THE DESIGN OF A HIGH PERFORMANCE SPARC BUS INTERFACE

by

Diana Fung-Yee Wong

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

February 1994

Author _____

Department of Electrical Engineering and Computer Science

February 4, 1994

Certified by _____

Dr. William J. Dally

Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor

Accepted by _____

Dr. Frederic R. Morgenthaler

Chairman, Departmental Committee on Graduate Students

THE DESIGN OF A HIGH PERFORMANCE SPARC BUS INTERFACE

by

Diana Fung-Yee Wong

Submitted to the

Department of Electrical Engineering and Computer Science

February 4, 1994

In Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the advance in VLSI and packaging technologies, the power of microprocessors has been increasing at a stunning rate and the computing power of a workstation is now edging close to that of a mainframe computer. While workstations are already commonly connected by networks, the computational power of these workstations can be greatly increased if they can be configured as an integrated system for distributed computation. Unfortunately, the bandwidth, latency, reliability and cost of current network technologies are inadequate to allow such distributed network organization to be of widespread use.

In this thesis, we designed and simulated a low cost SPARC Bus (SBus) interface board which connects workstations into a high performance tightly-coupled network. The interface provides high bandwidth communication channels between workstations and routers supporting high speed routing for both multiprocessor and multicomputer systems.

Thesis Supervisor: Dr. William J. Dally

Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgment

I would like to thank Bill Dally for his supervision and support in this thesis. Special thanks is due to Larry Dennison for being so helpful and patient. His guidance and encouragement have made this thesis possible. There were so many times that when I hit the brick walls he was there to rescue me.

To Kinhong Kan, Duke Xanthopoulos, Ed Ouellette, Andre Dehon, Andrew Chang and the whole 6th floor crew who have made the AI Lab a fun and friendly working environment, I would like to express my appreciation. I am also grateful to Nate Osgood, Kathy Knobe and John Keen for so kindly reading my indecipherable draft and providing me with invaluable comments. My thanks go to Ronson for his companionship throughout my undergraduate years; to Cliff for teaching me how to think and how to care. I thank him for everything he has done for me.

I give my deepest gratitude to my family: Dad, Mom, Mary, Winnie and Shirley for their unfailing support and encouragement throughout the years. It is them who teach me how to become a real human being.

Finally, I praise the Lord for His mercy and grace. Without His love, I certainly will not be here today.

"I lift up my eyes to the hill —
where does my help come from?
My help comes from the Lord,
the Maker of heaven and earth."

Psalm 121:1

To Dad, Mom,
Mary, Winnie and Shirley

Contents

List of Figures

Chapter 1

Introduction

As VLSI and packaging technology improve, the computing power delivered by a single processor has been increasing rapidly. The current generation of workstations can deliver more power than mainframe computers a decade ago and in a more cost-effective manner. The productivity of these workstations could be increased even further by harnessing the aggregate power of multiple workstations connected in a network, thereby enabling researchers to develop new ways of using groups of these low cost and high performance workstations to deliver unprecedented computing power. However, current network technologies cannot offer such options owing to the following limitations:

1. **Bandwidth.** The most common local area network technology is implemented by Ethernet. While Ethernet can deliver a maximum bandwidth of 10 Mbits/sec, the actual throughput is typically between 2 to 3 Mbits/sec [6]. Such a low bandwidth makes communication extremely expensive and seriously limits the capabilities of networked workstations.

2. **Latency.** Many interactive or real time applications require low latency. The high overhead of packet-switching protocols, such as the circuit set-up and tear-down time required by ATM, preclude their use in environments where communication latency is critical.

3. **Cost.** There are numerous high speed network architectures being proposed and developed. Unfortunately, most of these architectures (for instance, FDDI

and ATM) have exceedingly high costs that prevent their widespread use in low budget situations.

This thesis provides a possible solution to the above stated problems by presenting the design of an interface board which connects a workstation to a router to form a low latency and high bandwidth network. The prototype board is a SPARC Bus (SBus) board connected to a SPARCStation. The use of SBus is motivated by the SBus' status as a popular industrial standard which provides low cost, compact form factor and high bandwidth transfer between the mother board and the card. The SBus system design also permits the use of a large installed base of SPARCStations at the AI Lab as a testbed.

To provide a clear picture of the functionality of the SBus board, let us consider the following scenario:

> Suppose there is a network of SPARCStations as shown in Figure 1.1. Each of the SPARCStations is connected to an NxN mesh of routers through its own SBus board. If workstation A (host) wants to send a message to workstation B (destination) on the network, it writes to its SBus board which relays the message to the router via fiberoptic cable. The router delivers the message through the mesh network to the SBus board on B which then forwards the message to the kernel.

While the primary design objective of this thesis is to solve the above mentioned limitations of current network technologies, we also need to keep the following design issues in mind:

1. **Transparency and Modularity.** The board is intended to be an interface connecting workstations to networks and it is important that this interface be well-defined and transparent (i.e. has a small number of handshake signals).

2. **Small Form Factor.** All the components must fit on a single-width SBus expansion board (146.70mm x 83.82mm) given by the SBus specification. Thus, the parts count should be minimized to make the design more compact.

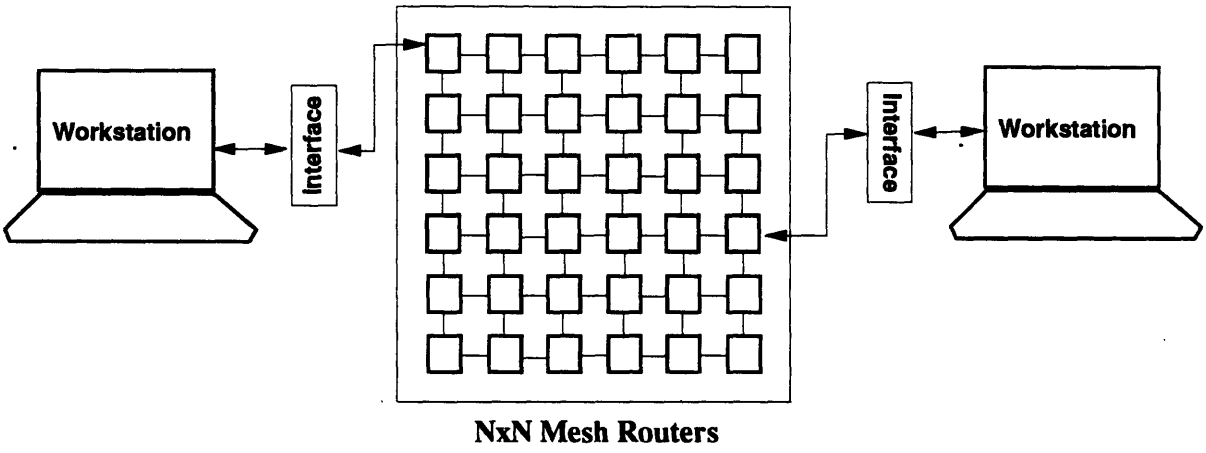

Figure 1.1: Interface Board Connecting Workstations to Routers.

3. **Ease of Implementation.** Due to constraints of time and effort, ease of implementation is clearly an important consideration. While we may choose to cut corners to increase the throughput or to lower the cost, it is important to ensure that the design remains robust. The design must effectively trade off constraints of speed, size, cost, ease of implementation and maintenance.

The thesis is organized into 7 chapters. Chapter 2 presents an overview of the SBus interface board and describes features of the system. Chapter 3 describes the various modules on the board while Chapter 4 turns to discuss the operation of the board and provides a detailed example of message flow. Chapter 5 describes the simulation used to verify the design. Chapter 6 presents the performance evaluation and the various design options of the work. Finally, Chapter 7 presents the conclusions.

Chapter 2

Design Overview

The SBus interface board is designed for a host-based system with a 32-bit I/O interface between the SBus system on a SPARCStation and a remote end connected to networks. A typical SBus system consists of three participants: an SBus *master* that initiates the transfer request, a *slave* that performs the operation and an SBus *controller* that oversees the whole SBus transaction. The SBus board in this thesis is implemented as an SBus slave that responds with an acknowledgment to *Address Strobe* and *Slave Select* asserted by the controller. However, the slave is not capable of initiating an SBus transaction, except indirectly by generating an Interrupt Request. The words SBus interface board, SBus board and SBus slave will be used interchangeably in this thesis.

A detail specification of the SBus board is provided in Appendix A and the overall design is as follows:

- **Clocking:**

  The SBus board operates at 25 MHz. This clock frequency provides a good balance between high performance and ease of system design and integration. It also satisfies the restriction imposed by the SBus clock which specifies the board to run between 16.67 MHz and 25 MHz. The FDDI ENcoder DECoder (ENDEC), due to its FDDI property however, performs at a frequency of 12.5 MHz. All of the modules on the SBus board are synchronized to the Nibble Clock (NCLKOUT) of the ENDEC.

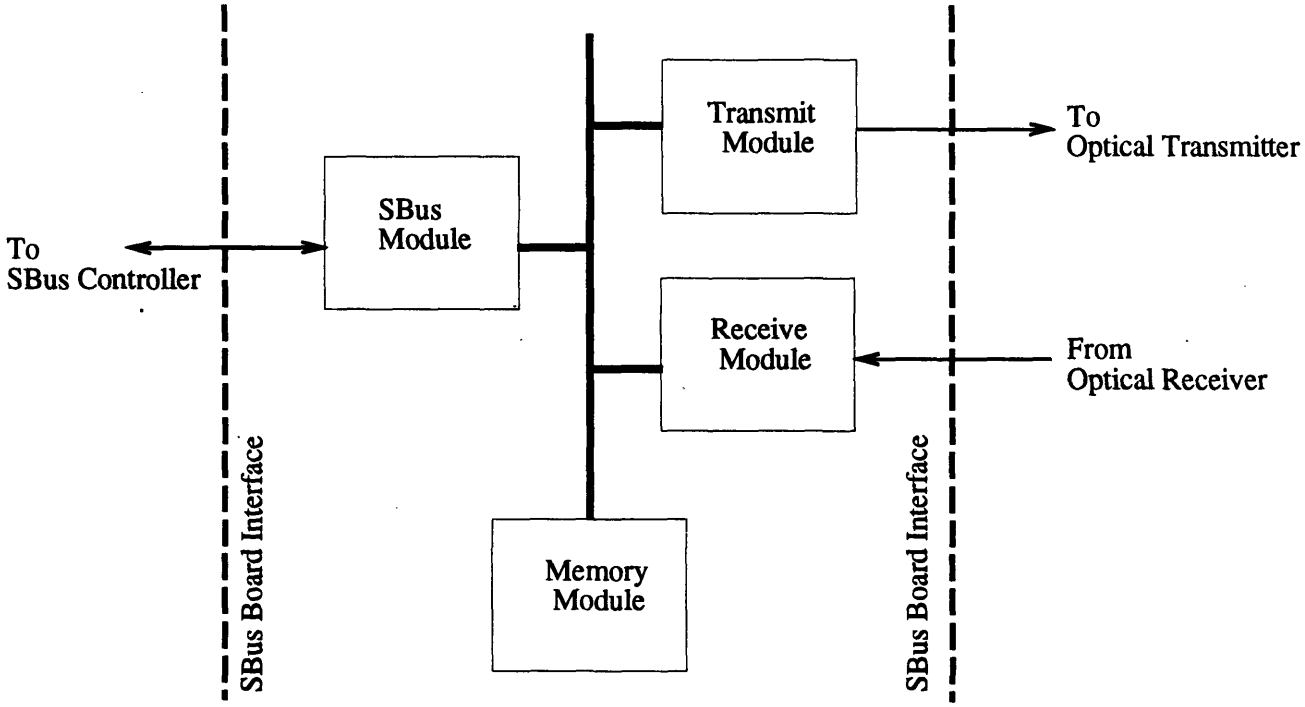

Figure 2.1: Block Diagram for the SBus Board Design.

- **Logic Partitioning:**

  The SBus board consists of the following major logic blocks (refer to Figure 2.1):

  1. Memory Module. The Memory module provides storage for messages delivered among the SBus module, the Transmit module, and the Receive module.
  2. SBus Module. The SBus module serves as an interface between the SBus controller on the SPARCStation and the SBus board. It provides handshake signals to initiate transfer of data between the CPU master on the host machine and the SBus board.
  3. Transmit Module. The Transmit module (Tx) is responsible for sending messages and relinquishing buffers that have been transmitted. It also

generates a header word and a checksum for each message, and sends flow-control information to the Receive module at the remote end.

4. Receive Module. The Receive module (Rx) is the dual of the Transmit module. It is responsible for requesting buffers to handle incoming packets, decoding the header packets, verifying the checksum, and relaying incoming flow-control signals from the remote end to the Transmit module.

- **TDMA plan:**

  The SBus board is operated under the Time Division Multiplex Access (TDMA) plan. Under this protocol, access to the SRAM is time-multiplexed among the three logic blocks, namely the SBus module, the Tx and the Rx.

  The SRAM cycles are divided into the following cycles, during when a specific logic block has exclusive access to the SRAM :

  - Receive Cycle (Rx_cycle)
  - SBus Write Cycle (SWr_cycle)
  - Transmit Cycle (Tx_cycle)
  - SBus Read Cycle (SRd_cycle)

  The 12.5 MHz ENDEC constitutes a bottleneck of the system. Since 8 clock cycles are required to transmit and to receive a 32-bit word to and from the ENDEC, each logic block can only effectively access the SRAM once every 8 clock cycles. There are 4 SRAM cycles and thus, the optimal partitioning required for each SRAM cycle is 80 ns. The cycles on the board are orchestrated by three different Finite State Machines (FSMs), namely the Transmit FSM (Txfsm), the Receive FSM (Rxfsm) and the SBus FSM (SBusfsm). These three FSMs are coordinated by a Next FSM (Nxtfsm) which enforces the TDMA plan. The simplicity of the interfaces among the FSMs makes the SBus board easy to implement and debug.

Most of the FSM controls and interface logic are implemented in an Altera Erasable Programmable Logic Device (EPLD) for design customization and cost minimization.

While the design can be implemented using existing hardware components, such an approach will cause the board size to exceed the SBus form factor specification. Although one may resort to a double-sided card using surface mount technology, this would increase both the cost and the complexity of the design. Instead, we chose the Altera EPLD because it provides high-density logic integration, greater cost-effectiveness, shorter development cycles, a smaller board area requirement and better ease of modification.

Chapter 3

Board Modules

This chapter presents the details of the logic blocks described in the previous chapter. The functionalities of the Transmit module (Tx), the Receive module (Rx), the SBus module and the Memory module are discussed. Furthermore, the submodules inside each logic block are examined. The block diagrams and signal definitions of the modules are included in Appendix A.

3.1 Modules

The SBus interface board consists of the following four modules:

3.1.1 Memory Module

The SRAM memory is accessed by three logic blocks (SBus module, Tx and Rx) under the TDMA plan. It is a 4 megabit (128Kx32) memory cell and is divided into 512 1KByte buffers where each buffer can hold a maximum of 256 32-bit words. The 17-bit SRAM address is composed of a buffer pointer (9 bits) and an offset (8 bits) into the buffer.

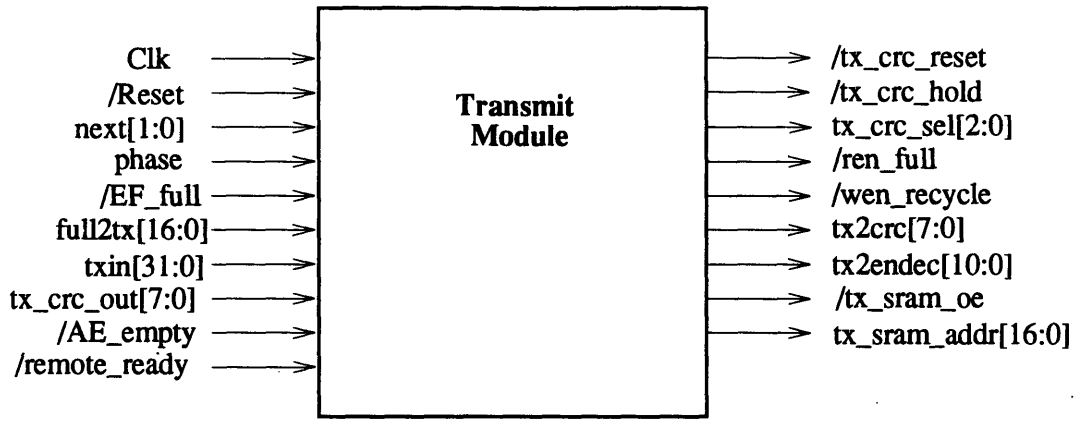

Figure 3.1: Transmit Module Signaling

3.1.2 Transmit Module

The Tx is responsible for obtaining an address pointer to a full SRAM buffer and sending messages to the remote end. It reads a 32-bit word from the SRAM during the Tx_cycle, and transmits the word to the Tx_ENDEC during the subsequent 8 clock cycles. Upon completing the transmission of the message, it puts the buffer pointer into a recycle FIFO and asserts an Interrupt signal to the SBus controller, which then returns the buffer pointer to the free-pointer queue in the kernel.

The Tx is composed of the followings:

- Multiplexor for selecting whether data, control bytes, or checksum is sent across the link.
- Parity generator for outputting the odd-parity bit to the Tx_ENDEC.
- Registers for synchronizing the inputs and outputs to and from the Tx.
- Tx_ENDEC for encoding 4B/5B code.

3.1.3 Receive Module

The Rx receives messages from the remote end and writes them to the SRAM during the Rx_cycle. It consists of a 2-deep 32-bit-wide FIFO which allows data to be written

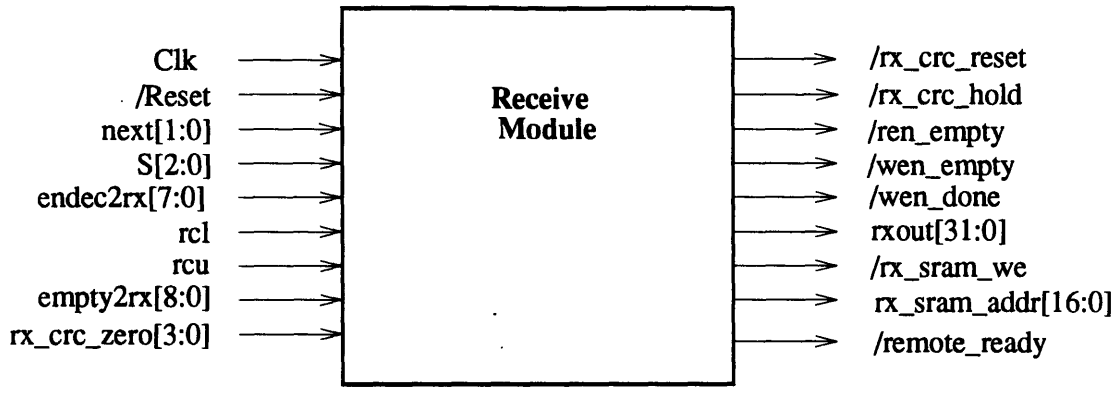

Figure 3.2: Receive Module Signaling

to the SRAM and to be read from the Rx_ENDEC simultaneously.

Upon completion of message reception, if there is no error in the received message, an Interrupt Request is generated to the SBus controller for transferring the message to the host machine. If an error is encountered during the receiving process, the Rx simply relinquishes the buffer by putting the buffer pointer back to an empty FIFO for later use.

The Rx is composed of the followings:

- FIFO consisting of two registers:
  - Register A for storing data from Rx_ENDEC.
  - Register B for outputting data to SRAM.
- Rx_ENDEC performing 4B/5B decoding of received data.
- Rx_EDS (ENDEC Data Separator) for recovering clock and data from the FDDI bit stream.

3.1.4 SBus Module

The SBus module consists of the SBusWrite submodule and the SBusRead submodule. The SBusWrite submodule is responsible for writing messages from the controller

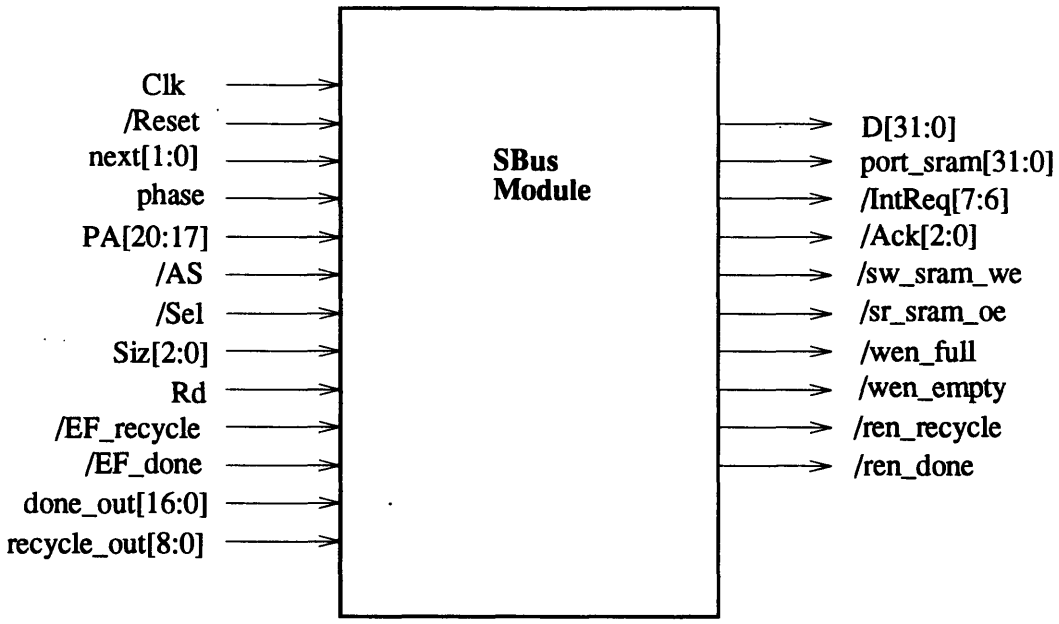

Figure 3.3: SBus Module Signaling

to the SRAM and the SBusRead submodule is responsible for reading messages from the SRAM to the controller. Since the traffic between the SBus controller and the slave SBus module is bidirectional, the datapath between the two employs tristate logic to avoid contention.

The SBus module is composed of the followings:

- sram2sbus tristate buffer for driving the SRAM data onto the SBus.
- sram2sbus register for synchronizing the bus input from the SRAM.
- sbus2sram tristate buffer for driving the SBus data to the SRAM.
- sbus2sram register for synchronizing the bus input from the SBus.

3.2 Submodules

Within each module just described, there are submodules that complement the functions of the logic blocks.

3.2.1 Bus Exchange Unit (Bux)

The Bus Exchange Unit is a multiple bus exchange device that time-multiplexes the SRAM datapath among the SBus module, the Tx and the Rx. The unit is organized as two I/O ports, one Input port and one Output port, where each port can be used as either a source or destination under independent control to implement a digital cross-point switch. This device facilitates the communication among the four ports: data can be routed from the SRAM to the Tx, from the Rx to the SRAM and between the SBus and the SRAM. Each port has a 16-bit databus. To form a 32-bit interface, two Buxes are cascaded together. This module is implemented in an Altera EPLD.

3.2.2 ENDEC

In both the Tx and the Rx, an ENDEC is used to perform 4B/5B encoding and decoding of data to and from the FDDI link. The Tx_ENDEC enables line states to be forced onto the link for control purposes and the Rx_ENDEC decodes line-state information from data that has been received from the fiberoptic link. The data frame is transmitted and is received in the form of 8-bit bytes accompanied by two control characters and one odd-parity bit. The ENDEC, which operates at 12.5 MHz, transmits or receives a byte of data every other clock, thus delivering a transfer rate of 100 Mbits/sec.

3.2.3 EDS

The ENDEC Data Separator recovers clock and data from an FDDI bit stream. It extracts the receive bit clock from the serial frames on the fiberoptic line. The EDS also provides timing information for the Rx_ENDEC to perform 4B/5B decoding before sending data to the Rx.

3.2.4 Cyclic Redundancy Checks (CRC)

The checksum mechanism is performed by 4 block check registers (BCRs) using the CRC-12 cyclic code [14]. The CRC-12 code provides error detection of all burst errors

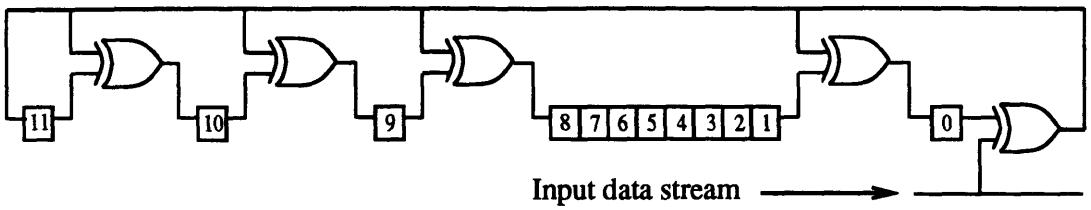

Figure 3.4: CRC-12: Check Polynomial $G(X) = X^{12} + X^{11} + X^3 + X^2 + X + 1$

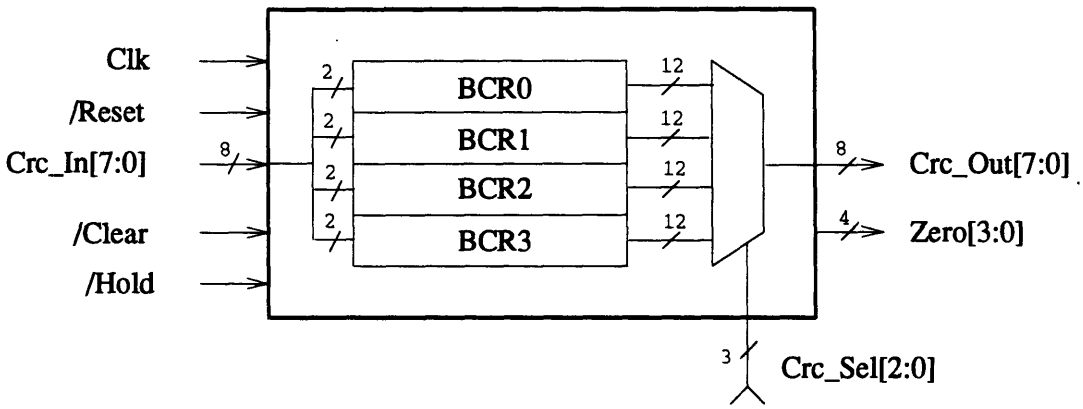

Figure 3.5: CRC Logic Block

of length 12 or less. The data length can be arbitrary. Thus, redundancy and coverage probability change with the data length.

Each BCR is implemented by a serial shift XOR circuit (Figure 3.4). Since a byte of data is transmitted every 2 clock cycles, 8 bits of data need to be serially shifted through the BCR circuit in 2 clocks. However, to avoid using a higher frequency clock, 4 BCR circuits are used in parallel, where each BCR circuit performs a checksum on 2 bits of data (as shown in Figure 3.5).

In operation, each Tx_BCR (Transmit BCR) is preloaded with an zero initial value. The data is simultaneously transmitted and fed to the input of the Tx_BCRs. Once the last data bit has been transmitted, the Tx_BCRs contain the check bits of the code word which are then transmitted following the data. Identical BCRs are used at the receiving end (also initialized with the same value as that used in the Tx_BCRs).

The output of the Rx_ENDEC is fed to the Rx_BCRs' (Receive BCRs') input. Similar to the Tx_BCRs, the received check bits are fed into the Rx_BCRs following the data bits. When there is no error in the message, the result in the receiver is zero.

Notice that checksum is performed only when data is transmitted from the SBus interface board to the remote end and when data is received from the remote end to the board. The link between the SBus controller and the SBus slave is assumed to be reliable and no checksum is performed.

This module is implemented in an Altera EPLD.

3.2.5 FIFOs

The memory is partitioned into 512 1KByte buffers. Upon initialization, half of the buffers are allocated for message transmission and half of the buffers are allocated for message reception.

The status of the buffers is monitored by both the kernel and the SBus slave. The SBus slave keeps track of the SRAM buffer status by means of the four FIFOs on board. Each buffer pointer can be in one of the following five states:

- Tx_Empty: ready to be written by kernel.
- Tx_Full: ready to be transmitted to the remote end.
- Rx_Empty: ready to receive a message from the remote end.
- Tx_Recycle: ready to be reclaimed by the kernel.
- Rx_Done: ready to be transmitted to the kernel.

1. Full FIFO

   The Full FIFO contains the pointers and lengths of allocated SRAM buffers whose contents are ready for transmission. The SBus controller writes the buffer information into the FIFO after transferring data to the SRAM. The Tx then reads the pointers and transmits data to the remote end.

2. Empty FIFO

   The Empty FIFO contains pointers to free buffers which are ready for reception of data from the remote end. When a message arrives, the Rx retrieves a free pointer from the Empty FIFO to store the message. The buffer pointer is returned to the Empty FIFO when:

   (a) the SBus controller finishes reading the received message.

   (b) the received message contains errors. (In this case, the incoming message is discarded.)

3. Done FIFO

   The Done FIFO contains pointers to buffers which are ready to be transmitted from the SBus slave to the host machine. The Rx writes to the FIFO after receiving an error-free message. The FIFO is then read by the SBus controller to transfer data to the host machine.

4. Recycle FIFO

   After the Tx finishes sending a message, it writes the transmitted buffer pointer to the Recycle FIFO. The Recycle FIFO is read by the SBus controller to reclaim the buffer for another data transmission.

Chapter 4

Board Operation

This chapter describes the operation of the logic blocks and details the means by which a message is sent to and received from the SBus interface board.

4.1 TDMA plan

Under the Time Division Multiplex Access (TDMA) plan, the SRAM is accessed by the Tx, the Rx and the SBus module during Tx_cycle, Rx_cycle, SRd_cycle and SWr_cycle (Figure 4.1). These four cycles are automated by a Next FSM performing SRAM cycle partition and databus/addressbus arbitration. The Next FSM selects which module drives the SRAM address and datapath, asserts (or deasserts) */WE* and */OE* SRAM signals accordingly, and enables tristates and registers at the module interface to guarantee that there is only one module driving the bus at any particular time (Figure 4.2).

4.2 The SBus Operation

The SBus slave in the design connects the host machine to the remote end. There are 8 handshake signals associated with data transfer between the SBus controller and the slave:

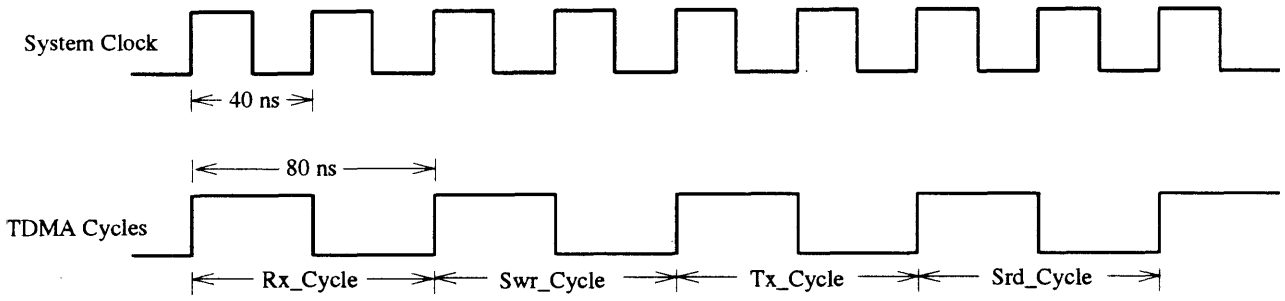

Figure 4.1: SRAM TDMA Cycles

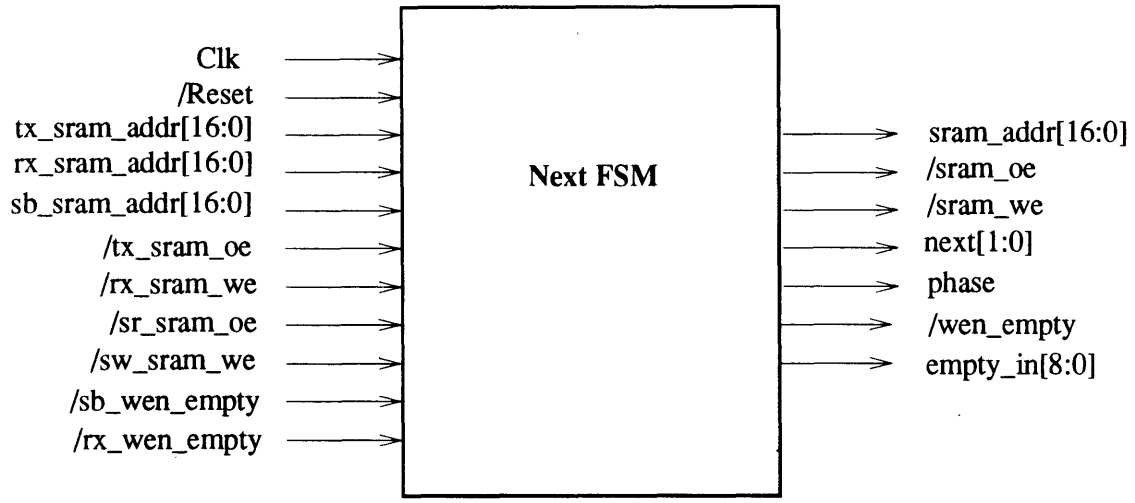

Figure 4.2: Next FSM Signaling

- */AS* (Address Strobe) : used by the SBus controller to initiate a slave cycle.
- */Sel* (Slave Select) : The SBus is a geographically-addressed bus. Each SBus slave receives an unique unary encoded */Sel* address signal. The assertion of this signal indicates that the given slave is addressed.
- *Rd* (Read) : signals whether the SBus controller will read data from the selected slave (*Rd* asserted) or write data to the slave (*Rd* unasserted).
- *PA[27:0]* (Physical Address) : *PA[16:0]* is used to address the SRAM. *PA[20:17]* are used as flags to signal special conditions.
- *Siz[2:0]* (Size) : used to indicate the number of bytes to be transferred to and from the SBus controller.
- *D[31:0]* (Data) : transfer data and buffer pointers.
- */Ack[2:0]* (Transfer Acknowledge) : indicates the types of data that have been transferred (Data Ack), or that the current slave cycle should be terminated (Error Ack), or both.
- */IntReq[7:1]* (Interrupt Request) : used by the SBus slave to request service from the host machine.

4.2.1 The Slave Cycle

The slave cycle occurs when there is data transfer between the host machine and the slave. It ends only after the slave acknowledges the last word of data, or issues an Error Ack in response to a transfer size that it is unable to support. The slave cycle can be either one of the following two types:

1. SBusWrite Cycle

   When writing data to a slave, the SBus controller drives the first datum onto the bus in the clock cycle when */AS* is asserted. At a later time (subject to timeout), the slave generates a Data Ack for one clock cycle.

2. SBusRead Cycle

   When reading data from a slave, the slave can generate a Data Ack at any time (subject to timeout), beginning with the clock cycle following the assertion of */AS*. The data corresponding to the acknowledgment is driven onto the databus for exactly one clock cycle during the clock cycle immediately following the Data Ack.

4.2.2 Service for Interrupt Request

Interrupt Request (*/IntReq[7:1]*) provides a mechanism for the SBus slave to interrupt the CPU. When the not-empty flag of the Done FIFO is asserted, the SBus slave asserts one of the Interrupt lines to asynchronously signal the CPU to read from the slave. After an Interrupt has been serviced, the SBus controller clears the asserted */IntReq* line. The SBus slave, however, cannot deassert an Interrupt until polled by the controller. The SBus has a total of seven shared Interrupt lines which a slave can assert at any time.

4.3 Message

Each message consists of a start delimiter, data, an end delimiter and a checksum codeword. The kernel is responsible for assembling the header information (such as destination and message length) into the data frame before handing the message to the SBus. The SBus board is responsible for providing a start delimiter and appending an end delimiter and a checksum at the end of each message frame. The message frame format is composed of the followings:

1. Idle bytes (16 or more bytes).
2. Start delimiter (4 control bytes).
3. Data (1 to 256 32-bits-word).
4. End delimiter (2 control bytes).

5. Checksum (6 bytes of CRC checksum)

4.3.1 Sending and Receiving a Message

The transmission of a message from a host begins with the assertion of Address Strobe (*/AS*) and Slave Select (*/Sel*) by the SBus controller. The buffer pointer to the SRAM is provided by the kernel of the host machine on the Physical Address (*PA*) lines. SBus starts writing the message to the SRAM. After the entire message is written, the buffer pointer and the message length are added to the Full FIFO. During the Tx_cycle, the Txfsm reads the pointer and message length from the Full FIFO and starts message transmission. Upon completion, the Txfsm writes the buffer pointer to the Recycle FIFO and asserts an Interrupt Request signal. When the SBus controller services the Interrupt Request, it returns the buffer pointer to the kernel for later use.

The reception of a message begins with the arrival of the start delimiter of a message frame. A free buffer pointer is retrieved from the Empty FIFO to store the received message. Upon receiving the end of a message, if there is no error in the message, the buffer pointer is written to the Done FIFO and an Interrupt Request is signaled. The SBus controller services the Interrupt by reading the received message from SRAM to host and freeing the buffer by returning it to the Empty FIFO.

We now turn to look at how a particular message flows among the different logic blocks on the SBus board.

1. **Message Flow from the SBus Module to the Transmit Module**

   The Txfsm controls the message flow from the SBus module to the Tx. When the Tx has no message to send, the Tx_ENDEC outputs idle bytes over the fiberoptic link to the remote end to indicate no valid data. At least 16 idle bytes are transmitted before any valid data is sent in order to synchronize the source Tx_ENDEC and the destination Rx_ENDEC. After completing the idle transmission, if the Full FIFO is not empty, the Txfsm sends a start delimiter followed by data and checksum. If the Full FIFO is empty, the Txfsm will return to the idle state and wait for the next message arrival.

2. **Message Flow from the Receive Module to the SBus Module**

   The Rxfsm controls the message flow from the Rx to the SBus Module. The Rxfsm spins in the idle state until the start delimiter of a message is received. Data bytes are first put into Register A until a word is composed which will then be shifted into Register B. Register A is ready for receiving the next word and Register B is ready to be read by the SRAM. This process continues until the end of message is indicated. If there is no error in the message, the received buffer will be sent to the SBus module. If the received message contains an error, the buffer will be ignored and the buffer pointer will be reused for further message reception.

3. **Message Flow between the SBus Module and the SBus Controller**

   The SBusfsm controls the message flow from the SBus module to the SBus controller. The assertion of Address Strobe (*/AS*) and Slave Select (*/Sel*) by the controller indicates the beginning of a slave cycle when data is transferred between the SBus slave and the host machine. The physical address *PA[27:0]*, *Rd*, *Siz[2:0]* and *D[31:0]* (if performing a write) at this time are driven onto the bus. During an SBusWrite, a free buffer pointer is read from the kernel. Data is transferred from the controller to the slave and is written to the SRAM during SWR_cycle.

   An SBusRead is performed in response to an Interrupt requested by the slave. Such Interrupts can serve for either reading data from the slave during the SRd_cycle or for returning transmitted buffer pointers back to the kernel. A Data Ack from the slave terminates the slave cycle.

4.4 Flow-Control

To prevent message overflow, a simple flow-control mechanism applies back pressure to the transmitting end when the receiving end is running out of buffers.

The Empty FIFO, which holds pointers to empty Rx buffers, asserts an *almost_empty* flag when the number of pointers in the FIFO is less then a predetermined number

(7, in our case). When the Tx detects the *almost_empty* flag, Tx_ENDEC sends halt bytes in place of the normal control sequence.

At the remote end, when the Rx detects halt bytes in the incoming stream, it deasserts the *remote_ready* flag, causing the remote Tx to cease sending messages.

Eventually, when the SBus controller finishes reading the received messages from the SRAM, it returns the freed buffer pointers to the Empty FIFO, causing the *almost_empty* flag to be deasserted. When the Tx detects the deassertion of the *almost_empty* flag, it stops sending halt bytes and sends the normal byte sequence.

When the remote Rx detects the normal control bytes in the message stream, it asserts *remote_ready* to enable the remote Tx to resume message transmission.

4.5 Interface to the SunOS

The software in the SPARCStation kernel is responsible for the followings:

- Assembling any header information (such as destination and message length) required by the routers in data frames.
- Keeping a list of free-buffer pointers. When there is a message to send, it obtains a free buffer pointer and generates the corresponding SRAM address on the address lines (*PA[16:0]*).
- Asserting *PA[17]* to indicate End Of Write after a message is sent to the slave.
- Asserting *PA[18]* to indicate End of Read after a message sent from the slave is read. The freed buffer pointer is returned to the Empty FIFO for later use.
- Asserting *PA[19]* when servicing Interrupt Request[7] (*/IntReq[7]*), the request for message transfer from the Rx to the SBus controller.
- Asserting *PA[20]* when servicing Interrupt Request[6] (*/IntReq[6]*), the request for reclaiming a freed buffer pointer from the Recycle FIFO.

Chapter 5

Simulation

5.1 Generation of Simulation Results

To verify the correctness of the SBus interface board and to obtain timing information, modules were written in Verilog and Altera Hardware Description Language (AHDL) to model the components' behavior on the board.

The models of the SBus controller, the SRAM, the FIFOs and the ENDEC were written in Verilog. To reflect the real behavior of each component, timing constraints (such as signal hold-time and clock-to-data-valid latency) as specified in the data sheet were modeled as closely as possible in each of these modules.

The Txfsm, the Rxfsm, the SBusfsm, the Nxtfsm and the CRC submodule were written in AHDL and compiled into Edif files using the Altera MaxplusII software. The Edif files were converted to Verilog for simulation.

Each of the above modules was simulated, debugged and modified with separate drivers and test vectors in isolation before all modules were integrated for the final simulation.

5.2 Simulation Results

A loopback simulation was used to validate the operation of the SBus interface board. The operation of sending and receiving a message, as described in the previous chap-

ters, was verified in the following loop:

1. **SBus Controller to SRAM**: Messages are generated by the SBus controller connected to the SBusfsm. The SBusfsm writes the data from the controller into a SRAM buffer specified by the controller. When the End of Write (EOW) is detected, the buffer pointer and length are written to the Full FIFO.

2. **SRAM to Tx**: The Tx constantly monitors the Full FIFO for message arrival. When the Full FIFO is not empty, the Tx fetches the first buffer pointer in the queue and transmits the message to the Tx_ENDEC.

3. **Rx to SRAM**: To form a loop, the Tx_ENDEC is looped back to the Rx_ENDEC. When the Rx detects the start delimiter of a message, it retrieves a buffer pointer from the Empty FIFO for the incoming message. When an error-free message is received, the Rx writes the pointer to the Done FIFO.

4. **SRAM to SBus Controller**: The SBusfsm monitors the Done FIFO for message arrival. When the Done FIFO is not empty, the SBusfsm asserts an Interrupt Request to the SBus controller. The SBus controller then reads the message from the SRAM and verifies the correctness of the message.

To avoid any a priori assumptions, the message lengths, contents and arrival times are generated by random number generators provided in Verilog, allowing the message arrival at the SBus controller to be at a random time, of random length and with random content.

5.2.1 Verification of TDMA Plan

Since each module of the SBus board operates concurrently, the SBus controller can be writing a message to the memory module while the Tx is transmitting a second message and the Rx is receiving a third message. The proper functioning of the TDMA was verified by examining the multiplexed signals which drive the memory module and the interleaving reads and writes initiated by the various modules.

5.2.2 Verification of Flow-Control

In order to verify the flow-control mechanism, the size of the Empty FIFO was adjusted to simulate a low-memory condition. The Empty FIFO was initialized to contain 8 buffer pointers.

The flow-control mechanism was tested as follows:

1. SBus controller writes a message to SRAM.
2. Tx transmits the message to Rx (due to the looping of messages in simulation).
3. Rx detects the message and retrieves a buffer pointer from Empty FIFO, causing the *almost-empty* flag to be asserted.
4. Tx detects the *almost-empty* flag and sends halt bytes in place of idle bytes (if in the idle state), or in place of the start control bytes (if transmitting the start delimiter) or in place of the CRC control bytes (if transmitting the CRC sequence).
5. Rx detects halt bytes in the incoming bit stream and deasserts *remote_ready.*
6. Tx (after sending its current message) detects the deassertion of *remote_ready* and sends halt bytes, even if there are pending messages in the Full FIFO.
7. After reading the message from SRAM, the SBus controller frees the buffer by returning its pointer to Empty FIFO.
8. Empty FIFO deasserts the *almost_empty* flag (because there are now more than 7 buffer pointers in the FIFO).
9. Tx detects the deassertion of *almost_empty* and sends the normal control bytes, instead of the halt bytes.
10. Rx detects the normal control bytes in the incoming bit stream and asserts *remote_ready.*
11. Tx detects the assertion of *remote_ready* and resumes message transmission if there is any pending message in the Full FIFO.

Thus far, no mention has been made of the simulation of the SBus interface board under different network traffic conditions. The network traffic can be modeled in Verilog simulation. This can be accomplished by using queueing models which use predicted job creation and processing rates to simulate the operation of the system under varying loads. The jobs' arrival rates and processing times can be represented by random functions which approximate the real world. Meanwhile, dynamic information concerning the current and maximum queue length, mean inter-arrival time and average waiting time can be gathered. Verilog provides random number generators which return integer values distributed according to standard probabilistic functions, such as Exponential, Poisson, and Erlang functions. Although the analysis remains to be done, this kind of stochastic network traffic analysis offers the potential for insights into the performance ramifications of the board design.

Chapter 6

Performance Evaluation and Design Options

Previous chapters explained the details of the physical components and operation of the SBus interface board. This chapter presents a performance evaluation of the board.

6.1 Performance Evaluation

6.1.1 Throughput

The SBus board adopts a host-based design supporting only single word transfer (i.e. non-burst mode). Operating at 25 MHz, the peak transfer rate of the SBus controller is 100 MBytes/s. With single word transfer mode between the SBus controller and the board, the effective I/O bandwidth is approximately 33 MBytes/s, since each SBus transaction requires three cycles: address strobe, data transfer and acknowledgment.

To evaluate the performance of the Tx and the Rx, we define the protocol efficiency, η, as follows:

$$\eta = \frac{length\ of\ data}{length\ of\ message}$$

For data length l and protocol overhead τ, the efficiency η_l is given by the following equation:

$$\eta_l = \frac{l}{\tau + l}$$

The overhead, τ, included in every message is fixed at 7 words (4 words of control bytes, 1 word of start delimiter and 2 words of end delimiter and checksum).

Due to the lack of real application statistics concerning message data lengths, the efficiency is computed under several assumptions regarding the probability distribution of data lengths (in words), $Pr(l)$:

1. $Pr(l)$ is uniform or Gaussian with the expected data length, $E[l]$, equals 128. The efficiency η_{128} is about 95%.

2. $Pr(l)$ is a shifted Gaussian with the expected data length, $E[l]$, equals 64. The efficiency η_{64} is about 90%.

3. $Pr(l)$ is a shifted Gaussian with the expected data length, $E[l]$, equals 32. The efficiency η_{32} is about 80%.

Thus, after excluding the protocol overhead, the Tx and the Rx are capable of transmitting and receiving at an average rate of 80–95 Mbits/s, when data lengths are longer than 32 words. The bandwidth is significantly better than those of the existing LAN architectures. For example, the peak transfer rate of Ethernet is 10 Mbits/s [6], while the peak transfer rate of Token Ring is either 4 or 16 Mbits/s [7]. The transfer rate of the SBus board interface is comparable to that of FDDI, since both of them operate in the range of 80–95 Mbits/s [16]. However, FDDI and our intended network design use different network topologies and exhibit different scaling properties. The FDDI network is a ring network. The addition of workstations will degrade the performance of the ring and will eventually saturate the network. On the other hand, the SBus board is intended to be used in a network with a fast centralized switching hub whose performance will be less sensitive to an increase in network size.

The ATM packet-switching architecture is composed of 53-byte cells where each cell consists of 5 control bytes. Hence the efficiency of ATM network is approximately 90%, which is comparable to η_{64} calculated above. ATM is a scalable architecture permitting Gbits/s data transfer. The most common rate that is being used currently is OC-3 (155.52 Mbits/s) which delivers a throughput of 140 Mbits/s. One can see that the ATM offers a better performance when compared to the SBus board. However, the higher throughput is achieved at a higher cost, since the ATM chip sets are fairly expensive.

6.1.2 Latency

Another performance metric by which the SBus board can be judged is *message latency*, which we define as the time elapsed between the first word written to the SBus board by the sender and the last word read from the destination SBus board by the receiver.

As described in Chapter 4, a message transfer from host A to host B (each equipped with an SBus board) requires the following steps:

- Step 1: Host A to Memory on SBus board A. The message is written from host A to its SBus board memory.
- Step 2: Memory to Tx. The message is retrieved by the Tx and sent to the router.
- Step 3: Routing. The router delivers the message to the SBus board at host B.
- Step 4: Rx to Memory. The message is written to the SBus board memory by the Rx at host B.
- Step 5: Memory on SBus board B to host B. The message is read by host B from the SBus board.

To derive the latency due to the SBus board in the process above, the routing time is ignored since it is independent of our board design. We define *board latency*, $\mathcal{T}$, as

the amount of time that data resides on the SBus board, i.e. message latency minus routing time. Assuming that the length of a data frame is l words long, the steps in a message transfer involving the SBus board require the following numbers of clock cycles:

- Step 1: $8 \times l$. A word is written from host A to the SBus board A once every 8 clock cycles (during SWR_cycle).
- Step 2: $8 \times (l + \tau)$, where τ is the protocol overhead (section 6.1.1). The Tx_ENDEC transmits one word every 8 clock cycles.
- Step 4: $8 \times (l + \tau)$. The Rx_ENDEC receives one word every 8 clock cycles.
- Step 5: $8 \times l$. A word is read from the SBus board B by host B once every 8 clock cycles (during SRd_cycle).

The protocol overhead τ was calculated to be 7 words in section 6.1.1. For the purpose of latency calculation, step 4 can be considered as overlapping in time with step 2 (which can be seen if the routing time is assumed to be 0 for example), the board latency of message length l, $\mathcal{T}$, is given by the following equation:

$$\begin{aligned} \mathcal{T} &= 8l + 8(l + 7) + 8 + 8l \\ &= 24l + 64 \end{aligned}$$

In order to see the order of magnitude of the board latency, $\mathcal{T}$ is computed for three different values of l:

- $l = 32$: $\mathcal{T}$ equals 832 cycles or 33 μsec. The shortest possible time required by a 32-word transfer over a fiberoptic link is approximately 10 μsec.
- $l = 64$: $\mathcal{T}$ equals 1600 cycles or 64 μsec. The shortest possible time required by a 64-word transfer over a fiberoptic link is approximately 21 μsec.
- $l = 128$: $\mathcal{T}$ equals 3136 cycles or 125 μsec. The shortest possible time required by a 128-word transfer over a fiberoptic link is approximately 41 μsec.

The latency figures above underscore the fact that the board latency is largely due to the buffering of messages between the host machine (or more precisely the SBus controller) and the board. The buffering is necessary due to the following two considerations:

1. The host machine may compose several messages simultaneously.

   In deriving the board latency, it is assumed that the host writes the data words of a message consecutively. However, the host machine may in fact compose several messages at the same time. For example, the board memory may be directly mapped into the host machine's address space (to avoid copying) and several applications may simultaneously use the mapped memory to compose their messages. The support of such a technique of message creation requires buffering.

2. The SBus controller may take an arbitrary amount of time to service a request asserted by the SBus board.

   Because the SBus is generally shared by multiple SBus devices, buffering of messages is essential. The SBus controller may need to service other SBus devices on the bus, and therefore may take an arbitrary amount of time to service requests asserted by the SBus board.

6.2 Design Options

The SBus system described in this thesis can be improved in several ways. Some of the more important of these possibilities are described below:

1. **Higher Throughput**

   As the memory module can deliver 100 MBytes/s I/O bandwidth and greatly exceeds the current requirement of the SBus module, the Tx and the Rx, the bandwidth of the board can be easily increased by two-fold.

   With the TDMA plan, one-fourth of the SRAM bandwidth is available to the Tx, another one-fourth to the Rx, and the remaining to the SBus module. The

Tx and the Rx can double their throughputs and fully utilize their SRAM bandwidth by each making use of a pair of ENDECs. To complement this change, the SBus module can also increase its throughput to 50 MBytes/s by using double word burst mode transfer. Assuming that the number of messages sent from the board roughly equals the number of messages received, the throughputs of the SBus controller, the Tx and the Rx will be balanced in this case (in the sense that none of the subsystems presents a bottleneck for others).

While the throughput can be easily improved by a factor of two, the options for attaining a yet higher performance are not as straight-forward. Such a goal would require significant redesign to improve the bandwidth of all parts of the board. One way to achieve the goal is to separate the Tx from the Rx by requiring each block to have its own SRAM and FSM logic. While increasing the transfer rate, such a technique may require two SBus expansion slots instead of one. This option limits the flexibility of the SBus expansion system and may not be a practical solution.

Notice that the peak data rate of an SBus system is 100 MBytes/s. A host-based SBus provides burst transfers of up to 16 words with two clock cycles of overhead, yielding a burst transfer rate of 88 MBytes/s. If this throughput of the SBus system is judged to be inadequate, a different computer architecture could be exploited for higher data bandwidth. For instance, the Alpha architecture delivers better performance and higher throughput than a SPARCStation and an increasing number of system developers are adopting the Alpha architecture. Unfortunately, the design of a high bandwidth Alpha interface would require very careful engineering.

2. **Error Detection and Frame Resend**

In the present system, when the Rx detects an error in a received frame, the message is immediately dropped. Therefore, the responsibility of retransmission is delegated to the application software and no resend mechanism is provided. The performance of the interface can be improved by having a link-level retry if an error frame is received. However, since fiberoptic cables, which have very low error rate (approximately 10^{-12}), are used as the transmission media, the

usefulness of a link-level retry mechanism may not be worthwhile.

Chapter 7

Conclusion

This thesis presented the design of a SPARC Bus interface board. The board is intended as an expansion board in any SBus system. It requires a minimal number of handshaking signals and features a very simple flow-control protocol. The modular design allows the system to be treated as a black box which is capable of connecting any SBus system into a network of arbitrary size.

The current design effectively trades off constraints in speed, size, cost and ease of implementation. As mentioned in Chapter 6, it is possible to adopt a system with higher performance in terms of speed and bandwidth, but only at the cost of dealing with a more complex and expensive design.

In summary, the design objectives stated in the Introduction were fulfilled in the following ways:

- **High Bandwidth:** The peak transfer bandwidth that the SBus board supports is 100 Mbits/s. This is permitted by the use of the FDDI protocol and a fiberoptic link.
- **Compact Form Factor:** Most of the logic on the board is implemented in an Altera EPLD, permitting a densely integrated system. In addition to the EPLD, only a handful of devices are required on the board: memory devices, FDDI Encoder/Decoders and optical transmitter/receiver. The components will readily fit on the specified size of the SBus board.

- **Ease of Implementation and Maintenance:** The board provides transparency among modules and facilitates modification and debugging. It is designed in a modular fashion (SBus module, Tx and Rx), and each module needs only to know the interface signals (rather than the internal operation) of other modules. This gives the designer the flexibility to modify and debug each subsystem independently, as was done during the simulation and verification phase of the board.
- **Low Protocol Overhead:** We have adopted a simple protocol which incurs little overhead in transferring messages. For the average message lengths between 32 to 128 bytes, the overhead is 5% to 20%. As described in Chapter 4, flow-control handshaking information is piggybacked onto the control bytes in the message stream, thereby incurring no extra overhead.

Several lessons were learned from this project. First, one should start with a simple design and improve upon it. The original design objectives were ambitious in terms of transfer rate, flow-control and link-level error recovery mechanisms, which made the project difficult to implement. While the revised objectives for the project were more humble, they allow us to come up with a design which can be improved upon readily (as mentioned in the previous chapter).

The second major lesson learned was to expect bugs in software. The Altera MaxplusII compiler has numerous problems in compiling big modular designs (for instance, Txfsm) and some of them were very hard to detect. (For example, the compiler could not compile correctly when the module had a certain number of states.)

In conclusion, this thesis presented the design of a prototype interface board which represents a stepping stone towards the goal of developing local area supercomputing. It serves as an indicator that the goal of having powerful and tightly-coupled networks can be realized in the future.

Bibliography

[1] Advanced Micro Devices. *The SUPERNET Family for FDDI.* 1991/1992 World Network Data Book.

[2] Advanced Micro Devices. *Ethernet/IEEE 802.3 Family* 1992 World Network Data Book.

[3] Altera. *Application Handbook*, 1992.

[4] Altera. *Data Book*, 1993.

[5] Altera. *MAX+PLUS II Text Editor & AHDL*, 1991.

[6] Computer Design, *High-performance networks challenge Ethernet.* July 1992, P77-92.

[7] Dimitri Bertsekas and Robert Gallager. *Data Networks.* Second Edition, Prentice Hall, 1992.

[8] Robert H. Halstead, Jr. *6.033 Notes on Networking and Communication*, 1987.

[9] Integrated Device Technology, Inc. *Specialized Memories and Modules.* 1992 Data Book.

[10] James D. Lyle. *SBus, Information, Applications, and Experience.* Springer-Verlag New York, Inc., 1992.

[11] Robert M. Metcalfe and David R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Communications of the ACM*, C-19(7); 395-404, July 1976

[12] Craig Partridge. *Gigabit Networking.* Addison-Wesley, 1994.

[13] Michael D. Schroeder et al. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Area sin Communications.* Vol.9, No.8; 1318-1335, October 1991.

[14] Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems, Design and Evaluation.* Second Edition, Digital Press, 1992.

[15] Sun Microsystems. SBus Specification B.0, 1990.

[16] Andrew S. Tanenbaum. *Computer Networks.* Second Edition, Prentice Hall, 1989.

Appendix A

SBus Interface Board Specification

A.1 Device Components

- FDDI ENDEC: Part Number AM7984A (Advanced Micro Device).
- FDDI EDS: Part Number AM7985A (Advanced Micro Device).
- SRAM: 4 megabit memory cell (128K x 32). Part Number IDT7M4013 (Integrated Device Technology).
- FIFO: 256 x 9-bit parallel SyncFIFO with clocked read and write controls. Part number IDT72201 (Integrated Device Technology).
- Altera EPLD: MAX 7000 Programmable Logic. Part Number EPM7192 (Altera).

A.2 Device Timing

The timing specification of each component is as follows:

- System Clock: 25 MHz, aligned to the Nibble Clock of the ENDEC.
- ENDEC Clock: 12.5 MHz.

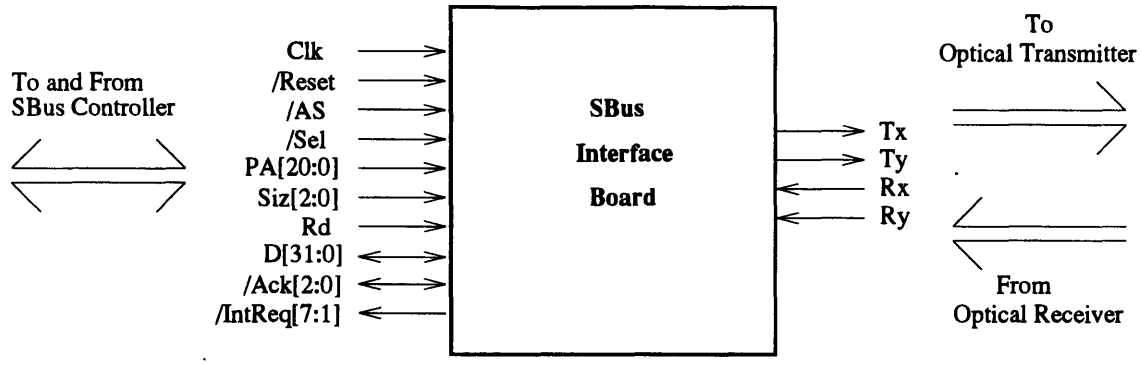

Figure A.1: SBus Interface Signals

- SRAM: 15 ns access time.
- FIFO: Asynchronous read/write with 10 ns access time.
- Altera: EPM7192, with 12 ns propagation delay.

A.3 Board Interface Signaling

The SBus board interface signals are shown in Figure A.1.

A.4 Intermodular Signaling

The top level module interconnection among SRAM, Tx, Rx, and SBus module is shown in Figure A.2.

A.5 Format of Message Frame

The format of a message frame is closely related to that of the FDDI Encoding Table [1]. During normal operation when there is no back pressure applied to the Tx, the message frame format is as follows:

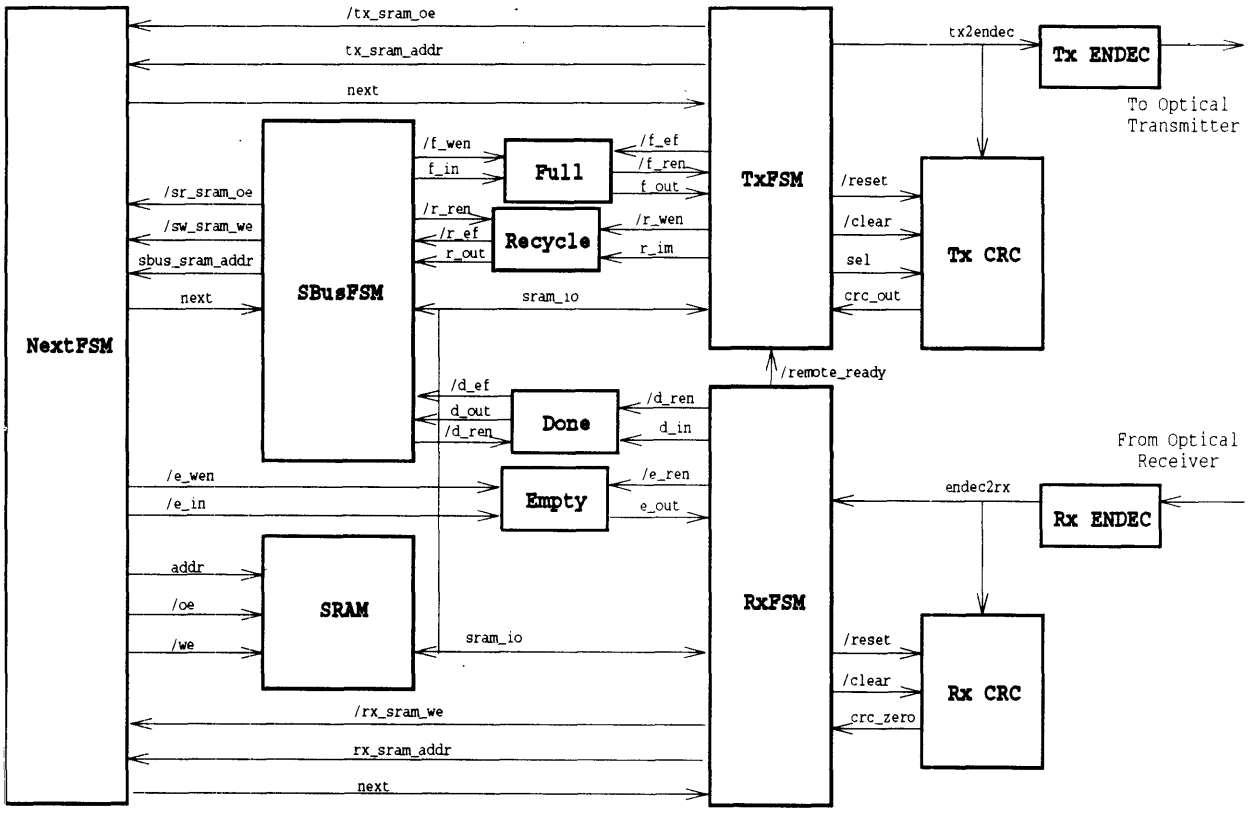

Figure A.2: Intermodular Signaling

1. Idle bytes (16 or more IDLE_bytes).
2. Start delimiter (a JK_byte followed by 3 ZERO_bytes).
3. Data (1 to 256 32-bits word).
4. End delimiter (2 JK_bytes).
5. Checksum (6 bytes of CRC checksum).

When the flow-control mechanism applies back pressure to the Tx, the message frame format becomes the followings:

1. Halt bytes (16 or more HALT_bytes).
2. Start delimiter (a JK_byte followed by two ZERO_bytes and a HALT_byte).
3. Data (1 to 256 32-bits words).
4. End delimiter (a JK_byte followed by a HALT_byte).
5. Checksum (6 bytes of CRC checksum).

A.6 Memory Module

Name	I/O	Description
addr[16:0]	I	Address lines.
io[31:0]	I/O	Data lines for SRAM content.
/we	I	Write enable.
/oe	I	Output enable.

The SRAM module (implemented in Verilog) has 128K x 32-bit memory cells. The *addr[16:0]* addresses a particular cell whose content is output when */oe* is asserted, or the cell is written with data on *io[31:0]* when */we* is asserted.

A.7 FIFO

There are 4 FIFOs on the SBus board, namely:

1. Empty FIFO (256 x 9-bit)
2. Recycle FIFO (256 x 9-bit)
3. Full FIFO (256 x 17-bit)
4. Done FIFO (256 x 17-bit)

Empty FIFO and Recycle FIFO are used to keep track of buffer pointers, so a 9-bit port is sufficient. Full FIFO and Done FIFO are used to record both the buffer pointers and length of messages, hence a 17-bit port is required.

Each FIFO has the following I/O signals (where W is the respective width of each FIFO):

Name	I/O	Description
Wclk	I	Write Clock.
Rclk	I	Read Clock.
/reset	I	System reset.
in[W:0]	I	FIFO input.
out[W:0]	O	FIFO output.
/ren	I	Read enable.
/wen	I	Write enable.
/ef	O	Empty flag.
/ff	O	Full flag.
/ae	O	Almost-empty flag.
/af	O	Almost-full flag.

The FIFOs are capable of asynchronous Read/Write, i.e. it can service both Read and Write simultaneously.

The empty flag (*/ef*) and the full flag (*/ff*) are asserted when the FIFO is empty and full respectively. The programmable flags almost-empty (*/ae*) and almost-full (*/af*) are asserted when the FIFO contains (7) or (Full - 7) items respectively.

A.8 Next FSM

Name	I/O	Description
sysclk	I	System clock.
/reset	I	System reset.
tx_sram_addr[16..0]	I	SRAM address driven by Tx.
rx_sram_addr[16..0]	I	SRAM address driven by Rx.
sb_sram_addr[16..0]	I	SRAM address driven by SBus module.
/rx_sram_we	I	SRAM write enable driven by Rx.
/sw_sram_we	I	SRAM write enable driven during a SBusWrite.
/tx_sram_oe	I	SRAM output enable driven by Tx.
/sr_sram_oe	I	SRAM output enable driven during a SBusRead.
sram_addr[16..0]	O	SRAM address.
/sram_oe	O	SRAM output enable.
/sram_we	O	SRAM write enable.
/sb_wen_empty	I	Empty FIFO write enable driven by SBus module.
/rx_wen_empty	I	Empty FIFO write enable driven by Rx.
/wen_empty	O	Empty FIFO write enable.
empty_in[8..0]	O	Empty FIFO input.
next[1..0]	O	Current TDMA cycle.
phase	O	Phase (0 or 1) of the current TDMA cycle.

The Nxtfsm is the coordinator for the SBusfsm, the Txfsm and the Rxfsm. It generates *next[1:0]* and *phase* signals to indicate the current TDMA cycle. During a particular TDMA cycle, inputs from the specified FSM are driven onto *sram_addr*, */sram_oe* and */sram_we*. The Next FSM also coordinates the Writes from both the Rxfsm and the SBusfsm to the Empty FIFO.

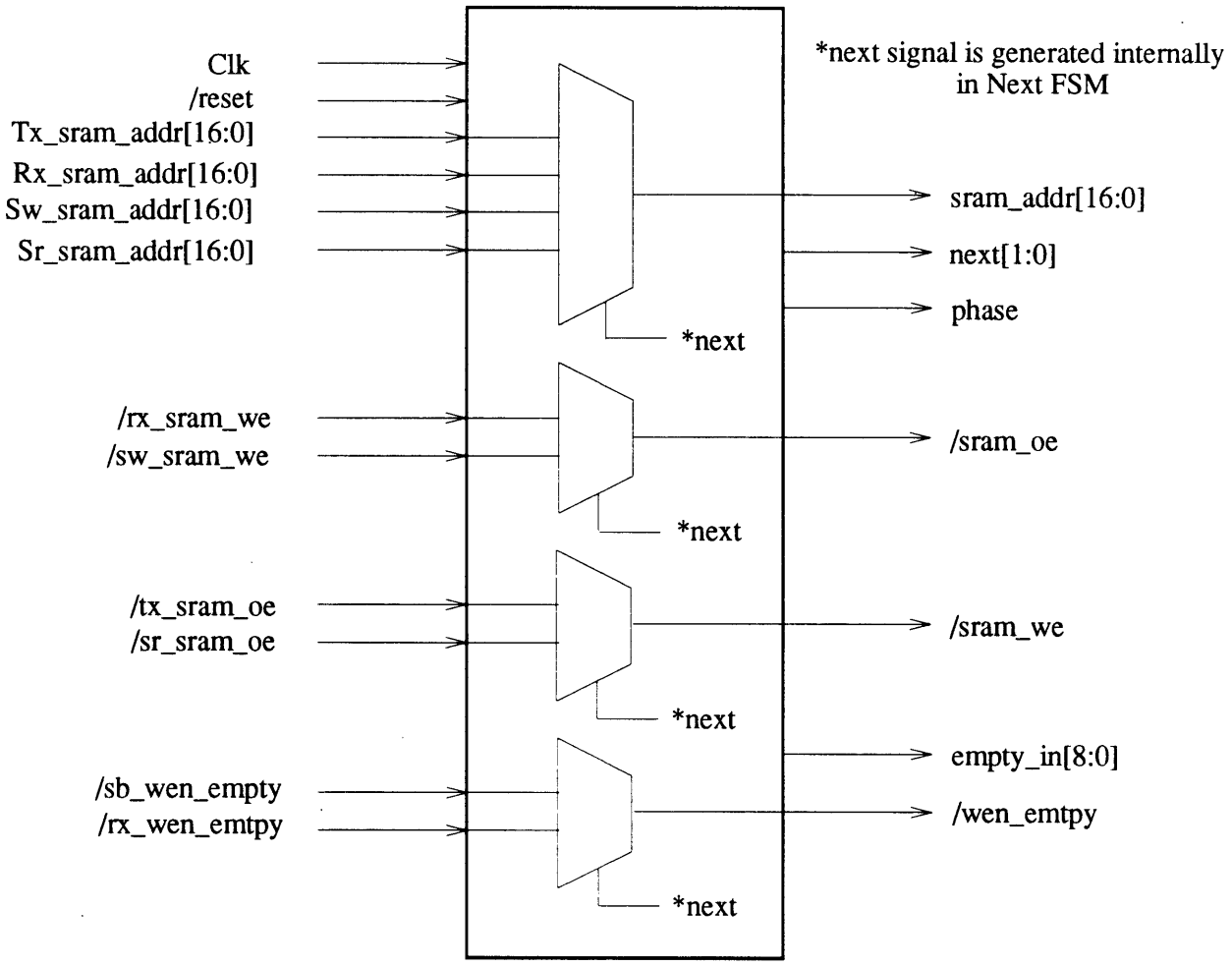

Figure A.3: Next FSM

A.9 SBus Module

Name	I/O	Description
sysclk	I	System clock.
/reset	I	System reset.
next[1:0]	I	Current TDMA cycle.
phase	I	Phase (0 or 1) of current TDMA cycle.
/ef_recycle	I	Recycle FIFO empty flag.
/ef_done	I	Done FIFO empty flag.
done_out[16:0]	I	Input from the Done FIFO (containing buffer pointer and length of received messages).
recycle_out[8:0]	I	Input from the Recycle FIFO (containing buffer pointer to be reclaimed by SBus controller).
PA[20:17]	I	Control lines signalled by SBus controller.
/AS	I	SBus Address Strobe, asserted when device is selected.
/Sel	I	SBus Slave Selected, asserted when device is selected.
Rd	I	SBus Read, asserted when current cycle is SBusRead.
/IntReq[7:6]	O	SBus Interrupt Requests, asserted when slave requires service from controller.
/Ack[2:0]	O	SBus Acknowledgment, asserted after read/write request has been serviced by slave.
D[31:0]	I/O	SBus data lines, for writing/reading data to/from interface board.
port_sram[31:0]	I/O	SRAM data lines, for writing/reading data to/from SRAM.
/sw_sram_we	O	SRAM write enable for SBusWrite.
/sr_sram_oe	O	SRAM output enable for SBusRead.
/wen_full	O	Full FIFO write enable, asserted after SBus finishes writing message to SRAM.
/wen_empty	O	Empty FIFO write enable, asserted after SBus finishes reading message from SRAM.
/ren_recycle	O	Recycle FIFO read enable, asserted when SBus services */IntReq[6]*, i.e. reclaiming buffer pointer.
/ren_done	O	Done FIFO read enable, asserted when SBus services */IntReq[7]*, i.e. reading pointer and length of received buffer.

The SBusfsm serves as an interface between the SBus controller and the rest of the board. It responds to read and write requests from the controller. Notice that */IntReq[7]* and */IntReq[6]* are connected to the empty flags of the Done FIFO and the Recycle FIFO respectively. If all of the control bits (*PA[20:17]*) are deasserted,

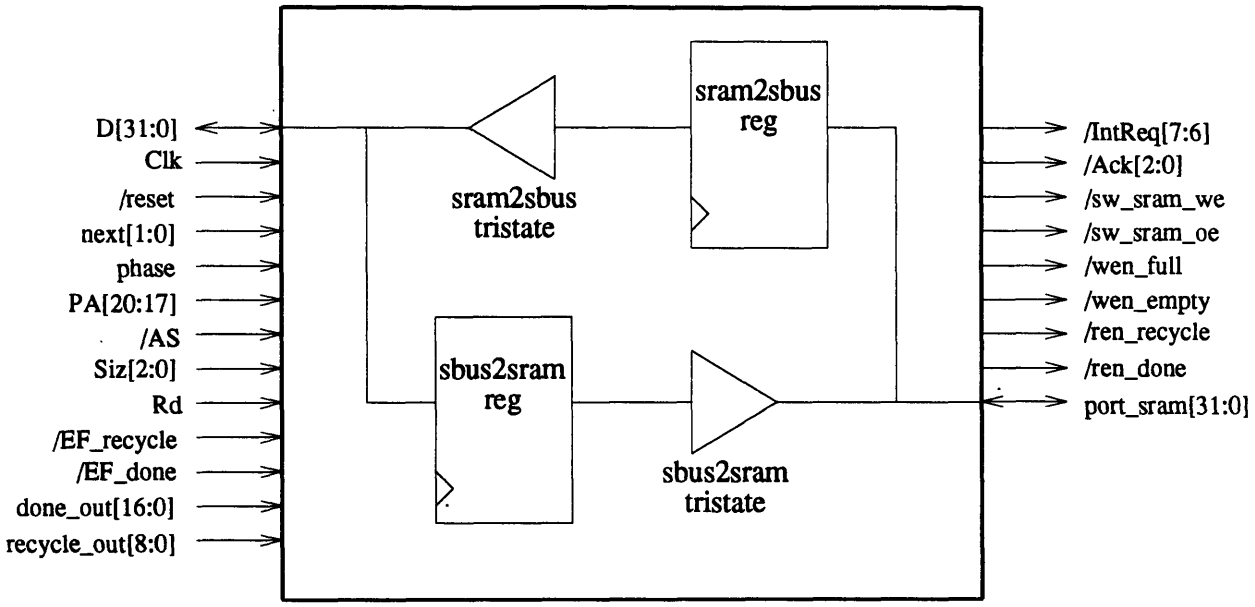

Figure A.4: SBus Module

the request is deemed to be a normal read or write:

1. SBusRead (*/AS*, */Sel* and *Rd* asserted):

   When an SBusRead is requested, the SBusfsm waits for a SRd_cycle. During the SRd_cycle, the SBusfsm generates an acknowledgment, enables SRAM output by asserting */rx_sram_oe* and drives data onto *D[31:0]*.

2. SBusWrite (*/AS*, */Sel* and *Rd* deasserted):

   When an SBusWrite is requested, the SBusfsm waits for a SWr_cycle before asserting the SRAM write enable (*/sw_sram_we*), driving data from the SBus (*D[31:0]*) onto the SRAM (*sram_io[31:0]*) and generating an acknowledgment to the SBus controller.

During both SBusRead and SBusWrite cycles, the Physical Address (*PA[16:0]*) driven by the SBus controller is used directly to address the SRAM. If one of control bits (*PA[20:17]*) is asserted, the following actions are taken:

1. *PA[17]*: signals an End of Message Write from SBus to SRAM. *PA[16:0]*, which contains the buffer pointer and the length of the last written message, is written to the Full FIFO, followed by an acknowledgment from the slave to the SBus controller.

2. *PA[18]*: signals an End of Message Read from SRAM to SBus. *PA[16:8]*, which points to the buffer read, is written to the Empty FIFO. Since the Empty FIFO is also addressed by the Rx, the SBusfsm waits for a SBus Cycle (SRd_cycle or SWr_cycle) before asserting */wen_empty*.

3. *PA[19]*: signals Interrupt Service for */IntReq[7]*, which is asserted when there are messages ready to be read by the host machine (i.e. when Done FIFO is not empty). When the controller services the request, an acknowledgment is generated by the slave. The Done FIFO read enable is asserted and the output (which contains buffer pointer and length of received message) is driven onto the SBus data lines (*D[31:0]*).

4. *PA[20]*: signals Interrupt Service for */IntReq[6]*, which is asserted when there are buffer pointers to be reclaimed by the host machine (i.e. when the Recycle FIFO is not empty). In this case, an acknowledgment is generated by the slave, the Recycle FIFO read enable is asserted and the output (which contains buffer pointer ready to be reclaimed by the kernel) is driven onto the SBus data lines (*D[31:0]*).

A.10 Transmit Module

Name	I/O	Description
sysclk	I	System clock.
/reset	I	System reset.
/ef_full	I	Full FIFO empty flag, asserted when Full FIFO is not empty.
/ae_empty	I	Empty FIFO almost-empty flag, asserted when Empty FIFO is almost empty (i.e. with less than 7 pointers in the FIFO.)
full2tx[16:0]	I	Full FIFO output, contains pointers and lengths of buffers ready for transmission.
txin[31:0]	I	Data input from SRAM.
next[1..0]	I	Current TDMA cycle.
/remote_ready	I	Remote ready flag, asserted by Rx when remote end is ready for receiving messages.
tx_crc_out[7:0]	I	Input from Tx_CRC.
/tx_crc_reset	O	Signal to reset Tx_CRC.
/tx_crc_hold	O	Signal to freeze output of Tx_CRC.
tx_crc_sel[2:0]	O	Select one of the six Tx_CRC bytes.
/ren_full	O	Full FIFO read enable.
/wen_recycle	O	Recycle FIFO write enable.
tx2endec[10..0]	O	Data, control and parity bits from Tx to ENDEC.
/tx_sram_oe	O	SRAM output enable.
tx_sram_addr[16:0]	O	SRAM address.

After initialization, the Tx enters the idle state when idle bytes are sent. When the Full FIFO is not empty, the Tx fetches a pointer and message length from the Full FIFO, transmits a start delimiter followed by data, an end delimiter and a checkcode. When the Tx is done transmitting the message, it writes the transmitted pointer into the Recycle FIFO.

Since the Tx_ENDEC sends one byte of data once every 2 clocks, the Tx latches a 32-bit word from the SRAM and sends the word during the subsequent 8 clock cycles before proceeding to read another word. Due to the propagation delay inherent in an Altera EPLD, the parity bit (an XOR function with 9 terms) cannot be computed in one clock. To circumvent the problem, the SRAM word is latched 1 cycle before its first byte is sent. Then, during the cycle when each byte is sent, the XOR terms of

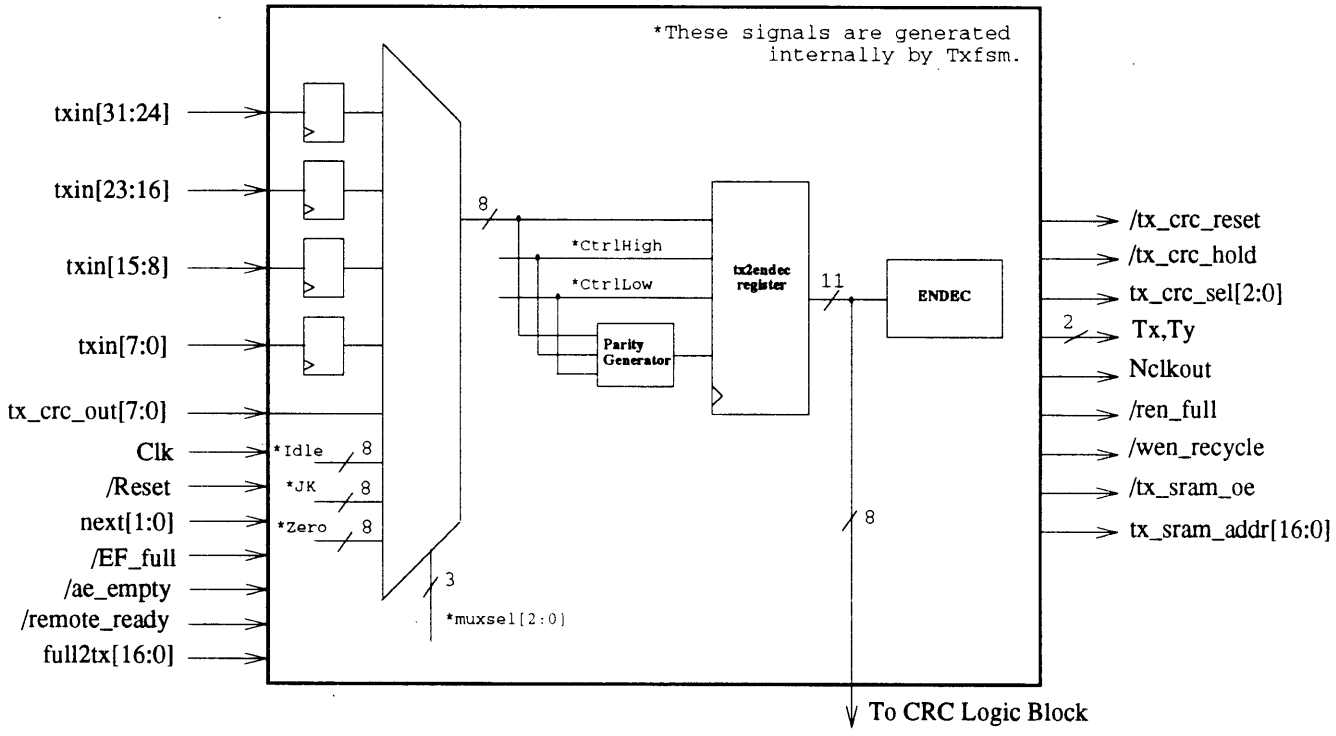

Figure A.5: Transmit Module

the lower and upper nibbles are computed and the odd parity bit is generated when the data byte is latched out to the Tx_ENDEC.

Notice that since the Tx can only access the SRAM during a Tx_cycle under the TDMA plan, the Tx spins in the idle state until it ensures that the SRAM read falls on a Tx_cycle.

Tx_crc_reset is asserted just before the first data byte is transmitted to the Tx_ENDEC and tx_crc_hold is asserted after the first JK pair of the CRC sequence is transmitted.

Flow control is implemented by asserting back pressure to the remote transmitter. The assertion of the almost-empty flag (*/ae*) of the Empty FIFO indicates that the remote receiver is running out of receiving buffers. The Tx will then send halt bytes and wait for the remote ready flag (*/remote_ready*) to be asserted before transmitting new messages.

A.11 Receive Module

Name	I/O	Description
sysclk	I	System clock.
/reset	I	System reset.
S[2:0]	I	ENDEC line status.
endec2rx[7:0]	I	Data input from ENDEC.
rcl	I	Ctl bit for lower nibble of ENDEC input.
rcu	I	Ctl bit for upper nibble of ENDEC input.
empty2rx[8:0]	I	Output of Empty FIFO, containing free buffer pointers.
next[1:0]	I	Current TDMA cycle.
rx_crc_zero[3:0]	I	Zero flag for each of the BCR in Rx_CRC.
/rx_crc_reset	O	Rx_CRC reset.
/rx_crc_hold	O	Rx_CRC hold.
/ren_empty	O	Empty FIFO read enable.
/wen_empty	O	Empty FIFO write enable.
/wen_done	O	Done FIFO write enable.
rxout[31:0]	O	Tri-stated output to SRAM.
/rx_sram_we	O	SRAM write enable.
rx_sram_addr[16:0]	O	SRAM address.
/remote_ready	O	Remote ready flag.

After reset, the Rx enters the idle state and waits for the arrival of the start delimiter of a message. When the first byte of the start delimiter is detected, the Rx retrieves a buffer pointer from the Empty FIFO to hold the incoming message. When an error-free message is received, the pointer and the length of the buffer are written to the Done FIFO. If errors appear in the message, the buffer pointer is returned to the Empty FIFO, essentially discarding the message.

Since the Rx can only write to the SRAM during a Rx_cycle under the TDMA plan and a message can arrive at the Rx_ENDEC at any arbitrary time, the Rx has a 2-deep 32-bit-wide FIFO consisting of Register A and Register B. Each byte from the Rx_ENDEC is first written to Register A. When a 32-bit word is composed in Register A, it is shifted to Register B. Register A is ready to receive the next incoming byte from the Rx_ENDEC, while Register B is ready to be written to the SRAM during the next Rx_cycle.

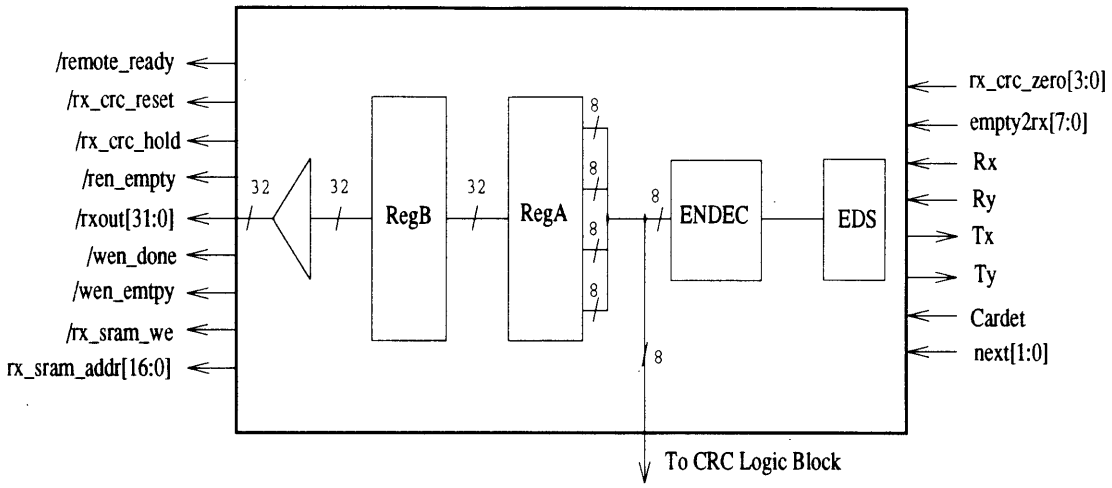

Figure A.6: Receive Module

The Rx is also responsible for asserting and deasserting the */remote_ready* flag. After reset, */remote_ready* is asserted; it is deasserted when the Rx detects halt bytes in the incoming bit stream and reasserted when normal control sequence is detected.

Appendix B

Module Codes

Codes are written in Verilog and Altera Hardware Description Language (AHDL) to model the behaviors of the components on the SBus interface board. The modules of the SBus controller, SRAM, FIFO and ENDEC are written in Verilog while the Txfsm, Rxfsm, SBusfsm, Nextfsm and CRC logic block are written in AHDL.

```
% This programmable device performs checksum using CRC-12 cyclic code %

TITLE 'BLOCK CHECK REGISTER';

SUBDESIGN crc12
(
        clk             : INPUT;
        /reset          : INPUT;
        /clear          : INPUT;
        /hold           : INPUT;  % hold the output when GND %
        crc_sel_[2..0]  : INPUT;  % select one of the six outputs %
        crc_in_[7..0]   : INPUT;  % input data %
        crc_out_[7..0]  : OUTPUT; % crc output %
        zero_[3..0]     : OUTPUT; % each bit represent if the crc_out is 0 %
)

VARIABLE
       c0_[11..0] : DFFE;
       c1_[11..0] : DFFE;
       c2_[11..0] : DFFE;
       c3_[11..0] : DFFE;

       feedback_[3..0]: node;

       datain_[3..0]: node;

       crc_state : MACHINE of BITS (crc)
                   WITH STATES (crc0  = 0,
                                crc1  = 1);

BEGIN

c0_[].clk = global(clk);
c0_[].clrn = /clear;
c0_[].ena =  /hold;

c1_[].clk = global(clk);
c1_[].clrn = /clear;
c1_[].ena =  /hold;

c2_[].clk = global(clk);
c2_[].clrn = /clear;
c2_[].ena =  /hold;

c3_[].clk = global(clk);
c3_[].clrn = /clear;
c3_[].ena =  /hold;

crc_state.clk = global(clk);
crc_state.reset = !global(/reset);

CASE (crc_state) IS

WHEN crc0 =>

    IF (/hold != GND) THEN
        datain_[0] = crc_in_[0];
        datain_[1] = crc_in_[2];
        datain_[2] = crc_in_[4];
        datain_[3] = crc_in_[6];
        crc_state = crc1;
    ELSE
        crc_state = crc0;
    END IF;

WHEN crc1 =>

        datain_[0] = crc_in_[1];
        datain_[1] = crc_in_[3];
        datain_[2] = crc_in_[5];
        datain_[3] = crc_in_[7];
        crc_state = crc0;

END CASE;

  %-------------------------------------------------------%
  % logic                                                 %
  %-------------------------------------------------------%

  IF (/hold != GND) THEN

       feedback_0 = c0_0 $ datain_0;
       c0_11 = feedback_0;
       c0_10 = feedback_0 $ c0_11;
       c0_9  = feedback_0 $ c0_10;
       c0_8  = feedback_0 $ c0_9;
       c0_7  = c0_8;
       c0_6  = c0_7;
       c0_5  = c0_6;
       c0_4  = c0_5;
       c0_3  = c0_4;
       c0_2  = c0_3;
       c0_1  = c0_2;
       c0_0  = feedback_0 $ c0_1;

       feedback_1 = c1_0 $ datain_1;
       c1_11 = feedback_1;
       c1_10 = feedback_1 $ c1_11;
       c1_9  = feedback_1 $ c1_10;
       c1_8  = feedback_1 $ c1_9;
       c1_7  = c1_8;
       c1_6  = c1_7;
       c1_5  = c1_6;
       c1_4  = c1_5;
       c1_3  = c1_4;
       c1_2  = c1_3;
       c1_1  = c1_2;
       c1_0  = feedback_1 $ c1_1;

       feedback_2 = c2_0 $ datain_2;
       c2_11 = feedback_2;
       c2_10 = feedback_2 $ c2_11;
       c2_9  = feedback_2 $ c2_10;
       c2_8  = feedback_2 $ c2_9;
       c2_7  = c2_8;
       c2_6  = c2_7;
       c2_5  = c2_6;
       c2_4  = c2_5;
       c2_3  = c2_4;
       c2_2  = c2_3;
       c2_1  = c2_2;
       c2_0  = feedback_2 $ c2_1;

       feedback_3 = c3_0 $ datain_3;
       c3_11 = feedback_3;
       c3_10 = feedback_3 $ c3_11;
       c3_9  = feedback_3 $ c3_10;
       c3_8  = feedback_3 $ c3_9;
```

```
      c3_7  = c3_8;
      c3_6  = c3_7;
      c3_5  = c3_6;
      c3_4  = c3_5;
      c3_3  = c3_4;
      c3_2  = c3_3;
      c3_1  = c3_2;
      c3_0  = feedback_3 $ c3_1;

 END IF;

 IF (c0_[] == 0) THEN
    zero_0 = GND;
 ELSE
    zero_0 = VCC;
 END IF;

 IF (c1_[] == 0) THEN
    zero_1 = GND;
 ELSE
    zero_1 = VCC;
 END IF;

 IF (c2_[] == 0) THEN
    zero_2 = GND;
 ELSE
    zero_2 = VCC;
 END IF;

 IF (c3_[] == 0) THEN
    zero_3 = GND;
 ELSE
    zero_3 = VCC;
 END IF;

 IF (crc_sel_[] == 0) THEN
      crc_out_[] = (
                c3_[1], c3_[0],
                c2_[1], c2_[0],
                c1_[1], c1_[0],
                c0_[1], c0_[0]);

 END IF;

 IF (crc_sel_[] == 1) THEN
      crc_out_[] = (
                c3_[3], c3_[2],
                c2_[3], c2_[2],
                c1_[3], c1_[2],
                c0_[3], c0_[2]);
 END IF;

 IF (crc_sel_[] == 2) THEN
      crc_out_[] = (
                c3_[5], c3_[4],
                c2_[5], c2_[4],
                c1_[5], c1_[4],
                c0_[5], c0_[4]);

 END IF;

 IF (crc_sel_[] == 3) THEN
      crc_out_[] = (
                c3_[7], c3_[6],
                c2_[7], c2_[6],
                c1_[7], c1_[6],
                c0_[7], c0_[6]);

 END IF;

 IF (crc_sel_[] == 4) THEN
      crc_out_[] = (
                c3_[9], c3_[8],
                c2_[9], c2_[8],
                c1_[9], c1_[8],
                c0_[9], c0_[8]);

 END IF;

 IF (crc_sel_[] == 5) THEN
      crc_out_[] = (
                c3_[11], c3_[10],
                c2_[11], c2_[10],
                c1_[11], c1_[10],
                c0_[11], c0_[10]);

 END IF;

END;
```

```
% This programmable device implements the Next State FSM of the SBus Board %

TITLE 'NEXT FINITE-STATE-MACHINE';

INCLUDE 'const.inc';

DESIGN IS 'nxtfsm'
BEGIN
    OPTIONS XOR_SYNTHESIS = ON;
    OPTIONS TURBO = ON;
    OPTIONS PARALLEL_EXPANDERS = ON;

   DEVICE 'nxtfsm' IS 'EPM7192QC160-2'
   begin
   end;
end;

SUBDESIGN nxtfsm
(
  %-------------------------------------------------------%
  % The clock of this fsm is running at 40 ns (sysclk)    %
  %-------------------------------------------------------%

sysclk               : INPUT; % system clock %
/reset               : INPUT; % system reset %

tx_sram_addr_[16..0] : INPUT; % SRAM addr driven by Tx %
rx_sram_addr_[16..0] : INPUT; % SRAM addr driven by Rx %
sb_sram_addr_[16..0] : INPUT; % SRAM addr driven by SBus %

/rx_sram_we          : INPUT; % SRAM /we driven by Rx %
/sw_sram_we          : INPUT; % SRAM /we driven by SBusWrite %
/tx_sram_oe          : INPUT; % SRAM /oe driven by Tx %
/sr_sram_oe          : INPUT; % SRAM /oe driven by SBusRead %

/sb_wen_empty        : INPUT; % Empty FIFO /wen driven by the SBus %
/rx_wen_empty        : INPUT; % Empty FIFO /wen driven by the Rx %

/wen_empty           : OUTPUT; % Empty FIFO /wen %
empty_in_[8..0]      : OUTPUT; % Empty FIFO input %
sram_addr_[16..0]    : OUTPUT; % SRAM address %
/sram_oe             : OUTPUT; % SRAM output enable %
/sram_we             : OUTPUT; % SRAM write enable %

next_[1..0]          : OUTPUT; % SRAM cycle %
phase                : OUTPUT; % phase of SRAM cycle %

)
```

```
VARIABLE
        nxt_state  : MACHINE of BITS (nxt[2..0])
                     WITH STATES ( nxt_state0   = 0,
                                   nxt_state1   = 1,
                                   nxt_state2   = 2,
                                   nxt_state3   = 3,
                                   nxt_state4   = 4,
                                   nxt_state5   = 5,
                                   nxt_state6   = 6,
                                   nxt_state7   = 7
                                 );
```

```
BEGIN

DEFAULTS

   /sram_we          = VCC;
   /sram_oe          = VCC;

END DEFAULTS;

nxt_state.clk     = global(sysclk);
nxt_state.reset   = !global(/reset);

CASE (nxt_state) IS

%-------------------------------------------------------%
% During each state, output SRAM cycle on next_[] and,  %
% select the corresponding signals from Tx, Rx, and SBus %
%-------------------------------------------------------%

WHEN nxt_state0 =>
     next_[] = RX_CYCLE;
     phase   = GND;

     sram_addr_[] = rx_sram_addr_[];
     /sram_we     = /rx_sram_we;

     /wen_empty   = /rx_wen_empty;
     empty_in_[] = rx_sram_addr_[16..8];

     nxt_state = nxt_state1;

WHEN nxt_state1 =>
     next_[] = RX_CYCLE;
     phase   = VCC;

     sram_addr_[] = rx_sram_addr_[];
     /sram_we     = /rx_sram_we;

     /wen_empty   = /rx_wen_empty;
     empty_in_[] = rx_sram_addr_[16..8];

     nxt_state = nxt_state2;

WHEN nxt_state2 =>
     next_[] = SWR_CYCLE;
     phase   = GND;

     sram_addr_[] = sb_sram_addr_[];
     /sram_we     = /sw_sram_we;

     /wen_empty   = /sb_wen_empty;
     empty_in_[] = sb_sram_addr_[16..8];

     nxt_state = nxt_state3;

WHEN nxt_state3 =>
     next_[] = SWR_CYCLE;
     phase   = VCC;

     sram_addr_[] = sb_sram_addr_[];
     /sram_we     = /sw_sram_we;

     /wen_empty   = /sb_wen_empty;
     empty_in_[] = sb_sram_addr_[16..8];

     nxt_state = nxt_state4;

WHEN nxt_state4 =>
     next_[] = TX_CYCLE;
     phase   = GND;

     sram_addr_[] = tx_sram_addr_[];
     /sram_oe     = /tx_sram_oe;

     /wen_empty   = /rx_wen_empty;
     empty_in_[] = rx_sram_addr_[16..8];

     nxt_state = nxt_state5;

WHEN nxt_state5 =>
     next_[] = TX_CYCLE;
     phase   = VCC;

     sram_addr_[] = tx_sram_addr_[];
     /sram_oe     = /tx_sram_oe;

     /wen_empty   = /rx_wen_empty;
     empty_in_[] =  rx_sram_addr_[16..8];

     nxt_state = nxt_state6;

WHEN nxt_state6 =>
     next_[] = SRD_CYCLE;
     phase   = GND;

     sram_addr_[] = sb_sram_addr_[];
     /sram_oe     = /sr_sram_oe;

     /wen_empty   = /sb_wen_empty;
     empty_in_[] = sb_sram_addr_[16..8];

     nxt_state = nxt_state7;

WHEN nxt_state7 =>
     next_[] = SRD_CYCLE;
     phase   = VCC;

     sram_addr_[] = sb_sram_addr_[];
     /sram_oe     = /sr_sram_oe;

     /wen_empty   = /sb_wen_empty;
     empty_in_[] = sb_sram_addr_[16..8];

     nxt_state = nxt_state0;

END CASE;

END;
```

```
% This programmable device implements the Receive FSM of the SBUS Board %

TITLE "RECEIVE FINITE-STATE-MACHINE";

INCLUDE "const.inc";

DESIGN IS "rxfsm"
BEGIN
    OPTIONS XOR_SYNTHESIS = ON;
    OPTIONS TURBO = ON;
    OPTIONS PARALLEL_EXPANDERS = ON;

   DEVICE "rxfsm" IS "EPM7192QC160-2"
   begin
   end;
end;

SUBDESIGN rxfsm
(
  %-----------------------------------------------------%
  % The clock of this fsm is running at 40 ns (sysclk) %
  %-----------------------------------------------------%

 sysclk                : INPUT; % system clock %
 /reset                : INPUT; % system reset %

 S_[2..0]              : INPUT; % line status of ENDEC %
 endec2rx_[7..0]       : INPUT; % data input from the endec %
 rcl                   : INPUT; % ctl bit for lower nibble of ender input %
 rcu                   : INPUT; % ctl bit for upper nibble of ender input %

 empty2rx_[8..0]       : INPUT; % output of Empty FIFO %
 next_[1..0]           : INPUT; % from nxtfsm %
 phase                 : INPUT; % 0 if it is the first half of a SRAM cycle %

 rx_crc_zero_[3..0]    : INPUT; % from rx crc, crc_zero[] == 0 mean crc checked %

 /rx_crc_reset         : OUTPUT; % reset for Rx_CRC %
 /rx_crc_hold          : OUTPUT; % hold for Rx_CRC %

 /ren_empty            : OUTPUT; % read enable for Empty_FIFO %
 /wen_empty            : OUTPUT; % write enable for Empty FIFO %
 /wen_done             : OUTPUT; % write enable for Done FIFO %

 rxout_[31..0]         : OUTPUT; % the tri-stated output to SRAM %
 /rx_sram_we           : OUTPUT; % output the latched data in bux to Sram %
 rx_sram_addr_[16..0]  : OUTPUT;

 /remote_ready         : OUTPUT; % flow control signal %
)
```

```
VARIABLE

  %-----------------------------------------------------%
  % Registers                                           %
  %-----------------------------------------------------%
    sram_ptr[8..0] : DFFE;

    % ============================================
      a 2-deep buffer:
      data bytes are composed in rx_rega to form a word.
      when a word is composed in rx_rega, it is shifted into rx_regb
      so that rx_rega is ready for receiving next word, and
      rx_regb is ready to be read by the bux/SRAM
    ============================================ %

    rx_rega[31..0]     : DFFE;
    rx_regb[31..0]     : DFFE;
    rx_regb_ena        : node;

    % rxout_valid is asserted when rx_regb has valid data %
    rxout_valid        : DFFE;

  %-----------------------------------------------------%
  % SRAM address counter                                %
  %-----------------------------------------------------%
    sram_cnt[7..0] : DFF;
    inc_sram_addr  : node;
    clr_sram_cnt   : node;

  %-----------------------------------------------------%
  % Tri-stated SRAM data path                           %
  %-----------------------------------------------------%

    rx_tri[31..0]  : tri;
    rx_tri_ena     : node;

  %-----------------------------------------------------%
  % flow control, indicating if remote is ready for message %
  %-----------------------------------------------------%
    /remote_ready  : DFFE;
```

```
%-------------------------------------------------------%
% FSM                                                   %
%-------------------------------------------------------%
      rx_state    : MACHINE of BITS (rx[4..0])
                    WITH STATES ( rx_idle      = 0,
                                  rx_idlewait  = 1,
                                  rx_ready0    = 2,
                                  rx_dummy0    = 3,
                                  rx_ready1    = 4,
                                  rx_dummy1    = 5,
                                  rx_ready2    = 6,
                                  rx_dummy2    = 7,

                                  rx_data0     = 8,
                                  rx_wait0     = 9,
                                  rx_data1     = 10,
                                  rx_wait1     = 11,
                                  rx_data2     = 12,
                                  rx_wait2     = 13,
                                  rx_data3     = 14,
                                  rx_wait3     = 15,

                                  rx_crc_jk1      = 16,
                                  rx_crc_jk1_wait = 17,
                                  rx_crc0      = 18,
                                  rx_crc0_wait = 19,
                                  rx_crc1      = 20,
                                  rx_crc1_wait = 21,
                                  rx_crc2      = 22,
                                  rx_crc2_wait = 23,
                                  rx_crc3      = 24,
                                  rx_crc3_wait = 25,
                                  rx_crc4      = 26,
                                  rx_crc4_wait = 27,
                                  rx_crc5      = 28,
                                  rx_crc5_wait = 29,
                                  rx_done      = 30,
                                  rx_done_wait = 31
                                );
```

```
BEGIN

  %-------------------------------------------------------%
  % Defaults Statements                                   %
  %-------------------------------------------------------%

DEFAULTS

   /rx_sram_we       = VCC;
   /ren_empty        = VCC;
   /wen_empty        = VCC;
   /wen_done         = VCC;
   /rx_crc_reset     = VCC;
   /rx_crc_hold      = VCC;

END DEFAULTS;

    rx_state.clk = global(sysclk);
    rx_state.reset = !global(/reset);

    sram_ptr[].clk = global(sysclk);
    sram_ptr[].clrn= global(/reset);
    sram_ptr[].prn = VCC;

    sram_cnt[].clk = global(sysclk);
    sram_cnt[].clrn= global(/reset);
    sram_ptr[].prn = VCC;

    rx_rega[].clk = global(sysclk);
    rx_rega[].clrn = global(/reset);
    rx_rega[].prn = VCC;

    rx_regb[].clk = global(sysclk);
    rx_regb[].ena = rx_regb_ena;
    rx_regb[].clrn = global(/reset);
    rx_regb[].prn = VCC;

    rxout_valid.clk =  global(sysclk);
    rxout_valid.clrn=  global(/reset);
    rxout_valid.prn =  VCC;

    /remote_ready.clk =  global(sysclk);
    /remote_ready.clrn=  global(/reset);
    /remote_ready.prn =  VCC;
```

```
CASE (rx_state) IS

%-------------------------------------------------------%
% since the endec is running at half of our speed,      %
% each data is hold for 2 cycles.                       %
% data is read at the latter cycle                      %
%-------------------------------------------------------%

WHEN rx_idle =>
  rx_state = rx_idlewait;

WHEN rx_idlewait =>

  % H_BYTE indicates that remote is running out of buffer %
  % deassert /remote_ready to prevent local Tx from      %
  % sending further messages                             %
  IF (rcu & rcl & (endec2rx_[] == H_BYTE)) THEN
     /remote_ready.d     = VCC;
     /remote_ready.ena   = VCC;
  END IF;

  % IDLE_BYTE indicates that the remote is ready for %
  % receiving msg, assert /remote_ready              %
  IF (rcu & rcl & (endec2rx_[] == IDLE_BYTE)) THEN
     /remote_ready.d     = GND;
     /remote_ready.ena   = VCC;
  END IF;

  % JK indicates start of message, enter reception loop %
  IF ((S_[] == ALS) & rcu & rcl & (endec2rx_[] == JK_BYTE))  THEN
     % receive a JK byte %
     % should be followed by 3 ZERO bytes before valid data %
     rx_state = rx_ready0;
  ELSE
     rx_state = rx_idle;
  END IF;

WHEN rx_ready0 =>
  % get buffer ptr from Empty FIFO %
  /ren_empty = GND;
  rx_state = rx_dummy0;

WHEN rx_dummy0 =>
  % enable the sram_ptr to latch in the fifo_ptr %
  sram_ptr[].ena = VCC;

  % clear sram_cnt, since we are writing into a new buffer %
  clr_sram_cnt  = VCC;

  % clear the data_valid flag %
  rxout_valid.ena = VCC;
  rxout_valid.d = GND;

  rx_state = rx_ready1;

WHEN rx_ready1 =>
  % ignore the first ZERO byte %
  rx_state = rx_dummy1;

WHEN rx_dummy1 =>
  rx_state = rx_ready2;

WHEN rx_ready2 =>
  % ignore the second ZERO byte %
  rx_state = rx_dummy2;

WHEN rx_dummy2 =>
  % if the third byte is H, then remote is running out buffers %
  % desert remote ready %
  IF (rcu & rcl & (endec2rx_[] == H_BYTE)) THEN
     /remote_ready.d    = VCC;
  ELSE
     /remote_ready.d    = GND;
  END IF;

  /remote_ready.ena    = VCC;

  % reset the Rx_CRC %
  % next clock, Rx_CRC should see the first byte of valid data %
  /rx_crc_reset = GND;
  rx_state = rx_data0;

  %-------------------------------------------------------%
  % receiving data                                        %
  %-------------------------------------------------------%

WHEN rx_data0 =>
  % enable rx_rega to assure that valid data is latched %
  % (the input is latched at the next clock edge)       %
  rx_rega[7..0].ena = VCC;
  rx_state = rx_wait0;

WHEN rx_wait0 =>
  % test if the input is a JK, which indicates end of message %
  % if end of message, enter the crc sequence %
  IF (rcl & rcu & (endec2rx_[] == JK_BYTE)) THEN
      rx_state = rx_crc_jk1;
  ELSE
     rx_state = rx_data1;
  END IF;

WHEN rx_data1 =>
  rx_rega[15..8].ena = VCC;
  rx_state = rx_wait1;

WHEN rx_wait1 =>
  rx_state = rx_data2;

WHEN rx_data2 =>
  rx_rega[23..16].ena = VCC;
  rx_state = rx_wait2;

WHEN rx_wait2 =>
  rx_state = rx_data3;

WHEN rx_data3 =>
  rx_rega[31..24].ena = VCC;
  rx_state = rx_wait3;

WHEN rx_wait3 =>
  % shift the word just received into the rx_regb %
  % assert rxout_valid, since rx_regb will be valid the next cycle %
  rxout_valid.ena  = VCC;
  rxout_valid.d    = VCC;
  rx_regb_ena      = VCC;
  rx_state         = rx_data0;
```

```
  %-------------------------------------------------------%
  % CRC sequence                                          %
  %-------------------------------------------------------%
WHEN rx_crc_jk1 =>
    % the data should be another JK or an H_BYTE %
    % if it is H_BYTE, desert /remote_ready %
    IF (rcu & rcl & (endec2rx_[] == H_BYTE)) THEN
       /remote_ready.d = VCC;
    ELSE
       /remote_ready.d = GND;
    END IF;

    /remote_ready.ena = VCC;

    % bypass Rx_CRC, since the this byte wasn't checked by Tx_CRC %
    /rx_crc_hold = GND;
    rx_state = rx_crc_jk1_wait;

WHEN rx_crc_jk1_wait =>
    % bypass the crc %
    /rx_crc_hold = GND;
    rx_state = rx_crc0;

WHEN rx_crc0 =>
    rx_state = rx_crc0_wait;

WHEN rx_crc0_wait =>
    rx_state = rx_crc1;

WHEN rx_crc1 =>
    rx_state = rx_crc1_wait;

WHEN rx_crc1_wait =>
    rx_state = rx_crc2;

WHEN rx_crc2 =>
    rx_state = rx_crc2_wait;

WHEN rx_crc2_wait =>
    rx_state = rx_crc3;

WHEN rx_crc3 =>
    rx_state = rx_crc3_wait;

WHEN rx_crc3_wait =>
    rx_state = rx_crc4;

WHEN rx_crc4 =>
    rx_state = rx_crc4_wait;

WHEN rx_crc4_wait =>
    rx_state = rx_crc5;

WHEN rx_crc5 =>
    rx_state = rx_crc5_wait;

WHEN rx_crc5_wait =>
    rx_state = rx_done;

  %-------------------------------------------------------%
  % done                                                  %
  %-------------------------------------------------------%
WHEN rx_done =>
   % done with message reception, stop the Rx_CRC %
   /rx_crc_hold = GND;
   rx_state = rx_done_wait;

WHEN rx_done_wait =>
   % done with message reception, stop the Rx_CRC %
   /rx_crc_hold = GND;

   % check for error in the message %
   IF (rx_crc_zero_[] == CRC_CHECKED) THEN
      % there is NO error in the received message %
      % write the buffer ptr and length to Done FIFO %
      /wen_done = GND;
      rx_state = rx_idle;
   ELSIF ((next_[] == RX_CYCLE) # (next_[] == TX_CYCLE)) THEN
      % there are errors in the received message %
      % discard the message by return the buffer to Empty FIFO %
      % the Empty is also written by the SBus, %
      % so wait for a Rx or Tx cycle %
      /wen_empty = GND;
      rx_state = rx_idle;
   ELSE
       rx_state = rx_done;
   END IF;

END CASE;
```

```
%-------------------------------------------------------%
%-------------------------------------------------------%
% Logic                                                 %
%-------------------------------------------------------%
%-------------------------------------------------------%

%-------------------------------------------------------%
% SRAM address ptr and counter                          %
%   A[16..8] from sram_ptr       A[7..0] from counter     %
%-------------------------------------------------------%

  sram_ptr[].d = empty2rx_[];

  IF clr_sram_cnt THEN
      sram_cnt[] = 0;
  ELSIF inc_sram_addr THEN
     sram_cnt[] = sram_cnt[] + 1;
  ELSE
     sram_cnt[] = sram_cnt[];
  END IF;

  rx_sram_addr_[] = (sram_ptr[].q, sram_cnt[].q);

%-------------------------------------------------------%
% rx_rega, rx_regb, rx_out                              %
%-------------------------------------------------------%

  % shift register A (rega) content into register B %
  % assert rx_data_valid to allow the bux to read rxout_ %

  rx_rega[7..0].d   = endec2rx_[];
  rx_rega[15..8].d  = endec2rx_[];
  rx_rega[23..16].d = endec2rx_[];
  rx_rega[31..24].d = endec2rx_[];

  rx_regb[].d = rx_rega[].q;

  % tri-stated, since the SRAM data path is bidirectional %
  rx_tri[].oe = rx_tri_ena;
  rx_tri[].in = rx_regb[];
  rxout_[] = rx_tri[].out;

%-------------------------------------------------------%
% write the content of rx_regb to the SRAM              %
%-------------------------------------------------------%

% the phase is used to ensure %
% that the write is on the first half of an RX_CYCLE %
IF (rxout_valid & (phase == GND) & (next_[] == RX_CYCLE)) THEN

   % enable the path to sram%
   /rx_sram_we = GND;
   rx_tri_ena  = VCC;

   % dessert the data valid signal %
   rxout_valid.ena = VCC;
   rxout_valid.d   = GND;

   inc_sram_addr = VCC;
END IF;

END;
```

```
% This programmable device implements the SBUS FSM of the SBus Board %
% This is using single word transfer                                 %

TITLE "SBUS FINITE-STATE-MACHINE";

INCLUDE "const.inc";

DESIGN IS "sbusfsm"
BEGIN
    OPTIONS XOR_SYNTHESIS = ON;
    OPTIONS TURBO = ON;
    OPTIONS PARALLEL_EXPANDERS = ON;

   DEVICE "sbusfsm" IS "EPM7192QC160-2"
   begin
   end;
end;


  %-----------------------------------------%
  %          W:   BUS WIDTH [31..0]         %
  %-----------------------------------------%
CONSTANT W = 31;

SUBDESIGN sbusfsm
(
  %-----------------------------------------------------%
  % The clock of this fsm is running at 40 ns (sysclk) %
  %-----------------------------------------------------%

 sysclk             : INPUT; % system clock %
 /reset             : INPUT; % system reset %

 next_[1..0]        : INPUT; % SRAM cycle %
 phase              : INPUT; % current phase of SRAM cycle %
 PA_[20..17]        : INPUT; % PA_[27..21] are ignored, PA_[16..0] %
                             %  connected to the input of Full FIFO %
 /AS                : INPUT; % Address Strobe %
 /Sel               : INPUT; % SBus Slave Select %
 Rd                 : INPUT; % if VCC, it is a SBusRead, else it is a SBusWrite %

 /EF_Recycle        : INPUT; % empty flag of Recycle FIFO %
 /EF_Done           : INPUT; % empty flag of Done FIFO %
 done_out_[16..0]   : INPUT; % output of Done FIFO %
 recycle_out_[8..0] : INPUT; % output of Recycle FIFO %

 /IntReq_[7..6]     : OUTPUT; % Interrupt from board to sbus controller %
 /ACK_[2..0]        : OUTPUT; % Ack from SBus board to sbus controller %
 /sw_sram_WE        : OUTPUT; % SRAM write enable %
 /sr_sram_OE        : OUTPUT; % SRAM output enable %
 /wen_full          : OUTPUT; % write enable of Full FIFO %
 /wen_empty         : OUTPUT; % write enable of Empty FIFO %
 /ren_recycle       : OUTPUT; % read enable of Recycle FIFO %
 /ren_done          : OUTPUT; % read enable of Done Fifo    %
 D_[31..0]          : BIDIR;  % Bidir of data between SBus and Sram %
 port_sram_[W..0]   : BIDIR;  % SRAM data path %
)

VARIABLE

  %-----------------------------------------------------%
  % Register                                            %
  %-----------------------------------------------------%
 sram2sbus_reg[W..0]  : DFFE;
 sram2sbus_reg_ena    : node;
 clr_sram2sbus_reg    : node;

 sbus2sram_reg[W..0]  : DFFE;
 sbus2sram_reg_ena    : node;
 reg_sel_[1..0]       : node;

  %-----------------------------------------------------%
  % Tristate                                            %
  %-----------------------------------------------------%
 sram2sbus_t[W..0]    : TRI;
 sbus2sram_t[W..0]    : TRI;

 sram2sbus_t_oe       : node;
 sbus2sram_t_oe       : node;

  %-----------------------------------------------------%
  % FSM                                                 %
  %-----------------------------------------------------%
        s_state    : MACHINE of BITS (s[2..0])
                     WITH STATES (s_idle          = 0,
                                  s_write         = 1,
                                  s_read          = 2,
                                  s_read_wait0    = 3,
                                  s_Done          = 4,
                                  s_Recycle       = 5,
                                  s_ena           = 6,
                                  s_ack           = 7
                                 );
```

```
BEGIN

  %-------------------------------------------------------------%
  % Defaults Statements                                         %
  %-------------------------------------------------------------%

DEFAULTS
  /ACK_[]       = B"111";
  /sw_sram_WE   = VCC;
  /sr_sram_OE   = VCC;
  /wen_full     = VCC;
  /wen_empty    = VCC;
  /ren_recycle  = VCC;
  /ren_done     = VCC;
END DEFAULTS;

  sram2sbus_reg[].clk = global(sysclk);
  sram2sbus_reg[].ena = sram2sbus_reg_ena;
  sram2sbus_reg[].clrn= global(/reset);
  sram2sbus_reg[].prn = VCC;

  sbus2sram_reg[].clk = global(sysclk);
  sbus2sram_reg[].ena = sbus2sram_reg_ena;
  sbus2sram_reg[].clrn= !clr_sram2sbus_reg;
  sbus2sram_reg[].prn = VCC;

  s_state.clk = global(sysclk);
  s_state.reset = !global(/reset);

CASE (s_state) IS

WHEN s_idle =>

   % SBUS WRITE %
   % if address strobe asserted, end of msg not asserted %
   %   it is write to SBus slave cycle and transfer size is single word %
   % if yes, then SEL*, PA_[27..0] are also valid      %
   IF ((/AS == GND) & (/Sel == 0) & (PA_[20..17] == 0) & (Rd == 0)) THEN
        sbus2sram_reg_ena = VCC;
        s_state = s_write;
   END IF;

   % SBUS READ %
   IF ((/AS == GND) & (/Sel == 0) & (PA_[20..17] == 0) & (Rd == 1)) THEN
        clr_sram2sbus_reg = VCC;
        s_state = s_read;
   END IF;

%-------------------------------------------------------------------
  PA[17] == 1: end of a SBUS WRITE
-------------------------------------------------------------------%

   % END OF SBUS WRITE %
   % put the PA_[16..0] to FullBuf_Tx Fifo %
   IF ((/AS == GND) & (/Sel == 0) & (PA_[17] == 1)) THEN
       /wen_full = GND;
       s_state = s_ack;
   END IF;

%-------------------------------------------------------------------
  PA[18] when asserted indicates that the SBus controller has finished
   reading the buffer sent by the Receive logic.
-------------------------------------------------------------------%

   IF ((/AS == GND) & (/Sel == 0) & (PA_[18] == 1) & (phase == VCC) &
       ((next_[] == SRD_CYCLE) # (next_[] == SWR_CYCLE))) THEN
       /wen_empty = GND;
       s_state = s_ack;
   END IF;

   IF ((/AS == GND) & (/Sel == 0) & (PA_[18] == 1) &
       ((phase != VCC) #
        ((next_[] != SRD_CYCLE) & (next_[] != SWR_CYCLE)))) THEN
       s_state = s_idle;
   END IF;

%-------------------------------------------------------------------
  PA_[19] when asserted indicates that the SBus controller is servicing
   /IntReq_[7], i.e. reading done_fifo;
-------------------------------------------------------------------%

   IF ((/AS == GND) & (/Sel == 0) & (PA_[19] == 1)) THEN
        clr_sram2sbus_reg = VCC;
        /ren_done = GND;
        s_state = s_Done;
   END IF;
```

```
%-----------------------------------------------------------------
  PA_[20] when asserted indicates that the SBus controller is servicing
   /IntReq_[6], i.e. reading recycle_fifo
-----------------------------------------------------------------%

   IF ((/AS == GND) & (/Sel == 0) & (PA_[20] == 1)) THEN
        clr_sram2sbus_reg  = VCC;
        /ren_recycle       = GND;
        s_state = s_Recycle;
   END IF;

  %-----------------------------------------------------------%
  % Write from SBus to Sram                                   %
  %-----------------------------------------------------------%
WHEN s_write =>

   % write data D_[] from sbus to sram during SWR_CYCLE %
   IF ((next_[] == SWR_CYCLE) & (phase == GND)) THEN
       /sw_sram_WE        = GND;
       sbus2sram_t_oe     = VCC;
       s_state            = s_ack;
     ELSE
       s_state = s_write;
    END IF;

  %-----------------------------------------------------------%
  % Read from SBus to Sram                                    %
  %-----------------------------------------------------------%

WHEN s_read =>
   % wait for a SRD_CYCLE %
   IF ((next_[] == SRD_CYCLE) & (phase == GND))
     THEN
       /sr_sram_OE        = GND;
       s_state            = s_read_wait0;
     ELSE
       s_state            = s_read;
   END IF;

WHEN s_read_wait0 =>
   % deassert the /ACK lines and %
   % output the data corresponding to the /ACK for exactly one clock %

   /sr_sram_OE        = GND;
   reg_sel_[]         = 0;
   sram2sbus_reg_ena  = VCC;
   /ACK_[]            = B"011";
   s_state            = s_ena;

WHEN s_Done =>

   % assert the /ACK lines %
   % drive the data from Done FIFO onto D_[] the next cycle %
   reg_sel_[]         = 1;
   sram2sbus_reg_ena  = VCC;
   /ACK_[]            = B"011";
   s_state            = s_ena;

WHEN s_Recycle =>
   % assert the /ACK lines %
   % drive data from Recycle FIFO onto D_[] the next cycle %
   reg_sel_[]         = 3;
   sram2sbus_reg_ena  = VCC;
   /ACK_[]            = B"011";
   s_state            = s_ena;

WHEN s_ena =>
   % enable the tristate to output data on D_[] %
   sram2sbus_t_oe   = VCC;
   s_state          = s_idle;

WHEN s_ack =>
   % unack the ack %
   /ACK_[]          = B"011";
   s_state          = s_idle;

END CASE;
```

```
%-------------------------------------------------------%
%-------------------------------------------------------%
%  Logic                                                %
%-------------------------------------------------------%
%-------------------------------------------------------%

%-------------------------------------------------------%
%  Sram to SBus                                         %
%-------------------------------------------------------%

% select output of SRAM %
IF (reg_sel_[] == 0) THEN
   sram2sbus_reg[].d = port_sram_[];
END IF;

% select output of Done FIFO %
IF (reg_sel_[] == 1) THEN
   sram2sbus_reg[31..17].d = 0;
   sram2sbus_reg[16..0].d  = done_out_[16..0];
END IF;

% select output of Recycle FIFO %
IF (reg_sel_[] == 3) THEN
   sram2sbus_reg[31..9].d = 0;
   sram2sbus_reg[8..0].d  = recycle_out_[8..0];
END IF;

% tri-state path to D_[] %
sram2sbus_t[].in = sram2sbus_reg[].q;
sram2sbus_t[].oe = sram2sbus_t_oe;
D_[] = sram2sbus_t[].out;

%-------------------------------------------------------%
% SBus to Sram                                          %
%-------------------------------------------------------%
sbus2sram_reg[].d = D_[];

sbus2sram_t[].in = sbus2sram_reg[].q;
sbus2sram_t[].oe = sbus2sram_t_oe;
port_sram_[] = sbus2sram_t[].out;

%-------------------------------------------------------%
% Interrupt                                             %
%-------------------------------------------------------%

% If Done Fifo is not empty, then assert /IntReq_[7] %
% SBus Ctrl asserts PA_[19] when servicing the interrupt %
/IntReq_[7] = !/EF_Done;

% If Recycle Fifo is not empty, then assert /IntReq_[6] %
% SBus Ctrl asserts PA_[20] when servicing the interrupt %
/IntReq_[6] = !/EF_Recycle;

END;
```

```
% This programmable device implements the Transmit FSM of the SBus Board %

TITLE "TRANSMIT FINITE-STATE-MACHINE";

INCLUDE "const.inc";

DESIGN IS "txfsm"
BEGIN
    OPTIONS XOR_SYNTHESIS = ON;
    OPTIONS TURBO = ON;
    OPTIONS PARALLEL_EXPANDERS = ON;

   DEVICE "txfsm" IS "EPM7192QC160-2"
   begin
   end;
end;

SUBDESIGN txfsm
(
  %-------------------------------------------------------%
  % The clock of this fsm is running at 40 ns (sysclk)    %
  %-------------------------------------------------------%

 sysclk               : INPUT; % system clock %
 /reset               : INPUT; % system reset %

 /ef_full             : INPUT; % empty flag from Full FIFO %
 /ae_empty            : INPUT; % almost-empty flag from empty FIFO %
 full2tx_[16..0]      : INPUT; % output of Full FIFO %
 txin_[31..0]         : INPUT; % input from SRAM %
 next_[1..0]          : INPUT; % next sram cycle %
 /remote_ready        : INPUT; % asserted if remote end is ready for receiving message %

 tx_crc_out_[7..0]    : INPUT;  % output of Tx_CRC block %
 /tx_crc_reset        : OUTPUT; % Tx_CRC reset %
 /tx_crc_hold         : OUTPUT; % Tx_CRC hold, to stop Tx_CRC from reading input %
 tx_crc_sel_[2..0]    : OUTPUT; % select one of the six Tx_CRC bytes %

 /ren_full            : OUTPUT; % read enable for Full FIFO %
 /wen_recycle         : OUTPUT; % write enable for Recycle FIFO %

 tx2endec_[10..0]     : OUTPUT; % input to Tx_ENDEC %
 /tx_sram_oe          : OUTPUT; % enable the patch from sram to tx %
 tx_sram_addr_[16..0] : OUTPUT; % SRAM address %
)


VARIABLE

  %-------------------------------------------------------%
  % Registers                                             %
  %-------------------------------------------------------%
    txreg[31..0]    : DFFE;
    clr_txreg       : node;
    txreg_ena       : node;

  %-------------------------------------------------------%
  % Parity                                                %
  %-------------------------------------------------------%
    d[1..0]         : DFF;
    parity          : node;
    clr_d           : node;

%-------------------------------------------------------%
% Counters                                              %
%-------------------------------------------------------%
   ptr[8..0]    : DFFE;
   len[7..0]    : DFFE;
   clr_ptr_len  : node;
   ptr_len_ena  : node;

   idle_cnt[3..0] : DFF;
   inc_idle_cnt   : node;
   clr_idle_cnt   : node;

   sram_cnt[7..0] : DFF;
   inc_sram_addr  : node;
   clr_sram_cnt   : node;

%-------------------------------------------------------%
% ENDEC Input                                           %
%-------------------------------------------------------%
   tx2endec_[10..0] : DFFE;
   tx2endec_ena     : node;

%-------------------------------------------------------%
% FSM                                                   %
%-------------------------------------------------------%
      tx_state     : MACHINE of BITS (tx_[4..0])
                   WITH STATES (
                                 tx_init          = 0,
                                 tx_idle          = 1,
                                 tx_idlewait      = 2,

                                 tx_jk0           = 4,
                                 tx_jkwait0       = 5,
                                 tx_jk1           = 6,
                                 tx_jkwait1       = 7,
                                 tx_jk2           = 8,
                                 tx_jkwait2       = 9,
                                 tx_jk3           = 10,
                                 tx_jkwait3       = 11,

                                 tx_selB0         = 12,
                                 tx_sendB0        = 13,
                                 tx_selB1         = 14,
                                 tx_sendB1        = 15,
                                 tx_selB2         = 16,
                                 tx_sendB2        = 17,
                                 tx_selB3         = 18,
                                 tx_sendB3        = 19,
                                 tx_sendB3_end    = 20,

                                 tx_crc_jk0       = 21,
                                 tx_crc_jk0wait   = 22,
                                 tx_crc_jk1       = 23,
                                 tx_crc_jk1wait   = 24,


                                 tx_crc1          = 25,
                                 tx_crc1wait      = 26
                                );
```

```
BEGIN

  %-------------------------------------------------------%
  % Defaults Statements                                    %
  %-------------------------------------------------------%

DEFAULTS
   /tx_sram_oe        = VCC;
   /ren_full          = VCC;
   /wen_recycle       = VCC;
   /tx_crc_reset      = VCC;
   /tx_crc_hold       = VCC;

END DEFAULTS;

   d[].clk     = global(sysclk);
   d[].clrn    = !clr_d;

   ptr[].clk     = global(sysclk);
   ptr[].clrn    = !clr_ptr_len;
   ptr[].ena     = ptr_len_ena;

   len[].clk     = global(sysclk);
   len[].clrn    = !clr_ptr_len;
   len[].ena     = ptr_len_ena;

   idle_cnt[].clk     = global(sysclk);
   idle_cnt[].clrn    = global(/reset);

   tx2endec_[].clk     = global(sysclk);
   tx2endec_[].clrn    = global(/reset);
   tx2endec_[].prn     = VCC;
   tx2endec_[].ena     = tx2endec_ena;

   txreg[].clk     = global(sysclk);
   txreg[].clrn    = !clr_txreg;
   txreg[].prn     = VCC;
   txreg[].ena     = txreg_ena;

  %-------------------------------------------------------%
  % Signal Qualification                                   %
  %-------------------------------------------------------%
   %qualified with 20 ns delay clk%


tx_state.clk = global(sysclk);
tx_state.reset = !global(/reset);

CASE (tx_state) IS

  %-------------------------------------------------------%
  % Transmit at least 4 32-IDLE-bits over the Link        %
  %-------------------------------------------------------%

WHEN tx_init =>
   %-------------------------------------------------------%
   % dummy state to make sure jkwait2 falls on phase 0     %
   %-------------------------------------------------------%
   tx_state = tx_idle;

WHEN tx_idle =>
   clr_ptr_len  = VCC;
   clr_sram_cnt = VCC;
   clr_d        = VCC;
   clr_txreg    = VCC;

   % if Empty FIFO is almost empty, i.e. running out of buffer %
   % then send HH instead of IDLE %
   IF (/ae_empty == GND) THEN
     tx2endec_[] = HH;
   ELSE
     % select IDLE_BYTE %
     tx2endec_[] = IDLE;
   END IF;

   tx2endec_ena = VCC;
   tx_state = tx_idlewait;

WHEN tx_idlewait =>
   % SRAM data for Tx can only be read on a TX_CYCLE %
   % the test for next_[] ensures this %
   IF ((idle_cnt[].q == B"1111") & (/remote_ready == GND) &
       (/ef_full == VCC) & (next_[] == SRD_CYCLE)) THEN
       /ren_full = GND;
       tx_state = tx_jk0;
   END IF;

   IF ((idle_cnt[].q == B"1111") &
       ((/remote_ready != GND) # (/EF_FULL != VCC) #
        (next_[] != SRD_CYCLE))) THEN
       tx_state = tx_idle;
   END IF;

   IF (idle_cnt[].q != B"1111") THEN
       inc_idle_cnt = VCC;
       tx_state = tx_idle;
   END IF;

WHEN tx_jk0 =>
   % send a JK byte %
   tx2endec_[] = JK;
   tx2endec_ena = VCC;

   % latch data from Full FIFO enabled in tx_idlewait %
   ptr_len_ena = VCC;

   tx_state = tx_jkwait0;

WHEN tx_jkwait0 =>
   tx_state = tx_jk1;

WHEN tx_jk1 =>
   % send a ZERO_BYTE %
   tx2endec_[] = ZERO;
   tx2endec_ena = VCC;
   tx_state = tx_jkwait1;

WHEN tx_jkwait1 =>
   tx_state = tx_jk2;

WHEN tx_jk2 =>
   % send a ZERO_BYTE %
   tx2endec_[] = ZERO;
   tx2endec_ena = VCC;

   tx_state = tx_jkwait2;
```

```
WHEN tx_jkwait2 =>
   % output enable the SRAM %
   % data will be latched at the next cycle %
   /tx_sram_oe = GND;
   tx_state = tx_jk3;

WHEN tx_jk3 =>

   % latch the data from SRAM %
   txreg_ena   = VCC;

   % send HH if Empty FIFO is running low %
   IF (/ae_empty == GND) THEN
      tx2endec_[] = HH;
   ELSE
      tx2endec_[] = ZERO;
   END IF;

   tx2endec_ena = VCC;

   tx_state = tx_jkwait3;

WHEN tx_jkwait3 =>

   % compute the parity %
   % since the logic is not fast enough to compute %
   % the parity in one cycle, it is computed in two cycles %
   % that is why SRAM data is read two cycles ahead %
   d0 = txreg[7] $ txreg[6] $ txreg[5] $ txreg[4];
   d1 = txreg[3] $ txreg[2] $ txreg[1] $ txreg[0];

   tx_state = tx_selB0;

WHEN tx_selB0 =>
  % reset the Tx_CRC if sending data of a new message %
  IF (sram_cnt[] == 0) THEN /tx_crc_reset = GND; END IF;

  % select the first (lowest) data byte to send %
  parity = d0 $ d1 $ VCC;
  tx2endec_[] = (txreg[7..0], 0, 0, parity);
  tx2endec_ena = VCC;

  tx_state = tx_sendB0;

WHEN tx_sendB0 =>
  d0 = txreg[15] $ txreg[14] $ txreg[13] $ txreg[12];
  d1 = txreg[11] $ txreg[10] $ txreg[9] $ txreg[8];

  tx_state = tx_selB1;

WHEN tx_selB1 =>
  % select the second data byte to send %
  parity = d0 $ d1 $ VCC;
  tx2endec_[] = (txreg[15..8], 0, 0, parity);
  tx2endec_ena = VCC;
  tx_state = tx_sendB1;

WHEN tx_sendB1 =>
  inc_sram_addr = VCC;

  d0 = txreg[23] $ txreg[22] $ txreg[21] $ txreg[20];
  d1 = txreg[19] $ txreg[18] $ txreg[17] $ txreg[16];

  tx_state = tx_selB2;

WHEN tx_selB2 =>
  % select the third data byte to send %
  parity = d0 $ d1 $ VCC;
  tx2endec_[] = (txreg[23..16], 0, 0, parity);
  tx2endec_ena = VCC;

  tx_state = tx_sendB2;

WHEN tx_sendB2 =>
  d0 = txreg[31] $ txreg[30] $ txreg[29] $ txreg[28];
  d1 = txreg[27] $ txreg[26] $ txreg[25] $ txreg[24];

  IF (len[] != sram_cnt[]) THEN
     /tx_sram_oe  = GND;
  END IF;
  tx_state = tx_selB3;

WHEN tx_selB3 =>
  % select the fourth data byte to send %
  parity = d0 $ d1 $ VCC;
  tx2endec_[] = (txreg[31..24], 0, 0, parity);
  tx2endec_ena = VCC;

  IF (len[] != sram_cnt[]) THEN
     txreg_ena    = VCC;
     tx_state = tx_sendB3;
  ELSE
     tx_state = tx_sendB3_end;
  END IF;

WHEN tx_sendB3 =>

  d0 = txreg[7] $ txreg[6] $ txreg[5] $ txreg[4];
  d1 = txreg[3] $ txreg[2] $ txreg[1] $ txreg[0];

  tx_state = tx_selB0;

WHEN tx_sendB3_end =>
  % clear the idle_cnt since we want to use it as a counter %
  % enter crc send sequence %
  clr_idle_cnt = VCC;
  % done with the buffer ptr %
  /wen_recycle  = GND;

  tx_state = tx_crc_jk0;
  %-------------------------------------------------------%
  % CRC seq is JK JK crc0 crc1 crc2 crc3 crc4 crc5        %
  %-------------------------------------------------------%
WHEN tx_crc_jk0 =>
  % send a JK %
  tx2endec_[] = JK;
  tx2endec_ena = VCC;
  tx_state = tx_crc_jk0wait;

WHEN tx_crc_jk0wait =>
  tx_state = tx_crc_jk1;

WHEN tx_crc_jk1 =>
  IF (/ae_empty == GND) THEN
     tx2endec_[] = HH;
  ELSE
     tx2endec_[] = JK;
  END IF;
```

```
  tx2endec_ena = VCC;
  tx_state = tx_crc_jk1wait;

WHEN tx_crc_jk1wait =>

  % bypass the CRC for the second JK (or HH) %
  % because there is no time to compute the CRC parities otherwise %
  /tx_crc_hold = GND;
  d0 = tx_crc_out_[7] $ tx_crc_out_[6] $ tx_crc_out_[5] $ tx_crc_out_[4];
  d1 = tx_crc_out_[3] $ tx_crc_out_[2] $ tx_crc_out_[1] $ tx_crc_out_[0];

  inc_idle_cnt = VCC;
  tx_state = tx_crc1;

WHEN tx_crc1 =>

  % send 6 cycle_byte %
  % idle_cnt equals number of crc seq that have been sent %

  % stop the Tx_CRC from reading the data %
  /tx_crc_hold = GND;
  parity = d0 $ d1 $ VCC;
  tx2endec_[] = (tx_crc_out_[], 0, 0, parity);
  tx2endec_ena = VCC;

  tx_state = tx_crc1wait;

WHEN tx_crc1wait =>

  /tx_crc_hold = GND;
  d0 = tx_crc_out_[7] $ tx_crc_out_[6] $ tx_crc_out_[5] $ tx_crc_out_[4];
  d1 = tx_crc_out_[3] $ tx_crc_out_[2] $ tx_crc_out_[1] $ tx_crc_out_[0];

  IF (idle_cnt[] == 6) THEN
     clr_idle_cnt = VCC;
     tx_state = tx_idle;
  ELSE
     inc_idle_cnt = VCC;
     tx_state = tx_crc1;
  END IF;

END CASE;
```

```
%-------------------------------------------------------------%
%-------------------------------------------------------------%
% LOGIC                                                       %
%-------------------------------------------------------------%
%-------------------------------------------------------------%

%-------------------------------------------------------------%
% SRAM address counter                                        %
%   A[16..8] from ptr            A[7..0] from len             %
%-------------------------------------------------------------%
  sram_cnt[].clk  = global(sysclk);
  sram_cnt[].clrn = !clr_sram_cnt;

  IF inc_sram_addr THEN
     sram_cnt[].d = sram_cnt[].q + 1;
  ELSE
     sram_cnt[].d = sram_cnt[].q;
  END IF;

  tx_sram_addr_[] = (ptr[], sram_cnt[]);

%-------------------------------------------------------------%
% Full FIFO output, containing the ptr and length             %

%-------------------------------------------------------------%
  ptr[].d = full2tx_[16..8];
  len[].d = full2tx_[7..0];

%-------------------------------------------------------------%
% Idle Counter                                                %
%-------------------------------------------------------------%
  IF clr_idle_cnt     THEN idle_cnt[].d = 0;
  ELSIF inc_idle_cnt  THEN idle_cnt[].d = idle_cnt[].q + 1;
  ELSE                     idle_cnt[].d = idle_cnt[].q;
  END IF;


%-------------------------------------------------------------%
% register for SRAM input                                     %
%-------------------------------------------------------------%
  txreg[] = txin_[];

%-------------------------------------------------------------%
% Tx_CRC, selecting one of the six CRC bytes                  %
%-------------------------------------------------------------%
  tx_crc_sel_[] = idle_cnt[2..0];

END;
```

```
`timescale 100 ps / 100 ps

/* Verilog model for the SRAM, IDT7M4013, a 128K x 32-bit memory cell */

module sram(clk, reset_l, io, addr, we_l, oe_l);
  input clk, reset_l;
  inout [31:0] io;   // 32-bit i/o line
  input [16:0] addr; // 17-bit addr line
  input we_l, oe_l;

  `include "params.vh"

  parameter
   LINE_WIDTH     = 32,     // 32 bit-wide
   N_LINES        = 131072, // 128K
   TOE            = 150,    // 15 ns
   TOHZ           = 150,    // 15 ns
   TWHZ           = 170;    // 17 ns


  reg [LINE_WIDTH - 1 : 0] sram[N_LINES - 1 : 0];
  reg [LINE_WIDTH - 1 : 0] out;

  tri [LINE_WIDTH - 1 : 0] io = (oe_l == 0) ? out : `bz;

  always @(negedge oe_l)
    // read cycle
    begin
      out = `bz;
      #(TOE); // OE to output valid;
        // out = sram[addr];
        out = sram[addr];
        $display("sram[%h] --> %h", addr, out);
    end

  always @(posedge oe_l)
    // disable the output data
    begin
      #(TOHZ); // OE to Z
      out = `bz;
    end

  always @(negedge we_l)
    begin
    #(TWHZ); // for io line to change to data
      if (we_l == 0)
         begin // WRITE_CYCLE
           // wait half cycle for data to get ready
           // we need to make sure the data hold valid for at least 19 ns
           // #(half_cycle);
           $display("sram[%h] <-- %h", addr, io);
           sram[addr] = io;
         end
    end

endmodule
```

```
`timescale 100 ps / 100 ps

/* Verilog model for the SBus controller */

module sbus(clk, reset_l, as_l, sel_l, rd, pa, d, ack_l, intreq_l);

  `include "params.vh"

  input  clk, reset_l;
  output as_l, sel_l;
  output rd;
  output [27:0] pa;
  input  [31:0] d;
  input  [2:0]  ack_l;
  input  [7:6]  intreq_l;

  reg [8:0] buf_in;
  wire [8:0] buf_out;
  wire ef_buf_l, ff_buf_l, af_buf_l, ae_buf_l;
  reg wen_buf_l, ren_buf_l;

  fifo  buf_fifo(clk, reset_l, buf_in, buf_out,
                 ren_buf_l, wen_buf_l, ef_buf_l,
                 ff_buf_l, af_buf_l, ae_buf_l);

  integer len, i;

  defparam
       buf_fifo.id = "    buf",
       buf_fifo.width = 9,
       buf_fifo.preloadi = 0,
       buf_fifo.preloadn = 128;

  reg as_l_r, sel_l_r, rd_r;
  reg [27:0] pa_r;
  reg [31:0] out_r;
  reg oe;
  tri[31:0] d = (oe == 1) ? out_r : 'bz;

  reg   [31:0] data;
  reg   write_cycle, read_cycle;

  assign as_l = as_l_r;
  assign sel_l = sel_l_r;
  assign rd = rd_r;
  assign pa = pa_r;

  always @(negedge reset_l)
     begin
       as_l_r  = 1;
       sel_l_r = 1;
       rd_r    = 0;
       pa_r    = 28'b0;
       out_r   = 32'b0;
       ren_buf_l = 1;
       wen_buf_l = 1;
       read_cycle = 0;
       write_cycle = 1;
       oe          = 0;
       read_cycle  = 0;
       write_cycle  = 1;
    end

always @(posedge clk)
  begin
    #(full_cycle * ($random & 'o77));
    // write cycle
    wait(write_cycle == 1);
    $display("************ START WRITE ************");
    ren_buf_l = 0;
    len = ($random & 'hf);
    #(full_cycle);
    ren_buf_l = 1;
    pa_r[27:17] <= 11'b0;
    pa_r[16:8] <= buf_out;
    $display("message len %h", len);
    for(i = 0; i < len; i = i + 1)
       begin
         oe = 1;
         pa_r[7:0] <= i;
         rd_r <= 0;
         sel_l_r <= 0;
         as_l_r  <= 0;
         out_r <= $random;
         wait(ack_l == 3'b011);
         $display("sbus[%h] --> %h", pa_r[16:0], out_r);
         #(full_cycle);
         oe      <= 0;
         sel_l_r <= 1;
         as_l_r  <= 1;
         #(full_cycle);
       end
    pa_r[27:18] <= 11'b0;
    pa_r[17]  <= 1;
    pa_r[7:0] <= i;
    rd_r <= 0;
    sel_l_r <= 0;
    as_l_r  <= 0;
    out_r <= 32'b1;
    #(full_cycle);
    wait(ack_l == 3'b011);
    sel_l_r <= 1;
    as_l_r  <= 1;
    #(full_cycle);
    write_cycle = 0;
    read_cycle  = 1;
    $display("************ END WRITE ************");
  end
```

```
always @(posedge clk)
  // handling intreq_l[7]
  if (intreq_l[7] == 0)
    begin
        wait(read_cycle == 1);
        $display("************ START READ ************");

        // read buffer ptr and len
        as_l_r   <= 0;
        sel_l_r  <= 0;
        rd_r     <= 1;
        pa_r     = 0;
        pa_r[19]  = 1;
        wait(ack_l == 3'b011);
        // one clock for ack cycle, plus
        // half a cycle for data to be valid
        #(full_cycle + half_cycle);
        as_l_r   <= 1;
        sel_l_r  <= 1;
        pa_r[16:8] <= d[16:8];
        len <= d[7:0];
        pa_r[27:17] <= 11'b0;
        #(full_cycle);
        $display("sbus -- %h:%h", pa_r[16:8], len);

        // read buffer
        for(i=0; i < len; i = i + 1)
           begin
             pa_r[7:0] <= i;
             rd_r    <= 1;
             sel_l_r <= 0;
             as_l_r  <= 0;
             wait(ack_l == 3'b011);
             #(full_cycle + half_cycle);
             data    <= d;
             sel_l_r <= 1;
             as_l_r  <= 1;
             #(full_cycle);
             $display("sbus[%h] <-- %h", pa_r[16:0], data);
           end

        // return the ptr to empty fifo
        pa_r[27:19] = 0;
        pa_r[18]    = 1;
        pa_r[17]    = 0;
        rd_r     <= 0;
        sel_l_r  <= 0;
        as_l_r   <= 0;
        wait(ack_l == 3'b011);
        #(full_cycle);
        sel_l_r <= 1;
        as_l_r  <= 1;
        #(full_cycle);
        $display("************ END READ ************");

        write_cycle = 1;
        read_cycle  = 0;
    end

always @(posedge clk)
  // handling intreq_l[6]
  if ((intreq_l[6] == 0)  && (intreq_l[7] != 0))
     begin
       wait(read_cycle == 1);

       pa_r     = 0;
       pa_r[20] = 1;
       rd_r     <= 1;
       as_l_r   <= 0;
       sel_l_r  <= 0;
       wait(ack_l == 3'b011);

       #(full_cycle + half_cycle);
       buf_in <= d[7:0];
       sel_l_r <= 1;
       as_l_r  <= 1;

       #(full_cycle);
       wen_buf_l = 0;
       #(full_cycle);
       wen_buf_l = 1;
       read_cycle = 0;
       write_cycle = 1;
     end

endmodule
```

```
`timescale 100 ps / 100 ps

/* A partial model of the FDDI Encoder and Decoder (ENDEC), AM7984A.
 * There is a 2-cycle latency between inputs and outputs. */

module endec(sysclk, reset_l, in, out, s, rcl, rcu, parity);

`include "params.vh"

input  sysclk, reset_l;
input  [10:0] in;
output [7:0] out;
output [2:0]  s;
output rcl, rcu, parity;

integer idle_cnt;
reg     [2:0] sr;
reg     [10:0] tmp1, tmp0;

assign parity = tmp1[0];
assign rcl    = tmp1[1];
assign rcu    = tmp1[2];
assign out    = tmp1[10:3];
assign s      = sr;

always @(posedge sysclk)
  if (reset_l == 0)
     begin
       sr = LSU;
       idle_cnt = 0;
       tmp1 = 11'b0;
       tmp0 = 11'b0;
     end

always @(posedge sysclk)
  if ((reset_l != 0))
     begin
       tmp1 = tmp0;
       tmp0 = in;
       if (tmp1[1] && tmp1[2])
          begin
            if (tmp1[10:3] == IDLE_BYTE)
               begin
                 if (idle_cnt >= 16) sr = ILS;
                 idle_cnt = idle_cnt + 1;
               end
            if (tmp1[10:3] == JK_BYTE)
               begin
                 sr = ALS;
                 idle_cnt = 0;
               end
          end

        if ((tmp1[1] == 0) || (tmp1[2] == 0)) idle_cnt = 0;
     end

endmodule
```

```
`timescale 100 ps / 100 ps

/* Verilog model for the FIFO, IDT72201, a 256 x 9-bit memory cell */

module fifo(clk, reset_l, in, out, ren_l, wen_l, ef_l, ff_l, af_l, ae_l);
  `include "params.vh"

   parameter
      width     = 17,
      depth     = 256,
      offset    = 7,
      empty_mark = offset,
      full_mark  = (depth - offset),
      preloadn  = 0,
      preloadi  = 0,
      id        = "";

   input  clk, reset_l;
   input  [width -1 :0] in;
   output [width -1 :0] out;
   input  ren_l, wen_l;
   output ef_l, ff_l, af_l, ae_l;

   reg [width-1 :0] mem[depth-1:0];
   reg [width-1 :0] out_r;
   reg              ef_l_r, ff_l_r, af_l_r, ae_l_r;

   integer head, tail, len;

   always @(negedge reset_l)
   begin
     if (preloadn == 0)
        begin
          len = 0;
          head = 0;
          tail = 0;
          ef_l_r = 0;
          ff_l_r = 1;
          af_l_r = 1;
        end
     if (preloadn != 0)
        begin
          len = preloadn;
          head = 0;
          ef_l_r = 1;
          for(tail = 0; tail < preloadn; tail = ((tail + 1) % depth))
             mem[tail] = preloadi + tail;
          ff_l_r = (len == depth) ? 0 : 1;
          af_l_r = (len >= full_mark) ? 0 : 1;
          ae_l_r = (len < empty_mark) ? 0 : 1;
        end
      $display("reseting %s_fifo [%d][%d]: (%d, %d)",
                id, depth, width, preloadi, preloadn);
   end

   assign ef_l = ef_l_r;
   assign ff_l = ff_l_r;
   assign af_l = af_l_r;
   assign ae_l = ae_l_r;
   assign out  = out_r;

   always @(negedge ren_l)
     // read from fifo
     begin
       if (ef_l_r !== 0)
          begin
            $display("%s_fifo[%h] --> %h", id, head, mem[head]);
            out_r = #(80) mem[head];
            head = (head + 1) % depth;
            len = len - 1;
            if (len == 0)          ef_l_r <= #(100) 0; // 10 ns
            if (len < empty_mark) ae_l_r <= #(100) 0; // 10 ns
            if (len < full_mark)  af_l_r <= #(80) 1;  // 8 ns
            ff_l_r <= #(80) 1; // 8 ns
          end
     end

   always @(negedge wen_l)
     // write to fifo
     if (ff_l_r !== 0)
        begin
          $display("%s_fifo[%h] <-- %h", id, tail, in);
          mem[tail] = in;
          tail = (tail + 1) % depth;
          len = len + 1;
          if (len == depth)       ff_l_r <= #(100) 0; // 10 ns
          if (len >= full_mark)   af_l_r <= #(100) 0; // 10 ns
          if (len >= empty_mark) ae_l_r <= #(80) 1;  // 8 ns
          ef_l_r <= #(80) 1; // 8 ns
        end

endmodule
```

```
'timescale 100 ps / 100 ps

/* Top level driver for the SBus board simulation */

module driver;

  `include "params.vh"

  reg reset_l;
  reg sysclk;

  // crc
  wire tx_crc_reset_l, tx_crc_hold_l, rx_crc_reset_l, rx_crc_hold_l;
  wire [7:0] tx_crc_out, rx_crc_out;
  wire [2:0] tx_crc_sel;
  reg  [2:0] rx_crc_sel;
  wire [3:0] tx_crc_zero, rx_crc_zero;

  // fifo
  wire [16:0] full_in;
  wire [16:0] full_out;
  wire [16:0] full2tx;
  wire wen_full_l;
  wire ren_full_l;
  wire ef_full_l, ff_full_l, af_full_l, ae_full_l;

  wire [16:0] done_in;
  wire [16:0] done_out;
  wire [16:0] rx2done;
  wire wen_done_l, ren_done_l;
  wire ef_done_l, ff_done_l, af_done_l, ae_done_l;

  wire [8:0] empty_in;
  wire [8:0] empty_out;
  wire [8:0] empty2rx;
  wire wen_empty_l, ren_empty_l;
  wire ef_empty_l, ff_empty_l, af_empty_l, ae_empty_l;

  wire [8:0] recycle_in;
  wire [8:0] recycle_out;
  wire wen_recycle_l, ren_recycle_l;
  wire ef_recycle_l, ff_recycle_l, af_recycle_l, ae_recycle_l;

  // sram
  wire sram_we_l, sram_oe_l;
  wire [31:0] sram_io;
  wire [16:0] sram_addr;

  wire [16:0] tx_sram_addr, rx_sram_addr;

  wire tx_sram_oe_l, rx_sram_we_l;
  wire sw_sram_we_l, sr_sram_oe_l;

  // SRAM cycle
  wire [1:0] next;
  wire       phase;

  // endec
  wire [10:0] tx2endec;
  wire tx2endec_ena;

  wire [7:0] endec2rx;
  wire [7:0] tx2crc;
  wire [2:0] s;

  wire rcl, rcu, parity;

  // SBus
  wire rf_tri, sbus_tri0, sbus_tri1;

  wire [27:0] pa;
  wire [2:0] ack_l;
  wire as_l;
  wire [31:0] d;
  wire [7:6] intreq_l;
  wire rd;
  wire sel_l;

  wire remote_ready;

  txfsm tf(ae_empty_l,ef_full_l,full2tx,next,remote_ready,
           ren_full_l,reset_l,sysclk,
           tx2endec,
           tx_crc_hold_l,tx_crc_out,tx_crc_reset_l,tx_crc_sel,
           tx_sram_addr,tx_sram_oe_l,sram_io,wen_recycle_l);

  rxfsm rf(rf_tri,rf_tri,
           empty2rx,endec2rx,next,phase,
           rcl,rcu,remote_ready,ren_empty_l,reset_l,
           rx_crc_hold_l,rx_crc_reset_l,rx_crc_zero,
           rx_sram_addr,rx_sram_we_l,
           sram_io,s,sysclk,
           wen_done_l,rx_wen_empty_l);

  nxtfsm nf(empty_in,
            next,phase,reset_l,
            rx_sram_addr,rx_sram_we_l,rx_wen_empty_l,
            pa[16:0],sb_wen_empty_l,sr_sram_oe_l,
            sram_addr,sram_oe_l,sram_we_l,
            sw_sram_we_l,sysclk,tx_sram_addr,tx_sram_oe_l,wen_empty_l);

  sbus sbctrl(sysclk, reset_l, as_l, sel_l, rd, pa, d, ack_l,
              intreq_l);

  sbusfsm sf(sbus_tri0,sbus_tri1,sbus_tri1,
             ack_l,as_l,d,
             done_out,ef_done_l,ef_recycle_l,intreq_l,
             next,pa[20:17],phase,sram_io,rd,recycle_out,
             ren_done_l,ren_recycle_l,
             reset_l,sel_l,
             sr_sram_oe_l,
             sw_sram_we_l,sysclk,sb_wen_empty_l,wen_full_l);

  crc12 tx_crc(tx_crc_reset_l, sysclk,
               tx2endec[10:3], tx_crc_out, tx_crc_sel,
               tx_crc_hold_l,  tx_crc_reset_l, tx_crc_zero);

  crc12 rx_crc(rx_crc_reset_l, sysclk, endec2rx, rx_crc_out, rx_crc_sel,
               rx_crc_hold_l, rx_crc_reset_l, rx_crc_zero);

  fifo  full_fifo(sysclk, reset_l, pa[16:0], full2tx,
                  ren_full_l, wen_full_l, ef_full_l,
                  ff_full_l, af_full_l, ae_full_l);

  fifo  done_fifo(sysclk, reset_l, rx_sram_addr, done_out,
                  ren_done_l, wen_done_l, ef_done_l,
                  ff_done_l, af_done_l, ae_done_l);
```

```
  fifo  empty_fifo(sysclk, reset_l, empty_in, empty2rx,
                   ren_empty_l, wen_empty_l,
                   ef_empty_l, ff_empty_l, af_empty_l, ae_empty_l);

  fifo  recycle_fifo(sysclk, reset_l, tx_sram_addr[16:8], recycle_out,
                     ren_recycle_l, wen_recycle_l, ef_recycle_l,
                     ff_recycle_l, af_recycle_l, ae_recycle_l);

  defparam
     empty_fifo.depth = 16,
     empty_fifo.width = 9,
     empty_fifo.preloadi = 128,
     empty_fifo.preloadn = 7,
     recycle_fifo.width = 9;

  defparam
     empty_fifo.id       = "  empty",
     full_fifo.id        = "   full",
     done_fifo.id        = "   done",
     recycle_fifo.id     = "recycle";


  sram s1(sysclk, reset_l, sram_io, sram_addr, sram_we_l, sram_oe_l);

  endec ed(sysclk, reset_l, tx2endec, endec2rx, s, rcl, rcu, parity);

  initial
    begin
      $gr_waves(

"sysclk",           sysclk,
"next  %d",         next,
"phase",            phase,
"tx    %d",         (tf.tx_4, tf.tx_3, tf.tx_2, tf.tx_1,tf.tx_0),
"tx_sram_addr %h", tx_sram_addr,
"idlecnt%h",        (tf.idle_cnt3,tf.idle_cnt2,tf.idle_cnt1,tf.idle_cnt0),
"txreg  %h",        (tf.txreg31, tf.txreg30, tf.txreg29,
                     tf.txreg28, tf.txreg27, tf.txreg26,
                     tf.txreg25, tf.txreg24, tf.txreg23,
                     tf.txreg22, tf.txreg21, tf.txreg20,
                     tf.txreg19, tf.txreg18, tf.txreg17,
                     tf.txreg16, tf.txreg15, tf.txreg14,
                     tf.txreg13, tf.txreg12, tf.txreg11,
                     tf.txreg10, tf.txreg9, tf.txreg8,
                     tf.txreg7, tf.txreg6, tf.txreg5,
                     tf.txreg4, tf.txreg3, tf.txreg2,
                     tf.txreg1, tf.txreg0),
"tx2crc %h",        tx2crc,
"tx2endec %h",      (tx2endec[10:3], 1'b0, tx2endec[2:0]),
"tx2endec_ena",     tx2endec_ena,

"rx    %d",         (rf.rx4, rf.rx3, rf.rx2, rf.rx1,rf.rx0),
"remote_ready",     remote_ready,
"line_status %b",   s,
"endec2rx %h",      endec2rx,
"rx_sram_addr %h", rx_sram_addr,
"rx_reg %h",        (rf.rx_regb31, rf.rx_regb30, rf.rx_regb29,
                     rf.rx_regb28, rf.rx_regb27, rf.rx_regb26,
                     rf.rx_regb25, rf.rx_regb24, rf.rx_regb23,
                     rf.rx_regb22, rf.rx_regb21, rf.rx_regb20,
                     rf.rx_regb19, rf.rx_regb18, rf.rx_regb17,
                     rf.rx_regb16, rf.rx_regb15, rf.rx_regb14,
                     rf.rx_regb13, rf.rx_regb12, rf.rx_regb11,
                     rf.rx_regb10, rf.rx_regb9, rf.rx_regb8,
                     rf.rx_regb7, rf.rx_regb6, rf.rx_regb5,
                     rf.rx_regb4, rf.rx_regb3, rf.rx_regb2,
                     rf.rx_regb1, rf.rx_regb0),

"Sbus %d",          (sf.s2, sf.s1, sf.s0),
"pa[16:0] %h",      pa[16:0],     // sbus address bits
"pa[20:17]",        pa[20:17],    // sbus control bits
"ack_l   %b",       ack_l,        // ack
"intreq_l %b",      intreq_l,     // interrupt request
"as_l",             as_l,         // address strobe
"d     %h",         d,            // sbus data lines
"sram2sbus %h",     (sf.sram2sbus_reg31, sf.sram2sbus_reg30, sf.sram2sbus_reg29,
                     sf.sram2sbus_reg28, sf.sram2sbus_reg27, sf.sram2sbus_reg26,
                     sf.sram2sbus_reg25, sf.sram2sbus_reg24, sf.sram2sbus_reg23,
                     sf.sram2sbus_reg22, sf.sram2sbus_reg21, sf.sram2sbus_reg20,
                     sf.sram2sbus_reg19, sf.sram2sbus_reg18, sf.sram2sbus_reg17,
                     sf.sram2sbus_reg16, sf.sram2sbus_reg15, sf.sram2sbus_reg14,
                     sf.sram2sbus_reg13, sf.sram2sbus_reg12, sf.sram2sbus_reg11,
                     sf.sram2sbus_reg10, sf.sram2sbus_reg9, sf.sram2sbus_reg8,
                     sf.sram2sbus_reg7, sf.sram2sbus_reg6, sf.sram2sbus_reg5,
                     sf.sram2sbus_reg4, sf.sram2sbus_reg3, sf.sram2sbus_reg2,
                     sf.sram2sbus_reg1, sf.sram2sbus_reg0),

// sram
"sram_addr %h",     sram_addr,
"sram_io   %h",     sram_io,
"sram_oe_l",        sram_oe_l,
"sram_we_l",        sram_we_l,
"oe_tx",            tx_sram_oe_l,
"we_rx",            rx_sram_we_l,
"we_sw",            sw_sram_we_l,
"oe_sr",            sr_sram_oe_l,

// fifo's
"done_r",       ren_done_l,
"done_w",       wen_done_l,
"done_ef",      ef_done_l,
"done_af",      af_done_l,
"done_ae",      ae_done_l,
"done_in",      rx_sram_addr,  // done is written from Rx
"done_out",     done_out,

"recycle_r",       ren_recycle_l,
"recycle_w",       wen_recycle_l,
"recycle_ef",      ef_recycle_l,
"recycle_af",      af_recycle_l,
"recycle_ae",      ae_recycle_l,
"recycle_in",      tx_sram_addr[16:8], // recycle is written from Tx
"recycle_out",     done_out,

"full_r",       ren_full_l,
"full_w",       wen_full_l,
"full_ef",      ef_full_l,
"full_af",      af_full_l,
"full_ae",      ae_full_l,
"full_in",      pa[16:0], // full is written from Sbus
"full_out",     full_out,

"empty_r",       ren_empty_l,
"empty_w",       wen_empty_l,
"empty_ef",      ef_empty_l,
"empty_af",      af_empty_l,
"empty_ae",      ae_empty_l,
```

```
"empty_in",      empty_in,  // empty is written by both Sbus and Rx
"empty_out",     empty_out,

// crc
"tx_crc",        tx_crc.crc,
"tx_reset",      tx_crc_reset_l,
"hold",          tx_crc_hold_l,
"sel  %d",       tx_crc_sel,
"out  %h",       tx_crc_out,
"zero %h",       tx_crc_zero,

"c0      %b",    (tx_crc.c0_11, tx_crc.c0_10, tx_crc.c0_9, tx_crc.c0_8,
                  tx_crc.c0_7, tx_crc.c0_6, tx_crc.c0_5, tx_crc.c0_4,
                  tx_crc.c0_3, tx_crc.c0_2, tx_crc.c0_1, tx_crc.c0_0),

"c1      %b",    (tx_crc.c1_11, tx_crc.c1_10, tx_crc.c1_9, tx_crc.c1_8,
                  tx_crc.c1_7, tx_crc.c1_6, tx_crc.c1_5, tx_crc.c1_4,
                  tx_crc.c1_3, tx_crc.c1_2, tx_crc.c1_1, tx_crc.c1_0),

"c2      %b",    (tx_crc.c2_11, tx_crc.c2_10, tx_crc.c2_9, tx_crc.c2_8,
                  tx_crc.c2_7, tx_crc.c2_6, tx_crc.c2_5, tx_crc.c2_4,
                  tx_crc.c2_3, tx_crc.c2_2, tx_crc.c2_1, tx_crc.c2_0),

"c3      %b",    (tx_crc.c3_11, tx_crc.c3_10, tx_crc.c3_9, tx_crc.c3_8,
                  tx_crc.c3_7, tx_crc.c3_6, tx_crc.c3_5, tx_crc.c3_4,
                  tx_crc.c3_3, tx_crc.c3_2, tx_crc.c3_1, tx_crc.c3_0),

"rx_crc",        rx_crc.crc,
"rx_reset",      rx_crc_reset_l,
"hold",          rx_crc_hold_l,
"out  %h",       rx_crc_out,
"zero  %h",      rx_crc_zero, // equals 0 when the CRC is checked
);
    end

  // set up the sys clock
  initial
    begin
      sysclk = 0;
      forever
        begin
          #(half_cycle);
          sysclk = !sysclk;
        end
    end

  // reset the fsm
  initial
    begin
      reset_l = 0;
      #(full_cycle * 2);
      reset_l = 1;
    end

  initial
    begin
      // for completeness
      rx_crc_sel    = 0;
    end
endmodule
```

"Now to Him who is able to do immeasurably more than all we ask or imagine, according to His power that is at work within us, to Him be glory in the church and in Christ Jesus throughout all generations, for ever and ever! Amen."

Ephesians 3:20-21