

Layered Protocol Wrappers for Internet Packet Processing in Reconfigurable Hardware

Florian Braun, John Lockwood, Marcel Waldvogel

WUCS-01-10

July, 2001

Department of Computer Science
Applied Research Lab
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130

Abstract

A library of layered protocol wrappers has been developed that process Internet packets in reconfigurable hardware. These wrappers can be used with a reprogrammable network platform called the Field Programmable Port Extender (FPX) to rapidly prototype hardware circuits for processing Internet packets. We present a framework to streamline and simplify the development of networking applications that process ATM cells, AAL5 frames, Internet Protocol (IP) packets and UDP datagrams directly in hardware.

Supported by: NSF ANI-0096052 and Xilinx Corp.

1 Introduction

In recent years, Field Programmable Gate Arrays (FPGAs) have become sufficiently capable to implement complex networking applications directly in hardware. By using hardware that can be reprogrammed network equipment can dynamically load new functionality. Such a feature allows, for example, firewalls to add new filters that can run at line speed. The Field Programmable Port Extender has been implemented as a flexible platform for the processing of network data in hardware. The library of wrappers discussed in this paper allows applications to be developed that process data at several layers of the protocol stack. Layers are important for networks because they allow applications to be implemented at a level where the details of a protocol layer can be abstracted for layers above. At the lowest layer, networks modify raw data that passes between interfaces. At higher levels, the applications process variable length frames or packages as in the Internet Protocol. At the user-level, applications may transmit or receive messages in User Datagram Protocol messages. An Internet router or firewall are important applications that use the wrapper library to route and filter packets.

2 Background

In the Applied Research Lab at Washington University in St. Louis, a rich set of hardware and software components for research in the field of networking, switching, routing and active networking have been developed. The Field Programmable port extender has been developed to enable modular components to be implemented in reprogrammable hardware. The modules described in this document are primarily targeted to this kit, though the design is written in portable VHDL and could be used in any FPGA-based system.

2.1 Switch Fabric

The central component of this research environment is the Washington University Gigabit Switch (WUGS) [1]. It is a fully featured 8-port ATM switch, which is capable of handling up to 20 Gbps of network traffic. Each port is connected through a line card to the switch. The WUGS provides space to insert extension cards between the line cards and the backbone.

2.2 Field Programmable Port Extender

The Field Programmable Port Extender (FPX) [2, 3] provides reprogrammable logic for user applications. It provides interfaces to both the switch and the line-card, so it can be inserted between these two cards, as illustrated in figure 1(a).

The FPX contains two FPGAs: the Network Interface Device (NID) and the Reprogrammable Application Device (RAD). The NID interconnects the WUGS, the line card and the RAD via an on-chip ATM switch core. It also provides the logic to dynamically reprogram the RAD. The RAD can be programmed to hold user-defined modules. This feature enables user-defined network modules to be dynamically loaded into the system. The RAD is also connected to two SRAM and two SDRAM components. The memory modules can be used to cache cell data or hold large tables. Figure 1(b) illustrates the major components on an FPX board.

2.3 FPX Modules

User applications are implemented on the RAD as modules. Modules are hardware components with a well-defined interface which communicate with the RAD and other infrastructure components. The basic data interface is a 32-bit wide Utopia interface. Internet packets enter the module using classical IP over ATM encapsulation and segmentation into ATM cells. The data bus carries header and payload of the cells. The other signals in the module interface are used for congestion control and to connect to memory controllers to access the off-chip memory. The complete module interface is documented in [4, 5].

Usually, two application modules are present on the RAD. Typically, one handles data from the line card to the switch (ingress) and the other handles data from the switch to the line card (egress). As with the Transmutable Telecom System [6], modules can be replaced by reprogramming the FPGA in the system at any time. In the case of the FPX, this functionality occurs via partial reprogramming of the RAD FPGA. A reconfiguration component performs a handshaking protocol with the modules to prevent loss of data.

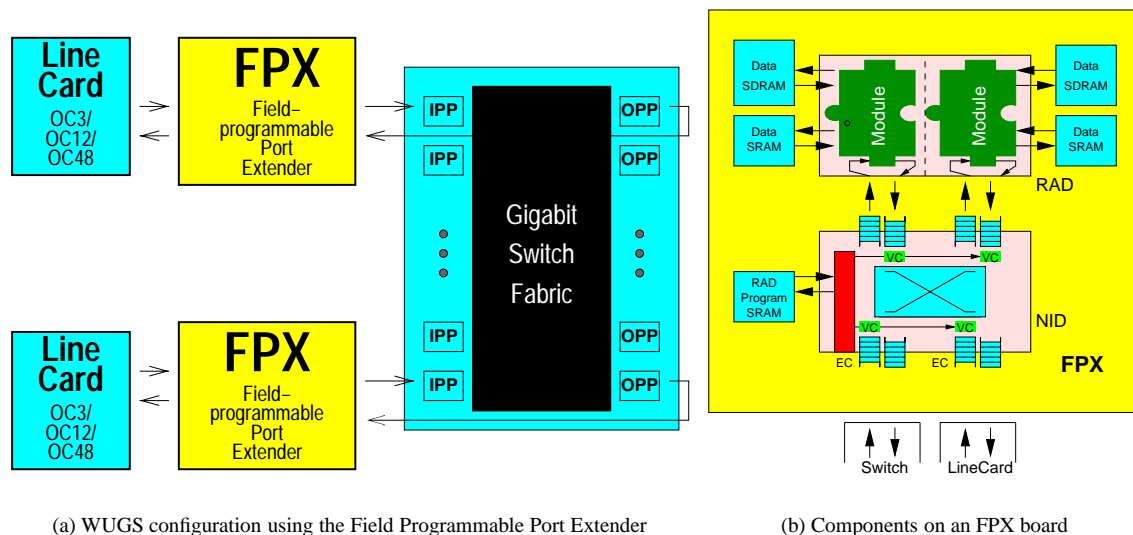


Figure 1: Components on an FPX board

3 Network Wrapper Concept

Network protocols are organized in layers. On the ATM data link layer, data is sent in fixed size cells. To provide variable length data exchange, a family of ATM Adaption Layers exists. Section 3.4.1 gives an introduction to the ATM Adaption Layer 5 (AAL5), which is generally used to transport IP data over ATM networks. The Network layer uses IP packets to support routing through multiple, physically separated networks.

Components have been developed for the FPX that allow applications to handle data on several protocol layers. Similar circuits have been implemented in static systems to implement IP over Ethernet [7]. Unlike systems that offload protocol processing to a co-processor [8], this library allows all packet processing functions to be implemented on the same chip.

Translation steps are necessary between layers. A classical approach creates components for each protocol translation. Such a system would instantiate one entity to translate data from the cell level to the AAL5 frame level. Such an implementation would also need to have a component to perform the reverse step as well. In our approach, we combine these two translation units into one component, which has four interfaces as a consequence: two to support the lower level protocol and two to provide a higher level interface, respectively. Also the two components are connected to each other. This is useful to exchange additional information or to bypass the application. The latter is done in the cell processor (section 3.2).

When an application module is embedded into a protocol wrapper, the new entity surrounds the user's logic like the letter U (figure 2). Regarding the data stream, the application only connects to the translating component, which wraps up the application itself. Therefore we will refer to the surrounding components as *wrappers*.

To support higher levels of abstraction, the wrappers can be nested. Since each has a well defined interface for an outer and an inner protocol level, they fit together within each other, as shown in figure 2. As a result, we get a very modular design method to support applications for different protocols and levels of abstraction. Associating each wrapper with a specific protocol, we get a layer model comparable to the well known OSI/ISO networking reference model. This modularity gives application developers freedom to implement functions at several protocol layers in their designs. They can interface their logic to a wrapper with the level of abstraction appropriate for their specific application. User-level applications, for example, can completely ignore handling of complicated protocol issues, like frame boundaries or checksums.

3.1 Wrapper Installation

There are two ways to use the wrapper library in a project.

1. A distribution file can be downloaded (REF!), which contains an EDIF- and a VHDL-file for every wrapper. The EDIF-file can be used for synthesizing a design, while the VHDL-file is for behavioral simulation. Except for the cell processor, a VHDL-file exists for every wrapper that combines a processor with all underlying wrappers for convenient inclusion in a new design. A description of the files in the distribution file is shown in Table 1. In order to simulate the wrappers correctly, the simulator must be configured to run in nanosecond resolution. For the Modelsim simulator this can be set when starting the program with the -t flag:

```
> vsim -t ns <design>
```

2. All wrappers are included in the FPX CVS tree. A working directory tree can be checked out in the ARL network with the command

```
> cvs -d /project/arl/fpx/cvsroot co FPX_ROOT
```

The wrappers are located below the subdirectory “RAD/MODULES/LIB” and can be compiled in a Modelsim library with the command

```
> make wrapper_lib
```

The library is then available under “RAD/MODULES/LIB/fplib”. In order to simulate the wrappers correctly with the library, design must use configurations and refer to the wrapper configurations as in the examples in Appendix A. The configuration names of the wrappers are *cellproc.conf* for the cell processor, *frameproc.conf* for the frame processor, *ipproc.conf* for the IP processor and *udpproc.conf* for the UDP processor.

3.2 Cell based processing

At the lowest level of abstraction, data is sent in fixed length cells. Applications or wrappers at this protocol level typically process the ATM header and filter cells by their virtual channel.

3.2.1 Control cells

The behavior of an FPX module can be modified via control cells. These are ATM cells with a well-defined structure and provide a communication path between an external controller (e.g. software) and the on-chip modules. This can be used to remotely configure modules, for example. Control cells contain an opcode field, multiple parameter fields, a sequence number, and a 16-bit CRC, to ensure data integrity (figure 3). They are sent on specific virtual channels, defaulting to VCI 34 for the NID and VCI 35 for the RAD.

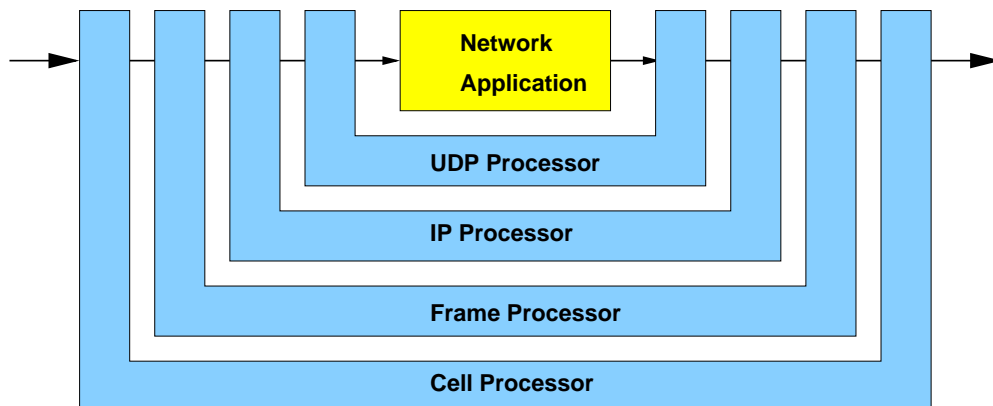


Figure 2: Wrapper concept

cellprocessor.edf	synthesized cell processor
cellprocessor_sim.vhd	VHDL file to simulate cell processor
frameproc.edf	synthesized frame processor
frameproc_sim.vhd	VHDL file to simulate frame processor
framewrapper.vhdl	frame wrapper combines cell and frame processor
ipproc.edf	synthesized IP processor
ipproc_sim.vhd	VHDL file to simulate IP processor
ipwrapper.vhdl	IP wrapper combines frame wrapper and IP processor
udpproc.edf	synthesized UDP processor
udpproc_sim.vhd	VHDL file to simulate UDP processor
udpwrapper.vhdl	UDP wrapper combines IP wrapper and UDP processor

Table 1: Files in the wrapper distribution

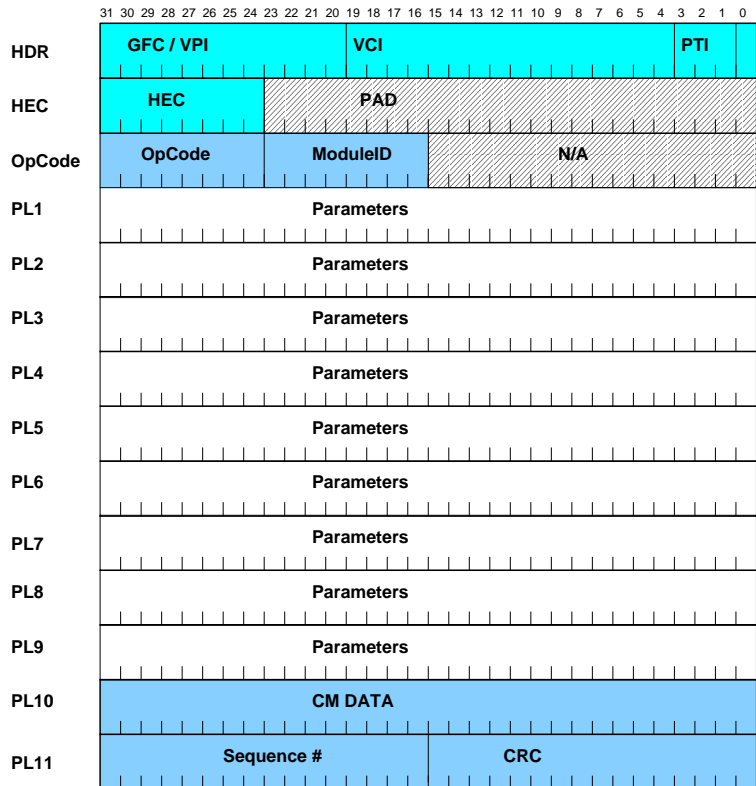
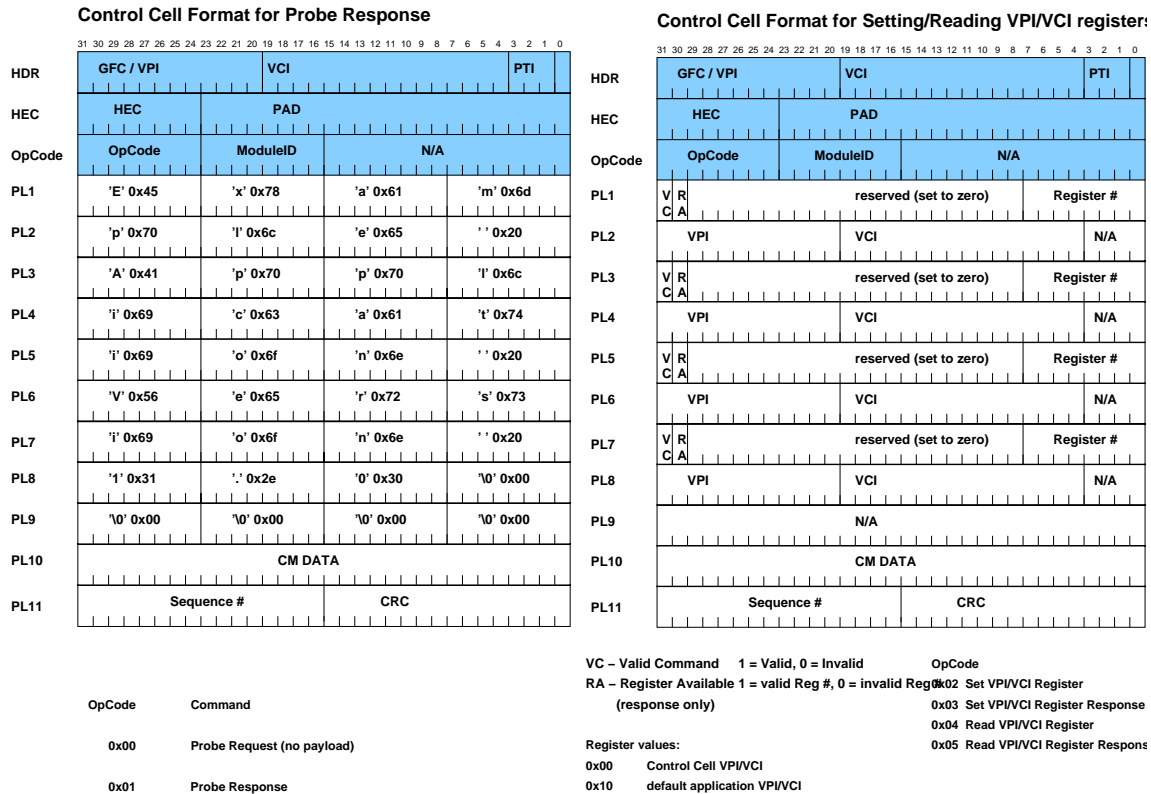


Figure 3: Control cell format



(a) Probe Response

(b) Set/Read VPI/VCI Registers

Figure 4: Standard Control Cell Opcodes

A standard control cell format has been developed to transmit information between software [9] that controls the FPX. Control cells to the NID are used to setup the routes between the line card, the switch and user applications on the RAD. They are also used to upload new application modules to the RAD.

Control cells to the RAD contain an additional field, the module ID, to address the application module. Some standard opcodes are understood by all FPX modules. Commands to change the VPI/VCI registers, for example, are supported by all modules so that so that they can operate on any virtual channel. For FPX modules opcodes in the range 0x00 to 0x0F are allocated for common use, while opcodes from 0x10 to 0xFF can be used for user applications. The following opcodes have been defined for common usage:

1. Opcode 0x00 is used as a “Probe Request”. Applications response with the “Probe Response” (0x01) and an identification string. This mechanism is used by the control software to query the configuration on the FPX. The control cell structure can be seen in figure 4(a).
2. Opcodes 0x02 and 0x04 are used to setup and query VPI/VCI registers. Applications can be configured to operate on any virtual channel by writing to one of these registers. Registers in the range from 0x00 to 0x0F are again allocated for common usage, while register numbers from 0x10 to 0xFF can be used by user applications. Register 0x00 defines the virtual channel on which control cells are sent, while 0x10 should be used as the application’s default channel. The control cell structure can be seen in figure 4(b).

The control cell processor (CCP) is a standard FPX module with the hardcoded module ID of zero. It is connected to all four off-chip memories and can modify the content of off-chip memory. It is useful for applications, which implement functionality such as lookup tables in off-chip memory.

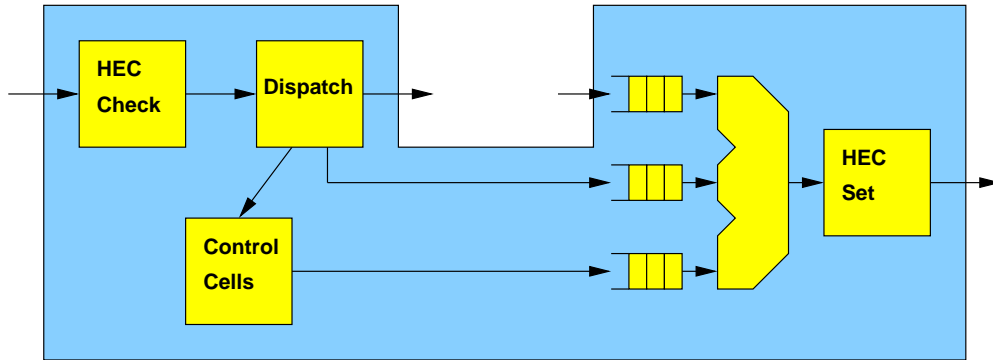


Figure 5: Cell processor

3.2.2 FPX Cell Processor

The wrapper on the lowest level is the cell processor (figure 5). It performs cell level operations that are common to all FPX modules. First, incoming ATM cells are checked against their Header Error Control (HEC) field, which is part of the 5 octet header. An 8 bit CRC is used to prevent errored cells from being misrouted. If the check fails, the cell is dropped.

Cells that are accepted are next processed by their virtual channel information. The cell processor distinguishes between three different flows:

1. The cell is on the data VC for this module. In this case, the cell will be forwarded to the inner interface of the wrapper and thus to the application.
2. The cell is on the control cell VC and is tagged with the correct module ID. Control cells are processed by the cell processor itself. This mechanism is covered later in this section.
3. None of the above, i.e. this cell is not destined for this module. These cells are bypassed and take a shortcut to the output of the cell processor.

The cell processor provides three FIFOs to buffer cells from either of the three paths. A multiplexer combines them and forwards the cells to their last stop. Just before they leave the cell processor, a new HEC is computed.

The control cell handling function inside the cell processor is designed to be very flexible, thus making it easy for application developers to extend its functionality to fit the needs of their modules. User applications typically support more control cell opcodes than the standard codes, so extensibility was an important goal in the design of the cell processor.

The control cell processing framework performs CRC check and generation functions, buffering of common data structures, and implements a mechanism to share common information. A master state machine waits for control cells destined for this module and then stores opcode, user data, the CM field and sequence number. At the same time it also checks the control cell CRC. Every opcode has its own state machine. So adding a new command does not interfere with existing ones. Every state machine polls the signal *state*, if a control cell with a valid CRC (*crc_ok = 1*) has been read (*state = CRC*) and becomes active on its opcode. For incoming control cells (requests), response cells should be sent, if the command has been processed successfully. Because there is a state machine for every opcode, that generates response cell, a multiplexer forwards the correct one to the output port. Every process for a control cell opcodes sends its response cells out on *data_XX* and indicates the start on *soc_XX*, where *XX* is the opcode. The process *ccselection* checks all *soc_XX* signals and forwards new control cells. They get a valid CRC before they are forwarded to the cell multiplexer. The currently selected opcode is presented in *opcode_sel*. Processes should check this signal to make sure their cell has been transmitted. A diagram for the cell processing framework can be seen in figure 6.

The process *sm00* is responsible for detecting Probe Requests (0x00), while the process *data00_out* generates the Probe Response cell. The default string is "Generic Cell Processor 1.0". To change the Response string, applications need only modify this string.

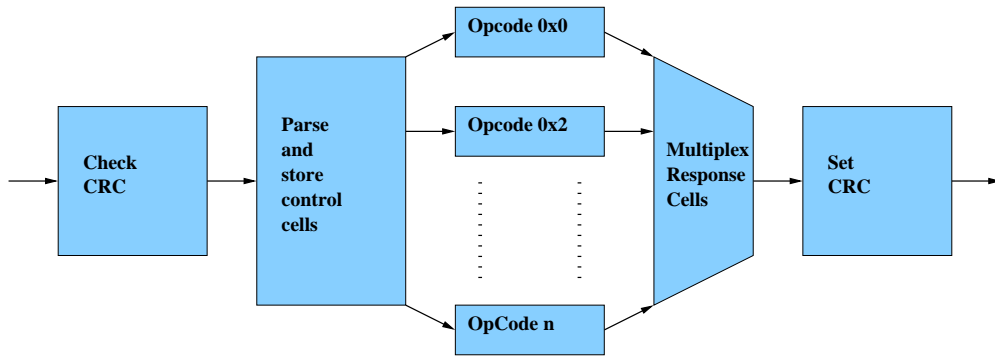


Figure 6: Control Cell Handling

cptestapp.vhdl	test application
module_cpctest.vhdl	connects application with cell processor
cellprocessor_sim.vhd or library from CVS	structural/behavioral description of cell processor

Table 2: Files to simulate cell processor application

The processes *sm02* and *sm04* are responsible for setting (0x02) and reading (0x04) VPI/VCI registers. The response cell is generated by *data04_out* in both cases. For opcode 0x02 the values are written to the registers just before they are read again by *sm04* for the response. Since the register values for the acknowledgment are always read from the actual register, this is a good mechanism to check if the write operation was successful. To support additional registers these processes need to be changed.

3.3 Example Application

Appendix A.1 lists the VHDL source code for an example application. It shows how an application interconnects with the cell processor and how a RAD design can be set up. The application itself simply forwards data, but can be easily replaced with any other module, that follows the FPX module interface specification.

For simulation the files shown in Table 2 are needed. The cell processor accepts a probe request (figure 7) and replies with a probe response (figure 8) in a simulator. For synthesis the files from Table 3 are needed.

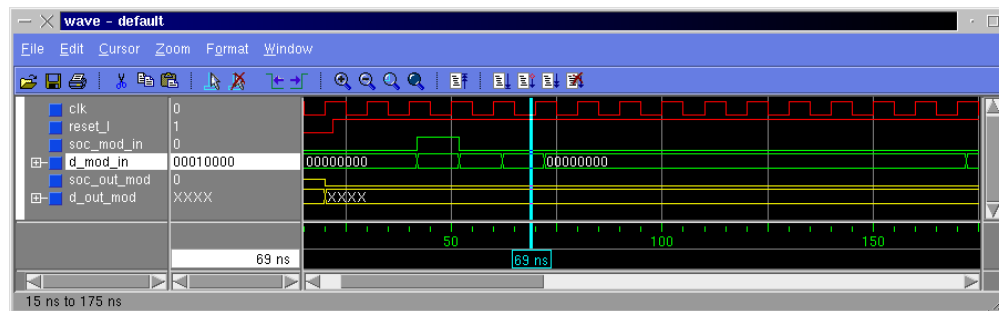


Figure 7: Simulation input for cell processor application

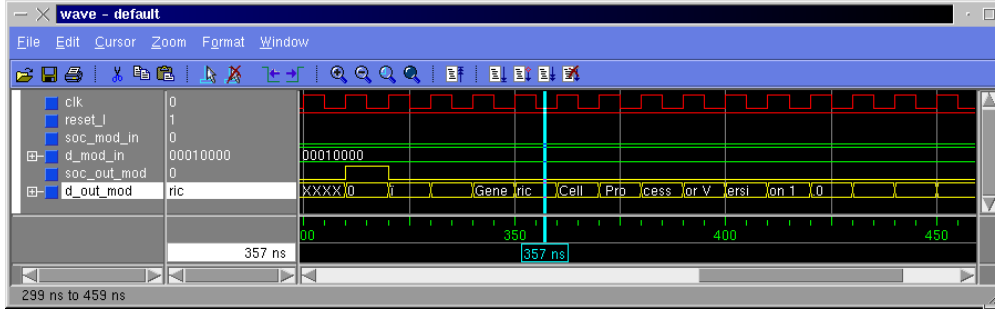


Figure 8: Simulation output for cell processor application

cptestapp.vhdl	test application
module_cptest.vhdl	connects application with cell processor
cellprocessor.edf	synthesized cell processor
loopback_module.vhd	loopback module for unused path
rad_cptest_core.vhd	instantiates modules on RAD
rad_cptest.vhd	top level design with extra buffers

Table 3: Files to synthesize cell processor application

3.4 Frame based processing

To handle data with arbitrary length over ATM networks, data is organized in frames, which are sent as multiple cells. Several adaption layers have been specified, which differ in the property of being connection-oriented or connection-less, in the ability to multiplex several protocols over one virtual channel and to reorder cells during transmission.

3.4.1 ATM Adaption Layer 5

ATM Adaption Layer 5 (AAL5) [10, 11] is widely used for IP networks and is also one of the simpler protocols. In AAL5 datagrams or frames of arbitrary length are put into protocol data units (PDU). In the simplest case (Flow Type 0), the frame starts at the beginning of the PDU, but it is also possible to prepend an additional header to distinguish between several protocols. The implementation for this modules is based on classical IP over ATM [12], which does not have this header. A PDU's length is always a multiple of 48 octets, because a PDU is sent as a multiple of ATM cells. One bit in the ATM header, the user bit of the PTI field, is used to indicate whether a cell is the last one of a PDU. The last 8 octets of the PDU are used by a trailer, which contains information about the actual length of the frame and a 32 bit CRC to ensure data integrity. Any gap between the frame and the trailer is filled with padding. Since PDUs are multiples of 48 octets, the trailer always ends at a cell boundary and can therefore be located. The segmentation of frames with AAL5 is illustrated in figure 9.

3.4.2 FPX Frame Processor

The frame processor is a wrapper module for the FPX to handle AAL5 frame data. Its interface is designed to give application modules a more abstract view of the data. The frame processor replaces the Start-of-Cell signal with three signals (figure 15, namely Start-of-Frame (SOF), End-of-Frame (EOF) and Data-Enable (DataEn).

As the name indicates, SOF indicates the transmission of a new frame. Note that the Header-Error-Control (HEC) is not available with this wrapper. It is assumed that only valid ATM cells are passed to this wrapper and that valid HECs are generated from outgoing cells.

The Data-Enable signal indicates valid payload data. It can be seen as an enable signal for the data processing application. It is completely independent from the cell structure. Applications can therefore resize frames or append data easily. They can also generate new frames. Note that the Data-Enable signal is not asserted when padding is sent,

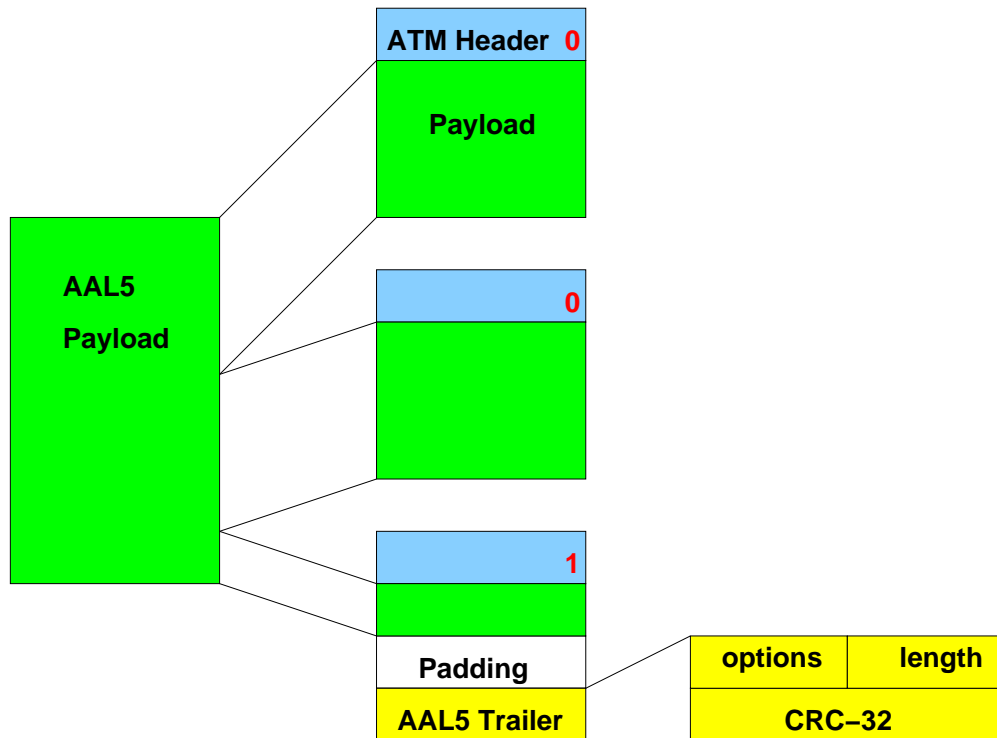


Figure 9: AAL5 frame segmentation

since it is not considered a part of the frame. The End-of-Frame signal is asserted with the last valid payload word being sent. Applications thus have enough time to start appending data to a frame, if necessary.

After the EOF signal, two more words are sent, while DataEnable is still '1'. These 8 octets represent the AAL5 trailer, but the interpretation is different. The first word contains the length and the option field. While the option field is simply copied to the new frame, the length field is not taken literally. The actual length of the frame can be determined from the three signals SOF, EOF and DataEn. Since the FPX uses 32 bits data width internally, it is only accurate to 4 octets, though. Thus the lower two bits of the length field are used to determine the valid octets within one 32 bit word. These two bits must be set by the application if the frame length changes by a number of octets not divisible by 4. The high-order bits of this field are changed by the frame processor according to the actual length of the frame. The second word is used to handle data integrity, i.e., the CRC-32. The frame processor handles the CRC-32 of AAL5 frames for the application. This happens right after cells enter the wrapper and just before they leave. The application sees an all-zero word if the frame is correct, otherwise the CRC field is replaced with a non-zero value. It is essential that applications copy and forward the two additional words.

The frame processor maintains a simple state machine to track frame boundaries and to generate the Start-of-Frame and End-of-Frame markers. Recall that only one bit is used to mark the last cell of a frame. At first glance, it may seem simple to generate Data-Enable – it would just assert the signal during the last 12 words of each cell, i.e., 48 payload octets. But if it was done that way, padding would also be recognized as valid data. Instead, the frame processor buffers the last cell of a frame and waits for the length field before it forwards that cell. In fact, only the last two words of every cell but the last have to be deferred.

On the output side of the processor, data is accumulated in a buffer, until either the size of one cell has been reached or the total size of the frame has been determined. The cells are then sent out. Since the ATM header is only given once together with the SOF signal, it is copied and prepended to all generated cells, while the user bit of the PTI field is set appropriately to indicate the last cell.

3.4.3 CRC issues

AAL5 frames are secured with a CRC-32 sent with the last cell. The CRC protects the payload data from transmission errors and also detects dropped cells.

Recall that there is no sequence number for cells in AAL5. Usually the data integrity is checked before anything is passed to applications. In case the check gives a negative result, the frame is being dropped. This approach requires buffering of the whole frame, before the CRC can be checked. That results in a long delay before data can be forwarded to the application. For outbound cells data can be forwarded immediately, since the (new) CRC is simply appended to the frame.

The FrameProcessor replaces the CRC field with an indication, to whether the packet is erroneous. For outbound data the frame processor computes a new CRC for the (possibly) new payload. Under normal circumstances, outgoing frames will have a correct CRC. For erroneous frames the indication will flip some bits of the computed CRC and thus invalidate the frame. Thus, the receiving node can still detect errors and ignore the frame. With this approach the wrapper can speed up frame processing and save buffer resources in the normal cases, while still detect in transmission errors.

3.5 IP Packet Processing

The Internet Protocol is a very popular communication means across several networks. Sub-protocols, built above IP, including UDP or TCP are used to send datagrams and establish reliable connections, respectively.

3.5.1 The FPX IP Processor

The IP processor was developed to support Internet Protocol based applications. It inherits the signalling interface from the frame processor and adds a Start-of-Payload (SOP) signal, to indicate the payload after the IP header, which can be of variable length. This wrapper serves three primary functions:

1. It checks the IP header integrity, to verify the correctness of the header checksum. Corrupt packets are dropped.
2. It decrements the Time To Live (TTL) field. As of RFC 1812 [13] all IP processing entities are required to decrement this field. Once this field reaches zero, the packet should not be forwarded any more. This is to prevent packets from looping around in networks due to mis-configured routers.
3. It recomputes the length and the header checksum on outgoing IP packets.

An IP header usually has the length of 20 bytes, or 5 words.¹ The whole header must be processed by a checksum circuit before any decision about its integrity can be made. The IP Processor computes and then compares the header checksum. On a failure, the IP-packet is dropped by not propagating any signal to the application. If the Time-To-Live field of an incoming packet is already zero, the packet is also dropped and an ICMP packet is sent instead. Otherwise the TTL field is decremented by one. On outgoing IP packets the length field in the header and the header checksum are set accordingly. Therefore a whole packet has to be buffered, before it can be sent out.

To save and share resources with other wrappers, the IP wrapper understands a protocol to update the contents of bytes sent earlier in the packet. The IP processor can apply changes to the packet payload for fields, such as in headers, that were sent when the packet originally streamed through the hardware. Update commands are optional and are inserted between the last payload word (EOF signal asserted hi) and the AAL5 trailer. An unused bit (15) in the AAL5 length field is used to indicate update words or the start of the trailer. The length field is also used to hold an error code, so that packets can be dropped before they are sent out. Update words contain a 16 bit update field and a 15 bit update offset address. The 16 bit word at the offset address in the buffer is replaced by the update field. The format of these field is illustrated in figure 10.

3.5.2 The FPX UDP Processor

The UDP processor is a wrapper to support user datagrams over IP. This wrapper computes and replaces the UDP checksum and the length field in the header for outgoing datagrams. Incoming datagrams are also checked for the checksum, but the result is only available after the whole packet has passed through the wrapper. The UDP processor

¹This applies to the vast majority of IP packets that do not contain any IP options.

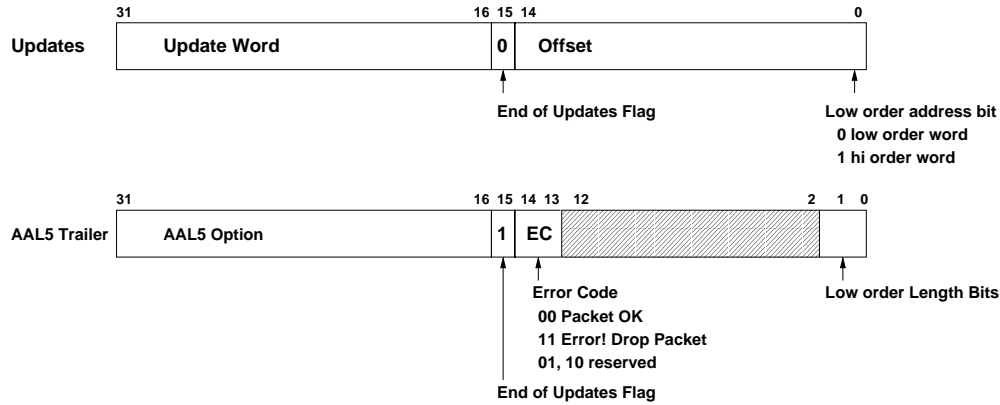


Figure 10: IP update protocol

Wrapper/Module	Space		Speed	Delay(s)		Delay(l)		Throughput(s)		Throughput(l)	
	LUTs	rel	(MHz)	in	out	in	out	rel.	max	rel.	max
Cell Processor	781	3%	125	4	6	4	6	100%	3.5	100%	3.5
Frame Processor	1251	5%	116	21	22	10	31	84%	2.7	93%	3.0
IP Processor	1009	4%	109	36	39	24	197	84%	2.6	93%	2.9
UDP Processor	550	2%	114	39	44	27	202	84%	2.6	93%	2.9

Table 4: Implementation results of the wrappers

uses similar signals as the IP processor. It replaces the SOP signal with the Start-of-Datagram (SOD) signal. Applications can simply process datagrams or even generate new ones without the need to interpret or generate UDP headers.

To determine the correct checksum for outgoing datagrams, the whole packet is buffered. Since the IP processor already buffers a full IP packet, it would have been a waste of on-chip resources to implement an additional buffer just for the UDP wrapper. Instead, the UDP processor informs the IP processor about necessary updates in the packet and leaves the buffering to that wrapper.

4 Implementation Results

Our framework is designed for the FPX. The system clock on the FPX is 100 MHz and the FPGA used is a Xilinx Virtex 1000E-7. Table 4 summarizes the size (in lookup tables and relative to RAD chip-size) and the maximum speed of our components. It also gives some information about the delays of data passed through the wrappers and the throughput, which can be achieved. These results have been measured by sending UDP packets of one cell-size (s) and with 512 bytes of payload (l). Delays are split up into delays before (in) and after (out) an embedded application.

5 IP Testbench Tool

To simplify simulation and testing of new hardware applications a set of tools has been developed to help developers creating test data. The toolkit is called the “IP testbench”, because it was originally meant to help in developing an IP application, but it is easily extendible and supports already more protocols.

The tools parse files that are in a simple format and contain descriptions of ATM cells, AAL5 frames, IP packets or UDP datagrams, i.e. data on different protocol levels. This data can be converted in any of the other protocols. For example, a developer can specify that he wants an IP packet and the tools compute the IP header, the AAL5 frame and segment that into multiple ATM cells. The tools save this information in either its own file format, a simulation file for Modelsim or a data file, which can be read from a VHDL testbench.

The tools can also read the results from a simulation, which have been written by a VHDL component. This is useful to verify that an application produces correct results.

5.1 Installation

The tools can be downloaded from LINK!. They are also in the FPX CVS repository and can be checked out from within the ARL network with the command

```
> cvs -d /project/arl/fpx/cvsroot co iptestbench
```

The tools can be compiled by typing “make” on the command line prompt.

5.2 Program descriptions

The IP testbench contains several programs to convert data from and to several file formats. Most of the tools work on the TBP-format (Test-Bench-Pattern), which will be explained in section 5.3. The programs *ip2raw*, *ip2sim* and *ip2fake* are the most often used tools. They read a file in TBP-format and convert all higher level protocols to raw-data, which is simply the data that is sent to an FPX application and consists of 14 32-bit words:

- the first word contains the ATM header
- the second word contains the Header Error Control (HEC) in the highest byte.
- the last 12 words contain the ATM payload (48 bytes).

The three programs differ only in the output file’s format:

- *ip2raw* writes an TBP-formatted output file.
- *ip2sim* writes a macro file that can be included and executed from Modelsim’s *vsim*.
- *ip2fake* writes a data format that can be read from the *fake_NID* component inside a VHDL testbench.

The *fake2raw* program is the only program which reads a different file format than TBP. It reads the output from a VHDL component inside a testbench and converts that format to TBP, so that it can be processed from other *iptestbench* programs.

The tool *raw2ip* reads a file in TBP format and converts all lower level protocols (usually raw) to the highest possible protocol level (IP or UDP). It is usually used in combination with *fake2raw*.

The *iptb* program is the most generic tool and is capable of converting all protocol levels in any other and also writes results in every file format. It takes at least one argument which is a string that describes what conversion should be performed. The characters ‘-’ and ‘+’ in the conversion string specify, in what direction the tools should convert the data. The ‘-’ flag will convert data to lower level protocols, while the ‘+’ flag regenerates higher level protocol data, respectively. An overview is printed when the argument “- -help” is given:

```
> iptb --help
usage: iptb [infile [outfile]] convertstring
convertstring uses the following characters
- down-convert mode
+ up-convert mode
: special options mode
convert-mode characters:
a ATM cells
f AAL5 frames
c control cells
4 IPv4
6 IPv6
i 4 and 6 combined
A all options in reasonable order
```

```

special-mode characters:
  r  raw output
  s  modelsim .do output
  f  fake-module file output
examples:
  iptb i.tbp o.tbp -A      convert to raw cells
  iptb i.tbp o.tbp -A:s   same, but output for vsim
  iptb i.tbp o.tbp +af    regenerate aal5 frames

```

The IP testbench tools are written in C. A description of the internal data structures and functions can be found in Appendix C. So new functionality can be added to cover more simulation scenarios.

5.3 The TBP file format

A TBP (TestBench-Pattern) file contains blocks of user data. Each block belongs to a certain protocol and can define additional attributes. The file format is line-oriented. Every line can be either empty or describe one of the following:

- Comments are ignored by the parser and start with a '#' on the first column.
- Block delimiters define a new data block and start with a '!', followed by a keyword describing the protocol and optional arguments.
- User data is given in 32 bit words per line and is written in hexadecimal values.

If the block length doesn't match the required length of a certain protocol, zero words are appended or the data is truncated. For example, the following file describes an ATM cell on VCI 20, which contains the words 0xAAAAAAAA and 0xBBBBBBBB followed by 10 zeros.

```

# example for an ATM cell
!CELL 20
aaaaaaaaa
bbbbbbbbbb

```

A description of all defined block types is given below, including a description of the optional arguments.

PAD is used to insert breaks between blocks for simulation. The argument specifies the number of clock cycles to wait, i.e. multiples of 10 ns.

RAW is used to sent a raw block of data to an application. The simulator will give an SOC signal followed by the RAW data block. The first word usually defines the ATM header, the second contains the Header Error Control (HEC) in the highest byte. RAW has no additional arguments.

CELL defines an ATM cell. The data block will be used as the CELL payload. It is expanded or truncated to 12 words, if the data block has a different length. This block type accepts two arguments. The first defines the VCI, the second is a flag whether the PTI field should be set (end of AAL5-frame marker). CELL blocks are converted to RAW by preceding an ATM header and a HEC.

CTRLCELL defines a control cell as it is used to configure FPX modules. It accepts three arguments: the opcode, the module ID and the control cell VCI. CTRLCELL blocks are converted to CELL by generating the control cell header, a sequence number, and the CRC.

FRAME defines an AAL5 frame. The VCI can be given as an argument. FRAME blocks are converted to (multiple) cells by generating a AAL5-PDU, appending padding, the length field and the CRC.

IPv4 is used to specify IP version 4 packets. It accepts up to four arguments: the destination IP address (in dotted quad notation), the protocol number, the TTL value and the VCI. IPv4 packets are converted to FRAME by preceding the IP version 4 header, including the length field and the checksum.

UDP is used to sent UDP datagrams. The destination IP address and the destination and source port can be specified as arguments. UDP is converted to IPv4 with the protocol number 7 by preceding a UDP header, including length field an checksum, to the payload.

6 Conclusions

We have presented a framework for IP packet processing applications in hardware. Although our current implementation was created for use in the Field Programmable Port Extender, the framework is very general and can easily be adapted to other platforms. A library of Layered Protocol wrappers has been implemented. Each handles a particular protocol level. By using an entity that surrounds an application module (a U-shape wrapper), the related logic to convert to and from a protocol are linked, increasing the flexibility and reducing the number of cross-dependencies. The common interface between layers also lowers the learning curve.

The framework is useful for developers of networking hardware components. Applications themselves don't have to take care about network protocol issues. The complete IP processing framework only utilizes 14% of the RAD FPGA on the FPX, leaving sufficient space to implement user-defined logic.

Appendix A Example Applications

This section lists VHDL code of example applications.

A.1 Cell Processor Test

The following VHDL listings describe an example application that uses the cell processor from the wrapper library.

cptestapp.vhdl

This file contains a simple test application. It just copies the input signals to the output signals. This file can be replaced by any other application that follows the FPX module interface specification.

```
-- $Id: cptestapp.vhdl,v 1.1 2001/07/09 22:07:53 florian Exp $
--
-- CellProcTest
-- test application for the Cell Processor. This module just connects the
-- input with the output interface to pass data through.
--
-- Author: Florian Braun
-- (c) 2001 Washington University, Applied Research Lab
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library fpxlib;

entity CPTestApp is

    port (
        CLK           : in  std_logic;           -- clock
        Reset_l       : in  std_logic;           -- synchronous reset, active low
        Enable_l       : in  std_logic;           -- reprogramming handshake
        Ready_l        : out std_logic;           -- reprogramming handshake
        SOC_MOD_IN     : in  std_logic;           -- start of cell
        D_MOD_IN       : in  std_logic_vector (31 downto 0); -- data
        TCA_MOD_IN     : out std_logic;           -- congestion control
        SOC_OUT_MOD    : out std_logic;           -- start of cell
        D_OUT_MOD      : out std_logic_vector (31 downto 0); -- data
        TCA_OUT_MOD    : in  std_logic;           -- congestion control
    );

end CPTestApp;

architecture struc of CPTestApp is

begin -- struc

    -----
    -- purpose: buffer and copy data on clock edge
    -- type    : sequential
    -- inputs  : CLK, Reset_l
    -- outputs:
    data_copy_proc: process (CLK)
    begin -- process data_copy_proc
        if CLK'event and CLK = '1' then -- rising clock edge
            SOC_OUT_MOD <= SOC_MOD_IN;
            D_OUT_MOD   <= D_MOD_IN;
            TCA_MOD_IN  <= TCA_OUT_MOD;
        end if;
    end process;
end struc;
```

```

        Ready_1 <= not Enable_1;
    end if;
end process data_copy_proc;

end struc;

```

module_cptest.vhdl

This file interconnects the application with the cell processor. It can be used as a template for other applications as well.

```

-- $Id: module_cptest.vhdl,v 1.1 2001/07/09 22:07:53 florian Exp $
--
-- CellProcTest
-- test application for the Cell Processor. This module connects the
-- application with the cell processor.
--
-- Author: Florian Braun
-- (c) 2001 Washington University, Applied Research Lab
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
library fpxlib;

entity CPTestModule is

    port (
        CLK           : in  std_logic;           -- clock
        Reset_1       : in  std_logic;           -- synchronous reset, active low
        Enable_1       : in  std_logic;           -- reprogramming handshake
        Ready_1        : out std_logic;           -- reprogramming handshake
        SOC_MOD_IN     : in  std_logic;           -- start of cell
        D_MOD_IN       : in  std_logic_vector (31 downto 0); -- data
        TCA_MOD_IN     : out std_logic;           -- congestion control
        SOC_OUT_MOD    : out std_logic;           -- start of cell
        D_OUT_MOD      : out std_logic_vector (31 downto 0); -- data
        TCA_OUT_MOD    : in  std_logic;           -- congestion control
    );

end CPTestModule;

architecture struc of CPTestModule is

    -----
    -- cell proc
    -----

    component CellProc
    port (
        CLK           : in  std_logic;
        Reset_1       : in  std_logic;
        Enable_1       : in  std_logic;
        Ready_1        : out std_logic;
        SOC_MOD_IN     : in  std_logic;
        D_MOD_IN       : in  std_logic_vector (31 downto 0);
        TCA_MOD_IN     : out std_logic;
        SOC_OUT_APPL  : out std_logic;
        D_OUT_APPL    : out std_logic_vector (31 downto 0);
    );

```

```

    TCA_OUT_APPL : in  std_logic;
    SOC_APPL_IN  : in  std_logic;
    D_APPL_IN    : in  std_logic_vector (31 downto 0);
    TCA_APPL_IN  : out std_logic;
    SOC_OUT_MOD  : out std_logic;
    D_OUT_MOD    : out std_logic_vector (31 downto 0);
    TCA_OUT_MOD  : in  std_logic);
end component;

-----
-- test application
-----

component CPTestApp
port (
    CLK          : in  std_logic;
    Reset_l     : in  std_logic;
    Enable_l    : in  std_logic;
    Ready_l     : out std_logic;
    SOC_MOD_IN  : in  std_logic;
    D_MOD_IN    : in  std_logic_vector (31 downto 0);
    TCA_MOD_IN  : out std_logic;
    SOC_OUT_MOD : out std_logic;
    D_OUT_MOD   : out std_logic_vector (31 downto 0);
    TCA_OUT_MOD : in  std_logic);
end component;

-----
-- input signals
-----

signal soc_in   : std_logic;           -- start of cell
signal data_in  : std_logic_vector (31 downto 0); -- data
signal tca_in   : std_logic;           -- tca

-----
-- output signals
-----

signal soc_out  : std_logic;           -- start of cell
signal data_out : std_logic_vector (31 downto 0); -- data
signal tca_out  : std_logic;           -- tca

-----
-- ready signals
-----

signal Ready_cp : std_logic;
signal Ready_test : std_logic;

begin -- struc

-----
-- instantiate cell proc
-----

cp: CellProc
port map (
    CLK          => CLK,
    Reset_l     => Reset_l,
    Enable_l    => Enable_l,
    Ready_l     => Ready_cp,

    SOC_MOD_IN  => SOC_MOD_IN,

```

```

D_MOD_IN      => D_MOD_IN,
TCA_MOD_IN    => TCA_MOD_IN,

SOC_OUT_APPL => soc_in,
D_OUT_APPL   => data_in,
TCA_OUT_APPL => tca_in,

SOC_APPL_IN  => soc_out,
D_APPL_IN    => data_out,
TCA_APPL_IN  => tca_out,

SOC_OUT_MOD  => SOC_OUT_MOD,
D_OUT_MOD    => D_OUT_MOD,
TCA_OUT_MOD  => TCA_OUT_MOD);

-----
-- instantiate test application
-----
test: CPTestApp
  port map (
    CLK          => CLK,
    Reset_l     => Reset_l,
    Enable_l     => Enable_l,
    Ready_l     => Ready_test,
    SOC_MOD_IN  => soc_in,
    D_MOD_IN    => data_in,
    TCA_MOD_IN  => tca_in,
    SOC_OUT_MOD => soc_out,
    D_OUT_MOD   => data_out,
    TCA_OUT_MOD => tca_out);

-----
-- misc connections
-----
Ready_l <= not (Ready_cp and Ready_test);

end struc;

configuration cptest_conf of CPTestModule is

  for struc

    for all : CellProc
      use configuration fpxlib.cellproc_conf;
    end for;

  end for;

end cptest_conf;

```

loopback_module.vhdl

This module just connects the input signals with the output signals and is used to pass data through the unused path on the RAD.

```

-- Applied Research Laboratory
-- Washington University in St. Louis
--
-- File: loopback_module.vhd

```

```

-- Entity declaration for RAD module
-- Created by: David E. Taylor (det3@arl.wustl.edu)
-- Created on: August, 16 2000
-- Last modified: August 18, 2000 @ 11:20 am
--
-- IMPORTANT: Refer to RAD Module Interface Specification
-- for explanations and timing specifications for all interface
-- signals.
--
library IEEE;
use IEEE.STD_LOGIC_1164.all;
--
ENTITY loopback_module IS
  PORT (
    -- Clock & Reset
    clk      : in STD_LOGIC;           -- 100MHz global clock
    reset_l  : in STD_LOGIC;           -- Synchronous reset, asserted-low

    -- Cell Input Interface
    soc_mod_in  : in STD_LOGIC;         -- Start of cell
    d_mod_in    : in STD_LOGIC_VECTOR(31 downto 0); -- 32-bit data
    tca_mod_in  : out STD_LOGIC;        -- Transmit cell available

    -- Cell Output Interface
    soc_out_mod : out STD_LOGIC;         -- Start of cell
    d_out_mod   : out STD_LOGIC_VECTOR(31 downto 0); -- 32-bit data
    tca_out_mod : in STD_LOGIC;         -- Transmit cell available
    test_data   : out STD_LOGIC_VECTOR(31 downto 0) -- 32-bit data
  );
end loopback_module;

architecture behavioral of loopback_module is

  signal soc : std_logic;           -- Start of Cell
  signal data : std_logic_vector(31 downto 0); -- 32-bit words of ATM cell
  signal tca : std_logic;           -- Transmit Cell Available
  --
begin -- behavioral
  --
  -- Pass cells through. On reconfig, hold TCA low.
  --
  CELL_IO_FF : process (clk)
  begin -- process CELL_FF
    if (clk='1' and clk'event) then
      if reset_l='0' then
        soc <= '0';
        soc_out_mod <= '0';
        data <= (others=>'0');
        d_out_mod <= (others=>'0');
        tca <= '0';
        tca_mod_in <= '0';
      else
        soc <= soc_mod_in;
        data <= d_mod_in;
        soc_out_mod <= soc;
        d_out_mod <= data;
        tca <= tca_out_mod;
        tca_mod_in <= tca;
      end if;
    end if;
  end process;
end architecture;

```

```

        end if;
    end process CELL_IO_FF;
    --
    test_data <= data;
    --
end behavioral;

```

rad_cptest_core.vhd

This file combines all modules on the RAD. The test application sits within the ingress path, the egress path is passed through the loopback module.

```

-- Applied Research Laboratory
-- Washington University in St. Louis
--
-- File: rad_loopback_core.vhd
-- Top level structure for RAD FPGA with ingress/egress loopback modules
-- Created by: John W. Lockwood (lockwood@arl.wustl.edu),
-- David E. Taylor (det3@arl.wustl.edu)
--
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
--library unisim;
--
ENTITY rad_loopback_core IS
    PORT (
        -- Clocks
        RAD_CLK  : IN STD_LOGIC;
        RAD_CLKB : IN STD_LOGIC;

        -- Reset & Reconfig
        RAD_RESET : IN STD_LOGIC;
        RAD_READY : OUT STD_LOGIC;
--    RAD_RECONFIG : INOUT STD_LOGIC_VECTOR(2 DOWNTO 0);

        -- NID Interface
        --
        -- Ingress Path
        -- Input
        SOC_LC_NID  : IN  STD_LOGIC;
        D_LC_NID    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        TCAFF_LC_RAD : OUT STD_LOGIC;
        -- Output
        SOC_LC_RAD  : OUT STD_LOGIC;
        D_LC_RAD    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        TCAFF_LC_NID : IN  STD_LOGIC;

        -- Egress Path
        -- Input
        SOC_SW_NID  : IN  STD_LOGIC;
        D_SW_NID    : IN  STD_LOGIC_VECTOR(31 DOWNTO 0);
        TCAFF_SW_RAD : OUT STD_LOGIC;
        -- Output
        SOC_SW_RAD  : OUT STD_LOGIC;
        D_SW_RAD    : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
        TCAFF_SW_NID : IN  STD_LOGIC;

        -----
-- note: SRAM and SDRAM interface has been removed for this example

```

```

-----
-- Test Connector Pins
RAD_TEST1 : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
RAD_TEST2 : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);

-- Test LED Pins
RAD_LED1 : OUT STD_LOGIC;
RAD_LED2 : OUT STD_LOGIC;
RAD_LED3 : OUT STD_LOGIC;
RAD_LED4 : OUT STD_LOGIC
);
END rad_loopback_core;
-----
architecture structure of rad_loopback_core is
--
-- Component Declarations
--
component loopback_module
port (
clk          : in  STD_LOGIC;
reset_l     : in  STD_LOGIC;
soc_mod_in  : in  STD_LOGIC;
d_mod_in    : in  STD_LOGIC_VECTOR(31 downto 0);
tca_mod_in  : out STD_LOGIC;
soc_out_mod : out STD_LOGIC;
d_out_mod   : out STD_LOGIC_VECTOR(31 downto 0);
tca_out_mod : in  STD_LOGIC;
test_data   : out STD_LOGIC_VECTOR(31 downto 0)
);
end component;
--
component CPTestModule
port (
CLK          : in  std_logic;
Reset_l     : in  std_logic;
Enable_l    : in  std_logic;
Ready_l     : out std_logic;
SOC_MOD_IN  : in  std_logic;
D_MOD_IN    : in  std_logic_vector (31 downto 0);
TCA_MOD_IN  : out std_logic;
SOC_OUT_MOD : out std_logic;
D_OUT_MOD   : out std_logic_vector (31 downto 0);
TCA_OUT_MOD : in  std_logic);
end component;
--
component blink
port (
clk1       : IN  STD_LOGIC;
clk2       : IN  STD_LOGIC;
reset_l    : IN  STD_LOGIC;
led1       : OUT STD_LOGIC;
led2       : OUT STD_LOGIC);
end component;
--
-- Signal Declarations
--
signal ingress_test, egress_test : std_logic_vector(31 downto 0); -- test pin data
signal logic0, logic1 : std_logic;    -- Vss and Vdd
--

```

```

begin -- structural
--
rad_ready <= NOT(rad_reset);
logic0    <= '0';
logic1    <= '1';

-- rad_reconfig(2) <= NOT(rad_reset);
--
-- Test Pin Flops
TEST_PIN_FF : process (RAD_CLK)
begin -- process TEST_PIN_FF
    if RAD_CLK'event and RAD_CLK = '1' then -- rising clock edge
        rad_test2 <= egress_test(31 downto 16);
        rad_test1 <= egress_test(15 downto 0);
--
        rad_led3 <= rad_reset;
        rad_led4 <= not rad_reset;
    end if;
end process TEST_PIN_FF;
--
-----
INGRESS : CPTestModule
port map (
    clk          => RAD_CLK,
    reset_l      => rad_reset,
    enable_l     => logic0,
    ready_l      => open,
    soc_mod_in   => soc_lc_nid,
    d_mod_in     => d_lc_nid,
    tca_mod_in   => tcaff_lc_rad,
    soc_out_mod  => soc_lc_rad,
    d_out_mod    => d_lc_rad,
    tca_out_mod  => tcaff_lc_nid);
--
    test_data    => ingress_test);
--
EGRESS : loopback_module
port map (
    clk          => RAD_CLKB,
    reset_l      => rad_reset,
    soc_mod_in   => soc_sw_nid,
    d_mod_in     => d_sw_nid,
    tca_mod_in   => tcaff_sw_rad,
    soc_out_mod  => soc_sw_rad,
    d_out_mod    => d_sw_rad,
    tca_out_mod  => tcaff_sw_nid,
    test_data    => egress_test);
--
BLINK1 : blink
port map (
    clk1        => RAD_CLK,
    clk2        => RAD_CLKB,
    reset_l     => rad_reset,
    led1        => rad_led1,
    led2        => rad_led2);
-----
end structure;

```

rad_cptest.vhd

This file is the top level design file and adds buffers to the cptest core.

```
-- Applied Research Laboratory
-- Washington University in St. Louis
--
-- File: rad_loopback.vhd
-- Top level structure for RAD FPGA with ingress/egress loopback modules
-- Created by: John W. Lockwood (lockwood@arl.wustl.edu),
-- David E. Taylor (det3@arl.wustl.edu)
--
```

```
-----
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
-- synthesis translate_off
LIBRARY unisim;
-- synthesis translate_on
-----
```

```
ENTITY rad_loopback IS
  PORT (
    -- Clocks
    RAD_CLK : IN STD_LOGIC;
    RAD_CLKB : IN STD_LOGIC;

    -- Reset & Reconfig
    RAD_RESET : IN STD_LOGIC;
    RAD_READY : OUT STD_LOGIC;
    RAD_RECONFIG : INOUT STD_LOGIC_VECTOR(2 DOWNTO 0);

    -- NID Interface
    --
    -- Ingress Path
    -- Input
    SOC_LC_NID : IN STD_LOGIC;
    D_LC_NID : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_LC_RAD : OUT STD_LOGIC;
    -- Output
    SOC_LC_RAD : OUT STD_LOGIC;
    D_LC_RAD : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_LC_NID : IN STD_LOGIC;

    -- Egress Path
    -- Input
    SOC_SW_NID : IN STD_LOGIC;
    D_SW_NID : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_SW_RAD : OUT STD_LOGIC;
    -- Output
    SOC_SW_RAD : OUT STD_LOGIC;
    D_SW_RAD : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_SW_NID : IN STD_LOGIC;
  );
END rad_loopback;
```

```
-----
-- note: SRAM and SDRAM interface has been removed for this example
-----
```

```
-- Test Connector Pins
RAD_TEST1 : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
RAD_TEST2 : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
```

```

-- Test LED Pins
RAD_LED1 : OUT STD_LOGIC;
RAD_LED2 : OUT STD_LOGIC;
RAD_LED3 : OUT STD_LOGIC;
RAD_LED4 : OUT STD_LOGIC
);
END rad_loopback;
-----
architecture structure of rad_loopback is
--
-- Component Declarations
--
COMPONENT IOBUF_F_12
  port (
    O : out  std_ulogic;
    I : in   std_ulogic;
    IO : inout std_logic;
    T : in   std_logic
  );
end COMPONENT;
--
COMPONENT BUFGDLL
  port ( O : out std_ulogic;
         I : in  std_ulogic
  );
end COMPONENT;
--
-- synthesis translate_off
for all : IOBUF_F_12 use entity unisim.IOBUF_F_12(IOBUF_F_12_V);
for all : BUFGDLL use entity unisim.bufgdl1(BUFGDLL_V);
-- synthesis translate_on
--
component rad_loopback_core
  port (
    RAD_CLK      : IN   STD_LOGIC;
    RAD_CLKB     : IN   STD_LOGIC;
    RAD_RESET    : IN   STD_LOGIC;
    RAD_READY    : OUT  STD_LOGIC;
    SOC_LC_NID   : IN   STD_LOGIC;
    D_LC_NID     : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_LC_RAD : OUT  STD_LOGIC;
    SOC_LC_RAD   : OUT  STD_LOGIC;
    D_LC_RAD     : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_LC_NID : IN   STD_LOGIC;
    SOC_SW_NID   : IN   STD_LOGIC;
    D_SW_NID     : IN   STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_SW_RAD : OUT  STD_LOGIC;
    SOC_SW_RAD   : OUT  STD_LOGIC;
    D_SW_RAD     : OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
    TCAFF_SW_NID : IN   STD_LOGIC;
    RAD_TEST1    : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    RAD_TEST2    : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0);
    RAD_LED1     : OUT  STD_LOGIC;
    RAD_LED2     : OUT  STD_LOGIC;
    RAD_LED3     : OUT  STD_LOGIC;
    RAD_LED4     : OUT  STD_LOGIC);
end component;
-----
--

```

```

-- Signal Declarations
--
signal rad_clk_dll      : std_logic;  -- DLL clock output
signal rad_clkb_dll    : std_logic;  -- DLL clock output
--
signal RAD_RESET_i     : STD_LOGIC;
signal RAD_READY_i     : STD_LOGIC;
signal SOC_LC_NID_i    : STD_LOGIC;
signal D_LC_NID_i      : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_LC_RAD_i  : STD_LOGIC;
signal SOC_LC_RAD_i    : STD_LOGIC;
signal D_LC_RAD_i      : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_LC_NID_i  : STD_LOGIC;
signal SOC_SW_NID_i    : STD_LOGIC;
signal D_SW_NID_i      : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_SW_RAD_i  : STD_LOGIC;
signal SOC_SW_RAD_i    : STD_LOGIC;
signal D_SW_RAD_i      : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_SW_NID_i  : STD_LOGIC;
signal RAD_TEST1_i     : STD_LOGIC_VECTOR(15 DOWNTO 0);
signal RAD_TEST2_i     : STD_LOGIC_VECTOR(15 DOWNTO 0);
signal RAD_LED1_i      : STD_LOGIC;
signal RAD_LED2_i      : STD_LOGIC;
signal RAD_LED3_i      : STD_LOGIC;
signal RAD_LED4_i      : STD_LOGIC;
--
signal RAD_RESET_pad   : STD_LOGIC;
signal RAD_READY_pad   : STD_LOGIC;
signal SOC_LC_NID_pad   : STD_LOGIC;
signal D_LC_NID_pad     : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_LC_RAD_pad : STD_LOGIC;
signal SOC_LC_RAD_pad   : STD_LOGIC;
signal D_LC_RAD_pad     : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_LC_NID_pad : STD_LOGIC;
signal SOC_SW_NID_pad   : STD_LOGIC;
signal D_SW_NID_pad     : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_SW_RAD_pad : STD_LOGIC;
signal SOC_SW_RAD_pad   : STD_LOGIC;
signal D_SW_RAD_pad     : STD_LOGIC_VECTOR(31 DOWNTO 0);
signal TCAFF_SW_NID_pad : STD_LOGIC;
signal RAD_TEST1_pad   : STD_LOGIC_VECTOR(15 DOWNTO 0);
signal RAD_TEST2_pad   : STD_LOGIC_VECTOR(15 DOWNTO 0);
signal RAD_LED1_pad     : STD_LOGIC;
signal RAD_LED2_pad     : STD_LOGIC;
signal RAD_LED3_pad     : STD_LOGIC;
signal RAD_LED4_pad     : STD_LOGIC;
-----
begin -- structural
rad_clk_dll  <= rad_clk;
rad_clkb_dll <= rad_clkb;

-- purpose: Double buffer all off-chip signals
IOB_Flops : process (rad_clk_dll)
begin -- process IOB_Flops
    if rad_clk_dll'event and rad_clk_dll = '1' then -- rising clock edge
        RAD_RESET_pad    <= RAD_RESET;
        RAD_READY         <= RAD_READY_pad;
        SOC_LC_NID_pad    <= SOC_LC_NID;
        D_LC_NID_pad      <= D_LC_NID;
    end if;
end process;
end begin;

```

```

TCAFF_LC_RAD      <= TCAFF_LC_RAD_pad;
SOC_LC_RAD        <= SOC_LC_RAD_pad;
D_LC_RAD          <= D_LC_RAD_pad;
TCAFF_LC_NID_pad <= TCAFF_LC_NID;
SOC_SW_NID_pad   <= SOC_SW_NID;
D_SW_NID_pad     <= D_SW_NID;
TCAFF_SW_RAD     <= TCAFF_SW_RAD_pad;
SOC_SW_RAD       <= SOC_SW_RAD_pad;
D_SW_RAD         <= D_SW_RAD_pad;
TCAFF_SW_NID_pad <= TCAFF_SW_NID;
RAD_TEST1        <= RAD_TEST1_pad;
RAD_TEST2        <= RAD_TEST2_pad;
RAD_LED1         <= RAD_LED1_pad;
RAD_LED2         <= RAD_LED2_pad;
RAD_LED3         <= RAD_LED3_pad;
RAD_LED4         <= RAD_LED4_pad;

```

--

```

RAD_RESET_i      <= RAD_RESET_pad;
RAD_READY_pad    <= RAD_READY_i;
SOC_LC_NID_i     <= SOC_LC_NID_pad;
D_LC_NID_i       <= D_LC_NID_pad;
TCAFF_LC_RAD_pad <= TCAFF_LC_RAD_i;
SOC_LC_RAD_pad   <= SOC_LC_RAD_i;
D_LC_RAD_pad     <= D_LC_RAD_i;
TCAFF_LC_NID_i   <= TCAFF_LC_NID_pad;
SOC_SW_NID_i     <= SOC_SW_NID_pad;
D_SW_NID_i       <= D_SW_NID_pad;
TCAFF_SW_RAD_pad <= TCAFF_SW_RAD_i;
SOC_SW_RAD_pad   <= SOC_SW_RAD_i;
D_SW_RAD_pad     <= D_SW_RAD_i;
TCAFF_SW_NID_i   <= TCAFF_SW_NID_pad;
RAD_TEST1_pad    <= RAD_TEST1_i;
RAD_TEST2_pad    <= RAD_TEST2_i;
RAD_LED1_pad     <= RAD_LED1_i;
RAD_LED2_pad     <= RAD_LED2_i;
RAD_LED3_pad     <= RAD_LED3_i;
RAD_LED4_pad     <= RAD_LED4_i;

```

end if;

end process IOB_Flops;

rad_loopback_core_1 : rad_loopback_core

port map (

```

RAD_CLK      => rad_clk_dll,
RAD_CLKB     => rad_clkb_dll,
RAD_RESET    => RAD_RESET_i,
RAD_READY    => RAD_READY_i,
SOC_LC_NID   => SOC_LC_NID_i,
D_LC_NID     => D_LC_NID_i,
TCAFF_LC_RAD => TCAFF_LC_RAD_i,
SOC_LC_RAD   => SOC_LC_RAD_i,
D_LC_RAD     => D_LC_RAD_i,
TCAFF_LC_NID => TCAFF_LC_NID_i,
SOC_SW_NID   => SOC_SW_NID_i,
D_SW_NID     => D_SW_NID_i,
TCAFF_SW_RAD => TCAFF_SW_RAD_i,
SOC_SW_RAD   => SOC_SW_RAD_i,
D_SW_RAD     => D_SW_RAD_i,
TCAFF_SW_NID => TCAFF_SW_NID_i,
RAD_TEST1    => RAD_TEST1_i,

```

```
RAD_TEST2    => RAD_TEST2_i,  
RAD_LED1     => RAD_LED1_i,  
RAD_LED2     => RAD_LED2_i,  
RAD_LED3     => RAD_LED3_i,  
RAD_LED4     => RAD_LED4_i);  
end structure;
```

Appendix B Interface Description

The interface of the wrappers is derived from the FPX module interface [4]. The common part consists of the signals *CLK*, *Reset_L*, *Enable_L* and *Ready_L*. The data-path interface is split into an inner and an outer interface. The outer interface uses an analog naming convention as for FPX modules, i.e. incoming signals are named *xxx_MOD_IN* (e.g. *SOC_MOD_IN*), while outgoing signals use *xxx_OUT_MOD* (e.g. *SOC_OUT_MOD*). The inner interface connects the wrapper to the application or a higher level wrapper. The naming convention here is *xxx_OUT_APPL* (e.g. *SOC_OUT_APPL*) for signals to the application, and *xxx_APPL_IN* (e.g. *SOC_APPL_IN*) for signals coming back into the wrapper. None of the wrappers connects to neither the SRAM nor the SDRAM controller. These interfaces can be used by the application.

The interfaces for all wrappers are shown in Figures 11–14. The signals are described below, while a timing diagram can be seen in figure 15.

CLK

This is the clock of the FPX (i.e. 100 MHz). All signals are synchronous to this clock.

Reset_L

This is the reset of the FPX. It is a synchronous, active low reset which is asserted low for one clock cycle.

Enable_L

This signal is used to enable the wrapper. See the RAD module interface documentation [4] for further details.

Ready_L

This signal performs the handshake in response to the *Enable_L* signal. After *Enable_L* is deasserted hi and after all buffers are flushed this signal is asserted low. See the RAD module interface documentation [4] for further details.

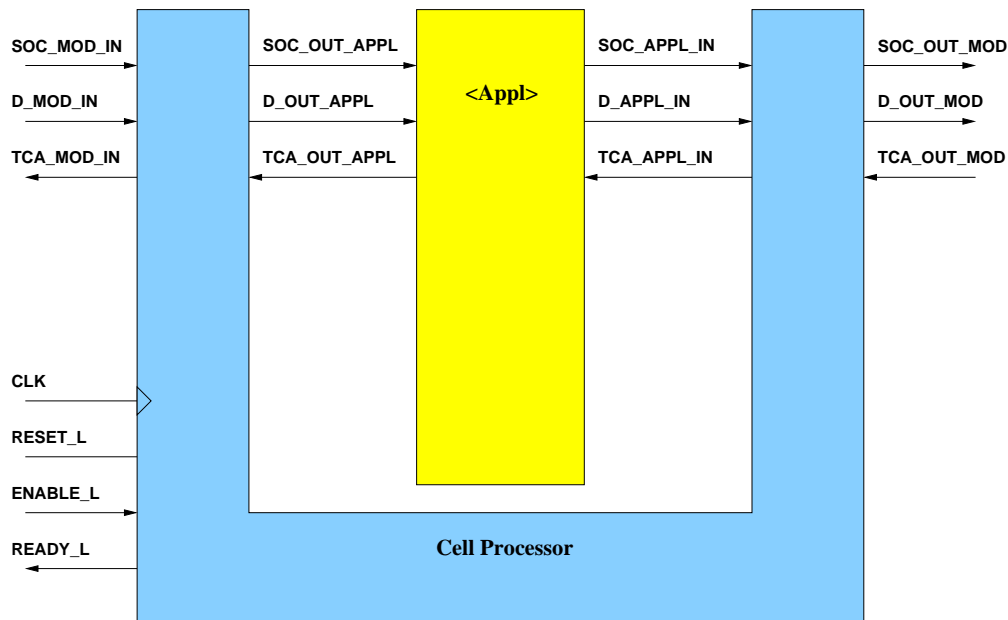


Figure 11: Cell Processor Interface

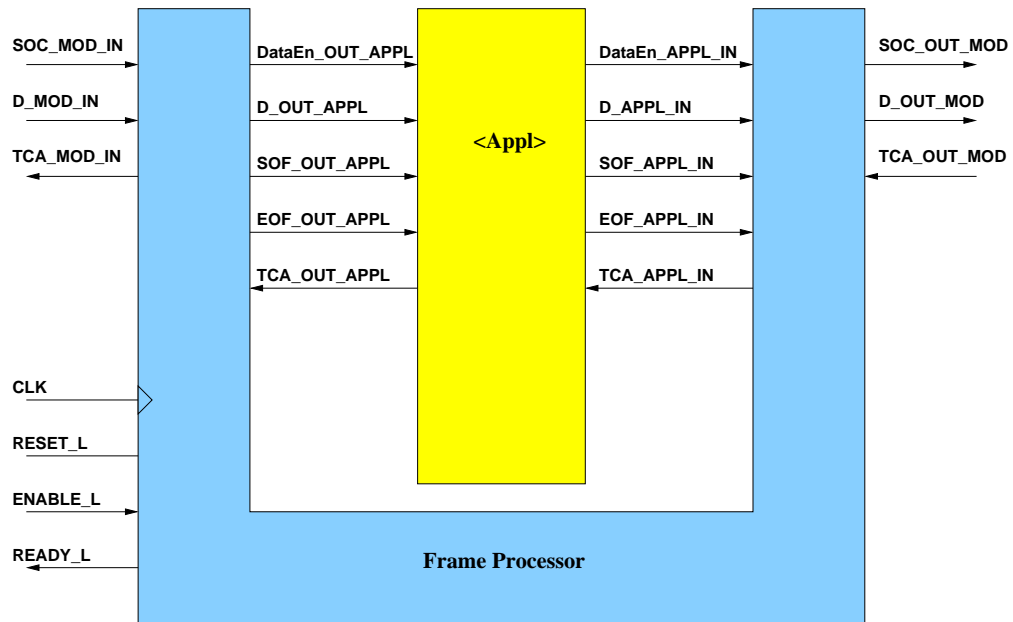


Figure 12: Frame Processor Interface

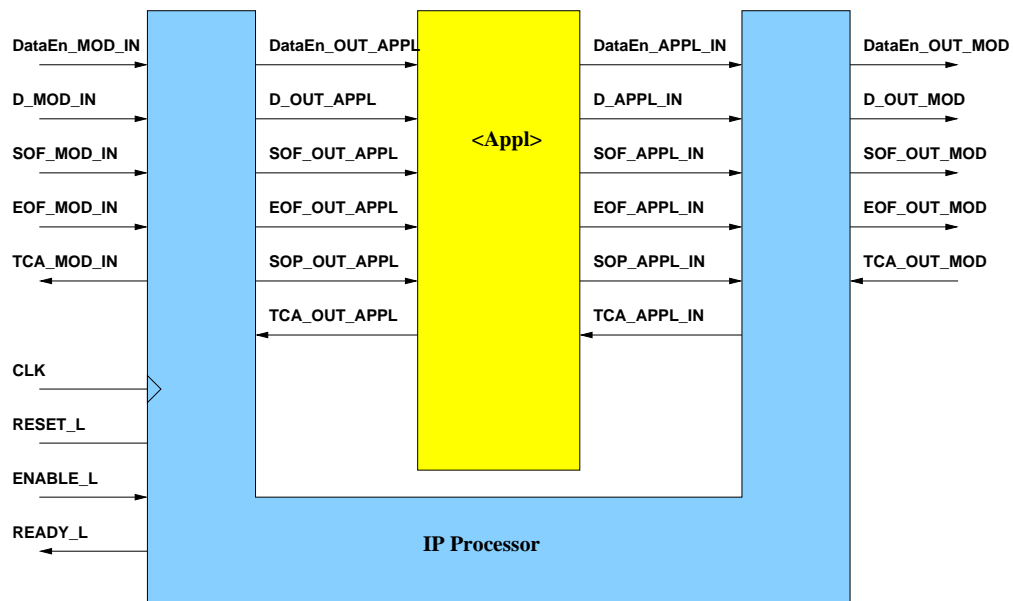


Figure 13: IP Processor Interface

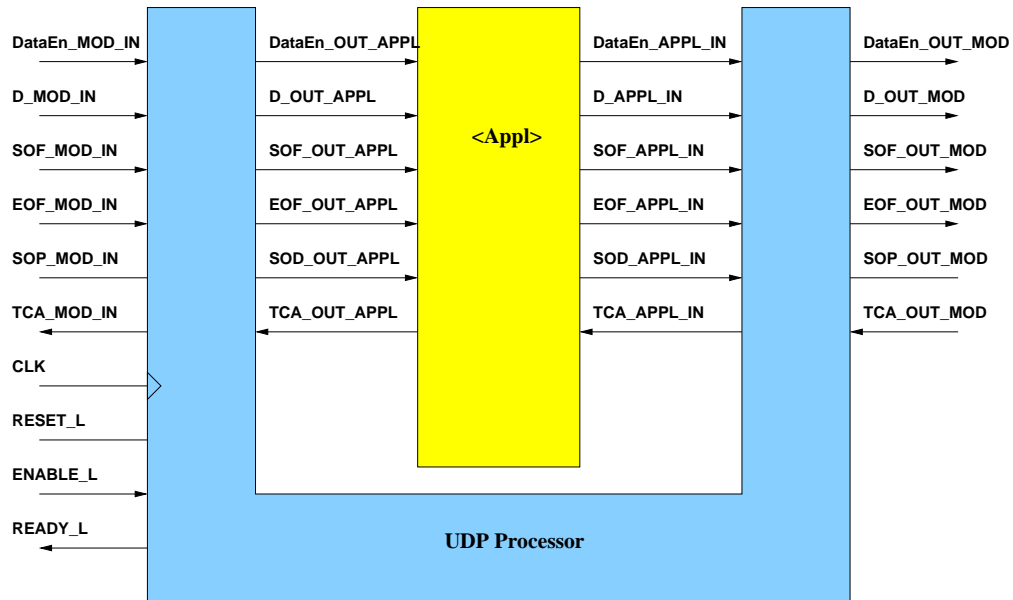


Figure 14: UDP Processor Interface

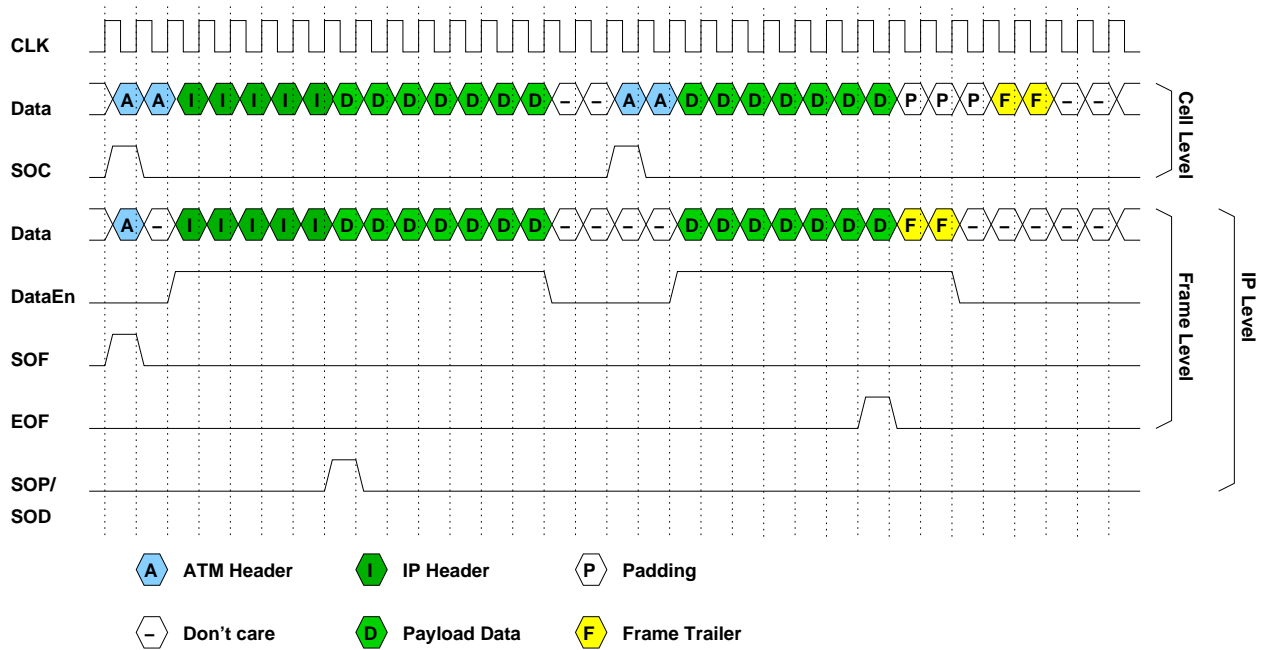


Figure 15: Timing diagram for different signalling protocols

D_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

This is a 32 bit databus which feeds the module with data. The data format is depending on the wrapper.

- The Cell Processor and the outer interface of the Frame Processor use a cell based data format. The first word of a cell is available when the SOC_XXX signal is asserted hi. A cell is exactly 14 words long (2 words header, 12 words payload). On each clock cycle one word is transmitted.
- The inner interface of the Frame Processor and all higher level wrappers are cell independent, i.e. valid data is only transmitted on this bus when the DataEn_XXX signal is asserted hi. The first word of the ATM header is available when the SOF_XXX signal is asserted hi (the DataEn_XXX signal is low then). After the EOF_XXX signal additional data is sent, which is described in the wrapper sections.

SOC_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *Start of Cell* signal is asserted hi if a new cell is transmitted through the D_XXX bus. The first word of the cell is available on the data bus on the same clock cycle as SOC_XXX is asserted. Otherwise this signal is set to lo. The ATM header is followed by the ATM HEC and 12 payload words.

SOF_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *Start of Frame* signal is asserted hi when a new frame is transmitted on the D_XXX bus. The ATM header of the first cell is available on the bus when this signal is asserted hi. Following this peak the DataEn_XXX signal indicates valid frame payload.

EOF_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *End of Frame* signal is asserted hi when the last payload word of a frame is sent. Following this signal usually two more words are sent on the bus: the option-/length-field and the CRC-field. For correct contents the CRC field is zero. The UDP Processor sends more update words to the IP Processor as described in section 3.5.

DataEn_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *Data Enable* signal indicates if data on D_XXX is valid payload of a frame or an IP packet. While payload is sent this signal is hi, otherwise it is lo. This signal is also hi for the frame trailer.

SOP_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *Start of Payload* signal is asserted hi when the first IP payload is transmitted on D_XXX. If the frame does not contain a valid IP packet this signal is not asserted at all.

SOD_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

The *Start of Datagram* signal is asserted hi when the first UDP header word is transmitted on D_XXX. If the frame is not an IP packet or the IP packet is not a UDP datagram this signal is not asserted.

TCA_MOD_IN, -OUT_APPL, -APPL_IN, -OUT_MOD

This signal performs the handshake back to the input for cell transfer. When this signal is asserted high, the input is free to send cells. Holding this signal low prevents the input from sending cells.

Appendix C Internal IP testbench structures

The IP testbench toolkit can be extended by new block types to support more simulation scenarios. The data flow inside the tools is usually as follows: a TBP file is read by a parser which stores the blocks inside the file in a single chained list of *StreamBlocks*. This list is processed by various filters, which convert blocks from one protocol into another. Finally the list is written to an output file.

All data structures are defined in the header file *testbench.h*. Blocks are organized in a *StreamBlock* structure (listing 1). This structure contains the block-type, a pointer to the payload data, a string which holds the arguments and a pointer to the next block. The functions *NewStreamBlock()* and *DeleteStreamBlock()* are used to create a new block or delete an unused block, respectively. Data-blocks can be created with *NewDataBlock()* and filled with *AddToDataBlock()*. Unused data blocks can be freed with *DeleteDataBlock()*. The initial block-list can be read from a file with *ParseFile()*.

The function *FilterStream()* applies a given filter to a block-list. A filter is a function which takes a *StreamBlock* as an input and returns a list of *StreamBlocks*. Filters convert blocks from one protocol into another by adding header and trailer data or verifying the structure of data. The filter type and all currently defined filters are listed in listing 2.

The final list can be written to a file in TBP format, Modelsim format or fake format using one of *DumpTB()*, *DumpSim()* or *DumpFake()*, respectively.

```
/* structure to store data */
typedef struct data_block_t {
    unsigned long *data;
    unsigned entries;
    unsigned size;
} DataBlock;

/* structure to organize stream */
typedef enum {NONE, ERROR, PAD,
             RAW, CELL, CTRLCELL, FRAME, IPv4, IPv6, UDP} BlockType;
typedef struct stream_block_t {
    BlockType btype; /* block type */
    DataBlock *data; /* data */
    char *options; /* options */
    struct stream_block_t *next; /* pointer to next block */
} StreamBlock;

/* functions to handle data */
DataBlock *NewDataBlock();
void AddToDataBlock (DataBlock *data, unsigned word);
void DeleteDataBlock (DataBlock *block);

/* functions to handle stream blocks */
StreamBlock *NewStreamBlock(BlockType btype);
void DeleteStreamBlock (StreamBlock *block);
```

Listing 1: Internal data structures in testbench.h

```

/* filter */
typedef StreamBlock* (*Filter)(StreamBlock*);
StreamBlock *FilterStream (StreamBlock*stream, Filter filter);
StreamBlock *FilterAll (StreamBlock*stream);
StreamBlock *ReassembleAll (StreamBlock *stream);

StreamBlock *UDPFiler(StreamBlock*);
StreamBlock *IPv4Filter(StreamBlock*);
StreamBlock *IPv6Filter(StreamBlock*);
StreamBlock *FrameFilter(StreamBlock*);
StreamBlock *CtrlCellFilter(StreamBlock*);
StreamBlock *CellFilter(StreamBlock*);

StreamBlock *RACellFilter(StreamBlock*);
StreamBlock *RACtrlCellFilter(StreamBlock*);
StreamBlock *RAFrameFilter(StreamBlock*);
StreamBlock *RAIPv4Filter(StreamBlock*);
StreamBlock *RAIPv6Filter(StreamBlock*);
StreamBlock *RAUDPFiler(StreamBlock*);

StreamBlock *TTLDecFilter(StreamBlock*);

```

Listing 2: Filters in testbench.h

References

- [1] Tom Chaney, J. Andrew Fingerhut, Margaret Flucke, and Jonathan S. Turner. Design of a gigabit ATM switch. Technical Report WU-CS-96-07, Washington University in Saint Louis, 1996.
- [2] John W. Lockwood, Jon S. Turner, and David E. Taylor. Field programmable port extender (FPX) for distributed routing and queuing. In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, pages 137–144, Monterey, CA, USA, February 2000.
- [3] John W. Lockwood, Naji Naufel, Jon S. Turner, and David E. Taylor. Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX). In *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, pages 87–93, Monterey, CA, USA, February 2001.
- [4] David E. Taylor, John W. Lockwood, and Naji Naufel. Generalized RAD Module Interface Specification of the Field-programmable Port eXtender (FPX). Technical report, WUCS-01-15, Washington University, Department of Computer Science, July 2001.
- [5] David E. Taylor, John W. Lockwood, and Naji Naufel. RAD Module Infrastructure of the Field-programmable Port eXtender (FPX). Technical report, WUCS-01-16, Washington University, Department of Computer Science, July 2001.
- [6] Toshiaki Miyazaki, Kazuhiro Shirakawa, Masaru Katayama, Takahiro Murooka, and Atsushi Takahara. A transmutable telecom system. In *Proceedings of Field-Programmable Logic and Applications*, pages 366–375, Tallinn, Estonia, August 1998.
- [7] Hamish Fallside and Michael J. S. Smith. Internet connected FPL. In *Proceedings of Field-Programmable Logic and Applications*, pages 48–57, Villach, Austria, August 2000.
- [8] Emmanuel A. Arnould, Francois J. Bitz, Eric C. Cooper, H. T. Kung, Robert D. Sansom, and Peter A. Steenkiste. The design of Nectar: A network backplane for heterogeneous multicomputers. In *Proceedings of the Third International conference on Architectural support for Programming Languages and Operating systems (ASPLOS-III)*, pages 205–216, 1989. Also available as Technical Report CMU-CS-89-101, School of Computer Science, Carnegie Mellon University, Pittsburgh.
- [9] Todd Sproull, John W. Lockwood, and David E. Taylor. Control and configuration software for a reconfigurable networking hardware platform. In *submitted to Globecom 2001*, San Antonio, TX, USA, November 2001.
- [10] B-ISDN ATM adaptional layer AAL Functional Description. CCITT: Recommendation I.362, 1991.
- [11] B-ISDN ATM adaptional layer AAL Specification. CCITT: Recommendation I.363, 1991.
- [12] Peter Newman et al. Transmission of flow labelled IPv4 on ATM data links. Internet RFC 1954, May 1996.
- [13] Fred Baker. Requirements for IP version 4 routers. Internet RFC 1812, June 1995.