Dorado Debugging Interface

Dorado is controlled either by its baseboard microcomputer, called MC here, or by an Alto working through the baseboard section. This chapter describes how these control processors (CP's) interface to and operate Dorado.

An Alto computer may be connected to baseboard sections of several Dorados via the Alto's Diablo printer interface. This interface consists of a 13-bit register loaded by storing into bits 0:12 of DoradoOut = 177016_8 and a 5-bit register read by fetching from location DoradoIn = 177030_8 (I.e., on the Alto, memory references to these high-core locations are intercepted and refer to the output bus or input bus rather than to the Alto's memory).

A particular baseboard is selected by operations described below, and the others disabled. Then the Alto may operate that Dorado directly, or it may communicate with the baseboard MC, and let the MC control the Dorado.

Apart from the "select Dorado" protocol, the control interface seen by the baseboard MC and that seen by the Alto are approximately the same. In other words, the Alto can neutralize the MC and pass commands through the baseboard section directly to a Dorado mainframe; or it can enable the MC. In the latter case, the operations executed by the MC have the same general form as those executed by the Alto, though a different mechanism from DoradoIn and DoradoOut is used.

The Alto Midas subsystem is the primary software tool for debugging Dorado's. It will load, step, start, and halt programs, display registers on the Alto display, changing the contents of these on request, and test registers and memories through the debugging interface. Midas also has a simulator that checks for inconsistencies among the 1,000-odd hardware signals accessible via the DMux.

CP Output

The Alto output bus corresponds to bits 0:12 of DoradoOut, interpreted as in Figure 17. The notation "Register[Data,Strobe]" is used below to denote a CP operation in which a word is stored in DoradoOut with the value for "Register" in its 3-bit address field, "Data" in its 9-bit data field, and "Strobe" in its 1-bit Strobe field.

The 3-bit address field in DoradoOut selects one of 8 registers that will be loaded from the 9-bit data field when a strobe is generated. The CP program must explicitly generate the strobe in a 3-step sequence as shown below:

```
Addr[Data,0]
Addr[Data,1]
Addr[Data,0]     This generates the strobe, carrying out the actions of Data
```

With the exception of ShiftDAddr and the four MIR-loading control signals (discussed below), the last part of the strobe-generation sequence can be overlapped with the first part of another to the *same* address, e.g.:

```
Addr1[Data1,0]
Addr1[Data1,1]
```

Addr1[Data1,0]	This generates the strobe for Addr1, Data1
Addr1[Data2,1]	
Addr1[Data2,0]	This generates the strobe for Addr1, Data2
Addr1[Data3,1]	
Addr1[Data3,0]	Strobe--full three-step sequence before changing Addr
Addr2[Data4,0]	
Addr2[Data4,1]	etc.
. . .	

However, for ShiftDAddr and MIR loading, and for operations to a different address, the above shortcut doesn't work, so the full three-step sequence must be used for each operation. In subsequent discussion the following short notation will be use for the above sequences:

Addr[Data,x] denotes three-step strobing sequence
Addr[Data,y] denotes two-step short strobing sequence

In practice only the three-step strobing sequence is used because, for Midas, it has been microprogrammed as an additional Alto instruction and because there are no cases when it doesn't work.

The registers that the CP can load with the strobing sequence discussed earlier are as follows (Figure 17):

MIR: MIR0, MIR1, MIR2, and MIR3 select different 9-bit bytes of the 36-bit (34 data + 2 odd parity bits) "MicroInstruction Register." The CP loads it in four suboperations and must compute the two odd parity bits. Data sent to MIR by the CP is ORed into the quantity already present, so MIR should be cleared with ClrMIR before loading, as described below. The bits in MIR are scrambled with respect to the normal arrangement of instructions. MIR is unscrambled into a normal instruction by taking the bits in the following sequence: MIR0[8], MIR0[0:3], MIR1[0:3], MIR2[0:3], MIR3[0:3], MIR0[4:7], MIR1[4:7], MIR2[4:7], MIR3[4:7], MIR2[8], MIR1[8], and finally MIR3[8]. This arranges the bits into a 34-bit instruction with the 2 parity bits at the end; the first parity bit covers bits 0 to 16, the second bits 17 to 33 of the unscrambled arrangement. The value in MIR can be read on the DMux by the CP.

CPReg: These two addresses hold 16 data bits destined for transfer into Dorado. The UseCPReg bit in the Clock register causes CPReg to be substituted for Link when the instruction specifies B←Link. The B←RWCPReg function will read data from CPReg irrespective of the UseCPReg bit. B←RWCPReg actually does Link←{B←CPReg}'. CPReg is also used to pass commands from the Alto to the baseboard section.

Clock: The interpretation of the bits in Clock is as follows:

DAddrBit:	Bit shifted into DMux address register if ShiftDAddr is 1
ShiftDAddr:	Causes DMuxAddr ← (DMuxAddr lshift 1)+DAddrBit
ClrReady:	Clears the "Ready" flipflops for tasks 1 to 15 on ContA.
GetTLINK:	Do LINK←TLINK[CTD] at t_2. This is used in the first instruction single-stepped after a Jam (see below) to get LINK loaded for the newly-selected task.
UseCPReg:	Make B←Link be B←CPReg instead. CPReg is the waystation register by which Dorado is loaded from the CP.
UseDMD:	Interpret DMux address as a control function (See Table 2). Note that UseDMD should be turned off immediately after doing the function because any strobing sequence will repeat the function while UseDMD remains true.
BaseBAtten:	"Baseboard attention" causes the MC to execute the hardware function encoded in CPReg, as discussed below.

Control: The interpretation of bits in Control is as follows:

SetRun: Starts the machine running, if Stop has been cleared.

SetSS: "Set single-step". In conjunction with SetRun, causes Dorado to stop after executing the next instruction.

ClrStop: Clears Stop flipflop. After stopping, SetRun is a no-op until ClrStop is issued. Note that *SetRun must be turned off before ClrStop is issued*; this cannot be done concurrently in one strobe operation.

StopAtt1: Enables stop after each 25 ns clock. StopAtt1 modifies the action of SetSS, so halts occur after both t_1 and t_2, rather than only after t_2. Dorado does not halt after intermediate clocks of longer TPC or IM read/write instructions. StopAtt1 should only be changed at t_0; otherwise, part of Dorado goes to t_0 while other parts don't. In other words, after stopping at t_1, step one more time with StopAtt1 true, before turning it off. StopAtt1 is used only for debugging, not during ordinary operation of the hardware. *Note that StopAtt1 probably will not work at full clock speed*--tslow paths in the clock-enable logic on the MemC and IFU logic boards would cause trouble.

ClrCT: Clears CTD ("Current Task Delayed") and CTASK ("Current Task")

Jam: CTASK ← CPReg[0:3] v CTASK, CTD ← CPReg[4:7] v CTASK. Jam forces Dorado to select a task while at a breakpoint. Because of the OR'ing, ClrCT is issued prior to Jam.

Freeze: Turns off clocks to BNT, BNPC, CIA, CTASK, and TLINKX (the extension of TLINK holding the NoTask flipflop and dispatch enables); prevents dispatch bits from or'ing into TNIA; clears the BNTgtCTASK flipflop; causes TLINK to always be addressed by CTD, never by BNT; forces FreezeBC on all instructions executed; resets the hold and task simulators. Freeze must be set during all instructions executed by the CP at a breakpoint, else the ability to continue the program is lost.

ClrMIR: Clears MIR ("MicroInstruction Register")

The five clear signals and Freeze and Jam are loaded into flipflops, so the Alto program has to *set them back to zero* after accomplishing the desired clear action. In other words, the clear function keeps happening until it is turned off again.

Run, Single-Step, and Stop

The processor contains two clock control flip-flops, Run and Stop. The system clocks are enabled with Run and Stop'. The Run flipflop is simply a synchronized version of SetRun from the CP. Stop is set by dStop =

Run & (Last phase of instruction execution) & (SetSS or Error)

The effect of this arrangement is that the CP can start Dorado by first clearing the Stopped flipflop (see below), then doing Control[SetRun,x]. Dorado then runs until an error occurs or the CP deliberately halts it. If Dorado is halted and Stop has been cleared, the CP can single-step Dorado with Control[SetRun+SetSS,x]. This stops Dorado after precisely one instruction--even for multi-phase IM and TPC read/write instructions.

SetRun is ineffective after stopping until ClrStop is strobed (see below). The CP should normally not turn off SetRun when the machine is running, since that halts uncleanly; instead, do Control[SetRun+SetSS,x], which stops cleanly after the next instruction. Requiring ClrStop before another SetRun, ensures that, if the machine halts just before Control[SetRun+SetSS,x] is executed, it will stay stopped, rather than single-stepping through another instruction; this avoids a race.

Once the machine stops, SetRun must be turned off in one strobe operation, then ClrStop

issued in another strobe operation; this cannot be done concurrently.

Alto-Baseboard Communication

An Alto may be connected to a number of Dorados, each identified by an 8-bit serial number known to its baseboard (and also pasted on the chasis, so you can tell the serial number by looking). Communication uses the CPReg register, BaseBAtten flipflops (in Clock registers), and MAsync DoradoIn signals of all the Doradoes on the Alto's bus. When communicating with the baseboard, CPReg is interpreted as follows:

CPReg[0]	AMsync. This bit going different from MASync, at a time when the MC is expecting a command, signals the MC to execute the command in CPReg[5:7]. The MC indicates completion by changing MASync to equal AMsync. MASync is readable via DoradoIn by the Alto.
CPReg[1:3]	Instantaneous hardware functions as follows: 0 No-op 1 Select Dorado from CPReg[8:15]. All other Doradoes deselect; only the selected baseboard responds to the Alto CP bus. 2 Interrupt selected MC. This forces the MC to start executing its control loop, which watches for CPReg[0] to indicate a command. 3 If CPReg[15] = 1, let Alto control CP bus of selected Dorado, else MC controls it; when the Alto *is not* controlling, DoradoOut commands are received only by the baseboard section; when it *is* controlling, DoradoOut commands pass through the baseboard to the mainframe. 4-7 Undefined
CPReg[4]	Hold interrupt after executing MC command
CPReg[5:7]	Command to MC (when CPReg[0] different from MASync): 0 No-op 1 Load high memory address from CPReg[8:15] 2 Load low part of memory address from CPReg[8:15] 3 Load low part of memory address and fetch 4 Fetch and increment address 5 Store and increment address 6 Call the subroutine indicated by CPReg[8:15], where if CPReg[8:15] is 1, then the subroutine pointed to by the double-byte item PTR (address[0:7]) and PTR+1 (address[8:15]) is called 7 Undefined
CPReg[8:15]	Parameter passed to MC command

To connect to a particular Dorado, the Alto first loads CPReg[8:15] with the 8-bit serial number of the desired Dorado, CPReg[1:3] with the "Select Dorado" function, and CPReg[5:7] with a No-op command. Then it does Clock[BaseBAtten,x]. Each of these steps is executed by all baseboards on the bus. At completion, all baseboards except the one with the desired serial number are deselected, and the desired one (if it exists) is selected.

For the selected Dorado it is unknown whether the Alto or the MC is controlling the CP bus. Hence, the Alto program should next execute the hardware function that gives the Alto or the MC control of the CP bus.

Since certain MC monitoring operations cannot be performed unless the MC controls the CP bus, the Alto Midas program should arrange to give the MC control of the CP bus at regular intervals when possible.

When BaseBAtten is strobed, CPReg[1:3] are interpreted by baseboard *hardware*, irrespective of what the MC is doing at that time. CPReg[4:7] is interpreted by the *selected* MC only when it is "listening" to the Alto and CPReg[0] is different from MASync.

Having selected the appropriate Dorado and gained control of the CP bus, the Alto must next synchronize with the MC, as follows:

(1) Execute the "Interrupt MC" hardware function with "hold interrupt" in CPReg[4], no-op in CPReg[5:7], and 0 in CPReg[0]. The "interrupt MC" function causes the MC to "listen" to the Alto and to carry out one command. Because "hold interrupt" is true, the MC will remain in its listen loop after completing the command. Either MASync was already 0 when this was received by the MC, or the MC will change MASync to 0 in response to this command. The Alto waits for MASync, read on DoradoIn, to become 0.

(2) Execute the "no-op" hardware function with "hold interrupt" in CPReg[4], no-op in CPReg[5:7], and 1 in CPReg[1]. When and only when the MC has completed this command will MASync become equal to 1.

(3) Now the Alto can request any sequence of MC commands by setting CPReg[0] to MASync' each time and always setting the "hold interrupt" bit. The Alto releases the MC on the last command by setting "hold interrupt" false, which lets the MC dismiss from its listen loop and resume normal monitoring functions.

(4) The MC should complete the no-op operation in less than 2 msec (****get tighter limit****).

A MC command may leave an 8-bit result in the MAReg register, readable by DoradoIn as discussed below.

Input

The CP's first method of reading information from Dorado is to single-step an instruction that puts interesting data onto B. Data continues on B after Dorado halts. B can be read via DoradoIn, as discussed below. The second input method is to strobe an address into the 11-bit DMux address registers on each card; the signal selected by the 11-bit address can be read via DoradoIn.

In the previous DoradoOut operation, DoradoOut[0:4], the top five bits of the data field, select one of 8 four-bit input registers in the mainframe or one of three four-bit registers in the baseboard. DoradoIn reads this. When DoradoOut[3:4] are 0, DoradoOut[0:2] decode to one of the 8 mainframe registers; DoradoOut[3:4] non-0 decodes to one of the three baseboard registers as shown in Figure 1.

It is not necessary to use a strobing sequence to address an input register--a single sta 0 @DoradoOut suffices. The information that may be read by DoradoIn is shown below:

Table 1: DoradoIn Decodes

DoradoOut[0:4]	DoradoIn	Meaning
0	0:3	B[0:3]
4	0:3	B[4:7]
10	0:3	B[8:11]
14	0:3	B[12:15]
20	0	IMrhPE from MIR[16:31] during previous instruction
	1	IMlhPE from MIR[0:15] during previous instruction
	2	PE from Md
	3	RAMPE enable
24	0	IOBPE during Pd←Input or Output←B function
	1	RAMPE (Parity from RM, STK, or T wrong)
	2	MemoryPE (causes discussed in Memory chapter)
	3	MemoryPE enable
30	0	PE in MIR[16:31] now
	1	PE in MIR[0:15] now
	2	Stopped
	3	MdPE enable
34	0	IMrhPE enable
	1	IMlhPE enable
	2	IOBPE enable
	3	MIRDebug enable
1 mod 4	0	MASync
	1:3	--
2 mod 4	0:3	MAReg[0:3] (result returned by MC)
3 mod 4	0:3	MAReg[4:7]

*DoradoIn[4] is the selected DMux signal irrespective of what's in 0:3

B data read by the Alto is sometimes inverted with respect to the description of B sources and destinations in the hardware manual, and sometimes not. The name assigned to B sources and destinations in the hardware manual referred to the sense of the data on the processor's *internal* alub bus--an appropriate naming convention for writing programs. However, the Alto reads data uninverted off the *external* BMux bus with the following implications: If the data is from an external B source (anything in the IFU, memory, or control sections), then the sense of the data is inverted from the way it appears on the processor's internal bus; If the data is from a source internal to the processor, then it is read uninverted by the Alto.

Parity errors do not halt the machine until *after* the instruction producing the error has been executed. The interpretation of error indications is discussed in detail in the "Error Handling" chapter of the hardware manual.

Basic Input/Output Subroutines

When Dorado is stopped, say, at a breakpoint, and when crucial control information has been read by the procedure described below, Midas leaves Freeze and UseCPReg on. In this condition the Alto will want to force Dorado through instructions to read and write assorted registers. This is done as shown below. First, readout or execute:

```
Control[Freeze,x]                   Turn off SetRun
Control[Freeze+ClrStop+ClrMIR,y]    Clear Stop and MIR
Control[Freeze,y]                   Turn off ClrStop and ClrMIR
MIR0[I0,x]                          Load byte 0 of instruction to be executed
MIR1[I1,x]                          Byte 1
MIR2[I2,x]                          Byte 2
MIR3[I3,x]                          Byte 3
Check MIR parity                    Check for parity errors in left or right halves of MIR
Control[Freeze+SetRun+SetSS,x]
                                    Single-step the instruction just loaded
```

Read 4-bit slices off BMux as discussed in the next section, if the instruction just executed has put something interesting onto B.

Next, writing from CPReg into some Dorado register:

```
Control[Freeze,x]
Control[Freeze+ClrStop+ClrMIR,y]
Control[Freeze,y]                   Turn off ClrStop and ClrMIR
CPReg0[D0,x]                        Load data byte 0
CPReg1[D1,x]                        Load data byte 1
MIR0[I0,x]                          Load byte 0 of instruction with Something←Link
MIR1[I1,x]                          Byte 1
MIR2[I2,x]                          Byte 2
MIR3[I3,x]                          Byte 3
Check MIR parity                    Check for parity errors in MIR
Control[Freeze+SetRun+SetSS,x]
                                    Single-step through the instruction
```

Since FF specifies B←Link or B←RWCPReg on a Write, data can only pass to destinations specifiable by other instruction fields. Hence, only LdTPC←, RdTPC←, LdIMLH←, LdIMRH←, Q←, and (through the ALU) T and RM/STK can be written directly from CPReg.

Data sent from CPReg to Q← and through the ALU must be loaded *uncomplemented* into CPReg. Data sent to LdTPC←, RdTPC←, LdIMLH←, LdIMRH←, and Link← must be loaded *complemented*.

The ALU can only be used if the operations stored in ALUFM are known. Consequently, the CP normally has to load several ALUFM locations first (which can only be done indirectly by loading Q first) before using the ALU.

Since single-stepping only executes through t_2, while most registers load at t_3 (or t_4), *it will usually be necessary to clock one more instruction* (perhaps a no-op) before clobbering CPReg with new data.

The above sequences are referred to as "Xct[(Instr)]", "Read[(Instr),CPDest]", and "Write[(Instr),CPSrc]" in subsequent discussion.

In addition, a variation of Read must be provided for the Link register. As a matter of choice, Write assumes that UseCPReg is set. The consequence of this choice is that the Alto has to turn UseCPReg off to read the Link register, and then turn it back on again. This subroutine is called "RdLink[(Instr)]" below.

Another subroutine is required for changing from task i to task j because a task-specific register can only be read/written by forcing Dorado to execute instructions as that task. This involves first clearing CTASK and CTD, then jam-loading these. The sequence for doing this, called "SelectTask" later, is as follows:

```
if i eq j then done
Xct[(Noop)]                          Finish TLINK write before changing CTD
Write[(RdTPC←Link), not j]           Read new task's PC before jamming
RdLink[(B←Link),NewTPC]              since it will get smashed.
Write[(B←RWCPReg),not SaveLink]      Restore i's TLINK saved earlier
Xct[(Noop)]                          Preserve CPReg past t3

CPReg0[(j in 0:3)+(j in 4:7),x]
Control[ClrCT,x]
Control[Jam,y]                       Load CTASK and CTD with j
Control[Freeze,y]                    Turn off Jam
Clock[GetTLINK+UseCPReg,x]
Xct[(Noop)]                          Step a Noop forcing LINK←TLINK[j]
Clock[UseCPReg,x]                    Turn off GetTLINK

RdLink[(B←Link),SaveLink]            Save registers smashed during readout
Read[(B←T),SaveT]
Xct[(T←Pointers)]
Read[(B←T),SavePointers]
Xct[(T←TIOA&StkP)]
Read[(B←T),SaveTIOA]

Write[(T←Link),SaveT]                Restore registers smashed during save
Xct[(Noop)]
Write[(B←RWCPReg),not SaveTPC]       Restore task i's PC saved earlier
Write[(LdTPC←Link),not i]
Write[(B←RWCPReg),not SaveLink]      Restore j's TLINK
SaveTPC←NewTPC
```

The above sequence had to cope with two problems. The first was completing the unfinished cycle of the last instruction at the old task. The processor pipelines the task number used for the first cycle of the instruction through to the second cycle, so doing the Jam of CTD and CTASK doesn't create any problems in the processor section. However, jamming CTD will screw up the write of LINK into TLINK on the control section, so a no-op is necessary before the Jam to avoid this.

The second problem was causing LINK to get loaded from the saved value in TLINK--jam-loading CTASK doesn't automatically accomplish this. Also, jam-loading CTASK screws up the task number in the processor section, which was loaded at the last t_0, so a no-op is required after the jam to propagate the new task number to the processor section.

A side effect of SelectTask is that the wakeup request for the new task (if any) might get cleared, depending upon the mechanism used to control wakeups (several methods discussed in "Slow IO" chapter of hardware manual).

DMux

Dorado contains a serial interface called the DMux over which bits are shifted in one-at-a-time by strobing the Clock register with ShiftDAddr = 1 and a new bit in DMuxAddr. This bit may be received on each card and loaded into a 12-bit shift register. The strobe causes DMux address = ((DMux address lshift 1)+DMuxAddr).

The 12-bit register is used in two ways: The last 11 bits address one of (potentially) 2048 signals in Dorado. This signal can be read by the CP. The full 12-bit address may also be interpreted as a control function when the UseDMD bit is strobed into the Clock register. At present the control functions defined are as follows:

Table 2: DMux Control Functions

DMux Address (Octal)	Interpretation
0	*PEHaltEnable*. Low six bits mapped as follows:
+40	IM[16:31] PE enable
+20	IM[0:15] PE enable
+10	IO PE enable
+ 4	RAM PE enable
+ 2	Mem PE enable
+ 1	Md PE enable
100	*IMControl*. Low four bits mapped as follows:
+10	IM write enable
+ 4	IM address enable
+ 2	IMData[0] for IM test
+ 1	0 selects right-half of IM; 1 selects left-half
200:277	IMData[1:6] for IM test (wire-or'ed with RBMux[0:5])
300:377	IMData[7:12] for IM test (wire-or'ed with RBMux[6:11])
400:477	IMData[13:16], parity, x for IM test (wire-or with RBMux[12:15], replace parity input to IM if AddressEnable)
500:577	IMAddr[4:9] for IM test (wire-or with BNPC[4:9])
600:677	IMAddr[10:15] for IM test (wire-or with BNPC[10:15])
700:777	*IMAddr2*. Low six bits mapped as follows:
+40	MIRDebug
+20	-- (some of these bits will be used for IMAddr[2:3])
+10	--
+ 4	--
+ 2	--
+ 1	--
2200:2217	Load low four bits into ClkRate[0:3]
2220:2237	Load low four bits into ClkRate[4:7]
2240	*RunEnable*. Low four bits mapped as follows:
+10	ECLup. Enables Dorado muffler/manifold system; if false the baseboard's muffler/manifold system is alive but not Dorado's.
+ 4	EnRefreshPeriod'
+ 2	IOReset' (and stay reset)
+ 1	RunRefresh
2260:2277	*MicroCom*. Four-bit MC command.
2 =	Shut down Dorado
3 =	Shut down Dorado; interesting item on external BMux. Baseboard multiplies number by 25.6 seconds and after that elapsed time, baseboard powers up and boots again.
2300	*PowerOn*. Low four bits mapped as follows:

+10	LogicPower. Turn-on power to -5, -2, and +12 volt supplies and four fans; +5 volt supply and one fan are controlled by a switch.
+ 4	DiskPower. Turn on solid state relay enabling 115 volt AC to disk drive 0.
+ 2	Sequence0 (starts disk drive rotating)
+ 1	undefined

The IM address and data manifold registers must be loaded with zeroes when not doing an IM test. IM storage can be tested directly from the CP using manifold operations to write IM[IMAddr] and then reading IM outputs from the mufflers. To do this, BNPC must first be loaded with 0 (because the address for the direct test or's onto the BNPC outputs) and the external BMux must be low (because the write data or's with BMux); this can be accomplished by loading Q with 0 and loading an instruction that does B←Q into MIR before starting the test.

In carrying out an IM write using the manifold operations, it is necessary to manually generate an IM write pulse in three steps: (1) Set the IM control register write enable false and address enable true; load IMData[0:16] and parity and IMAddr[4:15] appropriately; (2) Turn on the write enable (which starts the write pulse); (3) Turn off the write-enable bit (which terminates the write pulse).

The 8-bit ClkRate register loaded by the above control functions determines the rate of the basic Dorado clock, nominally 25 ns. This is done by multiplying a 500 khz reference signal by ClkRate+1. The following values of ClkRate are of interest (Computation formula is ClkRate = 2000/T):

Table 3: ClkRate vs. Clock Period

ClkRate (Octal)	Period (ηs)	ClkRate (Octal)	Period (ηs)	ClkRate (Octal)	Period (ηs)
175	16	123	24	77	32
166	17	120	25	75	33
157	18	115	26	73	34
151	19	112	27	71	35
144	20	107	28	70	36
137	21	105	29	66	37
133	22	103	30	65	38
127	23	101	31		

It is necessary to wait about half a second after setting the clock rate before doing anything with the hardware that depends upon the clocks.

The other use of the DMux is for passive readout of selected signals. The last 11 bits shifted in form an 11-bit address selecting one of (potentially) 2048 signals in Dorado. The selected signal appears in DoradoIn[4] independent of information in other bits of DoradoIn.

Each card may contain a number of 8:1 multiplexors called mufflers addressed by the DMux address. Figure 2 shows the arrangement. The address is shifted into address registers on all cards simultaneously, but only one signal is delivered and sent to the Alto. The idea is that when Dorado halts, the Alto will extract all 2048 signals at once, then present any signals of interest to the user.

Blocks of DMux signal numbers (octal) are assigned to Dorado cards as follows:

ContA	0-77 and 260-377
ContB	100-257
ProcH/ProcL	400-777 arranged so that the first 10 in each group of 20 are from ProcH, the last 10 from ProcL.
MemC	1000-1177
MemD	1200-1377
MemX	1400-1777
Disk	2000-2117
Ethernet	2120-2177
Base board	2200-2377
Junk IO/IFU	2400-2777
Display	3000-3177
Storage boards	none

A comprehensive list of DMux signals is given in the Midas documentation.

The following program can be used to read all 2048 DMux addresses in only 2060 shift-read cycles. In other words, it is a generator which does not repeat any addresses:

```
let RdDMux() be
[   let V = ShiftDAddr
    Clock[V,0]                      //Select Clock register
    for I = 0 to 11 do              //Zero DMux address
    [ Clock[V,y]                    //Shift 0 into DMux address bit
    ]
    let Table = vec 127
    Zero(Table,128)
    Table!0 = (@DoradoIn & 4000B) lshift 4 //Save DMux[0] in table
    let B = 1                       //Generator starts at DMux[1]
    for I = 1 to 2047 do
    [ Clock[V+(B lshift 15),x]      //Must use three-step strobe sequence
        if (@DoradoIn & 4000B) ne 0 do
      [ let W,T = B rshift 4, 100000B rshift (B & 17B)
        if (Table!W & T) ne 0 then CallSwat()    //Never happens
        Table!W = Table!W + T
      ]
//Develop the new address bit as the xor of two current address bits
        B = ((B & 1777B) lshift 1) + (((B rshift 10) + (B rshift 8)) & 1)
    ]
]
```

When hand-coded, the above program requires about .16 sec of Alto CPU time (The inner loop averages 23 machine instructions). The microcoded version requires only .01 sec.

Power Control

The +5.0 volt power supply and one fan are controlled by a switch on the chasis called the main breaker; when this switch is turned on, the CP can control the -5.0, -2.0, and +12.0 volt supplies and the other four fans for the Dorado mainframe by executing the PowerOn manifold operation given in the table earlier. Disk drives also have a front panel switch analogous to the main breaker; note that the disk controller can control up to four drives--only disk drive 0 (the one inside the Dorado enclosure) is controllable by the mechanism discussed below.

The approximate load imposed on each of the supplies is as follows:

+5.0 volt	350 watts	Main logic supply for about 700 TTL parts.
- 2.0 volt	150 watts	supplies terminating resistors for ECL logic.
+12.0 volt	300 watts	For MOS RAM's.
- 5.0 volt	750 watts	Main ECL logic supply.

Due to power surge problems, the -5.0, -2.0, and +12.0 volt supplies should be switched off when powering up the CalComp T-80 or T-300 disk drives unless an especially beefed up wall circuit is used.

To power up Dorado, proceed as follows:

(1) Turn on the disk drive front panel switch.

(2) Turn on the Dorado main breaker, which power resets other power stuff, clears registers, and boots the microcomputer. The microcomputer then does the followings:

(a) Turns on 115 volt AC to disk drive 0 and waits 20 seconds so that the disk drive will be receptive.

(b) Issues the Sequenc0 command to the disk drive to start the spindle turning and waits 20 seconds for it to reach speed.

(c) Turns on the Dorado logic supplies, etc.

To power down Dorado, proceed as follows:

(1) Halt the Dorado forcefully.

(2) Apply IOReset.

(3) Turn off Sequence0 and wait 20 seconds (The disk drive applies dynamic braking).

(4) Turn off 115 volt AC to disk drive 0 and shut down the Dorado logic supplies simultaneously.

Problem 1: Initialize Dorado After Power Up

After power up or whenever Dorado is in an unknown condition, Midas must carry out some initialization to get Dorado into a clean and operable state. To do this, it should:

a. Assert IOReset and then set the machine speed using the DMux control function. Continually turn off SetRun and SetSS for about half a second until things settle down. The io devices mustn't do anything bad during power up/down transients and during this machine-speed setting sequence. Particularly, the disk mustn't clobber its storage and the ethernet controller shouldn't pollute the ethernet. IOReset will accomplish reset for the disk, display, and Ethernet controllers.

b. Load the RunControl register.

c. Load the parity-error halt enables.

d. Execute manifold operations to turn off the IM testing stuff (UseDMD with 100, 200, 300, 400, 500, 600, and 700 in the DMux addresses puts zeroes in all the IM testing stuff, thereby disabling it.).

e. Hold&TaskSim← is reset by Freeze, so nothing special has to be done to reset it.

f. Do 40 Xct[(NOP)]'s allowing any existing hold to finish and getting the IFU section in a passive state.

g. Xct[(IFURes)] and Xct[(NoReschedule)] to clean out the IFU.

h. Load ALUFM[14] with "NOT A" and ALUFM[0] with "B", so RM and T can be written (This has to be done by routing data through Q.).

i. Load Mcr with ReportSE', NoWake, DisHold, and NoRef bits true to prevent hold and fault task wakeups.

j. Load ProcSRN with 0 and read FaultInfo to reset the fault task wakeup request.

k. Do Write[(Q←Link),0] and Xct[(InsSetOrEvent←Q)] to turn off event counters.

l. Do Write(Q←Link),1], Xct[(IFUTest←Q)] to permanently shut off the junk task wakeup.

m. For each task, do a SelectTask(i), Xct[(TaskingOn)] (with Freeze off), load T with good parity (0 is used), Xct[(Noop)], load LINK with a reasonable default value (177777_8 is used), Xct[(Noop)], load TIOA, MemBase, RBase with reasonable default values if desired (These are not initialized presently.).

n. SelectTask(0) and Xct two NOP's to clear CTD and CTASK.

o. Load TPC with a reasonable default value (177777_8) for tasks 1 to 15.

p. Write good parity in all words of RM, STK, T, IM, and IFUM. Goto[.], FreezeBC, BreakPoint in the IM words is currently used and 0 in the other memories.

q. Put 0 in StkP to prevent StkOvf.

r. Reset the map and cache as discussed in the fine print after the Map section in the Memory chapter of the hardware manual.

Problem 2: Dorado Running--Detect Halt, Save Volatile State for Continuing

Poll the "Stopped" flipflop accessible via DoradoIn from ESTAT. When "Stopped" becomes true, Dorado has halted for some reason. The Stopped signal also becomes true when power shuts off.

The Alto begins by reading all 2048 DMux signals. Then the external BMux and error status are read via DoardoIn. These are the signals which the Alto can access without issuing any processor clocks.

Error information in DoradoIn addresses 4 to 7 reveals why Dorado has halted. Error signals are clocked at t_2, so explicit error reset is unnecessary--the error turns off at the next t_2 after the level causing the error falls. It follows that the Alto must read error status before forcing Dorado to execute any instructions. Errors do not prevent single-stepping, but only running full speed.

Parity errors in both halves of MIR should usually be interpreted as a breakpoint. A single IM PE is an indication of IM storage failure.

After reading the DMux, crucial registers in the Control section are paralyzed by turning on

Freeze, and other information is read by single-stepping Dorado through instructions that put interesting data on B (where the Alto can read it via DoradoIn).

The Alto is not directly concerned with registers inaccessible to the programmer, but some internal control registers have to be restored to continue after a breakpoint. The problem registers in the Control section are as follows:

CTASK	"Current task"--read via the DMux, frozen by Freeze, can be cleared by ClrCT and loaded by Jam--must be preserved to continue. The wakeup request for any task whose number is jam-loaded into CTASK might be lost.
CIA	"Current instruction address"--read via DMux, frozen by Freeze--must be preserved for continue. CIA contains the address of the instruction about to be executed, possibly in a different task from the one that broke.
MIR	"MicroInstruction register"--contains IM[CIA]. It is OK to smash MIR during readout because it can be restored before continue.
CTD	"Current task delayed"--read via the DMux--the number of the task that executed the last instruction (i.e., that broke); CTD is cleared by ClrCT and loaded by Jam; it is unnecessary to preserve CTD because the first no-op instruction executed after the breakpoint will finish writing the memories addressed by CTD.
CIAINC	Last instruction's address+1--just written into LINK and about to be written into TLINK[CTD], if last instruction did Call or Return--it is OK to smash CIAINC because the first instruction single-stepped by the Alto will finish writing TLINK[CTD], and because the Alto will save Link then restore for continue. CIAINC allows the address of the instruction that broke to be determined.
TLINK*	"Task-specific Link register"--has to be preserved. If the last instruction did a Call, Return, IFUJump, or Link←B, Dorado is about to write TLINK[CTD] from CIAINC or from B, and it will complete the write on the first instruction single-stepped by the Alto. The Alto can only read TLINK[i] by doing SelectTask[i] first.
LINK	The Link register for the current task. It will not be smashed on instructions single-stepped by the Alto so long as none of the single-stepped instructions specifies a call location in its branch address.
TPCI	Contains the old task's PC (now in CIA unless a switch just occurred)--TPCI will be written into TPC[CTD] during the first single-stepped instruction by the Alto. It is OK to smash TPCI during readout because it gets restored before continue.
TPC*	"Task-specific PC register"--has to be saved now and restored later because it gets smashed examining task-specific registers. The CP saves TPC by single-stepping through CTASK instructions that read TPC for all other tasks--CIA is the PC for the current task and is frozen by Freeze during all CP operations after a breakpoint.
BNPC	"Best next PC" register--unimportant.
BNT	"Best next task"--cleared by Freeze.
Wakeups	Task wakeup request levels from io devices--these may get turned off because the Alto will single-step instructions for tasks, and some io devices turn off wakeup requests when CTASK (i.e., NEXT on the backplane) equals its task. Tasks with subtasks have to manage their wakeups in a more complicated way. The fault task wakeup request is disimissed when FaultCnt is 0, when StkUnd and StkOvf are cleared.

With the above comments in mind the next step is as follows:

```
Control[Freeze,x]
Clock[UseCPReg,x]
```

This clears SetRun and SetSS and freezes crucial registers in the control section. The machine is now in the "normal" state discussed earlier in which instructions can be single-stepped, routing data into or out of CPReg. These instructions have a local branch to a Goto location (i.e., not to a Call location), so that LINK and TLINK[CTASK] won't be smashed inadvertently, unless otherwise noted.

RdLink[(Noop),garb]	No-op with UseCPReg false, then set UseCPReg true
for i = 1 to 30 do Xct[(Noop)]	
RdLink[(B←Link), SaveLink]	Save CTASK's LINK. Also smashes TPC[CTASK] but don't care because CIA is frozen.
Read[(B←T),SaveT]	Save T, since smashed below
Read[(B←Q),SaveQ]	
Xct[(T←Md)]	
Xct[(T←TIOA&StkP)]	
Read[(B←T),SaveTIOAStkP]	
Write[(Q←Link),A'Control]	Load Q with alu control for "NOT A"
Xct[(T←(ALUFMEMRW←Q), ALUF[16])]	
Read[(B←T),SaveALUFM16]	Save ALUFM[16]
Write[(Q←Link),BControl]	Load Q with alu control for "B"
Xct[(T←(ALUFMEMRW←Q), ALUF[0])]	
Read[(B←T),SaveALUFM0]	Save ALUFM[0]
Xct[(T←Pointers)]	Save Pd sources via T
Read[(B←T),(SaveMBase,SaveRBase)]	Save MemBase, RBase
Read[(B←Config'),SaveSRN]	Save ProcSRN (mask and shift)
Xct[(RBase←0)]	
Read[(B←RM0),SaveR0]	Save RM 0
Write[(RBase←Link),SaveRBase]	Restore RBase
Write[(Q←Link),not SaveQ]	Restore Q
Xct[(Noop)]	
Write[(T←Link),not SaveT]	Restore T
Xct[(Noop)]	

At this point all other non-task-specific processor registers can be read analogously to the way Q and Pointers were read above. Task-specific registers for CTASK can also be read easily. All memories can be read via B or by loading T and then reading B. Anything can be written either directly from B or T and RM by routing B through the ALU. ALUFM[16] and ALUFM[0] are smashed and TPC[CTASK] is smashed. ALUFM will not be restored until the Alto is about to step or start the Dorado at a user program's address.

By first using the SelectTask procedure given earlier, it is possible to read and write all the task-specific registers, leaving them in any desired state, except that TPC for whatever task was last jam-loaded into CTASK and CTD is smashed.

The state of Dorado with respect to continuing is as follows: Everything is preserved, "don't care", or in the desired state except for CTASK, CTD, MIR, TPCI, LINK, and CIAINC.

Problem 3: Continue From BreakPoint or Forced Halt

Continuation is easy because CIA has been frozen by Freeze during all Midas operations. This is done as follows:

SelectTask[BreakTask]	Restores CTD, CTASK, and LINK
Write[(Q←Link),SaveALUFM16]	Restore ALUFM[16] and ALUFM[0]
Xct[(ALUFMEM←Q, ALUF[16])]	
Write[(Q←Link),SaveALUFM0]	
Xct[(ALUFMEM←Q, ALUF[0])]	
Write[(Q←Link),SaveQ]	Restore Q
Xct[(no-op)]	Have to do no-op because Q loaded at t3 and when UseCPReg is turned off below, the data source for the

```
                                        write will be disturbed.
Control[ClrStop+ClrMIR,x]
Control[0,x]
LoadMIR[BreakMIR]                       Restore MIR with value at breakpoint
Clock[0,x]                              Turn off UseCPReg
Control[SetRun,x] to run -or- Control[SetRun+SetSS,x] to single-step
```

Unfortunately, there are some circumstances when the ability to continue is lost. These are as follows:

1. Examining the IFU memory at a breakpoint (which Midas doesn't do unless you display an IFU location) will result in an IFU reset, so a program using the IFU cannot be continued in this situation.

2. Examining stuff inside the memory system (cache, etc.) will do something strange if a Fault task wakeup occurred at the breakpoint.

3. Examining the Map will clobber task 15's Pipe entry.

4. It is impossible to continue from a breakpoint on Fetch←mumble, T←Md, for example. ...

Problem 4: Start or Step Arbitrary Task at Arbitrary Address

Starting at an arbitrary address is usually preceded by some subset of the power-up reset operations discussed earlier. Midas currently resets the io devices, TPC for all tasks, reads FaultInfo to reset the fault task wakeup request, and issues the ClrReady function to reset the Ready flipflops in the control section. Then it proceeds as follows:

```
SelectTask i
Write[(Q←Link),SaveALUFM16]
Xct[(ALUFMEM←Q, ALUF[16])]
Write[(Q←Link),SaveALUFM0]
Xct[(ALUFMEM←Q, ALUF[0])]
Write[(Q←Link),SaveQ]
Xct[(no-op)]
Write[(B←RWCPReg),not address]          Puts new address in LINK

MIRx[(Return, B←RWCPReg)]               Return to new address while restoring Link
CPReg0[SaveLink byte 0,x]               This is same as Write but with Freeze off
CPReg1[SaveLink byte 1,x]
Control[ClrStop,y]                      Single-step with UseCPReg on.
Control[SetRun+SetSS,y]                 Since Link and CPReg have same value this is ok.
Clock[0,x]                              Turn off UseCPReg
Control[SetRun,x] -or- Control[SetRun+SetSS,x]
```

Problem 5: Dorado Running--Force It to Halt Cleanly

Control[SetRun+SetSS,x] does the job. Dorado halts cleanly after an instruction. The state save and restore is then the same as in the previous section.

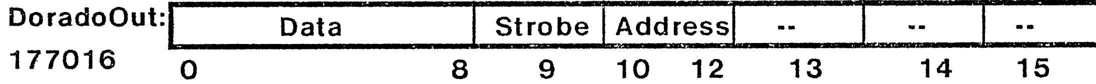
Use of Debugging Stuff From Dorado Microprograms

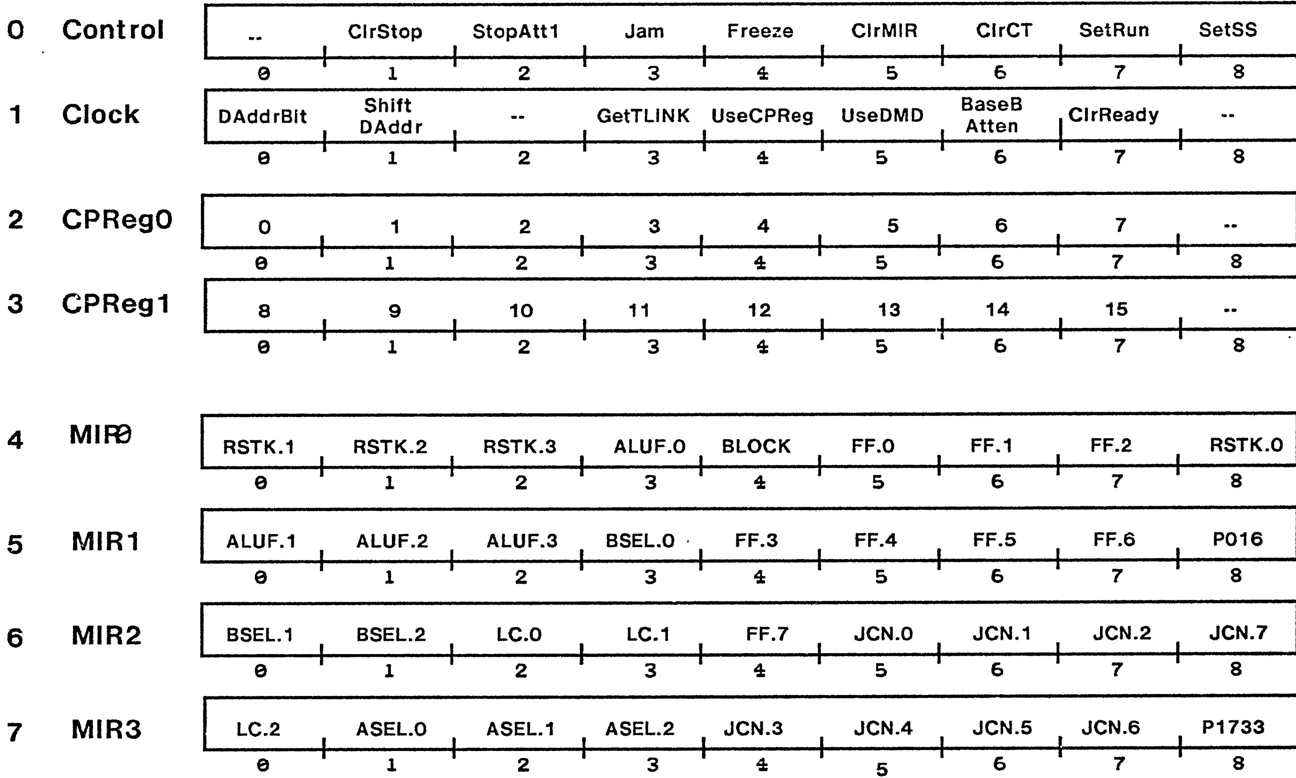
The MidasStrobe←B function shifts B[4] into the DMux address registers, so the new DMux address becomes (Old address lshift 1) + B[4]. The selected bit is readable by Pd←ALUFMEM, which puts DMuxdata on Pd[0].

The idea behind this feature is that Dorado programs can be written which test various hardware features invisible except via the DMux.

Unsolved Problems and Thoughts

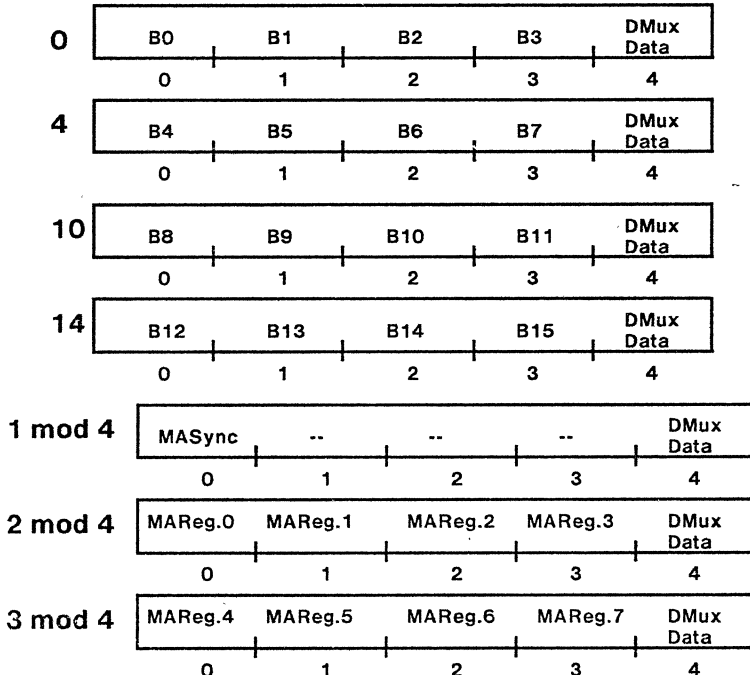
1. Reading memory from Alto (need io device?)
2. What if breakpoint with HOLD true?
3. Ready flipflops, Freeze discussion, inability to continue.
4. Discussion about MIRDebug.

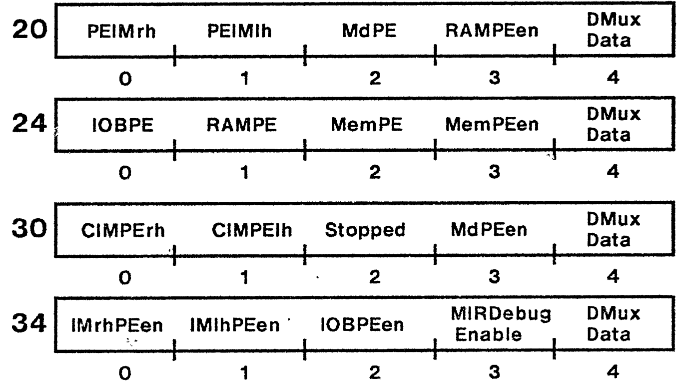




P016 = odd parity on MIR0[0:3], MIR1[0:3], MIR2[0:3], MIR3[0:3], and MIR0[8]

P1733 = odd parity on MIR0[4:7], MIR1[4:7], MIR2[4:8], and MIR3[4:7]





PEIMrh and PEIMlh are piped and show the condition that caused Dorado to halt. CIMrh and CIMlh are derived from the data presently in MIR.

Figure 1: Dorado Debugging Interface

D1Debug1.Sil

10/10/79

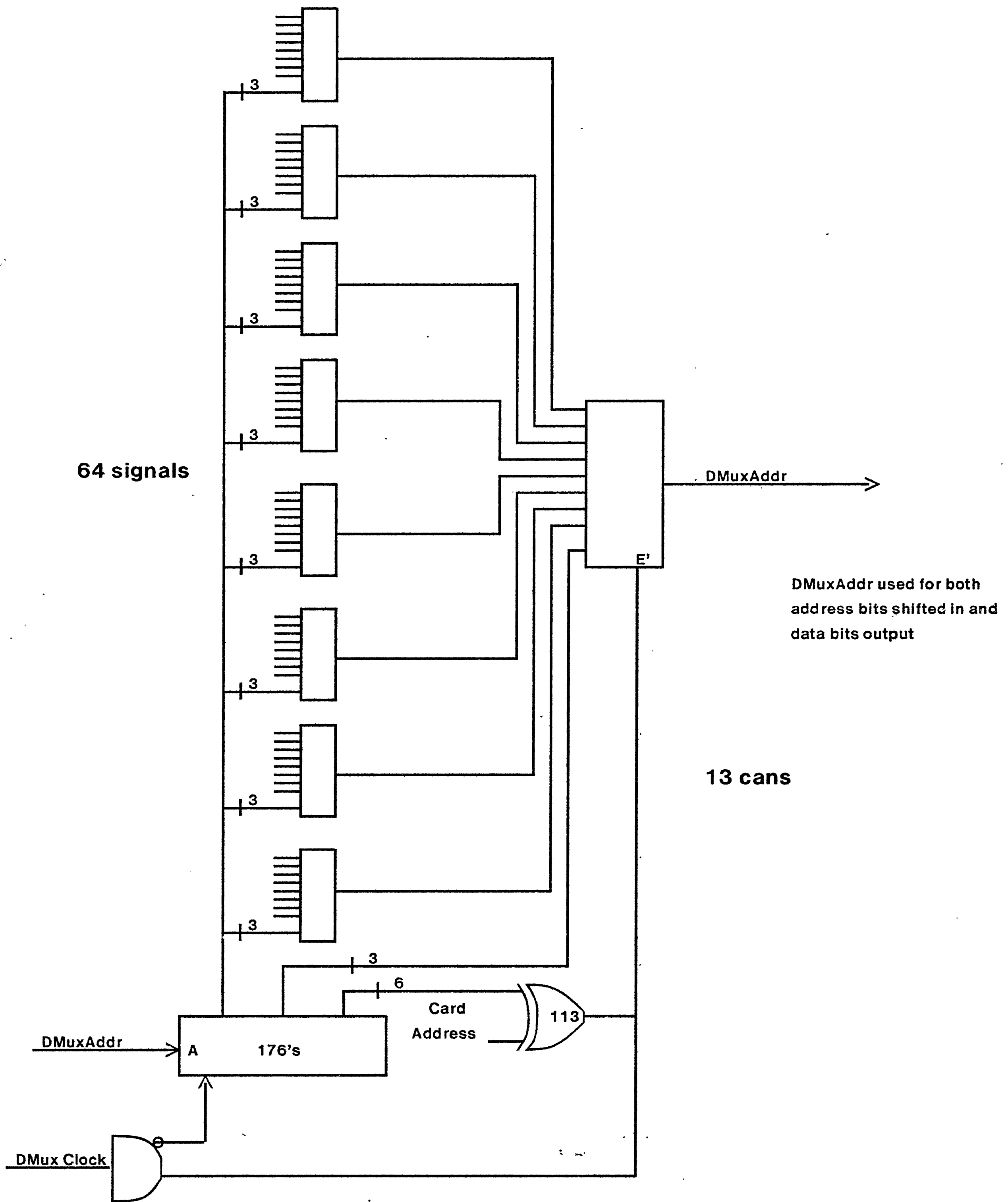


Figure 2: DMux

D1Debug2.sil
7/28/77