due to bandwidth limit.**

  Since we need to keep the number of transmission cables as small as possible, and since a rather large amount of data is transmitted from the MCC, event coding, which compresses data, is specially important.

- **Recovery from transmission errors.**

  Any transmission error must be recovered by the next event data. The chosen mechanism is to use a trailer whose value can never occur in the data. Since data fields are allowed to take any value we decided to add a Sync bit (bit set to '1') after every data field and a trailer containing a fixed number of zeroes.

- **Null data (wait).**

  In the coding the possibility of informing the receiver that the event is not finished yet, in case the MCC cannot supply new data, is included. This is optional for the MCC, but the design of the ROD must include this option for generality.

  An event frame at the output of the MCC is an ordered stream of data fields separated by synchronization bits.

2.1.2 Syntax for the event data

<Event>

::= <Header> L1ID <S> BCID <MccFlag>? <FrontEnd>* <Trailer>

<Header>

::= 11101

<S>

::= 1

<MccFlag>

::= <Sync> MCC-FLAG

<Sync>

::= <S>

||= <<S> NULL>+

<FrontEnd>

::= <Sync> MCC-FE# <Hit>+ <FeFlag>?

||= <Sync> MCC-FE# <Hit>* <FeFlag>

<Hit>

::= <Sync> ROW# COL#

||= <Sync> ROW# COL# TOT (if opt. ToT selected)

<FeFlag>

::= <Sync> FE-FLAG

<Trailer>

::= <S> 00 0000 0000 0000

||= <S> 00 0000 0000 0000 0000 0000 (if opt. ToT selected)

The following items summarize the format of the formal syntax description:

<Name>	name in lower case is a syntax construct defined by other syntax constructs or by a bit field.
NAME	in upper case is a bit field. Its definition is stated in Table 2.1.
<Name>?	is an optional field, 0 or 1 items occurrence.
<Name>*	is 0, 1 or more items.
<Name>+	is 1 or more items.
::=	gives a syntax definition to an item.
||=	introduces an alternative syntax definition

Table 2.1. Values of the keywords that appear in the event syntax

Keyword	Low Value	High Value	Description
BCID	0000 0000	1111 1111	Bunch crossing ID (0:255)
COL	0 0000	1 0111	Column number
FE-FLAG	1 1110	FFFF MMMM	Error/Warning: F=FE, M=MCC
LVID	0000 0000	1111 1111	Level 1 Trigger ID (0:225)
MCC-FE#	1110 0000	1110 1111	FE number in the module from 0 to 15
MCC-FLAG	0000 0000	1101 1111	Error / Warning or Message from the MCC
NULL	1111 1111		Null data
ROW#	0000 0000	1101 1111	Row number from 0 to 224
TOT	0000 0000	1111 1111	Time over Threshold (if opt. ToT is selected)

2.1.3 MCC to ROD Physical Layer Protocol

The MCC has different ways to transmit data to the ROD through the opto-links. Each link can be used as a single 40 or 80 Mb/s link. In addition the two links can operate as a single or a dual link. This allows peak bandwidth going from 40Mb/s to 160 Mb/s. There are four modes of operation. The four link modes are selected by the two bits in CSR. In each case of mode 40 x 1 and 80 x 1, the two links always transmit the same information and either one can be used. In case of dual link usage (40x2 and 80 x 2) the first bit of every event record always starts on DTO[1]-1 output, while the last bit may be either on DTO-1 or on DTO-O. All configuration data, instead, is always transmitted at 40 Mb/s on

[1] Data output

both links independently from the link mode setting. The ROD must be able to receive multiple event packets from an MCC for each L1A command issued. The number of event packets received per L1A issued will not change from event to event without the ROD first being reconfigured. However, the number need not be the same from link to link. The maximum number of event packets received from the MCC will not exceed sixteen. The MCC is configured to transmit multiple L1A commands to the pixel front end chips for each L1A command received from the ROD. In that case, the MCC will transmit a complete event packet to the ROD for each L1A, which is issued to the pixel front-end chips. The ROD must support this feature. The ROD design and implementation is explained in detail in the next chapter.

2.1.4 Input Command Set For Pixel RODs

Commands are divided into three groups. Trigger, fast and slow. In the trigger command group the only command is LV1. In the fast command group there are four commands and the slow command group allows up to 16 different commands out of which only 10 different commands are implemented in the MCC.

2.1.4.1 Trigger commands

LV1: Level 1 Trigger

In the trigger group the only command is level 1 Trigger (LV1). The Level 1 Trigger is executed if the MCC is in the RUN mode. The LV1 Triggers the acquisition of a new event in the Module and the main actions in the MCC are the following.

- LV1 is issued to the front end chips (FE chips), through MCC-LV1 output pin.

- The Corresponding L1ID and the BCID numbers are stored in the MCC and are used by the event Builder when the event is reconstructed.
- The L1ID counter, which keeps track of the Level 1 Trigger, is incremented.

2.1.4.2 Fast Commands

BCR: Bunch Counter Reset

In response to a BCR command, The BCR Counter in the MCC is reset to 0.This command does not affect the FE Chips.

Table 2.2. Control Commands

Type	Name	Field 1	Field 2	Field 3	Description
Trigger	LV1	11101			Level 1 Trigger
Fast	BCR	10110	0001		Bunch Counter reset
Fast	ECR	10110	0010		Event Counter reset
Fast	CAL	10110	0100		Calibration Pulse
Fast	SYNC	10110	1000		SyncFE signal
Slow	CMD	10110	1011	Command	Slow Command Header

ECR: Event Counter Reset

The ECR Command clears the Event Scoreboard. The value of L1ID is set to 0 and the system is ready for a new event. If there are pending events to be built they are all cleared. This command does not affect the FE chips.

CAL: Calibration

This is a pulse, which is generated by the MCC in response to a CAL command. The pulse width, in CK units is defined by the contents of CAL <8:0> register. While the delay of the pulse in approximately 0.5 micro seconds is defined by the contents of the CAL <15:9> register.

SYNC: SyncFE Signal

A SyncFE signal of one CK unit is generated by the MCC.

2.1.5 Timing requirements

- There are no restrictions on LV1 Commands: LV1s can follow each other without any gap. This implies that the minimum time between consecutive LV1 commands is 125ns, conforming to ATLAS specifications.

Table 2.3. Command timings

Command	Following Command	Time (CK units)
LV1	LV1	0
LV1	Fast, Slow	0
Fast	LV1	0
Fast: BCR	Fast, Slow	2
Fast: CAL, SYNC	Fast, Slow	0
Fast: ECR	Fast, Slow	1
Slow	Any Command	No specification

- Fast and slow commands can be issued without any gap. However a gap of 50ns is recommended between consecutive commands in order to minimize the risk of command misinterpretation in case of a bit flip. The requirements are listed in Table 2.3.
- After a slow command, if there is a bit flip, no prediction of the misinterpreted commands is assumed, due to the presence of variable length data field.

2.2 ROD

The Read Out Driver is a detector-specific device of the data acquisition electronics, situated in between the derandomizers and the ROB, which manages gathering of data from all FE circuits and provides data concentration on a higher level. The signals are formatted prior to be transferred to the Read Out Buffer (ROB). The ROD also performs error checking, zero-suppression and buffering before shipping the data to the next stage over high-speed optical fiber links.

2.2.1 The ROD crate

The ROD crate is a modular element, which means that it is replicated in the system in

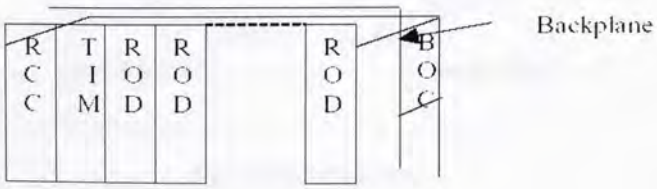


Fig. 2.1 The ROD crate.

many independent copies. There are sixteen RODs situated in each ROD crate for the SCT and pixel read out system. Each ROD interfaces with the Trigger Interface Module (TIM) and ROD Control Crate (RCC) via the backplane, and with the ROB via the Back of Crate (BOC) card. For the SCT, each ROD crate will serve 32 SCT detector modules (each half module contains data from a complete event) and for the pixel each ROD crate will serve 16 pixel detector modules. The SCT readout system will be housed in approximately three crates, while the pixel readout system will be housed in four crates.

2.3 ATLAS Pixel ROD

2.3.1 General Functionality of the ROD/dataflow

The pixel readout modules have a similar design with common general functions, which will be described at first, and the differences between the modules will be further explored below. The readout driver module works as the communication path between

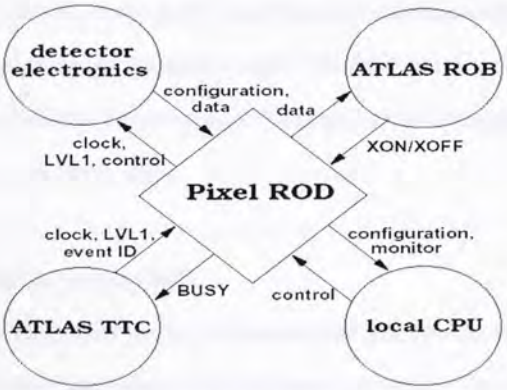


Fig. 2.2 ROD Data Flow

the FE electronics and the outside world. The ROD interfaces the detector electronics to the ROB, TTC and to the RCC (the local CPU), as shown in the Fig. 2.2.

2.3.2 Dataflow

The Trigger, Timing and Control system (TTC) transmits the level-1 trigger along with the event ID and the 40 MHz clock to the ROD so that the ROD can distribute the trigger to the front-end system together with the global clock and some control signals. The front-end electronics in turn sends back the required detector data for the ROD to process it. The ROD performs formatting, data collection, error checking, and buffering before sending the data to the ROB over an optical high-speed link. In case the readout buffer occupancy would be too high, an XON/XOFF mechanism is implemented on the readout link. When the buffer is nearly full the ROB asserts an XOFF bit to the ROD, which in turn provides a ROD_BUSY signal to the TTC system in order to stop the LV1 generation. The local CPU or the ROD Crate Controller coordinates the activities in the crate and communicates with the outside world. The ROD receives control commands from the RCC and transmits front-end and ROD status information and register contents, as well calibration data, to the RCC.

2.4 Functional Requirements of ROD

Input data streams are provided by the opto-receiver data streams from the BOC card. Input data can be any of three types: event, calibration, or register. The following requirements apply to receipt of data of any type.

- The ROD must be capable of its required performance whether input data from a pixel detector module arrives as normal, partly on each of two input data streams, or whether all input data from a module arrives on a single input data stream. Since each pixel detector module is instrumented with two output data fibers in order to provide redundancy, in case of failure of one output data stream, all module data is routed to the other fiber. The performance of the ROD should not be degraded in case of such a failure.
- The ROD must be capable of receiving data of three data types (event data, calibration data, and register data) on each input data stream. It must be capable of receiving these three data types from pixel modules. It must be capable of distinguishing register data from event and calibration data. The three data types have similar, but not identical formats for the pixels. Consequently, the ROD must be able to recognize the three data types and must be able to distinguish each type in order to appropriately process the incoming data. calibration data can be distinguished from event data by the prior issuance of a calibration command.
- The ROD is not required to be capable of simultaneously receiving data of different data types on different input data streams; however, simultaneous receipt of different data types on different input data streams must not cause ROD to become incapable of receiving subsequent input data of any given type.
- The ROD must merge input data records from the input data streams into output data records on an output data stream.
- The ROD must be capable of being configured from Versa Module Europe (VME) so as to eliminate data from any specified set of input data streams. All data on the

specified set of data streams should be eliminated from the time that the ROD is so configured until it is reconfigured to accept data on those input streams. This requirement allows masking of input data streams, e.g. those that carry invalid data. VME provides a convenient mechanism to control masking of input data streams.

- The ROD must be capable of being configured via VME such that it can operate normally if no signal is received on any one of a specified set of input data streams. This requirement allows masking of input data streams that carry no data. The ROD must be capable of receiving data on the input data streams that is formatted according to the protocol defined in the specifications of the production pixel module controller chip, MCC. Input data streams are produced in the detector mounted electronics in the pixels by MCCs.
- The ROD must be capable of receiving all data on the input data streams without loss of any valid data, except when downstream devices are non-functional. In the case of data loss during these circumstances, the ROD must flag the loss of data in the output data stream. Data lost by the ROD cannot be recovered. The readout system is designed to minimize data loss. The ROD can be designed in such a way that its data losses are negligible compared to data losses in the detector mounted electronics. If data is lost, even in very unusual circumstances, it should be flagged in the output data stream in order that subsequent processing can consider that relevant data may have been lost.
- The ROD must recognize error flags in the input data stream, and it must flag the occurrence in the output data stream, since MCCs flag some errors in the data streams that they transmit, the ROD should pass these flags to the data acquisition system.

- The ROD must recognize protocol errors in the input data stream, and it must flag the occurrence in the output data stream. Protocol errors which the ROD will recognize and report are event headers with a single bit error, illegal data fields, missing synchronization bits, trailers with a bit error and out of place trailers. Since unexpected errors in the protocol of the input data stream could signal invalid data or could cause incorrect interpretation, the possibility of invalid data must be flagged in the output data stream for consideration in subsequent processing.
- The ROD must respond to errors identified as fatal by performing a VME interrupt, and it must flag their occurrence in the output data stream. Fatal errors are caused by conditions such as overflow of buffers on the detector mounted electronics or losses of event synchronization that require resynchronization of the detector mounted electronics or the ROD. The VME interrupt serves to notify the RCC, which can interrogate the ROD in order to determine what action should be taken.
- The ROD must provide a mechanism to monitor the occurrence and frequency in each data stream of protocol errors and of error flags. The occurrence of errors must be monitored in cases where a response is required. The frequency of errors must be monitored in order that serious malfunctions can be distinguished from occasional or innocuous errors.
- The capability to pass raw data to the output of the ROD must exist in order to, for example, facilitate the diagnosis of problems with the ROD or with the detector mounted electronics. Detection of the unique bit patterns of headers and trailers is necessary for recognizing the beginning and ends of incoming records, consequently, identification headers and trailers cannot be disabled.

- No bit pattern on an input data stream must cause the ROD to crash. These include, but are not limited to, protocol errors, timeouts while waiting for expected data and timeouts while waiting for a trailer, i.e. a link that sends a record that exceeds the maximum possible length. Any of these situations may arise due to errors in other parts of the system. When possible, it is better if the ROD handles these errors locally or, in the case of sufficiently serious errors, interrupts the RCC, without interfering with data acquisition.
- The ROD must buffer output event data records for readout via VME or for transmission via Readout Links to Readout Buffer (ROB) modules. The ROD will be read out via VME or via Readout Links. It must buffer output event data records until they are read out.

2.5 Functional blocks of the ROD

The detector data arrive from the front-end electronic system to the readout module on fiber ribbons operating at 40 Mbps, via a custom optical receiver system. The BOC card is the device within the ROD crate, which handles the receiving of the optical signals and converts them into electrical signals. Each link transfers event fragment data for one complete event if the data is originated from the pixel detector modules. For the pixel detector data transmission, every two or every four (depending on which part of the pixel detector the data is originated from) links will carry data from one complete event.

The BOC system distributes the "optical to electrical" converted data on to the next level, where the serial to parallel conversion is performed in the formatter. In addition to

parallel to serial conversion, the formatter must also perform error checking and flag for errors in the input data. Due to the random arrival of the data, a derandomizing buffer is needed. For this purpose first-in first-out buffers are implemented. The FIFOs will also prevent data overflow by asserting a data flow control signal to the TTC in order to throttle the trigger. The ROD controller has the function of control and coordination of operations of the various other functional blocks of the ROD as well as managing control and configuration commands to the front-end electronics. An output control stage directs the flow of the output data. Two destinations are foreseen, the link to the ROB and a memory on the ROD that can be read from the VME. The event data are sent to the ROB via an electrical to optical conversion stage, over an optical link, the S-link. For monitoring reasons, the data are also written to the memory, except event data and calibration data.

2.6 Implementation Model Overview

The following implementation model is intended to familiarize the reader with the overall design details of the Hybrid ROD. ROD implements a token passing data push architecture in all stages. This means that when data is available for shipment to subsequent stages, the process begins immediately and proceeds in an orderly sequential manner without requiring intervention of, or control from, down-stream stages. This model is illustrated in Fig. 2.3. This model has a set of Formatter FPGAs, a Dual Event Fragment Builder, a pair of event fragment builder FIFOs, a Router, a ROD controller, Digital Signal Processors (DSPs) and modules related to the trigger interface Module and BOC. In this thesis the Formatter FPGA, Event Fragment Builder, Router and the ROD

Controller are implemented. Fig. 2.3 only illustrates a schematic data flow in the ROD. It does not specify the exact number of Formatter FPGAs in the ROD or links to the ROD, which is different for different layers of pixel detectors.

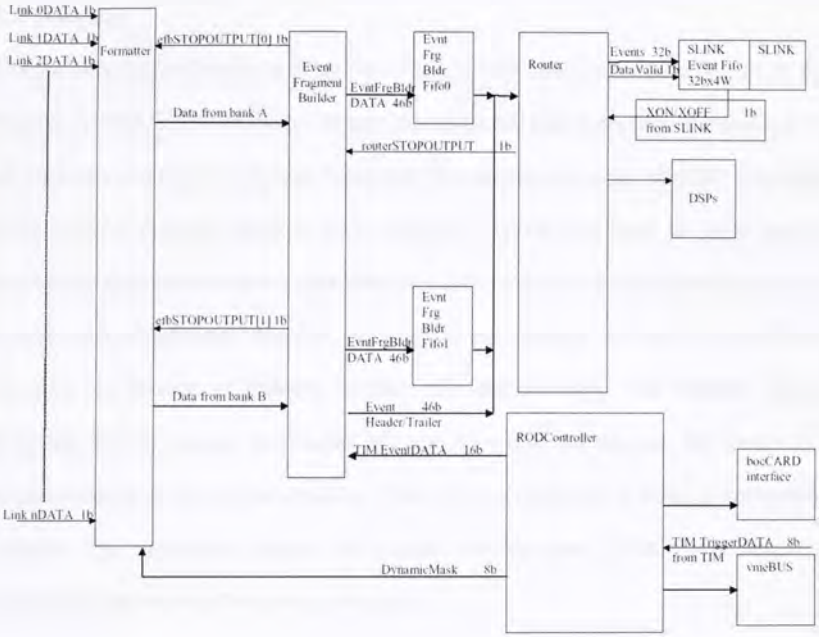


Fig. 2.3 ROD Implementation Model

2.7 Design Methodology

The design of the ROD architecture and selection of algorithms was based on simulation studies of the detailed response of ATLAS detector to physics and background events. A major fraction of the design effort was dedicated to the implementation of the algorithms described in VHDL code to operate within FPGAs. Design and prototyping efforts were

greatly speeded up by the use of a fully integrated Computer aided system (CAD) toolset that included the VHDL hardware design language, synthesis, simulation and automated board layout tool namely Mentor Graphics® [13].

2.8 Summary

The fundamental difference between the RODs for different layers of ROD will be the number of links. However the L1 Trigger command and fast commands are identical for all the layers coming from just one input link. This facilitates having one ROD Controller design for the different layers of Pixel detectors. For the data from the pixel layer 1 module the data arrives via two data links, however from pixel B-layer modules the data arrives via four data links, therefore the design of the formatter will have to be different for each one because of different decoder state machine logic. The Gatherer (Event Fragment Builder engine) and Router will also have identical designs. The design and implementation of the various modules of the ROD is explained in detail in subsequent chapters. The subsequent chapters will explain the Formatter FPGA, Event Fragment Builder, Router and ROD Resources Interface.

CHAPTER 3

Formatter FPGA

This chapter will discuss in detail the design of the Formatters for the Pixel Layer 1 and the B-layer. The Formatter is the first block downstream in the ROD. It decodes the data stream from the MCC and reformats and packs them into 32 bit words and sends them to the Event Fragment Builder for further processing. The Formatter has the task of identifying the various data types coming from the MCC and asserting the corresponding signals high when writing the output. A ROD has number of Formatters depending upon the different Pixel detector layers. This chapter also discusses the number of Formatters in each of the cases.

3.1 Formatter FPGA

Each link's decoder writes data into its own FIFO buffer as and when data is available from its serial link. The link's FIFO controller collects this data and when it possesses the token, begins to read it out as soon as complete formatted words are available in the FIFO buffer and not waiting for an event trailer. Upon completion of an event readout, the token, representing ownership of the data bus for the current event and link is passed from the active link to the next link for readout of its information. This process continues in sequential fashion until each link has received the token, read out its data, and the token has returned to the token controller. If more data is available at that time, the token controller issues the token to the first link, and the process repeats. Otherwise the token is stored in the token controller and idles there until triggers are again received. A formatter

FPGA consists of Link Formatters, Formatter FIFOs, Formatter write in and readout state machines. The number of link Formatters and the size of the FIFOs being different for each layer are explained in the subsequent sections. The Top level of the Formatter FPGA is depicted in Fig. 3.1.

3.1.1 Formatter FPGA Layer 1

In the case of Layer 1 there are totally 13 input Modules per ROD at 80 Mbps/module, which is equivalent to 26 links at a data rate of 40 Mbps/link. These 26 links are divided

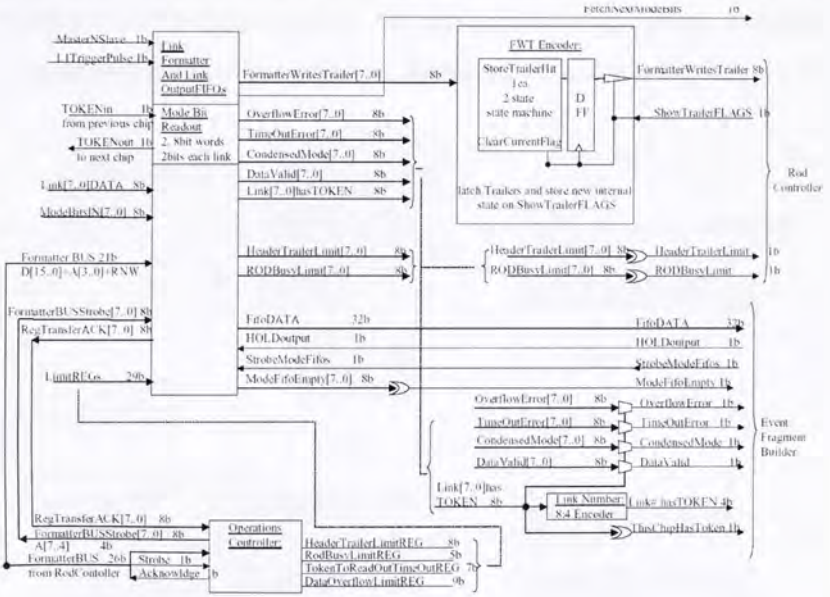


Fig. 3.1 Formatter FPGA

into four banks, with each bank having eight 40 Mbps links, equivalent to four 80 Mbps links being input to one Formatter FPGA. Six of these links are masked off by the DSP.

This masking operation is done by the DSP. Therefore the number of Link Formatters in each Formatter FPGA is four, each formatter decoding the data from two links of the same module. The decoded data from the Link formatter is written into a FIFO as and when they come.

3.1.1.1 Link Formatter and Link Output FIFO Design for Layer 1

A Link formatter and link output FIFO block consists of 4 link formatters and four Output FIFOs. Each Link Formatter outputs the decoded data into its FIFO. A FIFO Size of 1024x32 is provided for each link. The exact mapping of Link Formatter and Link output FIFO is shown in Fig. 3.2. The first link Formatter is input with links 0 and 1,

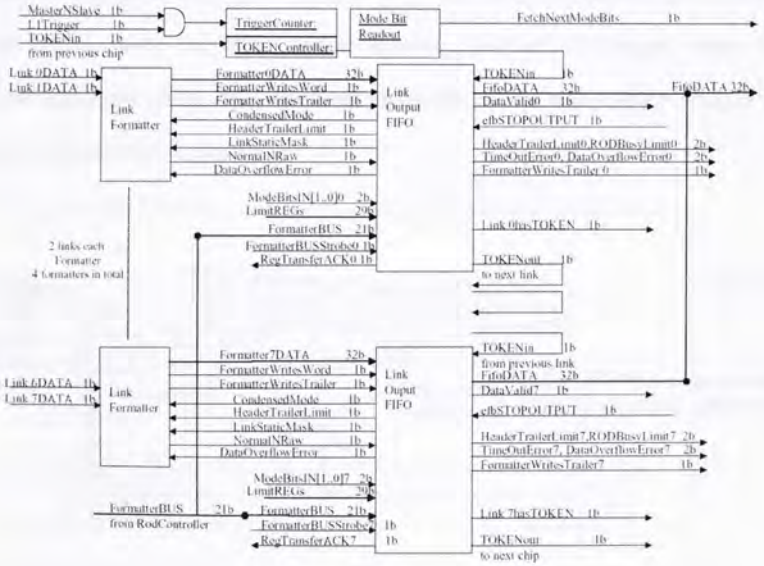


Fig. 3.2 Link Formatter and Link Output FIFO Mapping (layer 1)

the second with 2 and 3, the third with links 4 and 5 and the fourth with links 6 and 7. The signals indicated in this diagram will be explained in detail in subsequent sections.

3.1.1.1.1 Link Formatter Layer 1

The Link Formatter is designed to decode the data from the MCC. Each Link Formatter is expected to decode the data from one module. It is comprised of various Finite State Machines (FSMs) that were optimized for the task to be carried out by them. These Formatters receive detector data from the MCC, convert the serial data into parallel data and further transmit the parallel data to the FIFO Controller. The basic structure of a Link Formatter is represented in Fig. 3.3. Each of these four formatters decodes the serial data stream arriving from the MCC module searching for hits, FE flagged errors and synchronization bit errors. The Formatter FIFOs are intermediate buffers designed to store the output of the Formatters.

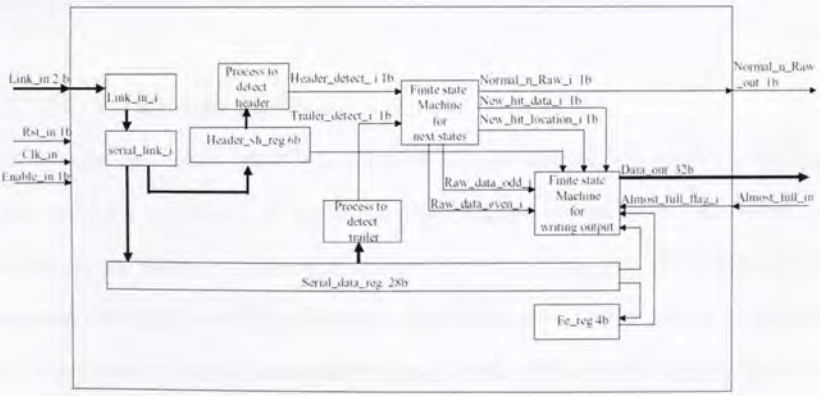


Fig. 3.3 Block Diagram of Link Formatter illustrating input and output signals

The format of the event arriving to the formatter is organized in fields separated by synchronization bit (1), starting with a header and ending with a trailer. The header is to wake up the receiver from the idle condition when no data is transmitted, while the purpose of the synchronization bit is to signal the end of one data stream and the purpose of the trailer is to define the end of an event. The trailer has been chosen to be a pattern that can never occur in the data stream, due to the insertion of the synchronization bit. The event data format requires a header at the beginning followed by level-1 ID (L1ID) and Bunch Crossing ID (BCID). This could be followed by a FE chip number, followed by a number of hits (row plus column), interrupted by another FE chip number plus the sequence of hits and so on. The format can also include warning and error flags.

However, the pixel event data format and the SCT event data format differ in the header. The header of the event data originating from the pixel detectors is a sequence 11101, while the header of the SCT event data is a sequence of numbers; 111010.

3.1.1.1.1.1 Decoder State Machine

The Decoder Finite State Machine in the formatter is in idle state till it detects a header. Once a header is detected, it checks for a valid L1ID and BCID. In the process of identifying the Header, the formatter allows a tolerance of one bit. This implies that a sequence with a pattern differing from the standard Header Pattern by one bit is allowed. If L1ID is detected then the next expected data is BCID . When Both L1ID and BCID are valid it starts decoding the data stream till it encounters a trailer, to know where the event

ends. After a BCID is detected a Synchronizing bit has to be detected. If a synchronizing bit is not detected then the Formatter Flips to the raw data mode. Once the State machine slips into raw data mode it asserts Normal_not_raw high. Once the Pixel decoder slips into raw data Mode it never recovers from that mode until it detects Valid Trailer. When a valid BCID is detected the next data can be either MCC FLAG or the FEID. If MCC FLAG has been detected the next data expected is FEID. The input data format to the formatter is listed in the table below.

Table 3.1 The Input Data Format

Header	11101
L1ID	(00000000-11111111)
BCID	1(00000000-11111111)
MCC FLAG	1(00000000-11011111)
FEID	1(11100000-11101111)
HIT	1(00000000-11011111)&&(00000-10111)&&(00000000-11111111)
FE FLAG	1(11110000-11111111)
TRAILER	1(00000000000000000000)

The Data expected after FEID is either Front end Error Flag or the Hit data. In the absence of any error the data is hit data. If a Hit data is not detected then a FE Flag is detected. It is also possible that a Hit is preceded by a FE Flag. If a new hit data is detected irrespective of whether it has occurred at the same front end ID or not, 'New_Hit_data' is asserted high. The state machine state diagram of the decoding machine is shown in Fig. 3.5. When the state machine shifts from one state to another the bit_count is set to a new value. This bit count keeps track of the number of bits being written into the Shift register. When a L1ID is detected and all the bits are buffered into the shift register write L1ID is asserted high, similarly write BCID is asserted high when a BCID is buffered into the shift register. When a MCC FLAG has been detected

completely the state machine asserts write MCC FLAG high. Similarly when a FEID has been detected and all bits have been buffered write FEID is asserted high. It is also likely that there may be multiple hits at the same Front End ID. This includes the possibility of getting Hit data again and again continuously. Once the Pixel Decoder encounters a trailer the event is complete.

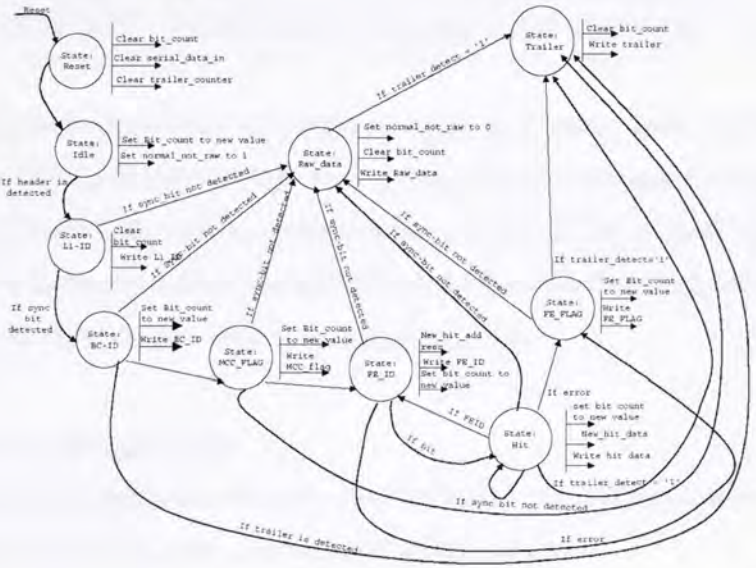


Fig. 3.4 Finite state Machine State Diagram of the Pixel Formatter

3.1.1.1.1.2 Output Write in State Machine

Whenever the decoder state machine shifts from one state to another, the Output Write in state machine writes the Serial Data Bits stored in the Serial Data Register into a 32bit word. The Output Data Format of the Decoder state Machine is given in the Table 3.2.

Table 3.2 Output Data Format

Event-ID	000HXXXXXXXXXXXXLLLLLLLLBBBBBBBB
MCC FLAG	101XXXXXXXXXXXXXXXXXXXXXMMMMMMMM
HIT Data	11XXXXXFFFFRRRRRRRRCCCCCTTTTTTTT
FE FLAG	100XXXXXXXXXXXXXXXXXXXXXFFFFEEEE
TRAILER	001TeXXXXXXXXXXXXXXXXXXXXXXXXXXX
RAW DATA	01DDDDDDDDDDDDDDDDDDDDDDDDDDDD

When the Trailer is written to the output, the signal, decoder_writes_trailer, is asserted high indicating the end of an event. Thus, the Output data from the Formatter is a 32-bit Word. The 3 higher order bits identify the output data written by the Formatter, except While Writing HIT Data and Raw data where the two higher order bits identify the type of data. The key for the output data format is given Appendix C.

3.1.1.1.2 Link Output FIFO

The Basic Building blocks of the Link output FIFO are FIFO, Write in State machine and readout state machine (FIFO state machines) and FIFO Controller.

3.1.1.1.2.1 FIFO

The size of the FIFO for each link is 1024x32.This was decided after taking into account the device being used. The design was accomplished by using a total of 32 block RAMs each 256 words deep and 16 bits wide. The 28 block RAMs are addressed commonly. Two block RAMs are tied together and addressed commonly to form a 256x32 FIFO. Fourteen such pairs are addressed together to form a 1024x32 FIFO. The input data is

written into the corresponding pair of Block RAMs by a write in logic. Depending on the Write in address the Block RAMs are enabled. If the write in address is less than 255 then the write enable for the first pair of block RAMs are asserted high and all the other write enables are asserted low. The read enable logic also works similar to the write enable logic. If the readout address is greater than 255 but less than 511, then the read enable of the second set of block RAMs are asserted high and the read enable signals for other block RAMs are asserted low.

3.1.1.1.2.2 Choice of the Size of FIFO

The size of the FIFO was decided after calculating the number of input modules to each FPGA. Consistent with ROD implementation style, each FPGA is input with eight

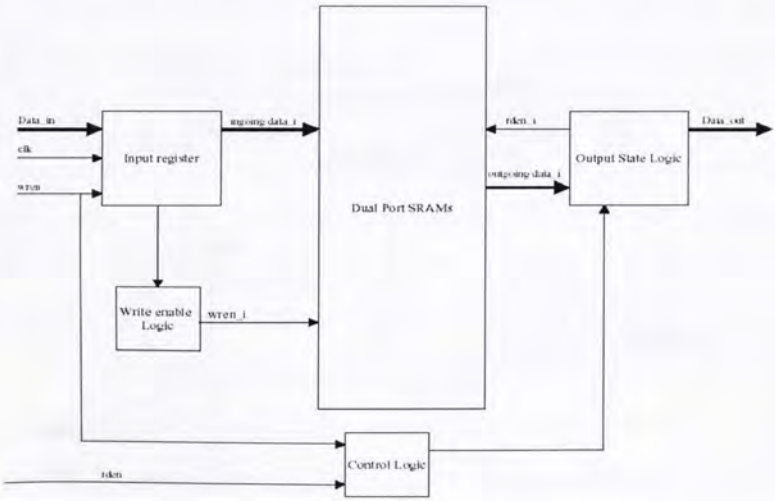


Fig. 3.5 FIFO Implementation

40 MHz links and an even number of FPGAs are used in the design process. It may be recalled that in this case (Pixel ROD layer 1) there are a total of 26 input links at a data rate of 40 MHz. Therefore it was decided to use 4 FPGAs with each ROD. However since only 26 modules were being input, 6 links are masked off. Xilinx Virtex FPGA XCV400 device is used in the implementation. Taking into account the number of Block RAMs available the FIFO size was calculated to be 1024x32 for each link. The specifications for this FPGA are given in Appendix E.

3.1.1.1.2.3 FIFO State Machine

The basic structure of the State machine implemented is shown in Fig. 3.6. It basically consists of a state Register, next state logic and an output state logic. The State register is updated every time the state changes. The next state logic decides the next state

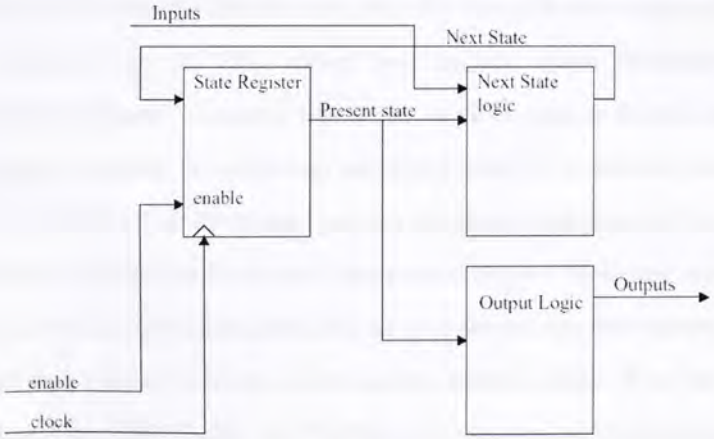


Fig. 3.6 FIFO State Machine Diagram

depending upon the input bits. The output logic asserts the corresponding signals depending upon the state. These state machines are synchronous state machine and are sensitive to a positive clock transition. The next state during n^{th} clock cycle becomes present state during the $n+1^{th}$ clock cycle.

3.1.1.1.2.3 FIFO Readout state machine

FIFO Readout state machines have two-clock cycles latency from input to output. Readout state machine reads out data on every clock cycle only if the state machine has a token. The FIFO Readout State machine has four different readout modes, Normal Mode, Link Masked off, Slip link and read over first FIFO.

Initially the FIFO readout state machine is in idle state. The read out state machine checks for the formatter FIFO that has a token. If a timeout Error or a dataoverflow error is detected then the FIFO readout state machine asserts "Linkhastoken" the "FIFOOutputEnable" is asserted high. It then passes the token to the next Link. If the Formatter "hasword" is asserted high and readout mode0 is '1' then the corresponding word is read out. The "FIFOOutput" enable is also asserted high along with Next address readword and data valid. If the readout state machine detects a "SeeHeader" asserted high along with a see trailer high and the link has word asserted high then it passes the token apart from clearing the timeout counter and data overflow counter. If the state machine detects a high 'HOLDoutput' or a 'NOThasword' high then the Corresponding FIFO is not enabled high. This is clearly illustrated in the state machine state diagram.

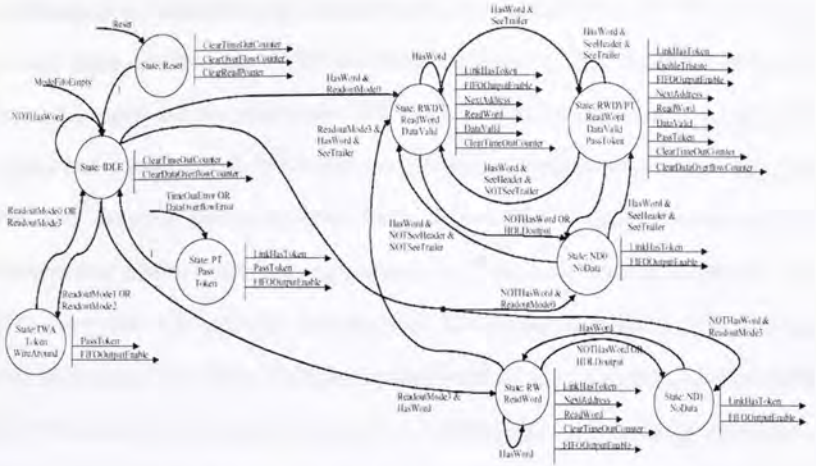


Fig. 3.7 FIFO Readout state machine state diagram

3.1.1.1.2.4 Write in state machine

The Write in State machine checks for the FIFO Full flag and FIFO empty Flag. If the FIFO is not full then the data is written into the FIFO.

3.1.1.1.2.5 FIFO Controller

The Main Function of the FIFO controller is to manage the reads and writes of the input memories. The memory is partitioned 1K words per link. The FIFO controller checks the links within each decoder block on a round-robin basis. A word is written if data is available and the link is not masked. If a link is near full, only a header-trailer is written. The link data for a given event is sent to EFB if the full data from that link is available and the previous link's data has been written. To perform event building two concurrent processes are running within the ROD. The first deals with filling the input memories

(FIFOs) with event data from the corresponding detector module, while the second one extracts event data from the FIFO's and builds up the event. The first task is performed by the formatter, and the latter by the EFB. Thus the FIFO Controller has to manage the writing and reading of four FIFOs and sort data by sequentially transferring a full event from each formatter data to the EFB. There are two FIFO controllers where each one contains four FIFOs, which are implemented as FPGAs with dual port memory. The FIFO controller also has other features, such as skipping data which times out, and controlling trigger throttling. The latter is performed by using a counter for each FIFO, which increments by one each time a trailer is written to the input memory. When all the four counters are greater than zero, a signal is sent to the ROD controller to stop producing level-1 triggers.

3.1.1.2 Operations Controller

The Operations Controller encodes all the strobes from high order address bits and 'formatter_bus_strobe' and encode all returning register transfer acknowledge signals onto 'formatter_bus_ack'.

3.1.1.3 Operation State Controller Register

It is a Register file for config flags, error flags, flag readout, debug data readout and write in control signals.

3.1.1.4 Header Trailer Detector

This module examines the Fifodata_out BUS for headers and trailers and asserts the appropriate flag. Headers will only be found in the high order portion of the word. Trailers will only be found in the high order or low order portion of the word.

3.1.1.5 Next Mode Manager

The next mode manager is a state machine, which latches and holds the information as it comes from the EFB for use in the FIFO State Machine. This module deals exclusively with the ModeBit1 and ModeBit0 mode bits from the EFB. These mode bits are cached up into a small FIFO.

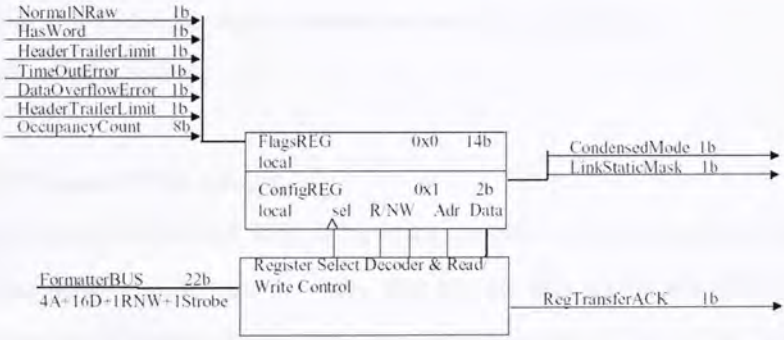


Fig. 3.8 Operation State Control Register

16x2 for readout and control of the FIFO State Machine. The other mode bits ModeBit3 and ModeBit2 are managed by/in the RegisterIOfile. Mode bits enumerate to the following when viewed as a vector.

0000 - EFB in control with (ModeBit1 and 0 used), readout 1st event in FIFO, normal data flow from decoder to EFB

0001 - EFB in control with (ModeBit1 and 0 used), this link masked off, masks disables write in/readout counters and occupancy counter

0010 - EFB in control with (ModeBit1 and 0 used), this link skipped, skipped for re-synchronization

0011 - EFB in control with (ModeBit1 and 0 used), readout 2nd event in FIFO, read over 1st event and dump on the floor

0100 - EFB control disabled (ModeBit3 and 2 used), disable/drop decoder input to FIFO, switch/route Formatter Bus data to FIFO Buffer and write data directly into the FIFO

1000 - EFB control disabled (ModeBit3 and 2 used), disable output of data to EFB, switch/route FIFO Buffer data to Formatter Bus and readout data directly

1100 - spare/undefined.

3.1.2 Formatter FPGA B-Layer

In the case of B-layer ROD there are 7 modules per ROD at 160 Mbits/module, or 28 links at 40Mbits/link. Consistent with the ROD implementation style these 28 Links are divided into four banks of eight links. Four links are masked off by the DSP. Each Formatter FPGA is input with 8 modules at 40Mb/module bandwidth equivalent to 2 modules at 160 Mbits/Module and each link was provided with a FIFO 2048 words deep and 32 bits wide.

3.1.2.1 Link Formatter and Link output FIFO B-Layer

The Link Formatter and link output FIFO contains two Link Formatters and two FIFOs. The mapping is similar to the Layer 1 except that there are only two Link Formatters and two FIFOs and not four each.

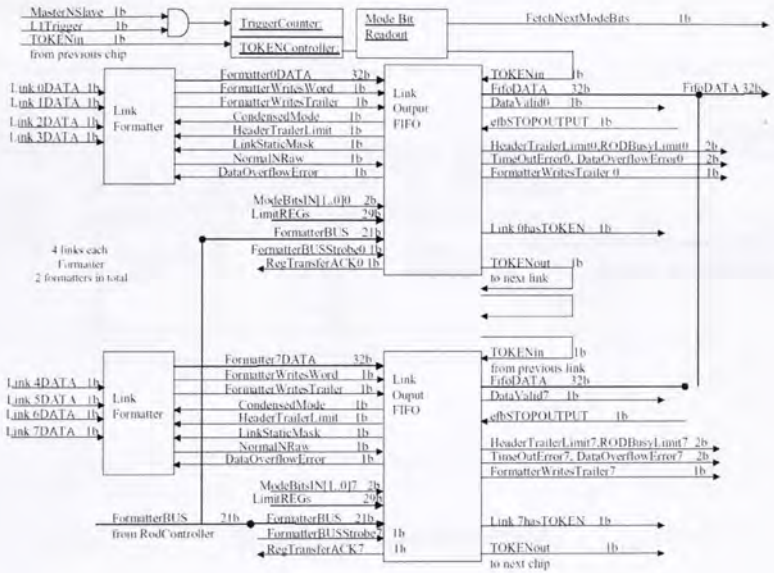


Fig. 3.9 Link Formatter and Link output FIFO (B-layer)

3.1.2.1.1 Link Formatter B-layer

The Link formatter for B-Layer detectors retains the same decoder design but the only difference is that the decoder is designed to decode data from four links instead of two. However the width of the shift register is increased in order to give the detector logic

more time to detect the events. A shift register 32 bits wide is used. Since the data is input by four links, four bits are input into the shift register for every clock cycle. The decoder logic checks for the bits only after a one clock cycle after the bits are stored in the shift register. Therefore by the time the output write signals are asserted the bits will be shifted out if the size of the shift register is small. Therefore the size of the shift register was increased.

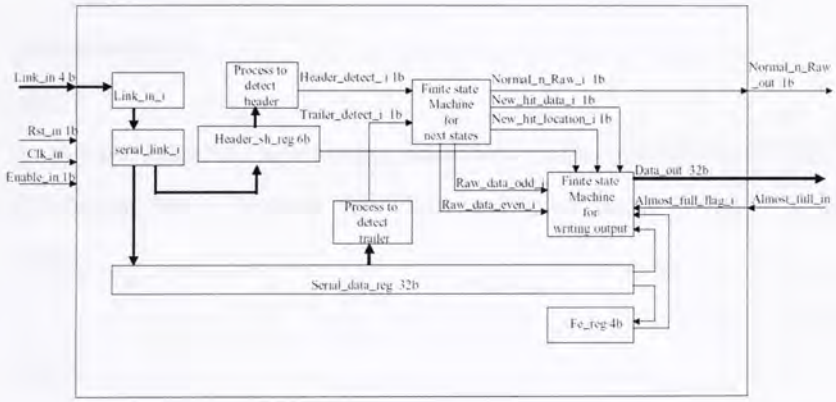


Fig. 3.10 Link Formatter (B-layer)

3.2 Problems faced

It was originally intended to pack the Event data into 16 bit words. This sixteen bits of data would also include 3 bits for event identification. However this could not be implemented because some of the event data exceeded 16 bits. The event ID bits (L1ID and BCID) had to be packed together. Each of them was of 8 bits. This along with the three identification bits put the total at 19 bits. Similarly the hit data could not be packed

into 16 bits. Therefore it was decided to increase the width of the output words to 32 bits. This however wasted a lot of bits while writing MCC Flag and FE Flag, which were only 8 bits each. However this affected the size of the FIFOs. Since the widths of the words were increased the FIFO depth had to be decreased in order to offset the increase. However, this would make the FIFOs fill quickly since the EFB reads data slower than the formatter.

3.3 Implementation

Layer 1

The implementation of the layer 1 design consumed 65 % of the logic blocks in a VirtexE XCV400 part. 80% of the block RAMs were used by the design consistent with the mapping.

B-layer

The implementation of the B-layer design consumed 55 % of the logic blocks in a VirtexE XCV400 part. 80% of the block RAMs were used by the design consistent with the mapping.

3.4 Simulation

A simulation of waveforms was performed by compiling the back-annotated VHDL code. The simulation was run for 5000 ns. The behavior of the back-annotated code was identical to the original simulation indicating that the code was synthesized and implemented without any errors. The waveforms are attached in Appendix A.

CHAPTER 4

Event Fragment Builder and Router

This chapter will deal with the Event Fragment Builder and the Router. The Formatter outputs the data and the event fragment data are passed on to the next stage, which is the most complex one of all the design blocks, the Event Fragment Builder (EFB). The EFB has the function of gathering and checking the data. It will use time stamps and event IDs to properly associate event data from different streams and to label the event records for subsequent assembly. The event labels are provided by the event FIFO.

In addition the EFB must be configurable to accept data from any combination of input links, since the combination of input data links will depend on where the data originated. The EFB transmits the data to a derandomizing buffer where calibration data and register data are separated, from event data, in the input data stream. It will also buffer the data for access by the ROD controller via the EFB. The data is output from the EFB to the Router where events are routed and trapped and sent to the DSPs for data analysis.

4.1 Event Fragment Builder FPGA

The multiple event fragment builder (EFB) engines are, in principle, similar in operation to the decoders. Multiple EFB engines are required in order to maintain acceptable bandwidth given fixed and statistical fluctuations in link to link switching latency and error/timeout conditions. EFB engines accept data being pushed into them from the

Formatter FIFOs, process it and store it into EFB FIFO buffers for later shipment to the router. Each EFB operates in an event manner on event data from the respective link's decoder FIFO buffers, checking the information for consistency with the event header information, marking and counting any discrepancies or errors and storing this information for the formation of the overall event header and trailer. The event header and trailer information are also formatted and issued from the EFB to make an overall complete event packet. The implementation model of the Event Fragment Builder engine is illustrated in Fig. 4.1. Two of these engines in parallel constitute a dual event fragment builder, one EFB engine for each bank of formatters. In the case of Pixel ROD there are two banks and each bank has two formatters. Each Event Fragment Builder engine is responsible for decoding, error checking the data from the respective Formatter FIFOs.

The EFB manages event fragment building, output and error counting. This module contains a single FPGA, which contains enough resources to implement 3 copies of data formatting, error checking, trigger consistency checking, link FIFO mode selection, re-syncing, and derandomizing buffering of incoming decoded data. The primary functions of the EFB are to receive event data from the FIFOs, receive event ID, build events and write to output memory as well as read from output memory. For each detector there is also an error counter, which allows counting of errors in header, trailer, L1ID, BCID as well as counting of missing synchronization bit or FE flagged.

When the EFB is considered done and the required tasks are performed, the formatted event data are transferred over to an output FIFO, which is needed to derandomize data

from EFB data streams. From the two EFB FIFOs the event data is available for the Router to send it down the S-link. EFB performs event BC/L1 ID checking, error detection and counting, data formatting and counting. It examines header and trailer to perform event ID checking and error counting. Errors are identified on a link- by- link basis. Data is formatted by examining header and trailer of each link. The Output of EFB to can be stopped if "Stop Output" is issued. However, this data can still flow through the EFB and fill the output memory.

EFB works in parallel on two data streams, in order to minimize the latency. When an event flagged with level-1 number *n* is ready, which the event builder can tell by looking for the event trailer, the event building process takes place. Since event word quantities may vary from decoders associated with each Event Fragment Builder, separate event header FIFOs are required. This allows each gatherer to process the different event incoming decoder FIFO information being pushed into it at that time and allows orderly readout and event packet formatting. If no header information is yet present, decoder FIFO buffer output must be halted until a header arrives – this necessarily does not inhibit the readout of the last event. If an event has been read through/processed and written into the EFB buffer, the next event may be serviced, and so the 'SendNextEvent' header signal is asserted to the Event Header FIFO which sends up the next header ID and throttle flag information. The EFB Engine knows when it has serviced all links for a given event by the readout of the trailer from the last link and the roll over of the current link number being serviced. Link Mode Register file holds in registers each link's static

mask, dynamic mask, mode bits and throttle count, and may be operated on by the ROD Controller over the EFB Engine BUS or TIM ID Throttle Flag header information bus.

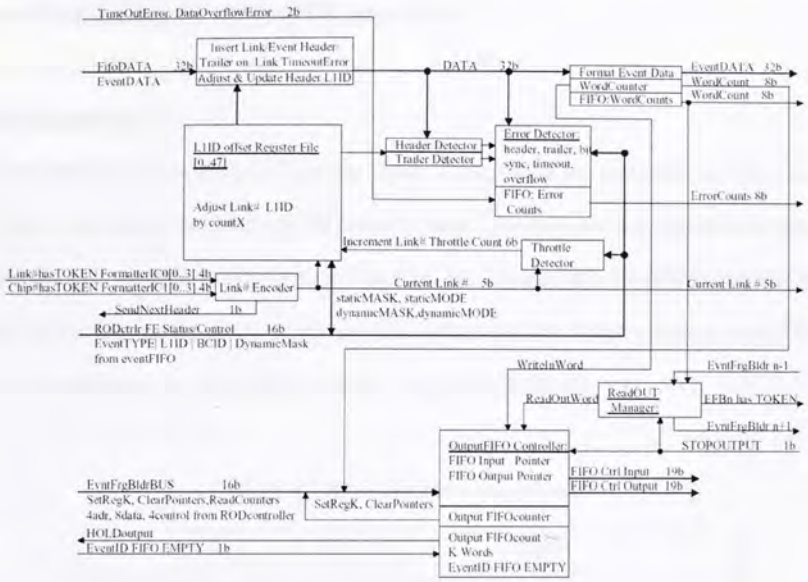


Fig. 4.1 Block Diagram of Event Fragment Builder

Link number encoder

This module encodes the four bits from each decoder an EFB Engine services into five bits. It inserts Link/Event Header Trailer on dynamic skipping/throttling. It also inserts a header Trailer into the data stream when a dynamic skipping or throttling condition has been met.

Throttle detector

This module detects the presence of throttled link flags in the event header and increments the appropriate link throttle counts. Adjust Update Header, updates the L1ID and Throttle Count by a throttled link event L1ID.

Error Detector

This module detects errors of specific types, counts them for inclusion into the event trailer, and buffers them into the FIFO error counts. The error detector decodes the errors coming into the EFB via the data_in bus. The data coming into the EFB is checked for errors and the corresponding errors are then written into the 'Error_summry_word'. The exact mapping of the 'Err_Summry_Word' is given in Table 4.1.

Table 4.1 Error summary word format

Bit	Error
Error_summry_word (0)	Header Bit Error
Error_summry_word (1)	Trailer Bit Error
Error_summry_word (2)	Flagged Error
Error_summry_word (3)	Sync Error
Error_summry_word (4)	Condensed Mode Hit Pattern Error
Error_summry_word (5)	L1_ID Error
Error_summry_word (6)	BC_ID Error
Error_summry_word (7)	Timeout Error
Error_summry_word (8)	Almost Full Error, data Truncated
Error_summry_word (9)	Data overflow

Format Event Data and word counter

The event data are repacked into new words. While repacking the data, the Event data decoder checks for valid words from the data_in. To make best possible use of bandwidth, Format event data drops 1/2word, no error trailers, and overwrites non-error trailers bottom of word with the next link header. It counts the number of words processed by this EFB Engine from this event and buffer the count into the 'FIFO Word Counts'.

Event data Decoder

This module reads the event FIFO data from the external FIFO and writes the 8 bits BCID and 8 bit L1ID data into the event ID FIFOs of each gatherer. The full L1, BCID and trigger type information are sent to the logic that generates the event fragment. A full

Table 4. 2 Output Data Format

Word 0 and Word 1	Trigger Type [7:0], L1 ID [15:0]
Word 2	BC ID [11:0], Dump Data Flag, Unused, TIM Event Type [1:0]
Word 3	ROD Event Type[15:0]
Word 4	Dynamic Mask for Links [7:0]
Word 5	Dynamic Mask for Links [15:8]
Word 6	Dynamic Mask for Links [23:16]
Word 7	Dynamic Mask for Links [31:24]

event of data is read out over 8 words from the external FIFO since there are many bits of data. The exact format of the event words is given in the table below.

Readout Manager

This module uses the information from 'FIFO Word Counts' to read out the appropriate number of words from the EFB Engine FIFO for event packet construction. It stops output of data on assertion of the 'STOPOUTPUT' signal from the data director.

Output FIFO Controller and Counter

This module manages the Write in Address and Readout Address of the EFB Engine FIFO and holds the output pointer when 'STOPOUTPUT' signal is asserted from the data director. When the EFB Engine FIFO occupancy reaches a configurable threshold, this counter puts backpressure on the decoder FIFO controllers to stop sending data . However this does not affect the read out of data.

4.2 Router FPGA

The Router contains multiple independent event data collectors, which route/trap data based on their configuration (Event Type, L1ID, Error Type) and the embedded routing information in the data, to slave DSP processing engines for analysis. In addition, the router is responsible for final updates to the event information and dumping data of specified types when so configured. The above describes the primary, data collection function of the ROD. In addition to this capability, the ROD contains a plethora of other modes and capabilities embedded for configuring, calibrating, and debugging itself, the BOC card, FE modules and slink.

The circuit is contained in a single FPGA, which contains resources for snooping the passing event data and trapping events and data specified by programmable trapping criteria and transmitting those events/data to computational engines for analysis on the ROD as well as the Crate processor. These computational engines are presently fast floating point DSPs. The decoder FIFO consists of a single FPGA, which contains enough resources to implement 12 (decoder, FIFO Buffer, FIFO Controller) engines to service 12 links.

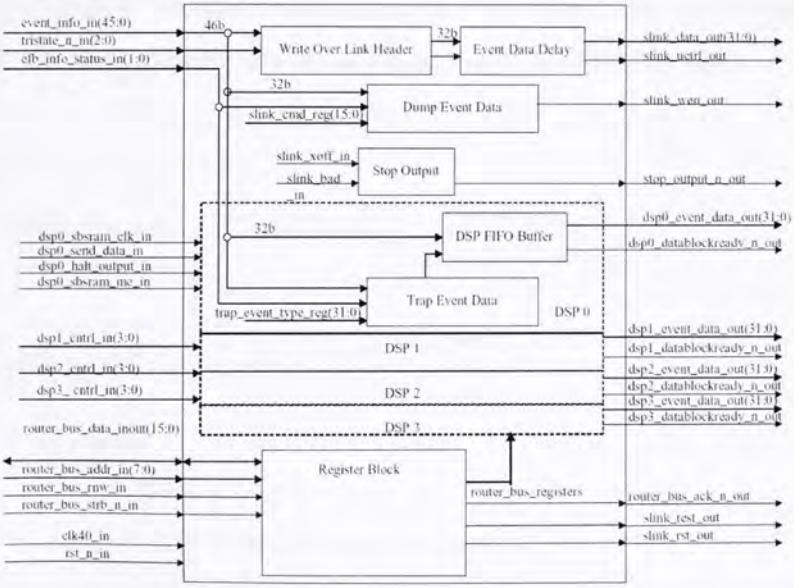


Fig. 4.2 Block Diagram of Router

Write over Link Header

The write over link header module looks at the event_data from the EFB, identifies the link headers, and writes the link and error information into specific bits of the link

header. The data output is input into a large array of pipeline registers that delays the S-Link data until the event header is read into the Router and analyzed to determine if the Event Fragment should be transmitted to the S-Link bus or purged. It is also used in the same way for the DSP data.

Dump Event Data

The dump data event block reads the event header and determines if the Event Fragment should be transmitted to the S-Link bus or dumped on the floor. This block only controls "slink_wen_n_out" and does not actually affect the data in any way. This block searches the event Headers for specific event types and dumps data as required. This module

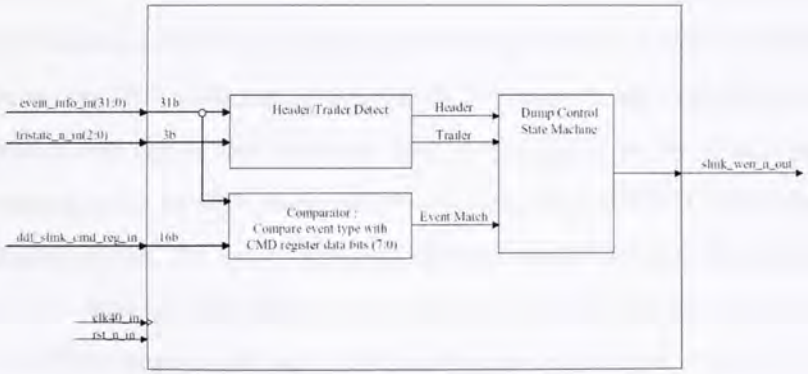


Fig. 4.3 Dump event data block

basically consists of a state machine, which determines the state of the S-link, Mode of the S-link. This state machine matches the bits of the ATLAS specific events with that of the L1 trigger events. If the data match then the state machines goes in to 'dump_event' or else it keeps the event and slips to 'keep_event'. Similarly the State machine matches

the TIM specific event with the L1 trigger type header word. If the data matches then the data is dumped else the data is kept and the FSM slips into 'keep_data'. Similarly the L1TT header word is matched with ROD specific event type. If the data does not match then the event is kept else it is dumped. This process is a must in order to identify the type of data. The FSM is depicted in Fig.4.3.

Trap Event Data

The trap data event block compares the trap event type registers of the router to the incoming data to determine if an event should be trapped in the Data FIFOs. The DSP begins the event collection process by setting the 'trap_enable' bit true. When the router sees this, starting with the next Beginning Of Fragment (BOF), the router will begin streaming data to the FIFO in parallel with any other destinations, e.g. S-Link or FIFOs for the other DSPs, which may already be active. The router can also be configured to select on event type for each destination. After it detects trap_enable true, it will begin streaming data to the FIFO starting with the next event, which matches a configurable selection criterion. The router will continue writing data to the FIFO until 'trap_enable' is cleared. When the FIFO reaches a certain threshold, say d = D/2, where D is the depth of the FIFO, the router will issue a dsp0_datablockready_n to ext_int4. This tells the DSP that it should burst in data from the FIFO to its internal data memory. This implies that the frame size of the direct memory Access (DMA) should be configured to 'd' 32 bit words. The DMA can go quite a bit faster than the 40 MHz maximum rate at which the FIFO can be filled, so when the DMA is done, the FIFO occupancy should be less than d. When it fills to d again, dsp0_datablockready_n will be issued again. This will continue

up to the point where N blocks have been written to the FIFO. On the last event, dsp0_uctrl_n will be issued. The last data block will usually have less than d words, but this does not harm anything. The DSP will just read trailing zeroes.

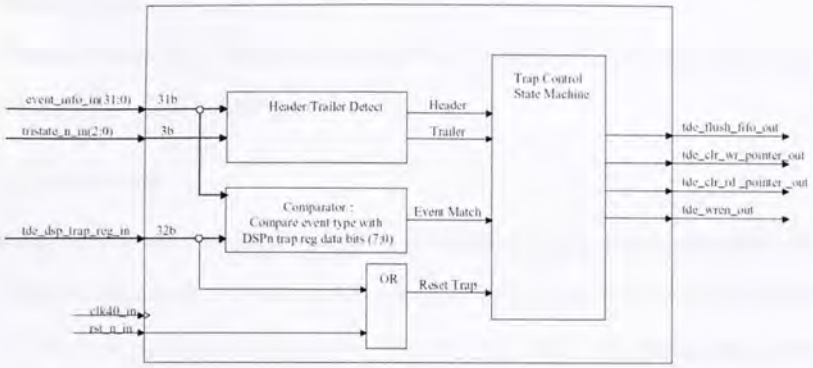


Fig. 4.4 Trap Event Data Block

The DSP processes the N events, e.g. histogram them, dump them directly to SDRAM, alert the master DSP that events are ready to transmit to the host, etc. When finished, the DSP can write again to an address in its CE1 memory space to collect N more events. Clock between the DSP and the router which is used to strobe the data runs at 80 MHz, twice as fast as the maximum rate that data will be written to the FIFO. During normal running the mean rate of filling the FIFO will be 20 MHz averaged over long time periods relative to the FIFO size. It may be possible for the DSP to read from the FIFO and write directly to SDRAM faster than the router can fill the FIFO. If this is the case, this makes several Mbytes of memory available for initial event storage before processing data (instead of 64 Kbytes of internal data memory) and so N can be quite large. It would be interesting to understand if the DSP can DMA from the router FIFO to the SDRAM

faster than the router fills the FIFO. This protocol will be used by DSPs making runtime occupancy histograms or collecting events for transmission to the VME host.

Register Block

This is a register file for the Router Register Bus. It examines the comments specifically to discover the exact register definition.

4.2.1 Error events

When the DSP sets trap_enable true, the router begins dumping events to the FIFO. The 'dsp0_datablockready_n' is issued when the d words are available in the FIFO. The DSP collects those words and just scans them for error flags. There is an error summary at the end of each event. If no errors, which require action, are detected, the next d words will just overwrite those just scanned. If errors, which require attention, are found, the next d words (if the DSP needs more data to diagnose the error) will be appended to the current set. Once the DSP has enough events with errors that it makes sense to run a diagnostic routine, it will ignore further FIFO entries while it diagnoses the error. After the error has been processed and whatever action has been performed that was deemed appropriate, e.g. a correction or an alarm, the DSP can begin scanning events for errors again. In this protocol, the router is just filling the FIFO nonstop. When the DSP stops reading in order to process errors, the FIFO in the router will fill. In this case, the DSP writing to the CE1 address space will cause the router to flush the FIFO and resume writing with the next BOF. The S-Link and the monitoring/global-error-scaling DSP have a common data format, which consists of a 9-word event header, a 5-word event trailer with the link data sandwiched in between.

Table 4.3 Event fragment header

H0	0xB0F00000 + UCTL	Beginning of fragment
H1	OxEEEEEEEE	Start of header
H2	0x9	Header
H3	Tbd	Format version number
H4	Tbd	Source identifier
H5	L1ID	Level 1 ID
H6	BCID	Bunch crossing id
H7	L1TT	Level 1 trigger type
H8	DET	Detector event type

Table 4.4 Event Fragment Trailer

T0	Error Count	Status 1: count of words with errors
T1	Error Flags	Status 2: bitted error type occurrence word
T2	0x2	number of status words
T3	nData	number of data words
T4	0x1	Status block position: 0/1 = before/after data
T5	0xE0F00000 + UCTL	End of fragment

4.2.2 S-Link Data Format

The Router FPGA directs the event data to the S-Link and to the Slave DSPs. The S-Link and the monitoring/global-error-scaling DSP have a common data format, which consists of a 9-word event header, a 5-word event trailer with the link data sandwiched in between. The 'out of router' column is valid for the S-Link and the monitoring DSP. The labels in this case, e.g. d1, d2, do not enumerate data words as in the case of h0, ... and t0, ..., rather they label types of data words. The link data is organized by link, and they

arrive in a fixed order from event to event. Each link block begins with a link header. The link trailer is only present if there are error bits set in it; these are the 'v', 't', and the 'h' in d2 and will be explained below. The link header is suppressed in the event if that it would be the only word in the event for that link and there are no error flags.

Data arrives at the router from the event fragment builder on a 43-bit (plus a couple of spares) bus. The first 32 bits are two of the 16 bit fields from the left column below 'into router' above. Bits 32-42 are the field in the right column below 'into router'. This field does not affect the other words (don't care) except for link headers. (It is also don't care for event fragment header and event fragment trailer words.) Because of this, if the first 16 bits are the last field for link n-1 and the next 16 bits are the first field for link n, the last 11 bits correspond to link n.

- Error bits added by the ROD are in lower case letters.
- Bits extracted directly from the incoming data are in upper case.
- Bits that are fixed are written explicitly as 0 or 1.
- Don't care bits are indicated by 'x'.
- Hit words always begin with 1.
    - Each link formatter instantiated in the formatter FPGA is in condensed or full mode. This determines how it formats the incoming link data. The mode will only change in response to an instruction from the crate processor. The router also knows what mode each of the formatters is in via up to three 32-bit

registers (the maximum number of links is 32) which are written at the same time the formatters are configured.

- During normal running the formatter operates in condensed mode. In this case, a 3-bit data field, which accompanies every hit coming from the modules, is dropped.
- Any link can be put in full mode, in which case the 3 data bits that accompany each hit are kept. These are the DDD fields in d5-d7. In this case, the first hit in any FEID (hit with lowest strip number) is indicated by b12=0.

- The 4 other types of word begin with '0' and are distinguished by b1 and b2.
- Link headers begin with 001. A single bit error in the event preamble coming from the detector module is indicated by p=1 at b3. The L1ID and BCID from the detector module are encoded as LLLLLLLLBBBBBBBB in b4-b19. The link number (up to 32) is encoded as MMMMMMM in the high order 11-bit field.
  - w=1 in case of a timeout
  - l=1 indicates a L1ID error (LLLLLLLL from detector did not match the lower order bits of the L1ID sent from the T/DAQ system. The latter is sent back to T/DAQ via the S-Link in h5.
  - b=1 indicates a BCID error (BBBBBBBB from the detector did not match the lower order bits of the BCID sent from the T/DAQ system. The latter is sent back to T/DAQ via the S-Link in h6.
- Link trailer words begin with 010. Link trailer words are inserted when the formatter FPGA detects a link trailer pattern coming up the link. The event fragment builder FPGA, after counting event errors and filling in the error

summary words, t0 and t2, removes trailer words (which contain no information) before they reach the router. The exception is the case when one of the error bits is set.

- t = trailer with error
- h = header only mode. This indicates that the derandomizing buffer at the output of the formatter is full and so data is being thrown away; only headers and trailers are kept.
- v = data overflow error. This indicates that data arrived on the link for a given event, but just kept on coming.
- Front-end chips can send 3 bit error codes up the data link to the ROD. When one of these error codes arrives it is formatted in a word beginning with 000. The error code is EEE and the number of the chip, which had the error, is, again, CCCC, which is the address of the front-end chip. The error module is detected from the link number.

4.2.3 Format for Error Diagnostic DSP

One of the slave DSPs is concerned with diagnosing errors. This DSP will need more than just the error summaries. It will need all error flags as well as the module number and the L1ID and BCID fields from the detectors.

- The condensed mode bits will not be used in the router (since they come on a separate link), but they will still be inserted into the Event Header in words (in words two and four of the event header). The information will be loaded directly into the Backend

DSP. This will allow the router to compact the Event Header by 3 words. This can be accomplished easily using a tapped pipeline for the DSP data.

- Because of the 00 marking no data allows the DSP to keep track of module numbers (module number = number of link header words read + number of intervening 00 codes), the MMMMMMM can also be dropped.

Table 4.6 Format for Error Diagnostic DSP

00	No data	The router did not get a header for this link, there were no hits and no errors
01	Good data on the link	No errors
10	BCID or L1ID error	This is included so that the DSP does not need to scan all of the data to find the error. This will point to a link with a problem. The DSP can then go to that link and determine whether the error was a BCID or L1ID and exactly what the error was.
11	Timeout error	Three of these words overwrite the event fragment trailer words t2- t4. The other three are additional words, which would be inserted before the E0F00000 word in the event fragment trailer.

4.3 Problems faced

The Event Fragment Builder was originally designed with one event fragment builder engine. However it was observed that the read out of the EFB was too slow when compared to the Readout of the FIFO. Therefore it was decided to divide the FIFOs into two banks and use one dedicated EFB engine for each bank. This change had to be made keeping in mind the 32 bit words being read out from the FIFO.

4.4 Simulations and Synthesis

The EFB and the Router blocks were implemented on a Xilinx VirtexE XCV 400 device. The codes were synthesized using the Synopsis FPGA Compiler and then placed and routed using Xilinx implementation tools. The back annotated codes were then compiled and simulated to check whether the output matched the expected output. The outputs matched and are to be implemented on board. The simulation of the back annotated VHDL code, which actually illustrates the behavior of the FPGA to the original code was run for 5000 ns and yielded proper results. The Simulation waveforms are attached in Appendix A.

CHAPTER 5

ROD Resources Interface

5.1 ROD controller

The ROD Controller consists of a computing engine, which buffers control and data signals to and from the various resources on the ROD. It is responsible for configuring each of the ROD blocks into the appropriate modes of operation, issuing the appropriate commands and data to each block to implement the tasks of each mode of operation to receive fast commands and event headers, maintaining Front End (FE) Module event occupancy counts, throttling FE Module triggers as appropriate, formatting slow and fast commands to the FE Modules, formatting and buffering event headers with throttle flags for shipment to the event EFBs, communicating with the crate processor via the VME bus interface and list message buffer configuring the BOC Card Buffers and moderates the flow of instructions and information to and from the backend slave DSPs, asserting the RODBusy signal when appropriate and necessary.

Bus buffers are required to implement connectivity and functionality. The rod resource busses are buffered to and from the main ROD Controller computing engine and exchange data and control messages. Each bus buffer has specific requirements to meet bus widths, data rates and access different memory types ranging from external FIFO memories, to registers and to FPGA memories. This is the portal for communication of the crate processor to on ROD resources. The Front End Trigger control block manages trigger throttle controls, as well as front end module fast and slow commands. Each link has its own occupancy counter, mask and throttle flag, and is paired with its next

neighbor to implement full module throttle control. It is important in the assembly or installation phase, to be sure that the two half module fibers get attached to the two ROD input links that are associated with that module's control link fiber. If this does not happen, throttle control will be erroneous and invalid. ROD Busy control is also handled in the ROD Controller block, and asserts, based on configuration, the busy signal if any one of the input links have reached the Header TrailerLimit or the RodBusyLimit.

Command Message List Processor contains both an input FIFO for instructions and data flowing into the ROD and an output FIFO for processed/trapped data on its way out to the crate processor. Given the specific "Register oriented" implementation of this approach, information is written directly to the command FIFO irrespective of address. That is, any address may be asserted and write information will be directed to the input FIFO while read information will be fetched from the output FIFO. In addition, this scheme precludes the RODs from direct communication with each other. All communication must be moderated by the crate processor.

The ROD Controller is illustrated in Fig. 5.1. It consists of a ROD Resources Interface FPGA, FPGA Program Manager, DSP Module, Trigger Interface Module interface and VME Bus Interface. However this chapter will focus only on the ROD Resources Interface FPGA (RRIF). The RRIF consists of a real-time data processing engine and a data processing engine, which works on quasi real-time data. The real time path is responsible for generating trigger pulses, command pulses and command streams. The

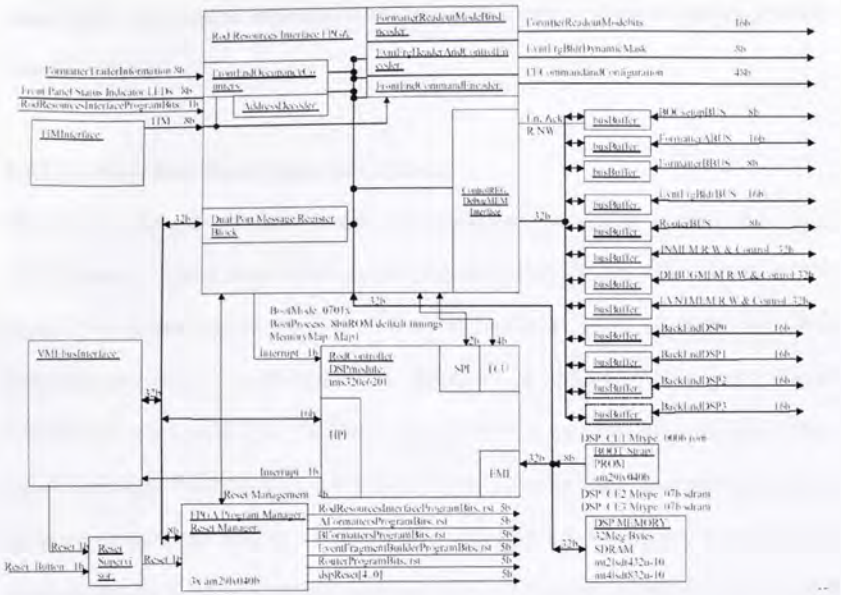


Fig. 5.1 ROD Controller

quasi real time block is responsible for generating mask and mode bits. The front end command processor works on the real-time data whereas the EFB header dynamic mask encoder and the formatter mode bits encoder work on the quasi real-time data.

5.1.1 Real time Data Path

5.1.1.1 Front End Command Processor

The Front end command Processor consists of front end short command generator, front end medium command generator together with mask logic and registers to form the Front End Trigger Command Module. The front end command processor is responsible for generating fast and long commands and subsequently masking them according to the

mask inputs. This module takes care of the links to be masked off or on depending on the command inputs.

5.1.1.1.1 Front End Short Command Generator

The Short command generator issues fast commands in response to Level 1 Accept (L1A), Bunch Counter Reset (BCR), Event Counter Reset (ECR), and Calibrate (CAL) inputs. The Pulses coming into the module (L1A, BCR, ECR, Calibrate) are 25ns synchronized exactly to the 40 MHz clock. This block generates a "110" on an L1 accept, a "1010010" on a BCR and "1010100" on an ECR to the front end modules. These signals are output from the Trigger Interface Module. The command generator takes three clock cycles from the moment L1Accept arrives to send 110, during which none of the other signals (BCR, ECR, Calibrate) shall be active. If the other signals are active then it does not send a command. Similarly a BCR takes seven clock cycles to send 1010010, during which time none of the other signals (L1A, ECR, Calibrate) shall be active. Also the ECR takes seven clock cycles to send 1010100, during which time none of the other signals (L1A, BCR, Calibrate) shall be active. In addition, to that no signal shall be active during the calibration sequence, 26 + 1 + count in cal_trig_latency_in clock cycles. Also no signal shall be active for 10 clock cycles during which L1A, BCR, ECR sequences are active. These rules have been laid down in order to be consistent with the TIM signals. If any violation of these protocols is experienced it indicates an error.

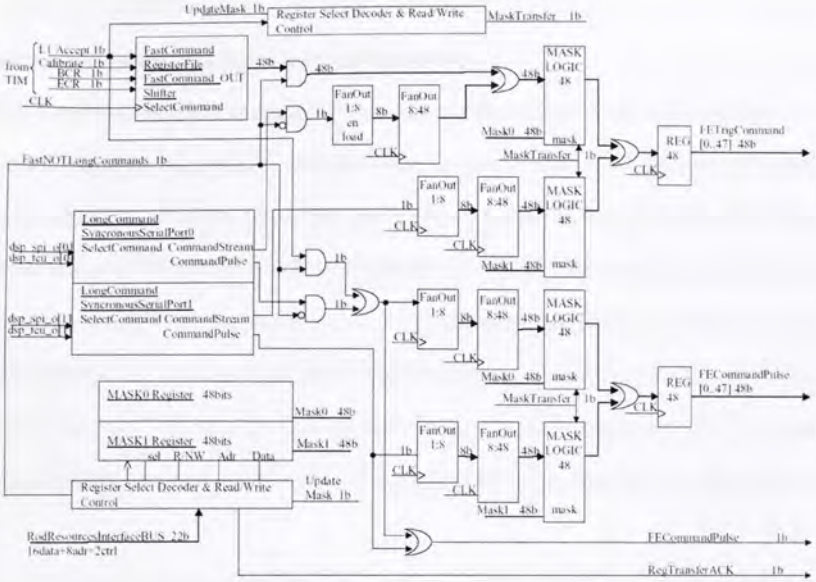


Fig. 5.2 Front end command processor

Fast_not_long command shall be high for fast commands to be written. New masks are loaded into the Mask register only after L1A+4BC clock cycles. Fast Command Register sends the fast command words L1A, BCR, ECR and Calibrate. Forty-eight copies of each command word are made and in each of the two output registers. Each copy corresponds to one input link. Since this design will also be used for the SCT ROD forty-eight copies were. However, only twenty-six of these links will be used for the pixels and others will not be considered.

5.1.1.1.2 Front End Medium Command Generator

The Front End Medium Command Generator implements the front end calibration bit stream. Some pre-determined cal-bit stream is placed into the RRIF cal command register. On a cal strobe, these bits get copied into the command shift register for transmission to the front end modules. Since the relaxation of the cal_pulse to cal output bitstream latency from three clocks to "a more reasonable number", an alternative lower gate utilization scheme is implemented here. In addition, the number of clocks from the start of the cal-bitstream to the l1_accept bit stream is configurable from 0-255. Care has been taken not to set the number below the length of the cal stream, as they will collide .

5.1.1.1.3 Long Command Register

Long command registers send the long command words, command stream and command pulse. The command stream input and command pulse output should not have the same sequence. When the command stream sequence dsp_spi(0) is 110 the corresponding command pulse input dsp_tcu(1) shall not be the same always. Dsp_spi(1:0) is mapped to command_stream(1:0), Dsp_tcu(1:0) is mapped to command_ pulse (1:0). Forty-eight copies of Command stream and 48 copies of command pulse are made using the fanout network.

5.1.1.1.4 Mask Logic

The Mask registers contain the Mask0 and Mask1 bits. Each Mask register is 48 bits wide. Mask bits are loaded into the Mask register by the register read/write control block.

Mask0 and Mask1 registers contain values that are mutually exclusive. In other words, bit wise addition of Mask0 and Mask1 shall never be 2 for any bit. When all Mask1 bits are set low and Mask0 bits are set high and fast_not_long_commands is low, then command_stream(0) and command pulse(0) are output. When all Mask1 bits are set low and Mask0 bits are set high and fast_not_long_commands is high, then long command is output. When some Mask1 bits are set high and corresponding Mask0 bits are set low and

Table 5.1: Truth Table of the Mask operation

Fast not long command	Mask0	Mask1	Reg0	Reg1
1	1	0	Fast_command	Fast_command
0	1	0	Command_stream(0)	Command_pulse(0)
0	0	1	Command_stream(1)	Command_pulse(1)
1	0	1	Output masked off	Output masked off

fast_not_long_commands is low, then command_stream(1) and command pulse(1) are output in the corresponding bits. If any Mask1 bit is '0' and corresponding Mask0 bit is '0', then the output is turned off. If any Mask1 bit is '1' and corresponding Mask0 bit is '1', then the output is turned off.

5.1.1.1.5 Front End Trigger Counter

This Module tracks the number of triggers in the front end modules. Each Front end trigger has two occupancy. Each of them keep track of the trigger from one link.

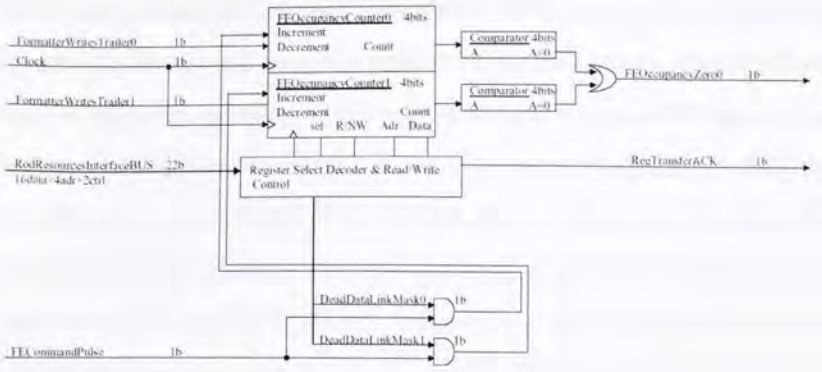


Fig. 5.3 Front end trigger counter

5.1.2 Quasi Real Time Data Path

5.1.2.1 Front End Command Pulse Formatter Accumulator

This module accumulates the L1trigger pulses and fe_cmd_mask_updated pulses from the front end command processor and issues controller interrupts and formatter bank A and B L1 trigger pulses after the specified delay. That is, it receives and counts the L1triggers as well as accumulates them into a FIFO along with the front_end_command_processor pulses. The FIFOed pulses are used to select the normal controller trigger interrupt or the alternate "mode mask has been updated" interrupt. The mode/mask interrupt is pulsed only once whenever a mode mask is updated. Since triggers can arrive at a rate of 5 to 25 40MHz clocks, and the controller cannot handle interrupts of that rapidity, it is necessary to buffer and issue these to the controller when its interrupt handler can deal with them. This is also the case with the EFB header dynamic mask encoder. In the best case when the RRIF internal dynamic_mask_fifo is not full, it takes: (13words * 2wait states/word) + 4 miscellaneous setup steps = 30

clocks before another dynamic mask can be written. If there are additional waits because the RRIF internal 'dynamic_mask_fifo' is full or not large enough dynamic masks will be dropped and an error condition will occur. In the best case: when the EFB internal write in FIFO is not full, and is not waiting, it will take: (16 words * 4wait states per word) + 4 miscellaneaous clocks = 68clocks to completely write in a dynamic mask to the EFB from the RRIF. The controller will set this delay(6 downto 0) from 32minimum to 127 maximum 25ns clk periods (0-3.18usec). Care must be taken not to set the delay to a value smaller than the runtime of the context switching and IRQ service routine in the controller OR the dynamic_mask writein time. Upon reset, the counter will be preloaded with its maximum delay.

5.1.2.2 EFB Header Dynamic Mask Encoder

EFB Header Dynamic Mask Encoder is responsible for transforming the incoming signals from the Trigger Signal Decoder (Event ID and Trigger type signals) and the trigger inputs to the Master DSPs from the Event Fragment Builder (Apply default Mask and Apply Corrective Mask) into 16 bit words.

All the input Signals coming into the module are 25ns synchronized exactly to the 40MHz clock. The Timing, Trigger and Control Signals SerialID and SerialTT bits are generated by the Programmable Logic in the TIM and are connected to the Trigger Signal Decoder Module in the ROD Resources Interface Block. The Trigger Signal Decoder de-serializes the Input Bit stream and Outputs the L1ID, BCID, Ttype bits. The signal

assignments are illustrated in Table 5.1. The control input from the control register controls the operation of the write-in and readout state machines.

- The control_in controls all the write-in and readout state machines.
- Control(0) when High resets the state Machines.
- Control(1) when Low enables the state Machines .
- The Control(2) and Control(3) are used to load and reset the counters.

The strobe signals are new_l1id_valid_in, new_bcid_valid_in, new_ttype_valid_in .The signals new_l1id_valid_in, new_bcid_valid_in should be high to write the Event ID bits into the Event_ID FIFO. Module. The signal new_ttype_valid_in should be high to write the Ttype bits into the Ttype FIFO. The Trigger Signal Decoder generates this signal.

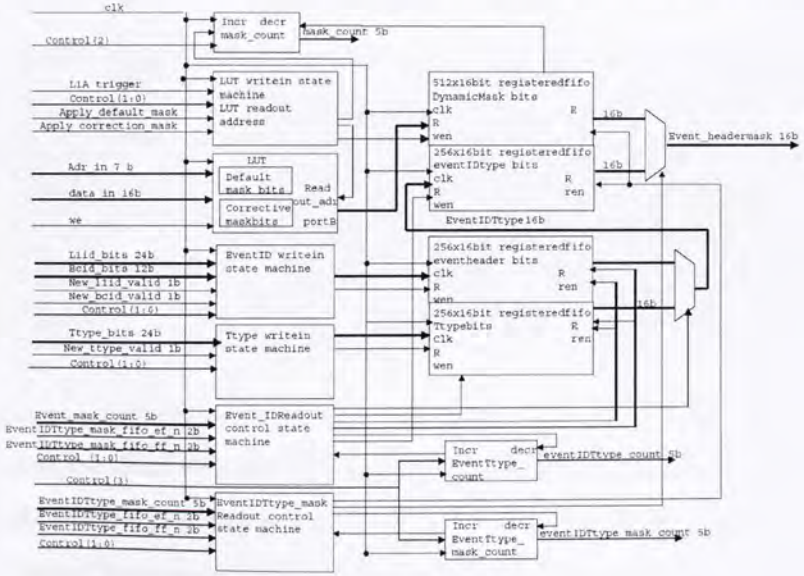


Fig. 5.4 EFB Header Dynamic Mask Encoder

The two trigger interrupt signals are the apply default mask and apply corrective mask. The Apply Default Mask is an interrupt to the Master DSP to signify that a default trigger has occurred and that the default Mask bits will be written to the EFB Header Dynamic Mask Encoder on this trigger. When 'Apply_Default_Mask_Signal' goes high the default mask bits are written into the EFB Header Dynamic Mask Encoder. The apply Corrective Mask is an interrupt to the Master DSP to signify that a corrective Trigger has occurred and that the corrective Mask bits will be written to the EFB Header Dynamic Mask Encoder on this trigger. When Apply Corrective Mask Signal goes high the Corrective Mask bits are written into the EFB Header Dynamic Mask Encoder.

Table 5.2 TIM Signal Assignments

Signal	Comment
L1Accept	Level-1 trigger Accept decision
ECReset	Event Counters (L1ID) are reset periodically
BCReset	Bunch Counters (BCID) are reset once per LHC orbit
CAL	Calibration pulse command for front-end amplifiers
SerialID	1 start bit + 24 L1ID bits + 12 BCID bits
SerialTT	1 start bit + 8 Trigger Type bits + 2 reserved bits

The controller bus writes the Default and Corrective Mask Bits into the Look Up Table (LUT). The address bus and data bus are directly mapped to the controller bus from lower order to higher order. LUT Write-in State Machine Writes the Default Mask bits and Corrective Mask bits stored in the LUT into the Dynamic Mask FIFO whenever Default Trigger interrupt and Corrective Trigger interrupt appear at the Input respectively.

EventID Write-in State Machine Writes the EventID bits into the EventID FIFO whenever the EventID bits appear at the input and EventID FIFO is not Full. The L1ID and BCID bits should be validated by the new_L1ID_valid_in and new_BCID_valid_in respectively (when new_L1ID_valid_in = '1' and new_BCID_valid_in ='1' respectively). TType Write-in State Machine Writes the Ttype bits into the Ttype FIFO whenever the Ttype bits appear at the input and the Ttype FIFO is not full. The Ttype signal should be validated by the new_Ttype_valid_in signal (when new_Ttype_valid_in = '1').

The Event ID Ttype Readout State Machine Reads out the Event ID bits and Ttype bits (Event ID Ttype) from the Event_ID FIFO and Ttype FIFO respectively using a multiplexer logic if the FIFOs are not empty. This state machine also generates Write Enable Signal to write the Event ID Ttype bits into the Event ID Ttype FIFO. The EventIDTtypeMask Readout State Machine Reads out the Dynamic Mask and Event ID and Ttype bits (Event Header Mask) through a multiplexer logic if the Dynamic Mask and EventID Ttype FIFOs are not empty. The Mask Count Keeps track of the number of Mask bits written into the Dynamic Mask FIFO. Event ID Ttype Count keeps track of the number of EventID and Ttype bits (Event ID Ttype) Written into the Event ID Ttype FIFO. EventID Ttype Mask Count Keeps track of number of EventID and Ttype and Dynamic Mask Bits (Event header Mask) read out of the Dynamic Mask and Event ID FIFOs. The relationship between the BCID and the BC reset signal is depicted in the Fig. 5.3. The TIM signals exhibit the following rules and relationship.

- BCReset is sent early by N clock periods, because its front-end protocol is N+1 bits long (instead of 1 bit), assuming the ROD latency is the same for BCReset and L1Accept. For SCT, N = 6.
- The ROD receives the ATLAS-wide BCID value (sent on to the Read-Out Buffer).
- The front-end BCID is offset from the ROD value:

  FE_BCID = ROD_BCID + M

  because the L1Accept front-end protocol is M+1 bits (instead of 1 bit), assuming the ROD latency is the same for BCR and L1Accept. For SCT, M = 2.
- The ROD_BCID value is in the range 0 - 3563, reset by BCR, when running with the LHC orbit cycle. If BCR is not generated, ROD_BCID is in the range 0 - 4095.
- The first L1ID value after ECR is 1.

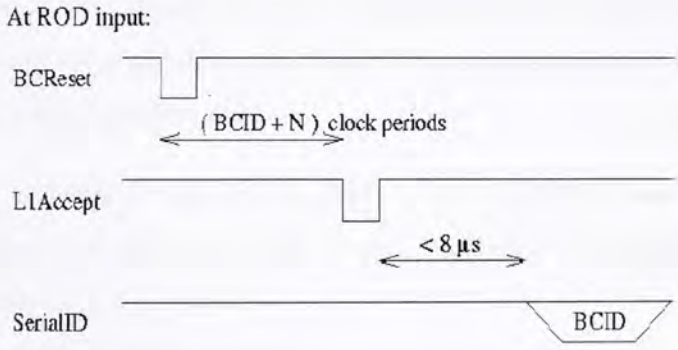


Fig. 5.5 Relationship Between BCR, L1A and Serial ID

- The CAL command is issued a programmable number of clock periods before its L1Accept.

- The SerialID and SerialTT values are independently queued before transmission for up to 8us and 10us after the L1Accept, respectively; either may be transmitted first for a given event.
- The timing relationships between the command signals should respect the front-end protocol: one command at a time, each taking a number of clock periods. Protection on TIM introduces dead time except for commands generated by the TTC system or the TIM sequencer.

5.1.2.3 Formatter Readout Modebits Encoder

The Formatter Readout Modebits encoder is located in the ROD Controller. The Formatter Readout Modebits encoder is responsible for transforming the incoming signals from the trigger inputs to the Master DSPs (Apply default Mode and Apply Corrective Mode) into 16 bit words.

All the input Signals coming into the module are 25ns synchronized exactly to the 40MHz clock. The control input from the control register controls the operation of the writein and readout state machines.

- The control_in controls all the write-in and readout state machines.
- Control(0) when High resets the state Machines.
- Control(1) When Low enables the state Machines .
- The Control(2) and Control(3) are used to load and reset the counters.

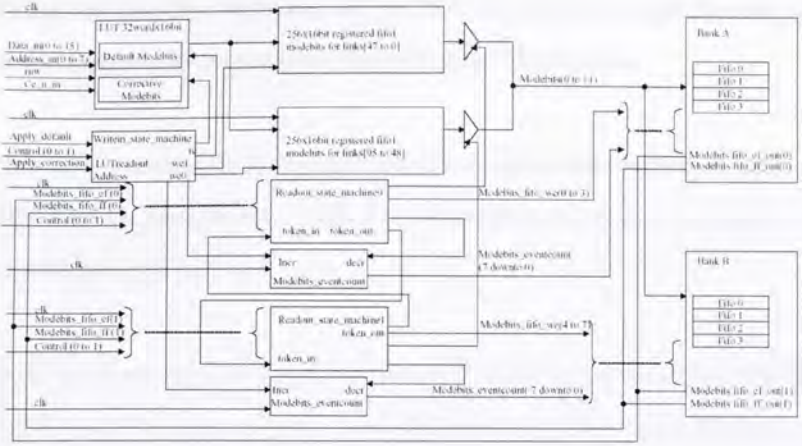


Fig. 5.6 Formatter Readout Modebits Encoder

The Two trigger interrupt signals are apply default mode and apply corrective mode. The apply Default Mode is an interrupt to the Master DSP to signify that a default Trigger has occurred and that the default Mode bits will be written to the Formatter Readout Mode bits encoder on this trigger. When Apply Default Mode Signal goes High the Default Mode bits are written into the Formatter Readout Mode bits encoder. The apply Corrective Mode is an interrupt to the Master DSP to signify that a corrective Trigger has occurred and that the corrective Mode bits will be written to the Formatter mode bits encoder on this trigger. When Apply Corrective Mode Signal goes High the Corrective Mode bits are written into the Formatter Mode bits Encoder.

The Controller bus is mapped directly to the FIFO from the lower order bit to the higher order bit (0 to 15) with link number in ascending order. The Controller bus writes the

Default and Corrective Mode Bits into the LUT. The Address bus and Data bus are directly mapped to the controller bus from lower order to higher order.

There are two data transfer engines, one FIFO and one readout state machine of each bank. When enabled by the control register, readout occurs until any of the formatter bank Mode bit FIFOs are full.

The LUT Write-in State Machine writes the Default Mode bits and Corrective Mode bits stored in the LUT into the Dynamic Mode FIFO whenever Default Trigger interrupt and Corrective Trigger interrupt appear at the Input respectively.

Readout State Machines read out the data from the FIFOs to the corresponding FIFO Bank. Whenever the target FIFO is full the Token is passed from one readout Machine to another asking to readout the data from the other Transfer FIFO. If both the target FIFOs are full the Readout state machine keeps passing the token until one of the Target Formatters are ready to write the Mode bits. Mode bits from the Transfer FIFOs are read out only when the FIFOs are not empty. When all the sixteen mode bit words are read out into the Target Formatter FIFOs the State Machine stops reading out data until the Transfer FIFOs are not empty.

There are three counters implemented in the design. Mode_Count Keeps track of the number of Mode bits written into the Dynamic Mode FIFO. Modebits_eventcount 1 keeps track of the number of words written into the Bank A of Formatter FIFOs.

'Modebits_eventcount 1' keeps track of the number of words written into the Bank A of Formatter FIFOs and 'Modebits_eventcount 2' keeps track of Bank B

5.1.2.4 L1 Pulse and Mask generator

This module consists of logic circuitry that generates L1 and Mask pulses depending upon l1_in and mask_in. The L1 pulse and mask generator is shown in Fig. 5.7

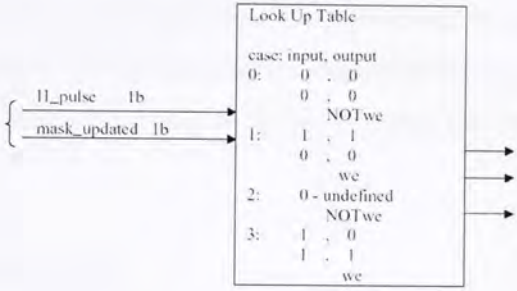


Fig. 5.7 L1 Pulse Mask Generator

The truth table is illustrated in Table 5.3. The l1_out and mask_out are then written into the front end command pulse formatter accumulator.

Table 5.3 Truth Table for L1 Mask Generator

L1_in	Mask_in	L1_out	Mask_out
1	1	0	1
0	0	0	0
1	0	1	0
0	1	0	0

5.2 Problems faced

The EFB Header dynamic mask encoder was initially designed with one write_in state machine for writing all the input bits (L1ID, BCID, Ttype) into the FIFOs. However this model had its own problems. The trigger type and Event ID bits were not input at the same time. They were triggered at different instants of time. This forced us to dedicate one write in state machine and one separate FIFO for the trigger type bits. Also the read out state machine control bits were appearing both in the synchronous process and in asynchronous. This created problems while implementing the code. Though the simulation of this code with QHDL Pro yielded expected results it was not implemented . However, this design was changed and a different readout state machine model was implemented.

5.3 Normal mode operation

Data flows in from FElinks and is decoded and stored in their FIFO buffer channels. Decoder Writes Trailer pulses and flags are trapped for time division multiplexed readout by the ROD Controller.

'HeaderTrailerLimit DecoderFIFO' and 'RodBusyLimit DecoderFIFO' occupancies are encoded with other channel occupancies for communication to the rod controller's RODBusy signal manager. Token To Readout Time Out Error flags and Data overflow error flags are muxed based on present link active selection, to the EFB for counting and inclusion into the event header. HOLDoutput from the EFB is fanned out to each FIFO

buffer. The signal 'Link numberhasToken' is encoded for the EFB so it may assert the appropriate gathering mode signals.

5.4 Synthesis and simulation

The simulation was performed on the back annotated VHDL code. The results matched the expected ones generated by simulating the original code. Particularly the fast command pulses, which are real time, did not give additional latency. The simulation waveforms are attached in Appendix A.

CHAPTER 6

Conclusion and Future Directions

6.1 Summary

As explained in this thesis, the ATLAS ROD code was implemented for the readout electronics for PIXEL detectors. The entire detector system was introduced in Chapter 1. Chapter 2 focused on the data format and protocols for the pixel detectors. The functional requirements for the RODs were also discussed. It also suggested how the design of readout driver for ATLAS Pixel detectors would be developed. Chapter 3 discussed in detail the design of the Link Formatter. Also explained is the output data format of the Link Formatter. Simulation test results and synthesis details of the Link Formatter are discussed in detail. The Formatter FPGA was implemented for the Layer 1 and the B-layer RODs. The Formatters were simulated for 5000 ns and the results matched the expected output. Chapter 4 explained in detail the design of Event Fragment Builder and the Router in detail. The Event Fragment Builder and the Router were simulated for 5000 ns and the outputs were as expected. The Chapter 5 described the module, which will be responsible for controlling, coordinating and triggering the various blocks of the Readout driver, the ROD Controller. The EFB header dynamic mask encoder worked as expected and the readout state machines in particular did not add any additional delay. The Formatter Readout Modebits Encoder also worked as expected and the token passing architecture simplified the process of reading out the data. The Front End Command Processor that generates real time command pulses and command streams did not add additional latency. The simulation test results and synthesis details are presented.

6.2 Conclusion

The readout driver prototype has been used on simulated detector data for the ATLAS Pixel Experiment. The simulation was performed by generating near real time data using the 'C' programming language. Two prototype models were designed. The Readout Driver Model depicted in this thesis is Model 2 of the prototype. Model 2 of the prototype is an extension of Model 1. Although Model 1 was not implemented real-time, it was simulated with near real-time data. The test results indicated that some changes should be made in order to overcome some trigger latency and bandwidth considerations and provide greater flexibility for routing the data to the appropriate FIFOs. Moreover, Model 1 was a data pull architecture. Apart from the data coming up link fibers, data was passed to subsequent stages by pulling/reading it out from previous stages. Studies found that token passing architecture would provide better performance. Accordingly changes in the design were made in the Event Gatherer Engine and Router design. A Dual Event Fragment Builder design was adopted in order to overcome the bandwidth problems. Though the design of the Pixel ROD has been completed, the implementation and testing of the entire ROD module has not been performed.

The functionality of the ROD controller was verified. It was simulated with a 40 MHz clock frequency and it worked well at that frequency. The trigger pulses and interrupts generated were the same as the expected results.

The functioning of the Event Fragment Builder and the Router were also same as expected. The event fragment builder stopped processing the data when none of the links

from the first formatter FPGA were active. This is exactly what the event fragment builder was expected to do. The testing of the module was performed with data generated to resemble the formatted output of the Link Formatter. The Event Fragment Builder was tested for all the 28 links of the Pixel ROD. The test results were satisfactory and met expectations.

6.3 Future Directions

The next step in the design of the RODs is the integration of various modules into the FPGAs. Once this process is completed the RODs will be tested and debugged on the board. After the completion of the implementation on board, the ROD designs will be completed for Layer 2 and the Disks. The first step towards the design of RODs is to modify the design of Formatter FPGA for the Layer 2 and the Disks. The number of links and also the size of the FIFO will have to be analyzed for each of these cases. The ROD controller designed for the Layer 1 and the B-layer will be used for the Layer 2 and Disks. A slight increase or decrease in the number of mask logic links is necessary. This will be decided after taking into account the number of links to be input. The design of the Event Fragment Builder and the Router will not be much different from the ones designed for the Layer 1 and the B-layer. However, due to the difference in input data rate and clocking, minor changes in the decoder logic may be necessary.

References

[1] ATLAS collaboration. Detector and Physics Performance Technical Design Report, CERN/LHCC/99-14 (1999).

[2] ATLAS collaboration, Technical Proposal for a general-purpose pp experiment at the Large Hadron Collider at CERN. CERN/LHCC/94-43 (1994).

[3] ATLAS collaboration, The Large Hadron Collider-conceptual design. CERN/AC/95-05 (1995).

[4] ATLAS collaboration, ATLAS DAQ, EF, LVL2 and DCS Technical Progress Report. CERN/LHCC/99-14 (1999).

[5] M.Abolins, J.Dawson, N.Ellis, Y.Ermoline, C.N.P.Gee, Ph.Farthouat, C.Schwick Specification of the LVL1/LVL2 trigger system. ATL-D-ES-0003, chap. 2 (1999).

[6] Xilinx software manual: Foundation series 3.1i software documentation

[7] ATLAS trigger-DAQ steering group, Trigger & DAQ interfaces with front-end systems: Requirement document. DAQ-No-103, Version 2.0 (1998)

[8] K. Dao, D. Fasching, O. Hayes, R. Jared, K. Marks, M. Nagel, J.M.Joseph, ATLAS SCT and pixel ROD FPGA Based Implementation. (1999).

[9] K. Dao, D. Fasching, O. Hayes, R. Jared, FPGA ROD Implementation. ATLAS-Wisconsin Group (1998)

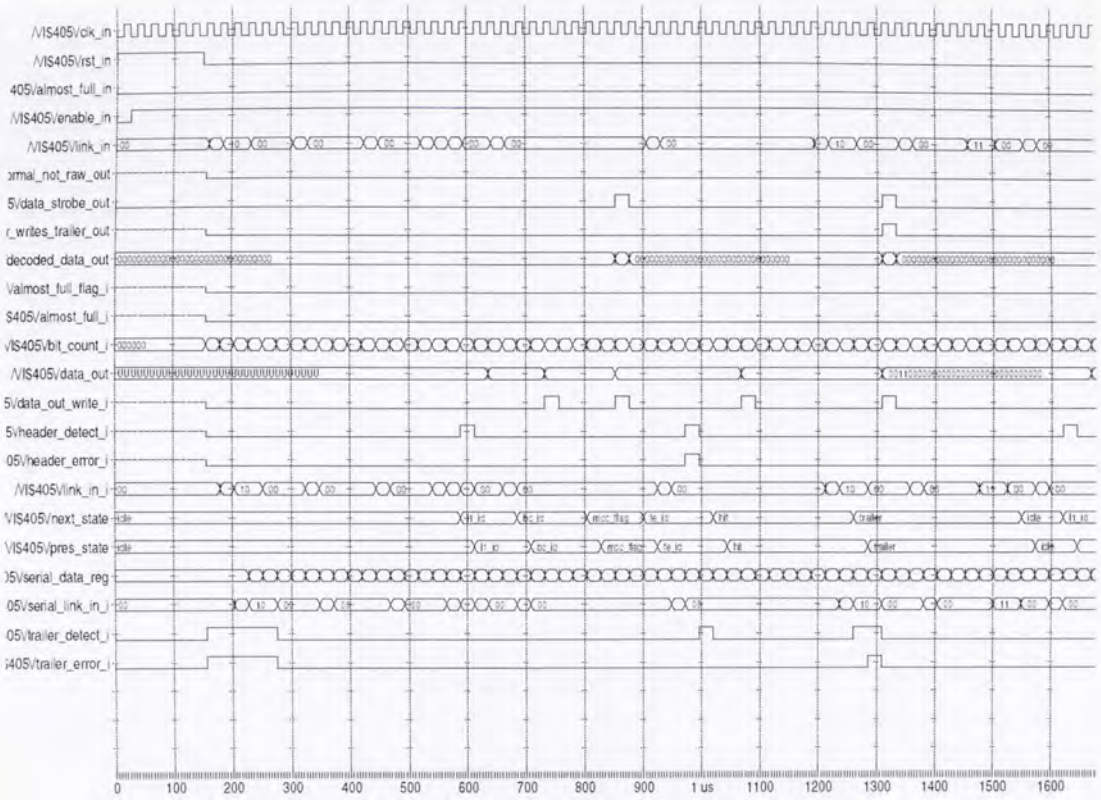
[10] ATLAS collaboration, Technical Computing Proposal. CERN/LHCC 96-43 (1996).

[11] A VHDL Primer, J.Bhaskar, 3rd Edition. Prentice Hall (1999).

[12] A VHDL Synthesis Primer, J.Bhaskar, 2nd Edition. Prentice Hall(1999).

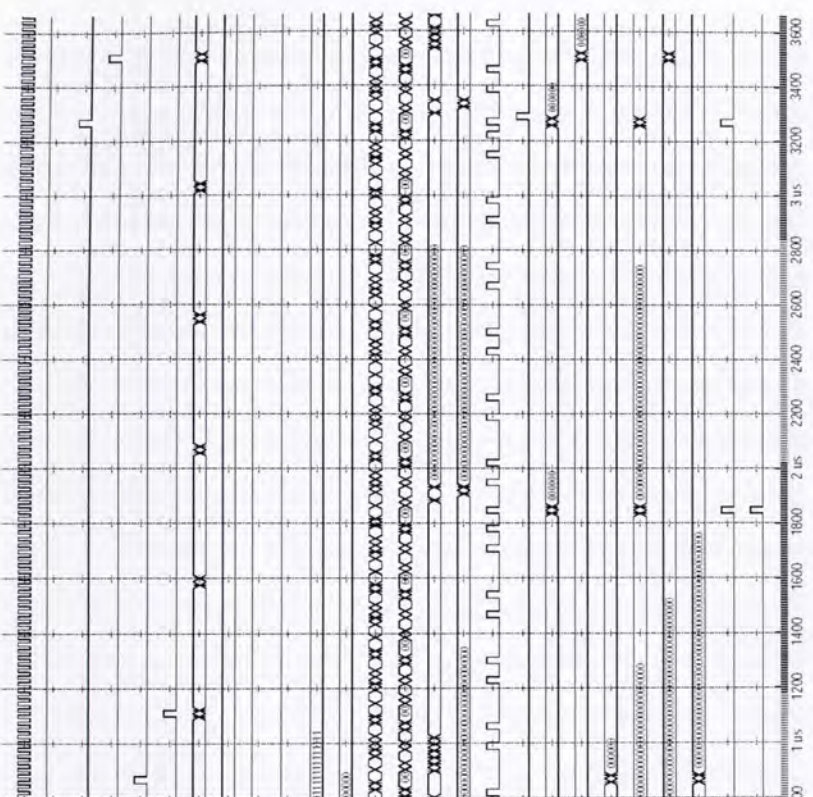
[13] "MentorGraphics," http://www.mentorgraphics.com, July 1996.

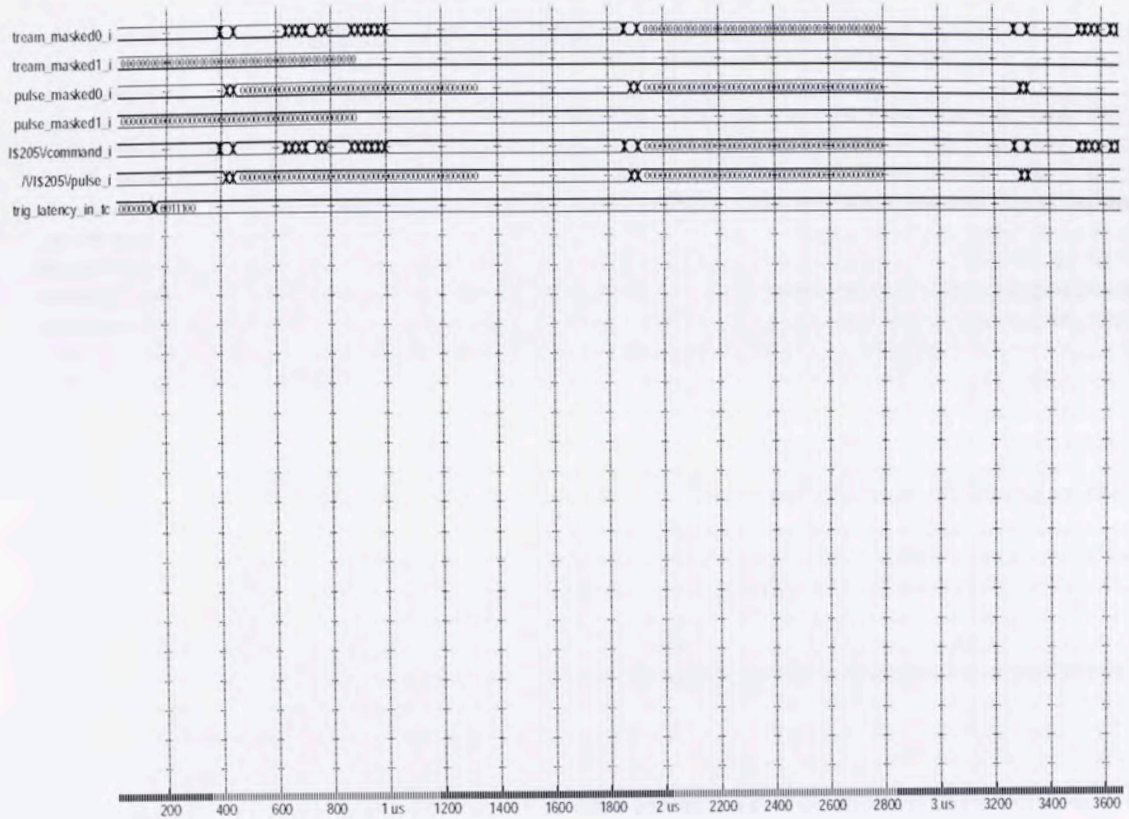
Appendix A


/VIS405/clk_in
/VIS405/rst_in
405/almost_full_in
/VIS405/enable_in
/VIS405/link_in
ormal_not_raw_out
5/data_strobe_out
r_writes_trailer_out
decoded_data_out
/almost_full_flag_i
S405/almost_full_i
/IS405/bit_count_i
/VIS405/data_out
5/data_out_write_i
5/header_detect_i
05/header_error_i
/VIS405/link_in_i
/IS405/next_state
/IS405/pres_state
)5/serial_data_reg
05/serial_link_in_i
05/trailer_detect_i
i405/trailer_error_i
0
100
200
300
400
500
600
700
800
900
1 us
1100
1200
1300
1400
1500
1600

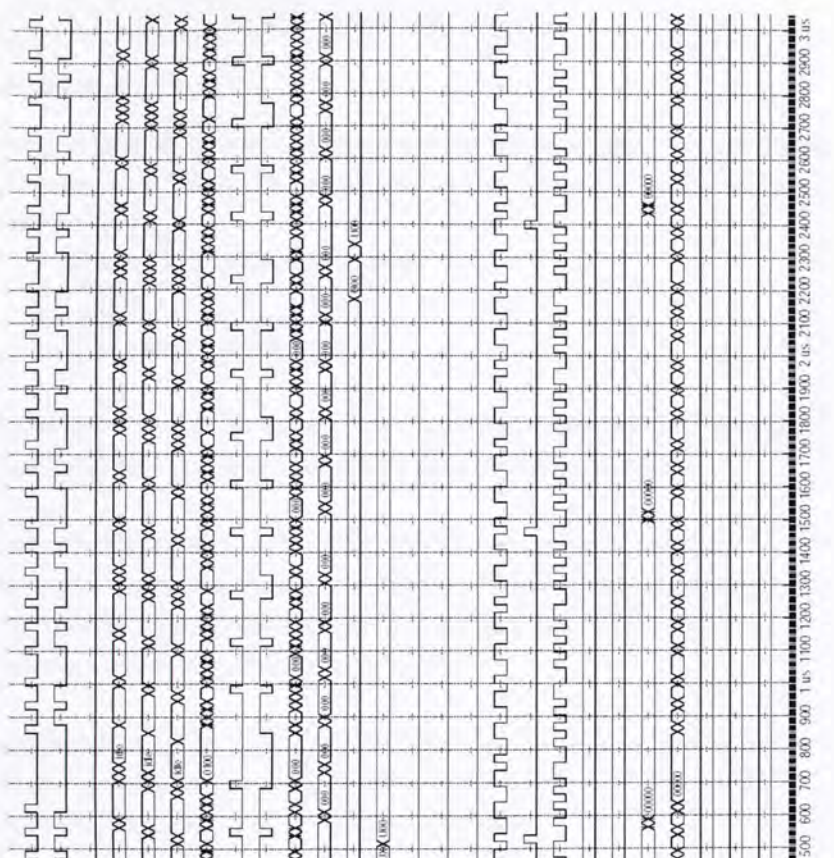
Link Formatter B-Layer

Front End Command Processor




tream_masked0_i
tream_masked1_i
pulse_masked0_i
pulse_masked1_i
I$205\/command_i
/\/I$205\/pulse_i
trig_latency_in_tc
200
400
600
800
1 us
1200
1400
1600
1800
2 us
2200
2400
2600
2800
3 us
3200
3400
3600

Formatter Readout Modebits Encoder



Appendix B

CAD Tools and Libraries

A1: Mentor

Design of the ROD was done using the mentor tools. Mentor is a popular Board level tool in common use in industry. The following Mentor tools were used for schematic capture and Simulation.

- Design Architect (Schematic Capture)
- QVHCOM (VHDL Compiler)
- QHDL Pro (VHDL Simulator)
- QuicksimII (Component Simulator)

QVHCOM: To compile for synthesis with a multi block hierarchical design, a compile script should be created. This Script uses QVHCOM to compile the design.

Design Architect: The compiled VHDL codes are converted into Symbols and Schematics using the Design architect. In one tool, one gets schematic, symbol, and VHDL editors, along with everything you need for top-down hierarchical design, including block creation and hierarchy navigation.

QuicksimII: QuicksimII is a component simulator. It lets the designer use hardware models, behavioral models, compiled models and HDL models in any combination--including models from over 200 ASIC and FPGA libraries.

QHDL Pro: It performs the functional simulation of the module created using QuicksimII.

A2: Synopsis

Synopsys FPGA express was used for compiling the models developed using Mentorgraphics.

A3: Xilinx

Xilinx UNISIM Models for I/O resources, Block and Dual Port RAMs, and other features internal to the Virtex and Virtex-E FPGAs were used for implementing FIFOs RAMs . Xilinx Foundation tools version 3.1 was used for place and routing

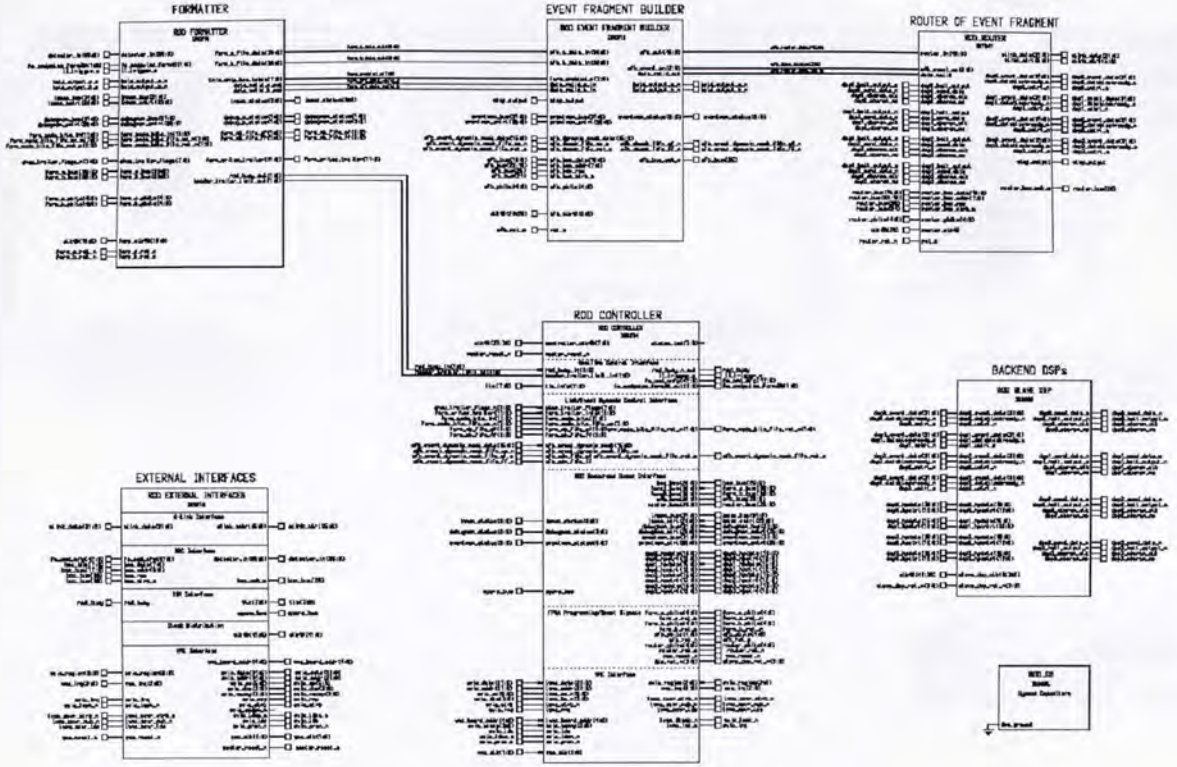
Appendix C

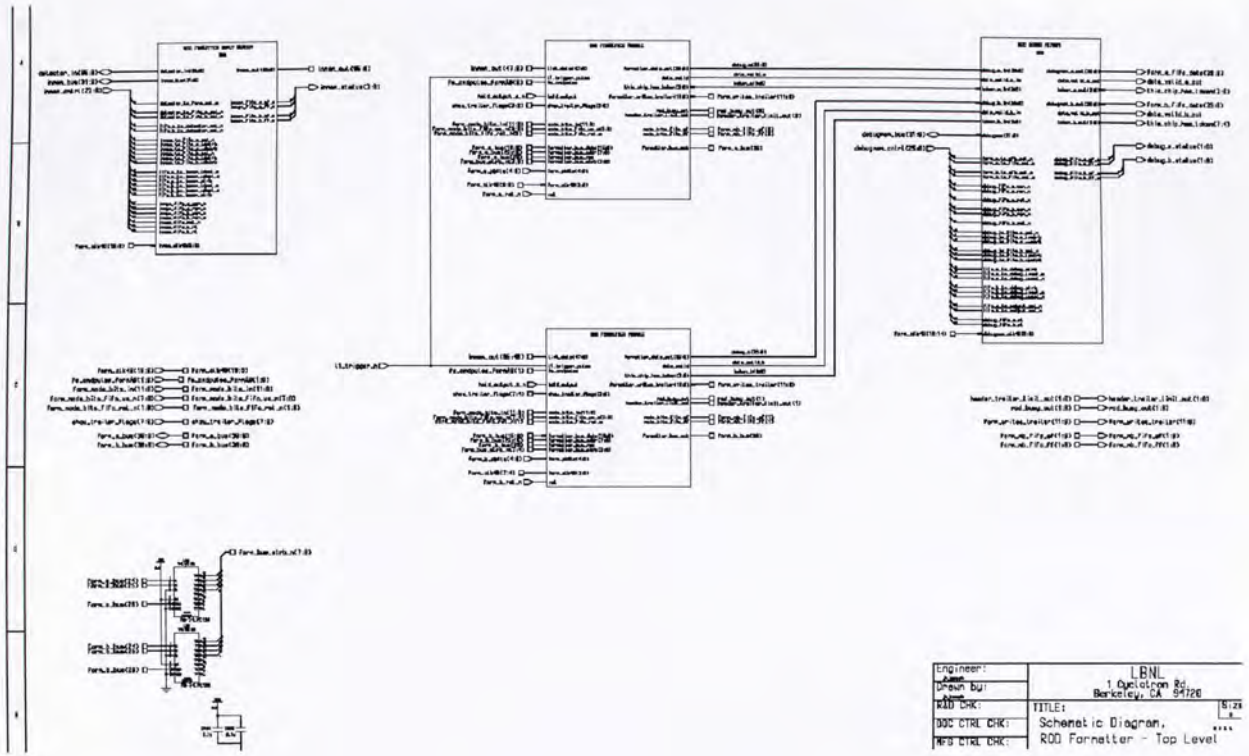
B1: Link Formatter Symbol Key

L	L1_ID
B	BC_ID
H	Header Error
T	TOT
C	Column
R	Row
M	MCC_FLAG
F	FE_ID
E	FE_FLAG
Te	Trailer Error
D	Raw_Data
K	Link Number
X	Don't Care

Appendix D

ROD Schematics


FORMATTER
EVENT FRAGMENT BUILDER
ROUTER OF EVENT FRAGMENT
ROD CONTROLLER
BACKEND DSPs
EXTERNAL INTERFACES


Engineer:
Drawn by:
KAD CHK:
DDC CTRL CHK:
MFB CTRL CHK:
LBNL
1 Cyclotron Rd.
Berkeley, CA 94720
TITLE:
Schematic Diagram,
ROD Formatter - Top Level

Appendix E

Data Sheet

Architectural Description

Virtex Array

The Virtex user-programmable gate array, shown in Figure 1, comprises two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs).

- CLBs provide the functional elements for constructing logic
- IOBs provide the interface between the package pins and the CLBs

CLBs interconnect through a general routing matrix (GRM). The GRM comprises an array of routing switches located at the intersections of horizontal and vertical routing channels. Each CLB nests into a VersaBlock™ that also provides local routing resources to connect the CLB to the GRM.

The VersaRing™ I/O interface provides additional routing resources around the periphery of the device. This routing improves I/O routability and facilitates pin locking.

The Virtex architecture also includes the following circuits that connect to the GRM.

- Dedicated block memories of 4096 bits each
- Clock DLLs for clock-distribution delay compensation and clock domain control
- 3-State buffers (BUFTs) associated with each CLB that drive dedicated segmentable horizontal routing resources

Values stored in static memory cells control the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device.

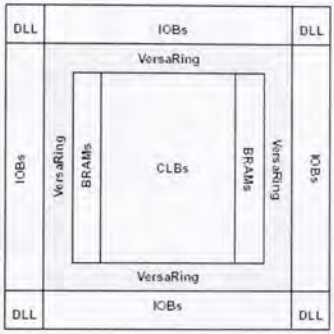
Input/Output Block

The Virtex IOB, Figure 2, features SelectIO™ inputs and outputs that support a wide variety of I/O signalling standards, see Table 1.

The three IOB storage elements function either as edge-triggered D-type flip-flops or as level sensitive latches. Each IOB has a clock signal (CLK) shared by the three flip-flops and independent clock enable signals for each flip-flop.

In addition to the CLK and CE control signals, the three flip-flops share a Set/Reset (SR). For each flip-flop, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear.

The output buffer and all of the IOB control signals have independent polarity controls.



vax_b.eps

Figure 1: Virtex Architecture Overview

All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Two forms of over-voltage protection are provided, one that permits 5 V compliance, and one that does not. For 5 V compliance, a Zener-like structure connected to ground turns on when the output rises to approximately 6.5 V. When PCI 3.3 V compliance is required, a conventional clamp diode is connected to the output supply voltage, V_{CCO}.

Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration, all pins not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs can optionally be pulled up.

The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration.

All Virtex IOBs support IEEE 1149.1-compatible boundary scan testing.

© 2001 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and disclaimers are as listed at http://www.xilinx.com/legal.htm. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

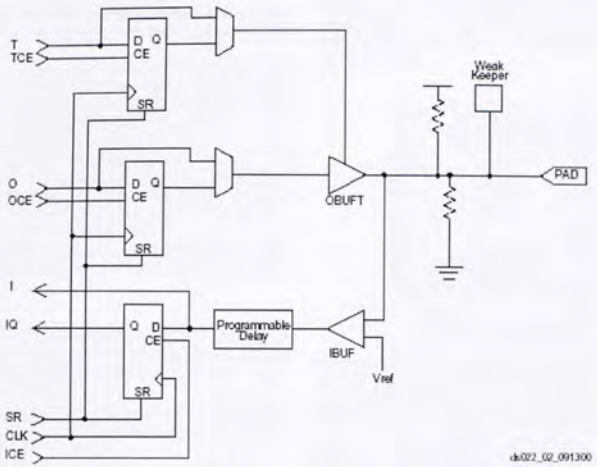


Figure 2: Virtex Input/Output Block (IOB)

Table 1: Supported Select I/O Standards

I/O Standard	Input Reference Voltage (V_{REF})	Output Source Voltage (V_{CCO})	Board Termination Voltage (V_{TT})	5 V Tolerant
LVTTL 2 – 24 mA	N/A	3.3	N/A	Yes
LVCMOS2	N/A	2.5	N/A	Yes
PCI, 5 V	N/A	3.3	N/A	Yes
PCI, 3.3 V	N/A	3.3	N/A	No
GTL	0.8	N/A	1.2	No
GTL+	1.0	N/A	1.5	No
HSTL Class I	0.75	1.5	0.75	No
HSTL Class III	0.9	1.5	1.5	No
HSTL Class IV	0.9	1.5	1.5	No
SSTL3 Class I & II	1.5	3.3	1.5	No
SSTL2 Class I & II	1.25	2.5	1.25	No
CTT	1.5	3.3	1.5	No
AGP	1.32	3.3	N/A	No

Input Path

A buffer in the Virtex IOB input path routes the input signal either directly to internal logic or through an optional input flip-flop.

An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the FPGA, and when used, assures that the pad-to-pad hold time is zero.

Each input buffer can be configured to conform to any of the low-voltage signalling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, V_{REF}. The need to supply V_{REF} imposes constraints on which standards can used in close proximity to each other. See I/O Banking, page 3.

There are optional pull-up and pull-down resistors at each input for use after configuration. Their value is in the range 50 kΩ – 100 kΩ.

Output Path

The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop.

The 3-state control of the output can also be routed directly from the internal logic or through a flip-flip that provides synchronous enable and disable.

Each output driver can be individually programmed for a wide range of low-voltage signalling standards. Each output buffer can source up to 24 mA and sink up to 48mA. Drive strength and slew rate controls minimize bus transients.

In most signalling standards, the output High voltage depends on an externally supplied V_{CCO} voltage. The need to supply V_{CCO} imposes constraints on which standards can be used in close proximity to each other. See I/O Banking, page 3.

An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the pin is connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way eliminates bus chatter.

Because the weak-keeper circuit uses the IOB input buffer to monitor the input level, an appropriate V_{REF} voltage must be provided if the signalling standard requires one. The provision of this voltage must comply with the I/O banking rules.

I/O Banking

Some of the I/O standards described above require V_{CCO} and/or V_{REF} voltages. These voltages externally and connected to device pins that serve groups of IOBs, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank.

Eight I/O banks result from separating each edge of the FPGA into two banks, as shown in Figure 3. Each bank has multiple V_{CCO} pins, all of which must be connected to the same voltage. This voltage is determined by the output standards in use.

Figure 3: Virtex I/O Banks

Within a bank, output standards can be mixed only if they use the same V_{CCO}. Compatible standards are shown in Table 2. GTL and GTL+ appear under all voltages because their open-drain outputs do not depend on V_{CCO}.

Table 2: **Compatible Output Standards**

V_{CCO}	Compatible Standards
3.3 V	PCI, LVTTL, SSTL3 I, SSTL3 II, CTT, AGP, GTL, GTL+
2.5 V	SSTL2 I, SSTL2 II, LVCMOS2, GTL, GTL+
1.5 V	HSTL I, HSTL III, HSTL IV, GTL, GTL+

Some input standards require a user-supplied threshold voltage, V_{REF}. In this case, certain user-I/O pins are automatically configured as inputs for the V_{REF} voltage. Approximately one in six of the I/O pins in the bank assume this role.

The V_{REF} pins within a bank are interconnected internally and consequently only one V_{REF} voltage can be used within each bank. All V_{REF} pins in the bank, however, must be connected to the external voltage source for correct operation.

Within a bank, inputs that require V_{REF} can be mixed with those that do not. However, only one V_{REF} voltage can be used within a bank. Input buffers that use V_{REF} are not 5 V tolerant. LVTTL, LVCMOS2, and PCI 33 MHz 5 V, are 5 V tolerant.

The V_{CCO} and V_{REF} pins for each bank appear in the device Pinout tables and diagrams. The diagrams also show the bank affiliation of each I/O.

Within a given package, the number of V_{REF} and V_{CCO} pins can vary depending on the size of device. In larger devices.

more I/O pins convert to V_{REF} pins. Since these are always a superset of the V_{REF} pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device if necessary. All the V_{REF} pins for the largest device anticipated must be connected to the V_{REF} voltage, and not used for I/O.

In smaller devices, some V_{CCO} pins used in larger devices do not connect within the package. These unconnected pins can be left unconnected externally, or can be connected to the V_{CCO} voltage to permit migration to a larger device if necessary.

In TQ144 and PQ/HQ240 packages, all V_{CCO} pins are bonded together internally, and consequently the same V_{CCO} voltage must be connected to all of them. In the CS144 package, bank pairs that share a side are interconnected internally, permitting four choices for V_{CCO}. In both cases, the V_{REF} pins remain internally connected as eight banks, and can be used as described previously.

Configurable Logic Block

The basic building block of the Virtex CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each Virtex CLB contains four LCs, organized in two similar slices, as shown in Figure 4.

Figure 5 shows a more detailed view of a single slice.

In addition to the four basic LCs, the Virtex CLB contains logic that combines function generators to provide functions of five or six inputs. Consequently, when estimating the number of system gates provided by a given device, each CLB counts as 4.5 LCs.

Look-Up Tables

Virtex function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM.

The Virtex LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing.

Storage Elements

The storage elements in the Virtex slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing the function generators.

In addition to Clock and Clock Enable signals, each Slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified for it in the configuration. BY forces it into the opposite state. Alternatively, these signals can be configured to operate asynchronously. All of the control signals are independently invertible, and are shared by the two flip-flops within the slice.

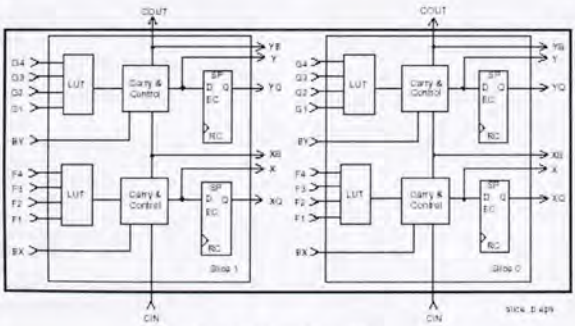


Figure 4: **2-Slice Virtex CLB**

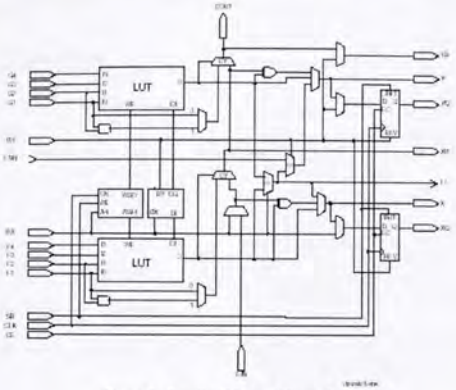


Figure 5: **Detailed View of Virtex Slice**

Additional Logic

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs.

Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs.

Each CLB has four direct feedthrough paths, one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources.

Arithmetic Logic

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Virtex CLB supports two separate carry chains, one per Slice. The height of the carry chains is two bits per CLB.

The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation.

The dedicated carry path can also be used to cascade function generators for implementing wide logic functions.

BUFTs

Each Virtex CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. See Dedicated Routing, page 7. Each Virtex BUFT has an independent 3-state control pin and an independent input pin.

Block SelectRAM

Virtex FPGAs incorporate several large Block SelectRAM memories. These complement the distributed LUT SelectRAMs that provide shallow RAM structures implemented in CLBs.

Block SelectRAM memory blocks are organized in columns. All Virtex devices contain two such columns, one along each vertical edge. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Virtex device 64 CLBs high contains 16 memory blocks per column, and a total of 32 blocks.

Table 3 shows the amount of Block SelectRAM memory that is available in each Virtex device.

Table 3: **Virtex Block SelectRAM Amounts**

Device	# of Blocks	Total Block SelectRAM Bits
XCV50	8	32,768
XCV100	10	40,960
XCV150	12	49,152
XCV200	14	57,344
XCV300	16	65,536
XCV400	20	81,920
XCV600	24	98,304
XCV800	28	114,688
XCV1000	32	131,072

Each Block SelectRAM cell, as illustrated in Figure 6, is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion.

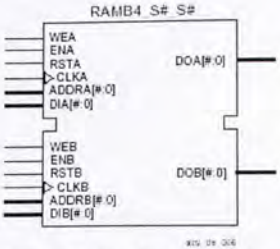


Figure 6: **Dual-Port Block SelectRAM**

Table 4 shows the depth and width aspect ratios for the Block SelectRAM.

Table 4: **Block SelectRAM Port Aspect Ratios**

Width	Depth	ADDR Bus	Data Bus
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

The Virtex Block SelectRAM also includes dedicated routing to provide an efficient interface with both CLBs and other Block SelectRAMs.

Programmable Routing Matrix

It is the longest delay path that limits the speed of any worst-case design. Consequently, the Virtex routing architecture and its place-and-route software were defined in a single optimization process. This joint optimization minimizes long-path delays, and consequently, yields the best system performance.

The joint optimization also reduces design compilation times because the architecture is software-friendly. Design cycles are correspondingly reduced due to shorter design iteration times.

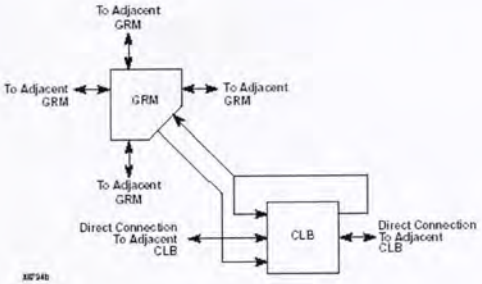


Figure 7: **Virtex Local Routing**

Local Routing

The VersaBlock provides local routing resources, as shown in Figure 7, providing the following three types of connections.

- Interconnections among the LUTs, flip-flops, and GRM
- Internal CLB feedback paths that provide high-speed connections to LUTs within the same CLB, chaining them together with minimal routing delay
- Direct paths that provide high-speed connections between horizontally adjacent CLBs, eliminating the delay of the GRM.

Appendix F

VHDL CODE

Link_formatter_layer1.vhd

```
----------------------------------------------------------------------------
-- University of Oklahoma and Lawrence Berkeley National Labs
-- ATLAS ROD ELECTRONICS
-- Author -- Sriram.S
----------------------------------------------------------------------------
-- Filename:
-- formatter_pxl.vhd
-- Description:
-- ROD Front End Decoder
-- Double Channel
-- PIXEL Version
--
-- This file contains the serial decoding state machine for the PIXEL front
-- end. You should refer to three documents to better understand how the
-- vhdl for the decoder works.
-- 1) "Format of Data from the PIXEL Front End Decoders" describes what
-- the output data words look like after the serial bitstreams have
-- been decoded and converted from serial to parallel.
-- /home/edafiles/atlas/rod/docs/pix_fe_format.txt(or .ps)
--
-- 2) "MCC Event Structure" presents a state transition diagram
-- that shows how to decode the MCC serial data. The VHDL logic
-- for the PIXEL decoder is closely based on this diagram.
-- /home/edafiles/atlas/rod/docs/mcc_format.ps


entity link_formatter_layer1 is
port (
clk_in : in std_logic;
rst_in : in std_logic;
almost_full_in : in std_logic; -- 1 = header/trailer
-- 0 = normal
enable_in : in std_logic; -- enable formatter activity
link_in : in std_logic_vector(1 downto 0); -- front end serial data
normal_not_raw_out : out std_logic; -- decoder is in 1= normal data mode, 0 = raw data mode
data_strobe_out : out std_logic; -- indicates a new output word is available
decoder_writes_trailer_out : out std_logic; -- trailer being written
decoded_data_out : out std_logic_vector(31 downto 0) -- output data word
);
end link_formatter_layer1 ;
architecture link_formatter_layer1_arch of link_formatter_layer1 is
type pixel_state_typedef is(IDLE, L1_ID, BC_ID, MCC_FLAG, FE_ID, FE_FLAG, HIT, TRAILER,
RAW_DATA);
signal almost_full_flag_i : std_logic ;
signal almost_full_i : std_logic ;
signal bit_count_i : unsigned (5 downto 0);
signal bit_count_reset_i : std_logic ;
signal data_out : std_logic_vector(31 downto 0) ;
signal data_out_temp : std_logic_vector(31 downto 0) ;
signal data_out_write_i : std_logic ;
signal data_strobe_lower_i : std_logic ;
signal decoder_writes_trailer_i : std_logic ;
signal fe_reg : std_logic_vector(3 downto 0) ;
signal header_6bit : std_logic ;
```

```
signal header_detect_i : std_logic ;
signal header_error_i : std_logic ;
signal header_error_del1_i : std_logic;
signal header_sh_reg : std_logic_vector(5 downto 0) ;
signal hit_pattern_invalid : std_logic ;
signal link_in_i : std_logic_vector(1 downto 0) ;
signal next_bit_count_i : unsigned ( 5 downto 0);
signal next_bit_count_ii : unsigned ( 5 downto 0);
signal next_state : pixel_state_typedef; -- pixel decoder fsm states
signal new_hit_data_i : std_logic ;
signal new_hit_location_i : std_logic ;
signal normal_not_raw_i : std_logic ;
signal raw_data_odd_i : std_logic ;
signal raw_data_even_i : std_logic ;
signal pres_state : pixel_state_typedef; -- pixel decoder fsm states
signal serial_data_reg : std_logic_vector(27 downto 0);
signal serial_link_in_i : std_logic_vector(1 downto 0) ;
signal trailer_counter_i : unsigned (4 downto 0);
signal trailer_detect_i : std_logic;
signal trailer_error_i : std_logic ;
signal trailer_error_del1_i : std_logic ;
signal trailer_error_flag_i : std_logic ;
signal trailer_zeroes : std_logic ;
-------------------------------------------------------------------------------
-- Constant Declaration
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- Componenent Declaration
-------------------------------------------------------------------------------
begin -- Main body of the vhdl code
-------------------------------------------------------------------------------
-- Component Instantiation
-------------------------------------------------------------------------------
input_register : process (clk_in, rst_in)
begin
if (rst_in = '1') then
serial_data_reg(27 downto 0) <= "0000000000000000000000000000";
serial_link_in_i <= (others => '0');
link_in_i <= (others => '0');
elsif (clk_in'event and clk_in = '1') then
almost_full_i <= almost_full_in;
link_in_i <= link_in;
serial_link_in_i(1) <= link_in_i(1) and enable_in;
serial_link_in_i(0) <= link_in_i(0) and enable_in;
serial_data_reg(27 downto 0) <= serial_data_reg(25 downto 0) & serial_link_in_i(1) & serial_link_in_i(0);
normal_not_raw_out <= normal_not_raw_i;
end if;
end process input_register;
-----
find_header : process(clk_in, rst_in)
begin -- find header
if (rst_in = '1') then
header_sh_reg(5 downto 0) <= "000000";
elsif (clk_in'event and clk_in = '1') then
-- correct header pattern is 11101.A header can also have upto one bit of error
-- in any location.
```

```
header_6bit <= '0';
header_sh_reg(5 downto 0) <= header_sh_reg(3 downto 0) & serial_data_reg(23) & serial_data_reg(22);
case header_sh_reg(5 downto 1) is
when "11101" => -- "111101 is a possible hit pattern with an error
header_detect_i <= '1';
header_error_i <= header_sh_reg(0);
header_6bit <= header_sh_reg(0);
when "01101" =>
header_detect_i <= '1';
header_error_i <= '1';
when "10101" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11001" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11111" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11100" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11010" =>
header_detect_i <= header_sh_reg(0); -- "111010" is a possible hit pattern with an error
header_error_i <= header_sh_reg(0);
header_6bit <= header_sh_reg(0);
when others =>
header_detect_i <= '0';
header_error_i <= '0';
end case;
end if;
end process find_header;
------
find_trailer : process(clk_in, rst_in) -- no errors allowed
begin -- find trailer
if (rst_in = '1') then
trailer_counter_i <= "00000";
trailer_detect_i <= '0';
trailer_error_i <= '0';
trailer_zeroes <= '0';
elsif (clk_in'event and clk_in = '1') then
if(serial_data_reg(21) = '0'and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+2;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
```

```
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-2;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
end if;
if (std_logic_vector(trailer_counter_i) = "00000") then
trailer_detect_i <= '1'; -- if 0 or 1 followed by 21 0's a trailer is detected
trailer_error_i <= not serial_data_reg(23); -- zero followed by 21 0's is a trailer with a bit error
trailer_zeroes <= '1'; -- trailer with 1 or 0 followed by 21 zeroes
else
trailer_detect_i <= '0';
trailer_error_i <= '0';
trailer_zeroes <= '0';
end if;
end if;
end process find_trailer;
-----
--------------------------------------------------------------------------------
-- the next process is all combinatorial logic and performs which state
-- transitions are to take place
--------------------------------------------------------------------------------
-- purpose: default FSM values
-- type : combinational
-- inputs : pres_state,bit_count_i,header_detect_i,serial_data_reg,trailer_error_i,
-- trailer_detect_i,data_out_write_i
--outputs: next_state,next_bit_count_i,trailer_error_flag_i
```

```
pxl_comb_fsm : process (pres_state,trailer_zeroes, bit_count_i, header_detect_i, serial_data_reg,
trailer_error_i, trail
er_detect_i, data_out_write_i,decoder_writes_trailer_i)
begin -- process pxl_comb_fsm
next_bit_count_i <= bit_count_i + "000010";
trailer_error_flag_i <= '0';
-- check all state transition conditions
if(trailer_detect_i = '1' and trailer_error_i = '0' and pres_state/= idle and pres_state/= l1_id and
pres_state/= fe_i
d and
pres_state/= trailer) then
next_state <= trailer;
trailer_error_flag_i <= '1';
next_bit_count_i <= "000000";
end if;
case pres_state is
when idle =>
if (std_logic_vector(bit_count_i) = "100110" and header_detect_i = '1') then
next_state <= l1_id;
next_bit_count_i <= "000011";
elsif (std_logic_vector(bit_count_i) = "000010" and header_detect_i = '1')then
next_state <= l1_id;
next_bit_count_i <= "000011";
elsif (std_logic_vector(bit_count_i) = "000100" and header_detect_i = '1') then
next_state <= l1_id;
next_bit_count_i <= "000011";
end if;
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
normal_not_raw_i <= '1';
when l1_id =>
if (std_logic_vector(bit_count_i) = "001001")then
if (serial_data_reg(26) = '1') then
next_state <= bc_id;
next_bit_count_i <= "000000";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
end if;
end if;
normal_not_raw_i <= '1';
when bc_id =>
if (std_logic_vector(bit_count_i) = "001000") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif(serial_data_reg(27) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000011";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(24) = '0') then
next_state <= mcc_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(24) = '1') then
next_state <= fe_id;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when mcc_flag =>
if (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000010";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000010";
fe_reg(3 downto 0) <= serial_data_reg(21 downto 18);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when fe_id =>
if (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0')then
next_state <= raw_data;
next_bit_count_i <= "000010";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (serial_data_reg(25 downto 22) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
```

```
raw_data_odd_i <= '0';
end if;
end if;
elsif(std_logic_vector(bit_count_i) = "001010") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if (serial_data_reg(26 downto 23) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when hit =>
if (std_logic_vector(bit_count_i) = "010110") then
if (serial_data_reg(26) = '0') then
if (serial_data_reg(27) = '1' and trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000000";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
end if;
elsif (serial_data_reg(26) = '1') then
if(trailer_detect_i = '1')then
next_state <= trailer;
next_bit_count_i <= "000010";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(25 downto 22) = "1110")then
next_state <= fe_id;
next_bit_count_i <= "000010";
new_hit_location_i <= '1';
fe_reg(3 downto 0) <= serial_data_reg(23 downto 20);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(25 downto 22) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000010";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000010";
new_hit_data_i <= '1';
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
elsif (std_logic_vector(bit_count_i) = "010111") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if(trailer_detect_i = '1')then
next_state <= trailer;
next_bit_count_i <= "000011";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(26 downto 23) = "1110")then
next_state <= fe_id;
next_bit_count_i <= "000011";
new_hit_location_i <= '1';
fe_reg(3 downto 0) <= serial_data_reg(22 downto 19);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26 downto 23) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(25) = '0') then
next_state <= hit;
next_bit_count_i <= "000011";
new_hit_data_i <= '1';
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when fe_flag =>
if (std_logic_vector(bit_count_i) = "001010") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000001";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000001";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000001";
fe_reg <= serial_data_reg(24 downto 21);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
elsif (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000000";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000010";
fe_reg(3 downto 0) <= serial_data_reg(23 downto 20);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when trailer =>
if (std_logic_vector(bit_count_i) = "010110") then
next_state <= idle;
next_bit_count_i <= "000000";
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
elsif (std_logic_vector(bit_count_i) = "010111") then
next_state <= idle;
next_bit_count_i <= "000000";
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
end if;
normal_not_raw_i <= '1';
when raw_data =>
if ((trailer_detect_i = '1' and trailer_error_i = '0')or trailer_zeroes = '1') then
next_state <= trailer;
trailer_error_flag_i <= trailer_error_i;
else
next_state <= raw_data;
end if;
when others => null;
end case;
normal_not_raw_i <= '0';
end process pxl_comb_fsm;
```

```
-- purpose: Flip flops holding the current state values are set here
-- type : combinational
-- inputs : clk_in,rst_in,next_state,bit_count_i
-- outputs: pres_state,bit_count_i
-------------------------------------------------------------------------------
set_clocked_fsm : process (clk_in, rst_in)
begin -- process set_clocked_fsm
if (rst_in = '1') then
pres_state <= idle;
bit_count_i <= "000000";
elsif (clk_in'event and clk_in = '1') then
pres_state <= next_state;
bit_count_i <= next_bit_count_i;
if (std_logic_vector(next_bit_count_i) = "000000") then
bit_count_reset_i <= '1';
else
bit_count_reset_i <= '0';
end if;
end if;
end process set_clocked_fsm;
-- purpose: This process looks at the current state and outputs the data
--words depending on which state and which bit is being shifted in
-- type : combinational
-- inputs : clk_in,rst_in
-- outputs: header_error_del1,data_out_write_i,decoder_writes_trailer_i,normal_not_raw,
process (clk_in, rst_in)
begin -- process
if (rst_in = '1') then
elsif (clk_in'event and clk_in = '1') then
header_error_del1_i <= header_error_i;
trailer_error_del1_i <= trailer_error_i;
data_out_write_i <= '0';
decoder_writes_trailer_i <= '0';
if (almost_full_i = '1') then
almost_full_flag_i <= '1';
end if;
case pres_state is
when idle =>
almost_full_flag_i <= '0';


when l1_id =>
if (std_logic_vector(bit_count_i) = "000011") then
data_out(31 downto 28) <= "000" & header_error_i;
data_out(15 downto 8) <= header_sh_reg(4) & serial_data_reg(27 downto 21);
end if;
when bc_id =>
if std_logic_vector(bit_count_i) = "000000" then
data_out_write_i <= '1';
data_out(7 downto 0) <= serial_data_reg(27 downto 20);
end if;
when mcc_flag =>
if (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "001";
data_out(7 downto 0) <= header_sh_reg(4) & serial_data_reg(27 downto 21);
```

```
end if;
when hit =>
if (std_logic_vector(bit_count_i) = "000010" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 30) <= "11";
data_out(29 downto 0) <= "00000" & fe_reg(3 downto 0) & serial_data_reg(27 downto 7);
elsif (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 30) <= "11";
data_out(29 downto 0) <= "00000" & fe_reg(3 downto 0) & header_sh_reg(4)
& serial_data_reg(27 downto 8);
end if;
when fe_flag =>
if (std_logic_vector(bit_count_i) = "000010" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "100";
data_out(11 downto 0) <= fe_reg(3 downto 0) & serial_data_reg(27 downto 20);
elsif (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "100";
data_out(11 downto 0) <= fe_reg(3 downto 0) &header_sh_reg(4) & serial_data_reg(27 downto 21);
end if;
when trailer =>
if (std_logic_vector(bit_count_i) = "000000" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 23) <= "001" & trailer_error_i & "00000" ;
decoder_writes_trailer_i <= '1';
end if;
when raw_data =>
if raw_data_even_i = '1' then
data_out <= "01" & data_out(27 downto 0) & serial_data_reg(27 downto 26);
elsif raw_data_odd_i = '1' then
data_out <= "01" & data_out(27 downto 0) & serial_data_reg(26 downto 25);
end if;
if (((next_state/= raw_data) or(std_logic_vector(bit_count_i) = "000000"))
and almost_full_flag_i = '0')then
data_out_write_i <= '1';
end if;
when others => null;
end case;
end if;
end process;
process (clk_in, rst_in, decoder_writes_trailer_i)
begin -- process
if (rst_in = '1') then
data_strobe_lower_i <= '0';
elsif (clk_in'event and clk_in = '1') then
if (data_out_write_i = '1') then
if (decoder_writes_trailer_i = '1') then
data_strobe_lower_i <= '0';
else
data_strobe_lower_i <= not data_strobe_lower_i;
end if;
end if;
end if;
decoder_writes_trailer_out <= decoder_writes_trailer_i;
```

```
end process;
-- purpose: to write the output signals
process (data_out_write_i, data_strobe_lower_i, decoder_writes_trailer_i, data_out)
begin -- process
data_strobe_out <= '0';
decoded_data_out <= (others => '0');
if data_out_write_i = '1' then
if data_strobe_lower_i = '1' then
data_strobe_out <= '1';
decoded_data_out <= data_out;
-- normal_not_raw_out <= normal_not_raw_i;
end if;
end if;
end process;
end link_formatter_layer1_arch;
```

Default and Corrective Mode Bits into the LUT. The Address bus and Data bus are directly mapped to the controller bus from lower order to higher order.

There are two data transfer engines, one FIFO and one readout state machine of each bank. When enabled by the control register, readout occurs until any of the formatter bank Mode bit FIFOs are full.

The LUT Write-in State Machine writes the Default Mode bits and Corrective Mode bits stored in the LUT into the Dynamic Mode FIFO whenever Default Trigger interrupt and Corrective Trigger interrupt appear at the Input respectively.

Readout State Machines read out the data from the FIFOs to the corresponding FIFO Bank. Whenever the target FIFO is full the Token is passed from one readout Machine to another asking to readout the data from the other Transfer FIFO. If both the target FIFOs are full the Readout state machine keeps passing the token until one of the Target Formatters are ready to write the Mode bits. Mode bits from the Transfer FIFOs are read out only when the FIFOs are not empty. When all the sixteen mode bit words are read out into the Target Formatter FIFOs the State Machine stops reading out data until the Transfer FIFOs are not empty.

There are three counters implemented in the design. Mode_Count Keeps track of the number of Mode bits written into the Dynamic Mode FIFO. Modebits_eventcount 1 keeps track of the number of words written into the Bank A of Formatter FIFOs.

'Modebits_eventcount 1' keeps track of the number of words written into the Bank A of Formatter FIFOs and 'Modebits_eventcount 2' keeps track of Bank B

5.1.2.4 L1 Pulse and Mask generator

This module consists of logic circuitry that generates L1 and Mask pulses depending upon l1_in and mask_in. The L1 pulse and mask generator is shown in Fig. 5.7

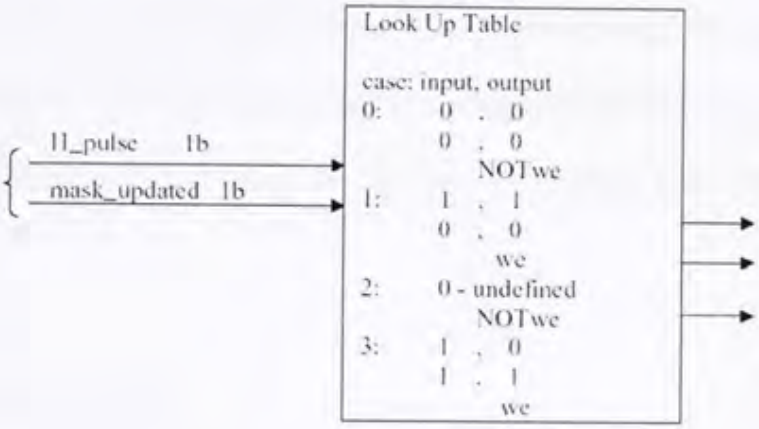


Fig. 5.7 L1 Pulse Mask Generator

The truth table is illustrated in Table 5.3. The l1_out and mask_out are then written into the front end command pulse formatter accumulator.

Table 5.3 Truth Table for L1 Mask Generator

L1_in	Mask_in	L1_out	Mask_out
1	1	0	1
0	0	0	0
1	0	1	0
0	1	0	0

5.2 Problems faced

The EFB Header dynamic mask encoder was initially designed with one write_in state machine for writing all the input bits (L1ID, BCID, Ttype) into the FIFOs. However this model had its own problems. The trigger type and Event ID bits were not input at the same time. They were triggered at different instants of time. This forced us to dedicate one write in state machine and one separate FIFO for the trigger type bits. Also the read out state machine control bits were appearing both in the synchronous process and in asynchronous. This created problems while implementing the code. Though the simulation of this code with QHDL Pro yielded expected results it was not implemented . However, this design was changed and a different readout state machine model was implemented.

5.3 Normal mode operation

Data flows in from FElinks and is decoded and stored in their FIFO buffer channels. Decoder Writes Trailer pulses and flags are trapped for time division multiplexed readout by the ROD Controller.

'HeaderTrailerLimit DecoderFIFO' and 'RodBusyLimit DecoderFIFO' occupancies are encoded with other channel occupancies for communication to the rod controller's RODBusy signal manager. Token To Readout Time Out Error flags and Data overflow error flags are muxed based on present link active selection, to the EFB for counting and inclusion into the event header. HOLDoutput from the EFB is fanned out to each FIFO

buffer. The signal 'Link numberhasToken' is encoded for the EFB so it may assert the appropriate gathering mode signals.

5.4 Synthesis and simulation

The simulation was performed on the back annotated VHDL code. The results matched the expected ones generated by simulating the original code. Particularly the fast command pulses, which are real time, did not give additional latency. The simulation waveforms are attached in Appendix A.

CHAPTER 6

Conclusion and Future Directions

6.1 Summary

As explained in this thesis, the ATLAS ROD code was implemented for the readout electronics for PIXEL detectors. The entire detector system was introduced in Chapter 1. Chapter 2 focused on the data format and protocols for the pixel detectors. The functional requirements for the RODs were also discussed. It also suggested how the design of readout driver for ATLAS Pixel detectors would be developed. Chapter 3 discussed in detail the design of the Link Formatter. Also explained is the output data format of the Link Formatter. Simulation test results and synthesis details of the Link Formatter are discussed in detail. The Formatter FPGA was implemented for the Layer 1 and the B-layer RODs. The Formatters were simulated for 5000 ns and the results matched the expected output. Chapter 4 explained in detail the design of Event Fragment Builder and the Router in detail. The Event Fragment Builder and the Router were simulated for 5000 ns and the outputs were as expected. The Chapter 5 described the module, which will be responsible for controlling, coordinating and triggering the various blocks of the Readout driver, the ROD Controller. The EFB header dynamic mask encoder worked as expected and the readout state machines in particular did not add any additional delay. The Formatter Readout Modebits Encoder also worked as expected and the token passing architecture simplified the process of reading out the data. The Front End Command Processor that generates real time command pulses and command streams did not add additional latency. The simulation test results and synthesis details are presented.

6.2 Conclusion

The readout driver prototype has been used on simulated detector data for the ATLAS Pixel Experiment. The simulation was performed by generating near real time data using the 'C' programming language. Two prototype models were designed. The Readout Driver Model depicted in this thesis is Model 2 of the prototype. Model 2 of the prototype is an extension of Model 1. Although Model 1 was not implemented real-time, it was simulated with near real-time data. The test results indicated that some changes should be made in order to overcome some trigger latency and bandwidth considerations and provide greater flexibility for routing the data to the appropriate FIFOs. Moreover, Model 1 was a data pull architecture. Apart from the data coming up link fibers, data was passed to subsequent stages by pulling/reading it out from previous stages. Studies found that token passing architecture would provide better performance. Accordingly changes in the design were made in the Event Gatherer Engine and Router design. A Dual Event Fragment Builder design was adopted in order to overcome the bandwidth problems. Though the design of the Pixel ROD has been completed, the implementation and testing of the entire ROD module has not been performed.

The functionality of the ROD controller was verified. It was simulated with a 40 MHz clock frequency and it worked well at that frequency. The trigger pulses and interrupts generated were the same as the expected results.

The functioning of the Event Fragment Builder and the Router were also same as expected. The event fragment builder stopped processing the data when none of the links

from the first formatter FPGA were active. This is exactly what the event fragment builder was expected to do. The testing of the module was performed with data generated to resemble the formatted output of the Link Formatter. The Event Fragment Builder was tested for all the 28 links of the Pixel ROD. The test results were satisfactory and met expectations.

6.3 Future Directions

The next step in the design of the RODs is the integration of various modules into the FPGAs. Once this process is completed the RODs will be tested and debugged on the board. After the completion of the implementation on board, the ROD designs will be completed for Layer 2 and the Disks. The first step towards the design of RODs is to modify the design of Formatter FPGA for the Layer 2 and the Disks. The number of links and also the size of the FIFO will have to be analyzed for each of these cases. The ROD controller designed for the Layer 1 and the B-layer will be used for the Layer 2 and Disks. A slight increase or decrease in the number of mask logic links is necessary. This will be decided after taking into account the number of links to be input. The design of the Event Fragment Builder and the Router will not be much different from the ones designed for the Layer 1 and the B-layer. However, due to the difference in input data rate and clocking, minor changes in the decoder logic may be necessary.

References

[1] ATLAS collaboration. Detector and Physics Performance Technical Design Report, CERN/LHCC/99-14 (1999).

[2] ATLAS collaboration, Technical Proposal for a general-purpose pp experiment at the Large Hadron Collider at CERN. CERN/LHCC/94-43 (1994).

[3] ATLAS collaboration, The Large Hadron Collider-conceptual design. CERN/AC/95-05 (1995).

[4] ATLAS collaboration, ATLAS DAQ, EF, LVL2 and DCS Technical Progress Report. CERN/LHCC/99-14 (1999).

[5] M.Abolins, J.Dawson, N.Ellis, Y.Ermoline, C.N.P.Gee, Ph.Farthouat, C.Schwick Specification of the LVL1/LVL2 trigger system. ATL-D-ES-0003, chap. 2 (1999).

[6] Xilinx software manual: Foundation series 3.1i software documentation

[7] ATLAS trigger-DAQ steering group, Trigger & DAQ interfaces with front-end systems: Requirement document. DAQ-No-103, Version 2.0 (1998)

[8] K. Dao, D. Fasching, O. Hayes, R. Jared, K. Marks, M. Nagel, J.M.Joseph, ATLAS SCT and pixel ROD FPGA Based Implementation. (1999).

[9] K. Dao, D. Fasching, O. Hayes, R. Jared, FPGA ROD Implementation. ATLAS-Wisconsin Group (1998)

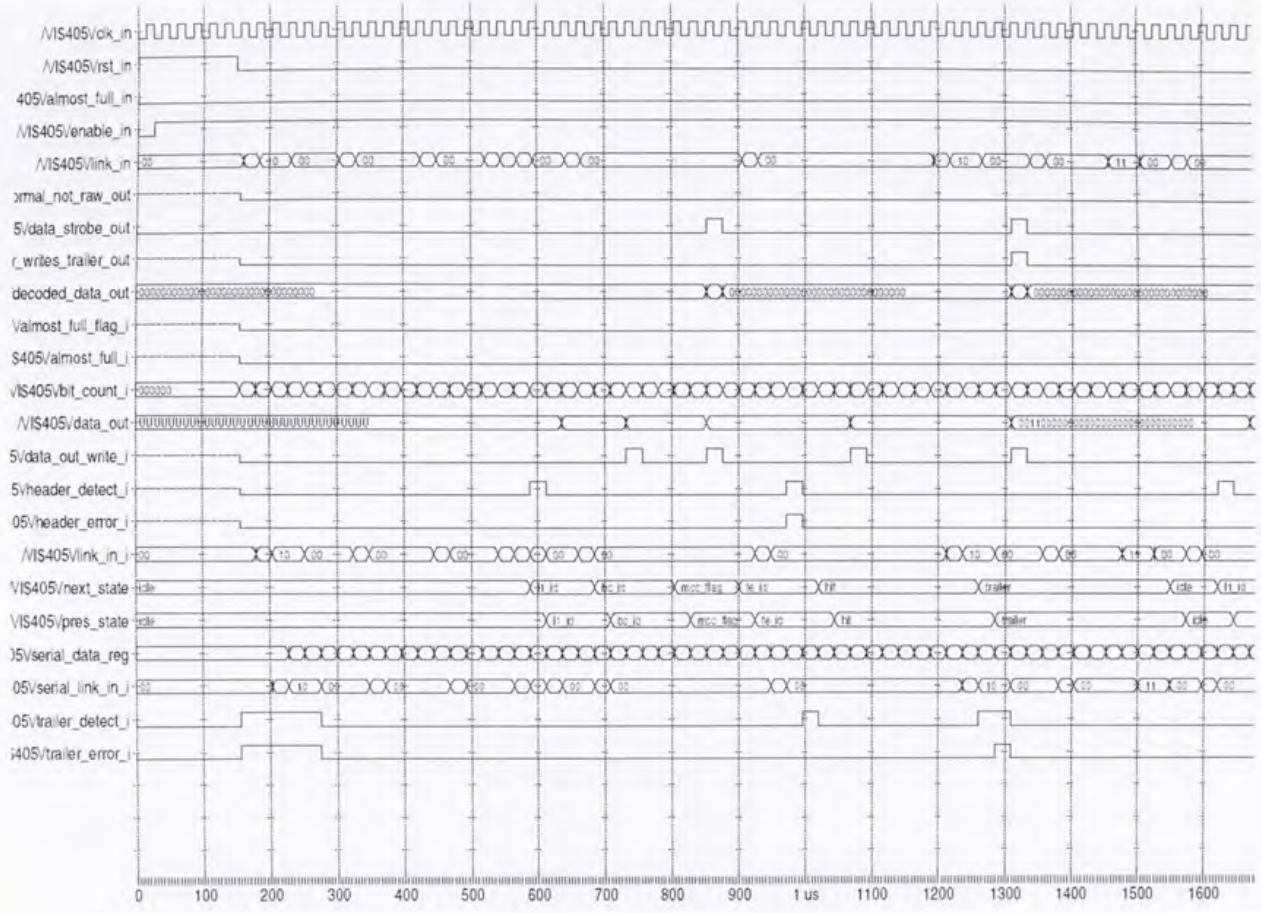
[10] ATLAS collaboration, Technical Computing Proposal. CERN/LHCC 96-43 (1996).

[11] A VHDL Primer, J.Bhaskar, 3rd Edition. Prentice Hall (1999).

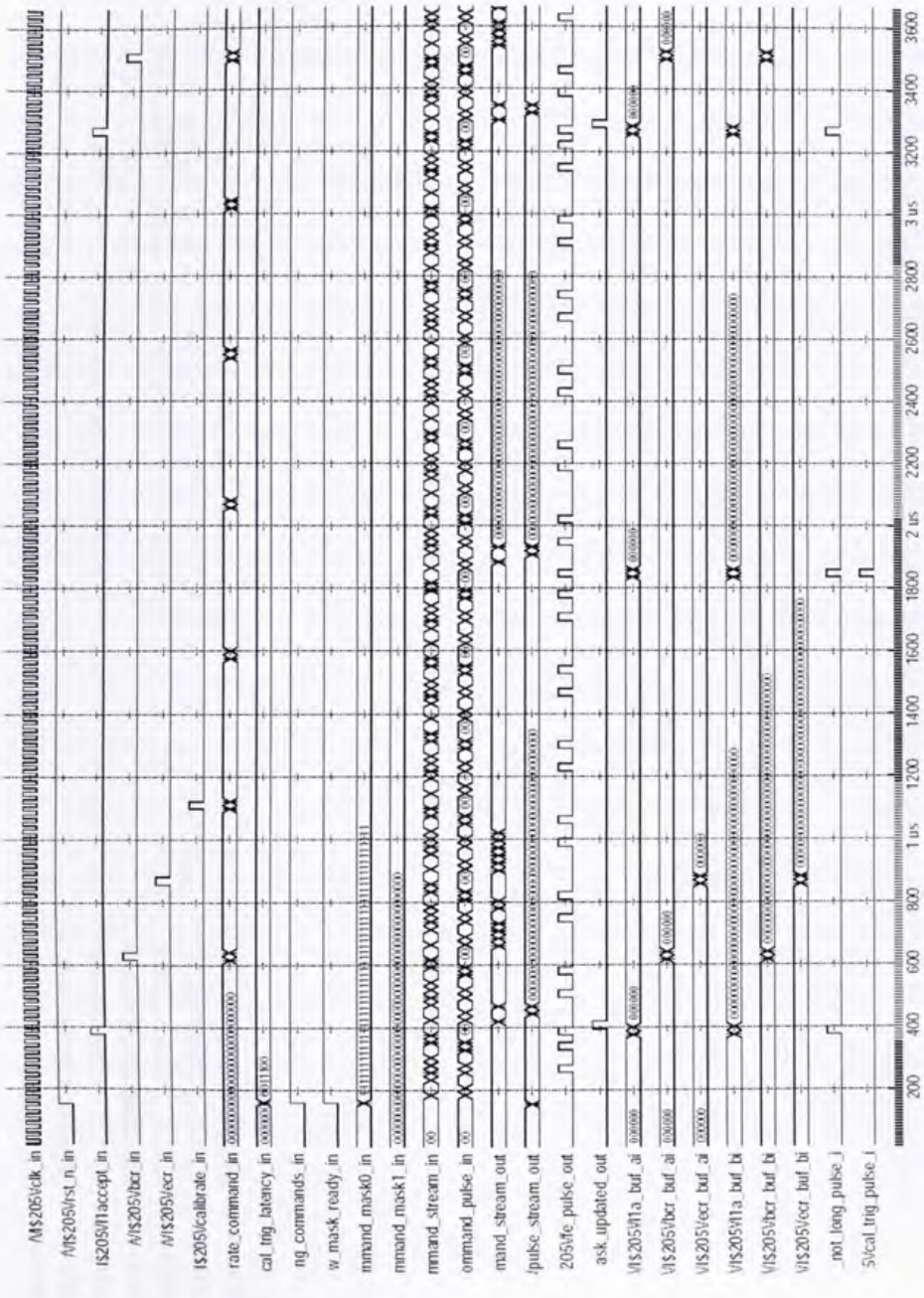
[12] A VHDL Synthesis Primer, J.Bhaskar, 2nd Edition. Prentice Hall(1999).

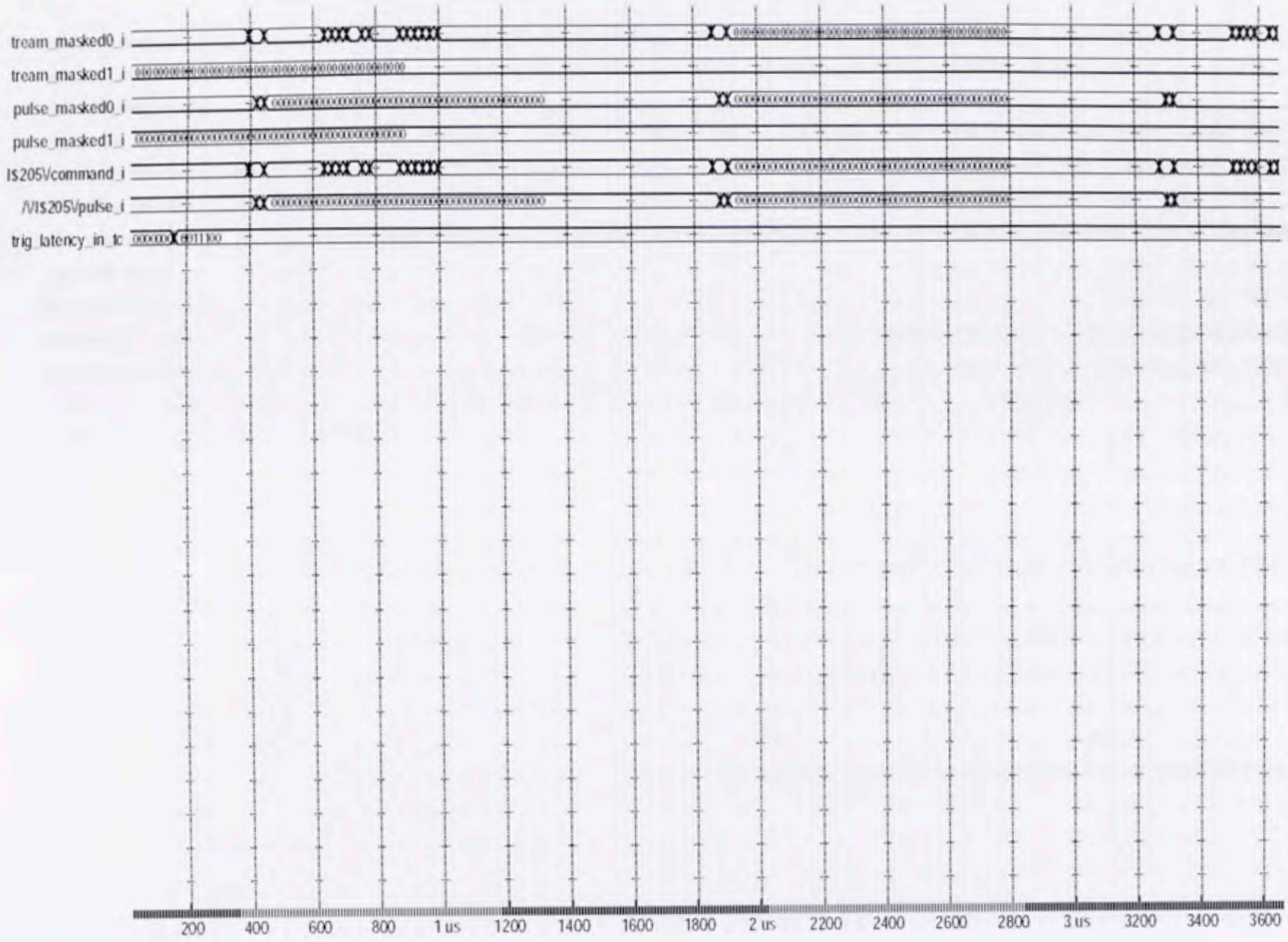
[13] "MentorGraphics," http://www.mentorgraphics.com, July 1996.

Appendix A

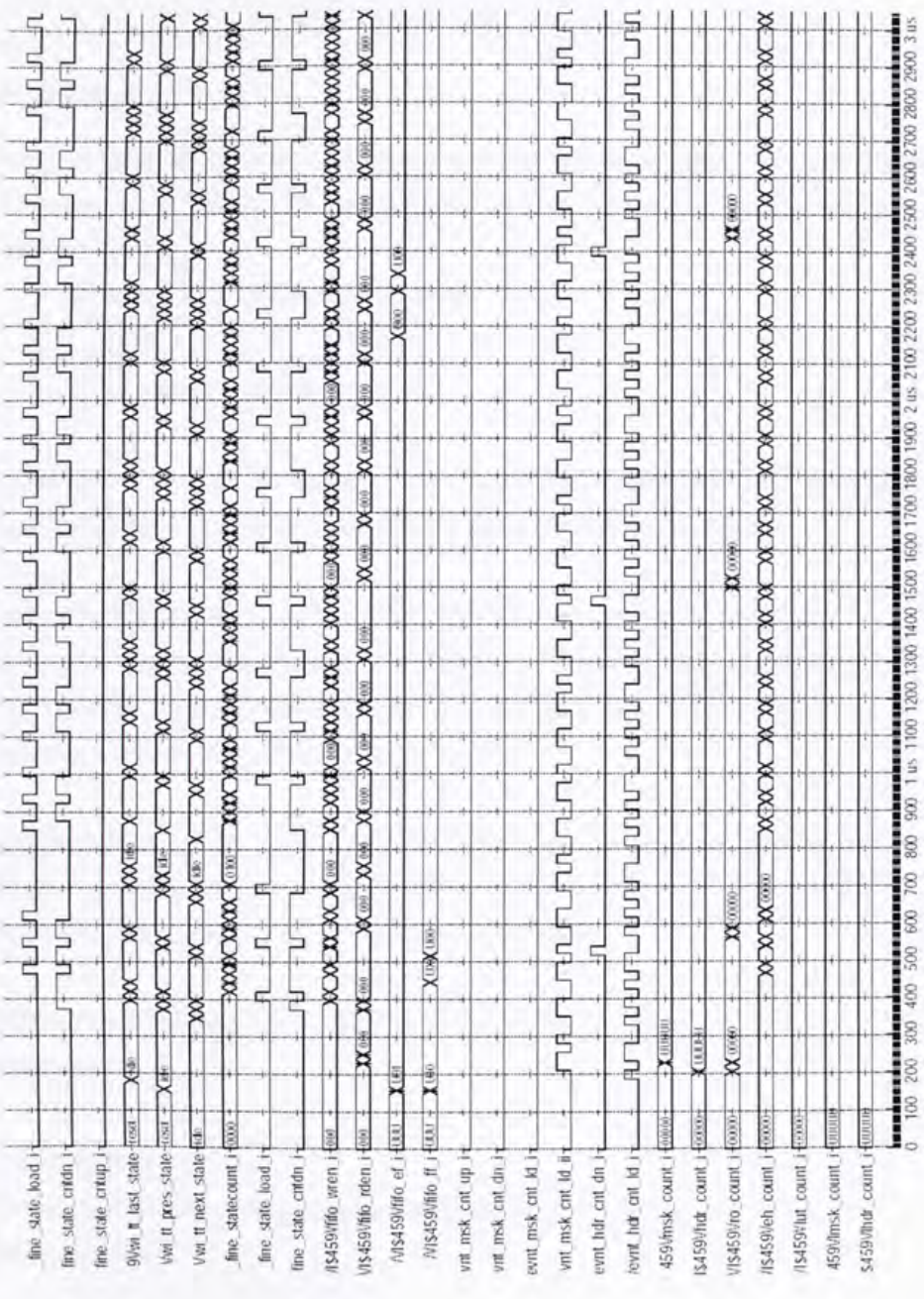

/VIS405/clk_in
/VIS405/rst_in
405/almost_full_in
/VIS405/enable_in
/VIS405/link_in
ormal_not_raw_out
5/data_strobe_out
r_writes_trailer_out
decoded_data_out
/almost_full_flag_i
S405/almost_full_i
/IS405/bit_count_i
/VIS405/data_out
5/data_out_write_i
5/header_detect_i
05/header_error_i
/VIS405/link_in_i
VIS405/next_state
VIS405/pres_state
)5/serial_data_reg
05/serial_link_in_i
05/trailer_detect_i
i405/trailer_error_i
0
100
200
300
400
500
600
700
800
900
1 us
1100
1200
1300
1400
1500
1600

Link Formatter B-Layer


/M$205/clk_in
/M$205/rst_n_in
I$205/l1accept_in
/I/I$205/bcr_in
/I/I$205/ecr_in
I$205/calibrate_in
rate_command_in
cal_trig_latency_in
ng_commands_in
w_mask_ready_in
mmand_mask0_in
mmand_mask1_in
mmand_stream_in
ommand_pulse_in
mand_stream_out
/pulse_stream_out
205/fe_pulse_out
ask_updated_out
\I$205\l1a_buf_ai
\I$205\bcr_buf_ai
\I$205\ecr_buf_ai
\I$205\l1a_buf_bi
\I$205\bcr_buf_bi
\I$205\ecr_buf_bi
_not_long_pulse_i
5\cal_trig_pulse_i
200
400
600
800
1 us
1200
1400
1600
1800
2 us
2200
2400
2600
2800
3 us
3200
3400
3600


tream_masked0_i
tream_masked1_i
pulse_masked0_i
pulse_masked1_i
I$205\/command_i
/\I$205\/pulse_i
trig_latency_in_tc
200
400
600
800
1 us
1200
1400
1600
1800
2 us
2200
2400
2600
2800
3 us
3200
3400
3600

Formatter Readout Modebits Encoder



Appendix B

CAD Tools and Libraries

A1: Mentor

Design of the ROD was done using the mentor tools. Mentor is a popular Board level tool in common use in industry. The following Mentor tools were used for schematic capture and Simulation.

- Design Architect (Schematic Capture)
- QVHCOM (VHDL Compiler)
- QHDL Pro (VHDL Simulator)
- QuicksimII (Component Simulator)

QVHCOM: To compile for synthesis with a multi block hierarchical design, a compile script should be created. This Script uses QVHCOM to compile the design.

Design Architect: The compiled VHDL codes are converted into Symbols and Schematics using the Design architect. In one tool, one gets schematic, symbol, and VHDL editors, along with everything you need for top-down hierarchical design, including block creation and hierarchy navigation.

QuicksimII: QuicksimII is a component simulator. It lets the designer use hardware models, behavioral models, compiled models and HDL models in any combination--including models from over 200 ASIC and FPGA libraries.

QHDL Pro: It performs the functional simulation of the module created using QuicksimII.

A2: Synopsis

Synopsys FPGA express was used for compiling the models developed using Mentorgraphics.

A3: Xilinx

Xilinx UNISIM Models for I/O resources, Block and Dual Port RAMs, and other features internal to the Virtex and Virtex-E FPGAs were used for implementing FIFOs RAMs . Xilinx Foundation tools version 3.1 was used for place and routing

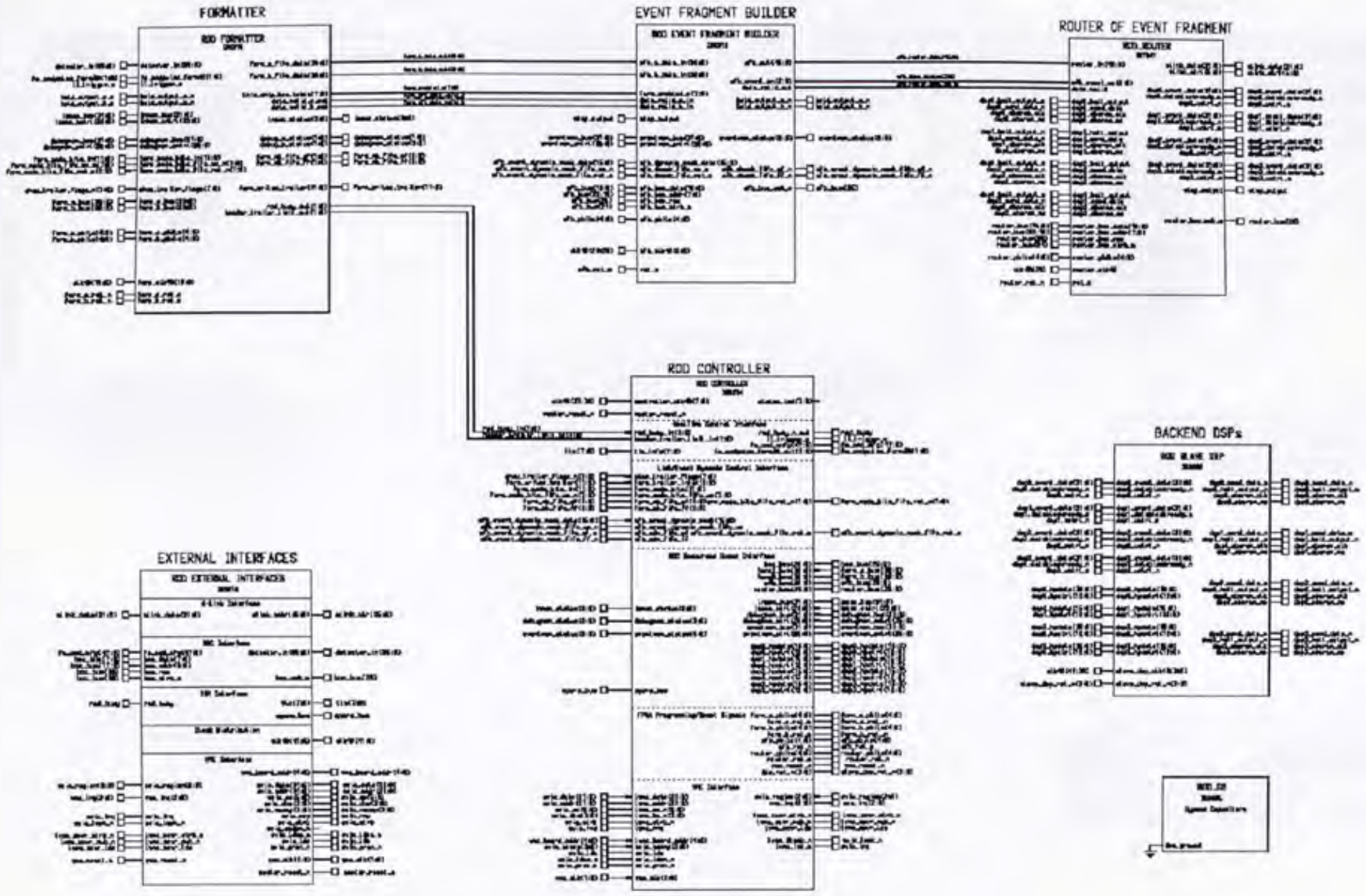
Appendix C

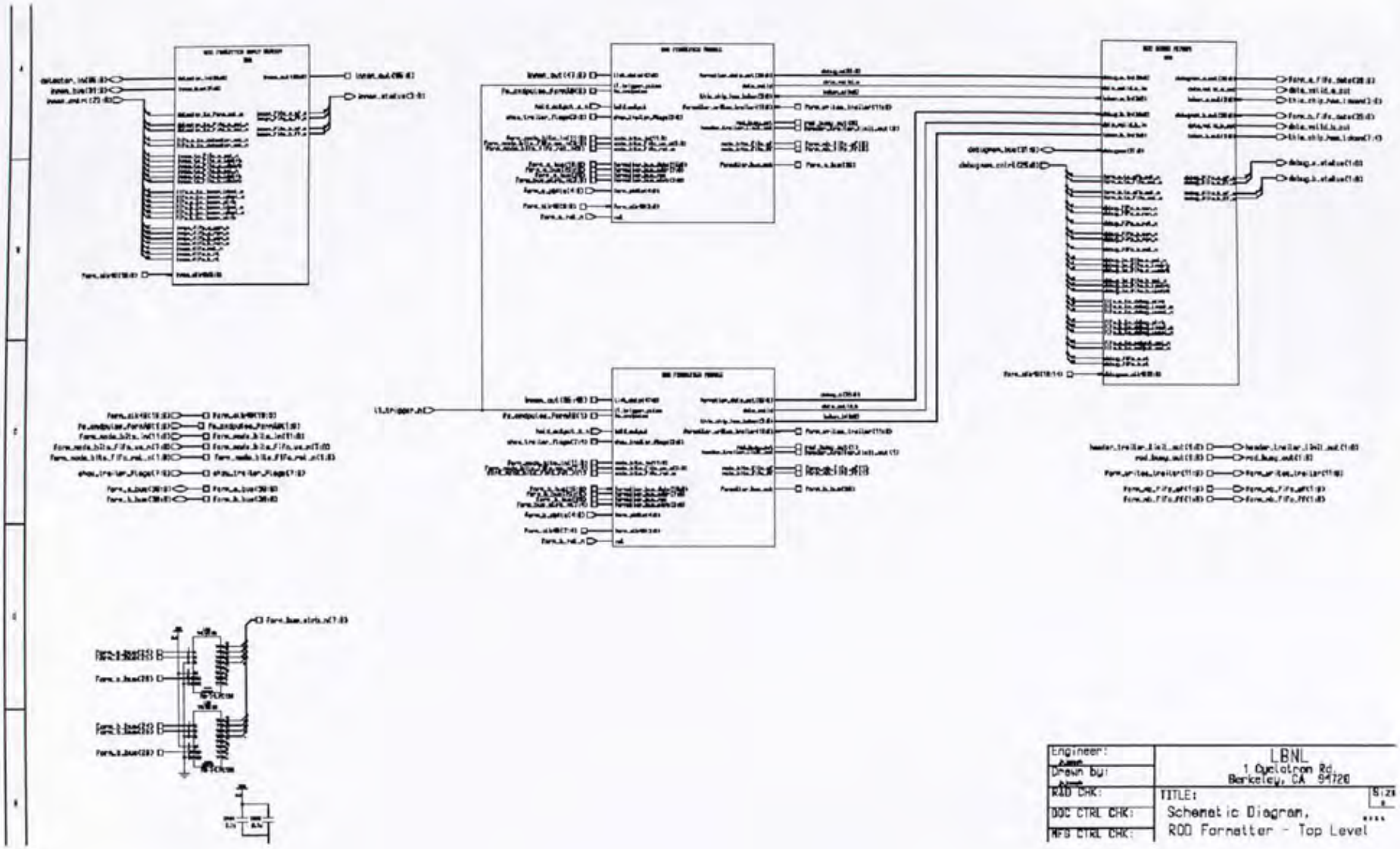
B1: Link Formatter Symbol Key

L	L1_ID
B	BC_ID
H	Header Error
T	TOT
C	Column
R	Row
M	MCC_FLAG
F	FE_ID
E	FE_FLAG
Te	Trailer Error
D	Raw_Data
K	Link Number
X	Don't Care

Appendix D

ROD Schematics


FORMATTER
EVENT FRAGMENT BUILDER
ROUTER OF EVENT FRAGMENT
ROD CONTROLLER
BACKEND DSPs
EXTERNAL INTERFACES



Engineer:	LBNL 1 Cyclotron Rd. Berkeley, CA 94720
Drawn by:	
RAD CHK:	TITLE: Schematic Diagram, ROD Formatter - Top Level
DOC CTRL CHK:	
MFG CTRL CHK:	

Appendix E

Data Sheet

Architectural Description

Virtex Array

The Virtex user-programmable gate array, shown in Figure 1, comprises two major configurable elements: configurable logic blocks (CLBs) and input/output blocks (IOBs).

- CLBs provide the functional elements for constructing logic
- IOBs provide the interface between the package pins and the CLBs

CLBs interconnect through a general routing matrix (GRM). The GRM comprises an array of routing switches located at the intersections of horizontal and vertical routing channels. Each CLB nests into a VersaBlock™ that also provides local routing resources to connect the CLB to the GRM.

The VersaRing™ I/O interface provides additional routing resources around the periphery of the device. This routing improves I/O routability and facilitates pin locking.

The Virtex architecture also includes the following circuits that connect to the GRM.

- Dedicated block memories of 4096 bits each
- Clock DLLs for clock-distribution delay compensation and clock domain control
- 3-State buffers (BUFTs) associated with each CLB that drive dedicated segmentable horizontal routing resources

Values stored in static memory cells control the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device.

Input/Output Block

The Virtex IOB, Figure 2, features SelectIO™ inputs and outputs that support a wide variety of I/O signalling standards, see Table 1.

The three IOB storage elements function either as edge-triggered D-type flip-flops or as level sensitive latches. Each IOB has a clock signal (CLK) shared by the three flip-flops and independent clock enable signals for each flip-flop.

In addition to the CLK and CE control signals, the three flip-flops share a Set/Reset (SR). For each flip-flop, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear.

The output buffer and all of the IOB control signals have independent polarity controls.

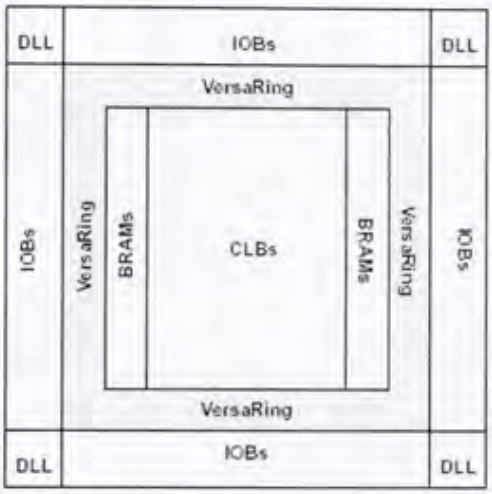


Figure 1: Virtex Architecture Overview

All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. Two forms of over-voltage protection are provided, one that permits 5 V compliance, and one that does not. For 5 V compliance, a Zener-like structure connected to ground turns on when the output rises to approximately 6.5 V. When PCI 3.3 V compliance is required, a conventional clamp diode is connected to the output supply voltage, V_{CCO}.

Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each pad. Prior to configuration, all pins not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs can optionally be pulled up.

The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration.

All Virtex IOBs support IEEE 1149.1-compatible boundary scan testing.

© 2001 Xilinx, Inc. All rights reserved. All Xilinx trademarks, registered trademarks, patents, and disclaimers are as listed at http://www.xilinx.com/legal.htm. All other trademarks and registered trademarks are the property of their respective owners. All specifications are subject to change without notice.

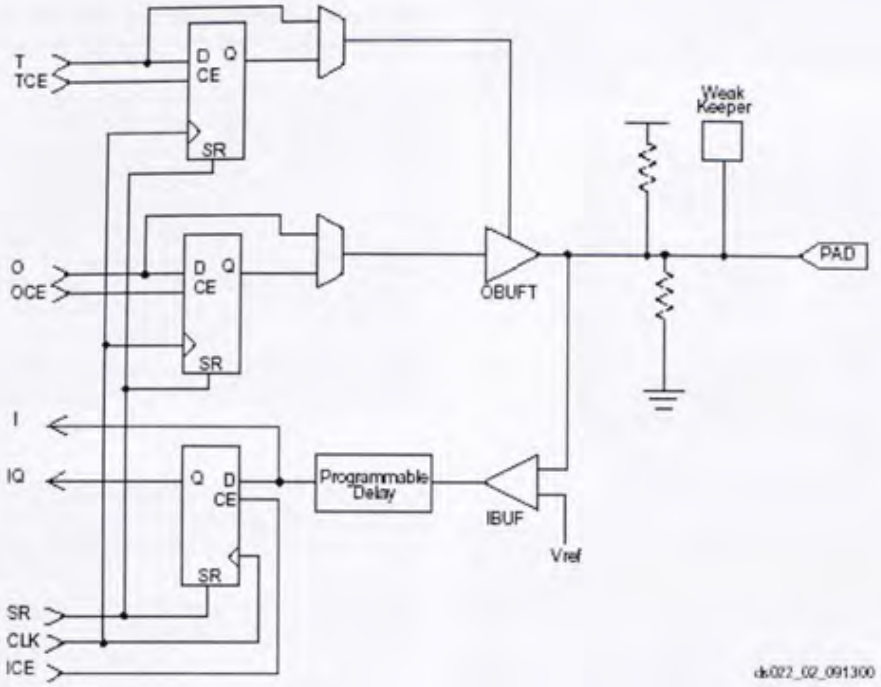


Figure 2: **Virtex Input/Output Block (IOB)**

Table 1: **Supported Select I/O Standards**

I/O Standard	Input Reference Voltage (V_{REF})	Output Source Voltage (V_{CCO})	Board Termination Voltage (V_{TT})	5 V Tolerant
LVTTL 2 – 24 mA	N/A	3.3	N/A	Yes
LVCMOS2	N/A	2.5	N/A	Yes
PCI, 5 V	N/A	3.3	N/A	Yes
PCI, 3.3 V	N/A	3.3	N/A	No
GTL	0.8	N/A	1.2	No
GTL+	1.0	N/A	1.5	No
HSTL Class I	0.75	1.5	0.75	No
HSTL Class III	0.9	1.5	1.5	No
HSTL Class IV	0.9	1.5	1.5	No
SSTL3 Class I & II	1.5	3.3	1.5	No
SSTL2 Class I & II	1.25	2.5	1.25	No
CTT	1.5	3.3	1.5	No
AGP	1.32	3.3	N/A	No

Input Path

A buffer in the Virtex IOB input path routes the input signal either directly to internal logic or through an optional input flip-flop.

An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the FPGA, and when used, assures that the pad-to-pad hold time is zero.

Each input buffer can be configured to conform to any of the low-voltage signalling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, V_{REF}. The need to supply V_{REF} imposes constraints on which standards can used in close proximity to each other. See I/O Banking, page 3.

There are optional pull-up and pull-down resistors at each input for use after configuration. Their value is in the range 50 kΩ – 100 kΩ.

Output Path

The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop.

The 3-state control of the output can also be routed directly from the internal logic or through a flip-flip that provides synchronous enable and disable.

Each output driver can be individually programmed for a wide range of low-voltage signalling standards. Each output buffer can source up to 24 mA and sink up to 48mA. Drive strength and slew rate controls minimize bus transients.

In most signalling standards, the output High voltage depends on an externally supplied V_{CCO} voltage. The need to supply V_{CCO} imposes constraints on which standards can be used in close proximity to each other. See I/O Banking, page 3.

An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the pin is connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way eliminates bus chatter.

Because the weak-keeper circuit uses the IOB input buffer to monitor the input level, an appropriate V_{REF} voltage must be provided if the signalling standard requires one. The provision of this voltage must comply with the I/O banking rules.

I/O Banking

Some of the I/O standards described above require V_{CCO} and/or V_{REF} voltages. These voltages externally and connected to device pins that serve groups of IOBs, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank.

Eight I/O banks result from separating each edge of the FPGA into two banks, as shown in Figure 3. Each bank has multiple V_{CCO} pins, all of which must be connected to the same voltage. This voltage is determined by the output standards in use.

Figure 3: Virtex I/O Banks

Within a bank, output standards can be mixed only if they use the same V_{CCO}. Compatible standards are shown in Table 2. GTL and GTL+ appear under all voltages because their open-drain outputs do not depend on V_{CCO}.

Table 2: **Compatible Output Standards**

V_{CCO}	Compatible Standards
3.3 V	PCI, LVTTL, SSTL3 I, SSTL3 II, CTT, AGP, GTL, GTL+
2.5 V	SSTL2 I, SSTL2 II, LVCMOS2, GTL, GTL+
1.5 V	HSTL I, HSTL III, HSTL IV, GTL, GTL+

Some input standards require a user-supplied threshold voltage, V_{REF}. In this case, certain user-I/O pins are automatically configured as inputs for the V_{REF} voltage. Approximately one in six of the I/O pins in the bank assume this role.

The V_{REF} pins within a bank are interconnected internally and consequently only one V_{REF} voltage can be used within each bank. All V_{REF} pins in the bank, however, must be connected to the external voltage source for correct operation.

Within a bank, inputs that require V_{REF} can be mixed with those that do not. However, only one V_{REF} voltage can be used within a bank. Input buffers that use V_{REF} are not 5 V tolerant. LVTTL, LVCMOS2, and PCI 33 MHz 5 V, are 5 V tolerant.

The V_{CCO} and V_{REF} pins for each bank appear in the device Pinout tables and diagrams. The diagrams also show the bank affiliation of each I/O.

Within a given package, the number of V_{REF} and V_{CCO} pins can vary depending on the size of device. In larger devices,

more I/O pins convert to V_{REF} pins. Since these are always a superset of the V_{REF} pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device if necessary. All the V_{REF} pins for the largest device anticipated must be connected to the V_{REF} voltage, and not used for I/O.

In smaller devices, some V_{CCO} pins used in larger devices do not connect within the package. These unconnected pins can be left unconnected externally, or can be connected to the V_{CCO} voltage to permit migration to a larger device if necessary.

In TQ144 and PQ/HQ240 packages, all V_{CCO} pins are bonded together internally, and consequently the same V_{CCO} voltage must be connected to all of them. In the CS144 package, bank pairs that share a side are interconnected internally, permitting four choices for V_{CCO}. In both cases, the V_{REF} pins remain internally connected as eight banks, and can be used as described previously.

Configurable Logic Block

The basic building block of the Virtex CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and a storage element. The output from the function generator in each LC drives both the CLB output and the D input of the flip-flop. Each Virtex CLB contains four LCs, organized in two similar slices, as shown in Figure 4.

Figure 5 shows a more detailed view of a single slice.

In addition to the four basic LCs, the Virtex CLB contains logic that combines function generators to provide functions of five or six inputs. Consequently, when estimating the number of system gates provided by a given device, each CLB counts as 4.5 LCs.

Look-Up Tables

Virtex function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16x1-bit dual-port synchronous RAM.

The Virtex LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing.

Storage Elements

The storage elements in the Virtex slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by the function generators within the slice or directly from slice inputs, bypassing the function generators.

In addition to Clock and Clock Enable signals, each Slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified for it in the configuration. BY forces it into the opposite state. Alternatively, these signals can be configured to operate asynchronously. All of the control signals are independently invertible, and are shared by the two flip-flops within the slice.

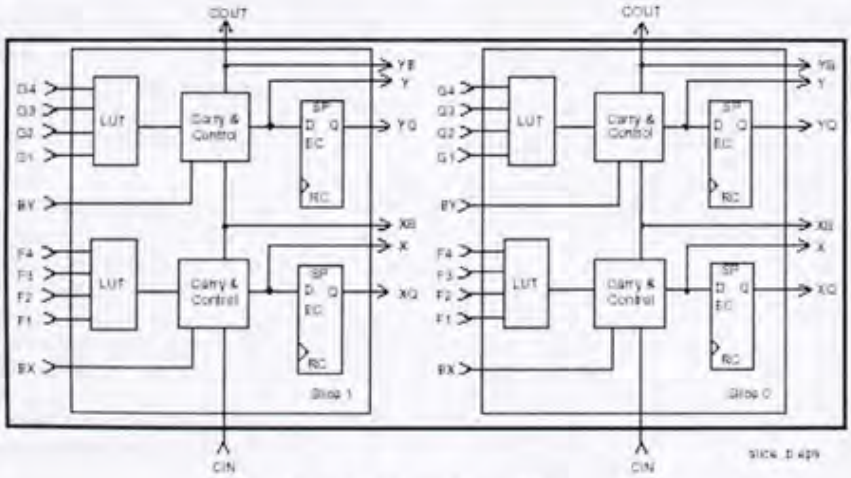


Figure 4: **2-Slice Virtex CLB**

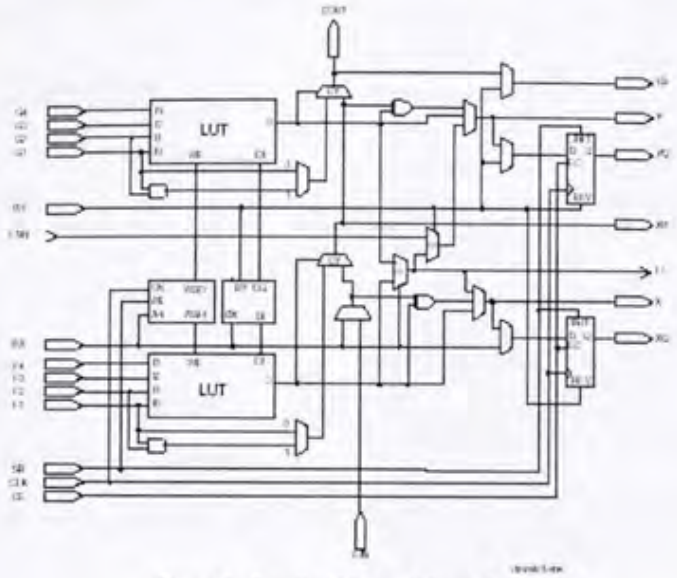


Figure 5: **Detailed View of Virtex Slice**

Additional Logic

The F5 multiplexer in each slice combines the function generator outputs. This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs.

Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs.

Each CLB has four direct feedthrough paths, one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources.

Arithmetic Logic

Dedicated carry logic provides fast arithmetic carry capability for high-speed arithmetic functions. The Virtex CLB supports two separate carry chains, one per Slice. The height of the carry chains is two bits per CLB.

The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementation.

The dedicated carry path can also be used to cascade function generators for implementing wide logic functions.

BUFTs

Each Virtex CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. See Dedicated Routing, page 7. Each Virtex BUFT has an independent 3-state control pin and an independent input pin.

Block SelectRAM

Virtex FPGAs incorporate several large Block SelectRAM memories. These complement the distributed LUT SelectRAMs that provide shallow RAM structures implemented in CLBs.

Block SelectRAM memory blocks are organized in columns. All Virtex devices contain two such columns, one along each vertical edge. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Virtex device 64 CLBs high contains 16 memory blocks per column, and a total of 32 blocks.

Table 3 shows the amount of Block SelectRAM memory that is available in each Virtex device.

Table 3: **Virtex Block SelectRAM Amounts**

Device	# of Blocks	Total Block SelectRAM Bits
XCV50	8	32,768
XCV100	10	40,960
XCV150	12	49,152
XCV200	14	57,344
XCV300	16	65,536
XCV400	20	81,920
XCV600	24	98,304
XCV800	28	114,688
XCV1000	32	131,072

Each Block SelectRAM cell, as illustrated in Figure 6, is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion.

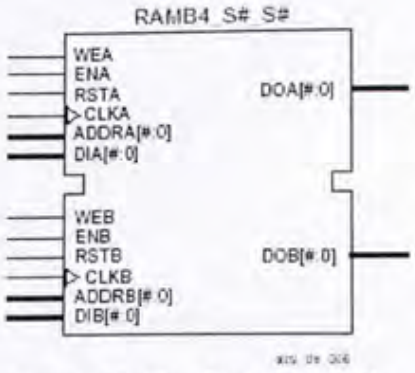


Figure 6: **Dual-Port Block SelectRAM**

Table 4 shows the depth and width aspect ratios for the Block SelectRAM.

Table 4: **Block SelectRAM Port Aspect Ratios**

Width	Depth	ADDR Bus	Data Bus
1	4096	ADDR<11:0>	DATA<0>
2	2048	ADDR<10:0>	DATA<1:0>
4	1024	ADDR<9:0>	DATA<3:0>
8	512	ADDR<8:0>	DATA<7:0>
16	256	ADDR<7:0>	DATA<15:0>

The Virtex Block SelectRAM also includes dedicated routing to provide an efficient interface with both CLBs and other Block SelectRAMs.

Programmable Routing Matrix

It is the longest delay path that limits the speed of any worst-case design. Consequently, the Virtex routing architecture and its place-and-route software were defined in a single optimization process. This joint optimization minimizes long-path delays, and consequently, yields the best system performance.

The joint optimization also reduces design compilation times because the architecture is software-friendly. Design cycles are correspondingly reduced due to shorter design iteration times.

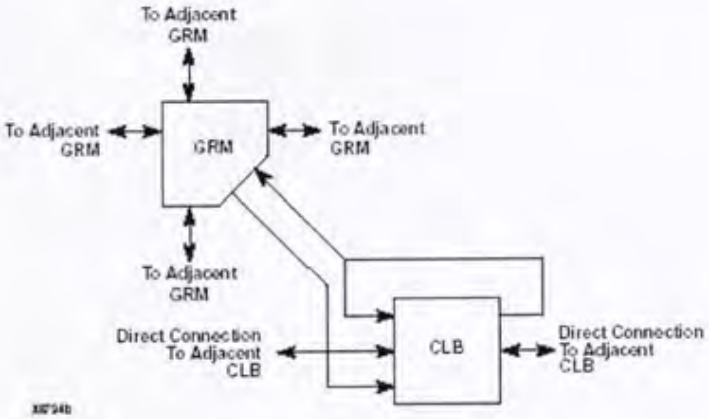


Figure 7: **Virtex Local Routing**

Local Routing

The VersaBlock provides local routing resources, as shown in Figure 7, providing the following three types of connections.

- Interconnections among the LUTs, flip-flops, and GRM
- Internal CLB feedback paths that provide high-speed connections to LUTs within the same CLB, chaining them together with minimal routing delay
- Direct paths that provide high-speed connections between horizontally adjacent CLBs, eliminating the delay of the GRM.

Appendix F

VHDL CODE

Link_formatter_layer1.vhd

```
--------------------------------------------------------------------------------
-- University of Oklahoma and Lawrence Berkeley National Labs
-- ATLAS ROD ELECTRONICS
-- Author -- Sriram.S
--------------------------------------------------------------------------------
-- Filename:
-- formatter_pxl.vhd
-- Description:
-- ROD Front End Decoder
-- Double Channel
-- PIXEL Version
--
-- This file contains the serial decoding state machine for the PIXEL front
-- end. You should refer to three documents to better understand how the
-- vhdl for the decoder works.
-- 1) "Format of Data from the PIXEL Front End Decoders" describes what
-- the output data words look like after the serial bitstreams have
-- been decoded and converted from serial to parallel.
-- /home/edafiles/atlas/rod/docs/pix_fe_format.txt(or .ps)
--
-- 2) "MCC Event Structure" presents a state transition diagram
-- that shows how to decode the MCC serial data. The VHDL logic
-- for the PIXEL decoder is closely based on this diagram.
-- /home/edafiles/atlas/rod/docs/mcc_format.ps

entity link_formatter_layer1 is
port (
clk_in : in std_logic;
rst_in : in std_logic;
almost_full_in : in std_logic; -- 1 = header/trailer
-- 0 = normal
enable_in : in std_logic; -- enable formatter activity
link_in : in std_logic_vector(1 downto 0); -- front end serial data
normal_not_raw_out : out std_logic; -- decoder is in 1= normal data mode, 0 = raw data mode
data_strobe_out : out std_logic; -- indicates a new output word is available
decoder_writes_trailer_out : out std_logic; -- trailer being written
decoded_data_out : out std_logic_vector(31 downto 0) -- output data word
);
end link_formatter_layer1 ;
architecture link_formatter_layer1_arch of link_formatter_layer1 is
type pixel_state_typedef is(IDLE, L1_ID, BC_ID, MCC_FLAG, FE_ID, FE_FLAG, HIT, TRAILER,
RAW_DATA);
signal almost_full_flag_i : std_logic ;
signal almost_full_i : std_logic ;
signal bit_count_i : unsigned (5 downto 0);
signal bit_count_reset_i : std_logic ;
signal data_out : std_logic_vector(31 downto 0) ;
signal data_out_temp : std_logic_vector(31 downto 0) ;
signal data_out_write_i : std_logic ;
signal data_strobe_lower_i : std_logic ;
signal decoder_writes_trailer_i : std_logic ;
signal fe_reg : std_logic_vector(3 downto 0) ;
signal header_6bit : std_logic ;
```

```
signal header_detect_i : std_logic ;
signal header_error_i : std_logic ;
signal header_error_del1_i : std_logic;
signal header_sh_reg : std_logic_vector(5 downto 0) ;
signal hit_pattern_invalid : std_logic ;
signal link_in_i : std_logic_vector(1 downto 0) ;
signal next_bit_count_i : unsigned ( 5 downto 0);
signal next_bit_count_ii : unsigned ( 5 downto 0);
signal next_state : pixel_state_typedef; -- pixel decoder fsm states
signal new_hit_data_i : std_logic ;
signal new_hit_location_i : std_logic ;
signal normal_not_raw_i : std_logic ;
signal raw_data_odd_i : std_logic ;
signal raw_data_even_i : std_logic ;
signal pres_state : pixel_state_typedef; -- pixel decoder fsm states
signal serial_data_reg : std_logic_vector(27 downto 0);
signal serial_link_in_i : std_logic_vector(1 downto 0) ;
signal trailer_counter_i : unsigned (4 downto 0);
signal trailer_detect_i : std_logic;
signal trailer_error_i : std_logic ;
signal trailer_error_del1_i : std_logic ;
signal trailer_error_flag_i : std_logic ;
signal trailer_zeroes : std_logic ;
-------------------------------------------------------------------------------------------------------------
-- Constant Declaration
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------
-- Componenent Declaration
-------------------------------------------------------------------------------------------------------------
begin -- Main body of the vhdl code
-------------------------------------------------------------------------------------------------------------
-- Component Instantiation
-------------------------------------------------------------------------------------------------------------
input_register : process (clk_in, rst_in)
begin
if (rst_in = '1') then
serial_data_reg(27 downto 0) <= "0000000000000000000000000000";
serial_link_in_i <= (others => '0');
link_in_i <= (others => '0');
elsif (clk_in'event and clk_in = '1') then
almost_full_i <= almost_full_in;
link_in_i <= link_in;
serial_link_in_i(1) <= link_in_i(1) and enable_in;
serial_link_in_i(0) <= link_in_i(0) and enable_in;
serial_data_reg(27 downto 0) <= serial_data_reg(25 downto 0) & serial_link_in_i(1) & serial_link_in_i(0);
normal_not_raw_out <= normal_not_raw_i;
end if;
end process input_register;
-----
find_header : process(clk_in, rst_in)
begin -- find header
if (rst_in = '1') then
header_sh_reg(5 downto 0) <= "000000";
elsif (clk_in'event and clk_in = '1') then
-- correct header pattern is 11101.A header can also have upto one bit of error
-- in any location.
```

```
header_6bit <= '0';
header_sh_reg(5 downto 0) <= header_sh_reg(3 downto 0) & serial_data_reg(23) & serial_data_reg(22);
case header_sh_reg(5 downto 1) is
when "11101" => -- "111101 is a possible hit pattern with an error
header_detect_i <= '1';
header_error_i <= header_sh_reg(0);
header_6bit <= header_sh_reg(0);
when "01101" =>
header_detect_i <= '1';
header_error_i <= '1';
when "10101" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11001" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11111" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11100" =>
header_detect_i <= '1';
header_error_i <= '1';
when "11010" =>
header_detect_i <= header_sh_reg(0); -- "111010" is a possible hit pattern with an error
header_error_i <= header_sh_reg(0);
header_6bit <= header_sh_reg(0);
when others =>
header_detect_i <= '0';
header_error_i <= '0';
end case;
end if;
end process find_header;
------
find_trailer : process(clk_in, rst_in) -- no errors allowed
begin -- find trailer
if (rst_in = '1') then
trailer_counter_i <= "00000";
trailer_detect_i <= '0';
trailer_error_i <= '0';
trailer_zeroes <= '0';
elsif (clk_in'event and clk_in = '1') then
if(serial_data_reg(21) = '0'and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+2;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i+1;
```

```
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-1;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i-2;
elsif (serial_data_reg(21) = '1' and serial_data_reg(20) = '1' and serial_data_reg(0) = '1'
and serial_data_reg(1) = '1') then
trailer_counter_i <= trailer_counter_i;
elsif (serial_data_reg(21) = '0' and serial_data_reg(20) = '0' and serial_data_reg(0) = '0'
and serial_data_reg(1) = '0') then
trailer_counter_i <= trailer_counter_i;
end if;
if (std_logic_vector(trailer_counter_i) = "00000") then
trailer_detect_i <= '1'; -- if 0 or 1 followed by 21 0's a trailer is detected
trailer_error_i <= not serial_data_reg(23); -- zero followed by 21 0's is a trailer with a bit error
trailer_zeroes <= '1'; -- trailer with 1 or 0 followed by 21 zeroes
else
trailer_detect_i <= '0';
trailer_error_i <= '0';
trailer_zeroes <= '0';
end if;
end if;
end process find_trailer;
-----
--------------------------------------------------------------------------------
-- the next process is all combinatorial logic and performs which state
-- transitions are to take place
--------------------------------------------------------------------------------
-- purpose: default FSM values
-- type : combinational
-- inputs : pres_state,bit_count_i,header_detect_i,serial_data_reg,trailer_error_i,
-- trailer_detect_i,data_out_write_i
--outputs: next_state,next_bit_count_i,trailer_error_flag_i
```

```
pxl_comb_fsm : process (pres_state,trailer_zeroes, bit_count_i, header_detect_i, serial_data_reg,
trailer_error_i, trail
er_detect_i, data_out_write_i,decoder_writes_trailer_i)
begin -- process pxl_comb_fsm
next_bit_count_i <= bit_count_i + "000010";
trailer_error_flag_i <= '0';
-- check all state transition conditions
if(trailer_detect_i = '1' and trailer_error_i = '0' and pres_state/= idle and pres_state/= l1_id and
pres_state/= fe_i
d and
pres_state/= trailer) then
next_state <= trailer;
trailer_error_flag_i <= '1';
next_bit_count_i <= "000000";
end if;
case pres_state is
when idle =>
if (std_logic_vector(bit_count_i) = "100110" and header_detect_i = '1') then
next_state <= l1_id;
next_bit_count_i <= "000011";
elsif (std_logic_vector(bit_count_i) = "000010" and header_detect_i = '1')then
next_state <= l1_id;
next_bit_count_i <= "000011";
elsif (std_logic_vector(bit_count_i) = "000100" and header_detect_i = '1') then
next_state <= l1_id;
next_bit_count_i <= "000011";
end if;
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
normal_not_raw_i <= '1';
when l1_id =>
if (std_logic_vector(bit_count_i) = "001001")then
if (serial_data_reg(26) = '1') then
next_state <= bc_id;
next_bit_count_i <= "000000";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
end if;
end if;
normal_not_raw_i <= '1';
when bc_id =>
if (std_logic_vector(bit_count_i) = "001000") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif(serial_data_reg(27) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000011";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(24) = '0') then
next_state <= mcc_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(24) = '1') then
next_state <= fe_id;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when mcc_flag =>
if (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000010";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000010";
fe_reg(3 downto 0) <= serial_data_reg(21 downto 18);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when fe_id =>
if (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0')then
next_state <= raw_data;
next_bit_count_i <= "000010";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (serial_data_reg(25 downto 22) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000010";
raw_data_even_i <= '0';
```

```
raw_data_odd_i <= '0';
end if;
end if;
elsif(std_logic_vector(bit_count_i) = "001010") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if (serial_data_reg(26 downto 23) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when hit =>
if (std_logic_vector(bit_count_i) = "010110") then
if (serial_data_reg(26) = '0') then
if (serial_data_reg(27) = '1' and trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000000";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
end if;
elsif (serial_data_reg(26) = '1') then
if(trailer_detect_i = '1')then
next_state <= trailer;
next_bit_count_i <= "000010";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(25 downto 22) = "1110")then
next_state <= fe_id;
next_bit_count_i <= "000010";
new_hit_location_i <= '1';
fe_reg(3 downto 0) <= serial_data_reg(23 downto 20);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(25 downto 22) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000010";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= hit;
next_bit_count_i <= "000010";
new_hit_data_i <= '1';
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
elsif (std_logic_vector(bit_count_i) = "010111") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000011";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if(trailer_detect_i = '1')then
next_state <= trailer;
next_bit_count_i <= "000011";
trailer_error_flag_i <= trailer_error_i;
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif(serial_data_reg(26 downto 23) = "1110")then
next_state <= fe_id;
next_bit_count_i <= "000011";
new_hit_location_i <= '1';
fe_reg(3 downto 0) <= serial_data_reg(22 downto 19);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26 downto 23) = "1111") then
next_state <= fe_flag;
next_bit_count_i <= "000011";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
elsif (serial_data_reg(25) = '0') then
next_state <= hit;
next_bit_count_i <= "000011";
new_hit_data_i <= '1';
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when fe_flag =>
if (std_logic_vector(bit_count_i) = "001010") then
if (serial_data_reg(27) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000001";
raw_data_odd_i <= '1';
raw_data_even_i <= '0';
elsif (serial_data_reg(27) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000001";
```

```
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000001";
fe_reg <= serial_data_reg(24 downto 21);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
elsif (std_logic_vector(bit_count_i) = "001001") then
if (serial_data_reg(26) = '0') then
next_state <= raw_data;
next_bit_count_i <= "000000";
raw_data_even_i <= '1';
raw_data_odd_i <= '0';
elsif (serial_data_reg(26) = '1') then
if (trailer_detect_i = '1') then
next_state <= trailer;
next_bit_count_i <= "000000";
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
else
next_state <= fe_id;
next_bit_count_i <= "000010";
fe_reg(3 downto 0) <= serial_data_reg(23 downto 20);
raw_data_even_i <= '0';
raw_data_odd_i <= '0';
end if;
end if;
end if;
normal_not_raw_i <= '1';
when trailer =>
if (std_logic_vector(bit_count_i) = "010110") then
next_state <= idle;
next_bit_count_i <= "000000";
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
elsif (std_logic_vector(bit_count_i) = "010111") then
next_state <= idle;
next_bit_count_i <= "000000";
raw_data_odd_i <= '0';
raw_data_even_i <= '0';
end if;
normal_not_raw_i <= '1';
when raw_data =>
if ((trailer_detect_i = '1' and trailer_error_i = '0')or trailer_zeroes = '1') then
next_state <= trailer;
trailer_error_flag_i <= trailer_error_i;
else
next_state <= raw_data;
end if;
when others => null;
end case;
normal_not_raw_i <= '0';
end process pxl_comb_fsm;
```

```
-- purpose: Flip flops holding the current state values are set here
-- type : combinational
-- inputs : clk_in,rst_in,next_state,bit_count_i
-- outputs: pres_state,bit_count_i
-------------------------------------------------------------------------------
set_clocked_fsm : process (clk_in, rst_in)
begin -- process set_clocked_fsm
if (rst_in = '1') then
pres_state <= idle;
bit_count_i <= "000000";
elsif (clk_in'event and clk_in = '1') then
pres_state <= next_state;
bit_count_i <= next_bit_count_i;
if (std_logic_vector(next_bit_count_i) = "000000") then
bit_count_reset_i <= '1';
else
bit_count_reset_i <= '0';
end if;
end if;
end process set_clocked_fsm;
-- purpose: This process looks at the current state and outputs the data
--words depending on which state and which bit is being shifted in
-- type : combinational
-- inputs : clk_in,rst_in
-- outputs: header_error_del1,data_out_write_i,decoder_writes_trailer_i,normal_not_raw,
process (clk_in, rst_in)
begin -- process
if (rst_in = '1') then
elsif (clk_in'event and clk_in = '1') then
header_error_del1_i <= header_error_i;
trailer_error_del1_i <= trailer_error_i;
data_out_write_i <= '0';
decoder_writes_trailer_i <= '0';
if (almost_full_i = '1') then
almost_full_flag_i <= '1';
end if;
case pres_state is
when idle =>
almost_full_flag_i <= '0';


when l1_id =>
if (std_logic_vector(bit_count_i) = "000011") then
data_out(31 downto 28) <= "000" & header_error_i;
data_out(15 downto 8) <= header_sh_reg(4) & serial_data_reg(27 downto 21);
end if;
when bc_id =>
if std_logic_vector(bit_count_i) = "000000" then
data_out_write_i <= '1';
data_out(7 downto 0) <= serial_data_reg(27 downto 20);
end if;
when mcc_flag =>
if (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "001";
data_out(7 downto 0) <= header_sh_reg(4) & serial_data_reg(27 downto 21);
```

```
end if;
when hit =>
if (std_logic_vector(bit_count_i) = "000010" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 30) <= "11";
data_out(29 downto 0) <= "00000" & fe_reg(3 downto 0) & serial_data_reg(27 downto 7);
elsif (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 30) <= "11";
data_out(29 downto 0) <= "00000" & fe_reg(3 downto 0) & header_sh_reg(4)
& serial_data_reg(27 downto 8);
end if;
when fe_flag =>
if (std_logic_vector(bit_count_i) = "000010" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "100";
data_out(11 downto 0) <= fe_reg(3 downto 0) & serial_data_reg(27 downto 20);
elsif (std_logic_vector(bit_count_i) = "000011" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 29) <= "100";
data_out(11 downto 0) <= fe_reg(3 downto 0) &header_sh_reg(4) & serial_data_reg(27 downto 21);
end if;
when trailer =>
if (std_logic_vector(bit_count_i) = "000000" and almost_full_flag_i = '0') then
data_out_write_i <= '1';
data_out(31 downto 23) <= "001" & trailer_error_i & "00000" ;
decoder_writes_trailer_i <= '1';
end if;
when raw_data =>
if raw_data_even_i = '1' then
data_out <= "01" & data_out(27 downto 0) & serial_data_reg(27 downto 26);
elsif raw_data_odd_i = '1' then
data_out <= "01" & data_out(27 downto 0) & serial_data_reg(26 downto 25);
end if;
if (((next_state/= raw_data) or(std_logic_vector(bit_count_i) = "000000"))
and almost_full_flag_i = '0')then
data_out_write_i <= '1';
end if;
when others => null;
end case;
end if;
end process;
process (clk_in, rst_in, decoder_writes_trailer_i)
begin -- process
if (rst_in = '1') then
data_strobe_lower_i <= '0';
elsif (clk_in'event and clk_in = '1') then
if (data_out_write_i = '1') then
if (decoder_writes_trailer_i = '1') then
data_strobe_lower_i <= '0';
else
data_strobe_lower_i <= not data_strobe_lower_i;
end if;
end if;
end if;
decoder_writes_trailer_out <= decoder_writes_trailer_i;
```

```
end process;
-- purpose: to write the output signals
process (data_out_write_i, data_strobe_lower_i, decoder_writes_trailer_i, data_out)
begin -- process
data_strobe_out <= '0';
decoded_data_out <= (others => '0');
if data_out_write_i = '1' then
if data_strobe_lower_i = '1' then
data_strobe_out <= '1';
decoded_data_out <= data_out;
-- normal_not_raw_out <= normal_not_raw_i;
end if;
end if;
end process;
end link_formatter_layer1_arch;
```