Audio Reverberator over IP

Ari Klein
Ashish Sharma
George Sirois
Sambuddho Chakravarty

CSEE 4840
Embedded Systems Design
Spring 2006
Final Project

Table of Contents

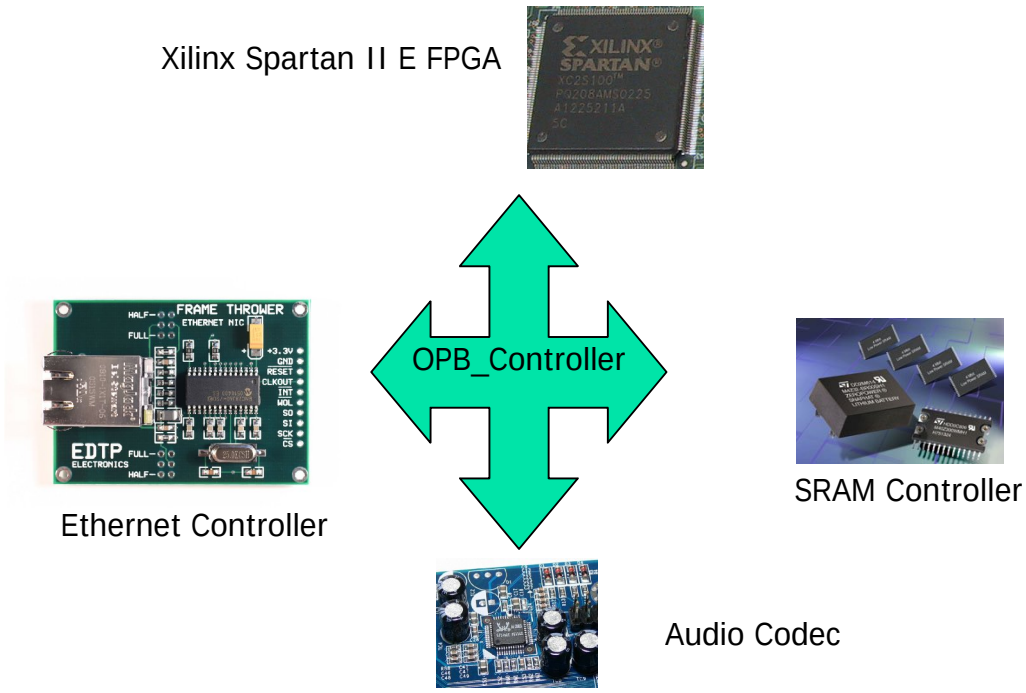
1. Introduction

1.1 Abstract

Our team set out to create a low cost and easy to implement audio effects processor for use with VoIP applications using the available peripherals on the Spartan 2E FPGA board. To achieve this, two FPGAs are connected via a crossover cable. Both FPGAs take data from the off-chip ADC, process it in the Microblaze, and send it to each other over Ethernet. The received data is sent out the DAC. Thus, audio gets processed, transmitted, and received.

1.2 Hardware Components

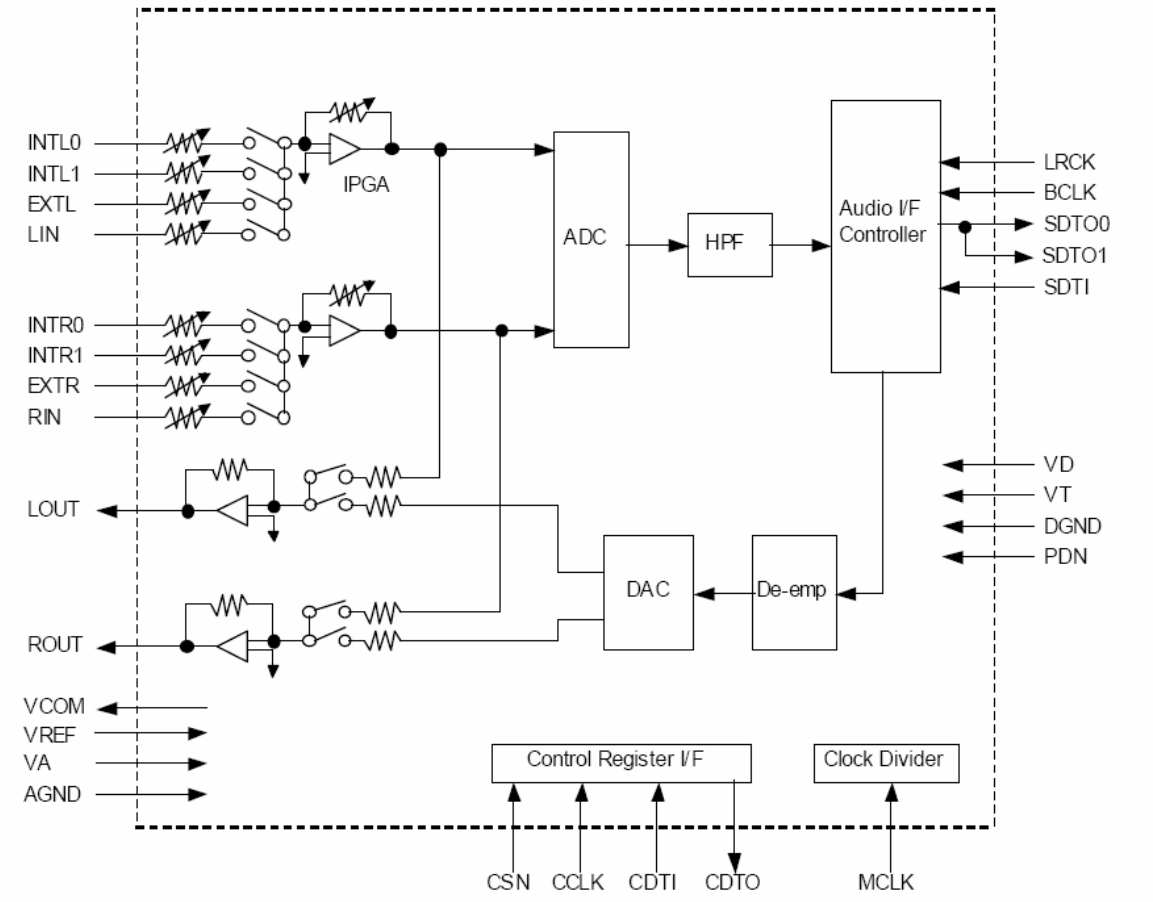
At a high level the system architecture consist of the SPARTAN IIE FPGA connected to the ASIX Ethernet AX88796L controller chip, the AKM AK4565 audio codec chip, and a TOSHIBA TC55V16256J 256Kx16-word SRAM controller chip through the On Chip Peripheral Bus (OPB), as shown below:



2. Audio

2.1 Audio Codec AKM AK4565

A block diagram of the audio codec is shown below. The diagram is taken from the AKM AK4565 data sheet.



2.2 Audio Controller

The audio codec contains a 20 bits per channel (2 channels) ADC, and a 20 bits per channel DAC. The ADC outputs audio data serially, and the DAC requires serial audio data. The codec also requires many different clock signals (of various periods) to maintain correct operation. For the audio controller, we used Cristian's VHDL code from two years ago. This controller provides an interface from the ADC to the microblaze, and from the microblaze to the DAC.

The controller turns the serial data from the ADC into samples which are 16 bits per sample per channel, or 32 bits per sample, at a sampling rate slightly higher than 48 kHz (48000 samples per second). Note that 4 of the 20 bits per sample per channel get ignored, which should reduce distortion (since the full range of the ADC is not used). The samples are stored in a BRAM FIFO buffer (instantiated in the VHDL itself), which is 512 MB, so it stores 128 samples (since each sample is 32 bits = 4 bytes). Similarly, there is another 512 MB BRAM FIFO which stores output samples for the DAC, and the controller takes these samples and converts them into serial audio data for the DAC. When the BRAM FIFO corresponding to the ADC is half full, meaning that it contains 64 samples, an interrupt is generated and sent to the microblaze. When the interrupt gets generated, the microblaze, via an interrupt service routine, is supposed to take those 64 samples out of the ADC's BRAM FIFO, and also put 64 samples into the DAC's BRAM FIFO. By only filling half the ADC FIFO before generating an interrupt, it is ensured that half of the ADC FIFO is always available for the ADC to write to, and half of the DAC FIFO is always available for the microblaze to write to, thus avoiding FIFO overflows. It is assumed that samples are always coming in, so the interrupts get generated on a clock, regardless of whether any audio is actually "connected" to the ADC. The audio controller also generates all the clock signals required by the codec.

The audio codec is configured as a peripheral on the OPB bus. When the interrupt occurs, data is sent from the ADC FIFO to the microblaze via Sin_DBus, with OPB_RNW=1 (since the microblaze is *reading* data from the peripheral), and then data is sent from the microblaze to the DAC FIFO via the OPB_Dbus, with OPB_RNW=0 (since the microblaze is *writing* data to the microblaze).

2.3 Interrupt Handler

When an interrupt gets sent from the audio controller to the microblaze, the normal execution of the program gets "interrupted", and the interrupt service routine gets called. This routine must run really quickly, so only a minimal amount of computation is allowed. Since we wanted to process the audio, we needed a way of getting the audio out of the VHDL's BRAM FIFO and into a large FIFO buffer, which we will refer to as the ADC buffer, which is easily accessible during normal execution. Also, the processed (and received) audio needs to be written to the DAC; it should be taken from another large FIFO buffer, which we will refer to as the DAC buffer. As large buffers are desirable, and space is limited in the BRAM, all our FIFOs were put into SRAM.

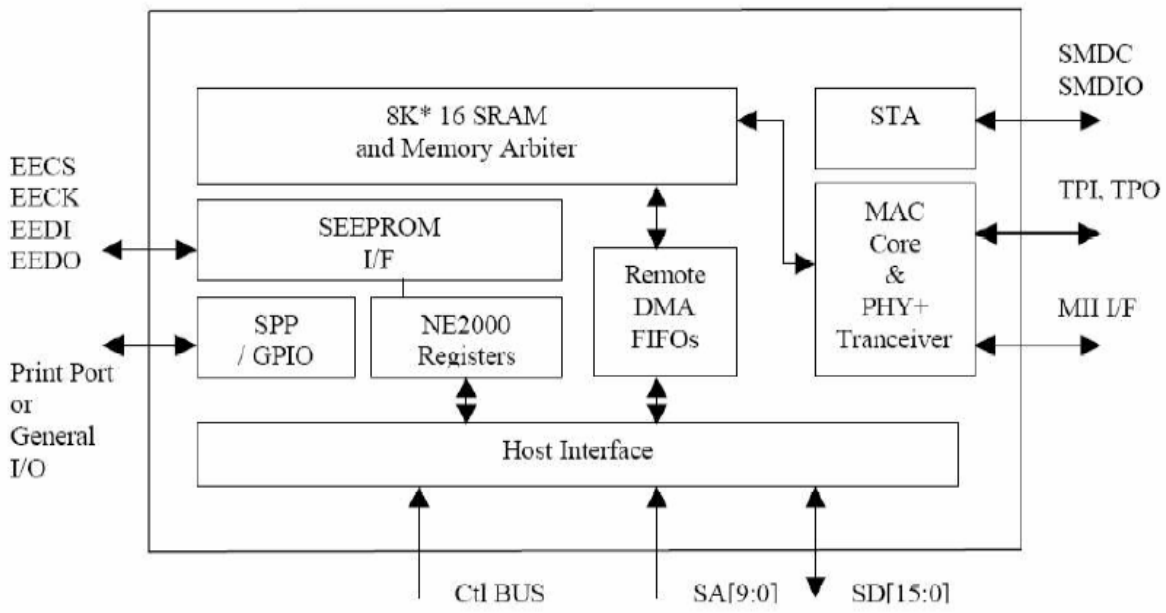
The FIFOs were implemented using read pointers and write pointers. Writing data into a FIFO increments its write pointer, while reading data from a FIFO increments its read pointer. When the pointers reach the FIFO size, they wrap around. When the read pointer equals the write pointer, if the last operation was a read, the FIFO is empty, so no more reads are allowed until the write pointer increments some more, whereas when the last operation was a write, the FIFO has overflowed, causing an error message to be printed to STDOUT during the main program.

We did not require stereo processing for this project, so buffering, sending, receiving and processing all 32 bits per sample from the ADC would have been redundant and inefficient. Instead, in the interrupt service routine, we threw out one of the channels (by doing bitwise AND with 0x0000FFFF), and thus only buffered, processed, sent and received 16 bits per channel. Also, in the interrupt service routine, the 16-bit data from the DAC buffer was copied into both the left and right channel outputs of the DAC, giving 32 bits per sample to send to the DAC.

Ideally, the interrupt service routine should not need to worry about what the main program does with the data put into the ADC buffer, nor should it need to worry about where the data in the DAC buffer came from. The interrupt service routine simply takes data from the ADC, throws out one channel, and moves 16 bits per sample (and 64 samples per interrupt) into the ADC buffer. The routine then takes data (16 bits per sample and 64 samples) from the DAC buffer, copies each 16 bit sample into both the left and right channels, and then sends the data out to the DAC.

3. Ethernet Send/Receive

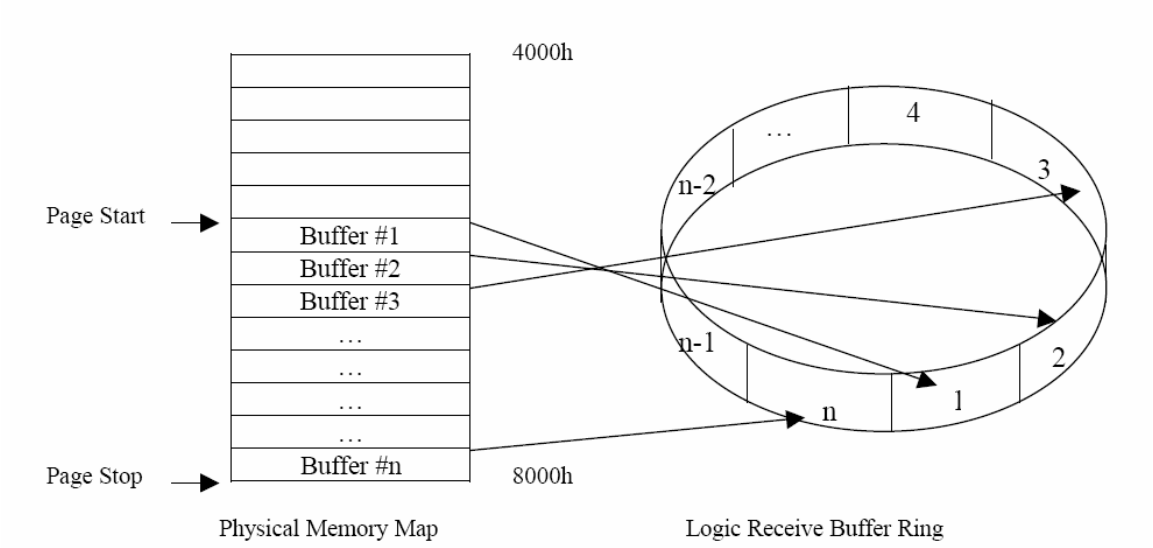
The Ethernet Controller Internals:



Memory Addressing:
NE2000 registers – 0xA00400 – 0xA0041F
SRAM Local Memory – 0xA04400 – 0xA0BFFF

Packet Reception

The Local DMA receive channel uses a Buffer Ring Structure comprised of a series of contiguous fixed length 256 byte (128 word) buffers for storage of received packets. The location of the Receive Buffer Ring is programmed in two registers, a Page Start and a Page Stop Register. Ethernet packets consist of minimum packet size (64 bytes) to maximum packet size (1522 bytes), the 256 byte buffer length provides a good compromise between short packets and longer packets to most efficiently use memory. In addition these buffers provide memory resources for storage of back-to-back packets in loaded networks. The assignment of buffers for storing packets is controlled by Buffer Management Logic in the AX88796. The Buffer Management Logic provides three basic functions: linking receive buffers for long packets, recovery of buffers when a packet is rejected, and recirculation of buffer pages that have been read by the host. At initialization, a portion of the 16k byte (or 8k word) address space is reserved for the receiver buffer ring. Two eight bit registers, the Page Start Address Register (PSTART) and the Page Stop Address Register (PSTOP) define the physical boundaries of where the buffers reside. The AX88796 treats the list of buffers as a logical ring; whenever the DMA address reaches the Page Stop Address, the DMA is reset to the Page Start Address.



INITIALIZATION OF THE BUFFER RING

Two static registers and two working registers control the operation of the Buffer Ring. These are the Page Start Register, Page Stop Register (both described previously), the Current Page Register and the Boundary Pointer Register. The Current Page Register points to the first buffer used to store a packet and is used to restore the DMA for writing status to the Buffer Ring or for restoring the DMA address in the event of a Runt packet, a CRC, or Frame Alignment error. The Boundary Register points to the first packet in the Ring not yet read by the host. If the local DMA address ever reaches the Boundary, reception is aborted. The Boundary Pointer is also used to initialize the Remote DMA for removing a packet and is advanced when a packet is removed. A simple analogy to remember the function of these registers is that the Current Page Register acts as a Write Pointer and the Boundary Pointer acts as a Read Pointer.

BEGINNING OF RECEPTION

When the first packet begins arriving the AX88796 and begins storing the packet at the location pointed to by the Current Page Register. An offset of 4 bytes is reserved in this first buffer to allow room for storing receive status corresponding to this packet.

LINKING RECEIVE BUFFER PAGES

If the length of the packet exhausts the first 256 bytes buffer, the DMA performs a forward link to the next buffer to store the remainder of the packet. For a maximal length packet the buffer logic will link six buffers to store the entire packet. Buffers cannot be skipped when linking, a packet will always be stored in contiguous buffers. Before the next buffer can be linked, the Buffer Management Logic performs two comparisons. The first comparison tests for equality between the DMA address of the next buffer and the contents of the Page Stop Register. If the buffer address equals the Page Stop Register, the buffer management logic will restore the DMA to the first buffer in the Receive Buffer Ring value programmed in the Page Start Address Register. The second comparison test for equality between the DMA address of the next buffer address and the contents of the Boundary Pointer Register. If the two values are equal the reception is aborted. The Boundary Pointer Register can be used to protect against overwriting any area in the receive buffer ring that has not yet been read. When linking buffers, buffer management will never cross this pointer, effectively avoiding any overwrites. If the buffer address does not match either the Boundary Pointer or Page Stop Address, the link to the next buffer is performed.

LINKING BUFFERS

Before the DMA can enter the next contiguous 256 bytes buffer, the address is checked for equality to PSTOP and to the Boundary Pointer. If neither are reached, the DMA is allowed to use the next buffer.

BUFFER RING OVERFLOW

If the Buffer Ring has been filled and the DMA reaches the Boundary Pointer Address, reception of the incoming packet will be aborted by the AX88796. Thus, the packets previously received and still contained in the Ring will not be destroyed. In a heavily loaded network environment the local DMA may be disabled, preventing the AX88796 from buffering packets from the network. To guarantee this will not happen, a software reset must be issued during all Receive Buffer Ring over flows (indicated by the OVW bit in the Interrupt Status Register). The following procedure is required to recover from a Receiver Buffer Ring Overflow. If this routine is not adhered to, the AX88796 may act in an unpredictable manner. It should also be noted that it is not permissible to service an overflow interrupt by continuing to empty packets from the receive buffer without implementing the prescribed overflow routine.

Note: It is necessary to define a variable in the driver, which will be called ``Resend''.

1. Read and store the value of the TXP bit in the AX88796's Command Register.

2. Issue the STOP command to the AX88796. This is accomplished be setting the STP bit in the AX88796's Command Register. Writing 21H to the Command Register will stop the AX88796.

3. Wait for at least 1.5 ms. Since the AX88796 will complete any transmission or reception that is in progress, it is necessary to time out for the maximum possible duration of an Ethernet transmission or reception. By waiting 1.5 ms this is achieved with some guard band added. Previously, it was recommended that the RST bit of the Interrupt Status Register be polled to insure that the pending transmission or reception is completed. This bit is not a reliable indicator and subsequently should be ignored.

4. Clear the AX88796's Remote Byte Count registers (RBCR0 and RBCR1).

5. Read the stored value of the TXP bit from step 1, above. If this value is a 0, set the ``Resend'' variable to a 0 and jump to step 6. If this value is a 1, read the AX88796's Interrupt Status Register. If either the Packet Transmitted bit (PTX) or Transmit Error bit (TXE) is set to a 1, set the ``Resend'' variable to a 0 and jump to step 6. If neither of these bits is set, place a 1 in the ``Resend'' variable and jump to step 6. This step determines if there was a transmission in progress when the stop command was issued in step 2. If there was atransmission in progress, the AX88796's ISR is read to determine whether or not the packet was recognized by the AX88796. If neither the PTX nor TXE bit was set, then the packet will essentially be lost and retransmitted only after a time-out takes place in the upper level software. By determining that the packet was lost at the driver level, a transmit command can be reissued to the AX88796 once the overflow routine is completed (as in step 11). Also, it is possible for the AX88796 to defer indefinitely, when it is stopped on a busy network. Step 5 also alleviates this problem. Step 5 is essential and should not be omitted from the overflow routine, in order for the AX88796 to operate correctly.

6. Place the AX88796 in mode 1 loopback. This can be accomplished by setting bits D2 and D1, of the Transmit Configuration Register to ``0,1''.

7. Issue the START command to the AX88796. This can be accomplished by writing 22H to the Command Register. This is necessary to activate the AX88796's Remote DMA channel.

8. Remove one or more packets from the receive □thernet□.

9. Reset the overwrite warning (OVW, overflow) bit in the Interrupt Status Register.

10. Take the AX88796 out of loopback. This is done by writing the Transmit Configuration Register with the value it contains during normal operation. (Bits D2 and D1 should both be programmed to 0.)

11. If the ``Resend'' variable is set to a 1, reset the ``Resend'' variable and reissue the transmit command. This is done by writing a value of 26H to the Command Register. If the ``Resend'' variable is 0, nothing needs to

END OF PACKET OPERATIONS

At the end of the packet the AX88796 determines whether the received packet is to be accepted or rejected. It either branches to a routine to store the Buffer Header or to another routine that recovers the buffers used to store the packet.

SUCCESSFUL RECEPTION

If the packet is successfully received as shown, the DMA is restored to the first buffer used to store the packet (pointed to by the Current Page Register). The DMA then stores the Receive Status, a Pointer to where the next packet will be stored and the number of received bytes. Note that the remaining bytes in the last buffer are discarded and reception of the next packet begins on the next empty 256 byte buffer boundary. The Current Page Register is then initialized to the next available buffer in the Buffer Ring. (The location of the next buffer had been previously calculated and temporarily stored in an internal scratchpad register.)

BUFFER RECOVERY FOR REJECTED PACKETS

If the packet is a runt packet or contains CRC or Frame Alignment errors, it is rejected. The buffer management logic resets the DMA back to the first buffer page used to store the packet (pointed to by CPR), recovering all buffers that had been used to store the rejected packet. This operation will not be performed if the AX88796 is programmed to accept either runt packets or packets with CRC or Frame Alignment errors. The received CRC is always stored in buffer memory after the last byte of received data for the packet. Error Recovery If the packet is rejected as shown, the DMA is restored by the AX88796 by reprogramming the DMA starting address pointed to by the Current Page Register.

PACKET TRANSMISSION

The Local DMA Read is also used during transmission of a packet. Three registers control the DMA transfer during transmission, a Transmit Page Start Address Register (TPSR) and the Transmit Byte Count Registers (TBCR0,1). When the AX88796 receives a command to transmit the packet pointed to by these registers, buffer memory data will be moved into the FIFO as required during transmission. The AX88796 Controller will generate and append the preamble, synch and CRC fields.

TRANSMIT PACKET ASSEMBLY

The AX88796 requires a contiguous assembled packet with the format shown. The transmit byte count includes the Destination Address, Source Address, Length Field and Data. It does not include preamble and CRC. When transmitting data smaller than 46 bytes, the packet must be padded to a minimum size of 64 bytes. The programmer is responsible for adding and stripping pad bytes. The packets are placed in the buffer RAM by the system. System programs the AX88796 Core's Remote DMA to move the data from the data port to the RAM handshaking with system transfers loading the I/O data port. The data transfer must be 16 bits (1 word) when in 16-bit mode, and 8 bits when the AX88796 Controller is set in 8-bit mode. The data width is selected by setting the WTS bit in the Data Configuration Register and setting the CPU[1:0] pins for ISA, 80186 or MC68K mode.

Destination Address	6 Bytes
Source Address	6 Bytes
Length / Type	2 Bytes
Data (Pad if < 46 Bytes)	46 Bytes Min.

General Transmit Packet Format

TRANSMISSION

Prior to transmission, the TPSR (Transmit Page Start Register) and TBCR0, TBCR1 (Transmit Byte Count Registers) must be initialized. To initiate transmission of the packet the TXP bit in the Command Register is set. The Transmit Status Register (TSR) is cleared and the AX88796 begins to prefetch transmit data from memory. If the Interpacket Gap (IPG) has timed out the AX88796 will begin transmission.

CONDITIONS REQUIRED TO BEGIN TRANSMISSION

In order to transmit a packet, the following three conditions must be met:

1. The Interpacket Gap Timer has timed out
2. At least one byte has entered the FIFO. (This indicates that the burst transfer has been started)
3. If a collision had been detected then before transmission the packet backoff time must have timed out.

COLLISION RECOVERY

During transmission, the Buffer Management logic monitors the transmit circuitry to determine if a collision has occurred. If a collision is detected, the Buffer Management logic will reset the FIFO and restore the Transmit DMA pointers for retransmission of the packet. The COL bit will be set in the TSR and the NCR (Number of Collisions Register) will be incremented. If 15 retransmissions each result in a collision the transmission will be aborted and the ABT bit in the TSR will be set.

TRANSMIT PACKET ASSEMBLY FORMAT

The following diagrams describe the format for how packets must be assembled prior to transmission for different byte ordering schemes. The various formats are selected in the Data Configuration Register and setting the CPU[1:0] pins for ISA, 80186, MC68K or MCS-51 mode.

D15 D8	D7 D0
Destination Address 1	Destination Address 0
Destination Address 3	Destination Address 2
Destination Address 5	Destination Address 4
Source Address 1	Source Address 0
Source Address 3	Source Address 2
Source Address 5	Source Address 4
Type / Length 1	Type / Length 0
Data 1	Data 0
…	…

BOS = 0, WTS = 1 in Data Configuration Register.
This format is used with ISA or 80186 Mode.

D15 D8	D7 D0
Destination Address 0	Destination Address 1
Destination Address 2	Destination Address 3
Destination Address 4	Destination Address 5
Source Address 0	Source Address 1
Source Address 2	Source Address 3
Source Address 4	Source Address 5
Type / Length 0	Type / Length 1
Data 0	Data 1
…	…

BOS = 1, WTS = 1 in Data Configuration Register.
This format is used with MC68K Mode.

D7 D0
Destination Address 0 (DA0)
Destination Address 1 (DA1)
Destination Address 2 (DA2)
Destination Address 3 (DA3)
Destination Address 4 (DA4)
Destination Address 5 (DA5)
Source Address 0 (SA0)
Source Address 1 (SA1)
Source Address 2 (SA2)
Source Address 3 (SA3)
Source Address 4 (SA4)
Source Address 5 (SA5)
Type / Length 0
Type / Length 1
Data 0
Data 1
…

BOS = 0, WTS = 0 in Data Configuration Register.
This format is used with ISA, 80186 or MCS-51 Mode.

Note: All examples above will result in a transmission of a packet in order of DA0 (Destination Address 0), DA1, DA2, DA3 . . . in byte. Bits within each byte will be transmitted least significant bit first.

FILLING PACKET TO TRANSMIT BUFFER (HOST FILL DATA TO MEMORY)

The Remote DMA channel is used to both assemble packets for transmission, and to remove received packets from the Receive Buffer Ring. It may also be used as a general purpose slave DMA channel for moving blocks of data or commands between host memory and local buffer memory. There are two modes of operation, Remote Write and Remote Read Packet.

Two register pairs are used to control the Remote DA, a Remote Start Address (RSAR0, RSAR1) and a Remote Byte Count (RBCR0, RBCR1) register pair. The Start Address Register pair points to the beginning of the block to be moved while the Byte Count Register pair is used to indicate the number of bytes to be transferred. Full handshake logic is provided to move data between local buffer memory (Embedded Memory) and a bidirectional I/O port.

REMOTE WRITE

A Remote Write transfer is used to move a block of data from the host into local buffer memory. The Remote DMA will read data from the I/O port and sequentially write it to local buffer memory beginning at the Remote Start Address. The DMA Address will be incremented and the Byte Counter will be decremented after each transfer. The DMA is terminated when the Remote Byte Count Register reaches a count of zero.

REMOVING PACKETS FROM THE RING (HOST READ DATA FROM MEMORY)

REMOTE READ

A Remote Read transfer is used to move a block of data from local buffer memory to the host. The Remote DMA will sequentially read data from the local buffer memory, beginning at the Remote Start Address, and write data to the I/O port. The DMA Address will be incremented and the Byte Counter will be decremented after each transfer. The DMA is terminated when the Remote Byte Count Register reaches zero.

Packets are removed from the ring using the Remote DMA or an external device. When using the Remote DMA. The Boundary Pointer can also be moved manually by programming the Boundary Register. Care should be taken to keep the Boundary Pointer at least one buffer behind the Current Page Pointer. The following is a suggested method for maintaining the Receive Buffer Ring pointers.

1. At initialization, set up a software variable (next_pkt) to indicate where the next packet will be read. At the beginning of each Remote Read DMA operation, the value of next_pkt will be loaded into RSAR0 and RSAR1.

2. When initializing the AX88796 set:
BNRY = PSTART
CPR = PSTART + 1
Next_pkt = PSTART + 1

3. After a packet is DMAed from the Receive Buffer Ring, the Next Page Pointer (second byte in AX88796
receive packet buffer header) is used to update BNRY and next_pkt.
Next_pkt = Next Page Pointer

BNRY = Next Page Pointer – 1
If BNRY < PSTART then BNRY = PSTOP – 1

Note the size of the Receive Buffer Ring is reduced by one 256-byte buffer; this will not, however, impede the operation of the AX88796. The advantage of this scheme is that it easily differentiates between buffer full and buffer empty: it is full if BNRY = CPR; empty when BNRY = CPR-1.

STORAGE FORMAT FOR RECEIVED PACKETS

The following diagrams describe the format for how received packets are placed into memory by the local DMA channel. These modes are selected in the Data Configuration Register and setting the CPU[1:0] pins for ISA, 80186, MC68K or MCS-51 mode.

D15 D8	D7 D0
Next Packet Pointer	Receive Status
Receive Byte Count 1	Receive Byte Count 0
Destination Address 1	Destination Address 0
Destination Address 3	Destination Address 2
Destination Address 5	Destination Address 4
Source Address 1	Source Address 0
Source Address 3	Source Address 2
Source Address 5	Source Address 4
Type / Length 1	Type / Length 0
Data 1	Data 0
…	…

BOS = 0, WTS = 1 in Data Configuration Register.
This format is used with ISA or 80186 Mode.

D15 D8	D7 D0
Receive Status	Next Packet Pointer
Receive Byte Count 0	Receive Byte Count 1
Destination Address 0	Destination Address 1
Destination Address 2	Destination Address 3
Destination Address 4	Destination Address 5
Source Address 0	Source Address 1
Source Address 2	Source Address 3
Source Address 4	Source Address 5
Type / Length 0	Type / Length 1
Data 0	Data 1
…	…

BOS = 1, WTS = 1 in Data Configuration Register.
This format is used with MC68K Mode.

D7	D0
Receive Status	
Next Packet Pointer	
Receive Byte Count 0	
Receive Byte Count 1	
Destination Address 0	
Destination Address 1	
Destination Address 2	
Destination Address 3	
Destination Address 4	
Destination Address 5	
Source Address 0	
Source Address 1	
Source Address 2	
Source Address 3	
Source Address 4	
Source Address 5	
Type / Length 0	
Type / Length 1	
Data 0	
Data 1	
…	

BOS = 0, WTS = 0 in Data Configuration Register.
This format is used with ISA, 80186 or MCS-51 Mode.

OTHER USEFUL OPERATIONS

MEMORY DIAGNOSTICS

Memory diagnostics can be achieved by Remote Write/Read DMA operations. The following is a suggested step
for memory test and assume the AX88796 has been well □thernet□ed.
1. Issue the STOP command to the AX88796. This is accomplished be setting the STP bit in the AX88796's Command Register. Writing 21H to the Command Register will stop the AX88796.

2. Wait for at least 1.5 ms. Since the AX88796 will complete any reception that is in progress, it is necessary to time out for the maximum possible duration of an Ethernet reception. This action prevents buffer memory from written data through Local DMA Write.

3. Write data pattern to MUT (memory under test) by Remote DMA write operation.

4. Read data pattern from MUT (memory under test) by Remote DMA read operation.

5. Compare the read data pattern with original write data pattern and check if it is equal.

6. Repeat step 3 to step 5 with various data pattern.

LOOPBACK DIAGNOSTICS

1. Issue the STOP command to the AX88796. This is accomplished be setting the STP bit in the AX88796's Command Register. Writing 21H to the Command Register will stop the AX88796.

2. Wait for at least 1.5 ms. Since the AX88796 will complete any reception that is in progress, it is necessary to time out for the maximum possible duration of an Ethernet reception. This action prevents buffer memory from written data through Local DMA Write.

3. Place the AX88796 in mode 1 loopback. (MAC internal loopback) This can be accomplished by setting bits D2 and D1, of the Transmit Configuration Register to ``0,1''.

4. Issue the START command to the AX88796. This can be accomplished by writing 22H to the Command Register. This is necessary to activate the AX88796's Remote DMA channel.

5. Write data that want to transmit to transmit buffer by Remote DMA write operation.

6. Issue the TXP command to the AX88796. This can be accomplished by writing 26H to the Command Register.

7. Read data current receive buffer by Remote DMA read operation.

8. Compare the received data with original transmit data and check if it is equal.

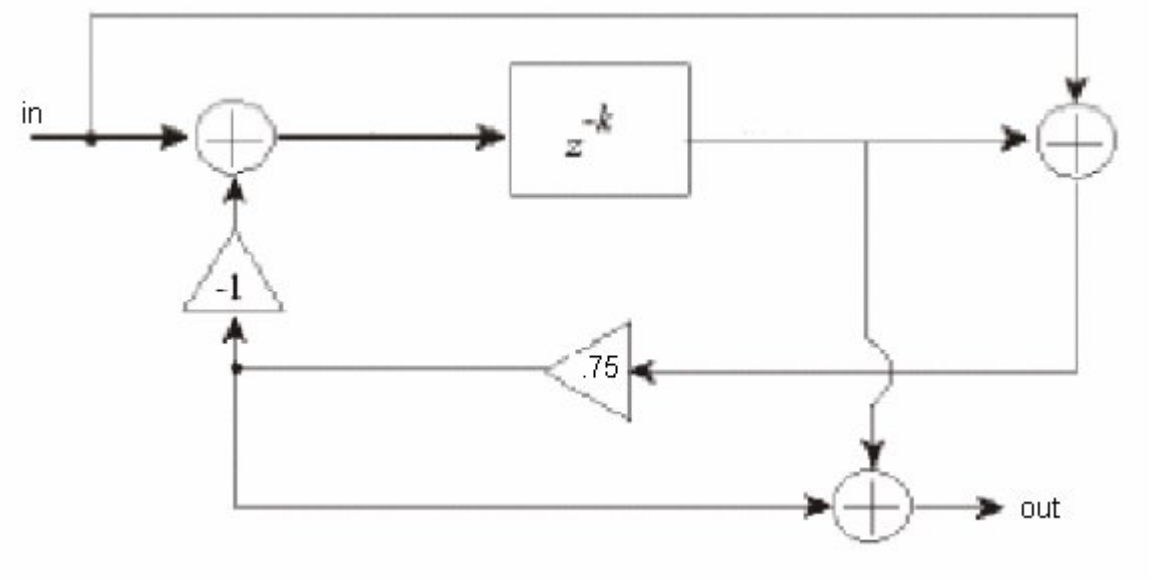
9. Repeat step 5 to step 8 for more packets test

4. Microblaze

4.1 Audio Effects: Reverberation

The main program should take data from the ADC buffer, process it, and send it out to the □thernet controller. The main program should also receive data from the Ethernet controller and place it into the DAC buffer. As of right now, we have successfully implemented the audio effects processor, and we are successfully sending the processed audio to the Ethernet controller. We are still troubleshooting the receiver. This section will only describe our working audio processor. We can already demonstrate that the audio processing works by connecting it directly between the ADC and DAC buffers of a single board (rather then getting the DAC buffer from the Ethernet controller).

In general, digital signal processors (DSPs) consist of adders, gains (which can be positive or negative), and delay blocks. We have implemented a particular DSP: a two stage reverberator. A reverberator generates echoes of an input signal. Our reverberator uses adders, positive and negative gains, and delay blocks. Thus, in the course of building the reverberator, we implemented all the parts needed for any DSP, and we would easily be able to modify our code to get any desired DSP effect. A block diagram for a single stage of the reverberator is shown below:



Our first stage used k=2048 (about 1/20 of a second), and our second stage used k=5376 (about 1/10 of a second). The sizes of the delays may be easily adjusted in the constant declarations; this adjusts the duration of the reverberations. We had a lot of fun experimenting with this. The second stage echoes the echoes generated by the first stage, producing a more natural sounding output.

We used unsigned 16 bit integers to represent the audio data. By looking at the audio data in minicom when no audio was connected, we noted that most of the MSBs were alternately either 0's or F's, implying a standard 2's complement representation. Implementing the adders was therefore trivial (+). To implement the positive gain, let x be the number which should be multiplied by .75. We did not want to do multiplication, since that is an expensive operation on an FPGA. Shifting, however, is cheap. We noted that $.75 = 2^{-1}+2^{-2}$, so to implement the gain, we would like to take x>>1 + x>>2. However, since x is an unsigned integer, we had to do sign extension manually. Thus, if x is negative (if the sign bit of x, or x>>15, is 1), then the rightmost bit of x>>1 and the rightmost two bits of x>>2 should be 1. This was easily implemented as (x>>1 | 0x8000)+(x>>2 | 0xc0000), where | is bitwise OR. Any arbitrary (but constant) gain may be written as a sum of powers of 2, so it may be implemented efficiently by adding shifted versions of x. To implement the multiplication by -1, we simply took the two's complement: (~x)+1, where ~ is bitwise NOT. For each delay block, we used a huge circular SRAM FIFO of size (k+1), initialized to all zeros (thus establishing the initial conditions of the delay block to be the zero state), with pointers to the input and output samples of the delay block. The output sample pointer starts at the beginning of the FIFO (index 0), while the input sample pointer starts at the end (index k). Thus, the two pointers start k samples apart, and are moved in synch; the pointers are therefore separated by k samples at all time-steps. Both pointers wrap around whenever they reach the end of the FIFO (index k+1). The amount of memory required per delay block is (k+1) times 16 bits, or (2k+2) bytes. Thus addition, delay, and positive and negative gains have been implemented efficiently on the FPGA. We combined these parts in the C code (see file main.c) to create the reverberator, but it should be stressed that these parts may be easily combined to create any desired DSP effect.

4.2 Ethernet Communication: IP/Xilnet

The proposed software – hardware interaction is much more complicated (tries to imitate UNIX like semantics for the Xilinx Microblaze Libraries) to be implemented within the constraints of the project.

The following modules outline what was proposed as the guidelines for implementing the TCP / UDP / IP library. However , later in this report we describe how the Xilnix Xilnet v 2.0 was hacked into our code to create a customized and optimized (only ~ 13K of memory footprint) UDP / IP / Ethernet stack , robust , small and efficient protocol stack for the Xilinx Spartan II E platform.

TCP Runtime Library:

The TCP Runtime Library is the core TCP subsystem. The main functionalities of this library are :

1. Connection Request / TCP connect() operation for the SIP based connection initiation and setup.
2. Maintain per connection connected socket descriptor block whose index into the connected socket descriptor index gives the source port of the connection as well as the socket descriptor number (similar to what is implemented in UNIX systems)
3. Maintaining per connection TCP send / receive buffers and TCP timers for connection retransmission timeouts and TCP states like the the TIME_WAIT state. This is also used to maintain and timeout the SIP session on behalf of the user program – i.e. the VoIP soft phone.
4. Implement the basic TCP functions like the TCP connect() , accept() , read() , write and close().
5. Implement a very simple TCP state machine.

UDP Runtime Library:

The UDP runtime library is used to communicate to the connected sockets' descriptor array and the update the connection details such as populate the UDP send buffers and remove the packets from the UDP frame buffer. Both the UDP and the TCP subsystems talk to wrapper functions to encapsulate the packet into UDP/IP or TCP/IP encapsulation routines to encapsulate the contents into datagrams which are transferred to the IP subsystem to be transferred out through the Ethernet subsystem. The following are the functions of the UDP Runtime Library:

1. Encapsulate a RTP voice packet to send and receive to the peer entity / peer FPGA based soft phone.
2. Remove the RTP payload from the received RTP datagram and send it to the user application which is the local FPGA based soft phone.
3. Implements simple UDP functions like the sendto() and recievefrom() to send and receive the UDP packets.

IP Layer (The Network Layer) Runtime Library:

The IP layer library is performs the following functions:

1. Receive UDP / TCP segments and encapsulate them into IP datagrams
2. Lookup the routing table to determine the appropriate network interface to be used to send the data out.
3. Associate appropriate source address to the packet which is being sent out.
4. If there is no entry for the IP to MAC mapping communicate with the ARP module which takes inputs only from the IP layers and returns appropriate MAC address for the frame to be associated with for the appropriate destination IP address .
5. Encapsulate the packet with a MAC layer header and send it to the Ethernet Packet Creation / Reception subsystem which sends it out of the MAC interface.
6. The important functionality of the IP layer is to implement the IP send and recv functions to send the packet to the MAC subsystem which copies it to the On-chip SRAM of the Ethernet controller which is read up by the OPB_Ethernet / On board Ethernet Processor Chip. The same functionality is also implemented for the receiver function which is used to read out the packets from the OPB_Ethernet. Whenever there is an Etherent DMA completion interrupt, the program (Ethernet packet creation/reception subsystem) takes out the frame from the On-chip SRAM buffer and passes it to the higher layers thereby renewing the operations of the Ethernet controller chip and the local DMA operations associated with it.

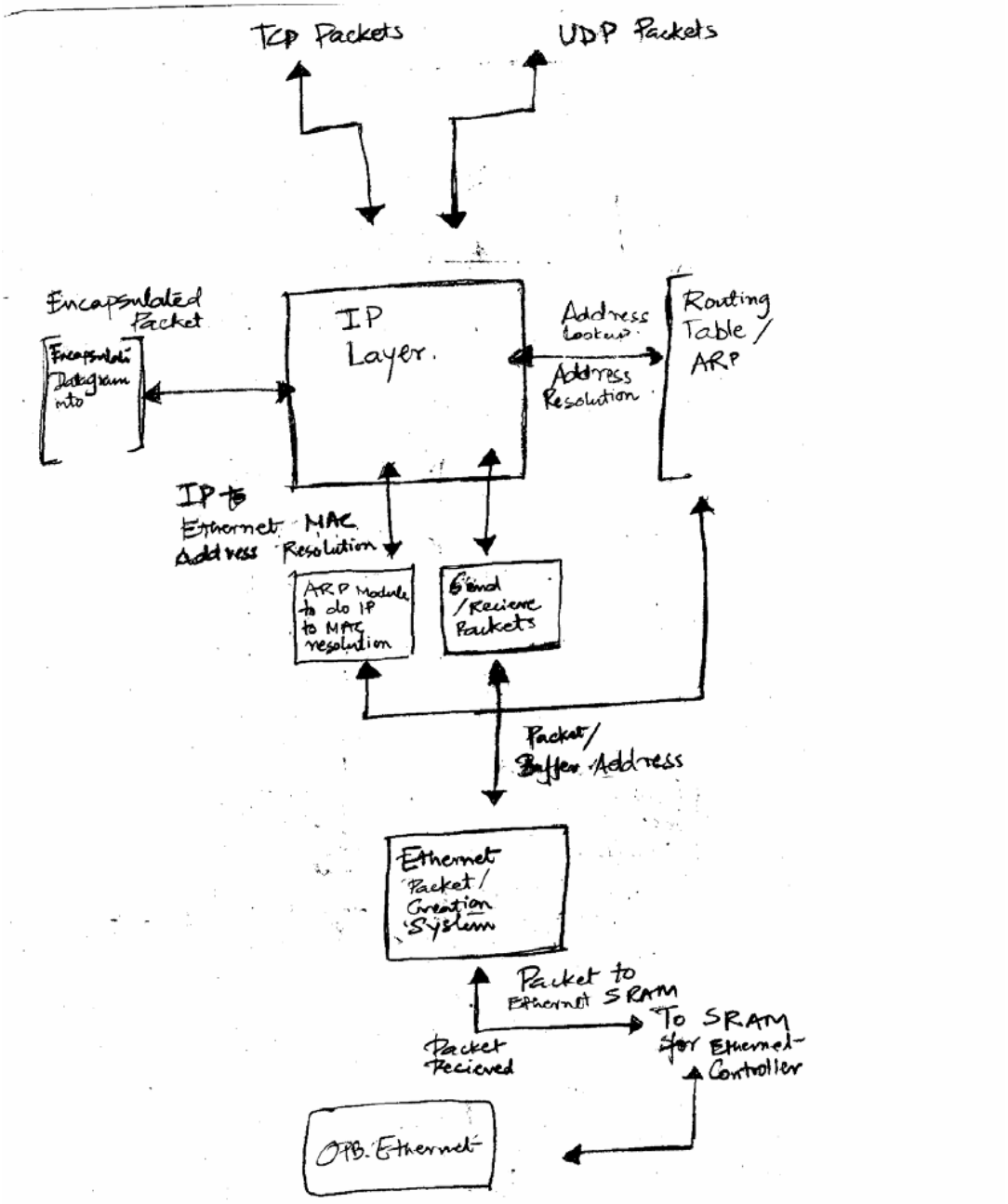
ARP Module:

The APR (Address Resolution Module) is the one that actually maps IP address to MAC layer addresses and appends the MAC layer header to the outgoing datagram with the MAC layer header with appropriate MAC layer destination and source address which is what is understandable to the Ethernet Controller (which independently handles the physical layer signaling , channel coding , carrier sensing , collision detections and binary exponential backoff in case of collision detection (the standard MAC layer mechanism used for collision avoidance in case of shared medium like the Ethernet). It further talks to the Ethernet Packet creation , transmission and reception subroutines to send and receive the ARP requests and replies and to the the IP layer routing table / route – ARP cache to refresh entries for the destination MAC layer addresses for the IP addresses selected.

Send / Receive Packets Module:

This module simply acts as a bridge to send packets from the IP layer / ARP packets and send it to the Ethernet Packet creation module to transfer it to the Ethernet controller via the OPB_Etherent SRAM contoller.

IP Layer Module:

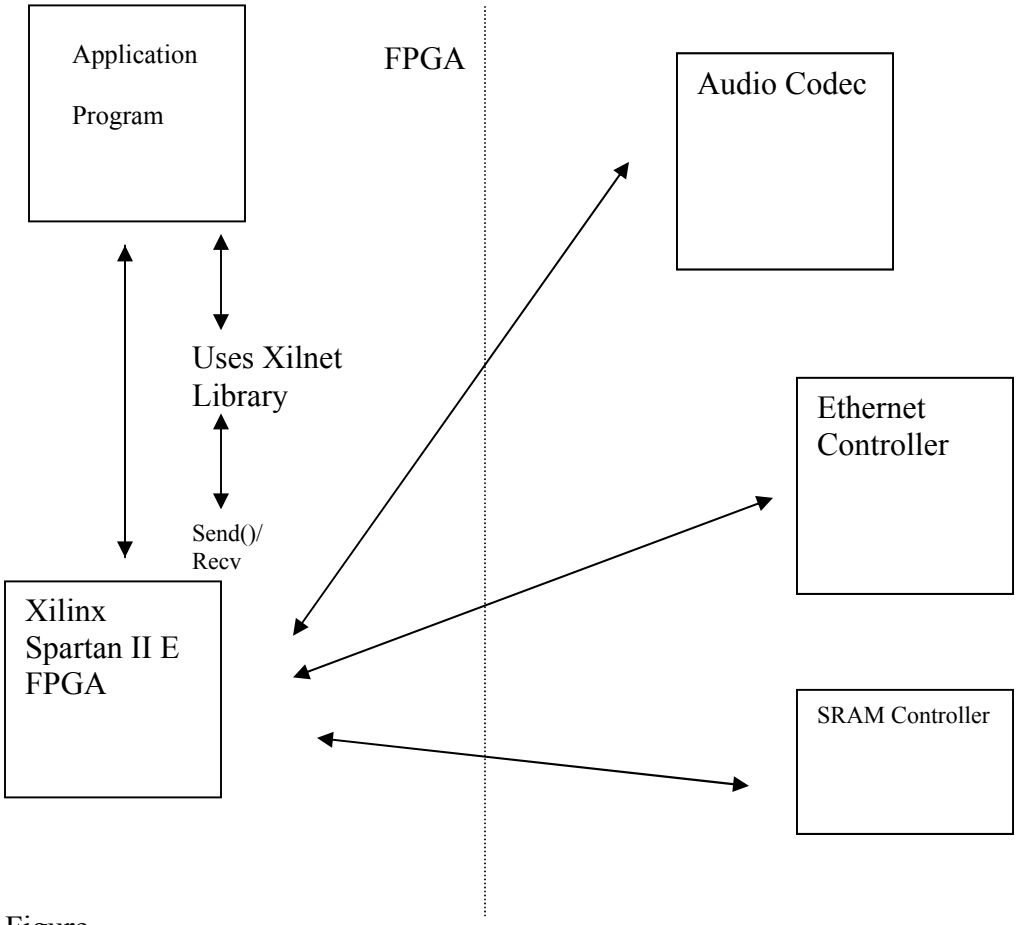


IMPLEMENTING THE UDP / ICMP / IP STACK

We started with the idea of creating a full fledged RTP / SIP TCP / UDP / ICMP / IP stack for the board but soon realized that the enormous amount of complexity involved in getting the correct functioning of the Ethernet Controller with the correct timing for the SRAM and the audio codec. Hence resorted to fixing up the hardware and some existing software to work correctly to send and receive Ethernet frame. Following which we took the Xilinx Xilnet Library to implemented a simple hacked in UDP / ICMP Stack which made use of our functions (viz. `send( )` and `receive( )`) to actually transmit and receive the Ethernet frames. Due to the lack of an interrupt mechanism for the Ethernet – FGPA interface we resorted to simply polling the Interrupt Status Register (ISR) of the Ethernet controller to determine if a packet was ready for reception by our program and then issued a series of DMA read operations to read in the program from the internal SRAM memory to our SRAM. This results in a severe issues with respect to the TCP. Being a single stack of execution , the stack of execution which is used to check the ISR register is the same as the one to call the `xilnet_TCP_accept( )` (Actually Xilnet's version of the TCP accept system called found in most socket APIs across various platforms). Hence , say a packet arrives , the ISR register is set and upon checking that the register we call the `xilnet_eth_recv_frame( )` function which is nothing but a wrapper upon our etherent `receive( )` function and thereby depending upon the type of the packet (weather it is ARP or and IP datagram) calls the appropriate function to process the packet. In case of an ARP packet the `xilnet_arp()` functions are called which further make a call to `xilnet_arp_reply()` . Also , it records in an internal table , the IP address (and the MAC address) of the host which sent the IP – MAC ARP request broadcasted over the entire subnet. This works as long as we are restricted to using MAC ARP packets. The moment we use `xilnet_tcp_accept()` function the function goes to an infinite blocking state waiting for the three way handshake to complete . However nothing can get it out of that blocking as there is only one flow / stack of execution which is able to determine the reception of packet by polling on the ISR. It is this flow is the one which is now blocked in the `xilinx_tcp_accept( )` call.

OVERALL S/W H/W INTERACTION

The following diagram describes the schematically how the Xilnet Library / the Ethernet functions, the FPGA, the SRAM controller and the Audio Codec interconnect.

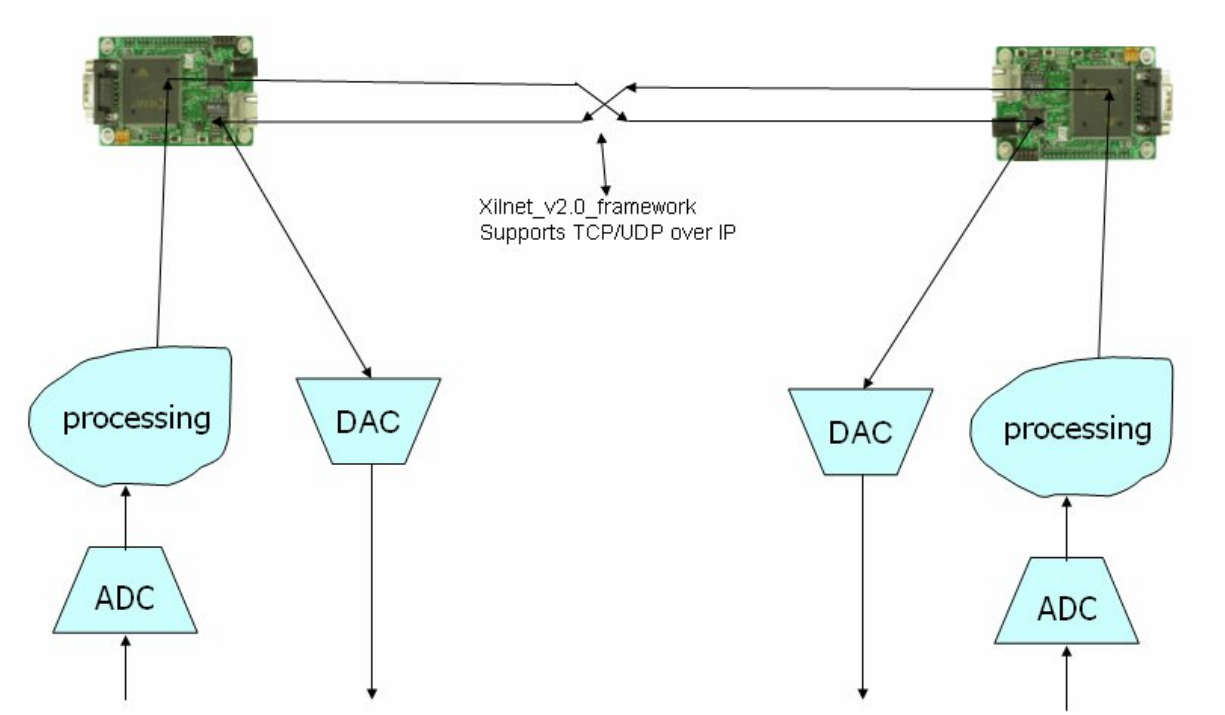


Figure

As evident from the figure … above the application programs calls the Xilnet Library functions which further depends on the underlying Ethernet Send() and Receive( ) functions to initiate the actual Ethernet frame sending and receiving operations

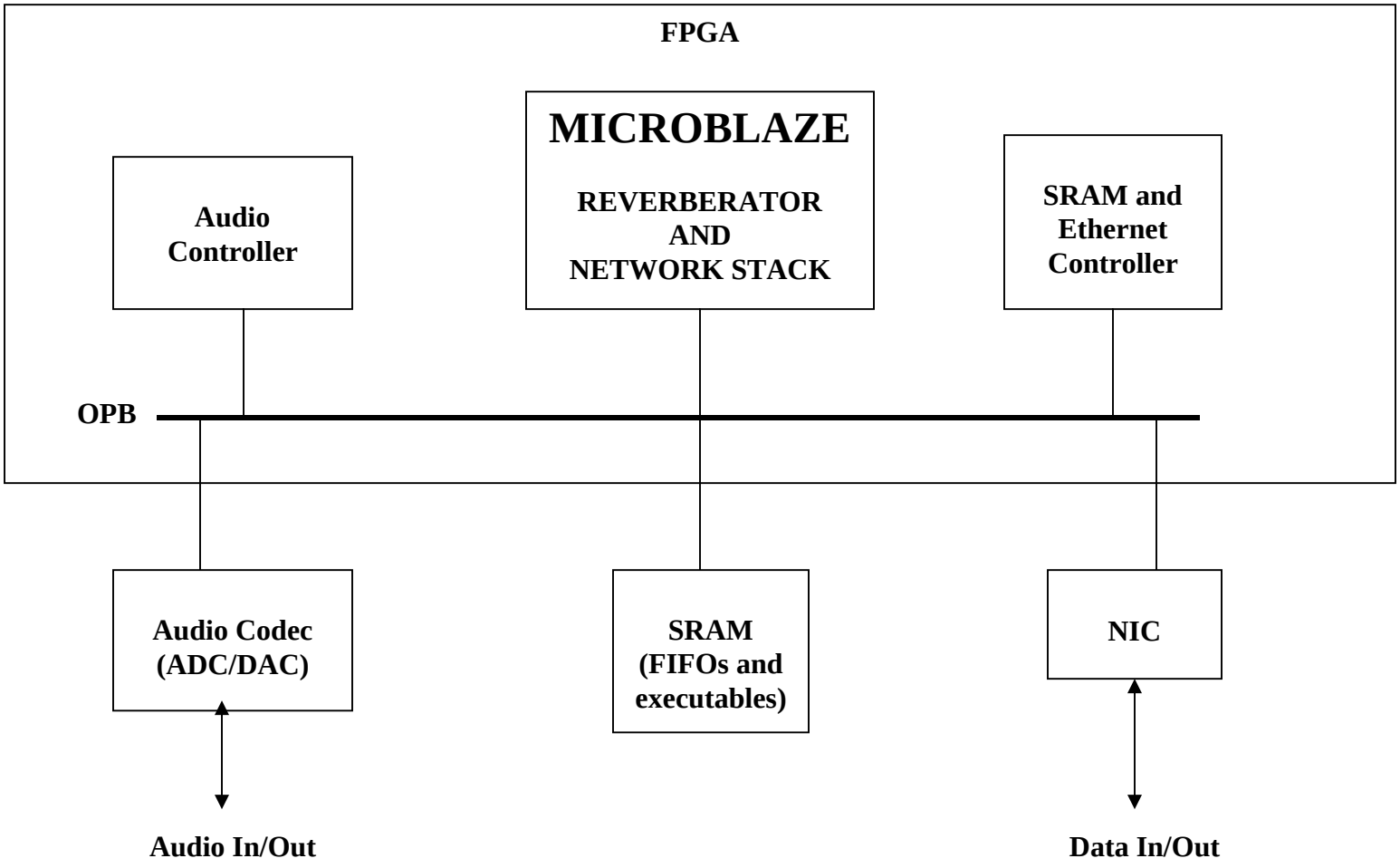
4.3 Complete System Description

On every audio interrupt, the interrupt service routine takes audio data from the ADC and puts this data into an SRAM FIFO, referred to as the ADC buffer. The interrupt routine then takes audio data from another SRAM FIFO, referred to as the DAC buffer, and sends it out to the DAC. The main program is responsible for processing data from the ADC buffer, and for filling the DAC buffer with processed data. This program passes the data from the ADC buffer through our reverberator, and puts the processed data into yet another SRAM FIFO, called the send buffer. The send buffer is 80 words (or 160 bytes), and when it is full, it is sent in its entirety to the □thernet controller. The main program should also receive data from the Ethernet controller in 80 word chunks, and should fill the DAC buffer with this data. The program described above is to be run on two FPGAs simultaneously. These two FPGAs are to be connected via a crossover cable. Data is passed between the two FPGAs using the IP and UDP protocols. The above description is diagrammed below:



We also wrote a linker script (a modified version of Cristian's) to place the most time critical parts of our code, specifically the audio interrupt routine, into the BRAM, while placing the bulk of our code into the SRAM (our entire code would simply not fit into the BRAM). The code in the SRAM was cached, so that it still ran reasonably fast.

SYSTEM ARCHITECTURE



5. Conclusions

5.1 Lessons Learned

For the most part, we all worked on this project as our group. We all learned essentially the same things. Instead of repeating it four times, here is what we have all learned:

Hardware is hard. And debugging is always the hardest part. We spent weeks attempting to program the board to essentially just connect a "wire" between the ADC and the DAC. Then we gave up, and just used Cristian's code. Cristian's code worked perfectly two years ago. Since then, Xilinx has changed everything so that old code does not get properly compiled without extensive modifications to some files. We spent a lot of time watching Cristian hack his two-year-old code to get it to work with the new tools. And compared to what we had to go through to get the Ethernet/SRAM hardware up and running, the audio was a breeze. Unfortunately, the SRAM and the Ethernet use the same data and address lines, and getting both of them to work together is extremely difficult. Again, Cristian's help was critical. We learned that as we are very inexperienced when it comes to hardware, it is extremely important to seek out the help of more knowledgeable people. There is simply too much "specialized" information required. We regularly spent hours trying to debug hardware problems and getting nowhere; Cristian would then fix the same problems in minutes. Debugging hardware is extremely difficult; when the hardware doesn't work, there are hundreds of things that could have gone wrong, with little information as to which specific things did. By starting from Cristian's VHDL files, which we knew worked, we were left with errors only in the compiler's files (MHS, MSS, etc.). Still, debugging was a significant challenge. If we also were unsure about our VHDL, debugging would have been virtually impossible.

We got reasonably proficient at hacking the files; once we had Ethernet/SRAM and audio working in two different projects, we had to combine both of them into one project, which of course involved yet more extensive hacking. The third time around, we managed to do almost all the hacking ourselves. Once we had all the hardware for the audio up and running, getting the microblaze to implement the reverberator was relatively straightforward. We learned that it is crucial to disable interrupts whenever we modify variables in the main program which are also modified in the interrupt service routine. We learned that it is important to know exactly which addresses go where. The SRAM was mapped to 8MB, but it is only 512KB; we spent weeks writing to non-existent memory locations, and wondering why nothing worked.

Debugging the C code is much easier than debugging the hardware. We learned that getting the UART to work should be the first step, so that print statements may be used for debugging. To find errors in the C code, print statements were inserted in various locations. We learned that print statements are so slow that the print statements themselves often make the programs crash. Almost all of our creative ideas and breakthroughs were actually in how we did the debugging!

5.2 Division of Labor

Ashish and Ari focused on getting the audio codec to work, writing the audio interrupt service routine, and programming the audio effects (the reverberator).

George and Sambuddho focused on getting the Ethernet/SRAM controller to work, getting the Ethernet and network stack running on the microblaze .

Combining the audio and the Ethernet was a group effort.

5.3 Advice for future projects

1. Get the UART working first so that you can debug.

2. Do all the hardware first in ONE project. Write VERY simple programs to test the hardware. Only then start writing C.

3. ASK FOR HELP. The professor and TA are there for a reason. Unless you've done this sort of thing before, ask questions, and ask them to help you debug. Many errors are of the type that inexperienced people (like us) would have no idea how to find.

5.4 Acknowledgements

We used code from a lot of sources, including:

- Jaycam
- Last year's VoIP (which didn't work)
- Cristian's audio controller, and SRAM/ethernet controller
- Xilinx code fragments (specifically Xilnet)

Of course, we modified all of the above extensively.

We would also like to acknowledge Professor Edwards and TA Cristian Soviani for all their help with our project. We would have gotten nowhere without their help. Thanks.

6. APPENDIX

6.1 List of C source files

-main.c
-audio.c
-uart_handler.c
-etherFunc.c
-etherSend.c
-etherReceive.c

6.2 List of HDL source files

-opb_xsb300.vhd [SRAM Controller]
- opb_xsb300_ak4565.vhd [Audio Controller]
-bram_elaborate.vhd [BRAM Controller]
-clkgen.v [Clock Generators]

6.3 Makefile and linkscript

-Makefile
-mylinkscript

6.4 UCF, MPD and MHS files

-system.ucf
- opb_xsb300_ak4565_v2_1_0.mpd [Audio Controller]
- opb_xsb300_v2_1_0.mpd [SRAM Controller]
-system.mhs

The above files are in the accompanying tar, so the actual file contents are not included in this document.