Dorado Hardware Manual

by E.R. Fiala

contributions to the manual by
R. Bates, D. Boggs, B. Lampson, K. Pier, and C. Thacker

other help by
D. Clark, W. Crowther, W. Haugeland, G. McDaniel,
S. Ornstein, and E. Taft

8 October 1979

The document describes the architecture and hardware design of the Dorado computer at a level appropriate for programming. At this time, three prototype machines are being used for firmware and software development, and production has commenced for most hardware subsections.

This release of the manual is substantially revised from the 14 February 1979 release. There are new chapters on the Disk, Ethernet, and Display controllers and the 'Junk IO' chapter has been replaced by an 'Other IO and Event Counters' chapter. There are many other changes, particularly in the 'Memory Section' and 'Instruction Fetch Unit' chapters.

XEROX

Palo Alto Research Center
Computer Sciences Laboratory
3333 Coyote Hill Rd.
Palo Alto, California 94304

Table of Contents

List of Tables

List of Figures

Introduction

Dorado is a high performance, medium cost microprogrammed computer designed primarily to implement a virtual machine for the Mesa language, as described in "The OIS Processor Principles of Operation," and to provide high storage bandwidth for picture-processing applications. Dorado aims more at word processing than at numerical applications.

The microprocessor has a nominal cycle time of 50 ns, and most Mesa opcodes will execute in one or two cycles; the overall average opcode execution time will be subject to a number of considerations discussed later. Dorado will also achieve respectable performance when implementing virtual machines for the Alto, Interlisp, and Smalltalk programming systems, although simple instructions for these run three to five times slower than Mesa.

Dorado is implemented primarily of MECL-10K integrated circuits; storage boards use MOS and Schottky-TTL components primarily. Backplanes and storage boards are printed circuits; other logic boards are stitchweld in the prototypes and will be multiwire in the production machines. The mainframe is divided into sections called Control, Processor, Instruction Fetch Unit (IFU), and Memory, and peripheral control is accomplished by the Disk, Ethernet, and Display Controller sections, as discussed in chapters of this manual. The main data paths, shown in Figure 1, are 16-bits wide (the word size). The control section is shown in Figure 5. The Baseboard section, used to control the mainframe, is discussed in the "Dorado Debugging Interface" document.

The processor is organized around an Arithmetic and Logic Unit (ALU) whose two inputs are the A and B data paths (Figure 1), and whose output is normally routed to the Pd data path. Inputs to A, B, and Pd include all registers accessible to the programmer. In addition, 16-bit literal constants can be generated on B. B appears on the backplane for communication with the IFU, Control, and Memory sections.

The processor also includes a 32-bit in/16-bit out shifter-masker optimized for field insertion and extraction and with specialized paths for the bit-boundary block transfer (BitBlt) instruction.

An instruction fetch unit (the IFU) operating in parallel with the processor can handle up to four instruction sets with 256 opcodes each.

Emulator and IFU references to main memory are made through a 4k-word high-speed cache. Main storage can be configured in various sizes up to a maximum of 2^{20} 16-bit words (i.e., 1,048,576 words) or eventually up to 2^{22} 16-bit words when 64k x 1 RAM's become available.

The processor initiates data transfers between main memory and fast input/output devices. 16 16-bit words are then transmitted without disturbing the main data paths of the microprocessor in about 1.4 μs (28 cycles). New requests can be initiated every 8 cycles, so total bandwidth of the memory, 640 mHz, is available for devices with enough buffering.

Overview

Control

Dorado supports up to 16 independent tasks at the microcode level. Each task has its own program counter (TPC), and other commonly-used registers are also replicated on a per-task basis. Tasks are scheduled automatically by the hardware in response to wakeup requests, where task 15 is highest priority, task 0, lowest.

Emulator microcode runs entirely in task 0 (lowest priority); fault conditions normally wakeup task 15, the "fault task" (highest priority). Other tasks are normally paired with io devices that issue wakeup requests when they need service. Task switching, discussed in "Control Section", is in most cases invisible to the programmer, because commonly-used registers are duplicated for each task.

Read the following with Figure 1 in front of you.

Data Paths

Primary registers and data paths in Dorado are shown in Figure 1. The main data paths are A, B, Pd ("processor data"), Md ("memory data"), Id ("IFU data"), IOA ("io address"), IOB ("io bus"), Fin ("fast input bus"), and Fout ("fast output bus").

"A" is used to supply memory addresses (via the Mar bus on the backplane) and as an ALU input. Several registers inside the processor section source A or receive A as an input, and Id (IFU section) sources A.

B is the other ALU input. B appears on the backplane for communication between the processor and other sections of Dorado. There are numerous external sources and destinations; most internal processor registers load from and source B.

The Pd path has the ALU, IOB, and several seldom-read registers inside the processor as sources; T and RM/STK are its destinations. The ALU input to this path can be shifted left or right one by various functions. Shift-and-mask operations replace any ALU field by zeroes or Md.

The TIOA register sources IOA on the backplane.

Fin and Fout are used for high bandwidth transfers between storage and io devices. Among standard peripherals, only the display controller uses the fast io system (Fout).

Registers and Memories

Tables 1 and 2 describe registers and memories available to the programmer.

Table 1: Memories

Memory	Comments
IM	IM is a 4096-word x 34-bit (+2 parity) RAM used to store instructions. When written, the address is taken from Link and data from B 16 bits at-a-time (1 extra bit and parity from RSTK field). When read, the address is taken from Link, and data is delivered to Link 9 bits at-a-time. The read or write is controlled by the JCN field and two or three low bits of RSTK.
ALUFM	ALUFM is a 16-word x 6-bit ALU control RAM addressed by the 4-bit ALUF field. Five ALUFM bits specify 16 boolean or 5 arithmetic operations on A and B. One bit is the input carry for arithmetic operations (modifiable by several functions). ALUFM[ALUF] is read onto Pd by the ALUFMEM function or both read onto Pd and loaded from B by the ALUFMRW← function.
RM	RM is a 256-word x 16-bit (+2 parity) RAM used for general storage by all tasks. The normal address is RBase[0:3],,RSTK[0:3]. Data can be read onto A or B and loaded from Pd or Md without using FF. Together with T, RM forms the input to the Shifter.
STK	STK is a 256-word x 16-bit (+2 parity) stack accessible only to the emulator, used instead of RM when the BLOCK bit in the instruction is 1. Its address comes from StkP, modified by -4 to +3 under control of RSTK.
IFUM	IFUM is a 1024-word x 24-bit (+3 parity) decoding memory containing 256 words for each of four instruction sets. The instruction set can be set by the InsSetOrEvent← function. The low 8 address bits are normally an opcode fetched from the cache, but can be loaded from B by the BrkIns← function to read or write IFUM itself. The IFUMLH← and IFUMRH← functions load, and the B←IFUMLH' and B←IFUMRH' functions read different bits of IFUM. During normal operation IFUM controls decoding of the stream of opcodes and operands fetched from memory relative to BR 31, the code base.
MAIN	Main storage consists of a 64-row x 4-column x 16-word virtual cache coupled with one to four 256k x 16-bit memory modules (using 16k-bit storage chips). The IFU and processor independently access the cache, with IFU references deferring to the processor. The processor has two dissimilar methods of reference, one primarily to the cache (with "misses" initiating main memory action) and one directly to main memory (invalidating cache hits on writes, using dirty cache hits on reads). Fetch←, Store←, IFetch←, LongFetch←, and PreFetch← are cache references. Md can be loaded into T or RM (LC field), routed onto B (BSEL field), onto A (FF field), or used in a shift-and-mask operation (ASEL and ALUF fields). IOFetch← and IOStore← (ASEL field) initiate a 16-word transfer between an io device and memory without further processor interaction (using Fin or Fout bus). Virtual addresses are transformed to absolute using the Map memory. All references leave information in the Pipe memory.
BR	A 32-word x 28-bit base register memory addressed by the MemBase register. The virtual address for any memory reference is BR[MemBase]+Mar. BR is loaded from Mar by the BrLo← and BrHi← functions and can be read indirectly onto B via the virtual address left in the Pipe after a memory reference (Pipe0 and Pipe1 functions).
Pipe	The 16-entry x 6-word pipe contains trace information left by memory references. This information includes the virtual address, map stuff, single-error and double-error information, cache control stuff, task and subtask. It is automatically loaded during any memory reference and can be read onto B by the Pipe0, Pipe1, ..., Pipe5' functions.
Map	The Map is a 16k-word x 19-bit (+parity) memory used to transform virtual addesses to absolute. Addressed by VA[10:23], map entries contain 16 bits of real page, write protect, dirty, and referenced bits. They can be written from B with Map← (ASEL) and read from the Pipe after main storage references.

Table 2: Registers

Register	Comments * = task specific
T*	16-bit (+2 parity) T sources either A (ASEL field or FA field with memory ops) or B (BSEL field), or the Shifter (ASEL) and loads from either Pd or Md (LC field).
RBase*	4-bit RBase,,RSTK field forms addresses for RM. RBase can be loaded from FF[4:7] or from B[12:15] by the RBase←SC, RBase←B, or Pointers←B functions; it is read onto Pd[12:15] by the Pd←Pointers function. RBase is loaded with 0 or 1 when the IFU dispatches to the first instruction for an opcode.
StkP	The emulator uses STK instead of RM when the BLOCK bit is 1. 8-bit StkP holds the address for STK. The RSTK field is interpreted as an adjustment to StkP, which can be modified -4 to +3 in conjunction with testing for overflow and underflow. This mechanism implements the Mesa evaluation stack. StkP can be loaded by the StkP←B function and read onto Pd[8:15] by the ←TIOA&StkP function (Stack overflow and underflow indicators are read into Pd[8:9] by the Pd←Pointers function.).
Q	16-bit Q is used as a shift register by multiply and divide. Q can be read onto A (FF field or FA with Fetch← or Store←) or B (BSEL field) and loaded from any B source except a constant (BSEL and FF fields). Functions implement Q lsh 1 and Q rsh 1.
Cnt	Cnt is a 16-bit counter that can be both decremented and tested for zero by a branch condition. Cnt can be loaded from FF[4:7] with 1 to 16 or from B (FF field) and can be read onto Pd (FF).
TIOA*	TIOA is an 8-bit io address register (see "Slow IO") loaded by the TIOA←B function and read onto Pd[0:7] with the Pd←TIOA&StkP function. TIOA[5:7] may also be loaded from FF[5:7].
ShC	16-bit ShC controls the shifter-masker (see "Shifter"). RF←A, WF←A, and ShC←B functions load ShC in various ways. ShC can be read onto Pd by the Pd←ShC function.
MemBase*	MemBase is a 5-bit register addressing BR for memory references. The MemBase←n functions load it from FF[3:7]; the MemBaseX←n functions load it from 0,,MemBX[0:1],,FF[6:7]. The IFU loads MemBase with a value between 0 and 3 relative to MemBX or with 34 to 37, as specified in IFUM, prior to executing the first instruction of an opcode. MemBase is read onto Pd[3:7] by the Pd←Pointers function and loaded from B[3:7] by the Pointers←B and MemBase←B functions.
MemBX	MemBX is a 2-bit register used like a stack pointer in conjunction with MemBase. The ideas behind this are discussed in "Memory Section".
Link*	16-bit Link holds subroutine return addresses, address-modification for dispatches, IM address for IM reads/writes, and data for TPC reads/writes. It can be read onto or loaded from B[0:15] by the B←Link or Link←B, BigBDispatch←B, or BDispatch←B functions, or from CIA+1 by CALLs and RETURNs.
PC	16-bit PC contains the byte displacement of the next opcode relative to BR 31, the code base. The IFU maintains this register, so only conditional jumps that don't jump and opcodes of type "pause" have to load it with the PCF←B function. The B←PCX' function reads PC.
TPC*	TPC contains the address of the next instruction for each task. It is addressed from B[12:15] and read/write control is in JCN. Data is read from/written into Link under control of the JCN field of the instruction.
Mcr	Memory control register--disables parts of memory system for initialization and checkout.

Timing

The terminology used in discussing timing is as follows:

clock	The 25 ns (nominal) atomic time period of the machine.
cycle	The duration of instructions--two clocks or 50 ns except for instructions that read/write IM or TPC.
t_0	The instant at which MIR (MicroInstruction Register) is loaded--the beginning of a cycle.
t_1	The next instant after t_0--always one clock later.
t_2	The instant following t_1--one clock after t_1 except for instructions that read/write IM or TPC. Additional clocks intervening for these special cases, which only affect the control section, are denoted by t_{1a}, t_{1b}, etc.
t_3, t_4	Subsequent instants for a instruction. t_3 of the previous instruction coincides with t_1 of the current instruction; t_4 with t_2.
First half cycle	The interval from t_0 to t_1 (or t_2 to t_3).
Second half cycle	The interval from t_1 to t_2 (or t_3 to t_4).

As implied by this terminology, Dorado initiates a new instruction every cycle. Instructions are pipelined, requiring a total of three cycles for execution. Timing for a typical instruction is shown in Figure 7. At t_{-2}, the next instruction address is determined and instruction fetch from IM begins; at t_0, the instruction is loaded into MIR from IM. During the first half cycle, the selected register is read from RM or STK, and at t_1 is loaded into a register. During the next two clocks (t_1-t_3), addition is performed in the ALU; at t_3 the result is loaded into a register for writing into RM/STK or T. During the final clock, RM is written.

Since a new instruction begins before the previous one finishes, paths exist to bypass the register being written if the following instruction specifies it as a source (These paths, inaccessible to the programmer, are not shown in Figure 1).

Most registers load from B at t_3 (i.e., at the mid-clock of the cycle following the load instruction). These may source B in the instruction after they are loaded. The load information and data are pipelined into the next cycle, as described above. Registers loaded at t_2 may be used during the first half-cycle of the following instruction. Usually, this type of register is used for some type of control information, since control registers are normally clocked at t_0 (= t_2 of previous instruction), data-oriented registers at t_1 (t_3 of previous instruction).

Table 3 summarizes the time at which loading takes place and some other information.

Table 3: Load Timing

Register/ Memory	Task Specific	Load Time	Data Source	Load Control	Comment
MIR*	no	t0	IM	JCN	Microinstruction
CIA	no	t0	TNIA,BNPC	JCN	Microinstruction address
CIAinc*	no	t1	CIA	--	
TPCI*	no	t2	TNIA, CIA	--	
TPC	yes	FHC	TPCI	HOLD	
		t2	LINK(??)	JCN,B	Reading/writing takes 3 cycles
Link	yes	t2	B	FF	Also loaded by CALL, RETURN, and dispatches--readout valid t1 to t3
IM	no	--	B	JCN	Reading/writing require 3 cycles
CTASK	no	t0	Next	Switch	Current task
CTD	no	t1	CTASK	--	Current task delayed
Ready	no	t0	PEnc	Switch	Task-ready flipflops
StkP	no	t2	B	FF	New value read if it changes in the same instruction
RBase	yes	t2	F2	F1	RAM writte at t3, bypassed
Cnt	no	t2	F2	F1	Br cond to sub 1 and test
			B	FF	
ALUFM	no	t2	B	FF	Addressed by ALUF The output is valid t1 to t3
TIOA	yes	t2	B	FF	Readout valid till t3
MemBX	no	t2	F2	F1	Readout valid till t3
MemBase	yes	t2	F2	F1	Readout valid till t3
			MemBase xor 1	FF	
ShC	no	t3	A,B	FF	RF←A, WF←A, ShC←B
		t1	FF	ASEL,BSEL	
Q	no	t3	B	FF,BSEL	
		t3	ALU[15],,Q[0:14]	FF	Multiply
		t3	Q[1:15],,ALUcry	FF	Divide
		t3	Q	FF	Q rsh 1, Q lsh 1
RM	no	SHC	Pd,Md	LC,RSTK	Bypassed
STK	no	SHC	Pd,Md	LC,RSTK	Bypassed
T	yes	SHC	Pd,Md	LC,FF	Bypassed
IFUM	no	SHC	B	FF	IFUMLH←/IFUMRH←
BrkIns	no	t2	B	FF	
PC*	no	t3	B	FF	Level F PC loaded, level X read
Br	no	t2	A	FF	BrLo←/BrHi←, ←Pipe0, ←Pipe1
MapBuf*	no	FHC	B	ASEL, FA	Written on Map←, TestSyndrome←, ProcSRN←, LoadMcr
DBuf	no	FHC	B	ASEL, FA	Written on Store←
Md	yes	t5	cache	--	Bypassed
CFlags	no	t2	Mar	FF	For debugging, initialization
Mcr	no	t3	Mar, MapBuf	FF	For debugging, initialization
Asrn	no	t2	Asrn	--	Addresses the pipe for ring refs
ProcSRN	no	t3	MapBuf	FF	Addresses the pipe for Pipe0 to Pipe5
TestSyndrome	no	t3	MapBuf	FF	For debugging error correction
Pipe0	no	t3	Br, etc.	ASEL, FA	Written on ref., B←Pipe0
Pipe1	no	t3	Br, etc.	ASEL, FA	Written on ref., B←Pipe1
Pipe2	no	t3	--	ASEL, FA	Written on ref., B←Pipe2
Pipe3	no	t14	map	ASEL, FA	Valid after any storage access or Map←
Pipe4	no	t14,t48	map, EC	ASEL, FA	Valid after any storage access
Pipe5	no	t3,t4	cache	ASEL, FA	Written on ref., B←Pipe5

*Cannot be read as data by the processor

Instruction Fields

The 34-bit instruction is divided into the following fields:

Table 4: Instruction Fields

Field	Size	Purpose (may have other effects, described below)
RSTK	4 bits	Selects RM register to be read and/or written
ALUF	4 bits	Selects ALU function or shifter operation
BSEL	3 bits	Selects source for B
LC	3 bits	Controls source and loading of RM and T
ASEL	3 bits	Source/destination control for A
BLOCK	1 bit	Blocks io task unless wakeup is waiting Selects stack operations for emulator task
FF	8 bits	Function (FA = FF[0:1], FB = FF[2:4], FC = FF[5:7])
JCN	8 bits	Jump control
P016	1 bit	Odd parity on first word of instruction
P1733	1 bit	Odd parity on second word of instruction
Total	34 bits + 2 parity	

The above instruction layout emphasizes compactness at the expense of programming flexibility. The following comments explain some of these tradeoffs

1. The RSTK field specifies only four of the eight address bits needed for addressing RM. The other four are taken from the RBase register (loaded by a function). In the emulator task, BLOCK causes STK to be used instead of RM, and RSTK is decoded to cause modifications of StkP.

2. ALUF addresses the 16-word ALUFM memory in which 16 of 26-odd useful ALU operations are stored. For the shift operation decode of ASEL, the first three bits of ALUF select the kind of shift, while the ALUFM address is forced to 14 or 15.

3. BSEL decodes the most common data sources for B. Less common B sources are selected by FF, and then BSEL encodes one of several destinations for the source.

4. ASEL specifies the source and destination for A. The default source is the RM address selected by RSTK. Four ASEL decodes specify the most common memory operations, where the virtual address is BR[MemBase] + A. These decodes consume the two leading bits of FF to specify alternate sources (T or Id) or less frequent memory operations. The remaining four ASEL decodes select alternate sources T, Id, or the shifter, where the shifter decodes work in combination with ALUF, as discussed later.

5. LC specifies loading of RM/STK and T from Fd and Md.

6. FF is the catch-all field in which operations or data not otherwise specifiable can be encoded. Operations encoded in FF are called "functions". There are five ways FF is used:

a. To extend the branch address encoded in JCN (long goto, long call).
b. To form a constant on B as selected by BSEL.
c. To specify one of 64 common functions and branch conditions while the two leading bits modify the memory reference operation specified in ASEL.
d. To specify one of 256 functions and branch conditions, some of which use low bits of FF as literal values.
e. As a shift control value when ASEL decodes to "shift" and BSEL to a constant.

When FF is used as a function, it sometimes modifies the interpretation of other fields in the instruction. For example:

a. 16 FF decodes modify RM write address bits which would otherwise have come from RSTK or StkP.
b. 16 FF decodes modify RM write-address bits which would otherwise come from RBase.
c. 16 FF decodes select less common B sources, causing BSEL to encode a destination rather than a source for B.

7. JCN (in conjunction with current address) encodes the next instruction address as follows:

a. One of 64 global Calls.
b. One of 60 local Gotos.
c. One of 4 local Calls.
d. One of 14 local conditional branches with 7 branch conditions.
e. One of 16 long Gotos/Calls (use FF field for rest of address).
f. One of 4 IFU jumps for next opcode (high 10 address bits from IFU).
g. Return.
h. TPC read/write.
i. IM read/write (Use low bits of RSTK also).

8. P0 and P1 are odd parity on the left and right halves of IM. When wrong, these give rise to error signals (see "Theory of Operations") which stop the machine after (unfortunately) the instruction with bad parity has been executed. The artifice of deliberately loading both parity bits incorrectly is used to implement breakpoints.

Notation

The notation used in referring to fields in the instruction is that the left-most bit of the field is denoted as 0. Hence, the fields in the instruction are as follows: RSTK[0:3], ALUF[0:3], BSEL[0:2], LC[0:2], ASEL[0:2], BLOCK[0], FF[0:7], JCN[0:7].

The BLOCK bit is also called StackSelect, for its use in choosing STK instead of RM for the emulator task.

Processor Section

The processor section implements most registers accessible to the programmer and decodes all instruction fields except JCN. The FF field of the instruction is also decoded by the control, memory, IFU, and Junk sections.

Read this chapter with Figure 1 in front of you.

The processor section contains the Q, ShC, Cnt, StkP, and MemBX registers, the T, RBase, MemBase, and TIOA task-specific registers, and the ALUFM, RM, and STK memories. It contains the arithmetic and logic unit (ALU) and the shifter.

The processor communicates with the control, memory, and IFU sections via B; with io devices via the IOB bus. It exports MemBase and Mar to the memory system for addressing, IOA to devices for io addressing, and branch conditions to the control section. It imports Md from the memory system and Id from the IFU.

RM and STK Memories, StkP and RBase Registers

The RM and STK memories each store 256 words x 16 data bits with odd parity on each byte of data. The STK memory is accessible only to the emulator. Either RM or STK is read at t_0 and latched at t_1. Data may be routed to A, B, or the shifter, and branch conditions (see "Control") test the sign bit (R<0) and low bit (R odd). Either RM or STK may be written between t_3 and t_4 with data from Md or Pd.

The RM read address is RBase[0:3],,RSTK[0:3]. For io tasks SubTask[0:1] (discussed in "Slow IO") are or'ed with RBase[2:3]. Each task can thus select from 16 RM registers in the block pointed to by RBase.

Normally, this read address is also used for the write part of the instruction (if any). However, two groups of FF decodes discussed below modify the write address.

The RBase←SC function loads RBase with FF[4:7], selecting any block of 16 registers; RBase←B loads RBase from B[12:15]; Pointers←B loads RBase from B[12:15] while also loading MemBase from B[3:7] (Previous RBase value is used for both the read and write portions of the instruction.). The IFU initializes the emulator task's RBase to 0 or 1 before dispatching to the first instruction of an opcode.

In the emulator (task 0), if BLOCK (i.e., StackSelect) is 1, RM is disabled and STK used instead. STK is addressed by the 8-bit StkP register, and RSTK controls the adjustment of StkP; StkP may be decremented or incremented by any value between -4 and +3.

Unadjusted StkP is always the read address and normally the write address, but the ModStkPBeforeW FF decode forces *adjusted* StkP to be used for the write. STK is divided into four separate regions, each 100_8 words long. Valid addresses are 1 to 77_8 within each region. That is, StkP[0:1] select the region, stack overflow occurs at the onset of a instruction that would increment StkP[2:7] > 77_8, and underflow occurs when location 0 is either read or written or when StkP[2:7] is decremented below 0.

StkP[2:7] are initialized to 0, denoting the empty stack. A push could do StkP←StkP + 1 and

write in one instruction. A pop does StkP←StkP-1, and the item being popped off can be referenced in the same instruction if desired.

Table 5: RSTK Decodes for Stack Operations

RSTK[0]

0 = no underflow on StkP = 0 at start or end
1 = underflow when StkP originally 0 or finally 0.

RSTK[1:3]	Meaning
0	no StkP change
1	StkP←StkP+1
2	StkP←StkP+2
3	StkP←StkP+3
4	StkP←StkP-4
5	StkP←StkP-3
6	StkP←StkP-2
7	StkP←StkP-1

In other words, RSTK[1:3] treated as a signed number are added to StkP[2:7] (StkP[0:1] don't change.). In the emulator, an attempt to underflow or overflow the stack generates the signal StkError:

```
StkError = (BLOCK eq 1) & Emulator &
    [((StkP[2:7] + RSTK[1:3]) < 0) % ((StkP[2:7] + RSTK[1:3]) > 77₈) %
    ((RSTK[0] eq 1) & ((StkP[2:7] eq 0) % ((StkP[2:7] + RSTK[1:3]) eq 0)))]
```

StkError generates HOLD and wakes up the fault task (task 15) to deal with the situation, so the instruction causing StkError has not been executed when the fault task runs. StkUnd and StkOvf are remembered in flipflops read by the Pd←Pointers function. These get cleared (i.e., recomputed) when the next stack operation is executed by the emulator. The fault task can read them to decide whether stack underflow or overflow action is necessary.

Interpretation of underflow: StkP eq 0 denotes the empty stack. A stack adjustment may occur either by itself or with a read or write stack reference. *StkP originally equal 0* underflows if the top of stack is read or written; *decrementing StkP below 0* is always an underflow error; *StkP equal 0 after modification* underflows iff writing at the modified address. Consequently, the assembler sets RSTK[0] equal 1 for a stack reference only when either reading STK and incrementing the pointer or writing at the modified address and decrementing the pointer.

In other words, the microassembler must tell the hardware when to make the StkP equal 0 underflow checks, and it must do this correctly when the ModStkPBeforeW FF decode is used.

StkP is saved at t_2 of an instruction dispatched to by the IFU. The saved value may be reloaded into StkP at t_2 by the RestoreStkP function.

The utility of RestoreStkP will not be known until we decide how to continue from map faults. It will only be useful if opcodes are *restarted* after servicing map faults. However, we are also arranging for the IFU state, branch conditions, etc. of an interrupted opcode to be readable and reproducible, so that it will be possible to simply *continue* from the instruction that faulted. RestoreStkP will be useless if the continue-method of restarting is adopted.

Two groups of FF decodes change the RM address for the write portion of an instruction.

The first group of 16 FF decodes forces the write address to come from RBase[0:3],,FF[4:7]. This allows different registers in the same group of 16 to be used for the read and write portions of the instruction, or allows STK[StkP] to be used for the read portion and any of the 16 registers pointed to by RBase in the write portion. Note: *SubTask does not affect the write address for these functions.*

The second group of 16 FF decodes forces the top four write address bits to come from FF[4:7]. The complete RM write address becomes FF[4:7],,RSTK[0:3]. This allows an arbitrary RM address to be written without having to load RBase in a previous instruction. Alternatively, if the i'th register in a group of 16 is read from RM, it permits the i'th register in a different group of 16 to be written in the same instruction. In conjunction with a read of STK, RSTK[0:3] will encode the StkP modification, and whatever RM word this happens to point to will be written (Programmers will have to struggle to use this with a STK read.).

The RisId FF decode causes Id to be substituted for RM/STK in the A, B, or shifter multiplexing.

There are branch conditions to test R[0] (R<0) and R[15] (R odd). These branch conditions are *unaffected* by the RisId FF decode; actual data from RM/STK is tested.

Cnt Register

The 16-bit Cnt register is provided for use as a loop counter. Since it is not task-specific, io tasks cannot conveniently use it.

Cnt can be decremented and tested for 0 by the Cnt = 0&-1 branch condition; loaded from B[0:15] or from small constants 1 to 16 (FF decodes), and read onto the Pd path (into T or RM/STK) by an FF decode.

Q Register

The 16-bit Q register is provided primarily for use as a shift register with multiply and divide, but will probably be used more widely by the emulator. Since it is not task-specific, io tasks cannot conveniently use it.

Q can be read onto B (BSEL) or onto A (FF); it can be loaded from B (FF) and when FF specifies an external B source in the memory, ifu, or control sections, it can also be loaded from B (BSEL). Q can be left-shifted or right-shifted one (bringing 0 into the vacant bit) by two FF decodes.

T Register

The 16-bit T register is the primary register for data manipulation in the processor. Since it is task-specific io tasks do not have to save and restore it. T can be read onto B (BSEL) or A (ASEL); it can be loaded from Pd or Md (LC).

BSEL: B Multiplexor Select

BSEL normally selects the source for B. However, when this selection is overruled by one of the FF decodes for external B sources (an assortment of stuff in the memory, IFU, and control sections), then BSEL may instead encode the destination for B. B sources that originate outside the processor are called "external". FF decodes for other B sources are discussed later (see "Special Function").

The sources selected by BSEL are:

Table 6: BSEL Decodes

BSEL	Primary	External
0	Md	--
1	RM/STK	--
2	T	--
3	Q	Q←B *
4	0,,FF	Inapplicable
5	377_8,,FF	Inapplicable
6	FF,,0	Inapplicable
7	FF,,377_8	Inapplicable

*Note: BSEL decode for Q←B is needed in initializing Dorado from the baseboard or Alto. Because ALUFM contents may be unknown, and data from the Alto is transmitted via the B←Link FF decode, some other field is needed to encode a destination that can then be routed into ALUFM.

The values selected by BSEL = 4-7 are 16-bit constants obtained by concatenating the 8-bit FF field with zeroes or ones. When this is done, normal effects of functions are disabled, so primary and external BMux stuff doesn't apply. In conjunction with a shift operation on A, BSEL = 4 to 7 will cause the shifter controls to come directly from FF rather than from ShC as discussed in "Shifter"; *the Q-register sources B when an FF-controlled shift is carried out.*

The TisId and RisId FF decodes may be used with the B←T or B←RM/STK BSEL decodes, respectively, to accomplish B←Id.

The "External" decode of BSEL applies with Link, DBuf, Pipe0-Pipe5, FaultInfo, PCX, DecLo, DecHi, and other functions that source B on the backpanel, as selected by the FF decode. For these external sources, BSEL is interpreted as the destination for B rather than the source.

Note: When the memory or control section sources the external B bus, it is *illegal* to execute A + B or A-B alu operations; these sources are not electrically stable soon enough to permit the extra 10 ns required for carry propagation. *But*: if you are sure carries will not propagate into the high 8 bits of ALU result, then the hardware is fast enough.

However: Arithmetic is permitted when the IFU sources the external B bus, provided the previous instruction was not one of the slow B sources from the memory or control sections. This permits (Id)-(PCX')-1, common in emulator microcode.

This implies that an io task must never block on an instruction that sources B from a slow external source.

Hardware Implementation

The processor's internal version of B, called Alub, is driven by a 4-input multiplexor when sourced from within the processor; in this case an identical multiplexor drives the external bus, called Bmux (High-true). When the B source is external, both of these multiplexors are disabled, and the backpanel Bmux (Low-true) is inverted through a gate onto Alub. The multiplexor arrangement is shown in Figure 3.

The IFU section is on/off of Bmux by t1 + 6 ns and the processor section is off by t1 + 7 ns, but the memory and control sections are not on/off until t1 + 16 ns; hence, a slow Bmux source in the previous instruction prevents Bmux from stabilizing until t1 + 16 ns of the current instruction, allowing insufficient time to propagate Bmux onto Alub and finish carry propagation. However, because Bmux is gated onto Alub, and the gate shuts off quickly, arithmetic on internal Alub sources is always permissible.

Bmux sources in this manual are given high or low true names that agree with the way signals appear on Alub. For external sources this is inverted with respect to the sense of these signals on Bmux. However, because external sources cannot feed external destinations (no way to encode this in an instruction), the signal inversion is invisible to programmers.

ASEL: A Source/Destination Control

The AMux drives the A input to the ALU, and is the data source for the read-field (RF←) and write-field (WF←) methods of loading ShC. The shifter also drives A, in which case the AMux is usually disabled.

A copy of the AMux drives the backplane Mar bus on processor memory references. The IFU may also drive Mar, when the processor isn't using it.

The three-bit ASEL field controls the source and destination for A as follows:

Table 7a: ASEL Decodes When FF is ok*

ASEL	FF[0:1]	Meaning
0	0	PreFetch←RM/STK
	1	Map←RM/STK (emulator or fault task) -or- IOFetch←RM (ic task)
	2	LongFetch←RM/STK
	3	Store←RM/STK
1	0	DummyRef←RM/STK
	1	Flush←RM/STK (emulator or fault task) -or- IOStore←RM (io task)
	2	IFetch←RM/STK
	3	Fetch←RM/STK
2	0	Store←Md
	1	Store←Id
	2	Store←Q
	3	Store←T
3	0	Fetch←Md
	1	Fetch←Id
	2	Fetch←Q
	3	Fetch←T
4	--	A←RM/STK
5	--	A←Id--see "Instruction Fetch Unit"
6	--	A←T
7	--	Shift operation--see "Shifter" (uses ALUF)

Table 7b: ASEL Decodes When FF is not ok*

ASEL	Meaning
0	Store←RM/STK
1	Fetch←RM/STK
2	Store←T
3	Fetch←T
4	A←RM/STK
5	A←Id
6	A←T
7	Shift operation--see "Shifter" (uses ALUF)

*FF is ok when not used in a long goto, long call, as a BSEL constant, or in an FF-controlled shift.

When FF is ok and ASEL = 0 to 3, the decoding of FF as a function is forced to be in the range 0 to 63. In other words, FF[0:1], stolen to modify the memory operation on A, do not participate in the FF decode. Hence, only functions 0 to 63 can be used in the same

instruction with a memory reference.

In the above tables, each instance where the source for A is RM/STK can be overruled by one of the 4 FF decodes for A sources or the FF decodes that put FF[4:7] on A. These FF decodes are illegal with the ASEL or ASEL-FF[0:1] values that select Id or T, and the source for A is undefined when this restriction is violated.

The notation "Fetch←A", "Store←A", etc. in the above table is compatible with the microlanguage. These routing expressions mean, for example, that the displacement originating on A is routed onto the Mar bus on the backplane, added to BR[MemBase] in the memory section and loaded into the memory address register. Then the Fetch, Store, etc. is started as detailed in "Memory Section".

ASEL does a pretty thorough job of encoding possible actions on A: Store← and Fetch← references take the address from RM/STK, T, Md, Id, or Q; other references take the address from RM/STK; LongFetch← takes the low 16 bits of address from RM/STK and high 8 bits from B.

The FF field can be used to select any of the following sources:

FF[4:7] (small constant)
RM/STK
Q
T
Md

These functions are illegal except on shifts (ASEL = 7) or when the source otherwise selected would be RM/STK (ASEL = 0, 1, or 4). On shifts these functions cause the A source to be wire-or'ed with the shifter output (otherwise the A source would be disabled); with references, these functions overrule RM/STK as the source.

Hardware Implementation

A is driven by a 4-input multiplexor as shown in Figure 3. A similar arrangement drives Mar, which is disabled except on memory references or when one of the 8 FF decodes that use Mar is executed; the IFU may use Mar when the processor does not. The 4-input multiplexors are usually disabled on shifts, which OR onto A independently.

However, the A multiplexor is *not* disabled when the source for A is encoded in FF, so it is possible to OR any A input except Id with the (complemented) shifter data--this is useful for BitBlt and other complicated uses of the shifter. Since shifter data on A is low-true, and since the normal ALU operation is NOT A on shifts, the effect of enabling both the shifter and the normal A multiplexor is [Shiftdata and not A].

ALUF, ALU Operations

The 4-bit ALUF field controls the ALU operation. It addresses a RAM (ALUFM) containing control for the MC10181 ALU chips.

ALUFM is 8-bits wide, of which 6 bits are used. ALUFM[0] controls the carry-in for arithmetic ALU operations. It is a "don't care" for the 16 logical ALU operations. The XorSavedCarry function causes the saved carry-out of a previous operation to be xor'ed with this bit. The XorCarry function complements the value from ALUFM. ALUFM[3:7] select the ALU function performed as below. The carry-out (task-specific) changes whenever an arithmetic operation is performed in the ALU unless explicitly disabled by the FreezeBC function (freeze branch conditions).

The Carry20 function forces the bit 12 carry-in to one. Assuming that this carry-in would otherwise have been zero, then this function adds 20_8 to the (arithmetic) ALU output. Adding 20_8 is expected to be useful because the cache, fast input bus, and fast output bus deal with 20_8-word munches.

The table below shows the logical and (useful) arithmetic ALU operations.

Table 8: ALUFM Control Values (Octal)

Logical		Arithmetic (No Carry)		Arithmetic (With Carry)	
*1	NOT A	*0	A	*0	A+1
3	(NOT A) OR (NOT B)	6	2*A	6	2*A+1
5	(NOT A) OR (B)	*14	A+B	*14	A+B+1
7	All-ones output	*22	A-B-1	*22	A-B
11	(NOT A) AND (NOT B)	*36	A-1	36	A
*13	NOT B				
15	A XNOR B (Assembler makes "EQV" and "=" synonyms for XNOR)				
17	A OR (NOT B)				
21	(NOT A) AND B				
*23	A XOR B (Assembler makes "#" synonym for XOR)				
*25	B				
*27	A OR B				
31	All-zeroes output				
33	A AND (NOT B)				
*35	A AND B				
37	A				

*System microcode can count on these operations being defined

On a barrel shift (selected by ASEL = 7), the first three ALUFM address bits are forced to 1 (ALUF[0:2] selects the kind of shift in this case). The intent of this arrangement is that ALUFM[16_8] selects the "NOT A" ALU operation. Nearly all shifter operations use this ALU function to route shifter output through the ALU. ALUFM[17_8] is loaded with assorted controls (i.e., used as a variable) by BitBlt or other opcodes that do more complicated things.

ALUFM can be read onto Pd by the ALUFMEM function, loaded from B by the ALUFMEM← function, or both loaded from B and read onto Pd by the ALUFMEMRW function.

External B sources from the IFU and internal sources are ready in time for arithmetic, but external sources from the memory and control sections are not (see the earlier section on "BSEL: B Multiplexor Select"). Internal A sources except shifter are ready in time for arithmetic. Unless explicitly disabled by the FreezeBC function, the branch conditions ALU<0, ALU = 0, Carry' (ALU carry out'), and Overflow are available for testing on the control card at t_3.

The Overflow branch condition, defined as carry-out from bit 0 unequal to carry-out from bit 1, is true iff a signed arithmetic operation yields an incorrect result.

Normally, the ALU is routed directly onto Pd, and Pd is then written into either T or RM/STK. However, several functions route ALU output shifted left or right 1 position onto Pd. The right shifts are:

ALU rsh 1	(0 onto Pd[0])
ALU rcy 1	(ALU[15] onto Pd[0])
ALU arsh 1	(ALU[0] onto Pd[0] preserving the sign)
ALU brsh 1	(ALUcarry onto Pd[0])
Multiply	(ALUcarry onto Pd[0]).

The left shifts are:

ALU lsh 1	(0 onto Pd[15])
ALU lcy 1	(ALU[0] onto Pd[15])
Divide	(Q[0] onto Pd[15])
CDivide	(Q[0] onto Pd[15]).

Multiply, Divide, and CDivide have other effects as well discussed later.

Note: The barrel shifter discussed in the "Shifter" section also use the Pd multiplexor for masking, so it is illegal to combine barrel shifts and ALU shifts in the same instruction.

Note: ALU<0, ALU = 0, Carry', and Overflow branch conditions test the ALU output of the *previous* instruction executed by the task and any shifting or masking that takes place in the Pd input multiplexor does *not* affect the result of these branch conditions.

Note: The value of Carry' and Overflow change only on *arithmetic* ALU operations. However, ALU←A operation may be either an arithmetic or a logical operation; in order to use XorCarry with ALU←A, we will probably use the arithmetic form of ALU←A, but the consequence of this is that Carry' will change on ALU←A. Programmers will have to be wary of this.

Note: Overflow is implemented correctly only for the A+B, A+B+1, A-B, and A-B-1 operations; other arithmetic ALU operations may modify the branch condition erroneously.

LC: Load Control for RM and T

This field controls the loading and source selection for the RM/STK memory and T register. The eight combinations are:

Table 9: LC Decodes

LC	Meaning
0	No Action
1	T←Pd
2	T←Md, RM/STK←Pd
3	T←Md
4	RM/STK←Md
5	T←Pd, RM/STK←Md
6	RM/STK←Pd
7	T←Pd, RM/STK←Pd

The only missing combination is T←Md, RM/STK←Md. T←Md, RM/STK←Md can be accomplished by combining an LC value of 5 with the TgetsMd FF decode. It is illegal to use TgetsMd with other LC decodes.

FF: Special Function

This field is the catch-all for functions not otherwise encoded in the instruction. For consistency with the hardware implementation, the 8-bit FF field is shown below as a two-bit field FA (= FF[0:1]) and two 3-bit fields, FB (= FF[2:4]) and FC (= FF[5:7]). Field values are given in octal.

The FF field is interpreted as a function iff:

(BSEL not selecting a constant) and
JCN does not select a "long" goto or call

When ASEL selects one of the memory references, the FF decode is forced to be that of FA = 0 because the FA field specifies the source for A or alternate memory reference in this case.

The decoding assignments have been made with the following considerations:

Functions that source the external BMux are grouped for easy decode of the signal that turns off the processor's B-multiplexors.

Operations that might be useful in conjunction with a memory reference are put in the first 64 decodes (FA = 0) since FA is decoded as zero on memory references.

Functions decoded by different hardware sections are arranged in groups to reduce decoding logic.

Table 10a: FF Decodes (FA = 0)

FB	FC	Function
0-1	--	A[12:15] ← FF[4:7]
* The 4 A←xx decodes below do not disable the AMux, if ASEL selects a shift		
2	0	A ← RM/STK
2	1	A ← T
2	2	A ← Md
2	3	A ← Q
2	4	XorCarry (complements ALUFM carry bit)
2	5	XorSavedCarry
2	6	Carry20 (carry-in to bit 11 of ALU = 1)
2	7	ModStkPBeforeW (Use modified StkP for write address of STK)
3	0	--
3	1	ReadMap. Modifies action of Map← (see "Memory Section")
3	2	Pd ← Input (checks for IOB PE)
3	3	Pd ← InputNoPE (no check for IOB PE)
3	4	RisId (causes Id to replace RM/STK in A←RM/STK, B←RM/STK, and shifter)
3	5	TisId (causes Id to replace T in A←T, B←T, and shifter)
3	6	Output ← B
3	7	FlipMemBase (MemBase ← MemBase xor 1)
4-5	--	Replace RMaddr[0:3] by RBase[0:3] and RMaddr[4:7] by FF[4:7] for write of RM; Forces RM to be written even if STK was read.
6	0-7	Branch conditions (see "Control"). In conjunction with an IFU jump in JCN, if the condition is true, IFU advance is disabled (see "IFU")
7	0	BigBDispatch ← B (256-way dispatch on B[8:15]. See "Control")
7	1	BDispatch ← B (8-way dispatch on B[13:15]. See "Control")
7	2	Multiply (Pd[0:15] ← ALUcarry,,ALU[0:14]; Q[0:15] ← ALU[15],,Q[0:14]; Q[14] OR'ed into TNIA[10] as slow branch--see "Multiply")
7	3	Q ← B
7	4	--
7	5	TgetsMd (In conjunction with LC=5, this causes T←Md, RM/STK←Md)
7	6	FreezeBC (freezes previous values of ALU and IOAtten' BC's for 1 cycle)
7	7	Reserved as a no-op

Table 10b: FF Decodes (FA = 1)

FB	FC	Action
0	0	PCF ← B. Load PCF and starts fetching instructions
0	1	IFUTest ← B, dismisses junk wakeup, bits used as follows: 0:7 TestFG 8 TestParity 9 TestFault 10 TestMemAck 11 TestMakeF←D 12 TestFH' 13 TestSH' 14 enables testing 15 Disables junk wakeups
0	2	IFUTick
0	3	RescheduleNow
0	4	--
0	5	MemBase←B[3:7]
0	6	RBase←B[12:15]
0	7	Pointers←B (MemBase←B[3:7] and RBase←B[12:15])
1	0:7	--

Table 10c: FF Decodes (FA = 1)

FB	FC	Action
*The following 8 FF decodes drive Mar from A.		
2	0-1	--
2	2	CFlags ← A' (see Figure 10) (Mar must be stable during prev. instr.)
2	3	BrLo ← A. BR[16:31] ← A[0:15]
2	4	BrHi ← A. BR[4:15] ← A[4:15]
2	5	LoadTestSyndrome from DBuf (see Figure 10)
2	6	LoadMcr[A.B] (see Figure 10)
2	7	ProcSRN ← B[12:15]
3	0	InsSetorEvent ← B. If B[0] = 0, then B[4:15] are controls for EventCntA and EventCntB; if B[0] = 1, then B[6:7] are loaded intothe IFU's InsSet register.
3	1	EventCntB ← B or equivalently GenOut←B (General output to printer, etc.)
3	2	Reschedule
3	3	NoReschedule
*B data must setup during previous instruction and not glitch when writing IFUMLH/RH--see IFU section.		
3	4	IFUMRH ← B. Packeda←B.5, IFaddr'←B[6:15]
3	5	IFUMLH ← B. Sign←B.0, PE[0:2]←B[1:3], Length'←B[4:5], RBaseB'←B.6, MemB←B[7:9], TPause'←B.10, TJump←B.11, N←B[12:15]
3	6	IFUReset. Reset IFU
3	7	BrkIns ← B. Opcode←B[0:7] and set BrkPending
4	0	UseDMD (see "Control Section")
4	1	MidasStrobe ← B (see "Control Section")
4	2	TaskingOff
4	3	TaskingOn
4	4	StkP ← B[8:15]
4	5	RestoreStkP
4	6	Cnt ← B (overrules Cnt=0&-1 in the same instruction)
4	7	Link ← B
5	0	Q lsh 1 (Q[0:14] ← Q[1:15], Q[15] ← 0)
5	1	Q rsh 1 (Q[1:15] ← Q[0:14], Q[0] ← 0)
5	2	TIOA[0:7] ← B[0:7] (Note: loaded from *left-half* of B)
5	3	--
5	4	Hold&TaskSim ← B (Hold reg ← B[0:7], Task reg ← B[9:15]. See "HOLD and Task Simulator")
5	5	WF ← A (load ShC with write-field controls--see "Shifter")
5	6	RF ← A (load ShC with read-field controls--see "Shifter")
5	7	ShC ← B (see "Shifter")
6	0	B ← FaultInfo'. B[8:11]←SRN for 1st fault, B[12:15]←number of faults
6	1	B ← Pipe0 (B←VaHi--see Figure 10)
6	2	B ← Pipe1 (B←VaLo--see Figure 10)
6	3	B ← Pipe2' (see Figure 10)
6	4	B ← Pipe3' (B←Map'--see Figure 10)
6	5	B ← Pipe4' (B←Errors'--see Figure 10)
6	6	B ← Config' (see Figure 10)
6	7	B ← Pipe5' (see Figure 10)
7	0	B ← PCX'
7	1	B ← EventCntA' (see "Performance Issues")
7	2	B ← IFUMRH' (low part of IFUM)
7	3	B ← IFUMLH' (high part of IFUM)
7	4	B ← EventCntB' (see "Performance Issues")
7	5	B ← DBuf (normally non-task-specific data from last Store← -- see "Memory")
7	6	B ← RWCPReg (= Link←B' and B←CPReg)
7	7	B ← Link

Table 10d: FF Decodes (FA = 2)

FB	FC	Action
0-1	--	RBase ← FF[4:7]
2-3	--	Replace RMaddr[0:3] by FF[4:7] for write of RM. Forces RM to be written even if STK was read.
4	--	TIOA[5:7] ← FF[5:7] (TIOA[0:4] unchanged)
5	0-3	MemBaseX ← FF[6:7] (MemBase[0] ← 0, MemBase[1:2] ← MemBX[0:1], MemBase[3:4] ← FF[6:7])
5	4-7	MemBX ← FF[6:7]
6	0-1	--
6	2	Pd ← ALUFMRW (Pd ← ALUFMEM as below, ALUFMEM ← B.8, B[11:15])
6	3	Pd ← ALUFMEM (Pd.0 ← DMux data, Pd.8 and Pd[11:15] ← ALUFMEM[ALUF])
6	4	Pd ← Cnt
6	5	Pd ← Pointers (Pd[1:2] ← MemBX, Pd[3:7] ← MemBase, Pd[8] ← StkOvf, Pd[9] ← StkUnd), Pd[12:15] ← RBase
6	6	Pd ← TIOA&StkP (Pd[0:7]←TIOA, Pd[8:15]←StkP)
6	7	Pd ← ShC
7	0	Pd ← ALU rsh 1 (Pd[0] ← 0)
7	1	Pd ← ALU rcy 1 (Pd[0] ← ALU[15])
7	2	Pd ← ALU brsh 1 (Pd[0] ← ALUcarry)
7	3	Pd ← ALU arsh 1 (Pd[0] ← ALU[0] preserving sign)
7	4	Pd ← ALU lsh 1
7	5	Pd ← ALU lcy 1
7	6	Divide (Pd[0:15]←ALU[1:15],,Q[0]; Q[0:15]←Q[1:15],,ALUcarry)
7	7	CDivide (Pd[0:15]←ALU[1:15],,Q[0]; Q[0:15]←Q[1:15],,ALUcarry')

Table 10e: FF Decodes (FA = 3)

0-3	--	MemBase ← FF[3:7]
4-5	--	Cnt ← small constant (Cnt[0:10] ← 0, Cnt[11] ← 0 if FF[4:7] # 0 else 1, Cnt[12:15] ← FF[4:7]; i.e., values of 1 to 16 are loadable)
6-7	--	Wakeup[n] -- Initiate wakeup request for task FF[4:7]

Multiply and Divide

The Multiply, Divide, and CDivide functions operate on unsigned 16-bit operands. Unsigned rather than signed operands are used so that the algorithms will work properly on the extra words of multiple-precision numbers.

The actions caused by these functions are as follows:

Multiply:

```
Result ← ALUCarry..ALU/2
Q ← ALU[15]..Q/2
Next branch address ← whatever it is OR 2 if Q[14] is 1.
```

Divide, CDivide:

```
Result ← 2*ALU..Q[00]
Q ← 2*Q..ALUCarry' -or- 2*Q..ALUCarry
```

Complete examples for Multiply and Divide subroutines are given in the microassembler document. The inner loop time is 1 cycle/bit for multiply and 2 cycles/bit for divide.

Shifter

See Figure 4.

Dorado contains a 32-bit barrel shifter and associated logic optimized for field extraction, field insertion and the BitBlt instruction.

The shifter is controlled by a 16-bit register ShC. To perform a shift operation, ShC is loaded in one of three ways discussed below with 14 bits of control information, and one of eight shift-and-mask operations is then executed in a subsequent instruction. Alternatively, (a limited selection of) shift controls may be specified in FF and BSEL concurrent with a shift; in this case, ShC is not modified. ASEL = 7 causes a shift and ALUF[0:2] select the kind of masking.

The execution of a shift instruction (after ShC has been loaded in a previous instruction) proceeds as follows:

ShC[2] selects between T and RM/STK for the left-most 16 bits input to the shifter; ShC[3] selects between T and RM/STK for the right-most 16 bits. Using the RisId or TisId FF decode in the same instruction allows Id to replace either T or RM/STK in the shift. This 32-bit quantity is then left-cycled by the number of positions (0-15) given by ShC[4:7]. When ShC[2] and ShC[3] are both 1, then the shifter left-cycles T; when both 0, RM/STK. In these cases it operates as a 16-bit cycler. When ShC[2] and ShC[3] are loaded with complementary values, then it left-cycles the 32-bit quantity R..T or T..R.

The low order 16 bits of shifted data are placed *complemented* on A by the shift, and normal A source is disabled (except when the source for A is encoded in FF--see the ASEL section).

ALUF[0:2] select one of eight mask operations (see below) and the first three

ALUFM address bits are forced to 1, so that the ALU operation in either ALUFM 14 or ALUFM 15 can be performed. This must be a logical ALU operation using the shifted data on A and data on B because there is insufficient time to propagate carries for an arithmetic operation. The intent is that ALUFM 14 contain the control for the "NOT A" ALU operation normally desired, while ALUFM 15 is used by BitBlt and other opcodes that need computed ALU operations.

ALU output passes to the masking logic. The mask operation determines which of two independent masks in ShC are applied to the data. LMask contains 0 to 15 ones starting at bit 0, RMask 0 to 15 ones starting at bit 15. The masked area(s) of ALU output are replaced either with zeroes or with corresponding bits from Md according to the shift-and-mask function selected. Replace-with-Md generates HOLD if Md isn't ready yet, and the timing for this is the same as Md onto B (i.e., data is never ready sooner than the second instruction after the Fetch←).

Masked data is routed onto Pd, then sent to the destination specified by LC.

Note: The Pd input multiplexor is used to carry out masking, so it is illegal to combine a shifter operation with an ALU shift in the same instruction.

Three functions load ShC: RF←A and WF←A treat A[8:15] as a Mesa field descriptor and transform the bits appropriately before loading ShC. ShC←B allows an arbitrary value to be placed in ShC (used by BitBlt).

The shift controls come directly from FF if ASEL=7 (a shift) and if BSEL = 4, 5, 6, or 7, selecting a constant. This specifies complete shift control in the instruction which does the shift, so ShC doesn't have to be loaded in a previous instruction, and ShC isn't clobbered, so io tasks don't have to save and restore it. When BSEL controls a shift in this way, the B source is forced to be Q.

The mask operations are as follows:

Table 11: ALUF Shift Decodes

ALUF[0:2]*	
0	ShiftNoMask
1	ShiftLMask--masked bits on the left-hand-side of the word replaced with 0's
2	ShiftRMask--masked bits on the right with 0's
3	ShiftBothMasks--masked bits on both sides replaced with 0's
4	ShMdNoMask--unused (falls out of decoding)
5	ShMdLMask--masked bits replaced with Md
6	ShMdRMask--masked bits replaced with Md
7	ShMdBothMasks--masked bits replaced with Md

*ALUF[3] selects the ALU operation in either ALUFM 14 or 15

ShiftLMask implements right shift and load-field operations; ShiftRMask implements left shift; ShiftBothMasks deposits the selected field into a word of zeroes; ShMdBothMasks deposits the selected field into data coming from memory; and ShiftNoMask implements various cycle operations.

Note: On a shift the ALU branch conditions apply to the *unmasked* ALU output.

The microcode for the Mesa RF (Read Field) and WF (Write Field) instruction is shown as an example of the use of the shifter. RF and WF both take a pointer from the top of the stack and add α to it as a displacement. RF fetches the word, and pushes the field specified by β onto the stack; WF fetches the word, and inserts a field from the rightmost bits of the word in the second position of the stack into it, then restores the word to memory.

```
RF:   IFetch←Stack, TisId;          *Calculate the pointer.  α replaces BR[MemBase] (MDS);
                                    *this value is then added to Stack to compute the
                                    *address for the pointer.
      Stack←Md, RF←Id;              *IFU supplies β, the field descriptor
      IFUJump[0], Stack←ShiftLMask; *Right-justify & mask the field, IFU to next instruction

WF:   IFetch←Stack&-1, TisId;       *Start fetch of word containing field
      WF←Id;                        *IFU supplies β, the field descriptor
      T←ShMdBothMasks[Stack&-1];
      IFUJump[0], Store←Rtemp, Md←T;
```

Hold and Task Simulator

Hold&TaskSim←B loads HOLDSIM[0:7] from B[8:14]..0 and TASKSIM[0:6] from B[1:7]. HOLDSIM is a recirculating shift register in which the presence of a 1 in bit 7 causes HOLD two instructions later. For example, Hold&TaskSim←200_8 will complete three instructions after the Hold&TaskSim←, HOLD the next cycle, and HOLD every seventh instruction (i.e., every eighth cycle) thereafter. Since this register cannot be loaded with all 1's, HOLD of infinite duration is impossible.

To disable this debugging feature, the register must be loaded with 0.

TASKSIM is a seven-bit counter which determines the number of cycles before a task wakeup occurs. The task selected for wakeup must be jumpered on the backplane (else no-op). Whenever TASKSIM is loaded with a non-zero value, it counts up to 177_8, then generates a wakeup request when the counter overflows to 200_8. The wakeup request remains true until TASKSIM is reloaded.

Control Section

The control section interfaces the mainframe to the baseboard microcomputer or Alto which controls it as detailed in the "Dorado Debugging Interface" document. In addition, the control section stores instructions in 4k x 34-bit (+2 parity) IM ("instruction memory") and contains logic for sequencing through instructions and switching among tasks.

The current instruction is clocked into the MIR register at t_0 and exported to the processor, memory, and IFU sections for decoding. The control section itself decodes the JCN field, the BLOCK bit, and its own FF decodes (Wakeup, B←Link, B←RWCPReg, Link←B, TaskingOn, TaskingOff, BDispatch←B, BigBDispatch←B, Multiply, MidasStrobe←B, UseDMD, and branch conditions).

The control section also exports the task number via the Next bus, which somewhat after t_{-2} contains the task number that will execute an instruction at t_0.

Figure 5 shows the overall organization of the control section. Figure 6 shows how branch control is encoded in JCN. Figure 7 shows the timing for regular instructions and for the multi-cycle TPC and IM read/write instructions.

Tasks

Dorado provides sixteen independent priority-scheduled tasks at the microcode level. Task 15 is highest priority, task 0 lowest. Task 15 (the "fault task") is woken by StkError and by memory map and data error faults. Tasks 1-14 provide processing functions for io controllers implemented partially in hardware, partially in firmware; the present assignment of these tasks to device controllers is given in the "Slow IO" chapter. Task 0 (the "emulator") implements instruction sets (Mesa, Alto, etc.). In the absence of io activity, task 0 (always awake) controls the processor.

Essentially, io devices are paired to tasks when built, and a device controller can assert a wakeup request for the task with which it is paired. A program cannot modify the assignment of controllers to tasks (although the hardware change for this is easy). Additional flexibility in this area is not thought to be worth additional hardware cost.

Each task has its own program counter and subroutine return link, stored in the (task-specific) TPC and TLINK registers when the task is inactive. TPC may also be treated as a memory, so program counters for tasks other than the current task can be read and written by a program. This is discussed later in this chapter.

Task Switching

When device hardware requires service from a task, it activates its wakeup request line at t_0. Wakeup requests are priority-encoded, and the highest priority request (BNT or "Best Next Task") is clocked at t_2 and competes with the current task (CTASK) for control of the machine. If BNT is higher priority than CTASK, or if the current (non-emulator) instruction has BLOCK = 1, a task switch will take place; in this case, CTASK will be loaded from BNT at t_4. This implies that the shortest delay from a wakeup request to the first instruction of

the associated task is two cycles.

The Wakeup[task] functions allow any task to be woken, just as though a hardware device had activated its wakeup line. The task responding to a Wakeup must not block sooner than the second instruction, or it will get reawakened. A minimum of two cycles elapses after the instruction containing Wakeup before the task executes its first instruction.

When a task has been woken by Wakeup[task] or has executed one or more instructions and then deferred to a higher priority task, the fact that it is runnable is remembered in a Ready flipflop. The Ready flipflop is cleared only when the associated task blocks. In other words, there is no way to deactivate a task, after its ready flipflop has been set, except by forcing it to execute an instruction that blocks.

> The baseboard and Alto controllers may also clear the Ready flipflops by another mechanism, discussed in "Dorado Debugging Interface".
>
> The emulator has no Ready flipflop and cannot block; the BLOCK bit in the instruction is interpreted as StackSelect for the emulator.

Task switching may occur after every instruction unless explicitly disabled by the TaskingOff function. The TaskingOn function reverses the effect of TaskingOff.

> It would be a programming error for a task to block with tasking off. Any erroneous attempt to block would fail, and it would continue execution.

It is illegal for a task to block in an instruction that might be held, if the wakeup line for the task might be dropped at t_0 of the instruction. If this occurred, the instruction might inadvertently be repeated before the block occurred.

Remark

Multiple tasks seem better than a more conventional priority interrupt system because interference by input/output tasks is substantially reduced. As to the exact implementation, variations are possible. The current scheme requires more hardware than one in which the program explicitly indicates when a task switch is legal (as on Alto and D0). However, because Hold may last for about 30 cycles, a reliance upon explicit tasking would result in inadequate service for high priority tasks.

Next Address Generation

> *This section gives a low-level view of jump control. Because the microassembler and loader handle details of instruction placement automatically, programmers need not struggle with the encodings directly. For this reason, programmers may wish to skim this section while concentrating on high-level jump concepts described in "Dorado Microassembler".*

Read this with Figure 6 in front of you.

For the most part, instruction memory (IM) addressing paths are 16 bits wide, although only 12 bits are presently used; the extra width allows for future expansion to 13 or 14 bits, when sufficiently fast 4kx1 ECL RAMS are economically available; there are no plans to utilize the remaining 2 bits, but since nearly all hardware components in the control data paths are packaged 4/can, the extra two bits are almost free. Also, the 16-bit wide Link register can be used to hold full word data items.

The various registers and data paths that contain IM addresses are numbered 0:15, where bits 4:15 are significant for the 4k-word microstore, while the quadrant bits 2:3 are ignored. This numbering conveniently word-aligns the bits while also allowing for future expansion. The discussion below assumes a 4k-word microstore.

Dorado does not have an incrementing instruction-address counter. Instead, the address of the next instruction is determined by modifying the current instruction address (CIA) in various ways. The Tentative Next Instruction Address (TNIA) is determined from JCN[0:7] in the instruction according to rules in Figure 6. TNIA addresses IM for the fetch of the next instruction unless a task switch occurs. If SWITCH occurs, the program counter for the highest priority competing task (BNPC or "Best Next PC") addresses IM.

An IM quadrant is viewed as containing 64 pages of 64 instructions. Values in JCN are provided for the following kinds of branches:

Local branches to any of the 64 locations in the current page;

Global branches to location 0 on any of the 64 pages;

Long branches to any location in the quadrant using the 8-bit FF field to extend JCN (normal interpretation of FF is disabled);

Conditional branches to any of 14 even locations in the current page, if the selected condition is false, or to the adjacent odd location, if the condition is true (7 branch conditions are available);

IFU jumps to a starting address supplied by the IFU; JCN selects any one of up to 4 entries in the starting address vector (This is motivated by an entry-vector scheme discussed in "Instruction Fetch Unit".);

read/write IM and read/write TPC, after which execution continues at .+1;

Return to the address in Link;

Branch conditions may also be specified in FF, as discussed below. Several dispatches may also be specified in FF. These 'OR' bits into the branch address computed by the *following* **instruction.**

If IM is expanded to 16k words, branching from one quadrant to another will only be possible by loading the Link register with a 14-bit address and then returning; jumps, calls, and IFUJumps will be confined to the current 4k-word IM quadrant.

Remarks on JCN Encoding

JCN cleverly encodes in 8 bits almost as much programming flexibility as would be possible with an arbitrarily large and general field. The main disadvantage is that MicroD is needed to postprocess assemblies and place **instructions.**

The earliest prototype of Dorado used a 7-bit JCN encoding that had fewer global and conditional branch targets, so programming was harder and additional instructions had to be inserted in a few places. This was slightly worse than the 8-bit encoding, but it would have been feasible to stay with the 7-bit encoding and employ the bit thus saved for some other use in the instruction.

Local, global, and long branches are analogous, respectively, to local, page-zero, and indirect branches used on many minicomputers. However, Dorado scatters its global locations over the microstore rather than concentrating them in page-zero; this is better than the minicomputer scheme for the following reason. During instruction placement, when a cluster of instructions is too large to fit on one page, a global allows it to be divided between

two pages; but if all globals were in page zero, then page zero itself would quickly fill up. In other words, dispersing the globals is theoretically more powerful than concentrating them in page zero; because MicroD does all the tedious work of placing instructions, this theoretical advantage is made practical; minicomputers have not employed any program like MicroD, so they have used the less powerful but simpler page-zero scheme.

Local branches on Dorado are within a 64-word page, where minicomputers usually branch relative to the current PC. Relative branching is probably more powerful, but it cannot be used on Dorado because of insufficient time for addition.

Long branches on Dorado use 4 bits of JCN in conjunction with the 8-bit FF field to specify any location in the 4k-word quadrant. Since BSEL never selects a constant in this case, an improvement on our scheme would have used 3 bits of JCN in conjunction with BSEL.0 and the 8-bit FF field; this would have freed 8 values of JCN to encode some other kind of branch.

Conditional Branches

IM is organized in two banks, with odd addresses in one bank, even in the other. The address is needed shortly after t_0, but the bank-select signal not until 15 ηs after the address. For this reason conditional branches select between an even-odd pair of instructions (i.e., between the two banks) according to branch conditions that need not be stable until a little after t_1.

Alternatively, a conditional branch may be encoded in FF in conjunction with any addressing mode except a long branch in JCN. When this is done, the result of the branch test is ORed with TNIA[15].

This implies that for both FF-encoded and JCN-encoded branch conditions, the false target address is even and the true target is odd.

Hence, it is possible to conditionally branch using only JCN, while using FF for an unrelated function, or to encode a branch condition in FF while using any addressing mode in JCN. If branch conditions are encoded in both FF and JCN, the branch test results are OR'ed, providing further flexibility.

The branch condition encodings are:

Table 12: Branch Conditions

JCN[5:7]	FF	Branch Condition
0	60	ALU = 0
1	61	ALU<0
2	62	ALUcarry'
3	63	Cnt = 0&-1 (decrements count *after* testing)
4	64	R<0
5	65	R ODD
6	66	IOAtten' (non-emulator) or ReSchedule (emulator)
--	67	Overflow'

ALU = 0 and ALU<0 are the results of the last ALU operation executed by the current task. ALUcarry' (the saved carry-out of the ALU) and Overflow are the result of the last *arithmetic* ALU operation executed by the current task (ALU←A may be stored in ALUFM as either an arithmetic or logical operation, so programmers should be wary of smashing these branch conditions when ALU←A is used.). These are saved in a RAM and may be frozen by the FreezeBC function for one cycle. In other words, the branch conditions are ordinarily loaded into the RAM at t_3, but if FreezeBC is present, then the RAM is not loaded and values from the previous instruction for the same task will apply.

The IOAtten' branch condition tests the task-specific IOAttention signal optionally generated by the io device associated with the current (non-emulator) task.

Remark on Target Pairs

The bank-select toggling trick, which allows branch conditions to be developed very late, is valuable. Without this trick, it would be necessary to choose between slowing the instruction cycle or restricting branch conditions to signals stable at t_0. Neither of these alternatives is palatable.

A more traditional implementation of conditional branches would go to the branch address, if a condition were true, or fall through to the instruction at .+1, if it were false. This traditional scheme is never faster but is sometimes more space-efficient than the target-pair scheme because the target-pair requires a duplicated instruction for every instance of a conditional branch to a single target, which is fairly common. The traditional scheme does not allow DblGoto and DblCall constructs discussed in "Dorado Microassembler," but these are infrequent.

Subroutines and the Link Register

Dorado provides single-level subroutines by means of the (task-specific) Link register. A Call occurs on any instruction whose destination address is 0 mod 16 before any modification of TNIA due to branch conditions or dispatches. On a Call, Return, or IFUJump, Link is loaded with CIA+1.

> Because Return loads Link with CIA+1, CoReturn constructs are possible. Because IFUJump also loads Link with CIA+1, the conditional exit feature discussed in the "Instruction Fetch Unit" chapter is possible.

CIA+1 is used rather loosely in discussion here; the actual value loaded into Link by a call or return is [(CIA & 177700_8) + ((CIA+1) & 77_8)]. In other words, a call or return in location 77_8 of any page loads Link with location 0 of that page.

Link may be loaded and read by programs, so deeper subroutine nesting is possible, if Link is saved/restored across calls.

> The functions Link←B and B←RWCPReg and the B dispatch functions discussed below, all of which load Link from B, overrule a call. In other words, if there are conflicting reasons for loading Link, Link←B wins over Link←CIA+1.
>
> The B←RWCPReg function (= Link←B, B←CPReg') is provided primarily for initialization from the baseboard computer and for use by the Midas debugging program.

Remark on Stack vs. Task-Specific Link Register

An alternate implementation would replace the task-specific Link register with a 16-word stack, using about the same quantity of hardware as the current scheme. This would have the following implications: All tasks would use the multi-level stack, so saving and restoring Link across nested procedure calls would be unnecessary; maximum subroutine nesting by all tasks could not exceed 16. The emulator could ignore the distinction

between call and jump at the top level, as is the case with the task-specific Link register. However, tasks other than the emulator would have to preserve the stack, so they could use only jump locations at the top level and could not Block inside subroutines.

In other words, the stack scheme would permit nested subroutine calls at the expense of more restrictive instruction placement and a caveat against blocking inside subroutines for task 1 to 15 microcode. Nested subroutines seem uncommon in emulator microcode, so it is not obvious that this revised scheme would be better than the one we have chosen.

Remark on Call/Jump

Deciding between *call* and *jump* based on target address saves one bit in the instruction and costs little for the following reasons. Instructions can be divided into three groups: those always jumped to, those always called, those for which Link can be smashed (i.e., "don't care" about call or jump), and those both jumped to and called.

A realistic guess is that over half of all instructions will be "don't care"; namely, these will be executed at the top level, not inside a subroutine, and the Link register will not contain anything of importance. Assembly language declarations make this information available to MicroD.

The hardware makes 1/16 of the locations in each page "call locations". It is estimated that this is somewhat more than real programs will need, on the average (although we vacillated about whether 1/8 or 1/16 of the targets should be calls).

In each page, MicroD first places instructions that must be called or must be jumped to. Because there are so many "don't care" instructions, it is unlikely that either call or jump slots in a page will be exceeded. Consequently, it will nearly always be possible to complete allocation of the call and jump targets without overflowing due to the call/jump restriction. After this "don't care" instructions fill in the remaining slots.

The remaining situation, with which Dorado cannot cope, is an instruction both called and jumped to. This would arise in a subroutine whose entry instruction closed a loop (uncommon). On Dorado, this situation requires duplicating the entry instruction, so it costs one location but no extra time.

Dispatches

Several FF decodes are *dispatches* which OR various bits with TNIA[8:15] during the *following* instruction. The dispatch bits must be stable by t_2.

Dispatches are:

BigBDispatch←B	B[8:15] (256-way dispatch)
BDispatch←B	B[13:15] (8-way dispatch)
Multiply	OR's Q[14] into TNIA[14]

The two B dispatches load Link register from B, then OR appropriate bits of Link into TNIA during the next instruction for the task. Since TLINK is task-specific, this works correctly across task switching. The Q-bit is only loaded during a multiply, and is illegal for tasks other than the emulator.

The decision between call and jump in the instruction after a dispatch is unaffected by dispatch bits--it depends only upon JCN. In other words, the instruction following a dispatch is a Call if its unmodified target address is 0 mod 16, else a jump.

It is possible to neutralize any bits in a dispatch by placing target instructions at locations with 1's in the neutralized bits. In other words, a dispatch on B[8:10] could be accomplished by locating the 8 target instructions at IM locations whose low five address bits were 1, e.g. at 37_8, 77_8, 137_8, 177_8, 237_8, 277_8, 337_8, and 377_8, and by branching to 37_8 in the instruction after the BigBDispatch←B.

Note: Methods discussed later for resuming a program interrupted by a page fault do not permit continuation when a fault occurs between a dispatch and the following instruction; for this reason, programmers should ensure that no fault can possibly occur by holding for memory faults with ←Md prior to or concurrent with the dispatch; also, stack operations that might overflow/underflow may not be used in the instruction after a dispatch.

Note: When the PC for another task is loaded using the LdTPC← operation discussed later, any pending dispatch conditions for that task are cleared. The debugging program Midas does not clear pending dispatches, however, so it should be ok to put a breakpoint on the instruction after a dispatch or to single-step through a dispatch.

IFU Addressing

The IFU supplies ten bits of opcode starting address to the processor. During the last instruction of every opcode, exit to the next opcode is accomplished by IFUJump[n] (n = 0 to 3) which selects among four entry locations for the next opcode. The starting address supplied by the IFU is used for TNIA[4:13] and TNIA[14:15] are set to n. If the IFU is unprepared, it supplies a trap address instead of a starting address, and control goes to the nth location in a trap vector.

IFUJump's always load Link with CIA + 1. This is necessary to implement the following conditional exit feature for opcodes.

If an FF-encoded branch condition is true in the same instruction as an IFUJump, IFU advance to the next opcode is disabled. This kludge allows an opcode with common and uncommon exit conditions to finish, for example, with IFUJump[2,condition]. If the condition is false (common case), then the IFU advances normally to the next opcode, starting at location 2 of the entry vector. Otherwise (uncommon case), control continues at location 3 of the entry vector, but the IFU does not advance, so emulation of the current opcode can continue.

Utilization of IFUJump and conditional IFUJump is discussed in "Instruction Fetch Unit."

IFU trap addresses and other reserved locations in the microstore are as follows:

Table 13: Reserved Locations in the Microstore

Reason	*Locations*	*Comment*
Reschedule request	*14-17	Indicates that some previous instruction executed the ReSchedule function.
IFUM parity error	*74-77	Indicates a hardware failure in the IFUM storage.
IFU not ready	*34-37	The instructions in this vector should contain IFUJump[n], waiting for the IFU to become ready.
IFU data parity error	*4-7	Parity wrong on data from cache.
IFU map fault	*0-3	The IFU buffers the fact of a map fault and completes all opcodes in the pipe ahead of the one experiencing the fault. Upon dispatch to the first instruction for the opcode affected by the fault, this trap occurs.
Midas Call command	7776	
Midas Crash detect	7777	

*Ifu traps OR the 1's complement of the instruction set into bits 8:9 of the trap address, so actual trap locations for Reschedule, for example, are 14-17, 114-117, 214-217, and 314-317. The trap vector is 1 to 4 instructions long according to the IFUJump programming convention, as discussed in the "Instruction Fetch Unit" chapter.

IM and TPC Access

See figures 6 and 7.

IM is read and written by programs using a special decode of JCN in conjunction with the RSTK field of the instruction; TPC is also read and written using a special JCN decode. *TaskingOff must be in force, and anything that might cause hold is illegal in the same instruction; hold is also illegal in the instruction after an IM or TPC read, when the data is accessed using B←Link.*

After the read or write instruction, control passes to the next sequential instruction, i.e., to CIA+1 (with wrap-around at 64-word page boundaries). CIA+1 also winds up in Link.

Note: The hardware does not actually load Link with the IM or TPC data; instead B←Link in the next cycle routes inverted data onto B using an alternate path. The Link register itself is smashed with CIA+1 as discussed above, and this value would be read (assuming it wasn't overwritten) in later instructions.

This implies that continuation from a breakpoint or program-interrupt halt on the instruction following an IM or TPC read (i.e., on the B←Link instruction) won't work correctly.

Total time for an IM or TPC read or write operation is 6 clocks (i.e., thrice as long as a normal instruction).

A 34 (+2 parity)-bit IM word is read as four 9-bit quantities. The read address is taken from Link. *Data must be read from Link[7:15] in the instruction immediately after the IM read;* this data is inverted; Link[0:6] contain 1's, so that when the entire word is 1's complemented the desired data will have leading 0's. The byte select is RSTK[2:3].

IM writes also take the write address from Link, 16 bits of data from B and 2 bits from RSTK; the half-word affected is also specified in RSTK.

Any task can read or write TPC for an arbitrary task other than itself (an attempt to set TPC of the running task is unpredictable). The task number is B[12:15], and data is taken from or written into Link. The assembly language notations for these are RdTPC←task and LdTPC←task. After RdTPC←B, the 16 bits of data in Link are 1's complemented.

Note: The dispatch-pending conditions for a task whose TPC is loaded by LdTPC← are cleared, so LdTPC← works even when that task has just executed a BDispatch←B or BigBDispatch←B.

Hold

Many events in the memory system, StkError and the hold simulator in the processor, and several IFU error conditions generate hold (The IFU error conditions cause a one-cycle hold iff an IFUJump occurs on the first cycle of the error.). The control section itself forces hold when a task switch occurs concurrent with TaskingOff. This signal, clocked at t_1, occurs when the current instruction cannot be completed. Its effect on the hardware is to suspend the current instruction, while completing parts of the previous instruction that have been pipelined into the current cycle. Approximately, it converts the current instruction into a Goto[.] while preserving branch conditions and some other stuff.

Higher priority tasks are *not* prevented from running when the current task is experiencing **Hold.**

Remark

The fact that the address of the next instruction is needed at t_0, while Hold is not generated until t_1 means that concurrence of Hold and BLOCK with a switch to a lower priority task produces an anomalous situation called "Next Lies". The hardware disables clocks to CIA, TPC, and MIR when this occurs, so that the current instruction is repeated. This results in some hardware complications discussed in the "Slow IO" chapter, but programmers need not worry about it.

Program Control of the DMux

Dorado contains a large number of multiplexors called mufflers which allow a selected signal from a set of up to 2048 signals to be observed on a data bus called the DMux. This provides a passive method by which the Baseboard section or the external Midas debugger can examine internal control signals and registers not otherwise observable.

The particular DMux signal is selected by shifting in an 11-bit address one bit at-a-time. Each board with mufflers contains a 12-bit address register that responds to the shifted address bits; the highest bit is ignored for the purposes of selecting the signal to be read. "Dorado Debugging Interface" discusses a clever generator algorithm that allows all 2048 signals to be read into a table in 2048 + 11 shift-read cycles.

In addition, the DMux address can also be executed as a control function. In this case the full 12-bit address determines what function is executed. This "manifold" mechanism is used to control power supplies, set clock rate, enable/disable error halt conditions, and test IM without involving other hardware.

The DMux facility can also be controlled directly by Dorado programs by means of the MidasStrobe←B and UseDMD functions. Essentially, the DMux address mechanism is controlled externally by the Baseboard or by Midas operating through the Baseboard when Dorado isn't running, and by Dorado when Dorado is running.

The MidasStrobe←B function causes B[4] to be shifted out as an address bit. This takes three cycles, so the program must execute three more instructions before doing another MidasStrobe←B function. The DMux signal selected by the last 11 address bits shifted out is read on B[0] when the Pd←ALUFMEM function is executed.

The UseDMD function causes the current DMux address to be executed as a manifold operation.

The following subroutine reads the DMux signal selected by the address in T:

```
Subroutine;
ReadDMux:
        Cnt←10S;
RdDMuxLp:
        MidasStrobe←T;          *Shift out address in T[4]
        Noop;
        Noop;
        T←(T) lsh 1, Goto[RdDMuxLp,Cnt#0&-1];
        T←ALUFMEM;              *T[0] returns selected DMux address
        Return;
```

Memory Section

Dorado supports a linear 22-bit to 28-bit virtual address space and contains a cache to increase memory performance. All memory addressing is done in terms of virtual addresses; later sections deal with the map and page faults. Figure 8 is a picture of the memory system; Figure 9 shows cache, map, and storage addressing. As Figure 8 suggests, the memory system is organized into three more-or-less independent parts: storage, cache data, and addressing.

Inputs to the memory system are NEXT task from the control section, subtask from io devices, Mar (driven from A or by the IFU), MemBase, B, the fast input bus, and an assortment of control signals. Outputs are B, Md to the processor, the F/G registers for the IFU, the fast output bus (data, task, and subtask), and Hold.

The processor references the memory by providing a base register number (MemBase) and 16-bit displacement (Mar) from which a 28-bit virtual address *VA* is computed: the kind of reference is encoded in the ASEL field of the instruction in conjunction with FF[0:1]. Subsequently, cache references transfer single 16-bit words between processor and cache; fast io references independently transfer 256-bit *munches* between io devices and storage. There is a weak coupling between the two data sections, since sometimes data must be loaded into the cache from storage, or returned to storage.

Storage is heavily pipelined, allowing new requests every 8 cycles, but requiring 28 cycles to complete a read. The state of the pipeline is recorded in a ring buffer called the pipe, where new entries are assigned for each storage reference. The processor can read the pipe for fault reporting or for access to internal state of the memory system.

Memory Addressing

Processor memory references supply (explicitly) a 16-bit displacement *D* on Mar and (implicitly) a 5-bit task-specific base register number MemBase. Subtask[0:1] (See "Slow IO") are OR'ed with MemBase[2:3] to produce the 5-bit number sent to the memory. MemBase addresses 1 of 32 28-bit base registers. The full virtual address VA[4:31] is BR[MemBase] + D. D is an unsigned number.

> The 28 bits in BR, VA, etc. are numbered 4:31 in the discussion here, consistent with the hardware drawings. This numbering conveniently relates to word boundaries.

MemBase can be loaded from the five low bits of FF, and the FlipMemBase function loads MemBase from its current value xor 1. In addition, MemBase can be loaded from 0.MemBX[0:1].FF[6:7], where the purpose of the 2-bit MemBX register is discussed in "IFU Section." The IFU loads the emulator task's MemBase at the start of each opcode with a MemBX-relative value between 0 and 3.

The intent is to point base registers at active structures in the virtual space, so that memory references may specify a small displacement (usually 8 or 16 bits) rather than full 28-bit VA's. In the Mesa emulator, for example, two base registers point at local (MDS + L) and global (MDS + G) frames.

In any cycle with no processor memory reference, the IFU may make one. IFU references always use base register 31, the code base for the current procedure; the D supplied by the IFU is a word displacement in the code segment.

> Programmers may think of Mar as an extension of A since, when driven by the processor, Mar contains the same information as A.

The base register addressed by MemBase can be loaded using BrLo←A and BrHi←A functions. VA is written into the pipe memory on each reference, where it can be read as described later. The contents of the base register are VA-D on any reference.

Processor Memory References

Memory *references* are initiated only by the processor or IFU. This section discusses what happens only when references proceed unhindered. Subsequent sections deal with map faults, data errors, and delays due to Hold.

Processor references (encoded in the ASEL and FF[0:1] instruction fields) have priority over IFU references, and are as follows:

Fetch←	Initiates one-word fetch at VA. Data can be retrieved in any subsequent instruction by loading Md into R or T, onto A or B data paths, or masking in a shift operation.
Store←	Stores data on B into VA.
LongFetch←	A fetch for which the complete 28-bit VA is (B[4:15],,Mar[0:15]) + BR.
IFetch←	A fetch for which BR[24:31] are replaced by Id from the IFU. When BR[24:31] are 0 (i.e., when BR points at a page boundary), this is equivalent to BR + Mar + Id, saving 1 instruction in many cases. *Note: the IFU does not advance to the next item of ←Id for IFetch←, so an accompanying TisId or RisId function is needed to advance.*
PreFetch←	Moves the 16-word munch containing VA to the cache.
DummyRef←	Loads the pipe with VA for the reference without initiating cache, map, or storage activity.
Flush←	Removes a munch containing VA (if any) from the cache, storing it first if dirty (emulator or fault task only).
Map←	Loads the map entry for the page containing VA from B and clears *Ref*; action is modified by the ReadMap function discussed later (emulator or fault task only).
IOFetch←	Initiates transfer of munch from memory to io device via fast output bus (io task only).
IOStore←	Initiates transfer of munch from io device to memory via fast input bus (io task only).

(Inside the memory system, there are three other reference types: IFU reads, dirty cache victim writes, and FlushStore fake-reads that result from Flush← references which hit dirty cache entries.)

The notation for these memory references has been confusing to people who first start writing microprograms. The following examples show how each type of reference would appear in a microprogram:

```
Fetch←T;                    *Start a fetch with D coming from T via Mar
T←Md;                       *Read memory data for the last fetch into T
Store←Rtemp, DBuf←T;        *Start a store with D coming from an RM
                            *address via Mar and memory data from T via B.
PreFetch←Rtemp;
Flush←Rtemp;
IOFetch←Rtemp;
IOStore←Rtemp;
Map←Rtemp, MapBuf←T;        *Start a map write with D coming from an RM
                            *address (Rtemp) via Mar, data from T via B
Map←Rtemp, ReadMap;         *Start a map read with D coming from an Rm
                            *address (Rtemp) via Mar.
LongFetch←Rtemp, B←T;       *Start a fetch reference with
                            *VA = BR[4:31] + (T[4:15],,Rtemp[0:15]).
IFetch←Stack;               *Start a fetch reference with Id replacing BR[24:31]
                            *and with D coming from Stack.
IFetch←Stack, TisId;        *Start a fetch as above and also advance the IFU to the
                            *next item of ←Id.
```

The tricky cases above are Store←, Map←, and LongFetch←, which must be accompanied by another clause that puts required data onto B. DBuf← and MapBuf← are synonyms for B←, and do not represent functions encoded in FF; these synonyms are used to indicate that the implicitly loaded buffer registers (DBuf on MemD and MapBuf on MemX) will wind up holding the data.

The encoding of these references in the instruction was discussed in the "Processor" section under "ASEL: A Source/Destination Control". The ten possible memory reference types have the following properties:

Fetch←, IFetch←, and LongFetch←

These three are collectively called "fetches" and differ only in the way VA is computed. In any subsequent instruction, memory data Md may be read. If Md isn't ready, Hold occurs, as discussed below. If the munch containing VA is in the cache and the cache isn't busy, Md will be ready at t_3 of the instruction following the fetch, with the following implications:

If Md is loaded directly into RM or T (loaded between t_3 and t_4), it can be read in the instruction after the fetch without causing Hold. This is called a *deferred* reference.

If Md is read onto A or B (needed before t_2) or into the ALU masker by a shift (needed before t_3), it is not ready until the second instruction after the fetch (Hold occurs if Md is referenced in the first instruction.). This is called an *immediate* reference.

The above timing is minimum, and delays may be longer if data is not in the cache or if the cache is still busy with an earlier reference.

Md remains valid until and during the next fetch by the task. If a Store← intervenes between the Fetch← and its associated ←Md, then ←Md will be held until the Store← completes but will then deliver data for the fetch exactly as though no Store← had intervened.

Store←

Store← loads the memory section's DBuf register from B data in the same instruction. On a hit, DBuf is passed to the cache data section during the next cycle. On a miss DBuf remains busy during storage access and is written into the cache afterwards.

Because DBuf is neither task-specific nor reference-specific, any Store←, even by another task, holds during DBuf-busy. However, barring misses, Store←'s in consecutive instructions never hold. A fetch or ←Md by the same task will also hold for an unfinished Store←.

PreFetch←

PreFetch← is useful for loading the cache with data needed in the near future. PreFetch← does not clobber Md and never causes a map fault, so it can be used after a fetch before reading Md.

IOFetch←

An IOFetch← is initiated by the processor on behalf of a fast output device. When ready to accept a munch, a device controller wakes up a task to start its memory reference and do other housekeeping.

An IOFetch← transfers the *entire munch* of which the requested address is a part (in 16 clocks, each transferring 16 data + 2 parity bits); the low 4 bits of VA are ignored by the hardware. If not in the cache, the munch comes direct from storage, and no cache entry is made. If in the cache and not dirty, the munch is still transferred from storage. Only when in the cache and dirty is the munch sent from the cache to the device (but with the same timing as if it had come from storage). In any case, no further interaction with the processor occurs once the reference has been started. As a result, requested data not in the cache (the normal case) is handled entirely by storage, so processor references proceed unhindered barring cache misses.

The destination device for an IOFetch← identifies itself by means of the task and subtask supplied with the munch (= task and subtask that issued IOFetch←). The fast output bus, task, and subtask are bussed to all fast output devices. In addition, a Fault signal is supplied with the data (correctable single errors never cause this fault signal); the device may do whatever it likes with this information. More information relevant to IOFetch← is in the "Fast IO" chapter.

IOFetch← does not disturb Md used by fetches, DBuf used by Store←, or MapBuf used by Map←.

There is no way to encode either IOFetch← or IOStore← in an emulator or fault task instruction, and there should never be any reason for doing this.

IOStore←

IOStore← is similar to IOFetch←. The processor always passes the reference to storage. The cache is never used, but a munch in the cache is unconditionally removed (without being stored if dirty). A munch is passed from device to memory over the fast input bus, while the memory supplies the task and subtask of the IOStore← to the device for identification purposes. The device must supply a munch (in 16 clocks, each transferring 16 bits) when the memory system asks for it.

The Carry20 function may be useful with IOFetch← and IOStore←. This function forces the carry-in to bit 11 of the ALU to be 1, so a memory address D on A can be incremented by 16 without wasting B in the same instruction.

Map←

This is discussed later.

Flush←

Flush← unconditionally removes a munch containing VA from the cache, storing it first if dirty. It is a no-op if no munch containing VA is in the cache; it immediately sets *Vacant* in the cache entry and is finished on a clean hit; it gives rise to a FlushStore reference on a dirty hit.

Only emulator or fault task instructions can encode Flush←, using the private pipe entry (0 or 1) pointed at by ProcSRN. The FlushStore triggered, if any, uses the ring-buffer part of the pipe. FlushStore turns on *BeingLoaded* in the cache entry and trundles through a (useless but harmless) storage access to the item being flushed; when this finishes *Vacant* is set in the cache entry; then the dirty-victim is written into storage.

Unfortunately, Flush← clobbers the Victim and NextV fields in the cache row, which causes the cache to work less efficiently for awhile.

Some applications of Flush← are discussed later in the Map section.

DummyRef←

DummyRef← writes VA into the pipe entry for the reference without initiating cache, map, or storage activity. This is provided so diagnostic microcode can exercise VA paths of the memory system without disturbing cache or memory.

DummyRef← is essential only to Midas; diagnostic microcode could get along without it. DummyRef← allows Midas to read base registers without disturbing the cache.

IFU References

The F and G data registers shown in the IFU picture (Figure 11) are physically part of the memory system. The memory system fetches words referenced by the IFU directly into these registers. The IFU may have up to two references in progress at-a-time, but the second of these is only issued when the memory system is about to deliver data for the first reference.

An IFU reference cannot be initiated when the processor is either using Mar or referencing the Pipe; for simplicity of decoding, the hardware disables IFU references when the processor is either making a reference or doing one of the functions 120_8 to 127_8 (CFlags←A', BrLo←A, BrHi←A, LoadTestSyndrome, or ProcSRN←B); or 160_8 to 167_8 (B←FaultInfo', B←Pipe*i*, or B←Config').

The IFU is not prevented from making references while the processor is experiencing Hold, unless the instruction being held is making a reference or doing one of the functions mentioned above.

Memory Timing and Hold

Memory system control is divided into three more or less autonomous parts: address, cache data, and storage sections. The storage section, in turn, has several automata that may be operating simultaneously on different references. Every reference requires one cycle in the address section, but thereafter an io reference normally deals only with storage, a cache reference only with the cache data section. Address and cache data sections can handle one reference per cycle if all goes well. Thus, barring io activity and cache misses, the processor can make a fetch or store reference every cycle and never be held.

If the memory is unready to accept a reference or deliver Md, it inhibits execution with *hold* (which converts the instruction into a Goto[.] while freezing branch conditions, dispatches, etc.). The processor attempts the instruction again in the next cycle, unless a task switch occurs. If the memory is still not ready, hold continues. If a task switch occurs, the instruction is reexecuted when control returns to the task; thus task switching is invisible to hold.

In the discussion below, *cache references* are ones that normally get passed from the address section to the cache data section, unless they miss (fetches, stores, and IFU fetches), while *storage references* unconditionally get passed to storage (IOFetch←, IOStore←, Map←, FlushStore arising from Flush← with dirty hit, and dirty-victim writes). PreFetch← and DummyRef← don't fall into either category.

Situations When Hold Occurs

A fetch, store, or ←Md is held after a preceding fetch or store by the same task has missed until all 16 words of the cache entry are loaded from storage (about 28 cycles).

Store← is held if DBuf is busy with data not yet handed to the cache data or storage sections. LongFetch← (unfortunately) is also held in this case. Since DBuf is not task-specific, this hold will occur even when the preceding Store← was by another task.

An immediate ←Md is held in the cycle after a fetch or store, and in the cycle after a deferred ←Md.

> Because the task-specific Md RAM is being read t_2 to t_3 for the deferred ←Md in the preceding cycle, and t_0 to t_1 for the immediate ←Md in the current cycle, which are coincident, hold is necessary when the tasks differ. Unfortunately, hold occurs erroneously when the immediate and deferred ←Md's are by the same task.

Any reference or ←Md is held if the address section is busy in one of the ways discussed below.

> ←Md is erroneously held when the address section is busy, an unfortunate consequence of the hardware implementation, which combines logic for holding ←Md on misses with logic for holding references when the address section is busy.

B←Pipe*i* is held when coincident with any memory system use of the pipe. Each memory system access uses the pipe for one cycle but locks out the processor for two cycles. The memory system accesses the pipe t_2 to t_4 following any reference, so B←Pipe*i* will be held in the instruction after any reference. Storage reads and writes access the pipe twice more; references that load the cache from storage access the pipe a third time.

Map←, LoadMcr, LoadTestSyndrome, and ProcSRN← are *not held for MapBuf busy*; the program has to handle these situations itself by polling *MapBufBusy* or waiting long enough, as discussed in the Map section.

Flush←, Map←, and DummyRef← *are not held* until a preceding fetch or store has finished or faulted. The emulator or fault task should force Hold with ←Md before issuing one of these references, if it might have a fetch or store in progress.

In the processor section, stack overflow and underflow and the hold simulator may cause holds; in the control section TaskingOff or an IFUJump in conjunction with the onset of one of the rare IFU error conditions may cause one-cycle holds; there is also a backpanel signal called ExtHoldReq to which nothing is presently connected--this is reserved for input/output devices that may need to generate hold in some situation. All of these reasons for hold are discussed in the appropriate chapters.

Address Section Busy

The address section can normally be busy only if some previous reference has not yet been passed to the cache data section (for a cache reference that hits) or to storage (for a storage reference, or a cache reference or PreFetch← that misses). A reference is passed on immediately unless either its destination is busy or the being-loaded condition discussed below occurs.

The address section is always busy in the two cycles after a miss, or in the cycle after a Flush←, Map←, IOFetch←, or IOStore←.

> This allows Asrn to advance; for emulator and fault task fetch and store misses, which do not use Asrn, this hold is unnecessary. Unfortunately, the display controller's word task finishes each iteration with IOFetch← and Block, so many emulator fetches and stores will be held for one cycle when a high-bandwidth display is being driven.

There are six other ways for the address section to be busy:

A cache reference or PreFetch← that misses, or a FlushStore, transfers storage data into the cache. At the end of this reference, as the first data word arrives, storage takes another address section cycle.

The preceding cache reference hit but cannot be passed to the cache data section because the data section is busy transferring munches to/from storage (or to an io device if an IOFetch← finds dirty data in the cache). Total time to fetch a munch from storage is about 28 cycles, but the cache data section is busy only during the last 10 of these cycles (9 for PreFetch or IOFetch← with dirty hit), while data is written into the cache. The cache data section is free during the interim.

The preceding storage reference, or cache reference or PreFetch← that missed has not been passed on to storage because the storage section is busy. Storage is busy if it received a reference less than 8 cycles previously, and may be busy longer as follows:

successive cache references must be 10 cycles apart;
successive write references must be 11 cycles apart;
with 4k storage ic's, successive references must be 13 cycles apart.

A cache write (caused by a miss with a dirty victim or FlushStore) ties up the address section until the storage reference for the write is started; this happens 8 cycles after the storage reference for the miss or FlushStore is started. Note that the new munch fetch starts *before* the dirty victim store and that hold terminates right after the store is *started*.

A reference giving rise to a cache write that follows any other cache miss will tie up the address section until the previous miss is finished.

The address section is busy in the cycle after any reference that hits a cache row in which *any column* is being loaded from storage.

Any reference except IOFetch←, DummyRef←, or Map← that hits a cache row in which *any column* is being loaded from storage remains in the address section until the *BeingLoaded* flag is turned off--i.e., for the first 19 of the 28 cycles required to service a miss, the reference is suspended in the address section; during the last 9 cycles of the miss, when the munch is transferred into the cache data section, the reference proceeds (except that a fetch or store will still be held because the cache data section is busy during these 9 cycles). This is believed to be very infrequent.

A more perfect implementation would suspend a reference in the address section only when the hit column, rather than any column in the row, was being loaded. However, the situation is only costly when the suspended reference is by another task; since there are 64 rows, ~1.5% of all references will be held whenever any task is experiencing a miss. There is more discussion of this in the "Performance Issues" chapter.

References to storage arise as follows:

a cache miss (from a cache reference or PreFetch←) causes a storage read;

a cache reference or PreFetch← miss with dirty victim also causes a storage write immediately after the read;

a Flush← which gets a dirty hit causes a FlushStore read reference which in turn causes a storage write of the dirty victim;

every io reference causes a storage read or write;

a Map← causes a reference to storage (actually only the map is referenced, but the timing is the same as for a full storage reference).

The following table shows the activity in various parts of the memory system during a fetch that misses in the cache and displaces a dirty victim; the memory system is assumed idle initially and nothing unusual happens.

Table 14: Timing of a Dirty Miss

Time (Cycles)	*Activity of Fetch*	*Time (Cycles)*	*Activity of Dirty-Victim Write*
0	Fetch← starts		
1	in address section	2-9	in address section (wait for map)
		3-18	in ST automaton (generate syndrome, transport to storage)
2-9	in map automaton *	10-17	in map automaton *
7-14	in memory automaton *	15-22	in memory automaton *
14-21	in Ec1 automaton	22-29	in Ec1 automaton **
21-28	in Ec2 automaton	29-36	in Ec2 automaton **
27	←Md succeeds		

* The map automaton continues busy for two cycles after a reference is passed to the memory automaton because it is necessary for the Map storage chips to complete their cycle.

** The work of the dirty-victim write is complete after it has finished with the memory automaton, but it marches through Ec1 and Ec2 anyway for fault reporting.

STOP! The sections which follow are about the Map, Pipe, Cache, Storage, Errors, and other internal details of the memory system. Only programmers of the fault task or memory system diagnostic software are expected to require this information. Since there are many complications, you are advised to skip to the next chapter.

The Map

VA is transformed into a real address by the map on the way to storage. The hardware is easily modifiable to create a page size of either 256, 1024, or 4096 words and to use either 16k, 64k, or 256k ic's for map storage. The table below shows the virtual memory (VM) sizes achievable with different map configurations. However, the cache configuration limits VM size independently, as discussed later, and this limit may be smaller than the Map limit.

Table 15: Map Configurations

Map IC Size	*Page Size*	*VM Size*	*Map Addressed By*	
2^{14}	2^{8}	2^{22}	VA[10:23]	
2^{14}	2^{10}	2^{24}	VA[8:21]	
2^{14}	2^{12}	2^{26}	VA[6:19]	requires 16k-word cache
2^{16}	2^{8}	2^{24}	VA[8:23]	
2^{16}	2^{10}	2^{26}	VA[6:21]	requires 16k-word cache
2^{16}	2^{12}	2^{28}	VA[4:19]	requires 16k-word cache sans parity
2^{18}	2^{8}	2^{26}	VA[6:23]	2^{18}-bit ic's don't exist yet
2^{18}	2^{10}	2^{28}	VA[4:23]	2^{18}-bit ic's don't exist yet

Larger page sizes increase the virtual memory size limit. Since the 4k-word cache imposes a 2^{25}-word size limit (2^{26} if the parity bit in the address section is converted into another address bit), the largest VM sizes are only achievable in conjunction with a 16k-word cache. Larger page sizes might reduce map and storage management overhead; our experience in this area is inconclusive but suggests that 4k-word pages would only be desirable with very large storage configurations.

Note that the physical storage size limit is unaffected by either cache parameters, map ic size, or page size because RP is large enough to address the largest possible storage configuration (4 modules using 2^{18}-bit MOS RAM components), even when the smallest page size is used; this maximum size is 2^{24} words.

The cache handles virtual addresses, so the map is never involved in cache references unless they miss.

A consequence of virtual addresses in the cache is that it is illegal to map several virtual pages into the same real page (unless all instances are write-protected). This restriction prevents cache and storage from becoming inconsistent.

A map entry contains a 16-bit real page number (RP) and three flags called *Dirty*, *Ref*, and *WP*, which have the following significance:

WP — write-protects the page; a fault occurs if a write is attempted.

Dirty — indicates that storage has been modified; set by any IOStore← or by a dirty-victim write; Store← does not set Dirty.

Ref — indicates that the page has been referenced; set by any storage reference except Map←; cleared by Map←.

The combination WP = true with Dirty = true makes no sense, and encodes the *Vacant* state of the map entry. A map entry is vacant if it has no corresponding page in real memory.

Faults

Every storage reference causes a mapping operation. If mapping yields a valid real address, the reference proceeds normally. Otherwise, the storage reference is aborted, and *MapTrouble* is reported as discussed later. There are two kinds of faults:

Page fault	reference to a vacant map entry (WP = true, Dirty = true)
WP fault	Store← that misses, IOStore←, or dirty-victim write with WP true. (Dirty-victim WP faults should not occur if the map and cache are handled as proposed below.)

Writing the Map

Map←, which can only be encoded in an emulator or fault task instruction, is used to write the map; like other storage references, it returns previous map contents in the pipe, where they can be read.

Map← first writes B[0:15] and TIOA[0:1] into *MapBuf* (a buffer register on the MemX board) and turns on *MapBufBusy* in the pipe; 9 cycles later (barring delays) MapBuf has been written into the Map entry addressed by the appropriate bits of VA and MapBufBusy is turned off.

B[0:15] are the real page number, TIOA.0 is WP, and TIOA.1 is Dirty. Map← zeroes Ref, and there is no direct way to write a map entry with Ref = 1; a fetch, Store←, or PreFetch← to the appropriate page after loading the map entry will set Ref = 1.

Map← never wakes up the fault task.

> If previous map contents indicated *Vacant* or had a parity error, *MapTrouble* will be true in the pipe but not reported to the fault task. *Quadword* and *syndrome* in the pipe, not written by Map←, contain previous values.

For all programming purposes, Map← is complete on the cycle when MapBufBusy is turned off; at this time, previous map contents are valid in the pipe entry. Polling *MapBufBusy* until it is false is the only way to find out when the pipe entry is valid.

Since Map← never faults and doesn't use any pipe information clobbered by an overlapping reference, another reference may be started without waiting for Map← to finish, unless the following reference is another Map←. Also, Map← does not interfere with Md or DBuf, so its only interference with other kinds of references is its use of the private pipe entry (0 or 1) pointed at by ProcSRN. However, *it is illegal to issue another Map←, LoadMCR, LoadTestSyndrome, or ProcSRN← without waiting for Map← to finish.* These functions (discussed later) share the MapBuf register with Map←; there is no Hold arising from *MapBufBusy*, so the microprogram must ensure that MapBuf is free when one of these functions is executed.

> B is latched in MapBuf during t_2 to t_3 and TIOA[0:1] are clocked into MapBuf at t_2 for all of these; then MapBuf is written into MCR, TestSyndrome, or ProcSRN at t_3 or into the Map at t_{14} (if no delays). In other words, MapBuf is a buffer register for all registers on the MemX board that are loaded from B.

Reading the Map

Every storage reference causes mapping and returns old contents of the relevant map entry in the pipe. I.e., Ref and Dirty may change as a result of a reference--old values appear in the pipe.

A ReadMap function accompanying Map← prevents the map entry from being modified, so that old contents can be read from the pipe without also smashing the map entry.

Flushing One Page From the Cache

Any cache reference or PreFetch← that misses or any IOFetch← or IOStore← sets Ref in the map; IOStore←'s set Dirty as well. If the victim for the miss or hit for the Flush← is dirty, Ref and Dirty for its map entry also get set. However, Store← does not set Dirty in the map entry until that munch is chosen as victim.

For this reason, any calculation based upon Dirty must first validate the map Dirty bit by flushing associated cache entries, as discussed below.

In addition, almost any change to a map entry requires a flush, again because of problems with dirty cache entries. The following examples illustrate this point:

Before changing RP, a flush prevents dirty victims from being written into the previous real page (if the old page had WP false).

Before turning on WP, a flush prevents dirty cache entries from being written into the now write-protected page.

Before turning off WP, a flush prevents multiple cache entries for a munch, one write-protected, the other not (The cache will not work correctly, if there are multiple entries for a munch.).

Before sampling Ref, flushing is required so that subsequent references to the page will set Ref = 1 and so that dirty munches in the cache will not erroneously set Ref = 1 when they are displaced.

Before clearing Dirty, a flush prevents dirty munches subsequently displaced from the cache from erroneously setting Dirty again.

To flush a 256-word page from the cache, 16 Flush← references may be made, one to each munch of the page (64 with 1024-word pages). Flush← invalidates any existing cache entry for the munch (and stores the munch if dirty).

This succeeds iff there are no anomalous multiple cache entries for a particular munch. Multiple cache entries for a munch should never occur except prior to system initialization or when some of the debugging features are turned on in Mcr.

Flushing the Entire Cache

Depending upon what kind of storage management algorithm is used, it may be desirable to clear out the entire cache; for example, this might be useful before looping through all the map entries to sample Ref. It would be extremely expensive to do this with Flush← one page at-a-time (2^{16} Flush←'es for 1M words of storage). The cleverest method which we

have thought of for doing this is as follows:

Designate 4 consecutive 256-word pages (64-row cache) or 16 consecutive pages (256-row cache) that contain vacant map entries; the munch VA's in these pages will span every row in the cache. Make one pass through the cache for each column; before each pass, load Mcr with UseMcrV true and McrV equal to the column--even though it is usually illegal to modify Mcr while the memory system is active, it should be safe to change these particular fields. Then do PreFetch←'es for all 64 or 256 munches in the designated pages; these PreFetches will all miss and map fault, leaving the selected column filled with vacant cache entries. After clearing all four columns, restore Mcr to its previous value. While this flush is going on, io tasks may continue to reference memory, but they will experience more misses and longer miss wait than usual. The total time for this algorithm is about 9 cycles/PreFetch or about 115 μsec with 64 rows or 460 μsec with 256 rows in the cache.

Map Hardware Details

The map and its control are on the MemX board. Physically, map storage consists of 21 16k, 64k, or 256k x 1 MOS RAM's; in addition to the 19 bits discussed earlier, there are a duplicate of the *Dirty* bit and an odd parity bit.

Dirty is duplicated so map parity won't change when both *Dirty* bits are set. Ideally *Ref* should also be duplicated, but it is not, and *Ref* is not checked by map parity. The parity written on Map← is the exclusive-or of the two B byte parity bits and TIOA.0 (i.e., *WP*). Parity failure on any map read will cause *MapTrouble* and *MemError* and wakeup the fault task when appropriate.

On a cache reference that misses or on an IOFetch← or IOStore←, the map read starts at t_4 and the real address is passed to storage at t_{14}.

The MOS RAM's in the map require refresh, carried out like the storage refresh discussed later.

An Automatic Storage Management Algorithm

We envision for Mesa, Lisp, etc. an automatic storage manager that will pick pages in storage which have not been referenced recently for replacement by new ones. This manager will use the Ref bits in map entries to determine which pages have not been referenced for a long time.

The storage manager discussed here controls N pages, where N is some subset of all pages in storage; in general N will vary. A procedure called DeliverPage() returns RP for one of the N pages to the caller and removes that page from N; a procedure called ManagePage(RP,Age) adds a page to N. A page returned by DeliverPage has been removed from the virtual space; pages accepted by ManagePage may or may not be vacant.

Entries for the N pages are sorted into 8 bins, such that entries in the bin 0 have age 0, bin 1 age 1, etc. Whenever DeliverPage has been called N/8 times or after a specified elapsed time has occurred, all N pages are aged, which means:

(a) Entries originally in bin 7 wind up in bin 0 if they have been referenced, bin 7 if not referenced;

(b) Entries in bin i (i # 7) wind up in bin 0 if referenced or bin i+1 if not referenced.

This aging is performed by first clearing the entire cache using the clever algorithm discussed earlier, then sampling and zeroing Ref.

DeliverPage first returns the RP of any page on the vacant queue. If the vacant queue is empty, it next scans entries on the disk write-complete queue; if one is found that has not been referenced in the interim, its map entry is cleared and its RP is returned; if referenced, it is moved to bin 0. If the disk write-complete queue is empty, entries in bin 7 are scanned; if this bin is exhausted, bin 6 is scanned, etc., until finally bin 0 is scanned. When an entry has been referenced, it is moved to bin 0; when unreferenced but dirty, it is put on the disk write queue; when unreferenced and clean it is returned.

The caller of DeliverPage will frequently be a disk read or new page creation procedure. It should complete its work and then call ReturnPage(RP,0) to restore the page to the storage manager. ReturnPage will put the page on the vacant queue, if it is vacant, or into bin 0.

Mesa Map Primitives

Basic Mesa mapping primitives are:

Associate[vp,rp,flags] adds virtual pages to the real memory, or removes them if flags = Vacant.

SetFlags[vp,flags] RETURNS oldValue: flags reads and sets the flags. If flags = Vacant, the page is removed from the real memory.

These are defined as *indivisible operations* and are implemented trivially on a machine with no cache (e.g., D0). For example, if a SetFlags clears Dirty and sets WP, the returned value of Dirty tells correctly whether the page has been changed--no store into the page may occur between reading Dirty and setting WP. A ReadFlags primitive could also be provided, but no use for it is known.

One intended use of the primitives is illustrated by the following Mesa sequence for removing a virtual page from real memory:

```
oldFlags ← SetFlags[v,WP];
IF oldFlags.Dirty THEN WritePage[...]
SetFlags[v,Vacant]
```

This sequence prevents the page from being changed during the write. Another possibility would be just to clear Dirty, and then to check it again after the write. This must be done properly, however, to avoid a race condition:

```
WHILE (oldFlags←SetFlags[v, Vacant]).Dirty
    DO SetFlags[v, [Dirty: false]]; WritePage[...]; ENDLOOP
```

To avoid inconsistent map and cache entries, SetFlags[*v*, ...] must remove entries for page *v* from the cache. Unfortunately, since we don't want to make the cache removal process atomic, parts of the page already passed over by the removal process could be brought back into the cache before the process is complete. The implementation of the primitive must allow for this.

On Dorado it is, unfortunately, impossible to implement these primitives as indivisible operations because almost any change in the map flags must be preceded by clearing all cache entries in the page. However, it is unacceptable to do this with TaskingOff because the time required might be as long as 16*10 cycles with 256-word pages or 64*10 cycles with 1024-word pages (if every munch in the page is in the cache and dirty), which is too long. Consequently, io tasks will be active during the removal process, and one of them might move a munch back into the cache after it has been passed over by the removal process.

For this reason, the primitives are indivisible on Dorado only if io tasks are known not to touch any page for which the flags are changing. Subject to this restriction, the implementation of SetFlags(v, ...) proceeds as follows (Associate is similar.):

Flush all cache entries for the page in question. If any entry is dirty, removal will cause a write and set Dirty in the map, as discussed earlier.

oldFlags ← map[v].Flags

If turning on WP: map[v].flags ← [WP: true, Dirty: false, Ref: false].

If setting Vacant: map[v].flags ← [WP: true, Dirty: true, Ref: false].

If turning off WP: map[v].flags ← [WP: false, Dirty: false, Ref: false].

These are done with Map← after which old data is retrieved from the pipe (The Map← and pipe readout are indivisible.).

Flush all cache entries for the page again; if some io task dirtied the page after the initial flush commenced, then the fault task will crash the system during this second flush (assumption about io tasks not dirtying the page is violated). Otherwise, the second flush will clear all munches in the page from the cache without disturbing the map.

```
oldFlags.Dirty ← oldFlags.Dirty OR map[v].Dirty;
oldFlags.Ref ← oldFlags.Ref OR map[v].Ref;
map[v].Flags ← f.Flags.
```

These are done with another Map← which again retrieves the old dirty flag from the pipe (possibly followed by PreFetch to set map[v].Ref true).

Note: These primitives do not support the complete cache clear discussed earlier; another primitive will probably needed to do this. Also, we really want a primitive that will allow the flags to be sampled and Ref zeroed without changing the value of WP or Dirty. And efficiency may demand primitives particularly tailored to the needs of whatever storage management algorithm is employed.

The Pipe

Information about each reference is recorded in the 16-word pipe memory. Pipe layout is shown in Figure 10, which you should have in front of you while reading this section. The processor reads the pipe with the B←Pipe0, ..., B←Pipe5 functions. *You should note that Pipe0, 1, and 5 are read high-true, while Pipe2 and 3 are read low-true; Pipe4 contains a mixture of high and low-true fields; 150361_8 xor Pipe4' produces high-true values for all fields in Pipe4. The discussion in this section assumes that all low-true fields have been inverted.*

It is illegal to do ALU arithmetic on pipe data (not valid soon enough for carry propagation), and B←Pipe*i* is illegal in the same instruction with a reference because Hold won't be computed properly.

> The *EmulatorFault*, *NFaults*, and *SRNFirstFault* stuff in Pipe2, which duplicate what B←FaultInfo would read back, is not part of the pipe, although it is read by B←Pipe2'; B←Pipe2' is simply a convenient decode for reading it back--this will be discussed in the section on fault handling, not here. Similarly, *Dirty*, *Vacant*, *WP*, *BeingLoaded*, *NextVictim*, and *Victim* stuff in Pipe5 is not part of the pipe and is read back by B←Pipe5 purely for decoding convenience. This information, used primarily for debugging, is discussed later.

The *Task*, *SubTask*, VA, and cache control stuff in Pipe0, 1, 2, and 5 is used both internally by the memory system and externally by the processor. Map and error stuff in Pipe3 and 4 is solely for memory management and diagnostic activities carried out by the processor.

Two main problems in dealing with the pipe are:

> Finding the pipe entry for a particular reference;
> Knowing when various bits become valid;

How the Pipe Is Addressed

System microcode is expected to use the pipe in only two situations: fault handling by task 15 (the "fault task") and reading map or base registers by task 0 (the "emulator"). Other tasks will not read the pipe. This rigid view of how the pipe will be used during system operation has motivated the implementation discussed below.

Pipe entries are addressed by 4-bit *storage reference numbers*, or SRNs, assigned to each storage reference. All task 0 and task 15 references except PreFetch← with miss (and implicit FlushStore and Victim references) use the SRN contained in ProcSRN exclusively; all other references share SRN's 2 to 15, which form a ring buffer addressed by an invisible register called ASRN.

To read a pipe entry, first ProcSRN←B addresses the pipe entry, then the contents of that entry are read with B←Pipe*i*. In system microcode, the emulator is expected to keep the value 0 in ProcSRN to avoid smashing the ring buffer on references; if the fault task needs to make a reference, it will normally load ProcSRN with 1 and use that SRN for the reference; the fault task will manipulate ProcSRN however it likes to examine the pipe but always restore it to 0 before blocking; other tasks will not use ProcSRN. This implementation is welded to the assumption that only the fault task will probe the pipe when io tasks are running.

To io task references and emulator PreFetch←'es that miss, the cache address section's SRN, called ASRN, is assigned at t_2. ASRN will be advanced to the next ring value iff the reference starts the map. In all other cases ASRN remains unchanged and is used by the next reference as well.

A reference starts the map unless it is a DummyRef←, a cache reference or PreFetch← that hits, or a Flush← that misses or gets a clean hit. A convenient way to guarantee that the map is started without worrying about the contents of the cache is to do a Map← in the emulator or an IOFetch← in any other task. The reasoning behind this treatment of ASRN is explained in the section on fault reporting.

Tasks 1 to 14 generally cannot find out the SRN for their last reference. Even if this were determined somehow by polling all the pipe entries, there would be no assurance that, meanwhile, a higher priority task didn't clobber the pipe entry.

Because of its single pipe entry, the emulator must wait for an earlier reference to finish or fault, before starting another. Of all emulator references, only a fetch, Store←, or PreFetch← might fault. However, PreFetch← doesn't use the private pipe entry, so only a preceding fetch or Store← might still be in progress when a new reference is issued. If the new reference is another fetch or Store←, it will hold until the preceding one finishes (no problem). Hence, the only restriction imposed by the private pipe entry is that the emulator must cause hold with ←Md before issuing Map←, Flush←, or DummyRef←, if a fetch or Store← might still be in progress.

Timing constraints do not permit generating Hold in the above case.

When the Pipe is Accessed

Conceptually, the pipe is three different memories. First, VA, task, subtask, and cache control bits in Pipe0, 1, 2, and 5 are written during the reference. Next, the 20 bits of map information in Pipe3 and Pipe4 are written following the map read-write (if any). Finally, the error correction-detection stuff in Pipe4 is written following the storage read (if any). The memory system needs one cycle for each of these accesses.

However, the hardware treats the pipe as only two separate memories internally, or as only a single memory for purposes of holding the processor. In other words, within the memory system Pipe0, 1, 2, and 5 may be accessed by one part of the pipeline, while another part independently accesses Pipe3 and 4. But processor accesses by B←Pipe*i* are held, if the memory system wants *any* part of the pipe. Worse, the memory system uses the pipe between even clocks (t_0 to t_2), the processor between odd clocks (t_1 to t_3), so the processor is locked out for two cycles during each of these intervals.

Programs can safely read Pipe0, Pipe1, Pipe2, or Pipe5 (i.e., task, subtask, VA, and cache control stuff) in the cycle after any reference, since these are updated at the end of the cache address section cycle. B←Pipe*i* in the cycle after a reference will hold for one cycle while the memory system uses the pipe.

Values in a pipe entry are *not reset* at the onset of a reference and Pipe3 and Pipe4 are not written at all unless storage is accessed. Consequently, Pipe3 and Pipe4 may refer to a previous reference ***Caution***.

The control bits in Pipe2' and Pipe5, used by the memory system, also indicate (to the fault task) what kind of reference is described in the pipe, as follows:

PreFetch	a PreFetch← reference
CacheRef	a fetch or store
Store'	Store←'
IFURef	IFU fetches
RefType	distinguishes read, write, and Map← storage references
FlushStore	dirty victim write triggered by Flush←
ColVic	cache column of a hit, or of the victim on a miss

DummyRef← finishes immediately and only VA in Pipe0 and Pipe1 and the stuff in Pipe2 are relevant. For Flush←, cache information in Pipe5 is also valid. Flush← finishes immediately because the resulting FlushStore and dirty-victim write references (if any) are started in ring-buffer pipe entries.

Programs can read map stuff (Pipe3 and *Ref, WP, Dirty, MapTrouble,* and *MemError* in Pipe4) as soon as that part of the reference is complete. For Map←, completion of the map read is coincident with *MapBufBusy* going false, determined by polling. For a fetch or store, there is no way to distinguish completion of the map read from completion of the entire reference. Consequently, Pipe3 and Pipe4 are normally read by doing ←Md (which holds for completion), then reading the pipe.

For IOFetch←, IOStore←, and PreFetch← there is no way to tell when the reference has finished, except by waiting longer than the memory can possibly take to complete the reference.

IOStore←'s and dirty victim writes zero the *Syndrome* and *EcFault* fields in Pipe4. Hence, the only reference that leaves junk in these bits is Map←; the fault task can distinguish pipe entries for Map← by means of the *RefType* field.

All data in Pipe0, 1, 2, and 5 except *FlushStore* and *ColVic* are written at t_3, and can be read immediately after a reference. However, *FlushStore* and *ColVic* are written at t_4. Ordinarily, this would mean that their values could not be read safely; however, since B←Pipe*i* is held in the cycle after a reference, the values will always be ok.

In the best case, map information in Pipe3 and Pipe4 will be loaded at t_{14}, fault and error corrector information in Pipe4 at t_{48}.

Faults and Errors

Remember that high-true values for all fields in the Pipe are used in the following discussion.

Errors

Several events cause memory errors from which the system does not recover. Errors halt the processor if the *MemoryPE* error is enabled (see "Error Handling"). If *MemoryPE* is disabled, the program will continue without any error indication. *MemoryPE* conditions are:

Byte parity errors from the cache data memory (checked on write of a dirty victim, not on ←Md or IFU reads); the processor checks Md parity (see "Error Handling") and the IFU checks F/G parity;

Byte parity errors from fast input bus;

Cache address memory parity errors.

Faults

Other events cause *faults*. A fault condition is indicated in the *MapTrouble, MemError,* and *EcFault* fields of Pipe4 when it occurs; in addition, the fault task is woken to deal with the situation unless *NoWake* is true in Mcr. The encoding of the various errors is as follows:

Table 16: Fault Indications

Kind of Error	*Name*	*MapTrouble*	*MemError*	*EcFault*
Map parity error	*MapPE*	1	1	--
Page fault	*PageFlt*	1	0	--
Write-protect	*WPFlt*	1	0	--
Single error	*SE*	0	0	1
Double error	*DE*	0	1	1

In the above table, *WPFlt* and *PageFlt* have the same encoding; these must be distinguished by means of the *RefType* field in Pipe2 and the *WP* bit in Pipe4; *WPFlt* can only occur for Store←, IOStore←, or dirty-victim stores that encounter *WP* true.

MapTrouble might be true and reported to the fault task on a fetch or store that misses or an IOFetch←, IOStore←, FlushStore, or dirty-victim write. Flush← and DummyRef← never cause *MapTrouble*. Map←, PreFetch←, or IFU fetches might record *MapTrouble* in the pipe but never wake the fault task. Map faults on IFU fetches are reported instead to the IFU, which buffers the fault indication until an IFUJump occurs to an opcode with at least one instruction byte in the word affected by the map fault; then a trap occurs, as discussed in "Instruction Fetch Unit".

In system microcode, we expect a *WPFlt* and *PageFlt* due to IOFetch←, IOStore←, FlushStore, or a victim write to indicate a programming error; however *MapPE* might occur. Note that if any kind of *MapTrouble* occurs on a storage write (i.e., on an IOStore←, FlushStore, or victim write), storage is not modified and contains the old value; however, the map's *Dirty* bit will be true, even though the storage write has not completed.

SE and *DE* may occur on any cache reference or PreFetch← that misses or on an IOFetch←. Map←, IOStore←, DummyRef←, and Flush← never cause these errors. Also note that fault task wakeup on an *SE* requires not only *NoWake* false but also *ReportSE* true in Mcr; the fault indication transmitted with the munch for an IOFetch← is set only for *DE*, never for *SE*.

Unlike map faults, data errors on IFU fetches and PreFetch←'es are reported to the fault task. This must be done for DE's, which are fatal; for corrected SE's, the fault causes no disruption to the program because the fault task, after logging the failure, simply lets the task that faulted continue.

The special things about a fault are:

If a program obeys the rules given earlier, hold will occur until any fault is reported or until the program can proceed safely.

EmulatorFault in B←FaultInfo is set true if a fault is described by the emulator or fault task pipe entry (0 or 1) pointed at by ProcSRN;

FirstFaultSRN in B←FaultInfo is loaded if *FaultCnt* is -1 (indicating no faults) or if *FirstFaultSRN* was previously zero;

FaultCnt in B←FaultInfo is incremented;

B←FaultInfo stuff is updated and the fault task is woken at the *end* of the storage pipeline, but sufficiently in advance of hold termination that it will surely run first. For this reason, any operation that might fault is illegal with tasking off.

References leave the pipeline in the order that they entered.

Subtleties: In the event of a miss with a dirty victim, the new cache entry read *starts* and *finishes* before the victim write. However, *data transport* of the victim to storage finishes before data transport of new data into the cache starts--storage actually reads new data first, but meanwhile transports the victim into a holding register on the storage board, from which it is written into storage after the read.

Pipe entries identified by *EmulatorFault*, *FirstFaultSRN*, and *FaultCnt* represent complete storage references;

The task that faulted is not blocked; hold terminates as though no fault had occurred; the task will continue unless the fault task changes its PC.

The fault task is expected to read B←FaultInfo, service all faults it describes, service stack underflow or overflow, then block. Because it is highest priority, the fault task cannot do much computing (io tasks that are lower priority have to be serviced); probably it should not make any memory references itself. Its normal actions are:

crash (uncorrectable data errors, map faults by tasks other than the emulator);

block letting the task that faulted continue (correctable data errors); or

change the TPC of the emulator to an appropriate trap routine (emulator map faults, stack overflow or underflow).

EmulatorFault and *FaultCnt* are automatically reset by B←FaultInfo. These can be read without reset in B←Pipe2 (primarily for use by Midas).

Several faults could occur while the fault task is running (due to references initiated before the fault task was awakened). In this case, when the fault task blocks, it will continue because of the pending wakeup, and so service the faults. Only while the fault task is running or while tasking is off is it possible for *FaultCnt* to become greater than one.

Remarks

The careful scheme in which ASRN is advanced only for storage references, and faults reported in precise order is essential. If faults were reported out of sequence, then the fault task might see Pipe0 to Pipe2 stuff inconsistent with Pipe3 and Pipe4 error indicators for a previous loop through the ring buffer.

The hardware must not and does not report *MapTrouble* until the end of the pipeline. If this were not true, then an SRN might report *MapTrouble* before its predecessor reported *SE* or *DE*; this could screw up the fault task.

In tasks other than the emulator, map faults will probably represent programming errors. In the emulator, page-not-in-memory and write-protect faults are expected, and the fault task will trap the emulator to a fault-handling Mesa program. Information saved by the trap microcode must be sufficient to continue the faulted opcode at the instruction that faulted.

The B←DBuf FF decode permits the fault task to retrieve data being written when a Store← faults.

Error Correction Faults

For error correction purposes, *munches* are divided into four *quadwords*, each containing 64 data and 8 check bits.

At the end of a storage read, the hardware indicates DE after a double-error or SE after a single error as discussed earlier. The SE or DE indication is unambiguous *assuming* at most two bits in error in any 64-bit quadword; for an odd number of errors greater than 2, the hardware erroneously reports an SE; for an even number of errors greater than 2, DE is reported. If several quadwords in a munch suffer errors, the hardware reports the *first* DE, if any, or the *last* SE, if no DE's.

Error correction can be enabled/disabled by the LoadTestSyndrome function discussed later; when enabled, the hardware will complement (= correct) any SE; for DE's the hardware does not modify any bits from storage.

The absolute address of the quadword containing the reported error is RP[0:15]..VA[24:27]..quadword[0:1] with 256-word pages, or RP[2:15]..VA[22:27]..quadword[0:1] with 1024-word pages. I.e., the word address of the first word in the quadword would be these 22 bits with two low-order zeroes appended.

SE and DE are derived from the 8-bit syndrome field in Pipe4. Syndrome = 0 means no error; neither DE nor SE should be true in this case. Syndrome non-0 with an odd number of 1's should have SE indicated. Syndrome non-0 with an even number of 1's or an invalid word code (discussed below) should have DE indicated.

See Figure 11 for the correspondence between syndrome and bits within the quadword.

For SE's, syndrome specifies exactly which of the 64 data bits or 8 check bits was in error. If syndrome has a single one in it, then the corresponding checkbit was in error. When syndrome contains more than one 1, then syndrome[4:6] indicate which word in the quadword suffered the error as follows:

word 0	011
word 1	101
word 2	110
word 3	111

The other four values of syndrome[4:6] are impossible for an SE and are reported as a DE.

Syndrome[0:3] indicates the bit position within that word; unfortunately these bits are reversed, so that the bit number is given when the bits are taken in the order 3, 2, 1, 0. Syndrome[7] is the parity of the syndrome, and a double error is indicated by a non-zero syndrome having even parity.

Storage writes leave garbage in the *EcFault* and *Syndrome* fields of the Pipe; the fault task must distinguish these cases by means of the *RefType* field in Pipe2.

As discussed below in the "Testing" and "Initialization" sections, *TestSyndrome* is xor'ed with check bits that would otherwise be written on storage writes. This means that Syndrome-of-read equals TestSyndrome-of-write is an exact indication of no-error. However, the hardware always reports non-zero syndrome as an error, as discussed above, regardless of what's in TestSyndrome.

Dirty is set in the cache after a Store← that misses, despite any fault, so when that munch is chosen as victim, it will be written back into storage. Consequently, if the fault task attempts recovery from a double error on a Store←, it may have to clear the cache address section's Dirty bit for the munch using the tricky sequences discussed later.

Storage

Storage is organized into *modules* consisting of two identical boards per module. The modules appear in the chasis as shown in Figure 2. Depending on whether 16k-bit or 64k-bit IC's are used, a module stores 256k or 1m 64-bit (+8 check bit) quadwords. A Dorado can have up to 4 modules, for a maximum of 16m words. Every module must be the same size--it is illegal to mix module sizes.

The module in slot 0 supplies the first quarter of real memory; slot 1, second quarter; slot 2, third quarter; and slot 3, fourth quarter. In other words, real memory addresses are not interleaved among modules and the address range covered by a particular module cannot be controlled by the firmware.

The B←Config' function (Figure 10) returns *M0*, *M1*, *M2*, and *M3* which are true only when a module is plugged into the corresponding storage board slot. *ChipSize* indicates what size ic's are used on the storage boards. The memory system automatically adjusts itself to operate according to the IC size in use on the storage boards.

When 256k x 1 MOS storage ic's become available, we plan to replace the *4k* and *16k* wires on the backplane by an extra address wire and a *256k* wire; at this time we will lose the ability to handle 4k x 1 and 16k x 1 ic's and the hardware will allow either 64k or 256k storage ic's to be used.

MOS ic's used on storage boards (and in the map) must be refreshed at regular intervals, else they drop data. This occurs during refresh references once every 16 μs. Every MOS ic on every storage board participates in every refresh reference, and one row of data is refreshed each time. This means that 64 (4k ic's), 128 (16k ic's), or 256 (64k ic's) refresh references are required to refresh all data (So total refresh time is 1, 2, or 4 ms--the spec on 4k and 16k ic's is 2 ms per refresh, believed very conservative. 64k ic's are not specified yet.).

The time for each refresh reference is 8 cycles (13 cycles with 4k-bit ic's), same as normal references. Refresh hardware competes for storage access with the cache data section and fast io references. During the first 8 μs of a 16 μs period, refresh defers to normal references; during the last 8 μs, it preempts normal references.

The Cache

The physical cache structure consists of 256 entries in an array of 64 rows by 4 columns. Each entry holds 15 address bits, a parity bit for the address bits, four flag bits, and one munch of data (= 256 data bits + 32 check bits). Hence, the cache holds a total of 4k words of data.

> The address section is implemented with 256-word RAM's, but only 64 words are presently used. The data section uses 1kx1 RAM's for storage. When sufficiently fast 4kx1 ECL RAM's become available, we plan to use them in the cache data section and utilize all 256 words in the address section. In this case, the cache geometry will be 256 rows by 4 columns (16k words in the data section).

The cache address section stores 4 flag bits discussed below, 15 VA bits, and 1 parity bit. The way the VA bits are assigned depends upon whether or not 4k x 1 ECL RAM's are used in the cache data section. VA[7:19] are stored in the address section for all configurations. Two other bits are either VA[5:6] or VA[20:21]; VA[5:6] are used with 4k ic's in the cache data section (VA[20:21] then appear in the row address of the cache, so they don't have to be stored). The hardware is also arranged so that the parity bit may be replaced by VA[4].

In other words, the cache initially implements a 2^{25}-word virtual memory with provision for expanding this to 2^{27} words when 4k x 1 RAM's are available or to 2^{28} words at the cost of eliminating the parity bit in the address section. However, the map organization also limits virtual memory, probably to a smaller size than the cache limit, as discussed earlier.

Normally, the cache is invisible to the programmer except for problems with map/cache consistency discussed in the map section. However, features discussed below in "Testing" allow more direct access for checkout, initialization, and error recovery.

An address VA, if in the cache at all, must be in one of the four columns of the row addressed by VA[22:27] (or VA[20:27] if the cache is expanded). References compare the appropriate 15 or 16 bits of VA[4:21] with the values stored in each of the 4 columns to determine which cache entry, if any, contains VA.

The VNV memory contains two two-bit entries for each row of the cache. The *Victim* field specifies the cache column displaced if a miss occurs in this row. The *NextV* field is the next victim. When a miss or a hit in Victim occurs, Victim←NextV is done. When a miss,

hit in Victim, or hit in NextV occurs, NextV←Victim.0',,NextV.1' is done (i.e., NextV is loaded with a value different from both the original NextV and Victim). This strategy is not quite LRU, since there is a 50-50 guess on the ordering of the third and fourth replacements. This treatment of VNV is used for fetches, Store←, PreFetch←, and IFU fetches but not for IOFetch←, IOStore←, or Map←, which don't use the cache.

On a Flush←, Victim is written with 0 on a miss or with the column of the hit and NextV is written with Victim.0',,NextV.1'. If the Flush← hit a dirty cache entry, then a FlushStore reference is fabricated which will wind up writing Victim (= column hit by the Flush←) back into storage. The FlushStore reference will also do Victim←NextV and NextV←Victim.0',,NextV.1' again. This means that the VNV entry for the row touched by a Flush← is effectively garbaged, which probably won't affect performance much.

A better strategy for Flush← and IOStore← would be as follows: On a miss, Victim and NextV remain unchanged; on a hit in a column different from Victim, Victim←hit column, NextV←Victim; on a hit in Victim, no change.

The *UseMcrV* feature discussed in "Testing" allows Victim and NextV to be replaced by *McrV* and ***McrNV.***

Associated with each cache entry are four flag bits that keep track of its state, as follows:

Dirty - set by Store←, cleared when loaded from storage. This bit does not imply anything about the map's Dirty bit. The cache Dirty bit causes a storage write when the entry is chosen as victim, and the map's Dirty bit is set at that time.

Vacant - set by hit on Flush←, hit on IOStore←, or Store← into a write-protected entry, cleared when the entry is loaded from storage. Vacant is not set after an SE or DE. Vacant prevents the entry from matching any VA presented to the cache.

WriteProtect - a copy of the map's WP bit. It is copied from the map when the cache entry is loaded and not subsequently changed. If a Store← is attempted into a write-protected entry, the entry is invalidated, there is a cache fault, and a write protect fault will be reported by the map.

BeingLoaded - set while an entry is waiting for data from storage. Any reference that hits in the same row will remain in the cache address section until the bit goes off; any reference or ←Md following the one which hit a row being loaded will be held.

Remark

At the end of a miss, data from the error-corrector is loaded into the cache 16 bits/clock. Not until all 16 words of the munch have been loaded is Md loaded and the task (which has been held) allowed to continue. A scheme whereby the word being waited for is loaded into Md concurrent with writing it into the 1kx1 RAM's has been considered but rejected as too complicated. This would reduce average miss time from about 28 cycles to about 24.

Initialization

This section outlines the order in which parts of the memory system can be initialized.

Clocks

The instruction decoding flipflops of the memory section are enabled when the processor clocks are enabled. All other memory clocks are enabled by a signal called *RunRefresh*, as discussed in "Dorado Debugging Interface".

When RunRefresh is true, clocks internal to the memory system always happen, even if the processor is halted. When RunRefresh is false, memory clocks run with the processor. Except for low-level debugging of the memory system itself, RunRefresh should be true. Otherwise, storage will not retain data at breakpoints.

Mcr Register

The Memory Control Register (Mcr) contains fields that affect the memory system (see Figure 10). Mcr is intended to facilitate testing, and in some cases initialization. The register can be loaded with the Mcr← fu·ıction and read back over the DMux. Bits in Mcr are as follows (Some of these bits are loaded from A and others from B, as indicated in Figure 10):

dVA←Vic — On each reference, write the cache address entry selected by the row of VA and column of Victim (Note: Victim determines the column, even on a hit) into VA of the pipe, so that VA[4:21] in the pipe contain the address from the cache. Also prevent both map and storage automata from starting (which prevents ring buffer pipe entries from being allocated to these as well). FDMiss should always be true when dVA←Vic is true.

FDMiss — "Force dirty miss" forces each cache reference to miss and store the victim, even if not dirty. Misses caused by FDMiss do not cause Hold (***details***).

UseMcrV — Use McrV as victim and McrNV as next victim for all cache misses instead of Victim and NextV from VNV.

McrV — The two-bit victim, or cache column used on a miss, when UseMcrV is true.

McrNV — The two bit next-victim when UseMcrV is true.

DisBR — "Disable base registers" prevents base registers from being added to *D* in computing VA and prevents BR from being written.

DisCF — "Disable cache flags" forces cache flags to read out as zeroes and prevents them from being written.

DisHold — "Disable Hold" unconditionally prevents hold and BLretry from occurring.

NoRef — Disable storage references.

WMiss — Wakeup fault task on every miss.

ReportSE' — Don't wake up fault task after (correctable) single errors.

NoWake Never wakeup fault task.

During normal operation every bit in Mcr should be 0, except possibly *ReportSE'*, if correctable errors are not being monitored. It is illegal to load Mcr while references are in progress (Changing DisHold is known to cause problems).

System Initialization

System initialization must get the map initialized as desired and the cache in agreement with the map. Initialization firmware should allow for cache rows containing several entries for the same address, which might occur after power up or after running diagnostics.

There are many ways to carry out this initialization. One is as follows:

1. Set NoWake and DisHold true in Mcr, so the fault task won't disturb initialization, and so that BeingLoaded conditions won't cause trouble.
2. Clear TestSyndrome.
3. Load the map as desired. Clear the cache as discussed in the Map section. After this the cache will be empty and *Ref* and *Dirty* in map entries will be smashed.
4. Reload the map as desired.
5. Read FaultInfo to kill any pending wakeup for the fault task.
6. Setup Mcr for normal activity (0 or ReportSE').

Testing

This section outlines the order in which parts of the memory system can be tested, so that only a few new components are involved at each step.

VA, Adder, BR's

The first step is to set NoWake, FDMiss, and DisBR to true. Now processor references will deposit Mar in VA of pipe entry 0 (in the emulator), or of every other pipe entry (in other tasks), so that this part of the pipe can be tested (LongFetch← allows all the VA bits to be tested). Next, setting DisBR false, loading BR's, and making more processor references will allow BR's and adder to be tested.

Cache Address Storage

Then set NoWake, FDMiss and UseMcrV to true and use McrV and McrNV to select one column of the cache at-a-time. Each processor reference will store its VA into that column, and into the pipe, and will read out the old VA into the next ring buffer pipe entry (as the victim because FDMiss is true). This allows the VA bits in the address memory to be initialized and tested. The column number in Pipe2 should read back the value in McrV in this case.

Above, address memory values are read using FDMiss, then VA is checked in the pipe entry created for the victim. A simpler method of reading any address section VA is as follows: Turn on DisBR, UseMcrV, and dVA←Vic. On processor references, the cache entry addressed by Mar[6:11] (the row) and McrV (the column) will then have its VA[7:21] written into VA[7:21] of the pipe entry for the reference.

The flag bits in the address section can be directly tested using B←Pipe5 and CFlags←A. These functions operate on the cache entry addressed by the row of the last reference and column of the hit or victim on a miss. Since the IFU or another task could have issued the last reference, these functions are realistically limited to initialization and checkout, where the last reference is known. Normally these will be used with UseMcrV and FDMiss true in Mcr, so McrV will select the column.

B←Pipe5 also reads V and NV from the selected row. CFlags←A won't work if DisCF is true, and B←Pipe5 will read zeroes for all four flags in this case.

CFlags←A requires that Mar data continue without glitching during the preceding instruction as well. This means that data originating in RM or T must not have been loaded during either of the two previous instructions (else a glitch might occur when the multiplexor switched from the bypass to direct path) and that no higher priority tasks may intervene between the two instructions. Issuing CFlags←A in both instructions is the easiest way to drive Mar continuously for two cycles.

Cache Data Storage

Next, initialize the cache address section VA's and flags so that the cache data section can be tested. To do this turn off FDMiss while leaving on NoWake, dVA←Vic, UseMcrV. Initialize the address section to a convenient range of virtual addresses by Store←'s to each munch with appropriate McrV values. In the instruction after each reference, write the flags to WP = false, BeingLoaded = false, Vacant = false with CFlags←A.

At the end of this setup, the address section will be loaded and have write access to the desired virtual addresses. Hence, Fetch←'es and Store←'s to these VA's will not miss, and will access the 4k of cache data memory, which can thus be systematically tested.

Map

Next, turn off UseMcrV, leaving only NoWake turned on and use Map← to test the map. At the end of this test initialize the map, say, to map virtual addresses into corresponding real addresses.

Main Storage

Then finally the storage can be accessed and tested with fetches and Store←. FDMiss can be used to force storage references.

Fault Reporting

NoWake can be turned off and methods similar to the above can be used to test fault reporting.

IOFetch←, IOStore←, Fast IO Busses

Special hardware will be needed to test these.

Error Correction

In normal operation TestSyndrome contains 0 and Syndrome, written by the error corrector, should be 0 if no error was corrected or detected. For test purposes, TestSyndrome can be loaded with any non-zero value and one bit disables error correction altogether. If there are no storage failures, TestSyndrome should wind up in Syndrome after a storage read.

The error-corrector, MemError, ECfault, ReportSE', and fault reporting can be tested using TestSyndrome.

The LoadTestSyndrome function causes TestSyndrome to be loaded from DBuf. This should normally be done after a Store←, as follows:

```
TaskingOff;
Store←RMAddr, DBuf←T; *DBuf←data for TestSyndrome
LoadTestSyndrome;
TaskingOn;
```

TaskingOff is required because an intervening higher priority task might change the contents of DBuf.

Instruction Fetch Unit

The instruction fetch unit, or IFU, decodes a stream of bytes from memory into a sequence of 8-bit opcodes and operands using a writeable decoding memory, and presents the results to the processor for efficient interpretation. The next section contains an overview of IFU function, supplemented by details in later sections.

Read this chapter with Figure 12 in front of you.

Overview of Operation

The IFU handles four independent instruction sets. Opcodes are 8-bit bytes, which may be followed in memory by 0, 1, or 2 operand bytes. Hence, the total length of an instruction is 1, 2, or 3 bytes. The first operand byte is called α, the second β.

The term *PC* refers to the displacement of an opcode byte from the *codebase*, which is BR 31. PC's are 16-bit items, where 0:14 are an unsigned word displacement relative to the codebase, and bit 15 selects the byte. In other words, codebase points at a 32k segment of virtual memory; a PC selects a byte in this segment.

> Since the IFU's PC is only 16 bits, overflowing either end of the code segment causes wraparound. This programming error is not detected by the hardware.

For Alto compatibility reasons, we currently have the following kludge. Instruction sets 0 and 1 treat byte 0 in the selected word as bits 0:7, 1 as bits 8:15; instruction sets 2 and 3 treat byte 0 as bits 8:15, 1 as 0:7. Eventually, this may be changed so that all instruction sets use 0 for the byte in 0:7 and 1 for 8:15.

The IFU is started by first selecting an instruction set (InsSetOrEvent←B function) and then loading the F-level PC (PCF←B function). The IFU then starts fetching the byte stream starting at the word BR[31] + PCF[0:14], byte PCF[15], from the cache and prepares opcodes for interpretation by the processor.

Bytes from the cache then march through the IFU pipeline beginning with the F and G full-word buffer registers on the MemD board; single bytes from F/G then move into J or H on the IFU board. InsSet[0:1] and the opcode byte in J address the decoding memory, IFUM, a 1024-word x 24-bit (+3 parity) RAM containing the information in the table below. Although IFUM is writeable, it will normally be loaded with the microprogram and not subsequently changed (Diagnostics are, of course, an exception.).

Table 17: IFUM Fields

Name	Size	Contents
Length'	2	Opcode length: 1, 2 or 3 bytes (0 length is illegal).
TPause'	1	The opcode is of type *pause*.
TJump'	1	The opcode is of type *jump*.
IFaddr'	10	TNIA[4:13] of the first instruction to be executed in interpreting this opcode (TNIA[14:15] from the IFUJump in the exit of the previous opcode).
RBaseB'	1	RBase initialization, discussed below.
MemB	3	MemBase initialization, discussed below.
Sign	1	Operand sign extension, discussed below.
Packedα	1	Packed α, discussed below.
N	4	Operand encoded in the opcode, discussed below.

Length', TPause', TJump', Sign, Packedα, and *N* are used by the IFU to prepare operands and to sequence correctly to the next opcode; *IFaddr'* is passed to the control section; and the processor uses *MemB* and *RBaseB'* to initialize MemBase and RBase when the microcode for the opcode commences.

Length' determines the number of operand bytes; α for a two or three-byte instruction will be in H, while β for a three-byte instruction will be in F/G, when the assembled instruction is ready to proceed. The assembled instruction and α then drop into the M level.

IFUJump[n] (see "Control Section") transfers control to the starting instruction for the opcode assembled in M, where TNIA[4:13]←IFaddr, TNIA[14:15]←n (n is 0 to 3) is the location of the entry instruction. A 4-long entry vector, rather than a single starting address, can be utilized for faster execution, as discussed later. *IFaddr* may be overruled by a trap address when appropriate.

At t_0 of the starting instruction, the processor initializes RBase to (*RBaseB'*)' and MemBase to 0..MemBX[0:1]..MemB[1:2] if MemB[0] = 0, or to 34_8 + MemB[1:2] if MemB[0] = 1. MemBX is interpreted as a stack pointer to a 4-entry stack with 4 base registers in each entry, and MemB[1:2] in IFUM select a particular base register from the current entry. The MemBX kludge may reduce computation on procedure call/return, as discussed later. Other information about the opcode and α are copied into the X level.

Instructions that implement the opcode then reference operands in sequence using the A←Id, RisId, or TisId operations discussed in "Processor Section" or the IFetch← operation discussed in "Memory Section," which read operands from the X level. The operand sequence delivered by the IFU in response to ←Id is as follows:

Table 18: Operand Sequence for ←Id

Type	*Length*	*Packedα*	*Sequence*
--	0	--	Illegal
Jump	1	--	*Length, Length, Length, . . .* *Packedα, sign,* and *N* determine jump displacement.
Jump	2	--	*Length, Length, Length, . . .* *Packedα* and *N* are unused; *sign* extends the sign of α for the jump displacement.
Jump	3	--	Illegal
Regular	1	--	*N* if *N* ne 17_8, *Length, Length, Length, . . .* *Packedα* and *sign* are unused.
Regular	2	0	*N* if *N* ne 17_8, α, *Length, Length, . . .* α is sign-extended if *sign* = 1.
Regular	2	1	*N* if *N* ne 17_8, α[0:3], α[4:7], *Length, Length, . . .* *Sign* is unused.
Regular	3	0	*N* if *N* ne 17_8, α, β, *Length, Length, . . .* α is sign-extended if *sign* = 1.
Regular	3	1	*N* if *N* ne 17_8, α[0:3], α[4:7], β, *Length, Length, . . .* *Sign* is unused.
Pause	x	x	Same as regular

Regular and *pause* opcodes have an optional 4-bit operand *N* that is delivered first (*N* isn't supplied when $N = 17_8$). This is followed by α and β, if they exist; α is sign-extended when *sign* = 1 or split into two 4-bit nibbles if *Packedα* = 1. Subsequently, ←Id delivers *Length*. For jumps, all of these operands are consumed in computing the jump displacement, and ←Id delivers *Length*.

The normal opcode references all of its N, α, and β operands; however, except on three-byte opcodes, the IFU hardware does not *require* that these operands be referenced--the processor could exit to the next opcode without reading all the operands, if that was desirable for some reason. However, for opcodes of length 3, the processor must consume the α byte with ←Id (both α[0:3] and α[4:7] if Packedα = 1) before dispatching the IFU to the next opcode.

The types of opcodes are distinguished as follows: A *pause* has no successor, and the IFU must be restarted with PCF←B before the next IFUJump. A *regular*'s successor is the byte following its last operand; a *jump*'s successor is determined by adding a displacement to the current PC as follows:

If *Length* = 1, then *Sign.Packedα.N* forms a six-bit signed displacement. In other words, the jump is to any byte in the range PC-40_8 to PC+37_8.

If *Length* = 2, then *Packedα* and *N* are unused; the jump displacement is α, if *sign* is 0, or sign-extended α, if *sign* is 1.

A jump with *Length* = 3 is illegal.

The IFU pipeline follows the instruction stream and fills up when it is five or six bytes ahead of the current opcode. When a *pause* opcode is recognized, further memory references are not made. When a *jump* opcode is recognized in J, the IFU discards any bytes in F, G, and H and refills these pipe levels with bytes along the jump path.

The B←PCX' function reads PC (inverted) for the current opcode. Note that PCF←B does not affect the value of PCX; B←PCX' continues to read the displacement of the current opcode, which does not change until an IFUJump is done.

An opcode that *conditionally* jumps can be encoded in IFUM with type either *jump* or *regular*. If encoded as type *jump*, when the condition is false, the program must issue PCF←B to restart the IFU at the fall-through address. Similarly, if *regular*, PCF←B must be issued to restart at the jump address.

The *Length* argument delivered by ←Id after other operands have been referenced is useful in conditional jump calculations. Note that the fall-through address for a conditional jump is Length + PCX, so:

```
T←(Id)-(PCX')-1;  *Id = Length for type jump
PCF←T;
IFUJump[0];
```

restarts the IFU at the fall-through address for type *jump*.

Following PCF←B, the IFU flushes its pipeline; it is illegal for either the instruction containing PCF←B or the one immediately after it to do an IFUJump, but any subsequent instruction can issue an IFUJump; however, the processor will spin uselessly at the IFU "NotReady" trap until the fifth cycle after PCF←B (earliest) or later (longer opcodes, cache misses, Mar traffic).

Table 19: IFU FF Decodes

Name — **Action**

IFUReset — Halt and clear the IFU pipeline and clear errors, testing features, and BrkPending (i.e., BrkIns).; Reschedule condition and instruction set are not cleared.

B←IFUMLH' — Read the high-order IFUM word, InsSet, and IdCnt onto B (low-true) as follows:

Field	*B bits*	
IdCnt	0:2	Count of ←Id's since start of opcode
InsSet	3:4	Instruction set number
Packedα	5	Packed α
IFaddr'	6:15	Starting address

IFUMLH←B — Load the high-order IFUM word from B (t_1 to t_3), where the Packedα and IFaddr fields are in the same form as B←IFUMLH'.

IFUMRH←B — Load the low-order IFUM word from B (t_1 to t_3) as follows:

Field	*B bits*	
Sign	0	
IPar.0	1	Odd parity over N, MemB[1:2], and IFAD[0:1]
IPar.1	2	Odd parity over IFAD[2:9]
IPar.2	3	Odd parity on Packedα, Sign, Length', MemB.0, RBaseB', TPause, and TJump
Length'	4:5	Instruction length (low true)
RBaseB'	6	1-bit RBase initialization
MemB	7:9	3-bit MemBase initialization
TPause'	10	Type pause (low true)
TJump'	11	Type jump (low true)
N	12:15	4-bit operand

B←IFUMRH' — Read IFUM fields in the same format as IFUMRH←B (inverted).

PCF←B — Load PCF at t_3, clear and restart the pipeline.

B←PCX' — Read PC for the currently executing opcode (inverted).

BrkIns←B — Load BrkIns from B[0:7] at t_3, and set BrkPending (ill-defined unless the IFU has been reset). BrkIns replaces the next opcode loaded into J; then BrkPending is cleared. BrkIns also addresses IFUM on IFUMLH/RH← and B←IFUMLH'/RH'.

InsSetOrEvent←B — Select the instruction set i = 0 to 3 at t_3; junk io also uses this function. A following PCF←B starts the IFU interpreting using the new instruction set.

Reschedule — Trap the second following IFUJump at the Reschedule trap location; Reschedule is testable by a branch condition (emulator only); *not* cleared by IFUReset. The trap instruction is executed as though it were the first instruction of the second opcode and ←Id and IFUJump will work as though the next opcode were in progress.

RescheduleNow — Trap the next IFUJump at the Reschedule trap location; in other respects like Reschedule.

NoReschedule — Turn off the Reschedule trap and branch condition.

IFUTest←B — Load the test-control register from B (load with 0 or do IFUReset when not testing) as follows:

Field	*B bits*	
TestFG	0:7	Substituted for cache data
TestFGParity	8	Substituted for cache parity bit
TestFault	9	Substituted for memory fault signal
TestMemAck	10	Substituted for memory MemAck signal
TestMakeF←D	11	Substituted for memory MakeF←D signal
TestFH'	12	enable FHCP and t_1 when IFUTick executed
TestSH'	13	enable SHCP and t_2 when IFUTick executed
TestEn	14	test enable
WakeEn	15	enable junk wakeups

IFUTick — Tick the IFU's clock once according to TestFH and TestSH in the IFUTest register.

The IFUJump Entry Vector

An IFUJump[n], encoded in the JCN field of the instruction, sends control to an address partly determined by the IFU and partly by the IFUJump clause. The four possible targets of an IFUJump are called an "entry vector".

An opcode leaves its results in *one of several convenient forms* agreed to by convention, then chooses an entry instruction in its successor with IFUJump[n], where n = 0 to 3. Every opcode in the instruction set must have an entry vector of the same length. Careful choice of forms may reduce execution time by one cycle for many opcodes.

A true branch condition (FF-encoded) with IFUJump prevents starting the next opcode. For example, IFUJump[2,condition] sends control to the next opcode's entry 2, if condition is false, or entry 3, if condition is true. However, no other IFU activities associated with starting the new opcode take place when condition is true, so entry 3 is executed in the context of the opcode that did the IFUJump[2,condition]; however, the processor initializes RBase and MemBase as though the next opcode were starting, so this part of the state is lost. Thus, at a cost of one entry instruction in every opcode of an instruction set, it may be possible to shorten the execution time of some opcodes using a conditional exit.

An opcode with common and uncommon exit cases, for example, can exit with IFUJump[2,condition], where entry 2, the common case, starts the next opcode, while entry 3 is reached for the uncommon case. Since IFUJump loads Link with .+1, entry 3 can either Return, to execute more code associated with the uncommon case, or it can do something more explicit, if an appropriate convention is followed by all opcodes.

The following example shows how an instruction set with four opcodes (Push, Add, Store, and Jzero) is implemented using a four-long entry vector. The opcodes in this example deal with the stack like Mesa opcodes do, and we expect the entry conventions used by Mesa to be similar to those in this example.

```
%Entry
    0:  Stk[StkP] holds top-of-stack, T holds garbage
    1:  T and Stk[StkP-1] hold previous top of stack, Stk[StkP] garbage, Md top-of-stack.
    2:  T and Stk[StkP+1] hold top-of-stack, Stk[Stkp] previous top of stack.
    3:  Results in same form as entry 2, but restart IFU at NewPC = (Id)-(PCX')-1
%

*Push the memory location pointed to by N.
Push:   Fetch←Id, T←Stack&+1, IFUJump[1];
        Fetch←Id, T←Stack&+1←Md, IFUJump[1];
        Fetch←Id, StkP+2, IFUJump[1];
        T←(Id)-(PCX')-1, StkP+1, Return;

*Replace the top two stack entries by their sum.
Add:    T←Stack&-1, Branch[.+2];
        Stack←Md;
        T←Stack&-1←T+(Stack&-1), IFUJump[2];
        T←(Id)-(PCX')-1, StkP+1, Return;

*Store the top-of-stack into the memory location pointed to by N and pop the stack.
Store:  Store←Id, DBuf←Stack&-1, IFUJump[0];
        Stack←Md, Branch[Storex];
```

```
            Store←Id, DBuf←T, IFUJump[0];
            T←(Id)-(PCX')-1, StkP+1, Return;
Storex:     Store←Id, DBuf←Stack&-2, IFUJump[2];

*Pop the stack and branch if the top-of-stack was zero, else fall through
*This opcode is of type jump.
Jzero:      Pd←Stack&-1, Branch[ZTest];
            Pd←Md, Stkp-1, Branch[ZTest];
            Pd←T, Branch[ZTest];
            T←(Id)-(PCX')-1, StkP+1, Return;
ZTest:      T←Stack&-1, IFUJump[2,ALU=0];
*Return here when the jump doesn't take.
            T←Stack&-1, PCF←T;
            IFUJump[2];
```

Push thus requires 1 execution cycle; Store and Add take either 1 or 2 cycles depending upon the entry point; Jzero takes 2 cycles when the jump takes or 9 cycles when the opcode falls through (because the IFU isn't ready until the fifth cycle after PCF←B).

Although every opcode in an instruction set must have an entry vector following the same conventions, it is not necessary that the vector be four-long. In the above example, a single-entry scheme would probably use the entry 2 convention followed above. In that event, Push, Add, Store, and JZero would require 2, 1, 2, and 3 cycles (common case), respectively, compared to 1, 1 or 2, 1 or 2, and 2 or 3 cycles for the four-entry scheme above.

Since Mesa requires about 79 IFU entries for its 256 opcodes, the cost of the second entry in the vector is between 0 and 79 locations, and 79 locations each for the third and fourth entries. Since Mesa (sans BitBlt) is implemented by about 600 instructions using entry vectors of length 1, a vector of length 2 scheme would require ~640, length 3 ~719, and length 4 ~798 instructions. The implementor of an instruction set should decide when the additional locations expended for larger entry vectors are no longer worth the additional speed.

The overall effect of this approach on the Mesa emulator is roughly that average execution time reduces from about 5 to about 4.2 cycles/opcode (ignoring cache misses); microstore increases from about 600 locations to about 800. In other words, about 180 locations are expended to increase speed about 20%.

Note: IFU trap locations discussed below must also be entry vectors that follow the same convention.

Timing Summary

From the detailed timing discussion at the end of this chapter, the following generalizations about IFU timing can be drawn:

> Assuming no misses and no delays because the processor uses Mar, IFUJump will successfully dispatch to the entry instruction of the next opcode on the fifth cycle after PCF←B if the new opcode either is one byte long or is two bytes long and starts at an even byte; otherwise it will succeed on the sixth cycle.

A *jump* opcode causes a 3 cycle gap in the IFU pipe. The effect of the gap would be a 3 cycle delay if each opcode were executed in exactly one cycle. However, the gap can overlap with extra cycles taken on the *jump* opcode itself or either of the two preceding opcodes. As usual in timing considerations, a 3-byte opcode counts as two normal opcodes.

If a long stream of regular one-byte opcodes is being executed by the processor at the fastest possible rate (one instruction/opcode), and if the IFU neither misses nor faults nor waits for the processor's use of Mar or the cache, then it will always have the next opcode ready for IFUJump. If the IFU waits one cycle for the processor to use Mar, it will shortly fill its pipe again, so scattered Mar references by the processor will not result in IFU NotReady.

If a long stream of regular two-byte opcodes, each of which has an α but no N (This is the worst case.), is being executed by the processor at the fastest possible rate (one instruction/opcode), and if the opcodes in the stream start at the even bytes in words, and if the IFU neither misses nor faults, and if the processor never uses Mar, then the IFU will give 25% NotReady. Each cycle in which the processor uses Mar adds one cycle of delay. If the opcodes in the stream start at the odd bytes in words, then the processor will get NotReady 40% of the time.

Three-byte opcodes are not as bad as two-byte opcodes because, in the worst case, the processor cannot reference both α and β in less than 2 instructions. Hence, a stream of three-byte opcodes has timing approximately the same as a stream in which each three-byte opcode is replaced by a one-byte opcode followed by a two-byte opcode.

Mar traffic may be an important timing factor if many opcodes finish in one or two cycles. Whenever the processor is making a reference, the IFU cannot use Mar, and the IFU must make one reference for every two bytes in the instruction stream. Note that if a processor reference is held, the IFU will also be prevented from making references (but the IFU is not prevented from making references when ←Md is held).

Use of MemBX and the Duplicate Stk Regions

An early Mesa emulator on Dorado model 0 required about 80 cycles for a procedure call and 40 cycles for a return; we estimated that about 40% of execution time would be spent in call/return.

Since about 70% of all calls return before calling any other procedure, if a caller's base registers and stack were left untouched, then this information would neither have to be saved during the call nor restored during the return in most cases.

The hardware that supports this idea consists of the MemBX register, pointing at one of four blocks of 4 base registers each, and StkP, pointing at one of four stacks of 64 registers each. During a procedure call, StkP and MemBX may be advanced by 1 region, leaving the caller's state intact; if the callee makes nested calls, then eventually the MemBX and Stk regions would be exhausted and some would have to be saved and (eventually) restored. However, if the callee returns without too many nested calls, then its caller's

state would still be intact.

We have not constructed examples that use this idea, and at this time it is doubtful that much improvement will result from it.

Traps

The IFU may trap for not ready, reschedule request, map faults, cache data errors, and IFUM parity errors. When a trap condition occurs, the IFU substitutes a trap address for IFaddr on the next IFUJump. Hence, the next IFUJump sends control to one of the entries in the trap vector.

Locations assigned to these trap vectors are given in "Control Section"; note that each instruction set has independent trap locations.

Each trap vector is dispatched into by IFUJump exactly as though it were an opcode. *B←PCX' reads the PC of the opcode that would have been executed if the trap had not occurred (in every case except NotReady).*

The relative priority of traps is as follows: IFUM parity error is highest, then NotReady, reschedule, cache data parity error, and map fault.

The NotReady trap occurs whenever the IFU does not have both an opcode and its associated operands (α, β) ready for the processor. Since PCX is invalid, the trap microcode *must* wait for the IFU to become ready. The following code sequence will work for all instruction sets that do not use a conditional exit:

```
NotReady:
        FreezeBC, IFUJump[0];
        FreezeBC, IFUJump[1];
        FreezeBC, IFUJump[2];
        FreezeBC, IFUJump[3];
```

For the sample instruction set given earlier, which uses entry 3 as a conditional exit, the following sequence would be appropriate:

```
NotReady:
        T←Stack&-1, IFUJump[2];          *Convert case 0 and 1 exits to case 2, which
        T←Stack&-1←Md, IFUJump[2];       *is usually best for the next opcode
        IFUJump[2];
        T←(Id)-(PCX')-1, StkP←StkP+1, Return; *Resume the opcode which didn't really exit
```

If the IFU detects bad parity on any read of IFUM, the IFUJump to the opcode affected by this parity error will trap to the IFUM parity error trap location.

The IFU will trap at the cache data parity error location, if it detected invalid parity on any byte sent by the memory system. PCX will always correctly point at the opcode that would have been executed next had the trap not occurred; however, the opcode and operands pointed at by PCX are not necessarily the ones that suffered the parity error. This occurs because the pipe has continued ahead of PCX. The most confusing case occurs when the opcode following PCX was a jump; in this case the opcode fetched by the jump may have caused the parity error, in which case PCX+/- jump displacement is limited to the range

$PCX\text{-}400_8$ to $PCX+377_8$.

The IFU will hold an IFUJump in the cycle prior to a cache data parity error or IFUM parity error trap.

Note that IFUReset must be given after an IFUM or cache data parity error and before restarting the IFU.

The Reschedule function is used by io tasks to request service by the emulator; it is also used when continuing an opcode which previously experienced a fault. The IFU will honor this trap request on the *second* IFUJump after it is executed, as discussed in a later section. The RescheduleNow function is like the Reschedule function, but the IFU honors it on the *first* IFUJump after it is executed, rather than the second.

An IFU fetch may experience a map fault. The memory system does not report IFU map faults to the fault task. Instead, it signals the IFU that a map fault has occurred, and the IFU passes this indication through its pipeline. Eventually, the IFUJump that would have sent control to the opcode affected by the map fault will instead transfer to the map fault trap vector.

> Although IFU map faults are not reported to the fault task, the fault task must be careful to pass over any pipe entries that were created by IFU map faults when it is woken for some other reason.
>
> Erroneous bytes fetched after a *pause* or *jump* opcode might cause map faults, but the IFU discards these before they reach the end of the pipeline, so the processor is never informed. Consequently, erroneous references interfere with processor memory activity and delay the IFU's efforts to refill its pipe on a *jump*, but don't have any disastrous effect.

An IFU fetch may experience single or double storage failures. Unlike map faults, these are reported to the fault task just as on processor fetches. The memory system pipeline will finish loading the cache munch just as though the data were ok, and the cache entries will have valid byte parity. The IFU will continue running just as though no error had occurred.

However, the fault task will be woken soon enough that it will run before the IFU's F register is loaded with a byte from the bad munch. Hence, the fault task will run before the emulator can possibly execute an IFUJump to the byte that suffered the error.

For a recoverable error, the fault task can simply carry out some logging action and block; no harm will occur because the IFU will actually have gotten valid data, and the cache will contain valid data. For an irrecoverable error, the fault task must clear the bad cache munch and use the RescheduleNow function to trap the next IFUJump to code for dealing with the irrecoverable error.

> Erroneous bytes fetched after a *pause* or *jump* opcode might suffer irrecoverable errors. The fault task has no reasonable way to distinguish these from bytes really in the instruction stream, so it will cause a Reschedule trap anyway.

Remark

Although independent trap vectors for each instruction set are probably inessential, performance should be better when the NotReady trap, which occurs frequently, is distinct for each instruction set. This allows the various IFUJump exits to be transformed into the form most likely to be convenient for the next opcode.

The other traps could have been implemented to use a common trap for all locations. This would be more economical for IFUM and FG parity error traps, if these simply result in an uncontinuable crash when running system microcode. However, different trap vectors for each instruction set are probably more convenient for Reschedule and Map fault traps, which have to save the state of the emulator currently running.

In any case, reserving locations for these traps costs at most 5 traps * 4 instruction sets * 4 entries/trap = 100_8 locations, and realistically is much less than this because many instruction sets will not need 4 entries and there will probably be fewer than 4 instruction sets concurrently active.

IFU Reset

The processor can reset the IFU by executing the IFUReset function. This clears all IFU error conditions, prevents further IFU memory references, clears the BrkIns← feature discussed earlier and the test features discussed later, and generally puts the IFU in a clean and operable state. The Reschedule feature is not affected by IFUReset.

IFUReset should be executed after power-on to get the IFU shut off. A single IFUReset will make the IFU passive with respect to operating the rest of Dorado. However, the IFU might not be operable until a second IFUReset is executed because of a pathological condition (If BrkIns is loaded and Testing is true, then the first IFUReset will clear Testing but not BrkIns; a second IFUReset is required to clear BrkIns in this case).

If the IFU has any outstanding memory references pending at the time the first IFUReset is executed, those references will complete and disturb the top part of the IFU pipeline. A second IFUReset must be issued after these references have all finished prior to reading or writing IFUM. If the second IFUReset is executed 36 or more cycles after the first, then it will for sure completely reset the IFU.

> The worst case is when a miss has just started the storage pipeline with an IFU reference in the cache address section. In this case the IFU reference does not enter the storage pipeline until the 8th cycle and then takes 28 cycles to complete.

IFUReset should be executed *prior* to using BrkIns←. It should also be executed *after* reading or writing IFUM (to reset the BrkPending condition that is still lurking).

Rescheduling

Io tasks request service from the emulator by first indicating a request in some way (Presently an RM location is used as a 16-bit table in which 1's indicate requests.), then executing the Reschedule function, and finally blocking. The IFU and the processor store the reschedule condition in flipflops which remain set until the NoReschedule function turns them off.

The next IFUJump after Reschedule transfers to the entry vector for the opcode as usual; the second IFUJump transfers to the Reschedule trap location. IFUJump's that experience a NotReady trap are not counted.

The entry vector at the reschedule trap location is entered *as though it were the next opcode*. When Reschedule is used by io tasks to request the wakeup of another process, this fact is unimportant. However, the other use of Reschedule is in continuation from map (and other) faults. In this application, the reschedule trap will wind up restoring the IFU

state by executing an appropriate number of ←Id's and eventually branching back to the instruction that experienced the fault. The continuation method is discussed later.

Opcodes which might execute for a long time, such as block transfer and BitBlt, must check for rescheduling explicitly, and the (emulator only) Reschedule branch condition makes this check easier. If such opcodes did not check for rescheduling, then service to the io device might be postponed for too long.

The reschedule flipflops are not cleared by IFUReset, so the NoReschedule function must be executed as part of system reset.

When the reschedule trap vector is entered, the IFU is in a paused state, and PCF← is needed to restart the IFU at the continuation address.

Breakpoints

BrkIns←B implements debugging breakpoints straightforwardly. The idea is that a one-byte opcode, BrkP, is used to transfer control to a debugger while saving emulator state needed to continue later, and another opcode, Continue, is used to continue from breakpoints (For Mesa, BrkP and Continue are special cases of Xfer.(?)).

BrkP may be substituted for any opcode in a program. The debugger gets control when BrkP is executed, saves state, and eventually can execute Continue to restore state from values saved by BrkP.

Continue first restores registers, then loads BrkIns with the opcode for which BrkP was substitued; then it uses PCF←B to restart the IFU at the breakpoint. The IFU will then start running; the first opcode fetched will again be the BrkP opcode, but the contents of BrkIns will be substituted for the one fetched from memory, and the program will continue correctly.

Without BrkIns←B the debugger would have to simulate the broken opcode before continuing at the following opcode, which would be harder. The example below shows a code sequence for the final part of Continue.

```
Continue:
        . . .
        IFUReset;                       *Stop future IFU fetches and clear pipe
        T←41C;
        Cnt←T;
        IFUReset, Goto[.,Cnt#0&-1];     *Reset after previous IFU fetches complete
        BrkIns←Opcode;                  *Load opcode which BrkP replaced
        PCF←BreakAddress;               *Restart IFU at address of BrkP
        Noop;                           *No-op required after PCF← before IFUJump
        IFUJump[0];                     *Resume program
```

Note: IFUReset is required before BrkIns←, even when an opcode of type Pause is in progress.

Reading and Writing IFUM

In addition to its function related to breakpoints, BrkIns←B is used to address IFUM when reading or writing that memory.

When IFUM is loaded, it is addressed by the instruction set InsSet[0:1] and BrkIns. The data must remain on B for two cycles, so tasking must be disabled and the instruction following the one with IFUMLH/RH← must put the same data on B. If this data comes from RM or T, the register must not have been loaded in the cycle preceding the IFUMLH/RH← (because the bypass logic will change the B select from Pd or Md to RM or T, possibly glitching data on B). The following subroutines illustrate loading and reading back IFUM.

```
WriteIFUM:
        IFUReset;                       *Stop future IFU fetches and clear the pipe
        T←41C;
        Cnt←T;
        IFUReset, Goto[.,Cnt#0&-1];     *Reset after previously issued fetches complete
        InsSetOrEvent←RMaddr0;          *Load 2 instruction set bits forming IFUM address
        BrkIns←RMAddr1;                 *Load 8 opcode bits forming IFUM address
        TaskingOff;                     *Must ensure no B glitch below
        IFUMLH←RMdataHi;                *Write high part of IFUM
        B←RMdataHi;                     *Keep data good a little longer (mustn't glitch)
        IFUMRH←RMdataLo;                *Write low part of IFUM
        B←RMdataLo, TaskingOn;          *Keep data good a little longer
        IFUReset, Return;               *Clear BrkIns

ReadIFUM:
        IFUReset;                       *Stop future IFU fetches and clear the pipe
        T←41C;
        Cnt←T;
        IFUReset, Goto[.,Cnt#0&-1];     *Reset after previously issued fetches complete
        BrkIns←RMaddr1;                 *Load 8 opcode bits forming IFUM address
        InsSetOrEvent←RMaddr0;          *Load 2 instruction set bits forming IFUM address
        Noop;                           *Two instructions must elapse after loading BrkIns
                                        *one after loading InsSet (?Two noops after loading InsSet
                                        *might be better since this is a tight path?)
        RMdataHi←IFUMLH;                *Read IFUM into RM.
        RMdataLo←IFUMRH;
        IFUReset, Return;               *Clear BrkIns
```

Continuing from Processor Faults

Saving and restoring the state of an interrupted program requires some cleverness not only in the IFU, but also in the Control, Processor, and Memory sections. The emulator might fault for a data error, map fault, or stack overflow/underflow; for io tasks, stack overflow/underflow is impossible and map faults will probably be illegal, so only data error faults are legitimate. The discussion here will concentrate on map faults, though the same approach could be used for other fault conditions as well.

The fault task must use as few instructions as possible so that io tasks won't be preempted for too long. The minimum is to copy all pipe entries that contain memory faults into RM or Stk buffers, preserve DBuf, and save the emulator's TPC; the fault task must itself deal with data error faults by io tasks; it then restarts the emulator at a trap address. The emulator microprogram then saves the rest of the emulator state and deduces the nature of the fault(s) using methods discussed in "Memory Section".

The emulator fault microcode first saves ALU branch conditions and task-specific registers, then other information of interest. The saved information is stored where the Mesa (or whatever) program can get at it; then the trap microcode restarts Mesa at a trap procedure that will service the map fault (probably swap in a page from the disk); eventually, state will be restored and the opcode that faulted will be resumed at the instruction that faulted.

The IFU state may be saved via B←IFUMLH' and B←PCX'. B←IFUMLH' reads the current instruction set and IdCnt from B[0:4]; B[5:15] are IFUM bits which are not of interest when saving the state of the program, so the tricky code sequence given earlier for reading IFUM is not required. B←PCX' reads the current PC.

The 3-bit counter, IdCnt, keeps track of how many ←Id's have been done; to avoid overflowing this counter, no more than 7 ←Id's should be done when executing any opcode. This is one (harmless) restriction on coding emulators. The other is that *emulators never map fault on the instruction after a dispatch* (BDispatch←B, BigBDispatch←B, or Multiply); this can be assured by doing ←Md prior to or concurrent with any dispatch.

Sample microcode for saving emulator state is as follows:

```
%Must first save the volatile branch conditions; Overflow and Carry won't change unless an arithmetic
ALU operation is executed, so saving them can be deferred.  T, the first item saved, is written into the
RM region reserved for Save using the change-RBase-for-write FF decode.
%
Save:     FreezeBC,  DblGoto[ALUls,ALUge,ALU<0];
ALUls:    SavedT←T;
          T←0C,  Goto[SaveBC];
ALUge:    SavedT←T,  DblGoto[ALUgr,ALUeq,ALU#0];
ALUgr:    T←1C,  Goto[SaveBC];
ALUeq:    T←2C;
*Have a code, 0, 1, or 2, in T indicating the state of the ALU<0 and ALU=0 branch conditions.
SaveBC:   SavedALULEZ←T;                  *Save the branch condition code
          T←Pointers;                     *T←MemBase, MemBX, and RBase
          T←T OR (100000C);               *Make negative
          RBase←RBase[SaveRMRegion];
*Now choose two numbers such that their sum produces the correct ALUcry and Overflow branch
*conditions.
          SavedPointers←T,  MemBase←SaveBaseReg,  DblGoto[Cry,NoCry,Carry];
Cry:      DblGoto[CryOvf,CryNoOvf,Overflow];
NoCry:    DblGoto[NoCryOvf,NoCryNoOvf,Overflow];
CryOvf:   SaveA1←100000C;
          SaveA2←100000C, Goto[SaveRest]; *Numbers such that SaveA1+SaveA2 produces
                                          *Overflow and Carry result
NoCryNoOvf:
          SaveA2←0C,  Goto[.+2];
CryNoOvf: SaveA2←1C;
          SaveA1←177777C,  Goto[SaveRest];
NoCryOvf: SaveA1←77777C;
          SaveA2←77777C,  Goto[SaveRest];

SaveRest:
          SavedPCX←not(PCX');
          T←not(IFUMLH');                 *Read IdCnt and InsSet in IFUMLH[0:4]
          SavedIdCnt←LDF[T,0,2];
          T←T and (14000C);
          T←RSH[T,2];
          SavedInsSet←T+(100000C);        *Set up word for InsSetOrEvent← below
          . . .                           *Code to save rest of state (all easy)
```

Sample microcode for continuing is given below:

```
Resume:
        RSked←(RSked) or (ResumeBit);   *Indicate reschedule trap should branch to
                                        *the Resume1 location
        . . .                           *Restore all processor registers except T, Cnt, RBase,
                                        *and MemBase.
        InsSetOrEvent←SavedInsSet;      *Restore the IFU instruction set number.
        RescheduleNow;                  *Make the next IFUJump trap at the reschedule
                                        *trap address.
        PCF←SavedPCX;                   *Restart IFU at address of the opcode that faulted
        Noop;                           *No-op required after PCF← before IFUJump
        Cnt←SavedIdCnt, IFUJump[0];     *Continue execution at the reschedule trap
                                        *location (ReSked below)

ReSked:
        Pd←(RSked) and (ResumeBit);
        Branch[Resume1,alu#0];
        . . .

        A←Id;
Resume1: Goto[.-1,Cnt#0&-1];            *Reissue the appropriate number of ←Id's to put
                                        *the IFU in the state it was in at the fault.
        . . .                           *Repeat the Fetch← or Store← that faulted using a
                                        *convenient base register, and restore the base register
                                        *(complicated code here needs careful thought).
        Cnt←SavedCnt;                   *Restore Cnt
        T←SaveA1;
        Pd←T+SaveA2;                    *Restore Carry and Overflow branch conditions.
        T←SavedT, Goto[.+2];            *Restore T register
*Below, WakeUp takes on the 3rd cycle after it is issued, so two instructions following the WakeUp are
*usually executed, unless higher priority tasks intervene.  Hence the five instruction loop is repeated
*until no higher priority task intervenes.
COK:    RBase←ContRBase, At[COKloc];    *Set RBase to the block of registers used by the
                                        *continuation microcode.
        BDispatch←SavedALULEZ;          *Dispatch to 0, 1, or 2 in table based on
                                        *ALU>0, ALU<0, or ALU=0.
        WakeUp[ContTask];               *Wakeup the special task reserved for continuation.
        Link←SavedLink, At[ConTab,0];   *Restore Link and ALU>0
        Pd←Not(Pointers←SavedPointers), Goto[COK];
        Link←SavedLink, At[ConTab,1];   *Restore Link and ALU<0
        Pd←Pointers←SavedPointers, Goto[COK];
        Link←SavedLink, At[ConTab,2];   *Restore Link and ALU=0
        Pd←(SavedPointers) xor (Pointers←SavedPointers), Goto[COK];

*The special restart task needed for continuation
ContinueInit:
        RBase←RBase[SavedTPC];          *Initialization code for the task
ContinueTaskLp:
        Block, T←COKloc';               *T←emulator TPC' if no intervening task switch
        TaskingOff;
        RdTPC←0C;                       *Read the emulator's TPC
        T←T xor (Link);
        Link←SavedTPC, Branch[.+2,alu#0];
        LdTPC←0C;                       *Restart emulator at saved continue address
        TaskingOn, Branch[ContinueTaskLp];
```

IFU Testing

The IFU test control register is loaded by the IFUTest←B function; when not testing, this register should contain 1, and it is loaded with 1 by the IFUReset function. IFUTest.15 disables the periodic wakeup request to the Junk task discussed in the "Slow IO" chapter;

when IFUTest.15 is 0, the junk wakeups occur 60 times/sec and are dismissed by any IFUTest← function.

IFUTest.14 (TestEn) enables IFU test mode; it is illegal for this bit to change from 0 to 1 when the IFU is active because, if this occurred in the same cycle that an IFU memory reference was issued, then the IFU would pollute the Mar bus indefinitely, making the memory system unusable by the processor.

The test features aim at two situations. First, they allow the IFU clocks to be controlled by a program, so a diagnostic can slowly step the IFU pipeline through its stages. Secondly, they allow data supplied by a diagnostic to be substituted for signals that would otherwise come from the memory system. This allows the IFU to be tested in the absence of the memory system, which allows scope probes to be inserted easily and decouples IFU problems from memory system problems.

The TestFH' and TestSH' bits in the IFUTest register enable the first-half-cycle and second-half-cycle clocks, respectively, which will occur between t_2 and t_4 of the cycle *after* the one issuing the IFUTick function. Thus, the IFU can be stepped through a PCF←B function as follows:

```
IFUTest←TestEn;
IFUTick;
PCF←value;
```

where PCF←value is just an example--any other IFU function or an IFUJump could be used instead.

The IFU's memory interface is simulated by the TestFG, TestParity, TestFault, TestMemAck, and TestMakeF←D bits in IFUTest. Memory references are not issued by the IFU when TestEn is true. TestFG and TestParity are substituted for the FG byte and parity bit from the memory system; the other signals are control signals sent by the memory system in response to IFU references. They are supposed to work as follows:

MemAck occurs at t_2 of a cycle in which the IFU makes a reference at t_1, iff the memory system accepted the reference; if the memory system was busy and did not accept the reference, then *MemAck* does not occur, and the IFU should repeat its reference. The absence of *MemAck* serves approximately the same purpose for the IFU that Hold serves for the processor.

MakeF←D occurs at t_1 of a cycle in which the memory system loads F at t_3; in the event of a map fault, *MakeF←D* occurs at t_1 of the cycle in which the memory system would have loaded F at t_3 if the map fault had not occurred. The IFU can try to start a reference at t_1, even though it has an unfinished reference in progress. The memory system will accept the reference iff MakeF←D occurs; otherwise, it will refuse the reference. In other words, the IFU's second reference starts at t_1 iff the first reference will deliver data at t_3.

Fault is concurrent with (?) *MakeF←D* and indicates that the IFU reference experienced a map fault.

In other words, a memory reference can be simulated with the IFU test feature by (1) ticking the IFU through a cycle in which it makes a reference; (2) ticking the *TestMemAck* response of the memory system with IFUTest←B and IFUTick; (3) ticking *TestMakeF←D*; (4)

ticking with *TestFG* and *TestParity* holding simulated memory data.

Details of Pipe Operation

The IFU is a six-stage pipeline, starting with words fetched from memory, and ending with opcode starting addresses delivered to the control section and operands delivered to the processor. The levels are named: F, G, H, J, M and X. Each level has a data-valid bit indicating whether or not it contains something useful.

PCF, PCJ, PCM, and PCX are PC's for the corresponding pipe levels (except that PCF is a word PC rather than a byte PC). PCF, PCM, and PCX are independent of each other since jumps and PCF← may result in these all being different; PCJ is related to PCF by the number of valid bytes in the F/G/H levels; the hardware also uses PCFG, which contains PCF plus the number of valid bytes in the F/G levels. Operationally, F/G are a FIFO in which PCF is the write pointer, incremented as words are fetched from the cache, and PCFG is the read pointer, incremented as bytes are moved from F/G into J/H. Note that there is no PCH because PCH would equal PCJ+1.

Pipe control is straightforward in principle. The F and G levels are 16-bit registers filled from the cache. Following PCF←B, if there is space in the pipeline for another word, the IFU will start a reference at t_1 of any cycle in which the processor is not using Mar (so as many as 2 IFU references can be outstanding). Cache words are stored in F at t_1, then dropped into G at t_2; bytes drop into H at t_3 or J at t4; there are bypass paths to get bytes directly from F/G into J when H is invalid. As the processor executes opcodes, F and G become invalid, and the IFU refills them from memory automatically. This continues until the IFU is reset by the processor, or encounters a *pause* opcode.

> The F and G registers are physically located on the MemD board. The four bytes in F/G are inputs to a multiplexor controlled by the IFU, and the multiplexor output is sent across the backplane to the IFU. BrkIns[0:7] or IFUTest[0:7] replace F/G data when using breakpoints, reading/writing IFUM, or using IFU test features.
>
> While following the opcode stream, a *jump* will invalidate data in F. However, if a reference is in progress and F has not yet been filled by the memory system, then the IFU will invalidate the data when it arrives and restart the next reference immediately. In other words, the IFU cannot abandon the useless fetch; it must wait for it to finish and discard the result.

The J and H levels are one byte wide. For one-byte opcodes it is possible to consider H and J as independent levels of the pipe; however for two or three-byte opcodes, it is appropriate to consider J/H as a single level in which J holds the opcode and H holds α.

If J is invalid, then it will be loaded from the next opcode (which may be in G, F, or H according to various conditions) at an even clock (t_0) and H will be loaded from the byte after the opcode (which is always in G) at the following odd clock (t_1); if the byte after the opcode isn't ready, it will drop into H at the next odd clock after it is ready. The InsSet and J registers address IFUM and IFUM outputs reveal whether the byte in H is α (Length = 2 or 3) or the next opcode (Length = 1).

The conditions under which the M level can be loaded from J are that M is invalid (or about to become invalid) and:

Length = 1 -or-
Length = 2 and H is valid -or-
Length = 3 and H is valid and either F or G is valid.

If these conditions are met, then the M level is loaded (t_2) with information from IFUM and with α, if Length = 2 or 3. If Length = 3, then β will drop from G into H (t_3).

If *Length* < 3, then the H/J level is now free to work on the next opcode. If Length = 1 and the next opcode happens to be in H, then H will drop into J at the same time (t_2); otherwise, J will be loaded from the next opcode in F/G when it is ready.

When the processor does an IFUJump[n], level M presents information needed by the next opcode as follows:

IFaddr is TNIA[4:13] for the IFUJump;
MemBase is set to 0.MemBX.MemB[1:2] or 34_8 + MemB[1:2];
RBase is set to 0 or 1;
N, Sign, Length, Packedα, and α are loaded into the X level;
β is loaded into the M level if Length = 3.

Referencing IFU operands with A←Id, TisId, or RisId affects the IFU in two ways: it causes the IFU to advance to the next item of Id, and for a 3 byte instruction when α is taken (α[4:7] when Packedα = 1) it causes β to drop from M to X, freeing M for the next instruction.

IFetch← also uses Id, as discussed in memory section, but does not advance the IFU to the next item of Id.

For a one or two-byte opcode, it is permissible for the processor to do an IFUJump before referencing any operands with ←Id; this will advance normally to the next opcode. However, for a three-byte opcode the processor must reference all of α, so that β drops into X, before doing an IFUJump.

When a *pause* or *jump* is recognized, the IFU may already have filled the F and G levels erroneously (i.e., 4 bytes ahead). These levels are flushed and refilled along the jump path.

Timing Details

This section discusses timing details of the IFU pipeline assuming that all IFU references hit in the cache and are never deferred for processor references.

First case: Restart IFU at even byte

t0: An instruction with PCF←FOO is started, where FOO is *even*.

t2: F, G, H, J, and M levels are made invalid; IFUJump will trap at NotReady.

t3: Reference the word containing FOO.

t5: Reference word containing FOO+2.

t7: Load F with data from the FOO reference; reference the word containing FOO+4.

t8: Load the first byte from F into J; load G from F; F becomes invalid; start reading the IFUM entry for J.

t9: Load the putative operand byte from G into H; G becomes invalid; load F from the FOO+2 reference.

t10: Distinguish 5 cases below.

FOO is a one-byte regular opcode

t10: Load M from IFUM; IFUJump will now succeed; load J from H (FOO+1); load G from F (FOO+2 and FOO+3); F and H become invalid; start reading the IFUM entry for J.

t11: Load H from G (FOO+2); load F from FOO+4 reference.

t12: -- (The FOO+1 opcode would pop into M if IFUJump were done at t10.)
IFU is quiescent; F has two useful bytes, G one byte, J/H has two bytes; M level is ready and waiting for IFUJump.

FOO is a two-byte regular opcode

t10: Load M from IFUM and M[α] from H; IFUJump will now succeed; load J from F (FOO+2); load G from F (garbage and FOO+3); F and H become invalid; start reading the IFUM entry for J.

t11: Load H from G (FOO+3); G becomes invalid; load F from FOO+4 reference; reference the word containing FOO+6.

t12: Load G from F; F becomes invalid.

t15: Load F from the FOO+6 reference; now quiescent.

FOO is a three-byte regular opcode

t10: Load M from IFUM and M[α] from H; IFUJump will now succeed; load G from F (FOO+2 and FOO+3); H and F become invalid; J goes to special state (β in H).

t11: Load H from G (FOO+2 = β); load F from the FOO+4 reference; now quiescent.

t12: -- (The FOO+2 byte would pop from H into M[β] if IFUJump were done at t10.)

FOO is a one-byte jump opcode

t10: Load M from IFUM; IFUJump will now succeed; J, H, G, and F become invalid.

t11: Discard the FOO+4 reference; reference the first word along the jump path.

t13: Reference the second word along the jump path.

t15: Load F from the first word along the jump path.

t16: Load J from F, etc.

FOO is a two-byte jump opcode

t10: Load M from IFUM and M[α] from H; IFUJump will now succeed; G and F become invalid; J and H are in a special jump state, computing the jump address.

t11: Discard the FOO+4 reference; reference the first word along the jump path.

t12: J and H become invalid.

t13: Reference the second word along the jump path.

t15: Load F from the first word along the jump path, etc.

Second case: Restart IFU at odd byte

t0: An instruction with PCF←FOO is started, where FOO is *odd*.

t2: F, G, H, J, and M levels are invalid; IFUJump will trap at NotReady.

t3: Reference the word containing FOO.

t5: Reference word containing FOO+1.

t7: Load F with data from the FOO reference; reference the word containing FOO+3.

t8: Load the second byte from F into J; F becomes invalid; start reading the IFUM entry for J.

t9: Load F from the FOO+1 reference.

t10: Distinguish 3 cases below (and the one and two-byte jump cases which are not repeated below).

FOO is a one-byte opcode

t10: Load M from IFUM; IFUJump will now succeed; load J from F (FOO+1); load G from F (garbage and FOO+2); F becomes invalid; start reading the IFUM entry for J.

t11: Load H from G (FOO+2); G becomes invalid; load F with the FOO+3 reference; reference the word containing FOO+5.

t12: Load G from F; F becomes invalid.

t15: Load F from the FOO+5 reference; now quiescent.

FOO is a two-byte opcode

t10: Load G from F (FOO+1 and FOO+2); F becomes invalid.

t11: Load H from G (FOO+1); load F with the FOO+3 reference.

t12: Load M from IFUM and M[α] from H; IFUJump will now succeed; load J from G (FOO+2); load G from F; F and H become invalid; start reading the IFUM entry for J.

t13: Reference the word containing FOO+5; load H from G (FOO+3).

t17: Load F with data from the FOO+5 reference; now quiescent.

FOO is a three-byte opcode

t10: Load G from F (FOO+1 and FOO+2); F becomes invalid.

t11: Load H from G (FOO+1); load F from the FOO+3 reference.

t12: Load M from IFUM and M[α] from H; IFUJump will now succeed; H becomes invalid; J is in a special state (β in H).

t13: Load H from G (FOO+2); load G from F (FOO+3 and FOO+4); F becomes invalid; reference the word containing FOO+5.

t17: Load F from the FOO+5 reference; now quiescent.

Slow IO

The slow io facility allows data transfers between the processor and any of up to 256 independently addressed io registers. It is intended that the slow io facility will be used to load and read control information associated with high speed io devices (> 20 x 10^6 bits/sec), which will then use the fast io system for their data transfers. Low speed devices (< 20 x 10^6 bits/sec) will use the slow io bus for all phases of their operation. Very slow or polled devices may be driven directly from an emulator.

Device controllers for Dorado interact with the processor by exchanging data over a 16-bit bidirectional bus IOB ("input/output bus"). There may be a total of up to 256 *io registers* in all controllers connected to a single system. The unique 8-bit device numbers assigned to particular devices or uses that appear in every system are discussed in subsequent chapters and summarized in the table below.

Table 20: IO Register Addresses

Number	*Name*	*Comment*
10	DskC	Disk control register
11	DskMA	Disk muffler control
12	DskD	Disk FIFO data
13	DskF	Disk format RAM
14	DskT	Disk tag register
15	EthD	Ethernet input or output data
16	EthC	Ethernet control and status
366	Mixer	DDC Mixer
367	PClock	DDC pixel clock
370	DStatus	DDC muffler and OIS data
372	MiniMixer	DDC MiniMixer
373	DWTFlag	DDC word task control
374	DHTFlag	DDC horizontal task control
375	HRam	DDC horizontal waveform control
376	NLCB	DDC next line control block
377	Statics	DDC debugging control

Input/Output Functions

In most cases, a task will need to do many sequential io operations to the same io register. The 8-bit task-specific register TIOA holds the device address being referenced by each task.

TIOA is loaded at t_2 from B[0:7] by the TIOA←B function, or TIOA[5:7] can be loaded from FF[5:7] while preserving TIOA[0:4] by the TIOA←small constant function. Pd←Input, Pd←InputNoPE, or Output←B functions can be issued in the instruction immediately following the one that loads TIOA.

Most input registers include odd byte parity with IOB data. The Pd←Input function reads IOB data and checks parity. The Pd←InputNoPE function reads IOB data without a parity check; this is useful when determining whether a device exists (IOB has bad parity if a

nonexistent register is selected). The enabling and timing of parity error halts is discussed in the "Errors" chapter.

The Output←B function sends 16 bits of data with parity to the io register selected by TIOA. Many controllers check the parity and report parity errors as part of their status.

The tasks reserved for standard peripherals are given in the table below.

Table 21: Task Assignments

Number	*Name*	*Comment*
0	EMU	The emulator
1	JNK	Junk task (awakened every 32 μs)
2	CON	Special task for restarting emulator after faults
5	DHT	Display horizontal task (?)
6	EIT	Ethernet input task
7	EOT	Ethernet output task
10	SIM	Task simulator
12	DSK	Disk io
14	DWT	Display word task (?)
15	FLT	The fault task

IO Opcodes

The Mesa instruction set has two opcodes for dealing with the slow io system:

INPUT:

```
TIOA←α, Stkp←Stkp+1;
Stack←Input, IFUJump[0];
```

OUTPUT:

```
TIOA←α;
Output←Stack&-1, IFUJump[0];
```

These opcodes allow a Mesa program to have full access to the io system. The intent is that these instructions will be used to set up registers in firmware-driven devices, and do all the service required by polled slow devices (e.g. the keyboard). In many cases, the use of an INPUT or OUTPUT instruction is not sensible (doing io to a device normally driven by firmware, for example), but the capability should prove useful for testing and diagnostics.

Wakeup, Block, and Next

The "Control Section" chapter discussed task switching, and the material which follows is an elaboration of that discussion.

Note that a task for which a wakeup request is issued at t_0 cannot commence its next instruction until t_4; i.e., at least two cycles elapse after a wakeup before the next instruction is executed. The task then runs until it does a Block; in order to avoid an erroneous extra wakeup, *the task must lower its wakeup request at least one cycle before issuing Block.*

Consequently, an io device may turn off its wakeup request according to one of three strategies:

The first is to turn off the request when Next becomes equal to its task number; in this case the wakeup request is lowered at t_0 of the first instruction executed for the task, and it *must not block until the second instruction to prevent an erroneous second wakeup.* The special situation in which Next is invalid ("Next Lies") must be dealt with by device controllers that do this. This situation occurs as follows:

Suppose that a task blocks with the following instruction:

```
Branch[Loop], Fetch←Address, Block;   *Fetch next word
```

This generates Switch and the task in Bnt is broadcast over the Next bus. If the Fetch← causes hold and Bnt < Ctask, then no task switch will occur. However, the Next bus is incorrectly broadcasting Bnt. Since hold occurs after t_1, there is insufficient time to change the Next bus back to Ctask in this case.

Consequently, controllers using Next detect "Next Lies" and disable any actions that would otherwise be performed when it occurs.

A pathological lockout problem should be noted: Since task T's wakeup request was lowered at t_2 when Next = T was noted at t_0, the Next Lies condition will (correctly) result in repeating the held instruction at t_2; however, some task of lower priority than T may erroneously execute at t_4. This might be a problem if some high demand task of higher priority is coded so that it always creates Next Lies (say, by doing Block and immediate ←Md in the instruction after a Fetch←).

Another consequence of "Next Lies" is that IOAtten may be incorrect when "Next Lies" is occurring. Consequently, branch on IOAtten is illegal during an instruction that blocks and might cause hold.

The second strategy monitors TIOA becoming equal to a particular device value. In this case the wakeup request is lowered at t_0 of the second instruction following a wakeup, and the task *must not block until the third instruction.*

The third strategy waits for some Output←B or Pd←Input operation to reset the wakeup condition. This would reset the condition at t_3 or t_5 of the Output←B

instruction, and the wakeup would be lowered at t_4 or t_6; in this case the task *must not block until the third or fourth instruction after the Output←B or Pd←Input* to avoid an erroneous wakeup. The exact requirement depends upon the io controller--the disk controller, for example, lowers its wakeup request at t_4 and can block in the third instruction after Output←B, while the display controller horizontal task lowers its wakeup request at t_5 and can block in the fourth instruction.

If loops naturally run for at least three instructions, use of TIOA is more economical than use of Next because TIOA decoding is mandatory in any case, while Next is needed only for short loop devices, devices that use the fast io system, and devices that drive the SubTask lines.

SubTasks

When an io device sees Next becoming equal to its task, it can (optionally) present a two-bit SubTask number as well.

The processor, control, and memory sections clock SubTask into flipflops at t_0. The processor OR's SubTask [0:1] into RBase[2:3] and into MemBase[2:3]. This allows the same firmware to control several identical io devices concurrently--each device, represented by a SubTask, gets its own RM region with 16 RM locations and its own pair of MemBase registers; if only SubTask[0] is driven, then two RM regions and four MemBase registers are available to each subtask. Note that the 16 change-RBase-for-write functions do *not* OR SubTask into the changed address, so they cannot be used; also, if RBase is read by the processor the value read out has SubTask OR'ed in. However, the 16 change-RSTK-for-write functions *do* work.

Note also that when the debugging processor (Baseboard microcomputer or Alto running Midas) asserts the *Freeze* signal, the affect of the subtask on RBase[2:3] is disabled, but subtask continues to affect MemBase[2:3].

In the memory section, the task and SubTask that issued an IOFetch← is bussed to fast output devices with data from storage. The device receiving the data identifies itself by means of this information. IOStore←'s are handled similarly.

A task presenting SubTask signals generally *must Block at the same location each iteration* since there is only a single TPC value for all of the SubTasks. Hence, the full generality of tasking is unavailable--the microcode for these tasks must be coded as though the wakeup mechanism were a priority interrupt.

Illegal Things IO Tasks Must Not Do

(1) It is illegal to Block in an instruction that does B←ExternalSource, where ExternalSource is anything except one of the sources on the IFU board. This restriction is needed so that the emulator will be able to do arithmetic on B←PCX'.

(2) The IOAtten branch condition is illegal in an instruction that Blocks and might be held, because NextLies might occur, as discussed above.

(3) A task may not Block on an instruction that might be held, if its wakeup request might be dropped at t_0 of the instruction. If this occurred, the instruction might inadvertently be repeated before the Block took effect.

(4) It is illegal to Block with TaskingOff in force.

(5) A task must not Block until one cycle after its wakeup request is turned off.

Fast IO

The fast input/output system provides high-bandwidth data transfers between storage and io devices. Transfers occur in units of one munch (= 16 words); the addresses of the 16 words must be i, $i+1$, ..., $i+15$, where $i \bmod 16 = 0$. One word is transferred every clock, for a peak bandwidth of 640 x 10^6 bits/second. A fast device is also interfaced to the slow io system, from which it receives its control information, since there is no way for the device to communicate directly with the processor using the fast io system.

A single transaction of the fast io system transfers exactly one munch. Successive transactions are completely independent of each other, whether they involve the same or different devices, as far as the io system is concerned. The only relationship between transactions is that storage references of two transactions occur in the order that they were issued.

Each fast io transaction is initiated by an IOFetch← or IOStore← reference coded in ASEL. Once this instruction has been executed, the transaction proceeds without further interaction with the processor (except for fault reporting). The transaction itself involves a storage reference, and *transport* of the data between main storage and the device. In the case of a fetch, transport happens at the end of the reference, after the munch has been error-corrected. For a store, transport happens at the beginning of the reference, in parallel with mapping the VA and starting the storage chips. As a result of this difference, the transport for a fetch may overlap or even follow the transport for a following store.

Transport

The device is only concerned with the transport of the data, and has no way of knowing exactly how or when the storage reference take place. The transport happens in 16 clocks, each transporting a single word using the Fin bus (IOFetch←'es) or Fout bus (IOStore←'s). The two busses are independent, and transport can be happening on both of them simultaneously.

The two busses have much in common. Both have Task and Subtask lines, on which the memory presents the task and subtask involved in the transport about to begin and a Next signal used for synchronization. The Fout bus has a Fault line which is high at the time the last word of the transaction is delivered if there was a memory fault during the fetch (other than a corrected single error).

Both data busses are 18 bits wide: 16 data bits, numbered 0..15, and two byte partiy bits, numbered 16 (bits 0..7) and 17 (bits 8..15). The parity bits have the same timing as the data bits. A device is invited to check the parity of data on Fin, and is required to generate parity for data on Fout.

Wakeups and Microcode

The normal interface between a device and its task involves one wakeup for each munch transferred. The device must keep track of the number of wakeups it has issued, since data may not arrive from storage for several microseconds, but there is no way to stop the

data from arriving once the task has started the memory reference.

Typical microcode for a fast output device is given in the "Display Controller" chapter.

Latency

Suppose that the highest priority fast io task issues its wakeup request at t_0; then it will execute its first instruction at t_4. Some other task can cache fault with clean victim in the cycle starting at t_0, and another task can cache fault with dirty victim in the cycle starting at t_2. The first reference gives rise to one storage reference and the second to two storage references; each of these three storage references takes 8 cycles to handle, so the fast io reference will not begin for about 24 cycles. From the time it begins until the last data word is delivered to the device is 23.5 cycles, for a total of 47.5 cycles, to which 2 cycles must be added for the time between the wakeup and the first executed instruction. In this situation, the transport is not finished until 49.5 cycles after the wakeup. Lower priority tasks are delayed by an additional 8 cycles for each reference which might be made by a higher priority task.

The above is one possible worst case. Another is the execution time of higher priority tasks; a wakeup might be delayed by sum of the longest normal execution of the fault task and of other higher priority tasks. The fault task execution time is presently unknown.

A store reference is slightly better, since its transport is finished 8 cycles after the reference starts, for a total latency of 40 cycles.

> All these numbers assume that a reference can be started every 8 cycles. If successive references are to 4k modules, however, they can happen only every 13 cycles, and the calculations must be adjusted accordingly. Also, data is returned from a 4k module 3.5 cycles later.

Disk Controller

This chapter describes the Dorado disk controller, which uses the Slow IO system to control up to four CalComp Trident disk drives. Either the $80x10^6$-byte T-80 or the $300x10^6$-byte T-300 drives can be used. An extension of the controller onto a second logic board (not designed) would allow control of up to 31 disk drives; alternatively, duplicating the present controller (with different TIOA, task, and muffler assignments) would allow independent control of four additional drives.

Keep Figure 13 in view while reading this chapter.

The disk controller uses task 14_8 and the first five values of the TIOA addresses in block 10_8 - 17_8 (The Ethernet controller, on the same logic board, uses two of the other three.). Either the task or TIOA block can be modified by changing a SIP component on the logic board. TIOA assignments are as follows:

10_8 Output←B to control register
11_8 Output←B muffler control and Pd←Input to read muffler
12_8 Pd←Input to read FIFO or Output←B to write FIFO data
13_8 Output←B to format RAM
14_8 Output←B to tag register

The controller is interfaced to the disk drives by a *daisy chain cable* bused to all drives and by an independent *radial cable* to each drive. The radial cables contain the following signals:

data line (bidirectional, differentially driven)
data clock (from drive, differentially driven)
subsector/index line (from drive)
selected line (from drive)
select line (from controller)
sequence line (from controller, controlled by the baseboard for drive 0 and grounded for other drives (???))
two VCC lines and scope trigger (from controller)

The daisy-chain cable contains the following signals:

16 control "tags" driven by the controller and received by the selected drive
9 error and status signals from the drive as follows:
CylOffset'
ReadOnly'
NoTerminator
HeadOvfl'
SeekInc'
DevCheck'
NotOnLine
NotReady
Index' (unused)

The controller OR's the NoTerminator error (which means that the daisy-chain cable isn't terminated) into the NotOnLine error; the other error indications are discussed later.

Disk Addressing

The disk system is accessed through a many level addressing scheme. First a particular disk drive is selected. Then a data surface or *head* and a *cylinder* are selected (5 surfaces, 815 cylinders on a T-80). Each cylinder is further divided into *sectors* which consist of *blocks*.

Firmware may control the following parameters:

Sector size (1378 words max., limited by 4-bit subsector counter)
Number of blocks within one sector (1 to 4)
Block sizes (2 to 2684 words)

Note: Various limits on the sizes of blocks and sectors will be discussed. The processor interface allows a six-bit subsector counter of which only four bits are presently implemented, and this is the most significant length limit at present (1378 words). If the subsector counter were enlarged to six bits, then the block size limit imposed by the error correction algorithm (2684 data words) would apply. We are, however, unlikely to find any of these length limits significant unless we enlarge the memory page size to 4096 words.

Because record formats are flexible, firmware can adjust the controller to system needs. For example, initial systems may want an Alto-compatible disk format; a Diablo 31, dual 31, or Diablo 44 disk can be emulated on 100 cyclinders of a T-80, while an Alto Trident disk is implemented on the remaining cylinders.

Sector Layout Considerations

Each block within a sector can be either read, written, or checked. However, once any block is written, later blocks in that sector cannot be read during that disk revolution (Later blocks can (?) be read on subsequent disk revolutions). Reading or writing must start with the first block in the sector and continue; since check bits are stored at the end of each block, the entire block must be read to verify its data or correct errors; however, one does *not* have to read or write subsequent blocks in the sector. After a write-block operation is started, the controller inhibits writing later blocks within a sector without a specific "OK" from the firmware.

Our general plan is to use the first block in a sector as a *header* identifying the disk address; all headers will be written when a disk pack is initialized; subsequently, the disk task compares the header with the disk address it thinks it is accessing. The header not only provides a useful safeguard against positioning errors but also allows faster sector determination when switching to a new drive, as discussed later.

The second block might identify information stored in the sector (e.g., the Label block in Alto Trident format). The third block might be the data block. The fourth block could hold reference, backup, or archiving information. All of these choices are a matter of programming convention.

Feasible sector layouts are determined by several considerations. First, each disk drive generates 117 *subsector* pulses/revolution. The disk controller has a *subsector counter* for each drive that is initialized to N when an index pulse is received from the drive; it then counts down to -1, generates a sector pulse, and reinitializes itself. The firmware can specify N (0 to 17_8) independently for each disk drive and thus create 117/(N+1)

sectors/revolution. If this division leaves any remainder, then there will be several unused subsectors at the end of the cylinder.

Various delays must be provided at the beginning and end of each block to allow for electrical and mechanical tolerances within the disk drive. To define a sector format, one simply needs a summary of "words lost" for each block:

Total words/cylinder =	10,080
Words lost for the 1st block in a sector =	32
Words lost for successive blocks =	14
Required gap at end of sector =	14

"Words lost" for each block include 2 words of error detection and correction (32 bits of ECC code) which are always added at the end of the data written, 1 word of trailing zeroes (to ensure that all bits are sent before the write electronics is turned off), 1 word for the disk controller to execute the read/write command, and some delay words required by the disk drive for mechanical and electronic delays.

Note that the quantization of cylinders into subsectors allows a sector size to be specified in units of 10,080/117 = 86.1 words/subsector.

For the Alto Trident format there is a 2-word Header block, 10-word Label block, and 1024-word data block; total words lost for disk formatting is 32 for the first block, 14 for the second, 14 for the third, and 14 at the end of the sector; altogether, this requires 1110 words/sector, so 1110/86.1 = 13 subsectors/sector are required, and 117/13 = 9 sectors/revolution.

Using this kind of analysis, reasonable sector layouts on the T-80 are as follows:

29	sectors	of	256	data words	each,
16	sectors	of	512	data words	each, or
9	sectors	of	1024	data words	each.

Table 22: T-80 Specifications and Characteristics

Capacity	82.1 million 8-bit bytes unformatted
Transfer rate	9.67 x 10^6 bits/sec (= one 16-bit word/1.65 ms)
Cylinder positioning time	6 ms cylinder to cylinder maximum (3 ms typical) 30 ms average 55 ms maximum
Rotational speed	3600 rpm (16.66 ms/revolution)
Sector length selection	12-bit increments through jumpers on sector board
Densities	370 cylinders/inch 6060 bits/inch max. recording density
Disk pack characteristics	IBM 3336-type components 5 recording surfaces plus 1 servo surface 815 cylinders/surface
Operating methods	Modified frequency modulation recording Linear positioning motor with cylinder following servo
Mechanical specifications	Size - 17.8" wide x 10.5" high x 32" deep Weight - 230 lbs.
Error rate	Recoverable: 1 error/10^{10} bits Irrecoverable: 1 error/10^{13} bits Positioning: 1 error/10^6 seeks
Pack start/stop time	20 sec start time 20 sec stop time (with dynamic braking)
Controls and indicators	Ready Indicator Off = disk not spinning Flashing = spinning up/down On = Ready Fault Indicator Start/Stop switch Read-only switch Degate switch (inside the drive; takes disk off-line for testing)

General Firmware Organization

This section gives a general overview of how the disk controller firmware is organized; more detailed descriptions follow later.

The disk drive generates *subsector* and *index* pulses on one line in the radial cable; the controller distinguishes these according to pulse width. In the normal Idle loop, the controller looks only at these pulses from the different drives. A four-bit counter for each drive counts down subsector pulses and generates *sector* pulses. Upon either a sector or an index pulse from the *selected* drive, the controller generates a disk task wakeup. The disk task then either increments (sector wakeup) or zeroes (index wakeup) its firmware sector counter, clears the wakeup condition, checks for a new command, and blocks.

Because there are no hardware sector counters, the disk task must maintain a sector counter itself; this implies that the rotational position is generally unknown on all deselected drives.

When first selecting a drive, there are two strategies for determining the sector position: (1) Wait for an index wakeup, at which time the sector position becomes known; (2) Wait for a sector wakeup and then read the sector number stored in the header block (This can only be done if the disk is not moving to a new cyclinder.). The most efficient strategy appears to be a combination: Select the drive and start a seek to the correct cylinder; if an index wakeup arrives before the seek is finished, then the sector position is synchronized with no loss of time. If the seek finishes first, then read the next header to determine the sector number.

When a new disk operation is noted, firmware will perform the following steps:

Execute a drive-select command, if the drive differs
Load the sector size only if different & block until index
Load the format RAM only if word count or commands differ
Execute a seek command only if the cylinder differs
Execute a head select command
Block until, at a sector wakeup, the *next* sector is the one wanted
Load the appropriate transfer command into the control register
Block until the next sector wakeup

At the start of the *next* sector, the controller will become active and sequence through commands under control of the format RAM and two sequence proms (one for reading, one for writing).

The sequence proms define what operations the controller must go through, and the format RAM contains all parameters that might change from one implementation to another. Actual commands for the Trident disk are stored in the format RAM along with count values such as words/block, words of ECC, and words of delay before some operation; the commands are loaded into the tag register and executed by the controller during the transfer.

Once a transfer has started, the disk task will be woken according to the number of words in the FIFO, and it will send or receive the appropriate number of words. Read and compare operations are performed by firmware, as well as detecting checksum errors at the end of reading. During writing, firmware must provide one word of sync bits (#201 standard, #001 for Alto Trident emulation) followed by the specified number of words for that block (the controller will append 2 words of checksum). During read, the controller will look for, and discard, the first word of sync bits, then firmware must accept the specified number of words for that block, followed by two words of checksum to be discarded, followed by the ECC remainder to be used for error detection/correction.

Task Wakeups

The controller may wakeup the disk task for many conditions; the disk task must detemine the cause and take appropriate action, which must in some way cause the wakeup to go away.

In general, there are two ways to determine the wakeup condition: read the wakeup condition, or assume the condition knowing the state of the disk task (which implies the state of the controller). When expecting a sector or index wakeup, the disk task must test carefully to count sectors reliably, but in the middle of word transfer operations, it will assume the wakeup reason to minimize overhead. The various conditions are as follows:

IndexTW, SectorTW, SeekTagTW, RdFifoTW, and WrFifoTW; these wakeup conditions are detailed in the "Muffler Input" section.

Control Register

The control register is a collection of flip-flops defining the state of the controller; on Output←B to the control register, B is interpreted as follows:

B[5]	Clear EnableRun
B[6]	Set DebugMode
B[7]	Set BlockTillIndex
B[8:9]	Operation for first block of sector, where the operations are: 0 = Done 1 = Write 2 = Read and check 3 = Read
B[10:11]	Operation for second block of sector, as above.
B[12:13]	Operation for third block of sector, as above.
B[14:15]	Operation for fourth block of sector, as above.

EnableRun determines whether the controller is active at all. It is initially cleared by IOReset, and can only be set by completing the loading of the format RAM (see below).

DebugMode allows the controller to be exercised by diagnostics when no disk is present; in this case, diagnostic firmware provides fake disk bit-clocks and data. The flip-flop is cleared by DisableRun.

BlockTillIndex can be set to disable disk task wakeups until an index pulse is seen from the disk. It is cleared by an index wakeup.

A request for a sector transfer is initiated by loading bits 8 and 9 of the control register with a non-zero value. Then the controller will wait until the *next* sector pulse to set the "Active" flip-flop and execute the transfer. Once a transfer has been started, it may be aborted by loading a new value into the control register twice. The first will clear the Active flip-flop, and the second will load the control register (When Active, the control register is enabled for shifting commands rather than loading of io data.).

Format RAM

The format RAM is a 16-word by 12-bit register that holds commands and delay counts used by the controller during a transfer; on a format RAM output, words within the RAM have the following meaning:

Addr	*Description*	*Default Value*
00	Word count of the first block	0001
01	Word count of the second block	0007
02	Word count of the third block	0377
03	Word count of the fourth block	0000
04	Tag command for a read operation	0104
05	Tag command for a write operation	0204
06	Tag command leading to a read or write operation	0004
07	Tag command to reset disk	0000
08	Word count to write zeroes before writing the 1st block of a sector	0033
09	Word count to write zeroes before writing the sucessive blocks	0006

10	Word count to wait before reading the 1st block of a sector	0011
11	Word count to wait before reading the sucessive blocks	0002
12	Word count of ECC words plus one	0002
13	Word count of 2	0001
14	Word count of 1 (minimum count)	0000
15	Not used	0000

Notice that the format RAM contains both word counts and tag commands. *Word counts are 1 less that the desired count.* Tag commands will be loaded into the tag register (see below) and then used as a "control tag function" by the Trident disk. The values in the right column are those used for the Alto Diablo emulation format. Notice that all but the first 4 values are determined by characteristics of the drive being used. The meaning of the tag command values can be found in the "Tag Register" section.

The format RAM is addressed in two ways. During a transfer, sequence PROMs move data from the RAM into either a tag register or a count register. At other times, the Dorado may address the RAM with the *RAM Address counter*, which is zeroed when the control register is written; writing the format RAM uses the current address and then increments the counter. Loading the last word in the format RAM turns on the EnableRun flip-flop allowing normal disk control activity. The format RAM may be read via the muffler scheme discussed later.

Tag Register

The 16-bit tag register drives the tag bus on the daisy-chain cable; all disk drive commands are initiated through the tag register. The tag register is sometimes loaded from B on an output, sometimes from the format RAM. Loading a head select, cylinder select, or other command into the tag register activates a timing circuit that handles all timing requirements of the Trident drive as follows: Only the tag bus bits are enabled for the first 200 ns; then the Tag[0:3] bits are also enabled for 1.2 μs; finally, the Tag[0:3] bits are disabled again and the SeekTag or ControlTag flip-flop is set to wakeup the disk task (indicating completion of the Tag instruction). The drive select tag (Tag[0]) does not activate the timing circuit, since the timer counts disk clock cycles, but disk clocks are invalid during drive select changes.

Bits 4 through 15 of the tag register are interpreted according to the following table:

Tag[0] Drive select and subsector count
Tag[4:15] are interpreted by the controller to effect drive select or subsector counter changes. The tag timing and wakeup circuit is not activated; firmware must take care of the timing by first loading Tag[4:15] as desired but with Tag[0:3] equal 0, then or-ing in the Tag[00] bit and outputting again.

4:9 Subsector count
Divide the 117 subsector pulses from disk by subsector count + 1 to form Sector pulses (Tag[4:5] are presently unused).
Tag[4:9] = 3 yields 29 sectors large enough for 256-word data blocks
Tag[4:9] = 6 yields 16 sectors large enough for 512-word data blocks
Tag[4:9] = 14_8 yields 9 sectors large enough for 1024-word data blocks

10 Load subsector from Tag[4:9] for the drive selected prior to the execution of this tag instruction.

11:15 Drive select
The basic controller handles up to 4 disk drives; additional units may be accommodated by adding drive dependent logic on an additional board and connecting it in in place of drive 3. To allow this, the 5 bit drive select field is interpreted as follows.

0 - 3	select drive 0 to 3, respectively
4 - 36_8	select drive 3
37_8	don't select any drive

Tag[1] Head Select

A SeekTag wakeup occurs at completion (1.6 μs).

4:7 Unused

8 Off Cylinder--may be activated during a read to attempt recovery of unreadable data. It causes cylinder positioning to be offset 80 micro-inches.

9 Determines direction of offset if bit 8 is set.

10:15 Head Select--values from 0 to 4 are valid for a T-80, 0 to 19 for a T-300. The drive will turn on "EndOfCylinder" (alias HeadOverflow) error if an invalid head address is issued.

Tag[2] Cylinder Select Tag

A SeekTag wakeup occurs at command completion (1.6 μs) **and** at seek completion.

4:5 Unused

6:15 CylinderAddress--the drive will be non-ready for from 6 ms (adjacent cylinder) to 55 ms (cylinder 0 to 814). Typical seek times appear to be around 3 ms for adjacent cylinders and 50 ms for full range.

Tag[3] Control Tag

A ControlTag wakeup occurs at command completion (1.6 μs) **and** at completion of a transfer operation.

4 AltoLeader--special flag to the controller that allows disks written by an Alto Tricon Controller to be read. This bit should only be used for the Alto Trident simulation.

5 Unused

6 Strobe Late--causes data recovery circuits within the drive to sample data early within the data bit time (for recovery when the drive is experiencing excessive read errors).

7 Strobe Early--like StrobeLate except in the obvious way.

8 Write--turns on the write circuits.

9 Read--turns on the read circuits.

10 Unused

11 Reset Head register--zeroes the head address register.

12 Device Check Reset--resets all latched error conditions in the drive.

13 Head Select--turns on the head selection circuits; should only be turned on by tag commands in the format RAM.

14 ReZero--reposition the heads to cylinder 0 (if the heads are loaded) and reset the head address register; resets "Seek Incomplete" and illegal cylinder address error conditions.

15 Head Advance--advance the head address by one.

FIFO Register

Data to/from the disk is buffered through a 16-word FIFO (25 μs of buffer), which is read/written with Pd←Input/Output←B when TIOA selects the FIFO. Each FIFO word holds 16 data bits, 2 parity bits, and a 2-bit field indicating that the next word to be read is either write, read, or read&check type data. During output to the disk, the controller checks parity both when receiving data on the io bus and again when reading the FIFO. During a disk read, parity is computed before writing into the FIFO, is passed through the FIFO, and is then written on the io bus for the processor to test.

Muffler Input

Dorado uses a multiplexor scheme called the muffler system for reading miscellaneous logic signals during debugging from the Alto or baseboard. The disk controller also allows a muffler address to be specified on an output to the Muff register; in this way, any DskEth board signal available through the multiplexors (mufflers) is also available for firmware sampling. Other bits of the Muff register output specify other operations as follows:

B[0]	Simulate read data of 1 for 1 cycle (for use by diagnostic programs)
B[1]	Simulate read clock of 1 for 1 cycle (for use by diagnostic programs)
B[2]	Clear CompareErr--done by disk task if a read&compare is found to be OK
B[3]	Set ReadDataErr--done by disk task to inhibit future writes
B[4]	Clears the index wakeup flip-flop
B[5]	Clears the sector wakeup flip-flop
B[6]	Clears the seek/tag wakeup flip-flop
B[7]	Clears all error flip-flops within the controller (not the disk drive)
B[8:15]	Muffler address--signals are enumerated below

Following an output to the Muff register, the firmware must wait one cycle before inputting the selected muffler signal with Pd←Input. The state of the signal selected will be driven on IOB[15]. Various signals are grouped into 16-bit words (specify the word name to Midas) as in the following table. The bits within each word, and an appropriate explanation follows:

KSTATE various bits indicating the state of the controller

000	TempSense	see "Dorado Debugging Interface" document
001	IndexTW	disk task wakeup is due to an index pulse; index pulses occur once/disk revolution (16.7 ms) and are used to synchronize the firmware sector counter.
002	SectorTW	disk task wakeup is due to a sector pulse.
003	SeekTagTW	disk task wakeup is due to a completed seek or tag command (seeks in progress can be noted by reading the state of NotReady).

004	RdFifoTW	disk task wakeup is due to enough full FIFO words for input (3 for normal read or 1 for read-and-check); the wakeup is cleared by reading FIFO words until the condition is no longer met; after dropping the FIFO count below 3 in a read, the disk task must wait 4 instructions before it can block (A 2 instruction/word loop will execute 3 times and leave the FIFO empty); after dropping the FIFO count below 1 in a read-and-check, the disk task must wait 2 instructions before it can block (results in a 4 instruction/data word loop).
005	WrFifoTW	disk task wakeup is due to 4 or more free words in the FIFO on a write; the wakeup is cleared by writing FIFO words until the condition is no longer met; after raising the FIFO count above 12, the disk task must wait 4 instructions before it can block (A 3 instruction/2 data word loop will execute 2 times and leave the FIFO full).
006	ReadData	Data bit from the disk (available for diagnostics)
007	WriteData	Data bit to the disk (available for diagnostics)
010	EnableRun	Format RAM has been written, and wakeups are enabled
011	DebugMode	Controller has been placed in debug mode
012	RdOnlyBlock'	The controller is processing a sector in normal read mode
013	WriteBlock'	The controller is processing a sector in write mode
014	CheckBlock'	The controller is processing a sector in read&check mode
015	Active	The controller is processing a sector
016:017	Select.0..1	The address of the currently selected drive unit

KSTAT various bits indicating the status of the drive/controller. The controller will turn on WriteInhibit for the remainder of the sector after any of the following errors are detected, but still going through all the motions of word transfers.

020	SeekInc	The disk drive has not correctly positioned the heads within the last 700 ms. A ReZero command must be issued to clear this error.
021	HeadOvfl	The head address given to the disk drive is invalid (i.e. greater than 4).
022	DevCheck	One of the following errors occurred: Head select, Cylinder select, or Write command and disk not ready Illegal cylinder address. Offset active and cylinder select command. Read-Only and Write. Certain errors during writing, such as more than one head selected, no transitions of encoded data or heads more than 80 micro-inches off cylinder. A ReZero command must be issued to clear this error.
023	NotSelected	The selected drive is in "off-line" test mode or the selected drive is not powered up
024	NotOnLine	The drive is in test mode or the heads are not loaded
025	NotReady	There is a cylinder seek in progress or the heads are not loaded
026	SectorOvfl	The controller detected that a command was active when the next sector pulse occurred. This error implies either a hardware malfunction or a discrepancy between the sector format of the drive and the word count the program thinks is appropriate.
027	FifoUnderflow	Either the FIFO became empty while writing (task got behind) or the FIFO had too many words taken out of it while readng (microcode word count or wakeup error).
030	FifoOverflow	Either the FIFO became full while reading (task got behind) or the FIFO had too many words put into it during writing (microcode word count or wakeup error).
031	ReadDataErr	A flip-flop in the controller for latching one of three errors:

	CompareErr'	A CheckBlock was processed where the microcode failed to clear the CheckError flag.
	ECCError	the microcode can set the ReadDataErr flag if it determines that the ECC words after reading one block are non-zero in order to inhibit future writes.
	ECCComputeErr	The ECC hardware within the disk controller failed to generate a single "1" bit (i.e. a hardware malfunction).
032	ReadOnly	The "Read-Only" switch on the drive is on.
033	CylinderOffset	The cylinder position is currently offset. This is a mode used for recovery of bad data.
034	IOBParityErr	The controller detected bad parity on the IOB bus.
035	FifoParityErr	The controller detected bad parity on the data out of the FIFO.
036	WriteErr	OR of errors on muffler addresses 020-035
037	ReadErr	OR of errors on muffler addresses 020-031 and 034-035

KRAM contents of the format RAM

040:043	Address of format RAM word
044:057	contents of format RAM word

KTAG contents of the tag register

060:077	20 bit value last loaded into the tag register

KFIFO state of the io control logic

100	ShiftIn	The controller is currently shifting data into the FIFO
101	ShiftOut	The controller is currently shifting data out of the FIFO
102	ComputeECC	The controller is currently shifting data and computing the ECC checksum
103	NextBlock	Occurs between blocks within a sector
104	LoadTag	Indicates that the next word read from the format RAM should be loaded into the tag register as opposed to the count register
105	CntDone'	Indicates that the count register is again zero, and a new value from the format RAM will be loaded next
106	OutRegFull	The holding register on the input to the FIFO has been loaded, but not transferred into the FIFO.
107	InRegFull	The holding register out of the FIFO has been loaded, but not read via Pd←Input or loaded into the output shift register.
110:113	FifoWaddr	The 4-bit address indicating where the next word will be written into the FIFO
114:117	FifoRaddr	The 4-bit address indicating where the next word will be read from the FIFO. if FifoWaddr equals FifoRaddr then the FIFO is defined as empty.

Error Detection and Correction

To allow high data density and a few surface imperfections during manufacture, Trident disk packs are not required to be perfect. A disk pack is defined as suitable when no more than three bad areas occur on any data surface; a bad area is defined as one which could potentially cause read errors of no more than 11 bits in length. To correct errors arising from these imperfections as well as other (infrequent) read errors, the controller implements an error detection and correction scheme which will detect (with very high probability) errors of any length, and will allow correction of any burst error of 11 bits or less.

Warning: If an error burst longer than 11 bits occurs, there is a significant possibility that the error correction algorithm detailed below will fail and double the number of bad bits! Consequently, disk handling programs should try other methods of error recovery before invoking the error-correction algorithm.

To avoid problems, it is good practice to run diagnostic programs on new disk packs; note bad sectors and don't use these during normal operation.

When an error does occur, the first step is to try rereading the offending sector several times. One of these reads may succeed. If not, try rereading with the cylinder position offset or with the data strobe early or late as discussed in the "Tag Register" section. If these attempts all fail, then try error correction.

Error correction is accomplished through a mixture of disk controller hardware (for ECC generation and checking) and system software/firmware (for error recovery). This is a compromise between capability, speed, and cost. The basic capabilities and restrictions of the 32-check-bit scheme are summarized below.

1) A single error burst of length less than 12 data bits (i.e., a scattering of error bits within the bit stream, all of which fit within an 11-bit span) can be corrected in blocks shorter than 2685 data words. (Example: for the data "0001100101", the data "0000101101" contains a single burst error of length 4.). The code implemented will detect errors in arbitrarily long blocks, but not enough information exists to correct longer blocks.

2) Simple error detection--two words are returned by the hardware which are both zero if the read is successful.

3) Software/firmware error correction can be completed in less than one disk revolution. The correction procedure is well suited to a mixture of software and firmware. If done entirely in firmware, error correction would take less than 1 ms.

4) Not all uncorrectable errors will be detected as such. An uncorrectable error requires two bad spots on the disk surface within one sector (the pack is bad - throw it out!), an electronic error in a sector with a bad spot, or two electronic errors within one sector. If such an error has occurred, it can, with a probability of say 20 percent, result in an error pattern and displacement that seems valid. This will result in leaving the error bits uncorrected and changing some bits which were in fact correct. This means that for high data security, a check code should be generated and imbedded as part of the data file before writing on the disk.

The error-correcting code (ECC) generated is referred to as a Fire Code (see Error-Correcting Codes by Peterson). The following is a detailed description of this code and recovery procedure.

The code calls for dividing the outgoing data stream by a polynomial of the form:

$$P(X) = P_1(X)(X^m + 1)$$

Where $P_1(X)$ is an irreducible polynomial of degree n (n = burst length) and m is > 2*n. For this particular application the polynomials chosen are:

$$P(X) = (X^{11} + X^2 + 1)(X^{21} + 1)$$

During a write, the two polynomials are multiplied together and implemented by hardware in the form:

$$P(X) = X^{32} + X^{23} + X^{21} + X^{11} + X^2 + 1$$

The data stream is premultiplied by X^{32} to make room for the 2 word ECC and then reduced modulo P(X). This is accomplished by the normal feedback shift register technique with the difference that to perform premultiplication, the output of the register is exclusive-or'd with the incoming data and then fed back. After all data bits have been shifted out, the contents of the ECC shift registers are appended to the disk block.

During a read, the feedback shift register is reconfigured such that the two original polynomials are implemented separately. The incoming data stream, including the 2 appended words of ECC, is independently reduced modulo $P_0(X)$ and $P_1(X)$, where

$$P_0(X) = X^{21} + 1$$

$$P_1(X) = X^{11} + X^2 + 1$$

After reading in all words off the disk, the contents of the two polynomial shift registers are read out of the FIFO. If the data is recovered without error, then reducing it modulo $P_0(X)$ and $P_1(X)$ results in the registers containing all zeroes.

If the data contains an error, then the two registers will be non-zero. If one but not both registers is non-zero, then the error is irrecoverable.

To recover from an error, a procedure is undertaken which determines the pattern of bits which are in error, and the displacement of this pattern from the end of the record. I am simply going to present the magic equation to be solved, and some magic constants to be used for solving this equation. Much of the polynomial implementation and the equations, which use the "Chinese Remainder Theorem" are discussed in technical reports from CALCOMP (Calcomp Technical Report TR-1035-04, by Wesley Gee and David George) and XEROX (Xerox XDS preliminary report "Error Correction Code for the R.M. Subsystem," by Greg Tsilikas, 28 March 1972.).

The basic equation is:

$$D = Q*LCM - (A_0*M_0*S_0 + A_1*M_1*S_1)$$

where:

E_i = modulus of the polynomial
LCM = least common multiple of E_0 and E_1
M_i = LCM/E_i
A_i = a constant such that A_i*M_i modulo E_i = 1
Q = smallest integer to make D positive

S_i = number of shift operations to the appropriate polynomial remainders as described below.
D = displacement of right-most incorrect bit from the end of the record.

The values of E_0 and E_1 were found by programming the procedure outlined in the CALCOMP report, and yielded the following result:

$E_0 = 21$ $E_1 = 2047$

The least common multiple (LCM) of E_0 and E_1 is simply the product of E_0 and E_1 since the two numbers have no factors in common. Thus the LCM, which is also the record length which can be corrected, is 42,987 bits, or 2686-2 words.

Knowing LCM and E_0 and E_1, the values of M_0 and M_1 are easily found to be

$M_0 = 2047$ $M_1 = 21$

The values of A_0 and A_1 are next determined using a trial and error approach that I put in a small program. The results can easily be confirmed, and are given below:

$A_0 = 19$ $A_1 = 195$

All of the above values derived so far are constants determined for the particular polynomials chosen. The values of S_0 and S_1 are determined in the software from the error patterns returned at the end of a disk transfer.

S_0 is first determined by a software procedure using the following steps:

1) The remainder from dividing the input data by $X^{21} + 1$ is found in ECC[11:31]; if this remainder is zero, then the error is uncorrectable.

2) Test the low order 10 bits for all zeroes, and if not then perform a left circular shift on the 21 bits. When the low order 10 bits are all zeroes, the error pattern is in the upper 11 bits of the word, and S_0 is the number of times the circular shift was performed.

3) If the low order 10 bits don't become all zeroes within 20 shifts (1 full cycle), the error is uncorrectable.

S_1 is then determined in microcode as follows:

1) The remainder from dividing the input data by $X^{11} + X^2 + 1$ is found in ECC[0:10]; if this remainder is zero, then the error is uncorrectable.

2) Test this number to see if it is equal to the error pattern determined in step 3 of S_0, and if not reduce this number modulo $X^{11} + X^2 + 1$ (left shift and XOR feedback). When the contents of this word equals the error pattern (it is guaranteed to happen before 2047 reductions), S_1 is determined as the number of reductions performed (In the hardware implementation of switching from the write polynomial to the read polynomials, it was easier to implement a polynomial that

premultiplied by X^{11}. This means that the remainder returned by the hardware already has had 11 shifts performed. To compensate, when S_1 has been determined by the above procedure, you must add 11 to the value, and subtract 2047 if the result is greater than or equal to 2047.).

The basic equation for the displacement now looks like

$$D = Q*42{,}987 - 19*2047*S_0 - 195*21*S_1$$

where:

$$0 \leq S_0 \leq 20$$
$$0 \leq S_1 \leq 2046$$

Notice that the straightforward solution to this equation cannot be done with single-precision arithmetic on the Dorado; to avoid double precision, the following manipulation of the equations is useful:

$$D = Q*2047*21 - 19*2047*S_0 - 4095*S_1$$
$$D = Q*2047*21 - 19*2047*S_0 - 2*2047*S_1 - S_1$$
$$D' = Q*21 - 19*S_0 - 2*S_1$$

where:

$$0 \leq D' \leq 20$$
$$D = 2047*D' - S_1 \qquad \text{(add 42,987 if } D' = 0\text{)}$$

For some reason that we don't understand, the actual required calculation must be $D = 2047*(D'+1) - S_1$ *in the last step. Also* D' *is conveniently calculated as* $(215*21 - 19*S_0 - 2*S_1)$ *rem 21.*

Display Controller

The Dorado Display Controller (DDC) uses the fast io system to obtain representations of video images from storage; it then transforms these representations into control signals for monitors. Its three design objectives are:

(1) To handle a variety of color, grey-level, and binary (black-and-white) monitors;
(2) To utilize the full power of the fast io system in producing high-bandwidth computer graphics;
(3) To allow various compromises in color and spatio-temporal resolution for experimental purposes. Clock rates, video signals, and other monitor waveforms should be controllable by firmware.

There are *two* independent video channels capable of running in a variety of modes. Two channels allows text to be displayed on one channel, graphics on another, or the main picture on one, cursor on the other.

The DDC must readily handle OIS monitors which we expect to be standard for most systems. Bit maps, display control blocks, and monitor control blocks, similar to the Alto display system, are expected to provide the client interface to the DDC. The OIS seven-wire video interface makes provision for one or more low bandwidth input devices (keyboard, pointing device, etc.); our current provisions for keyboard and cursor position input are also discussed in this chapter.

Keep Figure 14 in view while reading this chapter.

Operational Overview

Video scan lines are encoded in *bitmaps*, which are contiguous blocks of virtual memory; the two channels, A and B, have independent bitmaps and data paths in the DDC. The high-priority *DWT* (Display Word Task) runs on behalf of either A or B using the subtask mechanism; it transmits each bitmap to a FIFO consisting of 15 munches/channel. The bitmap stream emerging from the FIFO is then sorted into *items* (1, 2, 4, or 8 bits wide) for each channel which are combined, mapped, and transformed into *pixels* (picture cells) on the screen.

In addition to the two channels, the DDC supports a programmable cursor that is 16 pixels x 1 bit/pixel wide.

A lower priority *DHT* (Display Horizontal Task) handles horizontal and vertical retrace and sets up starting addresses and munch counts, cursor data, and formatting information in the *NLCB* (Next Line Control Block) for the DDC. The NLCB is then copied into the *CLCB* (Current Line Control Block) during horizontal retrace prior to the next scan line.

The rate-of-flow of items is governed by the *resolution* and *pixel clock period*. Resolution may be independently programmed for each channel so that items flow at 1/4, 1/2, or 1 times the pixel clock period. If the DispM board is present, then the pixel clock period is also progammable; otherwise, it is determined by a crystal oscillator on the DispY board, which must have a frequency appropriate for the monitor being driven.

The DDC is implemented on two logic boards. DispY contains the crystal oscillator, the MiniMixer discussed below, and all control logic for non-color monitors. Color monitors, such as those using an RS343 standard video signal system, also require the DispM board, which contains the Mixer, a programmable pixel clock, and a collection of data formatters. A backpanel jumper determines whether the crystal oscillator or programmable clock is used.

Items can be treated in one of three ways: First, an Alto monitor can be driven. Secondly, items can be mapped through the 256-word x 4-bit *MiniMixer* into video data for an OIS or grey-level monitor.

Three separate interfaces are provided on the DispY board: An Alto monitor interface or's one-bit items from the A and B channels with the cursor, and then xor's by polarity to produce one-bit pixels for an Alto display; a seven-wire interface outputs 1 bit/pixel for a binary OIS monitor; and an 8-bit digital-to-analog converter (DAC) produces grey-level video.

Thirdly, items may be mapped by the *Mixer* (or color map), a 1024-word x 24-bit RAM, into controls for a color or grey-level monitor; a variety of modes determine which bits from the A and B items address the mixer or bypass the mixer. Mixer output consisting of 8-bits for each of the red, green, and blue guns can then be digital-to-analog converted for color monitors.

Video Data Path

Fast IO Interface and FIFO

The fast io system delivers data to the DDC at a rate of 16 bits/clock; words are received alternately in the REven (t_1) and ROdd (t_2) registers shown in Figure 14, then written into the FIFO, a 256-word x 32-bit RAM, during the first half of the next Dorado cycle (t_2 to t_3), leaving the second half of the cycle free for read access by the video channels. In other words, the REven and ROdd registers widen the data path from 16 to 32 bits to allow sufficient time to both write and read the FIFO in one cycle.

The 256 double-words in the FIFO are divided evenly among the two channels, so each has buffer storage for 16 munches. Each channel has write and read pointers that address the FIFO when appropriate.

Write pointers are initialized once during vertical retrace and then sequence through addresses for the entire display field; a write pointer is incremented after each double-word write for its channel, so that the next word to be written is addressed at all times. Since the fast io system delivers only one munch at-a-time, there is never any problem in deciding which of the two write pointers should address the FIFO.

Read pointers, however, are initialized during each horizontal retrace, so that the correct first double-word is read at the start of every scan line. This is required because the fast io system always delivers complete munches, but unused double words may appear at the end of the last munch for the previous scan line, or at the beginning of the first munch for the current scan line; the read pointer has to be reinitialized to skip over these. FIFO reads alternate between channels A and B, so the data rate for one channel is limited to 32 bits/2 cycles (= 16 bits/cycle).

Note that *bitmaps are required to start at even addresses* because the FIFO is 32 bits wide.

Item Formation

At the output end of the FIFO there is a multiplexor shared by both channels and, for each channel, two intermediate buffers (*FIB* and *SIB*), and a shift register *SR*. The multiplexor permutes the 32-bit quantity emerging from the FIFO so that when the double-word has marched through FIB and SIB and is finally loaded into SR, successive shifts will produce successive items of the selected size (8, 4, 2, or 1 bits).

The SR is tapped as follows:

SR.0	Item[0] for item sizes 1, 2, 4, or 8;
SR.16	Item[1] for sizes 2, 4, or 8, gated to 0 for size 1;
SR.8, SR.24	Item[2:3] for sizes 4 or 8, gated to 0 for sizes 1 or 2;
SR.4, SR.12, SR.20, SR.28	Item[4:7] for size 8, gated to 0 for sizes 1, 2, or 4.

All eight Item bits are gated to 0 if the channel is off. It is useful to think at this point that, regardless of a channel's item size, an 8-bit wide item is produced, whose bits contain non-zero data only in those positions dictated by the item size; i.e., for size 1 only the most significant bit may be non-zero; size 2 allows data in the topmost two bits, etc.

The SR loads on the *item clock* after its last item has been used; the item clock rate is the pixel clock rate divided by the resolution (1, 2, or 4 for full, half, or quarter, respectively). Hence, for 8, 4, 2, or 1-bit items, SR will be shifted 3, 7, 15, or 31 times, repectively, and be reloaded from SIB on the following item clock.

Synchronization of SR, which uses the item clock, with FIB and SIB, which use the Dorado system clock, is a little tricky. SIB←FIB will occur no later than (4.6 ns) + C + (1.1 ns) + C + C = 3*C + 5.7 ns after SR←SIB, where C is the period of the Dorado system clock and 4.6 ns and 1.1 ns are the worst case propagation delay and setup time of the components in the synchronizer; FIB←FIFO will occur at this time or on one of the next three Dorado clocks, depending upon which of these four clocks corresponds to t_2 of the cycle in which this channel can read the FIFO. Allowing for propagation delay through SIB (5.0 ns) and setup time for SR (1.7 ns), the worst case minimum spacing between loads of SR is 3*C + (5.7 ns) + (6.7 ns) = 3*C + 12.4 ns. This must be less than the time for emptying SR which is I*(32/ItemSize), where I is the period of the item clock. Hence, I > (3*C + 12.4)/4 for ItemSize = 8, or I > 21 ns for a Dorado clock period of C = 25 ns.

The 8-bit items from the two channels are then presented to either the Mixer section on the DispM board or the MiniMixer or Alto video interface on the DispY board.

Mixer

The Mixer is controlled by the *A8B2*, *24Bit*, *ABypass*, and *BBypass* mode signals; it is a 1024-word x 24-bit RAM for which the 10 bits of address required may be obtained from two possible source distributions, depending upon the A8B2 mode. When A8B2 is true, the address consists of AItem[0:7] and BItem[0:1]; when false (called A6B4), the address is AItem[0:5] and BItem[0:3].

αBypass mode (α = A or B) prevents αItem from contributing to the Mixer address; instead, αItem[0:7] bypass the mixer and are or'ed with eight specific Mixer output bits. For example, with ASize = 8, BSize = 4, BBypass, and A8B2 true, and with appropriate

values in the Mixer RAM, the controller may be thought of as three 4/bits pixel channels driving three color guns. One channel is bypassed data from B, while the other two are mapped through the Mixer. Another example: AItemSize = 8, BItemSize = 8, both A and B bypassed, yields two 8 bit/pixel color channels.

24Bit mode also bypasses both channels around the mixer. It is intended to run a three-channel color display directly from memory. In this case, ASize = 8, BSize = 8, and the A channel will directly drive two color guns by sending out alternate 8-bit items (properly aligned, of course) to the two guns. Meanwhile, the B channel will run at half the A channel rate, be bypassed, and drive the third gun.

After permutation as dictated by the mixer modes, chosen bits are loaded into the mixer address and/or bypass registers, causing the mixer to produce a new video value every pixel clock (every two pixel clocks in 24Bit mode). Mixer output is loaded into the mixer output register every pixel clock; the output register drives the hardware that produces actual video waveforms. A triplet of very fast DAC (digital-to-analog converter) modules are available to produce a Red-Green-Blue triple of analog signals for a color monitor, or up to three grey-level video signals. For simple binary monitors, one or more mixer outputs might be simply level-converted to produce TTL compatible outputs. In conjunction with monitor control circuitry, which produces sync, blank, and composite waveforms, a wide variety of monitors can be attached to the Dorado.

Alto Video Interface

A small circuit on the DispY board produces video for an Alto monitor. This circuit or's CursorData, AItem[0], and BItem[0], then xor's by the polarity, and finally or's with the vertical and horizontal blanking signals.

MiniMixer

A small video mixer on the DispY board, not to be confused with the large Mixer on the DispM board, can drive either a DAC or the OIS seven-wire interface discussed later. The MiniMixer is a 256 word x 4-bit RAM written with the MiniMixer output command; when not being written, it is addressed by a combination of AItem, BItem, and state bits, as shown in Figure 14. On every pixel clock, dDAC[0:3] are loaded from MiniMixer output, while dDAC[4:7] are loaded directly from AItem[4:7]. The MiniMixer aims at experiments with grey level monitors and initial experiments with mixing channels.

Horizontal and Vertical Control

The DDC has a set of registers called the *CLCB* (Current Line Control Block) which controls video generation for the current scan line. The DHT sets up parameters for the next scan line in *NLCB* (Next Line Control Block), a 16-word x 12-bit RAM. The first 32 pixel clocks of horizontal retrace are called the *HWindow*; during HWindow parameters for the next line are copied from NLCB into CLCB. Vertical control is also handled through the NLCB.

Every monitor requires horizontal synchronizing and blanking waveforms. Interlaced monitors must be able to distinguish fractions of a scan line to implement interlacing. In general, the duration and phasing of sync/blank waveforms is unique to a given monitor. The DDC uses the 1024-word x 3-bit *HRam* (Horizontal RAM) to control horizontal sync/blank.

The interpretation of fields in NLCB and HRam are shown in Figure 15 and loading will be discussed in the "Slow IO Interface" section; the use of the different information is discussed here. The top part of Figure 14 shows how horizontal timing is controlled.

Line Control Blocks

The fields in NLCB/CLCB are interpreted as follows, where α denotes that the item is channel-specific (i.e., copies exist for both A and B channels):

αPolarity. A single bit, used only for Alto emulation, that inverts black and white (APolarity and BPolarity are or'ed by the hardware).

αResolution. A 2-bit field that controls item clock generation; values of 0, 2, and 3 cause quarter, half, and full resolution, respectively.

αItemSize. A 4-bit field unary encoded as αSize1, αSize2, αSize4, or αSize8, denoting bits/pixel for the channel; setting multiple bits is illegal.

αLeftMargin. A 12-bit field in units of pixel clocks specifying 33 less than the number of pixel clocks to wait after HWindow completes before turning the channel on. This value is not a straightforward constant, but depends upon monitor-specific horizontal retrace time; if the horizontal retrace time is R pixel clocks and the desired visible left margin is L pixel clocks after the end of horizontal retrace, then αLeftMargin should be loaded with R + L-32-33 = R + L-65, independent of resolution. Since L may be 0, this implies that the horizontal retrace time for the monitor must be greater than 65 pixel clocks. Since high-speed monitors typically have greater than 4 μsec horizontal retrace times, and are this fast only with high speed pixel clocks, this restriction is not expected to be significant.

Note: For an OIS interface, αLeftMargin must be R + L-70, rather than R + L-65, because video signals are delayed from horizontal control waveforms by 5 pixel clocks.

The CLCB entries for αLeftMargin are counters, and the counter states are decoded to initiate channel activity--FIFO reads are initated just before the channel turns on; since L = 0 is legal, the counter must start during horizontal blanking, not at the end of blanking.

αWidth. A 12-bit counter that counts at the pixel clock rate as soon as the channel turns on; when the counter overflows (or when horizontal retrace starts, whichever is earliest), the channel is turned off. Its value must also be adjusted for resolution.

αFifoAddr. An 8-bit quantity pointing to the munch and word within the munch for the first FIFO read for the next scanline; this must be an even number because doublewords are fetched from the FIFO. Firmware must keep track of the number

of used munches for any given line and advance αFifoAddr by exactly the right amount, adjusting for munch boundaries, interlacing, and data breakage. The CLCB register for αFifoAddr is the channel read pointer itself.

MixerModes. A set of bits that control the mixer; these are NOT channel-specific. These will normally be changed infrequently, maybe at the field rate or during display initialization. However, they are in the NLCB to allow modes to change on the fly.

Vertical Control Word (VCW). A word controlling the vertical retrace operation of the monitor; it contains the vertical blank bit, vertical sync bit, and interlace field bit discussed in the "Vertical Waveform Generator" section below.

Cursor and CursorX. The 12-bit CursorX value is loaded into a counter which starts counting at the end of HWindow. When the counter overflows, the 16-bit Cursor value is shifted out onto the CursorVideo line. This is used by the Alto video interface and in the MiniMixer address.

Horizontal Waveform Generator

The 1024-word x 3-bit HRam contains control information for these waveforms. Under normal operation, HRam is addressed by a 12-bit counter (HRamAddr[0:11]) which is reset at the leading edge of horizontal sync and then increments every pixel clock until the next leading edge of horizontal sync; HRamAddr[1:10] address the RAM, and the output is loaded into the HRamOut register every other pixel clock. The three bits in HRamOut are intended to represent horizontal synch, horizontal blank, and half-line; these three bits are combined and level shifted by a logic network appropriate for the monitor being driven.

The 1024-word HRam imposes the uninteresting restriction that there be less than 2048 pixels/scan line.

As shown in the diagram at the top of Figure 14, horizontal blanking (HBlank) is true from the end of one scan line to the beginning of the next. During horizontal blanking, HSync is turned off to initiate the horizontal retrace and turned on again when horizontal retrace is finished. HBlank then continues for a monitor-specific interval. Note that if a channel's visible left margin is non-zero, then the horizontal scan will begin before that channel is producing any data; in this case, the video channel outputs zero items to the mixing stages until the channel is turned on.

Vertical Waveform Generator

Only 2:1 interlaced monitors are supported in this design, but more complicated vertical control could be provided, if desired. To support 2:1 interlace, HRam contains a waveform called HalfLine, which is a pulse at the horizontal line frequency, 180^{o} out of phase with HSync.

Vertical control is handled by DHT through the NVCW word in the NLCB, which specifies whether or not vertical blank or retrace should begin or end during the next scan line. DHT must keep track of scan lines to enable vertical signals at the appropriate times. The CVCW register in CLCB is the first register loaded from NLCB near the beginning of

HWindow; since VSync must be stable before the rising edge of HSync, this allows the rising edge of HSync to occur as early as 2 pixel clocks after the beginning of HBlank.

The three VCW bits are called *VBlank*, *VSync*, and *OddField*. VSync enables vertical sync to begin on the next line, and the OddField bit chooses either HSync or HalfLine on which to do vertical syncing (OddField = 1 implies HalfLine phasing for vertical sync). This phase will alternate from the start of the line to the middle of the line and back for successive fields. The blanking signal for the monitor is VBlank or'ed with HBlank.

Pixel Clock System

The programmable pixel clock on the DispM board determines the fundamental video data rate for a given monitor. The pixel clock frequency is determined by multiplying a 200 kHz **(?to be determined, but this is about optimal?)** reference signal by PClock[0:7] + 1, where the 8-bit PClock register is loaded by a slow io operation; implementation is analogous to that used for the Dorado system clock. Hence, the pixel clock period becomes 5000/(PClkRate + 1) ns, as shown in the table below. Note that the pixel clock will not stabilize until about 1/2 second after the PClock register is loaded.

Table 23: PClock vs. Pixel Clock Period
(table will be filled in after completion of the design)

PClock (Octal)	Period (ns)	PClock (Octal)	Period (ns)	PClock (Octal)	Period (ns)
377	20	247	xx	117	xx
367	xx	237	xx	107	xx
357	xx	227	xx	77	xx
347	xx	217	xx	67	xx
337	xx	207	xx	57	xx
327	xx	177	xx	47	xx
317	xx	167	xx	37	xx
307	xx	157	xx	27	xx
277	xx	147	xx	17	xx
267	xx	137	xx	7	833
257	xx	127	xx	1	3333

The parts of the DDC synchronized to the rest of Dorado do, of course, use the Dorado system clock. As discussed earlier, the synchronization logic for refilling SIB after SR←SIB puts a lower bound on the pixel clock period of (3*C + 11)/4 ns (= 21.5 ns for a Dorado clock period of C = 25 ns), for an item size of 8 on either channel. We anticipate that pixel clock rates in the range 10 to 50 MHz (100 to 20 ns/pixel) will be required, so the lower bound is approximately consistent with this.

OIS Seven-Wire Video Interface

So that a number of different controller and terminal types may be freely interconnected in D0 and Dorado-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 DDC transmits digital video and sync to the terminal over six pairs of a seven-pair cable. The input data is encoded by a microcomputer in the terminal and sent back serially over the seventh pair (the "back channel"). Video and control (sync) are time-multiplexed, and four bits are transmitted in parallel to reduce the cable bandwidth required.

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

Video Output

The four output lines are interpreted as either a 4-bit nibble of video or four control signals according to the phases of the two clock signals; the DDC places data on the data lines at the falling edge of ClkA, and the terminal samples this data on the rising edge of ClkA. If ClkB is 1 at this time, the nibble is interpreted as four bits of video, else 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 a nibble is interpreted as control information, bit 2 is reserved for horizontal sync and bit 3 for vertical sync, while 0:1 are undefined; different types of terminals may use 0:1 for any purpose.

A circuit on the DispY board drives the seven-wire interface from the MiniMixer. MinMixer[0] is serial-to-parallel converted into four-bit nibbles, which are held in a register for transmission. Sync, blank, and clock phases are generated in accordance with the OIS Interface Specification.

Back Channel

Data from low bandwidth input devices at the terminal are transmitted serially over the back channel. Data are clocked by the terminal on the rising edge of the horizontal blank pulse and are sampled by DHT during the subsequent scan line after HWindow.

By convention the terminal microcomputer encodes 24-bit messages (delivered in 24 scan lines); each message begins with a 1, and after the 24th bit of the message the DHT ignores the backchannel until the start of another message is indicated by another 1. The first 8 bits of a message indicate a message type, the last 16 bits are the body of the message; since the first message bit is always 1, this allows 128 message types, each with a 16-bit body.

The terminal microcomputer perpetually cycles through all 64 keys on the keyboard, dectecting changes in state of the keys; the state of the keyboard then exists in four 16-bit words, and a back channel message is defined for each. Whenever one of these words

changes value, it is sent to the Dorado in a message; changes in mouse x,y coordinates are reported once per field (i.e., twice/frame or typically 60 times/sec); if the mouse has not changed position, then one keyboard word is reported instead of the mouse position change.

Alto-compatible firmware would sample and report these four keyboard words in memory locations 177034 to 177037 at a rate of 60 Hz, and synchronization of key depressions, debouncing, etc. would be left to software. Cursor X would be reported in memory location 424 and cursor Y in 425.

Table 24: OIS Terminal Microcomputer Messages
(*may be bugs here*)

8-bit Message Number	*Comments*
200	First keyboard word
210	Second keyboard word
220	Third keyboard word
230	Fourth keyboard word
240	Mouse buttons (??)
250	8-bit changes in X-coordinate (0:7 of the message body) and Y-coordinate (8:15 of the message body)

The terminal microcomputer perpetually cycles through all 64 keys on the keyboard, reporting the state of each in turn; about one key is reported every 50 or 60 μs, so the full cycle takes place about every 3 to 4 ms. The state of the 64 keys is then transmitted over the back channel in four messages.

Processor Task Management

This section outlines the implementation requirements of DHT and DWT and discusses the hardware associated with task wakeups and DWT subtask arbitration between the two channels.

Since DHT must do a lot of processing, it runs at low priority and is awakened once/scan line at the end of HWindow. When it runs, it must calculate all parameters for the next scan line, load the NLCB appropriately for each channel, and set up the munch address and count for each channel in the RM registers DWTnextaddr and DWTnextcount referred to in the DWT sample code below; then it sets the αNextWCBFlag flags discussed below. The DHT wakeup will remain active until any NLCB output command is executed, so the DHT must execute at least one NLCB output command every time it wakes up, and this must occur at least three instructions prior to blocking.

DWT is a very high priority task which may run on behalf of either channel: channel A is subtask 0; channel B, subtask 2. Since it uses the subtask mechanism, DWT must always block at the same instruction each iteration. DWT does not explicitly know the channel for which it is executing at any given time; its two parameters, a start address and munch count, are received from DHT in RM registers specific to the subtask. In the normal case, DWT initates an IOFetch and blocks, where the following code is prototypical:

```
%RM registers for channel A, indicated by names beginning with "A" below, are used in the program, but
the corresponding set of registers for channel B, in a different RM region, will be referenced when
SubTask is 2.

Note that TIOA selects the DWT Output←B operation and T contains 20 at the beginning of the loop, so
the second instruction is used both to increment the munch address and to signal the hardware that an
IOFetch is commencing.
%
DWTStart:       ACount←(ACount)-T, Branch[DWTCheck, R<0];
                AAddress←(IOFetch←AAddress)+(Output←T), Block, Branch[DWTStart];

DWTCheck:       Branch[DWTRefill, R odd], AAddress←T-T; *branch if new line
*Finish old line--output ClrFlag, use AAddress as flag shadow.
*Next wakeup = (Not CurWCBFlag) & NextWCBFlag
                AAddress←(AAddress)+1, Output←AAddress, Block, Branch[DWTStart];

%Note that the change-RSTK-for write function used below is ok, but the change-RBase-for-write functions
are illegal because of subtasking.
%
DWTRefill:      ACount←ANextCount;  *from DHT, # munches to fetch -1 in 0:11
                BrLo←ANextAddrLo;
                BrHi←ANextAddrHi;
                Output←ASetCWCBFlagAndClearNWCBFlag, Block, Branch[DWTStart];
```

DWT lowers its wakeup request at the onset of the DWTStart instruction, and the DDC remembers that DWT is in progress. No further DWT wakeups will be generated while the task is running or is preempted by a higher priority task. Whenever DWT blocks, a counter is loaded with a value -N; when the counter overflows, DWT wakeups are allowed again. This counter has two purposes. First, within a munch loop it spaces out IOFetch references to the memory system by 8 or more cycles (depending upon N, which is adjustable through a hardware SIP component), so as not to clog the memory pipeline. Secondly, the decision to generate subsequent DWT wakeups is based upon the state of flags that may be altered by output commands; these commands take time to get from the processor to the DDC and alter the state. Other tasks may have the processor while these state changes take effect.

After N cycles have elapsed, DWT will be woken whenever αWantsDWT is true for one of the channels. Two channel-specific flags are involved in DWT wakeup control: αCurrentWCBFlag is true when α is actively moving words into the FIFO; αNextWCBFlag is set true by DHT after it has loaded the munch address and munch count into DWTnextaddr and DWTnextcount for α. After fetching the last munch for a scan line, DWT clears αCurrentWCBFlag and blocks unless αNextWCBFlag is true. In other words, αWantsDWT when

(αNextWCBFlag & not αCurrentWCBFlag) %
(αCurrentWCBFlag & αFifoAvailable).

If only AWantsDWT or only BWantsDWT, no conflict arises and the requesting channel gets DWT. However, if both channels want DWT, the channel that ran least recently will run next.

Two observations must be made about the DWT microcode. First, because the final instruction is normally an IOFetch←, the next instruction executed (by another task) will be held one cycle if it initiates any memory reference. Secondly, the two instruction loop above requires that the hardware cope with the NextLies condition discussed in the "Slow

IO" chapter; a pathological lockout problem could occur if a high demand task of higher priority is coded so that it always creates NextLies (say, by doing Block and immediate ←Md in the instruction after a fetch). This would result in the DWT wakeup being frequently delayed by 2 cycles.

Note: Neither DWT nor DHT drives the IOAtten branch condition.

Slow IO Interface

DDC manages all control functions via the slow io system. At this point you should study Figure 15, which shows the format of the various output and input commands; there are six output devices and one input device on the DispY board and two output devices on the DispM board (if present). Output commands are handled uniformly: TIOA is clocked into a register at t_1; the register output is decoded and identified as one of the DDC commands; if the processor is doing an Output←B, then at t_3 IOB data from the processor is clocked into a register and one of the "TIOA command" pulses occurs from t_3 to t_5, at which point the desired action is complete.

The IOB data received at t_3 of an Output←B will remain in the DDC buffer register (RIOB) until the next output command. This is useful for debugging and for muffler readout of the NLCB (because an NLCB address can be loaded into RIOB for multiple cycles).

The MiniMixer (if used), Mixer (if used), and HRam are RAM's that will generally be loaded during system initialization and not often changed while pictures are being displayed. The programmable pixel clock will also be loaded during initialization, if it is being used instead of the fixed crystal oscillator.

The Mixer address has two independent sources, Dorado and the video channels. Video path mixer addressing is disabled during loading from the Dorado. The Mixer output command is interpreted as follows: The KeepMixer' bit is saved in a flipflop loaded by every mixer io command; as long as KeepMixer' is true (i.e., low), video path addressing is off. If LoadMixerAddr is true, then IOB[5:14] are loaded into the MixerAddr register and IOB[15] into the hi/lo select. If WriteMixer' is true, the appropriate half of the mixer is written from the data field; in addition, the MixerAddr/HiLo register increments after writing, so the Mixer can be loaded sequentially, beginning at any address and proceeding upwards in Mixer memory, alternately loading high and low halves. A Mixer output command with both KeepMixer' and WriteMixer' false releases the Mixer from control of Dorado and returns it to the video data path.

The HRam is loaded like the Mixer; an HRam output command has KeepHRam', WriteHRam', and LoadHRamAddr bits analogous to the corresponding bits in a Mixer output command. As long as KeepHRam is true, the HRam output register is reset to zeroes; an HRam output command with all three bits false returns control of the HRam to the horizontal control logic. The HRamAddr register is incremented after writing to permit sequential loading of HRam during initialization.

The MiniMixer is loaded by a single output instruction that specifies both the address and data to be loaded. During the command pulse from t_3 to t_5 of the Output←B instruction, the video channel address to the MiniMixer is replaced by the address being loaded, so if

the video channel is active, garbage may appear at the output during this cycle.

The 16-word x 12-bit NLCB is also loaded by single output instructions that specify both the address and data. For the NLCB, output instructions are only effective when HWindow is not occurring--during HWindow the RAM address is supplied by a counter that successively copies the NLCB words into CLCB. The format of each of the words in NLCB is shown in Figure 15. Note that any NLCB output operation will dismiss the wakeup request for DHT, and DHT must not block any sooner than the fourth instruction after the first NLCB output operation is issued.

The Statics output command is used for debugging and initialization. Two bits in the Statics register called DHTShutUp and DWTShutUp are discussed in the "DDC Initialization Requirements" section below. Three other fields called *FakePClk*, *UseFakePClk*, and *MufAddr* are used for debugging. When UseFakePClk is true, the regular pixel clock is degated; if FakePClk is true, then a pixel clock will occur at t_5 (?) of the Statics output command; otherwise no clock occurs. Every Statics command also loads the hardware signal addressed by MufAddr into a flipflop (at t_5) which can be read by the DStatus input command discussed below. In combination, the fake pixel clock and muffler readout features allow diagnostic firmware to checkout most of the internal data paths in the DDC--by simulating a very slow pixel clock and "stepping" the DDC through various states, the diagnostic can check nearly all of the data paths between fake pixel clocks. The hardware signals selected by MufAddr[5:11] are given in the table below.

Table 25: DDC Muffler Signals

MufAddr	*Signal*	*MufAddr*	*Signal*
0	ACurrentWCBFlag	70	AFifoFull
01:07	AReaderPtr[1:7]	71	BFifoFull
10	ANextWCBFlag	72	ASize8
11:17	AWriterPtr[1:7]	73	ASize8-4
20	BCurrentWCBFlag	74	ASize8-4-2
21:27	BReaderPtr[1:7]	75	BSize8
30	BNextWCBFlag	76	BSize8-4
31:37	BWriterPtr[1:7]	77	BSize8-4-2
40:47	AItem[0:7]	100	AOn
50:57	BItem[0:7]	101	BOn
60:63	AServicePtr[1:4]	102:103	ARes[0:1]
64:67	BServicePtr[1:4]	104:105	BRes[0:1]

A single input device called DStatus is implemented. It is used to return the currently selected muffler bit (discussed below) and the OIS seven-wire interface received data bit.

DDC Initialization Requirements

The two low-order bits in the Statics register are called DWTShutUp and DHTShutUp. They are forced true by IOReset and prevent the respective task wakeups from happening. They are individually set or cleared by the Statics output command. In addition, IOReset sets the signal DoradoHasHRam; this will prevent horizontal sync from being sent to monitors until the HRam has been loaded and released by firmware. Blanking is sent to monitors as long as DHTShutUp remains true. It is anticipated that DHTShutUp will be left true until all DDC initialization has been completed by the emulator (or by the DHT running in response to Notify's).

Some other initialization requirements are as follows: αLeftMargin should be loaded with the maximum value (7777_8) in case one of the channels remains unused forever; the Cursor in NLCB should be zeroed in case the cursor is completely off-screen forever; HRam must be loaded with monitor-specific waveforms; the pixel clock rate must be set; mixer modes must be set. In addition, the DHT must explicitly set the αAddress registers to zero on behalf of the DWT, which cannot initialize itself completely for each subtask.

Speed and Resolution Limits

High performance color monitors are typified by the following performance limits:

22 μs	horizontal scan time
5 μs	horizontal blanking time
800 μs	vertical blanking time

Parameters for a particular monitor can be modified slightly through hardware adjustments, but cannot be controlled by the DDC, which must provide control signals with timing appropriate for the monitor. Consequently, a monitor must be chosen that conforms to the speed limitations of the DDC.

One important speed limitation is how fast bits can be moved from storage through the DDC. This limit is derived using the following parameters:

F Frame update rate. High speed phosphors require a minimum update rate of 30 frames/sec with interlaced operation for reasonable visual effects; this is marginal and faster update is desirable.

S Scan lines/frame.

VR Vertical retrace time; with interlaced operation, there will be two vertical retraces/frame.

HB Horizontal blanking time.

HS Horizontal scan time. The FIFO must not go empty during the horizontal scan or garbage will be displayed.

T Time/munch or the rate at which storage can deliver data for IOFetches; this is 1 munch/8 cycles = 1 munch/0.4 μs.

M Munches/scan line that the fast io system can deliver.

The time required to fill the FIFO for both channels is a little longer than 30*8+20 cycles (= 276 cycles) or about 13.8 μsec at a Dorado clock period of 25 ns; this follows from the fact that there are 15 munches/channel or a total of 30 munches of FIFO storage, and the fast io system can deliver one munch per 8 cycles with the first munch arriving 20 cycles after the first IOFetch←. 13.8 μsec is much smaller than the vertical blanking time and longer than the horizontal blanking time, so the FIFO will start out full at the beginning of a field and will be actively refilling itself during HS+HB of each scan iine. If the memory system keeps up with the demands of the video channels, then the FIFO will tend to refill itself after momentary transients in which it empties out a little.

Consequently, we know that HS+HB = 1/(S*F) - 2*VR, and that M = (HS+HB)/T less corrections for refresh references, storage references by other tasks, hold, and delays for tasks of higher priority than DWT. At F = 30 frames/sec, VR = 800 μs, and S = 1000 scan lines, we get HS+HB = 31.7 μs and M = 31.7/0.4 = 79 munches less corrections. There will be an average of two refresh references/scan line, so we get an upper bound of 77 munches = 19,712 bits/scan line from storage.

However, the DWT will not get all storage bandwidth. The DWT wakeup spacing is controlled by a SIP; the smallest reasonable spacing would result in one IOFetch every 8 cycles--closer spacing would result in hold while a preceding IOFetch completed, so more processor cycles would be consumed without improving data rate. At this tightest spacing, DWT runs for 2 cycles out of every 8. Conceivably, worst case memory activity discussed in the "Fast IO" chapter could occur during these 6 cycles (a clean miss 3 cycles before the IOFetch, followed by a dirty miss 2 cycles before the IOFetch, each by a different task). However, the large amount of storage in the FIFO allows us to rely upon statistics to average out memory competition, so it is probably reasonable to allow DWT at least 80% of storage bandwidth or about 16,000 bits/scan line in the above example, which would accommodate 1000 line x 1000 pixels/line x 16 bits/pixel. For HB = 5 μs this is equivalent to a pixel clock period of 26.7 ns.

This is only one speed limitation. Since the 32-bit wide FIFO is accessed once/cycle alternately by the A and B channels (i.e., 16 bits/cycle/channel), and since exactly three doublewords are fetched before the horizontal scan begins for each channel, the maximum bits/scan line for each channel is about (3*32 bits) + [(26.7 ns/pixel)*(16 bits/50 ns)*(1000 pixels/line)] = 8640 bits/scan line. This means that unless both channels are running at the same data rate, the data rate will be significantly below the upper bound determined above. For example, in 24Bit mode, if the A channel runs at full resolution and gets 8640 bits/scan line, the B channel will run at half resolution and get only 4320 bits/scan line, so the maximum data rate would be about 1000 lines x 538 pixels/line x 24 bits/pixel.

Ethernet Controller

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

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

Read this chapter with Figure 16 in view.

Ethernet Packets

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

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

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

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

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

Remarks

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

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

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

Controller Overview

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

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

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

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

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

The transmitter and receiver each have 16-word x 20-bit Fifos. The bits are 16 data + 2 parity + 2 spare (the receiver uses one of the spare bits). Each Fifo has read and write pointers, multiplexed into the address inputs of the storage chips, to select the next location to be read or written; these pointers are zeroed by IOReset. A Fifo is empty when the pointers are equal and full when (WritePtr + 1) mod 16 equals ReadPtr. There are *bus registers* between the Fifos and IOB. Service requests from the Ether side of a Fifo are given priority. The Fifos are synchronous to t_1.

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

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

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

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

Receiver

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

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

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

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

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

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

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

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

Transmitter

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

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

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

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

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

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

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

A collision may be detected by either the transceiver or PD. The occurrence of a collision is captured, synchronized, and used to abort the outgoing packet. The output of the first stage of the TxCollision synchronizer is wire-or'ed with PD output to jam the Ether after a collision. The jam lasts for one or two bit times, being the delay through the TxCollision synchronizer, TxFSM, and TxGo synchronizer.

Clocks

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

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

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

Task Wakeups

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

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

Muffler Input

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

Table 26: Ethernet Muffler Signals

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

Bus Registers

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

Control Register

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

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

Status Register

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

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

Other IO and Event Counters

In addition to the disk, ethernet, and display controllers discussed in earlier chapters, Dorado contains a *general input/output interface* and a *junk task wakeup* located on the IFU board; the two registers used in this interface may alternatively be used as *event counters* in performance monitoring, and that use is also discussed here.

Since the IFU board is not interfaced to the IOB, it cannot use the slow io system to control these features, so functions are used instead.

Junk Task Wakeup

The IFU board contains a circuit which wakes up the junk task (task 1) every 32 μs. The wakeup is permanently shutoff by loading the IFUTest register (IFUTest←B) with a 1 in the low-order bit. The junk task can dismiss the wakeup by doing IFUTest←B with any value on B (but B[15] must be 0 to reenable the wakeup at the next 32 μs tick).

Junk task microcode will, among other things, maintain a Real Time clock.

General IO

A 16-bit register called GenIn (synonym EventCntA) is used for general input; it can be read with the B←GenIn (synonym B←EventCntA) function but cannot be written by firmware. When used for general input, GenIn is written with information that is TTL-to-ECL converted from the backpanel.

A 16-bit register called GenOut (synonym EventCntB) is used for general output; it can be either read with the B←GenOut (synonym B←EventCntB) function or written with the GenOut←B (synonym EventCntB←B) function. GenOut is connected to the backpanel through ECL-to-TTL converters.

The plan is that devices such as Diablo printers can be connected to the GenIn and/or GenOut signals via backpanel connectors.

The choice of using one of these registers for general io or for event counting is determined by the InsSetOrEvent←B function discussed below.

Event Counters

The GenIn and GenOut registers can alternatively be used as event counters. They cannot, of course, be used simultaneously for general io. The registers are setup for either io or event counting by the InsSetOrEvent←B function, where B[0:15] are interpreted as follows:

If B[0] is 1, then InsSet[0:1] are loaded as discussed in the "Instruction Fetch Unit" chapter. If B[0] is 0, then its the general io/event counters as follows:

B[4] enables counting of EventCntA

B[5] enables counting of EventCntB

B[8:10] select the event type to be counted by EventCntA as follows:

0	True (i.e., every cycle)
1	Hold
2	Processor memory reference (not held)
3	Good IFUJump (i.e., not held and not an exception)
4	Miss
5-7	Backpanel events A, C, and E, respectively

B[12:14] select the event type to be counted by EventCntB as follows:

0	True
1	Hold
2	Successful IFU memory reference
3	IFUJump that wasn't ready
4	Miss
5-7	Backpanel events B, C, and D, respectively

B[15] causes the event to be counted for all cycles if 1 or only for emulator or fault task cycles if 0.

To use the event counters, you first stop them counting and read their current values; then you tell them what to count and start them counting and your system running. Note that they never get reset, but just keep counting from wherever they are--it's up to the user to worry about counter turnover.

The expected mode of operation is that the junk task will detect counter overflow and update double or triple-precision vectors in RM that count events; even if the counter is counting once per 50 ns cycle, counter wraparound only occurs every 3.28 ms, so a double-precision vector could count for at least 214 seconds and triple-precision for 163 days. Sample microcode for maintaining a double-precision counter is given in the example below:

```
*The double-precision vector consisting of two RM locations, CountHi and CountLo
*is initialized such that CountHi eq 0 and CountLo contains minus the value in
*the event counter, and another RM location called CountFlag is initialized to 0.
*The microcode below increments CountHi whenever the event counter cycles.
*At any instant, the high part of the total count is in CountHi and the low part
*is CountLo+event counter; CountHi has to be incremented by 1 if the counter
*just overflowed.

(CountLo) - (EventCntB') - 1;                 *CountLo + event counter
Pd←CountFlag, Branch[.+2,alu>=0];
CountFlag←T-T-1, Branch[.+3];                 *Set CountFlag to -1 in 2nd half of the counter cycle.
CountFlag←T-T, Branch[.+2,alu>=0];            *Set CountFlag to 0 in 1st half of the counter cycle,
CountHi←(CountHi)+1;                          *and increment CountHi, if we were in the 2nd half
. . .                                         *of the counter cycle last time.
```

The microcode for reading the counter when it is updated like this is as folows:

```
*Return to caller high part of event count in T, low part in Q.

TaskingOff, Pd←CountFlag;
T←(CountLo) - (EventCntB') - 1, Branch[.+3,alu>=0];  *CountLo + event counter = low part of result
TaskingOn, Branch[.+3,alu<0];                 *Low part ovf iff CountFlag<0 and low sum >=0
T←(CountHi)+1, Q←T, Return;                   *High part of result = CountHi+1
TaskingOn;                                    *High part of result = CountHi
```

```
T←CountHi, Q←T, Return;
. . .
```

Error Handling

In addition to single-error correction and double-error detection on data from storage, Dorado also generates, stores, and checks parity for a number of internal memories and data paths. The general concepts on handling various kinds of detected failures are as follows:

(1) Failures of the processor or control sections should generally halt Dorado because these sections must be operational before any kind of error analysis or recovery firmware can be effective.

(2) Failures arising from memory and io sections should generally result in a fault task wakeup and be handled by firmware. In some situations, such as map parity errors, it is especially important to report errors this way rather than immediately halting because firmware/software may be able to bypass the hardware affected by the failure and continue normal operation until a convenient time for repair occurs. In other situations, the firmware may be able to diagnose the failure and leave more information for the hardware maintainers before halting.

(3) IFU section failures and memory section failures detected by the IFU should generally be buffered through to the affected IFUJump, then reported via a trap; in this way, if it is possible to recover from the failure, then it will be possible to restart the IFU at the next opcode and continue.

(4) Memories and data paths involving many parts should generally be parity checked. It is not obvious that this is always a good idea because extra parts in the parity logic will be an additional source of failures, but instantly detecting and localizing a failure seems preferable to continuing computation to an erroneous and undetected result.

(5) When Dorado halts due to a failure, information available on mufflers and in the 16-bits of passively available error status (ESTAT) should localize the cause of the error as precisely as possible.

Since the MECL-10K logic family has a fast 9-input parity ladder component, the hardware uses parity on 8-bit bytes in most places; there is usually insufficient time to compute parity over larger units. IM and MIR, two exceptions, compute parity over the 17-bits of data in each half of an instruction; and the cache address section computes parity over the 15 address bits and WP bit.

Odd parity is used throughout the machine, except that the cache address section uses even parity. Odd parity means that the number of ones in the data unit, including the parity bit, should be odd, if the data is ok.

The control processor (Midas or the baseboard microcomputer) independently enables various kinds of error-halt conditions by executing a manifold operation discussed in the "Dorado Debugging Interface" document. It also has to initialize RM, T, the cache address and data sections, the Map, and IFUM to have valid parity before trying to run programs. Reasons for this will be apparent from the discussion below.

When Dorado halts, error indicators in ESTAT indicate the primary reason for the halt, and

muffler signals available to the control processor further define the halt condition; ESTAT also shows the halt-enables. Midas will automatically prettyprint a message describing the reasons for an error halt. The exact conditions that cause error halts are detailed in the sections below; the table here shows the ESTAT and muffler information which is relevant.

Table 27: Error-Related Signals

ESTAT Error Bit	*ESTAT Enable Bit*	*Task Experiencing Halt*	*Related Muffler Signals and Meaning*
RAMPE	RAMPEen	Task2Bk	STK, RM, or T parity failure. *RmPerr* and *TmPerr* mufflers on each processor board indicate which byte of RM/STK or T had a parity failure. *StkSelSaved* indicates that *RmPerr* applies to STK.
MdPE	MdPEen		processor-detected Md parity failure
		Task2Bk	if immediate ←Md (←MDSaved false)
		Task3Bk	if deferred ←Md (←MDSaved true)
			MdPerr muffler on each processor board shows which byte of Md failed.
IMrhPE	IMrhPEen	CTD	parity failure of IM[17:33]
IMlhPE	IMlhPEen	CTD	parity failure of IM[0:16]
IOBPE	IOBPEen	Task2Bk	Pd←Input parity failure if *IOBoutSaved* false
		Task2Bk	Output←B parity failure if *IOBoutSaved* true
			IOPerr mufflers on each processor board show which byte failed.
MemoryPE	MemoryPEen	--	cache address section parity failure, cache data parity failure on write of dirty victim or dirty Flush← hit., or fast input bus parity failure.

Processor Errors

The processor has parity ladders on each byte of the following:

input to RM/STK	generate parity for write of RM/STK
input to T	generate parity for write of T
B	generate parity for DBuf←B, MapBuf←B, Output←B, IM←B
IOB	check parity for Pd←Input and Output←B
Md	check parity for ←Md
R	check parity for ←RM/STK (unless bypassed from Pd or Md or replaced by ←Id)
T	check parity for ←T (unless bypassed from Pd or Md or replaced by ←Id)

Input ladders to RM/STK and T generate parity stored with data in the RAM; these ladders are not used for detecting errors.

The processor computes parity on its internal B bus (alub). The generated parity may be transmitted onto IOB when an Output←B function is executed; Store← references write B data and parity in the cache; parity for IM writes and map writes is computed from B parity. None of the other B destinations either check or store B parity. External B sources do not

generate parity.

Parity on the R/T ladders is checked only when the R/T data path is sourced from the RAM, not when bypassing from Md or Pd is occurring, and not when R/T is sourced from Id. A detected failure causes the *RAMPE* error halt, which indicates that some byte of RM, STK, or T had bad parity. The muffler signals that further describe this error are in the PERR word: *StkSelSaved* is true if the source for R was STK, false if the source for R was RM; each processor board has *RmPerr* and *TmPerr* signals; *RmPerr* is true if the RM/STK byte on that board had bad parity, *TmPerr* if the T byte had bad parity. Note that if an instruction beginning at t_0 suffered an error, Dorado halts immediately after t_4; the muffler signals apply to the instruction starting at t_0. The *Task2Bk* muffler signals show the task that executed the instruction at t_0.

Md parity is checked whenever ←Md is done; a failure causes the *MdPE* error-halt when enabled. The *←MDSaved* muffler signal in PERR is true when a deferred ←Md caused the error (T←Md, RM/STK←Md), false when an immediate ←Md (A←Md, B←Md, or ShMdxx) caused the error. On a deferred ←Md error, Dorado halts after t_6 and *Task3Bk* shows the task that executed the instruction starting at t_0; on an immediate ←Md, Dorado halts after t_4, and *Task2Bk* shows the task. The *MDPerr* muffler signals on each processor board show which byte of Md was in error.

Io devices (optionally) compute and send odd parity with each byte of data; the processor checks parity when the Pd←Input function is executed, but not when the Pd←InputNoPE function is executed. When enabled, an *IOBPE* error halts the processor at t_4 of the instruction that suffered the error; *Task2Bk* shows the task that executed the instruction. The processor also checks IOB parity on Output←B, and an error halts at t_4 as for Pd←Input. The *IOBoutSaved* muffler signal distinguishes Pd←Input from Output←B errors; an *IOPerr* muffler signal on each processor board shows which byte of IOB was in error; all of these are in the PERR muffler word.

The processor generally does not pass parity at one stage through multiplexing to the next stage, so any failure in the multiplexing between one stage and the next will go undetected (exception: B parity passed through to IOB).

For example, the processor could write Md parity sent by the cache into the T RAM, when T is being written from Md. Instead, however, it checks Md parity independently, but then recomputes the parity written into T with the input ladder. Hence, a parity failure detected on a byte of T can only indicate a failure in either (1) the input parity ladder; (2) the output parity flipflop; (3) the output parity ladder; (4) one of three 16x4 T RAM's; (5) one of two 4-bit latches clocked at t_1 (Figure 3) through which the output of the T RAM passes; (6) one of two 4-bit latches clocked by preSHC'.

Parity is handled similarly for writes of RM/STK.

Parity is similarly recomputed on B.

The processor does not generate or check parity on the A, Mar, or Pd data paths. Any failures of the A, Mar, B, Pd, or shifter multiplexing or of the ALU go undetected; failures of Q, Cnt, RBase, MemBase, ALUFM, or branch conditions go undetected.

Remark

Since 256x4 and 16x4 RAM's are used for RM, STK, and T, and since the processor is implemented with the high byte (0:7) on ProcH and the low byte (8:15) on ProcL, byte parity requires an additional 4-bit storage element on each board, of which only 1 bit is used. We could conceivably have used all 4 bits to implement a full error-correcting code for each byte of R and T data. However, there is insufficient time to correct the data. (Also, we use 256x1 RAM's instead of 256x4 RAM's for the RM and STK parity bits.)

Alternatively, parity could be computed over each 4-bit nibble rather than each 8-bit byte; the MC170 component allows nibble parity to be computed just as economically as byte parity. If this were done, then a parity failure would be isolated to a particular nibble. With byte parity, a detected failure could be any of 9+ components; with nibble parity, it would be isolated to one of 6+ components. Implementing nibble parity for RM/STK and T would require about 4 more ic's per board than byte parity.

It is hard to say whether the additional precision of nibble parity would be worth the additional parts.

Control Section Errors

The control section stores parity with each 17-bit half of data in IM. When IM is written, the two byte-parity bits on B are xor'ed with the 17th data bit to compute the odd parity bit written into IM. It is possible to specify that bad (even) parity be written into IM, and this artifice is used to create breakpoints; bad parity from both halves of IM is assumed to be a deliberately set breakpoint by Midas.

IM RAM output is loaded into MIR and parity ladders on each 17-bit half give rise to error indicators that, when enabled, will halt the processor *after* t_2 of the instruction suffering an error. For testing purposes, halt-on-error can be independently enabled for each half of MIR. Both the unbuffered output of the MIR parity ladders and values buffered at t_2 appear in ESTAT. The buffered values show the cause of an error halt, and the unbuffered signals allow Midas to detect parity errors in MIR before executing instructions or when displaying the contents of IM.

The special MIRDebug feature discussed in the "Dorado Debugging Interface" document prevents MIR from being loaded at t_2. In other words, when the MIRDebug feature is being used, all of the t_2 clocks in the machine will occur except the ones to MIR. This feature prevents the instruction that suffered an error from being overwritten at the expense of being unable to continue execution after the error. MIRDebug can be enabled/disabled by the control processor.

IFU Errors

The IFU never halts the processor; any errors it detects are buffered until an IFUJump transfers control to a trap location. The errors it detects, discussed in "IFU Section", are parity failures on bytes from the cache, IFUM parity failures, and map parity failures.

Memory System Errors

There is no parity checking on Mar or on data in BR, so any failure in the address computation for a reference goes unchecked. However, valid parity is stored with VA in the cache, and any failure detected will cause the MemoryPE error to occur, halting the system (if MemoryPE is enabled).

Parity is also stored in the Map (computed from B parity) and an error causes a fault task wakeup in most situations (Exceptions: IFU references and Map← references do not wakeup the fault task when a map parity error occurs).

The cache data section stores valid parity with each byte of data. When a munch is loaded

from storage, the error corrector carries out single-error correction and double error detection using the syndrome and recomputes parity on each 8-bit byte of data stored in the cache. When a word from B is Store←'d in the cache, byte parity on B is stored with the data.

A MemoryPE error occurs if, when storing a dirty victim back into storage, the memory system detects bad parity on data from the cache.

The IFU and processor also check parity of data from the cache, as discussed previously.

Sources of Failure

In a full 4-module storage configuration, Dorado will have 1173 MOS storage, about 700 Schottky-TTL, 3000 MECL-10K, and 60 MECL-3 ic's, and about 1500 SIPs (7-resistor packages). This logic is connected with over 100,000 stitch-welded or multiwire connections to sockets into which the parts plug; logic boards connect to the backpanel through about 2500 edge pins. Given all these potential sources of failure, reliable operation will be a surprising achievement.

Error Correction

The reliability situation is improved by error-correction on storage. The Dorado error-correction unit of 64 data and 8 check bits (quadword), guards 1152 MOS storage components from single failures but almost no other parts on the storage boards or in the error corrector are guarded.

Our Alto experience suggests that some machines repeatedly fail under normal use due to storage failures which diagnostics are unable to find. For this reason, error correction should be viewed as guarding against not only new failures but also imperfect testing of parts that are either already bad or subject to noise (e.g., cosmic rays) or other kinds of intermittent failure. The latter may be more important in our environment.

We have little information on which to base a reliability comparison of 16k or 64k MOS RAMs with other parts in Dorado. Although more complicated internally, it does not follow that these parts will be less reliable than others in the mainframe.

We do know that one large user of MOSTEK 4k MOS RAM's experienced .02% failures/KHrs, which would average 2 ic failures/year on a Dorado. If we do this well, a Dorado will run for years without uncorrectable storage failures. We also know that the dominant failure mode appears to be single-bit failures with row and column addressing failures affecting many bits somewhat less frequent, but we don't know the distribution of these.

If MOS failures do become significant, different strategies may be needed for single- and multi-address failure modes. With a multi-address failure, another failure in the same quadword causes a double error; but many single-address failures can occur in the same quadword without double errors.

The failure model used below shows that if no periodic testing and replacement of bad MOS RAM's took place, the fatal failure statistics of the 1152 MOS storage ic's would approximate the failure statistics of a 108 ic unerror-corrected store. By thoroughly testing storage and replacing bad parts 4 times more often than the mean time to total failure of a part (defined below), the likelihood of an uncorrectable storage failure crashing the system can be made insignificant compared with other sources of failure.

Although system software could bypass all pages affected by a multi-address ic failure, the entire module, 25% of storage, would be eliminated, so this is impractical except on an emergency basis. Continuing execution despite a multi-address ic failure will result in a double error when any other coincident storage failure occurs in the same quadword; 1/16 of future failures will do this.

Some interesting questions are: How does mean-time-to-failure vary with the EC arrangement? Mean-time-to-failure is pertinent if we let Dorados run until they fail. Alternatively, how likely is a failure in the next day, week, or month, if we test the memory that often and replace bad ic's? These questions can be asked assuming perfect testing (no failures at t=0) or imperfect testing (some likelihood of failures at t=0 because diagnostics didn't find them).

To answer them, MOS RAM failures are modelled as one of two types: those affecting a single address in the ic (called SF's), and those affecting all addresses (called TF's). We assume that TF's occur about 1/4 as often as SF's in 4Kx1 RAM's. Ic failures are assumed exponentially distributed, correct if the failure rate doesn't change with time; over the time range of interest, this is reasonable. Finally, perfect testing is assumed, so there are 0 failures at t=0. These assumptions give rise to the following:

let p = prob that an ic has a TF = $1 - e^{-at}$
let q = prob that an ic has a SF = $1 - e^{-bt}$
let n = number of MOS storage ic's in the memory

Without error correction, mean time to failure is the integral from 0 to infinity of $[(1-p)(1-q)]^n = 1/n(a+b)$. With b = 4a, in our 4-module system with n = 1024, this is 1/5120a = **.00018/a.**

With error correction, failure occurs when, in a single EC unit, a TF coincides with either another TF or an SF. This ignores two coinciding SF's which is about 1000 (4k ic's), 4000 (16k ic's), or 16000 (64k ic's) times less likely.

let n = number of ic's in an error correction unit
then Prob[no failure] = Prob[no TF] + Prob[1 TF and 0 SF]

Prob[no TF] = $(1-p)^n$
Since failure modes are independent,
Prob[1 TF and 0 SF] = $np[(1-p)(1-q)]^{n-1}$
Prob[no failure] = $P_{ok} = (1-p)^n + np((1-p)(1-q))^{n-1}$
$P_{ok} = e^{-nat} + n(1-e^{-at})(e^{-(a+b)(n-1)t})$

This is the probability for a single EC unit, so mean time to failure for all MOS storage is P_{ok} raised to a power equal to the number of EC units. In other words, the argument of

the integral for a 4-module x 4 quadwords/module system is P_{ok}^{16} with n = 64+8; it is P_{ok}^{4} with n = 256+10 for a one munch EC unit.

Then, expected time to failure for our 16 x n=64+8 memory system, is about:

$$(1/n) * (1/16a + 16a/(16a+b)^2 + 240a^2/(16a+2b)^3 + 3360a^3/(16a+3b)^4)$$
$$= (1/an) * (1/16 + 1/25 + 5/288 + 105/17208)$$
$$= (1/16an) * (1 + .64 + .28 + .006) = 1.93/16an$$
$$= 1.93/16*72*a = .00168/a$$

In other words, mean time to failure is about 1.93 times longer than the time to the first TF = 9.5 times better than with no error correction = as often as 1024/9.5 = 108 un-error-corrected storage ic's.

The results don't change much when imperfect testing is assumed. The effect of this is to replace densities for p and q by $1 - Ae^{-at}$, where A would be .999 if there was a 1/1000 chance of a MOS ic being bad at t=0.

Remarks

On each storage board, data from MemD is transported to a shift register consisting of 8 flipflops which are then written into the MOS RAM's after transport has been completed. This arrangement is unfortunate--any failure in one of these components will cause a multiple error, and there are about 250 of these parts in a full storage configuration.

One way to eliminate this problem while simultaneously reducing the part count on each storage board would be to make modules consist of four storage boards, rather than two, so that there are only four flipflops receiving data on each bit path during transport; since each of these is in a different quadword, single failures will not cause multiple errors.

The Dorado error-corrector (EC) operates on *quadwords*, requiring 8 check-bits/64 data bits, or a 12.5% storage penalty. Alternatives to the scheme we are using are: 10 check bits/256 data bits (3.9%); 9 check bits/128 data bits (7.4%); 7 check bits/32 data bits (22%); and no error correction at all (0%).

The implementation of the EC pipeline is such that wider error correct units significantly increase the time for a miss. The current quadword error corrector requires 7 clocks (3 clocks for setup and correction, 1 clock per word of the quadword); this would become 11 clocks with a 128-bit EC scheme or 19 clocks with a 256-bit EC scheme. Unless miss frequency will be less than we suspect, some implementation avoiding this delay would be needed before larger units of error correction became attractive.

If our quadword error-correct unit were replaced by a 4 x n=256+10 scheme:

$1/4na + 4a/n(4a+b)^2 + 3a^2/2n(2a+b)^3$, where for b = 4a this is
$(1/4na)*(1 + 1/4 + 1/36) = 1.28/4na = .0012/a$

In other words, mean time to failure is about 1.28 times longer than the time to the first TF. So error correction has increased time to failure by a factor of 6.2 over no error correction; alternatively, a 1064-ic error-corrected memory fails as frequently as a 1064/6.7 = 159 ic un-error-corrected memory.

Surprisingly, the 64+8 EC scheme has only 42% longer time to failure than a 256+10 EC scheme. My feeling is that this improvement is not worth the 96 additional MOS storage and 80 other ic's (required for address buffering) which it costs; if the 80 additional ic's have the *same* failure statistics as MOS storage (TF+SF), then they introduce 1.6 times as many system failures as they save, being a net loss.

The other method of maintaining our systems is to regularly test storage and replace bad ic's. Then the likelihood of no double error before replacement is simply the value of the probability distribution (P_{ok}^{4} and

P_{ok}^{16} above) at the selected instant. This reduces to an approximation of the form $P_{ok} = [e^{-x} + xe^{-x}]^m$ where x = nat, m is 4 or 16, and n = 72 for m=4 or 266 for m=16. If this is evaluated at t = 1/mna, 1/2mna, 1/4mna, etc. the following results are obtained:

Table 28: Double Error Incidence vs. Repair Rate

m	1/mna	1/2mna	1/4mna	1/8mna
4	.52	.81	.94	.98
16	.79	.84	.98	.99

The interpretation of this table is as follows: Measure mean time to total failure (TF) of a MOS ic and call this time 1/a; then assume 4 SF's per TF. Then the rate at which TF's occur in storage will be 1/mna. So the above tables show probability that the Dorado hasn't suffered a double error when tested and fixed as often, 1/2 as often, 1/4 as often, or 1/8 as often as the mean rate of TF's.

Performance Issues

This chapter discusses two issues:

(1) How rapidly will Dorado be able to execute Mesa, Lisp, SmallTalk, etc. macroprograms;

(2) What relationship do some of the design parameters bear to performance;

General Performance Issues

The first issue is cycle time. Dorado has been designed for a 50 ns cycle time; there are presently several paths that are slower based upon worst case calculations, so we will be lucky to achieve 50 ns.

Next, what *small-scale* features or changes would significantly reduce the number of instructions executed by Mesa and other opcode sets? A paper study of Dorado Mesa microcode with opcode execution frequency statistics extrapolated from Alto Mesa suggests that on Dorado Mesa about 92% of all executions will require one or two instructions each and about 3% somewhere between 25 and 100 instructions. The last are various forms of procedure call and return (XFER).

No further small-scale modifications considered would reduce instructions executed for the 92% of Mesa executions that already finish in only one or two cycles. XFER might be speeded substantially, and the MemBX kludge has been added in the hope that it will be useful. However, larger improvement might be possible through respecifying what these opcodes do. In any case, it is unclear how best to make improvements in this area.

For SmallTalk and Lisp instruction sets, performance is much worse than Mesa. Careful studies should be made to understand the reasons for this fully, but one comment can be offered now. For Lisp the 16-bit word size is a serious limitation because long storage pointers are used extensively; the instruction set would run substantially faster on a machine with, say, 32-bit data paths.

Speed is not the only issue--some reduction in microstore requirements might be possible through design changes. Our current projection on space requirements for programs is as follows:

Table 29: Utilization of the Microstore

Mesa opcode set	750
Alto opcode set	650
Lisp opcode set	650
SmallTalk opcode set	650
BitBlt subroutine	300
Multiply, divide subroutines	60
Fault handling	350
Ethernet driver	100
Disk driver	200
Display driver	200

Junk io driver 100

Since we do not require that more than two emulators be loaded in the microstore at one time, there may be a little space left for application programs. However, MicroD might not be able to utilize the microstore completely, so code compactness will be important.

The third performance issue is cache efficiency and miss wait; the fourth is available io bandwidth and io task cycle consumption. These are discussed in sections below.

Cache Efficiency and Miss wait

The value of shortening the wait for a storage read is roughly proportional to miss likelihood. Suppose that the prototypical opcode was a one-byte opcode implemented by the following microcode:

```
Fetch←IFUData, Stkp←Stkp+1;
Stack←Md, IFUJump[0];
```

For this example, execution time on a hit is 2 cycles; on a miss, 28 cycles. Delay for IFU misses must be added to this. Since the IFU is 6 bytes ahead of the current opcode, its misses delay 28 cycles less execution time for preceding 6 bytes; if any of the 6 bytes itself causes a miss, IFU delay will be 0 because it will catch up; the IFU never gets two misses (in this example) because it crosses at most one munch boundary. Hence, execution time will be $2 + 26*(1-H) + (28-12)*H^6*(1-H)$, with the following results:

Table 30: Execution Time vs. Cache Efficiency

Hit %	*Execution Cycles*	*IFU Cycles*	*% Miss Wait*
100	2.00	.00	0
98	2.52	.28	29
96	3.04	.50	44
94	3.56	.67	53
92	4.08	.79	59
90	4.60	.85	64
88	5.12	.89	67

This analysis, though crude, shows that cache efficiency dominates other factors in determining system performance; we will probably want to expand the Dorado cache to 16k words, and we have made provision for this.

Performance Degradation Due to IO Tasks

To first approximation, only the display controller word task (DWT) uses enough storage bandwidth to interfere significantly with emulators. Since it uses the fast io system, DWT requires service once/munch and will require two instructions/wakeup in the ordinary case. In addition, if the next instruction (by another task) issues a memory reference, it will always be held one cycle while the DWT's IOFetch← advances ASRN.

A quick calculation shows that at an io bandwidth of 256 x 10^6 bits/sec (10^6 munches/sec) the display controller will use 40% of storage bandwidth and 10% of processor cycles at 50 ns/cycle.

The earlier example showed that with no io interference and a 95% hit rate, the emulator spent 50% of cycles in miss wait, 50% in useful execution. With a 256 x 10^6 bit/sec dislay active, emulator misses are slowed about 2 cycles each, so the overall effect of the display is that about 43% of all cycles are emulator executions, 10% display task executions, and 47% are in hold; the one cycle holds for IOFetch← will make performance somewhat worse than this.

An IOFetch← by the display task to the same cache row as an emulator miss will remain in the address section, increasing display task latency and requiring more buffering. However, this won't degrade emulator performance.

The Alto monitor only uses 14.7 x 10^6 bits/sec (1/17 of the above) and would not interfere appreciably with emulators.

The disk controller is the fastest "slow" io device among standard peripherals. When running, its word interrupt task reads a double word from the cache every 3.2 μs in a 3 instruction/interrupt inner loop, consuming about 5% of all cycles at 50 ns/cycle. Its memory references consume the cache at a rate of .04 munches/μs, low enough that storage interference with the emulator isn't significant. However, a 256-word disk transfer displaces about 1/16 of the cache entries, so the emulator may experience a slightly lower hit rate.

Cache and Storage Geometry

As discussed earlier, even with 95% cache efficiency, memory wait could account for half of execution time. The current geometry was chosen without measurements or simulation of programs (It isn't clear what measurements would have been helpful.).

The following parameters are relevant:

1 word as the unit of storage inside the memory pipeline;
16-word *munch*;
256 munches in the cache (expandable to 1024);
4 *columns* in the cache.

Munch Size

A 16-word munch size was chosen primarily because 8 cycles for transport balances 10 cycles for storage access, avoiding loss of bandwidth. The use of 256x4 RAM's to implement the cache address section allows the original 4k-word cache (implemented with 1kx1 RAM's) to be expanded to 8k words or 16k words, when 4kx1 RAM's are economically available--this is possible because only 64 of the 256 words in the address section are being used with the 4k-word cache. Miss wait is about 28 cycles and storage bandwidth about 640 x 10^6 bits/sec with 16-word munches.

8-word munches would lower the storage bandwidth to about 320 x 10^6 bits/sec, probably unacceptable. Also 8-word munches would limit cache expansion to 8k words. However, miss wait would be reduced to about 24 cycles because transport would require only 4 cycles.

32-word munches would not allow greater storage bandwidth to fast io devices because bandwidth is already limited by transport with 16-word munches. Nor would it allow expansion to a larger cache data section because we have no way to build a data section larger than 16k words. Also, miss wait would be slowed to 36 cycles, so it does not seem that this munch size is attractive.

An interesting factor *not* discussed above is hit percentage. For a given size of the cache data section, with smaller munches the cache will tend to stabilize with a larger amount of useful information; however, when a program is changing contexts, larger munches might bring the new context into the cache more quickly.

Also, fast io tasks will interfere less with the emulator on larger munches because fewer wakeups and IOFetch←'es will be required. However, the extra buffering and longer miss wait offsets this advantage somewhat.

Considered together, these factors suggest that the 16-word munch we are using is substantially better than either 8 or 32-word munches.

Data Path Width

Having only 16 bit wide data paths slows misses. Doubling the paths to 32 bits would reduce EC time by 1 cycle and transport time into the cache by 4 cycles (i.e., delay on misses would be 23 cycles instead of 28). There were not enough edge pins to do this. However, if a method of doubling the path width were found, the storage system would probably be arranged as two modules of four storage boards each rather than four modules of two boards each, and 32-word munches might be better than 16-word munches.

Cache Columns

The reason for multiple columns is to approximate LRU reloading; the columns are moderately expensive because separate hit logic has to be provided for each one; the V-NV stuff also costs a few ic's with more than two columns. Altogether the current 64x4 cache is about 40 ic's larger than a 128x2 cache (Because of its 50-50 LRU behavior on the fourth column, our cache is somewhere between the 64x4 and 128x2 or 128x3 caches below.). The table below shows likelihood that the Nth LRU munch is no longer in the cache for various geometries:

Table 31: Cache Geometry vs. LRU Behavior

N	32x4	64x2	128x2	32x3	64x3	128x3	64x4	128x4
4	.000	.001	.000	.000	.000	.000	.000	.000
8	.000	.006	.002	.002	.000	.000	.000	.000
16	.001	.025	.007	.013	.002	.000	.000	.000
32	.017	.089	.026	.077	.014	.002	.002	.000
64	.140	.264	.090	.323	.079	.014	.018	.002
128	.570	.596	.264	.767	.323	.080	.141	.019
256	.960	.910	.595	.987	.764	.323	.568	.142
512	--	--	--	--	--	.763	.959	.567

These numbers are computed from a binomial distribution using the following formulae:

let R = rows in cache
let C = columns in cache
then p = (R-1)/R = probability that a munch of VA is in its row
then q = 1/R = probability that a munch of VA is not in its row
then probability of a miss for the nth element is:

C	P(miss)
1	$1 - p^n$
2	$1 - p^n - nqp^{n-1}$
3	$1 - p^n - nqp^{n-1} - n(n-1)q^2p^{n-2}/2!$
4	$1 - p^n - nqp^{n-1} - n(n-1)q^2p^{n-2}/2! - n(n-1)(n-2)q^3p^{n-3}/3!$
etc.	

Without extensive measurements on programs, it is impossible to know how much better, say, a 32x4 cache is than a 64x2 cache, or to know whether a 128x2 cache is better or worse than a 32x4 cache, for example. If a particular program is confining itself to a very small set of munches, then more closely approximating LRU reloading is most important. However, if the likelihood of reference flattens out after a small N, then it won't matter much that LRU reloading isn't very well approximated--the total size of the cache will be a more important determinant of performance.

Glossary

bypassing - a number of memories and task-specific registers in Dorado (RM, STK, and T, for example) are written with data that might be needed before the write occurs. These are implemented so that data about-to-be-written is substituted for data read from the register or memory when appropriate. This substitution is called *bypassing* and enables Dorado to run considerably faster than would otherwise be possible.

cache entry - a munch together with VA of the munch and 4 flag bits. For a 64 row x 4 column cache, VA[28:31] are the word in the munch, VA[22:27] address the row, and VA[7:21] are stored in the cache entry.

column - one of 4 groups of 64 (expandable to 256) cache entries. The cache column in which a word with VA resides is determined by comparing VA[7:21] with the corresponding bits stored in the four columns at row VA[22:27]. Thus a memory word may occupy one of 4 locations in the cache.

control processor - the microcomputer on Dorado's baseboard, or the Midas program operating Dorado from an Alto.

dirty - a *cache entry* is dirty if the information in it differs from information in storage, because a store has been done into the cache, and storage has not yet been updated. A *page* is dirty if a store has been done into the page since its map dirty bit was cleared.

emulator - the lowest priority task, number 0, always awake. The emulator is distinguished by the fact that it cannot block, can use Stk, and has a private pipe entry. Primarily the emulator task will implement instruction sets.

entry vector - the exit microinstruction of an opcode sends control to the first microinstruction of the next opcode by means of IFUJump[n] (n = 0 to 3), where n chooses one of 4 entry microinstructions for the next opcode; these four microinstructions are the next opcode's *entry vector*.

fault task - the highest priority task, number 15, woken whenever a memory fault or stack error occurs.

hit - a reference which finds the desired word in the cache.

Midas - the Alto program used for loading and debugging Dorado remotely.

miss - a reference which does not find the desired word in the cache.

module - the unit in which storage is packaged, either 64K, 256K, or 1M words. A machine may have 1 to 4 modules.

munch - 256 bits, or 16 machine words; the unit of data for main storage.

parity - the parity of a data unit is the exclusive-or of all bits in the data unit; parity has the property that changing any single bit in the data unit will also change the parity, so it can be used to detect single failures. A data unit has *odd parity* when the number of 1's in the

unit is odd, *even parity* when the number of 1's is even. Dorado uses odd parity everywhere, which means that the number of 1's in the data unit including its associated parity bit should be odd when data is correct.

PC - "program counter". In this manual PC refers to the 16-bit byte displacements relative to BR 31 (the codebase) which are maintained by the IFU for the current instruction set. This term should be distinguished from TPC, which refers to the address of the next microinstruction for a task.

pipe - a 16-entry memory which records the state of the last few storage references.

RAM - "random access memory"; selected words in the memory can be both read and written.

reference - a reference to the memory, initiated by the processor or by the IFU. A *processor reference* transfers a single word between the cache and the processor; an *io reference* transfers a munch between storage and an io device.

ROM - "read-only memory"; the contents of the memory are specified when the hardware is constructed and cannot be modified during program execution. ROM elements used on Dorado can be reprogrammed with a special device constructed for the purpose.

row - one of the 64 or 256 groups of 4 cache entries. The cache row in which a word resides is determined by bits 20..27 of its virtual address.

storage - the main memory of the machine, organized in munches of 256 bits, or 16 machine words.

storage reference - a reference to the storage, initiated as a result of a memory reference. A processor reference causes a storage reference if there is a cache miss or if the *FDMiss* control is true in the memory control register; an io reference always causes a storage reference.

storage reference number (SRN) - an address of a pipe entry which identifies a particular storage reference.

subtask - a two-bit number presented by an io device to the processor and memory system while its task is running. The processor OR's subtask with RBase[3]..RSTK[1] in determining the RM address and with MemBase[2:3] in determining the base register selection. The memory system buffers the subtask for fast io devices, and then sends it over the Fin or Fout bus as part of device identification.

tag - The extra bit in Md readout which complements for successive Fetch←'es and Store←'s by the same task. Agreement of the bit in Md with the current value equals reference finished.

task - one of the 16 priority scheduled tasks. Special tasks are the emulator (task 0, lowest priority) and the fault task (task 15, highest priority). Other tasks are paired with io controllers.

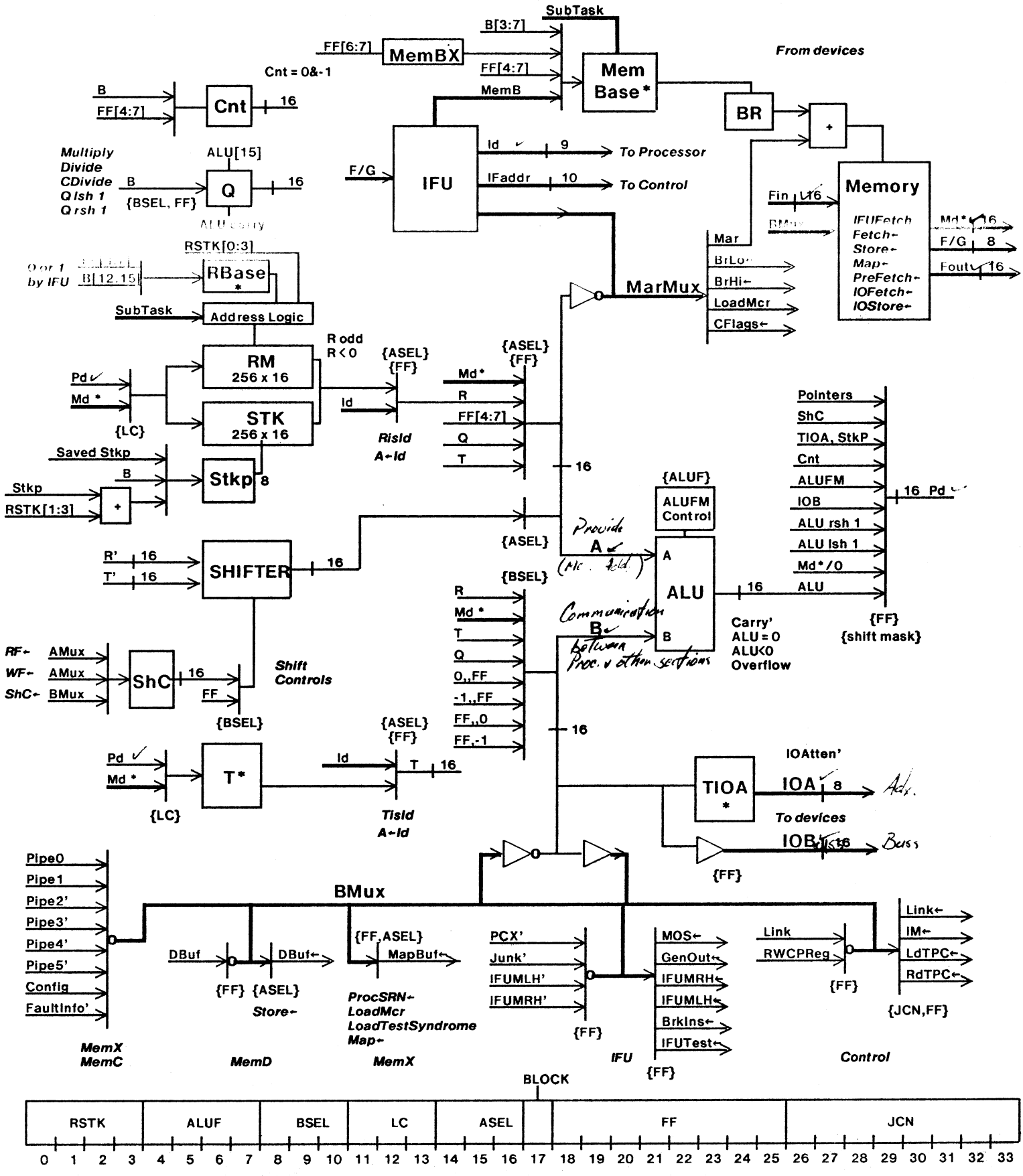
VA - virtual address.

Vacant - a cache entry or map entry which does not contain valid data.

Victim (Vic) memory - stores 4 bits for each cache row. Two of the bits specify the *victim* which will be chosen if a reference to that row results in a miss, and the other two are the next victim.

victim - on a processor reference that causes a cache miss, the cache entry chosen to be replaced by the referenced data.

WP - write protected. Map entries and cache entries have bits with this name.

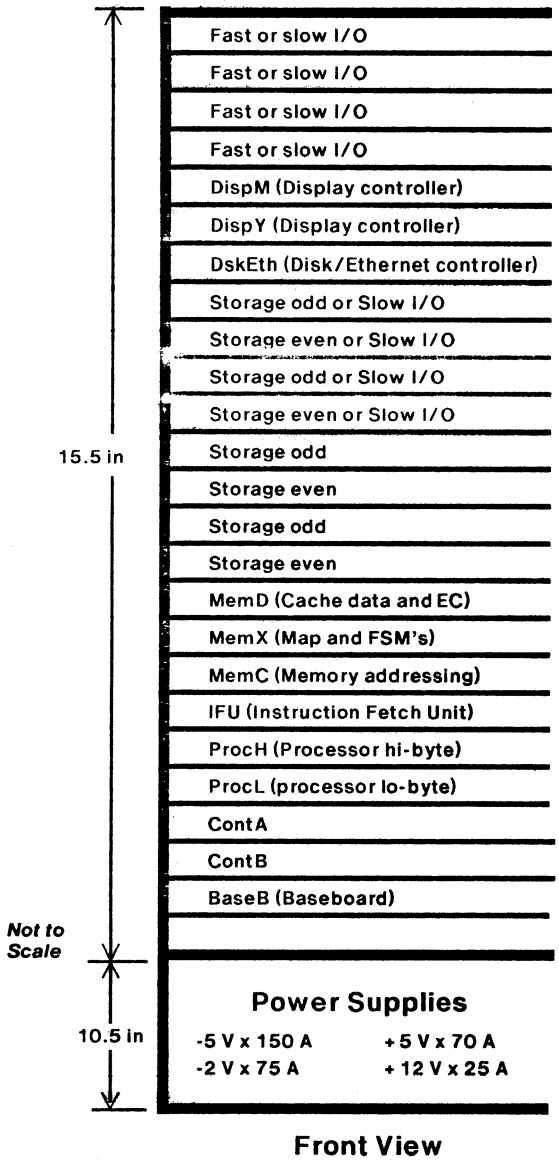


* Task-Specific

{xxx} Source of Control

Figure 1
Dorado
Programmer's View

D1ProgView.sil
8/31/79



Front View

.625 in

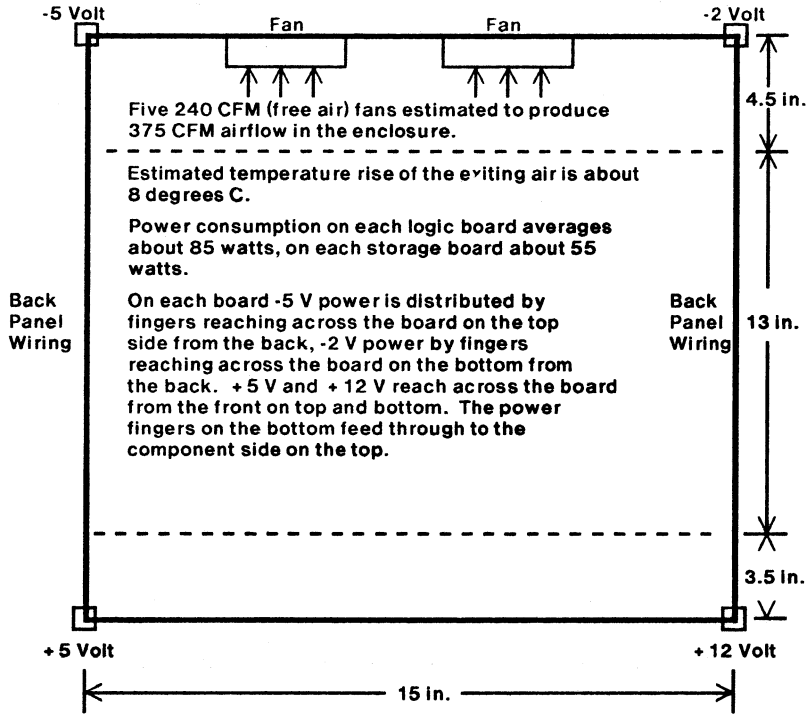
The +5 V supply and one fan are controlled by a switch; -5 V, +12 V, and -2 V supplies, the other four fans, and the disk logic and spindle power are controlled by the baseboard microcomputer (or the controlling Alto).

BaseB and ContB boards are equipped with temperature sensors that are repetitively monitored by the baseboard microcomputer; most other boards have temperature sensors that can be monitored when the microprocessor is halted. In the event some temperature exceeds 60 degrees C, the microcomputer will shut down the three power supplies that it controls.

The microcomputer also monitors power supplies; when any voltage or current deviates from its allowed range, the microcomputer shuts off power to the three supplies that it controls.

The card cage shown here is beneath the Trident T80 disk, and both are inside an enclosure designed to reduce the noise level for an office environment. The total enclosure size is about 4 feet high x 4 feet deep x 2 feet wide (ugh).

The machine weighs between 500 and 600 lbs



Top View

The 11 logic boards in production models will be multiwire; 2 to 8 storage boards and the two backpanels are printed circuits.

The following shows approximate component count:

11 Logic boards:		*8 Storage boards:*	
2315	ic's of random logic	1056	ic's of random logic
246	1kx1 ECL RAM's	1152	16Kx1 MOS RAM's
71	16x4 ECL RAM's		
24	256x4 ECL RAM's		
21	16Kx1 MOS RAM's		
1600	SIP's		

Each board can mount 24 x 12 or 288 IC's. Normal MECL-10000 IC's are connected to the ground plane and -5 V supply. Logic nets are terminated through 100-ohm resistors at one or both ends to the -2 V supply. The resistors are in low-profile SIP's that mount bettween the IC's (144 SIP's per board).

The +5 V supply is used for TTL/ECL conversions and for TTL components. The MOS IC's on the memory storage boards and in the Map use the +12 V supply.

Figure 2
Card Cage

D1CardCage.sil
9/5/79

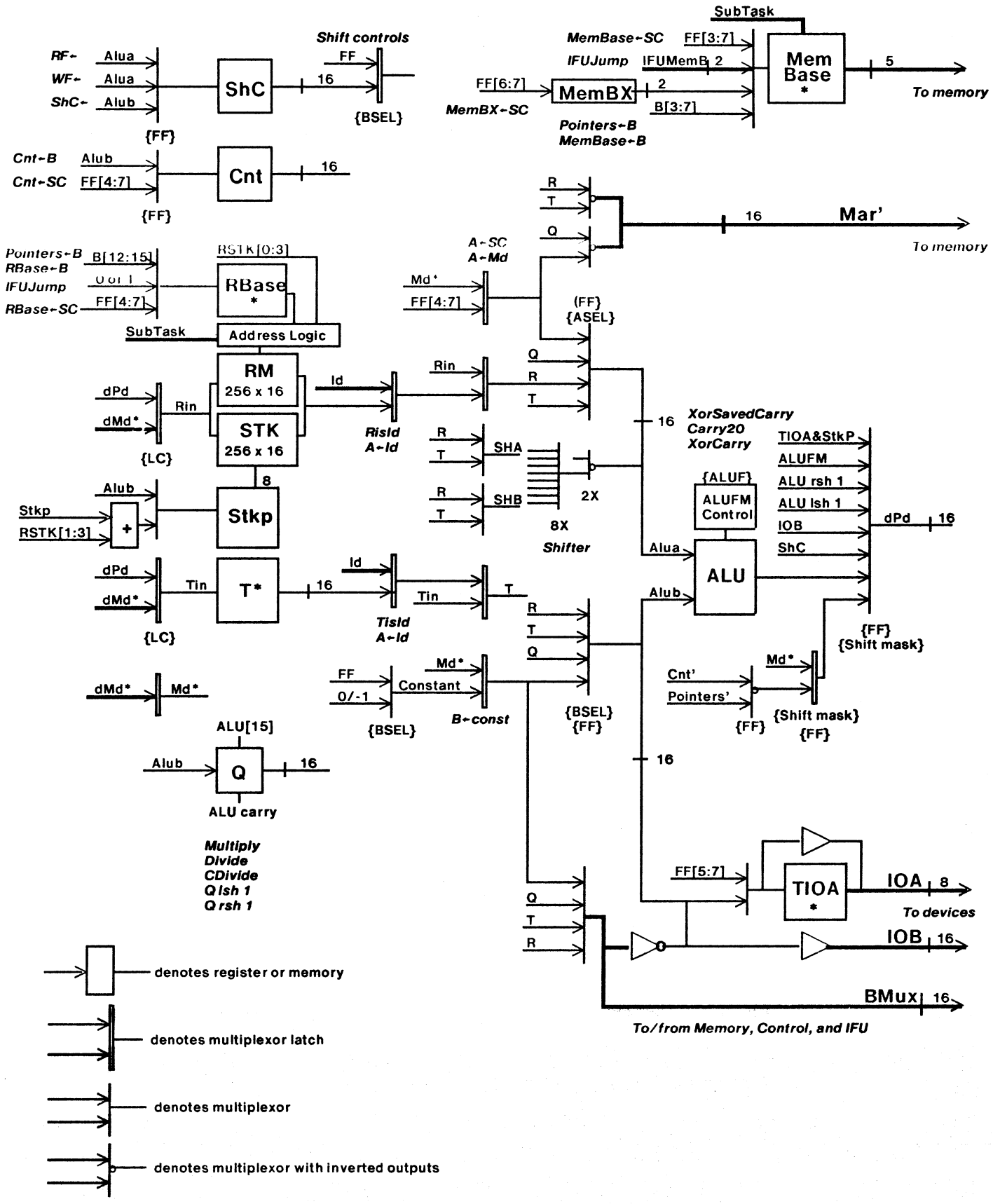
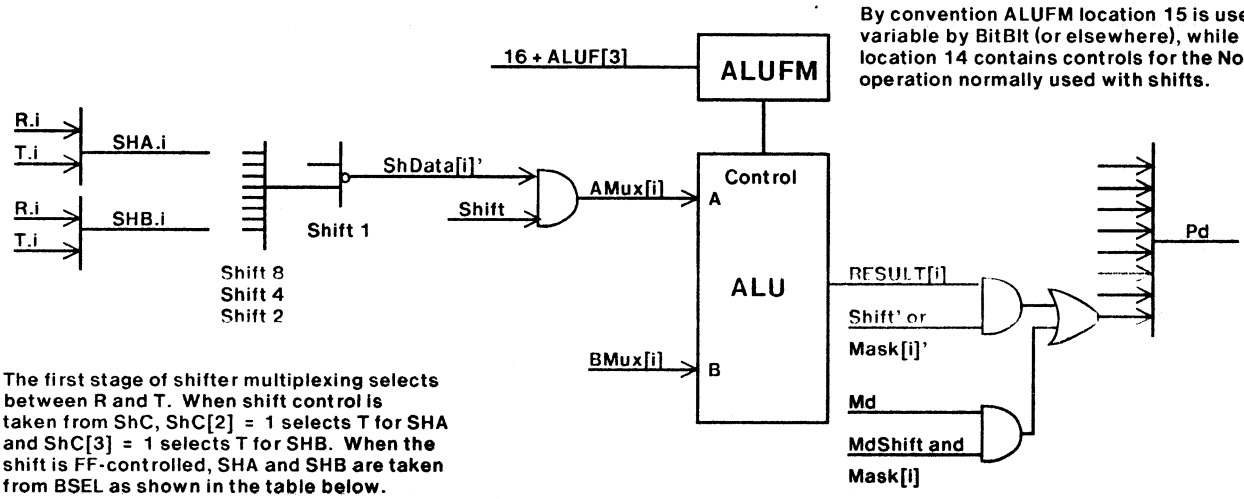


Figure 3
Processor
Hardware View

* Task-Specific

D1Processor.sil
8/31/79



By convention ALUFM location 15 is used as a variable by BitBlt (or elsewhere), while ALUFM location 14 contains controls for the Not-A operation normally used with shifts.

The first stage of shifter multiplexing selects between R and T. When shift control is taken from ShC, ShC[2] = 1 selects T for SHA and ShC[3] = 1 selects T for SHB. When the shift is FF-controlled, SHA and SHB are taken from BSEL as shown in the table below.

The 32-bit quantity SHA..SHB is then left-shifted through an 8-in multiplexor controlled by the shift 8, shift 4, and shift 2 controls.

The final stage is an inverting 2-in multiplexor which is disabled when no shift is taking place.

Mask[i] = LMask[i] or RMask[i]
ALUF[0:2] controls masking

The hardware actually uses two inputs of the Pd multiplexor when shifting. One of these is the normal ALU path, and the other is either Md (on a replace-with-Md shift) or 0. The multiplexor select is changed to the Md/0 path when the bit is being masked out.

Shift Data Paths

Field	SHA	SHB	Shift Count	RMask	LMask
ShC bits:	2	3	4:7	8:11	12:15
RF←A	A[2]	A[3]	P + S + 1	undefined	15-S
WF←A	A[2]	A[3]	16-P-S-1	16-P-S-1	P
ShC←B	B[2]	B[3]	B[4:7]	B[8:11]	B[12:15]
BSEL.0 = 1	BSEL.1	BSEL.2	FF[4:7]	FF[4:7]	FF[0:3]

Functions that load ShC

Shift controls come from ShC except when BSEL.0 is 1 in the microinstruction that shifts

Shift controls come from FF when BSEL.0 is 1, and the source for B is changed to Q.

P = A[8:11] = number of bits to the left of the field
S = A[12:15] = number of bits in the field - 1

The values for RMask, LMask, and Shift Count are and'ed by 17-octal.
The 32-bit quantity SHA[0:15]..SHB[0:15] are left-cycled by the shift count and the right-most 16 bits are the shift data.

RF← and WF← are intended for use with "reasonable" values of P and S.

Derivation of Shift Controls

Figure 4
Shifter

D1Shifter.sil
9/5/79

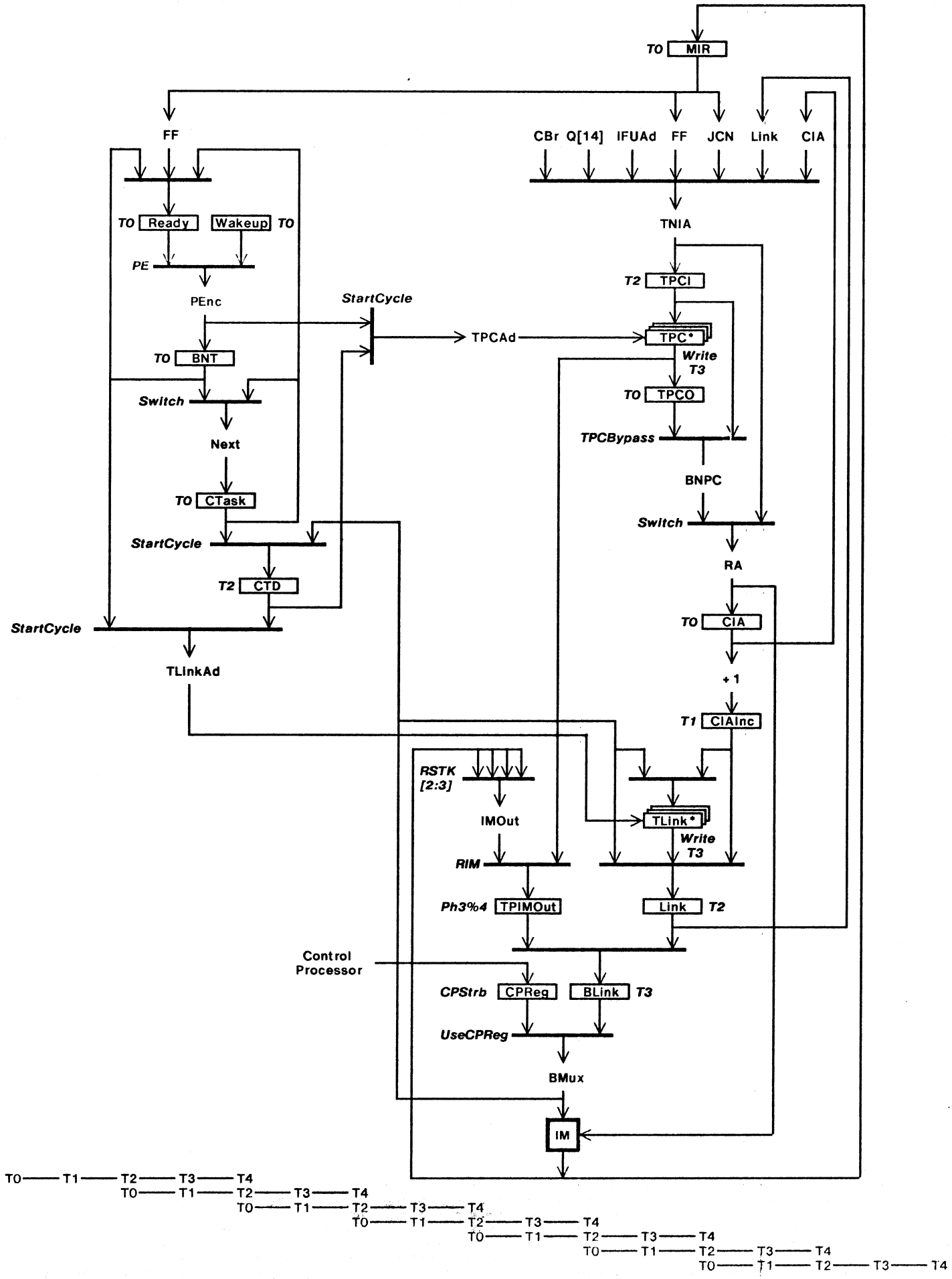
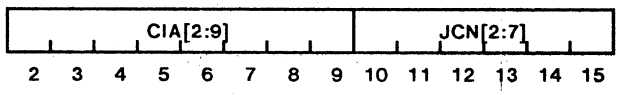
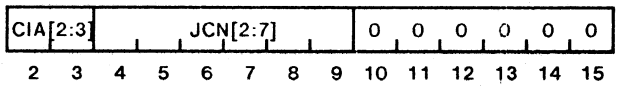
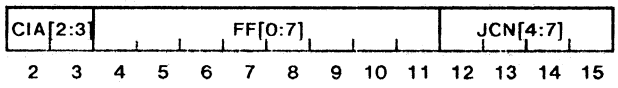
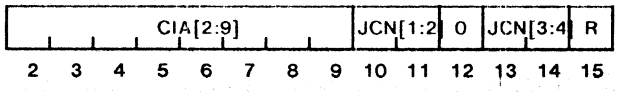
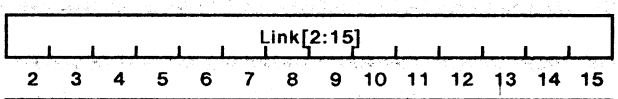
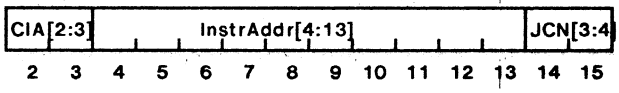


Figure 5
Control Section

D1Control.sil
9/5/79

JCN 0 1 2 3 4 5 6 7		TNIA:
1 0 ADDRESS BITS	Local Jump/Call	
1 1 ADDRESS BITS	Global Call	
0 0 0 0 *ADDRESS BITS*	Long Jump/Call	
0 *ADDRESS BITS # 000x* *BRANCH CONDITION*	Conditional Jump/Call	 *R is result*
0 1 *RETURN FUNCTION* 1 1 1	Return	
0 0 1 *NEXT NUMBER* 1 1 1	IFU Jump	
0 0 0 1 x 1 1 1	undefined	

A long, local, or conditional branch is a call iff, before any modification of TNIA by branch conditions or dispatches, TNIA[12:15] is 0; otherwise, it is a jump.

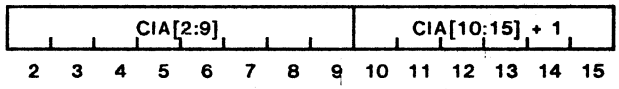
Conditional Branch

JCN[5:7]	*-or-*	*FF*	*BRANCH CONDITION*
0		60	ALU = 0
1		61	ALU < 0
2		62	Carry'
3		63	Cnt = 0&-1 (decrement Cnt after testing)
4		64	R < 0
5		65	R odd
6		66	IOAtten' (non-emulator) -or- Reschedule (emulator)
--		67	Overflow'

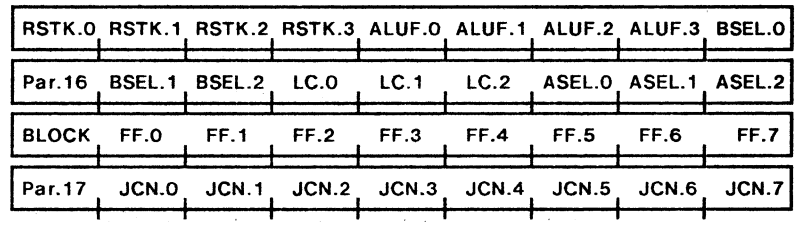
Return

JCN[2:4]	*FUNCTION*
0	Subroutine Return
1	unused
2	unused
3	unused
4	Read TPC
5	WriteTPC
6	Read Instruction Memory
	Address is in Link.
	Data appears on B[7:15] when B←Link executed in following microinstruction.
7	Write Instruction Memory
	Address is in Link. RSTK.3 is 1 to write the left half of IM, 0 to write the right half.

Loaded into Link by Call, Return, or IFUJump



RSTK[2:3]



Good (odd) parity is written if RSTK.1 is 0, else bad (even) parity is written.

The most significant bit of data is RSTK.2 and the least significant 16 bits are B[0:15].

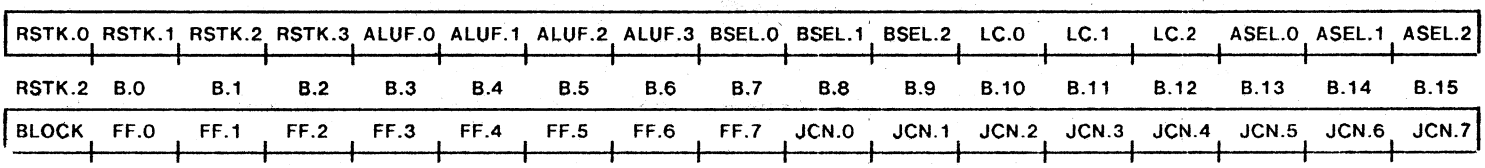


Figure 6

Next Address Formation

D1Branching.sil
8/31/79

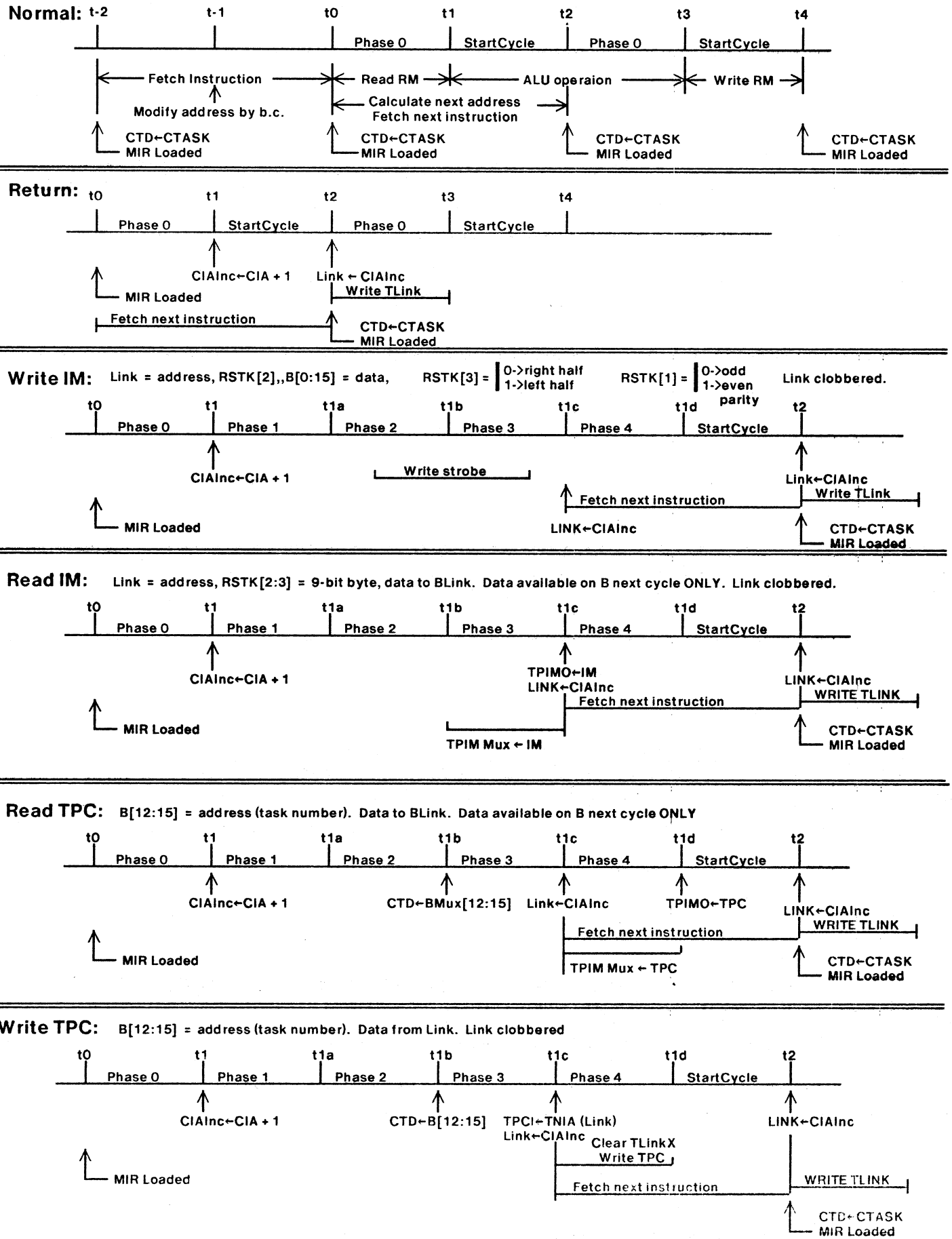


Figure 7
Instruction Timing

D1InstTiming.sil
10/8/79

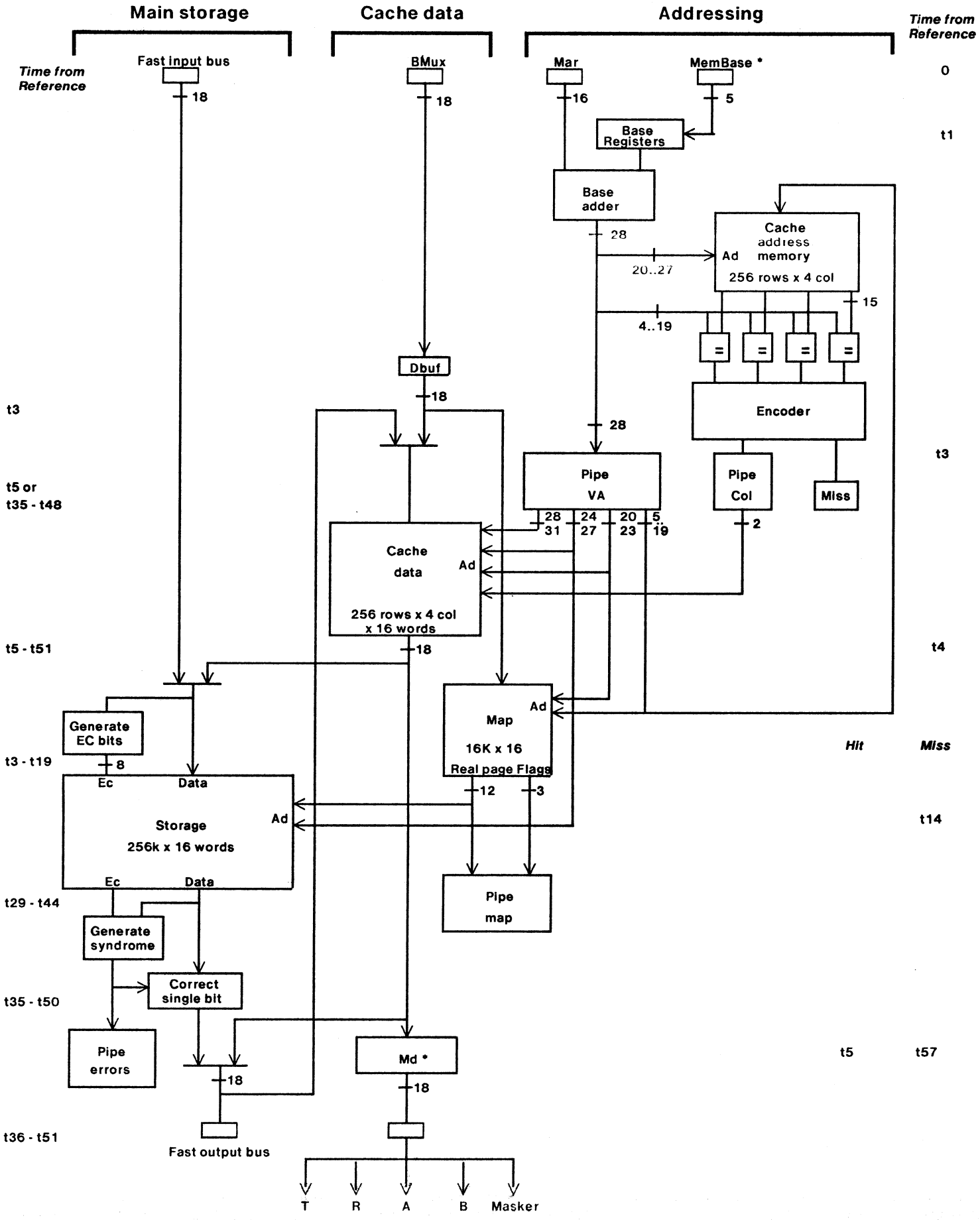
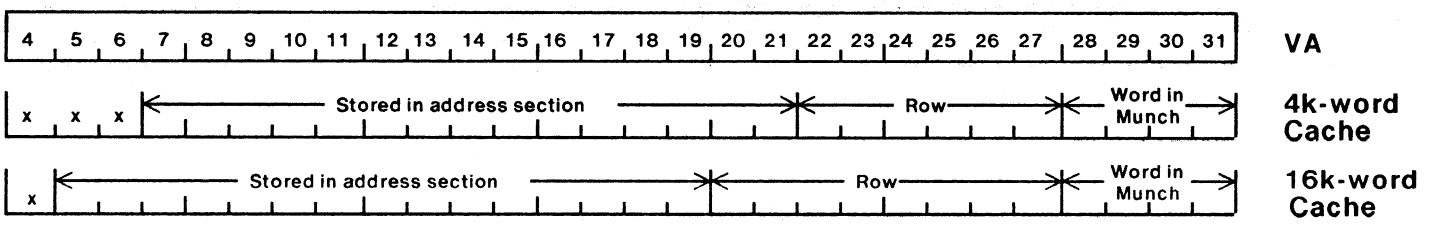
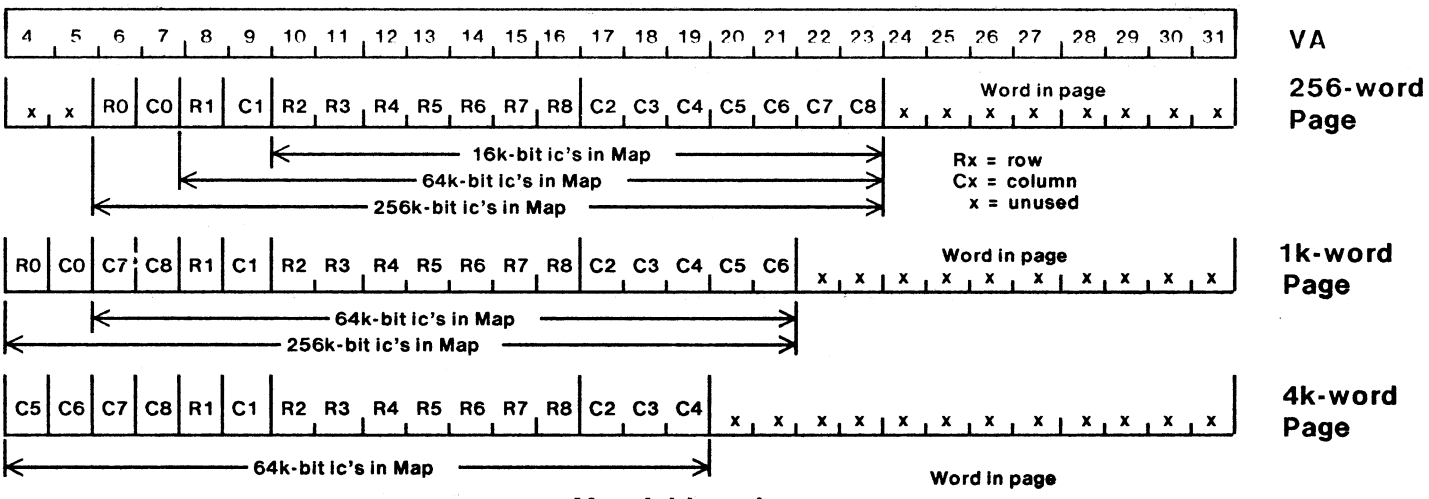


Figure 8
Overall Structure of the Memory System

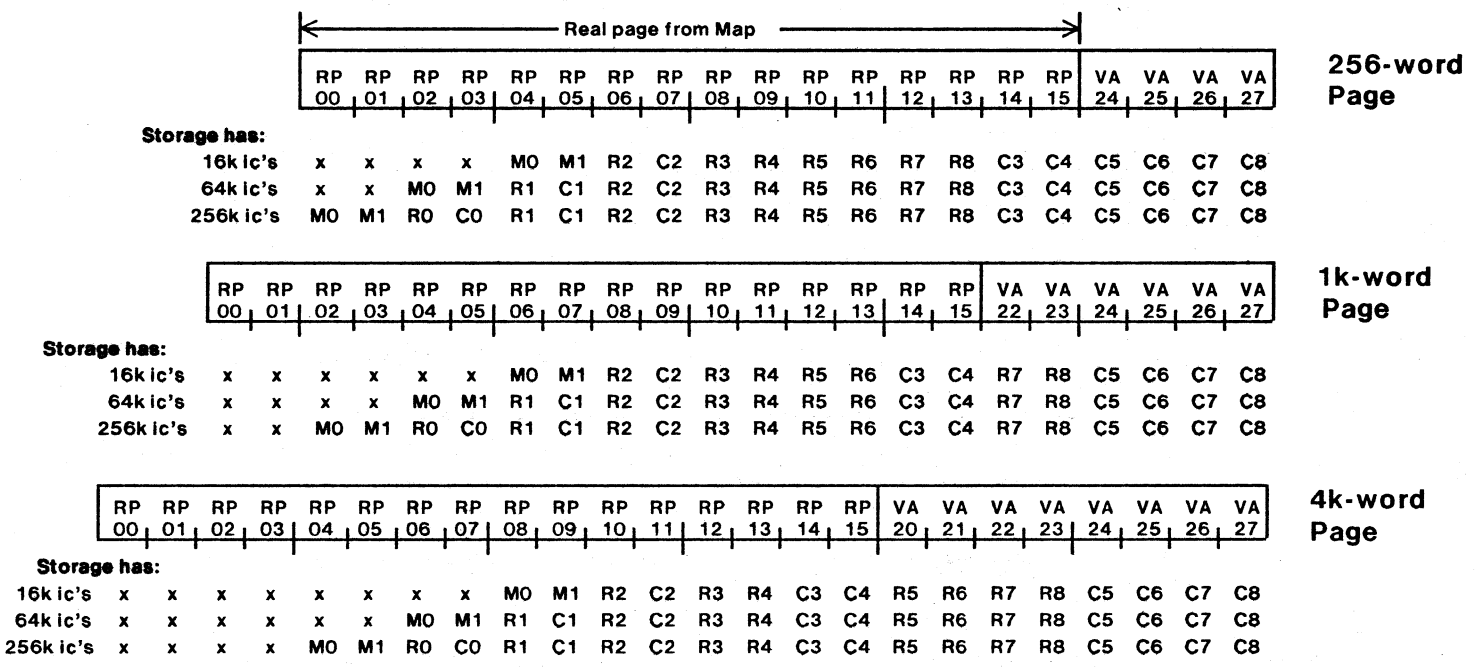
D1Memory.sil
9/5/79



Cache Addressing



Map Addressing



Storage Addressing

Figure 9

Cache, Map, and Storage Addressing

D1MemAddr.sil
8/31/79

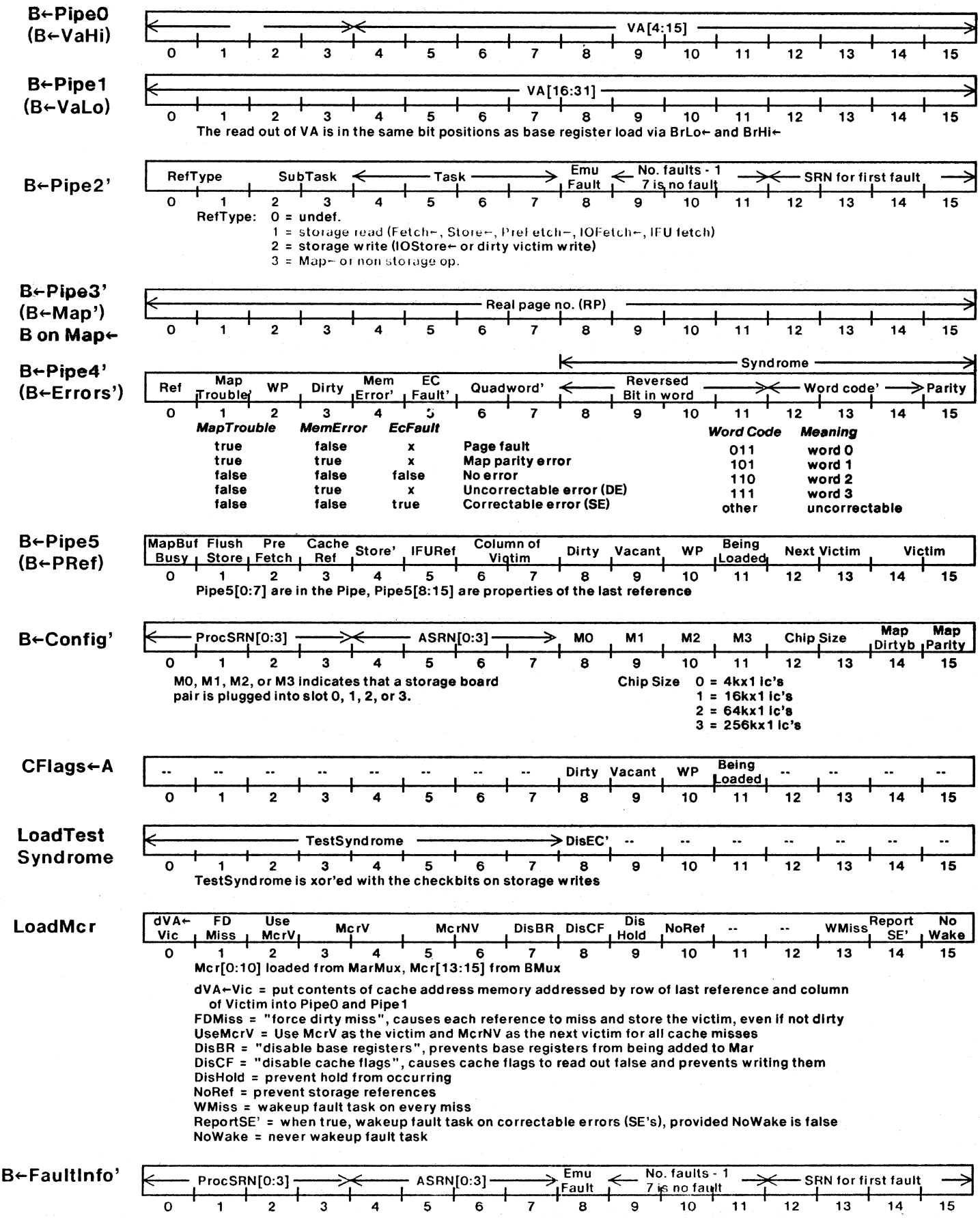
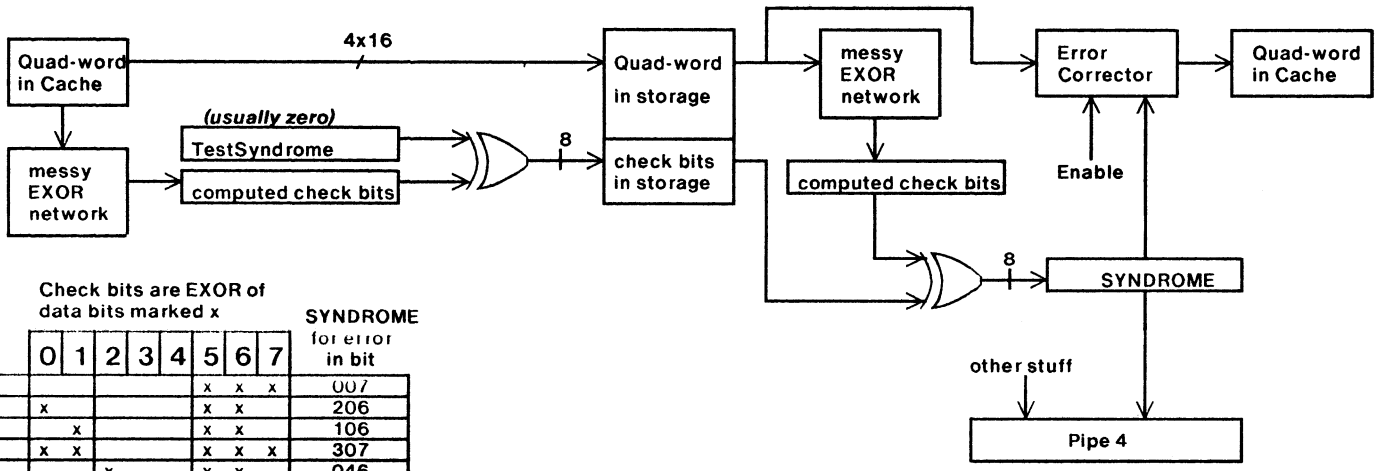


Figure 10

The Pipe and Other Memory Registers

D1MemReg.sil
9/28/79



Check bits are EXOR of data bits marked x

	0	1	2	3	4	5	6	7	SYNDROME for error in bit
WORD 0						x	x	x	007
	x					x	x		206
		x				x	x		106
	x	x				x	x	x	307
			x			x	x		046
	x		x			x	x	x	247
		x	x			x	x	x	147
	x	x	x			x	x		346
				x		x	x		026
	x			x		x	x	x	227
		x		x		x	x	x	127
	x	x		x		x	x		326
			x	x		x	x	x	067
	x		x	x		x	x		266
		x	x	x		x	x		166
	x	x	x	x		x	x	x	367
WORD 1					x		x	x	013
	x				x		x		212
		x			x		x		112
	x	x			x		x	x	313
			x		x		x		052
	x		x		x		x	x	253
		x	x		x		x	x	153
	x	x	x		x		x		352
				x	x		x		032
	x			x	x		x	x	233
		x		x	x		x	x	133
	x	x		x	x		x		332
			x	x	x		x	x	073
	x		x	x	x		x		272
		x	x	x	x		x		172
	x	x	x	x	x		x	x	373
WORD 2					x	x		x	015
	x				x	x			214
		x			x	x			114
	x	x			x	x		x	315
			x		x	x			054
	x		x		x	x		x	255
		x	x		x	x		x	155
	x	x	x		x	x			354
				x	x	x			034
	x			x	x	x		x	235
		x		x	x	x		x	135
	x	x		x	x	x			334
			x	x	x	x		x	075
	x		x	x	x	x			474
		x	x	x	x	x			174
	x	x	x	x	x	x		x	375
WORD 3					x	x	x		016
	x				x	x	x	x	217
		x			x	x	x	x	117
	x	x			x	x	x		316
			x		x	x	x	x	057
	x		x		x	x	x		256
		x	x		x	x	x		156
	x	x	x		x	x	x	x	357
				x	x	x	x	x	037
	x			x	x	x	x		136
		x		x	x	x	x		236
	x	x		x	x	x	x	x	337
			x	x	x	x	x		076
	x		x	x	x	x	x	x	277
		x	x	x	x	x	x	x	177
	x	x	x	x	x	x	x		376

Interpretation of SYNDROME

If SYNDROME is	and number of ones in Syndrome is	and furthermore	THEN
0	ALWAYS	ALWAYS	NO ERROR
Not 0	ODD	syndrome bits 4,5,6 have 2 or 3 ones on	SINGLE ERROR (data bit) Bits 4,5,6 give bad word: 456 | word 011 | 0 101 | 1 110 | 2 111 | 3 Bits 3,2,1,0 give bad bit: 3210 | bit 0000 | 00 0001 | 01 ⋮ | ⋮ 1110 | 14 1111 | 15 Bad bit will be corrected if error correction is enabled
Not 0	ODD	exactly 1 one in Syndrome	SINGLE ERROR (check bit) Syndrome | bad check bit 200 | 0 100 | 1 040 | 2 020 | 3 010 | 4 004 | 5 002 | 6 001 | 7 No data bits will be changed.
Not 0	ODD	syndrome bits 4,5,6 have 0 or 1 ones on	TRIPLE ERROR!! (but no data bits will be changed) Syndrome is nonsense.
Not 0	EVEN	ALWAYS	DOUBLE ERROR No data bits will be changed. Syndrome is nonsense.

Figure 11

Error Correction

D1ErrCorr.sil
9/5/79

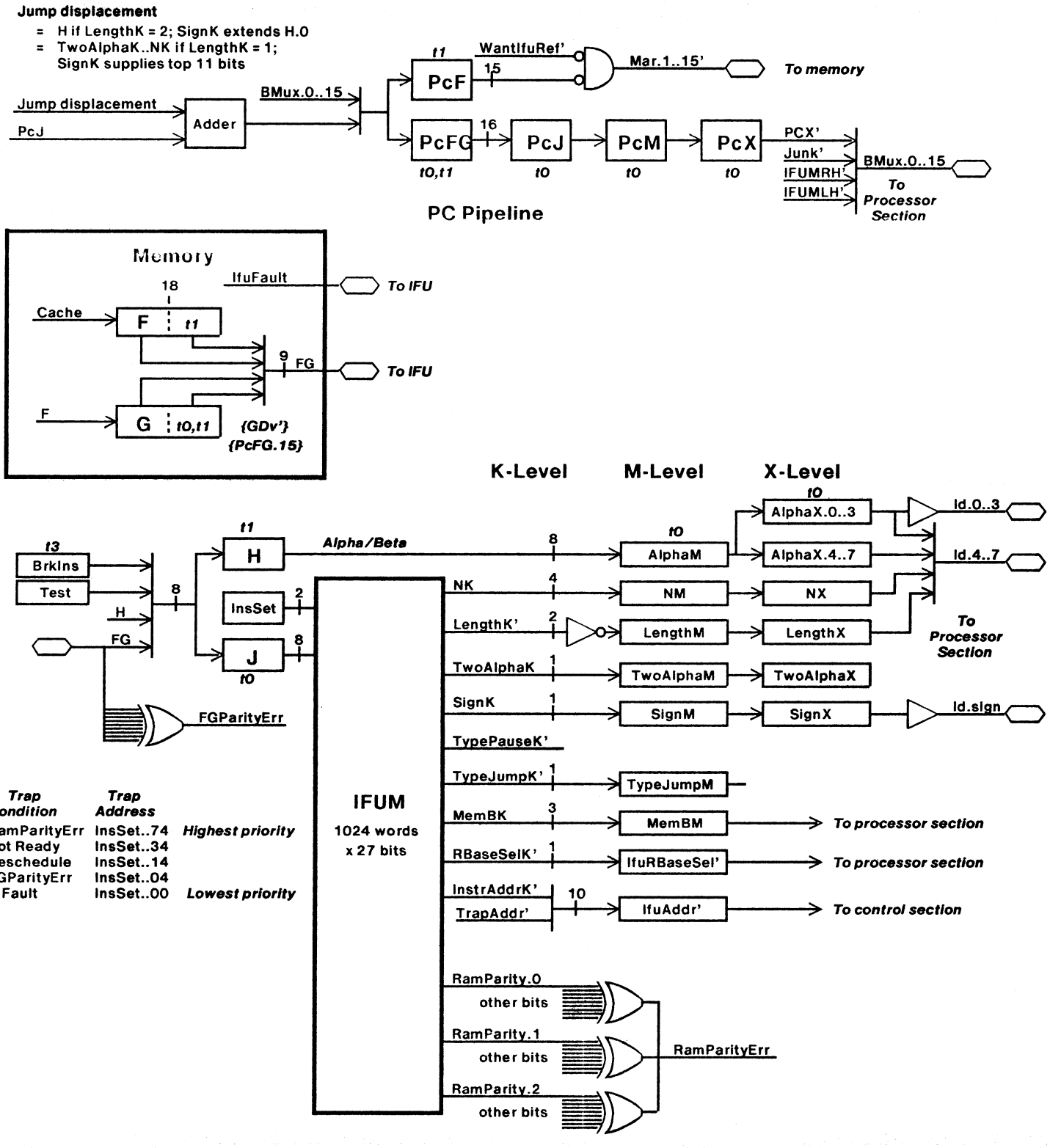


Figure 12

Instruction Fetch Unit Organization

D1IFU.sil

9/5/79

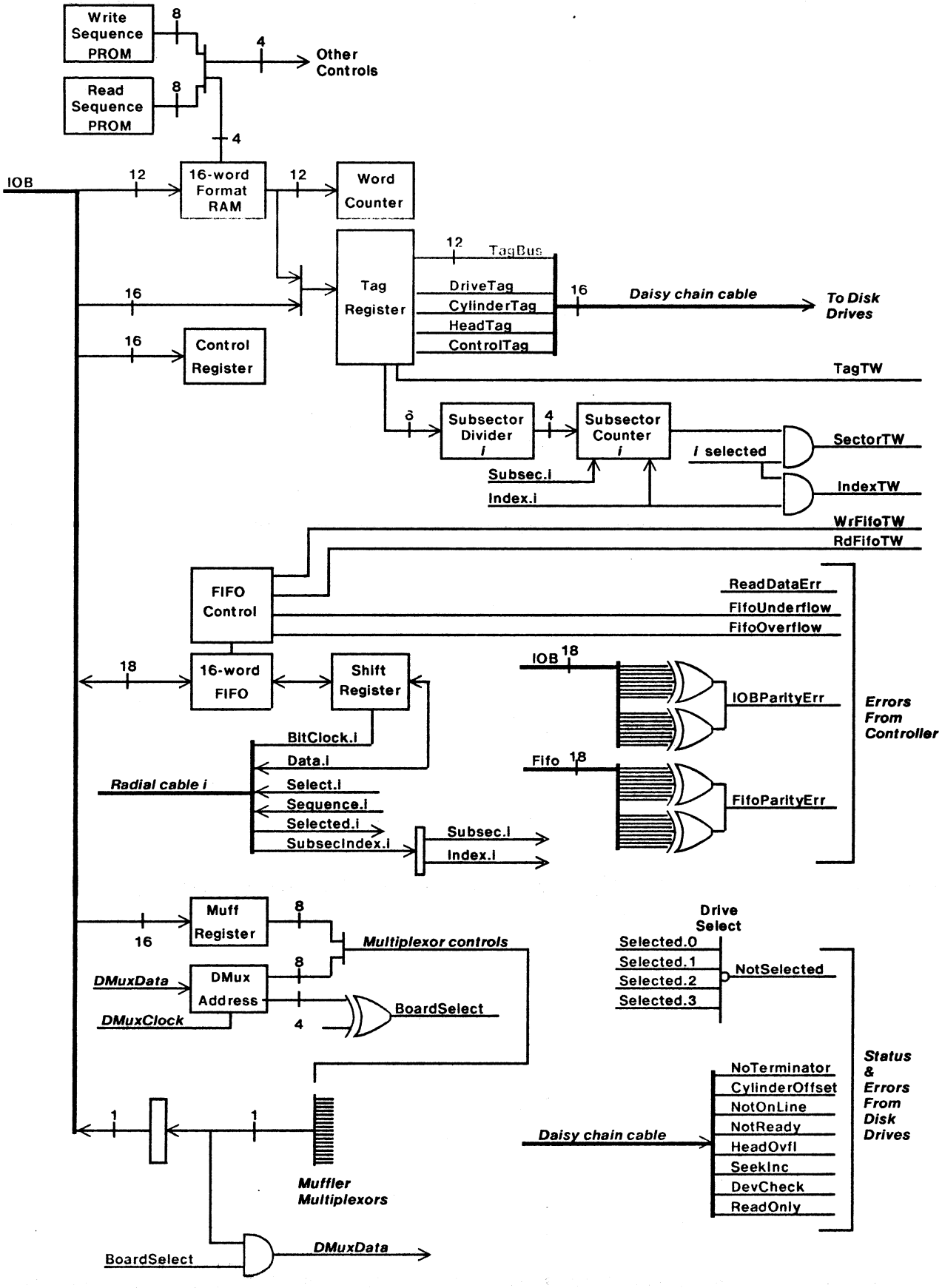
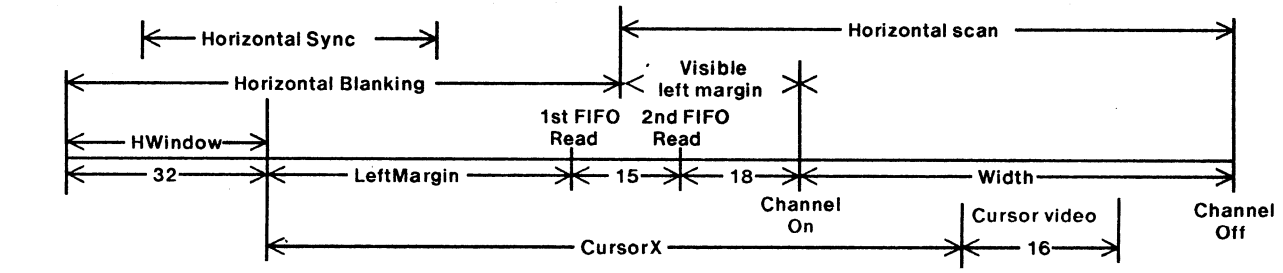


Figure 13
Disk Controller

D1Disk.sil
9/5/79



A or B channel timing (in pixel clocks, not to scale)

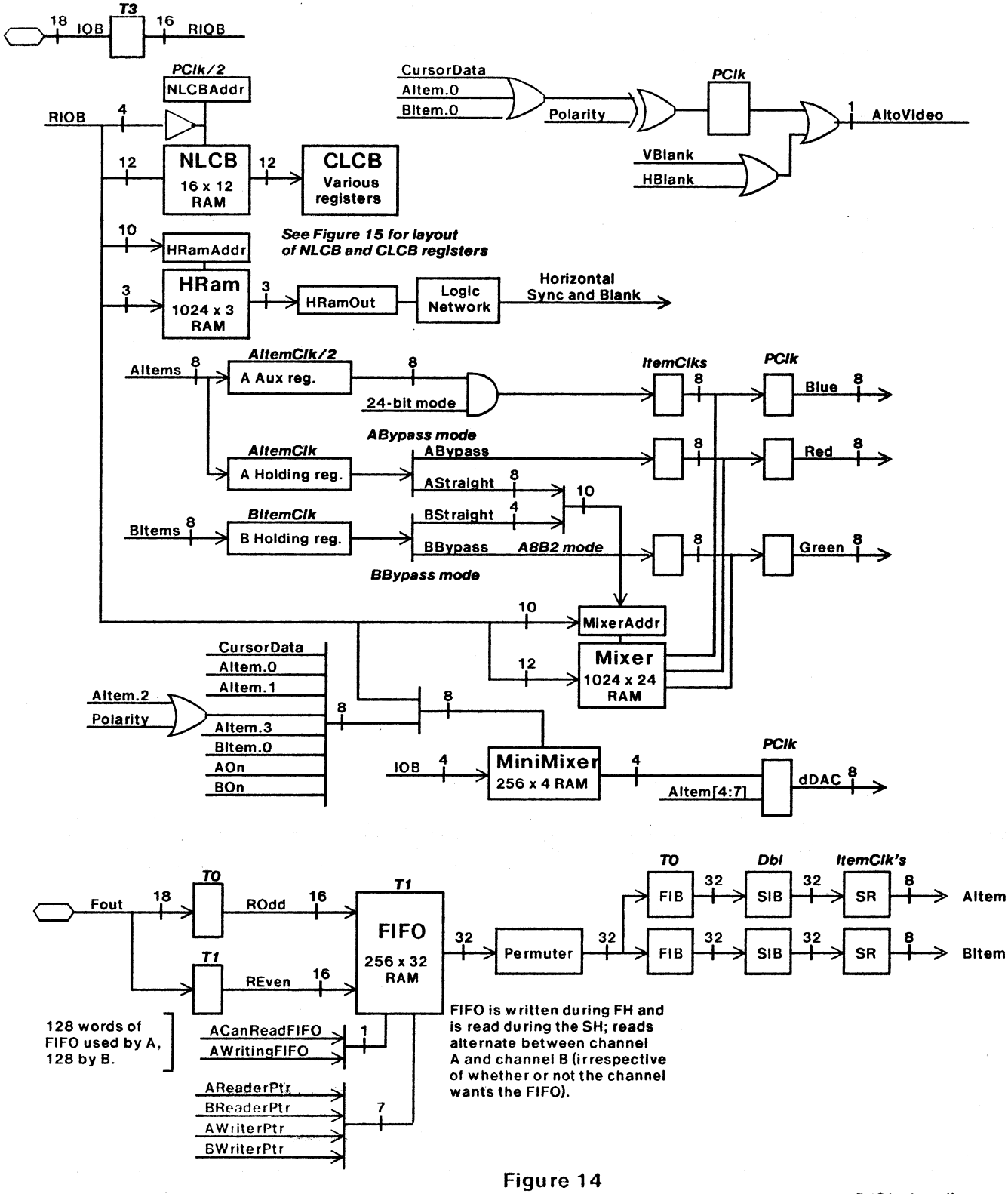
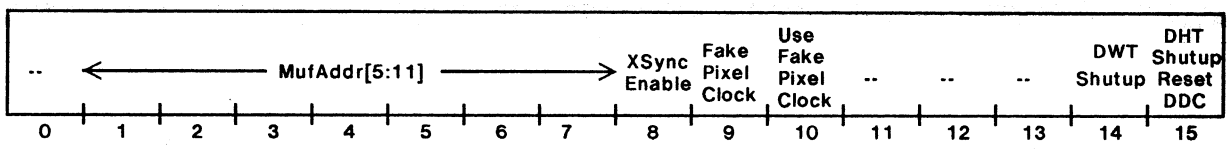


Figure 14
Display Controller

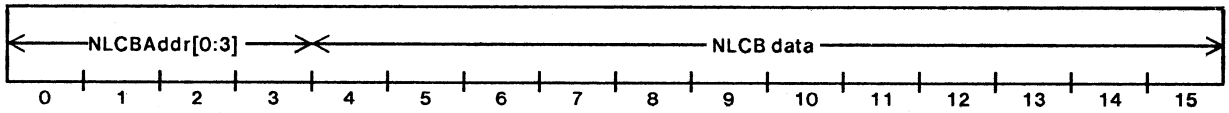
D1Display.sil
10/8/79

TIOA Output←B

377 Statics



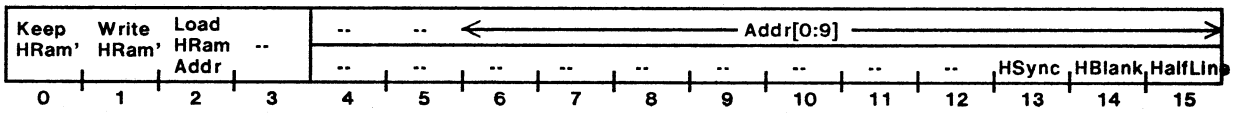
376 NLCB



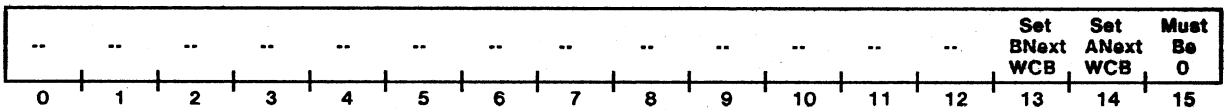
0	VCW	VBlank, VSync, OddField ← data[13:15]	10	--	
1	AMargin	AMargin ← data[4:15]	11	BMargin	BMargin ← data[4:15]
2	AWidth	AWidth ← data[4:15]	12	BWidth	BWidth ← data[4:15]
3	AFifoAddr	AFifoAddr ← data[8:15] (even)	13	BFifoAddr	BFifoAddr ← data[8:15] (even)
4	AScan	see below	14	BScan	see below
5	MixerMode	see below	15	CursorX	CursorX ← data[4:15]
6	--		16	CursorLo	Cursor[8:15] ← data[8:15]
7	--		17	CursorHi	Cursor[0:7] ← data[8:15]

AScan, BScan: Polarity, Resolution[0:1], Size8, Size4, Size2, Size1 ← data[9:15]
MixerMode: 24Bit, ABypass, BBypass, A8B2 ← data[12:15]

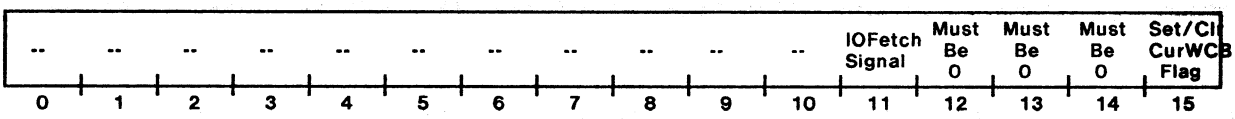
375 HRam



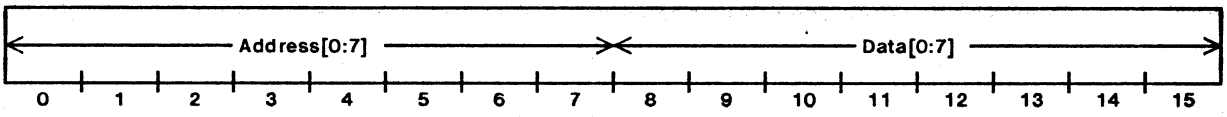
374 DHTFlag



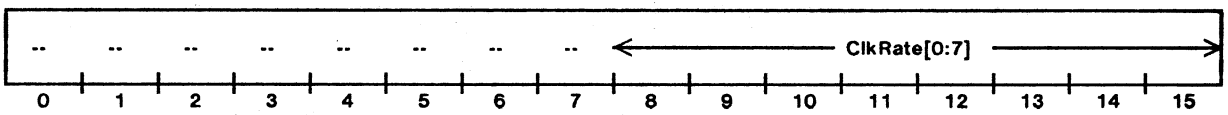
373 DWTFlag



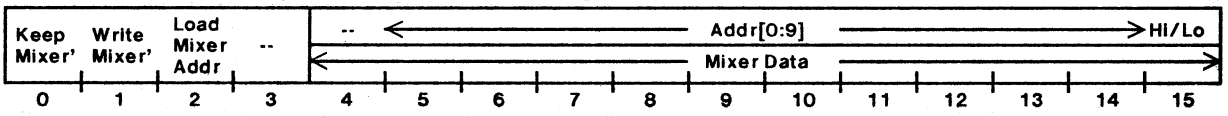
372 MiniMixer



367 PClock



366 Mixer



TIOA Pd←Input

370 DStatus

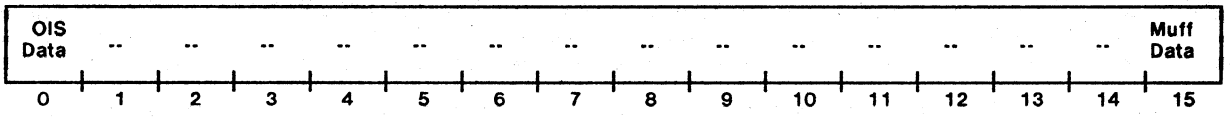
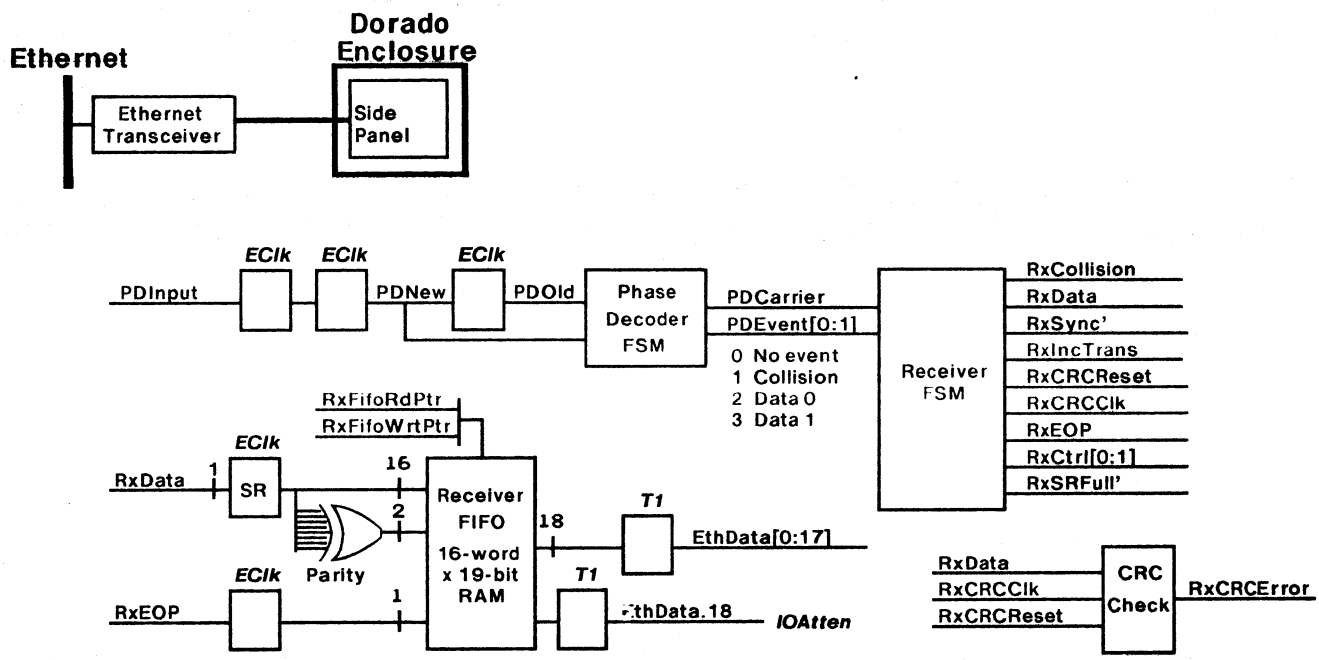
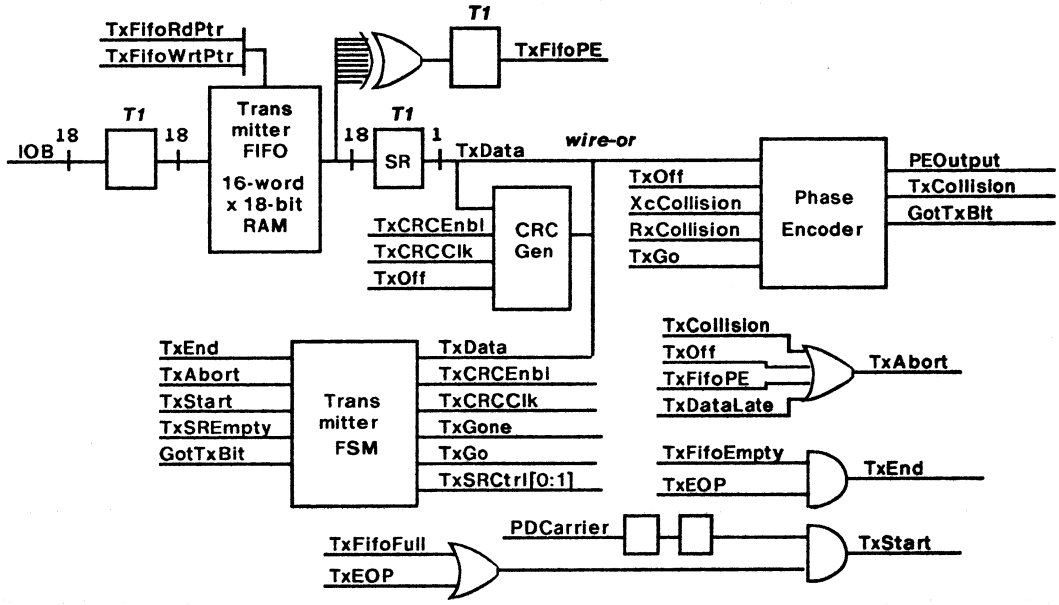


Figure 15
Display Controller IO Registers

D1DispReg.sil
10/8/79



Receiver



Transmitter

TIOA = 016

EthC Output←B

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Tx Cmd Enbl'	TxOn	TxEOP	Tx Cnt Dwn	Rx Cmd Enbl'	RxOn	RxBOP'	--	Test Cmd Enbl'	Loop Back	Single Step	No Wake ups	Test Clock	Test Coll'	Test Data	Report Colls

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

EthC d←Input

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
← Host Address →								RxOn	TxOn	Loop Back	TxColl	No Wake ups	Tx Data Late	Single Step	Tx Fifo PE

Host Address is set by backpanel jumpers

Figure 16
Ethernet Controller

BSEL Decodes

BSEL	Primary	External
0	Md	--
1	RM/STK	--
2	T	--
3	Q	Q← B
4	0,,FF	
5	377,,FF	
6	FF,,0	
7	FF,,377	

LC Decodes

LC	Meaning
0	No action
1	T← Pd
2	T← Md, RM/STK← Pd
3	T← Md
4	RM/STK← Md
5	T← Pd, RM/STK← Md
6	RM/STK← Pd
7	T← Pd, RM/STK← Pd

FF Decodes

000-17	A[12:15]← FF[4:7]
020	A← RM/STK
021	A← T
022	A← Md
023	A← Q
024	XorCarry
025	XorSavedCarry
026	Carry20
027	ModStkPBeforeW
030	--
031	ReadMap
032	Pd← Input
033	Pd← InputNoPE
034	RisId
035	TisId
036	Output← B
037	FlipMemBase
040-57	Replace RSTK by FF[4:7] for write
060-67	Branch conditions
070	BigBDispatch← B
071	BDispatch← B
072	Multiply
073	Q← B
074	--
075	TgetsMd
076	FreezeBC
077	Noop
100	PCF← B
101	IFUTest← B
102	IFUTick
103	RescheduleNow
104	--
105	MemBase← B[3:7]
106	RBase← B[12:15]
107	Pointers← B
110-17	--
120-21	--
122	CFlags← A'
123	BrLo← A
124	BrHi← A
125	LoadTestSyndrome
126	LoadMcr[A,B]
127	ProcSRN← B[12:15]
130	InsSetorEvent← B
131	EventCntB← B
132	Reschedule
133	NoReschedule
134	IFUMRH← B
135	IFUMLH← B
136	IFUReset
137	BrkIns← B
140	UseDMD
141	MidasStrobe← B
142	TaskingOff

ASEL Decodes (FF is ok)

ASEL	FF[0:1]	Meaning
0	0	PreFetch← RM/STK
	1	Map← RM/STK (emu/flt)
		IOFetch← RM (io)
	2	LongFetch← RM/STK
	3	Store← RM/STK
1	0	DummyRef← RM/STK
	1	Flush← RM/STK (emu/flt)
		IOStore← RM (io)
	2	IFetch← RM/STK
	3	Fetch← RM/STK
2	0	Store← Md
	1	Store← Id
	2	Store← Q
	3	Store← T
3	0	Fetch← Md
	1	Fetch← Id
	2	Fetch← Q
	3	Fetch← T
4	--	A← RM/STK
5	--	A← Id
6	--	A← T
7	--	Shift operation

ASEL Decodes (FF not ok)

ASEL	Meaning
0	Store← RM/STK
1	Fetch← RM/STK
2	Store← T
3	Fetch← T
4	A← RM/STK
5	A← Id
6	A← T
7	Shift operation

RSTK Decodes for STK Operations

RSTK[0]: 0 = No ovfl/undfl check; 1 = Ovfl/undfl check

RSTK[1:3]	Meaning
0	No StkP change
1	StkP← StkP + 1
2	StkP← StkP + 2
3	StkP← StkP + 3
4	StkP← StkP - 4
5	StkP← StkP - 3
6	StkP← StkP - 2
7	StkP← StkP - 1

ALUFM Control Values

Logical Value	Addr	Meaning
1	16	NOT A
3		NOT A OR NOT B
5		NOT A OR B
7		A1 (all ones)
11		NOT A AND NOT B
13	14	NOT B
15		A XNOR B, A EQV B, A = B
17		A OR NOT B
21		NOT A AND B
23	10	A XOR B, A # B
25	0	B
27	7	A OR B
31	11	A0 (all zeroes)
33	15	A AND NOT B
35	6	A AND B
37		A

Arithmetic (no carry) Value	Addr	Meaning
0	1	A
6		2*A
14	2	A + B
22	5	A - B - 1
36	13	A - 1

Arithmetic (with carry) Value	Addr	Meaning
200	12	A + 1
206		2*A + 1
214	3	A + B + 1
222	4	A - B
236		A

ALUFM addresses for operations in standard system microcode

ALUF Shift Decodes

ALUF[0:2]	Meaning
0	ShiftNoMask
1	ShiftLMask
2	ShiftRMask
3	ShiftBothMasks
4	ShMdNoMask
5	ShMdLMask
6	ShMdRMask
7	ShMdBothMasks

Derivation of Shift Controls

Field:	SHA	SHB	Count	RMask	LMask	
ShC bits:	2	3	4:7	8:11	12:15	
RF← A	A[2]	A[3]	P + S + 1	undefined	15-S	Shift controls come from Shc when BSEL[0] = 0 in the microinstruction that shifts
WF← A	A[2]	A[3]	16-P-S-1	16-P-S-1	P	
ShC← B	B[2]	B[3]	B[4:7]	B[8:11]	B[12:15]	
	BSEL.1	BSEL.2	FF[4:7]	FF[4:7]	FF[0:3]	Shift controls come from FF when BSEL[0] = 1, and the source for B is changed to Q

P = A[8:11] = number of bits to the left of the field
S = A[12:15] = number of bits in the field - 1

143	TaskingOn
144	StkP← B[8:15]
145	RestoreStkP
146	Cnt← B
147	Link← B
150	Q lsh 1
151	Q rsh 1
152	TIOA[0:7]← B[0:7]
153	--
154	Hold&TaskSim← B
155	WF← A
156	RF← A
157	ShC← A
160	B← FaultInfo'
161	B← Pipe0 (VaHi)
162	B← Pipe1 (VaLo)
163	B← Pipe2'
164	B← Pipe3' (Map')
165	B← Pipe4' (Errors')
166	B← Config'
167	B← Pipe5
170	B← PCX'
171	B← EventCntA'
172	B← IFUMRH'
173	B← IFUMLH'
174	B← EventCntB'
175	B← DBuf
176	B← RWCPReg
177	B← Link
200-17	RBase← FF[4:7]
220-37	Replace RBase by FF[4:7] for write
240-47	TIOA[5:7]← FF[5:7]
250-53	MemBaseX← FF[6:7]
254-57	MemBX← FF[6:7]
260-61	--
262	Pd← ALUFMRW
263	Pd← ALUFMEM
264	Pd← Cnt
265	Pd← Pointers
266	Pd← TIOA&StkP
267	Pd← ShC
270	Pd← ALU rsh 1
271	Pd← ALU rcy 1
272	Pd← ALU brsh 1
273	Pd← ALU arsh 1
274	Pd← ALU lsh 1
275	Pd← ALU lcy 1
276	Divide
277	CDivide
300-37	MemBase← FF[3:7]
340-57	Cnt← FF[4:7]
360-67	Wakeup[FF[4:7]]

Figure 17
Programmers' Crib Sheet

D1CribSheet.sil
9/26/79