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. VMEbus, 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 few 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 interconnect. When information needs to be communicated, the sender requests a path through 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,

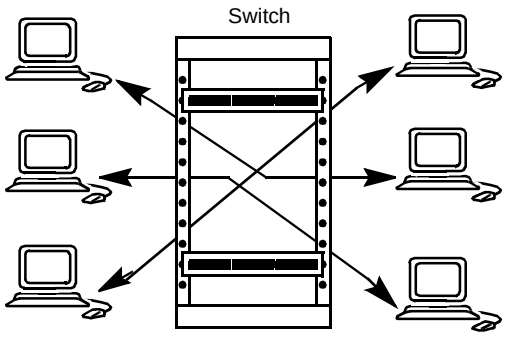


Figure 1. ATM Connections Through Switch

well-defined connection procedures, and flexible access speeds.

Using ATM, information is segmented into multiple fixed-length cells, transported to the destination, and reassembled. The ATM cell has a fixed length of 53 bytes. Being both fixed-length and small in size allows different traffic types to be mixed on the same network. Each cell is broken into two main sections, a header and a payload. The payload (48 bytes) contains the data to be conveyed—either voice, data, or video. The header (5 bytes) contains 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*). Each layer of this 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. This allows layers in the stack to be updated without affecting the communication model.

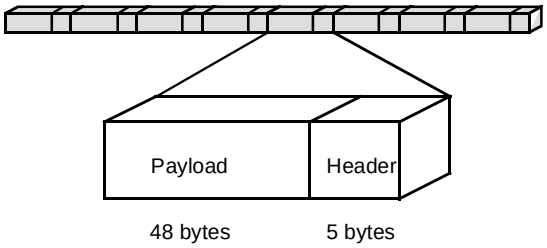


Figure 2. ATM Cell Format

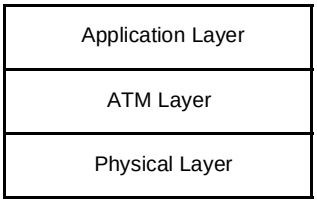


Figure 3. ATM Protocol Stack

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 4* and described in *Table 1*.

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.

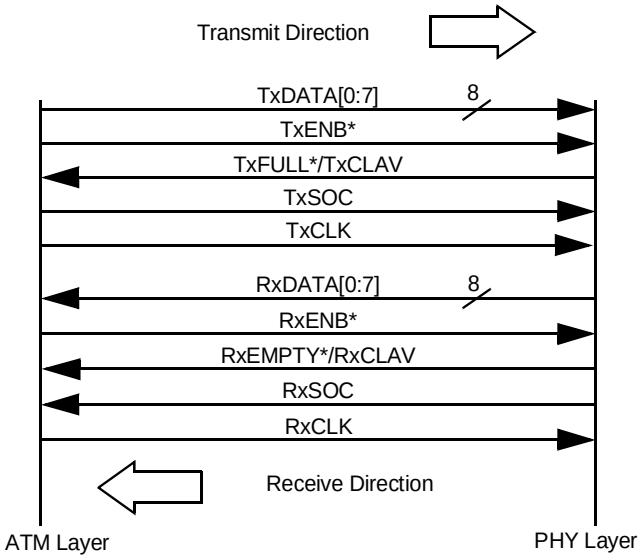


Figure 4. UTOPIA Signals

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

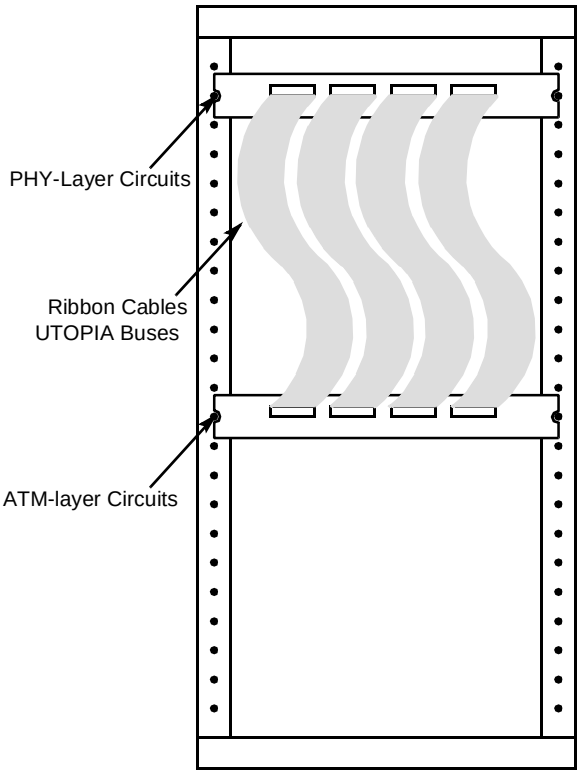


Figure 5. UTOPIA in a Rack Mount Switch

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 limited impedance control of the ribbon cable. These three characteristics can lead to skew across the signals of the UTOPIA bus as shown in *Figure 6*.

In a parallel cable environment, the skew at the destination end of the cable, as shown in *Figure 6,* can often violate the set-up and/or hold times of the UTOPIA bus at the load end. In this environment, reliable data communication over the bus is not possible. This effect is typical when high-speed parallel buses are driven over long distances. While differential transmission may correct some of the transmission problems, it has the disadvantage of doubling the number of conductors in the already bulky ribbon cable, and it still does not solve all the skew problems (skew can still result from differences in propagation delays for each signal through its respective differential driver, cable, and receiver).

The Serial Solution

A good solution to the skew problems described here is to transmit the parallel bus data as a high-performance 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 destination. 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 or balanced pair).

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 standard serial communication interfaces utilize more than one conductor (e.g., RS232), the additional signals in the interface are for other parts of the protocol and not generally for moving serial data. 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 a signal and its complement). 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 encoding or scrambling schemes and use clock extraction circuits.

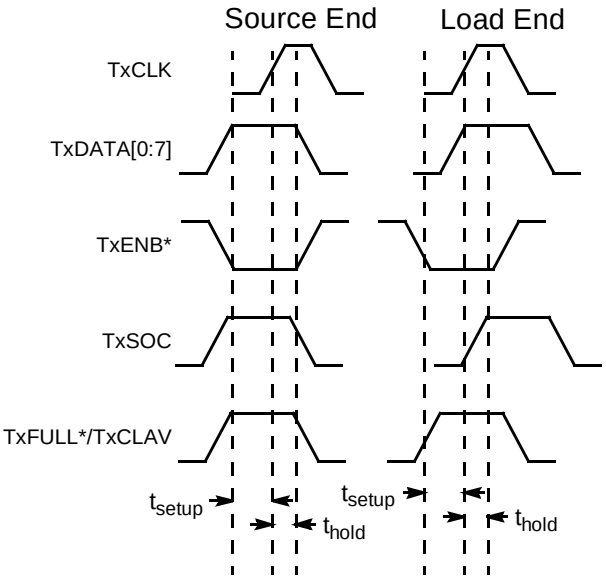


Figure 6. Effect of Skew on UTOPIA Bus

The clock extraction circuits rely on the special characteristics of the data encoding or scrambling scheme to recover or generate a clock of the same frequency and phase (with respect to the serial data) as the clock used to output the data onto the serial link. The serial-to-parallel converter in the receiver uses this recovered clock to sample the serial data stream and places the captured data into a parallel register. When this register is full, the serial-to-parallel converter presents the register contents (in parallel format) along with a synchronous character clock (generated by dividing down the recovered bit-rate clock). Because all data is recovered at the bit-rate, there is no skew between the output clock and parallel data.

A serial link has four significant advantages over a parallel bus:

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.
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 high-performance Phase-Locked Loops (PLLs). PLLs are closed-loop control systems that align two different signals. A clock-multiplier PLL aligns the phase and frequency of a divided output clock with an input reference clock. A clock-recovery PLL aligns the phase of a bit-rate clock with the edges in the input data stream. Block diagrams of PLLs performing these functions 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 loop filter. 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 NRZ 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. If these runs contain too many bits without a transition, the clock-recovery PLL may lose bit sync or fall out of phase lock. To reliably perform clock and data 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 uncontrolled unencoded data,

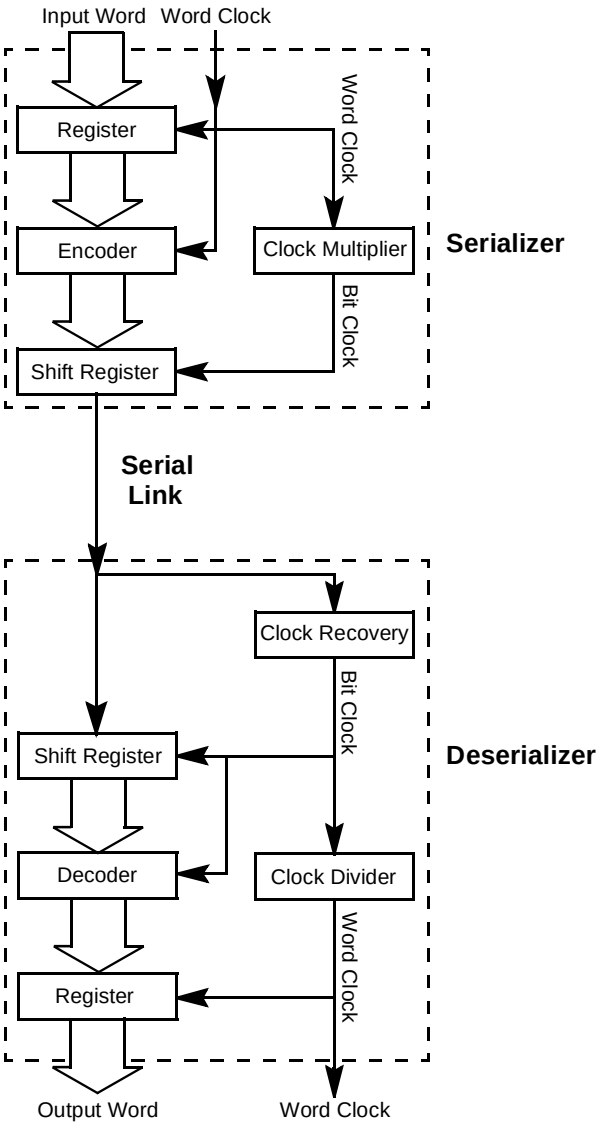


Figure 7. Architecture of a Serial Link

since a user is free to send any data pattern. Some serial patterns, for example 00000000, contain no transitions and therefore could be transmitted indefinitely resulting in a serial link without any transitions.

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

On the ATM side, the serializer converts the parallel UTOPIA transmit data into a serial stream, embedding the UTOPIA

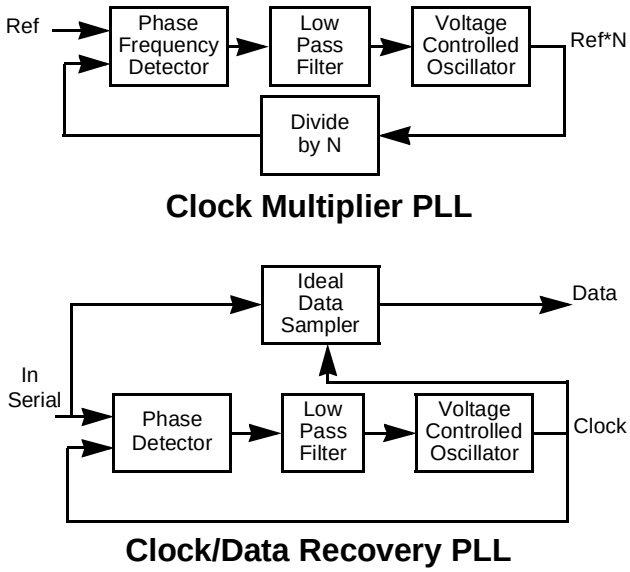


Figure 8. Multiplication and Clock/Data Recovery PLLs

transmit clock with the data. The deserializer converts the serial receive data stream (from the PHY layer) back into parallel data along with a receive clock. The FIFO memory is used 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 UTOPIA 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 along with 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. UTOPIA transmits make use of two handshaking signals: TxFull* (sourced at the PHY layer) and TxEnb* (sourced at the ATM layer). A transfer is initiated when there is room in the transmit FIFO (TxFull* is HIGH),

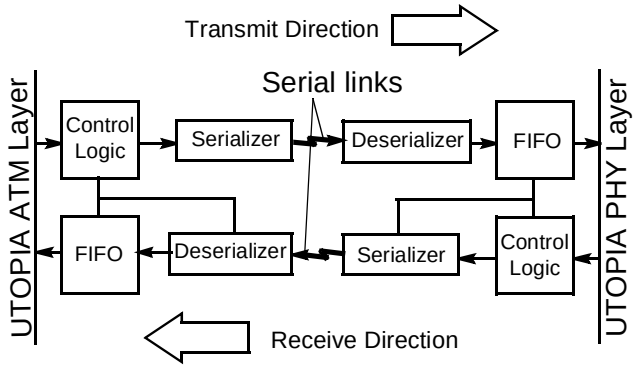


Figure 9. UTOPIA Serializer Block Diagram

and TxEnb* is asserted LOW with UTOPIA data placed onto the TxData bus. If TxFull* goes LOW at any time, the transfer must stop (per the UTOPIA specification) within four write cycles. However, since TxFull* is sourced at the PHY layer and sampled at the ATM layer (on the other side of the serial link), there is a time delay for any change of state of TxFull* 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 TxEnb* and TxData from the ATM layer to the PHY layer. A problem arises if a transfer is in progress and TxFull* is asserted LOW. The transfer begins successfully, and several octets are placed onto the serial link. However, at clock cycle 1, the TxFull* signal on the PHY side is asserted LOW, indicating that the PHY-layer transmit FIFO is full. According to the UTOPIA specification, the transfer must stop writing data (TxEnb* must go HIGH) within four character times of TxFull* being asserted LOW. In order for TxEnb* to go HIGH, the ATM layer must recognize the change in state of TxFull*, 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 characters are shown as being transmitted before TxFull* is recognized at the ATM layer). Since the PHY-layer may not recognize the change in state of TxFull* within the four byte specification limit, there is a potential for data loss.

Note that the latency in the link (the source of the delay in the above example) is not entirely due to the serializer and deserializer. As the cable length of the serial link is increased, the link latency also increases. Depending on the required operating distance of the link (along with the type of media used) this can amount to tens or hundreds of bytes.

The latency condition is solved by buffering the data coming out of the deserializer. A small FIFO works perfect for this application. With the FIFO buffer, the effects of the link latency are masked. 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 TxFull* signal. The data can then be read out of the FIFO when the PHY-layer UTOPIA interface is ready.

The UTOPIA Extender

Utilizing the concepts described earlier in this application note, a serialized UTOPIA extender was implemented. It follows the block diagram in *Figure 9*, and the hierarchical schematics in Appendix A. By serializing the UTOPIA bus, it can essentially be extended to any length, thus creating a UTOPIA Extender. The major components required to implement this design are listed in *Table 2*.

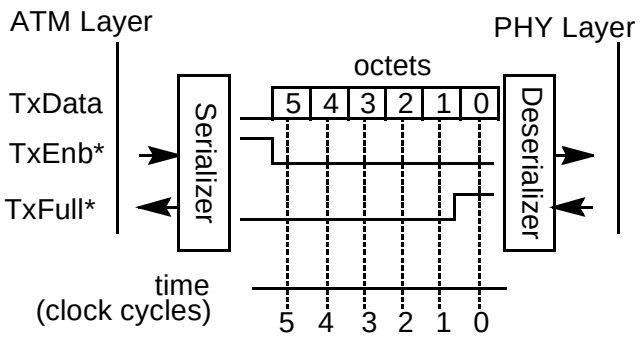


Figure 10. Round Trip Latency Example

Table 2. Cypress UTOPIA Extender Components

Function	Part Description
Serializer	CY7B923 HOTLink Tx
Deserializer	CY7B933 HOTLink Rx
FIFO	CY7C451 512x9 clocked FIFO
Control Logic	CY7C371i 32-macrocell Flash PLD

The top level of the 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 data and protocol are passed. 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 schematics are the components and logical constructs used to better describe the previous hierarchical level in greater detail. 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.

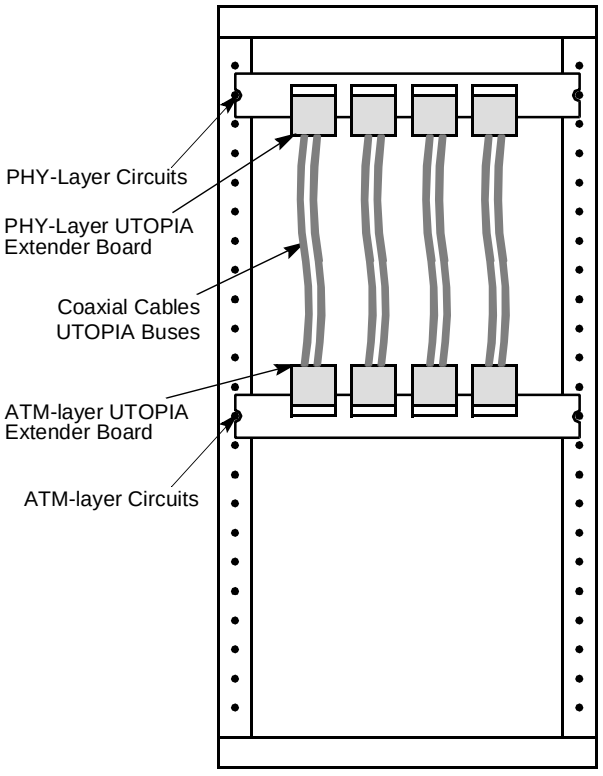


Figure 11. UTOPIA Extender in a Rack Mount Switch

The Media Interface block performs the function of interfacing the serial transmit and receive electrical signals (comprising the serial links carrying the serialized UTOPIA bus) to the specific media used in the design (in this case to coaxial cable). The Media Interface schematic contains termination networks and transformers used to interface the transmit and receive serial signals to coaxial cables.

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: Steady State and FIFO State Update. Steady State mode moves the UTOPIA transmit and receive data between the ATM and PHY layers, and handles generation of the necessary control signals. FIFO State Update mode handles the control of the buffering FIFOs to ensure that no data is lost due to overfilling of these buffers.

FIFO State Update mode also handles the condition of the CLAV signal 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 (CLH) protocol.

Cell Level Handshaking

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, the UTOPIA standard allows transmission of back-to-back cells (where every clock carries data). For the design implemented here, back-to-back cell transmissions are not directly supported (this is accomplished through special considerations in the UTOPIA control logic). The UTOPIA status flags are used to force a small gap between each cell.

This gap serves three purposes:

- To allow for the communication of the CLAV control codes from the PHY layer to the ATM layer.
- To update the status of the buffering FIFOs.
- And to allow for easy generation of the SOC signal at the receive-end of the serial link.

Steady State Mode

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

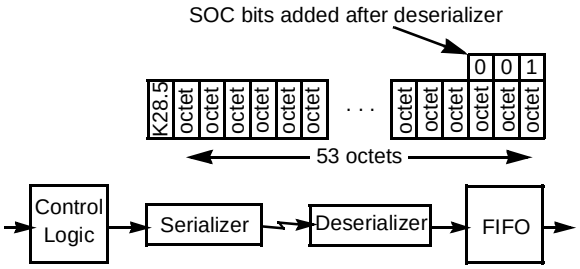


Figure 12. Transmission Data Flow

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 (using the forced gap between cells as the cell boundary indicator), an additional bit is written to the FIFO along with the first data character to generate the TxSOC signal. The remaining 52 characters are also placed into the FIFO, but without the TxSOC bit set. The TxEnb* signal to the UTOPIA interface is then generated from the TxClav 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.

FIFO State Update

The other mode of operation is FIFO State Update. This mode handles the case when the CLAV signal changes state. When TxClav is deasserted, no data is read out of the PHY-side buffering FIFO. When this FIFO fills beyond a check point, a code is sent back to the ATM-layer-side to halt data transmission until the FIFO has sufficient room. 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 the next gap in the data stream going back to the ATM-layer side (remember a gap is forced between successive cells). During this gap, the control logic inserts a FIFO-Full control code into the HOTLink Transmitter in place of one of the Idle characters (see *Figure 13*). This FIFO-Full code travels across the link back to the ATM-layer side. The ATM-layer control logic then

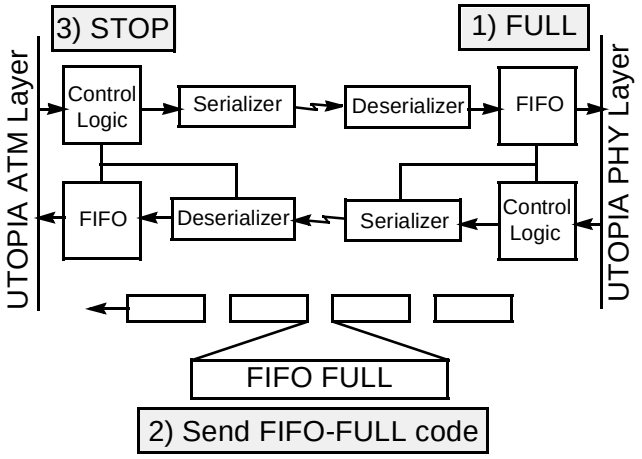


Figure 13. State Updating, FIFO Full

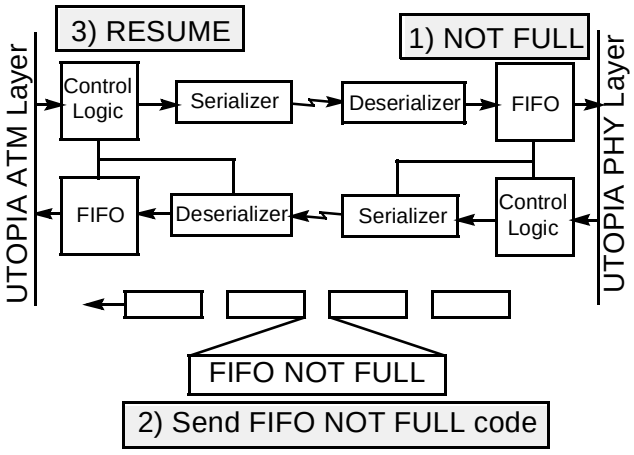


Figure 14. FIFO State Updating, FIFO Not Full

interprets the FIFO-Full code and deasserts the TxClav signal at the ATM-layer UTOPIA interface, stopping transmission at the next cell boundary.

Eventually, the PHY-layer FIFO will empty past another predefined level, indicating data transmission can resume. The control logic on the PHY-layer side then waits for a gap in the data stream back to the ATM-layer side, and inserts a FIFO-Not-Full code in place of one of the Idle characters (see *Figure 14*). This code travels across the link to the ATM-layer side where it is interpreted by the ATM-layer control logic. The control logic then asserts the TxClav 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 connections specific to the particular ATM/PHY-layer circuits used in the system. In addition, some ATM and/or PHY-layer circuits require additional components 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 may differ from that shown here depending on the unique ATM and PHY-layer circuits used in the system.

The design shown in Appendix A implements a complete UTOPIA Extender. This design was validated using a PHY-layer Circuit from Duke Communications, model DC-202® SONET/ATM UNI Transceiver Module. 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.

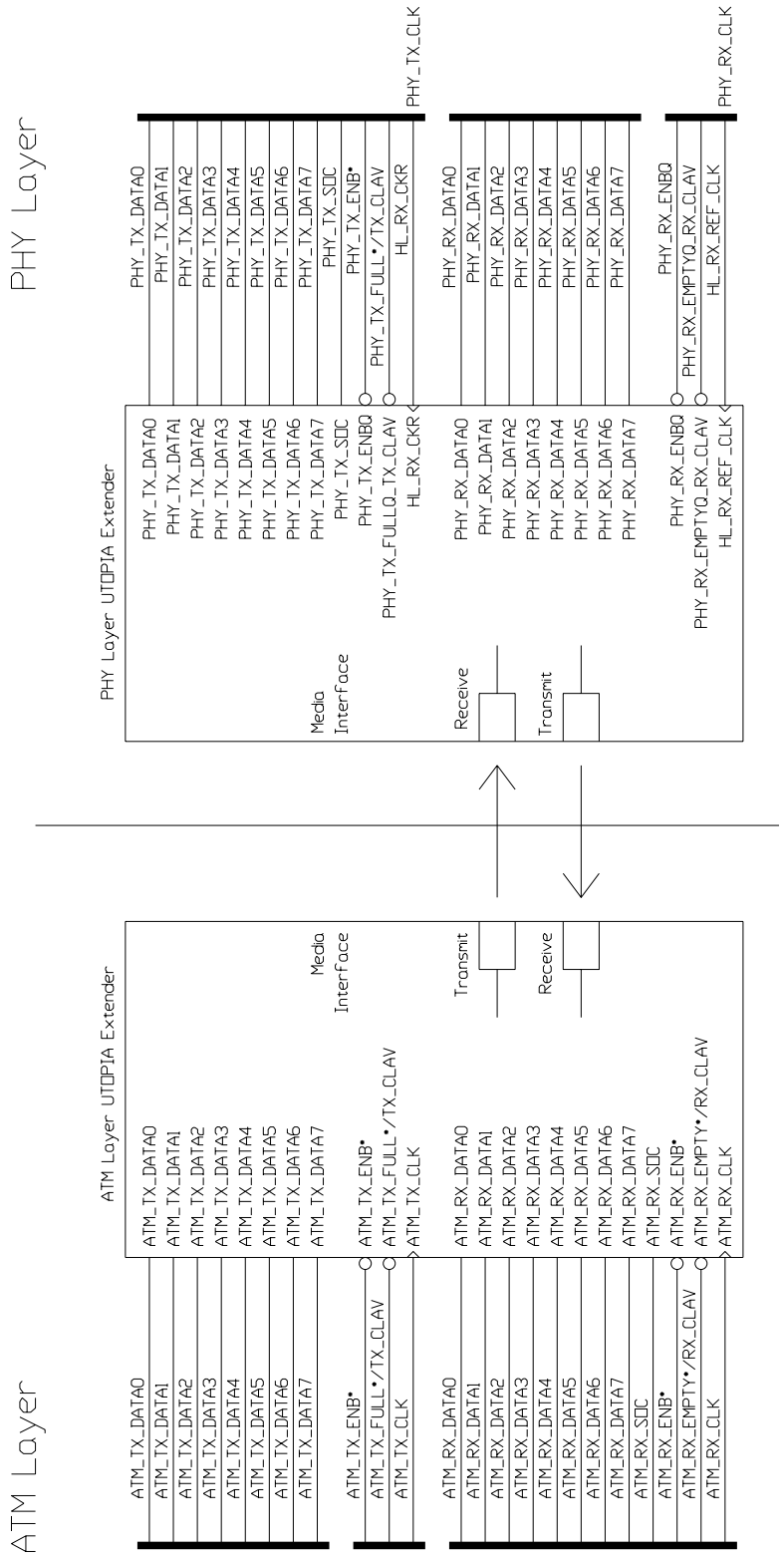
Conclusions

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

Through the addition of a FIFO and some small control logic at the transmit end of each link, it is possible to implement this same link without the required forced inter-cell gap at the UTOPIA interface. The development of this extension is left to the reader.

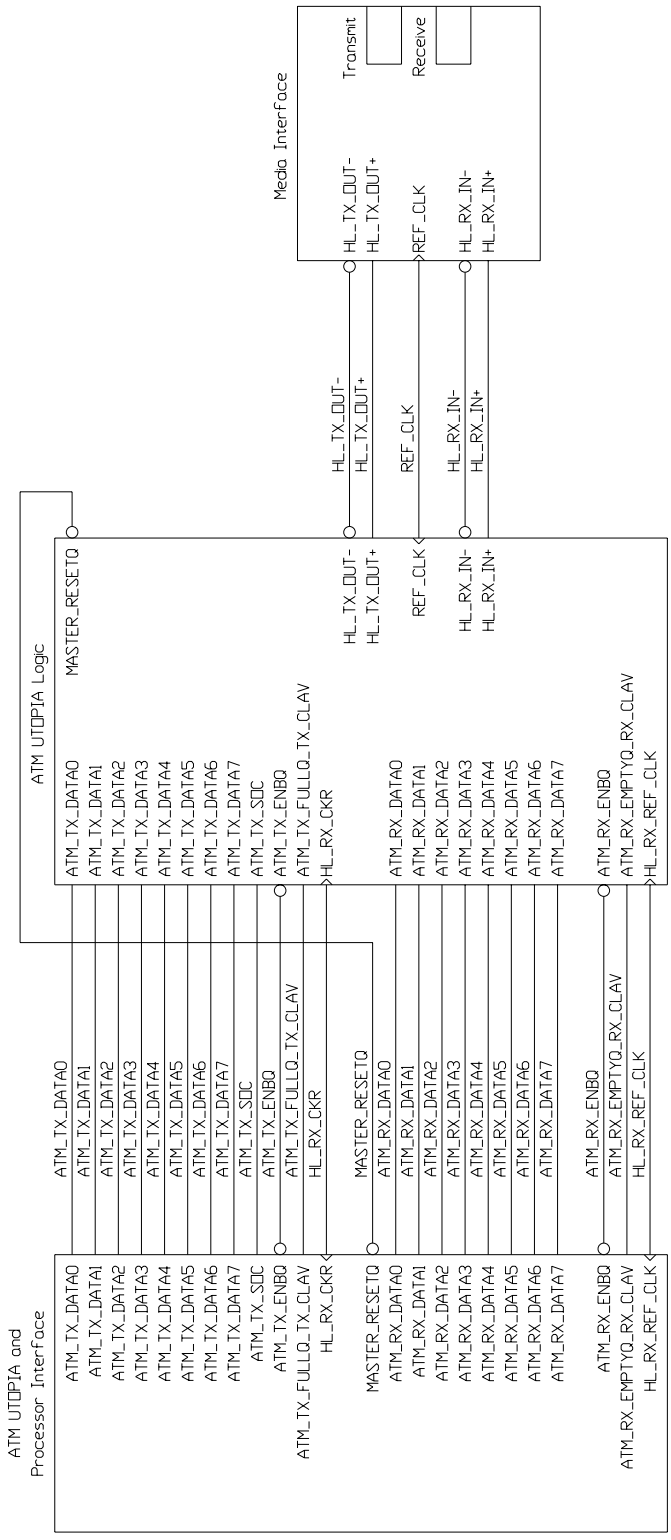
Appendix A. Hierarchical Schematics

Sheet 1 of 7: Top Level



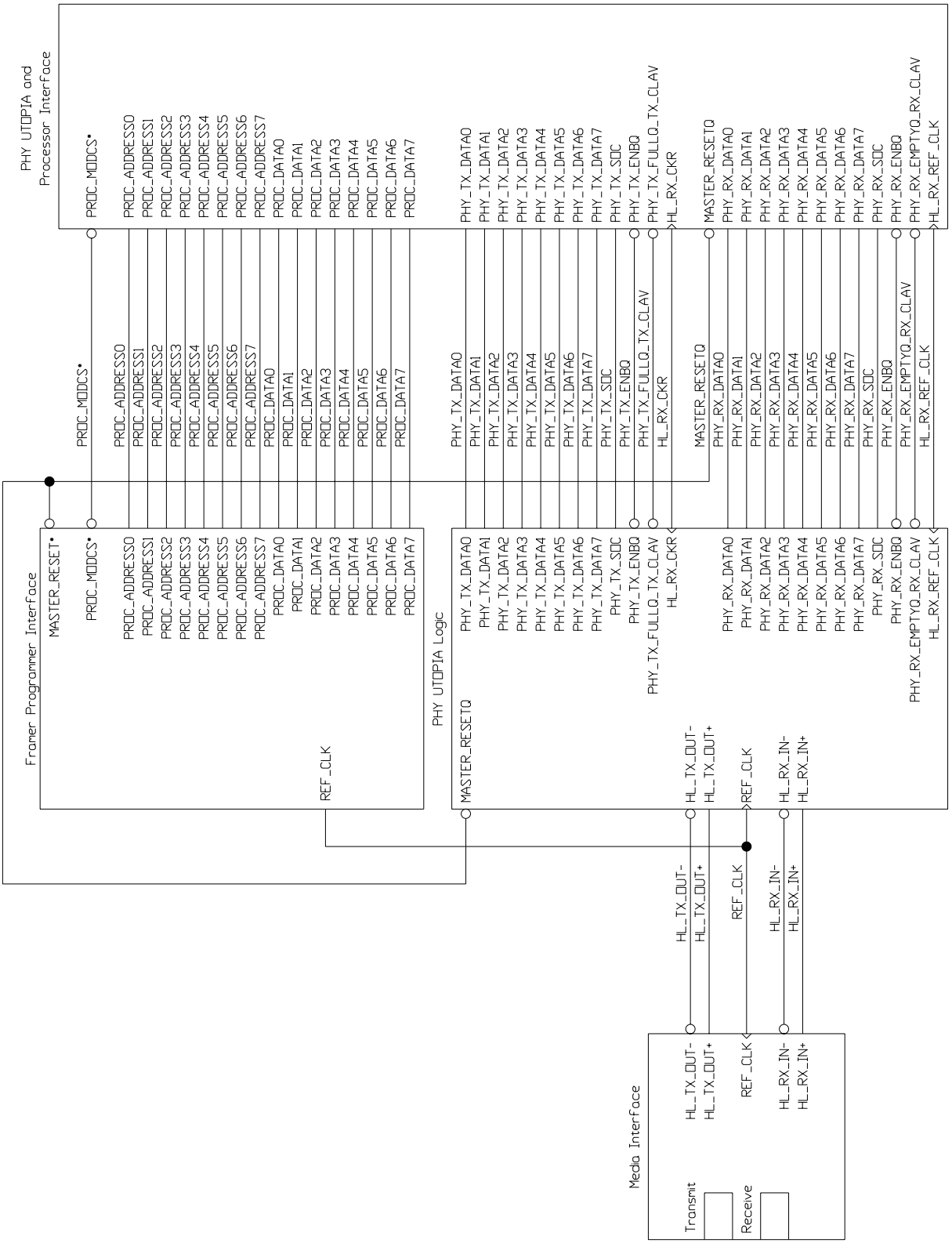
Appendix A. Hierarchical Schematics

Sheet 2 of 7: ATM-Layer UTOPIA Extender



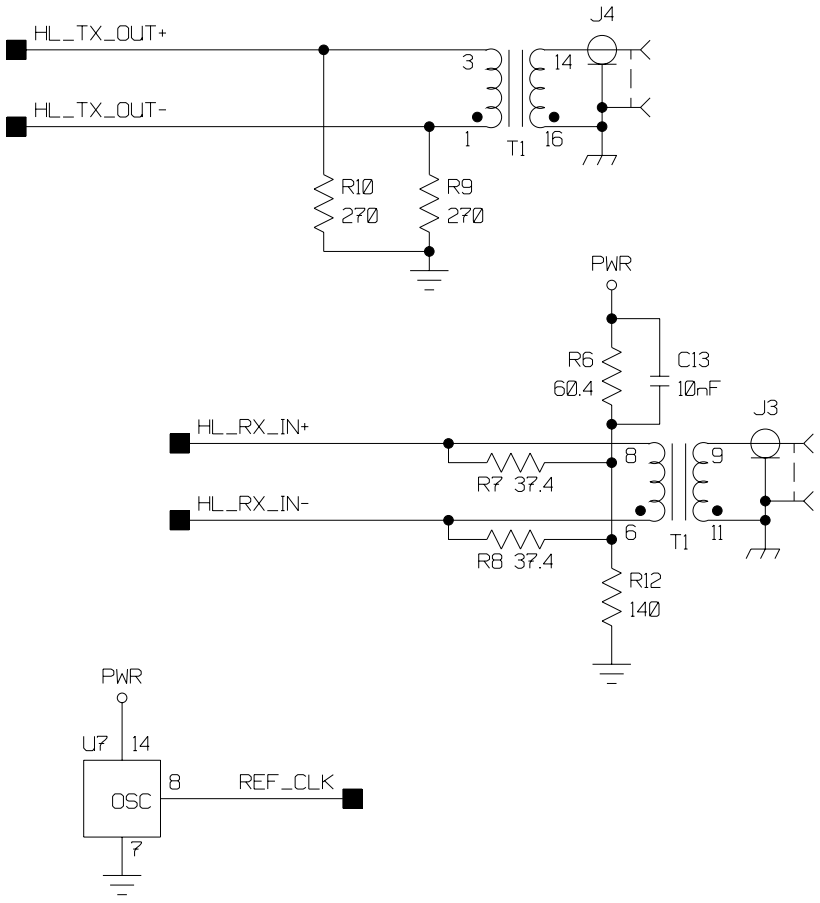
Appendix A. Hierarchical Schematics

Sheet 3 of 7: PHY-Layer UTOPIA Extender



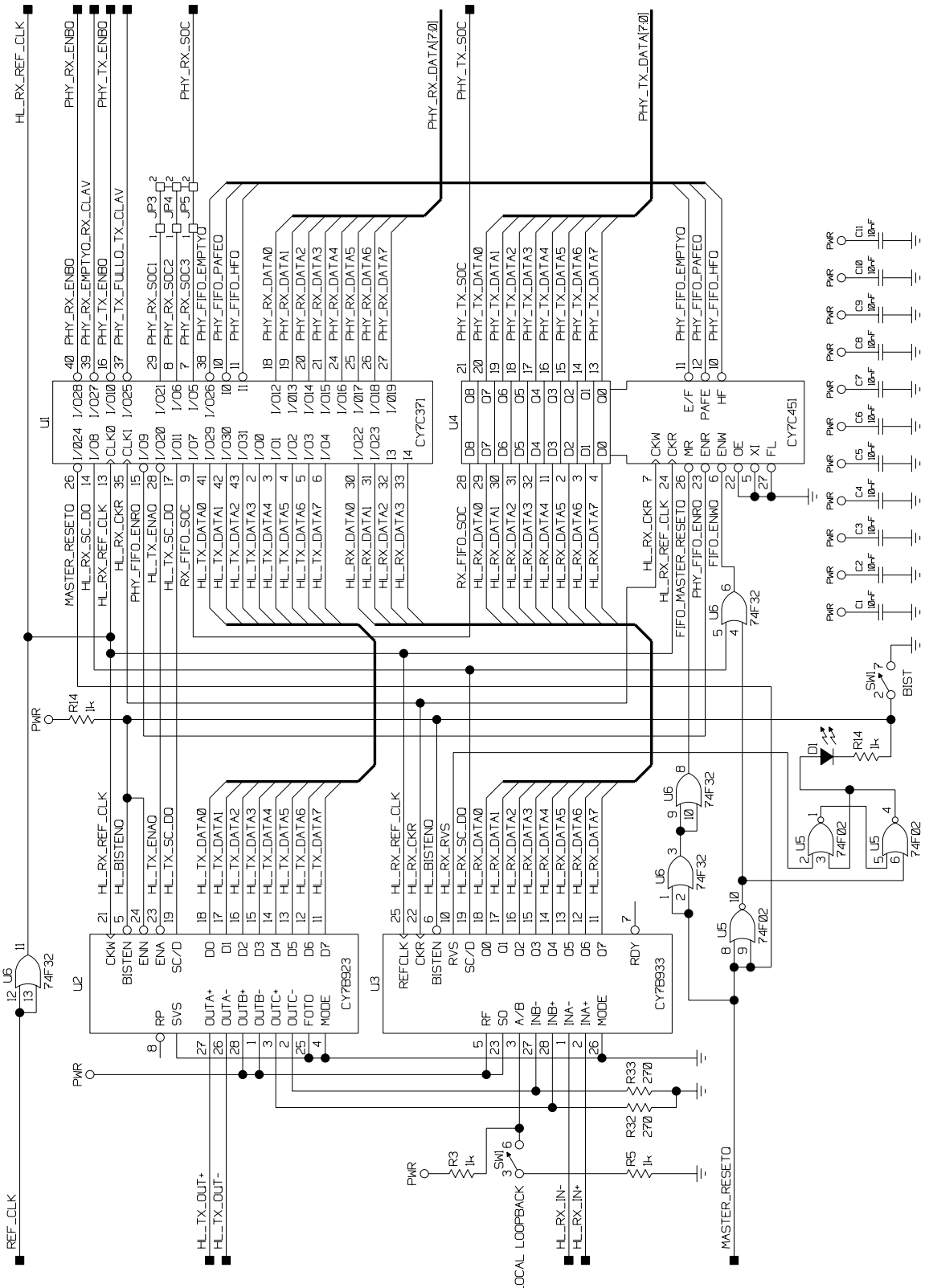
Appendix A. Hierarchical Schematics

Sheet 4 of 7: Media Interface



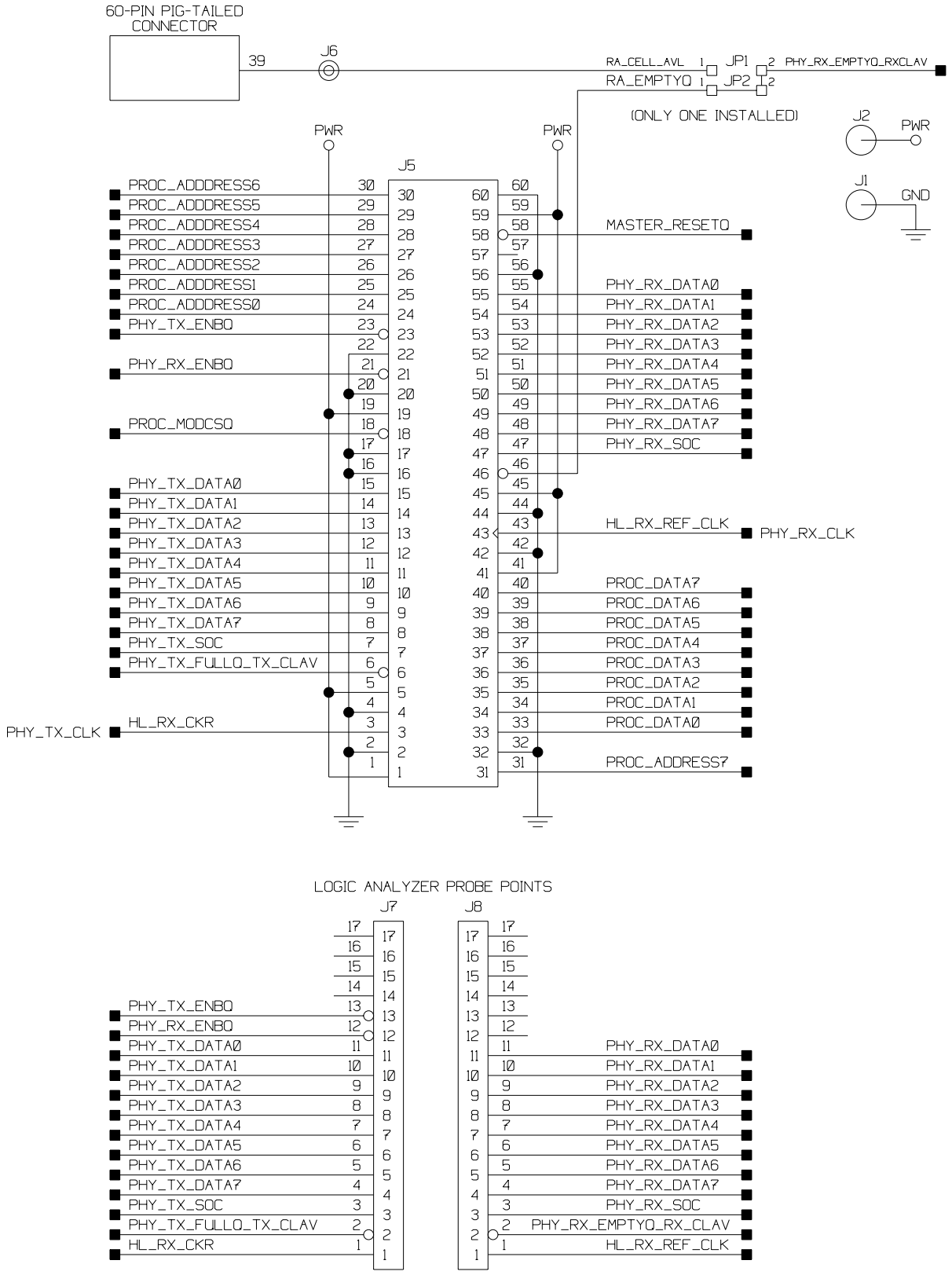
Appendix A. Hierarchical Schematics

Sheet 5 of 7: PHY UTOPIA Logic



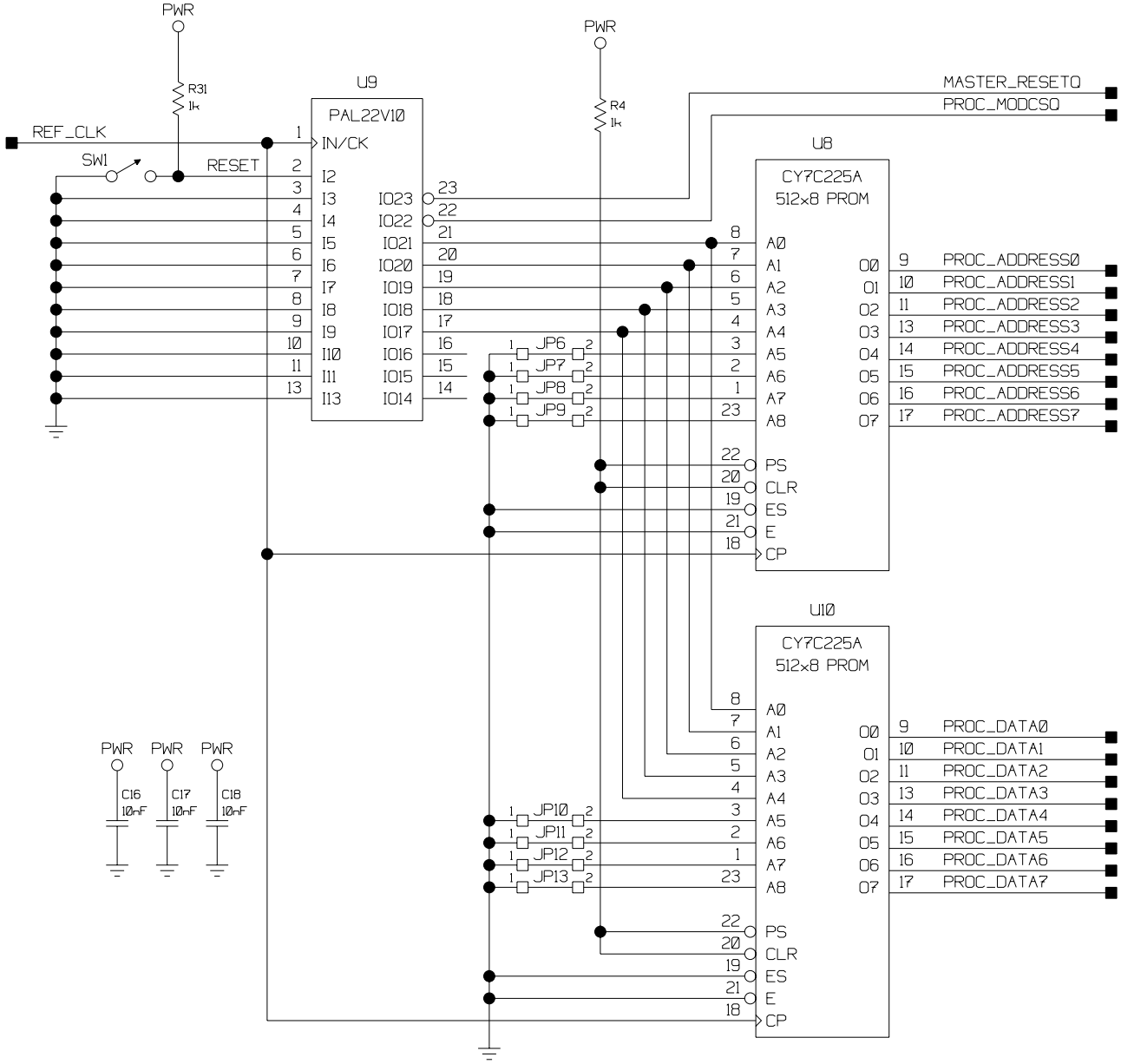
Appendix A. Hierarchical Schematics

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_TxFull_TxClav,
                phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                rx_fifo_soc,
                phy_TxEnb, 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_TxData                      : INOUT BIT_VECTOR(0 to 7));
        ATTRIBUTE pin_numbers OF phy_utopia:ENTITY IS
                "hl_TxData(3):2 " &
                "hl_TxData(4):3 " &
                "hl_TxData(5):4 " &
                "hl_TxData(6):5 " &
                "hl_TxData(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_TxEnb: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_TxFull_TxClav:37 " &
                "phy_fifo_empty:38 " &
                "phy_rx_empty_rx_clav:39 " &
                "phy_rx_enb:40 " &
                "hl_TxData(0):41 " &
                "hl_TxData(1):42 " &
                "hl_TxData(2):43 ";
```

Appendix B: VHDL Code

UTOPIA Extender, PHY Layer (continued)

```
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;
BEGIN
        U1: phy_utopia_transmitter
                PORT MAP (hl_rx_ckr, hl_rx_sc_d, master_reset,
                                phy_TxFull_TxClav, 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_TxEnb, 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_TxData);
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_TxFull_TxClav,
                                                phy_fifo_hf, phy_fifo_pafe,
                phy_fifo_empty                  : IN BIT;
                hl_rx_data                      : IN BIT_VECTOR(0 to 3);
                phy_fifo_hf_state,
                                                  x_fifo_soc, atm_fifo_hf_code,
                atm_fifo_not_hf_code,
                phy_TxEnb, 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_TxFull_TxClav,
                                                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_TxEnb, 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_TxEnb_wait                   : BIT;
BEGIN
        -- Generate the FIFO read enable signal using the invert of
        -- phy_TxFull_TxClav.  Also, want to disable when resetting.
        phy_fifo_enr <= NOT(phy_TxFull_TxClav) 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_TxEnb
        -- one clock from the FIFO read enable.

        PROCESS
        BEGIN
                WAIT UNTIL hl_rx_ckr = '1';
                phy_TxEnb_wait <= phy_fifo_empty AND phy_TxFull_TxClav;
        END PROCESS;
        phy_TxEnb <= NOT(phy_TxEnb_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_TxSOC 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_TxData                     : 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_TxData                     : 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_TxData <= 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_TxData <= fifo_hf_code;
                                ELSE
                                        hl_TxData <= 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",
                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;
```