Report Number: WUCS-01-14

2001-01-01

The FPX KCPSM Module: An Embedded, Reconfigurable Active Processing Module for the Field Programmable Port Extender (FPX)

Henry Fu and John W. Lockwood

While hardware plugins are well suited for processing data with high throughput, software plugins are well suited for implementing complex control functions. A plugin module has been implemented for the FPX that executes software on an embedded soft-core processor. By including this module in an FPX design, it is possible to implement active networking functions on the FPX using both hardware and software. The KCPSM, an 8-bit microcontroller developed by Xilinx Corp., has been embedded into an FPX module. The module includes circuits to be reprogrammed over the network and to execute new programs between the processing of data... **Read complete abstract on page 2.**

The FPX KCPSM Module: An Embedded, Reconfigurable Active Processing Module for the Field Programmable Port Extender (FPX)

Henry Fu and John W. Lockwood

Complete Abstract:

While hardware plugins are well suited for processing data with high throughput, software plugins are well suited for implementing complex control functions. A plugin module has been implemented for the FPX that executes software on an embedded soft-core processor. By including this module in an FPX design, it is possible to implement active networking functions on the FPX using both hardware and software. The KCPSM, an 8-bit microcontroller developed by Xilinx Corp., has been embedded into an FPX module. The module includes circuits to be reprogrammed over the network and to execute new programs between the processing of data packerts. A sample application, called the FPX KXPSM Module has been developed that illustrates how easily an application can make use of the hybrid system. This module loads the program memory of the KCPSM from an incoming UDP packet, and executes the new program upon receiving a new incoming UDP packet. The resulting circuit runs at 70MHz and occupies 35% on a Xilinx XCV1000E-7-FG680.

The FPX KCPSM Module: An Embedded, Reconfigurable Active Processing Module for the Field Programmable Port Extender (FPX)

Henry Fu and John W. Lockwood

WUCS-01-14

July 2001

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

The FPX KCPSM Module: An Embedded, Reconfigurable Active Processing Module for the Field Programmable Port Extender (FPX)

Henry Fu
John W. Lockwood

WUCS-01-14

July 26, 2001

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

Abstract

While hardware plugins are well suited for processing data with high throughput, software plugins are well suited for implementing complex control functions. A plugin module has been implemented for the FPX that executes software on an embedded soft-core processor. By including this module in an FPX design, it is possible to implement active networking functions on the FPX using both hardware and software. The KCPSM, an 8-bit microcontroller developed by Xilinx Corp., has been embedded into a FPX module. The module includes circuits to be reprogrammed over the network and to execute new programs between the processing of data packets. A sample application, called the FPX KCPSM Module has been developed that illustrates how easily an application can make use of the hybrid system. This module loads the program memory of the KCPSM from an incoming UDP packet, and executes the new program upon receiving a new incoming UDP packet. The resulting circuit runs at 70MHz and occupies 35% on a Xilinx XCV1000E-7-FG680.

Supported by: NSF ANI-0096052 and Xilinx Corp.

1 Introduction

A small, active, and reconfigurable processing node has been implemented as a module for the FPX [1] using the Protocol Wrappers [2] and the KCPSM [3]. The Protocol Wrappers process the incoming ATM cells and provide the module with valid UDP packets. The module then loads the contents of the packets into the program or data memory of the processor according to the first word of the payload. If the module takes in a program packet, the KCPSM will reset and run the new program after it has finished processing the current packet. If the module takes in a data packet, the KCPSM will process the new packet after it has finished processing all of the previous packets. When the module is not accepting any incoming packets, it will send the completed program and data packets back to the Protocol Wrappers. The Protocol Wrappers pack the completed packets into valid ATM cells. The FPX KCPSM Module is shown in Figure 1.

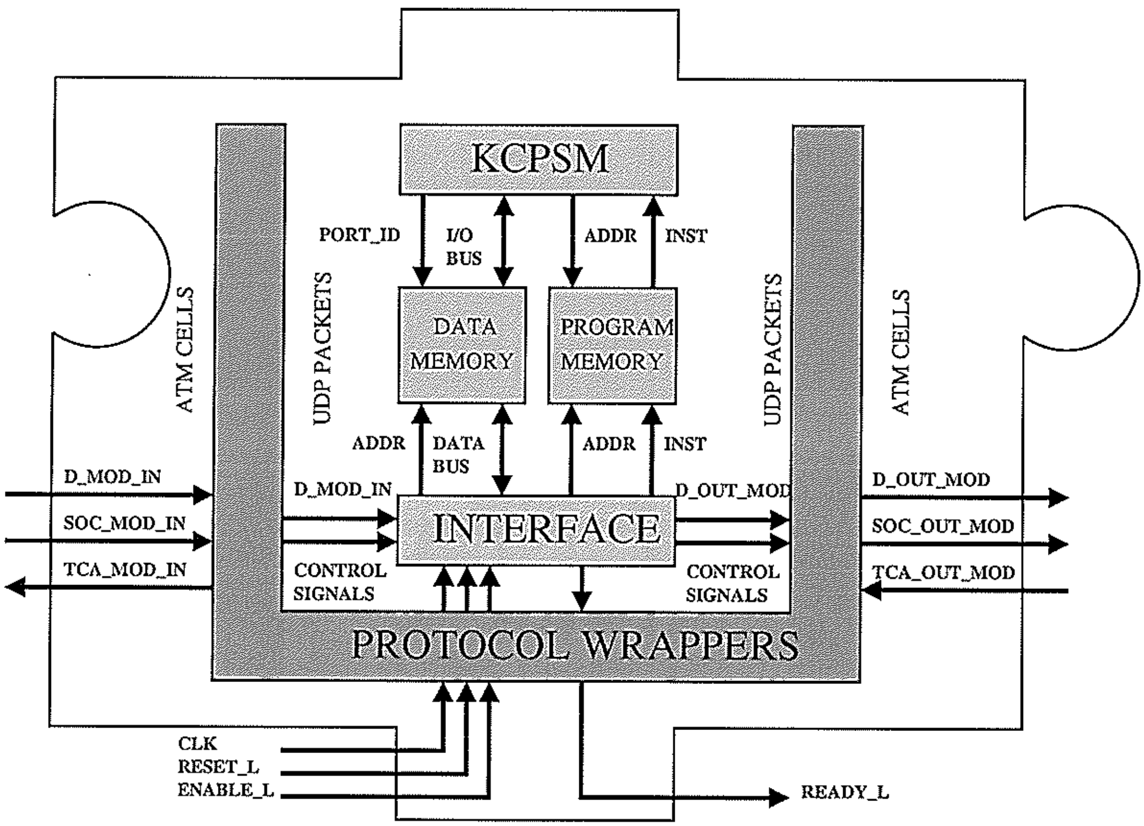

Figure 1: The FPX KCPSM Module

For this project, a circuit is implemented so that it allows the program memory of the KCPSM to be dynamically reprogrammed over the network through the use of UDP Datagrams. Therefore, the function of the processing module can be changed dynamically, on a packet by packet basis. This functionality is over and above the feature of dynamically reprogramming FPX hardware modules [4]. Currently, up to four packets with a maximum length of 256 bytes can be stored in the module for processing and up to two programs with a maximum length of 256 instructions can be stored in the module for execution. These limits can be easily extended for 32-bit softcore processors, such as the Microblaze [5].

2 Background of the FPX

The FPX (Field Programmable Port Extender) [6] [7] is a reprogrammable logic device that provides a hardware platform for the user to deploy network modules. It acts as an interface between the line cards and the WUGS (Washington University Gigabit Switch) [8], and it can be inserted between these two cards as shown in Figure 2.

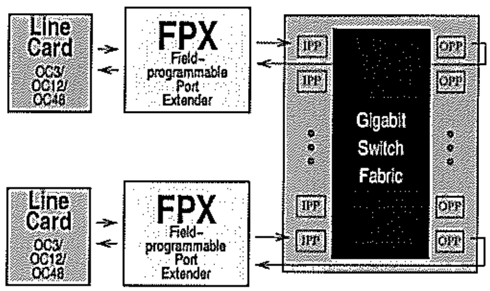

Figure 2: Configuration for the WUGS, FPX, and the Line Cards

The FPX is composed of two FPGAs: the Network Interface Device (NID) and the Reprogrammable Application Device (RAD) [9]. The NID interconnects the WUGS, the line cards, and the RAD, and provides the logic to dynamically reprogram the RAD. The RAD can be programmed to hold user-defined network modules, and is connected to two SRAM and two SDRAM components. Figure 3 shows the major components of the FPX.

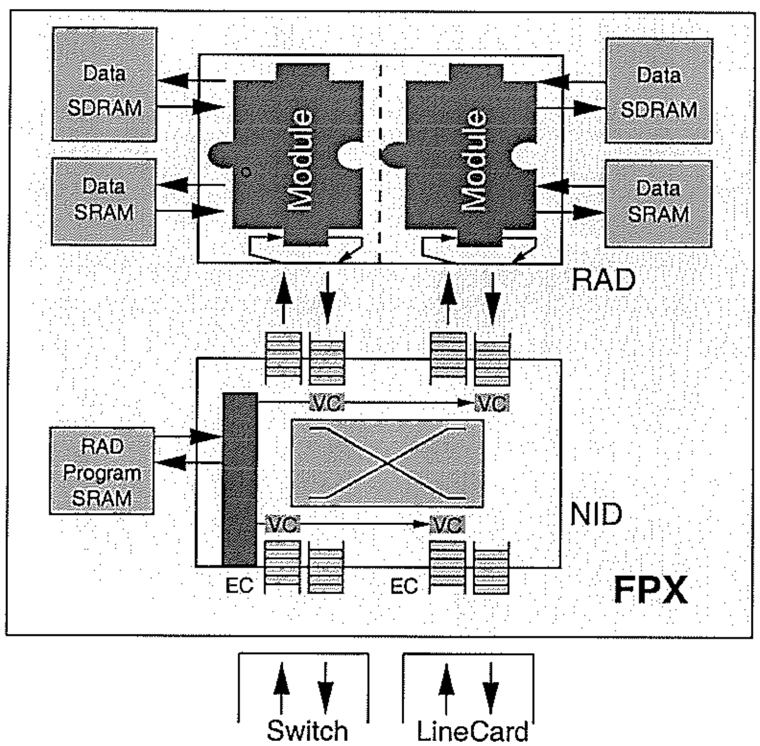

Figure 3: Major Components of the FPX

Although this project is primarily designed for and targeted to the FPX and the WUGS research environment, it is written in portable VHDL and can be used in any FPGA-based system.

3 Background of the KCPSM

The KCPSM (Constant (K) Coded Programmable State Machine) is a 8-bit microcontroller and takes only 35 CLBs in a FPGA. It provides 49 different instructions, 16 registers, 256 directly and indirectly addressable ports, and a maskable interrupt at 35 millions instructions per second (MIPS). It can be used in conjunction with an UART [10] to process serial input and output. Its functions and performance are adequate to process packets and control network traffic.

The KCPSM was developed by Ken Chapman of Xilinx Corp. and is designed for use with its Virtex and Spartan-II devices. It is provided in the form of an EDIF macro and can be embedded into any FPGA design. It also includes an assembler and debugger for writing and testing programs for the KCPSM. Two modules of dual-port memory are required in order to use the KCPSM. One dual-port memory acts as the program memory while the other acts as the data memory. Both are generated using the COREGEN program provided from Xilinx Corp. The block diagram of the KCPSM is shown in Figure 4.

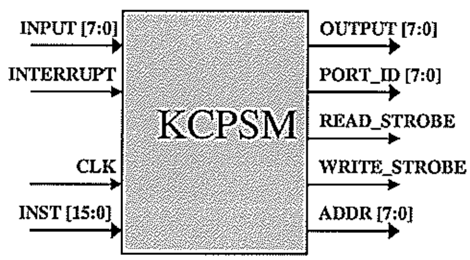

Figure 4: Block Diagram of the KCPSM

This document serves as a reference for implementing a softcore processor within a FPX module. Other softcore processors, such as the MicroBlaze [5] from Xilinx Corp. could be implemented in a similar manner.

4 Background of the Protocol Wrappers

The Protocol Wrappers [11] are used in the FPX KCPSM module to streamline and simplify the networking functions to process ATM cells, AAL5 frames, IP packets and UDP datagrams directly in hardware. It is a layered design and consists of different processing circuits in each layer; the block diagram of the Protocol Wrappers is shown in Figure 5. At the lowest level, the Cell Processor processes raw ATM cells between network interfaces. At the higher levels, the Frame Processor processes variable length AAL5 frames while the IP Processor processes IP packets. At the user level, the UDP Processor transmits and receives UDP messages. Different layers of abstraction is important for networks because it allows application to be implemented at a level where important details are exposed and irrelevant details are hidden. This way, an application that interacts with IP packets or an application that interacts with UDP messages can equally use the Protocol Wrappers effectively.

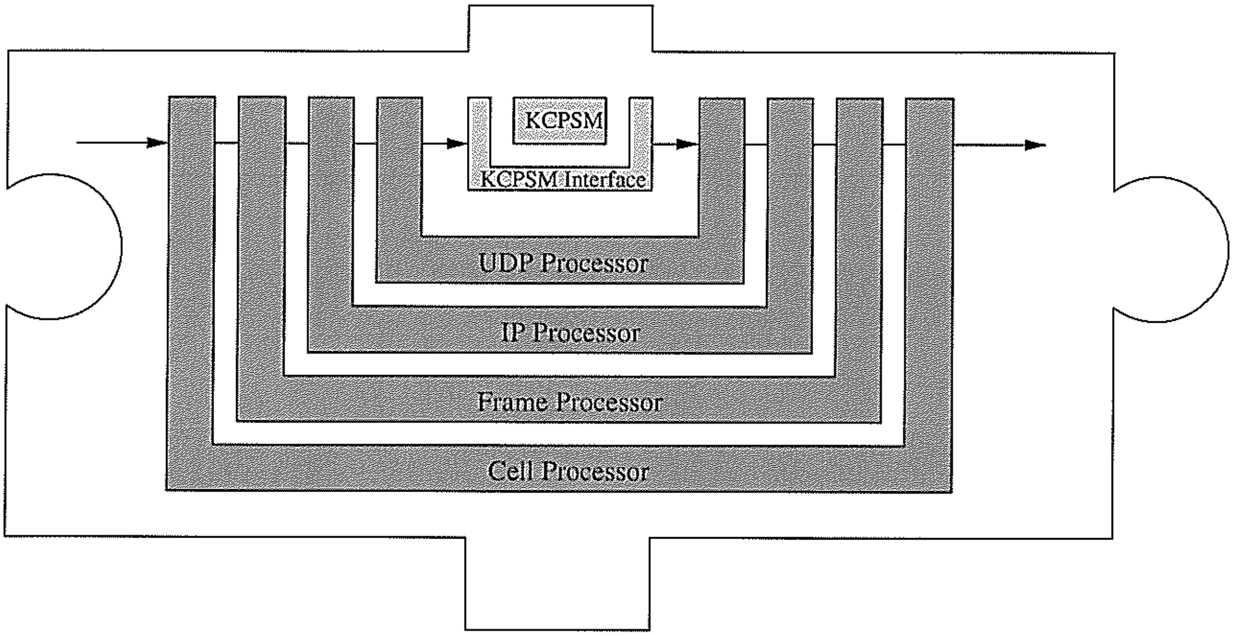

Figure 5: Block Diagram of the Protocol Wrappers.

For detailed instructions on how to use the Protocol Wrappers, please refer to the Protocol Wrappers webpage [12].

5 Implementation of the Interface

All operations of the FPX KCPSM module are controlled by an interface circuit that resides between the KCPSM and the Protocol Wrapper. It switches the source and destination IP address and port number, buffers the incoming UDP packets, checks if they are Data packets or Program packets, and stores them into the Data or Program Memory respectively. It also resets the KCPSM, echoes the Program packets, and writes the completed Data packets back to the sender. The overview of the Interface is shown in Figure 6.

5.1 Switching the Source and Destination IP Address and Port Number of the UDP Packets

The KCPSM Interface switches the source and destination IP address and port number of the UDP packets so that the unmodified program packet and the completed data packet can be echoed back to the sender.

5.2 UDP Packets FIFO Control

A FIFO buffers incoming UDP packets. Buffering of the data is required because the UDP Packets Store Control unit must wait for the results from the UDP Packets Type Check unit in order to determine whether to store the contents of the UDP packets into the program memory or the data memory.

5.3 UDP Packets Type Check

While the FIFO buffers incoming UDP packets, the UDP Packets Type Check unit checks them to see if they are Data packets or Program packets. It looks for the first word of the UDP payload; a 0x00000000

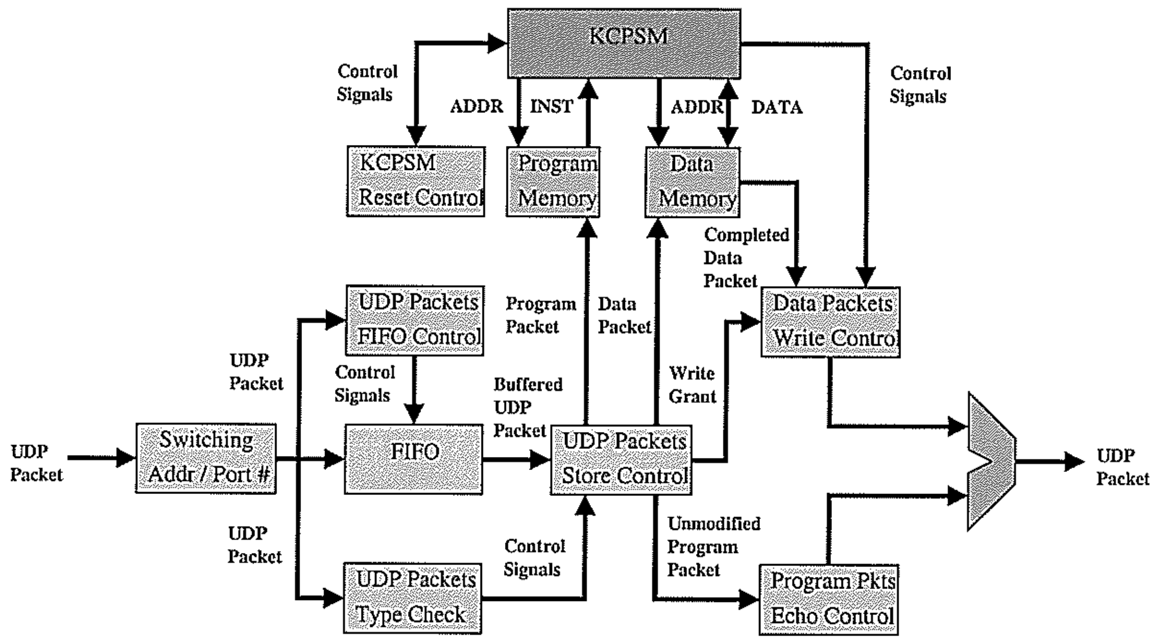

Figure 6: Overview of the KCPSM Interface

indicates a Program packet, and a 0x00000001 indicates a Data packet. Once it knows the type of the incoming packets, the UDP Packets Store Control unit can process them accordingly.

5.4 UDP Packets Store Control

The UDP Packets Store Control unit stores the Data packets and the Program packets differently. If they are Data packets, it will store the whole packets including the ATM, AAL5 Frame, IP and UDP headers, the UDP payload, and the ATM trailers into the Data Memory. If they are Program packets, it will only store the UDP payload, which is the KCPSM program, into the Program Memory. After it stores a Program packet, it will increment the program queue counter to indicate that a new program has just been loaded. It will also increment the bank counter of the Data Memory or the Program Memory accordingly so that the next Data packet or the next Program packet will store into the next bank of the Data Memory or the next bank of the Program Memory.

5.5 KCPSM Reset Control

The KCPSM Reset Control unit will only reset the KCPSM once the KCPSM has finished processing the current Data packet. It will first check if there is unprocessed Data packet stored in the Data Memory. If there is unprocessed data packet, it will increment the bank counter of the Data Memory so that the next Data packet stored in the Data Memory will be processed next. It will also check if there is a new program stored in the Program Memory. If there is no new program, it will just reset the KCPSM so that the KCPSM will process the next Data packet using the current program. If there is a new program, it will increment the bank counter of the Program Memory so that the KCPSM will process the next Data packet using the new program. In either case, the KCPSM Reset Control unit resets the KCPSM by asserting the Interrupt input of the KCPSM for two clock cycles.

5.6 Unmodified Program Packets Echo Control

Once the UDP Packets Store Control unit knows that the incoming UDP packets contain KCPSM programs, the Program Packets Echo Control unit will automatically echo them back to the network. Reception of a valid program packet is guaranteed because the Protocol Wrappers verifies that a new program has been transferred.

5.7 Completed Data Packets Write Control

The Data Packets Write Control unit will only write the completed data packets back to the Protocol Wrappers when there is no new incoming data packet. If a new data packet arrives while it is writing a packet, it will pause the current writing process. The Data Packets Write Control unit will also increment the write queue counter when the KCPSM indicates that it has just finished processing a data packet. When it is ready to write, it will write the number of completed Data packets indicated by the write queue counter.

6 Program Packet Processing

This module has two banks of program memory, and each bank can store a program with a maximum length of 256 instructions. The Program Memory is built using dual-port RAMs. The KCPSM reads the current program on one port while the Interface stores the new program on the other port. Because the Interface stores the new program in the alternate bank of the program memory and because the two data ports of the dual-port RAM operate independently, the KCPSM and the Interface can perform their task simultaneously. As soon as the Interface begins storing a new program in the program memory, it also writes it back out to the Protocol Wrappers so that the unmodified program packet is echoed back to the sender. The overview of this scheme is shown in Figure 7.

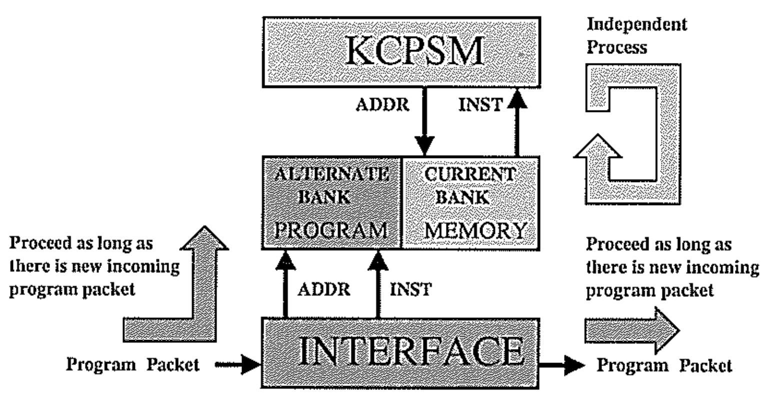

Figure 7: Overview of the Program Packet Processing Configuration

Once the entire content of a new program is stored in the alternate bank of the program memory, and once the current data packet (if any) is completely processed by the KCPSM, the context of the KCPSM is switched to the new program. The KCPSM then processes the next data packet (if any) stored in the next bank of the data memory.

7 Data Packet Processing

This module has four banks of data memory, and each bank can store a data packet with a maximum length of 256 bytes, including the ATM, AAL5, IP and UDP header, the UDP payload, and the ATM trailer. The Data Memory is also built using dual-port RAMs, so the KCPSM can read the current data packet on one port while the Interface stores a new data packet on the other. Because the KCPSM and the Interface work on different banks and because the two data ports of the dual-port RAM operate independently, the KCPSM and the Interface can perform their task simultaneously. Because the Interface only uses one port, it cannot read and write to the Data Memory simultaneously. The Interface will only write the completed data packet back to the Protocol Wrappers when there is no incoming data packet. If a new incoming data packet arrives while the Interface is writing, it will pause the write process, store the new packet to the Data Memory, and then later resume the write process afterwards. The overview of this scheme is shown in Figure 8.

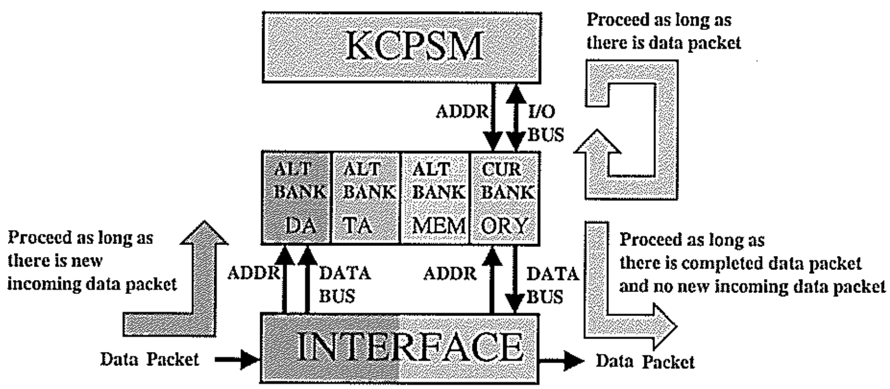

Figure 8: Overview of the Data Packet Processing Configuration

In order to implement this design, the Interface increments a write queue counter when the KCPSM signals that it has finished processing a data packet, and decrements the counter when it has written a completed data packet back to the Protocol Wrappers. The Interface will continue writing completed data packets back to the Protocol Wrappers so long as the write queue counter is not zero and there is no incoming data packet.

8 Programming the KCPSM

The KCPSM executes programs stored in the Program Memory. As described earlier, the Program Memory can store two programs at the same time, and each program is limited to 256 bytes. Programs should be assembled to start at address 0x00. The Interrupt input of the KCPSM is used to reset the processor. Once the Interrupt input is asserted, the program address of the KCPSM is forced to address 0xFF. In order to properly reset the KCPSM, the instruction located at address 0xFF should be a NOP instruction, so on the next clock cycle, the program address can roll over and return back to address 0x00. Therefore, each program is effectively limited to 255 bytes.

8.1 Assembling a KCPSM Program

An assembler called KCPSMBLE is included in the KCPSM package from Xilinx Corp and needs to be executed in a PC platform. KCPSM programs are best written using plain text editors such as Notepad. The program file needs to be saved in a 8.3 file format with a .PSM extension.

The command line used to assemble a KCPSM program is: kcpsmble prog_name.psm.

After KCPSMBLE finishes assembling the program, several files are generated:

- Direct EDIF netlist (<prog_name>.edn). It is a "black box" for a single-port block RAM, and its content has been initialized to the machine code of the program starting at address 0x00. This file can be used for synthesis immediately if the KCPSM is connected to the Program Memory using a single-port block RAM in a ROM configuration.
- Coefficient File (<prog_name>.coe). It is a coefficient file used by COREGEN. This file can be modified so that the KCPSM can connect to the Program Memory using any blok RAM configuration. For this project, the coefficient file generated by KCPSMBLE has been modified so that COREGEN can generate a dual-port block RAM module with its content being initialized to the machine code of the program.
- FORMAT.PSM File. It is the orginal file reformatted.
- Log File (<prog_name>.log). It shows the assembly process in details.
- Xilinx Foundation Simulation File (<prog_name>.hex). It is used to initialize the contents of the block RAM in simulation.

8.2 Registers, Constants and Special Instructions

The KCPSM has 16 8-bit registers and can be used in the program using the "sX" format, where "X" is one of the following: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Constants can also be used in the program and is specified in the form of a two-digit hexadecimal value ranging from 0x00 to 0xFF.

The KCPSM provides several special instructions used to handling Interrupt and I/O operations:

- ENABLE INTERRUPT. This instruction enables future interrupt requests. If the program wants to support interrupt requests, this instruction should be the first instruction of the program because the KCPSM does not enable interrupt request by default.
- DISABLE INTERRUPT. This instruction disables future interrupt requests.
- RETURNI ENABLE. This instruction is a special variation of the RETURN instruction. It concludes an interrupt service routine and specifies that future interrupt requests are enabled.
- RETURNI DISABLE. This instruction is the same as the RETURNI ENABLE instruction except it specifies that future interrupt requests are disabled.
- INPUT. This instruction enables 8-bit data values external to the KCPSM to be transferred to any one of the internal registers. The port address can be specified directly as a constant value or indirectly as the contents of any other registers.
- OUTPUT. This instruction enables 8-bit data values in any one of the internal registers to be transferred to logic external to the KCPSM. The port address can be specified directly as a constant value or indirectly as the contents of any other registers.

For detailed explanation of every instruction available to the KCPSM, please refer to the KCPSM Application Notes [3].

8.3 A Simple Program

An sample KCPSM program that implements a down counter and outputs the values of the counter on port address 0x01 is shown below.

```
;This program implements a DOWN counter
;By Henry Fu
;(c) 2001 Washington University, Applied Research Lab

        ENABLE  INTERRUPT
INIT:   LOAD    SA,80              ;INITIAL VALUE IS 15
LOOP:   OUTPUT  SA,01              ;SEND IT OUT ON PORT 01
        SUB     SA,01              ;DECREMENT BY 1
CONT:   JUMP    NZ,LOOP            ;KEEP COUNTING
        JUMP    INIT               ;KEEP DOING IT FOREVER
```

The assembler output of the example program is shown below.

```
KCPSM Assembler v1.11     Ken Chapman (Xilinx-UK) 2000

PASS 1 - Reading PSM file with basic format checking

;This program implements a DOWN counter
;By Henry Fu
;(c) 2001 Washington University, Applied Research Lab
        ENABLE  INTERRUPT
INIT:   LOAD    SA,80              ;INITIAL VALUE IS 15
LOOP:   OUTPUT  SA,01              ;SEND IT OUT ON PORT 01
        SUB     SA,01              ;DECREMENT BY 1
CONT:   JUMP    NZ,LOOP            ;KEEP COUNTING
        JUMP    INIT               ;KEEP DOING IT FOREVER

PASS 2 - Computing program addresses and building Label cross reference

00    ;This program implements a DOWN counter
00    ;By Henry Fu
00    ;(c) 2001 Washington University, Applied Research Lab
00  ENABLE INTERRUPT
01  INIT: LOAD sA,80 ;INITIAL VALUE IS 15
02  LOOP: OUTPUT sA,01 ;SEND IT OUT ON PORT 01
03  SUB sA,01 ;DECREMENT BY 1
04  CONT: JUMP NZ,LOOP ;KEEP COUNTING
05  JUMP INIT ;KEEP DOING IT FOREVER

PASS 3 - Resolving operands, constants and labels

  ;This program implements a DOWN counter
  ;By Henry Fu
  ;(c) 2001 Washington University, Applied Research Lab
ENABLE INTERRUPT
INIT: LOAD sA,80 ;INITIAL VALUE IS 15
LOOP: OUTPUT sA,01 ;SEND IT OUT ON PORT 01
SUB sA,01 ;DECREMENT BY 1
CONT: JUMP NZ,LOOP ;KEEP COUNTING
```

```
JUMP INIT ;KEEP DOING IT FOREVER

PASS 4 - Writing formatted version of program to file 'format.psm'


PASS 5 - Assemble - Log file 'test.log'.


PASS 6 - Writing ROM template file 'test.coe'.
         Writing HEX file for simulation utilities 'test.hex'.


PASS 7 - Writing EDIF program ROM design file 'test.edn'.


KCPSMBLE complete.
```

8.4 Debugging a KCPSM Program

A simple debugger called PSMDEBUG is included in the KCPSM package from Xilinx Corp and needs to be executed in a PC platform. It allows the user to test the operation of the program and provides internal program details such as program flow, I/O operations, and values of internal registers and flags. Notice that the ".coe" file used must be the one generated by KCPSMBLE and be unmodified.

The command line used to debug a KCPSM program is: `psmdebug prog_name.coe`.

Detail instructions on how to debug the program will appear on the screen after PSMDEBUG is executed.

8.5 Converting a KCPSM Program for Simulation, Synthesis, and Testing

Because the ".coe" file generated by KCPSMBLE is targeted to use with Single-Port block RAMs generated by COREGEN, some modifications to that file must be made before it can be used with the Dual-Port block RAMS. Also, the raw content of the KCPSM program stored in the ".coe" file must be converted before it can be used for simulation or laboratory testing. A C program called CONVERT is written to accomplish these tasks.

The command line used is: `convert < prog_name.coe`.

Two files are generated after the CONVERT program is executed:

- `program.coe`. It is a coefficient file used by COREGEN to generate a dual-port block RAM. The content of the generated dual-port block RAM will be preloaded with the KCPSM program.
- `INST.TBP`. It is a file used by the IP2FAKE and UDPTEST programs. IP2FAKE takes in the INST.TBP file to produced a fake ATM cell for simulation, and UDPTEST takes in the INST.TBP to send a UDP datagram over the network.

8.6 A KCPSM Program Template

All KCPSM programs should follow a template in order for them to work with the Interface module. A program should first enables the interrupt of the KCPSM so that the Interface unit is able to reset the KCPSM. Next, it should check if a data packet is stored in the memory. If there is no data packet stored in the memory, it should signal to the Interface that it has done processing and there is no write by writing 0x00 to address 0xFF. If there is a data packet stored in the memory, it can start processing. After the processing is done, it should signals to the Interface that it has done processing by writing 0xFF to address 0xFF. Finally, it should suspends the KCPSM. A KCPSM program template is shown below.

```
;Enable KCPSM Interrupt so that the KCPSM Interface can reset the KCPSM
        ENABLE  INTERRUPT

;Check if there is new data packet in memory
;if no data, signal KCPSM Interface that process is done with no write
;if there is data, proceed to process data
INIT:   INPUT   SA,00           ;LOAD M[00] TO SA
        SUB     SA,00           ;COMPARE SA TO 00
        JUMP    NZ,CHK          ;DATA IN MEMORY, PROCEED
        LOAD    SA,00           ;NO DATA, LOAD 00 TO SA, no write, signal done
        OUTPUT  SA,FF           ;WRITE 00 TO M[FF], no write, signal done
        JUMP    WAIT            ;WAIT

;Assembly code to process data
CHK:    ;Assembly code goes here

;Signal KCPSM Interface that process is done, and needs to write it out
DONE:   LOAD    SA,FF           ;LOAD FF TO SA, signal done
        OUTPUT  SA,FF           ;WRITE FF TO M[FF], write, signal done

;Suspend KCPSM
WAIT:   JUMP    WAIT
```

8.7 A Complete Program: String Replacement

The following example exercises the KCPSM and demonstrates the software reconfigurability of this FPX module. The first program does not perform any modification to the UDP packet. The second program looks for the string 'Hello' in the payload of an UDP packet, and once it finds a match, it replaces the string 'Hello' with 'Henry'. The third program looks for the string 'Hello' in the payload of an UDP packet, and once it finds a match it appends the string ' Henry' after 'Hello'.

The source code for the first program is shown below.

```
;Check if there is a new data packet in memory,
;If yes, signal done
;By Henry Fu
;(c) Washington University Applied Research Lab
```

```
        ENABLE  INTERRUPT

;Check if there is new data packet in memory
INIT:   INPUT   SA,00           ;LOAD M[00] TO SA
        SUB     SA,00           ;COMPARE SA TO 00
        JUMP    NZ,DONE         ;DATA IN MEMORY, PROCEED
        LOAD    SA,00           ;NO DATA, LOAD 00 TO SA, no write, signal done
        OUTPUT  SA,FF           ;WRITE 00 TO M[FF], no write, signal done
        JUMP    WAIT            ;WAIT

;Signal processing done
DONE:   LOAD    SA,FF           ;LOAD FF TO SA, signal done
        OUTPUT  SA,FF           ;WRITE FF TO M[FF], write, signal done

;Suspend KCPSM
WAIT:   JUMP    WAIT
```

The source code for the second program is shown below.

```
;Check if the data at Address 24 to 28 = 'Hello',
;If yes, change it to 'Henry'
;By Henry Fu
;(c) Washington University Applied Research Lab

        ENABLE  INTERRUPT

;Check if there is new data packet in memory
INIT:   INPUT   SA,00           ;LOAD M[00] TO SA
        SUB     SA,00           ;COMPARE SA TO 00
        JUMP    NZ,CHK          ;DATA IN MEMORY, PROCEED
        LOAD    SA,00           ;NO DATA, LOAD 00 TO SA, no write, signal done
        OUTPUT  SA,FF           ;WRITE 00 TO M[FF], no write, signal done
        JUMP    WAIT            ;WAIT

;Check the string 'Hello' in payload
CHK:    INPUT   SA,24           ;LOAD M[24] TO SA
        SUB     SA,48           ;COMPARE SA TO 'H'
        JUMP    NZ,DONE         ;NOT EQUAL => NO CHANGE
        INPUT   SA,25           ;LOAD M[25] TO SA
        SUB     SA,65           ;COMPARE SA TO 'e'
        JUMP    NZ,DONE         ;NOT EQUAL => NO CHANGE
        INPUT   SA,26           ;LOAD M[26] TO SA
        SUB     SA,6C           ;COMPARE SA TO 'l'
        JUMP    NZ,DONE         ;NOT EQUAL => NO CHANGE
        INPUT   SA,27           ;LOAD M[27] TO SA
        SUB     SA,6C           ;COMPARE SA TO 'l'
        JUMP    NZ,DONE         ;NOT EQUAL => NO CHANGE
        INPUT   SA,28           ;LOAD M[28] TO SA
        SUB     SA,6F           ;COMPARE SA TO 'o'
        JUMP    NZ,DONE         ;NOT EQUAL => NO CHANGE

;Replace the string 'Hello' with 'Henry'
```

```
        LOAD     SA,48            ;LOAD 'H' TO SA
        OUTPUT   SA,24            ;WRITE SA TO M[24]
        LOAD     SA,65            ;LOAD 'e' TO SA
        OUTPUT   SA,25            ;WRITE SA TO M[25]
        LOAD     SA,6E            ;LOAD 'n' TO SA
        OUTPUT   SA,26            ;WRITE SA TO M[26]
        LOAD     SA,72            ;LOAD 'r' TO SA
        OUTPUT   SA,27            ;WRITE SA TO M[27]
        LOAD     SA,79            ;LOAD 'y' TO SA
        OUTPUT   SA,28            ;WRITE SA TO M[28]

;Signal processing done
DONE:   LOAD     SA,FF            ;LOAD FF TO SA, signal done
        OUTPUT   SA,FF            ;WRITE FF TO M[FF], write, signal done

;Suspend KCPSM
WAIT:   JUMP     WAIT
```

The source code for the third program is shown below.

```
;Check if the data at Address 24 to 28 = 'Hello',
;If yes, change the data at Address 29 to 2E = ' Henry'
;By Henry Fu
;(c) 2001 Washington University Applied Research Lab

        ENABLE   INTERRUPT

;Check if there is new data packet in memory
INIT:   INPUT    SA,00            ;LOAD M[00] TO SA
        SUB      SA,00            ;COMPARE SA TO 00
        JUMP     NZ,CHK           ;DATA IN MEMORY, PROCEED
        LOAD     SA,00            ;NO DATA, LOAD 00 TO SA, no write, signal done
        OUTPUT   SA,FF            ;WRITE 00 TO M[FF], no write, signal done
        JUMP     WAIT             ;WAIT

;Check the string 'Hello' in the payload
CHK:    INPUT    SA,24            ;LOAD M[24] TO SA
        SUB      SA,48            ;COMPARE SA TO 'H'
        JUMP     NZ,DONE          ;NOT EQUAL => NO CHANGE
        INPUT    SA,25            ;LOAD M[25] TO SA
        SUB      SA,65            ;COMPARE SA TO 'e'
        JUMP     NZ,DONE          ;NOT EQUAL => NO CHANGE
        INPUT    SA,26            ;LOAD M[26] TO SA
        SUB      SA,6C            ;COMPARE SA TO 'l'
        JUMP     NZ,DONE          ;NOT EQUAL => NO CHANGE
        INPUT    SA,27            ;LOAD M[27] TO SA
        SUB      SA,6C            ;COMPARE SA TO 'l'
        JUMP     NZ,DONE          ;NOT EQUAL => NO CHANGE
        INPUT    SA,28            ;LOAD M[28] TO SA
        SUB      SA,6F            ;COMPARE SA TO 'o'
        JUMP     NZ,DONE          ;NOT EQUAL => NO CHANGE
```

```
;Move ATM trailer one word down in memory to create
;space for the appended string ' Henry'
TRAIL:  INPUT   SA,33           ;LOAD M[33] TO SA
        OUTPUT  SA,37           ;WRITE SA TO M[37]
        INPUT   SA,32           ;LOAD M[32] TO SA
        OUTPUT  SA,36           ;WRITE SA TO M[36]
        INPUT   SA,31           ;LOAD M[31] TO SA
        OUTPUT  SA,35           ;WRITE SA TO M[35]
        INPUT   SA,30           ;LOAD M[30] TO SA
        OUTPUT  SA,34           ;WRITE SA TO M[34]
        INPUT   SA,2F           ;LOAD M[2F] TO SA
        OUTPUT  SA,33           ;WRITE SA TO M[33]
        INPUT   SA,2E           ;LOAD M[2E] TO SA
        OUTPUT  SA,32           ;WRITE SA TO M[32]
        INPUT   SA,2D           ;LOAD M[2D] TO SA
        OUTPUT  SA,31           ;WRITE SA TO M[31]
        INPUT   SA,2C           ;LOAD M[2C] TO SA
        OUTPUT  SA,30           ;WRITE SA TO M[30]

;Append the string ' Henry' after the string 'Hello'
        LOAD    SA,20           ;LOAD ' ' TO SA
        OUTPUT  SA,29           ;WRITE SA TO M[29]
        LOAD    SA,48           ;LOAD 'H' TO SA
        OUTPUT  SA,2A           ;WRITE SA TO M[2A]
        LOAD    SA,65           ;LOAD 'e' TO SA
        OUTPUT  SA,2B           ;WRITE SA TO M[2B]
        LOAD    SA,6E           ;LOAD 'n' TO SA
        OUTPUT  SA,2C           ;WRITE SA TO M[2C]
        LOAD    SA,72           ;LOAD 'r' TO SA
        OUTPUT  SA,2D           ;WRITE SA TO M[2D]
        LOAD    SA,79           ;LOAD 'y' TO SA
        OUTPUT  SA,2E           ;WRITE SA TO M[2E]
        LOAD    SA,00           ;LOAD NULL TO SA
        OUTPUT  SA,2F           ;WRITE NULL TO M[2F]

;Update the length of the UDP packet
        LOAD    SA,18           ;LOAD 0x18 TO SA
        OUTPUT  SA,1E           ;WRITE SA TO M[1E]
        LOAD    SA,00           ;LOAD 0x00 TO SA
        OUTPUT  SA,1F           ;WRITE SA TO M[1F]

;Signal processing done
DONE:   LOAD    SA,FF           ;LOAD FF TO SA, signal done
        OUTPUT  SA,FF           ;WRITE FF TO M[FF], write, signal done

;Suspend KCPSM
WAIT:   JUMP    WAIT
```

9 Implementation Results

The circuit to implement the FPX KCPSM module was simulated using Modelsim. Once the functionality of the circuit is verified, it was then synthesized to a Xilinx XCV1000E-FG680-7 using Synplicity Pro and Xilinx backend synthesis tools.

9.1 VHDL Implementation

The VHDL source code of the Interface module is shown below.

```
-- $Id: interface.vhd,v 1.2 2001/07/26 18:42:54 hwf1 Exp $
--
-- KCPSM Interface
-- This interface parses a UDP datagram. If it is a KCPSM instruction code,
-- stores it into the dual port program ram. If it is a KCPSM data, stores
-- it into the dual port data ram. After KCPSM has processed the datagram,
-- it echoes back it out.
--
-- Author: Henry Fu
-- (c) 2001 Washington University, Applied Research Lab
--
-- this is the actual Interface entity
--

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.all;
use IEEE.std_logic_unsigned.all;

entity Interface is

  port (
    CLK            : in  std_logic;                        -- clock
    Reset_l        : in  std_logic;                        -- reset

    D_MOD_IN       : in  std_logic_vector (31 downto 0);  -- data
    DataEn_MOD_IN  : in  std_logic;                        -- data enable
    SOF_MOD_IN     : in  std_logic;                        -- start of frame
    EOF_MOD_IN     : in  std_logic;                        -- end of frame
    SOD_MOD_IN     : in  std_logic;                        -- start of datagram
    TCA_MOD_IN     : out std_logic;                        -- congestion control

    D_OUT_MOD      : out std_logic_vector (31 downto 0);  -- data
    DataEn_OUT_MOD : out std_logic;                        -- data enable
    SOF_OUT_MOD    : out std_logic;                        -- start of frame
    EOF_OUT_MOD    : out std_logic;                        -- end of frame
    SOD_OUT_MOD    : out std_logic;                        -- start of datagram
    TCA_OUT_MOD    : in  std_logic;                        -- congestion control

    KCPSM_INTERRUPT: out std_logic;                        -- KCPSM Interrupt
    KCPSM_READ_STROBE : in std_logic;
    KCPSM_WRITE_STROBE : in std_logic;
```

```
    KCPSM_PROG_BANK: out std_logic;
    KCPSM_DATA_BANK: out std_logic_vector (1 downto 0);
    KCPSM_PORT_ID  : in  std_logic_vector (7 downto 0);
    KCPSM_OUTPUT   : in  std_logic_vector (7 downto 0);

    PROGRAM_ADDRB  : out std_logic_vector (7 downto 0);   -- pram address line
    PROGRAM_DIB    : out std_logic_vector (31 downto 0);  -- pram data in line
    PROGRAM_WEB    : out std_logic;                       -- pram write enable
    PROGRAM_ENB    : out std_logic;                       -- pram chip enable
    PROGRAM_DOB    : in  std_logic_vector (31 downto 0);  -- pram data out line
    DATARAM_ADDRB  : out std_logic_vector (7 downto 0);   -- dram address line
    DATARAM_DIB    : out std_logic_vector (31 downto 0);  -- dram data in line
    DATARAM_WEB    : out std_logic;                       -- dram write enable
    DATARAM_ENB    : out std_logic;                       -- dram chip enable
    DATARAM_DOB    : in  std_logic_Vector (31 downto 0)); -- dram data out line

end Interface;


architecture behavior of Interface is

  ------------------------------------------------------------------------------
  -- HEADER FIFO
  ------------------------------------------------------------------------------
  component fifo_63x36
    port (
      din: IN std_logic_VECTOR(35 downto 0);
      wr_en: IN std_logic;
      wr_clk: IN std_logic;
      rd_en: IN std_logic;
      rd_clk: IN std_logic;
      ainit: IN std_logic;
      dout: OUT std_logic_VECTOR(35 downto 0);
      full: OUT std_logic;
      empty: OUT std_logic);
  end component;

  ------------------------------------------------------------------------------
  -- UDP Echo
  ------------------------------------------------------------------------------
  component UDPEcho
    port (
      CLK            : in  std_logic;      -- clock
      Reset_l        : in  std_logic;      -- synchronous reset, active low
      D_MOD_IN       : in  std_logic_vector (31 downto 0);      -- cell data
      DataEn_MOD_IN  : in  std_logic;      -- data enable
      SOF_MOD_IN     : in  std_logic;      -- start of frame
      EOF_MOD_IN     : in  std_logic;      -- end of frame
      SOD_MOD_IN     : in  std_logic;      -- start of datagram
      TCA_MOD_IN     : out std_logic;      -- congestion control
      D_OUT_MOD      : out std_logic_vector (31 downto 0);      -- cell data
      DataEn_OUT_MOD : out std_logic;      -- data enable
      SOF_OUT_MOD    : out std_logic;      -- start of frame
      EOF_OUT_MOD    : out std_logic;      -- end of frame
```

```
    SOD_OUT_MOD    : out std_logic;      -- start of datagram
    TCA_OUT_MOD    : in  std_logic);     -- congestion control
end component;

------------------------------------------------------------------------------
-- input buffer
------------------------------------------------------------------------------
signal data_in : std_logic_vector (31 downto 0);  -- data input
signal dataen_in : std_logic;           -- data enable
signal sof_in : std_logic;              -- start of frame
signal eof_in : std_logic;              -- end of frame
signal sod_in : std_logic;              -- start of datagram
signal tca_out : std_logic;

------------------------------------------------------------------------------
-- udpecho out buffer
------------------------------------------------------------------------------
signal data_e : std_logic_vector (31 downto 0);  -- data input
signal dataen_e : std_logic;            -- data enable
signal sof_e : std_logic;               -- start of frame
signal eof_e : std_logic;               -- end of frame
signal sod_e : std_logic;               -- start of datagram
signal tca_e : std_logic;

------------------------------------------------------------------------------
-- fifo out buffer
------------------------------------------------------------------------------
signal data_f : std_logic_vector (31 downto 0);
signal dataen_f : std_logic;
signal sof_f : std_logic;
signal eof_f : std_logic;
signal sod_f : std_logic;

------------------------------------------------------------------------------
-- final buffer
------------------------------------------------------------------------------
signal dataen_buf : std_logic;
signal sof_buf : std_logic;
signal eof_buf : std_logic;
signal sod_buf : std_logic;

------------------------------------------------------------------------------
-- intermediate buffer
------------------------------------------------------------------------------
signal dataen_int : std_logic;
signal sof_int : std_logic;
signal eof_int : std_logic;
signal sod_int : std_logic;

------------------------------------------------------------------------------
-- output buffer
------------------------------------------------------------------------------
signal data_out : std_logic_vector (31 downto 0);
signal dataen_out : std_logic;
```

```
signal sof_out : std_logic;
signal eof_out : std_logic;
signal sod_out : std_logic;

-----------------------------------------------------------------------------
-- fifo signal
-----------------------------------------------------------------------------
signal fifo_din2 : std_logic_vector (35 downto 0);
signal fifo_dout2 : std_logic_vector (35 downto 0);
signal ainit : std_Logic;
signal wr_en2 : std_logic;
signal rd_en2 : std_Logic;
signal full : std_logic;
signal empty : std_logic;

-----------------------------------------------------------------------------
-- kcpsm signal
-----------------------------------------------------------------------------
signal kcpsm_read_strobe_in : std_logic;
signal kcpsm_write_strobe_in : std_logic;
signal kcpsm_port_id_in : std_logic_vector (7 downto 0);
signal kcpsm_output_in : std_logic_vector (7 downto 0);

-----------------------------------------------------------------------------
-- control signal
-----------------------------------------------------------------------------
signal addr_count : unsigned (6 downto 0);   -- pram address counter
signal bank_count : unsigned (0 downto 0);   -- pram address bank counter
signal addrd_count : unsigned (5 downto 0);  -- dram address counter (rd)
signal bankd_count : unsigned (1 downto 0);  -- dram address bank counter (rd)
signal prog_bank : unsigned (0 downto 0);    -- KCPSM pram address bank
signal data_bank : unsigned (1 downto 0);    -- KCPSM dram address bank
signal addrd2_count : unsigned (5 downto 0); -- dram address counter (wr)
signal bankd2_count : unsigned (1 downto 0); -- dram address bank counter (wr)
signal payload_count : unsigned (5 downto 0);
signal write_queue_count : unsigned (1 downto 0);
signal program_queue_count : unsigned (0 downto 0);
signal enb : std_logic;
signal web : std_logic;
signal k_int2 : std_logic;
signal enbd : std_logic;
signal enbd2 : std_logic;
signal webd : std_logic;
signal is_start : std_logic;
signal is_data : std_logic;
signal is_program : std_logic;
signal wr_grant : std_logic;
signal wr_done : std_logic;
signal fifo2udp : std_logic;
signal ram2udp : std_logic;
signal is_loaded : std_logic;
signal is_reload : std_logic;

-----------------------------------------------------------------------------
```

```
-- udp to fifo state machine
------------------------------------------------------------------------------
type UDP2FIFO_StateType is (Idle, UDP2FIFO, Done, Done2); -- states
signal udp2fifo_state    : UDP2FIFO_StateType;  -- current udp2dpram state
signal nx_udp2fifo_state : UDP2FIFO_StateType;  -- next udp2dpram state

------------------------------------------------------------------------------
-- check data / program state machine
------------------------------------------------------------------------------
type CHK_D_I_StateType is (Idle, Req, Req2, CHK_D_I, Program, Data);
signal chk_d_i_state    : CHK_D_I_StateType;  -- current udp2dpram state
signal nx_chk_d_i_state : CHK_D_I_StateType;  -- next udp2dpram state

------------------------------------------------------------------------------
-- udp to dpram state machine
------------------------------------------------------------------------------
type FIFO2RAM_StateType is (Idle, Start,
                            fifo2program_req, fifo2program_udp,
                            fifo2program_chk, fifo2program_copy,
                            fifo2program_go, fifo2program_go2,
                            fifo2dataram_copy,
                            fifo2dataram_go, fifo2dataram_go2);
signal fifo2ram_state    : FIFO2RAM_StateType;  -- current udp2dpram state
signal nx_fifo2ram_state : FIFO2RAM_StateType;  -- next udp2dpram state

------------------------------------------------------------------------------
-- fifo to udp state machine
------------------------------------------------------------------------------
type FIFO2UDP_StateType is (Idle, Start, fifo2udp_copy,
                            fifo2udp_go, fifo2udp_go2);
signal fifo2udp_state    : FIFO2UDP_StateType;  -- current udp2dpram state2
signal nx_fifo2udp_state : FIFO2UDP_StateType;  -- next udp2dpram state

------------------------------------------------------------------------------
-- ram to udp state machine
------------------------------------------------------------------------------
type RAM2UDP_StateType is (Idle, ram2udp_copy, ram2udp_copy2,
                           ram2udp_sof, ram2udp_ip_header,
                           ram2udp_ip_header2, ram2udp_ip_header3,
                           ram2udp_ip_header4, ram2udp_ip_header5,
                           ram2udp_udp_header, ram2udp_udp_header2,
                           ram2udp_payload, ram2udp_trail_copy,
                           ram2udp_trail_copy2, ram2udp_trailer,
                           ram2udp_trailer2, ram2udp_done);
signal ram2udp_state    : RAM2UDP_StateType;  -- current udp2dpram state2
signal nx_ram2udp_state : RAM2UDP_StateType;  -- next udp2dpram state
signal pv_ram2udp_state : RAM2UDP_StateType;  -- previous udp2dpram state

------------------------------------------------------------------------------
-- reset KCPSM state machine after data write
------------------------------------------------------------------------------
type RESET_KCPSM_StateType is (Idle, Check, K_Go1, K_Go2,
                               NW_Check, NW_Go1, NW_Go2);
signal reset_kcpsm_state    : RESET_KCPSM_StateType;
```

```
  signal nx_reset_kcpsm_state : RESET_KCPSM_StateType;

begin  -- behavior

  myfifo : fifo_63x36 port map (
    din         => fifo_din2,
    wr_en       => wr_en2,
    wr_clk      => clk,
    rd_en       => rd_en2,
    rd_clk      => clk,
    ainit       => ainit,
    dout        => fifo_dout2,
    full        => full,
    empty       => empty);

  myecho : udpecho port map (
    CLK             => clk,
    Reset_l         => Reset_l,
    D_MOD_IN        => D_MOD_IN,
    DataEn_MOD_IN   => DataEn_MOD_IN,
    SOF_MOD_IN      => SOF_MOD_IN,
    EOF_MOD_IN      => EOF_MOD_IN,
    SOD_MOD_IN      => SOD_MOD_IN,
    TCA_MOD_IN      => TCA_MOD_IN,
    D_OUT_MOD       => data_e,
    DataEn_OUT_MOD  => dataen_e,
    SOF_OUT_MOD     => sof_e,
    EOF_OUT_MOD     => eof_e,
    SOD_OUT_MOD     => sod_e,
    TCA_OUT_MOD     => tca_e);

  -----------------------------------------------------------------------------
  -- purpose: copy data on clock edge
  -- type   : sequential
  -- inputs : CLK, Reset_l
  -- outputs:
  -----------------------------------------------------------------------------
  clock_copy: process (CLK, Reset_l)
  begin  -- process clock_copy
    if CLK'event and CLK = '1' then  -- rising clock edge
      -- reset
      if Reset_l = '0' then
        sof_in  <= '0';
      else
        sof_in  <= sof_e;
      end if;

      -- input buffer stage
      data_in   <= data_e;
      dataen_in <= dataen_e;
      eof_in    <= eof_e;
      sod_in    <= sod_e;
      tca_out   <= TCA_OUT_MOD;
```

```
      -- fifo buffer stage
      data_f    <= FIFO_DOUT2 (31 downto 0);
      dataen_f <= FIFO_DOUT2 (32);
      sof_f     <= FIFO_DOUT2 (33);
      eof_f     <= FIFO_DOUT2 (34);
      sod_f     <= FIFO_DOUT2 (35);

      -- kcpsm buffer stage
      kcpsm_read_strobe_in <= KCPSM_READ_STROBE;
      kcpsm_write_strobe_in <= KCPSM_WRITE_STROBE;
      kcpsm_port_id_in <= KCPSM_PORT_ID;
      kcpsm_output_in <= KCPSM_OUTPUT;

      -- intermediate buffer stage
      dataen_int <= dataen_buf;
      sof_int <= sof_buf;
      eof_int <= eof_buf;
      sod_int <= sod_buf;

      -- final buffer stage
      data_out <= DATARAM_DOB;
      dataen_out <= dataen_int;
      sof_out <= sof_int;
      eof_out <= eof_int;
      sod_out <= sod_int;

      -- state machines
      udp2fifo_state <= nx_udp2fifo_state;
      chk_d_i_state <= nx_chk_d_i_state;
      fifo2ram_state <= nx_fifo2ram_state;
      fifo2udp_state <= nx_fifo2udp_state;
      ram2udp_state <= nx_ram2udp_state;
      pv_ram2udp_state <= ram2udp_state;
      reset_kcpsm_state <= nx_reset_kcpsm_state;

  end if;
end process clock_copy;

-------------------------------------------------------------------------------
-- purpose: statemachine to copy incoming data to fifo
-- type    : combinational
-- inputs : udp2fifo_state, sof_in, eof_in, Resetl
-- outputs: nx_udp2fifo_state, wr_en2
-------------------------------------------------------------------------------
udp2fifo_state_machine : process (Reset_l, udp2fifo_state, sof_in, eof_in,
                                  dataen_in, data_in)

variable tmp_state : udp2fifo_statetype;
variable tmp_wr_en2 : std_logic;

begin -- process udp2fifo_state_machine

  -- default values
  tmp_state := udp2fifo_state;
```

```
  tmp_wr_en2 := '0';

  if Reset_l = '0' then
    tmp_state := Idle;
  else
    case udp2fifo_state is
      when Idle       => if sof_in = '1' then
                           tmp_state := udp2fifo;
                           tmp_wr_en2 := '1';
                         end if;
      when udp2fifo   => if eof_in = '1' then
                           tmp_state := Done;
                         end if;
                         tmp_wr_en2 := '1';
      when Done       => if dataen_in = '1' and data_in(15) = '1' then
                           tmp_state := Done2;
                         end if;
                         tmp_wr_en2 := '1';
      when Done2      => if sof_in = '1' then
                           tmp_state := udp2fifo;
                         end if;
                         tmp_wr_en2 := '1';
    end case;
  end if;

  -- assign signals
  nx_udp2fifo_state <= tmp_state;
  wr_en2 <= tmp_wr_en2;

end process udp2fifo_state_machine;

-------------------------------------------------------------------------------
-- purpose: statemachine to check if the incoming udp packet is data or inst
-- type   : combinational
-- inputs : Reset_l, chk_d_i_state, sof_in, sod_in, data_in, eof_in
-- outputs: nx_chk_d_i_state, is_data, is_program
-------------------------------------------------------------------------------
chk_d_i_state_machine: process (Reset_l, chk_d_i_state, dataen_in,
                                sof_in, sod_in, data_in, eof_in)

variable tmp_state : CHK_D_I_StateType;
variable tmp_is_start, tmp_is_data, tmp_is_program : std_logic;

begin -- process chk_d_i_state_machine

  -- default values
  tmp_state := chk_d_i_state;
  tmp_is_data := '0';
  tmp_is_program := '0';
  tmp_is_start := '0';

  if Reset_l = '0' then
    tmp_state := Idle;
  else
```

```
    case chk_d_i_state is
      when Idle       => if sof_in = '1' then
                           tmp_state := Req;
                         end if;
      when Req        => if sod_in = '1' then
                           tmp_state := Req2;
                         end if;
      when Req2       => if dataen_in = '1' then
                           tmp_state := CHK_D_I;
                         end if;
                         tmp_is_start := '1';
      when CHK_D_I    => if dataen_in = '1' and data_in = X"00000000" then
                           tmp_state := Program;
                         elsif dataen_in = '1' and data_in = X"00000001" then
                           tmp_state := Data;
                         end if;
      when Program    => if eof_in = '1' then
                           tmp_state := Idle;
                         end if;
                         tmp_is_program := '1';
      when Data       => if eof_in = '1' then
                           tmp_state := Idle;
                         end if;
                         tmp_is_data := '1';
    end case;
  end if;

  -- assign signals
  nx_chk_d_i_state <= tmp_state;
  is_start <= tmp_is_start;
  is_program <= tmp_is_program;
  is_data <= tmp_is_data;

end process chk_d_i_state_machine;

-------------------------------------------------------------------------------
-- purpose: statemachine to copy incoming udp packet into ram
-- type   : combinational
-- inputs : Reset_l, fifo2ram_state, is_start, is_program, is_data,
--          dataen_f, sof_f, sod_f, eof_f
-- outputs: nx_fifo2ram_state, rd_en2, enb, web, k_int,
--          enbd, webd, wr_grant
-------------------------------------------------------------------------------
fifo2ram_state_machine: process (Reset_l, fifo2ram_state, is_start,
                                 is_program, is_data, dataen_f,
                                 sof_f, sod_f, eof_f, data_f)

variable tmp_state : FIFO2RAM_StateType;
variable tmp_rd_en2, tmp_wr_grant : std_logic;
variable tmp_enb, tmp_web, tmp_enbd, tmp_webd, tmp_is_loaded : std_logic;

begin -- process fifo2ram_state_machine

  -- default values
```

```
tmp_state := fifo2ram_state;
tmp_rd_en2 := '0';
tmp_enb := '0';
tmp_web := '1';
tmp_enbd := '0';
tmp_webd := '1';
tmp_wr_grant := '0';
tmp_is_loaded := '0';

if Reset_l = '0' then
  tmp_state := Idle;
else
  case fifo2ram_state is
    when Idle               => if is_start = '1' then
                                 tmp_state := Start;
                                 tmp_rd_en2 := '1';
                               end if;
    when Start              => if is_program = '1' and sof_f = '1' then
                                 tmp_state := fifo2program_req;
                               elsif is_data = '1' and sof_f = '1' then
                                 tmp_state := fifo2dataram_copy;
                                 tmp_enbd := '1';
                                 tmp_webd := '0';
                               else
                                 tmp_wr_grant := '1';
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_req   => if sod_f = '1' then
                                 tmp_state := fifo2program_udp;
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_udp   => if dataen_f = '1' then
                                 tmp_state := fifo2program_chk;
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_chk   => if dataen_f = '1' then
                                 tmp_state := fifo2program_copy;
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_copy  => if eof_f = '1' then
                                 tmp_state := fifo2program_go;
                               end if;
                               if dataen_f = '1' then
                                 tmp_enb := '1';
                                 tmp_web := '0';
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_go    => if dataen_f = '1' and data_f(15) = '1' then
                                 tmp_state := fifo2program_go2;
                               end if;
                               tmp_rd_en2 := '1';
    when fifo2program_go2   => if dataen_f = '1' then
                                 tmp_state := Start;
                                 tmp_is_loaded := '1';
```

```
                                end if;
                                tmp_rd_en2 := '1';
      when fifo2dataram_copy  => if eof_f = '1' then
                                  tmp_state := fifo2dataram_go;
                                end if;
                                if dataen_f = '1' then
                                  tmp_enbd := '1';
                                  tmp_webd := '0';
                                end if;
                                tmp_rd_en2 := '1';
      when fifo2dataram_go    => if dataen_f = '1' and data_f(15) = '1' then
                                  tmp_state := fifo2dataram_go2;
                                end if;
                                if dataen_f = '1' then
                                  tmp_enbd := '1';
                                  tmp_webd := '0';
                                end if;
                                tmp_rd_en2 := '1';
      when fifo2dataram_go2   => if dataen_f = '1' then
                                  tmp_state := Start;
                                  tmp_enbd := '1';
                                  tmp_webd := '0';
                                end if;
                                tmp_rd_en2 := '1';
    end case;
  end if;

  -- assign signals
  nx_fifo2ram_state <= tmp_state;
  rd_en2 <= tmp_rd_en2;
  enb <= tmp_enb;
  web <= tmp_web;
  enbd <= tmp_enbd;
  webd <= tmp_webd;
  wr_grant <= tmp_wr_grant;
  is_loaded <= tmp_is_loaded;

end process fifo2ram_state_machine;

-----------------------------------------------------------------------------
-- purpose: control address counter
-- type   : sequential
-- inputs : Clk, Reset_l
-- outputs: addr_count, bank_count, addrd_count, bankd_count,
-----------------------------------------------------------------------------
addr_count_control : process (Clk, Reset_l)

begin -- process addr_count_control

  if CLK'event and CLK = '1' then
    if Reset_l = '0' then
      bank_count <= "1";       -- don't overwrite program initial content
      bankd_count <= "00";     -- don't overwrite data initial content
      addr_count <= "0000000";
```

```
      addrd_count <= "000000";
    else
      case fifo2ram_state is
        when Start             => if is_data = '1' and sof_f = '1' then
                                    addrd_count <= addrd_count + 1;
                                  end if;
        when fifo2program_copy=> if dataen_f = '1' then
                                    addr_count <= addr_count + 1;
                                  end if;
        when fifo2dataram_copy=> if dataen_f = '1' then
                                    addrd_count <= addrd_count + 1;
                                  end if;
        when fifo2program_go2 => addr_count <= "0000000";
                                  bank_count <= bank_count + 1;
        when fifo2dataram_go  => addrd_count <= addrd_count + 1;
        when fifo2dataram_go2 => addrd_count <= "000000";
                                  bankd_count <= bankd_count + 1;
        when others           => null;
      end case;
    end if;
  end if;

end process addr_count_control;

-----------------------------------------------------------------------------
-- purpose: statemachine to reset KCPSM after a data write
-- type   : combinational
-- inputs : Reset_l, reset_kcpsm_state, kcpsm_write_strobe_in,
--          kcpsm_write_strobe_in, bankd_count, data_bank,
--          program_queue_count
-- outputs: nx_reset_kcpsm_state, k_int2, data_bank
-----------------------------------------------------------------------------
reset_kcpsm_state_machine: process (Reset_l, reset_kcpsm_state, bankd_count,
                                    kcpsm_write_strobe_in, kcpsm_port_id_in,
                                    kcpsm_output_in,
                                    data_bank, program_queue_count)

variable tmp_state : RESET_KCPSM_StateType;
variable tmp_k_int2, tmp_is_reload : std_logic;

begin -- process reset_kcpsm_state_machine

  -- default values
  tmp_state := reset_kcpsm_state;
  tmp_k_int2 := '0';
  tmp_is_reload := '0';

  if Reset_l = '0' then
    tmp_state := Idle;
  else
    case reset_kcpsm_state is
      when Idle       => if kcpsm_write_strobe_in = '1' and
                            kcpsm_port_id_in = "11111111" then
                           if kcpsm_output_in = "11111111" then
```

```
                        tmp_state := Check;
                      elsif kcpsm_output_in = "00000000" then
                        tmp_state := NW_Check;
                      end if;
                    end if;
      when Check      => if data_bank + 1 /= bankd_count then
                      tmp_state := K_Go1;
                    end if;
      when K_Go1      => tmp_state := K_Go2;
                    tmp_k_int2 := '1';
                    if std_logic_vector (program_queue_count) = "1" then
                      tmp_is_reload := '1';
                    end if;
      when K_Go2      => tmp_state := Idle;
                    tmp_k_int2 := '1';
      when NW_Check   => if data_bank /= bankd_count then
                      tmp_state := NW_Go1;
                    end if;
      when NW_Go1     => tmp_state := NW_Go2;
                    tmp_k_int2 := '1';
      when NW_Go2     => tmp_state := Idle;
                    tmp_k_int2 := '1';
    end case;
  end if;

  -- assign signals
  nx_reset_kcpsm_state <= tmp_state;
  k_int2 <= tmp_k_int2;
  is_reload <= tmp_is_reload;

end process reset_kcpsm_state_machine;

-------------------------------------------------------------------------------
-- purpose: logic to update the KCPSM pram dram address bank counter
-- type   : combinational
-- inputs : Reset_l, Clk
-- outputs: data_bank
-------------------------------------------------------------------------------
kcpsm_addr_count_control: process (Clk, Reset_l)

begin -- process write_queue_count_control

  if CLK'event and CLK = '1' then
    if Reset_l = '0' then
      prog_bank <= "0";
      data_bank <= "00";
    else
      case reset_kcpsm_state is
        when K_Go1     => data_bank <= data_bank + 1;
                    if std_logic_vector (program_queue_count) = "1" then
                      prog_bank <= prog_bank + 1;
                    end if;
        when others    => null;
      end case;
```

```
    end if;
  end if;
end process kcpsm_addr_count_control;

-------------------------------------------------------------------------------
-- purpose: logic to update the program queue count
-- type   : combinational
-- inputs : Reset_l, Clk
-- outputs: program_queue_count
-------------------------------------------------------------------------------
program_queue_count_control: process (Clk, Reset_l)

begin -- process program_queue_count_control

  if CLK'event and CLK = '1' then
    if Reset_l = '0' then
      program_queue_count <= "0";
    elsif is_loaded = '1' then
      program_queue_count <= program_queue_count + 1;
    elsif is_reload = '1' then
      program_queue_count <= program_queue_count - 1;
    end if;
  end if;
end process program_queue_count_control;

-------------------------------------------------------------------------------
-- purpose: logic to update the write queue count
-- type   : combinational
-- inputs : Reset_l, Clk
-- outputs: write_queue_count
-------------------------------------------------------------------------------
write_queue_count_control: process (Clk, Reset_l)

begin -- process write_queue_count_control

  if CLK'event and CLK = '1' then
    if Reset_l = '0' then
      write_queue_count <= "00";
    elsif kcpsm_write_strobe_in = '1' and kcpsm_port_id_in = "11111111" then
      if kcpsm_output_in = "11111111" then
        write_queue_count <= write_queue_count + 1;
      end if;
    elsif wr_done = '1' then
      write_queue_count <= write_queue_count - 1;
    end if;
  end if;
end process write_queue_count_control;

-------------------------------------------------------------------------------
-- purpose: statemachine to copy data from fifo to udp packet
-- type   : combinational
-- inputs : Reset_l, fifo2udp_state, is_start, is_program, sof_f, eof_f
-- outputs: nx_fifo2udp_state
-------------------------------------------------------------------------------
```

```
fifo2udp_state_machine: process (Reset_l, fifo2udp_state, is_start,
                                 is_program, sof_f, eof_f, dataen_f, data_f)

variable tmp_state : FIFO2UDP_StateType;
variable tmp_fifo2udp : std_logic;

begin -- process fifo2udp_state_machine

  -- default values
  tmp_state := fifo2udp_state;
  tmp_fifo2udp := '0';

  if Reset_l = '0' then
    tmp_state := Idle;
  else
    case fifo2udp_state is
      when Idle              => if is_start = '1' then
                                  tmp_state := Start;
                                end if;
      when Start             => if is_program = '1' and sof_f = '1' then
                                  tmp_state := fifo2udp_copy;
                                  tmp_fifo2udp := '1';
                                end if;
      when fifo2udp_copy     => if eof_f = '1' then
                                  tmp_state := fifo2udp_go;
                                end if;
                                tmp_fifo2udp := '1';
      when fifo2udp_go       => if dataen_f = '1' and data_f(15) = '1' then
                                  tmp_state := fifo2udp_go2;
                                end if;
                                tmp_fifo2udp := '1';
      when fifo2udp_go2      => tmp_state := Start;
                                tmp_fifo2udp := '1';
    end case;
  end if;

-- assign signals
nx_fifo2udp_state <= tmp_state;
fifo2udp <= tmp_fifo2udp;

end process fifo2udp_state_machine;

-------------------------------------------------------------------------------
-- purpose: statemachine to copy data from ram to udp packet
-- type   : combinational
-- inputs : Reset_l, ram2udp_state, is_start, KCPSM_WRITE_STROBE,
--          KCPSM_PORT_ID, wr_grant, payload_count, data_out
-- outputs: nx_ram2udp_state, ram2udp
-------------------------------------------------------------------------------
ram2udp_state_machine: process (Reset_l, ram2udp_state, pv_ram2udp_state,
                                write_queue_count,
                                wr_grant, payload_count, data_out)

variable tmp_state : RAM2UDP_StateType;
```

```
variable tmp_ram2udp, tmp_wr_done, tmp_enbd : std_logic;
variable tmp_dataen_buf, tmp_sof_buf, tmp_eof_buf, tmp_sod_buf : std_logic;

begin -- process ram2udp_state_machine

  -- default values
  tmp_state := ram2udp_state;
  tmp_ram2udp := '0';
  tmp_wr_done := '0';
  tmp_enbd := '0';
  tmp_dataen_buf := '0';
  tmp_sof_buf := '0';
  tmp_eof_buf := '0';
  tmp_sod_buf := '0';

  if Reset_l = '0' then
    tmp_state := Idle;
  elsif wr_grant = '1'then
    case ram2udp_state is
      when Idle              => if std_logic_vector(write_queue_count) /=
                                   "00" then
                                  tmp_state := ram2udp_sof;
                                end if;
      when ram2udp_sof       => tmp_state := ram2udp_ip_header;
                                tmp_enbd := '1';
                                tmp_sof_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_ip_header  => tmp_state := ram2udp_ip_header2;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_ip_header2 => tmp_state := ram2udp_ip_header3;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_ip_header3 => tmp_state := ram2udp_ip_header4;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_ip_header4 => tmp_state := ram2udp_ip_header5;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_ip_header5 => tmp_state := ram2udp_udp_header;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_udp_header => tmp_state := ram2udp_copy;
                                tmp_enbd := '1';
                                tmp_dataen_buf := '1';
                                tmp_sod_buf := '1';
                                tmp_ram2udp := '1';
      when ram2udp_copy       => tmp_state := ram2udp_copy2;
                                tmp_enbd := '1';
```

```
                              tmp_ram2udp := '1';
when ram2udp_copy2        => if pv_ram2udp_state = ram2udp_copy then
                                tmp_state := ram2udp_udp_header2;
                              else
                                tmp_state := ram2udp_copy;
                              end if;
                              tmp_enbd := '1';
                              tmp_ram2udp := '1';
when ram2udp_udp_header2=> if pv_ram2udp_state = ram2udp_copy2 then
                                if data_out (31 downto 16) = X"0008" then
                                  tmp_state := ram2udp_trail_copy;
                                else
                                  tmp_state := ram2udp_payload;
                                end if;
                                tmp_dataen_buf := '1';
                              else
                                tmp_state := ram2udp_copy;
                              end if;
                              tmp_enbd := '1';
                              tmp_ram2udp := '1';
when ram2udp_payload      => if std_logic_vector (payload_count) =
                                 "000011" then
                                tmp_state := ram2udp_trail_copy;
                                tmp_eof_buf := '1';
                              end if;
                              tmp_enbd := '1';
                              tmp_dataen_buf := '1';
                              tmp_ram2udp := '1';
when ram2udp_trail_copy => tmp_state := ram2udp_trail_copy2;
                              tmp_enbd := '1';
                              tmp_ram2udp := '1';
when ram2udp_trail_copy2=> if pv_ram2udp_state = ram2udp_trail_copy then
                                tmp_state := ram2udp_trailer;
                              else
                                tmp_state := ram2udp_trail_copy;
                              end if;
                              tmp_enbd := '1';
                              tmp_ram2udp := '1';
when ram2udp_trailer      => if pv_ram2udp_state=ram2udp_trail_copy2 then
                                if data_out (15) = '1' then
                                  tmp_state := ram2udp_trailer2;
                                else
                                  tmp_state := ram2udp_trail_copy;
                                end if;
                                tmp_dataen_buf := '1';
                              else
                                tmp_state := ram2udp_trail_copy;
                              end if;
                              tmp_enbd := '1';
                              tmp_ram2udp := '1';
when ram2udp_trailer2     => tmp_state := ram2udp_done;
                              tmp_enbd := '1';
                              tmp_dataen_buf := '1';
                              tmp_ram2udp := '1';
```

```
      when ram2udp_done        => tmp_state := Idle;
                                  tmp_wr_done := '1';
    end case;
  end if;

-- assign signals
nx_ram2udp_state <= tmp_state;
ram2udp <= tmp_ram2udp;
wr_done <= tmp_wr_done;
enbd2 <= tmp_enbd;
dataen_buf <= tmp_dataen_buf;
sof_buf <= tmp_sof_buf;
eof_buf <= tmp_eof_buf;
sod_buf <= tmp_sod_buf;

end process ram2udp_state_machine;

-------------------------------------------------------------------------------
-- purpose: control write output address counter
-- type    : sequential
-- inputs : Clk, Reset_1
-- outputs: addrd2_count, bankd2_count,
-------------------------------------------------------------------------------
addr2_count_control : process (Clk, Reset_l)

begin -- process addr2_count_control

  if CLK'event and CLK = '1' then
    if Reset_l = '0' then
      bankd2_count <= "00";
      addrd2_count <= "000000";
    elsif wr_grant = '1' then
      case ram2udp_state is
        when ram2udp_sof         => addrd2_count <= addrd2_count + 1;
        when ram2udp_ip_header   => addrd2_count <= addrd2_count + 1;
        when ram2udp_ip_header2  => addrd2_count <= addrd2_count + 1;
        when ram2udp_ip_header3  => addrd2_count <= addrd2_count + 1;
        when ram2udp_ip_header4  => addrd2_count <= addrd2_count + 1;
        when ram2udp_ip_header5  => addrd2_count <= addrd2_count + 1;
        when ram2udp_udp_header  => addrd2_count <= addrd2_count + 1;
        when ram2udp_udp_header2 => if pv_ram2udp_state = ram2udp_copy2 then
                                      addrd2_count <= addrd2_count + 1;
                                      payload_count <=
                                      unsigned (data_out (23 downto 18));
                                    end if;
        when ram2udp_payload     => addrd2_count <= addrd2_count + 1;
                                    payload_count <= payload_count - 1;
        when ram2udp_trailer     => addrd2_count <= addrd2_count + 1;
        when ram2udp_trailer2    => addrd2_count <= addrd2_count + 1;
        when ram2udp_done        => addrd2_count <= "000000";
                                    bankd2_count <= bankd2_count + 1;

        when others              => null;
      end case;
    end if;
```

```
    end if;
  end process addr2_count_control;

  ------------------------------------------------------------------------------
  -- set address line
  ------------------------------------------------------------------------------
  PROGRAM_ADDRB (6 downto 0) <= std_logic_vector (addr_count);
  PROGRAM_ADDRB (7 downto 7) <= std_logic_vector (bank_count);
  DATARAM_ADDRB (5 downto 0) <= std_logic_vector (addrd_count)
                                when ram2udp = '0' else
                                std_logic_vector (addrd2_count);
  DATARAM_ADDRB (7 downto 6) <= std_logic_vector (bankd_count)
                                when ram2udp = '0' else
                                std_logic_vector (bankd2_count);
  KCPSM_PROG_BANK <= std_logic (prog_bank(0));
  KCPSM_DATA_BANK <= std_logic_vector (data_bank);

  ------------------------------------------------------------------------------
  -- set data line
  ------------------------------------------------------------------------------
  FIFO_DIN2 (31 downto 0) <= data_in;
  FIFO_DIN2 (32) <= dataen_in;
  FIFO_DIN2 (33) <= sof_in;
  FIFO_DIN2 (34) <= eof_in;
  FIFO_DIN2 (35) <= sod_in;
  PROGRAM_DIB <= data_f;
  DATARAM_DIB <= data_f;

  ------------------------------------------------------------------------------
  -- set dpram control signal
  ------------------------------------------------------------------------------
  PROGRAM_ENB <= enb;
  PROGRAM_WEB <= web;
  DATARAM_ENB <= enbd when ram2udp = '0' else enbd2;
  DATARAM_WEB <= webd;

  KCPSM_INTERRUPT <= k_int2;

  ------------------------------------------------------------------------------
  -- wire output
  ------------------------------------------------------------------------------
  D_OUT_MOD      <= data_f when fifo2udp = '1' else data_out;
  DataEn_OUT_MOD <= dataen_f when fifo2udp = '1' else dataen_out;
  SOF_OUT_MOD    <= sof_f when fifo2udp = '1' else sof_out;
  EOF_OUT_MOD    <= eof_f when fifo2udp = '1' else eof_out;
  SOD_OUT_MOD    <= sod_f when fifo2udp = '1' else sod_out;

  tca_e          <= tca_out;
  ainit          <= not(reset_l);

end behavior;
```

The VHDL source code of the UDPEcho module is shown below.

```
-- $Id: udpecho.vhd,v 1.2 2001/07/26 18:42:55 hwf1 Exp $
--
-- KCPSM Interface
-- This interface parses a UDP datagram. If it is a KCPSM instruction code,
-- stores it into the dual port program ram. If it is a KCPSM data, stores
-- it into the dual port data ram. After KCPSM has processed the datagram,
-- it echoes back it out.
--
-- Author: Henry Fu
-- (c) 2001 Washington University, Applied Research Lab
--
-- this module changes src/dest ip-address and port of a UDP datagram
--

library IEEE;
use IEEE.std_logic_1164.all;

entity UDPEcho is

  port (
    CLK            : in  std_logic;     -- clock
    Reset_l        : in  std_logic;     -- synchronous reset, active low
    D_MOD_IN       : in  std_logic_vector (31 downto 0);      -- cell data
    DataEn_MOD_IN  : in  std_logic;     -- data enable
    SOF_MOD_IN     : in  std_logic;     -- start of frame
    EOF_MOD_IN     : in  std_logic;     -- end of frame
    SOD_MOD_IN     : in  std_logic;     -- start of datagram
    TCA_MOD_IN     : out std_logic;     -- congestion control
    D_OUT_MOD      : out std_logic_vector (31 downto 0);      -- cell data
    DataEn_OUT_MOD : out std_logic;     -- data enable
    SOF_OUT_MOD    : out std_logic;     -- start of frame
    EOF_OUT_MOD    : out std_logic;     -- end of frame
    SOD_OUT_MOD    : out std_logic;     -- start of datagram
    TCA_OUT_MOD    : in  std_logic);    -- congestion control

end UDPEcho;


architecture behavior of UDPEcho is

  ------------------------------------------------------------------------------
  -- input signals
  ------------------------------------------------------------------------------
  signal data_in   : std_logic_vector (31 downto 0);           -- cell data
  signal dataen_in : std_logic;             -- data enable
  signal sof_in    : std_logic;             -- start of frame
  signal eof_in    : std_logic;             -- end of frame
  signal sod_in    : std_logic;             -- start of datagram

  ------------------------------------------------------------------------------
```

```
-- previous data_in buffer
-------------------------------------------------------------------------------
signal data_pv   : std_logic_vector (31 downto 0);          -- cell data
signal dataen_pv : std_logic;              -- data enable
signal sof_pv    : std_logic;              -- start of frame
signal eof_pv    : std_logic;              -- end of frame
signal sod_pv    : std_logic;              -- start of datagram

-------------------------------------------------------------------------------
-- next data_in buffer
-------------------------------------------------------------------------------
signal data_nx   : std_logic_vector (31 downto 0);          -- cell data
signal dataen_nx : std_logic;              -- data enable
signal sof_nx    : std_logic;              -- start of frame
signal eof_nx    : std_logic;              -- end of frame
signal sod_nx    : std_logic;              -- start of datagram

-------------------------------------------------------------------------------
-- output signals
-------------------------------------------------------------------------------
signal data_out   : std_logic_vector (31 downto 0);         -- cell data
signal dataen_out : std_logic;             -- data enable
signal sof_out    : std_logic;             -- start of frame
signal eof_out    : std_logic;             -- end of frame
signal sod_out    : std_logic;             -- start of datagram

-------------------------------------------------------------------------------
-- state machine
-------------------------------------------------------------------------------
type StateType is (Idle, IPHdr1, IPHdr2, IPHdr3, SrcIPAddr, DestIPAddr,
                   UDPPort); -- states
signal state, nx_state : StateType;   -- current and new state


begin  -- behavior

-------------------------------------------------------------------------------
-- purpose: copy signals on clock edge
-- type    : sequential
-- inputs : CLK, Res,
-- outputs:
-------------------------------------------------------------------------------
clock_copy: process (CLK)
begin  -- process clock_copy

  if CLK'event and CLK = '1' then     -- rising clock edge

    -- previous buffer
    data_pv <= D_MOD_IN;
    dataen_pv <= DataEn_MOD_IN;
    sof_pv <= SOF_MOD_IN;
    eof_pv <= EOF_MOD_IN;
    sod_pv <= SOD_MOD_IN;
```

```
    -- input buffer
    data_in <= data_pv;
    dataen_in <= dataen_pv;
    sof_in <= sof_pv;
    eof_in <= eof_pv;
    sod_in <= sod_pv;

    -- next buffer
    data_nx <= data_in;
    dataen_nx <= dataen_in;
    sof_nx <= sof_in;
    eof_nx <= eof_in;
    sod_nx <= sod_in;

    -- statemachine
    state <= nx_state;

  end if;

end process clock_copy;

-----------------------------------------------------------------------------
-- purpose: state machine
-- type   : combinational
-- inputs : Reset_l, state, sof_in
-- outputs: nx_state
-----------------------------------------------------------------------------
state_machine: process (Clk, Reset_l, state, sof_in)
  variable tmp_state : StateType;     -- new state
  variable tmp_data  : std_logic_vector (31 downto 0);
begin  -- process state_machine

  -- default value
  tmp_state := state;
  tmp_data  := data_in;

  if Reset_l = '0' then
    tmp_state := Idle;
  else
    case state is
      when Idle       => if sof_in = '1' then
                           tmp_state := IPHdr1;
                         end if;
      when IPHdr1     => tmp_state := IPHdr2;
      when IPHdr2     => tmp_state := IPHdr3;
      when IPHdr3     => tmp_state := SrcIPAddr;
      when SrcIPAddr  => tmp_state := DestIPAddr;
                         tmp_data  := data_pv;
      when DestIPAddr => tmp_state := UDPPort;
                         tmp_data  := data_nx;
      when UDPPort    => tmp_state := Idle;
                         tmp_data (31 downto 16) := data_in (15 downto 0);
                         tmp_data (15 downto 0)  := data_in (31 downto 16);
    end case;
```

```
    end if;

    -- set state
    nx_state   <= tmp_state;
    data_out   <= tmp_data;
    dataen_out <= dataen_in;
    sof_out    <= sof_in;
    eof_out    <= eof_in;
    sod_out    <= sod_in;

  end process state_machine;

  ---------------------------------------------------------------------------
  -- wire output
  ---------------------------------------------------------------------------
  D_OUT_MOD <= data_out;
  DataEn_OUT_MOD <= dataen_out;
  SOF_OUT_MOD <= sof_out;
  EOF_OUT_MOD <= eof_out;
  SOD_OUT_MOD <= sod_out;
  TCA_MOD_IN  <= TCA_OUT_MOD;

end behavior;
```

The VHDL source code of the FPX KCPSM module is shown below.

```
-- $Id: module.vhd,v 1.3 2001/07/26 18:42:55 hwf1 Exp $
--
-- KCPSM Interface
-- This interface parses a UDP datagram. If it is a KCPSM instruction code,
-- stores it into the dual port program ram. If it is a KCPSM data, stores
-- it into the dual port data ram. After KCPSM has processed the datagram,
-- it echoes back it out.
--
-- Author: Henry Fu
-- (c) 2001 Washington University, Applied Research Lab
--
-- this file packages all necessary entities into an FPX module
--

library IEEE;
use IEEE.std_logic_1164.all;
library FPXLIB;
use FPXLIB.UDPProcessor.all;

ENTITY InterfaceModule IS
  PORT (
    -- Clock & Reset
    clk     : in STD_LOGIC;             -- 100MHz global clock
    reset_l : in STD_LOGIC;             -- Synchronous reset, asserted-low
```

```
    -- Enable & Ready
    -- Handshake for module reconfiguration.
    enable_l : in  STD_LOGIC;            -- Asserted low
    ready_l  : out STD_LOGIC;            -- 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;

    -- Test Data Output
    test_data      : out STD_LOGIC_VECTOR(31 downto 0));

end InterfaceModule;

architecture struc of InterfaceModule is

  ------------------------------------------------------------------------------
  -- Interface
  ------------------------------------------------------------------------------
  component Interface
    port (
      CLK             : in  std_logic;                      -- clock
      Reset_l         : in  std_logic;                      -- reset

      D_MOD_IN        : in  std_logic_vector (31 downto 0); -- data
      DataEn_MOD_IN   : in  std_logic;                      -- data enable
      SOF_MOD_IN      : in  std_logic;                      -- start of frame
      EOF_MOD_IN      : in  std_logic;                      -- end of frame
      SOD_MOD_IN      : in  std_logic;                      -- start of datagram
      TCA_MOD_IN      : out std_logic;                      -- congestion control

      D_OUT_MOD       : out std_logic_vector (31 downto 0); -- data
      DataEn_OUT_MOD  : out std_logic;                      -- data enable
      SOF_OUT_MOD     : out std_logic;                      -- start of frame
      EOF_OUT_MOD     : out std_logic;                      -- end of frame
      SOD_OUT_MOD     : out std_logic;                      -- start of datagram
      TCA_OUT_MOD     : in  std_logic;                      -- congestion control

      KCPSM_INTERRUPT: out std_logic;                       -- KCPSM Interrupt
      KCPSM_READ_STROBE : in std_logic;                     -- KCPSM read strobe
      KCPSM_WRITE_STROBE : in std_logic;
      KCPSM_PROG_BANK: out std_logic;                       -- KCPSM prog bank
      KCPSM_DATA_BANK: out std_logic_vector (1 downto 0);   -- KCPSM data bank
      KCPSM_PORT_ID   : in  std_logic_vector (7 downto 0);  -- KCPSM port id
      KCPSM_OUTPUT    : in  std_logic_vector (7 downto 0);

      PROGRAM_ADDRB   : out std_logic_vector (7 downto 0);  -- dpram address line
```

```
    PROGRAM_DIB    : out std_logic_vector (31 downto 0); -- dpram data in line
    PROGRAM_WEB    : out std_logic;                      -- dpram write enable
    PROGRAM_ENB    : out std_logic;                      -- dpram chip enable
    PROGRAM_DOB    : in  std_logic_vector (31 downto 0); -- dpram dataout line

    DATARAM_ADDRB  : out std_logic_vector (7 downto 0);
    DATARAM_DIB    : out std_logic_vector (31 downto 0);
    DATARAM_WEB    : out std_logic;
    DATARAM_ENB    : out std_logic;
    DATARAM_DOB    : in  std_logic_vector (31 downto 0));
end component;

-----------------------------------------------------------------------------
-- KCPSM
-----------------------------------------------------------------------------
component kcpsm port
       (ADDR          : out std_logic_vector(7 downto 0);
        I             : in  std_logic_vector(15 downto 0);
        INPUT         : in  std_logic_vector(7 downto 0);
        OUTPUT        : out std_logic_vector(7 downto 0);
        PORT_ID       : out std_logic_vector(7 downto 0);
        READ_STROBE   : out std_logic;
        WRITE_STROBE  : out std_logic;
        INTERRUPT     : in  std_logic;
        CLK           : in  std_logic);
end component;

-----------------------------------------------------------------------------
-- DPRAM
-----------------------------------------------------------------------------
component program
  port (
    addra: IN std_logic_VECTOR(8 downto 0);
    clka: IN std_logic;
    addrb: IN std_logic_VECTOR(7 downto 0);
    clkb: IN std_logic;
    dia: IN std_logic_VECTOR(15 downto 0);
    wea: IN std_logic;
    dib: IN std_logic_VECTOR(31 downto 0);
    web: IN std_logic;
    ena: IN std_logic;
    enb: IN std_logic;
    rsta: IN std_logic;
    rstb: IN std_logic;
    doa: OUT std_logic_VECTOR(15 downto 0);
    dob: OUT std_logic_VECTOR(31 downto 0));
end component;

-----------------------------------------------------------------------------
-- DATARAM
-----------------------------------------------------------------------------
component dataram
  port (
    addra: IN std_logic_VECTOR(9 downto 0);
```

```
    clka: IN std_logic;
    addrb: IN std_logic_VECTOR(7 downto 0);
    clkb: IN std_logic;
    dia: IN std_logic_VECTOR(7 downto 0);
    wea: IN std_logic;
    dib: IN std_logic_VECTOR(31 downto 0);
    web: IN std_logic;
    ena: IN std_logic;
    enb: IN std_logic;
    rsta: IN std_logic;
    rstb: IN std_logic;
    doa: OUT std_logic_VECTOR(7 downto 0);
    dob: OUT std_logic_VECTOR(31 downto 0));
end component;

-------------------------------------------------------------------------------
-- signals kcpsm - dpram
-------------------------------------------------------------------------------
signal addra : std_logic_vector (8 downto 0);
signal clka  : std_logic;
signal dia   : std_logic_vector (15 downto 0);
signal wea   : std_logic;
signal ena   : std_logic;
signal rsta  : std_logic;
signal doa   : std_logic_vector (15 downto 0);


-------------------------------------------------------------------------------
-- signals interface - dpram
-------------------------------------------------------------------------------
signal addrb : std_logic_vector (7 downto 0);
signal clkb  : std_logic;
signal dib   : std_logic_vector (31 downto 0);
signal web   : std_logic;
signal enb   : std_logic;
signal rstb  : std_logic;
signal dob   : std_logic_vector (31 downto 0);

-------------------------------------------------------------------------------
-- signals kcpsm - dataram
-------------------------------------------------------------------------------
signal addrad : std_logic_vector (9 downto 0);
signal clkad  : std_logic;
signal diad   : std_logic_vector (7 downto 0);
signal wead   : std_logic;
signal enad   : std_logic;
signal rstad  : std_logic;
signal doad   : std_logic_vector (7 downto 0);

-------------------------------------------------------------------------------
-- signals interface - dataram
-------------------------------------------------------------------------------
signal addrbd : std_logic_vector (7 downto 0);
signal clkbd  : std_logic;
```

```
    signal dibd   : std_logic_vector (31 downto 0);
    signal webd   : std_logic;
    signal enbd   : std_logic;
    signal rstbd  : std_logic;
    signal dobd   : std_logic_vector (31 downto 0);

    signal interrupt : std_logic;
    signal read_strobe : std_logic;
    signal nwead : std_logic;

    -----------------------------------------------------------------------------
    -- signals udpproc - udpecho
    -----------------------------------------------------------------------------
    signal data_u2h : std_logic_vector (31 downto 0);  -- data
    signal dataen_u2h : std_logic;                     -- data enable
    signal sof_u2h : std_logic;                        -- start of frame
    signal eof_u2h : std_logic;                        -- end of frame
    signal sod_u2h : std_logic;                        -- start of payload
    signal tca_u2h : std_logic;                        -- congestion control

    -----------------------------------------------------------------------------
    -- signals hellobob - udpproc
    -----------------------------------------------------------------------------
    signal data_h2u : std_logic_vector (31 downto 0);  -- data
    signal dataen_h2u : std_logic;                     -- data enable
    signal sof_h2u : std_logic;                        -- start of frame
    signal eof_h2u : std_logic;                        -- end of frame
    signal sod_h2u : std_logic;                        -- start of payload
    signal tca_h2u : std_logic;                        -- congestion control

begin  -- struc

    -----------------------------------------------------------------------------
    -- UDPWrapper
    -----------------------------------------------------------------------------
    upw : UDPWrapper port map (
      CLK           => CLK,
      Reset_l       => reset_l,
      Enable_l      => enable_l,
      Ready_l       => ready_l,
      SOC_MOD_IN    => SOC_MOD_IN,
      D_MOD_IN      => D_MOD_IN,
      TCA_MOD_IN    => TCA_MOD_IN,
      SOC_OUT_MOD   => SOC_OUT_MOD,
      D_OUT_MOD     => D_OUT_MOD, -- open
      TCA_OUT_MOD   => TCA_OUT_MOD,
      D_OUT_APPL       => data_u2h,
      DataEn_OUT_APPL => dataen_u2h,
      SOF_OUT_APPL     => sof_u2h,
      EOF_OUT_APPL     => eof_u2h,
      SOD_OUT_APPL     => sod_u2h,
      TCA_OUT_APPL     => tca_u2h,
      D_APPL_IN        => data_h2u,
      DataEn_APPL_IN   => dataen_h2u,
```

```
    SOF_APPL_IN      => sof_h2u,
    EOF_APPL_IN      => eof_h2u,
    SOD_APPL_IN      => sod_h2u,
    TCA_APPL_IN      => tca_h2u);

--------------------------------------------------------------------------------
-- Interface
--------------------------------------------------------------------------------
inter : Interface port map (
    CLK              => CLK,
    Reset_l          => reset_l,
    D_MOD_IN         => data_u2h,
    DataEn_MOD_IN    => dataen_u2h,
    SOF_MOD_IN       => sof_u2h,
    EOF_MOD_IN       => eof_u2h,
    SOD_MOD_IN       => sod_u2h,
    TCA_MOD_IN       => tca_u2h,
    D_OUT_MOD        => data_h2u,
    DataEn_OUT_MOD   => dataen_h2u,
    SOF_OUT_MOD      => sof_h2u,
    EOF_OUT_MOD      => eof_h2u,
    SOD_OUT_MOD      => sod_h2u,
    TCA_OUT_MOD      => tca_h2u,
    KCPSM_INTERRUPT=> interrupt,
    KCPSM_READ_STROBE => read_strobe,
    KCPSM_WRITE_STROBE => nwead,
    KCPSM_PROG_BANK=> addra(8),
    KCPSM_DATA_BANK=> addrad(9 downto 8),
    KCPSM_PORT_ID    => addrad(7 downto 0),
    KCPSM_OUTPUT     => diad,
    PROGRAM_ADDRB    => addrb,
    PROGRAM_DIB      => dib,
    PROGRAM_WEB      => web,
    PROGRAM_ENB      => enb,
    PROGRAM_DOB      => dob,
    DATARAM_ADDRB    => addrbd,
    DATARAM_DIB      => dibd,
    DATARAM_WEB      => webd,
    DATARAM_ENB      => enbd,
    DATARAM_DOB      => dobd);

--------------------------------------------------------------------------------
-- DPRAM
--------------------------------------------------------------------------------
myprogram : program port map (
    addra              => addra,
    clka               => clka,
    addrb              => addrb,
    clkb               => clkb,
    dia                => dia,
    wea                => wea,
    dib                => dib,
    web                => web,
    ena                => ena,
```

```
  enb              => enb,
  rsta             => rsta,
  rstb             => rstb,
  doa              => doa,
  dob              => dob);

------------------------------------------------------------------------------
-- DATARAM
------------------------------------------------------------------------------
mydataram : dataram port map (
  addra            => addrad,
  clka             => clkad,
  addrb            => addrbd,
  clkb             => clkbd,
  dia              => diad,
  wea              => wead,
  dib              => dibd,
  web              => webd,
  ena              => enad,
  enb              => enbd,
  rsta             => rstad,
  rstb             => rstbd,
  doa              => doad,
  dob              => dobd);

------------------------------------------------------------------------------
-- KCPSM
------------------------------------------------------------------------------
cpu : kcpsm port map (
  addr         => addra(7 downto 0),
  i            => doa,
  input        => doad,
  output       => diad,
  port_id      => addrad(7 downto 0),
  read_strobe  => read_strobe,
  write_strobe => nwead,
  interrupt    => interrupt,
  clk          => clk);

------------------------------------------------------------------------------
-- dummy
------------------------------------------------------------------------------
clka <= clk;
dia <= (others => '0');
wea <= '1';
ena <= '1';
rsta <= reset_l;

clkb <= clk;
rstb <= reset_l;

clkad <= clk;
wead <= not(nwead);
enad <= '1';
```

```
  rstad <= reset_l;

  clkbd <= clk;
  rstbd <= reset_l;

  test_data (31) <= sof_h2u;
  test_data (30) <= eof_h2u;
  test_data (29) <= sod_h2u;
  test_data (28) <= dataen_h2u;
  test_data (27 downto 16) <= data_h2u (11 downto 0);
  test_data (15) <= sof_u2h;
  test_data (14) <= eof_u2h;
  test_data (13) <= sod_u2h;
  test_data (12) <= dataen_u2h;
  test_data (11 downto 0) <= data_u2h (11 downto 0);

end struc;

-- synthesis translate_off
configuration interface_conf of InterfaceModule is

  for struc
    for all : UDPWrapper
      use configuration fpxlib.udpwrapper_conf;
    end for;
  end for;

end interface_conf;
-- synthesis translate_on
```

9.2 Simulation Results

A simulation testbench has been setup to test the functionality of this module. In order to inject fake ATM cells into the testbench, a C program called IP2FAKE is written to accomplish this task.

The command line option used is: `ip2fake filename.TBP`.

In order to generate a fake program packets/cells for simulation, use the ".TBP" file generated by the CONVERT program as the input to the IP2FAKE program. In order to generate a fake data packets/cells for simulation, use a plain text editor to modify the following sample ".TBP" file. Please note that the first word of the payload must be 0x00000001 to indicate that it is a data packet/cell.

```
# demo testbench
#
# comments start with #
# commands start with !
# parameters are optional
#
# UDP block, will be prepended by an UDP header
# parameter: ip-address, dest-port, src-port
!UDP 192.168.10.1 7 64000
00000001
6C6C6548
0000006F
```

After the IP2FAKE program is executed, a `TESTCELL.DAT` is produced. This file is the fake ATM cells and can be used by the simulation testbench for testing. The IP2FAKE program is part of the IPTESTBENCH program. For detailed instructions on how to use the IPTESTBENCH, please refer to the IPTESTBENCH webpage [13].

Some waveforms demonstrating the fucntionality of the circuit are shown below.

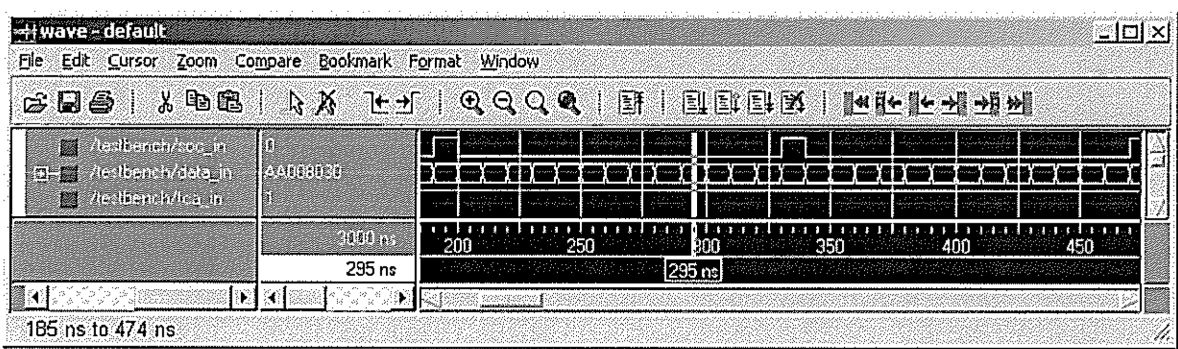

Figure 9: Arrival of the CHKHELLO ATM Cell

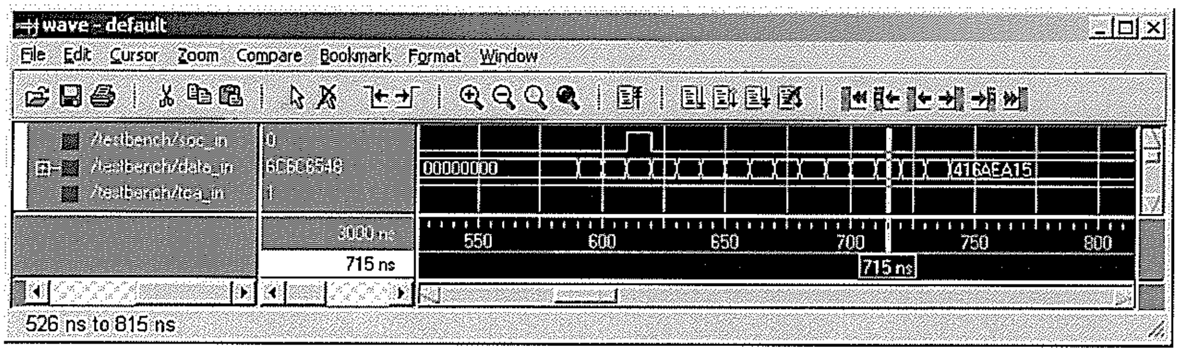

Figure 10: Arrival of the 'Hello' ATM Cell

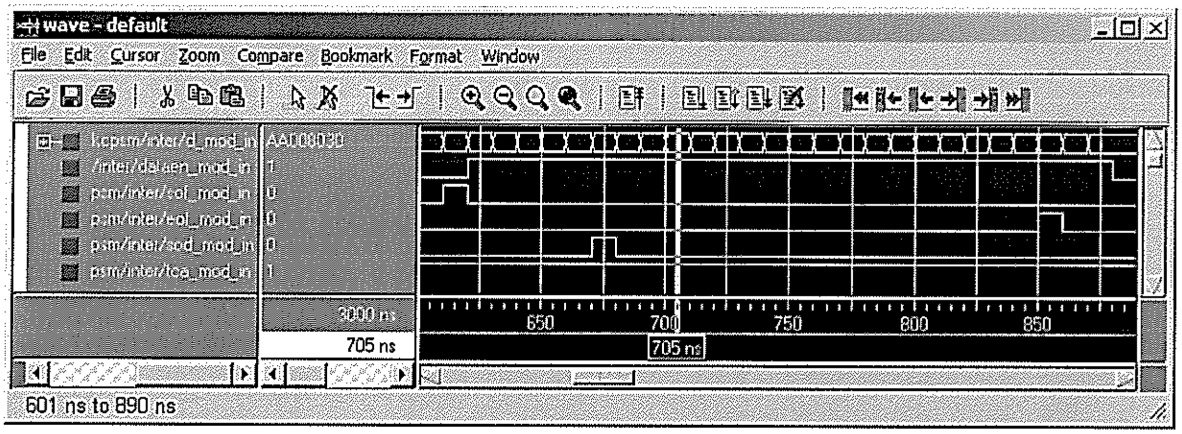

Figure 11: Arrival of the CHKHELLO Program Packet

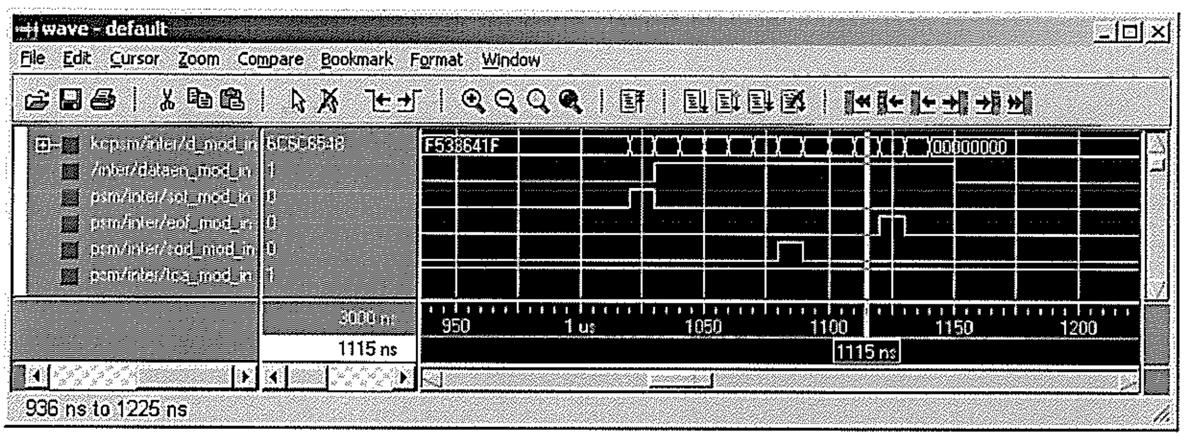

Figure 12: Arrival of the 'Hello' Data Packet

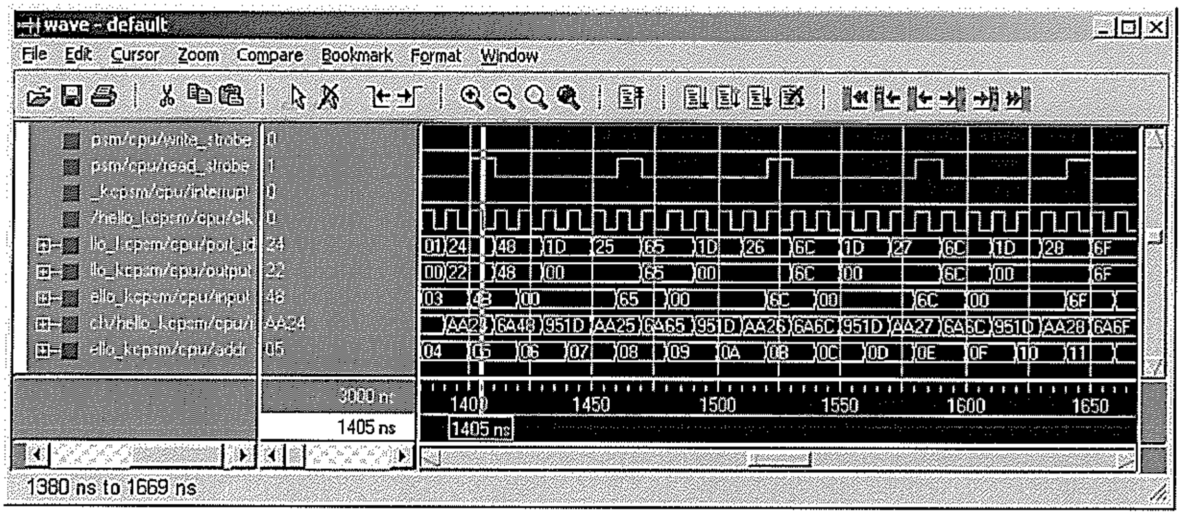

Figure 13: KCPSM Process that looks for the string 'Hello'

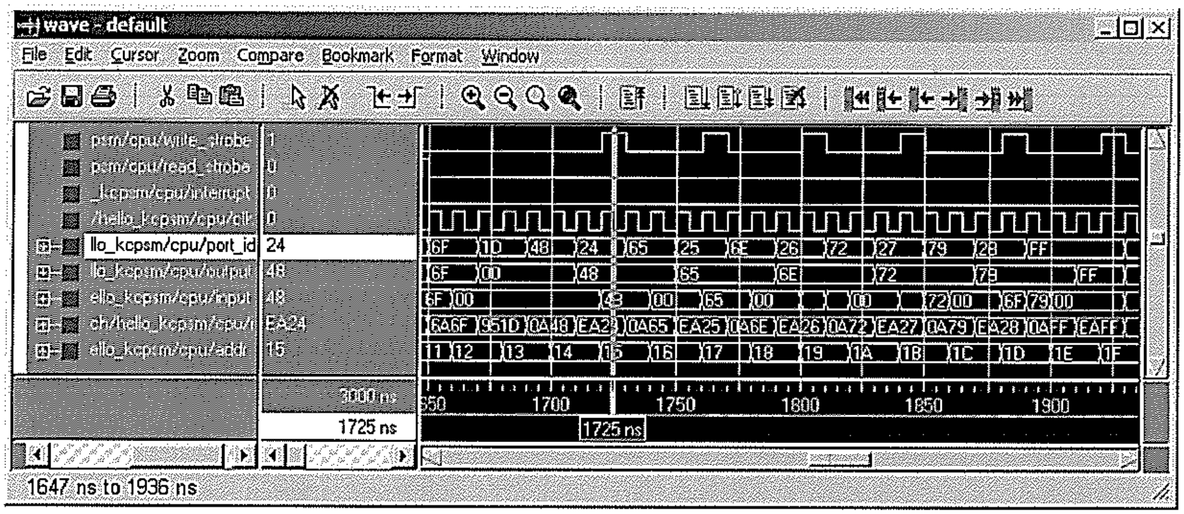

Figure 14: KCPSM Process that writes out the string 'Henry'

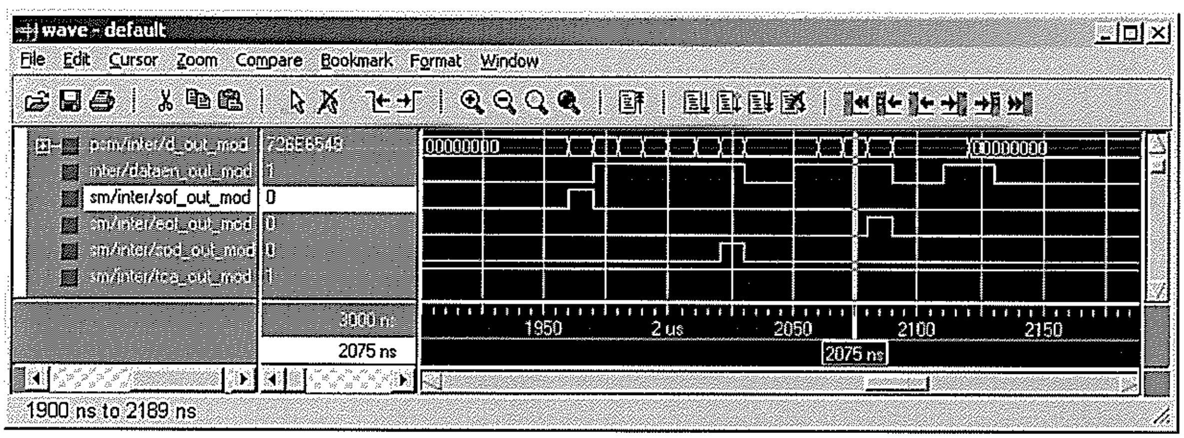

Figure 15: Departure of the 'Henry' Data Packet

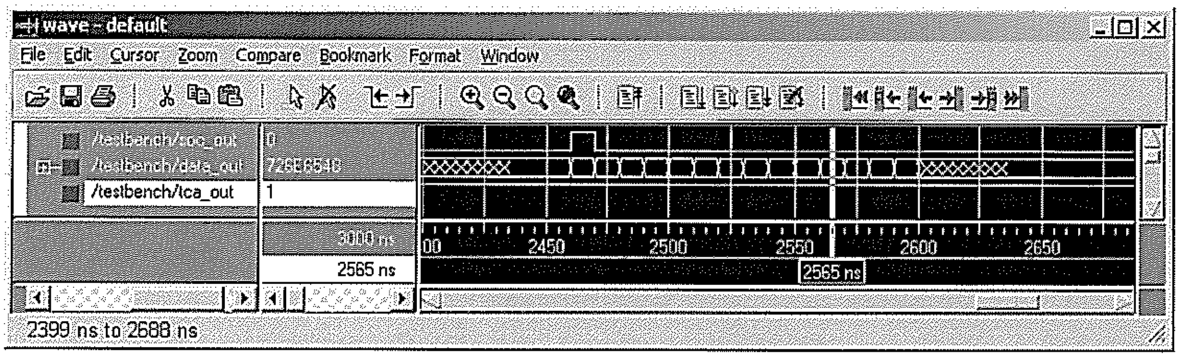

Figure 16: Departure of the 'Henry' ATM Cell

9.3 Synthesis Results

The circuit, including the KCPSM, Program Memory, Data Memory, Interface, and the Protocol Wrappers, runs at 70MHz and occupies 35% of a Xilinx XCV1000E-7-FG680.

A summary of th synthesis results is shown below:

- Maximum Frequency: 70 MHz
- Chip Utilization: 35% (4305 / 12288 slices)
- External Input Buffers: 69 uses
- External Output Buffers: 105 uses
- Total LUTs: 3807 uses

9.4 Xilinx Backend Synthesis Results

The script used to synthesize the circuit is shown below.

```
#!/bin/bash

part=xcv1000e-7-fg680
design=rad_loopback

ngdbuild -p ${part} ${design} -uc ${design}.ucf
map -p ${part} -o top.ncd ${design}.ngd ${design}.pcf
par  -w -ol 2 top.ncd ${design}.ncd ${design}.pcf
trce ${design}.ncd ${design}.pcf -e 3 -o ${design}.twr -xml ${design}_trce.xml
bitgen ${design}.ncd -b  -l -w -f bitgen.ut
```

9.5 Laboratory Results

The FPX KCPSM module was downloaded to the FPX. UDP packets are launched from the FPX controller (fpx.arl) into the system. IP over ATM was configured for IP address 192.168.10.1 and VPI/VCI 100/101. UDP packets were transmitted and received on fpx.arl using a C program called UDPTEST. In order to send a program packet, use the ".TBP" file generated by the CONVERT program as the input to UDPTEST. In order to send a data packet, use the ".TBP" file generated during simulation as input.

The command line used to send a UDP datagram is: `udptest < filename.TBP`.

The source code for that program is shown below.

```
/* A C program used to send and receive a UDP datagram */
/* There is an optional verbose mode [-v] */
/* By Henry Fu */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

#define MYPORT 3490     /* the source port */
#define TOPORT 8128     /* the destination port */
#define MYSIZE 1536     /* the deault datagram size will be 1536 bytes */

/* Globals for getting command line arguments */
static char *options = "v";
static char *usage = "[-v]";
int vflg = 0, errflg = 0;
char *host;
int port = TOPORT;
int myport = MYPORT;
int bufsize = MYSIZE;
int size;
unsigned char *buf, *rbuf, *lbuf;
int i;
int x0, x1, x2, x3, y0, y1, y2, y3;

/* Required by getopt */
extern char *optarg;
extern int optind, opterr;

/* Function to get command line arguments */
void parseargs(int argc, char *argv[])
{
    int c;
    int ok = 1;

    while ((c=getopt(argc,argv,options)) != -1)
        switch (c)
        {
        case 'v':
          vflg++;
          break;
```

```
        default:
          ok = 0;
        }

    if (!ok || optind < argc) {
      fprintf (stderr, "usage : %s %s\n", argv[0], usage);
      exit(0);
    }
}

/* Function to read input */
int parseinput()
{
    int i = 0;

    while (fgets(lbuf, 80, stdin) != 0) {
      switch (lbuf[0])
      {
      case '#':
        if (vflg) fprintf (stdout, "%s",lbuf);
        break;
      case '!':
        if (vflg) fprintf (stdout, "%s", lbuf);
        sscanf(lbuf, "!UDP %s %d %d\n", host, &port, &myport);
        break;
      case '\n':
        if (vflg) fprintf (stdout, "%s", lbuf);
        break;
      default :
        if (vflg) fprintf (stdout, "%s", lbuf);
        sscanf(lbuf, "%2x%2x%2x%2x", &x0, &x1, &x2, &x3);
        buf[i] = (char) x0;
        buf[i+1] = (char) x1;
        buf[i+2] = (char) x2;
        buf[i+3] = (char) x3;
        i += 4;
        break;
      }
    }
    if (vflg) fprintf( stdout, "size: %d\n", i);
    return i;
}

int main(int argc, char *argv[])
{
    int sockfd;
    struct hostent *he;
    struct sockaddr_in my_addr;
    struct sockaddr_in their_addr; /* connector's address information */

    buf = calloc(bufsize, sizeof(char));
    rbuf = calloc(bufsize, sizeof(char));
    lbuf = calloc(80, sizeof(char));
    host = calloc(80, sizeof(char));
```

```
parseargs(argc, argv);
size = parseinput();

if (vflg) fprintf (stdout, "UDPTest is getting host name ...\n");

if ((he=gethostbyname(host)) == NULL) {  /* get the host info */
    perror("UDPTest failed at gethostbyname");
    exit(1);
}

if (vflg) fprintf (stdout, "UDPTest is allocating socket ...\n");

if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("UDPTest failed at socket");
    exit(1);
}

my_addr.sin_family = AF_INET;
my_addr.sin_port = htons(myport);
my_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(my_addr.sin_zero), 8);

if (vflg) fprintf (stdout, "UDPTest is binding ...\n");

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))
                                                             == -1) {
    perror("UDPTest failed at bind");
    exit(1);
}

their_addr.sin_family = AF_INET;      /* host byte order */
their_addr.sin_port = htons(port);    /* short, network byte order */
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
bzero(&(their_addr.sin_zero), 8);     /* zero the rest of the struct */

if (vflg) fprintf (stdout, "UDPTest is sending ...\n");

if (sendto(sockfd, buf, size, 0,
           (const struct sockaddr *) &their_addr, sizeof(their_addr)) < 0) {
    perror("UDPTest failed at sendto");
    exit(1);
}

fprintf (stdout, "Sent: \n");
for(i=0; i<size; i+=4) {
  x0 = buf[i];
  x1 = buf[i+1];
  x2 = buf[i+2];
  x3 = buf[i+3];
  fprintf (stdout, "%.2X%.2X%.2X%.2X\n",x0,x1,x2,x3);
}

if (vflg) fprintf (stdout, "UDPTest is receiving ...\n");
```

```
    if ((size = read(sockfd, rbuf, bufsize)) < 0) {
        perror("UDPTest failed at read");
        exit(1);
    }

    fprintf (stdout, "Received: \n");
    for(i=0; i<size; i+=4) {
      y0 = rbuf[i];
      y1 = rbuf[i+1];
      y2 = rbuf[i+2];
      y3 = rbuf[i+3];
      fprintf (stdout, "%.2X%.2X%.2X%.2X\n",y0,y1,y2,y3);
    }

    close(sockfd);

    return 0;
}
```

In the first test, the C program just sent a UDP packet with the content of 'Hello'. After receiving this packet, the FPX KCPSM module just echoed back the same UDP packet. In the second test, the C program first sent a UDP packet with the CHKHELLO program and then sent a UDP packet with the content of 'Hello'. After receiving these two packets, the FPX KCPSM module first echoed back the same CHKHELLO program packet and then replied with a UDP packet with the content of 'Henry'.

10 Distribution

All of the files in this project are included in a single TAR file. Detailed description of the directory structure of the TAR file beginning from the root level is discussed below.

- `/iptestbench/` It includes the source files, input/output files, and compile script for the `IP2FAKE` program.
- `/udptestbench/` It includes the soruce files, input files, and compile script for the `UDPTEST` program.
- `/FPX_ROOT/RAD/MODULES/KCPSM/`
  - `coregen/` It includes the COREGEN project files and the output files for the COREGEN components used in this module.
  - `package/` It includes the Xilinx KCPSM Package. It includes the `KCPSMBLE` assembler, the `PSMDEBUG` debugger, and other relevant files.
    - `convert/` It includes the C source files and the compile script for the `CONVERT` program.
    - `chkhello/` It includes the assembly source files for the string replacement program.
    - `chksum/` It includes the assembly source files for a test program to shorten the data packet by appending fake UDP checksum in the trailer.
    - `test/` It includes the assembly source files for the simple counter program.
  - `sim/` It includes the VHDL source files, the input/output files, and the simulation script for the testbench that simulates this module in Modelsim.

–syn/ It includes the script used to generate the simulation VHDL files for the KCPSM and COREGEN components.

–vhdl/ It includes the VHDL source files for this module.

• /FPX_ROOT/RAD/MODULES/LIB/

  – CellProcessor/

    * coregen/ It includes the COREGEN project files and the output files for the COREGEN components used in the Protocol Wrappers.
    * sim/ It includes the compile script used to compile the CellProcessor.
    * syn/ It includes the script used to generate the simulation VHDL files and the EDIF file for the CellProcessor.
    * vhdl/ It includes the VHDL source files for the CellProcessor.

  – FrameProcessor/

    * crc32/ It includes the source files for the CRC32 library that is used by the FrameProcessor.
    * sim/ It includes the compile script used to compile the FrameProcessor.
    * syn/ It includes the script used to generate the simulation VHDL files and the EDIF file for the FrameProcessor.
    * vhdl/ It includes the VHDL source files for the FrameProcessor.

  – IPProcessor/

    * sim/ It includes the compile script used to compile the IPProcessor.
    * syn/ It includes the script used to generate the simulation VHDL files and the EDIF file for the IPProcessor.
    * vhdl/ It includes the VHDL source files for the IPProcessor.

  – UDPProcessor/

    * sim/ It includes the compile script used to compile the UDPProcessor.
    * syn/ It includes the script used to generate the simulation VHDL files and the EDIF file for the UDPProcessor.
    * vhdl/ It includes the VHDL source files for the UDPProcessor.

  – simprims/ It includes the Xilinx compiled library used by the simulation VHDL files.

• /FPX_ROOT/RAD/TOP/RAD_KCPSM/

  – sim/ It includes the compile script to compile and simulate the top-level design of this module.

  – syn/ It includes the Synplicity Pro project files of the top-level design of this module.

    * rad-xcve1000/ It includes the output files generated by the Xilinx backend tools and the resulting bit file of the top-level design of this module.

  – vhdl/ It includes the VHDL source files of the top-level design of this module.

11 Exercises

• Assemble and Debug a KCPSM Program. Edit the sample KCPSM program so that it replies with your name after a UDP datagram with the string 'Goodbye' was sent.

  – Edit the source ".PSM" file.
  – Assemble the KCPSM program.
  – Run PSMDEBUG.

• Convert the above KCPSM program into fake ATM cells.

–Use the CONVERT program to generate ".TBP" file.
–Use the IP2FAKE program to generate fake ATM cell.

- Generate new data packets and convert them into fake ATM cells.
  –Use the sample ".TBP" file as a general guide.
  –Use the IP2FAKE program to generate fake ATM cell.
- Send UDP datagram to test your new design.
  –Use the UDPTEST program to send program and data UDP datagrams.

12 Conclusion

This project demonstrates how a softcore processor can be embedded into an FPX module and how to use the Protocol Wrappers to implement active networking functions for the data and programs carried in Internet-Protocol packets. The FPX KCPSM module combines the flexibility of a software program and the performance benefits of a hardware module to allow the content of the Program Memory to be dynamically reprogrammed over the network through the use of UDP datagrams. Although the current implementation is targeted to work with the KCPSM in the WUGS and the FPX research environment, it can be adapted to work with other softcore processors in other FPGA-based system. It synthesizes to run at 70MHz and use 35% of the available gates on a Xilinx XCV1000E-7-FG680.

13 Acknowledgements

The authors would like to thank Ken Chapman of the Xilinx Corp. for his development of the KCPSM and Dave Parlour of the Xilinx Corp. for his support on the development of this project.

References

[1] "Field Programmable Port Extender Homepage." Online `http://www.arl.wustl.edu/arl-/projects/fpx/`, Aug. 2000.

[2] F. Braun, J. W. Lockwood, and M. Waldvogel, "Layered protocol wrappers for internet packet processing in reconfigurable hardware," Tech. Rep. WU-CS-01-10, Washington University in Saint Louis, Department of Computer Science, June 2001.

[3] K. Chapman, "8-Bit Microcontroller for Virtex Devices." Xilinx XAPP213, Online `http://www.-xilinx.com/xapp/xapp213.pdf`, Oct. 2000.

[4] J. W. Lockwood, "An open platform for development of network processing modules in reprogrammable hardware," in *IEC DesignCon'01*, (Santa Clara, CA), pp. WB–19, Jan. 2001.

[5] Xilinx Corp., "MicroBlaze Soft Processor Overview." Online `http://www.xilinx.com/-ipcenter/processor_central/microblaze.htm`, May 2001.

[6] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field programmable port extender (FPX) for distributed routing and queuing," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2000)*, (Monterey, CA, USA), pp. 137–144, Feb. 2000.

[7] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," in *ACM International Symposium on Field Programmable Gate Arrays (FPGA'2001)*, (Monterey, CA, USA), pp. 87–93, Feb. 2001.

[8] T. Chaney, J. A. Fingerhut, M. Flucke, and J. S. Turner, "Design of a gigabit ATM switch," Tech. Rep. WU-CS-96-07, Washington University in Saint Louis, 1996.

[9] D. E. Taylor, J. W. Lockwood, and N. Naufel, "RAD Module Infrastructure of the Field-programmable Port eXtender (FPX)," tech. rep., WUCS-01-16, Washington University, Department of Computer Science, July 2001.

[10] K. Chapman, "200 MHz UART with Internal 16-Byte Buffer." Xilinx XAPP223, Online `http://www.xilinx.com/xapp/xapp223.pdf`, Jan. 2001.

[11] F. Braun, J. Lockwood, and M. Waldvogel, "Reconfigurable router modules using network protocol wrappers," in *to appear: Proceedings of Field-Programmable Logic and Applications*, (Belfast, Northern Ireland), pp. xx–xx, Aug. 2001.

[12] "Wrapper webpage." `http://www.arl.wustl.edu/arl/projects/fpx/wrappers`, July 2001.

[13] "IP testbench webpage." `http://www.arl.wustl.edu/arl/projects/fpx/-research/iptestbench.html`, July 2001.

Additional Information about the FPX is available on-line:
`http://www.arl.wustl.edu/arl/projects/fpx/`