FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems

By

Charles H. Ng

5055:TR:82

CALIFORNIA INSTITUTE OF TECHNOLOGY

Computer Science

5055:TR:82

FIFO Buffering Transceiver: A Communication Chip Set for Multiprocessor Systems

by

Charles H. Ng

In Partial Fulfillment of the Requirements for the Degree of Master of Science

December, 1982

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

ABSTRACT

This thesis describes a family of VLSI chips designed to link a number of processors on a one-to-one basis. With these chips as communication system building blocks, a complex multiprocessor system can be built. Inter-processor communication within the multiprocessor system is accomplished by passing messages composed of data packets.

The resulting chip, called a First-in-first-out Buffering Transceiver (FIBT), provides a full duplex communication channel between any two processors. FIFO queues are provided for buffering data on each communication channel. FIBT accepts data packets from the host processor via a parallel data bus and serially sends them out to the destined processor. FIBT handshakes with the processor by using asynchronous interrupt signals.

Linkage between any two FIBTs is accomplished by using only two wires. Both data bits and handshaking signals are sent by these two lines. The FIBT system is neither a synchronous nor an asynchronous one; instead, it is an "one-clock-different-phases" system. A clock signal sets up the frequency reference; the start and stop bits set up the phase reference.

Finally, FIBT is implemented in nMOS technology. The design of the circuit is discussed in detail. The design is generalized enough so that data packets of various sizes can be handled. The layout of the chip is coded in an integrated circuit descriptive language. Any member of the family of chips can be obtained by changing three basic parameters. Techniques used in verifying the circuit are shown, and several observations about VLSI design are offered.

ACKNOWLEDGEMENTS

I would like to express my gratitude to all the students, staff and faculty at the Caltech Computer Science Department who gave me their help and support during the course of this project. Particularly, I owe special thanks to my advisor, Chuck Seitz, who taught me the VLSI magic, introduced me to this project, and patiently guided me throughout the project. I thank Doug Whiting for his work which greatly influenced this project, and Dick Lang for his many interesting and inspiring ideas.

I would like to thank Chris Kingsley, Tom Hedges, and Randy Brayant for their great VLSI design tools. Also, thanks goes to Don Speck, Jimmy Lam, and Peter Hunter, who helped me greatly in laying out the chip.

I am indebted to Caltech whose financial support allowed me to obtain such an excellent education in this country. I am even more indebted to my parents who foster me, support me, and love me. I am grateful to my friends – Wu Yin Ching, Lau Yuk Kong, and Florence Tang – who greatly influenced me during my years of development.

Finally, and most importantly, I thank God for His many blessings and guidances.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. Motivation

I can still remember when in my youth I built a small six-transistor radio. I used to wonder what kind of magic they put into those little metal capsules. Today, advances in semiconductor technology have added a new dimension to the tricks of electronics and computer science. "Very Large Scale Integration" is as much advanced over my transistor radio as a modern jet is over the Wright flyer. Computer scientists must discover new and simpler ways to use this technology. The family of communication chips discussed in this thesis is intended as one step toward these objectives.

One notable attempt at making digital systems design simpler was the macromodule project [Clark 67]. The macromodules were a set of building blocks for digital systems, such as register, arithmetic unit, random access storage, and control unit. Each module was contained in a small box that could be inserted into a rack. The system could be built by a person without any experience in digital system design, with the assurance that if the parts fit together, the system would function as connected. Moreover, the designer did not have to be concerned about timing rules for the modules, because the control in macromodules was self-timed. Timing rules were taken care of "automatically".

What are the macromodules for the 1980's ? Should they still be registers and arithmetic units. No, the macromodules can be bigger and more complex. We should be able to connect computers together to form

super-computers.

The concept of multi-processing is very natural to VLSI. and many multiprocessor systems have been suggested [Seitz 82]. These systems are characterized by the replication of one kind of element, ranging in size from a storage cell to an entire computer. This research is concerned only with the ensemble of relatively large processors, the ensemble being referred to as a "homogeneous machine" [Locanthi 80] [Lang 82].

A homogeneous machine has the structure shown in figure 1.1, in which each processor has its own storage, and can communicate by passing messages with some or all of the other processors in the ensemble. There are several ways in which messages can be passed. Messages can be broadcast on a contention network, such as the ethernet, or on a bus by the sending processor to the entire system. The destined receiving processors

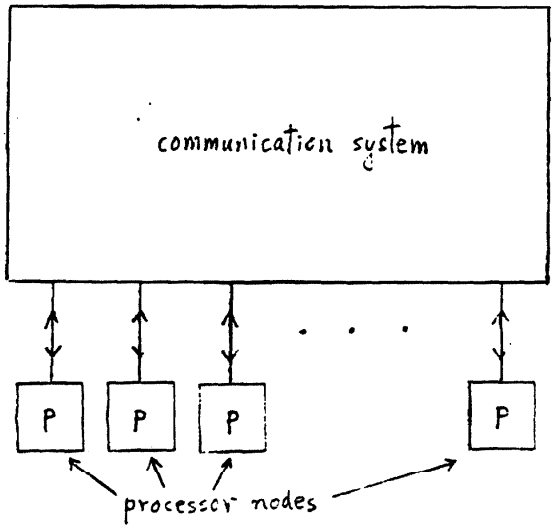

fig. 1.1 Basic Model of a Homogeneous Machine

then pick up the messages. This approach is severely limited by traffic congestion in large networks, unless a mechanism is provided for separating the communication medium, filtering out local messages, and forwarding non-local messages. Chips for this kind of filtering and forwarding communication system have recently been investigated by Whiting [Whiting 82], and his work greatly influenced the work reported here.

Instead of this "one-to-many" linkage, processors can be linked on a "one-to-one" basis. Each processor has one channel to communicate with each of its neighboring processors. In order to send a message to a distant processor, the sending processor first sends the message to its neighboring processor, and the neighboring processor sends it to the next neighboring processor. This process continues until the message is received by the destination processor. This method calls for a routing algorithm for sending messages in order to achieve high efficiency and to avoid deadlocks. Moreover, there are the questions of network topology, required bit rates, packet sizes, and buffering requirements. In answering these questions, we have been guided largely by the simulations reported in [Lang 82].

The aim of this research is to develop the communication channel needed to connect any two processors on a one-to-one basis within a multiprocessor system. The requirements for this communication channel are :

First, it must interface with the host processor via a parallel data bus.

Secondly, it provides serial data linkage between any two processors.

Thirdly, each channel must have a full duplex linkage.

Fourthly, there are data buffers for each communication channel.

The family of chips, referred here as FIrst-in-first-out Buffering Transceivers (FIBTs), were developed to meet these requirements.

1.2. Outlines of the FIBT

The FIBT can be divided into five major functional blocks : 1) Processor-interfacing circuit, 2) Transmitting buffer, 3) Receiving buffer, 4) Transmitter, 5) Receiver. Figure 1.2 is both a block diagram and a basic floor plan of the chip.

As mentioned before, in the multiprocessor system, inter-processor communication is done by passing messages. A message is comprised of a number of packets. Each packet has L words; each word has W bits. The variables L & W will be used throughout this paper to designate the packet size. The FIBT is designed to pass data in a complete data packet in order to

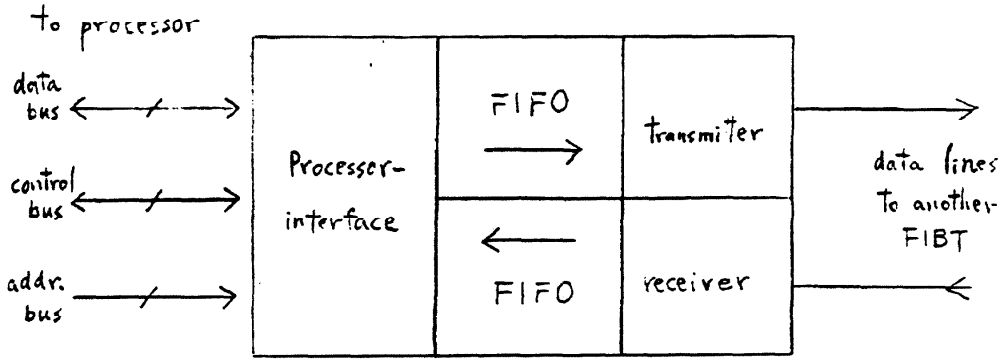

fig. 1.2 Functional Blocks of the FIBT

reduce the overhead of handshaking. The FIBT is ready to accept a data packet, either from the host processor or another FIBT, only if there is enough room inside the FIBT to store a complete packet. Similarly, if the FIBT has data ready for the processor, the data must be in a complete packet. The host processor is expected to send out (or receive) a whole packet of data to (from) the FIBT. Thus, if a processor has only a few words (say X) of data to send, it must send out (L - X) words of unused data to complete the packet. Hence, the word size of a packet is important. If we choose a very large packet size, then when the processor wants to send out a small message, much space will be wasted. If we choose a very small packet size, much handshaking is needed. The choice of the packet size greatly depends on the general communication pattern within the system.

Obviously, the FIFO buffer, the core of the FIBT, is also designed to handle data packets. Let us see how it works. For the moment, let us assume the length of a queue be longer than 2 * L. In fig. 1.3, data passes through the FIFO from left to right. The length of both the leftmost section (section 1) and the rightmost section (section 3) of the queue are L. If section 1 is all empty, then the queue can send off an "empty" signal, telling the external world that there is enough room for a complete packet. Likewise, if section 3 is filled by a complete packet, then the queue can signal the external world that there is a packet ready to be read. The remaining part of the queue (section 2) simply provides extra buffering space.

In order to hold a complete packet, the minimum length of the queue must be L. However, a minimum length of 2 * L is much more desirable. If the length of the queue is no less than 2 * L, then the process of passing out a packet does not interfere with the process of accepting a packet, and both

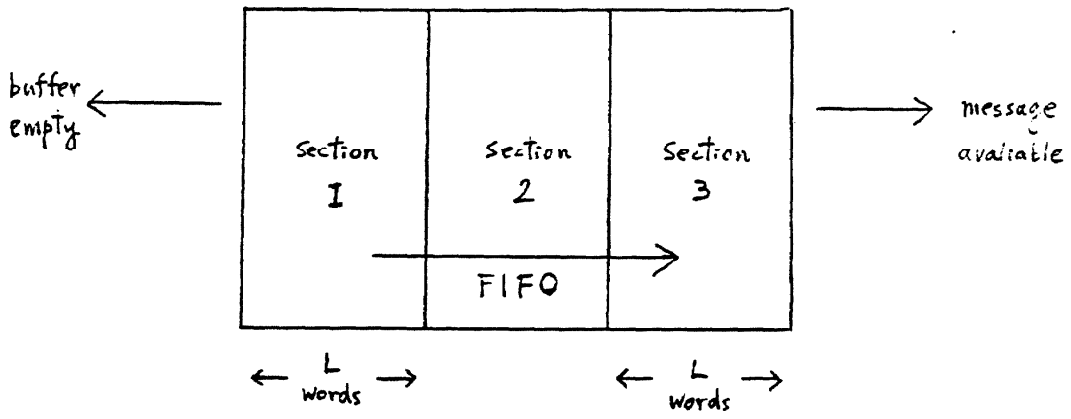

fig. 1.3 Simplified Diagram of the FIFO

processes can run concurrently. If the queue is shorter than 2 * L, and if there is already an existing packet inside the queue, then the external world cannot push another packet of data into the queue unless the queue is emptied first.

The FIBT uses a parallel bus, a control bus, and an address bus to communicate with the host processor. Between any two FIBTs, two serial data lines are used for communication.

When the FIBT is transmitting data, it takes in words of data from the processor via the processor interfacing circuit, pushes the data into the buffer, passes them to the transmitter, which then links up the data bits, and drives them out serially.

When the FIBT is receiving data bits from another FIBT, the receiver accepts the data bits, links them up, pushes them into the FIFO, and passes

them to the processor.

1.3. Overview

The following chapters describe in detail the underlying features of the FIBT, and how the FIBTs were designed.

Chapter Two discusses how the FIBT interfaces the host processor. It describes the parallel data bus, the control bus, the address bus, and the internal registers of the FIBT. It explains how to program the processor to send and receive data packets.

Chapter Three describes how two FIBTs communicate with each other. It shows how each chip uses only two lines to pass the data bits and the handshaking control character to another FIBT. Also, it explains how two communicating FIBTs are synchronized by using a frequency reference and a start & stop bit.

Chapter Four describes in detail how the FIBT is actually designed, laid out, and verified. Several methodologies in designing VLSI systems are discussed. Several layout techniques and design tools are also shown.

Appendix A shows how a parameterized leaf cell is coded in Earl – an integrated circuit descriptive language. Different instances of the leaf cell are composed together to form an address decoder.

Appendix B gives a sample run of the logic simulator MOSSIM. The run shows how the processor writes data words into the FIBT, how the transmitter sends off data bits, and how the receiver accepts incoming data bits.

CHAPTER 2

Interfacing the Processor

On the processor interfacing side, the FIBT uses a parallel data bus, a control bus, and an address bus to interface the host processor. This chapter describes the underlying features of the interfacing circuit.

2.1. Data Format

The way data is formated in a data packet greatly effects the design of the processor interfacing circuit. Thus, it is necessary for the reader to first understand the data format of a packet.

As mentioned before, we have to be careful when choosing the packet size in order to obtain good performance in the communication system. A study of the communication pattern of the homogeneous machine [Lang 82] reveals that the mean message size is about 750 bits, with a standard deviation of about 250 bits. Although the resulting FIBT of this research is generalized enough to handle various packet sizes [1]; for the FIBT prototype, we decided to let 256 bits be the standard packet size. Moreover, we use a 16-bit wide data bus, which means we let both L and W to be 16, and a standard packet is 16 16-bit words. The format of a packet is summarized in fig. 2.1 .

Although the content of the packet is of no concern to the FIBT, the first word of the packet is usually routing information of the packet. Hence, any processor which has just received a packet can read the first word and

[1] Refer to section 4.2

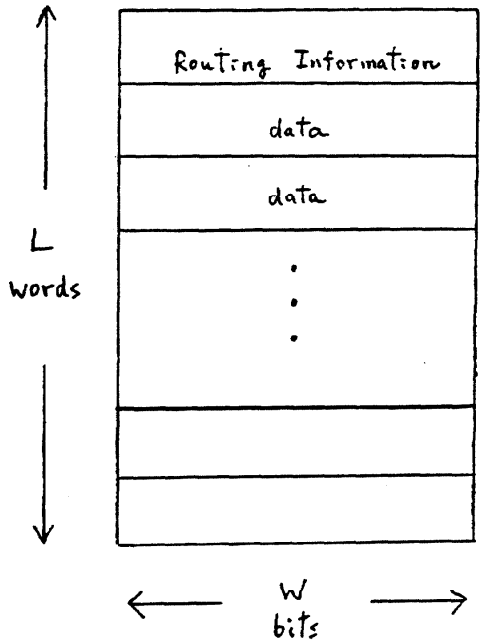

fig. 2.1 Format of the Data Packet

investigate the routing information. The processor can then decide where to store the packet, depending on whether the packet is for itself or not.

2.2. Control and Address Lines

Like many other processor peripheral chips, the FIBT has the following common control lines and address lines to interface with the processor :

Inputs :		Outputs :	
~ Reset	(~Reset)	~ Buffer empty interrupt	(~BEI)
Read / ~write	(R/~W)	~ Request next word	(~RNW)
Chip select	(CS)	~ Message available interrupt	(~MAI)
Address line 0	(A0)	~ Next word available	(~NWA)
Address line 1	(A1)		

Whenever the processor wants to reset the FIBT, the "~Reset" line should be held low. In order to read (write) the FIBT, the processor should put the valid address onto the address lines, set "R/~W" high (low), and then set "CS" high.

There are two output control lines for the receiving side of the FIBT. When a complete packet is available to be read, the "~ Message Available Interrupt" line, if enabled [2], will be driven low. After a word of the packet is read out, the interrupt line will return high. It will not go low again until another packet is ready.

Whenever a word of data is available to be read, regardless of whether the FIBT has received the complete packet, the "~ Next Word Available" line will be driven low. The simplified circuit diagram is shown in fig. 2.2 . .

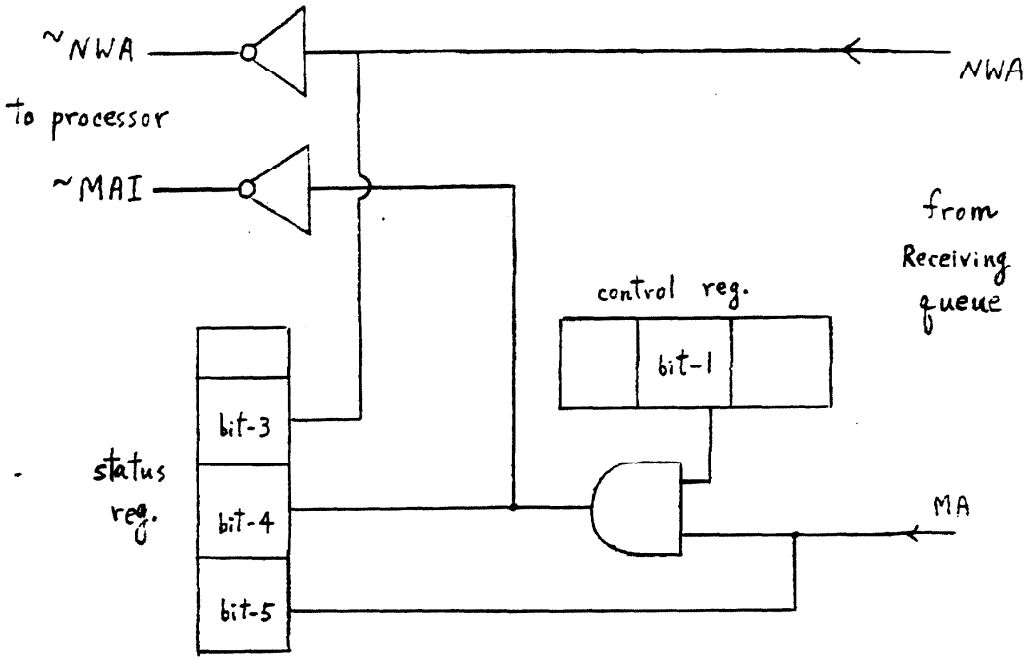

fig. 2.2 Simplified Circuit Diagram of the Receiver Processor Interface

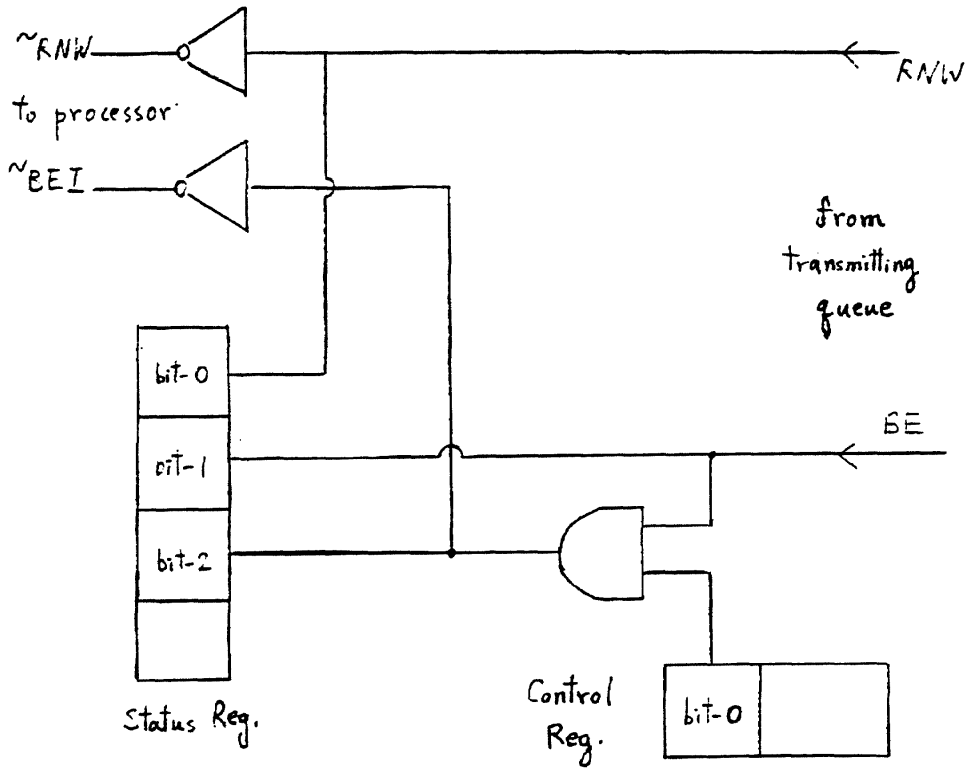

fig. 2.3 Simplified Circuit Diagram of the Transmitter Processor Interface

Similarly, there are two output control lines for the transmitting side of the FIBT. When the transmitting buffer has enough room for one complete message, the "~ Buffer Empty Interrupt" line, if enabled [2], will be driven low. Whenever the transmitting buffer has enough room for another word of data, the "~ Request Next Word" line will be driven low (fig. 2.3).

[2] *Refer to the control register in section 2.3*

2.3. Internal Registers

There are several registers inside the FIBT, and their locations are given in the following table.

Reg. name	R/~W	A1	A0
Receiving Buffer (RB)	1	0	0
First Word Reg (FWR)	1	0	1
Status Reg (SR)	1	1	X
Transmitting Buffer (TB)	0	X	0
Control Reg (CR)	0	X	1

When the "~ Next Word Available" line goes low, the processor can read the "Receiving Buffer" to obtain the data word. Data words will then be shifted down the FIFO automatically. However, if the "~ Next Word Available" line has not gone low, and the processor attempts to read the "Receiving Buffer", it will not receive any valid data.

Similarly, only after the "~ Request Next Word" line goes low should the processor write a data word into the "Transmitting Buffer". Otherwise, any word written to the "Transmitting Buffer" will simply be lost.

Reading the "First Word Register" is not quite the same as reading the "Receiving Buffer". As the processor reads the "First Word Reg.", it will get the first word from the receiving queue, but thereafter the queue will not shift its data words. In other words, the data word is read, but it is not pulled away from the queue. With this register, the processor can read the first word of a received packet without "disturbing" the receiving queue.

The mapping of the Status Register is as follows :

Bit	0 (LSB)	Request Next Word	(RNW)
	1	Buffer Empty	(BE)
	2	Buffer Empty Interrupt	(BEI)
	3	Next Word Available	(NWA)
	4	Message Available Interrupt	(MAI)
	5	~ Message Available	(~MA)

On the receiving side of the FIBT, whenever there is a packet ready, the "~ Message Available" status bit (bit 5) will be cleared. If the message available interrupt is enabled, the "Message Available Interrupt" status bit (bit 4) will be set too. After a word of the message is read out, the "MAI" status bit will be cleared and the "~MA" status bit will be set. Also, whenever a word of data is available, the "Next Word Available" status bit (bit 3) will be set. (fig. 2.2)

On the transmitting side of the FIBT, the "Buffer Empty" status bit (bit 1) will be set whenever the transmitting buffer has enough room for another packet. The "Buffer Empty Interrupt" status bit (bit 2) will be set too, if the interrupt has been enabled. Both the "BE" and the "BEI" status bits can be cleared by writing a word of data into the transmitting queue to partially fill it up. Also, when the transmitting buffer is ready for taking another word of data, the "Request Next Word" status bit (bit 0) will be set (fig. 2.3) .

The mapping of the Control Register is as follows :

Bit	0 (LSB)	Buffer Empty Interrupt Enable	(BEIE)
	1	Message Available Interrupt Enable	(MAIE)

The "Buffer Empty Interrupt Enable" control bit (bit 0) is for enabling the "~ Buffer Empty Interrupt" line and the "Buffer Empty Interrupt" status bit. If a "1" has been written into this control bit, then whenever the transmitting buffer is empty, the "~BEI" line will be driven low and the "BEI" status bit will be set. However, if a "0" has been written into the control bit instead, independent of the state of the transmitting buffer, the "~BEI" line will always remain high and the "BEI" status bit will remain cleared (fig. 2.3) .

Similarly, the "Message Available Interrupt Enable" control bit (bit 1) works the same to enable the "~ Message Available Interrupt" line and the "Message Available Interrupt" status bit (fig. 2.2) .

2.4. Procedure for Receiving Messages

Let us discuss how the FIBT transfers a newly arrived message to the processor. After the FIBT has received a complete packet from another FIBT, the FIBT will clear the message available status bit (bit 5). If the interrupt is enabled, then the "~ Message Available Interrupt" line will be driven low, and the "Message Available Interrupt" status bit (bit 4) will be set. Thus, in order to detect any newly arrived packet, the host processor can either poll the "Message Available" status bit, or turn on the interrupt line and wait for the interrupt.

After the processor detects a newly arrived packet, it can then read the "First Word Register" (FWR) to find the destined address of the message. If

the processor decides to read out the message, the processor can repeatedly read the "Receiving Buffer" L times to get the complete packet. After the first word of the packet is read out, the "~MAI" line will return high, the "MAI" and the "MA" status bit will be cleared. Figure 2.4 shows the precedence of all the signals mentioned above. Such a diagram is called the sequence diagram and will be discussed in greater detail in section 4.3.1.

In reading the "Receiving Buffer", the processor must monitor the "~ Next Word Available" signal all the time. The FIBT will drive the "~ Next Word Available" line low and set the "Next Word Available" status bit whenever a word of data is ready to be read. After a word of data is read out, the FIBT will drive the "~NWA" line high, clear the "NWA" status bit, and start shifting in the next word for the processor to read. When the next word is ready, the "~NWA" line will be driven low, and the "NWA" status bit will be set again.

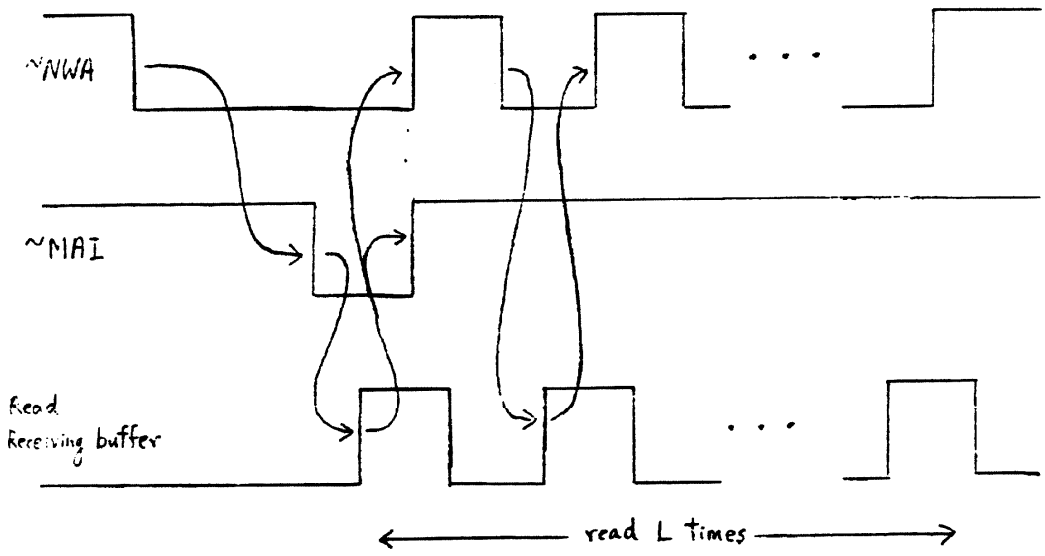

fig. 2.4 Reading the Receiving Buffer

The "~NWA" signal is meant for synchronizing the FIBT with the host processor. The host processor can be a micro-processor, or a Direct Memory Access (DMA) controller. If the latter is used, the "~NWA" signal can be routed to the "DMA request" input of the controller. After a word is read, the "~NWA" signal will go low again in about 200ns (max.) [3] If the host processor handles a word at a slower rate, a new word will always be available after the previous word is read, and the "~NWA" signal can be left unused.

The processor must always follow the rule of handling data in the form of a complete packet. Otherwise, the continuous running of the communication system is jeopardized. The processor can jam the receiving queue if the processor reads out a word of data when the whole message is not ready to be read. For example, assume the FIBT has just received one word of data. The "~NWA" line goes low while the "~MAI" line remains high. If the processor erroneously reads out this word of data from the FIBT, the L-word packet is destroyed. The FIBT can never get a complete packet to fill up the receiving queue, and the "~MAI" line will never go low again. The receiving queue is jammed permanently, and the situation calls for a system reset.

2.5. Procedure for Sending Messages

When the processor has a packet to send, it should look at the "Buffer Empty" status bit (bit 1) and determine if there is enough room in the FIBT to hold one whole packet. If there is no room available, the processor can turn on the "~Buffer Empty Interrupt" line. Later, when FIBT has enough room, it will interrupt the processor.

---

[3] 200ns is determined by SPICE simulation run. The period may vary on different fabrication runs.

After the processor determines that the FIBT has enough room for a message, the processor can start writing all L words of data to the "Transmitting Buffer" (TB). After a word is written to the transmitting buffer, the "~BEI" line will return high, the "BEI" and the "BE" status bit will be cleared. (fig. 2.5)

Notice that after the words of data have been shifted down to another end of the FIFO, there will be an "empty" buffer left behind, and the "~BEI" line can go low again, sending an interrupt signal to the processor. The FIBT's FIFO is sufficiently faster than a processor that at the exit of an interrupt routine the processor will be interrupted again. In order to avoid this undesirable interrupt, the processor node must have some logic units to mask the interrupt while a data packet is being transferred and unmask the interrupt after the transfer is done.

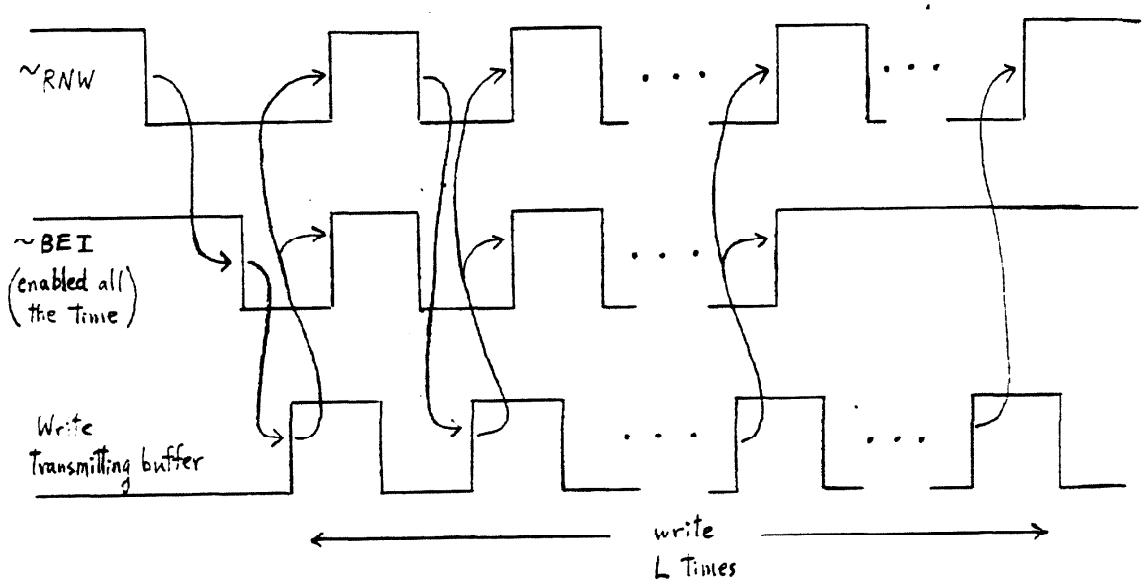

fig. 2.5 Writing to the Transmitting Buffer

Again, similar to the "~ Next Word Available" output line, the "~ Request Next Word" line is used to synchronize the FIBT with the host processor. During the time when the processor is writing a word into the FIBT, the "~RNW" line goes high. After 200ns (max), when the FIBT is ready to receive another word from the processor, the "~RNW" line will go low again.

CHAPTER 3

Communication between Two FIBTs

On the transmitter-receiver side, the FIBT uses two serial data lines to transmit and receive data, and takes in a clock signal as a frequency reference. This chapter discusses how two FIBTs communicate with each other.

3.1. Data Format

Again, similar to the way messages are passed from the processor to the FIBT, messages passed between FIBTs are handled in the form of a packet. However, the format of a packet is different from the one which the FIBT uses to communicate with the processor. The major difference is that data is passed serially.

There are two kinds of message : data packet and control character.

At the beginning of the data packet, there is a start bit, then a message type bit, which is a "0" for the data packet. These are followed by L*W bits [1] of data and the packet is terminated by a stop bit. (fig. 3.1)

The control character begins with a start bit, followed by a "1" for the message type bit, and ends with a stop bit.

Notice that the data packet size for inter-FIBT communication does not strictly have to be L*W. It can be a fraction of L*W. In effect, the FIBT divides a data packet into smaller sub-packets. With this method, more start and stop bits are needed, and the data transmission rate would be lowered

[1] For the prototype, it is 16 * 16 bits of data

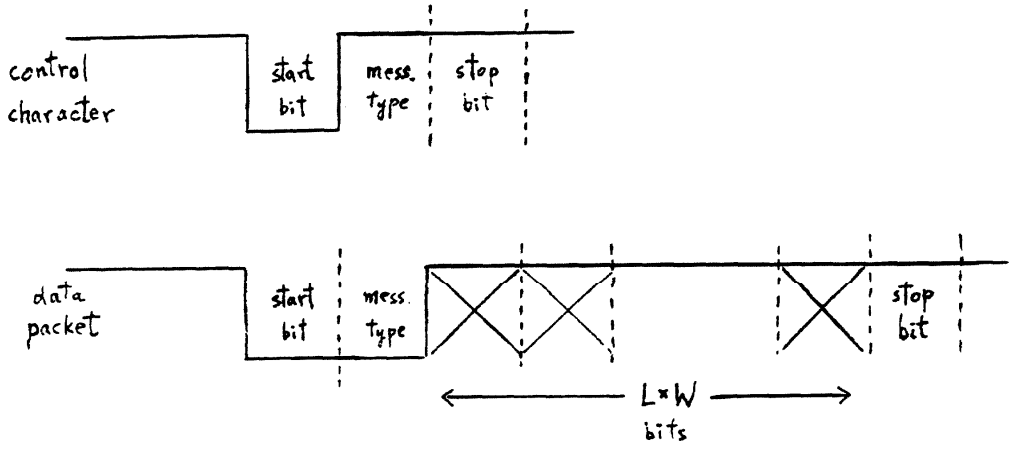

fig. 3.1 Format of the Data Packet and the Control Character

slightly. Obviously, it does not make much sense for the FIBT to pack data packets into a larger data packet. The channel may be left idle though there is a data packet ready to be transferred, and communication deadlock can be created.

3.2. Control

Years ago, a similar transmitter-receiver was built. In this system, the transmitter uses two lines to link with the receiver; in other words, each transmitter-receiver uses four lines to link with another transmitter-receiver. Two of these lines are data lines, and the other two are acknowledge lines (fig. 3.2). After the receiver has received a packet from the transmitter, it sends off an acknowledge signal back to the transmitter. For a good sized packet, it is obvious that the acknowledge line does not work very hard. The data lines must run at a frequency L*W times higher than the

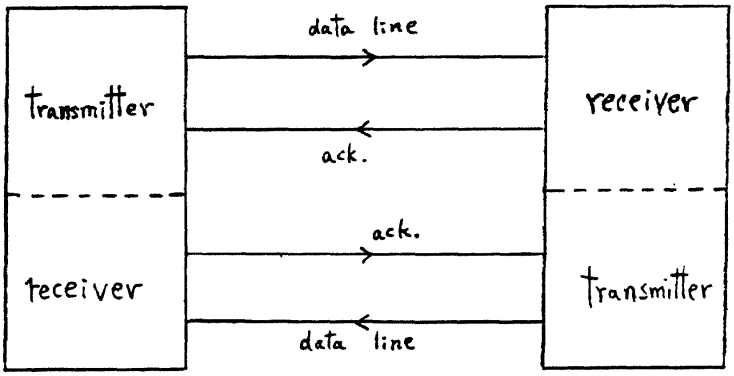

fig. 3.2 An Interesting Transmitter-receiver Design

acknowledge lines.

One would think that a better system can be built by combining the acknowledge lines with the data lines, and this is exactly the scheme that the FIBT employs. Any two FIBTs are linked together by two data lines. All the hand-shaking is done by sending control characters onto the data lines.

If the FIBT has a packet to send to another FIBT, it must first determine whether the receiving FIBT has enough room to store one complete packet. Let us suppose both FIBT-A and FIBT-B have just been reset, and all their buffering queues are empty. Let us suppose FIBT-B has a data packet to send to FIBT-A. B first sends off the start bit, which signals A to pick up the packet. Then, B sends off the (L*W)-bit packet, and terminates the packet by a stop bit. After A receives the whole packet, A will monitor its receiving queue to see if there is enough room for another packet. When A sees an "empty"

receiving queue, it will send an "empty" control character to B. This control character signals B that A is ready to receive another data packet. On the other hand, after B finishes sending a data packet, B is not allowed to begin sending another packet, though it may have one to send. B must turn to listen to A and wait for the "empty" control character. After B receives the "empty" control character, it can then proceed to send another data packet.

The sending of a control character is set to have a higher priority than the sending of a data packet. In other words, A will send off the control character as soon as the transmitting line is idle. There are two reasons for this priority. Firstly, the control character is much shorter than the data packet. Secondly, the sending of the control character will free the communication channel in the other direction. Thus, the time that B must wait is minimized. If a data packet is divided into sub-packets, then the waiting time can be reduced further. In between transferring sub-packets, which are smaller than a data packet, the transmitter will have the chance to transfer a control character.

3.3. Synchronization

Synchronization is accomplished by the combination of a start bit, a stop bit, and a clock signal. The system is not quite asynchronous; instead, it is a hybrid of a synchronous and an asynchronous system. The whole multiprocessor system takes the same clock signal to clock all the FIBTs, but each FIBT may be working at a different phase. The clock is a frequency reference to all FIBTs, and the start and stop bit gives the phase reference to each FIBT. In this "one-clock-different-phases" system, the clock skew problem does not exist. There can be a time lead or time lag between any two FIBTs, and they can still communicate reliably with each other. Moreover,

the number of bits of data that can be placed between the start and stop bit is theoretically infinite. Also, the one clock system has the advantage of simplifying the problem of clocking different parts of the system with different clocks [2] . The combination of all these features makes the hybrid system an attractive system to be tested.

In order to mark the phase correctly, the internal clock rate of the FIBT is three times the data transmission rate. Every data bit is sampled three times, and a bit is always latched at the second sample. In fig. 3.3, the period of a data bit is divided up into three equal parts. Two extreme cases are given to mark the time when the data bit will be latched. We see that only the middle part of a data bit will be latched. Thus, if the set up time of a

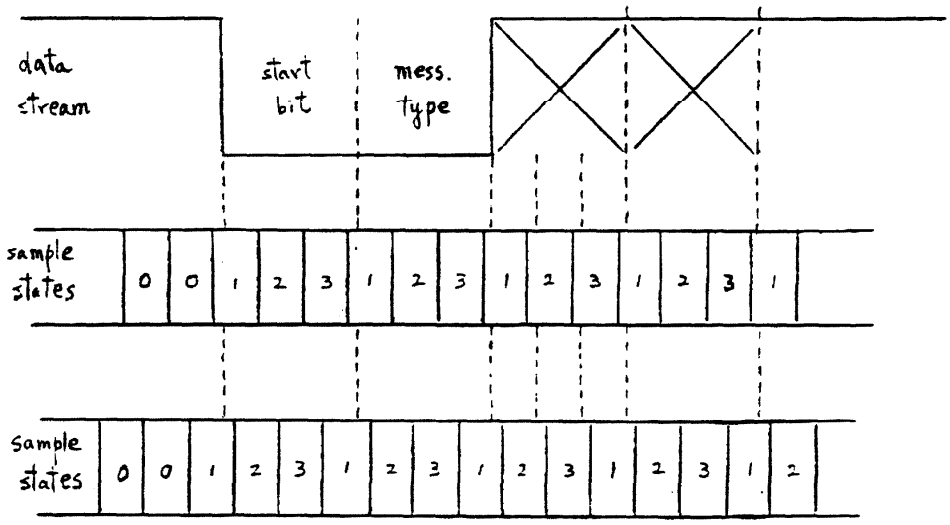

fig. 3.3 Sampling Input Data Bits

[2] Actually, different FIBTs can be clocked separately with different clock signals with a specified tolerance. If the data packet is divided into sub-packets, the tolerance can even be bigger.

data bit is shorter than a third of the period time, data will be latched correctly. Of course, each data bit can be sampled at a higher rate in order to allow for longer set up time. However, since there is an upper bound on the internal clock rate, a higher sample rate would mean a low data transmission rate. As a compromise, the finite state machines of the FIBT are designed to run at 6 MHz, and the data transmission rate is 2 MBPS.

For a synchronous system, a receiver in this case, to synchronize with an free-running input signal, synchronization failure is doomed to happen [Seitz 80]. The designer can only work on lowering the probability that the failure will happen. The smaller the probability is, the more reliable the system functions.

A clocked storage element, called a synchronizer, is used to synchronize the input signal with the system clock (fig. 3.4). Each synchronizer is a bistable element which helps to restore the logic level from a signal close to the switching level.

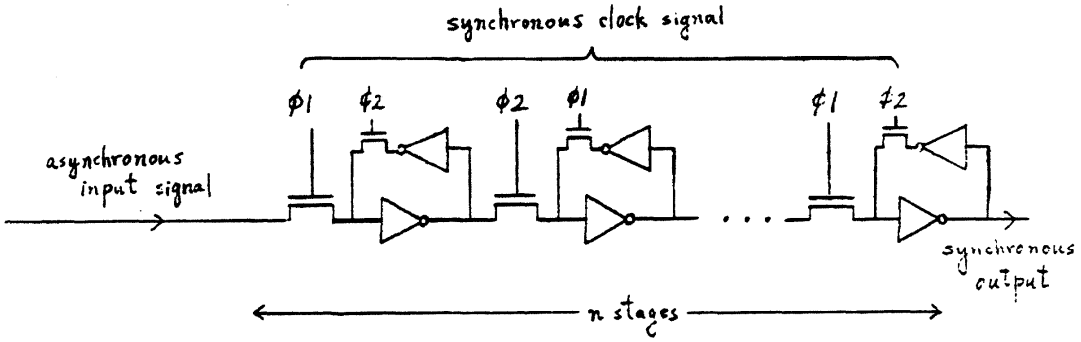

fig. 3.4 Synchronizer

The probability of a system failure on each synchronization event can be given in the following equation :

$$f * t * e^{-p*d*n}$$

Where "f" is the frequency of transition of the input signal. Variable "t", a parameter of the synchronizer, is the sample time period in which a transition of the input signal would cause a system failure. The rightmost term describes the metastable decay of the synchronizer. After a bistable unit goes into a metastable condition, the probability that it will remain in the metastable state after a time period of "d" is e^{-p*d}. Variable "p" is the rate of the Poisson decay process, and it depends on the circuit characteristics. For a synchronizer with "n" stages of bistable unit, the probability is simply e^{-p*d*n}.

Experimental results show that "t" is about T/10, where T is about 0.2ns for 4-micron technology [3]. Thus, "t" is about 20 picoseconds.

We can assume "p" to be 1/((1+r) T) for a ratio logic of r. Because the bistable unit is made up of an inverter with r=8 and an inverter with r=4, the worst case of "p" is 1/9T or 1/1.8ns.

For each data packet, after the start bit is well synchronized, the whole packet is synchronized [4]. In other words, there is only one transition in a packet that can cause a failure. Since the data transmission rate is 2MBPS and there are 16*16 bits in a packet, the frequency of transition, "f", is $1/256 * 2 * 10^6$.

---

[3] T is the scaled time unit used in Mead's lambda model

[4] Refer to the receiver FSM in section 4.3.4

For a synchronizer running with a two phase 6MHz clock, the time left for a bistable element to exit from the metastable state is only 80ns (half the period time) . In order to provide extra margin for preset time, 60ns is used for "d".

Thus the probability of a failure for each synchronization event of a two stage synchronizer ("n" = 2) is :

$$(1/256 * 2 * 10^{6}) * (20 * 10^{-12}) * e^{-1/1.8 * 60 * 2}$$

$$\sim 1.7 * 10^{-36}$$

Since synchronization event is running at 6MHz, the failure rate per second is $(1.7 * 10^{-36}) * (6 * 10^{6})$ or $1 * 10^{-29}$. This means that an average of one failure will occur in about $3 * 10^{21}$ years.

CHAPTER 4

Implementation

The FIBT has been implemented in nMOS technology. In this chapter, two main design concepts are discussed. Then, the circuit design, the actual layout, and the verification are shown.

4.1. Modular Design

The FIBT can be divided into 5 major modules : 1) processor interfacing circuit, 2) transmitting FIFO, 3) receiving FIFO, 4) transmitter, 5) receiver (fig. 1.3).

Similar to structured programming, in VLSI design there are many obvious reasons for the modular design. Modular design helps to control complexity, simplify verification, minimize organizational overhead. In turn, this approach shortens design times and saves manpower [Mead 80] [Buchanan 80] [Moore 79] . After the outputs, the inputs, and the process of each module are defined, each module can be independently designed, verified, laid out, simulated, fabricated, and tested. If every module works, the chance for the whole system to work is much higher. However, since this is not a totally self-timed system, there may be timing problems when the modules are connected. This must be taken into consideration when the modules are defined.

4.2. Parameterized Design

"VLSI technology is oriented to replication, and all the parts of a chip are made in parallel." [Seitz 82] The major building blocks of the FIBT are

made by repeating some basic building blocks for a specific number of times. The root definition of the FIBT is parameterized by three important parameters : queue length (QL), packet length (L), and packet width (W). Any particular FIBT can be generated from the root definition by specifying the three parameters.

The ability to parameterize the design is a very powerful tool in VLSI design. Parameterized VLSI design is very similar to parameterized programming. Alternative chips can be obtained by changing the parameters of the root definition. "One can even say that the relationship with software is not by analogy but by equality." [Buchanan 80]

4.3. The Circuit Design

Although the design is based on the nMOS transistor level, it can be generalized easily. Most of the building blocks are fairly straightforward, and the reader should be able to understand them.

4.3.1. FIFO Buffer

There are two generally accepted ways of building a FIFO buffer. There is the random access memory based FIFO, which uses a big RAM together with two pointers pointing at the last read and the last write locations [Whiting 82]. Or, there is the shift register based asynchronous FIFO buffer [Cohen 77] [Clark & Seitz 77] [Galvin & Kohn 78] [Sutherland 76].

A rough comparison between Whiting's RAM based FIFO and the FIBT's shift register based FIFO was made. The major differences lie in the size and the speed of the FIFO.

The controller of the RAM based FIFO has about the same number of logic gates as in the controller of the shift register based FIFO. However, the

former takes more wiring connection than the latter does. As Sutherland and Mead pointed out [Mead & Sutherland 77]: wiring has become relatively expensive in terms of layout area. The final layout of the RAM based FIFO controller takes up approximately twice the area that the shift register based FIFO controller does. However, the storage unit of the RAM based FIFO can be smaller than the storage unit of the shift register based FIFO [1]. Thus, for a FIFO with a lot of storage units, the RAM based FIFO is probably smaller. For a 32*16-bit FIFO, the two layouts take about the same area.

In the shift register based FIFO, each data bit is passed by a series of drivers -- the shift registers' drivers. The series of drivers, each acting like a repeater, can drive the data bit very quickly down a long FIFO [2]. Also, these drivers are part of the shift registers, and they come almost at no cost. In the RAM based FIFO, extra drivers are needed to transfer data between the storage unit and the external world. In order to build strong drivers for long data lines, a number of drivers are cascaded together, and they take up space! If area is a problem, then smaller drivers can be used, but the FIFO will be slower.

The shift register based FIFO is chosen for the FIBT. Each FIBT carries two FIFOs of length QL words and width W bits [3]. The actual circuit is given in fig. 4.1 and 4.2. The controlling unit of the FIFO employs mainly two Muller-C elements to do the handshaking. The circuit is not totally speed independent. In each stage, the data must be valid at the output lines before the request line to the next stage is raised. In order to achieve this sequence constraint,

[1] This is especially true for dynamic RAM unit, such as the three transistor dynamic RAM unit.

[2] Actually, in the shift register based FIFO, the speed limiting factor probably lies in the controller, rather than the shift registers. Thus, the designer should concentrate on designing a fast controller.

[3] For the prototype, QL = 32, W = 16.

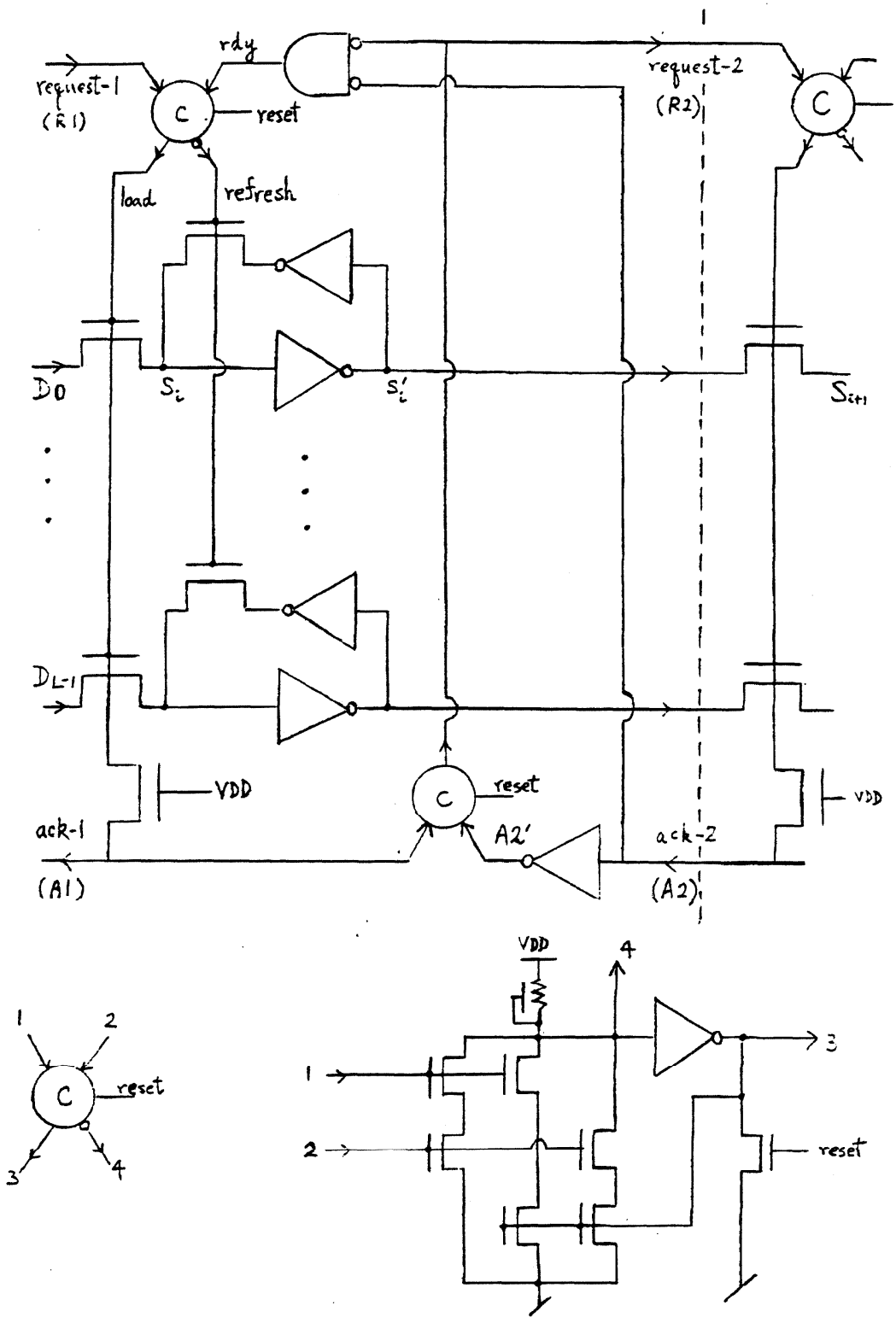

fig. 4.1 Basic Cell of the FIFO

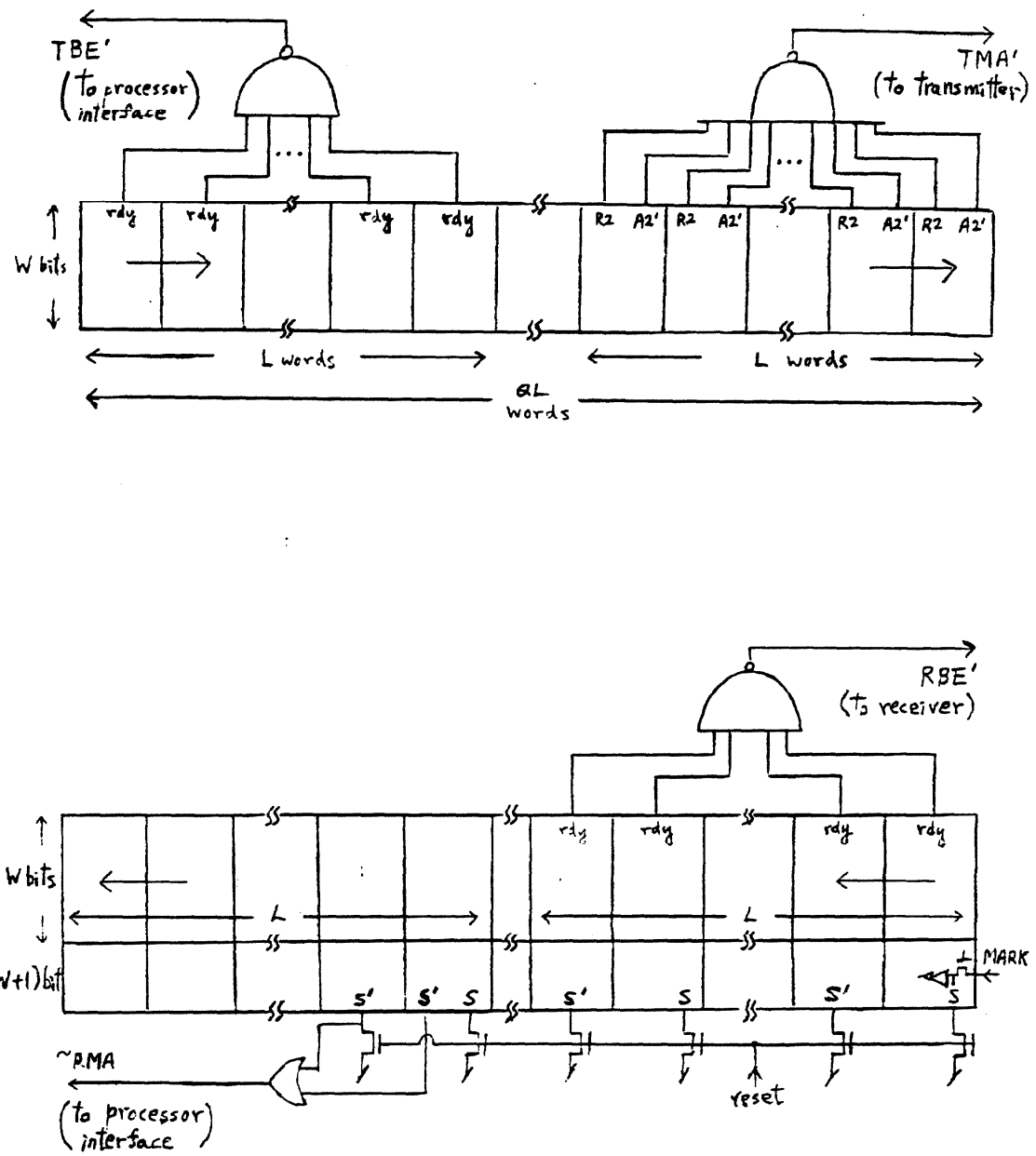

fig. 4.2 Design of the FIFO

an extra pass gate is added into the controlling unit to introduce the necessary delay time.

The circuit is verified by the use of sequence diagrams (fig. 4.3). Arrows in the diagram show the precedence relationship among the different logic signals. Concurrent activities occur when a number of arrows fork off from the same root transition. A dotted arrow means that the precedence must be enforced by the external environment, and not by the circuit itself. An arrow with a little circle shows an assumed precedence. Usually, the designer knows that such an assumption holds in the timing domain. The understanding of the sequence diagrams requires some intuition. With some patience, the reader should be able to see that the circuit works as it is intended. The sequence diagram is helpful in designing speed independent circuits. Other methods have been suggested, such as the use of Petri nets

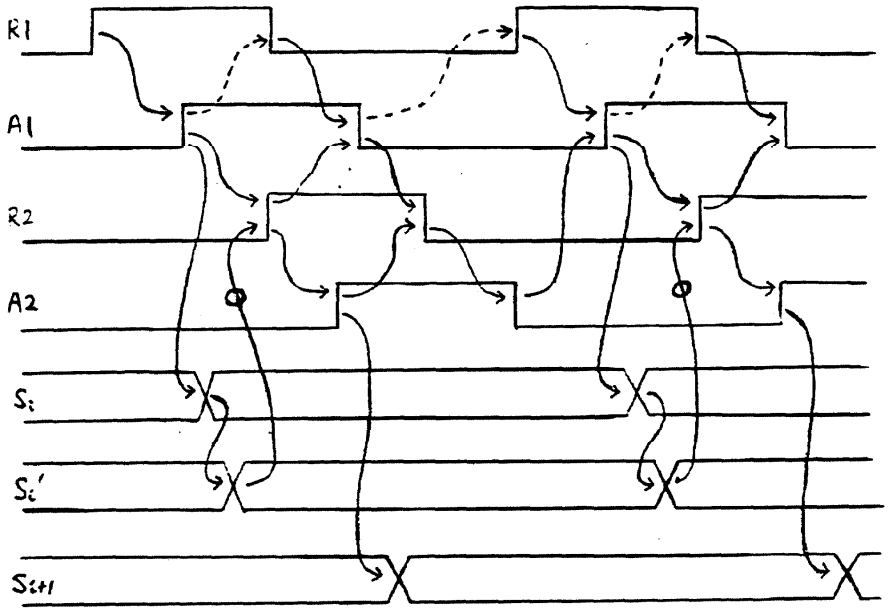

fig. 4.3 Verification of the FIFO

[Seitz 70], and the use of ternary logic simulation [Bryant 80]. However, there is no easy way to simulate this kind of circuit using today's simulators [Whiting 82] . Further research in this area is needed.

4.3.2. Processor Interface

The processor interfacing circuit for the transmitting side of the FIBT is very similar to the one for the receiving side of the FIBT (fig. 4.4). Each of them uses a Muller-C element to handshake with the FIFO, and the remaining logic elements build up the status register and the control register.

4.3.3. Transmitter

The major building blocks of the transmitter are : a parallel-in-serial-out (PISO) shift register, a multiplexer, two latches, two counters, and a PLA (fig. 4.5). The PISO shift register is for shifting out the data bits serially. The multiplexer picks the correct data word that should be loaded into the shift register. Latch 1 stores the flag which decides whether a control character should be sent off. Latch 2 stores the flag which grants the permission to send off a data packet. One of the two counters is the bit counter which keeps track of which bit is being sent off. The other counter keeps track of which word is being sent off. The PLA builds up the transmitter's finite state machine (FSM). The logic states of the FSM is given in fig. 4.6. There are two major paths in the FSM : one is for sending out a control character, and the other is for sending out a data packet. Notice how a data bit is transferred for every three clock cycles. The construction of the FSM is shown in fig. 4.7.

In this part of the circuit, most of the logic has been incorporated into the PLA, and the amount of random logic elements has been minimized. The idea is to keep the design simple, regular, and structural. Once the PLA has

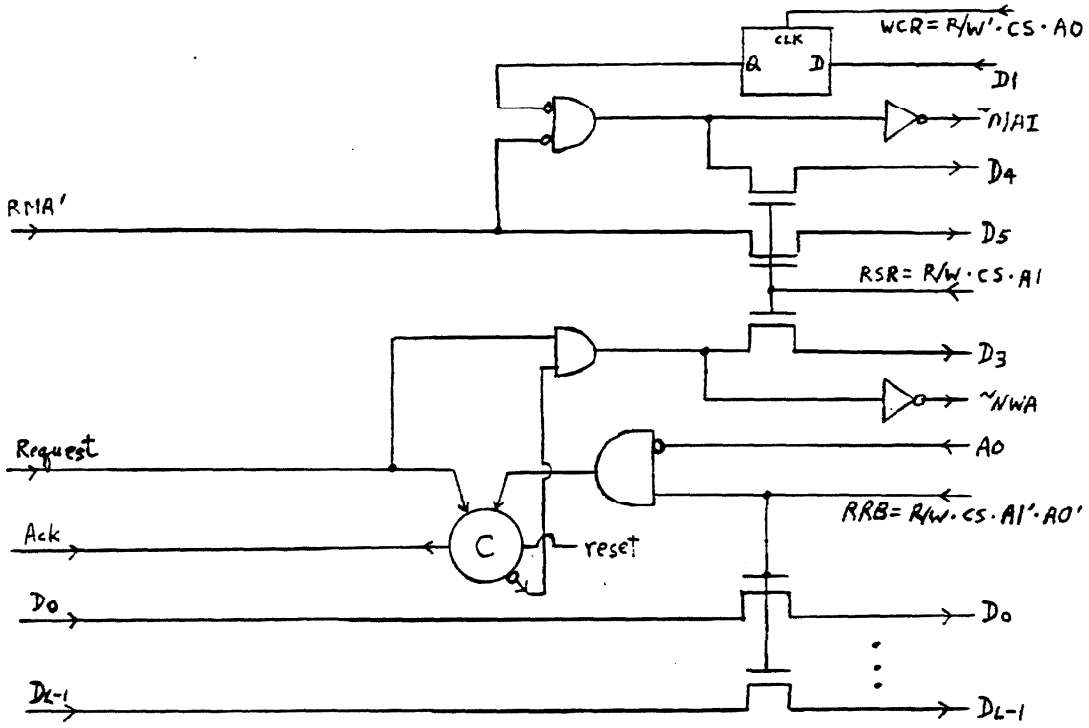

fig. 4.4a Circuit of the Receiver Processor Interface

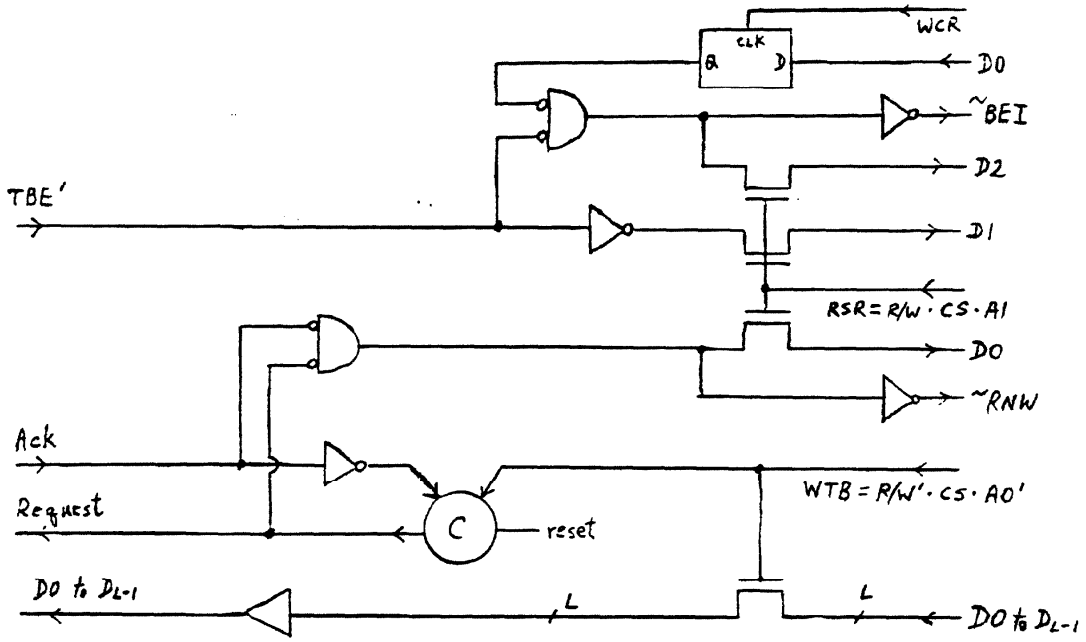

fig. 4.4b Circuit of the Transmitter Processor interface

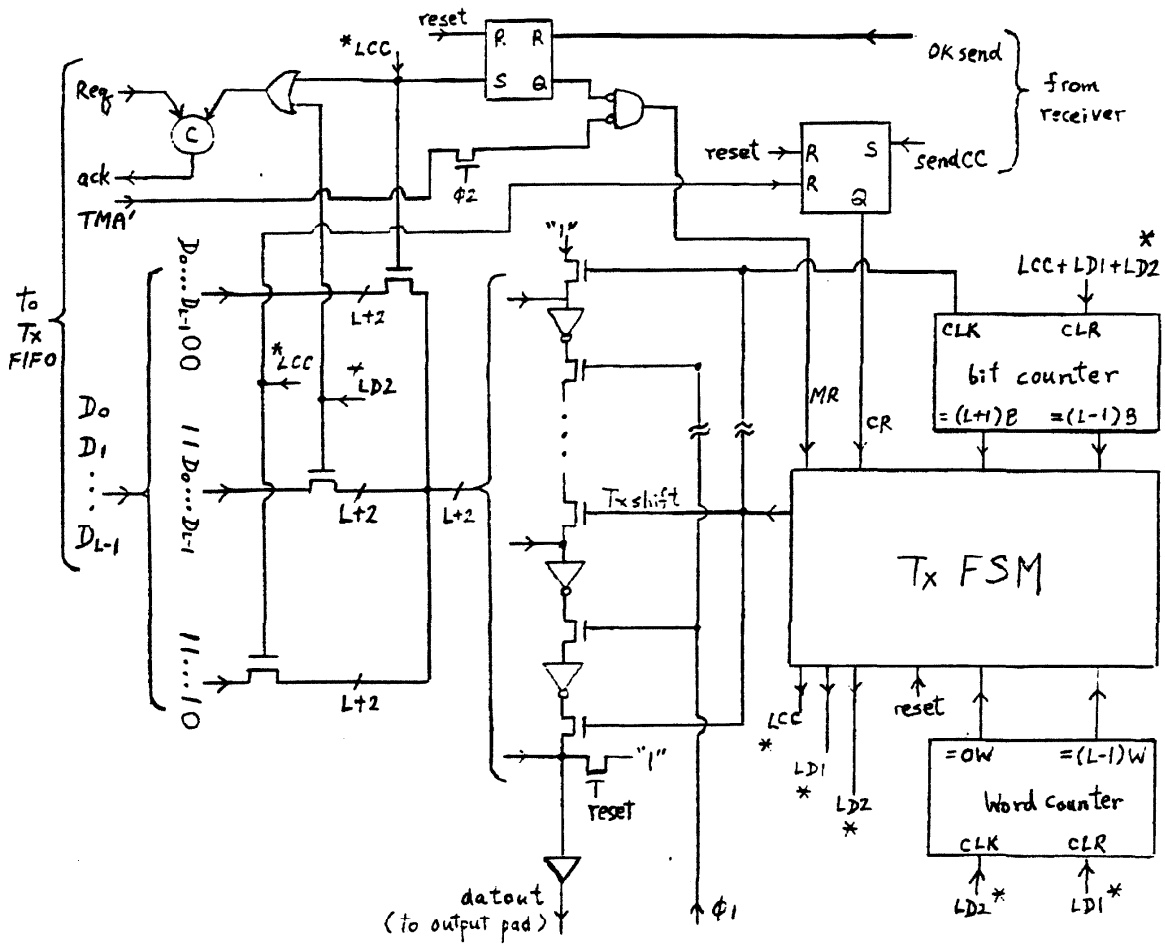

fig. 4.5 Circuit of the Transmitter

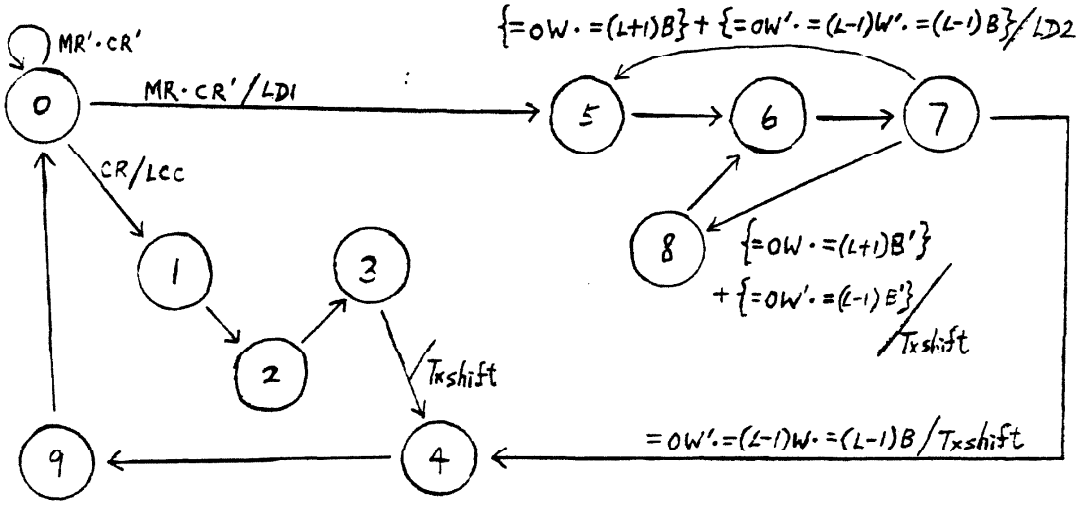

fig. 4.6 State Diagram of the Transmitter FSM

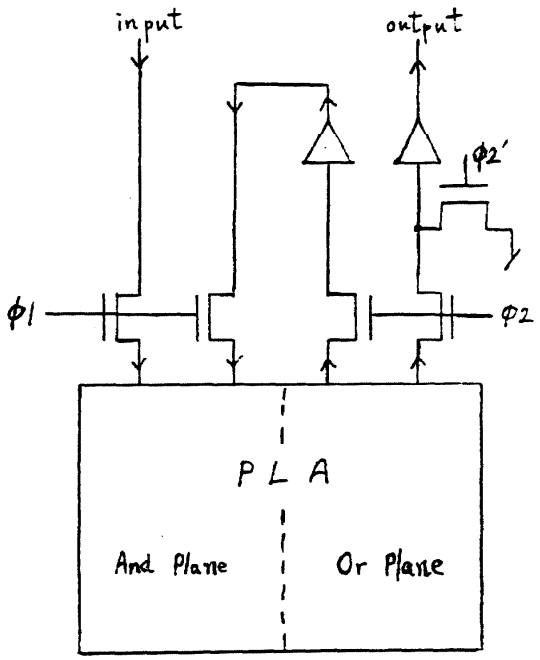

fig. 4.7 Finite State Machine

been tested to be working correctly, the designer can have more confidence in the logic design of the transmitter.

4.3.4. Receiver

The receiver is similar to the transmitter. The major building blocks of the transmitter are : a serial-in-parallel-out (SIPO) shift register, two counters, and a PLA (fig. 4.8). The SIPO shift register is for shifting in the data bits serially, and loading them into the FIFO. The two counters are the bit counter and the word counter.

Again, the main portion of the logic is included in the receiver's FSM. There are two major paths in the FSM (fig. 4.9). One is for receiving a control character, and the other is for receiving a data packet. State 8 is for monitoring the receiving queue to determine if there is enough room for a complete data packet. If so, a signal is sent to the transmitter's latch 1 to initial the sending of an "empty" control character. Also, whenever an "empty" control character has been received from another FIBT, a signal is

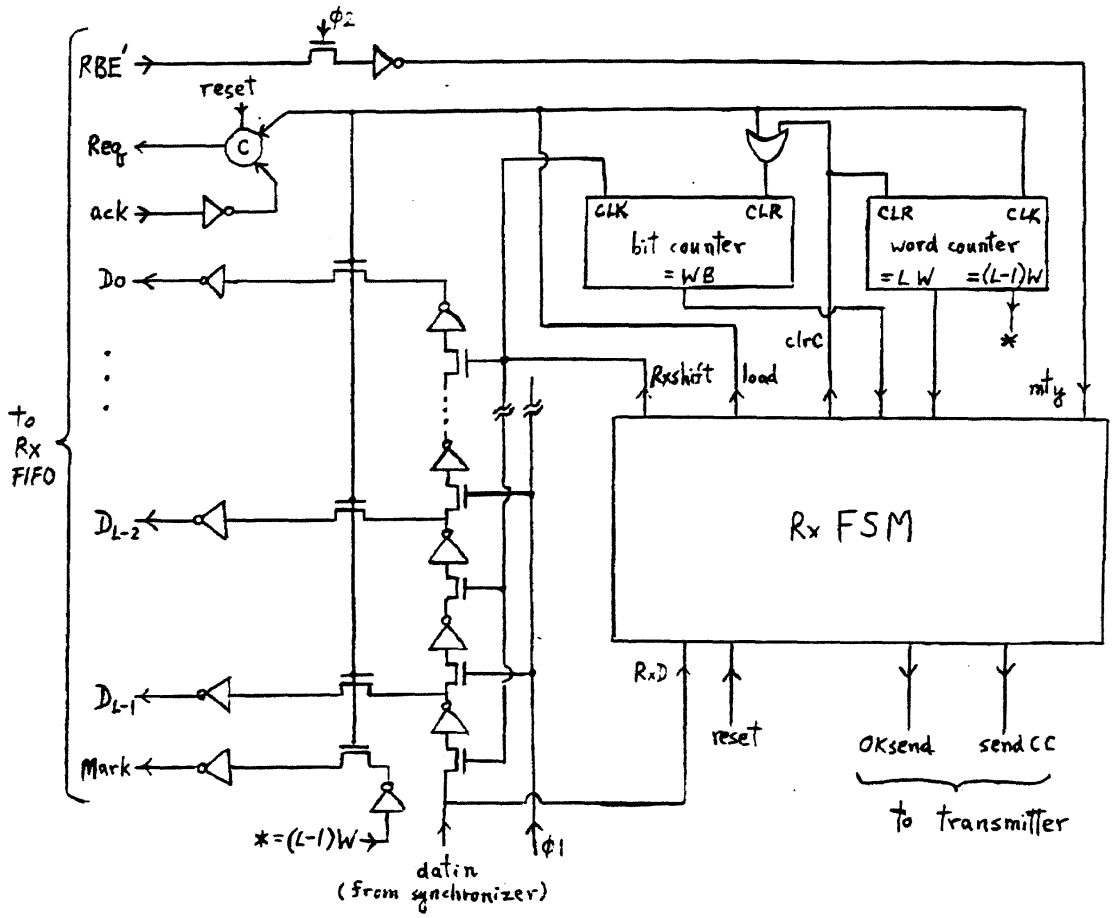

fig. 4.8 Circuit of the Receiver

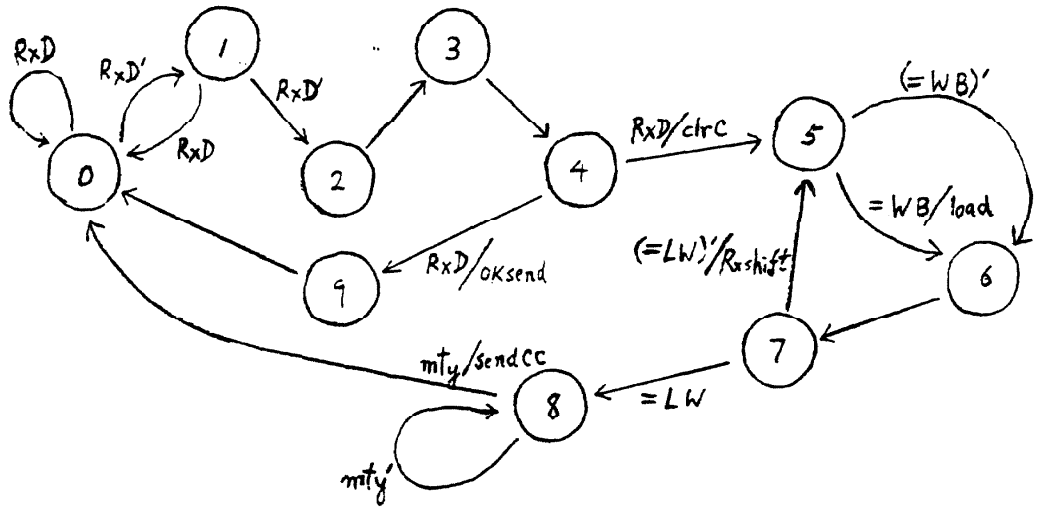

fig. 4.9 State Diagram of the Receiver FSM

sent to the transmitter's latch 2 to grant the permission to send off a data packet.

4.4. The Layout

The prototype FIBT, which carries two 16*32-bit FIFOs, has about 11,000 transistors. The chief design tool used to lay out the chip is Earl [Kingsley 82] – an integrated circuit design language developed at Caltech. The designer first describes each leaf cell independently, then composes the leaf cells together to form the whole structure. The designer can constrain each leaf cell in a way that it can be stretched to fit with other leaf cells. The ability to stretch a cell turns out to be very helpful in designing wiring cells. The designer need not worry where each wire should exactly go; Earl routes it correctly for you. Also, most of the wiring cells can be scaled as the whole FIBT is scaled by parameters.

The code of an example leaf cell, "deco", is given in appendix A, and the cell is drawn in fig. 4.10. "Deco" is a parameterized cell capable of generating any one of the gates which build up the address decoder. The final decoder is drawn in fig. 4.11.

From the simple example of the decoder, the reader can conceive how powerful a parameterized cell can be. In fact, not only the leaf cells are parameterized, the whole FIBT layout is parameterized.

Three layouts have been done for the FIBT. The first layout is a non-parameterized FIBT which carries two 16*32-bit FIFOs (QL = 32, L = 16, W = 16), and the layout's size is about 4,800 micron * 4,600 micron (fig. 4.12). In trying to understand the layout, the reader may find the floor plan given in fig. 4.13 helpful. The second layout is a parameterized FIBT with the same FIFO size, and it is about 5,000 micron * 5,100 micron large (fig. 4.14). The

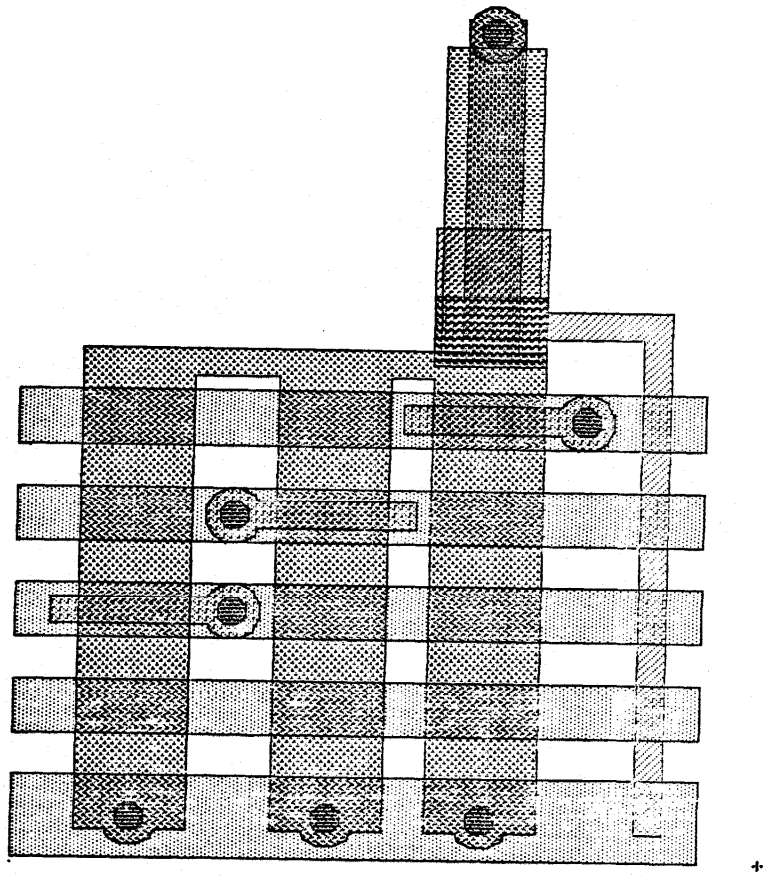

fig. 4.10 An Instance of "Deco"

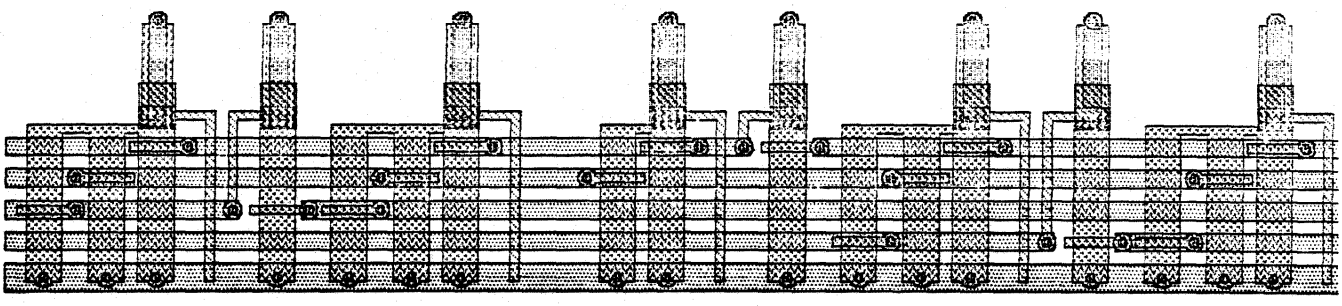

fig. 4.11 The Decoder

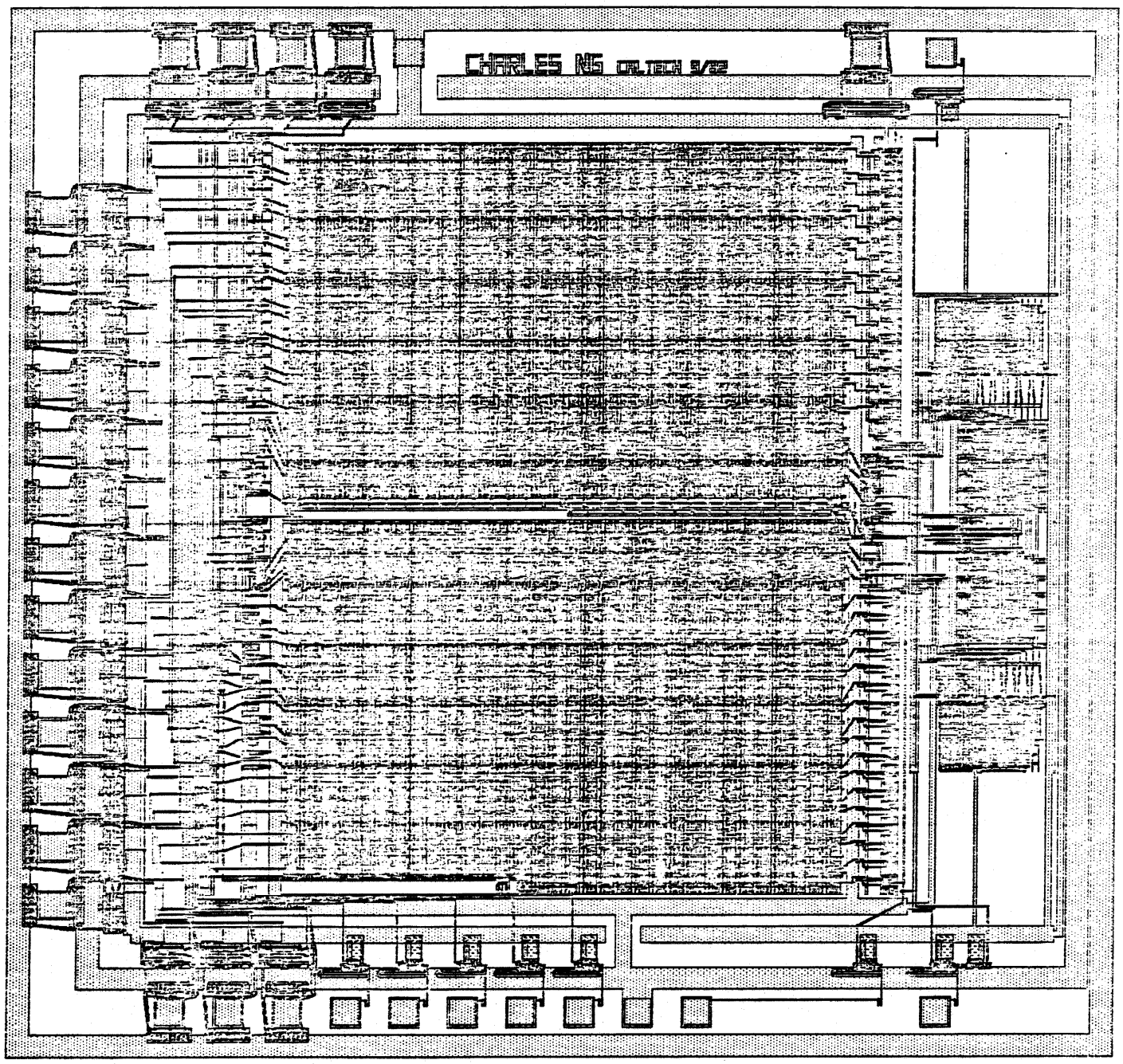

fig. 4.12 Non-parameterized FIBT with 16*32-bit FIFOs

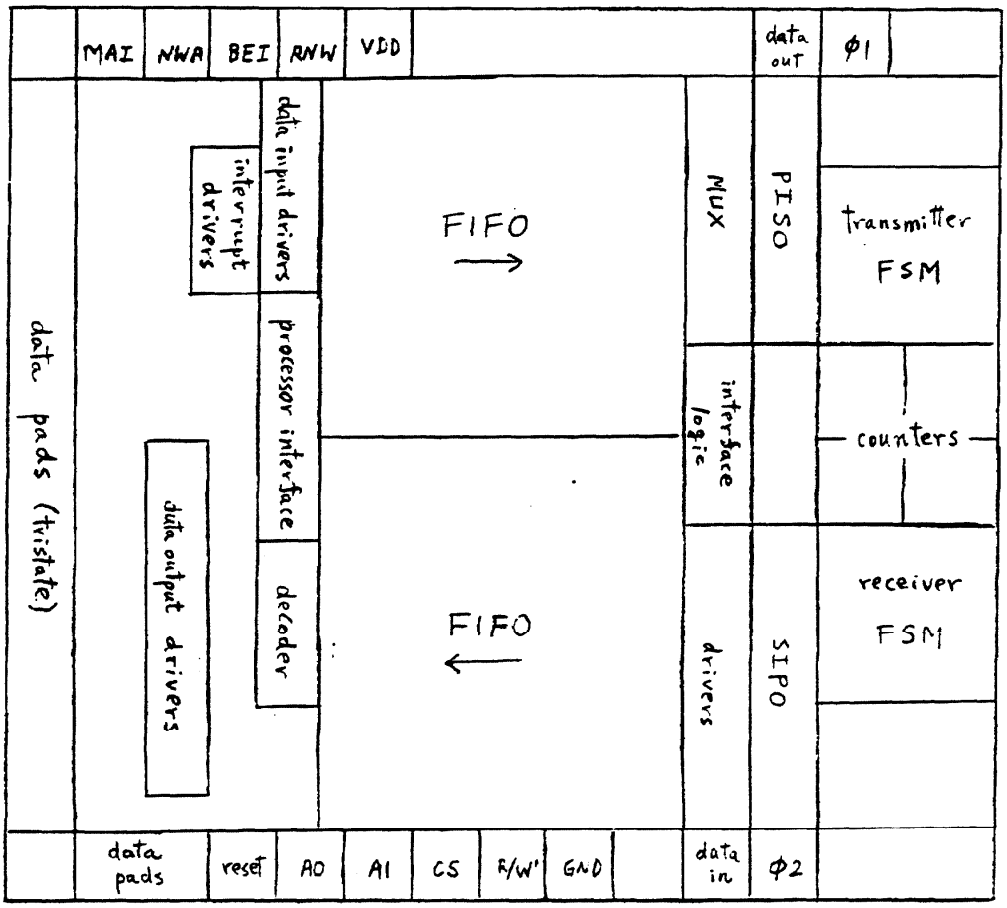

fig. 4.13 Floor Plan of the FIBT

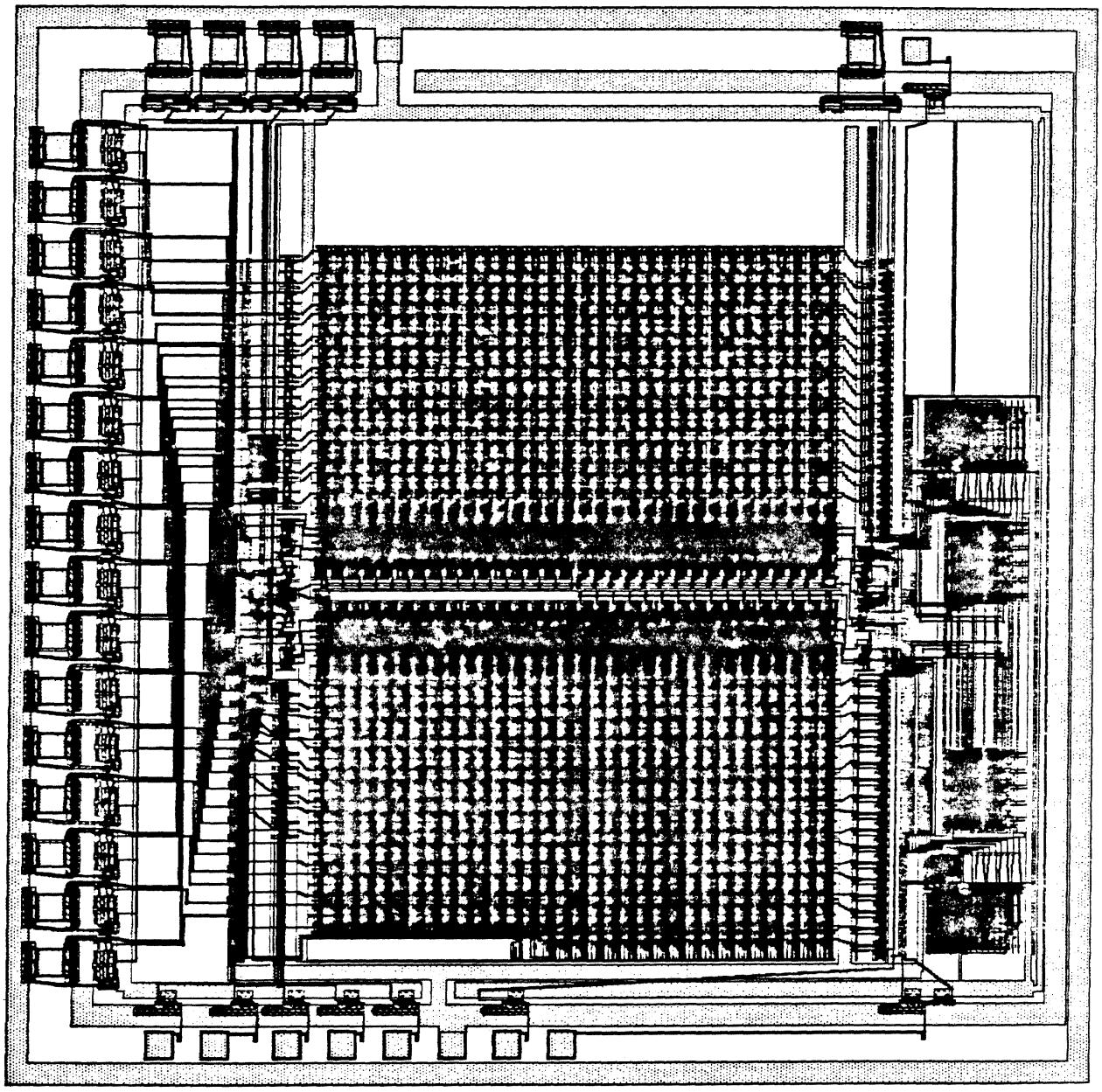

fig. 4.14 Parameterized FIBT with 16*32-bit FIFOs

third layout is done by changing the parameters of the second layout to generate a new FIBT with two 8*10-bit FIFOs (QL = 10, L = 4, W = 8), and the size is 3,200 micron * 3,400 micron (fig. 4.15). The first layout took about three man-months to do, and the second one took an additional man-month. In contrast, the third one only took about one man-hour. With a little extra work, a whole family of FIBTs can be generated [4].

By comparing the three different layouts, we can easily see that more silicon area is traded for a more generalized layout. For instance, in the parameterized layout, all the data tri-state pads are laid on one side of the chip. Also, the wiring cell next to the tri-state pads gives a channel for each data line. Whereas in the non-parameterized layout, the chip is tailored to give a particular FIBT. The tri-state pads are wrapped around the chip, and the wiring cell gives a channel to every two data lines.

In additional to parameterization, another concept used in laying out cells is the idea of "included bus". "Buses run through rather than around the functional blocks." [Rowson 80]. The idea is best illustrated by looking at some of the wires around the decoder (fig. 4.16). The data bus runs through the decoder, and goes to the pass gates which control what data should be given to the output drivers. The output drivers then drive the data to the output pads. Also, there are control buses which run "underneath" the output drivers. The idea is to put transistors on the top of data buses, and not to bring buses to transistors.

[4] The present Earl code has several limitations on the parameters. All parameters must be even integers. QL must be no less than 2*L. L must be greater than 4. And W must be greater than 6.

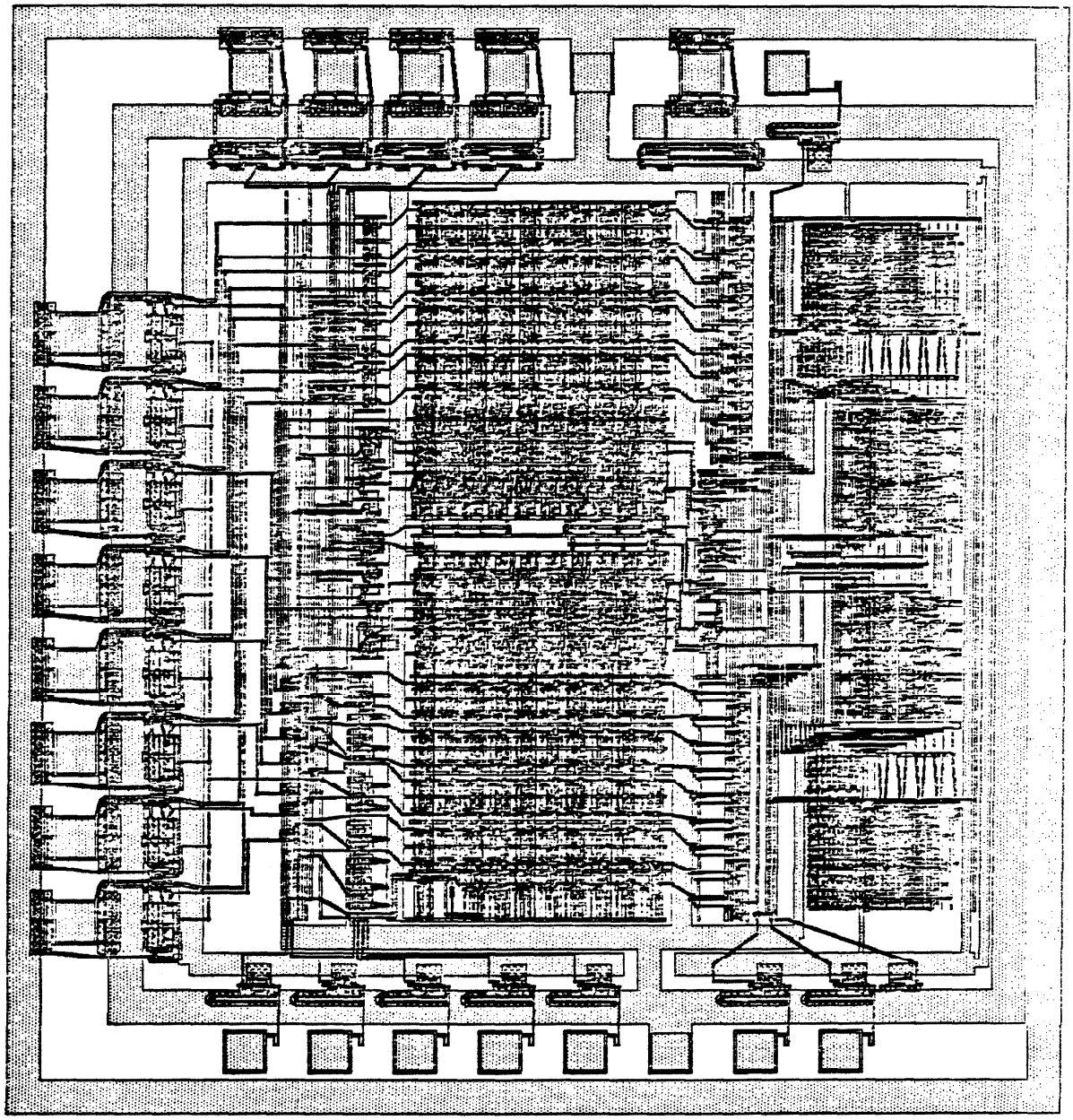

fig. 4.15 Parameterized FIBT with 8*10-bit FIFOs

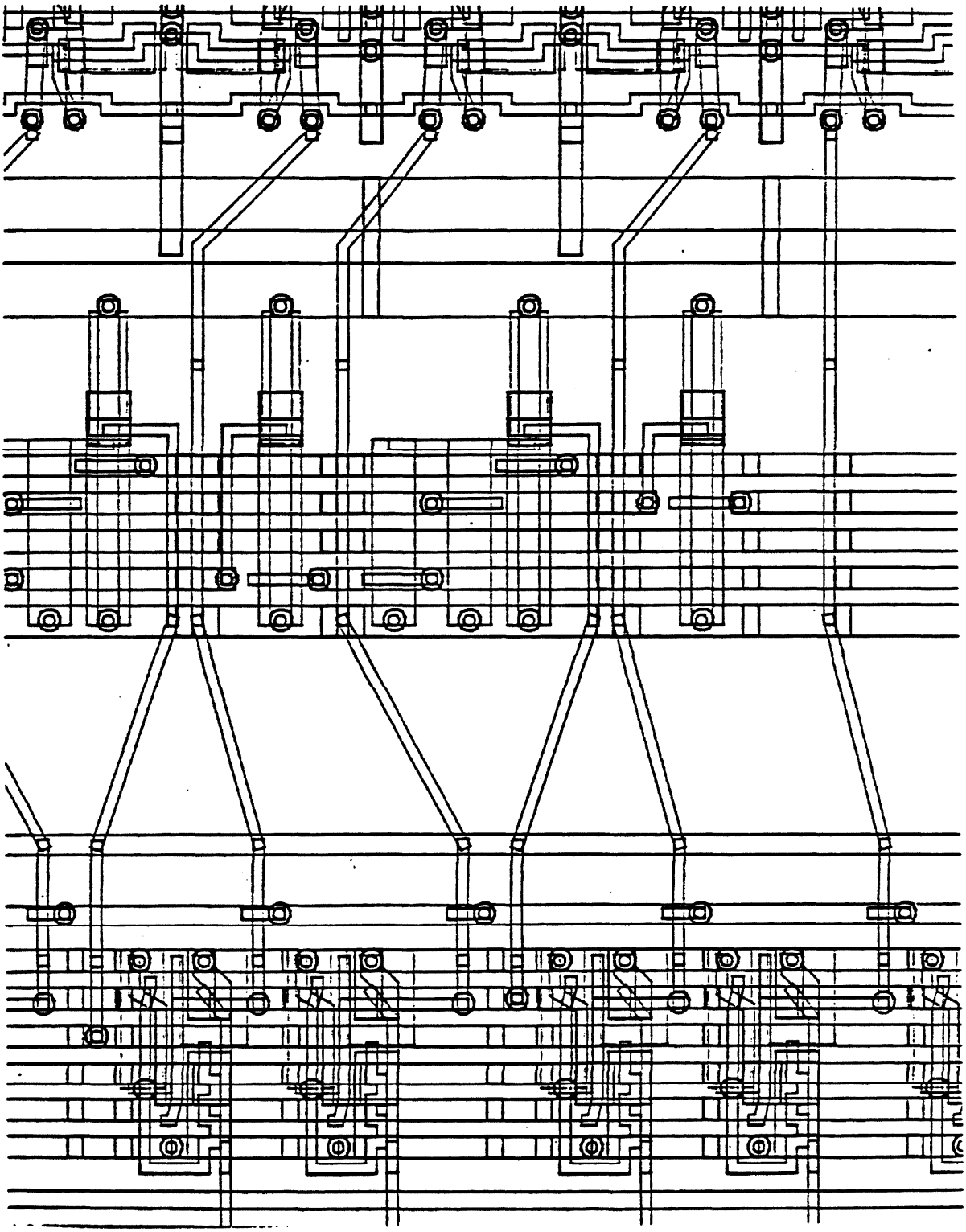

fig. 4.16 Buses Around the Decoder

4.5. Verification

VLSI design has brought out design problems that have never appeared before in man's history. Today, VLSI designers must rely on a new set of computer aided design tools. Among all these design tools, geometrical design rule checkers and simulators are probably the most important ones.

4.5.1. Geometrical Design Rule Checking

A complete geometrical design rule checking program is not available at Caltech at the time of this research. Fortunately, the hierarchical design organizes the checking by eye to a systematic process [Rowson 80].

A limited design rule checking program is used to check the layout. After running Earl, a complete "Caltech Intermediate Form" (CIF) file [Mead & Conway 80] is generated. The CIF file is then extracted [Hedges 82] to generate a network file for simulation run. In extracting the CIF file, limited diagnostics are done on the CIF file. Layout errors, such as shorted power lines, implant layer errors, incomplete one contact transistor, and multiple pullups, can be detected.

4.5.2. Simulation

There are several kinds of simulator which simulate circuits on different levels of abstraction. On the low level side, there is the circuit simulator, such as SPICE [Nagel 75], which models the electrical behavior of every semiconductor device. On a higher level, there is the switch-level simulator, such as MOSSIM [Bryant 82], which keeps track of all the 1's and 0's in the system. Besides these, there is the functional block simulator [5], which simulates the circuit on the functional block level. Among all these

[5] At the time when this research is done, a functional block simulator called "Register Transfer Simulator" is under construction at Caltech.

simulators, there is the tradeoff between how accurately the simulators model the circuits and how fast they can run. For the critical sections where timing can be a problem, the circuit simulators should be used. Switch-level simulators are good for detecting logic design errors. And functional block simulators are good for designing the overall architecture.

In simulating the FIBT circuit, a bottom-up approach is used. A small part of the circuit is extracted for an accurate simulation run. The behavior of this part of the circuit is abstracted for a higher level simulation run, which runs faster and simulates a larger part of the circuit. This approach proves to be rather satisfactory, though how one should abstract the result of a simulation run and put that into a higher level simulation run is still a research topic.

In terms of timing, the FIFO buffer is the most critical section in the FIBT. Thus, SPICE is used to determine the behavior of the FIFO buffer (fig. 4.17). In designing the FIFO, a sequence diagram is used to verify the design. However, there is an assumption : the data lines become valid before the control lines do. In order to guarantee that the assumption holds, a certain amount of delay should be introduced to the control signal path. The amount of delay is controlled by the size of the extra pass gate shown in fig. 4.1. The greater the resistance the pass gate has, the more delay the control signal path contains. SPICE helps to assure that the delay is enough to make the assumption hold.

Also, SPICE helps to estimate the response time of the FIFO. From fig. 4.17, we see that after request-1 has been raised, it takes about 80ns before request-2 is raised, and this is the time taken to propagate the request signal down a stage of the FIFO. Also, acknowledge-1 returns to low in about 130ns

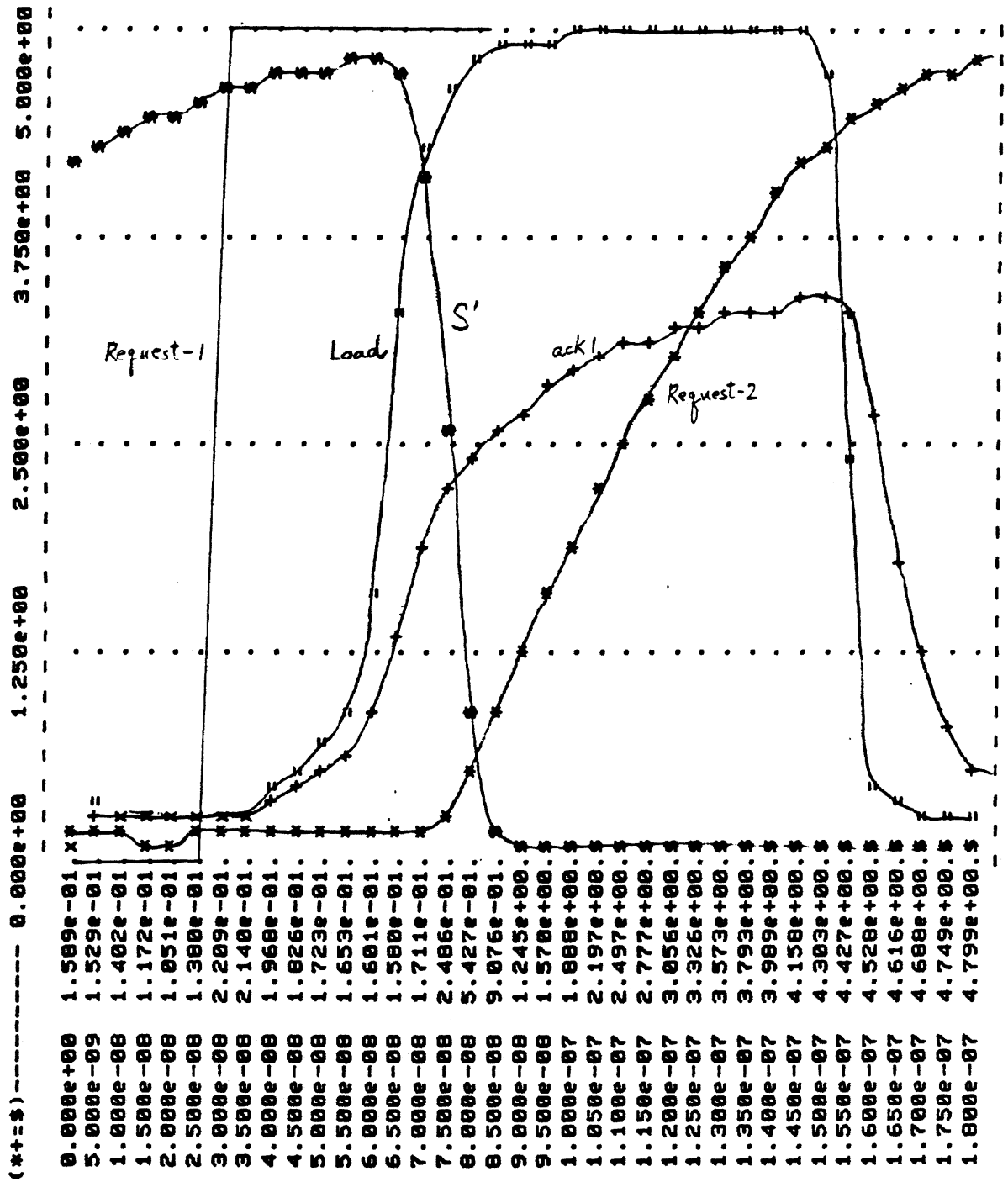

fig. 4.17 Sample Run of SPICE

after request-1 is raised, and this is how long the FIFO takes to handle an input data word.

Although the SPICE program models the MOS devices rather accurately, the final simulation run may produce substantial errors, depending upon how accurately the electrical parameters and the transistors' sizes are specified. The electrical parameters used are drawn from the MOSIS service [MOSIS] that helped to fabricate these chips. The transistors' sizes are obtained by extracting the CIF file produced by Earl. Another program is used to convert the extraction output file into a SPICE input file. The extraction program has a big limitation in deciding the transistors' sizes[6] . For a crooked transistor, like the one shown in fig. 4.18, only the length and width of the square covered by the transistor, and the area covered by the transistor itself are obtained. Owing to this limitation, the resistance and the strength of such a

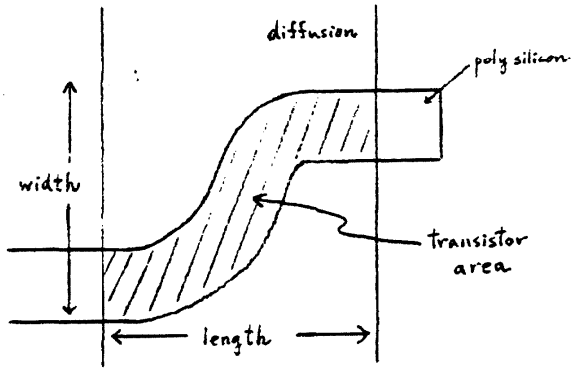

fig. 4.18 Crooked Transistor

[6] A more accurate extraction program is not available at the time of this research.

transistor cannot be modelled correctly. Thus, the final results of these simulation runs have to be interpreted carefully.

In additional to the circuit simulator, MOSSIM, a switch-level simulator, is used to simulate the FIBT. A sample simulation run is shown in appendix B. Again, the CIF file is extracted, and a program is used to convert the extraction output file into the MOSSIM input file. Because sizes of transistors are not modelled accurately, some transistors are edited to give a set of more accurate parameters to MOSSIM. In running MOSSIM, unit time delay is assumed for each transistor. Although this does not represent the true picture, there is no other easy way to simulate the complete circuit. If ternary simulation is used, the unclocked feedback circuit in the FIFO controller will kill the simulation [Bryant 80]. Again, this is the problem of simulating a self-timed circuit. Nevertheless, MOSSIM is used with great success, and it helps to find a number of design and layout errors.

APPENDIX A

Earl Code for the Decoder

```
/* noi (no. of inputs) : 1-3 */
/* gate1-3 (gate position) : 0 = no gate; 1 = a1; 2 = a0; 3 = cs; 4 = rw */
/* outp (output position) : 0=go south; 1=a1; 2=a0; 3=cs; 4=rw */
cell deco(noi,gate1,gate2,gate3,outp);

/* external ports declaration */
if outp=0 then south group out; fi;
west group gnd, a0, a1, cs, rw, vdd;
east := west;

/* internal ports declaration */
ports x1, x2, x3, x4, x5;
if outp<>0 then ports out; fi;

/* setting constraint on the ports */
constr
if outp=0 then xcon west |>7| x1 |>14| x2 |>11| x3 |>12| out |>2| east;
        xcon x1 |>7| x4 |>| x3 |>7| x5;
        else xcon west |>3| x4 |>10| x3 |>7| x5 |>2| east; fi;
ycon south |>| gnd |>5+3| a0 |>7| a1 |>7| cs |>7| rw |>17| vdd
        |>| north;

geom
a11 := x4.x#a1.y;
a12 := x4.x#a0.y;
a13 := x4.x#cs.y;
a14 := x4.x#rw.y;
a21 := x4.x#a1.y;
a22 := x4.x#a0.y;
a23 := x4.x#cs.y;
a24 := x4.x#rw.y;
a31 := x5.x#a1.y;
a32 := x5.x#a0.y;
a33 := x5.x#cs.y;
a34 := x5.x#rw.y;
b := dm x3.x#gnd.y;
c := dm x3.x#north.y+11;
d := x1.x#gnd.y;
e := x2.x#gnd.y;
metal, 6 wire gnd;
        if outp=0 then 4 wire a1;
                4 wire a0;
                4 wire cs;
                4 wire rw; fi;
```

```
        if outp=1 then 4 wire a31, east.x#@.y;
                4 wire a21, west.x#@.y;
                4 wire a0;
                4 wire cs;
                4 wire rw; fi;
        if outp=2 then 4 wire a32, east.x#@.y;
                4 wire a22, west.x#@.y;
                4 wire a1;
                4 wire cs;
                4 wire rw; fi;
        if outp=3 then 4 wire a33, east.x#@.y;
                4 wire a23, west.x#@.y;
                4 wire a1;
                4 wire a0;
                4 wire rw; fi;
        if outp=4 then 4 wire a34, east.x#@.y;
                4 wire a24, west.x#@.y;
                4 wire a1;
                4 wire a0;
                4 wire cs; fi;
poly    box x3.x-4#rw.y+6, x3.x+4#@.y+8;
        if gate1=1 then pm a11; wire a11, x1.x-5#@.y; fi;
        if gate1=2 then pm a12; wire a12, x1.x-5#@.y; fi;
        if gate1=3 then pm a13; wire a13, x1.x-5#@.y; fi;
        if gate1=4 then pm a14; wire a14, x1.x-5#@.y; fi;

        if gate2=1 then pm a21; wire a21, x1.x-5#@.y; fi;
        if gate2=2 then pm a22; wire a22, x2.x+5#@.y; fi;
        if gate2=3 then pm a23; wire a23, x2.x+5#@.y; fi;
        if gate2=4 then pm a24; wire a24, x2.x+5#@.y; fi;

        if gate3=1 then pm a31; wire a31, x3.x-5#@.y; fi;
        if gate3=2 then pm a32; wire a32, x3.x-5#@.y; fi;
        if gate3=3 then pm a33; wire a33, x3.x-5#@.y; fi;
        if gate3=4 then pm a34; wire a34, x3.x-5#@.y; fi;

        if outp=0 then wire x3.x#rw.y+7, out.x#@.y, @.x#south.y; fi;
        if outp=1 then pm a21;
                wire x3.x#rw.y+7, a21.x#@.y, a21; fi;
        if outp=2 then pm a22;
                wire x3.x#rw.y+7, a22.x#@.y, a22; fi;
        if outp=3 then pm a23;
                wire x3.x#rw.y+7, a23.x#@.y, a23; fi;
        if outp=4 then pm a24;
                wire x3.x#rw.y+7, a24.x#@.y, a24; fi;

diff,   4 wire b, c;
        8 wire b, @.x#rw.y+4;
        if noi=3 then wire x3.x#rw.y+4, x1.x#@.y;
                8 wire x1.x#rw.y+4, @.x#gnd.y; dm d;
                8 wire x2.x#rw.y+4, @.x#gnd.y; dm e; fi;
        if noi=2 then wire x2.x#rw.y+4, x3.x#@.y;
                8 wire x2.x#rw.y+4, @.x#gnd.y; dm e; fi;
```

```
buried box x3.x-4#rw.y+4, x3.x+4#@.y+5;
implant       box x3.x-3.5#rw.y+4.5, x3.x+3.5#c.y-1;
end;

/* define all the gates needed */
wr3 horiz deco(3,2,3,4,0);
rd3 horiz deco(3,1,3,4,0);
read horiz deco(2,0,3,4,0);
a1inv horiz deco(1,0,0,1,1);
a0inv horiz deco(1,0,0,2,2);
csinv horiz deco(1,0,0,3,3);
rwinv horiz deco(1,0,0,4,4);

/* final composition to make a decoder */
decoder horiz rd3, a1inv, rd3, read, rwinv, wr3, a0inv, wr3, csinv;
```

APPENDIX B

Sample MOSSIM Run

```
>
>comment declaration of some identifiers
>
>clock phi1.in:0100 phi2.in:0001
>comment data1 & data2 are tri-state data pads
>comment due to some technical problems, bit-13 is not there
>vector data1 0.tri 1.tri 2.tri 3.tri 4.tri 5.tri 6.tri 7.tri
>vector data2 8.tri 9.tri 10.tri 11.tri 12.tri 14.tri 15.tri
>comment inter is the output interrupt signals
>vector inter rnw.out tbe.out nwa.out_node71
>comment addr is the control lines & address lines
>vector addr res.in a0.in a1.in rw.in
>
>vector tfcon tfreq tfack tffill
>vector rfcon rfreq rfack rfmty
>vector tpcon tpreq tpack tpmty
>vector rpcon rpreq rpack rprnw
>
>comment tpladd shows the state of the TxFSM
>vector tpladd tpla8 tpla7 tpla6 tpla5
>comment tplaout : Txshift LCC LD2 LD1
>vector tplaout tpla1 tpla2 tpla3 tpla4
>comment tplain : MR CR reset =15B =17B =0W =15W
>vector tplain tpla9 tpla10 tpla11 tpla12 tpla13 tpla14 tpla15
>
>comment rpladd shows the state of the RxFSM
>vector rpladd rpla8 rpla7 rpla6 rpla5
>comment rplaout : Rxshift load OKsend sendCC clrC
>vector rplaout rpla1 rpla2 rpla3 rpla4 rpla42
>comment rplain : RxD mty reset =16W =16B
>vector rplain rpla9 rpla10 rpla11 rpla12 rpla13
>
>comment shift1 & shift2 is the transmitter PISO
>vector shift1 dout.out 0.shi1 1.shi1 2.shi1 3.shi1 4.shi1 5.shi1 6.shi1
        7.shi1 8.shi1
>vector shift2 9.shi1 10.shi1 11.shi1 12.shi1 13.shi1 14.shi1 15.shi1
        16.shi1
>comment shift3 & shift4 is the receiver SIPO
>vector shift3 17.shi2 18.shi2 19.shi2 20.shi2 21.shi2 22.shi2
        23.shi2 24.shi2
>vector shift4 25.shi2 26.shi2 27.shi2 28.shi2 29.shi2 30.shi2
        31.shi2 32.shi2 din.in
>
>comment monitor some of the signals
```

```
>
>watch /4 inter addr cs.in data1 data2
>watch /4 tpcon rpcon tfcon rfcon
>
>watch /4 tpladd tplain tplaout
>watch /4 shift1 shift2
>
>watch /4 rpladd rplain rplaout
>watch /4 shift3 shift4
>
>comment define some useful constants
>
>const reset 0000
>const rb    1001
>const fwr   1101
>const sr    1011
>const tb    1000
>const cr    1100
>
>comment increase the simulation step limit
>limit step:200
>
>comment start the simulation
>
>comment reset the FIBT
>set addr:reset cs.in:0 din.in:1
>cyc
  1.4| inter:0X11 addr:0000 cs.in:0 data1:XXXXXXXX tpcon:000 rpcon:000
     | tfcon:001 rfcon:000 tpladd:0000 tplain:001XXXX tplaout:0000
     | shift1:1XXXXXXXXX shift2:XXXXXXXX rpladd:0000 rplain:111XX
     | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX1 data2:XXXXXXX
>comment write to the control register to enable interrupt lines
>set addr:cr /2 cs.in:1 /3 cs.in:0
>for data1:11XXXXXX
>cyc
  2.4| inter:0011 addr:1100 cs.in:0 data1:11XXXXXX tpcon:000 rpcon:000
     | tfcon:001 rfcon:000 tpladd:0000 tplain:000XXXX tplaout:0000
     | shift1:1XXXXXXXXX shift2:XXXXXXXX rpladd:0000 rplain:110XX
     | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX1 data2:XXXXXXX
>comment write a data word to the transmitting buffer
>set addr:tb /2 cs.in:1 /3 cs.in:0
>for data1:11110000 data2:1100100
>cyc
  3.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
     | tfcon:101 rfcon:000 tpladd:0000 tplain:000XXXX tplaout:0000
     | shift1:1XXXXXXXXX shift2:XXXXXXXX rpladd:0000 rplain:110XX
     | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX1 data2:1100100
>comment force the transmitting buffer message available true
>comment notice that the transmitter is activated
>for tffill:0
>cyc
  4.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
     | tfcon:010 rfcon:000 tpladd:0101 tplain:0000010 tplaout:0001
```

```
    | shift1:0011110000 shift2:11001X00 rpladd:0000 rplain:110XX
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX1 data2:1100100
>comment force a start bit on the serial data input line & activate
>comment the receiver
>for din.in:0
>cyc
 5.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0110 tplain:0000010 tplaout:0000
    | shift1:0011110000 shift2:11001X00 rpladd:0001 rplain:010XX
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
>cyc 2
 6.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0111 tplain:0000010 tplaout:0000
    | shift1:0011110000 shift2:11001X00 rpladd:0010 rplain:010XX
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
 7.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:1000 tplain:0000010 tplaout:1000
    | shift1:0111100001 shift2:1001X001 rpladd:0011 rplain:010XX
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
>comment set the message type bit 0, i.e. a data packet
>cyc 3
 6.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0110 tplain:0000010 tplaout:0000
    | shift1:0111100001 shift2:1001X001 rpladd:0100 rplain:010XX
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
 9.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0111 tplain:0000010 tplaout:0000
    | shift1:0111100001 shift2:1001X001 rpladd:0101 rplain:01000
    | rplaout:00001 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
10.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:1000 tplain:0000010 tplaout:1000
    | shift1:1111000011 shift2:001X0011 rpladd:0110 rplain:01000
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
>comment let the 1st data bit be 0
>cyc 3
11.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0110 tplain:0000010 tplaout:0000
    | shift1:1111000011 shift2:001X0011 rpladd:0111 rplain:01000
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
12.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0111 tplain:0000010 tplaout:0000
    | shift1:1111000011 shift2:001X0011 rpladd:0101 rplain:01000
    | rplaout:10000 shift3:XXXXXXXX shift4:XXXXXXXX0 data2:1100100
13.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:1000 tplain:0000010 tplaout:1000
    | shift1:1110000110 shift2:01X00111 rpladd:0110 rplain:01000
    | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXX10 data2:1100100
>comment let the 2nd data bit be 1
>for din.in:1
>cyc 3
14.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
    | tfcon:000 rfcon:000 tpladd:0110 tplain:0000010 tplaout:0000
    | shift1:1110000110 shift2:01X00111 rpladd:0111 rplain:11000
```

```
   | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXXX11 data2:1100100
15.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
   | tfcon:000 rfcon:000 tpladd:0111 tplain:0000010 tplaout:0000
   | shift1:1110000110 shift2:01X00111 rpladd:0101 rplain:11000
   | rplaout:10000 shift3:XXXXXXXX shift4:XXXXXXX11 data2:1100100
16.4| inter:0011 addr:1000 cs.in:0 data1:11110000 tpcon:000 rpcon:000
   | tfcon:000 rfcon:000 tpladd:1000 tplain:0000010 tplaout:1000
   | shift1:1100001100 shift2:1X001111 rpladd:0110 rplain:11000
   | rplaout:00000 shift3:XXXXXXXX shift4:XXXXXX101 data2:1100100
>
>comment end of sample run
```

BIBLIOGRAPHY

Bryant 80	Randy Bryant "A Switch-level Simulation Model for Integrated Logic Circuits" PhD Thesis, MIT, 61.
Bryant 82	Randy Bryant, Mike Schuster, & Doug Whiting "MOSSIM II: A Switch-level Simulator for MOS LSI User's Manual" Technical Report 5033, Caltech, Aug 82.
Buchanan 80	Irene Buchanan "Modelling and Verification in Structured Intergrated Circuit Design" Technical Report 3642, PhD Thesis, Caltech, May 80.
Clark 67	W. A. Clark "Macromodular Computer System" AFIPS Conference Proceedings, 30:335-6, Spring, 1967
Clark & Seitz 77	Becky Clark & Chuck Seitz "Asynchronous Self Timed Queue" U.S.Patent, 4,058,773 Nov.1977
Cohen 77	Danny Cohen "FIFO Implementation" Display File #554, Caltech, March 77.
Galvin & Kohn 78	Greg Galvin & Jack Kohn "FIFO Implementation" Display File #1717, Caltech, May 78.

Hedges 82	Tom Hedges
	"CIF Extractor Help File"
	SSP Program Documentation, Caltech, 82.
Kingsley 82	Chris Kingsley
	"Earl : An Integrated Circuit Design Language"
	Technical Report 5021, MS Thesis, Catech, Feb 82.
Lam 83	Jimmy Lam
	"RTsim User Guide"
	Caltech, Nov 82.
Lang 82	Charles R. Lang
	"The Extension of Object-Oriented Languages to a Homogeneous
	Concurrent Architecture"
	Technical Report 5014, PhD Thesis, Caltech, May 82.
Locanthi 80	Bart Locanthi
	"The Homogeneous Machine"
	Technical Report 3759, PhD Thesis, Caltech, Jan 80.
Mead & Conway 80	Carver Mead & Lynn Conway
	"Introduction to VLSI systems"
	Addison Wesley, 1980
Mead & Sutherland 77	Carver Mead & Ivan Sutherland
	"Microelectronic and Computer Science"
	Scientific American, vol 237, pp 210-229, Sept 77.
Moore 79	G. E. Moore
	"Are We Really Ready for VLSI?"
	Caltech Conference on VLSI, Jan 79.
MOSIS	MOSIS
	Information Science Institute, U. of Southern California

Nagel 75 — L. W. Nagel
"SPICE2 : A Computer Program to Simulate Semiconductor Circuits"
University of California at Berkeley, 75.

Rowson 80 — James Rowson
"Understanding Hierarchical Design"
Technical Report 3710, PhD Thesis, Caltech, April 80.

Seitz 70 — Chuck Seitz
Asynchronous Machines Exhibiting Concurrency
In "Proceedings of the Project MAC Conference on Concurrent Systems and Parallel Computation"
ACM Conference Record, Woods Hole, Massachusetts, June 70.

Seitz 80 — Chuck Seitz
System Timing
"Intorduction to VLSI System" by C. Mead & L. Conway
pp. 236 - 242, Addison-Wesley, 1980

Seitz 82 — Chuck Seitz
"Ensemble Architecture for VLSI - A Survey and Taxonomy"
Proceedings of the Conference on Advanced Research in VLSI, M.I.T., Jan 82.

Sutherland 76 — Ivan Sutherland
"First In First Out Stack"
Display file #172, Caltech, Aug 76.

Whiting 82 — Doug Whiting
"A Self-timed Chip Set for Multiprocessor Communication"
Technical Report 5000, MS Thesis, Caltech, Feb 82.