Serializing High Speed Parallel Buses to Extend Their Operational Length

Introduction

Parallel buses are used in many designs for the purpose of moving data from one point to another. VME, ISA, EISA, VESA, PCI, SBus, and NuBus are some of the more familiar bus architectures. These buses are usually configured with a single bus master and multiple users, all communicating over a shared set of address and data lines. Some bus architectures, however, involve only two nodes on the bus, creating a point-to-point data link. Regardless of the architecture, the trend in bus design is for higher bandwidth achieved by increasing the width and transfer rate of the bus. When wide, high-speed, parallel buses are operated over distances of more than a couple of feet, problems can result. The source of these problems relates to the high-frequency signals interfering with each other over the long parallel conductors of the bus. This application note uses the UTOPIA bus as an example of how to serialize a high speed parallel point-to-point bus in order to allow the bus to operate over any distance.

The topics covered in this application note are as follows:

1. The UTOPIA Bus
2. UTOPIA Applications
3. Problems with Parallel Buses
4. The Serial Solution
5. Serial Links and HOTLink™
6. Serializing the UTOPIA Bus
7. Round Trip Latency
8. The UTOPIA Extender
9. Conclusions

The UTOPIA Bus

A good example of a high speed point-to-point parallel bus is the Universal Test and Operations Physical Interface for ATM (or UTOPIA). UTOPIA is used in ATM (or Asynchronous Transfer Mode) applications. ATM is a network protocol that has grown out of the need for a worldwide standard to allow interoperability of information, regardless of the "end-system" or type of information. With ATM, the goal is one international standard.

ATM is a method of communication which can be used as the basis for both LAN and WAN technologies. When information needs to be communicated, the sender negotiates a "requested path" with the network for a connection to the destination. When setting up this connection, the sender specifies the type, speed, and other attributes of the call, which determine the quality of service. Thus ATM is a switch-based technology (see *Figure 1*). By providing connectivity through a switch (instead of a shared bus) ATM delivers several benefits including dedicated bandwidth per connection, higher aggregate bandwidth, well-defined connection procedures, and flexible access speeds.

Using ATM, information to be sent is segmented into a fixed-length cell, transported to and reassembled at the destination. The ATM cell has a fixed length of 53 bytes. Being fixed-length allows different traffic types on the same network. The cell itself is broken into two main sections, the header and the payload. The payload (48 bytes) is the portion that

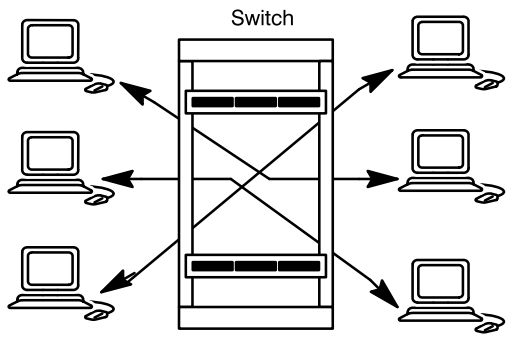


Figure 1. ATM Connections Through Switch

carries the actual information—either voice, data, or video. The Header (5 bytes) is the addressing mechanism (see *Figure 2*).

ATM closely follows the International Standards Organization's (ISO) Open Systems Interconnection (OSI) model for communication. This model breaks down any communication process into several sub processes arranged in a stack (see *Figure 3*).

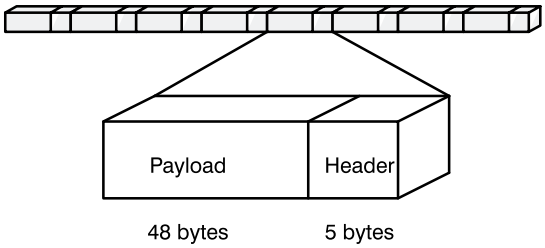


Figure 2. ATM Cell Format

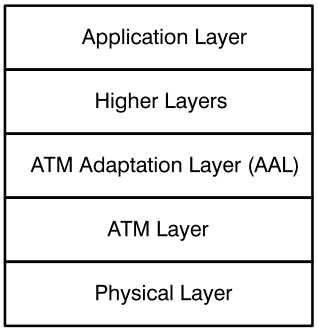


Figure 3. ATM Protocol Stack

Each layer of the "protocol stack" provides services to the layer above that allow the top most processes to communicate. The idea is that two different devices, using hardware and software from different vendors, but still conforming to the model, can communicate over an ATM network. The layers of the protocol stack can be thought of as modules in software code. Each layer performs a specific function and must provide data to other layers according to a specified interface. However, how that layer accomplishes its task is immaterial. Thus, layers in the stack can be updated without affecting the communication model.

The UTOPIA bus is a standard defined by the ATM forum for moving data between the physical (or PHY) and Asynchronous Transfer Mode (or ATM) layers in the ATM protocol stack. The PHY layer interfaces directly to the network media (i.e., fiber, twisted pair, etc.) and also handles "transmission convergence" (that is, extracting the ATM cells from the transport coding scheme). The ATM layer processes the cell headers and directs routing. The signals used by the UTOPIA bus are shown in *Figure 2* and described in *Table 1*.

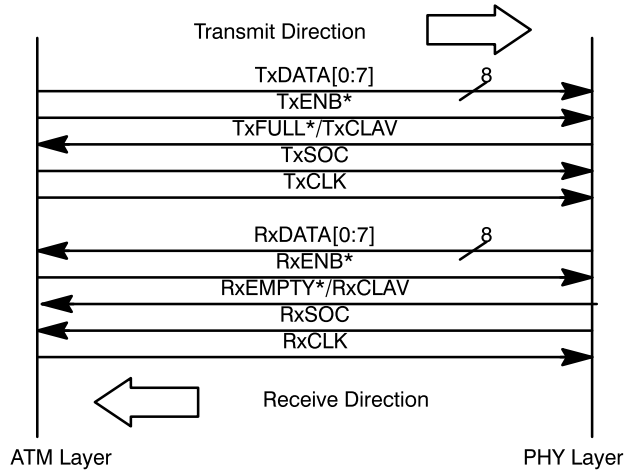


Figure 4. UTOPIA Signals

Table 1. UTOPIA Signals

Signal Name	Description
TxDATA[0:7]	Data lines for transmit (from ATM to PHY layer)
TxENB*	Indicates data on this cycle is valid
TxFULL*	Indicates Tx FIFO on PHY layer can only accept 4 more bytes (used only in Octet Level Handshaking)
TxCLAV	Indicates Tx FIFO on PHY layer is capable of storing an entire cell
TxSOC	Indicates data on this clock cycle is the start of a cell
TxCLK	Clock for Tx signals and data
RxDATA[0:7]	Data lines for receive (from PHY to ATM layer)
RxENB*	Indicates data on this cycle is valid
RxEMPTY*	Indicates Rx FIFO on PHY layer is empty (used only in Octet Level Handshaking)
RxCLAV	Indicates Rx FIFO on PHY layer is currently storing an entire cell
RxSOC	Indicates data on this clock cycle is the start of a cell
RxCLK	Clock for Rx signals and data

UTOPIA Applications

The UTOPIA bus is present in any ATM system that makes use of the ATM and PHY layers. Typical applications utilizing UTOPIA include Network Interface Cards and ATM switches. The ATM switch application for UTOPIA is of particular interest. Many switches are built using a rack mounted architecture as shown in *Figure 5*.

In this type of switch, individual shelves of the rack are dedicated to PHY layer circuits, and others to ATM layer circuits. Thus the UTOPIA bus is used to move the data between the different shelves of the switch. Usually, the interconnect between the shelves is a simple multi-conductor ribbon cable. Since the shelves can be fairly far apart, the ribbon cable required to connect the shelves can be anywhere from 1 to 6 feet in length.

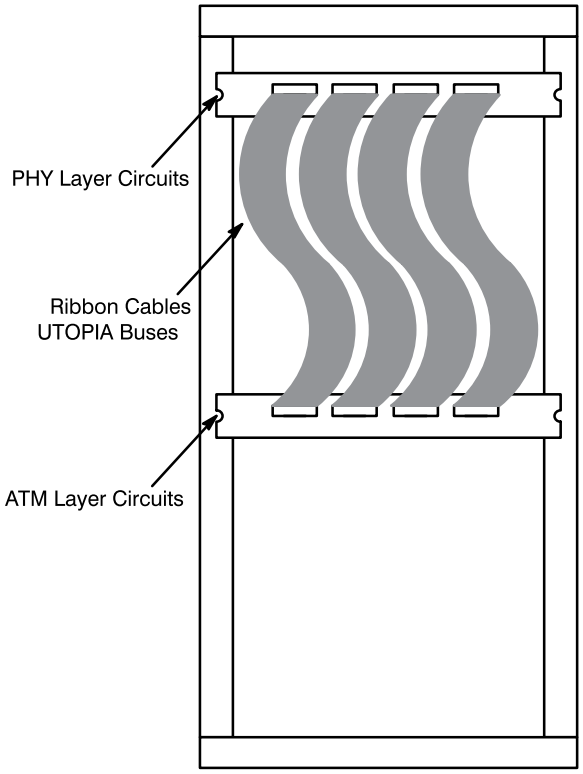


Figure 5. UTOPIA in a Rack Mount Switch

Problems with Parallel Buses

The difficulty with the use of ribbon cable for the UTOPIA switch application is related to the width and bandwidth requirements of the bus, combined with the uncontrolled impedance of the ribbon cable. These three characteristics can lead to skew across the signals of the UTOPIA bus as shown in *Figure 6*.

Note the skew shown in *Figure 6* has violated the set-up and/or hold times of the UTOPIA bus at the load end. Therefore, data communication over the bus will be corrupted. This effect is typical when high-speed parallel buses are driven over long distances. One possible solution is to drive each line of the bus differentially, but this also has the disadvantage of increasing the already bulky ribbon cable, and it is not guaranteed to solve the skew problem (skew can

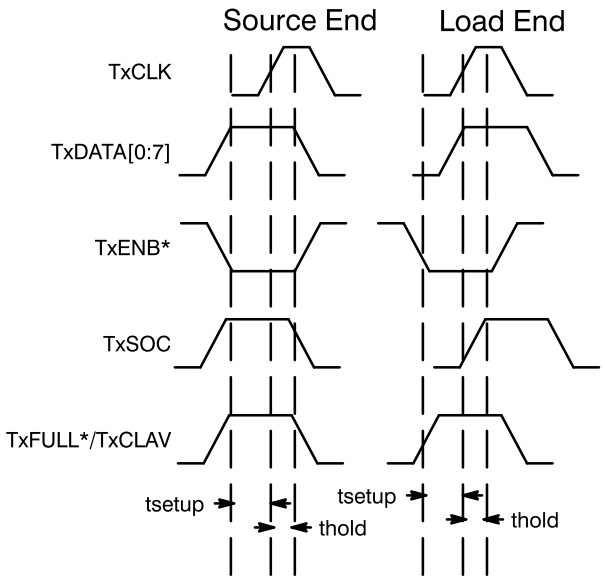


Figure 6. Effect of Skew on UTOPIA Bus

still result from differences in propagation delays for each signal through its respective differential driver/cable/receiver).

The Serial Solution

A good solution to the skew problems described above is to transmit the parallel bus data as a serial data stream. Transmitting the data serially requires a parallel-to-serial conversion of the UTOPIA data at the source end and a corresponding serial-to-parallel conversion at the load end. With such a scheme, the skew problems associated with operating a high-speed parallel bus over long distances are eliminated. In addition, the cable size is reduced from a multi-conductor ribbon cable to a two-conductor serial cable (such as coaxial cable).

The method by which a serial data transfer eliminates the skew problems associated with parallel buses is related to how serial links operate. Although some "serial" communication systems utilize more than one conductor (e.g., RS232), more serial links provide for transmission of only one signal. Note that to transmit one signal over copper media requires two conductors. This transmission can be either single-ended (requiring one conductor for the signal and one reference or ground) or differential (requiring two conductors for one signal). Both clock and data information must be included in this single signal. To accomplish this clock and data multiplexing function, serial links make use of special encoding schemes and use clock recovery circuits. The clock recovery circuits rely on the special characteristics of the data encoding scheme in order to recover or generate a clock of the same frequency and phase (with respect to the serial data) as the clock used to shift the data onto the serial link. The serial-to-parallel converter then uses this recovered clock to resample or retime the serial data before placing this data into a parallel word register. When this register is full, the serial-to-parallel converter presents the data in the register (in a parallel format) along with a parallel word clock (generated by dividing down the recovered serial clock). Thus, there is no skew between the clock and parallel data.

The main advantages of a serial link over a parallel bus are: (1) the clock is embedded with data, thus there is no skew between clock and data signals, (2) the distance over which the serial link is operated can be changed and the link will remain operational, (3) the transfer rate of the serial link can be scaled up and the link will remain operational, and (4) the cables required are smaller in size.

Serial Links and HOTLink™

The Cypress HOTLink™ chipset performs all of the functions shown in the simplified block diagram in *Figure 7*. The CY7B923 HOTLink Transmitter serves as the serializer while the CY7B933 HOTLink Receiver operates as a deserializer. In the HOTLink chipset, clock multiplication and clock recovery are accomplished using Phase Locked Loops (or PLLs). PLLs are closed loop control systems which align an output waveform in phase and frequency with an input waveform. Block diagrams of PLLs performing clock multiplication and clock recovery are shown in *Figure 8*.

PLLs operate by constantly comparing their output waveform with their input (or reference) waveform. Deviations in phase or frequency are then corrected at a rate governed by the Low Pass Filter (LPF). A wide bandwidth LPF allows a PLL to track high-frequency phase deviations between the reference and the output waveforms. A narrow bandwidth LPF dictates that the PLL rejects high-frequency phase

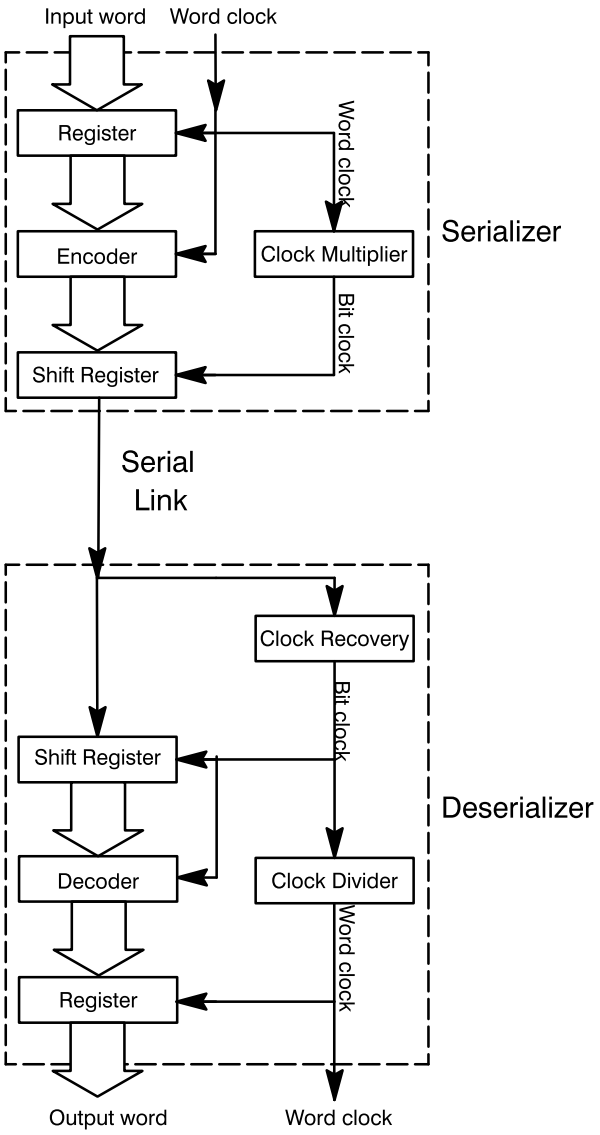


Figure 7. Architecture of a Serial Link

deviations between the reference and output waveform. Ideally, an input waveform would have a transition at a regular periodic rate, thus allowing the PLL to check its alignment constantly. However, such a signal would contain no information (essentially the link would be composed of one baseband frequency and its harmonics) and is not useful for data communication. Actual serial streams do not have data transitions at strictly periodic intervals. Instead, there are often "runs" of consecutive ones or zeros, which result in short periods where the serial stream has no transitions. The lack of transitions in the serial stream can cause the clock recovery PLL to fall out of phase lock, and eventually out

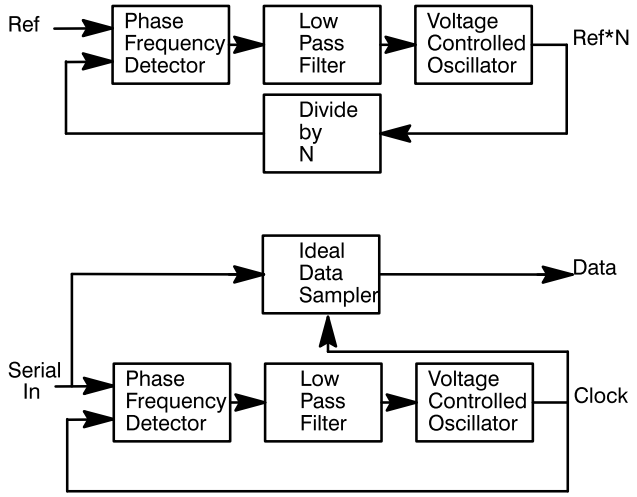


Figure 8. Multiplication and Clock/Data Recovery PLLs

of frequency lock. In order to reliably perform clock recovery with PLLs, the serial data needs to be encoded in such a way as to ensure there are frequent transitions (either from HIGH to LOW or LOW to HIGH) in the serial stream. These transitions cannot be ensured when sending unencoded data, since a user is free to send any data pattern. Some serial patterns like 00000000 contain no transitions and therefore could be transmited indefinitely resulting in a serial link without any transitions.

The HOTLink chipset utilizes an encoding scheme known as 8B/10B. This code takes in a 8-bit data word and converts it into a 10-bit transmission character. The transmission characters are chosen such that their run length is limited to 5 consecutive ones or zeros. With this encoding scheme, the HOTLink Receiver's clock recovery circuit can maintain lock and recover the clock from the serial data stream.

Serializing the UTOPIA Bus

Operating the UTOPIA bus over a serial link is accomplished using the architecture shown in *Figure 9*.

The basic block functions are as follows: On the ATM side, the serializer converts the parallel UTOPIA transmit data into a serial stream, embedding the UTOPIA transmit clock with the data. The deserializer converts the serial receive stream (from the PHY layer) back into parallel data and a receive clock. The First In First Out (FIFO) memory works

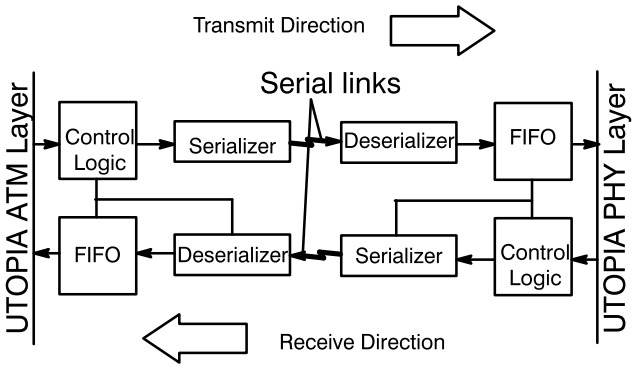


Figure 9. UTOPIA Serializer Block Diagram

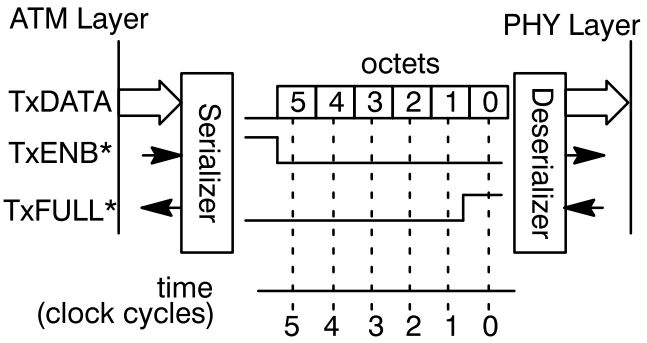


Figure 10. Round Trip Latency Example

as an elastic buffer, queuing the parallel receive data until the ATM layer parallel interface is ready to accept the data. The control logic provides control for all of the blocks. On the PHY side, the blocks perform similar functions. The serializer converts the parallel receive data into a serial stream, embedding the UTOPIA receive clock into the data. The deserializer converts the serial transmit stream (from the ATM layer) back into parallel data and a transmit clock. The FIFO provides buffering for the transmit interface, and the control logic manages all of the blocks.

Round Trip Latency

The purpose of the FIFO in the serialized UTOPIA architecture is to account for latency in the system. To understand the importance of the FIFO, consider a design which implemented a serialized UTOPIA bus. For UTOPIA transmits, there are two handshaking signals TX_FULL* (sourced at the PHY layer) and TX_ENB* (sourced at the ATM load). A transfer is initiated when TX_FULL* goes HIGH, followed by TX_ENB* going LOW and the UTOPIA data placed onto the bus. If TX_FULL* should go LOW at any time, the transfer must stop (according to the UTOPIA specification) within four write cycles. However, since TX_FULL* is sourced at the PHY layer and sampled at the ATM layer, there is a time delay for any change of state of TX_FULL* at the PHY layer to be recognized at the ATM layer. *Figure 10* shows an example of the timing relationships of the critical UTOPIA signals. This time delay is the latency through the serializer, serial media, and deserializer. There is a similar latency with respect to the TX_ENB* and TX_DATA from the ATM layer to the PHY layer. A problem arises if a transfer is in progress and TX_FULL* goes LOW. The figure shows that the transfer began successfully and several octets were placed onto the serial link. However, at clock cycle 1, the TX_FULL* signal on the PHY side went LOW, indicating that the PHY layer is full. According to the UTOPIA specification, the transfer must stop (TX_ENB* must go HIGH) within four byte times of TX_FULL* going LOW. In order for TX_ENB* to go HIGH, the ATM layer must recognize the change in state of TX_FULL*, but there is a delay from the PHY layer to the ATM layer. During this delay, the ATM layer may have already sent out too many bytes (in *Figure 10* five bytes are shown as being transmitted before TX_FULL* is recognized at the ATM layer). Since it is possible to not recognize the change in state of TX_FULL* within the four byte specification, there is the potential for data loss at the PHY layer.

Note that the latency in the link that is the source of the problem in the above example is *not* entirely due to the serializer and deserializer. If the serial link itself is long enough, the mere time delay required for the electrical pulses to travel down the link may be enough to cause the problems described above.

The latency issue is solved by buffering the data coming out of the deserializer. A FIFO is an adequate buffer for this application. With the FIFO buffer, the effects of the link latency are corrected. When the PHY layer UTOPIA interface indicates it has no more room for data, the FIFO can store the octets that are sent by the ATM layer before it receives the TX_FULL* signal. The data can then be

read out of the FIFO when the PHY layer UTOPIA interface is ready.

The UTOPIA Extender

Following the block diagram shown in *Figure 9*, and the hierarchical schematics shown in Appendix A, a serialized UTOPIA bus can be implemented. With the bus serialized, it can essentially be extended to any length, thus the design results in a "UTOPIA extender." The major components required to implement such a design are shown in *Table 2*.

Table 2. Cypress UTOPIA Extender Components

Generic Part	Cypress Part
Serializer	CY7B923 HOTLink Tx
Deserializer	CY7B933 HOTLink Rx
FIFO	CY7B451 512x9 clocked FIFO
Control Logic	CY7C371 32-macrocell Flash PLD

The "Top Level" hierarchical schematic shows a generic breakdown of the entire design. The "ATM Layer UTOPIA Extender" block implements all of the functions at the ATM layer interface necessary to serialize the UTOPIA bus. Likewise, the "PHY Layer UTOPIA Extender" block implements all of the functions at the PHY layer interface. Between these two blocks are two serial links over which the serialized UTOPIA bus operates. A system level application of the UTOPIA Extender is shown in *Figure 11*.

Both the "ATM" and "PHY Layer UTOPIA Extender" blocks have additional hierarchical schematics associated with them. Within these lower-level hierarchical schematics are additional blocks that show more detail than the previous levels. Each block performs a specific function necessary for the operation of the entire design. Some functions are common to both the "ATM" and "PHY Layer UTOPIA Extender" blocks, such as the "Media Interface" block. The "Media Interface" block performs the function of interfacing the transmit and receive electrical signals (comprising the serial links carrying the serialized UTOPIA bus) to the specific media interface used in the design (in this case to co-

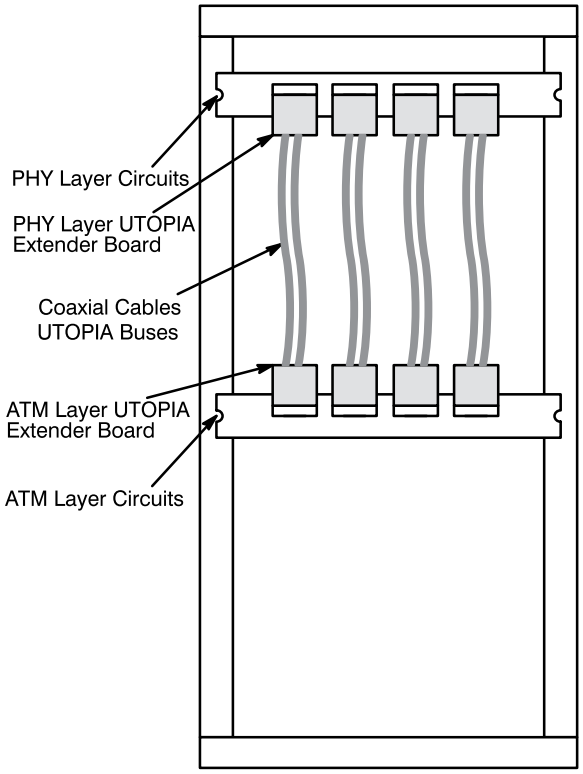


Figure 11. UTOPIA Extender in a Rack Mount Switch

axial cable). The "Media Interface" schematic contains termination networks and transformers used to interface the transmit and receive serial signals to the coaxial cable.

The "ATM" and "PHY UTOPIA Logic" blocks contain all of the circuits used to serialize the UTOPIA bus. These blocks contain the serializers, deserializers, FIFOs, and PLDs used to implement the logic for the UTOPIA extender.

The operation of the UTOPIA extender, implemented in the "ATM" and "PHY UTOPIA Logic" blocks, can be broken down into two modes. The first mode, or Steady State mode, moves the UTOPIA transmit and receive data between the ATM and PHY layers, and handles generation of the necessary control signals. The second mode, or FIFO State Update mode, handles the control of the buffering FIFOs assuring that no data is lost due to overfilling of these buffers. This mode also handles the case of the CLAV signals going inactive, indicating the UTOPIA interface cannot accept more

data. Regardless of the mode of operation, the basic link operation revolves around the Cell Level Handshaking (or CLH) protocol.

The main characteristic of CLH is that once a cell transmission begins, all 53 octets of the cell are sent in succession on consecutive clocks. In this mode, back to back cell transmissions are also possible. For this design, however, back to back cell transmissions will not be allowed (this is accomplished through special considerations in the UTOPIA control logic). A gap will be forced between all cells. This gap serves two purposes. The first is to allow for the communication of the CLAV control codes from the PHY layer to the ATM layer and also to update the status of the buffering FIFOs. The second reason for the gap is to allow for easy generation of the SOC signal at the load end of the serial link.

The Steady State mode of operation for the UTOPIA extender is defined as the condition when neither buffer FIFO is overfilled. When in this mode, there is a minimal amount of control logic necessary to implement the extender. As an example, consider a UTOPIA transmit (defined as data movement from the ATM to the PHY layer). When a 53-octet cell becomes available on the ATM layer side, it is immediately placed into the HOTLink transmitter and sent over to the PHY side. Following the first octet, the remaining 52 octets of the cell are sent consecutively. Following transmission of the 53rd byte, the link pauses to implement the forced cell gap. During this pause, the HOTLink Transmitter is disabled and sends idle characters (called K28.5 or "Commas") across the link. If there is another cell available from the ATM layer, it is sent across after the cell gap. If no data is available, the link remains disabled. The flow of data under the steady state mode is shown in *Figure 12*.

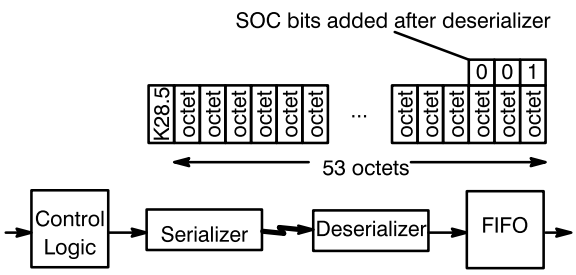


Figure 12. Transmission Data Flow

Upon receiving the octets from the ATM layer, the output of the HOTLink Receiver is immediately placed into the buffering FIFO. In addition, when the first octet out of the receiver is sensed (by taking advantage of the forced gap between cells), an additional bit, serving as the TX_SOC signal, is placed into the FIFO coincident with the first octet. The remaining 52 octets are also placed into the FIFO, but without the TX_SOC bit set. The TX_ENB* signal to the UTOPIA interface is then generated from the TX_CLAV signal and the FIFO status signals. The PHY UTOPIA interface directly reads the output of the buffering FIFO. Data movement in the UTOPIA receive direction is similar.

The other mode of operation is FIFO State Updating. This mode basically serves to handle the case when the CLAV signals change state. That is, if the TX_CLAV is deasserted, no data will be read out of the PHY side buffering FIFO. Eventually, this FIFO will fill beyond a check point and a code will be sent back to the ATM layer side indicating no more data should be sent until the FIFO is read beyond a certain level. The operation of this mode requires some additional control logic. Again, consider the case of UTOPIA transmission. A FIFO state update begins when the control logic on the PHY layer side detects that the buffering FIFO has filled beyond a predefined level. The control logic then waits for a pause in the data stream going back to the ATM layer side (remember a gap is forced between successive cells). During this pause, the control logic inserts a "FIFO Full" control code into the HOTLink transmitter in place of one of the comma characters (see *Figure 13*). This FIFO Full code travels across the link back to the ATM layer side. The ATM layer control logic then interprets the FIFO Full code and deasserts the TX_CLAV signal at the ATM layer UTOPIA interface, thus stopping transmission on the next cell boundary.

Eventually, the PHY layer FIFO will empty past another predefined level, thus indicating data transmission can begin again. The control logic on the PHY layer side then waits for a pause in the data stream back to the ATM layer side, and inserts a "FIFO Not Full" code in place of one of the comma characters (see *Figure 14*). This code travels down the link back to the ATM layer side where it is inter-

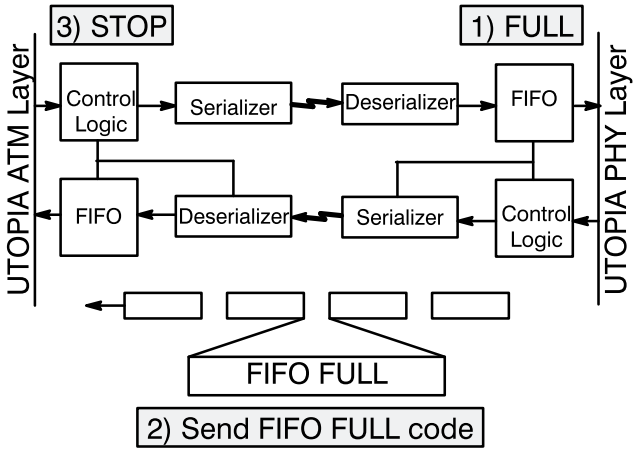


Figure 13. FIFO State Updating, FIFO Full

preted by the ATM layer control logic. The control logic then asserts the TX_CLAV signal to the ATM layer UTOPIA interface allowing data transmission to resume. Operation then reverts back to the Steady State mode.

The remaining blocks in the UTOPIA Extender ("ATM UTOPIA and Processor Interface," "PHY UTOPIA and Processor Interface," and "Framer Processor Interface") are used to interface the "ATM" and "PHY UTOPIA Logic" blocks to the UTOPIA bus of the ATM and PHY Layer Circuits as shown in *Figure 11*. In general, these remaining blocks contain connectors with pinouts specific to the particular ATM/PHY layer circuits used in the system. In addition, some ATM and/or PHY layer circuits require additional circuits to configure and/or monitor their operation. Thus the actual design of the "ATM UTOPIA and Processor Interface," "PHY UTOPIA and Processor Interface," and "Framer Processor Interface" blocks differs depending on the unique ATM and PHY layer circuits used in the system.

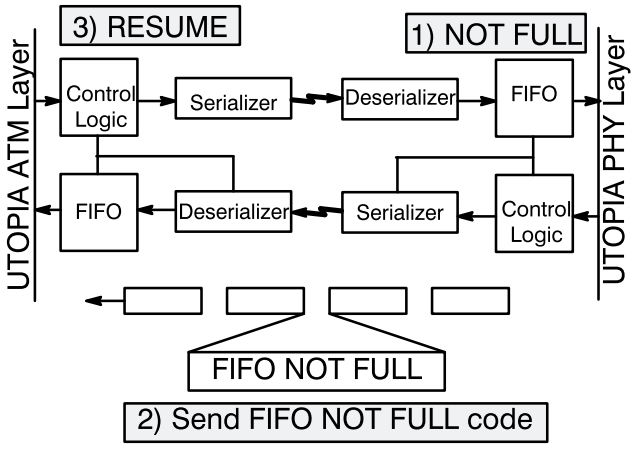


Figure 14. FIFO State Updating, FIFO Not Full

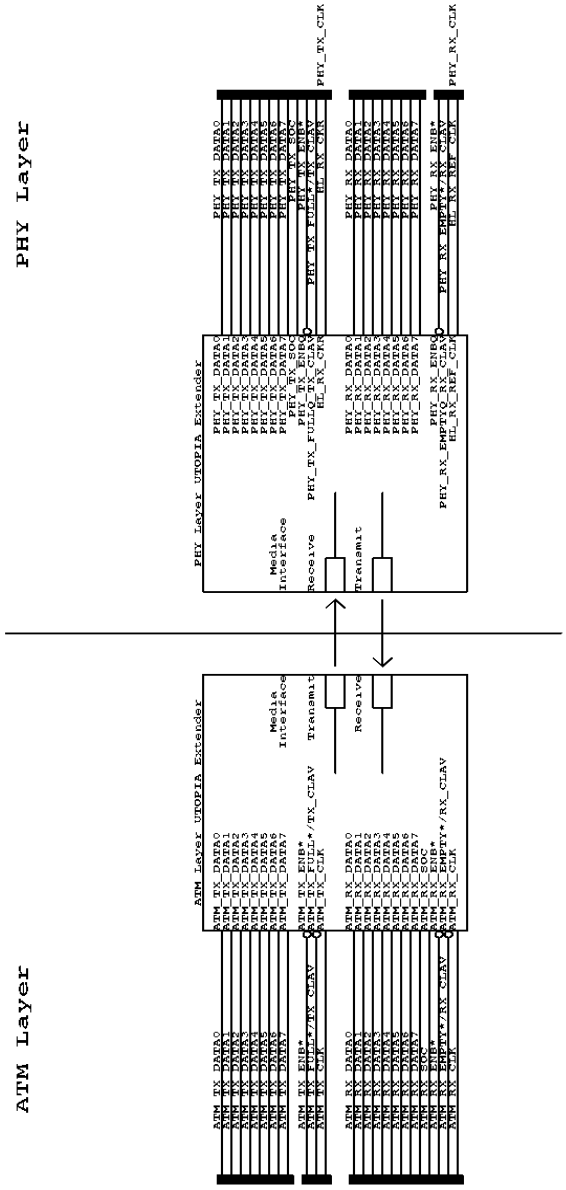
To exemplify a system using the UTOPIA Extender, a complete design of the PHY Layer is shown in the schematics (that is, only the "PHY Layer UTOPIA Extender" is shown fully implemented). The PHY Layer Circuit used was a Duke Communications DC-202® SONET/ATM UNI Transceiver Module. Thus the "PHY UTOPIA and Processor Interface" block was tailored to interface to the DC-202. In addition, the "Framer and Processor Interface" block was required to configure the DC-202 for proper operation. VHDL code for the "Framer and Processor Interface Block" is included in Appendix B. Also included in Appendix B is VHDL code implementing the algorithms for the "PHY UTOPIA Logic" PLD.

Conclusions

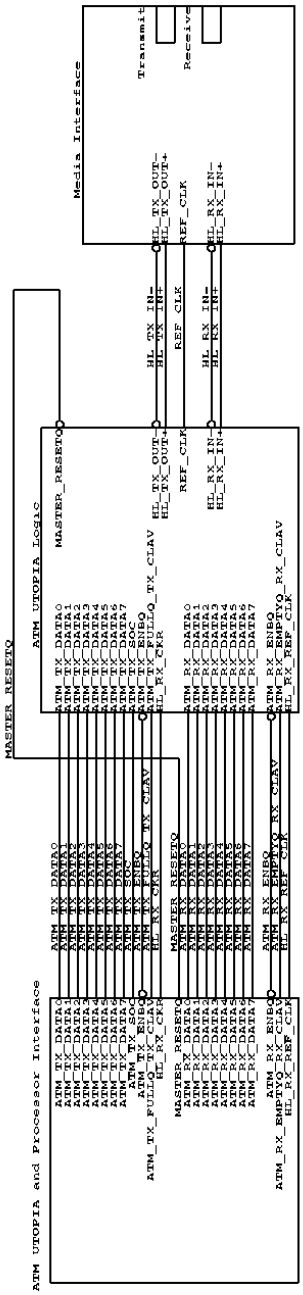
This application note has shown that signal skew across a ribbon cable can limit the operational distance of high-speed parallel buses such as UTOPIA. Serial links can operate over longer distances since they are not susceptible to the skew effects that limit parallel buses. This application note describes the design of a serialized parallel bus called the "UTOPIA Extender." Implementation of the UTOPIA Extender requires only a minimal amount of logic, with most of the work being performed by a high-speed serial-link chipset such as the Cypress HOTLink chipset.

HOTLink is a trademark of Cypress Semiconductor.
DC-202 is a registered trademark of Duke Communications.

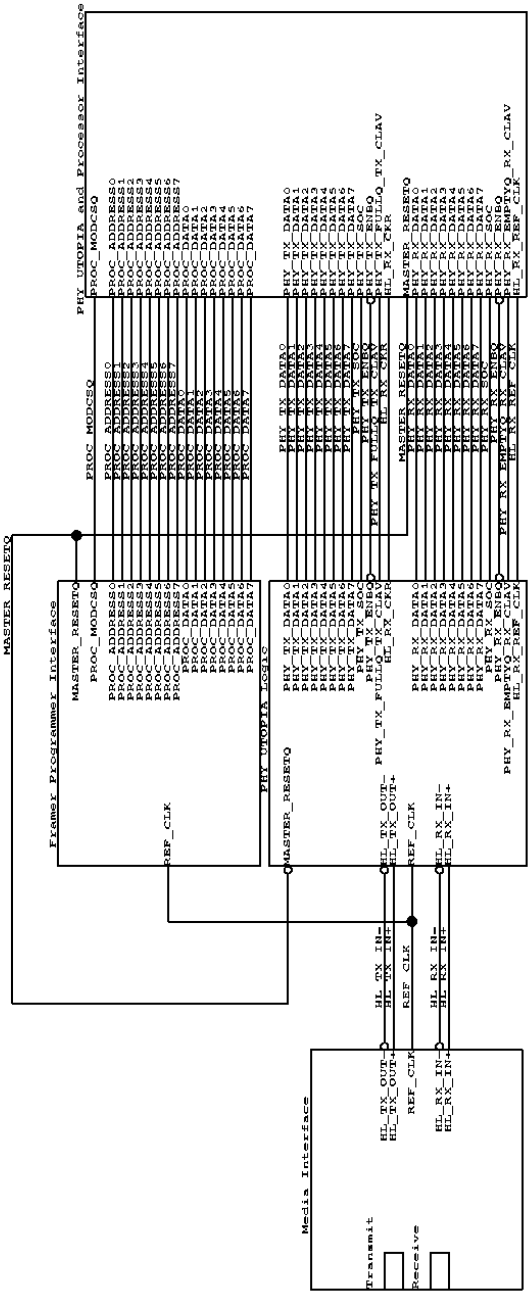
Appendix A. Hierarchical Schematics
Sheet 1 of 7: Top Level



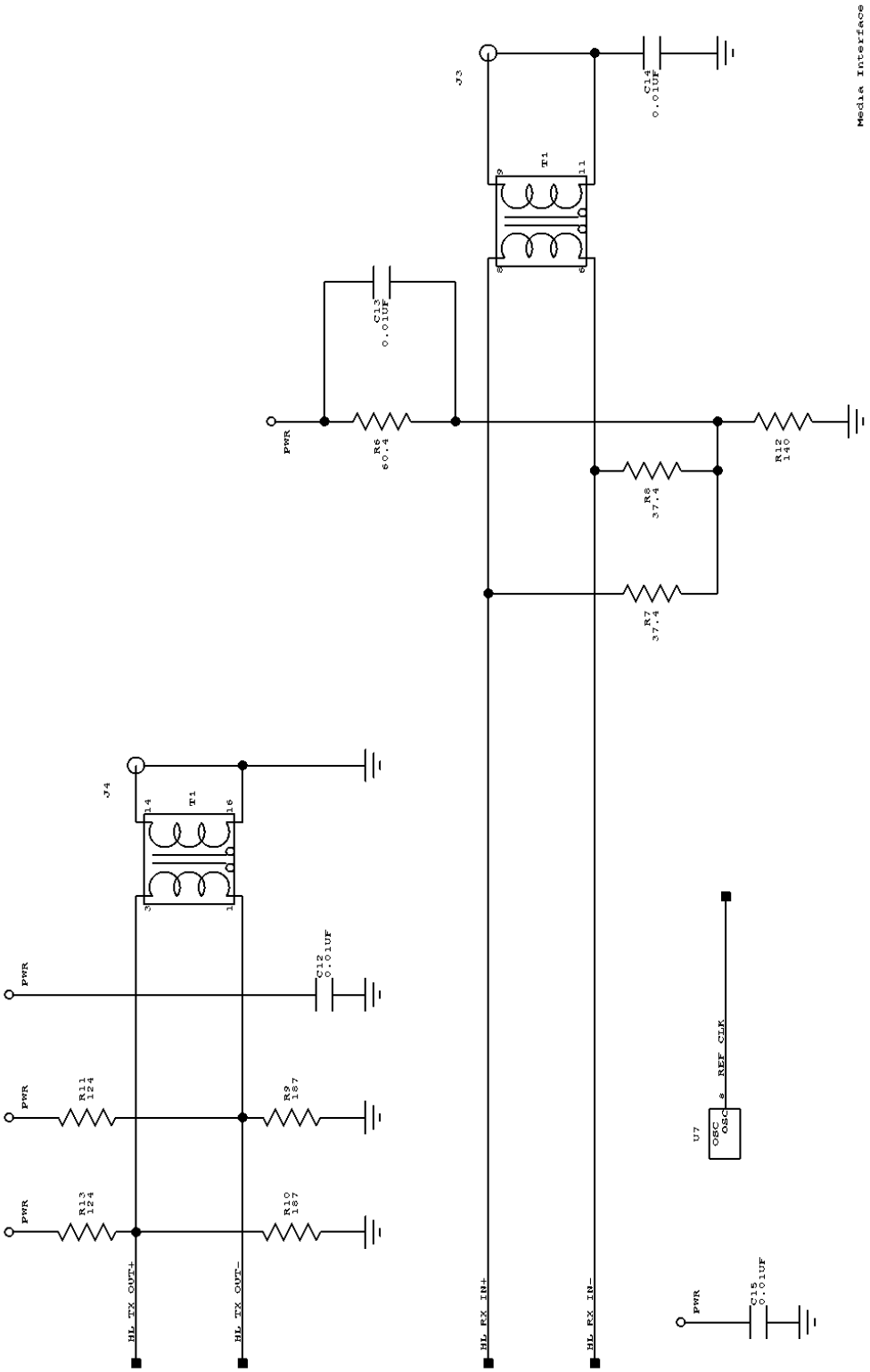
Appendix A. Hierarchical Schematics
Sheet 2 of 7: ATM Layer UTOPIA Extender



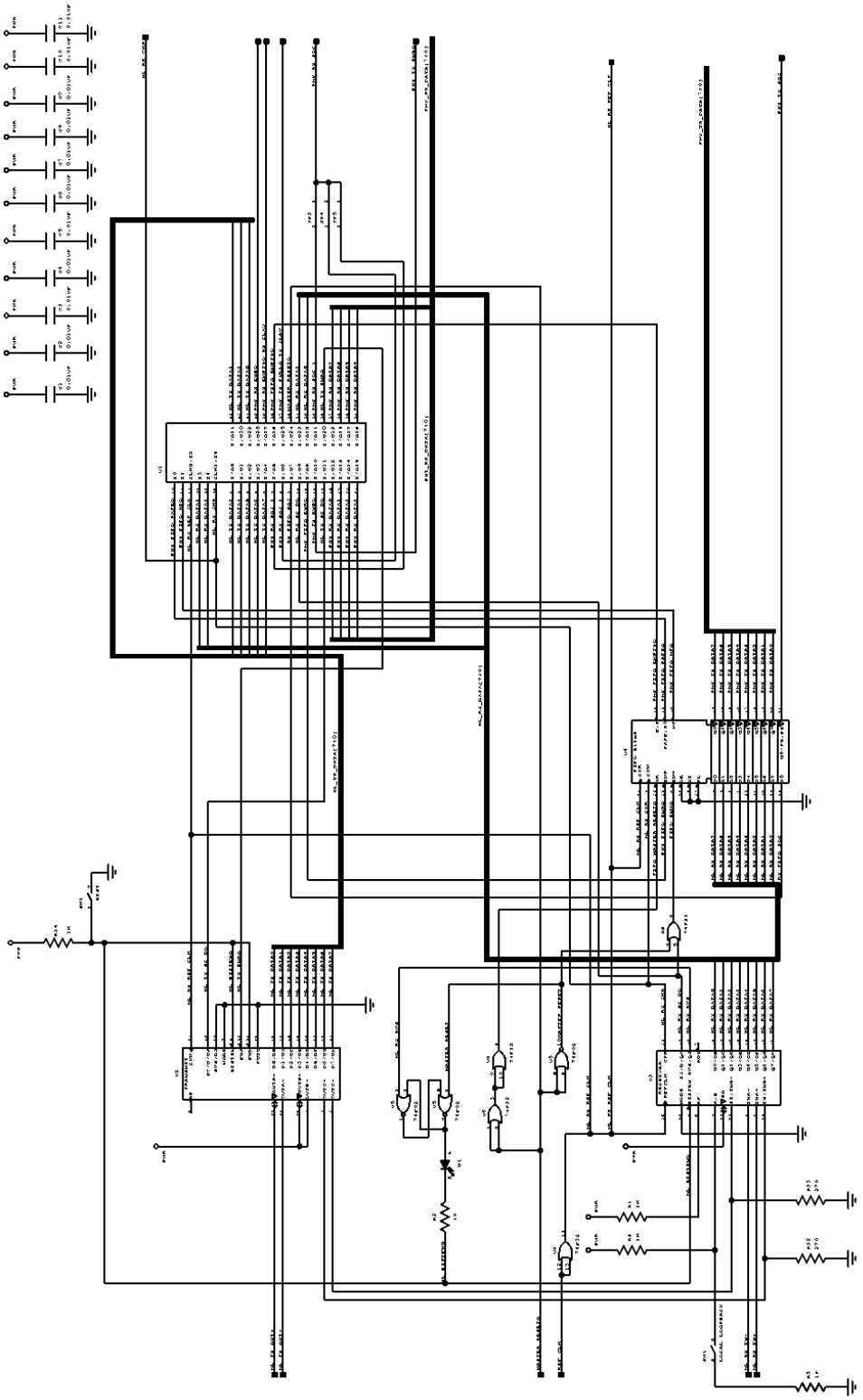
Appendix A. Hierarchical Schematics
Sheet 3 of 7: PHY Layer UTOPIA Extender



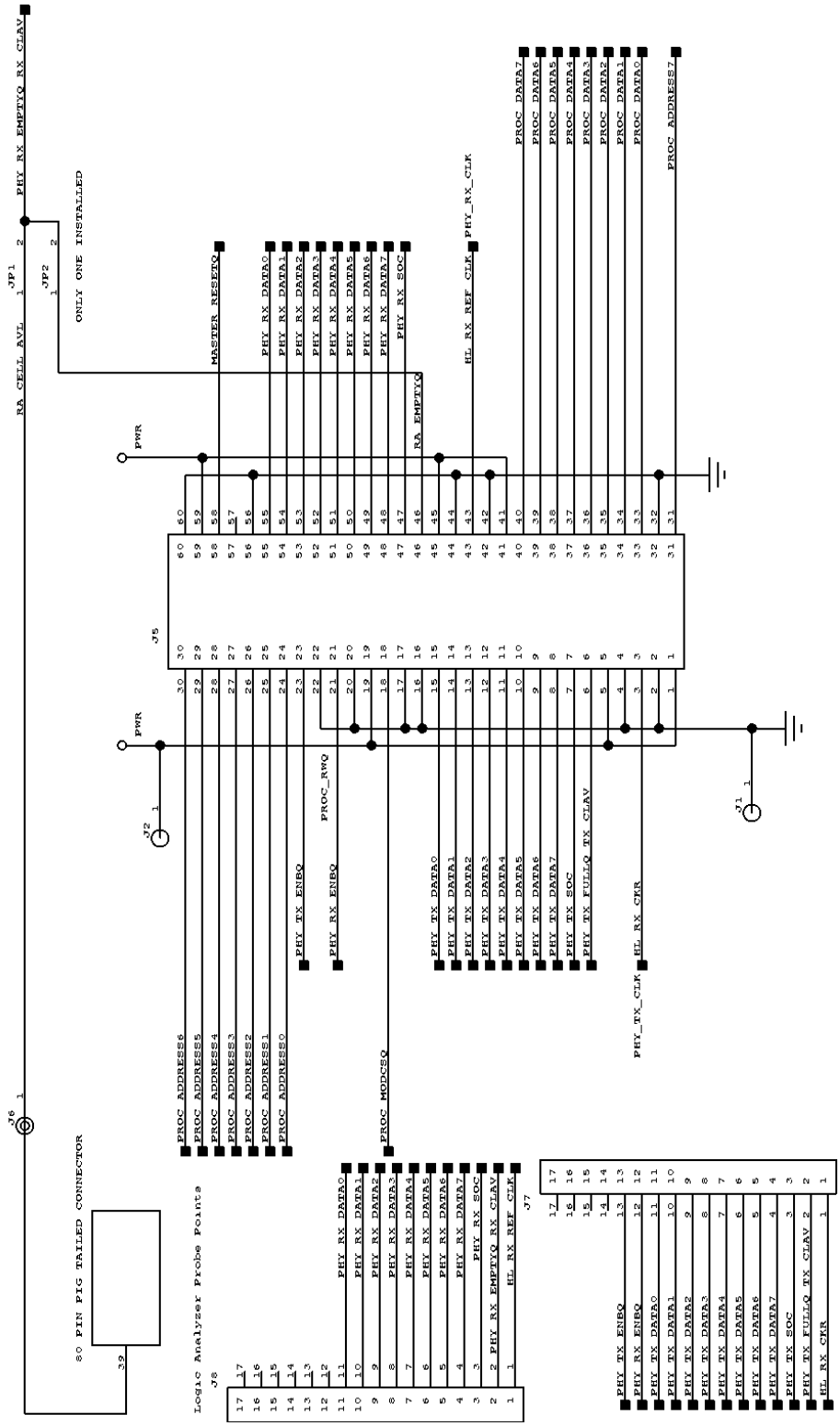
Appendix A. Hierarchical Schematics
Sheet 4 of 7: Media Interface



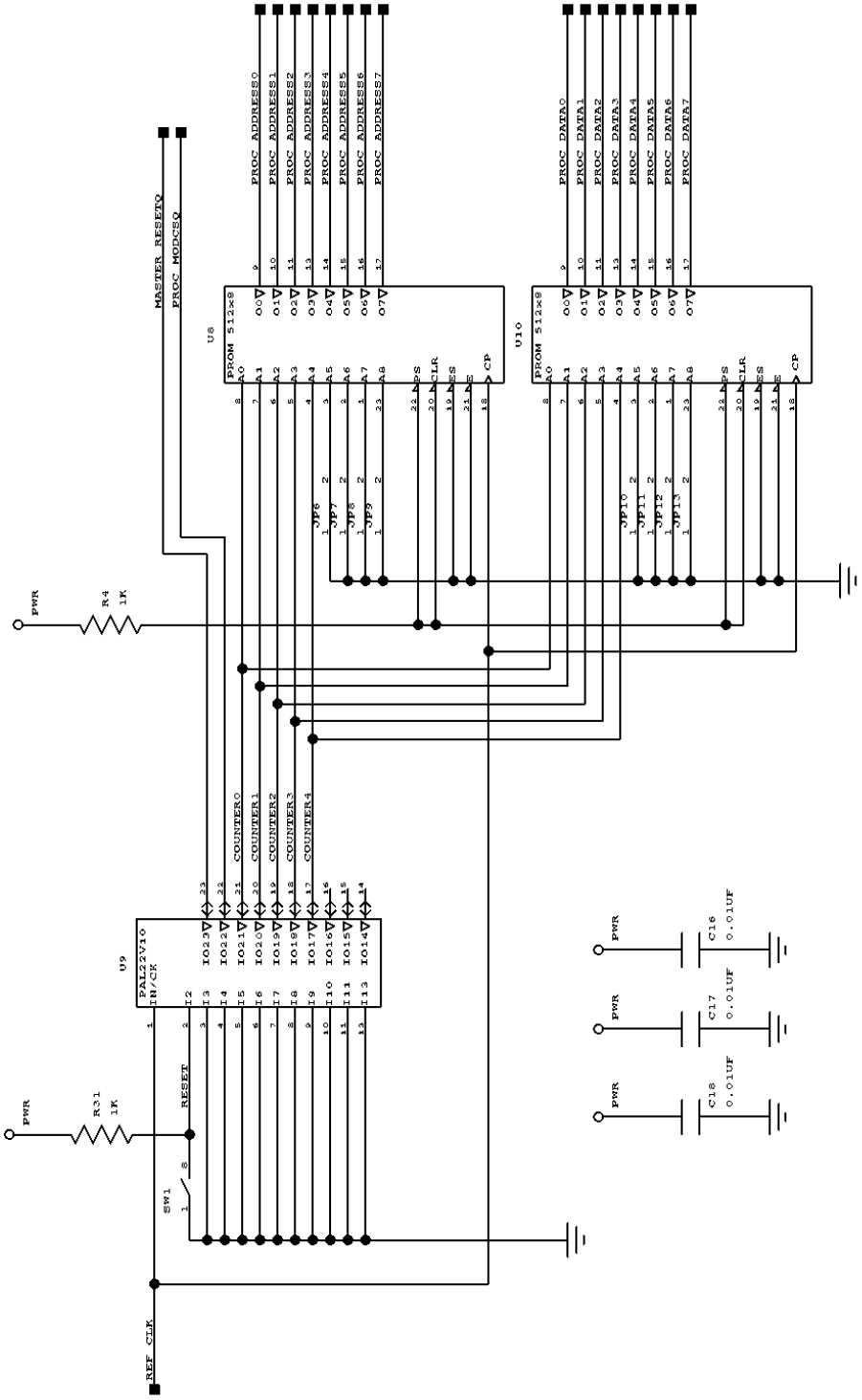
Appendix A. Hierarchical Schematics
Sheet 5 of 7: PHY UTOPIA Logic



Appendix A. Hierarchical Schematics
Sheet 6 of 7: PHY UTOPIA Processor Interface



Appendix A. Hierarchical Schematics
Sheet 7 of 7: Framer Programmer Interface



Appendix B: VHDL Code
UTOPIA Extender, PHY Layer

```
-- UTOPIA extender, PHY layer
--
--

USE WORK.phy_utopia_transmitter_package.ALL;
USE WORK.phy_utopia_receiver_package.ALL;

ENTITY phy_utopia IS
        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
                phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                rx_fifo_soc,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT;

                phy_rx_clk,
                phy_rx_empty_rx_clav            : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));

        ATTRIBUTE pin_numbers OF phy_utopia:ENTITY IS
                "hl_tx_data(3):2 " &
                "hl_tx_data(4):3 " &
                "hl_tx_data(5):4 " &
                "hl_tx_data(6):5 " &
                "hl_tx_data(7):6 " &
                "rx_fifo_soc:9 " &
                "phy_fifo_pafe:10 " &
                "phy_fifo_hf:11 " &
                "phy_rx_clk:13 " &
                "hl_rx_sc_d:14 " &
                "phy_fifo_enr:15 " &
                "phy_tx_enb:16 " &
                "hl_tx_sc_d:17 " &
                "phy_rx_data(0):18 " &
                "phy_rx_data(1):19 " &
                "phy_rx_data(2):20 " &
                "phy_rx_data(3):21 " &
                "phy_rx_data(4):24 " &
                "phy_rx_data(5):25 " &
                "phy_rx_data(6):26 " &
                "phy_rx_data(7):27 " &
                "hl_tx_ena:28 " &
                "hl_rx_data(0):30 " &
                "hl_rx_data(1):31 " &
                "hl_rx_data(2):32 " &
                "hl_rx_data(3):33 " &
                "hl_rx_ckr:35 " &
                "master_reset:36 " &
                "phy_tx_full_tx_clav:37 " &
                "phy_fifo_empty:38 " &
                "phy_rx_empty_rx_clav:39 " &
                "phy_rx_enb:40 " &
                "hl_tx_data(0):41 " &
                "hl_tx_data(1):42 " &
                "hl_tx_data(2):43 ";

END phy_utopia;

ARCHITECTURE netlist OF phy_utopia IS

        SIGNAL atm_fifo_hf_code                 : BIT;
        SIGNAL atm_fifo_not_hf_code             : BIT;
       SIGNAL phy_fifo_hf_state             : BIT;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer (continued)

```
BEGIN
        U1: phy_utopia_transmitter
                PORT MAP (hl_rx_ckr, hl_rx_sc_d, master_reset,
                                phy_tx_full_tx_clav, phy_fifo_hf,
                              phy_fifo_pafe, phy_fifo_empty, hl_rx_data,
                                phy_fifo_hf_state, rx_fifo_soc,
                              atm_fifo_hf_code, atm_fifo_not_hf_code,
                              phy_tx_enb, phy_fifo_enr);

        U2: phy_utopia_receiver
                PORT MAP (phy_rx_clk, phy_rx_empty_rx_clav, master_reset,
                              atm_fifo_hf_code, atm_fifo_not_hf_code,
                              phy_fifo_hf_state, phy_rx_data, hl_tx_sc_d,
                                hl_tx_ena, phy_rx_enb, hl_tx_data);
END netlist;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Transmitter Interface (PHY to ATM)

```
-- UTOPIA extender, PHY layer transmitter interface (PHY to ATM).
--

PACKAGE phy_utopia_transmitter_package IS

COMPONENT phy_utopia_transmitter

                -- Note, hl_rx_ckr = phy_tx_clk.

        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
               phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                phy_fifo_hf_state,
               rx_fifo_soc, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT);

END COMPONENT;
END phy_utopia_transmitter_package;

ENTITY phy_utopia_transmitter IS
        PORT(   hl_rx_ckr, hl_rx_sc_d,
                master_reset,
                phy_tx_full_tx_clav,
               phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                phy_fifo_hf_state,
               rx_fifo_soc, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
                phy_tx_enb, phy_fifo_enr        : INOUT BIT);

END phy_utopia_transmitter;

ARCHITECTURE behavior OF phy_utopia_transmitter IS

        -- Codes received from ATM side pertaining to the state
        --      of the ATM side FIFO.  Note, the 'fifo_hf_code'
        --      is a HOTLink K28.0 code, while the 'fifo_not_hf_code'
        --      is a HOTLink K28.2 code.

        CONSTANT fifo_hf_code : BIT_VECTOR := X"2";
        CONSTANT fifo_not_hf_code : BIT_VECTOR := X"0";

        SIGNAL phy_tx_enb_wait                  : BIT;

BEGIN

        -- Generate the FIFO read enable signal using the invert of
        -- phy_tx_full_tx_clav.  Also, want to disable when resetting.

        phy_fifo_enr <= NOT(phy_tx_full_tx_clav) OR NOT(master_reset);

        -- Note that data out of the FIFO is valid on the rising edge
        -- AFTER the data is read out.  So, want to delay the phy_tx_enb
        -- one clock from the FIFO read enable.

        PROCESS
        BEGIN
                WAIT UNTIL hl_rx_ckr = '1';
                phy_tx_enb_wait <= phy_fifo_empty AND phy_tx_full_tx_clav;

        END PROCESS;

        phy_tx_enb <= NOT(phy_tx_enb_wait) OR NOT(master_reset);

        -- Essentially, rx_fifo_soc is a one clock delay (w.r.t.
        -- hl_rx_ckr) of the hl_rx_sc_d pin.  This is then used to
        -- generate the input bit to the FIFO for the phy_tx_soc signal.
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Transmitter Interface (PHY to ATM) (continued)

```
        PROCESS
        BEGIN

              WAIT UNTIL hl_rx_ckr = '1';
              rx_fifo_soc <= hl_rx_sc_d;

        END PROCESS;

      PROCESS
      BEGIN

              WAIT UNTIL hl_rx_ckr = '1';

              IF ((hl_rx_data = fifo_hf_code) AND (hl_rx_sc_d = '1')) THEN
                      atm_fifo_hf_code <= '1';
              ELSIF ((hl_rx_data = fifo_not_hf_code) AND (hl_rx_sc_d = '1'))
                      THEN
                      atm_fifo_not_hf_code <= '1';
              ELSE
                      atm_fifo_hf_code <= '0';
                      atm_fifo_not_hf_code <= '0';
              END IF;
      END PROCESS;

        PROCESS (master_reset, phy_fifo_pafe, phy_fifo_hf)

        -- Hysterisis is added to the PHY FIFO half-full flag via the
        --      input 'phy_fifo_hf_state'.  Thus, the half-full state
        --      is set to TRUE (1) when 'phy_fifo_hf' = 0.  The half-full state
        --      is set to FALSE (0) when 'phy_fifo_pafe' = 0.

        BEGIN

        phy_fifo_hf_state <= (NOT(phy_fifo_hf) OR (phy_fifo_pafe AND
                              phy_fifo_hf_state)) AND (master_reset);

        END PROCESS;
END behavior;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM)

```
-- UTOPIA extender, PHY layer receiver interface (PHY to ATM).
--
--

PACKAGE phy_utopia_receiver_package IS

COMPONENT phy_utopia_receiver
        PORT(   phy_rx_clk, phy_rx_empty_rx_clav,
               master_reset, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
               phy_fifo_hf_state             : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));

END COMPONENT;
END phy_utopia_receiver_package;

ENTITY phy_utopia_receiver IS

        PORT(   phy_rx_clk, phy_rx_empty_rx_clav,
                master_reset, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
               phy_fifo_hf_state             : IN BIT;
                phy_rx_data                     : IN BIT_VECTOR(0 to 7);
                hl_tx_sc_d, hl_tx_ena,
                phy_rx_enb                      : INOUT BIT;
                hl_tx_data                      : INOUT BIT_VECTOR(0 to 7));


END phy_utopia_receiver;

ARCHITECTURE behavior OF phy_utopia_receiver IS

        -- Codes received from ATM side pertaining to the state
        --      of the PHY side FIFO.  Note, the 'fifo_hf_code'
        --      is a HOTLink K28.0 code, while the 'fifo_not_hf_code'
        --      is a HOTLink K28.2 code.
        -- 'packet_size' is the number of bytes in a packet (i.e. 53 bytes)
        -- 'packet_gap' is the minimum number clocks allowed between
        --      packets.
        -- 'packet_start_delay' is the number of clocks from when 'phy_rx_enb'
        --      is valid to when data appears at the PHY UTOPIA receiver
        --      interface.  Currently, this is defined by the UTOPIA spec.
        --      as 1 clock.

        CONSTANT fifo_hf_code                   : BIT_VECTOR := X"02";
        CONSTANT fifo_not_hf_code               : BIT_VECTOR := X"00";
        CONSTANT packet_size                    : INTEGER := 53;
        CONSTANT packet_gap                     : INTEGER := 1;
        CONSTANT packet_start_delay             : INTEGER := 0;

        -- State of ATM side FIFO maintained on PHY side as 'atm_fifo_hf'.
        -- State of PHY side FIFO as known on ATM side is
        --      'phy_fifo_hf_on_atm'.

        SIGNAL atm_fifo_hf                      : BIT:='0';
        SIGNAL phy_fifo_hf_on_atm               : BIT:='0';

        -- The 'counter' signal is used to establish the length of
        --      the packet from the PHY UTOPIA receiver interface.  It
        --      is also used to assure that there are a sufficient number
        --      of clocks in between packets as defined by 'packet_gap'.
        --      The 'hotlink_idle' signal is used to indicate no data
        --      is being transmitted by the HOTLink Tx and thus the
        --      Tx could be used to send FIFO update codes.

        SIGNAL counter                          : INTEGER(0 to packet_size):=0;
        SIGNAL hotlink_idle                     : BIT:='0';
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
        TYPE    state_type IS (wait_here, start_delay, count, cell_gap);
        SIGNAL  present_state, next_state     : state_type := wait_here;

BEGIN

        PROCESS (master_reset, atm_fifo_hf_code, atm_fifo_not_hf_code)
        BEGIN

                IF (master_reset = '0' OR atm_fifo_not_hf_code = '1') THEN
                        atm_fifo_hf <= '0';
                ELSIF (atm_fifo_hf_code = '1') THEN
                        atm_fifo_hf <= '1';

                        -- Set 'atm_fifo_hf' to 1 when receive
                        --      'atm_fifo_hf_code' and clear when receive
                        --      'atm_fifo_not_hf_code'.

                END IF;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL phy_rx_clk = '1';

                IF (present_state /= next_state)
                THEN
                        counter <= 1;
                ELSE
                        counter <= counter +1;
                END IF;

        END PROCESS;

        PROCESS(present_state, counter, phy_rx_empty_rx_clav, atm_fifo_hf,
                master_reset)

                -- 'phy_rx_empty_rx_clav' is 1 when the PHY side has
                --      a full cell (53 bytes).  So, if the ATM side
                --      FIFO is not half-full, then set 'phy_rx_enb'
                --      to 0 and start transmitting cells back to the
                --      ATM side.  Stop (i.e. set 'phy_rx_enb' to 1)
                --      after 53 bytes to prevent back to back cell
                --      transfers from the PHY UTOPIA receiver interface.
                --      Wait an additional 'packet_gap' number of clocks
                --      before reenabling the receiver via 'phy_rx_enb'.
                --      We must assure that there are at least packet_gap
                --      bytes between packets in order to recreate the
                --      rx_soc signal on the ATM side.  This gap will
                --      also be used to send PHY FIFO state codes to
                --      the ATM side.
        BEGIN

                CASE present_state IS

                WHEN wait_here =>

                        phy_rx_enb <= '1';
                        hotlink_idle <= '1';
                        IF (phy_rx_empty_rx_clav = '1' AND atm_fifo_hf = '0'
                                AND master_reset = '1')
                        THEN
                                IF (counter < packet_start_delay)
                                THEN
                                        next_state <= start_delay;
                                ELSE
                                        next_state <= count;
                                END IF;
                        ELSE
                                next_state <= wait_here;
                        END IF;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
        WHEN start_delay =>
                phy_rx_enb <= '0';
                hotlink_idle <= '1';
                IF ((counter < packet_start_delay)
                        AND master_reset = '1')
                THEN
                        next_state <= start_delay;
                ELSIF (master_reset = '0')
                THEN
                        next_state <= wait_here;
                ELSE
                        next_state <= count;
                END IF;
        WHEN count =>
                  phy_rx_enb <= '0';
                hotlink_idle <= '0';
                IF ((counter < packet_size)
                        AND master_reset = '1')
                THEN
                        next_state <= count;
                ELSIF (master_reset = '0')
                THEN
                        next_state <= wait_here;
                ELSE
                        next_state <= cell_gap;
                END IF;
        WHEN cell_gap =>
                  phy_rx_enb <= '1';
                hotlink_idle <= '1';
                IF (counter < packet_gap)
                THEN
                        next_state <= cell_gap;
                ELSIF (phy_rx_empty_rx_clav = '1'
                        AND atm_fifo_hf = '0' AND master_reset = '1')
                THEN
                        IF (packet_start_delay < packet_gap)
                        THEN
                                next_state <= count;
                        ELSE
                                next_state <= start_delay;
                        END IF;
                ELSE
                        next_state <= wait_here;
                END IF;
        END CASE;
END PROCESS;

 PROCESS
 BEGIN
        WAIT UNTIL phy_rx_clk = '1';
        present_state <= next_state;
END PROCESS;

PROCESS (phy_fifo_hf_state, phy_fifo_hf_on_atm, hotlink_idle,
                phy_rx_clk)
BEGIN
        -- If hotlink_idle = '0' send data.
        IF (hotlink_idle = '0')
        THEN
                hl_tx_ena <= '0';
                hl_tx_sc_d <= '0';
                hl_tx_data <= phy_rx_data;
```

Appendix B: VHDL Code
UTOPIA Extender, PHY Layer Receiver Interface (PHY to ATM) (continued)

```
                -- If the HOTLink is idle (no data being sent) and
                --      the FIFO state needs updating, send the code.

                ELSE
                        hl_tx_sc_d <= '1';
                        IF (phy_fifo_hf_state /= phy_fifo_hf_on_atm)
                        THEN
                                hl_tx_ena <= '0';
                                IF (phy_fifo_hf_state = '1') THEN
                                        hl_tx_data <= fifo_hf_code;
                                ELSE
                                        hl_tx_data <= fifo_not_hf_code;
                                END IF;
                        ELSE
                                hl_tx_ena <= '1';
                        END IF;
                END IF;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL phy_rx_clk = '1';

                IF hotlink_idle = '1'
                THEN
                        phy_fifo_hf_on_atm <= phy_fifo_hf_state;
                END IF;
        END PROCESS;
END behavior;
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer

```
-- UTOPIA extender, Duke PHY board programmer
--

PACKAGE duke_programmer_package IS

COMPONENT duke_programmer
        PORT(   ref_clk, reset                      : IN BIT;
               proc_modcs, master_reset      : INOUT BIT;
               counter                       : INOUT INTEGER(0 to 24));

END COMPONENT;
END duke_programmer_package;

ENTITY duke_programmer IS
        PORT(   ref_clk, reset                      : IN BIT;
               proc_modcs, master_reset      : INOUT BIT;
               counter                       : INOUT INTEGER(0 to 24));

       ATTRIBUTE pin_numbers OF duke_programmer:ENTITY IS
               "reset:2 " &
               "ref_clk:1 " &
               "counter(0):21 " &
               "counter(1):20 " &
               "counter(2):19 " &
               "counter(3):18 " &
               "counter(4):17 " &
               "proc_modcs:22 " &
               "master_reset:23 ";

END duke_programmer;

ARCHITECTURE behavior OF duke_programmer IS

       CONSTANT num_values                  : INTEGER :=24;

       TYPE state_type IS (wait_here, do_reset, count1, count2, count3);

       TYPE addrdata IS ARRAY(0 to num_values – 1) OF BIT_VECTOR(0 to 7);

       CONSTANT addresses : addrdata :=
       (       X"81",
               X"81",
               X"8D",
               X"8D",
               X"20",
               X"80",
               X"82",
               X"83",
               X"84",
               X"85",
               X"86",
               X"87",
               X"88",
               X"89",
               X"8A",
               X"8B",
               X"8C",
               X"8E",
               X"8F",
               X"90",
               X"91",
               X"92",
               X"9E",
               X"9F");
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer (continued)

```
        CONSTANT data : addrdata :=
        (       X"01",
                X"00",
                X"01",
                X"00",
                X"0A",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00",
                X"00");

        SIGNAL  present_state, next_state     : state_type := wait_here;

BEGIN

        PROCESS(present_state, reset, ref_clk)
        BEGIN

                CASE present_state IS

                WHEN wait_here =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        IF (reset = '0')
                        THEN
                                next_state <= do_reset;
                        ELSE
                                next_state <= wait_here;
                        END IF;

                WHEN do_reset =>

                        master_reset <= '0';
                        proc_modcs <= '1';
                        next_state <= count1;

                WHEN count1 =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        next_state <= count2;

                WHEN count2 =>

                        master_reset <= '1';
                        proc_modcs <= '0';
                        next_state <= count3;
```

Appendix B: VHDL Code
UTOPIA Extender, Duke PHY Board Programmer (continued)

```
                WHEN count3 =>

                        master_reset <= '1';
                        proc_modcs <= '1';
                        next_state <= count2;
                        IF (counter < num_values - 1)
                        THEN
                                next_state <= count1;
                        ELSE
                                next_state <= wait_here;
                        END IF;

                END CASE;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL ref_clk = '1';

                present_state <= next_state;
        END PROCESS;

        PROCESS
        BEGIN

                WAIT UNTIL ref_clk = '1';

                IF (present_state = count3)
                THEN
                        counter <= counter + 1;
                END IF;
        END PROCESS;

END behavior;
```