The Design and Implementation of the TAXI Fiber Optic Interface Support Module

by

Stanley C. Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

© Stanley C. Wang, MCMXCV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author ............................................................... Department of Electrical Engineering and Computer Science

January 25, 1995

Certified by .................................................... James L. Kirtley, Jr.
Professor of Electrical Engineering
Thesis Supervisor

Accepted by ....................................................... F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses
The Design and Implementation of the TAXI Fiber Optic Interface Support Module

by

Stanley C. Wang

Submitted to the Department of Electrical Engineering and Computer Science on January 25, 1995, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

The TAXI Fiber Optic Interface Support Module (TAXI-ISM) is a printed circuit board (PCB) that is part of a larger system called the Airborne Seeker Test Bed Data Multiplexer II, developed by MIT Lincoln Laboratory Group 105. The primary purpose of the TAXI-ISM is to implement the interfaces between the Data Multiplexer II and all fiber optic sensors. The TAXI-ISM can receive up to two 15 MByte/sec fiber optic channels. Data is received in the form of serial fiber optic bytes by the TAXI-ISM, and sequenced into 16-Bit words. After the serial bytes have been sequenced into words, the TAXI-ISM processes the data streams, logging all status and error conditions. It then arbitrates between the two channels and transmits a single data stream to the Data Multiplexer II. This data is sent in the form of parallel words to from a 30 MByte/sec data stream.

Thesis Supervisor: James L. Kirtley, Jr.
Title: Professor of Electrical Engineering
Acknowledgments

First of all, I would like to acknowledge MIT Lincoln Laboratory for giving me this opportunity and for allowing me to actually use this project as my thesis. I would like to thank both of my advisors at Lincoln, Greg Zancewicz and Nicholas Spencer, for all their help, supervision, and time.

I would like to thank Professor James L. Kirtley, Jr. of MIT for taking the time to advise the actual writing of the thesis document. He was one of the only professors at MIT who was actually willing to give up the time to supervise the thesis.

Thank you also to Anne Hunter, the Course VI Secretary who knows everything there is to know about Course VI. Thank you Anne for patiently answering all of my questions and dealing with my worrying about administrative stuff for my five years at MIT.

I would also like to thank the people at TEK Microsystems, Incorporated for all of their advice and support.

I would like to thank Chris Casey for his help with the artwork for the final printed circuit board. I had no idea how to even start the artwork, and Chris showed me how to do it step by step. I would also like to acknowledge his help in laying the foundation for the serial ID controller.

I would like to thank Andy Reddig for all of his support and advice. You’ve been great Andy.

I would like to acknowledge the work of Bradley Lichtenstein, a fellow student at MIT, for his help with the sixteen bit PRN generator for the BIT function.

Most of all, I would like to thank my family. I owe a great deal to my parents Christine Wang and Meng-chian Wang. I couldn’t have done it without all of their support through Kindergarten, elementary school, high school, and finally MIT. And of course, my sister Stephanie. This thesis is dedicated to my family.
Contents

1 Introduction .................................................. 11
  1.1 Motivations for Development ............................. 13
  1.2 Functional Overview ...................................... 15
  1.3 Block Diagram ............................................. 16
  1.4 Subsystem Layout .......................................... 17
  1.5 System Input/Outputs ...................................... 19
  1.6 EPLD File Tutorial ........................................ 19

2 Control Module .............................................. 21
  2.1 Hardware Implementation ................................... 23
  2.2 Control Register Memory Map ............................... 24
  2.3 Control Register Descriptions ............................. 26
    2.3.1 PCB Identification Control/Status Register .......... 26
    2.3.2 PCB Identification Data Register ...................... 26
    2.3.3 Control Register ......................................... 27
    2.3.4 Error Register ........................................... 27
    2.3.5 Built-in Test Control Register ......................... 28
    2.3.6 Built-in Test Base Pattern Register .................. 28
    2.3.7 EPLD Identification Registers ......................... 29
    2.3.8 Channel Control Register ............................... 29
    2.3.9 Channel Status Register ................................ 30
    2.3.10 Channel Frame Synchronization Byte Registers ..... 31
    2.3.11 Channel Receiver Error Register ..................... 31
## List of Figures

1-1 Block Diagram of a TAXI-ISM/GBC Interface Module ........ 15
1-2 Block Diagram of the Entire Data Multiplexer II System ....... 17
1-3 Block Diagram of the TAXI-ISM .................................. 18

2-1 Block Diagram of the Control Module ......................... 23
2-2 Control Register Write Interface Timing Specifications ........ 33
2-3 State Diagram of the Control Write FSM ...................... 35
2-4 Control Register Read Interface Timing Specifications ........ 40
2-5 Control Read FSM State Diagram ............................... 41
2-6 State Diagram of the Error Control Register FSM ............ 47
2-7 State Diagram of the Status Register FSM .................... 49
2-8 State Diagram of the ID_CLK CONTROLLER FSM ............ 50

3-1 Block Diagram of the Receiver Module ....................... 57
3-2 Receiver Module Frame Synchronizer FSM diagram ............ 63

4-1 Block Diagram of the Data Module ............................ 67
4-2 Data Module Interface Write Timing ........................... 69
4-3 Data Module Arbiter FSM State Diagram ..................... 73
4-4 Implementation of the Built-in Test Function ................. 83
4-5 State Diagram of the DUTYFSM Finite State Machine ........ 89
4-6 State Diagram of the DATAFSM Finite State Machine ........ 90

B-1 Channel 1 Receiver Module Schematics ...................... 103
B-2 Channel 2 Receiver Module Schematics ...................... 104
B-3 Control and Data Module Schematics .......................... 105
B-4 Input/Output P1 Connector Schematics .......................... 106
B-5 Decoupling and Miscellaneous Schematics ...................... 107

D-1 Control Module EPLD (13021) Timing Diagram ................. 178
D-2 BIT EPLD (13022) Control Register Timing Diagram .......... 180
D-3 Receiver Module EPLD (12847) Timing Diagram ............... 182
D-4 Arbiter EPLD (12848) Timing Diagram ......................... 184
D-5 BIT EPLD (13022) Operation Timing Diagram ................. 186
List of Tables

2.1 Control Module Inputs and Outputs .................................. 22
4.1 Data Module Inputs and Outputs ...................................... 68
4.2 Data Module Arbiter FSM State Definitions ......................... 72
4.3 Pseudo-random Noise BIT Pattern Generator Algorithm ............ 87
A.1 Control Register Address Map ......................................... 100
A.2 Built-in Test Control Register Duty Cycle Bit Selector Definitions . 101
A.3 Built-in Test Pseudo-random Noise Pattern Sample .................. 101
D.1 Control Module EPLD (13201) Timing Diagram Parameters ........... 179
D.2 Data Module BIT EPLD (13022) Control Register Timing Diagram Parameters ......................................................... 181
D.3 Receiver Module EPLD (12847) Timing Diagram Parameters ........... 183
D.4 Data Module Arbiter EPLD (12848) Timing Diagram Parameters ... 185
D.5 Data Module BIT EPLD (13022) Operation Timing Diagram Parameters 187
Chapter 1

Introduction

The TAXI Fiber Optic Interface Support Module (TAXI-ISM) is a printed circuit board (PCB) that is part of a larger system called the Airborne Seeker Test Bed Data Multiplexer II, developed by MIT Lincoln Laboratory Group 105. TAXI stands for Transparent Asynchronous Xmitter-Receiver Interface [1]. The notion of TAXI comes from the TAXIchip set developed by Advanced Micro Devices. The TAXIchip set provides the means to establish a transparent high speed serial link between two high performance parallel buses [1].

The primary purpose of the TAXI-ISM is to implement the fiber optic interfaces between the Data Multiplexer II system and all fiber optic input streams. The TAXI-ISM uses the TAXIchip protocols to receive fiber optic serial communications from outside sources. The complete TAXIchip set involves a transmitter and a receiver. An outside source uses a TAXIchip transmitter and a serial fiber optic data link transmitter to serially encode and transmit data using a fiber optic cable. The TAXI-ISM then uses a serial fiber optic data link receiver and a TAXIchip receiver to receive fiber optic data streams and convert them into parallel data streams. The TAXI-ISM processes the data streams and sends them to the remainder of the Data Multiplexer II system.

The input stream arrives in frames of data. Frames are of variable length, but are preceded by between one and four frame synchronization bytes. The number of bytes that the TAXI-ISM uses to look for the beginning of a frame can be set by
internal control registers. The process of searching for the beginning of a frame is called frame synchronization. The TAXI-ISM can handle up to two channels of data. It processes frames of data for each channel and stores them in buffer memory. The TAXI-ISM then generates an address for each data and writes an output data stream to the remainder of the Data Multiplexer II system.

The design of the TAXI-ISM involved various stages. The initial stage involved the design of the hardware schematics for the system. This included logic design, which was accomplished using electronically programmable logic devices (EPLDs). Strict timing analysis was then performed on all control and data signals to verify the design of the system. After the design was completed, research was conducted to find the appropriate components for the printed circuit board. Cost/performance tradeoffs were taken into account. Finally, the artwork for the physical PCB was designed using the schematic diagrams and component mechanical data. The parts were placed on the PCB in an efficient manner, and an auto-router was used to connect the signals with metal traces.

The TAXI-ISM consists primarily of three modules: Control Module, Receiver Module, and Data Module.

Chapter 2 describes the design of the Control Module, including a detailed description of control interfaces, control registers, and control functions.

Chapter 3 describes the architecture and implementation of the Receiver Module. This includes a functional description, a description of the receiver interface, hardware implementation, and logic design.

Chapter 4 describes the design of the Data Module, including a description of the data interfaces with the remainder of the Data Multiplexer II System.

Chapter 5 summarizes the entire design process of the TAXI-ISM system printed circuit board, detailing all design phases and their involvement. It also describes the current status of the TAXI-ISM.
1.1 Motivations for Development

The purpose of the Data Multiplexer II is to provide improved capabilities over the original Data Multiplexer I. The system primarily provides a base system for receiving and transferring data from various radar and sensor systems to a digital cassette recorder. It receives data from various sensors, multiplexes and formats the data streams appropriately, and transfers the data to an Ampex Digital Cartridge Recorder System (Ampex DCRS).

MIT Lincoln Laboratory intends to use the Data Multiplexer II system to perform missile studies. An array of various sensors was placed on an airplane owned by MIT Lincoln Laboratory. This array forms the Airborne Seeker Test Bed (ASTB). The ASTB is currently connected to a Data Multiplexer I, which is then connected to a single Ampex DCRS. MIT Lincoln Laboratory plans to shoot missiles past the airplane in mid-flight, and record various data such as radar footprints, infrared signatures, etc. Analysts can later review the data recorded, and draw appropriate conclusions. Since missiles are very expensive, MIT Lincoln Laboratory would like to record all possible information from each missile, and repeat launchings as infrequently as possible. Due to the cost of an Ampex DCRS, as well as the quantity of sensor data streams, only a few Ampex DCRS systems have been obtained. In order to prevent repeated missile launchings, the Data Multiplexer I was built to buffer and multiplex data streams to a single Ampex DCRS. This provided for a simple, efficient, cost-effective solution.

A shortcoming of the original Data Multiplexer I is that it does not meet the intended performance specifications. The proposed project goal for the Data Multiplexer I was to achieve data transmission rates of 30 MBytes/sec between sensors and the Ampex recorder. This goal has not been met by the existing Data Multiplexer I, which operates at data rates of only 10 MBytes/sec. Failure to meet the proposed system specifications motivated the development of a new improved system called the Data Multiplexer II. The new Data Multiplexer II will replace the old Data Multiplexer I in the ASTB program, providing all the functionality of a Data Multiplexer
I in addition to new features.

The Data Multiplexer II represents a significant improvement over the existing Data Multiplexer I. The main attribute of the new system is that it achieves the system specification of an overall 30 MByte/sec data transmission rate between sensors and an Ampex DCRS.

Another motivation for the development of the Data Multiplexer II is modularity. Most of the sensor systems are generic sensors that can record parallel digital data directly to an Ampex DCRS via parallel emitter-coupled logic (ECL) twisted-pair transmission lines. The transmission of parallel data via twisted-pair cable over long distances is both impractical and unreliable. As a result, fiber optic input capability was added to the Data Multiplexer I. The Data Multiplexer I system now interfaces with several sensors via TAXI-type serial fiber optic data links to facilitate and improve data transfers between the sensors and the Data Multiplexer I over long distances. As a result, the original system contains several different interface cards for each type of sensor that it can interface to. These specialized interface boards contain a tremendous amount of redundant hardware and logic.

The new Data Multiplexer II system replaces each specialized interface card with a Generic Buffer Channel (GBC) card that contains all of the redundant hardware. Several interface support modules were developed as extensions of the GBC card to interface with specific sensor types. The GBC card controls the operation of the interface support modules, and acts as an interface between the interface support modules and the rest of the Data Multiplexer II system.

The TAXI-ISM was designed to replace the existing fiber optic interface card from the old Data Multiplexer I. Coupled with a GBC card, the TAXI-ISM implements the necessary interfaces between the Data Multiplexer II and all fiber optic sensors. Figure 1-1 shows the coupling of a TAXI-ISM with a GBC card.
1.2 Functional Overview

The TAXI-ISM can receive and process up to two 15 MByte/sec fiber optic channels. These data streams are designated Channel 1 and Channel 2. Data is received in the form of serial fiber optic transmission bytes, sequenced into 16-Bit words, and stored in a buffer. The TAXI-ISM then processes each sequenced word, checking for error and status conditions. Some common errors are a data transmission violation produced during the serial fiber optic transmission, or carrier loss produced by a dark fiber optic cable.

After the serial bytes have been sequenced into words, the TAXI-ISM arbitrates between the two channels and transmits data to the remainder of the Data Multiplexer II system. The TAXI-ISM implements a “round-robin arbiter” under normal operating conditions, but gives priority to those data streams which require immediate attention during busy periods. A round-robin arbiter simply gives each channel a turn to write data; first Channel 1 is given the opportunity to write data, then Channel 2, and so on and so forth. If either channel does not have any data to write, it will give up its slot to the other channel. During periods of heavy channel activity, those channels that are approaching buffer full conditions will be given priority to those with less activity.

Data transmitted to the remainder of the Data Multiplexer system is sent to the GBC card in the form of 16-bit parallel words to form a 30 MByte/sec data stream. An address field is also sent with each word in the data stream. Each channel has a separate address space in GBC memory, and this address indicates where the data words for each channel are to be written. In this manner, data in the address space
for Channel 1 belongs to Channel 1, and those in the Channel 2 space belong to Channel 2.

A final concept in the operation of the TAXI-ISM is the Coherent Processing Interval (CPI). A CPI serves to synchronize the arriving input data stream with the remainder of the Data Multiplexer II system, and to prevent memory overflow for each channel. At the start of a new CPI, all the address generators are reset to the base address in each address space, and the TAXI-ISM continues to process data for each channel.

Figure 1-2 shows a block diagram of the entire Data Multiplexer II system. Figure 1-2 also illustrates how the TAXI-ISM fits in with the rest of the Data Multiplexer II system. The Data Multiplexer II consists of several interface modules consisting of a GBC card and an interface support module (ISM). The TAXI-ISM implements one type of interface support module. Another type is the Multi-Band Interface Support Module (MB-ISM). The MB-ISM handles the interface issues concerning wideband and narrowband video or radar data streams. Several interface modules are integrated into each Data Multiplexer II to provide access to various data streams. Both ISM types are shown in Figure 1-2.

The Data Multiplexer II combines or multiplexes the data streams from each interface module into one data stream that is recorded to an Ampex DCRS (hence the name Data Multiplexer II). An additional module in the Data Multiplexer II is the Integrated XBUS Controller (IXC) card, which performs the actual multiplexing of the data streams and controls the operation of the Ampex DCRS. The IXC card also tags each data sample with the GBC identification number and channel from which the data sample originated from. This allows analysts to later review the data recorded to the Ampex DCRS and match the data to the specific sensor.

1.3 Block Diagram

A block diagram of the TAXI-ISM is shown in Figure 1-3. The block diagram is intended as a guideline and as a point of reference, and does not reflect the actual
physical arrangement of the TAXI-ISM.

1.4 Subsystem Layout

The TAXI-ISM consists primarily of three modules: Control Module, Receiver Module, and Data Module. The Control Module operates the Receiver and Data Modules, and does not actually process the data streams. Both channels of data are received and processed by the Receiver Module, and then transmitted to the GBC card through the Data Module.

The Control Module is responsible for the operation of the TAXI-ISM system. The GBC card operates the TAXI-ISM through a series of control and status registers. These registers are 16-Bit read/write or read-only registers, and are defined in the
TAXI-ISM control register address space. A Control Interface exists between the TAXI-ISM and the GBC card by which the GBC card accesses and operates the Control Module.

The Receiver Module is responsible for receiving both channels of fiber optic input from outside sources through a Receiver Interface. The Receiver Module will sequence the 8-bit serial transmissions received through the Receiver Interface into 16-bit parallel words, recording all error and status conditions. The Receiver Module will then put data from both channels into buffers in preparation for transmission to the rest of the Data Multiplexer II system.

The Data Module is responsible for transmitting the 30 MByte/sec parallel data stream to the GBC card. The Data Module writes data to the GBC card through a Data Interface. The Data Module arbitrates between the two channels, giving
priority to those channels that are experiencing heavy traffic. The Data Module is also responsible for calculating the appropriate address field for each data sample. An additional feature of the Data Module is a Built-in Test function (BIT). The BIT function provides the means to generate a known data pattern to be transmitted to the GBC card to test the TAXI-ISM/GBC Data Interface. The BIT function is capable of generating a 16-Bit pseudo-random noise pattern or an alternating bit pattern. During normal operation, the BIT module will appear transparent to the Data Module and will not affect the data stream. During BIT operation, the BIT module will inject the appropriate pattern to test the TAXI-ISM/GBC Data Interface, ignoring all incoming fiber optic data patterns.

1.5 System Input/Outputs

The TAXI-ISM will accept both channels through fiber optic data links designated J1 and J2 for Channel 1 and Channel 2, respectively. Control signals for the Control Module and the Data Module are provided through a connector designated P1 on the TAXI-ISM, and will be discussed in subsequent chapters.

1.6 EPLD File Tutorial

The EPLDs in this project were designed and programmed using the commercially available Altera family of EPLDs. This section provides a tutorial for reading and understanding the EPLD files that were used in this project to implement the logic for the TAXI-ISM.

The EPLD code consists of primarily four sections: Design Section, Subdesign Section, Variable Section, and Logic Section. The Design Section declares the pin assignments for the EPLD by assigning physical pin numbers to each input and output signal. Unspecified pin numbers are unconnected in the actual device.

The Subdesign Section declares the actual input and output signals of the EPLD. The Subdesign Section also assigns a physical type, INPUT, OUTPUT, or BIDIR
(for bi-directional or tri-state signals) to each signal.

The Variable Section declares the logic type of each input/output pin. Logic types are either DFF for DQ flip flops, DFFE for DQ flip flops with enable signals, TRI for tri-state buffers, or SOFT for non-inverting buffers. Flip flops have the standard signals: d, q, clrn, prn, and clk; these signals are accessed by \textit{name.signal}. This section also declares the finite state machines (FSMs) used in the EPLD, as well as any other internal logic variables needed for the operation of the EPLD. The Variable Section also contains a Default Section, where values of any signals that are not driven to a specific logic level are declared.

The Logic Section contains the combinational logic equations used in the EPLD, and begins after the Default Section of the Variable Section. This section implements the combinational logic for the EPLD using internal logic variables or external inputs to the EPLD. The Logic Section uses syntax similar to conventional C programming, such as “IF-THEN” clauses and “CASE” statements.

As a result of logic programming, two additional designations are assigned to each EPLD besides the manufacturer part number. A reference designator, such as U20, is assigned to each integrated circuit chip. Programmable components are assigned an additional file designator to indicate the file name that was used to program the circuit chip. A sample designation for an EPLD might be U20, 13022 (reference designator U20, file 13022).
Chapter 2

Control Module

The Control Module is responsible for the operation of the TAXI-ISM system, including the Receiver and Data Module. Operation of the TAXI-ISM is achieved primarily through writing and reading 16-bit control registers. Each control register has a distinct address and function. The Control Module manages control register reads and writes, performs card resets, and controls the data flow through the Receiver and Data Module.

Table 2.1 shows the input and output signals that make up the interface between the Control Module and the GBC card. Active low signals are followed by the letter N, indicating that they operate according to active low signal levels.

The signal CLK is a 32 MHz clock signal provided by the GBC card. This signal is used as the system clock for most subsystems of the TAXI-ISM card.

TAXI-ISM system resets are controlled by the RESETN signal. During a card reset, all registers are set to their default, startup condition, all buffer memory is erased, and all address counters are reset to their base addresses.

The signals INTRN, ILEV[2..0], and IACKN are for generating interrupt requests through the host Data Multiplexer II system. The TAXI-ISM is not responsible for generating interrupt requests, and INTRN and ILEV[2..0] will be driven to an inactive state, but included for future compatibility. The Control Module will ignore all transitions on the IACKN signal.

The remaining signals are involved with actual control register read and write
<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLK</td>
<td>32 MHz system clock</td>
<td>Input</td>
</tr>
<tr>
<td>RESETN</td>
<td>Board reset signal</td>
<td>Input</td>
</tr>
<tr>
<td>INTRN</td>
<td>Interrupt request</td>
<td>Output</td>
</tr>
<tr>
<td>ILEV[2..0]</td>
<td>Interrupt level</td>
<td>Output</td>
</tr>
<tr>
<td>IACKN</td>
<td>Interrupt Acknowledge</td>
<td>Input</td>
</tr>
<tr>
<td>CTLWRLON</td>
<td>Control register write low</td>
<td>Input</td>
</tr>
<tr>
<td>CTLWRHIN</td>
<td>Control register write high</td>
<td>Input</td>
</tr>
<tr>
<td>CTRLDN</td>
<td>Control register read</td>
<td>Input</td>
</tr>
<tr>
<td>ACKN</td>
<td>Control read/write acknowledge</td>
<td>Output</td>
</tr>
<tr>
<td>CTLADDR[5..0]</td>
<td>Control address bus</td>
<td>Input</td>
</tr>
<tr>
<td>CTLDATA[15..0]</td>
<td>Control data bus</td>
<td>Bidirectional</td>
</tr>
</tbody>
</table>

Table 2.1: Control Module Inputs and Outputs

operations. The control address bus, CTLADDR[5..0], will contain the address of the control register that is being accessed during a control write or read operation. The control data bus, CTLDATA[15..0], will contain the data to be written to a register during a control write, or the data read from a register during a control read. The signal CTRLDN is the control read strobe, and the signals CTLWRLON and CTLWRHIN are the control write strobes.

A block diagram of the Control Module is shown in Figure 2-1. The Control Module consists of two Finite State Machines (FSMs) to control read and write operations. In addition to the Control Write FSM and Control Read FSM, two generic control registers are shown in Figure 2-1. The Control Read FSM decodes the address on the control address bus during a read operation, and drives data onto the control data bus by enabling the output buffer of the appropriate control register. During a control write, the Control Write FSM decodes the address on the control address bus, and writes the data from the control data bus to the appropriate register by enabling the input buffer from the data bus to the register. Figure 2-1 is intended as a guideline and as a point of reference, and does not reflect the actual physical arrangement of the Control Module.
2.1 Hardware Implementation

The logic for the Control Module is encapsulated in an Altera EPLD, manufacturer part number EPM7192GC160-12. The primary control EPLD is designated U6, and is shown in section B of Figure B-3. This control EPLD manages most of the major control functions. The logic for the device is shown in Appendix C.2. This file contains all of the control logic that was used to implement the Control Module. It contains finite state machines to control register reads and writes, all Control Module control registers, and the logic to control an external serial electrically erasable programmable read-only memory (E²PROM).

The E²PROM was used to implement an identification device for the PCB. The E²PROM will contain information such as project revision level, schematic file names,
EPLD file names, etc. The E²PROM is shown in section C1 of Figure B-3, manufacturer part number X25040P. The Control Module can read or write information serially to this E²PROM upon request through control registers. A hardware write protection scheme was implemented to protect the information in the serial E²PROM once it has been programmed. The signal IDWPOUTN functions as a write protect input for the serial E²PROM. A jumper, JP1, was inserted between the control EPLD output for IDWPOUTN and the E²PROM’s input. The E²PROM side of the jumper was then tied to ground through a 4.7 KOhm resistor. During programming of the serial E²PROM, the jumper should be installed, and the Control Module can control the IDWPOUTN signal, driving it high during a write and low during write protection. Once the identification device has been programmed with the proper information, the jumper can be removed from the PCB, and the IDWPOUTN input to the serial E²PROM will stay grounded, protecting the programmed information for future writes, regardless of the Control Module output. The Control Module can then read the true state of the IDWPOUTN signal as defined by the jumper before it attempts to write to the serial E²PROM, preventing any accidental or extraneous writes.

A transparent latch, manufacturer part number IDT74FCT573TSO, was used to latch the address presented on the control address bus, CTLADDR[5..0], from the control interface. The transparent latch is shown in section B3 of Figure B-3. The Control Module controls the latch enable signal, CADDRLE, allowing it to know precisely when the address is valid and invalid.

2.2 Control Register Memory Map

Operation of the TAXI-ISM is achieved primarily through writing and reading 16-bit control registers. All registers are either read/write or read-only. No write-only registers were implemented.

The TAXI-ISM implements a sixty-four register address space. Each register has a unique 6-bit address defined by the control address bus, CTLADDR[5..0]. Table A.1
shows the control register address map. Some registers are designated “Reserved for future use,” meaning that they are currently undefined. Write operations to these registers will have no effect, and read operations of these registers shall return all zeroes. In addition, some bits within each register are also reserved. Write operations to reserved bits will have no effect, and read operations to reserved bits shall return all zeroes. All bits in each register default to zero upon reset or power up, except where specified.

Control registers were implemented as banks of DQ flip flops with enables, making them suitable for storage inside an EPLD. Since there are several EPLDs on the TAXI-ISM, it was more efficient to store relevant control registers in separate EPLDs instead of in a single EPLD; however, the main control EPLD, 13021, still manages all the control read and write operations.

The Control Module registers are stored in EPLD 13021, reference designator U6. These are located in address 0x00 (hex) to 0x09 and 0x0C to 0x1F.

Each Channel has its own specific registers. A separate EPLD was implemented for each channel. The Channel 1 registers are located at addresses 0x20 to 0x27, and these are contained in EPLD 12847, reference designator U5. The Channel 2 registers are located at addresses 0x30 to 0x37, and these are contained in EPLD 12847, reference designator U13.

The remaining registers belong to the Data Module. The BIT functions of the Data Module are contained in a separate EPLD. These registers are located at addresses 0x0A and 0x0B, and are stored in EPLD 13022, reference designator U20. An additional set of registers called Ending Address Registers also belong to the Data Module. Ending address registers store the last address written by the Data Module incremented by one for the previous CPI, and are used to perform length calculations. They are located in addresses 0x28 to 0x2D, and are stored in EPLD 13035, reference designator U21.
2.3 Control Register Descriptions

This section describes the bit definitions of each of the TAXI-ISM control registers.

2.3.1 PCB Identification Control/Status Register

This register controls the reading and writing of the X25040P serial E²PROM, and implements part of the PCB Identification Device as described in Section 2.1. This register is a 16-bit read/write register located at address offset 0x01. Bits 15 to 2 are reserved for future use.

Bits 2 to 1 implement the hardware write protection function discussed in Section 2.1. Bit 1 is the Control Module output for the write-protect signal to the serial E²PROM. The Control Module can write a logic low (0) on bit 1 to attempt to put the PCB Identification Device in write protected mode, or it can write a logic high (1) to attempt to enable writing to the X25040P. Bit 2 shows the true state of the serial E²PROM write enable signal, which is derived from the physical pin on the X25040P.

Bit 0 implements a chip select function, whereby the Control Module can manually control the chip select of the X25040P chip. A logic low (0) deselects the X25040P, and a logic high (1) selects the E²PROM.

2.3.2 PCB Identification Data Register

This register is used to provide the data interface to the X25040P serial E²PROM, and is a 16-bit read/write register located at address offset 0x03. Bits 15 to 8 are reserved.

Bits 7 to 0 contain the data to interface with the X25040P. During a write to the E²PROM, the data that will be written to the serial E²PROM is placed in this register by the Control Module. During a read operation, the data that will be read from the serial E²PROM will be placed in this register by the X25040P after a read cycle has completed.
2.3.3 Control Register

This register is the primary control register for the TAXI-ISM. It is a 16-bit read/write register located at address offset 0x08. Bits 15 to 1 are reserved for future use.

Bit 0 controls the operation of the Receiver and Data Modules. When this bit is a logic low (0), operation of the Receiver and Data Module is suspended and the TAXI-ISM is put into an idle mode. No data is received by the Receiver Module through the Receiver Interface, no data is written by the Data Module through the Data Interface, and no error or status conditions are logged. When this bit is a logic high (1), the TAXI-ISM operates in a normal state.

The Control Module is not affected by the operation of this bit, and never enters an idle mode. All control registers will still function in idle mode. The purpose of this bit is to allow proper initialization of the TAXI-ISM without any interference from incoming data.

2.3.4 Error Register

This register is the primary error register for the TAXI-ISM, and logs all major card errors. It is a 16-bit register located at address offset 0x09. Bits 15 to 4 and 1 to 0 are reserved for future use.

Bit 3 represents a GBC Data Interface overflow. Data is written by the Data Module to the GBC card through the Data Interface, and eventually stored in GBC buffer memory. If an overflow of the GBC buffer memory is detected by the Data Module, this bit shall be written with a one, otherwise it shall be a zero.

Bit 2 represents a data loss error. If the TAXI-ISM detects that data has been lost, it shall set this bit. Data loss can occur as a result of a GBC Data Interface overflow (signified by bit 3), or if any of the Data Module address generators overflow (which will overwrite previous data causing the data loss).
2.3.5 Built-in Test Control Register

This register controls the operation of the Built-in Test function of the Data Module, and is a 16-bit read/write register located at address 0x0A. Bits 15 to 12 are reserved for future use.

Bits 11 to 7 determine the test pattern to be generated by the BIT function. A value of 0b000000 (binary) selects alternating data, where the bits of the pattern alternate between 1 and 0. A value of 0b000001 selects pseudo-random noise patterns. All other values of bits 11 to 7 are reserved for future use.

Bits 6 to 1 determine the BIT duty-cycle. The BIT function simulates an active/inactive duty cycle of 1,024 writes to the GBC card. Table A.2 shows the definitions for the duty-cycle selectors. For example, a selection of 0b000011 will select a duty-cycle of 512 active cycles and 512 inactive cycles. During the active portion of the duty-cycle, the Data Module is free to send data to the GBC card through the Data Interface. During the inactive portion of the duty-cycle, the Data Module will not send any data to the GBC card. This feature allows the TAXI-ISM to simulate bursty data streams, and test the variation of the Data Module to changes in data rates.

Bit 0 enables or disables the BIT functions. A logic low (0) disables BIT functions; a logic high (1) enables BIT functions.

2.3.6 Built-in Test Base Pattern Register

This register specifies the base pattern used by the BIT function, and is a 16-bit read/write register located at address 0x0B.

For the alternating data pattern, this register selects the initial value. For example, if the base pattern register contains 0xAA (hex), then the BIT function will generate the pattern 0xAAAA, 0x5555, 0xAA, etc. For pseudo-random noise patterns, this selects the initial seed a random number generator will use to generate the pseudo-random noise pattern.
2.3.7 EPLD Identification Registers

This register is an EPLD identification register that contains the current revision level of the logic programs that were used to program each EPLD. Every EPLD on the TAXI-ISM is required to have an identification register. This register is a 16-bit read-only register. Bits 15 to 8 are reserved, and bits 7 to 0 contain the revision level. The initial revision shall have a value of zero. Subsequent revision levels shall have sequentially increasing values.

There are a total of six EPLDs on the TAXI-ISM printed circuit board: the Channel 1 Receiver Module EPLD (12847), Channel 1 Receiver Module EPLD (12847), Ending Address Register Controller EPLD (13035), Built-in Test Controller EPLD (13022), and Control Module EPLD (13021). The functionality of each of these EPLDs will be deferred to their relevant sections. The location of the respective identification registers is shown in Table A.1.

2.3.8 Channel Control Register

This register implements the control for the Receiver Module. Each channel has a Channel Control Register. The Channel 1 Control Register is located at address 0x21, and the Channel 2 Control Register is located at address 0x31. This is a 16-bit read/write register. Bits 15 to 8 are reserved.

Bit 7 serves as a reset bit to the memory buffers. This bit defaults to 1 after a card reset. A logic high (1) clears the memory buffer; a logic low (0) does not affect the memory buffer.

Bit 6 serves as a TAXIchip receiver reset. A logic high (1) causes the TAXIchip to reset its clock circuits and start receiving new data. A logic low (0) does not affect the TAXIchip. If asserted, this signal should be asserted for a minimum of 1 millisecond to ensure proper reset [1].

Bit 5 serves as a signal detect enable. The Receiver Module contains a signal detect that monitors whether or not an active fiber optic cable is connected to the input of the TAXI-ISM. If this bit is a logic high (1), then the Receiver Module will
ignore incoming data unless the signal detect signal verifies that an active fiber optic
cable is connected to the input. If this bit is a logic low (0), then the Receiver Module
will record all data regardless of the state of the signal detect.

Bits 4 to 3 define the number of bytes to use for frame synchronization. The
Receiver Module scans the input stream for the frame synchronization bytes and sets
an appropriate status signal if it has found those bytes. A value of $0b00$ selects the
use of only one frame synchronization byte; $0b01$ selects two bytes; $0b10$ selects three
bytes; $0b11$ selects four bytes.

Bit 2 defines the byte-to-word sequencing mode of the Receiver Module. If this
bit is a logic high (1), then the byte-to-word sequencing is reset when the TAXIchip
receives a valid synchronization signal. A logic low (0) will not affect the byte-to-word
sequencing.

Bits 1 to 0 define the method of byte-to-word sequencing. The Receiver Module
takes two sequential input bytes and sequences them into a single 16-bit word. A value
of $0b00$ disables the byte-to-word sequencing; $0b01$ selects the first byte received as
the most significant byte, and the second byte as the least significant byte; $0b10$
selects least significant byte first, most significant byte second; $0b11$ is reserved for
future use.

### 2.3.9 Channel Status Register

This register contains the status flags for the Receiver Module. Each channel has a
Channel Status Register. The Channel 1 Status Register is located at address 0x22,
and the Channel 2 Status Register is located at address 0x32. This register is a 16-bit
read-only register. Bits 15 to 2 are reserved for future use.

Bit 1 records the status of the Receiver Module frame synchronizer. This bit is
set if the Receiver Module has detected the number of frame synchronization bytes
specified by bits 4 to 3 of the Channel Control Register.

Bit 0 records the status of the TAXIchip. This bit is set if the TAXIchip receives
valid synchronization.
2.3.10 Channel Frame Synchronization Byte Registers

These registers provide the synchronization bytes that the Receiver Module can use for frame synchronization. There are four possible bytes that the Receiver Module can use for Frame Synchronization. Each byte can be selected by selecting the appropriate mode using bits 4 to 3 of the Channel Control Register.

Each channel has four Channel Frame Synchronization Byte Registers. Channel 1's Frame Synchronization Byte Registers are located at addresses 0x23, 0x24, 0x25, and 0x26. Channel 2's Frame Synchronization Byte Registers are located at addresses 0x33, 0x34, 0x35, and 0x36.

These registers are 16-bit read/write registers. Bits 15 to 8 of each register are reserved for future use. Bits 7 to 0 contain the actual frame synchronization byte.

2.3.11 Channel Receiver Error Register

This register contains the Receiver Module receiver errors. Each channel has a Channel Receiver Error Register. The Channel 1 Receiver Error Register is located at address 0x27, and the Channel 2 Receiver Error Register is located at address 0x37. This is a 16-bit read/write register. Bits 15 to 4 are reserved for future use.

Bit 3 records if an overflow has occurred in the channel address generators, i.e. if the address counters have passed their maximum address limit. Bit 2 records if the channel buffer memory overflows due to a buffer full condition. Bit 1 records if an error occurs during the fiber optic relaying of an input byte. Bit 0 records if a signal loss has occurred.

2.3.12 Channel Ending Address Low Register

This register serves to record the low sixteen bits of the ending address for each channel. The ending address is the last address written to the GBC card by the Data Module before a new CPI is initiated, incremented by one. The ending addresses can then be used in length calculations, recording the number of bytes written by each channel during the previous CPI.
The addresses written to the GBC card are twenty-two bits wide, so the ending address was divided across two registers, a low and a high, since registers are only sixteen bits wide. Each channel has a Channel Ending Address Low Register. The Channel 1 Ending Address Low Register is located at address 0x28, and the Channel 2 Ending Address Low Register is located at address 0x2A. This is a 16-bit read-only register. Bits 15 to 0 contain the low sixteen bits of the ending address.

### 2.3.13 Channel Ending Address High Register

This register serves to record the high six bits of the ending address for each Channel. Each channel has a Channel Ending Address High Register. The Channel 1 Ending Address High Register is located at address 0x29, and the Channel 2 Ending Address High Register is located at address 0x2B.

This is a 16-bit read-only register. The Ending Address Register defaults to the base address of each channel’s address space after a card reset; 0x000000 for Channel 1 and 0x200000 for Channel 2. As a result, the Channel 2 Ending Address Register High Register defaults to 0x0020 after a card reset. In addition, the Channel 2 Ending Address High Register has one more bit than Channel 1 because the ending address for Channel 2 can be as high as 0x400000 (0x3FFFFF plus 1). Therefore, bits 15 to 6 of the Channel 1 Ending Address High Register are reserved for future use, and bits 5 to 0 contain the high six bits of the ending address for Channel 1. Bits 15 to 7 of the Channel 2 Ending Address High Register are reserved for future use, and bits 6 to 0 contain the high seven bits of the ending address for Channel 2.

### 2.4 Control Register Write Operations

Control register writes are accomplished when the GBC card drives the control address and data buses, CTLADDR[5..0] and CTLDATA[15..0], to valid signal levels and asserts either of the control write lines, CTLWRHIN or CTLWRLON. All control registers are sixteen bits wide. Asserting either control write line independently will perform an eight bit write to the corresponding byte of the register, while leaving
the other byte unaffected. For example, asserting CTLWRHIN will write the data on the top byte of the control data bus to only the top eight bits (15 to 8) of a control register. Asserting both control write lines simultaneously will write all sixteen bits of the control data bus to the control register chosen by the CTLADDR[5..0] bus.

Once the Control Module completes writing to a control register, it asserts an acknowledge signal, ACKN, to signify to the GBC card that the write operation has completed. The Control Module will continue to assert the ACKN signal until both of the control write lines have been deasserted. A diagram specifying the timing constraints of a write operation is shown in Figure 2-2. Figure 2-2 details all of the timing guarantees for inputs to the Control Module involved in a control write operation, as well as all the timing constraints on the outputs of the Control Module.
2.4.1 Selecting Control Registers

Each group of control registers is selected by a combinational logic variable. The variable SELID is asserted (logic high) if the address on the control address bus designates the Control Module EPLD (13021) Identification Register, address 0x2F. The variable SELCTL is asserted if the control address bus is selecting a Control Module register, all of which are located in addresses 0x00 to 0x0F (see Table A.1). The variable SELCHN is asserted if the control address bus is selecting a channel register. All of the Channel 1 and 2 control registers are located in addresses 0x20 to 0x27 and 0x30 to 0x37, all of which are of the form 0b1*0*** (where * denotes a don’t care). Using this strategy, the variable SELCHN is asserted if bits 5 and 3 have the values of 1 and 0, respectively. The variable SELRES is asserted if any other control register is selected by the control address bus, i.e. those registers in addresses 0x28 to 0x2E.

The latch enable signal for the control address bus transparent latch was implemented by logically ANDing the control read signal CTLRDN, the control write signals CTLWRLON and CTLWRHIN, and the acknowledge signal ACKN. In this manner, the address will be held valid by the transparent latch throughout an entire control read or write operation, from the assertion of the control signal to the deassertion of the acknowledge signal.

2.4.2 Control Write Finite State Machine

An independent finite state machine (FSM) was implemented in the Control Module EPLD (13021) to manage control writes. This FSM is designated WRITEFSM. The control write FSM consists of six states: WRITEIDLE, WRITEDATA, WRITEID-WAIT, WRITECHN, WRITENOTSEL, and WRITEACK. A state diagram of the control write FSM is shown in Figure 2-3.

The WRITEIDLE state waits for a control register write operation to begin. If both control write lines are inactive, then the FSM stays in the WRITEIDLE state. If the variable SELCTL and the control write line CTLWRLON are asserted, then
If SELCHN and CTLWRRLON are asserted, then a channel control register is selected and the write FSM will transition to the state WRITENOTSEL. If either CTLWRHIN or CTLWRRLON is asserted and none of the selector variables are asserted, then the selected register is not in the Control Module EPLD or the Channel 1 and 2 Receiver Module EPLDs, and the write FSM transitions to the state WRITENOTSEL.

The WRITENOTSEL state writes data to Control Module control registers. A case statement in the WRITENOTSEL state writes the data on the low byte of the control

Figure 2-3: State Diagram of the Control Write FSM

a Control Module register is selected and the write FSM will transition to the state WRITEDATA. If SELCHN and CTLWRRLON are asserted, then a channel control register is selected and the write FSM will transition to the state WRITECHN. If either CTLWRHIN or CTLWRRLON is asserted and none of the selector variables are asserted, then the selected register is not in the Control Module EPLD or the Channel 1 and 2 Receiver Module EPLDs, and the write FSM transitions to the state WRITECHN.
data bus to the selected control register. An additional wait state is required if the selected register is the PCB Identification Data Register, since writing to this register initiates a write cycle to the X25040P serial E²PROM. If this register is selected, the write FSM will transition to the WRITEIDWAIT state, otherwise it will transition to the WRITEACK state.

The WRITEIDWAIT state initiates and waits for the completion of a write cycle to the serial identification E²PROM. A variable ID.WRITE initiates the serial write operation, and a variable ID.LAG.DONE signifies the end. The WRITEIDWAIT state first asserts ID.WRITE, and then remains in this state until the variable ID.LAG.DONE is a logic high. The write FSM then transitions to the WRITEACK state.

The WRITECHN state manages writes to the Channel 1 and 2 control registers. The Receiver Module EPLDs perform the actual writing of the channel registers, and then assert a channel acknowledge signal at the end of the write cycle. The Channel 1 and 2 control write acknowledges are provided by the variables CH1ACK and CH2ACK, respectively. The WRITECHN state waits for both of these acknowledges to become asserted before proceeding to the state WRITEACK.

The WRITENOTSEL state is a wait state to allow for all control register writes besides Control Module and channel registers to complete. Since all other modules operate on the same system clock as the Control Module, only a single FSM state is required to allow the completion of a control register write.

The final state in the write FSM is the state WRITEACK. The WRITEACK state acknowledges the completion of a control write to the GBC card. This is done by asserting the signal ACKN. This acknowledge is asserted until both control write lines have been deasserted. Once this has occurred, the write FSM transitions back to the state WRITEIDLE, where it waits for another control write to occur.

The logic for the control write FSM is shown in Appendix C.2.
2.4.3 Other EPLD Control Write Finite State Machines

Since the control registers are spread across several EPLDs on the TAXI-ISM, each EPLD that contains a control register has its own simplified version of the Control Module WRITEFSM to manage writing of the data to its own registers. All the individual finite state machines collectively perform a control write operation and make up the Control Write FSM shown in Figure 2-1. The main Control Module WRITEFSM still manages the overall control write operation, but these EPLDs “snoop” the address bus to see if the register selected is their own register and perform the actual register writes.

The Channel 1 and 2 Receiver Module EPLDs (12847) and Built-in Test Function EPLD (13022) both have their own simplified write FSMs to perform the writing of their respective registers.

Channel 1 and 2 Receiver Module EPLDs (12847)

The Channel 1 and 2 Receiver Module write FSM is also designated WRITEFSM, and is shown in the EPLD file contained in Appendix C.1. The state diagram is similar to that of the main control write FSM shown in Figure 2-3, and is omitted. A main difference is that this FSM contains only three states: WRITEIDLE, WRITEACK, and WRITEWAIT. A similar selector variable, FNSEL, is asserted if a Channel 1 or 2 register is selected.

The WRITEIDLE state waits for a control write operation to occur. If the variable FNSEL is asserted, then the channel EPLD knows that a write to one of the channel registers is occurring. The write FSM writes the data to the appropriate register, and transitions to the WRITEACK state. If a control register write is occurring but FNSEL is not asserted, the write FSM will transition to the WRITEWAIT state.

The WRITEACK state acknowledges the completion of a channel register write to the Control Module WRITEFSM by driving the channel acknowledge signal, CH1ACKN or CH2ACKN. It waits for the control write line to become deasserted, and then transitions to the WRITEIDLE state to wait for another control register
The WRITEWAIT state serves as a wait state for those writes to registers that are not channel registers. It waits for the control write line to become deasserted, and then transitions to the WRITEIDLE state to wait for another control register write operation.

**Built-in Test Controller EPLD (13022)**

The Built-in Test Controller EPLD write FSM is also designated WRITEFSM, and is shown in the EPLD file contained in Appendix C.3.3. The state diagram is similar to that of the main control write FSM shown in Figure 2-3, and is omitted. A main difference is that this FSM contains only five states: WRITEIDLE, WRITEHILO, WRITELOW, WRITEHIGH, and WRITEWAIT. Another difference is that the BIT write FSM is responsible for writing the high byte of control registers, since the Built-in Test Base Pattern Register is one of the few registers that utilizes the top byte (see Section 2.3.6). A similar selector variable, SELBIT, is asserted if a BIT register is selected.

The WRITEIDLE state waits for a control write operation to occur. If the variable SELBIT is asserted, then the BIT EPLD knows that a write to one of the BIT registers is occurring. The write FSM transitions to the state WRITEHILO if both control write lines are asserted, WRITELOW if only CTLWRLON is asserted, and WRITEHIGH if only CTLWRHIN is asserted. If a control register write is occurring but SELBIT is not asserted, the write FSM will transition to the WRITEWAIT state.

The WRITEHILO state performs a 16-bit write operation to the selected BIT register. Once the write is complete, the FSM transitions to the state WRITEWAIT.

The WRITELOW state performs a 8-bit write operation to the low byte of the selected BIT register. Once the write is complete, the FSM transitions to the state WRITEWAIT.

The WRITEHIGH state performs a 8-bit write operation to high byte of the selected BIT register. Once the write is complete, the FSM transitions to the state WRITEWAIT.
The WRITEWAIT state serves as a wait state for those writes to registers that are not BIT registers. It waits for the control write lines to become deasserted, and then transitions to the WRITEIDLE state to wait for another control register write operation.

2.5 Control Register Read Operations

Control register reads are accomplished when the GBC card drives the control address bus, CTLADDR[5..0], to a valid signal level and asserts the control read line, CTLRDN. All control registers reads are sixteen bits wide. When the appropriate register has been located, the Control Module will drive the data inside the register onto the control data bus.

Once the Control Module drives the contents of the selected control register onto the control data bus, it asserts an acknowledge signal, ACKN, to signify to the GBC card that the read operation has completed and valid data is on the control data bus. The Control Module will continue to assert the ACKN signal until the control read line has been deasserted. A diagram detailing the timing specifications of a read operation is shown in Figure 2-4. Figure 2-4 details all of the timing guarantees for inputs to the Control Module involved in a control read operation, as well as all the timing constraints on the outputs of the Control Module.

2.5.1 Control Read Finite State Machine

An independent finite state machine (FSM) was implemented in the Control Module EPLD to manage control reads. This FSM is designated READESM, and consists of four states: READIDLE, READID, READCHN, and READACK. Selection of the control registers to be read is performed in the same manner as described in Section 2.4.1. A state diagram of the control read FSM is shown in Figure 2-5.

The READIDLE state waits for a control register read operation to begin. If the control read line is inactive, then the FSM stays in the READIDLE state. During a control read operation, a case is performed on the selection variables to determine
which type of control register is being requested. If the PCB Identification Data Register is selected, then the read FSM will transition to the READID state. If a channel register is selected, then the read FSM will transition to the state READCHN. If any other register is selected, then the read FSM will transition immediately to the state READACK, where it will assert the acknowledge signal once the selected register’s contents have been driven onto the control data bus.

The READID state initiates and waits for a read cycle to the serial identification E²PROM to complete. A variable ID_READ initiates the serial identification read cycle, and a variable ID_LAG_DONE signifies the end. The READID state first initiates the read cycle by asserting ID_READ, and then remains in this state until the variable ID_LAG_DONE is a logic high. The read FSM then transitions to the READACK state.

The READCHN state waits for reads from the Channel 1 and 2 control registers to complete. The Receiver Module EPLDs perform the actual reading of channel registers, and assert a channel acknowledge signal once this operation has completed. The
Channel 1 and 2 control read acknowledges are provided by the signals CH1ACK and CH2ACK, respectively. The READCHN state waits for both of these acknowledges to become asserted before proceeding to the state READACK.

The READACK state drives the contents of the appropriate control register onto the low byte of the control data bus and asserts the control acknowledge signal. This state performs a case on the control address bus to select the correct register. If any of the channel registers or those registers that do not belong to the Control Module are selected, the READACK state does not drive the control data bus, and instead lets the appropriate EPLD drive the bus. It still asserts the acknowledge signal to signify valid data is on the bus. Once the control read line has been deasserted, the read FSM will transition back to the initial READIDLE state and wait for another control register read operation to begin.

The logic implementing the control read FSM is shown in Appendix C.2.
2.5.2 Other EPLD Control Read Finite State Machines

Since the control registers are spread across several EPLDs on the TAXI-ISM, each EPLD that contains a control register has its own simplified version of the Control Module READFSM to manage reading of the data from its own registers. All the individual finite state machines collectively perform a control read operation and make up the Control Read FSM shown in Figure 2-1. The main Control Module READFSM still manages the overall control read operation, but these EPLDS “snoop” the address bus to see if the register selected is their own register and perform the actual register reads.

The Channel 1 and 2 Receiver Module EPLD (12847), Built-in Test Function EPLD (13022), and Ending Address Register Controller EPLD (13035) all have their own simplified read FSMS to perform the reading of their respective registers.

Channel 1 and 2 Receiver Module EPLD (12847)

The Channel 1 and 2 Receiver Module read FSM is also designated READFSM, and is shown in the EPLD file contained in Appendix C.1. The state diagram is similar to that of the main control read FSM shown in Figure 2-5, and is omitted. A main difference is that this FSM only contains three states: READIDLE, READACK, and READWAIT. A similar selector variable, FNSEL, is asserted if a Channel 1 or 2 register is selected.

The READIDLE state waits for a control register read operation to begin. If the control read line is inactive, then the FSM stays in the READIDLE state. If the variable FNSEL is asserted, then the channel EPLD knows that a read from one of the channel registers is occurring. The read FSM reads the data from the appropriate register, drives the data onto the low byte of the control data bus since channel registers are only defined for the low byte, and transitions to the READACK state. If a control read is occurring, but FNSEL is not asserted, the FSM will transition to the READWAIT state.

The READACK state drives the contents of the appropriate control register onto
the low byte of the control data bus and acknowledges the completion of the read operation to the Control Module by driving the channel acknowledge signal. This state performs a case on the control address bus to select the correct register. It then waits for the control read line to become deasserted, and transitions to the READIDLE state to wait for another control register read operation to begin.

The READWAIT state serves as a wait state for those reads from registers that are not channel registers. It waits for the control read line to become deasserted, and then transitions to the READIDLE state to wait for another control register read operation to begin.

**Built-in Test Controller EPLD (13022)**

The Built-in Test Controller EPLD read FSM is also designated READFSM, and is shown in the EPLD file contained in Appendix C.3.3. The state diagram is similar to that of the main control read FSM shown in Figure 2-5, and is omitted. A main difference is that this FSM only contains three states: READIDLE, READDATA, and READWAIT. In addition, the BIT EPLD drives the entire control data bus since some BIT registers are defined for all sixteen bits. A similar selector variable, SELBIT, is asserted if a BIT register is selected, while another variable, SELID, is asserted if the Built-in Test Controller EPLD (13022) ID Register is selected.

The READIDLE state waits for a control register read operation to begin. If the control read line is inactive, then the FSM stays in the READIDLE state. If the variable SELBIT or SELID and the control read line are asserted, then the BIT EPLD knows that a read from one of the BIT registers is occurring. The read FSM reads the data from the appropriate register, drives the data onto the entire control data bus, and transitions to the READACK state. If a control read is occurring, but FNSEL is not asserted, the FSM will transition to the READWAIT state.

The READDATA state continues to drive the data onto the control data bus for additional clock cycles until the read operation has completed. Once the CTLRDN line has been deasserted, the FSM will transition back to the READIDLE state to wait for another control register read operation to begin.
The READWAIT state serves as a wait state for those reads from registers that are not BIT registers. It waits for the control read line to become deasserted, and then transitions to the READIDLE state to wait for another control register read operation to begin.

Ending Address Register Controller EPLD (13035)

The Ending Address Register Controller EPLD read finite state machine is also designated READFSM, and is shown in the EPLD file contained in Appendix C.3.2. The state diagram is similar to that of the main control read FSM shown in Figure 2-5, and is omitted. A main difference is that this FSM only contains three states: READIDLE, READDATA, and READWAIT. Another difference is that the Ending Address Register Controller EPLD read FSM is responsible for driving the high byte for control register reads for all registers other than BIT Registers. A similar selector variable, SELCTL, is asserted if an ending address register is selected, while another variable, SELBIT, is asserted if a BIT register is selected.

The READIDLE state waits for a control register read operation to begin. If the control read line is inactive, then the FSM stays in the READIDLE state. If the variable SELCTL and the control read line are asserted, then the Ending Address Register Controller EPLD knows that a read from one of the ending address registers is occurring. The read FSM reads the data from the appropriate register, drives the data onto the entire control data bus, and transitions to the READACK state. If a control read is occurring, but SELCTL is not asserted, the FSM will transition to the READWAIT state.

The READDATA state continues to drive the data onto the entire control data bus for additional clock cycles until the read operation has completed. Once the CTLRDN line has been deasserted, the FSM will transition back to the READIDLE state to wait for another control register read operation to begin.

The READWAIT state drives the high byte of the control data bus for all register reads except BIT registers. If SELBIT is asserted, the READWAIT state does not drive the high byte of the control data bus since the BIT Controller EPLD will
handle this. If SELBIT is not asserted, the READWAIT state will drive zeroes on the high byte of the control data bus, since undefined bits in control registers are zero when read. Then, the READWAIT state waits for the control read line to become deasserted, and transitions to the READIDLE state to wait for another control register read operation to begin.

2.6 Control Register Implementation

All of the registers that were used to control the TAXI-ISM were implemented as banks of DQ flip flops using the logic capabilities of EPLDs. The writing and reading of these registers are handled by specific finite state machines as described in previous sections. A separate implementation was used for registers that store control information, such as error and status registers. These two registers have additional finite state machines to handle the updating of information.

2.6.1 Error Registers

An error register is used to store all the errors that occur during the operation of the TAXI-ISM. Error registers are 16-bit read/write registers. The bits in this register are set (logic high) when the corresponding error condition occurs. Once set though, all bits in this register remain set until either explicitly cleared or a reset occurs. Bits in this register are cleared by writing a value with a one in the bit position or positions to be cleared. For example, if the 0th, 2nd, and 3rd bits of an error register are set (it contains 0x000B), writing a value of two (0b0010) shall have no effect; writing a value of four (0b0100) shall clear the 2nd bit and leave the other bits set; writing a value of eleven (0b1101) shall clear all the bits. This function prevents any accidental clearing of error bits.

The Error Register (address 0x09), Channel 1 Error Register (address 0x27), and Channel 2 Error Register (address 0x37) were implemented in this manner. An FSM designated ERRORFSM in the Control Module EPLD (13021) was used to implement the Error Register. A similar FSM designated FNRERFSM in the Channel
1 and 2 Receiver Module EPLD (12847) was used to implement the channel error registers. A description of ERRORFSM will be given only since both FSMs are essentially identical. State names for FNRERFSM will be given in parentheses where appropriate.

The error register FSM consists of four states: ERIDLE (FNRERIDLE), ERWAIT (FNRERIWAIT), ERNOTI (FNRERNOTI), and ERNIWAIT (FNRERNIWAIT). A state diagram of the FSM is shown in Figure 2-6. The first two states control the error registers during idle operation of the Control Module. The latter two states control the error registers during non-idle operation. The Control Module is put into idle mode during two operations. The first is controlled by bit 0 of the Control Register as described in Section 2.3.3. The second is during operation of the BIT function of the Data Module. This is controlled by bit 0 of the Built-in Test Control Register as described in Section 2.3.5.

The ERNOTI (FNRERNOTI) state controls non-idle operation of the error registers. The bits of the error registers are constantly updated in this state by logically ORing the bits of the register with their corresponding error signals. The first thing that this state does is check if the TAXI-ISM is not idle. If it is idle, then the FSM will transition to the ERIDLE (FNRERIDLE) state.

During a control read operation of the error register, the error registers will be updated with the current error conditions. An interface register called ERROR_OUT (FNRER_OUT) is used to interface with the control data bus. The contents of the chosen error register are copied into the interface register at the beginning of a control read. Then, the contents of the interface register are driven onto the control data bus, while the error register is updated with the current error conditions. This prevents fluctuations in the control data bus during a read operation, while still recording the current error conditions. Once a read operation has begun, the data driven will not change, even if the error conditions have. These new errors will be reflected in the next reading of the error registers. The FSM will then transition to the state ERNIWAIT (FNRERNIWAIT) where it will wait until the control read operation has completed.
During a control write operation, bits in the error registers are cleared if there is a one on the control data bus in the corresponding bit position. If a new error condition occurs, this will override a clear operation, and the current error state will be recorded. The FSM will then transition to the state ERNIWAIT (FNRERNIWAIT) where it will wait until the control operation has completed.

If a control read or write operation of another register besides the error registers is occurring, the FSM will transition immediately to the state ERNIWAIT (FNRERNIWAIT), where it will wait until the control read or write operation is completed.

The ERIDLE (FNRERIDLE) state controls the error registers during idle mode. This state operates similarly to the ERNOTI (FNRERNOTI) state, except that the
bits of the error register are not constantly updated since the TAXI-ISM is in idle mode. All bits are set to zero when cleared, regardless of the current error conditions. In addition, transitions are made to the state ERIWAIT (FNRERIWAIT), an idle wait state where the FSM will wait until the current control read or write operation is completed.

2.6.2 Status Registers

A status register is used to store the status conditions that arise during operation of the TAXI-ISM. Status registers are 16-bit read-only registers. Bits in this register are set when the corresponding status condition is satisfied. Once set, the bits shall remain set until a control register read of this register has occurred. To erase the register, a read operation of this register must be completed.

The Channel 1 Status Register (address 0x22) and Channel 2 Status Register (address 0x32) were implemented in this manner. An FSM designated FNSRFSM in the Channel 1 and 2 Receiver Module EPLDs (12847) was used to implement the channel status registers.

The status register FSM consists of four states: FNSRIDLE, FNSRIRDWAIT, FNSRNOTI, and FNSRNIRDWAIT. Figure 2-7 shows a state diagram of the status register FSM. The first two states control the status registers during idle operation of the Control Module. The latter two states control the status registers during non-idle operation.

The FNSRIDLE state controls the status register during idle mode. The first thing that this state does is check if the TAXI-ISM is in idle mode. If it is not, then the FSM will transition to the state FNSRNOTI. Otherwise, during a control read of the status registers, the FSM copies the contents of the status register to an interface register that is used by the Control Module READFSM to drive data onto the control data bus. It then resets the status register bits to zero, and transitions to the FNSRIRDWAIT state to await the completion of the current read operation. If a read of another register besides the status register is occurring, the FSM will transition immediately to the FNSRIRDWAIT state.
2.7 PCB Identification Device

The control of the serial PCB Identification Device is implemented by a finite state machine designated ID_CLK_CONTROLLER in the Control Module EPLD (13021).
A state diagram for the ID_CLK CONTROLLER is shown in Figure 2-8.

A serial write operation to the X25040 serial E²PROM PCB Identification Device is performed by first writing the appropriate control information (enable chip select and write enable as specified in Section 2.3.1) to the PCB Identification Control/Status Register (address 0x01), and then writing the data to be transferred to the serial E²PROM in the PCB Identification Data Register (address 0x03). The Control Module WRITEFSM will initiate a serial transfer by asserting the variable ID_WRITE. The ID_CLK CONTROLLER will then perform the serial write operation, and signify its completion by asserting ID_LAG_DONE.

A serial read operation from the X25040P serial E²PROM is performed by first
writing the appropriate control information (enable chip select and write enable as specified in Section 2.3.1) to the PCB Identification Control/Status Register (address 0x01), and then reading the requested information from the PCB Identification Data Register (address 0x03). The Control Module READFSM will initiate a serial transfer by asserting the variable ID_READ. The ID_CLK_CONTROLLER will then perform the serial read operation, and signify its completion by asserting ID_LAG_DONE.

The primary purpose of the ID_CLK_CONTROLLER FSM is to serially shift out the data to be written to the serial E²PROM during a write operation, and serially shift in the data read from the serial E²PROM during a read operation. An additional function of the FSM is to generate the X25040P serial clock, which has a maximum frequency of 1 MHz [9]. The ID_CLK_CONTROLLER FSM consists of six states: ID_IDLE, ID_WRITE_LOW, ID_WRITE_HIGH, ID_READ_LOW, ID_READ_HIGH, and ID_SDOUT_LAG. The shifting of the data byte was performed using an 8-bit shift register implemented using a temporary register, ID_TEMP. A counter, ID_SHIFT_COUNTER, was used to keep track of the number of shifts (eight total) that have occurred for each serial transfer (the variable ID_SHIFT_COUNTER_TC in Figure 2-8 is asserted when the counter is down counting). A final counter, ID_CLK_COUNTER, was used to divide the system clock into the desired serial clock frequency.

The serial clock generation was performed by driving ID_CLK_COUNTER at the system clock frequency of 32 MHz. ID_CLK_COUNTER was loaded with a value of sixteen, and allowed to count down to zero (the variable ID_CLK_COUNTER_TC in Figure 2-8 is asserted when the counter is done counting down to zero). In this manner, ID_CLK_COUNTER will finish counting from sixteen to zero at an approximate frequency of 0.5 MHz. The serial clock was then generated by toggling the state of a flip flop every time ID_CLK_COUNTER finished a counting cycle, generating the desired 1 MHz serial clock frequency (the actual signal is less than 1 MHz to satisfy timing requirements).

The ID_IDLE state waits for a serial transfer to initiate. If a write cycle is initiated by the assertion of ID_WRITE, the shift register is loaded with the data from the PCB
Identification Data Register, and all counters are loaded with their respective values. The FSM then transitions to the ID_WRITE_LOW state. If a read cycle is initiated, the counters are loaded and the FSM transitions to the state ID_READ_LOW.

The states ID_WRITE_LOW and ID_WRITE_HIGH together make up one cycle of the serial clock. The ID_WRITE_LOW state corresponds to the low portion of the clock cycle, and ID_WRITE_HIGH to the high portion. During each state, ID_CLK_COUNTER is loaded with sixteen and allowed to count down to zero. In the state ID_WRITE_LOW, once ID_CLK_COUNTER has completed counting, the FSM transitions to the state ID_WRITE_HIGH. In the state ID_WRITE_HIGH, once ID_CLK_COUNTER has completed counting, a check of ID_SHIFT_COUNTER is performed. If another data bit needs to be shifted out to the E²PROM, then the data in the shift register is shifted, ID_SHIFT_COUNTER is decremented by one, and the FSM transitions back to ID_WRITE_LOW for another serial clock cycle. If the data has been completely shifted, then the FSM transitions to the state ID_SDOUT_LAG.

The states ID_READ_LOW and ID_READ_HIGH together make up one cycle of the serial clock. During each state, ID_CLK_COUNTER is loaded with sixteen and allowed to count down to zero. In the ID_READ_LOW state, once ID_CLK_COUNTER has completed counting, a new bit is shifted into the shift register from the serial E²PROM, and the FSM transitions to ID_READ_HIGH. In the ID_READ_HIGH state, once ID_CLK_COUNTER has completed counting, the contents of the shift register are copied into the PCB Identification Data Register, and a check of the counter ID_SHIFT_COUNTER is performed. If another data bit needs to be shifted in from the E²PROM, then ID_SHIFT_COUNTER is decremented by one, and the FSM transitions back to ID_READ_LOW for another serial clock cycle. If the data has been completely shifted in, then the FSM transitions to the state ID_SDOUT_LAG.

The ID_SDOUT_LAG state provides an extra half low serial clock cycle to signify the end of a serial transfer to the E²PROM, as suggested in [9]. ID_CLK_COUNTER is loaded with a value of sixteen and allowed to count down to zero, ID_LAG_DONE (which signifies the end of a serial transfer) is asserted, and the FSM waits until both ID_WRITE and ID_READ have become deasserted. Once this has occurred, the FSM
will transition back to the ID_IDLE state to wait for another serial transfer to the E²PROM PCB Identification Device to occur.

The logic used to implement the ID_CLK_COUNTER FSM is shown in Appendix C.2.

2.8 Control Module Timing Analysis

Timing analysis was performed on the Control Module logic to verify the design. A timing diagram showing how the main Control Module read and write FSMs function typically is shown in Figure D-1. Table D.1 details the definitions of the timing parameters. A second timing diagram showing the operation of the other EPLD control read and write FSMs is shown in Figure D-2. Table D.2 details the definitions of the timing parameters.

All of the EPLD timing parameters were measured with the Altera EPLD simulation software, which provides minimum and maximum timing guarantees. Hardware timing parameters were taken from their respective data books.

2.8.1 Main Control Register EPLD Reads/Writes

Figure D-1 shows the operation of the main control register read and write FSMs. It shows the reading of an arbitrary register, and then the writing of data to an arbitrary register.

Control Reads

The period between 0 ns and 650 ns corresponds to a control register read operation. Initially the READFSM is in the READIDLE state, and all signals are driven to an inactive state. The READFSM responds to the control read strobe, CTLRDN. The address of the location to be read is simultaneously driven onto the control address bus, CTLADDR, by the GBC card according to the timing guarantees of Figure 2-4. The address is latched by a transparent latch as discussed in Section 2.1. The signal ADDRLEN serves as a latch enable signal for the transparent latch. Whenever
ADDRLEN is driven low, the transparent latch will latch the data that is on its inputs on the falling edge of ADDRLEN.

CTRLRDN is synchronized by two registers to the system clock to prevent metastability (dual-rink synchronization) to produce the CTRLDNSYNC signal. This synchronized signal is used by the FSM and is also passed on to the other control read FSMs. Once the CTRLDNSYNC signal is asserted (driven to a logic low), the READFSM will transition to the READACK state. In the READACK state, the READFSM will drive the output enable for the control data bus, CTLLOOEOUTN and CTLHIOEOUTN (for the low and high bytes, respectively), to a logic low, drive the value in the selected register onto the data bus (CTLDATA), and assert the control acknowledge signal (ACKN) signifying that valid data from the chosen register is available on the control data bus. The READFSM will continue driving the data in this state until CTRLDNSYNC is deasserted. The READFSM will then transition back to the READIDLE state to await another control read operation.

It is fairly easy to verify that the timing diagram and parameters in Figure D-1 and Table D.1 satisfy the timing constraints in Figure 2-4. The relevant constraints are $t_{ACK\_ON}$, $t_{ACK\_OFF}$, $t_{DV}$, $t_{DZ}$, and $t_{DH}$. By matching the constraints with the appropriate timing parameters in Table D.1, it can be verified that the Control Module design meets all the control read timing constraints.

Control Writes

The period between 650 ns and 1250 ns corresponds to a control register write operation. Initially the WRITEFSM is in the WRITEIDLE state, and all signals are driven to an inactive state. The WRITEFSM responds to the control write strobes, CTLWRLO and CTLWRHIN (for the low and high byte writes, respectively). The address is presented on the CTLADDR bus and latched by the ADDRLEN signal in the same manner as a control read operation. In addition, the data to be written to the chosen control register is driven on the control data bus, CTLDATA, by the GBC card according to the timing guarantees presented in Figure 2-2.

The control write lines are also synchronized by two registers to the system clock to
prevent meta-stability. Once the synchronized control write lines are asserted (driven
to a logic low), the WRITEFSM will transition to the WRITEDATA state. In this
state, the WRITEFSM will write the data on the control data bus into the chosen
control register. The WRITEFSM will then transition to the WRITEACK state,
where it will assert the control acknowledge signal, ACKN, signifying that the data
on the control data bus has been written to the chosen register. The WRITEFSM
will continue to assert ACKN until both the control write lines are deasserted (driven
to a logic high), and then it will transition back to the WRITEIDLE state to await
further control write operations.

It is fairly easy to verify that the timing diagram and parameters in Figure D-1
and Table D.1 satisfy the timing constraints in Figure 2-2. The relevant constraints
are \( t_{\text{ACK,ON}} \) and \( t_{\text{ACK,OFF}} \). By matching the constraints with the appropriate
timing parameters in Table D.1 it can be verified that the Control Module design
meets all the control write timing constraints.

### 2.8.2 Other Control Register EPLD Reads/Writes

Figure D-2 shows a timing diagram of a control read and write for a Built-in Test
control register. The period between 0 ns and 340 ns corresponds to a control read
operation, and the period between 340 ns and 800 ns corresponds to a control write
operation. The timing diagram is very similar to the one shown in Figure D-1, and
is representative of the operation of the other EPLD control read and write FSMs as
discussed in Sections 2.4.3 and 2.5.2.
Chapter 3

Receiver Module

The Receiver Module is responsible for receiving both channels of fiber optic input from outside sources, and processing the data streams. The Receiver Module sequences the 8-bit serial transmissions into 16-bit parallel TTL words, and sets error and status bits in control registers when the appropriate conditions are satisfied. The Receiver Module then puts data from both channels into buffers in preparation for transmission to the rest of the Data Multiplexer II system.

Figure 3-1 shows a functional block diagram of the Receiver Module. Each channel consists of an 8-bit serial fiber optic input stream that is first received with a fiber optic data link receiver. The data stream is then formatted by a TAXIchip receiver, that converts the 8-bit serial fiber optic data stream into a 8-bit parallel TTL byte. Each byte is then passed through an input processor that sequences the bytes into 16-bit words, logs errors, and checks the status of the data stream. A separate input processor was implemented for each channel. The input processors are identical in functionality and design, except for a jumper that indicates which module is Channel 1 and which is Channel 2. Once the words have been processed, they are placed in buffer memory, where they are stored until they are ready to be transmitted by the Data Module.
3.1 Hardware Implementation

Figures B-1 and B-2 show the schematic diagrams for the Channel 1 and Channel 2 sections of the Receiver Module, respectively. Signals are designated CnNAME to differentiate between signals belonging to each module (the 'n' represents the channel number). Active low signals have an N appended to the signal name.

3.1.1 Receiver Circuitry

The receiver circuitry is shown in sections A through C of Figures B-1 and B-2, and consists primarily of an AT&T ODL 200 Series Lightwave Data Link and a Transparent Asynchronous Xmitter-Receiver Interface chip (TAXIchip). The Lightwave Data
Link converts serial fiber optic transmissions to serial electrical signals compatible with a TAXIchip. The TAXIchip converts serial fiber optic transmissions to parallel TTL data bytes.

The Lightwave Data Links are designated J1 and J2, manufacturer part number 1352N. They receive the serial fiber optic input transmissions from outside sources and covert them to differential signal levels. These signals are designated CnDATA and CnDATAN. Power to the Lightwave Data Links was filtered to prevent noise from contaminating the system and degrading system performance as suggested in [5]. The power filtering network is a simple inductor/capacitor (LC) network, and is shown in section A2 of Figures B-1 and B-2.

The Lightwave Data Links also provide a flag signal, or signal detect, that indicates when data transmissions are present on the input data stream. These are designated CnFLAG and CnFLAGN. During idle periods, the fiber optic input data streams will contain idle fill, so as not to deactivate the signal detect. The differential CnFLAG and CnFLAGN signals are converted to TTL signal levels using a precision comparator, manufacturer part number AD790JR. This comparator is shown in sections A1 and B1 of Figures B-1 and B-2. The comparator is a simple operational amplifier circuit that converts a differential signal to a TTL logic level signal. The output of the comparator is designated CnSIGDET.

The TAXIchips are are shown in sections C1 and C2 of Figures B-1 and B-2, manufacturer part number AMD7969-175JC, and are designated U15 and U16. The TAXIchips are capable of receiving up to 140 MBits per second (17.5 MBytes per second) at 17.5 MHz, satisfying the input data rate requirements of 15 MBytes per second. The TAXIchip converts 1 MByte of serial data per 1 MHz of its input clock [1], so to achieve an input rate of 15 MBytes per second, an input clock of 15 MHz, Cn15MHZ, derived from an external crystal oscillator (G1 and G2) was supplied to drive the TAXIchip. Power and ground to the TAXIchip were decoupled to prevent noise from contaminating the system and degrading system performance following the suggested procedure in [1, p. 73]. The decoupling circuitry is shown in section B2 and C2 of Figures B-1 and B-2.
The input of the TAXIchip is connected to the output of the Lightwave Data Link. The TAXIchip and the Lightwave Data Link are AC-coupled, as specified in [1, p. 69]. Resistors R1, R2, R6, R7, R8 and R9 provide line terminations for the coupling of Channel 1, and were derived using Equations 3.1 to 3.5 [1, p. 69]. For simplicity, the derivation is shown for Channel 1 only, but the Channel 2 terminations were derived in the same manner. Zo is the impedance of the DATA and DATAN traces, which is typically 100 ohms, and \( V_{bb} \) is the midpoint of the Lightwave Data Link signal swing as specified in [5].

\[
Z_o = R6 \times R7 / (R6 + R7) = R1 \times R2 / (R1 + R2)
\] (3.1)

\[
V_{bb} = VCC \times R2 / (R1 + R2)
\] (3.2)

\[
V_{bb} = 3.7V = 5 \times R2 / (R1 + R2)
\] (3.3)

\[
186 < R8 < 7.2 \times Z_o
\] (3.4)

\[
R8 = R9; R1 = R7; R2 = R6
\] (3.5)

Using these design equations, R1 was found to be 130 ohms, R2 was found to be 392 ohms, and R8 was found to be 301 ohms. The AC-coupling circuitry is shown in section B2, B3, C2 and C3 of Figures B-1 and B-2. Resistors with a maximum 5% tolerance were used in fabricating the actual TAXI-ISM printed circuit board, with a maximum power dissipation of 1/10 watts.

The TAXIchip outputs several signals that are used by the input processor to process the data stream. The 8-bit parallel input stream is provided by the data bus \( CnTDn[7..0] \), and is the formatted 8-bit input data stream to the TAXI-ISM after it has passed through the Lightwave Data Link and the TAXIchip. The 4-bit parallel command bus, \( CnTCn[3..0] \), is used to signal gaps in data transmission corresponding
to idle fill between real data transmissions. Any value other than zero on the parallel command bus will result in a TAXIchip synchronization flag. Timing information for the parallel data and command bus is provided by the signals CnTDSTRB and CnTCSTRB. Data is valid on the rising edge of these signals [1], which are used to latch the data on the buses.

In addition to the data signals, the TAXIchip also provides the control signals CnTCLK and CnTVLTN. CnTCLK is a TAXI specific clock which is synchronized to the serial data that is received from the Lightwave Data Link. This clock signal is derived from the input clock, Cn15MHZ [1], resulting in a 15 MHz clock that can be used as a system clock by the input processor to process the data stream. Error status is provided by the CnTVLTN signal. An error during the receiving of the fiber optic input stream will trigger the CnTVLTN signal, signalling that an error has occurred during the formatting by the TAXIchip.

### 3.1.2 Input Processor Circuitry

The input processor circuitry is shown in section D of Figures B-1 and B-2. The input processor is encapsulated using the logic capabilities of an Altera EPLD, manufacturer part number EPM7192GC160-12. This EPLD is labeled U13 for Channel 1, and U5 for Channel 2.

The input processor receives the 8-bit data stream from the TAXIchip, and sequences it to form a 16-bit data stream, CnFDn[15..0]. This data stream is then stored in buffer memory.

The clock for the input processor EPLDs is the CnTCLK signal provided by the TAXIchip, instead of the normal system clock that the rest of the TAXI-ISM system operates on. The reason for this is that the input processor manipulates the data bytes output by the TAXIchip. The data stream is synchronized to the CnTCLK signal, so it is appropriate for the input processor to use CnTCLK as its clock source.
3.1.3 Buffer Memory Circuitry

The buffer memory was implemented by First-In-First-Out (FIFO) buffers. Two 8-bit 16 KByte FIFOs were used to store the 16-bit data stream, manufacturer part number IDT7206L15J. The buffers are designated U11 and U12 for Channel 1, and U1 and U2 for Channel 2. U11 and U1 contain the low byte for each channel, and U12 and U2 contain the high byte. These are shown in section E of Figures B-1 and B-2.

The FIFO buffer memory also supplies two status signals, CnFEFN and CnFHFN. These signals indicate when the buffer memory is empty or half full, respectively, and are used by the Data Module to implement the data arbitration circuitry.

Data is read from the FIFO buffers and latched into registers. These registers are attached to a bus, and are equipped with output enables that allow each register to drive its contents onto the bus when its output enable is asserted. The registers were implemented using octal D-type flip flops, manufacturer part number IDT74FCT574ATSO. The Channel 1 buffer memory latches are designated U11 for the low byte, and U12 for the high byte. The Channel 2 buffer memory latches are designated U1 for the low byte, and U2 for the high byte. These are also shown in section E of Figures B-1 and B-2.

3.2 Input Processing

The input processor includes four primary functions: perform frame synchronization, process or sequence the data stream bytes from the TAXIchip into 16-bit words for the Data Module, log and respond properly to the status of the receiver circuitry, and log errors that occur in the receiver circuitry.

The input processor logic is shown in the EPLD file 12847 in Appendix C.1. This file contains the input processor, as well as some additional logic for the Control Module. This file is a generic file that was used to implement the Receiver Modules for both channels. An input, JUMPER, differentiates between the channels. If JUMPER is set to a logic high, then the Channel 1 functions are selected. If JUMPER is set to a logic low, then the Channel 2 functions are selected.
The relevant inputs to the input processor are: FIFOFULLN, SIGDET, TCLK, TVLTN, the 4-bit parallel command bus TCn[3..0] and its strobe TCSTRB, and the 8-bit parallel data bus TDn[7..0] and its strobe TDSTRB. These are the internal EPLD signal names and are specified as inputs in the appropriate sections of the file 12847.

The relevant outputs of the input processor are the 16-bit FIFO data bus FIFO-DATA[n][15..0], and the control signals FIFORSTN, FIFOWRTN, and TAXIRSTN. These are the internal EPLD signal names and are specified as outputs in the appropriate sections of the file 12847.

3.2.1 Frame Synchronization

The input data stream arrives in frames of data. These frames are delimited by frame synchronization bytes. These bytes are specified by the Channel 1 or 2 Frame Synchronization Byte n Registers. Any number between one and four bytes can be chosen by setting bits 4 and 3 to the appropriate values in the Channel 1 or 2 Control Register as specified in Section 2.3. The input processor examines the input data stream before it is sequenced into a 16-bit output data stream. Once it finds a sequence of the appropriate number of frame synchronization bytes, it asserts the frame synchronization detect bit in the appropriate Channel Status Register.

The logic for the input processor frame synchronization is shown in Appendix C.1. Combinational logic variables FSnMATCHL and FSnMATCHH are used to indicate if the low and high four bits of the current data byte match the low and high four bits of the Channel 1 or 2 Frame Synchronization Byte n Register.

An independent finite state machine, designated FSYNC, uses these variables to see if the input data stream contains the correct sequence of frame synchronization bytes. FSYNC first looks for a match for the first byte, then sequentially looks for the second byte, third byte, and finally the fourth. At any time, if a byte does not match the appropriate frame synchronization byte register in the sequence, the finite state machine resets and starts the search from the beginning. A finite state machine diagram for FSYNC is shown in Figure 3-2.
The valid frame synchronization bit in the Channel 1 or 2 Status Register is set by comparing the state of FSYNC and the number of bytes chosen to use for frame synchronization in the Channel 1 or 2 Control Register. If only one byte is selected and FSYNC has found the first frame synchronization byte (FSYNC is in the SYNC1 state), then valid frame synchronization is found and the input processor will assert the FSYNCDET signal. A check for two, three or four bytes is performed in the same manner, and FSYNCDET is asserted if any of the checks produces a valid result.

### 3.2.2 Byte-to-Word Sequencing

Another function of the input processor is to sequence the incoming 8-bit data stream into a 16-bit data stream. The input processor takes two sequential bytes in the data
stream, and constructs a 16-bit word. The choice between whether the first byte is the high or low byte of the output word is controlled by bits 1 and 0 in the Channel 1 or 2 Control Register. If the input processor byte-to-word sequencing is disabled, then no bytes will be sequenced.

The input processor byte-to-word sequencing can be reset by the detection of a valid TAXIchip synchronization. Bit 2 of the Channel 1 or 2 Control Registers controls this option. TAXIchip synchronization is provided by the TAXIchip on the 4-bit parallel command bus, TCn[3..0]. At any time, if this bus does not contain 0b0000, then valid TAXIchip synchronization is detected by the Receiver Module. If TAXIchip synchronization is enabled and valid TAXIchip synchronization is detected, the input processor will set bit 0 in the appropriate channel status register, and reset the byte-to-word sequencing.

The logic for the input processor byte-to-word sequencing is shown in the logic file contained in Appendix C.1.

### 3.3 Timing Diagram

Timing analysis was performed on the Receiver Module logic to verify the design. A timing diagram showing how the Receiver Module input processor and control read/write FSMs function typically is shown in Figure D-3. Table D.3 details the definitions of the timing parameters.

All of the EPLD timing parameters were measured with the Altera EPLD simulation software, which provides minimum and maximum timing guarantees. Hardware timing parameters were taken from their respective data books.

The timing diagram in Figure D-3 shows how the Receiver Module control read and write FSMs function, as well as how the input processor functions. The period between 0 ns and 700 ns illustrates a control read operation of the channel registers; the period between 1200 ns and 1900 ns illustrates a control write operation of the channel registers. The discussion of how the control read and write FSMs is similar to that given in Section 2.8, and will be omitted here for brevity.
The period between 700 ns and 1200 ns illustrates how the Receiver Module receives and processes the fiber optic input stream. Four data bytes are being processed into two 16-bit data words and stored in the channel buffer memory. The TAXIchip receives the serial input data stream and converts it to a parallel TTL data byte as discussed in Section 3.1.1. The TAXIchip then drives each data byte sequentially onto the TAXI data bus, TD, as shown in Figure D-3. All the values in Figure D-3 are given in hexadecimal.

The TAXI data strobe clocks the data into the input processor by providing timing information according to the timing parameter $T_{tds}$. The input processor takes two adjacent data bytes and combines them into a single 16-bit data word, which it then writes to buffer memory. The byte-to-word sequencing mode can be chosen by bits 1 to 0 in the Channel Control Register as discussed in Section 3.2.2. Most significant byte first is simulated in Figure D-3. In the timing diagram, the input processor takes the input data stream (0x60, 0x01, 0x61, 0x11) and constructs the word sequence (0x6001, 0x6111). The input processor then writes the words into buffer memory using the FIFO write strobe, FIFOWRTN. The signals FCYCLE and TCLKDEL are used to combinatorially generate the FIFOWRTN strobe according to the logic shown in Appendix C.1.
Chapter 4

Data Module

The Data Module generates the output data stream for the TAXI-ISM. It takes the 16-bit processed input stream for each channel from the Receiver Module and arbitrates between the two channels to pick a single 16-bit data word from one channel. A round-robin arbiter with priority protocols was used to arbitrate between the data channels. A normal round-robin arbiter strategy alternately gives each channel a turn to write data; if either channel does not have any data to write, it will give up its slot to the other channel. During periods of heavy traffic, the arbiter will give priority to the channel that is experiencing a heavier volume of traffic. The Data Module then generates an address and a 4-bit command code for the data word, and writes the word along with the corresponding address and command code to buffer memory on the host GBC card through the Data Interface. This represents the end of the TAXI-ISM’s involvement with the input stream, and it will continue to process further data.

An additional function of the Data Module is to perform tests of the Data Interface between the Data Module and the GBC card. The Data Module includes a Built-in Test (BIT) function that generates known arbitrary 16-bit data patterns that may be written in place of the output data stream to the GBC card buffer memory. The BIT function is capable of generating a 16-bit alternating pattern (each bit in the pattern alternates between a 1 and a 0) or a pseudo-random noise pattern.

A block diagram of the Data Module is shown in Figure 4-1. The arbiter chooses
which data stream to write to the GBC card during normal operation. During BIT operation, data from the arbiter is ignored, and Built-in Test patterns are written to the GBC card by the BIT function. The address associated with each data sample is generated by the address generator. An additional function of the Data Module is to record the ending address for each channel. The ending address is the last address to be written to the GBC card by the Data Module, incremented by one. These addresses can then be used in length counts, recording the total number of bytes that each channel processed in the previous CPI. The ending address register controller handles these ending address register issues.

4.1 Data Module Input and Output Signals

Table 4.1 shows the input and output signals that make up the Data Interface between the Data Module and the GBC card. Active low signals are followed by the letter N, indicating that they operate according to active low signal levels.

The sample data bus, SD[15..0], contains the 16-bit output data stream from the TAXI-ISM. This output data stream contains data from both input channels, and
can operate with data rates up to 30 MBytes/sec or 15 MWords/sec.

The sample address bus, SA[21..0], contains the corresponding address generated for each data sample. The GBC card buffer memory contains a separate page of memory for each channel. Data for the Channel 1 input stream is written to the address space between 0 and 2,097,151 inclusive. Data for the Channel 2 input stream is written to the address space between 2,097,152 and 4,193,303 inclusive. This memory space spans the full address space addressable using the 22-bit sample address bus. If the address generators for each channel reach the maximum address in their dedicated address space, then an address overflow condition occurs.

The sample control bus, SC[3..0], contains the corresponding command code generated for each data sample. Valid command codes are 0b0000 or 0b0001. After a new Coherent Processing Interval (CPI) has started, the TAXI-ISM will reset its address counters to the base address of the address space for each channel, record the next address to write data to the GBC card for each channel (ending address), and will perform a write of the command code 0b0001 within 500 ns of the start of the new CPI. This command code acknowledges to the GBC card that the TAXI-ISM has acknowledged and begun a new CPI. The data on the sample data and address buses is not relevant during this write, and may be driven to any value. The command codes will be set to 0b0000 in every other case, indicating normal operation. A falling edge on the CPISTART signal indicates the start of a new CPI.

The FIFOWRTN signal is the data write strobe for the Data Interface. Figure 4-2 shows the timing specifications for a data write by the TAXI-ISM to the GBC card.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD[15..0]</td>
<td>Sample data bus</td>
<td>Output</td>
</tr>
<tr>
<td>SA[21..0]</td>
<td>Sample address bus</td>
<td>Output</td>
</tr>
<tr>
<td>SC[3..0]</td>
<td>Sample command bus</td>
<td>Output</td>
</tr>
<tr>
<td>FIFOWRTN</td>
<td>Data write strobe</td>
<td>Output</td>
</tr>
<tr>
<td>FIFOFFN</td>
<td>GBC card memory overflow flag</td>
<td>Input</td>
</tr>
<tr>
<td>CPISTART</td>
<td>New Coherent Processing Interval (CPI) start flag</td>
<td>Input</td>
</tr>
</tbody>
</table>

Table 4.1: Data Module Inputs and Outputs
Figure 4-2 specifies the timing guarantees on input signals to the TAXI-ISM, as well as the output timing constraints that output signals must obey.

The FIFOFFN signal represents an overflow condition for the GBC card buffer memory. If the FIFOFFN signal is asserted by the GBC card, then the TAXI-ISM will perform at most one additional write to the GBC card buffer memory, and set bits 3 and 2 of the Error Register as specified in Section 2.3.4. The TAXI-ISM will then wait between 500 and 1000 ns before attempting to perform additional writes to the GBC card, in order to allow some time for the GBC’s buffer memory to empty.

### 4.2 Hardware Implementation

The Data Module is completely implemented using the logic capabilities of three Altera EPLDs, manufacturer part number EPM7192GC160-12. The primary EPLD, 12848, is the Arbiter EPLD. This EPLD is designated U22, and is shown in section D and E of Figure B-3. The logic for the device is shown in Appendix C.3.1.
A second EPLD is responsible for managing the ending address registers discussed in Section 2.3. After a new CPI is initiated, the Data Module is responsible for recording the next address for each data word to be written to on the GBC card for each channel. This EPLD is called the Ending Address EPLD, 13035, and it records the ending address for each channel in the appropriate registers. This EPLD is designated U21, and is shown in section E and F of Figure B-3. The logic for the device is shown in Appendix C.3.2.

A final EPLD is responsible for controlling the BIT functions of the Data Module. This EPLD is called the BIT EPLD, and is designated U20, 13022. It is shown in section C and D of Figure B-3. The logic for the device is shown in Appendix C.3.3.

### 4.3 Data Arbitration

The Data Module is responsible for arbitrating between the processed input data streams of each channel, and choosing only one 16-bit data sample from either channel to write to the GBC card. The Data Module implements a round-robin arbiter with priority protocols. A normal round-robin arbiter strategy simply gives each channel a turn to write data; first Channel 1 is given the opportunity to write data, then Channel 2, and so on and so forth. If either channel does not have any data to write, it will give up its slot to the other channel.

#### 4.3.1 Priority Protocols

During periods of heavy traffic, the arbiter will give priority to the channel that is experiencing a heavier volume of traffic. The Data Module reads data from the FIFO buffer memories of the Receiver Module. It monitors the half-full and empty status signals of each channel’s buffers, and controls the FIFO read and output enable signals. It also monitors the status of the address generators for each channel, in case of address overflow. The half-full signal indicates if the memory buffer is either half or more than half full. If either channel is singularly experiencing a buffer half full condition, the arbiter will devote full attention to that channel, and will read only
from that channel until the buffer drops below the half full status. If both channels are experiencing half full conditions, then the arbiter will implement a normal round-robin strategy. During address overflow conditions, the arbiter will continuously read and throw away data from the buffers. This allows the arbiter to write data for non-full channels and help out full channels concurrently, as well as to prevent the overwriting of data already in the GBC card's buffer memory.

4.3.2 Arbiter Finite State Machine Overview

The Data Module arbiter is controlled by logic in the arbiter EPLD, 12848. The arbiter is primarily implemented by a finite state machine (FSM). The FSM has states to handle address generator overflows, new CPIs, and GBC buffer memory overflows. The state bits are actual signals that the FSM uses for address generation, reading from the buffer memories, and writing to the GBC card's buffer memory.

Figure 4-3 shows the FSM state diagram. For simplicity, only the states pertinent to Channel 1 are shown in Figure 4-3. Their Channel 2 counterparts are similar in design and logic. Table 4.2 shows the FSM states and their associated state bit levels. The state bits correspond to the signals: ADDRINC, FIFO2RDN, FIFO2OENFSM, FIFO1RDN, FIFO1OENFSM, GBCFIFOWRTN, SACK2, SACK1. A 0 in Table 4.2 represents a logic low, and a 1 represents a logic high.

The state bit ADDRINC is used by the address generators to increment the addresses for each channel. If it is a 1, then the appropriate address will increment by one on the next clock cycle, otherwise if it is a 0 then the address will not change. The state bits FIFO2RDN and FIFO1RDN are used to read data from the Channel 2 and 1 buffer memories, respectively. A low to high transition of these bits will read data from the appropriate buffer memory and latch it in the appropriate buffer latch as described in Section 3.1.3. The state bits FIFO2OENFSM and FIFO1OENFSM are the output enables for the Channel 2 and 1 buffer memories, respectively. When these bits are low, the appropriate buffer latch will drive its contents on the sample data bus, SD[15..0]. The state bit GBCFIFOWRTN serves as the Data Interface write strobe, FIFOWRTN, that writes data to the GBC buffer memory. The state
<table>
<thead>
<tr>
<th>State</th>
<th>State Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>0 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>NEWCPI1</td>
<td>0 1 1 1 1 1 1 0 1</td>
</tr>
<tr>
<td>NEWCPI2</td>
<td>0 1 1 1 1 0 1 1 1</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>0 0 1 0 1 1 1 1 1</td>
</tr>
<tr>
<td>OVER1</td>
<td>0 1 1 0 1 1 1 1 1</td>
</tr>
<tr>
<td>OVER2</td>
<td>0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>SEL1LOV2</td>
<td>0 0 1 0 0 1 1 1 1</td>
</tr>
<tr>
<td>SEL1HOV2</td>
<td>1 1 1 1 0 0 1 1 1</td>
</tr>
<tr>
<td>SEL2LOV1</td>
<td>0 0 0 0 1 1 1 1 1</td>
</tr>
<tr>
<td>SEL2HOV1</td>
<td>1 1 0 1 1 0 1 1 1</td>
</tr>
<tr>
<td>SEL1L</td>
<td>0 1 1 0 0 1 0 1 1</td>
</tr>
<tr>
<td>SEL1H</td>
<td>1 1 1 1 0 0 0 1 1</td>
</tr>
<tr>
<td>SEL2L</td>
<td>0 0 0 1 1 1 1 1 0</td>
</tr>
<tr>
<td>SEL2H</td>
<td>1 1 0 1 1 0 1 1 0</td>
</tr>
<tr>
<td>GBCFULL</td>
<td>0 0 1 0 1 1 0 0 0</td>
</tr>
<tr>
<td>GBCFULL1</td>
<td>0 1 1 1 1 1 0 0 0</td>
</tr>
</tbody>
</table>

Table 4.2: Data Module Arbiter FSM State Definitions

bits SACK1 and SACK2 serve as acknowledges that the FSM uses to indicate the completion of a write by Channel 1 and 2, respectively. If SACK1 is a 1, then Channel 1 has just completed a write and Channel 2 may write data. These two bits are only relevant during non-overflow conditions, and are set to 1 in most other states.

In the IDLE state, all the state bits are set to an inactive level as shown in Table 4.2. ADDRINC is set to 0 to prevent address increments. The FIFO read strobes, FIFO1RD and FIFO2RD, and the Data Interface write strobe, GBCFIFOWRT, are driven to an arbitrary level of 1. FIFO2ENFSM and FIFO1ENFSM are driven to a level of 1, deasserting the output enables of the Channel 2 and 1 memory buffer latches, respectively. SACK1 and SACK2 are driven to a level of 1, allowing either channel to commence writing once it has any data available.

Two logic variables, RDY1 and RDY2, signify when Channel 1 and 2 is ready or has data available to write to the GBC card. RDY1 is asserted if Channel 2 has just completed a write (signified by a logic high on the SACK2 state bit), Channel 1’s buffer memory is half full and Channel 2’s is not, Channel 2’s buffer memory is...
empty, Channel 1’s buffer is not empty and its address generator is not overflowing, the GBC card’s buffer memory is not overflowing, and the BIT functions are not being operated. Similar logic is implemented to assert RDY2.

4.3.3 Normal Round-robin Arbitration

The normal round-robin strategy alternates between Channel 1 and 2, allowing each channel to write a data word every other write cycle. During periods of heavy traffic, the channel with the heavier volume of data will be given priority in writing data to the GBC card. Priority is given based upon the half-full status condition of each channel’s buffer memory, and is handled by the RDY1 and RDY2 logic variables.
Each write cycle to the GBC card consists of two cycles of the FSM. During these two cycles, the current address from the address generators is driven onto the sample address bus, SA[21..0], data from the appropriate channel's Receiver Memory buffer memory latch is driven onto the sample data bus, SD[15..0], and a write to the GBC card is performed. Channel 1 data writes are performed using the states SEL1L and SEL1H, and Channel 2 writes using SEL2L and SEL2H.

The two FSM cycles correspond to a “low” state and a “high” state. Data is read from the Channel 1 buffer memory by the Channel 1 buffer memory read strobe, FIFO1RDN, and from the Channel 2 buffer memory by FIFO2RDN. A low to high transition on FIFO1RDN or FIFO2RDN reads data from the channel buffer memory, and latches it into the buffer memory latch as discussed in Section 3.1.1. The “low” state drives the buffer memory read strobes low, and the “high” state then drives the strobes high, reading and latching a data sample from the appropriate channel’s buffer memory until the next read occurs. This is verified by the level of the state bits as shown in Table 4.2.

Data is written to the GBC card using the signal FIFOWRTN. A low to high transition on FIFOWRTN writes data to the GBC card as shown in Figure 4-2. Since data is latched on the “high” or second cycle of the write, the FIFOWRTN signal should be high for the “low” state and low for the “high” state, as shown in Table 4.2. This way, data is written when the next write to the GBC begins, which causes a low to high transition of FIFOWRTN.

During normal operation, after Channel 1 completes its write, SACK1 is asserted with a logic high, driving RDY1 low and RDY2 high. The FSM will then branch from the SEL1H state to the SEL2L state, initiating a write for Channel 2. After Channel 2 completes its write, SACK2 is asserted, driving RDY2 low and RDY1 high. The FSM will then branch from the SEL2H state to the SEL1L state, beginning the round-robin arbitration strategy all over again. The FSM toggles between the SEL1L-SEL1H and SEL2L-SEL2H sequence. If either channel experiences a half-full condition, its ready variable (RDY1 or RDY2) will continue to be asserted and the other channel’s ready signal will become deasserted. The FSM will then stay only in the states belonging
to the channel whose ready variable is asserted. If both channels experience half-full conditions, then neither channel will be given priority and the RDY1 and RDY2 signals will toggle as in normal operation. This is implemented by the logic for RDY1 and RDY2 as shown in Appendix C.3.1.

If either channel or the GBC card experiences an overflow condition, the FSM will transition to the appropriate overflow state as shown by the state diagram in Figure 4-3.

### 4.3.4 New Coherent Processing Intervals (CPI)

The two states NEWCPI1 and NEWCPI2 are responsible for handling the arbitration issues during the beginning of a new Coherent Processing Interval (CPI). After the falling edge of the CPISTART signal, the FSM will branch to the NEWCPI1 state, which will drive the command code 0b0001 on the sample command bus, SC[3..0], indicating that a new CPI has begun. The FSM will then branch to NEWCPI2, which will continue to drive the sample command bus until the CPISTART signal is deasserted.

Data is written by the FIFOWRTN signal, which is high for NEWCPI1 and low for NEWCPI2, as specified by Table 4.2. The reasoning behind this is similar to that detailed in Section 4.3.3.

### 4.3.5 Channel Address Overflow Conditions

There are two types of overflow conditions that the Data Module arbiter must deal with. The first type of overflow is caused by the address generators for each channel. Each channel has a specific address space on the GBC card to write data to, and an address overflow occurs when the address generators have reached their maximum allowable address in the specified address space. Three scenarios of address overflows can arise during operation of the Data Module: a single channel can overflow while the other channel is idle, a single channel can overflow while the other channel still has data to write to the GBC card, or both channels can overflow. During an address
overflow, data is indiscriminately read from the overflowing channel and thrown away (i.e. not written to the GBC card). This prevents the overwriting of any data already in the GBC card at the same address location, since the address generators are overflowing and generating addresses that already have been used in the current CPI.

Two signals, OVFLOW2 and OVFLOW1, indicate when the address generators have reached their maximum allowable addresses for Channel 2 and 1, respectively. If either signal is asserted, the appropriate bits in the error registers are set as described in Section 2.3, and an address overflow condition occurs.

Single channel overflows while the other channel has no data to write to the GBC card are handled by the states OVER1 and OVER2. These states simply read and throw out data from the channel buffer memories until the overflow condition is rectified. Reading data from the buffer memories is accomplished by setting FIFO1RDN or FIFO2RDN to 0 as specified in Table 4.2, and then driving them to 1 by transitioning to the IDLE state (which has the state bits appropriately set). OVER1 reads data from Channel 1 when it experiences an address overflow and Channel 2 does not have any data to write. Similarly, OVER2 reads data from Channel 2 when it experiences an address overflow and Channel 1 does not have any data to write.

The state OVERFLOW handles address overflows for both channels simultaneously. OVERFLOW simply reads data from both Channel 1 and Channel 2, but doesn’t write it to the GBC card. This is accomplished by driving both FIFO1RDN and FIFO2RDN to a logic low as shown in Table 4.2. A state transition on the next cycle to the IDLE state drives both of these strobes to a logic high, which performs the actual read operation.

Address overflows for a single channel that occur while the other channel still has data to write to the GBC card are handled by reading data from both buffer memories, but only writing the non-overflowing channel’s data to the GBC card. The state sequence SEL1LOV2 and SEL1HOV2 handles overflows of Channel 2 while Channel 1 still has data to write. The functions of the state bits of SEL1LOV2 are simply a combination of those performed in the states SEL1L (as discussed in
Section 4.3.3 and OVER2, which drive both read strobes low. The functions of the state bits of SEL1HOV2 are simply a combination of those performed in the states SEL1H and OVER2, which drive both read strobes high to perform the reads but only enabling the Channel 1 buffer memory latch to drive data on the sample data bus to be written to the GBC card. Similarly, the state sequence SEL2LOV1 and SEL2HOV1 handles overflows of Channel 1 while Channel 2 still has data to write.

### 4.3.6 GBC Card Buffer Memory Overflow

The second type of overflow is caused by the GBC card. Data is taken from the TAXI-ISM and stored in buffer memory on the GBC card. If the GBC card buffer memory becomes too full, then the second type of overflow condition will occur. This is signified by a logic low on the FIFOFFN signal.

During GBC card buffer memory overflows, data is indiscriminately read from both channels and thrown away (i.e. not written to the GBC card). If a channel is empty, a read operation will have no effect. The Data Module will then wait between 500 and 1000 ns before attempting to write data to the GBC card again.

The state sequence GBCFULL to GBCFULL1 performs a simultaneous read of both channel buffer memories, but does not write the data to the GBC card. FIFO1RDN and FIFO2RDN are driven to a logic low in GBCFULL, then to a logic high in GBCFULL1, to perform a read operation as shown in Table 4.2. Equation 4.2 calculates the number of executions of the state sequence GBCFULL and GBCFULL1 that are needed to make the Data Module wait between 500 and 1000 ns before attempting another write to the GBC card. Equation 4.1 shows that eight full sequences of GBCFULL to GBCFULL1 is enough to satisfy Equation 4.2, given that the system clock for the Data Module is 32 MHz as discussed in Chapter 2.

\[
T_{FSM} = \frac{1}{CLK_{system}} = \frac{1}{32MHz} = 3.125 \times 10^{-8} \quad (4.1)
\]

\[
\frac{500}{2 \times T_{FSM}} \leq Number_{sequences} \leq \frac{1000}{2 \times T_{FSM}} \quad (4.2)
\]
A counter, GBCFULLCLK, is used to count eight sequential executions of the state sequence GBCFULL to GBCFULL1. After the eight sequences have been executed, if FIFOFFN is still a logic low (signifying that the GBC card buffer memory is still overflowing) the FSM will reset GBCFULLCLK and perform eight more sequences. Otherwise, the FSM transitions to the IDLE state, from which it determines its next course of action.

### 4.3.7 Arbiter Timing Analysis

Timing analysis was performed on the Arbiter EPLD logic to verify the design. A timing diagram showing how the Arbiter typically operates is shown in Figure D-4. Table D.4 details the definitions of the timing parameters. All of the EPLD timing parameters were measured with the Altera EPLD simulation software, which provides minimum and maximum timing guarantees. Hardware timing parameters were taken from their respective data books.

Figure D-4 shows the writing of data for each channel to the GBC card through the TAXI-ISM/GBC data interface. The timing diagram shows two writes for Channel 1, and then three writes for Channel 2. Simultaneous operation is simply a combination of the two, and is excluded for simplicity. The arbiter reacts to the two channel FIFO buffer memory empty flags, FIFO1EFN and FIFO2EFN. When these signals are low, the arbiter resides in the IDLE state. When these signals are high, the arbiter will transition to the correct state according to Figure 4-3.

**Channel 1 Writes**

The period between 0 ns and 400 ns in Figure D-4 corresponds to two writes of the Channel 1 data stream. Initially, the Arbiter FSM is in the IDLE state, and all signals are in the deasserted state. The FIFO1EFN signal transitions from the empty to non-empty state at approximately 100 ns, initiating arbitration. The current address for Channel 1 is 0x01ab45. The Arbiter FSM will respond by transitioning to the SEL1L state, since only Channel 1 is active. In the SEL1L state, the FSM drives FIFO1RDN
low, reading data from the Channel 1 FIFO buffer memory. The FSM then transitions to the state SEL1H.

In the SEL1H state, the Arbiter FSM begins the write cycle to the GBC card. It drives the address and data bus to the appropriate values, enables the output latch for the Channel 1 FIFO buffer memory, and drives GBCFIFOWRTN low to initiate the write. The FSM then transitions back to the SEL1L state, completing the write, and repeats the process again to start the second write to the GBC card. The process continues until the Channel 1 FIFO buffer memory empty flag is deasserted, and then the Arbiter FSM transitions to the IDLE state.

### 4.3.8 Channel 2 Writes

The period between 400 ns and 750 ns in Figure D-4 corresponds to three writes of the Channel 2 data stream. The operation of Channel 2 writes is very similar to that of Channel 1 writes, except that the Channel 2 status and control signals are used.

A few simple mathematical calculations reveals that the timing shown in Figure D-4 satisfies the timing constraints of Figure 4-2. The first three constraints in Figure 4-2 are on the data write strobe, GBCFIFOWRTN. The constraints specify that the write cycle time must be greater than 25 ns, with a minimum of 10 ns for the low and high portions of the write strobe. Figure D-4 shows that the write strobe lasts two cycles of the 32 MHz system clock, CLK. Since the CLK period is approximately 32 ns, these constraints are satisfied. The remaining constraints are on the rising edge of the write strobe. A minimum setup time of 6 ns and a minimum hold time of 3 ns are specified in Figure 4-2. These correspond to $T_{gsu}$ and $T_{ghd}$ in Figure D-4. Simple calculations reveal that the BIT functions provide more than enough setup and hold time to satisfy these constraints, verifying the design specifications of Figure 4-2.

### 4.4 Address Generation

The GBC card buffer memory contains a separate page of memory for each channel. The arbiter decides which channel to write data for. Data for the Channel 1 input
stream is written to the address space between 0 and 2,097,151 (0x1FFFFFFF) inclusive. Data for the Channel 2 input stream is written to the address space between 2,097,152 (0x200000) and 4,193,303 (0x2FFFFFFF) inclusive. This memory space spans the full address space addressable using the 22-bit sample address bus. If the address generators for each channel reach the maximum address in their dedicated address space, then an address overflow condition occurs.

The address generators for both channels are located in the Arbiter EPLD, 12848. The address generator for Channel 1 is implemented using a twenty-two bit counter, and the address generator for Channel 2 using a twenty-three bit counter. After a falling edge of the CPISTART signal, a new Coherent Processing Interval (CPI) begins. The address generators are reset at the beginning of a new CPI after the current data has completed being written to the GBC card. The Channel 1 address generator resets to 0x000000, and the Channel 2 address generator resets to 0x200000. An overflow for each counter is checked for by examining the leading bit in each counter. This is more efficient than performing a twenty-two bit comparison. The Channel 1 address has overflown if bit twenty-two of its address generator is a 1, and the Channel 2 address has overflown if bit twenty-three of its address generator one is a 1. Two variables, OVFLOW1 and OVFLOW2, record if an overflow has occurred for Channel 1 or Channel 2, respectively.

Each write cycle by the Data Module to the GBC card requires two cycles as discussed in Section 4.3.3. At the end of the second state in the write cycle, the appropriate address generator increments its counter by one in preparation for the next data to be written. The address generators use the state bits of the arbiter FSM to determine when to increment their counters. The state bits SACK1 and SACK2 signify the completion of a write by Channel 1 and Channel 2, respectively, and the state bit ADDRINC signifies that the address generators should increment their counters by one (as discussed in Section 4.3.2). The Channel 1 address generators increment if SACK1 and ADDRINC have a logic high, indicating that Channel 1 has just completed a write. Similarly, the Channel 2 address generators increment if SACK2 and ADDRINC have a logic high.
The logic for the address generators is shown in Appendix C.3.1.

4.5 Ending Address Registers

The Data Module implements two separate address generators for each channel as discussed in Section 4.4. These address generators provide a latch function that records the ending address for each channel. The ending address is the last address to be written to the GBC card by the Data Module, incremented by one. These addresses can then be used in length counts, recording the total number of bytes that each channel processed in the previous CPI. The latch registers are written upon detection of a falling CPISTART edge, indicating a new CPI, and are accessible through the Control Module at any time during the new CPI without affecting data write operations to the GBC card.

The latch registers are called Ending Address Registers, as discussed in Section 2.3. Two separate registers were used to implement the Ending Address Registers, a low register containing the low sixteen bits and a high register containing the top bits (six for Channel 1 and seven for Channel 2 as discussed in Section 2.3.13).

An independent EPLD, 13035, maintains the ending address registers and records the ending address for each channel in the appropriate registers. This EPLD is designated U21, and is shown in section E and F of Figure B-3. The Ending Address EPLD, 13035, “snoops” the sample address bus, SA[21..0], and stores each address in a temporary register (one for each channel). The EPLD checks the most significant bit (bit 21) of each address written to the GBC card. If the most significant bit is a 1, then the address must belong to Channel 2, since Channel 2’s address space is between 2,097,152 (0x200000) and 4,193,303 (0x2FFFFF) inclusive. If the leading bit is a 0, then the address must belong to Channel 1, since Channel 1’s address space is between 0 and 2,097,151 (0x1FFFFFF) inclusive. The EPLD then waits for a new CPI to begin. The Arbiter EPLD, 12848, provides a synchronized CPISTART signal to the Ending Address EPLD, 13035. The Ending Address EPLD waits for a valid CPISTART signal from the Arbiter EPLD, and then copies the contents of each of
the temporary registers into the appropriate Ending Address Registers, adding one since the latch function stores the next address to write data to on the GBC card. The logic for the Ending Address EPLD is shown in Appendix C.3.2.

4.6 Built-in Test

The Data Module is also designed to provide a means to test the Data Interface between the TAXI-ISM and the GBC card, which the Data Module uses to write the 16-bit output data stream. This means is called the Built-in Test or BIT function. The BIT function generates a known test pattern, and writes the pattern to the GBC card using the Data Interface protocol shown in Figure 4-2. BIT data is similar to normal output data streams, possessing a data word, an address, and a command code. BIT data is written over the entire GBC memory space from 0 to 4,193,303 (0x2FFFFF), testing both of the channel address spaces.

Using this strategy, BIT patterns can isolate bugs in the system. If the BIT pattern is successfully written to and processed by the GBC card, then the Data Interface between the Data Module and the GBC card is functioning, and any problems that exist are narrowed down to the TAXI-ISM Control or Receiver Module. If the BIT pattern is not successfully written to the GBC card, then any problems are narrowed down to the Data Interface or the actual GBC card.

4.6.1 Implementation Concept

The Built-in Test function is implemented in the BIT EPLD. This EPLD is designated U20, 13022, and is shown in section C and D of Figure B-3. The logic for the device is shown in Appendix C.3.3.

The basic scheme for the BIT function is that during normal operation, the BIT EPLD appears transparent to the Receiver and Data Modules. During BIT operation, the BIT EPLD should shunt the input data streams, and replace them with a single BIT pattern. The BIT EPLD was designed to act as a channel FIFO buffer memory in the Receiver Module, complete with FIFO empty flag, FIFO full flag, FIFO half-
full flag, FIFO read strobe, and FIFO output enable signals. It was then inserted between the Channel 2 FIFO buffer memory and the Arbiter EPLD (12848), as depicted in Figure 4-4. The BIT EPLD receives the buffer memory status signals from the Channel 2 buffer memory, and supplies new status signals to the Arbiter EPLD. It also receives the buffer memory control signals from the Arbiter EPLD, and supplies new ones to the Channel 2 buffer memory. The Channel 2 buffer memory still thinks and behaves as if it is connected to the Arbiter EPLD, and vice versa.

During normal operation, the BIT EPLD simply passes all the signals in both directions without alteration, except for a minor propagation delay. During BIT operation, the BIT EPLD intercepts the signals in both directions. The BIT EPLD intercepts the Channel 2 buffer memory status signals, and drives the status signals to the Arbiter to a continuously active state. The FIFO empty flag and full flag are deasserted, and the FIFO half full flag is asserted. In this manner, the Arbiter EPLD will continue to read BIT data from the BIT EPLD with a high priority, thinking that it is the Channel 2 buffer memory. The BIT EPLD also intercepts the FIFO control signals from the Arbiter EPLD, and drives the control signals to the Channel 2 Buffer memory to an idle state. The FIFO read strobe and output
enable are both deasserted. The BIT EPLD then interprets the control signals from the Arbiter EPLD and provides data to the Arbiter EPLD in place of the Channel 2 buffer memory whenever a read strobe is detected and it is output enabled.

This strategy allows the BIT functions to examine the full functionality of the Data Module and GBC card, since the BIT EPLD is effectively pretending it is a Channel 2 input stream. It also allows the Data Module to operate in a normal mode, responding to new CPIs, address overflows, GBC buffer memory overflows, etc.

It also allows for increased modularity between subsystem components. The fact that the BIT EPLD emulates a channel buffer memory allows it to be removed from the TAXI-ISM completely, without affecting its operation. It also allows the easy integration of the BIT functions, without too much change in the rest of the system.

Two slight modifications had to be made to the Arbiter EPLD (12848) to accommodate the BIT EPLD. The first and most important change is to eliminate operation of the Channel 1 input data stream during BIT operation, since the BIT function utilizes only the Channel 2 data path. This was accomplished by deactivating the Channel 1 ready signal, RDY1. BIT functions are initiated by bit 0 of the Built-in Test Control Register, as described in Section 2.3.5. If bit 0 of the BIT Control Register is a 1, then BIT functions are enabled. The RDY1 signal was logically ANDed with the inverse of bit 0, so that during BIT operation, RDY1 will always be deasserted and the Arbiter EPLD will only try to read data from the Channel 2 data path.

The second modification concerns the address generators. BIT data is written over both Channel 1 and Channel 2 address spaces, from 0 to 4,193,303 (0x2FFFFF). But, the Channel 2 address generators operate between 2,097,152 (0x200000) and 4,193,303 (0x2FFFFF) only, and reset to 2,097,152 (0x200000), the base address of the Channel 2 address space. Since BIT functions use the Channel 2 data path, the Channel 2 address generators had to be modified to reset to 0 during BIT operations, and 2,097,152 (0x200000) during normal operations. This was accomplished by simply resetting the Channel 2 address generator to 0 at the beginning of BIT operation, and
having it reset to 0 after a new CPI begins as long as the Data Module remains in BIT operation. The Channel 2 address generator was allowed to count normally after that, encompassing the entire address space specified by the 22-bit sample address bus, SA[21..0]. If the BIT function is deactivated at any time, the Channel 2 address generator resumes normal operation, resetting to the base address of the Channel 2 address space.

### 4.6.2 Built-in Test Patterns

The BIT function has the capability of generating two test patterns: alternating and pseudo-random noise. The choice between the two patterns is controlled by bits 11 to 7 in the Built-in Test Control Register, as described in Section 2.3.5.

#### Alternating Test Pattern

The alternating pattern takes the base seed from the Built-in Test Base Pattern Register, and outputs a pattern consisting of the seed and its bit-wise inverse (logical complement), output alternately. For example, if the base pattern register contains 0xAAAA, then the BIT function will generate the pattern 0xAAAA, 0x5555, 0xAAAA, 0x5555 etc. The seed in the BIT Base Pattern Register is always the first value in the BIT pattern. The alternating pattern tests the capability of the data interface to handle rapidly changing data patterns, since the bits change each write cycle.

The generation of the alternating BIT pattern is implemented in the BIT EPLD, 13022. A variable, ALT, toggles every cycle between 1 and 0 and is used to determine which word to output, the seed or its inverse. If the alternating pattern is chosen by the BIT Control Register, the seed will be written to the GBC card if ALT is a 0, and its inverse will be written if ALT is a 1. The logic for implementing the alternating pattern is shown in Appendix C.3.3.
**Pseudo-random Noise Test Pattern**

The pseudo-random noise or PRN pattern represents real data streams better than the alternating pattern. Actual data patterns don’t change their bits according to an arbitrary pattern. Instead, the bits change in a natural, or random order. The BIT PRN pattern takes the seed in the Built-in Test Base Pattern Register, and uses it to generate a random 16-bit number sequence. The pattern is called pseudo-random since an initial seed must be supplied, and the pattern depends on this seed. The pseudo-random pattern produces a sequence of numbers that does not repeat until all $2^{16}$ possible numbers (since it is a 16-bit pattern) have been generated. Different seeds start the pattern in their respective locations, but identical seeds will produce the same pattern (hence it is only a pseudo-random pattern). A truly random sequence of numbers does not need a seed, and usually contains a few repetitions of previous numbers.

The pseudo-random number generator is implemented in the BIT EPLD, 13022, and is based on “primitive polynomials modulo 2 [8].” There are special polynomials whose coefficients are 0 or 1 that define a relation for generating a new random bit from $n$ previous ones. This relation is guaranteed to cycle through all possible sequences of $n$ bits (except all zeros) before it repeats, i.e. the seed will generate $2^{n+1} - 1$ more numbers before it repeats [8]. A degree $n$ polynomial will generate a $n$-bit pseudo-random pattern. Equation 4.3 from [8] shows the degree sixteen polynomial used to generate the 16-bit pseudo-random noise pattern. If the seed is contained in a 16-bit shift register, then a new bit can be generated using Equation 4.4 (for a number with bits numbered 1 to $n$) and shifted into the shift register. After $n$ iterations, the shift register will contain the next number in the pseudo-random number sequence.

$$x^{\text{new}} = x^{16} + x^5 + x^3 + x^2 + x^0$$  \hspace{1cm} (4.3)$$

$$x_0 = x_{16} + x_5 + x_3 + x_2 + x_0$$  \hspace{1cm} (4.4)$$

A more efficient algorithm based on Equation 4.3 was derived. This algorithm
Table 4.3: Pseudo-random Noise BIT Pattern Generator Algorithm

<table>
<thead>
<tr>
<th>Next State</th>
<th>Previous State</th>
</tr>
</thead>
<tbody>
<tr>
<td>( prn_0 )</td>
<td>( prn_4 ) XOR ( prn_8 ) XOR ( prn_{10} ) XOR ( prn_{11} ) XOR ( prn_{12} ) XOR ( prn_{13} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_1 )</td>
<td>( prn_0 ) XOR ( prn_2 ) XOR ( prn_3 ) XOR ( prn_9 ) XOR ( prn_{11} ) XOR ( prn_{12} ) XOR ( prn_{13} ) XOR ( prn_{14} )</td>
</tr>
<tr>
<td>( prn_2 )</td>
<td>( prn_1 ) XOR ( prn_3 ) XOR ( prn_4 ) XOR ( prn_{10} ) XOR ( prn_{11} ) XOR ( prn_{12} ) XOR ( prn_{13} ) XOR ( prn_{14} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_3 )</td>
<td>( prn_0 ) XOR ( prn_3 ) XOR ( prn_4 ) XOR ( prn_{11} ) XOR ( prn_{13} ) XOR ( prn_{14} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_4 )</td>
<td>( prn_0 ) XOR ( prn_4 ) XOR ( prn_2 ) XOR ( prn_3 ) XOR ( prn_{12} ) XOR ( prn_{14} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_5 )</td>
<td>( prn_0 ) XOR ( prn_4 ) XOR ( prn_2 ) XOR ( prn_3 ) XOR ( prn_{12} ) XOR ( prn_{14} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_6 )</td>
<td>( prn_0 ) XOR ( prn_4 ) XOR ( prn_2 ) XOR ( prn_3 ) XOR ( prn_{13} ) XOR ( prn_{15} )</td>
</tr>
<tr>
<td>( prn_7 )</td>
<td>( prn_1 ) XOR ( prn_2 ) XOR ( prn_3 ) XOR ( prn_{14} )</td>
</tr>
<tr>
<td>( prn_8 )</td>
<td>( prn_0 )</td>
</tr>
<tr>
<td>( prn_9 )</td>
<td>( prn_1 )</td>
</tr>
<tr>
<td>( prn_{10} )</td>
<td>( prn_2 )</td>
</tr>
<tr>
<td>( prn_{11} )</td>
<td>( prn_3 )</td>
</tr>
<tr>
<td>( prn_{12} )</td>
<td>( prn_4 )</td>
</tr>
<tr>
<td>( prn_{13} )</td>
<td>( prn_5 )</td>
</tr>
<tr>
<td>( prn_{14} )</td>
<td>( prn_6 )</td>
</tr>
<tr>
<td>( prn_{15} )</td>
<td>( prn_7 )</td>
</tr>
</tbody>
</table>

uses several XOR gates to generate more than one new bit in the next number in the sequence. Table 4.3 shows the algorithm for a pattern with bits numbered 0 through \((n - 1)\). The XOR gates generate eight new bits per iteration, so two iterations were performed to generate a 16-bit number. The logic for implementing this algorithm is shown in Appendix C.3.3. The advantage is that the next number in the sequence is generated in only two cycles, instead of sixteen for a 16-bit pattern as required by the polynomial method. Table A.3 shows first 160 samples of the 16-bit pseudo-random pattern, starting with initial seed 0x0001.
4.6.3 Duty-cycle Generation

An additional feature of the Data Module BIT Function is a variable burst duty-cycle. This duty-cycle is generated from a block of 1,024 writes to the GBC card. The BIT Function will divide the block of 1,024 writes into an active and an inactive portion. Data is written by the Data Module during the active portion of the 1,024 write block, and no data is written for the inactive portion.

The specific duty-cycle is chosen by bits 6 to 1 of the Built-in Test Control Register, as described in Section 2.3.5. The Data Module divides the 1,024 write block into eight time slices, each 128 writes long. Using this strategy, anywhere from a 1/8 active/inactive duty-cycle to an 8/0 duty-cycle can be chosen. As an example, for a 3/8 duty-cycle, the BIT function will perform 384 writes of the possible 1,024 to the GBC card, and will be idle for the remaining 640 writes. Equation 4.5 shows the derivation of the 3/8 duty-cycle.

\[
\frac{3}{8} \times 1024 = 384
\]  

(4.5)

The implementation of the burst duty-cycle is performed by a finite state machine called DUTYFSM in the BIT EPLD, 13022. Figure 4-5 shows the state diagram for the DUTYFSM finite state machine. Figure 4-5 is only functional, and does not shown the exact logic for the state transitions. See Appendix C.3.3 for the detailed FSM logic.

The duty-cycle FSM has eight sequential states numbered one through eight, one for each time slice. An enable signal which enables writes called ENABLE (active high) is constructed from an S-R Latch, with the set and reset signals controlled by the FSM. The set signal of the S-R Latch is asserted high to enable the data generators. This is done during a card reset, or at then end of the last state (EIGHT) in the FSM. In each state of the FSM, a counter counts 128 writes to the GBC card. When the counter finishes, it polls the duty-cycle select inputs to see if they match the current FSM state number. If the inputs match the current FSM state, the FSM resets the ENABLE latch, disabling the writing of data to the GBC card for the remainder of...
the current duty-cycle. If the inputs do not match the current DUTYFSM state, then the counter is reset, and the FSM proceeds to the next state. When the FSM is at the end of state EIGHT, the FSM will start over with state ONE, repeating the process.

### 4.6.4 Built-in Test Finite State Machine

The control of the Data Module BIT Functions is provided by a finite state machine called DATAFSM in the BIT EPLD, 13022. This FSM controls when to output the current sample in the BIT pattern on the sample data bus, SD[15..0]. The FSM consists of four states: BITSTOP, BITGO, BITLOW, and BITHIGH. A state diagram of the FSM is shown in Figure 4-6.

The BITSTOP and BITHIGH states control the BIT functions when BIT func-
tions are disabled and enabled, respectively. When the BIT functions are disabled, the state BITSTOP will disable writing to the GBC card and load the seed into the appropriate pattern generator (alternating or pseudo-random noise). Once the BIT functions are enabled, the FSM waits until the Arbiter EPLD requests a new data sample. The Arbiter EPLD reads a new sample from the BIT EPLD by asserting the FIFO2RDN line, since it thinks that the BIT EPLD is the Channel 2 FIFO buffer memory as discussed in Section 4.6.1. The DATAFSM will then transition to the BITHIGH state and will only return to the BITGO state once BIT functions have been enabled.

The BITGO state serves to disable writing to the GBC card during the inactive
portion of the current duty-cycle. The DATAFSM uses the FIFO 2 empty flag that it supplies to the Arbiter EPLD to detect the active and inactive portions of the duty-cycle. The duty-cycle FSM generates an enable signal which affects the state of the FIFO empty flag that the BIT EPLD sends to the Arbiter EPLD. When the enable signal is asserted by DUTYFSM, the FIFO 2 empty flag will be in the non-empty state (deasserted); when the enable signal is deasserted by DUTYFSM, the FIFO 2 empty flag will be in the empty state (asserted). If the FIFO 2 empty flag is in the empty state, then the DATAFSM will stay in the BITGO state. Once the FIFO 2 empty flag is in the non-empty state, the DATAFSM will output the current sample on the sample data bus and transition to the BITHIGH state.

The BITHIGH and BITLOW states control the state of the pattern generators, and correspond to the high and low portion of the FIFO2RDN signal generated by the Arbiter EPLD. The BIT FSM enters the BITHIGH state when the ARBITER drives FIFO2RDN low, requesting another data sample. The BITHIGH state disables the pattern generators until FIFO2RDN is driven high, and then transitions to the BITLOW state. If the FIFO 2 empty flag is in the empty state, indicating that the duty-cycle has entered the inactive portion, then BITHIGH state will transition to the BITGO state.

The BITLOW state is entered when the FIFO2RDN signal is driven high, completing a write cycle. It enables the data generators so that the next data sample in the chosen BIT pattern will be generated. The BITLOW state then waits until the FIFO2RDN signal is driven low again beginning a new write cycle, and the FSM will transition to the BITHIGH state. If the BIT functions are disabled, then the FSM will transition to the BITSTOP state. Otherwise, if the inactive portion of the duty-cycle is entered, the FSM will transition to the BITGO state where it will wait until the next active portion of the duty-cycle.

The logic for the DATAFSM is shown in Appendix C.3.3.
4.6.5 Built-in Test Timing Analysis

Timing analysis was performed on the BIT logic to verify the design. A timing diagram showing how the BIT function typically operates is shown in Figure D-5. Table D.5 details the definitions of the timing parameters. All of the EPLD timing parameters were measured with the Altera EPLD simulation software, which provides minimum and maximum timing guarantees. Hardware timing parameters were taken from their respective data books.

The timing diagram shows the generation of three samples in a BIT pattern. Initially, the BIT functions are in an idle state, corresponding to the BITSTOP or BITGO state in the DATAFSM. The BITGO state corresponds to the inactive portion of the duty-cycle. The pseudo-random noise pattern generator is loaded with the seed value of 0x0001, and the alternating pattern generator is loaded with the seed value of 0x2345. All signals are driven to an inactive state.

The BIT functions become non-idle when the Arbiter EPLD requests the next sample in the BIT pattern. The FIFO 2 empty flag, FIFO2EFN8OUT, transitions from the empty state to the non-empty state, enabling the BIT pattern generators. The Arbiter EPLD then requests another data sample by driving the Channel 2 FIFO buffer memory read strobe, FIFO2RDN8IN, low and then high. The DATAFSM transitions to BITHIGH and then to BITLOW, driving the appropriate pattern on the sample data bus, SD.

The Arbiter EPLD generates the appropriate transitions on the GBC data write strobe, GBCFIFOWRTN, and the sample address bus, SA, performing the actual Data Interface protocol shown in Figure 4-2. Once the FIFO2EFN8OUT signal becomes deasserted (logic high), corresponding to either the disabling of BIT functions or the inactive portion of the duty-cycle, the DATAFSM will transition to BITGO or BITSTOP, and all signals become deasserted.

Figure D-5 verifies that the BIT functions satisfies the timing constraints of the Data Interface Protocol as shown in Figure 4-2. The first three constraints in Figure 4-2 are on the data write strobe, GBCFIFOWRTN. The constraints specify that write
cycle time must be greater than 25 ns, with a minimum of 10 ns for the low and high portions of the write strobe. Figure D-5 shows that the write strobe lasts two cycles of the 32 MHz system clock, CLK. Since the CLK period is approximately 32 ns, these constraints are satisfied. The remaining constraints are on the rising edge of the write strobe. A minimum setup time of 6 ns and a minimum hold time of 3 ns are specified in Figure 4-2. These correspond to $T_{gfsu}$ and $T_{gfho}$ in Figure D-5. Simple calculations reveal that the BIT functions provide more than enough setup and hold time to satisfy these constraints, verifying the design specifications of Figure 4-2.
Chapter 5

Conclusion

The purpose of this Master of Engineering thesis was to design and implement the TAXI Fiber Optic Interface Support Module (TAXI-ISM). The TAXI-ISM has been completely designed, and a prototype printed circuit board has been built based upon the design presented in this thesis.

5.1 Design Summary

The design of the TAXI-ISM involved several design stages. In the first stage, the entire schematics for the TAXI-ISM printed circuit board were developed. These schematics are shown in Appendix B. After the schematics had been designed, detailed timing analysis was performed as a first-cut verification of the design. These timing diagrams are shown in Appendix D.

The design of the TAXI-ISM also involved a great deal of logic, as evidenced by the numerous finite state machines, their associated state diagrams, and the large number of EPLDs used in the design. A significant portion of the design effort was invested in designing the logic for each of the modules in order to reduce the amount of time spent in the debug phase. This included the actual logic design, as well as waveform simulations to verify the logic. Various logic files for the different modules of the TAXI-ISM are shown in Appendix C.

This concluded the digital systems design portion of the TAXI-ISM. A small bit
of mechanical design was also necessary to complete the TAXI-ISM design effort. Research concerning the parts used in the hardware schematics was performed to ascertain specific component manufacturers and part numbers. Cost/performance tradeoffs were also considered. Surface-mount components were chosen due to component space and height restrictions. Detailed parts lists were then compiled, but are not included since they do not elicit any new information.

After the parts had been identified, the artwork for the printed circuit board was designed using the PCAD design package. The mechanical specifications for each component were used to place the components in the artwork. Trace lengths and cross-overs were also taken into account. After the components had been placed, an auto-router was used to route the traces for each signal between their origins and destinations. The printed circuit board was finally designed as a double-sided eight layer board, with six signal layers, two power layers, and components on both sides of the printed circuit board. The artwork drawings are not included because they do not introduce any additional information to that presented by the schematic diagrams in Appendix B.

This completes the design portion of the TAXI-ISM. An outside vendor then used the artwork to construct the printed circuit board. After the circuit board had been constructed, a second outside vendor placed all the components on the board, and a prototype TAXI-ISM printed circuit board was completed.

5.2 Current Status of the TAXI-ISM

Due to design delays with the remainder of the Data Multiplexer II system, the thesis proposal did not include the debugging and testing of the TAXI-ISM printed circuit board. From the standpoint of the thesis proposal, this thesis involved only the complete design of a TAXI-ISM printed circuit board up to the debug phase, and has been completed. Nevertheless, the development of the remainder of the Data Multiplexer II has progressed to a point that allows the testing of the TAXI-ISM. Due to the immense amount of time and effort invested in the design phase, a prototype has
been successfully tested and speedily verified with only minor problems encountered. The debug effort has validated all of the subsystems of the TAXI-ISM.

The initial testing focused on verifying the Control Interface. The functionality of the Control Module has been verified. A test was developed to write all the read/write registers of the TAXI-ISM with an arbitrary data pattern, and then to read the contents of the registers. The test is successful if the data written to the registers matches the data read from the registers. This test was performed on the TAXI-ISM prototype, and the Control Interface and all the registers are functioning.

The second debug effort focused on the serial identification device in the Control Module. This identification device is the serial E2PROM as discussed in Chapter 2. A serial identification read/write test was devised that writes all the locations in the serial E2PROM with an arbitrary data pattern, and then reads back the entire contents. The test is successful if the data written to the serial E2PROM matches the data read. The serial identification device of the TAXI-ISM has been verified as of the writing of this thesis.

The next module to be debugged was the Data Module. The first issue that was examined was the ability of the TAXI-ISM to respond correctly to new coherent processing intervals (CPI) as discussed in Chapter 4. As a review, after a falling edge on the CPISTART signal, a new CPI is initiated. The Data Module should respond by writing a sample to the GBC card with the command code 0b0001 on the sample control bus, SC[3..0]. A test was designed that caused the GBC card to start a new CPI by driving CPISTART low. The test is successful if the TAXI-ISM responds by writing a sample with the right value on the sample control bus. The TAXI-ISM CPI functions have been verified as of the writing of this thesis.

Debug efforts were then focused on getting the Data Module to correctly write data to the GBC card buffer memory. The first step involved testing the TAXI-ISM/GBC data interface using the built-in test (BIT) functions of the Data Module as discussed in Chapter 4. The Data Module was set to write the entire address space of the GBC card with a pseudo-random noise pattern starting with a seed of 0h0001, i.e. until the Channel 2 address generators overflowed. The GBC card then started
a new coherent processing interval, CPI, resetting the address generators. A second pattern starting with a seed of 0h0002 was then written to the GBC card. A software program written in C was used to check the two pseudo-random noise patterns that the TAXI-ISM wrote to GBC memory. The program starts with a seed of 0h0001 and checks if the BIT data that the TAXI-ISM wrote matches the correct pattern. It then checks the pattern starting with seed 0h0002. If the results contain zero errors, then the test is successful and the TAXI-ISM/GBC data interface is verified. The prototype TAXI-ISM was set up to perform this test nine sequential times. Successful results were contained, verifying the functionality of the Data Module BIT function and the TAXI-ISM/GBC data interface. This test also verified the ability of the TAXI-ISM to record and respond correctly to error messages.

The final step involved generating external fiber optic input data streams and passing them through the entire TAXI-ISM to the GBC card. A test fixture called the TAXI Fiber Optic Test Fixture (TTF), developed as an MIT Undergraduate Advanced Project, was used to generate fiber optic patterns for the TAXI-ISM to process. The TTF generates an 8-bit pseudo-random noise pattern which can then be processed by the TAXI-ISM. Tests were performed to verify operation during single and multiple channel modes. The TAXI-ISM was first set up to operate the Channel 1 data stream only. The entire bandwidth of the Channel 1 GBC address space was written with an 8-bit pseudo-random noise pattern that was sequenced into a 16-bit pattern by the Receiver Module input processor. A C program was then written to verify the pattern. The same test was performed for Channel 2 only, and both channels simultaneously. All three tests successfully passed, verifying that the TAXI-ISM does indeed process and record external data streams to the GBC card correctly. The frame synchronization, byte-to-word sequencing, and signal detect capabilities of the Receiver Module were also verified with various other tests.

This concludes the TAXI Fiber Optic Interface Support Module project. The results obtained from the debugging phase lead to the conclusion that the large amount of time and effort invested in the design phase contributed to the almost effortless debug process and the rapid completion of the project. Several final printed circuit
boards are currently being built as of the writing of this thesis, and the entire Data Multiplexer II system is being prepared for real-time operation. This project has been a successful Master of Engineering thesis project both from the design standpoint, as well as the testing standpoint. The amount of effort spent on the design was truly a great accomplishment, leading to a relatively easy testing phase and virtually little debugging effort.
Appendix A

Reference Tables
<table>
<thead>
<tr>
<th>Address (Hex)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>0x01</td>
<td>PCB ID Control/Status Register</td>
</tr>
<tr>
<td>0x02</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>0x03</td>
<td>PCB ID Data Register</td>
</tr>
<tr>
<td>0x04 - 0x07</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>0x08</td>
<td>Control Register</td>
</tr>
<tr>
<td>0x09</td>
<td>Error Register</td>
</tr>
<tr>
<td>0x0A</td>
<td>Built-in Test Control Register</td>
</tr>
<tr>
<td>0x0B</td>
<td>Built-in Test Base Pattern Register</td>
</tr>
<tr>
<td>0x0C</td>
<td>Interrupt Vector Register</td>
</tr>
<tr>
<td>0x0D - 0x1F</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>0x20</td>
<td>Channel 1 Receiver Module EPLD (12847) ID Register</td>
</tr>
<tr>
<td>0x21</td>
<td>Channel 1 Control Register</td>
</tr>
<tr>
<td>0x22</td>
<td>Channel 1 Status Register</td>
</tr>
<tr>
<td>0x23</td>
<td>Channel 1 Frame Synchronization 1 Register</td>
</tr>
<tr>
<td>0x24</td>
<td>Channel 1 Frame Synchronization 2 Register</td>
</tr>
<tr>
<td>0x25</td>
<td>Channel 1 Frame Synchronization 3 Register</td>
</tr>
<tr>
<td>0x26</td>
<td>Channel 1 Frame Synchronization 4 Register</td>
</tr>
<tr>
<td>0x27</td>
<td>Channel 1 Receiver Error Register</td>
</tr>
<tr>
<td>0x28</td>
<td>Channel 1 Ending Address Low Register</td>
</tr>
<tr>
<td>0x29</td>
<td>Channel 1 Ending Address High Register</td>
</tr>
<tr>
<td>0x2A</td>
<td>Channel 2 Ending Address Low Register</td>
</tr>
<tr>
<td>0x2B</td>
<td>Channel 2 Ending Address High Register</td>
</tr>
<tr>
<td>0x2C</td>
<td>Ending Address Register Controller EPLD (13035) ID Register</td>
</tr>
<tr>
<td>0x2D</td>
<td>Reserved for Future Use</td>
</tr>
<tr>
<td>0x2E</td>
<td>Built-in Test Controller EPLD (13022) ID Register</td>
</tr>
<tr>
<td>0x2F</td>
<td>Control Module EPLD (13021) ID Register</td>
</tr>
<tr>
<td>0x30</td>
<td>Channel 2 Receiver Module EPLD (12847) ID Register</td>
</tr>
<tr>
<td>0x31</td>
<td>Channel 2 Control Register</td>
</tr>
<tr>
<td>0x32</td>
<td>Channel 2 Status Register</td>
</tr>
<tr>
<td>0x33</td>
<td>Channel 2 Frame Synchronization 1 Register</td>
</tr>
<tr>
<td>0x34</td>
<td>Channel 2 Frame Synchronization 2 Register</td>
</tr>
<tr>
<td>0x35</td>
<td>Channel 2 Frame Synchronization 3 Register</td>
</tr>
<tr>
<td>0x36</td>
<td>Channel 2 Frame Synchronization 4 Register</td>
</tr>
<tr>
<td>0x37</td>
<td>Channel 2 Receiver Error Register</td>
</tr>
<tr>
<td>0x38 - 0x3F</td>
<td>Reserved for Future Use</td>
</tr>
</tbody>
</table>

Table A.1: Control Register Address Map
<table>
<thead>
<tr>
<th>Bits 6..1</th>
<th>Duty Cycle (on/off)</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>128/896</td>
</tr>
<tr>
<td>000001</td>
<td>256/768</td>
</tr>
<tr>
<td>000010</td>
<td>384/640</td>
</tr>
<tr>
<td>000011</td>
<td>512/512</td>
</tr>
<tr>
<td>000100</td>
<td>640/384</td>
</tr>
<tr>
<td>000101</td>
<td>768/256</td>
</tr>
<tr>
<td>000110</td>
<td>896/128</td>
</tr>
<tr>
<td>000111</td>
<td>1024/0</td>
</tr>
<tr>
<td>001000 to 111111</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table A.2: Built-in Test Control Register Duty Cycle Bit Selector Definitions

<table>
<thead>
<tr>
<th>0001</th>
<th>7A16</th>
<th>B43F</th>
<th>7FCA</th>
<th>C190</th>
<th>5F55</th>
<th>118A</th>
<th>BCC5</th>
</tr>
</thead>
<tbody>
<tr>
<td>6369</td>
<td>56A0</td>
<td>4EAA</td>
<td>F878</td>
<td>335E</td>
<td>BEC0</td>
<td>89C7</td>
<td>AB35</td>
</tr>
<tr>
<td>CF7E</td>
<td>6BBA</td>
<td>6EB4</td>
<td>52C9</td>
<td>6FAB</td>
<td>AC56</td>
<td>4752</td>
<td>118C</td>
</tr>
<tr>
<td>DAA4</td>
<td>6FD5</td>
<td>29D4</td>
<td>0A5B</td>
<td>64D3</td>
<td>47EB</td>
<td>889A</td>
<td>A1D9</td>
</tr>
<tr>
<td>0A54</td>
<td>262A</td>
<td>82D2</td>
<td>F62D</td>
<td>6A8A</td>
<td>D2FD</td>
<td>FD5F</td>
<td>2B29</td>
</tr>
<tr>
<td>03AB</td>
<td>D3A9</td>
<td>D3AB</td>
<td>2786</td>
<td>4FF8</td>
<td>B06D</td>
<td>334D</td>
<td>8DE7</td>
</tr>
<tr>
<td>AEF2</td>
<td>D778</td>
<td>11A1</td>
<td>BCEA</td>
<td>21BF</td>
<td>D14F</td>
<td>B7F2</td>
<td>CA73</td>
</tr>
<tr>
<td>D9FC</td>
<td>8F97</td>
<td>116B</td>
<td>C61F</td>
<td>1B77</td>
<td>BEE5</td>
<td>61B2</td>
<td>39E1</td>
</tr>
<tr>
<td>B7BA</td>
<td>94A3</td>
<td>21EB</td>
<td>FE41</td>
<td>ADE9</td>
<td>B95F</td>
<td>70F9</td>
<td>FF2E</td>
</tr>
<tr>
<td>EE1B</td>
<td>ADA9</td>
<td>B901</td>
<td>F554</td>
<td>F0F1</td>
<td>1CAB</td>
<td>C9BE</td>
<td>6C45</td>
</tr>
<tr>
<td>97FB</td>
<td>C1A9</td>
<td>C6FE</td>
<td>61AD</td>
<td>C6FB</td>
<td>89F7</td>
<td>1606</td>
<td>76DC</td>
</tr>
<tr>
<td>1047</td>
<td>0B88</td>
<td>566D</td>
<td>F89D</td>
<td>DBC8</td>
<td>A21C</td>
<td>E163</td>
<td>43FC</td>
</tr>
<tr>
<td>2C19</td>
<td>B8FE</td>
<td>0B07</td>
<td>1429</td>
<td>36FE</td>
<td>BAC0</td>
<td>8C2F</td>
<td>F1E5</td>
</tr>
<tr>
<td>3281</td>
<td>40BC</td>
<td>2FC9</td>
<td>068F</td>
<td>4558</td>
<td>B9DB</td>
<td>E208</td>
<td>90B6</td>
</tr>
<tr>
<td>7145</td>
<td>8F18</td>
<td>532F</td>
<td>087C</td>
<td>7A7F</td>
<td>90D6</td>
<td>7134</td>
<td>4863</td>
</tr>
<tr>
<td>DB1F</td>
<td>0394</td>
<td>2C63</td>
<td>AF30</td>
<td>22CC</td>
<td>E197</td>
<td>6C60</td>
<td>7F8E</td>
</tr>
<tr>
<td>5382</td>
<td>BE3A</td>
<td>9EB4</td>
<td>89A4</td>
<td>257F</td>
<td>D444</td>
<td>63D5</td>
<td>27EC</td>
</tr>
<tr>
<td>E52A</td>
<td>62D2</td>
<td>3AE1</td>
<td>4B1D</td>
<td>5D13</td>
<td>7541</td>
<td>18BC</td>
<td>6D3D</td>
</tr>
<tr>
<td>7562</td>
<td>96A8</td>
<td>F3AA</td>
<td>72D2</td>
<td>2D40</td>
<td>205E</td>
<td>AAEF</td>
<td>D958</td>
</tr>
<tr>
<td>1D49</td>
<td>3C25</td>
<td>DEAE</td>
<td>C09E</td>
<td>66C0</td>
<td>7638</td>
<td>82C7</td>
<td>A36B</td>
</tr>
</tbody>
</table>

Table A.3: Built-in Test Pseudo-random Noise Pattern Sample
Appendix B

Hardware Schematics
Figure B-1: Channel 1 Receiver Module Schematics
Figure B-2: Channel 2 Receiver Module Schematics
Figure B-3: Control and Data Module Schematics
Figure B-4: Input/Output P1 Connector Schematics
Figure B-5: Decoupling and Miscellaneous Schematics

1. LAST REFERENCE DESIGNATOR USED:
   - C101
   - G2
   - C102
   - A4
   - C103
   - F1
   - H10
   - M4
   - U21

2. SPARE COMPONENTS: U22 21

3. ALL RESISTOR VALUES ARE IN OHMS; 1% TOLERANCE

4. ALL CAPACITOR VALUES ARE IN MICROFARADS WITH MAXIMUM TOLERANCE OF ±20% UNLESS OTHERWISE NOTED

POWER AND GROUND TABLE, DECOUPLING, MISCELLANEOUS

SCHEMATIC

CONTACT

PROJECT: ASTOR-12844

SWN: 12844

DATE: 11/23/93

TAXI Fiber Optic Interface Support Module

DRAWING

DRAWING NO.: 12844

CLASSIFICATION: UNCLASSIFIED

SIGNATURE

DATE: 11/23/93

Sheet 5 of 5
Appendix C

Logic Design (Firmware)

C.1 Receiver Module: 12847

TITLE "12847-";

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
**
** DWG 12847 REV -
** Project: TAXI Fiber Optic Interface Support Module **
** Designer: Stanley C. Wang **
**
** Description: EPLD design file for Sequencer Function. **
** Used on ASTB Data Multiplexer II TAXI Fiber **
** Optic Interface Support Module (TAXI-ISM) **
**
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DESIGN IS "12847-"
BEGIN

DEVICE "12847-" IS "EPM7160LC84-10"
BEGIN

bitcr0 @ 10 : INPUT ;
ctladdr0 @ 30 : INPUT ;
ctladdr1 @ 29 : INPUT ;
ctladdr2 @ 28 : INPUT ;
ctladdr3 @ 27 : INPUT ;
ctladdr4 @ 25 : INPUT ;

108
ctladdr5   @ 24 : INPUT;
ctloeinn   @ 84 : INPUT;
ctlrln     @  1 : INPUT;
ctlwrln    @  2 : INPUT;
fifofulln  @ 77 : INPUT;
ismcr0     @ 12 : INPUT;
jumper     @ 14 : INPUT;
overflow   @  8 : INPUT;
resetn     @ 31 : INPUT;
sigdet     @  9 : INPUT;
tclk       @ 83 : INPUT;
tcstrb     @ 40 : INPUT;
tc0        @ 36 : INPUT;
tc1        @ 35 : INPUT;
tc2        @ 34 : INPUT;
tc3        @ 33 : INPUT;
tdstrb     @ 41 : INPUT;
td0        @ 54 : INPUT;
td1        @ 52 : INPUT;
td2        @ 51 : INPUT;
td3        @ 50 : INPUT;
td4        @ 49 : INPUT;
td5        @ 48 : INPUT;
td6        @ 45 : INPUT;
td7        @ 44 : INPUT;
tv1tn       @ 37 : INPUT;
ackn       @ 11 : OUTPUT;
tloeooutn  @  4 : OUTPUT;
fifodata0  @ 56 : OUTPUT;
fifodata1  @ 57 : OUTPUT;
fifodata2  @ 58 : OUTPUT;
fifodata3  @ 60 : OUTPUT;
fifodata4  @ 61 : OUTPUT;
fifodata5  @ 62 : OUTPUT;
fifodata6  @ 63 : OUTPUT;
fifodata7  @ 64 : OUTPUT;
fifodata8  @ 65 : OUTPUT;
fifodata9  @ 67 : OUTPUT;
fifodata10 @ 68 : OUTPUT;
fifodata11 @ 69 : OUTPUT;
fifodata12 @ 70 : OUTPUT;
fifodata13 @ 71 : OUTPUT;
fifodata14 @ 73 : OUTPUT;
fifodata15 @ 74 : OUTPUT;
fiforstn   @ 75 : OUTPUT;
CONNECTED_PINS
BEGIN
  ctoeoutn : OUTPUT;
  ctoeinn : INPUT;
END;
END;

SUBDESIGN '12847-
(t
  tclk : INPUT; % TAXI Output Clock %
  % Board Reset %
  resetn : INPUT; % Hardware Reset %
  % GBC Control Interface %
  ctlrdsn,ctlwrln : INPUT; % Control Read AND Write Strobes %
  jumper : INPUT; % EPLD Jumper for Reg. Address %
  % Gnd indicates Channel 2 %
  % VCC indicates Channel 1 %
  % Control Address Bus %
  ctladdr[5..0] : INPUT;
  % Control Data Bus %
  ctldata[7..0] : BIDIR;
  % Control Address Bus %
  ackn : OUTPUT; % Channel Control Acknowledge %
  % ISM and BIT Control Reg. Bit 0 %
  ismcr0, bitcre0 : INPUT;
  % TAXI Receiver Signals %
  tvlttn : INPUT; % TAXI Violation %
  tcstrtb : INPUT; % TAXI Command Strobe %
  tdstrtb : INPUT; % TAXI Data Strobe %
  sigdet : INPUT; % Receiver Signal Detect %
  tc[3..0] : INPUT; % TAXI Parallel Command Output %
  td[7..0] : INPUT; % TAXI Parallel Data Output %
  taxirstn : OUTPUT; % TAXI /RESET Signal %
% FIFO Control %

fifowrtn : OUTPUT; % FIFO Write Signal %
fifodata[15..0] : OUTPUT; % FIFO Data %
fiforstn : OUTPUT; % FIFO /RESET Signal %
fifofulln : INPUT; % FIFO Full Flag %

% Data Interface %

overflow : INPUT; % Address Counter Overflow Flag %

cctloeinn : INPUT; % EPLD Output Enable %
cctloeoutn : OUTPUT;

)

variable

% Define ctldata external pads %

c CDDLATA_t[7..0] : TRI;
fnsel : SOFT; % See if this channel is selected %

cctlwrlonsync[1..0] : DFF; % Synchronize control signals %
cctlrdnsync[1..0] : DFF;
ackn : DFF;

% Control Write FSM %

writefsm : MACHINE WITH STATES
( writeidle, writeack, writewait);

% Control Read FSM %

readfsm: MACHINE WITH STATES
( readidle, readack, readwait);

% Define registers %

fnchr[7..0] : DFFE; % Ch. n Control Register %
fnsr[1..0] : DFFE; % Ch. n Status Register %
fnsr_out[1..0] : DFFE;
fnfs0r[7..0] : DFFE; % Ch. n Frame Sync 0 Register %
fnfs1r[7..0] : DFFE; % Ch. n Frame Sync 1 Register %
fnfs2r[7..0] : DFFE; % Ch. n Frame Sync 2 Register %
fnfs3r[7..0] : DFFE; % Ch. n Frame Sync 3 Register %
fnrer[3..0] : DFFE; % Ch. n Receiver Error Reg. %
fnrer_out[3..0] : DFFE;

bitcr0sync[1..0] : DFF;
ismcr0sync[1..0] : DFF;
% Ch. n Status Register FSM %
fnsrfsm: machine with states
(fnsrridle, fnsrirdwait, fnsrnoti, fnsrnirdwait);

% Ch. n Receiver Error Register %nrerfsm: machine with states
(fnreridle, fnreriwait, fnrernoti, fnrerniwait);

% Frame Synchronizer %
fsync: machine with states % Frame Sync State Machine %
(idle, sync1, sync2, sync3, sync4);

fsyncreset : DFF;

fs0matchl, fs0matchh, fs1matchl, fs1matchh : DFFE;
fs2matchl, fs2matchh, fs3matchl, fs3matchh : DFFE;

fsyncdet : DFFE;
tsyncdet : DFFE;

% Synchronize Inputs %
resetnsync[1..0] : DFF;
tdstrbo : DFFE;
fcycle : DFFE;
tclkdel : DFFE;
sigdetsync[1..0] : DFF;

% FIFO Control %
fifodata[15..0] : DFF; % FIFO Output Data %
fifowrtn : DFFE; % FIFO Write Signal %

BEGIN
DEFAULTS
cloeeoutn = VCC;

% Registers %
fncr[7..0].clrn = VCC;
fncr[7..0].prn = VCC;
fncr[7..0].ena = GND;
fnsr[1..0].clrn = VCC;
fnsr[1..0].prn = VCC;
fnsr[1..0].ena = GND;
fnsr_out[1..0].clrn = VCC;
fnsr_out[1..0].prn = VCC;
fnsr_out[1..0].ena = GND;
fnfs0r[7..0].clrn = VCC;
fnfs0r[7..0].prn = VCC;
fnfs0r[7..0].ena = GND;
fnfs1r[7..0].clrn = VCC;
fnfs1r[7..0].prn = VCC;
fnfs1r[7..0].ena = GND;
fnfs2r[7..0].clrn = VCC;
fnfs2r[7..0].prn = VCC;
fnfs2r[7..0].ena = GND;
fnfslr[7..0].clrn = VCC;
fnfslr[7..0].prn = VCC;
fnfslr[7..0].ena = GND;
fnfs2r[7..0].clrn = VCC;
fnfs2r[7..0].prn = VCC;
fnfs2r[7..0].ena = GND;
fnfs3r[7..0].clrn = VCC;
fnfs3r[7..0].prn = VCC;
fnfs3r[7..0].ena = GND;
fnrer[3..0].clrn = VCC;
fnrer[3..0].prn = VCC;
fnrer[3..0].ena = GND;
fnrer_out[3..0].clrn = VCC;
fnrer_out[3..0].prn = VCC;
fnrer_out[3..0].ena = GND;

bitcr0sync[].clrn = VCC;
bitcr0sync[].prn = VCC;
ismcr0sync[].clrn = VCC;
ismcr0sync[].prn = VCC;

ackn.d = VCC;
ackn.clrn = VCC;

% FIFO %
fifowrtn.clrn = VCC;
fifowrtn.prn = VCC;
fifodata[15..0].clrn = VCC;
fifodata[15..0].prn = VCC;

tclkdel.clrn = VCC;
tclkdel.prn = VCC;

% Frame Sync %
fsyncreset.clrn = VCC;
fsyncreset.prn = VCC;
fsyncreset.clrn = VCC;
fsyncreset.prn = VCC;
tsyncdet.clrn = VCC;
tsyncdet.prn = VCC;
tsyncdet.clrn = VCC;
tsyncdet.prn = VCC;

fs0matchl.clrn = VCC;
fs0matchl.prn = VCC;
fs0matchh.clrn = VCC;
fs0matchh.prn = VCC;
fs1matchl.clrn = VCC;
fs1matchl.prn = VCC;
fs1matchh.clrn = VCC;
fs1matchh.prn = VCC;
fs2matchl.clrn = VCC;
fs2matchl.prn = VCC;
fs2matchh.clrn = VCC;
fs2matchh.prn = VCC;
fs3matchl.clrn = VCC;
fs3matchl.prn = VCC;
fs3matchh.clrn = VCC;
fs3matchh.prn = VCC;
fcycle.clrn = VCC;
fcycle.prn = VCC;

% Synchronize Inputs %
resetnsync[1..0].clrn = VCC;
resetnsync[1..0].prn = VCC;
sigdetsync[1..0].clrn = VCC;
sigdetsync[1..0].prn = VCC;

END DEFAULTS;

% Synchronize Reset %
resetnsync[1..0].clk = GLOBAL (tclk);
resetnsync[1].d = resetn;
resetnsync[0].d = resetnsync[1].q;

% Synchronize Control Signals %
ctlrdnsync[].clk = GLOBAL(tclk);
ctlrdnsync[].clrn = VCC;
ctlrdnsync[].prn = resetnsync[0].q;
ctlrdnsync[1].d = ctlrdn;
ctlrdnsync[0].d = ctlrdnsync[1].q;

ctlwrlonsync[].clk = GLOBAL(tclk);
ctlwrlonsync[].clrn = VCC;
ctlwrlonsync[].prn = resetnsync[0].q;
ctlwrlonsync[1].d = ctlwrlon;
ctlwrlon-sync[0].d = ctlwrlon-sync[1].q;

ackn.clk = GLOBAL(tclk);
ackn.prn = resetnsync[0].q AND (!ctlrdn OR !ctlwrlon);

% Channel Selector %
% CTLADDR b100 IS Ch. 1 %
% CTLADDR b110 IS Ch. 2 %
fnssel = (ctladdr[5..3] == (B"1", !jumper, B"0"));

% Control Reads %
ctldata[7..0] = ctldatat[7..0];
ctldatat[7..0].oe = !ctloeinn;

readfsm.clk = GLOBAL(tclk);
readfsm.reset = !resetnsync[0].q;

CASE readfsm IS
  WHEN readidle =>
    IF (!ctlrdnsync[0].q AND fnssel) THEN
      readfsm = readack;
    ELSIF (!ctlrdnsync[0].q) THEN
      readfsm = readwait;
    ELSE
      readfsm = readidle;
    END IF;
  WHEN readack => % Acknowledge completion of Ctl. Read %
CASE ctladdr[2..0] IS
  WHEN 0 => % EPLD ID Reg. (Revision Level) %
    ctldatat[7..0] = H"00";
  WHEN 1 => % Channel Control Register %
    ctldatat[7..0] = fncr[7..0];
  WHEN 2 => % Channel Status Register %
    ctldatat[7..0] = (H"0",B"00",fnr_out[1..0]);
  WHEN 3 => % Frame Sync 1 Register %
    ctldatat[7..0] = fnfs1r[7..0];
  WHEN 4 => % Frame Sync 2 Register %
    ctldatat[7..0] = fnfs2r[7..0];
  WHEN 5 => % Frame Sync 3 Register %
    ctldatat[7..0] = fnfs3r[7..0];
  WHEN 6 => % Frame Sync 4 Register %
    ctldatat[7..0] = fnfs4r[7..0];
  WHEN 7 => % Channel Receiver Error Reg. %
    ctldatat[7..0] = (H"0",fnrer_out[3..0]);

115
END CASE;

cloexoutn = ctrlrdn;

% Wait until sync registers are clear %
IF (ctrlrdnsync[0].q) THEN
  readfsm = readidle;
ELSE
  ackn.d = GND;
  readfsm = readack;
END IF;
WHEN readwait =>
  IF (ctrlrdnsync[0].q) THEN
    readfsm = readidle;
  ELSE
    readfsm = readwait;
  END IF;
END CASE;

% Control Writes %
writefsm.clk = GLOBAL(tclk);
writefsm.reset = !resetnsync[0].q;

CASE writefsm IS
  WHEN writeidle =>
    IF (!ctlwrlonsync[0].q AND fnsel) THEN
      CASE ctladdr[2..0] IS
        WHEN 1 =>
          fncr[].ena = VCC;
          fncr[7..0].d = ctldata[7..0];
        WHEN 3 =>
          fnfs0r[].ena = VCC;
          fnfs0r[].d = ctldata[7..0];
        WHEN 4 =>
          fnfs1r[].ena = VCC;
          fnfs1r[].d = ctldata[7..0];
        WHEN 5 =>
          fnfs2r[].ena = VCC;
          fnfs2r[].d = ctldata[7..0];
        WHEN 6 =>
          fnfs3r[].ena = VCC;
          fnfs3r[].d = ctldata[7..0];
      END CASE;
      writefsm = writeack;
    END IF;
END CASE;

writefsm = writeack;

116
ELSIF (!ctlwrlonsync[0].q) THEN
    writefsm = writewait;
ELSE
    writefsm = writeidle;
END IF;
WHEN writeack => % Acknowledge completion of Ctl. Wrt. %
    IF (ctlwrlonsync[0].q) THEN
        writefsm = writeidle;
    ELSE
        ackn.d = GND;
        writefsm = writeack;
    END IF;
END CASE;

% Ch. n Control Register %
fnr[7..0].clk = GLOBAL (tclk);
fnr[7].prn = resetnsync[0].q;
fnr[6..0].clrn = resetnsync[0].q;
taxirstn = !fnr[6].q;
ffirstn = !fnr[7].q;

% Ch. n Frame Sync 0 Register %
fnfs0r[7..0].clk = GLOBAL (tclk);
fnfs0r[7..0].clrn = resetnsync[0].q;

% Ch. n Frame Sync 1 Register %
fnfs1r[7..0].clk = GLOBAL (tclk);
fnfs1r[7..0].clrn = resetnsync[0].q;

% Ch. n Frame Sync 2 Register %
fnfs2r[7..0].clk = GLOBAL (tclk);
fnfs2r[7..0].clrn = resetnsync[0].q;

% Ch. n Frame Sync 3 Register %
fnfs3r[7..0].clk = GLOBAL (tclk);
fnfs3r[7..0].clrn = resetnsync[0].q;

% Bit Control Register and DC Control Register Bits %
bitcr0sync[].clk = GLOBAL (tclk);
bitcr0sync[].clrn = resetnsync[0].q;

bitcr0sync[1].d = bitcr0;
bitcr0sync[0].d = bitcr0sync[1].q;

ismcr0sync[].clk = GLOBAL (tclk);
ismcr0sync[].clrn = resetnsync[0].q;

ismcr0sync[1].d = ismcr0;
ismcr0sync[0].d = ismcr0sync[1].q;

% Synchronize SIGDET %
sigdetsync[].clk = GLOBAL(tclk);
sigdetsync[].clrn = resetnsync[0].q;
% If fnr[5] = 1 then enable sigdet %
% If fnr[5] = 0 then disable %
sigdetsync[].prn = fnr[5].q OR !resetnsync[0].q;
sigdetsync[1].d = sigdet;
sigdetsync[0].d = sigdetsync[1].q;

% Frame Synchronization %
tdstrb0.clk = GLOBAL (tclk);
tdstrb0.clrn = resetnsync[0].q;
tdstrb0.d = tdstrb AND sigdetsync[0].q;
tdstrb0.ena = ismcr0sync[0].q AND !bitcr0sync[0].q;

fs0matchh.clk = GLOBAL (tclk);
fs0matchl.clk = GLOBAL (tclk);
fs1matchh.clk = GLOBAL (tclk);
fs1matchl.clk = GLOBAL (tclk);
fs2matchh.clk = GLOBAL (tclk);
fs2matchl.clk = GLOBAL (tclk);
fs3matchh.clk = GLOBAL (tclk);
fs3matchl.clk = GLOBAL (tclk);

fs0matchh.clrn = resetnsync[0].q;
fs0matchl.clrn = resetnsync[0].q;
fs1matchh.clrn = resetnsync[0].q;
fs1matchl.clrn = resetnsync[0].q;
fs2matchh.clrn = resetnsync[0].q;
fs2matchl.clrn = resetnsync[0].q;
fs3matchh.clrn = resetnsync[0].q;
fs3matchl.clrn = resetnsync[0].q;
fs0matchh.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs0matchl.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs1matchh.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs1matchl.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs2matchh.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs2matchl.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs3matchh.ena = ismcrosync[0].q AND !bitcr0sync[0].q;
fs3matchl.ena = ismcrosync[0].q AND !bitcr0sync[0].q;

fs0matchh.d = (td[7..4] == fnfs0r[7..4]);
fs0matchl.d = (td[3..0] == fnfs0r[3..0]);
fs1matchh.d = (td[7..4] == fnfs1r[7..4]);
fs1matchl.d = (td[3..0] == fnfs1r[3..0]);
fs2matchh.d = (td[7..4] == fnfs2r[7..4]);
fs2matchl.d = (td[3..0] == fnfs2r[3..0]);
fs3matchh.d = (td[7..4] == fnfs3r[7..4]);
fs3matchl.d = (td[3..0] == fnfs3r[3..0]);

% Frame Sync FSM %
fsync.clk = GLOBAL (tclk);
fsyncreset.clk = GLOBAL(tclk);
fsyncreset.d = !resetnsync[1].q OR !ismc0sync[0].q
               OR bitcr0sync[0].q;
fsync.reset = fsyncreset.q;

CASE (fsync) IS
  WHEN idle =>
    IF (tdstrb0 AND fs0matchh AND fs0matchl) THEN
      fsync = sync1;
    END IF;
  WHEN sync1 =>
    IF (tdstrb0 AND fs1matchh AND fs1matchl) THEN
      fsync = sync2;
    ELSIF (tdstrb0) THEN
      fsync = idle;
    END IF;
  WHEN sync2 =>
    IF (tdstrb0 AND fs2matchh AND fs2matchl) THEN
      fsync = sync3;
    ELSIF (tdstrb0) THEN
      fsync = idle;
    END IF;
  WHEN sync3 =>
    IF (tdstrb0 AND fs3matchh AND fs3matchl) THEN
      fsync = sync4;
ELSIF (tdstrb0) THEN
    fsync = idle;
END IF;
WHEN sync4 =>
    fsync = idle;
WHEN OTHERS =>
    fsync = idle;
END CASE;

IF ((fncr[4..3] == 0) AND (fsync == sync1)) THEN
    fsyncdet = VCC;
ELSIF ((fncr[4..3] == 1) AND (fsync == sync2)) THEN
    fsyncdet = VCC;
ELSIF ((fncr[4..3] == 2) AND (fsync == sync3)) THEN
    fsyncdet = VCC;
ELSIF ((fncr[4..3] == 3) AND (fsync == sync4)) THEN
    fsyncdet = VCC;
END IF;

% Updating and Reading Ch. n Status Register %
fsyncdet.clk = GLOBAL (tclk);
fsyncdet.clrn = resetnsync[0].q;
fsyncdet.ena = ismcrOsync[0].q AND !bitcrOsync[0].q;
tsncdet.d = tcstrb AND (tc[3..0] == 0) AND sigdetsync[0].q;
tsncdet.clk = GLOBAL (tclk);
tsncdet.clrn = resetnsync[0].q;
tsncdet.ena = ismcr0sync[0].q AND !bitcr0sync[0].q;

fnsr[1..0].clk = GLOBAL (tclk);
fnsr[1..0].clrn = resetnsync[0].q;
fnsr_out[1..0].clk = GLOBAL (tclk);
fnsr_out[1..0].clrn = resetnsync[0].q;

fnsrfsm.reset = !resetnsync[0].q;
fnsrfsm.clk = GLOBAL (tclk);

CASE (fnsrfsm) IS
  WHEN fnsridle =>
    IF (ismcr0sync[0].q) THEN
      fnsrfsm = fnsrnoti;
    ELSIF (!ctlrdnsync[0] AND fnsel AND
            (ctladdr[2..0] == 2)) THEN
      fnsrfsm = fnsrirdwait;
  WHEN fnsrnoti =>
    ...
fnsr_out[1..0].ena = VCC;
fnsr_out[1..0] = fnsr[1..0];
fnsr[].ena = VCC;
fnsr[1..0] = GND;
ELSIF (!ctlrdnsync[0].q) THEN
  fnsrfsm = fnsrirdwait;
ELSE
  fnsrfsm = fnsridle;
END IF;
WHEN fnsrirdwait =>
  IF (ctlrdnsync[0]) THEN
    fnsrfsm = fnsridle;
  ELSE
    fnsrfsm = fnsrirdwait;
  END IF;
WHEN fnsrnoti =>
  IF (!ismcr0sync[0].q) THEN
    fnsrfsm = fnsridle;
    fnsr[1] = fnsr[1] OR fsyncdet.q;
    fnsr[0] = fnsr[0] OR tsyncdet.q;
  ELSIF (!ctlrdnsync[0] AND fnsel AND
tladdr[2..0] == 2) THEN
    fnsrfsm = fnsrnirdwait;
    fnsr_out[1..0].ena = VCC;
    fnsr_out[1..0] = fnsr[1..0];
    fnsr[1] = GND OR fsyncdet.q;
    fnsr[0] = GND OR tsyncdet.q;
  ELSIF (!ctlrdnsync[0].q) THEN
    fnsr[1] = fnsr[1] OR fsyncdet.q;
    fnsr[0] = fnsr[0] OR tsyncdet.q;
    fnsrfsm = fnsrnoti;
  ELSE
    fnsr[1] = fnsr[1] OR fsyncdet.q;
    fnsr[0] = fnsr[0] OR tsyncdet.q;
    fnsrfsm = fnsrnoti;
  END IF;
END IF;

fnsr[].ena = VCC;
WHEN fnsrnirdwait =>
  IF (ctlrdnsync[0]) THEN
    fnsrfsm = fnsrnoti;
  ELSE
    fnsrfsm = fnsrirdwait;
  END IF;
fnr[] .ena = VCC;
fnr[1] = fnr[1] OR fsyncdet.q;
fnr[0] = fnr[0] OR tsyncdet.q;
END CASE;

% Receiver Error Register %
fnrer[3..0].clk = GLOBAL (tclk);
fnrer[3..0].clrn = resetnsync[0].q;
fnrer_out[3..0].clk = GLOBAL (tclk);
fnrer_out[3..0].clrn = resetnsync[0].q;

fnrerfsm.reset = !resetnsync[0].q;
fnrerfsm.clk = GLOBAL (tclk);
CASE (fnrerfsm) IS
  WHEN fnreridle =>
    IF (ismcrOsync[0].q) THEN
      fnrerfsm = fnrernoti;
    ELSIF (!ctlrdnsync[0] AND fnsel AND
            (cldaddr[2..0] == 7)) THEN
      fnrer_out[].ena = VCC;
      fnrer_out[3..0] = fnrer[3..0];
      fnrerfsm = fnrerwait;
    ELSIF (!ctlwrlonsync[0] AND fnsel AND
            (cldaddr[2..0] == 7)) THEN
      fnrer[].ena = VCC;
      fnrer[3..0] = !cldata[3..0] AND fnrer[3..0];
      fnrerfsm = fnrerwait;
    ELSIF (!ctlrdnsync[0].q OR !ctlwrlonsync[0].q) THEN
      fnrerfsm = fnrerwait;
    ELSE
      fnrerfsm = fnreridle;
    END IF;
  WHEN fnrerwait =>
    IF (ctlrdnsync[0] AND ctlwrlonsync[0].q) THEN
      fnrerfsm = fnreridle;
    ELSE
      fnrerfsm = fnrerwait;
    END IF;
  WHEN fnrernoti =>
    IF (!ismcrOsync[0].q) THEN
      fnrerfsm = fnreridle;
      fnrer[3].d = fnrer[3].q OR overflow;
      fnrer[2].d = fnrer[2].q OR !fifofulln;
      fnrer[1].d = fnrer[1].q OR tvltln;
    END IF;
  WHEN fnrerwait =>
    IF (ctlrdnsync[0] AND ctlwrlonsync[0].q) THEN
      fnrerfsm = fnreridle;
    ELSE
      fnrerfsm = fnrerwait;
    END IF;
  WHEN fnrernoti =>
    IF (!ismcrOsync[0].q) THEN
      fnrerfsm = fnreridle;
      fnrer[3].d = fnrer[3].q OR overflow;
      fnrer[2].d = fnrer[2].q OR !fifofulln;
      fnrer[1].d = fnrer[1].q OR tvltln;
    END IF;
  END CASE;

fnrer[0].d = fnrer[0].q OR !sigdetsync[0].q;
ELSIF (!ctlrdnsync[0] AND fnsel AND
    (cldaddr[2..0] == 7)) THEN
    fnrerfsm = fnrerniwait;
    fnrer_out[].ena = VCC;
    fnrer_out[3..0] = fnrer[3..0];
    fnrer[3].d = fnrer[3].q OR overflow;
    fnrer[2].d = fnrer[2].q OR !fifofulln;
    fnrer[1].d = fnrer[1].q OR tvltn;
    fnrer[0].d = fnrer[0].q OR !sigdetsync[0].q;
ELSIF (!ctlwrlonsync[0] AND fnsel AND
    (cldaddr[2..0] == 7)) THEN
    fnrerfsm = fnrerniwait;
    fnrer[0] = (!clddata[0] AND fnrer[0]) OR !sigdetsync[0].q;
    fnrerfsm = fnrerniwait;
ELSIF (!cldrdnsync[0].q OR !cldwrlonsync[0].q) THEN
    fnrer[3].d = fnrer[3].q OR overflow;
    fnrer[2].d = fnrer[2].q OR !fifofulln;
    fnrer[1].d = fnrer[1].q OR tvltn;
    fnrer[0].d = fnrer[0].q OR !sigdetsync[0].q;
    fnrerfsm = fnrerniwait;
ELSE
    fnrer[3].d = fnrer[3].q OR overflow;
    fnrer[2].d = fnrer[2].q OR !fifofulln;
    fnrer[1].d = fnrer[1].q OR tvltn;
    fnrer[0].d = fnrer[0].q OR !sigdetsync[0].q;
    fnrerfsm = fnrernoti;
END IF;

fnrer[].ena = VCC;
WHEN fnrerniwait =>
    IF (cldrdnsync[0].q AND cldwrlonsync[0].q) THEN
        fnrerfsm = fnrernoti;
    ELSE
        fnrerfsm = fnrerniwait;
    END IF;

fnrer[].ena = VCC;
fnrer[3].d = fnrer[3].q OR overflow;
fnrer[2].d = fnrer[2].q OR !fifofulln;
fnrer[1].d = fnrer[1].q OR tvltn;
fnrer[0].d = fnrer[0].q OR !sigdetsync[0].q;

END CASE;

% Writing to the FIFO %
ifodata[15..0].clk = GLOBAL (tclk);
ifodata[15..0].clrn = resetnsync[0].q;

IF (tdstrb AND sigdetsync[0].q AND ((fcycle AND (fnr[1..0] == 1)) OR (fcycle AND (fnr[1..0] == 2)))) THEN
  ifodata[15..8] = td[7..0];
ELSE
  ifodata[15..8] = ifodata[15..8];
END IF;

IF (tdstrb AND sigdetsync[0].q AND ((fcycle AND (fnr[1..0] == 1)) OR (!fcycle AND (fnr[1..0] == 2)))) THEN
  ifodata[7..0] = td[7..0];
ELSE
  ifodata[7..0] = ifodata[7..0];
END IF;

fcycle.clk = GLOBAL (tclk);

IF (!resetnsync[0].q) THEN
  fcycle = GND;
ELSIF (!fnr[2] AND tdstrb AND sigdetsync[0].q) THEN
  fcycle = !fcycle;
ELSIF (fnr[2] AND tcstrb AND sigdetsync[0].q AND (tc[3..0] == 0)) THEN
  fcycle = GND;
ELSIF (fnr[2] AND tdstrb AND sigdetsync[0].q) THEN
  fcycle = !fcycle;
ELSE
  fcycle = fcycle;
END IF;

fifowrtn.clk = GLOBAL (tclk);
fifowrtn.prn = !tclkdel;
!fifowrtn.d = fcycle AND tdstrb AND ((fnr[1..0] == 1) OR (fnr[1..0] == 2)) AND sigdetsync[0].q AND fifofulln AND ismcr0sync[0].q AND
!bitcr0sync[0].q;

tclkdel.d  = VCC;
tclkdel.clk = !tclk;
tclkdel.clrn = !fifowrtn.q;
END;
C.2 Control Module: 13021

TITLE "13021-";

CONSTANT ID_CMD_CTL_ADDR = H"1";
CONSTANT ID_CMD_DATA_ADDR = H"3";

CONSTANT ID_CLK_DIVISOR = 16;
CONSTANT ID_SHIFT_COUNT = 7;

DESIGN IS "13021-

DEVICE "13021-" IS "EPM7128LC84-10"
BEGIN

bitcr0  @ 74  : INPUT ;
ch1ackn @ 67  : INPUT ;
ch2ackn @ 68  : INPUT ;
clk    @ 83  : INPUT ;
ctladdr0 @ 60 : INPUT ;
ctladdr1 @ 61 : INPUT ;
ctladdr2 @ 62 : INPUT ;
ctladdr3 @ 63 : INPUT ;
ctladdr4 @ 64 : INPUT ;
ctladdr5 @ 65 : INPUT ;
ctloeinn @ 2  : INPUT ;
ctlrdn  @ 69 : INPUT ;
ctlwrhin  @ 70 : INPUT ;
ctlwrlon  @ 71 : INPUT ;
gbcfifoffn @ 33 : INPUT ;
ackn  @ 56 : INPUT ;
id_sdin  @ 52 : INPUT ;
id_wp_inn  @ 51 : INPUT ;
oveflow1  @ 34 : INPUT ;
oveflow2  @ 35 : INPUT ;
resetn  @ 1 : INPUT ;
ackn  @ 55 : OUTPUT ;
addrlen  @ 54 : OUTPUT ;
ctloeoutn  @ 4 : OUTPUT ;
ctldrdsync  @ 36 : OUTPUT ;
ctlwrhsync  @ 37 : OUTPUT ;
ctlwrlnsync  @ 39 : OUTPUT ;
iacknout  @ 57 : OUTPUT ;
id_csn  @ 50 : OUTPUT ;
id_holdn  @ 49 : OUTPUT ;
id_sclk  @ 48 : OUTPUT ;
id_sdout  @ 41 : OUTPUT ;
id_wp_outn  @ 45 : OUTPUT ;
ilev0  @ 75 : OUTPUT ;
ilev1  @ 76 : OUTPUT ;
ilev2  @ 77 : OUTPUT ;
intrn  @ 79 : OUTPUT ;
ismcr0  @ 73 : OUTPUT ;
ctlldata0  @ 31 : BIDIR ;
ctlldata1  @ 30 : BIDIR ;
ctlldata2  @ 29 : BIDIR ;
ctlldata3  @ 28 : BIDIR ;
ctlldata4  @ 27 : BIDIR ;
ctlldata5  @ 25 : BIDIR ;
ctlldata6  @ 24 : BIDIR ;
ctlldata7  @ 23 : BIDIR ;

END;

CONNECTED_PINS
BEGIN
ctloeoutn : OUTPUT ;
ctloeinn : INPUT ;
END;

END;

SUBDESIGN '13021-1'
clk : INPUT; % 32 MHz System Clock %
resetn : INPUT;

% Control Reads/Writes %
ctlrdn : INPUT; % Control Read Strobe %
ctlwrhin : INPUT; % Control Write High Strobe %
ctlwrlon : INPUT; % Control Write Low Strobe %
ctlrdnsync : OUTPUT; % Synced Control Read Strobe %
ctlwrlonsync : OUTPUT; % Synced Control Write Strobes %
ctlwrhinsync : OUTPUT;
ackn : OUTPUT; % Control Read/Write Acknowledge %
addrlen : OUTPUT; % Address Latch Enable %

ch1ackn, ch2ackn : INPUT; % Channel Control Acknowledges %

ismcrO : OUTPUT; % ISM Control Register Bit 0 %
bitcrO : INPUT; % BIT Control Register Bit 0 %

ctladdr[5..0] : INPUT; % Control Address and Data Buses %
ctldata[7..0] : BIDIR;

ctloeinn : INPUT; % Ctl Data Bus Low Output Enable %
ctloeoutn : OUTPUT;

% Interrupts %
intrn : OUTPUT; % Requests host interrupts %
ilev[2..0] : OUTPUT; % Contents of bits 4..2 of Ctl %
iackn : INPUT; % Interrupt Acknowledge %
iacknout : OUTPUT; % Dummy pin to keep iackn from %
% being removed by compiler %

% GBC Data Interface %
gbcfifoffn : INPUT; % GBC FIFO Full Flag %

ovflow1, ovflow2 : INPUT; % Address overflow Signals %

% Signals for the ID serial EEPROM. %
id_wp_outn : OUTPUT;
id_sdin : INPUT;
id_csn : OUTPUT;
id_sdout : OUTPUT;
id_sclk : OUTPUT;
id_holdn : OUTPUT;

id_wp_inn : INPUT;
id_clk_controller : MACHINE OF BITS
(dummy2, dummy1, id_sclk) WITH STATES
(id_idle = B"000",
id_write_low = B"010",
id_write_high = B"011",
id_read_low = B"100",
id_read_high = B"101",
id_sdout_lag = B"110");

% Variables for the ID register FSM. %
id_clk_counter[4..0] : DFF;
id_shift_counter[2..0] : DFFE;
id_temp[7..0] : DFFE;
id_write : DFF;
id_read : DFF;

% The following bits are used to pipeline the
id_clk_controller state machine more efficiently. %
id_clk_counter_tc : DFF;
id_shift_counter_tc : DFF;
id_lag_done : DFF;

% Synchronize reset %
resetnsync[1..0] : DFF;

% Synchronize ID EEPROM Input Signals %
id_wp_inn_sync : DFF;

BEGIN
DEFAULTS
ctloeoutn = VCC;
ackn.d = VCC;
addrilen.d = VCC;

ismcr0.ena = GND;
error[].ena = GND;
error_out[].ena = GND;

id_cmd_data[].ena = GND;
id_cmd_ctl[].ena = GND;
id_shift_counter[].ena = GND;
id_temp[].ena = GND;
id_write.d = GND;
id_read.d = GND;

130
END DEFAULTS;

% Synchronize reset %
resetnsync[].clk = GLOBAL (clk);
resetnsync[].clrn = VCC;
resetnsync[].prn = VCC;

resetnsync[1].d = resetn;
resetnsync[0].d = resetnsync[1].q;

% Interrupts %
% Keep iackn for forward compatibility %
% intrn = VCC since TAXI-ISM doesn't have interrupts %
intrn = VCC;
ilev[2..0] = GND;
iacknout = iackn;

% Synchronize Control Signals %
ctlrdsnync1.clk = GLOBAL(clk);
ctlrdsnync1.clrn = VCC;
ctlrdsnync1.prn = resetnsync[0].q;
ctlrdsnync1.clk = GLOBAL(clk);
ctlrdsnync1.clrn = VCC;
ctlrdsnync1.prn = resetnsync[0].q;

ctlrdsnync1.d = ctlrdn;
ctlrdsnync.d = ctlrdsnync1.q;

ctlwrlnonsync1.clk = GLOBAL(clk);
ctlwrlnonsync1.clrn = VCC;
ctlwrlnonsync1.prn = resetnsync[0].q;
ctlwrlnonsync1.clk = GLOBAL(clk);
ctlwrlnonsync1.clrn = VCC;
ctlwrlnonsync1.prn = resetnsync[0].q;

ctlwrlnonsync1.d = ctlwrlon;
ctlwrlnonsync.d = ctlwrlnonsync1.q;

ctlwrhinsync1.clk = GLOBAL(clk);
ctlwrhinsync1.clrn = VCC;
ctlwrhinsync1.prn = resetnsync[0].q;
ctlwrhinsync1.clk = GLOBAL(clk);
ctlwrhinsync1.clrn = VCC;
ctlwrhinsync1.prn = resetnsync[0].q;
ctlwrhinsync1.d = ctlwrhin;
ctlwrhinsync.d = ctlwrhinsync1.q;

% Control Acknowledge %
ackn.clk = GLOBAL(clk);
ackn.prn = resetnsync[0].q AND
( !ctlrdn OR !ctlwrlon OR !ctlwrhin );
ackn.clrn = VCC;

ch1acknsync[].clk = GLOBAL(clk);
ch1acknsync[].prn = resetnsync[0].q;
ch1acknsync[].clrn = VCC;

ch1acknsync[1].d = ch1ackn;
ch1acknsync[0].d = ch1acknsync[1].q;

ch2acknsync[].clk = GLOBAL(clk);
ch2acknsync[].prn = resetnsync[0].q;
ch2acknsync[].clrn = VCC;

ch2acknsync[1].d = ch2ackn;
ch2acknsync[0].d = ch2acknsync[1].q;

% Address Latch Enable %
addrlen.clk = GLOBAL(clk);
addrlen.prn = resetnsync[0].q;
addrlen.clrn = VCC;

addrlen.d = ackn AND ctlrdnsync.q AND ctlwrlonsync.q
AND ctlwrhinsync.q;

% Control enable if any reads/writes are ID Registers %
se лид = (ctladdr[5..0] == B"101111");

% Control enable if any reads/writes are Control Registers %
se lctl = (ctladdr[5..4] == 0);

% Control enable if any reads/writes are Other EPLD Regs. %
se lres = (ctladdr[5..4] == 2);

% Control enable if any reads/writes are Channel Regs. %
se lchn = ctladdr[5] AND !ctladdr[3];

% Control Writes %
writefsm.clk = GLOBAL(clk);
writefsm.reset = !resetnsync[0].q;

CASE writefsm IS
    WHEN writeidle =>
        IF (!ctlwrlonsync.q AND selctl) THEN
            writefsm = writedata;
        ELSIF (!ctlwrlonsync.q AND selchn) THEN
            writefsm = writechn;
        ELSIF (!ctlwrhinsync.q OR !ctlwrlonsync.q) THEN
            writefsm = writenotsel;
        ELSE
            writefsm = writeidle;
        END IF;
    END IF;
    WHEN writedata => % Data is written here %
        CASE ctladdr[3..0] IS
            WHEN ID_CMD_DATA_ADDR =>
                id_cmd_data[].ena = VCC;
                id_cmd_data[] = ctldata[7..0];
            WHEN ID_CMD_CTL_ADDR =>
                id_cmd_ctl[1..0].ena = VCC;
                id_cmd_ctl[1..0] = ctldata[1..0];
            WHEN 8 =>
                ismcr0.ena = VCC;
                ismcr0.d = ctldata[0];
        END CASE;
    END CASE;
    IF (ctladdr[3..0] == ID_CMD_DATA_ADDR) THEN
        writefsm = writeidwait;
    ELSE
        writefsm = writeack;
    END IF;
    WHEN writeidwait => % Wait for the X25040 to finish %
        IF (id_lag_done.q) THEN
            writefsm = writeack;
        ELSE
            writefsm = writeidwait;
        END IF;
        id_write.d = VCC;
    END CASE;
    WHEN writechn => % Wait for the channel registers %
        IF (chlacknsync[0].q AND ch2acknsync[0].q) THEN
            writefsm = writechn;
        ELSE
            ackn.d = GND;
            writefsm = writeack;
        END IF;
    END CASE;

133
END IF;
WHEN writenotsel =>
  writefsm = writeack;
WHEN writeack => % Acknowledge completion of Ctl. Wrt. %
  IF (ctlwrlnonsync.q AND ctlwrhinsync.q) THEN
    writefsm = writeidle;
  ELSE
    ackn.d = GND;
    writefsm = writeack;
  END IF;
END CASE;

% Control Reads %
ctldata[7..0] = ctldatat[7..0];
ctldatat[7..0].oe = !ctloeinn;

readfsm.clk = GLOBAL(clk);
readfsm.reset = !resetnsync[0].q;

CASE readfsm IS
WHEN readidle =>
  IF (!ctlrdnsync.q AND selctl AND
      (ctladdr[3..0] == ID_CMD_DATA_ADDR)) THEN
    readfsm = readid;
  ELSIF (!ctlrdnsync.q AND selchn) THEN
    readfsm = readchn;
  ELSIF (!ctlrdnsync.q) THEN
    readfsm = readack;
  ELSE
    readfsm = readidle;
  END IF;
WHEN readid => % Wait for the X25040 to finish %
  IF (idlag_done.q) THEN
    readfsm = readack;
  ELSE
    readfsm = readid;
  END IF;
id_read.d = VCC;
WHEN readchn => % Wait for the channel registers %
  IF (ch1acknsync[0].q AND ch2acknsync[0].q) THEN
    readfsm = readchn;
  ELSE
    readfsm = readack;
  END IF;
WHEN readack => % Acknowledge completion of Ctl. Read %

IF (selctl) THEN
  CASE ctladdr[3..0] IS
    WHEN ID_CMD_DATA_ADDR =>
      ctloeoutn = ctlrdn;
      ctldata_t[7..0] = id_cmd_data[];
    WHEN ID_CMD_CTL_ADDR =>
      ctloeoutn = ctlrdn;
      ctldata_t[7..0] =
        (B"00000",id_cmd_ctl[2..0].q);
    WHEN 8 => % ISM Control Register %
      ctloeoutn = ctlrdn;
      ctldata_t[7..0] = (B"0000000",ismcr0);
    WHEN 9 => % ISM Error Register %
      ctloeoutn = ctlrdn;
      ctldata_t[7..0] = (B"00000",
        error_out[3].q,error_out[2].q,B"00");
    WHEN H"A" => % BIT Control Register %
      ctloeoutn = VCC;
    WHEN H"B" => % BIT Base Pattern Register %
      ctloeoutn = VCC;
    WHEN others => % Undefined Registers %
      ctloeoutn = ctlrdn;
      ctldata_t[7..0] = H"00";
  END CASE;
ELSIF (selid) THEN
  ctloeoutn = ctlrdn;
  ctldata_t[7..0] = H"00";
ELSIF (selres OR selchn) THEN
  ctloeoutn = VCC;
ELSE
  ctloeoutn = ctlrdn;
  ctldata_t[7..0] = H"00";
END IF;

% Wait until sync registers are clear %
IF (ctlrdnsync.q) THEN
  readfsm = readidle;
ELSE
  ackn.d = GND;
  readfsm = readack;
END IF;
END CASE;

% ISM Control Register %
ismcr0.clk  = GLOBAL(clk);
ismcr0.clrn  = resetnsync0.q;
ismcr0.prn  = VCC;

% Error Register 
% Error register only holds gbcfifo overflow error. 
% Other errors in fo channel sequencer EPLDs 
error[].clk = GLOBAL(clk);
error[].clrn = resetnsync0.q;
error[].prn = VCC;
error_out[].clk = GLOBAL(clk);
error_out[].clrn = resetnsync0.q;
error_out[].prn = VCC;

errorfsm.clk  = GLOBAL(clk);
errorfsm.reset = !resetnsync0.q;

CASE (errorfsm) IS
  WHEN eridle =>
    IF (ismcr0.q) THEN
      errorfsm = ernoti;
    ELSIF (!ctlrdnsync.q AND selctl AND
      (ctladdr[3..0] == H"9") ) THEN
      errorfsm = eriwait;
      error_out[].ena = VCC;
      error_out[3].d = error[3].q;
      error_out[2].d = error[2].q;
    ELSIF (!ctlwrlonsync.q AND selctl AND
      (ctladdr[3..0] == H"9") ) THEN
      errorfsm = eriwait;
      error[].ena = VCC;
    ELSIF (!ctlrdnsync.q OR !ctlwrlonsync.q) THEN
      errorfsm = eriwait;
    ELSE
      errorfsm = eridle;
    END IF;
  WHEN eriwait =>
    IF (ctlwrlonsync.q AND ctlrdnsync.q) THEN
      errorfsm = eridle;
    ELSE
      errorfsm = eriwait;
    END IF;
  WHEN ernoti =>

136
IF (!ismcr0.q) THEN
  error[3].d = error[3].q OR !gbcfifoffn;
  error[2].d = error[2].q OR !gbcfifoffn OR ovflow1 OR ovflow2;
  errorfsm = eridle;
ELSIF (!ctlrdnsync.q AND selctl AND (ctladdr[3..0] == H"9")] THEN
  error[3].d = error[3].q OR !gbcfifoffn;
  error[2].d = error[2].q OR !gbcfifoffn OR ovflow1 OR ovflow2;

  error_out[].ena = VCC;
  error_out[].d = error[].q;

  errorfsm = erniwait;
ELSIF (!ctlwrlnonsync.q AND selctl AND (ctladdr[3..0] == H"9") THEN

  errorfsm = erniwait;
ELSIF (!ctlrdnsync.q OR !ctlwrlnonsync.q) THEN
  error[3].d = error[3].q OR !gbcfifoffn;
  error[2].d = error[2].q OR !gbcfifoffn OR ovflow1 OR ovflow2;

  errorfsm = ernoti;
ELSE
  error[3].d = error[3].q OR !gbcfifoffn;
  error[2].d = error[2].q OR !gbcfifoffn OR ovflow1 OR ovflow2;
  errorfsm = ernoti;
END IF;

error[].ena = VCC;
WHEN erniwait =>
  IF (ctlwrlnonsync.q AND ctlrdnsync.q) THEN
    errorfsm = ernoti;
  ELSE
    errorfsm = erniwait;
  END IF;

error[].ena = VCC;
error[3].d = error[3].q OR !gbcfifoffn;
error[2].d = error[2].q OR !gbcfifoffn OR
ovflow1 OR ovflow2;

END CASE;

% ************************************************* %
% The following section contains the implementation %
% of the Standard PWB Identification Device. %
% ************************************************* %

% Define the clocks, resets, and presets %
% for the ID Control registers. %
id_cmd_data[].clk = GLOBAL (clk);
$id_cmd_data[].clrn = resetnsync[0].q;
$id_cmd_data[].prn = VCC;

$id_cmd_ctl[].clk = GLOBAL (clk);
$id_cmd_ctl[].clrn = resetnsync[0].q;
$id_cmd_ctl[].prn = VCC;

$id_wp_inn_sync.clk = GLOBAL(clk);
$id_wp_inn_sync.clrn = resetnsync[0].q;
$id_wp_inn_sync.prn = VCC;
$id_wp_inn_sync.d = id_wp_inn;

% Serial EEPROM /CS Signal %
$id_csn = !id_cmd_ctl[0].q;

% Serial EEPROM /HOLD Signal %
$id_holdn = VCC;

% Serial EEPROM /WP Signal %
$id_wp_outn = id_cmd_ctl[1].q;
$id_cmd_ctl[2].d = id_wp_inn_sync.q;
$id_cmd_ctl[2].ena = VCC;

% Serial EEPROM SO Signal %
% Serial data out always = the MSB of the %
% temporary shift register. %
$id_sdout = id_temp[7];

% Define clocks, resets, and presets for the fsm counters %
id_clk_counter[].clk = GLOBAL (clk);
id_clk_counter[].prn = VCC;
!id_clk_counter[].clrn = !resetnsync[0].q;

id_shift_counter[].clk = GLOBAL (clk);
id_shift_counter[].prn = VCC;
!id_shift_counter[].clrn = !resetnsync[0].q;

id_temp[].clk = GLOBAL (clk);
id_temp[].prn = VCC;
!id_temp[].clrn = !resetnsync[0].q;

% Set up a bit to mark terminal count of clock counter so that we don’t have to compare all four bits all the time. %
id_clk_counter_tc.d = (id_clk_counter[] == 1) #
   ((id_clk_controller == id_sdout_lag) &
   (id_clk_counter_tc));

id_clk_counter_tc.clk = GLOBAL (clk);
id_clk_counter_tc.clrn = resetnsync[0].q;
id_clk_counter_tc.prn = VCC;

% Set up a bit to mark the terminal count of the shift counter so that we don’t have to compare all three bits all of the time. %
IF (id_shift_counter[] == 0) THEN
   id_shift_counter_tc = VCC;
END IF;

id_shift_counter_tc.clk = GLOBAL (clk);
id_shift_counter_tc.clrn = resetnsync[0].q;
id_shift_counter_tc.prn = VCC;

% Set up a bit to mark the end of the Tlag after a serial interface with the X25040 %
id_lag_done.clk = GLOBAL(clk);
id_lag_done.clrn = resetnsync[0].q;
id_lag_done.prn = VCC;

% Set up a bit to tell the ID_CLK_CONTROLLER to start writing/reading %
id_write.clk = GLOBAL(clk);
id_write.clrn = resetnsync[0].q;
id_write.prn = VCC;

id_read.clk = GLOBAL(clk);
id_read.clrn = resetnsync[0].q;
id_read.prn = VCC;

% ID Controller FSM %
id_clk_controller.clk = GLOBAL (clk);
id_clk_controller.reset = !resetnsync[0].q;

CASE id_clk_controller IS
  WHEN id_idle =>
    IF (id_write.q) THEN
      id_temp[].ena = VCC;
      id_temp[] = id_cmd_data[];
      id_clk_controller = id_write_low;
    ELSIF (id_read.q) THEN
      id_clk_controller = id_read_low;
    ELSE
      id_clk_controller = id_idle;
    END IF;
  END IF;
  id_clk_counter[] = ID_CLK_DIVISOR;
  id_shift_counter[].ena = VCC;
  id_shift_counter[] = id_shift_COUNT;
  WHEN id_write_low =>
    IF (!id_clk_counter_tc) THEN
      id_clk_counter[] = id_clk_counter[] - 1;
    ELSE
      id_clk_counter[] = ID_CLK_DIVISOR;
      id_clk_controller = id_write_high;
    END IF;
  END IF;
  WHEN id_write_high =>
    IF (!id_clk_counter_tc) THEN
      id_clk_counter[] = id_clk_counter[] - 1;
    ELSE
      IF (!id_shift_counter_tc) THEN
        id_temp[].ena = VCC;
        id_temp[] = (id_temp[6..0], id_temp[7]);
        id_shift_counter[].ena = VCC;
        id_shift_counter[] = id_shift_counter[] - 1;
        id_clk_counter[] = ID_CLK_DIVISOR;
        id_clk_controller = id_write_low;
      ELSE
        id_clk_counter[] = ID_CLK_DIVISOR;
        id_clk_controller = id_sdout_lag;
      END IF;
    END IF;
  END IF;
END CASE;
WHEN id_read_low =>
  IF (!id_clk_counter_tc) THEN
    id_clk_counter[] = id_clk_counter[] - 1;
  ELSE
    id_clk_counter[] = ID_CLK_DIVISOR;
    id_temp[].ena = VCC;
    id_temp[] = (id_temp[6..0], id_sdin);
    id_clk_controller = id_read_high;
  END IF;
WHEN id_read_high =>
  IF (!id_clk_counter_tc) THEN
    id_cmd_data[].ena = VCC;
    id_cmd_data[] = id_temp[];
    id_clk_counter[] = id_clk_counter[] - 1;
  ELSE
    IF (id_shift_counter_tc) THEN
      id_clk_counter[] = ID_CLK_DIVISOR;
      id_clk_controller = id_sdout_lag;
    ELSE
      id_shift_counter[].ena = VCC;
      id_shift_counter[] = id_shift_counter[] - 1;
      id_clk_counter[] = ID_CLK_DIVISOR;
      id_clk_controller = id_read_low;
    END IF;
  END IF;
WHEN id_sdout_lag =>
  IF (!id_clk_counter_tc) THEN
    id_clk_counter[] = id_clk_counter[] - 1;
    id_clk_controller = id_sdout_lag;
  ELSIF (id_write.q OR id_read.q) THEN
    id_lag_done.d = VCC;
    id_clk_controller = id_sdout_lag;
  ELSE
    id_clk_counter[] = ID_CLK_DIVISOR;
    id_clk_controller = id_idle;
  END IF;
END CASE;
END;
C.3 Data Module

C.3.1 Arbiter: 12848

TITLE "12848-";

%******************************************************************************
** **
** DWG 12848 REV -
** Project: TAXI Fiber Optic Interface Support Module **
** Designer: Stanley C. Wang **
** **
** Description: EPLD design file for Arbiter Function. **
** Used on ASTB Data Multiplexer II TAXI Fiber **
** Optic Interface Support Module (TAXI-ISM) **
** **
******************************************************************************%

DESIGN IS "12848-"
BEGIN

DEVICE "12848-" IS "EPM7160LC84-10"
BEGIN
  bitcr0 @ 54 : INPUT;
  clk @ 83 : INPUT;
  cpistart @ 74 : INPUT;
  fifo1efn @ 33 : INPUT;
  fifo1hfn @ 34 : INPUT;
  fifo2efn @ 40 : INPUT;
  fifo2hfn @ 41 : INPUT;
  gbcfifoffn @ 60 : INPUT;
  ismcr0 @ 55 : INPUT;
  resetn @ 73 : INPUT;
  cpisync @ 31 : OUTPUT;
  fifo1oen @ 35 : OUTPUT;
  fifo1rdn @ 36 : OUTPUT;
  fifo2oen @ 44 : OUTPUT;
  fifo2rdn @ 45 : OUTPUT;
  gbcfifowrtm @ 61 : OUTPUT;
  ovflow1 @ 70 : OUTPUT;
  ovflow2 @ 52 : OUTPUT;
  sa0 @ 23 : OUTPUT;
  sa1 @ 22 : OUTPUT;

142
SUBDESIGN '12848-'
(
resetn : INPUT;
clk : INPUT;

% Control Interface %
cpistart : INPUT; % CPI Start Signal %
cpisync : OUTPUT; % Synchronized CPISSTART signal %

ismcr0 : INPUT; % ISM Control Register Bit 0 %
bitcr0 : INPUT; % BIT Control Register Bit 0 %

% FIFO Control %
fifolefn, fifo2efn : INPUT; % FIFO Empty Flags %
fifo1hfn, fifo2hfn : INPUT; % FIFO Half Empty Flags %
fifo1rdn, fifo2rdn : OUTPUT; % FIFO READ Signals %
fifoloen, fifo2oen : OUTPUT; % Output Enable for FIFO Regs. %
VARIABLE

% Synchronize Inputs %
resetnsync0 : DFF;
resetnsync1 : DFF;
cpistartsync[2..0] : DFF;
gbcfifoffnsync[1..0] : DFF;
bitstartsnc[1..0] : DFF; % Start BIT Marker %

fifolefnsync[1..0] : DFF; % Channel 1 FIFO Status Flags %
fifolhfnsc[1..0] : DFF; % Channel 2 Synced in 13022 %

% Arbiter %
rdy1, rdy2 : SOFT;

fsm : MACHINE OF BITS
(addrinc, fifo2rdn, fifo2enfsm, fifo1rdn,
fifo1enfsm, gbcfifowrtn, sack2, sack1)

WITH STATES (idle = B"01111111",
newcpi1 = B"01111110",
newcpi2 = B"01111011",
overflow = B"00101111",
over1 = B"01101111",
over2 = B"00111111",
selllov2 = B"00100111",
sellhov2 = B"11110011",
selllov1 = B"00001111",
sellhov1 = B"11011011",
selllo = B"01100101",
sellh = B"11110001",
sel2l = B"00011110",
sel2h = B"11011010",
gbcfull = B"00101100",
gbcfull1 = B"01111100");

gbcfullclk[3..0] : DFF;
cpisave : SRFF;

% Error Indicators %
ovflow1, ovflow2 : SOFT;

BEGIN
DEFAULTS

% Data Interface %
sa[21..0].clr = VCC;
sa[21..0].pr = VCC;
sc[3..0] = GND;
sc_q[0].d = GND;
sc_q[0].clr = VCC;
sc_q[0].pr = VCC;
car1[21..0].clr = VCC;
car1[21..0].pr = VCC;
car2[22..0].clr = VCC;
car2[22..0].pr = VCC;
fifo2oen.clr = VCC;
fifo2oen.pr = VCC;
fifoloen.clr = VCC;
fifoloen.pr = VCC;
gbcfullclk[3..0].clr = VCC;
gbcfullclk[3..0].pr = VCC;
cpisave.clr = VCC;
cpisave.pr = VCC;
cpisave.s = GND;
cpisave.r = GND;

% Synchronize Inputs %
resetnsync0.clr = VCC;
resetnsync0.pr = VCC;
resetnsync1.clr = VCC;
resetnsync1.pr = VCC;
cpistartsync[2..0].clr = VCC;
cpistartsync[2..0].pr = VCC;
gbcfifo0nsync[] .clr = VCC;
gbcfifoffnsync[] .prn = VCC;
bitstartsync[1..0].clrn = VCC;
bitstartsync[1..0].prn = VCC;

fifo1efnsync[1..0].clrn = VCC;
fifo1efnsync[1..0].prn = VCC;
fifo1hfnsync[1..0].clrn = VCC;
fifo1hfnsync[1..0].prn = VCC;

END DEFAULTS;

% Synchronize Reset %
resetnsync0 .clk = GLOBAL(clk);
resetnsync1 .clk = GLOBAL(clk);
resetnsync1 .d = resetn;
resetnsync0 .d = resetnsync1 .q;

% GBCFIFOFFN synchronization %
gbcfifoffnsync[] .clk = GLOBAL(clk);
gbcfifoffnsync[] .prn = resetnsync0 .q;
gbcfifoffnsync[1] .d = gbcfifoffn;
gbcfifoffnsync[0] .d = gbcfifoffnsync[1] .q;

% CPSTART Signal %
cpistartsync[2] .clk = !cpistart;
cpistartsync[1..0] .clk = GLOBAL(clk);
cpistartsync[1..0] .clrn = resetnsync0 .q;
cpistartsync[2] .d = VCC;
cpistartsync[0] .d = cpistartsync[1] .q;
cpistartsync[2] .clrn = !cpistartsync[0] .q;
cpisync = cpistartsync[0] .q;

% BIT Start Marker %
bitstartsync[] .clk = GLOBAL(clk);

bitstartsync[1] .d = bitcr0;
bitstartsync[0] .d = bitcr0 AND !bitstartsync[1] .q;

% Channel 1 Status Flags %
fifo1efnsync[] .clk = GLOBAL(clk);
fifo1efnsync[] .clrn = fifo1efn AND resetnsync0 .q;
fifolefnsync[1].d = fifolefn;
fifolefnsync[0].d = fifolefnsync[1].q;

fifo1hfnsync[].clk = GLOBAL(clk);
fifo1hfnsync[].clr = resetnsync0.q;

fifo1hfnsync[1].d = fifo1hfn;
fifo1hfnsync[0].d = fifo1hfnsync[1].q;

% Address Generators %
sa[21..0].clk = GLOBAL(clk);
sa[21..0].clr = resetnsync0.q;
car1[21..0].clk = GLOBAL(clk);
car1[21..0].clr = resetnsync0.q;
car2[22..0].clk = GLOBAL(clk);
car2[22..0].clr = resetnsync0.q;
car2[21].clr = !((resetnsync0.q AND bitstartsnc[0].q);
car2[21].prn = resetnsync0.q;
car2[22].clr = resetnsync0.q;

ovflow1 = (car1[21] == B"1") & ismcr0;
ovflow2 = (car2[22] == B"1") & ismcr0;

IF (cpistartsync[0].q AND (fsm != sel11)) THEN
  car1[21..0] = 0;
ELSIF (addrinc & sack1) THEN
  car1[21..0] = car1[21..0] + 1;
ELSE
  car1[21..0] = car1[21..0];
END IF;

IF (cpistartsync[0].q AND (fsm != sel21) AND bitcr0) THEN
  car2[22..0] = H"000000"
ELSIF (cpistartsync[0].q AND (fsm != sel21)) THEN
  car2[22..0] = H"200000"
ELSIF (addrinc & sack2) THEN
  car2[22..0] = car2[22..0] + 1;
ELSE
  car2[22..0] = car2[22..0];
END IF;

% Sample Control Bus %
sc[3..1] = GND;
sc[0] = sc_q[0].q;
sc_q[0].clk = GLOBAL(clk);
% Round-robin arbiter %
rdy1 = ((sack2 & fifo1hfn!sync[0].q & fifo2hfn) #
     (sack2 & !fifo1hfn!sync[0].q) #
     (!fifo1hfn!sync[0].q & fifo2hfn) # !fifo2efn)
& fifo1efn!sync[0].q & !ovflow1 & gbcfifoffn
& !bitcr0 & ismcr0;

rdy2 = ((sack1 & fifo2hfn & fifo1hfn!sync[0].q) #
     (sack1 & !fifo2hfn) #
     (!fifo2hfn & fifo1hfn!sync[0].q) # !fifo1efn!sync[0].q
     # bitcr0) & fifo2efn & !ovflow2 & gbcfifoffn & ismcr0;

% Define FSM states %
fsm.clk = GLOBAL(clk);
fsm.reset = !resetnsync[0].q;

CASE (fsm) IS
    WHEN idle =>
        cpisave.r = VCC;
        IF (cpisave.q OR cpistartsync[0].q) THEN
            fsm = newcpi1;
        ELSIF (!gbcfifoffn!sync[0].q) THEN
            fsm = gbcfull;
        ELSIF (rdy1 AND ovflow2) THEN
            fsm = sel1lov2;
        ELSIF (rdy2 AND ovflow1) THEN
            fsm = sel2lov1;
        ELSIF (rdy1) THEN
            fsm = sel1;
        ELSIF (rdy2) THEN
            fsm = sel2;
        ELSIF (ovflow1 AND ovflow2) THEN
            fsm = overflow;
        ELSIF (ovflow1) THEN
            fsm = over1;
        ELSIF (ovflow2 AND !bitcr0) THEN
            fsm = over2;
        ELSE
            fsm = idle;
        END IF;
    WHEN newcpi1 =>
        sc_q[0].d = B"1";
        cpisave.r = VCC;
fsm = newcpi2;

WHEN newcpi2 =>
    sc_q[0].d = B"1";
    cpisave.r = VCC;

IF (cpistartsync[0].q) THEN
    fsm = newcpi2;
ELSFIF (!gbcfifoffnsync[0].q) THEN
    fsm = gbcfull;
ELSIF (rdy1 AND ovflow2) THEN
    fsm = sel1lov2;
ELSIF (rdy2 AND ovflow1) THEN
    fsm = sel2lov1;
ELSIF (rdy1) THEN
    fsm = sel1l;
ELSIF (rdy2) THEN
    fsm = sel2l;
ELSIF (ovflow1 AND ovflow2) THEN
    fsm = overflow;
ELSIF (ovflow1) THEN
    fsm = over1;
ELSIF (ovflow2 AND !bitcr0) THEN
    fsm = over2;
ELSE
    fsm = idle;
END IF;

WHEN overflow =>
    fsm = idle;

WHEN over1 =>
    fsm = idle;

WHEN over2 =>
    fsm = idle;

WHEN sel1lov2 =>
    sa[21..0] = car1[21..0];

    IF (ovflow1) THEN
        fsm = idle;
    ELSE
        fsm = sel1hov2;
    END IF;

WHEN sel1hov2 =>
    sa[21..0] = car1[21..0];

    IF (cpistartsync[0].q) THEN
fsm = newcpil;
ELSIF (!gbcfifoffnsync[0].q) THEN
    fsm = gbcfull;
ELSIF (rdy1) THEN
    fsm = sel1lov2;
ELSIF (ovflow1) THEN
    fsm = overflow;
ELSE
    fsm = over2;
END IF;
WHEN sel2lov1 =>
    sa[21..0] = car2[21..0];

IF (ovflow2) THEN
    fsm = idle;
ELSE
    fsm = sel2hov1;
END IF;
WHEN sel2hov1 =>
    sa[21..0] = car2[21..0];

IF (cpistartsync[0].q) THEN
    fsm = newcpil;
ELSIF (!gbcfifoffnsync[0].q) THEN
    fsm = gbcfull;
ELSIF (rdy2) THEN
    fsm = sel2lov1;
ELSIF (ovflow2) THEN
    fsm = overflow;
ELSE
    fsm = over1;
END IF;
WHEN sel11 =>
    sa[21..0] = car1[21..0];

IF (ovflow1) THEN
    fsm = idle;
ELSIF (!gbcfifoffnsync[0].q) THEN
    fsm = gbcfull;
ELSE
    fsm = sel1h;
END IF;
WHEN sel1h =>
    sa[21..0] = car1[21..0];
IF (cpistartsync[0].q) THEN
   fsm = newcpil;
ELSIF (gbcfifoffnsync[0].q) THEN
   fsm = gbcfull;
ELSIF (rdy1 AND ovflow2) THEN
   fsm = sel1lov2;
ELSIF (rdy2 AND ovflow1) THEN
   fsm = sel2lov1;
ELSIF (rdy1) THEN
   fsm = sel1l;
ELSIF (rdy2) THEN
   fsm = sel2l;
ELSIF (ovflow1 AND ovflow2) THEN
   fsm = overflow;
ELSIF (ovflow1) THEN
   fsm = over1;
ELSIF (ovflow2) THEN
   fsm = over2;
ELSE
   fsm = idle;
END IF;
WHEN sel2l =>
  sa[21..0] = car2[21..0];

   IF (ovflow2) THEN
       fsm = idle;
   ELSIF (gbcfifoffnsync[0].q) THEN
       fsm = gbcfull;
   ELSE
       fsm = sel2h;
   END IF;
END IF;
WHEN sel2h =>
  sa[21..0] = car2[21..0];

   IF (cpistartsync[0].q) THEN
       fsm = newcpil;
   ELSIF (gbcfifoffnsync[0].q) THEN
       fsm = gbcfull;
   ELSIF (rdy1 AND ovflow2) THEN
       fsm = sel1lov2;
   ELSIF (rdy2 AND ovflow1) THEN
       fsm = sel2lov1;
   ELSIF (rdy1) THEN
       fsm = sel1l;
   ELSIF (rdy2) THEN

fsm = sel21;
ELSIF (ovflow1 AND ovflow2) THEN
  fsm = overflow;
ELSIF (ovflow1) THEN
  fsm = over1;
ELSIF (ovflow2 AND !bitcr0) THEN
  fsm = over2;
ELSE
  fsm = idle;
END IF;
WHEN gbcfull =>
  IF (cpistartsync[0].q) THEN
    cpisave.s = VCC;
  END IF;

  IF (gbcfifoffnsync[0].q) THEN
    gbcfullclk[] = gbcfullclk[];
  ELSE
    gbcfullclk[] = GND;
  END IF;

  fsm = gbcfull1;
WHEN gbcfull1 =>
  IF (cpistartsync[0].q) THEN
    cpisave.s = VCC;
  END IF;

  IF (!gbcfifoffnsync[0].q) THEN
    gbcfullclk[] = GND;
    fsm = gbcfull;
  ELSIF (gbcfullclk[] < 8) THEN
    gbcfullclk[] = gbcfullclk[] + 1;
    fsm = gbcfull;
  ELSE
    fsm = idle;
  END IF;
  WHEN others =>
    fsm = idle;
END CASE;
gbcfullclk[3..0].clk = GLOBAL(clk);
cpisave.clk = GLOBAL(clk);
fifo2oen.clk = GLOBAL(clk);
fifo1oen.clk = GLOBAL(clk);
fif02oen.prn = resetnsync0.q;
fifo1oen.prn = resetnsync0.q;

IF (ovflow1) THEN
    fif01oen.d = VCC;
ELSE
    fif01oen.d = fif01oenfsm;
END IF;

IF (ovflow2) THEN
    fifo2oen.d = VCC;
ELSE
    fifo2oen.d = fifo2oenfsm;
END IF;
END;
C.3.2 Ending Address Register Controller: 13035

TITLE "13035-";

******************************
** **
** DWG 13035 REV - **
** Project: TAXI Fiber Optic Interface Support Module **
** Designer: Stanley C. Wang **
**
** Description: EPLD design file for Ending Address Register **
** Control. Used on ASTB Data Multiplexer II **
** TAXI Fiber Optic Interface Support Module **
** (TAXI-ISM) **
**
******************************%

DESIGN IS "13035-"
BEGIN

DEVICE "13035-" IS "EPM7128LC84-10"
BEGIN

clk @ 83 : INPUT ;
cpistartsync @ 65 : INPUT ;
ctladder0 @ 67 : INPUT ;
ctladder1 @ 68 : INPUT ;
ctladder2 @ 69 : INPUT ;
ctladder3 @ 70 : INPUT ;
ctladder4 @ 71 : INPUT ;
ctladder5 @ 73 : INPUT ;
ctlhioeinn @ 84 : INPUT ;
ctllooeinn @ 2 : INPUT ;
ctlrdnsync @ 74 : INPUT ;
gbcfifowrtm @ 64 : INPUT ;
resetn @ 75 : INPUT ;
sa0 @ 33 : INPUT ;
sa1 @ 34 : INPUT ;
sa2 @ 35 : INPUT ;
sa3 @ 36 : INPUT ;
sa4 @ 37 : INPUT ;
sa5 @ 39 : INPUT ;
sa6 @ 40 : INPUT ;
sa7 @ 41 : INPUT ;

154
sa8  @  44  :  INPUT  ;
sa9  @  45  :  INPUT  ;
sa10 @  46  :  INPUT  ;
sa11 @  48  :  INPUT  ;
sa12 @  49  :  INPUT  ;
sa13 @  50  :  INPUT  ;
sa14 @  51  :  INPUT  ;
sa15 @  52  :  INPUT  ;
sa16 @  54  :  INPUT  ;
sa17 @  55  :  INPUT  ;
sa18 @  56  :  INPUT  ;
sa19 @  57  :  INPUT  ;
sa20 @  58  :  INPUT  ;
sa21 @  60  :  INPUT  ;
ctlhioeoutn @  81  :  OUTPUT;
ctlhioeoutn @   4  :  OUTPUT;
ctlldata0 @  31  :  BIDIR  ;
ctlldata1 @  30  :  BIDIR  ;
ctlldata2 @  29  :  BIDIR  ;
ctlldata3 @  28  :  BIDIR  ;
ctlldata4 @  22  :  BIDIR  ;
ctlldata5 @  21  :  BIDIR  ;
ctlldata6 @  20  :  BIDIR  ;
ctlldata7 @  18  :  BIDIR  ;
ctlldata8 @  17  :  BIDIR  ;
ctlldata9 @  12  :  BIDIR  ;
ctlldata10 @  11  :  BIDIR  ;
ctlldata11 @  10  :  BIDIR  ;
ctlldata12 @   9  :  BIDIR  ;
ctlldata13 @   8  :  BIDIR  ;
ctlldata14 @   6  :  BIDIR  ;
ctlldata15 @   5  :  BIDIR  ;

END;

CONNECTED_PINS
BEGIN
  ctllooeoutn : OUTPUT;
  ctllooeinn  : INPUT  ;
END;

CONNECTED_PINS
BEGIN
  ctlhioeoutn : OUTPUT;
  ctlhioeinn  : INPUT  ;
END;
% Control Interface %
ctladdr[5..0] : INPUT; % Control Address Bus %
ctldata[15..0] : BIDIR; % Control Data Bus %
ctldnsync : INPUT; % Synced Control Read Strobe %

cctllooeinn : INPUT; % Ctl. Data Bus Low Output Enable %
cctllooeoutn : OUTPUT;
cctlhioeinn : INPUT; % Ctl. Data Bus Hi Output Enable %
cctlhioeoutn : OUTPUT;

% Data Interface %
sa[21..0] : INPUT; % Sample Address Bus %
gbcfifowrttn : INPUT; % GBC Data Write Strobe %

cpistartsync : INPUT; % Synced CPISTART signal from 12848 %

VARIABLE
% Control Interface %
ctldata_t[15..0] : TRI;

readfsm : MACHINE WITH STATES (readidle, readdata, readwait);

% Data Interface %

ear1[21..0] : DFFE; % Ending Address Registers %

% Synchronize Inputs %
resetnsync0 : DFF;
resetnsync1 : DFF;
BEGIN

DEFAULTS

% Control Interface %
ctllooeoutn = VCC;
ctlhioeoutn = VCC;

% Data Interface %
ear1[].clrn = VCC;
ear1[].prn = VCC;
ear1[].ena = GND;
ear2[].clrn = VCC;
ear2[].prn = VCC;
ear2[].ena = GND;
temp1[].clrn = VCC;
temp1[].prn = VCC;
temp2[].clrn = VCC;
temp2[].prn = VCC;

loader[].prn = VCC;
loader[].clrn = VCC;

% Synchronize Inputs %
resetnsync0.clrn = VCC;
resetnsync0.prn = VCC;
resetnsync1.clrn = VCC;
resetnsync1.prn = VCC;

END DEFAULTS;

% Synchronize Reset %
resetnsync0.clk = GLOBAL(clk);
resetnsync1.clk = GLOBAL(clk);
resetnsync1.d = resetn;
resetnsync0.d = resetnsync1.q;

% Control enable if any control reads/writes %
% are 12848 Registers %
selctl = (ctladdr[5..3] == B"101");

% Enable if any control reads/writes are BIT Registers %
selbit = (ctladdr[5..1] == B"00101");

% Control Reads %
ctldata[15..0] = ctldata_t[15..0];
ctldata_t[7..0].oe = !ctllooeinn;
ctldata_t[15..8].oe = !ctlhioeinn;

readfsm.clk = GLOBAL(clk);
readfsm.reset = !resetnsync[0].q;

CASE readfsm IS
  WHEN readidle =>
    IF (!ctrlrdnsync AND selctl) THEN
      CASE ctladdr[2..0] IS
        WHEN 0 => % EAR1L Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] = earl[15..0];
        WHEN 1 => % EAR1H Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] =
            (H"00", B"00", earl[21..16]);
        WHEN 2 => % EAR2L Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] = ear2[15..0];
        WHEN 3 => % EAR2H Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] =
            (H"00", B"0", ear2[22..16]);
        WHEN 4 => % EPLD ID Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] = H"0000";
        WHEN 5 => % Undefined Register %
          ctllooeoutn = GND;
          ctlhioeoutn = GND;
          ctldata_t[15..0] = H"0000";
        WHEN 6 => % 13022 ID Register %
          ctllooeoutn = VCC;
          ctlhioeoutn = VCC;
        WHEN 7 => % 13021 ID Register %
          ctllooeoutn = VCC;
          ctlhioeoutn = GND;
          ctldata_t[15..8] = H"00";
      END CASE;
  END CASE;
readfsm = readdata;
ELSIF (!ctlrdnsync) THEN
  readfsm = readwait;
ELSE
  readfsm = readidle;
END IF;
WHEN readdata => % Acknowledge completion of Read %
CASE ctlassr[2..0] IS
  WHEN 0 => % EAR1L Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] = ear1[15..0];
  WHEN 1 => % EAR1H Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] =
      (H"00", B"00", ear1[21..16]);
  WHEN 2 => % EAR2L Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] = ear2[15..0];
  WHEN 3 => % EAR2H Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] =
      (H"00", B"0", ear2[22..16]);
  WHEN 4 => % EPLD ID Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] = H"0000";
  WHEN 5 => % Undefined Register %
    ctllooeoutn = ctlrdnsync;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..0] = H"0000";
  WHEN 6 => % 13022 ID Register %
    ctllooeoutn = VCC;
    ctlhioeoutn = VCC;
  WHEN 7 => % 13021 ID Register %
    ctllooeoutn = VCC;
    ctlhioeoutn = ctlrdnsync;
    ctdata_t[15..8] = H"00";
END CASE;
% Wait until sync registers are clear %
IF (ctlrdnsync) THEN
  readfsm = readidle;
ELSE
  readfsm = readdata;
END IF;
WHEN readwait =>
  IF (selbit) THEN
    ctlhioeoutn = VCC;
  ELSE
    ctlhioeoutn = ctlrdnsync;
    ctldata_t[15..8] = H"00";
  END IF;

IF (ctlrdnsync) THEN
  readfsm = readidle;
ELSE
  readfsm = readwait;
END IF;
END CASE;

% Address Generators %
% EAR1 set to Ox00000 after CPISTART %
% EAR2 set to Ox20000 after CPISTART %
ear1[].clk = GLOBAL(clk);
ear1[21..0].clrn = resetnsync0.q;

% TEMP1 set to Ox00000 - 1 after CPISTART %
% TEMP2 set to Ox20000 - 1 after CPISTART %
temp1[].clk = gbcfifowrtn;
temp1[].prn = resetnsync0.q;
temp2[].clk = gbcfifowrtn;
temp2[20..0].prn = resetnsync0.q;
temp2[22..21].clrn = resetnsync0.q;

loader[].clk = GLOBAL(clk);

IF (cpistartsync) THEN
  loader[1].d = VCC;
ELSIF (gbcfifowrtn) THEN
  loader[1].d = loader[1].q;
ELSE

160
loader[1].d = GND;
END IF;

loader[0].d = loader[1].q;

IF (loader[0].q) THEN
  temp2[22..0] = H"1FFFFF";
  temp1[21..0] = H"3FFFFF";
ELSIF (sa[21]) THEN
  temp2[21..0].d = sa[];
  temp2[22].d = GND;
  temp1[].d = temp1[].q;
ELSE
  temp1[].d = sa[];
  temp2[].d = temp2[].q;
END IF;

% EARs store NEXT address to write to %
IF (cpistartsync) THEN
  ear1[].ena = VCC;
  ear1[].d = temp1[].q + 1;
  ear2[].ena = VCC;
  ear2[].d = temp2[].q + 1;
END IF;
END;
C.3.3 Built-in Test Controller: 13022

TITLE "13022-";

%*******************************************************************************
**                     **
** DWG 13022 REV -       **
** Project: TAXI Fiber Optic Interface Support Module **
** Designer: Stanley C. Wang **
**                     **
** Description: EPLD design file for the BIT Function. Used **
** on ASTB Data Multiplexer II TAXI Fiber Optic **
** Interface Support Module (TAXI-ISM) **
**                     **
*******************************************************************************%

% Addresses of the BIT registers %
CONSTANT BITCR_ADDR = B"0";
CONSTANT BITBPR_ADDR = B"1";

DESIGN IS "13022-"
BEGIN

DEVICE "13022-" IS "EPM7160LC84-10"
BEGIN

clk @ 83 : INPUT ;
ctladdr0 @ 60 : INPUT ;
ctladdr1 @ 61 : INPUT ;
ctladdr2 @ 62 : INPUT ;
ctladdr3 @ 63 : INPUT ;
ctladdr4 @ 64 : INPUT ;
ctladdr5 @ 65 : INPUT ;
cloeinn @ 84 : INPUT ;
cltrdnsync @ 11 : INPUT ;
ctlwrhinsync @ 10 : INPUT ;
ctlwrlnsync @ 12 : INPUT ;
fifo2efn7in @ 77 : INPUT ;
fifo2hfn7in @ 76 : INPUT ;
fifo2oen8in @ 70 : INPUT ;
fifo2rdn8in @ 69 : INPUT ;
ismcr0 @ 73 : INPUT ;
resetn @ 1 : INPUT ;
sdoeinn @ 2 : INPUT ;

162
<table>
<thead>
<tr>
<th>Pin</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitcr0</td>
<td>71</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>ctloeoutn</td>
<td>81</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>fifo2efn8out</td>
<td>68</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>fifo2hfn8out</td>
<td>67</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>fifo2oen7out</td>
<td>75</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>fifo2rdn7out</td>
<td>74</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sdoeoutn</td>
<td>4</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd0</td>
<td>33</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd1</td>
<td>34</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd2</td>
<td>35</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd3</td>
<td>36</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd4</td>
<td>37</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd5</td>
<td>40</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd6</td>
<td>41</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd7</td>
<td>44</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd8</td>
<td>45</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd9</td>
<td>48</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd10</td>
<td>49</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd11</td>
<td>50</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd12</td>
<td>51</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd13</td>
<td>52</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd14</td>
<td>54</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>sd15</td>
<td>55</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>ctlldata0</td>
<td>31</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata1</td>
<td>30</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata2</td>
<td>29</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata3</td>
<td>28</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata4</td>
<td>27</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata5</td>
<td>25</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata6</td>
<td>24</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata7</td>
<td>23</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata8</td>
<td>22</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata9</td>
<td>21</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata10</td>
<td>20</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata11</td>
<td>18</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata12</td>
<td>17</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata13</td>
<td>16</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata14</td>
<td>15</td>
<td>BIDIR</td>
</tr>
<tr>
<td>ctlldata15</td>
<td>14</td>
<td>BIDIR</td>
</tr>
</tbody>
</table>

END;

CONNECTED_PINS
BEGIN
  ctloeoutn : OUTPUT;
  ctloeinn  : INPUT;
CONNECTED_PINS
BEGIN
sdoeoutn : OUTPUT;
sdoeinn : INPUT;
END;

SUBDESIGN '13022-
( 
clk : INPUT; % System Clock IS 32 MHz %
resetn : INPUT; % Card Reset %
% Control Reads/Writes %
ctlrdnssync : INPUT; % Control Read Strobe %
ctlwrhnsync : INPUT; % Control Write High Strobe %
ctlwrlnsnc : INPUT; % Control Write Low Strobe %
ctladdr[5..0] : INPUT; % Control Address Bus %
ctldata[15..0] : BIDIR; % Control Data Bus %
ismcr0 : INPUT; % ISM Control Register Bit 0 %
bitcr0 : OUTPUT; % BIT Control Register Bit 0 %
ctlloeinn : INPUT; % Control Data Bus Output Enable %
ctlloeoutn : OUTPUT;
% GBC Data Interface %
sd[15..0] : OUTPUT; % GBC Data Bus %
sdoeinn : INPUT; % GBC Data Bus Output Enable %
sdoeoutn : OUTPUT;
% Input signals from 12847 %
fifo2efn7in, fifo2hfn7in : INPUT;
% Output signals to 12848 side %
fifo2efn8out, fifo2hfn8out : OUTPUT;
% Input signals from 12848 side %
fifo2rdn8in, fifo2oen8in : INPUT;
% Output signals to 12847 side %
fifo2rdn7out, fifo2oen7out : OUTPUT;
VARIABLE
  % Control data lines. %
ctldata_t[15..0] : TRI;

  % Control Write FSM %
writefsm : MACHINE WITH STATES
  (writeidle, writehilo, writelow, writehigh, writewait);

  % Control Read FSM %
readfsm : MACHINE WITH STATES
  (readidle, readdata, readwait);

  % BIT Registers %
bitcr[11..0] : DFF;  % BIT Control Register %
bitbpr[15..0] : DFF;  % BIT Base Pattern Register %

  % FSM for Burst Duty-Cycle Control %
dutyfsm: machine with states
  (one, two, three, four, five, six, seven, eight);
  enable : SRFF;

  % Counter for Duty-Cycle Control %
count[7..0] : DFF;
  count_done : SOFT;

  % Synchronize reset %
resetnsync[1..0] : DFF;

  % Channel 2 FIFO Status Flags %
fifo2efnsync[1] : DFF;
fifo2hfn8sync[1] : DFF;

  % GBC Data Bus %
sd[15..0] : TRI;
sd_q[15..0] : DFFE;

datafsm : MACHINE WITH STATES
  (bitstop, bitgo, bithigh, bitlow);

fifo2efn8out, fifo2hfn8out : DFF;
% Alternating Pattern Register%
alt: DFF;

% 16 Bit PRN Register%
prn[15..0]: DFFE;  % 16 Bit PRN Word%
prnena: SOFT;
prnload: DFF;
t[9..1]: SOFT;  % Temporary Variables to reduce%
d[8..2]: SOFT;  % logic complexity and logic cells%

BEGIN

DEFAULTS

% Control Output Enable%
ctloeoutn = VCC;

% Registers%
bitcr[].prn = VCC;
bitbpr[].prn = VCC;

% Duty-cycle FSM Signals%
count[7..0].clrn = VCC;
count[7..0].prn = VCC;
enable.clrn = VCC;
enable.prn = VCC;
enable.s = GND;
enable.r = GND;

% Synchronize Reset%
resetnsync[].clrn = VCC;
resetnsync[].prn = VCC;

% Channel 2 Synchronization%
fifo2efnsync[].clrn = VCC;
fifo2efnsync[].prn = VCC;
fifo2hfnsync[].clrn = VCC;
fifo2hfnsync[].prn = VCC;

fifo2efn8out.clrn = VCC;
fifo2efn8out.prn = VCC;
fifo2hfn8out.clrn = VCC;
fifo2hfn8out.prn = VCC;

% GBC Data BUS%
sd_q[].clrn = VCC;
sd_q[] .prn = VCC;
prn[].prn = VCC;
prn[].clrn = VCC;
alt.prn  = VCC;
alt.clrn  = VCC;

prnena = GND;
prnload.d = GND;
prnload.clrn = VCC;
prnload.prn  = VCC;

sdoeoutn = VCC;

END DEFAULTS;

% Synchronize reset %
resetnsync[].clk = GLOBAL (clk);
resetnsync[1].d = resetn;
resetnsync[0].d = resetnsync[1].q;

% Define the clocks, resets, and presets of the BIT registers %
bitcr[].clk = GLOBAL (clk);
bitcr[].clrn = resetnsync[0].q;

bitbpr[].clk = GLOBAL (clk);
bitbpr[].clrn = resetnsync[0].q;

% Enable if any control reads/writes are BIT Registers %
selbit = (ctladdr[5..1] == B"00101");

% Enable if any reads/writes are EPLD ID Registers %
selid = (ctladdr[5..0] == B"101110");

% Control Reads of Registers %
ctldata[15..0] = ctldata_t[15..0];
ctldata_t[15..0].oe = !ctloeinn;

readfsm.clk = GLOBAL(clk);
readfsm.reset = !resetnsync[0].q;

CASE readfsm IS
   WHEN readidle =>
      IF (!ctlrdnsync AND selbit) THEN
         CASE ctladdr[0] IS

167
WHEN BITCR_ADDR =>
    ctldata_t[15..0] = (H"0",bitcr[11..0]);
WHEN BITBPR_ADDR =>
    ctldata_t[15..0] = bitbpr[15..0];
END CASE;

ctoeoutn = GND;
readfsm = readdata;
ELSIF (!ctllrdnsync AND selid) THEN
    ctldata_t[15..0] = H"00";
    ctoeoutn = GND;
    readfsm = readdata;
ELSIF (!ctllrdnsync) THEN
    readfsm = readwait;
ELSE
    readfsm = readidle;
END IF;
WHEN readdata =>
    IF (ctllrdnsync) THEN
        readfsm = readidle;
    ELSE
        readfsm = readdata;
    END IF;

IF (selbit) THEN
    CASE ctladdr[0] IS
        WHEN BITCR_ADDR =>
            ctldata_t[15..0] = (H"0",bitcr[11..0]);
        WHEN BITBPR_ADDR =>
            ctldata_t[15..0] = bitbpr[15..0];
        END CASE;
    ELSIF (selid) THEN
        ctldata_t[15..0] = H"00";
    END IF;

ctoeoutn = ctllrdnsync;
WHEN readwait =>
    IF (ctllrdnsync) THEN
        readfsm = readidle;
    ELSE
        readfsm = readwait;
    END IF;
END CASE;

% Control Writes of Registers %

168
writefsm.clk = GLOBAL(clk);
writefsm.reset = !resetnsync[0].q;

CASE writefsm IS
  WHEN writeidle =>
    IF (!ctlwrlonsync AND !ctlwrhinsync AND selbit) THEN
      writefsm = writehilo;
    ELSIF (!ctlwrlonsync AND selbit) THEN
      writefsm = writelow;
    ELSIF (!ctlwrhinsync AND selbit) THEN
      writefsm = writehigh;
    ELSIF (!ctlwrlonsync OR !ctlwrhinsync) THEN
      writefsm = writewait;
    ELSE
      writefsm = writeidle;
    END IF;
  END CASE;

  WHEN writehilo =>
    CASE ctladdr[0] IS
      WHEN BITCR_ADDR =>
        bitcr[].d = ctldata[11..0];
        bitbpr[].d = bitbpr[].q;
      WHEN BITBPR_ADDR =>
        bitcr[].d = bitcr[].q;
        bitbpr[15..0].d = ctldata[15..0];
    END CASE;
  END CASE;

  WHEN writelow =>
    CASE ctladdr[0] IS
      WHEN BITCR_ADDR =>
        bitcr[].d = ctldata[11..0];
        bitbpr[7..0].d = bitbpr[7..0].q;
      WHEN BITBPR_ADDR =>
        bitcr[].d = bitcr[].q;
        bitbpr[7..0].d = ctldata[7..0];
    END CASE;

    bitbpr[15..8].d = bitbpr[15..8].q;
    writefsm = writewait;

  WHEN writehigh =>
    IF (ctladdr[0] == BITBPR_ADDR) THEN
      bitbpr[15..8].d = ctldata[15..8];
    END IF;

  ELSE
    writefsm = writeidle;
  END IF;

bitcr[].d = bitcr[].q;
bitbpr[].d = bitbpr[].q;

END CASE;
ELSE
  bitbpr[15..8].d = bitbpr[15..8].q;
END IF;

bitcr[].d = bitcr[].q;
bitbpr[7..0].d = bitbpr[7..0].q;
writefsm = writewait;
WHEN writewait =>
  IF (ctlwrloesync AND ctllwrhinsync) THEN
    writefsm = writeidle;
  ELSE
    writefsm = writewait;
  END IF;

bitcr[].d = bitcr[].q;
bitbpr[].d = bitbpr[].q;
END CASE;

% Duty-cycle Calculations %
count[7..0].clk = GLOBAL(clk);
count[7..0].clrn = resetnsync[0].q AND ismcr0 AND bitcr[0].q;
dutyfsm.clk = GLOBAL(clk);
dutyfsm.reset = !resetnsync[0].q OR !ismcr OR !bitcr[0].q;
able.clk = GLOBAL(clk);
able.s = !resetnsync[0].q OR ((dutyfsm == eight)
  AND count_done AND !count[0].q) OR !bitcr[0].q;

count.done = (count[7..1] == B"1111111");

CASE (dutyfsm) IS
  WHEN one =>
    IF ((bitcr[3..1] == B"000") AND count_done AND
        !count[0].q) THEN
      enable.r = VCC;
      count[] = count[] + 1;
    ELSIF (count_done AND count[0].q) THEN
      count[].d = GND;
      dutyfsm = two;
    ELSIF (fifo2efn8out) THEN
      count[] = count[] + 1;
    ELSE
      count[].d = GND;
    END IF;
  WHEN two =>
    IF ((bitcr[3..1] == B"001") AND count_done AND
!count[0].q) THEN
  enable.r = VCC;
  count[] = count[] + 1;
ELSIF (count_done AND count[0].q) THEN
  count[].d = GND;
  dutyfsm = three;
ELSE
  count[] = count[] + 1;
END IF;
WHEN three =>
  IF ((bitcr[3..1] == B"010") AND count_done AND
    !count[0].q) THEN
    enable.r = VCC;
    count[] = count[] + 1;
  ELSIF (count_done AND count[0].q) THEN
    count[].d = GND;
    dutyfsm = four;
  ELSE
    count[] = count[] + 1;
  END IF;
WHEN four =>
  IF ((bitcr[3..1] == B"011") AND count_done AND
    !count[0].q) THEN
    enable.r = VCC;
    count[] = count[] + 1;
  ELSIF (count_done AND count[0].q) THEN
    count[].d = GND;
    dutyfsm = five;
  ELSE
    count[] = count[] + 1;
  END IF;
WHEN five =>
  IF ((bitcr[3..1] == B"100") AND count_done AND
    !count[0].q) THEN
    enable.r = VCC;
    count[] = count[] + 1;
  ELSIF (count_done AND count[0].q) THEN
    count[].d = GND;
    dutyfsm = six;
  ELSE
    count[] = count[] + 1;
  END IF;
WHEN six =>
  IF ((bitcr[3..1] == B"101") AND count_done AND
    !count[0].q) THEN

171
enable.r = VCC;
count[] = count[] + 1;
ELSIF (count.done AND count[0].q) THEN
  count[].d = GND;
dutyfsm = seven;
ELSE
  count[] = count[] + 1;
END IF;
WHEN seven =>
  IF ((bitcr[3..1] == B"110") AND count.done AND
      !count[0].q) THEN
    enable.r = VCC;
cnt[] = count[] + 1;
  ELSIF (count.done AND count[0].q) THEN
    count[].d = GND;
dutyfsm = eight;
  ELSE
    count[] = count[] + 1;
  END IF;
WHEN eight =>
  IF (count.done AND !count[0].q) THEN
    count[] = count[] + 1;
  ELSIF (count.done AND count[0].q) THEN
    count[].d = GND;
dutyfsm = one;
  ELSE
    count[] = count[] + 1;
  END IF;
END CASE;

% GBC Data Bus %
sdoeoutn = !bitcr[0].q;
sd[].oe = !sdoeinn;
sd[] = sd_q[].q;

sd_q[].clk = GLOBAL(clk);
sd_q[].clrn = resetnsync[0].q;
sd_q[].ena = !((datafsm == bitlow) OR (datafsm == bitgo));

IF (bitcr[7].q) THEN
  sd_q[] = prn[].q;
ELSIF (alt.q) THEN
  sd_q[] = !bitbpr[].q;
ELSE
  sd_q[] = bitbpr[].q;

END IF;

% Channel 2 Status Flags %
fifo2efnsync[].clk = GLOBAL(clk);
fifo2efnsync[].clrn = fifo2efn7in AND resetnsync[0].q;

fifo2efnsync[1].d = fifo2efn7in;

fifo2hfn8out.clk = GLOBAL(clk);
fifo2hfn8out.clrn = resetnsync[0].q;

fifo2hfn8out[1].d = fifo2hfn7in;

% Data Interface%
fifo2efn8out.clk = GLOBAL(clk);
fifo2hfn8out.clk = GLOBAL(clk);

IF (bitcr[0].q) THEN
    fifo2efn8out.clrn = enable.q;
    fifo2efn8out.d = enable.q;
    fifo2hfn8out.d = GND;
ELSE
    fifo2efn8out.clrn = fifo2efn7in;
    fifo2efn8out.d = fifo2efnsync[1].q;
    fifo2hfn8out.d = fifo2hfn8out[1].q;
END IF;

IF (bitcr[0].q) THEN
    fifo2rdn7out = VCC;
    fifo2oen7out = VCC;
ELSE
    fifo2rdn7out = fifo2rdn8in;
    fifo2oen7out = fifo2oen8in;
END IF;

% Alternating Data Pattern %
alt.clk = GLOBAL(clk);
alt.clrn = resetnsync[0].q;

% 16-Bit PRN Generator %
prn[].clk = GLOBAL(clk);
prn[].ena = prnena;
prn[].clrn = resetnsync[0].q;

prnload.clk = GLOBAL(clk);
prnload.prn = resetnsync[0].q;

% 16 bit PRN algorithm

New bit 0 is found by XORing old bits: 4 8 10 11 12 13 15
New bit 1 is found by XORing old bits: 0 2 3 9 11 12 13 14
New bit 2 is found by XORing old bits: 1 3 4 10 12 13 14 15
New bit 3 is found by XORing old bits: 0 3 4 11 13 14 15

New bit 4 is found by XORing old bits: 0 1 2 3 4 12 14 15
New bit 5 is found by XORing old bits: 0 1 4 13 15
New bit 6 is found by XORing old bits: 0 1 3 14
New bit 7 is found by XORing old bits: 1 2 4 15

New bit 8 is found by delaying bit 0
New bit 9 is found by delaying bit 1
New bit 10 is found by delaying bit 2
New bit 11 is found by delaying bit 3

New bit 12 is found by delaying bit 4
New bit 13 is found by delaying bit 5
New bit 14 is found by delaying bit 6
New bit 15 is found by delaying bit 7

IF (prnload.q) THEN
  prn[] . d = bitbpr[] . q;
ELSE
  prn[15..8] = prn[7..0];
  prn[7] = t2  $ t8;
  prn[6] = d2;
  prn[5] = t2  $ t5  $ prn13;
  prn[4] = d2  $ d3;
  prn[3] = d4  $ d5;
  prn[2] = d6  $ d7;
  prn[1] = d4  $ d8;
  prn[0] = t2  $ t3  $ t4;
END IF;

% Temporary Variables %
t1 = prn3  $ prn14;
t2 = prn4  $ prn15;
t3 = prn11 $ prn13;
t4 = prn8  $ prn10 $ prn12;
t5 = prn0 $ prn1;
t6 = prn2 $ prn12;
t7 = prn1 $ prn10 $ prn13;
t8 = prn1 $ prn2;
t9 = prn0 $ prn9;

d2 = t1 $ t5;
d3 = t2 $ t6;
d4 = t1 $ t3;
d5 = prn[0] $ t2;
d6 = t1 $ t7;
d7 = t2 $ prn[12];
d8 = t6 $ t9;

% BIT FSM %
datafsm.clk = GLOBAL(clk);
datafsm.reset = !resetnsync[0].q OR !ismcr0 OR !bitcr[0].q;

CASE datafsm IS
  WHEN bitstop =>
    IF (fifo2rdn8in AND bitcr[0].q AND fifo2efn8out.q)
    THEN
      prnload.d = GND;
datafsm = bithigh;
    ELSE
      prnload.d = VCC;
      prnena = VCC;
datafsm = bitstop;
    END IF;

    alt.d = GND;
  WHEN bitgo =>
    alt.d = alt.q;

    IF (!bitcr[0].q) THEN
      datafsm = bitstop;
    ELSIF (fifo2rdn8in AND fifo2efn8out.q) THEN
      prnena = GND;
datafsm = bithigh;
    ELSE
      datafsm = bitgo;
    END IF;
  WHEN bithigh =>
    alt.d = alt.q;

IF (!fifo2rdn8in) THEN
    prnena = VCC;
    datafsm = bitlow;
ELSIF (fifo2efn8out.q) THEN
    datafsm = bithigh;
ELSE
    datafsm = bitgo;
END IF;
WHEN bitlow =>
    % Generate the Alternating Pattern %
    alt.d = !alt.q;
    IF (!bitcr[0].q) THEN
        datafsm = bitstop;
    ELSIF (fifo2rdn8in AND fifo2efn8out.q) THEN
        prnena = VCC;
        datafsm = bithigh;
    ELSE
        prnena = VCC;
        datafsm = bitgo;
    END IF;
END CASE;
END;
Appendix D

Timing Diagrams
Figure D-1: Control Module EPLD (13021) Timing Diagram
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trdnsync</td>
<td>Control read strobe, CTRDNN, synchronization delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Twrnsync</td>
<td>Control write strobes, CTLWRON and CTLWRHIN, synchronization delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tctldata</td>
<td>CTLDATA bus data tri-state to data valid propagation delay</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Trdfsms</td>
<td>Control read FSM propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Tctloen</td>
<td>CTLDATA bus output enable propagation delay</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>Tctldoen</td>
<td>CTLDATA bus data valid to data tri-state propagation delay</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Tackn</td>
<td>Control acknowledge signal, ACKN, delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tdataready</td>
<td>Control interface ACKN assertion timing constraint</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Twrftsm</td>
<td>Control write FSM propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Treg</td>
<td>Control register propagation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tdh</td>
<td>Control interface data bus hold time</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Taddrlen</td>
<td>74FCT573A ADDRLEN (address latch enable) delay</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>T573</td>
<td>74FCT573A transparent latch data disable delay</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Trdnoen</td>
<td>Control data bus output enable delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Tdackn</td>
<td>Control interface acknowledge, ACKN, disable delay</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Tack_off</td>
<td>Control interface acknowledge, ACKN, disable constraint</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Tdz</td>
<td>Control interface data bus data valid to tri-state constraining</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

Table D.1: Control Module EPLD (13201) Timing Diagram Parameters
Figure D-2: BIT EPLD (13022) Control Register Timing Diagram
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tctlrdn</td>
<td>Control read strobe, CTLRDN, synchronization delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tclwrm</td>
<td>Control write strobes, CTLWRLON and CTLWRHIN, synchronization delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tctldta</td>
<td>CTLDATA bus data tri-state to data valid propagation delay</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Tpdfsm</td>
<td>Control read FSM propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Tctloen</td>
<td>CTLDATA bus output enable propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>T573</td>
<td>Delay until address invalid at 74FCT573A transparent latch</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Tctldon</td>
<td>CTLDATA bus data valid to data tri-state propagation delay</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Tackn</td>
<td>Control acknowledge signal, ACKN, delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tdatarea</td>
<td>Control interface ACKN assertion timing constraint</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>Twrftsm</td>
<td>Control write FSM propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Treg</td>
<td>Control register propagation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tdh</td>
<td>Control interface data bus hold time</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table D.2: Data Module BIT EPLD (13022) Control Register Timing Diagram Parameters
Figure D-3: Receiver Module EPLD (12847) Timing Diagram
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trdfsm</td>
<td>Control read FSM propagation delay</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Twrfsm</td>
<td>Control write FSM propagation delay</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Toe</td>
<td>CTLOEOUTN signal propagation delay</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Tack</td>
<td>Channel ACKN signal propagation delay</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Tdoe</td>
<td>CTLOEOUTN disable propagation delay</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Tdack</td>
<td>Channel ACKN disable propagation delay</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>Treg</td>
<td>Register clock-to-Q propagation delay</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Ttd</td>
<td>TAXI output data bus propagation delay</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Tstrb</td>
<td>TAXI strobe propagation delay</td>
<td>16</td>
<td>28</td>
</tr>
<tr>
<td>Tcycle</td>
<td>FCYCLE propagation delay</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Ttclkdel</td>
<td>TCLKDEL propagation delay</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Tdf</td>
<td>FIFOWRTN signal disable propagation delay</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Tdd</td>
<td>TCLKDEL disable propagation delay</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Tfw</td>
<td>FIFOWRTN signal propagation delay</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Tfds</td>
<td>Channel FIFO buffer memory setup time</td>
<td>11</td>
<td>-</td>
</tr>
<tr>
<td>Tfd</td>
<td>FIFO input data propagation delay</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Ttds</td>
<td>Guaranteed TAXI strobe to data delay</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>Tctldata</td>
<td>CTLDATA bus valid data delay</td>
<td>0</td>
<td>19</td>
</tr>
</tbody>
</table>

Table D.3: Receiver Module EPLD (12847) Timing Diagram Parameters
Figure D-4: Arbiter EPLD (12848) Timing Diagram
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tfsm</td>
<td>Arbiter FSM propagation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tf1rdn</td>
<td>Channel 1 FIFO buffer memory read strobe delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tsa</td>
<td>Sample address bus propagation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tf1efn</td>
<td>Channel 1 FIFO empty flag delay</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Tcar1</td>
<td>Channel 1 address generation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tgfwrtn</td>
<td>GBCFIFOWRTN write strobe propagation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tf1oen</td>
<td>Channel 1 FIFO output enable delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>T574</td>
<td>Clock-to-Q delay of a 74FCT574A register</td>
<td>0</td>
<td>6.5</td>
</tr>
<tr>
<td>Tgsu</td>
<td>GBCFIFOWRTN data setup time</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Tghd</td>
<td>GBCFIFOWRTN data hold time</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>T574oe</td>
<td>Output disable delay of a 74FCT574A register</td>
<td>0</td>
<td>5.5</td>
</tr>
<tr>
<td>Tf2efn</td>
<td>Synchronized Channel 2 FIFO empty flag delay from 13022</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tf2rdn</td>
<td>Channel 2 FIFO buffer memory read strobe delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tf2oen</td>
<td>Channel 2 FIFO output enable delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tcar2</td>
<td>Channel 2 address generation delay</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tfsmwu</td>
<td>Arbiter FSM setup time for Channel 1 status flags</td>
<td>24</td>
<td>-</td>
</tr>
<tr>
<td>Tfsmwu</td>
<td>Arbiter FSM setup time for channel FIFO buffer memory status flags</td>
<td>24</td>
<td>-</td>
</tr>
</tbody>
</table>

Table D.4: Data Module Arbiter EPLD (12848) Timing Diagram Parameters
Figure D-5: BIT EPLD (13022) Operation Timing Diagram
<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tf2efn</td>
<td>Channel 2 FIFO empty flag generation delay in 13022</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Tf2rdn</td>
<td>Channel 2 FIFO buffer memory read strobe generation delay in 12848</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tgfwrtn</td>
<td>GBCFIFOWRTN write strobe generation delay in 12848</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Ts</td>
<td>Sample address bus propagation delay in 12848</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tdatafsm</td>
<td>BIT FSM, DATAFSM, propagation delay in 13022</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Tal</td>
<td>Alternating pattern generation delay in 13022</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Ts</td>
<td>Sample BIT data bus propagation delay in 13022</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Tprn</td>
<td>PRN pattern generation delay in 13022</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Tgfmsu</td>
<td>GBCFIFOWRTN data setup time constraint</td>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>Tgfhd</td>
<td>GBCFIFOWRTN data hold time constraint</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>Tdfsmsu</td>
<td>BIT FSM, DATAFSM, setup time constraint</td>
<td>8</td>
<td>-</td>
</tr>
</tbody>
</table>

Table D.5: Data Module BIT EPLD (13022) Operation Timing Diagram Parameters
Bibliography


