# Instruction Fetch Unit

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

*Read this chapter with Figure 12 in front of you.*

## Overview of Operation

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

> One method of dealing with operations longer than 3 bytes is to encode them in IFUM as 1-byte jumps to the next operation. This gives up the possibility of referencing N, a, or b with _Id but avoids having to restart the IFU. The processor then must compute the proper place in the instruction stream and reference a, b, g, etc. without help from the IFU.

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

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

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

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

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

**Table 18: IFUM Fields**

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

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

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

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

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

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

**Table 19: Operand Sequence for _Id**

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

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

The normal opcode references all of its N, a, and b operands; however, except on three-byte opcodes, the IFU hardware does not *require* that these operands be referenced the processor could exit to the next opcode without reading all the operands, if that was desirable for some reason. However, for opcodes of length 3, the processor must consume the a byte with _Id (both a[0:3] and a[4:7] if Packeda=1) *before* going to the next opcode with an IFUJump it does *not* suffice to consume the last a byte with _Id concurrent with IFUJump. An opcode must never do more than 7 _Id's for reasons that will be discussed later.

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

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

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

A jump with *Length*=3 is illegal.

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

and H and refills these pipe levels with bytes along the jump path.

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

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

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

```
T_(Id) (PCX') 1;          *Id = Length for type jump
PCF_T;
Noop;
IFUJump[0];
```

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

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

## Table 20: IFU FF Decodes

| Name | Action |
|---|---|
| IFUReset | Halt and clear the IFU pipeline and clear errors, testing features, and BrkPending (i.e., BrkIns); Reschedule condition and instruction set are not cleared. |
| B_IFUMLH' | Read the high-order IFUM word, InsSet, and IdCnt onto B (low-true) as follows: |

| *Field* | *B bits* | |
|---|---|---|
| IdCnt | 0:2 | Count of _Id's since start of opcode |
| InsSet | 3:4 | Instruction set number |
| Packeda | 5 | Packed a |
| IFaddr' | 6:15 | Starting address |

| | |
|---|---|
| IFUMLH_B | Load the high-order IFUM word from B ($t_1$ to $t_3$), where the Packeda and IFaddr fields are in the same form as B_IFUMLH'. Must have at least one intervening instruction after a preceding BrkIns_ or InsSetorEvent_. |
| IFUMRH_B | Load the low-order IFUM word from B ($t_1$ to $t_3$) in the format given below; must have at least one intervening instruction after a preceding BrkIns_ or InsSetorEvent_: |

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

| | |
|---|---|
| B_IFUMRH' | Read IFUM fields in the same format as IFUMRH_B (inverted). |
| PCF_B | Load PCF at $t_3$, clear and restart the pipeline. |
| B_PCX' | Read PC for the currently executing opcode (inverted). |
| BrkIns_B | Load BrkIns from B[0:7] at $t_3$, and set BrkPending (ill-defined unless the IFU has been reset). BrkIns replaces the next opcode loaded into J; then BrkPending is cleared. BrkIns also addresses IFUM on IFUMLH/RH_ and B_IFUMLH'/RH'. |
| InsSetOrEvent_B | If B[0]=1, then B[6:7] are loaded into the InsSet register at $t_3$; if B[0]=0, then B[4:15] control event counters as discussed in the "Other IO and Event Counters" chapter. A following PCF_B starts the IFU interpreting using the new instruction set. Illegal except when the IFU is paused or reset or when PCF_ will be done before the next IFUJump. |

**Table 20: IFU FF Decodes (continued)**

| Name | Action |
|---|---|
| Reschedule | Cause a reschedule trap on the *second* or third "successful" IFUJump. "Successful" means that an IFUJump is not trapped for some other reason such as not-ready. The second IFUJump will be trapped if it does not occur in the instruction immediately after the first successful IFUJump; otherwise, the third successful IFUJump will be trapped. The trap instruction is executed as though it were the first instruction of the rescheduled opcode, and _Id and IFUJump will work as though that opcode were in progress.<br>Also set the Reschedule branch condition (emulator only) to true. |
| RescheduleNow | RescheduleNow is guaranteed to trap the *next* successful IFUJump, so long as the next IFUJump appears in the second cycle after RescheduleNow, or later. The Reschedule branch condition is not affected. |
| NoReschedule | Turn off the Reschedule trap and branch condition. |
| IFUTest_B | Load the test-control register from B (load with 0 or do IFUReset when not testing) as follows: |

| *Field* | *B bits* | |
|---|---|---|
| TestFG | 0:7 | Substituted for cache data |
| TestFGParity | 8 | Substituted for cache parity bit |
| TestFault | 9 | Substituted for memory fault signal |
| TestMemAck | 10 | Substituted for memory MemAck signal |
| TestMakeF_D | 11 | Substituted for memory MakeF_D signal |
| TestFH’ | 12 | enable FHCP and $t_1$ when IFUTick executed |
| TestSH’ | 13 | enable SHCP and $t_2$ when IFUTick executed |
| TestEn | 14 | test enable |

| | |
|---|---|
| IFUTick | Tick the IFU’s clock once according to TestFH and TestSH in the IFUTest register. |

### The IFUJump Entry Vector

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

An opcode leaves its results in *one of several convenient forms* agreed to by convention, then chooses an entry instruction in its successor with IFUJump[n], where n =0 to 3. Every opcode in the instruction set must have an entry vector of the same length. Careful choice of forms may reduce execution time by one cycle for some opcodes without increasing execution time for successor opcodes.

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

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

something more explicit, if an appropriate convention is followed by all opcodes.

The following example shows how an instruction set with four opcodes (Push, Add, Store, and JNZ) is implemented using a four-long entry vector. The opcodes in this example deal with the stack like Mesa opcodes do, and the first three entry conventions are, in fact, ones which might be used by the current Mesa emulator.

```
%Entry
    0:  Stk[StkP] holds top-of-stack (if any garbage if stack empty), T holds garbage
    1:  T and Stk[StkP-1] hold previous top of stack (garbage if stack empty),
        Stk[StkP] garbage, Md holds top-of-stack.
    2:  T and Stk[StkP+1] hold top-of-stack,
        Stk[StkP] holds previous top of stack (garbage if stack empty).
    3:  Results in same form as entry 2, but restart IFU at NewPC = (Id) (PCX') 1
Note that Stack&+1 references must not check for underflow when the stack may legitimately be
empty.
%

*Push the memory location pointed to by N.
Push:   Fetch_Id, T_StackNoUFL&+1, IFUJump[1];
        Fetch_Id, T_StackNoUFL&+1_Md, IFUJump[1];
        Fetch_Id, StkP+2, IFUJump[1];
        T_(Id) (PCX') 1, StkP+1, Return;

*Replace the top two stack entries by their sum.
Add:    T_Stack& 1, Branch[.+2];
        Stack_Md;
        T_Stack& 1_T+(Stack& 1), IFUJump[2];
        T_(Id) (PCX') 1, StkP+1, Return;

*Store the top-of-stack into the memory location pointed to by N and pop the stack.
Store:  Store_Id, DBuf_Stack& 1, IFUJump[0];
        Stack_Md, Branch[Storex];
        Store_Id, DBuf_T, IFUJump[0];
        T_(Id) (PCX') 1, StkP+1, Return;
Storex: Store_Id, DBuf_Stack& 2, IFUJump[2];

*Pop the stack and branch if the top-of-stack was zero, else fall through
*This opcode is of type jump.
JNZ:    Pd_Stack& 1, Branch[ZTest];
        Pd_Md, StkP 1, Branch[ZTest];
        Pd_T, Branch[ZTest];
        T_(Id) (PCX') 1, StkP+1, Return;
ZTest:  T_Stack& 1, IFUJump[2,ALU#0];
*Return here when the jump doesn't take.
        T_Stack& 1, PCF_T;
        IFUJump[2];
```

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

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

respectively, compared to 1, 1 or 2, 1 or 2, and 2 or 3 cycles for the four-entry scheme above.

Since Mesa requires about 120 IFU entries for its 256 opcodes, the cost of the second entry in the vector is between 0 and 120 locations, and 120 locations each for the third and fourth entries. Since Mesa is implemented by about 1044 instructions using entry vectors of length 1, a vector of length 2 scheme would require ~1100, length 3 ~1220, and length 4 ~1340 instructions. The implementor of an instruction set should decide when the additional locations expended for larger entry vectors are no longer worth the additional speed.

Although we originally hoped for as much as 8% faster inner loops and 4% overall speed improvement, Gene McDaniel measured only 2% faster execution for Mesa (excluding disk wait) using a length 3 entry vector; microstore increased about 120 locations. Investigation revealed that increased traffic on Mar (by overlapped Fetch_ and _Md) was causing IFU not ready to occur more often, offsetting the fact that fewer processor cycles were needed. Forwarding saved about .2 cycles/opcode.

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

**Timing Summary**

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

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

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

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

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

uses Mar adds one cycle of delay. If the opcodes in the stream start at the odd bytes in words, then the processor will get NotReady 40% of the time.

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

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

**Use of MemBX and the Duplicate Stk Regions**

The present Mesa implementation requires 34 cycles for a local XFER and 54 cycles for an external XFER, excluding memory wait, and measurements made on the Mesa compiler showed that 38% of all cycles were spent in XFER. For this reason, speed improvements in XFER are an important objective.

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

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

We have not constructed examples that use this idea, but a savings of 50% in average XFER timing has been projected for Mesa.

**Traps**

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

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

Each trap vector is dispatched into by IFUJump exactly as though it were an opcode. *B_PCX' reads the PC of the opcode that would have been executed if the trap had not occurred and RBase, MemBase, and _Id stuff are set according to that opcode (in every case except NotReady all are undefined at a NotReady trap)*.

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

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

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

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

```
NotReady:
        IFUJump[0];                          *Can't convert to IFUJump[2] because stack may be empty
        T_Stack& 1_Md, IFUJump[2];           *Convert case 1 to case 2
        IFUJump[2];
        T_(Id) (PCX') 1, StkP_StkP+1, Return;        *Resume the opcode which didn't really exit
```

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

The IFU will trap at the cache data parity error location, if it detected invalid parity on any byte sent by the memory system. PCX will always correctly point at the opcode that would have been executed next had the trap not occurred; however, the opcode and operands pointed at by PCX are not necessarily the ones that suffered the parity error. This occurs because the pipe has continued ahead of PCX. The most confusing case occurs when the opcode following PCX was a jump; in this case the opcode fetched by the jump may have caused the parity error, in which case PCX+/ jump displacement is limited to the range PCX $400_8$ to PCX+$377_8$.

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

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

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

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

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

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

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

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

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

*Remark*

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

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

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

**IFU Reset**

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

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

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

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

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

**Rescheduling**

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

The *next* IFUJump after Reschedule transfers to the entry vector for the opcode as usual; the reschedule trap address will drop into the IFAddr register at $t_2$ of this instruction, and the first IFUJump after that will dispatch into the reschedule trap vector. This means that second IFUJump will trap unless the second IFUJump occurs on the instruction immediately after the first IFUJump, in which case the trap will not occur until the third IFUJump. IFUJump's that experience a NotReady trap are not counted.

The entry vector at the reschedule trap location is entered *as though it were the next opcode*. When Reschedule is used by io tasks to request the wakeup of another process, this fact is unimportant. However, the other use of Reschedule is in continuation from map (and other) faults. In this application, the reschedule trap will wind up restoring the IFU state by executing an appropriate number of _Id's and eventually branching back to the instruction that experienced the fault. The continuation method is discussed later.

Opcodes which might execute for a long time, such as block transfer and BitBlt, must check for rescheduling explicitly, and the (emulator only) Reschedule branch condition

makes this check easier. If such opcodes did not check for rescheduling, then service to the io device might be postponed for too long.

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

When the reschedule trap vector is entered, the IFU is in an undefined state except for PCX', and PCF_ is needed to restart the IFU at the continuation address.

### Breakpoints

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

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

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

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

```
Continue:
        . . .
        IFUReset;                       *Stop future IFU fetches and clear pipe
        T_41C;
        Cnt_T;
        IFUReset, Goto[.,Cnt#0&-1];     *Reset after previous IFU fetches complete
        BrkIns_Opcode;                  *Load opcode which BrkP replaced
        PCF_BreakAddress;               *Restart IFU at address of BrkP
        Noop;                           *No-op required after PCF_ before IFUJump
        IFUJump[0];                     *Resume program
```

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

### Reading and Writing IFUM

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

When IFUM is loaded, it is addressed by the instruction set InsSet[0:1] and BrkIns. The data must remain on B for two cycles, so tasking must be disabled and the instruction

following the one with IFUMLH/RH_ must put the same data on B. If this data comes from RM or T, the register must not have been loaded in the cycle preceding the IFUMLH/RH_ (because the bypass logic will change the B select from Pd or Md to RM or T, possibly glitching data on B). The following subroutines illustrate loading and reading back IFUM.

```
WriteIFUM:
        IFUReset;                       *Stop future IFU fetches and clear the pipe
        T_41C;
        Cnt_T;
        IFUReset, GoTo[.,Cnt#0& 1];     *Reset after previously issued fetches complete
        InsSetOrEvent_RMaddr0;          *Load 2 instruction set bits forming IFUM address
        BrkIns_RMAddr1;                 *Load 8 opcode bits forming IFUM address
        TaskingOff;                     *Ensure no B glitch below and let BrkIns_ settle for 1 cycle
        IFUMLH_RMdataHi;                *Write high part of IFUM
        B_RMdataHi;                     *Keep data good a little longer (mustn't glitch)
        IFUMRH_RMdataLo;                *Write low part of IFUM
        B_RMdataLo, TaskingOn;          *Keep data good a little longer
        IFUReset, Return;               *Clear BrkIns

ReadIFUM:
        IFUReset;                       *Stop future IFU fetches and clear the pipe
        T_41C;
        Cnt_T;
        IFUReset, GoTo[.,Cnt#0& 1];     *Reset after previously issued fetches complete
        BrkIns_RMaddr1;                 *Load 8 opcode bits forming IFUM address
        InsSetOrEvent_RMaddr0;          *Load 2 instruction set bits forming IFUM address
        Noop;                           *Two instructions must elapse after loading BrkIns
                                        *one after loading InsSet (?Two noops after loading InsSet
                                        *might be better since this is a tight path?)
        RMdataHi_IFUMLH;                *Read IFUM into RM.
        RMdataLo_IFUMRH;
        IFUReset, Return;               *Clear BrkIns
```

**Continuing from Processor Faults**

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

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

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

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

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

Sample microcode for saving emulator state is as follows:

```
%Must first save the volatile branch conditions; Overflow and Carry won't change unless an arithmetic
ALU operation is executed, so saving them can be deferred.  T, the first item saved, is written into the
RM region reserved for Save using the change-RBase-for-write FF decode.
%
Save:     FreezeBC, DblGoTo[ALUls,ALUge,ALU<0];
ALUls:    SavedT_T;
          T_0C, GoTo[SaveBC];
ALUge:    SavedT_T, DblGoTo[ALUgr,ALUeq,ALU#0];
ALUgr:    T_1C, GoTo[SaveBC];
ALUeq:    T_2C;
*Have a code, 0, 1, or 2, in T indicating the state of the ALU<0 and ALU=0 branch conditions.
SaveBC:   SavedALULEZ_T;                  *Save the branch condition code
          T_Pointers;                     *T_MemBase, MemBX, and RBase
          T_T Or (100000C);               *Make negative
          RBase_RBase[SaveRMRegion];
*Now choose two numbers such that their sum produces the correct ALUcry and Overflow branch
*conditions.
          SavedPointers_T, MemBase_SaveBaseReg, DblGoto[Cry,NoCry,Carry];
Cry:      DblGoTo[CryOvf,CryNoOvf,Overflow];
NoCry:    DblGoTo[NoCryOvf,NoCryNoOvf,Overflow];
CryOvf:   SaveA1_100000C;
          SaveA2_100000C, GoTo[SaveRest];   *Numbers such that SaveA1+SaveA2 produces
                                            *Overflow and Carry result
NoCryNoOvf:
          SaveA2_0C, GoTo[.+2];
CryNoOvf: SaveA2_1C;
          SaveA1_177777C, GoTo[SaveRest];
NoCryOvf: SaveA1_77777C;
          SaveA2_77777C, GoTo[SaveRest];

SaveRest:
          SavedPCX_Not(PCX');
          T_Not(IFUMLH');                 *Read IdCnt and InsSet in IFUMLH[0:4]
          SavedIdCnt_LdF[T,0,2];
          T_T and (14000C);
          T_RSh[T,2];
          SavedInsSet_T+(100000C);        *Set up word for InsSetOrEvent_ below
          . . .                           *Code to save rest of state (all easy)
```

Sample microcode for continuing is given below:

```
Resume:   . . .                           *Restore all processor registers except T, Cnt, RBase,
                                          *and MemBase.
          InsSetOrEvent_SavedInsSet;      *Restore the IFU instruction set number.
          PCF_SavedPCX;                   *Restart IFU at address of the opcode that faulted
          WakeUp[ContTask];               *Wakeup the special task used for continuation.
          Noop;                           *No-op required so that the instruction after the IFUJump
```

```
                                        *below will be executed by the continuation task.
            Cnt_SavedIdCnt, IFUJump[0]; *Continue execution in the continuation task at Cont0

Resume1:  Skip[Cnt=0& 1], At[Resume1Loc];   *Reissue the appropriate number of _Id's to put
            A_Id, GoTo[. 1];                *the IFU in the state it was in at the fault.
            Cnt_SavedCnt;                   *Restore Cnt
            . . .                           *Restore Md by fetching from a convenient storage
                                            *location.  Then repeat the Fetch_ or Store_ that
                                            *faulted using a convenient base register and restore
                                            *the base register (complicated code here needs careful
                                             *thought).
            T_SaveA1;
            Pd_T+SaveA2;                    *Restore Carry and Overflow branch conditions.
            T_SavedT, TaskingOff;           *Restore T register
*Below, the TaskingOff, WakeUp, TaskingOn sequence insures that precisely one emulator instruction will
*be executed after the TaskingOn before the continuation task runs.
            BDispatch_SavedALULEZ;          *Dispatch to 0, 1, or 2 in table based on
                                            *ALU>0, ALU<0, or ALU=0.
            WakeUp[ContTask];               *Wakeup the special task reserved for continuation.
            Link_SavedLink, At[ConTab,0];   *Restore Link and ALU>0
            TaskingOn;
            Pd_Not(Pointers_SavedPointers), GoTo[COK];
            Link_SavedLink, At[ConTab,1];   *Restore Link and ALU<0
            TaskingOn;
            Pd_Pointers_SavedPointers, GoTo[COK];
            Link_SavedLink, At[ConTab,2];   *Restore Link and ALU=0
            TaskingOn;
            Pd_(SavedPointers) xor (Pointers_SavedPointers), GoTo[COK];
COK:        FreezeBC, GoTo[.];

*The special restart task needed for continuation
ContinueInit:
            RBase_RBase[SavedTPC];          *Initialization code for the task
*First of two wakeups comes here change emulator's TPC to Resume1 and block.
Cont0:      Block;
            T_Resume1Loc;
            Link_T, TaskingOff;
            LdTPC_0C;                       *Restart emulator at Resume1
            TaskingOn;
            Block;
*Second of two wakeups comes here.  Reload emulator TPC with continuation address.
Cont1:      Link_SavedTPC;
            LdTPC_0C;                       *Restart emulator at saved continue address
            Branch[Cont0];
```

**IFU Testing**

The IFU test control register is loaded by the IFUTest_B function; when not testing, this register should contain 1, and it is loaded with 1 by the IFUReset function. IFUTest.15 disables the periodic wakeup request to the Junk task discussed in the "Slow IO" chapter; when IFUTest.15 is 0, the junk wakeups occur 60 times/sec and are dismissed by any IFUTest_ function.

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

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

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

```
TaskingOff;
IFUTest_TestEn;
IFUTick;
PCF_value;
```

where PCF_value is just an example any other IFU function or an IFUJump could be used instead.

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

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

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

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

In other words, a memory reference can be simulated with the IFU test feature by (1) ticking the IFU through a cycle in which it makes a reference; (2) ticking the *TestMemAck* response of the memory system with IFUTest_B and IFUTick; (3) ticking *TestMakeF_D*; (4) ticking with *TestFG* and *TestParity* holding simulated memory data.

**Details of Pipe Operation**

The IFU is a six-stage pipeline, starting with words fetched from memory, and ending with opcode starting addresses delivered to the control section and operands delivered to the

processor. The levels are named: F, G, H, J, M and X. Each level has a data-valid bit indicating whether or not it contains something useful.

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

Pipe control is straightforward in principle. The F and G levels are 16-bit registers filled from the cache. Following PCF_B, if there is space in the pipeline for another word, the IFU will start a reference at $t_1$ of any cycle in which the processor is not using Mar (so as many as 2 IFU references can be outstanding). Cache words are stored in F at $t_1$, then dropped into G at $t_2$; bytes drop into H at $t_3$ or J at $t_4$; there are bypass paths to get bytes directly from F/G into J when H is invalid. As the processor executes opcodes, F and G become invalid, and the IFU refills them from memory automatically. This continues until the IFU is reset by the processor, or encounters a *pause* opcode.

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

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

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

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

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

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

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

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

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

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

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

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

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

## Timing Details

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

### First case: Restart IFU at even byte

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

t2: F, G, H, J, and M levels are made invalid.

t3: Reference the word containing FOO.

t5: Reference word containing FOO+2.

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

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

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

t10: Distinguish 5 cases below.

*FOO is a one-byte regular opcode*

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

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

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

### *FOO is a two-byte regular opcode*

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

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

t12: Load G from F; F becomes invalid.

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

### *FOO is a three-byte regular opcode*

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

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

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

### *FOO is a one-byte jump opcode*

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

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

t13: Reference the second word along the jump path.

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

t16: Load J from F, etc.

### *FOO is a two-byte jump opcode*

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

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

t12: J and H become invalid.

t13: Reference the second word along the jump path.

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

## Second case: Restart IFU at odd byte

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

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

t3: Reference the word containing FOO.

t5: Reference word containing FOO+1.

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

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

t9: Load F from the FOO+1 reference.

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

### *FOO is a one-byte opcode*

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

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

t12: Load G from F; F becomes invalid.

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

### *FOO is a two-byte opcode*

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

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

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

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

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

### *FOO is a three-byte opcode*

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

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

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

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

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

## Slow IO

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

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

**Table 21: IO Register Addresses**

| *Number* | *Name* | *Comment* |
|---|---|---|
| 10 | DiskControl | Disk control register |
| 11 | DiskMuff | Disk muffler control |
| 12 | DiskData | Disk FIFO data |
| 13 | DiskRam | Disk format RAM |
| 14 | DiskTag | Disk tag register |
| 15 | EData | Ethernet input or output data |
| 16 | EControl | Ethernet control and status |
| 360 | PixelClock | DDC pixel clock |
| 361 | Mixer | DDC mixer |
| 362 | CMap | DDC CMap |
| 363 | | DWTFlag* (DispM analog of DWTFlag) |
| 364 | | DHTFlag* (DispM analog of DHTFlag) |
| 365 | BMap | DDC BMap |
| 366 | | NLCB* (DispM analog of NLCB) |
| 367 | | Statics* (DispM analog of Statics) |
| 370 | Status | DDC muffler and OIS data |
| 372 | MiniMixer | DDC MiniMixer |
| 373 | DWTFlag | DDC word task control |
| 374 | DHTFlag | DDC horizontal task control |
| 375 | HRam | DDC horizontal waveform control |
| 376 | NLCB | DDC next line control block |
| 377 | Statics | DDC debugging control |

### Input/Output Functions

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

TIOA is loaded at $t_2$ from B[0:7] by the TIOA_B function, or TIOA[5:7] can be loaded from FF[5:7] while preserving TIOA[0:4] by the TIOA_small constant function. Pd_Input, Pd_InputNoPE, or Output_B functions can be issued in the instruction immediately following the one that loads TIOA.

Most input registers include odd byte parity with IOB data. The Pd_Input function reads IOB data and checks parity. The Pd_InputNoPE function reads IOB data without a parity check; this is useful when determining whether a device exists (IOB has bad parity if a nonexistent register is selected). The enabling and timing of parity error halts is discussed in the "Errors" chapter.

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

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

**Table 22: Task Assignments**

| *Number* | *Name* | *Comment* |
|---|---|---|
| 0 | EMU | The emulator |
| 1 | CON | Special task for restarting emulator after faults |
| 2 | JNK | Junk task (awakened every 32 ms) |
| 3 | DHT | Display horizontal task |
| 4 | AHT | DispM terminal interface horizontal task |
| 6 | EOT | Ethernet output task |
| 7 | EIT | Ethernet input task |
| $11_8$ | AWT | DispM terminal interface word task |
| $12_8$ | SIM | Task simulator |
| $13_8$ | DWT | Display word task |
| $14_8$ | DSK | Disk io |
| $17_8$ | FLT | The fault task |

### IO Opcodes

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

```
INPUT:
        TIOA_a;
        Stkp_Stkp+1;
        Stack_Input, IFUJump[0];

OUTPUT:
```

```
TIOA_a;
Output_Stack& 1, IFUJump[0];
```

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

**Wakeup, Block, and Next**

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

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

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

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

Suppose that a task blocks with the following instruction:

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

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

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

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

Another consequence of "Next Lies" is that IOAtten may be incorrect when

"Next Lies" is occurring. Consequently, branch on IOAtten is illegal during an instruction that blocks and might cause hold.

The second strategy monitors TIOA becoming equal to a particular device value. In this case the wakeup request is lowered at $t_0$ of the second instruction following a wakeup, and the task *must not block until the third instruction*. The disk controller has used this strategy, which has the draw back that if TIOA inadvertently assumes the particular device value for any other task, the hardware will malfunction. A consequence of *any* device using this strategy is that *all* tasks must be careful to initialize TIOA properly when first awakened.

The third strategy waits for some Output_B or Pd_Input operation to reset the wakeup condition. This would reset the condition at $t_3$ or $t_5$ of the Output_B instruction, and the wakeup would be lowered at $t_4$ or $t_6$; in this case the task *must not block until the third or fourth instruction after the Output_B or Pd_Input* to avoid an erroneous wakeup. The exact requirement depends upon the io controller the disk controller, for example, lowers its wakeup request at $t_4$ and can block in the third instruction after Output_B, while the display controller horizontal task lowers its wakeup request at $t_5$ and can block in the fourth instruction.

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

**SubTasks**

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

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

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

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

A task presenting SubTask signals generally *must Block at the same location each iteration*

since there is only a single TPC value for all of the SubTasks. Hence, the full generality of tasking is unavailable the microcode for these tasks must be coded as though the wakeup mechanism were a priority interrupt.

**Illegal Things IO Tasks Must Not Do**

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

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

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

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

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

(6) It is illegal to issue Wakeup[n] if task n might run in the next cycle. Wakeup[n] must be executed with TaskingOff in such circumstances.

# Fast IO

The fast input/output system provides high-bandwidth data transfers between storage and io devices. Transfers occur in units of one munch (= 16 words); the addresses of the 16 words must be *i*, *i*+1, ..., *i*+15, where *i* mod 16 = 0. One word is transferred every clock, for a peak bandwidth of 533 x $10^6$ bits/second. A fast device is also interfaced to the slow io system, from which it receives its control information, since there is no way for the device to communicate directly with the processor using the fast io system.

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

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

### Transport

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

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

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

### Wakeups and Microcode

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

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

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

**Latency**

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

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

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

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

## Disk Controller

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

*Keep Figure 13 in view while reading this chapter.*

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

| | | |
|---|---|---|
| $10_8$ | DiskControl | Output_B to control register |
| $11_8$ | DiskMuff | Output_B muffler control and Pd_Input to read muffler |
| $12_8$ | DiskData | Pd_Input to read FIFO or Output_B to write FIFO data |
| $13_8$ | DiskRam | Output_B to format RAM |
| $14_8$ | DiskTag | Output_B to tag register |

*Note:* other tasks *must not select these TIOA addresses* at any time; doing so may cause the disk controller to malfunction.

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

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

The daisy-chain cable contains the following signals:

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

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

### Disk Addressing

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

Firmware may control the following parameters:

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

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

Because sector formats are flexible, firmware can adjust the controller to system needs. The sector formats specifically envisioned in the design of the controller include 28 256-word sectors for Alto Diablo emulation and Pilot, 16 512-word sectors for Juniper, and 9 1024-word sectors for Alto Trident emulation.

### Sector Layout Considerations

Each block within a sector can be either read, written, or checked. However, once any block is written, later blocks in that sector cannot be read during that disk revolution. (Later blocks should be readable on subsequent disk revolutions, though this is not guaranteed and no existing software depends on this.) Reading or writing must start with the first block in the sector and continue; since check bits are stored at the end of each block, the entire block must be read to verify its data or correct errors; however, one does *not* have to read or write subsequent blocks in the sector. After a check-block operation is started, the controller inhibits writing later blocks within a sector without a specific "OK" from the firmware.

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

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

Feasible sector layouts are determined by several considerations. First, each disk drive is configured to generate 117 *subsector* pulses/revolution. The disk controller has a *subsector counter* for each drive that is initialized to N when an index pulse is received from the drive; it then counts down to -1, generates a sector pulse, and reinitializes itself. The firmware can specify N (0 to $17_8$) independently for each disk drive and thus create 117/(N+1) sectors/revolution. If this division leaves any remainder, then there will be one or more unused subsectors at the end of the cylinder.

Note that the quantization of cylinders into subsectors allows a sector size to be specified in units of 10,080/117 = 86.15 words/subsector.

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

| | |
|---|---|
| Total words/track = | 10,080 |
| Words lost for the 1st block in a sector = | 38 |
| Words lost for successive blocks = | 14 |
| Required gap at end of sector = | (microcode-dependent) |

A *track* is the path swept through one revolution by a single head at a single cylinder. "Words lost" for each block include 2 words of error detection and correction (32 bits of ECC code) which are always added at the end of the data written, plus preamble, postamble, and various other delays required by the controller and drive electronics. These are detailed later under "Format RAM and Sequence PROMs". Additionally, to enable the microcode to process consecutive sectors, there must be some gap between the end of the last block and the end of the sector; the number of words required depends on the amount of time the microcode requires to complete processing the last block and issue a command for the next sector.

For the Alto Trident format there is a 2-word Header block, 10-word Label block, and 1024-word data block; total words lost for disk formatting is 38 for the first block, 14 for the second, and 14 for the third; altogether, this requires 1100 words/sector. The next larger multiple of the subsector size is 86.15*13 = 1119 words, leaving 19*1.65 = 31.35 ms of gap at the end of the sector. Thus 13 subsectors/sector are required, yielding 117/13 = 9 sectors/revolution.

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

29 sectors of 256 data words each (4 subsectors/sector),
16 sectors of 512 data words each (7 subsectors/sector), or
9 sectors of 1024 data words each (13 subsectors/sector).

*Note:* The 29-sector and 16-sector formats do not divide the disk evenly but rather yield an unusable leftover fraction of a sector; the 9-sector format does divide the disk evenly. The 9-sector format is compatible with the Alto Trident 9-sector format (used by BCPL Trident software such as IFS). The 16-sector format is *not* compatible with the Alto Trident 16-sector format (used by Juniper), though it is usable if interchangeability of disk packs with Altos is not required. The 29-sector format has no Alto analogue.

**Table 23: T-80 Specifications and Characteristics**

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

### General Firmware Organization

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

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

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

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

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

Execute a drive-select command, if the drive differs.
Load the sector size only if different, and block until index.
Load the format RAM only if word count or commands differ.
Execute a Control Tag (seek) command only if the cylinder differs, and wait (continuing to count sectors) until the drive becomes ready again.
Execute a Head Tag command.
Block until, at a sector wakeup, the *next* sector is the one wanted.
Load the appropriate transfer command into the control register
Block until the next sector wakeup.

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

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

Once a transfer has started, the disk task will be woken according to the number of words in the FIFO, and it will send or receive the appropriate number of words. Read and compare operations are performed by firmware, as well as detecting checksum errors at the end of reading. During writing, firmware must provide one word of sync bits ($201_8$ standard, $001_8$ for Alto Trident emulation) followed by the specified number of words for that block (the controller will append 2 words of checksum). During read, the controller will look for, and discard, the first word of sync bits, then firmware must accept the specified number of words for that block, followed by two words of checksum to be discarded, followed by the ECC remainder to be used for error detection/correction.

**Task Wakeups**

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

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

assume the wakeup reason to minimize overhead. The various conditions are as follows: IndexTW, SectorTW, TagTW, RdFifoTW, and WrFifoTW; these wakeup conditions are detailed in the "Muffler Input" section.

### Control Register

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

| | |
|---|---|
| B[5] | Clear EnableRun |
| B[6] | Set DebugMode |
| B[7] | Set BlockTilIndex |
| B[8:9] | Operation for first block of sector, where the operations are:<br>0 = Done (finished with all blocks in this sector)<br>1 = Write<br>2 = Read and check<br>3 = Read |
| B[10:11] | Operation for second block of sector, as above. |
| B[12:13] | Operation for third block of sector, as above. |
| B[14:15] | Operation for fourth block of sector, as above. |

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

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

BlockTilIndex can be set to disable sector and index task wakeups until (a) the selected drive is ready, and (b) an index pulse is received from the drive. It is cleared by an index wakeup. This is useful after switching drives or executing a ReZero operation, either of which causes the controller to lose sector synchronization with the drive. BlockTillIndex prevents the wakeup conditions from being set until these conditions are met, but does *not* clear any such wakeups that have already occurred. To prevent races, it is necessary to clear SectorTW and IndexTW, then set BlockTillIndex, then clear SectorTW again.

A request for a sector transfer is initiated by loading bits 8 and 9 of the control register with a non-zero value. Then the controller will wait until the *next* sector pulse to set the "Active" flip-flop and execute the transfer. Once a transfer has been started, it may be aborted by loading a new value into the control register twice. The first will clear the Active flip-flop, and the second will load the control register. (When Active, the control register is enabled for shifting commands rather than loading of io data.)

### Format RAM and Sequence PROMs

The format RAM is a 16-word by 12-bit register that holds commands and delay counts used by the controller during a transfer. Words within the RAM are used according to the following table; the example values are appropriate for Alto Diablo disk emulation (2-word header, 8-word label, and 256-word data record).

| *Addr* | *Description* | *Example Value* |
|---|---|---|
| 00 | Word count of the first block | 0001 |
| 01 | Word count of the second block | 0007 |
| 02 | Word count of the third block | 0377 |
| 03 | Word count of the fourth block | 0000 |
| 04 | Control tag command for a read operation | 0104 |
| 05 | Control tag command for a write operation | 0204 |
| 06 | Control tag command to set Head Select | 0004 |
| 07 | Control tag command to zero the tag bus | 0000 |
| 08 | Word count to write zeroes before writing the 1st block of a sector | 0033 |
| 09 | Word count to write zeroes before writing the sucessive blocks | 0006 |
| 10 | Word count to wait before reading the 1st block of a sector | 0011 |
| 11 | Word count to wait before reading the sucessive blocks | 0002 |
| 12 | Word count of ECC words plus one | 0002 |
| 13 | Word count of 2 | 0001 |
| 14 | Word count of 1 (minimum count) | 0000 |
| 15 | Not used | 0000 |

Notice that the format RAM contains both word counts and tag commands. *Word counts are 1 less that the desired count*. Tag commands will be loaded into the tag register (see below) and then used as a "control tag function" by the Trident disk. The values in the right column are those used for the Alto Diablo emulation format. Notice that all but the first 4 values are determined by characteristics of the drive being used as opposed to the specific sector format. The meaning of the tag command values can be found in the "Tag Register" section.

The format RAM is addressed in two ways. During a transfer, sequence PROMs move data from the RAM into either a tag register or a count register. At other times, the Dorado may address the RAM with the *RAM Address register*, which is zeroed when the control register is written; executing an Output to the DiskRam register writes IOB into the RAM at the current address and then increments the address. Loading the last word in the format RAM turns on the EnableRun flip-flop allowing normal disk control activity. The format RAM may be read via the muffler scheme discussed later.

There are two sequence PROMs, one for reading (or checking) and one for writing. The PROMs are addressed by a program counter that is initialized to zero at the beginning of a sector and is incremented upon completion of each PROM program action. Either the read PROM or the write PROM is selected according to the operation being performed on the current block.

The sequence PROMs are clocked by WordClock, which is derived from the disk bit clock, which in turn is derived from timing information pre-recorded on the disk pack. The subsector pulses generated by the drive are also derived from this timing information. This enables very precise placement of the data on the disk, in a manner that is independent of the disk’s rotational velocity or the Dorado’s clock rate.

The read and write sequence PROMs are described in the following tables.

| *Write Sequence PROM* Addr | *Description* | *Duration (WordClocks)* |
|---|---|---|
| 00 | Issue tag command in RAM[6] (head select) | 1 |
| 01 | Delay (wait for head select to settle) | RAM[13]+1 |
| 02 | Issue tag command in RAM[5] (write command) | 1 |
| 03 | Write long preamble for first block | RAM[8]+1 |
| 04 | Write sync word | 1 |
| 05 | Write data for first block | RAM[0]+1 |
| 06 | Write first ECC word | RAM[14]+1 |
| 07 | Write second ECC word and 2 postamble words | RAM[12]+1 |
| 08 | Advance control register to the operation for the next block | RAM[14]+1 |
| 09 | Issue tag command in RAM[5] (write command) | 1 |
| 10 | Write short preamble for second block | RAM[9]+1 |
| 11 | Write sync word | 1 |
| 12 | Write data for second block | RAM[1]+1 |
| 13 | Write first ECC word | RAM[14]+1 |
| 14 | Write second ECC word and 2 postamble words | RAM[12]+1 |
| 15 | Advance control register to the operation for the next block | RAM[14]+1 |
| 16-22 | Same as 09-15, except step 19 uses RAM[2]+1 | |
| 23-29 | Same as 09-15, except step 26 uses RAM[3]+1 | |
| 30 | Zero the tag bus | 1 |
| 31 | Not used | |

| *Read Sequence PROM* Addr | *Description* | *Duration (WordClocks)* |
|---|---|---|
| 00 | Issue tag command in RAM[6] (head select) | 1 |
| 01 | Delay (wait for head select to settle) | RAM[13]+1 |
| 02 | Delay (skip over early part of preamble) | RAM[10]+1 |
| 03 | Issue tag command in RAM[4] (read command) | 1 |
| | *Note: WordClocks cease until controller has read sync word from disk* | |
| 04 | Read data for first block | RAM[0]+1 |
| 05 | Read ECC words | RAM[13]+1 |
| 06 | Compute first word of ECC remainder, issue tag command in RAM[6] | 1 |
| 07 | Compute second word of ECC remainder | RAM[14]+1 |
| 08 | Advance control register to the operation for the next block | RAM[14]+1 |
| 09 | Delay (skip over early part of preamble) | RAM[11]+1 |
| 10 | Issue tag command in RAM[4] (read command) | 1 |
| | *Note: WordClocks cease until controller has read sync word from disk* | |
| 11 | Read data for second block | RAM[1]+1 |
| 12 | Read ECC words | RAM[13]+1 |
| 13 | Compute first word of ECC remainder, issue tag command in RAM[6] | 1 |
| 14 | Compute second word of ECC remainder | RAM[14]+1 |
| 15 | Advance control register to the operation for the next block | RAM[14]+1 |
| 16-22 | Same as 09-15, except step 18 uses RAM[2]+1 | |
| 23-29 | Same as 09-15, except step 25 uses RAM[3]+1 | |
| 30 | Zero the tag bus | 1 |
| 31 | Not used | |

### Tag Register

The 16-bit tag register drives the tag bus on the daisy-chain cable; all disk drive commands are initiated through the tag register. The tag register is sometimes loaded from IOB via an Output command to DiskTag, sometimes from the format RAM. Loading a Head Tag, Cylinder Tag, or Control Tag into the tag register (from either source) activates a timing circuit that handles all timing requirements of the Trident drive as follows: Only the tag bus bits are enabled for the first 200 ns; then the Tag[0:3] bits are also enabled for 1.2 ms; finally, the Tag[0:3] bits are disabled again and the TagTW flip-flop is set to wakeup the disk task (indicating completion of the Tag instruction). The Drive Select Tag (Tag[0]) does not activate the timing circuit, since the timer counts disk clock cycles, but disk clocks are invalid during drive select changes.

Bits 4 through 15 of the tag register are interpreted according to the following table:

Tag[0] Drive select and subsector count

Tag[4:15] are interpreted by the controller to effect drive select or subsector counter changes. The tag timing and wakeup circuit is not activated; firmware must take care of the timing by first loading Tag[4:15] as desired but with Tag[0:3] equal 0, then or-ing in the Tag[0] bit and outputting again.

| | |
|---|---|
| 4:9 | Subsector count<br>Divide the 117 subsector pulses from disk by subsector count+1 to form Sector pulses (Tag[4:5] are presently unimplemented).<br>Tag[4:9] = 3 yields 29 sectors large enough for 256-word data blocks<br>Tag[4:9] = 6 yields 16 sectors large enough for 512-word data blocks<br>Tag[4:9] = $14_8$ yields 9 sectors large enough for 1024-word data blocks |
| 10 | Load subsector from Tag[4:9] for the drive selected *prior* to the execution of this tag instruction. |
| 11:15 | Drive select<br>The basic controller handles up to 4 disk drives; additional units may be accommodated by adding drive dependent logic on an additional board and connecting it in in place of drive 3. To allow this, the 5 bit drive select field is interpreted as follows. |

| | |
|---|---|
| 0 - 3 | select drive 0 to 3, respectively |
| 4 - $36_8$ | select drive 3 |
| $37_8$ | don't select any drive |

Tag[1] Head Tag

Loads a register in the drive that selects the head to be used during subsequent read/write commands. A Tag wakeup occurs at completion (1.6 ms).

| | |
|---|---|
| 4:7 | Unused |
| 8 | Off Cylinder may be activated during a read to attempt recovery of unreadable data. It causes cylinder positioning to be offset 80 micro-inches. |
| 9 | Determines direction of offset if bit 8 is set. |
| 10:15 | Head number values from 0 to 4 are valid for a T-80, 0 to 19 for a T-300. The drive will turn on "EndOfCylinder" (alias HeadOverflow) error if an invalid head address is issued. |

Tag[2] Cylinder Tag

Causes the drive to seek to the specified cylinder. A Tag wakeup occurs after the tag timing sequence has completed (1.6 ms), and the NotReady status bit is raised until the seek has completed (3 to 55 ms depending on the seek distance).

| | |
|---|---|
| 4:15 | Cylinder number (0 to 814 for Trident disks presently in use). An illegal cylinder number will cause DeviceCheck to be raised. |

Tag[3] Control Tag

A Tag wakeup occurs at command completion (1.6 ms) *and* upon completion of the last read/write operation in a sector. Generally, Control Tag commands are issued only by the controller itself (using tag commands from the format RAM) rather than by the microcode; Device Check Reset and ReZero are an exception.

| | |
|---|---|
| 4 | AltoLeader special flag to the controller that allows disks written by an Alto Tricon Controller to be read. This bit should only be used for the Alto Trident simulation. |
| 5 | Unused |
| 6 | Strobe Late causes data recovery circuits within the drive to sample data early within the data bit time (for recovery when the drive is experiencing excessive read errors). |
| 7 | Strobe Early like StrobeLate except in the obvious way. |
| 8 | Write turns on the write circuits. |
| 9 | Read turns on the read circuits. |
| 10 | Unused |
| 11 | Reset Head register zeroes the head address register in the drive. |
| 12 | Device Check Reset resets all latched error conditions in the drive. |
| 13 | Head Select turns on the head selection circuits, in conjunction with a Read or Write. |
| 14 | ReZero repositions the heads to cylinder 0 (if the heads are loaded) and resets the head address register; resets SeekIncomplete and DeviceCheck error conditions. |
| 15 | Head Advance increments the head address register in the drive. |

### FIFO Register

Data to/from the disk is buffered through a 16-word FIFO (25 ms of buffer), which is read/written with Pd_Input/Output_B when TIOA selects DiskData. Each FIFO word holds 16 data bits, 2 parity bits, and a 2-bit field indicating that the next word to be read is either write, read, or read-and-check type data. During output to the disk, the controller checks parity both when receiving data on the io bus and again when reading the FIFO. During a disk read, parity is computed before writing into the FIFO, is passed through the FIFO, and is then written on the io bus for the processor to test.

### Muffler Input

Dorado uses a multiplexor scheme called the muffler system for reading miscellaneous logic signals during debugging from the Alto or baseboard. The disk controller also allows a muffler address to be specified on an Output to the DiskMuff register; in this way, any DskEth board signal available through the multiplexors (mufflers) is also available for firmware sampling. Other bits of the DiskMuff register output specify other operations as follows:

| | |
|---|---|
| B[0] | Simulate read data of 1 for 1 cycle (for use by diagnostic programs) |
| B[1] | Simulate read clock of 1 for 1 cycle (for use by diagnostic programs) |
| B[2] | Clear CompareErr done by disk task if a read&compare is found to be OK |
| B[3] | Set ReadDataErr done by disk task to inhibit future writes |
| B[4] | Clear the index wakeup flip-flop |
| B[5] | Clear the sector wakeup flip-flop |
| B[6] | Clear the tag wakeup flip-flop |

| | |
|---|---|
| B[7] | Clear all error flip-flops within the controller (not the disk drive) |
| B[8:15] | Muffler address signals are enumerated below |

Following an output to the DiskMuff register, the firmware must wait one cycle before inputting the selected muffler signal with Pd_Input. The state of the signal selected will be driven on IOB[15], and the remaining bits will be zero. For the purpose of examination from Midas, the signals are grouped into 16-bit words, as shown in the following table. The bits within each word and an appropriate explanation follow:

KSTATE various bits indicating the state of the controller

| | | |
|---|---|---|
| 000 | TempSense | see "Dorado Debugging Interface" document |
| 001 | IndexTW | disk task wakeup is due to an index pulse; index pulses occur once/disk revolution (16.7 ms) and are used to synchronize the hardware subsector counter and the firmware sector counter. An index pulse also causes a SectorTW. |
| 002 | SectorTW | disk task wakeup is due to a sector pulse. To maintain a reliable sector count in a race-free manner, the microcode must (a) check for SectorTW, and upon finding it set increment the sector number and clear SectorTW; (b) check for IndexTW, and upon finding it set zero the sector number and clear *both* IndexTW and SectorTW. |
| 003 | TagTW | disk task wakeup is due to completion of a Head Tag, Cylinder Tag, or Control Tag command. This occurs 1.6 μs after issuing an Output to the DiskTag register, and also upon completion of the last read/write transfer in a sector. |
| 004 | RdFifoTW | disk task wakeup is due to presence of at least 3 words in the FIFO during a normal read or 1 word during a read-and-check. During a normal read, an Input that reduces the FIFO below 3 words will drop RdFifoTW in time for a Block to take effect on the 5th cycle following the Input; this permits a 2-cycle loop (Input, Block). During a read-and-check, an Input that empties the FIFO will drop RdFifoTW in time for a Block to take effect on the 3rd cycle following the Input; this permits a 4-cycle loop (Input, no-op, no-op, Block). |
| 005 | WrFifoTW | disk task wakeup is due to space for at least 4 words in the FIFO. An Output that reduces the free space below 4 words will drop WrFifoTW in time for a Block to take effect on the 5th cycle following the Output; this permits a 2-cycle loop (Output, Block). WrFifoTW is enabled to occur by selecting TIOA[DiskData] when a write command is in progress; it is disabled by TIOA[DiskControl], which the microcode executes after outputting the last data word of a block. One more WrFifoTW will occur after all data has actually been sent to the disk. |
| 006 | ReadData | Data bit from the disk (available for diagnostics) |
| 007 | WriteData | Data bit to the disk (available for diagnostics) |
| 010 | EnableRun | Format RAM has been written, and wakeups are enabled |
| 011 | DebugMode | Controller has been placed in debug mode |
| 012 | RdOnlyBlock' | The controller is processing a block in normal read mode |
| 013 | WriteBlock' | The controller is processing a block in write mode |
| 014 | CheckBlock' | The controller is processing a block in read and check mode |
| 015 | Active | The controller is processing a command for the current sector |
| 016:017 | Select.0..1 | The address of the currently selected drive unit |

KSTAT various bits indicating the status of the drive/controller. The controller will turn on WriteInhibit for the remainder of the sector after any of the following errors are detected, but will still go through all the motions of word transfers.

| | | |
|---|---|---|
| 020 | SeekInc | The disk drive has not correctly positioned the heads within the last 700 ms. A ReZero command must be issued to clear this error. |
| 021 | HeadOvfl | The head address given to the disk drive is invalid (i.e., greater than 4 for a T-80 drive). |
| 022 | DevCheck | One of the following errors occurred:<br>Head select, Cylinder select, or Write command and disk not ready<br>Illegal cylinder address.<br>Offset active and cylinder select command.<br>Read-Only and Write.<br>Certain errors during writing, such as more than one head selected, no transitions of encoded data or heads more than 80 micro-inches off cylinder.<br>A ReZero command may be necessary to clear this error. |
| 023 | NotSelected | The selected drive is in "off-line" test mode or the selected drive is not powered up |
| 024 | NotOnLine | The drive is in test mode or the heads are not loaded |
| 025 | NotReady | There is a cylinder seek in progress or the heads are not loaded |
| 026 | SectorOvfl | The controller detected that a command was active when the next sector pulse occurred. This error implies either a hardware malfunction or a discrepancy between the sector format of the drive and the word count the program thinks is appropriate. |
| 027 | FifoUnderflow | Either the FIFO became empty while writing (task got behind) or the FIFO had too many words taken out of it while readng (microcode word count or wakeup error). |
| 030 | FifoOverflow | Either the FIFO became full while reading (task got behind) or the FIFO had too many words put into it during writing (microcode word count or wakeup error). |
| 031 | ReadDataErr | A flip-flop in the controller for latching one of three errors:<br>CompareErr — a read-and-check operation was executed on a block, and the microcode did not issue ClearCompareErr before the beginning of the next block.<br>ECCError — the microcode can set the ReadDataErr flag if it determines that the ECC words after reading one block are non-zero in order to inhibit future writes.<br>ECCComputeErr — The ECC hardware within the disk controller failed to generate a single "1" bit (i.e., a hardware malfunction). |
| 032 | ReadOnly | The "Read-Only" switch on the drive is on. |
| 033 | CylinderOffset | The cylinder position is currently offset. This is a mode used for recovery of bad data. |
| 034 | IOBParityErr | The controller detected bad parity on the IOB bus. |
| 035 | FifoParityErr | The controller detected bad parity on the data out of the FIFO. |
| 036 | WriteErr | OR of errors on muffler addresses 020-035 |
| 037 | ReadErr | OR of errors on muffler addresses 020-031 and 034-035 |

## KRAM contents of the format RAM

| | |
|---|---|
| 040:043 | Address of format RAM word |
| 044:057 | contents of format RAM word |

## KTAG contents of the tag register

| | |
|---|---|
| 060:077 | 20 bit value last loaded into the tag register |

## KFIFO state of the io control logic

| | | |
|---|---|---|
| 100 | ShiftIn | The controller is currently shifting data into the FIFO |
| 101 | ShiftOut | The controller is currently shifting data out of the FIFO |
| 102 | ComputeECC | The controller is currently shifting data and computing the ECC checksum |
| 103 | NextBlock | Occurs between blocks within a sector |
| 104 | LoadTag | Indicates that the next word read from the format RAM should be loaded into the tag register as opposed to the count register |
| 105 | CntDone' | Indicates that the count register is again zero, and a new value from the format RAM will be loaded next |
| 106 | OutRegFull | The holding register on the input to the FIFO has been loaded, but not transferred into the FIFO. |
| 107 | InRegFull | The holding register out of the FIFO has been loaded, but not read via Pd_Input or loaded into the output shift register. |
| 110:113 | FifoWaddr | The 4-bit address indicating where the next word will be written into the FIFO |
| 114:117 | FifoRaddr | The 4-bit address indicating where the next word will be read from the FIFO. if FifoWaddr equals FifoRaddr then the FIFO is defined as empty. |

### Error Detection and Correction

To allow high data density and a few surface imperfections during manufacture, Trident disk packs are not required to be perfect. A disk pack is defined as suitable when no more than three bad areas occur on any data surface; a bad area is defined as one which could potentially cause read errors of no more than 11 bits in length. To correct errors arising from these imperfections as well as other (infrequent) read errors, the controller implements an error detection and correction scheme which will detect (with very high probability) errors of any length, and will allow correction of any burst error of 11 bits or less.

> **Warning**: If an error burst longer than 11 bits occurs, there is a significant possibility that the error correction algorithm detailed below will fail and double the number of bad bits! Consequently, disk handling programs should try other methods of error recovery before invoking the error-correction algorithm.
>
> To avoid problems, it is good practice to run diagnostic programs on new disk packs; note bad sectors and don't use these during normal operation.
>
> When an error does occur, the first step is to try rereading the offending sector several times. One of these reads may succeed. If not, try rereading with the cylinder position offset or with the data strobe early or late as discussed in the "Tag Register" section. If these attempts all fail, then try error correction.

Error correction is accomplished through a mixture of disk controller hardware (for ECC generation and checking) and system software/firmware (for error recovery). This is a compromise between capability, speed, and cost. The basic capabilities and restrictions of the 32-check-bit scheme are summarized below.

1) A *single* error burst of length less than 12 data bits (i.e., a scattering of error bits within the bit stream, all of which fit within an 11-bit span) can be corrected in blocks shorter than 2685 data words. (Example: for the data "000 1100101", the data "000 0101101" contains a single burst error of length 4.). The code implemented will detect errors in arbitrarily long blocks, but not enough information exists to correct longer blocks.

2) Simple error detection two words are returned by the hardware which are both zero if the read is successful.

3) Software/firmware error correction can be completed in less than one disk revolution. The correction procedure is well suited to a mixture of software and firmware. If done entirely in firmware, error correction would take less than 1 ms.

4) Not all uncorrectable errors will be detected as such. An uncorrectable error requires two bad spots on the disk surface within one sector (the pack is bad throw it out!), an electronic error in a sector with a bad spot, or two electronic errors within one sector. If such an error has occurred, it can, with a probability of say 20 percent, result in an error pattern and displacement that seems valid. This will result in leaving the error bits uncorrected and changing some bits which were in fact correct. This means that for high data security, a check code should be generated and imbedded as part of the data file before writing on the disk.

The error-correcting code (ECC) generated is referred to as a Fire Code (see *Error-Correcting Codes* by Peterson). The following is a detailed description of this code and recovery procedure.

The code calls for dividing the outgoing data stream by a polynomial of the form:

$$P(X) = P_1(X)(X^m + 1)$$

Where $P_1(X)$ is an irreducible polynomial of degree n (n = burst length) and m is > 2*n. For this particular application the polynomials chosen are:

$$P(X) = (X^{11} + X^2 + 1)(X^{21} + 1)$$

During a write, the two polynomials are multiplied together and implemented by hardware in the form:

$$P(X) = X^{32} + X^{23} + x^{21} + X^{11} + X^2 + 1$$

The data stream is premultiplied by $X^{32}$ to make room for the 2 word ECC and then reduced modulo P(X). This is accomplished by the normal feedback shift register technique with the difference that to perform premultiplication, the output of the register is exclusive-or'd with the incoming data and then fed back. After all data bits have been shifted out, the contents of the ECC shift registers are appended to the disk block.

During a read, the feedback shift register is reconfigured such that the two original polynomials are implemented separately. The incoming data stream, including the 2 appended words of ECC, is independently reduced modulo $P_0(X)$ and $P_1(X)$, where

$$P_0(X) = X^{21} + 1$$

$$P_1(X) = X^{11} + X^2 + 1$$

After reading in all words off the disk, the contents of the two polynomial shift registers are read out of the FIFO. If the data is recovered without error, then reducing it modulo $P_0(X)$ and $P_1(X)$ results in the registers containing all zeroes.

If the data contains an error, then the two registers will be non-zero. If one but not both registers is non-zero, then the error is irrecoverable.

To recover from an error, a procedure is undertaken which determines the pattern of bits which are in error, and the displacement of this pattern from the end of the record. I am simply going to present the magic equation to be solved, and some magic constants to be used for solving this equation. Much of the polynomial implementation and the equations, which use the "Chinese Remainder Theorem" are discussed in technical reports from CALCOMP (Calcomp Technical Report TR-1035-04, by Wesley Gee and David George) and XEROX (Xerox XDS preliminary report "Error Correction Code for the R.M. Subsystem," by Greg Tsilikas, 28 March 1972.).

The basic equation is:

$$D = Q*LCM \quad (A_0*M_0*S_0 + A_1*M_1*S_1)$$

where:

$E_i$ = modulus of the polynomial
LCM = least common multiple of $E_0$ and $E_1$
$M_i$ = LCM/$E_i$
$A_i$ = a constant such that $A_i*M_i$ modulo $E_i$ = 1
Q = smallest integer to make D positive
$S_i$ = number of shift operations to the appropriate polynomial remainders as described below.
D = displacement of right-most incorrect bit from the end of the record.

The values of $E_0$ and $E_1$ were found by programming the procedure outlined in the CALCOMP report, and yielded the following result:

$$E_0 = 21 \qquad E_1 = 2047$$

The least common multiple (LCM) of $E_0$ and $E_1$ is simply the product of $E_0$ and $E_1$ since the two numbers have no factors in common. Thus the LCM, which is also the record length which can be corrected, is 42,987 bits, or 2686 2 words.

Knowing LCM and $E_0$ and $E_1$, the values of $M_0$ and $M_1$ are easily found to be

$$M_0 = 2047 \qquad M_1 = 21$$

The values of $A_0$ and $A_1$ are next determined using a trial and error approach that I put in a small program. The results can easily be confirmed, and are given below:

$A_0 = 19 \qquad A_1 = 195$

All of the above values derived so far are constants determined for the particular polynomials chosen. The values of $S_0$ and $S_1$ are determined in the software from the error patterns returned at the end of a disk transfer.

$S_0$ is first determined by a software procedure using the following steps:

1) The remainder from dividing the input data by $X^{21} + 1$ is found in ECC[11:31]; if this remainder is zero, then the error is uncorrectable.

2) Test the low order 10 bits for all zeroes, and if not then perform a left circular shift on the 21 bits. When the low order 10 bits are all zeroes, the error pattern is in the upper 11 bits of the word, and $S_0$ is the number of times the circular shift was performed.

3) If the low order 10 bits don't become all zeroes within 20 shifts (1 full cycle), the error is uncorrectable.

$S_1$ is then determined in microcode as follows:

1) The remainder from dividing the input data by $X^{11} + X^2 + 1$ is found in ECC[0:10]; if this remainder is zero, then the error is uncorrectable.

2) Test this number to see if it is equal to the error pattern determined in step 3 of $S_0$, and if not reduce this number modulo $X^{11} + X^2 + 1$ (left shift and XOR feedback). When the contents of this word equals the error pattern (it is guaranteed to happen before 2047 reductions), $S_1$ is determined as the number of reductions performed (In the hardware implementation of switching from the write polynomial to the read polynomials, it was easier to implement a polynomial that premultiplied by $X^{11}$. This means that the remainder returned by the hardware already has had 11 shifts performed. To compensate, when $S_1$ has been determined by the above procedure, you must add 11 to the value, and subtract 2047 if the result is greater than or equal to 2047.).

The basic equation for the displacement now looks like

$$D = Q*42{,}987 \quad 19*2047*S_0 \quad 195*21*S_1$$

where:

$0 \leq S_0 \leq 20$

$0 \leq S_1 \leq 2046$

Notice that the straightforward solution to this equation cannot be done with single-precision arithmetic on the Dorado; to avoid double precision, the following manipulation of the equations is useful:

$D = Q*2047*21 \quad 19*2047*S_0 \quad 4095*S_1$
$D = Q*2047*21 \quad 19*2047*S_0 \quad 2*2047*S_1 \quad S_1$
$D' = Q*21 \quad 19*S_0 \quad 2*S_1$

where:

$0 \leq D' \leq 20$
$D = 2047*D' \quad S_1$ (add 42,987 if D' = 0)

*For some reason that we don't understand, the actual required calculation must be* $D = 2047*(D'+1) \quad S_1$ *in the last step. Also D' is conveniently calculated as* $(215*21 \quad 19*S_0 \quad 2*S_1)$ *rem 21.*

# Display Controller

The Dorado Display Controller (DDC) uses the fast io system to obtain representations of video images from storage; it then transforms these representations into control signals for monitors. Its three design objectives are:

(1) To handle a variety of color, grey-level, and binary (black-and-white) monitors;
(2) To utilize the full power of the fast io system in producing high-bandwidth computer graphics;
(3) To allow various compromises in color and spatio-temporal resolution for experimental purposes. Clock rates, video signals, and other monitor waveforms should be controllable by firmware.

There are *two* independent video channels capable of running in a variety of modes. Two channels allow text to be displayed on one channel, graphics on another, or the main picture on one, cursor on the other.

The DDC must readily handle Alto-style and LF (large format) monitors which we expect to be standard for most systems. Bit maps, display control blocks, and monitor control blocks, similar to those used on the Alto, provide the software interface to the DDC. The "seven-wire" video interface makes provision for one or more low bandwidth input devices (keyboard, pointing device, etc.); our current provisions for keyboard and mouse input are also discussed in this chapter.

*Keep Figure 14 in view while reading this chapter.*

## Operational Overview

Video scan lines are encoded in *bitmaps*, which are contiguous blocks of virtual memory; the two channels, A and B, have independent bitmaps and data paths in the DDC. The high-priority *DWT* (Display Word Task) runs on behalf of either A or B using the subtask mechanism; it transmits each bitmap to a FIFO consisting of 15 munches/channel. The bitmap stream emerging from the FIFO is then sorted into *items* (1, 2, 4, or 8 bits wide) for each channel which are combined, mapped, and transformed into *pixels* (picture cells) on the screen.

In addition to the two channels, the DDC supports a programmable cursor that is 16 pixels x 1 bit/pixel wide.

A lower priority *DHT* (Display Horizontal Task) handles horizontal and vertical retrace and sets up starting addresses and munch counts, cursor data, and formatting information in the *NLCB* (Next Line Control Block) for the DDC. The NLCB is then copied into the *CLCB* (Current Line Control Block) during horizontal retrace prior to the next scan line.

The rate-of-flow of items is governed by the *resolution* and *pixel clock period*. Resolution may be independently programmed for each channel so that items flow at 1/4, 1/2, or 1 times the pixel clock period. If the DispM board is present, then the pixel clock period is also progammable; otherwise, it is determined by a crystal oscillator on the DispY board, which must have a frequency appropriate for the monitor being driven.

Items can be treated in one of three ways: First, an Alto monitor can be driven. Second, items can be mapped through the 256-word x 4-bit *MiniMixer* into video data for a black-and-white or grey-level monitor.

> Three separate interfaces are provided on the DispY board. An Alto monitor interface ORs one-bit items from the A and B channels with the cursor, and then XORs by polarity to produce one-bit pixels for an Alto display. A seven-wire interface outputs 1 bit/pixel for a binary monitor. And an 8-bit digital-to-analog converter (DAC) produces grey-level video.

Third, items may be mapped by the *Mixer* (or A color map), a 1024-word x 24-bit RAM, into signals for a color or grey-level monitor. A variety of modes determine which bits from the A and B items address the mixer. Mixer output consisting of 8 bits for each of the red, green, and blue guns is then digital-to-analog converted for color monitors. Additionally, there is a 24-bit/pixel mode in which the Dorado supplies 8 bits for each of the three colors; the colors are independently mapped through the Mixer and two additional 256-word x 8-bit RAMs called the BMap and the CMap.

The DDC is implemented on two Dorado main logic boards, called DispY and DispM. DispY contains all the logic necessary for vertical and horizontal sweep control, channel data paths, and video data for binary and grey-level monitors running at a fixed pixel clock rate. DispM contains the color maps, the programmable pixel clock, and the three DACs for driving a color monitor. Additionally, DispM contains an independent terminal controller that is structurally similar to a one-channel, one bit/pixel DispY but is specialized to driving a 7-wire terminal.

Thus there are two principal DDC configurations. On a Dorado with only a 7-wire terminal and no color monitor, only the DispY board is present; it is programmed for Alto terminal emulation, and only a small subset of its capabilities are used. However, on a Dorado with both a 7-wire terminal and a color monitor, the DispM board is also present; all of DispY and the color hardware on DispM are used to drive the color monitor, and the independent controller on DispM is used to drive the 7-wire terminal.

## Video Data Path

### *Fast IO Interface and FIFO*

The fast io system delivers data to the DDC at a rate of 16 bits/clock; words are received alternately in the REven ($t_1$) and ROdd ($t_2$) registers shown in Figure 14, then written into the FIFO, a 256-word x 32-bit RAM, during the first half of the next Dorado cycle ($t_2$ to $t_3$), leaving the second half of the cycle free for read access by the video channels. In other words, the REven and ROdd registers widen the data path from 16 to 32 bits to allow sufficient time to both write and read the FIFO in one cycle.

The 256 double-words in the FIFO are divided evenly among the two channels, so each has buffer storage for 16 munches. Each channel has write and read pointers that address the FIFO when appropriate.

Write pointers are initialized once during vertical retrace and then sequence through addresses for the entire display field; a write pointer is incremented after each double-word write for its channel, so that the next word to be written is addressed at all times. Since

the fast io system delivers only one munch at a time, there is never any problem in deciding which of the two write pointers should address the FIFO.

Read pointers, however, are initialized during each horizontal retrace, so that the correct first double-word is read at the start of every scan line. This is required because the fast io system always delivers complete munches, but unused double words may appear at the end of the last munch for the previous scan line, or at the beginning of the first munch for the current scan line; the read pointer has to be reinitialized to skip over these. FIFO reads alternate between channels A and B, so the data rate for one channel is limited to 32 bits/2 cycles (=16 bits/cycle).

Note that *bitmaps are required to start at even addresses* because the FIFO is 32 bits wide.

*Item Formation*

At the output end of the FIFO there is a multiplexor shared by both channels and, for each channel, two intermediate buffers (*FIB* and *SIB*), and a shift register *SR*. The multiplexor permutes the 32-bit quantity emerging from the FIFO so that when the double-word has marched through FIB and SIB and is finally loaded into SR, successive shifts will produce successive items of the selected size (8, 4, 2, or 1 bits).

The SR is tapped as follows:

| | |
|---|---|
| SR.0 | Item[0] for item sizes 1, 2, 4, or 8; |
| SR.16 | Item[1] for sizes 2, 4, or 8, gated to 0 for size 1; |
| SR.8, SR.24 | Item[2:3] for sizes 4 or 8, gated to 0 for sizes 1 or 2; |
| SR.4, SR.12, SR.20, SR.28 | Item[4:7] for size 8, gated to 0 for sizes 1, 2, or 4. |

All eight Item bits are gated to 0 if the channel is off. It is useful to think at this point that, regardless of a channel's item size, an 8-bit wide item is produced, whose bits contain non-zero data only in those positions dictated by the item size; i.e., for size 1 only the most significant bit may be non-zero; size 2 allows data in the topmost two bits, etc.

The SR loads on the *item clock* after its last item has been used; the item clock rate is the pixel clock rate divided by the resolution (1, 2, or 4 for full, half, or quarter, respectively). Hence, for 8, 4, 2, or 1-bit items, SR will be shifted 3, 7, 15, or 31 times, repectively, and be reloaded from SIB on the following item clock.

> Synchronization of SR, which uses the item clock, with FIB and SIB, which use the Dorado system clock, is a little tricky. SIB_FIB will occur no later than (4.6 ns)+C+(1.1 ns)+C+C = 3*C+5.7 ns after SR_SIB, where C is the period of the Dorado system clock and 4.6 ns and 1.1 ns are the worst case propagation delay and setup time of the components in the synchronizer; FIB_FIFO will occur at this time or on one of the next three Dorado clocks, depending upon which of these four clocks corresponds to $t_2$ of the cycle in which this channel can read the FIFO. Allowing for propagation delay through SIB (5.0 ns) and setup time for SR (1.7 ns), the worst case minimum spacing between loads of SR is 3*C+(5.7 ns)+(6.7 ns) = 3*C+12.4 ns. This must be less than the time for emptying SR which is I*(32/ItemSize), where I is the period of the item clock. Hence, I > (3*C+12.4)/4 for ItemSize=8, or I > 25.6 ns for a Dorado clock period of C = 30 ns.

The 8-bit items from the two channels are then presented to either the Mixer section on the DispM board or the MiniMixer or Alto video interface on the DispY board.

*Mixer*

The Mixer is controlled by the *A8B2*, *BBypass*, and *24Bit* mode controls. It is a 1024-word x 24-bit RAM for which the 10 bits of address required may be obtained from two possible source distributions, depending upon the A8B2 mode. When A8B2 is true, the address consists of AItem[0:7] and BItem[0:1]; when false (called A6B4), the address is AItem[0:5] and BItem[0:3].

Another mode, the BBypass mode, can be enabled independently for the B channel. If B is bypassed, none of its bits contribute to the Mixer address. Instead, they bypass the mixer and address a 256 x 8 RAM, the BMap, whose outputs are ORed with the mixer outputs for the blue DAC. For example, with ASize=8, BSize=4, BBypass true, and A8B2 true, and with appropriate values in the Mixer RAM, the controller may be thought of as three 4/bits pixel channels driving three color guns. One channel is bypassed data from B, while the other two are mapped through the Mixer.

24Bit mode, used in conjunction with BBypass mode, is used to run a three-channel color display directly from memory. In this mode, items from the A channel alternately address the Mixer (called the AMap in this mode) and another 256 x 8 RAM called the CMap. Meanwhile, the B channel runs at half the A channel rate and addresses the BMap as described above. (That is, the B channel must be set to one-half the resolution of the A channel.) With suitable values in the color maps, the AMap, BMap, and CMap independently generate outputs for the red, blue, and green DACs respectively.

> *Note:* when the A channel is turned on, the first AItem addresses the AMap and the second AItem addresses the CMap. For the A and B pixels to align properly on the display in 24Bit mode, the left margin counts must be set to start the B channel one pixel clock earlier than the A channel. The blue and green portions of the AMap must be entirely zeroed, since the blue and green outputs are ORed with the BMap and CMap.

After routing as dictated by the mixer modes, chosen items are loaded into the map address registers, causing the color maps to produce a new video value every pixel clock (every two pixel clocks in 24Bit mode), and these values are latched in the three 8-bit mixer output registers. Three very fast DAC modules then produce a Red-Green-Blue triple of analog signals for a color monitor, or up to three grey-level video signals. In conjunction with the sync, blank, and composite waveforms produced by the monitor control circuitry, these signals can drive a wide variety of monitors attached to the Dorado.

*Alto Video Interface*

A small circuit on the DispY board produces video for an Alto monitor. This circuit ORs CursorData, AItem[0], and BItem[0], then XORs by the polarity, and finally ORs with the vertical and horizontal blanking signals. *This interface is obsolete and is no longer in active use.*

*MiniMixer*

A small video mixer on the DispY board, not to be confused with the large Mixer on the DispM board, can drive either a DAC or the seven-wire interface discussed later. The MiniMixer is a 256 word x 4-bit RAM addressed by a combination of AItem, BItem, and state bits, as shown in Figure 14. On every pixel clock, dDAC[0:3] are loaded from MiniMixer

output, while dDAC[4:7] are loaded directly from AItem[4:7]. The MiniMixer aims at experiments with mixing channels and driving grey level monitors.

### Horizontal and Vertical Control

Every monitor requires horizontal synchronizing and blanking waveforms. Interlaced monitors must be able to distinguish fractions of a scan line to implement interlacing. In general, the duration and phasing of sync/blank waveforms is unique to a given monitor. The DDC uses the 1024-word x 3-bit *HRam* (Horizontal RAM) to control horizontal sync/blank.

The DDC has a set of registers called the *CLCB* (Current Line Control Block) which controls video generation for the current scan line. The DHT sets up parameters for the next scan line in *NLCB* (Next Line Control Block), a 16-word x 12-bit RAM. The first 32 pixel clocks of horizontal blanking are called the *HWindow*; during HWindow parameters for the next line are copied from NLCB into CLCB. Vertical control is also handled through the NLCB.

The interpretation of fields in NLCB and HRam are shown in Figure 15 and loading will be discussed in the "Slow IO Interface" section; the use of the different information is discussed here. The top part of Figure 14 shows how horizontal timing is controlled.

#### *Line Control Blocks*

The fields in NLCB/CLCB are interpreted as follows, where a denotes that the item is channel-specific (i.e., copies exist for both A and B channels):

a Polarity. A single bit, used only for binary monitors, that inverts black and white (APolarity and BPolarity are or'ed by the hardware).

a Resolution. A 2-bit field that controls item clock generation; values of 0, 2, and 3 cause quarter, half, and full resolution, respectively.

a ItemSize. A 4-bit field unary encoded as aSize1, aSize2, aSize4, or aSize8, denoting bits/pixel for the channel; setting multiple bits is illegal.

a LeftMargin. A 12-bit field in units of pixel clocks specifying 31 less than the number of pixel clocks to wait after HWindow completes before turning the channel on. This value is not a straightforward constant, but depends upon monitor-specific horizontal blanking time. If the horizontal blanking time is B pixel clocks and the desired beginning of data is L pixel clocks after the end of horizontal blanking, then aLeftMargin should be loaded with B+L 32 31 = B+L 63, independent of resolution. Since L may be 0, this implies that the horizontal blanking time for the monitor must be greater than 63 pixel clocks. Since high-speed monitors typically have greater than 4 ms horizontal blanking times, and are this fast only with high speed pixel clocks, this restriction is not expected to be significant.

*Note:* For a monitor connected via the 7-wire interface, aLeftMargin must be B+L 68, rather than B+L 63, because video signals are delayed from horizontal control waveforms by 5 pixel clocks.

*Note:* The value loaded into aLeftMargin must actually be the *negative* of the left margin count computed above.

aWidth. A 12-bit counter that counts at the pixel clock rate as soon as the channel turns on; when the counter runs out (or when horizontal retrace starts, whichever is earliest), the channel is turned off. Precisely, if the channel is to run for W pixel clocks, the width counter must be loaded with (W+255).

aFifoAddr. An 8-bit quantity pointing to the munch and word within the munch for the first FIFO read for the next scan line; this must be an even number because doublewords are fetched from the FIFO. Firmware must keep track of the number of used munches for any given line and advance aFifoAddr by exactly the right amount, adjusting for munch boundaries, interlacing, and data breakage. The CLCB register for aFifoAddr is the channel read pointer itself.

MixerModes. A set of bits that control the mixer; these are *not* channel-specific. These will normally be changed infrequently, maybe at the field rate or during display initialization. However, they are in the NLCB to allow modes to change on the fly.

Vertical Control Word (VCW). A word controlling the vertical retrace operation of the monitor; it contains the vertical blank bit, vertical sync bit, and interlace field bit discussed in the "Vertical Waveform Generator" section below.

Cursor and CursorX. The 12-bit CursorX value is loaded into a counter which starts counting at the end of HWindow. When the counter runs out, the 16-bit Cursor value is shifted out onto the CursorVideo line. This is used by the Alto video interface and in the MiniMixer address. Precisely, if horizontal blanking is B pixels in duration, and the leftmost bit of the cursor is to appear X pixels beyond the end of horizontal blanking, then the CursorX register must be loaded with (B+X+226), or (B+X+221) when using the 7-wire interface.

*Horizontal Waveform Generator*

The 1024-word x 3-bit HRam contains control information for these waveforms. Under normal operation, HRam is addressed by a 12-bit counter (HRamAddr[0:11]) which is reset at the leading edge of horizontal sync and then increments every pixel clock until the next leading edge of horizontal sync; HRamAddr[1:10] address the RAM, and the output is loaded into the HRamOut register every other pixel clock. The three bits in HRamOut control horizontal sync, horizontal blank, and half-line; these three bits are combined and level shifted by a logic network appropriate for the monitor being driven.

The 1024-word HRam imposes the uninteresting restriction that there be fewer than 2048 pixels/scan line.

As shown in the diagram at the top of Figure 14, horizontal blanking (HBlank) is true from the end of one scan line to the beginning of the next. During horizontal blanking, HSync is turned on to initiate the horizontal retrace and turned off again when horizontal retrace is

finished. HBlank then continues for a monitor-specific interval. Note that if a channel's visible left margin is non-zero, then the horizontal scan will begin before that channel is producing any data; in this case, the video channel outputs zero items to the mixing stages until the channel is turned on.

> Due to an implementation error, when the 7-wire interface is being driven from DispY, the value of HBlank[*i*] may differ from HBlank[*i* 1] only when *i* is even, where *i* is HRamAddr[1:10].

*Vertical Waveform Generator*

Only 2:1 interlaced monitors are supported in this design, but more complicated vertical control could be provided, if desired. To support 2:1 interlace, HRam contains a waveform called HalfLine, which is a pulse at the horizontal line frequency, $180^{o}$ out of phase with HSync.

Vertical control is handled by DHT through the NVCW word in the NLCB, which specifies whether or not vertical blank or retrace should begin or end during the next scan line. The DHT microcode must keep track of scan lines to enable vertical signals at the appropriate times.

The three VCW bits are called *VBlank*, *VSync*, and *OddField*. VSync enables vertical sync to begin on the next line, and the OddField bit chooses either HSync or HalfLine on which to do vertical syncing (OddField=1 implies HalfLine phasing for vertical sync). This phase will alternate from the start of the line to the middle of the line and back for successive fields. The blanking signal for the monitor is VBlank ORed with HBlank.

## Pixel Clock System

The programmable pixel clock on the DispM board, if present, determines the fundamental video data rate for a given monitor. The pixel clock is controlled by loading the PixelClk register via the slow io system. The pixel clock frequency is (312.5*(241 M))/(16 D) KHz, where M is PixelClk[4:11] and D is PixelClk[12:15]. Note that the pixel clock will not stabilize until about 1/2 second after the PixelClk register is loaded.

The parts of the DDC synchronized to the rest of Dorado do, of course, use the Dorado system clock. As discussed earlier, the synchronization logic for refilling SIB after SR_SIB puts a lower bound on the pixel clock period of (3*C+12.4)/4 ns (= 25.6 ns for a Dorado clock period of C = 30 ns), for an item size of 8 on either channel. We anticipate that pixel clock rates in the range 10 to 50 MHz (100 to 20 ns/pixel) will be required, so the lower bound is approximately consistent with this.

**Seven-Wire Video Interface**

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

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

*Video Output*

The four output lines are interpreted as either a 4-bit nibble of video or four control signals according to the phases of the two clock signals; the DDC places data on the data lines at the falling edge of ClkA, and the terminal samples this data on the rising edge of ClkA. If ClkB is 1 at this time, the nibble is interpreted as four bits of video, else as sync and control information. ClkA and ClkB are transmitted in quadrature so that the terminal can reconstitute a clock at the video bit rate.

When a nibble is interpreted as control information, bit 2 is reserved for horizontal sync and bit 3 for vertical sync, while 0:1 are undefined; different types of terminals may use 0:1 for any purpose.

A circuit on the DispY board drives the seven-wire interface from the MiniMixer. MinMixer[0] is serial-to-parallel converted into four-bit nibbles, which are held in a register for transmission. Sync, blank, and clock phases are generated in accordance with the seven-wire interface specification.

*Back Channel*

Data from low bandwidth input devices at the terminal are transmitted serially over the back channel. Data are clocked by the terminal on the rising edge of the horizontal blank pulse and are sampled by DHT during the subsequent scan line after HWindow.

By convention the terminal microcomputer encodes 32-bit messages (delivered in 32 scan lines); each message begins with a 1, and after the 32nd bit of the message the DHT ignores the backchannel until the start of another message is indicated by another 1. The message consists of a start bit, 3 unused bits, a 4-bit message type, a 16-bit message body, and finally an 8-bit trailer which must be $200_8$.

The terminal microcomputer perpetually cycles through all possible keys on the keyboard (as well as mouse buttons and keyset paddles), detecting changes in state of the keys; the state of the keyboard then exists in seven 16-bit words, and a back channel message is defined for each. Whenever one of these words changes value, it is sent to the Dorado in a message. Additionally, changes in mouse x,y coordinates are reported once per field (i.e., twice/frame or typically 60 times/sec). If the mouse has not changed position during

a field, then one keyboard word is reported instead of the mouse position change; thus, the correct state of the keyboard is eventually reported even if transitions are missed.

**Table 24: Terminal Microcomputer Messages**

| *Message Type* | *Comments* |
|---|---|
| 00B | Illegal ignored |
| 01B | Keyboard word 0 (corresponds to Alto memory location 1077034B) |
| 02B | Keyboard word 1 (Alto 177035B) |
| 03B | Keyboard word 2 (Alto 177036B) |
| 04B | Keyboard word 3 (Alto 177037B) |
| 05B | Mouse buttons and keyset (Alto 177033B) |
| 06B | 8-bit changes in X-coordinate (0:7 of the message body) and Y-coordinate (8:15 of the message body), represented in excess-200B notation |
| 07B | Illegal ignored |
| 10B | Keyboard word 4 (Star keyboards only; no Alto analogue) |
| 11B | Keyboard word 5 (Star) |
| 12B 16B | Illegal ignored |
| 17B | Boot message. Actually, depressing the boot button jams the data to one continuously, rather than generating a valid terminal message. Furthermore, when the boot button is let up, there may be as many as 8 bits of garbage following the last consecutive one bit; these must be ignored by the firmware. The firmware should also ignore boot button pushes less than 10 ms in duration, as these may be caused by noise or contact bounce. |

### Processor Task Management

This section outlines the implementation requirements of DHT and DWT and discusses the hardware associated with task wakeups and DWT subtask arbitration between the two channels.

Since DHT must do a lot of processing, it runs at low priority and is awakened once/scan line at the end of HWindow. When it runs, it must calculate all parameters for the *next* scan line (i.e., the one after the scan line that is just starting), load the NLCB appropriately for each channel, and set up the munch address and count for each channel in the RM registers aNextAddr and aNextCount referred to in the DWT sample code below; then it sets the aNextWCBFlag flags discussed below. The DHT wakeup will remain active until any NLCB output command is executed, so the DHT must execute at least one NLCB output command every time it wakes up, and this must occur at least three instructions prior to blocking.

DWT is a very high priority task which may run on behalf of either channel: channel A is subtask 0; channel B, subtask 2. Since it uses the subtask mechanism, DWT must always block at the same instruction each iteration. DWT does not explicitly know the channel for which it is executing at any given time; its two parameters, a start address and munch count, are received from DHT in RM registers specific to the subtask. In the normal case, DWT initates an IOFetch and blocks. The following is the main-line DWT microcode presently in use:

```
%RM registers for channel A, indicated by names beginning with "A" below, are used in the program, but the
corresponding set of registers for channel B, in a different RM region, will be referenced when SubTask is 2.

Note that TIOA selects the DWTFlag register and T contains 20 at the beginning of the loop, so the second
instruction is used both to increment the munch address and to signal the hardware that an IOFetch is
commencing.
%
DWTStart:       ACount_(ACount) T, Branch[DWTCheck, R<0];
                AAddress_(IOFetch_AAddress)+(Output_T), Block, Branch[DWTStart];

%AAddress will be even if we just exhausted a scan line.  AAddress will be odd if we have just been awakened to
start a new scan line.  In either case, isolate flag in AAddress[15] for use in adjusting the WCB flags.
%
DWTCheck:       AAddress_ (AAddress) AND (1C), Branch[DWTAdjustWCBFlags, R even];
%Note that the change-RSTK-for write function used below is ok, but the change-RBase-for-write functions are
illegal because of subtasking.
%
DWTRefill:      ACount_ANextCount;      *from DHT, # munches to fetch -1 in 0:11
                BrLo_ANextAddrLo;       *first munch address
                BrHi_ANextAddrHi;

%Now adjust WCB flags, as follows: If we just exhausted a scan line, AAddress=0 now; execute Output_0 to
clear the CurWCB flag, and set AAddress to  1 for the next wakeup.  If we are starting a new scan line,
AAddress=1 now; execute Output_1 to set the CurWCB flag and clear the NextWCB flag, and set AAddress to 0
for the first IOFetch.
%
DWTAdjustWCBFlags:
                AAddress_ (AAddress) 1, Output_AAddress, Block, Branch[DWTStart];
```

DWT lowers its wakeup request at the onset of the DWTStart instruction, and the DDC remembers that DWT is in progress. No further DWT wakeups will be generated while the task is running or is preempted by a higher priority task. Whenever DWT blocks, a counter is initialized to a constant value N and counts once per Dorado cycle; when the counter runs out, DWT wakeups are allowed again. This counter has two purposes. First, within a munch loop it spaces out IOFetch references to the memory system by 8 or more cycles (depending upon N, which is adjustable through a hardware SIP component), so as not to clog the memory pipeline. Second, the decision to generate subsequent DWT wakeups is based upon the state of flags that may be altered by output commands; these commands take time to get from the processor to the DDC and alter the state. Other tasks may have the processor while these state changes take effect.

After N cycles have elapsed, DWT will be woken whenever aWantsDWT is true for one of the channels. Two channel-specific flags are involved in DWT wakeup control: aCurrentWCBFlag is true when a is actively moving words into the FIFO; aNextWCBFlag is set true by DHT after it has loaded the munch address and munch count into DWTnextaddr and DWTnextcount for a. After fetching the last munch for a scan line, DWT clears aCurrentWCBFlag and blocks unless aNextWCBFlag is true. In other words, aWantsDWT when

(aNextWCBFlag & not aCurrentWCBFlag) %
(aCurrentWCBFlag & aFifoAvailable).

If only AWantsDWT or only BWantsDWT, no conflict arises and the requesting channel gets DWT. However, if both channels want DWT, the channel that ran least recently will run next.

Two observations must be made about the DWT microcode. First, because the final instruction is normally an IOFetch_, the next instruction executed (by another task) will be held one cycle if it initiates any memory reference. Secondly, the two instruction loop above requires that the hardware cope with the NextLies condition discussed in the "Slow IO" chapter; a pathological lockout problem could occur if a high demand task of higher priority is coded so that it always creates NextLies (say, by doing Block and immediate _Md in the instruction after a fetch). This would result in the DWT wakeup being frequently delayed by 2 cycles.

*Note:* Neither DWT nor DHT drives the IOAtten branch condition.

**Slow IO Interface**

DDC manages all control functions via the slow io system. At this point you should study Figure 15, which shows the format of the various output and input commands; there are six output devices and one input device on the DispY board, and eight output devices and one input device on the DispM board (if present). Output commands are handled uniformly: TIOA is clocked into a register at $t_1$; the register output is decoded and identified as one of the DDC commands; if the processor is doing an Output_B, then at $t_3$ IOB data from the processor is clocked into a register and one of the "TIOA command" pulses occurs from $t_3$ to $t_5$, at which point the desired action is complete.

The IOB data received at $t_3$ of an Output_B will remain in the DDC buffer register (RIOB) until the next output command. This is useful for debugging and for muffler readout of the NLCB (because an NLCB address can be loaded into RIOB for multiple cycles).

The HRam, MiniMixer, Mixer, BMap, and CMap are RAMs that will generally be loaded during system initialization and not often changed while pictures are being displayed. The programmable pixel clock will also be loaded during initialization, if it is being used instead of the fixed crystal oscillator.

The HRam, Mixer (AMap), BMap, and CMap addresses each have two independent sources: the Dorado slow io system and the video system. Video system addressing is disabled during loading from the Dorado. The output commands to each of these RAMs are interpreted as follows: The Keep' bit is saved in a flipflop loaded by every RAM output command; as long as Keep' is true (i.e., low), video system addressing is off. If LoadAddr is true, then IOB[4:15] are loaded into the RAM address register. If Write' is true (i.e., low), the currently-addressed word of the RAM is written from the data field; additionally, the RAM address register increments after writing, so the RAM can be loaded sequentially at high speed. A RAM output command with Keep' false (i.e., high) releases the RAM from Dorado control and returns it to the video system.

> *Note:* the LoadAddress and Write' bits of a RAM output command take effect only if the Keep' flipflop is *already* true (i.e., set to zero by a *previous* RAM output command).
>
> *Note:* in the case of the Mixer, the RAM address is loaded from IOB[4:14] and a Hi/Lo Select bit is loaded from IOB[15]. The latter bit determines which 12 bits of the 24-bit wide mixer word will be loaded by the next Write'. The Hi/Lo Select bit behaves as a low-order extension of the Mixer address counter, so successive Write' commands will alternate between the halves of one mixer word before advancing to the next.

The MiniMixer is loaded by a single output instruction that specifies both the address and data to be loaded. During the command pulse from $t_3$ to $t_5$ of the Output_B instruction, the video channel address to the MiniMixer is replaced by the address being loaded, so if the video channel is active, garbage may appear at the output during this cycle.

The 16-word x 12-bit NLCB is also loaded by single output instructions that specify both the address and data. For the NLCB, output instructions are only effective when HWindow is not occurring during HWindow the RAM address is supplied by a counter that successively copies the NLCB words into CLCB. The format of each of the words in NLCB is shown in Figure 15. Note that any NLCB output operation will dismiss the wakeup request for DHT, and DHT must not block any sooner than the fourth instruction after the first NLCB output operation is issued.

The Statics output command is used for debugging and initialization. Two bits in the Statics register called DHTShutUp and DWTShutUp are discussed in the "DDC Initialization Requirements" section below. Three other fields called *FakePClk*, *UseFakePClk*, and *MufAddr* are used for debugging. When UseFakePClk is true, the regular pixel clock is degated; if FakePClk is true, then a pixel clock will occur at $t_5$ of the Statics output command; otherwise no clock occurs. Every Statics command also loads the hardware signal addressed by MufAddr into a flipflop (at $t_5$) which can be read by the Status input command discussed below. In combination, the fake pixel clock and muffler readout features allow diagnostic firmware to checkout most of the internal data paths in the DDC by simulating a very slow pixel clock and "stepping" the DDC through various states, the diagnostic can check nearly all of the data paths between fake pixel clocks. The hardware signals selected by MufAddr[5:11] are given in the table below.

**Table 25: DDC Muffler Signals**

| *MufAddr* | *Signal* | *MufAddr* | *Signal* |
|---|---|---|---|
| 0 | ACurrentWCBFlag | 70 | AFifoFull |
| 01:07 | AReaderPtr[1:7] | 71 | BFifoFull |
| 10 | ANextWCBFlag | 72 | ASize8 |
| 11:17 | AWriterPtr[1:7] | 73 | ASize8-4 |
| 20 | BCurrentWCBFlag | 74 | ASize8-4-2 |
| 21:27 | BReaderPtr[1:7] | 75 | BSize8 |
| 30 | BNextWCBFlag | 76 | BSize8-4 |
| 31:37 | BWriterPtr[1:7] | 77 | BSize8-4-2 |
| 40:47 | AItem[0:7] | 100 | AOn |
| 50:57 | BItem[0:7] | 101 | BOn |
| 60:63 | AServicePtr[1:4] | 102:103 | ARes[0:1] |
| 64:67 | BServicePtr[1:4] | 104:105 | BRes[0:1] |
| | | 106 | MonitorType |

Muffler 106 (MonitorType) is the only one of interest during normal operation. It identifies the type of monitor connected via the 7-wire interface: zero denotes an Alto-style monitor; one denotes an LF (large format) monitor.

A single input device called Status is implemented. It is used to return the currently selected muffler bit and the seven-wire interface received data bit.

The MapInLo and MapInHi input devices read the current values output from the color maps (Mixer, BMap, and CMap, whichever are active). When the color maps are controlled by the video system, these outputs change too rapidly for reading them to be useful (unless the DDC is being single-stepped by means of *UseFakePixelClk*). However, when the color maps are controlled by the Dorado, this input device can be used to read out the color map entries addressed by their respective RAM address registers.

MapInHi[0] is the 7-wire terminal input bit for the independent terminal interface on DispM; its position corresponds to Status[0] on DispY (see below). MapInHi[1] is a constant 1 if a DispM board is installed; if DispM is not installed, an Input from the nonexistent register yields a zero value. This enables firmware to detect the presence or absence of a DispM board. MapInLo[0:3] are a 4-bit color monitor type jumpered on the Dorado backpanel.

> *Note:* the MapInLo and MapInHi input devices do not generate IOB parity, so they must be read by the Pd_InputNoPE function to disable parity checking.

**DispM Terminal Interface**

The independent terminal interface on the DispM board functions much the same as a single-channel DispY board, but is specialized to driving a binary monitor via a 7-wire interface. The data path is one bit/pixel; the resolution is full; there is no MiniMixer; and the horizontal waveforms are fixed by a PRom (which must be changed when a different type of 7-wire terminal is installed).

Aside from these limitations, the DispM terminal interface operates almost identically to the A channel of DispY. In particular, the io addresses are grouped parallel to the ones on DispY, and the data formats are identical; so a microprogram can initialize TIOA to the correct group and subsequently use the function that changes only TIOA[5:7] to select registers within that group. This enables practically all the microcode for driving a 7-wire terminal to be shared between DispY and DispM.

In Figure 15, the DispY io operations that are also defined for DispM are marked with an asterisk. Note that outputs to unused NLCB addresses are ignored.

Due to hardware differences between DispY and DispM, the ALeftMargin and CursorX values must be computed slightly differently. For DispM driving the 7-wire interface, ALeftMargin = (B+L 130) and CursorX = (B+X+190).

> *Note:* DispM does not have a muffler system. In particular, the MonitorType muffler value is always read from DispY. By convention, this refers to the type of 7-wire terminal attached to the Dorado, whether that terminal is connected to DispY or to DispM. Also by convention, the 7-wire terminal is always connected to DispM if DispM is installed.

### DDC Initialization Requirements

The two low-order bits in the Statics register are called DWTShutUp and DHTShutUp. They are forced true by IOReset and prevent the respective task wakeups from happening. They are individually set or cleared by the Statics output command. In addition, IOReset sets the signal DoradoHasHRam; this will prevent horizontal sync from being sent to monitors until the HRam has been loaded and released by firmware. Blanking is sent to monitors as long as DHTShutUp remains true. It is anticipated that DHTShutUp will be left true until all DDC initialization has been completed by the emulator (or by the DHT running in response to a Notify).

Some other initialization requirements are as follows: aLeftMargin should be loaded with a large negative value in case one of the channels remains unused forever; the Cursor in NLCB should be zeroed in case the cursor is completely off-screen forever; HRam must be loaded with monitor-specific waveforms; the pixel clock rate must be set; mixer modes must be set; the MiniMixer must be loaded. In addition, the DHT must explicitly set the aAddress registers to zero on behalf of the DWT, which cannot initialize itself completely for each subtask.

### Speed and Resolution Limits

High performance color monitors are typified by the following performance limits:

| | |
|---|---|
| 22 ms | horizontal scan time |
| 5 ms | horizontal blanking time |
| 800 ms | vertical blanking time |

Parameters for a particular monitor can be modified slightly through hardware adjustments, but cannot be controlled by the DDC, which must provide control signals with timing appropriate for the monitor. Consequently, a monitor must be chosen that conforms to the speed limitations of the DDC.

One important speed limitation is how fast bits can be moved from storage through the DDC. This limit is derived using the following parameters:

F Frame update rate. High speed phosphors require a minimum update rate of 30 frames/sec with interlaced operation for reasonable visual effects; this is marginal and faster update is desirable.

S Scan lines/frame.

VR Vertical retrace time; with interlaced operation, there will be two vertical retraces/frame.

HB Horizontal blanking time.

HS Horizontal scan time. The FIFO must not go empty during the horizontal scan or garbage will be displayed.

T Time/munch or the rate at which storage can deliver data for IOFetches; this is 1 munch/8 cycles = 1 munch/0.4 ms.

M Munches/scan line that the fast io system can deliver.

The time required to fill the FIFO for both channels is a little longer than 30*8+20 cycles (= 276 cycles) or about 13.8 ms at a Dorado clock period of 25 ns; this follows from the fact that there are 15 munches/channel or a total of 30 munches of FIFO storage, and the fast io system can deliver one munch per 8 cycles with the first munch arriving 20 cycles after the first IOFetch_. 13.8 ms is much smaller than the vertical blanking time and longer than the horizontal blanking time, so the FIFO will start out full at the beginning of a field and will be actively refilling itself during HS+HB of each scan line. If the memory system keeps up with the demands of the video channels, then the FIFO will tend to refill itself after momentary transients in which it empties out a little.

Consequently, we know that HS+HB = 1/(S*F) 2*VR, and that M = (HS+HB)/T less corrections for refresh references, storage references by other tasks, hold, and delays for tasks of higher priority than DWT. At F = 30 frames/sec, VR = 800 ms, and S = 1000 scan lines, we get HS+HB = 31.7 ms and M = 31.7/0.4 = 79 munches less corrections. There will be an average of two refresh references/scan line, so we get an upper bound of 77 munches = 19,712 bits/scan line from storage.

However, the DWT will not get all storage bandwidth. The DWT wakeup spacing is controlled by a SIP; the smallest reasonable spacing would result in one IOFetch every 8 cycles closer spacing would result in hold while a preceding IOFetch completed, so more processor cycles would be consumed without improving data rate. At this tightest spacing, DWT runs for 2 cycles out of every 8. Conceivably, worst case memory activity discussed in the "Fast IO" chapter could occur during these 6 cycles (a clean miss 3 cycles before the IOFetch, followed by a dirty miss 2 cycles before the IOFetch, each by a different task). However, the large amount of storage in the FIFO allows us to rely upon statistics to average out memory competition, so it is probably reasonable to allow DWT at least 80% of storage bandwidth or about 16,000 bits/scan line in the above example, which would accommodate 1000 line x 1000 pixels/line x 16 bits/pixel. For HB = 5 ms this is equivalent to a pixel clock period of 26.7 ns.

This is only one speed limitation. Since the 32-bit wide FIFO is accessed once/cycle alternately by the A and B channels (i.e., 16 bits/cycle/channel), and since exactly three doublewords are fetched before the horizontal scan begins for each channel, the maximum bits/scan line for each channel is about (3*32 bits)+[(26.7 ns/pixel)*(16 bits/50 ns)*(1000 pixels/line)] = 8640 bits/scan line. This means that unless both channels are running at the same data rate, the data rate will be significantly below the upper bound determined above. For example, in 24Bit mode, if the A channel runs at full resolution and gets 8640 bits/scan line, the B channel will run at half resolution and get only 4320 bits/scan line, so the maximum data rate would be about 1000 lines x 538 pixels/line x 24 bits/pixel.

# Ethernet Controller

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

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

*Read this chapter with Figure 16 in view.*

## Ethernet Packets

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

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

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

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

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

*Remarks*

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

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

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

## Controller Overview

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

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

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

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

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

The transmitter and receiver each have 16-word x 20-bit Fifos. The bits are 16 data + 2 parity + 2 spare (the receiver uses one of the spare bits). Each Fifo has read and write pointers, multiplexed into the address inputs of the storage chips, to select the next location to be read or written; these pointers are zeroed by IOReset. A Fifo is empty when the pointers are equal and full when (WritePtr+1) mod 16 equals ReadPtr. There are *bus registers* between the Fifos and IOB. Service requests from the Ether side of a Fifo are given priority. The Fifos are synchronous to $t_1$.

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

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

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

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

**Receiver**

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

The peculiar placement of status bits in Figure 16 eases emulation of the Alto Ethernet controller.

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

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

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

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

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

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

**Transmitter**

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

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

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

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

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

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

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

A collision may be detected by either the transceiver or PD. The occurrence of a collision is captured, synchronized, and used to abort the outgoing packet. The output of the first stage of the TxCollision synchronizer is wire-or'ed with PD output to jam the Ether after a collision. The jam lasts for one or two bit times, being the delay through the TxCollision synchronizer, TxFSM, and TxGo synchronizer.

### Clocks

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

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

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

### Task Wakeups

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

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

### Muffler Input

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

**Table 26: Ethernet Muffler Signals**

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

### IOB Registers

TIOA equals $15_8$ selects the IOB registers (called EthD). The transmitter bus register is loaded by Output_B and the receiver bus register is read with Pd_Input. At end-of-packet, after the last data word, the receiver delivers first the CRC word and then a status word containing the following bits:

| | |
|---|---|
| RxCollision | Receiver-detected collision occurred (can happen only if ReportCollisions has been set in the control word). |
| RxDataLate | Receiver data-late occurred one or more words of the last packet were lost. |
| RxCRCError | CRC was incorrect in last packet. |
| RxIncTrans | Last packet did not end on a word boundary. |

### Control Register

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

| | |
|---|---|
| TxCmdEnbl' | enables decoding of transmitter commands. |
| TxOn | enables the transmitter. The transmitter may be reset at any time by clearing this bit. Cleared by IOReset. |
| TxEOP | disables transmitter wakeups. EOT sets this bit after outputing the last word of a packet. It is cleared by the controller when the PE shuts down after an abort or normal end. Cleared by TxOn=0. |
| TxCntDwn | disables transmitter wakeups. Set by EOT to time a retransmission interval after a collision; cleared by the controller when the next rising edge of Pendulum occurs (period = 16 ms). N.B. the binary exponential backoff is done by firmware. Cleared by TxOn=0. |
| RxCmdEnbl' | enables decoding of receiver commands. |
| RxOn | enables the receiver, which may be turned off at any time by clearing this bit. Cleared by IOReset. |
| RxBOP' | disables receiver wakeups. Cleared by EIT to discard the currently arriving packet; set by the controller when the first word of the next packet is available. Cleared by RxOn=0. |
| TestCmdEnbl' | enables decoding of test commands |
| LoopBack | disconnects the transceiver, loops PE output to PD input, and enables TestColl'. Cleared by IOReset. |
| SingleStep | disables the 23.53 mHz oscillator. Changing this bit can produce a runt clock. Reset the transmitter first and expect an occasional bad receiver status. Cleared by IOReset. |
| NoWakeups | disables all controller wakeups. Cleared by IOReset. |
| TestClock | injects a single Dorado clock pulse ($t_3$ of the Output instruction) into the EtherClk logic. SingleStep must already be set. |
| TestColl' | injects a single Dorado clock pulse ($t_3$ of the Output instruction) into the collision synchronizer. LoopBack must already be set. |
| TestData | wire ORs with PD input. LoopBack must already be set and TxOn must already |

| | |
|---|---|
| | be false. Do not issue TestClock in an instruction that changes TestData. Cleared by IOReset. |
| ReportCollisions | allows the PD to report malformed bits as collisions. Cleared by IOReset. |

### Status Register

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

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

# Other IO and Event Counters

In addition to the disk, ethernet, and display controllers discussed in earlier chapters, Dorado contains a *general input/output interface* and a *junk task wakeup* located on the IFU board; the two registers used in this interface may alternatively be used as *event counters* in performance monitoring, and that use is also discussed here.

Since the IFU board is not interfaced to the IOB, it cannot use the slow io system to control these features, so functions are used instead.

## Junk Task Wakeup

The IFU board contains a circuit which wakes up the junk task (task 1) every 32 ms. The wakeup is dismissed by the AckJunkTW_B function; this function interprets B[15] as follows: a 1 enables wakeups; a 0 disables them; B[0:14] are ignored. The junk task can dismiss the wakeup by doing IFUTest_B with any value on B (but B[15] must be 0 to reenable the wakeup at the next 32 ms tick).

Junk task microcode will, among other things, maintain a Real Time clock.

## General IO

A 16-bit register called GenIn (synonym EventCntA) is used for general input; it can be read with the B_GenIn (synonym B_EventCntA) function but cannot be written by firmware. When used for general input, GenIn is written with information that is TTL-to-ECL converted from the backpanel.

A 16-bit register called GenOut (synonym EventCntB) is used for general output; it can be either read with the B_GenOut (synonym B_EventCntB) function or written with the GenOut_B (synonym EventCntB_B) function. GenOut is connected to the backpanel through ECL-to-TTL converters.

The plan is that devices such as Diablo printers can be connected to the GenIn and/or GenOut signals via backpanel connectors.

The choice of using one of these registers for general io or for event counting is determined by the InsSetOrEvent_B function discussed below.

## Event Counters

The GenIn and GenOut registers can alternatively be used as event counters. They cannot, of course, be used simultaneously for general io. The registers are setup for either io or event counting by the InsSetOrEvent_B function, where B[0:15] are interpreted as follows:

If B[0] is 1, then InsSet[0:1] are loaded as discussed in the "Instruction Fetch Unit" chapter. If B[0] is 0, then its the general io/event counters as follows:

B[4] enables counting of EventCntA

B[5] enables counting of EventCntB

B[8:10] select the event type to be counted by EventCntA as follows:

| | |
|---|---|
| 0 | True (i.e., every cycle) |
| 1 | Hold |
| 2 | Processor memory reference (not held) |
| 3 | Good IFUJump (i.e., not held and not an exception) |
| 4 | Miss |
| 5-7 | Backpanel events A, C, and E, respectively |

B[12:14] select the event type to be counted by EventCntB as follows:

| | |
|---|---|
| 0 | True |
| 1 | Hold |
| 2 | Successful IFU memory reference |
| 3 | IFUJump that wasn't ready |
| 4 | Miss |
| 5-7 | Backpanel events B, C, and D, respectively |

B[15] causes the event to be counted for all cycles if 1 or only for emulator or fault task cycles if 0.

To use the event counters, you first stop them counting and read their current values; then you tell them what to count and start them counting and your system running. Note that they never get reset, but just keep counting from wherever they are it's up to the user to worry about counter turnover.

The expected mode of operation is that the junk task will detect counter overflow and update double or triple-precision vectors in RM that count events; even if the counter is counting once per 60 ns cycle, counter wraparound only occurs every 3.93 ms, so a double-precision vector could count for at least 255 seconds and triple-precision for 228 days. Sample microcode for maintaining a double-precision counter is given in the example below:

```
*The double-precision vector consisting of two RM locations, CountHi and CountLo
*is initialized such that CountHi eq 0 and CountLo contains minus the value in
*the event counter, and another RM location called CountFlag is initialized to 0.
*The microcode below increments CountHi whenever the event counter cycles.
*At any instant, the high part of the total count is in CountHi and the low part
*is CountLo+event counter; CountHi has to be incremented by 1 if the counter
*just overflowed.

(CountLo) - (EventCntB') - 1;                *CountLo + event counter
Pd_CountFlag, Branch[.+2,alu>=0];
CountFlag_T-T-1, Branch[.+3];                *Set CountFlag to -1 in 2nd half of the counter cycle.
CountFlag_T-T, Branch[.+2,alu>=0];           *Set CountFlag to 0 in 1st half of the counter cycle,
CountHi_(CountHi)+1;                         *and increment CountHi, if we were in the 2nd half
. . .                                        *of the counter cycle last time.
```

The microcode for reading the counter when it is updated like this is as folows:

```
*Return to caller high part of event count in T, low part in Q.

TaskingOff, Pd_CountFlag;
T_(CountLo) - (EventCntB') - 1, Branch[.+3,alu>=0];    *CountLo + event counter = low part of result
TaskingOn, Branch[.+3,alu<0];                *Low part ovf iff CountFlag<0 and low sum >=0
T_(CountHi)+1, Q_T, Return;                  *High part of result = CountHi+1
TaskingOn;                                   *High part of result = CountHi
```

```
T_CountHi, Q_T, Return;
. . .
```

## Error Handling

In addition to single-error correction and double-error detection on data from storage, Dorado also generates, stores, and checks parity for a number of internal memories and data paths. The general concepts on handling various kinds of detected failures are as follows:

(1) Failures of the processor or control sections should generally halt Dorado because these sections must be operational before any kind of error analysis or recovery firmware can be effective.

(2) Failures arising from memory and io sections should generally result in a fault task wakeup and be handled by firmware. In some situations, such as map parity errors, it is especially important to report errors this way rather than immediately halting because firmware/software may be able to bypass the hardware affected by the failure and continue normal operation until a convenient time for repair occurs. In other situations, the firmware may be able to diagnose the failure and leave more information for the hardware maintainers before halting.

(3) IFU section failures and memory section failures detected by the IFU should generally be buffered through to the affected IFUJump, then reported via a trap; in this way, if it is possible to recover from the failure, then it will be possible to restart the IFU at the next opcode and continue.

(4) Memories and data paths involving many parts should generally be parity checked. It is not obvious that this is always a good idea because extra parts in the parity logic will be an additional source of failures, but instantly detecting and localizing a failure seems preferable to continuing computation to an erroneous and undetected result.

(5) When Dorado halts due to a failure, information available on mufflers and in the 16-bits of passively available error status (ESTAT) should localize the cause of the error as precisely as possible.

Since the MECL-10K logic family has a fast 9-input parity ladder component, the hardware uses parity on 8-bit bytes in most places; there is usually insufficient time to compute parity over larger units. IM and MIR, two exceptions, compute parity over the 17-bits of data in each half of an instruction; and the cache address section computes parity over the 15 address bits and WP bit.

Odd parity is used throughout the machine, except that the cache address section and IFUM use even parity. Odd parity means that the number of ones in the data unit, including the parity bit, should be odd, if the data is ok.

The control processor (Midas or the baseboard microcomputer) independently enables various kinds of error-halt conditions by executing a manifold operation discussed in the "Dorado Debugging Interface" document. It also has to initialize RM, T, the cache address and data sections, the Map, and IFUM to have valid parity before trying to run programs. Reasons for this will be apparent from the discussion below.

When Dorado halts, error indicators in ESTAT indicate the primary reason for the halt, and

muffler signals available to the control processor further define the halt condition; ESTAT also shows the halt-enables. Midas will automatically prettyprint a message describing the reasons for an error halt. The exact conditions that cause error halts are detailed in the sections below; the table here shows the ESTAT and muffler information which is relevant.

**Table 27: Error-Related Signals**

| *ESTAT Error Bit* | *ESTAT Enable Bit* | *Task Experiencing Halt* | *Related Muffler Signals and Meaning* |
|---|---|---|---|
| RAMPE | RAMPEen | Task2Bk | STK, RM, or T parity failure. *RmPerr* and *TmPerr* mufflers on each processor board indicate which byte of RM/STK or T had a parity failure. *StkSelSaved* indicates that *RmPerr* applies to STK rather than RM. |
| MdPE | MdPEen | Task2Bk<br>Task3Bk | processor-detected Md parity failure<br>if immediate _Md (_MDSaved false)<br>if deferred _Md (_MDSaved true)<br>*MdPerr* muffler on each processor board shows which byte of Md failed. |
| IMrhPE | IMrhPEen | CTD | parity failure of IM[17:33] |
| IMlhPE | IMlhPEen | CTD | parity failure of IM[0:16] |
| IOBPE | IOBPEen | Task2Bk<br>Task2Bk | Pd_Input parity failure if *IOBoutSaved* false<br>Output_B parity failure if *IOBoutSaved* true<br>*IOPerr* mufflers on each processor board show which byte failed. |
| MemoryPE | MemoryPEen | | cache address section parity failure, cache data parity failure on write of dirty victim or dirty Flush_ hit, or fast input bus parity failure. |

### Processor Errors

The processor has parity ladders on each byte of the following:

| | |
|---|---|
| input to RM/STK | generate parity for write of RM/STK |
| input to T | generate parity for write of T |
| B | generate parity for DBuf_B, MapBuf_B, Output_B, IM_B |
| IOB | check parity for Pd_Input and Output_B |
| Md | check parity for _Md |
| R | check parity for _RM/STK (unless bypassed from Pd or Md or replaced by _Id) |
| T | check parity for _T (unless bypassed from Pd or Md or replaced by _Id) |

Input ladders to RM/STK and T generate parity stored with data in the RAM; these ladders are not used for detecting errors.

The processor computes parity on its internal B bus (alub). The generated parity may be transmitted onto IOB when an Output_B function is executed; Store_ references write B data and parity in the cache; parity for IM writes and map writes is computed from B parity. None of the other B destinations either check or store B parity. External B sources do not

generate parity.

Parity on the R/T ladders is checked only when the R/T data path is sourced from the RAM, not when bypassing from Md or Pd is occurring, and not when R/T is sourced from Id. A detected failure causes the *RAMPE* error halt, which indicates that some byte of RM, STK, or T had bad parity. The muffler signals that further describe this error are in the PERR word: *StkSelSaved* is true if the source for R was STK, false if the source for R was RM; each processor board has *RmPerr* and *TmPerr* signals; *RmPerr* is true if the RM/STK byte on that board had bad parity, *TmPerr* if the T byte had bad parity. Note that if an instruction beginning at $t_0$ suffered an error, Dorado halts immediately after $t_4$; the muffler signals apply to the instruction starting at $t_0$. The *Task2Bk* muffler signals show the task that executed the instruction at $t_0$.

Md parity is checked whenever _Md is done; a failure causes the *MdPE* error-halt when enabled. The *_MDSaved* muffler signal in PERR is true when a deferred _Md caused the error (T_Md, RM/STK_Md), false when an immediate _Md (A_Md, B_Md, or ShMdxx) caused the error. On a deferred _Md error, Dorado halts after $t_6$ and *Task3Bk* shows the task that executed the instruction starting at $t_0$; on an immediate _Md, Dorado halts after $t_4$, and *Task2Bk* shows the task. The *MDPerr* muffler signals on each processor board show which byte of Md was in error.

Io devices (optionally) compute and send odd parity with each byte of data; the processor checks parity when the Pd_Input function is executed, but not when the Pd_InputNoPE function is executed. When enabled, an *IOBPE* error halts the processor at $t_4$ of the instruction that suffered the error; *Task2Bk* shows the task that executed the instruction. The processor also checks IOB parity on Output_B, and an error halts at $t_4$ as for Pd_Input. The *IOBoutSaved* muffler signal distinguishes Pd_Input from Output_B errors; an *IOPerr* muffler signal on each processor board shows which byte of IOB was in error; all of these are in the PERR muffler word.

> The processor generally does not pass parity at one stage through multiplexing to the next stage, so any failure in the multiplexing between one stage and the next will go undetected (exception: B parity passed through to IOB).
>
> For example, the processor could write Md parity sent by the cache into the T RAM, when T is being written from Md. Instead, however, it checks Md parity independently, but then recomputes the parity written into T with the input ladder. Hence, a parity failure detected on a byte of T can only indicate a failure in either (1) the input parity ladder; (2) the output parity flipflop; (3) the output parity ladder; (4) one of three 16x4 T RAM's; (5) one of two 4-bit latches clocked at $t_1$ (Figure 3) through which the output of the T RAM passes; (6) one of two 4-bit latches clocked by preSHC'.
>
> Parity is handled similarly for writes of RM/STK.
>
> Parity is similarly recomputed on B.

The processor does not generate or check parity on the A, Mar, or Pd data paths. Any failures of the A, Mar, B, Pd, or shifter multiplexing or of the ALU go undetected; failures of Q, Cnt, RBase, MemBase, ALUFM, or branch conditions go undetected.

*Remark*

Since 256x4 and 16x4 RAM's are used for RM, STK, and T, and since the processor is implemented with the high byte (0:7) on ProcH and the low byte (8:15) on ProcL, byte parity requires an additional 4-bit storage element on each board, of which only 1 bit is used. We could conceivably have used all 4 bits to implement a full error-correcting code for each byte of R and T data. However, there is insufficient time to correct the data. (Also, we use 256x1 RAM's instead of 256x4 RAM's for the RM and STK parity bits.)

Alternatively, parity could be computed over each 4-bit nibble rather than each 8-bit byte; the MC170 component allows nibble parity to be computed just as economically as byte parity. If this were done, then a parity failure would be isolated to a particular nibble. With byte parity, a detected failure could be any of 9+ components; with nibble parity, it would be isolated to one of 6+ components. Implementing nibble parity for RM/STK and T would require about 4 more ic's per board than byte parity.

It is hard to say whether the additional precision of nibble parity would be worth the additional parts.

### Control Section Errors

The control section stores parity with each 17-bit half of data in IM. When IM is written, the two byte-parity bits on B are xor'ed with the 17th data bit to compute the odd parity bit written into IM. It is possible to specify that bad (even) parity be written into IM, and this artifice is used to create breakpoints; bad parity from both halves of IM is assumed to be a deliberately set breakpoint by Midas.

IM RAM output is loaded into MIR and parity ladders on each 17-bit half give rise to error indicators that, when enabled, will halt the processor *after* $t_2$ of the instruction suffering an error. For testing purposes, halt-on-error can be independently enabled for each half of MIR. Both the unbuffered output of the MIR parity ladders and values buffered at $t_2$ appear in ESTAT. The buffered values show the cause of an error halt, and the unbuffered signals allow Midas to detect parity errors in MIR before executing instructions or when displaying the contents of IM.

The special MIRDebug feature discussed in the "Dorado Debugging Interface" document prevents MIR from being loaded at $t_2$ when MIR parity is bad. In other words, when the MIRDebug feature is being used, all of the $t_2$ clocks in the machine will occur except the ones to MIR. This feature prevents the instruction that suffered an error from being overwritten at the expense of being unable to continue execution after the error. MIRDebug can be enabled/disabled by the control processor.

### IFU Errors

The IFU never halts the processor; any errors it detects are buffered until an IFUJump transfers control to a trap location. The errors it detects, discussed in "IFU Section", are parity failures on bytes from the cache, IFUM parity failures, and map parity failures on IFU fetches.

### Memory System Errors

There is no parity checking on Mar or on data in BR, so any failure in the address computation for a reference goes undetected. However, valid parity is stored with VA in the cache, and any failure detected will cause the MemoryPE error to occur, halting the system (if MemoryPE is enabled).

Parity is also stored in the Map (computed from B parity) and an error causes a fault task wakeup in most situations (Exceptions: IFU references and Map_ references do not wakeup the fault task when a map parity error occurs).

The cache data section stores valid parity with each byte of data. When a munch is loaded from storage, the error corrector carries out single-error correction and double error detection using the syndrome and recomputes parity on each 8-bit byte of data stored in the cache. When a word from B is Store_'d in the cache, byte parity on B is stored with the data.

A MemoryPE error occurs if, when storing a dirty victim back into storage, the memory system detects bad parity on data from the cache.

The IFU and processor also check parity of data from the cache, as discussed previously.

**Sources of Failures**

In a full 4-module storage configuration, Dorado will have 1173 MOS storage, about 700 Schottky-TTL, 3000 MECL-10K, and 60 MECL-3 DIPs, and about 1500 SIPs (7-resistor packages). This logic is connected with over 100,000 stitch-welded or multiwire connections to sockets into which the parts plug; logic boards connect to sidepanels through about 2500 edge pins. Sockets are used for all the RAM DIPs in the machine; other parts are soldered in. Given all these potential sources of failure, reliable operation has been a surprising achievement.

Initial debugging of new machines has been slow and difficult, requiring expertise not easily available in a production environment. In addition to mechanical assembly, board stuffing, and testing for shorts and opens both before and after stuffing, each machine has averaged about one man month of expert technician time to repair other malfunctions before it could be released to users.

Once released, the Dorados have been pretty reliable. During a 100-day period (6 October 1980 to 14 January 1981) the CSL technicians kept records of service calls made for approximately 15 Dorados in service at that time. The following summarizes the 43 service calls that were made.

37 days mean time between service calls per machine.

45 days mean time between failures (some service calls were for microcode or software problems).

2.5 hours per machine per month average service time.

13% of failures and 5% of time reseating logic boards in the chasis (connectors not making contact).

11% of failures and 17% of time on open nets.

13% of failures and 12% of time repairing 16k MOS RAM failures (standard configuration was 2 modules).

37% of failures and 28% of time replacing other DIPs and SIPs.

5% of failures and 10% of time on T80 problems.

13% of failures and 11% of time on power supply failures.

2% of failures and 2% of time on Terminal and display problems.

4% of failures and 20% of time on repairing boards damaged during manufacturing or overheating.

The power supply failures were due to problems that have since been corrected, and most of the service calls for microcode or software problems would not happen in the more mature environment we have today. However, the other failures are believed to be representative. Note that none of the MOS RAM failures was the reason for a service call. These were found when testing a machine with diagnostics after a service call had been made for some other reason.

**Error Correction**

Reliability has been improved by error-correction on storage. The Dorado error-correction unit of 64 data and 8 check bits (quadword), guards 1152 MOS RAMs from single failures, but almost no other parts on storage boards or in the error corrector are guarded.

Our Alto experience suggests that some machines repeatedly fail under normal use due to undiagnosable failures. For this reason, error correction should be viewed as guarding not only against new failures but also against imperfect testing of parts that are either already bad or subject to noise (e.g., cosmic rays) or other kinds of intermittent failure. The latter may be more important in our environment.

The failure summary above indicates, for a small sample, that 16k MOS RAMs, accounting for 6% of all DIPs and SIPs (because the 15 Dorados had 2-module configurations, half the maximum) average about 4 times the failure rate of other parts and account for about 1.5 failures/year/Dorado this would become 3 failures/year with a 4-module configuration. If we continue to do this well, a Dorado with error correction should run for years without uncorrectable MOS RAM failures. The manufacturer’s literature indicates that the dominant failure mode appears to be single-bit failures with row and column addressing failures affecting many bits somewhat less frequent, but we don’t know the distribution of these.

If MOS failures do become significant, different strategies may be needed for single- and multi-address failure modes. With a multi-address failure, another failure in the same quadword causes a double error; but many single-address failures can occur in the same quadword without double errors.

The failure model used below shows that with no periodic testing and replacement of bad MOS RAMs, fatal failure statistics of the 1152 RAMs would approximate those of a 108 RAM uncorrected store. By thoroughly testing storage and replacing bad parts 4 times more often than the mean time to total failure of a part (defined below), the likelihood of an uncorrectable RAM failure crashing the system can be made insignificant compared with other sources of failure.

Although system software could bypass all pages affected by a multi-address RAM failure, the entire module, 25% of storage, would be eliminated, so this is impractical except on an emergency basis. Continuing execution despite a multi-address RAM failure will result in a double error when any other coincident storage failure occurs in the same quadword; 1/16

of future failures will do this.

Some interesting questions are: How does MTBF vary with the EC arrangement? MTBF is pertinent if we let Dorados run until they fail. Alternatively, how likely is a failure in the next day, week, or month, if we test the memory that often and replace bad RAMs? These questions can be asked assuming perfect testing (no failures at t=0) or imperfect testing (some likelihood of failures at t=0 because diagnostics didn't find them).

To answer them, MOS RAM failures are modelled as one of two types: those affecting a single address in the RAM (called SF's), and those affecting all addresses (called TF's). We assume that TF's occur about 1/4 as often as SF's in 4Kx1 RAM's. RAM failures are assumed exponentially distributed, correct if the failure rate doesn't change with time; over the time range of interest, this is reasonable. Finally, perfect testing is assumed, so there are 0 failures at t=0. These assumptions give rise to the following:

let p = prob that an ic has a TF = $1 \quad e^{at}$
let q = prob that an ic has a SF = $1 \quad e^{bt}$
let n = number of MOS RAMs in the memory

Without error correction, MTBF is the integral from 0 to infinity of $[(1\ p)(1\ q)]^n$ = $1/n(a+b)$. With b = 4a, in our 4-module system with n = 1024, this is $1/5120a$ = $.00018/a$.

With error correction, failure occurs when, in a single EC unit, a TF coincides with either another TF or an SF. This ignores two coinciding SF's which is about 4000 (16k RAMs) or 16000 (64k RAMs) times less likely.

let n = number of RAMs in an error correction unit
then Prob[no failure] = Prob[no TF] + Prob[1 TF and 0 SF]

Prob[no TF] = $(1\ p)^n$
Since failure modes are independent,
Prob[1 TF and 0 SF] = $np[(1\ p)(1\ q)]^{n\ 1}$
Prob[no failure] = $P_{ok} = (1\ p)^n + np((1\ p)(1\ q))^{n\ 1}$
$P_{ok} = e^{nat} + n(1\ e^{at})(e^{(a+b)(n\ 1)t})$

This is the probability for a single EC unit, so mean time to failure for all MOS storage is $P_{ok}$ raised to a power equal to the number of EC units. In other words, the argument of the integral for a 4-module x 4 quadwords/module system is $P_{ok}^{16}$ with n = 64+8; it is $P_{ok}^{4}$ with n = 256+10 for a one munch EC unit.

Then, expected time to failure for our 16 x n=64+8 memory system, is about:

$(1/n) * (1/16a + 16a/(16a+b)^2 + 240a^2/(16a+2b)^3 + 3360a^3/(16a+3b)^4)$
$= (1/an) * (1/16 + 1/25 + 5/288 + 105/17208)$
$= (1/16an) * (1 + .64 + .28 + .006) = 1.93/16an$
$= 1.93/16*72*a = .00168/a$

In other words, mean time to failure is about 1.93 times longer than the time to the first TF = 9.5 times better than with no error correction = as often as 1024/9.5 = 108

uncorrected storage ic's.

The results don't change much when imperfect testing is assumed. The effect of this is to replace densities for p and q by 1 Ae $^{at}$, where A would be .999 if there was a 1/1000 chance of a MOS ic being bad at t=0.

*Remarks*

On each storage board, data from MemD is transported to a shift register consisting of 8 flipflops which are then written into the MOS RAM's after transport has been completed. This arrangement is unfortunate any failure in one of these components will cause a multiple error, and there are about 250 of these parts in a full storage configuration.

One way to eliminate this problem while simultaneously reducing the part count on each storage board would be to make modules consist of four storage boards, rather than two, so that only four flipflops receive data on each bit path during transport; since each of these is in a different quadword, single failures would not cause multiple errors.

The Dorado EC operates on *quadwords*, requiring 8 check-bits/64 data bits, or a 12.5% storage penalty. Alternative schemes are: 10 check bits/256 data bits (3.9%); 9 check bits/128 data bits (7.4%); 7 check bits/32 data bits (22%); and no error correction at all (0%).

The implementation of the EC pipeline is such that wider correction units significantly increase the time for a miss. The current quadword error corrector requires 7 clocks (3 clocks for setup and correction, 1 clock per word of the quadword); this would become 11 clocks with a 128-bit EC scheme or 19 clocks with a 256-bit EC scheme. Although cache hit rate seems to be above 99%, some implementation avoiding this delay would still be needed to make larger correction units attractive.

If our quadword correction unit were replaced by a 4 x n=256+10 scheme:

$1/4na + 4a/n(4a+b)^2 + 3a^2/2n(2a+b)^3$, where for b = 4a this is
$(1/4na)*(1 + 1/4 + 1/36) = 1.28/4na = .0012/a$

In other words, MTBF is about 1.28 times longer than the time to the first TF. So error correction has increased MTBF by a factor of 6.2 over no error correction; alternatively, a 1064-RAM corrected memory fails as frequently as a 1064/6.7 = 159 RAM uncorrected memory.

Surprisingly, the 64+8 EC scheme has only 42% longer MTBF than a 256+10 EC scheme. This improvement may not be worth the 96 additional MOS RAM and 80 other DIPs required for address buffering; the 80 additional DIPs might cause more failures than they save, being a net loss.

The other method of maintaining our systems is to regularly test storage and replace bad RAMs. Then the likelihood of no double error before replacement is simply the value of the probability distribution ($P_{ok}^4$ and $P_{ok}^{16}$ above) at the selected instant. This reduces to an approximation of the form $P_{ok} = [e^x + xe^x]^m$ where x = nat, m is 4 or 16, and n = 72 for m=4 or 266 for m=16. If this is evaluated at t = 1/mna, 1/2mna, 1/4mna, etc. the following results are obtained:

**Table 28: Double Error Incidence vs. Repair Rate**

| m | 1/mna | 1/2mna | 1/4mna | 1/8mna |
|---|---|---|---|---|
| 4 | .52 | .81 | .94 | .98 |
| 16 | .79 | .84 | .98 | .99 |

The interpretation of this table is as follows: Measure mean time to total failure (TF) of a MOS RAM and call this time 1/a; then assume 4 SF's per TF. Then the rate at which TF's occur in storage will be 1/mna. So the above tables show probability that the Dorado hasn't suffered a double error when tested and fixed as often, 1/2 as often, 1/4 as often, or 1/8 as often as the mean rate of TF's.

# Performance Issues

This chapter discusses two issues:

(1) How rapidly will Dorado be able to execute Mesa, Lisp, SmallTalk, etc. macroprograms;

(2) What relationship do some of the design parameters bear to performance;

## Cycle Time

The first issue is cycle time. Dorado was designed for a 50 ns cycle time; the first three prototypes used stitchweld technology for interconnections and operated correctly at 55 ns cycle time; however, subsequent machines are being built using multiwire technology and will not operate faster than about 60 ns cycle time. The baseboard at present initializes the clock period to 64 ns for all machines during a boot, although there is some indication that design changes made recently and repair of a few lingering slow path problems would permit 5 to 10 ns faster operation.

With respect to achievable cycle time, the two important differences between stitchweld and multiwire technology are that stitchweld uses point-to-point wiring and has wire impedance of about 100 ohms (which is ideal), but multiwire uses Manhattan (square-corner) wiring with wire impedance of about 50 ohms on the inner layer and 70 ohms on the outer layer of wiring (Most signals are in the outer layer.); longer wires and imperfect impedance matching result in slower speed.

## Emulator Performance

Gene McDaniel's measurements of the Alto Mesa compiler have been adjusted to make them compatible with Pilot Mesa and are summarized below. It must be pointed out that the compiler makes heavier use of short pointers than do Pilot Mesa programs; programs being developed now are heavily biased toward long pointers and would be slower than the execution rate below indicates. Average execution rate was about 5.6 cycles/opcode excluding disk wait. About 38% of all cycles are consumed by XFER opcodes (i.e., subroutine call or return) and account for about 6% of opcodes executed. If these are excluded, the remaining 94% average about 3.1 cycles/opcode; if jumps and conditional jumps are also excluded (about 14% of executions), the others average 2.5 cycles/opcode. These times include all memory and IFU delays.

These excellent results indicate that there are no unusual delays due to problems with the memory or IFU and that the processor is completing most opcodes quickly. Since XFER opcode take 34 (local) to 54 (external) cycles/opcode excluding memory delays, speeding, respecifying, or reducing executions of XFER seem to be the most promising ways of improving performance.

In the above results, instruction forwarding has saved an average of about .25 cycles/opcode or about 4% overall, in agreement with our expectations.

For SmallTalk and Lisp instruction sets, performance is much worse than Mesa (averaging over 30 cycles/opcode on Smalltalk 76). Careful studies should be made to understand the reasons for this fully, but one reason is that the 16-bit word size is a serious limitation. Long storage pointers are used extensively, so execution would be substantially faster on a machine with, say, 32-bit data paths.

### IFU Not-Ready Wait

For the Mesa compiler, 19.5% of all cycles were in IFU not-ready wait; 16% due to incorrectly predicted jumps, 2.5% to cache miss wait, and 1% to other causes. The 16% due to incorrectly predicted jumps might be improved.

The Mesa microcode presently predicts that all conditional jumps will not jump; it is desirable to predict not-jump unless more than 75% of executions jump due to the overhead of restarting the IFU an extra time. 40% of the time the prediction is wrong and a jump occurs, so it seems that the microcode is doing the best it can.

However, some loops ("while J ne 0 do," for example) are compiled as a normally-false conditional jump at the beginning of the loop and an unconditional jump from the end of the loop back to the beginning; a faster sequence is a normally-true conditional jump at the end of the loop, eliminating the unconditional jump altogether. The general objectives in changing the compiler would be as follows: (1) Eliminate unnecessary jumps and conditional jumps; (2) Make the jump/not-jump execution of conditional jumps be as predictable as possible; and (3) Make the not-jump path be the most likely, unless this conflicts with objective (1).

### Microstore Requirements

Speed is not the only issue some reduction in microstore requirements might be possible through design changes. Space requirements for a 1981 release of the Alto/Mesa emulator system were as follows:

**Table 29: Utilization of the Microstore**

| | |
|---|---|
| Mesa basic opcode set | $2024_8$ |
| Cedar allocator & collector | $576_8$ |
| Floating point | $457_8$ |
| Alto opcode set | $1163_8$ |
| Alto BCPL Runtime | $226_8$ |
| BitBlt subroutine | $416_8$ |
| Fault handling | $65_8$ |
| Ethernet driver | $255_8$ |
| Disk driver | $430_8$ |
| Display driver | $500_8$ |
| Junk io driver | $76_8$ |
| LoadRam | $100_8$ |
| Initialization | $150_8$ |

Total $7673_8$ leaving $105_8$ free locations

Since we do not require that more than two emulators be loaded in the microstore at one time, there is presently a little space left for extensions. MicroD is able to utilize well over 99% of the available microstore.

The third performance issue is cache efficiency and miss wait; the fourth is available io bandwidth and io task cycle consumption. These are discussed in sections below.

**Cache Efficiency and Miss wait**

The value of shortening the wait for a storage read is roughly proportional to miss likelihood. Suppose that the prototypical opcode was a one-byte opcode implemented by the following microcode:

```
Fetch_Id, StkP+1;
Stack_Md, IFUJump[0];
```

For this example, execution time on a hit is 2 cycles; on a miss, 28 cycles. Delay for IFU misses must be added to this. Since the IFU is 6 bytes ahead of the current opcode, its misses delay 28 cycles less execution time for preceding 6 bytes; if any of the 6 bytes itself causes a miss, IFU delay will be 0 because it will catch up; the IFU never gets two misses (in this example) because it crosses at most one munch boundary. Hence, execution time will be 2 + 26*(1 H) + (28-12)*$H^6$*(1 H), with the following results:

**Table 30: Execution Time vs. Cache Efficiency**

| *Hit %* | *Execution Cycles* | *IFU Cycles* | *% Miss Wait* |
|---|---|---|---|
| 100 | 2.00 | .00 | 0 |
| 99 | 2.26 | .15 | 17 |
| 98 | 2.52 | .28 | 29 |
| 96 | 3.04 | .50 | 44 |
| 94 | 3.56 | .67 | 53 |
| 92 | 4.08 | .79 | 59 |

This crude analysis shows the importance of cache efficiency in determining system performance. Fortunately, measurements made by Doug Clark and Gene McDaniel indicated the following surprisingly high cache hit statistics:

> Overall cache hit rate on three Mesa programs was 99.2% to 99.8%. 4.9% to 8.1% of all cycles were held. 10% to 19% of references were Store_'s, the rest fetches. 16% to 66% of misses had dirty victims, which cause additional cycles to be held while the cache address section is busy.
>
> Another measurement showed a 99.7% hit rate for IFU references.
>
> The processor obtains a word from the cache in 16% of all cycles and the IFU in 32% of all cycles; the processor actually shuts out the IFU by making its own

reference about 20% of the time.

Provision has been made to expand the Dorado cache to 16k words, when 4k x 1 MECL RAM's are economically available, but the existing cache is so efficient that this may never be necessary.

**Performance Degradation Due to IO Tasks**

To first approximation, only the display controller word task (DWT) uses enough storage bandwidth to interfere significantly with emulators. Since it uses the fast io system, DWT requires service once/munch and will require two instructions/wakeup in the ordinary case. In addition, if the next instruction (by another task) issues a memory reference, it will always be held one cycle while the DWT's IOFetch_ advances ASRN.

A quick calculation shows that at an io bandwidth of 256 x $10^6$ bits/sec ($10^6$ munches/sec) the display controller will use 48% of storage bandwidth and 12% of processor cycles at 60 ns/cycle.

The earlier example showed that with no io interference and a 99% hit rate, the emulator spent 17% of cycles in miss wait, 83% in useful execution. With a 256 x $10^6$ bit/sec display active, emulator misses are slowed about 2 cycles each, so the overall effect of the display would be that about 78% of all cycles are emulator executions, 12% display task executions, and 16% hold; the one cycle holds for IOFetch_ would make performance somewhat worse than this.

An IOFetch_ by the display task to the same cache row as an emulator miss will remain in the address section, increasing display task latency and requiring more buffering. However, this won't degrade emulator performance.

The Alto monitor only uses 14.7 x $10^6$ bits/sec (1/17 of the above) and would not interfere appreciably with emulators.

The disk controller is the fastest "slow" io device among standard peripherals. When running, its word interrupt task reads a double word from the cache every 3.2 ms in a 3 instruction/interrupt inner loop, consuming about 5.6% of all cycles at 60 ns/cycle. Its memory references consume the cache at a rate of .04 munches/ms, low enough that storage interference with the emulator isn't significant. However, a 256-word disk transfer displaces about 1/16 of the cache entries, so the emulator may experience a lower hit rate.

**Cache and Storage Geometry**

The current geometry was chosen without measurements or simulation of programs, but measurements made since then have indicated a surprisingly good cache performance, so not much could be gained through changes.

The following parameters are relevant:

1 word as the unit of storage inside the memory pipeline;
16-word *munch*;

256 munches in the cache (expandable to 1024);
4 *columns* in the cache.

*Munch Size*

A 16-word munch size was chosen primarily because 8 cycles for transport balances 10 cycles for storage access, avoiding loss of bandwidth. The use of 256x4 RAM's to implement the cache address section allows the original 4k-word cache (implemented with 1kx1 RAM's) to be expanded to 8k words or 16k words, when 4kx1 RAM's are economically available this is possible because only 64 of the 256 words in the address section are being used with the 4k-word cache. Miss wait is about 28 cycles and storage bandwidth about 533 x $10^6$ bits/sec with 16-word munches.

8-word munches would lower the storage bandwidth to about 262 x $10^6$ bits/sec, probably unacceptable. Also 8-word munches would limit cache expansion to 8k words. However, miss wait would be reduced to about 24 cycles because transport would require only 4 cycles. 32-word munches would not allow greater storage bandwidth to fast io devices because bandwidth is already limited by transport with 16-word munches. Nor would it allow expansion to a larger cache data section because we have no way to build a data section larger than 16k words. Also, miss wait would be slowed to 36 cycles, so it does not seem that this munch size is attractive.

For a given size of the cache data section, with smaller munches the cache will tend to stabilize with a larger amount of useful information; however, when a program is changing contexts, larger munches might bring the new context into the cache more quickly. Also, fast io tasks will interfere less with the emulator on larger munches because fewer wakeups and IOFetch_'es will be required. However, the extra buffering and longer miss wait offsets this advantage somewhat.

Considered together, these factors suggest that the 16-word munch we are using is substantially better than either 8 or 32-word munches.

*Data Path Width*

Having only 16 bit wide data paths slows misses. Doubling the paths to 32 bits would reduce EC time by 1 cycle and transport time into the cache by 4 cycles (i.e., delay on misses would be 23 cycles instead of 28). There were not enough edge pins to do this. However, if a method of doubling the path width were found, the storage system would probably be arranged as two modules of four storage boards each rather than four modules of two boards each, and 32-word munches might be better than 16-word munches.

*Cache Columns*

The reason for multiple columns is to approximate LRU reloading; the columns are moderately expensive because separate hit logic has to be provided for each one; the V-NV stuff also costs a few ic's with more than two columns. Altogether the current 64x4 cache is about 40 ic's larger than a 128x2 cache (Because of its 50-50 LRU behavior on the fourth column, our cache is somewhere between the 64x4 and 128x2 or 128x3 caches below.). The table below shows likelihood that the Nth LRU munch is no longer in the cache for various geometries:

**Table 31: Cache Geometry vs. LRU Behavior**

| N | 32x4 | 64x2 | 128x2 | 32x3 | 64x3 | 128x3 | 64x4 | 128x4 |
|---|---|---|---|---|---|---|---|---|
| 4 | .000 | .001 | .000 | .000 | .000 | .000 | .000 | .000 |
| 8 | .000 | .006 | .002 | .002 | .000 | .000 | .000 | .000 |
| 16 | .001 | .025 | .007 | .013 | .002 | .000 | .000 | .000 |
| 32 | .017 | .089 | .026 | .077 | .014 | .002 | .002 | .000 |
| 64 | .140 | .264 | .090 | .323 | .079 | .014 | .018 | .002 |
| 128 | .570 | .596 | .264 | .767 | .323 | .080 | .141 | .019 |
| 256 | .960 | .910 | .595 | .987 | .764 | .323 | .568 | .142 |
| 512 | | | | | | .763 | .959 | .567 |

These numbers are computed from a binomial distribution using the following formulae:

let R = rows in cache
let C = columns in cache
then p = (R 1)/R = probability that a munch of VA is in its row
then q = 1/R = probability that a munch of VA is not in its row
then probability of a miss for the nth element is:

| C | P(miss) |
|---|---|
| 1 | $1\ p^n$ |
| 2 | $1\ p^n\ nqp^{n\ 1}$ |
| 3 | $1\ p^n\ nqp^{n\ 1}\ n(n\ 1)q^2p^{n\ 2}/2!$ |
| 4 | $1\ p^n\ nqp^{n-1}\ n(n\ 1)q^2p^{n\ 2}/2!\ n(n\ 1)(n\ 2)q^3p^{n\ 3}/3!$ |
| etc. | |

Without extensive measurements on programs, it is impossible to know how much better, say, a 32x4 cache is than a 64x2 cache, or to know whether a 128x2 cache is better or worse than a 32x4 cache, for example. If a particular program is confining itself to a very small set of munches, then more closely approximating LRU reloading is most important. However, if the likelihood of reference flattens out after a small N, then it won't matter much that LRU reloading isn't very well approximated the total size of the cache will be a more important determinant of performance.

## Glossary

a - the first 8-bit operand of a two-byte or longer opcode.

b - the second 8-bit operand of a three-byte or longer opcode.

*bypassing* - a number of memories and task-specific registers in Dorado (RM, STK, and T, for example) are written with data that might be needed before the write occurs. These are implemented so that data about-to-be-written is substituted for data read from the register or memory when appropriate. This substitution is called *bypassing* and enables Dorado to run considerably faster than would otherwise be possible.

*cache entry* - a munch together with VA of the munch and 4 flag bits. For a 64 row x 4 column cache, VA[28:31] are the word in the munch, VA[22:27] address the row, and VA[7:21] are stored in the cache entry.

*column* - one of 4 groups of 64 (expandable to 256) cache entries. The cache column in which a word with VA resides is determined by comparing VA[7:21] with the corresponding bits stored in the four columns at row VA[22:27]. Thus a memory word may occupy one of 4 locations in the cache.

*control processor* - the microcomputer on Dorado's baseboard, or the Midas program operating Dorado from an Alto.

*dirty* - a *cache entry* is dirty if the information in it differs from information in storage, because a store has been done into the cache, and storage has not yet been updated. A *page* is dirty if a store has been done into the page since its map dirty bit was cleared.

*emulator* - the lowest priority task, number 0, always awake. The emulator is distinguished by the fact that it cannot block, can use Stk, and has a private pipe entry. Primarily the emulator task will implement instruction sets.

*entry vector* - the exit microinstruction of an opcode sends control to the first microinstruction of the next opcode by means of IFUJump[n] (n = 0 to 3), where n chooses one of 4 entry microinstructions for the next opcode; these four microinstructions are the next opcode's *entry vector*.

*fault task* - the highest priority task, number 15, woken whenever a memory fault or stack error occurs.

*hit* - a reference which finds the desired word in the cache.

*Midas* - the Alto program used for loading and debugging Dorado remotely.

*miss* - a reference which does not find the desired word in the cache.

*module* - the unit in which storage is packaged, either 64K, 256K, or 1M words. A machine may have 1 to 4 modules.

*MTBF* - mean time between failures.

*munch* - 256 bits, or 16 machine words; the unit of data for main storage.

*parity* - the parity of a data unit is the exclusive-or of all bits in the data unit; parity has the property that changing any single bit in the data unit will also change the parity, so it can be used to detect single failures. A data unit has *odd parity* when the number of 1's in the unit is odd, *even parity* when the number of 1's is even. Dorado uses odd parity everywhere, which means that the number of 1's in the data unit including its associated parity bit should be odd when data is correct.

*PC* - "program counter". In this manual PC refers to the 16-bit byte displacements relative to BR 31 (the codebase) which are maintained by the IFU for the current instruction set. This term should be distinguished from TPC, which refers to the address of the next microinstruction for a task.

*pipe* - a 16-entry memory which records the state of the last few storage references.

*quadrant* - one of the four 4k-word regions in a 16k-word control store.

*RAM* - "random access memory"; selected words in the memory can be both read and written.

*reference* - a reference to the memory, initiated by the processor or by the IFU. A *processor reference* transfers a single word between the cache and the processor; an *io reference* transfers a munch between storage and an io device.

*ROM* - "read-only memory"; the contents of the memory are specified when the hardware is constructed and cannot be modified during program execution. ROM elements used on Dorado can be reprogrammed with a special device constructed for the purpose.

*row* - one of the 64 or 256 groups of 4 cache entries. The cache row in which a word resides is determined by bits 20..27 of its virtual address.

*storage* - the main memory of the machine, organized in munches of 256 bits, or 16 machine words.

*storage reference* - a reference to the storage, initiated as a result of a memory reference. A processor reference causes a storage reference if there is a cache miss or if the *FDMiss* control is true in the memory control register; an io reference always causes a storage reference.

*storage reference number (SRN)* - an address of a pipe entry which identifies a particular storage reference.

*subtask* - a two-bit number presented by an io device to the processor and memory system while its task is running. The processor OR's subtask with RBase[3]..RSTK[1] in determining the RM address and with MemBase[2:3] in determining the base register selection. The memory system buffers the subtask for fast io devices, and then sends it over the Fin or Fout bus as part of device identification.

*tag* - The extra bit in Md readout which complements for successive Fetch_'es and Store_'s by the same task. Agreement of the bit in Md with the current value equals reference finished.

*task* - one of the 16 priority scheduled tasks. Special tasks are the emulator (task 0, lowest priority) and the fault task (task 15, highest priority). Other tasks are paired with io controllers.

*VA* - virtual address.

*Vacant* - a cache entry or map entry which does not contain valid data.

*Victim (Vic) memory* - stores 4 bits for each cache row. Two of the bits specify the *victim* which will be chosen if a reference to that row results in a miss, and the other two are the next victim.

*victim* - on a processor reference that causes a cache miss, the cache entry chosen to be replaced by the referenced data.

*WP* - write protected. Map entries and cache entries have bits with this name.