*** Preliminary Draft ***

# Dandelion Hardware Manual

| | |
|---|---|
| Date: | March 1982 |
| Version: | 2.2 |
| Prepared by: | Ron Crane, Dan Davies, Robert Garner, and Roy Ogus |

Release Stage: DRAFT/released/issued

XEROX
Office Products Division
Systems Development Department
3450 Hillview Avenue, Palo Alto, California 94304, USA

This document contains proprietary information and is for internal Xerox use only.

# Dandelion Hardware Manual

## Table of Contents

# 1.0 Overview

This manual describes the Dandelion (Series 8000 Processor) hardware. There are no page-by-page descriptions of schematic diagrams nor listings of PROMS and microcode. This manual should help the microcoder understand the hardware and help the trouble-shooter understand the schematics. It may also be used to get a general understanding of the machine.

This introductory chapter looks at the major characteristics of each controller and processor in the system. The logical boundaries of the Dandelion are shown in Figure 1. There are two main processors: the Central Processor (CP) and the Input/Output Processor (IOP). The CP controls the high bandwidth periperal devices and emulates the target language (e.g., Mesa). It is a high performance, microprogrammable processor which has been optimized for cost. The IOP, on the other hand, supervises the lower-speed devices, such as the mouse and keyboard, and controls the booting process. It is also used as a base to run diagnostics. It employs a traditional microprocessor (8085) in an 8-bit bus architecture. To the CP, the IOP appears as another high bandwidth I/O device.

## Central Processor

The CP executes microcode to control device controllers and main memory. Only the CP can access main memory. When devices request CP cycles (they get three per request), they can read or write one memory location. The processor, together with the memory, are time-division multiplexed among the device controllers in a round-robin fashion. The idea is that the (expensive) high-speed processor is shared among the (inexpensive) controllers. The controllers can be made very small because the round-robin nature of the memory access mechanism *guarantees* maximum memory latencies compatible with the controller bandwidths (unlike general bus architectures).

Time is divided up into *rounds*, where a single round consists of five slots, called *clicks*. Each click is preallocated to one (or more) of the device controllers. If a controller desires CP service or wants to transfer a word to or from memory, it raises its wakeup request and the CP will schedule the controller's microcode *task* for the next click in the round allocated to the device. If a controller does not desire any service, the click is allocated to the language Emulator instead. It can be seen that the CP hardware must preserve the microprogram counters for each of the controller's tasks.

Clicks are further divided into three *cycles*: exactly one microinstruction is executed in each cycle. Memory requests must always be started in the first cycle (c1) of a click, thereby guaranteeing the memory's latency for the controllers (and eliminating the need for more interclick state). A cycle is 137 nanoseconds in duration, thereby setting the memory access time at 411 nanoseconds (39 Mbits/sec). The simplest and most frequently executed Emulator instructions complete in one click (411 nS).

The CP executes microinstructions from a 4K-by-48-bit, writeable control store. The heart of the ALU is implemented with a high-speed 2901 bit-slice processor. There is an auxiliary register file of 256 words and a device used to rotate words 4, 8, or 12 bits. Every 48-bit microinstruction contains the 12-bit address of the next instruction. Branching and dispatching are accomplished by *or*ing condition bits into the "next address" field.

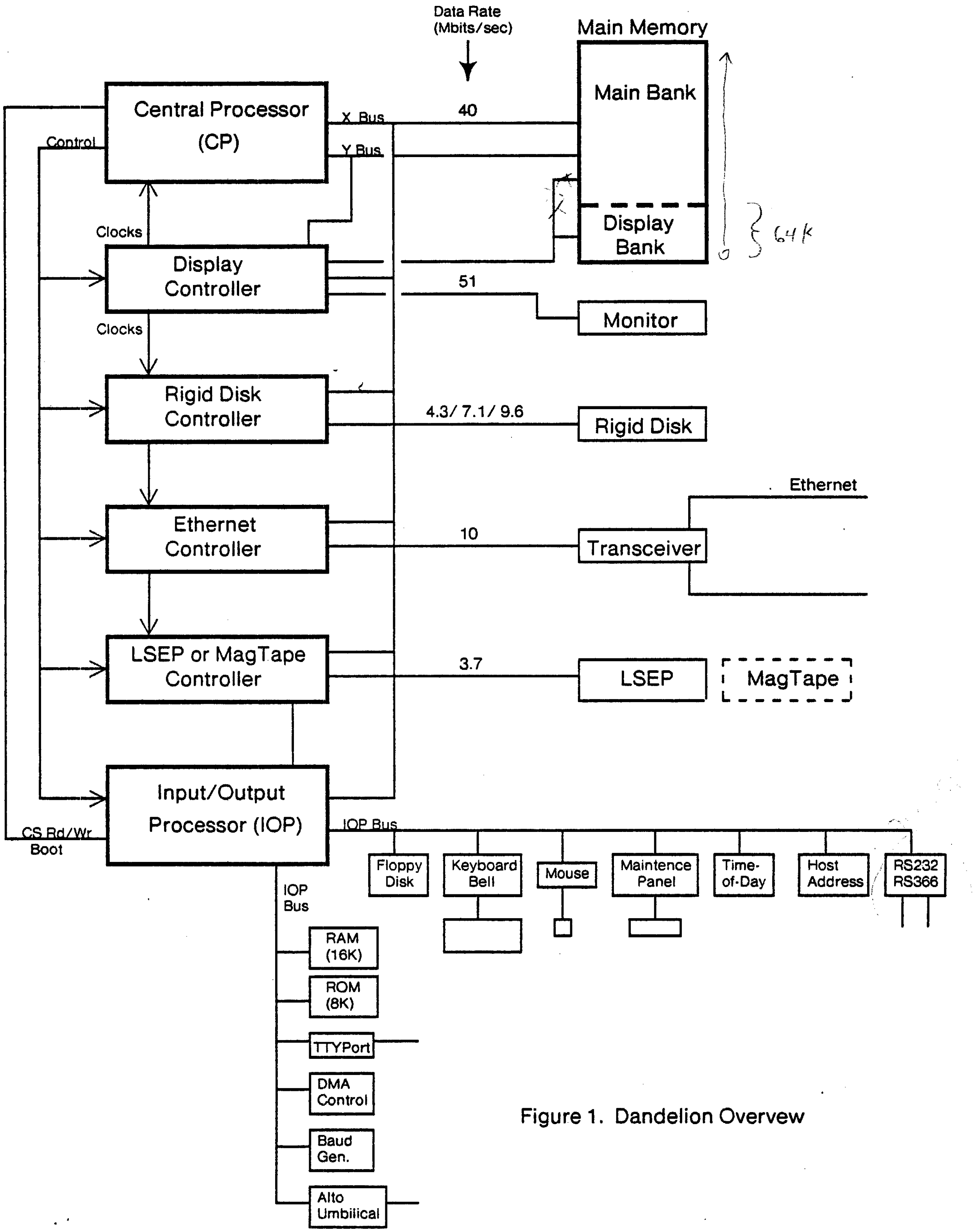


Figure 1. Dandelion Overview

## Main Memory

The Dandelion memory system is composed of two cards: the control card (MCC) and the storage card (MSC). They each come in two versions; one using 16K chips, the other using 64K chips. The versions of the cards using 64K chips are called MCC-X and MSC-X. Thus, the total memory size can vary from 64K to 768K words. The maximum size of real memory is 1024K words (20-bit real address).

The memory system has some hardware support for virtual addresses. The mapping between real and virtual page numbers is stored in a linear array (the Map) located in the real address space. The maximum size of the Map is 16K words for the MCC (22-bit virtual addresses) and 64K words for the MCC-X (24-bit virtual addresses). Smaller virtual addresses spaces can be used, reclaiming real memory space.

The memory system is logically divided into the display bank and the main bank. The display bank is the lowest 64K of memory and has two ports: one for the display controller and one for the CP. When the display is actually on and displaying bits, accesses to the display bank from the CP are reduced by 47%.

The memory is single-bit corrected and double-bit error detected. A double-bit error encountered in the Emulator causes the Emulator microcode to trap. Double-bit errors caused by I/O task microcode are recorded but cause no traps. The display controller runs the display bank at a faster rate than the CP and receives uncorrected data.

## Display and Clocks

The display controller reads words from a 51K-word bitmap in the memory display bank and produces video and synch signals for the 17" monitor. The screen has an active image of 808 lines by 1024 bits. A frame (an even followed by an odd field) is repainted 38.7 times a second. The bit rate to the monitor is 51MHz (19.6 nS). The phosphor is P40—white, with a long decay time (used in the Alto).

The controller can divide each scan line into up to seven segments. The bits shown in each segment may come from a different line in memory. Thus, windows can be scrolled vertically without having to move the bits of the window in memory. The cursor is implemented in the microcode as a 16-by-16 window (although it can be any size). In every frame, the microcode *or*s the cursor with the appropriate words in the active region, these words are written into a temporary area (in the display bank), and then this temporary area is used as the cursor window.

A total of 54 lines of top and bottom border are displayed. Each line consists of a repeated pattern generated from the controller border pattern register which can be loaded from software. There are also 32 bits of border on each end of every visible line. The display controller hardware orchestrates the horizontal line events, while the display microcode decides when the line state should change.

The display microcode also has the task of refreshing main memory. Two refresh pulses are required for each 28.8 μsec scan line with the normal controller communications. Furthermore, it maintains a 32-bit counter, incremented once per scan line, which can be used by the Emulator to measure short-duration events. The display microcode can also wake up an Emulator process at an arbitrary scan line in every field.

The CP's clocks are derived from the display clock: the display's bit period of 19.59 nS is multiplied by seven to give the CP's cycle time of 137.14 nS. There are exactly 14 rounds per horizontal display scan line.

## IOP

The Input/Output Processor (IOP) is an 8085 based processor which services the low speed I/O devices, can boot the CP, and read or write its microstore and task program counters. It is also a convenient place from which to exercise the CP and the high speed devices (e.g., floppy disk diagnostic programs).

The IOP supports the following low speed devices: (1) IBM-compatible floppy disk with both single and double density and double sided diskettes; (2) Star Level 4 Keyboard interface, mouse interface, and simple tone generator to drive the speaker in the keyboard; (3) TTY port interface (RS232C DCE) to connect a Lear Siegler terminal or Diablo 630 character printer; (4) the maintenance panel (4-digit 7-segment display plus 2 boot buttons); (5) the time-of-day clock; (6) the CP control store and task program counters; (7) the CP-IOP communication port; (8) the Alto umbilical debugging connection; and (9) the Ethernet host address PROM. In addition, located on the Option card, the IOP supports (10) the LSEP UART and baud-rate generator and (11) a Z80-SIO with RS232C and RS366 interfaces. The 48-bit Ethernet host address is located in a prom on the IOP.

The IOP has 16K bytes of RAM and four, socketed EPROMS for 8K bytes of read-only memory. The EPROM contains some simple 8085 diagnostics, the basic IOP boot supervisor, and some initial CP boot microcode for the various sources of boot files (rigid or floppy disk or Ethernet).

The IOP communicates with the CP via the CP-IOP port. This is a normal set of input/output registers in the CP's I/O system. IOP task microcode can read or write main memory with arguments supplied by the IOP through the port. The IOP can supply or accept data in a polled fashion or with DMA (one byte per 4 microseconds).

The 32-bit time-of-day clock counts seconds based on the 60 Hz power line. The clock continues to run when the Dandelion is turned off but is still plugged into the wall. This feature is disabled on current hardware.

### Rigid Disks

The HSIO board can operate either a Shugart SA1000 or a SA4000 drive, but not both and not multiple drives. The two controllers share some common circuitry; a wire in the interface cable distinguishes between the two types of drives. The SA1000 requires and expects phase encoded data, whereas the SA4000 does not. Another version of the HSIO card, the HSIO-L, can support up to 4 Trident T-300 or T-80 drives.

The following table summarizes the formatted capacity, average access time, and bit rates of the three types of drives:

| drive | capacity | avg access | bit rate |
|---|---|---|---|
| | (Mbytes) | (msec) | (MHz) |
| SA1004 | 8.38 | 80 | 4.27 |
| SA4008 | 23.17 | 75 | 7.14 |
| T-300 | 237.8 | 30 | 9.6 |

The SA1000 clock rate (234 nS/bit) is derived by dividing the display rate (51 MHz) by 12.

### Ethernet

The Ethernet controller contains: a phase encoder (based on a 20MHz crystal), a phase decoder (Phase Lock Loop), serial-to-parallel and parallel-to-serial converters, a 16-word FIFO buffer, a 9.6-μsec counter, a 51.2-μsec interval counter, and a state machine to manage the buffer among incoming and outgoing packets. The Ethernet task requires two clicks per round because of the high data rate: 1.6 μsecs per word. Moreover, the FIFO is required because of the software queue overhead and the microcode overhead required to check the destination-address field of incoming packets.

There is a single FIFO and CRC generator/checker shared between input and output. The controller hardware and microcode, to the extent possible, operate this half-duplex buffer so as not to lose packets—an incoming packet has priority over a packet being staged for transmission. The FIFO can hold more than one received packet. There is a "end-of-packet" marker maintained in the buffer.

The microcode appends packets to an input queue maintained by the Emulator in main memory. Similarly, it reads from an output queue set up by software. The microcode generates the retransmission interval when there is a collision (it uses the 51.2-μsec controller wakeup when deferring).

The controller has a special mode (which also requires special microcode) which will loopback a 15-word packet. The packet can be sent either through the transceiver or only through the phase encoder and decoder, thereby bypassing the transceiver and drop cable. In loopback mode, the CRC checker is also verified with a microcode-supplied checksum.

### LSEP

The Low-Speed Electronic Printer (laser ROS or thermal) controller has two parts on the Options card: a one-word "video" buffer, which is an I/O device controlled by the CP, and an IOP-controlled UART which is used to send and receive commands and status.

The buffer is a simple two-word, ping-pong arrangement: one shift register is loaded while the other is supplying data to the printer. The Raven microcode reads words from the display bank and then zeros them (all in one click) in order to prepare for the next page. The "video" clock is supplied by the printer. The speed of the command/status UART is set by an IOP controlled baud-rate generator.

The one click per round used by the LSEP is also shared with the special Refresh task needed when the display is off.

### MagTape

(to be added)

## Configuration

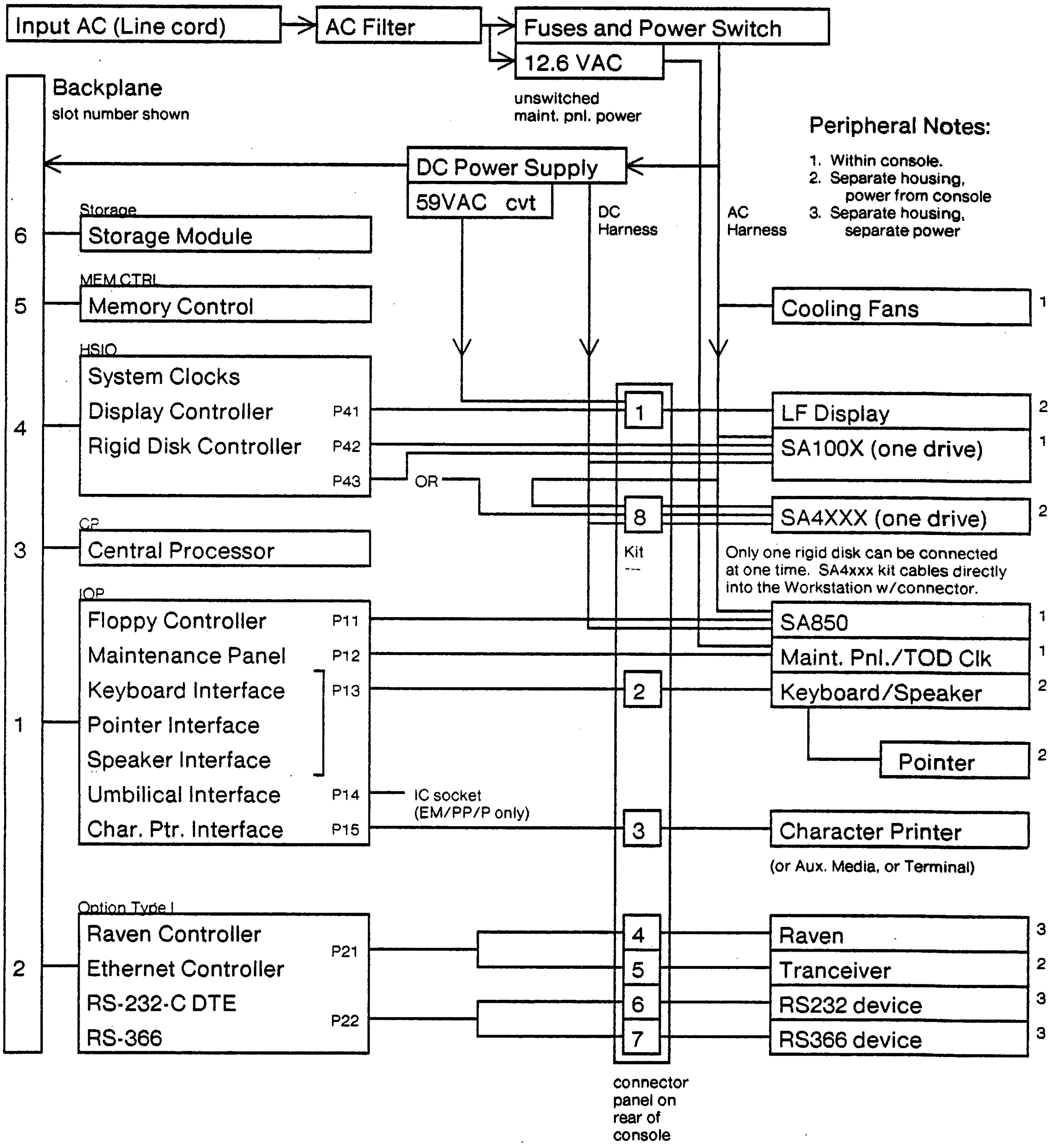
The figure on the next page shows the baseline configuration of the Dandelion using Shugart drives. The cards measure 11x14 inches. No remote power up or down is included. There is a mechanical, rocker-type, power switch on the front of the console under a cover.

Two momentary contact pushbuttons are included with a 4 digit maintenance panel under the cover with the power switch at the front of the machine. One of the buttons (boot) is hardwired to the 8085's reset line. If you push boot, the 8085 will boot the machine in the default way. If the second button (alternate boot) is held down while boot is pushed, the 8085 will slowly display a sequence of numbers in the maintenance panel which refer to alternate booting strategies. When the user releases the AltBoot button the selected strategy will take place. The strategies include booting from rigid disk, booting from floppy disk, from the Ethernet, and diagnostic boots. Note that when the power is turned on, the standard boot takes place without the need to push any buttons.

Whereas the SA1000 drive fits inside of the main cabinet, the SA4000 requires a separate housing. If power is lost in the middle of writing a disk page, the page will be lost.

Workstation Configuration

Base Line Workstation



Connector Panel Notes:

| | |
|---|---|
| 1 | D Series 25 Pin Socket |
| 2 | D Series 50 Pin Socket |
| 3 | D Series 25 Pin Socket |
| 4 | D Series 25 Pin Plug |
| 5 | D Series 25 Pin Socket |
| 6 | D Series 25 Pin Socket |
| 7 | D Series 25 Pin Socket |
| 8 | Kit |

# 2.0 Central Processor (CP)

## 2.1 Introduction

The Central Processor (CP) controls the high-speed I/O devices and the main memory of the Dandelion. It provides short-latency memory access and ALU service for the integral I/O controllers and can emulate the Mesa Processor as defined by the Mesa Processor Principles of Operation. It is composed of about 160 standard chips and resides entirely on one 11" by 15" printed circuit card located in slot 3.

This chapter presents the hardware structures of the Central Processor and its interfaces with the rest of the Dandelion. Another manual, the *Dandelion Microcode Reference (DMR)*, presents the assembler microcode format and is interspersed with hardware details and examples.[1]

The CP is a microprogrammed, 16-bit general-purpose computer. The microstore can hold up to 4096 48-bit microinstructions[2] and can be read or written by the low-speed Input/Output Processor (IOP). Each microinstruction is decoded and executed in 137 nanoseconds, a *cycle*.[3] All microinstruction operations are completed in one cycle; instruction execution is not pipelined over several cycles, except that while one is being executed its successor is being read from the microstore.

Cycles are grouped into *clicks*, where one click equals three successive cycles labeled c1, c2, and c3. Cycles are always enumerated in order c1, c2, c3, and then c1 again.[4] This sequence is never interrupted or altered; accordingly, both targets of a two-way branch must be specified with the same cycle number. (Strictly speaking, this is necessary only if the target microinstructions contain cycle-dependent operations.) The microcoder's task of aligning instructions so that they execute in successive cycles is a necessary outcome of the fixed-tasking, click structure. Moreover, when one desires code which is speed optimized, this structure usually requires the elimination of three microinstructions instead of one.

While the three microinstructions of a click are executing, a memory read or write can be performed: the address is sent to the memory in c1, a single data word may be sent during c2, and data is returned from memory in c3. A memory operation can *only* be initiated in cycle 2.

Clicks are grouped into *rounds*: five successive clicks (numbered 0..4) comprise a round, which is two microseconds in duration. Each click of a round is permanently allocated to one or more of the I/O controllers. If an I/O controller does not request the service of its corresponding *task* microcode, the Emulator-microcode task runs during that click instead of the device-microcode task. When there is a transition between tasks, the hardware preserves the outgoing task's microprogram counter and restores it when it runs again.

The click is a basic microcode time unit: devices and the Emulator are serviced in units of clicks and the microcode can transfer exactly one memory word in this time. For purposes of synchronization, the click is an atomic operation. Since a click is 411 nanoseconds in duration, the maximum memory bandwidth available through the CP is 40 Mbits/s (2.4 megawords/s).

The CP is implemented using four 2901 bit-slice chips plus external memories and registers. The 2901 provides 17 registers readily accessible to the microcoder, the usual logical and arithmetic functions, and single bit shifting.

Available to the microprogrammer and external to the 2901 are four register sets (U, RH, IB, and Link), a four-bit rotator, the I/O registers and memory, and four Emulator registers (stackP, ibPtr, pc16, and MInt). There are no task specific registers: all registers can be addressed by all tasks.

## 2.2 Microinstruction Format

The microinstruction format attempts to strike a balance between some naturally opposing constraints: control store width versus control store size, encoding schemes versus decoding hardware constraints, and coverage of all possible data operations versus exclusion of impracticable operations. The goal of the format is that frequently applied operations are encoded in the smallest number of bits. Furthermore, it was designed so that the most important Mesa Emulator and I/O operations execute in one click. The format is illustrated and summarized in Figure 2.

A 48-bit microinstruction has three major parts: 2901-control bits, miscellaneous functions, and a "goto"-address field. The field names are abbreviated as:

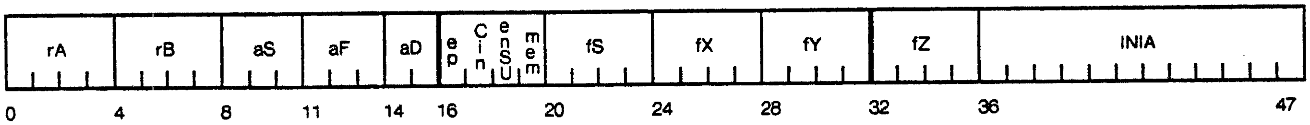
| | |
|---|---|
| rA, rB | R registers A and B |
| aS, aF, aD | ALU source address, function, destination address |
| ep | even parity |
| Cin | 2901 carry input |
| enSU | enable stack/U registers |
| mem | memory operation |
| fS | function fields selector |
| fX, fY, fZ | function fields X, Y, and Z |
| INIA | intermediate next instruction address. |

The 2901-control bits occupy the first word: rA, rB, aS, aF, and aD. The "goto" address, INIA, utilizes 12 bits. INIA is a control-store-destination address unless condition bits, specified by the previous microinstruction, are *or'd* into it, resulting in a branch or dispatch. Thus, every microinstruction is a potential jump instruction.

The fS field is broken into two subfields: fS[0-1] and fS[2-3]. These control the deciphering of the fY and fZ fields, respectively. Both the fY and fZ fields have four possible enumerations as defined by fS:

The fY field can, depending on fS[0-1]: (1) name a branch or multi-way dispatch, (2) specify a miscellaneous function, (3) name an I/O register to be loaded, or (4) equal the high nibble of an 8-bit constant. These four functions are called DispBr, fYNorm, IOOut, and Byte.

The fZ field can (1) enumerate a miscellaneous function, (2) equal a 4-bit constant or the low half of an 8-bit constant, (3) be the low half of a U register address, or (4) name an I/O register to be read. These four classes are abbreviated fYNorm, Nibble, Uaddr, and IOXIn, respectively.



| Field | Description |
|---|---|
| rA | 2901 A reg addr, U addr [0-3] |
| rB | 2901 B reg addr, RH addr |
| aS | 2901 alu Source operand pair |
| aF | 2901 alu Function |
| aD | 2901 alu Destination/shift control |
| ep | Even Parity |
| Cin | 2901 Carry In, Shift Ends, writeSU (if enSU = 1) |
| enSU | enable SU reg file |
| mem | MAR← (if c1), MDR← (if c2), ←MD (if c3) |
| fS | Function field Selector |
| fX | X Function |
| fY | Y Function |
| fZ | Z Function |
| INIA | Next Instruction Address |

| aS | R, S |
|---|---|
| 0 | A, Q |
| 1 | A, B |
| 2 | 0, Q |
| 3 | 0, B |
| 4 | 0, A |
| 5 | D, A |
| 6 | D, Q |
| 7 | D, 0 |

| aF | F |
|---|---|
| 0 | R + S + Cin |
| 1 | S − R − Cin' |
| 2 | R − S − Cin' |
| 3 | R or S |
| 4 | R and S |
| 5 | ~R and S |
| 6 | R xor S |
| 7 | ~R xor S |

| sh,,aD | B[rB]← | Q← | Ybus← |
|---|---|---|---|
| 0 | no write | F | F |
| 1 | no write | no write | F |
| 2 | F | no write | A |
| 3 | F | no write | F |
| 4 | F/2 | Q/2 | F |
| 5 | F/2 | no write | F |
| 6 | 2F | 2Q | F |
| 7 | 2F | no write | F |

sh ← (fX = shift) OR (fX = cycle) OR (fY = cycle)

| fS[0-1] | fY = |
|---|---|
| 0 | DispBr |
| 1 | fYNorm |
| 2 | IOOut |
| 3 | Byte |

| fS[2-3] | fZ = | SU addr[0-7] | |
|---|---|---|---|
| 0 | fZNorm | 0,,stackP | |
| 1 | Nibble | 0,,stackP | |
| 2 | Uaddr[4-7] | rA,,fZ \| rA,,Y[12-15]* | IF fZ = AltUaddr* |
| 3 | IOXIn | rA,,fZ \| rA,,Y[12-15]* | IF fZ = AltUaddr* |

* as executed by previous u-instr

| fX | fXNorm |
|---|---|
| 0 | pCall/Ret0 |
| 1 | pCall/Ret1 |
| 2 | pCall/Ret2 |
| 3 | pCall/Ret3 |
| 4 | pCall/Ret4 |
| 5 | pCall/Ret5 |
| 6 | pCall/Ret6 |
| 7 | pCall/Ret7 |
| 8 | Noop |
| 9 | RH← |
| A | shift |
| B | cycle |
| C | Cin←pc16 |
| D | Map← |
| E | pop |
| F | push |

| fY | fYNorm | DispBr | IOOut |
|---|---|---|---|
| 0 | ExitKern | NegBr | IOPOData← |
| 1 | EnterKernel | ZeroBr | IOPCtl← |
| 2 | ClrIntErr | NZeroBr | KOData← |
| 3 | IBDisp | MesaIntBr | KCtl← |
| 4 | MesaIntRq | PgCarryBr | EOData← |
| 5 | stackP← | CarryBr | EICtl← |
| 6 | IB← | XRefBr | DCtlFifo← |
| 7 | cycle | NibCarryBr | DCtl← |
| 8 | Noop | XDisp | DBorder← |
| 9 | Map← | YDisp | PCtl← |
| A | Refresh | XC2npcDisp | MCtl← |
| B | push | YIODisp | ←TStatus |
| C | ClrDPRq | XwdDisp | EOCtl← |
| D | | XHDisp | KCmd← |
| E | ClrRefRq | XLDisp | ←TIData |
| F | ClrKFlags | PgCrOvDisp | POData← |

| fZ | fZNorm | IOXIn |
|---|---|---|
| 0 | Refresh | ←EIData |
| 1 | IBPtr←1 | ←EStatus |
| 2 | IBPtr←0 | ←KIData |
| 3 | Cin←pc16 | ←KStatus |
| 4 | Bank← | KStrobe |
| 5 | pop | ←MStatus |
| 6 | push | ←KTest |
| 7 | AltUaddr | EStrobe |
| 8 | Noop | ←IOPIData |
| 9 | | ←IOPStatus |
| A | | ←ErrnIBnStkp |
| B | | ←RH |
| C | LRot0 | ←ibNA |
| D | LRot12 | ←ib |
| E | LRot8 | ←ibLow |
| F | LRot4 | ←ibHigh |

pCall when NIA[7] = 0. pRet when NIA[7] = 1.

Equivalent names: XDirtyDisp = XLDisp; EtherDisp = YIODisp; TAddr← = ClrDPRq; TCtl← = PCtl←; TOData← = POData←

Figure 1. Dandelion CP Microinstruction Format

## 2.3 Registers and Data Paths

Figure 2 illustrates the registers and data paths layout for the CP. The area inside the dashed lines represents the internal components of the 2901 ALU. The Y bus corresponds to the Y output of the 2901 and the X bus is connected to the 2901 D input. Both the X and Y buses are available on the backplane.

### 2.3.1 R and Q Registers and 2901 Data Paths

Figure 2 shows the 16-word, two-port register file called the R registers. One of the output ports is labeled A and the other B. These are the "fast" registers of the CP and can be used to hold temporaries, memory data and addresses, and arithmetic operands.

Every cycle, the contents of the R register given by the register-A (rA) field of the microinstruction is available at the A port, and likewise for the B port. If rA = rB, then the same data appears at both ports.

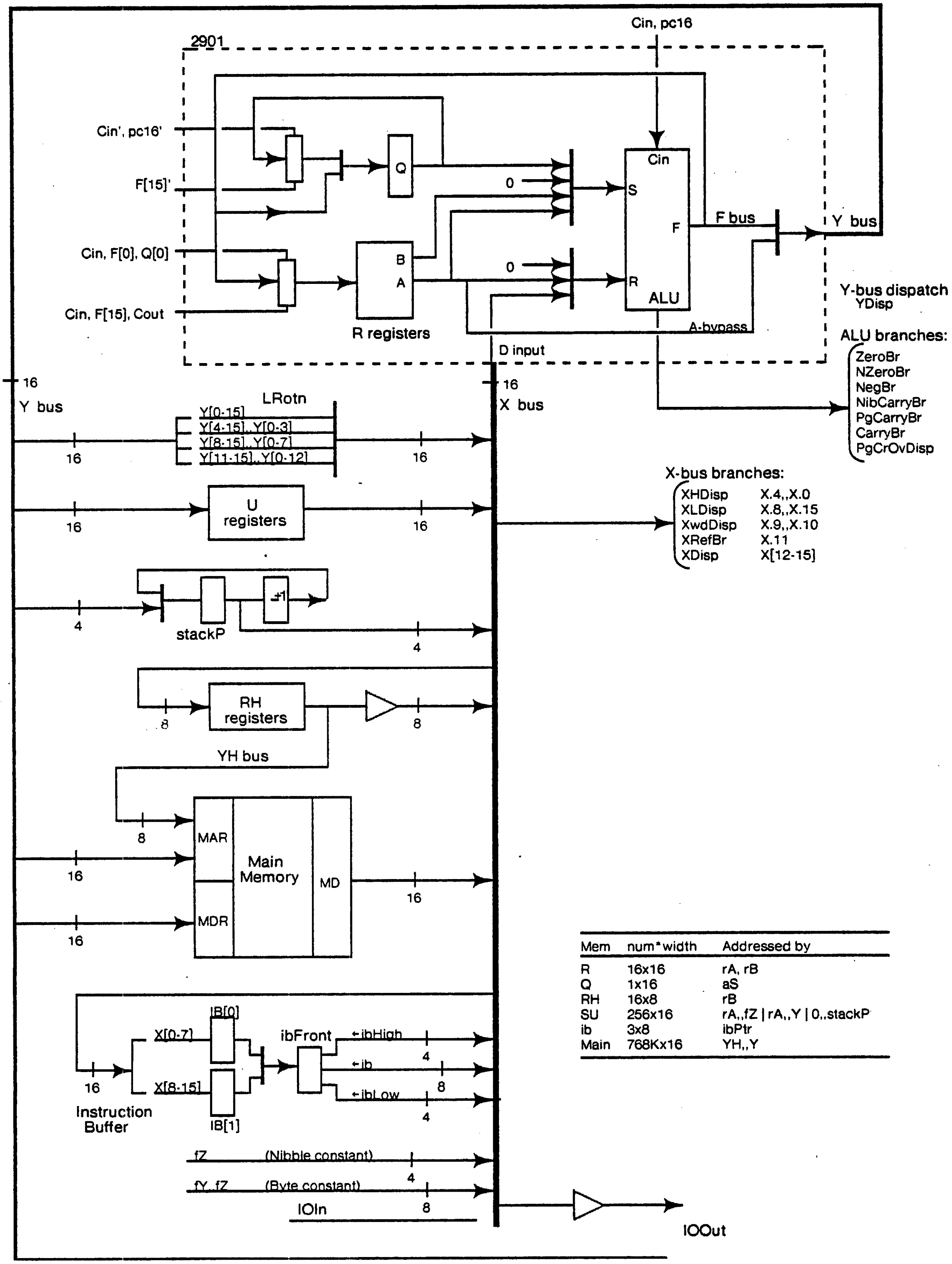
If the alu-Destination (aD) field specifies a write back into an R register, the rB field specifies which one: at the end of the cycle, register B is written with the ALU output (named F) or it is written with F shifted one bit.

The Q register holds 16 bits which can be written with the ALU output or its old value single-bit shifted left or right. It is implicitly referenced by the aS field of the microinstruction and can be used for double-word shifting.

The 2901 arithmetic unit has three inputs: R, S and Carryin (Cin). The R input can be set to the output of the A port, the value of the X bus, or zero. The S input can be driven by the output of the A or B ports, the value of the Q register, or zero. Cin can be either 0 or 1, or the value of the single-bit Emulator register pc16.

The 2901 can perform three arithmetic and five logical operations as specified by the alu-Function (aF) field. Arithmetic follows the two's-complement conventions. Three of the logical operations are symmetrical with respect to R and S: logical *or*, *and*, and *xor*. The remaining two logical operations complement R: ~R *xor* S and ~R *and* S.

Figure 3 shows a matrix of ALU computations as a function of possible aS and aF values. From the table it is clear there are many possible ways to generate zero within the ALU. All one's (0FFFF) is easily produced for some functions if rA = rB.



| Mem | num*width | Addressed by |
|---|---|---|
| R | 16x16 | rA, rB |
| Q | 1x16 | aS |
| RH | 16x8 | rB |
| SU | 256x16 | rA,,fZ \| rA,,Y \| 0,,stackP |
| ib | 3x8 | ibPtr |
| Main | 768Kx16 | YH,,Y |

Figure 2. Dandelion CP Data Paths

| aF | Cin \ aS | (A,Q) | (A,B) | (0,Q) | (0,B) | (0,A) | (D,A) | (D,Q) | (D,0) | rA = rB = R |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | (A,B) |
| R + S | 0 | A + Q | A + B | Q | B | A | X + A | X + Q | X | 2R |
| | 1 | A + Q + 1 | A + B + 1 | Q + 1 | B + 1 | A + 1 | X + A + 1 | X + Q + 1 | X + 1 | 2R + 1 |
| S-R | 0 | Q-A-1 | B-A-1 | Q-1 | B-1 | A-1 | A-X-1 | Q-X-1 | -X-1 | -1 |
| | 1 | Q-A | B-A | Q | B | A | A-X | Q-X | -X | 0 |
| R-S | 0 | A-Q-1 | A-B-1 | -Q-1 | -B-1 | -A-1 | X-A-1 | X-Q-1 | X-1 | -1 |
| | 1 | A-Q | B-A | -Q | -B | -A | X-A | X-Q | X | 0 |
| R or S | | A or Q | A or B | Q | B | A | X or A | X or Q | X | R |
| R and S | | A and Q | A and B | 0 | 0 | 0 | X and A | X and Q | 0 | R |
| ~R and S | | ~A and Q | ~A and B | Q | B | A | ~X and A | ~X and Q | 0 | 0 |
| R xor S | | A xor Q | A xor B | Q | B | A | X xor A | X xor Q | X | 0 |
| ~R xor S | | ~A xor Q<br>A xor ~Q | ~A xor B<br>A xor ~B | ~Q | ~B | ~A | ~X xor A<br>X xor ~A | ~X xor Q<br>X xor ~Q | ~X | -1 |

Figure 3. ALU Operations as a function of aS, aF, and Cin.

The F output of the ALU can be written into an R register, loaded into the Q register, or placed onto the Y bus. Although the F output is normally placed onto the Y bus, it is possible to route output-port A of the R register file onto the Y bus. This mode is called A-bypass or "A-pass-around."

The two-bit alu-Destination (aD) field, in combination with a one-bit value called sh, specifies whether R and/or Q is written and whether F or A-bypass is placed on the Y bus. The sh field is defined by certain functions of the microinstruction word (see Figure 1 for sh's definition). In general, when sh = 1 the F output is shifted one bit position before being written back into R or Q. This is accomplished inside the 2901 by 3-input multiplexers at the inputs to R and Q. What is shifted into the ends of R or Q determines the type of shift.

When sh concatenated with aD (sh,,aD) equals 001, neither an R register nor Q is written. This may be desired when writing an external register or when comparing two quantities. When sh,,aD = 000, Q is loaded with the ALU output. When sh,,aD is equal to 010 or 011, an R register is loaded with the ALU output.

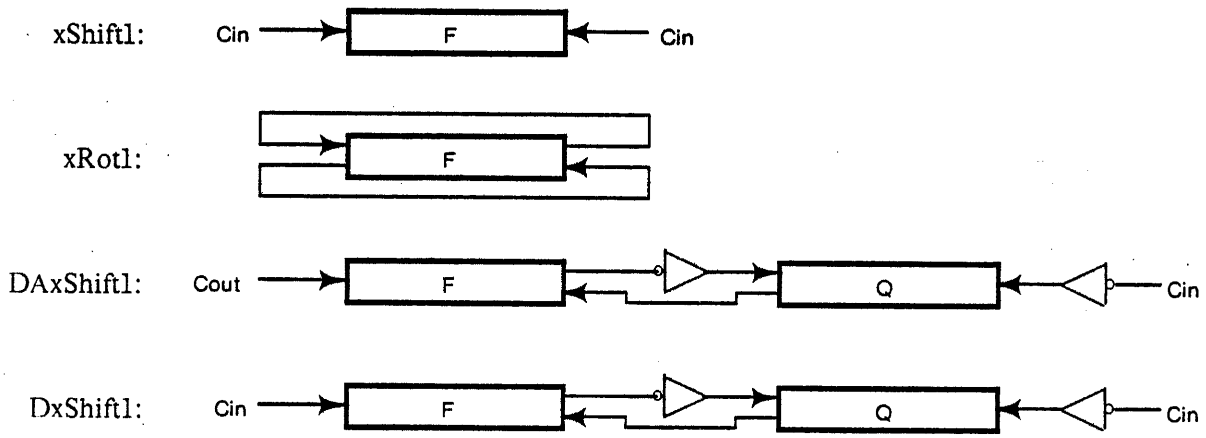
The Y bus gets the ALU output in all cases except when sh,,aD = 010, when it receives the A-bypass value. Two general rules: When A-bypass is utilized an R register must be written and it is not possible simultaneously to write R and Q with F.

When sh = 1, a single-bit shifting operation is performed on the ALU output and/or Q. There are two major types of shift operations (Figure 4): a double-word shift of F,,Q and a single-word shift of F alone. These two types of shifting, combined with the two directions, are named by the four values of aD when sh = 1.

For single-word shifts, the Q register is unaffected and the R register gets the ALU output shifted one bit to the left or right. The end of F which is vacated by the shift operation is replaced by Cin or the bit shifted out of the opposite side of F (a single bit rotate).

For double-word shifts, both the ALU output and the Q register are shifted together. The low-order bit of the ALU output is "connected" with the high-order Q bit to form a 32-bit quantity. The high-order bit of F which is vacated by a right double shift can be written with Cin or the Carryout (Cout) of the current ALU computation. Similarly, the low end of Q is written with the complement of Cin (~Cin) if the shift direction is left. Note that the high bit of Q is written with the complement of the low bit of F. A general rule: Shift inputs into Q are *complemented.*

In summary, the following 2901-related restrictions apply: (1) When A-bypass is utilized an R register must be written, (2) it is not possible simultaneously to load R and Q, and (3) A-bypass cannot be used with single bit shifts or when loading Q.



| function | aD | fX or fY |
|---|---|---|
| RShift1 | 1 | shift |
| LShift1 | 3 | shift |
| RRot1 | 1 | cycle |
| LRot1 | 3 | cycle |
| DARShift1 | 0 | shift |
| DALShift1 | 2 | shift |
| DLShift1 | 0 | cycle |
| DRShift1 | 2 | cycle |

Figure 4. CP Single-Bit Shifting

### 2.3.2 External 2901 Data Paths

There are two major 16-bit data buses external to the 2901: the X bus and Y bus. Both are present on the backplane; however, they are *not* general purpose, bidirectional buses. The YH bus, an 8-bit extension of the Y bus, is used for memory addressing.

The Y bus is driven only by the Y output of the 2901. It can be used to supply a memory address, memory data, U register data, or device output data.

The X bus is the major system bus and is connected to multiple drivers and multiple receivers.[5] X bus sinks are: the D input of the 2901, the RH registers, the Instruction Buffer (IB), and controller output registers. X bus sources are: the U registers, RH registers, the IB, constants, memory data, and controller input registers. The IB, RH, and controller output registers receive data from the X bus so that they can be loaded directly from memory in one cycle.

Data can be passed from the Y bus to the X bus via a 4-bit rotator, called LRotn. Data can be rotated zero, four, eight, or twelve positions to the left, as specified by the fZ field. A zero rotation allows Y bus data to be placed unaffected onto the X bus; an example is loading controller output registers from the ALU output.

Eight- or four-bit constants can be placed onto the X bus directly from the fY and/or fZ fields. The upper 8 or 12 bits of the X bus are set to zero.

The following table lists the registers which are addressable by the CP and the buses to which they are attached:

| Register | inputs from | Register | outputs to | |
|---|---|---|---|---|
| MAR← | YH,,Y | ←MD | X | Memory |
| Map← | YH,,Y | | | |
| IB← | X | ←ib, ←ibNA | X | Instruction Buffer |
| | | ←ibLow, ←ibHigh | | X[12-15] |
| | | ~ibPtr | X[10-11] | |
| RH← | X[8-15] | ←RH | X[8-15] | |
| U← | Y | ←U | X | |
| stackP← | Y[12-15] | ~stackP | X[12-15] | |
| MDR← | Y | EKErr | X[8-9] | |
| MCtl← | Y | ←MStatus | X | Memory |
| KOData← | X | ←KIData | X | Rigid Disk |
| EOData← | X | ←EIData | X | Ethernet |
| POData←/TOData← | X | ←TIData | X | LSEP/MagTape |
| IOPOData← | X | ←IOPIData | X | IOP |
| KCtl← | X | ←KStatus | X | Rigid Disk |
| KCmd← | X | ←KTest | X | Rigid Disk |
| EICtl← | X | ←EStatus | X | Ethernet |
| EOCtl← | X | | | |
| IOPCtl← | X | ←IOPStatus | X | IOP |
| DCtl← | X | | | Display |
| DBorder← | Y | | | |
| DCtlFifo← | Y | | | |
| PCtl←/TCtl← | X | ←TStatus | X | LSEP/MagTape |
| TAddr← | X | | | |

### 2.3.3 U Registers

A 256-word register file, called the U registers, can be written from the Y bus and read onto the X bus. These 16-bit general purpose, "slow" registers are used to hold a 16-word stack, virtual page addresses, temporaries, counters, and constants.

With respect to accessibility, U registers are situated between main memory and the R registers: they cannot be both read and written in the same cycle, nor can they be used as an operand or destination register in 16-bit ALU arithmetic.

As illustrated below, there are three ways to form an 8-bit U register address: normal, stack-pointer, and alternate.

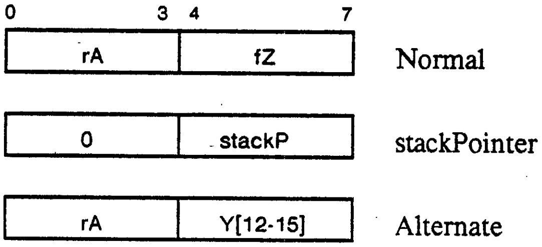


Figure 5. U Register Addressing Modes

In the normal mode, true when fS[2] = 1, the U register address is defined by the concatenation of the rA and fZ microinstruction fields. This sharing of the rA field between R and U register addresses has several implications. In general, a U register can be loaded into any R register since the rB field defines the write address. However, an arbitrary U register and an arbitrary R register cannot both be ALU operands unless the upper four bits of the U register address equal the R register address. This addressing mechanism partitions the U registers into sixteen, 16-word banks such that, in one cycle, a bank's U register can only be combined with the bank's corresponding R register.

In the stack-pointer addressing mode, used when fS[2] = 0, the U register is selected by the 4-bit stackPointer register (stackP) from the low bank; that is, the address is 0,,stackP. The stackP is *not* explicitly modified with this addressing mode and if an instruction uses this mode and also executes a pop or push function, the stackP before modification is used to access the U register.

The alternate mode provides indirect addressing and is used when fS[2] = 1 and fZ = AltUaddr for the *previously* executed microinstruction. In this mode, the low nibble of the U address equals the least significant Y bus nibble for the *previously* executed microinstruction—the same one that did the AltUaddr. Thus, instead of rA,,fZ, the U address is rA,,Y[12-15].

While reading or writing U registers, the fZ field can specify both a U register address and another function. Specifically, when fS[2-3] = 3, fZ can take on IOXIn values. This is commonly used to read an RH register or the IB while simultaneously writing a U register. When the stackPointer addressing mode is used, the fZ field is free to be interpreted as either fZNorm or a Nibble.

The U registers are also controlled by two other microinstruction fields: enSU and Cin. The enSU bit is 1 for any cycle which either reads or writes a U register. Cin must be 1 if writing, and 0 if reading. Thus, if a U register is written and the ALU function is addition or subtraction, these computations execute with Cin = 1. Note that normal two's complement subtraction implies Cin = 1.

### 2.3.4 RH Registers

Located on the X bus is the 16-by-8-bit RH register file, an extension of the R registers. The principle application of this small memory is to hold the highest-order memory address bits. Moreover, it can be utilized as general-purpose storage: for flags, counters, temporaries, and subroutine return pointers (see *DMR*).

The RH registers are addressed by the rB field, and, since this field names the R register to be written, an RH register can only be written into its corresponding R register (or the Q register).

Like the U registers, the RH registers cannot be both read and written in the same cycle. An RH register is written from the low byte of the X bus when fX = RH← and is read onto X[8-15] when fZ = ←RH. Whenever it is read onto the X bus, the high half of the bus is set to zero.

Every cycle, the 8-bit YH bus is driven with the value of the addressed RH register, thereby supplying the high order memory address bits to the Memory Control card. However, these bits are only used by the memory if a MAR← or Map← is specified. As a corollary to the rule that RH registers cannot simultaneously be read and written, an RH register cannot be loaded if the microinstruction also executes a MAR← or Map←.

### 2.3.5 Instruction Buffer

The Instruction Buffer (IB) was designed to hold up to three Emulator macroinstructions or data bytes. It is used in a first-in, first-out manner. Data loaded into the IB from the X bus can be read back onto the X bus or be used to define a 256-way dispatch in control store. The IB is loaded by special Emulator "refill" microcode (sec. 2.6.4) while the actual control of the registers is accomplished by a hardware state machine.

The IB is maintained by the Emulator in a way that guarantees all macroinstructions will find necessary code segment operands there. Furthermore, the IB is where the 256-way dispatch is made on the next macroinstruction to be executed. This dispatch (IBDisp) occurs in c2 so that the next macroinstruction begins in c1, thereby adjoining the previous one. However, when IBDisp is executed and the buffer is not full, a microcode trap occurs and the refill microcode loads the buffer with more bytes from memory. If an IBDisp is executed and there is a pending interrupt (MInt=1), special interrupt trap (IB-Refill) microcode runs instead of the refill microcode. Since the IB is so small, IBDisp's frequently trap; however, since the IB-Refill trap runs at memory speed, this scheme of supplying operand bytes to the macroinstructions is very efficient.

This scheme is efficient from both memory bandwidth and page-fault handling perspectives. In the former case, macroinstructions would otherwise have to call an operand-fetching subroutine, which would waste time becoming cycle aligned. In the latter case, macroinstructions need not worry about a page fault from the code segment. (The occurrence of a code segment page fault can add major complications to the implementation of macroinstructions since the microcode must, before processing the fault, restore the Mesa machine state to its value at the beginning of the instruction.) The IB insures that macroinstructions can always find code segment arguments present in the IB. In this sense, the IB is more like an operand data buffer than an instruction buffer.

The minimum number of bytes in the buffer required to prevent an IB-Refill trap is three (the maximum size of a Mesa macroinstruction) and they only occur between the execution of macroinstructions. The refill code completes in one click if the buffer requires two bytes and finishes in two clicks if four are needed. Because the buffer is small, the only codebytes which do not result in an IB-Refill trap are single-byte opcodes executed from even memory locations.

The instruction buffer itself consists of three 8-bit registers, called IB[0], IB[1], and ibFront. IB[0] holds the even code segment byte and IB[1] the odd. The bytes are shuffled through ibFront in even/odd, sequential order. There are four states which enumerate the location of data bytes among the holding registers. These states are indicated by the 2-bit register ibPtr and are defined below. The following diagram shows the four IB states (the cross-hatching indicates the position of the data bytes):

| state name | bytes in IB | ibPtr |
|---|---|---|
| *full* | 3 | 2 |
| *word* | 2 | 3 |
| *byte* | 1 | 1 |
| *empty* | 0 | 0 |

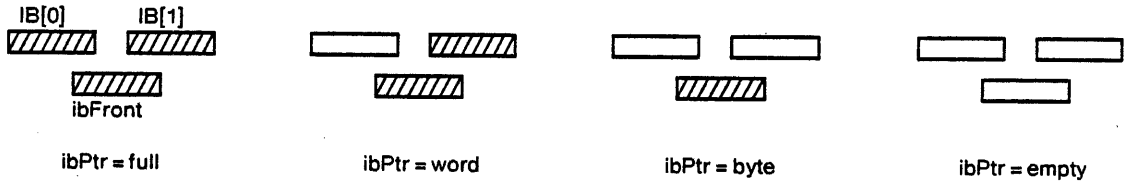


Figure 6. Instruction Buffer States

There is a total of 8 microinstruction functions which affect the IB. In general, the functions maintain the original even/odd byte ordering while updating ibPtr and ibFront. The following table lists the functions and their effect on ibPtr, ibFront, and the X bus. A discussion of the table follows, except that IB dispatches and IB-Refill traps are presented in sections 2.5.2 and 2.5.5.1.

| function | new ibPtr | new ibFront | X bus ← |
|---|---|---|---|
| ←ib | ibPtr-1 | IF ibPtr[1] = 0 THEN IB[0]<br>ELSE IB[1] | 0,,ibFront |
| ←ibNA | unchanged | unchanged | 0,,ibFront |
| ←ibHigh | unchanged | unchanged | 0,,ibFront[0-3] |
| ←ibLow | unchanged | unchanged | 0,,ibFront[4-7] |
| IBDisp | ibPtr-1 | IB[ibPtr[1]] | unaffected |
| AlwaysIBDisp | ibPtr-1 | IB[ibPtr[1]] | unaffected |
| IB← | IF empty THEN word<br>ELSE full | IF ibPtr = empty THEN X[0-7]<br>ELSE unchanged | unaffected |
| IB←, IBPtr←1 | IF empty THEN byte<br>ELSE full | IF ibPtr = empty THEN X[8-15]<br>ELSE unchanged | unaffected |
| IBPtr←0 | word | IB[0] | unaffected |
| IBPtr←1 | byte | IB[1] | unaffected |
| ←ErrnIBnStkp | unchanged | unchanged | X[10-11]←~ibPtr |

Figure 7. Effects of IB-related Functions

The IB is loaded from the X bus: the high-order, even byte is written into IB[0] and the low-order, odd byte into IB[1]. If the buffer is empty, then the X bus byte passes through IB[0] or IB[1] and is loaded directly into ibFront in one cycle; thus, the data can be used immediately in the cycle following the IB load.

The default IB write operation is to write ibFront with X[0-7]. However, if IBPtr←1 is coincident with IB←, then ibFront is written with X[8-15] instead, thereby throwing away the even data byte. If there are one or two bytes in the buffer, then IB[0] and IB[1] are loaded and there is no feed through into ibFront.

ibFront can be read onto the X bus: when the microcoder specifies a ←ib or ←ibNA, ibFront is placed onto X[8-15] and the high byte of the X bus is set to zero.

There are several variations to this basic read. With the ←ibHigh function, ibFront[0-3] is placed onto X[12-15]. Analogously, ←ibLow places ibFront[4-7] onto X[12-15]. In both cases the upper 12 bits of the X bus are set to zero.

When ←ib is executed, a funneling process occurs: ibFront is loaded with the next byte from either IB[0] or IB[1] and ibPtr is "decremented" by one. ibPtr is gray code decremented: 2, 3, 1, and then 0. Thus, the low order bit of ibPtr divides the values of ibPtr into two classes with respect to refill: empty and not empty. (This scheme equates the empty and full states, but note that the buffer is not full when the IB-Refill trap occurs.)

Several of the microcode functions have no effect on the state of the buffer: The ←ibNA function (used to read the IB without advancing ibPtr), ←ibHigh, and ←ibLow do not change ibPtr. Also, like the RH and U registers, it is not possible simultaneously to read and write IB; hence, the combination of IB← and ←ib in the same cycle does nothing.

The functions IBPtr←0 and IBPtr←1, when used alone, merely load ibFront from IB[0] or IB[1], respectively. They typically occur in the cycle after the IB has been loaded with a jump-target codebyte, thereby selecting the even or odd destination opcode.

The complement of ibPtr can be read onto X[12-13] with the ←ErrnIBnStkp function.

### 2.3.6 stackP Register

The 4-bit stack pointer, stackP, is used to address one location from U register bank 0 (Sec. 2.3.3) and can be incremented or decremented independently of the 2901. The pop function decrements (modulo 16) and the push function increments (modulo 16) the stackP at the end of a cycle. Unlike the U and RH registers, the stackP can be read and written in the same cycle.

The stackP can be loaded from Y[12-15] with an fY function. However, one cycle must intercede between a stackP← and a microinstruction which uses the stack-pointer addressing mode and expects the new value. A pop or push can be used in the intervening instruction and appropriately modifies the value loaded.

The pop and push functions have been sprinkled throughout the microinstruction function fields to ameliorate the checking of stack overflow or underflow. The push function occurs in all three function fields while pop is in fX and fZ. An outcome of this arrangement is that when push is specified in the same microinstruction as pop, the stackP does not change: it does not matter how many pop's or push's there are; as long as there are both, the stackP is unaffected. Also, multiple pops or pushs in the same instruction do not decrement or increment the stackP by more than one. Multiple pop and push functions are used to check for stack overflow or underflow (sec. 2.5.5.2).

### 2.3.7 pc16 Register

The pc16 register is designed to serve as a low-order, 1-bit extension of an R register; namely, the R register which holds the Emulator's macroprogram counter (PC). That is, pc16 can be used as the byte index of a PC memory address.

If fX or fZ is Cin←pc16, the pc16 bit becomes the carry input of the 2901 and pc16 is inverted at the conclusion of the cycle. Thus, Cin←pc16, in combination with ALU addition and subtraction, properly adjusts the 17-bit byte program counter PC,,pc16 (See *DMR*).

Since Cin is also the shift ends (Sec. 2.3.1), Cin←pc16 can be used to shift pc16 into the low-order bit of an R register in one cycle, thereby reconstructing a byte program counter in an R register.

Due to the hardware implementation of the carry input, when the Cin field of the microinstruction is 0, the fX version of Cin←pc16 must be used. If Cin = 1, then either the fX or fZ version of Cin←pc16 can be specified.

### 2.3.8 Timing Limitations

The architecture of the CP allows the execution of microinstructions which will not always properly complete. This is due to either "slow" X bus operands or "slow" destination registers; that is, certain sources can not be loaded into certain destinations because the source value is not stable in time. Basically, the delay time of the source plus the setup time of the destination must be less than the cycle time, 137 nS. MASS will flag such instructions with a timing violation error.

All ALU internal register-to-register operations complete on time. All Y bus destinations can be loaded as a result of any ALU operation which does not use the X bus as an operand (except for the high 12 bits of a U register).

If the ALU operation uses an X bus operand (aS = D,A, D,Q, D,0), depending on the register, the operation may not complete in time. In general, all X bus sources can at least be loaded into an R register, which is a logical operation (aS = D,0, aF = R*or*S).

Figure 7 should answer the question: "Is a microinstruction legal with respect to X bus timing?" The table deals with all possible X bus sources and destinations: X-bus-source-to-X-bus destination, X bus ALU operands (aS = D,A, D,Q, D,0), and X bus branching and dispatching. Intersections marked with a full, half, or quarter square blob indicate legal source/destination combinations or branching phrases.

X + R represents the 3 arithmetic operations (aF = R+S, S-R, R-S) and X or R the 5 logical operations (aF = R*or*S, R*and*S, ~R*and*S, R*xor*S, ~R*xor*S). B← implies the loading of an R register; Q← has the same timing. pgCross refers to the automatic page cross branch with MAR← and pageCross & OVR refer to PgCrOvDisp.

Branching and dispatching have different timing than the basic ALU operations and a potential statement must meet both conditions. In general, zero, negative, or overflow branching is not possible with any X bus operand.

The ALU performs arithmetic at three different speeds depending on which bits of the result you're looking at. Thus, figure 7 has three numbers for arithmetic operations depending on which bits of the result are of interest. ALU[0-7] are the slowest since they depend on a carry from the lookahead unit. ALU[8-11] are next as they depend on a ripple carry from the low nibble. Finally, ALU[12-15] are fastest since Cin arrivies very early relative to X bus sources. Thus, the low nibble always has the timing of a corresponding ALU logic operation.

Note that some "+1" or "-1" operations do not necessarily imply use of the X bus, but use Cin instead. Thus, R ← R + 1, NegBr is legal where R ← R + 2, NegBr is not.

All arithmetic operations with the ALU internal zero as an operand (aS = 0,Q, 0,B, 0,A, or D,0) complete on time. This obviously includes all X bus sources.

|  |  | X setup | X Source |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  | U | MD | RH | constant,<br>←ib<br>ErrIBStkp | IOIn | A LRotn | (A or B)<br>LRotn | (A + B)<br>LRotn |
|  | X Source Time |  | 75 | 97 | 74 | 59 | 63 | 91 | 102 | 131<br>127<br>105 |
| X Operation | B←X or R | 47 | ■ | ■ | ■ | ■ | ■ | ■ | — | — |
|  | B←X or R, ZeroBr | 63 | ■ |  | ■ | ■ | ■ |  | — | — |
|  | B←X or R, NZeroBr | 68 |  |  |  |  |  |  |  |  |
|  | B←X or R, NegBr | 62 | ■ |  | ■ | ■ | ■ |  | — | — |
|  | []←X or R, YDisp | 71 |  |  |  | ■ | ■ |  | — | — |
|  | B←LShift1 (X or R) | 58 | ■ |  | ■ | ■ | ■ | — | — | — |
|  | B←LRot1 X (X or R) | 66 | ■ |  | ■ | ■ | ■ | — | — | — |
|  | MAR← X or R | 78 |  | — |  | ■ |  | — | — | — |
|  | Map← X or R | 78 |  | — |  | ■ |  | — | — | — |
|  | MDR← X or R | 45 | ■ | — | ■ | ■ | ■ | — | — | — |
|  | U ← X or R | 87 | — |  |  |  |  | — | — | — |
|  | IOYOut← X or R | 64 | ■ |  | ■ | ■ | ■ | — | — | — |
|  | B←X + R | 78<br>71<br>47 | ◤ | ◤ | ◢ | ■ | ■ | ◤ | — | — |
|  | B←X + R, ZeroBr | 97 |  |  |  |  |  |  | — | — |
|  | B←X + R, NegBr | 91 |  |  |  |  |  |  | — | — |
|  | B←X + R, OVR | 90 |  |  |  |  |  |  | — | — |
|  | B←X + R, CarryBr | 81 |  |  |  | ■ |  |  | — | — |
|  | B←X + R, NibCarryBr | 60 | ◤ |  | ◤ | ◤ | ◤ |  | — | — |
|  | B←X + R, PgCarryBr | 65 | ◢ |  | ◢ | ◢ | ◢ |  | — | — |
|  | B←X + R, pageCross | 77 |  |  |  | ◢ | ◢ |  | — | — |
|  | MAR←X + R, pgCross | 72 | ◢ |  | ◢ | ◢ | ◢ |  |  |  |
|  | B←X + R, YDisp | 71 |  |  |  | ◢ | ◢ |  | — | — |
|  | B←RShift1 (X + R) | 91<br>87<br>56 | ◤ |  | ◤ | ◤ | ◤ | — | — | — |
|  | B←RRot1 (X + R) | 101<br>97<br>66 | ◤ |  | ◤ | ◤ | ◤ | — | — | — |
|  | MAR← X + R | 78 |  | — |  | ■ |  | — | — | — |
|  | Map← X + R | 110 |  | — |  |  |  | — | — | — |
|  | MDR← X + R | 77<br>70<br>48 | ◤ | — | ◤ | ■ | ■ | — | — | — |
|  | U ← X + R | 119<br>114<br>90 | — |  |  |  |  | — | — | — |
|  | IOOut ← X + R<br>stackP← | 80<br>73<br>51 | ◤ |  | ◤ | ■ | ◢ | — | — | — |
|  | Xbus ← X, XDisp | 32 | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ◤ |
|  | RH ← X | 36 | ■ | ■ | ■ | ■ | ■ | ■ | ■ |  |
|  | IB ← X | 37 | ■ | ■ | ■ | ■ | ■ | ■ | ■ |  |
|  | IOOut ← X | 32 | ■ | ■ | ■ | ■ | ■ | ■ | ■ | ◤ |

■ Timing OK across 16 bits of result ◢ OK across low byte ◤ OK across low nibble

Figure 7. Allowable X-bus Operations

## 2.4 Main Memory Interface

This section discusses the interface between the CP and the memory system. As outlined earlier, a memory address is sent to the Memory Controller in c1, any data to be written is sent during c2, and returning data is available in c3. Every click is a potential memory operation: if the Emulator kept the memory 100% busy and there were no I/O, it would have available up to 2.4 megawords/s (38 Mbits/s) of bandwidth.

The memory system accepts two types of addresses: real or virtual. Real references result in a read or write to the addressed location itself. Virtual references cause the memory system to ignore the low byte of the address and then, using the remaining 16 bits, read or write the Map, located at real address 10000 hex.

For both reference types, when the mem field is set in c2 a write occurs (MDR←) and when set in c3 a read occurs (←MD). If both a read and write are specified in the same click, the original value is returned and then the location is overwritten. Furthermore, if a click specifies a MDR← or ←MD without a corresponding MAR← then memory is not written and a potential memory Error trap does not occur.

As outlined in section x.xx, the memory system is available in a variety of sizes: real address size from 192K to 768K words and virtual address size from 4 to 16 megawords. This section assumes the maximum of both ranges: 20-bit real addresses and 24-bit virtual addresses.

### 2.4.1 Real Address References

When the mem bit is true in cycle 1, a real reference is caused. The microcoder specifies a real reference by using the MAR← macro in c1. The memory address is sent to the Memory Control card on the YH and Y buses. The Y bus can be driven from either the 2901's F bus or A-bypass; hence addresses can be either pre or postmodified. The YH bus, which supplies the high-order address bits, is always driven by the RH register addressed by rB. Furthermore, YH[0-3] are ignored by the memory.

Several important things happen with a MAR←: the 2901 is divided such that the high half executes a fixed function. a special "address-overflow" branch is enabled, and an MDR← or IBDisp in the next cycle is canceled if the branch is taken. Moreover, if a MAR← is executed with YH[4-7] = 0 and the display controller is enabled and actually transferring bits to the monitor, then the click is suspended (See sec. 2.5.6.5).

MAR← Effect: Split 2901

If mem = 1 in c1, the 2901 is divided such that the high half executes with its aS and aF inputs equal to (0,B) and (aF *or* 3), while the low half executes the aS and aF values given by the microinstruction. This causes the high byte of the ALU output to equal the high byte of the R register addressed by rB (or its complement if aF is in [4..7]). Thus, assuming the Y bus is driven from the F bus, the 20-bit real address is rhB[4-7],,rB[0-7],,F[8-15].

This change in normal ALU function was required by the fact that the most significant memory address bits must be ready very early in the click. Only logical operations would allow the address to pass through the ALU quickly enough. The requirements are not so strict on the low order bits, so arithmetic operations are allowed on the bottom byte. This change also facilitates the combining of the virtual page number returned by a Map reference with the offset into the page contained in the low byte of an R register (see the *DMR* for examples).

An outcome of this bipartition is that a carry out from the low half does not propagate into the high half: the high byte of rB remains unchanged after a MAR← (unless aF is in [4..7]), even if A-bypass is utilized.

The real address modes are illustrated below. In summary, if A-bypass is not used, the upper 12 bits of the memory address (the page address) come from the RH/R pair named by the rB field, while the lower 8 bits (the page displacement) are defined by the desired ALU operation. This feature can be used to combine the real-page number, as read from the Map in the previous cycle, with a displacement into the page. If A-bypass is specified, the lowest 16 address bits come from the R register addressed by rA. Hence, the 20-bit real address is rhB[4-7],,rA[0-15].

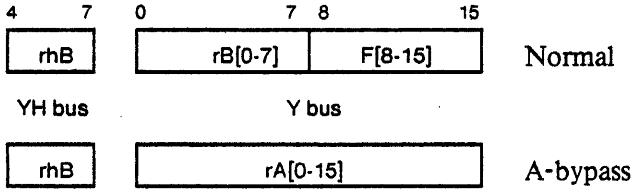


Figure 8. MAR Address Types

### MAR← Effect: pageCross Branch

The second effect of a MAR← is that it automatically specifies a pageCross branch: 1 is *or'd* into INIA[10] if the ALU operation results in a carry out from the low half. Thus, although the carry out from the low byte does not propagate into the high byte, as discussed above, it can be detected as a transfer of control. A true pageCross branch can imply that the real address is invalid and that a remapping of the virtual address which originally generated it is necessary. Since pageCross is not *or'd* into INIA[11], other simple branches can be concurrently specified.

pageCross is defined to be (pageCarry *xor* aF[2]), where pageCarry is the carry out from the low 2901 byte. The *xor* has the effect of toggling pageCarry when doing subtraction; pageCross equals pageCarry when doing addition. The aF = (R-S) form of subtraction does not cause pageCarry to be inverted since aF[2] = 0; however, the aF = (R-S) form covers the most common subtraction requirements. See the *DMR.*

A complication of the MAR← automatic pageCross branch is that pageCross can indeed equal 1 if the 2901 executes a logical, instead of arithmetic, function. See the *DMR.*

### MAR← Effect: Cancelation of c2 Functions

The third effect is that if pageCross = 1 during a MAR←, then a following MDR←, IBDisp, or AlwaysIBDisp in c2 is ignored. This mechanism can be used to prevent writing into the wrong page or dispatching on the next Emulator instruction when the corresponding virtual address should be remapped. This effect increases the need to avoid logic functions during a MAR←. See the *DMR.*

### 2.4.2 Virtual Address References

When either the fX or fY fields equal Map← in cycle 1, a memory reference to the virtual-to-real, page-translation Map is caused. The Map is a table whose first entry is at location 10000 hex, just after the display bank. During a Map reference, the memory system uses the upper 16 bits of the virtual address (14 bits in the case of a 22-bit virtual address) to index into the table. Each entry of the table contains a 12-bit real-page number and four flags pertaining to the virtual page. Currently, a 16K table is used by the Emulator. Figure 10 illustrates the process.

The virtual address is made available to the Memory Control card on the YH and Y buses. The low byte of the Y bus is ignored and, unlike MAR←, there are no ALU side effects. Since the Y bus can be driven from either the 2901's F bus or A-bypass, addresses can be either pre or postmodified:

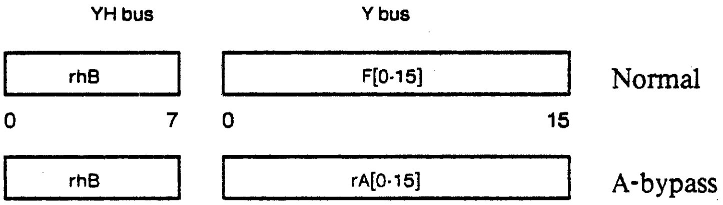


Figure 9. Map Address Types

For 24-bit virtual references, all of the YH bus is used. However, with early versions of the CP, which assumed a maximum 22-bit virtual address, if either YH[0] or YH[1] are 1, an Error trap resulted.

The following figure shows the format of a Map entry. See the *DMR* for a description of how the referenced, dirty, and present Map flag bits are maintained.

The mem field should not be set in c1 along with a Map← unless MAR←'s side effects are explicitly desired. Moreover, if YH[4-7] = 0, such clicks will be suspended due to display bank contention.

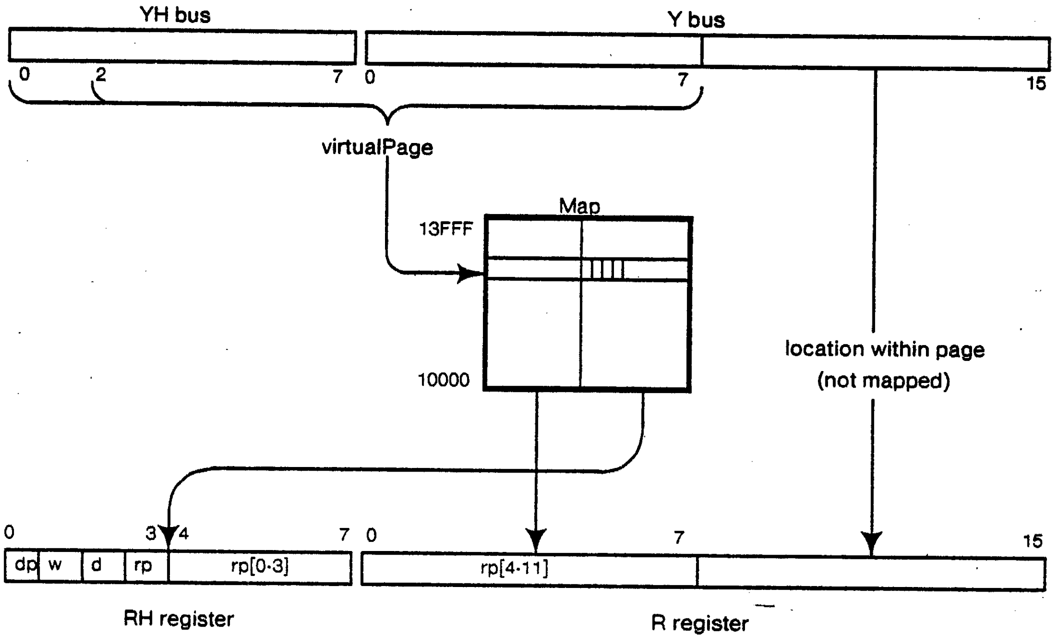
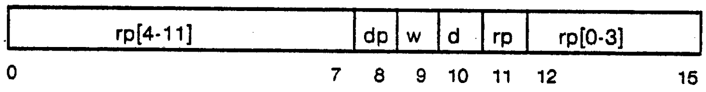


Figure 10. Virtual to Real Address Mapping



| | |
|---|---|
| rp[0-11] | Real Page Number |
| dp | Dirty & Present flag |
| w | Write Protect flag |
| d | Dirty flag |
| rp | Referenced & Present flag |

Figure 11. Map Entry Format

## 2.5 CP Control Architecture

This chapter discusses the algorithms used for controlling the execution of microinstructions and the interface between the IOP and the CP. Figure 12 is a block diagram of the control paths and registers.

As presented in the introduction, cycles are illimitably executed c1, c2, and c3. Every cycle, one microinstruction is decoded and executed while the next is being read from the control store (except in those clicks which have been suspended due to display bank contention). Since a device task does not execute in consecutive clicks, there is hardware to save the microprogram counter of each task while it is not running.

We first look at branching, dispatching, the Link registers, and the Error traps, as they can be described without reference to the tasking structure.

### 2.5.1 Conditional Branching and Dispatching

Every microinstruction can potentially branch: during each cycle, condition bits specified by the executing microinstruction are *or'd* into the next instruction's "goto"-address field (INIA) being read from control store. At the end of the cycle, this results in an address (NIA) which is used to read the next microinstruction. If the executing microinstruction does not specify a branch function, then 0 is *or'd* into INIA and, accordingly, a branch does not occur. When a microinstruction specifies a dispatch function, up-to-four bits are *or'd* into the INIA field; selecting one of up-to-sixteen target microinstructions. (The maximum of four dispatch bits was chosen in order to minimize the number which must be saved between task switches.)

Thus. all branches and dispatches take two cycles to complete: one cycle to specify the branch and one to read out the target microinstruction. The microinstruction bits required to specify a branch are fS[0-1] = DispBr and the fY field which names the branch or dispatch (Figure 13).

The notation used to specify the branching behavior is as follows: A microinstruction is located in control store at its Instruction Address. IA; the Next Instruction Address. NIA, is the control store address register; and the Intermediate Next Instruction Address, INIA, is the 12-bit "goto" address present in each microinstruction. Every cycle, the hardware *or's* the condition bits specified by fY (abbreviated DispBr) and together with a Link register specified by fX into INIA, thereby producing the NIA value used for the next cycle:

NIA[0-11] ← INIA[0-11] or DispBr[0-3] or Link[0-3].

In the case of dispatches. it is not always necessary for the microcoder to provide target instructions for each possible outcome. Any particular condition bit can be ignored by placing a 1 in its corresponding position in INIA. This method can also be used to cancel unwanted, pending branches. See the *DMR*.

Figure 13 enumerates the available branches and dispatches. Note that, in some cases, there is more than one way to branch on a particular bit and that any bit on the low half of the X bus can be branched on. The NZeroBr exists so that code can be more readily shared.

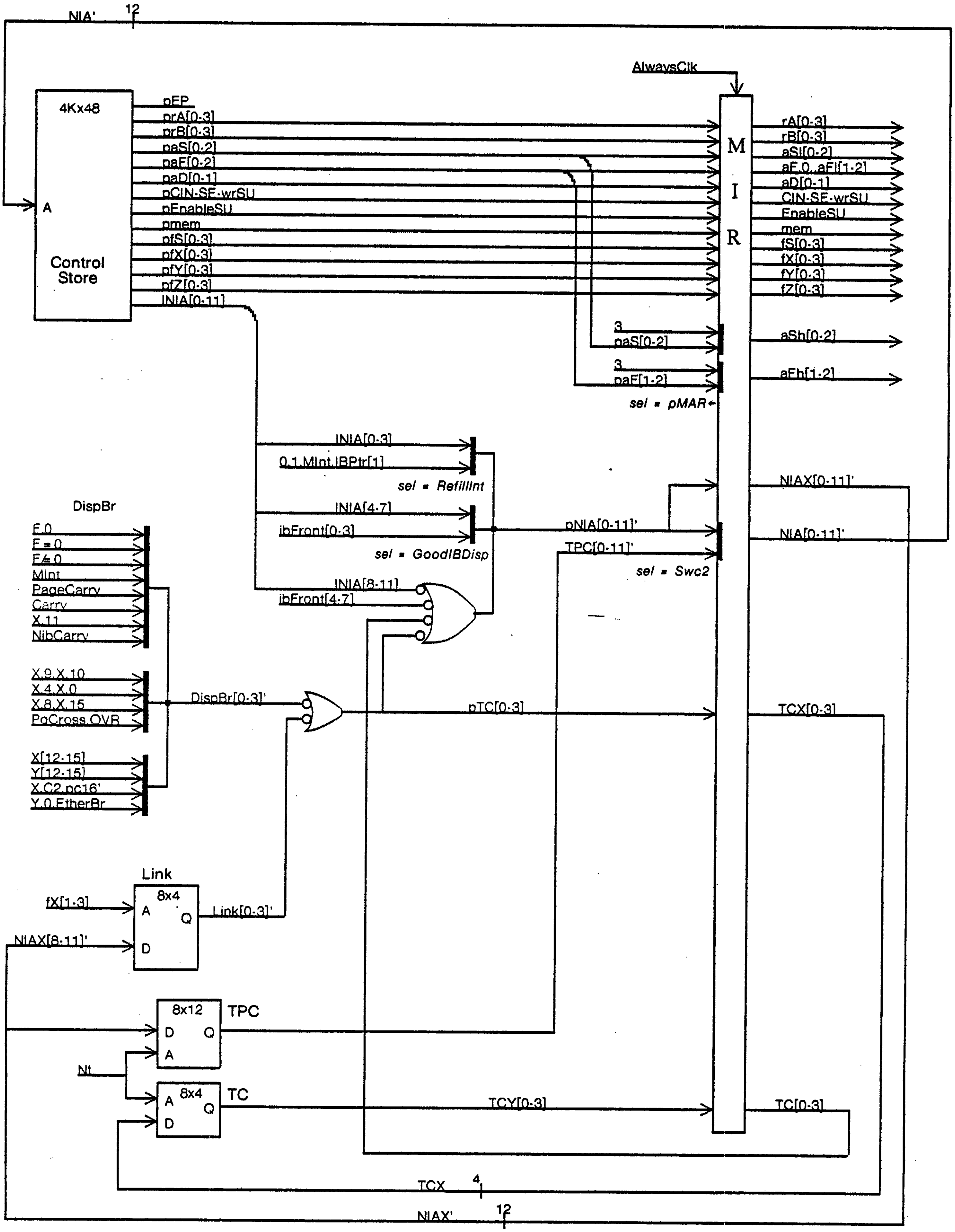


Figure 12. CP Control Paths

| | source | INIA | |
|---|---|---|---|
| NegBr | F[0] | 11 | sign of alu result (not necessarily Y[0]) |
| ZeroBr | F=0 | 11 | alu output equal to zero |
| NZeroBr | F≠0 | 11 | alu output not equal to zero |
| CarryBr | Cout[0] | 11 | alu carry out |
| NibCarryBr | Cout[12] | 11 | alu carry out from low nibble |
| PgCarryBr | Cout[8] | 11 | alu carry out from low byte |
| XRefBr | X[11] | 11 | present & referenced Map bit |
| MesaIntBr | MInt | 11 | Emulator Interrupt (see 2.5.3) |
| XwdDisp | X[9],,X[10] | [10-11] | write protect & dirty Map bits |
| XHDisp | X[4],,X[0] | [10-11] | X (high) bus |
| XLDisp | X[8],,X[15] | [10-11] | X (low) bus |
| PgCrOvDisp | PgCross,,OVR | [10-11] | pageCross & alu overflow |
| XDisp | X[12-15] | [8-11] | low nibble of X bus |
| YDisp | Y[12-15] | [8-11] | low nibble of Y bus |
| XC2npcDisp | X[12-13],,c2,,~pc16 | [8-11] | X bus, cycle2, inverse of pc16 |
| YIODisp | Y[12-13],,bp[39],,bp[139] | [8-11] | I/O branches (bp = backplane pin) |
| IBDisp | ibFront | [4-11] | Instruction Buffer |
| LnDisp | Linkn | [8-11] | Link register (n = 0..7) |

Equivalent names: EtherDisp = YIODisp, XDirtyDisp = XLDisp.

Figure 13. Branches and Dispatches

### 2.5.2 Instruction Buffer Dispatch

The instruction buffer dispatch, IBDisp, is a special dispatch since more than four bits are *or'd* into INIA. Consequently, IBDisp can only occur in c1 or c2, and, by convention, it is restricted to c2. See section 2.3.5 for a discussion of the instruction buffer.

Assuming that the instruction buffer is full, IBDisp can cause a 256-way dispatch based on the value of ibFront: NIA[4-7] is set to the high nibble of ibFront and the low nibble of ibFront is *or'd* with INIA[8-11]. (Due to the *or* operation into the low nibble of INIA, simultaneous Link register dispatches are possible.[6]) INIA[0-3] is unaffected by the IBDisp (except by the four IB-Refill trap values); therefore, up-to-twelve 256-way dispatch tables can be concurrently used.

If the buffer is not full (ibPtr ≠ full) when an IBDisp is executed, or there is a pending interrupt, then an IB-Refill trap occurs (See 2.5.5.1).

A special version of IBDisp, called AlwaysIBDisp, never IB-Refill traps: AlwaysIBDisp dispatchs on ibFront even if there is a pending interrupt (MInt = 1) or the buffer is not full. It is used in the Emulator refill and jump microcode (sec 2.6.4) to dispatch on ibFront while the buffer is still being filled. AlwaysIBDisp is encoded as fY = IBDisp and fZ= IBPtr←1.

If the microinstruction executed before an IBDisp or AlwaysIBDisp causes an IB-Empty Error trap, or it contains a MAR← and the 2901 computation results in pageCross = 1, then the IB dispatch (or possible IB-Refill trap) does not occur and ibPtr remains unaffected. Since INIA is not modified in this case, control transfers to the first entry of the macroinstruction dispatch table. (Accordingly, Emulator opcode 0 should not be assigned to a macroinstruction.)

### 2.5.3 MInt Register

The 1-bit MInt register can be used to interrupt the--contiguous execution of Emulator macroinstructions. When MInt is set in a antecedent cycle, IBDisp traps instead of dispatches (1.5.5.1). MInt is set with fY = MesaIntRq and cleared with fY = ClrIntErr. (ClrIntErr also resets the EKErr register.) See the *DMR* for user conventions.

### 2.5.4 Link Registers

The CP has eight, 4-bit Link registers which can be loaded from the low four bits of the control store address. Generally, these Link registers can be used to hold four bits of state information derived directly from the flow of control. Thus, previously determined state information can be easily recalled by dispatching on a Link register. Moreover, macroinstructions can share common code at various stages of their execution and Link registers can be used for subroutine call and return structures. See the *DMR*.

The Link register addressed by fX is written with the low nibble of NIAX (which equals NIA except during a task switch in c2. see 2.5.6.4). A Link register is written when fX is in [0..7] and NIA[7] = 0: Link[fX] ← NIAX[8-11].

A Link register is *or'd* into the low nibble of INIA when fX is in [0..7] and NIA[7] = 1, causing a potential 16-way dispatch. Since the Link register is designated by an fX function, the fY field is free to specify other condition bits which can be *or'd* into INIA[8-11].

If the preceding microinstruction does not specify a branch or dispatch condition, then the Link register is loaded with a constant. However, if the prior instruction contains a branch or dispatch, the value loaded depends on the outcome of the branch or dispatch. (The low four bits of the IB dispatch value can also be recorded in this way.) See the *DMR*.

### 2.5.5 Microcode Traps

There are two general classes of microcode traps: IB-Refill and Error. The former only occurs as the result of IBDisp's; hence between the execution of macroinstructions. There are four IB-Refill trap locations which are a function of ibPtr and MInt. Error traps can occur in any cycle and always trap to location 0 in c1. The Error traps have priority over the IB-Refill traps and cannot be disabled.

#### 2.5.5.1 IB-Refill Traps

If an IBDisp is executed and ibPtr ≠ full or MInt = 1, then the ibFront dispatch does not occur and instead an IB-Refill trap is caused. Specifically, ibPtr is unaffected, INIA[4-11] is not modified, and NIA[0-3] is set to the 4-bit quantity 0,,1,,MInt,,ibPtr[1]. The following table summarizes the interpretation of the IB-Refill trap locations. (If an IB-Refill trap occurs and MInt = 0, then ibPtr can not equal full.)

| NIA[0-3] | MInt | ibPtr |
|---|---|---|
| 4 | 0 | empty |
| 5 | 0 | not empty (i.e., byte or word) |
| 6 | 1 | empty or full |
| 7 | 1 | byte or word |

AlwaysIBDisp never IB-Refill traps and a MAR← caused pageCross branch or IB-Empty Error trap cancels a potential IB-Refill trap.

#### 2.5.5.2 Error Traps

Error traps can result when one or more predefined error conditions are detected in the CP or memory. All error traps cause the instruction at microstore location 0 to be executed in c1 by the Emulator or Kernel, depending on the error type. Error traps cannot be disabled.

The EKErr register, read onto X[8-9] with ←ErrnIBnStkp, names the type of error:

| EKErr | Type |
|---|---|
| 0 | control store parity error |
| 1 | Emulator memory error |
| 2 | stackPointer overflow or underflow |
| 3 | IB-Empty error |

If, coincidentally, two or more errors occur at the same time, smaller values of EKErr are reported. The error types are also accumulated until EKErr is reset: the minimum value is reported when EKErr is read. Error traps have priority over the IB-Refill trap. See the *DMR* for example error-handling microcode.

EKErr is reset by the ClrIntErr function which, as a side effect, also resets any pending interrupts.

With early CP modules, an EKErr value of 1 can also imply that a 23- or 24-bit virtual address had been used by the Emulator. In this case, the ErrorLogging register in the Memory Controller is read to determine whether the error is actually a double-bit memory error. Since the Memory Controller can now accept 24-bit virtual addresses, this interpretation of EKErr = 1 is no longer necessary (beginning with CP etch 4, Rev N).

### CS Parity Error

If the parity of a microinstruction read by any task is odd, then control is transferred to location 0 at the Kernel task level. Since the Kernel is the highest priority task, no other microcode tasks can execute. The CS-parity-error signal is sampled by the IOP, which can consequently sense a failed control store chip.

If the instruction read from microstore in c1 has bad parity, then the Kernel runs at location 0 in the next c1. If the parity error occurs in c2 or c3, then there is a one click delay before the Kernel executes at location 0 in c1. This intervening click can be executed by any task.

### Emulator Memory Error

If the Memory Controller indicates a double-bit memory error in c3 during an ←MD executed by the Emulator, then a trap to location 0 in c1 occurs at the Emulator task level.

The hardware requires the execution of one additional Emulator click between the c3 which errored and the trap at location 0. Thus, other tasks and an additional Emulator click can intervene between the occurrence of the error and the trap code.

This trap only occurs for memory errors incurred by the Emulator task: device tasks must explicitly utilize the ErrorLogging register in the Memory Controller. Yes, the memory address is lost (as well as the syndrome if other memory reads occurred since the error).

### Stack Pointer Overflow or Underflow

If a pop or push is executed with the values of the stackPointer given in the following table. then a trap to location 0 in c1 at the Emulator task level occurs (the stackP is still modified).

The hardware requires the execution of one additional Emulator click before the trap at location 0. Thus, other tasks and an Emulator click can intervene between the occurrence of the error and the trap code.

Multiple pop's and push's can be specified per microinstruction in order to ameliorate the detection of Stack overflow or underflow. For instance, fXpop (i.e., the pop in the fX field), fZpop, and push executed together leave the stackPointer unmodified, yet simulate two pop's with respect to stack underflow detection. fXpop with push checks for stack overflow while not moving the stackPointer, and, likewise, push and fZpop check for underflow. The following table enumerates the cases.

| functions | stackP | Trap is | if stackP is |
|---|---|---|---|
| pop | -1 | underflow | 0 |
| push | +1 | overflow | 15 |
| fXpop, push | 0 | underflow | 0 |
| push, fZpop | 0 | overflow | 15 |
| fXpop, fZpop | -1 | underflow | 0 or 1 |
| fXpop, fZpop, push | 0 | underflow | 0 or 1 |

If the Emulator top-of-stack (TOS) element is kept in an R register and the rest of the Stack in the U registers, and it is assumed that TOS can always be stored away into the Stack, then these values imply a maximum stack size of 14 words.

### IB-Empty Error

If an ←ib, ←ibNA, ←ibLow, or ←ibHigh is executed when ibPtr = empty, then an IB-Empty Error trap occurs to location 0 in c1 at the Emulator task level. If the IB-Empty Error occurs in c1, a MDR← in the next cycle is canceled. (Furthermore, an IBDisp is ignored, but this fact is of no particular value.)

In normal operation (sec. 2.3.5) the IB is always guaranteed to have enough operand bytes (two) before a macroinstruction begins executing. However, when the macroprogram counter points to the last word of a page, the buffer is intentionally not refilled by the Emulator "refill" microcode and the IB-Empty trap can occur, indicating that control has actually proceeded across a page boundary. See the *DMR*.

If the IB-Empty error occurs in c1, then control transfers to location 0 in the next Emulator c1. However, if the error occurs in c2 or c3, the hardware requires the execution of one additional Emulator click before the trap at location 0. Consequently, other tasks and an Emulator click can intervene between the occurrence of the IB-Empty error in c2 or c3 and the trap code. In particular, if such a click executed a MDR← with an address which was a function of an IB value read in the previous c2 or c3, then a random memory location can be written.

The IB is not read during c2 or c3 of a macroinstruction's last click. However, the microcoder must ensure that, immediately following an ←ib, ←ibNA, ←ibLow, or ←ibHigh function executed in c2 or c3, there is not a memory write with a MAR← or Map← address which is a function of the IB value read in c2 or c3. (This is not checked for by MASS.)

### 2.5.6 Task Scheduling and Switching

A task is the microcode which supports an IO device or the Emulator. A device task runs whenever the device controller in the Dandelion asserts its "wakeup" request. Since a device task can only run during its pre-allocated clicks, a controller's maximum memory latency and maximum memory bandwidth is an outcome of its preassigned location within the round.

The Emulator and Kernel tasks behave differently than device tasks. The Kernel task is a special task used for communication between the CP and IOP (see 2.5.6.6). The Emulator task has no fixed assigned slot in the round: it executes during a click which a controller has opted not to use. Since devices do not utilize all of the bandwidth implied by the round structure, there is always a minimum number of clicks available to the Emulator.

#### 2.5.6.1 Task Allocation

The CP can control a maximum of 8 tasks. Currently, there are 6 wakeup lines (5 of them on the backplane) which can request microcode service. The eight task numbers are allocated between the devices, Emulator, and Kernel as follows:

| | |
|---|---|
| 0 | Emulator |
| 1 | Display or LSEP or MagTape |
| 2 | Ethernet |
| 3 | Refresh (Auxiliary) |
| 4 | Disk (Rigid) |
| 5 | IOP |
| 6 | IOP control store read/write address |
| 7 | Kernel |

The Dandelion is configured at boot time so that either the Display, or the LSEP, or the MagTape can use task number 1, but all three can not simultaneously use task 2. Normally, the Display task controls the refreshing of memory, but when the LSEP or MagTape (or other Option board controller) is active instead of the Display, then the Refresh task has this responsibility. Similarly, the Disk task cannot be simultaneously used by both the SA4000 and SA1000. Task 6 is currently not assigned to an actual device: instead it is used by the IOP as an address register when reading or writing the control store (see 2.5.6.7).

#### 2.5.6.2 Click Allocation

There are two types of rounds: a standard 5-click round and an extended 10-click round. The standard round is used with the HSIO board (Shugart SA4002 or SA1002 disks) and the extended round with the HSIO-LD board (LDC, or LargeDiskController: Trident drives). The extended 10-click round is an "even" 5-click round followed by an "odd" 5-click round. In the even rounds, the Ethernet task has claim to click 3, and in the odd rounds the Trident disk controller does.

Click 4 is special because the Display Controller hardware guarantees that memory references to the display bank can never abort in this click. In order to refresh memory and maintain the cursor, the Display and Refresh tasks are assigned to this click. When the Display is on, the Display task will start in click 4 of the 11th round of a Display line. In contrast, the Refresh task will begin with the 1st round of a Display scan line.

The LSEP also uses click 4 since its band buffers are located in the Display Bank. Moreover, because of hardware pin limitations, the LSEP and Display wakeup requests are *or'd* together (on the HSIO board). Thus, if both the Display and LSEP are enabled, their wakeup requests will be irresolvable. (Note the single microcode function, ClrDPRq, is used to reset *both* their wakeup requests.) Also in click 4, the Display-LSEP wakeup request has priority over the Refresh request. Conversely, due to special hardware in the MagTape controller, the Refresh request has priority over the MagTape request.

The following tables show the standard and extended rounds:

| | Click | Task |
|---|---|---|
| Standard Round: | 0 | Ethernet |
| | 1 | SAx000 Disk |
| | 2 | IOP |
| | 3 | Ethernet |
| | 4 | Display-LSEP-MagTape OR Refresh |

| | Click | Task |
|---|---|---|
| Extended Round: | 0-0 | Ethernet |
| | 0-1 | Trident Disk |
| | 0-2 | IOP |
| | 0-3 | Ethernet |
| | 0-4 | Display-LSEP-MagTape OR Refresh |
| | 1-0 | Ethernet |
| | 1-1 | Trident Disk |
| | 1-2 | IOP |
| | 1-3 | Trident Disk |
| | 1-4 | Display-LSEP-MagTape OR Refresh |

### 2.5.6.3 Click Bandwidth Utilization

The following table summarizes the bandwidth availble to each device and the percentage which remains for the Emulator when the controller is transferring data. (Pre- and post-data-transfer overhead, which normally utilizes 100% of device clicks, is not included.) Note that the IOP only transfers one byte per click, so its maximum available rate is actually 3.9 Mbits/s.

| Device | BW allocated (Mbits/s) | BW used (Mbits/s) | % remaining for Emulator |
|---|---|---|---|
| Ethernet(w/SAx000) | 15.6 | 10.0 | 36 |
| Ethernet(w/Trident) | 11.7 | 10.0 | 15 |
| SA4000 | 7.8 | 7.14 | 9 |
| SA1000 | 7.8 | 4.27 | 45 |
| Trident | 11.7 | 9.6 | 18 |
| Display (microcode) | 7.8 | 1.1 | 86 |
| IOP | 7.8 | 2.0 | 26 |
| LSEP & Refresh | 7.8 | 3.7 + 1.1 | 38 |
| MagTape & Refresh | 7.8 | .6 + 1.1 | 78 |

Even with the Ethernet, SA1000, and IOP concurrently transferring data and the Display microcode refreshing memory, the Emulator still executes 60% of the time.

### 2.5.6.4 Tasking Hardware

The CP control hardware was designed to hide the details of task switching from the programmer. Since tasks are frequently resumed and suspended by controller wakeup requests, the hardware performs all the necessary start upand stop functions: every click it saves the current task's microprogram counters and pending condition bits and, when it is scheduled to run again, it restores them. Figure 14 illustrates the process, outlined below.

Every c2 the Schedule Prom in the CP, on the basis of the controller wakeups and click number, decides which task (Nt) will run in the next click. Also in c2, the Switch Prom, on the basis of Nt, the currently executing task (Ct), and Wait (x.xx), decides whether to activate the task switching logic (and, if so, sets Swc2 ← 1). A task switch has two parts dealing with the outgoing and incoming microprogram counter and conditions: (1) a restore process and (2) a save process.

(1) The Temporary Program Counter (TPC) array holds the eight 12-bit task microprogram counters. If it is cycle 2 and a task switch is occuring, the TPC, as addressed by the next task number, is the source of the control store address. The next task's first micronstruction is subsequently read in c3 and executed in the following c1. In short, NIA ← TPC[Nt] at the end of c2.

At the same time the next task's microprogram counter is being read from TPC[Nt], the saved condition bits are read out of the Temporary Conditions array, TC, and latched into the TC regsiter. During c3, TC is *or'd* with the next task's first microinstruction INIA field, which is being read from the microstore. In summary, the saved condition bits are read during c2 from TC[Nt], latched into the TC register, and in c3 *or'd* with INIA.

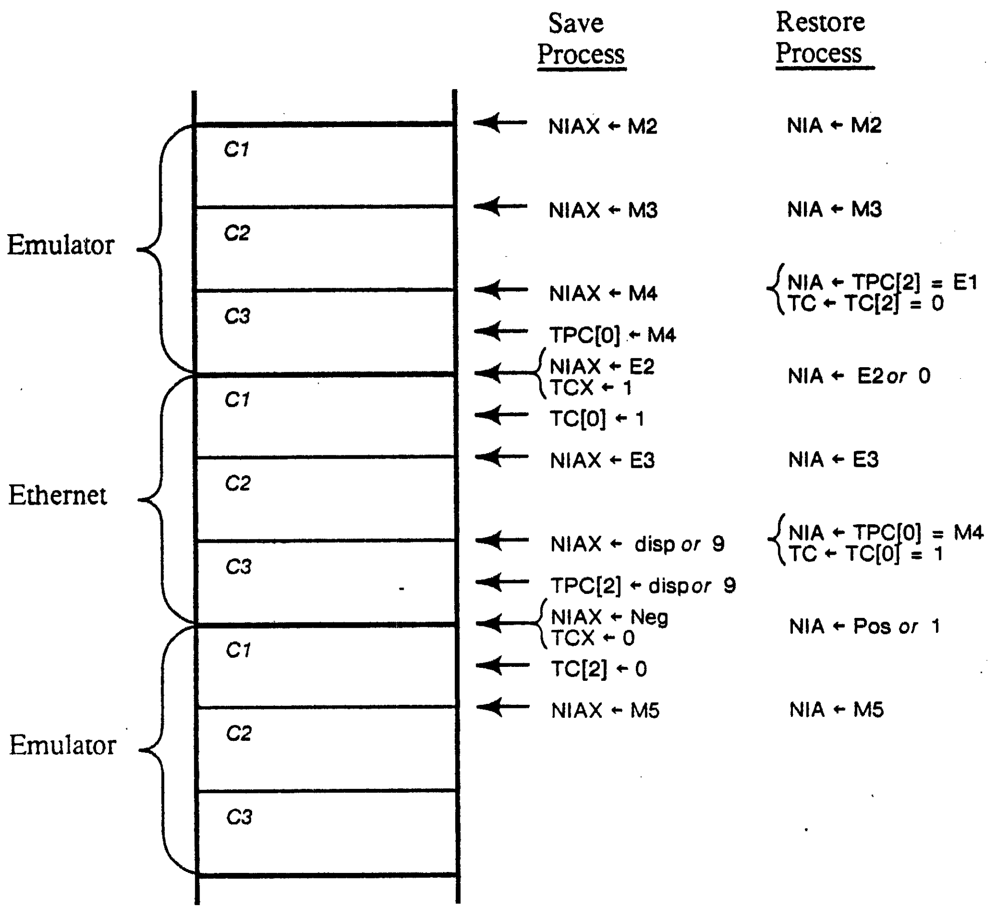
(2) The current task's Next Instruction Address (which would have been loaded into NIA if there were no task switch) is latched into the NIAX register at the end of c2 and then saved in the current task's TPC location during c3. In general, every c3, TPC[Nt] ← NIAX. (Note that in c3, Nt equals the task currently executing.)

Furthermore, the condition bits of the task currently executing (which would have been *or'd* into INIA) are latched into the TCX register at the end of c3 and then saved into the TC array in c1. In general, every c1, TC[Nt] ← TCX. (In c1, Nt actually equals the task which executed in the previous click. The condition bits are saved in c1 because there is not enough time in c3 to write them into a RAM.)

The following table summarizes when the TPC and TC are read and written and the interpretation of Nt:

| cycle | operation | Nt |
|---|---|---|
| end of c2 | NIA ← TPC[Nt] | next task |
| c3 | TPC[Nt] ← NIAX | current task |
| end of c3 | NIA ← INIA *or* TC | |
| end of c3 | TCX ← DispBr *or* Link | |
| c1 | TC[Nt] ← TCX | previous task |

The TPC and TC RAMs are written every click (except suspended clicks) even if there is not a pending task switch. Consequently, if the Emulator is suspended because of Display bank interference, it's correct restart address is available in the TPC.



```
{Emulator microcode for above example.}
M1:     Noop,                   c1;
M2:     Noop,                   c2;
M3:     [] ← -1, NegBr,         c3;

M4:     BRANCH[Pos, Neg],       c1;
Pos:    GOTO[ME]                c2;
Neg:    Noop                    c2;
M5:     Noop                    c3;
```

```
{Ethernet microcode for above example}

E1:     Noop                    c1;
E2:     XBus ← 9, XDisp,        c2;
E3:     DISP4[disp],            c3;
```

Figure 14. Demonstration of Tasking Mechanism:

Where the Emulator task (0) is preempted by the Ethernet task (2) for one click.

This example demonstrates a pending branch across the task switch for the Emulator and shows when the TPC and TC arrays are written and when NIAX is not equal to NIA.

The Save Process refers to the writing of the TPC & TC arrays, while the Restore Process refers to the reading out of TPC & TC.

### 2.5.6.5 Display Bank Interference

If any task references the dual-ported Display bank (lowest 64K of real memory) and the Display controller is reading the bank, then the task is suspended for the duration of that click; that is, no microinstructions are executed during the suspended click. Click suspension is always in multiples of clicks and the c1-c2-c3 structure is not modified. Device tasks should not reference the Display bank (unless the Display is off).

In particular, the Emulator task is suspended until either it is scheduled for click 4 or the Display controller relinquishes the low bank. This reduces the Emulator's maximum possible bandwidth into the low bank by about half (47%) when the Display is active: from 38.9 to 18.3 Mbits/s (1.1 megaword/s).[7]

Clicks are suspended by the signal Wait which gates off all clocks which can change sensitive state information. In the schematics, such clocks are labeled WaitClock, in contrast with the normal AlwaysClock. Wait is defined

```
Wait ← (MAR← and YH[4-7]=0 and Disp-Proc'=0) or (IOPWait and c1)
        or (Wait and c2) or (Wait and c3).
```

When Wait is true, the following registers and RAMs are not written: R, Q, U, RH, stackP, IB[0], IB[1], ibFront, ibPtr, Link, TC, TPC, MInt, pc16', and Errors (Memory, stackPointer, CSParity, IBEmpty). By contrast, the following are unaffected by Wait: MIR, NIA, NIAX, TCX, TC, KernelReq, EKErr, and schedular task states (Nt, Ct, Pt, Swc3).

Since the Microinstruction (MIR) and Next Instruction Address registers' (NIA) clocks are unaffected during suspended cycles, the decoded signals from the MIR can change during an aborted click. However, this does not result in a random sequence of decoded microinstructions: the MIR output in c1, c2, and c3 is equal to the values it would have had if the click were not suspended. This is because the microinstruction loaded into MIR is always defined by an NIA which is unaffected by any invalid states generated during the suspended click: cycle 1's MIR output is defined by the NIA read from the TPC (in c2), cycle 2's by the value of INIA *or* TC (computed in c3), and cycle 3's by INIA *or'd* with conditions bits specified in c1 (which are not effected by WaitClock in c1). Furthermore, if the Emulator is suspended for consecutive clicks, the MIR output is the same for each click since NIA is reloaded from the TPC during suspended clicks.

### 2.5.6.6 Kernel Task

The Kernel task is used for supporting the debugging of the CP (e.g., breakpoints, reading/writing CP registers) and handling the CP-IOP communication while booting (e.g., memory refresh during control store read/write). When the Kernel task is enabled, it executes in all clicks, preempting all device tasks and the Emulator.

The Kernel task runs if there is a CSParityError, IOPWait is true (2.5.6.7), or the microcode function EnterKernel is executed. If EnterKernel is executed in c1, the Kernel runs in the next click. However, if executed in c2 or c3, an Emulator or device click can intervene before the Kernel runs. When the Kernel task is started, the Switch Prom does not cause a task switch; hence, a breakpoint microinstruction can specify an entry point into the Kernel.

The Kernel task request remains active until reset by the ExitKernel function. An ExitKernel is overridden by a pending IOPWait or CSParityError. When ExitKernel is executed in c1, the next click can be executed by another task (depending on which click the ExitKernel is in and the wakeup requests).

### 2.5.6.7 CP-IOP Interface

The IOP interfaces with the CP as both a standard I/O controller and as a boot loader/debugger. This section deals with the booting interface: the control lines used to load the control store and initialize the tasks' microprogram counters (TPCs). The following signals are used between the IOP and CP:

| | |
|---|---|
| SwTAddr | high level causes Nt = IOPTPCHigh[0-2] and<br>NIAX[0-4] = IOPTPCHigh[3-7] and<br>NIAX[5-11] = IOPData bus |
| IOPWait | high level sets Kernel wakeup request and<br>WaitClock is suspended |
| WrTPCHigh | positive edge writes IOPTPCHigh with IOPData bus |
| WrTPCLow | pulse causes TPC[Nt] ← NIAX |
| CSWE[n]' | pulse writes a control store byte with IOPData bus |
| ReadCSEn' | places CS byte, TPC, & TC onto IOPData bus |
| ReadCS[n] | selects CS, TPC, & TC bits to use with ReadCSEn' |

The basic algorithm for reading or writing control store is to first write TPC[6] with the address of the location to be accessed and then read or write data bytes (addressed by CSWE[n]' or ReadCS[n]) while allowing the Kernel to Refresh memory if necessary. Although all of the tasks' TPCs can be initialized, the TC registers cannot be loaded by the IOP.

In general, when reading or writing a TPC location or CS byte, both SwTAddr and IOPWait must be high and the value of Nt (loaded into IOPTPCHigh) must be 6 or 7. When SwTAddr is true, Nt and NIAX are defined by the IOPTPCHigh register instead of their normal sources. This allows the IOP to address and supply data directly to the TPC RAM.

Setting IOPWait causes the Wait line to be high. Thus, registers clocked by WaitClock cannot be loaded with spurious data while a TPC or CS location is being written. (Moreover, the CSParityError trap cannot occur.) IOPWait also sets the Kernel wakeup request so that the Kernel task runs when IOPWait is removed.

While IOPWait = 1 and Nt = 6 or 7, the Switch Prom causes a continuous task switch; that is, Swc2 is always true and NIA is set to the value of TPC[6] or TPC[7]. In this state, the Kernel microcode does not run and its TPC does not change. However, after writing one byte of control store or one TPC location, it may be necessary to refresh main memory. In this case, IOPWait and SwTaddr are reset and, since the IOPWait caused the Kernel wakeup request to be set, the Kernel begins running at the saved TPC location and executes the required number of Refresh functions or performs a function enumerated by the IOP via the normal I/O interface (e.g., ←IOPIData, ←IOPStatus).

The following table shows which control store bytes are read or written with ReadCSEn' and CSWE[n]'. Note that when writing the control store the inverse of the data must be supplied on IOPData.

| ReadCS | CSWE[n] | IOPData[0-7] |
|---|---|---|
| 0 | a | rA, rB |
| 1 | b | aS, aF, aD |
| 2 | c | ep, Cin, EnableSU, mem, fS |
| 3 | d | fY, INIA[0-3] |
| 4 | e | fX, INIA[4-7] |
| 5 | f | fZ, INIA[8-11] |
| 6 | | TC, TPC[0-3] |
| 7 | | TPC[4-11] |

## 2.6 Input/Output Interface

The CP and the high speed devices were mutually designed within one framework and are inexorably bound together: the I/O bus is the same as the CP's main data bus (the X bus), the I/O register control is directly encoded into the microinstruction format, and the devices depend on the preallocated click structure for guaranteed memory latency and bandwidth. This intimate relationship between the devices and the processor exists in order to absolutely minimize the overall system cost. By sharing the ALU among several controllers, overlapping memory accesses with ALU computation, and guaranteeing memory latency, very small IO controllers can be built. This section exists because it is possible to design different disk or display controllers on the HSIO board, new high speed controllers on the Option board, and new Memory systems.

### 2.6.1 CP-IO Interface

The following signals and buses are used between the CP and a typical device controller, called Dev:

| | |
|---|---|
| X bus | 16-bit data to or from memory or 2901 |
| Y bus | 16-bit data from 2901 |
| DevReq' | task wakeup request to CP Schedule Prom |
| DevCtl←' | signal from CP to load controller control register from X or Y Bus |
| DevOData←' | signal from CP to load controller data register from X Bus |
| ←DevStatus' | signal from CP to place controller status onto X Bus |
| ←DevIData' | signal from CP to place controller data onto X Bus |
| ClrDevRq' | signal from CP to reset controller wakeup request |
| DevStrobe' | signal from CP for general use by controller |
| IODisp | CP branch on a controller state |
| Wait | level from CP to gate off WaitClock |

Normal CP-Controller interaction (for input) goes something like: (1) A controller receives a word of data, (2) the controller activates its wakeup request, (3) the controller's task runs in its allocated click, (4) the microcode reads the data from the controller to main memory or 2901, and (5) the controller resets its wakeup request. In general, the wakeup request is either explicitly turned off by the task via ClrDevRq' or is turned off by the controller when it senses a ←DevIData', DevOData←', or DevStrobe'. It is explicitly assumed that a controller only causes wakeups when data transfers are pending (or when directed by its task) in order to minimize the impact on the Emulator.

A device's wakeup request must be turned off by the end of the cycle 1 which follows the service click in order to prevent a task from accidentally running again. Since the device's wakeup request must be 2-level synchronized, this implies that the reset-wakeup function must be executed in c1 or c2 for those devices which have a two-click minimum separation.

In general, all controller control registers should be clock'd with WaitClock so that spurious device actions are prevented while writing control store. If a control signal can be used by an Emulator click which could be suspended, it should also be gate'd with WaitClock. Device tasks should not reference the Display bank unless the Display is off.

### 2.6.2 Controller Latencies

A controller's data buffer size depends on how often the buffer is serviced and what kind of wakeup scheme is employed. There are two basic wakeup strategies: post and prerequesting. In the former case, the wakeup request is raised after the device buffer is available to be read/written by the CP. In prerequesting, the wakeup request is raised before the device buffer is actually available. Only the SAx000 disk uses prerequesting. Where a task must process some of the data and cannot transfer a word per click, then a FIFO is usually used as a buffer (as in the Ethernet). However, when little or none of the data must be examined by the microcode, then a simple register buffer is sufficient (as in the rigid disk controllers and LSEP).

In order to avoid overruns with the postrequesting scheme, the maximum microcode service latency plus the wakeup-request synchronizer delay must be less then the data rate:

$$L_{max} + s_{max} < b/r,$$

where b is the number of bits of buffering, r is the data rate of the device (in Mbits/s), $L_{max}$ is the maximum latency (in μseconds), and $s_{max}$ is the synchronizer delay (equal to 2T, where T = .137 μsec). If the task microcode transfers one word per click, then

$$L_{max} = 3dT + 4T \text{ for output, and}$$
$$L_{max} = 3dT + 3T \text{ for input,}$$

where d is the maximum seperation between device clicks. If the microcode does not always transfer a word per click, then $L_{max}$ is correspondingly greater.

For prerequesting, the wakeup request cannot be made too early, thus the constraint

$$s_{min} + L_{min} - t_{handoff} > 0,$$

where $t_{handoff}$ is the time for the CP to read the buffer (equal to T) or the controller to read the buffer (about .05 μsec)). If prerequesting begins p device bit times before the buffer is ready, then

$$s_{min} = 2T - p/r, \text{ and}$$
$$s_{max} = T - p/r.$$

Since $L_{min}$ = 5T for output and 4T for input, p must satisfy the following conditions in order for prerequesting to work ($t_{handoff}$ = 0 for output):

$$[rT(3d + 6) - b] < p < 6rT \quad \text{for output, and}$$

$$[rT(3d + 5) - b] < p < 4rT \quad \text{for input.}$$

For SA4000 write or verifty operations: $4.54 < p < 5.51$ !

### 2.6.3 IO Controller Design Rules

Since replacement or augmented controllers are being designed for the Dandelion, the following design rules should be followed in order to guarantee correct operation. Figure 15 illustrates the proper application of the CP interface signals.

**(1)** CP control signals such as DevReq', DevCtl←', ←DevIData', ClrDevRq', and DevStrobe' originate from an SN74S138 decoder and therefore must not be used in an asynchronous way, such as the clock input of a register. These CP signals must be synchronized to the CP clock or gate'd with pAlwaysClk or pWaitClk.

**(2)** Controller input buffers must be either an SN74S240 or SN74S374 (or equiv) and the CP control signal which enables them onto the X bus, such as ←DevIData' or ←DevStatus', must be connected directly to the output enable input without being gate'd in any way.

**(3)** If there is more than one output register on the board, the X bus must be buffered with an SN74S241 (or equiv) before routed to the registers. The CP control signals which load the output registers, such as DevOData←' or DevCtl←', can be modified per the constraints of a clock qualifier signal (see (5)).

**(4)** The device wakeup request signal, DevReq', must come from an SN74S374 (or S74, or equiv) and must be synchronized by at least 2 such FF's.

**(5)** The clock qualifying structure shown in figure 8 must be used: the S02 is located in the position nearest backplane pins 1-10 and the "qualifier" gates are no further away then the center of the board, their preferred location. Clock qualifier terms should be valid by 94 nanoseconds after the positive (active) edge of AlwaysClk. Clock'd registers should be no more than 10" from their qualifier gate.

pWaitClk must be used for any register which, if spruiously loaded during a control store boot, can activate a device function (e.g., disk write enable). Such registers should also be reset by IOPReset' which is *or'd* with the power supply on/off reset.

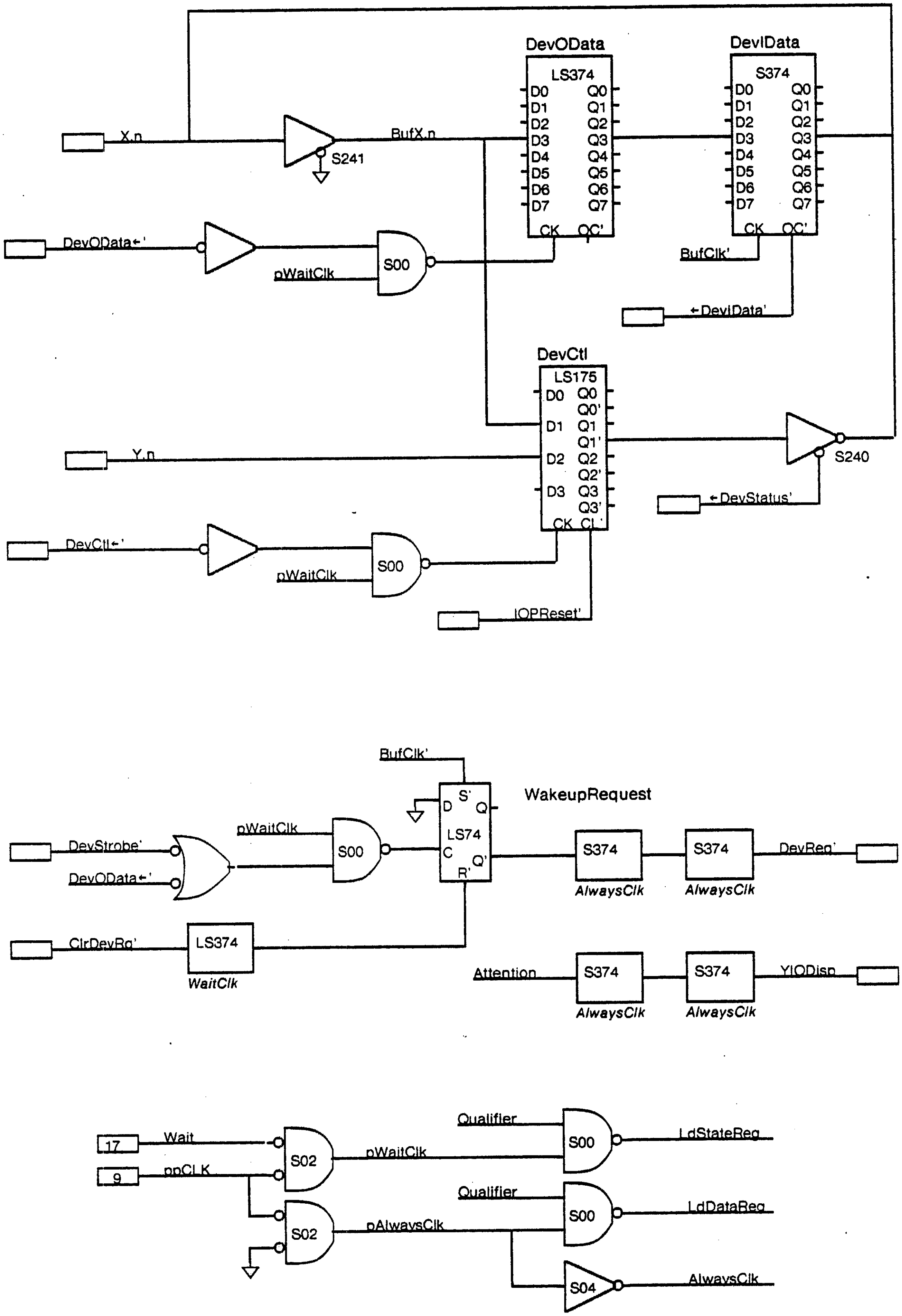


Figure 15. Controller Hardware Demonstrating I/O Rules & CP Interface

## 2.7 Example Microcode

Just as a melody, in order to be heard, requires both notes and intervals, the CP hardware should be viewed in light of its corresponding microcode. The following microcode examples illustrate how and in what time frame certain elementary functions are accomplished. There are seven examples, some simplified: Mesa Emulator Load Local n, Read n, Jump n, Refill, and the Ethernet, Disk, and LSEP inner loops. See the *DMR* for a description of the microcode format.

(1) The Mesa Emulator Load Local 1 (LL1) macroinstruction indexes the local frame pointer and then push's the addressed word from memory onto the Stack. It executes in one click if the indexing operation does not cross a page boundary and in three if a page cross occurs. If the Map flags must be updated (RMapFix), another two clicks are required.

```
@LL1:       MAR ← Q ← [rhL, L+1], L1←L1.PopDec, push,                        c1,  opcode[1'b];
LLn:        STK ← TOS, PC ← PC+PC16, IBDisp, L2←L2.LL, BRANCH[LLa,LLb,1],    c2;
LLa:        TOS ← MD, push, fZpop, DISPNI[OpTable],                          c3;
LLb:        Rx ← UvL,                                                        c3;

LSMap:      Noop,                                                            c1;
            Q ← Q - Rx, L2Disp,                                              c2;
            Q ← Q and 0FF, RET[LSRtn],                                       c3;

LLMap:      Map ← Q ← [rhMDS, Rx+Q],                                         c1,  at[3,10,LSRtn];
            Noop,                                                            c2;
            Rx ← rhRx ← MD, XRefBr,                                          c3;

            MAR ← [rhRx, Q + 0], L0←L0.R, BRANCH[RMUD,$],                    c1;
            IBDisp, GOTO[LLa],                                               c2;
RMUD:       CALL[RMapFix],                                                   c2;
```

(2) The Mesa Emulator Read 1 (R1) macroinstruction indexes the virtual address on the top of Stack and then push's the addressed word from memory onto the Stack. It executes in two clicks. Four are required if the page has been read the first time; that is, the Map flags must be updated.

```
@R1:        Map ← Q ← [rhMDS, TOS + 1], L1←L1.Dec, pop,                      c1,  opcode[101'b];
            push, PC ← PC + PC16,                                            c2;
            Rx ← rhRx ← MD, XRefBr,                                          c3;

            MAR ← [rhRx, Q + 0], L0←L0.R, BRANCH[RMUD,$],                    c1;
            IBDisp, GOTO[LLa],                                               c2;
```

(3) The Mesa Emulator Jump 2 (J2) macroinstruction increments the PC by 2 bytecodes and then refills the instruction buffer. It executes in two clicks. Five are required if the jump crosses a page boundary.

```
@J2:         MAR ← PC ← [rhPC, PC+1], push,                                     c1,opcode[201'b];
             STK ← TOS, L2 ← L2.Pop0IncrX, Xbus←0, XC2npcDisp, DISP2[jnPNoCross], c2;

jnPNoCross:  IB ← MD, pop, DISP4[JPtr1Pop0, 2],                                 c3,  at[0,4,jnPNoCross];
jnP1Cross:   Q ← 0FF + 1, L0 ← L0.JRemap, CANCELBR[UpdatePC, 0F],               c3,  at[2,4,jnPNoCross];

JPtr1Pop0:   MAR ← [rhPC, PC + 1], IBPtr←1, push, GOTO[Jgo],                    c1,  at[2,10,JPtr1Pop0];
JPtr0Pop0:   MAR ← [rhPC, PC + 1], IBPtr←0, push, GOTO[Jgo],                    c1,  at[3,10,JPtr1Pop0];
Jgo:         TOS ← STK, AlwaysIBDisp, L0 ← L0.NERefill.Set, DISP2[NoRCross],    c2;
```

(4) The Mesa Emulator instruction buffer refill code executes in one click if the buffer was not empty and in two if it was. Four to six clicks are required if the refill occurs across a page boundary.

```
{Buffer Empty Refill.  Control goes from NoRCross to RefillNE since RefillE+1 does not contain an IBDisp.}
RefillE:      MAR ← [rhPC, PC], PC ← PC-1, L0 ← L0.ERefill,          c1,  at[400];
              PC ← PC+1, DISP2[NoRCross],                              c2;

{Buffer Not Empty Refill.}
OpTable:         {"Noop" location of Instruction Dispatch table}
RefillNE:     MAR ← [rhPC, PC + 1],                                    c1,  at[500];
              AlwaysIBDisp, L0 ← L0.NERefill.Set, DISP2[NoRCross],     c2;

NoRCross:     IB ← MD, uPCCross ← 0, DISPNI[OpTable],                  c3,  at[0,4,NoRCross];
RCross:       Q ← 0FF + 1, GOTO[UpdatePC],                             c3,  at[2,4,NoRCross];
```

(5) The Ethernet input inner loop transfers one word per click until either a page boundary is crossed (ERead+2 or ERead+3), the maximum sized packet has been exceeded (ElTooLong), or the controller has signaled an abnormal condition (ERead+1 or ERead+3).

```
{main input loop}
ElnLoop:      MAR ← E ← [rhE, E + 1], EtherDisp, BRANCH[$,ElTooLong],          c1;
              MDR ← ElData, DISP4[ERead, 0C],                                  c2;

ERead:        EE ← EE - 1, ZeroBr, GOTO[ElnLoop],                              c3,  at[0C,10,ERead];
              E ← uESize, GOTO[EReadEnd],                                      c3,  at[0D,10,ERead];
              E ← ElData, uETemp2 ← EE, GOTO[ERCross],                         c3,  at[0E,10,ERead];
              E ← ElData, uETemp2 ← EE, L6←L6.ERCrossEnd, GOTO[ERCross],       c3,  at[0F,10,ERead];
```

(6) · The SAx000 disk write and verify inner loop transfers one word per click until the required number of words have been sent.

```
WrtVerLp:     MAR ← [RHRCnt, RCnt], RCnt ← RCnt+1,                     c1,  at[0,2,FinWrtVer];
              RAdr ← RAdr-1, ZeroBr, CANCELBR[$, 2],                   c2;
              KOData ← MD, BRANCH[WrtVerLp, FinWrtVer],                c3;
```

(7) The LSEP output inner loop outputs a band buffer entry from the display bank and then clears the entry. This continues until the required number of words have been transferred, which is detected by aligning the data on a page boundary.

```
scan:         MAR← [displayBase1, rX+0], ClrDPRq,                      c1;
              MDR← rY{= zero}, rX← rX+1, PgCarryBr,                    c2;
              POData← MD, BRANCH[scan, endLine],                       c3;
```

## 2.8 Footnotes

[1] All of the microcode-related specifications and rules presented in this chapter are validated by the microcode assembler and control-store-allocation program (MASS).

[2] The writeable control store is expensive: out of the 160 chips total, 70 are microstore chips.

A special version of the CP has been built which has a 16K control store partitioned into four, 4K banks. The 2-bit Bank register can be loaded from NIAX with fZ = Bank←. All non-Emulator tasks are forced to execute from bank 3. Error trap location 0 exists in each bank.

[3] Where did this (prime) number come from? All system timing is based on the Display's bit time, 19.59 nS (51.04 MHz, ± .05%). There are 7 bit times in a cycle and 210 cycles (14 rounds) in one horizontal display line. More precisely, the cycle time is 137.14 ± .57 nsec.

Alternatively, the cycle time (137) equals the inverse of the fine structure constant: a fundemental dimensionless constant equal to 2p times the square of the electron charge in electrostatic units, divided by the product of the speed of light and Planck's constant $(2pe^2/c\hbar)$ !

[4] This sequence has been likened to the triple time meter of a waltz!

[5] Because there are so many sources-and sinks on the X bus, it has a nonnegligible capacitance: it has been measured at 337 pF!

[6] The *or*ing of a Link register with the low 4 bits of the IB byte during an IBDisp is not encouraged as this feature will not exist in a future version of the processor.

[7] The 18.3 Mbits/s into the display bank is approximated as follows: There are 70 clicks per display scan line and, of these, the Display controller uses 4*10 = 40 clicks for a normal scan line. Furthermore, the display microcode uses 2 clicks for memory refresh. During 808 of the total 897 scan lines, the display controller is actually pumping bits out to the monitor. Thus, the Display controller and microcode use about (808/897)(42/70)(38.4 Mbits/s) = 20.6 Mbits/s of the bandwidth, lcaving 38.9-20.6 = 18.3 Mbits/s for the Emulator.

## 3.0 Memory System

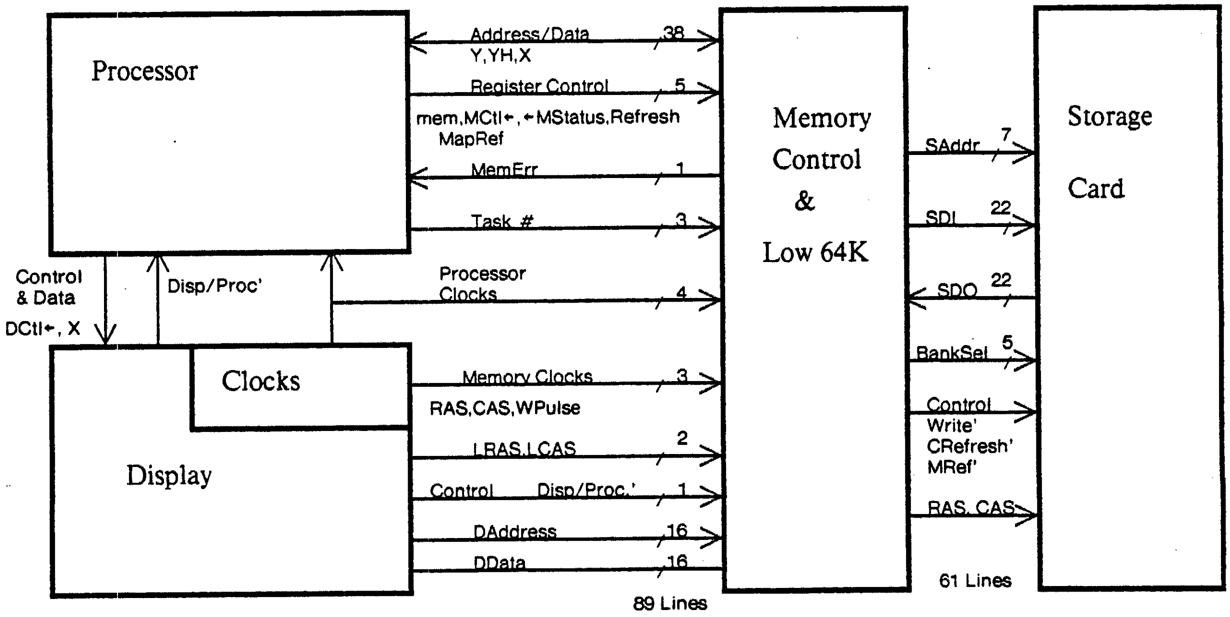
The memory system has two, 16-bit ports: one to the central processor (CP) and one to the display controller The CP shares the lowest 64K bank with the display and has exclusive use of the upper banks. Single-bit error correction and double-bit error detection is performed on all words delivered to the CP, but words used by the display are not corrected. The memory cycle time for the CP is 411 nanoseconds (nS), but for the display controller is either 293 (full) or 215 (page) nS.

The memory can be configured in at least five different sizes depending on the mix of Memory Control Cards (MCCs) and Memory Storage Cards (MSCs). The lowest 64K words (Display bank) are located on the memory control card along with the error correction and port logic. The storage card holds additional memory chips plus data and address drivers. The timing signals for the memory system are generated by display controller (sec. x.xx) and are synchronous to the processor clocks. Figure 17 is a block diagram of the memory controller.

The MCC comes in one of two sizes: 64K or 256K words. Likewise, the MSC has either 128K or 512K words (the large version is called MSC-X). With some modifications, the 256K MCC card (called MCC-X) can be used with the 128K storage card. The maximum real memory size is 1,048,576 words. The following configurations are standard:

| MCC | MSC | Total size (words) |
|---|---|---|
| 64K | none | 65,536 (64K) |
| 64K | 128K | 196,608 (192K) |
| 256K | none | 262,144 (256K) |
| 256K | 128K | 393,216 (384K) |
| 256K | 512K | 786,432 (768K) |

From the micropogrammer's perspective, the CP controls all accesses to the memory: the CP's X, Y, and YH buses are used to supply addresses and transfer data. Device controllers can only use memory via their corresponding microcode tasks. (See section 2.4, "Main Memory Interface.") The Display controller is the excemption: it actually constructs its own memory timing signals (RAS and CAS) in order to acheive the maximum bandwidth possible through its port (sec. x.xx). The Display controller does not use the X and Y buses, but has its own 16-bit address and data buses. The following figure is a block diagram of the memory system:



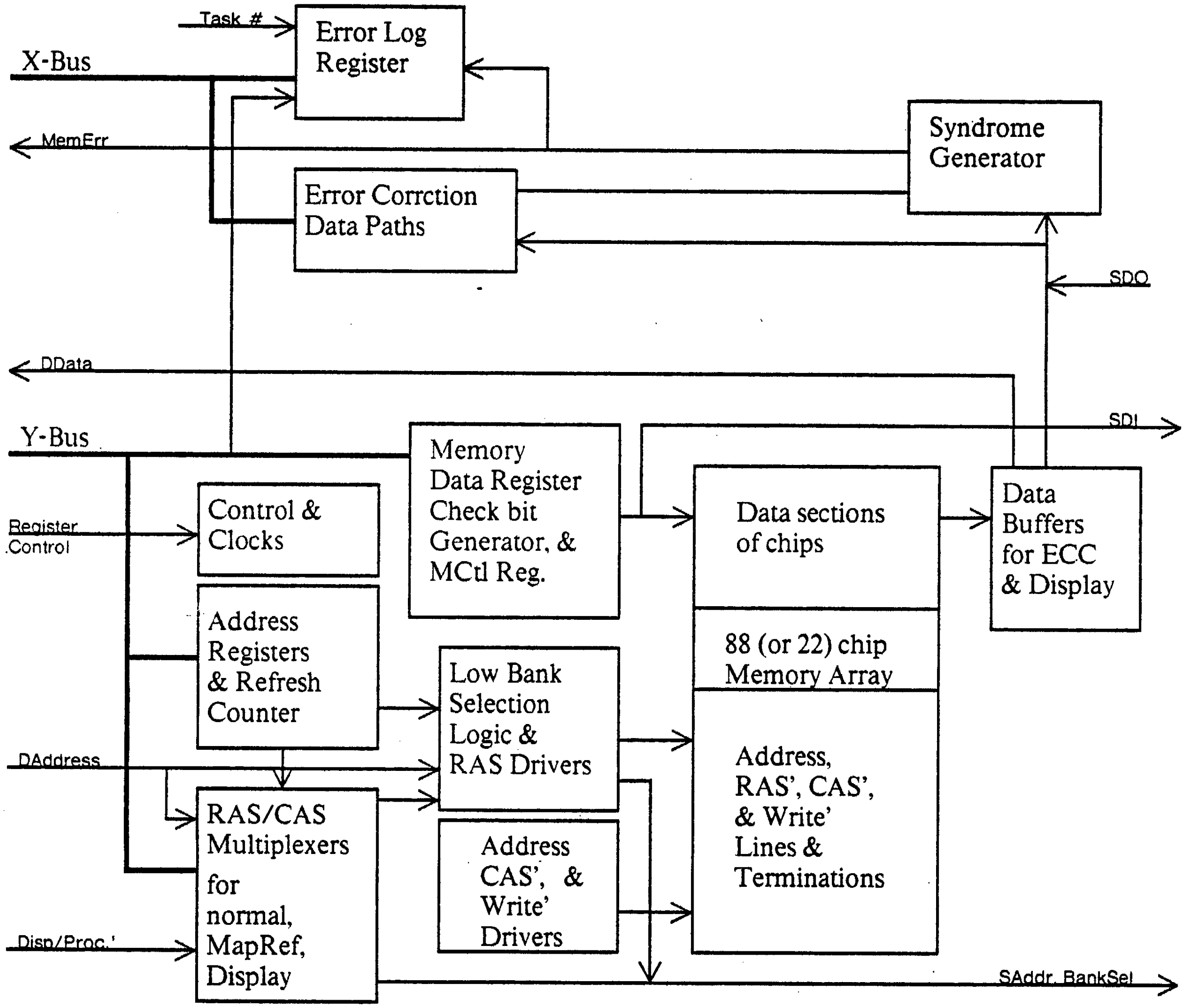

Task #
Error Log Register
X-Bus
MemErr
Syndrome Generator
Error Corrction Data Paths
SDO
DData
SDI
Y-Bus
Memory Data Register Check bit Generator, & MCtl Reg.
Register .Control
Control & Clocks
Data sections of chips
Data Buffers for ECC & Display
Address Registers & Refresh Counter
88 (or 22) chip Memory Array
Low Bank Selection Logic & RAS Drivers
DAddress
Address, RAS', CAS', & Write' Lines & Terminations
RAS/CAS Multiplexers for normal, MapRef, Display
Address CAS', & Write' Drivers
Disp/Proc.'
SAddr, BankSel

Figure 17. Memory Control and Low 64K Bank

## 3.1 CP Interface Summary

This section provides a summary of each of the functions of the memory system as viewed from the central processor. Figure 18 summarizes the functions. For a complete description of the microcode interface, see section 2.4.

### Read

A read operation is started by placing the memory address on the Y and YH busses and asserting mem in the first cycle of a click. The data can be read back to the X bus during the third cycle by asserting mem then. All data read by the processor is error corrected unless the correction inhibit bit is set in the Memory Control (MCtl) register.

### Write

The first cycle of a write operation is dedicated to sending the address to memory. It is identical to the first cycle of a read operation. The data to be stored must be delivered to the memory during the second cycle of a click, by asserting mem in the second cycle, and placing the data on the Y bus. Error correction check bits are always calculated and stored automatically by the memory system. If a write operation in the second cycle is followed by a read in the third, the data existing before the write is returned.

### Map Reference

The Dandelion's virtual memory map is kept in main memory. A map-reference-type memory read is identical to a standard read, except the bits supplied by the Y and YH busses are shifted to facilitate indexing into the Map. Microcode uses this feature to provide a 22-bit virtual memory system with the MCC and a 24-bit system with the MCC-X.

The virtual memory is divided into 256 word pages. The $\overline{\text{Map}}$← function discards the low 8 virtual address bits (since they reference the word location on the page), moving the high 14 bits (virtual page number) to the low 14 or 16 bits used for the real map address. The location of the 16K map is fixed between locations 10000 and 13FFF (hex) in real memory.

Each 16 bit entry in the Map contains 10 to 12 bits of real page number and four flags describing the page (present, dirty, referenced, etc). To derive a real address from a virtual one, the microcoder uses the map function (Map←), checks the flags and appends the original low order 8 bits to the 10 or 12 bits fetched (sec. 1.4.2). The presence of a Map← function in cycles 2 or 3 has no effect on the memory. mem should not be asserted, unless its side effects are desired (sec. 1.4.2).

### Refresh

The memory controller contains circuitry to facilitate memory refresh. Each memory chip is organized as a 128x128 (or 256x128x2) bit matrix. When the row address is received, all bits in the specified row are read. The column address is used to select one of them. At the end of the memory cycle, all 128 bits are rewritten to perform a refresh. Hence, a row of a chip may be refreshed by reading any bit in that row. If the column address is suppressed during refresh, a substantial section of the chip remains quiescent, saving power. During each refresh cycle, the memory controller broadcasts only a 7 (or 8) bit row address and row address strobe (RAS) to every memory chip. This row address is supplied by a counter on the MCC that is incremented at the end of the cycle.

Refresh is initiated by asserting the Refresh function from the CP during cycle 1 of a click when the display is quiescent. The Refresh line is ignored during cycles 2 and 3 and whenever the display accesses memory. All memory chips require that 128 rows be refreshed at least every 2 milliseconds. A horizontal line on the display takes 28.8 microseconds, hence, the memory should be refreshed at least 1.85 times per horizontal line. The standard display code performs two refresh cycles each line. The display microcode was chosen to do this because it can guarantee that the display hardware is inactive. Note that any displayless configuration of the Dandelion must contain some combination of hardware and microcode to perform the refresh task. The Refresh task is used in this case.

### Display Lockout

The low 64K of the memory is shared between the display and the CP. The display has priority. When actually scanning a line, the display consumes clicks 0 through 3, leaving click 4 for the CP. Thus, one click out of 5 is available for use by display handling microcode and accessesby the Emulator to the low bank. As discussed in section 2.5.6.5, "Display Bank Interference," the lockout (plus refresh & display microcode functions) reduces by about half the Emulator's maximum possible bankdwidth into the display bank: from 38.9 to 18.3 MBits/s.

Lockout occurs only if the processor and display attempt to access the low bank at the same time. Accesses to the high banks are not affected. Lockout does not occur during retrace intervals (horizontal and vertical), or during any other period of display inactivity (such as when the display is disabled). By convention, time critical hardware tasks using the first 4 clicks must never attempt access to the low (display) memory bank since a lockout could occur causing extra delay. In particular, one could not fill the bit map directly from an I/O device such as the disk or Ethernet without first disabling the display. See the display controller description for exact details of display timing.

Lockout is implemented by generation of a wait signal in the CP whenever a bank 0 (low 64K bank) access is attempted and the display is already using the low bank. The processor suspends the microcode which started in that click, and continues the normal arbitration of what runs in the next click. In this manner, lockout in one click does not hold up operation in the following click.

**MAR←** Memory Address Register

| YH Bus |  |  |  |  | Y Bus |
|---|---|---|---|---|---|
| ignored<br>0 | 4 | 5 | 6 | 7 | 0 … 15 |
|  | 20-bit real address |  |  |  |  |

MAR← mem during c1

Action: Contents of YH[4-7],,Y[0-15] is used as memory address.

**MDR←** Memory Data Register

| Y Bus |
|---|
| 0 … 15 |

MDR← mem during c2

Action: Contents of Y Bus go into memory location specified by contents of MAR as loaded during first cycle of click. No write occurs if the low 64K bank is selected and it is already being used by the display.

**← MD** Memory Data

| X Bus |
|---|
| 0 … 15 |

←MD mem during c3

Action: Memory data to X-Bus is single error corrected if MCtl bit 15 is set. The status of a given read operation can be found by looking in MStatus before the next memory read (←MD) is done. The occurance of both single and double errors are indicated here. This operation gives the contents of the memory cell specified during cycle 1, independent of whether a write was specified during cycle 2.

**Map←** Map Reference

| YH Bus | Y Bus |  |
|---|---|---|
| 0 … 7 | 0 … 7 | 8 … 15 |
| 16 (or 14) -bit virtual page number |  |  |

Map← in c1 only

Acton: This action is the same as a MAR← except that the physical address is derived differently. An access is started in the 65K - 80K bank of memory. The location accessed is specified by the page number.

**Refresh**

Refresh during cycle 1 of click. Ignored in c2 and c3.

Action: A RAS only cycle is initiated in all memory chips. Row Address is supplied from an internal 7 (or 8) bit counter which is incremented once per occurance of refresh.
DO NOT USE refresh if the display is using the low bank of memory during that cycle. No refresh will occur.
This can be guaranteed if used only in click 4.

**MCtl←** Memory Control Register

MCtl← during any cycle.

Y Bus

| 0 … 3 | EN Clr.<br>4 | Pt 0<br>5 | Pt 1<br>6 | Pt 2<br>7 | A<br>8 | B<br>9 | C<br>10 | D<br>11 | E<br>12 | F<br>13 | 14 | EN Cor<br>15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

clear error log for this Task (bits 4; 5-7)

Set bit = 1 to invert corresponding check bit written into memory. Testing only. (bits 8-13)

Not Used (bit 14)

Set = 1 Inhibit Correction (bit 15)

Action: Normally this register is set to 0. A-F can be set to one to test syndrome bits and error indications. Individual bits of the error log can be cleared by setting bit 4 and using Pt0-2 to specify the bit to be cleared. Bit 15, Inhibit correction, affects only the data being read. Check bits are always generated and stored in memory during writes.

**← MStatus** Memory Status

← MStatus during any cycle.

Action:
This register is loaded every time memory data is read by the processor (←MD). High byte has status of most recent memory access. Low byte latches any occurance of double error on a per task basis. Register is 0 if no errors logged.

X Bus

| A<br>0 | B<br>1 | C<br>2 | D<br>3 | E<br>4 | F<br>5 | S Err<br>6 | D Err<br>7 | T0<br>8 | T1<br>9 | T2<br>10 | T3<br>11 | T4<br>12 | T5<br>13 | T6<br>14 | T7<br>15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Syndrome Bits (bits 0-5)

Single Error (bit 6)

Double Error (bit 7)

Error Log (bits 8-15): If bit = 1 then a double memory error has occurred in indicated task (T0-T7) since last time the bit was cleared.

Figure 18. CP Memory Interface Summary

## 3.2 Error Correction

Since soft errors can occur in the memory (alpha particles from the package, etc.) error correction circuitry is included in the memory system. Six check bits added to the 16 bit word provide single error correction and double error detection (SEC-DED). No explicit indication of single errors is provided, although the status of any particular operation can be read from the Status & Errors (MStatus←) register after an operation. Error correction can be disabled, and the check bit positions in memory selectively set by writing into the MCtl register and reading the MStatus register.

A double error signal is available and also latched on a per task basis in the MStatus register. Thus, a task, upon entering a critical data transfer phase, could clear its particular bit, perform the task, and then check to see if its bit was set (double error). If an error did occur, its effect would be limited to events in that interval, over which some corrective action might be taken. If the emulator task caused the double bit error, a kernel trap is taken to location 0. See section 2.5.5.2, "Error Traps."

The following calculations yield probabilities of errors due to independent random processes in each chip. They do not include correlated events such as power line transients or static discharges which could affect all of the chips at the same time. A memory with 22 bits/word is assumed.

If the chips are assumed to (hard) fail at a rate of one per 2.5 million years (.04%/1000hr), then the mean time to a chip failure in a memory system with 12 banks (192K or 768K) is 9470 hours (13 months). By contrast, the mean time to failure with 4 banks (256K) is 28,410 hours (3 years, 3 months).

The soft error rate for the chips is assumed to be 1%/1000 hours. Following are the probabilities of 0. 1, and 2 soft errors in a 22 bit word in a 10 hour period. 10 hours was selected as the interval over which errors could accumulate. with the system being reset after 10 hours. The mean time between single errors is 38 intervals and the mean time between double errors is approximately 36.200 intervals. (It should be pointed that these probabilities are those that one would expect to measure with a program which continually scans through all memory cells looking for an error. If a program is confined to a small segment of memory, it would perceive a proportionately smaller probability of soft error.)

Prob.(1 single error in 22 bit word in 12 bank system in 10 hr. interval) =.0263

Prob.(1 double error in 22 bit word in 12 bank system in 10 hr. interval) =2.76 x $10^{-5}$

The following table shows the interpretation of the syndrome bits which can be read with the ←MStatus function after a memory read. The code table shows how the syndrome bits A-F are generated. When checking, syndrome bit F is parity over the entire word.

| *or* (A-F) | F | Meaning |
|---|---|---|
| 0 | 0 | no errors or >2 errors |
| 0 | 1 | not possible |
| 1 | 0 | double bit error |
| 1 | 1 | single bit error |

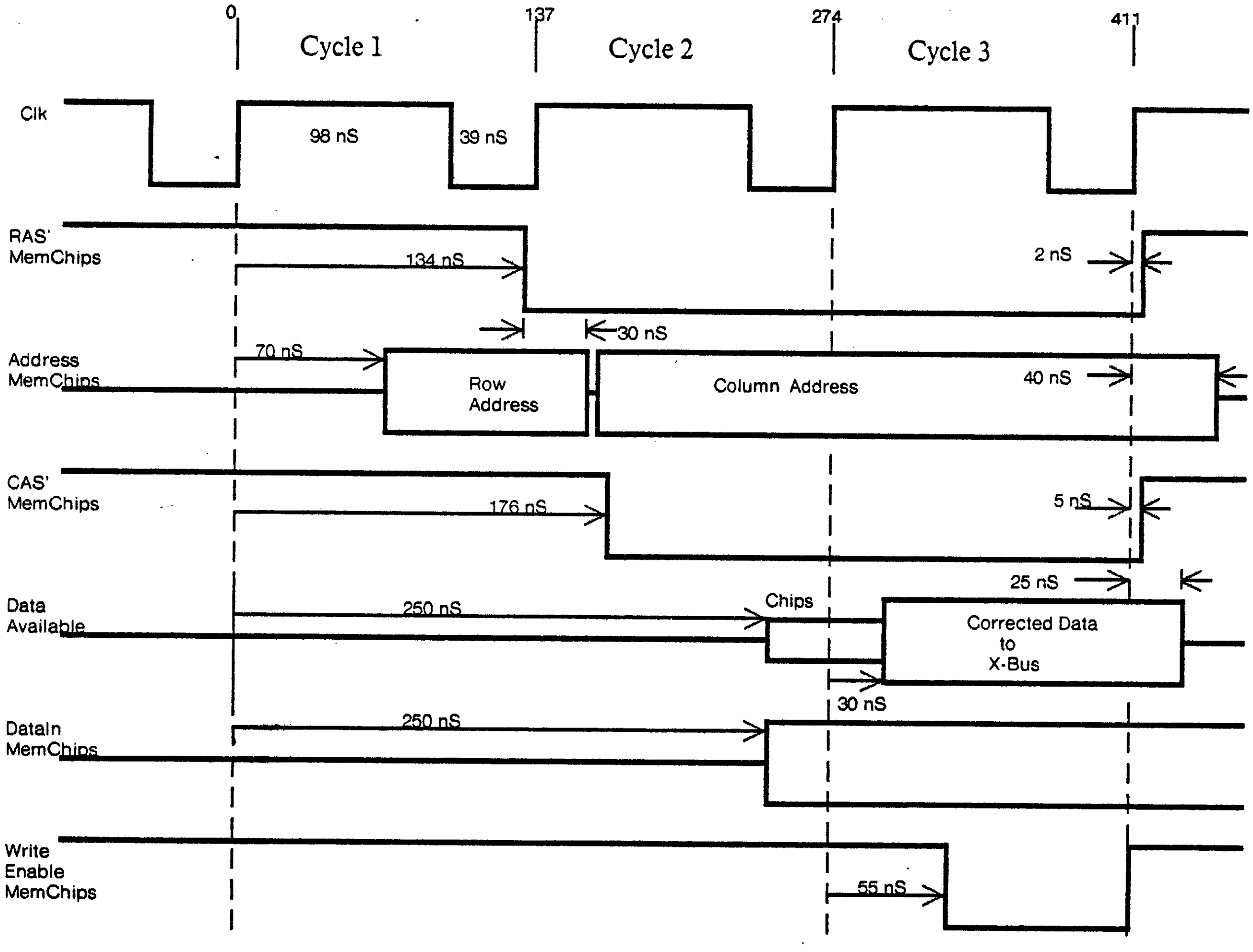
The SEC-DED code was optimized for 9-input parity chips. The following code table shows how the syndrome bits A-F are generated. Each row represents the inputs to a single parity chip. For example, syndrome bit A is the *xor* of data bits 0-3 and 10-13. Bit 0 will be inverted (corrected) during reading when A-F equals 110001 (from the column under 0). Any of the syndrome bits can be inverted when being generated by setting the corresponding bit in the MCtl register.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | a | b | c | d | e | f |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | + | + | + | + |  |  |  |  |  |  | + | + | + | + |  |  | + |  |  |  |  |  |
| B | + |  |  |  | + | + |  | + | + | + |  |  |  | + |  | + |  | + |  |  |  |  |
| C |  | + |  |  | + |  | + |  | + | + |  | + | + |  | + |  |  |  | + |  |  |  |
| D |  |  | + |  |  | + | + | + |  | + | + |  | + |  | + |  |  |  |  | + |  |  |
| E |  |  |  | + |  |  | + | + | + |  | + | + |  | + |  | + |  |  |  |  | + |  |
| F | + | + | + | + | + | + |  |  |  |  |  |  |  |  | + | + |  |  |  |  |  | + |

## 3.3 Memory Timing

Typical processor timing is shown in figure 19 below. The memory address must be valid on the Y and YH busses early enough that the proper bank is selected and address lines valid for RAS' (row address strobe). The column address bits are latched by the RAS' signal. The CAS' (column address strobe) signal occurs 42 nS after the RAS' signal and latches the column address in the memory chips. Data becomes valid at the output of the chips at a maximum of 150 nS after RAS' or 100 nS after CAS', whichever is later.

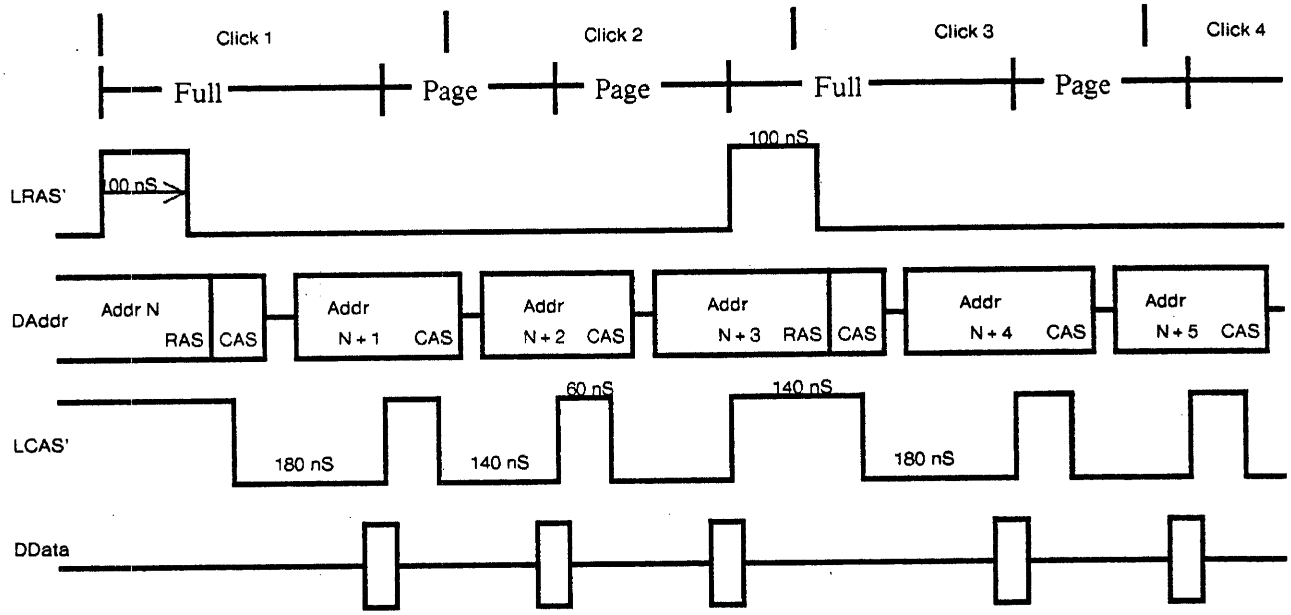
When writing into memory, the data to be written must be supplied during the second cycle of a click. The data is actually written in the latter half of the third click. Notice that up until the presence of the write pulse, all signalling is identical to a read cycle. The memory chips latch and hold the old data on their outputs during a write pulse if it occurs more than 150 nS after the RAS' signal. Thus, it is possible to write into a location and read data from it, all in the same memory cycle.



Normal Memory References through Processor Port

The display port supports both full and page mode accesses. The data delivered to the display port is not error corrected. The full access cycle time is 280 nS and the page mode access time is 200 nS. While the full access time is smaller than that specified in the data sheets (320 nS) for continuous operation, it is the average that is important, and the average cycle time in this case is 342 nS (6 full accesses per round, counting click 5). A page mode access occurs when the RAS' signal goes low and the CAS' signal cycles several times, strobing several different column addresses into the memory chips while retaining the same row address. (Because bit 12 is used during RAS, the maximum number of sequential page mode accesses between full accesses is 7, since bit 12 will change on every 8th access. The insertion of full accesses at the appropriate times is handled by the display controller.)

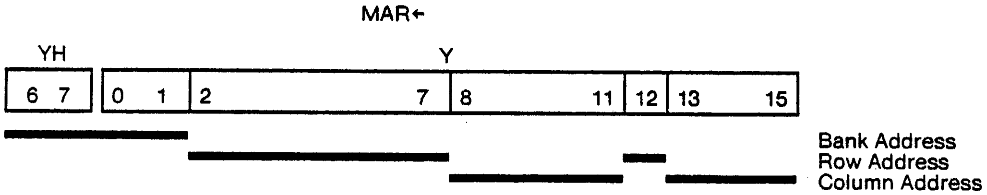
In normal operation, the display controller will seize the low bank of memory for 4 clicks of every round. It will start with a full access which is aligned on a click boundary, and then proceed with page and full accesses until the end of click 4. The other page or full accesses will not necessarily be synchronized with any click or cycle boundaries. They are packed so as to maximize the number of accesses during the 4 clicks the display has the memory.

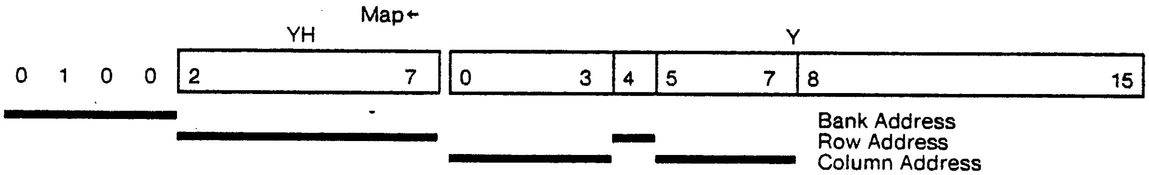


Display: Full and Page Mode Accesses

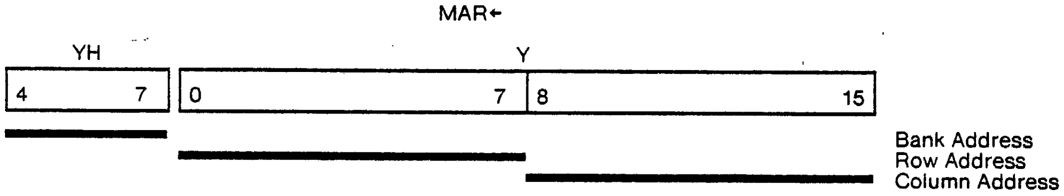
## 3.4 Row and Column Addressing

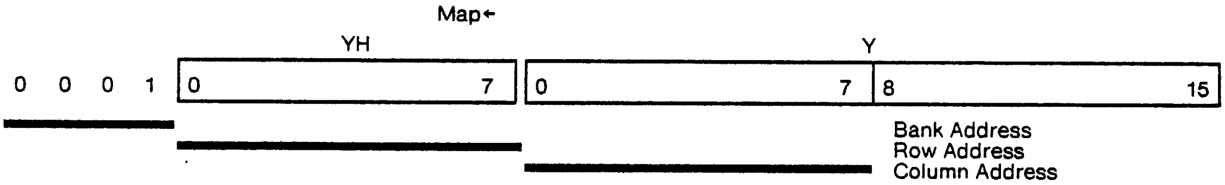
In the case of 16K chips (to which the Dandelion was originally desinged), one of the seven bits for the row address must come from the low byte. The maximum settling time of the high nibble of the low byte is too long if a carry from the low nibble occurs (sec. 2.3.8). Consequently, bit 12 (instead of bit 8) of the low byte is used during RAS. Consistent juggling occurs for map references so that this is invisible to the microcoder. The following figure shows how the row and column address bits map into the Y and YH buses for 16K chips:





When 64K chips are used, the row and column bits are "correct." The following table shows how they are derived from the Y and YH bus. (If it is known that only 64K chips are present in the system, the restriction that X bus arithmetic can not occur with Map← is no longer valid.)





# 4.0 Display Controller and Clocks

## 4.1 Overview

This chapter describes the Dandelion display controller. It is located on the high speed I/O (HSIO) board. Only the Display hardware is covered. The minimum microcode requirements are given.

## 4.2 Display Functions

The Dandelion large format display has the following parameters:

* 10" high by 12.8" wide bit map display.
* Separate Video, Horizontal and Vertical sync signals.
* Visible area = 808 lines x 1024 bits.
* Refresh rate = 38.7 frames/second (one frame every 25.8 ms)
* Memory used (808+16)*64 = 52.736 words in low 64k bank (16 lines for cursor).
* Border area = 26 lines at top. 26 lines at bottom, 32 bits at each side. Contents of user-settable register is repeated to form border pattern. Size of top and bottom borders set by microcode.
* Total frame (visible + non-visible) = 897 lines x 1088 bits.

The display hardware supports the scrolling of windows on the screen. These windows and cursors may be moved or scrolled vertically without actually moving bits in memory. Horizontal displacement requires the memory images to be moved.

Memory refresh is also performed by the display microcode.

## 4.3 Display Controller Hardware

The display controller uses a partitioned. two-port memory to reduce the loss of processor bandwidth while the display is running. The display controller blocks processor access to the low 64K memory bank *only* when it is acquiring data bits during an active horizontal line. The processor has complete access to the low bank at all other times (i.e. during one click in each round while the picture is being displayed. while the beam is turned off (blanking) and while the border is being displayed). When not being used by the display hardware, the low memory bank is identical in performance to the high banks. The display hardware cannot access the higher banks of memory and has no effect on processor access to these banks.

The following functions are performed by the display controller hardware/microcode.

1. Read data from memory and shift out blocks of 1024 bits.

2. Provide horizontal sync, vertical sync, and blanking signals.

3. Perform memory refresh.

Some versions of display microcode will automatically display a 16x16 cursor given its position. Others support smooth (continuous) scrolling of display windows. The hardware is constructed to support these features but does not supply them directly.

## 4.4 Partitioning Functions Between Hardware and Microcode

The tasks required of the display controller span a wide range of times (shifting bits, reading words, providing blanking and sync signals and composing fields and frames). It is important to minimize the amount of hardware used for any individual Dandelion controller while not requiring an excessive amount of the processor for a single I/O function. For the display controller, a horizontal line period (28.8 uS) was chosen as the dividing point between functions implemented in hardware and microcode. Memory accesses, parallel to serial conversion, and horizontal sync generation are done in hardware. Line counting, vertical sync, cursor insertion, scrolling support and memory refresh are handled with microcode. The hardware is capable of displaying only a single horizontal line. The microcode assembles the lines necessary to make a coherent picture.

## 4.5 Microcode - Hardware Interface

Display microcode uses three registers to control the display hardware. They are described below and summarized in the next figure. Use of this interface to operate the display will be described in the next section. The following terms appear in the discussion.

*Line Segment* - A subset of a horizontal line in which the displayed words come from contiguous memory locations. A line segment can be between 1 and 64 words long. The line segments which comprise a horizontal line must total 64 words in length. Each entry in the control FIFO (First-In-First-Out buffer) described below specifies one line segment.

*Window* - A rectangular region on the display made up of line segments on successive scan lines. The boundaries of the windows considered here are horizontal or vertical. The hardware does not preclude windows of arbitrary shape.

*Cursor* - This is a special case of window which is 16 scan lines high and two words wide. Contained in this region is a 16x16 array which is bit aligned. The remaining area in the two word wide area not covered by the 16x16 array is typically loaded with those bits from the main bit map over which the cursor is placed. The resulting image shows a 16x16 bit-aligned cursor.

Control Register

This register contains 7 bits which control the display operation.

> On - This bit enables requests to the processor for service during the display click. These requests begin at the end of every horizontal line and end when disabled by the display microcode. This bit does not affect memory accesses nor does it cause picture or border to be displayed. Its only function is to allow the processor to execute display-task microcode.
>
> Blank (Bk) - Setting this bit always causes the video beam to be turned off. No memory accesses will occur when this bit is set. Typically, the blanking bit will be set during vertical retrace.
>
> Picture (Pic) - Setting this bit will cause memory accesses unless Blank is also set (in which case there is no picture and there are no memory accesses). The contents of the control fifo is used to specify which locations are accessed and displayed. If both Pic and Bk are cleared, the border pattern will be displayed for all bits within a line and no memory accesses will occur. This is done to create the top and bottom picture borders.
>
> Invert (Inv) - Setting this bit causes the video signal to the monitor to be inverted. All areas of the screen (border and picture) shown while this bit is set will be inverted.

Odd (OD) - Setting this bit indicates to the controller that the odd field of a frame is being scanned. This is used by the controller to determine whether vertical sync pulse should start and stop at the beginning or middle of a line. It starts at the middle for an odd line. This bit should not be changed during a vertical sync pulse, since changing it during the sync pulse would cause the end of the sync pulse to occur at a different location in a line from where it started. Neglecting this could cause interlace problems on monitors triggered on the trailing edge of vertical sync. (Most of our monitors are triggered on the leading edge of vertical sync.)

Vertical (Vt.) - Vertical sync pulse line goes low when this bit is set. The exact time of the transition relative to a horizontal line time is determined by the odd bit.

Clear Control Fifo' (CCF) - When set to zero, this bit causes the contents of the control fifo to be declared invalid. The bits are not actually set to zero but the fifo is declared to be empty. Normally this bit is kept set to one. The Control Fifo should be cleared during each vertical retrace as a safety measure.

## Dandelion Display Controller Registers

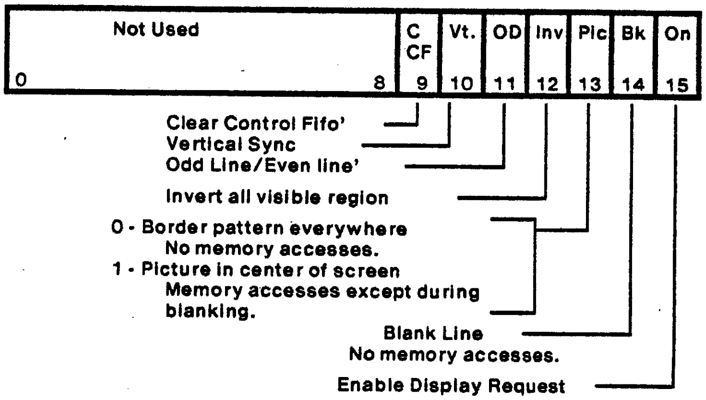
### Control Register (on X Bus)

DCtl may be written in any cycle.

Register controls display operation.

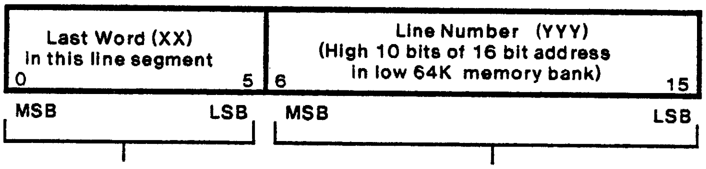
It is cleared to 0 by IOPReset when system is powered on. When cleared, the display will receive only horizontal sync, video will be the contents of border register at power-up, and there will be no display requests to the processor. There will be no vertical synchronization or blanking.

Vertical sync is a strobed version of the vertical bit in the control register. To produce interlaced scan, vertical sync is strobed and changes at the beginning of the line for even lines and middle of the line for odd lines, as specified by bit 11.



### Control Fifo Register (on Y Bus)

DCtlFifo may be written in any cycle.



Selects location (word #) along horizontal line after which display will jump to a new line in memory from which it reads data.

Horizontal line starts reading data at

| YYY | 00 |
|---|---|

in main memory, and continues reading until location

| YYY | XX |
|---|---|

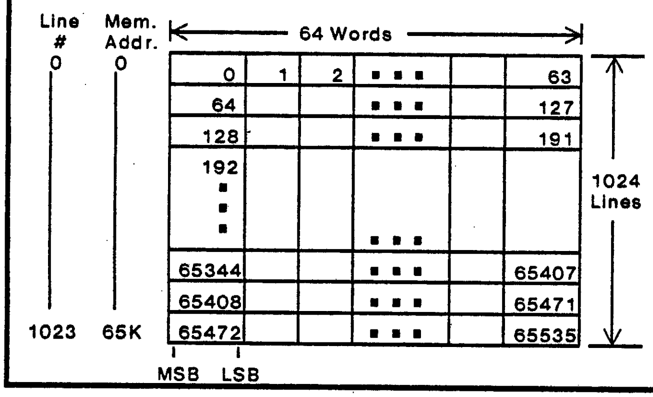
AFTER which it advances to the next control Fifo location specifying the next line segment:

| YYY' | XX + 1 |
|---|---|

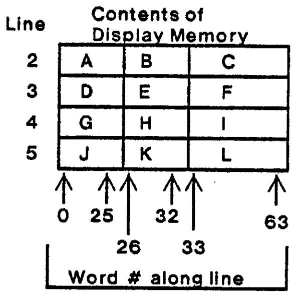
on the same output line.

One word is written into Fifo for each cycle in which DCtlFifo is asserted. Used to specify a location (line segment) in memory from which to retrieve data for display. The low 10 bits specify the line number, and the high 6 bits specify the Last Word to be read before changing line segments. If this Last Word is not 63 (end of line) then the next Fifo entry is used to identify the next line number from which a group of words will be taken. Note that the low 6 bits (word #) used for the address is incremented from 0 to 63 in each line. The control fifo only permits selecting the line number and the location along the line at which a transition is made from one line to another. Thus, as viewed on the monitor screen, this mechanism facilitates vertical movement of images, but not horizontal movement, since low 6 bits of address come from word counter. One control word is loaded into the fifo for each continuous segment of words in a horizontal line. Thus, a normal line with no cursor or window will have one control word. A line with a cursor in the middle will have 3 control words. While the fifo size is 16 words, no more than 10 entries should be for a single line. The last control word for a line should specify word 63 (decimal). The controller will "wrap around" to the next scan line in a field if necessary to advance to the word number specified in a control fifo entry. Control words can be loaded into the Fifo any time before the line in which they are used. Care must be taken not to insert extra control words, and lose sync. Clearing control fifo during vertical sync will purge the fifo. The control Fifo is cleared by setting bit 9 of control register to 0.

Display Memory (Low 64K bank of physical address space)



Example of Control Fifo Use



Contents of control fifo just before beginning of line X.

| | |
|---|---|
| 25 | 4 |
| 32 | 2 |
| 63 | 5 |

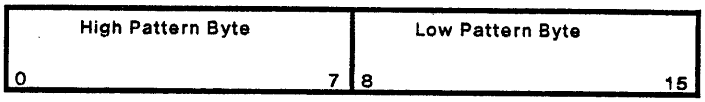
Line X as displayed on screen.

| * | G | B | L | * |
|---|---|---|---|---|

* 32 Bits of border pattern 1 byte repeated 4 times.

### Border Pattern Register (on Y Bus)

DBorder may be written in any cycle.



High pattern is repeated on lines 4n + 2, 4n + 3 where n is an integer.

Low pattern is repeated on lines 4n, 4n + 1 where n is an integer.

Border pattern is repeated on every line during the first and last 32 bits scanned. The high and low patterns are used on alternate pairs of lines. Lines are numbered starting from 0 at the top of the screen. The border pattern will be repeated all across a horizontal line if bits 13 (Pic) and 14 (Blank) of the control register are both 0.

Control Fifo Register

This register contains two fields; *last word* and *line number* which are used to specify a line segment.

> *Last word* is used to specify the number of the last word position (relative to lines aligned on 64 word boundaries in memory) to be used for a given line segment. *Last word* is the high 6 bits of the control fifo entry, and typically remains constant for a given window. The *last word* field of the control fifo entry for the last segment in a horizontal line must be 63.
>
> *Line number* is used to calculate the memory addresses in which bits for the displayed line segment are found. The controller hardware maintains a 6 bit counter for addressing words within a horizontal line. This counter always counts from 0 througn 63 as the line is displayed. The low 6 bits of the current memory address are the controller's 6 bit count. The high 10 memory address bits come from the *line number.* When the controller's count matches the *last word* from the current control fifo entry, the next fifo word containing the next *line number* is fetched. Using this mechanism, the user can define the mapping between memory address and screen position. Note the low 6 bits of memory address are not involved the mapping; they always specify the word's horizontal position on the screen. The display hardware supports only vertical displacements. For example, word 0000 in memory may be shown on any line of the display but must always be the first word in the line. This line number is typically incremented by 2 (because of interlacing) for successive lines within a window.

Border Pattern Register

This register contains the two border pattern bytes. Only one of the bytes is used in any given scan line. The low border byte is used during lines 4n, and 4n+1 (lines 0,1,4,5,8,9...) and the high border byte is used during lines 4n+2 and 4n+3 (lines 2,3,6,7,10,11....). The proper byte is repeated 4 times at the beginning and end of each horizontal line to form the side borders. If the Picture and Blank control bits are both off, the byte is also used to fill the picture area. The top and bottom borders are created in this fashion. The border pattern register need only be loaded once.

### 4.6 Using the Controller

In the Dandelion architecture, the processor is shared among a number of microcode *tasks.* One of these is a high level language emulator; the others control I/O devices. The processor is used in round-robin fashion by the tasks. Each I/O task is assigned one or more *clicks* in the processor *round.* There are five clicks per round. A task may perform one main memory access in parallel with three microinstructions in a click. The display is assigned click number 4 of each round. Clicks not used by their assigned I/O tasks are available to the emulator.

Each round takes 2.055 uS to execute. There are exactly 14 rounds per horizontal scan line (the processor clocks are derived from the display clock so there is no skew). Thus the display microcoder must ensure that any action scheduled to take place in one scan line can be done in 14 clicks.

This section outlines the actions the microcode must take to get an image in the low 64K of memory shown on the display. The following figure shows what is loaded into each register during the various parts of a frame. Note that the only differences between the "odd" and "even" fields of a frame are the setting of the odd field bit in the control register, the line offset used when loading the control fifo and the length of the vertical sync pulse.

Note also that the parameters for line n must be loaded during line n-1. For example, parameters for the first picture line are loaded during the last border line at the top of the screen. Assuming the microcode used to set the control and control fifo registers runs once per scan line; the proper order is:

Second to last Top Border line: Send a 41'X to DCtl (display line of border).

Last Top Border line: Send a 41'X to DCtl, load DCtlFifo with parameters for first line of screen.

First Picture Line: Send 45'X to DCtl (display picture) and load DCtlFifo with parameters for second picture line.

**Register Loading Sequence to Get Bit Map on Display**

| # Lines | Function | Control Register Loaded only once per function during first line | Control Fifo Loaded once per line in even and odd fields. |
|---|---|---|---|
| 1 | End Vert Sync, Blk | 3 | — |
| 13 | Top Border | $41_{16}$ | — |
| 404 | Even Field | $45_{16}$ | Last Word = 63<br>Line Number = even #'s |
| 13 | Bottom Border | $41_{16}$ | — |
| 18 | Start Odd V Sync | $73_{16}$ | — |
| 1 | End Vert Sync, Blk | 3 | — |
| 13 | Top Border | $41_{16}$ | — |
| 404 | Odd Field | $45_{16}$ | Last Word = 63<br>Line Number = odd #'s |
| 13 | Bottom Border | $41_{16}$ | — |
| 17 | Start Even V Sync | $63_{16}$ | — |

897 lines total 448.5 lines/field

numerical base

To add a cursor, the control fifo is loaded with 2 or 3 segments per line for a run of 8 lines in each field. This is shown in the next figure. Showing a window in addition to the cursor requires more segments per line. For both the cursor and the window, some computation must be made once per frame to determine the control fifo entries. In addition, the cursor bitmap must be updated each time the cursor moves.

## Register Loading Sequence to Get Bit Map with Cursor

| # Lines | Function | Control Register Loaded only once per function during first line | Control Fifo Loaded once per line in even and odd fields. |
|---|---|---|---|
| 1 | End Vert Sync, Blk | 3 | ——— |
| 13 | Top Border | $41_{16}$ | ——— |
| x | Even Field | $45_{16}$ | Last Word = 63<br>Line Number = even #'s |
| 8 | Even Cursor | $45_{16}$ | 3 Seg. for cursor |
| 396-x | Even Field | $45_{16}$ | Last Word = 63<br>Line Number = even #'s |
| 13 | Bottom Border | $41_{16}$ | ——— |
| 18 | Start Odd V Sync | $73_{16}$ | ——— |
| 1 | End Vert Sync, Blk | 3 | ——— |
| 13 | Top Border | $41_{16}$ | ——— |
| y | Odd Field | $45_{16}$ | Last Word = 63<br>Line Number = odd #'s |
| 8 | Odd Cursor | $45_{16}$ | 3 Seg. for cursor |
| 396-y | Odd Field | $45_{16}$ | Last Word = 63<br>Line Number = odd #'s |
| 13 | Bottom Border | $41_{16}$ | ——— |
| 17 | Start Even V Sync | $63_{16}$ | ——— |

897 lines total 448.5 lines/field

numerical base

During Cursor, 2 or 3 control fifo entries per line are used.

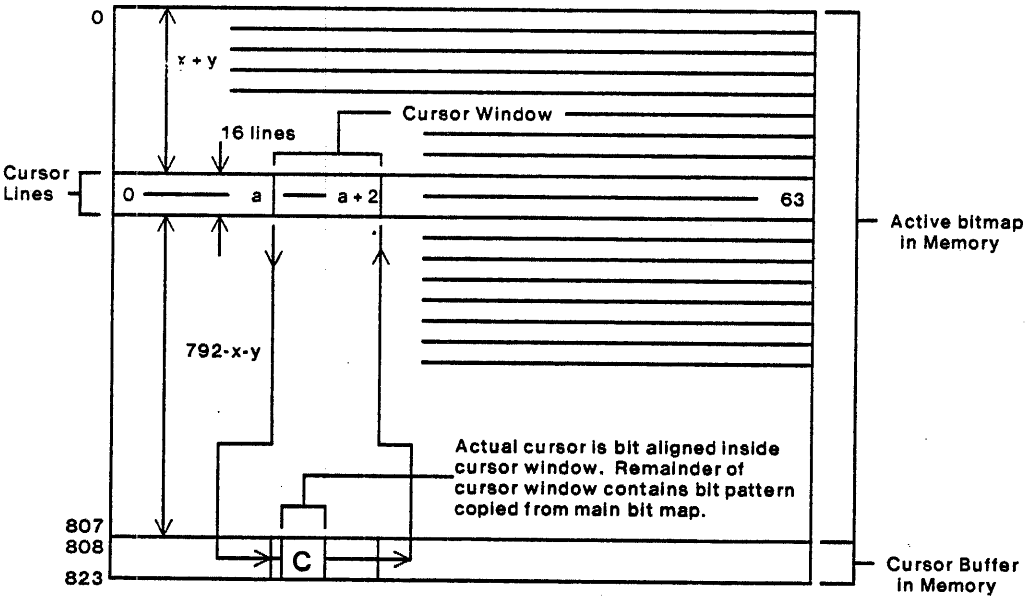
| Last Word | Line Number |
|---|---|
| a | x+y + 2n |
| b | 808 + 2n |
| 63 | x+y + 2n |

| | |
|---|---|
| a | x+y+1 + 2n |
| b | 809 + 2n |
| 63 | x+y+1 + 2n |

n is line increment within cursor ranging from 0 to 7 for successive lines in a field.

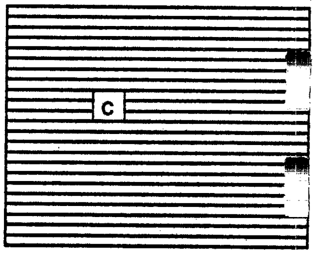
The size of the bitmap required may be reduced if more border pattern is shown or lines from memory are shown more than once on the screen.

### Memory



Actual cursor is bit aligned inside cursor window. Remainder of cursor window contains bit pattern copied from main bit map.

### Display Screen



During Cursor lines, each line is divided into 3 segments. The middle segment comes from the cursor bitmap.

Segment 1 Word 0 to a<br>
Segment 2 Word a + 1 to a + 2<br>
Segment 3 Word a + 3 to 63

Horizontal cursor position is set by horizontal position of cursor image in Cursor Buffer. Vertical cursor position is set by line number at which the image is displayed.

## 4.7 Display Hardware Implementation

### Display Controller (Horizontal line generator)

The display hardware handles only those functions which repeat on a horizontal line basis. If the processor had provided these functions, a great deal of its bandwidth would have been required for a fairly simple, repetitive task. Similar hardware for counting lines and controlling windows is not used because the processor bandwidth required for these tasks is available.

Horizontal Events

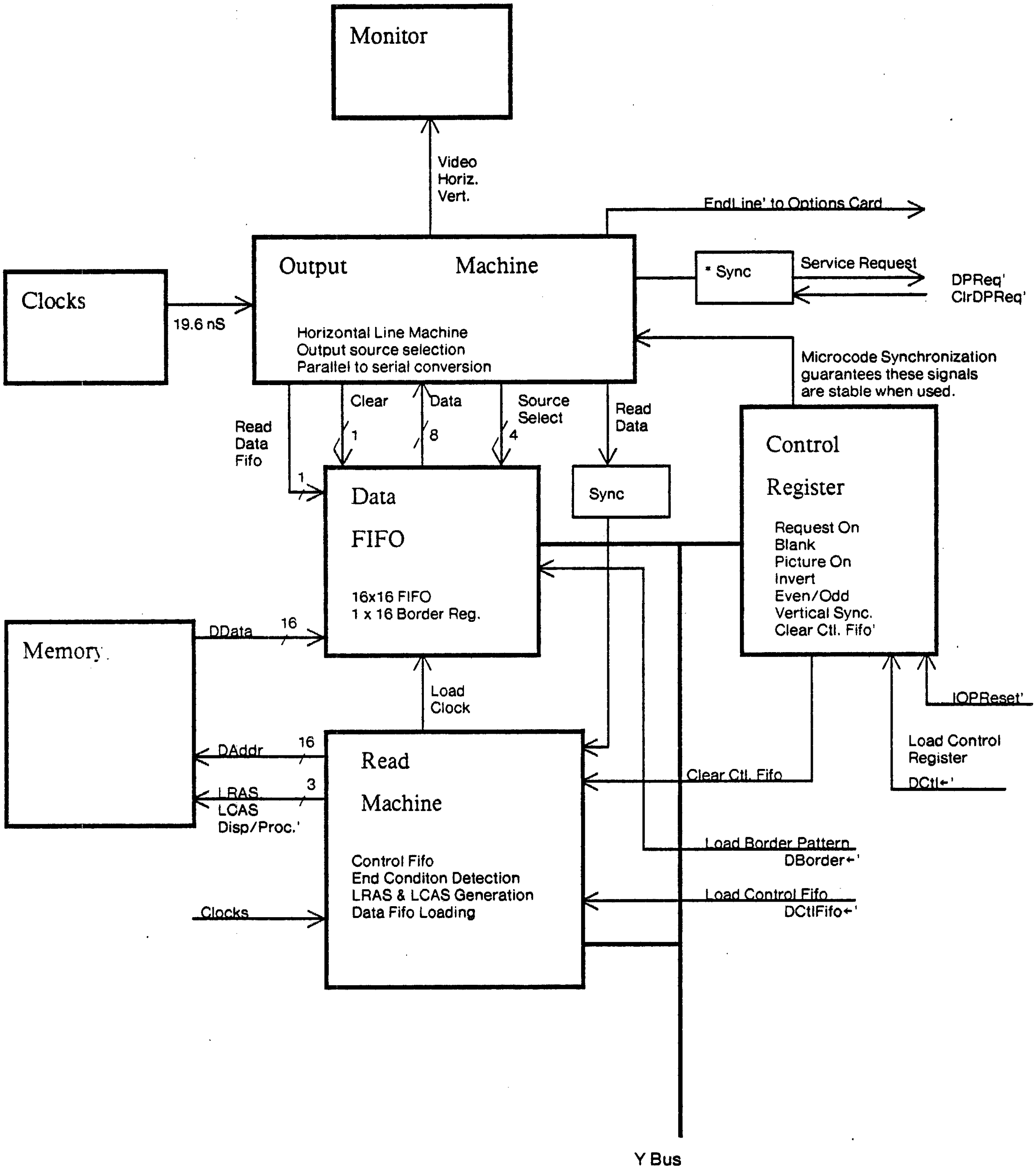
A horizontal line contains 32 bits of border pattern on the left, 1024 bits of picture, 32 bits of right border, and 382 bits of blanking. A horizontal sync pulse starts 8 bits after blanking starts and ends 8 bits before blanking ends.

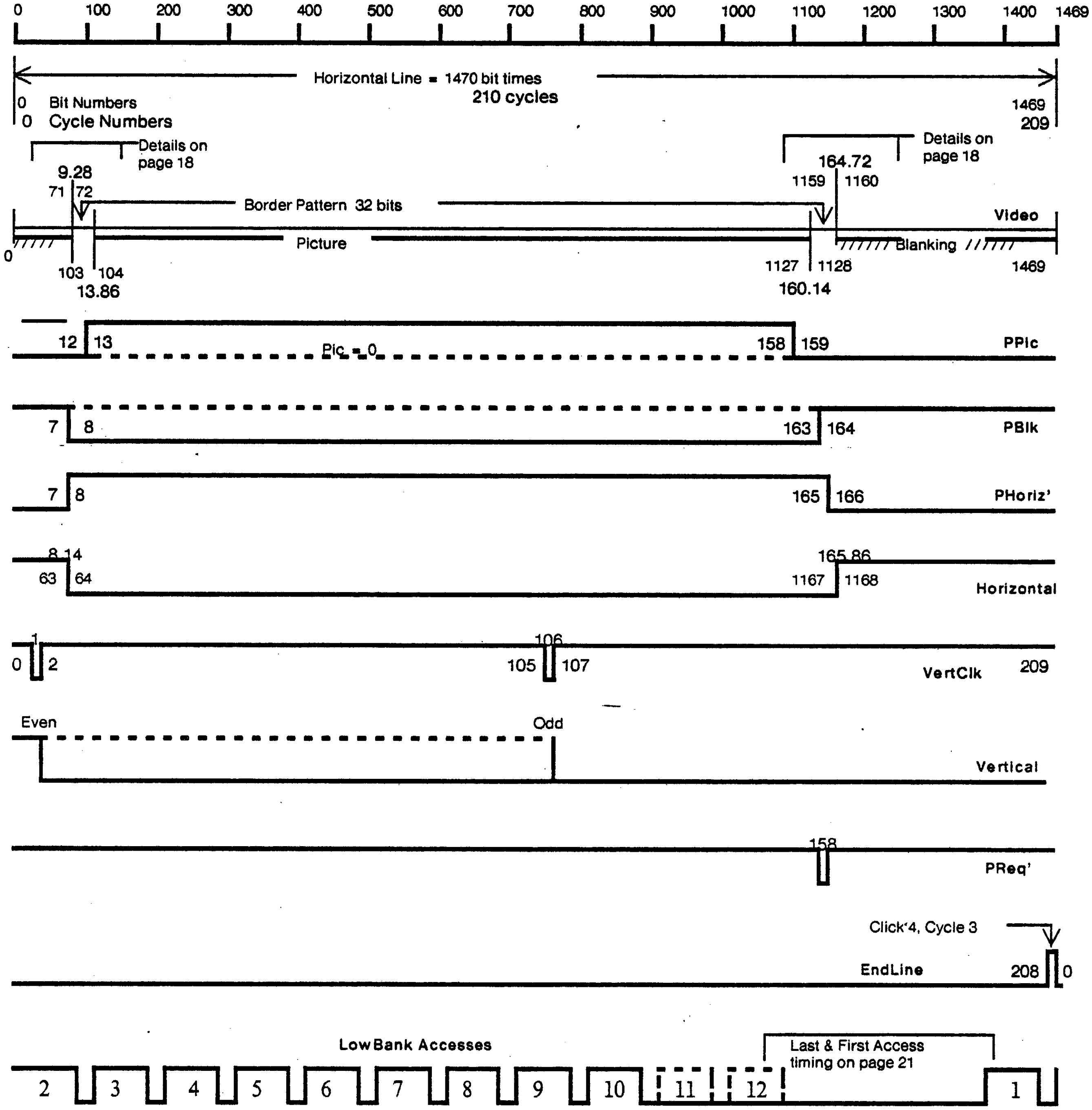
Vertical Events

A frame consists of an even and an odd field, each of which contains 13 lines of top border pattern, 404 lines of picture, 13 lines of bottom border, and 18.5 (18 lines in one field and 19 lines in other) lines of blanking during which vertical retrace takes place. The section covering controller use further describes vertical events. No further mention of vertical events will appear in this section.

The next two figures show a functional block diagram of the display controller and output machine timing diagrams. It has three principal parts; the output machine, a data fifo, and the read machine. Associated with the output machine is the control register. The border pattern register is associated with the data fifo. The read machine contains the control fifo and associated control fifo register, end condition logic to terminate memory accesses at the end of a round and at the end of a line, and the LRAS and LCAS memory clock generation. ¬Following are descriptions of these sections.

Display System Functional Block Diagram

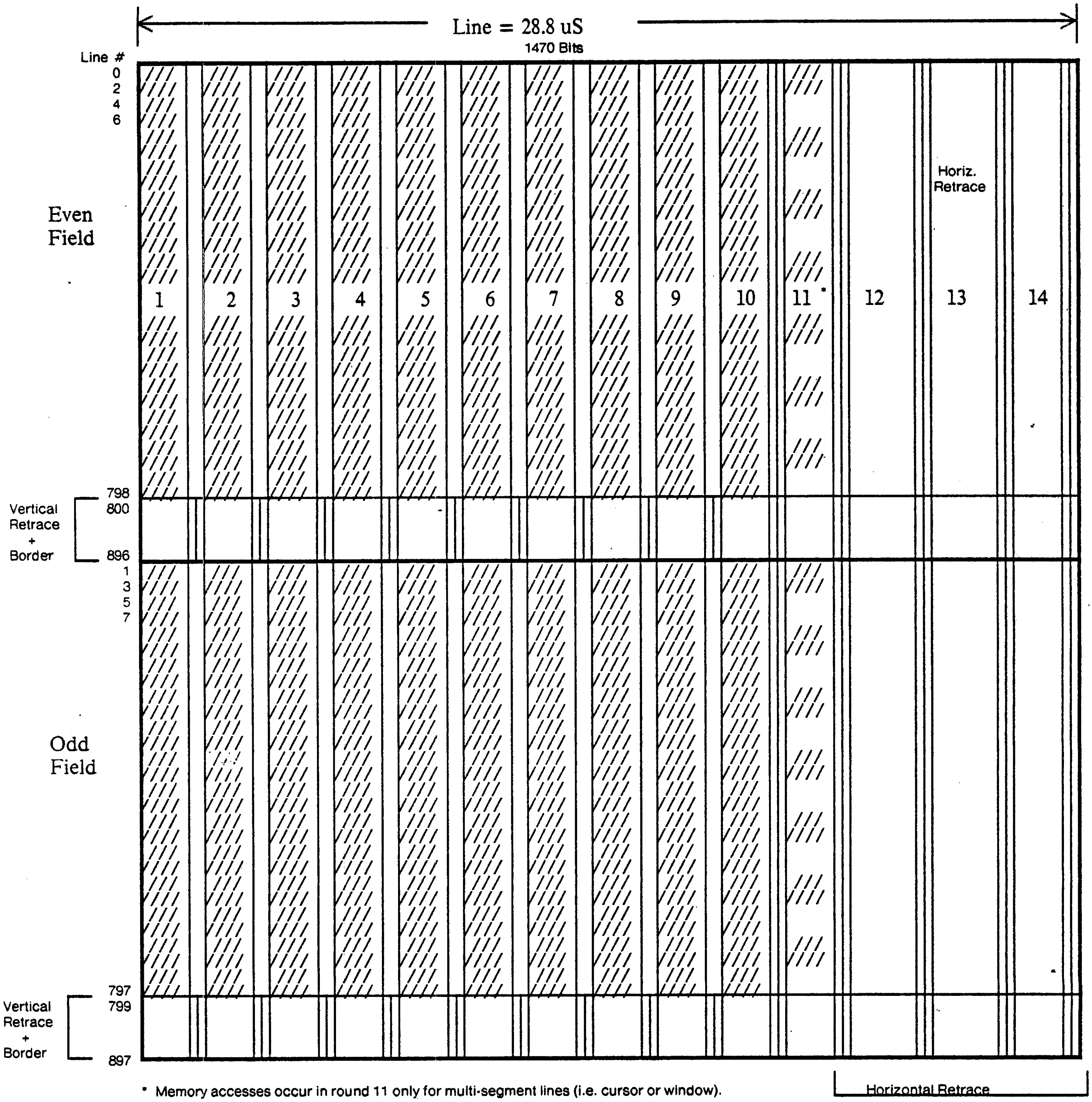




There are low bank accesses during rounds 1-10 for a line with 1 segment. Some amount of round 11 is used for lines with 2-10 segments. While not recommended, round 12 is available for lines with more than 10 segments. Care must be taken here since the data fifo can go empty in this round if more than 10 segments are specified. Word must be strobed into DataFifo at least 330 ns before it is read out. At beginning of line, DataFifo is filled to 13 words at the end of round 2.

| | | | |
|---|---|---|---|
| 3 cycles/ click | 210 cycles/ line | Bit Time = 19.59 nS | Cycle Time = 137.14 nS |
| 5 clicks/ round | 7 bits/ cycle | Bit clock = 51.04 MHz | Round Time = 2.0572 uS |
| 14 rounds/line | 1470 Bit times/line | | Line Time = 28.8 uS |

Figure x. Timing For Output Machine



* Memory accesses occur in round 11 only for multi-segment lines (i.e. cursor or window).

1 Frame = Even Field + Odd Field
Field = ~~399~~ 404 Active Lines + ~~32~~ 26 Border Pattern Lines + ~~17.5~~ 18.5 Blank LInes (retrace)

▨ Low 64K bank Memory used by display

▥ Processor use during display click 5 by display & memory refresh.

Figure x. Resource Use During Display Frame

## Output Machine and Control Register

The output machine will be described in terms of the actions that take place during the output of a horizontal line. Each horizontal line starts with 32 bits of border pattern, followed by 1024 bits of data from memory, followed by 32 bits of border pattern, and ending with a blanking period during which the horizontal retrace takes place. This sequence repeats every horizontal line and is shown in the output machine timing diagram.

The output machine is controlled by a prom state machine with 210 states (1 state per machine cycle). It is cycled through these states by the display prom counter, which is a part of the system clock. The timing diagram shows the outputs of the prom register marked with asterisks. There are two time references in this figure. There are 1470 bits per line and they are marked on the top of the figure. There are 210 cycles of 7 bit times each, which are labeled in italics.

Starting at bit position 0 (look at line labeled video), the F16 counter in the output machine has just been resynchronized to 0 by the EndLine and Tick7' pulse. The output shift register and blanking register are strobed at the beginning of bit 8 and every 8th bit thereafter during a horizontal line. Thus, shift register loading, blanking, and unblanking are done on byte intervals.

The first 32 bits of a line come from the border pattern register. The selection of the high or low byte is done by a flip-flop which is toggled every horizontal line in a field. This produces the signal BPBS (border pattern byte select). This signal is always reset by a vertical sync pulse so the first line of a field comes from the low byte of the border pattern register. (More specifically, it is the first line after the trailing edge of the vertical sync pulse. Note that since the first few lines of a field are often blanked, it may not correspond to the first visible line.)

On the first cycle boundary after the 4th border byte is loaded, PPic goes to a logic 1, such that the next byte loaded comes from the high byte of the data fifo. Byte selection is performed by the high bit of the bit counter. The fifo is clocked after the high byte is loaded. This process continues until all 64 words have been loaded into the shift register and shifted out. While the low byte of the 64th word is being shifted out, PPic goes low so that the next byte to go out comes from the border register. While the 4th border byte is being shifted out, PBlk comes on so that blanking starts on the next byte boundary. Blanking continues until the end of bit 71 (after the counter wraps around), after which the next horizontal line starts with the border pattern again.

The horizontal sync pulse starts 8 bit times after blanking starts and ends 8 bit times before blanking ends. Both horizontal and vertical sync signals pass through a low pass filter which increase the rise and fall times to approximately 100 nS. This helps reduce high-frequency radiation from the cable going to the monitor.

### Data Fifo

A 16 word data fifo provides buffering and solves the problem of synchronization between the memory system and the output machine. (While both memory and output machine run from the same clock, the largest common period is the 19.6 nS clock period which is too fine to be of any value.) Data is strobed into the holding register and fifo with DCAS' and DCASDly', respectively, both of which come from the read machine. Words are read out of the fifo with the signal ReadDataFifo, which comes from the output machine. The outputs of both the data fifo and border register are multiplexed onto a byte wide tri-state bus, then through TTL-ECL converter to the parallel input of the output shift register in the output machine. Selection of the appropriate output byte is done by the output machine. The output machine controls the read machine such that the fifo never overflows or underflows during a line.

### Read Machine

The read machine does memory accesses during the first 4 clicks of a round. It always starts at the beginning of a round and continues to either the end of the round or the last word of a line has been accessed, whichever occurs first. Reads in a round are initiated by a signal, PD/P, from the output machine. The read machine will determine the mix of full and page mode accesses necessary and do the maximum number of memory accesses possible within a round. The low 6 bits of the memory address always count from 0 to 63. The high 10 bits (line number) are specified in the control fifo entry. The last word to be used from a given line is also specified in the control fifo entry (6 bits) and is used to advance to the next fifo entry when that word number is reached. The three parts of the read machine (control fifo, end condition logic, and LRAS, LCAS generation) are described in the following paragraphs.

### Control Fifo

The control fifo contains 16 entries. Each entry identifies a line segment using 10 bits to specify the line number and 6 bits to specify the last word in the segment. The control fifo is loaded from the Y-Bus, unloaded by a signal from the end condition logic, and cleared by a bit from the control register. The microcode must take care to load only the entries for one scan line per horizontal line wakeup on the average. The control fifo should be cleared once per vertical field to eliminate the effects of noise and assure its state at the beginning of a field.

### Word Counter & End Condition Logic

The word counter counts from 0 to 63, synchronous with the memory accesses used to fill the data fifo. The output of this counter is compared with the 6 bit last word field of the current control fifo entry. When they are equal, the control fifo is advanced to the next entry. There is also logic to determine when a full (RAS and CAS) memory reference should take place. A full reference must take place whenever one of the RAS bits at the memory chips changes. This can occur on the first reference in a round, when the control fifo is advanced, and on every 8th memory reference due to the arrangement of bits in the memory system.

The number of accesses in a round depends on the number of full (293 nS) and page mode (215 nS) accesses that occur. A maximum of 5 full accesses, 4 full and 2 page accesses, or 1 full and 6 page accesses can occur. Thus the total number of accesses can range from 5 to 7. A prom state machine looks at the combination of accesses and drops the signal EndRndRead' during the last access of a round. The accesses in a round can end early if word 63 is reached. The signal InhibitRead also becomes true after word 63, locking out any further reads, independent of PD/P signal from output machine, until is reset by the signal ClrDataFifo' from the output machine. Details of the state machine and other logic timing are in the Clock and Display drawing package.

### LRAS-LCAS Generation

The signals LRAS and LCAS are the clocks for the low bank of the memory system. These signals are identical to RAS and CAS for processor memory references (411 nS cycles), but have a faster full cycle time (293 nS) and a page mode cycle (215 nS) when the display is using the low bank (indicated by Disp/Proc' in the high state). In all cases, CAS follows RAS by 49 nS. Both of these generators are simple state machines using one counter and discrete logic for decoding. They have a 19.6 nS cycle time.

## 4.8 Clock Generation

The CP cycle clock (137.14 nS) is derived by dividing the display's bit clock by seven. The next figure shows the relationships between the clocks generated on the HSIO card.

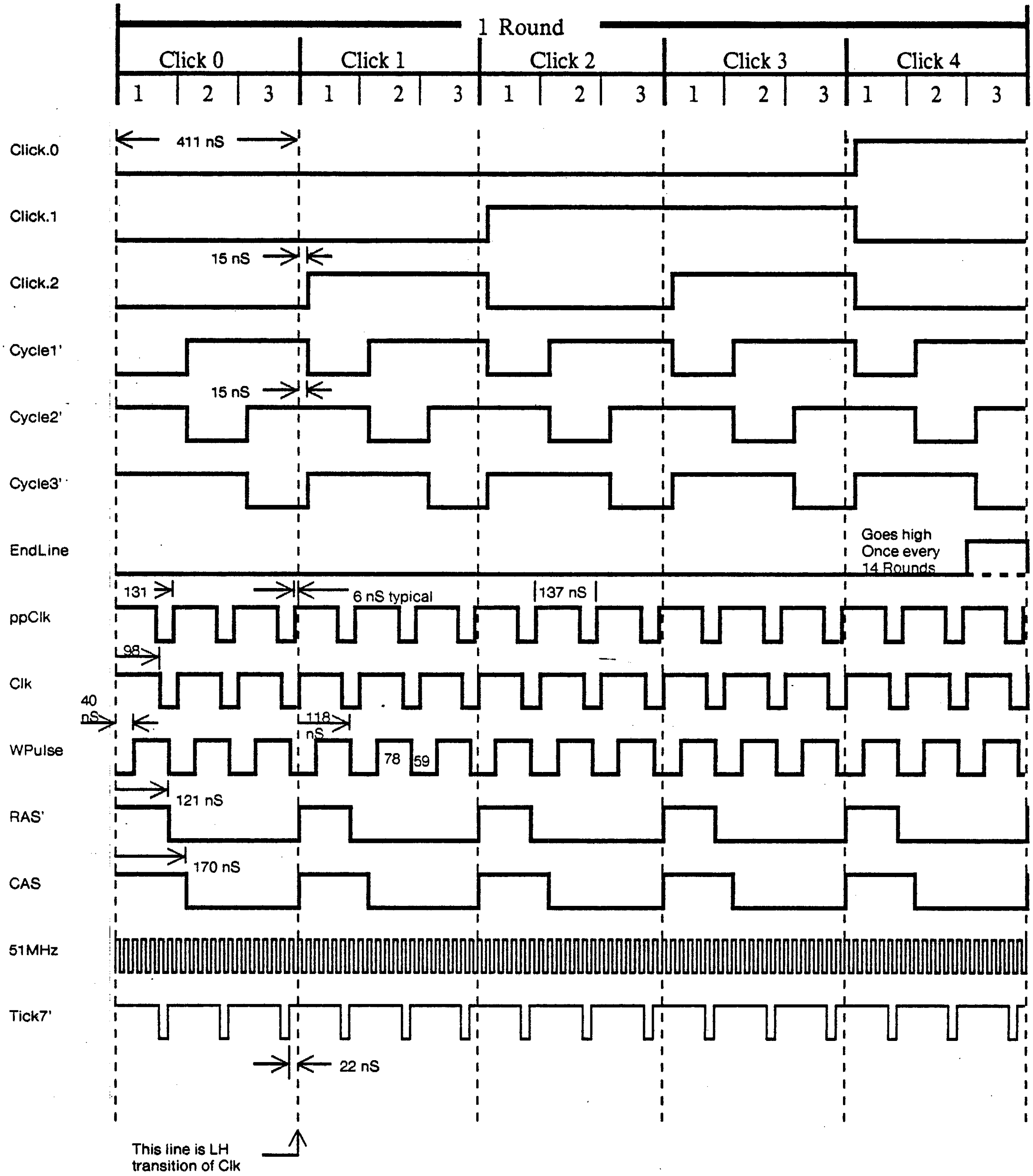


Figure x. System Clocks (Backplane Timing)

# 5.0 Disk Controllers

Two types of rigid disks can be controlled by the Dandelion. Section 5.1 discusses the Shugart disk controller located on the HSIO card. Section 5.2 describes the Trident disk controller, which is found on the HSIO-L card.

## 5.1 Shugart Disk Controller

### 5.1.1 Overview

This chapter is concerned with the Dandelion's controller for the Shugart SA4000 and SA1000 type disks. It identifies the major components of the system and their connections. It is assumed that the reader will have read the *SA4000 Fixed Disk Drive OEM Manual* and *SA1000 Fixed Disk Drive OEM Manual* from Shugart. This chapter is concerned with the function of the disk controller, not of the disk drive.

There are four major blocks in the Dandelion Disk Controller . They are the Input Conditioning, Output Conditioning, Processor Interface and Serializer/DeSerializer circuits. Disk read data, disk clocks and reference clocks arrive via the Input Conditioning circuits, as do disk status lines. The disk control lines, disk write data and write clocks are sent via the Output Conditioning circuits. The Processor Interface generates microcode service requests, detects the overrun condition and passes data, status and commands along the X-Bus. Disk data is converted from 16 bit parallel words to a serial data stream and back in the Serializer/DeSerializer.

### 5.1.2 Constraints

*Cost*

The Dandelion is intended to be a relatively low cost workstation. To this end, the hardware it contains should be minimized. This leads to low manufacturing, testing and service costs. The guiding principle of the controller's design has been that only functions which occur too quickly for microcode to handle or require hardware buffering are implemented in the controller. For example, step pulses may be sent relatively slowly, so the step line is toggled by having the microcode send control words in which the step line is alternately set and reset.

Another result of the cost constraint is that one controller board should serve to control both the SA1000 and the SA4000 drives. It is able to support drives with 2 to 32 heads. The effort required to change the board from an SA1000 configuration to an SA4000 configuration is small. In fact, it is limited to unplugging a set of SA1000 cables and plugging in a set of SA4000 cables.

*Disk Format*

The disk is divided into cylinders. Each cylinder represents a distinct position of the read/write heads. Each cylinder is divided into tracks, one per read/write head. The SA1004 drive has 4 heads, the SA4008 has 8 and the SA4104 has 16. Each track is divided into sectors. There are 28 sectors per track on the SA4x00 type drives, 16 sectors per track on SA1000 type drives. Each sector is divided into three fields, Header, Label and Data. The Header field is used to specify the sector's physical position on the disk (cylinder, head and sector numbers), the Label specifies the page's position in the file system and the Data field holds the actual data. Each field is broken into 4 areas. A pattern of all zeros is followed by a synchronization word or address mark, the field's data and a word of CRC checksum. The length of the synchronization pattern is 7 words on both drive types. A synchronization word of all ones is used to define the first word boundry on the SA4000 drive. An address mark serves a similar purpose on the SA1000 drive. The Header field contains 2 words of data, the Label field 12 words and the Data field 256 words. The CRC checksum word following the data area of each field is used to implement an error detecting code.

The controller hardware does not preclude other disk formats. It is designed to read, write or verify an individual field of a sector. The length of each field, the number of fields per sector and the number of sectors per track is set by the microcode. There is a restriction on the number of sectors on SA4000 type disks. The SeekComplete signal on those disks is sent before the heads have really settled so the controller adds a delay of 29 sector pulses before passing it on. Thus SA4000 type disks should have no more than 28 sectors per track (the 29 sectors pulses is intended to delay at least 20 mS) or should be prepared to add some sort of extra delay in microcode.

One of the constraints on the design is that it must be possible to read, write or verify each field in every sector of a cylinder at the rate of one revolution per track. This means that in addition to the raw data rate constraint, the inter-field, inter-sector and inter-track setup required by the hardware must be minimized. A design which requires a great deal of setup between sectors or fields may not be acceptable. It should be possible to perform almost any combination of operations on the fields of a sector. An exception to this rule is that when a write is performed to one field, further fields of that sector must either also be written or are assumed to be lost. The microcode must also be capable of aborting operations on later fields based on the results of operations on earlier ones. For example, if the Header and Label fields of a sector are to be verified before the Data field is written, the Data write should be aborted if either the Header or Label verify operations fail.

The SA1000 drive does not contain a data separator, the SA4000 drive does contain one. The controller board sends and recieves MFM (Modified Frequency Modulation) encoded data to and from the SA1000 drive and NRZ (Non Return to Zero) data to and from the SA4000. The SA1000 data rate is 4.27 MBits/Sec (234 ns/bit). The SA4000 data rate is faster at 7.14 MBits/Sec (140 ns/bit). The SA1000 data rate is governed by a clock in the Dandelion, the SA4000 data rate is set by drive itself.

*Function Allocation*

The most complex operation on a field is verify. It requires that each bit be checked against a template from memory, a CRC checksum be maintained, a memory address updated and a word count decremented. Four pieces of information must be maintained, an address, a word count, the data to be verified and some sort of checksum. While it would be possible to combine the address and word count by requiring all field templates to begin (or end) on page or nibble boundries, this is not generally acceptable. The designer has been unable to find an encoding scheme which makes it possible to combine the data to be compared and the checksum. These seem to be the only remotely workable combinations. Hence all four quantities must be kept independently.

The four quantities must be divided between the two R registers in the processor and registers in the controller. The lack of U register speed precludes their use. One must spend an entire click to update one U register (read it, change it, then store it), yet the microcode is only allowed one click per word transferred. Due to the main memory addressing scheme, the address must reside in one of the R registers. This leaves the other R register available for either the checksum, data to be verified or the word count.

Were the R register to be used for the checksum, the hardware would contain the word count and the data to be verified. This scheme would have the advantage of substituting a simple counter for a more complex CRC chip. However, the microcode would have to both read the disk data to maintain the checksum and send memory data to the controller to be verified. This scheme has latency difficulties. The disk controller and processor use different, unsynchronized clocks. After sending a Service Request, the controller expects an interval of random, but bounded, length will pass before microcode reads or writes the proper buffer. The Service Request is sent so that the controller will have the buffer ready before the minimum service time and will not require it again before the maximum service time. As seen from the processor side, there is a window during which each Service Request must be served. If the service takes place too soon, the buffer may not be ready; if it is too late, the controller may have used the buffer again. In the case of the SA4x00 type disks, the service window is barely one cycle wide. The Service Request is sent so this is cycle 2 during Read operations and cycle 3 during Write and Verify operations. Sending and receiving data in one click would require 2 cycles, hence a 2 cycle service window. This is reason the microcode cannot maintain the checksum while the controller does data verification.

It would be possible to compute the checksum and maintain the word count in the controller while doing the address and verification in microcode. Unfortunately, the microcode would be messy and the status of an operation would be partially in microcode, partially in hardware. The controller as designed allocates the address and the word count to microcode and the data and checksum to hardware.

### 5.1.3 Microcode - Hardware Interface

The controller has been designed with the idea of minimizing the amount of hardware used. As much functionality as possible has been left in the microcode and software. This results in fairly simple controller hardware.

Many of the lines used to control the disk are set directly by microcode and are ignored by the controller. For example, the Step' and Direction lines controlling the position of the disk's read/write heads are merely bits in the control register that are relayed directly to the drive. The same is true for many of the status signals returned by the drive, they are read and interpreted only by the microcode or software.

The controller contains one word of buffering for write and verify operations and one word for read operations. As explained above, the Dandelion architecture allows the designer to calculate the minimum and maximum latencies between a service request and the processor's response to ensure an overrun never occurs in normal operation. If the disk microcode stops servicing the hardware, the overrun flag is set and write operations are disabled to restrict the amount of random data written on the disk.

This section will begin with an overview of the status, control and data registers then proceed with a detailed description of each.

*Control Register*

This 16 bit register receives its inputs from the X-Bus, sending them to both the disk drive and to the controller. It is reset by IOPReset'. The control bits are arranged so that when reset, the controller and disk are dormant. It is expected that IOPReset' will be held active while power to the machine is being turned on or off.

*Status and Test Registers*

Three types of 16 bit quantities may be read from the controller. One is data from the disk, the second is the status of the current disk operation, the third is a group of test points on the disk and display controllers. The first will be discussed below under Read Data Register. The second two are independently sent to the X bus. The operation status is composed of some lines from the drive itself (Track00, DriveNotReady, etc) and some from the controller (Verify Error, Overrun, etc). These are the normal lines read using the ←KStatus command to guide the execution of a disk operation. The test lines are read using the ←KTest command by diagnostic microcode or software to directly test the control and status lines leading to the disk.

Some of the Status signals should only be sampled on word boundaries. The CRC error flag, for instance, is only valid after the last bit of the CRC checksum has been seen. Sampling on word boundries also gives the microcode an entire word time, as opposed to one bit time, to freeze the final status flags of a data transfer. This sampling is done by the Word Status Register.

*Write Data Register*

Data is sent from the processor to the controller in 16 bit words. The words are buffered in the Write Data register before being loaded into a shift register. The buffer is automatically cleared before a transfer begins. It is loaded by the microcode in response to each service request during a transfer. By calculating the minimum and maximum latencies between request and service, one may be assured that the buffer is always loaded after the previous word has been used but before the current word is needed.

*Read Data Register*

Like the Write Data register, this is a single word of 16 bits. It is loaded from the controller's shift register each time a word boundry passes. Just before it is loaded, a service request is sent, asking the disk microcode to remove the word. As with the Write Data buffer, one may assure oneself that this will always happen after the buffer is loaded but before it is loaded again.

A wrap-around feature has been included in this controller allowing diagnostic microcode to verify that data may be written and read correctly. The method for using the feature depends on the disk being controlled. The SA4000 provides one clock used throughout the controller. The data sent out is intercepted just before the final drivers and inserted into the input data stream. It is then shifted back into the shift register. By having the microcode start a write operation, then perform reads instead of writes, one may verify that the data being written is correctly re-received. Note that the re-received data will be a rotated version of the data sent.

The SA1000 drive supplies no clock. The clock used to write the data is derived from the stable processor clock. If this clock were used for the entire controller, the controller's data separator would not be tested. The data separator is tested by allowing it to re-produce the NRZ data using a clock derived from the re-received MFM data stream. Because of jitter between the derived clock and the reference clock, we may not reliably route the re-produced NRZ data back to the shift register. Hence one may not expect to see the data sent in the ReadData register. The address mark recognizer section of the data separator does record the polarity of bit 14 of the address mark however. It appears on the Header tag bit in the KStatus register. One may test the controller by sending address marks and sampling the Header tag status bit after each one. Each address mark must be sent in its own field, that is, the TransferEnable bit should be reset between each one. The Header tag status bit should match bit 14 of the address mark just written.

*Service Request / Overrun Machine*

As seen above, the controller must be able to generate service request to its microcode and determine whether the requests have been answered. This is the task of the Service Request/Overrun machine. The timing of Service Requests is based on the BitCount within a word, the time within a field, the operation being performed and the data rate of the disk. Only two disk types are supported and the data rates of both are fixed.

During data transfer operations, it is crucial that the disk microcode keep pace with the hardware. If the microcode is early or late, especially during write operations, disk data may be destroyed. The Overrun section of this machine will set the Overrun signal whenever a buffer is needed by the controller before it has been serviced by the microcode. Thereafter, no data may be written (the disk's WriteEnable line is turned off) and the Service Request signal is set until the microcode finishes the operation and turns it off. The microcode should sample the status at the end of every operation, testing the Overrun signal. An unexpected consequence of turning off WriteEnable very early in the writing of a field is that the drive will often get a WriteFault error. If WriteFault and Overrun occur together during debugging, it is best to investigate the Overrun first.

Service requests may be used not only to synchronize the transmission of data but also to sense status conditions. For example, it would be wasteful to burn 1/5 of the processor waiting 20 ms for an IndexFound signal. The same holds true for a SeekComplete. These and other signals may be used to generate service requests directly. The microcode may then yield its click to the emulator while waiting. The signals are chosen using the Operation field of the Control register.

### 6.1.4 Detailed Register Description

**KCtl Register**

| Head Select | | | | | Drive Select | Fault Clear | Reduce IW | Step | Direct In | Firmware Enable | Transfer Enable | Write CRC | Wakeup Control | | Write Enable |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 8 | 4 | 2 | 1 | | | | | | | | | 0 | 1 | |

**KStatus Register**

| Head Select' | | | | | Seek Complete | Track 00 | Firmware Enable | Index Found | Sector Found/ Header Tag | SA1000 / SA4000' | Drive Not Ready | Write Fault | Overrun | CRC Error | Verify Error |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 16 | 8 | 4 | 2 | 1 | | | | | | | | | | | |

**KTest Register**

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Disk Read Clk | Disk Read Data | Disk Output Clk | Disk Write Data | Seek Complete' | Direction In' | BHoriz | Reduce IW' | TTL-Video | Sector' | Drive Select' | BVert' | TTL-Video' | Step' | Read Gate' | Write Gate' |

*Control Register*

The register is loaded by the processor when a "KCtl ← xx" type instruction is executed in microcode. This may also be done as part of a Mesa "Output" instruction. The command word is divided into two parts intended for the drive and the controller. The meaning of the bits in the Drive Control field are explained fully in the appropriate Shugart manuals. They are listed below with a brief description. A list of Operation bit meanings is given below. Use of all bits in the control word will be given in the section on microcode usage. Control lines required by the drive but not listed below are the responsibility of the controller, not the microcode.

*Drive Control*

HeadSelect1 - HeadSelect16: These 5 bits are used to select one of the read/write heads. They are not latched by the drive; all commands must contain them. For example, when one writes a field by sending a write command and a write CRC in succession, the proper head select bits must be present in both commands. To ensure the drive's setup times are met, a command word containing the proper HeadSelect lines should be sent at least 20 uS before one containing the HeadSelect lines and the operation to be performed.

DriveSelect: The DriveSelect bit has been included even though only one drive may be connected at a time. This is because releasing DriveSelect has useful side effects. The SA1000 type drives lack a FaultClear input, Write Faults are cleared by de-activating the DriveSelect signal. The SA4000 drive has a feature enabling it to cut the power to its stepper motors when not selected. This can result in a substantial power savings. The power may be cut by software when the drive has been idle for some nominal interval. When re-selected, one must wait 20 ms before using the drive. This time interval may be sensed using the SeekComplete signal which is automatically cleared when the drive is de-selected.

FaultClear: The FaultClear bit is only active when an SA4000 drive is connected to the controller. Write Faults on the SA1000 are cleared by turning off DriveSelect as explained above. An SA4000 WriteFault is cleared by activating both DriveSelect and FaultClear then de-activating FaultClear. If the WriteFault remains, the drive is probably broken.

ReduceIW: This bit is only significant when writing on the SA1000 type drives. These drives require the write current to be reduced on cylinders 128 through 255. This bit should be set by the microcode when writing on these cylinders. During read and verify operations on SA1000 disks and during all operations on SA4000 disks, this bit is ignored.

Step, DirectionIn: The position of the read/write heads on both the SA1000 and SA4000 disks is controller by a stepper motor. The heads will move one cylinder for each complete pulse of the Step line. A pulse is sent by sending two control words. In the first, the Step line is set, the the second, it is reset. The direction of movement is governed by the DirectionIn line. When DirectionIn is set during a series of step commands, the heads will move towards the higher numbered cylinders in the middle of the disk. To satisfy the disk's setup times, a command word containing the proper DirectionIn bit should be sent at least one cycle before the first Step pulse It is also recommended that microcode should wait for SeekComplete before beginning any stepping operation.

Both the SA1000 and SA4000 drives have two stepping modes, *normal* and *buffered.* In normal mode, a 1 microsecond pulse is sent every millisecond. The heads move every time a pulse is sent. This mode is used during a recalibration so the Track00 signal may be sensed. In buffered step mode, a series of Step pulses with a mimimum period of 2 uS and minimum width of 1 uS is accumulated in the drive. Once 350 uS have elapsed without pulses, the drive moves the heads. If more step pulses arrive once the heads are in motion, their final position is undefined. Thus, buffered step mode should be used by microcode, not software, so pulse timing may be rigidly controlled.

*Operation Control*

FirmwareEnable: The FirmwareEnable bit is set whenever the disk microcode is running. In addition to acting as a status bit for higher level software, it is used to generate a service request for overhead operations.

TransferEnable: TransferEnable is set whenever a data transfer is taking place. A data transfer encompasses exactly one field of a sector. Writing or reading the data of a sector will generally require three data transfers (verify header, verify label and write or read data). The transfer operation includes the recognition or writing of the VFO synchronization pattern, sync word or address mark and the CRC checksum as well as transferring the data. When TransferEnable is reset, all the state machines used to transfer or recognize data are reset.

WriteCRC: The WriteCRC bit causes the CRC checksum to be written at the end of a field. The BTransferEnable and BWriteEnable lines must also be true for this to be accomplished. Proper use of this bit in writing a field will be explained in the section on microcode usage.

WakeupControl.(0,1): These bits together with TransferEnable are used to specify the condition generating the microcode service request. The conditions allowed are:

| TransferEnable | WakeupControl.(0,1) | Condition |
|---|---|---|
| 0 | 00 | FirmwareEnable |
| 0 | 01 | SeekComplete |
| 0 | 10 | SectorFound (valid only on SA4000) |
| 0 | 11 | IndexFound |
| 1 | 00 | Word Ready from Read operation |
| 1 | 01 | Word Needed for Write or Verify operation |
| 1 | 10 | <no wakeup> |
| 1 | 11 | <no wakeup> |

WriteEnable: The WriteEnable bit controls the write amplifier on the drive. In addition, it is used by the controller to decide when a write operation is taking place. The WriteGate to the drive is enabled onl:. when WriteEnable and TransferEnable are true and Overrun is false.

*Status Register*

The status register is read using the "← ~KStatus" clause in microcode. All status bits are inverted on the X bus because use of the comparable non-inverting drivers was forbidden when the board was designed. The bits will be described as though the inverstion were not present. It is expected that when the user either reads the bits into the CP or uses them as X bus branch conditions, the inversion will be taken into account.

There are two main purposes for status bits: diagnostic and operational. Some bits are included so diagnostic code may attempt to isolate a fault to either the drive or the controller. Operational bits are needed for normal operations. Diagnostic bits are generally those sent to the drive and also read back by the controller.

HeadSelect1'-HeadSelect16': These are diagnostic lines. They should give an inverted version of the head select lines in the control register. They are used to check that the proper head is actually being selected.

SeekComplete: This signal indicated the read/write heads are ready for use. It is set when the drive is ready , it is selected and the heads are not in motion. Head motion can be divided into two parts. First the stepper motor guides the heads to a new cylinder. Second, after they arrive, they vibrate for a few milliseconds. The first interval is called the seek time, the second is called the head settling time. The head settling time for both the SA1000 and SA4000 drives is about 20 mS. This can be much larger than the seek time for short seeks. The SA1000 drives supply their own head settling delay so their SeekComplete really means the heads have stopped. The SeekComplete signal from the SA4000 drives means only that the stepper motor has arrived, the 20 mS must be added externally. This is done in the controller hardware (by counting 29 sector pulses). Thus as far as the user is concerned, SeekComplete always means the heads have moved and settled. This counting of 29 sector pulses when an SA4000 type disk is attached is the controller hardware's only assumption about the number of sectors on a track. If the number of sectors on the SA4000 or SA4100 type of disk is increased, some sort if external delay will be needed.

Track00: This status line becomes active whenever the disk's read/write heads are over cylinder 0. It is probably only valid when SeekComplete is asserted. It is used by microcode and software to recalibrate the heads. Note there are a few cylinders beyond cylinder, just as there a few beyond the maximum cylinder. A recalibration algorithm should take this into account. In particular, simply stepping out from the current position is not guaranteed to lead to cylinder 0.

FirmwareEnable: This bit is used to indicate the microcode is active. It directly reflects the FirmwareEnable control bit. It is mostly by convention that this bit is set while the microcode is active; it would be possible to turn it off when the service requests are derived from another source. The convention is useful when synchronizing software with disk microcode.

IndexFound: The index pulse from the drive occurs once per revolution and lasts between 1 and 10 uS. It is used to mark a specific position on the disk, usually the beginning of sector 0 on all tracks. The IndexFound bit is a latched version of the drive's index pulse. The latch is cleared using the "ClrKFlags" clause in microcode. the IndexFound flag may also be used to generate service requests.

SectorFound/HeaderTag': The meaning of this bit depends on the drive connected. When an SA4000 or SA4100 type drive is being controlled, a latched version of the drive's sector pulse is available here. The latch may be cleared using the "ClrKFlags" clause in microcode. The SectorFound flag is commonly used to generate a service request so the microcode may detect the start of a sector.

The SA1000 drives have no sector pulse. In order to find the beginning of a sector, the microcode commands the controller to verify each field as it arrives. The address mark used for header fields differs from that used for label and data fields. The header address mark has a 0 in bit 14, the address mark used for label and data fields has a 1 there. After reading a field, the value of bit 14 is displayed on this status bit when an SA1000 type drive is connected. Using it, microcode may verify that the field seen was indeed a header field in addition to having the correct data and CRC. The polarity was chosen so this bit could be used as an error indicator when looking for the correct header (1 => not a header). Use of this bit is explained further in the section on microcode usage.

SA1000/SA4000': This bit is set when an SA1000 type drive is attached to the controller. It is reset when an SA4000 or SA4100 type drive is attached. The two classes of drives require completely different cables. This bit is connected to a line that is grounded in the SA4000 and SA4100 cables and is pulled up in the SA1000 cable. Note the controller gives no hint about the number of heads per track or other drive variables. Determination of other disk parameters is initially done using experimentation. It is expected that configuration information will be recorded on the disk for normal use.

DriveNotReady: The drive's Ready line is inverted and sent here. The Ready line indicates the drive has power, is warmed up, is selected and is generally ready for use. The line is inverted here so it may be used as an error flag (not ready => error). Software and/or microcode should wait for this line to become active after power on before initiating any operations. In addition, it should be checked after each operation to ensure the disk hasn't broken.

WriteFault: Each type of drive can detect some internal error conditions. On the SA4000 and SA4100 drives these include WriteGate without write current in the selected head or vice versa, multiple heads selected, WriteGate active when Ready inactive and WriteGate and ReadGate active simultaneously. The SA1000 set is less comprehensive including only write current without WriteGate and multiple heads selected. When a WriteFault occurs (not necessarily only during write operations), it is latched in the drive. This status bit is a buffered version of the drive's latch. In general, service personnel prefer that software not automatically clear this line when an error is detected. This gives them some chance to see which condition caused the problem. This line should be cleared at the beginning of an operation. On the SA4000 and SA4100 type drives, it is cleared by asserting both DriveSelect and FaultClear in a command word, then sending a command with only DriveSelect. The SA1000's WriteFault is cleared by de-selecting the drive (writing a command word with DriveSelect=0) for at least 500 ns. If, because of some hardware condition, an Overrun occurs, the controller will immediately clear WriteEnable. This sometimes causes a WriteFault. The WriteFault will then persist through subsequent operations until cleared though the Overrun may vanish with the next operation. When having a WriteFault problem, it is best to see if it is caused by an Overrun.

Overrun: It is important to minimize damage to the disk if the processor runs wild and spuriously enables a write operation. If the controller's service requests for data are not answered, the Overrun bit will be set and WriteEnable turned off. If this happens early in the field being written, the drive will sometimes detect a WriteFault as explained above. Presence of this bit means either the controller or the drive is broken or that the jumpers on the drive are not correct. Disk microcode should check this status bit after every operation.

CRCError: The controller contains a 16 bit cyclic redundancy code (CRC) generator and checker. The WriteCRC control bit is used to append the generator's contents to each field written. After each field read or verified, this bit should be checked by microcode to ensure the field had the correct CRC. Like all the error bits, this one is set only when there has been an error. The CRCError bit is valid only just after the checksum word has been processed by the checker. There is a one word window for the microcode to stop the transfer, freezing the status. This is discussed in the microcode usage section. The CRCError bit is reset using the microcode's "ClrKFlags" clause before each operation.

VerifyError: The verify operation compares bits on the disk with a template in memory. It is used mainly to find headers and check labels. The verify operation is implemented by writing the template to the controller while it is reading the disk data. If one or more of the bits differs, the VerifyError bit is set. It is reset using the "ClrKFlags" clause in microcode.

*Test Register*

This register is used by diagnostic code to read signals on the cables leading from the HSIO board. In this way, the diagnostic code may decide whether a particular fault lies in the HSIO card or in the attached peripheral. The register is read using the "← ~KTest" clause in microcode.

DiskReadClk: This signal is used only when controlling SA4000 and SA4100 drives. It allows the processor direct access to the disk's 140 ns clock. Since this clock is not synchronized with the processor clock, any given sample of it may return either a 1 or a 0. Diagnostic code should read it repeatedly to see if it changes state. The online diagnostics require detection only of stuck-at faults.

DiskReadData: This is the data directly from the disk. The SA4000 and SA4100 disks return NRZ (Non Return to Zero) data; the SA1000 returns 50 ns MFM (Modified Frequency Modulation) pulses. Again, the diagnostic microcode only hopes to catch this line changing state with repeated samples.

DiskOutputClk: The SA4000 drives use this clock to sample the controller's write data. The SA1000 drives use it as a time base for seek operations. It is another signal diagnostic code can sample.

DiskWriteData: This can actually be controlled by the diagnostic code. By writing words of either all 0's or all 1's this line can be set to 0 or 1.

SeekComplete': This is a version of SeekComplete directly from the cable. The controller delays an multiplexes this line before sending it to the KStatus port (see above).

DirectionIn': This is one of the signals sent to the drive that is re-received from the cable. It is used to test the control register and the drivers.

BHoriz: Display signals are also available in this register. This is the horizontal sync signal sent to the monitor. It is active for ~7 uS every 28.8 uS. As usual, it may be sampled by diagnostic code.

ReduceIW': The version of the ReduceIW signal (see Control Register above) on the interface cable to the SA1000 disk is available here. It may be directly controlled by the diagnostic code.

TTLVideo: This is the positive true version of the video signal sent to the monitor. Since this has a minimum pulse width of 19.59 ns, it probably shouldn't be sampled arbitrarily. One may set the border pattern to all zeros or all ones then have the display controller send all border pattern. In this way, the video signal will usually take on the known value. About 1/4 of the time (7/28.8) it will always be set to zero for horizontal retrace.

Sector': The SA4000 and SA4100 drives send a pulse at the beginning of each sector. The pulses are 1.1 uS in duration and occur roughly every 710 uS. By diligent sampling, diagnostic code may see this line change state.

DriveSelect': Like ReduceIW and DirectionIn, this line is available directly from the interface cable to test the control register and drivers.

BVert': This is the display's vertical sync signal. It is active LO. It may be set or reset directly in the DCtl register.

TTLVideo': This is the negative true version of the display's video signal. It was included in addition to TTLVideo so that both halves of the differential driver might be tested.

Step': This is another cable signal available to test the control register and drivers.

ReadGate': Only the SA4000 and SA4100 drives use ReadGate'. It is set by the controller during all read and verify transfers. Diagnostic code may start a read or verify operation then sample this signal.

WriteGate': This is the version of write enable sent to the drive. If data is not supplied by microcode after turning on WriteEnable, this signal should remain active LO for one word time, then go inactive. If the controller is serviced by either writing or reading data or writing a control word each time a service request is sent, this signal should remain active.

*ReadData Register*

Data read from the disk resided in one 16 bit buffer. It is read by microcode using the "← KIData" clause. When a tranfer is in progress, one word must be read each time the controller requests service. Since the controller will request service in consecutive disk clicks, the disk microcode may use only 1 click to transfer the data. In addition, when SA4000 or SA4100 drives are connected, the data in the ReadData register is only valid in cycle 2. The timing is so close that it could only be valid in one of the cycles. Cycle 2 was chosen so the data could be written to memory.

*WriteData Register*

Data to be written or verified is stored in this register using the "KOData ←" clause. The register holds a single 16 bit word and must be filled each time the controller sends a service request. As with all data tranfers, the microcode has only 1 click to read memory, increment the memory address, decrement the word count and decide if the end of the transfer has been reached. When the SA4000 or SA4100 is connected, the "KOData ←" statement should only be executed in cycle 3. Generally data is written to the disk from memory and memory data is available in cycle 3. Note that one may substitute a read from KIData or a write to KCtl for the write to KOData in cycle 3. The read from KIData might be used during a wrap-around test and the write to KCtl is always used to send the WriteCRC command at the end of a field.

### 5.1.5 Microcode Usage

The most useful document for one starting to write microcode for the disk is existing disk code. The Pilot disk microcode is stored on [Idun]<WDlion>DiskDlionA.mc and [Idun]<WDlion>DiskDlionB.mc. This code is amply commented. It is broken into two files only because it is too large for Bravo to handle. The disk microcode also makes use of two definitions files stored on [Idun]<WDlion>, DiskDlion.df and Dandelion.df.

The beginning microcoder should read the *Dandelion Microcode Reference* to become acquainted with a great many interesting and obscure Dandelion facts. This discussion will assume a reasonable facility with Dandelion microcode.

The DiskDlion microcode was written to provide adequate performance while taking as few microinstructions as possible. It was decided that the SA4000 and SA4100 type disks would have 28 sectors per track (same as the Dolphin) and the SA1000 disks would have 16. Each sector has the three standard fields, Header, Label and Data. The Header field has 2 words and the Data field has 256. The Label field was originally 8 words long but finally grew to 12 words. The microcode had to be written so operations could be carried out on runs of consecutive sectors crossing track boundries. It was hoped that the microcode could fit in 128 control store words but 256 words was acceptable. The current code fits in 236 words.

The requirement for processing consecutive sectors puts severe timing constraints on the code. It limits the amount of inter-field, inter-sector and inter-track overhead allowed. The original code took a compact command representation, parsed it and generated the necessary control words. This code not only did not meet the timing requirements, it was also much too large. The second version of the code required the user to specify series of disk operations as small program of simple instructions in the IOCB. This took advantage of the fact that the same task might be needed many times in a run of pages, but code to implement that task would only occur once. An instruction to the disk microcode might be Increment and Skip If Zero or TranferField. This approach also allowed the user great flexibility at the head level; diagnostics could use the standard disk microcode and the disk format could be changed without changing the microcode. The resulting code took only 128 words but did not satisfy the performance requirements. The final version is based on the second one, with a "Transfer Run of Pages" command and a "Load Parameters" added. The parameters specify the operation to be performed on each field, the length and location of each field in memory and the error mask to be used.

This document will assume that the reader wishes to know how to use the controller hardware, not how to load parameters or determine a disk format. The controller hardware is designed to assist with the transfer of a single field within a sector. It has no knowledge of the number of cylinders, heads or sectors on the disk (except as noted in the explanation of the SeekComplete status bit). The DiskDlion microcode has a subroutine called TransferField that accepts as input the field's operation, length and location in memory. It is used for all read, write and verify operations. The rest of this chapter will be concerned with the TransferField subroutine.

Although the same routine is used to perform all operations on all fields with both the SA1000 and SA4000 type disks, the operations will be explained separately. The reader may use the TransferField routine as an example of how they may be combined. General principles which apply to all operations will be explained first.

The controller hardware contains no information about the length of the field it is processing. When writing, it writes the data given until it receives a disabling control word instead of a data word. The same is true of reading and verifying. The length of each field is determined by microcode.

Timing, especially for the SA4000 and SA4100 disks, is critical. Those drives contain data separators which should only be enabled when the heads are over synchronization gaps containing all zeros. The microcode calculates the position of the read/write heads by dead reckoning. It can sense the index and sector pulses from the drive and can know the number of microinstructions executed since the pulse. As a result of this, the number of microinstructions executed between calls to TranferField cannot be a function of the operation being executed. In fact, the number clicks executed between the end of a field and the beginning of the next field must be independent of operation. Of course, it is reasonable for the number of instructions to be a function of the field. For example, the number of clicks executed between the end of the Label field and the beginning of the Data field should not depend on the operations performed on either the Label or Data fields though it may differ from the number of clicks between the Header and Label fields.

*Writing on the SA4000 and SA4100*

Each field on an SA4000 or SA4100 has 4 parts. These are:

| Name | length | value |
|---|---|---|
| Synchronization gap | 7 words | 0000 |
| Synchronization word | 1 word | FFFF'X |
| Data | 2 words | Header Field, or |
| | 12 words | Label Field, or |
| | 256 words | Data Field |
| CRC checksum word | 1 word | calculated CRC checksum |

The data separator in the drive needs at least 8 uS (~4 word times) to acquire the data stream. The microcode can only know the position of the read heads to an accuracy of plus or minus one click. Delaying one click after the nominal beginning of the synchronization pattern gives a real delay of from zero to two clicks. To ensure at least a 1 click delay, the code must wait for two clicks. This means the real delay could be three clicks so the synchronization gap is 3+4 or 7 words (where the time between clicks ~= 1 word time).

Code for writing on the SA4000 or SA4100 disk should proceed as specifed below. Writing the Header field is used as the example, differences between the Header and other field will be explaned later.

1. Prepare the parameters used for writing the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=3. This causes the next click to take place just after the next sector mark has been found. If the field is not the Header field, this step must be skipped. The command word would be 0426'x + 800'x*HeadNumber. Note that if the Header for Sector 0 is desired, one must have the microcode find the index mark and count 27 sectors marks before starting this step. Having done this, the next sector mark must belong to sector 0. One could simply find the index mark and start writing if one were willing to make the operation of writing the first sector different than that of writing the rest of them.

3. After finding the sector mark, $N_h$ clicks may be used for further field set up. The minimum time between when the Find Sector control word is sent and the write is started should be 10 uS (5 clicks) to give the drive time to select the heads properly.

4. The control word is sent starting the write. This control word contains the number of the head to be used, DriveSelect, FirmwareEnable, TransferEnable, WakeupControl=1 and WriteEnable. It is 0433'x + 800'x*HeadNumber.

5. The controller will write the first two words of synchronization pattern automatically. The microcode should provide 5 more words of 0 to KOData; all in cycle 3.

6. The microcode supplies one word of FFFF'x to the controller in cycle 3. This is the synchronization word used by the controller hardware to find the word boundries in the serial bit stream when the field is read.

7. Microcode should execute a loop which transfers one data word per click to the KOData port. All transfers should take place in cycle 3. See the DiskDlionB.mc file for an example of such a loop.

8. A control word should be sent causing the CRC checksum to be appended to the field. The control word is identical to the one used to start the write operation with the addition of the WriteCRC bit. It is 043B'x + 800'x*HeadNumber.

9. The same control word should be sent again. The controller is pipelined to the extent that one word is being sent to the disk while the next word is received from the processor. Thus the controller cannot be stopped now as this would cause the CRC to be chopped off. Some word must be sent to the controller to prevent an Overrun condition. Sending the same control word is as easy as anything else.

10. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

11. The DriveNotReady, WriteFault and Overrun status bits should be checked. If there was an error, the operation should probably be aborted. The disk task's double bit memory error flag in the MStatus register should also be checked. The errors recorded while the disk task is reading memory do not cause a trap but they are recorded.

The process of writing fields other that the Header is quite similar. Step 2 may be eliminated since the sector has been found and the head number established. The number of clicks used for setup should be minimized, there is no minimum value. One should take care that the number of clicks executed between fields is independent of the operation performed on the fields.

*Writing on the SA1000*

This is intentionally quite similar to writing on the SA4000. The differences are that the SA1000 has no sector marks, it uses an address mark instead of a synchronization word and one is required to wait for 2 clicks to elapse after starting the CRC write intead of 1.

Because of the fact that there are no sector marks on the SA1000, the position of a Header directly determines the position of a sector. For this reason, individual sectors cannot be formatted; one must format an entire track in one run. The microcode finds the index mark and writes sectors as fast as possible. Once a track is formatted, it is, of course, possible to write the Label and Data fields of its sectors individually. Shown below is the sequence used to write the Header of Sector 0 on a track. When writing other Headers in the formatting run, the step used to find the index mark is eliminated.

Because address marks are used to define the beginning of fields, all previous address marks on a track must be erased before formatting. This is done in the Pilot system by having the head tell the microcode to write a very long sector (the length of a track). Any legal MFM pattern is adequate.

The format for a field on the SA1000 is:

| Name | length | value |
|---|---|---|
| Synchronization gap | 7 words | 0000 |
| Address Mark | 1 word | A141'x - Header Field |
| | | A143'x - Label or Data Field |
| Data | 2 words | Header Field, or |
| | 12 words | Label Field, or |
| | 256 words | Data Field |
| CRC checksum word | 1 word | calculated CRC checksum |

1. Prepare the parameters used for writing the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=2. This causes the next click to take place just after the index mark has been found. If the field is not the Header field of Sector 0, this step must be skipped. The command word is 0424'x + 800'x*HeadNumber.

3. After finding the sector mark, $N_h$ clicks may be used for further field set up. The minimum time between when the Find Sector control word is sent and the write is started should be 10 uS (5 clicks) to give the drive time to select the heads properly.

4. The control word is written starting the write. This control word contains the number of the head to be used, DriveSelect, FirmwareEnable, TransferEnable, WakeupControl=1 and WriteEnable. It is 0433'x + 800'x*HeadNumber.

5. The controller will write the first two words of synchronization pattern automatically. The microcode should provide 5 more words of 0 to KOData; all in cycle 3.

6. The microcode writes the data word A141'x to the controller in cycle 3. This triggers the writing of the Header's address mark. The real address mark is an illegal MFM string. It can be distinguished from ordinary data and is used by the controller hardware to find the start of a field.

7. A loop should be executed which transfers one data word per click to the KOData port. All transfers should take place in cycle 3. See the DiskDlionB.mc file for an example of such a loop.

8. A control word should be sent causing the CRC checksum to be appended to the field. The control word is identical to the one used to start the write operation with the addition of the WriteCRC bit. It is 043B'x + 800'x*HeadNumber.

9. The same control word should be sent two more times. The controller is pipelined to the extent that one word is being sent to the disk while the next word is received from the processor. Thus the controller cannot be stopped now as this would cause the CRC to be chopped off. It additionally appears that if a short tail is not written after the CRC, it cannot be read correctly. This is why two extra cycles are taken. A word must be sent to the controller in each cycle to prevent an Overrun condition. Sending the same control word is as easy as anything else.

10. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

11. The DriveNotReady, WriteFault and Overrun status bits should be checked. If there was an error, the operation should probably be aborted. The disk task's double bit memory error flag in the MStatus register should also be checked. The errors recorded while the disk task is reading memory do not cause a trap but they are recorded.

Writing other fields in the same sector differs only in that Step 2 can be eliminated and the address mark written in Step 6 is A143'x. This allows the microcode to distinguish between Headers and other fields when the fields are read. It is still important that the number of clicks executed between fields is independent of the operations performed on those fields.

*Reading Data from the SA4000 and SA4100*

The main differences between reading and writing are that one must find the synchronization gap instead of creating it and read data instead of writing it. The operations for reading a Header will be shown. The differences involved in reading other fields will be explained later.

1. Prepare the parameters used for reading the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=3. This causes the next click to take place just after the next sector mark has been found. If the field is not the Header field, this step must be skipped. The command word would be 0426'x + 800'x*HeadNumber.

3. After finding the sector mark, $N_h$+2 clicks must be used for further field set up. Note that this is same $N_h$ used when writing a field. The extra two clicks are used to guarantee that the read heads are inside the synchronization gap when they are enabled. The minimum time between when the Find Sector control word is sent and the read is started should be 10 uS (5 clicks) to give the drive time to select the heads properly.

4. The control word is sent starting the read. This control word contains the number of the head to be used, DriveSelect, FirmwareEnable. TransferEnable and WakeupControl=0. It is 0430'x + 800'x*HeadNumber.

5. The controller will find the synchronization word automatically. The first service request announces that the synchronization word is in the KIData buffer. This should be read. It may then either be saved or discarded. It is provided for diagnostic purposes.

6. The microcode should execute a loop which transfers one data word per click from the KIData port. All transfers should take place in cycle 2. See the DiskDlionB.mc file for an example of such a loop. Note that if the buffer address calculated in cycle 1 crosses a page boundry, the memory write operation will be aborted. Data pages in the Pilot world are always page aligned so the last click executed when transferring data must not increment the memory address. See the *Dandelion Microcode Reference* for further details.

7. An extra word or command must be read or written. This gives the controller time to process the CRC checksum at the end of the field. If the extra transfer is left out, the controller will detect an Overrun. If a command is sent, it should be the original read command.

8. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

9. The DriveNotReady, WriteFault, Overrun and CRCError status bits should be checked. If there was an error, the operation should probably be aborted.

As usual, Step 2 is eliminated and the time is Step 3 is decreased when reading other fields. Note that the delay in the read version of Step 3 is always 2 clicks longer than in the write version of the corresponding field. For example, if there are 4 clicks between the times a Header operation is stopped and the write of a Label is started, there should be 6 clicks between the times a Header operation is stopped and a Label read is started. The extra two clicks are provided by TransferField in this code, so the time between calls to TransferField must be independent of the operation.

*Reading from an SA1000*

Headers are very seldom read. They are written only when the disk is being formatted. Normally they are verified. To read Header n of a track, one usually finds the index mark and reads the next n+1 Headers; all into the same buffer. This complication is not germane to this discussion. The process of reading Sector zero's Header will be shown, the normal modifications required to read other Headers and other fields will be pointed out.

1. Prepare the parameters used for reading the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=2. This causes the next click to take place just after the index mark has been found. If the field is not Sector zero's Header field, this step must be skipped. The command word would be 0426'x + 800'x*HeadNumber.

3. After finding the index mark, $N_h+2$ clicks must be used for further field set up. Note that this is same $N_h$ used when writing a field. The data separator used for the SA1000 is on the controller board and has no requirement about being turned on over the synchronization gap. The reading process should, however, begin promptly so Sector zero's Header will be found first.

4. The control word is sent starting the read. This control word contains the number of the head to be used, DriveSelect, FirmwareEnable, TransferEnable and WakeupControl=0. It is 0430'x + 800'x*HeadNumber.

5. The controller will find the address mark automatically. The first service request announces that the address mark is in the KIData buffer. This should be read. It may then either be saved or discarded. It is provided for diagnostic purposes.

6. The microcode should execute a loop which transfers one data word per click from the KIData port. All transfers should take place in cycle 2. See the DiskDlionB.mc file for an example of such a loop. Note that if the buffer address calculated in cycle 1 crosses a page boundry, the memory write operation will be aborted. Data pages in the Pilot world are always page aligned so the last click executed when transferring data must not increment the memory address. See the *Dandelion Microcode Reference* for further details.

7. An extra word or command must be read or written. This gives the controller time to process the CRC checksum at the end of the field. If the extra transfer is left out, the controller will detect an Overrun. Note if a command is sent, it should be the original read command.

8. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

9. The HeaderTag, DriveNotReady, WriteFault, Overrun and CRCError status bits should be checked. If there was an error, the operation should probably be aborted. Note the HeaderTag status bit here is set if the field read was not a Header. It is available on the status bit that would have been used for SectorFound if an SA4000 had been connected.

As usual, Step 2 is deleted when not looking for Sector zero's Header field. Label and Data fields are generally read by verifying all fields encountered until a match for the desired sector's Header field is found, then reading the next fields in order. There is no requirement that the SA1000 data separator be turned on over a field of zeros but it should be enabled at least 4 word times before the address mark of the field to be read or verified.

*Verifying Data on the SA4000 and SA4100*

A verify operation combines the read and write operations. Data is read both from the disk and from memory and compared on the controller board. As far as the microcode is concerned, a verify starts like a read with the data separator enabled to find the field. Once the field is found, a verify is like a write in that data is sent to the controller.

The procedure for verifying a Header will be shown. As explained above, this is by far the most common operation performed on Headers. DiskDlion microcode uses the verify operation to locate the Header for the proper sector. Microcode could easily be written that woke up on every SectorFound pulse and maintained a current sector number. This was not done for simplicity.

1. Prepare the parameters used for verifying the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=3. This causes the next click to take place just after the next sector mark has been found. If the field is not a Header field, this step must be skipped. The command word would be 0426'x + 800'x*HeadNumber.

3. After finding the sector mark, $N_h+2$ clicks must be used for further field set up. Note that this is same $N_h$ used when writing a field. The extra two clicks are used to guarantee that the read heads are inside the synchronization gap when they are enabled. The minimum time between when the Find Sector control word is sent and the read is started should be 10 uS (5 clicks) to give the drive time to select the heads properly.

4. The control word is sent to start the verify. This control word contains the number of the head to be used. DriveSelect, FirmwareEnable, TransferEnable and WakeupControl=1. It is 0432'x + 800'x*HeadNumber. In the same click as the control word is written, but after it is written, the first memory template word must be sent to the controller. It must be sent in the same click because the next service request will not be generated until the controller has started comparing the first memory and disk words. It must be sent after the control word because the WriteData buffer is held cleared until then. All words sent before the verify operation is enabled are lost.

5. The controller will find the synchronization word automatically. The first service request announces that the second template word is needed for comparison. This is the beginning of the verify loop.

6. The microcode should execute a loop which transfers one template word per click to the KOData port. All transfers should take place in cycle 3. See the DiskDlionB.mc file for an example of such a loop.

7. Two extra words or commands must be written. This gives the controller time to process the CRC checksum at the end of the field. If the extra transfers are left out, the controller will detect an Overrun. If commands are sent, they should equal the original verify command.

8. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

9. The DriveNotReady, WriteFault, Overrun, CRCError and VerifyError status bits should be checked. If there was an error, the operation should probably be aborted. Note that if the verify operation is being used to find the proper Header, errors in the DriveNotReady, WriteFault and Overrun are fatal whereas CRC and Verify errors only indicate the wrong Header was found. One should try every Header on the track before giving up.

The usual remarks about eliminating Step 2 and shortening the delay in step 3 apply when verifying Label or Data fields. A Verify or CRC error found when verifying a Label or Data field is always fatal. Pilot normally issues operations of the form: verify Header, verify Label, read or write Data.

*Verifying Data on an SA1000*

Verifying Headers is also the principle method used to find sectors on the SA1000 disks. Since the SA1000 has no sector marks however, one cannot guarantee where the reading process will begin unless the index mark is sensed. For this reason, address marks are used. These are MFM patterns that meet the data separator timing requirements but cannot occur in normal data. When enabled, the controller waits until an address mark is found before starting the verify operation. It is also quite likely that the first address mark found will not belong to a Header field. For this reason, Header address marks have a 0 in bit 14 while Label and Data address marks have a 1 there. This bit is shown on the Header Tag status bit. It may be used as an error indicator when reading or verifying Header fields.

For the sake of consistency, the process of verifying Sector zero's Header will be shown, though one seldom begins a verify operation by finding the index mark on the SA1000.

1. Prepare the parameters used for verifying the Header field.

2. Send a command to the controller containing the number of the head to be used, DriveSelect, FirmwareEnable and WakeupControl=2. This causes the next click to take place just after the index mark has been found. If the field to be verified is not Sector zero's Header field, this step must be skipped. It is normally skipped anyway. The command word would be 0426'x + 800'x*HeadNumber.

3. After finding the index mark, $N_h+2$ clicks may be used for further field set up. Note that this is same $N_h$ used when writing a field. The data separator used for the SA1000 is on the controller board and has no requirement about being turned on over the synchronization gap. The reading process should however begin promptly so Sector zero's Header will be found first if this is desired.

4. The control word is written starting the verify. This control word contains the number of the head to be used, DriveSelect, FirmwareEnable, TransferEnable and WakeupControl=1. It is 0432'x + 800'x*HeadNumber. In the same click as the control word is written, but after it is written, the first memory template word must be sent to the controller. It must be sent in the same click because the next service request will not be generated until the controller has started comparing the first memory and disk words. It must be sent after the control word because the WriteData buffer is held cleared until then. All words sent before the verify operation is enabled are lost.

5. The controller will find the address mark automatically. The first service request announces that the second template word is needed for comparison. This is the beginning of the verify loop.

6. The microcode should execute a loop which transfers one template word per click to the KOData port. All transfers should take place in cycle 3. See the DiskDlionB.mc file for an example of such a loop.

7. Two extra words or commands must be written. This gives the controller time to process the CRC checksum at the end of the field. If the extra transfers are left out, the controller will detect an Overrun. Note if a command is sent, it should be the original verify command.

8. A command should be sent disabling the Controller. It should contain the number of the head to be used in the next field, DriveSelect and FirmwareEnable. It is 0420'x+800'x*HeadNumber.

9. The HeaderTag, DriveNotReady, WriteFault, Overrun, CRCError and VerifyError status bits should be checked. If there was an error, the operation should probably be aborted. Note that if the verify operation is being used to find the proper Header, errors in the DriveNotReady, WriteFault and Overrun are fatal whereas HeaderTag, CRC and Verify errors only indicate the wrong field was found. One should try every field on the track before giving up.

When Label or Data fields are verified, Step 2 is left out and the delay in Step 3 can be shortened. The HeaderTag bit is also ignored at the end of a Label or Data field operation.

*Conclusion*

This concludes the section on usage of the controller. The grand scheme for using the disk proceeds as follows:

1. After power on, wait for DriveNotReady to drop.

2. Clear the WriteFault line as shown in the Control Register section.

3. Recalibrate the read/write heads by stepping 20 cylinders in, then 222 (202+20 for SA4000 or SA4100) or 276 (256+20 for SA1000) out, looking for Track00 after each step is complete.

4. Seek to the desired cylinder by having the microcode issue the proper number of pulses on the Step line with the desired direction set on DirectionIn.

5. Perform the desired data transfer as outlined above.

6. If there are errors, retry. If the errors involve the WriteFault line, clear it and retry. If WriteFault errors persist, make sure Overrun isn't responsible. If the errors indicate the proper sector can't be found, try recalibrating.

7. Repeat Steps 4 through 7 as necessary.

## 5.2 Trident Disk Controller

(To be added)

## 6.0 Ethernet Controller

(To be added)

## 7.0 LSEP Controller

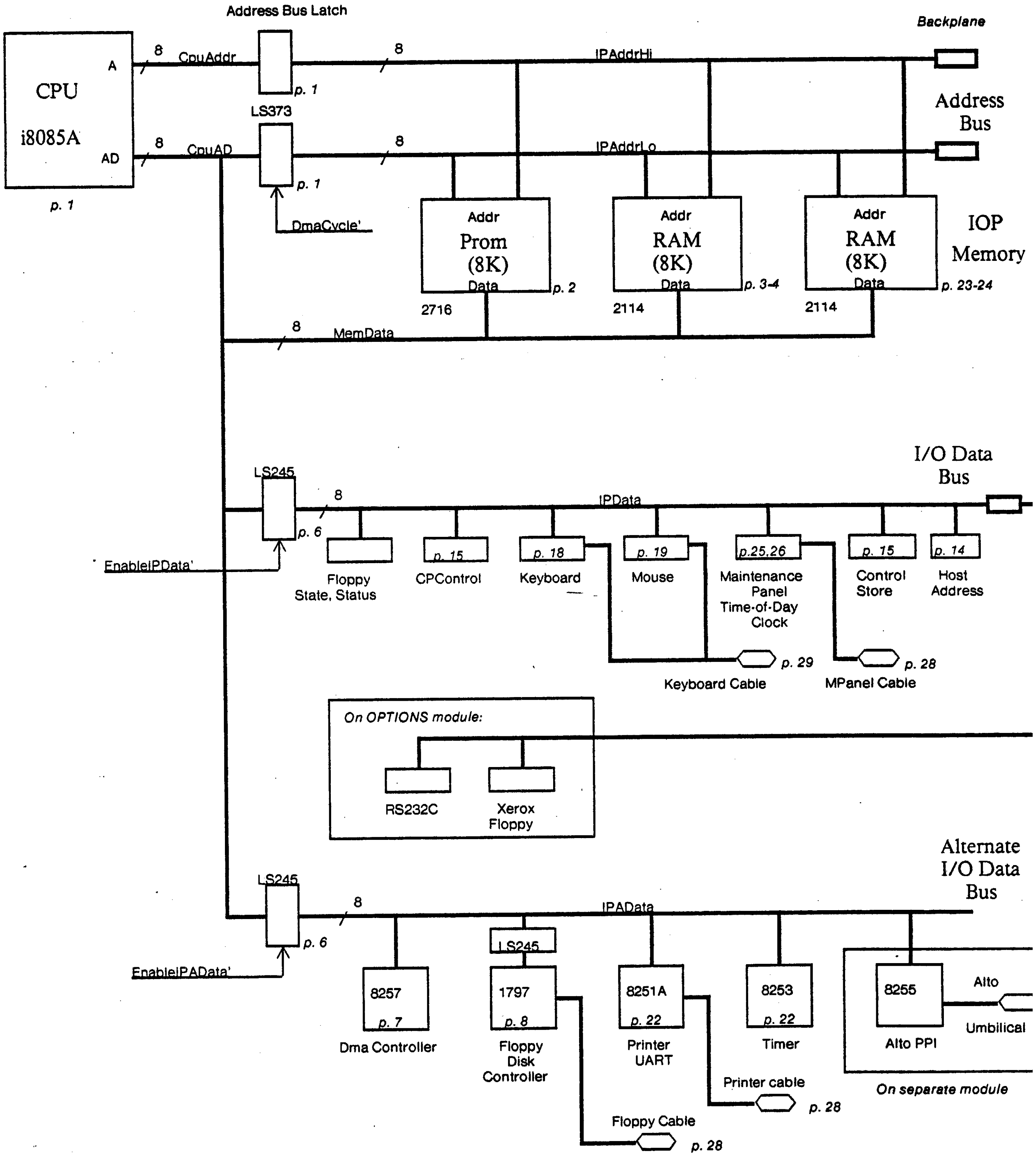
(To be added)

## 8.0 Magnetic Tape Controller

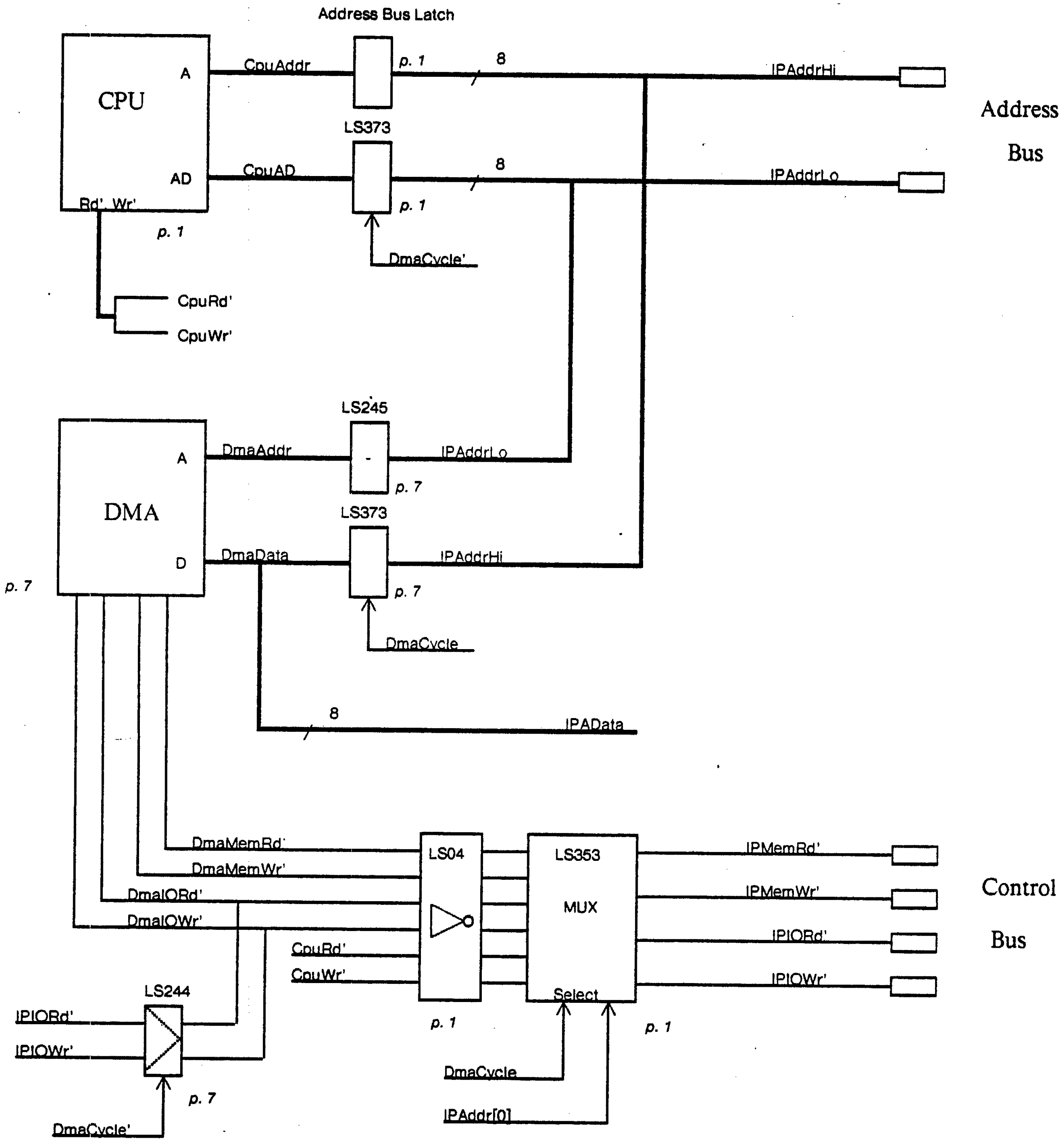
(To be added)

## 9.0 Input/Output Processor (IOP)

(To be added)



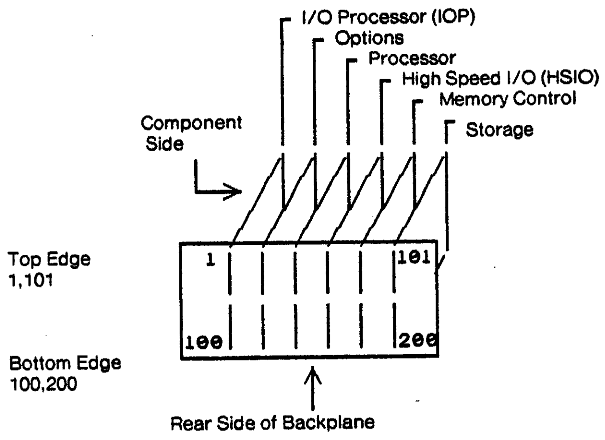
BLOCK DIAGRAM: I/O PROCESSOR DATA PATHS



BLOCK DIAGRAM: I/O PROCESSOR CONTROL ORGANIZATION

# Dandelion Backplane

## *Physical arrangement*



## *Files*

[Iris]<Workstation>Backplane>
Backplane-C.press
Backplane-C.dm

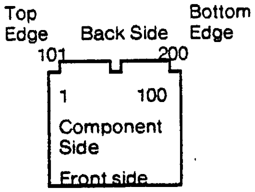
## *Backplane Signals*

| Card | IOP | Options | Central Processor | High Speed I/O | Memory Control | Storage | |
|---|---|---|---|---|---|---|---|
| Total Signal lines used | 131 | 166 | 140 | 141 | 155 | 66 | 170 max. per card |
| I/O Connectors on front of boards | Floppy<br>Keyboard<br>Printer<br>MaintenanceP<br>Alto umb. | LSEP/Ethernet<br>RS232/RS366 | | SA4XXX<br>SA100X<br>Display | | | |

## *Power distribution*

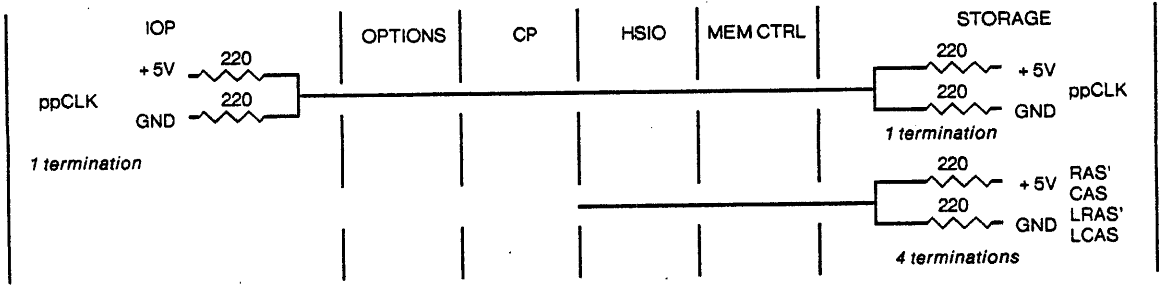
*Backplane Power & Ground* — *30 lines total*

| Voltage | Backplane Pins |
|---|---|
| +12 V | 1,101 |
| + 5 V | 50,51,150,151 |
| Gnd | 10,20,30,40,60,70,80,90,110,120,130,140,160,170,180,190 |
| - 5V | 100,200 |
| - 12 V | 98,198 |
| No Conn. | 97,99,197,199 |



2- 100 pin connectors
Pins on .1" centers
.6" between pins 50 & 51
AMP # 530826-3

## *Termination of clock signals*



Terminations are placed on the IOP and STORAGE cards.

| | | | | | | |
|---|---|---|---|---|---|---|
| XEROX<br>SDD | Project<br>Dandelion | Backplane Description<br>General Characteristics | File<br>WSBackplane.sil | Designer<br>Ogus | Rev<br>C | Date<br>9/26/80 |

| | IOP | | | OPTIONS | | | CP | | | HSIO | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 02 | | | 02 | Cycle.1' | Click.0 | 02 | Cycle1' | Click.0 | 02 | Cycle1' | Click.0 | 02 |
| 03 | | | 03 | Cycle.2' | Click.1 | 03 | Cycle2' | Click.1 | 03 | Cycle2' | Click.1 | 03 |
| 04 | | | 04 | Cycle.3' | Click.2 | 04 | Cycle3' | Click.2 | 04 | Cycle3' | Click.2 | 04 |
| 05 | Spare22 | Spare23 | 05 | Spare22 | Spare23 | 05 | | | 05 | RAS' | LRAS' | 05 |
| 06 | Spare2 | | 06 | Spare2 | | 06 | | | 06 | CAS | LCAS | 06 |
| 07 | Spare20 | Spare21 | 07 | Spare20 | Spare21 | 07 | Spare20 | Spare21 | 07 | WPulse | DR/C | 07 |
| 08 | Spare18 | Spare19 | 08 | Spare18 | Spare19 | 08 | Spare18 | Spare19 | 08 | | | 08 |
| 09 | ppCLK | | 09 | ppCLK | | 09 | ppCLK | | 09 | ppCLK | | 09 |
| 11 | Spare16 | Spare17 | 11 | Spare16 | Spare17 | 11 | AllowWrite | | 11 | AllowWrite | | 11 |
| 12 | IOPClk | | 12 | IOPClk | | 12 | | | 12 | | | 12 |
| 13 | Spare14 | Spare15 | 13 | Spare14 | Spare15 | 13 | MAR← | mem | 13 | MAR← | mem | 13 |
| 14 | Spare12 | Spare13 | 14 | Spare12 | Spare13 | 14 | | ←MStatus' | 14 | | ←MStatus' | 14 |
| 15 | IOPDataOut | BRClk | 15 | IOPDataOut | BRClk | 15 | MapRef | MCtl←' | 15 | MapRef | MCtl←' | 15 |
| 16 | SelTroyMode | Spare9 | 16 | SelTroyMode | Spare9 | 16 | Refresh' | | 16 | Refresh' | | 16 |
| 17 | Wait | IOPReset' | 17 | Wait | IOPReset' | 17 | Wait | IOPReset' | 17 | Wait | IOPReset' | 17 |
| 18 | TrIndex | TrHdLd | 18 | TrIndex | TrHdLd | 18 | Spare26 | Spare27 | 18 | Spare26 | Spare27 | 18 |
| 19 | TrReady | TrStep | 19 | TrReady | TrStep | 19 | Spare24 | Spare25 | 19 | Spare24 | Spare25 | 19 |
| 21 | IOPOData←' | IOPCtl←' | 21 | IOPOData←' | IOPCtl←' | 21 | IOPOData←' | IOPCtl←' | 21 | | | 21 |
| 22 | TrTK00 | TrDirIn | 22 | TrTK00 | TrDirIn | 22 | KOData←' | KCtl←' | 22 | KOData←' | KCtl←' | 22 |
| 23 | | | 23 | EOData←' | EICtl←' | 23 | EOData←' | EICtl←' | 23 | EOData←' | EICtl←' | 23 |
| 24 | TrWrProt | TrWrGate | 24 | TrWrProt | TrWrGate | 24 | DCtlFifo←' | DCtl←' | 24 | DCtlFifo←' | DCtl←' | 24 |
| 25 | TrRdData | TrWrData | 25 | TrRdData | TrWrData | 25 | DBorder←' | | 25 | DBorder←' | | 25 |
| 26 | EWrite' | EOCtl←' | 26 | EWrite' | EOCtl←' | 26 | EWrite' | EOCtl←' | 26 | EWrite' | EOCtl←' | 26 |
| 27 | KCmd←' | IOOutSp4←' | 27 | KCmd←' | IOOutSp4←' | 27 | KCmd←' | IOOutSp4←' | 27 | KCmd←' | IOOutSp4←' | 27 |
| 28 | | | 28 | POData←' | PCtl←' | 28 | POData←' | PCtl←' | 28 | POData←' | PCtl←' | 28 |
| 29 | Spare6 | Spare7 | 29 | Spare6 | Spare7 | 29 | Spare6 | Spare7 | 29 | Spare6 | Spare7 | 29 |
| 31 | | | 31 | EIData' | EStatus' | 31 | EIData' | EStatus' | 31 | EIData' | EStatus' | 31 |
| 32 | Spare4 | Spare5 | 32 | Spare4 | Spare5 | 32 | ←KIData' | ←KStatus' | 32 | ←KIData' | ←KStatus' | 32 |
| 33 | | | 33 | | KWrite' | 33 | ←KTest' | KWrite' | 33 | ←KTest' | KWrite' | 33 |
| 34 | ←IOPIData' | ←IOPStatus' | 34 | ←IOPIData' | ←IOPStatus' | 34 | ←IOPIData' | ←IOPStatus' | 34 | | | 34 |
| 35 | ←IOInSp2' | | 35 | ←IOInSp2' | PrtReq' | 35 | ←IOInSp2' | PrtReq' | 35 | ←IOInSp2' | PrtReq' | 35 |
| 36 | IOPALE | Spare3 | 36 | IOPALE | Spare3 | 36 | IOPALE | Spare3 | 36 | IOPALE | Spare3 | 36 |
| 37 | CSParErr | | 37 | CSParErr | EndLine' | 37 | CSParErr | EndLine' | 37 | | EndLine' | 37 |
| 38 | IODisp.0 | IODisp.1 | 38 | IODisp.0 | IODisp.1 | 38 | IODisp.0 | IODisp.1 | 38 | IODisp.0 | IODisp.1 | 38 |
| 39 | YIODisp.0 | YIODisp.1 | 39 | YIODisp.0 | YIODisp.1 | 39 | YIODisp.0 | YIODisp.1 | 39 | YIODisp.0 | YIODisp.1 | 39 |
| 41 | X.0 | X.1 | 41 | X.0 | X.1 | 41 | X.0 | X.1 | 41 | X.0 | X.1 | 41 |
| 42 | X.2 | X.3 | 42 | X.2 | X.3 | 42 | X.2 | X.3 | 42 | X.2 | X.3 | 42 |
| 43 | X.4 | X.5 | 43 | X.4 | X.5 | 43 | X.4 | X.5 | 43 | X.4 | X.5 | 43 |
| 44 | X.6 | X.7 | 44 | X.6 | X.7 | 44 | X.6 | X.7 | 44 | X.6 | X.7 | 44 |
| 45 | X.8 | X.9 | 45 | X.8 | X.9 | 45 | X.8 | X.9 | 45 | X.8 | X.9 | 45 |
| 46 | X.10 | X.11 | 46 | X.10 | X.11 | 46 | X.10 | X.11 | 46 | X.10 | X.11 | 46 |
| 47 | X.12 | X.13 | 47 | X.12 | X.13 | 47 | X.12 | X.13 | 47 | X.12 | X.13 | 47 |
| 48 | X.14 | X.15 | 48 | X.14 | X.15 | 48 | X.14 | X.15 | 48 | X.14 | X.15 | 48 |
| 49 | | | 49 | Y.0 | Y.1 | 49 | Y.0 | Y.1 | 49 | Y.0 | Y.1 | 49 |
| 52 | | | 52 | Y.2 | Y.3 | 52 | Y.2 | Y.3 | 52 | Y.2 | Y.3 | 52 |
| 53 | | | 53 | Y.4 | Y.5 | 53 | Y.4 | Y.5 | 53 | Y.4 | Y.5 | 53 |
| 54 | | | 54 | Y.6 | Y.7 | 54 | Y.6 | Y.7 | 54 | Y.6 | Y.7 | 54 |
| 55 | | | 55 | Y.8 | Y.9 | 55 | Y.8 | Y.9 | 55 | Y.8 | Y.9 | 55 |
| 56 | | | 56 | Y.10 | Y.11 | 56 | Y.10 | Y.11 | 56 | Y.10 | Y.11 | 56 |
| 57 | | | 57 | Y.12 | Y.13 | 57 | Y.12 | Y.13 | 57 | Y.12 | Y.13 | 57 |
| 58 | | | 58 | Y.14 | Y.15 | 58 | Y.14 | Y.15 | 58 | Y.14 | Y.15 | 58 |
| 59 | DmaReqC' | DmaAckC' | 59 | DmaReqC' | DmaAckC' | 59 | YH.0 | YH.1 | 59 | YH.0 | YH.1 | 59 |
| 61 | DmaReqA' | DmaAckA' | 61 | DmaReqA' | DmaAckA' | 61 | YH.2 | YH.3 | 61 | YH.2 | YH.3 | 61 |
| 62 | DmaReqB' | DmaAckB' | 62 | DmaReqB' | DmaAckB' | 62 | YH.4 | YH.5 | 62 | YH.4 | YH.5 | 62 |
| 63 | DmaCycle | ExtWaitReq' | 63 | DmaCycle | ExtWaitReq' | 63 | YH.6 | YH.7 | 63 | YH.6 | YH.7 | 63 |
| 64 | IOPIntReq0 | IOPIntReq1 | 64 | IOPIntReq0 | IOPIntReq1 | 64 | Pt.0 | Pt.1 | 64 | Pt.0 | Pt.1 | 64 |
| 65 | IOPIntReq2 | IOPIntReq3 | 65 | IOPIntReq2 | IOPIntReq3 | 65 | Pt.2 | | 65 | Pt.2 | | 65 |
| 66 | IOPSel.0' | IOPSel.1' | 66 | IOPSel.0' | IOPSel.1' | 66 | Disp-Proc' | MemErr | 66 | Disp-Proc' | MemErr | 66 |
| 67 | IOPSel.2' | IOPSel.3' | 67 | IOPSel.2' | IOPSel.3' | 67 | | | 67 | DAddr.0 | DAddr.1 | 67 |
| 68 | IOPSel.4' | IOPSel.5' | 68 | IOPSel.4' | IOPSel.5' | 68 | | | 68 | DAddr.2 | DAddr.3 | 68 |
| 69 | IOPAddr.00 | IOPAddr.01 | 69 | IOPAddr.00 | IOPAddr.01 | 69 | | | 69 | DAddr.4 | DAddr.5 | 69 |
| 71 | IOPAddr.02 | IOPAddr.03 | 71 | IOPAddr.02 | IOPAddr.03 | 71 | | | 71 | DAddr.6 | DAddr.7 | 71 |
| 72 | IOPAddr.04 | IOPAddr.05 | 72 | IOPAddr.04 | IOPAddr.05 | 72 | | | 72 | DAddr.8 | DAddr.9 | 72 |
| 73 | IOPAddr.06 | IOPAddr.07 | 73 | IOPAddr.06 | IOPAddr.07 | 73 | | | 73 | DAddr.10 | DAddr.11 | 73 |
| 74 | IOPAddr.08 | IOPAddr.09 | 74 | IOPAddr.08 | IOPAddr.09 | 74 | | | 74 | DAddr.12 | DAddr.13 | 74 |
| 75 | IOPAddr.10 | IOPAddr.11 | 75 | IOPAddr.10 | IOPAddr.11 | 75 | | | 75 | DAddr.14 | DAddr.15 | 75 |
| 76 | IOPAddr.12 | IOPAddr.13 | 76 | IOPAddr.12 | IOPAddr.13 | 76 | IOPAddr.12 | IOPAddr.13 | 76 | DData.0 | DData.1 | 76 |
| 77 | IOPAddr.14 | IOPAddr.15 | 77 | IOPAddr.14 | IOPAddr.15 | 77 | IOPAddr.14 | IOPAddr.15 | 77 | DData.2 | DData.3 | 77 |
| 78 | Spare0 | Spare1 | 78 | Spare0 | Spare1 | 78 | Spare0 | Spare1 | 78 | DData.4 | DData.5 | 78 |
| 79 | IOPMemRd' | IOPI/ORd' | 79 | IOPMemRd' | IOPI/ORd' | 79 | | | 79 | DData.6 | DData.7 | 79 |
| 81 | CSWE.a' | CSWE.b' | 81 | CSWE.a' | CSWE.b' | 81 | CSWE.a' | CSWE.b' | 81 | DData.8 | DData.9 | 81 |
| 82 | CSWE.c' | CSWE.d' | 82 | CSWE.c' | CSWE.d' | 82 | CSWE.c' | CSWE.d' | 82 | DData.10 | DData.11 | 82 |
| 83 | CSWE.e' | CSWE.f' | 83 | CSWE.e' | CSWE.f' | 83 | CSWE.e' | CSWE.f' | 83 | DData.12 | DData.13 | 83 |
| 84 | IOPReq' | ClrIOPReq' | 84 | IOPReq' | ClrIOPReq' | 84 | IOPReq' | ClrIOPReq' | 84 | DData.14 | DData.15 | 84 |
| 85 | | | 85 | DPReq' | ClrDPReq' | 85 | DPReq' | ClrDPReq' | 85 | DPReq' | ClrDPReq' | 85 |
| 86 | | | 86 | EReq' | ClrRefReq' | 86 | EReq' | ClrRefReq' | 86 | EReq' | ClrRefReq' | 86 |
| 87 | IOPMemWr' | IOPI/OWr' | 87 | IOPMemWr' | IOPI/OWr' | 87 | KReq' | ClrKFlags' | 87 | KReq' | ClrKFlags' | 87 |
| 88 | RefReq' | ReadCSEn' | 88 | RefReq' | ReadCSEn' | 88 | RefReq' | ReadCSEn' | 88 | RefReq' | | 88 |
| 89 | EORound | IOPWait | 89 | EORound | IOPWait | 89 | EORound | IOPWait | 89 | EORound | | 89 |
| 91 | WrTPCHigh' | WrTPCLow | 91 | WrTPCHigh' | WrTPCLow | 91 | WrTPCHigh' | WrTPCLow | 91 | | | 91 |
| 92 | IOPData.0 | IOPData.1 | 92 | IOPData.0 | IOPData.1 | 92 | IOPData.0 | IOPData.1 | 92 | | | 92 |
| 93 | IOPData.2 | IOPData.3 | 93 | IOPData.2 | IOPData.3 | 93 | IOPData.2 | IOPData.3 | 93 | | | 93 |
| 94 | IOPData.4 | IOPData.5 | 94 | IOPData.4 | IOPData.5 | 94 | IOPData.4 | IOPData.5 | 94 | | | 94 |
| 95 | IOPData.6 | IOPData.7 | 95 | IOPData.6 | IOPData.7 | 95 | IOPData.6 | IOPData.7 | 95 | | | 95 |
| 96 | SwTAddr | SwTAddr' | 96 | SwTAddr | SwTAddr' | 96 | SwTAddr | SwTAddr' | 96 | | | 96 |
| | 1-100 | 101-200 | | 1-100 | 101-200 | | 1-100 | 101-200 | | 1-100 | 101-200 | |

Above diagram is rear view (wiring side) of backplane.
ALL NUMBERS ARE IN DECIMAL.

# Dandelion Backplane Signals - 1

*Stamen1-4.sil in: [Iris]<Workstation>Backplane>Backplane-C.dm*

| | |
|---|---|
| *Rev* C | 9/26/80 |
| *Ogus* | |

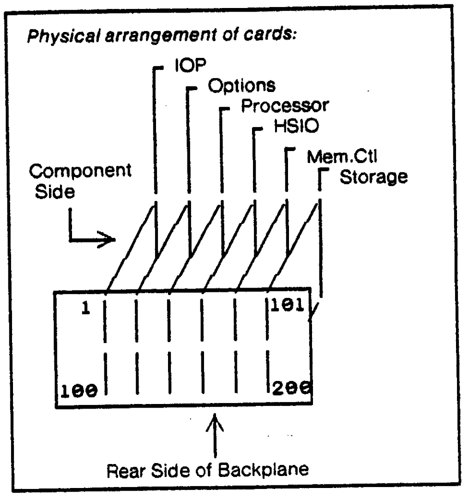
## Dandelion Backplane - 2

| Rev | C | 9/26/80 |
|---|---|---|
| Ogus | | |

| | MEM CTRL | | STORAGE | |
|---|---|---|---|---|
| Pin | 1-100 | 101-200 | 1-100 | 101-200 |
| 02 | Cycle1' | | | |
| 03 | Cycle2' | | | |
| 04 | Cycle3' | | | |
| 05 | RAS' | LRAS' | RAS' | LRAS' |
| 06 | CAS | LCAS | CAS | LCAS |
| 07 | WPulse | DR/C | | |
| 08 | | | | |
| 09 | ppCLK | | ppCLK | |
| 11 | AllowWrite | | | |
| 12 | Bank0' | | Bank0' | |
| 13 | MAR← | mem | | |
| 14 | | ←MStatus' | | |
| 15 | MapRef | MCtl←' | | |
| 16 | Refresh' | CRefresh' | Refresh' | CRefresh' |
| 17 | Wait | | | |
| 18 | SDO.00 | SDO.01 | SDO.00 | SDO.01 |
| 19 | SDO.02 | SDO.03 | SDO.02 | SDO.03 |
| 21 | SDO.04 | SDO.05 | SDO.04 | SDO.05 |
| 22 | SDO.06 | SDO.07 | SDO.06 | SDO.07 |
| 23 | SDO.08 | SDO.09 | SDO.08 | SDO.09 |
| 24 | SD0.10 | SDO.11 | SD0.10 | SDO.11 |
| 25 | SD0.12 | SDO.13 | SD0.12 | SDO.13 |
| 26 | SDO.14 | SDO.15 | SDO.14 | SDO.15 |
| 27 | SDO.16 | SDO.17 | SDO.16 | SDO.17 |
| 28 | SDO.18 | SDO.19 | SDO.18 | SDO.19 |
| 29 | SDO.20 | SDO.21 | SDO.20 | SDO.21 |
| 31 | | | | |
| 32 | SAddr.00 | SAddr.01 | SAddr.00 | SAddr.01 |
| 33 | SAddr.02 | SAddr.03 | SAddr.02 | SAddr.03 |
| 34 | SAddr.04 | SAddr.05 | SAddr.04 | SAddr.05 |
| 35 | SAddr.06 | SAddr.07 | SAddr.06 | SAddr.07 |
| 36 | Y1Latch | Y0Latch | Y1Latch | Y0Latch |
| 37 | Bank1' | Bank2' | Bank1' | Bank2' |
| 38 | MRef' | Write' | MRef' | Write' |
| 39 | | | | |
| 41 | X.0 | X.1 | | |
| 42 | X.2 | X.3 | | |
| 43 | X.4 | X.5 | | |
| 44 | X.6 | X.7 | | |
| 45 | X.8 | X.9 | | |
| 46 | X.10 | X.11 | | |
| 47 | X.12 | X.13 | | |
| 48 | X.14 | X.15 | | |
| 49 | Y.0 | Y.1 | | |
| 52 | Y.2 | Y.3 | | |
| 53 | Y.4 | Y.5 | | |
| 54 | Y.6 | Y.7 | | |
| 55 | Y.8 | Y.9 | | |
| 56 | Y.10 | Y.11 | | |
| 57 | Y.12 | Y.13 | | |
| 58 | Y.14 | Y.15 | | |
| 59 | YH.0 | YH.1 | | |
| 61 | YH.2 | YH.3 | | |
| 62 | YH.4 | YH.5 | | |
| 63 | YH.6 | YH.7 | | |
| 64 | Pt.0 | Pt.1 | | |
| 65 | Pt.2 | | | |
| 66 | Disp-Proc' | MemErr | | |
| 67 | DAddr.0 | DAddr.1 | | |
| 68 | DAddr.2 | DAddr.3 | | |
| 69 | DAddr.4 | DAddr.5 | | |
| 71 | DAddr.6 | DAddr.7 | | |
| 72 | DAddr.8 | DAddr.9 | | |
| 73 | DAddr.10 | DAddr.11 | | |
| 74 | DAddr.12 | DAddr.13 | | |
| 75 | DAddr.14 | DAddr.15 | | |
| 76 | DData.0 | DData.1 | | |
| 77 | DData.2 | DData.3 | | |
| 78 | DData.4 | DData.5 | | |
| 79 | DData.6 | DData.7 | | |
| 81 | DData.8 | DData.9 | | |
| 82 | DData.10 | DData.11 | | |
| 83 | DData.12 | DData.13 | | |
| 84 | DData.14 | DData.15 | | |
| 85 | SDI.20 | SDI.21 | SDI.20 | SDI.21 |
| 86 | SDI.18 | SDI.19 | SDI.18 | SDI.19 |
| 87 | SDI.16 | SDI.17 | SDI.16 | SDI.17 |
| 88 | SDI.14 | SDI.15 | SDI.14 | SDI.15 |
| 89 | SDI.12 | SDI.13 | SDI.12 | SDI.13 |
| 91 | SDI.10 | SDI.11 | SDI.10 | SDI.11 |
| 92 | SDI.08 | SDI.09 | SDI.08 | SDI.09 |
| 93 | SDI.06 | SDI.07 | SDI.06 | SDI.07 |
| 94 | SDI.04 | SDI.05 | SDI.04 | SDI.05 |
| 95 | SDI.02 | SDI.03 | SDI.02 | SDI.03 |
| 96 | SDI.00 | SDI.01 | SDI.00 | SDI.01 |

Above diagram is rear view (wiring side) of backplane.
All numbers are in DECIMAL.

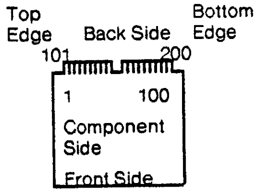
*Physical arrangement of cards:*



### Power & Ground

| Voltage | Pins |
|---|---|
| +12 V | 1,101 |
| +5 V | 50,51,150,151 |
| GND | 10,20,30,40,60<br>70,80,90,110,120,<br>130,140,160,170,<br>180,190 |
| -5 V | 100,200 |
| -12 V | 98,198 |
| No Conn | 97,99,197,199 |

### Card Edge Connector



2- 100 pin connectors
Pins on .1" centers
.6" between pins 50 & 51

2 - AMP # 530826-3

*Stamen5-6.sil in:*
*[Iri :]<Workstation>Backplane>Backplane-C.dm*