Dorado Hardware Manual

by E.R. Fiala

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

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

14 September 1981

The document describes the architecture and hardware design of the Dorado computer at a level appropriate for programming. At the date of this printing, approximately 22 systems have been released to users.

This release incorporates a major revision of the Display Controller chapter, medium revisions to the Disk Controller and Instruction Fetch Unit chapters, and minor revisions elsewhere.

Revision history:

14 February 1979	First complete manual exclusive of io controller chapters.
8 October 1979	Chapters on io controllers added; major revisions.
14 September 1981	Major revision to the Display Controller chapter, medium revision to Instruction Fetch Unit and Disk chapters, minor revisions elsewhere.

XEROX

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

This document is for internal Xerox use only.

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 Mesa 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 60 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 prototypes and multiwire or PC in 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; opcodes may independently be specified as one, two, or three bytes long.

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^{22} 16-bit words when 64k x 1 RAMs.

The processor initiates data transfers between main memory and fast input/output devices. 16 16-bit words are then transmitted without disturbing the processor data paths in about 1.68 ms (28 cycles). New references can be initiated every 8 cycles, so total bandwidth of the memory, 533 mHz, is available for devices with enough buffering.

Overview

Experience suggests that programmers will gradually develop a mental model something like Figure 1; until this mental model is well established, it is probably desirable to

Read the following with Figure 1 in view.

Dorado has Processor, Control, Memory, IFU, and IO controller sections.

Io controllers are independent of each other and of the other sections you will have to understand a particular io controller iff you are going to write microcode that controls it.

The memory and IFU are "slaves" to the processor/control section. In most situations, their external interface is simple relative to internal details of operation, and effective programming is usually possible without detailed understanding.

However, programmers will have to understand the processor thoroughly because the different parts of the processor are controlled directly by instruction fields, and most of the processor will be used, even in a small program.

Programmers must also understand most of the control section, although fairly simple assembly language contstructs are transformed into the complicated branch encodings needed by Dorado, so detailed understanding of Dorado branching is not required.

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.

In this manual, "instruction" refers to a microinstruction in the control store, as opposed to an opcode in the higher level language interpreted by a microprogram. The JCN field in an instruction encodes a variety of jumps, calls, conditional jumps and calls, instruction dispatches and returns for the current task.

Registers, Memories, and Data Paths

Tables 1, 2, and 3 describe memories, registers, and data paths in Dorado; these are diagrammed in Figure 1. The first two tables below focus on a particular register or memory and tell how it is used and where it connects; the third table focuses on particular data paths and shows how they connect various parts of the machine.

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_A and BrHi_A 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 or 64k-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 * = one of these for each task; i.e., "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.

Table 3: Data Paths

Path	Comments
A	The 16-bit high-true A bus (called "alua" in hardware drawings) may be driven from T, RM, STK, Q, Id, Md, a small constant between 0 and 17_8, or the shifter. It is also possible to 'or' the low-true shifter output with one of the other A sources. The A bus is totally inside the processor section, not connected to any other sections of Dorado, and it is one of the two Alu inputs. The RF_A and WF_A functions, which load ShC for subsequent shift operations, receive data from A.
Mar	The 16-bit Mar bus transmits the displacement for a memory reference from the processor or IFU section to the memory section. The CFlags register, some bits of the Mcr register, and the BR memory in the memory section are also loaded from Mar. The processor drives Mar only when it is starting a reference or executing one of the functions between 120_8 and 127_8 (i.e., CFlags_A' and LoadMcr[A,B] are in this group of functions); during other instructions, the IFU may use Mar to initiate instruction-fetches. Mar is driven low-true; when driven by the processor, it receives the same data as are driven onto A (but the shifter cannot drive Mar).
B	The 16-bit B bus consists of one data path inside the processor section (called "alub" in hardware drawings) and another on the backplane (called "Bmux" in hardware drawings); the IOB bus is driven from Alub on Output operations, when it also is an extension of B. Alub and Bmux may be directly driven high-true from registers inside the processor; alternatively, Bmux may be driven low-true from other sections, in which case the processor receives the data onto alub through inverters (so the data appears high-true on alub). The BSEL field in an instruction can specify that either T, RM/STK, Q, or Md sources B; other sources and destinations loaded from B are specified in the FF field; BSEL and FF are used in combination to specify that a literal 8-bit constant (in either the left or right byte of the word with 0's or 1's in the other byte) sources B. Alub is one of the two Alu inputs. The processor computes odd byte parity on alub; Bmux and IOB destinations may store or check the parity computed by the processor.
Pd	The Pd path (" Processor data") receives data from an 8-input multiplexor whose inputs are the Alu output, possibly shifted left or right one bit on Alu shift functions or masked on a shifter operation, io device input data, and the infrequently read registers in the processor section. Pd may be written into the T register or the RM or STK memories.
Id	The Id path (" IFU data") is used to send arguments from the IFU to the processor for interpretation. It can be routed onto A using ASEL (A_Id, Fetch_Id, Store_Id, or IFetch_RM/STK); alternatively, the TIsId or RIsId functions can be used to replace data from T or from RM/STK by IFU data these functions provide a roundabout method of getting Id onto B.
Md	The Md path (" Memory data") moves data from the cache in the memory section into the processor. The processor latches Md and can route it onto A or B, load it into T and RM/STK, or use it in a shift-and-mask operation.
IOA	The IOA bus (" Input-output address") is driven from the TIOA register; it specifies the io device affected by a Pd_Input or Output_B function.
IOB	The IOB bus (" Input-output bus") is driven from alub on an Output_B function or received on Pd by a Pd_Input function; it transmits data to or from an io device.
Fout	(" Fast output bus") transmits data from the error corrector to a fast output device.
Fin	(" Fast input bus") transmits data from a fast input device (Presently, there are no fast input devices) to the syndrome generator.
Sout	(" Storage output bus") transmits data from the syndrome generator to storage.
Sin	(" Storage input bus") transmits data from storage to the error corrector.

Timing

The terminology used in discussing timing is as follows:

clock	The 30 ns (nominal) atomic time period of the machine. Clock period can be controlled by the baseboard microcomputer or through the manifold system as discussed in the "Dorado Debugging Interface" document.[1]
cycle	The duration of instructions two clocks or 60 ns except for instructions that read/write IM or TPC.
t_0	The instant at which MIR (Micro Instruction 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 4 summarizes the time at which loading takes place and some other information.

[1] We actually operate with a clock speed of 32 ns, slower than the 30 ns nominal period, and production machines typically become unreliable at about a 29 ns clock period.

Table 4: Load Timing

Register/ Memory	Task Specific	Load Time	Data Source	Load Control	Comment
MIR*	no	t0	IM	JCN	Holds current instruction
CIA	no	t0	TNIA,BNPC	JCN	Holds current instruction 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 5: 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 16_8 or 17_8.

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 Pd 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, and IFU 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, RBase and StkP Registers

RM (" Register Memory," sometimes called "R") is the memory most easily available to microprograms; it stores 256 words x 16 data bits with odd parity on each byte of data. RM 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 Section") test the sign bit (R<0) and low bit (R Odd). RM 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.

The STK memory (sometimes called "stack") is accessible only to the emulator (task 0). Since the emulator cannot block, the instruction bit interpreted as BLOCK for io tasks is instead interpreted as StackSelect; when StackSelect is 1, RM is disabled and STK used instead. Like RM, STK stores 256 words x 16 data bits with odd parity on each byte of data. 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 6: 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 can be loaded from B[8:15] using the StkP_B function; however, this is *illegal in conjunction with a STK read or write in the same instruction* (e.g., T_Stack, StkP_T leaves StkP unchanged).

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; *RestoreStkP is illegal in conjunction with a STK read or write in the same instruction*.

> RestoreStkP is useful only 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.
>
> The opcode-restart method effectively prevents use of the IFU entry vector scheme discussed in "IFU Section," degrading performance perhaps 2%, so it is desirable to continue from rather than restart from faults. Also, complicated opcodes may require special-case code in the fault handler before opcode restart is possible, so continuing from the instruction that faulted is likely to be simpler overall.

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.

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.). Note: *SubTask does not affect the write address for these functions.*

Note that there is no way to read RM and write STK in one instruction.

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 must save and restore 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 must save and restore 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 one of the "internal" processor sources for B, as shown in the "Primary" column in the table below (Note that although Md originates in the memory section, it is latched by the processor and appears as an internal B source.). However, the FF field can be used to substitute some other source external to the processor there are many "external" sources in the control, IFU, and memory sections, and the codes for these are given in Table 11. When an external source is specified, then BSEL instead encodes the destination for B, as shown in the "External" column of the table below.

The sources selected by BSEL are:

Table 7: BSEL Decodes

BSEL	Primary	With External Source
0	Md	
1	RM/STK	
2	T	
3	Q	Q_B *
4	0,,FF	Inapplicable because FF is not available to encode an external source
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 external B sources are impossible. 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 arithmetic 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 reads 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 t_1+6 ns and the processor section is off by t_1+7 ns, but the memory and control sections are not on/off until t_1+16 ns; hence, a slow Bmux source in the previous instruction prevents Bmux from stabilizing until t_1+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 8a: 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 (io 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 8b: 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 9: 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.

**Emulator task 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 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. Note that the ALU output of this instruction are used (not the previous one) and that ALUcarry is undefined on a logical ALU operation. 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 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 (A+1, A-1, 2A, 2A+1, etc.) 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 10: 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 11a: FF Decodes (FA = 0)

FB	FC	Function
* The AMux is not disabled when A_xx decodes below are used while ASEL selects a shift.		
0-1		A[12:15] _ FF[4:7]
2	0	A _ RM/STK
2	1	A _ T
2	2	A _ Md
2	3	A _ Q
2	4	XorCarry (complements ALUFM carry bit) see the "ALUF, ALU Operations" section
2	5	XorSavedCarry see the "ALUF, ALU Operations" section
2	6	Carry20 (carry-in to bit 11 of ALU = 1) see the "ALUF, ALU Operations" section
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 parity error)
3	3	Pd _ InputNoPE (no check for IOB parity error)
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' branch conditions for 1 cycle)
7	7	Reserved as a no-op

Table 11b: 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
0	2	IFUTick
0	3	RescheduleNow (doesn't set Reschedule branch condition)
0	4	AckJunkTW_B. B[15]=1 shuts off junk task wakeups, =0 enables them; B[0:14] ignored
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	Unused

Table 11c: FF Decodes (FA = 1)

FB	FC	Action
*The following 8 FF decodes drive Mar from A.		
2	0-1	Unused
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 into the 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 (overrules loading of Link by Call or Return in same instruction)
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 "Other IO and Event Counters")
7	2	B _ IFUMRH' (low part of IFUM)
7	3	B _ IFUMLH' (high part of IFUM)
7	4	B _ EventCntB' (see "Other IO and Event Counters")
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 11d: 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 (If Cnt=0& 1 in same instruction, unmodified value is read)
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; if the instruction modifies StkP concurrently, the MODIFIED value is read)
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 11e: 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 16_8 or ALUFM 17_8 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 16_8 contain the control for the "NOT A" ALU operation normally desired, while ALUFM 17_8 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 corresponding to 1's in the mask 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; they also load ShC[2:3] from A[2:3]. ShC_B allows an arbitrary value to be placed in ShC (used by BitBlt).

Microcode for the Mesa RF (Read Field) and WF (Write Field) opcode is shown as an example of the use of the shifter. In these examples, a and b are the two operand bytes for the opcode, as discussed in "Instruction Fetch Unit." RF and WF both take a pointer from the top of the stack and add a to it as a displacement. RF fetches the word, and pushes the field specified by b 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. a replaces BR[MemBase] (MDS);
                                        *this value is then added to Stack to compute the
                                        *address for the pointer.
        Stack_Md, RF_Id;                *IFU supplies b, the field descriptor
        IFUJump[0], Stack_ShiftLMask;   *Right-justify & mask the field, IFU to next instruction

WF:     T_(IFetch_Stack&-1)+T, TisId;   *Start fetch of word containing field
        WF_Id, RTemp _T;                *IFU supplies b, the field descriptor
        T_ShMdBothMasks[Stack&-1];
        IFUJump[0], Store_RTemp, DBuf_T;
```

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 12: 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 16_8 or 17_8

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.

Hold and Task Simulator

The hold and task simulators are provided for hardware checkout (programmers skip this section).

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 and since its clocks are not disabled by HOLD, 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 16 Wakeup[task] FF decodes allow any task to be woken, just as though a hardware device had activated its wakeup line. A minimum of two cycles elapses after the instruction containing Wakeup before the task executes its first instruction. The task responding to a Wakeup must not block sooner than the second instruction, or it will get reawakened.

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 Wakeup[task] function must be executed with tasking off, if it is possible that the specified task might be waking up for some other reason (e.g., due to a wakeup request from an external device, or due to a wakeup issued by yet another task). Otherwise, the control section may get horribly confused, and the machine will hang in the same task forever.

An acceptable sequence is:

```
TaskingOff;
Wakeup[task];
TaskingOn;
```

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. TaskingOff is "atomic"; an instruction containing TaskingOff will be held if a task switch is pending; the next instrcution will be executed in sequence without any intervening task switches. TaskingOn is not immediately effective; at least two more instructions will be executed by the same task before task switching can occur.

It would be a programming error for a task to block with tasking off, but if it did, the 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 a task switch occurs, the program counter for the highest priority competing task (BNPC or "Best Next PC") addresses IM.

A 16k-word microstore is viewed as consisting of four 4k-word quadrants; each 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 of the current quadrant;

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. In addition, 5 of the 256 values of JCN are unused and 1 is a duplicate (See Figure 6 for the 5 unused decodes; the replicated decode is the Global call on the Local page.). We have variant JCN decodings that correct these problems, but they were not ready when the design was frozen.

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 ns 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 13: 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 (RM or STK, whichever is selected, *not* overruled by RIsId)
5	65	R Odd (RM or STK, whichever is selected, *not* overruled by RIsId)
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. Since the CPReg register clock is asynchronous to the Dorado clock system, a Dorado microprogram that reads CPReg (e.g., to receive information from the baseboard) must use some synchronization method to ensure that CPReg is stable during the cycle in which it is read.

Note: it is illegal to use an ALU branch condition in the instruction after Pd_RWCPReg, if CPReg might have been loaded during the cycle in which it is read this might result in an unstable IM address being presented to the control store.

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 value of Q[14] is captured in a flipflop at t_2 of the instruction containing the Multiply function and is OR'ed into TNIA[14] during the next instruction for the same task.)

Example:

```
BDispatch_T;
Branch[300];      *branches to 300 OR T[13:15]
```

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 Link is task-specific, this works correctly across task switching. The Q-bit is only loaded during a multiply, and tasks other than the emulator are not allowed to use the multiply function.

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 14: 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.*

It has been reported that IM_Md doesn't work because _Md causes hold at unexpected times.

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_B and LdTPC_B. 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 one-wire 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_13S;
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 (the task that will control the processor in the next cycle) 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.

The storage pipeline allows new requests every 8 cycles, but requires 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.
>
> Note that although the VA path is 28 bits wide, limitations imposed by cache and map geometry limit usable virtual memory to only 2^{22} or 2^{24} words in most configurations, as discussed in "The Map" section later.

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 as discussed in the "Processor Section" chapter) 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[MemBase].
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
RMap_Rtemp;                 *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. Note: it is necessary to hold until any preceding private pipe entry fetch or Store_ has finished by issuing _Md reasons for this are discussed in "The Pipe" section.

*DummyRef_*

DummyRef_ writes VA into the pipe entry for the reference without initiating cache, map, or storage activity. This is provided for reading base registers and so diagnostic microcode can exercise VA paths of the memory system without disturbing cache or memory. Note: it is necessary to hold until any preceding private pipe entry fetch or Store_ has finished by issuing _Md reasons for this are discussed in "The Pipe" section.

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 or coincident with 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_.

> *Hardware note:* 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. Asrn is the internal register that contains the pipe address for storage references.

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

(1) 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.

(2) 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.

(3) 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.

(4) 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*.

(5) 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.

(6) 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 15: 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 16: 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 RAM 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 reveals something other than *Vacant*, 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. For reasons discussed in "The Pipe" section later, Map_ should not be issued if a preceding fetch or Store_ might be in progress; normally issue a _Md to hold until preceding references cannot fault.

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. Note that if the real address space is less than 2^{24} words (2^{16} pages), high order RP bits are ignored during references though they are kept in the map and appear in the pipe.

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 138 ms with 64 rows or 553 ms with 256 rows in the cache at 60 ns/cycle.

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.
>
> GetFlags[vp] RETURNS [rp,flags].

These are defined as *indivisible operations* and are implemented trivially on a machine with no cache (e.g., Dolphin). 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.

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, difficult to implement these primitives as indivisible operations because almost any change to 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 present Mesa code flushes once with tasking on and then again with tasking off.

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.
>
> Disable tasking.
>
> Flush all cache entries for the page again.

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 (possibly followed by PreFetch to set map[v].Ref true).

Note: These primitives do not support the complete cache flush 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. It has been observed that issuing Map_ without holding for a previous Store_ to finish will result in infinite DBufBusy (i.e., infinite Hold), so do not fail to issue _Md before or concurrent with Map_ or RMap_.

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:

CacheRef	a fetch or Store_
Store'	Store_'
IFURef	IFU fetches
RefType	distinguishes read, write, Map_, and other 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 17: 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 *Store'* bit in Pipe5 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 RAMs 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 ms. Every MOS RAM on every storage board participates in every refresh reference, and one row of data is refreshed each time. This means that 64 (4k RAMs), 128 (16k RAMs), or 256 (64k RAMs) refresh references are required to refresh all data (So the refresh period is 2 or 4 ms the specification on both 16k and 64k RAMs is a 2 ms refresh period at the maximum operating temperature (85^o C). The dominant leakage term is exponential in temperature so the refresh period can be doubled each 5^o C drop in operating temperature. Because the specification is conservative and because we have no intention of operating anywhere near 85^o C, a 4 ms refresh period should be adequate.

The time for each refresh reference is 8 cycles (13 cycles with 4k-bit RAMs), same as normal references. Refresh hardware competes for storage access with the cache data section and fast io references. During the first 8 ms of a 16 ms period, refresh defers to normal references; during the last 8 ms, 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_ function 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 is needed to test these (the IOTest board).

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.