A.I. Memo
528

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

May 1979

CADR

Thomas F. Knight, Jr.
David 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-00643.

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 the 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 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 my 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 the particular order code of the LISP machine, bit which is general enough to at interpreting 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 accommodated by the 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 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 the results are stored; and finally the two instructions for controlling the ALU and the byte extractor.

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 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 the 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 specified 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 tow 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=1 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 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 the right mask memory) plus
                (the length minus 1 field), mod 32.

octal   LEFT MASK MEMORY contents             RIGHT MASK MEMORY contents
index
   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    00000000000111111111111111111111     11111111111100000000000000000000
  25    00000000001111111111111111111111     11111111111000000000000000000000
  26    00000000011111111111111111111111     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 specification cause it to hang over the left-hand edge of a word), then the masker does *not* truncate the byte at the left edge. Instead, it produces a zero mask, selecting no byte at all; thus, the output of the byte operation equals the A source.

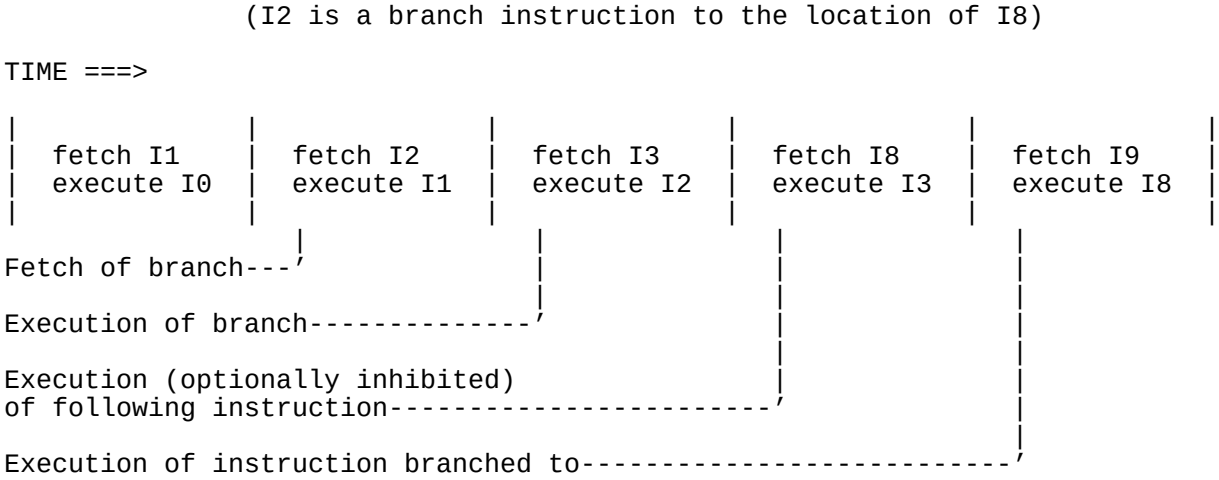
The reason for this is that an overflow occurs in calculating the index into the left mask memory, and so the final mask is zero. For example, if MR=1, rotation count=20 (octal), and length minus 1=27 (octal), then the right mask index is 20 and the left mask index is 477 (mod 32). This results in a 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 lookahead, *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 the 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 is specified by the 3-bit transfer type 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 of the SPC stack.
P=1 R=1 FALL THROUGH or I-MEM WRITE
    In a DISPATCH instruction, do not dispatch.
    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 not 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 *not* 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 sub-field 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 net, 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 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 ...

[Ed note: PAGE 18 is missing!]

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 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 it which it operates. It causes the same operations as the CALL transfer type, resulting in 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 extra 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 ablitiy to use program determined addresses of registers; for example, it can be used to index into the A scratchpad memory.

Function 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 address 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* insttruction. 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 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 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. **[Ed note: Figure:** *CADR cycle timing* **omitted]**

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 cocatenation of the output from the First Level Map and bit <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 correspoding 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 is 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 Frist 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 a 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 write 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 tranfer 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 respones 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 occured. Single-bit errors are corrected automatically

be 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 instructions 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 curent 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 variaous 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 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 has 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 an 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, whis 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 ste
(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 pos
                    (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 multipl
          (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 sutracts 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 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 number 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 divid
          (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
DIV0      ((M-AC) SETCM M-AC)                              ;1's complement high divi
          (CALL DIV2)                                      ;Now, call positive-divid
          (POPJ-AFTER-NEXT (M-1) SUB M-ZERO A-1)           ;Make the remainder negat
         ((M-AC) SUB M-ZERO A-AC)                          ;and change the sign of t

; 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
          ((M-1) DIVIDE-REMAINDER-CORRECTION-STEP M-1 A-2) ;M-1 gets remainder
          ((M-AC) Q-R)                                     ;Extract quotient from Q-
          (POPJ-AFTER-NEXT                                 ;Return after next, but i
           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 the 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 17400000-17777777, reference the Unibus. Each 32-bit word has a 16-bit Unibus word in bits 0-15, and zero in bit 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 17000000-17377777, are reserved for Xbus I/O devices. Locations 0-16777777 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 puproses. 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.

1Bit 17 of the debugee-Unibus address.

2Resets the debuggee’s Unibus and bus interface. Write a 1 here then write a 0.

4Timeout 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 debuggees’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.

4CADR 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.

10Unibus NXM Error. Set when a Unibus cycle times out for lack of response.

20CADR Parity Error. Set when data received from the processor has bad parity. Indicates trouble in the communication between the processor and the bus interface.

40Unibus 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 can interrupt the CADR machine. The following Unibus locations control interrupts and the Unibus arbitrator:

766040 Reading this location returns interrupt status, as follows:

1Disable 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.

2Local 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.

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

2000Enable 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.

4000Interrupt 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.

30000Bits 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.

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

100000Unibus 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.

766041 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 hte 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 positiely 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 28ma 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 50ns. 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 an 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.
- -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 syrchronous 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 one 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 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.