Internet Radio

Michael Salib and Jennifer Selby

December 14, 2001

Abstract

The Internet radio system streams live audio feeds over the Internet using a 10BaseT ethernet network. It consists of two parts, a transmitter and a receiver. The transmitter takes an analog audio input, processes it and puts an addressed data packet on the ethernet line. The receiver buffers data addressed to it by the transmitter, and plays the resulting audio data. Although we were unable to deliver a functional unit, we completed a design and learned a great deal.

Contents

List of Figures

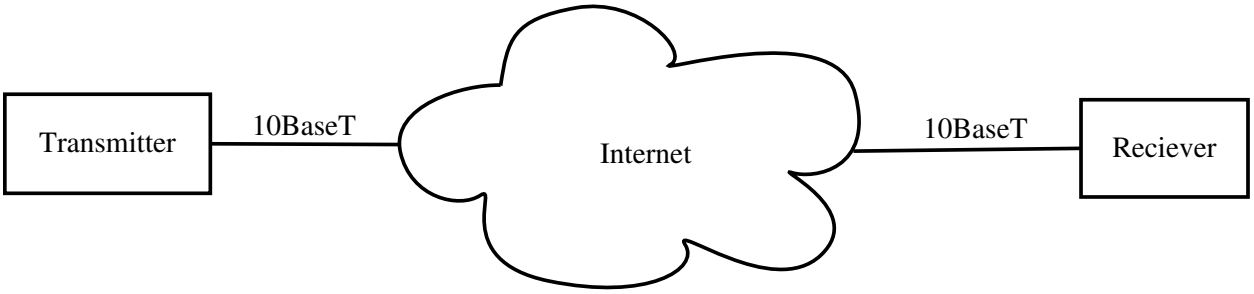

Figure 1: System Block Diagram

1 Overview

Many homes and offices are now equipped with local area networks. Although many people maintain extensive collections of digitized music on their personal computers, they're often unable to listen to that music when not seated at their computer. Others, perhaps, have no digital music on the computer at which they are seated, but know of a radio in the next room. We propose a system that would allow individuals to use their local area networks as a broadcast medium for network radio. That would enable them to enjoy their music anywhere they have network access without the need for a dedicated computer. The entire device could be used for this purpose, or either the receiver or transmitter could be omitted and replaced with a computer, as need demanded.

Our project consists of two components that function independently: a transmitter and a receiver (see Figure 1). The transmitter provides an audio input jack where a microphone or line input can be placed. Using an analog to digital converter, it samples the analog data on the line and buffers the resulting digital data before transmitting it to a specified host over an ethernet interface. The receiver listens for packets addressed to it on the ethernet. Upon finding one, it buffers the incoming digitized audio data before playing it back using a digital to analog converter.

Either component can be replaced with a computer running simple software. In order to build, debug, and test both the transmitter and receiver in parallel, we can use this software to simulate a fully functional version of the other component.

The core of this system is the 10BaseT ethernet transceiver. Ordinarily, the standard ethernet data rate of 10 Mbps would prevent us from building this project since we are unable to build systems that run at 10 MHz. However, these chips provide an external interface that delivers 4 bits of network data at a time, so that our systems can interface with them while running at a clock rate of only 2.5 MHz.

2 Description

2.1 Interoperability

Although the transmitter and receiver are well isolated components that can be designed and built independently, they must agree on certain conventions in order to interoperate

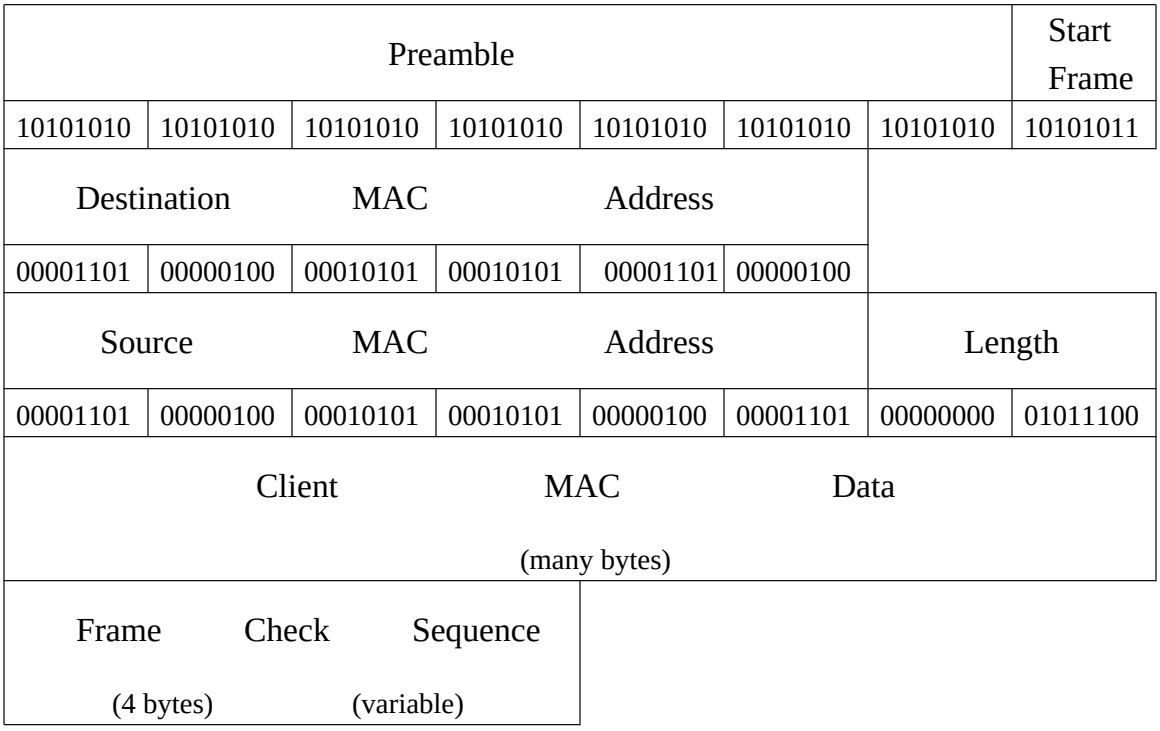

Figure 2: Ethernet Frame

successfully. In particular, they must share a common audio data format, network protocol, and addressing scheme.

2.1.1 Audio Format

We have elected to use a single channel of 8-bit pulse code modulated data sampled at 8,192 Hz as our audio data format. This means that an analog signal is sampled 8,192 times every second to generate a one byte digital sample. This should yield audio quality comparable to that of standard telephone lines. If we had analog to digital converters and digital to analog converters that supported higher precision sampling at 12 or 16 bits per sample, we could significantly improve the audio quality without increasing the data rate by using A-law or u-law compression.

2.1.2 Network Format

Our wire protocol consists of using UDP datagrams in an ethernet frame over 10BaseT ethernet. Doing so ensures that data can be sent and received by a computer using standard Internet protocols. The use of a datagram protocol simplifies the implementation dramatically, although that simplicity comes at a cost. Datagram service is strictly best effort service with no guarantees. This means that packets may arrive out of sequence, after arbitrary delays, or not at all. Local area networks usually have topologies that are simple enough and utilizations low enough to ensure that these problems won't adversely affect our project.

Version	Header Length	TOS/DS,ECN		Total		Length	
0100	0101	0000	0000	1100	0101	0000	0000
Identification				MBZ	Frag	Offset	
1101	0000	1101	0000	0	1	00 0000	0000	0000
Time to Live		Protocol		Header		Checksum	
1111	1111	0001	0001	1011	1011	1001	0011
Source				Address			
0011	0011	0000	0000	0000	0000	1010	0000
Destination				Address			
0010	0011	0000	0000	0000	0000	1010	0000

Figure 3: Internet Protocol Header

An ethernet frame is laid out as shown in figure 2.1.2. The ones and zeros represent the actual data that was being sent in our packets, broken into bytes. Within each byte, the highest order bit is on the left, but the bytes themselves are running with the low-order bytes on the left. In accordance with Internet Protocol 802.3, the bits are sent lowest-order first. The Preamble is a sequence used for synchronization of the sender and receiver. The start of actual data is signaled by the start frame delimiter, which has two high bits next to each other at the end. These bits are the signal for the receiving entity to begin collecting the data. The first piece of data is the destination MAC address. We chose to address and identify the packets by network address rather than MAC address, so the values in this section did not particularly matter for the project and are just a random (static) collection of numbers. The same is true of the Source MAC Address, which directly follows the Destination MAC Address. The length field contains the length of data in the MAC client data field, in bytes. We are sending 64 bytes of data in each packet, the IP headers add another 20, and the UDP header information is 8 bytes long, bringing the total to 92 bytes of data. The client data is the data that is being transmitted in the frame, which is explained in more detail below. All of the data is followed by the frame check sequence. This consists of a cyclical redundancy check done on all fields starting with the destination MAC address and ending with the client data. The method for the check is described in section **??**.

The first thing in the client data is the Internet Protocol (IP) Header, shown in figure 2.1.2. Unlike figure 2.1.2, this diagram breaks the data into groups of four (mostly). Again, in each group of four, the highest order bit is first, but the groups themselves run low to high order. The IP header format is defined in document RFC-791. The first field is the version

Source Port		Destination Port	
00001000	01010010	00001000	01010010
Length		Checksum (Optional)	
01001000	00000000	00000000	00000000
Data many bytes			

Figure 4: User Datagram Protocol Header

number of the internet protocol, the current one being four. The header length specifies how many words (groups of four bytes) are in the IP header. If no options are used, this number is always five. The next field is rarely used, and has been redefined in RFC-2481, section 19, as well as RFC-1122 and RFC-1349. Essentially, a string of zeros signifies delivering the packet normally, so that seemed the obvious choice of value for this field. The total length field refers to the length in bytes of the IP header and any data after it (not including the framecheck). This is the same number as the length field in the IP frame, 92. The identification field is used for purposes of fragmenting packets, which we would not be doing, since our packets are so small, and because we assume a dedicated network connection. The identification number could then be static. After a bit defined to always be zero, there are another two fields concerned with fragmenting. The first one is one bit. A high value in this field signifies no fragmenting of packets. The remainder of that byte and all of the next determine the offset of that packet from the beginning of the entire collection of data packets, so this field is always zero in our packets. The time to live field gives the amount of time that can elapse before the packet should be discarded if its data has not already been read. Again, this is not really a field our device uses, so it is simply the maximum value. IP expects to be used always in conjunction with another protocol, and the next field specifies which one that is. Seventeen denotes the User Datagram Protocol (UDP), as defined in RFC-790. The header checksum is a sixteen-bit one's complement addition of all groups of two bytes, starting with the IP header and ending with the end of the data. The source address and destination address fields specify the network addresses of the receiver and sender, respectively. Address mappings are defined in RFC-796.

Following the IP header is the UDP header, as shown in figure 2.1.2. This figure is done in the same manner as that of the ethernet frame diagram. The format for the UDP header is defined in RFC 768. The source and destination ports allow the user to specify which ports on the computer should be involved in the transfer of the data. The length field is the number of bytes in the datagram, including the headers, 72 for our packets. The last field is an optional checksum, which is meant to include the data. This would mean calculating the checksum for the data before sending, which could be done easily by buffering the data. However, the ethernet frame already provides a data check, so we decided that a second one was not necessary.

2.1.3 Network Addressing

There are two types of network addresses the system needs to deal with: an ethernet address and an Internet Protocol (IP) address. The ethernet address is a physical address associated with a particular network interface card. These addresses are assigned in the factory and are intended to be static throughout the lifetime of the device. Because these address are assigned statically, they contain no routing information, but they're also guaranteed to be unique. Consequently, the ethernet address is sufficient for directing frames on a local broadcast network, but is not suitable for routing packets globally.

Global routing is dealt with at the next layer of the network protocol stack. The Internet Protocol specifies a logical address that is globally routable. While ethernet addresses are tied to one and only one piece of hardware, IP addresses are associated with a host interface. This means that many IP addresses may point to one host or many hosts might share the same IP address.

We decided to use two randomly chosen 48-bit numbers for the transmitter and receiver's ethernet addresses. For IP addresses, we selected a pair of random addresses from the private network address classes. These address blocks are reserved for testing and private networks. As a result, routers refuse to carry packets from these addresses over the public internet. This decision ensures that should we generate malformed packets, any damage that results will be confined to the local network.

2.2 Transmitter (Jennifer Selby)

The transmitter is responsible for taking an analog audio input, converting it into digital information, and packaging that information in a User Datagram Protocol packet within an ethernet frame to be sent over twisted pair ethernet cable. The basic organization of the transmitter can be seen in figure 5. The control logic is responsible for enabling and controlling almost every component directly. The main exception to this is the audio sample clock, which, in effect, enables the controller. The RAM is the main storage for the device. All of the audio samples are stored and accessed in the RAM. The Prom is used only for the static information that is contained in the ethernet frame and protocol headers. The MAC is responsible for breaking the bytes of data into four-bit pieces for the transceiver. Those four bits are passed onto the CRC. The CRC performs a cyclical redundancy algorithm with those four bits, and then transparently passes the data onto the transceiver one clock cycle later. When the data in that packet ends, the CRC passes on the final checksum to the transceiver, just as it did with the other data. The counter is responsible for addressing the RAM and Prom as appropriate for the audio sampling and reading.

2.2.1 Ethernet Transceiver Chip

‘

The ethernet transceiver, when acting as a transmitter takes in four pins of input, an enable pin, and a clock. it outputs a clock at one quarter the speed of its input clock, since it will be transmitting each of the four data bits one by one. It will output these over a

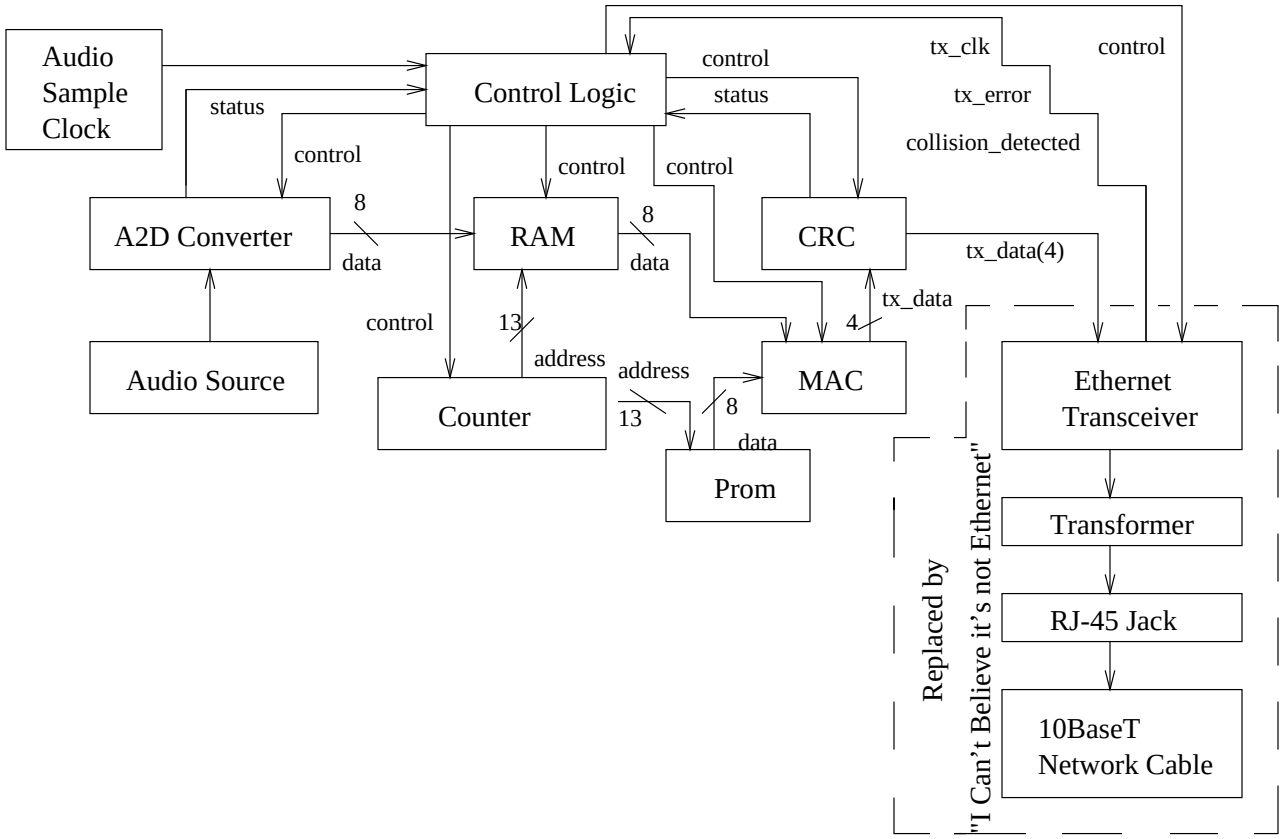

Figure 5: Transmitter Block Diagram

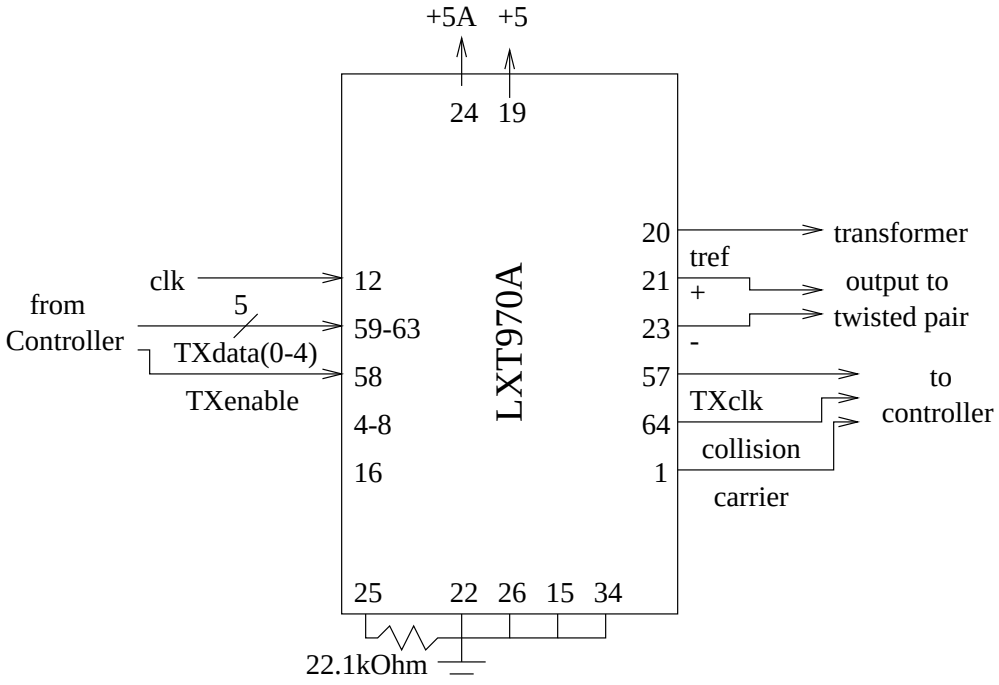

Figure 6: Wiring for the Ethernet Transceiver

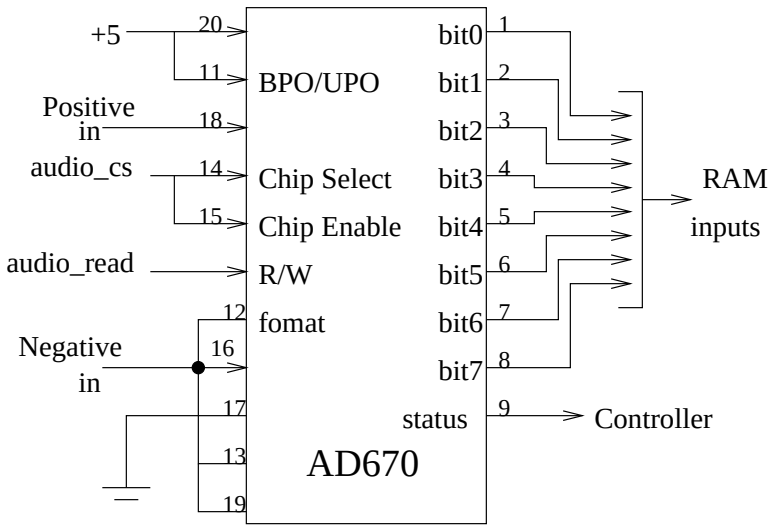

Figure 7: Wiring for the Analog to Digital Converter

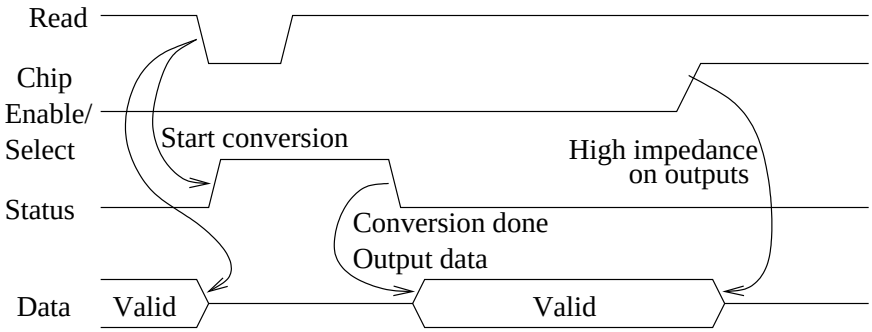

Figure 8: Timing Diagram for operation of AD670 Analog to Digital Converter

twisted pair. In addition, it will also send signals alerting of errors, collisions, or any traffic on the line.

2.2.2 Analog to Digital Converter

The analog to digital converter has a fairly simple operation. It takes as input two wires, one of which is ground. It converts the difference between these two wires into an 8-bit, or one byte piece of data. When there are signals ready for the AD670 chip, pulling chip enable and read low will start a conversion. When the conversion is finished, the converter asserts a logical high on its status pin. When read is brought back high but chip enable is still low, the converter data can be read from the 8 outputs on the chip. If the chip enable or select pins are low, then the converter will simply place a high impedance on the output pins. This timing is shown in figure 8

The signals which control these conversions (chip enable, read) come from the control unit. The status signal from the ad670 also is connected to a pin on the cpld. The data lines themselves, however, are simply connected to the input pins of the RAM. This wiring

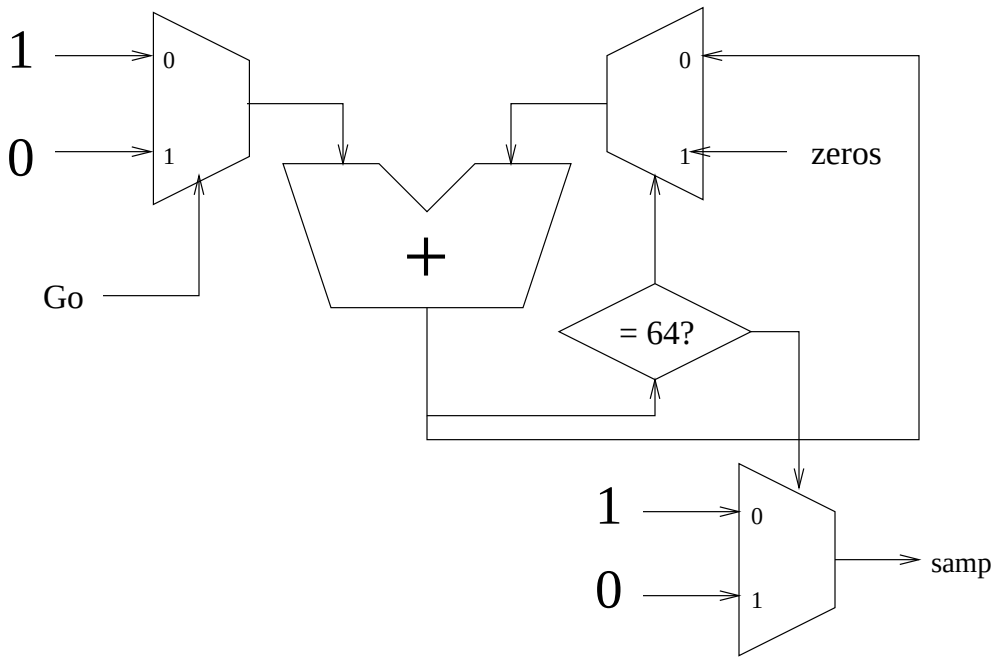

Figure 9: Conceptual Diagram of the Audio Timer Unit

is shown in figure 18.

2.2.3 Audio Timing Unit

The audio timer is a fairly simple device, programmed onto a 22V10 PAL. The objective was to get a signal that would be high 8,192 times a second. What this device does is produce a high signal every 305 clock cycles. The clock for the system is a 2.5 MHz clock, and 2,500,000 divided by 8,192 is 305.176, so this is close to the actual desired performance.

The audio timer is enabled by a go signal, which is attached to a switch – this is the switch that turns the entire device on, essentially. When go is asserted, the audio timer increments an internal counter every clock cycle. On the 305th clock cycle, it rolls back to zero, and asserts the output signal for that clock cycle.

2.2.4 Counter Unit

The counter subdevice is the only component of the system which directly addresses the RAM or PROM. Basically, the counter has three modes in which it operates: addressing the RAM for writing the audio samples, addressing the RAM for reading audio samples, or addressing the PROM. The counter is implemented in VHDL, programmed onto a 374I CPLD.

The counter will address the RAM for writing upon receiving a high aud_count signal. Other subsystems are dependent on aud_count, and it therefore will be high for more than one clock cycle each time the controller asserts it. The audio address should increase only once for each assertion of the singal, however. This problem is solved by having a signal that follows one clock behind aud_count. When aud_count is high, but this signal is low, that

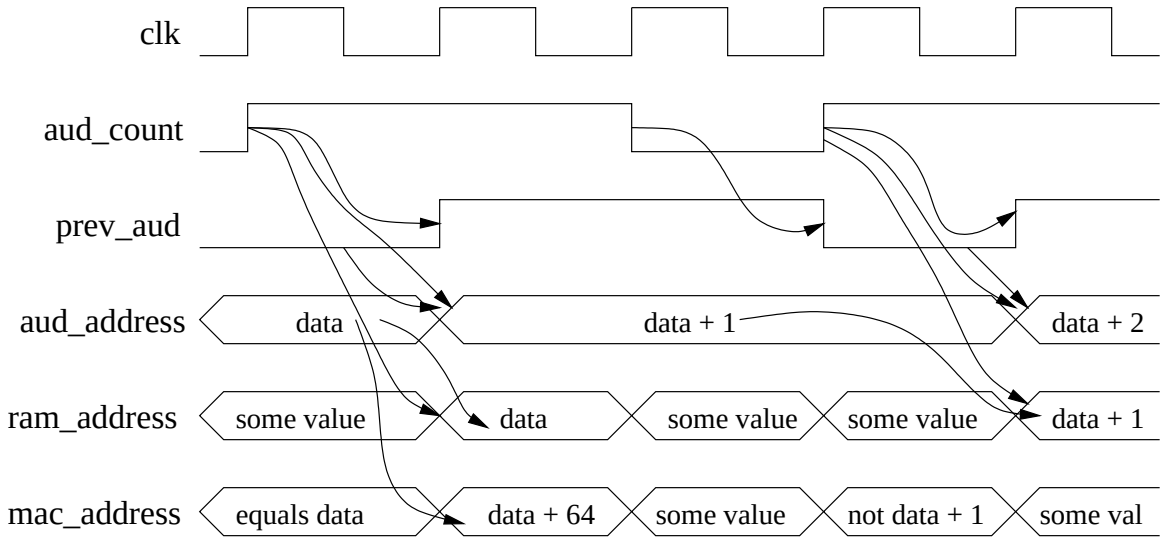

Figure 10: Timing Diagram when Counter is handling audio addressing

means that aud_count has just hit a rising edge. The increment in the audio signal is only triggered by this rising edge, so it only changes once per time aud_count is asserted.

However, the address that is changing is an internal signal, so as not to interfere with any other addressing. During this audio mode, ram_address, an external signal, follows the value of the audio address. The external ram_address is always one increment behind the audio address signal. he other possible pitfall when incrementing the audio address is over running the reading address. This is not likely to happen, but should the audio samples fill up more quickly then they can be sent, the system will simply drop some of the packets in an effort to catch up. In other words, if the new audio address is equal to the current mac_address (meaning that the audio address just moved up to be equal to the mac address), then the signal which contains the starting address for a packet will be incremented by 64, the size on one packet. The only time this will not happen is if the device has just been reset – i.e. the two addresses are equal because both were just set to zero.

The counter will address the prom when mac_count is high while ram_prom is low. When counting the prom, an interruption in the sequence of addresses means that something has gone wrong in the system. The device assumes that all of the addressing of the prom will be done in consecutive clock transitions. Therefore, in this mode, unlike the other two, there is no internal signal holding an address. Instead, when the prom addressing starts (noted by counting_mac being low), the external ram address is set to the starting address in the prom. After that, it is incremented every other clock cycle. The reason that the transition happen every other clock cycle rather than on each one is because the transceiver only takes 4 bits at a time. Therefore, the mac looks at the 8 bits of data coming out of the RAM, transmits the least significant bits in the first clock tick, and the higher order bits in the second clock tice. This was unecessary in audio mode because the conttoller will assert aud_count for several clock cycles, and the counter will keep the audio address on the line. Mac_count, on the other hand, instructs the devices it controls to continue incrementing with out a transition in the signal.

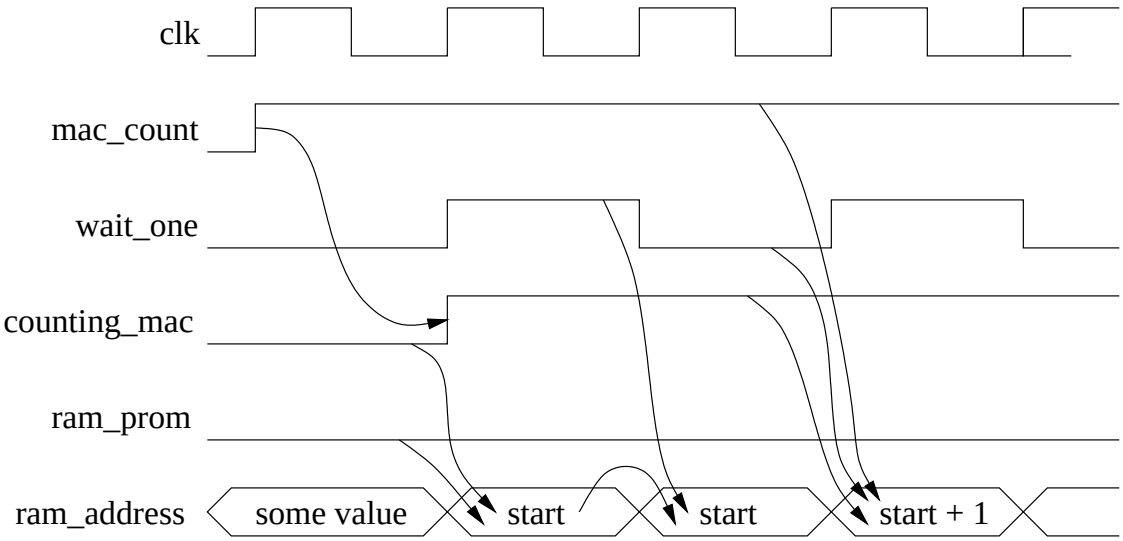

Figure 11: Timing Diagram when Counter is accessing the prom

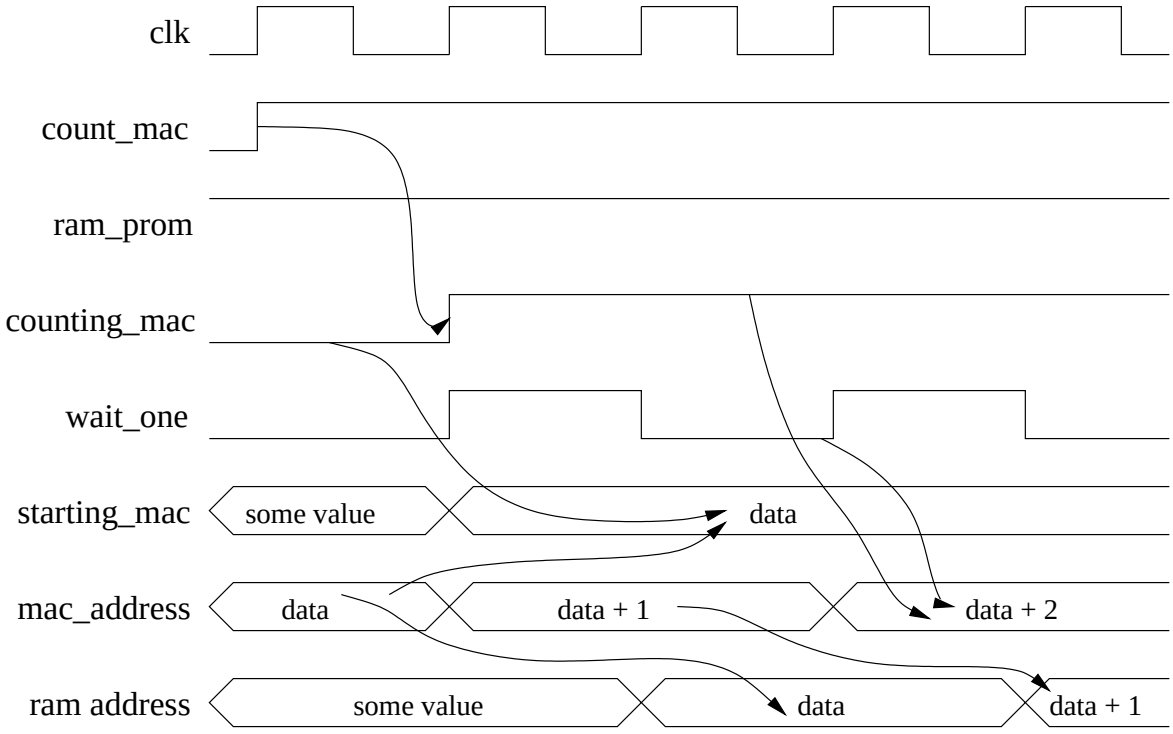

Figure 12: Timing Diagram of the Counter when addressing the RAM for the MAC

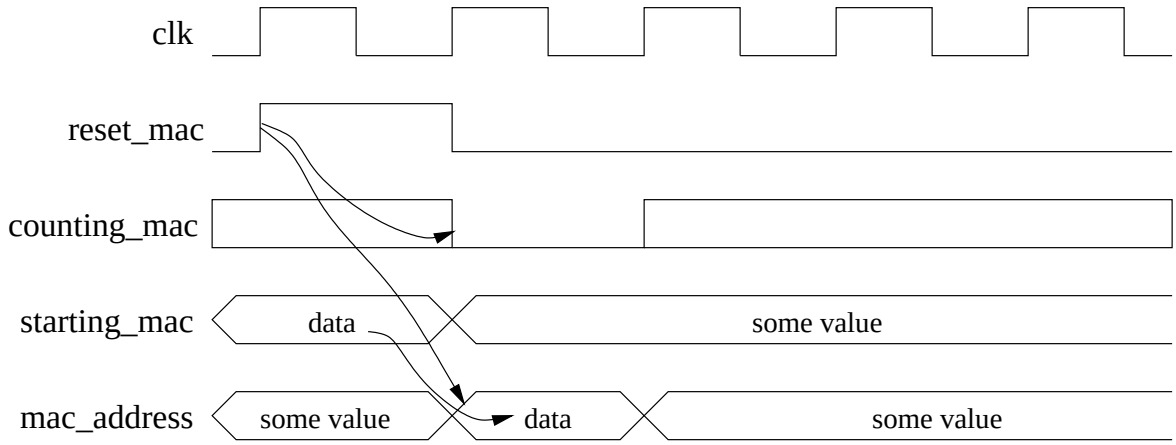

Figure 13: Timing Diagram for the Counter after the MAC is rest

When addressing the RAM for the MAC, the operation of the counter is a mix of the other two modes. Like the prom addressing, the mac_address must be incremented steadily if the count_mac signal remains high. Like the audio addressing, there is an internal signal which retains the current address. However, there is also another internal signal. This signal keeps track of where the first signal of the current packet was stored in the RAM. In this way, if the packet is interrupted for whatever reason, then the packet can be restarted where it left off.

When the packet is interrupted, because of a collision, or because it did not finish before the next audio sample was to be taken, then the controller asserts reset_mac. In this case, the address of the RAM for the MAC must return to that of the first data sample in the packet that just got interrupted. This signal will not affect any other values, other than counting_mac.

The reset signal, wired to a switch external to the system, is much more dramatic than reset_mac. This signal causes all internal addresses to return to zero.

2.2.5 Cyclical Redundancy Check

The Cyclical Redundancy Check Unit computes the CRC function for all the data which is getting passed over the ethernet wire. The controller instructs the CRC to start. In this mode of operation, the CRC both passes data through onto the transceiver and computes the checksum. The algorithm for computing this number is as follows: for each bit of data, shift all bits of the checksum to the left. Then exclusive or the data now in bits 26, 23, 22, 16, 12, 11, 10, 8, 7,5,4,2,and 1 with the data that was in the highest order position. Last, exclusive xor the bit of data with the data that was in the highest order position. All four bits can be done at once, by looking at where a bit ends up relative to its original location in the checksum. This unit was implemented in VHDL, using a 374I CPLD.

When the controller determines that the packet is done, then it sends the crc_send signal to the crc unit. At this time, the CRC stops calculating the checksum and instead starts sending it over as the last part of the ethernet frame. Any noise or signals on the data lines from the MAC will be ignored in favor of sending the CRC data.

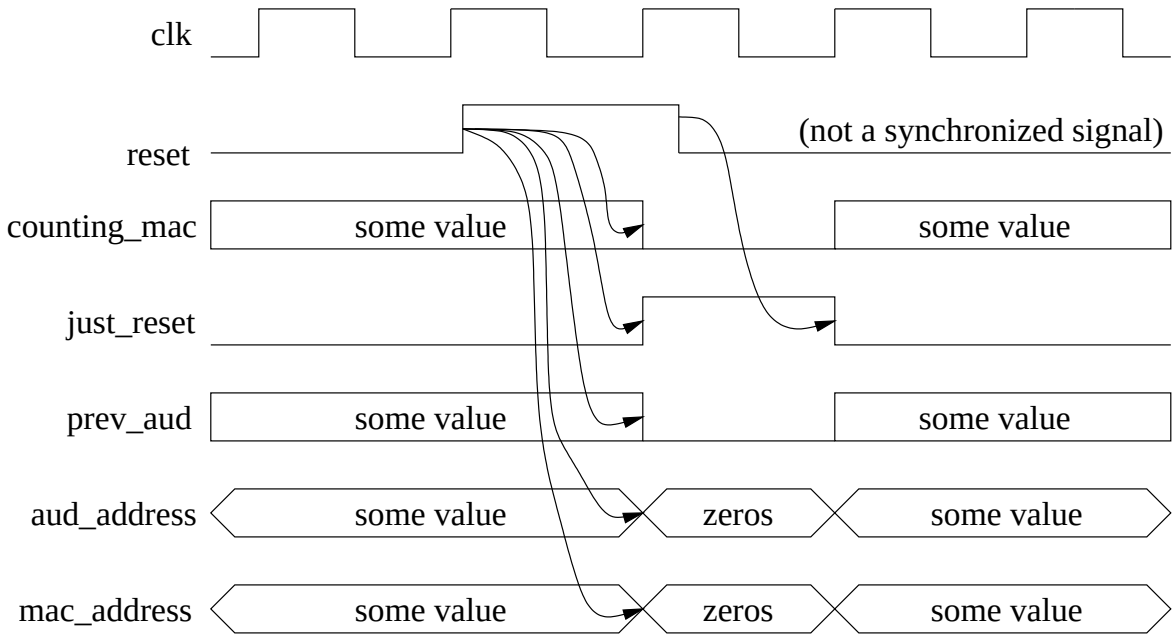

Figure 14: Timing Diagram for the Counter after a reset

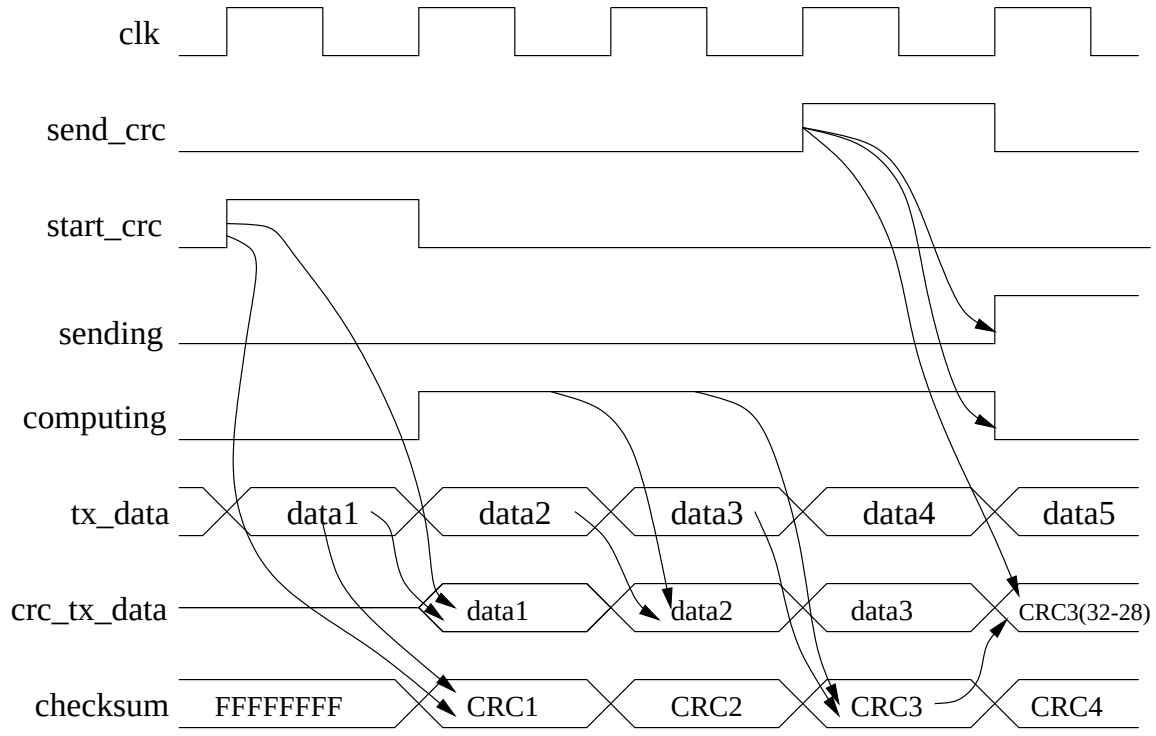

Figure 15: Timing Diagram for the CRC Unit, when computing the checksum

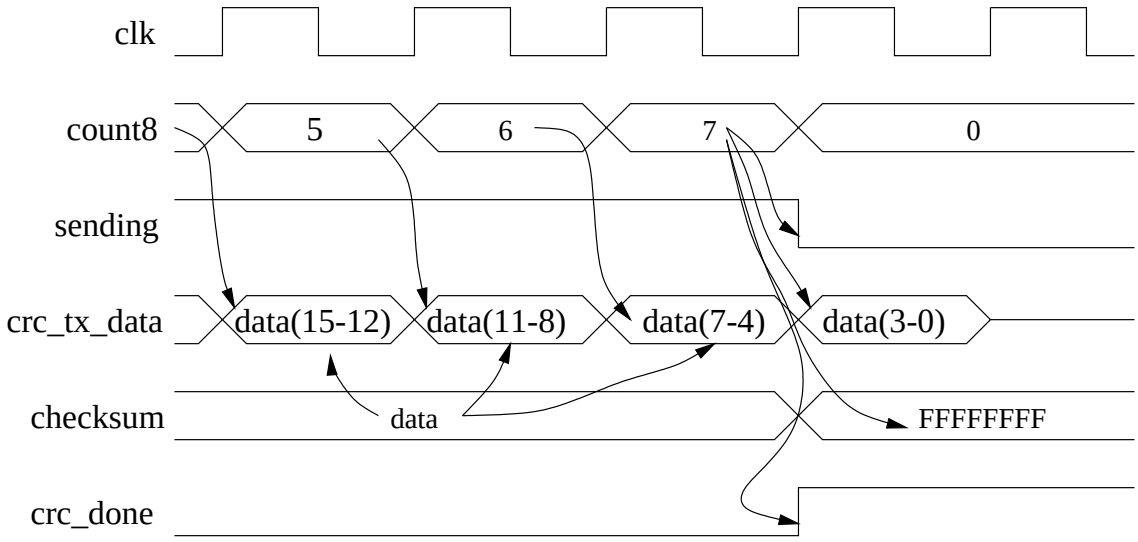

Figure 16: Wiring Diagrams for CRC Unit, when sending the computed checksum

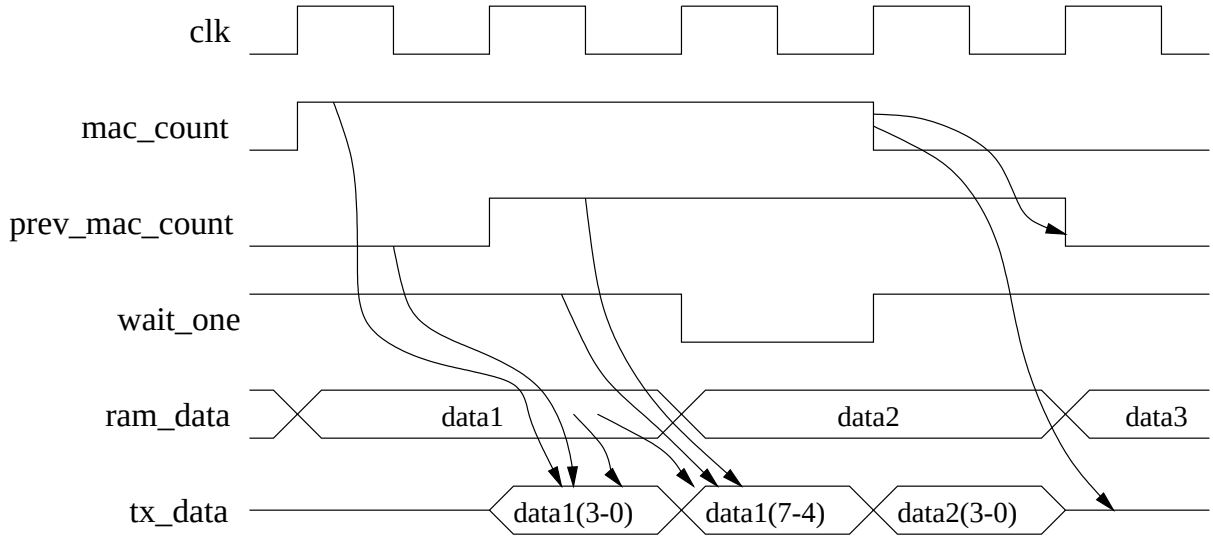

Figure 17: Timing Diagram for Operation of MAC

2.2.6 Media Access Controller

In its original implementation, the MAC was very much a media access controller. At this point, however, it simply has two jobs. The first is a minor CPLD issue. It must output a clock on the I09 data line so that the device on the first CPLD has a clock. The more important function of the MAC was to split up the data for the transceiver, from 8 bits to 4 bits. It assumed all data would stay on its pins for two clock cycles. In the first cycle, it samples and passes on the lower order bits. In the second cycle, it grabs the higher order bits.

2.2.7 Control Unit

The control unit was implemented as a two process fsm. The begin_state of the controller is entered only if reset is asserted at some time. At this point, all devices are disabled, and all internal signals are set to their appropriate base state (usually zero). From begin_state, the machine always transfers directly to the idle state. Most of the time, the idle state produces the same outputs as the begin_state and loops back to itself. However, if samp (the audio timing signal) is asserted, it will jump to the first audio state.

In the first audio state, the ram is enabled to write, and the analog to digital converter is enabled to do a conversion. Aud_count is asserted, as it was in the clock cycle before the fsm entered this state. In this way, the other devices waiting for the aud_count signal enter their audio states at the same time as the fsm. After one clock cycle, the machine advances to the second audio state. Here, the read signal to the converter is high, allowing the data to be read. The fsm checks to make sure that the status signal has gone low before proceeding. This conversion will generally happen fast enough that this is not really necessary, however. The expected clock cycles for aud_count to be asserted, then, is two, since it is not asserted if the machine is going to transition out of audioo2.

As the machine exits from the audio2 state, it increments an internal signal, effectively counting how many audio samples have been taken. When this count reaches 64, the machine goes from the idle state in the header_stuff state. In this state, it send control signals that will cause the counter to begin addressing the prom, and the MAC to split up the data coming out of the prom, and the CRC to begin calculating the checksum of the data passing through it. It also enables the transceiver to begin receiving data. The outputs from both the prom and ram are run through a tristate buffer, the LS244. The ram_prom signal is the controlling signal for these buffers, inverted for the ram buffer, since the buffers are active low. In this way, the controller dictates which data the MAC receives. The information for the ethernet frame, IP and UDP headers is stored statically on the PROM, to be accessed for each packet. The idea is that many source addressed could be burned onto the prom ahead of time, and a switch could decide which address would be receiving the packet (this is the purpose of the signal switch_source in the counter code).

Again, an internal count is used. The same signal is used, in fact, since it will be rolling over in the counting of audio samples anyway. In the header_stuff state, it counts only until forty, that being the number of bytes in the ethernet frame beginning, IP header, and UDP header. When it reaches this number, it jumps to the packeting state. In this state, the counter is again incremented until it reaches 64. In this time, the RAM output is enabled, and the Counter unit addresses it in sequence as long as the controller is in this state. The MAC and CRC continue to operate as they had during the header_stuff state.

When the packet is done, the controller enters into the framecheck state. Now, it stops the Counter and MAC, and has only the CRC operating, sending the computed checksum over the wire to finish the packet. When the CRC announces that it is done, then the controller jumps back into the idle state. Now, the transceiver is no longer enabled to receive data and knows that the packet has ended. The other devices are all similarly disabled. The controller will remain in that state until the samp timer becomes high again.

That is the normal operation of the fsm. However, there are some contingency plans. One

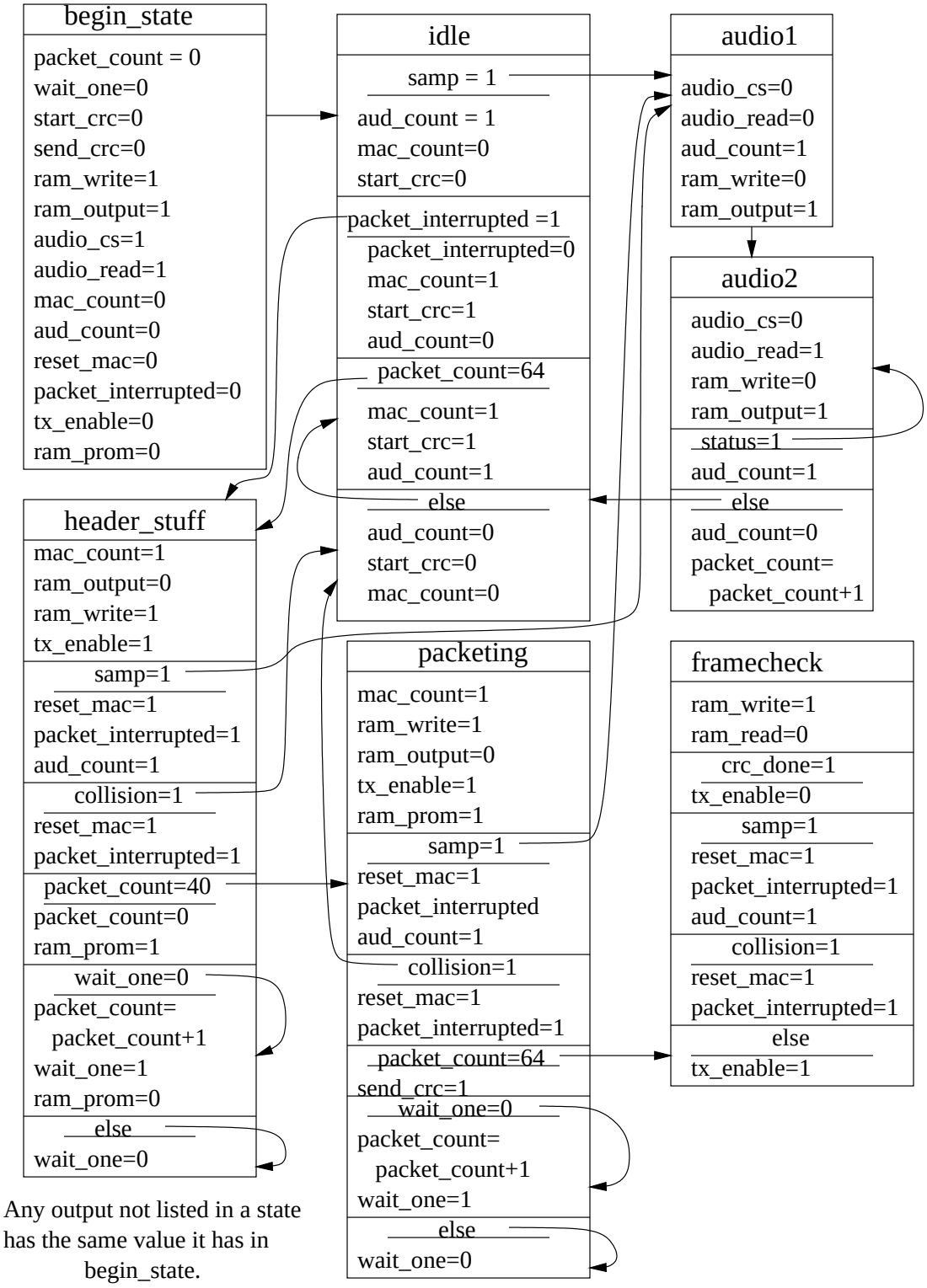

Figure 18: Transmitter Control FSM

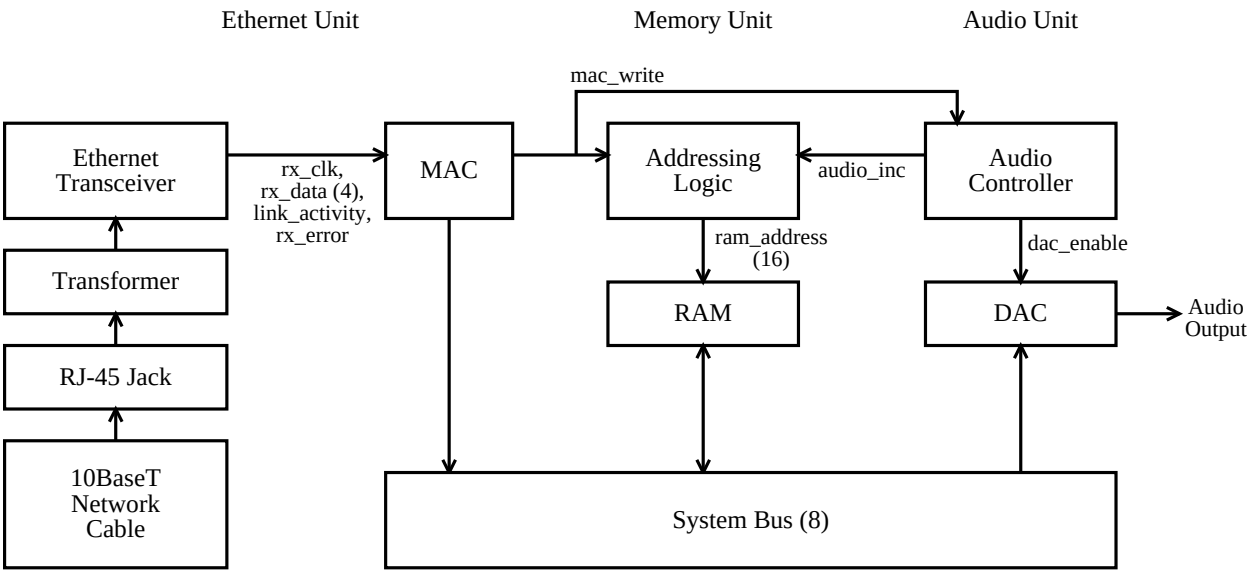

Figure 19: Receiver Block Diagram

way a packet could get interrupted is by having a collision on the line. Normally, if there is a collision, any devices attempting to send at that time back off for some unspecified amount of time (but at least 10 microseconds, usually). In this case, because we are expecting a dedicated cable, we simply attempt to send again right away. When a collision occurs, it sets the internal signal packet_interrupted high, asserts reset_mac, and jumps into the idle state. Because of the reset_mac, the mac address in the counter has returned to the first address of the data in the packet that was interrupted. In the idle state, the machine will see that there is data available to send, and will attempt to send it again. This brings up interesting timing issues. An entire packet is meant to be sent in between audio samples. There are approximately 300 clock cycles in between audio sampling time. The packets are 92 bytes long, and each byte requires two clock cycles. That is only 184 clock cycles, but it is certainly conceivable that collisions could push the pack back enough that the samp signal is being asserted in the middle of sending a packet.

In this case, the samp signal acts much like the collision. The mac is reset, the packet_interrrupted signal is set. The main difference is that with samp, the aud_count signal will be asserted, and the controller will jump to the first audio state rather than the idle state. Now, when it gets back to the idle state from the audio state, it will realize that there was an interrupted packet that is now waiting, and attempt to send that.

2.3 Receiver (Michael Salib)

The receiver is responsible for capturing audio data addressed to it on the ethernet network and buffering the data as needed. It also must convert the digitized samples to an analog signal for output to headphones or speakers. The receiver consists of three main components as shown in Figure 19. The Ethernet unit interfaces the receiver to the network and manages the actual filtering, reception, and unpacking of network data. The Memory unit buffers

incoming audio samples in preparation for their playback. The Audio unit provides the external audio interface for the receiver. It retrieves samples from the Memory unit at the proper times and plays them back using a digital to analog converter.

2.3.1 Ethernet Unit

This module interfaces the receiver to the network. It consists of an RJ-45 jack, a transformer, an ethernet transceiver, and a media access controller (MAC). The RJ-45 jack connects the unit to standard unshielded twisted pair category 5 cable (UTP Cat5). Immediately following that is a conditioning transformer, followed by the transceiver. The transceiver is an Intel LXT970A chip. It requires a 25 MHz clock for its internal operation but generates a 2.5 MHz receive clock that is used for all data and control operations.

The transceiver provides a standard interface to all media access controllers. That interface consists of output signals that indicate the recovered clock signal, link activity, received data validity, and the presence of received data errors. It also provides a data bus 4 bits wide on which data is driven synchronous to the recovered clock whenever valid data is available.

We built a media access controller to manage the ethernet transceiver's operations. This controller consists of a finite state machine clocked with the recovered 2.5 MHz clock signal from the transceiver. For each packet, the MAC examines network data provided by the transceiver, ignoring packets that aren't destined to the proper port and address. After ignoring unneeded header information, the MAC buffers the incoming nibbles of audio data into two bytes of local storage. Every four clock cycles, the MAC initiates a write cycle where it writes one byte of data per cycle for two cycles, leaving the RAM free for the remaining two clock cycles.

The MAC is controlled by an FSM described by the transition diagram in Figure **??**. For clarity, the details of the transitions associated with checking the IP address and UDP port are collapsed into single states. Details of those portions of the FSM are shown in Figures 21 and 22.

2.3.2 Memory Unit

The memory unit consists of a 32 kilobyte static RAM and some addressing logic. The RAM data bus is shared by the Ethernet unit's MAC and the Audio unit's digital to analog converter. The addressing logic contains two separate counters that can be used to address the RAM. These counters correspond to the current address for the Ethernet unit and Audio unit to be writing and reading data from respectively.

The system operates in one of two modes, depending on the value of the mac_write signal generated by the MAC. Ordinarily, when mac_write is low, the memory unit sets the RAM to write to the system bus and addresses the RAM with the audio counter. During these time periods, the Audio Unit can instruct the DAC to sample and hold the value on the system bus.

However, when the MAC has data ready to write, it asserts mac_write, indicating to both the Memory unit and the Audio unit that it will be writing data to the RAM for the next two clock cycles. The Memory unit responds by asserting the RAM's write enable and incrementing the MAC counter's value for each of the next two clock cycles. The Audio unit

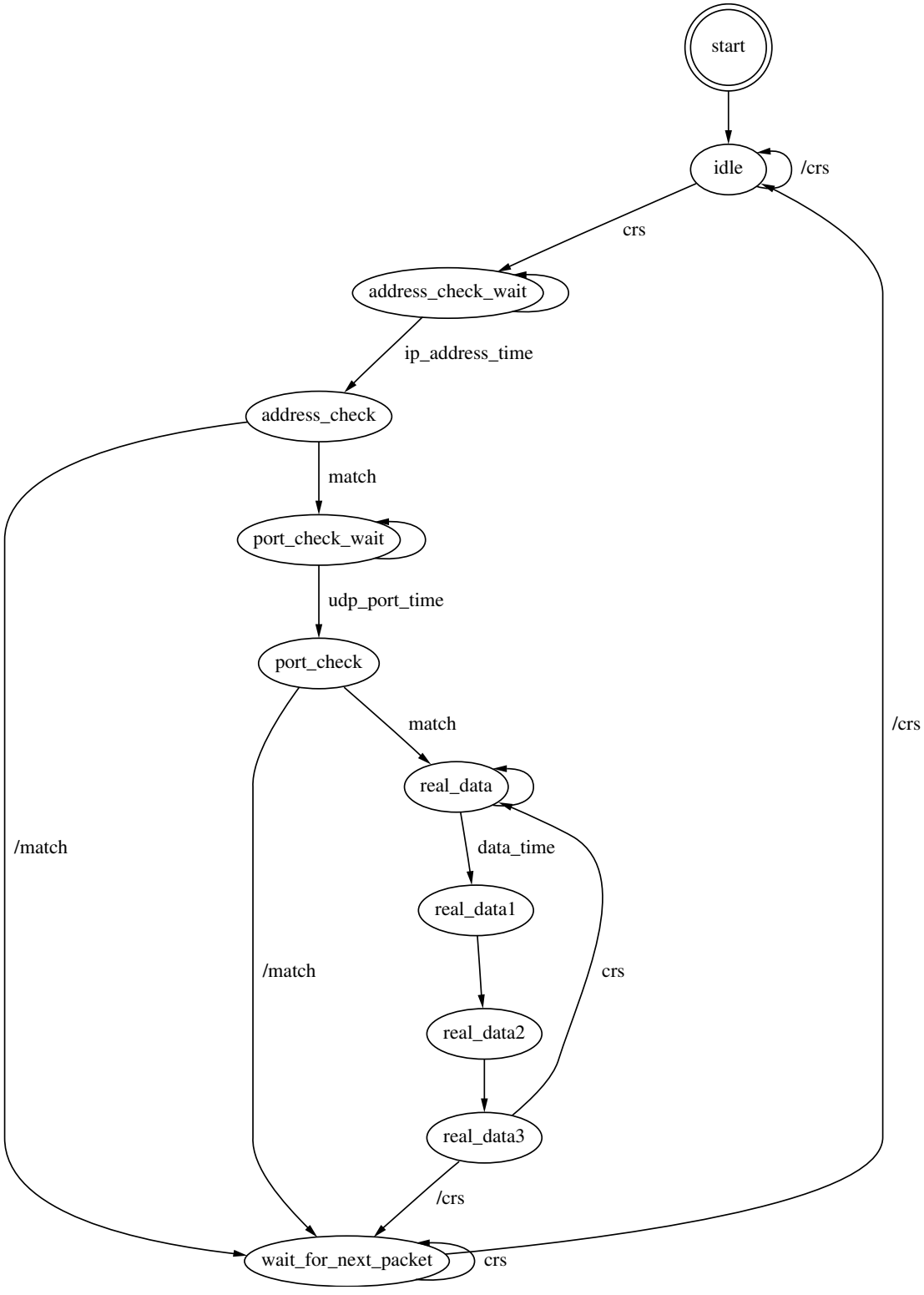

Figure 20: Receiver Media Access Controller FSM Diagram

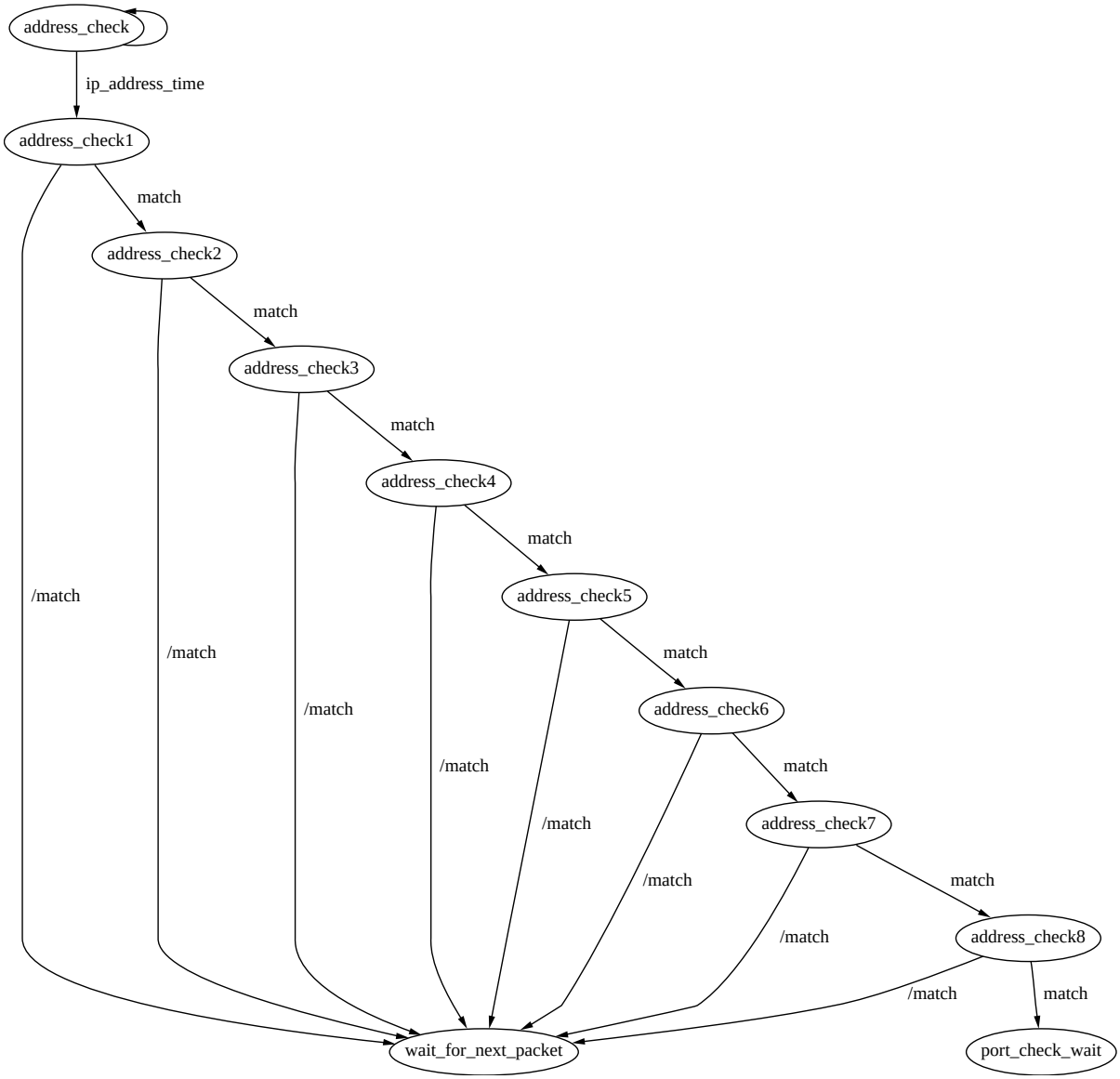

Figure 21: IP Address Checker details for the Receiver's MAC

responds by holding off on any sampling requests during those two clock cycles, postponing them for at most two clock cycles. In addition, the Audio unit can signal the Memory unit to increment its audio counter by asserting the audio-inc signal.

The system described above relies on a simple but effective buffering policy. Because the network can provide at most one byte of data for every two clock cycles, we can use a single shared RAM if read and write operations are interleaved. However, because the RAM requires that the address inputs be stable during write cycles, we would ordinarily be unable to write in two use consecutive cycles. To get around this restriction, we generate a RAM chip select input that is high during reads but is equivalent to the inverted system clock during writes. That guarantees that when the address inputs change between two consecutive writes, the RAM is temporarily disabled.

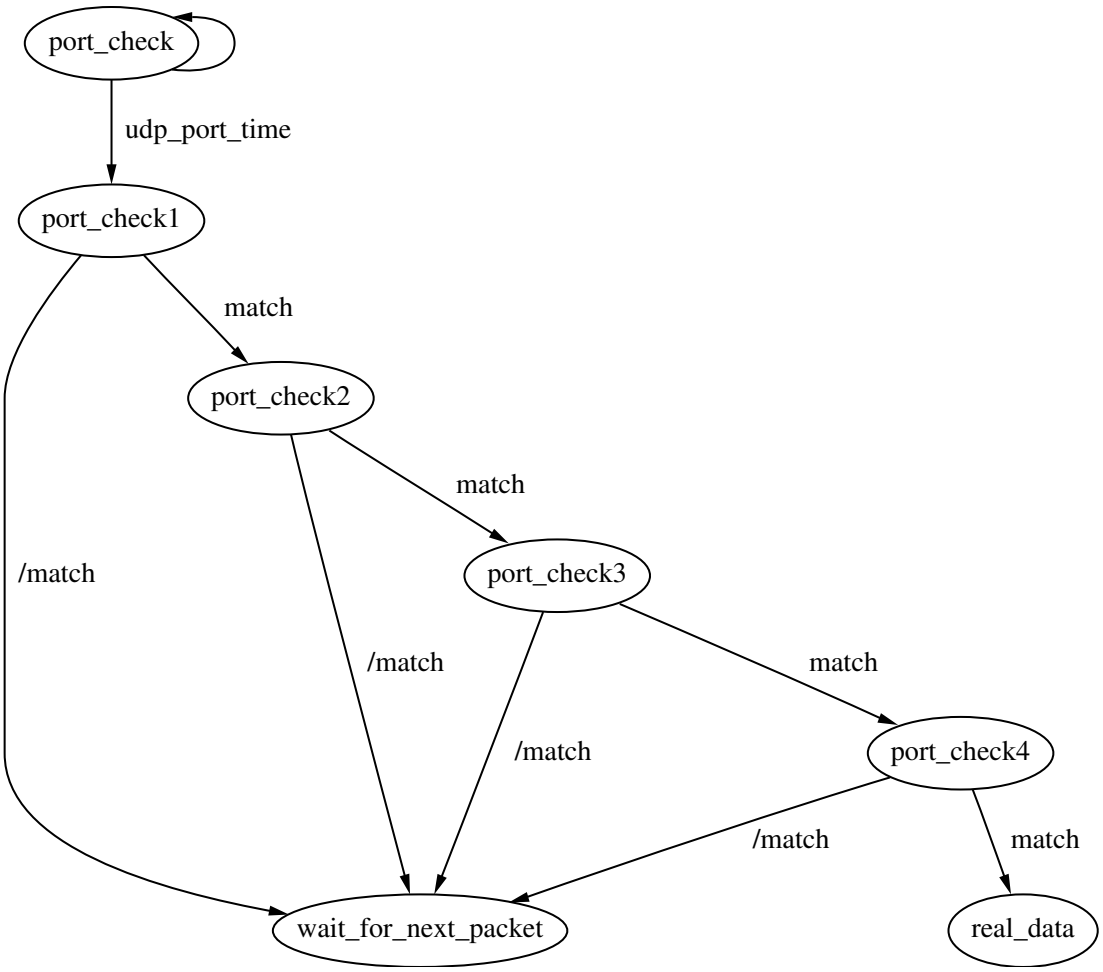

Figure 22: UDP Port Checker details for the Receiver's MAC

We address the RAM with two different counters, a network address and an audio address. As individual packets arrive, audio samples are placed into successively higher addresses in memory, incrementing the network address counter. Meanwhile, the audio unit is continually reading new samples from memory every 305 clock cycles, incrementing the audio address on each read. At any given time, all addresses above the audio address and below the network address represent valid network data that hasn't been played yet. Conversely, all addresses below the audio address and above the network address correspond to stale network data that has already been played.

2.3.3 Audio Unit

The Audio unit consists of an audio controller and a digital to analog converter (DAC). The audio controller consists of a finite state machine (FSM) as shown in Figure 23 and a timing counter. This timing counter is used to signal the FSM 8,192 times every second that a sample needs to be played. Upon receipt of this signal, the audio controller instructs the Memory unit to increment its audio address. If the mac_write signal is low, it instructs the

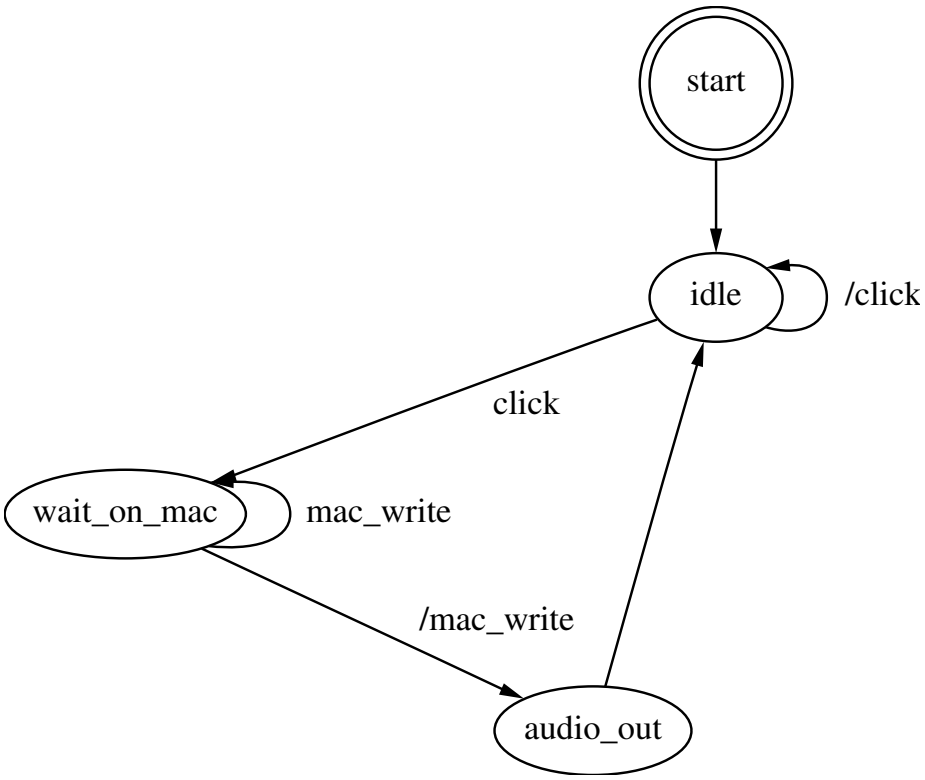

Figure 23: Receiver Audio Controller FSM Diagram

DAC to sample and hold the data on the system bus. Otherwise, it waits until mac_write goes low before telling the DAC to sample.

2.3.4 Failure Analysis

There are two possible failure modes that the Receiver must be prepared to deal with: underflow and overflow. Because we are using a datagram network protocol and because the underlying ethernet makes no guarantees about packet delivery, it is possible that the Receiver may play data faster than the transmitter can send it. Likewise, the lack of flow control suggests that the transmitter might send data faster than the receiver can play it, giving the receiver more data than it can buffer. In either case, it is impossible for the receiver to play the "correct" audio stream. Therefore, we've designed this buffering policy with the goal of minimizing the effect of underflow and overflow while striving to keep the implementation as simple as possible.

Our solution is to prevent the audio address from ever passing the network address. In the case of overflow, this means that the network address will be locked until the audio unit has cleared the backlog of available samples. The Receiver effectively throws out newly arriving network data once its buffer has been filled. In the case of underflow, this buffering policy means that the audio address will be locked until new data arrives from the network. In other words, the Receiver will continue to play the last sample of audio data available until new data arrives.

2.4 I can't believe its not ethernet! (Michael Salib)

Because we were unable to use the ethernet transceivers we had purchased for this project, we needed to create an alternative implementation that could substitute for ethernet so that we could test and complete our project. The system I devised was called "I can't believe its not ethernet!". It consisted of a transmitter and receiver connected by a specially constructed cable. The transmitter and receiver were designed to mimic the interface provided by actual ethernet transceivers.

The fake ethernet system I designed made use of an interconnect composed of 17 twisted pairs of wires. Each wire pair had a signal line and a ground line to mitigate the effects of interference and crosstalk. There were 16 data lines and one control line. Although the system I built supported only one way communication, it could easily be extended to handle bidirectional communication with collision detection in much the same way ethernet does. Doing so would involve adding another control line and merging the functions of the transmitter and receiver into a single component duplicated on both sides of the link. The only reason I did not do this was because we didn't need bidirectional communication and I believed it would be simpler to implement the two end points without regard for collision detection and arbitration issues.

The system I designed includes a detailed synchronization protocol to ensure that data is transmitted correctly even though the two end points are using clocks out of phase with each other and at slightly different frequencies. The synchronization protocol relies on a data valid signal generated by the transmitter. Every four clock cycles, the transmitter keeps data valid low for two cycles and then high for another two cycles. While data valid goes high, the transmitter writes 16 bits of data onto the data transmission lines and holds them for two cycles. The receiver samples the data on the transmission lines one cycle after it has seen the data valid signal goes high. This ensures that when the receiver samples network data, that data has been on the line unchanging for at least one (and probably more than one) clock cycle, giving time for the signals to propagate and any transient responses to settle out.

Unfortunately, this protocol mandates that data can be sent on only one out of every four clock cycles. Since standard ethernet transceivers move four bits of data on every clock cycle, we need to move four times as much data just to match ethernet. The ethernet substitute I designed needs many more wires (16 pairs versus 4 pairs for standard ethernet) and can only support two hosts (although in practice almost all ethernet connections are point to point). However, my implementation requires no sensitive analog components and runs with a 2.5 MHz clock, greatly simplifying implementation.

Both the transmitter and receiver portions are designed as two process FSMs as shown in Figures 24 and 25. These systems implement the synchronization protocol described above. The transmitter buffers incoming nibbles of data into a 12 bit buffer in preparation for transmission upon seeing the tx_enable signal go high. As a consequence of this buffering, data arrives at the reciever delayed by four clock cycles. The receiver is simpler; it only needs to buffer the incoming data before placing it on the 4-bit wide rv_data bus without worrying unduly about the special cases the transmitter must track.

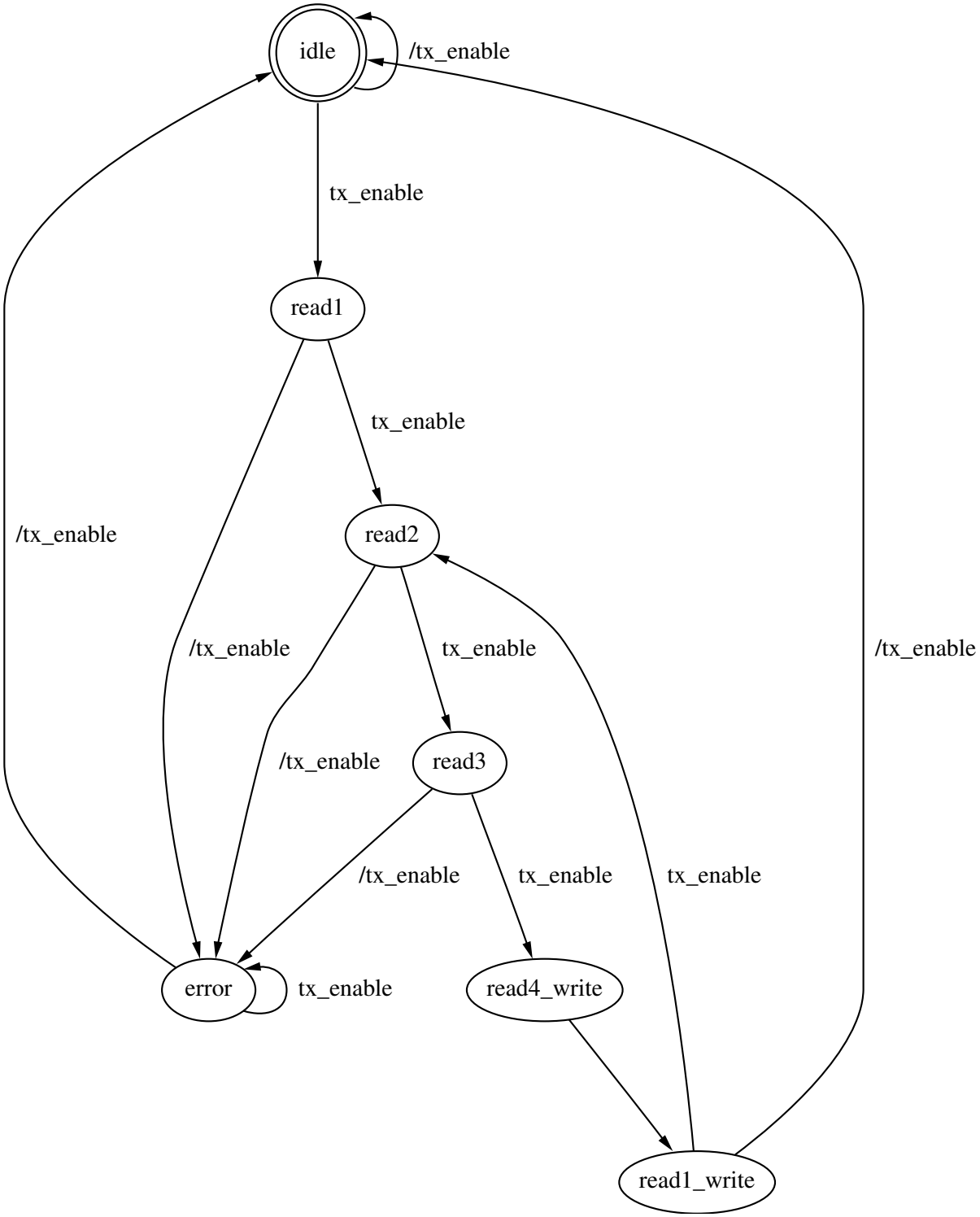

Figure 24: Ethernet Transmitter Control FSM

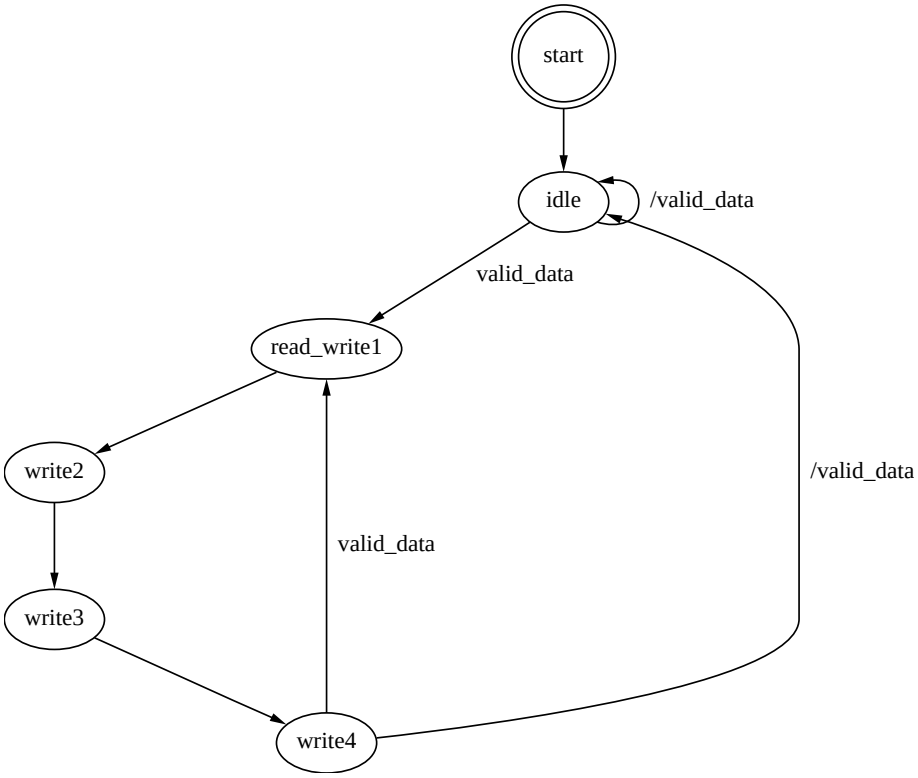

Figure 25: Ethernet Receiver Control FSM

3 Testing and Debugging

3.1 Transmitter Debugging (Jennifer Selby)

The vast majority of the transmitter was to be on the CPLD. Therefore, the early stages of debugging were mostly carefully going through the code and doing simulations using Nova. Once the simulations were working, each of the CPLDs would be programmed one at a time, and observed on the logic analyzer to check for correctness. The first was to be the controller, as the others would be much easier to test with a working controller. Once the controller appeared to be working, the counter would be added. The 8 bits that would be going into the MAC would be observed on the logic analyzer, and compared to the values expected for the IP headers and the values coming out of the analog to digital converter.

Once these bytes were satisfactory, the MAC would be programmed and wired into place. Again, the output, this time 4 bits, would be observed on the logic analyzer. Finally, the last piece, the CRC would be added. Testing this would require a lot of bit shifting to find the expected answer, but the most important thing would be to make sure that it was passing the values through and providing at least some sort of checksum, even if incorrect.

After that, it would just remain to connect the few wires to the transceiver and connect the ethernet cable either to a computer or to the receiver kit. Now, the test would be to listen to the music and compare it to the input music signals.

Alas, the project did not get nearly as far in these stages as I would hoped. Several

weeks were spent writing and rewriting the vhdl. The initial designs had nearly every sub component having both the address for the RAM and PROM and the data from them as buffers. Adding checksums and several internal counters meant that there were never enough macrocells or product terms available. Each setback involved an almost complete rewrite. Many times, the rewrite would involve a problem solved two versions ago in code that had other fundamental flaws.

I believe that the code I have now or something very close to it would work. However, I had left myself so little time to debug wiring after debugging my code, that I am unable to really verify my belief. In addition, I once again got bogged down in trivial wiring mistakes. At one point, over four hours were spent "debugging" my code, only to find out that while the wire to one particular signal had correctly been connected to the logic analyzer, it was one pin off for the actual CPLD unit, thereby shadowing the true error.

When I had to stop working on the lab, I had very little actual working parts. The sample timer worked, as did the audio states of the controller. However, the internal packet count seemed to remain static, as viewed on the logic analyzer, after making it available externally. My suspicion is that the fact that the state transition clock is itself not actually clocked may have had something to do with this. Had I more time, I would have attempted to rewrite the controller as a single clocked process state machine. In this way, the count would have to be a register, and would most likely increment correctly.

All of the modules appeared to work in simulation. I would have liked to test the other components more thoroughly on the logic analyzer, but I unfortunately allowed myself to get stuck on the controller, always thinking that it would start working any minute now, allowing for much easier testing of the other subdevices.

3.2 Receiver Test Plan (Michael Salib)

The Receiver will be built in several stages. Once a stage is completed, progress to the next stage will depend on successfully completing the tests associated with the first stage. All tests make use of a laptop to sit in place of the transmitter.

3.2.1 Basic Networking

This stage involves constructing all of the Ethernet Unit except for the MAC. There are two tests the system must pass on completion. For the first test, the link activity signal must go high while the laptop is sending any data onto the network and then go low when the laptop stops sending data. This ensures that the transceiver is wired correctly and functioning at a minimal level.

The second test requires that the laptop send packets of data filled with the same nibble. In order to pass, we must observe that particular nibble being driven onto the transceiver's four-bit data bus (preceded by some unspecified header data) using the logic analyzer.

3.2.2 Talking to the Network

Once the transceiver is working correctly, we can construct the MAC. Upon completion of this stage, the system should pass two tests. The first test involves watching the mac_write

signal and verifying that it goes high only when the laptop is sending data to the proper port and address. This test verifies that the MAC correctly recognizes only packets destined for the receiver.

The second test requires that we observe the MAC data bus being driven with the correct value while the receiver is being sent a constant data stream from the laptop. Whereas the second test in the previous stage examined the transceiver output bus, this test focuses on the MAC's output to the system bus. This test also requires that the proper timing of writes be observed on the system bus (data only gets written two cycles out of every four cycles).

3.2.3 Playing Basic Network Audio

When the Ethernet unit is complete, we'll being construction on the Audio unit. This involves adding the DAC and wiring it so that its enable signal is connected to mac_write. This configuration should cause the DAC to output whatever data the Receiver gets, continuously. We'll test this by sending a sine wave data stream and examining the output using a digital oscilloscope.

3.2.4 Memory Unit Integration

At this point we'll add the static RAM and it's associated addressing logic. Then we'll verify that the system still passes the same tests from the previous stage.

3.2.5 Playing High Quality Audio

For the final stage, we'll add the audio controller, making the Receiver complete. The first test involves watching the audio counter increment signal using the digital oscilloscope and verifying that it goes high for one clock cycle out of every 305 clock cycles. This ensures that the Receiver will play a new sample 8,192 times per second. Finally, we'll send a sine wave data stream to the Receiver and verify that it produces the correct output using a digital oscilloscope.

3.3 Receiver Debugging (Michael Salib)

We had originally planned to use ethernet transceivers made by Intel. Unfortunately, the only transceivers we could find either interfaced to MACs serially, which would require our system to run at 10 MHz, or came in package types not suitable for use in the 6.111 laboratory kit. We eventually located a supplier that could provide adapters between the Quad Flat Pack package the transceivers used and the Dual Inline Package that the laboratory kits supported. At the last minute, we learned that the adapters we needed were out of stock and would be unavailable until after the project was due. There were alternative suppliers of these components available, but their products were priced far outside the means of the class budget.

Without the adapters, we were unable to use our ethernet transceivers, the core component of our project. We investigated several approaches to salvaging them, such as soldering wire leads directly to the pins. However, once the parts arrived, we realized that the pins

were far too small and too closely spaced together for effective soldering. As a result, our ethernet transceivers were completely worthless. Not only did this impede the construction of our project, it made independent testing impossible. Without the transceivers, there was no way to connect our designs to a computer for testing as described above. This meant that we had to use the logic analyzer for debugging. While a powerful tool, it fell short of the task of packet debugging. The lack of transceivers also meant that we couldn't build and test the transmitter and receiver separately; full testing required both units to be functional. This effectively prevented me from using the test plan I had originally developed.

At the last minute, I designed the "I can't believe its not ethernet!" system to substitute for our ethernet transceivers. Afterwards I learned that my partner had used up all of her CPLDs and almost all of the available data lines. After trying unsuccessfully to coerce her code to fit into fewer CPLDs and use fewer input-output pins, I found out about the 372i CPLDs that the lab still stocked. These devices were only slightly smaller than the 374i CPLDs in the 6.111 lab kit, but more importantly, they didn't tie up valuable pins on the shared system buses! It took a great deal of effort to locate the chips and tools needed to program them, as well as people who knew how to use them. This effort comprised a fair amount of my debugging time.

In order to get the ethernet substitute system working, I devoted several hours to working out a synchronization protocol, designing and implementing the ethernet substitute and building a test environment in which I could debug it. The protocol had to be both safe enough to reliably transport data and fast enough for our needs. The test environment consisted of receiver component of the ethernet substitute implemented in a CPLD in my lab kit connected with our custom cable to a 372i on my lab kit that contained the transmitter portion of the ethernet substitute. I wired the transmitter's data inputs to a 4-bit binary counter and configured to transmit continuously. I used this environment to examine and debug interactions between the transmitter and receiver by using the logic analyzer to display the relevant control signals next to the sent and received data nibbles. This allowed me to debug the ethernet substitute thoroughly enough so that it worked successfully.

Debugging the ethernet substitute was considerably harder than it should have been due to bus contention issues and the excess of electrical noise. Oscilloscope traces showed that my system was exhibiting signals near and sometimes at invalid logic levels. Some of these issues were resolved when I discovered that the Digital to Analog Converter I was using has been inserted upside down, but many persisted. To some extent, these kinds of noise problems are expected when working with large bundles of long wires, but I experienced many of these problems even when I used short wiring directly to the bus instead of long cables.

Upon completing a working ethernet substitute, I began debugging the MAC in the hopes of getting a partially working system. By analyzing interactions between the MAC and the fake ethernet receiver using the logic analyzer, I was able to locate and correct a number of problems in the implementation. I had planned to demonstrate a functional (if stripped down) MAC that could at successfully decode packets of network data and play the resulting audio samples directly using the DAC. Such a system would bypass the RAM and memory unit entirely in the interest of simplicity and time. Unfortunately, I was unable to get the MAC working well enough to do this. The MAC behaved correctly in simulation, but not in

the actual system.

4 Conclusion

4.1 Jennifer Selby

Despite various troubles in working on this project, the principal cause that the transmitter is not is any kind of a working state is a due to poor time estimations. As mentioned above, far too much time was devoted to rewriting vhdl code rather than coming up with a good design upfront, sticking with it, and leaving plenty of time for debugging wiring errors. The time left for fixing errors in the actual kits, as opposed to an electronic simulation, was not anywhere near adequate. With more time, I would concentrate less on having all code fully complete, and more on getting small, manageable, parts working, that could eventually be integrated into one large working thing. In other words, I would attempt to design a system that was more modular, for both the purposes of building and testing. I would emphasize trying to find a protocol by which each component would communicate with the others, and keep that unchanging, even if implementations changed, rather than changing every component many times over.

There are some things also that I would have liked to add to my design. I would have liked a nice, clean way to set source addresses dynamically. Using the switches would just be horribly long and messy (6 bytes of MAC address, 4 of network, and 2 of port). Parts that could simplify this would have been a nice addition. This would have added a couple states to the controller, to handle switching back and forth between different storage devices. It also would mean that the IP checksum was no longer static, but that should not be too hard to compute, though it might mean that the device would require more than just the 4 374I CPLDs on the protoboard. I also noticed a few minor errors in my implementation of the ethernet frame while writing this report, such as taking the cyclical redundancy check of the entire frame, instead of starting with the MAC source address. That would be something fairly easy to fix, which would help for correctness. Basically, the CRC would pass through at all times, not just when the controller tells it to start or send. The start signal would mean start computing using this data, rather than start operating. This would not affect the transceiver, because the enable pin would still be controlled by the fsm, and it would only see the data at the correct times.

The one thing that I was happy about in my design is the passing of data. Among the CPLD components, there is one shared bus, between the MAC and CRC units, and it is only 4 bits long. As I mentioned above, my original implementations had many huge shared buses that took up much more room than necessary. I was happy to have come up with a way to split up the tasks in a way that left much room in the CPLDs, despite the very large data vectors necessary for the operation of this device.

4.2 Michael Salib

Overall, this project was a frustrating experience. Many of the problems we faced were directly related to our lack of real ethernet transceivers and the need to design, build, and

debug a substitute communications system on the spot. All of this took up valuable time that should have been spent working on debugging our actual project. If we had been able to use the ethernet transceivers we ordered, I think we would have completed the project on time, or at least completed a much larger portion of it.

Looking back, I feel I was responsible for our difficulty with the ethernet transceivers. I didn't anticipate the difficulty involved in getting modern chips to work in our lab kits. I think another part of our problem was my own optimism which prevented me from triaging the situation early enough. If I had realized that we weren't likely to complete a working project earlier than I did, we would have had more time to complete a reduced project that at least worked.

Despite the fact that we were unable to construct a working system, we did a great deal of work and learned a lot. I am confident that the system we designed would work if we had had more time for testing and debugging or if we had acquired usable ethernet transceivers. I am comforted by the fact that if nothing else, I produced a strongly decentralized design consisting of small, decoupled, easily understood components. I'm also gratified that despite the immense time pressure, I was able to successfully complete the ethernet substitute system.

A Receiver VHDL Code (Michael Salib)

A.1 Media Access Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity mac is
  port (
    clk, reset    : in  std_logic;
    crs, rx_error : in  std_logic;
    rx_data       : in  std_logic_vector(3 downto 0);
    mac_write     : out std_logic;
    sys_bus       : out std_logic_vector(7 downto 0));

 attribute pin_avoid of mac :entity is
   "11 21 22 32 42 43 44 53 63 64 74 83"&  -- Vdd, Gnd, VPP
   " 13 "&  -- This is I0-9. Can screw up the clock of C1. Be
                         -- careful when using this.
   " 23 62 65 "&
   --" 71 "&   --must be grounded for K1 interface

   -- this line lists all the logic analyzer connections...
   --"3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 71 75 76 77 78 79 80 81 82"&
```

```
   " 14 35 41 51 72 "; -- Used by Programmer. No external connection.

   attribute pin_numbers of mac :entity is
     "rx_data(3):3 rx_data(2):4 rx_data(1):5 rx_data(0):6 "&
     "sys_bus(0):31 sys_bus(1):30 sys_bus(2):29 sys_bus(3):28 "&
     "sys_bus(4):27 sys_bus(5):26 sys_bus(6):25 sys_bus(7):24 "&
     "mac_write:56 crs:57 rx_error:58 reset:59 "&
     "";
  -- rx_data: L1:0-3   sys_bus: A24-A31
  -- mac_write: A24  crs: A25  rx_error: A26  reset: A27
end mac;


architecture state of mac is
  type stateType is (start, idle, wait_for_next_packet,
                     address_check, port_check, real_data0,
                     address_check1, address_check2, address_check3, address_check4,
                     address_check5, address_check6, address_check7, address_check8,
                     real_data, real_data1, real_data2, real_data3,
                     port_check1, port_check2, port_check3, port_check4);

  signal present_state, next_state           : stateType;
  signal nibble_counter                      : unsigned(7 downto 0);
  signal pulsified_crs, current_crs, prev_crs : std_logic;
  signal ip_address_now, udp_port_now        : std_logic;
  signal data_now, data_end_now              : std_logic;
  signal a, b, c, d : std_logic_vector(3 downto 0);


  constant udp_port   : std_logic_vector(15 downto 0) := "0101001000001000";
  --"0101 0010 0000 1000";
  -- this is 21000 in big endian, i think
  constant ip_address : std_logic_vector(31 downto 0) := "00001010000000000000000000001100
  --  "0000 1010 0000 0000 0000 0000 0011 0011";
  -- 10 0 0 51 in big endian, i think

  constant ip_address_time : unsigned(7 downto 0) := "01011100";
  constant udp_port_time   : unsigned(7 downto 0) := "01101000";
  constant data_time       : unsigned(7 downto 0) := "01110100";
  constant data_end_time   : unsigned(7 downto 0) := "11110100";
  -- destination ip address occurs 368 bits after the first bit of preamble
  -- destination udp port occurs 416 after...
  -- data occurs 464 bits after...
  -- data ends 976 bits after...
```

```
  -- 92, 104, 116, 244 nibbles after...
  signal internal_sys_bus : std_logic_vector(7 downto 0);
  signal sys_bus_enable : std_logic;

begin  -- state

  state_clk: process(clk, reset)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_state <= start;
      else
        present_state <= next_state;
      end if;
    end if;
  end process state_clk;

  state_comb: process(present_state, next_state, crs, --mac_buffer,
                      ip_address_now, udp_port_now, data_now, data_end_now, rx_data)
  begin
    case present_state is
      when start =>
        next_state <= idle;
        mac_write <= '0';
        sys_bus_enable <= '0';

      when idle =>
        if crs = '1' then
          next_state <= address_check;
        else
          next_state <= idle;
        end if;

      when wait_for_next_packet =>
        mac_write <= '0';
        sys_bus_enable <= '0';

        if crs = '0' then
          next_state <= idle;
        else
          next_state <= present_state;
        end if;
```

```
when address_check =>
  if ip_address_now = '1' then
    next_state <= address_check1;
  else
    next_state <= present_state;
  end if;

when address_check1 =>
  if rx_data(3 downto 0) = ip_address(31 downto 28) then
    next_state <= address_check2;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check2 =>
  if rx_data(3 downto 0) = ip_address(27 downto 24) then
    next_state <= address_check3;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check3 =>
  if rx_data(3 downto 0) = ip_address(23 downto 20) then
    next_state <= address_check4;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check4 =>
  if rx_data(3 downto 0) = ip_address(19 downto 16) then
    next_state <= address_check5;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check5 =>
  if rx_data(3 downto 0) = ip_address(15 downto 12) then
    next_state <= address_check6 ;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check6 =>
  if rx_data(3 downto 0) = ip_address(11 downto 8) then
```

```
    next_state <= address_check7;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check7 =>
  if rx_data(3 downto 0) = ip_address(7 downto 4) then
    next_state <= address_check8;
  else
    next_state <= wait_for_next_packet;
  end if;

when address_check8 =>
  if rx_data(3 downto 0) = ip_address(3 downto 0) then
    next_state <= port_check;
  else
    next_state <= wait_for_next_packet;
  end if;


when port_check =>
  if udp_port_now = '1' then
    next_state <= port_check1;
  else
    next_state <= present_state;
  end if;

when port_check1 =>
  if rx_data(3 downto 0) = udp_port(15 downto 12) then
    next_state <= port_check2;
  else
    next_state <= wait_for_next_packet;
  end if;

when port_check2 =>
  if rx_data(3 downto 0) = udp_port(11 downto 8) then
    next_state <= port_check3;
  else
    next_state <= wait_for_next_packet;
  end if;

when port_check3 =>
  if rx_data(3 downto 0) = udp_port(7 downto 4) then
    next_state <= port_check4;
```

```
        else
          next_state <= wait_for_next_packet;
        end if;

      when port_check4 =>
        if rx_data(3 downto 0) = udp_port(3 downto 0) then
          next_state <= real_data;
        else
          next_state <= wait_for_next_packet;
        end if;

--mac_write leads the sys_bus writes by one cycle; its high iff the sysbus will
--be written to in the next cycle
      when real_data =>
        if data_now = '1' then
          next_state <= real_data0;
        else
          next_state <= present_state;
        end if;

      when real_data0 =>
        mac_write <= '0';
        a <= rx_data;
        sys_bus_enable <= '1';
        internal_sys_bus <= d & c;
        next_state <= real_data1;

      when real_data1 =>
        mac_write <= '0';
        sys_bus_enable <= '0';
        b <= rx_data;
        next_state <= real_data2;

      when real_data2 =>
        mac_write <= '1';
        sys_bus_enable <= '0';
        c <= rx_data;
        next_state <= real_data3;

      when real_data3 =>
        mac_write <= '1';
        sys_bus_enable <= '1';
        d <= rx_data;
        internal_sys_bus <= b & a;
```

```
      if (crs = '1') and not(data_end_now = '1') then
        next_state <= real_data0;
      else
        next_state <= wait_for_next_packet;
      end if;

    when others =>
      next_state <= start;

  end case;
end process state_comb;


sys_bus <= internal_sys_bus when sys_bus_enable = '1' else "ZZZZZZZZ";


-- turns crs into a pulse (delayed by one cycle)
pulsify_crs: process(clk)
begin
  if rising_edge(clk) then
    current_crs <= crs;
    prev_crs <= current_crs;
  end if;
end process pulsify_crs;
pulsified_crs <= not(prev_crs) and current_crs;

-- a counter: reset to zero on pulsified crs
-- crs is enable
nibble_count: process(clk)
begin
  if rising_edge(clk) then
    if pulsified_crs = '1' then
      nibble_counter <= "00000000";
      -- we need to start at one b/c our load signal
      --  (pulsified_crs) is delayed by one cycle
      --  from the real crs.
      -- but b/c we need the data in the state after
      --  the wait state, we start at zero.
    elsif crs = '1' then
      nibble_counter <= nibble_counter + "00000001";
    end if;
  end if;
end process nibble_count;
```

```
  ip_address_now <= '1' when nibble_counter = ip_address_time else '0';
  udp_port_now   <= '1' when nibble_counter = udp_port_time   else '0';
  data_now       <= '1' when nibble_counter >= data_time       else '0';
  data_end_now   <= '1' when nibble_counter >= data_end_time   else '0';

end state;
```

A.2 Top Level file for second CPLD

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.memory_unit;
use work.audio_controller;

entity cpld is
  port (
    clk, reset, mac_write  : in  std_logic;
    dac_enable, ram_cs     : out std_logic;
    not_ram_oe, not_ram_we : out std_logic;
    ram_address            : out unsigned(12 downto 0));

  attribute pin_avoid of cpld          : entity is
    --"11 21 22 32 42 43 44 53 63 64 74 83"&  -- Vdd, Gnd, VPP
    " 13 "&  -- This is I0-9. Can screw up the clock of C1. Be
                         -- careful when using this.
    " 23 62 65 "&
    --" 71 "&   --must be grounded for K1 interface

    -- this line lists all the logic analyzer connections...
    --"3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 71 75 76 77 78 79 80 81 82"&

    " 14 35 41 51 72 "; -- Used by Programmer. No external connection.

  attribute pin_numbers of cpld :entity is
    "reset:59 mac_write:56 "&
    -- reset:A27 mac_write:A24
    "dac_enable:60 not_ram_we:61 not_ram_oe:66 ram_cs:55 "&
    -- dac_enable:A28 not_ram_we:A29 not_ram_oe:A30 ram_cs:A23
    "ram_address(12):50 ram_address(11):49 ram_address(10):48 ram_address(9):47  "&
```

```
    "ram_address(8):46 ram_address(7):45 ram_address(6):40 ram_address(5):39 "&
    "ram_address(4):38 ram_address(3):37 ram_address(2):36 ram_address(1):34 ram_address
    -- ram_address:A8-A20
end cpld;

architecture x of cpld is
  signal ram_oe, ram_we, audio_inc : std_logic;
begin  -- x

  not_ram_we <= not(ram_we);
  not_ram_oe <= not(ram_oe);

  ac : audio_controller port map (
    clk        => clk,
    reset      => reset,
    mac_write  => mac_write,
    audio_inc  => audio_inc,
    dac_enable => dac_enable);

  mu : memory_unit port map (
    clk         => clk,
    reset       => reset,
    mac_write   => mac_write,
    audio_inc   => audio_inc,
    ram_oe      => ram_oe,
    ram_we      => ram_we,
    ram_cs      => ram_cs,
    ram_address => ram_address);

end x;
```

A.3 Audio Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity audio_controller is
  port (
    clk, reset, mac_write : in  std_logic;
```

```
    audio_inc, dac_enable : out std_logic);
end audio_controller;


architecture cc of audio_controller is
  -- counter
  signal   click     : std_logic;
  signal   counter   : unsigned(8 downto 0);
  constant max_count : unsigned(8 downto 0) := "100110001";
  -- 305 cycles for a 2.5MHz for a 1.8432 MHz clk use 225 = "0111000001"

  -- fsm controller
  type stateType is (start, idle, wait_on_mac, audio_out);
  signal present_state, next_state : stateType;

begin  -- cc
  -- timer
  timer: process(clk)
  begin
    if rising_edge(clk) then
      if counter = max_count then
        click <= '1';
        counter <= "000000000";
      else
        counter <= counter + "000000001";
        click <= '0';
      end if;
    end if;
  end process timer;


  -- fsm controller
  state_clk: process(clk, reset)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_state <= start;
      else
        present_state <= next_state;
      end if;
    end if;
  end process state_clk;

  state_comb: process(present_state, next_state, click, mac_write)
```

```
  begin
    case present_state is
      when start =>
        next_state <= idle;
        dac_enable <= '0';
        audio_inc <= '0';

      when idle =>
        dac_enable <= '0';
        audio_inc <= '0';
        if click = '1' then
          next_state <= wait_on_mac;
        else
          next_state <= idle;
        end if;

      when wait_on_mac =>
        if mac_write = '0' then
          next_state <= audio_out;
          dac_enable <= '1';
        else
          next_state <= wait_on_mac;
        end if;

      when audio_out =>
        next_state <= idle;
        audio_inc <= '1';
        dac_enable <= '0';

      when others =>
        next_state <= start;

    end case;
  end process state_comb;
end cc;
```

A.4 Memory Controller

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity memory_unit is
  port (
```

```
    clk, reset, mac_write, audio_inc : in  std_logic;
    ram_oe, ram_we, ram_cs           : out std_logic;
    ram_address                      : out unsigned(12 downto 0));
end memory_unit;


architecture ccs of memory_unit is
  signal audio_counter, net_counter : unsigned(12 downto 0);
  signal address_match : std_logic;
begin  -- ccs

  -- counter management is handled independantly of everything else
  counter_management: process(clk, audio_inc, mac_write)
  begin
    if rising_edge(clk) then
      if (address_match = '0') then
        if (mac_write = '1') then
          net_counter <= net_counter + "0000000000001";
        end if;

        if audio_inc = '1' then
          audio_counter <= audio_counter + "0000000000001";
        end if;

      elsif reset = '1' then
        net_counter <= "0000000000001";
        audio_counter <= "0000000000000";
      end if;

    end if;
  end process counter_management;

  address_match <= '1' when (net_counter = audio_counter) else '0';

  ram_cs <= not(clk) when mac_write = '1' else '1';

  controller: process(clk)
  begin
    if rising_edge(clk) then
      if mac_write = '1' then
        ram_address <= net_counter;
        ram_oe <= '0';
        ram_we <= '1';
      else
```

```
        ram_address <= audio_counter;
        ram_oe <= '1';
        ram_we <= '0';
      end if;
    end if;
  end process controller;
end ccs;
```

B I can't believe its not ethernet! (Michael Salib)

B.1 Transmitter Ethernet Implementation

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity tx_i_cant_believe_its_not_ethernet is
  port (
    clk, reset    : in  std_logic;
    tx_enable     : in  std_logic;
    tx_data       : in  std_logic_vector(3 downto 0);
    dv            : out std_logic;
    data          : out std_logic_vector(15 downto 0);
    crs, tx_error : out std_logic);

  attribute pin_numbers of tx_i_cant_believe_its_not_ethernet :entity is
    "clk:13 reset:2 tx_enable:3 tx_error:4 crs:5 "&
    "tx_data(0):6 tx_data(1):7 tx_data(2):8 tx_data(3):9 "&
    "data(0):14 data(1):15 data(2):16 data(3):17  "&
    "data(4):18 data(5):19 data(6):20 data(7):21  "&
    "data(8):24 data(9):25 data(10):26 data(11):27  "&
    "data(12):28 data(13):29 data(14):30 data(15):31  "&
    "dv:36";

  -- pin 11 is isr_enable
  -- Vcc pins = 22, 44
  -- Gnd pins = 1, 12, 23, 34
end tx_i_cant_believe_its_not_ethernet;


architecture cdc of tx_i_cant_believe_its_not_ethernet is
  type stateType is (idle, read1, read2, read3, read4_write,
                     read1_write, error_state);
```

```
  signal present_state, next_state : stateType;
  signal ethernet_buffer : std_logic_vector(11 downto 0);

begin  -- cdc

  state_clk: process(clk, reset)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_state <= idle;
      else
        present_state <= next_state;
      end if;
    end if;
  end process state_clk;


  state_comb: process(present_state, next_state, crs, tx_data,
                      tx_enable, ethernet_buffer, data)
  begin
    case present_state is
      when idle =>
        crs <= '0';
        dv <= '0';
        data <= "ZZZZZZZZZZZZZZZZ";
        tx_error <= '0';
        if tx_enable = '1' then
          next_state <= read1;
        else
          next_state <= idle;
        end if;

      when read1 =>
        crs <= '1';
        dv <= '0';
        ethernet_buffer(3 downto 0) <= tx_data;
        if tx_enable = '1' then
          next_state <= read2;
        else
          next_state <= error_state;
        end if;

      when read2 =>
        crs <= '1';
```

```
  dv <= '0';
  ethernet_buffer(7 downto 4) <= tx_data;
  if tx_enable = '1' then
    next_state <= read3;
  else
    next_state <= error_state;
  end if;

when read3 =>
  crs <= '1';
  dv <= '0';
  ethernet_buffer(11 downto 8) <= tx_data;
  if tx_enable = '1' then
    next_state <= read4_write;
  else
    next_state <= error_state;
  end if;

when read4_write =>
  crs <= '1';
  dv <= '1';
  data <= tx_data & ethernet_buffer;
  next_state <= read1_write;

when read1_write =>
  crs <= tx_enable;
  dv <= '1';
  ethernet_buffer(3 downto 0) <= tx_data;
  if tx_enable = '1' then
    next_state <= read2;
  else
    next_state <= idle;
  end if;

when error_state =>
  crs <= '0';
  dv <= '0';
  tx_error <= '1';
  data <= "ZZZZZZZZZZZZZZZZ";
  if tx_enable = '1' then
    next_state <= error_state;
  else
    next_state <= idle;
  end if;
```

```
      -- might have problems here if jen's
      -- code gets confused by a lo crs
      -- even though tx_enable is hi...
    end case;
  end process state_comb;
end cdc;
```

B.2 Receiver Ethernet Implementation

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


entity rv_i_cant_believe_its_not_ethernet is
  port (
    clk, reset    : in  std_logic;
    dv            : in  std_logic;
    data          : in  std_logic_vector(15 downto 0);
    rx_data       : out std_logic_vector(3 downto 0);
    crs, rx_error : out std_logic);

  attribute pin_avoid of rv_i_cant_believe_its_not_ethernet :entity is
   "11 21 22 32 42 43 44 53 63 64 74 83"&  -- Vdd, Gnd, VPP
   " 13 "&  -- This is I0-9. Can screw up the clock of C1. Be
                        -- careful when using this.
   " 23 62 65 "&
   --" 71 "&   --must be grounded for K1 interface

   -- this line lists all the logic analyzer connections...
   --"3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 71 75 76 77 78 79 80 81 82"&

   " 14 35 41 51 72 "; -- Used by Programmer. No external connection.

   attribute pin_numbers of rv_i_cant_believe_its_not_ethernet :entity is
     "rx_data(3):3 rx_data(2):4 rx_data(1):5 rx_data(0):6 "&
     "data(15):7 data(14):8 data(13):9 data(12):10 "&
     "data(11):15 data(10):16 data(9):17 data(8):18 "&
     "data(7):67 data(6):68 data(5):69 data(4):70 "&
     "data(3):75 data(2):76 data(1):77 data(0):78 "&
```

```
     "dv:79  crs:57 rx_error:58 reset:59";
  -- rx_data: L1:0-3 dv:L2-4 data : L1:4-15, L2:0-3
  -- crs: A25  rx_error: A26  reset: A27
end rv_i_cant_believe_its_not_ethernet;


architecture cdc of rv_i_cant_believe_its_not_ethernet is
  type stateType is (start, idle, read_write1, write2, write3, write4);
  signal present_state, next_state : stateType;
  signal ethernet_buffer : std_logic_vector(11 downto 0);
  signal valid_data, prev_dv, current_dv : std_logic;

begin  -- cdc

  edge_find: process(clk, dv, current_dv)
  begin
    if rising_edge(clk) then
      prev_dv <= current_dv;
      current_dv <= dv;
    end if;
  end process edge_find;
  valid_data <= not(prev_dv) and current_dv;

  state_clk: process(clk, reset)
  begin
    if rising_edge(clk) then
      if reset = '1' then
        present_state <= start;
      else
        present_state <= next_state;
      end if;
    end if;
  end process state_clk;


  state_comb: process(present_state, next_state, crs,
                      valid_data, ethernet_buffer, data)
  begin
    case present_state is
      when start =>
        crs <= '0';
        next_state <= idle;

      when idle =>
```

```
        if valid_data = '1' then
          next_state <= read_write1;
          crs <= '1';
        else
          next_state <= idle;
          crs <= '0';
        end if;

      when read_write1 =>
        next_state <= write2;
        crs <= '1';
        rx_data <= data(15 downto 12);
        ethernet_buffer <= data(11 downto 0);

      when write2 =>
        next_state <= write3;
        crs <= '1';
        rx_data <= ethernet_buffer(11 downto 8);

      when write3 =>
        next_state <= write4;
        crs <= '1';
        rx_data <= ethernet_buffer(7 downto 4);

      when write4 =>
        crs <= '1';
        rx_data <= ethernet_buffer(3 downto 0);
        if valid_data = '1' then
          next_state <= read_write1;
        else
          next_state <= idle;
        end if;
    end case;
  end process state_comb;

    rx_error <= '0';

end cdc;
```

C Transceiver VHDL Code (Jen Selby)

C.1 Audio Sample Clock

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity audio_timer is port (
   clk, go        : in std_logic;
   counter        : buffer std_logic_vector(8 downto 0);
   samp           : out std_logic);

end audio_timer;

--this does not give exactly 8192 samples a second
--it signals every 305 clock signals on a 2.5MHz clock, which is close

--compiled for the 20V8 chip

architecture behavioral of audio_timer is

   begin
   counting:process (clk, go)
     begin
       if (clk'event and clk = '1') then
          if go = '1' then
             if counter = "100110001" then
                counter <= "000000000";
                samp <= '1';
             else
                counter <= counter + "00000001";
                samp <= '0';
             end if;
          else
             samp <= '0';
          end if;
       end if;
   end process counting;
end architecture behavioral;
```

C.2 Device Controller

```
library ieee;
use ieee.std_logic_1164.all;
```

```
use work.std_arith.all;

entity controller is port(
   clk          : in std_logic;
   samp         : in std_logic;
   reset        : in std_logic;
   crc_done : in std_logic;
   collision    : in std_logic;
   status       : in std_logic;
   debugging    : out std_logic_vector(2 downto 0);
   audio_cs     : out std_logic;
   audio_read   : out std_logic;
   ram_prom     : out std_logic;
   ram_write    : out std_logic;
   ram_output   : out std_logic;
   start_crc    : out std_logic;
   send_crc     : out std_logic;
   mac_count    : out std_logic;
   aud_count    : out std_logic;
   tx_enable    : out std_logic;
   reset_mac    : out std_logic);

attribute pin_avoid of controller:entity is
  " 5 6 7 8 9 10 15 16 18 67 68 69 " &
--  " 78 " &
  " 30 31 33 34 36 37 38 39 " &
  " 40 45 46 47 48 52 54 56 57 " & -- pins used by other devices
  " 23 62 65 " & -- clocks
  " 13 " & -- I0-9
  " 14 35 41 51 72 " & -- programmer
  " 71 " & -- ground
  " 12 19 73 "; -- interconnect bus


attribute pin_numbers of controller:entity is
  " ram_write:3 ram_output:4 tx_enable:70 collision:75 " &
  " reset_mac:76 reset:77 samp:80 mac_count:24 ram_prom:25 " &
  " aud_count:26 status:27 audio_cs:28 audio_read:29  " &
  " start_crc:49 send_crc:50 crc_done:58 debugging(0):78 " &
  " debugging(1):79 debugging(2):81 ";

end controller;

architecture state_machine of controller is
```

```
--type StateType is (header_stuff, packeting, audio1, audio2,
--                   idle, begin_state, framecheck);
--signal present, nextstate : StateType;

signal present, nextstate : std_logic_vector(2 downto 0);
signal packet_count : std_logic_vector(6 downto 0);
signal wait_one, packet_interrupted : std_logic;

constant header_stuff : std_logic_vector(2 downto 0) := "000";         -- header_stuff
constant packeting : std_logic_vector(2 downto 0) := "001";          -- packeting
constant audio1 : std_logic_vector(2 downto 0) := "010";         -- audio1
constant audio2 : std_logic_vector(2 downto 0) := "011";         -- audio2
constant idle : std_logic_vector(2 downto 0) := "100";          -- idle
constant begin_state : std_logic_vector(2 downto 0) := "101";         -- begin_state
constant framecheck : std_logic_vector(2 downto 0) := "110";         -- framecheck

begin

state_change:process(present, status, samp, collision, packet_interrupted,
                     packet_count, wait_one, crc_done)
  begin
    case present is

      when begin_state   =>   packet_count <= "0000000";
                              wait_one <= '0';
                              start_crc <= '0';
                              send_crc <= '0';
                              ram_write <= '1';
                              ram_output <= '1';
                              ram_prom <= '0';
                              audio_cs <= '1';
                              audio_read <= '1';
                              mac_count <= '0';
                              aud_count <= '0';
                              reset_mac <= '0';
                              nextstate <= idle;
                              packet_interrupted <= '0';
                              tx_enable <= '0';

      when idle          =>   start_crc <= '0';
                              send_crc <= '0';
                              wait_one <= '0';
                              ram_write <= '1';
```

```
                        ram_output <= '1';
                        audio_cs <= '1';
                        audio_read <= '1';
                        mac_count <= '0';
                        reset_mac <= '0';
                        tx_enable <= '0';
                        if samp = '1' then
                           nextstate <= audio1;
                           aud_count <= '1';
                        elsif packet_interrupted = '1' then
                           packet_interrupted <= '0';
                           nextstate <= header_stuff;
                           mac_count <= '1';
                           start_crc <= '1';
                           aud_count <= '0';
                        elsif packet_count = "0100000" then
                           nextstate <= header_stuff;
                           mac_count <= '1';
                           start_crc <= '1';
                           aud_count <= '0';
                        else
                           nextstate <= idle;
                           aud_count <= '0';
                        end if;

when audio1        =>   audio_cs <= '0';
                        audio_read <= '0';
                        wait_one <= '0';
                        mac_count <= '0';
                        aud_count <= '1';
                        ram_write <= '0';
                        ram_output <= '1';
                        start_crc <= '0';
                        send_crc <= '0';
                        reset_mac <= '0';
                        nextstate <= audio2;

when audio2        =>   audio_cs <= '0';
                        audio_read <= '1';
                        wait_one <= '0';
                        mac_count <= '0';
                        ram_write <= '0';
                        ram_output <= '1';
                        start_crc <= '0';
```

```
                        send_crc <= '0';
                        reset_mac <= '0';
                        if status = '1' then
                           nextstate <= audio2;
                           aud_count <= '1';
                        else
                           nextstate <= idle;
                           aud_count <= '0';
                           packet_count <= packet_count + "0000001";
                        end if;

when header_stuff  =>   audio_cs <= '1';
                        audio_read <= '1';
                        mac_count <= '1';
                        ram_write <= '1';
                        ram_output <= '0';
                        ram_prom <= '0';
                        start_crc <= '0';
                        send_crc <= '0';
                        tx_enable <= '1';
                        if samp = '1' then
                           reset_mac <= '1';
                           nextstate <= audio1;
                           packet_interrupted <= '1';
                           aud_count <= '1';
                        elsif collision = '1' then
                           reset_mac <= '1';
                           nextstate <= idle;
                           packet_interrupted <= '1';
                           aud_count <= '0';
                        elsif packet_count = "0110010" then
                           packet_count <= "0000000";
                           nextstate <= packeting;
                           ram_prom <= '1';
                           reset_mac <= '0';
                           aud_count <= '0';
                        elsif wait_one = '0' then
                           packet_count <= packet_count + "0000001";
                           wait_one <= '1';
                           nextstate <= header_stuff;
                           reset_mac <= '0';
                           ram_prom <= '0';
                           aud_count <= '0';
                        else
```

```
                        wait_one <= '0';
                        nextstate <= header_stuff;
                        reset_mac <= '0';
                        aud_count <= '0';
                     end if;

when packeting     =>   audio_cs <= '1';
                        audio_read <= '1';
                        mac_count <= '1';
                        ram_write <= '1';
                        ram_output <= '0';
                        tx_enable <= '1';
                        start_crc <= '0';
                        ram_prom <= '1';
                        if samp = '1' then
                           reset_mac <= '1';
                           nextstate <= audio1;
                           packet_interrupted <= '1';
                           aud_count <= '1';
                           send_crc <= '0';
                        elsif collision = '1' then
                           reset_mac <= '1';
                           nextstate <= idle;
                           packet_interrupted <= '1';
                           aud_count <= '0';
                           send_crc <= '0';
                        elsif packet_count = "0011111" then
                           packet_count <= "0100000";
                           nextstate <= framecheck;
                           send_crc <= '1';
                           reset_mac <= '0';
                           aud_count <= '0';
                        elsif wait_one = '0' then
                           packet_count <= packet_count + "0000001";
                           wait_one <= '1';
                           nextstate <= packeting;
                           reset_mac <= '0';
                           aud_count <= '0';
                           send_crc <= '0';
                        else
                           wait_one <= '0';
                           nextstate <= packeting;
                           reset_mac <= '0';
                           aud_count <= '0';
```

```
                              send_crc <= '0';
                           end if;

   when framecheck    =>   send_crc <= '0';
                           mac_count <= '0';
                           audio_cs <= '1';
                           audio_read <= '1';
                           ram_write <= '1';
                           ram_output <= '0';
                           start_crc <= '0';
                           if crc_done = '1' then
                              nextstate <= idle;
                              tx_enable <= '0';
                              aud_count <= '0';
                           elsif samp = '1' then
                              reset_mac <= '1';
                              nextstate <= audio1;
                              packet_interrupted <= '1';
                              aud_count <= '1';
                           elsif collision = '1' then
                              reset_mac <= '1';
                              nextstate <= idle;
                              packet_interrupted <= '1';
                              aud_count <= '0';
                           else
                              nextstate <= framecheck;
                              tx_enable <= '1';
                              aud_count <= '0';
                           end if;
   when others         =>  nextstate <= begin_state;
   end case;
end process state_change;

state_clocked:process(clk, reset, nextstate)
  begin
    if rising_edge(clk) then
       if reset = '1' then
          present <= begin_state;
       else
          present <= nextstate;
       end if;
    end if;
    debugging <= packet_count(2 downto 0);
    end process state_clocked;
```

```
end architecture state_machine;
```

C.3 Counter / Addresser

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity counter is port(
   clk : in std_logic;
   mac_count : in std_logic;
   aud_count : in std_logic;
   reset : in std_logic;
   reset_mac : in std_logic;
   ram_prom : in std_logic;
   switch_source : in std_logic;
   ram_address : buffer std_logic_vector(12 downto 0));

attribute pin_numbers of counter:entity is
   " mac_count:24 ram_prom:25 aud_count:26 reset_mac:76 reset:77" &
   " ram_address(0):30 ram_address(1):31 ram_address(2):33 " &
   " ram_address(3):34 ram_address(4):36 ram_address(5):37 " &
   " ram_address(6):38 ram_address(7):39 ram_address(8):40 " &
   " ram_address(9):45 ram_address(10):46 ram_address(11):47" &
   " ram_address(12):48 switch_source:78 ";

attribute pin_avoid of counter:entity is
   " 3 4 5 6 7 8 9 10 15 16 17 18 67 68 69 70 75 80 " &
   " 27 28 29 49 50 52 54 56 57 58 " & -- pins used by other devices
   " 23 62 65 " & -- clocks
   " 13 " & -- IO-9
   " 14 35 41 51 72 " & -- programmer
   " 71 " & -- ground
   " 12 19 73 "; -- interconnect bus



end counter;

architecture behavioral of counter is
```

```
signal mac_address : std_logic_vector(12 downto 0);
signal starting_mac : std_logic_vector(12 downto 0);
signal aud_address : std_logic_vector(12 downto 0);
signal counting_mac, just_reset : std_logic;
signal wait_one : std_logic;
signal prev_aud : std_logic;

begin

addressing:process (clk, mac_count, aud_count, reset, reset_mac,
                    mac_address, aud_address, counting_mac)
  begin
    if (clk'event and clk = '1') then
       if reset = '1' then
          aud_address <= "0000000000000";
          mac_address <= "0000000000000";
          counting_mac <= '0';
          just_reset <= '1';
          prev_aud <= '0';
       elsif aud_count = '1' then
          just_reset <= '0';
          prev_aud <= '1';
          ram_address <= aud_address;
          counting_mac <= '0';
          if prev_aud = '0' then
             if aud_address = "1111111111111" then
                aud_address <= "0000000000000";
             else
                aud_address <= aud_address + "0000000000001";
             end if;
             if (mac_address = aud_address and just_reset = '0') then
                mac_address <= mac_address + "0000000100000";
             else
                mac_address <= mac_address;
             end if;
          end if;
       elsif reset_mac = '1' then
          prev_aud <= '0';
          just_reset <= '0';
          mac_address <= starting_mac;
          counting_mac <= '0';
       elsif mac_count = '1' then
          prev_aud <= '0';
```

```
            just_reset <= '0';
            counting_mac <= '1';
            if wait_one = '0' then
               wait_one <= '1';
               if ram_prom = '0' then
                  if counting_mac = '0' then
                     if switch_source = '0' then
                        ram_address <= "0000000000000";
                     else
                        ram_address <= "0000000110011";
                     end if;
                  else
                     ram_address <= ram_address + 1;
                  end if;
               else
                  if counting_mac = '0' then
                     starting_mac <= mac_address;
                     counting_mac <= '1';
                     if mac_address = "1111111111111" then
                        mac_address <= "0000000000000";
                     else
                        mac_address <= mac_address + "0000000000001";
                     end if;
                  elsif mac_address = "1111111111111" then
                     mac_address <= "0000000000000";
                  else
                     mac_address <= mac_address + "0000000000001";
                  end if;
                   ram_address <= mac_address;
               end if;
            else
               wait_one <= '0';
            end if;
         else
            counting_mac <= '0';
            prev_aud <= '0';
         end if;
      end if;
end process addressing;
end architecture behavioral;
```

C.4 Cyclical Redundancy Check Calculator

```
library ieee;
```

```
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity crc is port (
   clk : in std_logic;
   start_crc : in std_logic;
   send_crc : in std_logic;
   tx_data : in std_logic_vector(3 downto 0);
   crc_tx_data : out std_logic_vector(3 downto 0);
   crc_done : out std_logic);

attribute pin_avoid of crc:entity is
  " 3 4 5 6 7 8 9 10 15 16 70 75 76 77 78 80 24 25 26 27 28 29 30 31 " &
  " 33 34 36 37 38 39 40 45 46 47 48 " & -- pins used by other devices
  " 23 62 65 " & -- clocks
  " 13 " & -- IO-9
  " 14 35 41 51 72 " & -- programmer
  " 71 " & -- ground
  " 12 19 73 "; -- interconnect bus


attribute pin_numbers of crc:entity is
  " tx_data(0):18 tx_data(1):67 tx_data(2):68 tx_data(3):69 " &
  " start_crc:49 send_crc:50 crc_tx_data(0):52 crc_tx_data(1):54 " &
  " crc_tx_data(2):56 crc_tx_data(3):57 crc_done:58 ";

end crc;

architecture behavioral of crc is

signal checksum : std_logic_vector(31 downto 0);
signal count8 : std_logic_vector(2 downto 0);
signal data : std_logic_vector(3 downto 0);
signal computing : std_logic;
signal sending : std_logic;

begin

mode:process(clk, start_crc, send_crc, tx_data, count8,
                  computing, data, checksum)
  begin
    if (clk'event and clk = '1') then
       if start_crc = '1' then
           computing <= '1';
```

```
            sending <= '0';
        elsif send_crc = '1' then
            sending <= '1';
            computing <= '0';
        end if;
    end if;
  end process mode;

compute:process(clk, start_crc, send_crc, tx_data, count8,
                  computing, data, checksum)
  begin
    if (clk'event and clk = '1') then
       if (computing = '1' or start_crc = '1') then
          crc_tx_data <= tx_data;
          crc_done <= '0';
          checksum(31) <= checksum(27);
          checksum(30) <= checksum(26);
          checksum(29) <= checksum(25) xor checksum(31);
          checksum(28) <= checksum(24) xor checksum(31);
          checksum(27) <= checksum(23) xor checksum(29);
          checksum(26) <= (checksum(22) xor checksum(31)) xor checksum(28);
          checksum(25) <= (checksum(21) xor checksum(31)) xor checksum(30);
          checksum(24) <= (checksum(20) xor checksum(30)) xor checksum(29);
          checksum(23) <= (checksum(19) xor checksum(20)) xor checksum(28);
          checksum(22) <= checksum(18) xor checksum(28);
          checksum(21) <= checksum(17);
          checksum(20) <= checksum(16);
          checksum(19) <= checksum(15) xor checksum(31);
          checksum(18) <= checksum(14) xor checksum(30);
          checksum(17) <= checksum(13) xor checksum(29);
          checksum(16) <= checksum(12) xor checksum(28);
          checksum(15) <= checksum(11) xor checksum(31);
          checksum(14) <= (checksum(10) xor checksum(31)) xor checksum(30);
          checksum(13) <= ((checksum(9) xor checksum(31)) xor checksum(30))
                          xor checksum(29);
          checksum(12) <= ((checksum(8) xor checksum(30)) xor checksum(29))
                          xor checksum(28);
          checksum(11) <= ((checksum(7) xor checksum(31)) xor checksum(29))
                          xor checksum(28);
          checksum(10) <= ((checksum(6) xor checksum(31)) xor checksum(30))
                          xor checksum(28);
          checksum(9)  <= (checksum(5) xor checksum(30)) xor checksum(29);
          checksum(8)  <= (checksum(4) xor checksum(31) xor checksum(29))
                          xor checksum(28);
```

```
            checksum(7)  <= ((checksum(3) xor checksum(31)) xor checksum(30))
                           xor checksum(28);
            checksum(6)  <= (checksum(2) xor checksum(30)) xor checksum(29);
            checksum(5)  <= ((checksum(1) xor checksum(31)) xor checksum(29))
                           xor checksum(28);
            checksum(4)  <= ((checksum(0) xor checksum(31)) xor checksum(30))
                           xor checksum(28);
            checksum(3)  <= ((tx_data(0) xor checksum(31)) xor checksum(30))
                           xor checksum(29);
            checksum(2)  <= ((tx_data(1) xor checksum(30)) xor checksum(29))
                           xor checksum(28);
            checksum(1)  <= (tx_data(2) xor checksum(29)) xor checksum(28);
            checksum(0)  <= tx_data(3) xor checksum(28);
         elsif (sending = '1' or send_crc = '1') then
            if count8 = "111" then
               crc_done <= '1';
               count8 <= "000";
               checksum <= "11111111111111111111111111111111";
            else
               count8 <= count8 + "001";
               crc_done <= '0';
            end if;
            if count8 = "000" then
               crc_tx_data <= checksum(31 downto 28);
            elsif count8 = "001" then
               crc_tx_data <= checksum(27 downto 24);
            elsif count8 = "010" then
               crc_tx_data <= checksum(23 downto 20);
            elsif count8 = "011" then
               crc_tx_data <= checksum(19 downto 16);
            elsif count8 = "100" then
               crc_tx_data <= checksum(15 downto 12);
            elsif count8 = "101" then
               crc_tx_data <= checksum(11 downto 8);
            elsif count8 = "110" then
               crc_tx_data <= checksum(7 downto 4);
            elsif count8 = "111" then
               crc_tx_data <= checksum(3 downto 0);
            end if;
         end if;
      end if;

end process compute;
```

```
end architecture behavioral;
```

C.5 Media Access Controller

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity mac is port(
   clk      :  in std_logic;
   ram_data :  in std_logic_vector(7 downto 0);
   mac_count:  in std_logic;
   clk_out  :  out std_logic;
   tx_data  :  out std_logic_vector(3 downto 0));

attribute pin_avoid of mac:entity is
  " 3 4 70 75 76 77 78 80 25 26 27 28 29 30 31 33 34 36 37 38 39 40 45 46 " &
  " 47 48 49 50 52 54 56 57 " & --pins used by other devices
  " 23 62 65 " & -- clocks
  " 14 35 41 51 72 " & -- programmer
  " 71 " & -- ground
  " 12 19 73 "; -- interconnect bus

attribute pin_numbers of mac:entity is
  " ram_data(0):5 ram_data(1):6 ram_data(2):7 ram_data(3):8 " &
  " ram_data(4):9 ram_data(5):10 ram_data(6):15 ram_data(7):16 " &
  " tx_data(0):18 tx_data(1):67 tx_data(2):68 tx_data(3):69 " &
  " mac_count:24 clk_out:13";

end mac;


architecture behavioral of mac is

signal wait_one, tx_data_enable : std_logic;
signal prev_mac_count : std_logic;
signal actual_tx_data : std_logic_vector(3 downto 0);

begin

clocked:process(clk, mac_count)
  begin
    if (clk'event and clk = '1') then
       if mac_count = '1' then
```

```
          prev_mac_count <= '1';
          if (prev_mac_count = '0' or wait_one <= '0') then
             tx_data <= ram_data(3 downto 0);
             wait_one <= '1';
          else
             tx_data <= ram_data(7 downto 4);
             wait_one <= '0';
          end if;
       else
         prev_mac_count <= '0';
       end if;
    end if;
end process clocked;

clk_out <= clk;


end architecture behavioral;
```

C.6 Prom Specification File

```
op <7:0>;
address op <6:0>;
value op <7:0>;

ZEROS op<7:0>=0;

PREAMBLE op<7:0>=%b10101010;
STARTFRAME op<7:0>=%b10101011;

MAC1 op<7:0>=13;
MAC2 op<7:0>=4;
MAC3 op<7:0>=21;

LENGTH op<7:0>=92;

IPVERSION op<3:0>=4;
HDRLENGTH op<7:4>=5;

TOTALLNGTH1 op<7:0>=92;

IDNUM op<7:0>=13;

FRAGMENTS op<7:0>=%b00000010;
```

```
TIMETOLIVE op<7:0>=%b11111111;

UDPPROT op<7:0>=17;

IPCHECKSUM1 op<7:0>=%b10111011;
IPCHECKSUM2 op<7:0>=%b00111001;

ADDR1 op<7:0>=10;
SRCADDR op<7:0>=51;
WRONGDESTADDR op<7:0>=52;
DESTADDR op<7:0>=50;

PORT1 op<7:0>=%b00001000;
PORT2 op<7:0>=%b01010010;

UDPLENGTH op<7:0>=72;
```