This document is for Xerox internal use only

D0 Hardware Manual

May 16, 1979

XEROX

Reprographic Technology Group
Electronics Division
Los Angeles

This document is for Xerox internal use only

Table of Contents

1.0 Introduction
1.1 Notation
1.1.1 Numbers
1.1.2 Special Characters
1.1.3 Terms
1.1.4 Register Naming Conventions

2.0 Major Subsections
2.1 Timing
2.2 Control
2.3 Registers and Data Paths
2.4 Timing

3.0 Arithmetic Section
3.1 Register Summary
3.2 R Memory
3.2.1 R Address Formation
3.2.2 Stkp and SStkp
3.3 T Register
3.4 Constants
3.5 Arithmetic/Logic Unit
3.5.1 SALUF
3.5.2 Result Register
3.6 Cycler/Masker
3.6.1 Short Fields
3.6.2 Mesa Field Operations
3.6.3 BitBLT
3.6.3.1 BitBLT Registers
3.6.3.2 Transfer Inner Loop
3.6.4 Mesa Instruction and Operand Access
3.6.4.1 PCX and NewInst
3.7 Parity Register
3.8 Special Functions
3.9 Miscellaneous Registers
3.9.1 Watchdog Timer
3.9.2 Power Monitoring
3.9.3 RS232 Interface
3.9.4 Printer Interface
3.9.5 Maintenance Panel
3.9.6 Time of Day Clock
3.10 Timers

4.0 Control
4.1 Normal Instruction Sequencing
4.2 Conditional Branches
4.3 Subroutines and Tasking
4.4 Dispatches
4.5 Aborted Instructions
4.6 Faults
4.7 Notify
4.8 Writing and Reading Registers
4.9 Reading and Loading the Control Store
4.10 Bootstrapping

5.0 Memory
5.1 Organization
5.2 Memory Reference Instructions
5.2.1 Reference Types
5.2.2 R Addresses
5.2.3 Quadword Overflow
5.2.4 I/O Register Addresses
5.2.7 Address Calculation
5.3 R Interlocking
5.4 The Map
5.5 The Error Pipe
5.6 Error Correction
5.7 Refresh
5.8 Memory Timing
5.9 Storage Card Organization

6.0 Input-Output
6.1 Interface Signals
6.2 Controller Addressing
6.3 Task Wakeup Requests
6.4 Input and Output Operations
6.5 I/O Attention, I/O Strobe, Run

7.0 User Terminals and Controllers
7.1 Terminal To Controller Interface
7.1.1 Cables and Connectors
7.1.2 Drivers and Receivers
7.1.3 Video and Control Channel
7.1.4 Backchannel
7.2 UTVFC (User Terminal Variable Format Controller)
7.2.1 Introduction
7.2.2 Output Registers
7.2.3 Timing and Sync Generation
7.2.4 Data Buffering
7.2.5 Cursor
7.2.6 Backchannel
7.2.7 Controller Identification and Diagnostic Input

8.0 Rigid Disk Controller
8.1 Introduction
8.1 Disk Characteristics
8.3 D0-Controller Interface
8.3.1 Output Registers
8.3.2 Input Registers
8.3.3 Seek Control
8.3.4 Disk Commands
8.3.5 Wakeups and IO Attention
8.4 Hardware Organization
8.4.1 Timing
8.4.2 Sequence Address Generation
8.4.3 Format Sequencer
8.4.4 Buffer Control Sequencer
8.5 Basic Sequencer Operations
8.6 Error Correction

Appendix A: Time of Day Clock
Appendix B: MC1 and MC2 Microcode
Appendix C: Standard I/O Device Interface
Appendix D: RDC Sequencer Microcode

1.0 Introduction

This document describes the D0 processor, memory, and input-output system. The D0 is a microprogrammed machine which is customized to some extent to provide efficient emulation of the Mesa instruction set, and to provide high memory bandwidth for demanding input-output devices. The D0 has a multitasking control structure which multiplexes the processor among sixteen fixed priority tasks at the microcode level. The lowest priority task is used to implement the emulator for the Mesa instruction set.

The principal performance parameters of the D0 are:

> Clock Rate: TBDns. Microinstructions are pipelined, and require four cycles for execution. A new microinstruction is started every two clock times.
>
> Data Path Width: 16 bits
>
> Arithmetic: 2's complement
>
> Control Store: 4K words of 36 bits
>
> Main Storage: 22-bit virtual address space provided. 64K-768K words of real memory may be connected (1M word with 64K RAM chips). Main storage is error corrected over a 64-bit quadword.

1.1 Notation

Throughout this document, a number of conventions are used, which are described in this section.

1.1.1 Numbers

Numeric quantities are expressed in decimal unless otherwise specified. The suffix b is used to indicate octal.

* is used to indicate multiplication, ** is used to indicate exponentiation:

> 5d3 = 5000 = 5*10**3, 3b5 = 300000b = 3*8**5

For large multiples of a power of 2, K is used to designate 2**10, and M is used to designate 2**20:

> 32K = 32*2**10 = 2**15 = 32768,
> 1M = 1*2**20 = 2**20 = 1048576

1.1.2 Special Characters

<x> means "contents of x".

Square brackets [ ] are used to indicate indexing or to delimit the arguments of a function:

> x[3] = <x+3> means the contents of location x+3, i.e. the third element of the vector x
>
> hf[2] means the value returned by the function hf with argument 2.

Double commas are used to indicate the concatenation of two fields. If x is a 3-bit field and y is a 5-bit field, then x,,y is an eight-bit field with x in its high order bits.

1.1.3 Terms

A *word* is a sixteen bit quantity. Bit 0 is the most significant bit, bit 15 is the least significant bit. When diagrammed, bit 0 is on the left.

A *doubleword* is a thirty-two bit quantity, with bits numbered from 0 to 31. In main storage, the least significant bits (16-31) of a doubleword are stored in location n, the most significant bits (0-15) are stored in location n+1.

A *byte* is an eight bit quantity. Bit 0 is the most significant bit, bit 7 is the least significant bit. When diagrammed, bit 0 is on the left.

A *field* is a contiguous group of bits within a word or larger field. The bits are numbered from the left. For example, the field consisting of the least significant byte of x is indicated with x[8:15].

1.1.4 Register Naming Conventions

Registers in the D0 (usually) have names that are related to the function performed by the register. This section discusses the conventions used for register names in this document and in the logic diagrams for the machine. Note that these conventions are usually, but not universally, observed. In cases where clarity is increased by deviating from these conventions, we have deviated.

In the simplest case, a register holds a single n-bit value. The bits of the register are simply numbered, for example, H1[0:15]. In the logic drawings, the field notation is not used, since these drawings depict individual signals. Instead, the register name and the bit number are "dotted" to form a signal name, e.g., H1.00, or Stkp.7.

In some cases, a single register will contain subfields which are explicitly named. For example, the field ALUF[0:3] is a subfield of the microinstruction register MIR. In the logic drawings, the individual bits of a register are given the subfield name (e.g., ALUF.0), since this is more descriptive of the signal's function than giving the bit number in the larger register. Often, a group of signals which have a similar purpose or similar timing are treated as a register with a number of single-bit subfields. The Result register, which consists of the signals Overflow, ALUOut=0, ALUCarry, and ALU<0, is an example. On the logic drawings,

the name "Result[i]" never appears. The Result register is a logical entity provided solely for clarity of description.

In a number of cases, registers in the D0 hold complemented values, or the values are complemented as the register is read from one part of the processor to another. In this document, register names are used as if the register contained high-true values, but the tables which summarize the registers are explicit about complementation, as are the logic drawings. For example, we speak of "the H2 register", or "H2[0:15]" frequently. In fact, H2 holds complemented data, but the descriptions of the ALU functions have been chosen to provide the necessary inversion. In the logic drawings, the bits of this register are named H2.00' through H2.15' (i.e., low-true names).

2.0 Major Subsections

This section describes the major subsections of the D0. Later sections will provide increased detail. The principal registers and data paths of the processor are shown in figure 2.1. Figure 2.2 shows the control section, figure 2.3 shows the principal paths associated with the memory, and figure 2.4 shows miscellaneous control logic. These figures correspond to the partitioning of the logic onto printed wiring boards. Figure 2.5 summarizes the microinstruction format of the D0. Two formats are used, one for memory references, one for other instructions.

2.1 Registers and Data Paths

The arithmetic section of the D0 is organized around a 16-bit Arithmetic/Logic Unit (ALU). The ALU is fed by two registers H1 and H2 which are inaccessible to the programmer.

The H1 register is loaded at t1 from the R bus, which is usually fed from the R memory, but may also be driven by a number of other registers in the processor. The R bus is driven by tri-state drivers, and goes to all processor cards.

The H2 register is loaded at t1 from the T RAM or from the F1 and F2 fields of the microinstruction. F1 and F2 are interpreted as an eight bit constant in this case. The constant may be placed in either byte of a word; the other byte is zeroed.

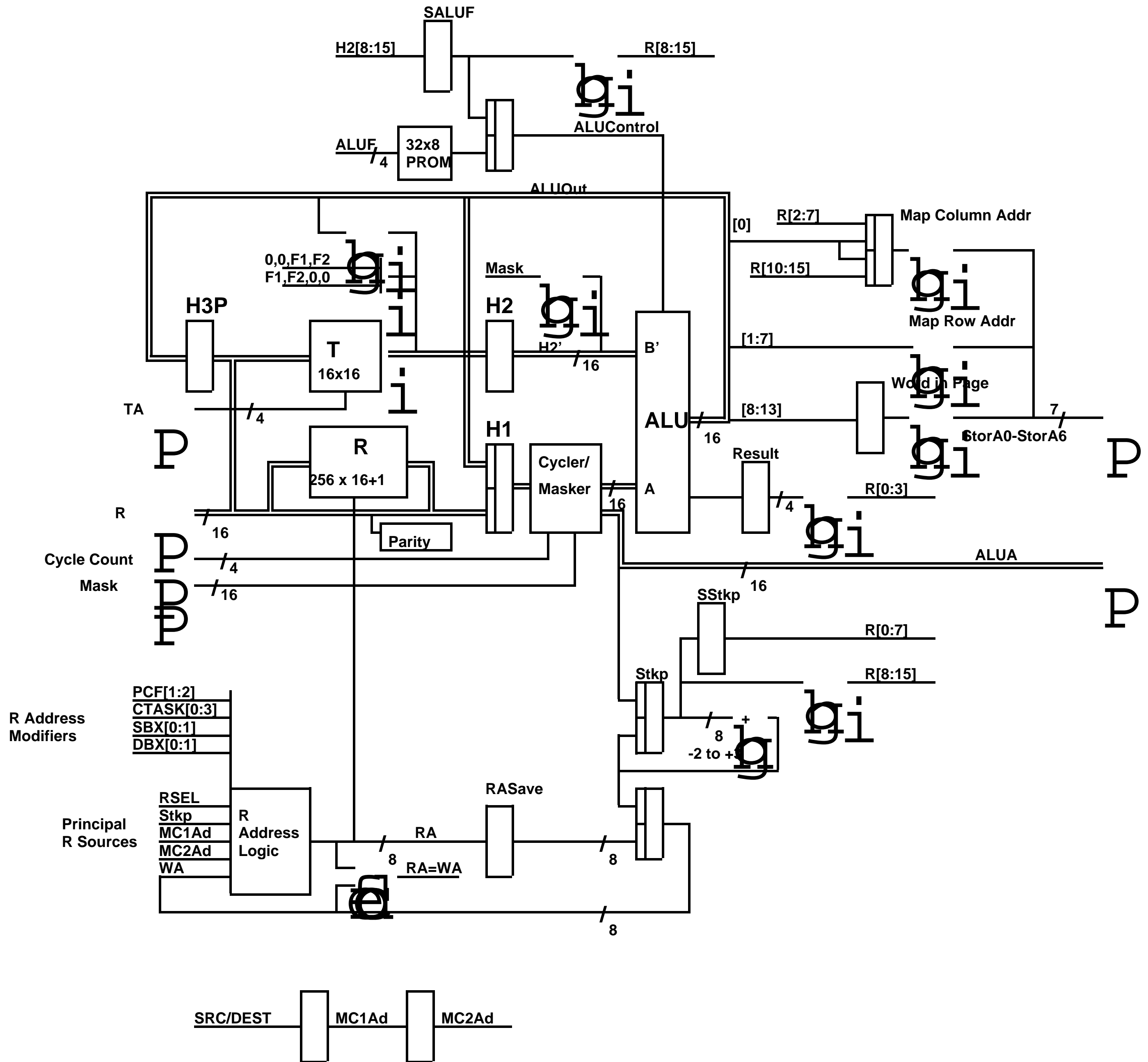


Figure 2.1 Processor Data Paths

2.2 Control

The D0 provides a task switching mechanism which multiplexes the processor among sixteen fixed priority microcoded tasks. The lowest priority task is used to implement an emulator for the Mesa instruction set. The highest priority task (task 15) is used for error handling. Task 14 is permanently assigned to an array of interval timers, and the remaining thirteen tasks are used to implement the microcoded portions of input-output device controllers.

A hardware device controller requests service from the processor by placing an encoded version of its associated task's number on three wakeup request lines. If this task number is greater than the task which is running on the processor, and if there is no higher priority device requesting a wakeup, the task associated with the requesting device will acquire the processor when the running task next executes a RETURN instruction. The T register and the microprogram location counter (TPC) are *task specific*, which makes task switching overhead small. Once a task has acquired the processor, it will keep control until it executes a RETURN instruction and its wakeup request line is no longer active or there is a higher priority wakeup request. Microprograms should do RETURNS every few cycles {maximum TBD} to avoid long latencies for higher priority tasks.

The lowest priority task (task 0) is always requesting a wakeup. This task contains the microcode for the Mesa emulator. Task 0 will run if there is no higher priority input-output activity. The highest priority task (task 15) is used for error handling functions, and is given control by a special mechanism when an error occurs.

The processor provides the facility for each task to modify the saved program counter of other tasks, and to cause another task to get control of the processor.

The processor does not have an incrementing program counter. Instead, each microinstruction specifies the address of its successor using one or more fields in the microinstruction. There are a number of branch formats, which use a varying amount of the microinstruction for their specification depending primarily on their degree of locality. The processor provides a single level of subroutine linkage for each task. When a subroutine call is executed, the return link is stored in the TPC RAM location associated with the current task (TPC[current]). When a RETURN is executed, the next microinstruction is accessed from the location in the control store pointed to by the TPC cell associated with the hightest priority requesting task (TPC[HTASK]). This is the reason that RETURNS cause task switches.

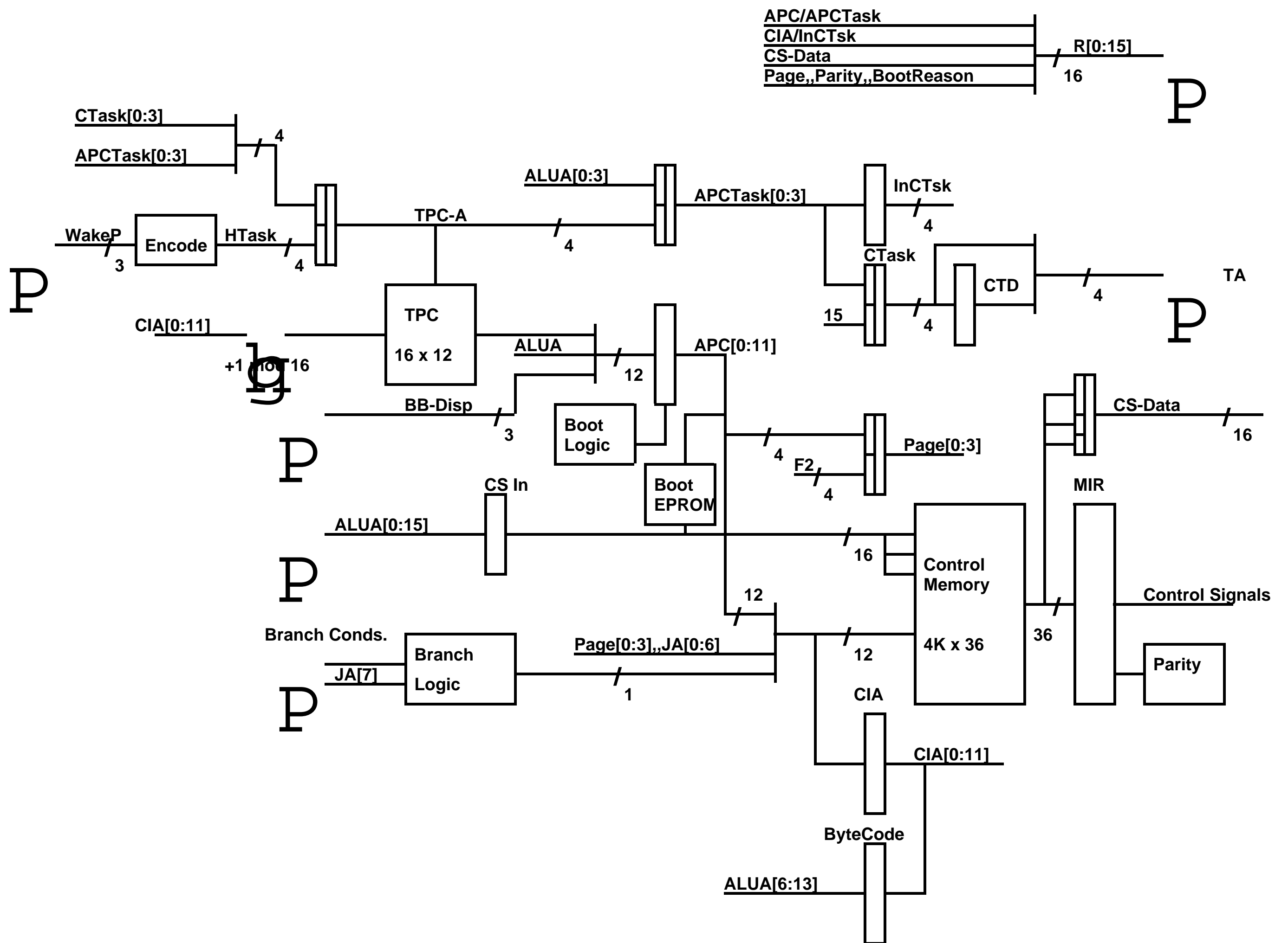


Figure 2.2 Processor Control

2.3 Memory and Input-Output

The memory system of the D0 supports a 2**22 word virtual address space and a 768K word real address space. When 64K RAM chips become available, it will be possible to attach up to 1M words of storage. All memory references are made by the processor, including references made on behalf of IO controllers.

A special microinstruction format is used to initiate memory references. The instruction specifies a pair of R registers to be used as a base register, the type of reference, and the source or destination of the data. Once the memory has been started, references proceed in parallel with processor activity.

The memory is pipelined, allowing two memory references to be in progress at once.

From the standpoint of the IO controllers, memory references and input-output operations are essentially identical, since they use the same busses for data and device address transfers. There are separate busses for input and output, and these activities may proceed in parallel if circumstances permit. The processor has an OUTPUT operation which transfers a single word from an R register to the device register addressed by H2[8:15], and an INPUT operation which reads the IO device register addressed by H2 into an R register.

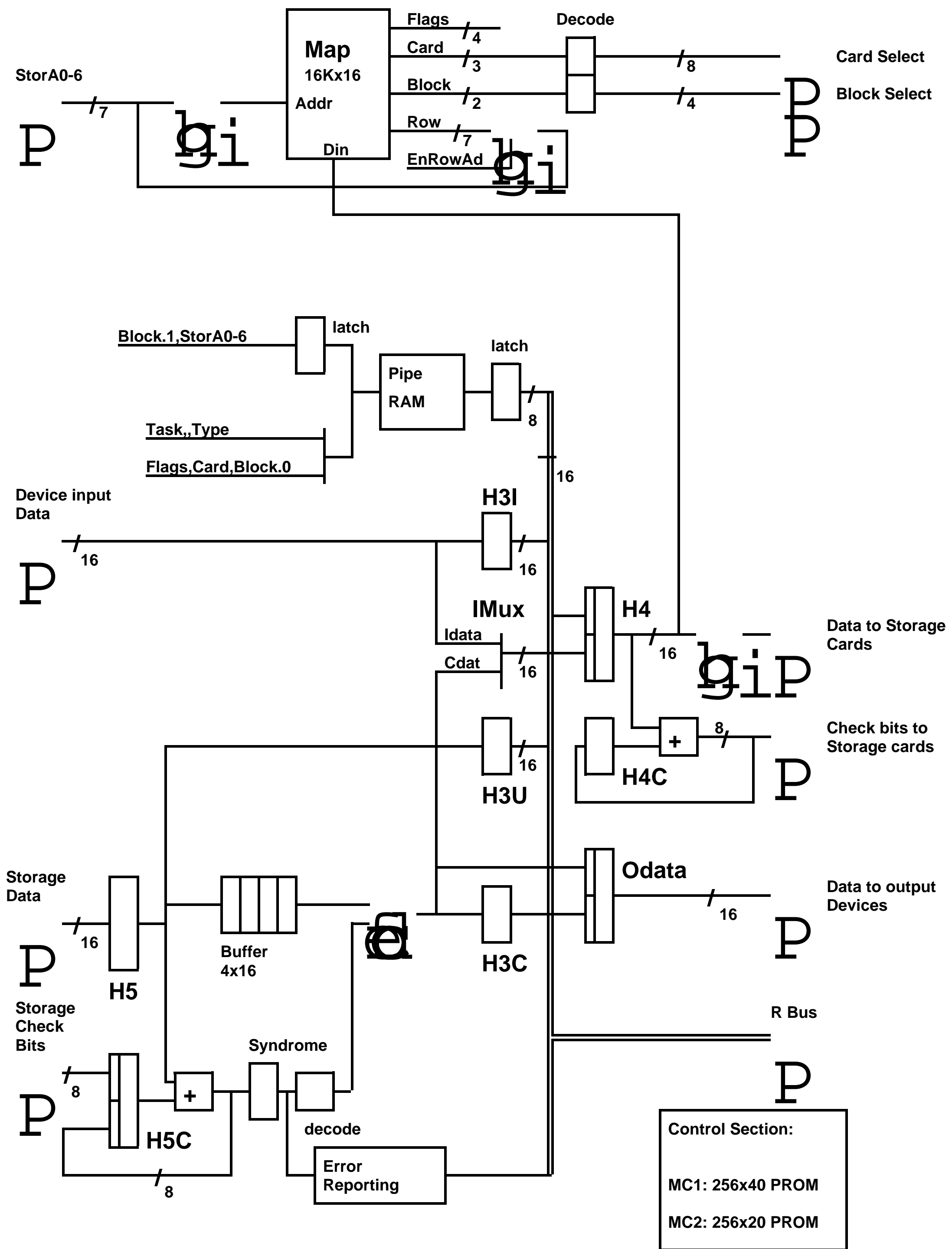


Figure 2.3 Memory Control Data Paths

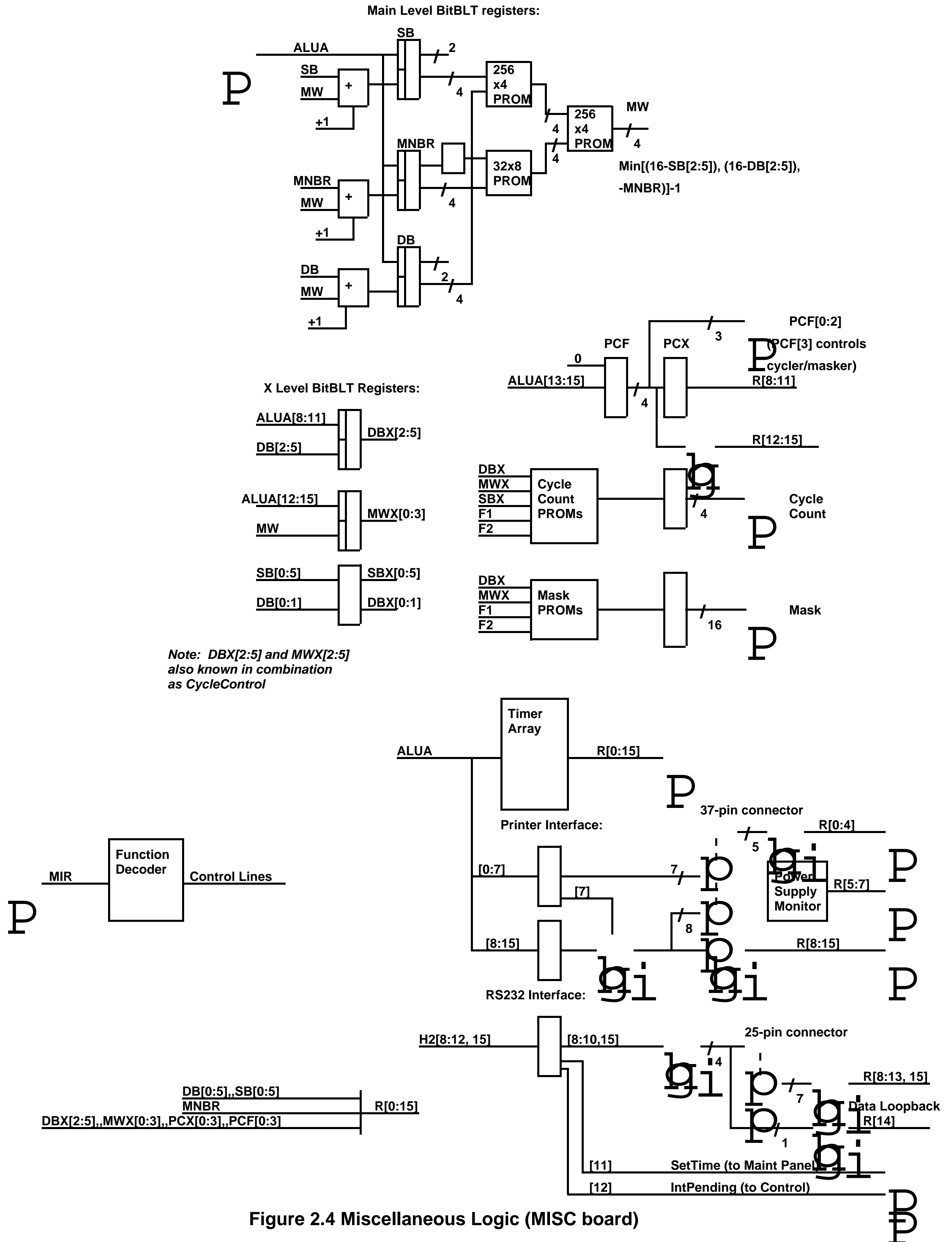
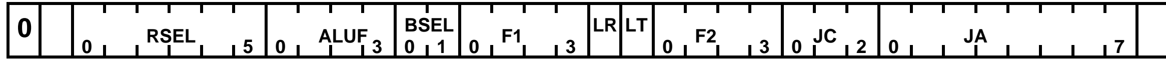


Figure 2.4 Miscellaneous Logic (MISC board)

Normal Instruction Format:



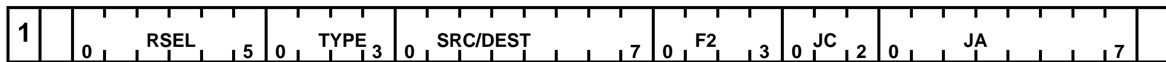
Note: these bit numbers are used when reading and writing the control store. See section 4.9

ALUF	Function
0	H2
1	ALUA
2	ALUA and H2
3	ALUA or H2
4	ALUA xor H2
5	ALUA and not H2
6	ALUA or not H2
7	ALUA xor not H2
8	ALUA+1
9	ALUA+H2
10	ALUA+H2+1
11	ALUA-1
12	ALUA-H2
13	ALUA-H2-1
14	unassigned
15	use SALUF

BSEL	Meaning
0	0,,F1,,F2
1	F1,,F2,,0
2	T
3	T (F1,,F2 is a field descriptor)

JC	Meaning
0-3	Conditional Branch
4	GoTo
5	Call
6	Return
7	Dispatch

Memory Reference Format:



DF2 means take the displacement from F2 instead of from T (F2 is treated as a 4-bit constant, and the normal F2 actions are disabled).

TYPE	SRC/DEST bits
0: -	-
1: IOfetch 4	device dest
2: ReadPipe	R dest (2)
3: Refresh	-
4: Pfetch1	R dest (2)
5: Pfetch2	R dest (2)
6: Pfetch4	R dest (2)
7: Input	R dest (1)
8: Pstore1	R src (2)
9: Pstore2	R src (2)
10: Pstore4	R src (2)
11: Output	R src (1)
12: IOfetch16	device dest
13: IOstore4	device src
14: XMap	R src/dest (2)
15: IOstore16	device src

(1): Device address is H2[8:11] or CTask[0:3],,H2[12:15]

(2): SRC/DEST=0 means use Stkp for R Address

Figure 2.5 Microinstruction Format

2.4 Timing

This section discusses the conventions used to describe the timing of the D0, and gives an overview of the timing of the basic data paths.

A *cycle* is the basic unit of time in the machine. Cycles are TBD ns. in length. The clock associated with a particular cycle occurs at the *end* of the cycle.

An *instruction time* is two cycles, since even though most instructions really require four cycles to finish all their activity, a new instruction may be started every two cycles.

Time within an instruction is counted from the time the instruction is loaded into the MIR (MicroInstruction Register). This is called t0.

The period between t0 and t1 is referred to as cycle0, that between t1 and t2 as cycle1, etc.

The timing of normal instruction execution is shown in figure 2.6. During cycle0, the data sources used in the instruction (usually the R and/or T registers) are accessed. This data is loaded into the H1 and H2 registers at t1. During cycle1 and cycle2, the data is operated on by the ALU, and the result is loaded into the H3P register at t3. During cycle3, data from H3P is written into R and T if the instruction specifies loading of these registers.

Calculation of the next instruction's address and the control store fetch for the next instruction is done during cycle0 and cycle1 in parallel with execution of an instruction. Note that t2 for one instruction is usually coincident with t0 for the next instruction, t3 is coincident with t1, and t4 is coincident with t2.

During execution of a conditional branch instruction, the control store access is begun at t0 using an even address, assuming that the test condition is false and that the branch will not be taken. The branch conditions become stable slightly after t1, and if the condition is true, an extra cycle is inserted to allow the control store sufficient time to access the odd location. This is not visible to most of the processor logic, since the CPU control store simply withholds one clock. (Note: EdgeClockFeed is withheld, RamClockFeed is not. This means that RAM writes during cycle1 *may* be done twice, but this is harmless.)

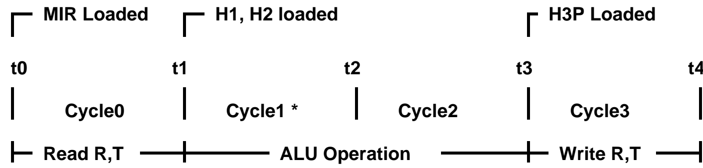
Under certain circumstances, an instruction may be *Aborted*, as described in section 4.5. Aborted instructions are repeated automatically unless a *fault* occurs (section 4.6).

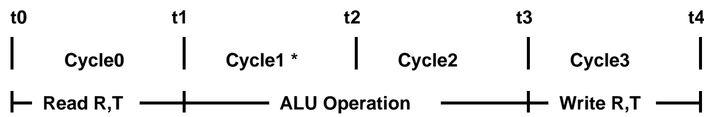
Microinstructions may also be *suspended*. Suspension occurs when the memory controller needs to access the R memory. The processor is suspended for one cycle for each word transferred between the memory and R. Suspension does not withold any clocks, so it is invisible to I/O devices (except for the logic that generates wakeup requests). Processor activities are simply delayed when a cycle is suspended.

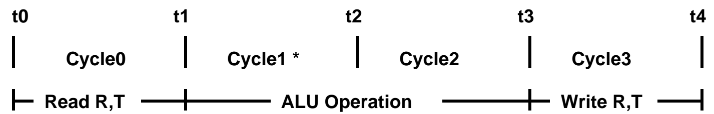
Most of the discrete registers in the processor are loaded at t2. The data used to load these registers is usually taken from the ALUA bus, which becomes stable at ~t1+45ns.

The only registers which are written later than t2 are R, T, and the Result register.

Normal Instruction flow







* Will be extended by one cycle if a conditional branch succeeds

Memory Reference Instruction

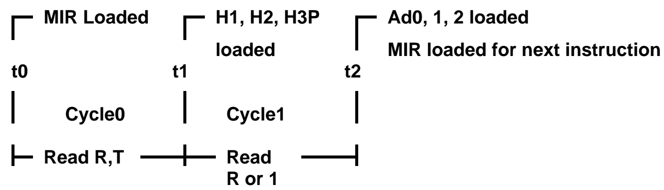


Figure 2.6 Instruction Timing

3.0 Arithmetic Section

3.1 Register Summary

Table 3.1 is a summary of all registers and memories which are accessible to the programmer.

3.2 R Memory

The R Memory is a 256 word by 16 bit memory used for high speed storage in the processor. The RSEL and RMOD fields of the microinstruction select an R register to be read or written during an instruction. In addition, a number of discrete registers in the processor are selected for reading (but not loading) by RSEL and RMOD. Information read from R is latched in the H1 register (inaccessible to the programmer) at t1, from which it is passed through the cycler/masker to one of the ALU inputs.

Since a microinstruction requires four cycles for execution, with R writes occupying the last cycle, and since instructions may be started every two cycles, an R location may be written by one instruction and read by the next before the write is actually done. The processor contains logic to bypass the R memory in this case so that the write appears to a microprogram to be done during the instruction in which it is specified.

A microinstruction generates a *single* R address. The selected R register may be used as a data source under control of other fields of the instruction, and is written from the output of the ALU if LR = 1.

3.2.1 R Address Formation

The R address is an 8-bit quantity. If RMOD = 0, the R address is formed by concatenating the two most significant bits of the current task number and the (6 bit) RSEL field. In addition RSEL[0:1] is *replaced* by the two least significant bits of the current task number if RSEL[0:1] = 0. The effect of this is to divide the 256 word R memory into four 64-word blocks. Four tasks share a block. Within the block and the four tasks which can directly address it, task 0 can address the entire block, tasks 1, 2, and 3 can only address locations 16-63. The idea here is that (1) the emulator task needs to be able to directly address an entire 64 word block, and (2) up to four IO controllers of the same type can share microcode if their task numbers are initialized to 0,1,2,3 mod 4. The microcode is written as if the device occupied task 0 of the block of four tasks, and the R addresses (and IO device addresses) will be modified according to which of the four tasks is in control.

If RMOD=1, the R address is modified as follows:

> If RSEL[4:5]=0, 1, or 2, RSEL[4:5] is replaced by the registers PCF[1:2], SBX[0:1], or DBX[0:1] respectively. This allows these registers to index a 4-word area in R. The PCF register contains the low order three bits of the Mesa byte program counter, SBX and DBX contain the current source and destination word numbers for two quadword buffers used by the BitBLT operation.
>
> If RSEL[4:5]=3, and RSEL[0:1]=3, the stackpointer (Stkp) provides the 8-bit R address. In this case,

RSEL[2:3] indicates the amount by which the stackpointer is to be incremented or decremented.

If RSEL[4:5]=3, and RSEL[0:1]=0, 1, or 2, the R memory is disabled for reading, and other R bus sources are enabled. A number of discrete registers in the processor are read in this way (see Table 3.1). Registers which are less than sixteen bits in length are packed into words as tightly as possible without crossing word boundaries. The idea here is that these registers may be accessed using *short field descriptors* (described later). The function RegShift (F2=0) is provided to increase the number of registers which may be selected in this way.

Table 3.1

Register	Size (bits)	Loaded From	Load Control	Load Time	Read To	Read Control	Section
R	256x16	H3P	LR=1	Cycle3	H1	RMOD=0	3.2
T	16x16	H3P	LT=1	Cycle3	H2	BSEL=2,3	3.3
Map	16Kx16	R	XMap	-	R	XMap	5.4
SALUF:							
MA'	1	H2[08]	F2=11b	t2	R[08] {1}	SR=7	3.5.1,3.6.3.2
MB	1	H2[09]	F2=11b	t2	R[09] {1}	SR=7	3.5.1,3.6.3.2
SALUF[0:5]	6	H2[10:15]	F2=11b	t2	R[10:15]{1}	SR=7	3.5.1
Result:							
ALU<0	1	ALUOUT[00]	GB=11b	t3	R[00]{1}	SR=7	3.5.2
ALUCarry	1	ALUOUT[01]	GB=11b	t3	R[01]{1}	SR=7	3.5.2
ALU#0	1	ALUOUT[01]	GB=11b	t3	R[02]{1}	SR=7	3.5.2
NoOverflow	1	ALUOUT[03]	GB=11b	t3	R[03]{1}	SR=7	3.5.2
Note: Result is also loaded at t3 of all instructions that do not contain F2=2 (FreezeResult)							
Stkp	8	ALUA[8:15]	F2=1	t2	R[8:15]{1}	SR=3	3.2.2
SStkp	8	Stkp[0:7]	{6}	t2	R[0:7]	SR=3	3.2.2
APCTask	4	ALUA[0:3]	GB=10b	t2	R[0:3]	SR=43b	4.3
APC	12	ALUA[4:15]	GB=10b	t2	R[4:15]	SR=43b	4.3
InCTsk	4	APCTask[0:3]	{3}	t2	R[0:3]	SR=47b	4.6
CIA	12	C-SAddress	{3}	t2	R[4:15]{1}	SR=47b	4.1
CSData	16	C-S	GB=14-17b,H2	t2	R[0:15]	SR=53b	4.9
Page	4	F2[0:3]	F1=5	t2	R[0:3]	SR=57b	4.1
Parity Errors:							
StackOvf	1	{3}	{4}	t2	R[04]	SR=57b	3.7
CS-ParErr	1	{3}	{4}	t2	R[05]	SR=57b	3.7
R-ParErr	1	{3}	{4}	t2	R[06]	SR=57b	3.7
MemErr	1	{3}	{4}	t2	R[07]	SR=57b	3.7
BootReason:							
RFB[0]	1	TesterBoot	{5}	boot	R[10]	SR=57b	4.10
RFB[1]	1	PanelBoot	{5}	boot	R[11]	SR=57b	4.10
RFB[2]	1	WDTBoot	{5}	boot	R[12]	SR=57b	4.10
RFB[3]	1	GB=6	{5}	boot	R[13]	SR=57b	4.10
RFB[4]	1	PwrBoot	{5}	boot	R[14]	SR=57b	4.10
RFB[5]	1	ParityBoot	{5}	boot	R[15]	SR=57b	4.10
DB	6	ALUA[10:15]	F2=6	t2	R[4:9]	SR=33b*	3.6.3.1
SB	6	ALUA[10:15]	F2=5	t2	R[10:15]	SR=33b*	3.6.3.1
MNBR	16	ALUA[0:15]	F2=13b	t2	R[0:15]	SR=37b*	3.6.3.1
CycleControl:							
DBX[2:5]	4	ALUA[8:11]	F2=4	t2	R[0:3]	SR=27b	3.6.2,3.6.3.1
MWX	4	ALUA[12:15]	F2=4	t2	R[4:7]	SR=27b	3.6.2,3.6.3.1
PCF	4	0,ALUA[13:15]	F2=14b	t2	R[12:15]	SR=27b	3.6.4.1
PCX	4	PCF[0:3]	{6}	t2	R[8:11]	SR=27b	3.6.4.1
RS232 In	8	{3}	-	-	R[9:15]	SR=37b	3.9.3
RS232 Out	8	H2[8:15]	F1=1	t2	-	-	3.9.3
Printer In	16	{3}	-	-	R[0:15]	SR=27b*	3.9.4
Printer Out	16	ALUA[0:15]	F2=17b	t2	-	-	3.9.4
Memory Syndromes:							
Slot A	8	{3}	-	-	R[0:7]	SR=13b	5.5
Slot B	8	{3}	-	-	R[8:15]	SR=13b	5.5

Notes:
SR=n means RSEL=n, RMOD=1, GB means Group B Function
* RShift (F2=0) must be asserted
{1} Register is read in complement form
{2} Cannot be read directly
{3} Cannot be loaded directly
{4} Loaded when an error is detected. Causes a trap to location 1 if nonzero
{5} Loaded when machine is bootstrapped for any reason
{6} Loaded at t2 of all instructions located at 2001b + 4*n, n=0-377b.

3.2.2 Stkp and SStkp

Stkp (stackpointer) is an 8-bit register used to address the R memory indirectly. The stackpointer is *not* task-specific, and tasks other than the emulator which use it must save and restore its value. The stackpointer may be read onto the R bus (in complement form) and loaded from ALUA as described in Table 3.1. The only way in which a task can address the entire R memory is via the stackpointer.

When the stackpointer is used to address the R memory (when RSEL[0:1]=0, RSEL[4:5]=3), it may be incremented at t2 under control of RSEL[2:3] and the function StackShift (F2=3). The amount of the increment is shown in Table 3.2.2. The increment is done modulo 16. Note that the function StackShift is also sent to controllers on the backplane as IOStrobe, which means that I/O microcode cannot increment Stkp by +2, +3, or -3 in one microinstruction without generating IOStrobe. The Stkp value used to read the R memory is the value before any increment specified by the instruction is done, the value used to write R is the incremented value.

Table 3.2.2

RSEL[2:3]	StackShift	Stkp Increment
0	0	0
1	0	+1
2	0	-1
3	0	-2
0	1	+2
1	1	+3
2	1	0
3	1	-3

There are several R locations that cannot be read or written indirectly via the stackpointer without causing a fault (see section 4.6). If Stkp=11b-17b, or if Stkp=0 and the current microinstruction is reading from the stack, a StackOvf fault will be caused. The intent here is to place the Mesa stack in locations 1-10b, and use this mechanism to generate the Mesa StackOverflow trap. The size of the stack is set by a PROM on the ALU board. (Note: The precise condition corresponding to "Instruction Reading Stack" is MemInst' and ALUF#0 and RSEL[0:1] = 0 and RMOD and RSEL[4:5] = 3. In particular, this means that an instruction must select R as one of the ALU inputs for StackOvf to be detected.)

The register SStkp (Saved Stkp) is provided to save the value of the stackpointer at the start of execution of every Mesa bytecode. This is necessary since the bytecode may cause a trap, and it is essential to be able to reset the state of the machine to its pre-trap value if this occurs. SStkp is loaded from Stkp at t2 of microinstructions located at 2001b + 4*n (n=0-377b). These locations contain the first microinstruction of every bytecode. SStkp may be read onto the R bus as indicated in Table 3.1.

3.3 T Register

T is a task-specific temporary register (i.e. there are 16 copies of T, one per task). T is read during cycle0 of an instruction, and the value read is loaded into H2 and used as one ALU operand if the BSEL field of the microinstruction equals 2 or 3. T will be loaded from the

output of the ALU (actually, from the H3P register) during cycle3 if the microinstruction bit LT is asserted. There is logic to bypass T so that if a write is specified during one instruction, the data may be used during the following instruction. This logic addresses T from CTask during Cycle0 and from CTD during Cycle1. [Note: Since writes of T are piped across memory references, CTD (which addresses T for writing) is not clocked if MemInst = 1. Also, if T is loaded by a microinstruction immediately preceding a memory reference and read by the instruction immediately following the memory reference, the reader will get the wrong value, since the ALU output will no longer contain the value about to be written into T, but bypassing will still be invoked. This situation must be avoided by the programmer.]

3.4 Constants

The processor provides two forms of constants which may be used as ALU operands. If BSEL=0, the eight-bit concatenation of the F1 and F2 fields is loaded into bits 8-15 of H2 at t1, and is used as the ALU operand during cycle1 and cycle2. Bits 0-7 of H2 are zeroed. If BSEL=1, F1 and F2 are placed in bits 0-7 of H2, and bits 8-15 are zeroed. When a constant is specified in an instruction, the normal actions of F1 and F2 are disabled.

3.5 Arithmetic/Logic Unit

The ALU is a 74S181, which can perform a number of arithmetic functions, as well as all the logical functions of two input variables. The ALU inputs are the output of the H2 register (which contains the datum selected by the BSEL field of the microinstruction), and the ALUA bus, which is the output of the cycler/masker. In normal instructions, the H1 and H2 registers are loaded at t1, and the results of the ALU operation are loaded into H3P and the RESULT register at t3. The ALUA bus is stable before t2, and is used as the source of data for a number of the discrete registers in the processor. The control signals for the ALU are normally taken from the ALUF field of the microinstruction. This four bit field is mapped into the control signals required by the 74S181 by a PROM which implements sixteen of the most frequently used ALU functions. The functions provided by the ALUF field are:

Table 3.5

ALUF	ALUOut =
0	H2
1	ALUA
2	ALUA and H2
3	ALUA or H2
4	ALUA xor H2
5	ALUA and not H2
6	ALUA or not H2
7	ALUA xnor H2
8	ALUA+Cy1
9	ALUA+H2+Cy0
10	ALUA+H2+Cy1
11	ALUA-Cy1
12	ALUA-H2-Cy0
13	ALUA-H2-Cy1
14	unassigned
15	use SALUF for ALU function

In instructions in which the function UseCoutAsCin is not used, Cy0=0 and Cy1=1 in Tables 3.5 and 3.5.1. If UseCoutAsCin is asserted, Cy0 = Cy1 = Result[1], i.e. the carry bit from

the last ALU operation is used as the ALU carry in.

3.5.1 SALUF

In addition to the functions provided by the ALUF field, there is an 8-bit register SALUF which may be loaded from H2[08:15] under control of an F decode, and subsequently used to execute any function of which the SN74S181 is capable. Six bits of this register are used to supply the six bits required by the ALU chips as described in Table 3.5.1, the remaining two bits are used by the BitBLT primitives.

Table 3.5.1

H2[10:15]	Function
0	* ALUA + Cy0
1	* ALUA + Cy1
2	* (ALUA or H2') + Cy0
3	(ALUA or H2') + Cy1
4	* (ALUA or H2) + Cy0
5	(ALUA or H2) + Cy1
6	- 1 + Cy0
7	- 1 + Cy1
10b	(ALUA and H2) + ALUA + Cy0
11b	(ALUA and H2) + ALUA + Cy1
12b	(ALUA or H2') + (ALUA and H2) + Cy0
13b	(ALUA or H2') + (ALUA and H2) + Cy1
14b	* ALUA + H2 + Cy0
15b	* ALUA + H2 + Cy1
16b	(ALUA and H2) - 1 + Cy0
17b	* (ALUA and H2) - 1 + Cy1
20b	(ALUA and H2') + ALUA + Cy0
21b	(ALUA and H2') + ALUA + Cy1
22b	* ALUA - H2 - 1 + Cy0
23b	* ALUA - H2 - 1 + Cy1
24b	(ALUA or H2) + (ALUA and H2') + Cy0
25b	(ALUA or H2) + (ALUA and H2') + Cy1
26b	(ALUA and H2') - 1 + Cy0
27b	*(ALUA and H2') - 1 + Cy1
30b	ALUA + ALUA + Cy0
31b	ALUA + ALUA + Cy1
32b	(ALUA or H2') + ALUA + Cy0
33b	(ALUA or H2') + ALUA + Cy1
34b	(ALUA or H2) + ALUA + Cy0
35b	(ALUA or H2) + ALUA + Cy1
36b	* ALUA - 1 + Cy0
37b	* ALUA - 1 + Cy1
40-41b	ALUA'
42-43b	ALUA' and H2
44-45b	ALUA' and H2'
46-47b	zero
50-51b	ALUA' or H2
52-53b	* H2
54-55b	* ALUA xor H2'
56-57b	* ALUA and H2
60-61b	ALUA' or H2'
62-63b	* ALUA xor H2
64-65b	H2'
66-67b	* ALUA and H2'
70-71b	- 1
72-73b	* ALUA or H2
74-75b	* ALUA or H2'
76-77b	* ALUA

* Normal ALUF field functions

3.5.2 Result Register

The four condition bits ALU<0, ALUCarry, ALU#0 and NoOverflow are held in the Result register, which is normally loaded at t3 of every instruction (this register is *not* task-specific). Normally, it is expected that branches on these conditions will be done in the instruction following the ALU operation which causes the condition. There is a function (FreezeResult, F2=2) which inhibits the loading of the register so that branches which test the ALU result can be deferred if desired. The Result register is not loaded with the results of an ALU operation if FreezeResult is executed during that instruction. The function UseCOutAsCin,

F2=16b, is provided to use ALUCarry as the carry into the ALU.

The Result register is read onto R[0:3] using RSEL=7, RMOD=1. The register is read in complement form. The Result register is loaded at t3 from ALUOUT[00:03] by the Group B function Restore (11b). This function is provided primarily to restore the state of the machine after processing of a fault.

Note: Since Result is not task specific, it cannot be used to return a value from a subroutine (since the RETURN may switch tasks).

3.6 Cycler/Masker

The cycler/masker is provided for three purposes:

1) To provide efficient implementation of the Mesa ReadField, WriteField, ReadString, WriteString, and Shift instructions.

2) To provide the capability of rapidly unpacking (and optionally dispatching on the value of) fields in an R register or other R bus source.

3) To provide an efficient implementation of the BitBLT operation.

The cycler/masker is controlled in a number of ways, which will be described in detail below.

3.6.1 Short Fields

When BSEL=3, F1 and F2 are concatenated and used as an 8-bit *Short Field Descriptor* which is used to control the cycler/masker. In all short field operations, H2 is loaded from T.

The values of F1,,F2 and their associated operations, as well as the assembler macros used to perform the short field operations are:

LDF[RBsource,POS,SIZE] is used to right-justify any field. POS and SIZE are octal constants which specify the first bit and the width of the field. RBsource is any register which can be placed on the Rbus.

0	- 17b:	16 1-bit fields starting at bit 0, 1, ..., 15
20b	- 36b:	15 2-bit fields starting at bit 0, 1, ..., 14
37b	- 54b:	14 3-bit fields starting at bit 0, 1, ..., 13
55b	- 71b:	13 4-bit fields starting at bit 0, 1, ..., 12
72b	-105b:	12 5-bit fields starting at bit 0, 1, ..., 11
106b	-120b:	11 6-bit fields starting at bit 0, 1, ..., 10
121b	-132b:	10 7-bit fields starting at bit 0, 1, ..., 9
133b	-143b:	9 8-bit fields starting at bit 0, 1, ..., 8
144b	-153b:	8 9-bit fields starting at bit 0, 1, ..., 7
154b	-162b:	7 10-bit fields starting at bit 0, 1, ..., 6
163b	-170b:	6 11-bit fields starting at bit 0, 1, ..., 5
171b	-175b:	5 12-bit fields starting at bit 0, 1, ..., 4
176b	-201b:	4 13-bit fields starting at bit 0, 1, 2, 3
202b	-204b:	3 14-bit fields starting at bit 0, 1, 2
205b	-206b:	2 15-bit fields starting at bit 0, 1

DISPATCH[RBsource,POS,SIZE] is used to load APC with the selected

field with SIZE =< 4 bits in preparation for a dispatch operation in the next microinstruction.

207b	-226b:	16 1-bit fields starting at bit 0, 1, ..., 17
227b	-245b:	15 2-bit fields starting at bit 0, 1, ..., 16
246b	-263b:	14 3-bit fields starting at bit 0, 1, ..., 15
264b	-300b:	13 4-bit fields starting at bit 0, 1, ..., 14

RSH[RBsource,shiftcount] right-shifts RBsource by shiftcount 1 to 15.

uses LDF[RBsource,0,(16 - shiftcount)] codes

LSH[RBsource,shiftcount] left-shifts RBsource by shiftcount 1 to 15.

301b -317b: 15 left shifts of 1, ..., 15 bits

LCY[RBsource,shiftcount] left-cycles RBsource by shiftcount 1 to 15.

320b -336b: 15 left cycles of 1, ..., 15 bits

RCY[RBsource,shiftcount] right-cycles RBsource by shiftcount 1 to 15.

uses LCY[RBsource,(16 - shiftcount)] codes

RHMASK[RBsource] is RBsource & 377b

uses LDF[RBsource,8,8] code

LHMASK[RBsource] is RBsource & 177400b

337b: RBsource & 177400b

ZERO is zero

340b: zero

FixVA[RBsource] is provided to propagate bits 0 and 8 of the high half of a memory base register pair into bits 1 and 9, so that the test for virtual addresses >22 bits will work properly.

341b: RSH[RBSource,1] and 40100b

Fields 342-355 are provided to mask the value in H1. The intent is to mask a register containing -1 to produce the indicated small constant on ALUA.

342b:	RBsource and 2
343b:	RBsource and 4
344b:	RBsource and 5
345b:	RBsource and 6
346b:	RBsource and 10b
347b:	RBsource and -2b
350b:	RBsource and -3b
351b:	RBsource and -4b
352b:	RBsource and -5b
353b:	RBsource and -6b
354b:	RBsource and -7b
355b:	RBsource and -10b

Field 356 places bits 0-3 into bits 8-11, and masks out the rest of the word.

356b: RSH[RBsource,10b] and 360b

The remaining fields have no effect.

3.6.2 Mesa Field Operations

A Mesa field descriptor is an eight bit quantity in which bits 0:3 indicate the bit number of the first (leftmost) bit in the field, and bits 4:7 indicate the width of the field in bits minus 1. The cycler contains logic to optimize the Mesa RF (read field) and WF (write field) instructions by controlling the cycler/masker directly from a field descriptor.

The RF function (F1=14b) causes the cycler to right justify the quantity from R, and masks out all but the rightmost *width* bits of the field. Precisely, RF does:

ALUA _ H1 LCY[DBX[2:5] + MWX[0:3] + 1] and MASK[MWX[0:3]],

where MASK[x] contains 1's in bits 15-x through 15.

The WFA operation takes a source word from R, shifts it to its correct position in the destination, and masks out all bits except those in the field. A second microinstruction containing WFB is then used to insert the field into the destination word. WFA is F1=11b, WFB is F1=13b. Precisely, WFA does:

ALUA _ H1 LCY[15 - (DBX[2:5] + MWX[0:3])] and MASK1[DBX[2:5], MWX[0:3]],

where MASK1[x,y] contains 1's in bits x through x+y. WFB does:

ALUA _ H1 and not MASK1[DBX[2:5], MWX[0:3]].

The CycleControl register is loaded at t2 from ALUA[8:15] by the function CycleControl_ ALUA (F2=4). CycleControl is not a discrete register, but is the concatenation of DBX[2:5] and MWX[0:3]. These two registers are also used by the BitBLT operations. Although CycleControl can be read onto the R bus (RSEL=27b, RMOD=1), tasks other than the emulator should not attempt to use CycleControl by saving its value, using it for an operation, then restoring the value, all between task switches. The reason for this is that when CycleControl is loaded, SBX[0:5] and DBX[0:1] are also loaded with information which is a function of the SB and DB registers, so these registers would also have to be saved and restored. Thus, although it is possible to define a set of conventions for the emulator microcode which would allow another task to save and restore SBX, DBX, and MWX, this would be a time consuming operation.

3.6.3 BitBLT

The purpose of the BitBLT (bit boundary block transfer) operation is to move information from one region of main storage to another, modifying the information at the destination as the transfer is done. The location of the source and destination areas are specified to a precision of one bit. The operation moves a number of *items* which are contiguous fields of fixed width separated in storage by a fixed increment (width and increment also have a precision of one bit). An operation of the form dest_dest *op* src is done, where *op* is any logical function.

Since the BitBLT operation is time consuming, a significant amount of hardware exists in the processor to optimize its operation. The transfer uses two quadword areas in R, one for the source and one for the destination. Two six-bit registers (SBX and DBX) are used to index these buffers; the most significant two bits of these registers select a word within the buffer, the low four bits point to a bit within the selected word. The R addressing logic uses the two most significant bits of these registers to index a word in R, the cycler/masker is controlled by the low four bits of the registers. Each time the inner loop of the microcode transfers a portion of an item (a "chunk") from source to destination, the registers are incremented by the number of bits in the chunk. The number of bits transferred by a single iteration of the inner loop (the chunk width) is determined by hardware which examines the values of SB and DB, and transfers as many bits as possible without overflowing a word boundary or exhausting the item.

The microcode for the BitBLT operation is divided into three major sections:

> 1) Startup and termination, which sets up the parameters of the instruction in a form suitable for the microcode, and handles the final state after the transfer is complete.
>
> 2) The inner loop, which does the transfer.
>
> 3) Routines which refill the source and destination quadword buffers from main storage, and update counts and addresses.

The operation of the inner loop will be described in a subsequent section.

3.6.3.1 BitBLT Registers

The D0 contains six registers whose primary purpose is to support BitBLT. The interconnection of these registers is shown in Figure 2.4, and their characteristics are summarized in Table 3.1. This section describes them in detail.

The registers are divided into two groups or *levels*. During each iteration of the two instruction inner loop of BitBLT, SB, MNBR, and DB (the main level) are clocked at t2 of the first instruction, and SBX, MWX, and DBX (the "X" level") are clocked at t2 of the second instruction. SB and DB are 6-bit pointers into two 64 bit (4 word) buffers in R, one for the source information and the other for the destination information. MNBR contains the (negative of the) number of bits remaining in the current item. During each iteration of the

inner loop, a quantity (MW) is calculated which represents the maximum amount by which the source and destination pointers may be advanced without crossing a source or destination word boundary or exhausting the item. MW is the number of bits (precisely, it is the number of bits-1) which will be transferred by one iteration of the inner loop.

Figure 3.6.1 shows the process in detail. At point A, SB, DB, and MNBR are loaded with the correct values for iteration n. From these values, MW for iteration n is calculated. At point B, SBX and DBX are loaded from SB and DB, MWX is loaded from MW, and iteration n begins under control of the X level registers. Between points B and C, MW (+1) is added to SB, DB, and MNBR, and these registers are updated at point C in preparation for the next iteration.

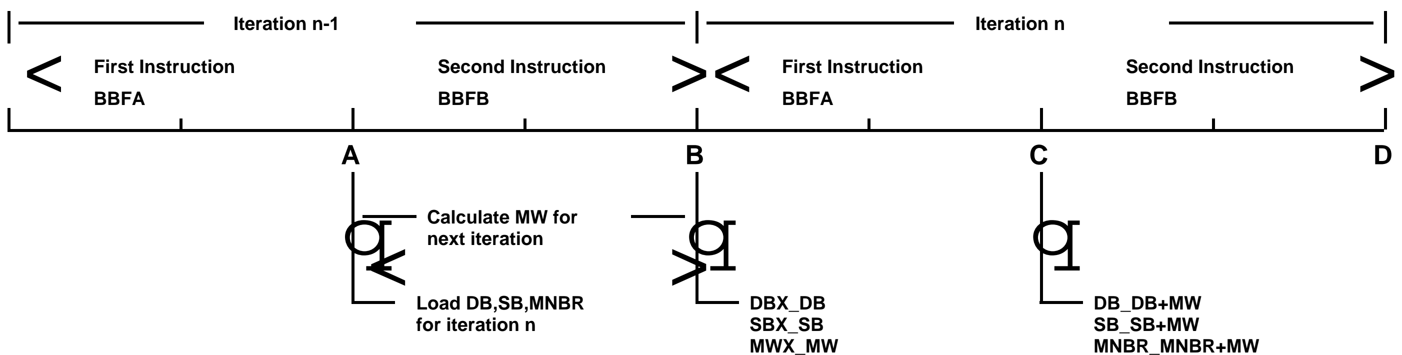


Figure 3.6.1 BitBLT Inner Loop Timing

The updating of the main level registers is done at the end of the first instruction of an iteration by the BBFA function (F1=0). BBFA also has other effects, described later. The updating of the X level from MW and the main level is done by the functions BBFB (F1=12b) or BBFBX (F1=15b).

SB, DB, and MNBR may be read as R bus source (see Table 3.1). When an item begins, SB, DB, and MNBR are loaded (from ALUA) with the pointers and count for the item, the quadword buffers in R are filled, and the main loop is entered with an instruction which includes BBFBX (this advances the main level into the X level in preparation for the first iteration).

During the first instruction of each iteration, the BBFA function sets up a 3-bit dispatch based on the values of SB, DB and MNBR for the *next* iteration. This dispatch determines whether or not the loop is to terminate, and if so, how. The conditions tested are:

1) If MNBR is about to overflow (i.e., if the most significant bit of the adder feeding MNBR=0), the current item is exhausted.

2) If SB is about to overflow (i.e., if the adder feeding SB produces a carry), the source quadword buffer is exhausted and must be refilled.

3) If DB is about to become zero, the destination buffer is filled and must be stored (and possibly refilled).

4) If none of the above occurs, another iteration should be done.

Note that if the buffers require refill, it is not necessary for the refill microcode to modify any of the registers, since they will have been set up properly for the iteration following the refill by the BBFB function in the second instruction of the loop. The various conditions are encoded into the dispatch value generated by BBFA as follows:

Dispatch Value	Meaning
3	Item Exhausted (MNBR about to become zero)
4	SB and DB Exhausted
5	SB Exhausted
6	DB Exhausted
7	Continue

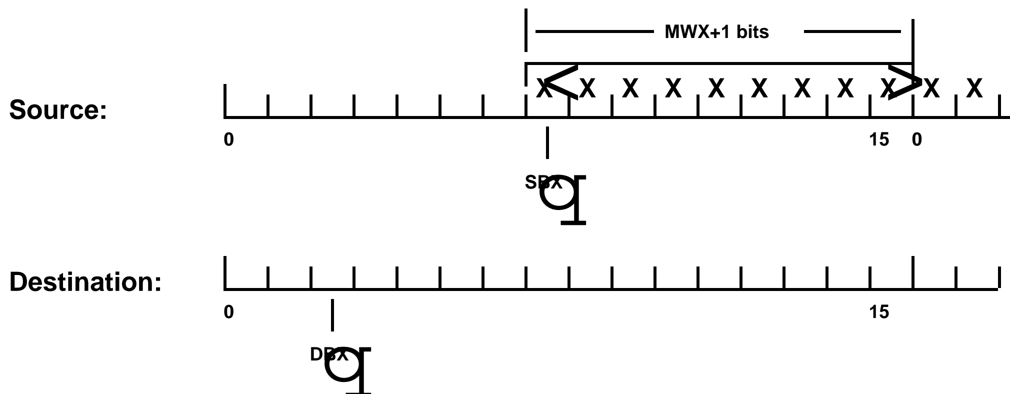
Values 0-2 cannot occur.

3.6.3.2 Transfer Inner Loop

The BitBLT inner loop transfers as much of one word as possible between the source and destination buffers. This number is the minimum of (1) the number of bits required to reach the next source word boundary (16-SB[2:5]), (2) the number of bits required to reach the next destination word boundary (16-DB[2:5]), and (3) the number of bits remaining in the current item (-MNBR). This quantity is calculated by PROMS from the registers SB, DB, and MNBR, and is loaded into the register MWX (precisely, MWX_MIN(...) -1) when BBFB or BBFBX is executed. The BBFA function (F1=0), left-cycles and masks the source data (from R) in the cycler masker. The cycle count is SBX-DBX, and the source mask extends from bit DBX to bit DBX+MWX. Figure 3.6.2 illustrates the source and destination words and the values of SBX and DBX before and after a single iteration of the BitBLT loop.

Before BitBLT Loop:

SBX = 7
DBX = 2
MWX = 8



After BitBLT Loop:

SBX = 0
DBX = 11

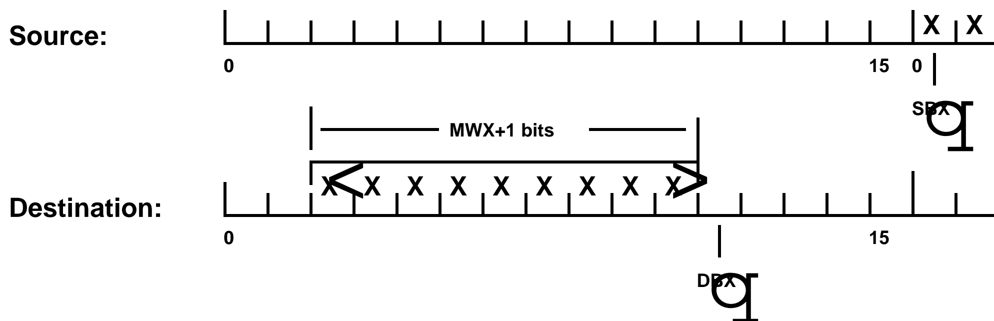


Figure 3.6.2 BitBLT Shifts and Masking

In addition to the bits that control the ALU, the SALUF register contains two control bits, MA and MB, which are used by the BitBLT operation. MA controls the data on the T (H2) side of the ALU during BBFA. If MA=1 during BBFA, bit positions in the ALU H2 input word *not* covered by the source mask are filled with 1's. If MA=0, these bits are filled with 0's. This will occur even though the BSEL field of the instruction specifies T as the input to H2 (T is disabled by special logic). The idea here is that the first instruction of the BitBLT loop will specify T_R OR T, BBFA, DBLGOTO[X,Y,MB] and the word which is placed in T will contain the proper field of the source, correctly aligned with the destination, filled with 1's if MA=1, or with 0's if MA=0. The second instruction of the BitBLT loop will do *dest_f(dest,T)* where f is some bitwise logical function (set up in SALUF before the transfer is begun). If f(dest,1)=dest (e.g., f=AND), MA should be initialized to 1, which will cause bits in the destination that are not a part of the operation (i.e., not covered by the source mask) to be preserved. If, on the other hand, f(dest,0)=dest (e.g., f=OR), MA should be initialized to 0, again preserving unused destination bits.

MB is a flag bit in SALUF, for which a branch test is provided. The intent is to use MB to indicate whether or not the bits of the destination covered by the source mask are to be cleared before combining them with the source data (in T). The first instruction of the BitBLT loop will contain a branch on MB. If MB=1, indicating that the destination bits are to be cleared, the branch will send control to an instruction that does dest_f(dest,T), BBFB. If the destination bits are not to be cleared, the other arm of the conditional will be an instruction which does dest_f(dest,T), BBFBX. Both BBFB and BBFBX load the X-level registers, but BBFB applies the complement of the source mask to the destination data, BBFBX does not.

To summarize, BBFA does:

1) ALUA_ H1 LeftCycle [SBX-DBX] and MASK (MASK=1's in bits DBX through DBX+MWX)
2) H2_ if MA then not MASK else 0
3) APC_ dispatch value for loop control (see section 3.6.3.1)
4) MNBR_MNBR+MWX+1
5) SB_SB+MWX+1
6) DB_DB+MWX+1

BBFB and BBFBX do:

1) SBX_SB
2) DBX_DB
3) MWX_MIN[(16-SB[2:5]), (16-DB[2:5]), -MNBR]-1

In addition, BBFB (not BBFBX) does:

4) ALUA_H1 and not MASK

3.6.4 Mesa Instruction and Operand Access

The D0 contains logic to buffer eight bytes from the instruction stream and deliver them to the control section as dispatch values, or to the arithmetic unit as data. The data from the instruction stream is held in a four word area in the R memory, and the four-bit register PCF is used to index this area. The least significant bit of PCF is used to control the cycler/masker during a NextInst or NextData function, and PCF[1:2] replaces the two least significant bits of the R address when RMOD=1, RSEL[4:5]=0. PCF[0] is a flag bit which indicates that the instruction buffer has been exhausted.

The function NextData (F=17b) is provided to access the next byte of the instruction buffer in R as an 8-bit operand. It does:

1) If PCF[0]=1, abort the instruction and trap to location 0 (see Section 4.4)
2) ALUA[8:15]_ if PCF[3] then H1[8:15] else H1[0:7]; ALUA[0:7]_0
3) PCF_PCF+1

An instruction containing NextData must specify a CALL in its JC field. Normally, CALLs save (CIA+1 mod 16) in the return link register TPC. When a NextData or NextInst traps, the increment is disabled by the hardware, so the buffer refill microcode can simply RETURN to reexecute the NextInst/NextData.

The NextInst function (F1=16b) is used in the next-to-last microinstruction of a Mesa bytecode. The last instruction of a bytecode is a special RETURN. NextInst starts a dispatch on the next available byte in the instruction buffer (the byte pointed to by PCF). It loads APC with (ALUA or 2001b), *providing that HTASK=0.* If HTASK #0 (because another task is requesting a wakeup), APC is loaded with TPC[HTASK], and the RETURN in the following microinstruction will cause a task switch.

In addition to (conditionally) loading APC, NextInst also loads the 8-bit ByteCode register from ALUA[6:13]. When a RETURN instruction with JA.7 = 1 is executed, the 12-bit quantity (2001b + 4*ByteCode) is written into TPC[CTASK] during Cycle0. This feature is provided so that the RETURN in the last microinstruction of a bytecode can task switch and still preserve the dispatch address generated by the NextInst. When control returns to task 0 after the higher priority task has run, execution will resume at the first microinstruction of the next bytecode.

In addition to providing normal instruction dispatching, NextInst also tests for interrupts generated by I/O microcode. Bit 12 of the RS232 output register (IntPending) is inspected by the logic that generates the mask used by NextInst. If IntPending is true, the mask is forced to 0, which will cause control to go to the code for bytecode 0 (Noop), rather than to the normal next instruction. A branch is provided on IntPending, so that the microcode for Noop can determine whether it got control due to an interrupt or by executing a zero from the instruction stream. The intent is that the I/O controllers wishing to interrupt the Mesa emulator will OR a one into IntPending, which will cause an interrupt at the completion of the next bytecode. The Noop microcode will clear the bit and cause the interrupt. [Note: Bit

12 of the RS232 register is used solely for convenience of implementation. Since this register has other purposes, a copy of it must be kept in R. I/O controllers will OR into the copy, then load the real register from the copy. The Noop code will mask out the bit in the copy, then load the real register to clear IntPending.]

In detail, NextInst does:

1) If PCF[0]=1 abort the instruction and trap to location 0 (see Section 4.4)
2) ALUA[6:13]_ if not IntPending then (if PCF[3] then H1[8:15] else H1[0:7]) else 0;
ALUA[0:5]_ALUA[14:15]_0
3) APC[2:9]_if HTASK = 0 then (ALUA or 2001b) else TPC[HTASK]
4) ByteCode[0:7] _ ALUA[6:13]
5) PCF_PCF+1

Like NextData's, instructions containing NextInst must specify CALL in their JC fields, so that if they trap, they will be reexecuted when the trap handler returns. Because of the arrangement of the NextInst dispatch bits, the microcode for each Mesa opcode must begin at address (2001b + 4*Opcode).

3.6.4.1 PCX

The PCX register is provided to hold the low order three bits of the Mesa program counter for the bytecode currently being executed by the Mesa emulator. If a bytecode causes a trap, this information must be recovered so that the instruction can be restarted when the (Mesa) trap handler has corrected the situation that caused the trap. PCX is loaded from PCF at t2 of all microinstructions located at 2001b + 4*n (n= 0-377b).

The first instruction of a bytecode will be aborted if the memory system has not finished mapping an emulator memory reference. The feature is provided so that the effects of faults will not propagate across instruction boundaries.

3.7 Parity Register

The D0 checks parity on every read of the R memory, and on the microinstruction in MIR [Note: Parity is not checked when the control store is read as data]. In addition, a number of error conditions are detected by the memory. When any of these errors occur, a *fault* is caused, and task 15 gets control as described in Section 4.6.

To allow the fault handling microcode to determine the reason for the fault, a four-bit Parity register is provided. This register is read as an R bus source using RSEL=57b, RMOD=1. The significant bits are:

Bit	Meaning
4	Stack Overflow
5	Control Store Parity Error
6	R memory Parity Error
7	Memory Fault

This register is cleared when the D0 is bootstrapped or when the function Reset Errors (GroupB=1) is executed.

It is not always possible to recover the R address or the control store address that caused the error, since one microinstruction is executed between the one that generated the error and the initiation of the fault.

A memory fault may indicate an error, or may be associated with a page fault or write protect violation. Task 15 must determine the cause of the fault as described in Section 5.5.

If task 15 is active and a fault occurs, the machine is bootstrapped.

3.8 Special Functions

Table 3.8 summarizes the F1 and F2 decodes. With the following exceptions, the F1 and F2 fields are independent and may both be used in a single microinstruction:

1) If BBFA (F1=0) is used, F2 must be 0.
2) LoadPage[F2] uses F2 as a 4-bit constant.
3) Group B (F1=7) selects 16 secondary functions encoded in the F2 field.
4) During memory reference instructions, F1's are not available. In addition, if DF2=1, F2 is used as a 4-bit displacement rather than as a function.

Table 3.8
Function Decodes

Code	F1	F2	Group B
00	BBFA*	RegShift	unused
01	RS232_ H2	Stkp_ ALUA	ResetErrors
02	LoadTimer[ALUA]	FreezeResult	IncMPanel
03	AddToTimer[ALUA]	StackShift/IOStrobe	ClrMPanel
04	unused	CycleControl_ ALUA	GenSRClock
05	LoadPage[F2]	SB_ ALUA	ResetWDT
06	unused	DB_ ALUA	Boot
07	Group B	SpareF2	Breakpoint
10	no-op	BranchShift	APC&APCTask_ ALUA
11	WFA	SALUF_ H2	Restore
12	BBFB	no-op	ResetFault
13	WFB	MNBR_ ALUA	UseCTask
14	RF	PCF_ ALUA	WriteCS0&2
15	BBFBX	ResetMemErrs	WriteCS1
16	NextInst	UseCOutAsCin	ReadCS
17	NextData	Printer_ ALUA	D0Off

*BBFA requires F2 to be RegShift

3.9 Miscellaneous Registers

The D0 processor includes a number of miscellaneous registers, described in detail in this section.

3.9.1 Watchdog Timer

The D0 contains a watchdog timer to ensure that infinite loops cannot hang the machine forever. The timer is a one-shot multivibrator with a time constant of ~100 ms. When the machine is bootstrapped, the watchdog timer is *cleared*. The timer is *started* by the function ResetWDT (GB=5), and once this function has been executed after a bootstrap operation, it must be executed at least every 100ms, or the D0 will be bootstrapped.

3.9.2 Power Monitoring

The D0 processor requires only +5V for its operation, but its power supply provides -5V, +12V, and +24V for the main memory and peripherals. The MISC board contains three sensors for these voltages which indicate that they are approximately correct. The sensors are read to the R bus with RSEL = 37b, RMOD = RegShift = 1. The bits have the following significance:

Bit	Meaning
5	1 => +12 ok
6	1 => +24 ok
7	0 => -5 ok

3.9.3 RS232 Interface

The RS232 interface consists of an output register, an input register, and level converters. All modem control functions will be implemented in microcode, although the hardware provides assistance in timing the delivery of the primary serial data stream. Connection to an external modem is made via a 25pin male D-shell connector on the rear edge of the Misc board. The output register is loaded from H2 by the function RS232 _ H2 (F1=1). Bits 8 through 10 of the output register are transmitted directly on the lines (after level conversion), but bit 15 is latched as NextBitToGo and transmitted at the next *Send* signal generated by a timer (see section 3.10). The drivers are arranged so that a "1" in H2 corresponds to a negative voltage (mark, or control function OFF) on the lines. Bit 11 of this register is used for the signal SetTime (see 3.9.6), and bit 12 is the signal IntPending (see 3.6.4). Bits 0-7 and 13-14 are not used.

The relationship between bits in H2, RS232 standard signal names, and connector pins is as follows:

Bit	RS232 Name	Connector Pin Number
8	Request To Send	4
9	Spare	11
10	Data Terminal Ready	20
15	Transmitted Data*	2

*Synchronized by timer Send function

RS232 input is handled similarly. The received data is sampled by a timer, and the sampled bit and all other modem status signals are read directly onto the R bus with RSEL=37b, RMOD=RegShift=1. The receivers are arranged so that a low voltage (mark) on the line is read as a "1" on R. The relationship between R bits, signal names, and connector pins is:

Bit	RS232 Name	Connector Pin Number
8	Received Data*	3
9	Clear To Send	5
10	Data Set Ready	6
11	Carrier Detect	8
12	TXClock	15
13	RCVClock	16
14	LoopBack for Transmitted Data	
15	Ring Indicator	22

*Latched by timer Sample function

In addition to being read onto R, the transmitter and receiver clocks (pins 15 and 16) are also used by the timer logic to sample and send the transmitted data to a synchronous modem, as described in section 3.10.

3.9.4 Printer Interface

The printer interface consists of eight output lines, five input lines, and eight bidirectional tri-state lines. Connection is made to a 37-pin female D-shell connector on the rear of the Misc board.

The output register is loaded from ALUA[0:15] by the function Printer _ ALUA (F1=17b). The most significant eight bits of this register are sent directly on the lines in pseudo-differential form (the complement of each bit is also sent, both at TTL levels). Bit 7 of the output register controls the gating of register bits 8-15 onto the bidirectional data bus. If this bit is 0, the outputs will be enabled to the bus. The correspondence between ALUA bits, signal names, and pin numbers on the 37-pin connector is:

ALUA bit	Signal Name	Pin
0	PO.0	20
	PO.0’	21
1	PO.1	22
	PO.1’	23
2	PO.2	24
	PO.2’	25
3	PO.3	26
	PO.3’	27
4	PO.4	28
	PO.4’	29
5	PO.5	30
	PO.5’	31
6	PO.6	32
	PO.6’	33
7	PO.7	34
	PO.7’	35
8	PD.0	1
9	PD.1	2
10	PD.2	3
11	PD.3	4
12	PD.4	5
13	PD.5	6
14	PD.6	7
15	PD.7	8

The thirteen input lines are read onto the R bus using RSEL=27b, RMOD=RegShift=1. The most significant five bits are simple receivers, the least significant bits are the signals PD.0-PD.7 (the bidirectional bus). The pin connections for the most significant bits are:

R bit	Signal name	Pin
0	PI.0	9
1	PI.1	10
2	PI.2	11
3	PI.3	12
4	PI.4	13

The cable connector has logic ground on pins 17,18,19,36, and 37. Pins 14 and 15 are connected to +12 volts through a 10 ohm, .25watt resistor. An LED is connected across this resistor so that if excessive current is drawn by external logic, the LED will light. The intent is to use this voltage to supply a small amount of logic in external interfaces.

During system checkout, the printer interface is used to pass data between the D0 and the Alto used to operate it. A communications protocol for the Alto program (Midas) and a D0 microprogram (the Kernel) has been designed which supports this communication and also allows the Alto to bootstrap the D0. To perform the bootstrap function, pin 11 (PI.2) is supplied to the control board as TesterBoot’. Since the machine is bootstrapped when this signal is made low, no normal interfaces should make use of this signal.

The signals PI.0 - PI.4, and PD.0 - PD.7 are terminated with 220 ohms to +5, 330 ohms to ground.

3.9.5 Maintenance Panel

The maintenance panel is provided to display the results of diagnostic tests when more sophisticated I/O facilities are malfunctioning, or during the bootstrap process. It consists of a four-digit LED display mounted on the front of the D0 cabinet. Two functions are provided to display information: ClrMPanel (GB = 3) clears the display to 0000, and IncMPanel (GB = 2) increments the contents of the panel by one. Since the panel is implemented with slow circuitry, these functions should not be issued more frequently than every TBD microinstructions.

The maintenance panel also contains the power control logic for the system. AC power is controlled by a solid state relay. This relay is operated by a START and a STOP button on the panel. When START is pressed, the SSR is activated. It is latched on by the D0 +5V supply. As long as START is depressed, the maintenance panel display is incremented at high speed, causing it to display 8888 (lamp test). Approximately 150ms. after START is released, the processor is booted (PanelBoot). If START is depressed when power is on, the machine is also booted. When STOP is depressed, or when the function D0Off (GB = 17b) is executed, AC power is turned off. [Note: D0Off is integrated by the power-off logic to reject noise. The correct instruction to turn the system off is "D0Off, Goto[.]"]

The maintenance panel contains an unswitched pilot power supply to operate the SSR and to charge the battery used by the TOD clock. When this supply is on (indicating that the D0 has AC power), the decimal point adjacent to the units digit of the display will be lit.

Since hard R or Control Store parity errors will render the processor completely inoperative (and unable to display failure information in the display), these conditions are handled specially. When either of these bits are on in the Parity register, the panel is incremented rapidly (by the hardware), and the center bar of the units and tens digits are disabled. Control store parity disbles the units digit, displaying 8880, R parity disables the tens digit,

displaying 8808.

3.9.6 Time of Day Clock

The maintenance panel contains a time-of-day clock implemented with a TI TMS1000C (CMOS) microprocessor with a crystal controlled clock. The clock is powered by a battery which is charged from the pilot supply whenever the system has AC power. The battery will operate the clock for approximately one week with power disconnected.

Whenever the D0 has DC power, the clock emits the time as a serial 32-bit number (at 1ms per bit) once per second. There is a branch condition in the D0 to test the state of this line (TimeOut). The standard microcode will provide conversion to some suitable format and an interface to Mesa programs.

The clock is set by sending it a 56-bit quantity as a serial bit stream. Bit 11 of the RS232 output register (SetTime) is reserved for this purpose.

In addition to providing time of day, the clock also accumulates total power-on hours for the system, and can be set to turn AC power on and boot the system at a predetermined time. Details of the clock are provided in Appendix A.

3.10 Timers

The D0 processor contains logic implementing high resolution timers at the microcode level. Up to sixteen timers can be established. A wakeup to the timer task (task 14) occurs when any of the timers expires. A timer consists of one or more *slots* in a sixteen word Timer RAM located on the CPU Misc board. Every four machine cycles, the timer control cycles into the next timer slot. The slot contains a four bit state and an eight bit data field. A state machine calculates a new state and data field value based on the contents of the slot (and possibly on other conditions), rewrites the contents of the RAM, and cycles into the next slot. In general, the action taken for the timer is to decrement the data field and cause a task wakeup when the field is less than zero. The timer continues to decrement even after the wakeup is requested, so that accurate timing can be maintained with varying wakeup latency. The D0 can load a timer with an initial value to be counted down, or it can add a value to the data. The add function is used to implement wakeups at precise intervals by utilizing the count after wakeup feature. *Loading* the data field causes a wakeup at an interval relative to the time that the *load* was done. *Adding* to the field causes a wakeup at an interval relative to the last time the *counter underflowed*.

Since the timer RAM has 16 slots, and a new slot is accessed every four cycles, the resolution of the timer is 64 cycles. The range of a simple timer is 128*64 cycles or 8192 cycles with a 100% overrange. For timers requiring more range, two or more slots can be cascaded. When the first slot decrements from -1 to -2, underflow of the timer is saved in a Carry flag. The slot immediately following can conditionally decrement its data field based on Carry. The wakeup can be generated on the sign bit of the last slot's data.

The timer mechanism also provides bit timing for RS-232 communications. The hardware can be programmed to *Sample* the received data bit (or *Send* the transmitted data bit) and cause a wakeup at appropriate intervals for both asynchronous and synchronous protocols. The hardware provides double buffered bit-at-a-time access to the RS-232 data.

Table 3.10 lists the 16 states a timer can be in, and the changes to the data and state fields on each round. There are two functions (LoadTimer and AddToTimer) that load a timer slot from the ALUA bus. The 16 bits of the ALUA are divided as follows: Bits 0-3 are the new state (loaded by both LoadTimer and AddToTimer), bits 4-11 are the new data (or data to be added to the current data) and bits 12-15 are the slot number to be loaded. The timer is permanently assigned to task 14. When any timer requests a wakeup, the state and data fields of the slot, along with its slot number are loaded into a register which can be read as an external R source (Timer), and a wakeup is requested. When the microcode reads the Timer register, the wakeup request is cleared. Timers which expire while a wakeup request is pending for some other slot remain in their current state until the register is read (but they continue to decrement). Thus the output register is a resource which can be held by only one slot at a time.

Table 3.10

State	Action	Use	Sel
0	if Carry then Data_Data-1,Carry_(Data=0&Carry)	Middle	Carry
1	if Carry then Data_Data-1 if Data<0 & not WakePending then [Wakeup;State_4]	Most Significant	Carry
2	if Carry then Data_Data-1 if Data=0&Carry then Send if Data<0 & not WakePending then [Wakeup;State_4]	Slow Async Xmit	Carry
3	if Carry then Data_Data-1 if Data=0&Carry then Sample if Data<0 & not WakePending then [Wakeup;State_4]	Slow Async Rcv	Carry
4	do nothing	Idle	
5	Data_Data-1; if Data<0 & not WakePending then [Wakeup;State_8]	Simple Timer	
6	if RD=0 then State_13	Fast Async Start Bit	RD
7	if RD=0 then State_9	Slow Async Start Bit	RD
8	Data_Data-1	Wait for Reload	
9	Data_Data-1;Cry_Data=0	Least Significant	
10	if XC=0 then [Send;State_1]	Sync Xmit	XC
11	if XC=1 then State_10	Sync Xmit	XC
12	Data_Data-1 if Data=0 then Send if Data<0 & not WakePending then [Wakeup;State_8]	Fast Async Xmit	
13	Data_Data-1 if Data=0 then Sample if Data<0 & not WakePending then [Wakeup;State_8]	Fast Async Rcv	
14	if RC=1 then [Sample;State_1]	Sync Rcv	RC
15	if RC=0 then State_14	Sync Rcv	RC

Here is how they are used:

A simple *short timer* starts in state 5, decrements until it goes negative, then requests a wakeup. When the wakeup is accepted, it switches to state 8.

A *two stage timer* starts in state 9. The following slot is in state 1. When the first slot causes Carry, the following slot decrements. When the second slot underflows, a wakeup is

requested. When it is accepted, the second slot's state goes to state 4.

A *multiple stage timer* is like a two stage timer except that additional slots in state 0 are placed between the first and the last slots (which are initialized to states 9 and 1 respectively).

A *fast asynchronous RS-232 receiver* starts in state 6, looking for the start bit. When it occurs, it switches to state 13, and changes into a timer. When the timer elapses (one half of a bit time later), the line is sampled, and a wakeup occurs (switching to state 8 when accepted). If the sample is still a 0, the start bit is found, and the program adds one bit time to the timer, resetting it to state 13. The timer then times out to the middle of the next data bit and wakes up again. The RS-232 data bit can be read as one of the bits of the RS-232 interface.

A *slow asynchronous RS-232 receiver* initializes one slot to state 7, and the next slot to state 3. When the start bit arrives, the first slot changes to state 9 and starts counting down. The next slot counts carries in state 3 until it gets to 0, whereupon it samples the line, and wakes up. Succeeding bits are timed by a state 9/3 slot pair.

A *fast asynchronous transmitter* uses state 12. The data bit (NextBitToGo) is double buffered. NextBitToGo is part of the RS232 output register, loaded from H2 by the function RS232_.

A *slow asynchronous transmitter* uses a 9/2 pair of slots.

A *synchronous RS232 receiver* waits for the low to high transition of the receive clock (RC) in state 14, then samples the data and goes to state 1. The data field of the slot should be initialized with a negative value, which causes an immediate wakeup. The program resets it to state 15, where it waits until it detects the high to low transition of the clock, and then reverts to state 14.

A *synchronous RS-232 transmitter* uses states 10 and 11 like the receiver uses 14 and 15.

The state numbers are arranged so that the most significant 2 bits address a 4 input multiplexer wired so that its output ("Sel" in the table above) reflects the Carry bit (Carry) for states 0-3, the Receive Data line (RD) for state 4-7, the transmit clock (XC) for states 8-11 and the Receive Clock (RC) for states 12:15. The state machine has 8 inputs: 4 state bits, the output of the Sel multiplexer, Data=0, Data<0 and WakePending. It outputs a new state, the control signals to sample the RS232 input line, send the next RS232 output bit, request a wakeup, set the Carry bit, and to decrement the data field.

4.0 Control

4.1 Normal Instruction Sequencing

"Normal" instructions are those which do not do conditional branches, dispatches, subroutine calls or returns. Normal instructions are coded with a Goto (JC = 4) in the *Jump Control* field of the microinstruction. These instructions calculate the address of their successor by concatenating the contents of the four bit *Page* register with the 8 bits of the *Jump Address* (JA) field in the current microinstruction. Thus a normal instruction's successor can be any location within the current 256 word page. Crossing pages is done with the function LoadPage[F2], which loads the Page register with the F2 field at t2. Accessing the Control Store for fetching the next instruction is overlapped with execution of the current instruction. Since the successor of an instruction which loads Page is fetched before the Page register is modified, the next instruction will be on the current page, its successor will be on the new page.

```
            ....,LOADPAGE[3],GOTO[X];   *Instruction on page n
X:          ....,GOTO[Y];               *Also on page n
Y:          ......;                     *Y is on page 3
```

At t2, MIR is loaded with the instruction, the *Current Instruction Address* register CIA is loaded with the address of the instruction, and execution of the instruction begins. Execution usually requires four cycles, designated cycle0-cycle3, but another instruction is started at t2 (at the end of cycle1). Figure 2.5 illustrates the timing of instructions.

4.2 Conditional Branches

Branches are two-way tests which send control to one of two locations whose addresses differ only in their low order bit. Branch conditions are encoded in the JC[1:2],,JA[07] field. There is a function (BranchShift, F2=10b) which changes the interpretation of the JC/JA field, doubling the number of available branch conditions. Table 4.2 summarizes these conditions. When a branch is specified in an instruction, the condition becomes bit 11 of the next instruction address. When the branch condition is true, instruction execution is extended by one cycle as described in section 2.4.

Table 4.2
Branch Conditions

JC[1,2],,JA[7]	BranchShift	Condition
0	0	ALU#0 (Tests ALU result from previous instruction)
1	0	Carry (Tests ALU result from previous instruction)
2	0	ALU<0 (Tests ALU result from previous instruction)
3	0	NoH2Bit8 (Tests value placed in H2.8 by the current instruction)
4	0	R<0 (Tests value placed in H1.0 by the current instruction)
5	0	R Odd (Tests value placed in H1.15 by the current instruction)
6	0	NoAttn (Flag from I/O controllers)
7	0	MB (bit in SALUF)
0	1	IntPending (Interrupt test)
1	1	NoOvf (Tests ALU result from previous instruction)
2	1	BPCChk (Tests PCF[0])
3	1	Spare
4	1	QWO (Tests result of Pfetch2/Pstore2 in the previous instruction)
5	1	TimeOut (Tests data from TOD clock)

4.3 Subroutines and Tasking

JC = 5 specifies a subroutine Call. The address of the successor is calculated exactly as in a Goto, but in addition, (CIA+1) is written into the *Task Program Counter* (TPC) RAM location for the current task during cycle 0. [Note: Only the low four bits of CIA are incremented.] This will be the *Return Address* for the subroutine.

At t2 of every instruction, the *Alternate Program Counter* register APC is loaded with TPC[HTask] except as noted below. HTask is the number of the highest priority task requesting a wake-up. The read of TPC is done during cycle1. If CTask = HTask, the return address will be loaded into APC at t2. If the next instruction specifies a *Return* (JC=6), the address for its successor will be the contents of APC. Since the write of TPC with the return address is done during cycle 0 of the call, and the read of TPC is done in cycle1, the first instruction of a subroutine can do a Return. Thus one instruction subroutines are allowed.

If CTask is not the same as HTask, APC will not contain the return address for the task which is running, but will have a return address for the highest priority requesting task and control will pass to the task whose return address *is* in APC. Thus a subroutine return can cause a task switch. When APC is loaded with TPC[HTask], the APCTask register is loaded with HTask. When the Return is executed, CTask is loaded from APCTask, thus completing the TASK function.

All subroutine returns will cause a TASK unless the function UseCTask is executed in the instruction immediately preceding the Return. UseCTask forces the read of TPC to be TPC[CTask] instead of TPC[HTask], thus APC will be loaded with the return address of the current task irrespective of the wake-up logic. UseCTask also loads CTask into APCTask so that the task number is not changed with the Return. APCTask always contains the task number corresponding to the contents of APC.

Subroutine return is the only way to cause a TASK. Long sequences of instructions which do not call subroutines can do TASKs by replacing two normal instructions (with Gotos) into a Call/Return pair.

```
Without Task:                      With Task:
X:   ......,GOTO[Y];               X:   ......,CALL[Y];
Y:   ......,GOTO[Z];               X+1:       .....;       *Returns to here
Z:   .....;                        Y:         .....,Return;
```

Multi-level subroutines can be done with explicit save/restore of the return address in an R location. The APC and APCTask registers can be read as an external R source (see Table 3.1). The UseCTask function forces APC&APCTask to be loaded with TPC[CTask]&CTask. This combination is used to save the return address. Another function is available to load APC&APCTask with the data on ALUA, which can then be used as the return address. This explicit loading of APC and APCTask overides the normal TPC[HTask] load. The coding for this is as follows:

```
         ......,Call[Subr1];     *1st level call
         .....;                  *1st level returns to here

Subr1:   .......;
         ......;
         .....,UseCTask;         *Forces APC to be loaded with return address.
         T_APC&APCTask;          *Save return address in T
         R_T,Call[Subr2];        *Save return address in R and call 2nd level
         .....;                  *2nd level returns to here
         .....;
         APC&APCTask_R;          *Load saved return address into APC
         ....,Return;            *Return to main level

Subr2:
         ....;
         ......,Return;          *Return to 1st level
```

The Page register is loaded from the top 4 bits of APC when a Return is executed. Cross-page subroutine calling can be done with the LoadPage[F2] function.

```
.....,LOADPAGE[3];          *Caller on page n
......,CALL[SubrOnPage3];   *Subroutine on page 3
.....;                      *Returns to page n
```

4.4 Dispatches

Dispatches provide a way of doing a multi-way branch. When a dispatch is specified, the value (usually ALUA) is loaded into APC at t2, and the bits involved replace the Page/JA bits for the next instruction. Dispatches are specified in the following ways:

1) A number of the short field operations extract a field from an R location or other R bus source, and load it into APC. The instruction following this must contain a Dispatch directive (JC=7). This will cause the lower 4 bits of the address to be taken from APC, with the remaing address bits coming from Page[0:3],,JA[0:3]. This results in a 16 way (or fewer) branch.

2) The NextInst function loads APC with the start address of the microcode for each of the 256 Mesa

bytecodes. The instruction following the NextInst must specify a Return, and all 12 address bits will come from APC (See 3.6.4).

3) The function BBFA does a dispatch based on the number of bits remaining in the current quadword buffer in R. The sucessor of the BBFA must specify Dispatch in the JC field.

4.5 Aborted instructions

There are several reasons for aborting an instruction:

1) A NextData or NextInst operation attempts to read a byte from the quadword indexed by PCF when PCF[0]=1.
2) The memory is busy and a memory reference instruction is attempted.
3) An R location for which a memory reference is pending is specified in an instruction.
4) A Fault occurs.

In general, aborting an instruction means that all irreversible actions that the processor would have taken will not be done. The net effect is that the instruction is not executed at all, including the load of MIR. Thus the instruction will be re-executed, since MIR did not change. There are two exceptions to this: The first is that TPC is written with CIA + 1 if the aborted instruction specifies a Call. This does not cause any ill effects.

The other exception is that in case 1 above, MIR is loaded, not with the successor specified by the instruction, but rather with ControlStore[0]. In addition, CIA is written into TPC. This is esssentially an unanticipated subroutine call to location 0 (a trap). The trap routine starting at location 0 refills the instruction buffer and restarts the instruction by executing a Return.

In cases 2 and 3 above, all register loads are inhibited, which will cause the instruction to be repeated endlessly until the memory becomes free or the R location is filled or used by the memory.

4.6 Faults

A fault occurs when the memory detects a page fault, write protect violation, or input data parity error, when an internal bus parity error is detected by the processor, or when the Breakpoint function is executed. Faults are expected to be infrequent, and are handled by a special mechanism since they occur asynchronously with respect to normal microinstruction sequencing.

When a fault occurs, the current instruction is aborted and the signal *Fault* is generated at t0. The next address logic is disabled, and a fetch of location 1 is forced. Fault causes the loading of CIA to be inhibited, thus freezing the location of the instruction which was executing when fault occured. A similar action freezes the Result register. There are two versions of the current task register which normally have the same contents, but become different while Fault is asserted. The internal CTask register, InCTsk, remains set to whatever task number was running when fault occurred. The InCTask register may be read as an external R bus source, but CTask cannot. CTask is used to modify the R addresses and to address T, and is set to 15 by Fault. Fault also disables the Page register's

contribution to the control store address, which forces all instructions of the fault handling microcode to be on page 0 until the ResetErrors function is executed. [Note: There is a ResetFault function, which only resets Fault, while the function ResetErrors both resets Fault and clears the Parity register. Since the Parity register must be cleared before Fault is reset (to avoid another fault), ResetFault is obsolete and should not be used.]

The fault trap code starting at location 1 must first save APC&APCTask since it will be overwritten at t2, and then save CIA&CTask (the Internal CTask) as well as the Result and Parity registers. If the fault was due to a stack overflow, the code must set the stackpointer to a legal value (not 0 or 11-17b). Then the fault code does a Notify (see below) to force task 15 to run, accompanied by the ResetErrors function. ResetErrors turns off Fault, clears the Parity register, and releases the freeze on CIA Result., and the Page register. At this point, Task 15 is running normally and can examine all the saved state to determine what action needs to be taken. When task 15 is finished, it returns to the running program using the APC&APCTask _ ALUA function as well as the function Restore. Restore loads APC&APCTask with ALUA and loads the Result register with H2.

The return sequence is:

```
T_SavedResultRegister;
APC&APCTask_SavedCIA&CTask;
APC&APCTask_SavedAPC&APCTask,Restore,Return;
```

The first instruction presets T to the saved condition code. The second loads the desired return address and task number into APC. The third specifies Return, thus the next instruction address will come from APC, and CTask will be loaded with APCTask. At the same time as the fetch of the instruction that was aborted when the fault started, APC and APCTask are loaded with the value that they had before the fault, and the contents of the Result register are restored at t3.

The fault handling code cannot do anything which could result in a fault. In particular, it cannot do any memory references, but must send messages to lower priority tasks to do any logging or recovery required as a result of the fault.

4.7 Notify

Notify is used to start another task at an arbitrary location. It is done by loading APC and APCTask with the address and Task number desired, followed by a Return directive. The next instruction after the return will be in the new task, regardless of HTask.

4.8 Writing and Reading Registers

APC and APCTask can be written with a function, and read as an external R source. CIA and CTask cannot be written explicitly but can be read as an R bus source (CIA is read in complement form). Page can be loaded with a function and read as an external R bus source. TPC cannot be written explicitly except by doing a Call. TPC for the current task can be read by doing UseCTask, followed by reading APC. TPC for another task can be read only by notifying the task at a location containing code to read TPC, then notifying back to the original task.

4.9 Reading and Loading the Control Store

Data for writing into the control store comes from a 16 bit register CSIn which is loaded from ALUA at t2 of every instruction which does not load APC, and from a 4 bit extension to that register which is loaded from H2 every t2. There are two functions which write the three sections of the control store as follows:

T _ Data for bits 32-35;
ALUA _ Data for bits 00-15;
APC _ Address;
WriteCS0&2;
ALUA _ Data for bits 16-31;
APC _ Address;
WriteCS1;

There is an additional register CSData, which can be read as an external R bus source, which is used in reading the control store. The ReadCS function reads the control store location whose address is in APC (same as write). The contents of H2 during the read determine which of the 3 sections of control store data are loaded into CSData. A 0 in H2 causes bits 0-15 to be read into CSData. A 1 causes bits 16-31 to be read, and a 3 causes bits 32-35 to be read into CSData[00:03].

When an instruction is loaded into the control store, the bits RSEL.0 and RSEL.1 must be complemented.

The read and write operations require two more cycles than an ordinary instruction, since these operations must address two locations in the control store (the location to be read or written and the next instruction). An internal state bit, CSOp, is provided to stretch the write and read operations appropriately.

CSOp is cleared at t2 of every instruction which does not include the functions WriteCS0&2, WriteCS1, or ReadCS. In instructions which include one of these functions, CSOp is complemented at t2.

When a control store function is done, if CSOp = 0, the effects are:

1) The write or read operation is done as specified, using APC as the control store address.
2) Loading of MIR, CIA, and CTask is inhibited
3) CSOp is set

If CSOp = 1 when a control store function is done,

1) The write or read operation is inhibited
2) The successor instruction is fetched, using Page,,JA for the address.
3) MIR, CIA, and CTask are loaded normally
4) CSOp is cleared

The single write or read microinstruction is thus executed twice, with the read or write taking

place during the first execution. Microinstructions that read or write the control store must be coded with a RETURN in the JC field, but unlike other RETURNs, the JA field is significant. For this reason, the successor of an instruction that reads or writes the control store must be placed at an even location unless the contents of the return link (TPC) are unimportant. This is because RETURNs with JA.7=1 write into TPC (see 3.6.4).

4.10 Bootstrapping

The machine can be bootstrapped by the START button or by the TOD clock, from the test hardware, from the watchdog timer, by a function (Boot), or when a parity error occurs while task 15 is running. When the boot signal is asserted, a small state machine takes control and loads 1024 microinstructions into the Control Store from a 2k x 16 EPROM. Control is then transferred to location 0. Since the boot ROM is 32 bits wide, while the control store is 36 bits wide, 4 bits of control store do not come from the ROM. Instead, the are loaded with a constant. The bits are:

> RSEL4 and RSEL5, which are forced to be 1's. This restricts boot code to only 16 R locations, but allows reading of external R sources.
>
> JA[0:1], which are forced to 00. This means that boot code occupies only the first 64 locations of each page of the control store.

The boot state machine uses APC as the source of address for loading the control store, and APC is implemented as a counter. The counter bits are wired so that APC[4:11] counts modulo 64, then increments APC[0:3]. At the time the machine is bootstrapped, the BootReason register (Table 3.1) is loaded with the reason for the boot, so that the microcode can take special action based on this information if it wishes.

5.0 Memory

5.1 Organization

The memory is organized as a pipeline composed of two stages, MC1 and MC2. Each stage has a microprogrammed controller which governs the activities of the stage, and a number of data registers which are managed by the controller. Figure 2.3 shows the data paths associated with the memory.

The controller for MC1 contains 256 40-bit control words, the controller for MC2 contains 256 20-bit words. Most of the complexity of the memory system is contained in these controllers, and for this reason, their microcode, annotated with comments, is included as Appendix B.

MC1 is responsible for mapping and for controlling main storage cycles. It controls the following items:

- Map memory cycles
- Main storage cycles
- R memory cycles (but R memory writes by MC2 have priority)
- Loading of the storage card data input registers (from H4)
- Loading of the storage card data output registers (from the RAM chips)
- The Idata bus and the H3I register
- The IMux and the H4 register

MC2 is responsible for the transport of fetched data from the storage card data output registers to the appropriate destination. MC2 controls:

- Reading of the storage card data registers
- The H3U register
- The ECC buffer
- The H3C register
- The Odata register
- R write cycles

5.2 Memory Reference Instructions

Memory reference microinstructions have special timing and a special format (see figure 2.5) in which the ALUF, BSEL, LR, LT, and F1 fields are replaced by a TYPE and SRC/DEST field. Once a memory reference has been started by executing a memory reference microinstruction, the processor and the memory operate in parallel. The memory will eventually transfer data from the source to the destination indicated by the instruction which started the reference. A single memory reference instruction specifies all the information required to complete a reference, including the reference type, the source and destination for the data, and the virtual address.

5.2.1 Reference Types

The D0 provides fifteen different memory operations, selected by the TYPE field of the microinstruction. They are:

PFetch 1/2/4, PStore 1/2/4:
These operations transfer single, double, or quad words between memory and R. The formation of the memory address is described in section 5.2.5. When multiword operations are specified, the least significant bit(s) of the address are ignored by the hardware, so the data for an n-word transfer must be aligned modulo n in memory. An exception to this rule occurs when doublewords are transferred to or from the stack. Unless the doubleword crosses a quadword boundary, the reference is done as specified, even if the words are not aligned.

The formation of the initial R address is described in section 5.2.2. When more than one word is transferred, sequential R locations are used. It is not essential that R locations be double- or quad-aligned for the memory to transfer correctly, but if they are not, the R interlock mechanism (section 5.3) may be defeated.

Input, Output:
These operations transfer single words beween an I/O device register (section 5.2.4) and an R register.

IOFetch 4/16, IOStore4/16:
These operations transfer four or sixteen word blocks between memory and an I/O device register. The data must be quad- or hex-aligned in memory.

XMap:
This operation reads and writes the map. It calculates the virtual address in the normal manner, but does not cycle the main memory. Instead, it writes the contents of the inital R location specified by the instruction into the map entry corresponding to the virtual address, then writes the original contents of the map entry into the next three R locations. The format of the data in these registers is shown in figure 5.2.1.

ReadPipe:
See section 5.5

Refresh:
See section 5.7

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	R Address:
Data Written To Map	LOGSE	WP	DIRTY	REF	REAL PAGE NUMBER												n
Data Read From Map									BLK.1 *	MAIN ROW ADDRESS *							n+1
									BLK.1 *	MAIN COLUMN ADDRSS *							n+2
									LOGSE *	WP *	DIRTY *	REF *	CARD*			BLK.0 *	n+3

* Read in complement form

Figure 5.2.1 XMap R Register Format

5.2.2 R addresses

The ten operations that transfer data to or from the R memory generate an initial R address at t2 of the memory reference instruction. This is the address that will be used for the first R access done by the operation. If the operation transfers more than one word, sequential R locations will be accessed.

The initial R address is (SRC/DEST or CTASK*16) if SRC/DEST#0. CTASK is ORed into the high four bits so that up to four I/O controllers of the same type operating at four task levels (0,1,2,3 mod 4) can share microcode while using different groups of up to 16 R registers per controller.

If SRC/DEST=0, Stkp (possibly incremented or decremented by 1) is used for the initial R address. The amount by which Stkp is incremented or decremented to form the initial R address is determined by the operation type and the number of words involved so that the PFetch1/2, PStore1/2, Input and Output operations will do pushes and pops to the Mesa stack. Stkp itself is also modified by these operations. Table 5.2.2 shows the initial R address and the amount by which Stkp is changed for each operation (n is the initial value of the Stkp).

Table 5.2.2

Operation	Final Stkp	Initial R Address
Pfetch1	n+1	n+1
Pstore1	n-1	n
Pfetch2	n+2	n+1
Pstore2	n-2	n-1
Input	n+1	n+1
Output	n-1	n
All Other	n	n

5.2.3 Quadword Overflow

When a PFetch2 or PStore2 is done to the stack, the hardware inspects the low order two bits of the virtual address. If these bits are 11, indicating that the reference crosses a quadword boundary, the memory reference is not done, and the flag QWO is set (QWO remains set until the next memory reference is started). A branch condition exists to test QWO in the microinstruction following the memory reference, so that the operation can be reexecuted as two single word operations. When quadword overflow occurs, the stackpointer is still updated as shown in Table 5.2.2.

5.2.4 I/O Register Addresses

The operations that transfer data to and from I/O devices generate an I/O register address at t2 of the memory reference instruction. This address is latched in the Iaddr register, and sent to the controllers on the backplane. For the IOFetch and IOStore operations, this address is (SRC/DEST or CTASK*16). For Input and Output, the address is (H2[8:15] or CTASK*16). H2 is normally loaded at t1 from T, but if DF2=1, H2 is loaded from the F2 field of the instruction. If DF2=0, F2 may be used normally, as may the JC and JA fields of the instruction.

5.2.5 Address Calculation

The 24-bit virtual address required by the memory is the sum of a 16-bit displacement D[0:15] and a 24-bit *base pointer* which is contained in a pair of R registers. Figure 5.2.5 shows the formation of an address. The base registers must be set up in advance such that if BP[0:23] is a base pointer, BP[8:23] is held in register r, and R register (r or 1) holds BP[0:7] in bits 0-7, and BP[0:7]+1 in bits 8-15. This arrangement makes it possible to do R addressing in the same way for memory and non-memory microinstructions, and allows the 24-bit address to be calculated with a 16-bit rather than a 24-bit add. Since a number of references will usually be done for each base register load, the calculation required to set up a base register should be insignificant.

The processor calculates the virtual address in two steps. During cycle0 of a memory

reference instruction, H1 is loaded from R[r] (which contains BP[8:23]), and H2 is loaded with the displacement. The displacement is taken from T if DF2=0, from F2 if DF2=1 (F2 is placed in bits 12-15, and the other 12 bits are zeroed). The R address used to access r is formed in a similar manner for memory and non-memory instructions, except that the RMOD bit of the instruction has another meaning (DF2) in the memory case, so it is not possible to select a base register pair using an indexed R address or with the stackpointer.

During the second cycle, H1+H2 is calculated in the ALU, and R[r or 1] is simultaneously accessed. The carry from the 16-bit addition is used to select the left or right byte of register R[r or 1] (BP[0:7] or BP[0:7]+1) for delivery to the memory controller. Note that if RSEL[5]=1 during a memory reference instruction, the odd base register will be used for *both* halves of the base pointer. This feature is useful when calculating the address to be used to refresh the memory, but is probably not useful otherwise.

At the end of the instruction (at t2), the calculation of the virtual address will be complete, and the memory will be started (unless the instruction is aborted for some reason). Note that there is no way for a memory reference instruction to specify a write into R or T, so no part of a memory reference instruction overlaps the following instruction.

If the instruction preceding a memory reference specifies a write of R or T, the data is held in H3P until the first non-memory instruction is executed, at which time H3P will be written into R or T (during cycle1). This introduces several programming restrictions which must be observed for proper operation:

1) An R register specified as the source of data for a Pstore4 must be written before the instruction preceding the memory reference, since the processor may not be able to deposit the data in R before the memory controller needs it. This is also true of the source of data for a PStore1/2 if the PStore might be followed immediately by another memory reference, e.g. as the result of a task switch.

2) At least one non-memory instruction must be executed between the loading of the odd R location of a base register pair and a memory reference instruction which uses the base pair, since the loading of R will be deferred until after the memory reference instruction, and bypassing does not work properly in this case.

3) During cycle0 of a memory reference instruction, R and T bypassing are done in the same way they are for a non-memory instruction. This means that if the even base register or T (the displacement) was loaded by the preceding instruction, the output of the ALU, rather than R or T, will be used to load H1 or H2. If the memory reference instruction is not aborted, these values are correct and all will go well, but if it is aborted, the output of the ALU will be the sum of the even base register and the displacement when the memory reference is finally executed. This quantity will be added to the base or displacement, yielding an incorrect virtual address. There is special logic in the processor which keeps H2 from being loaded in the cycle0 following an aborted instruction, which eliminates the problem for the displacement from T, but there is no such logic for H1. Accordingly, it is illegal to load the even base register in the instruction immediately preceding a memory reference unless it can be guaranteed that (1) the memory reference will not be aborted, or (2) the memory reference uses DF2 addressing with a displacement of zero (in this case, bypassing occurs, but the output of the ALU is the sum of the even base register and zero, which gives the correct result).

4) If several memory references are done with no intervening non-memory instructions, the comments above apply to the base registers used by all the references, since register writes are pipelined across

all the references (even references done in another task if a task switch occurs).

5) If an R register or T is loaded in the instruction preceding a memory reference, it cannot be read in the instruction immediately following the reference, since the write will not be done until cycle1 of the read instruction, but bypassing will be invoked during cycle0. The bypassed value is not the value written, but the result of the base register addition done by the memory reference.

Usually, this is a disadvantage, but there is one circumstance where the results are beneficial. This is the "bypass kludge".

Assume that a base register is set up pointing to a structure, and it is necessary to reference the nth item of the structure, then to modify the base register to point to this item (examples are a base register holding a program counter during a JUMP instruction, or a base register used to chase down a pointer chain). The following code provides this function efficiently:

```
T _ n;                            *displacement
PFetch1[Base, dest];              *fetch the item
Base _ T, goto[.+2, nocarry];     *update the low base register
BaseHi _ BaseHi + 400c + 1;       *update the high base if 64k boundary crossed
       .
       .
```

This code works because T was loaded immediately before the memory reference, and the write is piped across the reference. When the instruction Base_T is executed, bypassing is invoked, and the data written into Base is (Base+n). Also, the carry flag holds the result of the Base+n addition, so a test for 64k boundary crossing can be done, and the high half of the base register updated if necessary.

In summary, the following are illegal:

```
Base _ x;
BaseHi _ y;
PFetch1[Base, dest];      *high base loaded in preceding instruction

T _ Displacement;
BaseHi _ y;
Base _ x;
PFetch1[Base, dest];      *low base loaded - displacement nonzero

Source _ x; *this will occur after the PStore1 is done...
PStore1[Base, Source], return; *and the next instruction may be a memory reference in
another task
```

The following are legal:

```
T _ displacement;
PFetch1[Base, dest];

BaseHi _ y;
Base _ x;
PFetch1[Base, dest, 0]; *DF2 addressing with zero displacement

Source _ x;
PStore1[Base, source];    *ok to load source for PStore1/2 in preceding instruction
```

The virtual address in the D0 contains only 22 significant bits due to the limited size of the map, although a full 24 bits are provided by the virtual address calculation. If bit 1 of the virtual address is nonzero (i.e. bits 1 or 9 of the base register), the memory controller will report a bounds fault when a memory reference is attempted using the base register. When the high base register is set up, bit 0 should be ORed into bit 1, and bit 8 should be ORed into bit 9. The FixVA short field descriptor is provided to facilitate this operation.

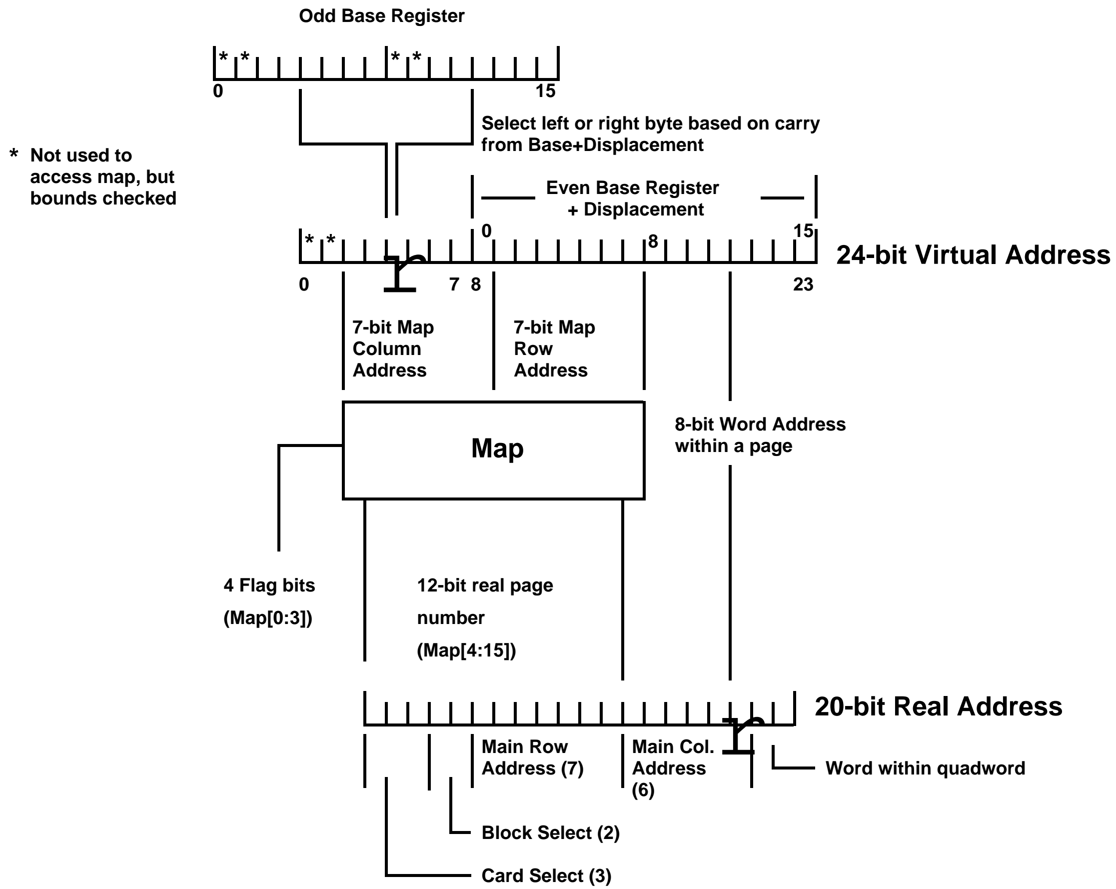


Figure 5.2.5 Address Calculation

5.3 R Interlocking

The worst-case time between the execution of a memory reference microinstruction which uses R and the data transfer which it causes can be quite long if the effects of errors are accounted for. Between the time the reference is started and the time the memory controller takes or delivers the R data, the processor must not be allowed to read an R location for which a fetch is pending, nor write an R location for which a store is pending. So that programs need not always observe the worst case times, logic is provided which compares the R addresses in MC1 and MC2 with the R address generated by the processor, and aborts the current microinstruction if the processor attempts to reference an R location for which a memory access is pending.

The precise conditions for aborting an instruction due to R conflicts are:

> MemInst' and RA=PA and [(Ftype and ALUF#0) or (Stype and LR)], where RA is the R address generated by the processor during cycle0 (the 'read address'), PA is the R address in MC1 or MC2 (the 'pipe address'), and Ftype and Stype are the reference type in the pipe stage. Note that ALUF#0 means 'processor is reading R'. Ftype operations are PFetch 1,2,4, ReadPipe, XMap, and Input. Stype operations are PStore1,2,4, and Output.

The comparisons are done during cycle0 for both MC1 and MC2 if the stage is active. At t1, the results are latched and used to abort the instruction during cycle1 if there is a conflict.

When a multiword is transferred to or from the memory, all R locations containing the item should be interlocked. Since there are only two address comparators, one for MC1 and one for MC2, R locations used for multiwords must be double- or quadword aligned for the interlock comparators to function. When the memory reference in MC1 or MC2 is a PFetch2 or PStore2, the low bit of the addresses are not checked, so an instruction can access either word of the doubleword and cause interlock. Quadword references are similar - the low order two bits are omitted from the comparison.

Note that the interlock comparators are not activated during memory reference instructions. This means that if a base register is fetched into R and then used in a memory reference instruction without being explicitly read first, the old version of the base register will be used if the base register has not been filled by the memory. This situation must be avoided by the programmer.

The organization of the memory and the Mesa stack has two other potential problems:

> 1) When a value is pushed onto the stack, the stackpointer is not updated until t2 of the instruction, after the comparisons with the addresses in MC1 and MC2 have been done. If a PStore is pending from the stack, this could cause the value on the stack to be overwritten before the memory has had a chance to store it.
>
> 2) When a PFetch2 is pending to the stack, it is possible for the processor to read the R location which will eventually get the second word of a doubleword (the quantity being fetched to the top of the stack) without the interlock comparators recognizing the situation (since the R addresses may not be

doubleword aligned).

To ensure that these situations cannot cause trouble, there are two additional conditions which cause a microinstruction to be aborted:

A non-memory reference microinstruction is aborted if (1) it attempts to increment the stackpointer when a PStore1 or PStore 2 is pending from the stack, or (2) if it attempts to decrement the stackpointer when a PFetch2 is pending to the stack.

Note that this means that when the microcode does a doubleword fetch to the stack, it must pop the value (read it and decrement the stackpointer) reather than simply reading the value, or the check will be defeated.

5.4 The Map

Address translation in the D0 is done by a 16K by 16 bit table lookup map located on the memory control card. The map receives bits 2-15 of the 24-bit virtual address from the ALU card, and produces the most significant 12 bits of the real address, plus four flag bits.

The virtual address is time multiplexed in two cycles over the 7-bit StorA bus. The map row address is sent first, in the first MC1 cycle. During the second MC1 cycle, the map column address (which was latched at t2) is sent to the map. If the reference type requires a map cycle, the controller will have delivered RAS and CAS to the map chips, and the map data will become stable ~170ns after t2 of the memory reference instruction.

The Real Page address occupies bits 4-15 of a map entry, and is distributed as follows:

Bits 4-6 are decoded into one of eight Card Select lines and sent to the storage cards.

Bits 7 and 8 are decoded into one of four Block Select lines and sent to the storage cards.

Bits 9-15 are the main Row Address, and are sent to the storage cards over the StorA bus under control ot the preRowAd signal generated by MC1.

The remaining six address bits required by the storage cards are latched on the ALU card at t2 of the instruction which began the sequence. These bits are sent to the storage cards one cycle after the row address, also on the StorA bus.

The flag bits in the map occupy bits 0-3 of a map entry. LogSnglErr (bit 0) and WriteProtect (bit 1) are written only when a map entry is loaded, but Dirty (bit 2) and Referenced (bit 3) will be set by MC1 during references of the appropriate type. These bits may also be written as part of map loading (by the XMap operation). The flag bits and the reference type determine whether a reference is permitted. A PROM in MC1 examines the flag bits and the reference type, and produces a signal (MC1Fault) which is tested by MC1 to determine if a reference is legal. The combination WriteProtect = Dirty = 1, Referenced = 0 (which cannot normally occur) is used to indicate Vacant, i.e., no real page is assigned to this virtual page.

The map is read and written by the processor with the XMap operation. This operation uses a block of four R registers. It reads the data from the map entry corresponding to the virtual address supplied to the operation into the last three registers, then writes the first word of the block into the map entry.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	R Address:	Pipe Ram Address:
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1		MAP ROW ADDRESS *							n	(0)
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1		MAP COLUMN ADDRESS *							n+1	(2)
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1	TASK *				REFERENCE TYPE *				n+2	(1)
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1	BLK.1 *	MAIN ROW ADDRESS *							n+3	(4)
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1	BLK.1 *	MAIN COLUMN ADDRSS *							n+4	(6)
MC2 ErA*	MC2 ErB*	MC1 ErA*	MC1 ErB*	H4 PE	Map Bnd	1	1	LOG SE *	WP *	DIRTY *	REF *	CARD*			BLK.0 *	n+5	(7)

* Read in complement form

Figure 5.5 ReadPipe R Register Format

5.5 The Error Pipe

A number of errors are possible during a memory reference. Errors detected by MC1 are:

Page Faults (an attempt to access a page whose map entry has D=WP=1, Ref = 0).
WriteProtect violations (an attempt to store into a page whose map entry has WP=1).
Input Bus Parity Errors (also called H4Parity Errors): These can only occur on INPUT or IOStore operations.
Bounds faults (Virtual address >22 bits).

Errors detected by MC2 are:

Double bit memory errors
Single bit memory errors. Single errors are ignored if the operation is IOFetch (corrected data is sent to the controller), and will only occur on a PFetch if the LogSnglErr bit is on in the map entry for the referenced page.

When an error occurs, the memory controller finishes the reference if it can, sets the Fault flag, which will cause task 15 to run and deal with the error, and becomes idle. In all cases, it is essential to recover the following information about the reference:

The reference type

The number of the task which initiated the reference.
The virtual address
The real address and map flags
The stage that detected the error (MC1/MC2)

If the error is a single or double memory error, the syndrome.
If the reference is an INPUT or IOStore, the H4 parity error flag

All of these items with the exception of the last two are contained in the error pipe memory. This memory is a 16 word by 8-bit RAM (although only 12 words are used) which is controlled by MC1.

When MC1 is started, if the reference type is anything other than Refresh, Output, or ReadPipe (which cannot cause errors), a 'pipe slot' bit is assigned to the reference. This bit determines which half of the pipe RAM will be used to hold information about the reference, and is assigned by complementing the bit assigned to the preceding reference. During the time MC1 is processing the reference, the pipe RAM is written.

If an error is detected by MC1, Fault is set, and MC1 will not start another reference until the processor has had time to start the fault handling microcode. In addition to setting Fault, MC1 also sets MC1ErA or MC1ErB, depending on which pipe slot was used for the reference.

If MC1 is error free, it starts MC2 if the reference is a Pfetch, IOfetch, Output, or Pstore1/2. The pipe slot bit is passed to MC2 so that if an error is detected, it can be reported by MC2. If MC2 detects an error, it sets MC2ErA or MC2ErB, depending on the pipe slot bit, and also loads the syndrome into one of two 8-bit registers, again depending on the pipe slot bit. An Output operation cannot cause an error in MC2.

If an error occurs, task 15 can recover the information in the pipe RAM with a ReadPipe operation, which will dump the contents of the pipe RAM into six contiguous R registers. Figure 5.5 shows the format of a pipe entry. The data in the Pipe RAM is returned in the low byte of the six registers, while the high byte contains the flag bits that indicate the source of the error (these bits appear identically in all six words). If the function ResetMemErrs is executed as part of the ReadPipe instruction, the B pipe entry is delivered. If not, the A entry is delivered. ResetMemErrs clears the flag bits returned in the left byte of the ReadPipe, so the A pipe entry should be read by the fault handler first.

The register containing the A and B syndromes may be read as an external R source. The A syndrome is in the left byte, the B syndrome is in the right byte.

5.6 Error Correction

The D0 memory is corrected over a 64-bit quadword. The check code generation (during stores), syndrome generation, and correction if necessary (during fetches) are done word-serially as the data are transported through the memory controller.

During PFetch operations, uncorrected data are transported into R on the assumption that there is no error. When the entire quadword has been transported, the syndrome is calculated, and if this assumption is correct, the processor is allowed to access the data. If correction is required, the data (which were saved in a buffer during the transport), are corrected on-the-fly, and sent to R a second time.

During IOFetch operations, corrected data are sent to the device from the output of the buffer. If a double-bit error is detected during transport, the signal OFault is sent to the controller, so that it can terminate its operation if appropriate.

Data errors generate processor faults as described in section 5.5. Single errors during I/O fetches are corrected, but no fault is generated, since this would degrade the bandwidth. Single errors detected during PFetches will generate a fault only if the LogSnglErr bit is on in the map entry for the page. Correction will always occur, however.

The H-matrix for the Hamming code used in the D0 is shown in figure 5.6. This code provides single error correction and double error detection. the decoding rule with which the bit in error is determined given the syndrome S[0:7] is as follows:

1) If S[0:7] = 0, there is no error
2) If the syndrome contains exactly one "1" bit, the associated check bit is in error.
3) If the syndrome contains an odd number of "1" bits and S[4:6] = 3, 5, 6, or 7, there is a single error. S[4:6] = 3 indicates an error in bits 0-15, S[4:6] = 5 indicates an error in bits 16-31, S[4:6] = 6 indicates an error in bits 32-47, and S[4:6] = 7 indicates an error in bits 48-63. The bit within the word is given by S[3], S[2], S[1], S[0], treated as a 4 bit number.
4) Otherwise, a double or multiple error exists.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S0		X		X		X		X		X		X		X		X
S1			X	X			X	X			X	X			X	X
S2					X	X	X	X					X	X	X	X
S3									X	X	X	X	X	X	X	X
S4																
S5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S6	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S7	X			X		X	X			X	X		X			X

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
S0		X		X		X		X		X		X		X		X
S1			X	X			X	X			X	X			X	X
S2					X	X	X	X					X	X	X	X
S3									X	X	X	X	X	X	X	X
S4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S5																
S6	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S7	X			X		X	X			X	X		X			X

	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
S0		X		X		X		X		X		X		X		X
S1			X	X			X	X			X	X			X	X
S2					X	X	X	X					X	X	X	X
S3									X	X	X	X	X	X	X	X
S4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S6																
S7	X			X		X	X			X	X		X			X

	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
S0		X		X		X		X		X		X		X		X
S1			X	X			X	X			X	X			X	X
S2					X	X	X	X					X	X	X	X
S3									X	X	X	X	X	X	X	X
S4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S5	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S6	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
S7		X	X		X			X	X			X		X	X	

Figure 5.6 Error Correction H-Matrix

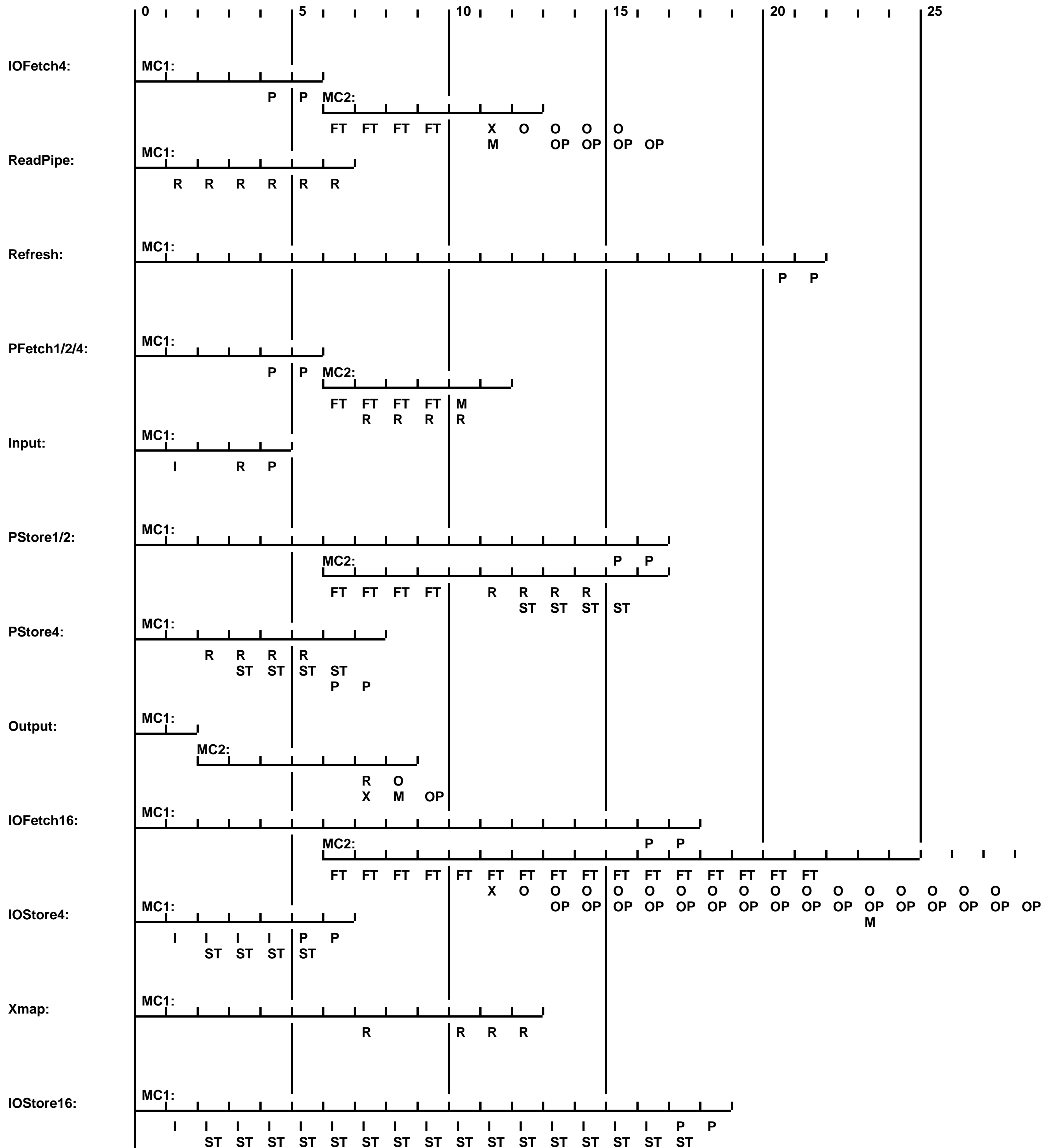
5.7 Refresh

The refresh operation is done by a task in the processor driven by a timer. A single execution of the Refresh operation refreshes four rows in both the map and main storage chips, so there need be only 32 Refresh operations done every 2ms (63usec/operation). The operation uses the 7-bit main column address register on the ALU card to supply the refresh address. This counter is loaded from bits 7-13 of the processor ALU output at t3 of the memory reference instruction, and is sent on the StorA0-6 lines two cycles after the MC1 control bit PreRowAd is asserted. The column counter is normally incremented after it is used (to provide for page mode operations), so no special provisions are required to increment it.

The refresh operation calculates a virtual address in the normal way (as do all memory reference instructions), but only bits 7-13 of the low order word are used. For this reason, the microcode should keep the refresh address in a single (odd) R register, increment the register by 16 when it is awakened (since the low order two bits are not used), then initiate the refresh operation with a displacement of 0.

5.8 Memory Timing

Figure 5.8 shows the timing of all memory reference operations. The time is in machine cycles, and t0 of the figure corresponds to t2 of the memory reference. This chart is a summary of Appendix B, and shows only the normal case for each reference. For the timings when errors occur, refer to the appendix.



Legend:

P: PreloadMC1 (MC1 will start in 2 cycles if cycle0 of a memory reference instruction occurs while this bit is on)

R: R Access (Processor suspended)

FT: Fetch Transport (H5 _ Storage Data)

ST: Store Transport (Storage _ H4)

I: Input (H3I or H4 _ IData)

O: Output (Controller _ OData)

X: MC2StartXport to Controller

M: Earliest allowed MC1StartMC2

OP: Output Parity bit to Device

Figure 5.8 Memory Timing

5.9 Storage Card Organization

The real address space of the D0 is 1 megaword, divided into eight 128k regions. The backplane of the processor is wired such that the first 8 slots after the CPU can accommodate a memory card or an IO device. Each slot corresponds to a fixed region (the first slot covers 0-128k, the second 128k to 256k and so forth). Currently, only 96k storage cards exist. Provision is made for a future 128k storage card when RAM density is increased beyond the current 16k bits per chip limit. Since the slot covers the entire 128k region, and a card can have as little as 64k of memory, there are holes in the real address space if 64k or 96k cards are used. In fact, since an IO device can be plugged into a memory slot, the holes can be as large as 128k. The processor determines the configuration of real memory at boot time and provides the software with a map showing the location of the holes. The system software must accommodate malfunctioning sections of real memory, so minimal extra work will be needed to cope with the holes. This scheme introduces the following configuration rule: Storage Cards must be plugged into the first 8 slots (this is enforced by keying the cards physically). I/O cards may be plugged into any slot, but there may be no gaps between any cards.

The 96k storage card is shown in block diagram form in figure 5.9 The card is divided into three 16k x 32 *banks* (plus 4 Error Correction bits per bank). For any memory cycle to the card, two of the three banks will be started, thus memory is cycled in 64 bit quadwords. Consider the 128k region of the address space covered by the card to be composed of four 32k *blocks* (blocks correspond to 32k sections of the real address space; banks correspond to physical arrays of RAM chips on the storage card). Block 0, when addressed, will cause banks 0 and 1 to be cycled. A cycle to block 1 will use banks 0 and 2, while block 2 will use banks 1 and 2. Block 3 is the 32k hole.

For each block, one of the two banks contains the even doubleword involved in the memory cycle while the other contains the odd doubleword. Bank 0 contains even double words for blocks 0 and 1. Bank 2 contains odd doublewords for blocks 1 and 2. Bank 1 contains odd doublewords for block 0, but contains even doublewords for block 2. The chart in figure 5.9 shows all of this in a readable form. The basic idea is that Bank 0 always contains even doublewords, bank 2 always contains odd doublewords, and bank 1 contains both even and odd doublewords.

The piped control section of the memory system allows the storage card to be considered as three independent pieces, Input Transport, Memory Cycle, and Output Transport. Input transport data arriving from the processor for a write cycle on the 16 bit bus is buffered and held in registers which are strobed in the sequence: low bits of even DW, high bits of even DW, low bits of odd DW, high bits of odd DW. Since bank 0 always gets even DWs, its input registers are clocked in the first two cycles of every transport. Similarly, the input registers for bank 2 are clocked on the 3rd and 4th cycles. Since the real address bits (and therefore the card and block number) are not available until mapping is finished (which is after the first two cycles of transport), bank 1 cannot determine whether to load even DWs or Odd DWs until the 3rd cycle. Therefore it loads all even DWs, and then loads the odd DW on top of it if block 0 was the addressed block (ie. if bank 1 contains odd DW's for the addressed region).

ErrorCorrection bits are transported with the 4th data word. Bank 0 always latches the upper 4 bits, bank 2 always latches the lower 4 bits. Bank 1 has two-way input multiplexing which selects between the upper and the lower bits as a function of BlockSel0. If BlockSel0 is asserted, bank 1 latches the lower bits. Otherwise, it latches the upper bits.

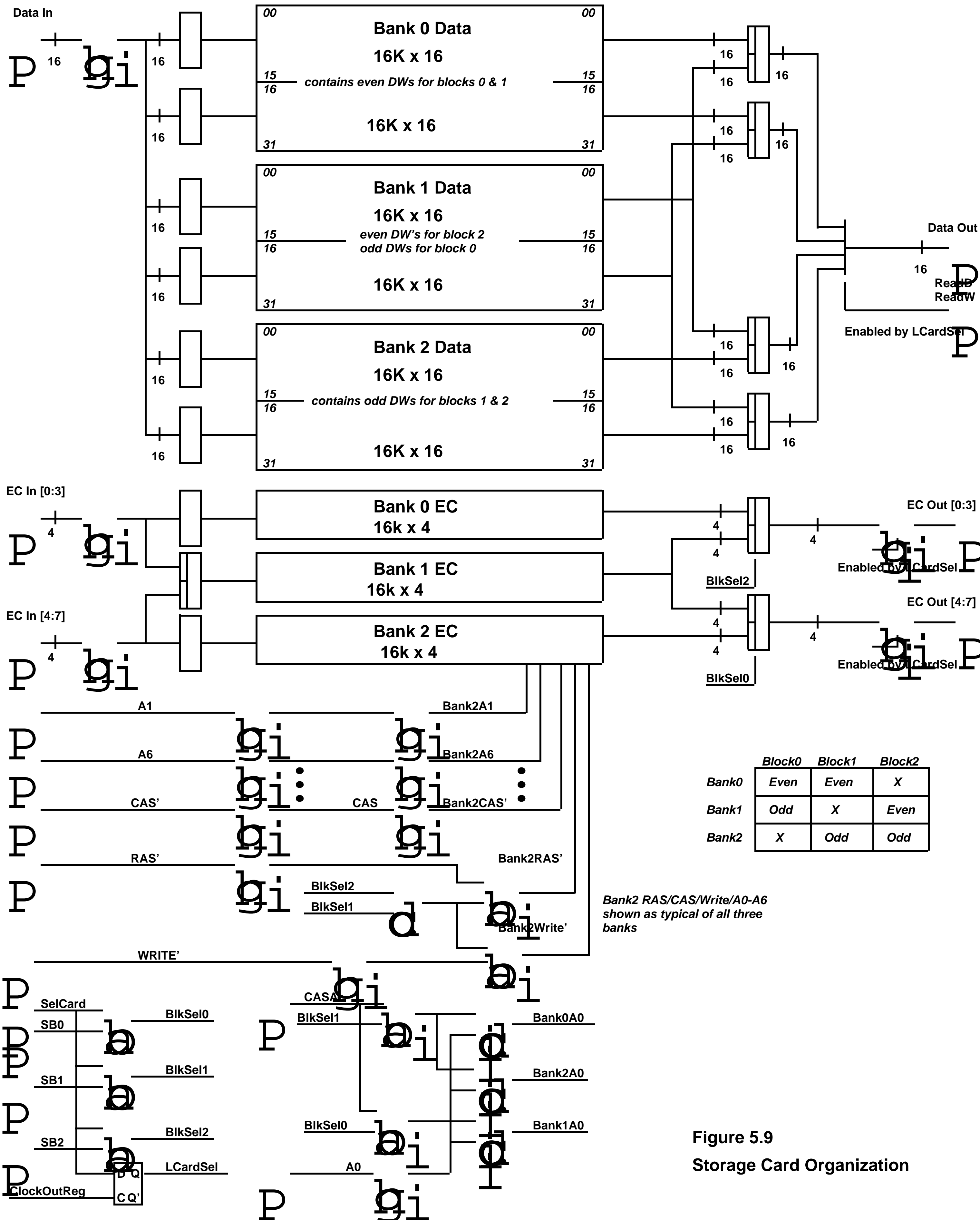
The memory control in the processor decodes the top 3 bits of the real address from the map to determine which of the 8 cards will be run. The decoder outputs are wired to the corresponding slot, so that no on-card decoding is needed on the storage card. The block bits (real address bits 4 and 5) are also decoded, but every card gets all four SelectBank signals SB0-SB3 (note that the 96k card does not use SB3, and the 64k card uses neither SB2 or SB3). The card *ands* the SelCard signal with the SB signals to produce BlockSel0, BlockSel1 and BlockSel2. Two of the three BlockSel signals are *ored* together and *nand*ed with buffered RAS to produce BankRAS' signals for each of the 3 banks. A similar circuit creates BankCAS' and BankWrite' for three banks. For refresh, the processor enables all CardSel lines as well as all three SB lines, so all chips get the refresh cycle.

While A1-A6 for the chips need only buffering for fan-out, A0 needs additional gating to select between the halves of the chip used for each of the two blocks for which the bank contains data. During RAS, the backplane A0 signal is sent to all chips. During CAS, one of the two banks gets a 1 in A0, while the other gets a 0. The memory control card will force the backplane A0 to a 0 during CAS and provide the CASA signal (an early version of CAS) to enable this logic.

For a read cycle, the Card/Bank selection circuitry and the addressing logic works the same way as it does for a write. When the data comes out of the RAM chips, the 96 output bits are multiplexed down to the 64 bits that actually contribute data and are latched. The multiplexing depends on the fact that the even DW can only come from bank 0 or bank 1, while the odd DW can only come from bank 1 or bank 2, so only two way multiplexing is necessary.

Once these bits are stored in the output register, MC2 controls the transfer of data to the processor. The 64 bits are muxed to 16 by a set of 4 way multiplexers with tri-state outputs. The enable for the outputs is a latched version of CardSel. MC2 controls the select inputs to the multiplexor sending the data into the processor in the order it was transported during a store. The EC bits have two way muxing similar to the data bits. All 8 bits are transported with the first word of data going in to the processor.

All input signals to the card are buffered to present 1 load to the bus (exceptions: SelCard has loading of 5, but is private to each card, clkOutputReg has a loading of 2. It can be driven with a high current driver and so should not cause problems). The drivers for the MOS RAMs are fanned out to 18 chips using an S series gate (32+4 chips per bank/2 drivers = 18 chips/driver) with a series damping resistor to minimize undershoot. Data outputs from the RAM chips drive two loads (two inputs of 25S09s). The only outputs from the card are the output data bits (which are tri-state outputs of S253s) and the EC output bits (8T96s).



	Block0	Block1	Block2
Bank0	Even	Even	X
Bank1	Odd	X	Even
Bank2	X	Odd	Odd

Figure 5.9
Storage Card Organization

6.0 Input-Output

The preceding section discussed the memory and I/O operations available to a microprogram. This section will discuss the interface between the processor/memory system and I/O controllers.

I/O controllers in the D0 are intimately associated with microcoded tasks in the processor. When the controller requires service (typically to request or deliver data from a device), it generates a wakeup request, which will (eventually) cause its associated task in the processor to run and provide the requested service. All memory requests are made by the processor, although the actual transfer of data will often occur directly between the memory and the controller. The processor can exchange status information or data with a controller using INPUT and OUTPUT instructions. There are also two lines, shared by all controllers, which allow a single bit of status to be transferred between the microprogram and the controller without incurring the delays introduced by the memory system when INPUT and OUTPUT operations are done.

The interface between a controller and the processor/memory consists of a fixed set of signals which use well-defined signalling protocols. Use of some of the signals is optional, but all controllers must provide a minimum set of capabilities.

Subsequent sections will discuss the interface signals, the way in which controller addressing is done, how wakeup requests are made, the sequence of events in data transfers, and the special signals in the interface.

Appendix C shows the logic required to implement the minimum set of functions required in all controllers.

6.1 Interface Signals

Table 6.1 lists all signals available to I/O controllers. These signals are grouped into six categories based on their function. Subsequent sections describe their functions in detail.

Table 6.1

Signal Name	Number of Lines	Input or Output/ Driver/Rcvr type[1]	Use
Signals that establish the controller address:			
SRIn’	1	IS	Controller address in
SROut’	1	OS	Controller address out
SRClock	1	IS	Clock for controller address
Signals associated with wakeup request generation:			
Phase1Next’	1	IS	Wakeup state machine control
WakeP1’	1	OW,IS	Wakeup Request
WakeP2’	1	OW,IS	Wakeup Request
WakeP3’	1	OW,IS	Wakeup Request
Data input signals (processor/memory <-- controller):			
IAddr[0:7]	8	IP	Input Address
IValid’	1	IP	Input Address Valid
IData[0:15]	16	OT	Input Data
IData[16]	1	OT	Input Data Parity
Data output signals (processor/memory --> controller):			
AdvancePipe’	1	IP	Address pipeline
MC2StartXPort	1	IP	Address pipeline
OValid’	1	IP	Indicates output data is valid
OData[0:15]	16	IP	Output Data
OData[16]	1	IP	Output Data Parity
OFault’	1	IP	Output Fault
Single-bit communication path signals:			
IOAttn’	1	OW	Attention signal to processor
IOStrobe	1	IP	Strobe from processor
Clocks and processor status signals:			
CTask[0:3]	4	IP	Task currently running on processor
RUN	1	IP	Low if processor not ready
EdgeClockFeed’	1	IS	Clock
RamClockFeed’	1	IS	Clock

1) Although signals are named relative to the processor (i.e. Output means data to a controller), these signals are specified relative to the controller as follows:

- IP: Each board will receive this input line with one PNP input load
- IS: Each board will receive this input line with one Schottky TTL load
- OS: Each board will drive this line with one Schottky TTL output (totem-pole)
- OT: Each board will drive this output line with one Schottky tri-state output
- OW: Each board will drive this line with one SN74S38 Wire-or driver

All signals in the machine begin to change on the rising edge of EdgeClock’ (which is delayed from the backplane signal EdgeClockFeed’ by two S levels). Signals generated in the processor will become stable no later that ~30ns after EdgeClock’ rises, and signals generated by controllers should be stable 30ns before the clock rises. EdgeClockFeed’ on

the backpanel is low for 25% of a cycle, RamClockFeed’ is low for 50% of a cycle, and the rising edges of these signals are conicident (within skew limits).

6.2 Controller Addressing

Each controller in a D0 system is associated with a task in the processor. A single controller may have up to sixteen internal registers which may be read or written by the processor/memory. The functional assignment of the registers within a controller is unspecified with the exception of INPUT register 0, which will return a 16-bit identification number unique to the board type when it is read. When the processor executes a memory or I/O operation directed to a particular controller, it specifies an eight-bit I/O Address. The most significant four bits of this address are the task number, the low order four bits select one of the sixteen registers within the controller. Since there will be many types of controller, it is not possible to build the task number into each controller when it is designed, nor is it desirable to establish the task number with switches when the controller is installed. Instead, the task number is assigned dynamically by the processor using the SRIn’, SROut’, and SRClock lines.

Each controller will provide a four-bit CAddr shift register, which holds the task number associated with the controller. The input of this register is SRIn’, the output is SROut’, and the clock is connected to SRClock (the register clocks on the positive-going clock transition). SROut’ from one card slot is wired to SRIn’ on the next card, and the SRClock lines are connected together. Storage cards jumper SRIn’ to SROut’, and SRIn’ for the slot closest to the processor (slot 5) is connected to H2.15.

The processor can set an address into each controller by serially placing a bit pattern in H2.15 and executing the function GenSRClock, which sends one clock to all the shift registers. For a 12-slot system, 48 clocks are required to initialize all possible controllers. The backplane is built so that slots 5-12 can accept either storage cards or I/O controllers, slots 13-16 can accept only I/O controllers. Subject to this limitation, storage cards and controllers can be mixed in any order, as long as there are no gaps between cards.

At bootstrap time, the microcode will load a fixed initialization pattern (which makes all controller addresses unique) into the register, then read register 0 in each device to determine the type of controller (if one is present). Load devices may be located in this manner, and the processor may then change the assignment of controllers to correspond to the proper task priority arrangements for the particular microcode and peripheral configuration present.

6.3 Task Wakeup Requests

When an IO device needs service from its associated task in the processor, it generates a *wakeup request*. In each controller, the CAddr’ register holds the complement of the number of the task associated with the controller. The number of the task is also its priority. A wakeup request sequence requires two distinct phases. During the first phase (phase 1), each controller which needs service ORs an encoded version of the two most significant bits

of its task number onto the WakeP1, WakeP2, and WakeP3 lines as follows (the WakeP lines are low true on the backplane):

Task Number	CAddr[0:3]	WakeP1	WakeP2	WakeP3
0-3	0 0 x x	0	0	0
4-7	0 1 x x	1	0	0
8-11	1 0 x x	0	1	0
12-15	1 1 x x	0	0	1

At the end of phase 1, the processor will encode the WakeP lines into a two-bit number corresponding the the most significant bits of the *highest priority* requesting task:

PT0 = WakeP2 OR WakeP3
PT1 = WakeP3 or (WakeP1 and not WakeP2)

At the same time, all controllers will examine the WakeP lines, and those which see lines corresponding to requests made by higher priority controllers will not participate in the second phase of the request sequence.

During the second phase of a request sequence (phase 2), the controlers which are still participating will OR a similarly encoded version of the least significant two bits of their task number onto the WakeP lines:

Task Number	CAddr[0:3]	WakeP1	WakeP2	WakeP3
4,8,12	x x 0 0	0	0	0
1,5,9,13	x x 0 1	1	0	0
2,6,10,14	x x 1 0	0	1	0
3,7,11	x x 1 1	0	0	1

The processor encodes these lines using the same process as in the first phase. The resulting two bits and the two bits which were saved from the first phase are latched in the processor. These bits are the task number of the highest priority requesting task (HTask).

The timing of the wakeup logic is shown in figure 6.2 The signal Phase1Next' is supplied to all controllers by the processor to synchronize the processor and controller logic. Phase1Next will always be exactly one cycle in length (although since EdgeClock may be withheld by the branch logic, the duration of a cycle is sometimes doubled). Request phase 1 occupies the cycle immediately following Phase1Next, and request phase 2 occupies all subsequent cycles unitl the next occurrence of Phase1Next. Normally, phase 2 will also be one cycle in length, but if processor cycles are suspended, Phase1Next will be deferred, and phase 2 will be extended. Controllers must present valid data on the WakeP lines even if suspension occurs. Note that the delay between the time that a controller asserts or removes the wakeup request and the time the processor responds by running the task is a minimum of three instruction times. Designs for controllers and their associated microcode must take this latency into account.

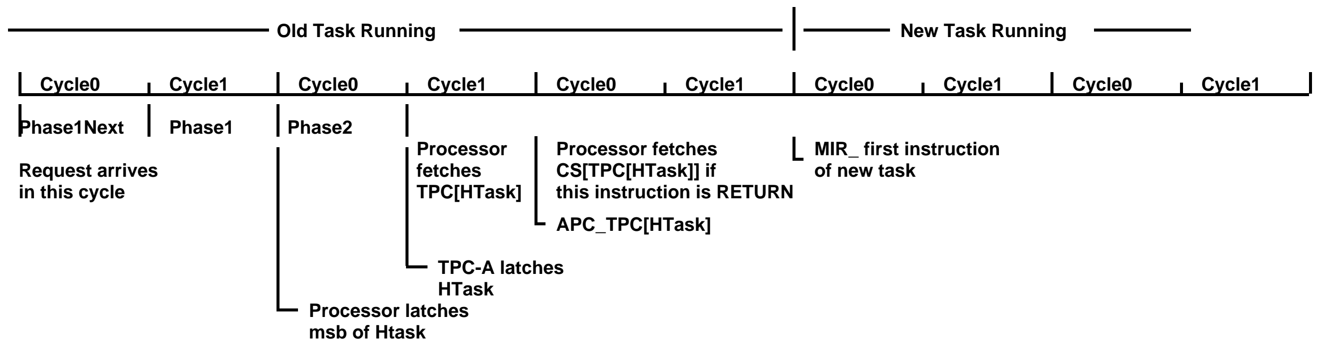


Figure 6.2 Wakeup Latency

6.3 Input and Output Operations

Data is read from device controllers with INPUT or IOStore operations. From the point of view of the controller, these operations are indistinguishable, since they result in the same sequence of signals.

An input operation is started when the processor starts MC1 and places the I/O address on the IAddr lines. Some number of cycles later (the number depends on the operation), the memory controller will assert IValid. The controller that recognized the address will return data in the cycle following the one in which IValid is asserted. The parity bit (IData[16]) must be returned in the same cycle as the data.

Output is more complex, since transport of data to the controllers is controlled by MC2, not MC1. An output operation begins in the same way as an input, with the loading of IAddr. Some number of cycles later, MC1 has completed its work and MC2 is started. At this time, the processor issues AdvancePipe, and *may* change IAddr. The controller is expected to use AdvancePipe to latch the data from IAddr in an internal register which is essentially a part of MC2.

Since the transport of data from MC2 can overlap the time at which MC2 is started for the next reference (probably for a different address), a third stage in the address pipeline is required. The memory controller issues the signal MC2StartXport just prior to the time it will begin to deliver data to the controller, and the controller is expected to use this signal to latch the output of the register clocked by AdvancePipe in yet another register. This final register is the one that the controller compares with CAddr to determine if it is the destination for an output operation. When the memory finally has data to deliver, it asserts the signal OValid. The controller should use OValid to latch the OData lines (OValid should, of course, be used to qualify EdgeClock). If the memory delivers OFault in the same cycle as the one in which OValid is delivered, it indicates that a double error has been detected.

The controller may use this signal in whatever way it wants. The memory will deliver the odd parity bit OData[16] in the cycle following the data. Controllers may choose to ignore the parity bit if data integrity is not crucial (e.g. data for a display).

6.6 I/O Attention, I/O Strobe, RUN

In many cases, the microcode for an IO device controller will use a tight loop to transfer data to or from a controller. Some devices may need to inform the processor of exceptional conditions which should terminate or modify such a loop. To provide this capability, the processor delivers the number of the task which has control of the processor on the CTask[0:3] lines. IO device controllers may receive these lines and return the (wire-ored) signal IOAttn' to the processor. At t1 of each microinstruction, IOAttn' is latched in the processor, and a branch condition is provided to test its value. Each board will drive the IOAttn' line with a single wire-or driver (SN74S38).

Note: Since the time for CTask[0:3] to reach the controller and be returned to the processor as IOAttn' is longer than one cycle, microprograms should not test the value of IOAttn in any instruction following a RETURN, since the RETURN may cause a task switch.

The IOStrobe signal allows the processor to send a single bit of information to the controller without incurring the delay associated with the OUTPUT operation. The function IOStrobe makes this line true for one cycle. Each controller can AND this line with a signal that indicates that its associated task is running (CTask = CAddr), and use the resulting signal for any purpose it wishes.

A reset signal is distributed to all IO controllers. This signal, RUN, is cleared during a bootstrap operation, and if the +5V supply drops below approximately 4.7V. In particular, this line should terminate writing on magnetic media devices. All controllers should reset and quiesce themselves when this line is dropped, and also when an OUTPUT operation with data=0 is directed to the devices's control register (the microcode will reset the IO system by doing for i = 0 to 255 do OUTPUT[i,0]) After RUN becomes high, device controllers should refrain from requesting wakeups until explicitly enabled by their microcode.

Device controllers may implement other task-specific reset operations via OUTPUT instructions if they need them.

7.0 User Terminals and Controllers

7.1 Terminal to Controller Interface

So that a number of different controller and terminal types may be freely interconnected in D0-based systems, a common interface between terminals and controllers has been defined. This interface assumes that a terminal contains a raster-scanned bitmap display and one or more low bandwidth input devices (keyboard, pointing device, etc.). The controller transmits digital video and sync to the terminal over six pairs of a seven-pair cable. The input data is transmitted to the controller serially over the seventh pair (the "backchannel"). Video and control (sync) are time-multiplexed, and four bits are transmitted in parallel to reduce the bandwidth required on the cable.

While the description in the following sections assumes a display having one bit per pixel, the basic signalling mechanism may be extended to support gray-level or color displays.

7.1.1 Cables and Connectors

The interface cable is Belden #9507 or equivalent. This cable consists of seven twisted pairs of #24 AWG stranded, vinyl insulated wire, surrounded by a foil shield with a stranded drain wire and an overall vinyl jacket. The outside diameter of the cable is .290".

Six of the seven pairs are laid in a spiral around the seventh pair (the red/black pair). As a result, the electrical length of the red/black pair is much less than that of the remaining pairs. Since the interface depends on having minimum skew between the six pairs used for the high speed section, the red/black pair is used for the backchannel.

The interface connector is a 15 pin Cannon "D" series or equivalent unit (several sources exist). The controller contains a male connector, the terminal contains a female connector.

7.1.2 Drivers and Receivers

Figure 7.1.1 shows the drivers and receivers used in the interface. ECL 10K differential drivers with 220 ohm pulldown resistors to -5.2v are used to drive each pair. The receiving end of each pair is terminated in 100 ohms, and differential ECL receivers are used to recover the data.

7.1.3 Video and Control Channel

This section of the interface uses six pairs in the cable. Four bits of data are transmitted in parallel from the controller to the terminal, accompanied by two clock signals. The four data bits are interpreted as video or control by the terminal, depending on the phase of the clock. Figure 7.1.3 shows the timing relationships in the interface. The controller places data on the data lines on the falling edge on ClkA. The data are sampled by the terminal on the rising edge of ClkA. If ClkB = 1 at this time, the nibble is interpreted as four bits of video.

If ClkB = 0, the nibble is interpreted as sync and control information. ClkA and ClkB are transmitted in quadrature so that the terminal can reconstitute a clock at the video bit rate.

When video data is serialized by the terminal, bit 0 is transmitted first, bit 3 is transmitted last. A logic 0 corresponds to a blanked display.

[Note: OSD display specifications refer to bit 0 as the "least significant bit" of a nibble, and state that this bit is transmitted first. In D0 memory, bitmaps are stored such that the most significant bit is transmitted first, but since the D0 convention is that bit 0 is the msb, the bit numbers in OSD specifications are the same as D0 bit numbering.]

When a nibble is interpreted as control information, bit 2 is reserved for horizontal sync, bit 3 is reserved for vertical sync. The interpretation of bits 0 and 1 is not defined; different types of terminal may use them for different purposes.

7.1.4 Backchannel

Data from low bandwidth input devices at the terminal are transmitted serially over this line. The data are clocked by the terminal on the falling edge of the horizontal sync pulse, and will be sampled by the controller during the subsequent scan line. Data are transmitted in a frame composed of a single "start" bit (a logic one), followed by a number of data bits. A frame is transmitted when any input signal at the terminal changes state. The idle state of the line is logic zero.

Since different types of terminals may have differing amounts of information to transmit, the format of a frame is not defined. The controller microcode must provide whatever capabilities are required by a particular terminal.

Controller	Cable: DA15S Connector	Cable (Belden #9507)		Cable: DA15P Connector	Terminal	Interpretation: As Data:	Interpretation: As Control:
ECL Driver, -5.2V, 220ohm (2)	7, 8	Data0 / NData0	GREEN / BLACK	7, 8	ECL Receiver, 100ohm	First video bit	Not Specified
ECL Driver, -5.2V, 220ohm (2)	5, 6	Data1 / NData1	BROWN / BLACK	5, 6	ECL Receiver, 100ohm	Second video bit	Not Specified
ECL Driver, -5.2V, 220ohm (2)	3, 4	Data2 / NData2	BLUE / BLACK	3, 4	ECL Receiver, 100ohm	Third video bit	Horizontal Sync
ECL Driver, -5.2V, 220ohm (2)	1, 2	Data3 / NData3	YELLOW / BLACK	1, 2	ECL Receiver, 100ohm	Fourth video bit	Vertical Sync
ECL Driver, -5.2V, 220ohm (2)	9, 10	ClkA / NClka	ORANGE / BLACK	9, 10	ECL Receiver, 100ohm	ClockA	
ECL Driver, -5.2V, 220ohm (2)	11, 12	ClkB / NClkB	WHITE / BLACK	11, 12	ECL Receiver, 100ohm	ClockB	
ECL Receiver, 100ohm	13, 14	DataOut / NDataOut	RED / BLACK	13, 14	ECL Driver, -5.2V, 220ohm		
Logic Ground	15		DRAIN	15	Logic Ground		

Figure 7.1.1 Terminal Interface Drivers, Cable, and Receivers

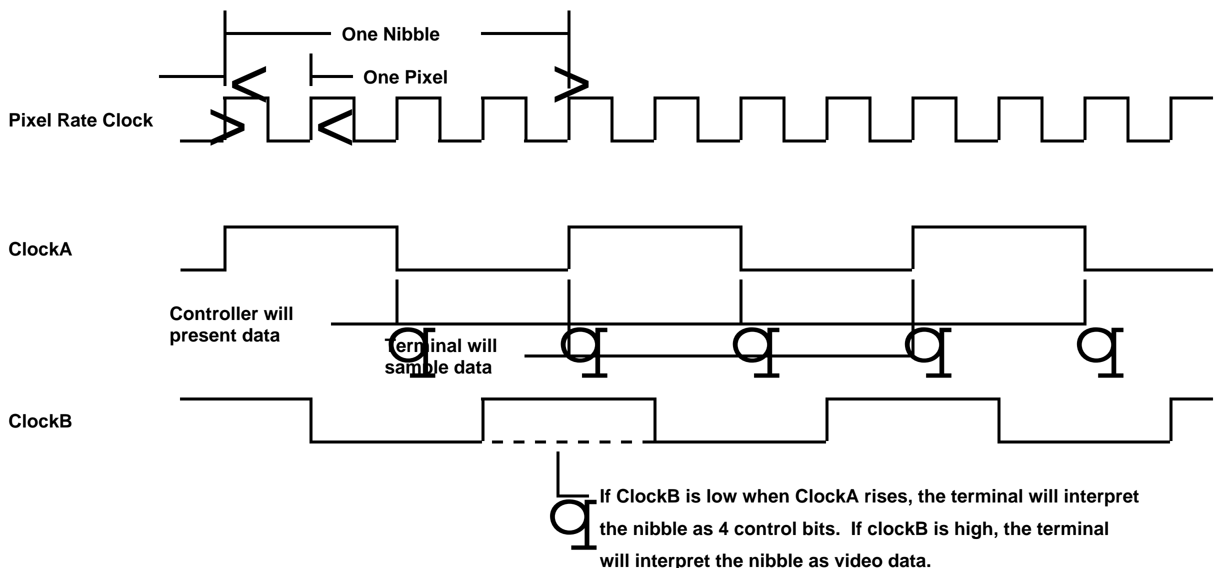


Figure 7.1.3 Terminal Interface Timing

7.2 UTVFC (User Terminal Variable Format Controller)

7.2.1 Introduction

The UTVFC occupies a single D0 board, and supports up to four user terminals utilizing the interface and protocol described in section 7.1. The major subsections of the board are shown in figure 7.2.1. The UTVFC provides horizontal sync for all channels, data buffering for up to 1024 bits per scan line for each channel, a 32 x 32 bit hardware cursor for two of the channels, receivers for the backchannel associated with each terminal, and the logic necessary to interface the D0 I/O system. The video bit rate, horizontal line rate, number of words per scan line and vertical field rate for all four channels must be identical. The bit rate is set by a crystal oscillator, the line rate is determined by the horizontal control RAM, which must be initialized by the processor, and the vertical field rate is determined by the controller microcode, which must count scan lines. The microcode is also responsible for accumulating the serial backchannel message.

The UTVFC also contains provisions for single-stepping the video clock, and reading a number of internal signals.

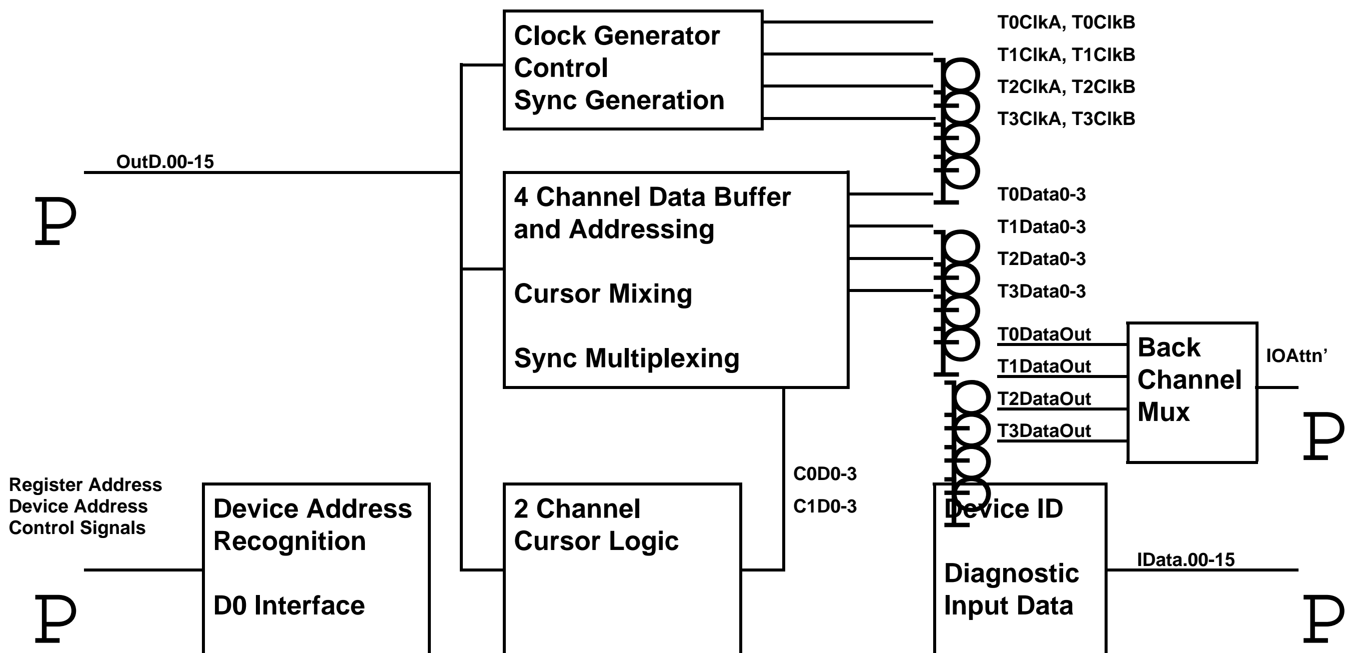


Figure 7.2.1 UTVFC Major Subsections

7.2.2 Output Registers

Figure 7.2.2 shows the assignment of output register addresses in the UTVFC. Eleven of the sixteen possible registers available to a controller are used. The function of each register is discussed briefly here, and more fully in following sections. These registers cannot be read directly, but most can be read indirectly via the diagnostic interface.

Register 0 is the control register. The most significant byte of the control register holds the two unassigned bits that are sent to each terminal as part of a control nibble. Bits 8, 14, and 15 control the clock generator and enable controller wakeup requests. Bits 9 and 10 control the polarity of the video for channels 0 and 1, and bits 11-13 control terminal blanking during vertical retrace and the generation of vertical sync.

Register 1 is the data buffer starting address register. This register must be initialized by the microcode with 64d-Nwrds (the number of memory words per scan line) as a function of the terminal type. Bits 0 and 1 should be 11 during normal operation; they are provided for diagnostic control and to initialize the horizontal control RAM.

Register 2 is the horizontal control RAM location addressed by AAR[0:7]. The horizontal control RAM is loaded only during controller initialization.

Doing an OUTPUT to register 3 does not cause data to be transferred from the D0, but loads IAR from START.

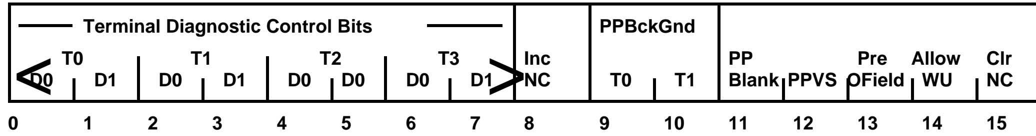
Registers 4 and 5 are the cursor control registers for channels 0 and 1. These registers are loaded by the microcode during every scan line preceding a line in which the cursor is visible.

Registers 6 and 7 are the cursor memory locations addressed by C0Addr and C1Addr. It is only necessary to load the cursor memory when the cursor bitmap is changed.

Registers 10b through 14b are the data buffers for the four display channels. These buffers are loaded with data to be serialized as video.

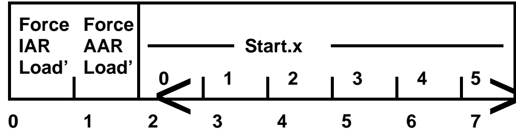
0:

Control Register _ OutD[0:15]



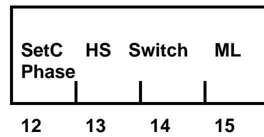
1:

Buffer Start _ OutD[0:7]



2:

Horizontal Control Ram [AAR[0:7]] _ OutD[12:15]

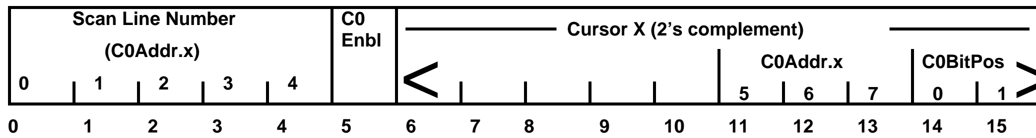


3:

IAR[0:5] _ Start[0:5]

4:

Cursor 0 _ OutD[0:15]



5:

Cursor 1 _ OutD[0:15]

6:

Cursor Memory 0 [C0Addr[0:7]] _ OutD[12:15]

7:

Cursor Memory 1 [C1Addr[0:7]] _ OutD[12:15]

10:

Buf0 [IAR[0:5]] _ OutD[0:15], IAR _ IAR+1

11:

Buf1 [IAR[0:5]] _ OutD[0:15], IAR _ IAR+1

12:

Buf2 [IAR[0:5]] _ OutD[0:15], IAR _ IAR+1

13:

Buf3 [IAR[0:5] _ OutD[0:15], IAR _ IAR+1

Figure 7.2.2 Output Register Functions

7.2.3 Timing and Sync Generation

The basic frequency source in the UTVFC is a crystal oscillator which operates at the video bit rate. This frequency is divided by four in a two-bit Gray code counter to form the signals ClkA, ClkB, and NClk (nibble clock). Most register transfers within that portion of the UTVFC synchronized to the video rate occur on the fall of NClk. The Gray counter sequence is:

ClkA	ClkB	NClk
0	0	0
0	1	0
1	1	0
1	0	1

Note that ClkA and ClkB are in quadrature, and that NClk falls when ClkA rises, which causes the data to the terminal to change in accordance with the terminal protocol. Internally, ClkB runs continuously, but the signal DropClockB will cause it to be suppressed at the terminal cable drivers.

The D0 microprogram can control the clock generator using the AllowWU, IncNC, and ClrNC bits of the control register. If AllowWU is true, the crystal oscillator is enabled. If AllowWU is false, the oscillator is disabled. While the oscillator is disabled, an OUTPUT to the control register with ClrNC true resets the Gray counter, and OUTPUT with IncNC true increments the counter. Four OUTPUTS generate one Nclk.

The logic that generates the synchronization signals and controls transmission of data to the terminal is shown in figure 7.2.3. The horizontal control RAM is the principal source of control signals. This RAM is addressed by the Active Address Register (AAR), which also addresses the data buffer whose contents are currently being sent to the terminals. Each scan line is divided into two segments by the signal ControlPhase. When ControlPhase is false, data are sent to the terminals. As each nibble is transmitted, AAR is incremented, which accesses the next address in the horizontal control RAM. During this segment of the scan line, the signals HS and Switch are forced to zero by the multiplexer on the input of the horizontal control register. Only the signals ML (associated with vertical sync generation) and SetCPhase are determined by the RAM contents.

When a RAM location with SetCPhase=1 is accessed, the ControlPhase flip-flop is set. ControlPhase forces the data being sent to the terminal to zero (blanking the display), and also switches the horizontal control register input multiplexer so that the signals HS and Switch are determined by the contents of the RAM. AAR continues to increment until a RAM location with Switch=1 is accessed. When Switch occurs, ControlPhase is cleared, AAR[0:7] is loaded from Start[0:5],,0, and the cycle repeats. At switch time, the synchronized control register is loaded from the control register in preparation for the next scan line. Switch also complements the Even/Odd Line flip flop, which causes the roles of the inactive and active data buffers to be reversed.

During ControlPhase, a horizontal sync signal will be sent to the terminal. The width of the sync pulse and its relationship to the blanking interval are determined by the contents of the

horizontal control RAM. Figure 7.2.4 shows the timing of events in the vicinity of a horizontal sync pulse in detail. The signal (HS or VS) sets the SendControl flip flop which causes DropClockB to be set one nibble time later. DropClockB causes the terminal to interpret the data lines as control information. SendControl is used in the data section of the UTVFC to gate HS and VS to the data lines (the bits in the most significant byte of the control register are also sent). When (HS or VS) becomes false, SendControl is extended for one extra nibble time to allow the control register in the terminal to be cleared.

Generation of vertical sync is the responsibility of the UTVFC microprogram. Both interlaced and non-interlaced displays may be driven by the UTVFC. The microcode must change the PPVS bit in the control register during the scan line preceding the one in which VS is to change. The state of the OddField bit determines whether VS will change on the falling edge of HS (OddField=1), or on the falling edge of ML (OddField=0).

The UTVFC requests a wakeup at the beginning of every horizontal line (at the fall of SWITCH). The wakeup request remains set until explicitly cleared by the IOStrobe function, except that a wakeup request will not be issued if the UTVFC's task is running. The AllowWU bit in the control register unconditionally disables wakeup requests.

Figure 7.2.3 Control and Sync Generation

Figure 7.2.4 Data and Control Multiplexing

7.2.4 Data Buffering

The UTVFC contains two data buffers, each of which holds a full scan line for four terminals. One buffer is loaded by the D0 while the other is being transmitted to the displays. The signal Switch, which is generated by the control logic at the end of each scan line, ping-pongs the buffers. Figure 7.2.5 shows the data buffers and their interconnection.

There are two address registers for the data buffers. IAR, the Inactive Address Register, addresses the buffer currently being loaded by the D0. AAR, the Active Address Register, addresses the buffer being read to the terminals. AAR also addresses the horizontal control RAM.

IAR is a six-bit register. It supplies the most significant bits of the inactive buffer address; the low two bits are functions of Oaddr.6 and Oaddr.7. IAR is incremented by one each time the D0 delivers a word to the inactive data buffer. IAR is loaded from the register START when IAR=63d and the D0 delivers a word to the buffer. START should contain 64d minus the number of words to be displayed on a scan line, so if IAR initially contains the value in START and the D0 delivers (64d-START) words during a scan line, IAR will end up with its initial value. This makes it unnecessary for the D0 to modify IAR unless START is changed. A bit in START (ForceIARLoad') forces IAR to be loaded from START each time the D0 delivers a word to the buffer. This is provided for testing and initialization.

AAR is an eight bit register. It is incremented by NClk. AAR is loaded with 4*START by Switch. There is a bit in START (ForceAARLoad) which loads AAR on every NClk. This is provided for testing, and to initialize the Horizontal Control RAM.

During each scan line, the processor delivers the data for the next scan line for the first terminal, then delivers a full scan line for the second terminal, and so forth. When the data are being transmitted to the terminals, the first nibble for all four terminals is read, then the second nibble, and so on. To accomplish this, each four-bit section of both buffers is independently addressable, and there is logic at the input and at the output of the buffers to shift the data appropriately.

When the D0 delivers a 16-bit word to the buffer, the data are cycled so that the most significant nibble for terminal n is placed in buffer n. In addition, the low order two bits of the buffer address are set as a function of the terminal number so that nibble n of the data for all terminals is stored in buffer location n. The buffer address is then incremented by 4 (by incrementing IAR by 1). This operation results in the data being located in the buffer as shown in figure 7.2.6.

When data are removed from the buffer, the words are accessed sequentially (since each word contains one nibble for each of the four terminals), but the data must be cycled to remove the cycle introduced when the buffer was loaded. This is done by the output shifter.

Buffer:

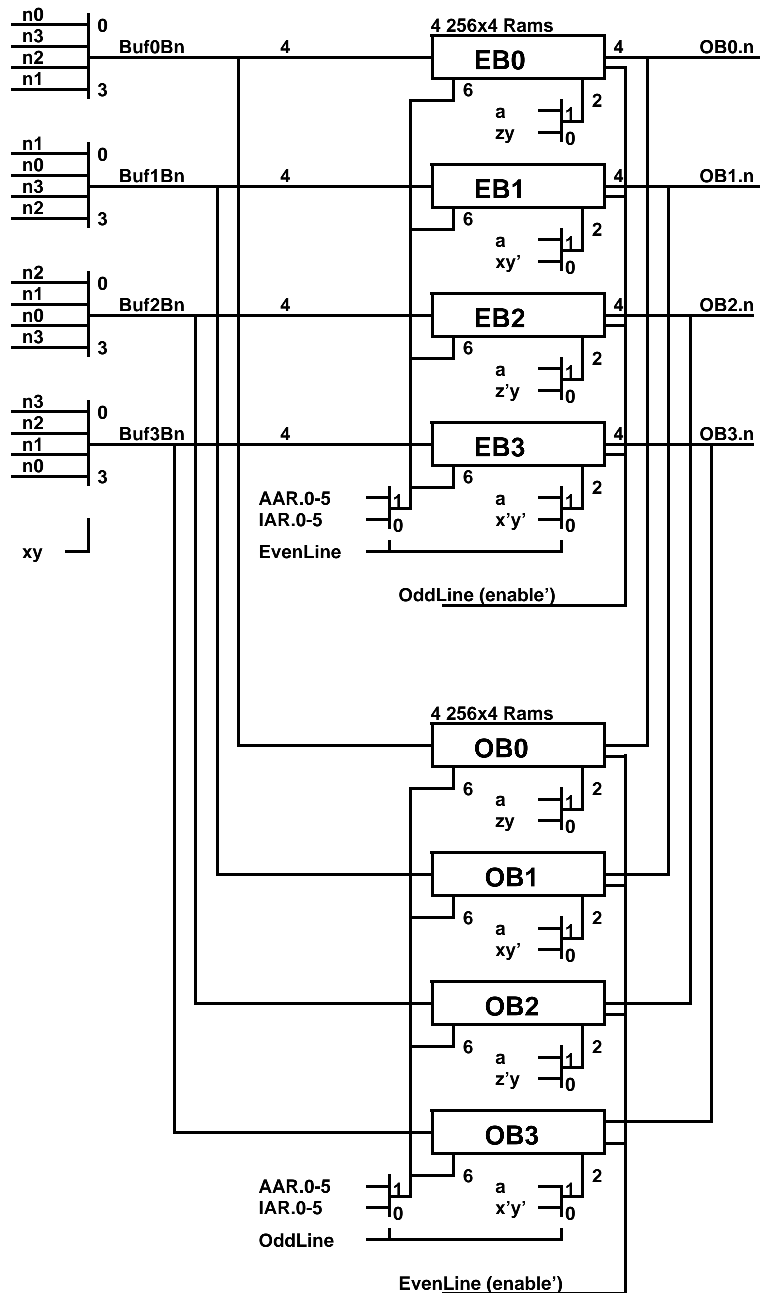
Buffer Location	B0	B1	B2	B3
Bit Number:	0 3	4 7	8 11	12 15
3	T1 N3	T2 N3	T3 N3	T0 N3
2	T2 N2	T3 N2	T0 N2	T1 N2
1	T3 N1	T0 N1	T1 N1	N2 N1
0	T0 N0	T1 N0	T2 N0	T3 N0

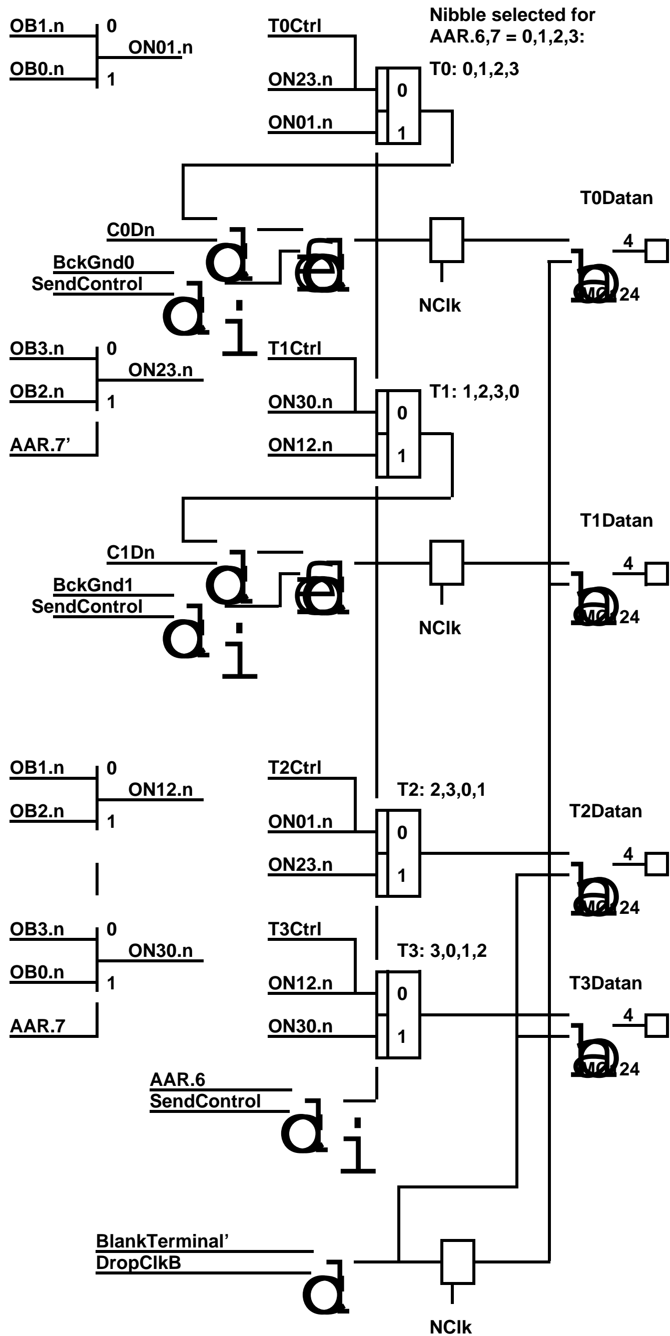
Tx = Terminal x
Ny = Nibble y

Figure 7.2.6 Buffer Contents as a function of location

Control information is multiplexed with the video data between the first and second rank of the output shifter. The first rank of the shifter is disabled by SendControl, and the terminal control bits, HS, and VS are tri-stated onto these lines. The second rank of the shifter is composed of multiplexer-latches that drive the data line level converters for channels 2 and 3 directly. Channels 0 and 1 have hardware to mix the cursor with the video data between the shifter register and the line drivers.

Channels 0 and 1 also have logic to control the polarity of the background video. If the BckGnd0/1 bit in the control register is zero, the associated channel will display white for zero bits in memory. The cursor is ORed with the video data before the background polarity is selected, so ones in the cursor memory always correspond to a polarity opposite of that of the background. For channels 2 and 3, zeros in memory correspond to a blanked display. The PPBlank bit in the control register causes the terminal to be blanked by disabling the line drivers (but the line drivers are enabled when control is transmitted). Because of the delay introduced by cursor mixing, the data and clocks for channels 0 and 1 have an extra level of latching, which delays them by one nibble relative to channels 2 and 3. This is only important for terminals (e.g. color terminals) that use more than one channel to drive a single display.





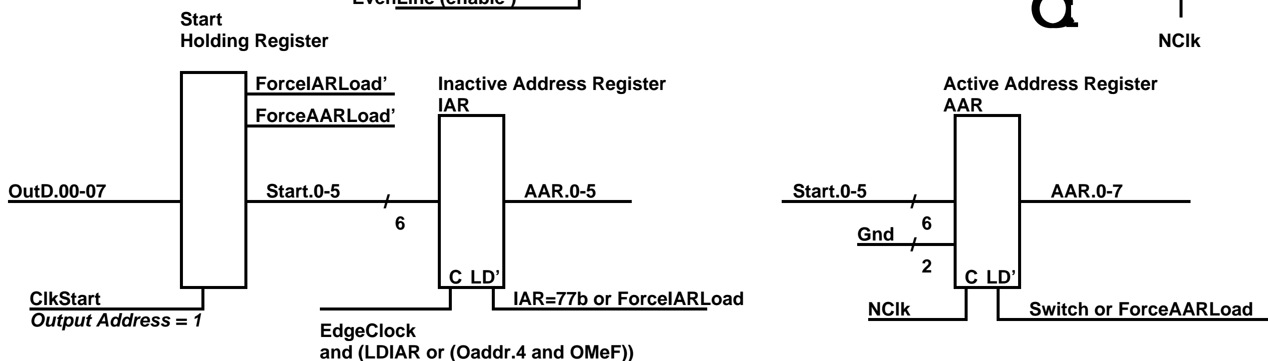
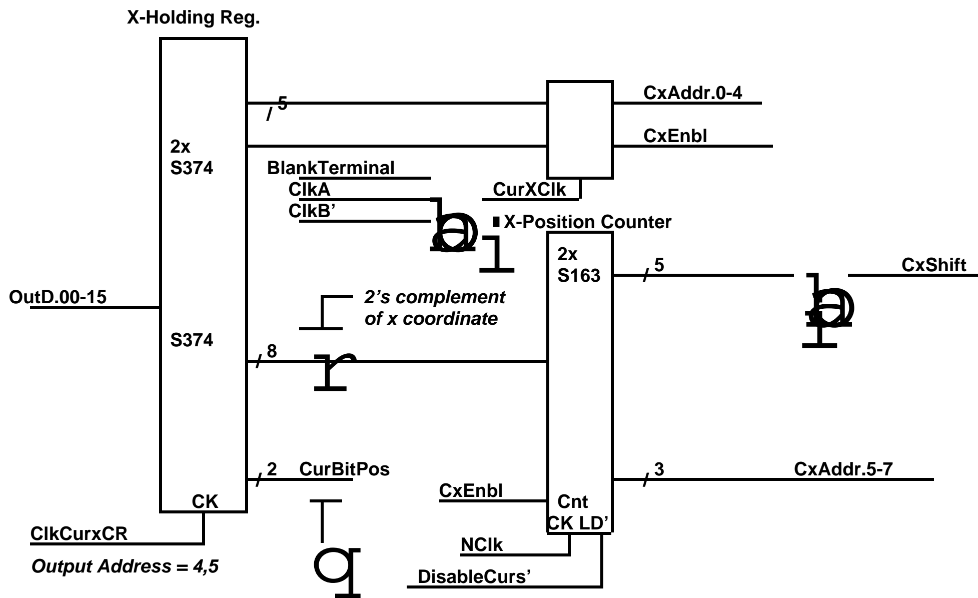


Figure 7.2.5 Data Buffers and Addressing

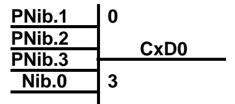
7.2.5 Cursor

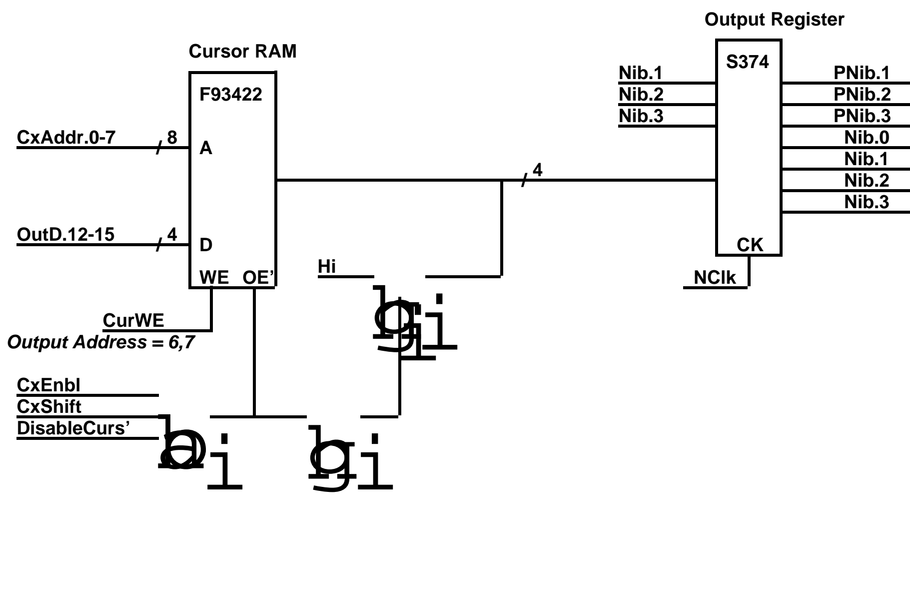
Channels 0 and 1 of the UTVFC contain logic to generate 32x32 bit cursors (see figure 7.2.7). A 256x4 RAM holds the cursor video pattern for each channel. This RAM is addressed by CxAddr[0:7]. Bits 0-4 of this register select one of 32 eight-nibble cursor segments to be displayed on a particular scan line. In the scan line preceding the one in which cursor segment S is to be displayed, the microcode will load the cursor control register with S in bits 0-4, with a one in bit 5 to enable the cursor logic, and with the negative of the x coordinate of the leftmost bit of the cursor in bits 6-15. During the next control phase, this quantity is loaded into the cursor x position counter. During the subsequent scan line, the x position counter is incremented by NClk until its most significant five bits are zero, at which time CxShift becomes true. During the next eight nibble times, the cursor RAM outputs are enabled, and the eight nibbles of cursor segment S are loaded into the output register (at all other times, the cursor RAM outputs are forced to zero). The final shift required to place the cursor on the scan line with a precision of one bit is provided by the cursor shifter, controlled by the low two bits of the x holding register. Up to three bits of a given nibble may be shifted so that they must be merged with the following nibble; these bits are recirculated through the output register. The cursor video (CxD0-3) is ORed with the main video as described in figure 7.2.4.

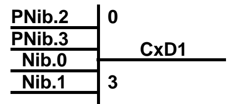
The cursor memory must be loaded during vertical retrace, since the value in the x position counter used to address the cursor RAM can only be set if PBlank=1. During this time, the x position counter is loaded from the x holding register, so the microprogram can load an address into CxAddr[0:7], then write the data for that location into the cursor RAM (a total of 512 OUTPUT operations are required to load the RAM with the full 32x32 cursor).

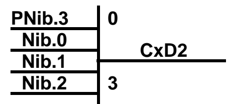


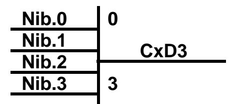
Cursor Shifter











CxBitPos 2

Figure 7.2.7 Cursor Logic (channels 0 and 1 only)

7.2.6 Backchannel

The state of the backchannel message bit from the terminal is provided on the IOAttn line. IOAttn will be true if the message bit is a logical one. The terminal that is selected to deliver its message is determined by IAddr.6-7. The intent is that the microcode will send the first block of data to a terminal with a memory operation (which will set IAddr.6-7 to the terminal number), then branch on IOAttn before doing a task switch or other memory reference. Since IOAttn must be sampled every scan line, this means that the microcode may have to do an innocuous memory operation to set the terminal number if it has no data to deliver.

7.2.7 Controller Identification and Diagnostic Input

The UTVFC provides only one Input register. This register is transmitted when any of the sixteen registers available to the UTVFC's task are accessed. The format of the register is shown in figure 7.2.8. The most significant byte of the register contains 2, the UTVFC's unique ID number. Bits 8 through 12 are determined by wiring on the platform containing the crystal oscillator, and indicate the frequency of the bitclock. Bits 13-14 are 10, and bit 15 is the Test Bit.

The Test Bit is the output of a forty-bit shift register whose inputs are connected to a number of internal signals in the controller (refer to the logic diagrams). When an INPUT is done from a register with address >3 (mod 16) these bits are loaded into the shift register. When an INPUT is done with address <4, the register is shifted. The intent is that a (Mesa) program can be written that controls the UTVFC clock and checks most of the internal logic of the UTVFC via this path.

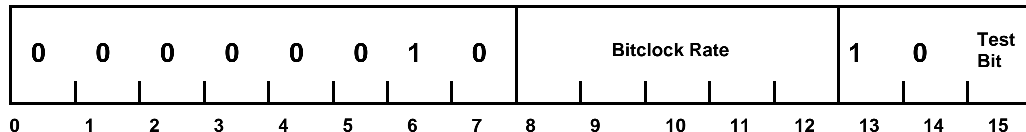


Figure 7.2.8 Input Register Format

8.0 Rigid Disk Controller

8.1 Introduction

The Rigid Disk Controller (RDC) is capable of operating up to four Shugart SA4000 disks. The RDC controls disk formatting, data buffering, error checking and ECC syndrome generation, data transfers to and from the D0, and generation of wakeup requests for the disk microcode. The RDC also contains self-test logic with which a program can simulate the disk drive and exercise the controller on a clock-by-clock basis.

8.2 Disk Characteristics

The principal characteristics of the SA4000 disk are shown in Table 8.1. The unformatted capacity of the disk is 29.1Mbytes. The useful data capacity after formatting is 45248 sectors * 512 bytes/sector = 23.1Mbytes. Each sector is formatted into three records, a *header,* a *label*, and a *data* record. Details of sector formatting are shown in Table 8.2. Each record in a sector may be independently read, written, verified, or not accessed, subject to limitations imposed by a command decoding PROM in the controller and to the restriction (imposed by read-amplifier recovery time) that a record cannot be read following a write to the preceding record.

Table 8.1

Number of tracks	202
Number of surfaces	4
Heads/surface	2
Bytes/track	18000
Sectors/track	28
Total sectors	45248
Transfer rate (peak)	7.1Mb/sec (2.2 us/word)
Rotation time	20ms
Seek time (min/avg/max)	20/65/140 ms

Output Registers

0:
General Reset (data is ignored)

1:
Drive/Head Register

Drive Sel		Head Sel			
0	1	0	1	2	3
10	11	12	13	14	15

2:
ErrReset (data is ignored)

3:
DevOpReg

Allow Wake	Seek	Dir	Inh Hdr Abrt	Header		Label			Data		
				Write	Read	Write	Read	Ver	Write	Read	Ver
4	5	6	7	8	9	10	11	12	13	14	15

4:
Buffer[MemBufAdr]_

5:
TestReg

Test Mode	Rdy	Wrt Fault	Seek Comp	Trk0	Index	Sect Pulse	Rd Data	Dev Clk
7	8	9	10	11	12	13	14	15

6:
MemBufAdr

MemBufAddr							
8	9	10	11	12	13	14	15

7:
PrimeIdata (data is ignored)

Input Registers

0:
Controller ID

0	0	0	0	0	Srv Late	Srv Late'	1	0	0	0	0	Wake Req	1	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

1:
Status

0	0	0	0	0	Srv Late	Srv Late'	Rate Err	Sect 0	Trk0	Seek Comp	Dev Sel OK	Buf Err	Rd Err	Wrt Fault	O Fault
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

12b:
Data Loopback

Seq Adr Xfer'	Step	Dir	Flt Clr	Drive Sel 1	Drive Sel 2	Drive Sel 3	Drive Sel 4	Hd Sel 8	Hd Sel 4	Hd Sel 2	Hd Sel 1	Rd Gate	Wrt Gate	Wrt Data	Wrt Clk
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

17b:
_Buffer[MemBufAdr]

Figure 8.3 RDC Output and Input Registers

Table 8.2 - Sector Format (as written)

Use:	Bytes:	
Preamble	13	0's
Sync	1	00001010
Header	4	
Header CRC	2	
Postamble	5	The write gate is on only during the first byte
Preamble	10	0's
Sync	1	00001010
Label	16	
Label CRC	2	
Postamble	5	The write gate is on only during the first byte
Preamble	10	0's
Sync	1	00001010
Data	512	
Data ECC	4	
Postamble	3	The write gate is on only during the first byte
Recovery gap	53	
Total Sector	642	

8.3 D0 - Controller Interface

The D0 and the RDC communicate using seven of the sixteen output registers and four of the sixteen input registers available to a controller. Figure 8.3 shows these registers. In the descriptions that follow, page references refer to the RDC logic diagrams.

8.3.1 Output Registers

The registers used to transmit information from the D0 to the RDC are:

0: GeneralReset (see "ErrReset" below).

1: Drive/Head: This register is buffered and sent directly to the drive(s). Values of the head field >7 select up to 8 optional fixed heads. After a drive change (or a Write Fault), ~2 sectors must pass before the signal DevSelOK becomes true (p13). DevSelOK inhibits writing.

2: ErrReset: The RDC may be reset in several ways. RUN' OR GeneralReset = Reset, which sets the IOStrobe f-f (p2), clears the Wakeup Counter (p3), the AllowWake f-f and the Seek Counter (the counter is loaded with 17b) (p4), portions of the Format Sequencer (p5) and the Buffer Control Sequencer (p6) the DevSelOK counter (p13), and the Read Gate (p14). Reset OR ErrReset also clears the WriteFault f-f in the drive, the RateError f-f (p13), the IntOp register (p4), the ReadError f-f (p11), and the Error Register and OFault f-f (p12).

3: DevOpRegLd: The DevOp register holds disk commands and the AllowWake bit (which must be set at each OUTPUT if wakeups are to be enabled). The command bits in the DevOp register are discussed below.

4: BufDataLd: Data directed to this register is written into Buffer[MemBufAdr].

5: TestRegLd: This register allows a program to simulate all signals generated by the disk. For normal operation, bit 7 of this register should be cleared.

6: MemBufAdrLd: This register holds the current pointer into the RDC's data buffer.

7: PrimeIData: This "register" must be accessed with an OUTPUT by the microcode before reading the first word of each sector from the data buffer. It loads the Idata register (p8) from the data buffer and increments MemBufAdr (p7). The data transmitted by the OUTPUT is ignored.

8.3.2 Input Registers

The four RDC input registers are:

0: Controller ID: This register returns the controller ID (1407b). It is valid only immediately after a reset, since it contains the ServiceLate and WakeRequest bits. The WakeReq f-f will be set by the SectorWake sgnal at the next sector pulse, making ID=1417b (although since AllowWake=0, no request will be made to the D0).

1: Status: This register contains the controller and drive status. The significance of the bits is as follows:

bits 0-4: 0

bit 5*: ServiceLate: This bit is set if DevOp is loaded during the header or label field, or if <16 words have been delivered to the buffer when the header sync byte is found [Precisely, if MemBufAdr[2:3] = 0 when sync is found].

bit 6*: ServiceLate': This bit is here to ease the problem of Idata Parity generation.

bit 7*: RateError: This bit is set if the Buffer Control Sequencer bit "RateErrorPossible" is set and the '2' bit of the Wakeup counter = 1. RateErrorPossible is only asserted during the data transfer phase of a WriteData or VerifyData operation; during these operations, if the processor falls more than one wakeup behind the disk's need for data, a rate error will be indicated.

bit 8: Sector0: This bit is true for the duration of physical sector 0, i.e., from index to the next sector pulse.

bit 9: Track0: This is a signal from the drive, asserted when the heads are at Track0.

bit 10: Seek Complete: This is a signal from the drive, indicating that a previous seek has completed. After this signal becomes true, the microcode must wait an additional 28 sector times (20ms) for the head arm to settle before doing a data transfer.

bit 11: DevSelOK: This signal indicates that a drive is selected, is ready, does not have a WriteFault, and has delivered at least two sector pulses since selection.

bit 12*: BufErr: This bit is set during a write if data taken from the RDC's buffer has bad parity.

bit 13*: RdErr: This bit is set if the CRC fails during a header read (unless InhHdrAbrt = 1), or during a label read or verify, or if the ECC indicates an error during a data read or verify.

bit 14*: WriteFault: This is a signal from the drive, indicating that a problem that would cause errors during a write exists.

bit 15*: OFault: This line is a latched version of the OFault line from the D0, which indicates that a double-bit error was detected during a memory reference.

Bits marked with * set Abort and remain set until explicitly cleared by ErrReset or GeneralReset..

12b: This register provides loopback for all signals transmitted to the drive. The sampling point for these signals is the disk cable, so all the controller logic may be tested (even without a disk).

17b: This register presents Buffer[MemBufAdr]. When accessed, MemBufAdr is incremented and the Idata register is loaded from the buffer.

8.3.3 Seek Control

The head positioning in the SA4000 is done by a stepping motor driven by pulses from the controller. The direction of head motion is controlled by a single bit. The drive contains a "buffered step" option which absorbs these step pulses more rapidly than the motor can move, and drives the motor at the correct rate; the RDC requires this option. Stepping is performed by two bits of the DevOp register. When an OUTPUT is done to the DevOp register with the seek bit (bit 5) equal to one, a single step pulse is issued to the drive. The value of DevOp.06 determines the direction; a one implies an inward step, i.e. toward higher track numbers. The RDC contains logic to generate step pulses of the proper width. The disk microcode must not issue OUTPUTs with seek = 1 more frequently that every 16 microinstructions, or this logic will be defeated. During a seek sequence, when the microcode has delivered the last seek pulse, it must clear the seek bit in the DevOp register, since this bit inhibits sector wakeups, and also forces the Format and Buffer Control sequencers to be idle independent of the command fields of DevOp. The Direction bit should not be changed for ~2us after the last seek pulse is sent, or the disk will be confused.

8.3.4 Disk Commands

The RDC is capable of executing four basic commands during each record of a sector: *Read*, in which data is transferred from the disk to memory, *Write,* which is the inverse, *Verify*, in which data from memory is compared with the data on the disk, and *NoOp,* which does nothing. If an operation in a record fails due to an error or to a non-compare on a Verify operation, the *Abort* flip-flop is set, and commands for subsequent records will be converted to Nops. During the header field, Verify is the only operation provided; the microprogram must supply data to be checked, and if if operations on subsequent records

are desired even if the check fails, the InhHdrAbrt bit in DevOp must be set. The label record is handled similarly. The microprogram must always supply data to be verified, but if the command is Read, the setting of Abort is inhibited. (Note that the operations on the header and label records are logically identical, but different mechanisms are provided to obtain the desired effect).

The basic commands are implemented by two automata, the Format Sequencer (section 8.4.3) and the Buffer Control Sequencer (section 8.4.4). The combination of legal commands for all records is determined by a PROM that examines the IntOp register and maps the commands into starting addresses for the sequencers if the combination is legal, or into Nops if the combination of commands is not legal. [Note: there is no way for the microcode to determine that an illegal command has been supplied to the controller]. The command sequences that are currently implemented in the RDC are:

Header	Label	Data
Nop	Nop	Nop
Verify	Nop	Nop
Verify	Write	Nop
Verify	Write	Write
Verify	Read	Nop
Verify	Read	Read
Verify	Read	Verify
Verify	Read	Write
Verify	Verify	Nop
Verify	Verify	Read
Verify	Verify	Verify
Verify	Verify	Write
Write	Nop	Nop

8.3.5 Wakeups and IO Attention

Wakeups are initiated by the RDC under two circumstances: A sector wake is generated at the end of the data record (~58 us before the physical sector pulse) if the *basic sequencer operation* (section 8.5) is Nop, Read, or Write [Note that Verify Data does *not* generate a sector wakeup. Instead, it generates an extra data wakeup, which is logically the same]. When the seek bit in the DevOp register is set, sector wakeups are inhibited. Sector wakeups originate in the Format Sequencer.

Data wakeups are generated by the Buffer Control Sequencer. The exact timing for data wakeups depends on the record and the operation, and is described in section 8.4.7. Data wakeup requests are accumulated in a four-bit up-down counter that is constrained to count between 0 and 17b. When the Buffer Control Sequencer issues a wakeup request, the counter is incremented; when the D0 issues an IO Strobe, the counter is decremented. The actual wakeup request is transmitted to the D0 if the disk task is not running and if the wakeup counter is nonzero.

When the disk task is running, IO Attention is transmitted to the D0 if the RateError or Abort bits are true [Note that Abort does not appear in the status register. The only way that a verification error can be inferred is to see IOAttn true, then read the status register and check that the other reasons for setting

Abort are not true].

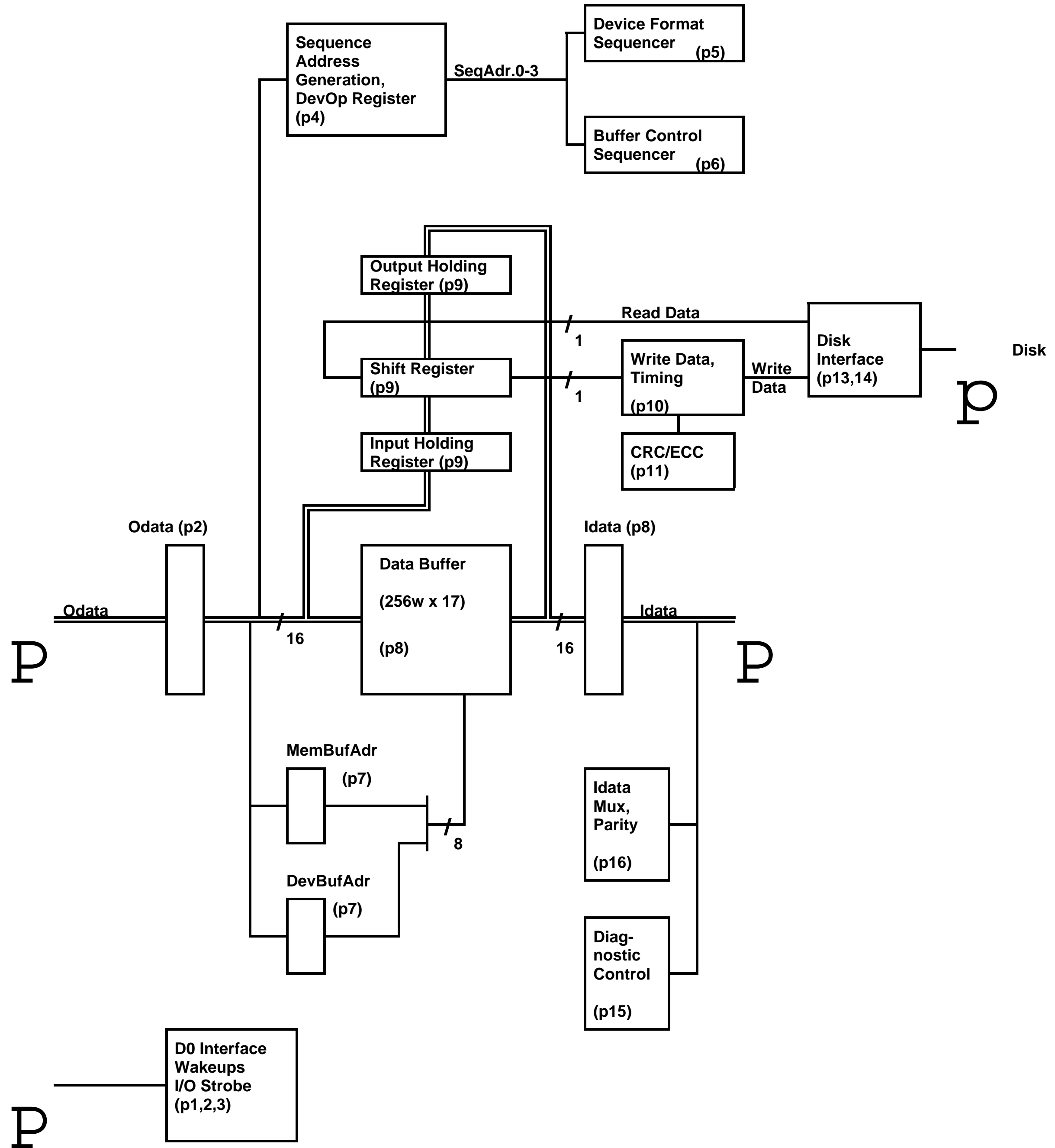
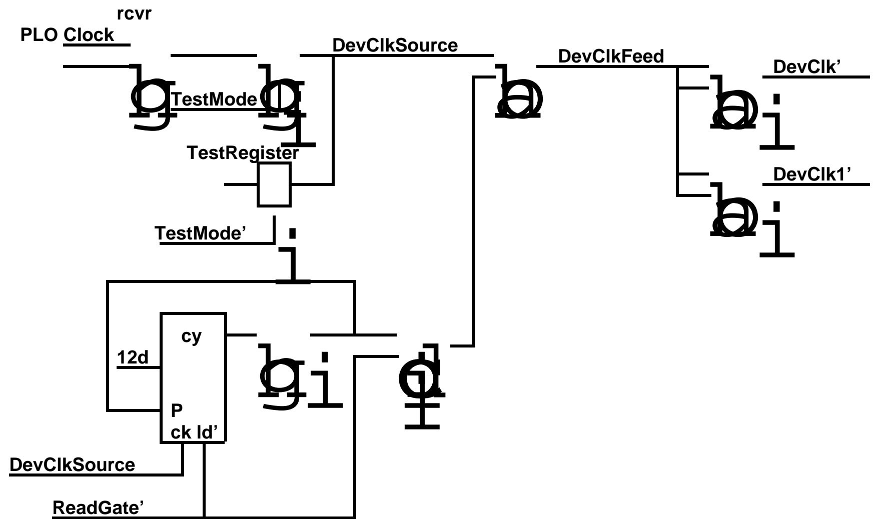
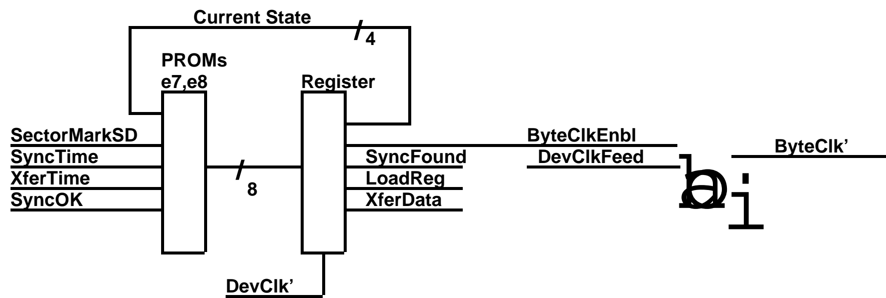


Figure 8.4

RDC Organization





e7,e8 Proms:

Inputs:								
SectorMarkSD	1	0	0	0	0	0	0	0
SyncTime	x	0	0	x	x	x	1	1
XferTime	x	0	0	1	1	1	0	0
SyncOK	x	x	x	x	x	x	0	1
CurrentState	x	14-5, 7-13	6,14	15-5, 7-13	6	14	x	x
Outputs:								
ByteClkEnable	1	0	1	0	1	1	0	1
SyncFound	0	0	0	0	0	0	0	1
LoadReg	0	0	0	0	0	1	0	1
XferData	0	0	0	0	0	1	0	1
NextState	15	n+1	n+1	n+1	7	15	0	0

Figure 8.4.1: RDC Timing

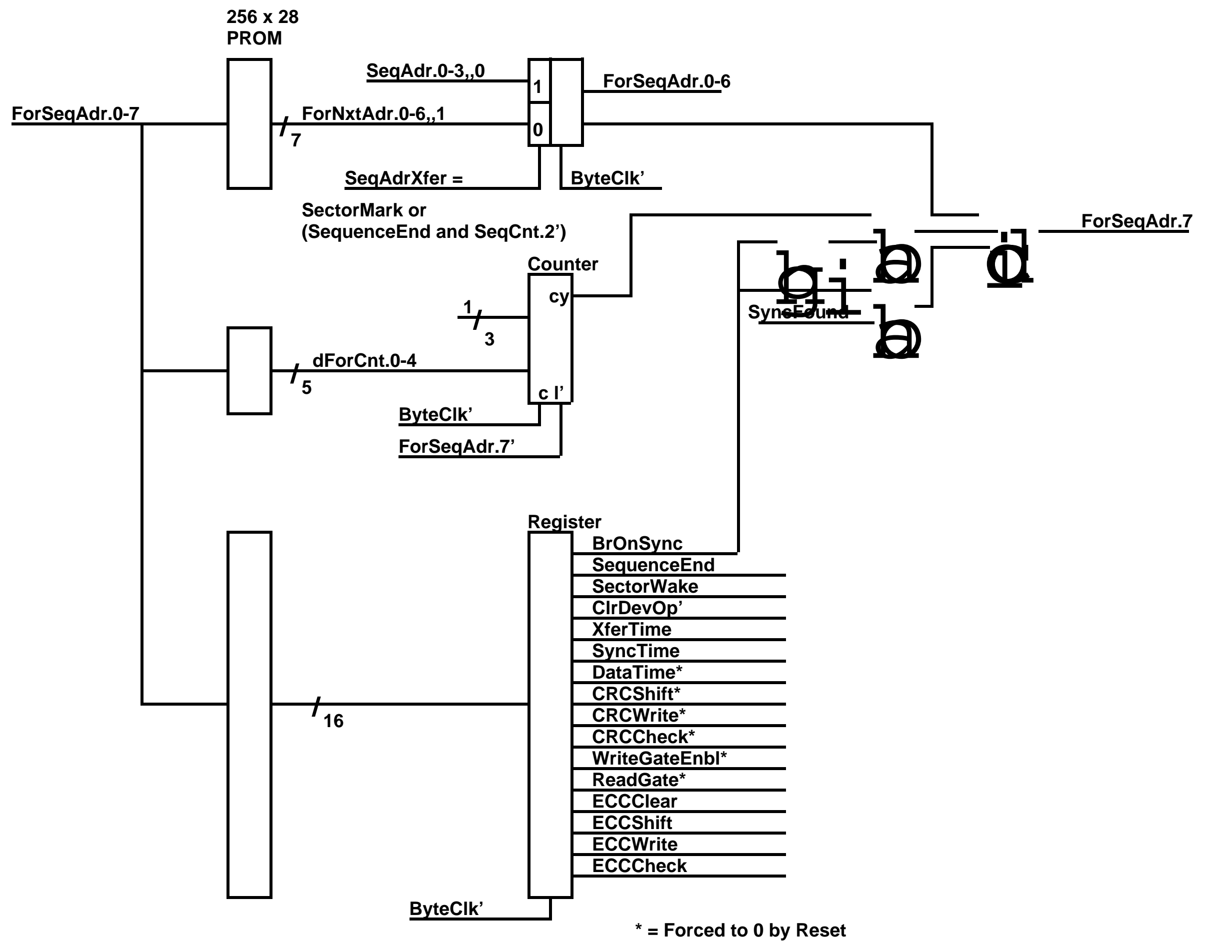


Figure 8.4.3: RDC Format Sequencer

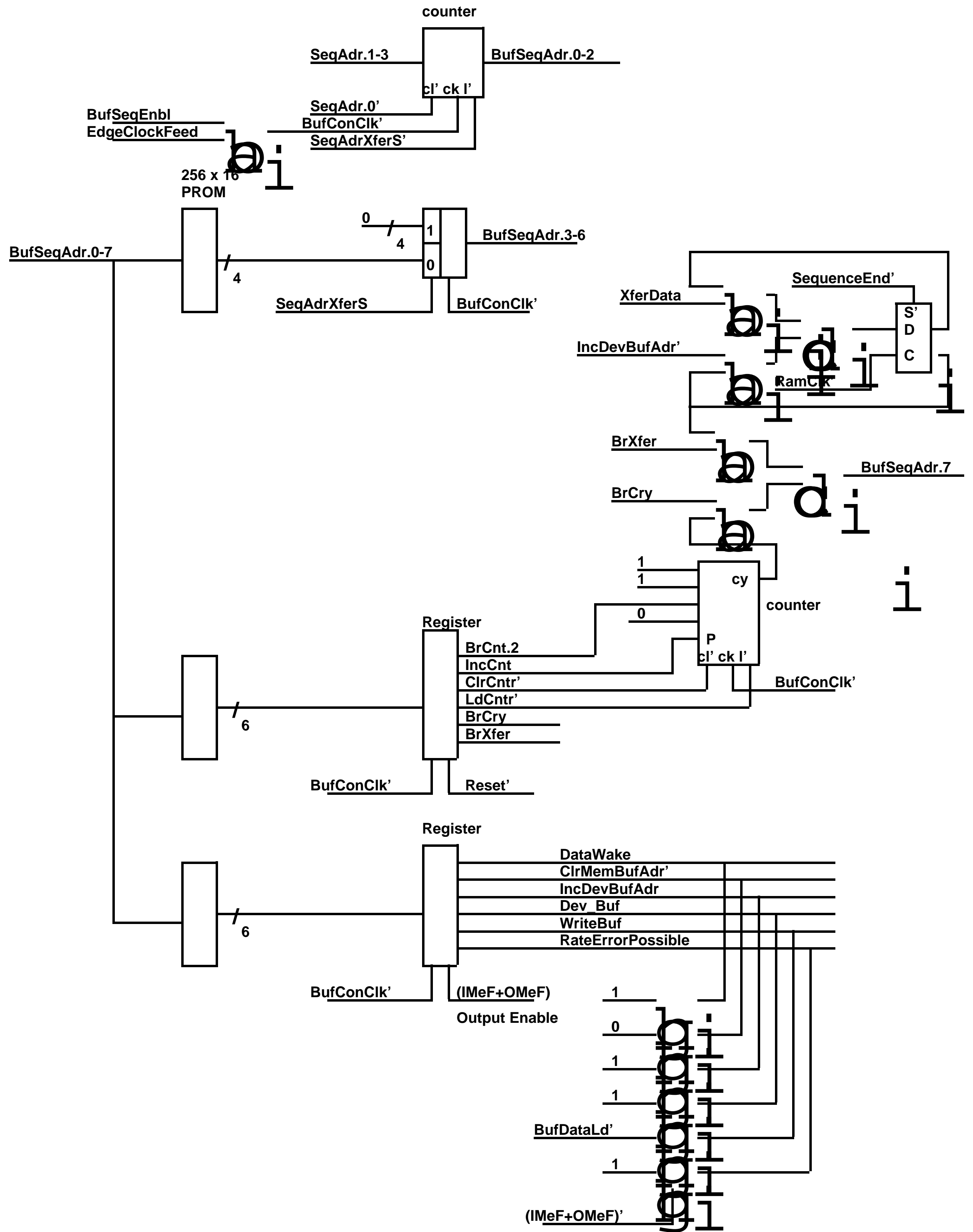


Figure 8.4.4: RDC Buffer Control Sequencer

8.4 Hardware Organization

The overall organization of the RDC is shown in figure 8.4. The page numbers in the figure refer to pages in the logic diagrams. The RDC contains a single data buffer of 256 16-bit words plus a parity bit. The buffer is addressed by two pointer registers, one used by the D0 (MemBufAdr), and one used by the controller (DevBufAdr). During a disk write, data flows from the D0 via the Odata bus into the Odata register, then into the buffer. As the disk requires data, it is transferred from the buffer to the output holding register, then into the shift register, where it is serialized and sent to the disk. During a read, data is deserialized in the shift register, placed in the input holding register, and sent to the buffer as new data arrives. From the buffer it is sent to the Idata register, then to the D0 via the Idata bus.

The D0 has highest priority for buffer access. When it needs to read or write the buffer, the Buffer Control Sequencer clock is witheld for a cycle [note that this implies that if 16 word transfers are done by the disk microcode, the D0's efective clock rate (including cycles lost for BranchBurp) cannot be slower than ~125 ns, since the disk transfers a word every 2.2 us, and the Buffer Control Sequencer must be able to access the buffer once per word].

During a write or verify operation, data from the D0 is parity-checked by serial logic associated with the shift register. During a read, this logic generates the parity bit that will be sent to the D0.

The RDC writes and checks a 16-bit CRC on the header and label records, and a 32-bit ECC on the data record. The CRC and the ECC are generated and checked by logic in the controller. If the ECC fails, the final syndrome can be recovered by the disk microcode so that a program can correct the data record.

8.4.1 Timing

The derivation of the timing signals in the RDC is shown in figure 8.4.1. The basic clock is the disk PLOClock, which is read from a timing track unless the disk is reading, when it is derived from the recorded data. After level conversion, the clock is gated with the output of a counter which ensures that when ReadGate _ 0, the clock is disabled for four cycles to avoid any discontinuities.

DevClk occurs once per bit (1.1 us); the disk ByteClk (which occurs every 8 bits) is generated by a PROM/register automaton. This logic also creates the XferData and LoadReg signals which occur once per word [these signals have identical timing], and the SyncFound signal.

8.4.2 Sequence Address Generation

The operations that will be performed during each of the three records of a sector are controlled by a two-bit and two three-bit fields of the DevOp register. The disk microcode must load this register with a command before the physical sector pulse occurs (~58 usec after the sector wakeup). When SectorMarkSP occurs, the eight bits of the DevOp register are mapped into six bits, and loaded into the IntOp register. The mapping is performed so that the commands for each record will be encoded into a single bit. The encoding is:

IntOp bit		DevOp bit
HdrWrt	_	WriteHeader (DevOp.08)
HdrRd	_	ReadHeader (DevOp.09)
LblWrt	_	WriteLabel (DevOp.10)
LblRd	_	ReadLabel (DevOp.11) OR VerifyLabel (DevOp.12)
DataWrt	_	WriteData (DevOp.13) OR Verify Data (DevOp.15)
DataRd	_	ReadData (DevOp.14) OR VerifyData (DevOp.15)

The six bits of the IntOp register and two bits of a counter that indicate the record about to be processed (SeqCnt.0,1) are mapped by a 256 x 4 PROM (g11) into a four-bit Sequence Starting Address, SeqAdr.0-3. This address is used to determine the starting address of the code for the record in the Buffer Control and Format Sequencers.

8.4.3 Format Sequencer

The Format Sequencer is shown in figure 8.4.3. This logic controls the formatting of data on the disk, and is synchronous with the disk ByteClk. The sequencer is implemented as a finite state machine controlled by a 256x28 PROM. The PROM produces the most significant seven bits of the next sequencer address to be accessed, five bits of data used to load a counter described below, a test bit, and fifteen control bits.

The flow of control in the format sequencer is determined by the BrOnSync bit. The least significant PROM address bit, ForSeqAdr.7 = [(ForCntCry and BrOnSync') or (SyncFound and BrOnSync) or (FirstSequenceLocation)].

The format sequencer counter is loaded whenever ForSeqAdr.7 = 1. This counter contains five bits, and is loaded from five bits of the sequencer PROM (dForCnt[0:4]). The counter is incremented by ByteCLK, which also clocks the sequencer output register.

The starting address of the sequence associated with a particular physical record is loaded into the address register at the start of the record. This address is SeqAdr[0:3],,0000, but the PROM location accessed is odd because of the inversion in the least significant bit of the address.

The sequencer will access the location corresponding to the first location of the sequence, and (because this address is odd) the counter will be loaded from the count field of the PROM.

When the sequencer is executing an instruction from an odd location of the PROM, the counter will have just been loaded. If the count value # 37b, the next instruction will be taken from an even location. If the counter was loaded with 37b, the next location will be taken from an odd location (since the counter will be producing a carry, forcing ForSeqAdr.7 to one).

When the sequencer is used to time events during a record, the normal situation is that an odd location (e.g. X) will point to an even/odd instruction pair (Y and Y+1) and the count field of X will contain 37b minus the number of ByteClk times that the sequencer is to execute instruction Y. The Next Address field of instruction Y will point to itself, so it will be executed repeatedly until the counter = 37b, at which time location Y+1 will be accessed. Location Y+1 will contain a new address and count, and the sequence will continue.

As mentioned earlier, if the odd location of a pair contains 37b in its count field, it will go directly to the odd location of the new pair, since the counter will produce a carry as soon as it is loaded.

When reading from the disk, the start of the sector is not precisely positioned, since the read clock (ByteClk) must be acquired by a phase-locked loop. The start of each record is indicated by a unique ’sync pattern’ written before the data, and there is logic to detect this pattern. When the sequencer is generating SyncTime, this logic is enabled, and the BrOnSync bit is provided to test the SyncFound signal that it generates. Once the sync pattern has been acquired, the sequencer is precisely aligned with the disk data, and can therefore determine when to generate the control signals associated with the transfer.

8.4.4 Buffer Control Sequencer

The buffer control sequencer is shown in figure 8.4.4. This logic controls the transfer of information between the data buffer and the two device holding registers. This sequencer is synchronous with the processor. It is implemented as a finite state machine controlled by a 256x16 PROM. The output bits of the sequencer are divided into three fields as follows:

1) BufSeqAdr[0:7]: This field selects the next address from the 256 word PROM. The three most significant bits of this register are loaded from SeqAdr[1:3] at the start of every disk record (Header, Label, Data, and

Recovery Gap), providing that SeqAdr.0 = 0. If SeqAdr.0 = 1, indicating that no data transfer will be done during the record, these bits are cleared. BufSeqAdr[0:3] are not changed during a record. BufSeqAdr[3:6] are loaded with zero at the start of every record. The least significant address bit, BufSeqAdr.7, is normally true, but may be made false by [(XferDataS and BrXfer) or (Counter=15d and BrCry)]. BrXfer and BrCry are two control bits that provide a conditional branch capability. XferDataS is a processor-synchronous version of the XferData signal from the Format Sequencer. It is set when XferData is asserted (indicating that a transfer between the buffer and one of the device data registers is required), and cleared when the Buffer Control Sequencer asserts IncDevBufAdr. XferDataS is also cleared by Sequence End.

2) Counter Control and Branch Bits. The Buffer Control Sequencer contains a four-bit counter that may be loaded, incremented, cleared, and tested for carry under control of five sequencer output bits. The counter and its control bits are local to the sequencer, and are not used anywhere else in the RDC.

3) Buffer and Wakeup Control bits. These six bits are the outputs of the sequencer used to control the data buffer and the wakeup logic. The bits are:

Data Wake: Increments the wakeup request counter.

ClrMemBufAdr: Clears the buffer address register used for D0-buffer transfers.

IncDevBufAdr: Increments the buffer address register used for device-buffer transfers. This register is also cleared by the ClrDevOpS bit from the Format Sequencer.

Dev_Buf: During disk writes, loads the buffer holding register. When the serializer needs a word, it will be loaded from this holding register under control of the Format Sequencer/Timing Generator.

WriteBuf: During disk reads, this bit causes the data in the serializer’s output holding register to be written into the buffer.

Rate Error Possible: See section 8.3.2

8.5 Basic Sequencer Operations

The Format and Buffer Control sequencers in the RDC are capable of eleven basic operations. The Sequence Starting Address PROM (g11) determines which operations will be executed based on the current record number and on the command in the IntOp register. Each operation transfers data between the disk and particular buffer locations, causes wakeups at defined times, and leaves the device buffer pointer at a fixed location when the operation is completed. In what follows, the number of an operation is the value of

SeqAdr[0:3]; recall that if SeqAdr.0 = 1, the Buffer Control Sequencer does not participate in the operation, but instead executes routine 0 (seek). The descriptions assume that the DevBufAdr register (the buffer pointer) is cleared before a sector starts. The eleven routines are summarized here. The contents of the buffer control and format sequencer proms are shown in Appendix D:

0: Seek: This operation transfers no data and requests no wakeups (sector wakeups will be disabled as long as this operation is in the DevOp register).

1: Write Header: This operation writes two words from buffer locations 0 and 1 onto the disk. It then increments the buffer pointer by 2, leaving it at location 4. No wakeups are generated by this operation.

2: Read/Verify Header: This operation first transfers two words from buffer locations 0 and 1 to the output holding register (this is the data to be compared with disk data). The microcode is assumed to have loaded the buffer before the operation started. It then takes two words from the input holding register (the data read from the disk), and places them into buffer locations 2 and 3. The operation leaves the buffer pointer at location 4. When the operation is complete, a wakeup is requested.

3: Write Label: This operation transfers data from buffer locations 4 through 11d to the disk. It then increments the buffer pointer four more times, leaving it at location 16d. No wakeups are requested by this operation.

4: Read/Verify Label: This operation first transfers two words from buffer locations 4 and 5 to the output holding register. At the end of this section, the device shift register will contain the first word to be verified, and the output holding register will contain the second word. At the next six XferDataS times, a word will be transferred from the buffer to the output holding register, a word will be transferred from the input holding register to the same location in the buffer, and the buffer pointer will be incremented. At the end of this section, the first six label words will be in locations 6 through 11d, and a wakeup will be requested. At the next two XferDataS times, a word will be transferred into the buffer, getting the last two label words. Finally, the pointer will be incremented by two, and a wakeup will be requested. The label read from the disk will be in buffer locations 6-13d, and the buffer pointer will be at 16d.

5: Write Data: This operation transfers the first data word to the disk from buffer location 16d, then requests a wakeup. It then transfers sixteen words to the disk with RateErrorPossible = 0. Then,, it repeats a loop in which it requests a wakeup, then transfers 16 words. During this loop, RateErrorPossible = 1. All transfers from the buffer to the disk are done in response to the XferDataS bit from the Format Sequencer/Timing Generator.

6: Read Data: First, this operation increments the buffer pointer (to 17d). It then transfers 17d words from the disk to the buffer, and requests a wakeup. From this point until the end of the record, it loops transferring 16d words into the buffer and requesting a wakeup.

7: Verify Data: The sequence of events during this operation is identical to that for Write Data.

10b: Nop Header: This operation does no data transfers and requests no wakeups. Only the Format Sequencer participates - it is used only to time the duration of the record.

11b: Nop Label: Same as Nop Header (except longer record).

12b: Nop Data: This operation times the length of the Data Record and generates a sector wakeup at the end of the field. It has no other function.

17b: Recovery Gap: This is a Format Sequencer operation used to wait for the next physical sector pulse.

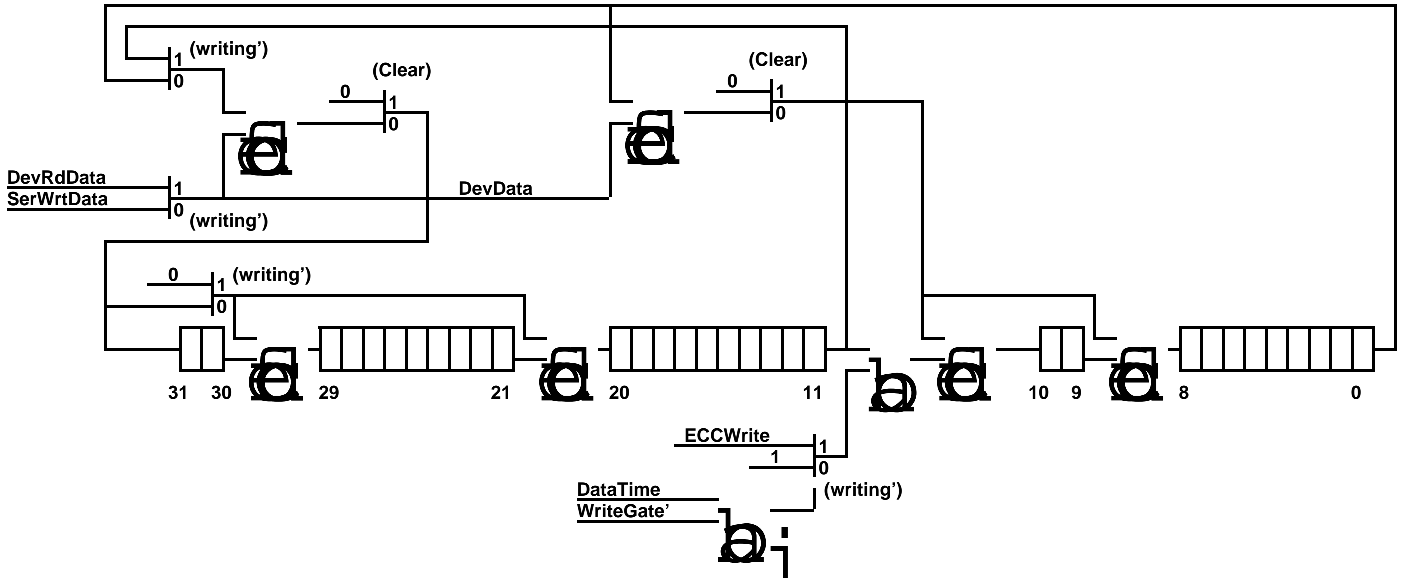


Figure 8.6a: ECC Logic

$x^{32} + x^{23} + x^{21} + x^{11} + x^2 + 1$

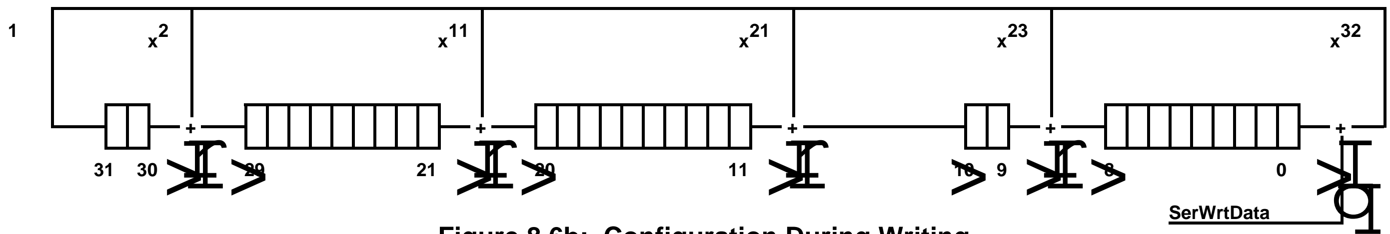


Figure 8.6b: Configuration During Writing

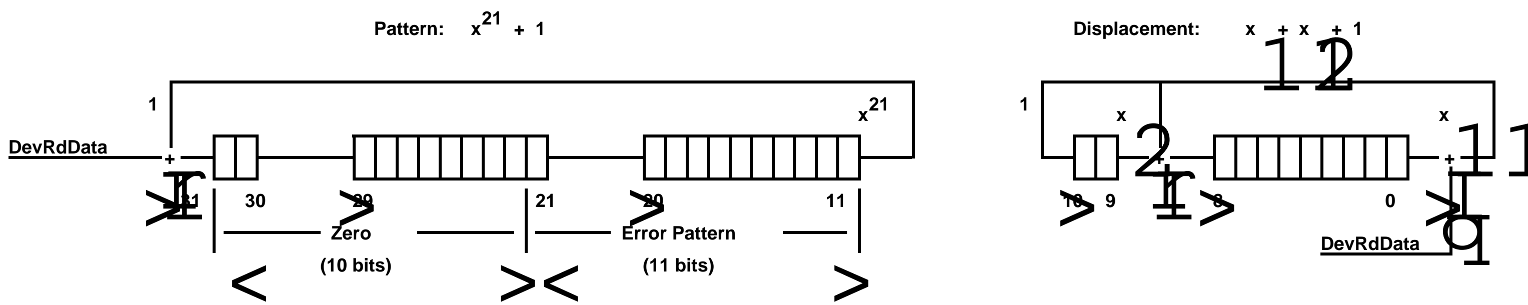


Figure 8.6c: Configuration During Reading

8.6 Error Correction

The header and label fields of a sector are error-checked using a F9401 CRC generator that writes a 16-bit CRC following the record. The CRC is not available to the disk microprogram - the only indication of an error is the RdErr bit of the status register.

Error correction of the data on the SA4000 is done by appending a number of *check bits* to the data record. The correction must be done by a program, given a *syndrome* generated by logic in the controller. The code used is a binary cyclic code known as a *Fire code*. A cyclic code is characterized by its *generator polynomial* g(x); for the Fire codes, g(x) has the form:

$$g(x) = P(x)\,(x^c-1),$$

where P(x) is an irreducible polynomial of degree m and order e, and c is not divisible by e. (x^c-1) is referred to as the *pattern polynomial,* and P(x) is the *displacement polynomial*. The length n of the code is LCM(e,c), the number of check bits is (m+c), and the number of information bits is k= n-m-c. Such a code can correct a burst error of up to m bits.

The particular code used in the RDC has:

$$g(x) = (x^{11} + x^2 + 1)\,(x^{21} + 1).$$

The order of P(x) is 2047, m=11, and c=21. The number of check bits is therefore 32, and the length of the code is n=42987 bits. The maximum number of information bits is 42955, or 2684 words.

The choice of generator polynomial and the associated hardware design are originally due to R. Bates, and are employed in the Alto Trident disk controller [see Bates, R. "TRIDENT disk for the ALTO" ,PARC CSL memo, 11 August 1976]. The theoretical background for Fire (and other) codes is developed in Peterson, W., "Error Correcting Codes", MIT Press, 1961. The correction procedure suggested here is a variation of that suggested by Peterson in section 10.6.

The hardware used to mechanize the code is shown in Figure 8.6a. Figure 8.6b shows the configuration provided when data is being written on the disk, and Figure 8.6c shows the configuration during reading. During writing, the data is premultiplied by x^{32} (to make room for the 32 check bits at the end of the record), and divided by g(x). After the data portion of the record has been written, the shift register holds the remainder from the division of M(x) by g(x), where M(x) is the polynomial representation of the data record. At this time, the feedback paths are disabled and the remainder (the check bits) is written on the disk.

During reading, the shift register is reconfigured as shown in Figure 8.6c, and two independent divisions by P(x) and (x^c-1) are performed. When the data record has been processed, the shift registers are reconfigured with the feedback paths disabled, and the resulting syndrome is shifted into the data shift register, then sent to the RDC's data buffer. The syndrome is placed in words 23b (syndrome bits 0-15), and 24b (bits 16-31), where they

can be accessed by the microdcode if an error occurs.

If no error occurs during reading, both the pattern register and the displacement register will contain zero. If a correctable error occurs, *both* these registers will be nonzero. The correction procedure must be done by a program using the syndrome recovered from the hardware. For clarity in what follows, it will be described as if it were done in the displacement and pattern registers.

The first part of the correction procedure isolates the burst pattern. The contents of the pattern register are cycled until the least significant ten bits are zero. If this does not occur within one complete cycle (21 shifts), an uncorrectable error has occured. Let the number of shifts required to isolate the error pattern be S_0. The error pattern (which will eventually be XORed with the data record is now in the most significant eleven bits of the pattern register.

To find the position of the error burst, the displacement register is clocked with the input equal to zero until its contents are equal to the most significant eleven bits of the pattern register. This will occur within 2047 shifts; let the number of shifts required to satisfy the condition be S_1. [Note: The implementation chosen premultiplies the displacement polynomial by x^{11}, which is equivalent to providing eleven shifts of the pattern register in advance. When S_1 is determined, it must be corrected by doing $S_1 _ (S_1 + 11) \bmod 2047$.]

The location *d* of the error burst relative to the *beginning* of the record is now determined [note that the "beginning" of the record is the beginning of the longest message the code is capable of correcting, 42987 bits. We are using a shortened form of the code, with zeros at the beginning of the messages. This does not change anything]. We know that:

$$S_0 = d \bmod c \quad (c = 21), \text{ and}$$
$$S_1 = d \bmod e \quad (e = 2047).$$

Since c and e are relatively prime, they can be used as the moduli m_i of a modular number system, and the Chinese Remainder Theorem [see Knuth, D., "The Art of Computer Programming", vol. 2, p249] guarantees that d is unique. To calculate d, we use:

$$d = [\ S_0\ (A_0 {*} M/m_0) + S_1\ (A_1 {*} M/m_1)\ ] \bmod M,$$

A_0 and A_1 are constants such that $A_i {*} M/m_i = 1 \bmod m_i$, and M is $m_0 {*} m_1$. R. Bates has supplied these constants; they are $A_0 = 19$, $A_1 = 195$, so:

$$d = [\ S_0\ (19 {*} 2047) + S_1\ (195 {*} 21)\ ] \bmod 42987$$

d is the displacement from the beginning of the record. To determine the location relative to the end of the record, use 42987 - d. Once the displacement is determined in this way, the burst pattern determined earlier is XORed with the appropriate bits of the record.

APPENDIX A

Time of Day Clock

```
;This is the program for the Time Of Day Clock in the MSI - D0.
;It Runs on a TI TMS1000C microprocessor.

;The programs outer loop runs once per second.
;Each time through the loop, it generates a one instruction time
; long pulse on OneSecondPulseR (for testing).
; It increments a 32 bit counter TimeOfDay once per outer loop time.

;The microprocessor has one of its K inputs tied to the 5 volt supply of the
; D0. If the power to the D0 is on (PowerOnK), the program increments
; PowerOnTime in the same manner as Time Of Day (32 bit count of the
; number of seconds).

;The program compares the TimeOfDay to another variable, AlarmClockTime
; If TimeOfDay = AlarmClockTime, the program will
; set the TurnProcessorOn R output (which will generate a 1 second pulse).
;It will also set the AlarmClockLatch R output, which will stay on until
; explicitly cleared.

;The current TimeOfDay is broadcast bit serially on the OutputData R output of
; the TMS1000.  The data is broadcast once a second and consists of one start
; bit (a 1), 56 data bits (of which only the last 32 are significant), and
; 943 stop bits (0s).
; A bit time is one millisecond. Since the message is 1000 bits long,
; at one ms per bit, this is one second per transmission.
; The time is broadcast msb first.

;During the time the program is broadcasting Time of Day, it also
; accumulates a message, bit serially, on the Input Data K input.
; The input data is sampled at the end of a bit time (ie the program sets
; the R output, waits 1 millisecond, then samples the input data).
; The input message is 56 bits and consists of a 16 bit password,
; an 8 bit command field and a 32 bit Data field.  After all 56 bits are
; accumulated, the program checks the password field.
; If it contains A1F5 (hex) [Sesame], the message is considered valid
; and the command field is acted upon.
; An invalid message is ignored, and the program reverts to sending Time of Day,
; no matter what it had previously been directed to transmit.
; No messsages are gathered or sent unless the D0 has power.

;The command field of a message is interpreted as follows
; Bit 0      TimeOfDay _ Data
; Bit 1      PowerOnHours _ Data
; Bit 2      AlarmClockTime _ Data
; Bit 3      Fudge _ Data (low 16 bits)            See below for explanation of Fudge
; Bit 4      Clear AlarmClockLatch
; Bit 5      Transmit PowerOnTime instead of TimeOfDay
; Bit 6      Transmit AlarmClockTime instead of TimeOfDay
; Bit 7      Clear PowerOnLatch (obsolete function)

;Bit 5 and 6 of the command field alter the data that is broadcast.  They are
; used to read out PowerOnTime and AlarmClockTime.  Note that bits 0:15
; of the 56 bit output message are the value of Fudge when bit 5 of
; the command field is set (ie you get both fudge in 8:23 and PowerOnTime
; in 24:55).
; The effect of bits 5 and 6 is continued until reset by another command
; (with 0s in 5 and 6)

;The program is written so that all branches result in exactly equal number
; of instruction executed.  The time base is thus the TMS1000's clock,
; which is crystal controlled at 1 MHz.  Because the crystal may not have
; enough calibration accuracy, the program maintains a variable (Fudge)
; which is used to adjust the time base.  Fudge is counted down to 0
; at the end of the main loop. The nominal value of Fudge (a 16 bit number)
; is 8000 (hex).  By changing the value of Fudge, the D0 can adjust the
; time base of the microprocessor. Fudge's accuracy is 2 instructions
; (ie setting Fudge to 13 will cause 26 more instructions to execute).
; The TMS1000C takes 6 cycles per instruction, so each Fudge count
; at 1 Mhz is 12 usec.

;To safeguard against invalid data due to loss of power or other causes,
; the program checks a 16 bit variable MasterLock against a constant
```

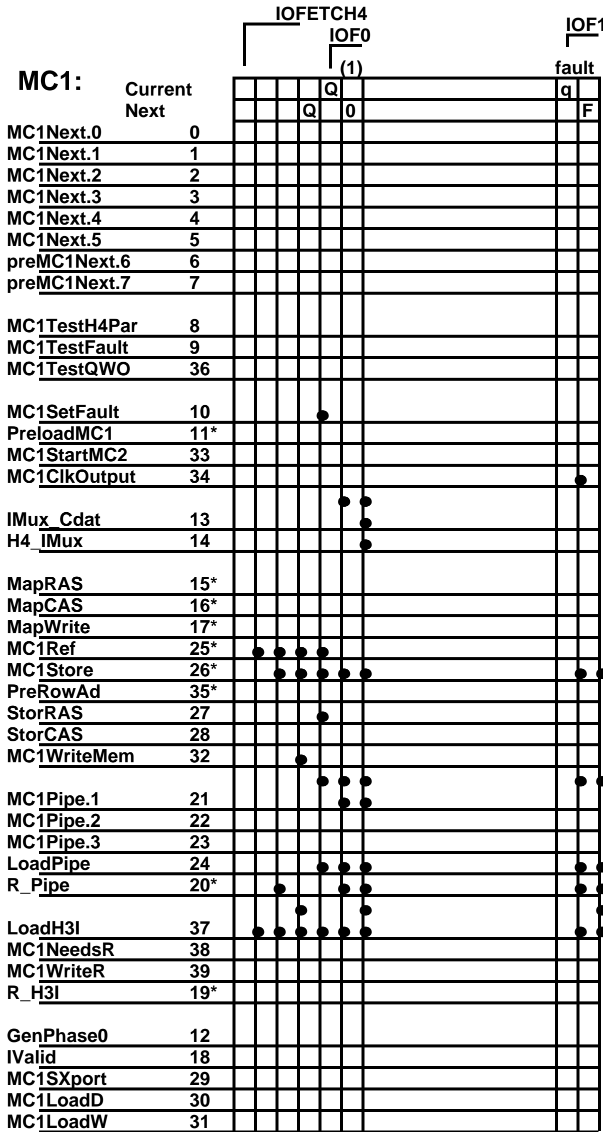
```
; (A1F5, the same Sesame as the input password).  If MasterLock is not Sesame,
; The program will zero out TimeOfDay.  MasterLock is set to Sesame upon
; receipt of a valid command.  The intention is that when the microprocessor
; has a hiccup or power failure, the registers will be scrambled.  The program
; detects this and causes time to be frozen at 00001 until a valid
; command is received

;The algorithm, in pseudo code is:
;forever do
;  begin
;    TimeOfDay _ TimeOfDay + 1
;    R.TurnProcessorOn _ if AlarmClockTime = TimeOfDay then 1 else 0
;    if K.PowerOn then
;      begin
;        PowerOnTime _ PowerOnTime + 1
;        R.OutputOutputData _ 0  !Start Bit
;        BitDelay()
;        for NibNum = 13 to 0 by -1 do
;          begin
;            OutputNibble _ case ReadOut of
;              case 2: PowerOnTime[NibNum]
;              case 4: AlarmClockTime[NibNum]
;              default: TimeOfDay[NibNum]
;            for BitNum = 3 to 0 by -1 do
;              begin
;                R.OutputData _ OutputNibble AND 8 !send msb
;                OutputNibble _ OutputNibble lshift 1
;                BitDelay()
;                InputNibble _ (InputNibble lshift 1) + K.InputData
;              end  !BitLoop
;            Input[NibNum] _ InputNibble
;          end !NibLoop
;        R.OutputData _ 1   !Stop Bits
;        if Input.Password = Sesame then
;          begin
;            if Input.Command AND 128 then TimeOfDay _ Input.Data
;            if Input.Command AND 064 then PowerOnTime _ Input.Data
;            if Input.Command AND 032 then Fudge _ Input.Data & 177777b
;            ReadOut _ Input.Command & 17b
;            MasterLock _ Sesame
;          end !ValidMessage
;        if MasterLock # Sesame then TimeOfDay _ 0
;      end  !Power On
;    Delay(XXX)   !Round out time to an even second
;    Delay(Fudge)
;  end

;last modified May 15, 1979  2:53 PM by CPT
;This program is assembled using a variant version of BCA (available from CT)
;The output file from BCA ("tod.mb") is then run through FIXTODMB.RUN, which produces
;"todprom.mb" and "todprom.list". Todprom.mb is then put into a 2708 EPROM.
```

APPENDIX B

MC1 and MC2 Microcode

(These are descriptions of MC1 revision E, MC2 revision D)

Operation: IO FETCH 4

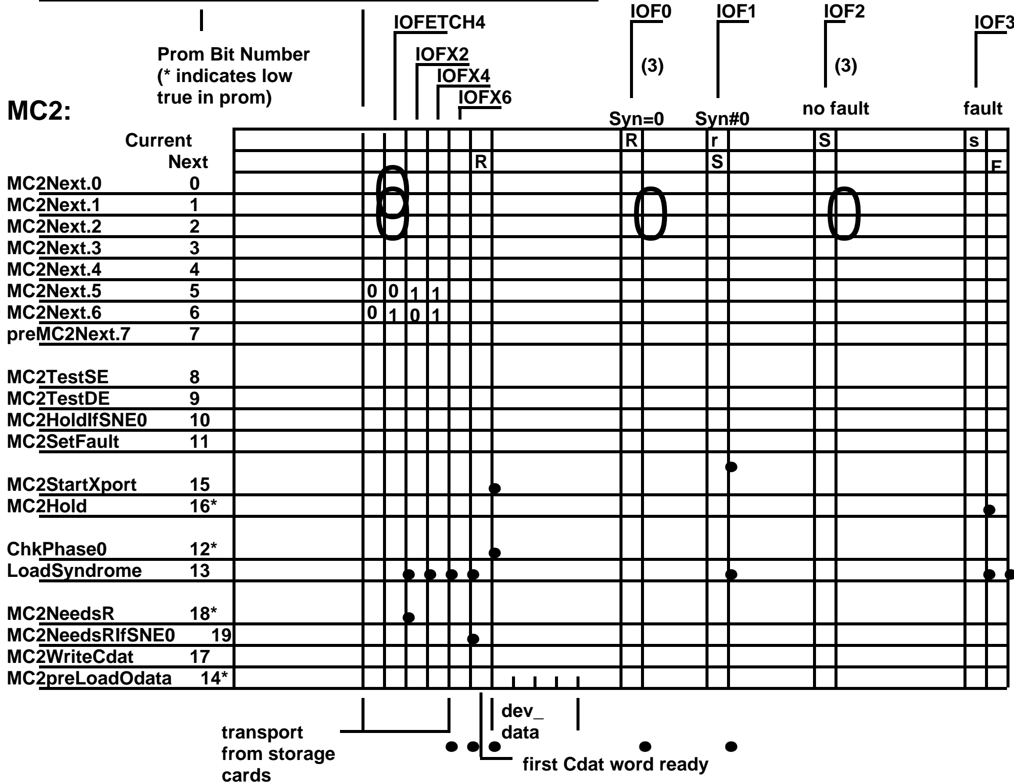


(1) If there is a request pending, MC1 will go directly to the first instruction of the next sequence

(2) MC2 may still be transferring data when another MC2 operation starts.

(3) If MC1 is waiting to start MC2, MC2 will go directly to the first instruction of the new sequence.

(4) If a fault occurs, control goes to location f (TWO) which waits 5 cycles to ensure that the processor takes the fault.



Word address to storage cards

generates OFault if a double bit error is present

9/24/78
memop01.sil

Operation: READ PIPE

RP0 RP2 RP1 RP3

R access conflicts

MC1:

Current / Next (T U T U 0 t u T U)	Prom Bit Number
MC1Next.0	0
MC1Next.1	1
MC1Next.2	2
MC1Next.3	3
MC1Next.4	4
MC1Next.5	5
preMC1Next.6	6
preMC1Next.7	7
MC1TestH4Par	8
MC1TestFault	9
MC1TestQWO	36
MC1SetFault	10
PreloadMC1	11*
MC1StartMC2	33
MC1ClkOutput	34
IMux_Cdat	13
H4_IMux	14
MapRAS	15*
MapCAS	16*
MapWrite	17*
MC1Ref	25*
MC1Store	26*
PreRowAd	35*
StorRAS	27
StorCAS	28
MC1WriteMem	32
MC1Pipe.1	21
MC1Pipe.2	22
MC1Pipe.3	23
LoadPipe	24
R_Pipe	20*
LoadH3I	37
MC1NeedsR	38
MC1WriteR	39
R_H3I	19*
GenPhase0	12
IValid	18
MC1SXport	29
MC1LoadD	30
MC1LoadW	31

Prom Bit Number
(* indicates low
true in prom)

9/24/78

memop02.sil

Operation: REFRESH

MC1:

Signal	Current Next
MC1Next.0	0
MC1Next.1	1
MC1Next.2	2
MC1Next.3	3
MC1Next.4	4
MC1Next.5	5
preMC1Next.6	6
preMC1Next.7	7
MC1TestH4Par	8
MC1TestFault	9
MC1TestQWO	36
MC1SetFault	10
PreloadMC1	11*
MC1StartMC2	33
MC1ClkOutput	34
IMux_Cdat	13
H4_IMux	14
MapRAS	15*
MapCAS	16*
MapWrite	17*
MC1Ref	25*
MC1Store	26*
PreRowAd	35*
StorRAS	27
StorCAS	28
MC1WriteMem	32
MC1Pipe.1	21
MC1Pipe.2	22
MC1Pipe.3	23
LoadPipe	24
R_Pipe	20*
LoadH3I	37
MC1NeedsR	38
MC1WriteR	39
R_H3I	19*
GenPhase0	12
IValid	18
MC1SXport	29
MC1LoadD	30
MC1LoadW	31

Main Column address on StorA0-6

Prom Bit Number
(* indicates low
true in prom)

5/4/78

memop03.sil

PF0 PF2 PF1 PF3

abort fault

Operation: Pfetch 1,2,4

MC1:

Current: P Q p q
Next: P Q 0 0 F (F = TWO)

Signal	Prom Bit
MC1Next.0	0
MC1Next.1	1
MC1Next.2	2
MC1Next.3	3
MC1Next.4	4
MC1Next.5	5
preMC1Next.6	6
preMC1Next.7	7
MC1TestH4Par	8
MC1TestFault	9
MC1TestQWO	36
MC1SetFault	10
PreloadMC1	11*
MC1StartMC2	33
MC1ClkOutput	34
IMux_Cdat	13
H4_IMux	14
MapRAS	15*
MapCAS	16*
MapWrite	17*
MC1Ref	25*
MC1Store	26*
PreRowAd	35*
StorRAS	27
StorCAS	28
MC1WriteMem	32
MC1Pipe.1	21
MC1Pipe.2	22
MC1Pipe.3	23
LoadPipe	24
R_Pipe	20*
LoadH3I	37
MC1NeedsR	38
MC1WriteR	39
R_H3I	19*
GenPhase0	12
IValid	18
MC1SXport	29
MC1LoadD	30
MC1LoadW	31

MC1Next.4 row: 0 0 0 0 1 0 0 0 0 0

Note 1: MC2WriteCdat has two functions. Here, it is selecting H3C or H3U as the source of R bus data. During OUTPUT, OValid is generated in the cycle following (MC2Next.4 and MC2WriteCdat). For this reason, MC2Next.4 must be zero when MC2WriteCdat is asserted to avoid generating OValid.

Prom Bit Number
(* indicates low true in prom)

PFETCH1,2,4 PFX2 PFX4 PFX6 PF0 PF1 PF2 PF2A PF2B PF3 PF3A PF3B

Syn=0 Syn#0 no fault fault

MC2:

Current: R r S s
Next: R S F (F = FAULTWAIT)

Signal	Prom Bit
MC2Next.0	0
MC2Next.1	1
MC2Next.2	2
MC2Next.3	3
MC2Next.4	4 — (note 1)
MC2Next.5	5
MC2Next.6	6
preMC2Next.7	7
MC2TestSE	8
MC2TestDE	9
MC2HoldIfSNE0	10
MC2SetFault	11
MC2StartXport	15
MC2Hold	16*
ChkPhase0	12*
LoadSyndrome	13
MC2NeedsR	18*
MC2NeedsRIfSNE0	19
MC2WriteCdat	17 — (note 1)
MC2preLoadOdata	14*

MC2Next.5 row: 0 0 1 1
MC2Next.6 row: 0 1 0 1

Transport from Storage Cards

MC2Ad is reloaded by MC2HoldIfSNE0
(MC2Ad is on the ALU card)

9/24/78

memop04-05-06.sil

Operation: INPUT

MC1:

MC2 R conflict

Signal	Prom Bit						
Current				R			r
Next			R		0		R
MC1Next.0	0						
MC1Next.1	1						
MC1Next.2	2						
MC1Next.3	3						
MC1Next.4	4	0 1	1	1	0		1
MC1Next.5	5						
preMC1Next.6	6						
preMC1Next.7	7						
MC1TestH4Par	8						
MC1TestFault	9						
MC1TestQWO	36						
MC1SetFault	10						
PreloadMC1	11*						
MC1StartMC2	33						
MC1ClkOutput	34						
IMux_Cdat	13						
H4_IMux	14						
MapRAS	15*						
MapCAS	16*						
MapWrite	17*						
MC1Ref	25*						
MC1Store	26*						
PreRowAd	35*						
StorRAS	27						
StorCAS	28						
MC1WriteMem	32						
MC1Pipe.1	21						
MC1Pipe.2	22						
MC1Pipe.3	23						
LoadPipe	24						
R_Pipe	20*						
LoadH3I	37						
MC1NeedsR	38						
MC1WriteR	39						
R_H3I	19*						
GenPhase0	12						
IValid	18						
MC1SXport	29						
MC1LoadD	30						
MC1LoadW	31						

— Used here to clock H4ParityError flip-flop (MC1SXport)

Prom Bit Number (* indicates low true in prom)

Goes to location 2 if H4 Parity Error (see IO Store 4)

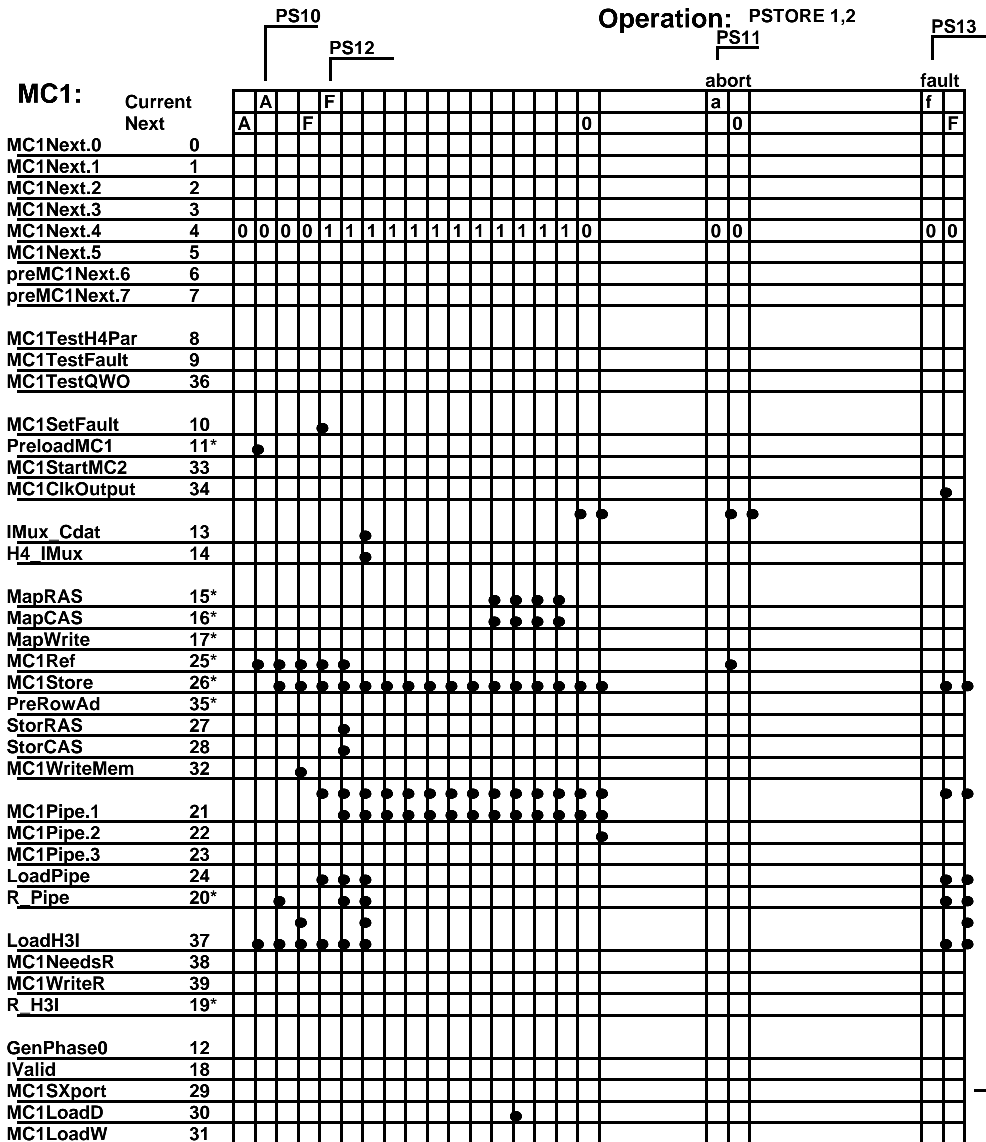
H4 loaded

Note: Pipe gets Type/task only. other words are garbage

9/24/78

memop07.sil

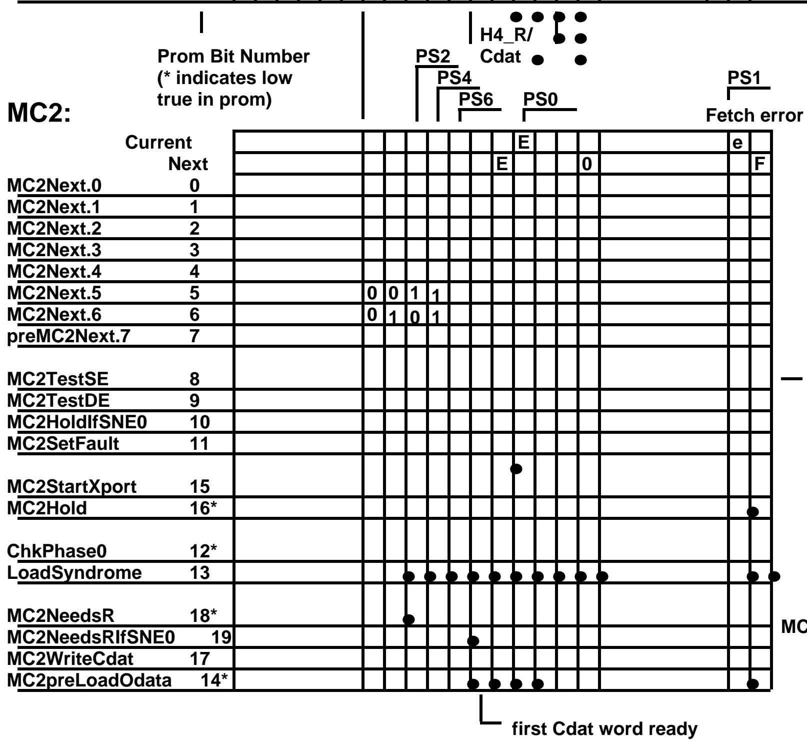
Operation: PSTORE 1,2



— Cannot cause H4 ParityError, since EnInputParChk is false during PSTORE1,2

Prom Bit Number
(* indicates low true in prom)

H4_R/ Cdat



— Single errors on the read are not checked or logged, since the data will be overwritten immediately.

MC2 does R access. Write is inhibited on the ALU card

first Cdat word ready

9/24/78

Memop08-09.sil

Operation: PSTORE 4

MC1:

Signal	Prom Bit	1	2	3	4	5	6	7	8	9	10	11	12	13
				PS42			PS410				PS43		PS411 / Fault	
Current				U			F				u		f	
Next			U			F			0		U			— FAULTWAIT
MC1Next.0	0													
MC1Next.1	1													
MC1Next.2	2													
MC1Next.3	3													
MC1Next.4	4	0	0	0	0	0	1	1	0		0		0	0
MC1Next.5	5													
preMC1Next.6	6													
preMC1Next.7	7													
MC1TestH4Par	8													
MC1TestFault	9													
MC1TestQWO	36													
MC1SetFault	10					●								
PreloadMC1	11*													
MC1StartMC2	33													
MC1ClkOutput	34												●	
								●	●					
IMux_Cdat	13													
H4_IMux	14													
MapRAS	15*													
MapCAS	16*													
MapWrite	17*													
MC1Ref	25*	●	●	●	●	●	●				●			
MC1Store	26*		●	●	●	●	●	●	●		●		●	
PreRowAd	35*													
StorRAS	27						●							
StorCAS	28						●							
MC1WriteMem	32			●										
					●	●	●	●	●				●	
MC1Pipe.1	21					●	●	●	●				●	
MC1Pipe.2	22								●					
MC1Pipe.3	23													
LoadPipe	24				●	●	●						●	
R_Pipe	20*		●			●	●						●	
				●			●						●	
LoadH3I	37	●	●	●	●	●	●						●	
MC1NeedsR	38													
MC1WriteR	39													
R_H3I	19*													
			●	●	●	●					●			
GenPhase0	12													
IValid	18													
MC1SXport	29													
MC1LoadD	30				●									
MC1LoadW	31													

Read R

Prom Bit Number
(* indicates low
true in prom)

9/24/78

memop10.sil

Operation: Output

MC1:

Signal	Prom Bit		
Current			
Next			0
MC1Next.0	0		
MC1Next.1	1		
MC1Next.2	2		
MC1Next.3	3		
MC1Next.4	4	0	0
MC1Next.5	5		
preMC1Next.6	6		
preMC1Next.7	7		
MC1TestH4Par	8		
MC1TestFault	9		
MC1TestQWO	36		
MC1SetFault	10		
PreloadMC1	11*		
MC1StartMC2	33		
MC1ClkOutput	34		
IMux_Cdat	13		
H4_IMux	14		
MapRAS	15*		
MapCAS	16*		
MapWrite	17*		
MC1Ref	25*		
MC1Store	26*		
PreRowAd	35*		
StorRAS	27		
StorCAS	28		
MC1WriteMem	32		
MC1Pipe.1	21		
MC1Pipe.2	22		
MC1Pipe.3	23		
LoadPipe	24		
R_Pipe	20*		
LoadH3I	37		
MC1NeedsR	38		
MC1WriteR	39		
R_H3I	19*		
GenPhase0	12		
IValid	18		
MC1SXport	29		
MC1LoadD	30		
MC1LoadW	31		

Prom Bit Number
(* indicates low true in prom)

OUTPUT

OUTX

MC2:

Signal	Prom Bit								
Current			0						
Next									0
MC2Next.0	0								
MC2Next.1	1								
MC2Next.2	2								
MC2Next.3	3								
MC2Next.4	4							1	
MC2Next.5	5								
MC2Next.6	6								
preMC2Next.7	7								
MC2TestSE	8								
MC2TestDE	9								
MC2HoldIfSNE0	10								
MC2SetFault	11								
MC2StartXport	15								
MC2Hold	16*								
ChkPhase0	12*								
LoadSyndrome	13								
MC2NeedsR	18*								
MC2NeedsRIfSNE0	19								
MC2WriteCdat	17								
MC2preLoadOdata	14*								

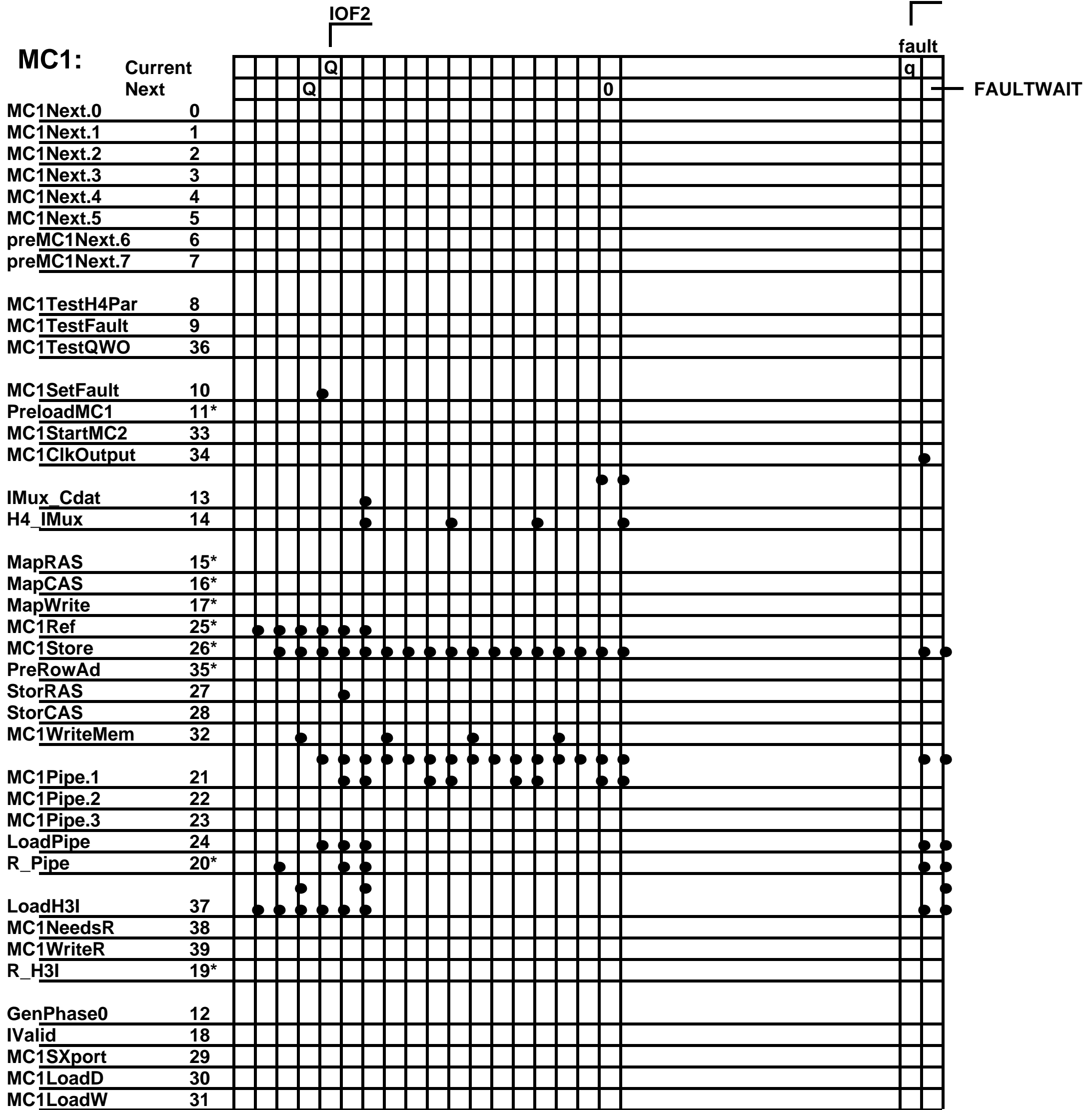
Note: MC2Next.4 and MC2WriteCDat => OValid in the Next cycle

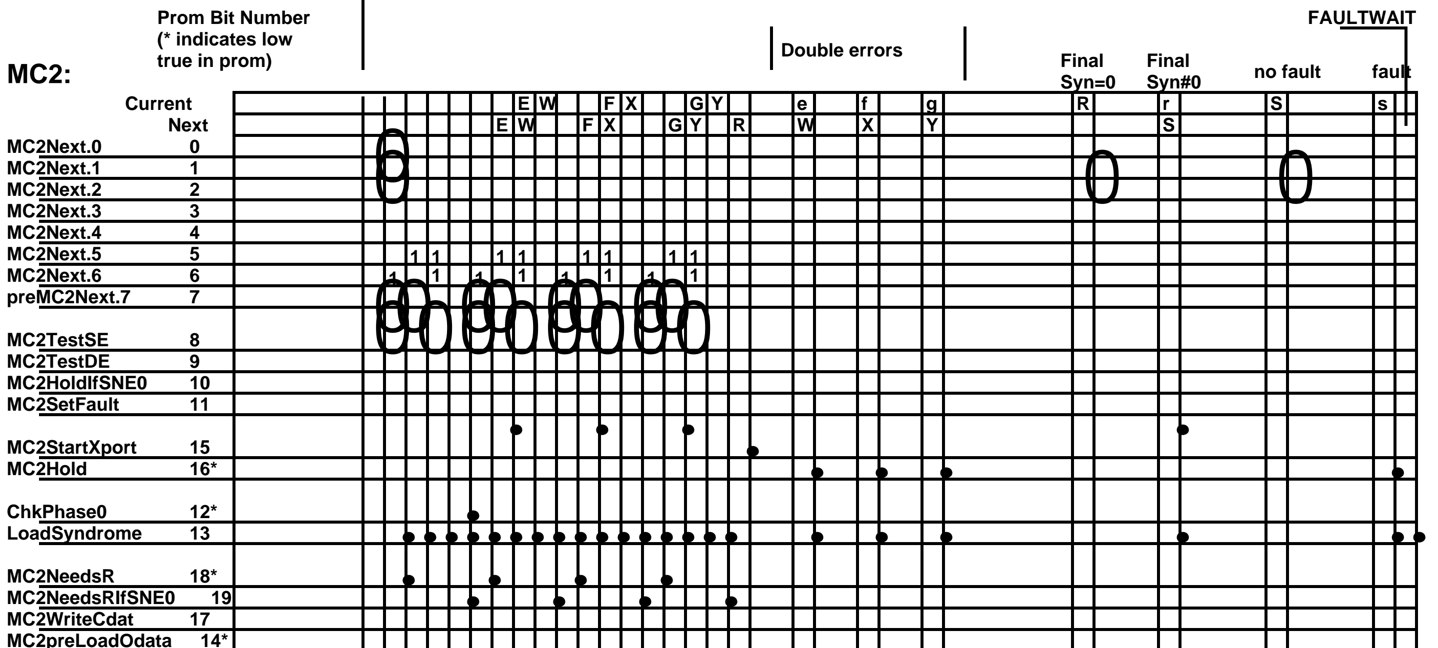
OValid=1

9/24/78

Memop11.sil

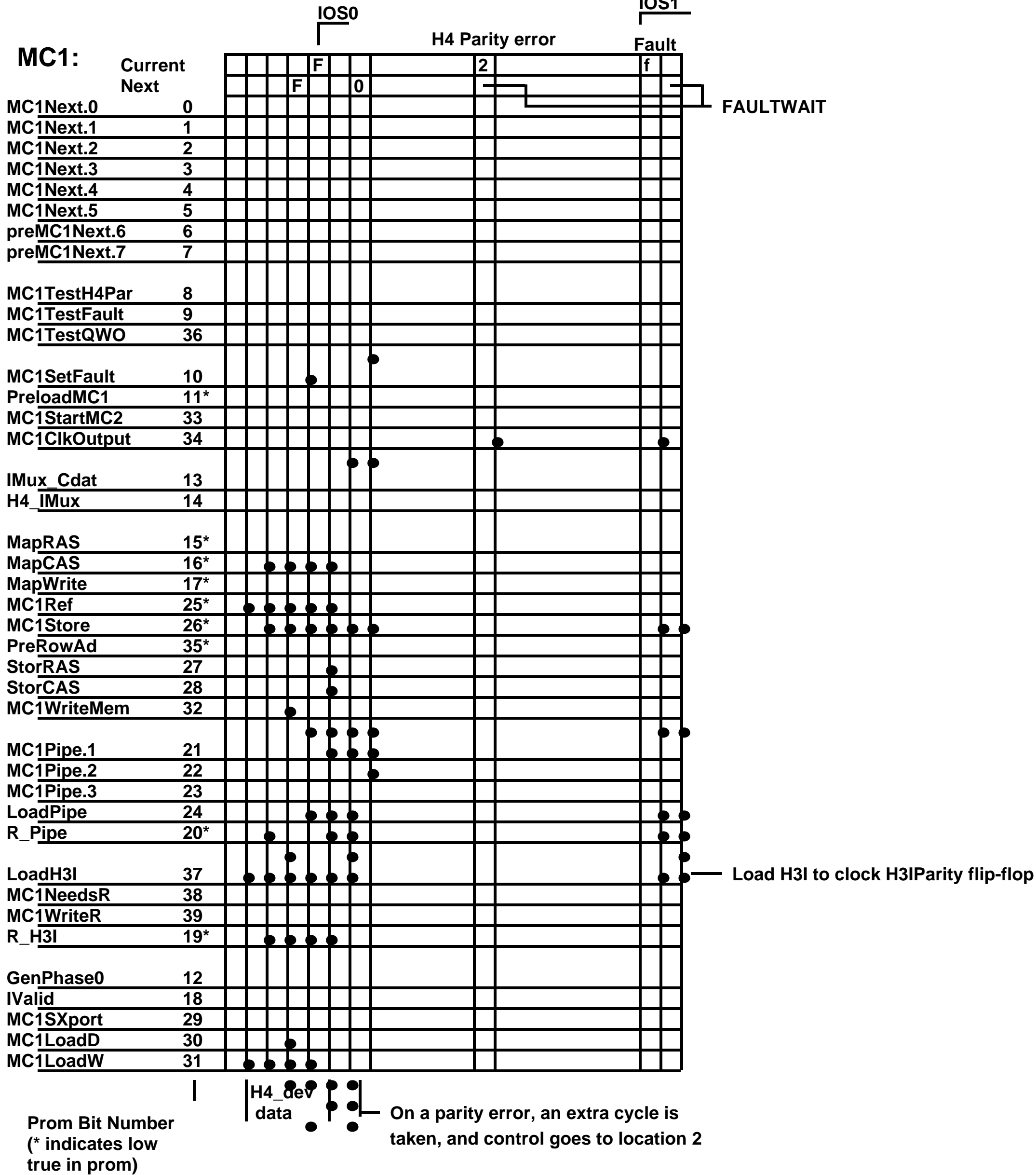
Operation: IO FETCH 16





9/24/78
memop12.sil

first Cdat word ready



5/4/78

memop13.sil

Operation: XMAP

XM0 XM1

MC1:

Signal	Prom Bit Number
Current	
Next	
MC1Next.0	0
MC1Next.1	1
MC1Next.2	2
MC1Next.3	3
MC1Next.4	4
MC1Next.5	5
preMC1Next.6	6
preMC1Next.7	7
MC1TestH4Par	8
MC1TestFault	9
MC1TestQWO	36
MC1SetFault	10
PreloadMC1	11*
MC1StartMC2	33
MC1ClkOutput	34
IMux_Cdat	13
H4_IMux	14
MapRAS	15*
MapCAS	16*
MapWrite	17*
MC1Ref	25*
MC1Store	26*
PreRowAd	35*
StorRAS	27
StorCAS	28
MC1WriteMem	32
MC1Pipe.1	21
MC1Pipe.2	22
MC1Pipe.3	23
LoadPipe	24
R_Pipe	20*
LoadH3I	37
MC1NeedsR	38
MC1WriteR	39
R_H3I	19*
GenPhase0	12
IValid	18
MC1SXport	29
MC1LoadD	30
MC1LoadW	31

Current row: R (XM0), r (XM1); Next row: R, R

Phase 1 | 2 | 3

Prom Bit Number
(* indicates low
true in prom)

The operation proceeds in three phases:

1) Load the pipe from the map entry for the selected VA.

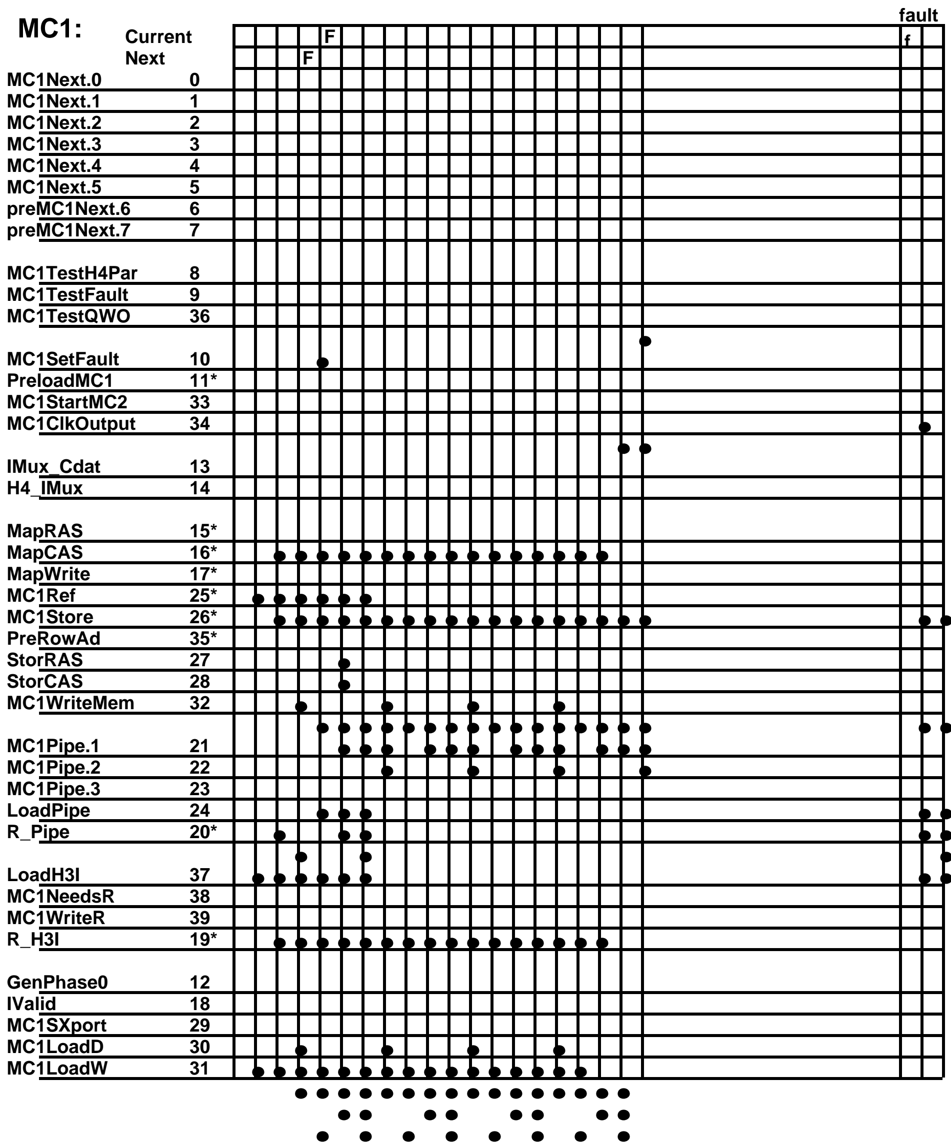
2) Read one word from R into H4, and write it into the map.

3) Dump the pipe into the next 3 R registers. Only the entries which contain map data are dumped.

9/24/78

memop14.sil

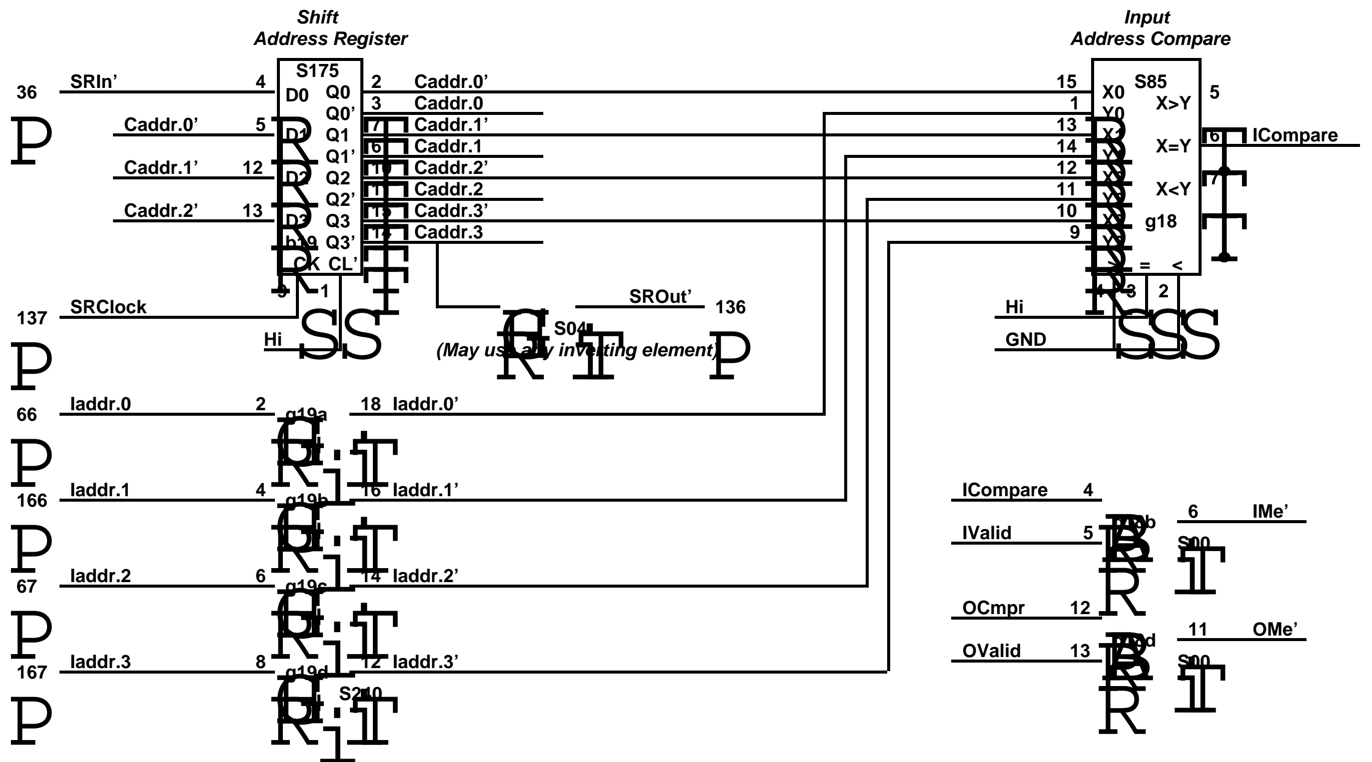
Operation: IOStore16

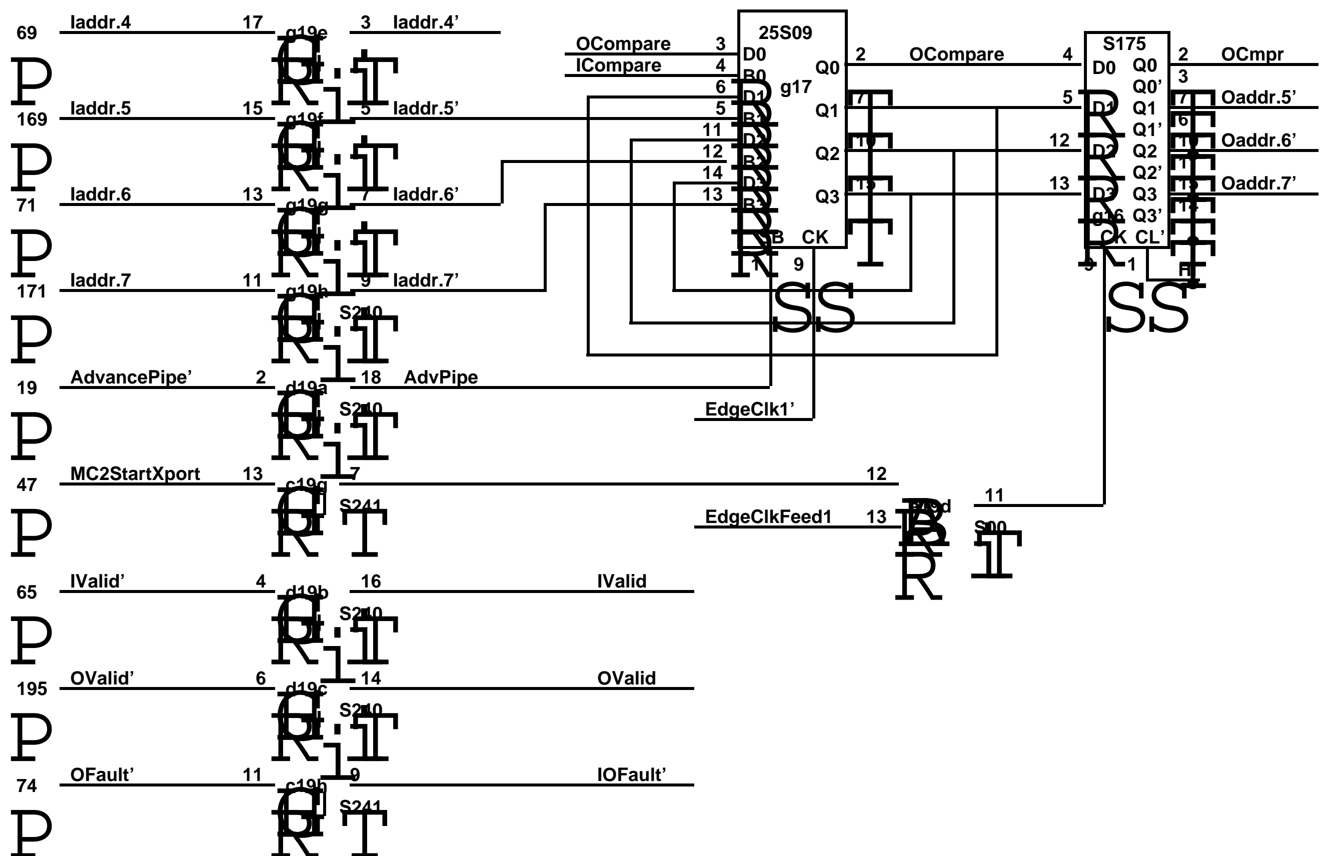


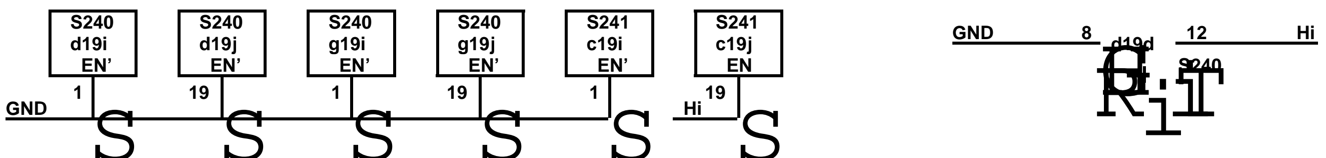
9/24/78
memop15.sil

APPENDIX C

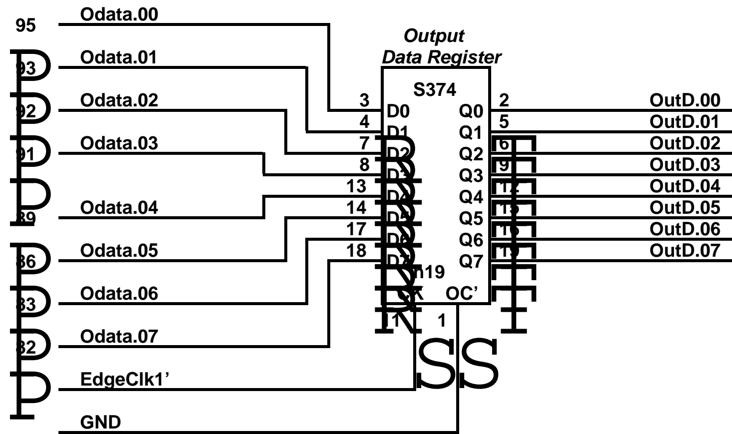
Standard I/O Interface

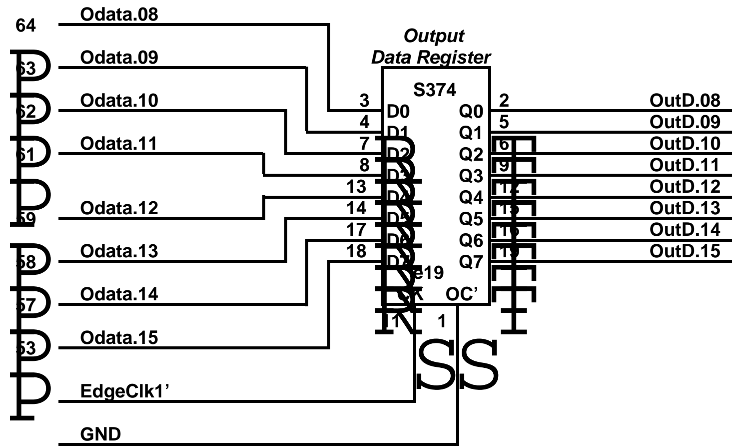


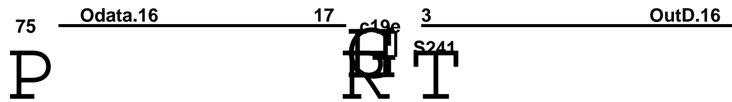


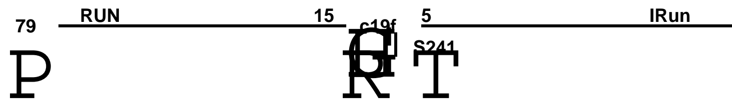


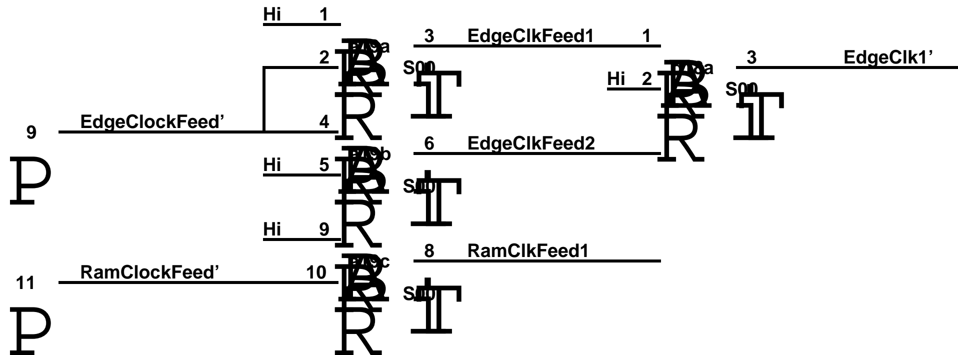
XEROX EOD	Project D0	IO INTERFACE Address	File D0IO01.sil	Designer DPPStaff	Rev S	Date 05/10/78	Page 1











XEROX EOD	Project D0	IO INTERFACE OData/Clocks	File D0IO02.sil	Designer DPPStaff	Rev B	Date 05/10/78	Page 2

G1 1
WakeP1' 3 5
G2 4
WakeP2' 6 6
G3 9
WakeP3' 8 7
Transmit 10
Transmit' 12
Phase1' 13
G123
G23
G3
Transmit
Phase1'
Phase1NextI'
MyTask'
GND
102 Phase1Next' 15
Phase1Next
WakeRequest
Wake Enable
25S09
Phase1NextI'
Phase1'
G1
G2
G3
HM7603
GND
Caddr.0'
Caddr.1'
Caddr.2'
Caddr.3'
G1P2
G1P1
G2P2
G2P1
G3P2
G3P1
G123
G23
Phase1Next
EdgeClk1'
dTx'
S175
Transmit'
Transmit
OMe'
OMeF'
OMeF
IMe'
IMeF'
IMeF
EdgeClk1'
Hi

147 IOStrobe
34 CTask.0 ICTask.0
Caddr.0'
134 CTask.1 ICTask.1
Caddr.1'
35 CTask.2 ICTask.2
Caddr.2'
135 CTask.3 ICTask.3
Caddr.3'
MyTask'
MyTask
NeedAttn
IOAttn' 172

XEROX *EOD*	*Project* D0	IO INTERFACE WakeUp/CTask	*File* D0IO03.sil	*Designer* DPPStaff	*Rev* B	*Date* 05/10/78	*Page* 3

APPENDIX D

RDC Buffer Control and Format Sequencers

The following code is for the Buffer Control Sequencer. The 1's and 0's are the address and data bits (x = don't care). The sequences have been given as programs to the right of the bits.

Data Bits Are:

```
0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC', dLdC'
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr'
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
------------------------------------------------------------------------------------------
000 0000 x = 0000 0011 0001 0000 NoDataTransfer: goto[.];

------------------------------------------------------------------------------------------
001 0000 x = 0001 0011 0001 0000 WriteHeader: goto[.+1];

001 0001 1 = 0010 1010 0001 0000    Cnt _ 1110, goto[.+1];

001 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,., XferDataS];
001 0010 0 = 0011 0111 1001 1100    Dev _ Buf, IncBufAdr, IncCtr,
goto[CntDone,CntWait,CntCry];

001 0011 1 = 0010 0011 0001 0000 CntWait: goto[WaitLoop];
001 0011 0 = 0100 0011 0001 1000 CntDone: goto[.+1], IncBufAdr;

001 0100 1 = 0101 0011 0001 1000    goto[.+1], IncBufAdr;
001 0101 1 = 0101 0011 0001 0000    goto[.];

------------------------------------------------------------------------------------------
010 0000 x = 0001 0011 0001 0000 Read/VerifyHeader: goto[.+1];

010 0001 1 = 0010 1010 0001 0000    Cnt _ 1110, goto[.+1];

;2-word loop transfers data to device holding register
010 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,., XferDataS];
010 0010 0 = 0011 0111 1001 1100    IncCtr, IncBufAdr, Dev _ Buf,
goto[CntDone,CntWait,CntCry];

010 0011 1 = 0010 0011 0001 0000 CntWait: goto[WaitLoop];
010 0011 0 = 0100 1010 0001 0000 CntDone: goto[.+1], Cnt _ 1110;

;2-word loop transfers data from device holding register to buffer
010 0100 1 = 0100 0011 0101 0000 WaitLoopA: goto[.+1,., XferDataS];
010 0100 0 = 0101 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf,
goto[CntDoneA,CntWaitA,CntCry];

010 0101 1 = 0100 0011 0001 0000 CntWaitA:goto[WaitLoopA];
010 0101 0 = 0110 0011 0001 0000 CntDoneA: goto[.+1];

;Wait for one more XferDataS, and generate a wakeup
010 0110 1 = 0110 0011 0101 0000    goto[.+1,.,XferDataS];
010 0110 0 = 0111 0011 0001 0000    goto[.+1];

010 0111 1 = 1000 0011 0011 0000    DataWake, goto[.+1];

010 1000 1 = 1000 0011 0001 0000    goto[.];
```

```
Data Bits Are:

0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC', dLdC'
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr'
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
---------------------------------------------------------------------------------------------
011 0000 x = 0001 0011 0001 0000 WriteLabel: goto[.+1];

011 0001 1 = 0010 0010 0001 0000    Cnt _ 1100, goto[.+1];

;4-word loop to transfer the first four label words to the device holding register from the
buffer
011 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,.,XferDataS];
011 0010 0 = 0011 0111 1001 1100    IncCtr, IncBufAdr, Dev _ Buf, goto[CntDone, CntWait,
CntCry];

011 0011 1 = 0010 0011 0001 0000 CntWait: goto[WaitLoop];
011 0011 0 = 0100 0010 0001 0000 CntDone: Cnt _ 1100, goto[.+1];

;4-word loop to transfer the second four label words to the device holding register from the
buffer
011 0100 1 = 0100 0011 0101 0000 WaitLoopA: goto[.+1,.,XferDataS];
011 0100 0 = 0101 0111 1001 1100    IncCtr, IncBufAdr, Dev _ Buf, goto[CntDoneA, CntWaitA,
CntCry];

011 0101 1 = 0100 0011 0001 0000 CntWaitA: goto[WaitLoopA];
011 0101 0 = 0110 0010 0001 0000 CntDoneA: Cnt _ 1100, goto[.+1];

;4 cycles are spent incrementing the device buffer address
011 0110 1 = 0110 0111 1001 1000    IncCtr, IncBufAdr, goto[.+1,.,CntCry];
011 0110 0 = 0111 0011 0001 0000    goto[.+1];

011 0111 1 = 0111 0011 0001 0000    goto[.+1];
```

```
Data Bits Are:

0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC', dLdC'
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr'
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
---------------------------------------------------------------------------------------------
100 0000 x = 0001 0011 0001 0000 Read/VerifyLabel: goto[.+1];

100 0001 1 = 0010 1010 0001 0000    Cnt _ 1110, goto[.+1];

;2-word loop transfers the first two words from the buffer to the device holding register.
100 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,.,XferDataS];
100 0010 0 = 0011 0111 1001 1100    IncCtr,IncBufAdr, Dev _ Buf, goto[CntDone, CntWait,
CntCry];

100 0011 1 = 0010 0011 0001 0000 CntWait: goto[WaitLoop];
100 0011 0 = 0100 0010 0001 0000 CntDone: Cnt _ 1100, goto[.+1];

;4-word loop sends a word to the device holding register, then reads a word from
;the device holding register into the buffer.
100 0100 1 = 0100 0011 0101 0000 WaitLoopA: goto[.+1,.,XferDataS];
100 0100 0 = 0101 0111 1001 0100    IncCtr, Dev _ Buf, goto[CntDoneA, CntWaitA, CntCry];

100 0101 1 = 0100 0011 0001 1010 CntWaitA: IncBufAdr, WrtBuf, goto[WaitLoopA];
100 0101 0 = 0110 1010 0001 1010 CntDoneA: Cnt _ 1110, IncBufAdr, WrtBuf, goto[.+1];

;2-word loop sends a word to the device holding register, then reads a word from
;the device holding register into the buffer.
100 0110 1 = 0110 0011 0101 0000 WaitLoopB: goto[.+1,.,XferDataS];
100 0110 0 = 0111 0111 1001 0100    IncCtr, Dev _ Buf, goto[CntDoneB, CntWaitB, CntCry];

100 0111 1 = 0110 0011 0001 1010 CntWaitB: IncBufAdr, WrtBuf, goto[WaitLoopB];
100 0111 0 = 1000 0011 0001 1010 CntDoneB: IncBufAdr, WrtBuf, goto[.+1];

100 1000 1 = 1001 0011 0001 0000    goto[.+1];

;Cause a wakeup
100 1001 1 = 1010 1010 0011 0000    Cnt _ 1110, DataWake, goto[.+1];

;2-word loop transfers the final two words from the device holding register to the buffer.
100 1010 1 = 1010 0011 0101 0000 WaitLoopC: goto[.+1,.,XferDataS];
100 1010 0 = 1011 0111 1001 1010  IncCtr, IncBufAdr, WrtBuf, goto[CntDoneC, CntWaitC,
CntCry];

100 1011 1 = 1010 0011 0001 0000 CntWaitC: goto[WaitLoopC];
100 1011 0 = 1100 0011 0001 0000 CntDoneC: goto[.+1];

;Wait for one more XferDataS, cause a wakeup, and increment the buffer address by 2.
100 1100 1 = 1100 0011 0101 0000    goto[.+1,.,XferDataS];
100 1100 0 = 1101 0011 0011 1000    DataWake, IncBufAdr, goto[.+1];

100 1101 1 = 1110 0011 0001 1000    IncBufAdr, goto[.+1];

100 1110 1 = 1110 0011 0001 0000    goto[.];
```

```
Data Bits Are:

0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC', dLdC'
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr'
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
---------------------------------------------------------------------------------------------
101 0000 x = 0001 0011 0001 0000 WriteData: goto[.+1];

101 0001 1 = 0010 0011 0001 0000    goto[.+1];

;Send one word to the device holding register
101 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,.,XferDataS];
101 0010 0 = 0011 0011 0001 1100  IncBufAdr, Dev _ Buf, goto[.+1];

;Wakeup, set up count
101 0011 1 = 0100 0001 0011 0000  Cnt _ 0000, DataWake, goto[.+1];

;Send 16 words with REP equal zero.
101 0100 1 = 0100 0011 0101 0000 FirstLoop: goto[.+1,.,XferDataS];
101 0100 0 = 0101 0111 1001 1100    IncCtr, IncBufAdr, Dev _ Buf, goto[CntDone, CntWait,
CntCry];

101 0101 1 = 0100 0011 0001 0000 CntWait: goto[FirstLoop];
101 0101 0 = 0110 0011 0001 0000 CntDone: goto[MainSetup];

;Cause a wakeup, send 16words (forever)  REP equals 1.
101 0110 1 = 0111 0001 0011 0001 MainSetup: Cnt _ 0000, DataWake, REP, goto[.+1];

101 0111 1 = 0111 0011 0101 0001 MainLoop: goto[.+1,.,XferDataS];
101 0111 0 = 1000 0111 1001 1101    IncCtr, IncBufAdr, Dev _ Buf, REP,  goto[CntDoneA,
CntWaitA, CntCry];

101 1000 1 = 0111 0011 0001 0001 CntWaitA: REP, goto[MainLoop];
101 1000 0 = 0110 0011 0001 0001 CntDoneA: REP, goto[MainSetup];
```

```
Data Bits Are:

0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC’, dLdC’
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr’
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
---------------------------------------------------------------------------------------------
110 0000 x = 0001 0011 0001 0000 ReadData: goto[.+1];

;Wait for XferData and increment the buffer address, but don’t write the buffer.
110 0001 1 = 0001 0011 0101 0000    goto[.+1,.,XferDataS];
110 0001 0 = 0010 0010 0001 1000    Cnt _ 1100, IncBufAdr, goto[.+1];

;4-word loop transfers data from the device holding register to the buffer - no wakeups yet.
110 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,.,XferDataS];
110 0010 0 = 0011 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf, goto[CntDone, CntWait,
CntCry];

110 0011 1 = 0010 0011 0001 0000 CntWait: goto[WaitLoop];
110 0011 0 = 0100 0010 0001 0000 CntDone: Cnt _ 1100, goto[.+1];

;4-word loop transfers data from the device holding register to the buffer - no wakeups yet.
110 0100 1 = 0100 0011 0101 0000 WaitLoopA: goto[.+1,.,XferDataS];
110 0100 0 = 0101 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf, goto[CntDoneA, CntWaitA,
CntCry];

110 0101 1 = 0100 0011 0001 0000 CntWaitA: goto[WaitLoopA];
110 0101 0 = 0110 0010 0001 0000 CntDoneA: Cnt _ 1100, goto[.+1];

;4-word loop transfers data from the device holding register to the buffer - no wakeups yet.
110 0110 1 = 0110 0011 0101 0000 WaitLoopB: goto[.+1,.,XferDataS];
110 0110 0 = 0111 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf, goto[CntDoneB, CntWaitB,
CntCry];

110 0111 1 = 0110 0011 0001 0000 CntWaitB: goto[WaitLoopB];
110 0111 0 = 1000 0010 0001 0000 CntDoneB: Cnt _ 1100, goto[.+1];

;4-word loop transfers data from the device holding register to the buffer - no wakeups yet.
110 1000 1 = 1000 0011 0101 0000 WaitLoopC: goto[.+1,.,XferDataS];
110 1000 0 = 1001 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf, goto[CntDoneC, CntWaitC,
CntCry];

110 1001 1 = 1000 0011 0001 0000 CntWaitC: goto[WaitLoopC];
110 1001 0 = 1010 0011 0001 0000 CntDoneC: goto[.+1];

;transfer one more word...
110 1010 1 = 1010 0011 0101 0000    goto[.+1,.,XferDataS];
110 1010 0 = 1011 0011 0001 1010    IncBufAdr, WrtBuf, goto[.+1];

;cause a wakeup and set up for 16 word loop
110 1011 1 = 1100 0001 0011 0000    Cnt _ 0000, DataWake, goto[.+1];

;16-word loop writes words to the buffer, then causes a wakeup (forever).
110 1100 1 = 1100 0011 0101 0000  MainLoop: goto[.+1,.,XferDataS];
110 1100 0 = 1101 0111 1001 1010    IncCtr, IncBufAdr, WrtBuf, goto[CntDoneD, CntWaitD,
CntCry];

110 1101 1 = 1100 0011 0001 0000 CntWaitD:  goto[MainLoop];
110 1101 0 = 1100 0001 0011 0000 CntDoneD: Cnt _ 0000, DataWake, goto[MainLoop];
```

```
Data Bits Are:

0-3:     BufNxtAdr[3:6]
4-7:     dBrC.2, dInC, dClC', dLdC'
8-11:    dBrC, dBrX, dDtW, dClrMemBufAdr'
12-15:   dIncDevBufAdr, dDev_Buf, dWrtBuf, dREP

Address:    Data:
-----------------------------------------------------------------------------------------
111 0000 x = 0001 0011 0001 0000 VerifyData: goto[.+1];

111 0001 1 = 0010 0011 0001 0000    goto[.+1];

;Send one word to the device holding register.
111 0010 1 = 0010 0011 0101 0000 WaitLoop: goto[.+1,.,XferDataS];
111 0010 0 = 0011 0011 0001 1100    IncBufAdr, Dev _ Buf, goto[.+1];

;Cause a wakeup, set up for 16-word loop.
111 0011 1 = 0100 0001 0011 0000    Cnt _ 0000, DataWake, goto[.+1];

;16-word loop...
111 0100 1 = 0100 0011 0101 0000 WaitLoopA: goto[.+1,.,XferDataS];
111 0100 0 = 0101 0111 1001 1100    IncCtr, IncBufAdr, Dev _ Buf, goto[CntDone, CntWait,
CntCry];

111 0101 1 = 0100 0011 0001 0000 CntWait: goto[WaitLoopA];
111 0101 0 = 0110 0011 0001 0000 CntDone: goto[MainLoopSetup];

;Cause a wakeup, set up for 16-word main loop, which transfers forever.
111 0110 1 = 0111 0001 0011 0001 MainLoopSetup: Cnt _ 0000, DataWake, goto[.+1];

111 0111 1 = 0111 0011 0101 0001 MainLoop: goto[.+1,.,XferDataS];
111 0111 0 = 1000 0111 1001 1101    IncCtr, IncBufAdr, Dev _ Buf, goto[CntDoneA, CntWaitA,
CntCry];

111 1000 1 = 0111 0011 0001 0001 CntWaitA: goto[MainLoop];
111 1000 0 = 0110 0011 0001 0001 CntDoneA: goto[MainLoopSetup];
```

```
The following listing is for the Format Sequencer.
The (decimal) location in the memory is given, followed by
the contents of the memory in the following order:

ForNxtAdr.0-6 (7 bits)
dForCnt.0-4 (5 bits)
dBrOnSync
dSequenceEnd
dSectorWake
dClrDevOp'
dXferTime
dSyncTime
dDataTime
dCRCShift
dCRCWrite
dCRCCheck
dWriteGate
dReadGate
dECCClear
dECCShift
dECCWrite
ddECCCheck
(28 bits total)

For each location, the control bits (marked with * above) are shown in text form.
The quantities in parentheses are the address of the instruction that will be executed
following the current instruction, and (if the current address is even) the number of
cycles that the sequencer will remain at that location.

The Basic Sequencer Operation is given at the start of each sequence.

Seek:
000: 0000000 xxxxx 0001 0000 0000 0000 (000)
001: 0000000 xxxxx 0001 0000 0000 0000 (000)
002: 0000001 xxxxx 0001 0000 0000 0000 (002)
003: 0000010 11111 0101 0000 0000 0000 (005,000) SeqEnd
004: 0000010 xxxxx xx01 0000 0000 0000 (004)
005: 0000011 xxxxx 0101 0000 0000 0000 (006)     SeqEnd
006: 0000011 xxxxx 0001 0000 0000 0000 (006)
007: 0000011 xxxxx 0001 0000 0000 0000 (006)
008: 0000100 xxxxx xx01 xxxx xxxx xxxx (008)
009: xxxxxxx xxxxx xx01 xxxx xxxx xxxx (000)
010: 0000101 xxxxx xx01 xxxx xxxx xxxx (010)
011: xxxxxxx xxxxx xx01 xxxx xxxx xxxx (000)
012: 0000110 xxxxx 0001 1010 0001 0100 (012)     XfrTime DataTime ReadGate ECCShift
013: 0000111 00000 0001 1010 0001 0100 (014,031) XfrTime DataTime ReadGate ECCShift
014: 0000111 xxxxx 0001 1010 0001 0100 (014)     XfrTime DataTime ReadGate ECCShift
015: 1000010 00000 0001 1010 0001 0100 (132,031) XfrTime DataTime ReadGate ECCShift
016: 0001000 xxxxx 0001 0000 0000 0000 (016)

Write Header:
017: 0001001 11111 0001 0000 0010 0000 (019,000) WriteGate
018: 0001001 xxxxx xx01 0000 0010 0000 (018)     WriteGate
019: 0001010 10110 0000 0000 0010 0000 (020,009) ClrDevOp WriteGate
020: 0001010 xxxxx 0001 0000 0010 0000 (020)     WriteGate
021: 0001011 11111 0001 1000 0010 0000 (023,000) XfrTime WriteGate
022: 0001011 xxxxx 0001 1000 0010 0000 (022)     XfrTime WriteGate
023: 0001100 11110 0001 1100 0010 0000 (024,001) XfrTime SyncTime WriteGate
024: 0001100 xxxxx 0001 1100 0010 0000 (024)     XfrTime SyncTime WriteGate
025: 0001101 11110 0001 1011 0010 0100 (026,001) XfrTime DataTime CRCShift WriteGate ECCShift
026: 0001101 xxxxx 0001 1011 0010 0100 (026)     XfrTime DataTime CRCShift WriteGate ECCShift
027: 0001110 11110 0001 0011 0010 0100 (028,001) DataTime CRCShift WriteGate ECCShift
028: 0001110 xxxxx 0001 0011 0010 0100 (028)     DataTime CRCShift WriteGate ECCShift
029: 0001111 11110 0001 0001 1010 0000 (030,001) CRCShift CRCWrite WriteGate
030: 0001111 xxxxx 0001 0001 1010 0000 (030)     CRCShift CRCWrite WriteGate
031: 0000001 11110 0001 0000 0010 0000 (002,001) WriteGate
032: 0010000 xxxxx 0001 0000 0000 0000 (032)

Read Header:
033: 0010001 11111 0001 0000 0000 0000 (035,000)
034: 0010001 xxxxx xx01 0000 0000 0000 (034)
035: 0010010 11110 0000 0000 0001 0000 (036,001) ClrDevOp ReadGate
```

```
036: 0010010 xxxxx 0001 0000 0001 0000 (036)     ReadGate
037: 0010011 11011 0001 0000 0001 0000 (038,004) ReadGate
038: 0010011 xxxxx 0001 0000 0001 0000 (038)     ReadGate
039: 0010100 11110 0001 1000 0001 0000 (040,001) XfrTime ReadGate
040: 0010100 xxxxx 0001 1000 0001 0000 (040)     XfrTime ReadGate
041: 0010101 xxxxx 1001 0100 0001 0000 (042)     BrSync SyncTime ReadGate
042: 0010101 xxxxx 1001 0100 0001 0000 (042)     BrSync SyncTime ReadGate
043: 0010110 11100 0001 1011 0001 0100 (044,003) XfrTime DataTime CRCShift ReadGate ECCShift
044: 0010110 xxxxx 0001 1011 0001 0100 (044)     XfrTime DataTime CRCShift ReadGate ECCShift
045: 0010111 11110 0001 1001 0001 0100 (046,001) XfrTime CRCShift ReadGate ECCShift
046: 0010111 xxxxx 0001 1001 0001 0100 (046)     XfrTime CRCShift ReadGate ECCShift
047: 0000001 11110 0001 0000 0101 0000 (002,001) CRCCheck ReadGate
048: 0011000 xxxxx xx01 xxxx xxxx xxxx (048)

Write Label:
049: 0011001 11000 0001 0000 0010 0000 (050,007) WriteGate
050: 0011001 xxxxx 0001 0000 0010 0000 (050)     WriteGate
051: 0011010 11111 0001 1000 0010 0000 (053,000) XfrTime WriteGate
052: 0011010 xxxxx 0001 1000 0010 0000 (052)     XfrTime WriteGate
053: 0011011 11110 0001 1100 0010 0000 (054,001) XfrTime SyncTime WriteGate
054: 0011011 xxxxx 0001 1100 0010 0000 (054)     XfrTime SyncTime WriteGate
055: 0011100 10010 0001 1011 0010 0100 (056,013) XfrTime DataTime CRCShift WriteGate ECCShift
056: 0011100 xxxxx 0001 1011 0010 0100 (056)     XfrTime DataTime CRCShift WriteGate ECCShift
057: 0011101 11110 0001 0011 0010 0100 (058,001) DataTime CRCShift WriteGate ECCShift
058: 0011101 xxxxx 0001 0011 0010 0100 (058)     DataTime CRCShift WriteGate ECCShift
059: 0011110 11110 0001 0001 1010 0000 (060,001) CRCShift CRCWrite WriteGate
060: 0011110 xxxxx 0001 0001 1010 0000 (060)     CRCShift CRCWrite WriteGate
061: 0000001 11110 0001 0000 0010 0000 (002,001) WriteGate
062: 0011111 xxxxx xx01 xxxx xxxx xxxx (062)
063: xxxxxxx xxxxx xx01 xxxx xxxx xxxx (000)
064: 0100000 xxxxx xx01 xxxx xxxx xxxx (064)

Read/Verify Label:
065: 0100001 11111 0001 0000 0000 0000 (067,000)
066: 0100001 xxxxx xx01 xxxx xxxx xxxx (066)
067: 0100010 11100 0001 0000 0001 0000 (068,003) ReadGate
068: 0100010 xxxxx 0001 0000 0001 0000 (068)     ReadGate
069: 0100011 11110 0001 1000 0001 0000 (070,001) XfrTime ReadGate
070: 0100011 xxxxx 0001 1000 0001 0000 (070)     XfrTime ReadGate
071: 0100100 xxxxx 1001 0100 0001 0000 (072)     BrSync SyncTime ReadGate
072: 0100100 xxxxx 1001 0100 0001 0000 (072)     BrSync SyncTime ReadGate
073: 0100101 10000 0001 1011 0001 0100 (074,015) XfrTime DataTime CRCShift ReadGate ECCShift
074: 0100101 xxxxx 0001 1011 0001 0100 (074)     XfrTime DataTime CRCShift ReadGate ECCShift
075: 0100110 11110 0001 1001 0001 0100 (076,001) XfrTime CRCShift ReadGate ECCShift
076: 0100110 xxxxx 0001 1001 0001 0100 (076)     XfrTime CRCShift ReadGate ECCShift
077: 0000001 11110 0001 0000 0101 0000 (002,001) CRCCheck ReadGate
078: 0100111 xxxxx 0001 1010 0001 0100 (078)     XfrTime DataTime ReadGate ECCShift
079: 1110000 00000 0001 1010 0001 0100 (224,031) XfrTime DataTime ReadGate ECCShift
080: 0101000 xxxxx xx01 xxxx xxxx xxxx (080)

Write Data:
081: 0101001 11000 0001 0000 0010 1100 (082,007) WriteGate ECClr ECCShift
082: 0101001 xxxxx 0001 0000 0010 1100 (082)     WriteGate ECClr ECCShift
083: 0101010 11111 0001 1000 0010 0000 (085,000) XfrTime WriteGate
084: 0101010 xxxxx 0001 1000 0010 0000 (084)     XfrTime WriteGate
085: 0101011 11110 0001 1100 0010 0000 (086,001) XfrTime SyncTime WriteGate
086: 0101011 xxxxx 0001 1100 0010 0000 (086)     XfrTime SyncTime WriteGate
087: 0101100 00000 0001 1010 0010 0100 (088,031) XfrTime DataTime WriteGate ECCShift
088: 0101100 xxxxx 0001 1010 0010 0100 (088)     XfrTime DataTime WriteGate ECCShift
089: 0101101 00000 0001 1010 0010 0100 (090,031) XfrTime DataTime WriteGate ECCShift
090: 0101101 xxxxx 0001 1010 0010 0100 (090)     XfrTime DataTime WriteGate ECCShift
091: 0101110 00000 0001 1010 0010 0100 (092,031) XfrTime DataTime WriteGate ECCShift
092: 0101110 xxxxx 0001 1010 0010 0100 (092)     XfrTime DataTime WriteGate ECCShift
093: 0101111 00000 0001 1010 0010 0100 (094,031) XfrTime DataTime WriteGate ECCShift
094: 0101111 xxxxx 0001 1010 0010 0100 (094)     XfrTime DataTime WriteGate ECCShift
095: 1100010 00000 0001 1010 0010 0100 (196,031) XfrTime DataTime WriteGate ECCShift
096: 0110000 xxxxx xx01 xxxx xxxx xxxx (096)

Read Data:
097: 0110001 11111 0001 0000 0000 0000 (099,000)
098: 0110001 xxxxx xx01 xxxx xxxx xxxx (098)
099: 0110010 11010 0001 0000 0001 1100 (100,005) ReadGate ECClr ECCShift
```

```
100: 0110010 xxxxx 0001 0000 0001 1100 (100)     ReadGate ECClr ECCShift
101: 0110011 xxxxx 1001 0100 0001 0000 (102)     BrSync SyncTime ReadGate
102: 0110011 xxxxx 1001 0100 0001 0000 (102)     BrSync SyncTime ReadGate
103: 0110100 11111 0001 0010 0001 0100 (105,000) DataTime ReadGate ECCShift
104: 0110100 xxxxx 0001 0010 0001 0100 (104)     DataTime ReadGate ECCShift
105: 0110101 00001 0001 1010 0001 0100 (106,030) XfrTime DataTime ReadGate ECCShift
106: 0110101 xxxxx 0001 1010 0001 0100 (106)     XfrTime DataTime ReadGate ECCShift
107: 0110110 00000 0001 1010 0001 0100 (108,031) XfrTime DataTime ReadGate ECCShift
108: 0110110 xxxxx 0001 1010 0001 0100 (108)     XfrTime DataTime ReadGate ECCShift
109: 0110111 00000 0001 1010 0001 0100 (110,031) XfrTime DataTime ReadGate ECCShift
110: 0110111 xxxxx 0001 1010 0001 0100 (110)     XfrTime DataTime ReadGate ECCShift
111: 0000110 00000 0001 1010 0001 0100 (012,031) XfrTime DataTime ReadGate ECCShift
112: 0111000 xxxxx xx01 xxxx xxxx xxxx (112)

Verify Data:
113: 0111001 11111 0001 0000 0000 0000 (115,000)
114: 0111001 xxxxx xx01 xxxx xxxx xxxx (114)
115: 0111010 11100 0001 0000 0001 1100 (116,003) ReadGate ECClr ECCShift
116: 0111010 xxxxx 0001 0000 0001 1100 (116)     ReadGate ECClr ECCShift
117: 0111011 11110 0001 1000 0001 0000 (118,001) XfrTime ReadGate
118: 0111011 xxxxx 0001 1000 0001 0000 (118)     XfrTime ReadGate
119: 0111100 xxxxx 1001 0100 0001 0000 (120)     BrSync SyncTime ReadGate
120: 0111100 xxxxx 1001 0100 0001 0000 (120)     BrSync SyncTime ReadGate
121: 0111101 00000 0001 1010 0001 0100 (122,031) XfrTime DataTime ReadGate ECCShift
122: 0111101 xxxxx 0001 1010 0001 0100 (122)     XfrTime DataTime ReadGate ECCShift
123: 0111110 00000 0001 1010 0001 0100 (124,031) XfrTime DataTime ReadGate ECCShift
124: 0111110 xxxxx 0001 1010 0001 0100 (124)     XfrTime DataTime ReadGate ECCShift
125: 0111111 00000 0001 1010 0001 0100 (126,031) XfrTime DataTime ReadGate ECCShift
126: 0111111 xxxxx 0001 1010 0001 0100 (126)     XfrTime DataTime ReadGate ECCShift
127: 0100111 00000 0001 1010 0001 0100 (078,031) XfrTime DataTime ReadGate ECCShift
128: 1000000 xxxxx 0001 0000 0000 0000 (128)

Nop Header:
129: 1000001 01100 0001 0000 0000 0000 (130,019)
130: 1000001 xxxxx 0001 0000 0000 0000 (130)
131: 0000001 11110 0000 0000 0000 0000 (002,001) ClrDevOp
132: 1000010 xxxxx 0001 1010 0001 0100 (132)     XfrTime DataTime ReadGate ECCShift
133: 1000011 00000 0001 1010 0001 0100 (134,031) XfrTime DataTime ReadGate ECCShift
134: 1000011 xxxxx 0001 1010 0001 0100 (134)     XfrTime DataTime ReadGate ECCShift
135: 1000100 00000 0001 1010 0001 0100 (136,031) XfrTime DataTime ReadGate ECCShift
136: 1000100 xxxxx 0001 1010 0001 0100 (136)     XfrTime DataTime ReadGate ECCShift
137: 1000101 00000 0001 1010 0001 0100 (138,031) XfrTime DataTime ReadGate ECCShift
138: 1000101 xxxxx 0001 1010 0001 0100 (138)     XfrTime DataTime ReadGate ECCShift
139: 1000110 00000 0001 1010 0001 0100 (140,031) XfrTime DataTime ReadGate ECCShift
140: 1000110 xxxxx 0001 1010 0001 0100 (140)     XfrTime DataTime ReadGate ECCShift
141: 1000111 00000 0001 1010 0001 0100 (142,031) XfrTime DataTime ReadGate ECCShift
142: 1000111 xxxxx 0001 1010 0001 0100 (142)     XfrTime DataTime ReadGate ECCShift
143: 1001001 00000 0001 1010 0001 0100 (146,031) XfrTime DataTime ReadGate ECCShift
144: 1001000 xxxxx xx01 xxxx xxxx xxxx (144)

Nop Label:
145: 0000001 00001 0001 0000 0000 0000 (002,030)
146: 1001001 xxxxx 0001 1010 0001 0100 (146)     XfrTime DataTime ReadGate ECCShift
147: 1001010 00000 0001 1010 0001 0100 (148,031) XfrTime DataTime ReadGate ECCShift
148: 1001010 xxxxx 0001 1010 0001 0100 (148)     XfrTime DataTime ReadGate ECCShift
149: 1001011 00000 0001 1010 0001 0100 (150,031) XfrTime DataTime ReadGate ECCShift
150: 1001011 xxxxx 0001 1010 0001 0100 (150)     XfrTime DataTime ReadGate ECCShift
151: 1001100 00000 0001 1010 0001 0100 (152,031) XfrTime DataTime ReadGate ECCShift
152: 1001100 xxxxx 0001 1010 0001 0100 (152)     XfrTime DataTime ReadGate ECCShift
153: 1001101 00000 0001 1010 0001 0100 (154,031) XfrTime DataTime ReadGate ECCShift
154: 1001101 xxxxx 0001 1010 0001 0100 (154)     XfrTime DataTime ReadGate ECCShift
155: 1001110 11100 0001 1000 0001 0100 (156,003) XfrTime ReadGate ECCShift
156: 1001110 xxxxx 0001 1000 0001 0100 (156)     XfrTime ReadGate ECCShift
157: 1001111 11100 0001 1000 0001 1111 (158,003) XfrTime ReadGate ECClr ECCShift ECCWrite
ECCChk
158: 1001111 xxxxx 0001 1000 0000 1111 (158)     XfrTime ECClr ECCShift ECCWrite ECCChk
159: 0000010 11111 0111 0000 0000 0000 (005,000) SeqEnd SectWk
160: 1010000 xxxxx xx01 xxxx xxxx xxxx (160)

Nop Data:
161: 1010001 00000 0001 0000 0000 0000 (162,031)
162: 1010001 xxxxx 0001 0000 0000 0000 (162)
```

```
163: 1010010 00000 0001 0000 0000 0000 (164,031)
164: 1010010 xxxxx 0001 0000 0000 0000 (164)
165: 1010011 00000 0001 0000 0000 0000 (166,031)
166: 1010011 xxxxx 0001 0000 0000 0000 (166)
167: 1010100 00000 0001 0000 0000 0000 (168,031)
168: 1010100 xxxxx 0001 0000 0000 0000 (168)
169: 1010101 00000 0001 0000 0000 0000 (170,031)
170: 1010101 xxxxx 0001 0000 0000 0000 (170)
171: 1010110 00000 0001 0000 0000 0000 (172,031)
172: 1010110 xxxxx 0001 0000 0000 0000 (172)
173: 1010111 00000 0001 0000 0000 0000 (174,031)
174: 1010111 xxxxx 0001 0000 0000 0000 (174)
175: 1011001 00000 0001 0000 0000 0000 (178,031)
176: 1011000 xxxxx 0001 0000 0000 0000 (176)

177: 1011000 xxxxx 0001 0000 0000 0000 (176)
178: 1011001 xxxxx 0001 0000 0000 0000 (178)
179: 1011010 00000 0001 0000 0000 0000 (180,031)
180: 1011010 xxxxx 0001 0000 0000 0000 (180)
181: 1011011 00000 0001 0000 0000 0000 (182,031)
182: 1011011 xxxxx 0001 0000 0000 0000 (182)
183: 1011100 00000 0001 0000 0000 0000 (184,031)
184: 1011100 xxxxx 0001 0000 0000 0000 (184)
185: 1011101 00000 0001 0000 0000 0000 (186,031)
186: 1011101 xxxxx 0001 0000 0000 0000 (186)
187: 1011110 00000 0001 0000 0000 0000 (188,031)
188: 1011110 xxxxx 0001 0000 0000 0000 (188)
189: 1011111 01111 0001 0000 0000 0000 (190,016)
190: 1011111 xxxxx 0001 0000 0000 0000 (190)
191: 1100000 01111 0001 0000 0000 0000 (192,016)
192: 1100000 xxxxx 0001 0000 0000 0000 (192)

193: 1100001 00000 0001 0000 0000 0000 (194,031)
194: 1100001 xxxxx 0001 0000 0000 0000 (194)
195: 0000001 10001 0011 0000 0000 0000 (002,014) SectWk
196: 1100010 xxxxx 0001 1010 0010 0100 (196)     XfrTime DataTime WriteGate ECCShift
197: 1100011 00000 0001 1010 0010 0100 (198,031) XfrTime DataTime WriteGate ECCShift
198: 1100011 xxxxx 0001 1010 0010 0100 (198)     XfrTime DataTime WriteGate ECCShift
199: 1100100 00000 0001 1010 0010 0100 (200,031) XfrTime DataTime WriteGate ECCShift
200: 1100100 xxxxx 0001 1010 0010 0100 (200)     XfrTime DataTime WriteGate ECCShift
201: 1100101 00000 0001 1010 0010 0100 (202,031) XfrTime DataTime WriteGate ECCShift
202: 1100101 xxxxx 0001 1010 0010 0100 (202)     XfrTime DataTime WriteGate ECCShift
203: 1100110 00000 0001 1010 0010 0100 (204,031) XfrTime DataTime WriteGate ECCShift
204: 1100110 xxxxx 0001 1010 0010 0100 (204)     XfrTime DataTime WriteGate ECCShift
205: 1100111 00000 0001 1010 0010 0100 (206,031) XfrTime DataTime WriteGate ECCShift
206: 1100111 xxxxx 0001 1010 0010 0100 (206)     XfrTime DataTime WriteGate ECCShift
207: 1101000 00000 0001 1010 0010 0100 (208,031) XfrTime DataTime WriteGate ECCShift
208: 1101000 xxxxx 0001 1010 0010 0100 (208)     XfrTime DataTime WriteGate ECCShift

209: 1101001 00000 0001 1010 0010 0100 (210,031) XfrTime DataTime WriteGate ECCShift
210: 1101001 xxxxx 0001 1010 0010 0100 (210)     XfrTime DataTime WriteGate ECCShift
211: 1101010 00000 0001 1010 0010 0100 (212,031) XfrTime DataTime WriteGate ECCShift
212: 1101010 xxxxx 0001 1010 0010 0100 (212)     XfrTime DataTime WriteGate ECCShift
213: 1101011 00000 0001 1010 0010 0100 (214,031) XfrTime DataTime WriteGate ECCShift
214: 1101011 xxxxx 0001 1010 0010 0100 (214)     XfrTime DataTime WriteGate ECCShift
215: 1101100 00000 0001 1010 0010 0100 (216,031) XfrTime DataTime WriteGate ECCShift
216: 1101100 xxxxx 0001 1010 0010 0100 (216)     XfrTime DataTime WriteGate ECCShift
217: 1101101 00010 0001 1010 0010 0100 (218,029) XfrTime DataTime WriteGate ECCShift
218: 1101101 xxxxx 0001 1010 0010 0100 (218)     XfrTime DataTime WriteGate ECCShift
219: 1101110 11110 0001 0010 0010 0100 (220,001) DataTime WriteGate ECCShift
220: 1101110 xxxxx 0001 0010 0010 0100 (220)     DataTime WriteGate ECCShift
221: 1101111 11100 0001 0000 0010 1110 (222,003) WriteGate ECClr ECCShift ECCWrite
222: 1101111 xxxxx 0001 0000 0010 1110 (222)     WriteGate ECClr ECCShift ECCWrite
223: 0000010 11111 0101 0000 0010 0000 (005,000) SeqEnd WriteGate
224: 1110000 xxxxx 0001 1010 0001 0100 (224)     XfrTime DataTime ReadGate ECCShift

225: 1110001 00000 0001 1010 0001 0100 (226,031) XfrTime DataTime ReadGate ECCShift
226: 1110001 xxxxx 0001 1010 0001 0100 (226)     XfrTime DataTime ReadGate ECCShift
227: 1110010 00000 0001 1010 0001 0100 (228,031) XfrTime DataTime ReadGate ECCShift
228: 1110010 xxxxx 0001 1010 0001 0100 (228)     XfrTime DataTime ReadGate ECCShift
229: 1110011 00000 0001 1010 0001 0100 (230,031) XfrTime DataTime ReadGate ECCShift
230: 1110011 xxxxx 0001 1010 0001 0100 (230)     XfrTime DataTime ReadGate ECCShift
```

```
231: 1110100 00000 0001 1010 0001 0100 (232,031) XfrTime DataTime ReadGate ECCShift
232: 1110100 xxxxx 0001 1010 0001 0100 (232)     XfrTime DataTime ReadGate ECCShift
233: 1110101 00000 0001 1010 0001 0100 (234,031) XfrTime DataTime ReadGate ECCShift
234: 1110101 xxxxx 0001 1010 0001 0100 (234)     XfrTime DataTime ReadGate ECCShift
235: 1110110 00000 0001 1010 0001 0100 (236,031) XfrTime DataTime ReadGate ECCShift
236: 1110110 xxxxx 0001 1010 0001 0100 (236)     XfrTime DataTime ReadGate ECCShift
237: 1110111 00000 0001 1010 0001 0100 (238,031) XfrTime DataTime ReadGate ECCShift
238: 1110111 xxxxx 0001 1010 0001 0100 (238)     XfrTime DataTime ReadGate ECCShift
239: 1111001 00000 0001 1010 0001 0100 (242,031) XfrTime DataTime ReadGate ECCShift
240: 1111000 xxxxx 0001 0000 0000 0000 (240)

Recovery Gap:
241: 1111000 xxxxx 0001 0000 0000 0000 (240)
242: 1111001 xxxxx 0001 1010 0001 0100 (242)     XfrTime DataTime ReadGate ECCShift
243: 1111010 00000 0001 1010 0001 0100 (244,031) XfrTime DataTime ReadGate ECCShift
244: 1111010 xxxxx 0001 1010 0001 0100 (244)     XfrTime DataTime ReadGate ECCShift
245: 1111011 00000 0001 1010 0001 0100 (246,031) XfrTime DataTime ReadGate ECCShift
246: 1111011 xxxxx 0001 1010 0001 0100 (246)     XfrTime DataTime ReadGate ECCShift
247: 1111100 00010 0001 1010 0001 0100 (248,029) XfrTime DataTime ReadGate ECCShift
248: 1111100 xxxxx 0001 1010 0001 0100 (248)     XfrTime DataTime ReadGate ECCShift
249: 1111101 11110 0001 0010 0001 0100 (250,001) DataTime ReadGate ECCShift
250: 1111101 xxxxx 0001 0010 0001 0100 (250)     DataTime ReadGate ECCShift
251: 1111110 11100 0001 0000 0001 0100 (252,003) ReadGate ECCShift
252: 1111110 xxxxx 0001 0000 0001 0100 (252)     ReadGate ECCShift
253: 1111111 11100 0001 0000 0001 1111 (254,003) ReadGate ECClr ECCShift ECCWrite ECCChk
254: 1111111 xxxxx 0001 0000 0000 1111 (254)     ECClr ECCShift ECCWrite ECCChk
255: 0000010 11111 0101 0000 0000 0000 (005,000) SeqEnd
```