A Versatile Ethernet Interface

Daniel S. Whelan

Computer Science Department
California Institute of Technology

4654:TR:81

CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science Department

Technical Report #4654

A VERSATILE ETHERNET INTERFACE

by

Daniel S. Whelan

In Partial Fulfillment of the requirements for the

Degree of Master of Science

May, 1981

The research described in this report was sponsored by the Defense Advanced Research Project Agency, ARPA Order number 3771, and monitored by the Office of Naval Research under contract number N00014-79-C-0597.

CHAPTER 1

INTRODUCTION

In the later part of 1979, it appeared that the Ethernet [Metcalf and Boggs, 1976] was one of the most viable local networks. The presense of several Altos on campus suggested that we connect other machines up to the Ethernet and provide the department with up to date local networking capabilities. Since several types of machines were to be connected to the Ethernet, it was decided to design one interface with enough flexibility to interface all of these machines to the net. This thesis describes the implementation of such an Ethernet interface and the architectural issues that were at play.

Some general knowledge of what Ethernet is and how well it performs as a local network should be useful to the reader. First, Ethernet can be traced as an descendant of the Aloha net [Abramson, 1970] which was an attempt at providing inexpensive local networking capabilities. The Aloha network and the Ethernet took the attitude that bandwidth is relatively inexpensive when compared with the cost of computer interfaces and therefore the design of the network transport protocol should try to minimize the cost of the computer interface. The result was a very simple algorithm for gaining access to the transmission channel. In the ALOHA nets case, transmitters transmitted packets when they needed to. Of course the possibility of collisions between two or more transmitters existed so the system required a transmitter to receive a postive acknowledgement within a certain time window or else the packet would have to be retransmitted. This simple technique worked quite well. The theoretical maximum channel utilization for such a system worked out to be 18.6%.

Ethernet came about as some simple changes to the ALOHA transmission strategy. First, transmitters don't transmit whenever they want to, they must first listen to the transmission channel and defer transmission until the channel is quiet. Secondly, transmitters must watch the transmission channel during packet transmission and if they notice that they are colliding with someone else, they must jam the net for some time so that every receiver notices that there was a collision and then they must get off the channel. After a collision, the transmitter may try retransmitting the packet and does so by tossing a random number and weighting it by an estimate of the network load and waiting that many clock ticks before trying to retransmit the packet. These simple changes provide Ethernet with channel efficiencies above 90%.

Of more importance than theoretical channel efficiencies is observed network performance. A performance study was performed on an Ethernet at Xerox [Schoch and Hupp, 1979]. Their study showed that even under high loads the network exhibited channel utilization above 97%. Furthermore, 99.18% of the packets were transmitted without having to defer. Less than a percent of the transmitted packets were delayed due to deference while 0.03% of the transmission attempts resulted in collisions. The statistics revealed by this paper were used to direct the design of this interface. For example, since packets are rarely involved in collisions, the retransmission of packets is handled by interrupt software and not by dedicated hardware as might have been the case had collisions been frequent events.

CHAPTER 2

ARCHITECTUAL ISSUES

2.1 Introduction

This chapter discusses the major design issues that come into play when building a general purpose Ethernet interface. A modular system architecture is developed to help define the design problem. Overall system performance is analyzed by comparing and contrasting plausible intermodule communications strategies. Bit serialization and synchronization are touched upon in an effort to understand what clock speed requirements will have to be met.

2.2 A General Architecture

Architectures serve to describe a set of solutions to a particular problem. The problem that we are dealing with is one of translating data in an unspecified protocol into Ethernet data packets. We must also be able to do the inverse translation. The solution to this problem will be a black box that connects up to an Ethernet transceiver on one side and allows the user to connect up any desired hardware protocol module on the other side. While this "unspecified" or "undefined" protocol is not known to us, for implementation purposes, it is reasonable for us to place an upper limit on its bandwidth. For the purpose of our discussions, we will require that this system be able to handle one megabyte/second burst transfer rates. This rate will accommodate a wide range of existing protocols such as UNIBUS, Q-BUS and HPIB to name a few.

Modularity is a useful design construct. It allows large problems to be subdivided into smaller tasks. It appears to be a necessary prerequisite for hierarchical decomposition of the design task. Fortunately, the network interfaces can be modularized very nicely. It makes sense to speak of Ethernet transmission modules and Ethernet receiver modules because the two tasks are fairly independent. Lets also assume that the "undefined protocol" can be implemented as two separate modules. If we can devise a way of hooking these four protocol modules together in a meaningful manner we will have a solution to our problem and thus a viable architecture. The definition of other modules may be required in the process.

In general some sort of transformation will have to be performed upon the data arriving via one protocol port before it can be transmitted via the other. Unfortunately, we have no way of knowing what kind of transformations will have to take place. Whatever these transformations look like, they can be decomposed into two transformations, one that converts data in a first protocol into an internal format and another that converts the internal format into the end protocol. Transformations may be history or even future dependent operations and thus necessitate the addition of buffer memory to our system. Buffer memory is necessary anyway because of possible transmission speed mismatches between the Ethernet protocol and the "undefined" protocol.

So far, we have developed an architecture that is perfectly general in nature. It consists of protocol modules, transformation modules and memory and is shown in Figure 2.1. One can actually envision implementing a pipelined system of this type if the transformations T1 and T1′ were known. Unfortunately, these transformations are not known. Building an implementation of Figure 2.1 for an unknown transformation implies that the transformation modules must be programmable. Once we accept this fact, we realize that an architecture like the one illustrated in Figure 2.2 is as general as the one in Figure 2.1.

This new architecture uses a microprocessor to provide the required programmable transformation capabilities. It also uses the same microprocessor as a system controller to coordinate the actions of the other modules. Although a single microprocessor may be slower at performing the transformations than four specialized hardware modules, it doesn′t reduce the burst transfer performance. In both models the burst rate is memory access limited at about six megabytes/second. This organization may hinder the system throughput more than a pipelined approach and will be studied more in more detail in the following section.

A block diagram like Figure 2.2, indicates that the modules can be described by a certain functional behavior. The lines between modules denote intermodule communications. Therefore the operation of the system as a whole depends on the implementation of communications strategies as well as the implementation of the modules.

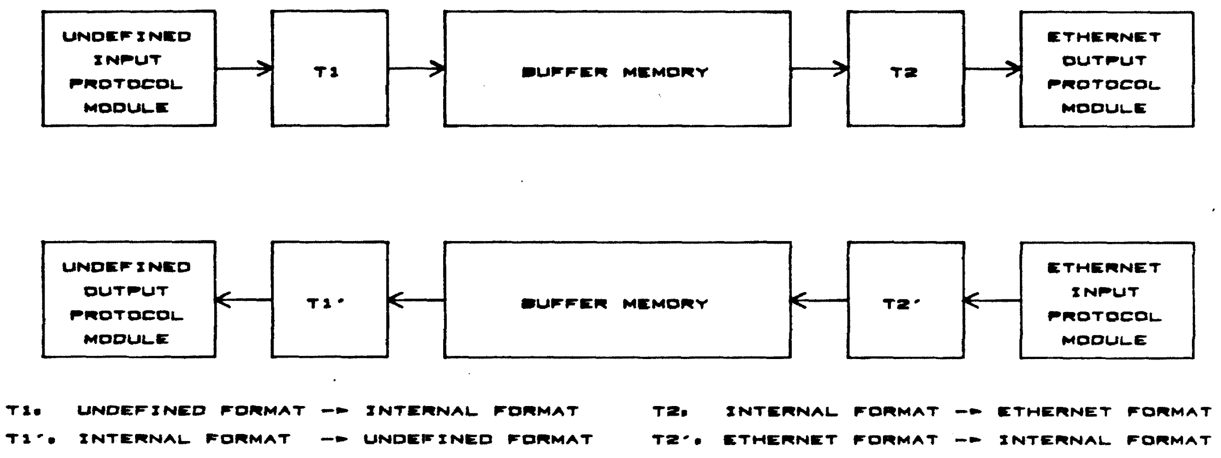

UNDEFINED INPUT PROTOCOL MODULE
T1
BUFFER MEMORY
T2
ETHERNET OUTPUT PROTOCOL MODULE
UNDEFINED OUTPUT PROTOCOL MODULE
T1'
BUFFER MEMORY
T2'
ETHERNET INPUT PROTOCOL MODULE
T1: UNDEFINED FORMAT -> INTERNAL FORMAT
T1': INTERNAL FORMAT -> UNDEFINED FORMAT
T2: INTERNAL FORMAT -> ETHERNET FORMAT
T2': ETHERNET FORMAT -> INTERNAL FORMAT

Figure 2.1: Generalized Architecture

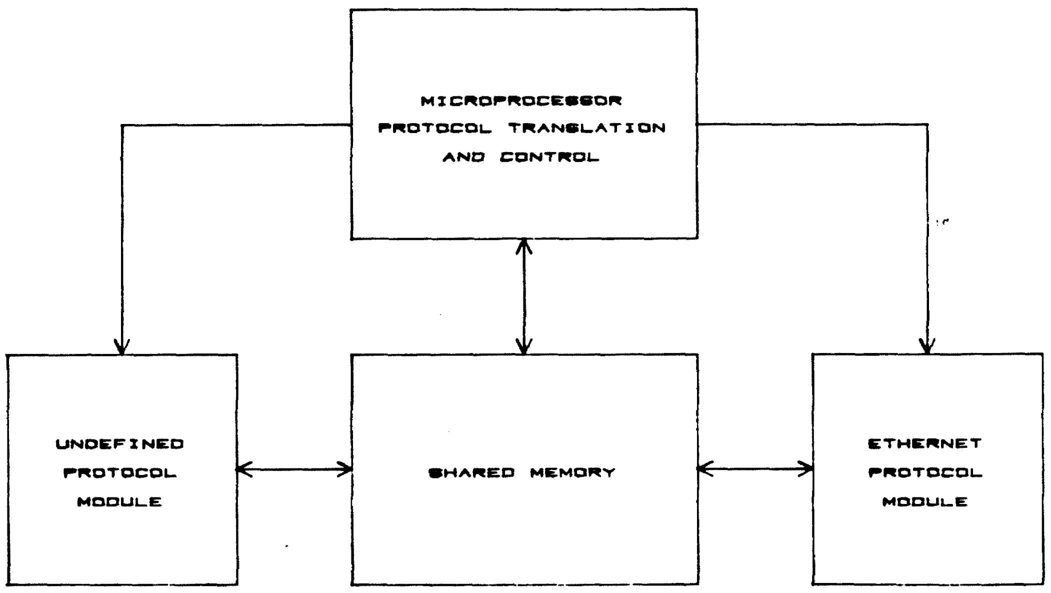

MICROPROCESSOR PROTOCOL TRANSLATION AND CONTROL
UNDEFINED PROTOCOL MODULE
SHARED MEMORY
ETHERNET PROTOCOL MODULE

FIGURE 2.2: AN IMPLEMENTABLE ARCHITECTURE

2.3 Communications Strategies

Given the current architecture of Figure 2.2, there are several ways to provide intermodule communications. Any particular technique can be characterized by a cost in parts and dollars and also by the maximum throughput that it allows the system to achieve. Higher throughputs mean that the controller can get more done in a fixed amount of time and therefore seem generally desirable but it may not mean anything. For instance, if we could build an Ethernet interface that could send packets back to back with zero dead time but we would be deceiving ourselves because the nature of Ethernet is that a user tends to be able to acheive an average share of the nets bandwidth and not monopolize it. High throughput through the interface does however allow higher peak throughputs than slower interfaces can acheive.

Several communications architectures for building Ethernet interfaces will be analyzed on the basis of maximum achievable throughput and implementation costs. Although these strategies will be presented for Ethernet interfaces, the same methods can be used for designing interfaces for other protocols. Generally the throughput analysis holds for any protocol with appropiate parameter adjustments.

2.3.1 Fixed Address Memory Mapped Buffers

This strategy, illustrated in Figure 2.3 was originally presented as the "cheap and simple" approach to building Ethernet interfaces. The interface looks like two fixed address memory mapped buffers plus some memory mapped control and status registers. The packet to be sent is either formatted in the transmission buffer or copied into it. After the packet is in place, a few hardware registers are written to initiate transmission of the packet. Since buffer memory is shared between the processor and the Ethernet transmitter, it can not be used by the

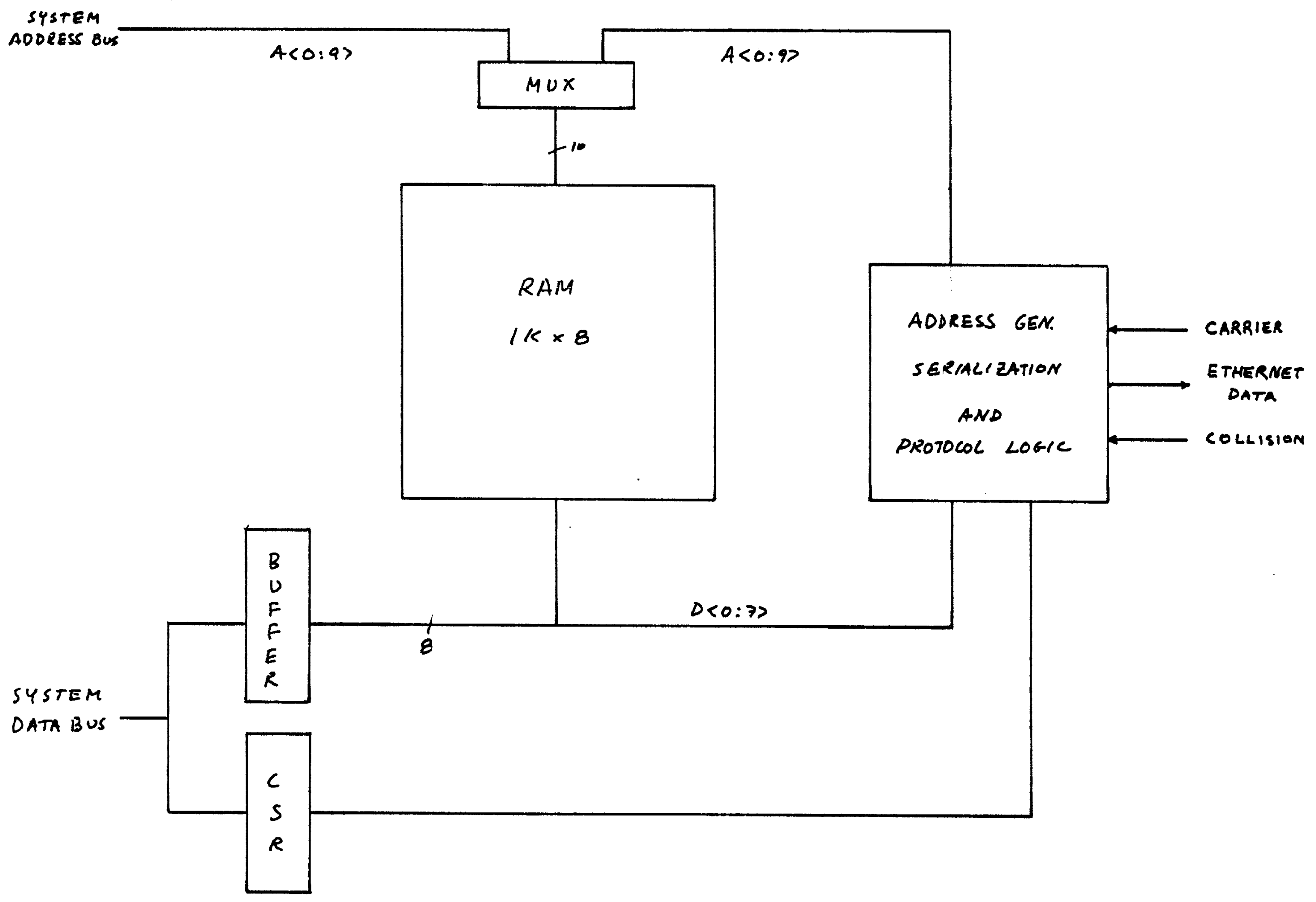


FIGURE 2.3: FIXED ADDRESS BUFFER

processor during transmission period. The size of the buffer also determines the maximum size packet that can be transmitted. When the hardware has finished transmitting the packet, the processor is notified of whether or not the transmission was successful and then is free to once again use the transmission buffer.

Reception is similar in nature. The Ethernet receiver notifies the processor when it has received a packet, which the processor then accesses in the reception buffer memory. Normally, the processor will remove the received packet from the reception buffer before notifying the hardware that it can receive another packet. Because it takes a considerable amount of time to remove the packet from the reception buffer, the Ethernet node can not receive another packet during a relatively large window following a packet reception. This dead period may have an adverse affect on the throughput of an Ethernet system. Most studies have determined Ethernet performance based on the temporal restraints imposed by the transmission protocol and have assumed that when packets made it onto the net, they would be received. This assumption may not hold when the receiver's dead time is considered.

2.3.1.1 Throughput And Cost Analysis

The maximum achievable throughput for the former type of communications stategy occurs when the packet is already assembled some place else in the processor's memory. In this case, a packet transmission time is the time required to move the packet into the fixed address buffer plus a constant overhead time for setting up registers plus the actual transmission time. Thus throughput is:

Equation 2.1:

$$\text{Throughput} = \frac{1}{\text{overhead} + \text{packet length} * (\text{transfer time} + \text{transmit time})}$$

Where Throughput is in bits/second,
Overhead is in seconds,
Packet Length is in bits,
Transfer and Transmit Times are in seconds/bit

The implementation costs for this type of structure are primarily due to address and data bus multiplexers required in order to map and unmap the memory buffer in the processors address space. A typical buffer might be arranged as 1K by 8 bits and would require ten address lines, eight bi-directional data lines and two control lines be multiplexed. A conservative estimate would require at least three twenty pin packages to implement this multiplexing. The Ethernet protocol module will also require some address generation circuitry which could be provided with a ten bit address counter and a ten bit comparator, probably another six packages. Add in several more packages for control logic and we are talking about more than ten MSI packages per buffer to implement this type of communications. This method was described as "cheap and simple", while it is simple, a TTL implementation may not necessarily be cheap. Cheap was originally meant to refer to an LSI implementation.

2.3.2 Processor Accessed FIFO Buffers

Another organization has been proposed which turns out to be identical in performance to the previous method. Instead of mapping transmit and receive buffers into memory space, map transmit and receive FIFOs into the memory space as illustrated in Figure 2.4 Thus to transmit a packet you simply copy the packet into the FIFO. Since the

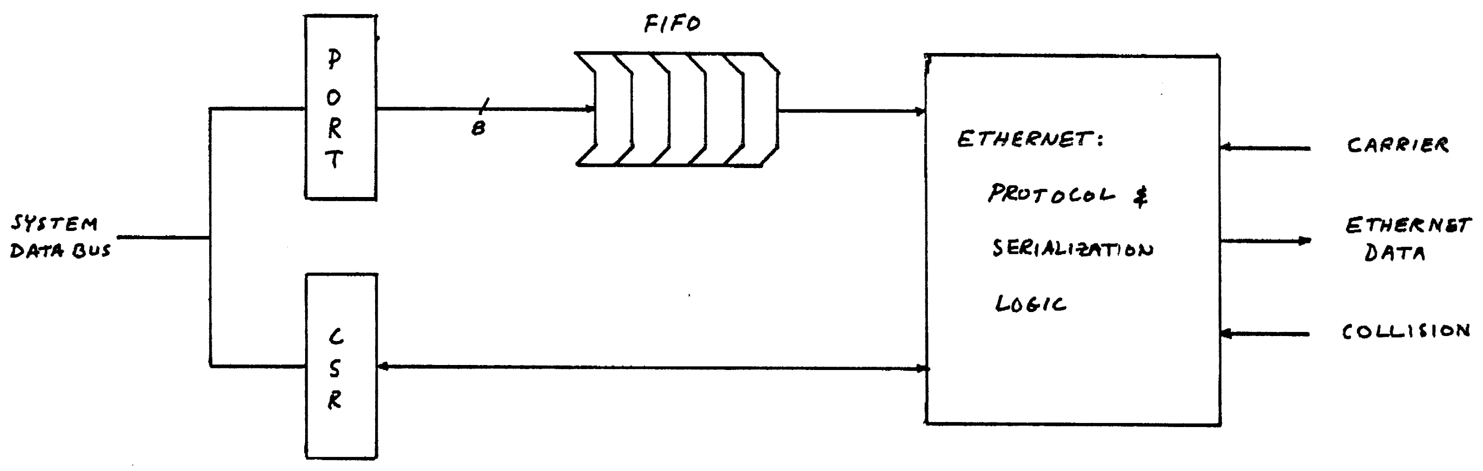


FIGURE 2.4: STRAIGHT FIFO

processor is slow at doing memory transfers into the FIFO relative to the network hardware's capacity to withdraw from the FIFO, the FIFO will have to be as long as the longest packet. With this assumption we see that it takes the same time to transfer a packet as it did before. The packet must first be transferred into the FIFO and there is also some overhead associated with setting up the hardware to do the transmission. Furthermore, the memory mapped fixed address buffers did allow the user to assemble the packet in the transmission buffer if he chose to. A FIFO scheme does not allow the packet to be assembled in the FIFO since a protocol header will usually be prefixed to a packet during assembly and will include a checksum over the data. Since the checksum is in the first few words of the packet, it requires that the whole packet be assembled elsewhere prior to being shoved into a FIFO.

2.3.3 Variable Address Memory Mapped Buffers

Figure 2.5 illustrates a third model for intermodule communications that allows the receive and transmit buffers to reside at variable addresses in the processor's address space. In addition to a register containing the length of the packet, this system requires a register with an address pointer to the packet. This technique generally requires what amounts to a DMA channel to access words in the processor's memory space.

There are at least two ways to build a DMA channel. An overly simplistic approach would be to grab the processors bus for the duration of the transfer (block DMA) whereas a more thoughtful way would be to grab the bus whenever a memory cycle is needed (burst DMA). The main difference is that with burst mode DMA, the processor can get bus cycles in during the transfer period. With block DMA, the processor is dead for the duration of the transfer period.

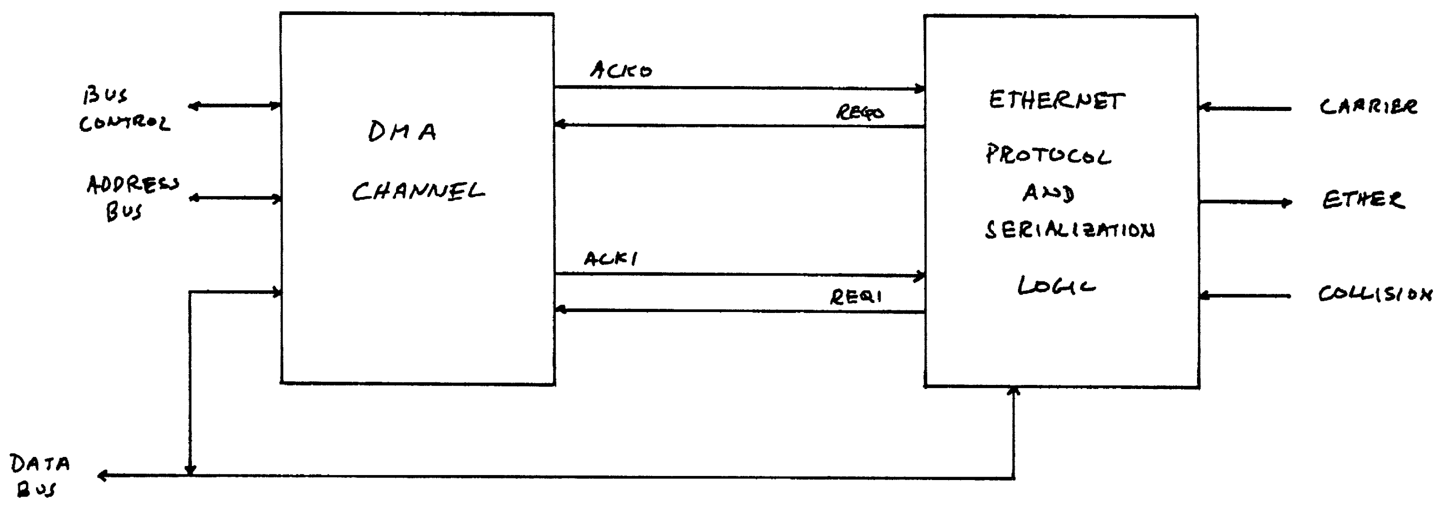


FIGURE 2.5: VARIABLE ADDRESS BUFFERS

2.3.3.1 Throughput And Cost Analysis

Even with block DMA, variable address buffers offer a great improvement over fixed address buffers. For instance, the time it takes to transmit a packet with block DMA is some constant time for setting up registers plus the packet transmission time. A burst mode DMA allows some of the constant overhead time to be processed in parallel with packet transmission but some overhead is still additive.

If we use a block mode DMA transfer, the maximum achievable throughput occurs when the packet is already assembled in memory. In this case, the time it takes to send a packet is a constant overhead for writing hardware registers plus the transmission time. Therefore the throughput is:

Equation 2.2:

$$\text{Throughput} = \frac{1}{\text{overhead time} + (\text{packet length} * \text{transmit time})}$$

If we use a burst mode DMA transfer, the maximum achievable throughput occurs under the same conditions. Under these circumstances, part of the transmission time can be parallel processed with the transmission time. If the DMA channel still allows the processor to see 100% throughput on its system bus, the time it takes to transmit a packet becomes:

Equation 2.3:

```
if (overhead time) < (packet length * transmit time) then
    Throughput = 1/(packet length * transmit time)
else Throughput = 1/(overhead time);
```

The implementation costs of both types of DMA channels are similar. Multichannel DMA controller chips are available from semiconductor manufacturers. These chips require a few peripheral components, usually about six to eight packages of latches and buffers. While one could build a block DMA controller out of discretes, it is not clear that there would be any cost advantage to doing so. It is clear that the parts count would be higher. This discrete controller would almost be equivalent to the fixed address buffer multiplexing and memory access hardware except that it would require a larger address space.

2.3.4 DMA And FIFO Combinations

The fourth approach, illustrated in Figure 2.6 is merely an improvement upon the burst mode DMA method mentioned above. Typically a DMA channel requires some overhead time during which it acquires control of the processor's bus. This time is typically about the same as the time it takes to transfer a word. Therefore if a DMA channel is set up to do burst transfers of single words, there is an 100% overhead associated with each transfer. If a channel can be set up to do 16 word block transfers instead, the overhead is reduced to 6.25%. We can build hardware to work in this manner by having the DMA channel feed a FIFO. When the FIFO reaches a low water mark, a request is made for enough transfers to fill up the FIFO. This technique allows either more processing cycles to happen in parallel with packet transmission or more DMA channels to be simultaneously active on a single bus.

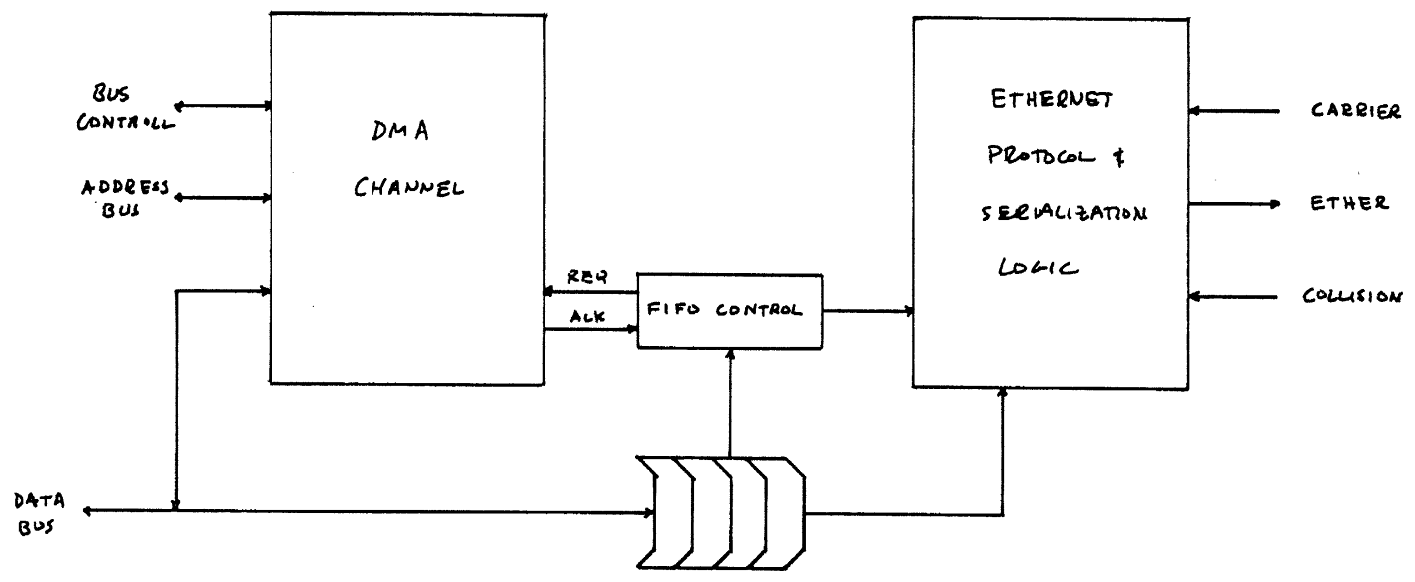


FIGURE 2.6: DMA & FIFO

2.3.5 Performance Comparisons

We have discussed several different architectures and have calculated their maximum achievable throughputs. Most of these equations contain two free variables. The first is an explicit overhead time which is an additive delay in the system. The second variable is the packet length, which determines how long it takes to transmit a packet as well as limiting the number of packets per second that can be transmitted. Since our design introduces the overhead time it makes sense to compare throughputs for a distribution of packets lengths that corresponds with reality. Real experience at Xerox shows that about 80% of the packets are 32 bytes long while about 20% of the packets are about 544 bytes long [Schoch and Hupp, 1979].

Figure 2.7 is a plot of throughputs versus overhead time for three architectures. One curve represents both the fixed address buffers of Section 2.3.1 and the FIFO organization discussed in Section 2.3.2. The two other curves represent the block and burst DMA techniques of implementing the variable address buffers of Section 2.3.3. This Figure shows that the block DMA is far better than the fixed address buffer scheme for reasonable packet overheads. The burst mode DMA is even better than block mode DMA but all three schemes converge when the throughput becomes dominated by the packet overhead. When we consider a uniform distribution of packet lengths, we notice that the DMA methods gain far more than the fixed address method because much more of the data is being transmitted in longer length packets.

Since DEC, Intel and XEROX have recently announced a joint venture to produce a 10MHz Ethernet [DEC, Intel and XEROX, 1980], I thought that it would be interesting to see how the three different methods faired as the clock rate gets turned up. Figures 2.9 and Figures 2.10 show that the rankings of the three techniques stay the same. The most notable effect of the higher bit rate is that the fixed address buffer scheme loses a great deal since its throughput, Equation 2.1, is more dependent

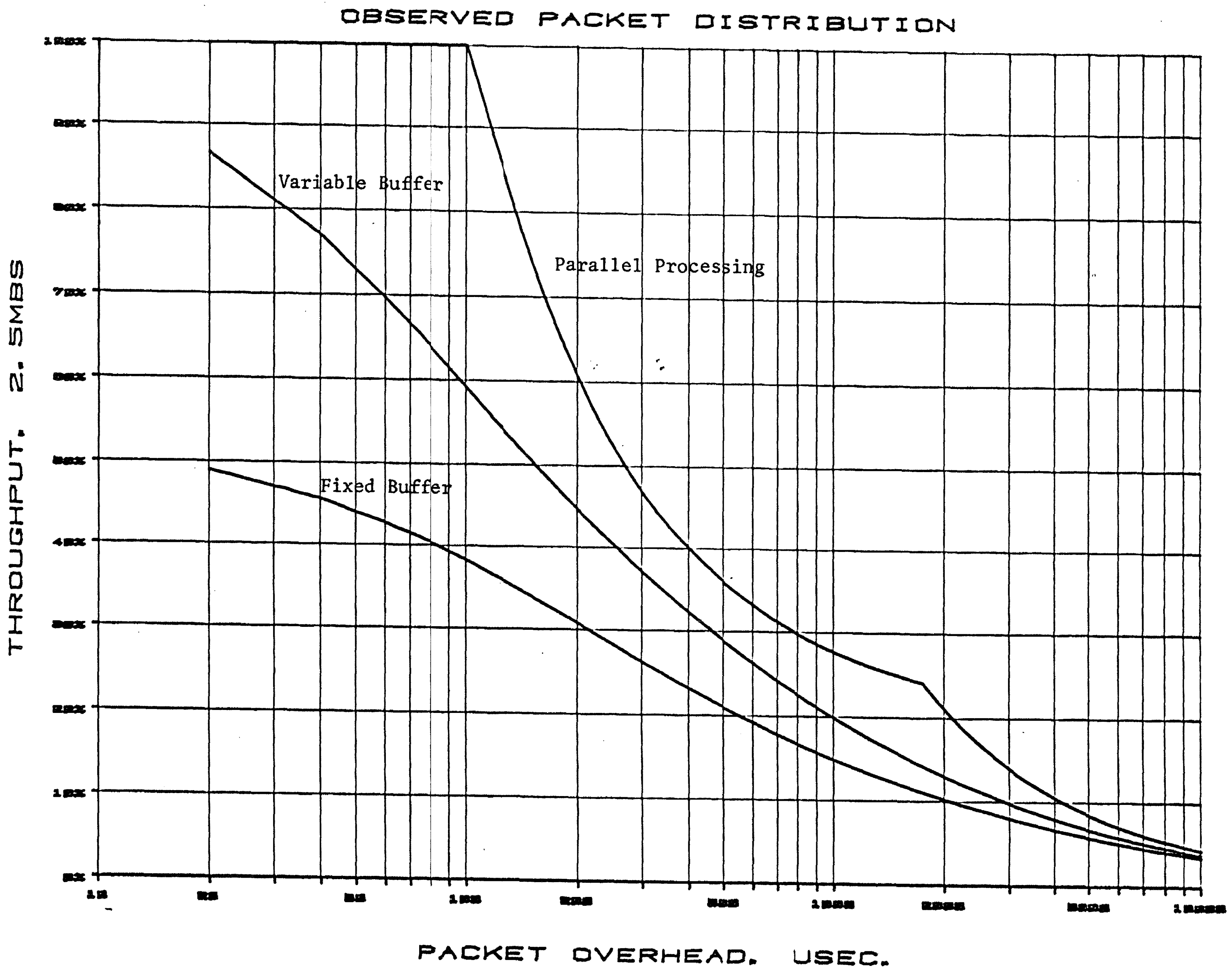

OBSERVED PACKET DISTRIBUTION
THROUGHPUT, 2.5MBS
Variable Buffer
Parallel Processing
Fixed Buffer
PACKET OVERHEAD, USEC.

FIGURE 2.7

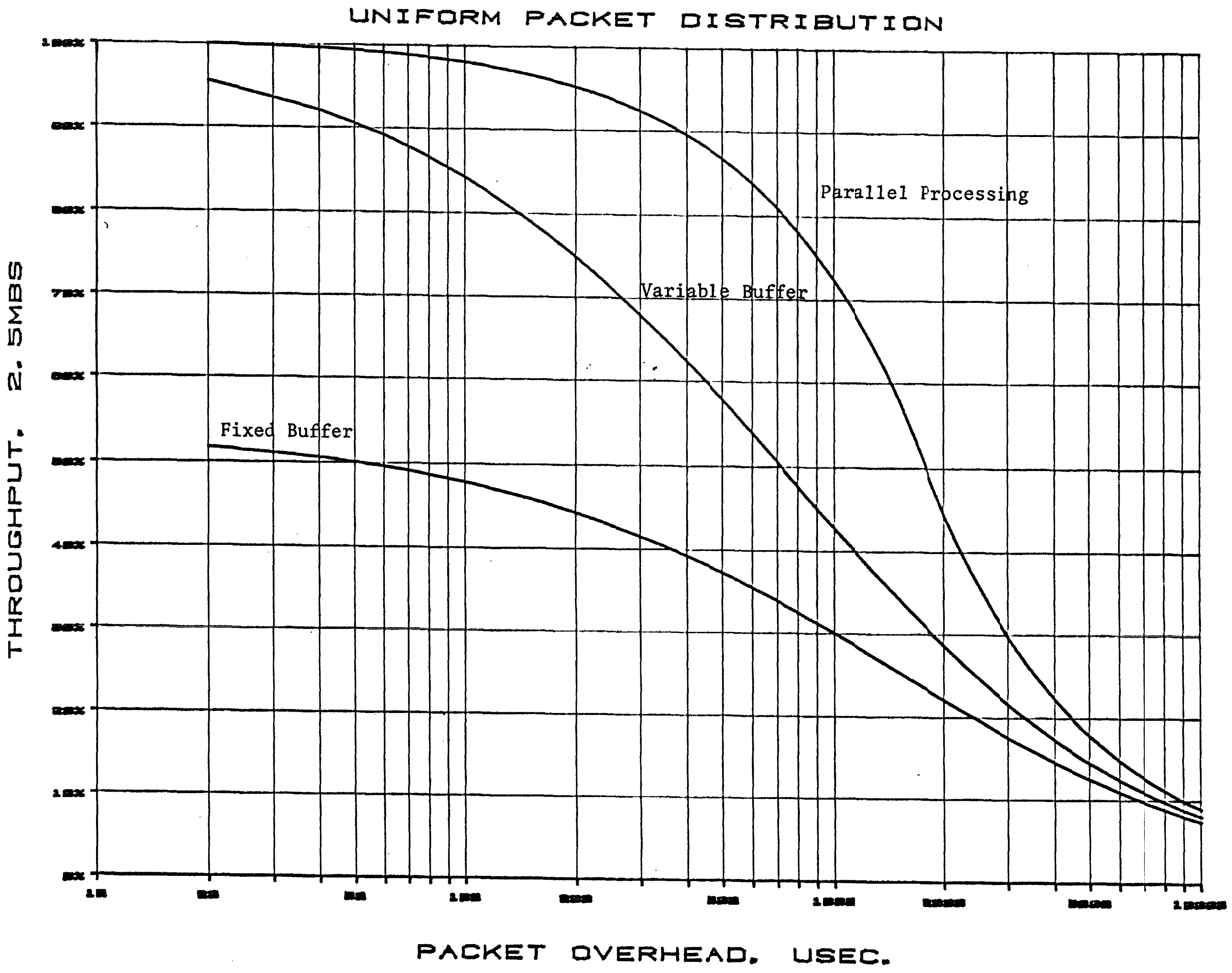

UNIFORM PACKET DISTRIBUTION
THROUGHPUT, 2.5MBS
PACKET OVERHEAD, USEC.
Parallel Processing
Variable Buffer
Fixed Buffer

FIGURE 2.8

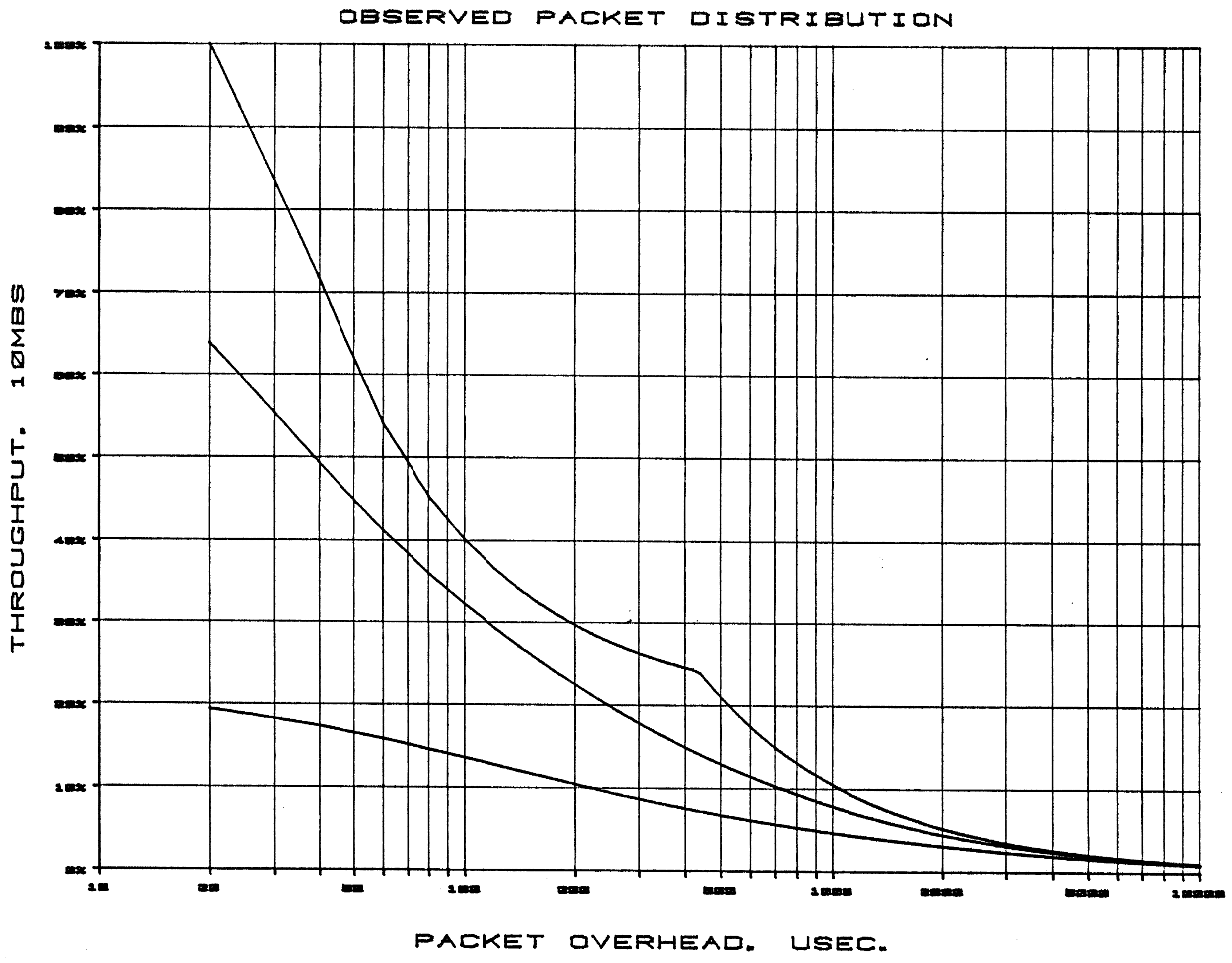

OBSERVED PACKET DISTRIBUTION
THROUGHPUT, 10MBS
PACKET OVERHEAD, USEC.
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
10
20
50
100
200
500
1000
2000
5000
10000

FIGURE 2.9

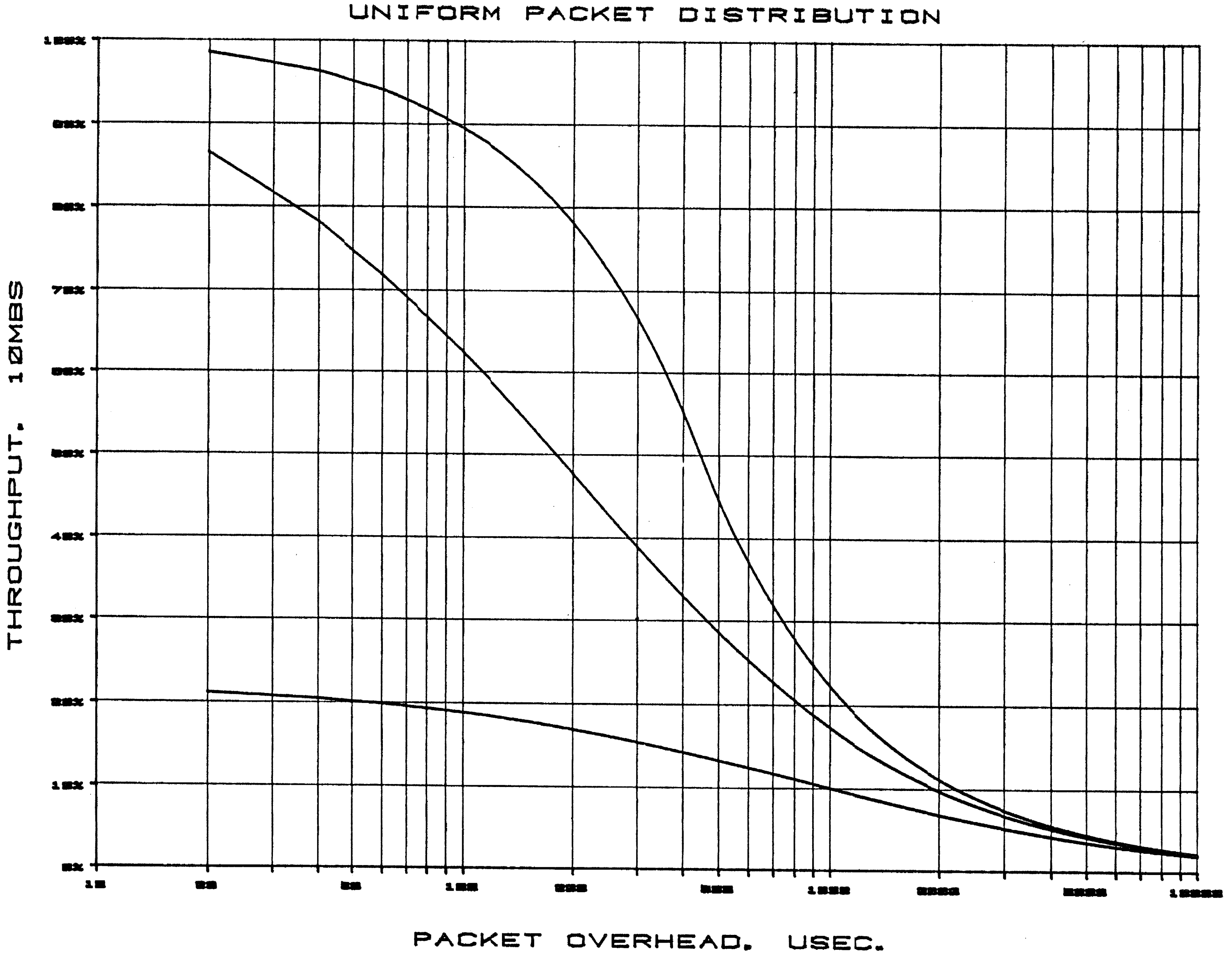

UNIFORM PACKET DISTRIBUTION
THROUGHPUT, 1ØMBS
PACKET OVERHEAD, USEC.

FIGURE 2.10

upon a transfer time that is dependent upon the processor's speed than it is upon the network's bit rate.

DMA implemented variable address buffers also seem to cost less to implement than the other techniques, not due to any inherent simplicity in the DMA hardware but rather to the high degree of integration that can be achieved using available controller chips.

2.4 Serialization And Synchronizaton

The Ethernet transmitter module's main task is to serialize and encode data. It must do a few other things such as prefixing the packet with a start bit and tacking on a CRC at the end. It must also interpret error conditions such as collisions and act accordingly. All of these things are pretty straightforward. However, the serialization and encoding process does set some system requirements as to clock rates. Serialization usually necessitates a clock rate equal to the bit rate. Performing the Manchester encoding on the data can be done with with a bit rate clock and some combinatorial logic but this type of implementation tends to be glitch prone. Glitchless encoding requires a state machine clocked at least twice the bit since Manchester encoded signals can have transitions at twice the bit rate.

The Ethernet receiver module's main task is to decode serial data and then parallelize it. There are two methods of decoding Manchester encoded signals. One method is to generate the receiver's clock from the signal either using phase lock loop techniques or one-shot circuits. The main disadvantage to this method is that implementations usually require components to be trimmed or rather sensitive analog circuitry. Digital sampling can be used to avoid both of these problems.

Unfortunately, sampling implies that the receiver must synchronize the incoming data to the receivers internal clock. Since synchronization introduces errors into the data [Molnar and Channey, 1973], it would be wise to design the synchronizer to reduce the probability of such errors and it would be ideal if the decoder could be designed to tolerate these errors. The sampler will require a clock that is some multiple of the bit rate. This multiple and the propagation characteristics of the logic family will determine what bit rates this type of decoder can be built for.

First, a little about Manchester encoding. If bits are encoded, the resulting waveform can be thought of as a stream of bit cells. The value of a bit cell is determined by the direction of the trailing edge of the bit cell. If it is positive going, it is a one and if negative going, it is a zero. Encoding a one after a one or a zero after a zero will require the insertion of a setup transition in the middle of the bit cell. Therefore to decode a bit, all one has to do is to look at the direction of the trailing edge. In fact the following Pascal program does just that. It synchs up on the first edge it sees and then loops waiting for trailing edges. It differentiates trailing edges from setup edges by keeping a count of the number of samples it is into the bit cell. An edge is classified as a trailing edge if this count is greater than the largest possible count that a setup edge could have. This type of decoder also has the nice property that it reacquires bit synch on every bit frame which means that the receivers clock need only be stable over a bit cell instead of over an entire packet.

```
program ManchesterDecoder;
const
    LostBitSynch = 2*Bitrate;
    SetupPeriod = ?;
var
    time, direction : integer;
    detected : boolean;
external procedure edge(var sign:integer; var found:boolean);
{ this procedure returns found = true if it detects an edge }
{ sign indicates the direction of the transition }
begin
    while true do
    begin
        Time:= 0;
        { acquire bit synch }
        repeat edge(direction,detected) until detected;
        repeat
            edge(direction,detected);
            time:= time+1;
            if detected and (time > SetupPeriod) then
            begin
                { we have found a trailing edge. Bit = direction }
                time:= 0
            end;
        until time > LostBitSynch;
        { we have lost bit synch, lets acquire it again }
    end;
end.
```

One can easily see how this algorithm can be implemented in hardware. The only question is what the minimum sampling rate and SetupPeriod are. The minimum sampling rate is determined by two requirements. The first is that we be able to detect all edges. The second is that we be able to differentiate between setup and trailing edges. Sampling introduces a one sample time ambiguity in locating an edge since the value of sampling on an edge must be though of as returning an indeterminate value. This introduction of temporal ambiguity means that to detect a pulse, we need at least three samples since the two samples on the edges are return indeterminate values. Therefore, we need at least five samples to detect the presence of a square wave and that reconstructed square wave may appear severely distorted. Five samples means sampling at four times the bit rate since the last sample of one bit cell is the first sample of the next bit cell. Therefore the requirement that we see all edges places a lower limit on the sampling frequency at 4f where f is the bit frequency.

When we sample at a certain frequency, we are dividing a bit cell into a discrete number of sampling times. The setup and trailing edges can be classified by the sampling times at which they occur. Because sampling makes edges temporally ambiguous, setup and trailing edges are characterized by a set of sampling times. To differentiate these edges, these sets must be disjoint. Therefore if we sample at nf, there are n+1 sampling times per bit cell. Sampling introduces a ± 1 ambiguity around the leading edge of the bit cell so it must introduce a ± 2 ambiguity around the setup and trailing edges. Then:

$$t(\text{setup edge}) = ((n+1)/2) \pm 2$$
$$t(\text{trailing edge}) = (n+1) \pm 2$$

In order that t(setup edge) not overlap t(trailing edge),

$$((n+1)/2) + 2 < (n+1) - 2$$
$$\Rightarrow (n+1)/2 < n - 3$$
$$\Rightarrow n+1 < 2n - 6$$
$$\Rightarrow n < 2n - 7$$
$$\Rightarrow n > 7$$

Requiring that the sampling frequency nf be greater than 7f. necessitates that the sampling frequency be at least 8f. Therefore a 3MHz Ethernet must be sampled at 24MHz. While a 24MHz sampling rate is certainly plausible for TTL receiver implementations, the 10MHz Ethernet will require an 80MHz sampling rate which will most likely cause 10MHz receivers to be of the phase lock loop type rather than digital samplers.

2.5 Conclusions

In this chapter, several communication techniques were presented and analyzed. Comparisions were made of the costs and throughputs of these several approaches and will be used in the next chapter which discusses the implementation. The previous study of the sampling rates required to decode incoming data also played an important role in the implementation of the interface and helped to point out how decoders will have to be built for the 10MHz Ethernet that is now becoming the de facto standard.

CHAPTER 3

THE HARDWARE IMPLEMENTATION

3.1 Design Criteria

The major design criteria for this system were (1) that it be capable of translating data from some hardware protocol into the Xerox PARC Ethernet protocol and vice versa, (2) that it be capable of sending a packet to itself over the Ethernet and (3) that it not require trimming of passive components to meet the Ethernet timing specifications.

The notion of the system as a general purpose translator has been discussed before. The second design criteria, the loopback requirement, is an useful tool for both system hardware testing and network software testing. It allows the programmer to write software to communicate with other machines and test it by having it communicate with his machine. The third requirement is one that is intended to both decrease production costs and decrease system failures.

3.2 System Organization

Chapter 2 presented a generalized system architecture. This system was composed of four parts. One part was a shared memory array that was accessed by a microprocessor, which acts as both a system controller and a protocol translator. The two other modules were protocol modules that also accessed the shared memory. It was also mentioned that the Ethernet module subdivides nicely into an Ethernet input module and an Ethernet output module. The undefined protocol is RS-232C and the protocol module represents four high speed UARTs in this implementation. The following discussion will concentrate on how these modules were implemented and integrated into a system. The discussion will relie heavily on the schematic drawings in Appendix A.

3.3 Microprocessor Module

It is convenient to think of the shared memory as being owned by the processor. The processor uses it for both program and data memory. When the processor wishes to allow the Ethernet protocol hardware to access its data memory, it enables operation of a DMA channel which works in a single transfer burst mode. This implementation fits the models used in Chapter 2 for variable address data buffers. While we are using a burst mode DMA channel, that does not mean that the system falls on the burst mode plots of Figures 2.7-2.10. If the transfers take all of the bandwidth of the data bus, the processor will not get any concurrent cycles and thus will be modeled by the block mode DMA curves which are less efficient than the burst mode DMA models.

The amount of bandwidth that the Ethernet protocol requires is determined by the width of the data bus and the speed of the DMA controller and memory. For example, with an eight bit wide data bus, a 3MHz Ethernet requires a data transfer rate of 375,000 bytes/second but running in loopback mode where it must receive and transmit at the same time, it requires 750,000 bytes/second. While many DMA controllers can handle this rate, there isn't much bandwidth left for the processor. Widening the data bus to sixteen bits brings us back down to 375,000 transfers/second worst case, whereas it is only a factor of two, its affect can be more pronounced on a pipelined processor with a memory fetch unit. A factor of two can mean that the processor doesn't notice the presence of the DMA transfers since its pipe is filled faster than it can withdraw. This is a more ideal case since it puts us on the burst mode DMA curve of Figures 2.7-2.10. After all if we are going to pay for burst DMA hardware, we should try to take full advantage of it.

This rationale necessitated providing a sixteen bit wide memory bus. While a sixteen bit wide memory system could be built for an eight bit microprocessor, the availability of the sixteen bit Intel 8086 microprocessor suggested that the processor also have a sixteen bit word. The advantages to using an 8086 over an eight bit microprocessor are that it has a pipelined memory prefetch unit and has a relatively high throughput. The 8086 is a controller type processor. It doesn't have the regularity or addressing power of some other sixteen bit microprocessor but had the advantage of being available and relatively fast.

This section will address the implementation of the microprocessor and memory modules as well as the serial communications module and the bulk of the DMA controller logic. Detailed schematics for these sections are labeled Sheets 1-8 in Appendix A. A good way to describe microprocessor system design is to start by presenting the view a programmer might have of the system and work down to the logic implementation.

Memory maps tend to describe microprocessor systems sufficiently for programmers to be able to write code for them. While they may need to look at specifications for peripheral chips before writing code, the memory map is usually detailed enough that once the programmer understands the peripherals, he can go on without asking many other questions about the implementation. The memory map for this system can be found in Figure 3.1. It shows that memory space has been divided into eight 8K byte pages. Three of these pages are mapped onto ROM. Another two pages have been assigned to RAM. One page is used for accessing the four UARTs and another is used for Ethernet registers, the DMA controller, the two interrupt controllers and a programmable timer chip.

3.3.1 Microprocessor Circuitry

Since Multibus compatibility and multiprocessor organizations were not required of the system, the 8086 has been configured in its minimum mode. This configuration, illustrated on Sheet #1, does not require an additional bus controller chip as does the maximum mode. An 8284 clock generator chip is used to generate system clock, reset and ready signals. Three eight bit tri-state latches, are used to latch addresses off of the processors AD<0:15>, A<16:19> and -BHE lines. When selected the outputs of these latches drive the system address bus A<00:19>,-BHE. The address latches normally drive the system address bus but can be tri-stated by the DMA controller when it is performing bus accesses.

System addressing deserves more discussion. While the 8086 has a twenty bit word address, it can read or write eight bit bytes also. It does this by either asserting -BHE (byte high enable) to select the high byte bank or -A0 to select the low byte bank. So, the 8086 actually has twenty-one address lines. This byte addressing requires that the address page select signals be generated for each byte bank. Two three to eight decoders are used to generate the address page select signals. These decoders are enabled when the processor is performing a memory cycle and the appropiate byte bank is selected or when the DMA

Figure 3-1: Address Alocation in the 8086 Microprocessor, Version 2

```
FFFF  ================
      |              |
      |              |
      |  ROM IMAGE   |
      |              |
      |              |
E000  ================
      |              |
      | ETHERNET,    |
      | DMA & INT    |
      |              |
      |              |
C000  ================
      |              |
      |              |
      |    UARTS     |
      |              |
      |              |
A000  ================
      |              |
      |              |
      |    UNUSED    |
      |              |
      |              |
8000  ================
      |              |
      |              |
      |     RAM      |
      |              |
      |              |
6000  ================
      |              |
      |              |
      |     RAM      |
      |              |
      |              |
4000  ================
      |              |
      |              |
      |     ROM      |
      |              |
      |              |
2000  ================
      |              |
      |  ROM IMAGE   |
1000  |--------------|
      |     ROM      |
      |              |
0000  ================
```

UART Register Definitions:

UART Channel 0:

Receive Holding Register	Address: A000 Read Only
Write Holding Register	Address: A000 Write Only
Status Register	Address: A002 Read Only
SYN1/SYN2/DLE Registers	Address: A002 Write Only
Mode Registers 1/2	Address: A004
Command Register	Address: A006

UART Channel 1:

Receive Holding Register	Address: A008 Read Only
Write Holding Register	Address: A008 Write Only
Status Register	Address: A00A Read Only
SYN1/SYN2/DLE Registers	Address: A00A Write Only
Mode Registers 1/2	Address: A00C
Command Register	Address: A00E

UART Channel 2:

Receive Holding Register	Address: A010 Read Only
Write Holding Register	Address: A010 Write Only
Status Register	Address: A012 Read Only
SYN1/SYN2/DLE Registers	Address: A012 Write Only
Mode Registers 1/2	Address: A014
Command Register	Address: A016

UART Channel 3:

Receive Holding Register	Address: A018 Read Only
Write Holding Register	Address: A018 Write Only
Status Register	Address: A01A Read Only
SYN1/SYN2/DLE Registers	Address: A01A Write Only
Mode Registers 1/2	Address: A01C
Command Register	Address: A01E

Register Definitions:

Ethernet Control Register: Address: C000 Write Only

```
+--------+--------+--------+--------+--------+--------+--------+--------+
|        |        |        |        |        |        |        |        |
| unused | unused |promisc.|loopback|R start |R reset |T start |T reset |
|        |        |        |        |        |        |        |        |
+--------+--------+--------+--------+--------+--------+--------+--------+
   bit7     bit6     bit5     bit4     bit3     bit2     bit1     bit0
```

T reset: This bit resets and disables the transmitter when cleared.

T start: This bit activates the transmitter when it is enabled. Transmission starts when the eneternet transmit delay counter is zero and there is no carrier present on the ether.

R reset: This bit resets and disables the receiver when cleared.

R start: This bit starts the ethernet receiver. It must be cleared inorder to restart the receiver again.

Loopback: This bit puts the ethernet interface into loopback mode where packets are not put on the ether but internally routed back to the receiver.

Promiscuous: This bit turns off packet address recognition enabling the receiver to eavesdrop on the ether.

Ethernet Status Register: Address: C000 Read Only

```
+--------+--------+--------+--------+--------+--------+--------+--------+
|        |        |        |        |        |        |        |        |
| unused |R DMAERR|CRC ERR |FRAMING | R done |T DMAERR|collide | T done |
|        |        |        |        |        |        |        |        |
+--------+--------+--------+--------+--------+--------+--------+--------+
   bit7     bit6     bit5     bit4     bit3     bit2     bit1     bit0
```

T done: This bit is set when packet transmission is terminated.

Collide: This bit is set when a collision caused the termination of a packet transmission.

T DMAERR: This bit is set when a DMA error caused the termination of a packet transmission.

R done: This bit is set when packet reception terminates.

Framing: This bit is set if there was a framing error on packet reception.

CRC ERR: This bit is set if there was a CRC error on packet reception.

R DMAERR: This bit is set if packet reception was terminated because of a DMA error.

DMA OUTPUT REGISTER: Address: C020 Write Only

This 16 bit wide register is used to write the ethernet packet address word inorder to prime the DMA channel.

SWITCH REGISTER: Address: C040 Read Only

This eight bit register contains the settings of the address selection switches.

Address Register: Address: C060 Write Only

This eight bit register should contain this Ethernet interface's address. The contents of this register is used for address recognition on incoming packets.

Timer Registers:

Counter 0	Address: C080
Counter 1	Address: C082
Counter 3	Address: C084
Control Register	Address: C086 Write Only

Interrupt Controller 0:

IRR, ISR or Interrupting Level	Address: C0A0 Read Only
IMR	Address: C0A2 Read Only
OCW2, OCW3 or ICW1	Address: C0A0 Write Only
OCW1, ICW2, ICW3, ICW4	Address: C0A2 Write Only

Interrupt Controller 1:

IRR, ISR or Interrupting Level	Address: C0C0 Read Only
IMR	Address: C0C2 Read Only
OCW2, OCW3 or ICW1	Address: C0C0 Write Only
OCW1, ICW2, ICW3, ICW4	Address: C0C2 Write Only

DMA Controller:

Channel 0 Address Register	Address: C0E0
Channel 0 Terminal Count Register	Address: C0E2
Channel 1 Address Register	Address: C0E4
Channel 1 Terminal Count Register	Address: C0E6
Channel 2 Address Register	Address: C0E8

Channel 2 Terminal Count Register	Address: C0EA
Channel 3 Address Register	Address: C0EC
Channel 3 Terminal Count Register	Address: C0EE
Mode Set Register	Address: C0F0 Write Only
Status Register	Address: C0F0 Read Only

Note: The Address and Terminal Count registers are sixteen bits wide internally but only eight bits wide at the pins. To write or read them, the first access is the least significant byte and the second access is the most significant byte. To assure that code actually accesses the registers in this order it is necessary to disable interrupts during these operations.

controller has asserted ADSTB.

The processors data bus AD<00:15> is bi-directionally buffered to form the system data bus D<0:15>. These buffers are enabled by the processor signal -DEN and the direction is selected by DT/-R. The processor's -RD and -WR lines are buffered to provide system -READ and -WRITE signals. An inverted clock processor clock signal, -CLK, is also provided. Since the processor floats some of its control lines when it grants another device the system bus, several lines have 22K pullups on them to bring them to a high state. The i8086 also requires that its hold input be synchronized. A D flip-flop performs this function. Since the D input, HRQ, is derived from the system clock by the DMA controller chip, it is always stable during clock transitions.

3.3.2 EPROM Circuitry

The memory map shows what looks like three pages of ROM. However there are only two physical banks of ROM that total 8K bytes not 24K as could be inferred from the memory map. Where the memory map indicates ROM IMAGE, it means that ROM is mapped onto multiple pages or subpages. For example, the primary ROM consists of two 2716s as in Sheet #2. These 2K by 8 EPROM chips are selected when either of the page selects for the 0th page or the 7th page are asserted low. This selection scheme has the affect of making the same ROM chips appear to be in two places, at the bottom of memory, at addresses 0H-1FFFH and at the top of memory at addresses E000H-FFFFH. Also because the chips are 2K byte devices, A12 is not used to select them. Another two images now appear so that the physical ROM is mapped into 0H-0FFFH, 1000H-1FFFH, E000H-EFFFH and F000H-FFFFH. Secondary ROM, Sheet #3 is mapped in a similar way but only onto 2000H-2FFFH and 3000H-3FFFH.

There are some reasons for this mapping. Mapping 2K byte ROMS into 4K byte banks was done so that when 2732s became available, the 2716s could be replaced with a minimal amount of work. The strange mapping of the primary ROM into both bottom and top pages resulted from the architecture of the 8086. The processor expects interrupt dispatch tables to be in the bottom page while it expects its reset vector to be in the top page. This mapping maps both interrupt vectors and the system reset vector into the primary ROM.

3.3.3 RAM Circuitry

Two pages of RAM are allocated in the memory map. These are illustrated in Sheets #4 and #5. A page of RAM is implemented with sixteen 2141s, which are 4K by 1 static RAMs. Each page consists of a high and a low bank which is selected by the appropiate bank select signal. The processors -WRITE signal is used to indicate the type of operation to be performed.

3.3.4 UART Circuitry

Four UARTs are provided for communications with computers and terminals. These UARTs, Signetics 2651s, are capable of transmitting and receiving data at 19.2K baud in asynchronous mode. Baud rates are software selectable. In addition the chip has two interrupt lines, one that indicates when a character has been received and another that indicates when the transmitter pipe is empty. The chip can also be software configured to communicate in several synchronous modes.

Sheet #7 illustrates the UARTs and associated hardware. Each UART is selected by the output of a 74LS138 which decodes address lines A<3:5>. It is enabled by -LOWUARTSEL and disabled by HLDA, which ensures that the UARTs are not inadvertently selected during DMA cycles. All of the interrupt condition outputs are open drain and must be pulled up and inverted before being applied to the eight interrupt lines on the 8059. This chip is an interrupt controller that generates vectored interrupt requests to the i8086. It is operating as a slave to the master 8059 interrupt controller detailed on Sheet #8.

3.3.5 Ethernet Support Devices

Sheet #6 illustrates the bulk of this circuitry. A 74LS138 is used to decode ethernet bank addresses. The least significant output, -ENETCSRSEL is gated with -WRITE to select the Ethernet Control Register and it is also gated with -READ to select the Ethernet Status Register. The control register is implemented with a 74LS273 and the status register with a 74LS244. Another decoder output, -SWITCHSEL, is used to enable the switch register. The switch register provides the programmer with an octal dipswitch that he may use for initializing the nodes net address. Still another decoder output, -TIMERSEL, is used to enable an 8253-5 which is a triple sixteen bit interval timer. This chip is configured so that Channel 0 is used as a clock prescaler and its output is used as clocks to Channels 1 and 2. Channel 1 is used by the Ethernet transmission hardware. It is necessary that the output, ETIMER, be asserted before packet transmission can proceed. The Channel 2 output, TIMER, is used to generate clock interrupts to the microprocessor.

Sheet #8 illustrates the DMA controller and the main interrupt controller. The DMA controller was finessed into working on a sixteen bit data bus. As a result, it can only transfer integral words. Also, the transfer start addresses that are written into the DMA channel address registers must be shifted right one bit so that they are word addresses. The DMA controller works by strobing part of the address into an eight bit register and forcing another eight bits of address onto the bus itself. The 8086 has five other address lines, A<00,17-19>,-BHE, that must also be taken care of. A tri-state buffer serves to force these lines to zero during the DMA transfer. This DMA controller does memory to I/O transfers in one bus read or write cycle unlike some other controllers that require two cycles. Since it takes the controller as long to acquire the bus from the processor as it does to do the transfer, the controller has been designed keep the bus if it has another transfer request pending.

The interrupt controller, an 8259A, generates vectored interrupts to the 8086. It has three primary interrupt inputs which are DONEINT, which is the Ethernet receiver interrupt signal, OUTEND, which is the Ethernet transmitter interrupt and TIMER which is the programmable timer interrupt. Another input is programmed as a slave and is connected to the other 8259A which handles the UART interrupts.

3.4 Ethernet Output Module

This hardware, which is responsible for the transmission of a data packet, is detailed on Sheets #9 and #10. The Ethernet protocol requires that it do a few things. First it must show "deference." This means that the transmitter must wait for the "carrier" to go away. Secondly, it must tack a start bit onto the front of the packet and a CRC-16 word to the end of the packet. Finally, the protocol requires "collision consensus enforcement" which means that when a collision is detected the transmitter must jam (pull down) on the "ether" for a period of time greater than a bit transmission period.

There are three ways to stop transmission of a packet. Two of these are valid error conditions. The first is a collision and has already been described. The second error condition is a DMA channel failure that does not provide the next data word when it is needed. The third is an abort from the CPU.

The output hardware is used in the following manner. When the background program wishes to send a packet, it will write the first word which contains the destination and source addresses into the DMAOUT port. It will then set up the Channel-0 DMA address register with a pointer to the first word of the message and then put the length of the message in the transfer count register. It will set the transmit bit in the ethernet control register (enetcr) enabling the hardware which will defer before sending. Upon completion, it will generate an interrupt. The interrupt routine will check the ethernet status register (enetsr) and if it was a error free transmission, it will convey that to the background program through a message buffer. If there was a collision, it will use the Xerox retransmission control algorithm [Metcalf and Boggs, 1976] to schedule retransmission of the packet. In the case of a DMA error it may retransmit or return an error condition to the background program. Typical interrupt routines can be found in Appendix C.

3.4.1 Normal Packet Transmission

Figure 3.2 illustrates the shortest normal packet transmission. The packet is three words long; the first word is the Ethernet address header word, the second word is a data word and the final word is the CRC word. Packet transmission is initiated by (1) writing the Ethernet address header word into the dma output register, (2) writing a timeout value into Channel 2 of the timer chip and (3) by setting both the transmitter reset and start bits in the EnetCR.

The transmitter must first wait for the timer to decrement to zero and must then defer by waiting for the carrier to be absent before proceeding. When these conditions exist, the signal EoutProceed is asserted which causes OutGO to be asserted two clock ticks later. OutGO is used to start up a micro-programmed sequencer which is responsible for requesting DMA transfers, serializing data and Manchester encoding the data for transmission. When this sequencer starts up, a 4-1 multiplexer has been forcing its output, EmuxData, high and it remains forced high for one clock tick after which it is logically connected to the output of the data shift register. Since EmuxData is the serial stream of bits to be encoded for transmission and used for calculating the CRC, forcing this signal high inserts the start bit into the data stream.

When the sequencer is encoding the start bit, it asserts -srload which causes the Ethernet address header word to be loaded into a shift register. This word is subsequently shifted out, encoded and transmitted. After the sequencer asserts -srload, it asserts DMArq for a clock cycle causing a transfer request to be issued to the DMA controller. The DMA controller has 15 clock cycles to complete the transfer before the next assertion of -srload. In this scenario, the DMA controller does so and the data word is latched in the DMA output register by DMAoutStb then loaded into the shift register and encoded and transmitted. When the data word was latched into the DMA output register, the DMA controllers terminal count signal was also latched. Since the DMA channel was programmed only to transfer one word, that signal was asserted during the transfer and can be seen in the assertion of the signal LastWord when the data word is loaded into the shift register.

The next occurrence of -srload causes the CRC word to be routed through the multiplexer and transmitted. The following occurrence of -srload delayed by one clock cycle generates the signal OutEnd which disables the sequencer and generates an interrupt to the processor.

OutEnd is also used to report the status of the transmitter via the Enetsr as the transmitter done bit.

3.4.2 Collision Concensus Enforcement

When the transmitter detects a collision on the Ethernet during packet transmission, it must jam the net for more than a bit time and then get off the net. Figure 3.3 illustrates how the transmitter handles collisions. In this implementation, a negative going transition on the transceivers signal NetCollision is used to set a flip-flop and produce a signal -collision which is asynchronous with respect to the system clock. This signal is synchronized by sampling it with a shift register. The first tap on the shift register is used to produce a signal -Jam which causes the sequencer to produce a low on XDat. The fourth tap is used to abort packet transmission three clock cycles after the jam condition was asserted. Abort conditions halt the transmitter by presetting the flip-flop that generates the OutEnd signal causing an interrupt to be generated and when the status register is polled, the collision bit will be seen as set.

3.4.3 DMA Failure

The system is designed such that two DMA transfers can occur within a word transmission time. If the DMA channel were to fail, the transmitter would retransmit the previous word and could send a correctly formatted packet which in actuality contained flawed data. For this reason, it is important to detect DMA channel errors and react appropiately. Figure 3.4 illustrates how the transmitter reacts under these circumstances.

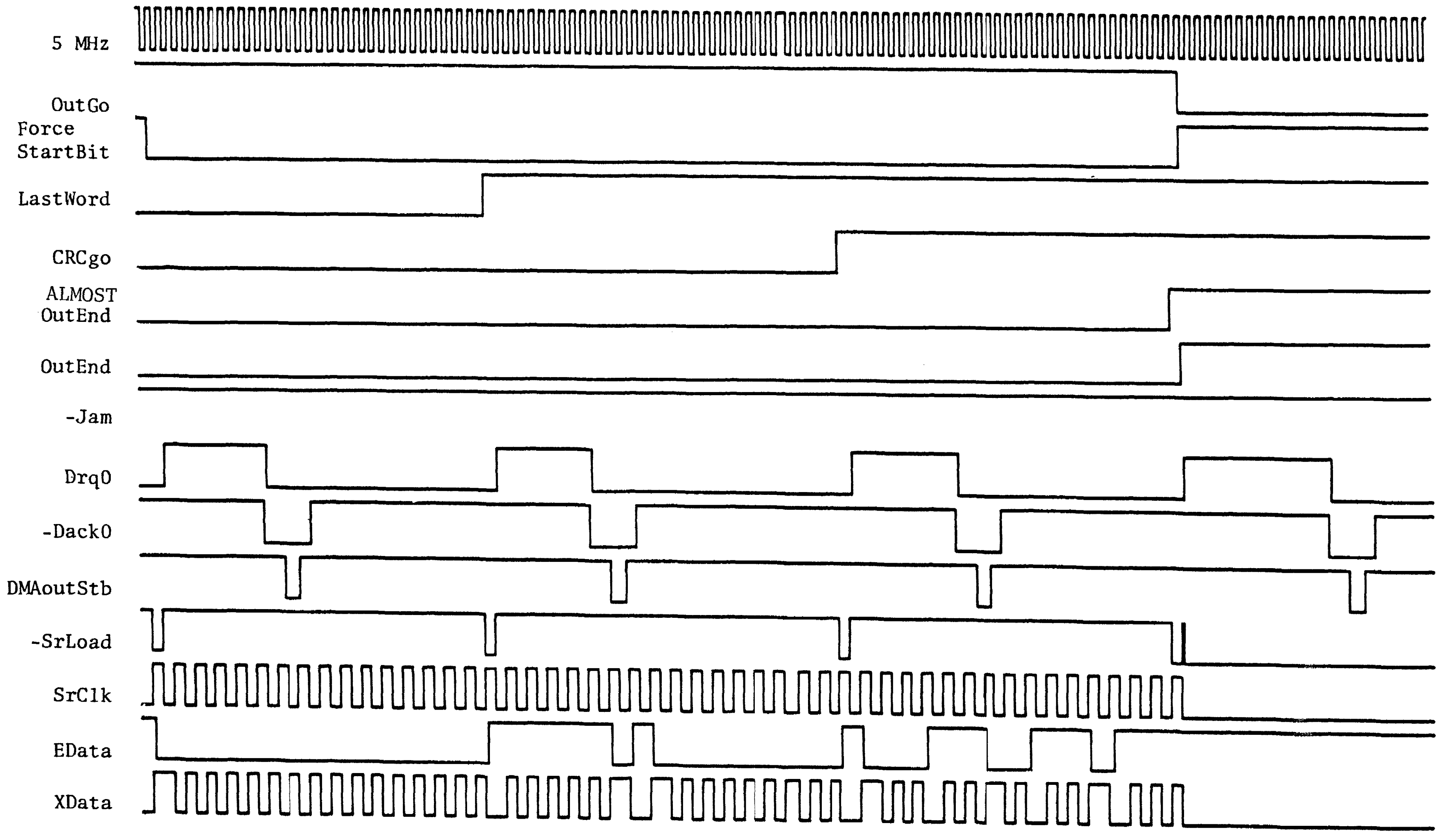


Figure 3.2: Normal packet transmission of a minimum length packet. The first word transmitted is the address header. The data word follows and the CRC-16 word is the last.

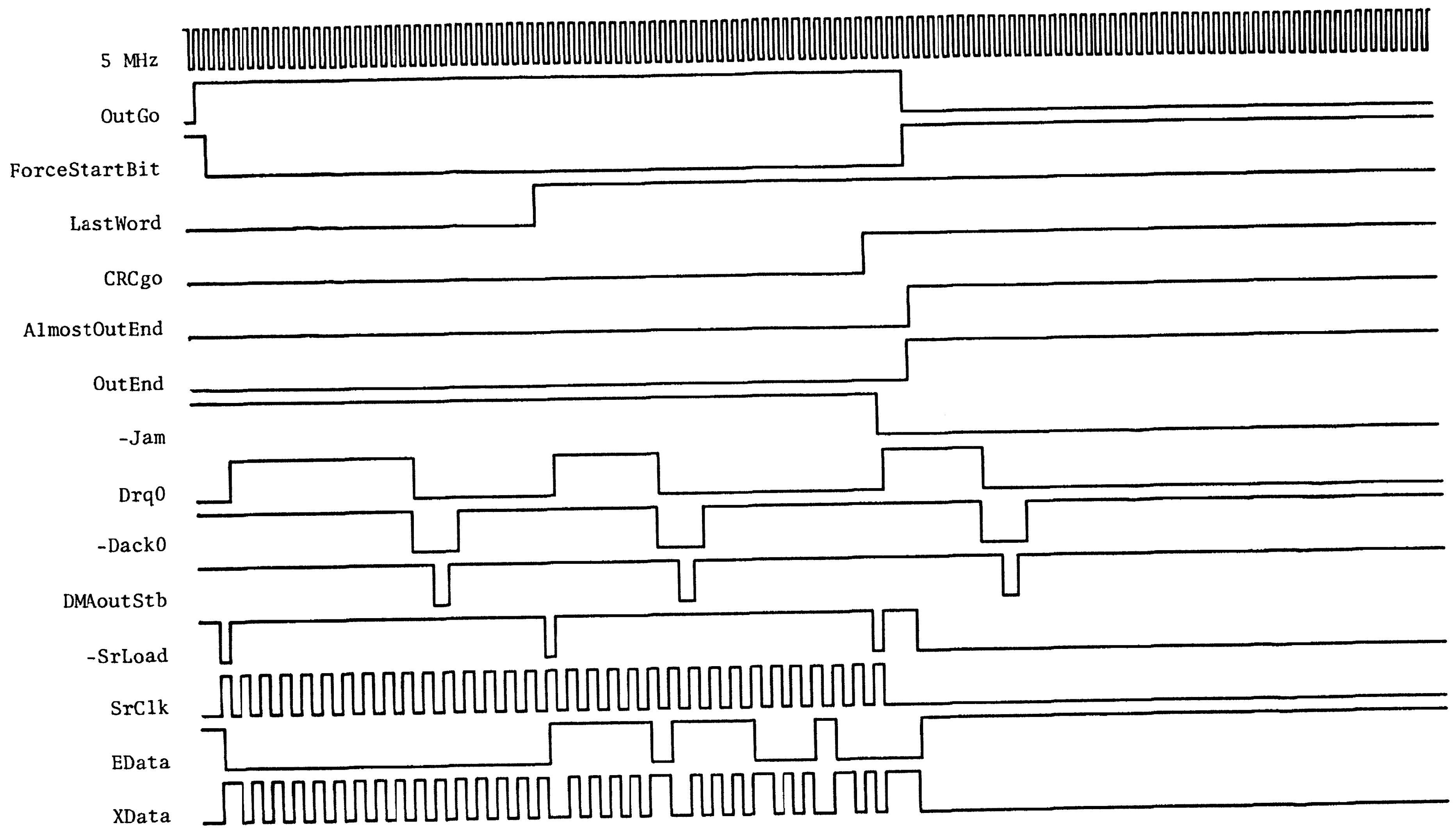


Figure 3.3: Packet transmission aborted due to collision. When a collision is detected, the -Jam signal is asserted causing the XData signal to be forced high for longer than a bit time.

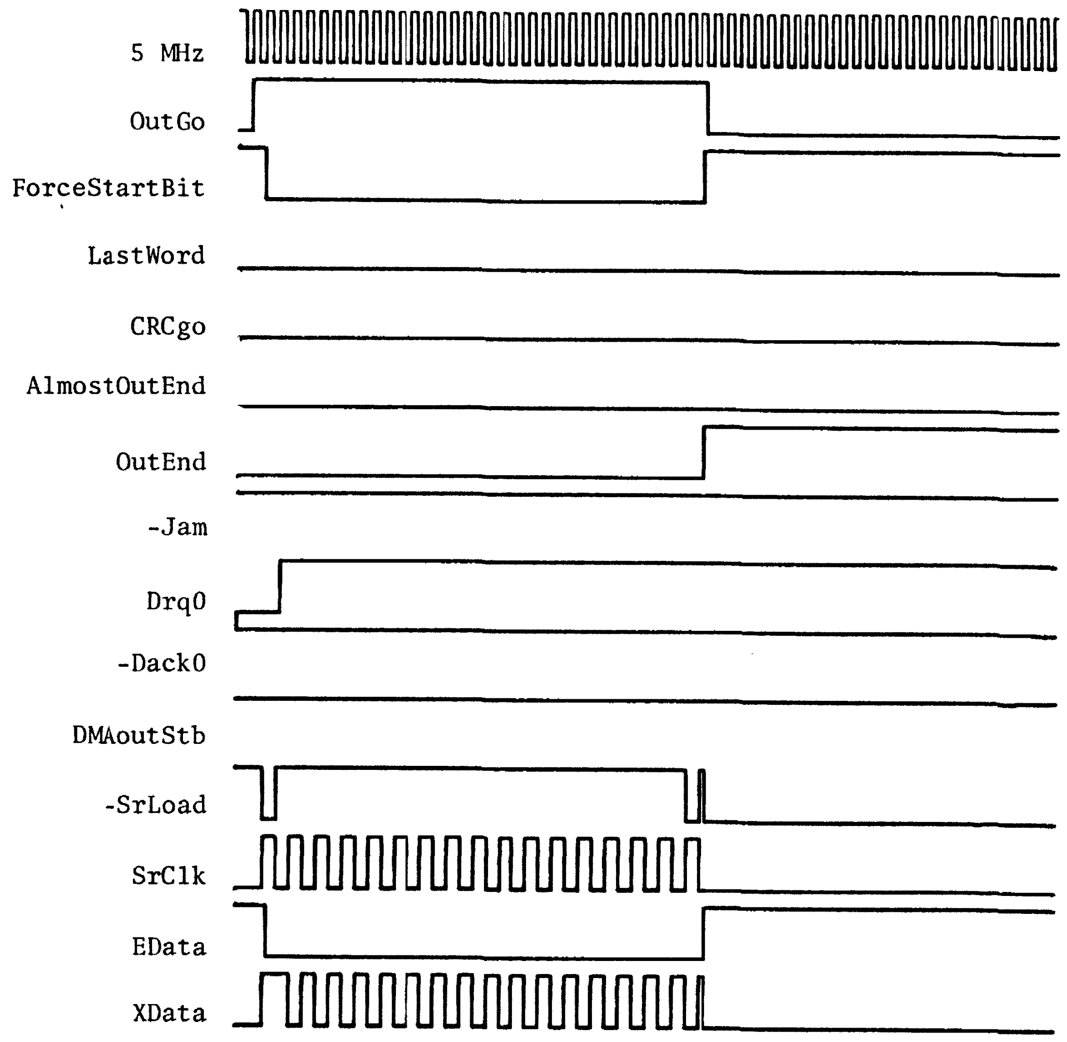


Figure 3.4 : Packet transmission aborted due to DMA channel failure.

A DMA channel failure can be characterized by the request, Drq0, still being asserted within two clock cycles of DMAREQ0 being asserted. Again, since Drq0 is asynchronous with respect to the system clock, a shift register is used to synchronize it. The signal at the second tap is currently ORed with -DMAREQ0 to produce an abort signal which is latched and appears as the transmitter DMA error bit in the status register. This abort signal shuts down the transmitter in a similar manner as with collisions but as the timing diagram illustrates, the Ether is not jammed.

3.5 Ethernet Input Module

The Ethernet receiver hardware is naturally divided into several sections. First, data coming off the net must be decoded. When the first word of a packet has been collected, its destination address must be qualified. If the node wishes to receive that packet, data must be transferred into the memory while a CRC is being calculated. After the packet has been collected, the hardware must report whether there were any error conditions. The receiver hardware is illustrated in Sheet #11.

3.5.1 Loopback Multiplexing

Ethernet received data is piped into a loopback multiplexer, U91, before making its way to the decoding circuitry. This multiplexer will either connect the output of the transmitter, XDAT, to the receiver or connect NETDATAIN to the receiver. Simultaneously, it will connect the transceiver data input to VCC or XDAT, respectively.

3.5.2 Manchester Decoding Circuitry

Received, post-loopback multiplexer data is Manchester encoded and must be decoded to provide several signals. The CARRIER signal is asserted true while there is data on the net. DECODED DATA is the value of the bit cell when DECODED CLOCK is asserted. -CLEAR COUNTER is used to clear a four bit counter, U96, which is used to count the number of decoded bits that have been accumulated. It is cleared such that the start bit is not counted.

The decoder works by sampling the received data at eight times the bit frequency and looking at triplets of samples at four times the bit frequency. The data is sampled by an octal shift register using the last three taps for the triplet. Using the last three taps gives the sampler five clock cycles to settle, which means that the probability that any signal in the triplet is in a metastable state is very small.

Since the triplets are latched by U95 at four times the bit frequency the state machine formed by the prom U87 and the latch U88 sees triplets that overlap by one sample. The coding of the decoder is similar in concept to that in Chapter 2 except that it is not as straight forward. The coding is described in Appendix B.

3.5.3 Word Accumulation And Address Detection Circuitry

The DECODED DATA output of the decoder is used as an input to a sixteen bit shift register in which the data word accumulates. Data is clocked into the shift register one-half a clock cycle after it appears on the DECODED DATA line. It is also clocked into a CRC generator, U100, at the same time and the clock is used to increment a four bit counter, U96, which is used to count the number of bits in the shift register. It is cleared by the decoder state machine such that the start bit is not counted. Its ripple carry output is delayed one-half clock cycle and used to latch the shift register data into the DMA Input Register, U52-53.

Address Detection is done by comparing the first received data byte with the contents of the Address Register, U43. This comparison is done with two four-bit comparators, U97-98. Two four-input NOR gates and an AND gate are used to detect when the destination address is zero. The resulting signals, MY ADDRESS, and BROADCAST are NORed with the PROMISCUOUS signal from the control register to generate a -goodpacket signal which is sampled by another state machine.

3.5.4 Manager State Machine

Another PROM and latch, U89 and U90, form a state machine which manages the receivers interaction with the DMA channel and the processor. It initializes the CRC generator, requests DMA cycles, detects DMA errors, generates an interrupt indicating the receiver needs attention and holds error status by disabling the clock to the CRC generator and the bit counter.

Figure 3.5 illustrates this small state machine. It can be initialized to State 0 by asserting start low and then high. It will remain in State 0 as long as there is no carrier. When carrier is asserted and start is not, it will enter State 1 where it will remain for the complete packet transmission, that is until carrier is asserted low. State 1 is used to prevent the reception of a packet that has already been partially received when the start bit is set by the processor.

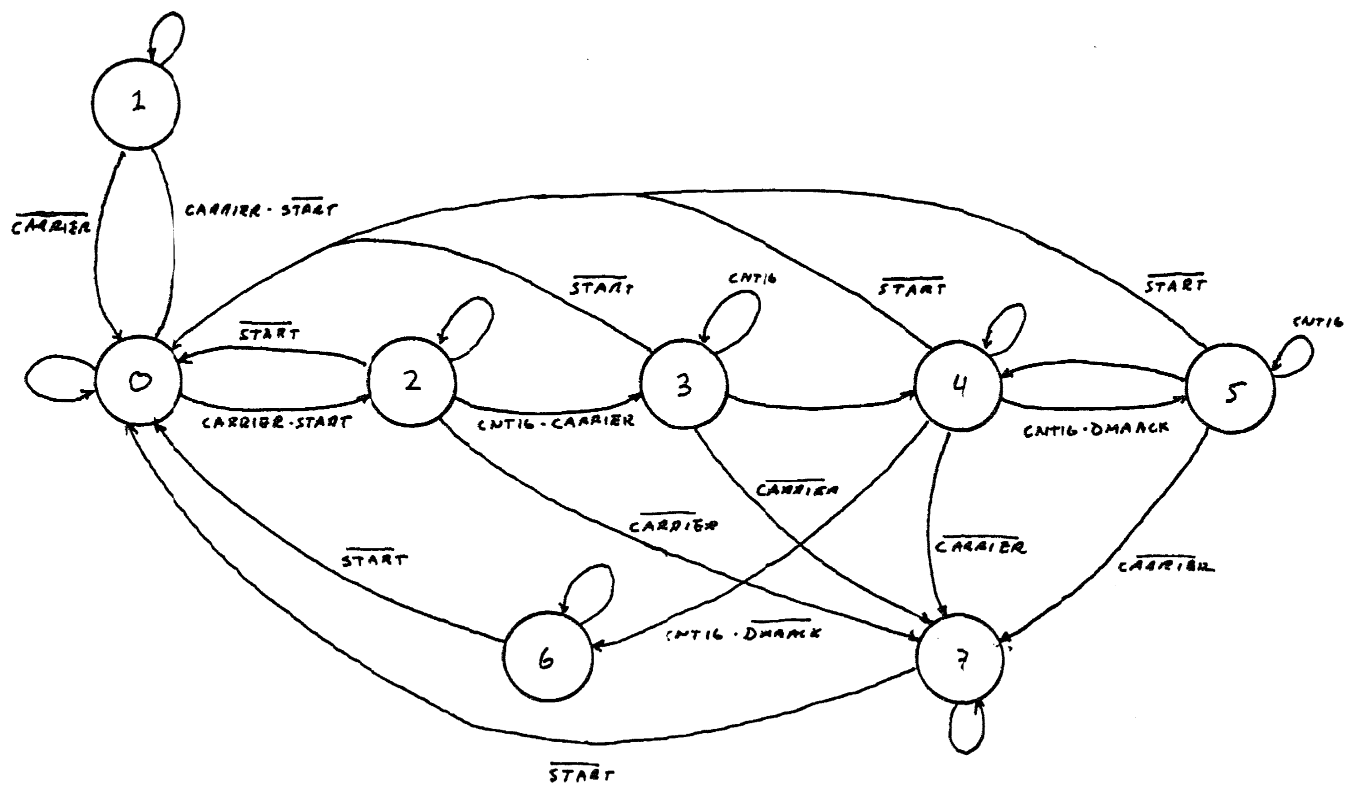


FIGURE 3.5 : MANAGER STATE MACHINE

If start is asserted when carrier is asserted, the machine will enter State 2. It will stay in State 2 until all sixteen bits have been accumulated in the shift register. When this has occured, Rippled will be asserted. If GoodPacketAddress is asserted simultaneously, the machine will progress to State 3 where it will wait until Rippled goes low before progressing to State 4. It will remain in State 4 until the next word is accumulated in the shift register and Rippled is once again asserted. From here it will enter either States 5 or 6. State 6 is entered upon detecting a DMA channel failure which is indicated by the presense of -dmaack when rippled is asserted. Otherwise, if dmaack is asserted, it will enter State 5, and stay there until Ripled is asserted low or carrier is asserted low in which case packet transmission is complete and the machine enters State 7.

States 6 and 7 are terminal states. Both generate interrupts to the processor and can be left only through processor intervention. State 7 is entered in the case of non-dma-error termination.

3.6 System Performance

The implementation appears to meet all three of its design objectives. It has been programmed and shown to be capable of receiving data on its serial ports and encoding that data in Ethernet packets and transmitting that data on the net. Likewise it is capable of receiving Ethernet packets from the net, decoding the data from the packets and transmitting that data onto the serial port.

It is also capable of transmitting packets to itself. Figure 3.6 illustrates that a relatively small amount of the data bus bandwidth is required during the loopback packet transmission-reception, suggesting that the processor is getting quite a few instruction cycles during packet transmission and reception and thus the system can be modelled by

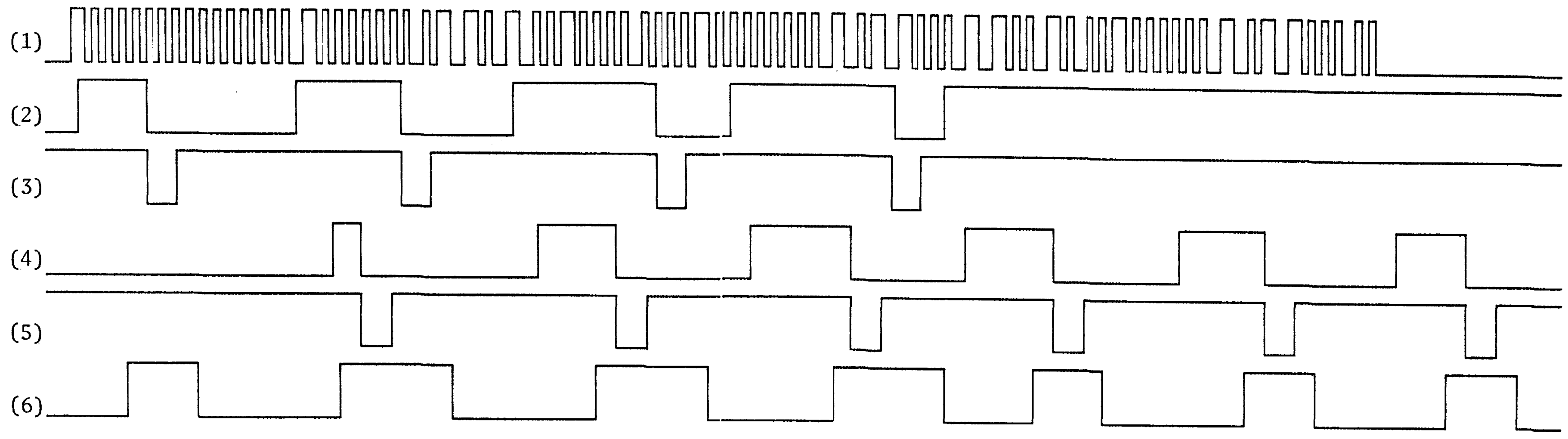


Figure 3.6: Data Bus bandwidth during loopback operation.
Signals from top to bottom are (1) transmitted Ethernet data, (2) Ethernet transmitter DMA request, (3) -(Ethernet transmitter DMA acknowledge), (4) Ethernet receiver DMA request, (5) -(Ethernet receiver DMA acknowledge) and (6) the CPU's Hold Acknowledge which indicates when the processor has given up control of the system data bus.
Note: The system acheives approximately 50% duty cycle during loopback operation.

the favorable parallel burst mode DMA throughput curve in Figure 2.7.

Since the system makes use of a digital Manchester decoder, no trimming of passive components is required as would be the case with a more traditional one-shot based decoder. This design should increase the MTBF for the decoder section. The MTBF should reflect component failure rates and not be dominated by passive component drift as would be the case in one-shot decoders.

CHAPTER 4

A PROPOSAL FOR AN VLSI IMPLEMENTATION

4.1 Higher Level Protocol Considerations

When one looks into network software, it becomes apparent that the performance of the network interface can be inconsequential when compared with the performance of the several layers of software protocol modules through which a packet must traverse. One can take the attitude that nothing the interface can do can improve the performance of these software protocol modules, however, the natural evolution of the network interface is towards a front end processor which must implement these software protocols, and a great deal may be gained by so doing.

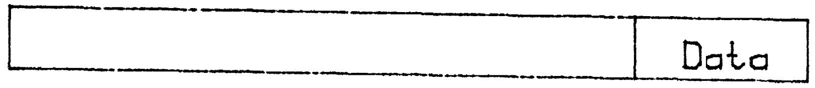
Most software protocols deal with adding or removing packet headers and trailers. So the process of sending a packet of data can be thought of an encapsulation process [Schoch et al, 1980] that takes a high level protocol packet and passes it down to a lower level protocol module where it is again encapsulated. Since encapsulation may happen many times before a packet is handed to the hardware for transmission, it is apparent that overall system performance depends heavily on the computer's ability to move data about and allocate buffer memory space.

Poor protocol module implementations may pass a formatted packet in a buffer to the next lower protocol module which may then request that a larger buffer be allocated and then copy the packet into that buffer. This technique results in allocating a lot of memory and moving data very often. An alternative and much improved technique allocates a memory buffer once and does not move the data. This is achieved by allocating a large buffer and by placing the initial data in the proper part of this buffer. While this technique has good performance, it has several drawbacks. Since a maximum size buffer must be allocated and the data must be properly placed, higher level protocol modules must have a substantial knowledge about what lower level protocol modules the packet will travel through and what they do to the packet. This knowledge of the lower level protocols goes against the rationale for using layered protocols, that is to isolate one functional level of work from another. In other words, it is just not clean.

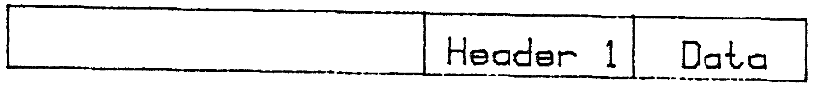
A redesign of the network interface can improve overall performance by not requiring data to be copied. If each protocol module requests a memory buffer for headers and trailers and links them to the data buffer through a link field in the memory buffer, the hardware can be passed a pointer to the packet header and it can follow the links while transmitting the packet. Such an implementation, as illustrated in Figure 4.1, is clean since the higher level protocols no longer require any knowledge about the workings of the lower level protocols.

This linked list architecture may be less efficient than the maximum size buffer technique since each protocol module may have to request that buffers be allocated for headers and trailers. However, the affect of more memory allocator calls may be less profound than before for two reasons. First, since higher level protocols tend to

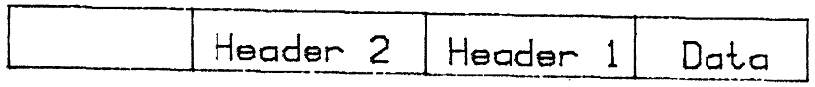
1. A maximum size buffer is allocated. Data is written into it near the end.



2. The buffer is passed to the next protocol module which prefixes a header to the data.



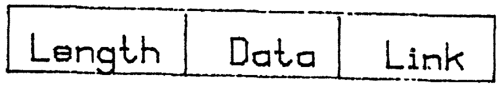
3. The buffer is passed to the next which encapsulates it again.



4. And so on.

Figure 4.1.a: Maximum Size Buffer Allocation

A Segment looks like:



Where Length is the length of the data field and the link is the address of the next segment of Ø if the end of the list. The example of Figure 4.1.a would look like:

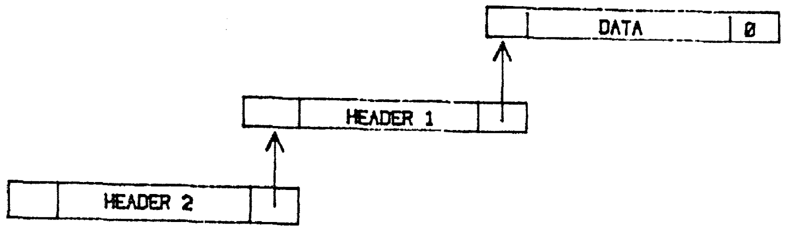


Figure 4.1.b: Linked List Buffers

deal with connections that are relatively long lived, an intelligent protocol implementation can build header and trailer templates that are kept for the duration of the connection. These templates need will need to be modified just slightly for each packet transmission. Thus the memory for the templates need only be allocated once per connection and not once per packet transmission. Secondly, using a linked list architecture makes the memory allocator easier to write and more efficient. Since arbitrary amounts of buffer memory can be built up from linking discontiguous memory segments together, the allocator only has to deal with fixed length memory segments and doesn't have to worry about compaction since the linking makes the segments appear contiguous. Furthermore, by using small segments, the allocator can make more efficient use of memory since most headers, trailers and data will fit in small segments.

The *linked list architecture requires building a fancier* DMA channel than was employed in the current implementation. When given a pointer to a linked list, the DMA channel should be able to either read data from it or write data into it. Reading from the linked list proceeds until the channel finds a null link which signifies the end of the linked list. Writing into a linked list operates similarly to reading. The channel must have a pointer to the linked list, each segment of which contains a length field, a data field and a next link field. The channel simply deposits data in the data field until it has filled it up, as indicated by the length field. The next write causes the channel to follow the link to the next segment and deposit the data in its data field. If for some reason, the channel finds a null link field, it must abort the write operation and report a buffer overflow condition. Such a DMA channel along with Ethernet receiver and transmitter modules, similar to the ones implemented in the current interface can be built using todays integrated circuit technologies (VLSI).

With time, one can expect the higher level protocol modules to be implemented in hardware rather than software in order to increase system performance. A hardware implementation of a typical layered protocol system lends itself to being a pipelined interconnection of protocol chips and shared memories.

To illustrate how higher level protocols might be realized with pipelined hardware modules, lets look at several of the Department of Defense Standard Protocols. First, from a users view, is a protocol called Transmission Control Protocol (TCP) [TCP, 1980] which can be used to establish connections and carry on reliable communications with another host. TCP uses a protocol called Internet Protocol (IP) [IP, 1980] to send datagrams to another host somewhere on an internetwork. IP in turn uses the transport level protocols to send the datagram on some network, for example an Ethernet or the ARPAnet.

An implementation of the transmitter functions of these protocols might look like Figure 4.2. The user composes data in the Buffer Memory and passes control information to the TCP Module. This control information contains at least a pointer to the data in the Buffer Memory and probably other data pertaining to the TCP services being used. The TCP Module uses the control information to compose a TCP header block in its local store, the TCP Header Memory. While the TCP Module may need to inspect the data packet to compose its header, the user never needs to see the TCP header. Because of this, the Buffer Memory is mapped into the TCP Modules address space but the TCP Header Memory is not mapped into the user's memory space. After the TCP module has formatted the TCP header, it passes control information down to the IP Module which builds an IP header in its local store, the IP Header Memory, and likewise, it maps both the Buffer Memory and the TCP Header Memory into its address space. Of course, as each protocol module builds a header block, that header is linked to the data portion of the packet. At the end, a pointer to this linked list is passed to the Ethernet Transmitter Module which gains access to the net and transmits the packet by

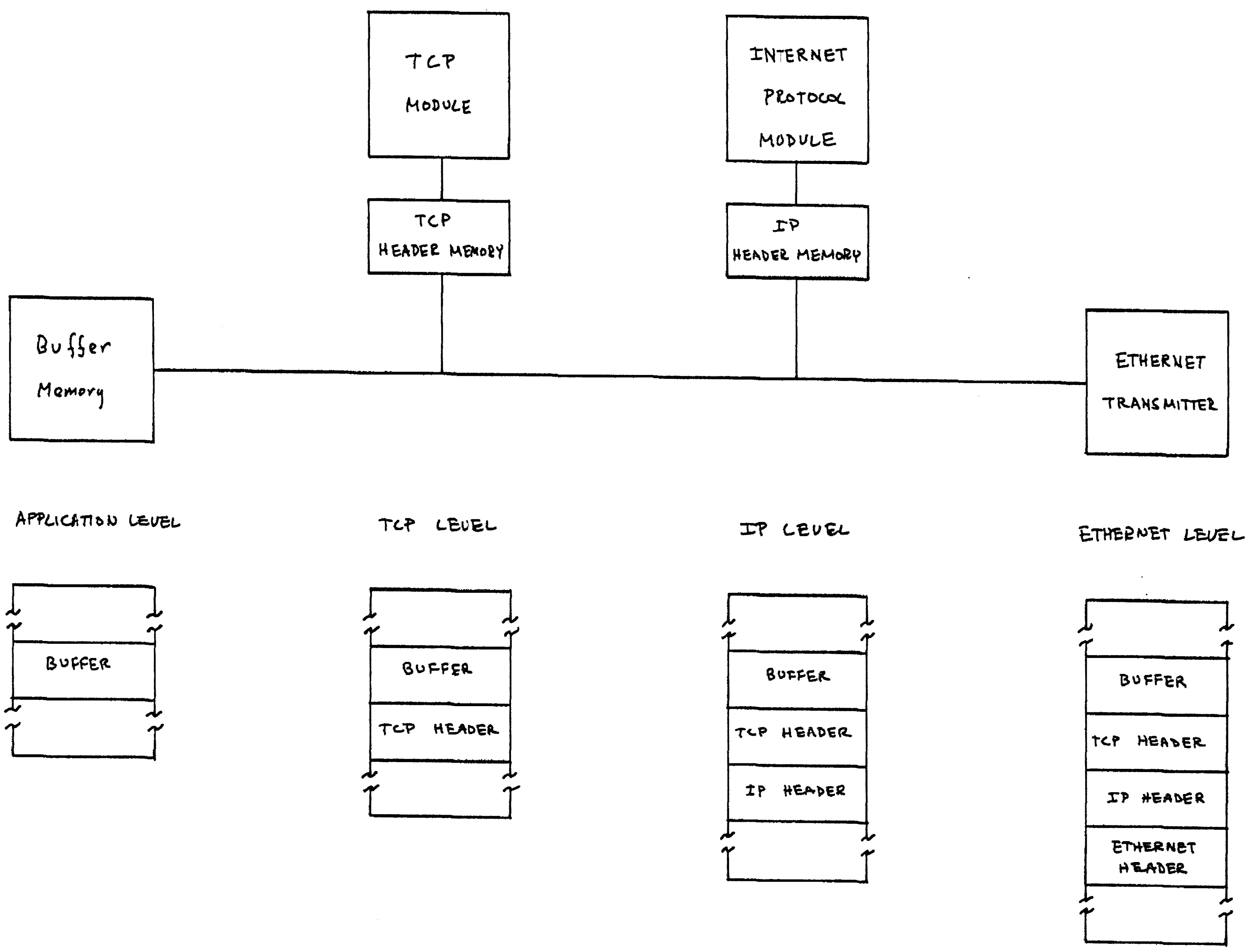


FIGURE 4.2: HIGHER LEVEL PROTOCOL MODULES & CORRESPONDING ADDRESS SPACES

following the linked list. In order to chase the linked list, the Ethernet Transmitter Module must have all of the header memories as well as the Buffer Memory mapped into its memory space.

It is important to note that by using a structure like Figure 4.2, the user looses no flexibility in the type of packets that he can send. The user can format any style packet in his Buffer Memory and pass appropiate control information to any of the protocol modules to affect any desired protocol.

Packet transmission is only half of the problem. A protocol like TCP deals with reliable communications which means that packet receptions are acknowledged by sending a packet back to the transmitting protocol module. This means that the acknowledgement packet must get back to the TCP Module that transmitted the original packet. Figure 4.2 does not illustrate the packet reception data and control pathways. During packet transmission, the datagram was fragmented into data sections and headers which were chained together in a linked list. This provided both improved efficiency and data security. On the receiver side, it is hard to take incoming data and fragment it into header and data blocks since these protocols use variable length headers. However, the incoming data can be stored in a linked list buffer without regard to the header/data boundaries. This results in an efficient implementation because a fixed sized segment memory allocator can be used and while a maximum sized buffer must be originally allocated, unused segments in the list can be returned to the memory allocator after the packet has been received.

Upon reception of a packet, receiver control flows very much like it did in the transmitter except that everything is reversed. Figure 4.3 illustrates the whole transmitter/receiver. As is shown, control information about a packet reception is passed to the IP Receiver Module if the Ethernet receiver determines the packet is an IP format packet. Otherwise, the packet is handed to the user to decode.

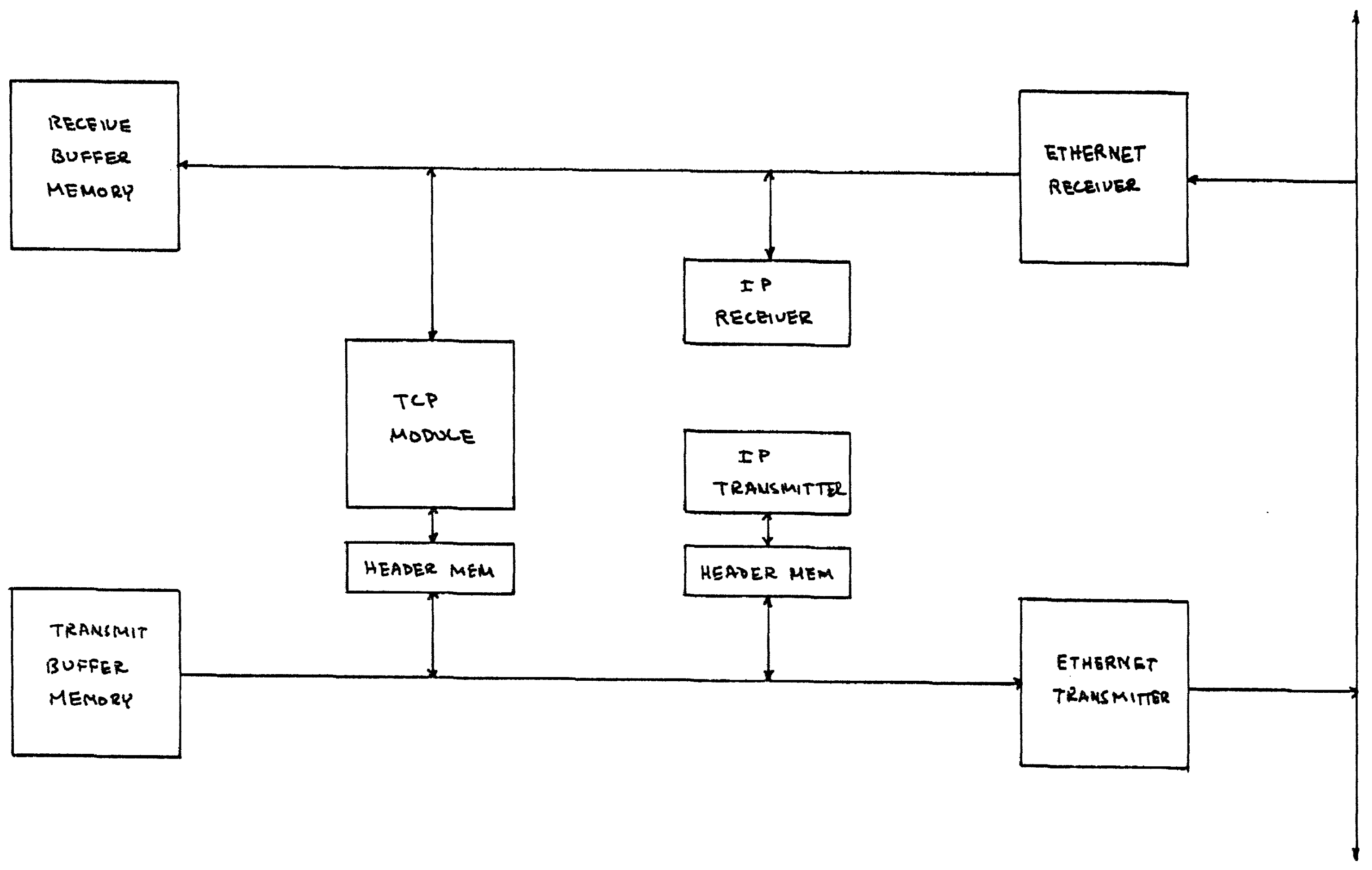


FIGURE 4.3: PIPELINED RECEIVER/TRANSMITTER

If IP is handed the packet, it decodes the packet and if it determines that the next higher protocol is TCP, it passes the appropiate control information to the TCP Module which determines what to do with the packet.

Such an hardware implementation of higher level protocols is relatively straightforward and many things can be done to improve it. The pipe can be widened in places by adding, for example, three IP module instead of one. Overall memory system bandwidth requirements can be decreased by adopting certain conventions such as carrying checksums around with the data in order to keep protocol modules from ever having to look into memories that are non-local. Finally, other architectures are well suited to this type of problem. For example, an architecture that pipes data as well as control could be very effective. Such an architecture would decode received data on the fly and thus would be able to easily separate headers from data.

In conclusion, much has been learned from implementing an Ethernet interface, but to be honest, more has been learned from programming it. No longer do I believe that the hardware protocol is of much consequence, having realized what performance degradations higher level protocols introduce. I feel, as I have tried to express in this chapter, that VLSI implementations have the ability to both provide more intelligent data structures and increase the performance of higher level protocols drastically. Either of which will have a noticeable effect on overall system performance.

REFERENCES

[Abramson, 1970]
N. Abramson, "The ALOHA system-Another alternative for computer communications," in 1970 Fall Joint Comput. Conf., AFIPS Conf. Proc., vol. 37. Montvale, NJ: AFIPS Press, 1970, pp. 281-285.

[Chaney and Molnar, 1973]
T.J. Chaney and C.E. Molnar, "Anomalous behavior of synchronizer and arbiter circuits," IEEE Trans. Elec. Comput., EC-22, April 1973, pp. 421-422.

[DEC, Intel and XEROX, 1980]
Digital Equipment Corporation, Intel Corporation and Xerox Corporation, "The Ethernet, A Local Area Network, Data Link Layer and Physical Layer Specifications," Version 1.0, September 30, 1980.

[IP, 1980]
"DOD Standard Internet Protocol," Editied by Jon Postel, January 1980.

[Metcalfe and Boggs, 1976]
R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed packet switching for local computer networks," CACM, vol. 19 no. 7, July 1976.

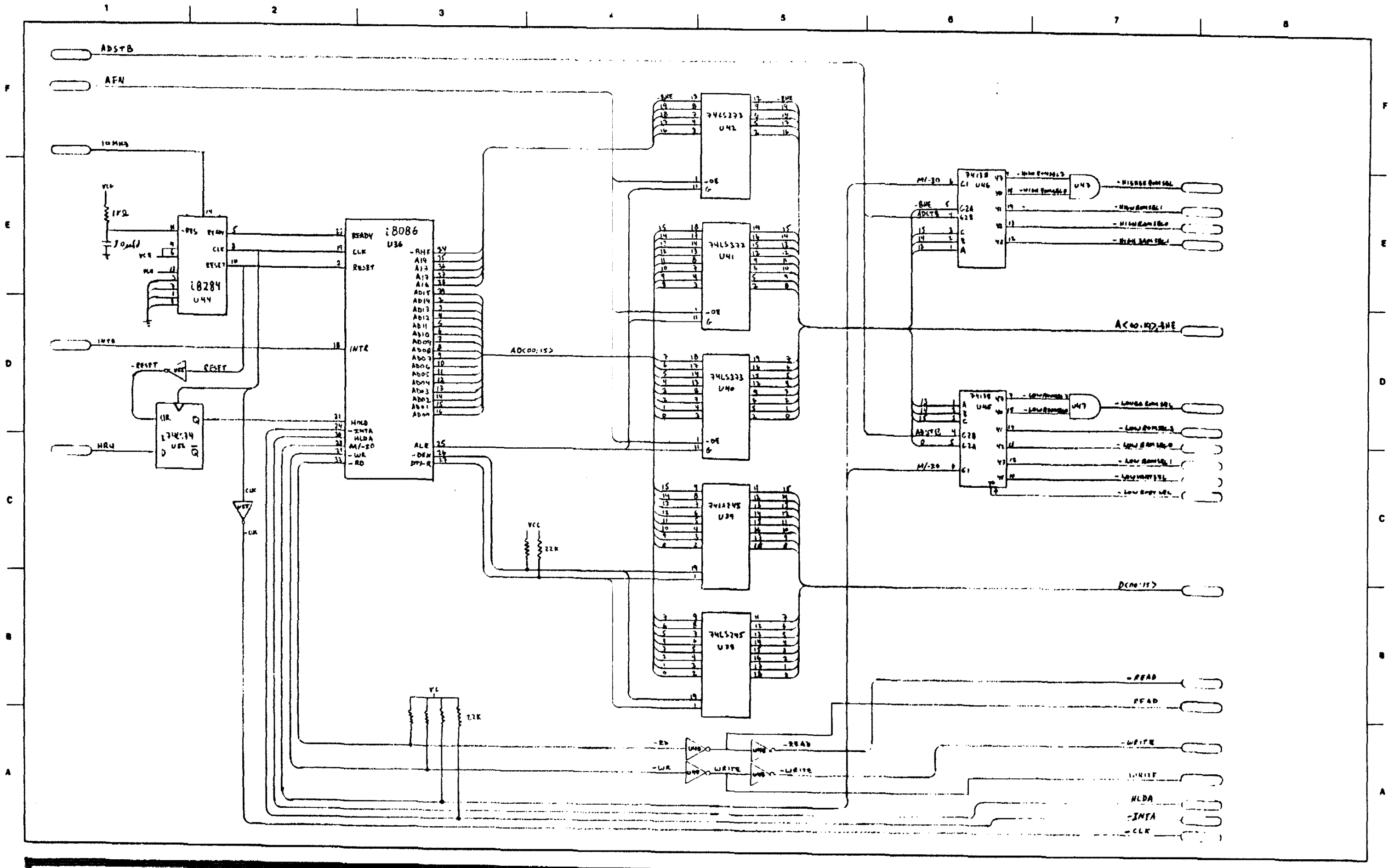
[Shoch and Hupp, 1979]
J.F. Shoch and J.A. Hupp, "Performance of an Ethernet Local Network -- A Preliminary Report," Local Area Communications Network Symposium, Mitre and NBS, May 1979.

[Shoch et al, 1980]
J.F. Shoch, D. Cohen and E.A. Taft, "Mutual Encapsulation of Internetwork Protocols," Trends and Applications: Computer Network Protocols, IEEE/NBS, May 1980.

[TCP, 1980]
"DOD Standard Transmission Control Protocol," Edited by Jon Postel, January 1980.

APPENDIX A

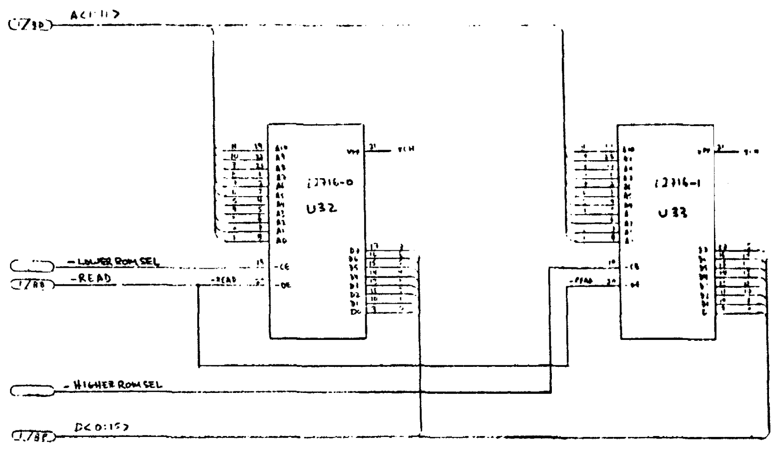
CIRCUIT SCHEMATICS



COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

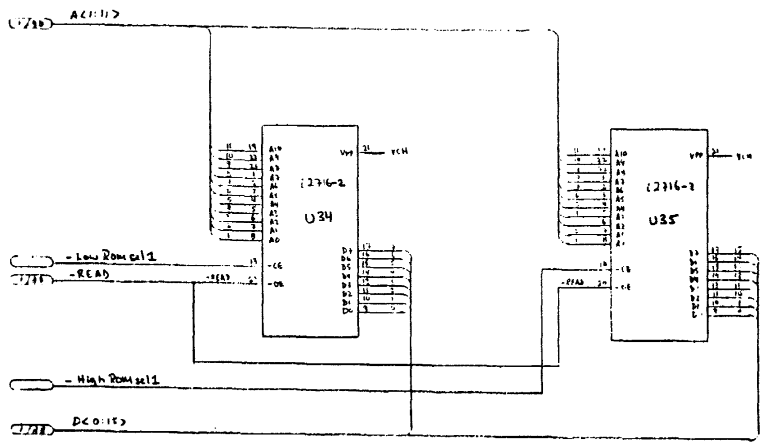
REV	DATE	DESCRIPTION	BY	CK'D	APVD

DESIGNED		SYSTEM ETHERNET	SHEET 1
DRAWN			
CHECKED		TITLE CPU	
APPROVED			
DATE APRIL 24, 1980		DRAWING NUMBER	
C.N.			OF 11 SHEETS

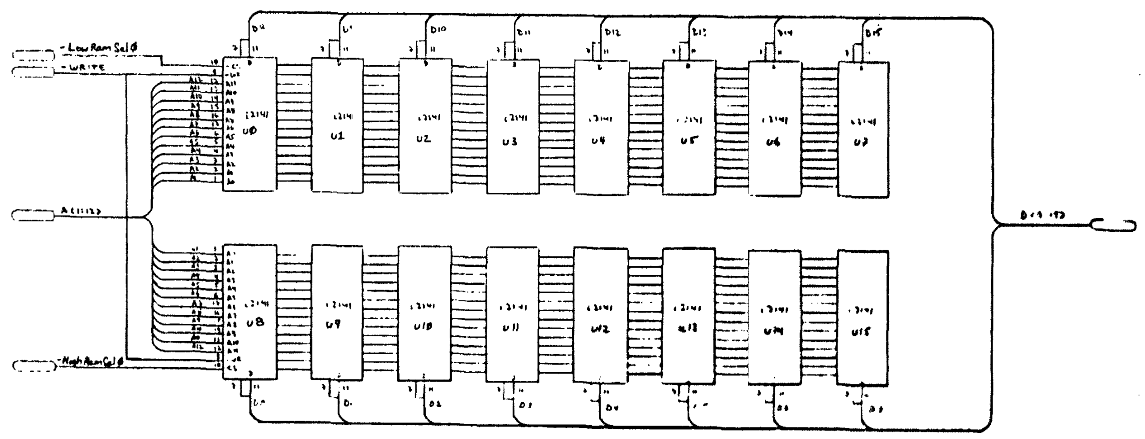


COMPUTER SCIENCE CALIFORNIA INSTITUTE OF TECHNOLOGY PASADENA, CALIFORNIA 91125		
DESIGNED D. S. Whelan	SYSTEM Ethernet	SHEET 2
DRAWN		
CHECKED	TITLE Low Bank Rom	
APPROVED		
DATE February 24 1981	DRAWING NUMBER	OF 11 SHEETS
C.N.		

REV	DATE	DESCRIPTION	BY	CK'D	APV'D



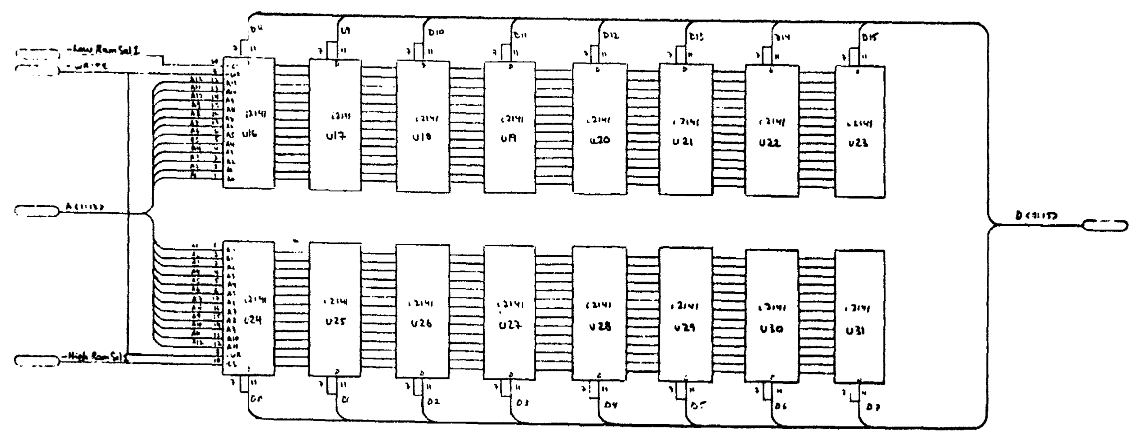
COMPUTER SCIENCE CALIFORNIA INSTITUTE OF TECHNOLOGY PASADENA, CALIFORNIA 91125	DESIGNED D.S. Whelan	SYSTEM Ethernet	SHEET 3
	DRAWN		
	CHECKED	TITLE High Bank Rom	
	APPROVED		
	DATE February 24, 1981	DRAWING NUMBER	
REV · DATE · DESCRIPTION · BY · CKD · APVD	C.M.		OF 11 SHEETS



COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

DESIGNED [illegible]	SYSTEM Ethernet	SHEET 4
DRAWN		
CHECKED	TITLE Low RAM	
APPROVED		
DATE February 24, [illegible]	DRAWING NUMBER	
C.A.		OF 11 SHEETS

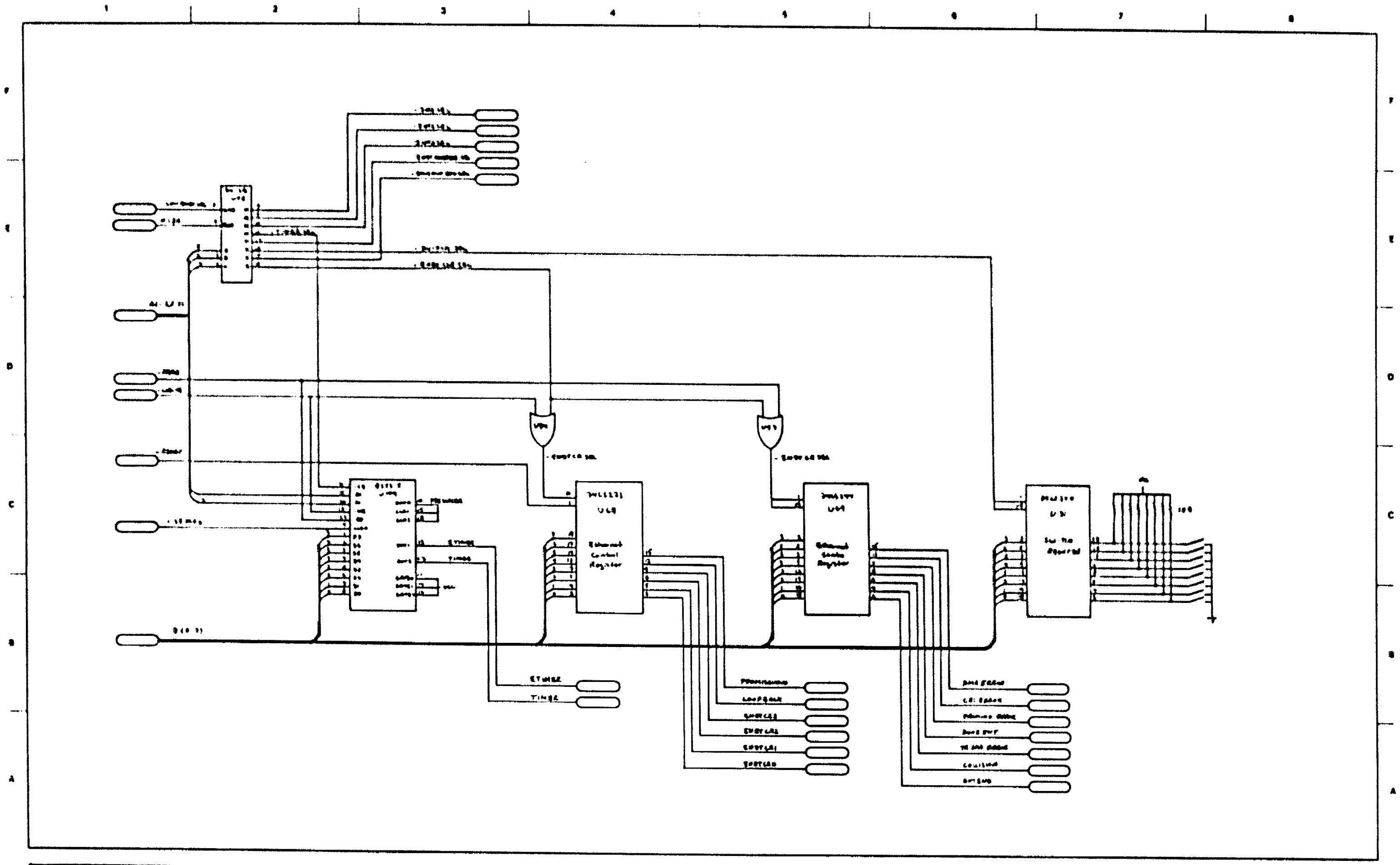
REV	DATE	DESCRIPTION	BY	CK'D	APP'D



COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CK'D	APVD

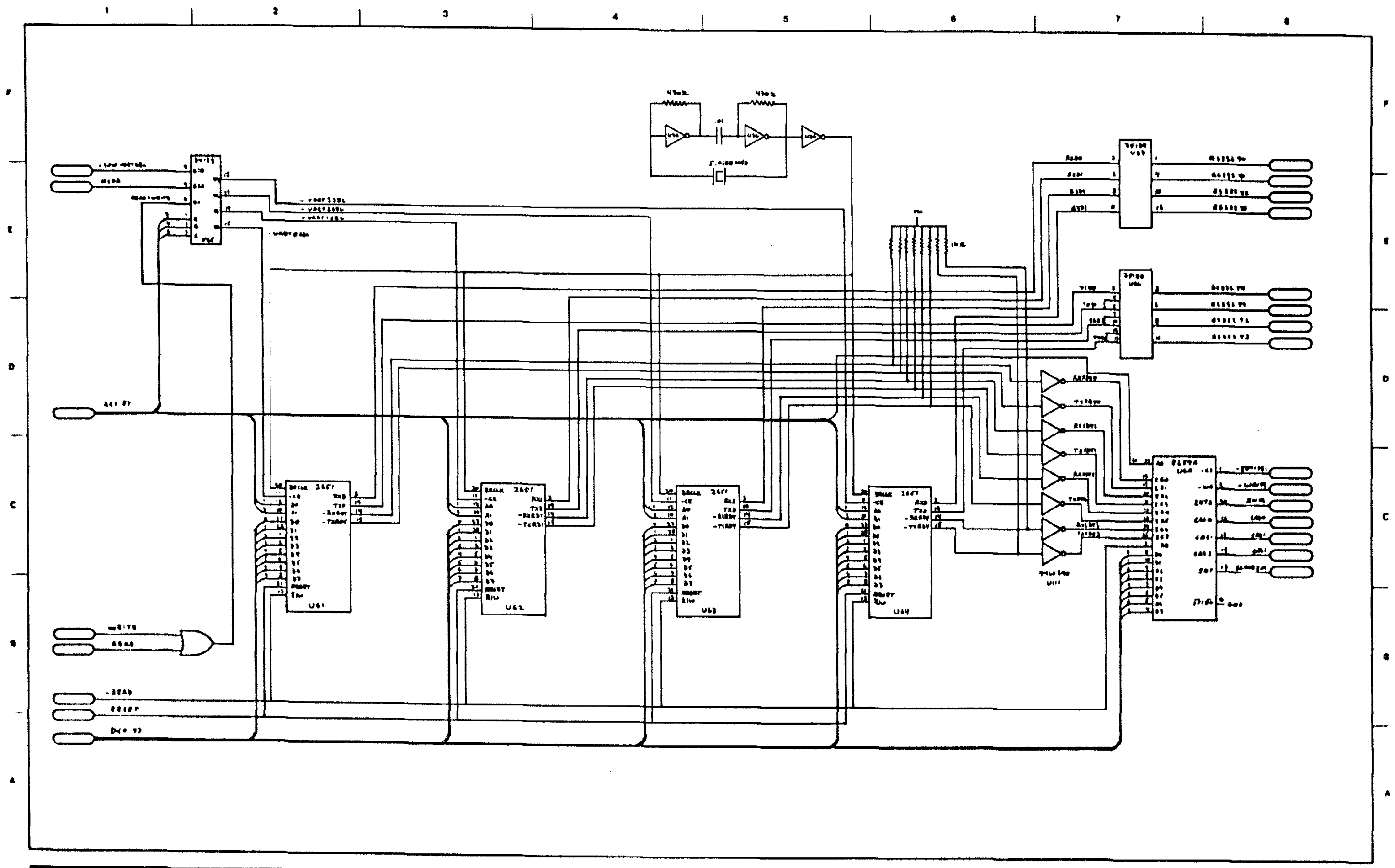
DESIGNED D.S. Whelan	SYSTEM Ethernet	SHEET 5
DRAWN		
CHECKED	TITLE High RAM	
APPROVED		
DATE February 27, 1981	DRAWING NUMBER	OF 11 SHEETS
C.N.		



COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

DESIGNED D. J. WHELAN	SYSTEM ETHERNET	SHEET 6
DRAWN		
CHECKED	TITLE REGISTERS & TIMER	
APPROVED		
DATE MARCH 25, 1981	DRAWING NUMBER	
C.N.		OF 11 SHEETS

REV	DATE	DESCRIPTION	BY	CK'D	APV'D



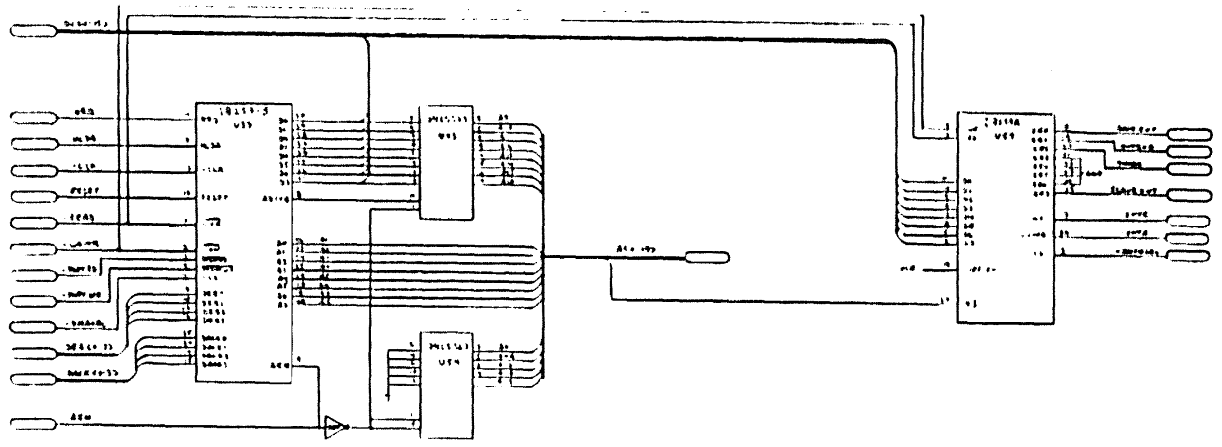
COMPUTER SCIENCE

CALIFORNIA INSTITUTE OF TECHNOLOGY

PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CK'D	APV'D

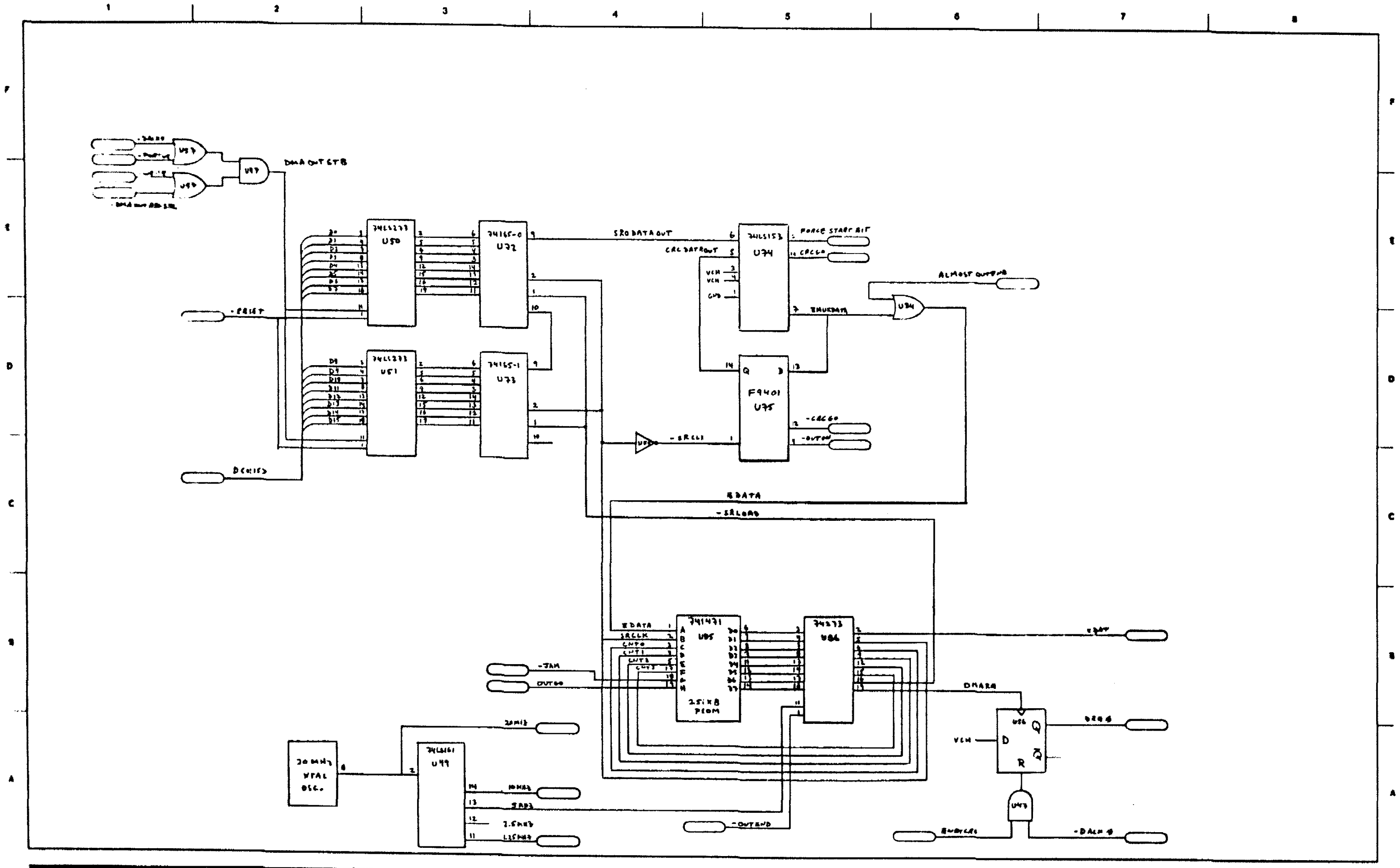
DESIGNED [illegible]	SYSTEM ETHERNET	SHEET 7
DRAWN		
CHECKED	TITLE UART'S	
APPROVED		
DATE [illegible]	DRAWING NUMBER	
C.N.		OF 11 SHEETS



COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CKD	APVD

DESIGNED [illegible]	SYSTEM ETHERNET	SHEET
DRAWN		8
CHECKED	TITLE DMA & INTERRUPT CONTROLLERS	
APPROVED		
DATE [illegible]	DRAWING NUMBER	
CN		OF 11 SHEETS



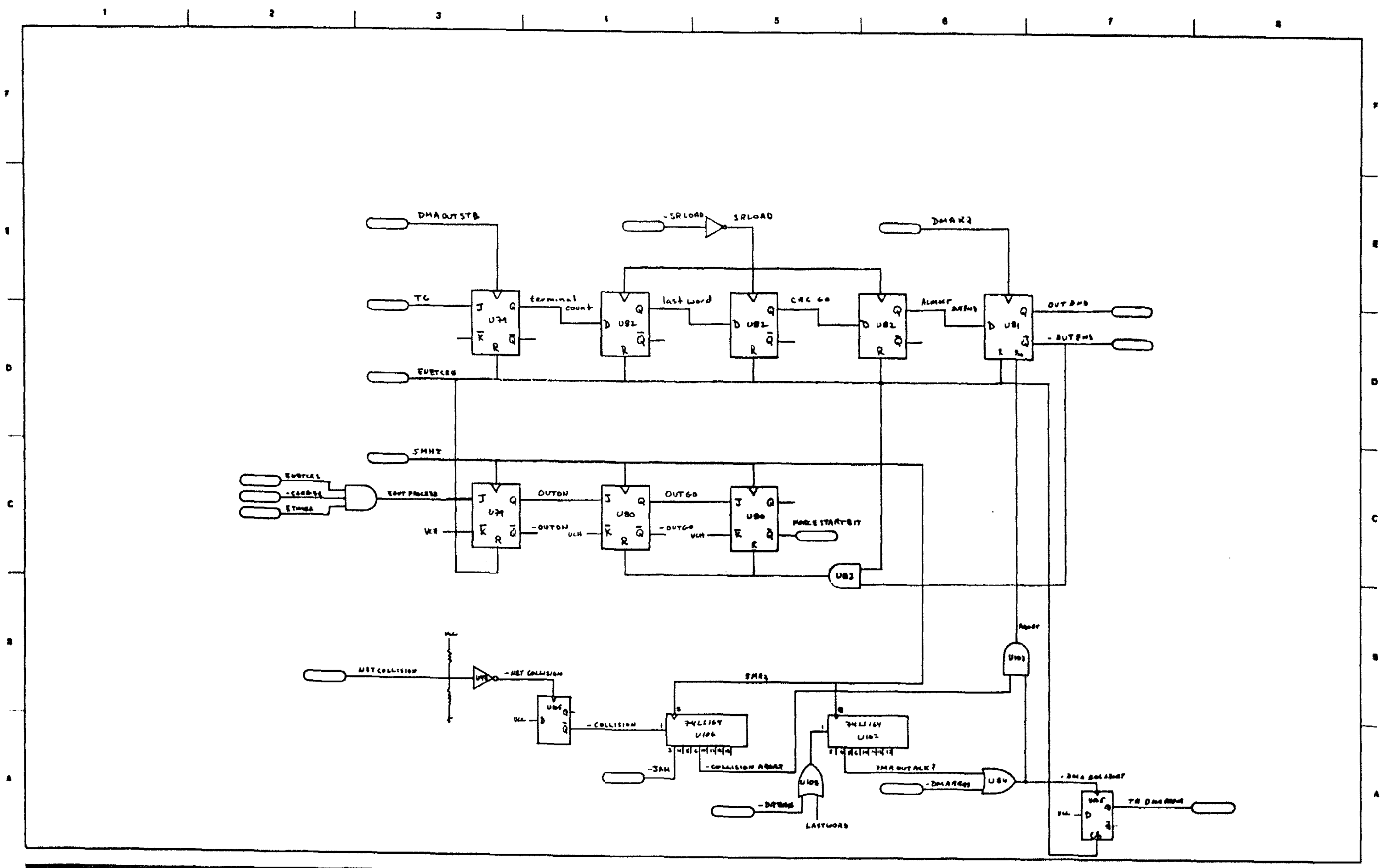
COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CK'D	APV'D

DESIGNED J. I. Winfield
DRAWN
CHECKED
APPROVED
DATE July 7, 1980
C.M.

SYSTEM Ethernet
TITLE Ethernet Transmitter
DRAWING NUMBER

SHEET 9 OF 11 SHEETS



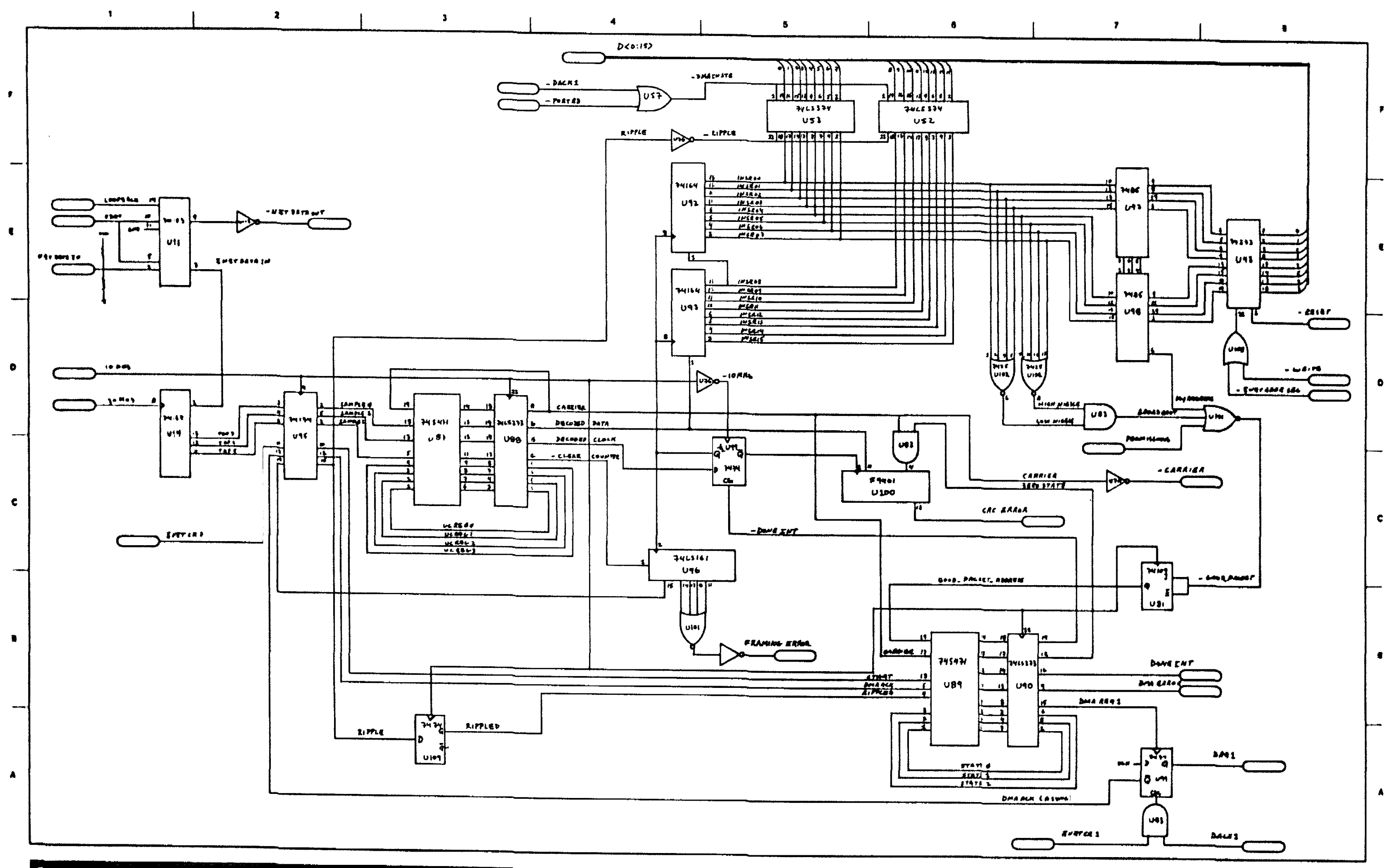
COMPUTER SCIENCE
CALIFORNIA INSTITUTE OF TECHNOLOGY
PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CK'D	APVD

DESIGNED D. S. WHELAN
DRAWN
CHECKED
APPROVED
DATE JULY 10, 1980
C.N.

SYSTEM ETHERNET
TITLE ETHERNET TRANSMITTER
DRAWING NUMBER

SHEET 10 OF 11 SHEETS



COMPUTER SCIENCE

CALIFORNIA INSTITUTE OF TECHNOLOGY

PASADENA, CALIFORNIA 91125

REV	DATE	DESCRIPTION	BY	CKD	APVD

DESIGNED D S WHELAN

DRAWN

CHECKED

APPROVED

DATE MARCH 17, 1981

C.N.

SYSTEM ETHERNET

TITLE ETHERNET RECEIVER

DRAWING NUMBER

SHEET 11

OF 11 SHEETS

APPENDIX B

MICROCODE DESCRIPTIONS

This Simula program emulates the manchester encoding portion of the Ethernet transmitter. The array rom contains the prom microcode and is initialized by the procedure init. The program reads data from a file called emul.dat and displays the resulting encoded output on the terminal. It creates two files. One is called eout.txt and contains a textual representation of the microcode. The other is called eout.rom and contains the Intel format code file for the prom programmer.

```
! This is <dan.thesis>eout.sim;
BEGIN
    EXTERNAL INTEGER PROCEDURE land, lnot, lor, lshift, lxor;
    REF(Outfile) ouf;
    REF(Infile) inf;
    INTEGER i,j,k,data,xdat,count,srpos,srdata;
    INTEGER ARRAY rom [0:255];
    TEXT Line;
    INTEGER PROCEDURE ones(len);
    INTEGER len;
    BEGIN
        INTEGER result, i;
        FOR i:= 1 STEP 1 UNTIL len DO result:= 2result+1;
        ones:= result
    END of procedure ones;
    INTEGER PROCEDURE extract(i,start,len);
    INTEGER i,start,len;
    extract:= land(lshift(i,-start),ones(len));
    INTEGER PROCEDURE set(i,start,len,j);
    INTEGER i, start, len, j;
    BEGIN
        i:= land(i,lnot(lshift(ones(len),start)));
        set:= lor(i,lshift(land(j,ones(len)),start))
    END of procedure set;
    PROCEDURE init;
    BEGIN
        INTEGER i;
        FOR i:= 128 STEP 1 UNTIL 128+63 DO
        BEGIN
            count:= extract(i,1,5);
            IF i>127 THEN
            rom[i]:= set(rom[i],0,1,lxor(extract(i,0,1),extract(i,1,1)));
            rom[i]:= set(rom[i],1,5,count+1);
            rom[i]:= set(rom[i],6,1,1);
            IF extract(i,1,5)=0 THEN rom[i]:= set(rom[i],6,1,0);
            IF extract(i,1,5)=1 THEN rom[i]:= set(rom[i],7,1,1);
        END;
    END of procedure init;
    ! Ok lets emulate the sequencer;
    PROCEDURE loadsr;
    BEGIN
        data:= inf.Inint;
    srpos:=15; srdata:= extract(data,srpos,1) END;
    PROCEDURE clocksr;
    IF srpos > 0 THEN
    BEGIN
        srpos:= srpos-1;
        srdata:= extract(data,srpos,1);
    END of procedure clocksr;
    init;
    BEGIN
        INTEGER latch,address;
        BOOLEAN crnext;
        inf:- NEW Infile("emul.dat");
        inf.Open(Blanks(80));
        srpos:= 15;
        srdata:=1;
        latch:= 0;
        address:= 0;
        i:= 0;
        WHILE NOT inf.Lastitem DO
        BEGIN
            IF crnext THEN Outimage;
            address:= set(address,1,5,Entier(latch/2)); ! get feedback terms;
            address:= set(address,0,1,srdata); ! get data term;
            address:= set(address,7,1,1); ! set go bit;
            ! address:= set(address,1,1,0); ! Is the xdat feedback term needed?
;
            latch:= rom[address]; ! get the next word;
            IF extract(latch,1,1)=1 THEN clocksr;
            IF extract(latch,6,1)=0 THEN loadsr;
            IF extract(latch,1,5)=2 THEN crnext:=TRUE
            ELSE crnext:=FALSE;
            Outint(extract(latch,0,1),1); !print data;
            IF Mod(i,2)=1 THEN Outtext(" ");
            IF extract(latch,7,1)=1 THEN Outtext(" DMA ");
            i:= i+1;
        END;
        Outimage;
    END;
    inf.Close;
    ouf:-NEW Outfile("eout.txt");
    ouf.Open(Blanks(80));
    Line:- Blanks(80);
    FOR i:=1 STEP 1 UNTIL 63 DO Line.Putchar('-');
    FOR i:= 0 STEP 32 UNTIL 255 DO
    BEGIN
        ouf.Outchar(Char(12));
        ouf.Outimage;
        ouf.Outtext("<GO><bit6><Cnt><Xclk><Data>");
        ouf.Outtext(" || <DMA>|<LSR>|<Cnt>|<Xclk>|<Xdat>");
        ouf.Outimage;
        FOR j:= 0 STEP 1 UNTIL 31 DO
        BEGIN
            ouf.Outtext(Line);
            ouf.Outint(extract(i+j,7,1),3);
            ouf.Outint(extract(i+j,6,1),6);
            ouf.Outint(extract(i+j,2,4),5);
            ouf.Outint(extract(i+j,1,1),6);
            ouf.Outint(extract(i+j,0,1),6);
            ouf.Outtext("  ||");
            ouf.Outint(extract(rom[i+j],7,1),4);
            ouf.Outtext("  |");
            ouf.Outint(extract(rom[i+j],6,1),3);
            ouf.Outtext("  |");
            ouf.Outint(extract(rom[i+j],2,4),3);
            ouf.Outtext("  |");
            ouf.Outint(extract(rom[i+j],1,1),4);
            ouf.Outtext("  |");
            ouf.Outint(extract(rom[i+j],0,1),4);
            ouf.Outtext("  |");
            ouf.Outimage;
        END;
    END;
    ouf.Close;
    ouf:- NEW Outfile("eout.rom");
    ouf.Open(Blanks(80));
    FOR i:= 0 STEP 8 UNTIL 255 DO
    BEGIN
        FOR j:=0 STEP 1 UNTIL 7 DO
        BEGIN ouf.Outint(rom[i+j],5); ouf.Outtext(".,") END;
        ouf.Outimage;
    END;
    ouf.Close;
END of program;
```

The second program simulates the entire Ethernet receiver. It contains a Class PROM which provides the prom's memory and a few usefull procedures for printing and dumping code files. The two prom's in this circuitry, the Manchester Decoder and the Manager prom are superclasses of class PROM. Each of these superclasses initializes the prom's memory. These initialization procedures are the algorithmic microcode descriptions. When the program is run, it prompts the user for several input words from the Ethernet. It turns these numbers into a serial bit stream which drives the simulator. The simulator displays the value of certain signals on the users terminal on a per clock tick basis. It also generates code files for the two proms.

```
! This is <dan.augat>newein.sim
BEGIN
    EXTERNAL INTEGER PROCEDURE land, lnot, lor, lshift, lxor;
    external text procedure frontstrip, upcase;
    external procedure outintel;
    INTEGER PROCEDURE ones(len);
    INTEGER len;
    BEGIN
        INTEGER result, i;
        FOR i:= 1 STEP 1 UNTIL len DO result:= 2result+1;
        ones:= result
    END of procedure ones;
    INTEGER PROCEDURE extract(i,start,len);
    INTEGER i,start,len;
    extract:= land(lshift(i,-start),ones(len));
    INTEGER PROCEDURE set(i,start,len,j);
    INTEGER i, start, len, j;
    BEGIN
        i:= land(i,lnot(lshift(ones(len),start)));
        set:= lor(i,lshift(land(j,ones(len)),start))
    END of procedure set;
    CLASS data;
    BEGIN
        INTEGER d;
    END of class data;
    ! This class emulates a 74ls164 serial in parallel out eight bit shift
    register;
    CLASS shiftregister(din);
    REF(data) din;
    BEGIN
        REF(data) ARRAY d[0:7];
        REF(data) dout;
        PROCEDURE clock;
        BEGIN
            INTEGER i;
            FOR i:= 7 STEP -1 UNTIL 1 DO d[i].d:= d[i-1].d;
            d[0].d:= din.d
        END of procedure clock;
        PROCEDURE print;
        BEGIN
            INTEGER i;
            FOR i:=0 STEP 1 UNTIL 7 DO outint(d[i].d,2)
        END of procedure print;
        ! init code;
        INTEGER i;
        FOR i:= 0 STEP 1 UNTIL 7 DO d[i]:- NEW data;
        dout:- NEW data;
        dout:- d[7];
    END of class shiftregister;
    ! This class emulates a 74ls174 hex flip flop;
    CLASS hexflipflop(d0,d1,d2,d3,d4,d5);
    REF(data) d0,d1,d2,d3,d4,d5;
    BEGIN
        REF(data) q0,q1,q2,q3,q4,q5;
        PROCEDURE clock;
        BEGIN
            q0.d:=d0.d;
            q1.d:=d1.d;
        q2.d:=d2.d;
        q3.d:=d3.d;
        q4.d:=d4.d;
        q5.d:=d5.d
    END of procedure clock;
    PROCEDURE print;
    BEGIN
        outint(q0.d,2);
        outint(q1.d,2);
        outint(q2.d,2);
        outint(q3.d,2);
        outint(q4.d,2);
        outint(q5.d,2)
    END of procedure print;
    ! init code;
    q0:- NEW data;
    q1:- NEW data;
    q2:- NEW data;
    q3:- NEW data;
    q4:- NEW data;
    q5:- NEW data;
END of class hexflipflop;
CLASS ff(d);
REF(data) d;
BEGIN
    REF(data) q;
    PROCEDURE clock;
    q.d:=d.d;
    ! init code;
    q:- NEW data;
END of class ff;
CLASS prom;             ! superclass of all 74s471 proms;
BEGIN
    REF(data) a0,a1,a2,a3,a4,a5,a6,a7;
    REF(data) d0,d1,d2,d3,d4,d5,d6,d7;
    INTEGER ARRAY da[0:255];
    PROCEDURE propagate;
    BEGIN
        INTEGER address,dataword;
        address:= set(address,0,1,a0.d);
        address:= set(address,1,1,a1.d);
        address:= set(address,2,1,a2.d);
        address:= set(address,3,1,a3.d);
        address:= set(address,4,1,a4.d);
        address:= set(address,5,1,a5.d);
        address:= set(address,6,1,a6.d);
        address:= set(address,7,1,a7.d);
        dataword:= da[address];
        d0.d:= extract(dataword,0,1);
        d1.d:= extract(dataword,1,1);
        d2.d:= extract(dataword,2,1);
        d3.d:= extract(dataword,3,1);
        d4.d:= extract(dataword,4,1);
        d5.d:= extract(dataword,5,1);
        d6.d:= extract(dataword,6,1);
        d7.d:= extract(dataword,7,1)
    END of procedure propagate;
    procedure emit(f);
    ref(outfile) f;
    outintel(f,da,255);
    ! init code;
    d0:- NEW data;
    d1:- NEW data;
    d2:- NEW data;
    d3:- NEW data;
    d4:- NEW data;
    d5:- NEW data;
    d6:- NEW data;
    d7:- NEW data;
END of class prom;
prom CLASS decoder;
BEGIN
    PROCEDURE list(f);
    REF(outfile) f;
    BEGIN
        INTEGER i;
        f.outtext("<carrier> | <sample> | <count> ||");
        f.outtext("<carrier> | <data> | <clock> | <?> | <count>");
        f.outimage;
        f.outtext("================================");
        f.outtext("==========================================");
        f.outimage;
        FOR i:=0 STEP 1 UNTIL 255 DO
        BEGIN
            f.outint(extract(i,7,1),9);
            f.outint(extract(i,4,3),11);
            f.outint(extract(i,0,4),10);
            f.outint(extract(da[i],7,1),12);
            f.outint(extract(da[i],6,1),9);
            f.outint(extract(da[i],5,1),10);
            f.outint(extract(da[i],4,1),6);
            f.outint(extract(da[i],0,4),10);
            f.outimage
        END;
    END of procedure list;
    ! init code;
    BEGIN
        INTEGER address;
        BOOLEAN PROCEDURE carrier;
        carrier:= IF extract(address,7,1)=1 THEN TRUE ELSE FALSE;
        INTEGER PROCEDURE sample;
        sample:= extract(address,4,3);
        INTEGER PROCEDURE count;
        count:= extract(address,0,4);
        PROCEDURE scarrier;
        da[address]:= set(da[address],7,1,1);
        PROCEDURE clearcounter;
        da[address]:= set(da[address],4,1,0);
        PROCEDURE sevent(i);
        INTEGER i;
        BEGIN
            IF i=1 THEN da[address]:= set(da[address],6,1,1);
            da[address]:= set(da[address],5,1,1);
        END of sevent;
        PROCEDURE scount(i);
        INTEGER i;
        da[address]:= set(da[address],0,4,i);
        FOR address:= 0 STEP 1 UNTIL 255 DO
        BEGIN
            da[address]:= set(da[address],4,1,1);  !  set  counter  bit;
            IF NOT carrier THEN
            BEGIN
                IF (sample=0 OR sample=7) THEN scount(0) ! idle;
                ELSE IF (sample=1 OR sample=3) THEN
                BEGIN ! start of packet;
                    scarrier;
                    clearcounter;
                    if sample=1 then scount(0) else scount(1);
                    sevent(1)
                END
                ELSE  scarrier;  !  collision;
            END
            ELSE BEGIN   ! carrier;
                IF ((sample=1 OR sample=6) AND (count=0 OR count=1))
                OR ((sample=3 OR sample=4) AND (count=1 OR count=2)) THEN
                BEGIN           ! collison;
                    scarrier;
                    scount(count+2)
                END;
                IF (sample=1 OR sample=6)
                AND (count>=2 AND count<=3) THEN ! setup;
                BEGIN scarrier; scount(count+2) END;
                IF (sample=3 OR sample=4)
                AND (count>=3 AND count<=5) THEN ! setup;
                BEGIN scarrier; scount(count+2) END;
                IF sample=1 AND (count>=4 AND count<=9) THEN ! 1 data;
                BEGIN scarrier; sevent(1); scount(0) END;
                IF sample=3 AND (count>=6 AND count<=10) THEN ! 1 data;
                BEGIN scarrier; sevent(1); scount(1) END;
                IF sample=4 AND (count>=6 AND count<=10) THEN ! 0 data;
                BEGIN scarrier; sevent(0); scount(0) END;
                IF sample=6 AND (count>=4 AND count<=9) THEN ! 0 data;
                BEGIN scarrier; sevent(0);scount(1) END;
                IF (sample=1 OR sample=6) AND (count>=10 AND count<=15) THEN
                BEGIN scarrier; scount(0) END; ! collision;
                IF (sample=3 OR sample=4) AND (count>=11 AND count<=15) THEN
                BEGIN scarrier; scount(1) END;
                IF (sample=2 OR sample=5) AND (count<=15) THEN ! collision;
                BEGIN scarrier; scount(count+2) END;
                IF (sample=0 OR sample=7) AND (count<=9) THEN ! active;
                BEGIN scarrier; scount(count+2) END;
                IF (sample=7) AND (count>=12 AND count<=15) THEN ! jam;
                BEGIN scarrier; scount(0) END;
                IF sample=0 AND (count>=12 AND count<=15) THEN ! end of
                packet;
                BEGIN scount(0) END;
            END;
        END;
    END;
END of prom class decoder;
CLASS octalff(d0,d1,d2,d3,d4,d5,d6,d7);
REF(data) d0,d1,d2,d3,d4,d5,d6,d7;
BEGIN
    REF(data) q0,q1,q2,q3,q4,q5,q6,q7;
    PROCEDURE clock;
    BEGIN
        q0.d:=d0.d;
        q1.d:=d1.d;
        q2.d:=d2.d;
        q3.d:=d3.d;
        q4.d:=d4.d;
        q5.d:=d5.d;
        q6.d:=d6.d;
        q7.d:=d7.d
    END of procedure clock;
    PROCEDURE print;
    BEGIN
        outint(q0.d,2);
        outint(q1.d,2);
        outint(q2.d,2);
        outint(q3.d,2);
        outint(q4.d,2);
        outint(q5.d,2);
        outint(q6.d,2);
        outint(q7.d,2)
    END of procedure print;
    ! init code;
    q0:- NEW data;
    q1:- NEW data;
    q2:- NEW data;
    q3:- NEW data;
    q4:- NEW data;
    q5:- NEW data;
    q6:- NEW data;
    q7:- NEW data;
END of class octalff;
CLASS counter;
BEGIN
    INTEGER count;
    REF(data) ripple;
    PROCEDURE clear;
    count:= 0;
    PROCEDURE clock;
    begin
    IF count=15 THEN ripple.d:=1;
    count:= mod(count+1,16);
    end;
    ! init code;
    ripple:- NEW data;
END of class counter;
prom CLASS manager;
BEGIN
    PROCEDURE list(f);
    REF(outfile) f;
    BEGIN
        INTEGER i;
        f.outtext("<st>|<car>|<dma>|<cnt16>|<state>||");
        f.outtext("<Sint>|<Dint>|<load>|<inhb>|<dmaerr>|<state>");
        f.outimage;
        f.outtext("================================");
        f.outtext("=========================================");
        f.outimage;
        FOR i:=0 STEP 1 UNTIL 127 DO
        BEGIN
            f.outint(extract(i,6,1),3);
            f.outint(extract(i,5,1),5);
            f.outint(extract(i,4,1),5);
            f.outint(extract(i,3,1),7);
            f.outint(extract(i,0,3),8);
            f.outint(extract(da[i],7,1),8);
            f.outint(extract(da[i],6,1),6);
            f.outint(extract(da[i],5,1),7);
            f.outint(extract(da[i],4,1),7);
            f.outint(extract(da[i],3,1),9);
            f.outint(extract(da[i],0,3),8);
            f.outimage;
        END;
    END of procedure list;
! This prom is used in the 2nd microsequencer. It controlls interrupt
generation, loading of the data from the shift registers into the port
and requesting a DMA transfer, reseting the CRC checker chip and reporting
DMA channel errors when the previous transfer was not complete when the
next one was demanded.
;
    ! init code;
    BEGIN
        INTEGER address;
!   output definitions:
        -done interrupt request:    bit 7
        done interrupt request:     bit 6
        dma request:                bit 5
        zerostate                  bit 4
        dma error:                  bit 3
        state feedback terms:       bits 0-2
    if we don't have a normal end flag bit in the status register then
    we must generate two types of interrupts. One is a first byte received
    interrupt and the other is an end of packet interrupt.
    Address Definitions:
        goodpacketaddress         A7
        enetcr[?] "start":          A6
        carrier:                    A5
        dmaack:                     A4
        count=16:                   A3
        state:                      A0-A2
;
        boolean procedure mine;
        mine:= if extract(address,7,1)=0 then false else true;
        BOOLEAN PROCEDURE start;
        start:= IF extract(address,6,1)=0 THEN FALSE ELSE TRUE;
        BOOLEAN PROCEDURE carrier;
        carrier:= IF extract(address,5,1)=0 THEN FALSE ELSE TRUE;
        BOOLEAN PROCEDURE dmaack;
        dmaack:= IF extract(address,4,1)=0 THEN FALSE ELSE TRUE;
        BOOLEAN PROCEDURE cnt16;
        cnt16:= IF extract(address,3,1)=1 THEN TRUE ELSE FALSE;
        INTEGER PROCEDURE state;
        state:= extract(address,0,3);
        PROCEDURE doneint;       ! set doneint and clear -doneint;
        da[address]:= set(da[address],6,2,1);
        PROCEDURE loadport;
```

```
da[address]:= set(da[address],5,1,1);
PROCEDURE zerostate;
da[address]:= set(da[address],4,1,1);
PROCEDURE dmaerror;
da[address]:= set(da[address],3,1,1);
PROCEDURE nextstate(i);
INTEGER i;
da[address]:= set(da[address],0,3,i);
FOR address:=0 STEP 1 UNTIL 255 DO
begin
da[address]:= set(da[address],7,1,1); ! complement doneint;
! State 0:                                   ;
!   to state 1 on carrier and not start      ;
!   to state 2 on carrier and start          ;
!   otherwise to state 0                     ;
IF state=0 THEN
BEGIN
    zerostate;
    IF carrier THEN
    BEGIN
        IF NOT start THEN nextstate(1) ELSE nextstate(2)
    END
    ELSE nextstate(0)
END
ELSE IF state=1 THEN
! State 1:                                   ;
!   to state 0 on not carrier else to state 1 ;
BEGIN
    IF NOT carrier THEN nextstate(0) ELSE nextstate(1);
END
ELSE IF state=2 THEN
! State 2:                                   ;
!   to state 0 on not start                  ;
!   to state 0 on not carrier                ;
!   to state 3 on cnt16 and carrier          ;
!   to state 2 on not cnt16                  ;
BEGIN
    IF NOT start THEN nextstate(0)
    ELSE IF NOT carrier THEN nextstate(0)
    ELSE IF cnt16 THEN nextstate(3)
    ELSE nextstate(2)
END
ELSE IF state=3 THEN
! State 3:                                   ;
!   to state 0 on not start                  ;
!   to state 0 on not carrier                ;
!   to state 3 on cnt16                      ;
!   to state 1 on not cnt16 and not mine     ;
!   to state 5 on not cnt16 and mine         ;
!   else to state 3                          ;
BEGIN
    IF NOT start THEN nextstate(0)
    ELSE IF NOT carrier THEN nextstate(0)
    else if cnt16 then nextstate(3)
    ELSE if mine then nextstate(5)
    else nextstate(1)
END
ELSE IF state=4 THEN
```

```
        ! State 4:                                          ;
        !   to state 0 on not start                         ;
        !   to state 7 on not carrier                       ;
        !   to state 5 on cntl6 and dmaack                  ;
        !   to state 6 on cntl6 and not dmaack              ;
        !   ELSE TO state 4                                 ;
        BEGIN
            IF NOT start THEN nextstate(0)
            ELSE IF NOT carrier THEN nextstate(7)
            ELSE IF cntl6 AND dmaack THEN nextstate(5)
            ELSE IF cntl6 AND NOT dmaack THEN nextstate(6)
            ELSE nextstate(4)
        END
        ELSE IF state=5 THEN
        ! State 5:                                          ;
        !   to state 0 on not start                         ;
        !   to state 7 on not carrier                       ;
        !   to state 5 on cntl6                             ;
        !   ELSE TO state 4                                 ;
        !   LOAD PORT                                       ;
        BEGIN
            IF NOT start THEN nextstate(0)
            ELSE IF NOT carrier THEN nextstate(7)
            else if cntl6 then nextstate(5)
            ELSE nextstate(4);
            loadport
        END
        ELSE IF state=6 THEN
        ! State 6:                                          ;
        !   to state 0 on not start                         ;
        !   ELSE TO state 6                                 ;
        !   DMAERROR and DONEINT                            ;
        BEGIN
            IF NOT start THEN nextstate(0) ELSE nextstate(6);
            dmaerror;
            doneint
        END
        ELSE IF state=7 THEN
        ! State 7:                                          ;
        !   to state 0 on not start                         ;
        !   ELSE TO state 7                                 ;
        !   DONEINT                                         ;
        BEGIN
            IF NOT start THEN nextstate(0) ELSE nextstate(7);
            doneint;
        END;
        end;
    END of init code;
END of prom class manager;
CLASS enetdecoder(din);
REF(data) din;
BEGIN
    REF(decoder) p;
    REF(octalff) ucreg,mreg;
    REF(shiftregister) sr,sr0,srl;
    REF(hexflipflop) inreg;
    REF(ff) ff0;
    REF(counter) c;
REF(manager) m;
BOOLEAN even;
PROCEDURE clock;
IF even THEN
BEGIN
    mreg.clock;      ! latch 2nd rom data;
    m.propagate;
    ucreg.clock;     ! latch rom data into ucreg;
    inreg.clock;     ! latch input data into inreg;
    p.propagate;     ! let rom code propagate;
    sr.clock;        ! shift data in;
    even := FALSE;
    IF ucreg.q4.d=0 THEN c.clear;
END
ELSE BEGIN
    IF c.ripple.d=1 THEN c.ripple.d:=0;
    ff0.clock;       ! clock ff0 on -10mhz;
    IF ff0.q.d=1 THEN
    BEGIN
        c.clock;
        sr1.clock;
        sr0.clock
    END;
    sr.clock;
    IF ucreg.q4.d=0 THEN c.clear;
    even:= TRUE
END;
PROCEDURE print;
BEGIN
    outtext("enetin:"); outint(din.d,2);
    outtext("   sample:");
    outint(p.a6.d,2); outint(p.a5.d,2); outint(p.a4.d,2);
    outtext("      carrier:"); outint(ucreg.d7.d,2);
    outtext("   data:"); outint(ucreg.d6.d,2);
    outtext("   clock:"); outint(ucreg.d5.d,2);
    outimage;
END of procedure print;
! init code;
even := TRUE;
sr:- NEW shiftregister(din);
inreg:- NEW hexflipflop(sr.d[7],sr.d[6],sr.d[5],
sr.d[4],sr.d[3],sr.d[2]);
p:- NEW decoder;
ucreg:- NEW octalff(p.d0,p.d1,p.d2,p.d3,p.d4,p.d5,p.d6,p.d7);
p.a0:- ucreg.q0;
p.a1:- ucreg.q1;
p.a2:- ucreg.q2;
p.a3:- ucreg.q3;        ! count feedback terms;
p.a4:- inreg.q2;
p.a5:- inreg.q1;
p.a6:- inreg.q0;        ! input sample terms;
p.a7:- ucreg.q7;        ! carrier feeedback term;
ff0:- NEW ff(ucreg.q5); ! delay clk by 1/2 clock;
sr0:- NEW shiftregister(ucreg.q6); ! hook data up to sr;
sr1:- NEW shiftregister(sr0.d[7]); ! link them together;
c:- NEW counter;
m:- NEW manager;
mreg:- NEW octalff(m.d0,m.d1,m.d2,m.d3,m.d4,m.d5,m.d6,m.d7);
```

```
    m.a0:- mreg.q0;
    m.a1:- mreg.q1;
    m.a2:- mreg.q2;          ! connect up state terms;
    m.a3:- c.ripple;
    m.a4:- NEW data;
    m.a4.d:=1;
    m.a5:- ucreg.q7;         ! carrier term;
    m.a6:- NEW data;
    m.a6.d:=1;
    m.a7:- NEW data;
END of class enetdecoder;
PROCEDURE make74472(f,rom);
REF(outfile) f;
INTEGER ARRAY rom;
BEGIN
    INTEGER ARRAY newrom[0:511];
    integer i;
    INTEGER PROCEDURE convadd(i);
    INTEGER i;
    BEGIN
        INTEGER j;
        j:= set(j,0,5,extract(i,0,5));
        j:= set(j,6,3,extract(i,5,3));
        convadd:= j
    END of procedure convadd;
    FOR i:=0 STEP 1 UNTIL 255 DO newrom[convadd(i)]:= rom[i];
    outintel(f,newrom,511)
END of procedure make74472;
! test program;
REF(enetdecoder) dec;
REF(data) d;
REF(outfile) ouf;
INTEGER number;
TEXT idat,car,odat,clk,dclk,cnt,clrcnt,st;
text sint,dint,load,inh,dma,t;
character char;
PROCEDURE make(i);
INTEGER i;
BEGIN
    INTEGER j;
    d.d:=i;
    FOR j:=1 STEP 1 UNTIL 4 DO
    BEGIN
        dec.clock;
        IF i=0 THEN idat.putchar('0') ELSE idat.putchar('1');
        IF dec.ucreg.q7.d=1 THEN car.putchar('1') ELSE car.putchar(' ');
        IF dec.ucreg.q6.d=1 THEN odat.putchar('1') ELSE odat.putchar('0');
        IF dec.ucreg.q5.d=1 THEN clk.putchar('1') ELSE clk.putchar(' ');
        IF dec.ff0.q.d=1 THEN dclk.putchar('1') ELSE dclk.putchar(' ');
        IF dec.ucreg.q4.d=1 THEN clrcnt.putchar('1')
        ELSE clrcnt.putchar('0');
        cnt.sub(cnt.pos,1).putint(dec.c.count);
        cnt.setpos(cnt.pos+1);
        st.sub(st.pos,1).putint(
        dec.mreg.q0.d+(2dec.mreg.q1.d)+(4dec.mreg.q2.d));
        st.setpos(st.pos+1);
        if dec.mreg.q7.d=1 then sint.putchar('1') else sint.putchar(' ');
        if dec.mreg.q6.d=1 then dint.putchar('1') else dint.putchar(' ');
```

```
            if dec.mreg.q5.d=1 then load.putchar('1') else load.putchar(' ');
            if dec.mreg.q4.d=1 then inh.putchar('1') else inh.putchar(' ');
            if dec.mreg.q3.d=1 then dma.putchar('1') else dma.putchar(' ');
        END;
    END of procedure make;
    d:- NEW data;
    dec:- NEW enetdecoder(d);
    !   ouf:- NEW outfile("enetin.rom2");
    !   ouf.open(blanks(80));
    !   dec.m.list(ouf);
    !   ouf.close;
    t:- sysin.image;
    outtext("Do you want new prom code files: (No) ");
    breakoutimage;
    inimage;
    t:- upcase(frontstrip(t).sub(1,1));
    if t = "Y" then
        begin
        ouf:- new outfile("decoder.pll");
        ouf.open(blanks(80));
        outtext("Making decoder.pll"); outimage;
      !  dec.p.emit(ouf);
        make74472(ouf,dec.p.da);
        ouf.close;
        ouf:- new outfile("manage.pll");
        ouf.open(blanks(80));
        outtext("Making manage.pll"); outimage;
      !  dec.m.emit(ouf);
        make74472(ouf,dec.m.da);
        ouf.close;
        end;
    image:- blanks(80);
    WHILE TRUE DO
    BEGIN
        INTEGER i,j;
        idat:- blanks(560);
        car:- blanks(560);
        odat:- blanks(560);
        clk:- blanks(560);
        dclk:- blanks(560);
        clrcnt:- blanks(560);
        cnt:- blanks(560);
        sint:- blanks(560);
        dint:- blanks(560);
        load:- blanks(560);
        inh:- blanks(560);
        dma:- blanks(560);
        st:- blanks(560);
        make(1);
        inimage;
        for j:=1 step 1 until 3 do
        begin
        outtext("input a number> "); breakoutimage;
        number:= inint;
        FOR i:=0 STEP 1 UNTIL 15 DO
        BEGIN
            IF extract(number,i,1)=0 THEN make(1) ELSE make(0);
            make(extract(number,i,1));
        END;
        end;
        make(0); make(0); make(0); make(0); make(0); make(0);
        outimage;
        for j:=0 step 1 until 7 do
        begin
            integer k;
        k:= (70j)+1;
        if idat.sub(k,1) NE " " then
        begin
        outtext("in data: "); breakoutimage; outtext(idat.sub(k,70));
        outimage;
        outtext("carrier: "); breakoutimage; outtext(car.sub(k,70));
        outimage;
        outtext("outdata: "); breakoutimage; outtext(odat.sub(k,70));
        outimage;
        outtext("clock:  "); breakoutimage;  outtext(clk.sub(k,70));
        outimage;
        outtext("srclock: "); breakoutimage; outtext(dclk.sub(k,70));
        outimage;
        outtext("clrcnt: "); breakoutimage;  outtext(clrcnt.sub(k,70));
        outimage;
        outtext("counter: "); breakoutimage; outtext(cnt.sub(k,70));
        outimage;
        outtext("state: "); breakoutimage;  outtext(st.sub(k,70)); outimage;
        outtext("-doneint:"); breakoutimage; outtext(sint.sub(k,70)); outimage;
        outtext("done int:"); breakoutimage; outtext(dint.sub(k,70)); outimage;
        outtext("    dma:"); breakoutimage; outtext(load.sub(k,70)); outimage;
        outtext("zerostat:"); breakoutimage; outtext(inh.sub(k,70)); outimage;
        outtext("dmaerror:"); breakoutimage; outtext(dma.sub(k,70)); outimage;
        outtext("shift registers:");
        dec.sr0.print; dec.srl.print;
        outimage;
        outimage;
        end;
        end;
    END;
END of program;
```

APPENDIX C

ETHERNET PROTOCOL DRIVERS

```
;
; ENPROC.ASM
;
;           This code implements the Ethernet input and output processes,
; EOUT and EIN. TINT is the transmitter interrupt processes while RINT is
; the receiver interrupt process.
;
        name    enproc
        title   'definitions'
        list    b,e
        nlist   m
;
        dseg
trials: dsw     1               ; storage word for counting transmissions
paddr:  dsw     1               ; storage word for holding packets address
plen:   dsw     1               ; storage word for holding packets length
tgood:  dsw     1               ; number of good packet transmissions
terrs:  dsw     1               ; number of bad packet transmissions
ecrsav: ds      1               ; byte for saving copy of enetcsr
dmasav: ds      1               ; byte for saving copy of dmacsr
rgood:  dsw     1               ; number of good packet receptions
rerrs:  dsw     1               : number of bad packet receptions
raddr:  dsw     1               ; storage word for receiver packet address
rlen:   dsw     1               ; storage word for receiver packet length
;
;   Queues:
;
        queue   eoutiq,16       ; queue for sending ethernet packets
        queue   einoq,16        ; queue for receiving ethernet packets
;
;
        ejec
        cseg
        extrn   twait,tsucc,tfail,rwait,rsucc,eoflag,erflag
        extrn   enetsr,enetcr,dmacsr,dmach0,dmach1,ocw1a,ocw2a,dmareg
        extrn   tod,tcr,tcl,nett,netr
        extrn   readq,writeq,initq
        extrn   alloc,dalloc,ialloc,tt0in,tt0out
        public  eoproc,eout,ein,einit,up,userpr
;
```

```
;
; EINIT
;       This procedure initializes the ethernet harware, dma channel and
; some save locations before turning on the interrupts.
;
einit:  mov     ax, 16
        mov     bp, eoutiq
        call    initq
        mov     bp, einoq
        call    initq
        mov     ax, 2
        mov     bp, userp
        call    initq                   ; initialize the queues
        movb    enetcr, 0
        movb    ecrsav, 0
        movb    dmacsr, 0
        movb    dmasav, 0
        mov     tgood, 0
        mov     terrs, 0
        movb    ocwla, 78H              ; turn on timer and tint interrupts
        ret
;
        ejec
;
;
; EOUT
;       This is the Ethernet Transmitter Process. It reads a two word
; message from its input buffer, formatted as <buffer-pointer,return-
; queue-pointer>. It transmitts the packet and returns a message formatted
; as <status-word,buffer-pointer> to the calling process via return-queue.
;
eout:   mov     ax, 2                   ; the message length is 2
        mov     bp, eoutiq              ; bp points to eout input queue
        call    readq                   ; read 2 words from the queue
        pop     bx                      ; get the buffer address in bx
;
; transmit the packet
;
        mov     ax,bx                   ; put buffer address in ax
        add     ax,[bx+2]               ; ax:= buffer address + data offset
        mov     paddr,ax                ; tell interrupt process where it is
        mov     ax,[bx+4]               ; ax:= length of packet
        dec     ax                      ; ax:= length of packet - 1
        mov     plen,ax                 ; tell interrupt process how long it is
        movb    eoflag, twait           ; set eoflag to twait
        mov     trials, 0               ; set trials to 0
        int     21H                     ; activate the transmit interrupt
        suspend nett                    ; wait for eoflag ne twait
;
; return status and packet pointer to sender
;
        pop     bp                      ; get senders queue address in bp
        push    bx                      ; push address of buffer
        movb    al,eoflag               ; get status byte in al
        push    ax                      ; push status
        mov     ax, 2                   ; get message length in ax
        call    writeq                  ; write the message
        jmp     eout
```

```
;
        ejec
;
; EIN
;       Ethernet Input Process
; EIN grabs a buffer, lets the interrupt process know where it is then
; fakes an unsuccessful interrupt to initialize the dma channel and ethernet
; input hardware.
;
ein:    call    alloc               ; get a buffer in ram
        mov     [bp+2], 6           ; data starts with the 4th word
        mov     ax,bp               ; get buffer address in ax
        add     ax, 6               ; make it the address of data
        mov     raddr,ax            ; tell the interrupt process about it
        mov     rlen, (32-8)/2      ; tell the interrupt process the length
        mov     erflag, rwait       ; set flag byte
        int     20H                 ; fake a receiver interrupt
        suspend netr                ; wait for a packet reception
        push    bp                  ; push address of buffer
        mov     ax, 1               ; length of message to be sent
        mov     bp, einoq           ; point bp to output queue
        call    writeq              ; write message
        jmp     ein
        ejec
;
;   TINT
;       This is the Ethernet Transmission Interrupt Routine. It is invoked
; by the hardware when the transmitter has finished either with or without
; error. In either case, the T Done bit in the ENETCR will be set. If it is
; invoked by a software interrupt, this bit will not be set. A packet is
; transmitted by generating a software interrupt after intializing PADDR to
; the address of the packet, PLEN to the length-2 of the packet and TRIALS
; to 2(16-n)-1 where n is the number of transmissions to be attempted.
; The code writes a tsucc to ETFLAG upon successful completion or tfail
; on failure. A SUSPEND ETRANS will return either value in AL.
;
tint:   push    ax
        push    bx
        push    dx                  ; save registers
;
; Get the Ethernet Status byte and reset the transmitter
;
        movb    ah,enetsr           ; get the status
        movb    al,ecrsav           ; get a copy of the ethernet control byte
        andb    al, OFCH            ; clear the T start and T reset bits
        movb    enetcr,al           ; actually do it now!
        movb    ecrsav,al           ; update ecrsav
;
; Was the transmission sucessful?
;
        testb   ah, 1               ; Is the T done bit set?
        jz      first               ; If it isn't, then this is the first try.
        testb   ah, 6               ; Are either of the error bits set?
        jnz     terror              ; Handle error condition if so.
        movb    eoflag, tsucc       ; signal that transmission was successful
        inc     tgood               ; update stats
        jmp     tout                ; get out
;
```

```
; Handle packet retransmissions. Note that the initial transmission is
; handled as a retransmission. This implys an interpacket transmission
; interval of at least 1 count grain, about 30 usec.
;
terror: inc     terrs               ; increment count of transmit errors
first:  stc                         ; set carry flag
        rcl     trials,1            ; shift a 1 into the low bit
        jnb     toss                ; toss a number if we didn't shift into C
        movb    eoflag, tfail       ; otherwise fail on 17th attempt
        jmp     tout
;
; Toss a random number and weight it with the Binary Exponential Backoff
;
toss:   mov     ax,tod              ; get low time-of-day word (Random?)
        and     ax,trials           ; weight it
        inc     ax                  ; increment it before putting in counter
        movb    tcr, 70H            ; set up timer channel 1
        movb    tcl,al              ; write low byte    (critical region)
        movb    tcl,ah              ; write high byte   (critical region)
;
; Setup DMA channel and start transmission
;
        movb    al,dmasav           ; get dma control word copy
        andb    al, 0FEH            ; reset bit 0
        movb    dmacsr,al           ; disable dma channel 0
        mov     bx,paddr            ; get address of packet in bx
        mov     dx,[bx]             ; get first word of packet in dx
        mov     dmareg,dx           ; prime the dma register
        mov     dx,bx               ; get address of packet in dx
        mov     bx, dmach0          ; get address of dma channel registers
        shr     dx,1                ; make this the word address of the packet
        inc     dx                  ; and point to the second word
        movb    [bx],dl
        movb    [bx],dh             ; write packet address to dma channel
        mov     dx,plen             ; get packet length
        andb    dh, 3FH             ; clear direction bits
        orb     dh, 40H             ; set direction to memory to port
        movb    [bx+2],dl
        movb    [bx+2],dh           ; write count
        orb     al, 1               ; set enable channel 0 bit
        movb    dmacsr,al           ; enable DMA channel 0
        movb    dmasav,al           ; update dmasav
        movb    al,ecrsav           ; get control byte
        orb     al, 3               ; set transmiter enable and go bits
        movb    enetcr,al           ; do it!
        movb    ecrsav,al           ; save copy of ethernet control byte
;
; Get out of here
;
tout:   pop     dx
        pop     bx
        pop     ax                  ; restore state
        movb    ocw2a, 20H          ; acknowledge interrupt to chip
        iret                        ; and return
        ejec
;
;   RINT
;       This is the Ethernet receiver interrupt routine.
```

```
;
rint:    push    ax
         push    bx
         push    dx                  ; save state
;
; Get the ethernet status byte and reset the receiver and dma channel
;
         movb    ah,enetsr           ; get the ethernet status byte
         movb    al,ecrsav           ; get a copy of the enetcr
         andb    al, 0F3H            ; clear the R reset and R start bits
         movb    enetcr,al           ; write it to the enetcr
         movb    ecrsav,al           ; update the copy
         movb    al,dmasav           ; get the dmacsr copy
         andb    al, 0FDH            ; clear channel 1 enable bit
         movb    dmacsr,al           ; do it
         movb    dmasav,al           ; update copy of dmacsr
;
; Was packet reception error free?
;
         testb   ah, 8               ; is the receiver done bit set
         jz      newbuf              ; if not, we have a new buffer
         testb   ah, 70H             ; are any of the error bits on?
         jnz     rerr                ; if so go handle errors
         movb    erflag, rsucc       ; otherwise signal successful completion
         inc     rgood               ; update stats
         movb    bl,dmach1+2         ; get low tc byte out of dma channel
         movb    bh,dmach1+2         ; get high tc byte out of dma channel
         andb    bh, 3FH             ; clear the direction bits
         mov     ax,rlen             ; get rlen in ax
         sub     ax,bx               ; get number of words transferred in ax
         mov     bx,raddr            ; get pointer to buffer in bx
         mov     [bx-2],ax           ; write length into buffer
         jmp     rout                ; get out
;
; Reception was in error - Try again
;
rerr:    inc     rerrs               ; update stats
newbuf:  mov     bx, dmach1          ; get address of dma channel 1 registers
         mov     dx,raddr            ; get address of buffer
         shr     dx,1                ; make it a word address
         movb    [bx],dl
         movb    [bx],dh             ; write buffer address to dma controller
         mov     dx,rlen             ; get the buffer length
         andb    dh, 3FH             ; clear direction bits
         orb     dh, 80H             ; set direction to port to memory
         movb    [bx+2],dl
         movb    [bx+2],dh           ; write termination count to the controller
         orb     dmasav, 2           ; set dma channel 1 enable bit in copy
         movb    al,dmasav           ; get copy of dmacsr
         movb    dmacsr,al           ; write it to the channel
         orb     ecrsav, 0CH         ; set T enable and start bits in copy
         movb    al,ecrsav           ; get the copy
         movb    enetcr,al           ; write it to the ethernet control register
;
; Get out of here
;
rout:    pop     dx
         pop     bx
pop     ax                  ; restore state
mov     ocw2a, 20H          ; acknowledge the interrupt controller
iret                        ; return
```