ALTO: A Personal Computer System Hardware Manual

August 1976

Abstract

This manual is a revision of the original description of the Alto: "Alto, A Personal Computer System." It includes a complete description of the Alto I and Alto II hardware and of the standard microcode (version 23).

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road / Palo Alto / California 94304

Contents

1.0 *INTRODUCTION*

This document is a description of the Alto, a small personal computing system originally designed at PARC. By 'personal computer' we mean a non-shared system containing sufficient processing power, storage, and input-output capability to satisfy the computational needs of a single user.

A standard Alto system includes:

An 875-line television monitor, oriented with the long tube dimension vertical. This monitor provides a 606 by 808 point display which is refreshed from main memory at 60 fields (30 frames) per second. It has programmable polarity, a low resolution mode which conserves memory space, and a cursor whose position and content are under program control.

An undecoded keyboard.

A mouse (pointing device) and five-finger keyset.

A Diablo Model 31 or Model 44 disk file.

An interface to the Ethernet, a 3 Mbps serial communications line that can connect a large number of Alto's and other computers.

A microprogrammed processor which controls the disk and display, and emulates a virtual machine whose characteristics are approximately those of the Data General Nova.

64K 16 bit words of 850ns semiconductor memory.

1K microinstruction RAM that can be read and written with special microcode to extend the facilities of the processor or to drive special I/O devices.

Optionally, a Diablo HyType printer.

The processor, disk, and their power supplies are packaged in a small cabinet. The other I/O devices may be a few feet away, and are pleasingly packaged for desk top use.

The remaining sections of this document will discuss the hardware and microcode of the standard configuration Alto. At present, two slightly different versions of the Alto exist: the Alto I and the Alto II. Most passages of this document pertain to both machines; those that apply to one only are clearly marked.

This document does not deal with non-standard peripheral devices that may have been interfaced to the Alto. Appendix C is a brief listing of non-standard interfaces and their designers.

People

The Alto was originally designed by Charles P. Thacker and Edward M. McCreight and was based on requirements and ideas contributed by several members of PARC's Computer Sciences Laboratory and Systems Sciences Laboratory. Bob Metcalfe and David Boggs designed the Ethernet and its controller. Tat Lam designed the Alto Analog Board.

The machine was re-engineered as the Alto II for ITG/SDD to a specification developed by John Ellenby. The engineering and production were carried out by EOD Special Programs Group, managed by Doug Stewart and coordinated on behalf of PARC and SDD by John Ellenby. The members of EOD/SPG who worked on the project are Doug Stewart, Ron Cude, Ron Freeman, Jim Leung, Tom Logan, Bob Nishimura, Abbey Silverstone, Nathan Tobol, and Ed Wakida.

This hardware manual has had a long history of modification and extension and has benefited from endless toil by numerous individuals. The present document is the responsibility of Diana Merry, Ed McCreight and Bob Sproull.

Conventions and Notation

Numbers in this document are decimal unless followed by "B"; thus 10 = 12B.

Bits in registers are numbered from the most significant bit (0) toward the least significant bit. Fields within registers are given by following the register name with a pair of numbers in parentheses: IR[a-b] describes the b-a+1 bit field of the IR register beginning with bit a and ending with bit b inclusive. IR[a] is short for IR[a-a].

The symbol "←" is used to mean "is replaced by." Thus IR[4-5] ← 2 means that the 2-bit field of IR including bits 4 and 5 is replaced by the bits 1 and 0 respectively. The symbol "=" is used as an equality test.

Memory is by convention divided into 256-word "pages." Page n thus contains addresses 256*n to 256*n+255 inclusive. The notation "rv(adr)" is used, as in Bcpl, to denote "the contents of the memory location with address adr."

2.0 *MICROPROCESSOR*

The microprocessor is shown schematically in Figures 1 and 2. A principal design goal in this system was to achieve the simplest structure adequate for the required tasks. As a result, the central portion of the processor contains very little application-specific logic, and no specialized data paths. The entire system is synchronous, with a clock interval of 170nsec. Microinstructions require one cycle for their execution.

A second design goal was to minimize the amount of hardware in the I/O controllers. This is achieved by doing most of the processing associated with I/O transfers with microprograms. To allow devices to proceed in parallel with each other and with CPU activity, a control structure was devised which allows the microprocessor to be shared among up to 16 fixed priority tasks. Switching between tasks requires very little overhead, and occurs typically every few microseconds.

2.1 *Arithmetic Section*

The arithmetic section of the processor consists of a 32-word by 16-bit register file R, and four registers, T, L, MAR, and IR. The registers are connected to the memory and to an ALU with a 16-bit parallel bus.

The ALU is a SN74181 type, restricted so that it can do only 16 arithmetic and logical functions. The ALU output feeds the L and MAR registers. T may also be loaded from the ALU output under certain conditions. L is connected to a shifter capable of left and right shifts by one place, and cycles of 8. It has a mode in which it does the peculiar 17-bit shifts of the Nova, and a mode which allows double-length shifts to be done.

The IR register is used exclusively by the Nova emulator to hold the current CPU instruction.

Attached to the bus is a 256-word read only memory (ROM) which holds arbitrary 16-bit constants.

The microprocessor executes instructions from a 1K word by 32-bit programmable read-only memory (PROM). The fields of the microinstruction are:

FIELD	NAME	MEANING
0-4	RSELECT	R Register Select
5-8	ALUF	ALU Function
9-11	BS	Bus Data Source
12-15	F1	Function 1
16-19	F2	Function 2
20		Load L
21		Load T
22-31	NEXT	Next microinstruction address (subject to modifiers)

R Select

The R select field specifies one of the 32 R cells to be loaded or read under control of the bus source field, or, in conjunction with the bus source field, one of the 256 locations to be read from the constant ROM.

The low order two bits of the R address (but not the constant ROM address) may be taken from fields in IR under control of the functions. This allows the emulator to address its central registers easily.

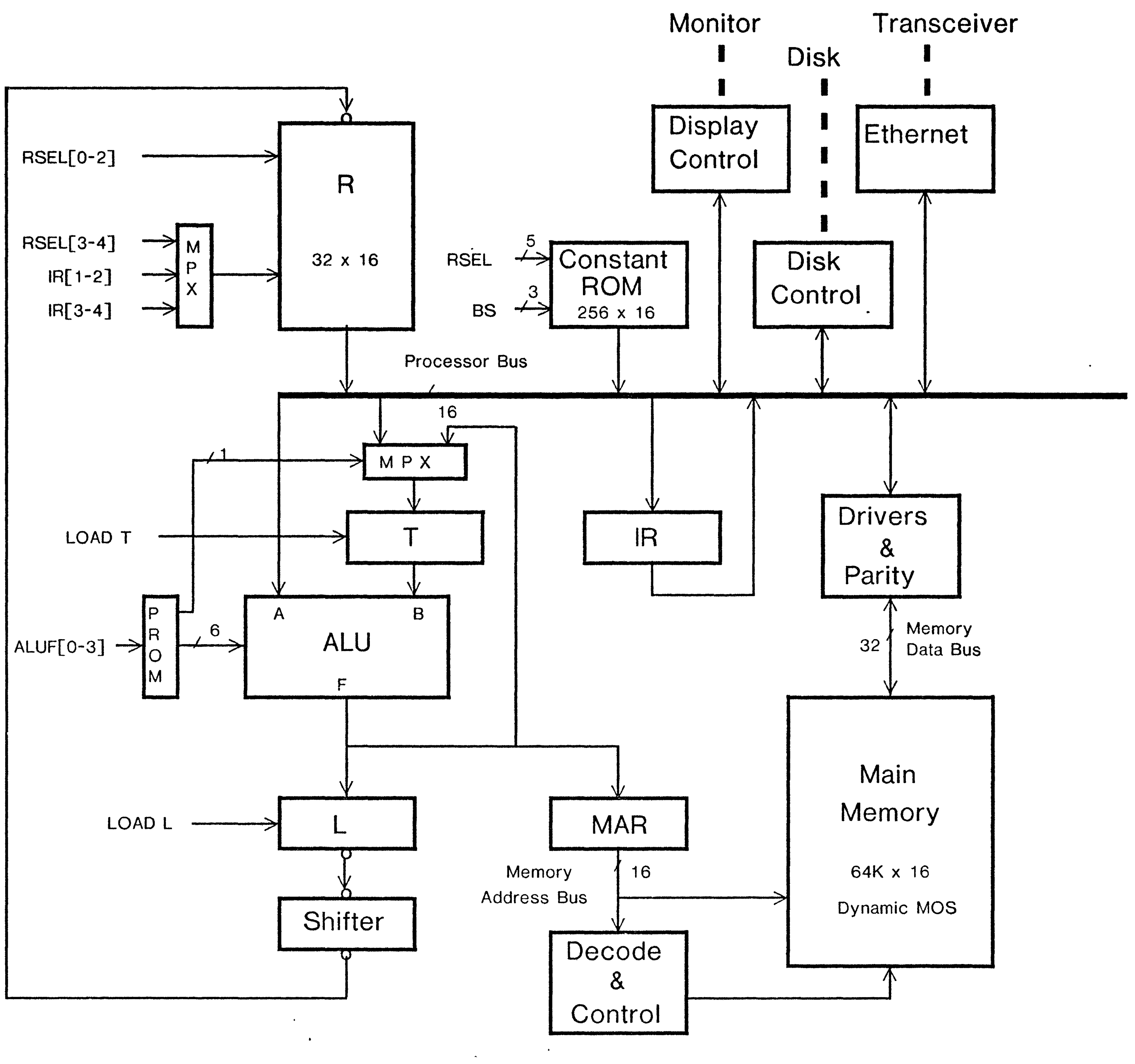

Figure 1 -- Processor Data Paths

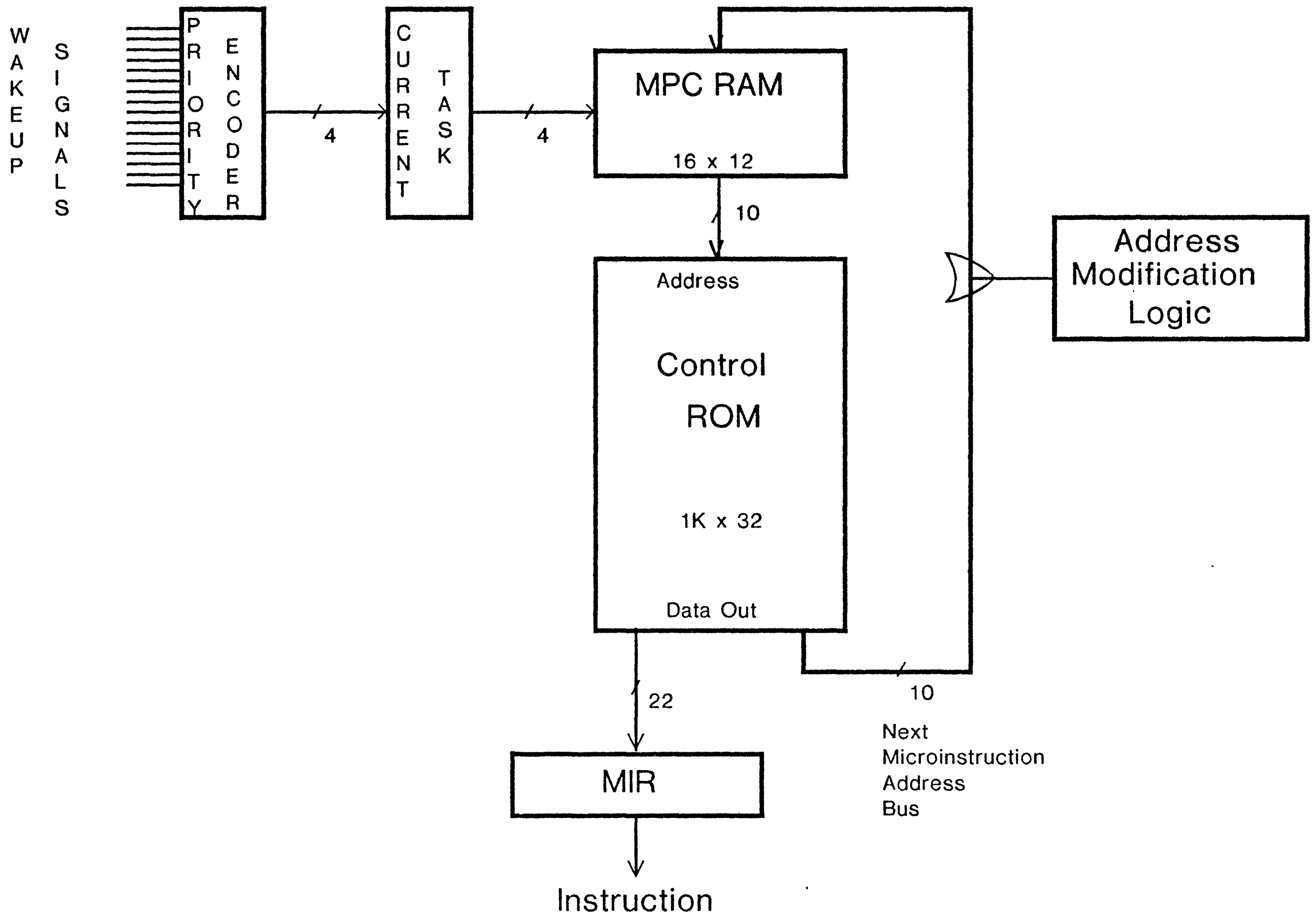

Figure 2 -- Processor Control

ALU Functions

The ALU function field controls the SN74181 ALU. This device can do a total of 48 arithmetic and logical operations, most of which are relatively useless. The 4-bit field is mapped by a PROM into the 16 most useful functions:

ALUF	FIELD FUNCTION	S3,S2,S1,S0,M,C INPUTS TO SN74181			
0	BUS	1111	1	0	(A)
1	T	1010	1	0	(B)
2	BUS OR T*	1110	1	0	(A+B)
3	BUS AND T	1011	1	0	(AB)
4	BUS XOR T	0110	1	0	(A XOR B)
5	BUS + 1*	0000	0	0	(A PLUS 1)
6	BUS - 1*	1111	0	1	(A MINUS 1)
7	BUS + T	1001	0	1	(A PLUS B)
10B	BUS - T	0110	0	0	(A MINUS B)
11B	BUS - T - 1	0110	0	1	(A MINUS B MINUS 1)
12B	BUS + T + 1*	1001	0	0	(A PLUS B PLUS 1)
13B	BUS+SKIP*	0000	0	SKIP'	(A PLUS 1)
14B	BUS.T* (AND)	1011	1	0	(AB)
15B	BUS AND NOT T	0111	1	0	(A & NOT B)
16B-17B	UNDEFINED				

*If T is loaded during an instruction which specifies this function, it will be loaded from the ALU output rather than from the bus.

Bus Sources

The bus data source field specifies one of 8 data sources for the bus:

VALUE	NAME	SOURCE
0	←RName	Read R
1	RName←	Load R*
2		Nothing (-1)
3	←KSTAT	Kstat (disk control status bits)**
4	←KDATA	Kdata (16 bits of disk data)**
5	←MD	Memory data
6	←MOUSE	Mouse data (4 bits, remainder of word is 1)
7	←DISP	Disp (low order 8 bits of IR, sign extended)

*This is not logically a source, but because R is gated to the bus during both reading and writing, it is included in the source specifiers. Load R forces the BUS to 0, so that T← ALUFunction(0,T) may be executed simultaneously.

**By convention, these bus sources are task specific, i.e., their meaning depends on the currently active task. ←KSTAT and ←KDATA are the interpretations used during the disk sector and word tasks.

Special Functions

The two function fields specify the address modifiers, register load signals (other than those for R, L and T), and other special conditions required in the processor. The first eight conditions specified by each field are interpreted identically by all tasks (except BLOCK), but the interpretation of the second eight depends on the active task. The task-independent functions are given below, the task-specific functions are included with the task descriptions.

FUNCTION 1:

F1	NAME	MEANING
0	----	No Activity

1	MAR←	Load MAR from ALU output; start main memory reference (see section 2.3).
2	TASK	Switch tasks if higher priority wakeup is pending.
3	BLOCK	Disable current task until re-enabled by hardware-generated condition. (Note: This is simply a hardware convention.)
4	←L LSH 1	Left shift L one place*
5	←L RSH 1	Right shift L one place*
6	←L LCY 8	Cycle L (8 places)*
7	←CONSTANT	Put on the bus the constant from the ROM location addressed by RSELECT.BS

*Modified by DNS (do Nova shifts) function, and MAGIC function.

FUNCTION 2:

F2	NAME	MEANING
0	----	No Activity
1	BUS=0	NEXT←NEXT or (if (BUS=0) then 1 else 0)
2	SH<0	NEXT←NEXT or (if (SHIFTER OUTPUT <0) then 1 else 0)
3	SH=0	NEXT←NEXT or (if (SHIFTER OUTPUT =0) then 1 else 0)
4	BUS	NEXT←NEXT or BUS(6,15)
5	ALUCY	NEXT←NEXT or LastALUC0*
6	MD←	Deliver BUS data to memory (see section 2.3)
7	←CONSTANT	Same as F1=7

*The carry used is that produced by the ALU function which last loaded the L register.

2.2 *Constant Memory*

The constant memory is a 256 x 16 PROM which holds arbitrary constants. The constant memory is gated to the bus by F1=7, F2=7, or BS≥4. The constant memory is addressed by the (8 bit) concatenation of RSELECT and BS. The intent in enabling constants with BS≥4 is to provide a masking facility, particularly for the ←MOUSE and ←DISP bus source. This works because the processor bus ANDs if more than one source is gated to it. Up to 32 such mask constants can be provided for each of the 4 bus sources ≥4.

Alto I: Note that it is not possible to use a constant other than -1 with the ←MD bus source, because memory parity is calculated on the bus, and a parity error will result if bits are marked off in a word fetched from memory.

2.3 *Main Memory*

Main memory references are handled differently on Alto I and Alto II. It is, however, possible to write most microcode so that it will operate correctly on both machines.

Alto I and Alto II: A memory reference is initiated by executing F1=6, MAR←. The results of a read operation are delivered somewhat later onto the bus with BS=5, ←MD. A store into the addressed memory location is achieved with F2=6, MD←. The microprogram partially controls memory timing, and must observe certain rules to insure correct operation.

a) A minimum of one microinstruction must intervene between the initiation of a memory reference and an MD← or ←MD.

b) Although the exact details of memory timing differ on Alto I and Alto II, both machines share the property that the processor will suspend execution of microinstructions if an ←MD or MD← is executed before the memory interface is prepared to deliver or accept data.

c) The memory checks parity on all fetches, unless the cycle is a refresh cycle or the address is between 177000B and 177777B inclusive, in which case an I/O device is being referenced. Parity errors result in activation of the highest-priority task (task number 15) whose purpose is to deal with the error (see section 5.6).

d) If RSELECT = 37B during the instruction which starts the memory, a refresh cycle is assumed and all memory cards are activated. This is used by the refresh task.

e) MAR← cannot be invoked in the same instruction as ←MD of a previous access.

Alto I:

f) During the fourth cycle after MAR has been loaded, if F2=6, MD←, a store of bus data into the word addressed by MAR will occur. The MD← may not be issued later than the fourth cycle. (Note: Some Alto I's have been modified to allow a "double-word store." On these machines, it is permissible to issue two MD← instructions in a row, the first coming in the fourth cycle following the MAR←, and the second following directly. If MAR is loaded with an even address adr, the two words will be stored at adr and adr+1 respectively.)

g) During the fourth cycle of a reference, if BS=5, ←MD, the reference is a fetch of the word addressed by MAR. During the fifth cycle of a reference, if BS=5, ←MD, the odd word of the doubleword addressed by MAR is delivered. The memory cycle is extended by one cycle if both words of a doubleword are fetched. If MD is referenced during the fifth cycle, it must have also been referenced during the fourth.

Alto II:

f) During the third cycle after MAR has been loaded, if F2=6, MD←, a store of bus data into the word addressed by MAR will occur. The MD← may not be issued later than the third cycle. Alto II's allow a "double-word store:" it is permissible to issue two MD← instructions in a row, the first coming in the third cycle following the MAR←, and the second following directly. If MAR is loaded with an address adr, the two words will be stored at adr and (adr XOR 1) respectively.

g) During the fourth cycle of a reference, if BS=5, ←MD, the reference is a fetch of the word addressed by MAR. During the fifth cycle of a reference, if BS=5, ←MD, the other word of the doubleword addressed by MAR is delivered. The memory cycle is extended by one cycle if both words of a doubleword are fetched.

Because the Alto II latches memory contents, it is possible to execute ←MD anytime after the fourth cycle of a reference and obtain the results of the read operation. This convention permits a double-word "exchange" operation to be coded as follows:

```
MAR←adr;
NOP;
MD←newContents1;          address= adr
MD←newContents2;          address= adr  XOR  1
L←MD;                     address= adr
T←MD;                     address= adr  XOR  1
oldContents1←L,  L←T;
oldContents2←L;
```

2.4 *Microprocessor Control*

Control of the Alto microprocessor is shared among 16 "tasks" arranged in a priority order. The tasks are numbered 0 to 15: 0 is the lowest priority task and 15 is the highest. The lowest priority task is the emulator task which fetches instructions and executes them.

The only state saved for each task is a "micro program counter," MPC. The current task number, saved in the current task register, addresses a 16 by 12 MPC RAM. The result is an MPC for the current task; it is used to address a 1K by 32-bit microinstruction memory (MI ROM). The microinstruction memory produces an instruction and the address of its successor NEXT[0-9]. This successor address may be modified by merging bits into it under control of the function fields of the current microinstruction. This limited branching capability makes coding more difficult than with a more general scheme, but not seriously so, as examples of microcode demonstrate.

The amount of memory available for microinstructions is often extended by an additional 1K of control memory implemented with RAM. Because the MPC RAM produces 12 bits, enough are available (11) to address both the microinstruction ROM and RAM. The microinstruction RAM may be loaded or read by special CPU instructions, and provisions exist for causing any of the 16 tasks to execute instructions from it (see section 8).

At the end of each cycle, the microinstruction register (MIR) and the MPC are loaded, and the cycle repeats. There is only one phase of the system clock. It is true during the last 25 ns. of every instruction.

Tasks

If the processor executes the TASK function (FI=2) during an instruction, the current task register is loaded (at the end of the instruction) with the number of the current highest priority task as determined by the priority encoder. This causes the next instruction to be fetched from the ROM location specified by the saved task's MPC. One additional instruction is executed before the switch becomes effective. A version of the current task register which is delayed from the MPC RAM address by one cycle exists so that this instruction can execute task-specific functions, but these functions must do no address modification, since any modification would affect the new task. The situation for two streams of instructions A-F and J-M in two different tasks is shown below:

Instruction Being Executed	Instruction Being Fetched	Address Stored MPC at End of Cycle
A	B	C
B	C	D
C *	D	E
D	J	K
J **	K	L
K ***	L	M
L	E	F
E	F	G

*Instruction C allows task switching. New task's MPC = J.
**Instruction J does an operation which removes its task's wakeup request.
***Instruction K allows task switching, and the original task is now highest priority.

The "wakeup signals" which drive the priority encoder are hardware-generated and are not accessible to the microprogram. When a running task executes the TASK function, control will switch to another task only if a higher priority task has a wakeup signal held true, or if the current task no longer has a wakeup signal true. In the latter case, control goes to a lower priority task. The lowest priority task is the CPU emulator, which is always requesting wakeup.

The BLOCK function (F1=3) is used, by convention, to signal a hardware device associated with the currently running task to remove its wakeup signal. This function is *not* accomplished by the Alto microprocessor, but rather by the individual device interfaces.

The TASK function should be executed only at times when the current task has no state in L or T, and has no main memory operations in progress, since there is no provision in the hardware for saving this information.

Initialization

The only way in which the microprogram can affect the task structure is to request a task switch. In particular, it cannot affect the MPC's of tasks other than itself. This presents an initialization problem which is solved by having each task start at the location which is its task number (thus the emulator task finds its first instruction to execute at MPC=0). Task numbers are written into the MPC RAM during a reset cycle, which may be initiated manually or by a CPU instruction (see SIO instruction in section 3.3).

3.0 *EMULATOR*

The standard microcode on the Alto contains an "emulator" as the lowest-priority task. This code fetches, decodes, and executes instructions resident in the Alto memory whose encoding resembles that of the Data General Nova computers. This "standard" emulator can be replaced by changing the microcode that is executed as the lowest priority task, often by executing special emulator microcode in the microcode RAM.

3.1 *Standard Instruction Set (Nova)*

The standard instruction set is that of the Data General Nova, with the following differences:

> An address requires 16 bits, rather than the 15 on the Nova. Therefore, multi-level indirection is not possible, and all 16 bits of a register used for indexing are significant.
>
> There are no auto-index locations.
>
> The interrupt system is entirely different (see section 3.2).
>
> The I/O class of instructions is not implemented. Instead, the Alto has augmented the instruction set (see section 3.3).

Registers

The emulator state is contained in several registers:

> PC: The "program counter," which contains the 16-bit address of the next instruction to be fetched and executed.
>
> AC0, AC1, AC2, AC3: The accumulators, each of which contains 16 bits. Instructions are available for transferring contents of accumulators to and from memory registers and for performing arithmetic and logical operations among accumulators. The notation AC(n) is often used to refer to the contents of accumulator n (n=0,1,2,3).
>
> C: The "carry" bit which is modified by most arithmetic operations.
>
> Memory: The Alto has "64K" 16-bit memory words, addressed by values ranging from 0 to 176777B. Addresses 177000B to 177777B are reserved for various I/O device uses (see Appendix B).

Operations

The instructions are best described by breaking them into four groups according to the way the instructions are formatted (see figure 3).

Several of the instructions compute an "effective address" based on the values of the I (indirect), X (index) and DISP (displacement) fields of the M-group, J-group and some S-group instructions. The effective address calculation is best described by a brief "program." We define the function SExtend(x) to represent the sign-extension of the 8-bit number x.

SExtend(x) = (if x ge 200B then x+177400B else x)

```
E() = [                              //The symbol "E" denotes effective address
E←(                                  //Values of I,X, and DISP are from the instruction
if     X=0 then DISP                 //"page 0 addressing"
elseif X=1 then SExtend(DISP)+PC     //"relative addressing"
elseif X=2 then SExtend(DISP)+AC(2)  //"base register addressing"
elseif X=3 then SExtend(DISP)+AC(3)  //"base register addressing"
   )
if I ne 0 then E←rv(E)               //Now do single-level indirection
```

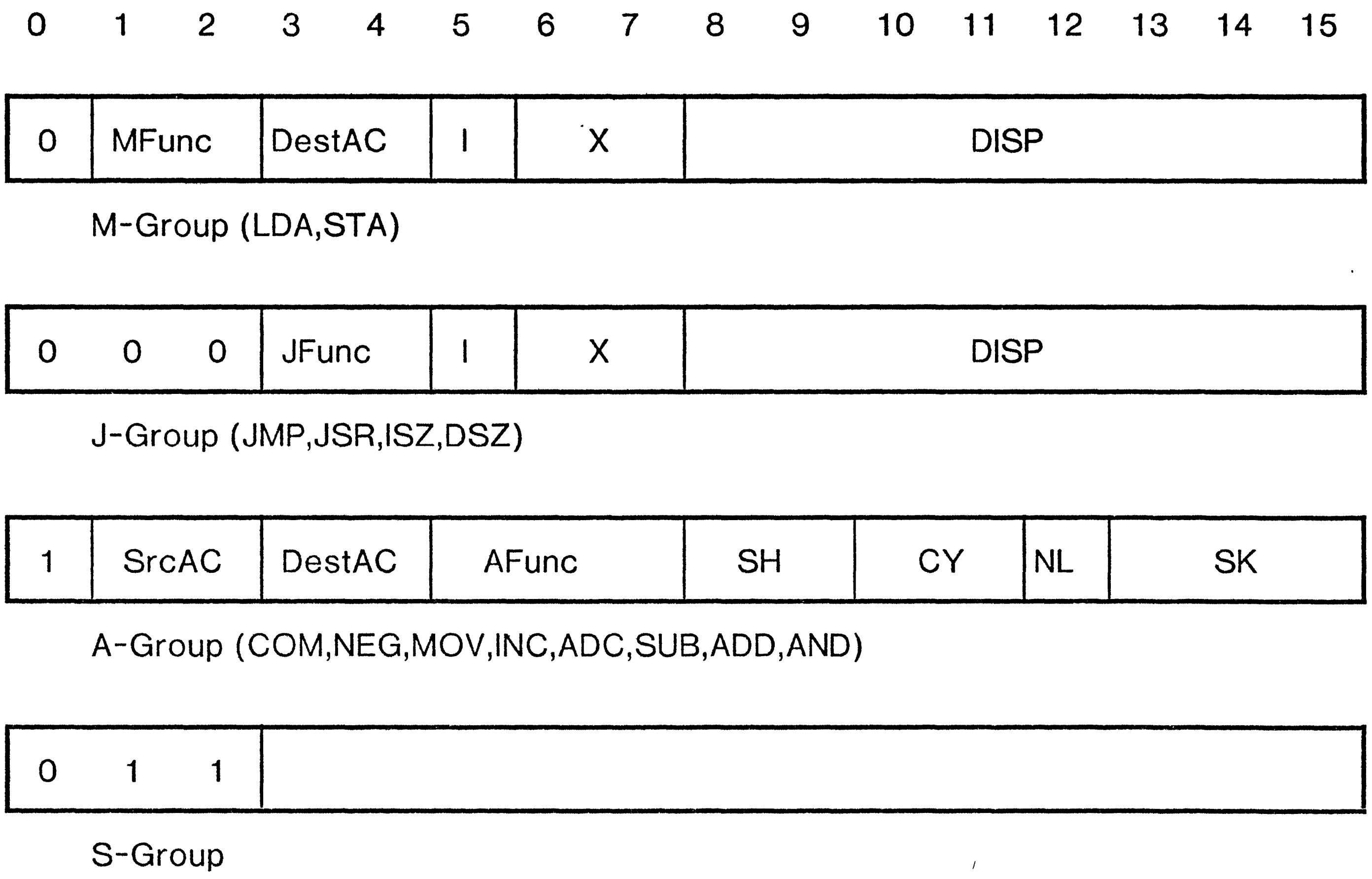
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 | MFunc | DestAC | I | X | DISP
M-Group (LDA,STA)
0 0 0 | JFunc | I | X | DISP
J-Group (JMP,JSR,ISZ,DSZ)
1 | SrcAC | DestAC | AFunc | SH | CY | NL | SK
A-Group (COM,NEG,MOV,INC,ADC,SUB,ADD,AND)
0 1 1
S-Group

Figure 3 -- Instruction Formats

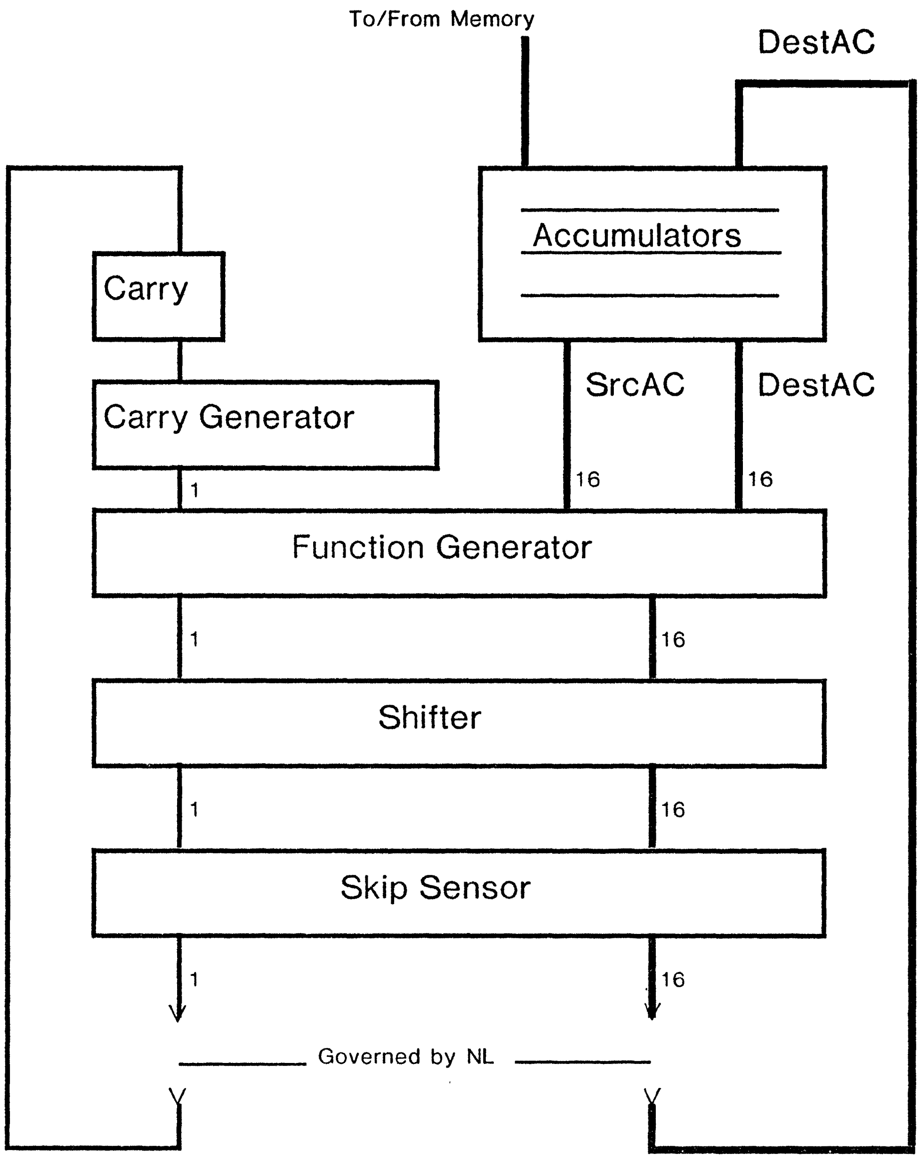

Figure 4 -- Instruction Execution

]

The notation for these addressing modes is demonstrated below. The DISP value is always specified first; the X value is not given explicitly, but is determined either by the address of the label or by a modifier ",2" or ",3" which specifies base register indexing:

```
JMP LABEL2          //Will use X=0 or 1 depending where LABEL2 is:
                    // If LABEL2 is in page 0, X=0; otherwise X=1.
JMP 15,3            // DISP=15; 3 means use AC3 as base register.
JMP @3              // The character @ causes I to be 1.
```

Note that instructions which compute an effective address always do so before any other operations. Thus JSR 1,3 computes the effective address of 1+AC(3) before saving PC+1 in AC3.

Memory Group Operations: The DestAC field specifies one of the four accumulators (DestAC=0 for AC0, DestAC=1 for AC1, etc.). The MFunc field specifies one of two operations:

LDA (MFunc=1): This operation loads an accumulator from memory. AC(DestAC)←rv(E).

STA (MFunc=2): This operation stores an accumulator into memory. rv(E)←AC(DestAC).

These instructions are written by giving the mnemonic, followed by the accumulator number (DestAC), followed by an effective address notation:

```
STA 3,.+4           //Store AC3 in the fourth location following this one
LDA 0,4,2           //Load AC0 from address=4+AC(2)
```

Jump and Modify Group Operations: The JFunc field specifies one of four operations:

JMP (JFunc=0): This operation causes a "jump" by changing the value of the PC. PC←E.

JSR (JFunc=1): This operation is useful when calling subroutines because it saves a return address in AC3. AC(3)←PC+1; PC←E.

ISZ (JFunc=2): This operation increments the contents of a memory cell and skips if the new contents are zero. rv(E)←rv(E)+1; if rv(E)=0 then PC←PC+1. This instruction does not alter the C bit.

DSZ (JFunc=3): This instruction decrements the contents of a memory cell and skips if the new contents are zero. rv(E)←rv(E)-1; if rv(E)=0 then PC←PC+1. This instruction does not alter the C bit.

These instructions are written by giving the mnemonic and the effective address notation:

```
JSR SUBR
JMP 1,3             //Jump to AC(3)+1
```

Arithmetic Group Operations: All 8 of these instructions operate on the contents of the accumulators and the carry bit. Typically, a binary operation involves the contents of the "source accumulator" (SrcAC) and the "destination accumulator" (DestAC) and leaves the result in the destination accumulator. The carry bit (C bit) and the PC can also be modified in the process.

The operation of the instructions is best explained by following the flow in figure 4. The 16-bit contents of the source and destination accumulators are fetched and passed to the function generator.

The carry generator produces an output that depends on the value of the C bit and the CY field of the instruction:

none (CY=0): The output is C.

Z (CY=1): The output is 0.

O (CY=2): The output is 1.

C (CY=3): The output is 1-C (i.e., the complement of C).

The function generator is controlled by the AFunc field; various values will be described below. It takes two 16-bit numbers and a carry input and generates a 16-bit Result and a carryResult.

The shifter is controlled by the SH field in the instruction:

none (SH=0): No shifting; the 17 output bits are the same as the 17 input bits.

L (SH=1): Rotate the 17 input bits left by one bit. This has the effect of rotating bit 0 left into the carry position and the carry bit into bit 15.

R (SH=2): Rotate the 17 bits right by one bit. Bit 15 is rotated into the carry position and the carry bit into bit 0.

S (SH=3): Swap the 8-bit halves of the 16-bit result. The carry is not affected.

The skip sensor tests various of the 17 bits presented to it and may cause a skip (PC←PC+1) if an appropriate condition is detected:

none (SK=0):	Never skip
SKP (SK=1):	Always skip
SZC (SK=2):	Skip if the carryResult is zero
SNC (SK=3):	Skip if the carryResult is non-zero
SZR (SK=4):	Skip if the 16-bit Result is zero
SNR (SK=5):	Skip if the 16-bit Result is non-zero
SEZ (SK=6):	Skip if either carryResult or Result is zero
SBN (SK=7):	Skip if both carryResult and Result are non-zero

The alert reader will detect that the SK field is microcoded. The skip condition can be described as:

```
skip = (SK2≠0) XOR
        ((SK0≠0 AND result=0) OR (SK1≠0 AND carryResult=0))
```

where SK0 is the first bit of the field, SK1 the second and SK2 the third.

The NL bit in the instruction controls the operation of the switch in the illustration. If NL=1, neither the destination accumulator nor the carry bit is loaded; otherwise the destination accumulator is loaded from Result and the carry bit from carryResult. The "no-load" feature is useful for instructions whose only use is testing some value. The character # is appended to the mnemonic for operations if the NL bit is to be set.

The AFunc operations are described below. Note that "Result" will be stored into the destination accumulator (DestAC) unless NL=1.

COM (AFunc=0) Complement: The function generator produces the logical complement of AC(SrcAC). It passes the carry bit unaffected.

NEG (AFunc=1) Negate: The function generator produces the two's complement of AC(SrcAC). If AC(SrcAC) contains zero, complement the value of the carry supplied to the function generator, otherwise supply the specified value.

MOV (AFunc=2) Move: The function generator passes AC(SrcAC) and the carry bit unaffected.

INC (AFunc=3) Increment: The Result produced is AC(SrcAC)+1; the carry is complemented if AC(SrcAC)=177777B.

ADC (AFunc=4) Add Complement: The Result produced is the sum of AC(DestAC) and the logical complement of AC(SrcAC). The carry bit is complemented if the addition generates a carry.

SUB (AFunc=5) Subtract: Subtracts by adding the two's complement of AC(SrcAC) to AC(DestAC). The carry bit is complemented if the addition generates a carry.

ADD (AFunc=6) Add. Adds AC(SrcAC) to AC(DestAC). The carry bit is complemented if the addition generates a carry.

AND (AFunc=7) And. The Result is the logical and of AC(SrcAC) and AC(DestAC). The carry is passed unaffected.

The arithmetic instructions are written by citing the AFunc mnemonic, followed optionally by the CY mnemonic, followed optionally by the SH mnemonic, followed optionally by the NL mnemonic. Then after a space, the source accumulator number is given, the destination accumulator number, and optionally an SK mnemonic. For example:

```
SUB 0,0             //Zero AC0 by subtracting it from itself
MOVZ 2,1            //Move AC2 to AC1, and zero C.
SUBZL 1,1           //Set AC1 to 1
ADC 0,0             //Set AC0 to 177777B
SUB# 2,3,SNR        //Skips if AC2 and AC3 are unequal but affects neither
COM# 1,1,SZR        //Skips if AC1 is 177777B but leaves it unchanged
SUBZ# 1,0,SZC       //Skips if AC0<AC1 unsigned
ADCZ# 1,0,SZC       //Skips if AC0≤AC1 unsigned
```

To subtract the constant 1 from AC1:

```
NEG 1,1
COM 1,1
```

To "or" together the contents of AC0 and AC1; results AC0:

```
COM 1,1
AND 1,0
ADC 1,0
```

To "xor" together the contents of AC0 and AC1; result in AC0:

```
MOV 0,2
ANDZL 1,2
ADD 1,0
SUB 2,0
```

To negate a double-length number in AC0 and AC1:

```
NEG 1,1,SNR
NEG 0,0,SKP
COM 0,0
```

To add the double-length number in AC2,AC3 to one in AC0,AC1:

```
ADDZ 3,1,SZC
INC 2,2
ADD 2,0
```

To subtract the double-length number in AC2,AC3 from one in AC0,AC1:

```
SUBZ 3,1,SZC
SUB 2,0,SKP
ADC 2,0
```

The Bcpl construct "if a gr b then ..." uses code which does a subtract and checks the sign. Unfortunately, this is not a true signed compare because the subtract may overflow. With this code, 2 gr 0 is true, but 077777B gr 100000B is false (077777B is the largest positive number and 100000B the largest negative. The code generated by Bcpl looks like:

```
LDA 0 4,2           //Pick up a
LDA 1 5,2           //Pick up b
ADCL# 1,0,SZC       //Subtract and check sign
JMP falsePart       //Not true
JMP truePart        //True
```

The "true signed compare" for a>b is:

```
        LDA 0 4,2               //Pick up a
        LDA 1 5,2               //Pick up b
        SUBZR 2,2               //Place 100000B in AC2
        AND 1,2                 //AC2=(if b<0 then 100000B else 0)
        ADDL 0,2                //CARRY=(if a and b signs differ then 1 else 0)
        ADC# 1,0,SNC            //
        JMP falsePart
        JMP truePart
```

3.2 *Interrupts*

The emulator microcode implements an interrupt structure which allows both I/O devices and programs to interrupt the main program. The interrupt system provides 15 channels of vectored interrupts with adjustable priority: the lowest-priority channel is numbered 1; the highest is numbered 15. The interrupt system uses one register in R (NWW, new wakeups waiting) which is inaccessible to the programmer, and a number of fixed locations in page 1:

ACTIVE (453B): This word contains 1's for the channels which are currently active. Bit n is set if channel n is active. Bit 0 is not used, and should not be set by any program.

WW (452B): This word contains bits for channels on which interrupts are pending. Bit 0 is not used.

PCLOC (500B): When an interrupt is initiated by the microcode, the PC is saved here.

INTVEC (501B) to INTVEC+14: Contains pointers to the service routines for the 15 interrupt channels. The first word corresponds to the highest priority interrupt channel (bit 15), the last corresponds to the lowest priority channel (bit 1).

The main loop of the emulator checks NWW during the fetch of each emulated instruction. If NWW is greater than zero, the microcode computes (NWW OR WW) AND ACTIVE. If this quantity is nonzero, an interrupt is caused. If not, NWW OR WW is stored in WW, NWW is cleared, and the instruction is restarted.

If the interrupt is caused, the microcode stores the program counter in PCLOC, sets bit 0 of NWW to disable further interrupts, clears the bit in NWW corresponding to the interrupt channel about to occur, and loads the PC with rv(INTVEC+15-CHANNEL).

Interrupts are caused by ORing into NWW or into WW. I/O device microcode usually has a dedicated location in which the program places a bitword for the interrupt(s) to be caused upon completion of I/O activity.

Only one interrupt channel is permanently assigned: the highest priority channel (bit 15) is triggered when a main memory parity error is detected.

The interrupt system uses three instructions:

DIR (61000B) Disable interrupts:
Sets bit 0 of NWW. Since NWW is negative, the check made at the start of every instruction will not process any new wakeup requests.

DIRS (61013B) Disable interrupts and skip if on:
Disable interrupts (see DIR, above), but skips the next instruction if interrupts were enabled at the start of this instruction.

EIR (61001B) Enable interrupts:
Clears bit 0 of NWW, and ORs WW into NWW to detect any interrupts which were requested (by ORing into WW) while interrupts were off.

BRI (61002B) Branch and return from interrupt:
This instruction clears bit 0 of NWW, ORs WW into NWW, and restores PC from PCLOC.

3.3 *Augmented Instruction Set*

Opcodes above 60000B, which are I/O instructions in the Nova, have been reassigned to instructions which augment the standard instruction set. Bits 3 through 7 of the instruction determine 32 opcodes, each of which may use the displacement field. One of these opcodes is used to represent up to 256 instructions which do not require a displacement or a parameter as part of the opcode.

Currently, only a small number of the available extra instructions have been implemented. The remaining unimplemented instructions all trap in some way. If no microcode RAM is present or IR[3-7]=37B, an unimplemented opcode causes the microcode to store the PC (which points one location beyond the instruction which caused the trap) in location TRAPPC, and simulate a JMP@ TRAPVEC ! IR[3-7]. TRAPPC (527B), and the 32 word trap vector starting at TRAPVEC (530B) are reserved locations in page 1. If a microcode RAM is present on the Alto, and IR[3-7] is not 37B, unimplemented opcodes will trap into the RAM (see section 8.6).

The currently assigned extra instructions and their operations are:

MUL (61020B) Unsigned multiply:

Multiplies the unsigned integers in AC1 and AC2 to generate a 32-bit product; add the product to the integer in AC0. Leave the high-order part of the result in AC0 and the low-order part in AC1. AC2 is unaffected.

DIV (61021B) Unsigned divide:

The double-length unsigned integer in AC0 and AC1 is divided by the unsigned integer in AC2. The quotient is left in AC1; the remainder in AC0. AC2 is unaffected. The instruction normally skips the next instruction; if overflow occurs (AC0 $\geq$ AC2 unsigned), DIV does not skip.

CYCLE (60000B):

Left cycle (rotate) the contents of AC0 by the amount specified in instruction bits 12-15, unless this value is zero, in which case cycle AC0 left by the amount specified in AC1. Leaves AC1 = cycle count mod 20B.

JSRII (64400B) Jump to subroutine double indirect, PC relative:

```
AC3←PC+1
PC←rv(rv(PC+DISP))
```

JSRIS (65000B) Jump to subroutine double indirect, AC2 relative:

```
AC3←PC+1
PC←rv(rv(AC2+DISP))
```

CONVERT (67000B);

The convert instruction does scan conversion of characters, i.e., it transfers data between an area of main memory containing a font and an area of memory containing a bit map to be displayed on the TV monitor.

Convert takes a number of arguments:

AC0 contains the address of the destination word into which the upper left corner of the character is to be placed, offset by NWRDS, the number of words to be displayed on each scan line (AC0=DWA-NWRDS).

AC3 points to a character pointer in the font for the character to be displayed (AC3=FONTBASE+CHARACTER CODE).

AC2+Displacement points to a two word table:

word 0:	NWRDS (number of words per to scan line); NWRDS < 128.
word 1:	DBA, the destination bit address corresponding to the left hand

edge of the character. Convert interprets this bit address reversed from the normal convention, i.e., 0 is the least significant bit, 15 the most significant bit.

Convert requires that a 16 word mask table be set up starting at MASKTAB (460B) in page 1. MASKTAB!n=(2**(n+1))-1 (0<n<16).

The format of an Alto font designed for use with CONVERT is given below; font files in this format conventionally have an extension ".AL". The CONVERT instruction does not examine the words at FONTBASE-2 and FONTBASE-1; these are provided solely for convenience of software.

FONTBASE-2:
The height of a line of text in scan lines. This number incorporates the effects of the highest and lowest character in the font, i.e. it is max(HD+XH)-min(HD) where the max and min are taken independently and HD and XH are defined below.

FONTBASE-1:

Bit 0:	0 = Fixed width font.
	1 = Proportional width font.
Bits 1-7:	Baseline -- number of scan-lines from top of highest character in font to the baseline.
Bits 8-15:	The width of the widest character in raster points.

FONTBASE to FONTBASE+377B:
Self-relative pointers to word XW of the character descriptor block for the character codes 0-377B.

FONTBASE+400B to FONTBASE+400B+EXTCNT-1:
These locations contain self-relative pointers to word XW of the character descriptor blocks for extensions, i.e., portions of characters which are wider than 16 bits.

FONTBASE+400B+EXTCNT to end:
Contains a number of character descriptor blocks of the form:

word 0 to word XW-1: The bit map for the character and surrounding spaces. The bit map does not include 0's at the top and bottom of the character, as the character will be vertically positioned by convert. The upper left-hand bit of the character is in the MSB of word 0.

word XW: If the character is less than 16 bits wide, this word contains (2*width)+1. If the character requires an extension, this word contains 2* a pseudo-character which is used as a character code to index the font. If this is the last extension block of a character, this word contains (2* the width of the final extension), rather than the total width. The pointer indexed by the character code points to this word.

word XW+1: In the left byte, HD. In the right byte, XH. HD is the number of scan lines to skip before displaying the character, XH is the height of the bit map.

The CONVERT instruction ORs the character bitmap into the display area. If the character does not require an extension, CONVERT skips, with the following information in the AC's:

AC0: unchanged

AC1: DBA and 17B
AC2: unchanged
AC3: the width of the character in bits

If the character requires an extension, convert returns normally. AC3 contains the pseudo-character code for the extension, and AC's 0-2 are as above.

RCLK (61003B) Read Clock:

The microcode maintains a 26 bit real time clock which is incremented by the memory refresh task at 38.08 microsecond intervals. The high order 16 bits of this clock are maintained in location RTC (430B) in page 1, the low order 10 bits are kept in the high order bits of R37. R37 is incremented by 100B each 38.08 microseconds. The low order 6 bits of R37 contain state information unrelated to the time.

RCLK loads AC0 with the contents of location RTC, and loads AC1 with the contents of R37. If the program then zeros bits 10-15 of AC1, it will have a clock value in units of .595 microseconds. AC0 alone is in units of 39 ms. The period of the clock is about 40 minutes.

SIO (61004B) Start I/O:

Start I/O is included to facilitate I/O control. It places the contents of AC0 on the processor bus and executes the STARTF function (F1=17B). By convention, bits of AC0 must be "1" in order to signal devices.

If bit 0 of AC0 is 1, and if an Ethernet board is plugged into the Alto, the machine will boot, just as if the "boot button" were pressed (see section 3.4 for a discussion of bootstrapping).

SIO also returns a result in AC0. If the Ethernet hardware is installed, the serial number and/or Ethernet number of the machine (0-377B) is loaded into AC0[8-15]. (On Alto I, the serial number and Ethernet number are equivalent; on Alto II, the value loaded into AC0 is the Ethernet number only.) Microcode installed after June 1976, which this manual describes, turns bit 0 of AC0 off. Microcode installed prior to June 1976 sets bit 0 of AC0; this is a quick way of acquiring the approximate vintage of a machine's microcode.

BLT (61005B) Block transfer:
BLKS (61006B) Block store:

These instructions use tight microcode loops to move a block of memory from one place to another (BLT) or to store a constant value into a block of memory (BLKS). Block transfer and block store take the following arguments:

AC0: Address of the first source word-1 (BLT), or data to be stored (BLKS).
AC1: Address of the last word of the destination area.
AC3: Negative word count.

Because these instructions are potentially time consuming, and keep their state in the AC's, they are interruptible. If an interrupt occurs, the PC is decremented by one, and the AC's contain the intermediate state. On return, the instruction continues. On completion, the AC's are:

AC0: Address of last source word+1 (BLT), or unchanged (BLKS).
AC1: Unchanged.
AC2: Unchanged.
AC3: 0.

The first word of the destination area (AC1 + AC3 + 1) is the first to be stored into.

SIT (61007B) Start interval timer:

The microcode implements an interval timer which has a resolution of 38 microseconds, and a maximum period of 10 bits. As the principal application for this timer is to do bit sampling for a serial EIA-RS232 compatible communications line, the timer is specialized for this purpose. It uses three dedicated locations in page 1:

ITTIME (525B): Contains the time at which the next timer interrupt should be caused. This is a 10 bit number, left justified in the 16 bit word. The low order 6 bits are not interpreted.

ITIBITS (423B): This word contains one or more bits specifying the channel or channels on which the timer interrupt is to occur.

ITQUAN (422B): When the interval timer interrupt is caused, the microcode stores a quantity in this location which depends on the mode.

The SIT instruction ORs the contents of AC0 into R37. The high 13 bits should be 0; the low order 2 bits determine the interval timer mode:

R37[14-15]

0 Off.

1 Normal mode. Every 38 microseconds, compare R37[0-9] with ITTIME[0-9]. If they are equal, cause an interrupt on the channel specified by ITIBITS. Store the current state of the EIA interface in ITQUAN, and set R37[14-15] to zero. The state of the EIA interface is bit 15 of location EIALOC (177701B) in page 377B. This bit is 0 if the line is spacing, 1 if it is marking.

2 Same as 0.

3 Every 38 microseconds, check the state of the EIA line. If the line is marking, do nothing. If the line is spacing, cause an interrupt on the channel specified by ITIBITS. Store the current value of R37 in ITQUAN, and set R37[14-15] to zero.

The intention is that a program which does EIA input can use mode 3 to monitor the line for the arrival of a character, and can then use mode 2 to time the center of each bit. By storing the state of the line, the interrupt latency can be as much as 1 bit time without errors.

JMPRM (61010B) Jump to RAM
RDRM (61011B) Read RAM
WTRM (61012B) Write RAM:

See Section 8.4.

VERS (61013B) Version:

AC0 is loaded with a number which is coded as follows:

bits 0-3	Alto engineering number (Alto I = 0 or 1, Alto II = 2)
bits 4-7	Alto build number.
bits 8-15	Version number of the microcode.

This instruction permits programs to know the differences among various kinds of Altos (e.g. Alto II's have special memory diagnosing features and additional emulator instructions to provide access to the diagnotics).

The two flavors of Alto maintain separate enumerations of microcode versions (see section 9 for some conventions).

DREAD (61015B) Double-word read (Alto II only):
  AC0←rv(AC3); AC1←rv(AC3 XOR 1)

DWRITE (61016B) Double-word write (Alto II only):
  rv(AC3)←AC0; rv(AC3 XOR 1)←AC1

DEXCH (61017B) Double-word exchange (Alto II only):
  t←rv(AC3); rv(AC3)←AC0; AC0←t
  t←rv(AC3 XOR 1); rv(AC3 XOR 1)←AC1; AC1←t

DIAGNOSE1 (61022B) Diagnostic instruction (Alto II only):
  This instruction starts a special double-word write cycle that also writes the Hamming code check bits.

  Hamming code ←AC2
  rv(AC3)←AC0; rv(AC3 XOR 1)←AC1

DIAGNOSE2 (61023B) Diagnostic instruction (Alto II only):
  This instruction writes the same memory location with two different values in quick succession:

  rv(AC3)←AC0
  rv(AC3)←AC0 AC1

BITBLT (61024B) Bit-boundary block transfer:
  An instruction for moving bits around in memory. It is particularly helpful for dealing with the display bit map.

Definitions

A bit map is a region of memory defined by **bca** and **bmr**, where **bca** is the *base core address* (starting location) and **bmr** is the *bit map raster width* in words; the number of scan lines is irrelevant for our purposes. (If both **bmr** and **bca** are even, then the bit map may be displayed on the screen using standard Alto facilities.)

A *block* is a rectangle within a bit map. It has four corners which need not fall on word boundaries. A block is described by 6 numbers:

Bit map's base core address (**bca**)
Bit map's width in words (**bmr**)
Block's Left **x** ("x offset" from first bit of scan-line)
Block's Top **y** ("y offset" from first scan-line)
Block's **width** (in bits)
Block's **height** (in scan-lines)

Example: A block is used to designate a sequence of bits in memory, such as a 16 wide 14 high region containing the bit pattern of a font character. In this case, bca points to the font character, bmw is 1, x and y are 0, width is 16, and height is 14.

Block Operations

The basic block operations operate by storing some bits into a "destination block." The source of these bits varies; often it is another block, the "source block." There are various functions that BITBLT can perform.

The *function* is encoded as the sum of two parts: *operation* + *sourcetype.* The *operation* codes (2 low-order bits) are:

0	Replace:	Destination Block ← *Source*
1	Paint:	Destination Block ← *Source* ior Destination

2	Invert:	Destination Block ← *Source* xor Destination
3	Erase:	Destination Block ← not *Source* and Destination

The *sourcetype* specifies how the *Source* as used in the above 4 operations is to be computed. The encodings (next 2 bits) are:

0	The *Source* is a block of a bit map
4	The *Source* is the complement of a block of a bit map
8	The *Source* is the logical "and" of a source block and the "gray block" (see below).
12	The *Source* is the "gray block."

The "gray block" is conceptually a block of infinite extent in which a pattern of dots is repeated. The pattern is specified by four words (Gray0 through Gray3). These give the patterns to write into the destination block where called for. The words will align with destination block word boundaries. While the BITBLT instruction takes care of going through these values appropriately, the table must be adjusted to eliminate *seams*. Specifically, if A B C D are the desired values of gray for lines 0 1 2 3 (mod 4), then two adjustments must be made:

Let Q = DTY + 1
If DTY $\leq$ STY, then exchange B and D and let Q = -(DTY+DH).
Rotate the pattern left by (Q rem 4)*4 bits.

When the source is a block of bit map, the width and height parameters of the block are not needed: the width and height of the destination block are also used as the width and height of the source block.

BITBLT requires the RAM to be present in order to use some S registers. If the RAM is not present, BITBLT will trap.

Calling sequence

The BITBLT function is invoked with:

AC1 = 0
AC2 = pointer to BBTable, which must be even.
Only AC2 is preserved by BITBLT.

The instruction is interruptable as it begins consideration of each scan line. If an interrupt happens, the state of its progress is saved in AC1 and the PC is backed up so that on return from the interrupt, BITBLT will finish its job.

The format of the BBTable is as follows:

Word	Name	Remarks
0	Function	=operation + sourcetype
1	unused	
2	DBCA	Destination bca
3	DBMR*	Destination bmr
4	DLX*	Destination left x
5	DTY*	Destination top y
6	DW*	Destination width
7	DH*	Destination height
8	SBCA	Source bca
9	SBMR	Source bmr
10	SLX*	Source left x
11	STY*	Source top y
12	Gray0	Four words to specify gray block...

13	Gray1
14	Gray2
15	Gray3

*Should all be positive values, although DH<0 or DW<0 will merely cause a NOP.

Timing Details

The microcode has roughly the following speed characteristics:

```
Horizontally, along one raster line (so to speak)
        Store constant          13 cycles/word
        Move block              23 cycles/word
                if skew not zero        add 6
                if source not zero      add 7
                1st or last word        add 13
                function not store      add 6

Vertical loop overhead (time to change raster lines)
        14-21 cycles, depending on source/dest alignment
                add 6 more if function uses gray

Initial setup overhead (time to start or resume from interrupt)
        approx 240 cycles

Total for a typical character, 8 wide by 14 high
        approx 1500 cycles
```

These are all in terms of Alto minor cycles and *do* include all memory wait time and *do not* include any degradation due to competing tasks, such as the display or disk. For typical characters on the Alto screen, BITBLT is about 2/3 the speed of CONVERT.

3.4 *Bootstrapping*

The emulator contains microcode for initializing the Alto in certain ways, and thereby "bootstrapping" a runnable program into the machine. A "boot," which is invoked either by pressing the small button at the rear of the keyboard or by executing an appropriate SIO instruction (see section 3.3), simply resets all micro-PC's to fixed initial values determined by their task numbers. Unless the Reset Mode Register specifies otherwise (see section 8.4), the emulator task is started in the PROM and performs a number of operations:

1. The current value of PC is stored in memory location 0. The accumulators are not altered during booting.

2. The display is cleared; i.e. rv(420B)←0.

3. Interrupts are disabled.

4. The first keyboard word (KBDAD, 177034B) is read to determine what sort of boot is to be done:

Disk Boot: If the <BS> key is not depressed, the microcode interprets any depressed keys reported in this keyboard word as a real disk address. If no keys are depressed, this results in a real disk address of 0.

The single disk sector at the given address is read: the 256 data words are read into locations 1 to 400B inclusive; the label is read into locations 402B to 411B inclusive. When the transfer is complete, PC←1, and the emulator is started. The disk status is stored in location 2, so the bootstrapping code must skip this location.

Ether Boot: If the <BS> key is depressed, the microcode anticpates breathing life into the Alto via the Ethernet. The Ethernet hardware is set up to read any packet with destination Alto number 377B into locations 1 to 400B inclusive. If a packet arrives with good status and with memory location 2 (i.e., the second word of the packet) equal to 602B (a "Breath-of-Life" packet), PC←3, and the emulator is started.

More information regarding boot loaders and boot file formats is found with Buildboot documentation in the Alto Subsystems Manual.

3.5 *Hardware*

There is a small amount of special hardware which is used exclusively by the emulator. This hardware is controlled by the task specific F2's, and by the ←DISP bus source.

The IR register is used to hold the current instruction. It is loaded with IR← (F2=14B). IR← also merges bus bits 0,5,6 and 7 into NEXT, which does a first level instruction dispatch. The high order bits of IR cannot be directly read, but the displacement field of IR (8 low order bits, sign extended), may be read with the ←DISP bus source.

There are two additional F2's which assist in instruction decoding, IDISP and ←ACSOURCE. The IDISP function (F2=15B) does a 16 way dispatch under control of a 256x4 PROM. The values are tabulated below:

Conditions		ORed onto NEXT
if	IR[1-2] = 0	then IR[3-4]
elseif	IR[1-2] = 1	then 4
elseif	IR[1-2] = 2	then 5
elseif	IR[4-7] = 0	then 1
elseif	IR[4-7] = 1	then 0
elseif	IR[4-7] = 6	then 16B
else		IR[4-7]

←ACSOURCE (F2=16B) has two roles. First, it replaces the two low order bits of the R select field with the complement of the SrcAC field of IR, (IR[1-2] XOR 3), allowing the emulator to address its accumulators (which are assigned to R0-R3). Second, a dispatch is performed:

Conditions		ORed onto NEXT
if	IR[0]=1	then IR[8-9] xor 3; the complement of the SH field of IR
elseif	IR[1-2] = 3	then IR[5]; the indirect bit of IR
elseif	IR[3-7] = 0	then 2
elseif	IR[3-7] = 1	then 5
elseif	IR[3-7] = 2	then 3
elseif	IR[3-7] = 3	then 6
elseif	IR[3-7] = 4	then 7
elseif	IR[3-7] = 11B	then 4
elseif	IR[3-7] = 12B	then 4
elseif	IR[3-7] = 16B	then 1
elseif	IR[3-7] = 37B	then 17B
else		16B

F2=13B, ACDEST, causes (IR[3-4] XOR 3) to be used as the low order two bits of the RSELECT field. This addresses the accumulators from the destination field of the instruction. The selected register may be loaded or read.

The emulator has two additional bits of state, the SKIP and CARRY flip flops. CARRY is identical to the Nova carry bit, and is set or cleared as appropriate when the DNS← (do Nova shifts) function is executed. DNS also addresses R from (IR[3-4] XOR 3), and sets the SKIP flip flop if appropriate. The PC is incremented by 1 at the beginning of the next emulated instruction if SKIP is set, using ALUF 13B. IR← clears SKIP.

Note that the functions which replace the low bits of RSELECT with IR affect only the selection of R; they do not affect the address supplied to the constant ROM.

The two additional emulator specific functions, BUSODD and MAGIC, are not peculiar to Nova emulation, but are included for their general usefulness. BUSODD merges BUS[15] into NEXT[9], and MAGIC is applied in conjunction with LSH and RSH to allow double length shifts. It shifts the high order bit of T into the low order bit of R on left shifts, and shifts the low order bit of T into the high order bit of R on right shifts.

The STARTF function (F1=17B) is used by the SIO instruction, and is used to define commands for I/O hardware, including the Ethernet.

4.0 *DISPLAY CONTROLLER*

4.1 *Programming Characteristics*

The display controller handles transfers between the main memory and the CRT. The CRT is a standard 875 line raster-scanned TV monitor, refreshed at 60 fields per second from a bit map in main memory. The CRT contains 606 points horizontally, and 808 points vertically, or 489,648 points total.

The basic way in which information is presented on the display is by fetching a series of words from Alto main memory, and serially extracting bits to become the video signal. Therefore, 38 16-bit words are required to represent each scan line; 30704 words are required to fill the screen.

The display is defined by one or more display control blocks in main memory. Control blocks (DCB's) are linked together starting at location DASTART(420B) in page 1:

DASTART:	Pointer to word 0 of the first (top on the screen) DCB, or 0 if display is off.
DASTART+1:	Vertical field interupt bit mask. Every 1/60 second, this word is OR'ed into NWW to cause interrupts.

Display control blocks must begin located at even addresses in memory, and have the following format:

DCB: Pointer to next DCB, or 0 if this is the last.

DCB+1: Bit 0: 0=high resolution mode
1=low resolution mode
Bit 1: 0=black on white background presentation
1=white on black background
Bits 2-7 (HTAB): On each scan line of this block, wait 16*HTAB bits before displaying information from memory.
Bits 8-15 (NWRDS): Each scan line in this block is defined by NWRDS 16 bit words. (NWRDS must be even). In order to skip space on the screen without requiring bit-map, set NWRDS to 0.

DCB+2 (SA): Bit map starting address, which must be even.

DCB+3 (SLC): This block defines 2*SLC scan lines, SLC in each field.

At the start of each field, the display controller inspects DASTART and DASTART+1. An interrupt is initiated on the channel(s) specified by the bit(s) in DASTART+1. The controller then executes each DCB sequentially until the display list or the field ends. At normal resolution, the first scan line of the first (even) field of a block is taken from location SA to SA+NWRDS-1, the first scan line of the odd field is taken from locations SA+NWRDS to SA+2*NWRDS-1. During each field, the bit map address is incremented by NWRDS between each scan line. Thus, although the display is interlaced, its representation in memory is not. In low resolution mode, the video is generated at half speed, and each scan line is displayed twice (once in each field). During each field, the bit map address is not incremented between the display of adjacent scan lines. This makes the format of the bit map in memory identical for both modes--only the size of the presentation is affected by the mode.

4.2 *Hardware*

The display controller consists of a sync generator, a data buffer and serializing shift register, and three microcode tasks which control data handling and communicate with the Alto program. The hardware is shown in block form in Figure 5. The 16 word buffer is loaded from the Alto bus with the DDR← function (F2=10B, specific to the display word task DWT). The purpose of the intermediate buffer is to synchronize data transfers between the main buffer, which is synchronous with the 170ns. master clock, and the shift register, which is clocked with an asynchronous bit clock. The sync generator provides this clock and the vertical horizontal synchronization signals required by the monitor.

The bit clock is disabled by vertical and horizontal blanking, and its rate can be set by the microcode to either 50 or 100 ns. by the function SETMODE (F2=11B, specific to the display horizontal task DHT). This function examines the two high order bits of the processor bus. If bit 0=1, the bit clock rate is set to 100ns period (at the start of the next scan line), and a 1 is merged into NEXT[9]. SETMODE also latches bit 1 of the processor bus and uses the value to control the polarity of the video output. A third function, EVENFIELD (F2=10B, specific to DHT and to the display vertical task DVT), merges a 1 into NEXT[9] if the display is in the even field.

The display control hardware also generates wakeup requests to the microprocessor tasking hardware. The vertical task DVT is awakened once per field, at the beginning of vertical retrace. The display horizontal task is awakened once at the beginning of each field, and thereafter whenever the display word task blocks. DHT can block itself, in which case neither it nor the word task can be awakened until the start of the next field. The wakeup request for the display word task (DWT) is controlled by the state of the 16 word buffer. If DWT has not executed a BLOCK, if DHT is not blocked, and if the buffer is not full, DWT wakeups are generated. The hardware sets the buffer empty and clears the DWT block flip-flop at the beginning of horizontal retrace for every scan line.

4.3 *Display Controller Microcode*

The display controller microcode is divided into three tasks. The highest priority task is DVT, the display vertical task, the next is DHT, the horizontal task, and the third is DWT. The display controller uses 6 registers in R:

CBA:	Holds the address of the currently active DCB+1.
AECL:	Holds the address of the end of the currently active scan line's bit map in main memory.
SLC:	Holds the number of scan lines remaining in the currently active DCB.
HTAB:	Holds the number of tab words remaining on the current scan line.
DWA:	Holds the address of the bit map doubleword currently being fetched for transmission to the hardware buffer.
MTEMP:	Is a temporary cell.

The vertical task initializes the controller by setting SLC to 0 and CBA to DASTART+1. It also merges the contents of DASTART+1 into NWW, which will cause an interrupt if the specified channel is active. DVT also sets up information required for the cursor (see below), TASKs and becomes inactive until the next field.

DHT starts by initiating a fetch to the word addressed by CBA. It checks SLC, and if it is zero, the controller is finished with the current DCB, and the link word of the DCB is fetched. If this word is non-zero, it replaces CBA and processing of a new block is begun. If the link word is zero, DHT blocks until the start of the next field.

If the check of SLC indicates that more scan lines remain in the current DCB, SLC is decremented by one and the fetch of (CBA) is used to obtain the second word of the DCB, rather than the link word. The contents of this word are used to set the display mode and polarity, and the tab count is extracted and put into HTAB. NWRDS is extracted, and used to increment DWA and AECL by the appropriate amount, depending on the mode and field. All the registers required by DWT have now been set up, and DHT TASKs and becomes inactive until DWT blocks.

If a new DCB is required, DHT fetches all four words of the new DCB, and initializes all the registers. During all scan lines of a DCB except the first, DHT only accesses the first doubleword of the block.

DWT has the sole task of transferring words from memory to the hardware. When it first awakens during horizontal retrace, it checks HTAB. If it is non-zero, it enters a loop which outputs HTAB 0's to the display. When HTAB is zero, a second loop is entered which fetches a doubleword from the location specified by DWA. DWA is compared with AECL, and if they are equal, DWT blocks until the next scan line. DWA is incremented by 2, in preparation for the fetch of the next doubleword. If DWA≠AECL, DWT continues to supply words to the buffer whenever it becomes non-full.

4.4 *Cursor*

Because of the difficulty of inserting a cursor at the appropriate place in the display bit map at reasonable speed, a hardware cursor is included in the Alto. The cursor consists of an arbitrary 16x16 bit patch, which is merged with the video at the appropriate time. The bit map for the cursor is contained in 16 words starting at location CURMAP(431B) in page one, and the x,y coordinates of the cursor are specified by location CURLOC (426B) and CURLOC+1 (427B) in page one. The coordinate origin for the cursor is the upper left hand corner of the screen. The cursor presentation is unaffected by changes in display resolution. Its polarity is that of the current DCB, or the last DCB processed if it is located on an area of the screen not defined by a DCB, The cursor may be removed from view in a number of ways. The most efficient in terms of processing time is to set the x coordinate to -1.

The cursor hardware consists of a 16 bit shift register which holds the information to be displayed on the current scan line, and a counter which is incremented by the bit clock, and determines the x coordinate at which the shift register begins shifting.

The hardware is loaded during horizontal retrace by the cursor task microcode, which simply copies the x coordinate and bit map segment from the R memory into the hardware.

The values of x and the bit map are set up in R by a section of the memory refresh task, whose wakeup and priority are arranged so that it runs during every scan line after DWT has done all necessary output and DHT has set up the information required by DWT for the next scan line. MRT checks the current y position of the display, and if it is in the range in which the cursor should be displayed, fetches the appropriate bit map segment from CURMAP. When the cursor y position is exceeded by the display, a flag is set in MRT to disable further processing. The x and y coordinates of the cursor are fetched from CURLOC and CURLOC+1 at the beginning of each display field by a section of the display vertical task microcode.

Cursor processing is distributed as it is to minimize the amount of processing which must be done during the monitor's horizontal retrace time. This time is approximately 6 microsec, and it must include the worst case latency imposed by tasks at lower priority than the display, plus the worst case disk word processing time (the disk word task is at higher priority than the display), plus the time necessary for DWT to partially fill the display buffer, plus cursor processing time.

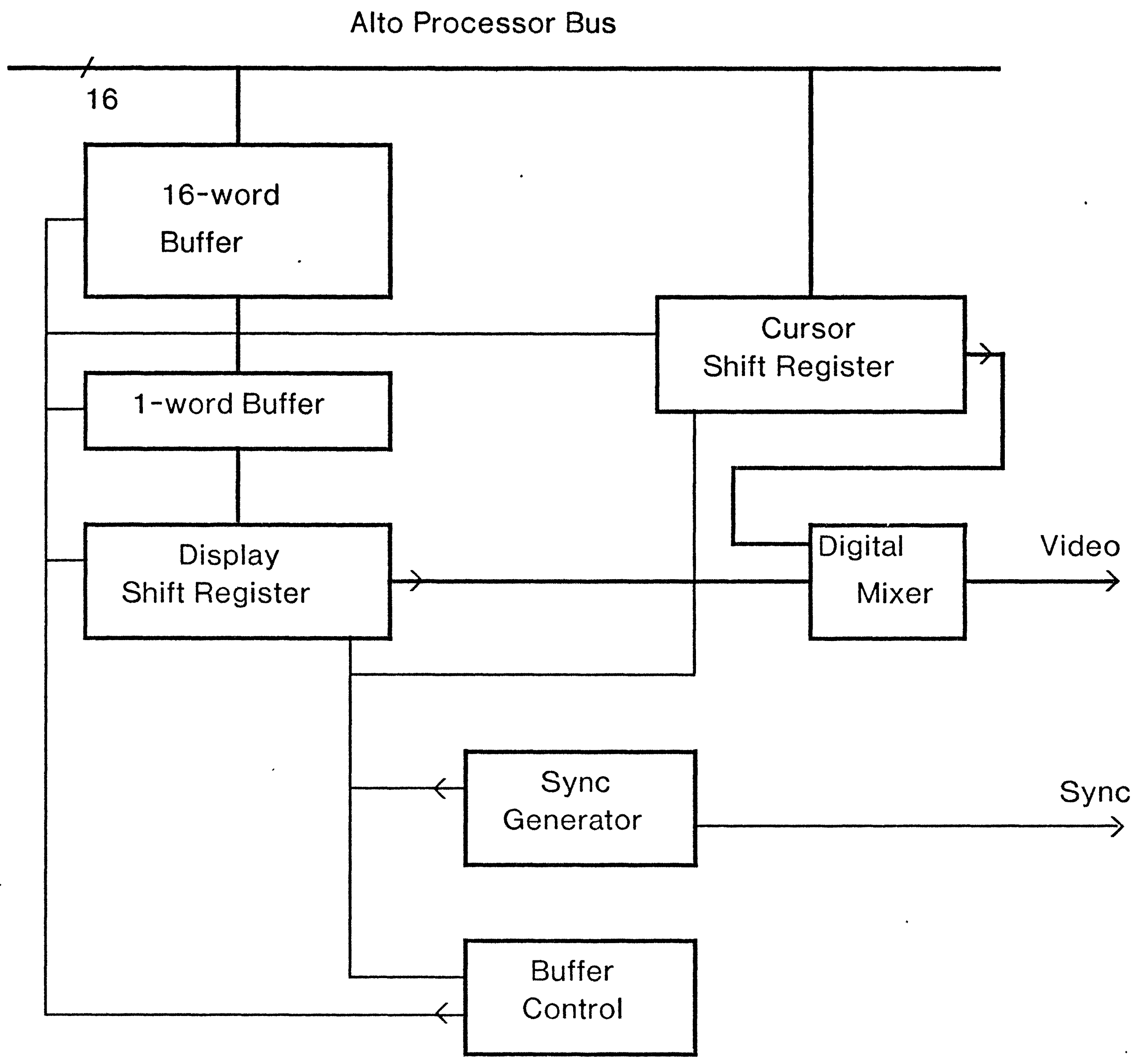

Figure 5 -- Display Control

5.0 *MISCELLANEOUS PERIPHERALS*

The Alto can have a number of slow peripherals which appear to programs as memory locations in the range 177000-177777B. The standard peripherals are described here and some non-standard devices are described in Appendix C. which describes each word in detail. In the reserved memory locations associated with keyboard, mouse, keyset and Diablo printer input, a more positive logic value reads as a 1.

5.1 *Keyboard*

The Alto keyboard contains 61 keys. It appears to the program as four 16 bit words in 4 adjacent locations starting at KBDAD (177034B). Depressed keys correspond to 0's in memory, idle keys correspond to 1's. Figure 6 shows layouts of the Alto I and Alto II keyboards, including keytops and the word number, bit number corresponding to each key.

ALTO I KEYBOARD

Bit	Word KBDAD	Word KBDAD+1	Word KBDAD+2	Word KBDAD+3
0	5	3	1	R
1	4	2	ESC	T
2	6	W	TAB	G
3	E	Q	F	Y
4	7	S	CTRL	H
5	D	A	C	8
6	U	9	J	N
7	V	I	B	M
8	0(zero)	X	Z	LOCK
9	K	O	<shift-left>	SPACE
10	-	L	.	[
11	P	,	;	+
12	/	"	RETURN	<shift-right>
13	\	]	←	<blank-bottom>
14	LF	<blank-middle>	DEL	xxx
15	BS	<blank-top>	xxx	xxx

ALTO II KEYBOARD

Bit	Word KBDAD	Word KBDAD+1	Word KBDAD+2	Word KBDAD+3
0	5	3	1	R
1	4	2	ESC	T
2	6	W	TAB	G
3	E	Q	F	Y
4	7	S	CTRL	H
5	D	A	C	8
6	U	9	J	N
7	V	I	B	M
8	0(zero)	X	Z	LOCK
9	K	O	<shift-left>	SPACE
10	-	L	.	[
11	P	,	;	+
12	/	"	RETURN	<shift-right>
13	\(FR2)	]	←(FR3)	FR1
14	LF(FL2)	FR4	DEL(FL1)	FL4
15	BS	BW	FL3	FL5

Figure 6

5.2 *Mouse*

The mouse is a hand-held pointing device which contains two encoders which digitize its position as it is rolled over a table-top. It also has three buttons which may be read as the three low-order bits of memory location UTILIN (177030B), in the manner of the keyboard. The bit/button correspondences in UTILIN are:

Bit 13:	Top or Left Button
Bit 14:	Bottom or Right Button
Bit 15:	Middle Button

The mouse coordinates are maintained by the MRT microcode in locations MOUSELOC(424B)=x and MOUSELOC+1(425B)=y in page one of the Alto memory. These coordinates are relative, i.e., the hardware only increments and decrements them. The resolution of the mouse is approximately 100 points per inch.

5.3 *Keyset*

The standard Alto includes a five-finger keyset which is presented to the program as 5 bits of memory location UTILIN (177030B), similar to the keyboard. Where key 0 is the left-most key on the keyset and key 4 the right-most, the bit/key correspondences in UTILIN are:

Bit 8:	Key 0 (left-most)
Bit 9:	Key 1
Bit 10:	Key 2
Bit 11:	Key 3
Bit 12:	Key 4 (right-most)

5.4 *Diablo Printer*

The Alto includes an interface to a Diablo HyType printer. The printer uses a portion of one memory location to report status, and another location into which the Alto program can store to send signals to the printer. None of the timing signals required by the printer are generated automatically--all must be program generated. For detailed information on the printer, refer to the Diablo manual.

The Diablo printer is accessed and controlled through two locations in high memory, an input status word and an output control word. The relevant bits of these two words are as follows:

Location UTILIN (177030B):

Bit 0: Paper ready bit. 0 when the printer is ready for a paper scrolling operation.

Bit 1: Printer check bit. Should the printer find itself in an abnormal state, it sets this bit to 0.

Bit 2: Unused.

Bit 3: Daisy ready bit. 0 when the printer is ready to print a character.

Bit 4: Carriage ready bit. 0 when the printer is ready for horizontal positioning.

Bit 5: Ready bit. Both this bit and the appropriate other ready bit (carriage, daisy, etc.) must be 0 before attempting any output operation.

Bit 6: Setting of "memory configuration switch", described in *Parity Error Detection* below.

Bit 7: Unused.

Bits 8-15: Used by mouse and keyset keys as set out above.

Location UTILOUT (177016B):

Several of the output operations are invoked by "toggling" a bit in the output status word. To toggle a bit, set it first to 1, then back to 0 immediately. In this memory location, a 1 writes as a more negative logic value.

Bit 0: Paper strobe bit. Toggling this bit causes a paper scrolling operation.

Bit 1: Restore bit. Toggling this bit resets the printer.

Bit 2: Ribbon bit. When this bit is 1 the ribbon is up (in printing position); when 0, it is down.

Bit 3: Daisy strobe bit. Toggling this bit causes a character to be printed.

Bit 4: Carriage strobe bit. Toggling this bit causes a horizontal positioning operation.

Bits 5-15: Argument to various output operations:

1. Printing characters. When the daisy bit is toggled bits 9-15 of this field are interpreted as an ASCII character code to be printed (it should be noted that all codes less than 40B print as lower case "w").

2. For paper and carriage operations the field is interpreted as a signed displacement (-1024 to +1023), in units of 1/48 inch for paper and 1/60 inch for carriage. Positive is down or to the right, negative up or to the left.

The printer is initialized by toggling the restore bit, then waiting for all ready bits to be 0. A typical output sequence, say printing a character, involves examining the check bit for abnormal status, waiting for both the ready and daisy ready bits to be 0, then writing in the printer output location the character code, the character code ORed with the daisy strobe bit, and the unmodified code again.

The device behaves more or less like a plotter, i.e. you must explicitly position each character in software; a print operation does not affect the position of either the carriage or the paper. All coordinates in paper or carriage operations are relative; the device does not know its absolute position. Again, you must keep track of this in software.

5.5 *Analog Board*

(Reserved for words from Ed McCreight)

5.6 *Parity Error Detection*

The detection and reporting of parity errors is accomplished somewhat differently on Alto I and Alto II. In both machines, the processing of errors is undertaken by the highest priority microtask, which is invoked very soon after an error occurs. The microtask reports a parity error by causing an interrupt on the highest-priority emulator interrupt channel, i.e. by oring into NWW bit 15. Bear in mind that parity errors can be generated by memory references undertaken by any microtask; as a result, it may be some time between the occurrence of the error and the next execution of the emulator task and consequent servicing of the interrupt.

Both Alto I and Alto II have a switch mounted just below the disk drive that affects the corresondance between addresses in the range 0-77777B and memory boards. Flipping the switch interchanges the roles of the first two 16K parts of memory. If a parity error at a known address is to be traced to a particular memory board, the setting of this switch must be known. All Alto II's and most Alto I's (a slight modification is necessary) report the switch setting in bit 6 of memory location UTILIN (177030B). The bit is "??" if the switch is in the normal ("left") position.

When a parity error happens, the parity task stores the contents of various R registers into some page 1 reserved locations. Unfortunately, the information recorded by the parity task is not sufficient to determine precisely where the parity error occurred. The registers saved in page 1 when on error are given below. The intent of the collection is to save values of the R registers most likely to be used as a source of memory addresses.

DCBR (614B)	Disk control block fetch pointer
KNMAR (615B)	Disk word fetch/store pointer
DWA (616B)	Display word fetch address
CBA (617B)	Display control block fetch address
PC (620B)	Current program counter in the emulator
SAD (621B)	Temporary register for indirection in emulator

The Alto II memory contains circuitry for correcting single-bit errors and detecting double-bit errors. The logic expects a good deal of set-up and in turn reports copious error information. Interaction with the error control is effected through three memory locations (177024B, 177025B and 177026B):

Memory Error Address Register (MEAR = 177024B). This register holds the address of the first error since the error status was last read. If no error has occurred, this register reports the address of the last memory access. Note that MEAR is set whenever an *error of any kind* is detected.

Memory Error Status Register (MESR = 177025B). This register reports specifics of the first error that occurred since MESR was last read. Reading the register resets the error logic and enables it to detect a new error. The bits in MESR are (all bits are "low true," i.e. if the bit is 0, the condition is true):

Bits 0-5	Hamming code reported from error
Bit 6	Parity OK
Bit 7	Memory parity bit
Bit 8-13	Syndrome bits
Bits 14-15	Spare

Memory Error Control Register (MECR = 177026B). Storing into this register is the means for controlling the memory error logic. Bits are "low true," i.e. a 0 bit enables the condition. This register is set to all ones (disable all interrupts) when the Alto is bootstrapped and when the parity error task first detects an error. When an error has occurred, the MEAR and MESR should be read before setting the MECR.

Bits 0-3	Spare
Bits 4-10	Test Hamming code (used only for special diagnostics)
Bit 11	Test mode (used only for special diagnostics)
Bit 12	Cause interrupt on single-bit errors
Bit 13	Cause interrupt on double-bit errors
Bit 14	Do not use error correction
Bit 15	Spare

Note that bits 12 and 13 govern only the initiation of interrupts; the MEAR and MESR hold information about the first error that occurs after reading MESR regardless of what kind of errors are to cause interrupts.

6.0 *DISK AND CONTROLLER*

The disk controller is designed to accommodate one of a variety of DIABLO disk drives, including models 31 and 44. Each drive accommodates one or two disks. Each disk has two heads, one per side. Information is recorded on each disk in a 12-sector format on each of up to 406 (depending on the disk model) radial track positions. Thus, each disk contains up to 9744 recording positions (2 heads x 12 sectors x 406 track positions). Figure 7 tabulates various useful information about the performance of the disk drives.

Device	Diablo 31	Diablo 44	
Number of drives/Alto	1 or 2	1	
Number of packs	1 removable	1 removable 1 fixed	
Number of cylinders	203	406	
Tracks/cylinder/pack	2	same	
Sectors per track	12	same	
Words per sector	2 header 8 label 256 data	same	
Data words/track	3072	3072	
Sectors/pack	4872	9744	
Rotation time	40	25	ms
Seek time (approx.)	15+8.6*sqrt(dt)	8+3*sqrt(dt)	ms
min-avg-max	15-70-135	8-30-68	ms
Average access to 1 megabyte	80	32 (using both packs)	ms
Transfer rate:			
peak/avg	1.6/1.22	2.5/1.9	MHz
peak/avg	10.2/13	6.7/8.	ns/word
per sector	3.3	2.1	ms
for full display	.460	.266	sec
for 64k memory	1.03	.6	sec
whole drive	19.3	44(both packs)	sec

Figure 7

The disk controller records three independent data blocks in each recording position. The first is two words long, and is intended to include the address of the recording position. This block is called the *Header block.* The second block is eight words long, and is called the *Label block.* The third block is 256 words long, and is the *Data block.* Each block may be independently read, written, or checked, except that writing, once begun, must continue until the end of the recording position.

When a block is checked, information on the disk is compared word for word with a specified block of main memory. During checking, a main memory word containing 0 has special significance. When this word is encountered, the matching word read from the disk is stored in its place and does not take part in the check. This feature permits a combination of reading and checking to occur in the same block. (It also has the drawback of making it impossible to use the disk controller to check for words containing 0 on the disk.)

The Alto program communicates with the disk controller via a four-word block of main memory beginning at location KBLK (521B). The first word is interpreted as a pointer to a chain of disk command blocks. If it contains 0, the disk controller will remain idle. Otherwise, the disk controller will commence execution of the command contained in the first disk command block. When a command is completed successfully, the disk controller stores in KBLK a pointer to the next command in the chain and the cycle repeats. If a command terminates in error, a 0 is immediately stored in KBLK and the disk controller idles. At the beginning of each sector, status information, including the number of the current sector, is stored in KBLK+1. This can be used by the Alto program to sense the readiness of the disk and to schedule disk transfers, for example. When the disk controller begins executing a command, it stores the disk address of that command in KBLK+2. This information is later used by the disk controller to decide whether seek operations or disk switches are necessary. It can be used by the Alto program for scheduling disk arm motion. If the Alto program stores an illegal disk address (like -1) in this word, the disk controller will perform a seek at the beginning of the next disk operation. (This is useful, for example, when the operating system wants to force a restore operation.) The disk controller also communicates with the Alto program by interrupts (see Section 3.2). At the beginning of each sector interrupts are initiated on the channels specified by the bits in KBLK+3.

KBLK (521B):	Pointer to first disk command block.
KBLK+1 (522B):	Status at beginning of current sector.
KBLK+2 (523B):	Disk address of most-recently started disk command.
KBLK+3 (524B):	Sector interrupt bit mask.

A disk command block is a ten-word block of memory which describes a disk transfer operation to the disk controller, and which is also used by the controller to record the status of that operation. The first word is a pointer to the next disk command block in this chain. A 0 means that this is the last disk command block in the chain. When the command is complete, the disk controller stores its status in the second word. The third word contains the command itself, telling the disk controller what to do. The fourth word contains a pointer to the block of memory from/to which the header block will be transferred. The fifth word contains a similar pointer for the label block. The sixth word contains a similar pointer for the data block.

The seventh and eighth words of the disk command block control the initiation of interrupts when the command block is finished. If the command terminates without error, interrupts are initiated on the channels specified by the bits in DCB+6. However, if the command terminates with an error, the bits in DCB+7 are used instead.

The ninth word is unused by the disk controller, and may be used by the Alto program to facilitate chained disk operations. The tenth word contains the disk address at which the current operation is to take place.

DCB:	Pointer to next command block.
DCB+1:	Status.
DCB+2:	Command.
DCB+3:	Header block pointer.
DCB+4:	Label block pointer.
DCB+5:	Data pointer.
DCB+6:	Command complete no-error interrupt bit mask.
DCB+7:	Command complete error interrupt bit mask.
DCB+8:	Currently unused.
DCB+9:	Disk address.

A disk address word A contains the following fields:

Field	*Range*	*Significance*
A[0-3]	0-13B	Sector number.
A[4-12]	0-625B (Model 44) 0-312B (Model 31)	Track number.
A[13]	0-1	Head number.
A[14]	0-1	Disk number (see also C[15]). 0 is removable pack on Model 44. 1 is optional second Model 31 drive.
A[15]	0-1	0 normally. 1 if track 0 is to be addressed via a hardware "restore" operation.

A disk command word C contains the following fields:

Field	*Range*	*Significance*
C[0-7]	110B	Checked to verify that this is a valid disk command.
C[8-9]	0-3	0 if Header block to be read. 1 if Header block to be checked. 2 or 3 if Header block to be written.
C[10-11]	0-3	0 if Label block to be read. 1 if Label block to be checked. 2 or 3 if Label block to be written.
C[12-13]	0-3	0 if Data block to be read. 1 if Data block to be checked. 2 or 3 if Data block to be written.
C[14]	0-1	0 normally. 1 if the command is to terminate immediately after the correct track position is reached (before any data is transferred).
C[15]	0-1	XOR'ed with A[14] to yield hardware disk number.

A disk status word S has the following fields:

Field	*Values*	*Significance*
S[0-3]	0-13B	Current sector number.
S[4-7]	17B	One can tell whether status has been stored by setting this field initially to 0 and then checking for non-zero.
S[8]	0-1	1 means seek failed, possibly due to illegal track address.
S[9]	0-1	1 means seek in progress.
S[10]	0-1	1 means disk unit not ready.
S[11]	0-1	1 means data or sector processing was late during the last sector. Data and current sector number unreliable.
S[12]	0-1	1 means disk interface was not transferring data last sector.
S[13]	0-1	1 means checksum error. Command allowed to proceed.
S[14-15]	0-3	0 means command completed correctly. 1 means hardware error (see S[8-11]) or sector overflow. 2 means check error. Command terminated instantly. 3 means disk command specified illegal sector.

Several clever programming tricks have been suggested to drive the disk controller. For an initial program load, KBLK should be set to point to a disk command block representing a read into location STRT. Before setting KBLK, the Alto program should put a JMP STRT instruction in STRT; afterward it should jump to STRT. The disk controller transfers data *downward*, from high to low addresses, so that when location STRT is changed the reading of the block is complete. (See section 3.4 on the standard bootstrap loading microcode.)

Another trick is to chain disk reads through their label blocks. That is, the label block for sector n contains part of the disk command block for reading sector n+l, and so on.

6.1 Disk Controller Implementation

The following walk-through of an average day in the life of the standard disk controller is not intended for the casual reader, but rather as a roadmap to ease the pain of learning the innermost workings of the controller. If you really want to benefit from this next section, you should have a copy of the standard disk controller microcode and logic drawings close at hand.

The disk controller consists of a modest amount of hardware and two microcode tasks (the sector task and the word task). Communication with the emulator is via the four special main memory words, the disk command blocks, and the interrupts described earlier. In following few paragraphs the actions of the standard disk controller microcode are described. Occasionally it may be unclear whether the actor is microcode or hardware. Referring to microcode listings and/or logic drawings will resolve any such questions.

The sector task is awakened by a sector signal from the disk. When awakened, it stores the status of the disk and controller in the special disk status word (KBLK+l). In addition, if this sector signal terminates a disk command (for example, a data transfer during the previous sector), the status of the disk and controller are stored in the status word of the disk command block containing the terminated command, and the command block pointer (KBLK) is advanced. If a command was terminated with an error, KBLK (DCB pointer) is set to 0 and KBLK+2 (current disk address) is set to -l. The effect of this is to cause the disk controller to abandon the current disk command chain and to forget where the disk arm is positioned.

Next, the sector task considers the first command on the disk command block chain (by using KBLK). If there is none, or if the disk unit is not ready to accept a command, the sector task goes to sleep until the next sector pulse. Otherwise, the sector specified in the new command is verified to be less than 13. Then, the disk and cylinder specified in the new command are compared with those stored in KBLK+2 (current disk address), and then the new disk address is stored in KBLK+2 and in the disk controller hardware. Part of the new command is also stored in the hardware. If the comparison is unequal, a seek is initiated and the sector task goes to sleep until the next sector pulse.

If the comparison was equal, the SEEKOK hardware flag is tested. If that is OK, then the no-transfer bit of the disk command (bit 14 of the command word of the current disk command block) is tested to see whether a data transfer is required. If not, the sector task goes to sleep such that the command will terminate at the next sector pulse. If a data transfer is required, the specified sector number and the current disk sector number are compared. If unequal, the sector task goes to sleep until the next sector pulse. If sector numbers are equal, awakening of the word task is enabled and the sector task goes to sleep such that the command will terminate at the next sector pulse.

The word task awakens when a word has been processed by the disk controller hardware and the word task has been enabled by the sector task. First, a starting delay is computed, based on whether the current record is to be read or written. Second, control is dispatched based on the current record number. A record length and main memory starting address are computed based on the record number. In addition, special starting delays are computed for record number 0. The disk unit is set into the delay mode appropriate for the operation (read/write) and the word task goes to sleep the appropriate number of times.

Then a sync word is written (if writing) or awaited (if reading). Finally the main transfer loop is entered. Here the word count is decremented, a memory operation is started, and control is dispatched on the transfer type. If read, the disk word is stored in memory. If write, the memory word is sent to the disk. If check, the memory word is compared with 0. If non-zero, the disk and memory words are compared. An unequal compare here terminates this sector's operation with an error immediately. If the memory word is 0, it is replaced by the disk word. In any case, the checksum is updated and control returns to the main transfer loop. Due to the ALU functions available, the main transfer loop moves in sequence from high to low main memory addresses.

After the wordcount reaches 0, the checksum is written or checked. A checksum error will be noted in the status word, but will not terminate this sector's operation. A finishing delay is computed, based on the current operation, the disk unit is set into a delay mode appropriate to the operation, and the delay happens. Finally, all disk transfers are shut off, the record number is incremented, and control returns to the beginning of the word task.

To accomplish all this, the disk controller hardware communicates with the microprocessor in four ways: first, by task wakeup signals for the sector and word tasks; second, by five task-specific F2's which modify the next microinstruction address; third, by seven task-specific F1's, four of which activitate bus destination registers, and the remaining three of which provide useful pulses; and fourth, by two task-specific BS's. The following tables describe the effects of these.

F1 Value	*Name*	*Effect*
17B	KDATA←	The KDATA register is loaded from BUS[0-15]. This register is the data output register to the disk, and is also used to hold the disk address during KADR← and seek commands. When used as a disk address it has the format of word A in section 6.0 above.
16B	KADR←	This causes the KADR register to be loaded from BUS[8-14]. This register has the format of word C in section 6.0 above. In addition, it causes the head address bit to be loaded from KDATA[13].
15B	KCOM←	This causes the KCOM register to be loaded from BUS[1-5]. The KCOM register has the following interpretation: (1) XFEROFF = 1 Inhibits data transmission to/from the disk. (2) WDINHIB = 1 Prevents the disk word task from awakening. (3) BCLKSRC 1: Take bit clock from disk input or crystal clock, as appropriate. 0: Forces use of crystal clock. (4) WFFO 0: Holds the disk bit counter at -1 until a 1-bit is read. 1: Allows the bit counter to proceed normally. (5) SENDADR 1: Causes KDATA[4-12] and KDATA[15] to be transmitted to disk unit as track address. 0: Inhibits such transmission.

14B	CLRSTAT	Causes all error latches in disk controller hardware to reset, clears KSTAT[13].
13B	INCRECNO	Advances the shift registers holding the KADR register so that they present the number and read/write/check status of the next record to the hardware.
12B	KSTAT←	KSTAT[12-15] are loaded from BUS[12-15]. (Actually, BUS[13] is ORed into KSTAT[13].) This enables the microcode to enter conditions it detects into the status register.
11B	STROBE	Initiates a disk seek operation. The KDATA register must have been loaded previously, and the SENDADR bit of the KCOMM register previously set to 1.

F2 Value	*Name*	*Effect*
10B	INIT	NEXT←NEXT OR (*if* WDTASKACT AND WDINIT) *then* 37B *else* 0)
11B	RWC	NEXT←NEXT OR (*if* current record to be written *then* 3 *elseif* current record to be checked *then* 2 *else* 0)
12B	RECNO	NEXT←NEXT OR MAP (current record number) where MAP(0) = 0 MAP(1) = 2 MAP(2) = 3 MAP(3) = 1
13B	XFRDAT	NEXT←NEXT OR (*if* current command wants data transfer *then* 1 *else* 0)
14B	SWRNRDY	NEXT←NEXT OR (*if* disk not ready to accept command *then* 1 *else* 0)
15B	NFER	NEXT←NEXT OR (*if* fatal error in latches *then* 0 *else* 1).
16B	STROBON	NEXT←NEXT OR (*if* seek strobe still on *then* 1 *else* 0).

BS Value	*Name*	*Effect*
3	←KSTAT	The KSTAT register is placed on BUS. It has the format of a disk status word.
4	←KDATA	The disk input data register is placed on BUS.

A feature of interest mostly to the diagnostic microcode writer is that if one reads the disk input data register while writing, what should appear is delayed written data correctly aligned on word boundaries. This is a painless way of checking most of the data paths in the disk controller hardware.

7.0 *ETHERNET*

The Ethernet is the principal means of communications between an Alto and the outside world. It is a broadcast, multi-drop, packet-switching, bit serial, digital communications network. Our object is to design a communication system which can grow smoothly to accomodate several buildings full of personal computers and the facilities needed for their support. In concrete terms, to connect up to 256 nodes, separated by as much as 1 kilometer, with a 2.94 megabits/sec channel. Like the computing stations to be connected, the communications facility had to be inexpensive. We chose to distribute control of the communications facility among the communicating computers to eliminate the reliability problems of an active central controller, to avoid a bottleneck in a system rich in parallelism, and to reduce the fixed costs which make small systems uneconomical.

The Ethernet is intended to be an efficient, low-level packet transport mechanism which gives its best efforts to delivering packets, but *it is not error free.* Even when transmitted without source-detected interference, a packet may still not reach its destination without error; thus, *packets are delivered only with high probability.* Stations requiring a residual error rate lower than that provided by this bare packet transport mechanism must follow mutually agreed upon packet protocols.

Alto Ethernets come in three pieces: the transceiver, the interface, and the microcode. The transceiver is a small device which taps into the passing Ether inserting and extracting bits under the control of the interface while disturbing the Ether as little as possible. The same device is used to connect all types of Ethernet interfaces to the Ether, so the transceiver design is not specific to the Alto, and will not be described here. Before describing the internals of the interface and microcode, we present their programming characteristics.

7.1 *Programming Characteristics*

Programs communicate with the interface and the microcode via the emulator instruction SIO and 9 reserved locations in page 1. Word counts, buffer addresses, etc. are put in the appropriate locations and then SIO is executed with an Ethernet command in AC0.

The special page 1 memory locations and their functions are:

EPLoc = 600b:	Post location. Microcode and interface status information is posted in this location when a command completes.
EBLoc = 601b:	Interrupt bit location. The contents of this location are ORed into NWW when a command completes, thereby causing interrupt(s) on the channels corresponding to the one bits in EBLoc.
EELoc = 602b:	End count location. The number of words remaining in the main memory buffer at command completion is stored here as part of the posting operation.
ELLoc = 603b:	Load location. This location is used by the microcode to hold a mask of 1's shifted in from the right for generating random retransmission intervals. This location should be zeroed before starting the transmitter.
EICLoc = 604b:	Input count location. The emulator program should put the size of the input buffer (in words) into this location before starting the receiver. If a packet arrives that is longer than EICLoc, the receiver will post an Input Buffer Overrun error status.
EIPLoc = 605b:	Input pointer location. The emulator program should put a pointer to the beginning of the input buffer into this location before starting the receiver.

EOCLoc = 606b:	Output count location. The emulator program should put the size of the output buffer (in words) into this location before starting the transmitter. By convention, packets should not be substantially longer than 256 words.
EOPLoc = 607b:	Output pointer location. The emulator program should put a pointer to the beginning of the input buffer into this location before starting the transmitter.
EHLoc = 610b:	Host address location. This location must contain zero in the left byte and the host address in the right byte. The microcode will match the host address against the first byte of a passing packet to decide whether to accept it.

SIO passes commands to the interface and returns the host address of the Alto. Commands to the Ethernet interface are encoded in the two low order bits of AC0 (the remaining bits may be interpreted by other devices and thus should be zero) and have the following meaning:

AC0 [14:15]:	0	Do nothing
	1	Start the transmitter
	2	Start the receiver
	3	Reset the interface and microcode.

The host address, returned in the right byte of AC0 by SIO, is set by wires on the Alto backpanel. This number is normally put in EHLoc thereby causing packets with destination addresses matching the address set with the wires to be accepted by the reciever. For more on addressing, see below.

Upon completion of a command, EPLoc contains the status of the microcode in the left byte and the status of the interface in the right byte. The possible values of the microcode status byte, EPLoc [0:7], and their meanings are:

EPLoc[0:7] = 0:	Input done. If the hardare status byte is 377b, the interface believes the packet was recieved without error.
EPLoc[0:7] = 1:	Output done. If the hardare status byte is 377b, the interface believes the packet was sent without error. The number of collisions experienced while sending the packet is $\log_2(\text{ELLoc}/2+1)-1$.
EPLoc[0:7] = 2:	Input buffer overrun. The recieved packet was longer than the buffer, and the excess words were lost. Buffer overrun causes an early exit from the microcode input main loop, so it is likely that the CRC error and Incomplete transmission bits in the hardware status byte will be set.
EPLoc[0:7] = 3:	Load overflow. The transmitter detected 16 collisions (assuming ELLoc was zeroed before starting the transmitter) while trying to transmit the packet described by EOPLoc and EOCLoc. ELLoc will be -1.
EPLoc[0:7] = 4:	Command specified a zero length buffer.
EPLoc[0:7] = 5:	Reset. Generally indicates that a reset command (SIO with AC0 = 3) was issued to the interface when it was idle or <u>any</u> command was issued when it was not idle.
EPLoc[0:7] = 6:	Microcode branch conditions that should never happen cause this code to be posted if they do happen. Call a repairman.

EPLoc[0:7] = 7-377B:	The microcode does not generate these values for status.

Note that the microcode statuses are small integers and not individual bits as in the interface status byte. Bits in the interface status byte, EPLoc [8:15], are low true. When zero, their meanings are:

EPLoc[8:9]	Unused. These should always be one.
EPLoc[10]	Input data late. The interface did not get enough processor cycles.
EPLoc[11]	Collision.
EPLoc[12]	Input CRC bad.
EPLoc[13]	Input command issued. (AC0 [14] in last SIO)
EPLoc[14]	Output command issued. (AC0 [15] in last SIO)
EPLoc[15]	Incomplete transmission - the received packet did not end on a word boundary.

Command completion can be detected in two ways: (1) zero EPLoc and wait for it to go non-zero, and (2) set bits in EBLoc corresponding to the channels on which interrupts are desired at command completion.

When a program wishes to send a packet, it must first turn off the receiver if it is on. If the receiver is actively copying a packet into memory, the transmitter should wait for the receiver to finish (a maximum of about 1.5 ms. assuming 250-300 word packets). The program can tell whether the receiver is actively transferring or idle by zeroing the first word of the input buffer before starting the reciever. When the program wants to start the transmitter, if the first word of the current input buffer is zero, then the receiver is idle (this assumes that the first word of all Ethernet packets is non-zero).

A program can determine the size of an input message (and though not too useful, the size of an output message) by subtracting the contents of EELoc from the original buffer count in ExCLoc. The microcode never modifies the buffer count or pointer locations.

To keep the receiver listening as much of the time as possible, if EICLoc is non-zero when an output command is issued, the microcode will start the receiver 'under' the transmitter: while the transmitter is counting down a random retransmission interval after a collision, the receiver is listening. If a message arrives addressed to the receiver, the transmission attempt is aborted and the incoming message is received into the buffer described by EICLoc and EIPLoc. The transmit command is not executed in this case, and must be reissued. The microcode status byte in EPLoc will have an 'input done' status value if the transmission attempt was aborted by an incoming packet.

The first word of all Ethernet packets must contain the address to which the packet is destined in the left byte, and the address of the sender (or 'source') in the right byte. Receivers examine at least the destination byte, and in some cases the source byte to determine whether to copy the message into memory as it passes by. Address zero has special meaning to the Ethernet. Packets with destination zero are broadcast packets, and all active receivers will receive them. If a program wishes to receive all packets on the Ether regardless of address, it should put zero instead of the machine host number (returned by SIO) into EHLoc.

By convention, the second word of all Ethernet packets is designated as the *packet type.* Communication protocols using the Ethernet should use the type word to describe the protocol to which the packet belongs (for example Pup protocol packets have 1000b in the type word). The type word is purely a software convention; no Ethernet hardware or microcode interprets the type word.

7.2 *The Ethernet Interface*

The Ethernet interface consists of a interface buffer, an output shift register and phase encoder, a clock recovery circuit, an input shift register, a CRC register, and one microcode task. The hardware is shown in block diagram form in figure 8. Packets on the Ether are phase encoded and transmitter synchronous: it is the responsibility of the receiver to decide where a packet begins (and thus establish the phase of the data clock), separate the clock from the data, and deserialize the incoming bit stream. The purpose of the write register is to synchronize data transfers between the input shift register whose clock is derived from the incoming data, and the interface buffer which is synchronous to the processor system clock. The large interface buffer is necessary because the Ethernet task is relatively low priority, and the worst case latency from request to task wakeup is on the order of 20 microseconds. The phase encoder uses the system clock where one bit time is two clock periods, so the output shift register is synchronous with the buffer, however latency is still a problem and the large buffer is useful.

Included in the clock recovery section is a one-shot which is retriggered by each level transition of a passing packet. This detects the envelope of a packet and is called its 'carrier'. Ethernet phase encoders mark the beginning of a packet by inserting a single 1 bit, called the *sync* bit, in front of all transmissions. The leading edge of the sync bit of a packet will trigger the carrier one-shot of a listening receiver and establish the receiver clock phase. The sync bit is clocked into the input shift register and recirculated every 16 bit times thereafter to mark the presence of a complete word in the register. If carrier drops without the sync bit at the end of the register, the transmission was incomplete, and is flagged in the hardware status bits. When the shift register is full, the word is transferred to the interface buffer write register where it sits until the buffer control has synchronized its presence and and there is room to write it into the buffer. If the shift register fills up again before the word has been transferred from the write register to the interface buffer, data has been lost and the input data late flip flop is set.

Ethernet transmitters accumulate a 16 bit cyclic redundancy checksum on the data as it is serialized, and append it to an outgoing packet after the last data word. As a receiver deserializes an incoming packet it recomputes the checksum over the data plus the appended CRC word. If the resulting receiver checksum is non-zero, the received packet is assumed to be in error, and the condition is flagged in the hardware status byte. Since the CRC is of no interest to the emulator program, a wakeup request to empty data from the interface buffer is only made when it contains two or more words. This reduces the effective size of the buffer by one word, but insures that the CRC will be left behind at the end of a packet.

The phase encoder is started when the microcode has decremented the countdown to zero, there is no carrier present, and either the interface buffer is full, or if the message is less than 16 words long, all of it has been transferred to the buffer. The phase encoder will not start up while there is carrier present. This means that collisions can only happen because of delay in sensing carrier between widely spaced transmitters. Collisions are detected at the transceiver by comparing the data the interface is supplying to the data being received off the Ether. If the two are not identical, a signal is returned to the interface which sets the collision flip flop causing a wakeup request to the microcode which resets the interface. Countdowns are accomplished by setting a flip flop from the microcode which will cause a wakeup request on the next occurrance of SWAKMRT. This makes the grain size of countdowns about 37 microseconds.

The interface and the transceiver are connected together by three twisted pairs for signals plus two supply voltages and ground supplied from the interface. The signals are (1) transmitted data to the transceiver, (2) received data from the transceiver, and (3) the collision signal from the transceiver indicating interference.

7.3 *Ethernet Microcode*

The Ethernet microcode uses a single task and 2 registers in R:

ECntr:	The number of words remaining in the buffer.

EPntr: Points at the word prior to that next to be processed.

The task and R registers are shared by input and output so that at any time they are (1) unused, (2) transmitting a packet, or (3) receiving a packet. When an Ethernet SIO is issued while the Ethernet microcode is reset, the code dispatches on whether it is an input, output, or reset command.

Each Ethernet SIO has a result which is posted. The states of the microcode and hardware at the time of the post are deposited in EPLoc, the contents of ECntr are deposited in EELoc, and the contents of EBLoc is ORed into NWW. Note that resetting the interface with EBLoc non-zero will result in an interrupt.

An input command (SIO with AC0 [14:15] = 2) causes the microcode to start the input hardware searching for the start of a packet and then blocks. When a packet begins to arrive, the hardware wakes up the microcode, which *looks* at the interface buffer - reads the first word without advancing the read pointer - and checks the packet's address against the filtering instructions left in EHLoc by the emulator program. The packet will be accepted if any of three conditions is true: (1) If EHLoc is zero, the receiver is said to be *promiscuous* - all packets are accepted; (2) if the destination address (left byte of the first word) of the packet is zero, the packet is *a broadcast* packet - all receivers accept broadcast packets; or (3) if the destination byte matches the right byte of EHLoc - the packet was sent to that specific host. If none of these conditions is met, the packet is rejected and the microcode resets the interface, causing it to hunt for the beginning of the next packet. If the packet is accepted, the microcode enters the input main loop.

The input main loop first loads ECntr and EPntr from EICLoc and EIPLoc. Note that EICLoc and EIPLoc are not read until the receiver is committed to transferring data to memory, so these locations should not be disturbed while the receiver is running. The main loop repeatedly counts down the buffer size in ECntr and advances the buffer pointer in EPntr depositing packet words until either the hardware says that the packet is over or the buffer overflows; in either case, the input operation terminates and posts.

An output command (SIO with AC0 [14:15] = 1) causes the microcode to compute a random retransmission interval, wait that long, and then start transmitting the packet described by EOCLoc and EOPLoc. The retransmission interval is computed by ANDing the contents of ELLoc with the contents of R37, the low part of the real time clock (ELLoc is not modified). Then a one bit is left shifted into ELLoc and the high order bit of the result is tested. If the high order bit is on, the transmission attempt is aborted with a 'load overflow' microcode status. The above process is repeated each time the transmitter detects a collision while transmitting the packet. If ELLoc started out zero, each collision will double the value of ELLoc, thus doubling the mean of the random number generated by ANDing ELLoc with the real time clock. If 16 consecutive collisions occur without successfully transmitting the packet, the attempt is aborted. Note that the mean of the first retransmission interval is zero, so the first transmission attempt will begin as soon as the Ether is quiet.

After the retransmission interval is generated, it is decremented every 37 microseconds (the memory refresh task wakeup is used) until it reaches zero, at which time ECntr and EPntr are loaded from EOCLoc and EOPLoc and the transmitter part of the interface is started. Actual transmission of the packet does not begin until the interface buffer has been filled by the output main loop (or if the packet is smaller than the buffer, until all of the packet is in the buffer) and there is silence on the Ether. During countdown, if EICLoc is non-zero, the receiver is turned on, and if a packet arrives with an acceptable address, the transmission attempt is forgotten and the microcode enters the input main loop as if an input command had been issued.

The output main loop repeatedly counts down the packet length in ECntr and advances the address in EPntr taking words from the output buffer and putting them in the interface buffer until either the main memory buffer is emptied or a hardware condition aborts the operation. The output main loop is awakened for a data word once every 5.44 microseconds on the average. The microcode signals the hardware when the main memory buffer is empty and waits for the hardware to terminate; it then posts status.

A reset command (SIO with AC0 [14:15] = 3) will always bring the interface back to a reset state. If the receiver was on, it is stopped even if a packet was pouring into memory. If the transmitter was on, it is stopped, even if it was in the middle of transmitting a packet (the result to the receiver of the interrupted packet will almost certainly be an incomplete transmission and incorrect CRC). The status will immediately be posted in EPLoc: the microcode will post the reset status (5) in the microcode status byte, and the hardware will post the conditions at the time of the reset in the hardware status byte. The contents of the ECntr R register will be deposited in EELoc, and the contents of EBLoc will be ORed into NWW, possibly causing interrupts. After doing this, the interface and microcode are reset and ready for another command.

The task specific microcode functions for the Ethernet interface are sumarized below.

EIDFct	BS = 4	Input Data Function. Gates the contents of the interface buffer to BUS [0:15], and increments the read pointer at the end of the cycle.
EILFct	F1=13B	Input Look Function. Gates the contents of the interface buffer to BUS [0:15] but does not increment the read pointer.
EPFct	F1=14B	Post Function. Gates interface status to BUS [8:15]. Resets the interface at the end of the cycle.
EWFct	F1=15B	Countdown Wakeup Function. Sets a flip flop in the interface that will cause a wakeup to the Ether task on the next tick of SWAKMRT. This function must be issued in the instruction after a TASK. The resulting wakeup is cleared when the Ether task runs.
EODFct	F2=10B	Output Data Function. Loads the interface buffer from BUS [0:15], then increments the write pointer at the end of the cycle.
EOSFct	F2=11B	Output Start Function. Sets the OBusy Flip Flop in the interface, starting data wakeups to fill the buffer for output. When the buffer is full, or EEFct has been issued, the interface will wait for silence on the Ether and begin transmitting.
ERBFct	F2=12B	Reset Branch Function. This command dispatch function merges the ICmd and OCmd flip flops, into NEXT [6:7]. These flip flops are the means of communication between the emulator task and the Ethernet task. The emulator task sets them from BUS [14:15] with the STARTF function, causing the Ethernet task to wakeup, dispatch on them and then reset them with EPFct.
EEFct	F2=13B	End of transmission Function. This function is issued when all of the main memory output buffer has been transferred to the interface buffer. It disables futher data wakeups.
EBFct	F2=14B	Branch Function. ORs a one into NEXT [7] if an input data late is detected, or an SIO with AC0 [14:15] non-zero is issued, or if the transmitter or reciever goes done. ORs a one into NEXT [6] if a collision is detected.
ECBFct	F2=15B	Countdown Branch Function. ORs a one into NEXT [7] if the interface buffer is not empty.
EISFct	F2=16B	Input Start Function. Sets the IBusy Flip Flop in the interface, causing it to hunt for the beginning of a packet: silence on the Ether followed by a transition. When the interface has collected

two words, it will begin generating a data wakeups to the microcode.

8.0 *CONTROL RAM*

The control RAM is an optional logic card containing a fast (90 nsec.) 1024-word by 32-bit read/write memory, an even faster (40 nsec.) 32-word by 16-bit read/write memory, and logic to interface those memories to the Alto's microinstruction bus, processor bus, and ALU output. Unlike other memories in the Alto, the larger memory of the control RAM can hold microinstructions and/or data, and may be used exactly as the memory of a von Neumann computer.

8.1 *RAM-Related Tasks*

The control RAM performs data manipulation (as distinct from microcode fetching) functions in response to certain values of the F1 and BS fields of the microinstruction. Not all tasks will likely be interested in these functions. More important, not all tasks will have the appropriate values of the F1 and BS fields uncommitted. A RAM-related task is defined as one during whose execution the control RAM card will respond to F1 and BS fields of microinstructions. The standard Alto is wired so that the emulator task is the only RAM-related task. At most two other tasks can be made RAM-related by a simple backpanel wiring change.

8.2 *Processor Bus and ALU Interface*

The Alto's ALU output and processor bus are each 16-bits wide and its microinstruction bus is 32-bits wide, so loading the control RAM from the ALU output and reading the control RAM onto the processor bus is slightly clumsy. It is done by using the RAM-related F1's WRTRAM and RDRAM (see Appendix A).

For both reading and writing, the control RAM address is specified by the control RAM address register, which is loaded from the ALU output whenever T is loaded from its source. This load may take place as late as the microinstruction in which WRTRAM or RDRAM is asserted. The bits of the ALU output have the following significance as a control RAM address:

Bit	*Use*		
0-3	Ignored.		
4	RAM/ROM		
		0	Means read/write the control RAM.
		1	Means read the control ROM. (This doesn't quite work the way you might think. See section 8.8 for details.)
5	HALFSEL		- Ignored on writing
		0	Means read out the low-order 16-bits of the addressed word.
		1	Means read out the high-order 16-bits of the addressed word.
6-15	Word address (0-1023).		

Since it was expected that reading the control RAM would be a relatively infrequent operation, a single assertion of RDRAM reads out only one half of a 32-bit control RAM word onto the processor bus. To read out both halves, the control RAM address register must be loaded twice and RDRAM invoked twice. Data resulting from RDRAM is AND'ed onto the processor bus during the microinstruction following that in which the RDRAM was asserted.

In contrast, it was expected that writing into the control RAM would occur frequently. Therefore a single application of WRTRAM writes both halves of a control RAM word at once. The M register contents (see section 8.7) after the microinstruction containing the WRTRAM will be written into the high-order half of the addressed control RAM word. The ALU output during the microinstruction following the WRTRAM will be written into the low-order half. This protocol mates well with doubleword main memory reads.

8.3 *Microinstruction Bus Interface*

The PC0 bit of the micro-program counter (MPC) of each Alto task specifies whether that task is currently executing microinstructions from the control ROM or the control RAM. The next microinstruction address field of a microinstruction is not wide enough to specify a transfer from ROM to RAM or vice-versa. A special transfer mechanism exists only for RAM-related tasks, in the form of SWMODE, a RAM-related F1. SWMODE inverts the PC0 bit of the running task, taking effect after the microinstruction following that in which the SWMODE appears. In other words, in RAM-related tasks SWMODE behaves much like an address modifier. Other tasks cannot switch between ROM and RAM.

The correspondence of ALU output bits with microinstruction fields appears in the following table:

High/Low Order Halfword	Bit of ALU Output	Meaning	Value in Example
H	0-4	R Register Select	0
H	5-8	ALU Function Select	0
H	9-11	Bus Data Source	5
H	12-15*	Function 1	2
L	0-3*	Function 2	0
L	4	Load T	0
L	5*	Load L	1
L	6-15	Next micro address	325B

Fields denoted by * are represented with their high-order bit inverted; this is an artifact of hardware microinstruction decoding.

As an example, consider the representation of the microinstruction

```
L←MD, TASK, :LOCA;
```

where LOCA is 325B. The values for the various microinstruction fields are listed in the table above. After complementing the appropriate high-order bits and concatenating, we see that the microinstruction above would be represented as 132B in its high-order halfword and 12325B in its low-order halfword.

8.4 *Reset Mode Register*

The RAM-related F1 RMR← causes the reset mode register to be loaded from the processor bus. This register is used to supply the initial value of the PC0 bit of each task's program counter during the next reset ("boot") operation. The 16 bits of the processor bus correspond to the 16 Alto tasks in the following way: the low order bit of the processor bus specifies the initial mode of task 0, the lowest priority task (emulator), and the high-order bit of the bus specifies the initial mode of task 15, the highest priority task. A task will commence in the control ROM if its associated bit in the reset mode register contains the value 1; otherwise it will start in the control RAM. Upon initial power-up of the Alto, and after each reset operation, the reset mode register is automatically set to all 1's, corresponding to starting all tasks in the control ROM.

8.5 *Standard Emulator Access*

The standard emulator includes three instructions allowing basic access to the control RAM. More sophisticated access may be implemented by using the basic access primitives to write sophisticated access microcode into the control RAM and then transferring control to that microcode.

RDRAM (61011B) Read from Control RAM:

Reads the control RAM halfword addressed by AC1 into AC0. The microcode is:

```
RDRM:
        T←AC1, RDRAM;
        L←ALLONES;    (AND'ed with control RAM data)
        AC0←L, :START;
```

WRTRAM (61012B) Write into Control RAM:

Writes AC0 into the high-order half and AC3 into the low-order half of the control RAM word addressed by AC1. The microcode is:

```
WTRM:
        T←AC1;
        L←AC0, WRTRAM;         (This loads the M register)
        L←AC3;
        :START;
```

JMPRAM (61010B) Jump to Control RAM:

Sends control of the emulator task to the RAM location in AC1 (mod 1024). This operation is fraught with peril. If done in error it is the one of the few emulator instructions which can cause the machine to plunge completely off the deep end. If the RAM is not installed, control will go to the ROM location in AC1 (mod 1024). Clever coders can use this feature to determine from within whether or not a control RAM is installed. However they are better advised to make this determination using WRTRAM and RDRAM. The microcode for JMPRAM is:

```
JMPR:
        T←AC1, BUS, SWMODE;
        :NOVEM;          (NOVEM = 0)
```

8.6 *Interpretation of Emulator Traps*

All unused opcodes except 77400B-77777B (which is used by Swat, the Alto debugger) and 61*xxx*B, where *xxx* is between 0 and 377B, transfer to microlocation RAMTRAP with the instruction in L, the instruction cycled by 8 bits in the R-register XREG, and the emulator's R-register PC counted one beyond the trapping instruction:

```
RAMTRAP:      SWMODE, :TRAP;
...
...
TRAP:  ...., :TRAP1;
```

The result of this is that if your machine has a control RAM, these instructions will cause control to enter it at a location which is equal to TRAP1 in the ROM microcode. If no RAM is present, the unimplemented opcode will be handled as described in Section 3.3.

8.7 *M and S Registers*

The control RAM card also includes an M register and 31 S registers. The M register is the analog of the basic Alto's L register. It provides data for the S registers, which are analogous to the basic Alto's R registers. These additional registers were provided to ease the tight constraint on R register availability which might have limited the utility of the control RAM.

The similarities between the M and L registers and between the R and S registers are striking. Both M and L are loaded from the output of the ALU, and only when the Load L bit of the microinstruction is active. R registers are loaded from L, and S registers are loaded from M. Both R and S registers output data onto the processor bus. Both R and S registers are addressed by the RSELECT field of the microinstruction. (Thus the same caveats which apply to the use of R37 apply to S37 (see section 2.3 f).) Loading and reading of both R and S registers are controlled by the BS field of the microinstruction.

Nevertheless there are considerable differences. To begin with, the M and S registers are active only when a RAM-related task is executing. This means, for example, that in the highest-priority RAM-related task it is not necessary to save the value of M across a TASK, since no higher-priority task can change the value of M. Unlike the data path from the L register to the R registers, the data path from the M register to the S registers contains no shifter. When an S register is being loaded from M, the processor bus is not set to zero. The emulator-specific functions ACSOURCE and ACDEST have no effect on S register addressing. And finally, when reading data from the S registers onto the processor bus, the RSELECT value 0 causes the current value of the M register to appear on the bus. (This explains why there are only 31 useful S registers.)

8.8 *Restrictions and Caveats*

Both RDRAM and WRTRAM cause the microprocessor's system clock to stop for one cycle. This may yield unspecified results if the system clock is also stopped for some other reason (e. g. waiting for memory data). As a general rule, the system clock should run without hesitation during the microinstruction following a RDRAM or WRTRAM, except for the effect of the RDRAM or WRTRAM itself. On Alto I, there is an additional timing problem which manifests itself in some machines, for example, in the following microcode sequence:

```
MAR←FOO;            Starts memory reference
T←FIE;              Loads the control RAM address register
L←MD, WRTRAM;       Save away the high-order word in M
L←MD;               Completes the write into the RAM
```

What happens is that the last instruction suspends the system clock for one microinstruction, and some Alto I memories cannot keep the memory data good for two microinstruction times, so a parity error may occur. The data is actually stored in the RAM at the end of the first microinstruction time, so there is probably no error in the data even if a parity interrupt subsequently occurs. This "phantom" parity error may be averted by the following code, which takes three more microinstruction times, but does not invoke the horrendous microcode overhead of parity error recording:

```
MAR←FOO;            Starts memory reference
NOP;                Required for memory timing
L←MD;               Save away the low-order word
T←MD;               Save away the high-order word
TEMP←L, L←T;
T←FIE, WRTRAM;      Loads the address register, starts the write.
L←TEMP;             Complete the write into the RAM
```

Another Alto I restriction is that one cannot reliably test BUS=0 in the first instruction after a task switch into a RAM-related task when the bus data being tested is coming from the M register or one of the S registers. This restriction arises from a timing problem. The signal that determines whether a RAM-related task is running changes rather late in a microinstruction, while BUS=0 requires correct bus data some considerable time before the end of a microinstruction.

9.0 *NUTS AND BOLTS FOR THE MICROCODER*

9.1 *Standard Microcode Conventions*

The microassembler which assembles microcode for the Alto is called Mu. By convention, microcode source files have the extension .MU, and binary files have the extension .MB. Standard Alto I ROM microcode versions will be called AltoCode*x*.MU; those for Alto II will be called AltoIICode*x*.MU. A microcode source file can be divided into three largely separable pieces: the language definitions, which tell Mu what names will be used for what octal values of what microcode fields; the constant definitions, which declare all constants that may later be referenced, and which cause the constant memory to be laid out; and the register declarations, microinstruction label declarations, and microinstructions.

In order for microprograms written to execute in the RAM to be compatible with those in the ROM, at a minimum the constants assumed by the RAM microcode must be a subset of those declared by the ROM microcode, and the subset must reside in the same addresses. As a practical matter, one should preface one's RAM microcode by the same constant definitions which were used in the assembly of one's ROM microcode. In order to facilitate and encourage this compatibility, the file AltoConsts*x*.MU will be maintained (the *x* corresponding to the latest AltoCode*x*) containing definitions and constants for both Alto I and Alto II. These can be logically incorporated into other microcode assemblies via the "include" feature of Mu (#ALTOCONSTS*x*.MU;).

If one or more microcode tasks pass control back and forth between ROM and RAM, it becomes necessary to associate addresses with microinstruction labels. It is possible to do this completely generally, based on the microcode version number. A more limited solution is simply to fix the addresses of certain useful labels. The following addresses are guaranteed in all standard Alto I microcode versions after 20, and all standard Alto II microcode versions (and are included in AltoConsts*x*.MU):

Address	Label	Semantics
20B	START	Beginning of emulator's main loop; starts a new emulated instruction.
37B	TRAP1	RAM location to which unfamiliar traps are sent; ROM location which implements trap sequence.
22B	RAMCYCX	Fast cyclic shift subroutine.
105B	BLT	Block transfer subroutine.
106B	BLKS	Block store subroutine.
120B	MUL	Multiply subroutine.
121B	DIV	Divide subroutine.
124B	BITBLT	BITBLT subroutine.
160B	L0	Cyclic shift dispatch table.

9.2 *Microcode Techniques Which Need Not Be Rediscovered*

For the most part, since the Alto is such a simple machine, writing Alto microcode is a straightforward exercise in rule-following. However, during the course of writing the few-odd thousand microinstructions which have ever been written by anybody for the Alto, a few microcoding techniques have emerged as particularly ingenious or useful or both. They are recorded here for posterity.

9.21 *Microcode Subroutines*

You have probably already noticed that that the Alto hardware does not provide an easy way of doing microcode-level subroutine calls and returns. Several subroutine-call techniques have evolved. Two of these are used for RAM-to-ROM subroutine calls, and these will be presented first.

PC Call (used with BLT, BLKS, MUL, DIV, BITBLT)

This call takes advantage of the assumption that nobody in his right mind would want the emulator to execute in the non-memory I/O area from 177000B to 177777B. Therefore when one of these ROM subroutines terminates, the R-register PC is examined. If it is outside the range 177000B-177777B, then control is passed to the beginning of the emulator's main loop in the ROM. Otherwise, control is passed to location PC AND 777B in the RAM.

Warning: Some of these ROM subroutines modify PC during execution. If BLT or BLKS or BITBLT is terminated by an interrupt condition, PC is decremented by 1 so that the instruction can be resumed later. If a DIV is successful, PC is incremented by 1 to cause a skip.

Register Call (used with RAMCYCX)

This call uses an R-register, in this case CYRET (R-register 5), to dispatch into a table of successor instructions. The cyclic shift subroutine, for example, is called from six places in the ROM. Each of these places sets CYRET to the index of its successor instruction in the return dispatch table [0-5], and then dispatches into the cycle table beginning at L0. The successor corresponding to RAMCYCX dispatches into the RAM using the low-order 10 bits of the PC register.

IR Calls

These calls use the emulator's IR register in various ways: some straightforward and some devious. The main advantages of IR calls are that

1) several levels of return can be encoded into a single number, because it is fairly easy to dispatch on various parts of IR, and

2) unlike R-registers, IR can be loaded in one microinstruction.

The most straightforward use of IR is dispatching on its low-order 8 bits using the DISP bus source. Since DISP is a bus source >3, a constant may be "and-ed" onto the bus with DISP, allowing one to dispatch on sub-fields of DISP.

The most devious use of IR involves a group of constants labeled sr0 to sr12, sr14 to sr17, and sr20 to sr37 (as you might suspect, the numbers on these constant names are octal). If the constant sr*i* has been loaded into IR, then the following code will cause control to transfer to location FOO OR *i*:

```
IDISP;          (see section 3.5)
:FOO;
```

The statement above is only true if *i* is less than 20B; otherwise an additional dispatch on the DISP field of IR is required to get the desired effect:

```
FOO13:  SINK←DISP,BUS;
        :FOO20;
```

(This explains why there is no sr13. Any of sr20-sr37 will carry control to the 13Bth entry in FOO's dispatch table, where an additional level of dispatch can be used to differentiate among them if necessary. You may be wondering what is special about 13B. You are in good company.)

9.22 *The Silent Boot*

Many of the effects of a hardware "reset" operation (invoked by the boot button, or BUS[0]=1 in conjunction with the emulator-specific F1 STARTF (17B)) can be faithfully simulated by emulated software. At least two important ones cannot. A reset operation is the only way of moving non-RAM-related tasks back and forth between ROM and RAM, and the only way of guaranteeing that all tasks are initialized. However, the time required for a reset operation is not necessarily longer than a few microseconds. On both Alto I's and Alto II's a reset operation does not alter the contents of the Alto's R or S registers, its microinstruction RAM, or its main memory. Therefore if these memories contain appropriate contents it is not really necessary to go through the full disk or Ethernet bootstrap load sequence, since the major purpose of those sequences is to initialize these memories with desired contents.

The "silent boot" consists first of getting the desired contents into the RAM and main memory. The RAM should contain an emulator task (beginning with address 0) which, for example, simply jumps into the main loop of the ROM emulator code, skipping all the bootstrap code. For example:

```
NOVEM:      SWMODE;     (RAM location 0, task 0's reset location,)
            :START;     (to ROM location 20B)
```

Second, the reset mode register should be set so that the reset operation will begin execution of the emulator task in the RAM, and the other tasks wherever they are desired. Finally, the reset operation is initiated, the emulator hiccoughs momentarily into the RAM, and then proceeds in the ROM as if nothing had happened.

APPENDIX A - MICROINSTRUCTION SUMMARY

FIELDS:

0-4	RSELECT
5-8	ALUF
9-11	BS
12-15	F1
16-19	F2
20	LOAD L
21	LOAD T
22-31	NEXT

All subsequent numbers on this page are in octal.

ALUF:

0: BUS	4: BUS XOR T	10: BUS-T	14: BUS.T*
1: T	5: BUS+1*	11: BUS-T-1	15: BUS AND NOT T
2: BUS OR T*	6: BUS-1*	12: BUS+T+1*	16: UNDEFINED
3: BUS AND T	7: BUS+T	13: BUS+SKIP	17: UNDEFINED

*Loads T from ALU output

BUS SOURCE:

0: ←RLOCATION	4: (task-specific)
1: RLOCATION←	5: ←MD
2: Undefined	6: ←MOUSE
3: (task-specific)	7: ←DISP

F1(STANDARD):

0: --	4: ←L LSH 1
1: MAR←	5: ←L RSH 1
2: TASK	6: ←L LCY8
3: BLOCK	7: ←CONSTANT

F2(STANDARD):

0: --	4: BUS
1: BUS=0	5: ALUCY
2: SH < 0	6: MD←
3: SH = 0	7: ←CONSTANT

BUS SOURCE(TASK SPECIFIC):

	0 CPU	7 ETHER	4,16 KSEC,KWD	RAM Related
3:	←SLOCATION	-	←KSTAT	←SLOCATION
4:	SLOCATION←	EIDFCT	←KDATA	SLOCATION←

F1(TASK SPECIFIC):

0 CPU	4,16 KSEC,KWD	7 ETHER	10 MRT	11 DWT	12 CURT	13 DHT	14 DVT	15 PART	RAM Related
10: SWMODE	-	-	-	-	-	-	-	-	SWMODE
11: WRTRAM	STROBE	-	-	-	-	-	-	-	WRTRAM
12: RDRAM	KSTAT←	-	-	-	-	-	-	-	RDRAM
13: RMR←	INCRECNO	ELFCT	-	-	-	-	-	-	RMR←
14: -	CLRSTAT	EPFCT	-	-	-	-	-	-	-
15: -	KCOMM←	EWFCT	-	-	-	-	-	-	-
16 RSNF	KADR←	-	-	-	-	-	-	-	-
17: START	KDATA←	-	-	-	-	-	-	-	-

F2(TASK SPECIFIC):

0 CPU	4,16 KSEC,KWD	7 ETHER	10 MRT	11 DWT	12 CURT	13 DHT	14 DVT	15 PART
10: BUSODD	INIT	EODFCT	-	DDR←	XPREG←	EVENFIELD	EVENFIELD	-
11: MAGIC	RWC	EOSFCT	-	-	CSR←	SETMODE	-	-
12: DNS←	RECNO	ERBFCT	-	-	-	-	-	-
13: ACDEST	XFRDAT	EEFCT	-	-	-	-	-	-
14: IR←	SWRNRDY	EBFCT	-	-	-	-	-	-
15: IDISP	NFER	ECBFCT	-	-	-	-	-	-
16: ACSOURCE	STROBON	EISFCT	-	-	-	-	-	-
17: -	-	-	-	-	-	-	-	-

APPENDIX B - RESERVED MEMORY LOCATIONS

Location	Name	Contents
Page 0:		
0-17		Set to 77400B by OS (Swat)
Page 1:		
420	DASTART	Display list header (Std. Microcode)
421	-	Display vertical field interrupt bitword (Std. Microcode)
422	ITQUAN	Interval timer stored quantity (Std. Microcode)
423	ITIBITS	Interval timer bitword (Std. Microcode)
424	MOUSELOC	Mouse X coordinate (Std. Microcode)
425	-	Mouse Y coordinate (Std. Microcode)
426	CURLOC	Cursor X coordinate (Std. Microcode)
427	-	Cursor Y coordinate (Std. Microcode)
430	RTC	Real Time Clock (Std. Microcode)
431-450	CURMAP	Cursor bitmap (Std. Microcode)
451	-	Color Map pointer (Color Alto)
452	WW	Interrupt wakeups waiting (Std. Microcode)
453	ACTIVE	Active interrupt bitword (Std. Microcode)
456	-	Mesa disaster flag (Mesa microcode)
457		=0 (Extension of MASKTAB by convention; set by OS)
460-477	MASKTAB	Mask table for convert (Std. Microcode)
500	PCLOC	Saved interrupt PC (Std. Microcode)
501-517	INTVEC	Interrupt Transfer Vector (Std. Microcode)
521	KBLK	Disk command block address (Std. Microcode)
522	-	Disk status at start of current sector (Std. Microcode)
523	-	Disk address of latest disk command (Std. Microcode)
524	-	Sector interrupt bit mask (Std. Microcode)
525	ITTIME	Interval timer time (Std. Microcode)
526	-	Trap exit instruction (PARC/SSL-SmallTalk)
527	TRAPPC	Trap saved PC (Std. Microcode)
530-567	TRAPVEC	Trap vector (Std. Microcode)
572-577	-	Timer data (OS)
600	EPLOC	Ethernet post location (Std. Microcode)
601	EBLOC	Ethernet interrupt bit mask (Std. Microcode)
602	EELOC	Ethernet EOT count (Std. Microcode)
603	ELLOC	Ethernet load location (Std. Microcode)
604	EICLOC	Ethernet input buffer count (Std. Microcode)
605	EIPLOC	Ethernet input buffer pointer (Std. Microcode)
606	EOCLOC	Ethernet output buffer count (Std. Microcode)
607	EOPLOC	Ethernet output buffer pointer (Std. Microcode)
610	EHLOC	Ethernet host number (Std. Microcode)
611-613	-	Reserved for Ethernet expansion (Std. Microcode)
614	DCBR	Posted by parity task when a main memory parity error is detected.
615	KNMAR	" " " " " (Std. Microcode)
616	DWA	" " " " "
617	CBA	" " " " "
620	PC	" " " " "
621	SAD	" " " " "
622	-	Tape control block head (Tape Controller)
630-633	-	Run-code display processor (PARC/SSL)
631-661	-	Hexadecimal floating-point microcode (PARC/CSL)
640-644	-	Trident disk control table (Trident Disk)
700-706	-	Saved registers (Swat)
720-777	-	Reserved for SLOT devices (PARC)
776-777	-	Reserved for music (PARC/SSL)
Page 376B:		
177016-177017	UTILOUT	Printer output (Std. Hardware)
177020-177023	XBUS	Utility input bus (Alto II Std. Hardware)
177024	MEAR	Memory Error Address Register (Alto II Std. Hardware)
177025	MESR	Memory error status register (Alto II Std. Hardware)
177026	MECR	Memory error control register (Alto II Std. Hardware)
177030-177033	UTILIN	Printer status, mouse, keyset (all 4 locations return same thing)
177034	KBDAD	First of 4 words of undecoded keyboard (Std. Hardware)
177035-177037		-- remaining keyboard words
177100-177177		Run-code display processor (PARC/SSL)
177140-177157		Organ keyboard (PARC/SSL)
177200-177204		PROM programmer (PARC/CSL)
177234-177237		Experimental cursor control (PARC/SSL)
177244-177247		Graphics keyboard (PARC/SSL)
177340-177777		EARS Data buffer (PARC)

Page 377B:

177700	-	EIA interface output bit (EIA Hardware)
177701	EIALOC	EIA interface input bit (EIA Hardware)
177720-177737		TV Camera Interface (PARC/SSL)
177764-177773		Redactron tape drive (PARC/SSL)
177776	-	Scriptographics tablet X (PARC/SSL)
177776	-	Digital-Analog Converter (DAC - PARC)
177776	-	Digital-Analog Converter (Joystick - PARC/SSL)
177777	-	Scriptographics tablet Y (PARC/SSL)
177777	-	Digital-Analog Converter (Joystick - PARC/SSL)

APPENDIX C - OPTIONAL HARDWARE DEVICES FOR THE ALTO

This section lists hardware items that have been interfaced to the Alto in quantities greater than 1. EOD/SPG is the source for information about many of these interfaces and devices, and may be willing to contract to provide necessary hardware. Sources in PARC are not committed to producing any hardware. No software guarantees are made about any of these devices.

HyType Printer. A spinning daisy printer that can be ordered from Diablo Systems, Inc. Arrangements can be made with SPG to build a cable that will connect the printer to the "printer connector" on the rear of the Alto. No additonal hardware is required.

Versatec Printer/Plotter. The Versatec models ?? and ?? printers can be connected to the Alto II without additional hardware. Arrangements can be made with SPG to build a cable. Documentation on the control of the printers is available from SPG.

Tape Controller. A one-card processor-bus interface to Bucode model ?? tape drives has been built for the Alto. It will handle 1600 bpi phase-encoded tapes only. Consult PARC/SSL.

Trident Disk Interface. An interface to the Trident family of disk drives, manufactured by Calcomp, has been built. Alto II owners should consult SPG, Alto I owners consult PARC/SSL.