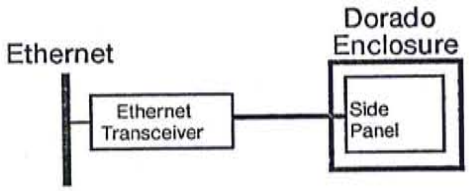
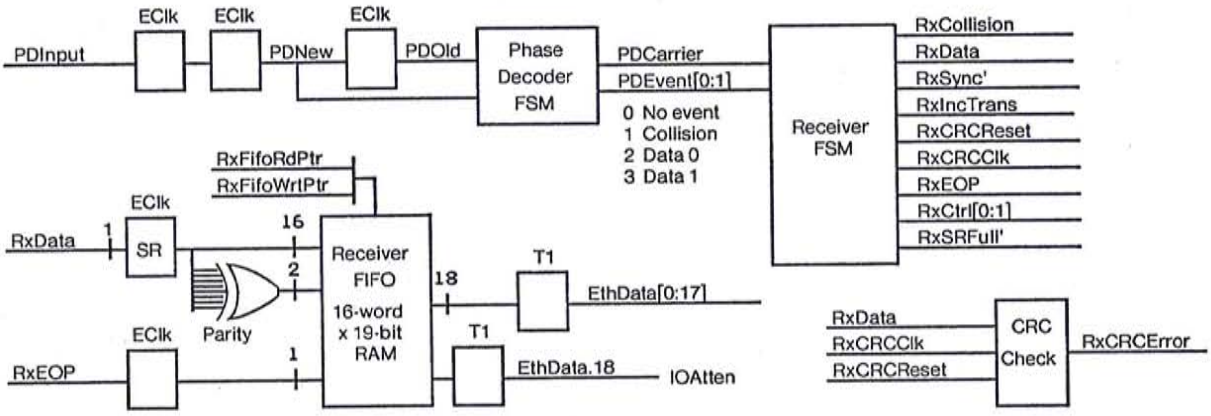
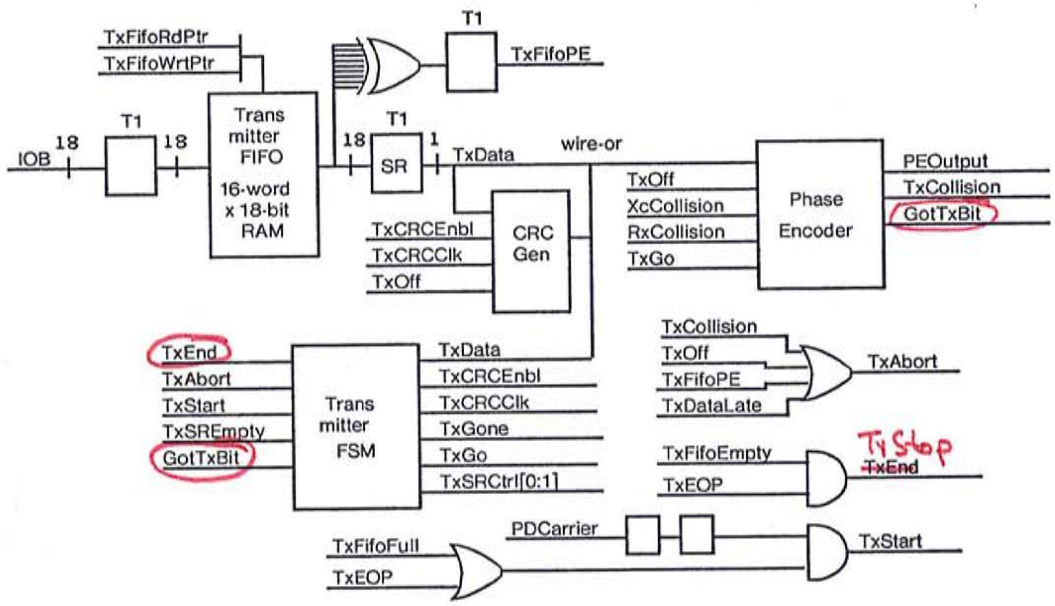




Receiver

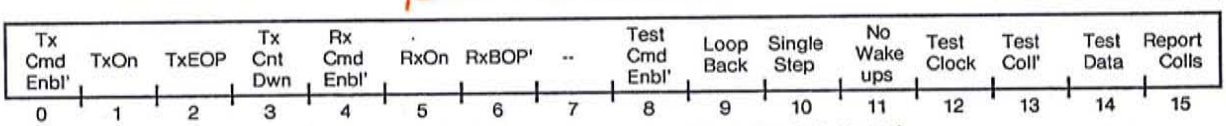


Transmitter

TIOA = 016

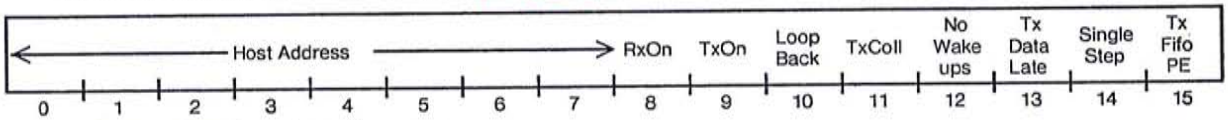
RcvR Status

EthC Output←B



TestCmdEnbl' enables setting of LoopBack, SingleStep, NoWakeups, TestClock, TestColl', TestData, and ReportColls
RxCmdEnbl' enables setting of RxOn and RxBOP'
TxCmdEnbl' enables setting of TxOn and TxEOP, TxCntDwn

EthC Pd←Input



Host Address is set by backpanel jumpers

Figure 16
Ethernet Controller

D1Ethernet.sil
10/3/79

Ethernet Controller

An Ethernet is the principal means of communication between a Dorado and the outside world. An Ethernet is a broadcast multi-access packet switched network which can connect up to 256 stations separated by as much as 1 kilometer with a 3 mHz channel. The 'Ether' is a passive coaxial cable to which each station is connected through a transceiver ~~that is high-impedance when receiving, low-impedance when driving.~~ which inserts and extracts BITS from the Ether.

Readers unfamiliar with the general concepts behind the Ethernet should refer to "Ethernet: Distributed Packet Switching for Local Computer Networks," by R. M. Metcalfe and D. R. Boggs, CACM, 19(7):395-404, July 1976; or to Design and Performance of Local Computer Networks, by John Shoch, published by University Microfilms, August 1979.

Read this chapter with Figure 16 in view.

Ethernet Packets

Ethernet data are encoded in *packets.* Packets are preceded by a low signal (i.e., silence) on the Ether; they begin with a one-bit prefixed by the transmitter, called the *start bit.* Bits in the packet are *phase encoded,* where the bit cell time is nominally 340 ns; phase encoded signals have one *data transition* per bit cell and its direction (low-to-high = 1) is the value of the bit. Midway between these there may be a *setup transition,* so that the next data transition can be in the correct direction.

Packets end when no transitions are detected for more than 1.5 bit times and the Ether is low. *Collisions* are transmissions that overlap in time and cause malformed and undecodable bits. Transmitters *jam* the Ether with a continuous high for several bit times after participating in a collision. Collisions are of four types: *too many transitions,* in which two transitions occur within .25 bit times; *too few transitions,* in which a transition occurs between 1.25 and 1.5 bit times after the last one; *end-of-packet* (EOP), in which no transitions occur for more than 1.5 bit times and the Ether is low; and *jam,* which is the same as EOP except that the Ether is high.

In a well-formed packet that does not experience a collision, the start bit is immediately followed by an 8-bit destination host number, then an 8-bit source host number. This is followed by an indefinite number of 16-bit data words, a 16-bit checksum, and finally silence.

Even when transmitted without a source-detected collision, a packet may fail to reach its destination; *packets are delivered only with high probability.* Stations requiring a lower residual error rate must follow mutually agreed upon communication protocols.

When the sender of a packet detects a collision, some method is needed to arbitrate (without communication) its use of the Ether with other stations contending for it. The algorithm used on the Ethernet, called the 'binary exponential backoff collision algorithm,' is discussed in the above references. It involves waiting a random interval and then reattempting transmission. The (ideal) distribution of the random intervals depends upon many factors.

Remarks

From the method of collision detection, it follows that in a noise free Ether with ideal transmitters and receivers, a bit cell time between 0.75*T and 1.25*T, where T is the nominal bit cell time (340 ns), can be decoded correctly.

Phase encoding has the undesirable property that only 50% of the transmission medium's theoretical bandwidth is utilized. A number of reasonably simple encodings are known that more nearly approach the theoretical limit, though phase encoding is simple to implement. If at some time we were willing to abandon compatibility with the existing Ethernet, we should reconsider the use of phase encoding.

A promising alternative to phase encoding is bit-stuffing, which averages 67%, 86%, or 93% of theoretical bandwidth for 0th, 1st, and 2nd order codes. This encoding outputs data bits in a cell time equal to 1/2 of the phase-encoded cell time; when 1 (0th order), 2 (1st order), or 3 (2nd order) data bits have been output without a transition, then a non-data transition is inserted into the bit stream. The 1st order encoding (86%) could be implemented with a few changes to the current controller.

Controller Overview

The Ethernet controller is a slow IO device packaged with the disk controller on the DskEth logic board. These two devices require more edge pins than are available in an MSA-IO slot, so the board must be mounted in a Fast IO slot (see Figure 2).

> It would be possible to package two Ethernet controllers on one logic board using different task and TIOA assignments for each. This might be appropriate if Dorados are ever used as Ethernet gateways.

A cable connects the controller to a *transceiver* outside the Dorado enclosure; this transceiver is almost identical to the ones used for Altos and other computers, the difference being that it uses +12 volts rather than +15. Dorado transceivers are painted bright red and have large block lettering saying "Dorado only". Plugging in the wrong type of transceiver will not damage anything; it just won't work. The cable between the controller and the transceiver contains twisted-pair signals for receiver data, transmitter data, collision, +5 v, and +12 v.

The controller has independent *transmitter* and *receiver* sections. Because these two sections are completely independent, the Dorado can receive its own transmissions. This is an important aid in hardware and software debugging and simplifies the device driver, which need not check for sending to itself. Furthermore, the receiver can receive consecutive packets separated by the minimum inter-packet spacing (510 ns). This means that the Dorado can receive, without loss, streams of packets directed to it by mulitple hosts and packets that immediately follow broadcasts. This capability is important for servers and other high-performance applications.

The controller uses two tasks, one for the transmitter (EOT for Ethernet Output Task) and one for the receiver (EIT for Ethernet Input Task). The receiver task is higher priority. To permit two instruction/wakeup loops, a wakeup request is removed whenever the Next bus says the task is about to run. This simple strategy can be fooled into removing a request when NextLies occurs, but this is harmless since the required service rate is low. To avoid a spurious wakeup, a wakeup is not requested again until after the task has blocked. A debugging control bit can be set which prevents wakeups even when all other conditions are satisfied.

The transmitter and receiver each have 16-word x 20-bit Fifos. The bits are 16 data + 2 parity + 2 spare (the receiver uses one of, the spare bits). Each Fifo has read and write pointers, multiplexed into the address inputs of the storage chips, to select the next location to be read or written; these pointers are zeroed by IOReset. A Fifo is empty when the pointers are equal and full when (WritePtr+1) mod 16 equals ReadPtr. There are *bus registers* between the Fifos and IOB. Service requests from the Ether side of a Fifo are given priority. The Fifos are synchronous

to t_1.

The basic clock for transmitting and receiving data from the Ether, called *EtherClk*, originates from a 23.5 MHz crystal oscillator (i.e., the period is 42.5 ns or 1/8 of the 340 ns bit cell time). The memory system's *Pendulum* clock (period 16 ms) is also used to time retransmissions after a collision, as discussed later.

The receiver runs continually; its *phase decoder (PD)* samples the Ether every EtherClk; a finite state machine (FSM) driven by the samples detects the presence or absence of packets on the Ether, zero/one transitions, and collisions. Another FSM accumulates the status of the packet and controls a shift register that assembles 16-bit words from the incoming data. Words in the shift register are written into the receiver's Fifo together with odd parity on each byte; the status is written into the Fifo after the last word of each packet and marked to distinguish it from data words. This allows the receiver to handle back-to-back packets; firmware decides what to do with each packet as it is read from the Fifo. EtherClk is used for receiver stages through the shift register; data in the shift register is synchronized to the Dorado system clock as it is written into the Fifo.

When the transmitter is turned on, it attempts to send one packet and then must be restarted by firmware. The EOT fills the Fifo; the transmitter FSM loads the shift register from the Fifo and supplies a serial bit stream to the *phase encoder (PE)*. Transmitter status is read directly from the controller status registers (unlike receiver status, which travels through the data path). Data is synchronized to EtherClk between the output of the shift register and the input of the PE. A collision may be detected by either the transceiver or the PD. The occurrence of a collision is captured, synchronized, and used to abort the outgoing packet after jamming the Ether briefly.

The controller has a number of features to help debugging. All of the interesting internal state is available via the IOB and the muffler system. The transceiver can be disconnected and PE output internally connected to PD input under firmware control. Task wakeups can be disabled permitting the controller to be driven entirely from emulator-level software. The internal clock can be single-stepped. These features permit the construction of a simulation program which compares its predictions with what the controller is actually doing.

Receiver

Most of the receiver runs continuously, tracking traffic on the Ether. The PD reports what it sees to the receiver FSM, which assembles packets in the shift register and buffers them in the Fifo. As words emerge from the Fifo into the bus register, they are either discarded or generate a wakeup request under control of the wakeup logic. Following the last data word of each packet as it travels through the Fifo are the CRC word and a status word. IOAtten branches when a status word is present in the receiver bus register. Data and status are synchronized to the Dorado clock between the output of the shift register and the input of the Fifo.

The peculiar placement of status bits eases emulation of the Alto Ethernet controller.

The PD is a FSM which takes in raw phase-encoded serial data and produces phase decoder events and carrier. Phase decoder events are 'saw a zero bit', 'saw a one bit', and 'saw a malformed bit'. Carrier indicates that the PD is seeing transitions on the Ether (i.e. the Ether is in use). Since the PD is completely digital, it can be single-stepped for debugging. Receiver collision detection, a by-product of this decoding technique, works as well as transceiver collision

detection.

The receiver control is another FSM that takes in PD output and produces control and status signals. RxSRCtrl controls the shift register and the bit counter. The bit counter decrements when a data bit is shifted into the shift register and resets to -1 when the status is parallel loaded into the shift register. RxSRFull' is low when the next shift will make the register full. RxEOP travels in parallel with each Fifo word and is true if the word is an ending status word. This bit is called EthData.18 when it is in the bus register where it can be tested with IOAtten.

Writing data or status from the shift register into the Fifo has priority over loading the bus register from the Fifo. Byte parity is computed at the shift register output and travels with the data through the Fifo and the bus register, down IOB and into the processor where it is checked.

The optimum point at which to synchronize received data with the Dorado clock system would be at the input to the PD, where there is only one signal to synchronize, except that this would make proper operation of the PD depend upon the Dorado clock period. The next best sync point is the PD output where the number of signals has only grown to three. The problem here is that the PD can produce events faster than they can be synchronized to the Dorado clock without buffering. Consequently, synchronization takes place after the shift register where the number of signals exceeds 20. This is not as unfortunate as it seems because status and data use the same paths and can share a single synchronizer, RxSRDump, which produces RxFifoWE' each time RxFSM pulses RxSync'. This leaves only RxCollision and PDCarrier which must be synchronized for the transmitter. RxCollision shares a synchronizer with XcCollision, and PDCarrier's is a simple level synchronizer.

A receiver data-late occurs when the receiver FSM requests a Fifo write and the Fifo is full. In this case the write does not happen and the data is lost. RxDataLate is cleared after an end-of-packet status word is successfully written into the Fifo. This status has the data late error bit set so that the EIT is notified that the preceding packet was bad.

is filled, RxOn, RxBOP true, NoWakeups false

EIT wakeup requests occur when the bus register contains an interesting word (and when the EIT is currently blocked, as discussed earlier). Words are interesting if they emerge from the Fifo into the bus register while RxOn and RxBOP are true and NoWakeups is false. RxBOP is set after the status word for a packet is discarded, so that the next word out of the Fifo (presumably the first word of the next packet) can generate a wakeup. It is reset by the EIT to discard the remaining words of a rejected packet (usually because the address didn't match). The receiver may be reset at any time by clearing RxOn. No more wakeups are generated and every word is discarded as it emerges from the Fifo. When RxOn is next set, the receiver will continue to discard words until it has discarded a status word. It will then set RxBOP, and the next word (first word of the first packet after turning on the receiver) will cause a wakeup.

Transmitter

When the transmitter is turned on, it attempts to send one packet and then must be restarted by firmware. At the request of the wakeup logic, the EOT fills the Fifo using Output←B to the bus register. The transmitter FSM loads the shift register from the Fifo and supplies a serial bit stream to the PE. Transmitter status is read directly from the controller status registers (unlike receiver status, which travels through the data path). Data is synchronized to the Ether clock between shift register output and PE input.

EOT wakeups occur when the bus register is empty, TxOn is true, and TxEOP, TxCntDwn, and NoWakeups are false (and when EOT is blocked, as discussed earlier). After delivering the last word of a packet, EOT wakeups are disabled by setting TxEOP. While counting down a collision retransmission interval, firmware can disable wakeups until the next tick of Pendulum by setting TxCntDwn. The transmitter may be reset at any time by clearing TxOn, which stops wakeup requests and shuts down the PE within 2 bit times.

The binary exponential backoff collision algorithm must be implemented in firmware. The controller merely provides a way to generate a wakeup on the next rising edge of Pendulum, making the grain size of countdown intervals 16 μs for the Dorado (compared to 38 μs for Altos and Novas). Note that setting TxCntDwn *prevents* a wakeup; for one to actually occur when Pendulum clears it, the bus register must be empty and TxEOP must be false. ~~Pendulum is considered to be a foreign signal so it is synchronized before being applied to the reset input of TxCntDwn.~~

Loading the shift register from the Fifo has priority over writing into the Fifo from the bus register. Byte parity is computed in the processor and travels with the data down IOB into the bus register, and through the Fifo to the shift register where it is checked.

The transmitter control is a FSM which takes in start, end, and abort signals and produces control signals. TxSRCtrl controls the shift register and bit counter. The bit counter decrements when a data bit is shifted into the shift register and resets to -1 when the next word is parallel loaded into the shift register. TxSREmpty' is low when the next shift will make the register empty. TxData wire-or's the start bit at the beginning of each packet. TxGone clears TxEOP to cause a wakeup at the end of each packet. The transmitter starts when the Fifo is full or, if the packet is less than 15 words long, when TxEOP is true. The transmitter ends normally when the Fifo is empty and TxEOP is true. The transmitter aborts when a collision, Fifo parity error or data late occurs. TxAbort can be tested with IOAtten.

A transmitter data late occurs when the TxFSM requests a Fifo read and the Fifo is empty. The PE sends one random bit and then stops. The resulting packet has an illegal length and probably a bad CRC.

The PE inverts and latches TxData at the start of each bit cell and inverts the latched value 1/2 bit time later. TxGo, synchronized to the beginning of a bit cell, enables the PE. The PE assumes that a data bit is available long before it is needed and acknowledges each bit after latching it by generating TxGotBit.

A collision may be detected by either the transceiver or PD. The occurrence of a collision is captured, synchronized, and used to abort the outgoing packet. The output of the first stage of the TxCollision synchronizer is wire-or'ed with PD output to jam the Ether after a collision. The

jam lasts for one or two bit times, being the delay through the TxCollision synchronizer, TxFSM, and TxGo synchronizer.

Clocks

The controller needs a clock with a nominal frequency of eight times the Ether bit rate. The SingleStep control bit selects either the 23.53 mHz crystal oscillator or single Dorado clocks injected under program control. The clocks for the ~~Ether-synchronous parts of the controller~~ are constructed from this basic clock.

The slowest Dorado clock period at which the transmitter works is 42.5 ns. Disabling the Dorado system clocks while TxOn is true causes a transmitter data late. If TxGo is true, the packet is chopped off, causing an incomplete transmission and probably a runt bit. When the clock is reenabled, the PE sends a few fragmentary bits and then the data late aborts the packet.

The slowest Dorado clock period at which the receiver works is 85 ns. Disabling the Dorado system clocks causes a receiver data late. The next packet that arrives after the clock is reenabled reports the data late.

Task Wakeups

The controller is designed for two completely independent tasks, with the receiver higher priority. Two IOAs select data and status/control registers. IOAtten may be tested to decide whether a wakeup request is just for another word or something special (ending status for the receiver, or PE aborted for the transmitter).

Task wakeups must, on the average, be serviced within 5.44 μs. The transmitter and receiver each have 17 words of buffering (bus register + 15 Fifo + shift register) so the variance can be quite large--accumulated delay of up to about 90 μs is tolerable, while longer delay will cause a data late error.

Muffler Input

All muffled signals on the DskEth board are accessible to Dorado firmware. The method by which a particular signal is selected and read out is discussed in the "Muffler Input" section of the "Disk Controller" chapter. Signal addresses 120_8 to 177_8 for the Ethernet controller are enumerated below. Unless it is obvious, signals which are specific to the receiver or transmitter have Rx or Tx respectively somewhere in their names.

Table 26: Ethernet Muffler Signals

Word Bit	*Name*	*Meaning*
ERX0		
120	PDNew	1/8 bit time sample of PD input signal
121	PDOld	PDNew delayed one sample time
122:125	PDCnt[0:3]	Number of samples since last data transition
126	PDCntCtrl	Increments or clears PDCnt
127	ReportCollisions	Control register bit that enables PD collision reporting
130	RxBOP	"Beginning Of Packet" enables receiver data wakeups
131	EthData.18	Marks status word terminating a packet
132	--	
133	RxCRCError	Output of receiver CRC checker
134	RxDataLate	Receiver Fifo overflowed
135	RxBusRegFull	Word in BusReg can be read with Pd←Input
136	RxFifoFull	Receiver Fifo is full
137	RxFifoEmpty	Receiver Fifo is empty
ETX		
140:142	TxState[0:2]	State of transmitter FSM
143	TxEOP	Transmitter data wakeups are disabled
144	TxBusRegFull'	Word is waiting to be written into the transmitter Fifo
145	TxGone	Transmitter FSM is shut down
146	TxSREmpty'	Transmitter shift register is empty
147	TxCntDwn'	Transmitter wakeups disabled until next pendulum clock
150	TxCRCEnbl	Shift/compute control for transmitter CRC
151	TxGo	Enable PE
152	TxData	Serial data input to PE
153:154	TxSRCtrl[0:1]	Transmitter shift register control
155	PEOutput	Phase Encoder (PE) output
156	TxFifoFull	Transmitter Fifo is full
157	TxFifoEmpty	Transmitter Fifo is empty
ERX1		
160:162	RxState[0:2]	State of receiver FSM
163	RxCollision	Receiver-detected collision
164	PDCarrier	The Ether is in use
165:166	PDEvent[0:1]	PD output (no event, collision, 0, and 1)
167	RxSRFull'	Receiver shift register is full
170	RxEOP	Marks status word terminating a packet
171	RxSync'	True for one cycle triggering write of SR into Fifo
172	RxIncTrans	Receiver incomplete transmission
173	RxCRCReset	Resets receiver CRC chip
174	RxCRCClk	Clocks receiver CRC ship
175	RxData	Serial data output from RxFSM
176:177	RxSRCtrl[0:1]	Receiver shift register control

Bus Registers

TIOA equals 15_8 selects the bus registers. The transmitter bus register is loaded by Output←B and the receiver bus register is read with Pd←Input.

Control Register

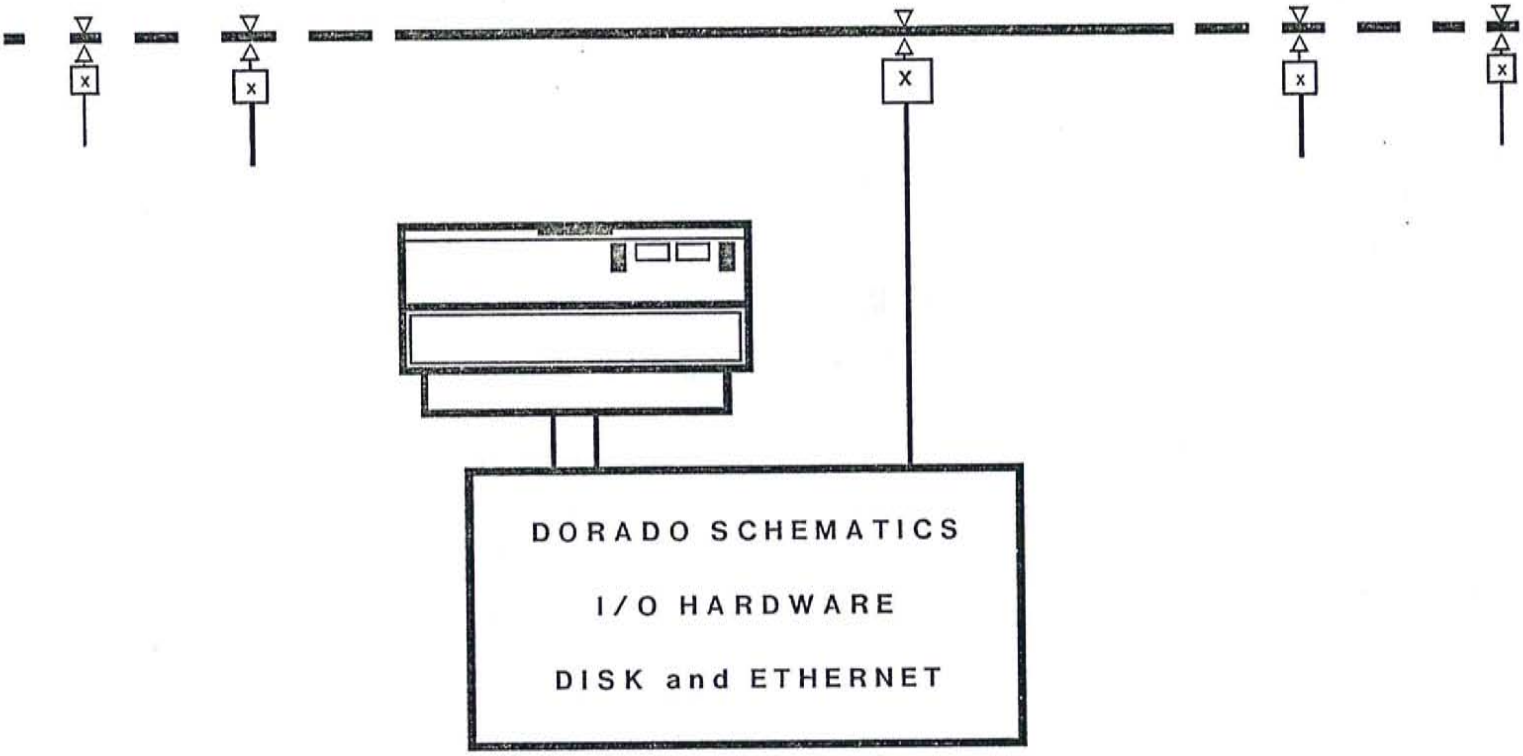
TIOA equals 16_8 selects either the (write-only) control register, discussed here, or the (read-only) status register, discussed in the next section. The control register has three fields: transmitter, receiver, and test. Bits in a field are decoded only if the command-enable bit for the field is true. Control bits with a single quote as their last character are true when zero.

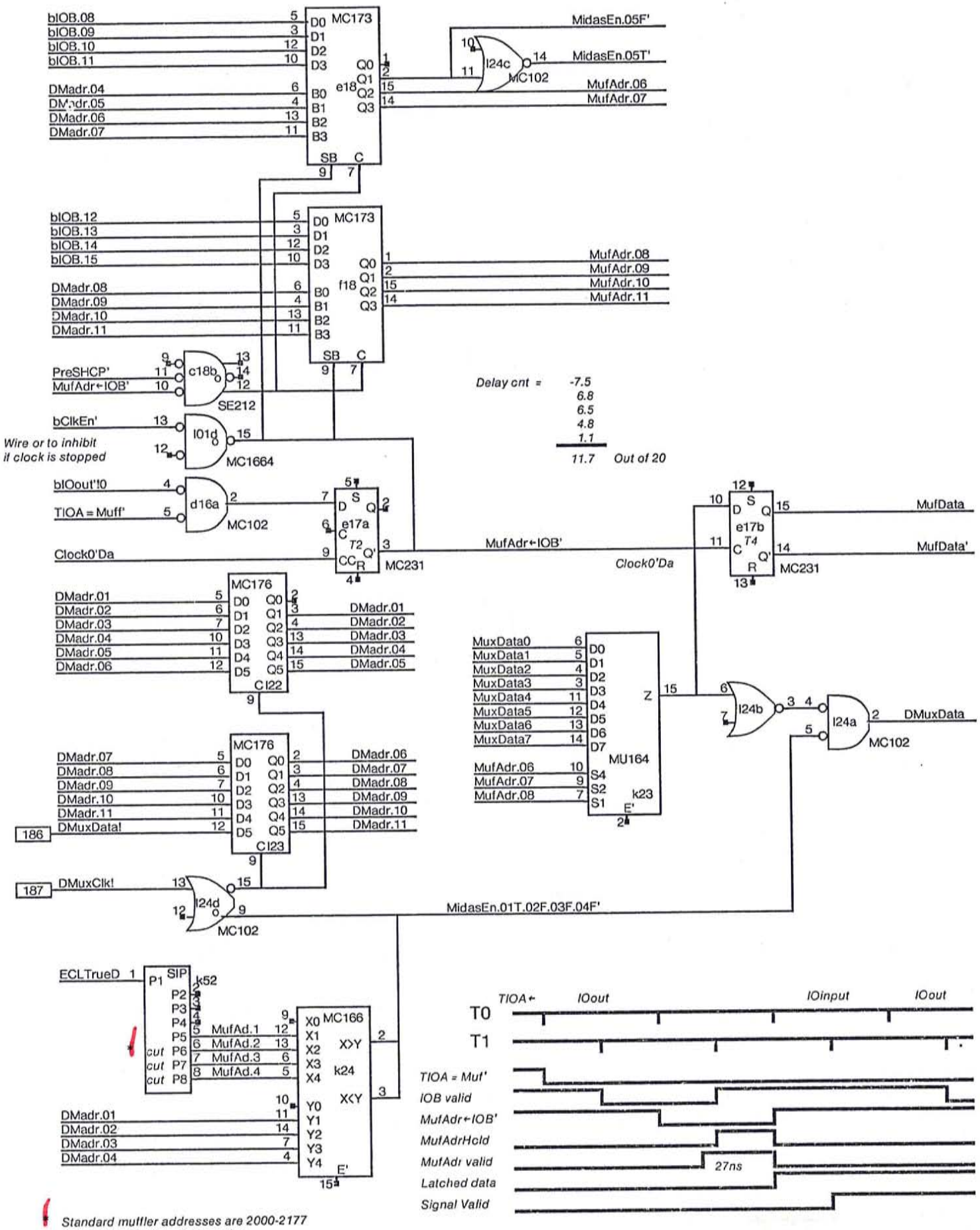
TxCmdEnbl'	enables decoding of transmitter commands.
TxOn	enables the transmitter. The transmitter may be reset at any time by clearing this bit. Cleared by IOReset.
TxEOP	disables transmitter wakeups. EOT sets this bit after outputing the last word of a packet. It is cleared by the controller when the PE shuts down after an abort or normal end. Cleared by TxOn=0.
TxCntDwn	disables transmitter wakeups. Set by EOT to time a retransmission interval after a collision; cleared by the controller when the next rising edge of Pendulum occurs (period = 16 μs). N.B. the binary exponential backoff is done by firmware. Cleared by TxOn=0.
RxCmdEnbl'	enables decoding of receiver commands.
RxOn	enables the receiver, which may be turned off at any time by clearing this bit. Cleared by IOReset.
RxBOP'	disables receiver wakeups. Cleared by EIT to discard the currently arriving packet; set by the controller when the first word of the next packet is available. Cleared by RxOn=0.
TestCmdEnbl'	enables decoding of test commands
LoopBack	disconnects the transceiver, loops PE output to PD input, and enables TestColl'. Cleared by IOReset.
SingleStep	disables the 23.53 mHz oscillator. Changing this bit can produce a runt clock. Reset the transmitter first and expect an occasional bad receiver status. Cleared by IOReset. Before and after changing this bit
NoWakeups	disables all controller wakeups. Cleared by IOReset.
TestClock	injects a single Dorado clock pulse (t_3 of the Output instruction) into the EtherClk logic. SingleStep must already be set.
TestColl'	injects a single Dorado clock pulse (t_3 of the Output instruction) into the collision synchronizer. LoopBack must already be set.
TestData	wire ORs with PD input. LoopBack must already be set and TxOn must already be false. Do not issue TestClock in an instruction that changes TestData. Cleared by IOReset.
ReportCollisions	allows the PD to report malformed bits as collisions. Cleared by IOReset.

Status Register

TIOA of 16_8 also selects the (read-only) status register. The bits in this register are the most interesting to the microcode. Less interesting state is available from the mufflers.

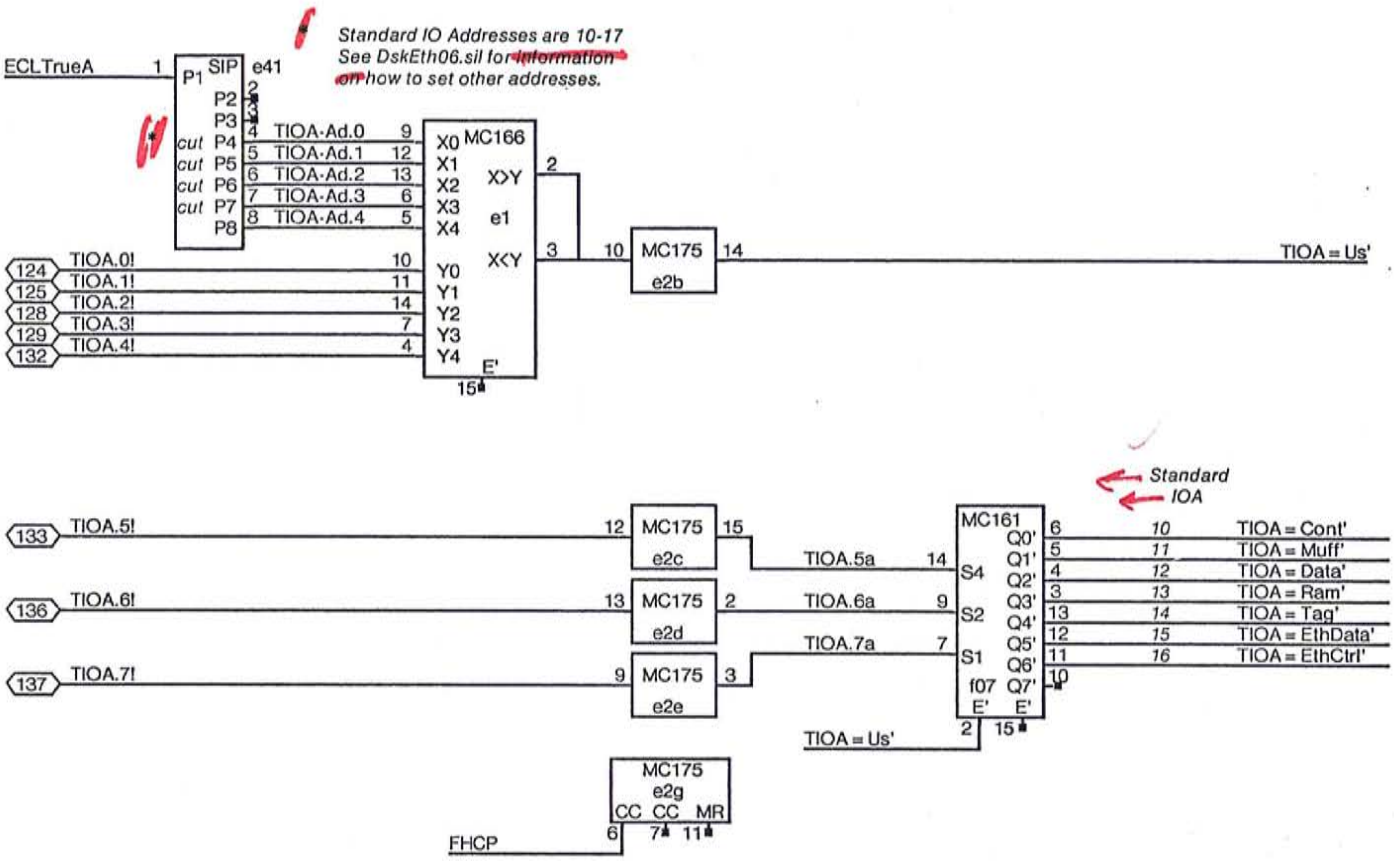
Host Addr	the host address set by pullups on the backplane.
RxOn	the receiver is enabled.
TxOn	the transmitter is enabled.
LoopBack	the interface is looped back.
TxColl	the current output packet was aborted by a collision.
NoWakeups	all wakeups are disabled.
TxDataLate	the current output packet was aborted by a data late.
SingleStep	the 23.53 mHz oscillator is disabled.
TxFifoPE	the current output packet was aborted by a parity error.





Standard muffler addresses are 2000-2177
See DskEth06.sil for information
on how to set other addresses

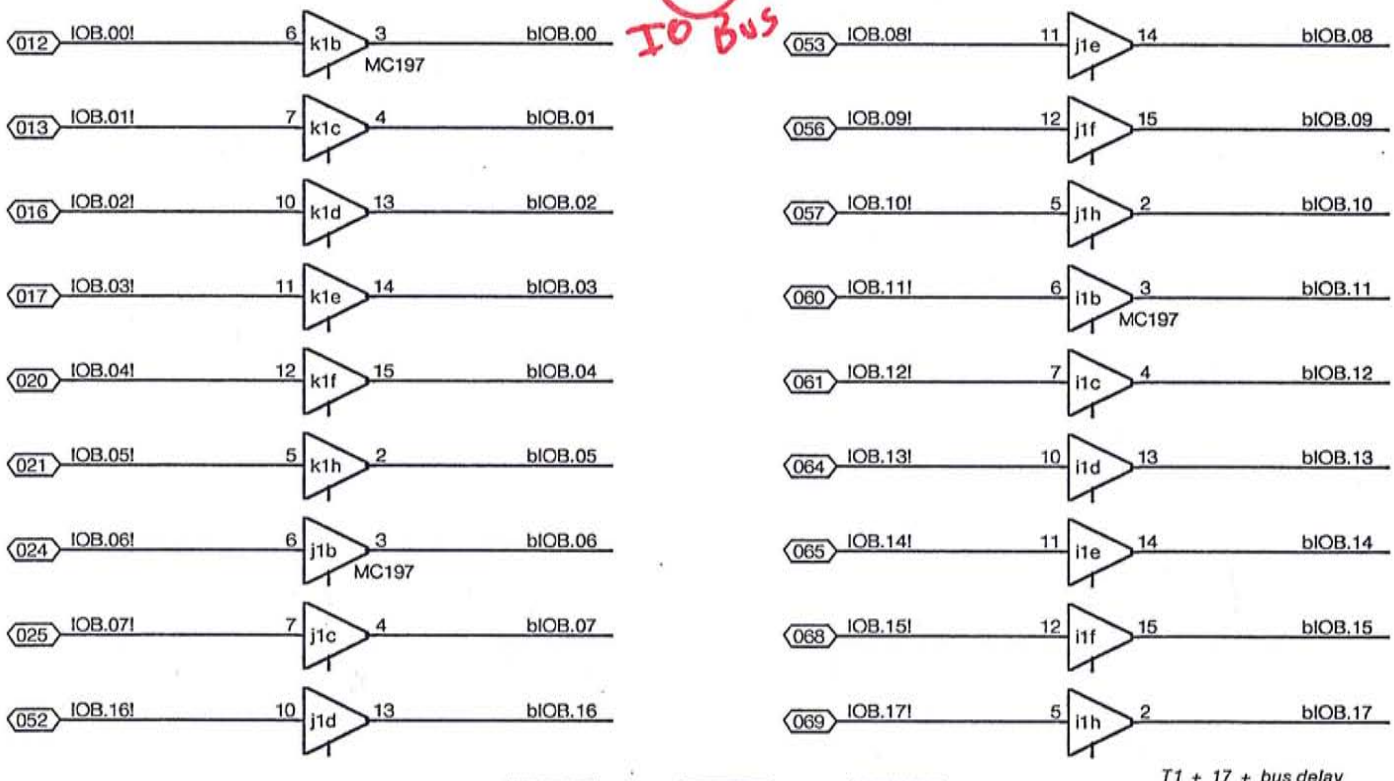
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	DMux Control	DskEth01.sil	Roger Bates	Ce	7/19/79	01



IO Address decoding

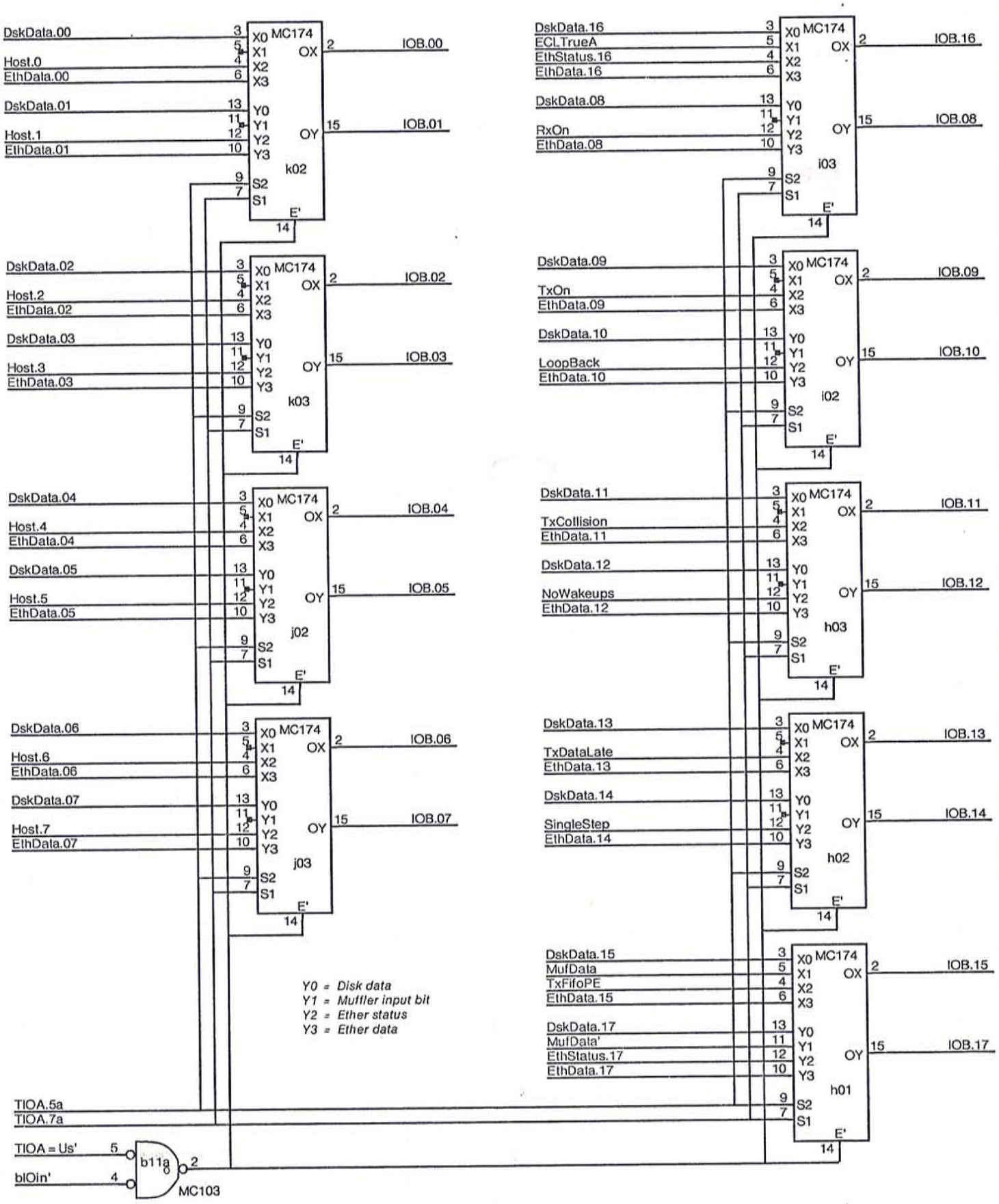
IOB receivers

IO Bus

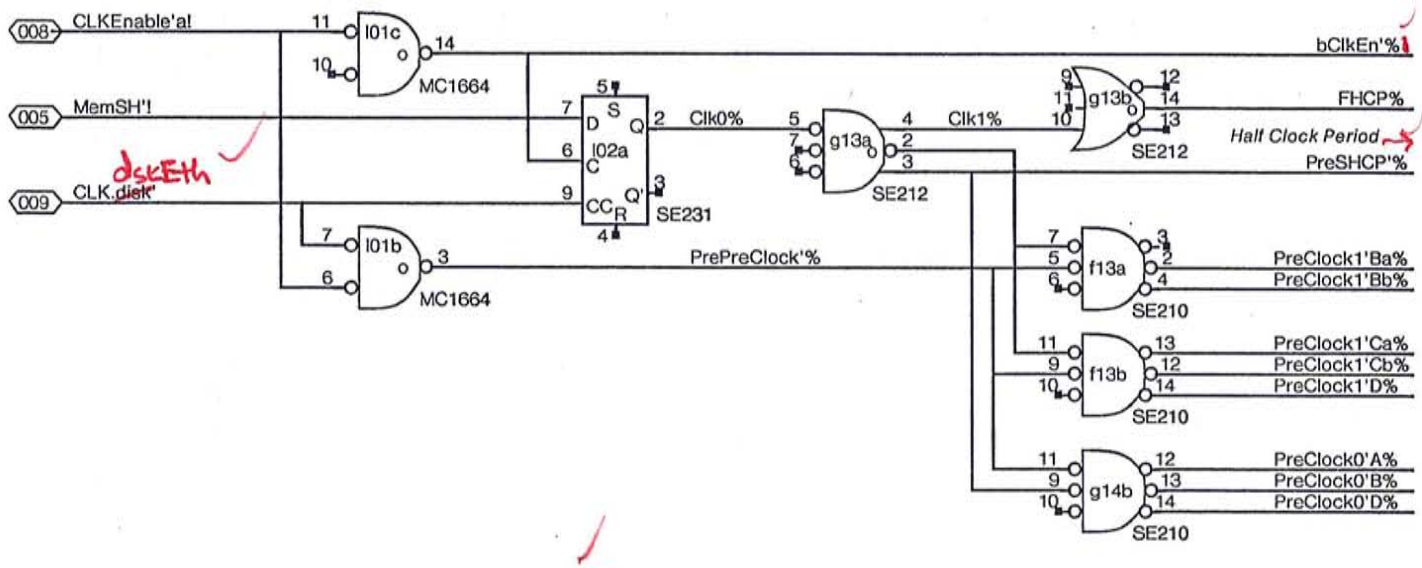


NOTE: IOB data received by this board is tested as part of the disk controller (Pg 18) Both the disk and the Ethernet check the data parity after it has gone through the respective FIFO's.

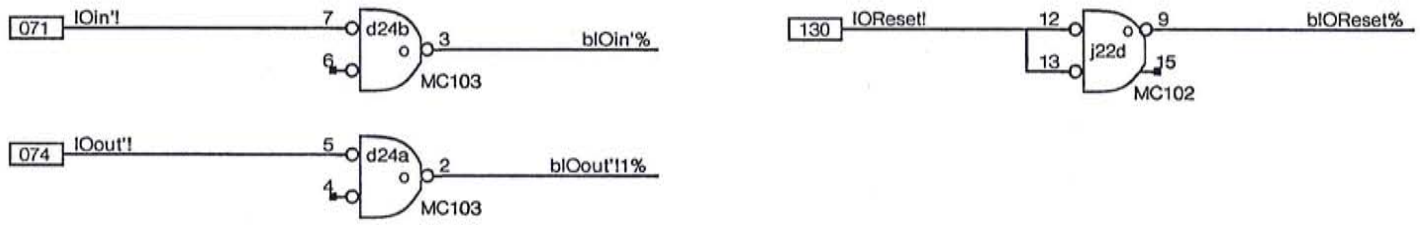
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	IOA & IOB	DskEth02.sil	Bates/Boggs	Ce	7/19/79	02



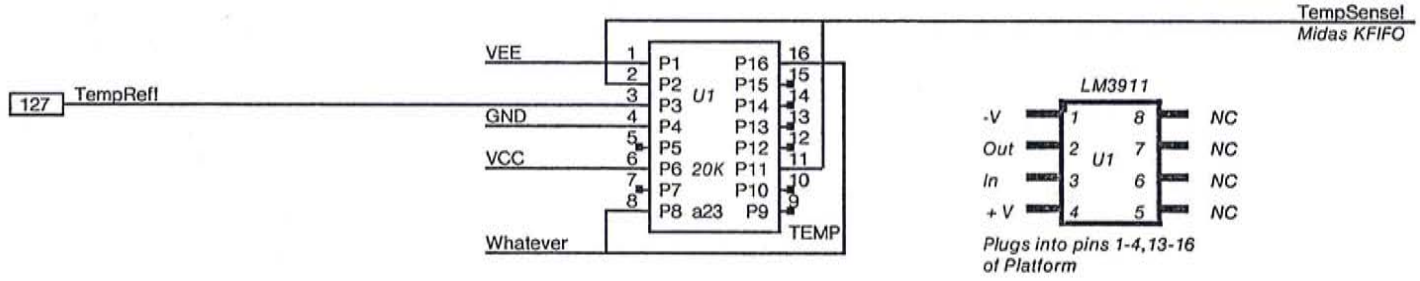
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	IOB ~~data Select~~/Drivers	DskEth03.sil	Bates/Boggs	Ce	9/24/79	03



Boad clocks



I/O control signals



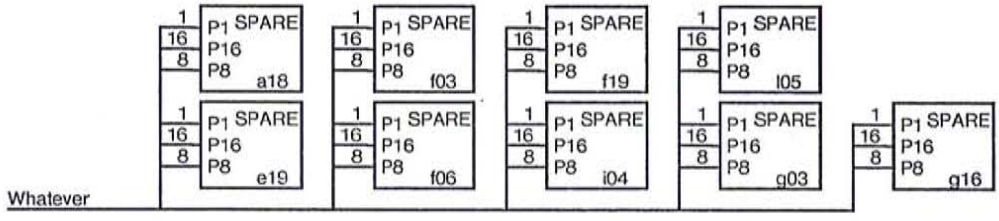
Temperature Sensor

Power connections are as follows:

StitchWeld: *1 & 16 are GND*
8 is -5

Multiwire: *16 is GND*
8 is -5

except in locations
a18, g03, and g16
which are uncommitted



Spare Socket locations for Multiwire

XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Clocks & IO Signals	DskEth04.sil	Bates/Boggs	Ce	7/23/79	04

-2v	Drive 3	Drive 2	IOA	Drive 1	Drive 0	IOB	Ethernet	IOB	Clk	+12v

	A (184)	B (168)	C (152)	D (136)	E (120)	F (102)	G (84)	H (68)	I (52)	J (36)	K (20)	L	
1	Sec Cnt LS169 X	8,13 MC102	Disk IO MC1650	Sec Cnt LS08	TIOA MC166 X	Disk IO MC1650	Dr Cont 8T98	IOB MC174 X	IOB MC197	IOB MC197	IOB MC197	Clock ,13 MC1664	1
41		SIP		SIP	SIP		SIP	SIP					41
2	Sec Cnt LS169 X	MC125	Disk IO MC1650	Disk IO N125	TIOA MC175	Disk IO MC1650	Disk IO N125	IOB MC174 X	MC174 X	MC174 X	IOB MC174 X	Clock SE231	2
3	Sec Cnt N123	ECC MC176	MC124	Sec Cnt LS153 X	TW MC109			IOB MC174 X	IOB MC174 X	IOB MC174 X	IOB MC174 X	5,6,12 MC195	3
42				SIP			SIP						42
4	Sec Cnt PLAT X	ECC MC176	ECC MC113	Dr Cont LS174 X	TW MC231	IOB MC170 X	IOB MC170 X	Rx TW MC231		Tx TW MC231	Rx Tw MC135	5,6 MC231	4
5	Sec Cnt N123	ECC MC176	ECC MC113	Dr Cont S288 X	TW MC231	Disk IO MC174 X	Host MC170 X	TW Com MC109	4,5 MC103	Rx CRC F9401 X	5,11 MC102		5
43													43
6	Sec Cnt LS169 X	ECC MC176	Clock ,8 SE212	Dr Cont LS155 X	TW Bit Cl MC231		Test MC175	4,8 MC231	Clock SE212	Clock SE210	RxCRC,11 MC125	6,4 MC135	6
7	Sec Cnt LS169 X	ECC MC158	ECC MC176	MC102	Mul MU164 X	TIOA MC161 X	Test MC158	1,4,10 MC124	Clock SE210	Clock SE210	RxSR Dmp MC135	Rx EOP MC231	7
44													44
8	Sec Cnt LS174 X	ECC,9 MC158	ECC MC176	FIFO F145A X	Out Reg MC176 X	Sh Reg MC176 X	In Reg MC173 X	Rx FSM MC176	Rx SR MC141 X	Rx Fifo F145A X	Rx Par MC170 X	4,12 MC105	8
9	Sec Cnt LS174 X	Out Reg MC231	MC231	FIFO F145A X	Out Reg MC170 X	Sh Reg F00 X	In Reg MC173 X	Rx FSM MCM149 X	Rx SR MC141 X	Rx Fifo F145A X	Rx Reg MC176 X	Rx Write F16 X	9
45													45
10	Sec Cnt LS174 X	Bit Cl F16 X	MC104	FIFO F145A X	Out Reg MC176 X	Sh Reg F00 X	In Reg MC173 X	Rx FSM MCM149 X	Rx SR MC141 X	Rx Fifo F145A X	Rx Reg MC176 X	Rx Fifo MCM149 X	10
11	Sec Cnt LS174 X	MC103	MC103	FIFO F145A X	Out Reg MC170 X	Sh Reg F00 X	In Reg MC173 X	Rx FSM MCM149 X	Rx SR MC141 X	Rx Fifo F145A X	Rx Reg MC176 X	Rx Fifo MC158 X	11
46													46
12	Mul MU164 X	MC231	MC195	FIFO F145A X	Out Reg MC176 X	Sh Reg F00 X	In Reg MC173 X	Rx FSM MC176	Rx FSM MC136 X	Rx Fifo F145A X	Rx Par MC170 X	Rx Read F16 X	12
13	Mul MU164 X	Mul MU164 X	Errors MC197	MC102	Cont MC135	Clock SE210	Clock SE212	Tx FSM MC176	Tx FSM MC136 X	Tx Fifo F145A X	Tx Par MC170 X	Tx Write F16 X	13
47													47
14	Fifo Cnt MC158 X	Fifo Cnt MCM149 X	Fifo Cnt ,4 MC176	Cont MC171 X	Cont MC231	Cont F00 X	Clock SE210	Tx FSM MCM149 X	Tx SR MC141 X	Tx Fifo F145A X	Tx Reg MC176 X	Tx Fifo MC158 X	14
15	Fifo Cnt F16 X	Fifo Cnt MC231	Fifo Cnt MC231	Bit Cl SE210	Cont MC231	Cont F00 X	Tx FSM MC102	Tx FSM MCM149 X	Tx SR MC141 X	Tx Fifo F145A X	Tx Reg MC176 X	Tx Fifo MCM149 X	15
48													48
16	Fifo Cnt F16 X	Clock SE212	Cont MC135	Fifo Cnt ,2 MC102	Ram MC145A X	Ram MC145A X		Tx FSM MCM149 X	Tx SR MC141 X	Tx Fifo F145A X	Tx Reg MC176 X	Tx Read F16 X	16
17	Mul MU164 X	Clock SE211	Clock SE210	TAG MC125	Mul MC231	Ram MC145A X	Tx CRC F9401 X	Tx FSM MC176	Tx SR MC141 X	Tx Fifo F145A X	Tx Par MC170 X	8,10 MC105	17
49													49
18		Clock SE210	Clock SE212	Mul MC173 X	Mul MC173 X	Mul MC173 X	ETX MU164 X	Tx CRC MC125	Clock SE210	Clock SE210	9,10 MC104	FIFOs MC176	18
19	3, 5 MC100	MC195	CNT MC136 X	Disk Mul MC173 X			ERX MU164 X	ETX MU164 X	EFIFO MU164 X	Pend MC135	Coll MC135	GotBit MC135	19
50													50
20	Ram MC139 X	Ram F16	CNT MC136 X	TAG MC135	TAG MC173 X	TAG MC173 X	Rx PD MC176 X	ERX MU164 X	8,12 MC124 X	Ether Clk K1115A X	TxGo MC231	Tx PE MC231	20
21	Ram MC139 X	Ram F16 X	CNT MC136 X	TAG MCM149 X	TAG MC173 X	TAG MC173 X	Rx PD F16 X	9, PE MC106	Ether Clk MC210	Ether Clk MC136 X	Status MC170 X	EFIFO MU164 X	21
51													51
22	Ram F16 X	Mul MU164 X	Mul MU164 X	TAG F16 X	TAG MC125	TAG MC125	TAG 8T98	Rx PD MCM149 X	10 MC231	7,10,11,13 MC102	7, 8 MC135	Mul MC176 X	22
23	Temp LM3911 X	Dr Stat MC109	Dr Stat MC103	Mul MU164 X	TAG MC125	TAG MC125	TAG 8T98	Next,8 MC105	Next,7,12 MC104	Next MC113	Mul MU164 X	Mul MC176 X	23
52				SIP	SIP					SIP	SIP		52
24	Next MC195	Dr Stat MC124	Dr Stat MC124	Clock MC103	Mul MU164 X	Mul MU164 X	TAG 8T98	Next MC176	Next MC104	5,6,13 MC176	Mul MC166 X	Mul MC102	24

A (10)	B (26)	C (42)	D (58)	E (74)	F (92)	G (110)	H (126)	I (142)	J (158)	K (174)	L

-5v	Next	Disk daisy chain cable	IOAtt	Muffler	+5v

31	chips common to Disk & Ether
137	chips specific to Disk
111	chips specific to Ether
9	spare chip positions
288	total chip positions

XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Stitch-Weld board Layout	DskEth05.sil	Bates/Boggs	Ce	9/24/79	05

Cut SIP legs at k52 to set the
Muffler addresses for the board.
* Standard addresses are 2000-2177.

	Muff Addr	P5	P6	P7	P8
	0000-0177	cut	cut	cut	cut
	0200-0377	cut	cut	cut	
	0400-0577	cut	cut		cut
	0600-0777	cut	cut		
	1000-1177	cut		cut	cut
	1200-1377	cut		cut	
	1400-1577	cut			cut
	1600-1777	cut			
*	2000-2177		cut	cut	cut
	2200-2377		cut	cut	
	2400-2577		cut		cut
	2600-2777		cut		
	3000-3177			cut	cut
	3200-3377			cut	
	3400-3577				cut
	3600-3777				

Cut SIP legs at j52 to set the
Task numbers for the Ethernet.
* Standard tasks are 6 & 7.

	Tasks	P6	P7	P8
	2 & 3			cut
	4 & 5		cut	
*	6 & 7		cut	cut
	8 & 9	cut		
	10 & 11	cut		cut
	12 & 13	cut	cut	
	14 & 15	cut	cut	cut

Cut SIP legs at e41 to set the
IOA bus addresses for the board.
* Standard addresses are 10-17.

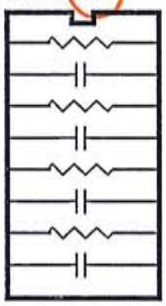
	IOA	P4	P5	P6	P7	P8
	000-007	cut	cut	cut	cut	cut
*	010-017	cut	cut	cut	cut	
	020-027	cut	cut	cut		cut
	030-037	cut	cut	cut		
	040-047	cut	cut		cut	cut
	050-057	cut	cut		cut	
	060-067	cut	cut			cut
	070-077	cut	cut			
	100-107	cut		cut	cut	cut
	110-117	cut		cut	cut	
	120-127	cut		cut		cut
	130-137	cut		cut		
	140-147	cut			cut	cut
	150-157	cut			cut	
	160-167	cut				cut
	170-177	cut				
	200-207		cut	cut	cut	cut
	210-217		cut	cut	cut	
	220-227		cut	cut		cut
	230-237		cut	cut	cut	
	240-247		cut		cut	cut
	250-257		cut			
	260-267		cut			cut
	270-277		cut			
	300-307			cut	cut	cut
	310-317			cut	cut	
	320-327			cut		cut
	330-337			cut		
	340-347				cut	cut
	350-357				cut	
	360-367					cut
	370-377					

The EtherFifo ROMs are identical and interchangeable.
The DiskRead, EtherRcvr and EtherXmtr ROMs are not.

Name	Type	#	Location		
DskEth					
Disk					
DiskRead	SG139	2	a20	a21	
DiskTag	MC149	1	d21		
DiskUnits	S288	1	d05		
DiskFifo	MC149	1	b14		
Ether					
EtherPD	MC149	1	h22		
EtherRcvr	MC149	3	h09	h10	h11
EtherFifo	MC149	2	l10	l15	
EtherXmtr	MC149	3	h14	h15	h16

"DoradoProms DskEth" gets all proms for the DskEth board.
"DoradoProms Disk" gets all proms for the Trident half of the board.
"DoradoProms Ether" gets all proms for the Ethernet half of the board.

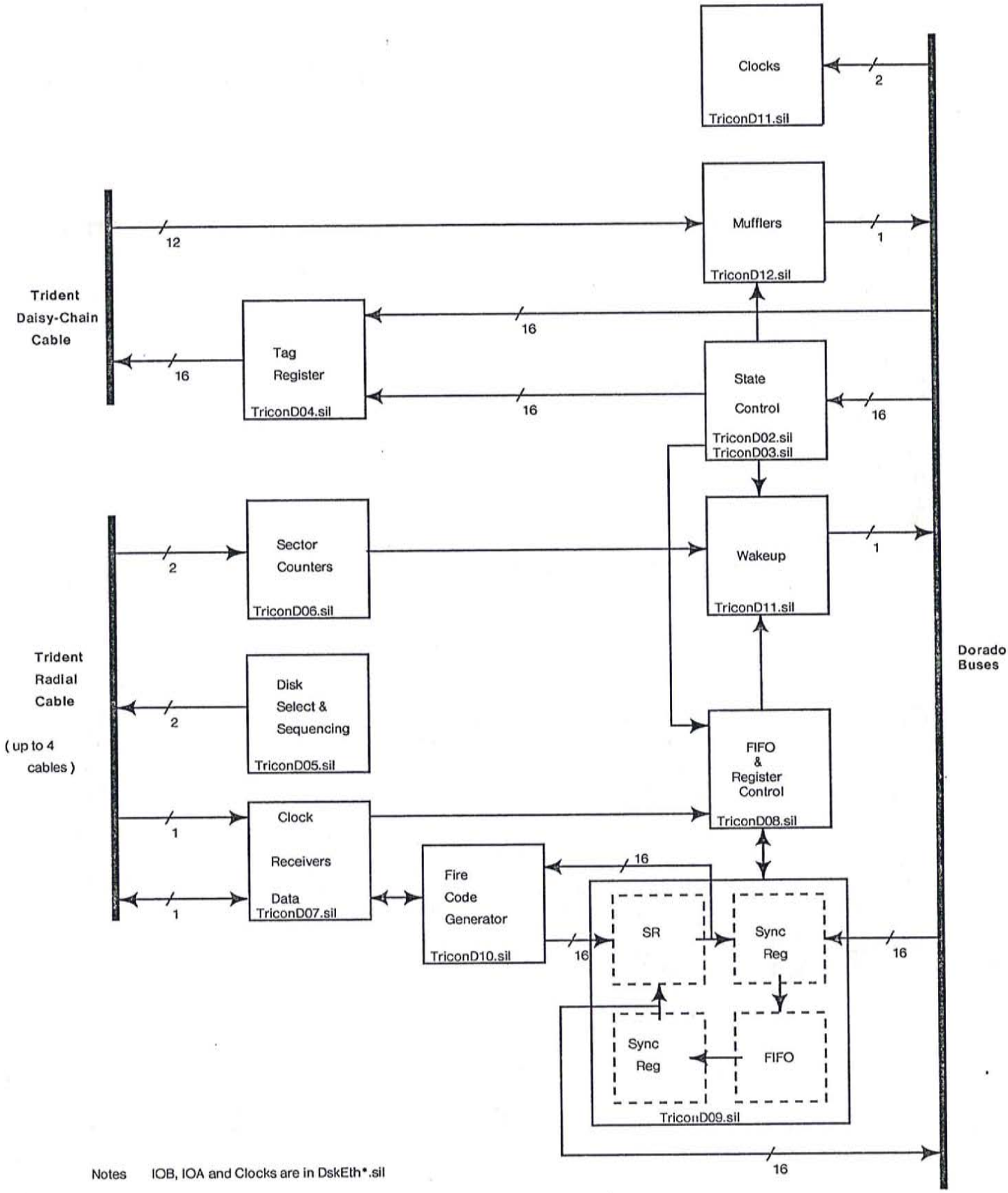
PLAT at a04



All resistors 27K
All capacitors 330pf

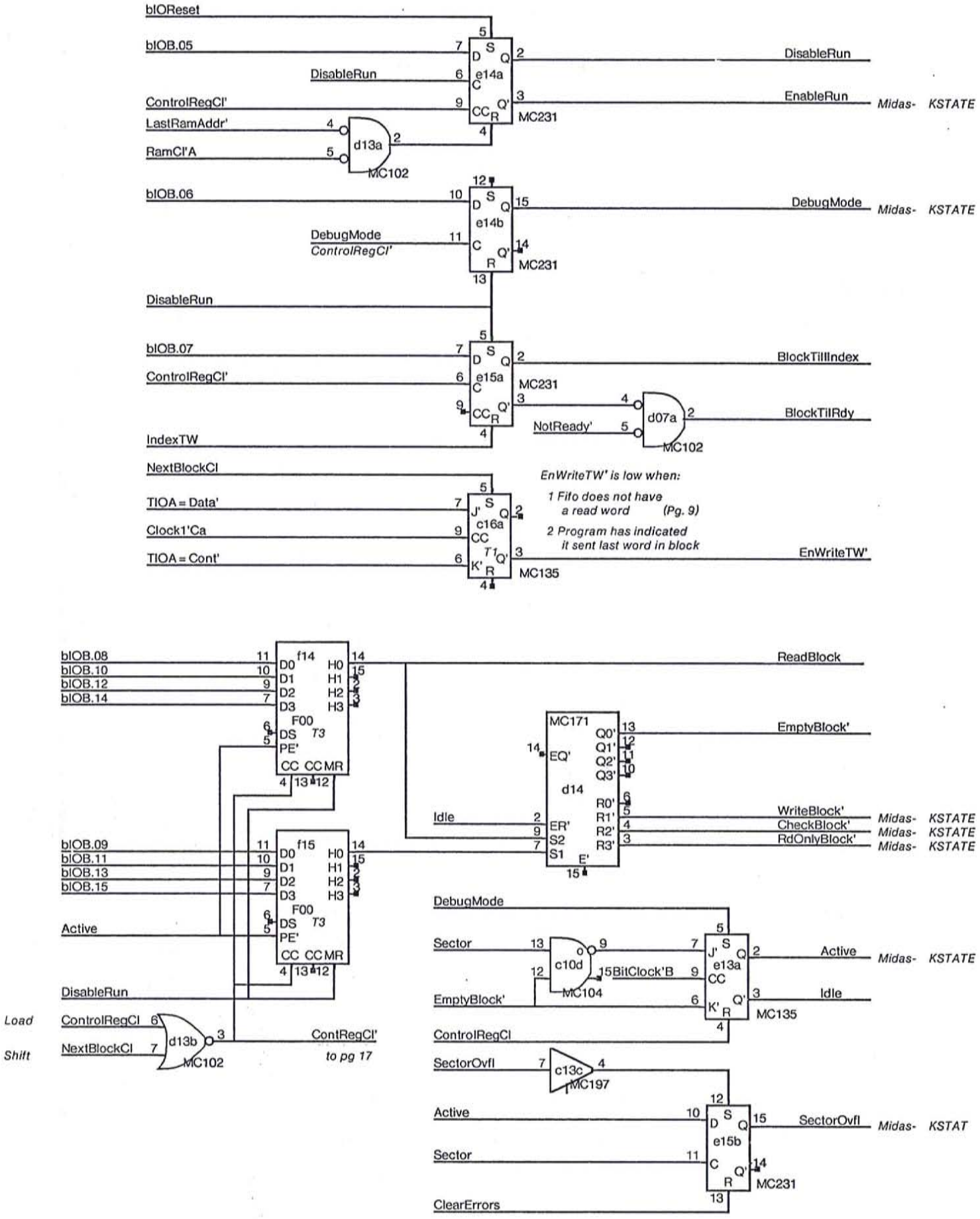
K1115A at j20

Logo is near pin 1.
This is a 14 pin pkg.
Board pins 8 & 9 aren't used.
Remove bypass cap above pin 1
on sw.

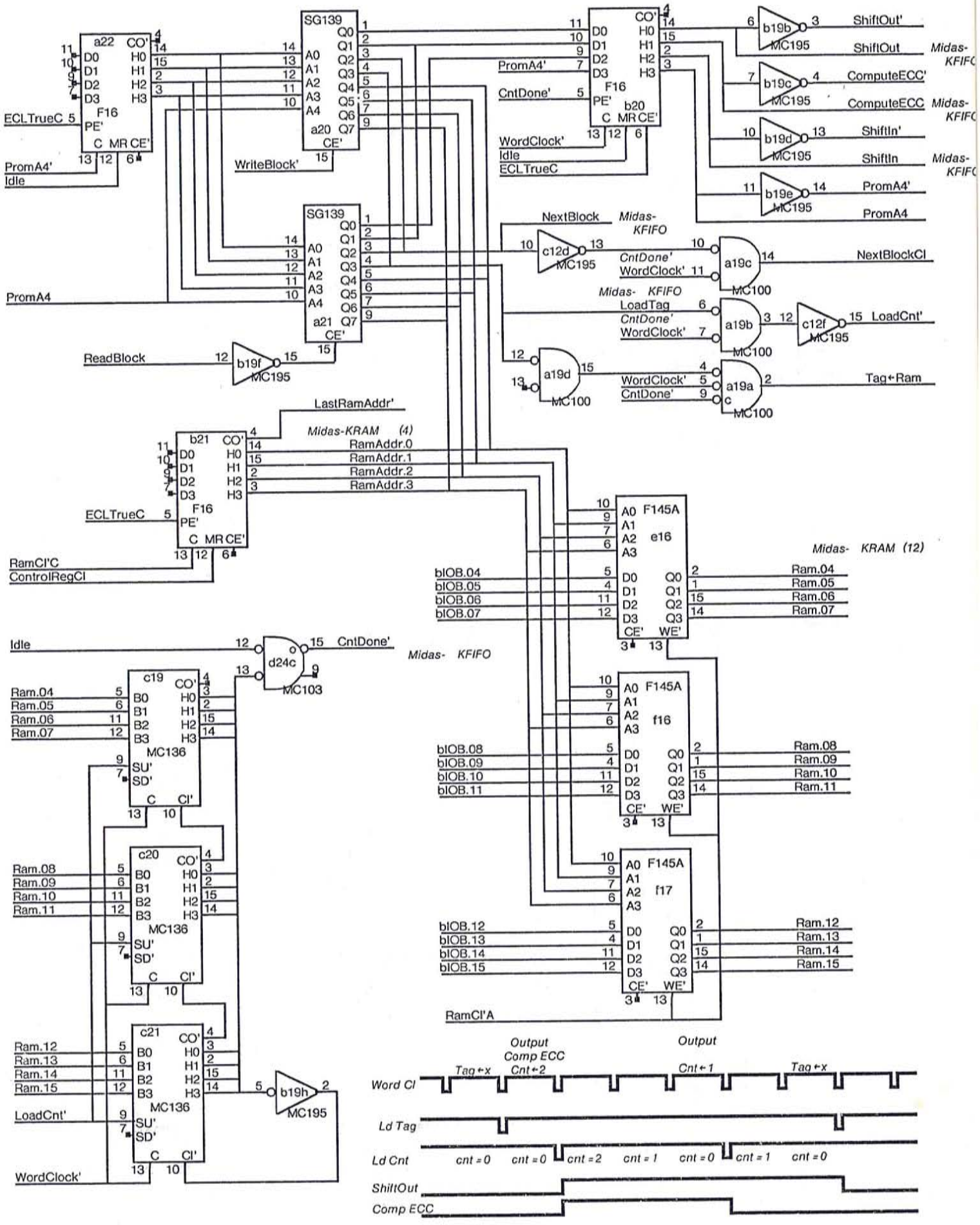


Notes IOB, IOA and Clocks are in DskEth*.sil

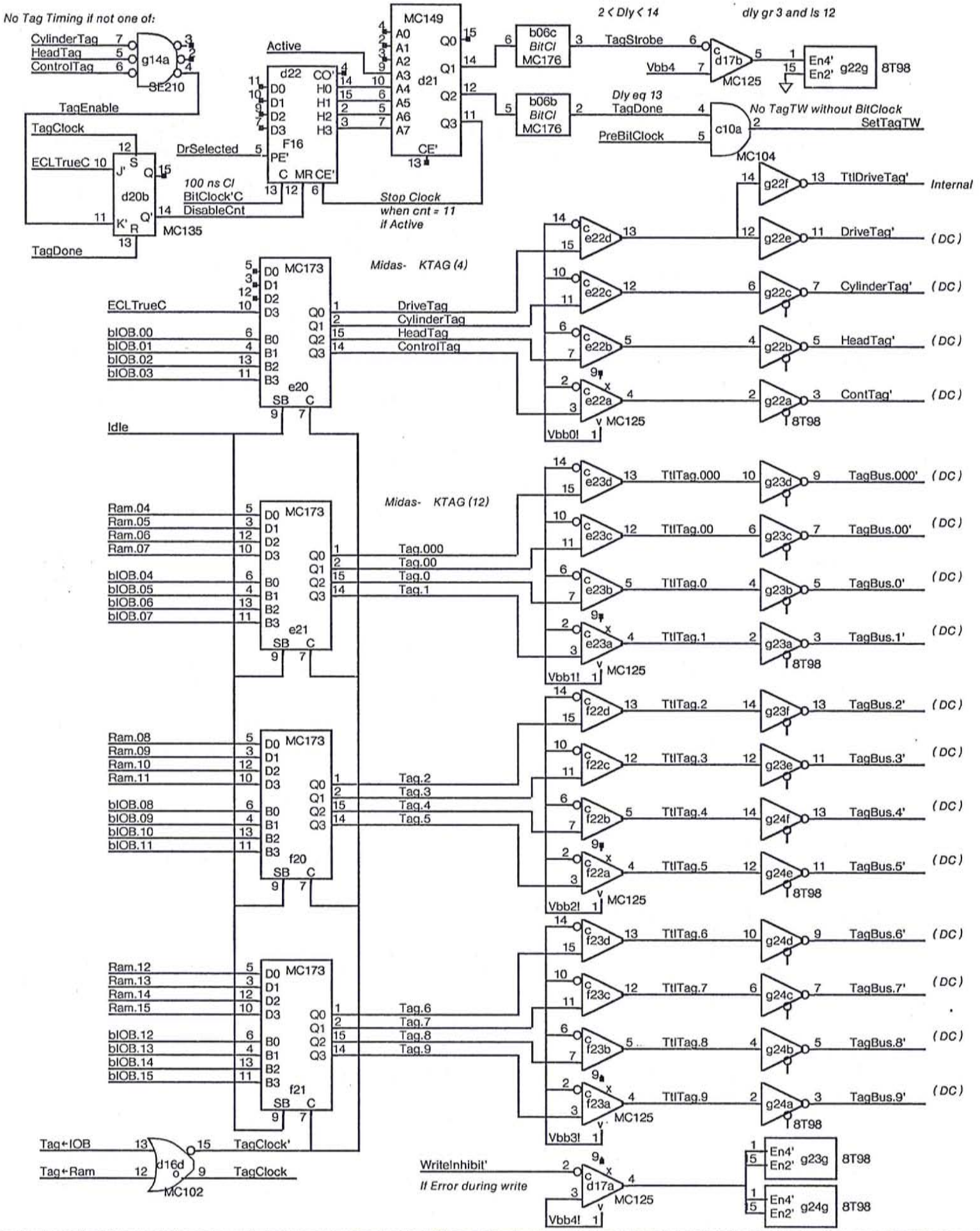
XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Dorado Disc Controller	TriconD01.sil	Roger Bates	Ce	7/19/79	07



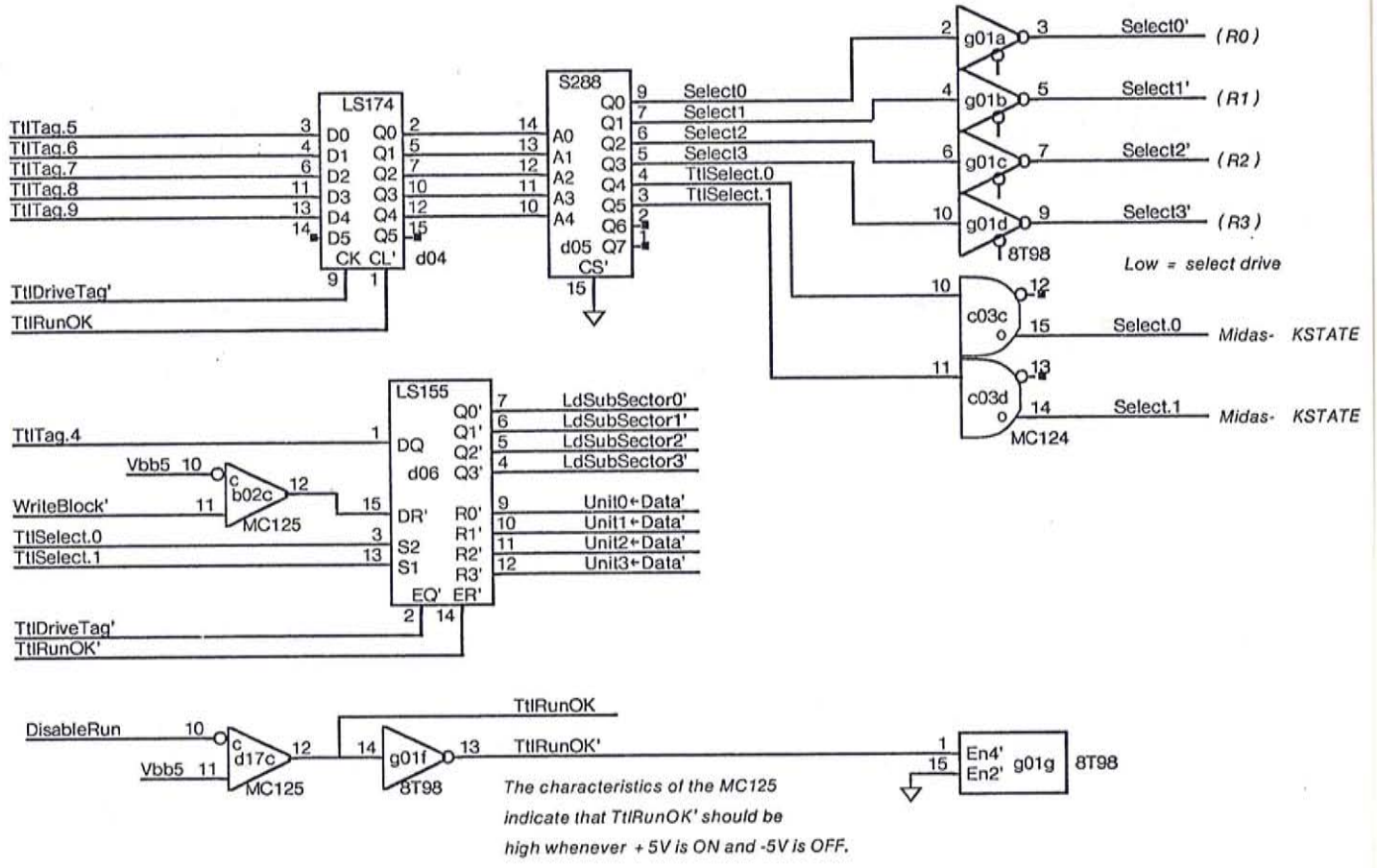
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	State Control Register	TriconD02.sil	Roger Bates	Ce	9/24/79	08

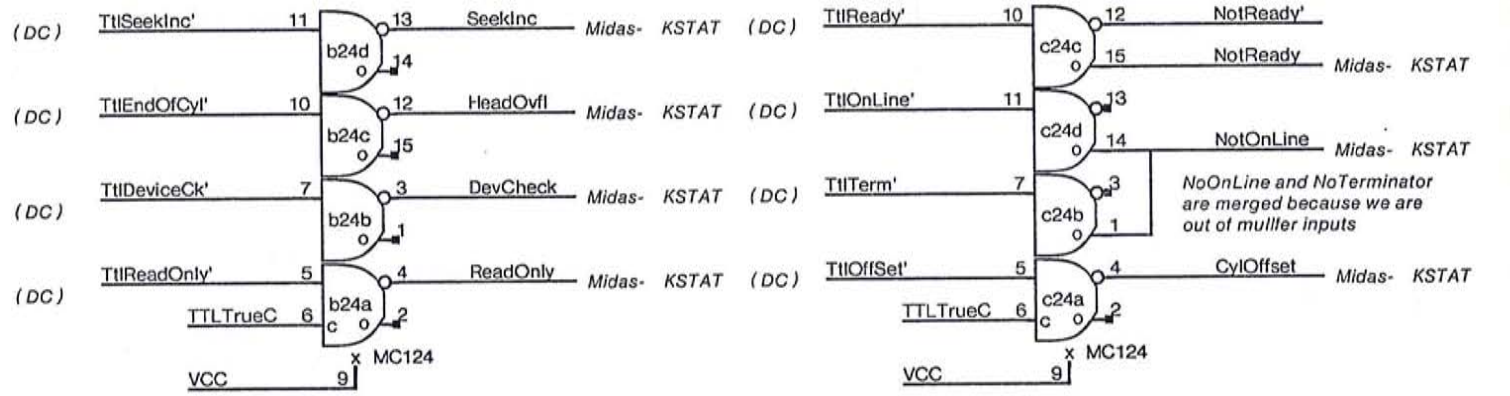


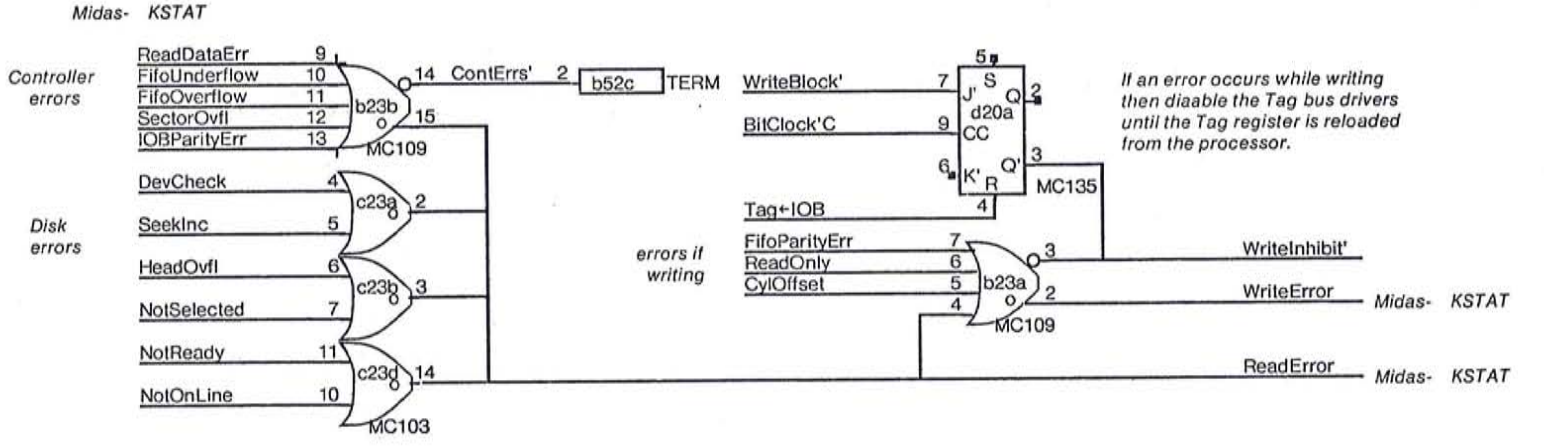
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Control RAM and Format Counter	TriconD03.sil	Roger Bates	Ce	9/24/79	09



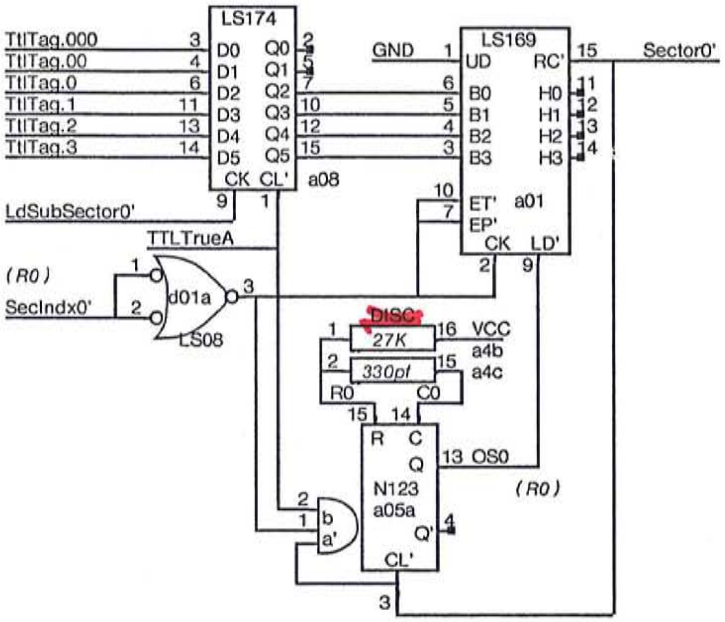
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	TAG register & Bus Drivers	TriconD04.sil	Roger Bates	Ce	7/23/79	10

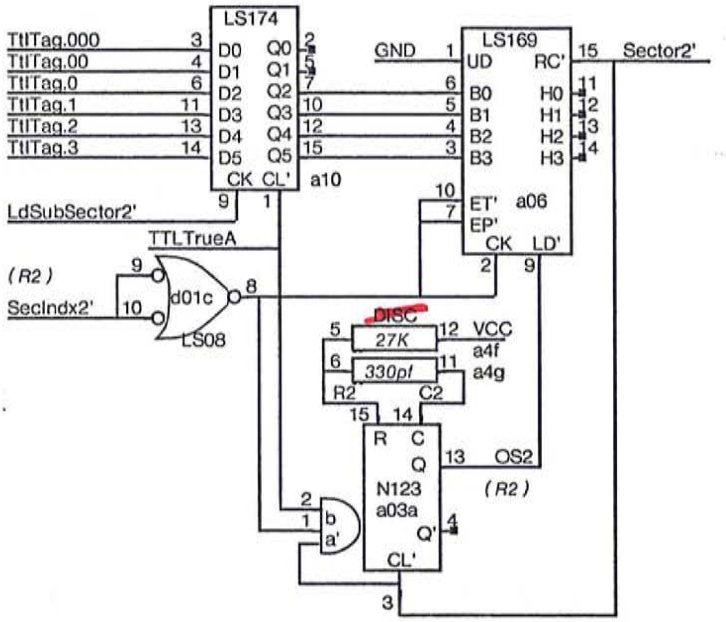


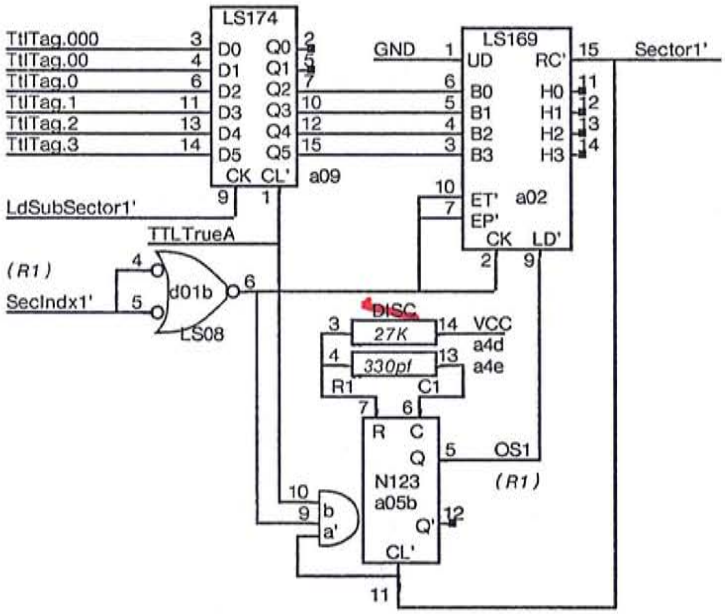


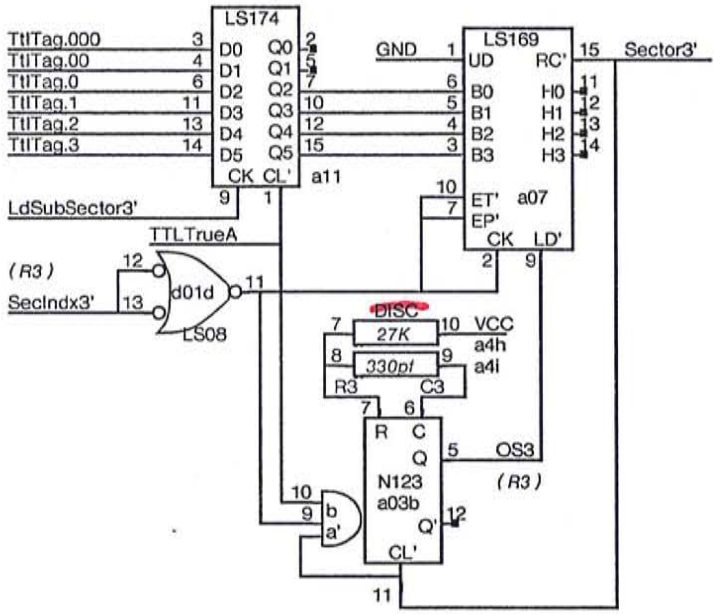


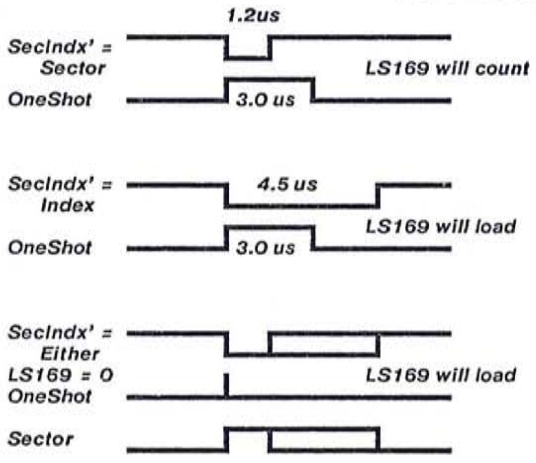
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Disk Drive Control	TriconD05.sil	Roger Bates	Ce	7/23/79	11

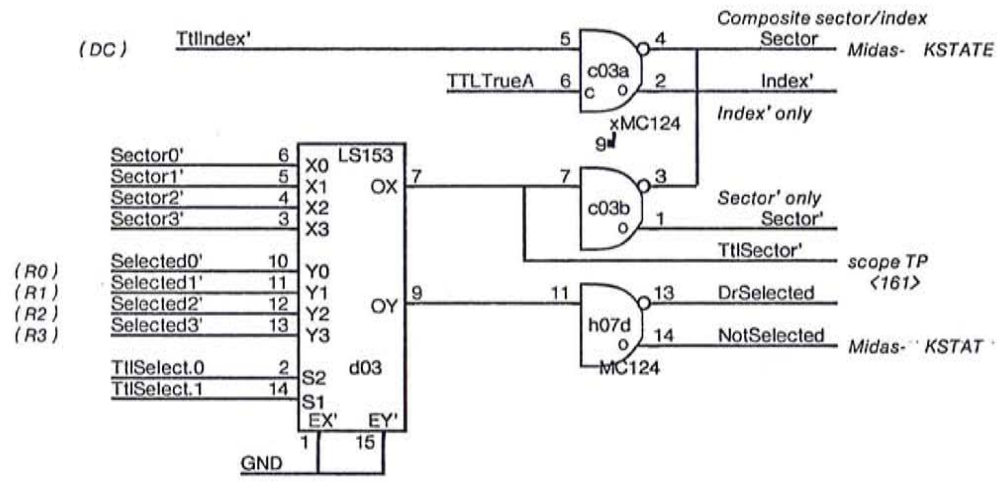








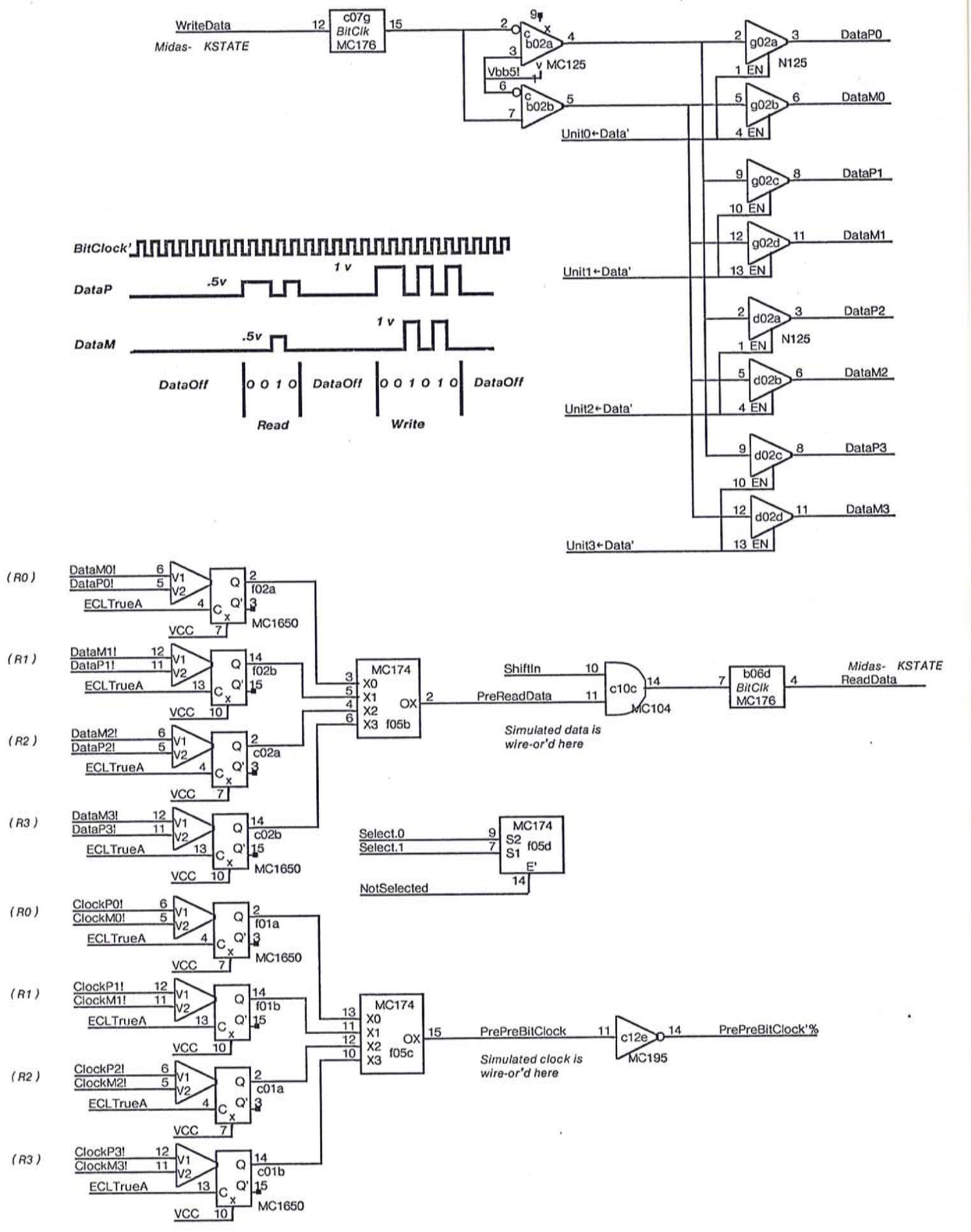




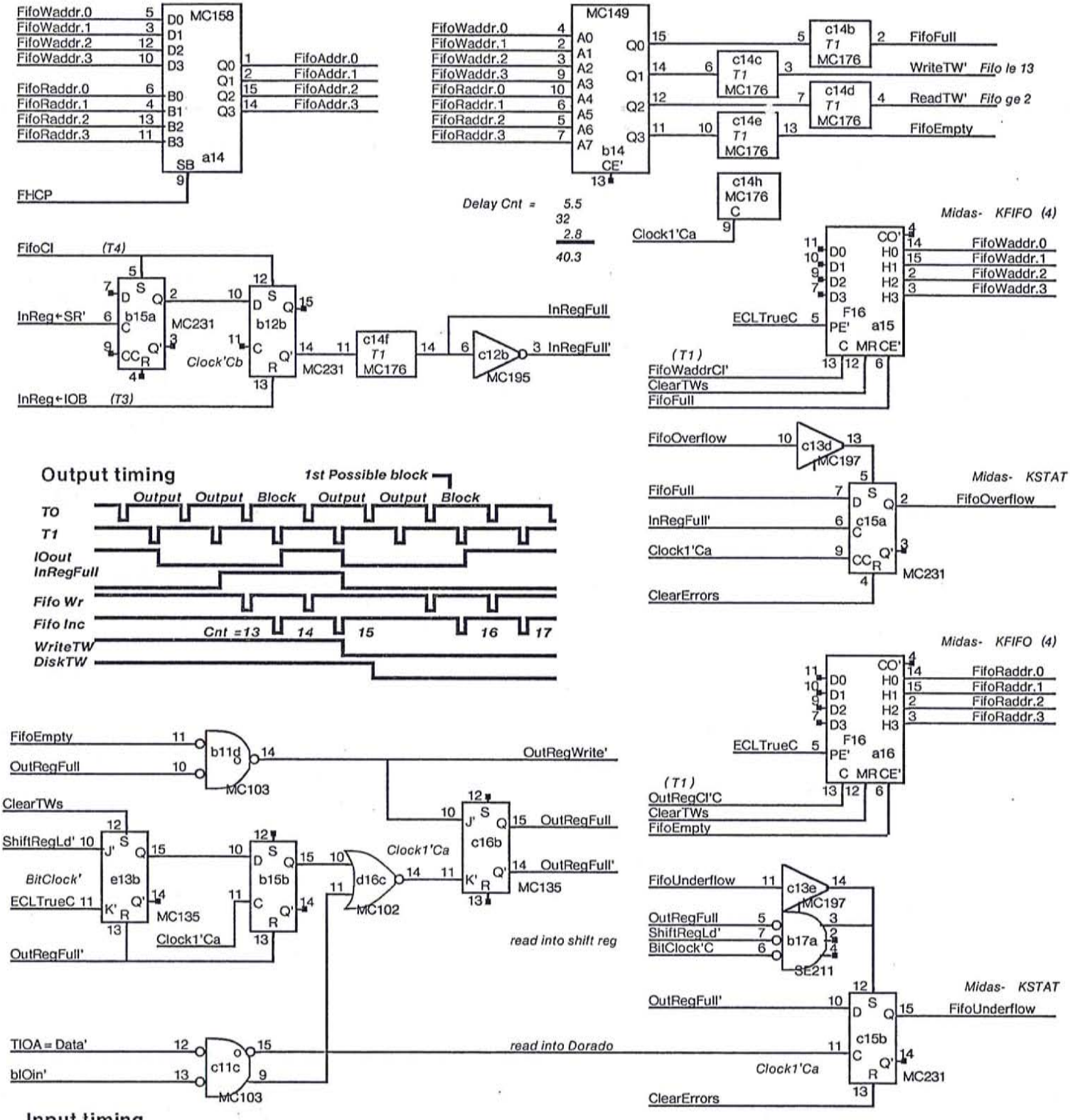
The above sector counters require the TRIDENT disk to have its sector counters set to provide 117 "sub-sector" pulses per revolution. This is done by setting the disk jumpers as follows:

	X6A	X6B
	4 - 11	2 - 13
	5 - 10	3 - 12
		4 - 11
		5 - 10
		6 - 09

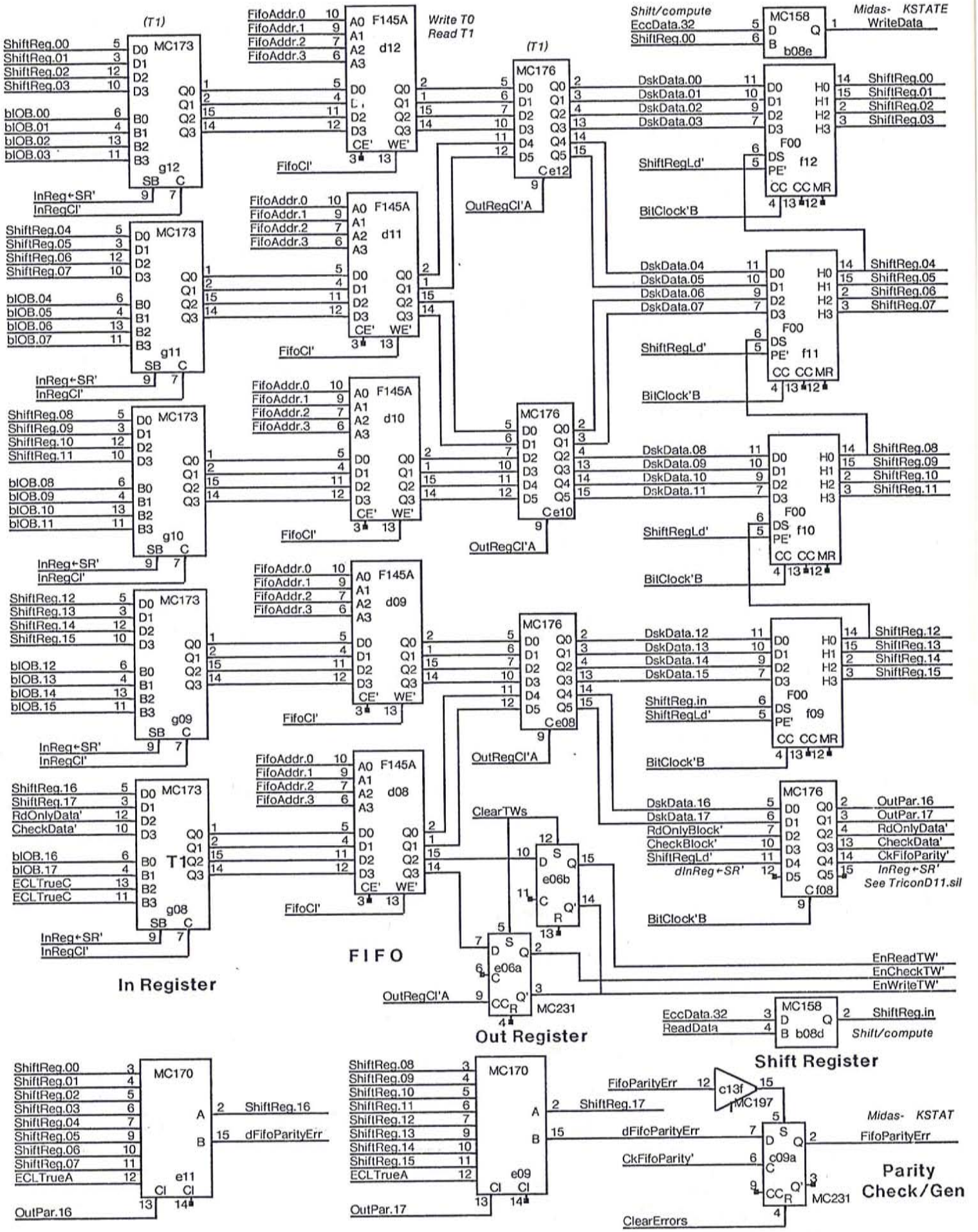
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Sector Length Counters	TriconD06.sil	Roger Bates	Ce	7/24/79	12



XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Clock and Data Input	TriconD07.sil	Roger Bates	Ce	7/23/79	13

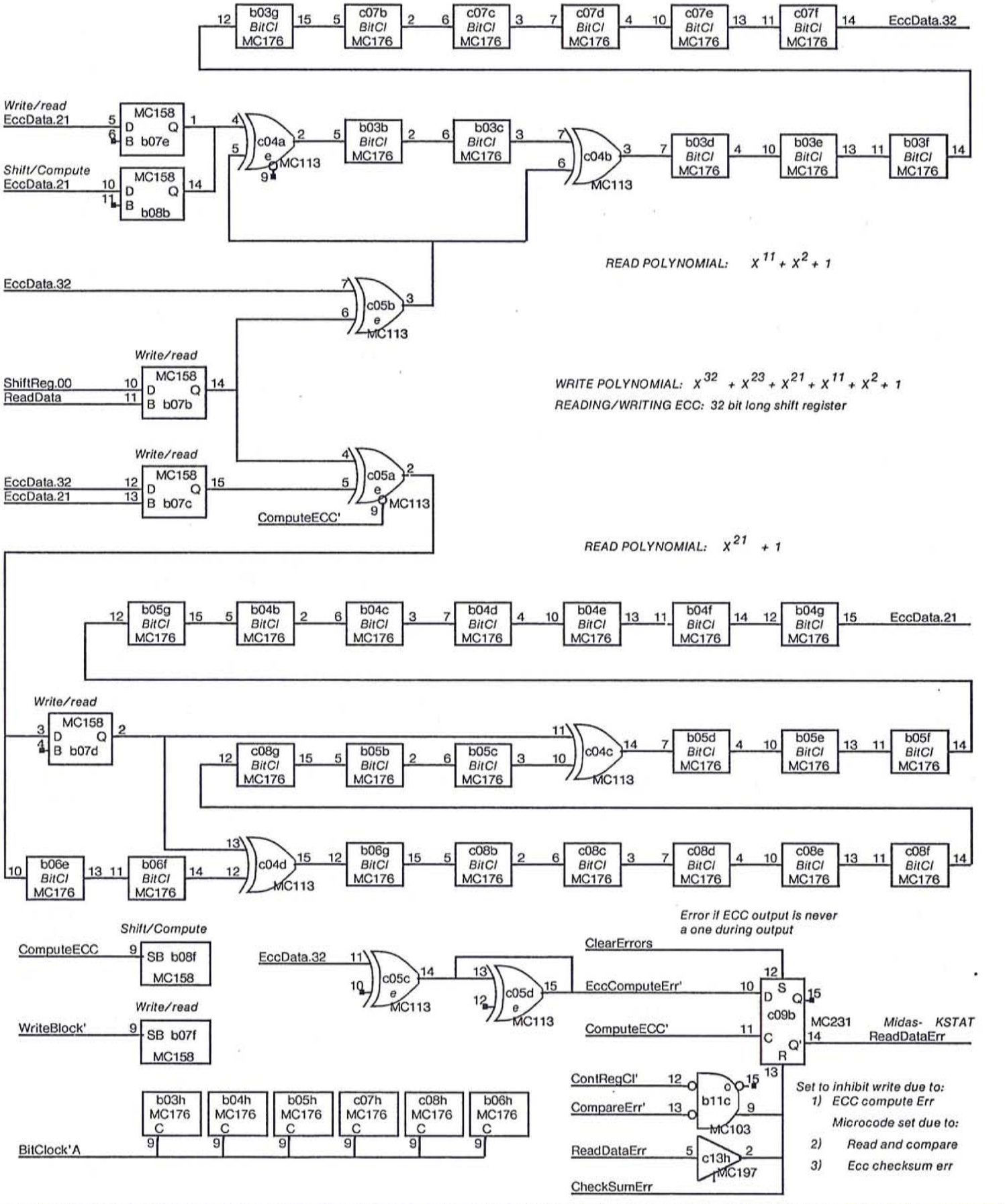


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	FIFO Control	TriconD08.sil	Roger Bates	Ce	9/24/79	14

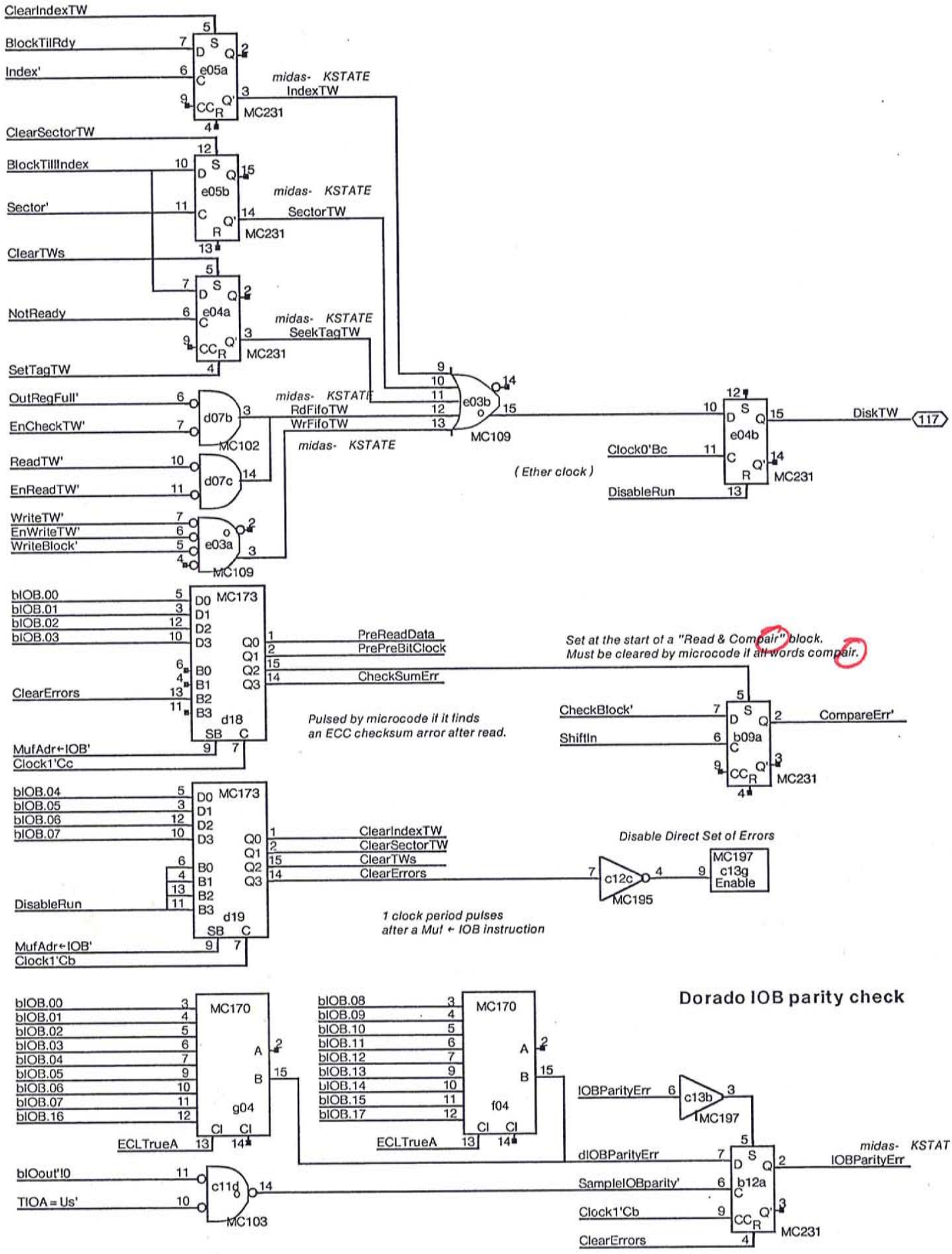


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	FIFO and Shift Register	TriconD09.sil	Roger Bates	Ce	7/08/79	15

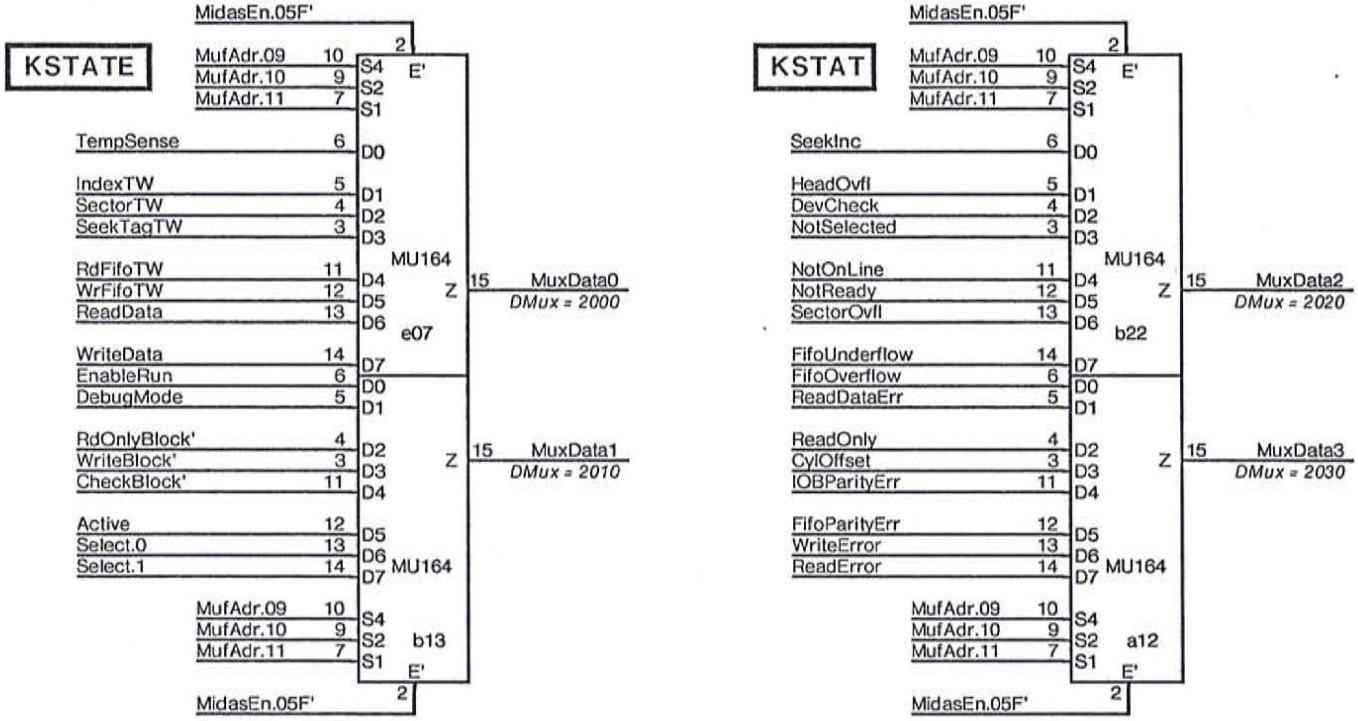
POLYNOMIAL DIVIDER FOR FIRE CODE GENERATION



XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Fire Code Generator	TriconD10.sil	Roger Bates	Ce	7/08/79	16



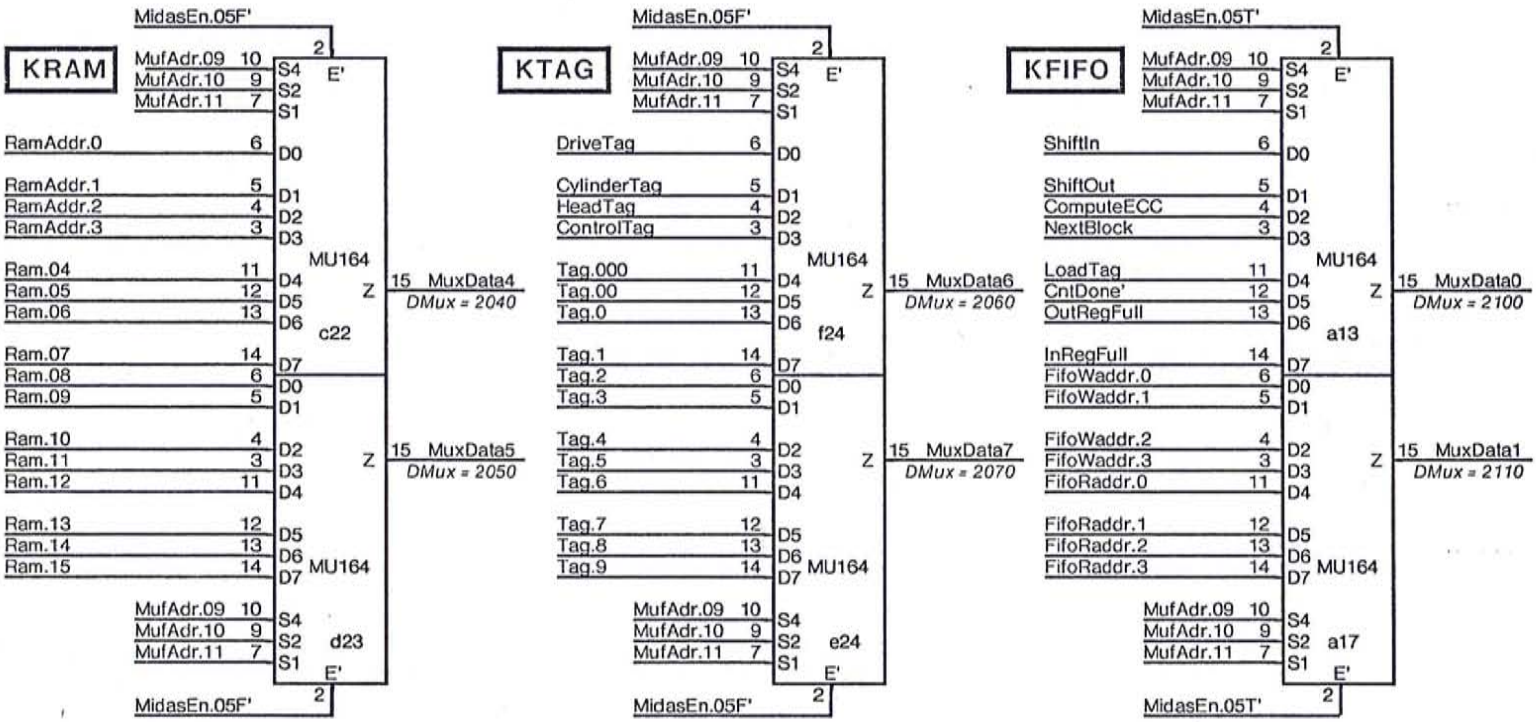
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Buss Parity Task Wake-Up	TriconD11.sil	Roger Bates	Ce	7/08/79	17



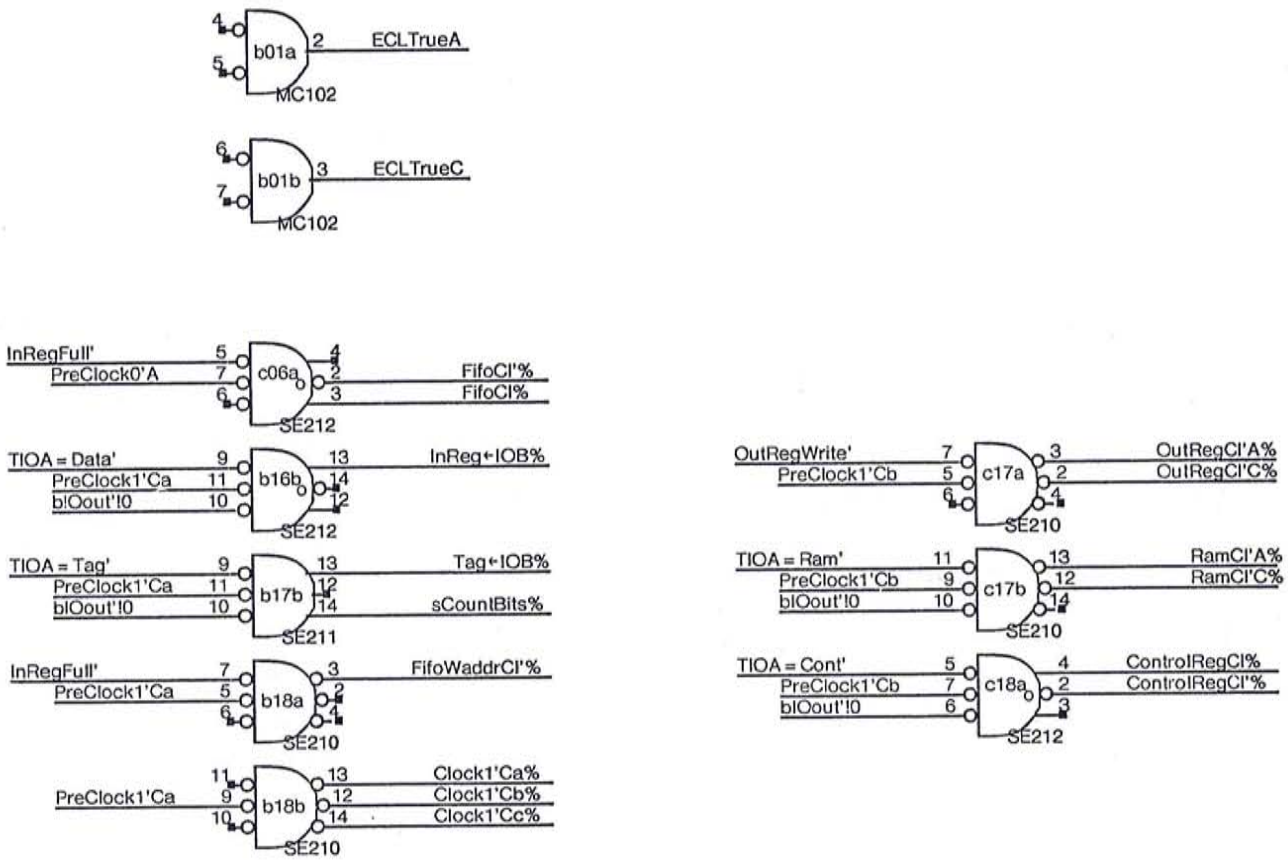
Muffler addresses are:

Value listed for Program input
Value plus 2000 for Midas input

Values from 120 to 177 are used by the Ethernet

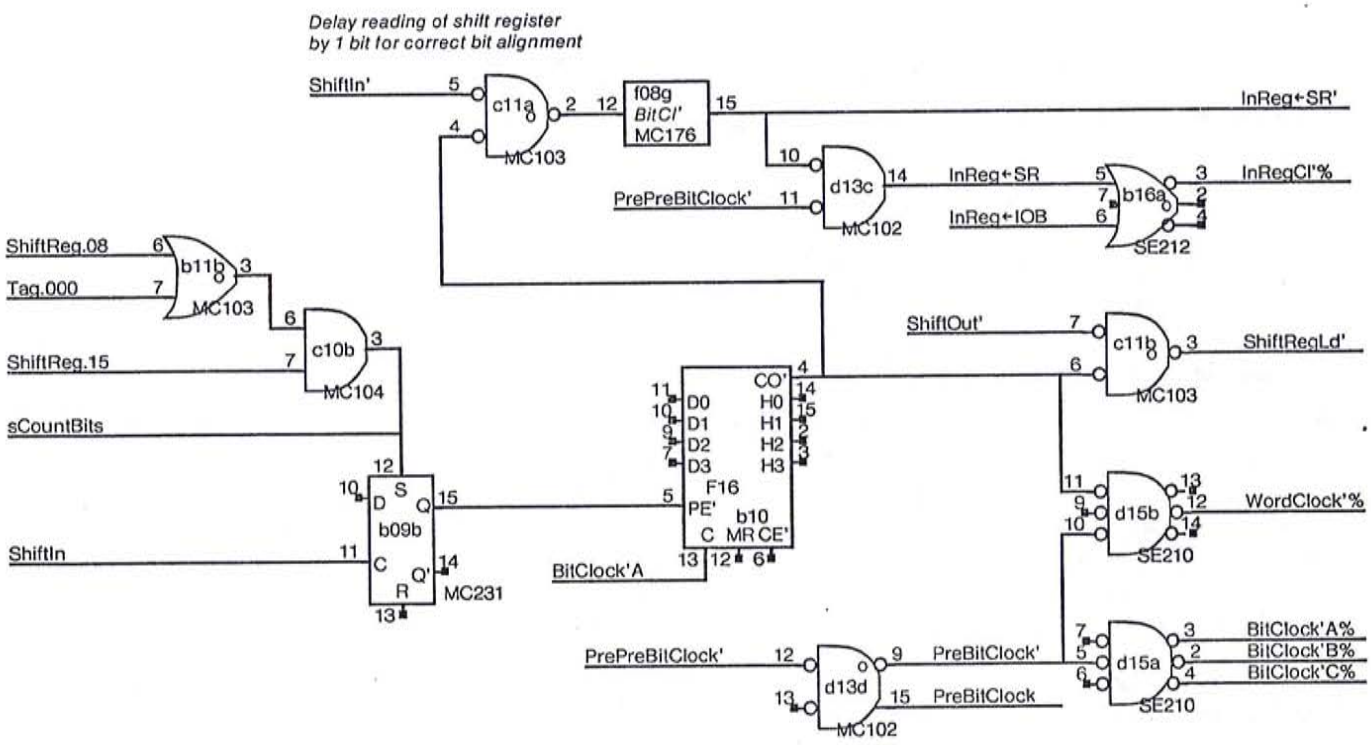


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	DMUX CONTROL	TriconD12.sil	Roger Bates	Ce	7/08/79	18



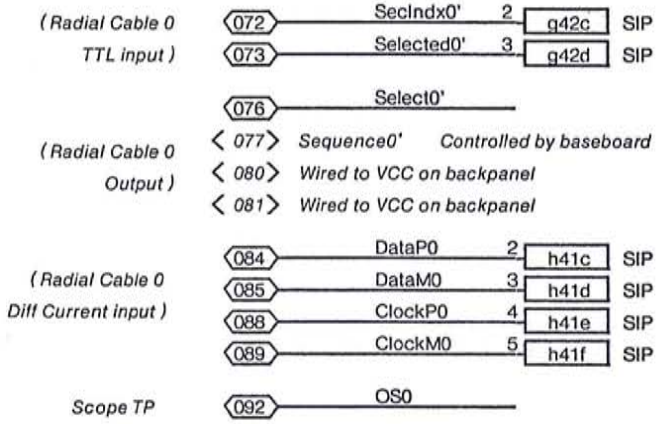
System Clocks

Disk Clocks

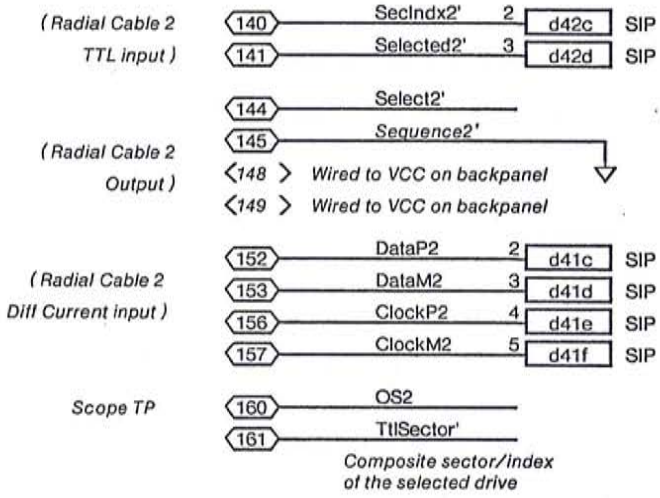


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Dorado Clock Distribution	TriconD13.sil	Roger Bates	Ce	7/23/79	19

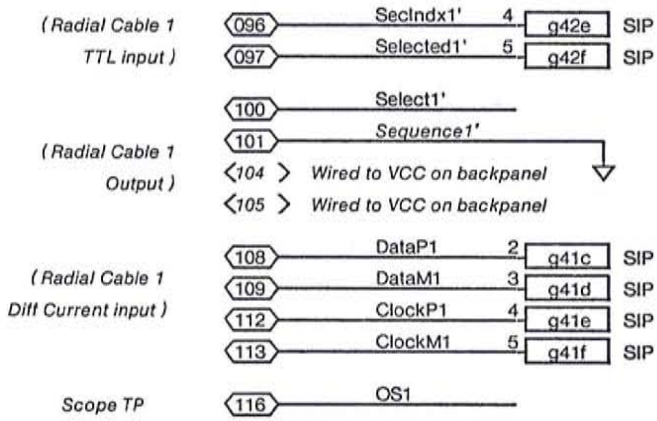
Radial Cable for Drive 0



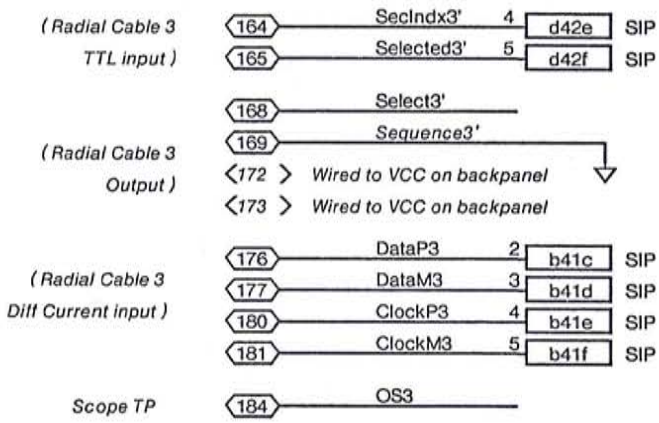
Radial Cable for Drive 2



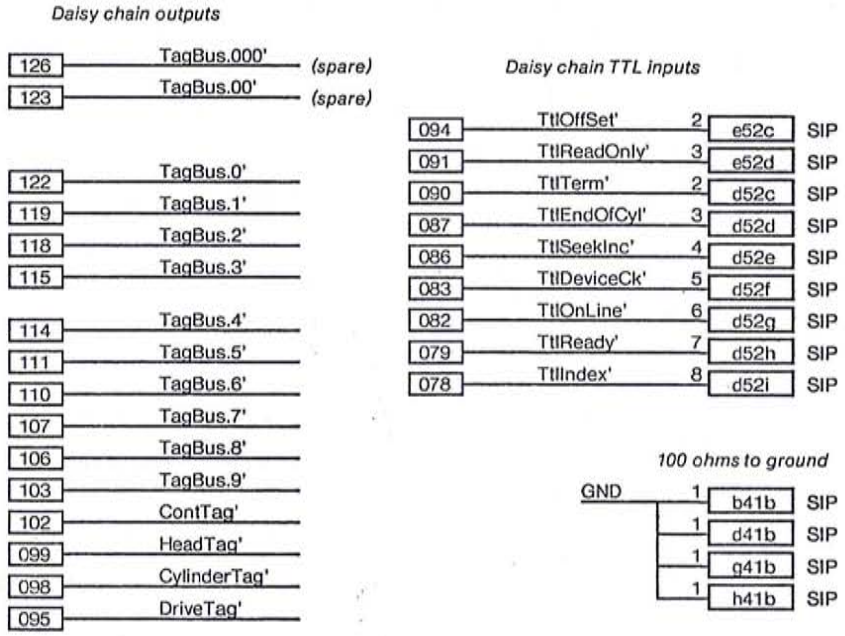
Radial Cable for Drive 1



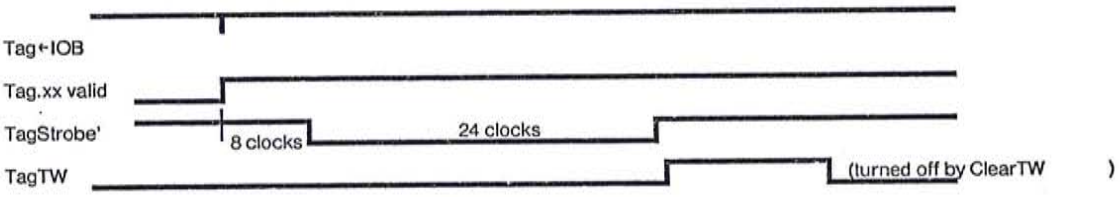
Radial Cable for Drive 3



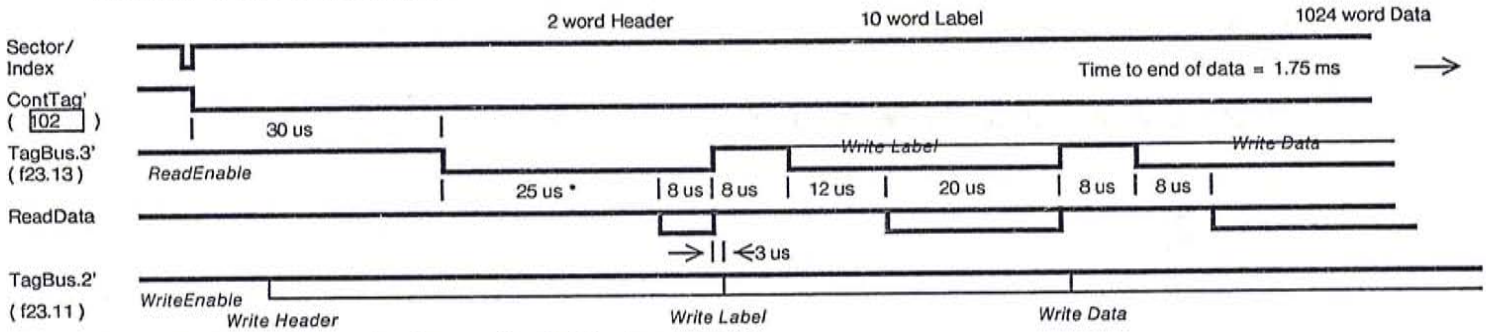
Daisy Chain Cable



Head or Cylinder Tag Instruction



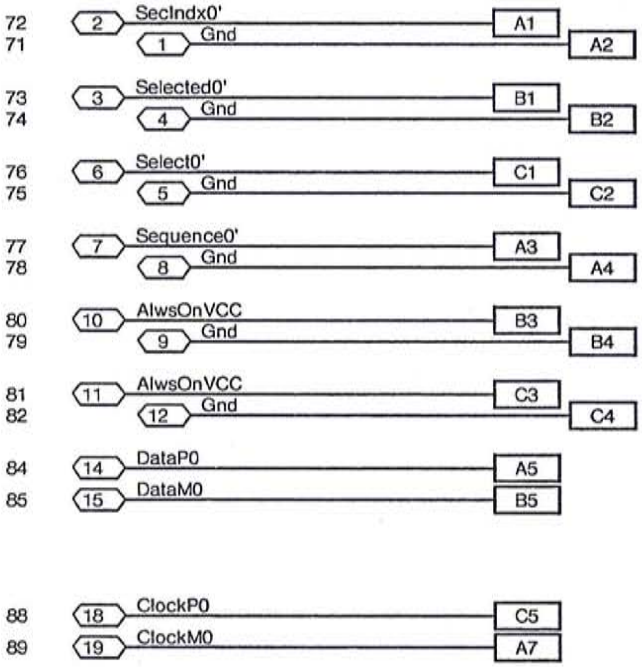
Read or Write Instruction



* This value if for reading a pack on the same drive that Headers were written
It may vary by + /- 15 us on other drives

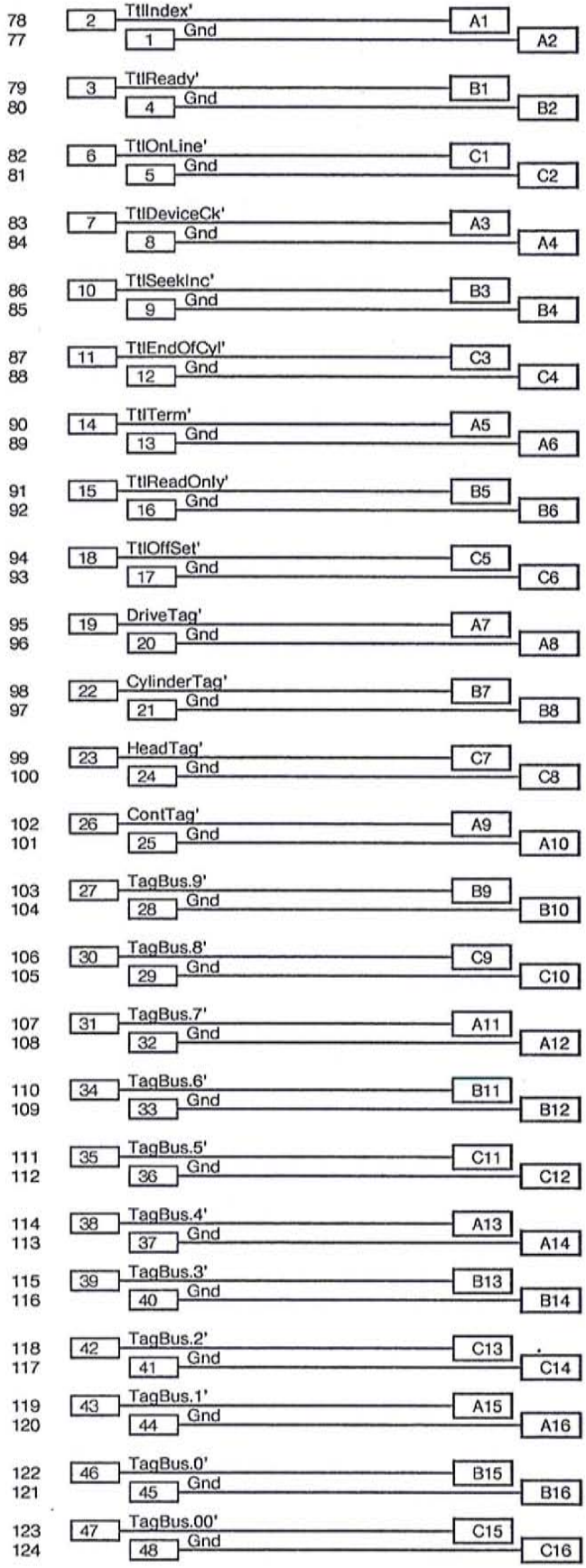
Radial Cable for Drive 0

AMP 204729-1



DAISEY CHAIN CABLE

AMP 204733-1



Radial Cable for Drive 0

AMP 204742-1

Cal-Comp
Assembly 12433

A1	SecIndx0'	violet	E04	
A2	Gnd	black		E03
B1	Selected0'	brown	E08	
B2	Gnd	black		E07
C1	Select0'	green	E12	
C2	Gnd	black		E11
A3	Sequence0'	blue	E10	
A4	Gnd	black		E09
B3	AlwsOnVCC	white	E02	
B4	Gnd	black		E05
C3	AlwsOnVCC	red	E01	
C4	Gnd	black		E05
A5	DataP0	Orenge	E14	
B5	DataM0	black	E16	
C5	ClockP0	Yellow	E18	
A7	ClockM0	Black	E20	

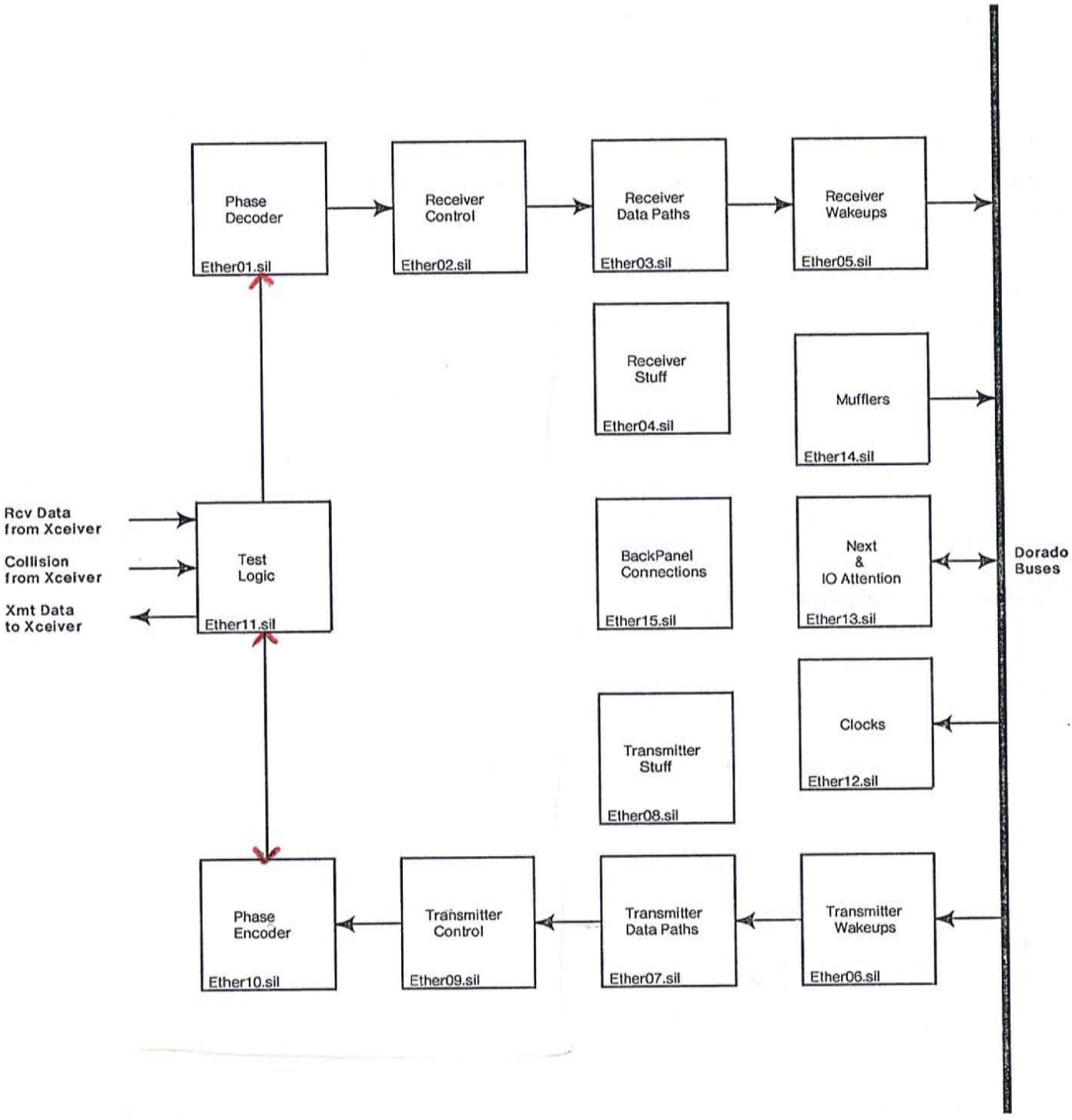
DAISEY CHAIN CABLE

AMP 204746-1

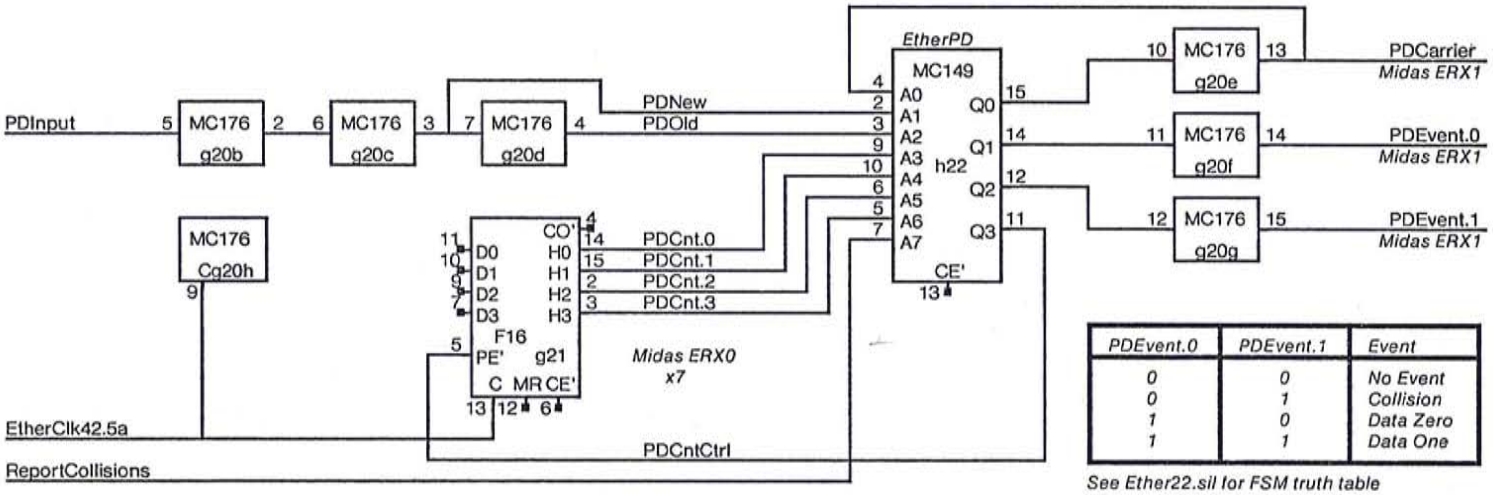
Cal-Comp
Assembly 12424

A1	TtlIndex'	E06	
A2	Gnd		E06G
B1	TtlReady'	E08	
B2	Gnd		E08G
C1	TtlOnLine'	E14	
C2	Gnd		E14G
A3	TtlDeviceCk'	E12	
A4	Gnd		E12G
B3	TtlSeekInc'	E16	
B4	Gnd		E16G
C3	TtlEndOfCyl'	E02	
C4	Gnd		E02G
A5	TtlTerm'	E35	
A6	Gnd		E35G
B5	TtlReadOnly'	E10	
B6	Gnd		E10G
C5	TtlOffSet'	E04	
C6	Gnd		E04G
A7	Not used		
A8	Gnd		
B7	CylinderTag'	E39	
B8	Gnd		E39G
C7	HeadTag'	E40	
C8	Gnd		E40G
A9	ContTag'	E37	
A10	Gnd		E37G
B9	TagBus.9'	E38	
B10	Gnd		E38G
C9	TagBus.8'	E36	
C10	Gnd		E36G
A11	TagBus.7'	E34	
A12	Gnd		E34G
B11	TagBus.6'	E32	
B12	Gnd		E32G
C11	TagBus.5'	E30	
C12	Gnd		E30G
A13	TagBus.4'	E28	
A14	Gnd		E28G
B13	TagBus.3'	E26	
B14	Gnd		E26G
C13	TagBus.2'	E24	
C14	Gnd		E24G
A15	TagBus.1'	E22	
A16	Gnd		E22G
B15	TagBus.0'	E20	
B16	Gnd		E20G
C15	Not used		
C16	Gnd		

XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Dorado to Trident Cables	TriconD17.sil	Roger Bates	Ce	7/08/79	22b



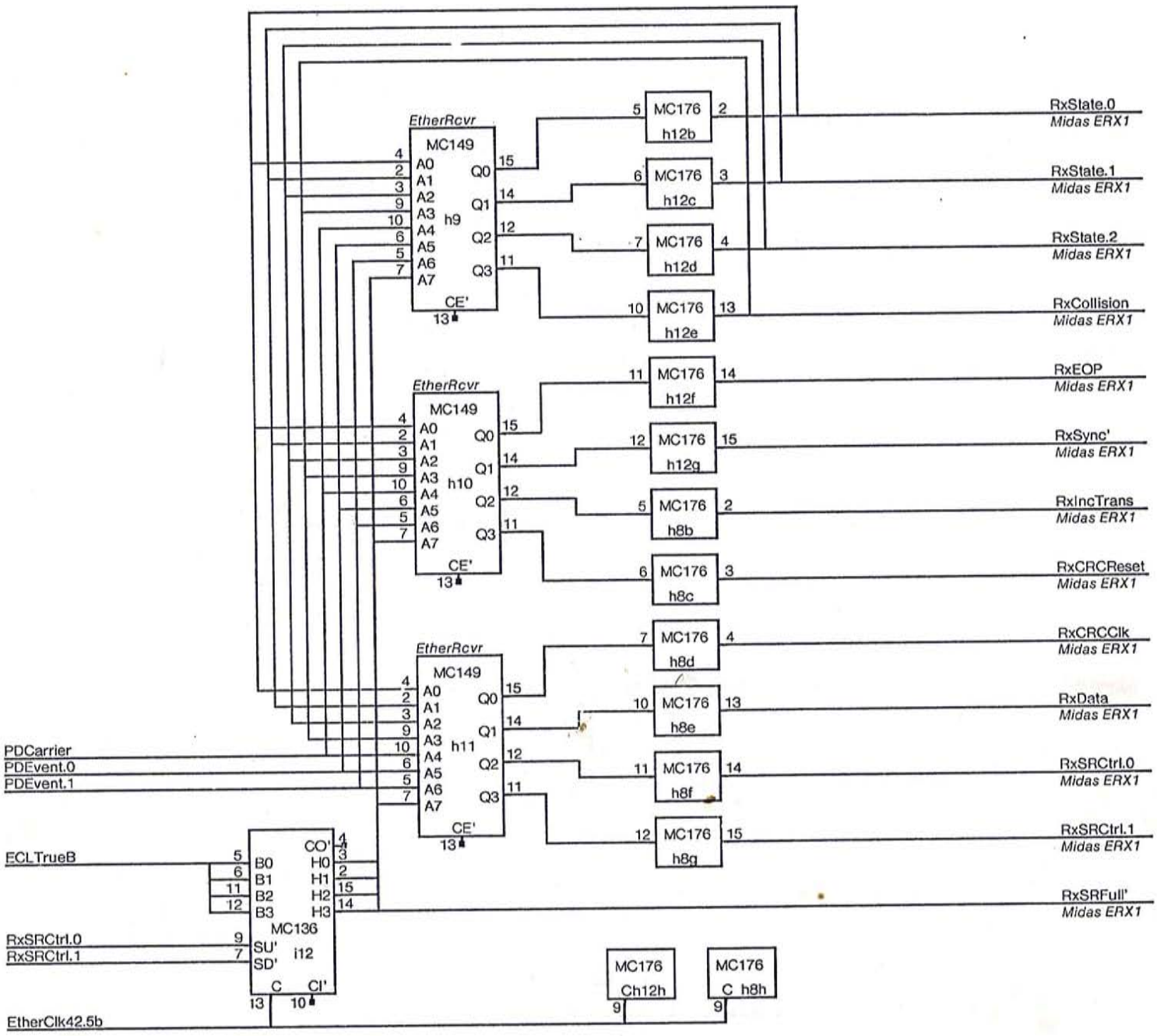
See DskEth*.sil for IOA, IOB, Muffler Control and Board Clocks



PDEvent.0	PDEvent.1	Event
0	0	No Event
0	1	Collision
1	0	Data Zero
1	1	Data One

See Ether22.sil for FSM truth table

XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Phase Decoder	Ether01.sil	David Boggs	Ce	7/08/79	24



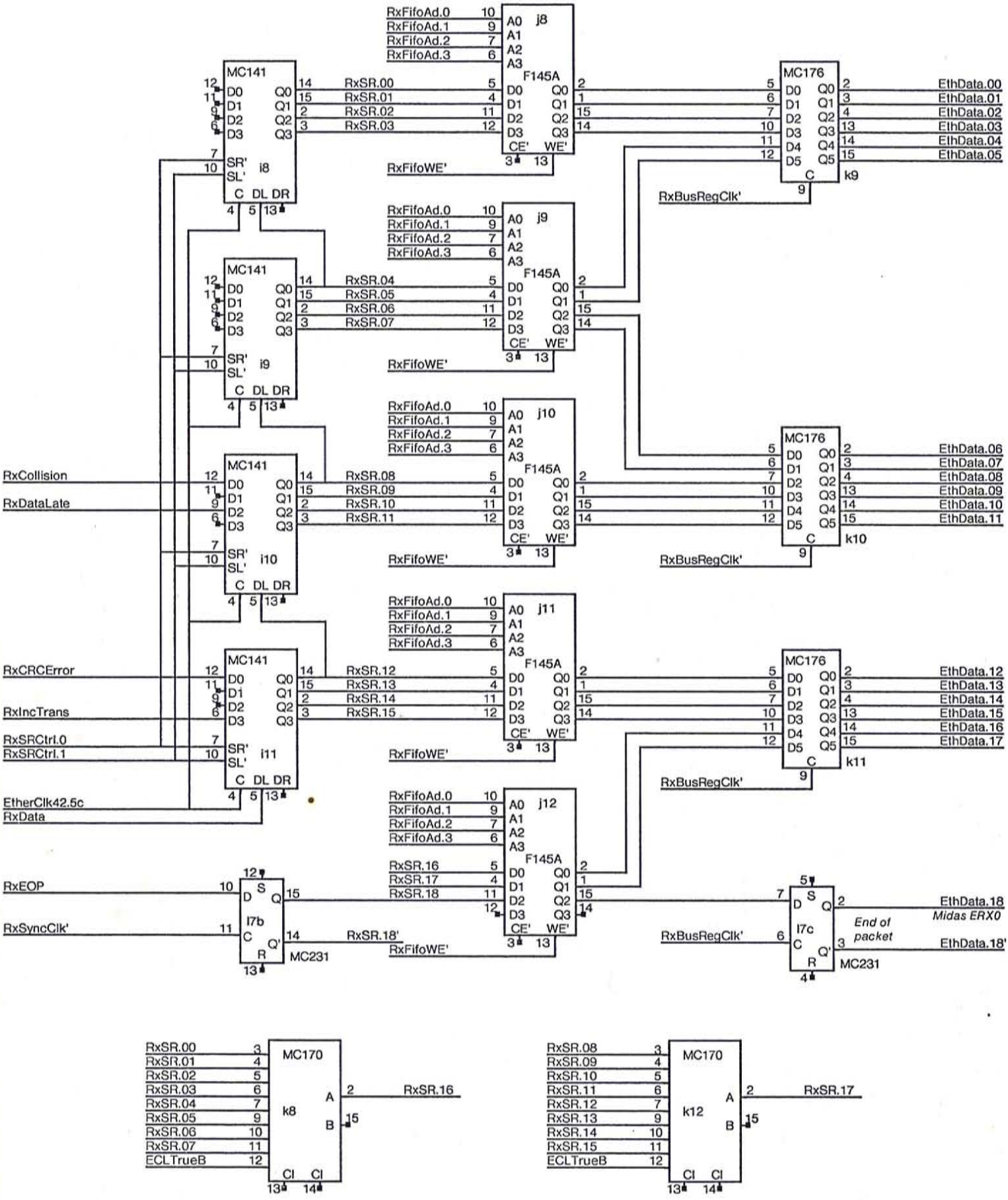
See Ether17.sil for timing diagrams

See Ether18.sil for FSM truth table

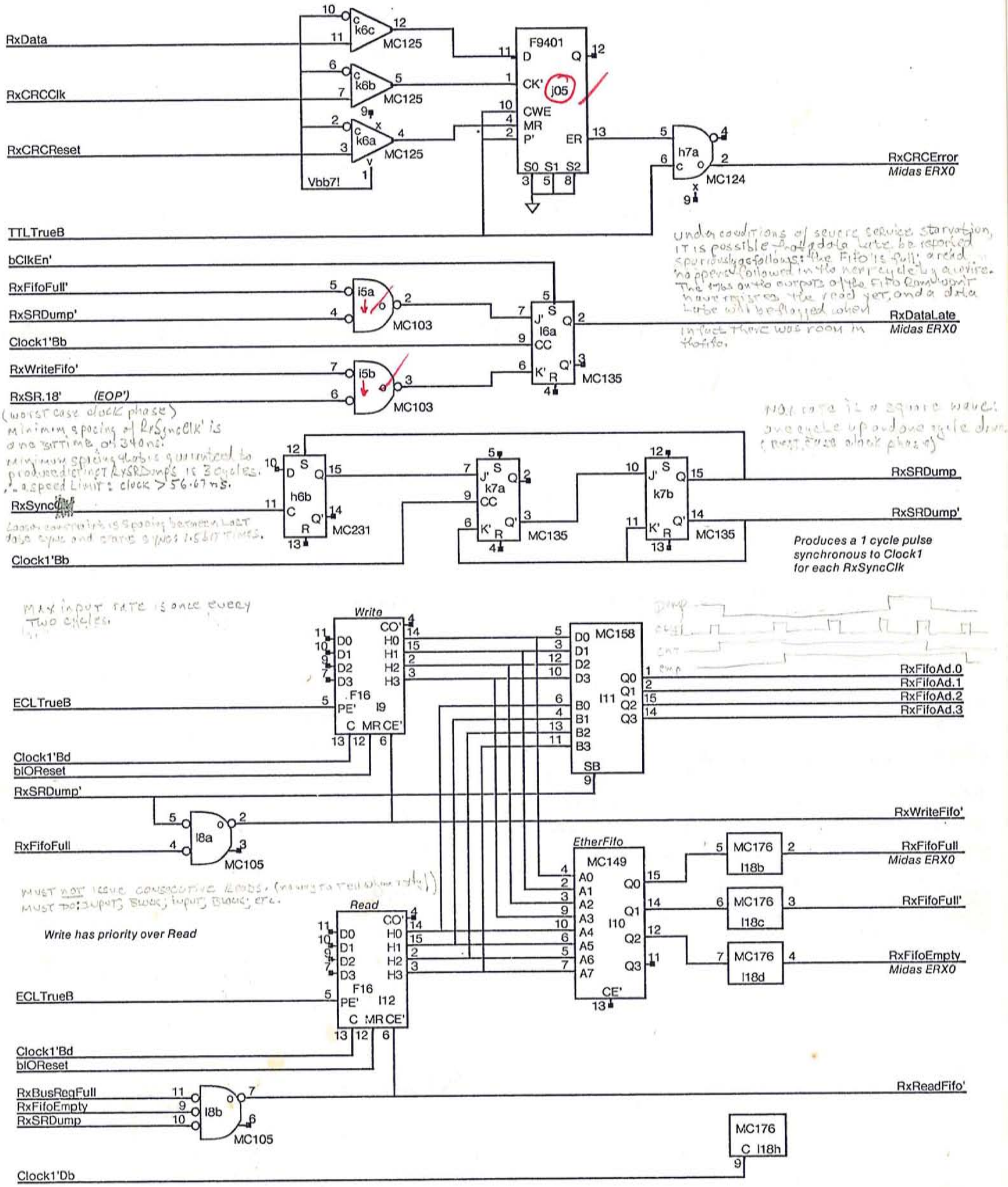
The slowest Dorado clock speed at which the receiver works is 56.67 ns (To to T1)

Max Receiver Speed at 25 ns = 6.67 mIs
35 ns = 4.75 mIs

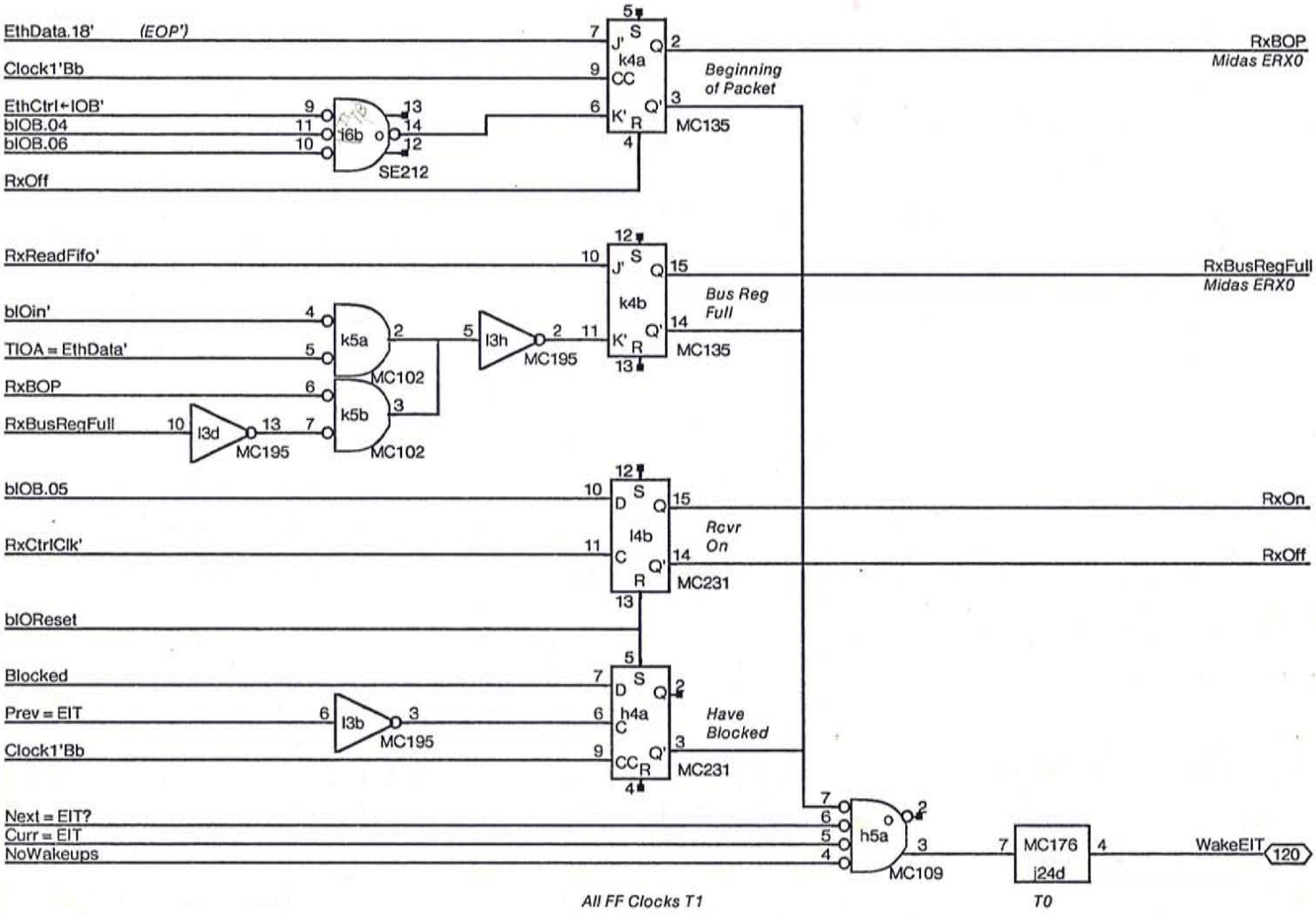
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Receiver Control	Ether02.sil	David Boggs	Ce	9/24/79	25



XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Receiver Data Paths	Ether03.sil	David Boggs	Ce	7/08/79	26

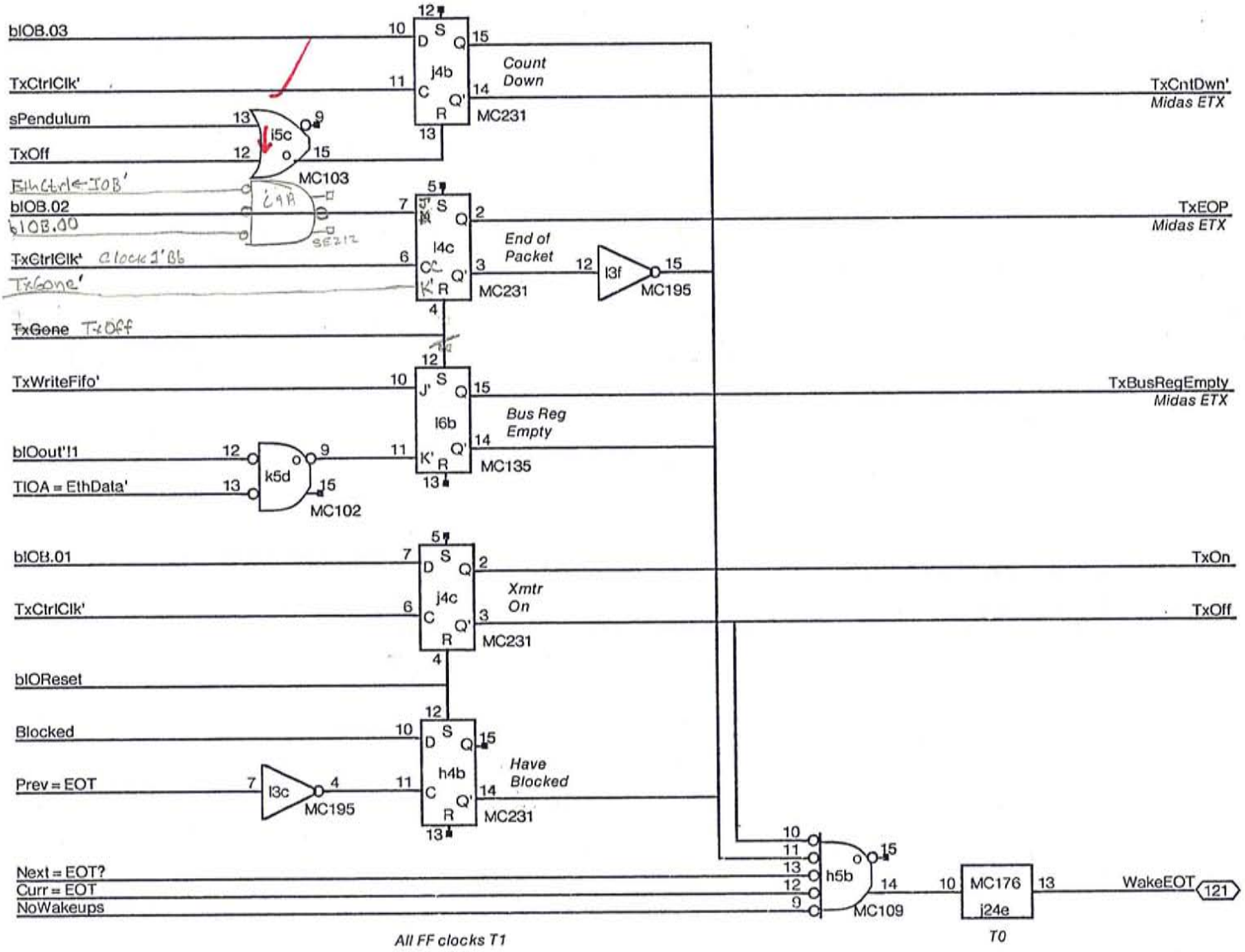


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Receiver Stuff	Ether04.sil	David Boggs	Ce	7/08/79	27



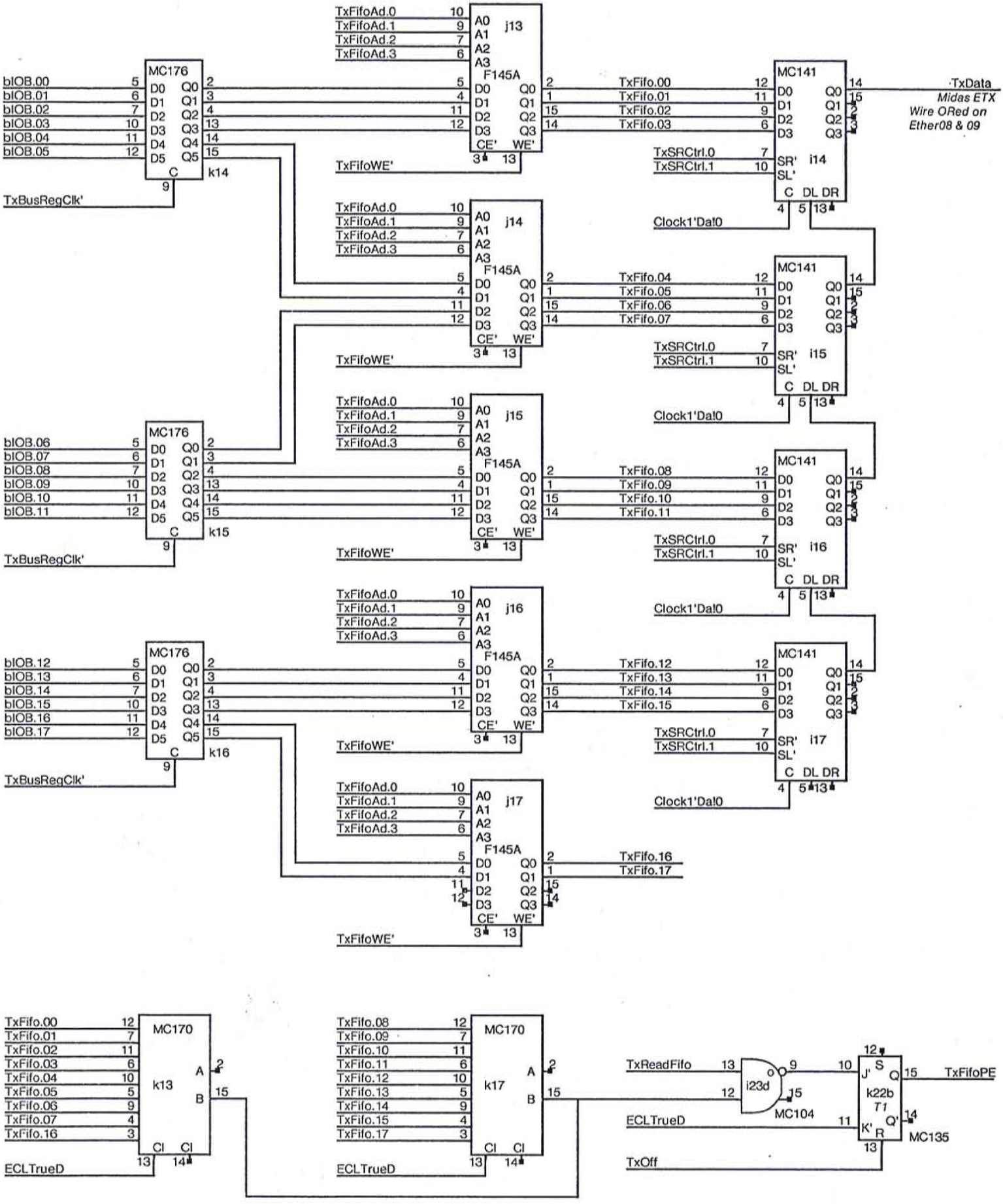
See Ether23 & 24.sil for wakeup timing diagrams

XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Receiver Wakeups	Ether05.sil	David Boggs	Ce	7/08/79	28

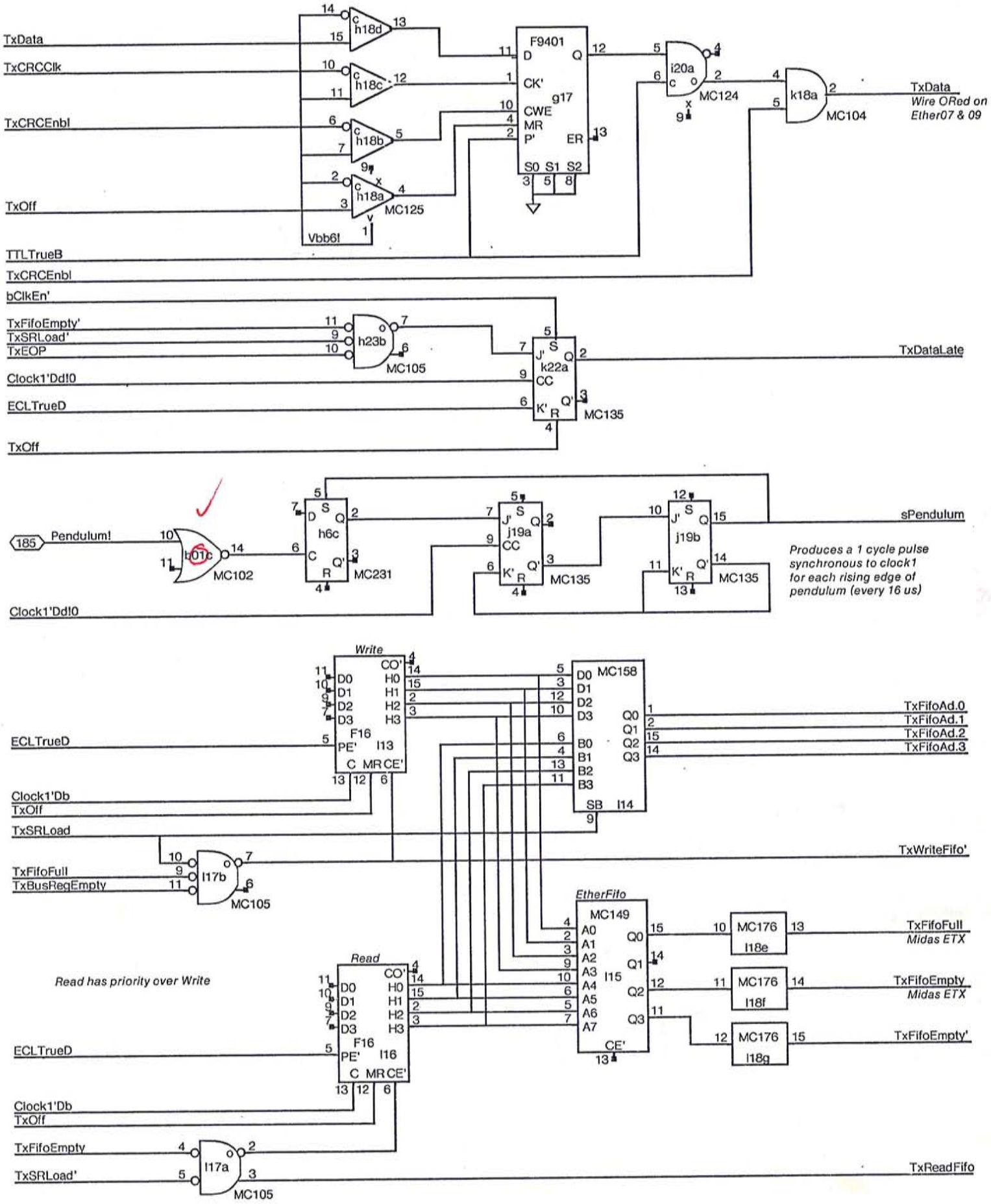


See Ether23 & 24.sil for wakeup timing diagrams

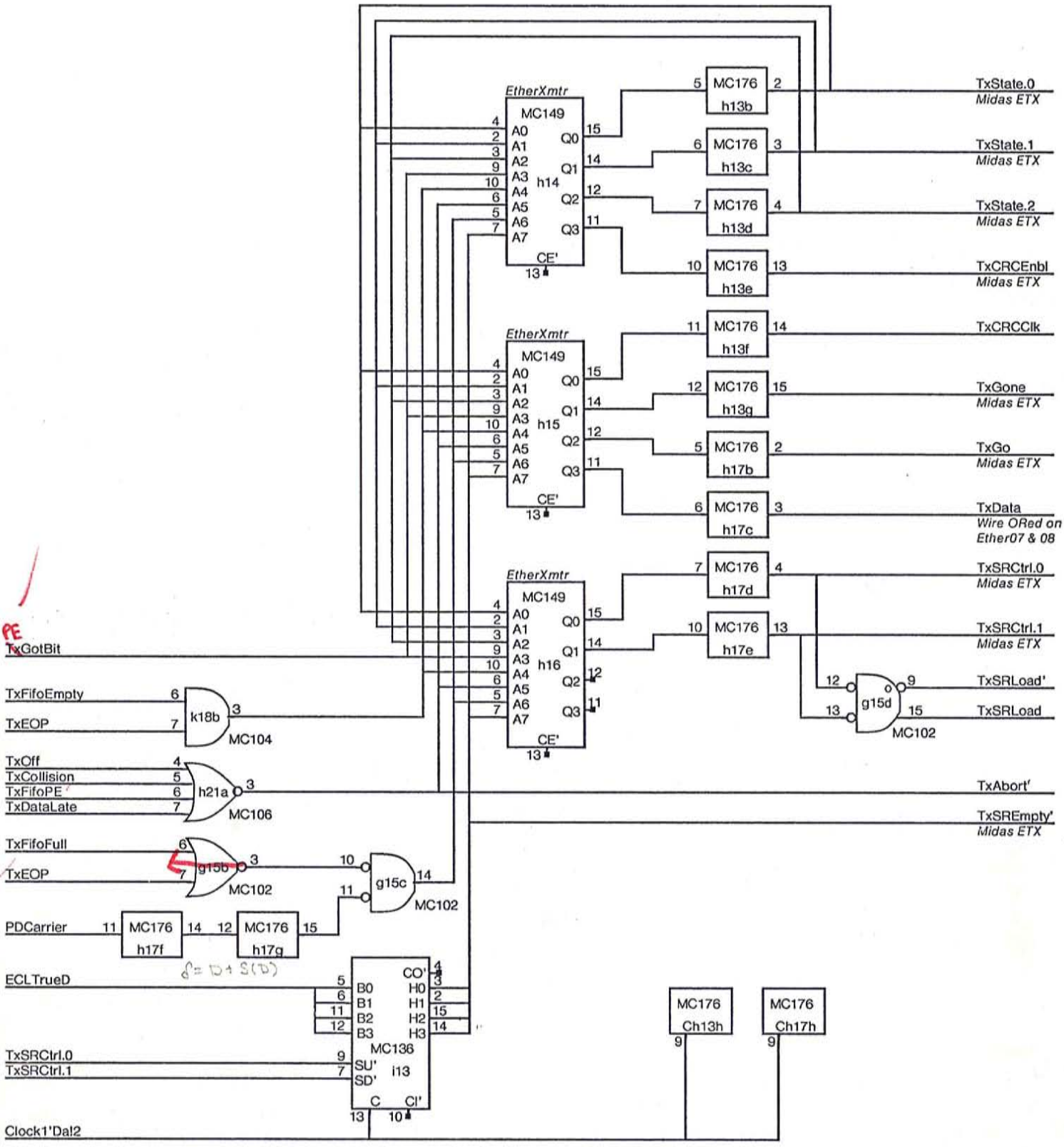
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Transmitter Wakeups	Ether06.sil	David Boggs	Ce	7/08/79	29



XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Transmitter Data Paths	Ether07.sil	David Boggs	Ce	7/08/79	30



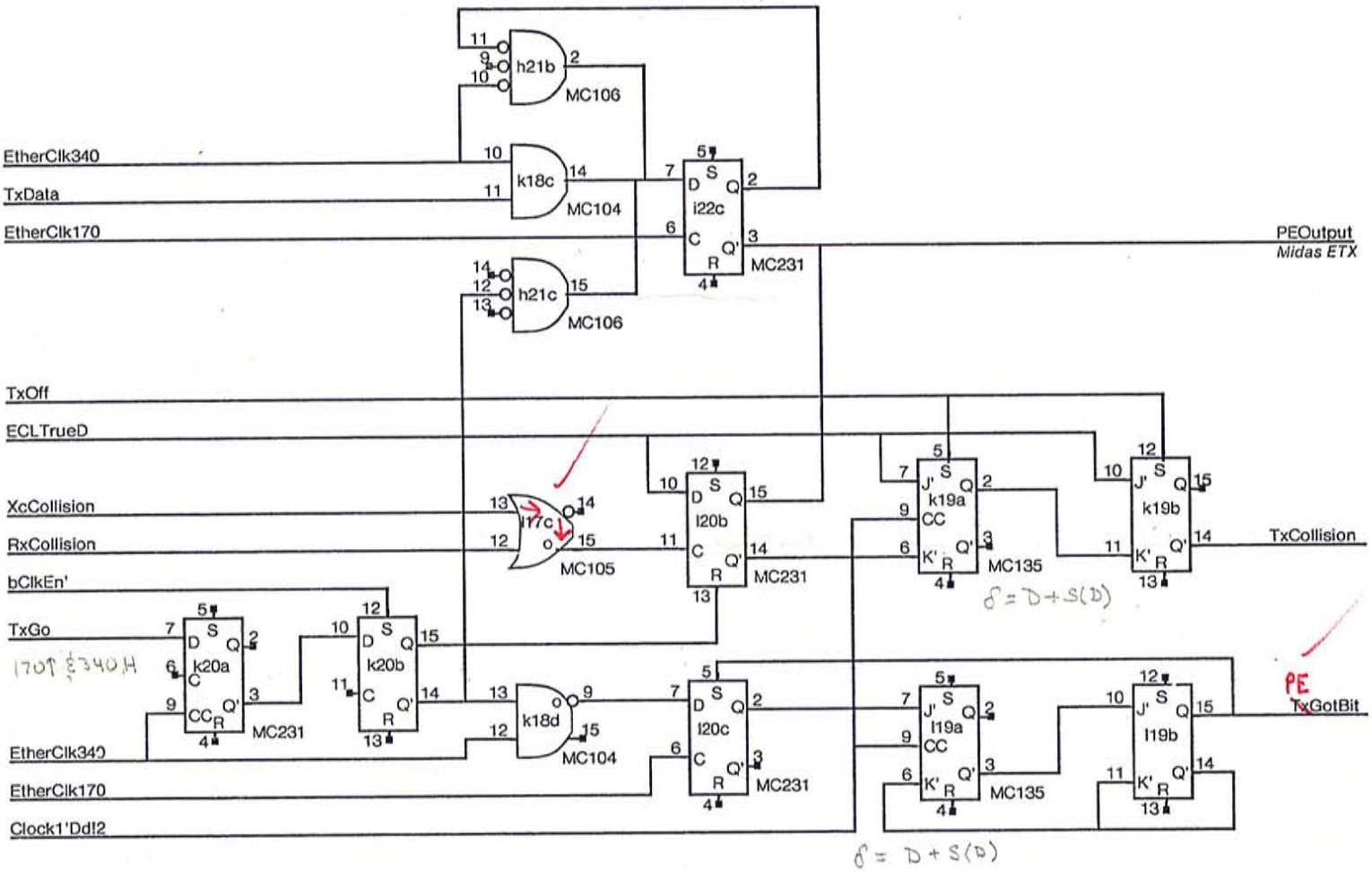
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Transmitter Stuff	Ether08.sil	David Boggs	Ce	7/08/79	31



See Ether19.sil for timing diagrams

See Ether20.sil for FSM truth table

XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Transmitter Control	Ether09.sil	David Boggs	Ce	9/24/79	32

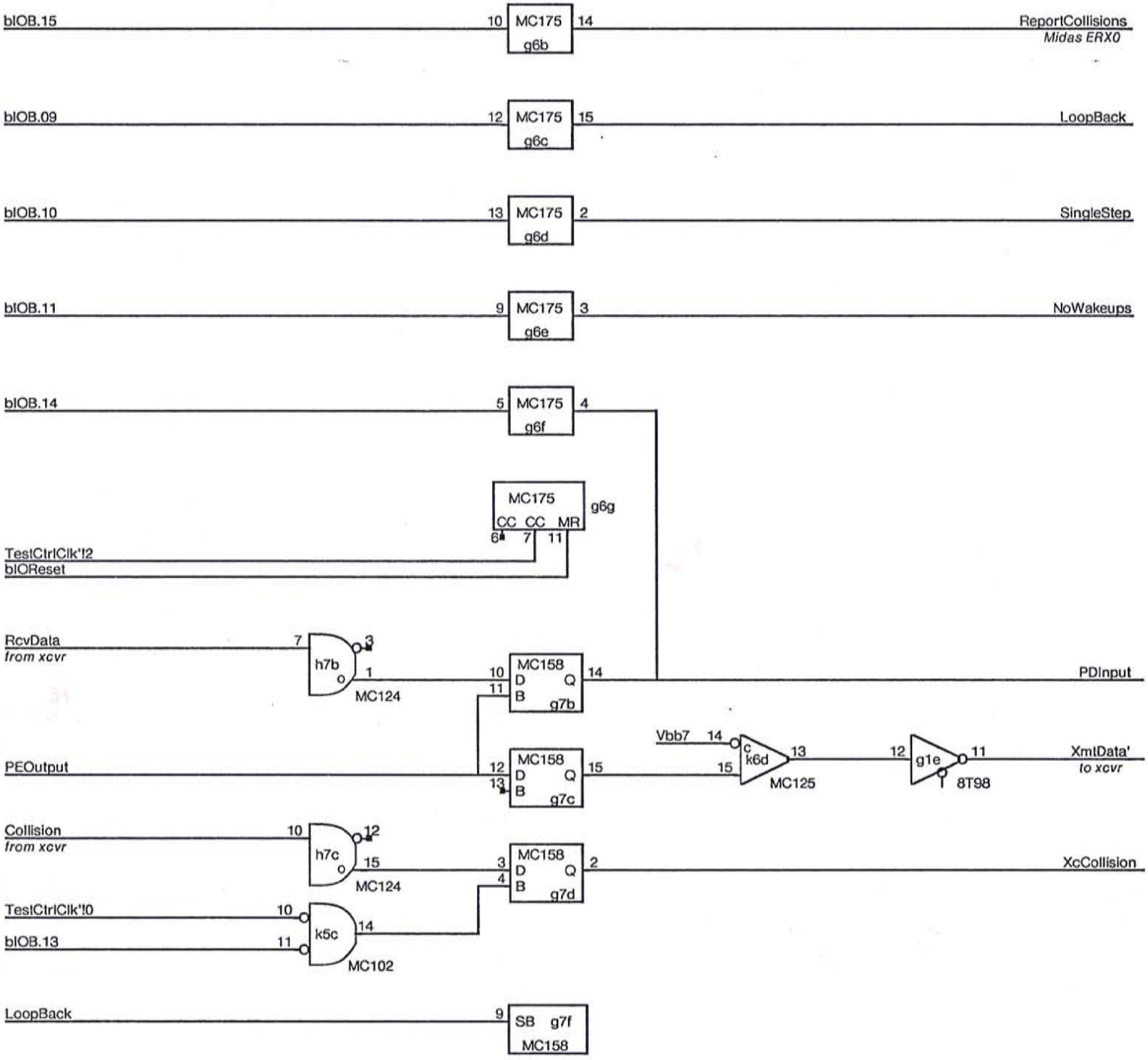


See Ether21.sil for timing diagrams

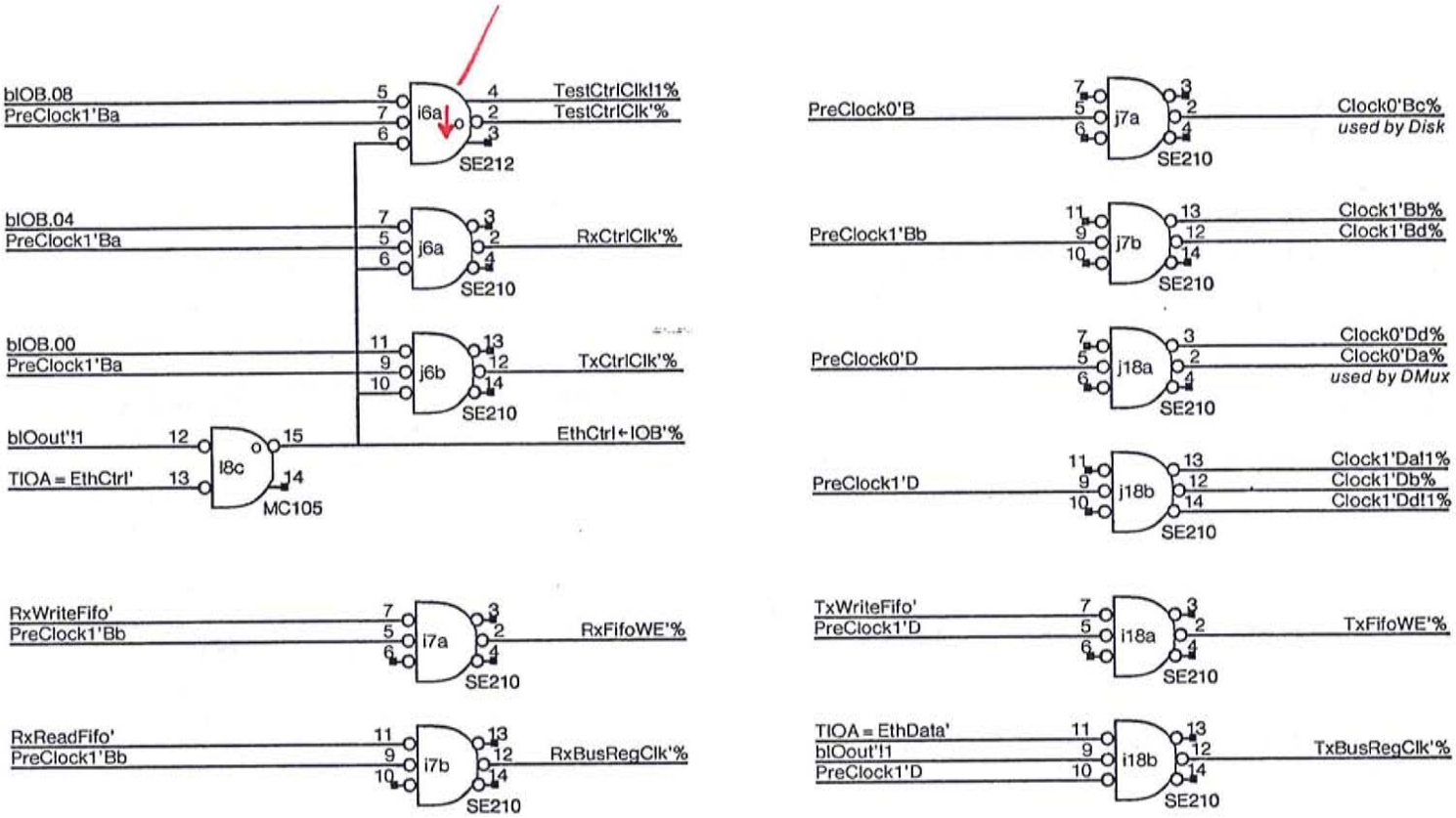
The slowest Dorado clock speed at which the transmitter works is 42.5 ns (T0 to T1)

525 ns < length of jam < 915 ns (at 25 ns clock)

XEROX PARC	Project Dorado	Drawing Phase Encoder	File Ether10.sil	Designer David Boggs	Rev Ce	Date 9/24/79	Page 33

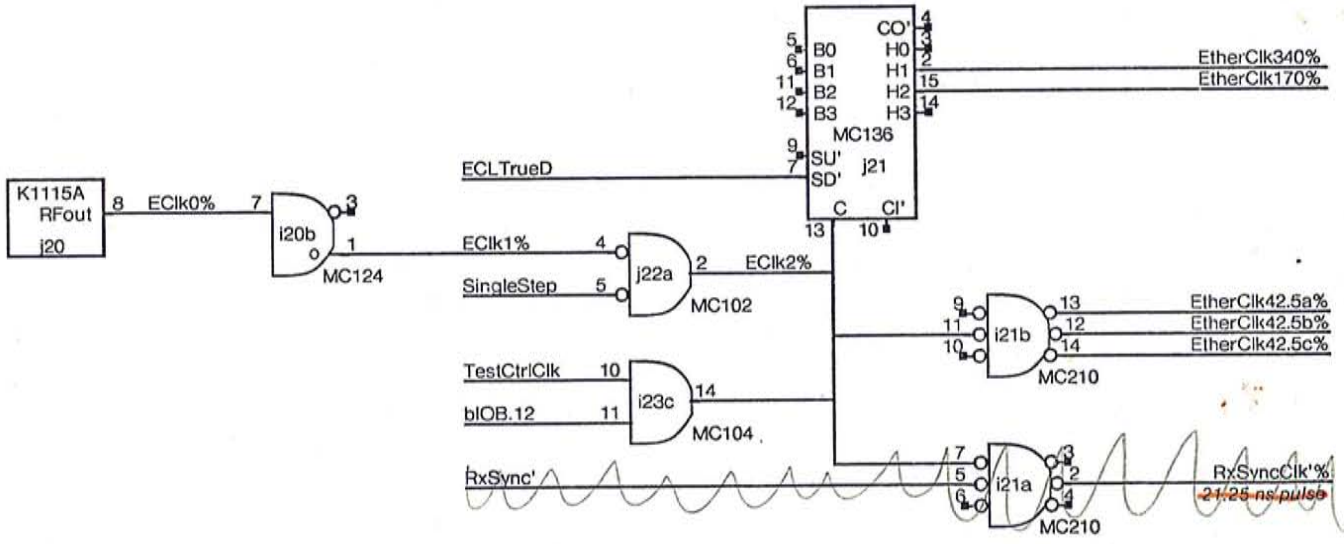


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Test Logic	Ether11.sil	David Boggs	Ce	7/08/79	34

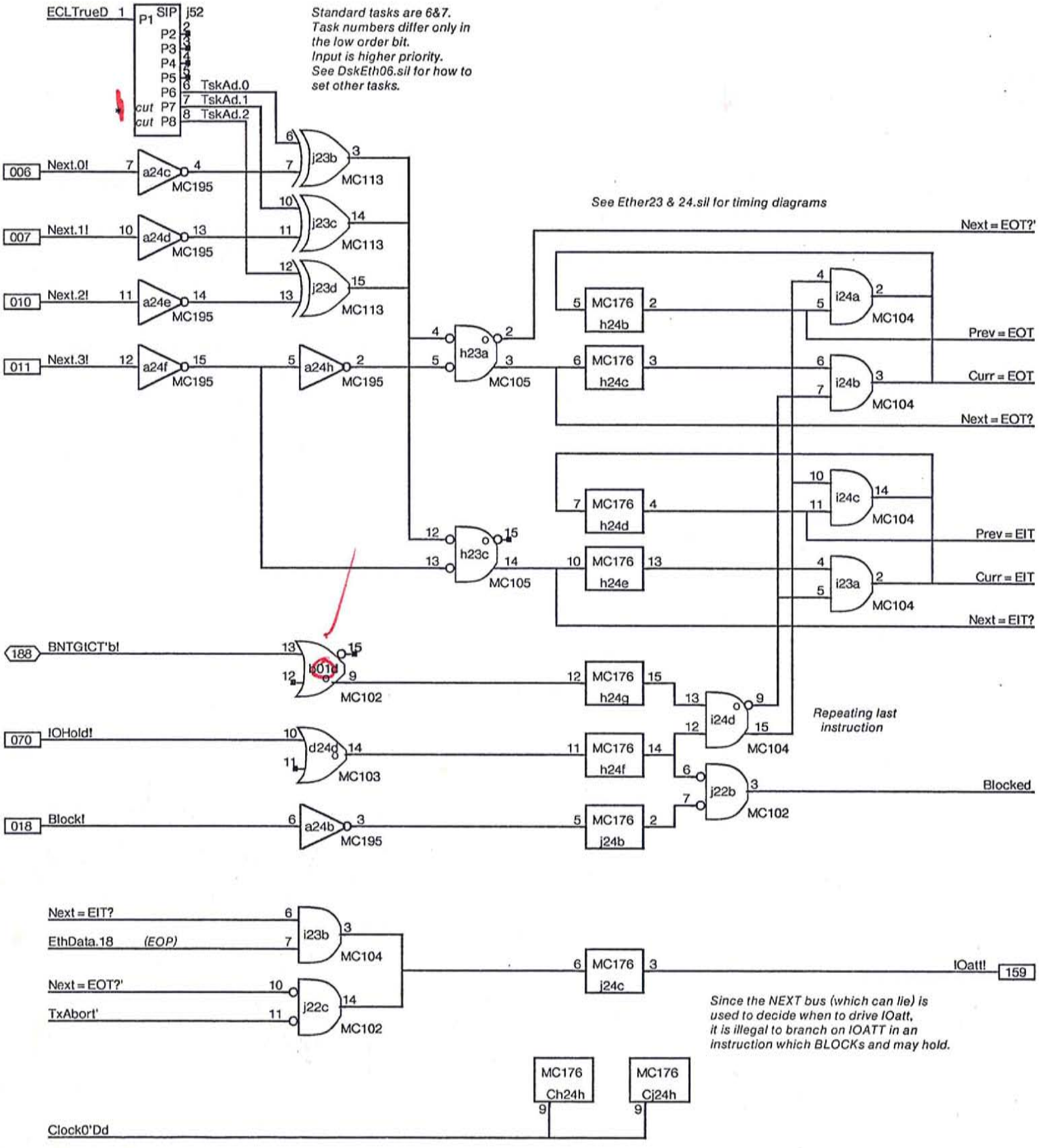


Dorado Synchronous Clocks

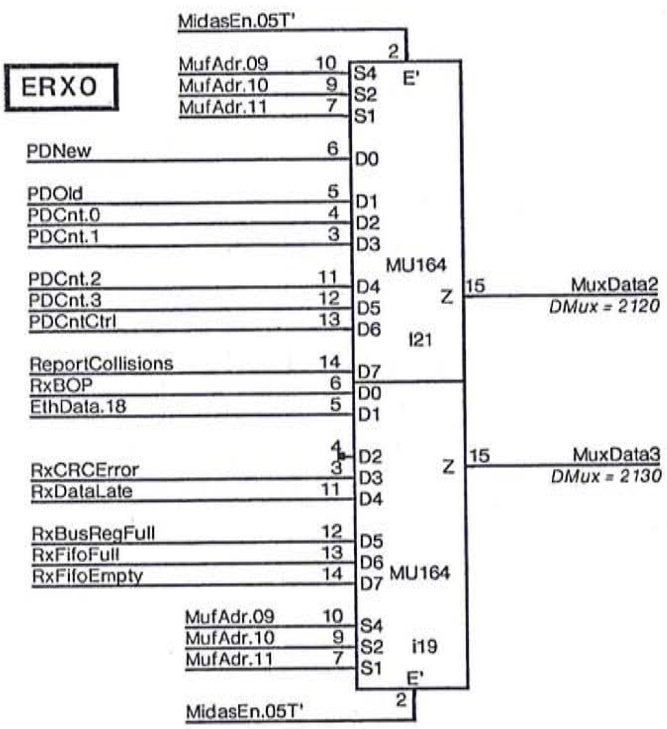
Free-running Ether Clocks

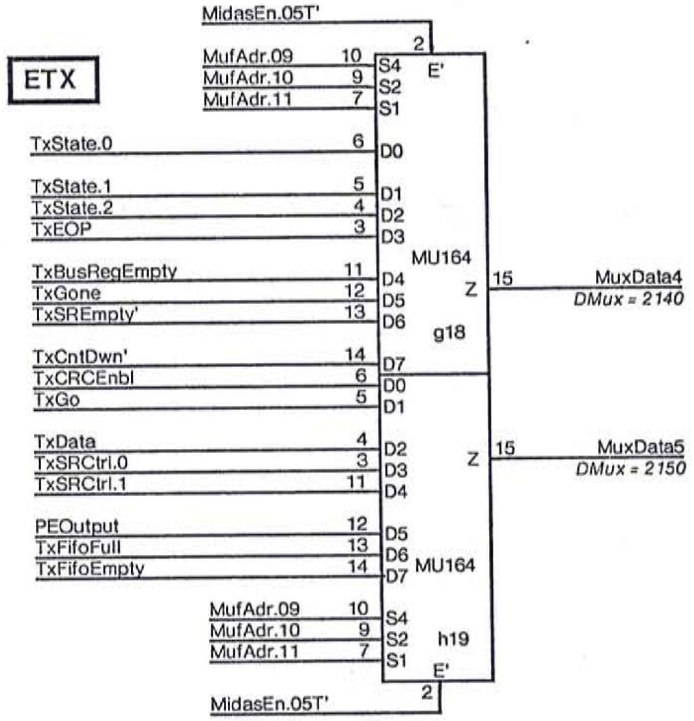


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Clocks	Ether12.sil	David Boggs	Ce	9/24/79	35

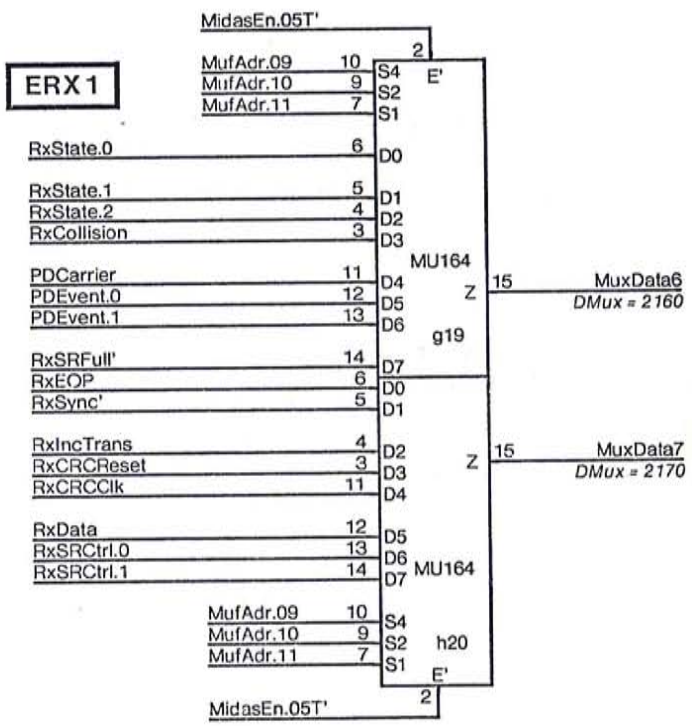


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Next & IOAttention	Ether13.sil	David Boggs	Ce	7/08/79	36



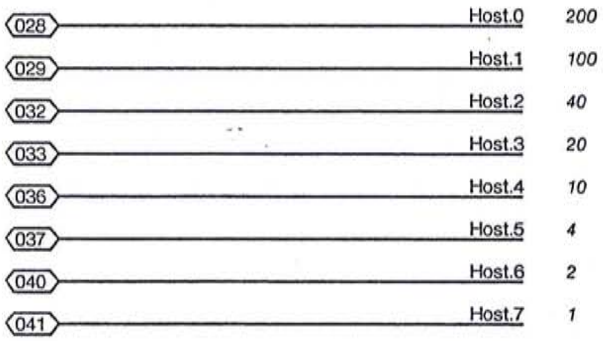


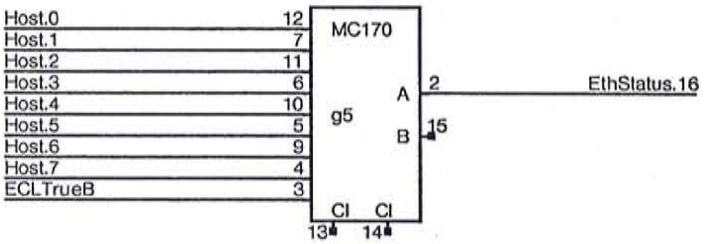
See DskEth01.sil for muffler control logic.
DMux addresses 2000-2117 are used by the disk.

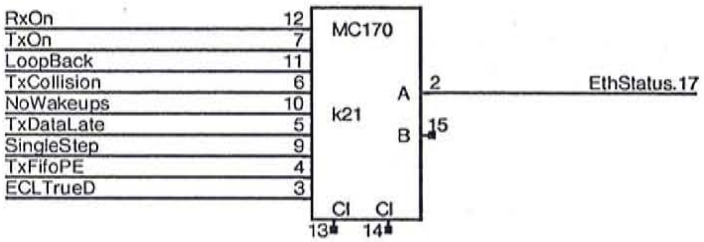


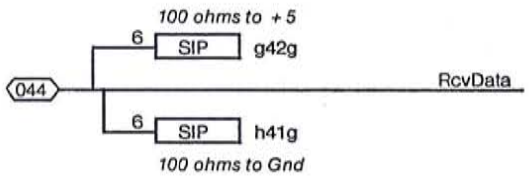
XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	Mufflers	Ether14.sil	David Boggs	Ce	7/08/79	37

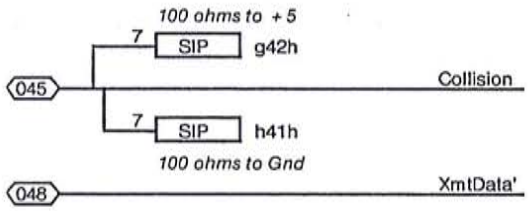
To set a host address bit to 1
pull it up to gnd through 91 ohms.

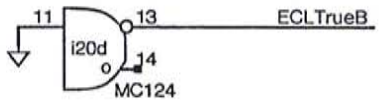


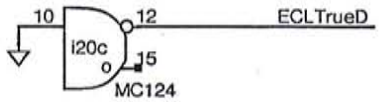


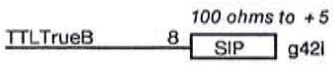




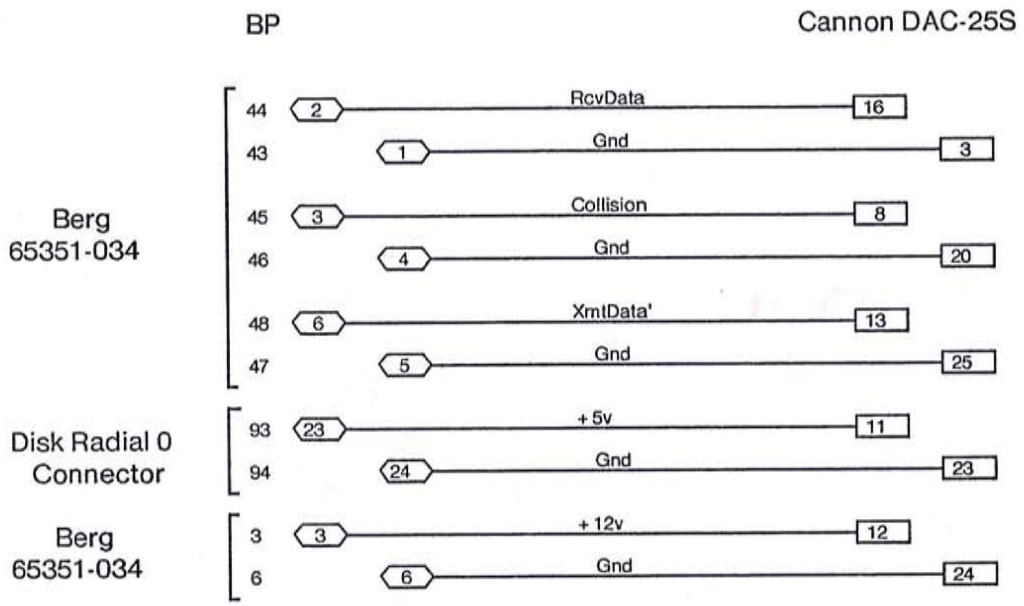






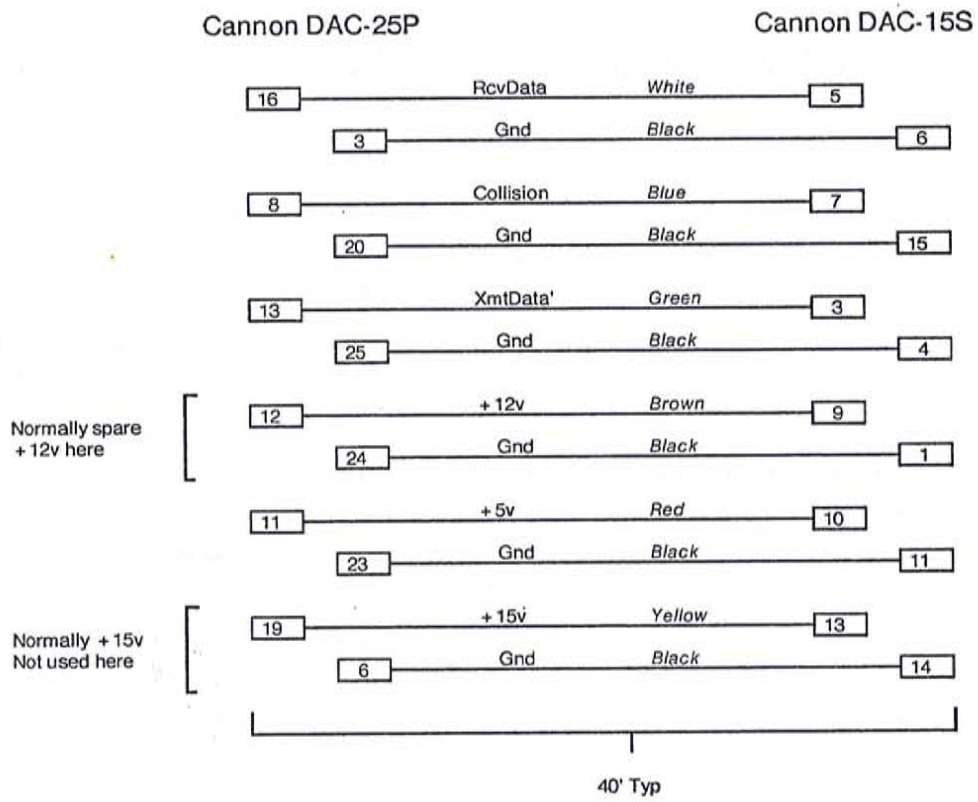


XEROX PARC	Project	Drawing	File	Designer	Rev	Date	Page
	Dorado	BackPanel Connections	Ether15.sil	David Boggs	Ce	9/24/79	38

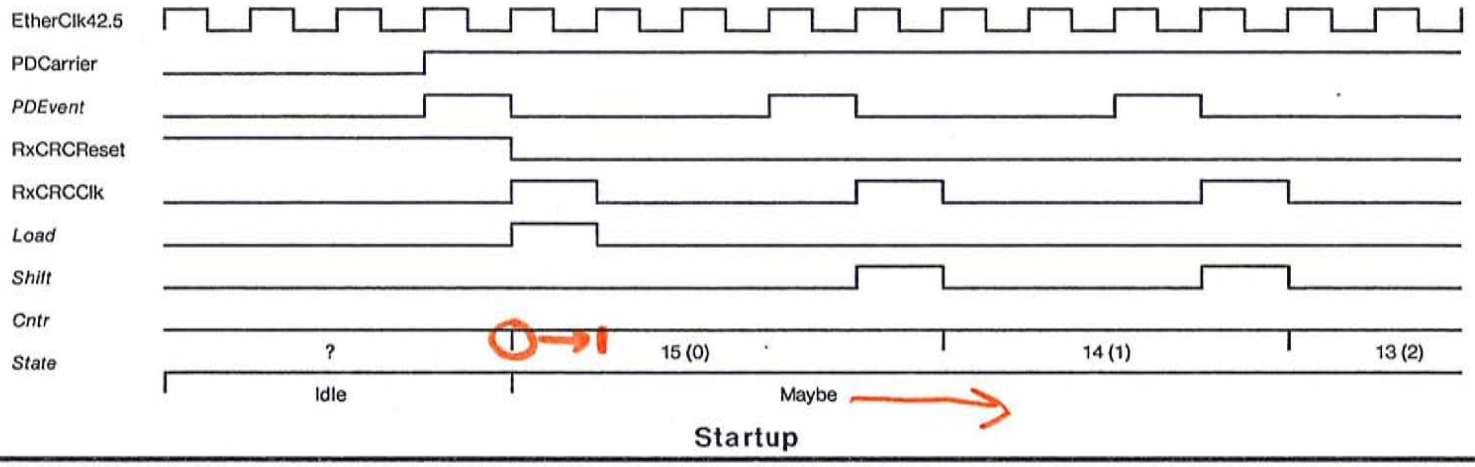


Internal Cable

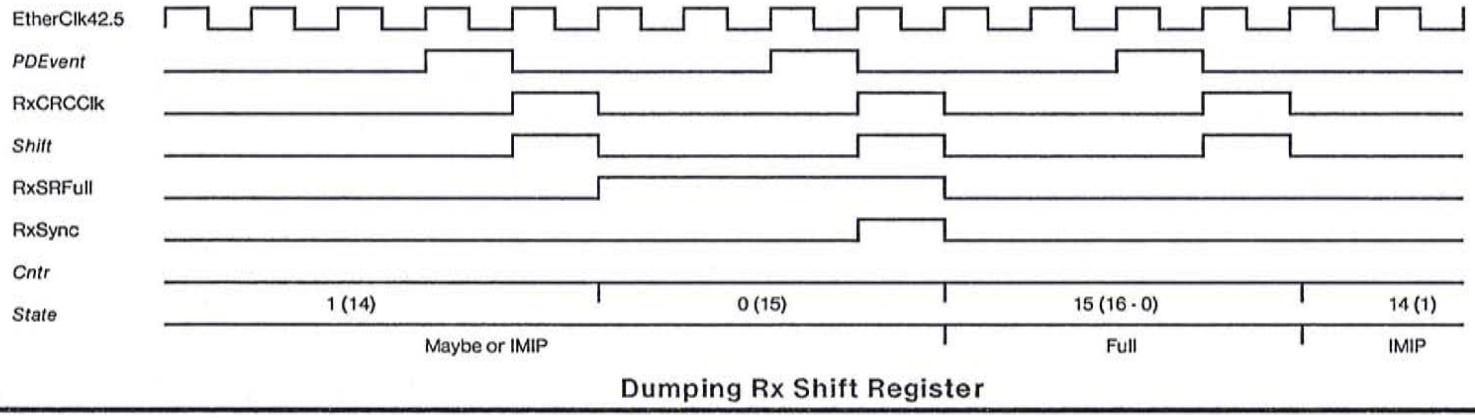
External Cable



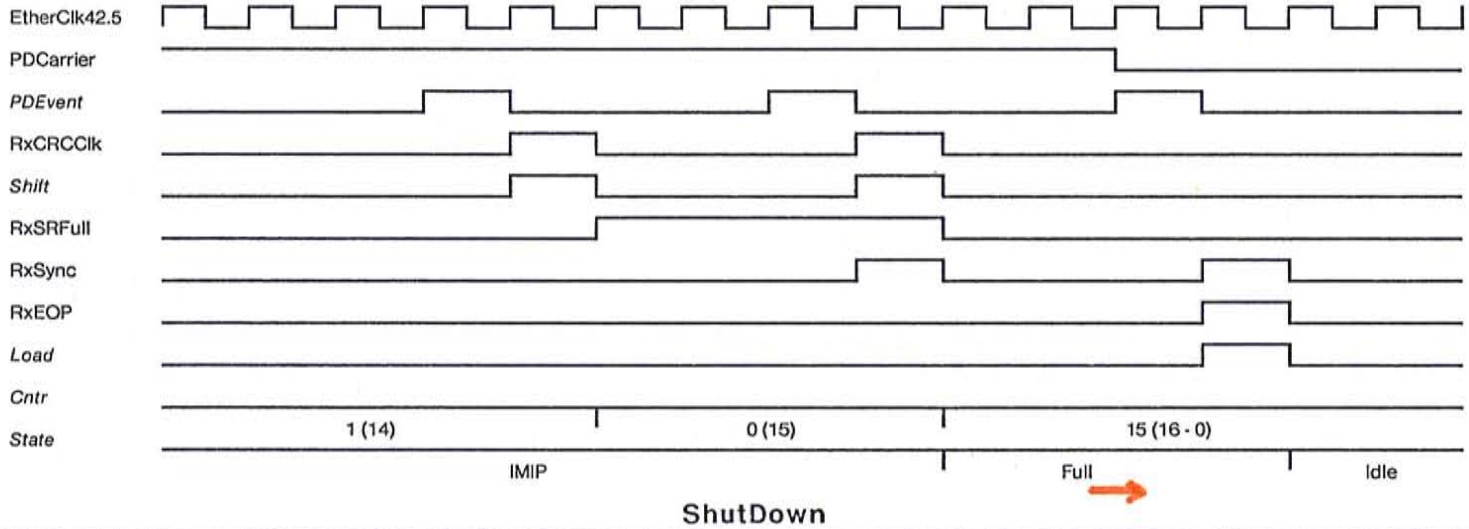
This is a standard Alto II
Ethernet external cable
part # 216411



Startup



Dumping Rx Shift Register

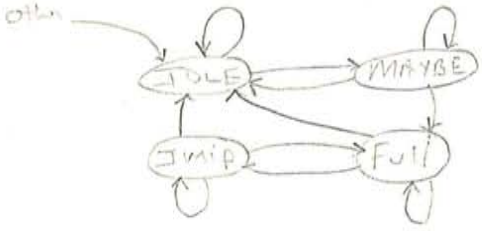


ShutDown

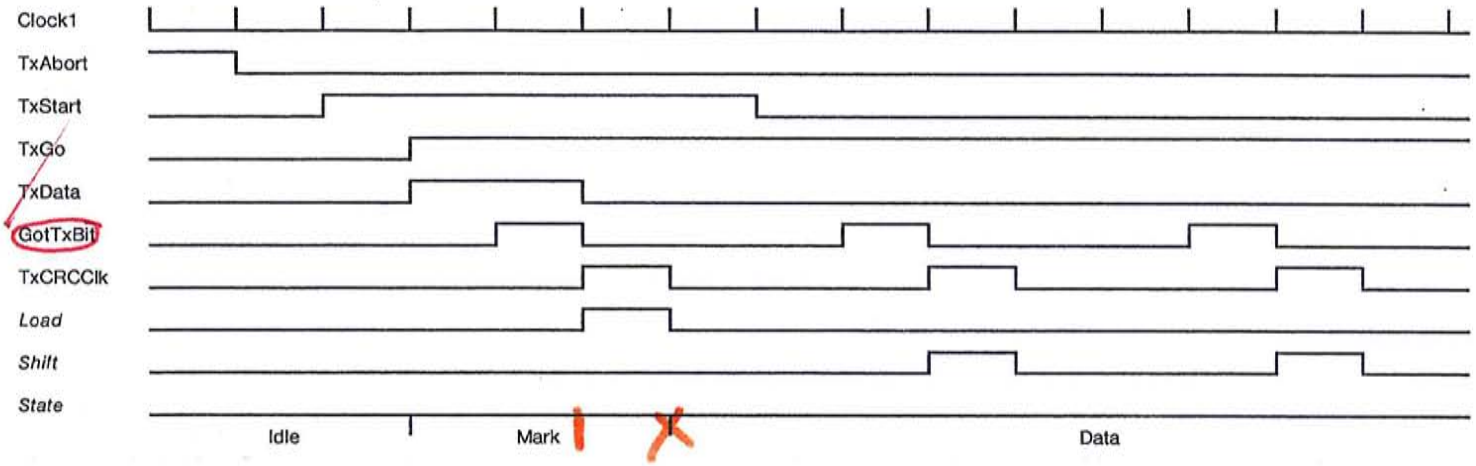
Cntr notation is <cntr value>(< # occupied postions>)
Phase decoder events are encoded in PDEvent.x
Shift and Load are encoded in RxSRCtrl.x
State is encoded in RxState.x
Shift also decrements Cntr

XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Receiver FSM Timing	Ether17.sil	David Boggs	Ce	7/08/79	41

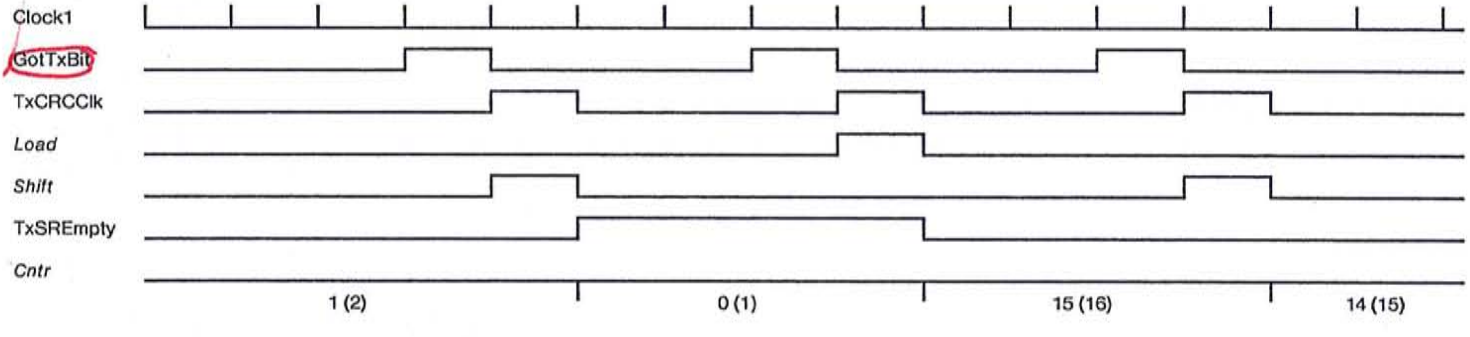
Inputs					Outputs								
State	Collision	Event	Carrier	SRFull	State	Collision	EOP	Sync	IncTrans	CRCReset	CRCClk	Data	SRCtrl
Idle	—	—	False	—	Idle	False	False	False	False	High	Low	Low	Hold
Idle	—	NoEvent	True	—	Maybe	False	False	False	False	High	Low	Low	Hold
Idle	—	Collision Zero	True	—	Maybe	True	False	False	False	High	Low	Low	Hold
Idle	—	One	True	—	Maybe	False	False	False	False	Low	High	High	Load
Maybe	False	—	False	—	Idle	False	False	False	False	Low	Low	Low	Hold
Maybe	True	—	False	—	Idle	True	True	True	True	Low	Low	Low	Load
Maybe	*	NoEvent	True	—	Maybe	*	False	False	False	Low	Low	Low	Hold
Maybe	—	Collision	True	—	Maybe	True	False	False	False	Low	Low	Low	Hold
Maybe	*	One Zero	True	False	Maybe	*	False	False	False	Low	High	High Low	Shift
Maybe	*	One Zero	True	True	Full	*	False	True	False	Low	High	High Low	Shift
Full	*	—	False	—	Idle	*	True	True	False	Low	Low	Low	Load
Full	*	NoEvent	True	—	Full	*	False	False	False	Low	Low	Low	Hold
Full	—	Collision	True	—	Full	True	False	False	False	Low	Low	Low	Hold
Full	*	One Zero	True	—	IMIP	*	False	False	False	Low	High	High Low	Shift
IMIP	*	—	False	—	Idle	*	True	True	True	Low	Low	Low	Load
IMIP	*	NoEvent	True	—	IMIP	*	False	False	False	Low	Low	Low	Hold
IMIP	—	Collision	True	—	IMIP	True	False	False	False	Low	Low	Low	Hold
IMIP	*	One Zero	True	False	IMIP	*	False	False	False	Low	High	High Low	Shift
IMIP	*	One Zero	True	True	Full	*	False	True	False	Low	High	High Low	Shift
Unused	*	—	—	—	Idle	*	False	False	False	Low	Low	Low	Hold



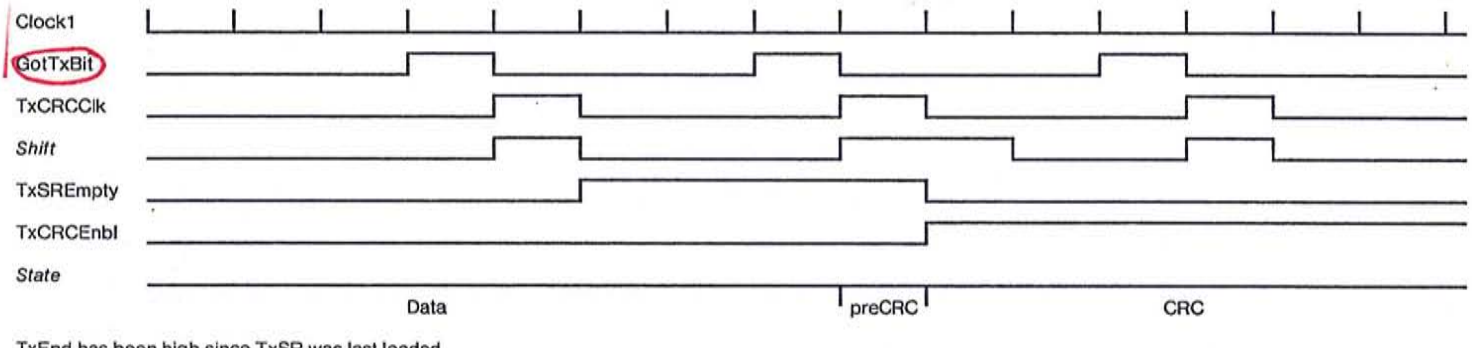
XEROX PRRC	Project: Dorado	Reference: Receiver FSM Truth Table	File: Ether18.sil	Designer: David Boggs	Rev: Ce	Date: 9/24/79	Page: 43



Startup

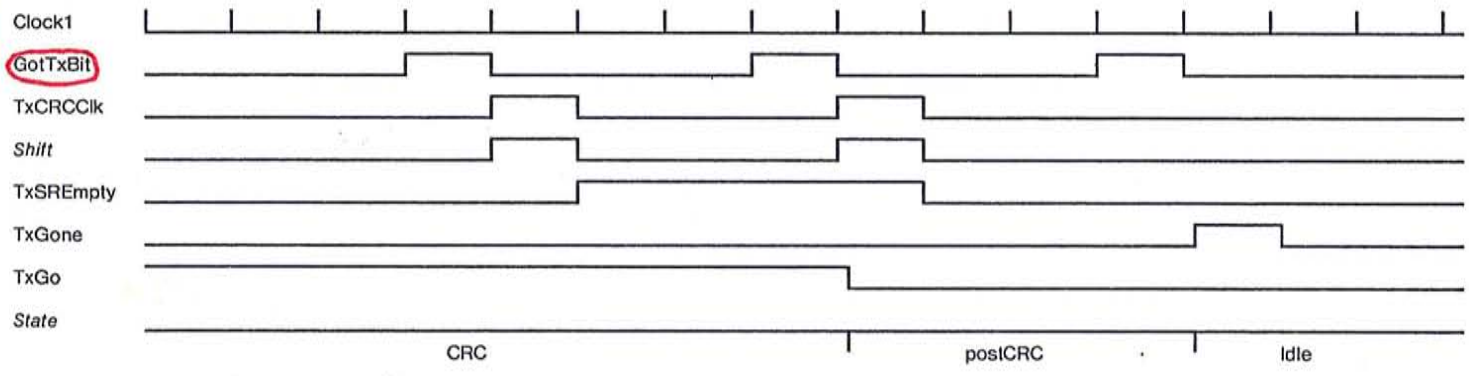


Loading Tx Shift Register



TxEnd has been high since TxSR was last loaded

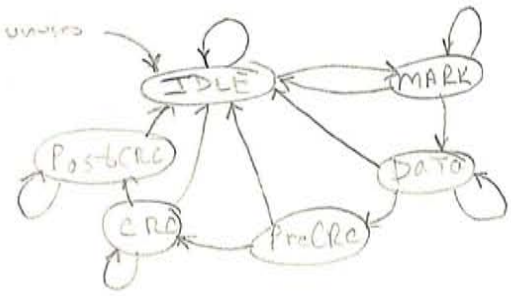
CRC

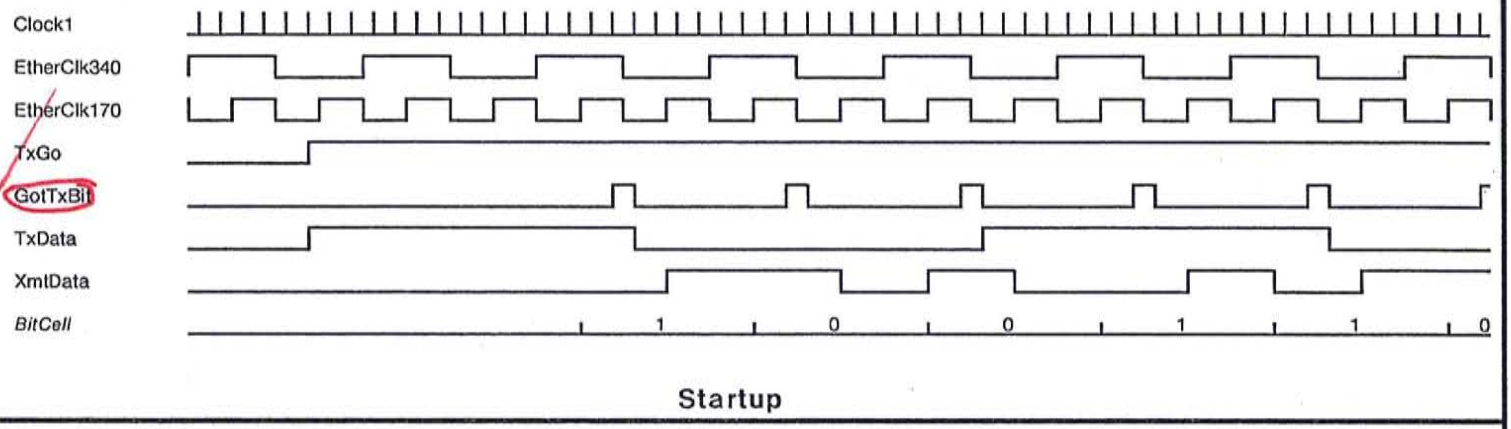


ShutDown

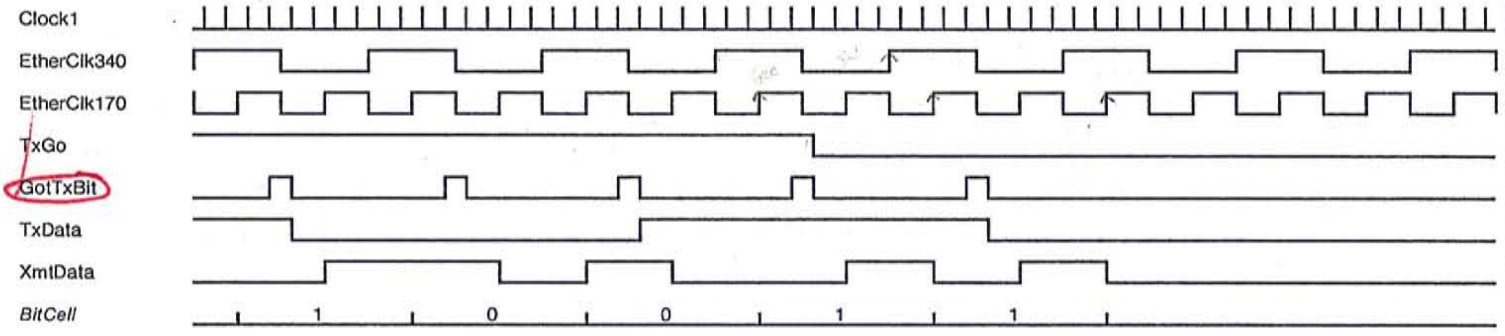
XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Transmitter FSM Timing	Ether19.sil	David Boggs	Ce	7/08/79	40

Inputs						Outputs						
State	Start	Stop	Abort	GotBit	SREmpty	State	Go	Gone	CRCEnbl	CRCClk	Data	SRCtrl
—	—	—	True	—	—	Idle	False	True	Low	Low	Low	Hold
Idle	False	—	False	—	—	Idle	False	False	Low	Low	Low	Hold
Idle	True	—	False	—	—	Mark	True	False	Low	Low	Low	Hold
Mark	—	—	False	False	—	Mark	True	False	Low	Low	High	Hold
Mark	—	—	False	True	—	Data	True	False	Low	High	High	Load
Data	—	—	False	False	—	Data	True	False	Low	Low	Low	Hold
Data	—	—	False	True	False	Data	True	False	Low	High	Low	Shift
Data	—	False	False	True	True	Data	True	False	Low	High	Low	Load
Data	—	True	False	True	True	PreCRC	True	False	Low	High	Low	Shift
PreCRC	—	—	False	—	—	CRC	True	False	High	High	Low	Shift
CRC	—	—	False	False	—	CRC	True	False	High	Low	Low	Hold
CRC	—	—	False	True	False	CRC	True	False	High	High	Low	Shift
CRC	—	—	False	True	True	PostCRC	False	False	High	High	Low	Shift
PostCRC	—	—	False	False	—	PostCRC	False	False	High	Low	Low	Hold
PostCRC	—	—	False	True	—	Idle	False	True	High	Low	Low	Hold
Unused	—	—	False	—	—	Idle	False	False	Low	Low	Low	Hold

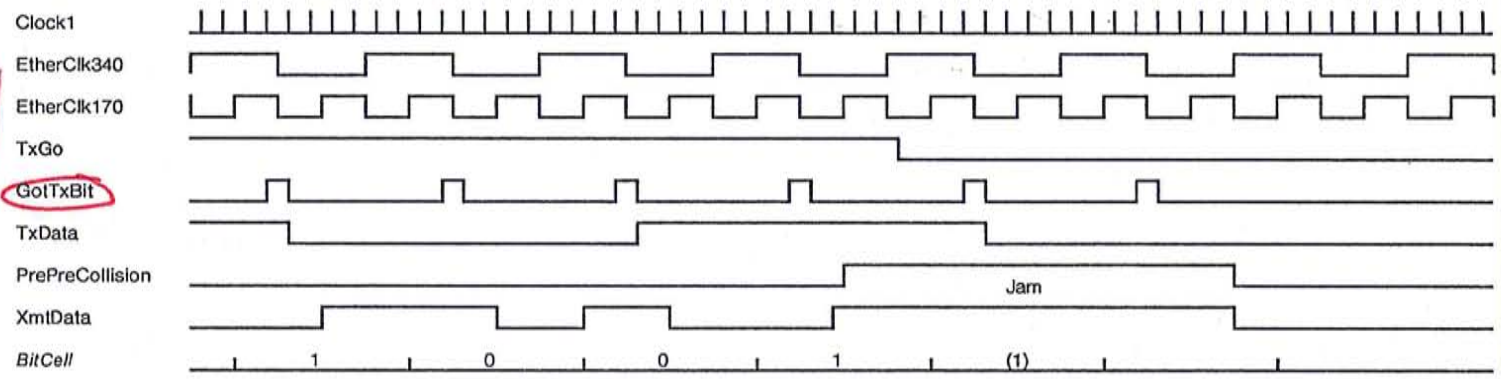




Startup



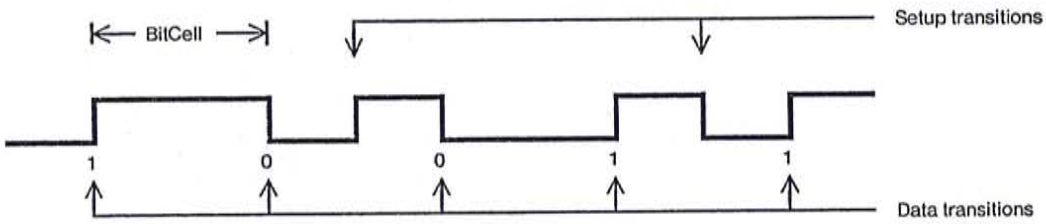
Shutdown



PrePreCollision is the output of the first stage of the Collision synchronizer

Collision

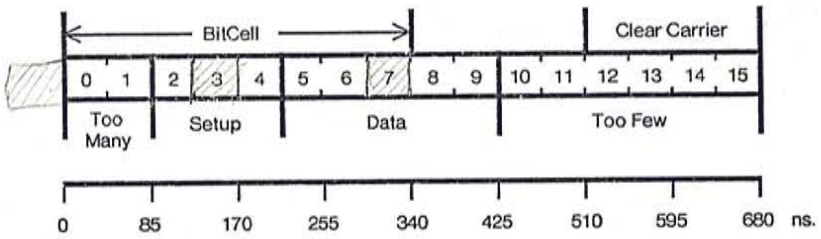
XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Phase Encoder Timing	Ether21.sil	David Boggs	Ce	7/08/79	42



A bitcell is nominally 340 ns.

A sample is nominally 42.5 ns.

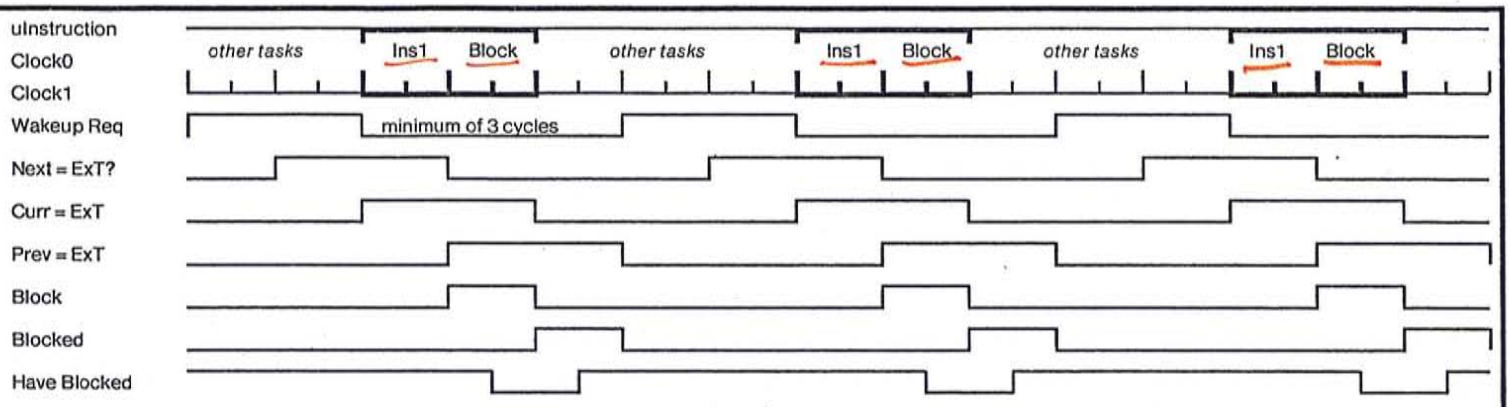
2.21 Mb/s
3.68 Mb/s



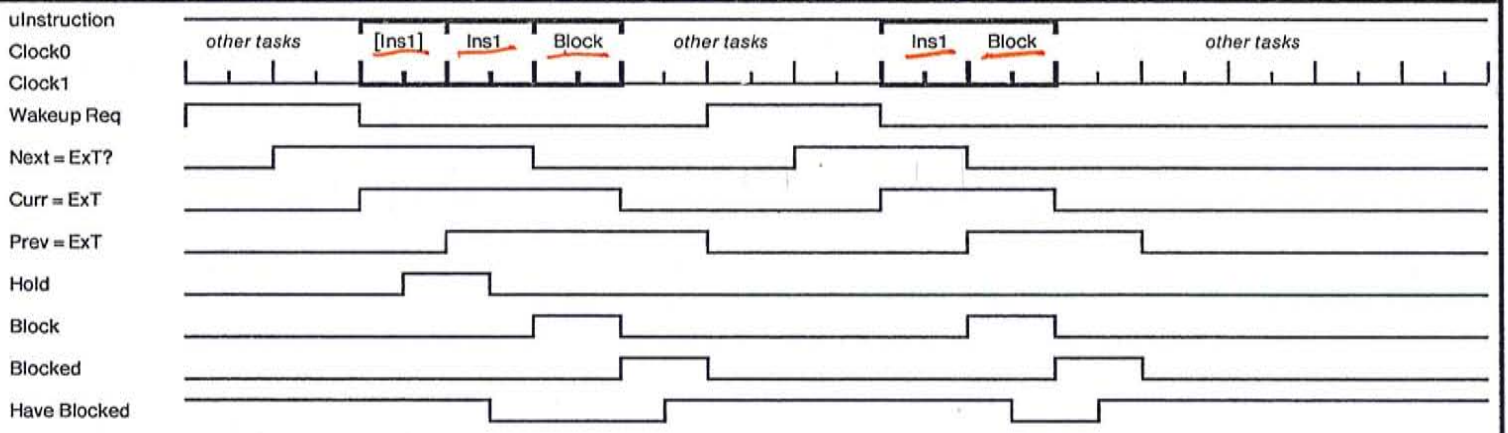
Inputs				Outputs			Comment
Carrier	Old	New	Cnt	Carrier	Event	CntCtrl	
Low	Low	Low	d/c	Low	NoEvent	Count *	Idle
Low	Low	High	d/c	High	One	Reset	Start of packet (start bit)
Low	High	Low	d/c	High	Collision	Reset *	Impossible
Low	High	High	d/c	Low *	NoEvent *	Count *	Impossible
High	Low	High	0-1	High	Collision	Count	Too many transitions
High	High	Low	0-1	High	Collision	Count	Too many transitions
High	Low	High	2-4	High	NoEvent	Count	Setup transition (zero next)
High	High	Low	2-4	High	NoEvent	Count	Setup transition (one next)
High	Low	High	5-9	High	One	Reset	Data transition
High	High	Low	5-9	High	Zero	Reset	Data transition
High	Low	High	10-15	High	Collision	Reset *	Too few transitions
High	High	Low	10-15	High	Collision	Reset *	Too few transitions
High	Low	Low	0-11	High	NoEvent	Count	Active
High	High	High	0-11	High	NoEvent	Count	Active
High	Low	Low	12-15	Low	Collision *	Reset	End of packet
High	High	High	12-15	High	Collision	Reset	Jam

"Impossible" conditions can happen right after power up.

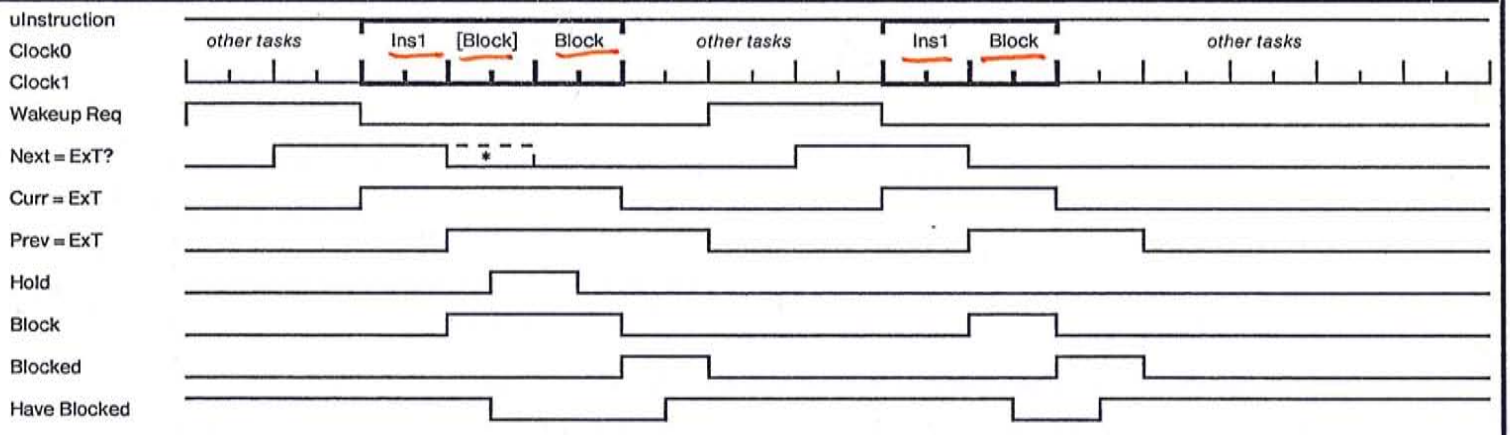
d/c means "don't care".



Normal case

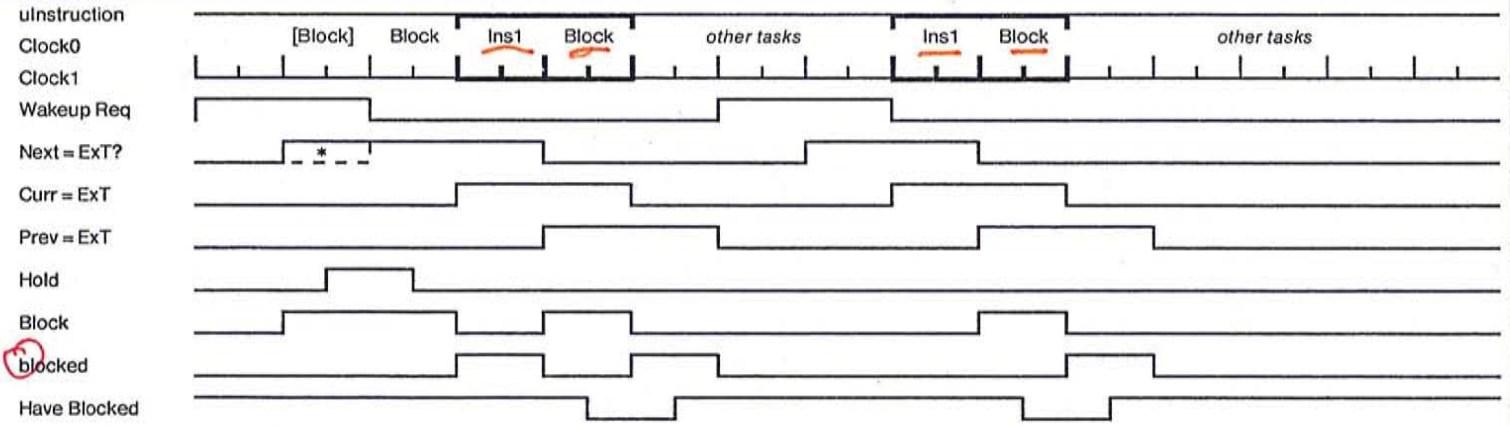


Hold in first instruction & not preempted



* NEXT bus is wrong

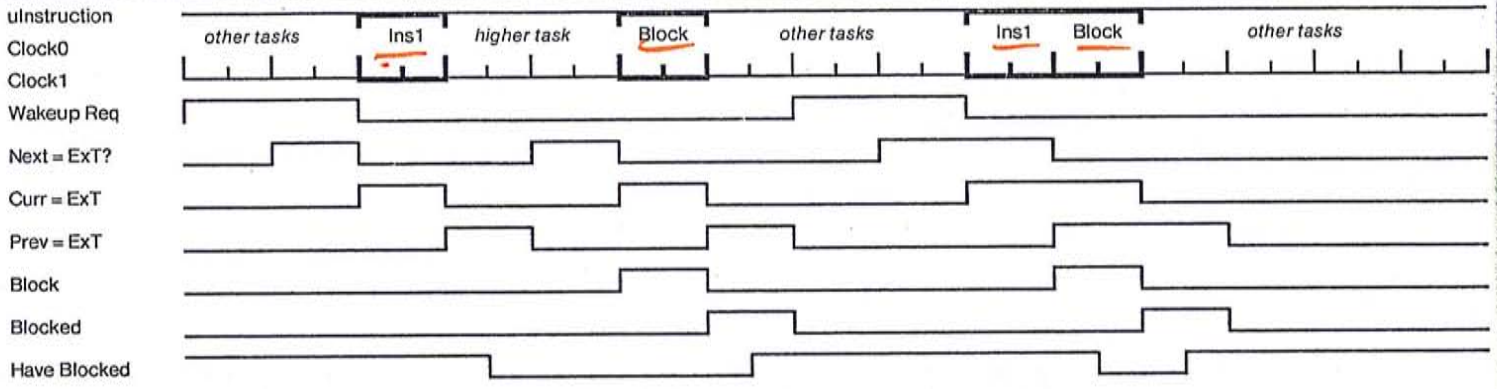
Block & Hold & not preempted



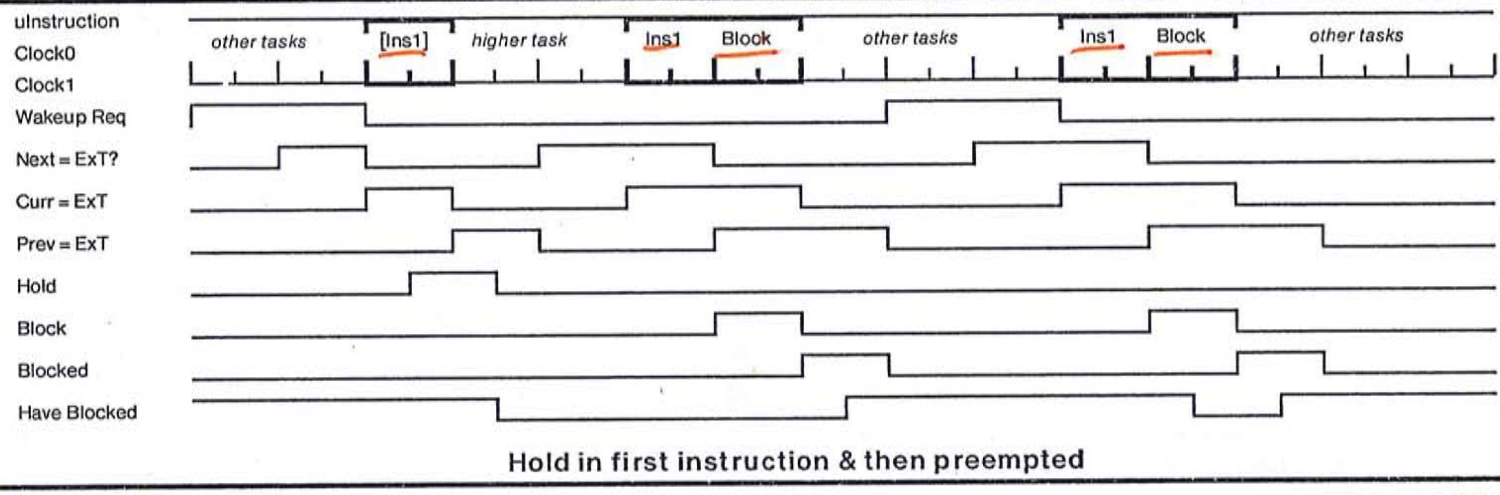
* NEXT bus is wrong

Block & Hold & (BNT = ExT < CTask)

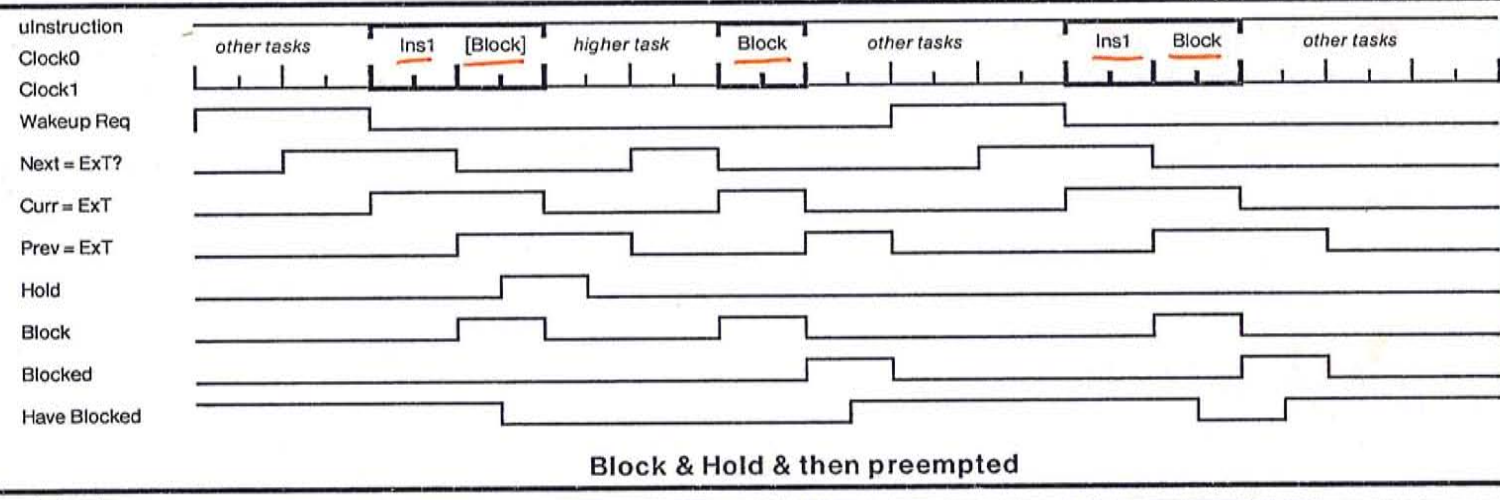
XEROX PARC	Project: Dorado	Reference: Wakeup Timing	File: Ether23.sil	Designer: David Boggs	Rev: Ce	Date: 7/08/79	Page: 44



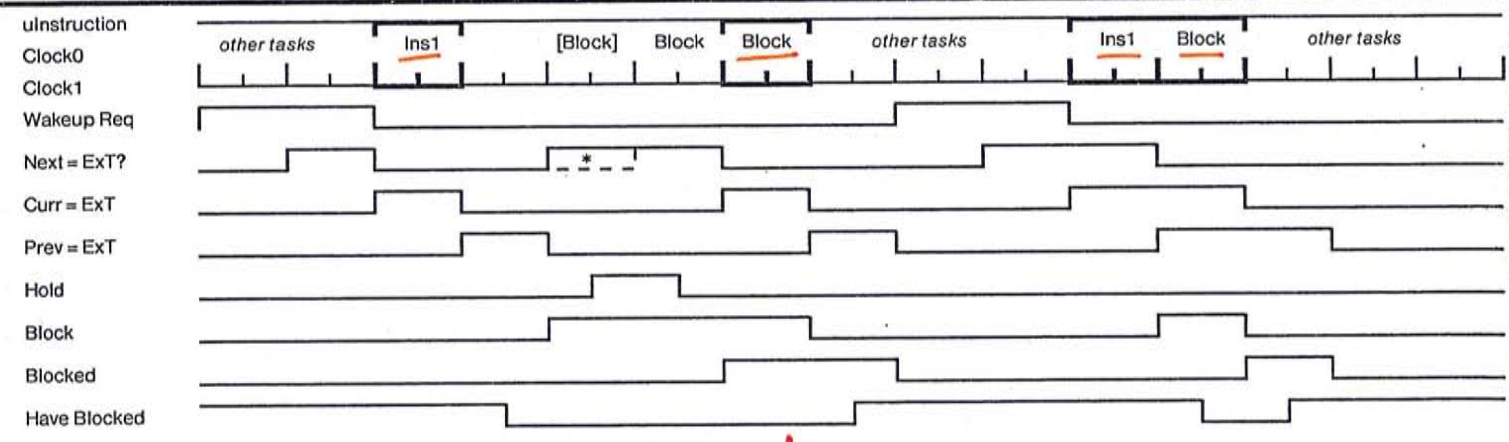
Preemption after first instruction



Hold in first instruction & then preempted



Block & Hold & then preempted



* NEXT bus is wrong

Block & Hold & (BNT = ExT < CTask) while preempted

XEROX PARC	Project	Reference	File	Designer	Rev	Date	Page
	Dorado	Wakeup Timing	Ether24.sil	David Boggs	Ce	7/08/79	45

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
Tx Cmd Enbl'	Tx On	Tx EOP	Tx Cnt Dwn	Rx Cmd Enbl'	Rx On	Rx BOP'		Test Cmd Enbl'	Loop Back	Single Step	No Wake ups	Test Clock	Test Coll'	Test Data	Report Colls

Output

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
MSB	Host Address						LSB	Rx On	Tx On	Loop Back	Tx Coll	No Wake ups	Tx Data Late	Single Step	Tx Fifo PE
028	029	032	033	036	037	040	041								
200	100	40	20	10	4	2	1								

Input

The host address is set by jumpers on the right backplane.
To set a bit to one, pull it up to ground through 91 ohms.

RCVR STATUS

```
// EtherRoms.bcpl -- Model 1
// Last modified January 10, 1980  12:54 AM by Boggs

get "DoradoProms.defs"

external EtherProms

//-------------------------------------------------------------------------
let EtherProms(mem) be
//-------------------------------------------------------------------------
[
let buff = vec 300
if StEq(mem, "DskEth") then mem!0 = DoAll

if StEq(mem, "EtherFifo") & ICtype eq MC10149 then  // define the Fifo Prom
   [
   for adr = 0 to 255 do buff!adr = EtherFifo(adr)
   Header("EtherFifo", 4, buff, 256, 0)
   PromCommand("Eth-l10")
   Header("EtherFifo", 4, buff, 256, 0)  // the second one is identical
   PromCommand("Eth-l15")
   ]

if StEq(mem, "EtherPD") & ICtype eq MC10149 then  // define the PD Prom
   [
   for adr = 0 to 255 do buff!adr = EtherPD(adr)
   Header("EtherPD", 4, buff, 256, 0)
   PromCommand("Eth-h22")
   ]

if StEq(mem, "EtherRcvr") & ICtype eq MC10149 then  // define the Rcvr Prom
   [
   for adr = 0 to 255 do buff!adr = EtherRcvr(adr)
   Header("EtherRcvr", 12, buff, 256, 0)
   PromCommand("Eth-h09") // command to blow the left nibble
   PromCommand("Eth-h10", "4")  // command to blow the middle nibble
   PromCommand("Eth-h11", "8")  // command to blow the right nibble
   ]

if StEq(mem, "EtherXmtr") & ICtype eq MC10149 then  // define the Xmtr Prom
   [
   for adr = 0 to 255 do buff!adr = EtherXmtr(adr)
   Header("EtherXmtr", 12, buff, 256, 0)
   PromCommand("Eth-h14") // command to blow the left nibble
   PromCommand("Eth-h15", "4")  // command to blow the middle nibble
   PromCommand("Eth-h16", "8")  // command to blow the right nibble
   ]
]
```

```
//-------------------------------------------------------------------------
and EtherFifo(input) = valof
//-------------------------------------------------------------------------
// An MCM10150 or MCM10149 (either type will work)
// The transmitter fifo and the receiver fifo each use one of these proms.
[
manifest
   [
   // The correspondence between voltage levels and bits in the .MB file:
   high = 1; low = 0
   ]

structure Input:
   [
   blank bit 8
   write bit 4          // msb is A0 pin 4
   read bit 4           // msb is A4 pin 10
   ]

structure Output:
   [
   full bit             // Q0 pin 15
   notFull bit
   empty bit
   notEmpty bit
   blank bit 12
   ]

let write = input<<Input.write
let read = input<<Input.read

let output = nil
output<<Output.full = ((read-1) & 17b) eq write? high, low
output<<Output.notFull = ((read-1) & 17b) eq write? low, high
output<<Output.empty = read eq write? high, low
output<<Output.notEmpty = read eq write? low, high
resultis output
]
```

```
//-------------------------------------------------------------------------
and EtherPD(input) = valof
//-------------------------------------------------------------------------
// An MCM10150 or MCM10149 (either type will work)
[
manifest
   [
   // The correspondence between voltage levels and bits in the .MB file:
   high = 1; low = 0
   // pdEvent values
   noEvent = low lshift 1 + low
   collision = low lshift 1 + high
   dataZero = high lshift 1 + low
   dataOne = high lshift 1 + high
   // cntCntrl values
   count = high; load = low
   ]

structure Input:
   [
   blank bit 8
   pdCarrier bit          // A0 pin 4
   newData bit
   oldData bit
   timer bit 4            // msb is A3 pin 9
   reportCollisions bit
   ]

structure Output:
   [
   carrier bit            // Q0 pin 15
   event bit 2
   cntCtrl bit
   blank bit 12
   ]

let pdCarrier = input<<Input.pdCarrier ne 0
let newData = input<<Input.newData ne 0
let oldData = input<<Input.oldData ne 0
let timer = input<<Input.timer
let reportCollisions = input<<Input.reportCollisions ne 0

let carrier = pdCarrier
let event = noEvent
let cntCtrl = count
```

// EtherPD (cont'd)

```
test pdCarrier
   ifso test oldData ne newData
      ifso switchon timer into
         [
         case 0 to 1:  // too many transitions
            [
            if reportCollisions then event = collision
            endcase
            ]
         case 2 to 4: endcase  // setup transition
         case 5 to 9:  // data transition
            [
            cntCtrl = load
            event = newData? dataOne, dataZero
            endcase
            ]
         case 10 to 15:  // too few transitions
            [
            cntCtrl = load
            event = reportCollisions? collision, newData? dataOne, dataZero
            endcase
            ]
         ]
      ifnot if timer ge 12 then  // jam or end of packet
         [
         carrier = newData? high, low
         cntCtrl = load
         ]
   ifnot if oldData ne newData then  // first transition of new packet
      [
      cntCtrl = load
      carrier = high
      event = newData? dataOne, collision
      ]

let output = nil
output<<Output.carrier = carrier
output<<Output.event = event
output<<Output.cntCtrl = cntCtrl
resultis output
]
```

```
//-------------------------------------------------------------------------
and EtherRcvr(input) = valof
//-------------------------------------------------------------------------
// Three MCM10150s or MCM10149s (either type will work)
[
manifest
   [
   // The correspondence between voltage levels and bits in the .MB file:
   high = 1; low = 0
   // srCtrl values
   srLoad = low lshift 1 + low
   srShift = high lshift 1 + low  // shift left, count down
   srHold = high lshift 1 + high
   // state values
   idle = 0; maybe = 1; full = 2; imip = 3
   // pdEvent values
   noEvent = 0; collision = 1; dataZero = 2; dataOne = 3
   ]

structure Input:
   [
   blank bit 8
   currentState bit 3   // rxState.0 is A0 pin 4
   rxCollision bit
   pdCarrier bit
   pdEvent bit 2        // pdEvent.0 is A5 pin 6
   rxSRFull bit         // *** Low True ***
   ]

structure Output:
   [
   nextState bit 3      // rxState.0 is Q0 pin 15
   rxCollision bit
   rxEOP bit            // Q0 pin 15
   rxSync bit           // *** Low True ***
   rxIncTrans bit
   rxCRCReset bit
   rxCRCClk bit         // Q0 pin 15
   rxData bit
   rxSRCtrl bit 2       // rxSRCtrl.0 is Q2 pin 12
   blank bit 4
   ]

let currentState = input<<Input.currentState
let rxCollision = input<<Input.rxCollision ne 0
let pdCarrier = input<<Input.pdCarrier ne 0
let pdEvent = input<<Input.pdEvent
let rxSRFull = input<<Input.rxSRFull eq 0

//rxCollision, rxIncTrans, rxEOP, and rxSync are treated as booleans and
// converted to the correct voltage level at the end of this procedure.
//rxCRCReset, rxCRCClk and rxData are treated as voltage levels throughout.

let nextState = currentState
let preCollision = rxCollision
let rxEOP = false
let rxSync = false
let rxIncTrans = false
let rxCRCReset = low
let rxCRCClk = low
let rxData = low
let rxSRCtrl = srHold
```

// EtherRcvr (cont'd)

```
switchon currentState into
   [
   case idle:
      [
      //In this state we are waiting for the beginning of a packet.
      rxCRCReset = high
      preCollision = false
      if pdCarrier then
         [
         nextState = maybe
         switchon pdEvent into
            [
            case collision:
            case dataZero:
               [
               //Getting dataZero means we are out of sync with
               // the phase decoder.  This may happen just after power-on.
               //Getting collision means the phase decoder missed the first
               // bit of the packet (a common failure of Ethernets which
               // are too long, aggravated by the too-short preamble).
               preCollision = true
               endcase
               ]
            case dataOne:
               [
               //All Ethernet packets are preceeded by a degenerate
               // preamble consisting of a single one bit.
               //Absense of carrier and then a one bit marks the
               // beginning of a packet.
               //The bit is clocked into the CRC register but not
               // into the receiver SR.
               rxCRCReset = low
               rxCRCClk = high
               rxData = high
               rxSRCtrl = srLoad  //really to init the counter
               endcase
               ]
            ]
         ]
      endcase
      ]
```

// EtherRcvr (cont'd)

```
case maybe:
   [
   //In this state we are in the first part of a packet.  We have
   // not put anything into the Fifo, so if things go sour, we can
   // just go back to the idle state without having to notify people
   // down the pipe.
   //Collisions will generally cause very short (less then 16 bit)
   // packets or at least packets with phase encoding violations
   // (too many or too few transitions).
   //If carrier drops and the phase decoder has reported a collision,
   // then its reportCollisions input must be true, which implies that the
   // microcode wants to see collisions, so report status for the fragment.
   //If carrier drops but the PD hasn't reported a collision, then
   // reportCollisions is probably false, so just go back to the
   // idle state without putting anything into the pipe.
   test pdCarrier
      ifnot
         [
         // We lost carrier.
         // This was probably a runt packet caused by a collision.
         // Note that we discard the data. overwrite the data in RSR with STATUS
         if rxCollision then
            [
            rxSRCtrl = srLoad
            rxIncTrans = true
            rxEOP = true
            rxSync = true
            ]
         nextState = idle
         ]
      ifso switchon pdEvent into
         [
         case collision:
            [
            preCollision = true
            endcase
            ]
         case dataZero:
         case dataOne:
            [
            rxSRCtrl = srShift
            rxCRCClk = high
            rxData = pdEvent eq dataOne? high, low
            if rxSRFull then
               [
               //The receiver SR is full - dump it into the Fifo.
               nextState = full
               rxSync = true
               ]
            endcase
            ]
         ]
   endcase
   ]
```

// EtherRcvr (cont'd)

```
case full:
   [
   //In this state we are at the end of a word.
   //When a packet ends, the phase decoder will drop carrier.
   //If this happens when the shift register is full,
   // then we assume that it is the normal end of a packet
   // (if it isn't the CRC will probably be wrong).
   //If carrier drops when the shift register is not full (states
   // IMIP or Maybe), then the packet is damaged so the incomplete
   // transmission status bit is set.
   test pdCarrier
      ifnot  //carrier dropped
         [
         nextState = idle
         rxSRCtrl = srLoad
         rxIncTrans = false
         rxEOP = true
         rxSync = true
         ]
      ifso switchon pdEvent into
         [
         case collision:
            [
            preCollision = true
            endcase
            ]
         case dataZero:
         case dataOne:
            [
            rxSRCtrl = srShift
            rxCRCClk = high
            rxData = pdEvent eq dataOne? high, low
            nextState = imip
            endcase
            ]
         ]
   endcase
   ]
```

// EtherRcvr (cont'd)

```
case imip:
   [
   //In this state, we are in the middle of a word
   test pdCarrier
      ifnot  //carrier dropped
         [
         nextState = idle
         rxSRCtrl = srLoad
         rxIncTrans = true
         rxEOP = true
         rxSync = true
         ]
      ifso switchon pdEvent into
         [
         case collision:
            [
            preCollision = true
            endcase
            ]
         case dataZero:
         case dataOne:
            [
            rxSRCtrl = srShift
            rxCRCClk = high
            rxData = pdEvent eq dataOne? high, low
            if rxSRFull then
               [
               //The receiver SR is full - dump it into the Fifo.
               nextState = full
               rxSync = true
               ]
            endcase
            ]
         ]
      endcase
      ]
   default:     //unused states go to idle
      [
      nextState = idle
      endcase
      ]
   ]

let output = nil
output<<Output.nextState = nextState
output<<Output.rxCollision = preCollision? high, low
output<<Output.rxEOP = rxEOP? high, low
output<<Output.rxSync = rxSync? low, high       //low true
output<<Output.rxIncTrans = rxIncTrans? high, low
output<<Output.rxCRCReset = rxCRCReset
output<<Output.rxCRCClk = rxCRCClk
output<<Output.rxData = rxData
output<<Output.rxSRCtrl = rxSRCtrl
resultis output
]
```

```
//-------------------------------------------------------------------------
and EtherXmtr(input) = valof
//-------------------------------------------------------------------------
//Three MCM10150s or MCM10149s (either type will work)
[
manifest
   [
   //the correspondence between voltage levels and bits in the .MB file:
   high = 1; low = 0
   //srCtrl values
   srLoad = low lshift 1 + low
   srShift = high lshift 1 + low  //shift left, count down
   srHold = high lshift 1 + high
   //state values
   idle = 0; mark = 1; data = 2; preCRC = 3; crc = 4; postCRC = 5
   ]

structure Input:
   [
   blank bit 8
   currentState bit 3   //TxState.0 is A0 pin 4
   gotTxBit bit
   txStop bit           //TxEOP & TxFifoEmpty
   txAbort bit          //TxCollision % TxDataLate % TxFifoPE % TxOff **Low
   txStart bit          //(TxFifoFull % TxEOP) & PDCarrier'
   txSREmpty bit        //*** Low True ***
   ]

structure Output:
   [
   nextState bit 3      //preTxState.0 is Q0 pin 15
   txCRCEnbl bit
   txCRCClk bit //Q0 pin 15
   txGone bit
   txGo bit
   txData bit
   txSRCtrl bit 2       //txSRCtrl.0 is Q0 pin 15
   spare bit 2
   blank bit 4
   ]

let currentState = input<<Input.currentState
let gotTxBit = input<<Input.gotTxBit ne 0
let txStop = input<<Input.txStop ne 0
let txAbort = input<<Input.txAbort eq 0 //*** low true
let txStart = input<<Input.txStart ne 0
let txSREmpty = input<<Input.txSREmpty eq 0  //*** low true

//txGone and txGo are treated as booleans
//txCRCEnbl, txCRCClk, and txData are treated as voltages

let nextState = currentState
let txCRCEnbl = low
let txCRCClk = low
let txGone = txAbort
let txGo = false
let txData = low
let txSRCtrl = srHold
```

// EtherXMTR (cont'd)

```
test txAbort
   ifso nextState = idle
   ifnot switchon currentState into
      [
      case idle:
         [
         //In this state the transmitter is shut down.
         if txStart then
            [
            //The Fifo is full or as full as it is going to get
            // (i.e. the packet is < the length of the Fifo), and there
            // is no carrier present so start transmitting.
            //A one bit (the 'mark' bit) is inserted before the first
            // user data bit to allow the receiver to aquire bit phase.
            nextState = mark
            txGo = true          //enable the phase encoder
            ]
         endcase
         ]
      case mark:
         [
         //In this state we are wire ORing a one onto the output of the
         // shift register to form the mark bit, and waiting for the
         // phase encoder to acknowledge it.
         txGo = true
         txData = high
         if gotTxBit then
            [
            //The Phase encoder has acked the mark bit.
            nextState = data
            txCRCClk = high
            txSRCtrl = srLoad
            ]
         endcase
         ]
      case data:
         [
         //In this state we are sending user data to the phase encoder.
         //Note: txCRCClk and txSRCtrl are set several times.
         txGo = true
         if gotTxBit then
            [
            //The phase encoder has acked a bit.  Clock the bit into
            // the CRC and shift the next bit into place.
            txCRCClk = high
            txSRCtrl = srShift
            if txSREmpty then
               [
               //That bit was the last one in the register.
               //Load instead of shifting.
               txSRCtrl = srLoad
               if txStop then
                  [
                  //That was the last bit of the last word of user data.
                  //Supply the CRC output to the phase encoder.
                  //We DO want to shift TxSR so that a zero comes out
                  // because the CRC output is wire ORed to it.
                  txSRCtrl = srShift
                  nextState = preCRC
                  ]
               ]
            ]
         endcase
         ]
```

```
// EtherXMTR (cont'd)
      case preCRC:
         [
         //We stay in this state for one Dorado cycle.
         //The purpose of this state is to predecrement the counter so that
         // we are notified of the end of the packet one bit time early.
         //This will also shift the shift register, but that's unimportant
         // since we are supplying the CRC output to the phase encoder.
         //However it is important that the shift register produce zeros
         // whenever we shift it since the CRC output is wire ORed to it.
         //Fortunately this is the case.
         txGo = true
         txSRCtrl = srShift
         txCRCEnbl = high
         nextState = crc
         endcase
         ]
      case crc:
         [
         //In this state we are sending CRC data to the phase encoder.
         txGo = true
         txCRCEnbl = high
         if gotTxBit then
            [
            //The phase encoder has acked another CRC bit.
            txCRCClk = high
            txSRCtrl = srShift
            if txSREmpty then
               [
               //That was an ack for the next to last CRC bit.
               txGo = false
               nextState = postCRC
               ]
            ]
         endcase
         ]
      case postCRC:
         [
         //In this state we are waiting for the phase encoder to acknowledge
         // that it has read the last bit of the CRC.
         txCRCEnbl = high
         if gotTxBit then
            [
            //As we transition to the idle state, generate txGone
            // for one cycle.  This will clear txEOP and wakeup the
            // microcode for the last time.
            txGone = true
            nextState = idle
            ]
         endcase
         ]
      default:
         [
         nextState = idle
         endcase
         ]
      ]

let output = nil
output<<Output.nextState = nextState
output<<Output.txCRCEnbl = txCRCEnbl
output<<Output.txCRCClk = txCRCClk
output<<Output.txGone = txGone? high, low
output<<Output.txGo = txGo? high, low
output<<Output.txData = txData
output<<Output.txSRCtrl = txSRCtrl
resultis output
]
```

```
*Dorado microcode emulating Alto Ethernet

* Last modified December 14, 1979  5:19 PM by Taft

TITLE[D1AltoEther.Mc];

*Hardware definitions

*TIOA assignments
DEVICE[EDATA, 15];      *Input and output data
DEVICE[ECONTROL, 16];   *Control and status

*Receiver status bits in end-of-packet word (EDATA).
*Assignments are Alto-compatible (except ESICOLL, which the Alto doesn't have)
MC[ESICOLL, 200];       *Receiver-detected collision
MC[ESIDL, 40];          *Input data late
MC[ESCRC, 10];          *CRC bad
MC[ESICMD, 4];          *Input command issued   ** Not in hardware:        **
MC[ESOCMD, 2];          *Output command issued  ** for Alto emulation only **
MC[ESIT, 1];            *Incorrectly terminated packet

*Transmitter and general interface status bits (ECONTROL)
*Left byte is Ethernet host address.
MC[ESION, 200];         *Receiver on
MC[ESOON, 100];         *Transmitter on
MC[ESLOOP, 40];         *Loop-back mode on
MC[ESOCOLL, 20];        *Transmitter-detected collision
MC[ESWAKE', 10];        *Task wakeups enabled if 0, disabled if 1
MC[ESODL, 4];           *Output data late
MC[ESSTEP, 2];          *Single-step mode on
MC[ESOPE, 1];           *Transmitter-detected IOB or Fifo parity error

*Status masks useful for Alto emulation
MC[EISMASK, ESICOLL, ESIDL, ESCRC, ESIT]; *Bits reported for input command
MC[EOSMASK, ESODL, ESOCOLL, ESOPE]; *Bits reported for output command
MC[ECMDBITS, ESICMD, ESOCMD]; *Command bits

*Control bits (ECONTROL) -- Transmitter control (bits 0-3)
MC[ECOENB, 007777];     *Bit 0 = 0 enables decoding of output commands
MC[ECOON, 40000];       *Turn on output if 1, off if 0
MC[ECOEND, 20000];      *Send end of packet if 1
MC[ECOCWAK, 10000];     *Generate countdown wakeup if 1

MC[EOTURNOFF, ECOENB];
MC[EOTURNON, ECOENB, ECOON];
MC[ECOUNTWAKE, ECOENB, ECOON, ECOCWAK];
MC[ESENDEND, ECOENB, ECOON, ECOEND];

*Control bits (ECONTROL) -- Receiver control (bits 4-7)
MC[ECIENB, 170377];     *Bit 4 = 0 enables decoding of input commands
MC[ECION, 2000];        *Turn on input if 1, off if 0
MC[ECIBOP', 1000];      *Wait for beginning of next packet if 0

MC[EITURNON, ECIENB, ECION, ECIBOP'];
MC[EITURNOFF, ECIENB];
MC[EIWAITBOP, ECIENB, ECION];

*Control bits (ECONTROL) -- Test control (bits 8-15)
MC[ECTENB, 177400];     *Bit 8 = 0 enables decoding of test commands
MC[ECLOOP, 100];        *Turn on loop-back mode if 1, off if 0
MC[ECSTEP, 40];         *Turn on single-step mode if 1, off if 0
MC[ECWAKE', 20];        *Enable task wakeups if 0, disable if 1
MC[ECCLOCK, 10];        *Generate one EtherClk if 1 (ECSTEP must be 1)
MC[ECCOLL', 4];         *Cause a collision indication if 0
MC[ECDATA, 2];          *Set flop that ORs a 1 into the received data if 1
MC[ECRCOL, 1];          *Report receiver detected collisions

*Command to reset the entire interface
MC[ERESETALL, ECCOLL']; *Enable output, input, test; ECCOLL' is low-true

% -- Defined in ADefs.mc --
TASKN[EOT, 6];          *Output task
TASKN[EIT, 7];          *Input task
%
```

```
*R-registers
************

*In the Alto emulation, we could assume that the input and output tasks
*never run at the same time and can share R-registers, since the emulated
*Alto interface is half-duplex.  However, when the microcode is rewritten
*to run input and output independently, this won't be possible.
*Therefore, we are staking out independent R-registers at this time.

*The input and output R-registers have to be in separate banks because
*the two tasks share task-independent subroutines that need to reference
*task-specific registers and temporaries.  However, we don't need very
*many R-registers in either bank, so these banks could be shared with
*other tasks.

% -- Defined in ADefs.mc --
RMREGION[EIREGS];       *Used by input task
RVN[EIPTR];             *Input main loop pointer/count
RVN[EITEMP1];           *Input temporaries
RVN[EITEMP2];

RMREGION[EOREGS];       *Used by output task
RVN[EOPTR];             *Output main loop pointer/count
RVN[EOTEMP1];           *Output temporary
RVN[EOTime];            *Time at end of last packet successfully transmitted
RVN[RNUM];              *State for random number generator
RV[RCONST, 33031];      *Constant (13849) for random number generator
RVN[RTCLOCK];           *Real-time clock, updated by display task
RVN[EOTime];            *Time at end of last packet successfully transmitted
%

*Region-independent R-register definitions
RVREL[EXPTR, AND[IP[EIPTR], 17]];
RVREL[EXTEMP1, AND[IP[EITEMP1], 17]];


*Memory base registers
**********************

% -- Defined in ADefs.mc --
BR[EIBR, 32];           *Input base register
BR[EOBR, 33];           *Output base register
BR[ECBR, 34];           *Command base register  {EmuBrHi,,600}
%

*The emulator defines AemBR0 to be a BR that always contains 0
* relative to the Alto emulator's virtual space.


*Control block addresses (for Alto emulation, relative to ECBR = 600)
** use fetch/store from small constant feature of Model1
************************

msc[EPLOC, 0];          *Post location
msc[EBLOC, 1];          *Interrupt bit mask
msc[EELOC, 2];          *Ending word count
msc[ELLOC, 3];          *Load mask
msc[EICLOC, 4];         *Input count
msc[EIPLOC, 5];         *Input pointer
msc[EOCLOC, 6];         *Output count
msc[EOPLOC, 7];         *Output pointer
msc[EHLOC, 10];         *Host address


*Microcode post codes (small integer in lh, ones in rh for XOR).
**********************

MC[ESIDON, 377];        *Input done
MC[ESODON, 777];        *Output done
MC[ESIFUL, 1377];       *Input buffer overflowed
MC[ESLOAD, 1777];       *Load overflow
MC[ESCZER, 2377];       *Word count zero in input or output command
MC[ESABRT, 2777];       *Command aborted (by SIO)
```

```
* Minimum inter-packet spacing for transmitter, in units of 9.5 usec.
MC[MinPktSpacing, 64];  * ~ 500 microseconds
```

```
*Note on instruction placement:
*Emulator and input task code are in separate pages (though neither
*one completely fills up its page, so could share with other code).
*Output task and subroutine code fit together on one page.

*This means that the emulator and input tasks cannot use FF in
*instructions that call common subroutines.  Also, calls to ERESI and
*ERESO must leave FF free.

TOPLEVEL;

*Initialization code
*********************

******
** Initialization code for Output Task
        Set[XTask, IP[EOT]];                    * not really needed
EOInit: nop,                        TaskInitAddr[EOT];
        RBASE← RBASE[EOREGS];               * Init random number generator
        T← 33000C;
        RCONST← T + (31C);          * Constant (13849) for random number generator
        RNUM← RTCLOCK, BRANCH[EXINIT];

******
** Initialization code for Input Task
        Set[XTask, IP[EIT]];
EIInit: nop,                        TaskInitAddr[EIT];
        MemBase← ECBR;
        T← 300C;
        T← T+T, RBase← RBase[AEmRegs];
        BrHi← EmuBrHiReg;
        BrLo← T;                            * ECBR← {EmuBrHiVal,,600}
ExInit: T← ECONTROL;
        TIOA← T, block;                     * full TIOA for initialization
        BRANCH[.], BREAKPOINT;              * Should never get here


* Subroutine to reset the Ethernet hardware (input, output, and test)
SUBROUTINE;

RESETETHER:
        T← ECONTROL;
        TIOA← T;
        T← ERESETALL;
        OUTPUT← T, RETURN;

TOPLEVEL;
```

```
*Emulator Task -- SIO instruction
**************

SET[XTask, ip[EMU]];            *Emulator mode

SET[SIOX, 3074];        *XXX0 or XXX4, for 4-way dispatch
KNOWRBASE[AEmRegs];

*Get here with AEmRegs selected and the SIO control bits in Top-of-Stack.
*The emulator branches here at the end of SIO emulation after
*first processing control bits for any other devices (e.g., Trident).
ESIO:   T← ECONTROL;
        Stack← (Stack) AND (3C);        *Mask Ethernet control bits
        TIOA← T, BRANCH[SIONOP, ALU=0]; *Select control register

*Control bits nonzero.  Must first reset the hardware and tasks.
*Addressing constraints require one instruction between the conditional
*BRANCH (above) and the CALL, and the CALLs must leave FF free.
        TASKINGOFF;
        CALL[ERESI];                    *Reset input
        CALL[ERESO];                    *Reset output
        BDISPATCH← Stack;               *Dispatch on function bits
        TASKINGON, BRANCH[SIO0];

*00 -- Dispatch can't happen
SIO0:   BREAKPOINT, AT[SIOX, 0];

*01 -- Start transmitter
SIO1:   T← EOTURNON, AT[SIOX, 1];       *Enable output wakeups
        OUTPUT← T, BRANCH[SIONOP];

*10 -- Start receiver
SIO2:   T← EITURNON, BRANCH[.-1], AT[SIOX, 2]; *Enable input wakeups

*11 -- Reset interface
SIO3:   T← ECMDBITS, AT[SIOX, 3];       *Manufacture "abort" status
        T← (T) XOR (ESABRT);
        CALL[EPOST];                    *Post it (must leave FF free)
        MEMBASE← AemBRO;                *Back to emulator MemBase

*00 -- Noop, just return Ethernet address.
*Note that this is the tail of the other 3 cases.
SIONOP: T← InputNoPE;                   *Read status
        PD← T, Link← ETemp;             *Prepare to return from doSIO
Subroutine;
        T← RSH[T, 10], BRANCH[.+2, ALU=0]; *Right-justify Ethernet address
        Stack← (T) OR (77400C), Return; *177 .. Ethernet address

* Apparently no Ethernet interface installed.
* Alto software expects to see 77777 in this case.
        Stack← 77777C, Return;

TopLevel;
```

```
*Ethernet Boot
**************

*The emulator branches here after doing all other initialization.
*When the Breath of Life packet has been received, we branch to
*the emulator main loop.

KNOWRBASE[AEmRegs];

EBOOT:  T← A0, MEMBASE← ECBR;           *Address page 1
        STORE← EBLOC, DBuf← T;          *EBLOC← 0 (interrupt bit mask)
        ETemp← (STORE← EOCLOC)-1, DBuf← T; *EOCLOC← 0 (output count)
        ETemp← (STORE← ETemp)-1, DBuf← 1C; *EIPLOC← 1 (input pointer)
        STORE← ETemp, DBuf← 400C;       *EICLOC← 400 (input count)
        ETemp← EHLOC;
        STORE← ETemp, DBuf← 377C;       *EHLOC← 377 (host address)
        T← ECONTROL;
        TIOA← T;
        TASKINGOFF;                     *Reset input hardware and microcode
        CALL[ERESI];
        TASKINGON;

*Loop to await Breath of Life packet.
*Turn on receiver, wait for a packet to arrive.  If an OK receiver status
*is posted and the packet type is Breath of Life, set the PC to 3 and
*start the emulator.
EBLOOP: ETemp← A0, MEMBASE← ECBR;       *Address page 1
EBLOO1: T← (STORE← EPLOC), DBuf← ETemp; *EPLOC← 0 (post location)
        FETCH← T, T← EITURNON;          *Fetch to prime MD for wait loop
        OUTPUT← T;                      *Turn on receiver

        FETCH← EPLOC, T← (PD← MD);      *Wait for EPLOC to go nonzero
        T← (T)-(377C), BRANCH[.-1, ALU=0]; *Test for OK receiver status
        T← 202C, BRANCH[EBLOO1, ALU#0];
        MEMBASE← AemBR0;                *Address page 0
        FETCH← 2S;                      *Fetch word 2 (packet type)
        T← (T)+(400C);                  *T← 602 = typeBreathOfLife
        PD← (T)-(MD);
        BRANCH[EBLOOP, ALU#0];          *Loop if type is wrong
        T← 3C, BRANCH[Start];           *Start emulator at location 3
```

```
*Input Task
***********

        SET[XTask, IP[EIT]];            *Non-emulator mode
        KNOWRBASE[EIREGS];

*Subroutine called to reset input hardware and task PC to idle state.
*Call at ERESI.  Assumes TIOA selects Ethernet control register.
*Tasking must be turned off by caller.
ERESI1: LDTPC← EIT;                     *Load input TPC from LINK
ERESI2: LINK← T, BRANCH[EIRETN];        *Restore caller's PC and return

SUBROUTINE;
ERESI:  T← EITURNOFF;                   *Turn off input hardware
        OUTPUT← T;
TOPLEVEL;
        T← Link, CALL[ERESI1];          *LINK← EIIDLE, save caller's PC


*Idle state of the Ethernet input task.
*Wake up here when first word of a new packet arrives.
EIIDLE: MEMBASE← ECBR;                  *Control block base register
        RBASE← RBASE[EIREGS];
        TIOA[EDATA];                    *Select data I/O address
        EITEMP2← INPUT, BRANCH[EIPLZ, IOATTEN]; *Save first word of packet
        TIOA[ECONTROL];                 *Select control register

*Address filtering.
        FETCH← EHLOC, T← A0;            *Fetch local host address
        T← RCY[T, EITEMP2, 10], CALL[EADRCK]; *T← destination host, test for zero
        pd← (T)-(MD), CALL[EADRCK];     *Test for destination host = me
        pd← MD, CALL[EADRCK];           *Test for me = 0 (promiscuous)

*Packet not accepted by filter.
*Tell hardware to ignore the rest of this packet.
        T← EIWAITBOP, BRANCH[EILAST];   *Wakeup at start of next packet

*Subroutine called by address filtering logic.
*Returns if ALU#0 but falls through (i.e., accepts the packet) if ALU=0.
SUBROUTINE;
EADRCK: BRANCH[.+2, ALU=0];
EIRETN: RETURN;
TOPLEVEL;

*Packet accepted by filter.
*Reset output task if it is on (probably in retransmission wait).
        TASKINGOFF;
        CALL[ERESO];                    *Must leave FF free
        TASKINGON;

*Set up pointer and count.
        T← EIPLOC;                      *Set up EIPTR, return pointer
        CALL[EGPCNT];                   *Must leave FF free
        MEMBASE← EIBR, BRANCH[EIWCZ, ALU=0]; *Test count for zero, select BR
        BRLO← T, T← EITEMP2, BLOCK;     *Load BR, recover data word, await next
```

```
*Input task (cont'd)

*Input main loop.
*Each iteration stores the word previously input from the interface
*and inputs the next word.  This facilitates end-of-packet handling.
*EIPTR has negative of number of words remaining in the buffer.
*IOATTEN branches if the data word being read is the end-of-packet status.
        EIPTR← (STORE← EIPTR)+1, dbuf← T, BRANCH[EIEND, IOATTEN];
        T← INPUT, BLOCK, BRANCH[.-1, ALU#0];

*Get here when buffer is exactly full (EIPTR=0).
*We know that the word in T is not EOP status since we would have taken
*the IOATTEN branch if it were.  If IOATTEN is now true, then the word in T
*is the CRC (which we must discard) and the next input is the status word.
*If IOATTEN is false, the buffer has overflowed.
        T← INPUT, BRANCH[EIEND1, IOATTEN]; *Branch if end-of-packet
        T← ESIFUL, BRANCH[EIPOST];      *Input buffer full post code

*Normal end-of-packet exit from main loop.
*One extra word (the CRC) has been stored in the buffer and must not be
*included in the count.  The next input is the status word.
EIEND:  EIPTR← (1S)-(EIPTR);            *Make ending count positive, add 1
        T← INPUT;                       *Read status word
EIEND1: T← (T) AND (EISMASK);           *Mask out uninteresting status bits
        T← (T) XOR (ESIDON);            *Post input done status

*Turn off interface and go to idle state (awaiting next SIO).
EIPOST: CALL[EPOST];                    *Post status in T (must not use FF)
        T← EITURNOFF;
EILAST: OUTPUT← T, BLOCK, BRANCH[EIIDLE];

*Word count zero when interface started, post error status
EIWCZ:  T← ESCZER, BRANCH[EIPOST];      *Word count zero post code

*Here if packet length zero, i.e., first word input was end-of-packet status.
*This should happen only if receiver-detected collision reporting is turned on,
*since runt packets are ordinarily filtered out by the phase decoder.
EIPLZ:  FETCH← EICLOC, T← EITEMP2;      *Fetch input word count
        EIPTR← MD, BRANCH[EIEND1];      *Use as ending count, go report status
```

```
*Output Task
************

SET[XTASK, IP[EOT]];
KNOWRBASE[EOREGS];

*Subroutine called to reset output hardware and task PC to idle state.
*Call at ERESO.  Assumes TIOA selects Ethernet control register.
*Tasking must be turned off by caller.
ERESO1: LDTPC← EOT;                     *Load output TPC from LINK
        BRANCH[ERESI2];                 *Restore LINK from T and RETURN

SUBROUTINE;
ERESO:  T← EOTURNOFF;                   *Turn off output hardware
        OUTPUT← T;
TOPLEVEL;
        T← Link, CALL[ERESO1];          *LINK← EOIDLE, save caller's PC


*Idle state of the Ethernet output task
EOIDLE: MEMBASE← ECBR;                  *Set up MEMBASE, TIOA
        RBASE← RBASE[EOREGS];
        TIOA[ECONTROL];

*Test and update load
        EOTEMP1← (FETCH← ELLOC);        *Fetch current load
        T← MD+MD+1;                     *Load lsh 1 +1
        STORE← EOTEMP1, DBuf← T, EOTEMP1← MD, *Store new load, save current
               CALL[RANDOM];            * T← random number
        T← RSH[T, 10];                  *Use leftmost 8 bits

*Test for load overflow (old load has sign bit set).
*Then mask countdown with old load and test for nonzero result.
*Note that the countdown clock ticks every 16 us, but we want to count in
*units of 32 us, so we double the count before using it.
        EOTEMP1← ((EOTEMP1) AND T) LSH 1, BRANCH[ELODOV, R<0];
        T← EOTime, BRANCH[EWait, ALU#0]; *Branch if wait required

* On first transmission attempt, enforce minimum inter-packet spacing.
* Note: RTClock time is in units of 38 usec, but we pretend it is 32.
* Note: RTClock counts in bits [4:13]
        T← T-(RTClock)-1;               * - Time since end of last transmission
        T← T OR (170000C);              * In case of wraparound
        T← T+(MinPktSpacing);           * Compare with min permitted
        EOTemp1← (T+1) RSH 1, Branch[EOGo, Carry']; * Start now if exceeded min
        Fetch← EICLoc, Branch[.+2];     * Wait for remainder of interval

*Must wait before retransmitting.  EOTemp1 has wait time in 16-usec units.
*If the input word count is nonzero, enable the receiver while waiting.
EWait:  FETCH← EICLOC;                  *Fetch input word count
        PD← MD;
        T← EITURNON, BRANCH[.+2, ALU=0]; *Branch if no count set up
        OUTPUT← T;                      *Enable input wakeups

*Retransmission countdown wait loop.
*ECOUNTWAKE turns off data wakeups and requests a wakeup at next
*tick of countdown clock.
        T← ECOUNTWAKE;
        EOTEMP1← (EOTEMP1)-1, OUTPUT← T;
        BLOCK, BRANCH[.-1, ALU#0];

*Done waiting.  Shut off the receiver if it was turned on.
        TASKINGOFF;
        CALL[ERESI];                    *Must leave FF free
        TASKINGON;

*Set up count and pointer
EOGO:   T← EOPLOC, CALL[EGPCNT];        *Set up EOPTR, return pointer
        MEMBASE← EOBR, BRANCH[EOWCZ, ALU=0]; *Test count for zero, select BR
        BRLO← T;                        *Load low BR
```

```
*Output task (cont'd)

*Select data I/O address, then pre-fetch the first word.
        EOPTR← (FETCH← EOPTR)+1;
        BRANCH[EOEND, ALU=0];

*Output main loop.
*Data transfer is pipelined:  each iteration outputs a data word
*fetched in the previous iteration and starts the fetch of the data for
*the next iteration.  This way, cache misses generally occur while our
*task is blocked, and lower-priority tasks are permitted to run while
*the cache is being loaded.
*EOPTR is the negative of the number of words remaining to be fetched.
*IOATTEN branches if a collision, data late, or Fifo PE has aborted output.
        EOPTR← (FETCH← EOPTR)+1, T← MD, BRANCH[EOABRT, IOATTEN];
        OUTPUT← T, BLOCK, BRANCH[.-1, ALU#0];

*Next-to-last word has been output and last word has been fetched
EOEND:  OUTPUT← MD;                     *Output last word
        TIOA[ECONTROL];                 *Select control register
        T← ESENDEND;                    *Declare end of packet
        OUTPUT← T, BLOCK;

*Wake up here only when packet completely sent or collision has occurred.
*Note that EOPTR=0, the proper value to post for normal ending word count.
        T← INPUT, BRANCH[EOABR1, IOATTEN]; *Branch if aborted, read status
        EOTime← RTClock;                * Remember current time
EODONE: T← (T) AND (EOSMASK);           *Mask out uninteresting status bits
        T← (T) XOR (ESODON);            *Post output done status

*Turn off interface and go to idle state (awaiting next SIO).
EOPOST: CALL[EPOST];                    *Post status in T
        T← EOTURNOFF;
EOLAST: OUTPUT← T, BLOCK, BRANCH[EOIDLE];

*Here if collision or other error occurred.
*If a collision, just try again.  If not, post output done.
EOABRT: EOPTR← (1S)-(EOPTR);            *Make count positive, add 1
        TIOA[ECONTROL];
        T← INPUT;                       *Read status

*If not a collision, post output done with whatever status bits we got.
*The status bits posted are not actually defined in the Alto, but the software
*will notice that the hardware status is not good and will therefore retry.
EOABR1: PD← (T) AND (ESOCOLL);          *Collision?
        EOTEMP1← EOTURNOFF, BRANCH[EODone, ALU=0]; *Branch if not
        OUTPUT← EOTEMP1;                *Turn off to reset collision bit
        T← EOTURNON, BRANCH[EOLAST];    *Turn on and try again

*Word count zero when interface started, post error status
EOWCZ:  T← ESCZER, BRANCH[EOPOST];      *Word count zero post code

*Load overflow, post error status
ELODOV: T← ESLOAD, BRANCH[EOPOST];      *Load overflow post code
```

```
*Task-independent Subroutines
*****************************

SUBROUTINE;

*Post command completion.
*Expects post code and status in T and ending word count in EIPTR or EOPTR.
*Assumes RBASE is set up but makes no assumption about MEMBASE or TIOA.
*Returns with standard MEMBASE and TIOA set up.
*Clobbers T and RBase.

EPOST:  MemBase← ECBR;                      *Select BR for page 1 I/O locations
        T← (Store← EPLOC)+1, DBuf← T;       *Store ending status in EPLOC
        T← (Fetch← T)+1;                    *Fetch bit mask (know EBLOC=EPLOC+1)
        Store← T, DBuf← EXPTR, T← MD;       *Store word count (know EELOC=EBLOC+1)
        RBase← RBase[NWW];                  *T has interrupt bit mask
        NWW← (NWW) OR T, TIOA[ECONTROL];    *Or bit(s) into NWW
        Reschedule, Return;                 *Force emulator to notice


*Get input or output pointer and count.
*Expects EIPLOC or EOPLOC in T (knows that EICLOC=EIPLOC-1, EOCLOC=EOPLOC-1).
*Sets up EXPTR (EIPTR or EOPTR) and returns memory base in T (for BRLO←).
*Upon return, ALU branches are conditioned by -count (for zero test).
*Also sets TIOA← EDATA.

EGPCNT: T← (FETCH← T)-1;                    *Fetch pointer
        T← MD, FETCH← T, EXPTR← A0;         *T← pointer, fetch count

*Want to set base register such that adding 2↑16-count (treating -count
*as an unsigned number) will address the first word of the packet.
*Note that (pointer-(2↑16-count)) mod 2↑16 = pointer+count
        T← (T)+(MD);                        *T← (pointer-(2↑16-count)) mod 2↑16
        EXPTR← (EXPTR)-(MD), TIOA[EDATA], RETURN; *EXPTR← -count


*Generate random number and return it in T.
*Must have RBASE[RNUM] (=EOREGS).
*Uses algorithm:
*       R← (2↑11 + 2↑2 + 2↑0)*R + 13849

KNOWRBASE[RNUM];

RANDOM: T← LSH[RNUM, 11];                   *T← 2↑9 * R
        T← T+(RNUM);                        *(2↑9 + 2↑0)*R
        T← LSH[T, 2];                       *(2↑11 + 2↑2)*R
        T← T+(RNUM);                        *(2↑11 + 2↑2 + 2↑0)*R
        T← RNUM← T+(RCONST), RETURN;        * +13849

TOPLEVEL;
```