MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 528 July 21, 1980

CADR

by
Thomas F. Knight, Jr.
David A. Moon
Jack Holloway
and Guy L. Steele, Jr.

Abstract:

The CADR machine, a revised version of the CONS machine, is a general-purpose, 32-bit microprogrammable processor which is the basis of the Lisp-machine system, a new computer system being developed by the Laboratory as a high-performance, economical implementation of Lisp. This paper describes the CADR processor and some of the associated hardware and low-level software.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Overview

The CADR microprocessor is a general purpose processor designed for convenient emulation of complex order codes, particularly those involving stacks and pointer manipulation. It is the central processor in the LISP machine project, where it interprets the bit-efficient 16-bit order code produced by the LISP machine compiler. (The terms "LISP machine" and "CADR machine" are sometimes confused. In this document, the CADR machine is a particular design of microprocessor, while the LISP machine is the CADR machine plus the microcode which interprets the LISP machine order code.)

The data paths of the CADR machine are 32 bits wide. Each 48-bit-wide microcode instruction specifies two 32-bit data sources from a variety of internal scratchpad registers; the two data-manipulation instructions can also specify a destination address. The internal scratchpads include a 1K pointer-addressable RAM intended for storing the top of the emulated stack, in a manner similar to a cache. Since in the LISP machine a large percentage of main memory references will be to the stack, this materially speeds up the machine.

The CADR machine has a 14-bit microprogram counter, which behaves much like that of a traditional processor, allowing up to 16K of writable microprogram memory. Also included is a 32-location microcode subroutine return stack.

Memory is accessed through a two-level virtual paging system, which maps 24-bit virtual addresses into 22-bit physical addresses.

There are four classes of micro-instructions. Each specifies two sources (A and M); the ALU and BYTE operations also specify a destination (A, or M plus functional). The A bus supplies data from the 1024-word A scratchpad memory, while the M bus supplies data from either the 32-word M scratchpad memory (a copy of the first 32 locations of the A scratchpad) or a variety of other internal registers. The four classes of microinstruction are:

ALU — The destination receives the result of a boolean or arithmetic operation performed on the two sources.

BYTE — The destination receives the result of a byte extraction, byte deposit, or selective field substitution from one source to the other. The byte so manipulated can be of any non-zero width.

JUMP — A transfer of control occurs, conditional on the value of any bit accessible to the M bus, or on a variety of ALU and other internal conditions such as pending interrupts and page faults.

DISPATCH — A transfer of control occurs to a location determined by a word from the dispatch memory selected by a byte of up to seven bits extracted from the M bus.

There are several sources and destinations whose loading and use invoke special action by the microprocessor. These include the memory address and memory data

registers, whose use initiates main memory cycles.

Some of the ALU operations are conditional, depending upon the low order bit in the Q register and the sign of A source. These operations are used for multiply and divide steps.

The main features of this machine which make it suitable for interpreting the LISP machine order code are its dynamically writable microcode, its very flexible dispatching and subroutining, its excellent byte manipulation abilities, and its internal stack storage. While the design of CADR was strongly influenced by the requirements of the LISP machine design, a conscious attempt was made to avoid features that are extremely special-purpose. The goal is a machine that happens to be good at interpreting the particular order code of the LISP machine, but which is general enough to interpret others almost as well. In particular, no critical parts of the LISP machine design (such as LISP machine instruction formats) are "wired in"; thus any changes to the LISP machine design can be easily accomodated by CADR. However, there are several "efficiency hacks" in the hardware, designed to speed up certain common operations of the LISP machine microcode, which might not be useful for other microcodes. These are described in later sections of this document.

Notational Conventions

All numbers used to describe bit positions, field widths, memory sizes, etc. are decimal. Octal is used only (and exclusively) to describe the values of fields. Bits within a word are consistently numbered from right to left, the least significant bit being bit <0>. Fields are described by the numbers of their most and least significant bits (e.g. "bits <22-10>").

Whenever a particular field value is described as "illegal", it does not mean that specifying that value will screw up the operation of the machine. It merely indicates a value which happens to have a certain function, not because it is considered directly useful, but because the internal workings of the machine may force certain selectors to that value for other reasons, and the user can select this value too even though it is not normally useful. These illegal values are described for the benefit of someone who may wish to fathom these inner workings.

A field value described as "unused" is reserved for possible design expansion and should not be used in programs. Bit fields described as "unused" should be zero in programs, for the sake of future compatibility.

Since the use of the term "micro" in referring to registers and instructions becomes redundant, its use will be dropped from here on in this part of the document. All instructions discussed are microinstructions.

The following bits are treated the same in every instruction. They will not be repeated in the individual instruction descriptions

IR<48> = Odd parity bit

IR<47> = Unused

IR<46> = Statistics (see the description of the Statistics Counter) This can be used to count how many times specified areas of the microcode are executed, to implement microcode breakpoints, or to stop the machine at a certain "time".

IR<45> = ILONG (1 means slow clock)

IR<44-43> = Opcode (0 ALU, 1 JUMP, 2 DISPATCH, 3 BYTE)

IR<42> = POPJ transfer. Causes a return from a micro subroutine, after executing one additional instruction.

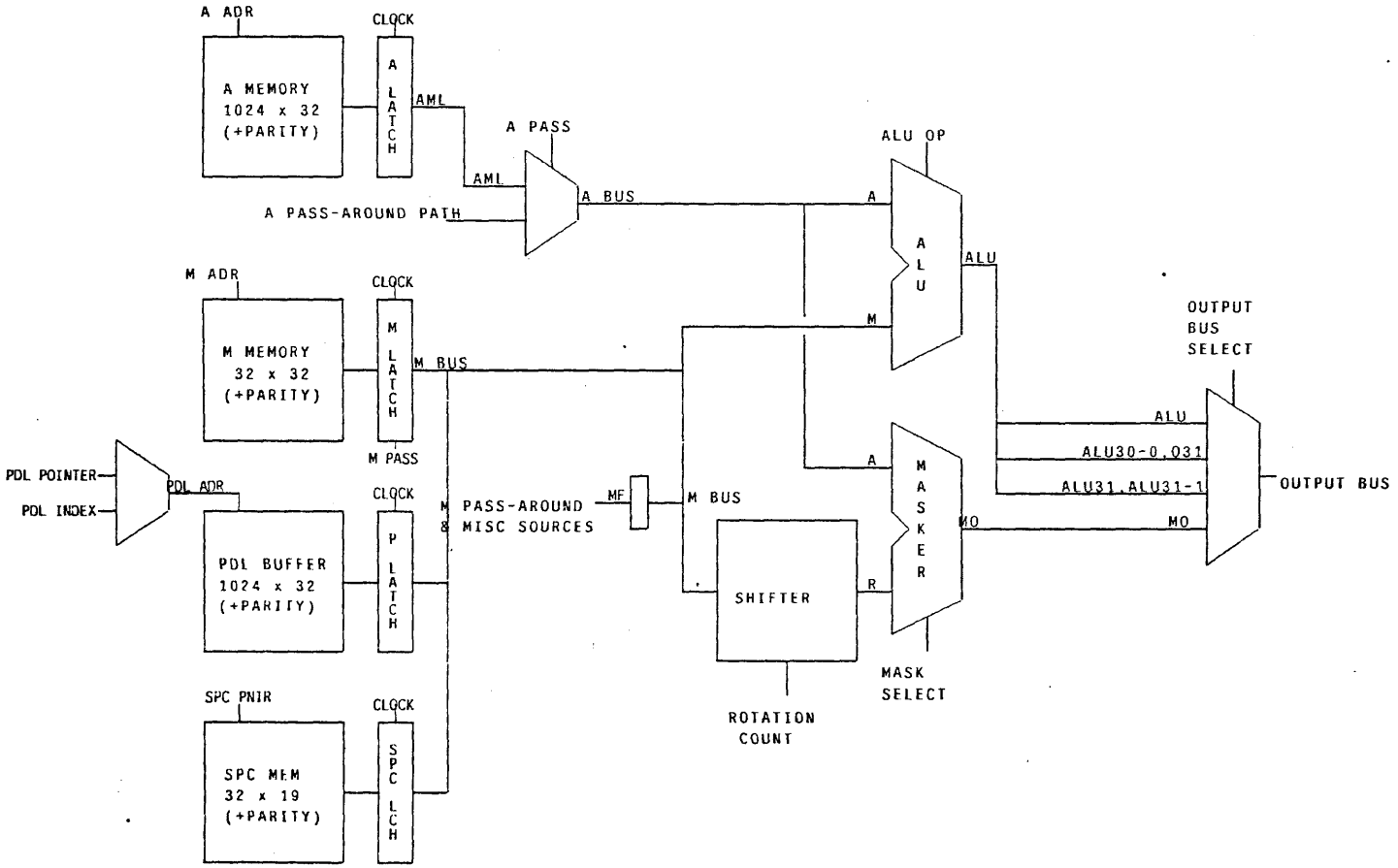
IR<11-10> = Miscellaneous Functions

- 0 Normal
- 1 Not used
- 2 Write dispatch memory, if opcode is DISPATCH.
- 3 Enable modification of the M-ROTATE field by the location counter (LC). See the description of the instruction-stream hardware.

Data Paths

The data paths of the machine consist of two source busses A and M, which provide data to the ALU and byte extractor, and an output bus OB, which is selected from the ALU (optionally shifted left or right) or the output of the byte extractor, and whose data can be routed to various destinations. We first describe the specification of the source busses, which are identically specified for all instructions; then the destination specifiers which control where results are stored; and finally the two instructions for controlling the ALU and the byte extractor.

<<picture CHODAM goes here - use DPLT>>


A ADR
CLOCK
A MEMORY
1024 x 32
(+PARITY)
A
L
A
T
C
H
AML
AML
A PASS
A BUS
A PASS-AROUND PATH
ALU OP
A
A
L
U
ALU
M ADR
CLOCK
M MEMORY
32 x 32
(+PARITY)
M
L
A
T
C
H
M BUS
M PASS
M
OUTPUT
BUS
SELECT
ALU
ALU30-0,031
ALU31,ALU31-1
MO
MO
OUTPUT BUS
PDL POINTER
PDL INDEX
PDL ADR
CLOCK
PDL BUFFER
1024 x 32
(+PARITY)
P
L
A
T
C
H
M
&
PASS-AROUND
MISC SOURCES
MF
M BUS
A
M
A
S
K
E
R
SHIFTER
R
MASK
SELECT
ROTATION
COUNT
SPC PNTR
CLOCK
SPC MEM
32 x 19
(+PARITY)
S
P
C
L
C
H

Sources

All instructions specify sources in the same way. There are two source busses in the machine, the A bus and the M bus. The A bus is driven only from the A scratchpad memory of 1024 32-bit words. The M bus is driven from the M scratchpad of 32 32-bit words and a variety of other sources, including main memory data and control registers, the PC stack (for restoring the state of the processor after traps), the internal stack buffer and its pointer registers, the macrocode location counter, and the Q register. Addresses for the A and M scratchpads are taken directly from the instruction. The alternate sources of data for the M source are specified with an additional bit in the M source field.

```
IR<41-32> = A source address
IR<31-26> = M source address
        If IR<31> = 0,
                IR<30-26> = M scratchpad address
        If IR<31> = 1,
                IR<30-26> = M "functional" source
                                0  Dispatch constant (see below)
                                1  SPC pointer <28-24>, SPC data <18-0>
                                2  PDL pointer <9-0>
                                3  PDL index <9-0>
                                5  PDL Buffer (addressed by Index)
                                6  OPC registers (see below) <13-0>
                                7  Q register
                               10  VMA register (memory address)
                               11  MAP[MD]
                               12  MD register (memory data)
                               13  LC (location counter)
                               14  SPC pointer and data, pop
                               24  PDL buffer, addressed by Pointer, pop
                               25  PDL buffer, addressed by Pointer
```

Functional sources not listed above should not be used and may have side effects. Sources 15, 16, and 17 are reserved for future expansion. Source 4 is the PDL buffer, indexed by the PDL Index, and the PDL pointer is decremented, presumably a useless operation.

Programming hint: it is often convenient to reserve one A memory word and one M memory word and fill them with constant zeros, to provide a zero source for each source bus. It is also convenient to have an M memory word containing all ones. These are particularly useful for byte extraction, masking, bit setting, and bit clearing operations. The CONSLP assembler in fact assumes that A memory location 2 and M memory location 2 are sources of zeros. The UCONS microcode stores all ones in location 3.

The M scratchpad normally contains a duplicate copy of the first 32 locations of the A scratchpad. The effect is as if there were a single scratchpad memory, the first 32 locations of which were dual-ported. This makes programming more convenient, since

these locations are accessible to both sides of the ALU and shifter.

Destinations

The 12-bit destination field in the BYTE and ALU instructions specifies where the result of the instruction is deposited. It is in one of two forms, depending upon the high-order bit. If the high-order bit is 1, then the low 10 bits are the address of an A memory location, and the remaining bit is unused. If the high order bit is 0, the low 10 bits are divided into a 5-bit "functional destination" field, and a 5-bit M scratchpad address, and both of the places specified by these fields get written into. The next-to-highest bit in the destination field is not used.

```
IR<25-14> = Destination
        If IR<25> = 1,
                IR<23-14> = A scratchpad write address
        If IR<25> = 0,
                IR<23-19> = Functional destination write address
                         0  None
                         1  LC (Location Counter)
                         2  Interrupt Control <29-26>
                                Bit 26 = Sequence-Break request
                                Bit 27 = Interrupt-Enable
                                Bit 28 = Bus-Reset
                                Bit 29 = LC Byte-mode
                        10  PDL (addressed by Pointer)
                        11  PDL (addressed by Pointer), push
                        12  PDL (addressed by Index)
                        13  PDL Index
                        14  PDL Pointer
                        15  SPC data, push
                        16  Next instruction modifier
                                ("OA register"), bits <25-0>
                        17  Next instruction modifier
                                ("OA register"), bits <47-26>
                        20  VMA register (memory address)
                        21  VMA register, start main memory read
                        22  VMA register, start main memory write
                        23  VMA register, write map.  The map is
                            addressed from MD and written from
                            VMA.  VMA<26>=1 writes the level 1
                            map from VMA<31-27>.  VMA<25>=1 writes
                            the level 2 map from VMA<23-0>.
                        30  MD register (memory data)
                        31  MD register, start main memory read
                        32  MD register, start main memory write
                        33  MD register, write map like 23
                IR<18-14> = M scratchpad write address
```

Functional destinations not listed may have strange results. Destinations 3-7 are reserved for expansion.

Note: If you write into the M-memory, the machine will also write into the corresponding A-memory address. Therefore you should never write into A-memory locations 0-37; this way the first 40 (octal) locations of A-memory "map into" the M-memory.

The full details of the more complicated functional destinations are described in later sections below. The Q register is loaded by using the Q-control field of the ALU instruction, not by using a functional destination. In addition, it loads from the ALU outputs, not the output bus. This means that the left and right shift operations are ineffective for data being loaded into Q.

Programming hint: if a functional destination is specified, an M scratchpad location must also be specified. It is convenient to reserve one location of the M scratchpad for "garbage"; this location can be specified when it is desired to write into a functional destination but not into any other M scratchpad location. Since the CONSLP assembler defaults the M write address to zero, it is best to let location 0 be the garbage location. Location 0 of the A scratchpad will also be written and is also reserved as a garbage location.

The ALU Instruction

The ALU operation performs most of the arithmetic in the machine. It specifies two sources of 32 bit numbers, and an operation to be performed by the ALU. The operation can be any of the 16 boolean functions on two variables, two's complement addition or subtraction, left shift, and several less useful operations. The carry into the ALU can be forced to be 0 or 1. The output of the ALU is optionally shifted one place, and then written into the specified destinations via the output bus. Additionally, the ALU instruction specifies one of four operations upon the Q register. These are do nothing, shift left, shift right, and load from the ALU outputs. An additional bit in the ALU operation field is decoded to indicate conditional operations; this is how the "multiply step" and "divide step" operations are specified. (Multiplication and division are explained in greater detail in another section.)

```
IR<44-43> = 0 (ALU opcode)
IR<41-32> = A source
IR<31-26> = M source
IR<25-14> = Destination
IR<13-12> = Output bus control
                0  Byte extractor output (illegal)
                1  ALU output
                2  ALU output shifted right one, with the correct
                   sign shifted in, regardless of overflow.
                3  ALU output shifted left one, shifting in Q<31>
                   from the right.
IR<9>     = not used
IR<8-4>   = ALU operation
                If IR<8> = 0,
                        IR<7-3> = ALU op code (see table)
                If IR<8> = 1,
                        IR<7-3> = Conditional ALU op code
                                  0  Multiply step
                                  1  Divide step
                                  5  Remainder correction
                                 11  Initial divide step
IR<2>     = Carry into low end of ALU
IR<1-0>   = Q control
                0  Do nothing
                1  Shift Q left, shifting in the inverse
                        of the sign of the ALU output (ALU<31>)
                2  Shift Q right, shifting in the low bit
                        of the ALU output (ALU<0>)
                3  Load Q from ALU output
```

ALU operation codes (from Table 1 of 74181 specifications). All arithmetic operations are two's complement. Note that the bits are permuted in such a way as to make the logical operations come out with the same opcodes as used by the Lisp BOOLE function. Names in square brackets are the CONSLP mnemonics for the operations.

	Boolean (IR<7>=1)		Arithmetic (IR<7>=0)			
IR<6-3>			Carry in = 0		Carry in = 1	
0	ZEROS	[SETZ]	-1		0	
1	M∧A	[AND]	(M∧A)-1		M∧A	
2	M∧¬A	[ANDCA]	(M∧¬A)-1		(M∧¬A)	
3	M	[SETM]	M-1		M	
4	¬M∧A	[ANDCM]	M∨¬A		(M∨¬A)+1	
5	A	[SETA]	(M∨¬A)+(M∧A)		(M∨¬A)+(M∧A)+1	
6	M⊕					
A	[XOR]	M-A-1	[M-A-1]	M-A	[SUB]	
7	M∨A	[IOR]	(M∨¬A)+M		(M∨¬A)+M+1	
10	¬A∧¬M	[ANDCB]	M∨A		(M∨A)+1	
11	M≡A	[EQV]	M+A	[ADD]	M+A+1	[M+A+1]
12	¬A	[SETCA]	(M∨A)+(M∧¬A)		(M∨A)+(M∧¬A)+1	
13	M∨¬A	[ORCA]	(M∨A)+M		(M∨A)+M+1	
14	¬M	[SETCM]	M		M+1	[M+1]
15	¬M∨A	[ORCM]	M+(M∧A)		M+(M∧A)+1	
16	¬M∨¬A	[ORCB]	M+(M∨¬A)		M+(M∨¬A)+1	
17	ONES	[SETO]	M+M	[M+M]	M+M+1	[M+M+1]

The BYTE Instruction

The BYTE instruction specifies two sources and a destination in the same way as the ALU instruction, but the operation performed is one of selective insertion of a byte field from the M source into an equal length field of the word from the A source. The rotation of the M source is specified by the SR bit as either zero or equal to the contents of the ROTATE field. The rotation of the mask used to select the bits replaced is specified by the MR bit as either zero or equal to the contents of the ROTATE field. The length of the mask field used for replacement is specified in the LENGTH MINUS 1 field. The four states of the SR and MR bits yield the following operations:

MR=0 SR=0 Not useful (This is a subset of other modes.)

MR=0 SR=1 LOAD BYTE PDP-10 LDB instruction (except the unmasked bits are from the A source). A byte of arbitrary position from the M source is right-justified in the output.

MR=1 SR=0 SELECTIVE DEPOSIT The masked field from the M source is used to replace the same length and position byte in the word from the A source.

MR=1 SR=1 DEPOSIT BYTE PDP-10 DPB instruction. A right-justified byte from the M source is used to replace a byte of arbitrary position in the word from the A source.

The BYTE instruction automatically makes the output of the byte extractor available by forcing the output bus select code to 0 (byte extractor output).

```
IR<44-43> = 3 (BYTE operation)
IR<41-32> = A source
IR<31-26> = M source
IR<25-14> = Destination
IR<13>    = MR = Mask Rotate (see above)
IR<12>    = SR = Source Rotate (see above)
IR<9-5>   = Length of byte minus 1 (0 means byte of length 1, etc.)
IR<4-0>   = Rotation count (to the left) of mask and/or M source
```

The byte operation rotates the M source by 0 (if SR=0) or by the rotation count (if SR=1), producing a result called R. It also uses the MR bit, the rotation count, and the length minus 1 field to produce a selector mask (see description below). This mask is all zeros except for a contiguous section of ones denoting the selected byte. This mask is used to merge the A source with R, bit by bit, selecting a bit from A if the mask is 0 and from R if the mask is 1. This result is then written into the specified destination(s).

Output of mask memories:

Right mask memory is indexed by 0 (MR=0) or by rotation count (MR=1).
Left mask memory is indexed by (the index into right mask memory) plus (the length minus 1 field), mod 32.

octal index	LEFT MASK MEMORY contents	RIGHT MASK MEMORY contents
0	00000000000000000000000000000001	11111111111111111111111111111111
1	00000000000000000000000000000011	11111111111111111111111111111110
2	00000000000000000000000000000111	11111111111111111111111111111100
3	00000000000000000000000000001111	11111111111111111111111111111000
4	00000000000000000000000000011111	11111111111111111111111111110000
5	00000000000000000000000000111111	11111111111111111111111111100000
6	00000000000000000000000001111111	11111111111111111111111111000000
7	00000000000000000000000011111111	11111111111111111111111110000000
10	00000000000000000000000111111111	11111111111111111111111100000000
11	00000000000000000000001111111111	11111111111111111111111000000000
12	00000000000000000000011111111111	11111111111111111111110000000000
13	00000000000000000000111111111111	11111111111111111111100000000000
14	00000000000000000001111111111111	11111111111111111111000000000000
15	00000000000000000011111111111111	11111111111111111110000000000000
16	00000000000000000111111111111111	11111111111111111100000000000000
17	00000000000000001111111111111111	11111111111111111000000000000000
20	00000000000000011111111111111111	11111111111111110000000000000000
21	00000000000000111111111111111111	11111111111111100000000000000000
22	00000000000001111111111111111111	11111111111111000000000000000000
23	00000000000011111111111111111111	11111111111110000000000000000000
24	00000000001111111111111111111111	11111111111100000000000000000000
25	00000000011111111111111111111111	11111111111000000000000000000000
26	00000000111111111111111111111111	11111111110000000000000000000000
27	00000001111111111111111111111111	11111111100000000000000000000000
30	00000011111111111111111111111111	11111111000000000000000000000000
31	00000111111111111111111111111111	11111110000000000000000000000000
32	00001111111111111111111111111111	11111100000000000000000000000000
33	00011111111111111111111111111111	11111000000000000000000000000000
34	00111111111111111111111111111111	11110000000000000000000000000000
35	01111111111111111111111111111111	11100000000000000000000000000000
36	11111111111111111111111111111111	11000000000000000000000000000000
37	11111111111111111111111111111111	10000000000000000000000000000000

After the two masks are selected, they are AND'ed together to get the final mask. This mask is all zeros, except for a field of contiguous ones defining the byte.

As an example, if MR=1, rotation count=5, and length minus 1=7, then the right mask index is 5 and the left mask index is 14 (octal). This results in a final mask as follows:

```
        Right mask 5    11111111111111111111111111100000
        Left mask 14    00000000000000000001111111111111
AND them together       --------------------------------
        Final mask      00000000000000000001111111100000
```

The byte is 8 bits wide, 5 positions from the right.

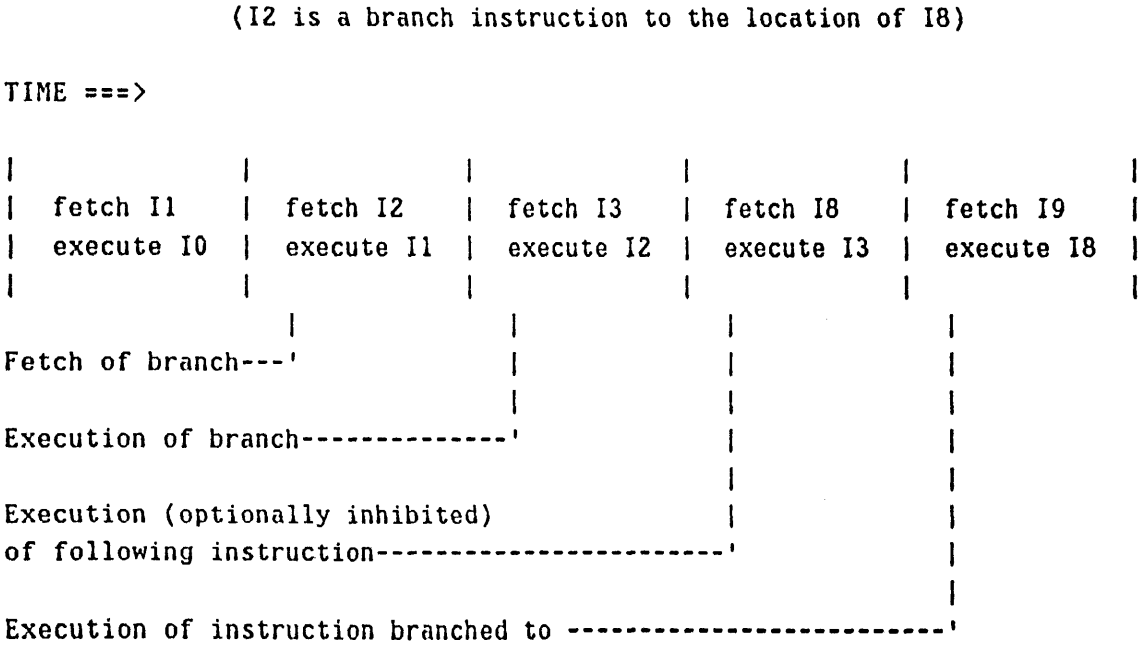
Programming hint: if the byte is "too large" (i.e. its position and size specifications cause it to hang over the left-hand edge of a word), then the masker does not truncate the byte at the left-hand edge. Instead, it produces a zero mask, selecting no byte at all; thus, the output of the byte operation equals the A source. The reason for this is that an overflow occurs in calculating the index into the left mask memory, and so the final mask is zero. For example, if MR=1, rotation count=20 (octal), and length minus 1=27 (octal), then the right mask index is 20 and the left mask index is 477 (mod 32). This results in a final mask as follows:

```
        Right mask 20   11111111111111110000000000000000
        Left mask 7     00000000000000000000000011111111
AND them together       --------------------------------
        Final mask      00000000000000000000000000000000
```

Control

The control section of the processor consists of a 14-bit program counter (the PC), a 32-location PC stack (SPC) and stack pointer (SPCPTR), and a 2K dispatch memory, used during the DISPATCH instruction. Unlike some microprocessors, and like most traditional machines, the normal mode of operation is to execute the next sequential instruction by incrementing the PC.

The processor uses single instruction look ahead, i.e. the lookup of the next instruction is overlapped with execution of the current one. This implies that after branching instructions the processor normally executes the following instruction, even if the branch was successful. Provision is made in these instructions to inhibit this execution (with the N bit), but the cycle it would have used will then be wasted.



Two types of instruction affect flow of control in the machine. The conditional JUMP specifies a new PC and transfer type in the instruction itself, while the DISPATCH instruction looks up the new PC and transfer type in the 2K dispatch memory. In either case, the new PC is loaded into the PC register, and the operation specified by the 3-bit transfer type is performed. These operations are:

N bit — If on, inhibits execution of the next instruction, i.e. the instruction at the address one greater than that of the transfer instruction. (This instruction needn't actually be at the address one greater, if a transfer of control was

already in progress.) The cycle that would have executed that instruction is wasted.

The P and R bits are decoded as follows:

P=0 R=0	BRANCH	Normal program transfer.
P=1 R=0	CALL	Save the correct return address on the SPC stack, and jump to the new PC address.
P=0 R=1	RETURN	Ignore new PC; instead pop PC off the SPC stack.
P=1 R=1	FALL THROUGH	In a DISPATCH instruction, do not dispatch.
	I-MEM WRITE	In a JUMP instruction, write into the instruction memory, and do not jump.

The BRANCH transfer type is the normal program transfer, without saving a return address.

The CALL transfer type pushes the appropriate return address onto the SPC stack. This stack is 32 locations long. It is the responsibility of the programmer to avoid overflows. The return address is PC+2, or PC+1 if the N bit is also on. Actually, if the N bit is on the address of the instruction NOP'ed is saved, which may not be identical to PC+1 if a transfer of control is already in progress. If the N bit is not on, 1 + the address of that instruction is saved. In the case of a dispatch, if the N bit is on and bit 25 of the instruction is on, save PC, the address of the dispatch instruction itself; this allows the dispatch to be re-executed upon return. (Actually, due to pipelining, when the above paragraph says PC it doesn't really mean PC.)

The RETURN transfer type pops a return PC from the SPC stack, ignoring the PC specified in the instruction or dispatch table.

The FALL THROUGH transfer type for dispatches allows some entries in a dispatch table to specify that the dispatch should not occur after all. The following instruction is executed (unless inhibited), followed by the one after that (unless the first following one branches and inhibits it!).

The I-MEM WRITE transfer type is the mechanism for writing instructions into the microprogram instruction memory, and is described in a later section. (The dispatch memory, unlike the instruction memory, is <u>not</u> written into by setting the P and R bits (after all, in a dispatch instruction these bits come from the dispatch memory!); instead, the Miscellaneous Function field is used.)

An additional bit in every instruction, including ALU and BYTE instructions, called the POPJ bit, allows specification of simultaneous execution of a RETURN transfer type along with execution of any instruction. That is, it does the same thing as if this instruction, in addition to whatever else it does, had executed a RETURN transfer type jump without the N bit on. It is the responsibility of the programmer to avoid conflicts in the use of this bit simultaneously with other types of transfers.

The POPJ bit should be used in a JUMP instruction only in conjunction with the RETURN transfer type. This will cause a RETURN operation in either case, but execution of the following instruction is conditional, controlled by the N bit and the conditional JUMP instruction. The POPJ bit, when used in a DISPATCH instruction, is

specially over-ruled by the JUMP and CALL transfer types. This allows you to RETURN normally, but jump off to other code in exceptional cases, using the same dispatch table as other dispatch instructions which do not want to return. The POPJ bit should not be used in conjunction with writing of dispatch or instruction memory, nor with the SPC pop and push functional source and destination. The machine doesn't bother to do anything reasonable in these cases.

The DISPATCH Instruction

The dispatch instruction allows selection of any source available on the M bus [see description of M bus sources in the Data Path section], and the dispatch on any subfield of up to 7 bits from the selected word. The selected subfield is ORed with the "dispatch address" field of the instruction to produce an 11 bit address. This address is used to look up a 14 bit PC and 3 bit transfer type in the dispatch memory. The SPC-pointer-and-data-pop source will not operate reasonably in conjunction with the dispatch instruction.

```
IR<44-43> = 2 (DISPATCH operation)
IR<41-32> = Dispatch constant (also A source when writing D-MEM)
IR<31-26> = M source
IR<25>    = Alter return address pushed on SPC by the CALL transfer
            type, if the N bit is set, to be the address of this
            instruction rather than the next instruction.
IR<24>    = Enable instruction-stream hardware (described later).
IR<23>    = Unused
IR<22-12> = Address in dispatch memory
IR<9-8>   = Control dispatching off the map, see below.
IR<7-5>   = Length of byte (not minus 1!) from M source to dispatch on
IR<4-0>   = Rotation count (to the left) of M source
```

The dispatch operation takes the specified M source word and rotates it to the left as specified by the rotation count. All but the low K bits are masked out, where K is the contents of the length field. The result is OR'ed with the dispatch address, and this is used to address the 2K dispatch memory, which supplies the new PC and the R, P, and N bits.

If bits 8 and 9 of IR are not zero, the bottom bit of the dispatch address comes from the virtual memory map rather than the rotator and masker. The address inputs to the map in this case come from MD. This is primarily useful for testing pointers just fetched from main memory for validity with respect to the garbage collector's conventions. IR<8> selects bit 14 of the second level map, and IR<9> selects bit 15. Selecting both bits ORs them together.

The dispatch constant field is loaded into the DISPATCH CONSTANT register on every dispatch instruction. This register is accessible as an M source. The dispatch constant field has nothing whatsoever to do with the operation of dispatching; it is merely a convenient device for loading a completely random register while doing something else. (Uses for this feature are discussed in a later section.)

Miscellaneous function 2 inhibits the normal action of the instruction and instead loads the dispatch memory with the low order contents of the A memory scratchpad location specified in the A source. Note that the A source address is the same field as the dispatch constant field. The dispatch constant is loaded anyway, but this can be ignored. The parity bit (bit 17) is also loaded, and it is the responsibility of the

programmer to load correct (odd) parity into the memory. Normal addressing of the dispatch memory is in effect, so it is advisable to have the length field contain 0 so that the dispatch memory location to modify is uniquely specified by the dispatch address in the instruction.

The JUMP Instruction

The JUMP instruction allows conditional branching based on any bit of any M source or on a variety of internal processor conditions, including ALU output. (While DISPATCH could also be used to test single M source bits, the use of JUMP saves dispatch memory.) The JUMP operation is also used, by means of a trick, to write into the instruction memory.

```
IR<44-43> = 1 (JUMP operation)
IR<41-32> = A source
IR<31-26> = M source
IR<25-12> = New PC
IR<9>     = R bit (1 means pop new PC off SPC stack)
IR<8>     = P bit (1 means push return PC onto SPC stack)
IR<7>     = N bit (1 means inhibit next instruction if jump successful)
IR<6>     = If 1, invert sense of jump condition
IR<5>     = If 0, test bit of M source; if 1, test internal condition
IR<4-0>   = If IR<5>=0, rotation count for M source.
            If IR<5>=1, condition number:
                0  Low bit of shifter output (illegal)
                1  M source < A source
                2  M source ≤ A source
                3  M source = A source
                4  Page fault
                5  Page fault or interrupt pending
                6  Page fault or interrupt pending or sequence break flag
                7  Unconditionally true
```

Page faults, interrupts, and sequence breaks are documented in later sections.

The jump condition is determined as follows. If IR<5>=0, then the M source is rotated left by the rotation count; the low-order bit of the result is then tested. Thus, to test the sign bit, a rotation count of 1 should be used. The jump condition is true if the low-order bit is 1. If IR<5>=1, then the specified internal condition is tested. In either case, the sense of the jump condition is inverted if IR<6>=1. In particular, this allows testing of all six arithmetic relations between the M and A sources.

If the final jump condition, possibly after inversion, is true, then the new PC field and the R, P, and N bits are used to determine the new contents of the PC. If the condition is not true, execution continues with the next instruction, modulo the POPJ bit.

If both the R and P bits are set (WRITE), then A and M sources are (conditionally!) written into the instruction memory. Bits <47-32> are taken from A source bits <15-0>; bits <31-0> are taken from M source <31-0>. Notice that this is not the same alignment of bits as is used for the "next instruction modify" functional destinations (16 and 17). The reason for the odd location of WRITE in the instruction

set is due to the way in which it operates. It causes the same operations as the CALL transfer type, resulting in the the old PC plus 1 or 2 being saved on the SPC stack and the PC register being loaded with the address to be modified. Then, when the instruction memory would normally be fetching the instruction to be executed from that location, a write pulse is generated, causing the saved data from the A and M sources to be written into the instruction memory. Meanwhile, the machine simulates a RETURN transfer instruction, causing the SPC stack to be popped back into the PC and instruction execution to proceed from where it left off. Note that this instruction requires use of a word on the SPC stack and requires an extra cycle. It is highly recommended that the N bit also be on in the JUMP instruction, since the processor will be executing a RETURN transfer type unconditionally during what should be the execution of the instruction following the write. If, however, this does not conflict with other things that this following instruction specifies, then the following instruction may be executed. Care is required.

Program Modification

A novel technique is used for variabilizing fields in the program instruction. Two of the "functional destinations" of the output bus are (conceptual) registers (sometimes collectively referred to as the OA register), whose contents get OR'ed with the next instruction executed. Combined with the shifter/masker ability to move any contiguous set of bits into an arbitrary field, this feature provides, for example, variable rotation counts and the ability to use program determined addresses of registers; for example, it can be used to index into the A scratchpad memory.

Functional destination 16 (OA-REG-LOW), when written into, effectively OR's bits <25-0> into bits <25-0> of the next instruction; functional destination 17 (OA-REG-HIGH) effectively OR's bits <21-0> into bits <47-26> of the next instruction. The place between bits <26> and <25> is a natural dividing line for all classes of instructions. Note that only one half of a particular instruction can be modified, since it is impossible to write into both functional destinations simultaneously.

When this feature is used, parity checking is disabled for the word fetched from the instruction memory, since the OA "register" is OR'ed into the output of the memory before parity is checked.

This feature is particularly useful for supplying the address of a location of instruction memory or dispatch memory to be written into, for specifying variable addresses in the A and M memories, and for operations on bytes of variable length or position. Examples of these are detailed in a later section.

Clocks

The CADR processor uses only one clock signal. This clock loads output data into the designated registers, and a new PC and instruction are also loaded. The only events which do not take place synchronous with the clock are the control signals for the A, M, and PDL scratchpads and the SPC stack. For these devices, a two stage cycle is performed. During the first phase, the source addresses of the respective devices are gated into the address inputs. After the output data has settled, the outputs of these devices are latched. Then, the address is changed to that specified as the write location from the previous instruction. After the address has settled, a write pulse is generated for the scratchpad memory to perform the write. Pass-around paths are provided (invisibly to the programmer) for the A and M memories, which notice and correct read references to a location which was written into on the previous cycle but has not yet actually been written into the scratchpad. No such pass-around path is provided for the PDL memory, because on any cycle in which the PDL memory is written into, the M scratchpad must also be written into, and so the next instruction can refer to that M scratchpad location, thereby using the M pass-around path. The SPC stack has a pass-around path when used by the RETURN transfer type, but does not have a pass-around path when used as an M source. The RETURN pass-around path makes it possible to have a subroutine only two instructions long. It would take extra hardware to provide the missing pass-around paths, and examination of actual microprograms showed that they would be very rarely used.

The clock cycle is of variable length. The duration of the first half of the cycle (the "read phase") is controlled by both the ILONG bit of the instruction (IR<45>) and by two "speed" bits from the diagnostic interface. The duration of the second half (the "write phase") is normally fixed. This clock serves as both the processor clock and a clock for the bus interface, memory, and external devices.

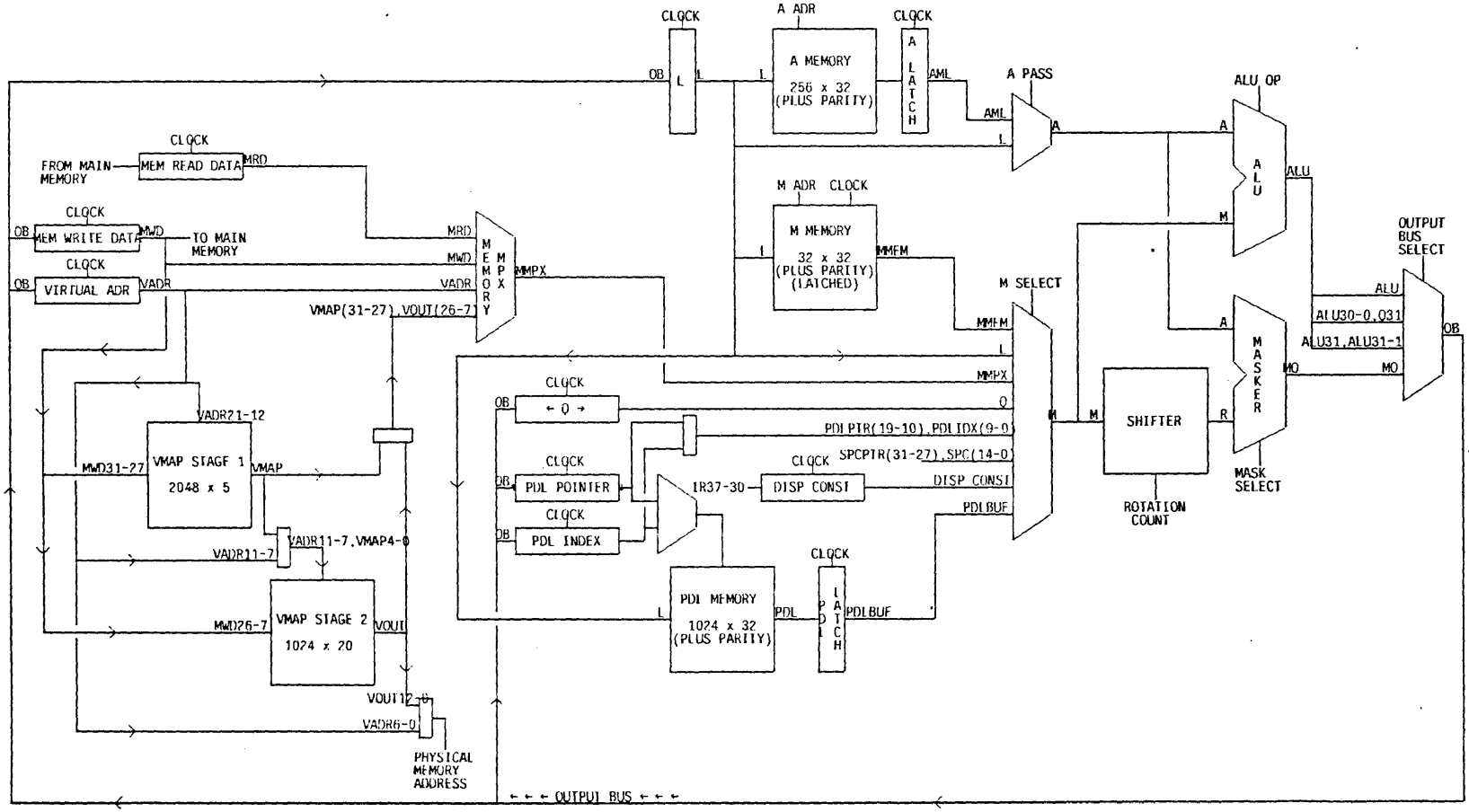
The clock can be stopped at the end of either phase, for several reasons. Usually the clock stops at the end of the read phase, referred to as "wait". This leaves the clock in the inactive high state, and leaves the latches on the memories open. The clock can wait because the machine was commanded to halt by the diagnostic interface, because a single-step commanded by the diagnostic interface has completed, because of an error such as a parity error, because of the statistics counter overflowing, or because of a memory-wait condition. This latter condition happens if a main memory cycle is initiated while a previous cycle is still in progress, or if the program calls for the result of a main memory read before the bus controller has granted the bus access needed to perform that read cycle. During a clock wait, the processor clock stops, but the clock to the rest of the system (the bus interface and XBUS devices), continues to run, allowing them to operate. When the processor finishes waiting the processor clock starts up in synchrony with the external clock.

The clock can also stop at the end of the write phase, referred to as "hang". This is used only during memory reads. If the processor calls for the result of a read which is in progress but has not yet completed, it hangs until the data has arrived from memory and sufficient time has passed for the data to flow through the data paths and appear on the output bus. This is also sufficient time for the parity of the data to be checked. In the case of a hang, both clocks stop, which allows them to restart synchronously without any extra delay. In this way, the speed of the processor is adjusted

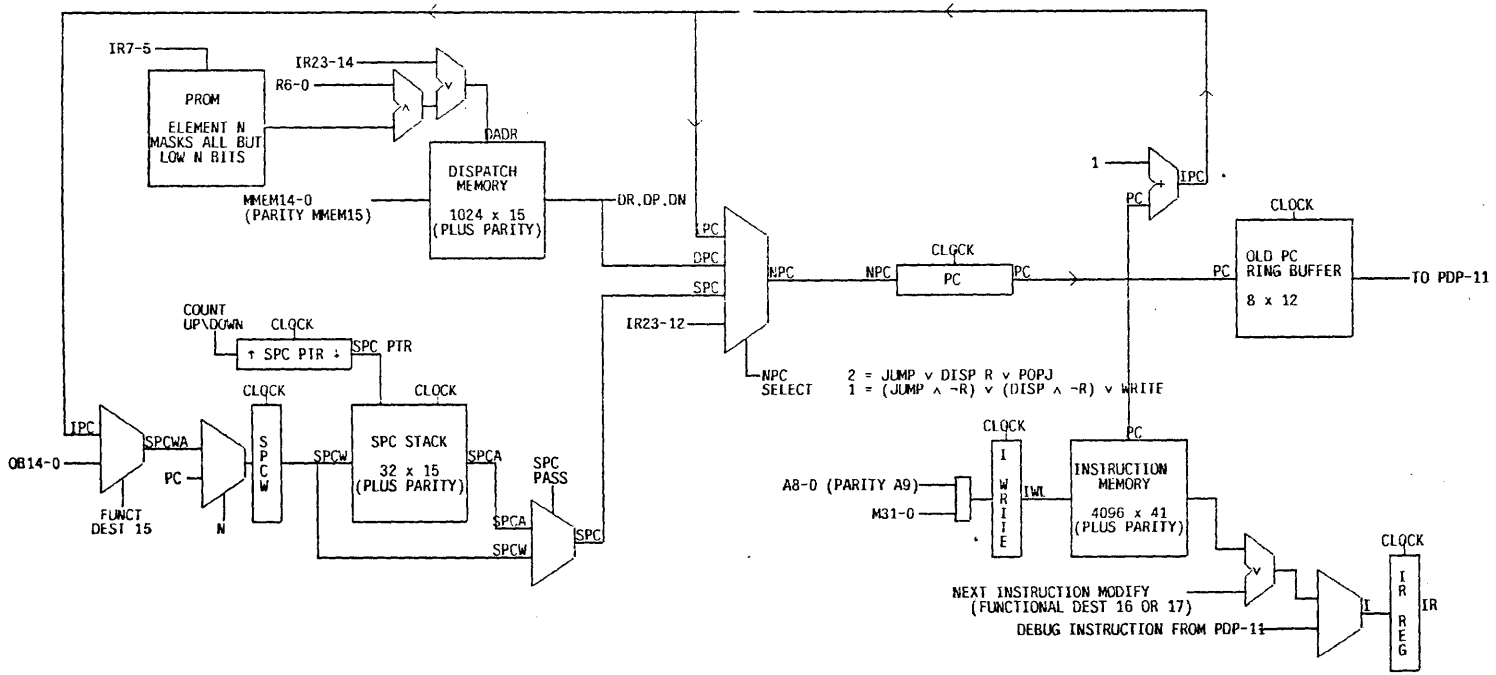
to exactly match the speed of the memory.

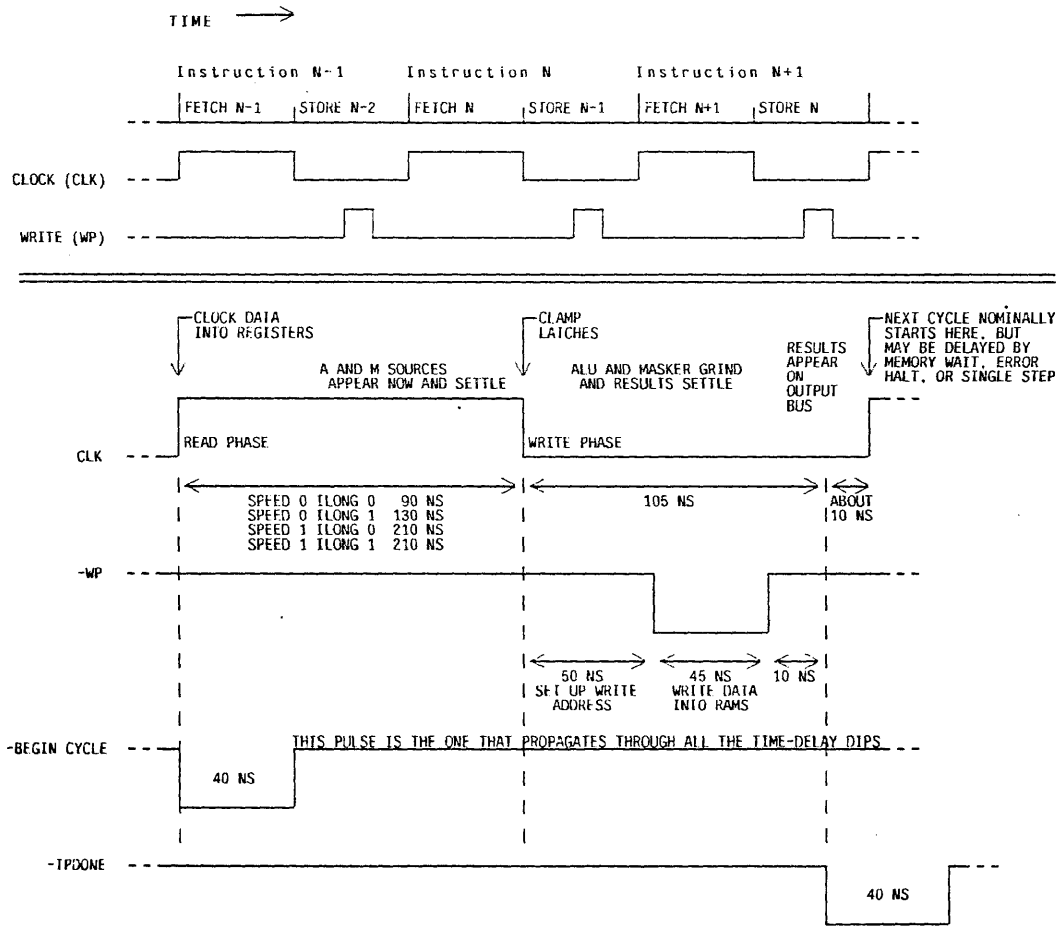
<<picture CHODTM goes here - use SCNV>>

<<picture CHODT1 goes here - use SCNV>>



CONS	MAIN DATA PATHS	17-OCT-1979 08:20	AI:LMDOC; CHOD1


IR7-5
IR23-14
R6-0
PROM
ELEMENT N
MASKS ALL BUT
LOW N BITS
DADR
DISPATCH
MEMORY
1024 x 15
(PLUS PARITY)
MMEM14-0
(PARITY MMEM15)
DR.DP.DN
1
IPC
PC
CLOCK
OLD PC
RING BUFFER
8 x 12
TO PDP-11
LPC
DPC
SPC
NPC
NPC
CLOCK
PC
PC
PC
IR23-12
NPC
SELECT
2 = JUMP v DISP R v POPJ
1 = (JUMP ∧ ¬R) v (DISP ∧ ¬R) v WRITE
COUNT
UP\DOWN
CLOCK
↑ SPC PTR ↓
SPC PTR
CLOCK
CLOCK
IPC
SPCWA
OB14-0
PC
FUNCT
DEST 15
N
SPCW
SPC STACK
32 x 15
(PLUS PARITY)
SPCA
SPC
PASS
SPCA
SPCW
SPC
CLOCK
PC
A8-0 (PARITY A9)
M31-0
IWL
INSTRUCTION
MEMORY
4096 x 41
(PLUS PARITY)
CLOCK
NEXT INSTRUCTION MODIFY
(FUNCTIONAL DEST 16 OR 17)
DEBUG INSTRUCTION FROM PDP-11
I
IR
IR



All times shown are nominal and subject to tweaking.
Diagram is not precisely to scale.

Accessing Memory

Access to main memory is accomplished through use of several functional sources and destinations. These perform three functions; first, they allow access to two registers, VMA (virtual memory address) and MD (memory data). Secondly, they can initiate memory operations. Thirdly, they can wait for a memory operation to be completed. Actually, this facility is not just for accessing main memory; it is used to access any device on the Xbus or the Unibus, which includes not only memory but peripheral equipment. For simplicity the term "memory" will be used, however.

There are eight functional destinations associated with the memory system. Four of these load data into the VMA, the other four load data into the MD. Each group of four consists of one with no other side effects, one which starts a read cycle, one which starts a write cycle, and one which writes into the virtual address map.

In a memory read operation, data from memory is placed in the MD register when it arrives, and can then be picked up by the program (using a functional source). In a memory write operation, the program places the data to be written into the MD register (by using a functional destination), whence it is passed to the memory.

The VMA register contains the virtual address of the location to be referenced. This is 24 bits long; the high 8 bits of the register exist but are ignored by the hardware. The VMA contains a "virtual" address; before being sent to the memory it is passed through the "map", which produces a 22 bit physical address, controls whether permission for the read or write operation requested is allowed, and remembers 8 bits which the software (microcode) can use for its own purposes.

Except when starting a memory cycle, the address to be mapped comes from bits <23-0> of the MD register, rather than the VMA register. The reason for this is to simplify the use of the map for checking what "space" a pointer being read from or written into memory points at, a frequently-needed operation in the Lisp machine garbage-collection algorithm.

The map consists of two scratchpad memories. The First Level Map contains 2048 5-bit locations, and is addressed by bits <23-13> of the VMA or MD. The Second Level map contains 1024 24-bit locations, and is addressed by the concatenation of the output from the First Level Map and bits <12-8> of the VMA or MD. The virtual address space consists of 2048 blocks, each containing 32 pages. Each page contains 256 words (of 32 bits, of course). Each block of virtual address space has a corresponding location in the First Level Map. Locations in the Second Level Map are not permanently allocated to particular addresses; instead, the First Level Map location for a block of virtual addresses indicates where in the Second Level Map those addresses are currently described. The Second Level Map contains sufficient space to describe 32 blocks, so at any given time most blocks must be described as "no information available." This done by reserving the last 32 locations in the Second Level Map for this purpose and filling them with "no information available" page descriptors; most First Level Map locations will point here.

The output of the Second Level Map consists of:

```
MAP<23>    = access permission
MAP<22>    = write permission
MAP<21-14> = available to software.  Note that bits 15 and 14 can
             be tested by the DISPATCH instruction.
MAP<13-0>  = physical page number
```

The physical address sent to memory is the concatenation of the physical page number and bits 7-0 of the VMA.

The two maps can be read by putting an appropriate address in the MD, and reading the functional source MEMORY-MAP-DATA (11):

```
MAP<31> = 1 if the most recent memory cycle was not performed because it
          was an attempt to write without write permission, i.e. a 1 in
          bit 22 of the second level map.
MAP<30> = 1 if the most recent memory cycle was not performed because there
          was no access permission, i.e. a 1 in bit 23 of the second level map.
          MAP<30> is 0 if no access fault exists, although a write fault may
          exist.  Note that bits <31-30> apply to the last attempted memory
          cycle, and have nothing to do with the map locations addressed by
          the contents of MD.
MAP<29> = 0 always.
MAP<28-24> = First Level Map
MAP<23-0> = Second Level Map
```

The maps can be written by using one of the functional destinations VMA-WRITE-MAP (23), MEMORY-DATA-WRITE-MAP (33). The MD supplies the address of the map location to be written, and the VMA supplies the data to be written, and tells which level of the map is being written. One register must be set up in a previous instruction, the other is written via the functional destination, and the actual writing into the map happens on the following cycle. There is no pass-around path and no latch for the map, so the following instruction must not use it.

The first level map is written from bits <31-27> of the VMA, if VMA<26> is a 1. (These are not the same bits as it reads into when using the MEMORY-MAP-DATA functional source.) The second level map is written from VMA<23-0>, if VMA<25> is a 1. Note that when writing the second level map the first level map supplies part of the address, and must have been written previously. Therefore it is not useful to write both at the same time, although it is possible to set both bits to 1.

Main memory operations are initiated by using one of the functional destinations VMA-START-READ (21), VMA-START-WRITE (22), and MEMORY-DATA-START-WRITE (32). There is also MEMORY-DATA-START-READ (31), but it is probably useless. In the case of a write, the VMA supplies the address and the MD supplies the data, so one register must be set up in advance and the other is set up by the functional destination that starts the operation. A main memory read can also be started by the

macro instruction-stream hardware, described later.

The register named (VMA or MD) is loaded with the result of the instruction (from the Output Bus) at the end of the cycle during which that instruction is executed. During the following cycle, the map is read. The instruction executed during this cycle should be a JUMP instruction which checks for a page fault condition. At the end of this cycle, if no page fault occurs, the memory operation begins. The processor continues executing while the memory operation happens, but if any operation which conflicts with the memory being busy is attempted, the machine waits or hangs until the memory operation has been completed. Such references include asking for the results of a read cycle by using the MEMORY-DATA (12) functional source, using any functional destination that refers to the VMA, MD, or MAP, or attempting to start a read cycle via the instruction-stream hardware.

The presence or absence of a page fault is remembered until the next time a memory cycle is started, so it is not strictly necessary to check for page fault immediately after starting a cycle, but is good practice.

The MEMORY-DATA-START-WRITE destination is useful for doing the second half of a read-followed-by-write operation, since the correct value is still in the VMA. Note that it is still necessary to check for a page fault after starting the write, since you may have read permission but not write permission.

There is a feature by which main memory parity errors can be trapped to the microcode. A bit in the diagnostic interface controls whether or not this is enabled. When the MEMORY-DATA functional source is used, and the last thing to be loaded into the MD was data from memory which had even parity, a main memory parity error has occurred. If trapping is enabled, the current instruction is NOPed and a CALL transfer to location 0 is forced. The following instruction is also NOPed. The trap routine must use the OPC registers to determine just where to return to if it plans to return, since if a transfer operation was in progress the address pushed on the SPC stack by the trap may have nothing to do with the address of the instruction which caused the trap. This is also true of the error-handler for microcode-detected programming errors. If a main memory parity error occurs, and trapping is not enabled, the machine halts if error-halting is enabled, just as it does in response to a parity error in an internal memory.

When using semiconductor main memory, which has single-bit error correction, a parity error trap indicates that an uncorrectable multiple-bit error occurred. Single-bit errors are corrected automatically by the hardware, and cause an interrupt so that the processor may, at its leisure, log the error and attempt to rewrite the contents of the bad location.

The Instruction-Stream Feature

The CADR processor contains a small amount of hardware to aid in the interpretation of an instruction stream which comes in units smaller than the CADR word size. For example, the Lisp-machine macrocompiled instruction set uses 16-bit units. The hardware speeds up both fetching and decoding of instructions by relieving the microcode of some routine bookkeeping.

Both 8-bit (byte) and 16-bit (halfword) instructions are supported, depending on a mode bit (bit 29 of the "Interrupt Control" register, functional destination 2.) The hardware decides when it is time to fetch a new main-memory word, containing the next 2 or 4 units of the instruction stream, and alters the flow of microprogram control. The hardware provides a feature by which the rotator control can be made to select the current unit of the instruction stream; this is used when dispatching on the instruction being interpreted, and when extracting fields of the instruction via the BYTE microinstruction.

There is a 26-bit register called the Location Counter (LC), which can be read by functional source 13 and written by functional destination 1. It always contains the address of the next instruction stream unit, in terms of 8-bit bytes. In halfword mode LC<0> is forced to zero. The LC is capable of counting by 1 or 2 (depending on byte vs. halfword mode) and has a special connection to the VMA; the VMA is loaded from the LC, divided by 4, when an instruction-fetch occurs.

The high 6 bits of functional source 13 are not part of the LC per se, but contain various associated status, as follows:

31 Need Fetch. This is 1 if the next time the instruction stream is advanced, a new word will be fetched from main memory. This is a function of the low 2 bits of LC, of byte mode, and of whether the LC has been written into since an instruction word was last fetched from main memory.

30 not used, zero.

29 LC Byte Mode. 1 if the instruction stream is in 8-bit units, 0 if it is in 16-bit units. This reflects bit 29 of the Interrupt Control register.

28 Bus Reset. This reflects bit 28 of the Interrupt Control register, which is set to 1 to reset the bus interface, the Unibus, and the Xbus.

27 Interrupt Enable. 1 if external interrupt requests are allowed to contribute to the JUMP condition. This reflects bit 27 of the Interrupt Control register.

26 Sequence Break. 1 if a sequence break (macrocode interrupt signal) is pending. This flag does nothing except contribute to the JUMP condition. This reflects bit 26 of the Interrupt Control register.

Bit 14 of the SPC stack is used to flag the return address containing it as the address of the main instruction-interpretation loop. The hardware recognizes a RETURN transfer with SPC<14>=1 as completing the interpretation of one instruction and initiating the interpretation of the next. The instruction stream will be advanced to its next unit (byte or halfword) in the cycle following the RETURN transfer. (It is delayed one cycle for obscure timing reasons.) This cycle is free to also execute a useful microinstruction, provided it does not use the LC, VMA, MD, and associated hardware.

Advancing the instruction stream increments the LC, by 1 or 2. If a new word needs to be fetched from main memory, the unincremented LC, divided by 4, is transferred to the VMA and a read cycle is started. A fetch can be required either because the LC points at the first unit of a word or because the LC has been modified since the last instruction stream advance (a branch occurred). It is legal for the instruction which does the RETURN transfer to modify the LC, and a fetch will always be required. If no fetch is required, the RETURN transfer is altered by forcing SPC<1> to 1, skipping over two microinstructions which, in the fetch case, check for a page fault (or interrupt or sequence break) and transfer the new instruction stream word from MD into a scratchpad location.

The instruction stream can also be advanced by a DISPATCH instruction with bit 24 set. In this case, no alteration of the SPC return address occurs. The dispatch should check the NEEDFETCH signal, which is available as bit 31 of the LC functional source, to determine whether a new word is going to be fetched. If a fetch occurs, the DISPATCH should call a subroutine to check for page fault and transfer the new instruction stream word from MD to a scratchpad location. If no fetch occurs, the DISPATCH should drop through. The instruction after the DISPATCH may then operate on the next unit of the instruction stream. This feature is provided to facilitate the use of multi-unit instructions.

The remaining hardware associated with the instruction stream feature implements miscellaneous function 3, which alters the M-rotate field to select the current unit of the instruction stream from the current word, which should be supplied as the M-source. This applies to any operation which uses the rotator: BYTE instructions, DISPATCH instructions, and JUMP instructions which test a bit. The instruction should be coded for the unit (byte or halfword) at the right-hand end of the word. In half-word mode, IR<4> is XOR'ed with LC<1> to produce the high-order bit of the rotate count. In byte mode, IR<4> is XOR'ed with (LC<1> XOR LC<0>), and IR<3> is XOR'ed with LC<0>. The effect, since the LC always has the address of the next instruction, and the bits are numbered from right to left, is as desired. In halfword mode, the low half of the M source is accessed for the even instruction, when LC<1>=1, and the high half is accessed for the odd instruction, when LC<1>=0.

Multiplication, Division, and the Q register

The Q register is provided in CADR primarily for multiplication and division. It is occasionally useful for other things because it is an extra place to put the results of an ALU instruction, and because it can be used to collect the bits which are shifted out when the OUTPUT-SELECTOR-RIGHTSHIFT-1 operation is used in an ALU instruction.

The Q register is controlled by two bits (IR<1-0>) in the ALU instruction. The operations are do nothing, shift it left, shift it right, and load it from the output of the ALU. (It loads from the ALU rather than the Output Bus for electrical reasons.) When the Q register shifts left, Q<0> receives -ALU<31>, the complement of the sign of the ALU output. When the Q register shifts right, Q<31> receives ALU<0>, the low bit of the ALU output. The Q register is also connected to the Output Bus shifter; when the Output Bus is shifted left, OB<0> receives Q<31>, the sign of the Q. These interconnections are dictated by the needs of multiplication and division.

Multiplication in CADR is a simple, 1 bit at a time, shift-and-add affair. The hardware provides a conditional-ALU operation, MULTIPLY-STEP, which is ADD if Q<0>=1, and SETM otherwise. This is used in combination with SHIFT-Q-RIGHT and OUTPUT-SELECTOR-RIGHTSHIFT-1. Initially the multiplicand is placed in an A-scratchpad location and the multiplier is placed in Q. 32 MULTIPLY-STEP operations are executed; as Q shifts to the right each of the bits of the multiplier appear in Q<0>. If the bit is 1, the multiplicand gets added in. The results of each operation go into an M-scratchpad location, which is fed back into the next step. The low bit of each result is shifted into Q. Thus, when the 32 steps have been completed, the Q contains the low 32 bits of the product, and the M-scratchpad location contains the high 32 bits.

This algorithm needs a slight modification to deal with 2's complement numbers. The sign bit of a 2's complement number has negative weight, so in the last step if Q<0>=1, i.e. the multiplier is negative, a subtraction should be done instead of an addition. The hardware does not provide this, so instead we do a subtraction after the last step, which is adding and then subtracting twice as much, which has the effect of subtracting. Note that this correction only affects the high 32 bits of the product, and can be omitted if we are only looking for a single-precision result. Consider the following code. (The CONSLP assembler format used is explained later in this document.)

```
; Multiply Subroutine.  A-MPYR times Q-R, low product to Q-R, high to M-AC.

MPY       ((M-AC) MULTIPLY-STEP M-ZERO A-MPYR)          ;Partial result = 0 in first step
(REPEAT 30. ((M-AC) MULTIPLY-STEP M-AC A-MPYR))         ;Do 30 steps
          (POPJ-IF-BIT-CLEAR-XCT-NEXT                   ;Return after next if A-MPYR positive
                    (BYTE-FIELD 1 0) Q-R)
         ((M-AC) MULTIPLY-STEP M-AC A-MPYR)             ;The final step
          (POPJ-AFTER-NEXT
           (M-AC) SUB M-AC A-MPYR)                      ;Correction for negative multiplier
```

```
(NO-OP)                                    ;Jump delay
```

To multiply numbers of less than 32 bits is also possible. With the same initial conditions, after n steps the high n bits of the Q contain the low n bits of the product, and the remaining bits of the product are in the low bits of the M-scratchpad location. Two BYTE instructions can be used to extract and combine these bits to produce a right-adjusted product, if the numbers are unsigned.

Division is a little more complex than multiplication. It too goes a bit at a time, using a non-restoring algorithm which either adds or subtracts at each stage. The basic idea is to keep subtracting the divisor from the dividend, shifted over by different amounts, as in long-division by hand. If the subtraction produces a positive result, it "goes in" and a quotient bit of 1 is produced. If the subtraction produces a negative result, it "fails to go in" and a quotient bit of 0 is produced. Instead of backing up and not doing the subtraction, we set a flag that too much has been subtracted, and add instead the next time. This works since the weight of the divisor on the next step is half as much, and B-(A/2) = B-A+(A/2). The "flag" is simply the complement of the quotient bit produced, except for the first step when the flag must be forced to OFF.

Division does not handle 2's complement numbers as easily as multiplication does. The algorithm essentially requires all positive numbers, however the hardware automatically takes the absolute value of the divisor by interchanging addition and subtraction if the divisor is negative. It is up to the microcode to make the dividend positive beforehand, and to determine the correct signs for the quotient and remainder afterward. The sign of the quotient should be the XOR of the signs of dividend and divisor. The sign of the remainder should be the same as the sign of the dividend.

Initially the positive dividend is in the Q register and the signed divisor is in an A-scratchpad location. Appropriate conditional-ALU operations are used in conjunction with the SHIFT-Q-LEFT and OUTPUT-SELECTOR-LEFTSHIFT-1 functions. An M-scratchpad location receives the result of each step, and is fed back to the next step. This location initially contains the high 32 bits of the double-length dividend, or 0 if the dividend is single-precision. At each step, the OUTPUT-SELECTOR-LEFTSHIFT-1 operation brings the high bit of the Q into the low bit of the M-scratchpad, bringing up another bit of the dividend. At each step, the complement of the sign of the ALU output represents a bit of the quotient and is shifted into the low end of Q. After 33 steps, Q contains the positive quotient (which is why it is called the Q-for-quotient register). The reason why it takes 33 steps rather than 32 is a little difficult to explain. The quotient bit produced by the first step, if 1, indicates "divide overflow", and is not really part of the quotient. When using a single-precision dividend, "divide overflow" can only happen if the divisor is zero, since the initial operation is zero minus the absolute value of the divisor, which is negative unless the divisor is zero.

What is left of the dividend after all the subtractions is the positive remainder. The last step does not use OUTPUT-SELECTOR-LEFTSHIFT-1, so that the M-scratchpad will receive the remainder rather than the remainder times 2. If the "too much has been subtracted" flag is set, it is necessary to do one final addition to correct

the remainder. This addition simply undoes the previous subtraction, not also doing a new subtraction, because of the omission of the left shift.

The ALU operations for division are:

DIVIDE-STEP The conditional add or subtract described above, SHIFT-Q-LEFT, and OUTPUT-SELECTOR-LEFTSHIFT-1. Q<0>=0 serves as the "too much has been subtracted" flag.

DIVIDE-FIRST-STEP Identical to DIVIDE-STEP except that the "too much has been subtracted" flag is forced to be off.

DIVIDE-LAST-STEP Identical to DIVIDE-STEP except that the OUTPUT-SELECTOR-LEFTSHIFT-1 is omitted.

DIVIDE-REMAINDER-CORRECTION-STEP The conditional add or subtract logic is used, except subtract is turned into SETM by invoking part of the multiply logic. The exchanging of add and subtract if the divisor is negative then applies, doing the right thing. No shifting occurs and Q is unchanged.

Division of numbers smaller than 32 bits can be accomplished in less than 33 steps by sufficiently careful shifting of the inputs and outputs.

To illustrate how it all fits together, and show how to do the sign-correction, here is the code for 32-bit division, with a double-precision dividend, in the CONSLP format explained later in this document:

```
; Division Subroutine.
; M-AC and M-1 are the high and low words of the dividend, respectively.
; M-2 is the divisor.  The quotient is in M-AC, the remainder in M-1.

DIV     (JUMP-GREATER-OR-EQUAL M-AC A-ZERO DIV1)                ;Check for negative dividend
        (JUMP-NOT-EQUAL-XCT-NEXT M-1 A-ZERO DIV0)               ;If so, change sign
       ((M-1 Q-R) SUB M-ZERO A-1)
        ((M-AC) SUB M-AC (A-CONSTANT 1))                ;Borrow from high if low is zero
DIV0    ((M-AC) SETCM M-AC)                             ;1's complement high dividend
        (CALL DIV2)                                     ;Now, call positive-dividend case
        (POPJ-AFTER-NEXT (M-1) SUB M-ZERO A-1)          ;Make the remainder negative,
       ((M-AC) SUB M-ZERO A-AC)                         ;and change the sign of the quotient

; Divide routine for positive dividend.
DIV1    ((Q-R) M-1)                                     ;Low dividend to Q-R
DIV2    ((M-1) DIVIDE-FIRST-STEP M-AC A-2)              ;First division step
        (JUMP-IF-BIT-SET (BYTE-FIELD 1 0) Q-R DIVIDE-OVERFLOW) ;Error check
(REPEAT 31. ((M-1) DIVIDE-STEP M-1 A-2)                 ;Middle division steps
        ((M-1) DIVIDE-LAST-STEP M-1 A-2)                ;Final step, quotient in Q-R
        ((M-1) DIVIDE-REMAINDER-CORRECTION-STEP M-1 A-2) ;M-1 gets remainder
        ((M-AC) Q-R)                                    ;Extract quotient from Q-R
        (POPJ-AFTER-NEXT                                ;Return after next, but if
         POPJ-GREATER-OR-EQUAL M-2 A-ZERO)              ; divisor is negative,
```

```
((M-AC) SUB M-ZERO A-AC)                ; change sign of quotient
```

The Bus Interface

The Bus Interface connects the CADR machine to two busses, the Unibus and the Xbus. The Unibus is a regular pdp11 bus, used to attach peripheral devices, especially commercial devices designed for the PDP11 line. The Xbus is a 32-bit bus used to attach memory and high-performance peripheral devices, such as disk. The bus interface also includes the diagnostic interface, which allows a unibus operator, such as a pdp10, a pdp11, or another lisp machine, to control the operation of the machine, hardware to pass interrupts from the Unibus and the Xbus to the processor, the logic which arbitrates the Xbus, and the logic which arbitrates the Unibus in the absence of a pdp11 on that bus.

The Bus Interface allows the CADR machine to access memory on the Xbus and devices on the Unibus, allows independent devices on the Xbus to access the Xbus (only), and allows Unibus devices to access Xbus memory (through a map since the Unibus address space is not big enough.) Buffering is provided when the Unibus accesses the Xbus, to convert a 32-bit word into a pair of 16-bit words.

The CADR machine sees a 22-bit physical address space of 32-bit words. The top 128K of this, locations 17,400,000-17,777,777, reference the Unibus. Each 32-bit word has a 16-bit Unibus word in bits 0-15, and zero in bits 16-31. There is no provision for using byte addressing on the Unibus, nor for read-pause-write cycles. The 128K immediately below the Unibus, locations 17,000,000-17,377,777, are reserved for Xbus I/O devices. Locations 0-16,777,777 are for Xbus memory.

The bus interface includes a number of Unibus registers which control its various functions:

Spy Feature

Unibus locations 766000-766036 are used for the Spy feature, which is described in detail elsewhere. These locations read and write various internal signals in the CADR machine, and provide the necessary hook for microcode loading and diagnostics.

Two-Machine Lashup

Two bus interfaces may be cabled together with a single 50-wire flat cable for maintenance purposes. One machine, the debugger, is able to perform reads and writes on the other machine's, the debuggee's, Unibus. Through registers on the Unibus (such as the Spy feature), the debuggee may be diagnosed and exercised. By using the debuggee's Unibus map (described below), the debuggee's Xbus can be exercised. The following locations on the debugger's Unibus control this feature:

766100 Reads or writes the debuggee-Unibus location addressed by the registers below.

766114 (Write only) Contains bits 1-16 of the debuggee-Unibus address to be accessed. Bit 0 of the address is always zero.

766110 (Write only) Contains additional modifier bits, as follows. These bits are reset to zero when the debuggee's Unibus is reset.

1 Bit 17 of the debuggee-Unibus address.
2 Resets the debuggee's Unibus and bus interface. Write a 1 here then write a 0.
4 Timeout inhibit. This turns off the NXM timeout for all Xbus and Unibus cycles done by the debuggee's bus interface (not just those commanded by the debugger).

766104 (Read only) These contain the status for bus cycles executed on the debuggee's busses. These bits are cleared by writing into location 766044 (Error Status) on the debuggee's Unibus. They are not cleared by power up. The bits are documented below under "Error Status".

Error Status

766044 Reading this location returns accumulated error status bits from previous bus cycles. Writing this location ignores the data written and clears the status bits. Note that these bits are not cleared by power up.

1 Xbus NXM Error. Set when an Xbus cycle times out for lack of response.
2 Xbus Parity Error. Set when an Xbus read receives a word with bad parity, and the Xbus ignore-parity line was not asserted. Parity Error is also set by Xbus NXM Error.
4 CADR Address Parity Error. Set when an address received from the processor has bad parity. Indicates trouble in the communication between the processor and the bus interface.
10 Unibus NXM Error. Set when a Unibus cycle times out for lack of response.
20 CADR Parity Error. Set when data received from the processor has bad parity. Indicates trouble in the communication between the processor and the bus interface.
40 Unibus Map Error. Set when an attempt to perform an Xbus cycle through the Unibus map is refused because the map specifies invalid or write-protected.

The remaining bits are random (not necessarily zero).

Interrupts

The bus interface allows the CADR machine to field interrupts on the Unibus, if no pdp11 is present. If a pdp11 is present, its program can forward interrupts to the CADR machine in a transparent way. The Xbus also can interrupt the CADR machine. The following Unibus locations control interrupts and the Unibus arbitrator:

766040 Reading this location returns interrupt status, as follows:

1 Disable Interrupt Grant. If this is set, the Unibus arbitrator will not grant BR4, BR5, BR6, and BR7 requests. It will continue to grant NPR

requests. Powers up to zero.

2 Local Enable (read only). 1 means that the bus interface is arbitrating the Unibus. 0 means that a pdp11 is present on the bus and is doing the arbitration.

1774 Bits 9-2 contain the vector address of the last Unibus interrupt accepted by the bus interface or simulated by the pdp11 program.

2000 Enable Unibus Interrupts. A 1 here causes bit 15 (Unibus interrupt) to be set when the bus interface accepts a Unibus interrupt. This bit is not reset by power-up.

4000 Interrupt Stops Grants. A 1 here causes bit 0 (Disable Interrupt Grant) to be set when the bus interface accepts a Unibus interrupt, thus preventing further interrupts until the CADR machine has processed the first interrupt. This bit is not reset by power-up.

30000 Bits 13-12 are the "interrupt level" for purposes of Unibus granting. The mapping to normal pdp11 levels is: 0->0, 1->4, 2->5, 3->6. To simulate level 7, turn on Disable Interrupt Grant. These bits are not reset by power-up.

40000 Xbus Interrupt (read only). This bit is the interrupt-request line on the Xbus.

100000 Unibus Interrupt. A 1 indicates that a Unibus interrupt has been accepted by the bus interface or simulated by a pdp11 program, and is awaiting processing by the CADR program. This bit clears on power-up. Note that the interrupt-request signal to the CADR machine is the OR of bits 14 and 15.

766040 Writing this location writes into bits 0 and 10-13 (mask 36001) of the above register. This is used to change the "interrupt level" and to re-enable acceptance of Unibus interrupts after processing an interrupt.

766042 Writing this location writes into bits 2-9 and 15 (mask 101774) of the above register. This is used to simulate Unibus interrupts and to clear bit 15 (Unibus Interrupt) after processing an interrupt.

Locations between 766040 and 766136 not mentioned above are duplicates of other locations, and should not be used.

Unibus Map

Unibus locations 140000-177777 are divided into 16 pages which can be mapped anywhere in Xbus physical address space. Each page is 512 16-bit words or 256 32-bit words long, the same size as the pages of the CADR virtual memory. The first 8 pages can be addressed by a pdp11, while the second 8 are hidden under the pdp11 I/O space. The Unibus map is intended to be used both as a diagnostic path to the Xbus and for operating Unibus peripherals that access memory.

Each Xbus location occupies 4 Unibus byte addresses. It takes two 16-bit Unibus cycles to read or write one 32-bit Xbus location. 16 buffers (one for each page) are provided to hold the data between the two Unibus cycles. As long as each page is only in

use by a single bus-master, the right thing will happen.

An additional feature is that writing an Xbus address of 17400000 or higher through the Unibus map writes into CADR's MD register. This provides a 32-bit parallel data path into the processor for diagnostic purposes. These Xbus addresses are otherwise unusable, because they are used by the processor to address the Unibus.

Unibus locations 766140-766176 contain the 16 mapping registers. Note that these power up to random contents, and should be cleared by an initialization routine. The bit layout is:

100000 Bit 15 is the map-valid bit. If this is 0, this mapping register is not set up, and will not respond to the Unibus; NXM timeout will occur and an Error Status bit will be set.

40000 Bit 14 is the write-permit bit. If this is 0, this mapping register will not respond to Unibus writes; NXM timeout will occur and an Error Status bit will be set.

37777 Bits 13-0 contain the Xbus page number. These bits are concatenated with bits 9-2 of the Unibus address to produce the mapped Xbus address.

The Xbus

The Xbus is the standard 32 bit wide data bus for the CADR processor. Main memory and high speed peripherals such as the disk control and TV display are interfaced to the Xbus. Control of the Xbus is similar to the Unibus, in that transfers are positively timed and (as far as the devices are concerned) asynchronous. The bus is terminated at both ends with resistive pullups of 390 ohms to ground and 180 ohms to +5 volts, for an effective 123 ohm termination to +3.42 volts. At ground, each termination draws 28 ma. for a total load of 56 ma. The bus is open collector, and may be driven with any device capable of handling the 56 ma. load. The recommended driver is the AMD 26S10, which also provides bus receivers.

A typical read cycle begins with placing the address for the transfer on the -XADDR lines and the parity of the address on the -XBUS.ADDRPAR line. The -XBUS.RQ line is then lowered, initiating the request. The responding device places the requested data on the 32 -XBUS lines and the parity of the data on the -XBUS.PAR line. Should it not be convenient for the device to produce parity (as in the case of I/O registers), the device may assert -XBUS.IGNPAR to notify the bus master that the transfer should not be checked for correct parity. The responding device then asserts -XBUS.ACK, which remains asserted until the -XBUS.RQ signal is removed by the master.

Write requests proceed identically, except that the master asserts -XBUS.WR and the data to be written on the -XBUS lines along with the address lines. All bus masters are required to produce good parity data on writes.

Deskewing delays are the responsibility of the bus master. In particular, it is the responsibility of the bus master to assert good address, write, and data lines 80 ns. prior to asserting -XBUS.RQ, and these lines must be held until the -XBUS.ACK signal drops in response to the master dropping -XBUS.RQ. Responding devices are allowed to assert -XBUS.ACK at the same time they drive read data onto the -XBUS lines. Thus, masters should delay 50 ns. after receiving -XBUS.ACK before dropping -XBUS.RQ and strobing the data. Responding devices are required to drop -XBUS.ACK immediately after -XBUS.RQ is no longer asserted.

Normal bus master arbitration between the CADR processor and the Unibus requests is handled by the bus interface. Devices on the Xbus which must become bus master, such as the disk control, do so by asserting the -XBUS.EXTRQ signal. When the bus becomes free, the bus interface responds by asserting -XBUS.EXTGRANT. This signal is daisy chained between bus master devices on the Xbus, coming in on the -XBUS.EXTGRANT.IN pin and leaving on the -XBUS.EXTGRANT.OUT pin. Within each device, the decision is made whether or not to pass the grant onto the next device. Unlike the Unibus structure, the decision on whether to pass grant and the act of becoming bus master happen synchronously with a master clock signal distributed on the -XBUS.SYNC line.

When a device initiates a request, it immediately asserts -XBUS.EXTRQ. At the falling edge of -XBUS.SYNC it clocks the request signal into a D flip flop which we will call REQ.SYNC. When -XBUS.EXTGRANT.IN goes low, the device asserts -XBUS.EXTGRANT.OUT unless it has either the REQ.SYNC flip flop set, or is already the bus master. At the next falling edge of -XBUS.SYNC the device which has both -XBUS.EXTGRANT.IN and REQ.SYNC set becomes bus master. The device should

immediately assert -XBUS.BUSY and may immediately begin asserting address lines for a transfer. -XBUS.BUSY may be dropped asynchronously, after the slave device drops -XBUS.ACK in response to the master's request.

The -XBUS.EXTGRANT.IN signal must be terminated with a resistive pullup of 180 ohms to +5 volts within each device which does not simply connect it to -XBUS.EXTGRANT.OUT.

XBUS Signal review:

Data lines:

-XBUS<31:0>	32 data lines, low when data is a one.
-XBUS.PAR	Parity of the 32 data lines. Required for writes.
-XBUS.IGNPAR	Ignore parity signal, may be asserted by any device for a read.

Address lines:

-XADDR<21:0>	22 address lines, low for address bit a one.
-XADDR.PAR	Odd parity for the address.

Cycle control lines:

-XBUS.RQ	Asserted by the master to request a read or write Minimum of 80 ns following stable -XADDR, -XBUS.WRITE, and -XBUS data.
-XBUS.ACK	Asserted by the slave in response to -XBUS.RQ No delay necessary following assertion of good read data.
-XBUS.WR	Asserted by the master during a write cycle.

Mastership control lines:

-XBUS.BUSY	Asserted when a device other than the bus interface is bus master. Only the bus interface examines this line. Asserted on a -XBUS.SYNC clock edge, dropped asynchronously after -XBUS.ACK drops.
-XBUS.EXTRQ	Asserted when a device other than the bus interface wishes to become bus master. Asserted asynchronously, may be removed asynchronously after the device becomes master, but before dropping -XBUS.BUSY.
-XBUS.EXTGRANT.IN	The daisy-chained mastership grant signal. Must be pulled up to +5V with a 180 ohm resistor. -XBUS.EXTGRANT.OUT Asserted initially by the bus interface, synchronously with the -XBUS.SYNC edge. The signal may be subject to synchronizer lossage, since it is a clocked version of -XBUS.EXTRQ which is not synchronous

with -XBUS.SYNC

Miscellaneous:

-XBUS.INIT — When low, resets all devices. This is low during power on and off, and when the machine is reset.

-XBUS.SYNC — Synchronization clock for mastership passing and other desired purposes. Devices become bus master synchronous with the edge of this signal. The request will normally follow the edge by 80 ns.

-XBUS.INTR — Driving this low requests an interrupt. All devices are required to initialize to a non-interrupt enable condition, and are required to have interrupt enable and disable bits which can selectively enable interrupts from that device. The "requesting interrupt" state must be readable in one of the device control register bits.

XBUS.POWER.OK — This line is HIGH when power is stable. It remains low for 3 seconds after power comes on, and goes low 3 seconds before power is turned off.

Error Checking

All internal memories in the CADR machine have parity checking. If bad parity is detected, the machine is halted, if that is enabled. The processor always completes the current instruction, and clocks the next one into the IR, before halting. This is done to simplify the timing and to ensure that it halts with the scratchpad memory latches open. It means that the data with bad parity will no longer be on the busses once the machine stops. Furthermore, one incorrect instruction will have been executed. The OPC registers can be helpful in reconstructing what must have happened.

Upon initial power-on, error halting is disabled, but it is expected that as soon as the bootstrap program has initialized all internal memories it will enable error halting.

Main memory parity is checked and can either halt the machine, cause a microcode trap, or be ignored, depending on mode flags in the diagnostic interface.

The data paths do not have any redundant checking. When the machine is bootstrapped it runs some simple diagnostics designed to detect solid failures in the memories and data paths.

Self Bootstrapping

When the machine is powered on it resets itself and the Unibus but does not automatically start up. A bootstrap sequence can be initiated in any of several ways. The diagnostic interface can command one. The diagnostic display panel, by grounding one wire, can start one. This is intended to be connected to a push button. The bus interface can start a bootstrap by grounding one wire. The chaos network interface, if it receives a certain sequence of messages from the network, will "push the boot button." The I/O board recognizes a special set of keyboard commands (left and right control-meta) as a boot signal. The character typed along with the left-right control-meta is available to the bootstrap for selection of software options.

The bootstrap sequence starts by resetting the machine, which will halt it if it is running. It turns on RUN, which will not do anything yet since the clock is stopped. It sets the machine to its slowest speed, disables parity traps, error halts, and the statistics counter, and enables the PROM (read-only) control memory. The trailing edge of the boot signal allows the clock to start, causing a trap to microcode location 0, just like the memory parity error trap. Location 0 of the PROM receives control. It must clear all internal memories (filling them with good parity), reset the Unibus (before first using it), enable error halts, set the machine speed to its normal value, run some diagnostic checks to be sure the machine is working to some extent, load the microcode from the disk, load the initial contents of main memory from the disk, and transfer control to the normal microcode at its start address by going over the Unibus and manipulating the diagnostic interface.

If the diagnostic self-test fails, the microcode goes into a loop, and the value of the PC can be read from the diagnostic display to determine what the problem seemed to be.

Interrupts and Sequence Breaks

Interrupts are hardware signals to the microcode - typically the microcode transfers data in or out of a buffer in main memory. When the signal requires the attention of full Lisp code, a sequence break is triggered. This consists of setting a sequence-break pending flag in A-memory, and, if a defer-sequence-break flag (also in A-memory) is not set, setting the hardware sequence-break flag. This flag is tested at various convenient points such as macroinstruction fetch, and causes the microcode to turn off the flag and enter the sequence-break routines. The sequence-break flag is tested by the same jump instruction that tests for page faults and interrupts.

Interrupts can be generated by both the Xbus and the Unibus. The exact protocol is documented in the section on the bus interface.

Sequence-breaks are software signals indicating the need to run the scheduler (a Lisp program). A sequence-break suggests that the condition for which some process is waiting may have become true. The scheduler checks all processes for runnability, and also checks if it is time to perform periodic actions which are not full processes. Lisp programs can defer sequence-breaks to protect critical areas, while still allowing interrupts so that real-time response at the lowest level is preserved.

Access to virtual memory in the Lisp Machine software environment is viewed as a primitive operation. Regardless of the actual location of a memory datum, the fetch of that item is continued. This view considerably simplifies coding of the system, but imposes moderately high potential latencies in responding to sequence breaks. Interrupts are handled entirely at the microcode level, and the response time for these will be quite short.

The interrupt-control register, writable by functional destination 2, and readable in the high bits of LC (functional source 13), contains three bits relevant to interrupts. Bit <27>, INTERRUPT ENABLE, allows the external interrupt signal from the bus interface to be seen by the JUMP instruction. Bit <26>, SEQUENCE-BREAK, is the sequence-break flag which is testable by the JUMP instruction.

Bit <28>, BUS-RESET, generates a RESET signal on the Unibus (BUS INIT L) and on the Xbus (XBUS.INIT L), and resets the bus interface, when it is written 1 and then 0. The machine also resets the busses when it is powered up.

Bit <29> is used by the Instruction-Stream feature.

The Statistics Counter

The statistics counter is a 32-bit counter, which increments whenever an instruction with bit 46 = 1 is executed. When the counter overflows from -1 to 0 the machine stops, after completing execution of the instruction which caused the overflow. (The stopping is under control of an enable bit in the diagnostic interface.) Bit 46 is always 0 in instructions from the PROM.

The statistics counter can be read and written using the diagnostic interface. It provides several facilities.

It can be used for metering, to measure how many instructions are executed, possibly restricted to a certain subset of the microprogram. The microcode debugger and console program has commands to set and clear the statistics bits in areas of control memory.

It can be used for breakpointing, by setting the counter to -1 and turning on the statistics bit in those instructions which have breakpoints set on them.

It can be used to find obscure bugs, by setting the statistics bit in all locations of control memory, and setting the appropriate number in the statistics counter to cause the machine to halt just before the point where the error appears, so that it can be single-stepped through the suspect microcode.

The statistics counter is loaded from the Instruction Write Register, rather than the normal diagnostic bus, because of its 32-bit width. Effectively it loads from the M bus with a 1-cycle delay. It is probably not possible for the machine to use the statistics counter on itself, although clever ways might be found.

The Diagnostic Interface

The diagnostic interface occupies 16 Unibus addresses. It includes a 16-bit diagnostic bus which can be used to read and write various portions of the machine. There are 16 readable locations, and 8 writable locations. A readable location and a writable location at the same address have no relation to each other. The diagnostic bus is used by debugging and maintenance programs, including the "console" program, and in a few cases by the machine itself during bootstrapping.

First we will describe the readable locations. These are sometimes called the "spy feature." Naturally, most of these are somewhat meaningless if read while the machine is running.

766000 IR<15-0>. The low 16 bits of the currently-executing instruction.

766002 IR<31-16>. The middle 16 bits of the currently-executing instruction.

766004 IR<47-32>. The high 16 bits of the currently-executing instruction.

766006 not used

766010 OPC. The OPCs are described below.

766012 PC. The current program counter, which is the address of the next instruction to be executed.

766014 OB<15-0>. The low half of the output bus.

766016 OB<31-16>. The high half of the output bus.

766020 Flag Register 1. This provides various signals associated with starting and stopping the machine. When the machine stops due to a hardware error, this register tells what happened. The bits are:

- <15> = -WAIT. 1 if the machine is running or runnable, 0 if it is waiting for memory. See the discussion of Clocks for the exact meaning of WAIT.
- <14> = -V1PE. Normally 1, 0 if the level-2 map had a parity error at the last clock.
- <13> = -V0PE. Normally 1, 0 if the level-1 map had a parity error at the last clock.
- <12> = HIGHOK. 1 if the high runs in the machine are all valid, 0 if some are not. This is essentially a power-supply check, and a check for broken wires.
- <11> = -STATHALT. Normally 1, 0 if the machine has been stopped by the statistics counter.
- <10> = ERR. 1 if an error condition is present. If ERRSTOP is on in the mode register, the machine is stopped.

<9> = SSDONE. 1 if a single-step operation has been completed.
<8> = SRUN. 1 if the machine is trying to run (but it may be stopped by a parity error, by a wait condition, or by the statistics counter).
<7> = -HIGHERR. 1 if there was HIGHOK at the last clock.
<6> = -MEMPE. Normally 1, 0 if there was a main memory parity error that was not caught by a trap at the last clock.
<5> = -IPE. Normally 1, 0 if there was a control memory parity error at the last clock.
<4> = -DPE. Normally 1, 0 if there was a dispatch memory parity error at the last clock.
<3> = -SPE. Normally 1, 0 if there was an SPC stack parity error at the last clock.
<2> = -PDLPE. Normally 1, 0 if there was a PDL-buffer parity error at the last clock.
<1> = -MPE. Normally 1, 0 if there was an M-scratchpad parity error at the last clock.
<0> = -APE. Normally 1, 0 if there was an A-scratchpad parity error at the last clock.

766022 Flag Register 2. This register contains flags associated with pipelining and some miscellaneous control signals which the debugging program likes to see. The bits are:

<15> = unused
<14> = unused
<13> = WMAPD. The previous cycle said to write the map, and this cycle will.
<12> = DESTSPCD. The previous cycle wrote into the SPC stack by using a functional destination (as opposed to a CALL transfer).
<11> = IWRITED. The previous cycle did an I-MEM WRITE type of JUMP instruction, and this cycle will write control memory, do a RETURN transfer, and NOP the following cycle.
<10> = IMODD. The previous cycle used the "OA register" to modify this cycle's instruction, or this cycle's instruction came from the DEBUG-IR (see below). This flag inhibits parity checking of the IR.
<9> = PDLWRITED. The previous cycle caused a write into the PDL-buffer, and this cycle will do it.
<8> = SPUSHD. The previous cycle caused a write into the SPC stack, and this cycle will do it.
<7> = unused
<6> = unused.
<5> = IR<48>. This is the parity bit of the IR.
<4> = NOP. The instruction currently in the IR is not really being executed; this cycle is a NOP cycle.
<3> = -VMAOK. The last attempt to start a main memory cycle was not successful because the map indicated a page fault.

<2> = JCOND. 1 if the jump-condition is satisfied. Meaningless if the instruction in IR is not a JUMP instruction.
<1-0> = PCS1-0. These 2 bits select the next PC (the address of the instruction after next.) The encoded values are:
0 = SPC<13-0> the SPC stack.
1 = IR<25-12> the address specified by a JUMP instruction.
2 = DPC<13-0> the dispatch memory.
3 = IPC<13-0> the PC+1.

766024 M<15-0>. The low half of the M-source selected by the instruction currently in IR.

766026 M<31-16>. The high half of the M-source.

766030 A<15-0>. The low half of the A-source selected by the instruction currently in IR.

766032 A<31-16>. The high half of the A-source.

766034 ST<15-0>. The low half of the statistics counter.

766036 ST<31-16>. The high half of the statistics counter.

Here is a description of the writable registers of the diagnostic interface.

766000 DEBUG-IR<15-0>. The low 16 bits of an instruction supplied by the diagnostic interface.

766002 DEBUG-IR<31-16>. The middle 16 bits.

766004 DEBUG-IR<47-32>. The high 16 bits.

766006 Clock control register. Resetting the machine sets this to zero. The following bits exist:
<4> = LDSTAT. Setting this to 1, then clocking the machine, causes the statistics counter to load from IWR<31-0>, which loaded from the M bus on the previous clock.
<3> = IDEBUG. Setting this to 1 causes the IR to load from the DEBUG-IR instead of the PROM or the control memory, when the machine is clocked. The primary way that the machine can be manipulated through the diagnostic interface is by executing instructions using this mechanism.
<2> = NOP11. Setting this to 1 forces NOP. This allows you to clock the machine, for instance to transfer DEBUG-IR into IR, without the present contents of the IR causing unwanted side-effects by getting

executed as an instruction. NOP11 does not prevent the PC from getting changed (in fact it will be incremented), and it does not prevent previously-scheduled pipelined writes from happening.

<1> = STEP. Setting this to 1, when SSDONE is 0, causes the processor clock to run for one cycle, and then set SSDONE. Setting STEP to 0 clears SSDONE. (Both of these operations really take several cycles of the clock to complete.) STEP is the way that the diagnostic interface "clocks" the machine. Note that the main clock is running all the time, even when the machine is stopped. STEP generates a single processor clock, in synchronism with the main clock.

<0> = RUN. Setting this to 1 causes the machine to start running. You first use STEP to set up the state of all the registers and memories, the PC, and the IR, then turn on RUN. The first instruction executed is the one you left in the IR.

766010 OPC control register. Resetting the machine sets this to zero. This register contains some bits which need to be used by the console program in order to completely restore the state of the machine from a saved state. The bits are:

<2> = OPCINH. Setting this to 1 inhibits the OPCs from being clocked by the processor clock. This bit must not be changed except when the clock is high (i.e. the machine is stopped). The process of restoring the OPCs consists of setting OPCINH, then getting the 8 values into the PC by executing JUMP instructions, and transferring those values into the OPCs via the OPCCLK bit. Once the OPCs have been restored, OPCINH remains set so that they will be undisturbed while the rest of the machine state is restored. Just before starting the machine, set OPCINH to 0.

<1> = OPCCLK. Setting this to 1 and then to 0 generates a clock to just the OPCs. This is used to read out the 8 OPC registers without disturbing the state of the rest of the machine.

<0> = LPC.HOLD. Setting this to 1 prevents the LPC register from loading from the PC register when the machine is clocked. This is used in restoring the LPC. The LPC is a duplicate copy of the first OPC register, used by the IR<25> feature of the DISPATCH instruction.

766012 Mode register. Resetting the machine sets this to zero. This register enables various features and controls the speed of the clock. The bits are:

<7> = PROG.BOOT. Setting this to 1 starts a bootstrap sequence.

<6> = PROG.RESET. Setting this to 1 resets the machine. Reset stops the machine by clearing RUN, forces the clock to stop until the RESET operation is over, clears the pipeline flags which cause things to happen in the next instruction, and clears the Clock, Mode, and OPC registers of the diagnostic interface.

<5> = PROMDISABLE. A 1 here disables the PROM. A 0 here replaces the first 1K locations of control memory with the PROM.

<4> = TRAPENB. A 1 here enables main memory parity errors to cause

microcode traps to location 0. A 0 here causes main memory parity errors to be treated the same as other parity errors.

<3> = STATHENB. A 1 here enables overflow of the statistics counter to halt the machine.

<2> = ERRSTOP. A 1 here enables hardware errors (HIGHERR and various parity errors) to halt the machine. A 0 causes it to continue blithely on.

<1-0> = SPEED<1-0>. These bits control the speed of the clock. The ILONG bit in the microinstruction also affects the speed, slowing it down by 40 nanoseconds. The speed codes are:

0 = Extra Slow
1 = Slow
2 = Normal
3 = Fast

766014 not used.

766016 not used.

The OPCs are a set of 8 registers which remember the last 8 values of the PC. This provides a useful history for debugging. It is also used by the microcode itself in certain trap-handling routines. You can only read the last of the 8 OPCs, which is what the PC was 8 clocks ago. Special control is provided over the clocking of the OPCs so that they can be read out without di so that they can be saved and restored by the microcode debugger. This is described above under 766010.

The OPCs can be read both by the diagnostic interface and as a functional source, for maximum flexibility.

The bus interface provides a special path by which the MD register may be loaded. This provides a parallel source of diagnostic input data. After loading MD, instructions can be executed via the DEBUG-IR to transfer the data to the desired destination.

There are several maintenance indicators (light-emitting diodes) scattered around the machine. Inside the front door, near the lower-left-hand corner, are 5 octal displays. These show the current value of the PC. The decimal points on these displays show various interesting conditions. From left to right:

1 - PROMENABLE. Indicates that the current instruction is coming from the PROM rather than the writable control memory.

2 - IPE. Indicates that control memory had a parity error at the last clock.

3 - DPE. Indicates that dispatch memory had a parity error at the last clock.

4 - TILT0. Indicates that the map or main memory had a parity error at the last clock.

5 - TILT1. Indicates that the A-scratchpad, the M-scratchpad, the PDL-buffer, or the SPC stack had a parity error at the last clock.

There is also provision for indicators for the various error conditions, "the machine is really running," and the status of the disk interface. The location of this indicator panel, and whether or not all machines will have one, is not yet determined.

The Disk Controller

The Lisp machine disk controller attaches from 1 to 8 disk units of the "Trident" family to the CADR machine's XBUS. The 1-unit version consists of one board, and a second board is added when more than one disk unit is to be used. The two versions are almost program compatible.

Interface Registers

The disk controller is operated by reading and writing four 32-bit registers which are on the XBUS. These are normally at physical addresses 17377774-17377777, which is just below the Unibus. The address can be changed by changing jumpers. Many bits in these registers refer to the "selected unit", which is that disk unit whose number is currently in bits <30:28> of the disk-address register.

When read, the registers are:

0 STATUS

<24:31> The block-counter of the selected unit. This tells you its current rotational position. Reading of this register is not synchronized to its incrementation, so you must read it twice and check that it came out the same both times.

<23> Internal Parity Error. This indicates that parity of the bits seen at the disk and parity of the bits seen at the memory failed to agree; something must have been lost inside the controller someplace. The Read All and Write All commands cause spurious internal parity errors. The Read Compare command causes a spurious internal parity error if it sets Read Compare Difference (bit 22) and the the disk data and the memory data differ in parity. This error does not stop the transfer.

<22> Read Compare Difference. This indicates that data from memory and data from the disk failed to agree. This bit is undefined unless the command is read-compare. This error does not stop the transfer.

<21> CCW Cycle. This bit being on in combination with Memory Parity Error or Nonexistent Memory Error indicates that the error happened while fetching a CCW, rather than while reading or writing data.

<20> Nonexistent Memory Error. Indicates that memory (or other XBUS device) failed to respond within 15 microseconds. This error stops the transfer.

<19> Memory Parity Error. Indicates that even parity was read from memory (or other XBUS device). This error stops the transfer.

<18> Header Compare Error. Indicates that a block-header read from disk failed to have the expected value. This may be because the disk head is not positioned at the proper place, because the disk is not correctly formatted, or because the header wasn't read correctly. This error stops the transfer.

<17> Header ECC Error. Indicates that the error-correcting code of a block header failed to check. Unfortunately most header ECC errors show up as header compare errors instead. Maybe this can be fixed? This error stops the transfer. Header ECC Error also happens if an attempt is made to continue a

read or write operation past the end of the disk.

<16> ECC Hard. Indicates that the error correcting code discovered an error, and was unable to correct it. The data read from disk is wrong, try reading again. This error stops the transfer.

<15> ECC Soft. Indicates that the error correcting code discovered an error, and was able to determine which data bits were in error. The program can correct it, see the ECC Register for how. The error correcting code will correct any single burst of up to 11 erroneous bits. This error stops the transfer.

<14> Read Overrun. Indicates that data arrived from the disk faster than it could be stored into memory. This error stops the transfer.

<13> Write Overrun. Indicates that memory did not supply data fast enough for the disk. This error stops the transfer.

<12> Start Block Error. Indicates that a start-of-block (sector pulse) happened at a time when it should not have. Either the disk is incorrectly formatted or it is generating spurious sector pulses. This error stops the transfer.

<11> Timeout Error. Indicates that a disk operation took longer than 2.5 seconds. This error stops the transfer.

<10> Selected Unit Seek Error. The selected unit is reporting failure of a seek operation. This error stops the transfer. Reset the error by using the Recalibrate command.

<9> Selected Unit not On-line. The heads are not loaded, the disk is not powered on, or there is no disk at the specified unit number. This error stops the transfer.

<8> Selected Unit not On-Cylinder. Generally indicates that a seek is in progress on the selected unit. Not an error. If the disk goes off-cylinder during a write operation, a fault will occur. If it goes off cylinder during a read, presumably a header-compare error or an ECC error will occur.

<7> Selected Unit Read-Only. The status of a switch on the disk. Note that the read-only status can only change to reflect a change in the switch when the drive is not selected. Storing into the Disk Address register momentarily deselects the current unit so that it may update its read-only status from the switch. Writing while the disk is read-only causes a fault.

<6> Selected Unit Fault. Indicates either trouble with the disk or a programming error, see the Trident manual. This error stops the transfer. Reset by using the Fault Clear and/or Recalibrate commands. This error lights the Device Check light on the drive.

<5> No Unit Selected. This error stops the transfer. Happens if no disk is plugged into the selected unit number, or the disk unit is powered off or "degated".

<4> Multiple Units Selected. This error stops the transfer. This indicates that more than one disk drive is selected, or the wrong drive is selected.

<3> Interrupt Request. 1 means the disk controller is asserting -XBUS.INTR.

<2> Selected Unit Attention. Reset using the At Ease command. Attention indicates seek completion, recalibrate completion, initial loading of

the heads, seek incomplete error, or an emergency head retract. "Implicit" seeks do not cause attention.

<1> Any Attention. Some unit has an attention, you have to select them one after another to find out which.

<0> Not Active. 0 means the controller is busy, 1 means it is ready to accept a command.

1 MEMORY ADDRESS

<31:24> not used

<23:22> Disk type. 00 Trident 01 Marksman 10 unused 11 Trident (old control)

<21:0> the address of the last memory reference made by the disk control. This is the address of a CCW if CCW Cycle is on in the status register, otherwise the address of a data word.

2 DISK ADDRESS

<31> not used

<30:28> Unit number. In the 1-unit version, always zero.

<27:16> Cylinder number. A T-80 has 815. cylinders.

<15:8> Head number. A T-80 has 5 heads. As it turns out, only the bottom 6 bits of the head number can work (this is a feature of the Trident.)

<7:0> Block number. A T-80 is usually formatted with 17. blocks per track. "Block" is mostly synonymous with "sector".

When a transfer is terminated by an error, the disk address register contains the address of the block being transferred when the error occurred. When a transfer terminates normally, the disk address register has the address of the last block transferred.

3 ERROR CORRECTION REGISTER

<31:16> Error pattern bits.

<15:0> Error bit position+1.

When a soft ECC error occurs, this register tells where in the last block transferred the error was. The disk address register has the disk address of the block containing the error, and the command list pointer points to the CCW which points to the memory page containing the error. The error pattern should be XOR'ed into the contents of memory at the specified bit address; it may overlap across a word boundary. Note that the bit position is off by 1; the first bit in the block is bit 1.

You should not write any register while a transfer is active, except for using the Reset command to stop a hung transfer, and even then you should expect to lose.

When written, the registers are:

0 COMMAND

Writing the command register does NOT initiate a transfer, unlike most disk

controllers. Use register 3 (START) to initiate a transfer, after setting up the other registers. However, writing the command register does reset the various error flags. Note that the command register cannot be read back.

<31:12> not used

<11> Done Interrupt Enable. Enables not-active (bit 0 of the status register) to cause an interrupt. The interrupt will keep happening until you clear this bit. (This is really an idle interrupt rather than a done interrupt.)

<10> Attention Interrupt Enable. Enables any-attention (bit 1 of the status register) to cause an interrupt. (The interrupt will only happen if the controller is not active. While the controller is active you couldn't do anything about it anyway.) The interrupt will keep happening until you select the drive and give an at-ease command, or clear this bit.

<9> Recalibrate. In combination with command 5, causes the disk to return the heads to cylinder 0.

<8> Fault Clear. In combination with command 5, resets most fault conditions in the disk.

<7> Data Strobe Late. For recovery of marginal data.

<6> Data Strobe Early. For recovery of marginal data.

<5> Servo Offset. For recovery of marginal data, offsets the heads slightly. Bit 4 controls which direction. Note that this is somewhat kludgey, if you try to seek while the heads are offset you get a fault (use command 6 first to clear the offset.) Transferring more than one block at a time while in servo offset mode, or even retrying a transfer without first doing an offset clear, will probably cause a fault. Of questionable worth anyway. Writing while the heads are offset causes a fault.

<4> Offset forward. 1 means offset forward, 0 means offset backward.

<3> I/O Direction. 1 means from-memory, 0 means to-memory. See below for valid combinations.

<2:0> Command code. The following combinations of bits are valid commands (here expressed in octal). Note that bits 10 and 11 may always be turned on, and bits 4 through 7 may be turned on in any reading command.

0000 Read.

0010 Read-compare. Reads from both disk and memory, and sets bit 22 of the status register if they don't agree.

0011 Write.

0002 Read All. Reads all bits of the disk starting at the specified rotational position. Note that internal parity errors will occur spuriously during this command, and that it will not automatically advance heads and cylinders. See the description of disk formatting below.

0013 Write All. Writes all bits of the disk starting at the specified rotational position. This is intended for formatting the disk, see below. The caveats under READ ALL apply to WRITE ALL also. In addition, it doesn't really write quite all of the last page; somewhere between zero and seventeen words will be lost.

0004 Seek. Initiates a seek to the cylinder specified in the disk address register. An attention will occur when the seek completes. Note that this command is not logically necessary; the controller always initiates a seek if necessary at the start of a data transfer command. The read, read-compare, and write commands also will seek in the middle of a transfer when necessary. The seek command is provided so you can overlap seeks on multiple units.

0005 At ease. Resets attention on the selected unit.

1005 Recalibrate. Seek to cylinder 0, without assuming the current position of the heads is correct. This is used to correct a seek error, and as part of error recovery. Recalibrate resets some error conditions in the drive, and causes an attention when complete.

0405 Fault clear. Resets most error conditions in the drive.

1405 This probably does both a Recalibrate and a Fault Clear.

0006 Offset clear. Take the heads out of the offset state. This does not wait for completion, but the next command will.

xxx7 This is a reserved command, and will currently hang the controller, causing a timeout error (bit 11 in the status register.)

0016 Reset. This stops the current transfer and resets the controller. This command takes effect as soon as it is stored in the command register; no store in START is required. After storing a Reset command you should store 0 in the command register to turn off the reset condition. Use of Reset while a transfer is in progress isn't guaranteed not to do strange things.

All commands except for the xxx5 group and Reset wait for completion of any previous seek operation on the selected unit before starting. Thus even the Seek and Offset Clear commands can take finite time before the controller is ready for the next command.

1 COMMAND LIST POINTER

This is the address of a vector of Channel Command Words (CCWs) which specify what memory pages, and how many, are to be transferred to/from disk. Only bits <15:0> of the CLP can count, so if you try to carry across this boundary your command list will wrap around.

The format of a CCW is:

<31:24> not used

<23:8> Main memory address of a page

<7:1> not used

<0> More flag. If this bit is 0, this is the last CCW in the list. If this bit is 1, there is another CCW in the following location.

2 DISK ADDRESS

See the description of the disk address register under reading. Note that in the 1-unit version, the unit number bits <30:28> are ignored and regarded as always zero.

3 START

Writing anything at this address initiates the operation specified in the command, disk address, and command list pointer registers.

Disk Structure

Each disk block contains one Lisp machine page worth of data, i.e. 256. words or 1024. bytes. You can transfer up to 65536. consecutive disk blocks to non-consecutive memory locations in a single operation, or you could if the machine supported that much main memory. A T-80 has 815. cylinders, each with 5 heads (tracks), each with 16. or 17. blocks depending on how you feel like formatting it. A T-300 is the same except it has 19. heads.

Formatting

The format is determined by the program that uses the Write All operation to format the disk, within the constraints determined by the hardware. A track contains (approximately) 20160. bytes (on a T-80 or a T-300). Jumpers in the disk are set to give 17. sector pulses per track, or one every 1164. bytes, with a little left over at the end of the track.

Everything goes low-order bit first and low-order byte first. Note that bits in the disk controller are the complement of bits seen by the drive. Thus all bits in the Trident manual should be thought of as complemented.

The format of a block is:

```
(sector pulse here)
PREAMBLE - 53. bytes of ones.
VFO LOCK - 8. bytes of ones.
SYNC - a byte containing octal 177
HEADER - a 32-bit word as follows:
   <31:30> next block address code:
        0 following block on same track
        1 block 0 on next track (next head)
        2 block 0 on head 0 of next cylinder
        3 end of disk
   <29:28> not used, should be zero
   <27:16> cylinder number, used to verify that the
           disk is positioned to the correct cylinder.
   <15:8> head number, used to verify the head selection.
   <7:0> block number, used to verify the rotational position.
HEADER ECC - a 32-bit checkword.
VFO RELOCK - 20. bytes of ones.
SYNC - a byte containing octal 177
```

```
PAD - a byte containing octal 377, which is here to fix
        a bug in the logic for read-compare.  (Ugh)
DATA - 1024. bytes of whatever you want.
DATA ECC - a 32-bit checkword.
POSTAMBLE - 44. bytes of ones.
```

To format the disk, you should do it one track at a time. Lay out in memory the bits to be written on the track. Truncate the length to a multiple of a page, but make sure that the last 17. words don't matter (in general you will be writing 19. pages, or 19456. bytes, leaving about 771. bytes at the end of the track which may not get written, depending on how full the fifo is when the operation terminates. Depending on the block length chosen, you may not get a chance to fully write the last block, but as long as you get into the data area it will be all right. Do a WRITE ALL command of this data, with a disk address whose block-number field (bits <7:0>) is zero. Ignore any internal parity error (bit 23 of the status register.) You can verify it by using the Read All command (but the internal parity and read-compare features will not work), or you can use the ordinary write and read commands. You must compute the ECC check-words manually. The polynomial is x^31+x^29+x^20+x^10+x^8+1 [if I understand this logic correctly.]

Note that, when using Read All, there is some ambiguity as to precisely where the data read starts. It is unlikely to line up the bytes on byte boundaries. The first several microseconds worth of data will be missing or corrupted.

Debugging

Connector J11 is provided for a flat cable to an LED display, with the following useful signals on it. These are ground when inactive, 15 milliamps at +3 volts or so when active.

1 Read Active. The controller is active and bit 0 of the command register is 0.

2 Write Active. The controller is active and bit 0 of the command register is 1.

3 Seek. The selected unit is not on-cylinder.

4 Transfer Lossage. This is the IOR of Timeout, Read Overrun, Write Overrun, Memory Parity Error, and Nonexistent Memory Error.

5 Format Lossage. This is the IOR of Start Block Error, Header Compare Error, Header ECC Error, and Reset.

6 ECC Lossage. This is the IOR of Hard ECC Error and Soft ECC Error.

7 Disk Lossage. This is the IOR of Multiple Units Selected, No Units Selected, Selected Unit Fault, Selected Unit not On-Line, and Selected Unit Seek Error.

8 Spare. This probably does not light up.

<<Here insert a one-page table of instruction formats and so forth>>

The CONSLP Assembler

CONSLP is a symbolic assembler written in Maclisp which reads in source code for the CADR machine and produces a file loadable by the CC debugger. The source code is written in the form of LISP S-expressions; symbols are LISP atomic symbols, and instructions or data items are written as lists. Comments can thus be written using Maclisp's semicolon convention. The input radix for numbers is 8 (octal), except that a trailing decimal point forces radix 10 (decimal).

Localities

A program can specify data to be loaded into the instruction, dispatch, A, and M memories. To specify which of the memories to assemble data for, the LOCALITY pseudo-op is used:

```
(LOCALITY I-MEM)        ;following data goes into instruction memory
(LOCALITY D-MEM)        ;ditto, dispatch memory
(LOCALITY A-MEM)        ;ditto, A memory
(LOCALITY M-MEM)        ;ditto, M memory
```

Location Tags and Symbols

When an atomic symbol is encountered in the instruction stream being assembled, it is taken to be a location tag (label). The tag is defined, as usual, to be the value of the next location in the current locality to be assembled into, but shifted to put the tag value into its "normal" position. For A memory tags, the normal position is the A-source field of an instruction; similarly for M memory tags. For I memory tags, the normal position is the New PC field of a JUMP instruction; for D memory tags, the Dispatch Offset field of a DISPATCH instruction. Thus, if FOO is a tag for location 7 of dispatch memory, then the effective value of FOO is 70000 .

By convention, tags in A memory begin with the letters "A-", and in M memory with "M-", but this is not enforced by CONSLP.

Symbols can also be defined by means of the ASSIGN pseudo-operation:

```
(ASSIGN <symbol> <value>)
```

For example:

```
(ASSIGN CDR-IS-NORMAL 0)
(ASSIGN CDR-IS-ILLEGAL 1)
(ASSIGN CDR-IS-NIL 2)
(ASSIGN CDR-IS-NEXT 3)
```

The <value> may be an expression program, in general. When a symbol is referenced,

the expression program is evaluated to produce the symbol's value (which may be conditional on the context in which it appears). Expression programs are discussed in a later section.

Instructions

In general, CONSLP assembles a list into a data item by evaluating all the elements of the list and adding them up. There is a fairly rich language for specifying complex expression programs and assigning symbolic names to them; for now, however, we will merely use the symbols predefined by CONSLP. CONSLP also allows the fields of an instruction to be written in almost any order, but we will describe only the conventional order for writing them.

The general form of an I-MEM instruction is:

```
(<popj> (<destinations>) <operation> <condition>
   <M-source> <byte-descriptor> <A-source> <target-tag> <other fields>)
```

The <popj> field is POPJ-AFTER-NEXT to specify that the POPJ bit be set.
The <destinations> field may be an A or M memory tag, or the name of a functional destination, or both an M memory tag and a functional destination.
The <operation> specifies the instruction type, and possibly other fields (such as the jump condition) as well.
The <condition> may also be a separate field, though it usually is encoded as part of the operation.
The <byte-descriptor> describes the byte to be used in a BYTE or DISPATCH instruction.
The <M-source> and <A-source> specify the sources; these may be tags in the appropriate memories, or, for the <M-source>, the name of an M multiplexor source.
The <target-tag> is an I-MEM tag for JUMP instructions, or a D-MEM tag for DISPATCH instructions.
The <other fields> can be such things as the Q control and Miscellaneous Functions.

Many of these fields can be omitted, and CONSLP will default them appropriately. If the <operation> is omitted, then ALU is assumed, unless a <byte descriptor> is present either implicitly or explicitly, in which case BYTE is assumed. If only one source is present in an ALU instruction, then an opcode of SETA is supplied for an A source, and SETM for an M source, thus causing a simple movement of data. If the A source is omitted in a BYTE instruction, then location 2 in A memory is assumed (which is supposed to contain zero).

Here are some examples of instructions, with commentary. We assume the convention described above for A and M memory tags.

```
((A-FOO) M-BAR)      ;move from BAR in M-MEM to FOO in A-MEM

(CALL ZAP)           ;do a CALL transfer to instruction ZAP (N bit set)
```

```
((A-FOO) SUB M-BAR A-BAZ)
                    ;subtract A-BAZ from M-BAR, put result in A-FOO

(JUMP-EQUAL-XCT-NEXT M-BAR A-FOO LOSE)
                    ;jump to LOSE if M-BAR equals A-FOO; N bit is clear,
                    ; so instruction after the JUMP is executed
                    ; whether or not the JUMP succeeds

(POPJ-AFTER-NEXT (M-FOO) MEMORY-DATA)
                    ;put data from memory into M-FOO,
                    ; and also POPJ after next instruction

((M-SAVE MEMORY-DATA-START-WRITE)
          ADD MEMORY-DATA A-ZERO ALU-CARRY-IN-ONE)
                    ;add one to the read memory data,
                    ; transfer to write memory data and M-SAVE,
                    ; and begin writing the data into main memory
                    ; at the address already in the VMA
```

Literals

CONSLP provides a facility for specifying literals in the A and M memories. The constructs

```
(A-CONSTANT <expression>)   and   (M-CONSTANT <expression>)
```

may appear as an A source or M source specification, causing CONSLP to allocate a word in the appropriate memory, assemble the literal expression there, and use the address of that location as the source location. If the same constant in the same memory is referenced many times, CONSLP will assemble only one copy of it. Two constants are considered the same if their final binary values are identical, regardless of the source expressions which reduced to those values. The zero constant is treated specially, and made to refer to location 2 of the appropriate memory (hence the user should reserve these locations as constant sources of zeros). Similarly the -1 constant is made to refer to location 3 of the appropriate memory.

Byte Specifications

Rather than requiring the user to calculate the rotation count and length (minus 1) fields for BYTE and DISPATCH instructions, CONSLP provides a uniform method for specifying a byte in terms of its size and position in the word; CONSLP then calculates the fields appropriately.

The simplest way to describe a byte is with the BYTE-FIELD construct:

(BYTE-FIELD <size in bits> <position from right>)

For example, (BYTE-FIELD 5 0) is the low five bits of a word, and (BYTE-FIELD 7 5) is the seven bits above them. The two arguments to BYTE-FIELD must be constant integers.

Another way to describe a byte is:

(LISP-BYTE <ppss>)

where the low two octal digits of <ppss> are the size and the next two are the position. The argument <ppss> is evaluated as a LISP form (see below under "Expression Programs").

When a byte specifier appears in an instruction, the op-code is defaulted to BYTE, and the type of byte instruction defaulted to "load byte". If specified elsewhere in the instruction, the op-code may be DISPATCH instead; the dispatch is based on the specified byte. The op-code may also be JUMP, but only if the byte is one bit wide; this means that the jump will test the specified bit of the M source.

When CONSLP assembles the final instruction, it constructs the rotation count and length minus 1 fields on the basis of the byte specifier and the operation to be performed. For JUMP, DISPATCH, and "load byte" type BYTE instructions, this involves subtracting the byte position from 32 to obtain the correct rotation count. (Recall that CADR rotates words to the left.) If Miscellaneous Function 3 (LOW PC BIT specifies half word) is enabled, then the position (which should be less than 16) is subtracted from 16 instead. For "deposit byte" and "selective deposit" type BYTE instructions, the byte position itself is used as the rotation count. The length minus 1 field for BYTE and JUMP is computed by subtracting 1 from the byte length, unless the byte length is zero, in which case zero is used. (Note that CADR cannot really handle zero-length bytes, but CONSLP allows them to be defined on the theory that the "next instruction modify" feature may be in use. Programs which use this feature must be aware of the hackery which the assembler pulls, and allow for the actual values of the fields at run time.) The DISPATCH instruction has a length field instead of a length minus 1 field, and so no subtraction of 1 is performed for it.

Here are some examples of the use of byte specifiers:

```
((M-X) (BYTE-FIELD 7 4) M-Y)
                    ;extracts a 7-bit byte, 4 bits from
                    ; the right, from M-Y, and puts this
                    ; byte right-justified in M-X.  The
                    ; A source is defaulted to 1, which
                    ; should be a constant zero so that the
                    ; other bits in M-X will be zero.

(JUMP-IF-BIT-SET (BYTE-FIELD 1 3) M-ZAP QUUX)
                    ;jump to QUUX if the "10" bit is set in M-ZAP

(DISPATCH (BYTE-FIELD 3 0) M-ZAP DTABLE)
```

```
;use the low three bits of M-ZAP to index
; into the dispatch table DTABLE
```

It is possible to create a symbolic name for a byte field by using the ASSIGN pseudo-operation:

```
(ASSIGN LOW-HEX-DIGIT (BYTE-FIELD 4 0))
```

Since this is a common operation, another pseudo-op exists for the purpose:

```
(DEF-DATA-FIELD <symbol> <byte size> <byte position>)
```

For example:

```
(DEF-DATA-FIELD LOW-HEX-DIGIT 4 0)
```

It is also possible to associate a name with a byte field in a particular register. One way to do this is to sum the byte specifier and the name of the register:

```
(ASSIGN CONDITION-CODES (PLUS (BYTE-FIELD 4 0) PDP-11-PS))
(ASSIGN TRACE-TRAP-BIT (PLUS (BYTE-FIELD 1 4) PDP-11-PS))
(ASSIGN PRIORITY (PLUS (BYTE-FIELD 3 5) PDP-11-PS))
```

This case too is common enough to warrant a special pseudo-operation for the purpose:

```
(DEF-BIT-FIELD-IN-REG <symbol> <byte size> <byte position> <register>)
```

For example:

```
(DEF-BIT-FIELD-IN-REG CONDITION-CODES 4 0 PDP-11-PS)
(DEF-BIT-FIELD-IN-REG TRACE-TRAP-BIT 1 4 PDP-11-PS)
(DEF-BIT-FIELD-IN-REG PRIORITY 3 5 PDP-11-PS)
```

Note that the <register> had better be in the M-scratchpad. With this definition, it is only necessary to mention, say, PRIORITY, in an instruction to cause an appropriate byte reference to occur:

```
((A-PRIORITY) PRIORITY)      ;extract the PRIORITY byte from PDP-11-PS
                             ; and place it right-justified in A-PRIORITY
```

By special dispensation, it also works to use such symbols in the destination field. The appropriate DPB is assembled.

Two more pseudo-operations make it easy to define names for many consecutive bits or fields in a register.

(DEF-NEXT-FIELD <symbol> <byte size> <register>) .sp This defines <symbol> to be a byte of the speicified size, in a position to the left of any fields already defined by DEF-NEXT-FIELD. If this is the first DEF-NEXT-FIELD for the specified register, then the field position is zero (at the low end of the word). For example:

```
(DEF-NEXT-FIELD REL-OFFSET 8 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD TAG-FIELD 2 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD FORMAT-BIT 1 IBM-1130-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 5 IBM-1130-INSTRUCTION)
```

would be entirely equivalent to:

```
(DEF-BIT-FIELD-IN-REG REL-OFFSET 8 0 IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG TAG-FIELD 2 8 IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG FORMAT-BIT 1 10. IBM-1130-INSTRUCTION)
(DEF-BIT-FIELD-IN-REG OP-CODE 5 11. IBM-1130-INSTRUCTION)
```

The pseudo-operation:

```
(DEF-NEXT-BIT <symbol> <register>)
```

is entirely equivalent to:

```
(DEF-NEXT-FIELD <symbol> 1 <register>)
```

and so allocates a single bit. It may be intermixed freely with DEF-NEXT-FIELD. For example:

```
(DEF-NEXT-FIELD CONDITION-CODES 4 PDP-11-PS)
(DEF-NEXT-BIT TRACE-TRAP-BIT PDP-11-PS)
(DEF-NEXT-FIELD PRIORITY 3 PDP-11-PS)
```

The construct:

```
(RESET-BIT-POINTER <register>)
```

may be used to reset the pointer into <register> used by DEF-NEXT-FIELD and DEF-NEXT-BIT. This is useful if the data in <register> can have several different formats. For example:

```
(DEF-NEXT-BIT C PDP-11-PS)
(DEF-NEXT-BIT V PDP-11-PS)
(DEF-NEXT-BIT Z PDP-11-PS)
(DEF-NEXT-BIT N PDP-11-PS)
(RESET-BIT-POINTER PDP-11-PS)
(DEF-NEXT-FIELD CONDITION-CODES 4 PDP-11-PS)
```

```
(DEF-NEXT-BIT TRACE-TRAP-BIT PDP-11-PS)
(DEF-NEXT-FIELD PRIORITY 3 PDP-11-PS)

(DEF-NEXT-FIELD DST-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD DST-MODE 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 4 PDP-11-INSTRUCTION)
(RESET-BIT-POINTER PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD BRANCH-OFFSET 8 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD BRANCH-CONDITION 3 PDP-11-INSTRUCTION)
(RESET-BIT-POINTER PDP-11-INSTRUCTION)
```

Dispatch Tables

When assembling into the dispatch memory (i.e. (LOCALITY D-MEM)) it is necessary to use two special pseudo-operations, START-DISPATCH and END-DISPATCH, to allocate blocks of dispatch memory. These pseudo-operations specify the length of the block required, and CONSLP undertakes to pack the various odd-sized blocks into the dispatch memory in an appropriate manner.

The typical form for a dispatch block is:

```
(START-DISPATCH <log2 of size> <constant data>)
<dispatch table tag>
        <first word of table>
        ...
        <last word of table>
(END-DISPATCH)
```

The <log2 of size> is the number of bits that will be dispatched on, that is, the logarithm base 2 of the size of the dispatch block. The <constant data> will be added into each of the words of the dispatch table; this is useful for the P, R, and N bits (which in CONSLP are called P-BIT, R-BIT, and INHIBIT-XCT-NEXT-BIT). The END-DISPATCH is logically not necessary, but is used for error checking. Exactly the correct number of words must be assembled between the START-DISPATCH and END-DISPATCH, or CONSLP will give an error message.

As an example of a dispatch table, consider this code:

```
(LOCALITY M-MEM)
PDP-11-INSTRUCTION          (0)      ;HOLDS SIMULATED PDP-11 INSTRUCTION
(DEF-NEXT-FIELD DST-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD DST-MODE 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD SRC-REG 3 PDP-11-INSTRUCTION)
(DEF-NEXT-FIELD OP-CODE 4 PDP-11-INSTRUCTION)
```

```
        ...

(LOCALITY I-MEM)
        (DISPATCH-CALL-XCT-NEXT DST-MODE D-DST-MODE)
        ...

(LOCALITY D-MEM)
(START-DISPATCH 3 P-BIT)
D-DST-MODE
        (DST-REGISTER)                  ;R0
        (DST-REG-INDIRECT)              ;@R0
        (DST-AUTO-INCREMENT)            ;(R0)+
        (DST-AUTO-INC-INDIRECT)         ;@(R0)+
        (DST-AUTO-DECREMENT)            ;-(R0)
        (DST-AUTO-DEC-INDIRECT)         ;@-(R0)
        (DST-INDEXED)                   ;N(R0)
        (DST-INDEXED-INDIRECT)          ;@N(R0)
(END-DISPATCH)
```

Note that the use in I-MEM of the op-code DISPATCH-CALL-XCT-NEXT is purely for cosmetic purposes, to indicate that the P bit but not the N bit is a constant in all of the dispatch table entries; it is otherwise identical to the DISPATCH op-code.

Standard Operation Codes

CONSLP supplies a large number of initial symbols for various operations, particularly for the various conditional jumps. While it is possible to define different ones, use of these standard ones is naturally encouraged. (These symbols are defined in the file LISPM; CONSYM >.)

ALU Operations

The standard ALU operations supplied by CONSLP are:

Boolean

SETCM	set to complement of M
ANDCB	AND together complements of both M and A
ANDCM	AND complement of M with A
SETZ	set to zeros
ORCB	OR together complements of both M and A
SETCA	set to complement of A
XOR	XOR (exclusive OR) M and A
ANDCA	AND M with complement of A
ORCM	OR complement of M with A

EQV	EQV M and A (complement of XOR)
SETA	set to A
AND	AND together M and A
SETO	set to ones
ORCA	OR M with complement of A
IOR	OR M and A (inclusive OR)
SETM	set to M

Arithmetic

ADD	M plus A (two's complement addition)
SUB	M minus A (two's complement subtraction)
M+M	M plus M (two's complement addition)
M+M+1	M plus M plus 1
M+A+1	M plus A plus 1
M-A-1	M minus A minus 1
M+1	M plus 1

Conditional Arithmetic

MULTIPLY-STEP
DIVIDE-FIRST-STEP
DIVIDE-STEP
DIVIDE-LAST-STEP
DIVIDE-REMAINDER-CORRECTION-STEP

The conditional ALU operations for multiplication and division are explained in detail in a later section.

The output bus selector field defaults to 1 (output bus gets ALU output). The other two choices must be specified explicitly:

```
OUTPUT-SELECTOR-RIGHTSHIFT-1
OUTPUT-SELECTOR-LEFTSHIFT-1
```

The Q control field of an ALU instruction may be specified by using one of these symbols:

SHIFT-Q-LEFT	shift Q left (shifts inverse of ALU<31> into Q<0>)
SHIFT-Q-RIGHT	shift Q right (shifts ALU<0> into Q<31>)
LOAD-Q	load Q from output bus

If none of these is present, the default is to do nothing to Q. (Instead of writing LOAD-Q, one may write Q-R in the destination portion of the instruction. This does not mean that Q is a functional destination; it merely forces the operation to be ALU, and forces the Q control field to be LOAD-Q.)

The carry field may be specified by ALU-CARRY-IN-ZERO or ALU-CARRY-IN-ONE. Note that the SUB, M+M+1, M+A+1, and M+1 operations have ALU-CARRY-IN-ONE as part of their definitions, so it is not necessary to specify it explicitly.

BYTE operations

If a byte specifier is present in an instruction and the op-code is not explicitly forced to be JUMP or DISPATCH, then the op-code is BYTE by default, performing a "load byte" type of operation.

To get a "deposit byte" type operation, the symbol DPB is used; similarly, to get a "selective deposit", SELECTIVE-DEPOSIT is used. For example:

```
((A-FOO) DPB M-BAR (BYTE-FIELD 3 6) A-FOO)
                    ;a true PDP-10 style DPB: the low octal
                    ; digit of M-BAR replaces the third lowest
                    ; octal digit of A-FOO.

((A-ZAP) DPB M-BAR (BYTE-FIELD 3 6) A-FOO)
                    ;similar, but the result is placed in
                    ; A-ZAP.  A-FOO is not altered.

((A-ZAP) SELECTIVE-DEPOSIT M-FOO (BYTE-FIELD 16. 8) (A-CONSTANT -1))
                    ;A-ZAP gets a copy of M-FOO with the high eight
                    ; bits and the low eight bits replaced with all ones
                    ; (alternatively, it gets a copy of the -1
                    ; with the middle 16. bits replaced with
                    ; the corresponding bits from M-FOO)
```

DISPATCH Operations

Four op-codes are defined in CONSLP for dispatching:

```
DISPATCH
DISPATCH-CALL
DISPATCH-XCT-NEXT
DISPATCH-CALL-XCT-NEXT
```

These are provided purely for cosmetic purposes, since the actual dispatch action is controlled by the dispatch table. CONSLP makes no attempt to check that the "correct" op-code is used with a given dispatch table. By convention, the XCT-NEXT versions are used iff the instruction following the dispatch instruction will be executed (N bit <u>not</u> set), and the CALL versions are used if the P bit is set.

To specify the value of the 10-bit "immediate argument" which is loaded into the DISPATCH CONSTANT register, one may use

```
(I-ARG <expression>)                ;immediate argument
```

in the dispatch instruction.

There is a special pseudo-op to facilitate use of the DISPATCH CONSTANT to pass a small, constant number as an argument to a subroutine. The form

```
((ARG-CALL FOO) (I-ARG BAR))
```

generates a DISPATCH instruction to a one-word table containing a CALL-type transfer to FOO, and puts BAR in the dispatch constant field of the dispatch instruction. FOO may then use the READ-I-ARG functional source to pick up and act on the argument.

Miscellaneous Function 2 (write into the dispatch memory) is specified by the symbol WRITE-DISPATCH-RAM.

JUMP Operations

CONSLP defines a large number of names for the various JUMP operations. These are all built out of a logical progression of pieces:

```
<type>  <condition>  <xct next>
```

The <type> may be either JUMP, CALL, or POPJ, meaning that no bits, the P bit, or the R bit is set. The <condition> may be one of the following:

```
IF-BIT-SET
IF-BIT-CLEAR
EQUAL
NOT-EQUAL
LESS-THAN
GREATER-THAN
GREATER-OR-EQUAL
LESS-OR-EQUAL
IF-PAGE-FAULT
IF-NO-PAGE-FAULT
IF-PAGE-FAULT-OR-INTERRUPT
IF-NO-PAGE-FAULT-OR-INTERRUPT
IF-PAGE-FAULT-OR-INTERRUPT-OR-SEQUENCE-BREAK
IF-NO-PAGE-FAULT-OR-INTERRUPT-OR-SEQUENCE-BREAK
```

If omitted, the <condition> is assumed to be "always". The <xct next>, if present, is XCT-NEXT; its absence denotes the presence of the N bit, which inhibits the instruction after the jump if the jump is successful. The three parts are connected by "-".

Examples of these operations:

```
CALL-LESS-THAN
```

```
JUMP-LESS-THAN-XCT-NEXT
CALL
POPJ-IF-BIT-SET
CALL-IF-PAGE-FAULT-OR-INTERRUPT
CALL-IF-BIT-CLEAR-XCT-NEXT
JUMP-XCT-NEXT
POPJ-XCT-NEXT
```

The POPJ-XCT-NEXT operation is not to be confused with POPJ-AFTER-NEXT, which may be used in any instruction to set the POPJ bit.

Jump instructions which perform an arithmetic comparison should have both an A and an M source; the sources are compared. Jump instructions which test a bit should have an M source and a byte specifier for a 1-bit byte to test.

Functional Sources

The following names are supplied by CONSLP for the various functional sources:

0	READ-I-ARG	The dispatch constant
1	MICRO-STACK-PNTR-AND-DATA	SPCPTR and SPC contents
	MICRO-STACK-POINTER	Byte specifier for bits <28-24>
	MICRO-STACK-DATA	Byte specifier for bits <18-0>
14	MICRO-STACK-PNTR-AND-DATA-POP	Like 1, but pops SPC stack
	MICRO-STACK-POINTER-POP	Like 1, but pops SPC stack
	MICRO-STACK-DATA-POP	Like 1, but pops SPC stack
2	PDL-BUFFER-POINTER	PDL-pointer register
3	PDL-BUFFER-INDEX	PDL-index register
5	C-PDL-BUFFER-INDEX	PDL-buffer addressed by index
25	C-PDL-BUFFER-POINTER	PDL-buffer addressed by pointer
24	C-PDL-BUFFER-POINTER-POP	PDL-buffer addressed by pointer, pop
6	OPC-REGISTER	The OPCs
7	Q-R	Q register
10	VMA	VMA register
11	MEMORY-MAP-DATA	MAP[MD]
12	MEMORY-DATA	MD
13	LOCATION-COUNTER	LC

Functional Destinations

The following names are provided by CONSLP for functional destinations. Note that some of them are the same names used for sources; CONSLP distinguishes usage by context.

1	LOCATION-COUNTER	LC
2	INTERRUPT-CONTROL	Interrupt Control Register
10	C-PDL-BUFFER-POINTER	Pdl location addressed by PDL POINTER
11	C-PDL-BUFFER-POINTER-PUSH	Push data onto pdl, increment PDL POINTER
12	C-PDL-BUFFER-INDEX	Pdl location addressed by PDL INDEX
13	PDL-BUFFER-INDEX	PDL INDEX register
14	PDL-BUFFER-POINTER	PDL POINTER register
15	MICRO-STACK-DATA-PUSH	Push data onto SPC stack
16	OA-REG-LOW	Next instruction modify, bits <25-0>
17	OA-REG-HI	Next instruction modify, bits <47-26>
20	VMA	VMA register
21	VMA-START-READ	VMA, initiate read cycle
22	VMA-START-WRITE	VMA, initiate write cycle
23	VMA-WRITE-MAP	VMA, MAP[MD] ← VMA
30	MEMORY-DATA	MD register
31	MEMORY-DATA-START-READ	MD, initiate read cycle
32	MEMORY-DATA-START-WRITE	MD, initiate write cycle
33	MEMORY-DATA-WRITE-MAP	MD, MAP[MD] ← VMA

The symbol Q-R may also be used as a destination; it causes an ALU instruction to have its Q control field to be set to "load Q from ALU output"; this is equivalent to specifying LOAD-Q in the instruction. Do not use the output bus shifter in connection with Q-R as a destination!

Operations Common to All Instructions

The symbol for the POPJ bit is POPJ-AFTER-NEXT.

Miscellaneous Function 3 is denoted by LOW-PC-BIT-SELECTS-HALF-WD. (This feature is described in greater detail in an earlier and a later section.)

Expression Programs in CONSLP

Wherever an expression may be used in CONSLP, the following arcane forms may be used. In particular, the value of a symbol is normally an expression instead of a simple number. Whenever an expression (or a symbol with an expression as its definition) is encountered, it is evaluated according to the following rules:

<number> Evaluates to itself.

(PLUS <exp1> <exp2>) Adds together the two expressions, and combines their properties (such as byte-specifier-ness).

(DESTINATION-P <exp>) A conditional: if encountered while assembling a destination, returns the value of <exp>, and otherwise NIL.

(SOURCE-P <exp>) A conditional: if encountered while assembling a source (M or A), returns the value of <exp>, and otherwise NIL.

(DISPATCH-INSTRUCTION-P <exp>) A conditional: if encountered while assembling a DISPATCH instruction, returns the value of <exp>, and otherwise NIL.

(JUMP-INSTRUCTION-P <exp>) A conditional: if encountered while assembling a JUMP instruction, returns the value of <exp>, and otherwise NIL.

(ALU-INSTRUCTION-P <exp>) A conditional: if encountered while assembling an ALU instruction, returns the value of <exp>, and otherwise NIL.

(BYTE-INSTRUCTION-P <exp>) A conditional: if encountered while assembling a BYTE instruction, returns the value of <exp>, and otherwise NIL.

(NOT <conditional>) Negation. <conditional> must be one of the above conditionsl forms.

(OR <cond1> ... <condn>) Like a LISP OR, returns the first non-NIL conditional.

(BYTE-FIELD <size> <pos>) As described earlier, defines a byte with the given size and position from the right.

(LISP-BYTE <ppss>) As described earlier; if ppss is written in octal, then this is like (BYTE-FIELD ss pp). If <ppss> is not a number, then it is a LISP expression (not a CONSLP expression!), and is evaluated in LISP.

(BYTE-MASK <byte specifier>) Value is a word which is zero everywhere except for being all ones in the specified byte. This is a kind of conditional, in that it returns NIL if the byte specifier doesn't really specify a byte.

(BYTE-VALUE <byte specifier> <value>) Value is a word which is zero everywhere, except that it contains <value> in the specified byte. This is a kind of conditional, in that it returns NIL if the byte specifier doesn't really specify a byte.

(OA-HIGH-CONTEXT <word>) Assembles <word> as an instruction, and returns the high half (bits <47-26>), as if for use by the OA register feature (next instruction modify, functional destination 17).

(OA-LOW-CONTEXT <word>) Assembles <word> as an instruction, and returns the low half (bits <25-0>), as if for use by the OA register feature (next instruction modify, functional destination 16).

(FORCE-DISPATCH <exp>) Returns value of <exp>, but also forces the instruction to be a DISPATCH instruction. A conflict causes an error.

(FORCE-JUMP <exp>) Returns value of <exp>, but also forces the instruction to be a JUMP instruction.

(FORCE-ALU <exp>) Returns value of <exp>, but also forces the instruction to be an ALU instruction.

(FORCE-BYTE <exp>) Returns value of <exp>, but also forces the instruction to be a BYTE instruction.

(FORCE-DISPATCH-OR-BYTE <exp>) Returns value of <exp>, but also forces the instruction to be a DISPATCH or BYTE instruction.

(FORCE-ALU-OR-BYTE <exp>) Returns value of <exp>, but also forces the instruction to be an ALU or BYTE instruction.

(I-MEM-LOC <tag>) Returns the address represented by <tag> in locality I-MEM as a right-justified value.

(D-MEM-LOC <tag>) Returns the address represented by <tag> in locality D-MEM as a right-justified value.

(A-MEM-LOC <tag>) Returns the address represented by <tag> in locality A-MEM as a right-justified value.

(M-MEM-LOC <tag>) Returns the address represented by <tag> in locality M-MEM as a right-justified value.

(EVAL <lisp exp>) Returns the result of evaluating in LISP the S-expression <exp>.

(FIELD <name> <value>) Makes a note that the field <name> has been specified, then multiplies together the values of <name> and <value>; if <name> has a LISP CONS-LAP-ADDITIVE-CONSTANT property, this is then added in. (This obscurity is the primitive from which all field specifications are made.)

(ERROR) Error if this is assembled. Useful in conditionals.

As examples of how conditionals might be used in expressions, consider these definitions (which are similar (but not identical) to the ones actually used in CONSLP):

```
(ASSIGN Q-R (OR (SOURCE-P (FIELD M-SOURCE 7))
                (FORCE-ALU 3)))

(ASSIGN MEMORY-DATA
        (OR (SOURCE-P (FIELD M-SOURCE 12))
            (FIELD FUNCTIONAL-DESTINATION 30)))

(ASSIGN MEMORY-DATA-START-WRITE
        (OR (SOURCE-P (ERROR))
            (FIELD FUNCTIONAL-DESTINATION 32)))
```

Miscellaneous Pseudo-Operations

Several identical words may be assembled consecutively by saying:

```
(REPEAT <count> <word>)
```

The location counter within the current locality may be set by

```
(LOC <value>)              ;sets it to <value>
(MODULO <n>)               ;advances it to the next multiple of <n>
```

If the MODULO operation is used in A-memory, wastage is avoided by filling in the skipped-over locations with constants.

CADR Features and Programming Examples

In this section the various features of the CADR machine are examined and discussed in detail. An attempt is made to give some feeling for how each feature fits into the overall structure of the machine, and the purposes for which the feature is intended. Short programming examples using each feature are presented.

Timing - The N Bit and the POPJ Bit

Because CADR fetches the next instruction at the same time it is executing the current one, by the time the effect of a JUMP or DISPATCH is known the instruction following the JUMP or DISPATCH has already been fetched. Unless suppressed by the N bit, this instruction is executed before the instruction branched to. The effect of this on programming is that one should "code the branch one instruction sooner". The mnemonics CONSLP provides for the various branching operations normally set the N bit, thus doing the straightforward thing at the cost of wasted cycles; one must append "-XCT-NEXT" to the mnemonic to clear the N bit and so bum the code.

For example, consider these two pieces of code:

```
((A-FOO) XOR M-BAR A-FOO)                    ;XOR M-BAR into A-FOO
(JUMP-IF-BIT-SET MUMBLE MUMBLIFY)            ;branch on MUMBLE bit

(JUMP-IF-BIT-SET-XCT-NEXT MUMBLE MUMBLIFY)   ;branch on MUMBLE bit
((A-FOO) XOR M-BAR A FOO)                    ;XOR M-BAR into A-FOO
```

These both perform an XOR and conditionally jump to MUMBLIFY, but the first one wastes a cycle if the JUMP is successful. Notice the convention of "exdenting" an instruction which is under the influence of an XCT-NEXT to make it more visible.

If a CALL transfer type is executed, the return address saved on the SPC stack depends on the N bit:

```
(CALL THE-SUBROUTINE)              ;call, N bit set
((A-FOO) XOR M-BAR A-FOO)          ;return here after call

(CALL-XCT-NEXT THE-SUBROUTINE)     ;call, N bit clear
((A-ARGUMENT) ADD M-BAZ A-FOO)     ;do this before entering the subroutine
((A-FOO) XOR M-BAR A-FOO)          ;return here after call
```

If the N bit is set, PC+1 is pushed on the SPC stack; otherwise PC+2 is pushed.

The POPJ bit may be set in any instruction. It causes a RETURN transfer, but only after the next instruction has

also been executed:

```
ADD-THREE-WORDS                ;subroutine to add together A-1, A-2, and A-3
        ((M-RESULT) A-1)
        (POPJ-AFTER-NEXT (M-RESULT) ADD M-RESULT A-2)
       ((M-RESULT) ADD M-RESULT A-3)
```

Again, the idea is to specify the desired control "one instruction early".

Consider the following program:

```
START   (JUMP-XCT-NEXT FOO)
       (JUMP-XCT-NEXT BAR)
        ...
FOO     (JUMP-XCT-NEXT FOO)
        ...
BAR     (JUMP-XCT-NEXT BAR)
```

When started at START, it will go into an infinite loop alternately executing FOO and BAR. Effectively it is in two "jump point" loops at the same time!

Byte Manipulation

By using M location 2 (by convention a source of zeros) with a BYTE instruction, one can clear any bit or field of bits in any A memory location:

```
((A-FOO) DPB M-ZERO A-FOO (BYTE-FIELD 1 31.))          ;clear sign bit
```

It is often convenient to reserve another M memory location to contain -1 (all ones), in order to be able to set bits easily:

```
((A-FOO) DPB M-ONES A-FOO (BYTE-FIELD 1 31.))          ;set sign bit
```

In a similar manner one can write a routine to extend a signed 24-bit number to 32 bits:

```
SIGN-EXTEND                                    ;extend 24-bit number in M-NUM
        (POPJ-AFTER-NEXT POPJ-IF-BIT-CLEAR M-NUM (BYTE-FIELD 1 23.))
       ((M-NUM) SELECTIVE-DEPOSIT M-NUM (BYTE-FIELD 24. 0) (A-CONSTANT -1))
```

Another way to do this, which doesn't require the use of POPJ, is to use OA modification to select whether the M source is M-ZERO or M-ONES:

```
        ((OA-REG-HI) (BYTE-FIELD 1 23.) M-NUM)          ;low M-source bit gets sign
       ((M-NUM) SELECTIVE-DEPOSIT M-ZERO (BYTE-FIELD 8 24.) A-NUM)
```

This requires that M-ZERO and M-ONES be an even/odd pair.

Normally bytes can only be loaded from an M source. However, it is possible to load a byte from A-memory, provided that it is at one end of the word, by the following trick:

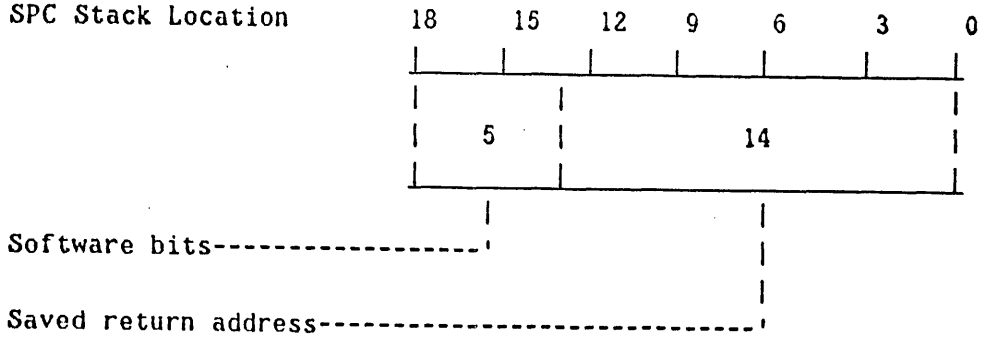
```
(DEF-DATA-FIELD X-FIELD 6 0)
(DEF-DATA-FIELD ALL-BUT-X-FIELD 32 6)

        ((DEST) SELECTIVE-DEPOSIT M-ZERO ALL-BUT-X-FIELD A-FOO)
```

The Instruction Stream

<<Some new stuff should be written for this>>

The SPC Stack

The SPC stack is 32 locations long, each location containing 19 bits (plus parity). It is indexed by SPCPTR, a 5-bit up/down counter. It is used primarily as a microcode subroutine return stack, but besides the 14 bits needed to save a microcode PC there are 5 bits for software use, one of which is the bit used for the macroinstruction pair fetch feature mentioned above.



There are two ways in which to write into the SPC stack memory; both of them also increment SPCPTR, thus causing a push operation. A JUMP or DISPATCH performing a CALL transfer type (P bit set, R bit clear) causes a return address to be pushed on the stack as described earlier. The five software bits are set to zero. Writing into functional destination 15 (MICRO-STACK-DATA-PUSH) pushes the low 19 bits of the output bus data onto the SPC stack.

The SPC stack is read by a JUMP or DISPATCH performing a RETURN transfer type (R bit set, P bit clear); the low 14 bits popped off the stack are put in the PC, and the software bits are ignored, except for bit 14 which causes NEXT-INSTR. It

can also be read as M functional sources 1 and 14. The first (MICRO-STACK-PNTR-AND-DATA) merely reads the data (and SPCPTR) on the top of the stack, while the second (MICRO-STACK-PNTR-AND-DATA-POP) pops the stack after reading the data.

There is no way to explicitly set the contents of SPCPTR. However, a good trick is to use the following loop:

```
FOO     ((M-TEMP) MICRO-STACK-POINTER-POP)          ;get just SPCPTR
        (JUMP-IF-EQUAL M-TEMP A-ZERO FOO)
```

A better trick is to use the following loop, which not only is shorter, but is recursive rather than iterative, and has the important advantage of being more obscure:

```
FOO     (CALL-NOT-EQUAL MICRO-STACK-PNTR-AND-DATA
                (A-CONSTANT (PLUS 1 (I-MEM-LOC FOO))) FOO)
```

This is a good thing to do on initialization so that the stack will begin in a known place, thus aiding debugging via the diagnostic interface.

There is no provision for detection of SPC stack overflow or underflow. It is the responsibility of the programmer to avoid nesting subroutines to a depth greater than 32.

The PDL BUFFER Memory

The PDL BUFFER is intended to be used as a special-purpose cache in the Lisp machine to contain the top portion of the Lisp pushdown stack. It has 1024 locations of 32 bits, and can be indexed by either the PDL POINTER or the PDL INDEX. PDL POINTER is a 10-bit up/down counter, while PDL INDEX is simply a 10-bit register.

The PDL BUFFER is manipulated through various functional sources and functional destinations. The PDL POINTER and PDL INDEX registers may be read and written. (On CONS, these could only be read together, but on CADR they are read separately to facilitate doing arithmetic with them without the need to extract a byte first.) The contents of the PDL BUFFER location addressed by the contents of PDL INDEX may be read and written. The contents of the location addressed by the contents of PDL POINTER may also be read and written, and in this case the PUSH and POP operations may optionally be done by incrementing or decrementing the PDL POINTER. The pointer decrements after reading and increments before writing, so it always points to the topmost valid location.

It doesn't work to specify both C-PDL-BUFFER-POINTER-PUSH and C-PDL-BUFFER-POINTER-POP in the same instruction. On the other hand, the same effect can always be achieved simply by using C-PDL-BUFFER-POINTER for both source and destination instead.

There is no provision for automatic overflow or underflow detection on pushes and pops of the PDL BUFFER. In the Lisp machine, the PDL POINTER is checked on entry to every function, and at a few other necessary places. If there is insufficient room left within the PDL BUFFER for a maximum size frame, some of the PDL BUFFER is stored into main memory to make room. If there is also insufficient space left within the

virtual memory allocated to the PDL, a PDL-OVERFLOW error is signalled. Similarly, the function exit code decides whether to pull some stack back in from main memory.

```
-*- Fundamental -*-

This file is supposed to document Unibus addresses used in the Lisp machine.
    Also see section on Xbus addresses after Unibus.

140000-177777   (in CADR)  UNIBUS region mapped to XBUS

764040  Summagraphics tablet status register. (interrupt vector 114)
764042  Tablet X register
764044  Tablet Y register (last tablet reg)

764000-764176 IOB addresses, detailed below:

764140  Chaos net -- Control/Status register.
764142  My number.
764144  Data register.
764146  Bit count register.
764152  Activate transmitter (upon being read!)

;document other IOB devices here.

764200-764216   Ethernet

766000 start of CADR processor UNIBUS area
766000-766036  CADR Spy locations
766040         CADR UNIBUS interrupt status
766042          also CADR UNIBUS interrupt related
766044         CADR XBUS error status

766100  CADR Debuggee's selected UNIBUS location
766104  CADR Debuggee's status info
766110  CADR Debuggee additional info
766114  CADR Debuggee selected address

766136  end of CADR processor UNIBUS area

766140 - 766176   CADR XBUS<-> UNIBUS mapping registers
        These map UNIBUS addresses 140000-177777.
        Bits are valid, write-enable, page.  If high 5 bits of page=1, writes MD register.
        Allocation of mapping registers:
                766140-766156   For use by software running on the same machine (the following
                                are merely suggestions):
                  766140        Ethernet
                  766142        Versatec

                766160-766170   unassigned

                766172          Debugging microprocessor

                766174          CC-WRITE-MD

                766176          DBG-READ-XBUS and related routines


776000  Cheops Control Register
776002  Cheops Status Register
776004  Cheops DBR Register
776006  Cheops MUX

777500  Versatec plotter byte count
777502  Versatec data buffer memory address extension
777504  Versatec printer byte count
777506  Versatec data buffer address
777510  Versatec plotter control and status
777512  Versatec plotter data buffer
777514  Versatec printer control and status
777516  Versatec printer data buffer

XBUS ADDRESSES (note: addresses here given in 24 bit form as on next machine.
   on current machine, the high bit must always be 0)

0          -    400000     real core
```

```
76,,776000 - 76,,777777    mapped to A memory locations 0-1777 by microcode.
                           (only 0-377 exist on current machine)
77,,000000 - 77,,377777    Xbus IO space.  see below for further breakdown
77,,400000 - 77,,777777    mapped to Unibus.   Each Xbus address is mapped to
                           one 16 bit Unibus location.
```

XBUS IO space (octant 6 of the XBUS)

Current XBUS physical address assignments

```
17,,000000 to 17,,077777        Main TV screen
17,,100000 to 17,,107777        Convolution box (offsets described below)
17,,110000 to 17,,110017        Scheme chip interface (offsets described below)
17,,140000 to 17,,177777        Audio i/o memory
17,,200000 to 17,,277777        Color TV
17,,377600 to 17,,377677        Vision hardware control
17,,377740 to 17,,377747        Audio i/o control registers
17,,377750 to 17,,377757        Color TV control
17,,377760 to 17,,377767        Main TV control
17,,377770 to 17,,377773        Second disk control
17,,377774 to 17,,377777        Disk control
```

Note: octant 7 of the XBUS cannot be accessed by the Lisp Machine,
because those physical addresses are mapped to the unibus.

CONVOLUTION BOX ADRS

```
00x xxx xxx xxx    -> PATCH MEM 0-1777    11:0
01x xxx xxx xxx    -> MASK ELEMENT 0-1777  9:0
10x xxx xxx xxx    -> MASK VALUE 0-1777   11:0
11x xxx xx0 x00    -> MASK ADR  11:0      -> MASK ADR
          0 x01    -> OFFSET     9:0      -> OFFSET
          0 x10    -> MSP                 -> START       DATA 3:0 GIVES SPEED
          0 x11    -> LSP                 -> SINGLE-STEP  ", ALSO, 100 BIT -> CLEAR DONE AND
                                                             PIPELINED DONE.
            100    ->                     -> RESET

11x xxx xx1 x00    -> READ-STATUS
```

Scheme chip interface registers:

00	RW	Clock duration - Delay 1		
01	RW	Clock duration - Phase 2		
02	RW	Clock duration - Delay 2		
03	RW	Clock duration - Phase 1		
04	RW	Write: Map address (for examine/deposit) Read: Memory address chip last referenced		
05	RW	Map data (addressed by Map Address register)		
06	RW	Control		
		Bit	RW	Function
		0	W	Reset
		1	RW	Run (turns on/off clock, clock registers should not be changed with run on)
		2	RW	External freeze (freezes chip, does not stop clock)
		3	R	Chip is frozen
		4	R	NXM (cleared by reset or mem start)
		5	R	Parity error (cleared by reset or mem start)
		6	R	Invalid map location addressed (cleared by reset or mem start)
		7	RW	Freeze on error (causes chip to be frozen when one of above conditions is true)
		8	RW	Freeze on GC (freeze when GC interrupt is set)
		9	R	GC Needed
		10	R	GC interrupt in progress
		11	R	External interrupt request
		12	R	External interrupt in progress
		13	R	Single stepping in progress
		14	RW	Read: Read request in progress Write: Mem restart (writing a 1 restarts the Scheme chip after a failing memory cycle. LISPM should have looked at the write data for a write, or stored

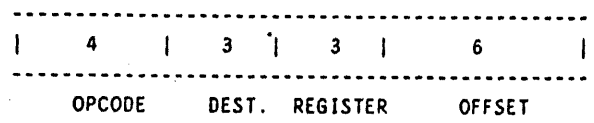
			the read data for a read)
		15 R	State hacking (reading or writing) in progress
		16 R	Chip read signal (chip is waiting for a memory cycle)
		17 R	Chip write signal (chip is starting a write)
		18 R	Chip cdr (whether current memory cycle is a car or a cdr)
		19 R	Chip is reading interrupt vector
		20 R	GC Needed
		21 R	Chip address latch enable
		22 R	Memory freeze -- chip is waiting for a memory cycle
07	RW	Single Step (single steps chip up to 256 times, only useful when Force Freeze is set and RUN is on)	
		Reading reads the current state of the counter. Useful if froze due to memory cycle	
10	RW	Reading reads latched state, writing causes state to be latched at next convenient time	
11	RW	Int Vector	
12	RW	GC vector (vector sent to chip for GC interrupt)	
13	W	Loads new state into chip	
14	RW	Memory data	
15	W	Clears external interrupt request, data is ignored	
16	W	Writing causes an external interrupt	

The LISP Machine Macro-instruction Set.

When a LISP Machine function is "macrocompiled," the compiler creates a "Function-Entry-Frame" (FEF), and puts a pointer to the FEF in the Function cell of the name of the function. The FEF contains several sections, which explain various things about the function. The last thing in the FEF is the actual MACROCODE, that is, the program.

Each macroinstruction is 16 bits long, and so two macroinstructions are stored in each word of the LISP machine. There are four conceptual "classes" of microinstructions, each of which is broken down into fields in a different way.

CLASS I:



There are nine class I instructions, designated by 0 through 10 (octal) in the OPCODE field. Each instruction has a source, whose address is computed from the "REGISTER" and OFFSET fields, and a destination given by the DESTINATION field. The instructions are:

OPCODE	NAME	
0	CALL	Open a call block on the stack, to call the function specified by the address. Whatever the function returns will go to the destination. The actual transfer of control will not happen until the arguments have been stored. (See destinations NEXT and LAST.)
1	CALL0	CALL in the case of a function with no arguments. The transfer of control happens immediately.
2	MOVE	Move the contents of E to the destination.
3	CAR	Put the CAR of the contents of E in the destination.
4	CDR	Analogous.
5	CADR	Analogous.
6	CDDR	Analogous.
7	CDAR	Analogous.
8	CAAR	Analogous.

The effective address, E, is computed from the "register" and the offset. The instructions really use addressing relative to some convenient place specified by the "register" field. The register may be:

REG	FUNCTION	
0	FEF	This is the starting location of the currently-running FEF, and is useful for addressing special variables, constants not on the constants page, etc.
1	FEF+100	Same as 0, plus 100 octal.
2	FEF+200	Analogous.
3	FEF+300	Analogous.
4	CONSTANTS PAGE	This is a page of widely used constants, such as T, NIL, small numbers, etc.
5	LOCAL BLOCK	This is the address of the local block on the PDL, and is useful for addressing local variables.
6	ARG POINTER	This is the argument pointer into the PDL, and is useful for addressing arguments of the function.
7	PDL	This is the pointer to the top of the stack. The offset is taken as negative. Thus PDL+0 refers to the last thing pushed, PDL+1 to the second to last, etc. PDL+77 is a special case. It means the operand should be popped off the stack.

(See the FORMAT file for how the PDL frame for each function is divided up into header, argument block, local block, and intermediate result stack.)

Note: The first 4 addressing modes are all provided to allow an effective 8-bit offset into the FEF.
Note: The same register-offset scheme is used in the class II instructions.

An additional complication in computing the "effective address" comes from invisible pointers. Once the register and offset have been used to compute an initial effective address E, the word at that location is examined (even if this is an instruction which uses E as a destination.) If the data type of that word is "Effective Address Invisible", the pointer field of that word is used as the effective address E. This is used, for example, to access value cells of special variables. The FEF "register" is used, and the location of the FEF addressed contains an effective address invisible pointer which points to the desired value cell. This scheme saves bits in the instruction, without requiring the use of extra instructions to make special value cells addressable. If the data type is "Garbage Collector Invisible", the word pointed to by that word is fetched and stored into that word.

The destination field is somewhat more complicated. First of all, before the result is moved to the destination, two "indicators" are set. The indicators are each stored as a bit, and correspond to processor status flags such as N and Z on the PDP-11. They are called the ATOM indicator, which is set if the result of the operation is an atom, and the NIL indicator, which is set if the resuly is NIL. The class III instructions (BRANCH) may look at the indicators.

Note: On the real machine, there are not actually any physical indicators. Instead, the last result computed is saved in an internal register, and examined by the BRANCH instructions. The functional effect is the same.

The destinations are:

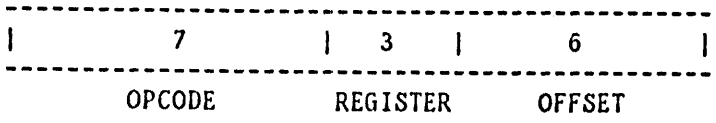
DEST	FUNCTION	
0	IGNORE	This is the simplest; the result is simply discarded.

		It is still useful, because it sets the flags.
1	TO STACK	This pushes the destination on the stack, which is useful for passing arguments to Class IV instructions, e
2	TO NEXT	This is actually the same thing as TO STACK, but it is used for storing the next argument to the open function when it is computed.
3	TO LAST	This is used for storing the last argument of the open function. It also pushes the result on the stack, and then it "activates" the open call block. That is, control will be passed to the function specified by the last CALL instruction, and the value returned by the function will be sent to the destination specified by the destination field of the call instruction.
4	TO RETURN	Return the result of the instruction as the value of this function. (i.e. return from subroutine.)
5	TO NEXT, QUOTED	This is the same as TO NEXT, except that for error checking, the USER-CONTROL bit of the word being pushed is set, telling the called function that it is getting a quoted argument (if it cares).
6	TO LAST, QUOTED	Analogous.
7	TO NEXT LIST	This one is fairly tricky. It is used in conjunction with the LIST (Class IV) instruction to efficiently perform the lisp "LIST" function. It is documented under the LIST instruction.

Note: 5 and 6 (the QUOTED) destinations have not been implemented as of 11/03/76.

Note: The same DESTINATION field is used by the class IV instructions.

CLASS II:



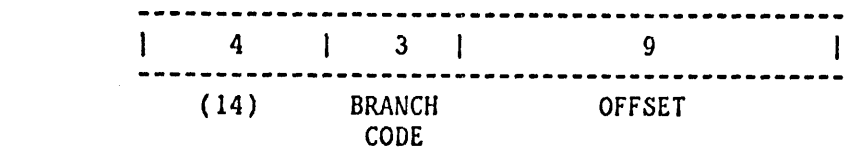
The class II instructions have no destination field; the result of the operation (if any) is either pushed on the stack (like a destination TO STACK or TO NEXT in a class one instruction) or is stored at the effective address. The "register" and offset are used in exactly the same way as in the class I instructions, except that the E calculated is sometimes used as a destination instead of a source.

The instructions are broken up into three subgroups by the first three bits of the opcode [in the microcode they are referred to as Non-destination instruction groups 1, 2 and 3], and then into the separate instuctions by the next four bits as follows: (a "-" in the left hand column means that this instruction pops the stack; a "+" means that it pushes something onto the stack.)

GRP.	OPCODE	FUNCTION	
11	0	MOVEM	Move the data on top of the stack to E.
11	1	+	Adds C(E) to the top of the stack and replaces the result on the stack.
11	2	-	Subtracts C(E) from the top of the stack.
11	3	*	Analogous.
11	4	/	Analogous.

	11	5	AND	Analogous.
	11	6	XOR	Analogous.
	11	7	OR	Analogous.
-	12	0	EQ	\ These compare C(E) to the top of the stack, and if the
-	12	1	>	|-condition being tested is true, the NIL indicator is cl
-	12	2	<	/ otherwise it is set. The stack is popped.
-	12	3	POP	Pop the top of the stack into E.
	12	4	SCDR	Get the CDR of C(E), and store it in E.
	12	5	SCDDR	Analogous.
	12	6	1+	Analogous.
	12	7	1-	Analogous.
	13	0	BIND	The cell at E is bound to itself. The linear binding pdl is used.
	13	1	BINDNIL	The cell at E is bound to NIL.
-	13	2	BINDPOP	The cell at E is bound to a value popped off the stack.
	13	3	SETNIL	Store NIL in E.
	13	4	SETZERO	Store fixnum 0 in E.
+	13	5	PUSH-E	Push a locative pointer to E on the stack.
	13	6		Unused.
	13	7		Unused.

CLASS III

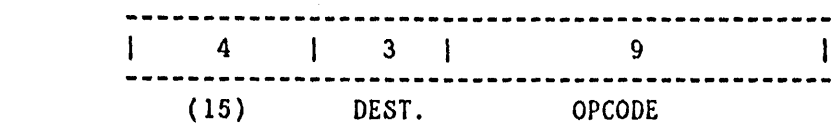


The class III instruction is for branching. There is a 3 bit Branch Code field which determines whether the branch happens, and sometimes what to do if it fails. It is decoded as follows:

BRANCH CODE	FUNCTION	
0	ALWAYS	Always branch.
1	NILIND	Branch if the NIL indicator is set, else drop through.
2	NOT NILIND	Branch if the NIL indicator is not set, else drop through.
3	NILIND ELSE POP	\ These two are the same as NILIND and NOT NILIND,
4	NOT NILIND ELSE POP/	| except that if the condition fails, the stack is popped.
5	ATOMIND	Branch if the ATOM indicator is set, else drop through.
6	NOT ATOMIND	Analogous.

If the decision is made to perform the branch the offset, considered as a signed (two's complement) offset is added to the PC (i. e. a relative branch, such as is used by the PDP-11). If the offset is 777, however, it is interpreted as meaning that this is a long-distance branch, and the real offset is obtained from the next (16-bit) halfword. The PC added to is always the incremented PC; i.e. the address of the instruction +1 in the short case, and the address of the instruction +2 in the long case.

CLASS IV



The class IV (miscellaneous) instructions take their arguments on the stack, and have a destination field which works the same way as the Class I instructions. They all have 15 (octal) in the first 4 bits, and the actual opcode is in the last 9 bits. Thus there can be up to 512. of them. Most of them may be called directly from interpretive LISP, and so some of them duplicate functions available in classes I and II. [Note on implementation: since there are far too many class IV instructions to dispatch on using the dispatch memory of the CONS machine, the starting locations of the routines are kept in main memory. The location of the base of the dispatch table is kept in A-V-MISC-BASE at all times.]

Since most of these functions are callable from interpretive level (the normal LISP user sees them), they form part of the nuclear system, and so are documented in the Lisp Machine Nuclear System document (LMNUC >).

The first 200 (octal) Class IV operations are not in the dispatch table in main memory, but are specially checked for. These are the "LIST" instructions, which work in cooperation with the NEXT-LIST destination.

Operations 0-77 are called LIST 0 through LIST 77. The list <N> instruction allocates N Q's in the default consing area [A-CNSADF] which is initialized to a CDR-NEXT, CDR-NIL style list of NILs. Then the instruction pushes three words on the stack: 1) A pointer to the newly allocated block, 2) The destination foeld of the LIST instruction, 3) A pointer to the newly allocated block (another copy). Note that the destination, as in the CALL instruction, is not used instantly; it is saved and used later.

After the LIST instruction has been performed, further instructions can store to destination NEXT-LIST; once the macro-code computes the next arg of what was the LIST function in the source code it stores it to NEXT LIST. What destination NEXT LIST does is: the word on the top of the stack is taken to be a pointer to the next allocated cell. The pointer is popped, the result of the instruction is stored where it points, and then the CDR of the pointer is pushed back on to the stack. If the CDR was NIL however, then we must be finished with the LIST operation, so the NIL is popped off the stack and discarded and a pointer to the newly allocated area (another copy of which was thoughtfully stored on the stack) is sent to the destination of the LIST <n> instruction (which was also stored on the stack), and the two remaining words which the LIST <n> pushed are popped off.

CLASS V

Opcodes 16 and 17 (octal) are not used and reserved for future expansion.

Lisp Machine storage conventions (attempt 4)

There are three kinds of lisp objects:

Type:	What it points to:
Inum-type	Nothing. The data is stored as an "immediate" number in the pointer field. This includes DTP-FIX, DTP-U-ENTRY, etc.
List	one or two Qs, identified by cdr codes (however, the GC tries to be clever, see below).
Structure	any number of Qs. The first Q in a structure always has a special data-type identifying it as a header, and from there the size and type of the structure can be determined. Cdr codes are also often used, to make part or all of a structure look like a list.

The machine needs to be able to tell what type storage is in three situations:

Normal reference - the data-type field of the object used to refer to the storage gives the information. For some types, additional breakdown occurs from a field in the header Q.

GC copy - when a pointer is discovered to point to storage in OLD SPACE, the storage must be copied before it can be accessed (see gc stuff below). There could be a pointer into the middle of an object, so it is necessary to find the whole containing structure, and discover its size and type, when given a pointer into any legal place in the middle.

GC scan - when the GC is scanning through already copied objects, looking to copy their sub-objects, it needs to be able to tell what type storage is just by looking at its first Q (since it is scanning linearly through memory.) If the storage is a structure, the Header data-type in the first Q gives this information. Otherwise the storage must be a list. This can work off of either the data type of the first Q or the storage-type of the containing area.

This generates rules for what objects and pointers can be legally used:

You may never have a pointer to "unboxed data", for instance the contents of a string or the macro-code portion of a fef. If such a pointer existed, in situation 2 above one would not be able to find the front of the containing structure because there would be non-data-type-containing Qs in the way. This means you may not even take an interrupt with the VMA pointing at unboxed data. This puts special requirements on interrupting out of a macrocode instruction fetch.

No Q may contain Header data-type except one which is the header of a structure. One may not generate a header in a variable and then store it. One may not even pass it as arg; it has to be generated in the place where it is finally going to be, using, for exmaple, %P-DPB or %P-DPB-OFFSET.

This means the READ-MEMORY-DATA and WRITE-MEMORY-DATA registers cannot be saved as part of the machine state when switching stack groups, because they cannot be guaranteed to contain type information. Therefore interrupts have to occur logically before reads and after writes.

In fact, it looks like interrupts will have to be disallowed completely after writes, in order to be sure the right thing is in MEMORY-MAP-DATA.

Any micro routine which manipulates unboxed data, or which could be called by the user to manipulate such data, or to manipulate header data, may not operate on such data in a register which is saved as part of the machine state when switching stack groups, unless it GUARANTEES to put something else in that register before the first possible error or interrupt. There are currently lots of violations of this. However it is OK to leave small un-typed numbers in a register, when they are sufficiently small in magnitude that the type field will be 0 or -1 (37).

Again, you even have to be careful about taking an interrupt with the VMA pointing at a non-typed location. You don't have to be careful about leaving the VMA pointing at such a thing when returning since it will always be changed before the next time interrupts are tested for (what about illops?)

Note that the "machine state", consisting of the contents of the saveable M and A registers and the contents of the pdl buffer, is logically all contained in the contents and leader of one array (the currently-running stack-group array).

Additional problems are caused by partially-built structures such as arrays and symbols. We need a non-interruptible microcode routine which allocates a specified amount of storage (via IABL) and stores the necessary headers into it. Note that when making an array it may take as many as 4 Qs to have a full header structure if it has a leader and a long-index-length. Probably one routine used for making symbols, fefs, and extended numbers should exist:

```
%ALLOCATE-AND-INITIALIZE <data type to return> <data type for header>
                         <header as fixnum> <area> <number of qs>
                                                   ^-- or as array pointer when
                                                       making a symbol.
```

And another routine used for making arrays:

```
%ALLOCATE-AND-INITIALIZE-ARRAY <header-as-fixnum> <index length> <leader length>
                               <area> <total number of qs>
```

PDL Numbers

When using extended numbers, each intermediate result has to be consed up in storage. To decrease the required number of garbage collections, the following hack is used. There is a special area, PDL-NUMBER-AREA, which is used only to contain extended numbers. Numbers in this area can only be pointed to by objects of type DTP-PDL-NUMBER, and such objects can ONLY be stored "in the machine" (in A and M memory and in the PDL Buffer). When such a pointer is stored into memory, a new number must be consed up in some regular area such as WORKING-STORAGE-AREA, and a pointer to this copy (with DTP-EXTENDED- NUMBER) is stored in place of the original pdl number pointer.

This works because EQ is not defined for numbers (for programming convenience, EQ is defined for fixnums, but people should use =), for extended numbers EQ is not defined (but probably doesn't signal an error either) and you have to use EQUAL. This is similar to the situation in Maclisp.

Values returned by extended-number functions are consed in PDL-NUMBER-AREA. When this area becomes full, it is not necessary to do a full garbage collection. Since there can be no pointers to this area outside of the machine, it is only necessary to do a stack-group-leave and a stack-group-enter, which will delete all pointers to the pdl-number area, and then reset the allocation pointer to the beginning.

The "PDLNMK" operation of checking for storing of a DTP-PDL-NUMBER object into memory can be done at no extra cost, since a dispatch on type and map is already being done during writing anyway. See below.

Note that when storing a pdl number into a block-type multiple value return, this looks like storing into memory, and consequently would cause copying and consing. This could be avoided by putting in a special case check where, when you think you just stored a pdl-number into memory, if that virtual address is really in the pdl buffer, you leave it there.

The Garbage Collector

The garbage collector will be an incremental copying algorithm. Virtual address space is quartered into three semispaces:

STATIC	Stuff which is not subject to copying. This includes the special areas in low core, magic addresses which refer to the Unibus or to A&M memory, and areas declared by the user to be static. Static areas may be (1) scanned by GC but not copied e.g. the "initial" areas in low core and the macro-compiled-program area. (2) connected to the outside world only via an exit area (3) Not used for typed data at all (e.g. the Unibus, the page-table-area.)
OLDSPACE	Contains lisp data which has not yet been proven to be either useful or garbage. Useful stuff in this space will be copied to NEWSPACE.
NEWSPACE	Contains lisp data which has been proven to be useful (it may have become garbage since that time.)

Each area is either STATIC or has an OLDSPACE region and a NEWSPACE region.

After a gc cycle is complete, a "flip" occurs in which OLDSPACE and NEWSPACE are exchanged, and the new NEWSPACE is made empty. This can only happen after a complete cycle since you need to have proved that nothing in OLDSPACE needs to be retained, and it can be recycled as a NEWSPACE.

Note that the operations of "flipping" (changing newspace into oldspace and starting a new newspace) could be logically separated from the operation of reusing the old space. The former can be done at any time if there is a reason to. (Would then require an area to have more than two regions.)

The division of virtual address space into these spaces is on a page by page basis, but really an area-by-area basis, at least for now. Each low-level area (i.e. index in AREA-ORIGIN etc.) is all in a single semi-space. When a page is swapped in, the AREA-MODE-BITS table is used to set up a bit in the map which is on for STATIC or NEWSPACE, and off for OLDSPACE or map not set up.

Each high-level area (seen by the user) is either static or consists of two low-level areas, one for OLDSPACE and one for NEWSPACE. (In future there may be more than two due to area expansion.) (When an area becomes full the options are (1) to expand it by adding another low-level area to it, or (2) to finish the current gc cycle in a non-incremental mode, then flip all areas.) [If you're lucky, the current gc cycle may already be finished.]

The garbage collector is continuously copying accessible storage from OLD SPACE to NEW SPACE. Rather than doing this all at once, it is done incrementally. When everything has been copied, a "flip" occurs; OLD SPACE and NEW SPACE are interchanged.

The problem of things that type out as e.g. #<DTP-FOO 123456>. If the user is remembering these octal numbers, and they change because of the garbage collector, confusion could arise. The simplest solution is a "debugging mode" flag which prevents flips (full areas would expand instead). When setting this flag is set you also need to do a complete gc cycle to make sure there are no

old spaces around so no copying happens. A way to make PRINT tag them with generated or user-specified names could also be done.

The Barrier

For reasons of simplicity and efficiency, the rule is made that no pointer to OLD SPACE can be "in the machine", therefore one only has to check when something is picked up out of memory. "In the machine" means in A&M memory or in the pdl buffer (as opposed to in main memory). A few A&M memory locations (not saved in stack group) are not "in the machine" in order to make things work.

When a FLIP occurs, every pointer in the machine has to be changed to point to NEW SPACE rather than OLD SPACE before the machine can run. (Immediately after a flip it's known that all pointers in the machine point to OLD SPACE.) Probably the stack-group mechanism can be used to do this.

When the machine is running, if a pointer to storage in OLD SPACE is discovered in memory, the storage must be copied into NEW SPACE before the pointer can be used. Also, the storage may have been copied already, in which case the prior copy should be used.

This copier is sometimes referred to as the Transporter (to distinguish it from the parts of the GC that can get invoked when you do a CONS.)

This barrier applies to reads from main memory. There is also a barrier for writes, which says that when certain data types are written into certain areas, special things have to happen. This will be used for automatic maintenance of exit-vectors for static areas, and also for pdl numbers (mentioned above.)

The exact algorithm is:

Immediately after a memory read of typed data, one has to check both the data type of the word fetched from memory and what semispace its pointer field points to. See next page for the exact instructions.

1) Let A be the address referenced and B the word read from that address. Note that the data type of A will always be some pointer type and the pointer field will always point to NEW SPACE or STATIC SPACE, since A came from "in the machine."

If the data-type of B is an "Inum-type" type, do nothing special; result (R) is B.

Otherwise the data type of B is a pointer type. If the pointer field points to NEW SPACE or STATIC SPACE, R := B and go to step 3.

2) Otherwise, we have just picked up a pointer which is not allowed to be in the machine because it points to OLD SPACE. Fetch the contents of the location pointed to by B, and call that C.

i) If C is of type DTP-GC-FORWARD, then it had better point to newspace.
R := the pointer field of C combined with the datatype field of B.

ii) Else, C is of a normal datatype. B might have been pointing into the middle of any sort of structure. Look at the datatype of B to determine the format of the structure. Also, look at the %%AREA-MODE-STORAGE-TYPE byte of the AREA-MODE-BITS entry for the area B points to. This indicates whether it is a list cell or a structure with a header Q. For header-type structures (everything other than list cells), scan backwards through memory from the place B points to until a location containing a header data-type is encountered. The structure will then be defined. Copy each Q of the structure into newspace, leaving behind a load of forwarding pointers. (DTP-GC-FORWARD). Each forwarding pointer points at the corresponding word in the copy, rather than the first word. During the copy DTP-FORWARD pointers can (probably) be snapped out.
In the case of list cells, check the cdr code to see whether the cell is one Q long or two. (If the cdr code is CDR-ERROR, this must be the second cell of a two-Q cell, provided the hack given below of using CDR-ERROR in forwarded cells is not pulled.)
R := the correct new address, with B's datatype.

You then "snap out", i.e. you store R in the location pointed to by A (where B came from).

3) R now has an object which is a candidate for being returned.

i) If it is not any sort of invisible, simply return it.

ii) If it is a DTP-FORWARD, the storage A points at was moved. The pointer field of R points at the corresponding Q of the new copy. Invisibleness has to happen.
If it is an external value cell pointer, and this is a setq or symeval type operation (including function cell, compiled stuff, etc.), or if it is a one q forward, invisibleness also has to happen. The pointer field of R points to the Q which is to be substituted for the Q we just accessed. So take R, use it as A, and go do a memory read as appropriate (don't do the whole operation over, e.g. in CDR don't look at cdr codes, but do make this

check on the datatype and address again).

NOTES: Writing operations that write into typed memory have to first do a read, then hopefully the VMA (really M-VMA-HELD) will end up set up to the correct location to be written. May not check for interrupts before doing the write. GC copying operation has to be sure about leaving the VMA set up properly. (The VMA is essentially A above.)

The difference between DTP-FORWARD and DTP-ONE-Q-FORWARD is as follows:

(1) When doing CDR (or RPLACD) when the first word of the CONS is read, if its type is DTP-FORWARD, ignore the cdr code and set the address of the first word of the cons to be used from the pointer field of the forwarding pointer. An alternative is to only do this if the cdr code is CDR-ERROR (so when RPLACD copies a CDR-NIL or CDR-NEXT node into a full node, it would change the cdr code of the original Q.) (See above for a reason not to do this.)

(2) When referencing a structure containing unboxed numbers (e.g. a FEF or a string array) one should always (UNINTERRUPTIBLY) check the header Q of the structure for a DTP-FORWARD before making the reference so as to win if it has been moved elsewhere. It is not necessary to do this when referencing a typed location, since when a structure is copied it is filled with forwarding pointers. This includes operations such as %P-LDB-OFFSET and so forth. This is not necessary to make the GC work, only to make non-gc copying such as in RPLACD and ADJUST-ARRAY-SIZE work.

Since it would not be reasonable to make this check on every instruction fetch, and even a little painful to do it on function entry and exit and so forth, probably we should make the restriction that only the garbage collector can copy fefs. It's not as important for fefs as for arrays anyway.

Proposed implementation depends on the machine:

On the present machine:
(which will be modified to allow a bit in the dispatch-constant to select the low bit of the dispatch address to come from a bit in the map. Eventually we would like to be able to select at least two and preferably three bits from the map.)
(See LMDOC; OPCS > for how to use the OPC registers to find out where the call came from.)

```
((VMA-START-READ M-VMA-HELD) ...) ;note need to hold VMA since gets clobbered
(CALL-CONDITIONAL PG-FAULT ...)          ;when addressing map.  SG must restore, too.
...
((VMA) READ-MEMORY-DATA)                 ;to address the map
(DISPATCH DISPATCH-LOW-BIT-FROM-MAP-STATIC-OR-NEWSPACE-BIT
          Q-DATA-TYPE-PLUS-ONE-BIT READ-MEMORY-DATA
          D-nnn)            ;Call if oldspace, invisible, or map not set up
;at this point READ-MEMORY-DATA and M-VMA-HELD are valid.
;Note that if we really wanted to write, VMA has been
;clobbered and has to be reloaded from M-VMA-HELD, on the
;next machine this won't be true.  An alternative scheme is
;to not test the map if we really wanted to write (and were
;reading only to check invisible pointers (and copy cdr code)).
```

On future machine offerings the M-VMA-HELD register and the ((VMA) READ-MEMORY-DATA) instruction may be dispensed with, since the map will be automatically addressed from the READ-MEMORY-DATA. Whether an instruction must be inserted in place of the ((VMA) READ-MEMORY-DATA), which awaits the memory cycle, in order to give the map time to set up, remains to be seen. Note that the old machine's code will work on the new machine, in any case. One possibility is to put an instruction such as

```
((M-T) Q-TYPED-POINTER READ-MEMORY-DATA)
```

there (just before the dispatch), and arrange that if the dispatch jumps away it returns to that instruction.

D-nnn can be any of various dispatch tables, depending on whether what we are doing is reading or writing and on which types of forwarding pointers should be invisible. The difference with writing is that we don't want to invoke the transporter, since the object pointed to by READ-MEMORY-DATA is being discarded. In addition, as indicated in the comment above, the dispatch may be on just the data type and not the map.

D-TRANSP	transporter dispatch for car/cdr operations. DTP-FORWARD and DTP-ONE-Q-FORWARD are invisible.
D-VC-TRANSP	transporter dispatch for value cell referencing operations. DTP-FORWARD, DTP-ONE-Q-FORWARD, and DTP-EXTERNAL-VALUE-CELL-POINTER are invisible.
D-HDR-TRANSP	transporter dispatch for header Q referencing operations. Only DTP-FORWARD is invisible. The other 3 kinds of forwarding pointers are ILLOPs. (This dispatch is somewhat optional, D-TRANSP could be used.) The intent is we are trying to see if the whole structure has been moved.
D-TRANSP-W	transporter dispatch for rplaca/rplacd operations. DTP-FORWARD and DTP-ONE-Q-FORWARD are invisible. This dispatch is for a read cycle that preceeds a write, and doesn't test the map.
D-VC-TRANSP-W	transporter dispatch for value-cell writing operations. DTP-FORWARD, DTP-ONE-Q-FORWARD, and DTP-EXTERNAL-VALUE-CELL-POINTER are invisible.

```
;note that the dispatch usually drops through,
;but sometimes it calls routines which think
;about garbage collection then return to the
;dispatch.  (Or the instruction before the
;dispatch if there has to be such an instruction
;for timing of the automatic map-from-vma on the new machine.)
;Also note that it works to put POPJ-AFTER-NEXT
;in the instruction BEFORE the dispatch, since the
;return address comes from OPC not USP.  This has to
;be done this way for ILLOP, and saves a good deal of time.)
;Another possibility is to have a POPJ-AFTER-NEXT
;on the dispatch itself.  This doesn't win with the
;current hardware (see LMDOC; POPJ >) but could be made
;to by having PR popj instead of dropping through and
;the call and jump transfers just jump without popping
;the SPC.  Then the POPJ would only happen when the
;dispatch dropped through, and would be deferred if
;you went off to the transporter.
```

When the dispatch dispatches, the first thing is to use the OPC to save the return address. Also, a POPJ may have happened before the dispatch, in which case that return address has to be pushed back onto the USP. Probably the best way to do this is to use a bit in the dispatch constant to signal that this has occurred.

Next, if the map was not set up, the call may be spurious, so the page-fault routines should be called to set up the map, then the dispatch should be re-executed.

Otherwise, we got here from the dispatch either because the word from memory has a pointer-type datatype and addresses a page in old space, in which case we want to enter the transporter, or because the word from memory has an invisible-pointer type data type, in which case we want to do

```
((VMA-START-READ M-VMA-HELD) DPB                    ;fetch location pointed to by invz
        READ-MEMORY-DATA Q-POINTER A-VMA-HELD)
(CALL-CONDITIONAL PG-FAULT PGF-R)
(jump back to the dispatch)
```

The registers clobberable by the transporter, and the maximum amount of pdl buffer space used, remain to be defined.

Implementation of the writing barrier (next machine only).

```
((WRITE-MEMORY-DATA) ...)
((VMA-START-WRITE) ...)                 ;other order works too
(CALL-CONDITIONAL PG-FAULT PGF-W)
(DISPATCH Q-DATA-TYPE-PLUS-ONE-BIT
  DISPATCH-LOW-BIT-FROM-MAP-NOT-EXIT-VECTOR-TYPE-AREA-BIT
  Q-DATA-TYPE WRITE-MEMORY-DATA D-W-BARRIER)
```

D-W-BARRIER implements PDLNMK, and also checks for storing a data type other than an Inum into an area which has to go indirect through an exit vector.

Note that in this dispatch the type bits come from the memory data, but the map is addressed by the VMA (in the read case it was addressed by the data.) This implies that INTSER-WRITE has to be removed, because if you interrupted and switched stack groups, upon return the last cycle would have been a read, and the map would be addressed from the data rather than the address. INTSER-WRITE and the stack-group-restore code could be hacked up to win, of course, but it's easier to remove it and it won't hurt. Note that interrupts are being revised anyway.

Unlike the read case, the map-not-set-up condition shouldn't happen. The sense of the bit could be reversed, to be normal-zero rather than normal-one, if desired.

Data type details - new revised scheme

Criteria for assigning codes:
  0 and 37 should be traps to help catch random numbers as pointers.
  Once you know an object is a function [C(AP)] a 3-bit dispatch
   should be able to tell what type it is, i.e. all the functions should
   be together. [Not very important but saves some time or some dispatch
memory.]
  When writing dispatch tables, it would be easier if similar types were
   on consecutive codes.
Where possible remain similar to present order.

Census:	Type class	Count	
	"Inum"	6	- counting two traps
	List	5	
	Structure	9	
	Forwarding	4	
	not yet used	8	
	Total	32	
	Functions	5	

U-ENTRY, LIST, FEF-POINTER, ARRAY-POINTER, MESA-FEF-POINTER

"Inum" types:

DTP-FIX	
DTP-U-ENTRY	
DTP-HEADER	For xnums, fefs, leaders, and any new structured types
DTP-ARRAY-HEADER	
DTP-TRAP (also DTP-37TRAP or whatever you want to call it)	

List types (don't point to headers, but may point to WITHIN a structure.)

DTP-LIST	
DTP-LOCATIVE	
DTP-CLOSURE	
DTP-FREE	this may be getting removed, not clear.
DTP-INVOKE	unclear what this points to but probably a list

Structure types

DTP-EXTENDED-NUMBER	Points to EXTENDED-NUMBER header. (not yet)
DTP-PDL-NUMBER	Points to ditto, but not a real structure exactly.
DTP-SYMBOL	Points to SYMBOL HEADER word.
DTP-SYMBOL-HEADER	Points to ARRAY-HEADER of pname.
DTP-NULL	Points to SYMBOL HEADER word.
DTP-FEF-POINTER	Points to FEF HEADER word.
DTP-ARRAY-POINTER	Points to ARRAY HEADER word.
DTP-STACK-GROUP	Points to ARRAY HEADER word.
DTP-MESA-FEF-POINTER	Points to MESA FEF HEADER word.

Obsolete:

DTP-MEM-POINTER	Can go away since LMI has.
DTP-LOCATIVE-INTO-SYMBOL	
DTP-LIST-INTO-SYMBOL	
DTP-ARRAY-LEADER	Offset to ARRAY HEADER. (Uses regular DTP-HEADER now.)

Invisible pointers etc.
 DTP-GC-FORWARD
 DTP-FORWARD
 DTP-ONE-Q-FORWARD
 DTP-EXTERNAL-VALUE-CELL-POINTER

Numbering scheme

```
0       DTP-TRAP
1       DTP-NULL
2       DTP-FREE
3       DTP-SYMBOL
4       DTP-SYMBOL-HEADER
5       DTP-FIX
6       DTP-EXTENDED-NUMBER
7       DTP-PDL-NUMBER
10      DTP-HEADER
11      DTP-INVOKE
12      DTP-GC-FORWARD
13      DTP-EXTERNAL-VALUE-CELL-POINTER
14      DTP-ONE-Q-FORWARD.
15      DTP-FORWARD
16      DTP-LOCATIVE
17      DTP-LIST
20      DTP-U-ENTRY
21      DTP-MESA-FEF-POINTER
22      DTP-FEF-POINTER
23      DTP-ARRAY-POINTER
24      DTP-ARRAY-HEADER
25      DTP-STACK-GROUP
26      DTP-CLOSURE
27-36   not used
37      trap
```

Function type from bottom 3 bits: (does this include all types that can appear in contents of pdl buffer addressed by M-AP ?)

```
0       DTP-U-ENTRY
1       DTP-MESA-FEF-POINTER
2       DTP-FEF-POINTER
3       DTP-ARRAY-POINTER
4
5
6
7       DTP-LIST
```

Types of header:

DTP-HEADER - pointer field is bit decoded including a subtype field of 5 bits,
%%HEADER-TYPE-FIELD.

```
0       %HEADER-TYPE-ERROR
1       %HEADER-TYPE-FEF
2       %HEADER-TYPE-ARRAY-LEADER
3       %HEADER-TYPE-MESA-FEF
4       %HEADER-TYPE-FLONUM
5       %HEADER-TYPE-COMPLEX
6       %HEADER-TYPE-BIGNUM
6       %HEADER-TYPE-RATIONAL
```

DTP-SYMBOL-HEADER - pointer field is address of the pname (implies DTP-ARRAY-POINTER)

DTP-ARRAY-HEADER - pointer field is bit decoded including a subtype field of 5 bits
- as now

Mapping from current types (3/7/77):

```
0       DTP-TRAP
1       DTP-NULL
2       DTP-FREE
3       DTP-SYMBOL
  --> insert DTP-SYMBOL-HEADER
4       DTP-FIX                     --> 5
5       DTP-EXTENDED-NUMBER         --> 6
  --> insert DTP-PDL-NUMBER, DTP-HEADER
6       DTP-INVOKE                  --> 11
7       DTP-GC-FORWARD              --> 12
10      DTP-SYMBOL-COMPONENT-FORWARD    --> rename --> 13
11      DTP-Q-FORWARD                   --> rename --> 14
12      DTP-FORWARD                 --> 15
13      DTP-MEM-POINTER             --> remove
14      DTP-LOCATIVE-TO-LIST        --> rename --> 16
15      DTP-LOCATIVE-INTO-STRUCTURE  --> delete
16      DTP-LOCATIVE-INTO-SYMBOL  --> delete
17      DTP-LIST                    --> 17
20      DTP-LIST-INTO-STRUCTURE --> delete
21      DTP-LIST-INTO-SYMBOL    --> delete
22      DTP-U-ENTRY                 --> 20
23      DTP-MESA-ENTRY              --> rename to DTP-MESA-FEF-POINTER --> 21
24      DTP-FEF-POINTER             --> 22
25      DTP-FEF-HEADER              --> delete
26      DTP-ARRAY-POINTER           --> 23
27      DTP-ARRAY-HEADER            --> 24
30      DTP-ARRAY-LEADER            --> delete (use DTP-HEADER)
31      DTP-STACK-GROUP             --> 25
32      DTP-CLOSURE                 --> 26
```

Note the order does not change, just insertions and deletions.

Per-area data required.
(These are the "low-level" areas.)
(* indicates wired)

AREA-ORIGIN *	virtual base address
AREA-LENGTH *	number of Qs virtual address space
AREA-MODE-BITS *	bits to go in map (access, status, static or newspace, ...) free storage mode may be obsolete storage mode (error (e.g. page table), indexed (e.g. area-name), list, or structure.) semispace type (Note CONS has to check this.) (new, old, static)
AREA-NAME	I suggest this normally be a symbol, but be a number if this is a sub-area of some other area. (A gced area has two semispace areas, one of which (arbitrarily) has the name and the other has the other's number.
AREA-SUBAREA-THREAD	Contains the area number of the next subarea in this logical area, or fixnum -1 if this is the last.
AREA-FREE-POINTER	Not valid in old semispace subarea. Relative address of the next Q to be allocated.
AREA-GC-SCAN-POINTER	Not valid in old semispace subarea. Relative address of the next Q to be scanned by the garbage collector. (The gc incrementally scans all areas looking for pointers to oldspace, besides being invoked when a pointer to oldspace is found during the normal course of the program.)

Can AREA-PARTIALLY-FREE-PAGE and AREA-FREE-PAGE-LIST be removed?

What about symbols such as WORKING-STORAGE-AREA ? When a flip happens, should their values be changed to the new subarea number? Maybe ONE-Q-FORWARD pointers should be stored in oldspace AREA-FREE-POINTER? Probably all functions that use these symbols will have to be changed anyway to know about subareas.

A good example of an area which should be static is MACRO-COMPILED-PROGRAM. P-N-STRING is probably another.

Issues about whether to do anything special with the OBARRAY.

Temporarily making a static area into an active one so that it gets compacted.

Certain functions such as MAKE-ATOM and MAKE-ARRAY-BLOCK (which should be MAKE-SYMBOL and MAKE-ARRAY) have problems. First of all, they need to be interrupt-protected, but also they can get screwed by a gc happening to their allocated but not filled in storage. When ALLOCATE-BLOCK is called it always initializes the storage to a cdr-next list of nils. MAKE-ATOM is all right if the gc never changes such a list to a non-cdr-next list when it copies it. Then what it has to do is store the SYMBOL-HEADER-Q then change its pointer from DTP-LIST to DTP-LIST-INTO-STRUCTURE without the possibility of any gc type operation happening between those steps. MAKE-ARRAY-BLOCK has to set up a multiword header structure (namely leader, header, and length word if the array is long) then change the type of its pointer. While the header structure is halfway made things are really inconsistent, this may require some additional mechanism to win. See above for mumbling about new microcoded primitives to help with this.

0019 EDT Sunday, 26 September 1976 DAM
revised 2/4/77 Revised again 8/8/77 to correspond to the actual code.
Revised for initial CADR implementation 7/15/78 (old CONS numbers in parens).
2/3/79 revised for GC changes and to delete obsolete CONS stuff.

Lisp machine paging.

The 24-bit VMA is divided into a 16-bit virtual page number and 8 bits of word address within the page. The virtual page number is looked up in the map to produce a 24-bit result: 2 access bits, 2 status bits, 6 meta bits (used to indicate various per-region attributes defined elsewhere), and a 14-bit physical page number. The last 512 physical pages (and the last 512 virtual pages) point at the Unibus (16 bits per word). The 512 physical and virtual pages before that are the I/O area of the Xbus (32 bits per word). The 4 virtual pages before that are mapped onto A-memory (by simulation in the page fault handler). This allows lisp-coded routines to access microcode variables which happen to be stored in A-memory rather than main memory with a minimum of fuss; indeed, they can even by lambda-bound. The remainder of the physical pages are memory (or a hole where memory could be inserted.)

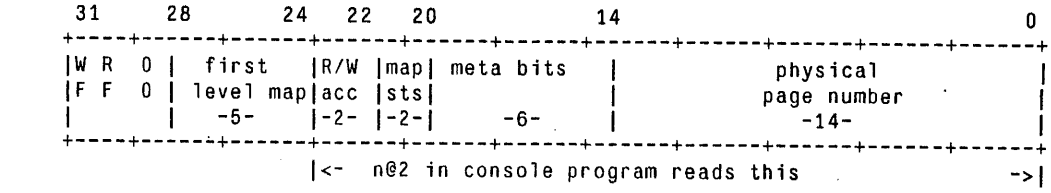
To avoid the need for a ridiculously huge mapping memory, or an associative memory, a two-level map is used. The second, or main, level consists of 32 blocks of 32 registers each. The first level is indexed by the high 11 bits of the virtual page number and specifies the block number in the second level. The remaining 5 bits of the virtual page number select a register within that block. The first 'n' locations in the second level map are permanently wired. 'n' is determined when the cold load is loaded. These pages have to be wired because they are used by critical microcode such as the page fault handler. Typically 'n' amounts to two blocks of map. These blocks are then permanently reserved to these pages so that the microcode doesn't have to worry about recursive page faults.

The last block is used for pages which aren't set up to a block of map. This is necessary because the first level map does not have a "valid bit;" instead, invalid entries are set up to point to the last block. All entries in the last block of the second level map must contain invalid page pointers.

There are some additional tables associated with paging: the Page Hash Table has an entry for each page that is "interesting", i.e. not just sitting placidly on the disk. However, the 1028 Unibus, Xbus, and A-memory pages don't have entries here and must be checked for explicitly at the appropriate points in the paging microcode. The Reverse First Level Map is an inversion of the first level map, which is used to control allocation of blocks of second level map. These will be described in more detail after the various fields in the second level map have been described.

The exact function of the six "meta" bits may be found in the %%REGION- definitions in QCOM, and in (non-existent) garbage collector documentation.

CADR on read from MEMORY-MAP-DATA



On CADR, bits 19 and 18 can be substituted into the result of the byte extraction part of the DISPATCH instruction. This feature is used by the transporter dispatch.

The values of the access bits are:

0	no access
1	no access
2	read only
3	read/write

These are looked at by the hardware.

The status bits are dispatched on by the page fault handler to find out how to handle the fault. Note that the 4's bit here is the same as the write-access bit. The possible page states are:

status	meaning	access	swapped-in
0	map miss	no	maybe
1	meta bits only	no	maybe
2	read only	R	yes
3	read/write first	R	yes
4	read/write	RW	yes
5	page might be in pdl buffer	no	yes
6	possible MAR trap	no	yes
7	not used		

Further discussion of page states. (Map status codes)

0 map miss
This can be either a first-leval-map-not-valid or a second-level-map-not-valid. These are distinguished by whether or not the contents of the first-level map is 37.

To handle a first-level-map-not-valid, a block of second level map must be allocated, filled with "second level map not valid" entries, presumably clobbering valid map entries that were previously there, and the first level map must be set up to point to it. Then drop into second-level-map-not-valid:

To handle second-level-map-not-valid:
First, check if this is a reference to a Unibus or Xbus page. If so, set up the map to read/write and the correct address (which the hardware will know means to access the Unibus or Xbus), and restart the reference. Otherwise, check to see if this is any other type of magic page, for instance one of four pages which correspond to the microcode's A-memory. If so, simulate the operation. Otherwise, consult the page hash table (see below) to get any known information about the page. If it is not found in the page hash table, then the page is stored on disk at the same address as its virtual address, and the other information about the page depends on what area and region it is contained in. Find a free page of main memory (see below), read in the referenced page from disk, and set up the page hash table appropriately, consulting one or more of the per-region arrays. (An inefficient mode of search could be used here since we are waiting for a disk transfer anyway.)
Now copy the information from the page hash table into the hardware page table, and restart the reference.
If the page hash table swap-status field contain "age trap", set it back to "normal." The swap out scheduler can set a page's swap-status to age trap and invalidate its second-level map entry if it wants to find out if that page is frequently being referenced.
Similarly, if the page is marked "flushable", change it back to normal. The "flushable" indicates that the swap-out scheduler thought the page wasn't being used and wanted it to be swapped out soon, but evidently it was mistaken.

1 meta-bits only
This code indicates that this map entry contains meta-bits information but does not contain page-location information. This type of map entry is created when a pointer to an object is used but the object itself is not referenced. The meta-bits in such a map entry are needed by the garbage collector. An attempt to access the storage associated with the object will be treated like a second-level-map-not-valid map miss.

2 read only
The only type of fault possible is an illegal attempt to write.

3 read/write first
If an attempt to write occurs, change the status in the second level map and in the page hash table (see below) to read/write, indicating the contents of the page has been modified, and restart the reference.

4 read/write
No fault should occur on this type of page.

5 page might be in pdl buffer
Certain areas which are used to contain pdls arrange to get the map set up this status for their pages (instead of 4, read/write). The microcode has to decide, on every reference, whether the page is in the pdl buffer or in main memory, and simulate the appropriate operation. It may be that only part (or none at all) of the page is in the pdl buffer.

Thus the page fault handler must test the virtual address to see if it
falls in the range which is really in the pdl buffer right now.
If not, temporarily turn on read/write access,
make the reference, and turn it off again.
Pages may be swapped out without regard for whether they are in the
pdl or not. (This works because the normal course of swapping out
invalidates the 2nd level map, of course. If the page is then referenced
as memory, it will be swapped in normally and its map status restored
from the REGION-BITS table, in the normal fashion.
This will then restore the page-might-be-in-pdl-buffer map status.)
Otherwise, the addressed word is in the pdl buffer.
Translate the virtual address to a pdl buffer address and make
the reference.

The page hash table is an array of fixnums kept in wired-down, always-mapped main memory, in the PAGE-TABLE-AREA. Each entry is two fixnums long. The number of entries should be 30% more than the number of pages of main memory; however, the current hashing algorithm requires that it be a power of two.

The hashing algorithm for the page hash table, available at the micro code entry %COMPUTE-PAGE-HASH, is to take VMA<23:14>, xor that with VMA<23:8>x4, and AND with the size of the page table area in Qs, minus 2. If probing the specified location reveals an entry whose virtual address doesn't match, and it isn't a free entry, add 2 to the index and wrap around. Note that the hash codes for consecutive pages differ by 4 to avoid concentration of hash entries in one part of the table. This hashing algorithm is likely to change. (Well, it hasn't in 2 years.)

The first fixnum of a page hash table entry looks as follows:

Q1<2:0> = swap status. See table below.
Q1<5:3> = not used. (zero) Reserved for swap-status expansion.
Q1<6> = 1 if valid entry present, 0 if hash table cell free.
Q1<7> = not used.
Q1<23:8> = VMA<23:8> the virtual page number of this entry
This can be -1 if this is a flushable page which was put in to represent available memory which hasn't yet been used. Note that there must always be a PHT entry for each physical page of memory, or the machine forgets that that physical page exists and stops using it.
Q1<32:24> = fixnum data type

The second fixnum of a page hash table entry looks as follows:

Q2<13:0> = M2<13:0> the physical page number of this entry.
Q2<19:14> = M2<19:14> the extra bits (meta bits).
Bits 19 and 18 can be fed through by the hardware for use in the transporter dispatch.
Q2<21:20> = M2<21:20> the map status. This is the map status desired in the second level map entry for this page, thus codes 0 and 1 will not appear here.
Q2<23:22> = M2<23:22>
Q2<32:24> = fixnum data type

swap status

0	invalid. Free hash table entries have this.
1	normal. An ordinary page is swapped in here.
2	flushable. Means that there is a page here, but probably no one is using it, so the memory can be used to swap a new page into. This page may first have to be written out if the map status indicates that it has been modified since last written (code=4). Also, when the machine is first loaded any free memory pages are set to this and have a garbage virtual address (-1).
3	not used. Formerly meant in pdl buffer.
4	age trap set. This page was in normal status, but is now being considered for swap-out. The second-level map may not be set up for this page. If someone references the page, the swap status should be set back to "normal".
5	wired down. The page swapping routines may not re-use the memory occupied by this page for some other page. This is used for the permanently-wired pages in low core, and for temporarily-wired pages involved in pdp10 I/O operations.
6	not used.
7	on paging device. The physical address is a paging device address rather than a main memory address. There is no paging device at the moment, so this code is reserved. The paging device might be e.g. a shift register memory.

Note that symbols with names beginning %%PHT and %PHT are defined
in QCOM for these fields and codes. Here they are:

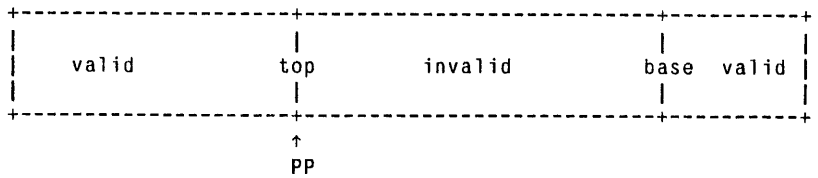
```
;PHT WORD 1
%%PHT1-VIRTUAL-PAGE-NUMBER 1020    ;ALIGNED SAME AS VMA
  %PHT-DUMMY-VIRTUAL-ADDRESS 377777 ;ALL ONES MEANS THIS IS DUMMY ENTRY
                                    ;WHICH JUST REMEMBERS A FREE CORE PAGE
%%PHT1-SWAP-STATUS-CODE 0003
  %PHT-SWAP-STATUS-NORMAL 1         ;ORDINARY PAGE
  %PHT-SWAP-STATUS-FLUSHABLE 2      ;SAFELY REUSABLE TO SWAP PAGES INTO
                                    ;MAY NEED TO BE WRITTEN TO DISK FIRST
  ;%PHT-SWAP-STATUS-PDL-BUFFER 3    ;ALL OR PARTIALLY IN PDL BUFFER (NO LONGER USED)
  %PHT-SWAP-STATUS-AGE-TRAP 4       ;LIKE NORMAL BUT TRYING TO MAKE FLUSHABLE
  %PHT-SWAP-STATUS-WIRED 5          ;NOT SWAPPABLE

%%PHT1-MODIFIED-BIT 0501            ;1 IF PAGE MODIFIED, BUT THE FACT NOT RECORDED
                                    ; IN THE MAP-STATUS, BECAUSE IT IS NOMINALLY READ-ONLY
                                    ; OR READ-WRITE-FIRST.

%%PHT1-VALID-BIT 0601               ;1 IF THIS HASH TABLE SLOT IS OCCUPIED.

;PHT WORD 2
%%PHT2-META-BITS 1606

%%PHT2-MAP-STATUS-CODE 2403
  %PHT-MAP-STATUS-MAP-NOT-VALID 0          ;LEVEL 1 OR 2 MAP NOT SET UP
  %PHT-MAP-STATUS-META-BITS-ONLY 1         ;HAS META BITS BUT NO PHYSICAL ADDRESS
  %PHT-MAP-STATUS-READ-ONLY 2              ;GARBAGE COLLECTOR CAN STILL WRITE IN IT
  %PHT-MAP-STATUS-READ-WRITE-FIRST 3       ;READ/WRITE BUT NOT MODIFIED
  %PHT-MAP-STATUS-READ-WRITE 4             ;READ/WRITE AND MODIFIED
  %PHT-MAP-STATUS-PDL-BUFFER 5             ;MAY RESIDE IN PDL BUFFER
  %PHT-MAP-STATUS-MAR 6                    ;MAR SET SOMWHERE ON THIS PAGE

%%PHT2-MAP-ACCESS-CODE 2602
%%PHT2-ACCESS-STATUS-AND-META-BITS 1612
%%PHT2-PHYSICAL-PAGE-NUMBER 0016
```

The reverse first level map.

For each block of 32 second-level map registers, there is an entry in the reverse first level map which gives the number of the first level map entry which points to this block, or else indicates that this block is unused. It contains a value which, if placed in the VMA, would address that first level map entry, or else it contains -1 to indicate that this block is not currently pointed-to. The reverse first-level map is stored in locations 440 through 477, which are in the SYSTEM-COMMUNICATION-AREA.

In the A scratchpad memory is a register called A-SECOND-LEVEL-MAP-REUSE-POINTER, which contains the number of the next block of second-level map to be reused, when a first-level-map-not-valid fault occurs. This pointer may not point at the magic last block, nor at one of the low blocks containing permanently-mapped wired pages. When a block is gobbled, the pointer is advanced to point to the following block. When any page fault occurs, if the pointer points at the block involved in the fault, the pointer should probably be advanced (currently it is not). The swap-out routines might also set the pointer to point at the block swapped-out from (however, they don't). This could provide a crude approximation to least-recently-used.

A-memory also contains the first virtual address which currently resides in the pdl buffer (in A-PDL-BUFFER-VIRTUAL-ADDRESS), and the pdl buffer index corresponding to that address (in A-PDL-BUFFER-HEAD). Note that the valid portion of the pdl buffer can wrap around. For instance:



Paging, Interrupts, and Processes

The lisp machine will support multiple processes. Passing of control from one process to another may happen either by explicit request (coroutining), or because of an interrupt (called a sequence-break). Processes are implemented by the mechanism of "stack groups". (See noplace.) Certain internal conditions will trigger a sequence break. External device interrupts will cause an interrupt, which is different. Interrupts are handled entirely in microcode and cannot touch pageable memory; these allow simple things to be done with fast response time.

It is desirable for paging to be at a "low level" in the system, i.e. not to depend on very much of the rest of the system, so that nearly everything in the system can depend on paging, using the full virtual address space freely without specially worrying about page faults.

Due to the complexity of the structure of a lisp process, it turns out to be highly desirable for sequence breaks to be at a "higher level" in the system than paging.

Therefore, the rule is made that PAGE FAULTS ARE UNINTERRUPTABLE. (In the sense of sequence breaks. Interrupts may happen during a page fault.) The page fault handler must always run completely in microcode, and may not ever allow sequence breaks to happen, even if it has to wait for a page to come in from the disk. Now a micro-coded routine can control where interrupts and sequence breaks happen while it is executing, simply by only checking for them at non-embarrassing places. It has to check for page faults after every memory reference, but it need not allow for the possibility of a sequence break after every memory reference.

This makes sequence break response rather slow, indeed it is conceivable that hundreds of milliseconds could elapse before a sequence break was serviced, in the unlikely worst case. However, the trade off is summarily declared to be worth it. Anything that has to be fast can be written in microcode as an interrupt, rather than in macrocode as a sequence break.

If the page fault handler is to be completely in microcode, it must be kept simple. Now the swap-in side of paging is always simple; just get the referenced page into core. It is only the swap-out side that involves complex scheduling. (In a one-user system.) This is not completely true, since some areas will be declared to be swapped in and out all at once, or at least in large chunks, but this is only a slight complexity. The swap-in side of paging has some connection to the swap-out side, since it has to be able to find a free memory block to read the page into, but this will be handled by having the swap-out side mark certain pages eligible for flushing. Most of the time the swap-in routines will use one of these, but if the supply has been exhausted it can simply choose any page at random, at some decrease in efficiency. If the supply of flushable pages is running low, a sequence break to the page swap-out scheduling process will be signalled, but this sequence break need not go off right away.
[It isn't really done this way now.]

So the operation of the swap-in routine, once it has discovered that indeed a page needs to be swapped in, is to start the disk seek, then find a main memory block to read into by groveling over the page hash table (almost always this will not involve doing a disk write), then start the disk read, then grovel over the area tables and gather up the information needed to go in the map (whether it will be read-only or read-write-first and what the three "meta" bits will be), then wait for the disk transfer to finish.

It seems best not to have any list of flushable pages, but rather to have the microcode scan through the page hash table at memory speed looking for pages with the appropriate swap-status field. It probably is a good idea to keep a count of the number of flushable pages, so that it is easy to tell when to awaken the swap-out scheduling process.

The function of the swap-out scheduling process is to somehow discover pages which are no longer needed in main memory, reset the second level map entry for the page to "second level map not set up" if it is still set up, write the page to disk if its map status is read/write, indicating that it has been modified since last read in from disk, and set the page's swap status in the page hash table to flushable, incrementing the count of flushable pages. We might choose to have the disk write take place asynchronously, with computation continuing unless a page fault is taken. This would require an extra swap status code in the page hash table, and checking for disk controller still busy with previous operation here and there. Similarly we might choose to do the reading of additional pages asynchronously when swapping in an area which is to be swapped in all at once, if they can't all be brought in with a single disk operation.

If the program should happen to refer to a page that was marked flushable before it is actually flushed, the map will be set up to point to it again and the flushable status in the page hash table will be turned off. Thus making a page flushable is not as drastic as actually committing to swap it out.

Note that the above hairy user-programmable swap-out-scheduler system is not currently used. It may never be used. What currently exists is a fairly simple-minded microcoded swap-out scheduler based on the "aging" scheme. Whenever the machine stops waiting for the disk, it runs the aging algorithm. The ager has a scan pointer, which is an index into the PHT, and a rate parameter, call it 'n'. For each disk operation, the scan pointer is advanced through the next 'n' PHT entries. Thus if the size of the PHT is 'm', each physical page of memory will be seen by the ager every 'm'/'n' page faults (actually a little more often since it ages during writes as well as during reads.) What the ager does is to look at the swap status of each PHT entry. If it is 'normal', it is changed to 'age trap'. If it is 'age trap', it is changed to 'flushable'. 'Flushable' pages tend to get swapped out and re-used when the 'findcore' circular scan through the PHT encounters them.

So, the net effect of the ager is that if a page is not referenced for 2*'m'/'n' page faults, it gets swapped out. 'n' should be adjusted so that 2*'m'/'n' is several times smaller than the size of physical memory.

The system currently does not attempt to do any overlapped writes, but that will probably be added in the future, in order to decrease the average time to read in a page.

The following special A-memory locations exist, and appear as Lisp variables. These allow the aging algorithm to be controlled and aid in metering performance. (Note that there should be percent signs in the names.)
[This table isn't necessarily up to date.]

A-FIRST-LEVEL-MAP-RELOADS	Number of times the first level map was reloaded. Equals the number of times a block of second level map was allocated.
A-SECOND-LEVEL-MAP-RELOADS	Number of times the second level map was reloaded. Equals the number of times information on a page was moved from the PHT to the map.
A-PDL-BUFFER-READ-FAULTS	Number of times a memory reference turned out to be a read from the pdl buffer.
A-PDL-BUFFER-WRITE-FAULTS	Number of times a memory reference turned out to be a write to the pdl buffer.

A-PDL-BUFFER-MEMORY-FAULTS	Number of times a memory reference trapped because it might have been to the pdl buffer, but it turned out to be to memory after all.
A-DISK-PAGE-READ-COUNT	Number of pages read in from disk.
A-DISK-PAGE-WRITE-COUNT	Number of pages written out to disk.
A-DISK-ERROR-COUNT	Number of recoverable disk errors. (Irrecoverable ones crash the machine. They also have never happened except when the disk was turned off.)
A-FRESH-PAGE-COUNT	Number of fresh pages created in core instead of being read in from disk.
A-AGING-RATE	Number of age steps to take per disk operation. Setting this to zero disables the ager.
A-PAGE-AGE-COUNT	Number of times the ager set an age trap.
A-PAGE-FLUSH-COUNT	Number of times the ager marked a page flushable.

The only "wired down" (non-swappable) storage in the system is that referenced by the microcoded page swap-in routine. This includes the page hash table and any area tables needed to get the read-only vs read-write status, the "meta" bits in the map, and whether to swap in only the referenced page or the whole area (or a large part of it) to optimize disk transfer rates. Also page 1 (the SYSTEM-COMMUNICATION-AREA) needs to be wired because certain locations in it are used by the console program and the page fault handler when writing into the READ-MEMORY-DATA register. Future hardware changes may eliminate the need for this. It is also possible for user programs to temporarily wire pages using the %CHANGE-PAGE-STATUS microcoded function. The only thing which uses this now is the temporary I/O system, which wires down buffer pages so that the pdp10 can get at them to do the I/O to its file system.

*** The remainder of this paper is wrong, and only here for the sake
    of saving it. This stuff has in fact been worked out, and should
    be documented some day. ***

[The remaining stuff about interrupts is mostly inoperative.
It will be revised later, and become a second paper about interrupts,
stack groups, scheduling, etc.]

The lowest level, microcoded functions in the system have direct
control over when interrupts can happen while they are running. Higher
level system functions, and user functions, need control over what
happens when they are interrupted, because when an interrupt occurs,
control of the machine passes to another process, which could do
anything. However, since such functions are written in Lisp, their
method of dealing with interrupts needs to be oriented to Lisp.
The simplest, most natural method, which will do the job in many cases,
but not all, is simply to defer interrupts entirely. A lambda-bindable
variable, %INTERRUPT-DEFER, will prevent any interrupt from occuring
if its value is non-NIL. The value of this variable will live in
A-memory, and will be pointed to by an invisible pointer in the value
cell.

Note that it is a loss to do anything complicated which can "trap
out" or run another stack group while interrupts are deferred.
This includes consing (might provoke a garbage collection),
referencing an invoke-type pointer, or getting a LISP error .
(transfers control to the error debugging process.)

Another useful variable is %INTERRUPT-PENDING, which might be T or NIL,
or might be a fixnum bit mask.

A more powerful, but hairier feature for interrupt control, will
be a variable %INTERRUPT-FINALIZE, which if non-NIL will be
funcalled (with no arguments) before the interrupt actually happens.
Most commonly this f.nction will simply do a THROW to a suitable
CATCH. THROW will be specially hacked to recognize that if the
machine is in progress of starting an interrupt, after unwinding
the environment back to the catch it should try again to take
the interrupt. %INTERRUPT-FINALIZE had better have been lambda-bound
inside the CATCH or an infinite loop will occur. During the execution
of the interrupt finalize routine, additional interrupts will
be %INTERRUPT-DEFERed. Consider the following code for INTERN,
which doesn't completely work but gives you the idea:

```
(DEFUN Q-INTERN (SYM &OPTIONAL (OBARRAY OBARRAY))
 (PROG (HSH BUCKETN TEM)
   (SETQ HSH (SXHASH SYM))
 A (CATCH
    ((LAMBDA (%INTERRUPT-FINALIZE BUCKET)
                    ;CAN'T INTERRUPT WHILE HACKING BUCKET
        (SETQ BUCKET (AR-1 OBARRAY (SETQ BUCKETN (\ HSH SIZE-OF-OB-TBL))))
        (COND ((SETQ TEM (MEMBER SYM BUCKET))
               (RETURN (CAR TEM)))
              (T (AS-1 (CONS SYM BUCKET) OBARRAY BUCKETN)
                 (RETURN SYM))))
     (FUNCTION (LAMBDA () (THROW NIL INTERN-CATCH)))
     NIL)
   INTERN-CATCH)
  (GO A)        ;INTERRUPTED, TRY IT ALL AGAIN
))
```

Note that the simpler interrupt-defer method could not be used because
it must CONS while it has the bucket "open", bringing in the possibility
of a garbage collection.

An alternative would be to make a cons first, then RPLACA and RPLACD it if
it was needed, however this would lose since nearly always the cons will
not be required since the symbol will always be on the obarray.

The reason this definition of INTERN doesn't work as it stands, besides not
having every feature that you might want it to have, is that where it says

"MEMBER" above it doesn't really mean the regular MEMBER function.

An even hairier method of dealing with interrupts would be PCLSRing as in ITS, which essentially is the same as the above except that the THROW is delayed until you are sure that you actually need it. Of course being sure requires a more elaborate system of explicit locks, etc.

Exact mechanics which remain to be worked out:

What is the exact structure of a stack group? Where is everything saved when an interrupt occurs? Just which registers get saved?

How are pending deferred interrupts remembered? What causes them to go off when interrupts are undeferred? How does the hardware for external interrupts work? (Not at all right now.)

How do the garbage-collector, swap-out-scheduler, and error-handler processes get signalled? In general, how does a process arrange to wake up on a certain condition? What happens when a process's wake up condition happens while the process is running? (Error in the error handler is bad, memory tight in the swap-out scheduler is all right and in fact sort of not very surprising; the swap-out scheduler certainly will take page faults.)

What does the initial cold load look like? Perhaps the most reasonable scheme is for "cold loading" to mean loading a complete lisp machine system onto the disk, including suitable contents for the page hash table area, and "booting" to mean bringing the first N blocks of the disk into core. Some of those blocks might then be paged out again, but the wired-down storage would never be written to disk, and the corresponding disk blocks would contain the initial image for the wired-down stuff, which would be read into core only at "boot" time.

Then cold loading is a console-computer function, while booting will eventually be accomplished by read-only micro code when a magic button is pushed. Extra complication- a machine can cold load itself if its disk is split into two halves, so that it runs off of one half while cold loading the other, then you toggle the switch which says which half to boot and run off of.
Extra complication- booting must include loading the microcode.
At the moment this is a console-computer function and totally separate from everything else, but eventually it should be part of booting. The right way to do this is to have an initial-micro-load area, which is read in from disk when you boot, then stuffed into the various memories, but then is free to be paged out. This is an extension of / replacement for the present scratch-pad-init area.