IMPLEMENTATION OF A LIST PROCESSING MACHINE

by

Thomas F. Knight, Jr.

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JANUARY, 1979

Signature of Author..
Department of Electrical Engineering and Computer Science,
January, 1979

Certified by ....
Thesis Supervisor

Accepted by..
Chairman, Departmental Committee on Graduate Students

Archives
MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

MAY 25 1979

LIBRARIES

IMPLEMENTATION OF A LIST PROCESSING MACHINE

by

Thomas F. Knight, Jr.

Submitted to the Department of Electrical Engineering and Computer Science in January, 1979 in partial fulfillment for the Degree of Master of Science

ABSTRACT

This thesis outlines the hardware support developed over the past several years for the M.I.T. Artificial Intelligence Laboratory's Lisp Machine project. The Lisp Machine is a personal computer system designed for the efficient execution of the Lisp programming language in a highly interactive user environment. Each user is provided with a moderately high performance central processor, a high resolution bit-mapped display, a large fast access swapping disk, and interconnection to a high speed local network.

The software system consists of an operating system, compiler, interpreter, and editor written entirely in two levels, the Lisp programming language, and the microcode of the processor.

The processor microcode instruction set is designed for flexibility in emulation of complex instruction sets, and for ease of programming. The main subject of the thesis is the structure of this processing engine, and comparison its performance with other similar computers.

Requirements for emulating complex instruction sets are discussed, and the performance of this processor at the task of emulating the Nova 1200 is compared with the performance of a second computer designed with similar intent.

An appendix contains complete details of the design with an english commentary.

Keywords: Lisp, personal computers, microprogramming, emulation, symbolic computation, high level languages

Name and Title of Thesis Supervisor: Marvin L. Minsky,
Donner Professor of Science

Contents

I. Introduction.. 4

II. Network Architecture.................................... 7

III. The Personal Machine................................... 11

IV. The Software Environment................................ 14

V. The Macro Architecture................................... 15

VI. The Microcode Environment............................... 25

VII. Evaluation of Some Architectural Features.............. 41

VIII. An Architectural Comparison........................... 43

IX. Comparisons with Previous Work.......................... 45

X. Summary.. 46

Acknowledgements.. 48

Bibliography.. 49

Appendix I. Nova Simulation CADR Microcode.................. 51

Appendix II. Nova Simulation 11/40E Microcode............... 61

Appendix III. Comparative Nova Emulation Instruction Trace.... 68

Appendix IV. Print Set and Print Commentary................. 71

PRINT ON THE FOLLOWING PAGES IS PARTIALLY ILLEGIBLE

I. Introduction

This report concerns the development of a new computer system designed to provide a high performance and economical implementation of the programming language Lisp.

Features which distinguish Lisp from the majority of other programming languages include its built-in dynamic storage allocation, its ability to symbolically treat its programs as data, and its ability to assist users in interactive development of complex programs through the running, debugging, and incremental refinement of partially specified programs. Lisp is also widely used as a basis for development of more complex problem oriented languages, such as Planner, Conniver, and ACT1.

The Lisp language [McCarthy 1958, 1978] enjoys wide use in the artificial intelligence research community and is rapidly gaining adherents outside this group. The language has excellent support in several dialects on the Digital Equipment Corporation PDP-10 series computer [Moon, 1975, Teitelman et al. 1978] and less enthusiastic support on many other machines such as the IBM 370. Our experience at M.I.T. has largely been with the MacLisp PDP-10 dialect, originally written here in 1965 for the PDP-6, an earlier computer which is instruction set compatible with the PDP-10 series.

Over the years, dramatic changes have taken place in the MacLisp implementation, sufficient that one might well say that there have been at least three different major generations of implementaion, except for the continuity of user support.

At a certain point, however, modification and reimplementation of a language on a given machine can no longer efficiently compensate for basic problems in the architecture of the computer system. We believe this is now the case on the PDP-10 and similar time-shared computer systems.

The main difficulties are:

1. inadequate virtual address space for large user programs

2. inadequate computing power for development of intelligent programming tools

3. inefficient information coding of compiled instructions

The address space problem is basic to the PDP-10 architecture, which efficiently stores and manipulates address sized quantities with half word instructions, packing two 18 bit pointers per 36 bit word. Lisp nodes are stored in this way, saving space and performing simple pointer manipulations in a single compiled PDP-10 halfword instruction. Unfortunately, this tidy scheme is not easy to expand to a larger address space, even if extended addressing hardware were provided in future PDP-10 processor designs. Expansion of the virtual address space in current PDP-10 Lisp implementations requires use of two words per Lisp node and drastic changes to both the interpreter and compiler.

The issue of computing power is much more difficult to describe precisely, but is just as real. The argument is essentially that the next generation of editor / debugger / programming assistant should be able to spend <u>very significant</u> amounts of time supporting user interactions. In the older systems, editing a function just changes a few characters in a file. In the new systems we are evolving, when you edit a function, the system should <u>know</u> which other functions call this one -- should warn you of possible interaction problems -- and should allow you to easily find and fix these problems. Winograd presents a sort of prospectus for such a system in his Reactive

PRINT ON THE FOLLOWING PAGES IS PARTIALLY ILLEGIBLE

Engine Paper [Winograd, 1975].

In a present day time shared environment, such a system is unlikely to develop. Such time sharing systems are structured at a fundamental level around the ideà that most users do nothing most of the time. In a reactive computing environment such as we envision this assumption is simply not true. A time shared computer could not support more than a few such users.

The third defect in the PDP-10 architecture we mentioned was the inefficient coding of instructions. The PDP-10 Lisp compiler produces a very restricted subset of those instructions available on the machine. Function entries and exits are highly stereotyped, and the compiler frequently produces multiple PDP-10 instructions for operations which are logically quite simple. These difficulties are not the result of bad compiler design; on the contrary, the MacLisp PDP-10 compiler often produces better numeric code than the PDP-10 fortran compiler. The difficulty is that there are substantial representational gains to be had by representing the common, complex operations by simple instructions. For this to be effective, however, the generality of a traditional machine architecture must be abandoned in favor of an architecture which takes a strong position on such issues as how data is represented, and how it is referenced. A good example of such a strong commitment to a higher level architecture and representation is the B6700 machine, with its commitment to the Algol environment.

Deutsch [Deutsch, 1973] and Greenblatt [Greenblatt, 1974] have both proposed much more bit efficient instruction sets for the object code produced by the Lisp compiler. By shrinking the size of compiled programs, the performance of the system is improved in several ways:

1. We reduce the size of primary memory needed to hold a given working set.

2. We reduce the required bandwidth and time for secondary memory access when it is required.

3. Our address space is more efficiently utilized.

The price paid for these improvements is an increase in the complexity of the order code, an issue we will discuss in the section on instruction set design. With modern microcoded processor designs, this increase in complexity adds very little to the total system cost.

Realizing some of these problems, Richard Greenblatt, Jack Holloway and I started work in the summer of 1974 on an ambitious long-term project to implement a new computer system in support of the next generation of artificial intelligence research. This thesis discusses the hardware architecture developed in support of this effort. The software aspects of the system are discussed in greater detail in the Lisp Machine Manual [Weinreb & Moon 1978].

First, we will consider the network architecture of a computer system suitable to support a large community of researchers. Each user in the system we envision has exclusive access to a medium size personal computer system.

The hardware resources comprising this personal computing resource is our next topic.

We then will briefly detour into what is really a software issue, the design of a bit efficient and easily compilable instruction set for the Lisp programming language. Our goal here is to provide a motivation for the features needed in the description of the microcode architecture.

This micro architecture is our next subject, followed by an evaluation of the microprocessor features from two very different standpoints -- its capability to emulate a small commercial minicomputer (the Nova), and through measurements of its performance while executing the Lisp Machine microcode.

We will conclude with a discussion of design techniques for the hardware, and a summary of what we have learned and what we believe should be done next.

The machine has been built, debugged, and is in "limited production" at M.I.T., where four machines are currently operational, and construction of six more is in progress. (December, 1978) Complete details of the hardware are available in the hardware manual CADR [Knight, Moon, & Steele 1978].

II. Network Architecture

A crucial early decision in the design of the Lisp Machine system was that a user of the system, at any time, would be assigned exclusive use of one of a pool of available Lisp Machine processors. Assigning a single user to a single machine improves the perceived system performance. allows some simplification of the system software, and places important constraints on the system design.

At a technical level, almost all of these constraints concern the interconnection of the processors: among themselves, with the users, and with other computer systems -- that is to say, the network architecture of the system. Our primary prototype for this level of the system design was the Xerox Palo Alto Research Center single user ALTO computer system [Alto] and its interconnecting network, the Ethernet [Metcalfe & Boggs 1975].

In the Xerox system, each user has an ALTO computer, raster display, keyboard, and disk drive physically located in his office. Users maintain files on removable disk cartridges. The computers are interconnected via a 3 Megabaud cable network. A rough sketch of the Xerox network is shown in figure 1.

Several aspects of the Xerox system appear less than ideal to us. First, despite great technological progress, a medium scale personal computer is still not well suited to an office environment. Air conditioning, power distribution, static electricity, and perhaps worst, acoustical noise problems make a computer an unwelcome office mate. Second, although one might feel more security and privacy associated with having one's own personal, removable files, we feel that this is far outweighed by the many advantages of a centralized file system. Notable among these advantages are file sharing and communication among users, and systematic and thorough file system backup.

The network architecture we envision for the Lisp Machine system is shown in figure 2. Its major components are:

1. several Lisp machine processors

2. a central file system

3. an interconnecting high speed network

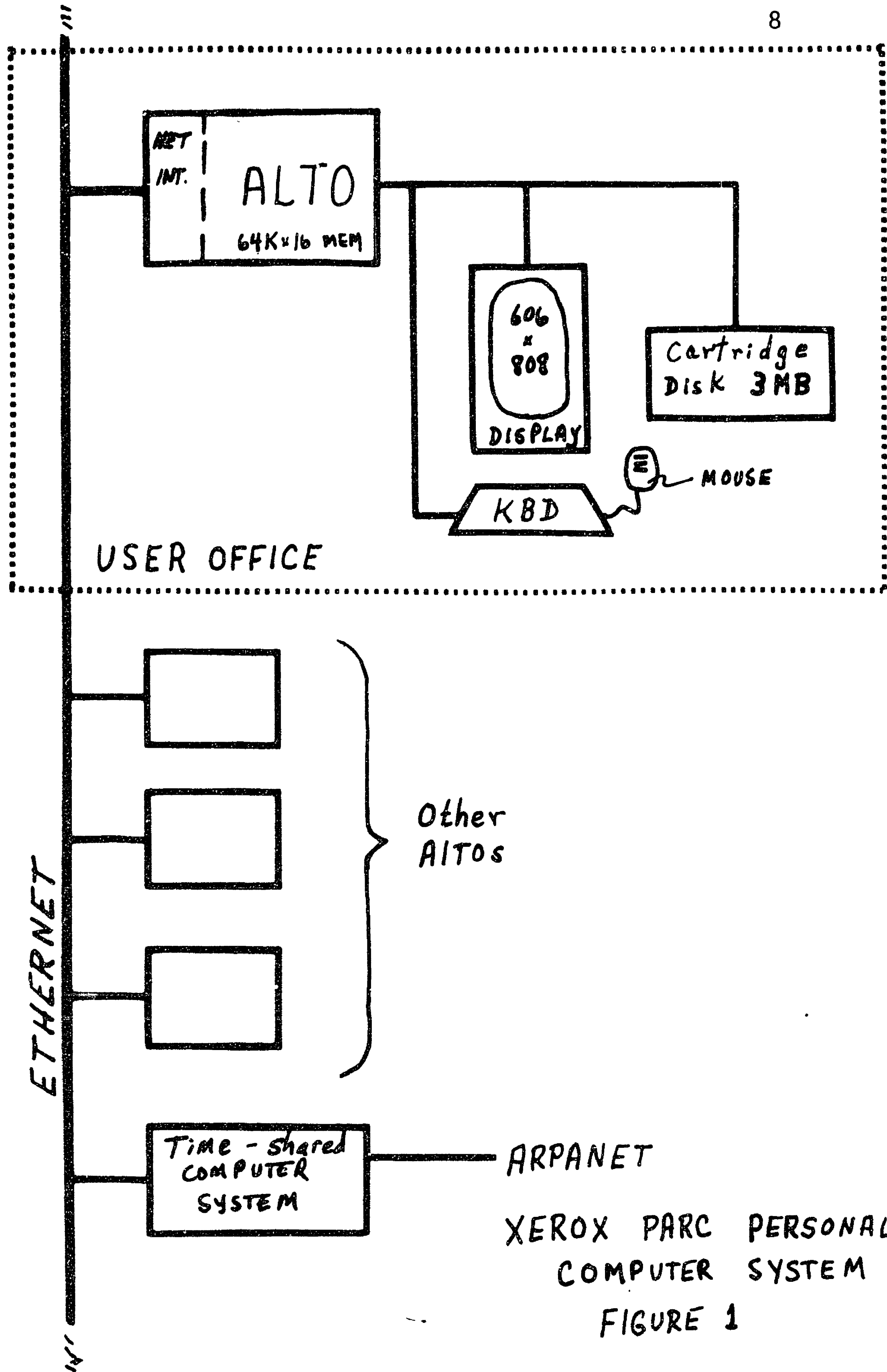
4. a video switch to connect remote display terminals

We believe this configuration retains the many advantages of the Xerox personal computer system, while addressing what we perceive as its difficulties.

Users of the system still occupy their natural work environment, the office. Sharing the office now, however, is simply a high resolution television monitor, a keyboard, and a graphical input device. The remainder of the computing equipment is remotely located.

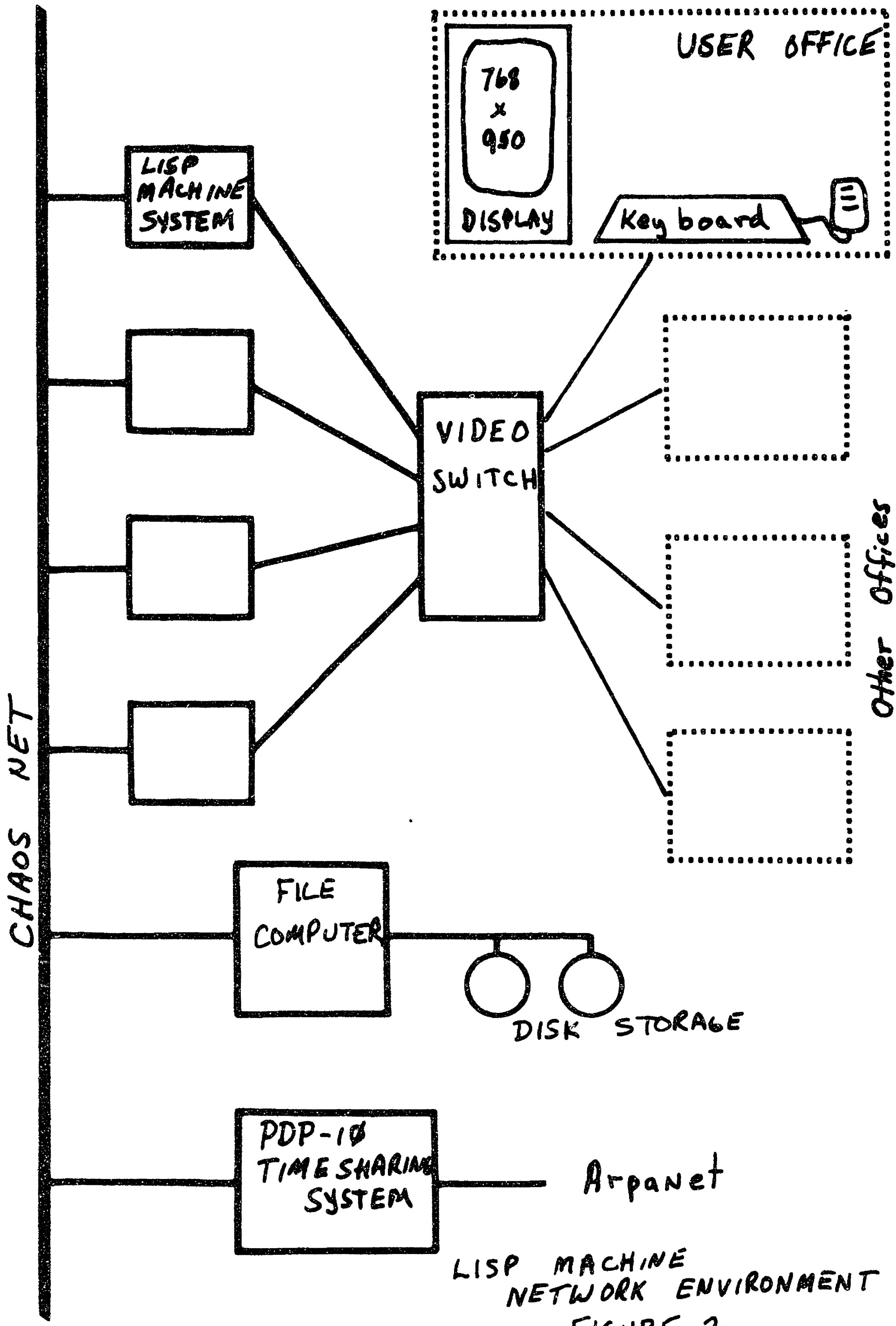
When the user initially requests access, he is assigned exclusive use of one of the pool of Lisp Machine processors. His terminal and keyboard are connected via the video crossbar switch to the assigned processor, and he interacts with that processor on a dedicated use basis.

His files are permanently resident in the so-called central file system (CFS), a machine dedicated to the reliable maintenance of a file system very similar to that found in a good time-shared operating system. Reliability and protection issues in the design of the CFS are greatly simplified by the fact that user written programs never execute on the CFS machine.



XEROX PARC PERSONAL COMPUTER SYSTEM

FIGURE 1



LISP MACHINE
NETWORK ENVIRONMENT
FIGURE 2

The high speed local network allows the user, and other users, to access and update his files in the CFS.

The CFS also provides a logical point of attachment for expensive or unique peripherals. A high quality document printer, for example, fits in naturally here. The CFS can also serve as a gateway computer for access to remote lower bandwidth networks, such as the ARPANET. In this role it will provide remote computers access to the local file system, but we do not believe that the bandwith of the ARPANET is sufficient to obtain the full benefits of effectively interacting with a personal computer over large distances.

There is no particular reason to restrict this network architecture to a single CFS processor, and, as the technical problems associated with reliably maintaining a distributed, possibly redundant file system are solved, we expect to take advantage of the solutions.

In effect, the network architecture we have planned tries to combine the best features of both a personal computer system and a sophisticated time-sharing operating system. The dedicated processor provides the attentive computing resources that we feel are needed to solve complex problems; the central file system provides interaction between users and economical methods for interconnecting unique peripheral devices.

III. The Personal Computer System

Each personal computer assigned to a user is moderately self-contained, having an independent computing engine (the CADR processor), main memory, swapping device, console peripherals and network interface. A typical arrangement of one such personal computer is shown in figure 3.

The processor, which will be discussed in much greater detail later, is a 32 bit wide microprogrammed computer with a cycle time of approximately 180 nanoseconds. It is implemented in Schottky TTL logic and occupies approximately 1000 integrated circuit packages, including the 16K word writable microprogram memory.

Main memory size in a typical personal computer configuration will be about 128K words, an amount we believe will properly balance the computing power and swapping capabilities of the system. In the prototype machine, this main memory was implemented with a core memory system built by Control Data. Production versions of the machine use main memory implemented with 16K dynamic RAMs. The RAM based memory includes single bit error correct, and double bit error detect circuitry for increased reliability. The memory is implemented with a basic card size of 64K words by 44 bits. In the present application of 32 bit words, this memory is depopulated to 39 bits per card, providing the necessary bits for the error detection and correction circuitry. The backpanel in the production machine accepts eight such cards, allowing expansion to 512K words without the addition of another backpanel.

The standard input/output interface in the system is based on the PDP-11 Unibus. By adopting the interface standards of an existing widely used processor, we can exploit existing devices for which PDP-11 interfaces may be purchased, and can apply devices of our own design both to Lisp Machine processors and to the many existing PDP-11 processors in our laboratory.

Since the PDP-11 Unibus is a 16 rather than a 32 bit bus, there are some problems in information transfer across the interface. A detailed discussion of these issues is deferred until the description of the main memory interface in the processor. For now it is sufficient to note that the Unibus interface compatibility is well enough implemented to allow swapping of main memory to occur via a standard PDP-11 compatible disk controller.

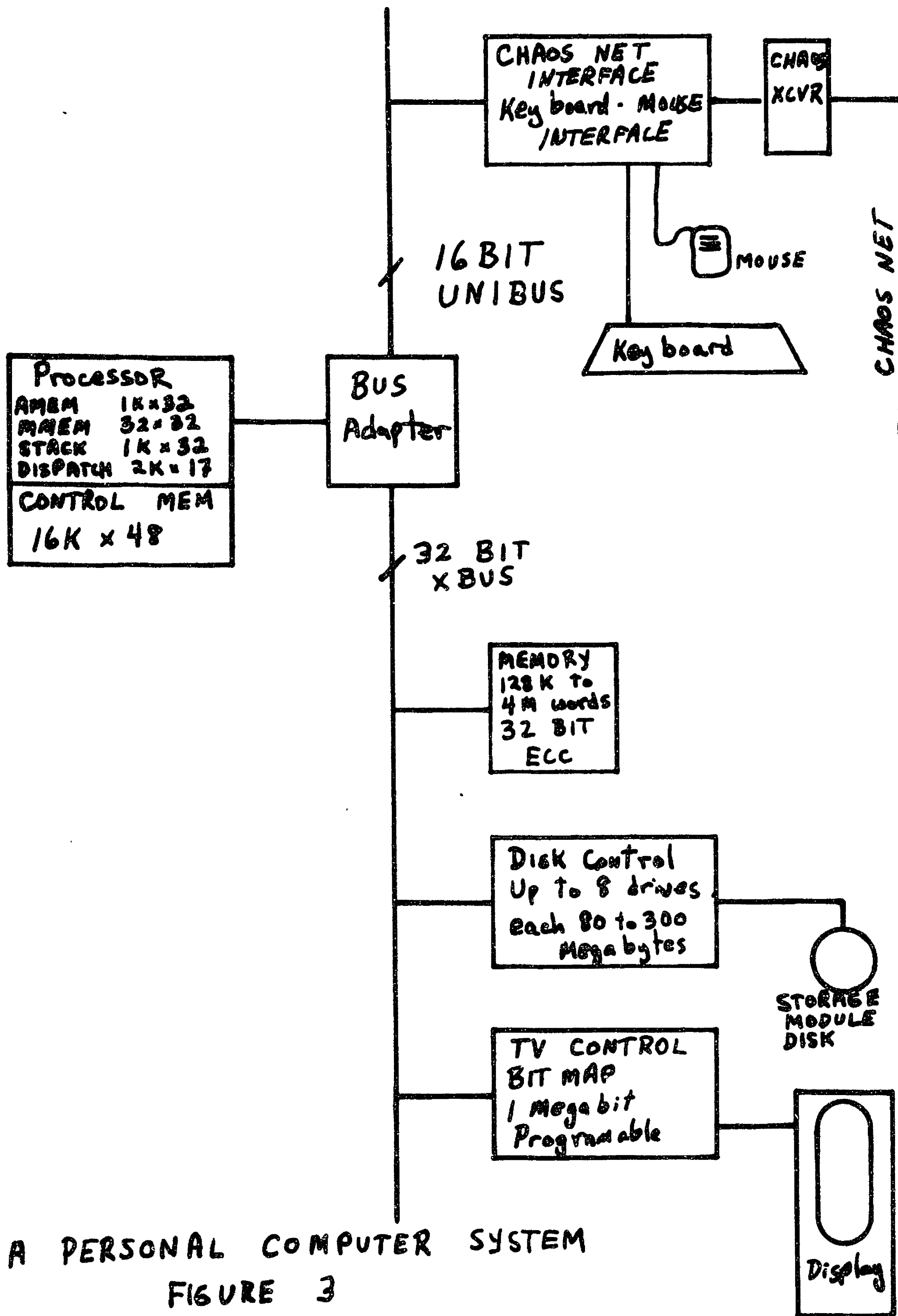
The swapping device exerts a crucial influence on the ultimate performance of the system. Since the primary memory is small compared to the virtual memory size, some portion of the memory references access locations which are disk resident at that moment. As the percentage of disk references rises, the access time for those references becomes a significant factor in the speed of the machine.

In a single user system, the full attention and bandwidth of this swapping device is available to the individual user -- a resource which must be shared (or fought over) in a time shared environment.

The swapping device used in our prototype system was an 80 megabyte Control Data storage module disk drive, with an average access time of 30 ms. The production machine uses an almost identical disk drive manufactured by Calcomp. Both of these drives feature fast head motion and small average rotational latency (3600 rpm).

We hope eventually to replace these rotating magnetic storage devices with some inexpensive non-mechanical storage, such as magnetic bubbles or charge coupled devices. At the moment, neither of these technologies look promising as cost-effective replacements, however. One promising development is the recent interest in non-removable media drives at very low cost. Such disks should be very effective in this application, since the swapping of disk packs is not necesary in a high bandwidth network environment.

The CDC disk drive was interfaced in the prototype system with a PDP-11 disk



A PERSONAL COMPUTER SYSTEM
FIGURE 3

controller built by Xylogics. Due to the high cost of this component of the prototype system, a customized replacement has been designed in-house by Dave Moon as part of the production version of the machine. The redesigned interface provides error correction of disk transfers, and interfaces over the 32 bit XBUS rather than the less efficient 16 bit Unibus.

The local network for the system is a modified form of the Xerox Ethernet [Metcalfe & Boggs 1975]. It operates at a bit rate of eight megabaud maximum data rate, delivering variable length packets (maximum 4K bits) between pairs of machines. The network interface for each machine contains two packet length buffers, one for receiving messages, and one for transmitting messages. Message transmission is slotted in time, unlike the Xerox Ethernet, thus avoiding collisions between message packets under normal circumstances. As with the Ethernet system, the entire transmission process makes only a best effort attempt at delivery of a message, relying on higher level software protocols for the required reliability and robustness needed in a multi-computer environment. The hardware provides a 16 bit cyclic redundancy error check for detection of garbled message packets.

The graphics display on the processor consists of a bit-per-point television compatible raster display. The resolution of this display is 768 bits horizontally and 900 bits vertically. The monitor used is a CPT corporation model H display, with a 60 kHz. horizontal line rate. It runs non-interlaced at a vertical frequency of 67 Hz. The high refresh rate allows the use of a pleasing white phosphor, rather than the long persistence green phosphors required in interlaced systems.

The video generation is done with a 64 by 16K two-port dedicated memory. One port is used for refresh of the display, while the other is driven from the 32 bit processor XBUS. From the standpoint of the processor, this memory looks exactly like any other memory in the system. Character drawing and graphics are done by writing the appropriate bit patterns into this memory. The line and character drawing instructions are microcode resident for speed.

Other forms of video can be produced from this display card. Of particular use is the generation of 4 bits per pixel at standard RS-170 video rates, which is useful in driving grey scale or color displays.

We intend to investigate the possible uses for multiple CRT displays. The opportunity to use more than one display to, for example, debug, edit, and run a complex program simultaneously seems enticing. In the computer aided design area, for example, it may be desirable to display a portion of a complex design in color on one monitor, while maintaining normal user interaction on the high resolution black and white display.

Graphical input is another area where research in human interface is likely to be productive. We have already investigated both traditional tablet techniques and the SRI/Xerox "mouse" technology for input, using a Summagraphics tablet and a mouse provided courtesy of Xerox PARC. An improved mouse has been consructed here by John Purbrick, which will become our standard graphical input device. The mouse consists of a spherical ball idler, driving two orthogonal optical shaft encoder. The pair of quadrature signals switch at the rate of 100 steps per inch.

The dedicated nature of the processor makes quite practical another form of computer output, namely high quality audio. Although we have not yet experimented with this interaction medium, the address space and response time of the Lisp Machine system appear to be adequate to provide very high quality real time audio response. The usefulness of this feature in system design is, at least to us, unknown. A starting point might be in providing audio feedback to keyboard strokes, or spoken rather than displayed error comments from executing programs.

IV. The Software Environment

Lisp, while not unique in this respect, has a long history of using multiple data representations for the same program. Viewed initially as a strictly interpretive system, the desire for faster performance rapidly inspired development of increasingly sophisticated compilers. Although the ability to run interpreted code remains an important language feature, today almost all serious work is done with compiled Lisp programs.

The Lisp Machine system offers two improvements over existing Lisp implementations in the area of program representation. First, we had the opportunity to define for ourselves a computer architecture well suited to the Lisp environment. The goals of this architecture are high bit efficiency in the representation of compiled instructions, and ease of the compiler construction. Implementation of these requirements led to an instruction set that, although conceptually simple, is rather complex in implementation. This led naturally to the choice of a microcoded architecture for the processing engine.

Second, the existence of the microprogrammable processor provided the opportunity to execute user functions in another way: by compiling the functions into the microcode of the machine.

We thus provide the Lisp user with three levels of program execution, each optimized for a different task. Interpreted execution provides the ability for programs to construct, modify, and analyze other programs, an ability almost unique to Lisp and likely to become more important with the development of more sophisticated program analyzing software, such as the work of Shrobe, and Rich [Rich & Shrobe, 1976]. The traditional role of interpreted execution as a simple debugging technique is largely supplanted in the Lisp Machine environment by the fast compiler and excellent compiled code run time error checking.

A second execution mode found in the Lisp Machine system is a modified form of traditional compiled code. Here, the user function is transformed into a bit-efficient and more quickly interpretable form we call macro-code. Emphasis in this process is on the bit efficiency of the resulting instructions, since it is anticipated that the vast majority of system software will be executed in this form. The high bit efficiency of this format reduces program size, resulting in smaller working sets, a smaller requirement for main memory, and less swapping bandwidth from secondary storage. Unlike many other compiled Lisp implementations, however, this execution mode performs many of the error checks found in the usual interpreted execution.

The third execution mode results from executing user functions as microcoded processor subroutines. In this mode, emphasis is on fast execution, rather than bit efficiency. The micro-compiled functions occupy a limited and expensive resource, the processor micro-code storage. Providing this quick access to the basic hardware of the machine is a goal which argued strongly for a single user machine -- it is very hard to protect users from one another in an environment allowing alterable microcode, not to mention the difficulties of allocating the resource effectively.

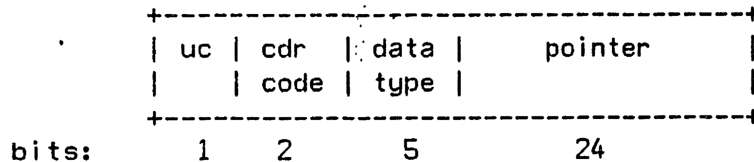
Next, we will examine in some detail the format of the macro instruction set, and the format of the data upon which it operates. The goal is not a complete understanding of the software basis of the Lisp Machine system, which is far beyond the scope of this report, but rather an appreciation for the type and complexity of operations which the microprocessor must support.

Full details of the software environment and data formats are available in the Lisp Machine Manual [Weinreb & Moon 1978].

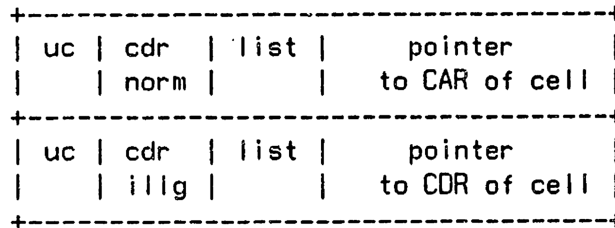
V. The Macro Architecture

Typed Data

The macro instruction set implements high level functions operating primarily on typed data of a format shown below:



Each 32 bit word is capable of holding one 24 bit pointer to any location in the virtual address space of the machine. The remainder of the word is used for a two bit CDR-CODE, a five bit DATATYPE (see below) and a one bit field allocated for users to specially mark individual storage nodes (the UC bit). A full Lisp node containing two pointers thus normally requires two sequentially located words. The first holds a pointer to the CAR of the Lisp node, and the second, a pointer to the CDR.



CDR Codes

By means of the two bit CDR CODE field in the word, however, this storage requirement can normally be halved. The CDR CODE allows a word to specify that the next sequential location is not a pointer to CDR of the node, but rather that it is the CDR of the node. A third state of the CDR CODE field specifies that the CDR of this node is NIL, so that the end of a list may be represented without an additional word. Thus, the two bit CDR CODE field is decoded into three useful states:

CDR-NORMAL	(the next word is the second half of a full node)
CDR-NEXT	(the next word is the CDR of this node)
CDR-NIL	(the CDR of this node is nil)

The combination of these codes allows representation of single level lists, such as (A B C D) as four sequential words, rather than as four two word full nodes.

The two bit CDR CODE field may be thought of as a very small address for the CDR of a cell, capable of specifying only the next word, NIL, or an escape specifying that the real address is contained in the next word. Clark and Green [Clark & Green, 1977] have collected statistics on the utility of short addresses in the Lisp environment. Some of the techniques presented in their work provide further savings at the expense of greater complexity in the use of

the addresses, but we believe the two bit field strikes the proper balance between complexity of implementation and bit efficiency of the resulting data structures.

The Datatype Field

The Lisp Machine references data which is strongly typed. Feustel [Feustel, 1973] explains many of the reasons why a typed architecture is effective.

The five bit datatype field found in each word is used to determine the representation and semantics of the data being referenced. Normal Lisp list structure contains datatype LIST, indicating it has a Lisp node representation such as that given above. Many other datatypes exist in the system, however. Integers of less than 24 bits, for example, are represented with datatype FIXNUM and with the two's complement binary number stored in the 24 bit pointer field. Arrays, strings, and other number representations, such as floating point and so-called BIGNUM integers (larger than 24 bits), have more complex representations, which we will not discuss here.

The datatype field is also important in the implementation of system storage conventions. Given a datum, we can immediately determine the storage conventions it uses, and determine how to reference it. In addition the so-called forwarding or invisible pointers allow us to remotely reference items and change the location of items in storage without influencing their external behaviour. These datatypes act somewhat like an indirect address reference in a traditonal instruction set architecture, in that cells so marked do not contain the data being referenced, but rather a pointer to that data. The important difference is that the forwarding pointer datatype is specified with the data, rather than in the instruction being executed. Several forms of the forwarding pointer exist in the machine, differing in the conditions under which this indirection is performed.

A good example of the use of forwarding pointers is the GC-FORWARDING datatype. When the garbage collector relocates an object, it replaces the original object data with a GC-FORWARDING pointer to the new location of the object. References which still point to the old location of the relocated data will encounter this forwarding datatype and follow the indirection, eventually locating the correct data. In the case of GC-FORWARDING pointers, the forwarding pointer is spliced out of the reference path by changing the original pointer to reflect the relocated position of the data.

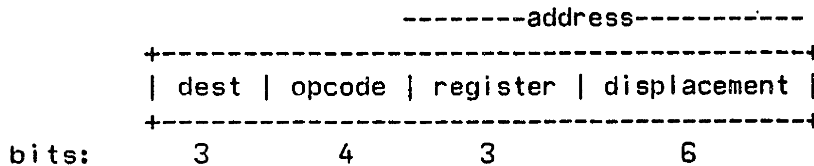
Other uses for the invisible data type include remote pointer references in the macro instruction function frame and in the efficient implementation of functional closures.

The Macro Instruction Set

We now turn to a description of the sixteen bit instructions which operate on this typed data. The basic instruction is sixteen bits long, packed two per 32 bit word. The instruction set is stack based, with operands often implicitly fetched from or stored onto the macro stack. Locations in virtual memory are never directly specified in the macro instruction set, but are rather specified with a set of automatically maintained base registers which point to relevant sections of the virtual address space.

Each instruction is divided into four fields, which are used differently in the four main instruction classes. The instruction class is specified in the four op code bits, which are decoded sixteen ways. Nine of these are address/destination instructions, and specify the operation directly in this field. Three of the sixteen combinations are decoded further with the destination field, and form the address only instruction class. One decodes the address field, providing 512 operations in the destination only class, and finally, one is decoded to provide the branch class.

The branch class decodes the destination field as a condition specifier, and uses the address field as a relative program counter increment.



Calculation of the Effective Address

The effective address for source/destination and source-only instructions is specified by the nine bit source address field. Three of these bits are used to select one of eight base registers; the remaining six bits are used as a positive displacement from the register. The registers are automatically updated by the processor when changes to the execution environment occur, such as during a function call. Four of the registers point to portions of the currently executing function entry frame, with their contents offset by 64 locations, such that a total of 256 locations may be referenced. One points to the origin of the argument list for the currently executing function, another to the local binding block for this function, and a third to a set of quoted constants, such as T and NIL. Finally, one state of the base register field is used to indicate special references. The six bit destination field is then used to specify the action. At the present time, only one state of this field is used, indicating that the addressed value is on the top of the stack, and that the stack pointer should be decremented after this reference.

The 256 locations addressable in the function entry frame often contain forwarding pointers to items of interest to the function. This is the way in which pointers to other called functions and special variables, for example, are stored.

A small class of instructions, such as MOVEM and POP (see below) use the source address to specify a location for writing; others modify the contents. The address is calculated identically in all cases.

Source/Destination Instructions

The nine source/destination instructions consist of one data moving operation [MOVE], six pointer following operations [CAR,CDR,CAAR,CDAR,CADR,CDDR] and two function calling instructions [CALL,CALL0]. Each instruction calculates its effective address from the source field, performs its operation on the data fetched, and disposes of its result as specified in the three bit destination field (see below). The CALL and CALL0 instructions behave differently, as we will see below in the discussion of the inverted calling sequence.

The Destination Field

The three bit destination field specifies the disposition of results from an instruction. The IGNORE code permits the macro instruction set to discard unneeded results, while still setting the condition code bits used for conditional branches. A destination of STACK permits pushing the result data onto the push down list. NEXT-LIST, in combination with the setup instruction LIST, provides a convenient mechanism for compiling the construction of lists. The LIST-N instruction allocates N sequential Q's from free storage and pushes its destination code and two pointers to the beginning of the allocated area. A destination field of NEXT-LIST stores

the data in CAR of what is pointed to by the top of the stack. It then replaces the top of the stack with its CDR, pointing now to the next location to be filled. If the result of this CDR is NIL, the list is full, and the other pointer to the allocated area is removed from the stack, and stored as specified by the destination code of the original LIST-N instruction.

The remaining destination field specifiers combine with the CALL and CALL0 operations to form the function calling and returning mechanism of the macro order code.

A function call with arguments is initiated by the CALL instruction. It specifies a pointer to the function cell of the called function in the address field. In the destination field, it specifies what is done with the value returned by this function call. Execution of the call instruction does not result in transfer of control to the function -- it merely sets up internal pointers and in effect prepares the machine for an eventual function call. The CALL instruction is typically followed by a series of macro instructions with a special destination code we call NEXT. It specifies that the result of this instruction is the next argument to the function which is about to be called, and causes it to be added to the argument list of that function calling frame. When a macro instruction with destination code LAST is finally executed, it stores the final argument to the function and initiates the actual transfer of control, base register modifications, and binding changes necessary to implement the call.

When the called function eventually encounters a macro instruction with the destination code RETURN it restores registers, unbinds variables, and fetches from the stack the stored destination code of the CALL instruction which originally set up its function call. It then uses that destination code to dispose of the returned function value as would any other macro instruction.

This entire process of function calling is termed the inverted calling sequence because, in contrast with the way compiled function calls occur in most Lisp implementations, the function to be called is specified prior to the evaluation of any of its arguments.

Functions of no argument are called with the CALL0 macro instruction, since there is no opportunity to terminate the actual function call with a destination code of LAST, and the function must be called immediately.

The utility of the inverted calling sequence arises from the (possible) need for the macro instruction set to behave differently in storing function arguments depending upon the level of execution of the function about to be called (macro, micro, interpreted, etc.). By providing the information concerning the function prior to the evaluation of its arguments, special action, if needed, can take place during storage of the arguments.

Destination Only Instructions

The destination only instruction class has no effective address field. Instead, it allocates the nine bit field to further specify one of 512 instructions. These instructions take arguments only on the stack, and provide the normal three bit destination code as described above. Included in this class of operations are LIST of 0 to 63 elements [see the discussion of destination NEXT-LIST above], all pointer manipulation instructions of the form CxxxR and CxxxxR (the x's representing either A or D for CAR or CDR), and a variety of Lisp primitive functions such as CONS, NCONS, GET, ASSOC, ASSQ and EQUAL. These functions could of course be called with the normal function calling sequence, but this method is more efficient.

Address Only Instructions

Two of the sixteen possible main instruction codes are further decoded from the

destination field. The sixteen operations provided in this way are:

MOVEM Stores the top of the stack into the effective address

SCDR Replaces the contents of the effective address with its CDR

SCDDR Replaces the contents of the effective address with its CDDR

1+ Replace the contents of the effective address with itself plus one

1- Replace the contents of the effective address with itself minus one

\>
<
EQ
Pop one argument from the top of the stack, compare with the contents of the effective address, and set the appropriate condition code indicators

ADD
SUB
MUL
DIV
REM
These arithmetic operations take one operand from the stack and the other from the effective address. The result replaces the operand on the stack.

Branches

All data handling operations set two condition code bits as a result of their execution. For list manipulating instructions these indicators are the ATOM indicator and the NIL indicator, and they are set or cleared according to whether the result was atomic or null, respectively.

The branch instruction decodes the three bit destination field as a branch condition. The conditions include an unconditional branch, either state of the two condition code bits, and two branch conditions which test the NIL indicator and, if it is set, pop the stack in addition to performing the normal conditional branch. This operation is useful in compiling the end test of functions which examine sequential elements of a list.

Sample Compiled Macrocode Sequences

Some simple examples of Lisp functions with the corresponding macro compiled instructions are presented below. For comparison of relative bit efficiencies, we also include the PDP-10 compiled version, produced by the MacLisp number compiler.

The trivial call to a built in function:

(defun add (x y)(+ x y))

Macro code

```
MOVE    D-PDL     ARG|0          ;FIRST ARGUMENT (X)
+                 ARG|1          ;SECOND ARGUMENT (Y)
                                 ;OTHER ARGUMENT ON THE STACK, RESULT TO STACK
MOVE    D-RETURN SPECIAL|77      ;RETURN THE TOP ELEMENT OF THE STACK,
                                 ;AND POP THE STACK POINTER
```

A simple recursive functional call

(defun fact (n)(cond ((= n 1) 1)(t (* n (fact (1- n))))))

```
MOVE    D-PDL           ARG|0               ;FIRST ARGUMENT TO STACK
EQ                      FEF|(QUOTE 1)       ;COMPARE WITH QUOTED CONSTANT IN FUNCTION ENTRY FRAME
BRANCH  NILIND-TRUE     MORE                ;BRANCH IF THE RESULT IS NIL
MOVE    D-RETURN        FEF|(QUOTE 1)       ;RETURN QUOTED CONSTANT STORED IN FUNCTION ENTRY FRAME

MORE:
MOVE    D-PDL           ARG|0               ;FIRST ARGUMENT TO STACK
CALL    D-PDL           FEF|(*FUNCELL FACT)         ;SETUP TO CALL FUNCTION, POINTER TO FUNCTION
                                            ;CELL LOCATION IS STORED IN FEF
MOVE    D-PDL           ARG|0               ;ARGUMENT TO STACK
1-      D-LAST                              ;SUBTRACT ONE FROM TOP OF STACK, INITIATE RECURSIVE
                                            ;FUNCTION CALL
*                       SPECIAL|77          ;MULTIPLY TOP OF STACK (RETURNED FUNCTION VALUE)
                                            ;WITH SECOND ELEMENT OF STACK (ARGUMENT OF THIS FUNCTION)
                                            ;RESULT TO STACK
MOVE    D-RETURN        SPECIAL|77          ;RETURN TOP OF STACK

total instructions: 10                      total bits:  160
```

PDP-10 MacLisp Compiled output

```
            PUSH P,1                        ;ARGUMENT TO STACK
            MOVE 7,(1)                      ;NUMERIC VALUE OF ARGUMENT
            SOJN 7,G0002                    ;SUBTRACT ONE; IF NON-ZERO, GOTO G0002
            MOVEI 1,(QUOTE 1)               ;PICKUP POINTER TO QUOTED VALUE
            JRST G0001                      ;TRANSFER TO EXIT

G0002:      MOVE 7,(1)                      ;PICK UP NUMERIC VALUE OF ARGUMENT
            SUBI 7,1                        ;SUBTRACT ONE
            PUSH FXP,7                      ;PUSH ONTO THE INTEGER STACK
            MOVEI 1,(FXP)                   ;PICK UP A POINTER TO THE TOP OF THE INTEGER STACK
            CALL 1,(QUOTE FACT)             ;CALL RECURSIVELY USING THIS POINTER AS AN ARGUMENT
            MOVE 7,@(P)                     ;PICKUP NUMERIC VALUE OF ARGUMENT
            IMUL 7,(1)                      ;MULTIPLY BY NUMERIC VALUE OF RETURNED DATA
            JSP T,FXCONS                    ;BOX THE RESULT, RETURNING A POINTER IN AC1
            SUB FXP,[1,,1]                  ;CLEANUP AND POP THE TWO STACKS
G0001:      SUB P,[1,,1]
            POPJ P,                         ;EXIT

total instructions: 16                      total bits:  576
```

A simple iterative function definition

```
(defun sum (x)(prog (result)
                    (setq result 0)
         a          (cond ((null x)(return result))(t (setq result (+ result (car x)))
                                                       (setq x (cdr x))
                                                       (go a)))))
```

```
MOVE      D-PDL     FEF|(QUOTE 0)           ;MOVE A QUOTED CONSTANT ONTO THE STACK
POP                 LOCBLOCK|0              ;STORE IN A LOCAL VARIABLE (RESULT)

A:
MOVE      D-IGNORE          ARG|0                     ;TEST FIRST ARGUMENT (X)
BRANCH    NILIND-FALSE      MORE            ;BRANCH IF NON-NIL
MOVE      D-RETURN          LOCBLOCK|0                ;RETURN RESULT

MORE:
MOVE      D-PDL     LOCBLOCK|0              ;PUSH LOCAL VARIABLE (RESULT) ONTO STACK
CAR       D-PDL     ARG|0                   ;PUSH CAR OF THE FIRST ARGUMENT ONTO STACK
+                   SPECIAL|77              ;ADD TOP TWO STACK ELEMENTS, RESULT TO STACK
POP                 LOCBLOCK|0              ;STORE TOP OF STACK IN LOCAL VARIABLE (RESULT)
SETE-CDR            ARG|0                   ;REPLACE THE ARGUMENT WITH ITS CDR
BRANCH    ALWAYS    A                       ;CLOSE THE LOOP
```

An examination of the macro compiled code shown above will probably convince the reader that this particular macro instruction set is not the ultimate in bit efficiency for compiled Lisp representation. Other bit efficient architectures have been described using eight bit instructions [Deutsch 1973], which, although smaller, tend to be much more restrictive in their ability to reference data. The tradeoffs come down to decisions on the complexity of the compiler construction and the ease of thinking about and implementing micro coded interpreters for these architectures. The beauty of a writable microcode processor is that this sort of investigation and tuning of the macro coded architecture is possible without change to the hardware. Indeed, it is possible to construct a system where several radically different macro instruction formats co-exist, each able to call the other, and each optimized for the compilation of a particular style of user or system function.

Other Microcode Components

The emulator for the macro code architecture is an important part of the microcode environment supported by the machine, but by no means comprises even the majority of the micro instructions written in support of the environment. Many other functions essential to the operation of the system reside in the microcode memory, amounting to essentially the operating system for the machine. Among these are support for the virtual memory system, device drivers for the disk, display, and network, and a wide selection of hand microcoded Lisp functions resident for efficiency reasons.

Storage Management

One major task for a Lisp runtime environment is mangement of storage in an automatically reclaimable way. Traditional Lisp systems manage storage by incremental allocation in the primitive CONS (or in array allocations), relying upon a garbage collection process to reclaim space occupied by discarded data. This Lisp system uses a new form of garbage collector discovered by Henry Baker [Baker, 1977]. The major feature of this collector is that it operates incrementally, never delaying ongoing computations for a sudden unexpected restructuring of the address space. Efficient implementation of this collector requires the ability to quickly determine to which portion of address space newly fetched data points. This feature is provided in our processor design with an automatic lookup of the pointer space via a special path into the memory map.

Conventional non-microprogrammed processors, which often provide convenient operations for pointer manipulations, find this sort of slight modification in their behaviour difficult without drastically increased overhead.

Summary

We have attempted to demonstrate the increase in the complexity of the interpretation of the macro instruction set as a result of the optimization of that architecture to the needs of a bit efficient representation for the compiled Lisp function. The complexity is introduced both in the execution of such features in the instruction set as the DESTINATION-LAST and DESTINATION-RETURN tags, and in the interpretation of the typed data, with its run time datatype checking and handling of complex storage conventions such as the CDR-CODE fields, the forwarding datatypes, and the Baker style relocating garbage collection.

The large quantities of microcode needed for operating system implementations makes

the ease with which the machine may be microprogrammed an important consideration. The eventual desire for microcode produced by a Lisp function compilation process constrains the complexity of the microcode and provides incentive for making the structure logically complete and conceptually simple.

The next chapter describes the architecture of the microprogrammable processor we have designed to implement this environment.

VI. The Microprogrammable Processor

This chapter concerns the features of the microprocessor which executes the Lisp Machine macro instruction set and supports the macrocode operating environment. First, we present the design of the machine in detail, and then come back to consider the aspects of the design which are unique, and how they help in execution of the macro instruction set we have just described.

The Processor

The processor is a synchronous 32 bit wide machine, with two main data manipulation paths, as shown in figure 4. On each cycle, a pair of 32 bit operands is fetched onto the A and M busses, an operation performed on them, and (usually) a 32 bit result written. The microprocessor is similar in many respects to a traditional computer, as distinguished from the majority of microprocessors. It has, for example, a traditional incrementing micro program counter, instead of the more commonly encountered micro instruction next address field. It is perhaps best compared to a three address (two sources and a destination) computer with a small address space (the registers) and extensive secondary memory (the main memory). The microcode instruction format is largely vertical, meaning that fields are decoded differently within different instructions, instead of taking direct action on the hardware.

The Microinstructions

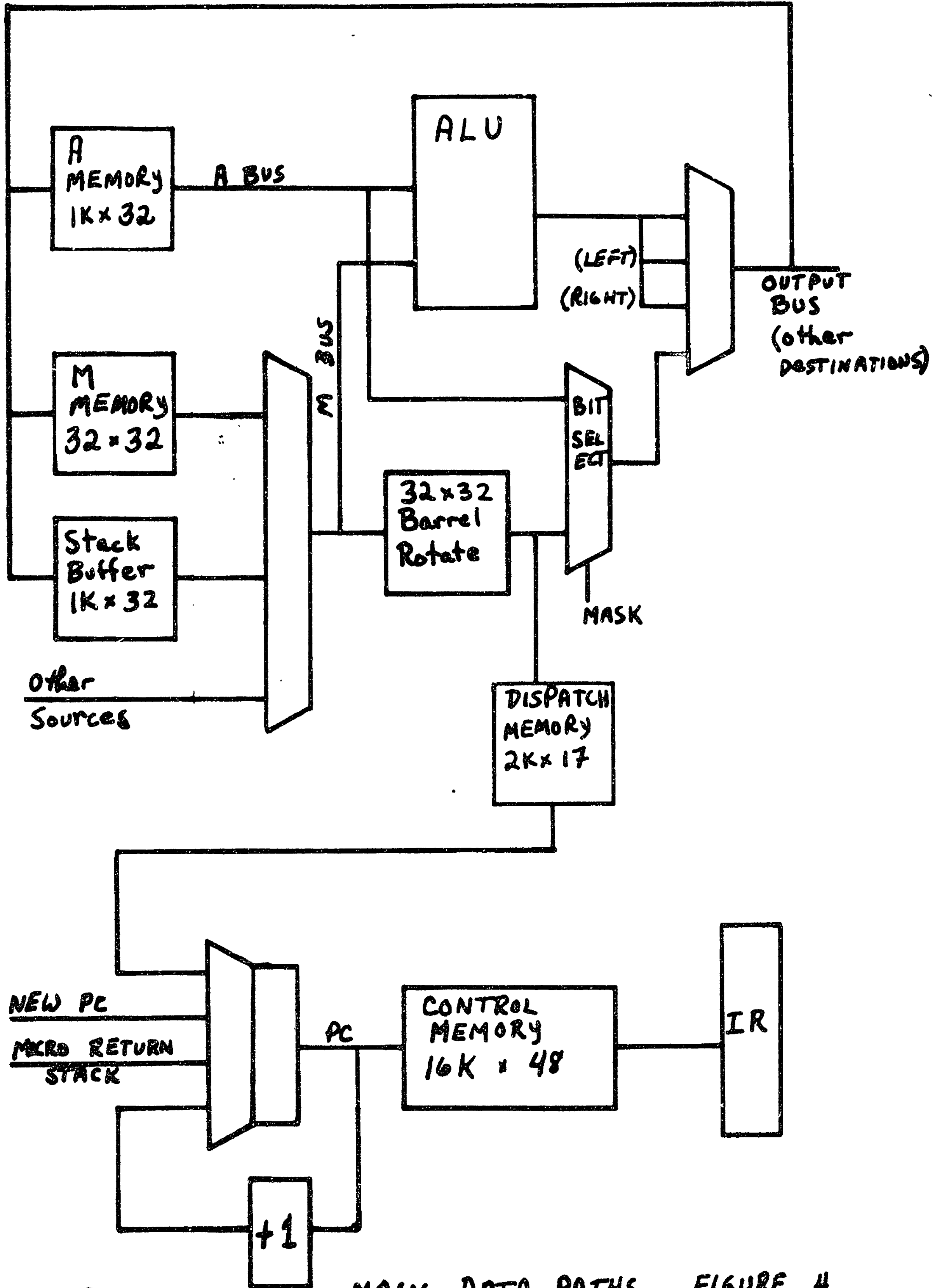
There are four major microinstructions, specified by a two bit field. Each begins with identical specification of operands to drive the A and M busses. Two, the ALU instruction and the BYTE instruction, perform data manipulations on these operands, and then return the result to a register specified in the micro instruction destination field. The other two instructions, JUMP and DISPATCH, do not store results, but rather affect the flow of control in the processor. The format of the microinstruction word for the four main instructions is shown in figure 5.

Sources

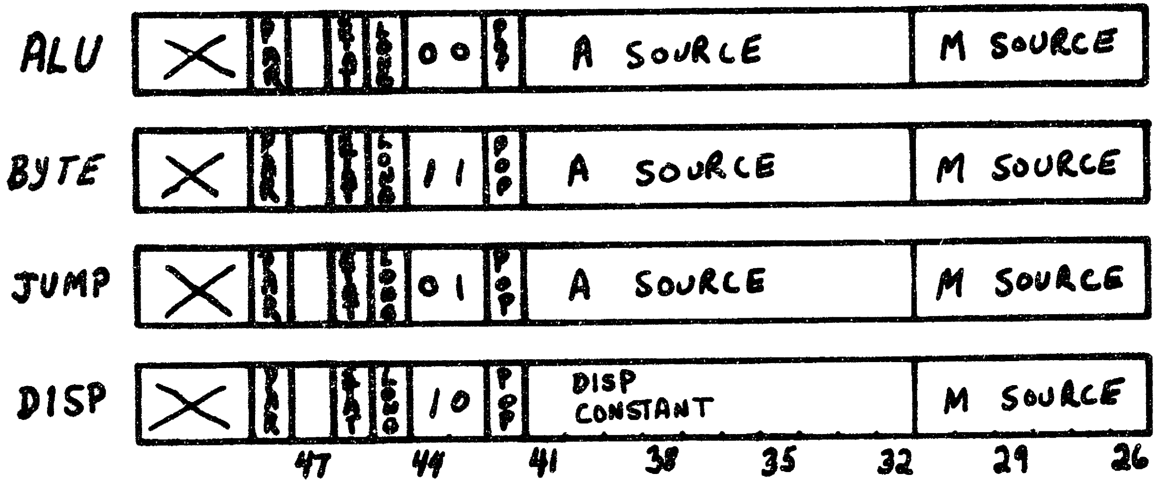
All cycles begin with uniform fetching of two operands, specified by two fields in the microinstruction. This uniformity of register reference in the micro order code allows fast gating of the register addresses into the register files following the main clock, an operation which is in the critical path for the micro cycle length.

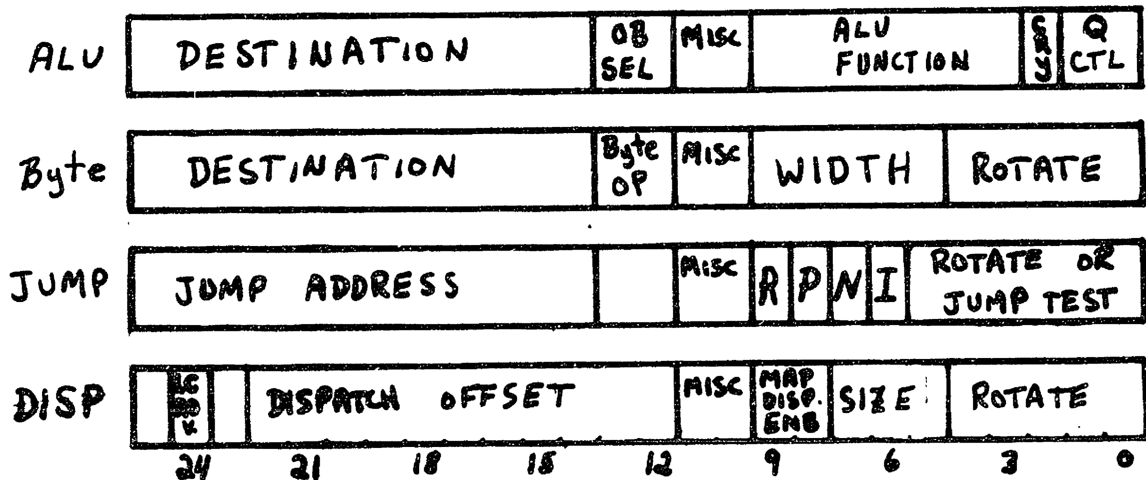
The ten bit A source field determines which one of the 1024 possible internal memory locations will drive the A bus. The six bit M source field is more complex. Values from 0 to 31 specify one of 32 internal memory locations, identical in contents to, but distinct in implementation from, the low 32 locations accessible from the A bus. The remaining values of the M source field specify that the M bus be driven from a variety of miscellaneous internal registers. Among these are the data from main memory, the stack buffer, the Q register and the micro program counter subroutine return stack. These special bus sources will be discussed later in more detail. Table 1 summarizes the source specifications and figure 6 shows a detailed datapath diagram of the A and M source section of the processor.

The ALU Datapath



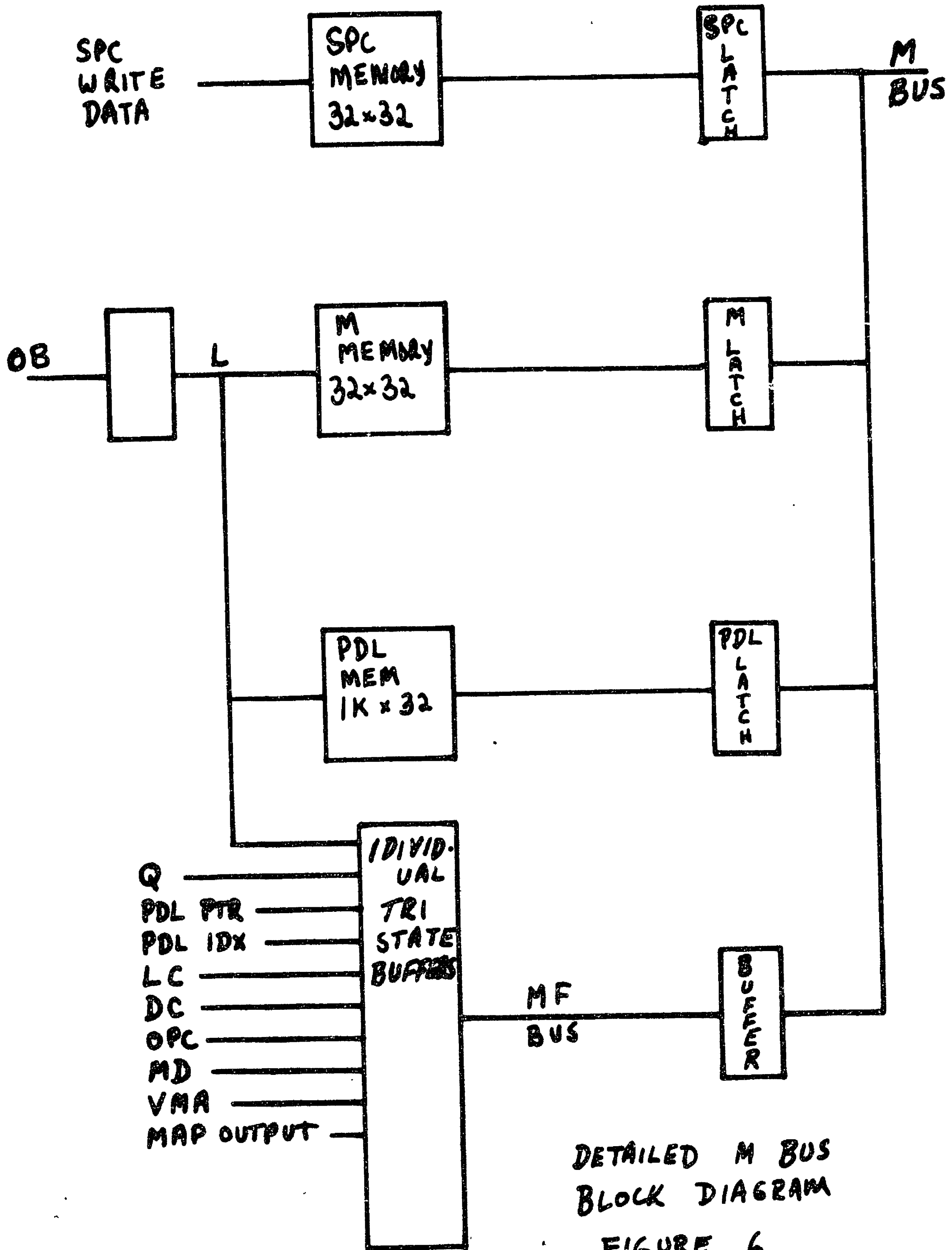
MAIN DATA PATHS FIGURE 4





MICRO INSTRUCTION FORMAT

FIGURE 5



DETAILED M BUS BLOCK DIAGRAM

FIGURE 6

The ALU instruction provides basic arithmetic and logical data manipulation in the machine. The two operands supplied on the A and M busses are gated into an arithmetic logic unit constructed from the 74S181 integrated circuit. This device provides all sixteen bitwise logical operations on the operands (such as $A \vee M$, $A \wedge \neg M$, A, $\neg M$ etc.) and a variety of arithmetic operations, such as $A+M$ and $M-A$. Notable by its absence is the operation of $A-M$. The carry into the low order bit position is controlled independently by the micro instruction. Normally, the operation to be performed on the data is specified in a six bit field in the micro instruction, but in the case of multiply step and divide step, the function performed is determined in part by the data being handled.

Divide and multiply use the 32 bit Q register to form extended precision 64 bit results. The Q shift register is controlled by a two bit field in the ALU instruction, and can be loaded or shifted left or right. Right shifts shift in the low order bit of the ALU output, and left shifts shift in the complement of the sign of the ALU output. These shift paths are primarily designed to allow the multiply step and divide step instructions to take place in one cycle.

The output of the arithmetic logic unit is optionally shifted left or right by one bit position, and written into the location specified in the destination field. Right shifts duplicate the ALU sign bit, and left shifts insert the high order bit of the Q register.

The Shifter/Masker Datapath

The shifter/masker datapath provides convenient mechanisms for extracting, depositing, and moving arbitrary length and position bit strings within the processor. The mechanism is implemented in two stages. First, the M bus data is routed through a combinatorial shifter, allowing an arbitrary rotation of the 32 bit word. This type of network is sometimes termed a "barrel shifter." The shifter output, together with the A bus data is routed to the masker. The masker then combines these inputs using a bitwise selection of each output bit from either the rotated shifter output or the A bus data. The masker selection is controlled by a 32 bit mask generated by a field in the microinstruction. The mask may specify any number of contiguous bits, positioned any number of bits from the right end of the word. Two five bit numbers control the mask generation process. A single five bit number specifies the rotation of the M bus data in the shifter. Three five bit numbers thus control the action of the shifter/masker datapath.

The useful operations performed in this datapath can, however, be specified in only two five bit fields. We observe that the operations that are most commonly desired are field extraction, with a right justified result; field deposit, with a right justified M bus source; and an operation we call selective deposit, which deposits a specified field of the M bus data into the same size and position field in the A bus data.

In all of these operations, the right most bit position of the mask and the amount by which the M bus data is rotated are either the same (field deposit), or one or the other is zero (field extract has right bit of mask zero; selective deposit has the rotate zero) Thus, we can specify the operation of the masker/shifter with two five bit fields, plus two bits to control the zeroing of either the rotate or right bit of mask inputs.

The operations we provide in the datapath are, then:

Extract Field extract a contiguous set of bits from the M bus data, right justify it, and replace the rightmost bits of the A bus data with those bits to form the result of the instruction. The mask right most bit field is forced to zero, the rotation is taken from the microinstruction.

Deposit Field take a right justified field from the M bus data, shift it left some amount, and replace the same length field in the A bus data with the rotator output to form the result. Both the rotator input and the right most bit of the mask are taken from the microinstruction.

Selective Deposit take a field from the M bus data and replace a similarly located field in the A bus data with it to form the result. The rotate input is forced to zero, the right most mask bit location is taken from the microinstruction.

As in the ALU instruction, the result is written into locations as specified in the destination field.

The Destination Field

The destination field controls where the result of the ALU and Byte operations is stored. Two distinct formats for this field exist. The first specifies a single address of 10 bits, adequate to specify any internal memory location. The second allows two locations to be written simultaneously. Five bits specify one of the low 32 internal memory locations (those accessible from both the A and M busses), while the remaining five bits specify one of a number of so-called functional destinations, such as the external main memory write data and the stack buffer. Writing certain of the functional destinations initiates special processor action, such as main memory accesses for either reads or writes. The detailed description of the destination field is shown in table 2.

Flow of Control

The processor normally executes sequential instructions in the microprogram memory, much like traditional machine language architectures. This normal instruction flow can be interrupted by execution of either of the two remaining main instruction types, the conditional jump or the dispatch. Microinstruction fetch in the processor is pipelined one level, resulting in the prior fetch of one instruction after any successful transfer of control. The programmer has the option, in these cases, of either executing this instruction in the normal way, or of discarding the instruction and wasting the cycle for which it was fetched.

Thus, for example, a two instruction loop could either be written as this:

tag: <instruction>
<transfer to tag / inhibit next prefetched instruction>

which would execute a total of three processor cycles per loop execution, or as this:

tag: <transfer to tag/ execute next prefetched instruction>
<instruction>

which loops every two processor cycles.

Most often, useful work can be found for the instruction following the transfer of control. All unconditional branches, for example, can be followed by the initial instruction of the called routine, adjusting the branched-to-location forward by one. Due to special logic in the microsubroutine call/return logic, this can even be done on unconditonal microsubroutine calls. Conditional transfers sometimes provide a problem in demanding additional cycles, but often the setup for one of the paths will not be harmful to the other, allowing optimization in some

percentage of the execution paths.

Jumps

The jump instruction allows conditional transfer of control depending on a wide variety of internal processor conditions. Like all other microinstructions, it specifies two operands for the A and M busses. Both main datapaths of the machine are used for processing this data.

One class of conditional jumps compares the A and M operands via the ALU datapath. The ALU is setup to perform an unconditional subtract of the A and M bus data. Special logic detects the all-zero output condition in the ALU data, and the sign bit of the ALU is also available for testing. Thus, this form of the conditional jump may be made conditional on the A data less than, less than or equal, equal, greater than, greater than or equal, or not equal to the M operand. The "always" and "never" conditions are also available here for completeness.

Several special conditions internal to the processor, such as pending interrupts and failure of the main memory associative mapping hardware to produce a match may also be tested. Table 3 shows the jump field specifiers available.

A second class of conditional jumps utilizes the shifter matrix for its conditional test. The M bus data is rotated by an amount specified in the microinstruction, and the rightmost bit is tested for being either a zero or a one. Thus, any single bit accessible from the M bus may be tested in one cycle.

The action of the conditional jump instruction upon the processor when the condition is satisfied is controlled by a three bit field in the microinstruction. One of these bits, the N bit, inhibits exection of the immediately following instruction, wasting the cycle that its execution would have occupied.

The remaining two bits, the P bit and the R bit, are decoded four ways to provide different types of transfers.

If both the P and R bits are zero, the processor executes a simple transfer of control to the location specified in the fourteen bit jump address field of the microinstruction.

If the P bit is a one, then the transfer is performed as in the previous case, but the current micro program counter (the PC). is saved on the micro subroutine return stack (the SPC). This allows control to be transfered back to the next instruction in sequence, if desired. A slight complication of this feature is the interaction between this and the next instruction execution flow of control. If the following instruction is executed, then the subroutine should return not to the instruction following the jump, but to the one after that. Thus, when the N bit is off, specifying execution of the following instruction, the saved PC plus one is saved as the subroutine return, rather than the PC.

Microsubroutine returns are specified by successful jumps with the P bit zero and the R bit one. In this case, the jump field of the microinstruction word is ignored, the new PC is taken from the top of the SPC stack and the stack is popped.

The return from microsubroutines is a sufficiently common operation, and one which is easily enough specified, that an independent, unconditional mechanism is provided for its execution. Each microinstruction has a bit we call the POPJ-AFTER-NEXT bit, which forces the PC to be loaded from the SPC stack. It does not inhibit execution of the following instruction, so the transfer of control becomes effective after the next instruction is executed. This saves both the time and space taken by the common case of returns from microsubroutines.

The final decoding of the P and R bits specifies a write of the microinstruction memory. The sequence of actions necessary to write microinstruction memory from the running

processor is rather complex. The goal is to load the program counter (which addresses the microprogram memory) with the address of the location to be written, then to initiate a write cycle, and then return to the normal instruction sequence. Since the PC must be loaded, in many ways the microinstruction write is similar to a jump, accounting for its place in the microcode instruction set.

The sequence of events on a cycle by cycle basis follows. First, the write instruction loads the PC with the jump address field of the microinstruction. Simultaneously, it saves the current PC on the SPC stack, in the same way a microsubroutine call would. The following cycle is normally specified as a NOP with the N bit set in the instruction specifying the write. During this cycle, the PC contains the address of the location to be written. The data (saved from the A and M bus operands on the previous cycle) is written into the addressed microinstruction location. The instruction fetched by the processor during this cycle is garbage, and will be automatically NOPed by the processor. A microsubroutine return is forced during this cycle, loading the PC with the address of the instruction following the original write instruction. The next cycle is NOPed since its microinstruction register contains invalid data. During this cycle, the contents of the location following the original write is being fetched prior to resumption of normal processor operation on the next cycle.

Dispatching

A key operation necessary for emulation of instruction sets is the ability to transfer to one of several different locations depending upon the contents of a specified field. The processor implements this operation as one of its four basic microinstructions. Making use of the same shift matrix as the BYTE instruction, the M bus operand is rotated by an amount specified in the microinstruction. The resulting word is masked to a variable number of low order bits, thus extracting a particular field of the M operand. This field is then bitwise ORed with an eleven bit microinstruction field, called the dispatch offset. The resulting eleven bit number is used as a table index into the 2048 location dispatch memory. There, a new program counter, plus a set of N, P, and R bits, similar in function to the microinstruction bits of the conditional jump are fetched. Data flow for the dispatch instruction is shown in figure 7.

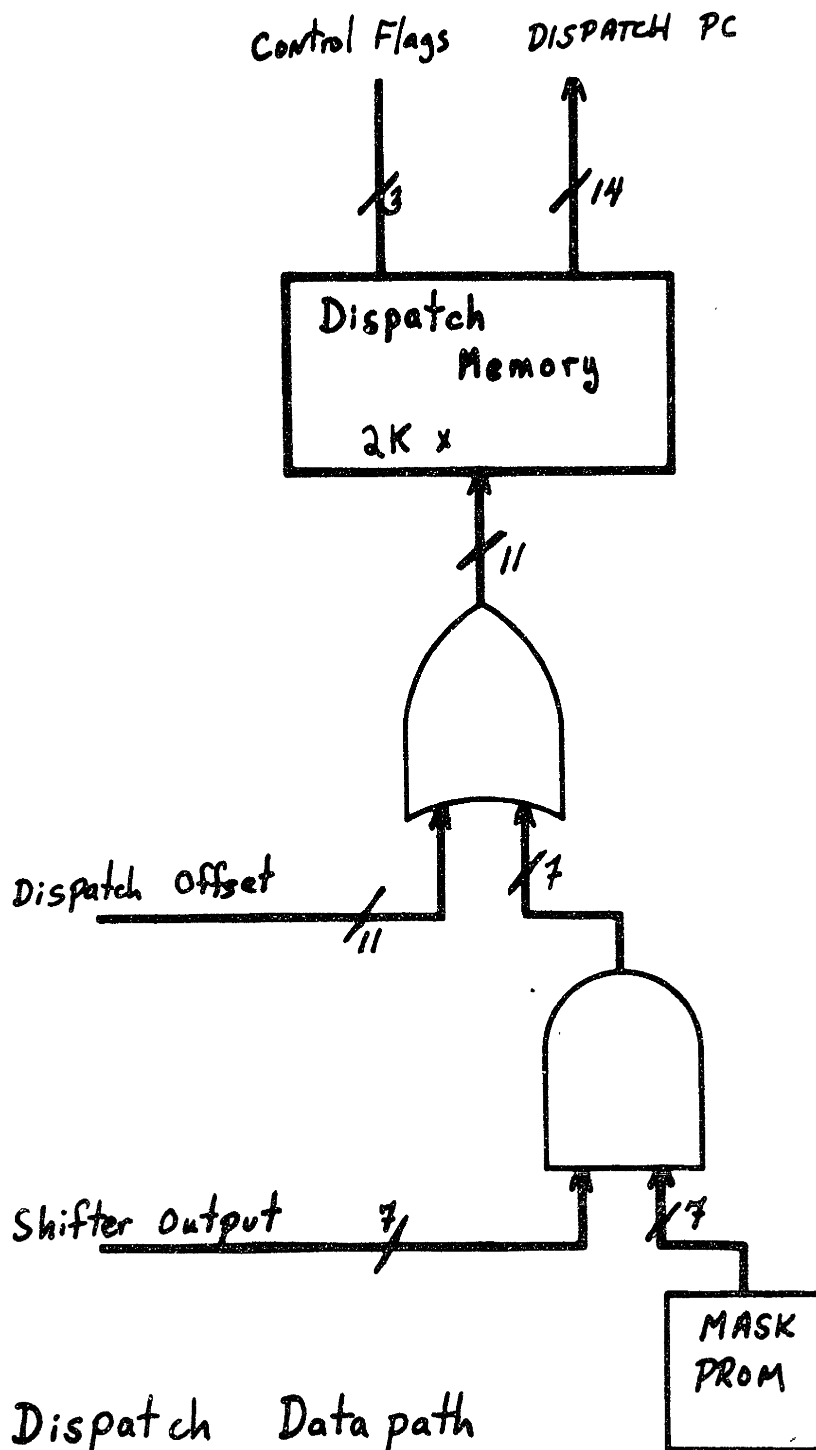
Thus, we allow, conditional on the contents of a specified arbitrarily placed field, execution of transfers, subroutine calls, or subroutine returns. The combination of P = 1 and R = 1 is decoded differently in the case of dispatch instructions, however. In dispatches it indicates that the instruction stream should not be interrupted, and is said to FALL-THROUGH. This feature is particularly good in testing for exceptional conditions, since in the normal case execution will continue without wasted cycles, while in the special conditions being tested, the N bit of the dispatch table entry may be specified to inhibit execution of the following cycle.

The dispatch instruction is also used, with miscellaneous function one set, to write the dispatch memory contents. Data to be written is taken from the low order bits of the M bus operand.

Main Memory Reference

References to main memory in the processor are implemented as special M bus sources and as special functional destinations.

Read memory cycles are initiated by loading the functional destination VMA-START-READ with the address of the location to be read. This initiates a map cycle (see below), and if the map is set up, it starts a main memory fetch at the physical address specified by the map. The



Dispatch Datapath
Figure 7

processor is free to execute additional microinstructions following the read cycle initiation. Testing of map misses and page faults is recommended as the immediately following cycle. The processor accesses the data returned by the memory system with any instruction specifying READ-MEMORY-DATA as an M source. If such an instruction is encountered prior to the availability of the data, the execution of processor cycles is delayed until the data is ready. Approximately three cycles following the initiation of the read are available prior to the data being ready.

Write cycles may be initiated in one of two different ways. Two operands are needed for a write, the memory data and the memory address. These may be specified by the functional destinations VMA, for the address, or MD for the memory write data. Initiation of the cycle may be specified along with the loading of either of the registers by specifying the functional destination VMA-START-WRITE or MD-START-WRITE. Again, prior to actually initiating the memory cycle, the processor executes a map cycle. The validity of the map entry should be checked on the cycle following the the write initiate by execution of a conditional jump instruction testing the page fault condition. Typical sequences for initiating main memory writes might look like this:

```
((VMA) M-ADDRESS)                       ;LOAD THE MEMORY ADDRESS REGISTER
                                        ;WITH THE VIRTUAL MEMORY LOCATION
                                        ;TO BE WRITTEN
((MD-START-WRITE) M-DATA)               ;LOAD THE WRITE DATA REGISTER WITH
                                        ;DATA TO BE WRITTEN, START THE WRITE
(CALL-CONDITIONAL PAGE-FAULT
         PAGE-FAULT-HANDLER)            ;TEST FOR MAP MISS OR PAGE FAULT
```

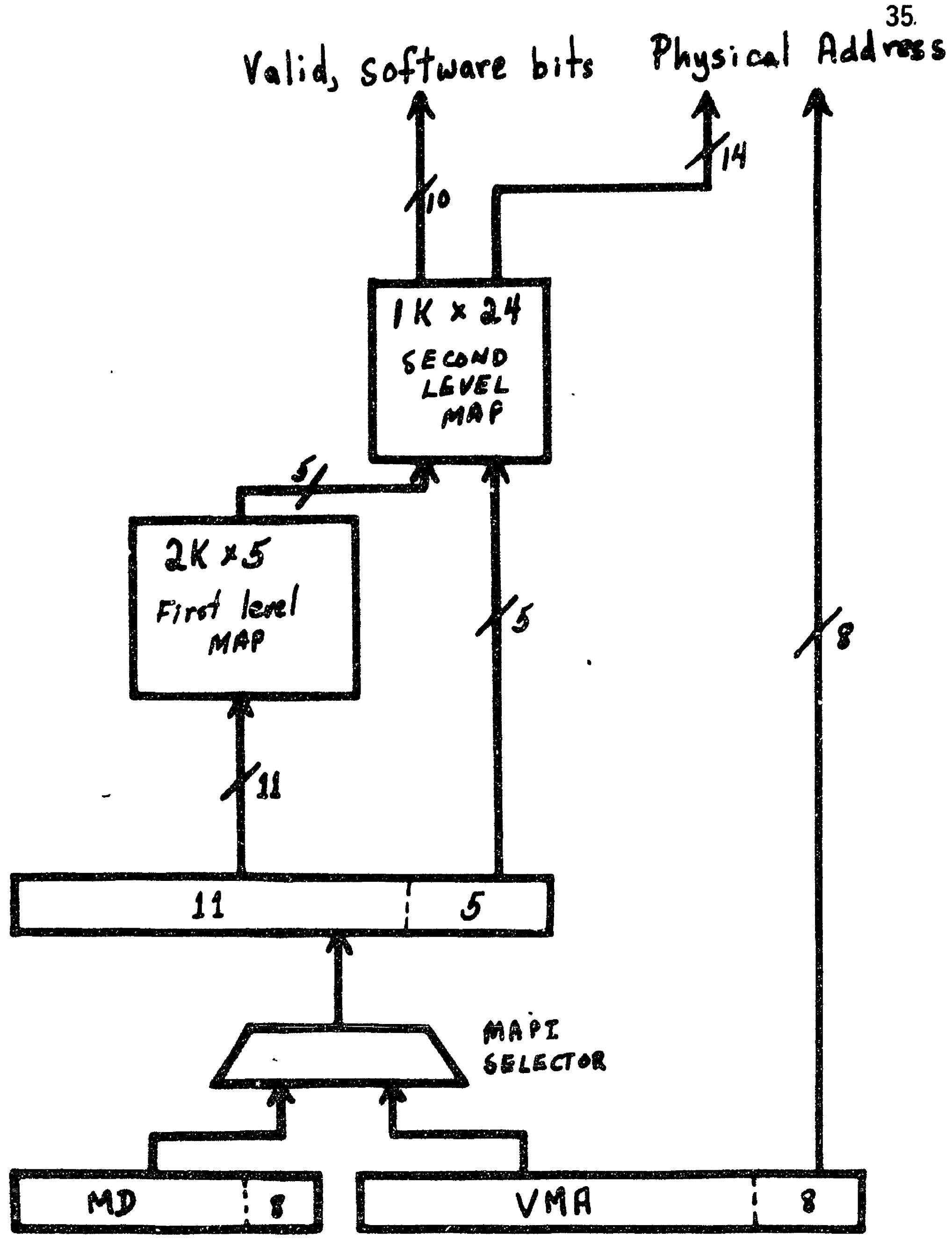
Often this case can be simplified when the VMA register already contains the correct data, as when a read has just taken place from the same location.

```
((MD) M-DATA)                           ;LOAD WRITE DATA
((VMA-START-WRITE M-ADDRESS)            ;LOAD VMA, INITIATE WRITE
(CALL-CONDITIONAL PAGE-FAULT
         PAGE-FAULT-HANDLER)            ;TEST FOR MAP MISS OR PAGE FAULT
```

No features of the processor are provided to allow read-pause-write operation, because of the complexity of the logic required and the use of a semiconductor based main memory system, where the only advantage a read-pause-write scheme has is for simple types of mult-process interlocks, which, in a single processor system, are not required.

The Map

The processor memory address register can address up to 24 bits of virtual memory. A mapping is made prior to read and write requests to produce a real core address for the reference. The table relating virtual to physical core locations is kept in main memory, and certain entries in that table are duplicated in fast access registers in the processor. The 24 bit address is divided into three portions for purposes of mapping (see figure 8). The low order eight bits are passed directly to main memory from the address register, implying a page size in the system of 256 locations. The high order eleven bits of the memory address register index into a table of 2048 entries of five bits each. Out of all of these entries, a maximum of 31, each with a different table



MEMORY MAP
DATA FLOW
FIGURE 8

value, are non-zero and are designated as "valid".

The five bits specifying the valid table entry and the remaining middle five bits of the memory address register form a ten bit address for access to a 1024 by 20 bit memory, which holds a page table entry.

The page table entry specifies read access, write access, and the physical page address in core memory. The real time garbage collection algorithm which we utilize requires extra bits in the map entry to specify the nature of the address space which a pointer is referencing.

The performance of this type of memory map is quite good. At a minimum, the map can hold 31 entries of the in-core page table, if only a single combination of the five middle address bits for each of the valid 31 first level entries exist. At its best, the map can hold 31*32 entries of the page table, when each of the 32 combinations of the middle bits references a valid page table entry. Due to the locality of references in the macro compiled code, and in the linearized list structure, we expect that the second level map will often contain more than one valid entry per non-zero first level map combination.

Map Interface to the Dispatch Instruction

Normally, a memory map is used for performing a translation between virtual and physical memory locations. In our processor design, the map performs not only this function, but also enforces the oldspace/newspace distinctions of memory pages implied by the Baker garbage collector. When unused for mapping newly issued memory requests, the map address inputs are taken from the memory data register. This provides the processor with access to the page table entry data for the pointer which is in the MD register. At the termination of a main memory read cycle, the MD is loaded with new data from main memory. The first thing which is done with this data in the vast majority of cases is to examine the data type of the entry, and, as required by the Baker garbage collector, to determine whether it is a new or old space pointer. With the map entry lookup on the MD register, the processor has available quickly the new/old space data for the newly fetched pointer. A special type of dispatch instruction may be executed which combines the old/new space map output bit with the normal field selected by the dispatch instruction (usually the datatype of the MD) to check in one intstruction both the datatype and old/new space location of the newly fetched data.

The Stack buffer

The stack buffer provides a sort of cache mechanism for processor memory references to the top of the macro code stack. It is a 1024 location register bank internal to the processor, some portion of which holds the top of the macro code push down list. On macro instruction calls and macro instruction returns, one of the housekeeping activities performed by the micro code is to fill or empty this register bank from main memory in order to maintain valid data in the stack buffer. Thus, between macro code calls and returns, (i.e. within a function) all references to the macro code stack can refer instead to the contents of this register bank.

Note that, unlike a cache, the data in the stack buffer is required to be present. There is no probablistic element in its utility. The stack buffer performs best in situations where the execution trace is relatively shallow, that is, has many function calls and returns at nearly the same level on the stack. This requirement is almost always satisfied in real programs. If this requirement is not met, the stack buffer mechanism results in effectively moving the location of the memory read or write, with no saving in time.

The stack buffer is implemented as a 1K by 32 static RAM array addressed by two ten

bit pointers. One pointer is an up/down counter, used to address the top of the macro stack within the stack buffer. the other pointer is a fixed register which can be loaded in order to index into a fixed location in the buffer. It is intended for use in applications where the processor wishes to access, for instance, the third element back from the top of the stack.

Special M source locations provide convenient access to data from the stack buffer indexed by either of these pointers. The up/down counter may be optionally decremented after a fetch of data, providing the macro code "pop" operation at the microcode level.

Likewise, special functional destinations provide for writing into the stack buffers, indexed by either of the pointers, and the up/down counter may be incremented prior to the write, providing the "push" operation.

On a macro function call, the microcode checks that at least some minimum amount of data space in the stack buffer is free, such that the called function can push data on the stack without further checks. If there is no free space, a portion of the stack buffer data is written into main memory, creating free space.

On macro function returns, the microcode checks that at least the entire stack frame being returned to is resident in the stack buffer, again assuring that all referenced data is within the stack buffer.

Main memory references to locations which are currently held in the stack buffer are intercepted by the microcode by making the pages upon which the data resides "invalid" in the page map. Thus, the rare reference to those locations by other than pushing or poping data in the macro code instruction stream can be caught and interpreted as a reference to the data held in the stack buffer.

Unibus references

The 22 physical address bits from the processor drive a 32 data bit main memory bus which we call the XBUS. Additionally, 1/32 of the processor physical address space is occupied by the address space of the Unibus. References to this portion of the physical address space initiate 16 bit reads or writes on the processor Unibus. The data for these transfers is supplied from and returned to the right most 16 bits of the 32 bit processor word.

Unibus devices may reference the main processor memory sAvia the XBUS. This ability is important if, as in the prototype, we wish to use Unibus compatible devices for disk swapping of main memory. A problem encountered in this mode of operation is that the Unibus transfers only 16 bits at a time, while the XBUS transfers are 32 bits wide. This problem is solved by having the Unibus interface buffer the even Unibus word on writes, and the odd Unibus words on reads from the main memory. The alternative approach of cycling the XBUS memories twice per 32 bit word transfered would not have transfered quickly enough to keep up with the disk.

Unibus devices also require an address translation mechanism to allow them access to the full address space of the XBUS. The Unibus address space of 17 bits (of 16 bit words) is not adequate for addressing the much larger XBUS space. Further, as pages are transfered to and from memory, the mapping from processor virtual address space to physical location in XBUS address space becomes complex. For these reasons, we provide a mapping register between the Unibus address space and the XBUS address space. This mapping register is itself a Unibus device, and can be loaded by either the processor, through its access to the Unibus address space, or by any Unibus device.

The Macro Program Counter and Automatic Macro Instruction Fetch

The main macro instruction execution loop in the microcode sequentially fetches and decodes the 16 bit macro instructions from the function entry frame. If this process was coded with the normal microcode instruction set, the resulting loop would include such overhead items as the macro program counter increment, the loading of the VMA with the macro program counter, and a set of moderately complex routines associated with the branch instruction microcode to handle the cases of transfer to macro instructions which do not fall on a 32 bit word boundary.

Many of these overhead items are handled automatically by this processor as built in functions. The main added feature is the inclusion of a macro program counter, called the LC (location counter) to distinguish it from the micro program counter. During execution of a macro instruction, the LC register holds the byte address of the macro instruction to be executed next. At the completion of execution of the currently executing macro instruction, the incrementing, overflow testing, and main memory references resulting from overflows are automatically performed on this register.

Normally, the word containing the currently executing macro instruction is held in an M memory location, so that byte and dispatch access to its fields is easily obtained. In the current software system, each macrocode instruction is 16 bits long, packed two per 32 bit word. The half of this word which is referenced by microcoded byte or dispatch instructions is determined by combining the shift specified in those instructions, with an additional bit from the LC register, thus effectively "shifting" the macrocode instruction when the LC register is incremented. This modification of the the shift field in byte, dispatch, and bit test jumps is enabled by a miscellaneous function field value of three.

For generality in emulating future, possibly different macro instruction sets, a feature is provided for packing four eight bit macro instructions per 32 bit word. The LC register thus holds an eight bit byte address for a macro instruction. In normal emulation of the Lisp system microcode, the hardware forces the low order bit zero and increments this register by two.

Two mechanisms trigger an increment of the LC register. First, since the main emulation loop performs a microsubroutine call to the routine which executes its instruction, the return following execution is by means of a POPJ instruction. The return address stored by the main loop on this top level call is explicitly loaded into the SPC register, and contains, in addition to the normal return address, a special tag bit which identifies this POPJ as the finish of a macro instruction execution, which in turn activates the LC increment sequence. The second way in which this sequence may be activated, is a dispatch instruction with one of the otherwise unused bits set. This feature is useful in the case where immediate information is stored as part of the macro instruction stream (for example, the extended branch offset). The dispatch path allows the LC to be incremented past these extended instructions without awkward special case programming.

In the normal case, incrementing the LC register will merely advance the low order bits of the LC, thus changing the behaviour of the specially marked byte, dispatch, and bit test jump instructions. When the LC counter overflows the end of the previously fetched macro instruction word, however, a new word must be fetched. In this case, the LC register, shifted right by two (making a word address) is automatically loaded into the VMA register, and a read cycle is initiated.

Two other actions must be performed, however. The specified read cycle may cause a page fault, so the page fault indicator must be tested - and when the data is finally available, it must be transfered from the MD into the M location holding the macro instruction word. These functions are performed with a pair of microcode instructions which immediately precede the main emulator microcode. When the incrementing of the LC as a result of the finish of a macro

instruction fails to overflow, the microprogram counter loaded from the SPC is modified by ORing a one into bit 1 of the return PC. This effectively jumps over the two instructions which, when performing a memory cycle, test for page faults and transfer the data.

Macrocode branching is vastly simplified with this hardware, because, whenever the LC register is loaded from the main data paths (which amounts to a macro code branch), a flag is set to force a new fetch the next time the LC is incremented, regardless of whether the counter overflowed.

It should be emphasized that the presence of this hardware in the machine, which is admittedly very specialized to performing a particular emulation task, does not force the programmer to make use of it; it merely provides a convenient and efficient mechanism for a common inner loop.

Micro Instruction Modification

It is often necessary to modify the actions of microinstructions as a result of data passing through the main processor data paths. An example of such a case might be the Lisp function LSH, which performs a logical shift of an integer. Here, we must fetch an integer, shift it an amount dependent on the argument of the function, and store the result. The architecture as presented so far provides only clumsy mechanisms for this operation. What we would like to do is to modify a shift instruction to specify the amount of shift, execute it, and store the results. Another example is the operation of storing and restoring the processor state around an interrupt or page fault. Here, we would like to cycle through all of the processor registers, save or restore their contents, and loop. For this to work, we need a technique for modifying the register address contained in the microinstruction.

The solution we implemented for this problem is a technique due to Fuller, first used in the PDP-11/40E. We observe that, due to the overlapped fetch of the next microinstruction with the execution of this one, just prior to clocking a new microinstruction, both the data on the output bus, and the next microinstruction to be executed are simultaneously available. If we OR this data together, then we can create a modified microinstruction, suitable for clocking into the instruction register. In effect, this allows a microinstruction to selectively modify its successor. The feature is enabled by specifying two "destinations" for the output bus data, one for bits in the high half of the microinstruction word (specifying the source and destination registers), and one for the low half of the microinstruction word (specifying the exact operation to be performed). Since the modifying instruction can by a deposit byte operation, shifting the source data to an appropriate location for doing the modificiation, we have a very effective technique for variablizing the execution of a microinstruction.

Diagnostic processor features

Another important device on the Unibus is the debugging interface for the processor. It provides an access path to the internal processor registers so that an external computer, such as a PDP-11, or a second Lisp Machine system, can assist in debugging the processor. The processor has no lights or switches interfaced, so the sole method of influencing or observing its behaviour is through this interface. The method by which the debugging processor influences that behaviour is by loading and executing microinstructions through a path independent of the normal control memory. Clock control is provided, so that the debugging processor can single step and start and stop the processor. The path by which the processor control memory is initially loaded is by execution of normal control memory write instructions loaded by the diagnostic processor.

Observational paths are considerably more common. The A, M and output busses, the program counter, and the instruction register can be directly observed, and by loading the appropriate microinstruction which references other processor registers, the contents of these registers may be observed on one of these directly accessible paths.

VII. Evaluation of Some Architectural Features

One way to evaluate a computer architecture is to measure the frequency of use of features of that architecture. Two areas in this architecture seemed unique enough to warrant a measurement of their utility. The first was the stack buffer mechanism, and its usefulness in shielding the processor from main memory requests, and the second was the shifter/masker as data manipulation element.

The stack buffer holds the top of the emulated push down stack during execution of macro coded instructions. In those cases where the depth of this stack (1024 words) is not often exceeded, the effect of this buffer is to replace a main memory data request, with its attendent overhead in memory mapping, bus delays, and awkward reference mechanisms, with a simple register access directly into or out of the main processor data paths. In this sense, it performs the same function as a cache memory, but without the probablistic element, and with considerably simpler and less expensive hardware.

Measurements were made of the use of the stack buffer in execution of the compiled Lisp evaluator, running a trivial interpreted infinite loop. In this environment, execution of a macro coded instruction takes roughly 30 microinstructions. Of the useful cycles (those whose execution is not inhibited by the transfer mechanism), 6.6% read data from the stack buffer, and 4.7% write such data. Without the stack buffer mechanism, each of these requests would take about 5 cycles, resulting in about a 56% slowdown in the average execution rate.

The stack buffer or some equivalent memory cacheing mechanism is thus an obviously worthwhile element in this type of processor design.

In contrast, about 6.6% of the processor cycles initiate main memory references, about 2.2 per macro instruction, on the average. A little more than .5 of these are due to the fetch of the macro instruction itself; the remainder are split between forwarding pointer references in the function definition, and the data being referenced. A cache mechanism would perform well on the instruction and forwarding pointer references, but would likely perform poorly on the random references to list structure (but see Clark & Green, 1976). Assuming a 75% hit rate on a cache, and an average saving of 3 cycles per cache hit, installation of a cache on this processor would improve performance by a little less than 15%. With speeds of main memory going down, the 3 cycle saving figure is generous today, and likely will continue to be reduced. In short, the addition of a cache to the processor design, though helpful, would not dramatically improve performance, and would add complexity to the design. The majority of the easily cached memory requests are already avoided in the stack buffer mechanism.

The utility of the shifter/masker as a data manipulation element was also measured, by determining what percentage of the executed microinstructions actually use the shifter. While this does not reveal how difficult execution of such code would be without such a data path, it does give an indication of its popularity with the programmers of the microcode emulator.

The raw breakdown of executed cycles into the various instruction classes is as follows:

USE	%USEFUL INSTRUCTIONS
ALU	37.8%
BYTE	18.5%
DISPATCH	20.9%
JUMP	22.8%

As can be seen from this breakdown, the combined byte and dispatch use is roughly the

same as the use of the ALU instruction, indicating the very heavy use of the combinatorial shift matrix as a data manipulation element. A small percentage of the JUMP instructions also use the shifter for bit testing. On the other hand, the large number of simple register to register transfer operations are all classified as ALU instructions. On balance, the operation of byte extraction and deposit is probably more heavily used in the processor than the truly arithmetic operations.

VIII. An Architectural Comparison

Another method for evaluating an architectural design is a comparison of it with other designs intended for similar applications. One machine, the PDP-11/40E designed at Carnegie Mellon University, has many of the same features as the Lisp Machine, and is currently being used in a similar application at Bolt Beranek and Newman. This machine is a modified version of the standard DEC PDP-11/40 processor, with the standard extended instruction set board replaced by a board containing several features which make the machine a more general purpose microprocessor.

The board contains, among other features, a 1K by 80 bit writable microcode memory, a micro program counter subroutine stack, and a 16 by 16 combinatorial shifter. These features provide a sizeable subset of the capabilities of the Lisp Machine processor, provided, however, in a sixteen bit environment, and lacking features for memory paging and larger than sixteen bit addresses.

In 1975 John Osterhout of CMU wrote a paper comparing this microprocessor design with the Standard Computer IC-9000 microprogrammable processor[Oakley, 1975]. One of the benchmarks used in his comparison was an emulation of the Data General Nova 1200 processor on both of these machines.

I coded the emulator for the Nova for the Lisp Machine processor for comparison with the emulators for these other processors. Both the Lisp Machine and PDP-11/40E microprograms are listed in appendices I and II. Both emulations occupied nearly the identical number of microcode bits, despite the larger number of instructions in the Lisp Machine version (283 versus 164). In both cases, the programs are optimized for speed of execution rather than size. The Nova instruction set was chosen for comparison because of its simplicity. Thus, many features of both processors go unused, yielding near identical performance on this task, where vast differences would exist in other applications. A simple example is the complete lack of paging and memory management aspects in the emulator design.

Yet, even in this restricted environment, the design we are using performed well. Appendix III has the execution traces for emulation of two Nova instructions, {LDA 1,DISP(PC)} and {NEG 1,2}. These traces show the heavy use of the combinatorial shifter for dispatching operations in both processor designs, and point out the features in the 11/40E design which make the processor less than optimally efficient. Primary among these is the difficulty of setting up the data which much be dispatched upon, and the difficulty of control of overflow flags and conditions in the processor.

Since the cycle times for these two processors are nearly identical, we can compare their performance on a number of cycles basis. The Lisp Machine processor used 12 cycles to execute the LDA, waiting a possible 4 for main memory response. The 11/40E performs the LDA in 16 cycles, waiting a possible 5. The difference for the NEG instruction is even larger, 16 cycles versus 24 cycles total.

The Lisp Machine emulator is even comparable in speed to the Nova 1200 computer system, taking only 3.2 microseconds for each of the sample instructions.

Another very important characteristic of any processor is the ability of a programmer to represent his algorithms in a natural way -- that is, how easily the machine can be programmed. Although this is a very subjective measurement, I programmed the emulator for the Nova in one night of fairly intensive work. The ease with which a processor may be programmed becomes increasingly important as the size and complexity of the microprograms increase. Thus, the complexity and non-trivial interactions between features in a microprocessor design such as the one for the DEC KL-10 processor effectively preclude its use for other than very simple and well

thought out modifications to the instruction set it was intended to emulate.

IX. Comparisons with Other Work

Several other attempts at developing higher level language computer systems have been made over the years, notably the early efforts at Rice on the Symbol computer system. Efforts to develop processors specifically for execution of Lisp, however, has been a fairly recent occurence, dating from the 1973 paper of Peter Deutsch [Deutsch, 1973]. Two of these efforts have reached the literature, the MBALM/1700 interpreter at Utah [Griss & Swanson, 1977], and the HP 21MX interpreter in Japan [Shimada, Yamaguchi, & Sakamura 1976]. The other efforts of which I am aware include the Alto Interlisp effort at Xerox, and the closely related PDP-11/40E Interlisp at BBN. Neither of these projects has produced published literature, though each is based on the Interlisp virtual machine concept [Moore, 1976].

All of these efforts attempt to make use of an existing microprogrammable processor to execute a macro compiled Lisp instruction set. The difficulties and limitations in these systems often arise from lack of features in these pre-defined microprogrammable environments. Perhaps the most illuminating paper from this point of view is the Japanese effort on the HP 21MX. They report not only the progress they made towards developing a computer system on the machine, but also a detailed critique of the micro architecture, listing those features which made their job difficult. Their list of desirable features is worth comparing to the features of the machine we have described. They include:

(1) Conditional Jump -- the ability to compare a field, and conditionally jump when a particular value is present in that field

(2) Multijump -- our Dispatch instruction

(3) Recursive microcode call

(4) Many registers

(5) Bit manipulation to replace single bit shifts (our Byte instructions)

(6) asynchronous memory access to avoid waiting for rewrite after reads

(7) Data Stack -- "The upper part of the stack must be put in a cache memory."
(our stack buffer)

To this list, I think, would have to be added, adequate address space for large programs, good virtual memory mapping hardware, a large control store, and a word size wide enough to contain a full address plus tag bits.

Many of these same points are made in the paper by Fuller et al. [Fuller, Lesser, Bell & Kaman 1976] concerning the requirements for general purpose emulation systems.

In view of this wide consensus concerning the features necessary for reasonable emulation of complex instruction sets, it is surprising that commercially available computer systems lack most of these features. It appears that the econmomics of the situation prevent the development of truly general purpose microprogrammable machines.

X. Summary

Computer science builds complex structures out of a hierarchy of many levels. Each of these levels transforms the data and control structures from a form easily implementable in hardware, into a more tractable form for the expression of algorithmic solutions to problems. The Lisp Machine system provides a good example of such a structure. The goal is to provide a user environment where programs may be easily expressed in a high level way, providing the illusion of infinite storage space, and user defined data structures. The implementation of this scheme requires two lower levels: the macrocode instruction set, and the microcode of the actual machine. With the development of yet more complex systems, this layering will become even more evident. Many users even today, for example, provide an intermediate layer between their programming and the basic Lisp system by using a high level language which is itself embedded in Lisp. Examples include Conniver, Planner, Scheme, and ACT1.

The introduction of microcode below the level of the basic architecture of the machine allows the macrocode instruction set to be cleanly designed, without regard to many of the ugly low level implementation details, such as page faulting, garbage collection, and process switching. Providing this level of support in the microcode of the machine is an entirely different use of the microcode programming environment than its more traditional use as a means of merely simplifying the design of the hardware.

The microcoded engine required to do this job bears a much closer resemblence to the instruction set of a conventional computer, than it does to the microcode of such machines. The key point is that no longer is the microcode of such processors designed once when the product is developed, but that it is, rather, a large complex software investment which is continuously refined. Ease of programming is a very important consideration is such circumstances. The facilities necessary in the microcode environment include adequate fast storage for temporaries, a recursive subroutining mechanism, virtual memory system support, and data manipulation primitives for decoding instructions and typed field data.

The building blocks which today's semiconductor manufacturers provide for construction of computer systems are often not those which would be desired for these purposes. Thus, the popular "bit slices" which provided a four bit wide portion of a data path for a micro machine, and the "controller chips" which provide sequencing for simple microcomputers are simply not usable for designing this style of processor. The major difficulties are the lack of access to critical internal busses, primarily a result of pinout limitations. This is especially critical for the sequencing devices, where the ability to manipulate the return addresses and program counter as registers in the main data path is crucial in a sophisticated software environment.

Trivial functions, such as the bitwise select in the masker, take several times as many devices to implement with traditional MSI than would be required with custom devices.

Thus the use of custom devices, even with today's densities would have a dramatic impact on the difficulty of implementation of processors similar to this one. We could anticipate very large reductions in the size of all portions of the processor with the exception of the register memories. In many cases, however, the design of the processor was made intentionally memory intensive, just because of the high available density in these devices. Thus, we chose to implement the memory map with a table lookup technique rather than a technology such as associative mapping, which while less logic intensive, has a much poorer representation in terms of available semiconductor devices.

With the development of good design tools for very large scale integrated circuits (VLSI), we can anticipate a time when designers of complex systems are again not dependent on

the whims of semiconductor manufacturers and the needs of volume production. Designers will take advantage of this ability to build the complex thousand-processor systems as an cohesive whole, all the way from the programming semantics to the logic gate.

Acknowledgements

This work would have been literally impossible without the cooperation, fellowship, support, and often dog-work of many other people involved in the Lisp Machine project. Richard Greenblatt was, of course, the originator of the idea for this sort of machine. In addition, he provided ideas for many features in the design, such as the virtual memory map, and the stack buffer. Jack Holloway was crucial in supporting the design aids required to build the processor, and again in providing architectural criticism. David Moon has become the expert programmer of this architecture, and has extensively influenced the design of the memory interface, divide logic, and the automatic instruction prefetch logic. Tom Callahan provided expert mechanical and electrical assembly and design.

More recently, Dan Weinreb, Richard Stallman, Alan Bawden, Bruce Edwards, and Howard Canon have taken on the job of turning a successful research effort into a useful, reliable, and reproducable hardware and software system.

Critical initial funding of this project was provided by IBM through an unrestricted grant to the Laboratory for Computer Science. Later funding was obtained from the Defense Advanced Research Projects Agency through the Artificial Intelligence Laboratory. I would like to thank Prof. Joel Moses, Prof. Patrick Winston and the sponsors, for their roles in obtaining this funding.

I would like to thank the Computer Science department of IBM Research in Yorktown for my fellowship support during the past years.

I would like to thank Prof. Marvin Minsky for his forsight in establishing an environment in which this sort of interesting research can take place, for his patience, and for a second chance.

Bibliography

[ALTO] There is no good published reference on the ALTO computer system. Perhaps the best publicly available document is "Personal Dynamic Media" by the Xerox Palo Alto Research Center Learning Research Group, Report SSL 76-1 (1976). In it, the ALTO is refered to euphemistically as an "Interim Dynabook."

Baker, H.G., List Processing in Real Time on a Serial Computer, Communications of the ACM Vol. 21 No. 4, April 1977

Bobrow, Daniel G., and Raphael, Bertram, New Programming Languages for AI Research, Xerox Palo Alto Research Center Report CSL 73-2 (1973)

Clark, D., and Green, C. C., An Empirical Study of List Structure in Lisp Communications of the ACM, Vol. 20, No. 2 February, 1977

Corbató, F. J., PL/I for Systems Programming, Datamation, May 1969

Deutsch, L. Peter, A Lisp Machine with Very Compact Programs, Proceedings, Third International Joint Conference on Artificial Intelligence 1973

Feustel, Edward A., On the Advantages of Tagged Architectures, IEEE Transactions on Computers, Vol. C-22 No. 7, July 1973

Fuller, Samuel H., Lesser, Victor R., Bell, C. Gordon, and Kaman, Charles H., The Effects of Emerging Technology and Emulation Requirements on Microprogramming, IEEE Transactions on Computers, Vol. C-25 No. 10, October, 1976

Griss, M.L., and Swanson, M.R., MBALM/1700: A Microprogrammed Lisp Machine for the Burroughs B1726, University of Utah report UCP-55, April 1977

Greenblatt, Richard D., The Lisp Machine, M.I.T. A.I. Laboratory Working Paper No. 79, November, 1974

Hewitt, Carl E., and Smith, Brian, Towards a Programming Apprentice, IEEE Transactions on Software Engineering, Vol. SE-1 No. 1, March 1975

Horn, Berthold, and Winston, Patrick H., Personal Computers, Datamation, May, 1975

Knight, Thomas F., CONS, M.I.T. A.I. Laboratory Working Paper No. 80 November, 1974

Knight, Thomas F., Moon, David A., and Steele, Guy L., CADR, M.I.T. Artificial Intelligence Laboratory Working Paper, to appear.

Learning Research Group, Xerox Palo Alto Research Center, Personal Dynamic Media March, 1976

Lonergan, W. and King, P., Design of the B5000 System, Datamation vol. 7 no. 5
May, 1961

McCarthy, J., et al., Lisp 1.5 Programmer's Manual, M.I.T. Press, Cambridge, Mass. 1962

McCarthy, J., History of Lisp, Sigplan Notices, Vol. 13 No. 8, August 1978

Metcalfe, R. M., and Boggs, D. R., Ethernet: Distributed Packet Switching for Local Computer Networks, Xerox Palo Alto Research Center report CSL 75-7 (1975)

Moon, David A., The MacLisp Reference Manual, M.I.T. Laboratory for Computer Science

Moore, J. Strother, The Interlisp Virtual Machine Specification, Xerox Palo Alto Research Center, Report CSL-76-5, September, 1976

Oakley, John D., A Comparison of Two Microprogrammable Processors: PDP-11/40E and MLP-900, Carnegie Mellon University Technical Report, May, 1975

Rich, C. and Shrobe, H.E., Initial Report on a Lisp Programmer's Apprentice, M.I.T. Artificial Intelligence Laboratory Technical Report AI-TR-354 December, 1976

Teitelman, Warren, et al., The Interlisp Reference Manual, Xerox Palo Alto Research Center October, 1978

Sandevall, Erik, Programming in the Lisp Environment: The Lisp Experience LiTH-MAT-76-9, Informatics Laboratory, Linkoeping University, S-58159 Linkoeping, Sweden 1976

Shimada, T., Yamaguchi, Y., and Sakamura, K., A Lisp Machine and Its Evaluation, Denshi Tsushin Gakkai Ronbunshi, Vol. 59-D, No. 6, June 1976 pp. 406-413 Translated in Systems-Computers-Controls Vol. 7, No. 3, 1976 pp. 59-65

Weinreb Daniel L., and Moon, David A., The Lisp Machine Manual, M.I.T. Artificial Intelligence Laboratory, 1978

Winograd, Terry, Breaking the Complexity Barrier (Again), ACM Sigplan Notices 10:1 January 1975

Winograd, Terry, The Reactive Engine Paper, The CoEvolution Quarterly, Fall 1975

Winston, Patrick H., Artificial Intelligence, Addison-Wesley, Reading, Massacusetts 1977

Appendix I

Nova Simulator in CADR Microcode

```
;Simulation program for the Data General Nova

(def-data-field word 16. 0)

;bit definitions for memory reference instructions

(def-next-field ir-displacement 8 m-ir)
(def-next-field ir-index 2 m-ir)
(def-next-field ir-indirect 1 m-ir)
(def-next-field ir-function 2 m-ir)
(def-next-field ir-op 3 m-ir)
(reset-bit-pointer m-ir)

(def-bit-field-in-reg ir-fun-ind 3 10.)
(def-bit-field-in-reg displacement-sign 1 7 m-ir)

;bit definitions for operate class instructions

(def-next-field ir-skip 3 m-ir)
(def-next-field ir-noload 1 m-ir)
(def-next-field ir-carry 2 m-ir)
(def-next-field ir-shift 2 m-ir)
(def-next-field ir-operfn 3 m-ir)
(def-next-field ir-dest 2 m-ir)
(def-next-field ir-source 2 m-ir)
(reset-bit-pointer m-ir)

(def-data-field address 15. 0)
(def-data-field adr-indirect 1 15.)

(def-data-field auto-index-field 2 3)

(locality d-mem)

(start-dispatch 3 0)
op-dispatch-table
          (op000)              ;isz/dsz/jmp/jsr
          (op001)              ;lda
          (op010)              ;sta
          (op011)              ;i/o instructions- not emulated
          (op100)              ;dispatch different places for the 2 bit source
          (op101)
          (op110)
          (op111)
(end-dispatch)

(start-dispatch 2 0)
index-dispatch-table
          (p-bit r-bit)                 ;fall through if page zero reference
          (p-bit x01)                   ;displaced off of pc
          (p-bit x10)                   ;displaced off of ac2
          (p-bit x11)                   ;displaced off of ac3
(end-dispatch)

(start-dispatch 1 0)
disp-sign-dispatch-table
          (r-bit)
          (p-bit r-bit)
(end-dispatch)

(start-dispatch 3 0)
function-indirect-dispatch-table
          (jmp)                         ;JMP
          (p-bit r-bit)                 ;JMP I
          (jsr)                         ;JSR
          (p-bit r-bit)                 ;JSR I
          (isz)                         ;ISZ
          (p-bit r-bit)                 ;ISZ I
```

```
        (dsz)                           ;DSZ
        (p-bit r-bit)                   ;DSZ I
(end-dispatch)

(start-dispatch 2 0)
function-dispatch-table
        (jmp)                           ;JMP
        (jsr)                           ;JSR
        (isz)
        (dsz)
(end-dispatch)

(start-dispatch 2 0)
auto-index-dispatch-table
        (r-bit inhibit-xct-next)                ;00000 to 00007
        (r-bit inhibit-xct-next)                ;00010 to 00017
        (p-bit r-bit)                   ;auto increment
        (auto-decrement inhibit-xct-next)
(end-dispatch)

(start-dispatch 3 0)
lda-indirect-dispatch-table
        (lda0)
        (p-bit r-bit)
        (lda1)
        (p-bit r-bit)
        (lda2)
        (p-bit r-bit)
        (lda3)
        (p-bit r-bit)
(end-dispatch)

(start-dispatch 2 0)
lda-dispatch-table
        (lda0)
        (lda1)
        (lda2)
        (lda3)
(end-dispatch)

(start-dispatch 3 0)
sta-indirect-dispatch-table
        (sta0)
        (op010i inhibit-xct-next)
        (sta1)
        (op010i inhibit-xct-next)
        (sta2)
        (op010i inhibit-xct-next)
        (sta3)
        (op010i inhibit-xct-next)
(end-dispatch)

(start-dispatch 2 0)
sta-dispatch-table
        (sta0)
        (sta1)
        (sta2)
        (sta3)
(end-dispatch)

(locality i-mem)

;;main loop - enter with valid new instruction present or on the way
;;into read-memory-data register

mloop   (dispatch-xct-next mdr-op op-dispatch-table)    ;dispatch on op field
       ((m-ir) read-memory-data)                        ;copy new instruction to IR
                                                        ;really dispatching
```

```
;; OPCODE 0 ISZ/DSZ/JMP/JSR instructions

op000     (dispatch ir-index index-dispatch-table)                      ;x dispatch-return w/m-disp
         ((m-disp) ir-displacement)                                     ;get displacement before
          (dispatch-xct-next ir-fun-ind function-indirect-dispatch-table)
op000i ((vma-start-read m-address) address m-disp)                      ;start memory fetch
                                                                        ;here if not indirect
          (call-less-than (a-constant 40) m-address auto-index)
          (jump-bit-set-xct-next adr-indirect read-memory-data op000i)
          ((m-disp) read-memory-data)
          (dispatch-xct-next ir-function function-dispatch-table)
         ((vma-start-read) address m-disp)


x01       (dispatch ir-displacement-sign disp-sign-dispatch-table)
         (add (m-disp) m-disp m-pc)                 ;popjs after this instruction for pos case
          (popj-after-next (m-disp) (a-constant -1) ir-displacement)
          (add (m-disp) m-disp m-pc)                ;popj here for negative disp case

x10       (dispatch ir-displacement-sign disp-sign-dispatch-table)
         (add (m-disp) m-disp m-ac2)                ;popjs after this instruction for pos case
          (popj-after-next (m-disp) (a-constant -1) ir-displacement)
          (add (m-disp) m-disp ac2)                 ;popj here for negative disp case

x11       (dispatch ir-displacement-sign disp-sign-dispatch-table)
         (add (m-disp) m-disp m-ac3)                ;popjs after this instruction for pos case
          (popj-after-next (m-disp) (a-constant -1) ir-displacement)
          (add (m-disp) m-disp ac3)                 ;popj here for negative disp case

auto-index
          (dispatch auto-index-field m-address auto-index-dispatch-table)
          (popj-after-next M+1 (write-memory-data-start-write) read-memory-data)
          ((vma-start-read) vma)

auto-decrement
          (popj-after-next M-A-1 (write-memory-data-start-write) read-memory-data)
          ((vma-start-read) vma)


isz       ((write-memory-data-start-write m-data) M+1 read-memory-data)
          ((m-data) word m-data)
          (jump-equal m-data (a-constant 0) skpret)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

dsz       ((write-memory-data-start-write m-data) M-A-1 read-memory-data)
          ((m-data word m-data)
          (jump-equal m-data (a-constant 0) skpret)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

skpret    ((m-pc vma-start-read) ADD m-pc (a-constant 2))
          (jump mloop)


jsr       ((m-ac3) M+1 m-pc)
jmp       (jump-xct-next mloop)
         ((m-pc) address m-disp)


op001     (dispatch ir-index index-dispatch-table)
         ((m-disp) ir-displacement)                                     ;get displacement before
          (dispatch-xct-next ir-fun-ind lda-indirect-dispatch-table)
op001i ((vma-start-read m-address) address m-disp)
          (call-less-than (a-constant 40) m-address auto-index)
          (jump-bit-set-xct-next adr-indirect read-memory-data op001i)
          ((m-disp) read-memory-data)
          (dispatch-xct-next ir-function lda-dispatch-table)
         ((vma-start-read) address m-disp)
```

```
lda0      ((m-ac0) read-memory-data)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

lda1      ((m-ac1) read-memory-data)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

lda2      ((m-ac2) read-memory-data)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

lda3      ((m-ac3) read-memory-data)
          ((m-pc vma-start-read) M+1 m-pc)
          (jump mloop)

op010      (dispatch ir-index index-dispatch-table)
          ((m-disp) ir-displacement)                                ;get displacement before
           (dispatch ir-fun-ind sta-indirect-dispatch-table)
          ((vma m-address) address m-disp)

op010i     ((vma-start-read m-address) address m-disp)
           (call-less-than (a-constant 40) m-address auto-index)
           (jump-bit-set-xct-next adr-indirect read-memory-data op010i)
          ((m-disp) read-memory-data)
           (dispatch ir-function sta-dispatch-table)
          ((vma m-address) address m-disp)


sta0       ((write-memory-data-start-write) word m-ac0)
           ((vma-start-read m-pc) M+1 m-pc)
           (jump mloop)

sta1       ((write-memory-data-start-write) word m-ac1)
           ((vma-start-read m-pc) M+1 m-pc)
           (jump mloop)

sta2       ((write-memory-data-start-write) word m-ac2)
           ((vma-start-read m-pc) M+1 m-pc)
           (jump mloop)

sta3       ((write-memory-data-start-write) word m-ac3)
           ((vma-start-read m-pc) M+1 m-pc)
           (jump mloop)


op100      (dispatch-xct-next ir-dest-funct dest-funct-dispatch-table)
          ((a-source) word ac0)

op101      (dispatch-xct-next ir-dest-funct dest-funct-dispatch-table)
          ((a-source) word ac1)

op110      (dispatch-xct-next ir-dest-funct dest-funct-dispatch-table)
          ((a-source) word ac2)

op111      (dispatch-xct-next ir-dest-funct dest-funct-dispatch-table)
          ((a-source) word ac3)


com        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
          ((m-result) ANDCA (m-constant 177777) a-source)

neg        ((m-result) ANDCA (m-constant 177777) a-source)
           (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
          ((m-result) M+1 m-result)

mov        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
          ((m-result) AND (m-constant 177777) a-source)
```

```
inc     (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) M+A+1 a-source)

adc0    ((m-result) word m-ac0)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

adc1    ((m-result) word m-ac1)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

adc2    ((m-result) word m-ac2)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

adc3    ((m-result) word m-ac3)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

sub0    ((m-result) word m-ac0)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) M+A+1 a-source m-result)

sub1    ((m-result) word m-ac1)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) M+A+1 a-source m-result)

sub2    ((m-result) word m-ac2)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) M+A+1 a-source m-result)

sub3    ((m-result) word m-ac3)
        ((a-source) ANDCA (m-constant 177777) a-source)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) M+A+1 a-source m-result)

add0    ((m-result) word m-ac0)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

add1    ((m-result) word m-ac1)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

add2    ((m-result) word m-ac2)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)

add3    ((m-result) word m-ac3)
        (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) ADD a-source m-result)


and0    (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) AND a-source m-ac0)

and1    (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) AND a-source m-ac1)

and2    (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
        ((m-result) AND a-source m-ac2)
```

```
and3    (dispatch-xct-next ir-shift-carry shift-carry-dispatch-table)
       ((m-result) AND a-source m-ac3)

sc0     (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) XOR m-result a-carryflag)

sc1     (dispatch ir-nl-skip nl-skip-alt-dispatch-table)

sc2     (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) XOR m-result (a-constant 200000))

sc3     (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) EQV m-result a-carryflag)

sc4     ((m-result) XOR m-result a-carryflag)
        ((m-result) DPB m-result (byte 17. 1))
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 1 17.) m-result)

sc5     ((m-result) DPB m-result (byte 17. 1))
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 1 17.) m-result)

sc6     ((m-result) XOR m-result (a-constant 200000))
        ((m-result) DPB m-result (byte 17. 1))
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 1 17.) m-result)

sc7     ((m-result) EQV m-result a-carryflag)
        ((m-result) DPB m-result (byte 17. 1))
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 1 17.) m-result)

sc10    ((m-result) XOR m-result a-carryflag)
        ((m-result) DPB m-result (byte 1 17.) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 17. 1) m-result)


sc11    ((m-result) DPB m-result (byte 17. 1))
        ((m-result) DPB m-result (byte 1 17.) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 17. 1) m-result)

sc12    ((m-result) XOR m-result (a-constant 200000))
        ((m-result) DPB m-result (byte 1 17.) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 17. 1) m-result)

sc13    ((m-result) EQV m-result a-carryflag)
        ((m-result) DPB m-result (byte 1 17.) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) LDB m-result (byte 17. 1) m-result)

sc14    ((m-result) XOR m-result a-carryflag)
        ((m-temp) m-result)
        ((m-result) LDB m-result (byte 8 8) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) DPB m-temp (byte 8 8) m-result)

sc15    ((m-result) DPB m-result (byte 17. 1))
        ((m-temp) m-result)
        ((m-result) LDB m-result (byte 8 8) m-result)
        (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
       ((m-result) DPB m-temp (byte 8 8) m-result)

sc16    ((m-result) XOR m-result (a-constant 200000))
        ((m-temp) m-result)
```

```
            ((m-result) LDB m-result (byte 8 8) m-result)
            (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
           ((m-result) DPB m-temp (byte 8 8) m-result)

scl7        ((m-result) EQV m-result a-carryflag)
            ((m-temp) m-result)
            ((m-result) LDB m-result (byte 8 8) m-result)
            (dispatch-xct-next ir-nl-skip nl-skip-dispatch-table)
           ((m-result) DPB m-temp (byte 8 8) m-result)

nls0        ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
nls0b       (dispatch-xct-next ir-dest dest-dispatch-table)
nls0a   ((vma-start-read m-pc) M+1 m-pc)

nls1        ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls2        (jump-bit-set-xct-next m-result (byte 1 16.) nls0b)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls3        (jump-bit-clear-xct-next m-result (byte 1 16.) nls0b)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls4        ((m-temp) LDB m-result (byte 8 16.))
            (jump-not-equal-xct-next m-temp nls0b)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls5        ((m-temp) LDB m-result (byte 8 16.))
            (jump-equal-xct-next m-temp nls0b)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls6        (jump-bit-clear-xct-next m-result (byte 1 16.) nls6a)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            ((m-temp) LDB m-result (byte 8 16.))
            (jump-not-equal-xct-next m-temp nls0a)
nls6a   (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))

nls7        (jump-bit-set-xct-next m-result (byte 1 16.) nls0a)
           ((a-carryflag) SELECTIVE-DEPOSIT m-result (byte 1 16.))
            ((m-temp) LDB m-result (byte 8 16.))
            (jump-equal-xct-next m-temp nls0a)
           (dispatch-xct-next ir-dest dest-dispatch-table)
           ((vma-start-read m-pc) ADD m-pc (a-constant 2))


nls10       ((vma-start-read m-pc) M+1 m-pc)
            (jump mloop)

nls11       ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
            (jump mloop)

nls12       (jump-bit-set m-result (byte 1 16.) nls10)
            ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
            (jump mloop)

nls13       (jump-bit-clear m-result (byte 1 16.) nls10)
            ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
            (jump mloop)
```

```
nls14    ((m-temp) LDB m-result (byte 0 16.))
         (jump-not-equal m-temp nls10)
         ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
         (jump mloop)

nls15    ((m-temp) LDB m-result (byte 0 16.))
         (jump-equal m-temp nls10)
         ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
         (jump mloop)

nls16    (jump-bit-clear m-result (byte 1 16.) nls16a)
         ((m-temp) LDB m-result (byte 0 16.))
         (jump-not-equal m-temp nls10)
nls16a   ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
         (jump mloop)

nls17    (jump-bit-set m-result (byte 1 16.) nls10)
         ((m-temp) LDB m-result (byte 0 16.))
         (jump-equal m-temp nls10)
         ((vma-start-read m-pc) ADD (a-constant 2) m-pc)
         (jump mloop)

dest0     (jump-xct-next mloop)
        ((m-ac0) word m-result)

dest1     (jump-xct-next mloop)
        ((m-ac1) word m-result)

dest2     (jump-xct-next mloop)
        ((m-ac2) word m-result)

dest3     (jump-xct-next mloop)
        ((m-ac3) word m-result)
```

Appendix II

Nova Simulator in PDP-11/E microcode

This appendix contains the Nova simulator written for the 11/40E by Paul Drongowski. It is reproduced here with his kind permission.

```
! SUPERNOVA EMULATOR
! PDP-11/40 MICROCODE


! AUTHOR: PAUL J. DRONGOWSKI
!          COMPUTER SCIENCE DEPARTMENT
!          CARNEGIE-MELLON UNIVERSITY
!          PITTSBURGH, PENNSYLVANIA 15213
! DATE:    APRIL 19, 1975

! VERSION 5.0 - PRODUCT
!                    (1) TELETYPE I/O
!                    (2) PAPER TAPE READER AND PUNCH (HIGH SPEED)
!                    (3) BASIC CPU CONTROL OPTIONS


! REGISTER DECLARATIONS

AC0:=R[0]$                    ! NOVA GENERAL REGISTER SET
AC1:=R[1]$
AC2:=R[2]$
AC3:=R[3]$

I:=R[4]$                      ! INSTRUCTION REGISTER
PC:=R[5]$                     ! PROGRAM COUNTER
T:=R[6]$                      ! TEMPORARY STORAGE
U:=R[7]$                      ! TEMP STORAGE - CONTENTS ALTERABLE BY SUBR
SRC:=R[10]$                   ! SOURCE REGISTER FOR ARITH-LOGIC INSTR
DST:=R[11]$                   ! DESTINATION REGISTER FOR ARITH-LOGIC INSTR
CBIT:=R[12]$                  ! CARRY BIT - R[12]=0 OR R[12]=1 ONLY
ADRSC:=R[13]$                 ! LAST CONSOLE ADDRESS
PCINTR:=R[14]$                ! REGISTER TO MAINTAIN PROCESSOR INTERRUPT STAT
PCMASK:=R[15]$                ! CURRENT INTERRUPT MASK FOR DEVICE ENABLE/DIS
DVINTR:=R[16]$                ! INTERRUPT DEVICE CODE
! MACRO DEFINITIONS


SERVICE:=P3;  SP←0;  CP=1;  D←0;  GOTO FETCH$
EXECIO:=SERVIC$

ONE:=C[1]$
TWO:=C[2]$
CCMASK:=C[12]$

P1:=CLK=2$
P2:=CLK=4$
P3:=CLK=6$
DISPLAY:=SDM=2$
CASE:=EUBC←$
DATI:=BUS=1$
DATIP:=BUS=3$
DATO:=BUS=5$
DATOB:=BUS=7$
AWBBY:=BUS=2$
PERIF:=BUS=6$
N.Z.V.C:=CLK=6; SPS=3$
N.Z.V:=CLK=6; SPS=2$
CLK.C:=CLK=6; SPS=1$
NOP:=NOOP$
PUSH:=S←$
RETURN:=EUBC←$
GOTO:=XUPF=$

! BUT MNEMONIC DEFINITIONS

BUT:=UBF$
REQUESTS:=26$
BR.WAIT.FETCH:=25$
INTR:=7$
```

```
SWITCH:=6$
D.EQL.0:=12$
HALT:=10$
REG.EXM:=4; P3$
REG.DEP:=3; P3$
VAR.SWTCH:=30$
B.BEGIN:=5$
CBR.HALT:=24$
RESET:=2$


SERVIC:             P3;  SP←0;  CP=1;  D←0;  GOTO FETCH


! FETCH/EXECUTE

FETCH:
                    BA←PC;D←PC+2; DATI; CLKOFF             ! FETCH NEXT
                                                                ! INSTRUCTION
FETCH1:             S,I←UNIBUS                             ! STORE INSTRUCTION
                    PC←D; CASE TOS<15:11>                  ! STORE NEW PC
                    BA←TOS<12:11>                 ! GET AC DEST

                    SET
                    PUSH EXJMP; GOTO ECALC                            ! 00  JUMP
                    D←PC; GOTO EXJSR                                  ! 01  JSR
                    PUSH EXISZ; GOTO ECALC                            ! 02  ISZ
                    PUSH EXDSZ; GOTO ECALC                            ! 03  DSZ
                    PUSH EXLDA; GOTO ECALC                            ! 04  LDA AC0
                    PUSH EXLDA; GOTO ECALC                            ! 05  LDA AC1
                    PUSH EXLDA; GOTO ECALC                            ! 06  LDA AC2
                    PUSH EXLDA; GOTO ECALC                            ! 07  LDA AC3
                    D←AC0; GOTO EXSTA                                 ! 10  STA AC0
                    D←AC1; GOTO EXSTA                                 ! 11  STA AC1
                    D←AC2; GOTO EXSTA                                 ! 12  STA AC2
                    D←AC3; GOTO EXSTA                                 ! 13  STA AC3
HLT:                GOTO EXECIO                                       ! 14  IO
                    GOTO EXECIO                                       ! 15  IO
                    GOTO EXECIO                                       ! 16  IO
                    GOTO EXECIO                                       ! 17  IO
                    D←AC0; CASE TOS<5:4>; GOTO EXECAL                 ! 20  AL INST
                    D←AC0; CASE TOS<5:4>; GOTO EXECAL                 ! 21  AL INST
                    D←AC0; CASE TOS<5:4>; GOTO EXECAL                 ! 22  AL INST
                    D←AC0; CASE TOS<5:4>; GOTO EXECAL                 ! 23  AL INST
                    D←AC1; CASE TOS<5:4>; GOTO EXECAL                 ! 24  AL INST
                    D←AC1; CASE TOS<5:4>; GOTO EXECAL                 ! 25  AL INST
                    D←AC1; CASE TOS<5:4>; GOTO EXECAL                 ! 26  AL INST
                    D←AC1; CASE TOS<5:4>; GOTO EXECAL                 ! 27  AL INST
                    D←AC2; CASE TOS<5:4>; GOTO EXECAL                 ! 30  AL INST
                    D←AC2; CASE TOS<5:4>; GOTO EXECAL                 ! 31  AL INST
                    D←AC2; CASE TOS<5:4>; GOTO EXECAL                 ! 32  AL INST
                    D←AC2; CASE TOS<5:4>; GOTO EXECAL                 ! 33  AL INST
                    D←AC3; CASE TOS<5:4>; GOTO EXECAL                 ! 34  AL INST
                    D←AC3; CASE TOS<5:4>; GOTO EXECAL                 ! 35  AL INST
                    D←AC3; CASE TOS<5:4>; GOTO EXECAL                 ! 36  AL INST
                    D←AC3; CASE TOS<5:4>; GOTO EXECAL                 ! 37  AL INST
                    TES


! ENTRY FOR NOVA START UP

!                   .=2000
!ENT00:             SP←0
!ENT01:             D,BA←0; S,I←D
!ENT02:             PC←D
!ENT03:             PS←D
!ENT03A:                      CBIT←D
!ENT03B:                      D←177777; PCMASK←D
!ENT03C:                      P3; IR←060000;        ! TO MAKE COND
```

```
!ENT04:    GOTO SERVIC                                       ! CODES WORK

! ARITHMETIC LOGIC INSTR

EXECAL:              SRC←D                ! STORE SOURCE OPERAND IN SRC
                     ! FORM BASE VALUE FOR CARRY BIT

                     SET
                       D←CBIT;                                    ! CB←C
                       D←0;  T←D;  CASE TOS<10:8>;  GOTO EXECAL1
                                                                  ! CB←0
                       D←1;  T←D;  CASE TOS<10:8>;  GOTO EXECAL1
                                                                  ! CB←1
                       D←CBIT XOR 1;                      ! CB←NOT C
                     TES

                     T←D;  CASE TOS<10:8>
EXECAL1:             B←R[BA]                    ! GET DEST OPERAND

                     SET
                       D←NOT SRC;  SRC←D;  N.Z.V.C                ! COM

                       START
                         D←NOT SRC; B←D;                          ! NEG
                         D←1+B;  SRC←D;  N.Z.V.C
                       END

                       D←SRC;      N.Z.V.C;                       ! MOV
                       D←SRC+1;  SRC←D;  N.Z.V.C;                 ! INC

                       START
                         D←NOT SRC; SRC←D;                        ! ADC
                         D←SRC+B;  SRC←D;  N.Z.V.C
                       END

                       START                                      ! SUB
                         D←NOT SRC;  SRC←D;
                         D←SRC+B+1;  SRC←D;  N.Z.V.C;
                       END
                       D←SRC+B;  SRC←D;  N.Z.V.C;                 ! ADD
                       D←SRC AND B;  SRC←D;  N.Z.V.C              ! AND
                     TES

! SHIFT - LOAD  ROUTINES

SHIFT1:
                     B←PS;  CASE TOS<7:6>;
                     P3;  D←T XOR B;  PS←D;

                     SET
                       D←SRC;  CASE TOS<3:3>                      ! NO SHIFT

                       START
                         D←PS AND ONE;  B←D;             ! LEFT ROTATE
                         D←SRC+SRC;  SRC←D;  N.Z.V.C;  CASE TOS<3:3>
                         D←SRC OR B;  GOTO NOLOD1;       ! OR IN CARRY BIT
                       END

                       START
                         D←SRC;  SCOM=2;  SRC←D/2;       ! RIGHT ROTATE
                         T←D;
                         D←T AND 1;  T←D;
                         D←(T OR NOT ONE) + 1;  N.Z.V.C;  CASE TOS<3:3>
                                                         ! GENERATE CARRY IF T=1
                         D←SRC;  N.Z.V;  GOTO NOLOD1;
                       END
```

```
                  START
                    B←SRC;  CASE TOS<3:3>                 ! SWAP BYTES
                    D←B.LH;  N.Z.V;  GOTO NOLOD1;
                  END
                TES

                NOOP
                SET                                        ! STORE RESULT
NOLOD1:           R[BA]←D                                  ! NOLOAD, TEST
                  D←PS; CASE S<2:0>; GOTO NOLSKP1
                TES

                D←PS AND ONE;
                CBIT←D                            ! GET CARRY BIT
                D←PS; CASE S<2:0>                 ! PUSH STATUS FOR SKIP
NOLSKP1:        S←D;

! SKIP ROUTINE - ARITHMETIC LOGIC GROUP

                SET
                  SERVICE                         ! NEVER SKIP
                  D←PC+2; PC←D; GOTO SERVIC       ! ALWAYS SKIP
                  CASE S<0:0>; GOTO NOTSKP        ! SKIP IF C=0
                  CASE S<0:0>; GOTO POSSKP        ! SKIP IF C=1
                  CASE S<2:2>; GOTO POSSKP        ! SKIP IF Z=1
                  CASE S<2:2>; GOTO NOTSKP        ! SKIP IF Z=0
                  CASE S<2:0>; GOTO ORSKIP        ! SKIP IF C=0  OR Z=1
                  CASE S<2:0>; GOTO ANDSKP        ! SKIP IF C=1 AND Z=0
                TES

POSSKP:         NOOP
                SET
                  SERVICE
                  D←PC+2; PC←D; GOTO SERVIC
                TES

NOTSKP:         NOOP
                SET
                  D←PC+2; PC←D; GOTO SERVIC
                  SERVICE
                TES

ANDSKP:         NOOP
                SET
                  SERVICE
                  D←PC+2; PC←D; GOTO SERVIC
                  SERVICE
                  D←PC+2; PC←D; GOTO SERVIC
                  SERVICE
                  SERVICE
                  SERVICE
                  SERVICE
                TES

ORSKIP:         NOOP
                SET
                  D←PC+2; PC←D; GOTO SERVIC
                  SERVICE
                  D←PC+2; PC←D; GOTO SERVIC
                  SERVICE
                  D←PC+2; PC←D; GOTO SERVIC
                  D←PC+2; PC←D; GOTO SERVIC
                  D←PC+2; PC←D; GOTO SERVIC
                  D←PC+2; PC←D; GOTO SERVIC
                TES

! EFFECTIVE ADDRESS CALCULATION
```

```
! RETURN EFFECTIVE ADDRESS IN BA AND D REGISTERS. CONTENTS OF B AND R
! REGISTERS ARE MODIFIED. PROPER EXIT IS THROUGH RETURN S.

ECALC:          S←I        CASE TOS<10:8>     ! TEST INDRT AND INDEX FIELDS
                B←S<7:0>                      ! B REG GETS THE DISPLACEMENT

                SET
                  D←B; GOTO ADRABS

                  START   ! PC RELATIVE ADDRESSING (ABSOLUTE)
                    D←PC-2;  SCOM=3;
                    U←D/2
                    D←U+B.EL; U,B←D; RETURN S; GOTO ECALC1
                  END

                  D←AC2+B.EL; GOTO ADRABS
                  D←AC3+B.EL; GOTO ADRABS
                  D←B; GOTO ADRIND

                  START   ! PC RELATIVE ADDRESSING (INDIRECT)
                    D←PC-2;  SCOM=3;
                    U←D/2
                    D←U+B.EL; S,U,B←D; GOTO ECALC2
                  END

                  D←AC2+B.EL; GOTO ADRIND
                  D←AC3+B.EL; GOTO ADRIND
                TES

ADRABS:         U,B←D; RETURN S               ! GET WORD ADDRESS
ECALC1:         BA,D←U+B; GOTO 0              ! TO MAKE RETURN S WORK RIGHT


! EFFECTIVE ADDRESS CALCULATION (2)

ADRIND:         S,U,B←D                       ! PUSH AND SAVE INDRCT ADDRS
ECALC2:         P3; D←TOS<15:5>;                        ! TEST FOR AUTO INC/DEC
                BUT=D.EQL.0
                BA,D←U+B; CASE S<4:3>         ! NOP FOR BUT, MAKE WORD ADDR

                SET
                  DATI; CLKOFF; GOTO ECALCC             ! NO AUTO INC/DEC
                  NOOP                                  ! NOOP FOR CASE
                TES

                SET
                  DATI; CLKOFF; GOTO ECALCC             ! NO AUTO, GET ADDRESS
                  DATI; CLKOFF; GOTO ECALCC             ! NO AUTO, GET ADDRESS
                  DATIP; CLKOFF; GOTO ECALC6            ! AUTO INCREMENT
                  DATIP; CLKOFF; GOTO ECALC7            ! AUTO DECREMENT
                TES

ECALC6:         T←UNIBUS
                D←T+1; S←D; GOTO ECALC8                 ! INC THEN WRITE LOC
ECALC7:         T←UNIBUS
                D←T-1; S←D; GOTO ECALC8                 ! DEC THEN WRITE LOC

ECALC8:         DATO; CLKOFF; CASE S<15:15>             ! TEST INDRCT BIT, WRITE
                U,B←D
                SET
ECALC9:           RETURN S; GOTO ECALC1       ! GOT EFFECTIVE ADDR, RETURN
                   S,U,B←D; GOTO ECALC2       ! PERFORM NEXT LEVEL IF INDRCT
                TES

                SET
ECALCC:           S,U,B←UNIBUS                ! GET POTENTIAL EFFECTIVE ADDR
                   S,U,B←UNIBUS
                   S,U,B←UNIBUS
```

```
                   S,U,B←UNIBUS
                TES
                D←B; CASE S<15:15>          ! TEST INDIRECT BIT
                GOTO ECALC9                 ! NOP FOR CASE

! EXECUTE JUMP.MODIFY AND MOVE.DATA INSTRUCTIONS

EXLDA:          DATI; CLKOFF       ! GET MEMORY OPERAND
                P2; BA←S<12:11>
                R[BA]←UNIBUS; GOTO SERVIC              ! STORE MEM OP

EXSTA:          T←D; PUSH EXSTA1; GOTO ECALC
EXSTA1:         D←T; DATO; CLKOFF; GOTO SERVIC          ! WRITE OPERAND

EXISZ:          DATIP; CLKOFF      ! GET MEM OPERAND, PAUSE
                T←UNIBUS                               ! MOVE OPERAND TO TEMP
                D←T+1; DATO; CLKOFF

TSTZER:         BUT=D.EQL.0                            ! TEST FOR ZERO RESULT
                NOOP
                SET
                   GOTO SERVIC                         ! NOT ZERO - NO SKIP
                   D←PC+2; PC←D; GOTO SERVIC  ! ZERO - SKIP NEXT INS
                TES

EXDSZ:          DATIP; CLKOFF      ! GET MEM OPERAND
                T←UNIBUS                               ! MOVE OPERAND TO TEMP
                D←T-1; DATO; CLKOFF; GOTO TSTZER

EXJMP:          PC←D;GOTO SERVIC

EXJSR:          SRC←D/2;PUSH EXJSR1;GOTO ECALC
EXJSR1:         PC←D
                D←SRC
                AC3←D;GOTO SERVIC

                FINIS
```

Appendix III

Comparative Execution Trace of Nova Simulations

This appendix shows comparative timings of execution of two Nova instructions on the 11/40E processor and the CADR processor, on a cycle by cycle basis, and demonstrates a few a features found in both processor designs.

SAMPLE MICROCODE SEQUENCES FOR SELECTED NOVA INSTRUCTIONS

```
Key:        -----                   Wasted cycle
            (wait)                  Possible delay due to main memory wait
```

CADR Processor Microcode Sequences

```
LDA 1,DISP(PC)                     ;LOAD ACCUMULATOR 1 FROM MEMORY
;                                  ADDRESSED BY DISPLACEMENT OFF PC

MLOOP     (wait)DISPATCH OPCODE FIELD
          READ-MEMORY-DATA → IR                                          **
OP001     DISPATCH INDEX FIELD
          DISPLACEMENT → DISP                                            **
X01       DISPATCH DISPLACEMENT SIGN
          DISP + PC → DISP (POPJ)                                        **
          DISPATCH FUNCTION+INDIRECT BIT FIELD
OP001I    DISP ∧ 77777 → MEMORY ADDRESS (START READ) → ADDRESS           **
LDA1      (wait)READ-MEMORY-DATA → AC1
          PC + 1 → MEMORY ADDRESS (START READ)
          JUMP MLOOP
          -----


NEG 1,2

MLOOP     (wait)DISPATCH OPCODE FIELD
          READ-MEMORY-DATA → IR                                          **
OP101     DISPATCH DESTINATION/FUNCTION FIELD
          AC1 → SOURCE                                                   **
NEG       ¬ SOURCE ∧ 177777 → RESULT
          DISPATCH SHIFT/CARRY
          RESULT + 1 → RESULT                                            **
SC0       DISPATCH NO LOAD/SKIP
          RESULT XOR CARRYFLAG → RESULT                                  **
NLS0      RESULT SELECTIVE DEPOSIT 0 → CARRYFLAG
          DISPATCH DESTINATION FIELD
          PC + 1 → MEMORY ADDRESS (START READ)                           **
DEST2     JUMP MLOOP
          RESULT DPB 0 → AC2                                             **
```

11/40E Microcode Sequences

```
LDA 1,DISP(PC)

SERVICE   0 → SP ; 0 → D
FETCH     PC → BA; PC + 2 → D; DATI;CLKOFF
          (wait) UNIBUS → I ; UNIBUS → STACK (PUSH)
          D → PC ; DISPATCH OP FIELD                                  **
          AC DESTINATION FIELD → BA
          PUSH EXLDA; JUMP ECALC
ECALC     I → STACK (PUSH)
          DISPATCH INDIRECT/INDEX FIELD                               **
          DISPLACEMENT → B
          PC - 2 → D ; SCOM = 3 ??
          D SHIFTED RIGHT 1 → U
          U + B.EL → D ; D → U ; D → B ; RETURN ; JUMP ECALC1         **
ECALC1    U + B → D ; U + B → BA
EXLDA     DATI ; CLOCKOFF;
          (wait) DESTINATION AC FIELD → BA
          UNIBUS → REGS (BA)


NEG 1,2

SERVICE   0 → SP ; 0 → D
FETCH     PC → BA; PC + 2 → D; DATI;CLKOFF
          (wait) UNIBUS → I ; UNIBUS → STACK (PUSH)
          D → PC ; DISPATCH OP FIELD                                  **
          AC DESTINATION FIELD → BA
          AC1 → D ; DISPATCH CARRY CONDITION; JUMP EXECAL
EXECAL    D → SRC
          CBIT → D
          D → T ; DISPATCH OPERATION FIELD
EXCAL1    REGS (BA) → B
          ¬ SRC → D ; D → B
          B + 1 → D ; D → SRC (SET CONDITION CODES)
SHIFT1    PS → B ; DISPATCH SHIFT FIELD
          T XOR B → D; D → PS
          SRC → D; DISPATCH NOLOAD BIT
          -----
NOLOD1    D → REGS (BA)
          PS ∧ 1 → D
          D → CBIT
          PS → D; DISPATCH SKIP FIELD
NOLSKP1   D → STACK (PUSH)
```

Appendix IV

Print Set and Print Commentary

This appendix contains logic diagrams and explanatory text for the CADR microprocessor design. The diagrams omit pin numbers and location information, and some condensation by elimination of repetititve and uninteresting logic, such as parity computation logic has been done.

Clock Generation

The main clock for the processor is of approximately 180 nanosecond period, with roughly equal phases. The processor registers are clocked on the rising edge of this waveform, which marks the beginning of a new processor cycle.

Memory devices in the processor requiring write pulses are written by a write pulse occuring during the second (low) phase of the main processor clock.

These fundamental timing signals are generated by the circuity on the CLOCK1 and CLOCK2 prints. Initially, the cross coupled nand flip flop in the upper left corner of the CLOCK1 print is set, either by completion of the previous cycle or by power up reset of the processor. In the absence of the HANG signal indicating a main memory response delay, the rising edge of the CYCLECOMPLETED flip flop travels down the series of tapped delay lines, producing the sequence of TPRxx signals. TPR40 is used to clear CYCLECOMPLETED, establishing the width of the pulse in the delay lines at about 40 nanoseconds.

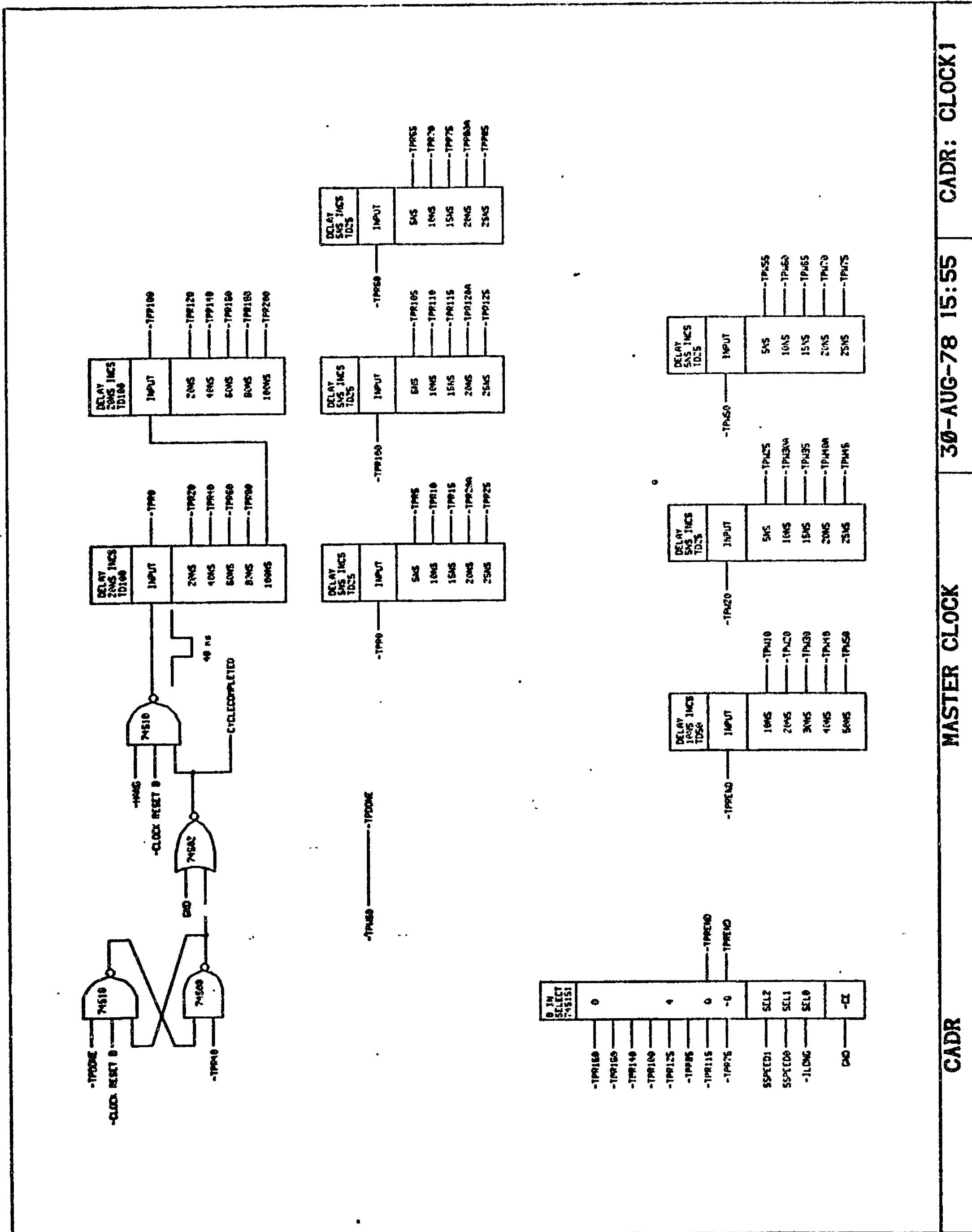
The rising edge of the main processor clock is produced by TPR0, which sets the TPCLK flip flop on the CLOCK2 print. Depending upon the speed with which the processor is running, a selected pulse from the TPRxx chain is gated into TPREND, which both clears TPCLK, establishing the time of fall for the main processor clock, and initiates the TPWxx delay line pulse. The TPWxx timing chain is used to produce the processor write pulse, and TPDONE which sets CYCLECOMPLETED in preparation for a new processor cycle.

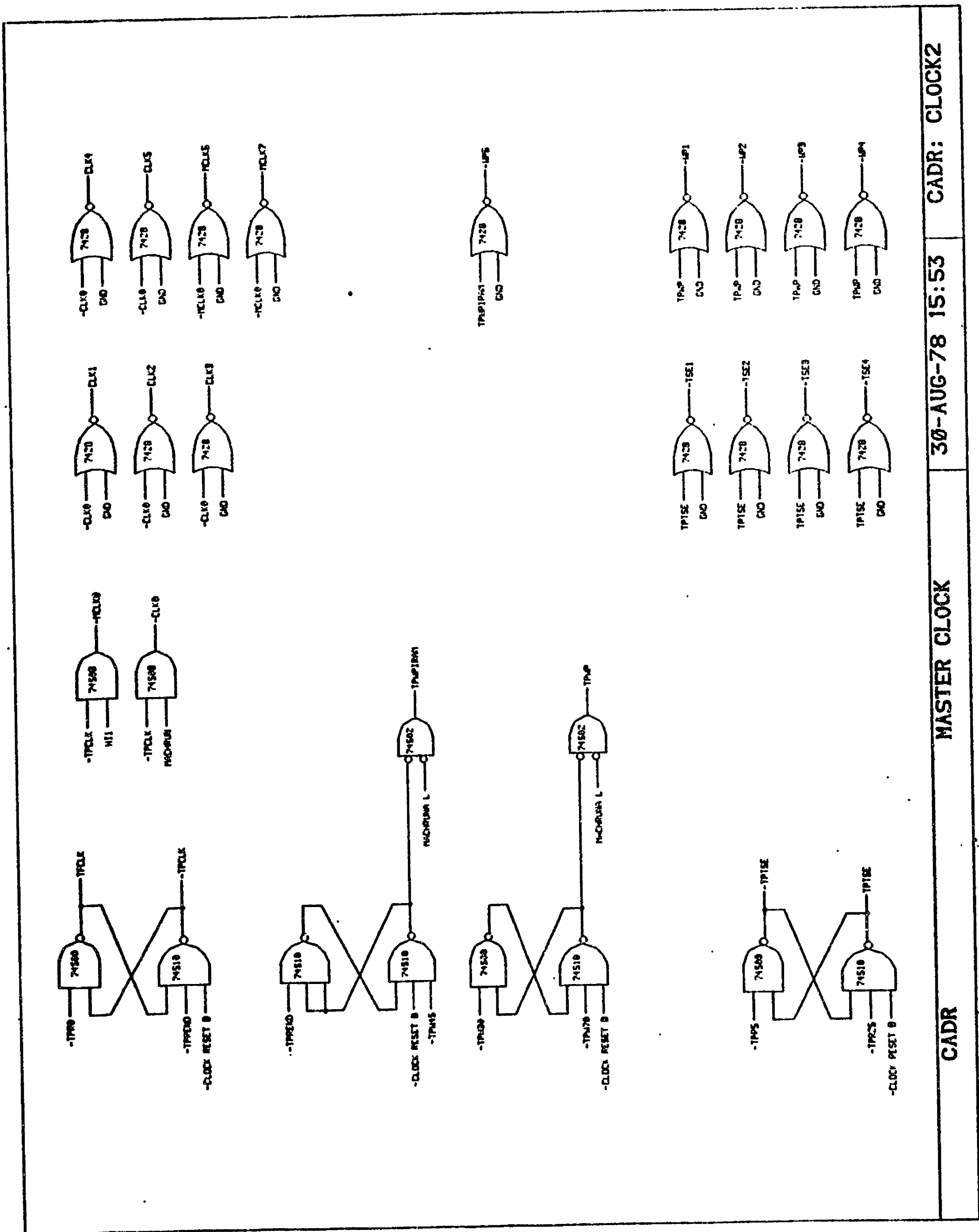
Buffering and gating of the main clock signals is done on the CLOCK2 print. The TPCLK and TPWP flip flop outputs are gated with the MACHRUN signal (processor running) to produce the final processor clock and write pulse. These are buffered once here in the clock circuitry, and once again local to their use in the processor, as shown in the CLOCKD print.

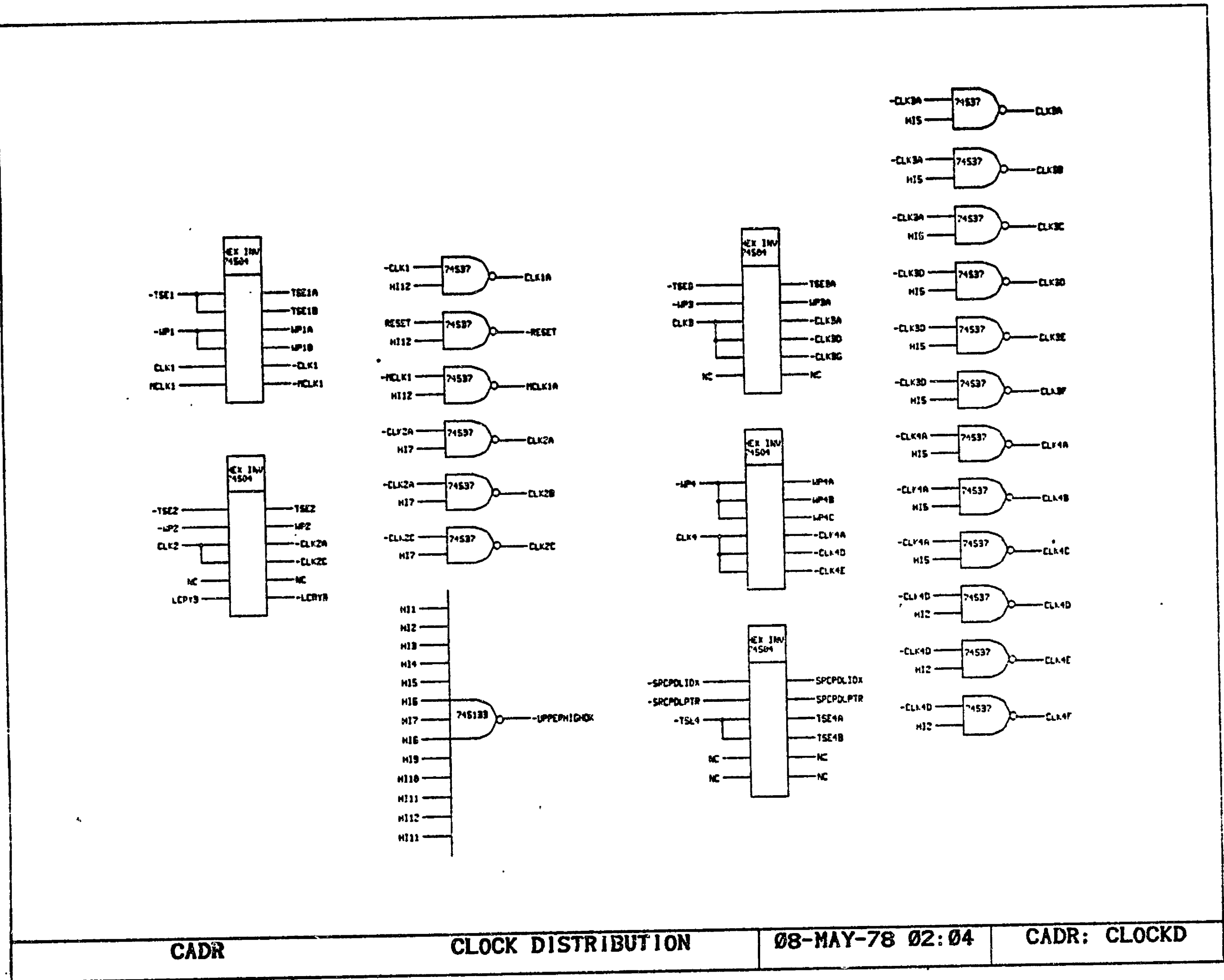
Two other timing signals are generated from the main timing chain. TPWPIRAM is used to produce a slightly wider and earlier write pulse for the control memory rams. TPTSE is a signal generated during the very first part of each processor cycle which disables all tristate drivers on the M and A busses, preventing noise problems due to tristate enable overlaps caused by instruction decoding skews.

The choice of a delay line clock generator rather than a totally synchronous crystal oscillator generator was based on a desire to allow the processor to "hang" waiting for a main memory response, and then immediately resume when the data was available. With a totally synchronous design, the delay before the available data could be used might be quite high due to synchronizer problems and quantization of time in units of the fastest oscillator interval.

PRINT ON THE FOLLOWING PAGES IS PARTIALLY ILLEGIBLE




CADR: CLOCK2
30-AUG-78 15:53
MASTER CLOCK
CADR

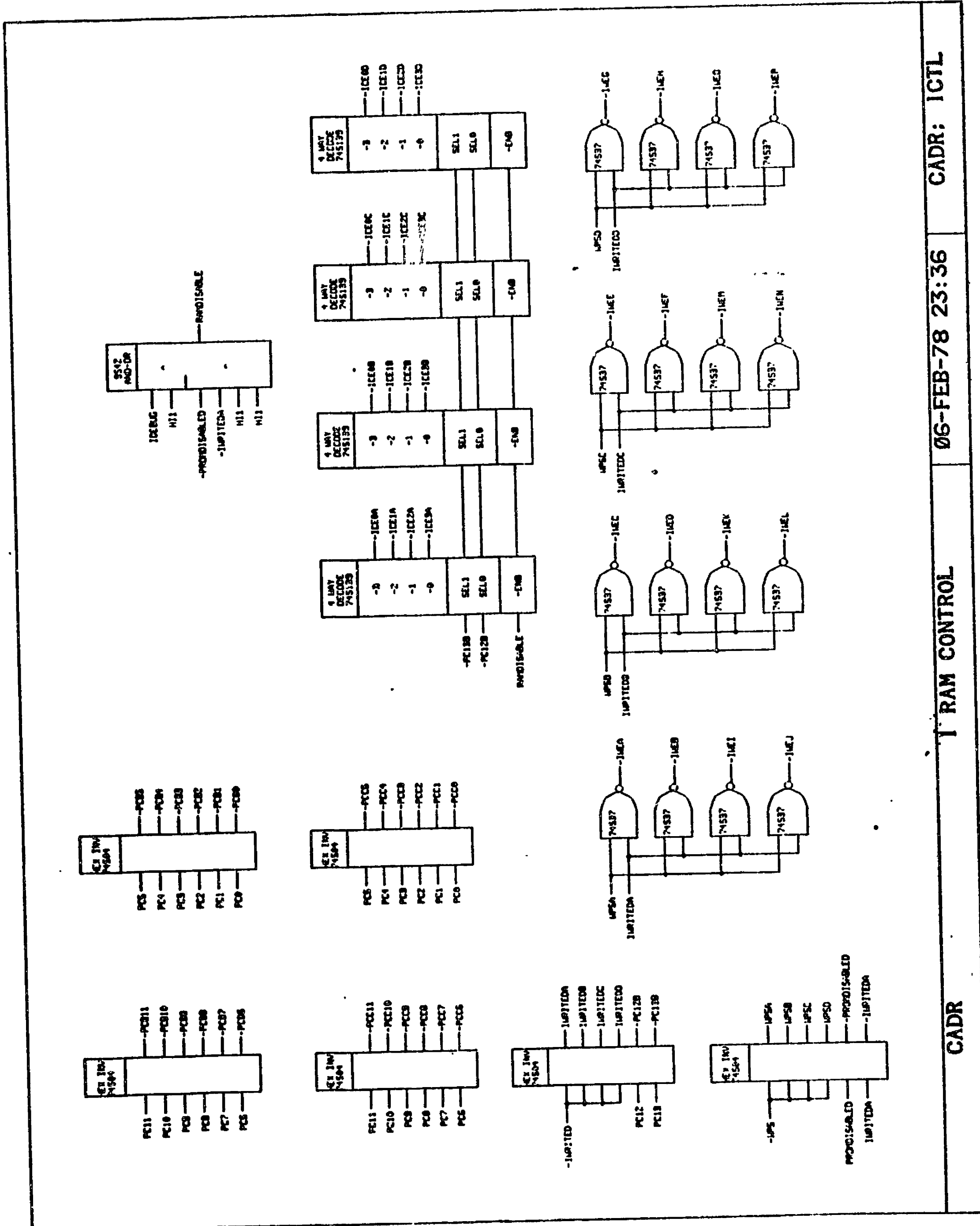

CADR
CLOCK DISTRIBUTION
Ø8-MAY-78 Ø2:Ø4
CADR: CLOCKD

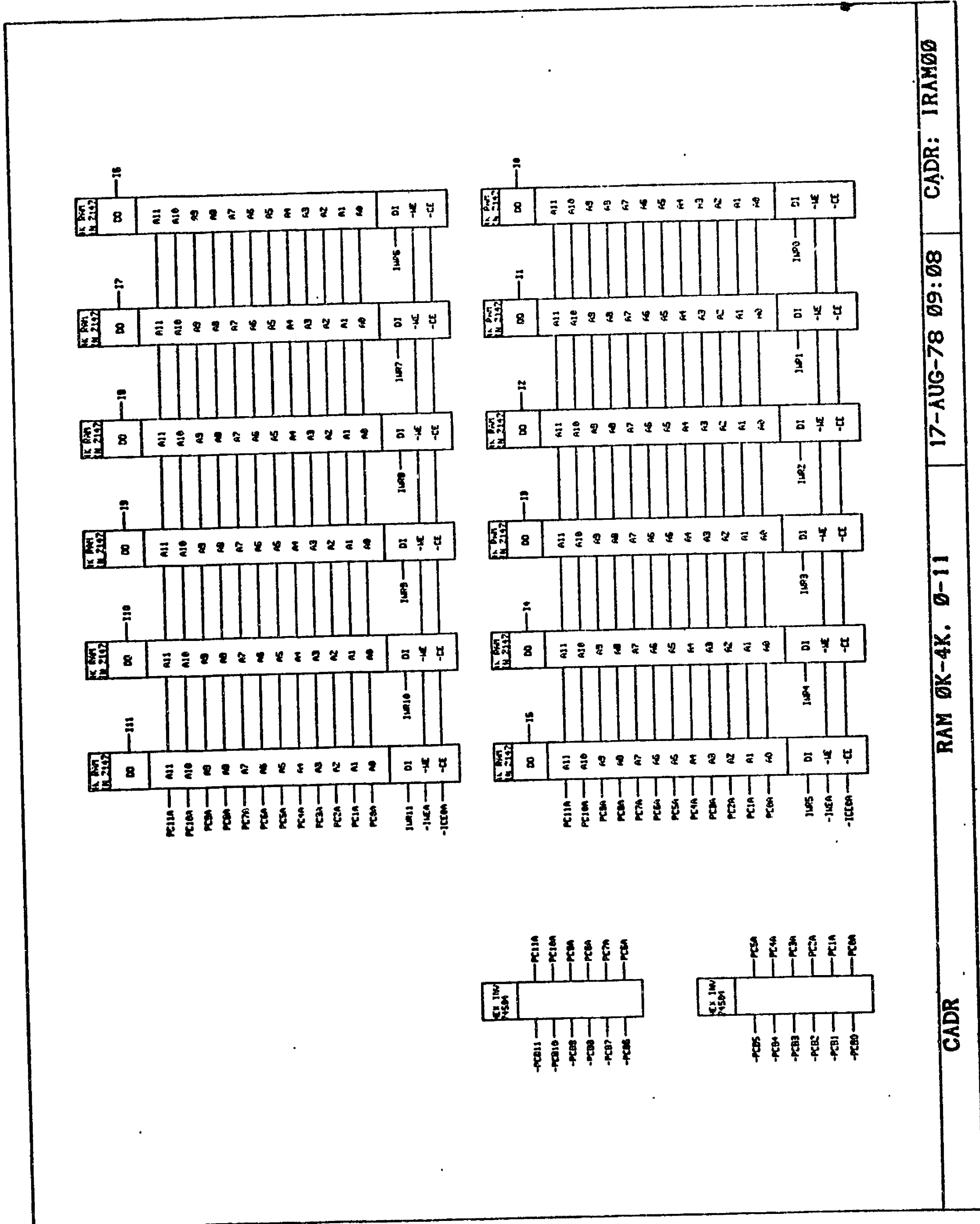
Microinstruction Fetch

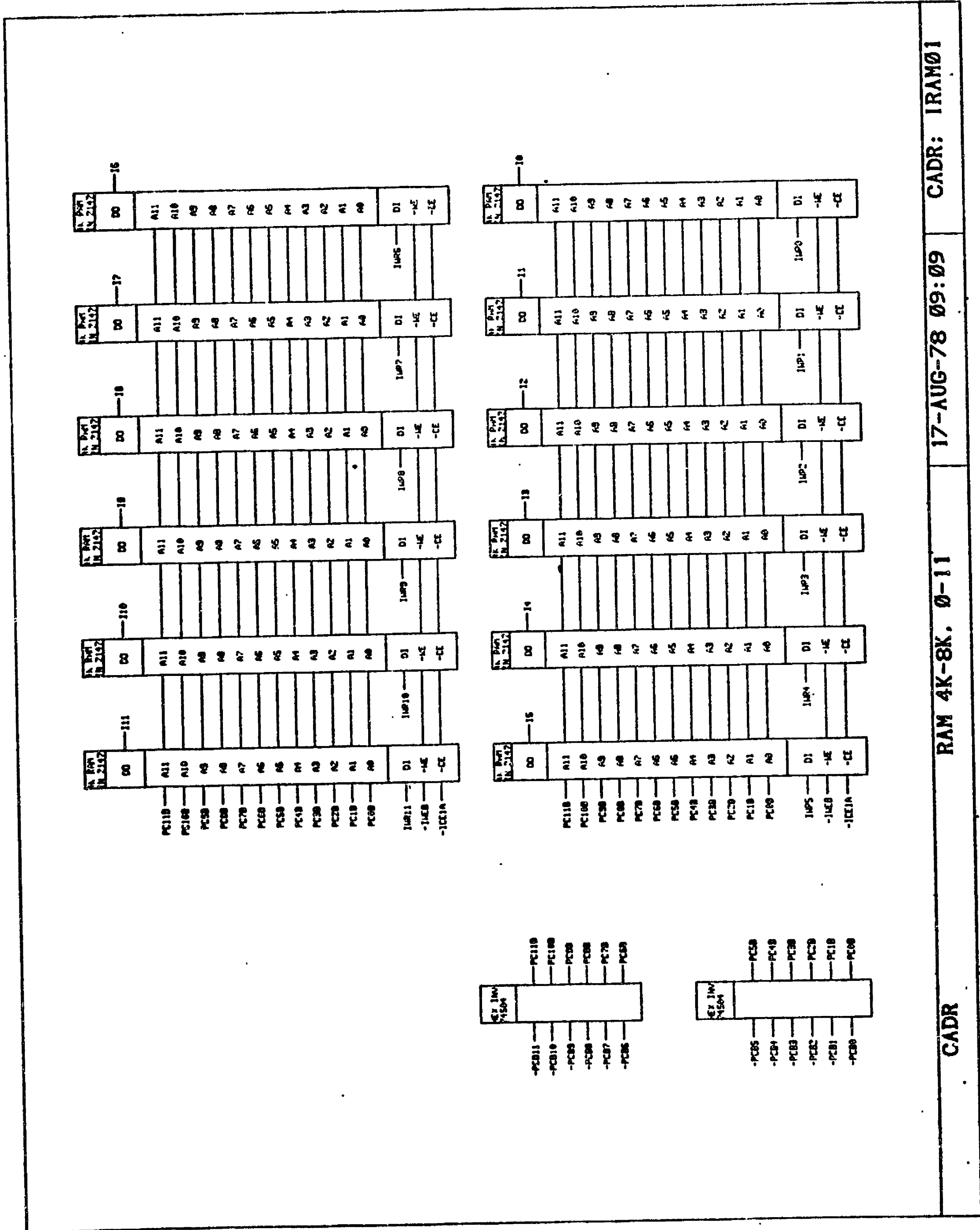
During each microcycle, the fetch of the next microinstruction to be executed is taking place. The PC contains the address of this next microinstruction. The ICTL print shows the decoding and initial buffering of the PC for driving the control memory address lines. A portion of the control memory itself is shown on the IRAM00 through IRAM03 prints. The microinstruction fetched is 49 bits wide, including the parity bit. The IWR print shows the register used to hold data which is about to be written into control memory (the A and M bus data from the control memory write instruction). The control memory write pulses are produced and buffered as shown on the ICTL print.

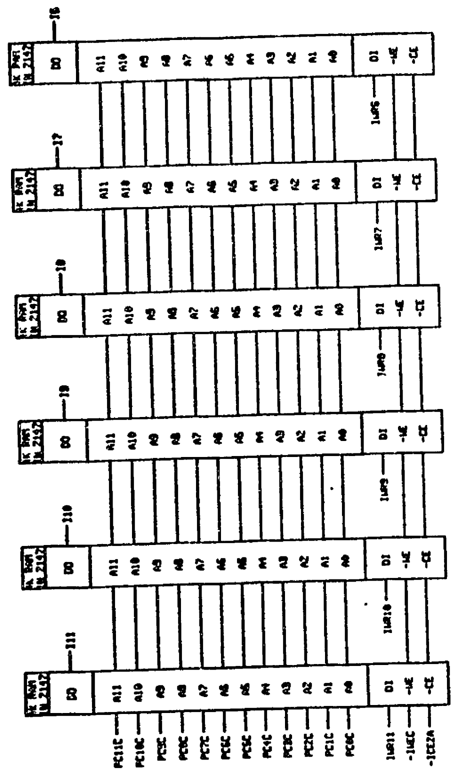
Two other sources of microinstructions exist. The first is the permanent PROM control memory, which is normally only active during the processor initial bootstrap. It is 1K by 48, the additional bit being forced to zero. The address buffering and enable logic is shown on the PCTL print, while the proms themselves are shown on the PROM0 and PROM1 prints.

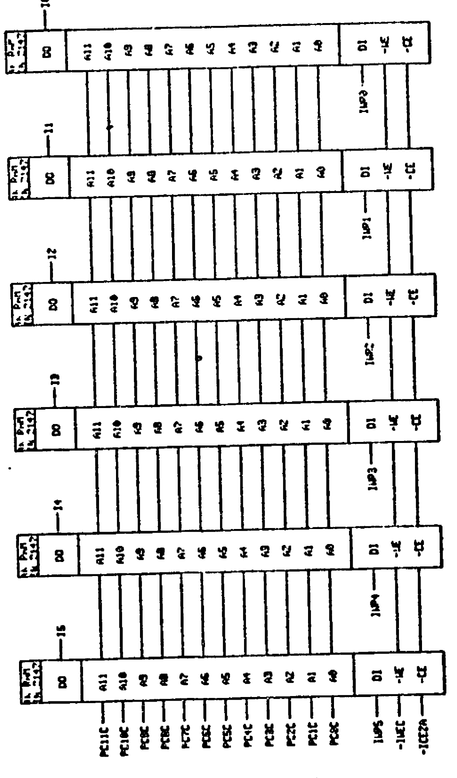
The second additional source of microinstructions is from a register loadable from the console computer interface. This source allows the debugging computer to execute arbitrary instructions on the processor, excercising its datapaths, and allowing access to internal processor state which is not directly readable through other console computer paths. The execution of microinstructions through this path is the main mechanism by which the console computer exerts control over the processor.

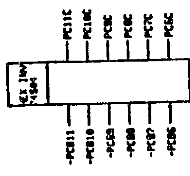

CADR: ICTL
Ø6-FEB-78 23:36
I RAM CONTROL
CADR

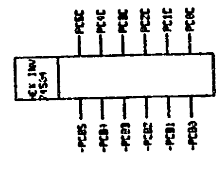

CADR
RAM ØK-4K. Ø-11
17-AUG-78 Ø9:Ø8
CADR: IRAMØØ


CADR: IRAMØ1
17-AUG-78 Ø9:Ø9
RAM 4K-8K. Ø-11
CADR







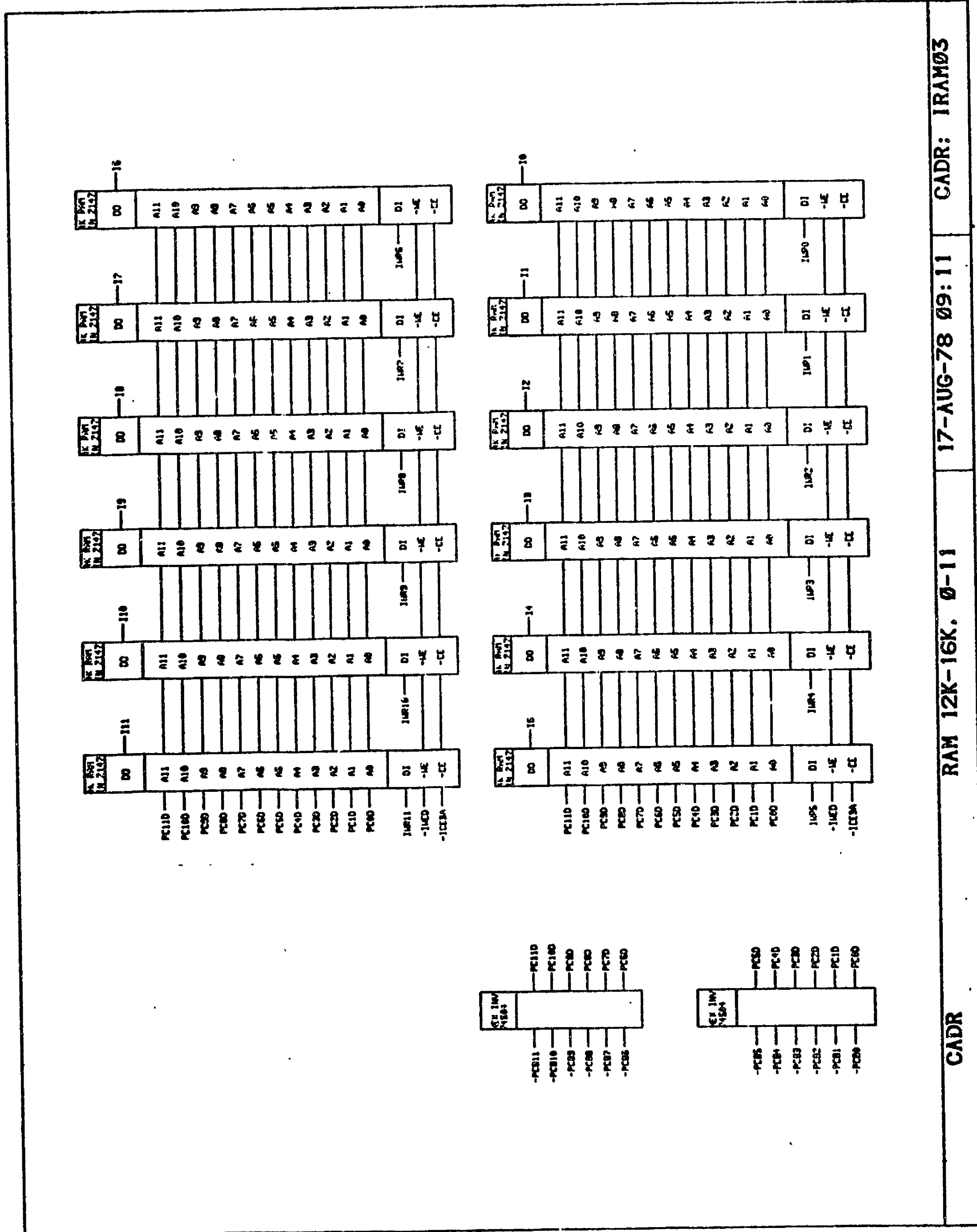


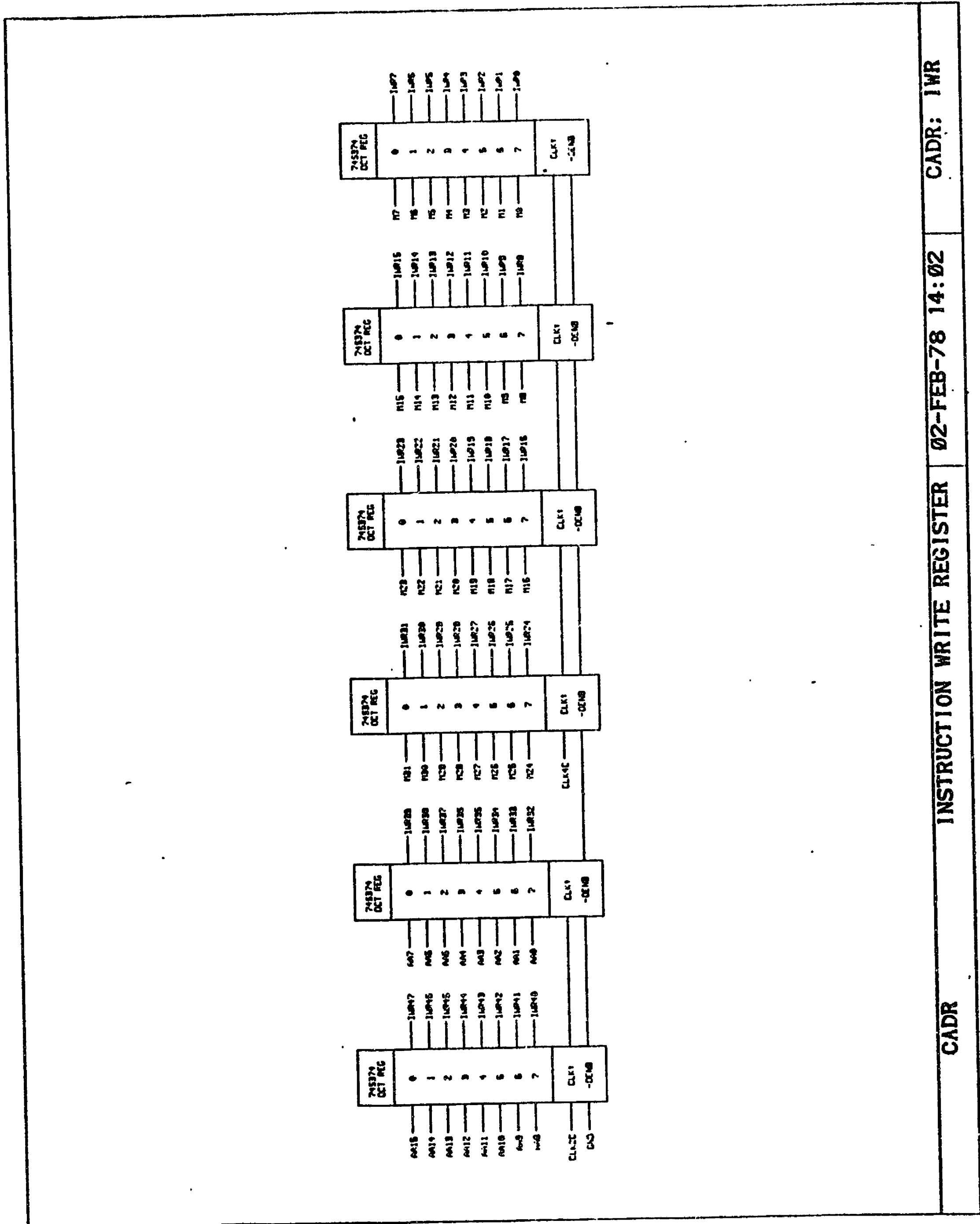
CADR

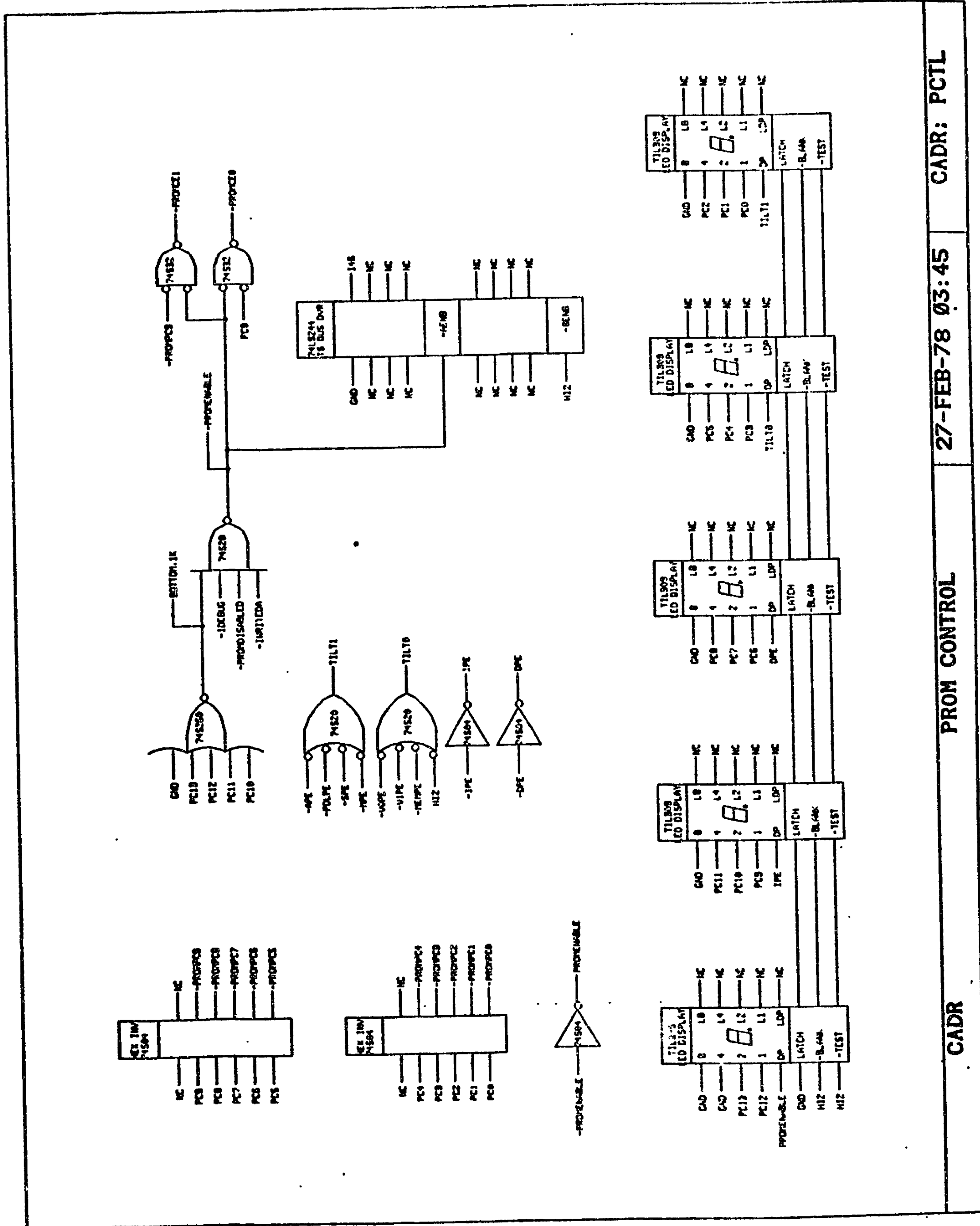
RAM 8K-12K. Ø-11

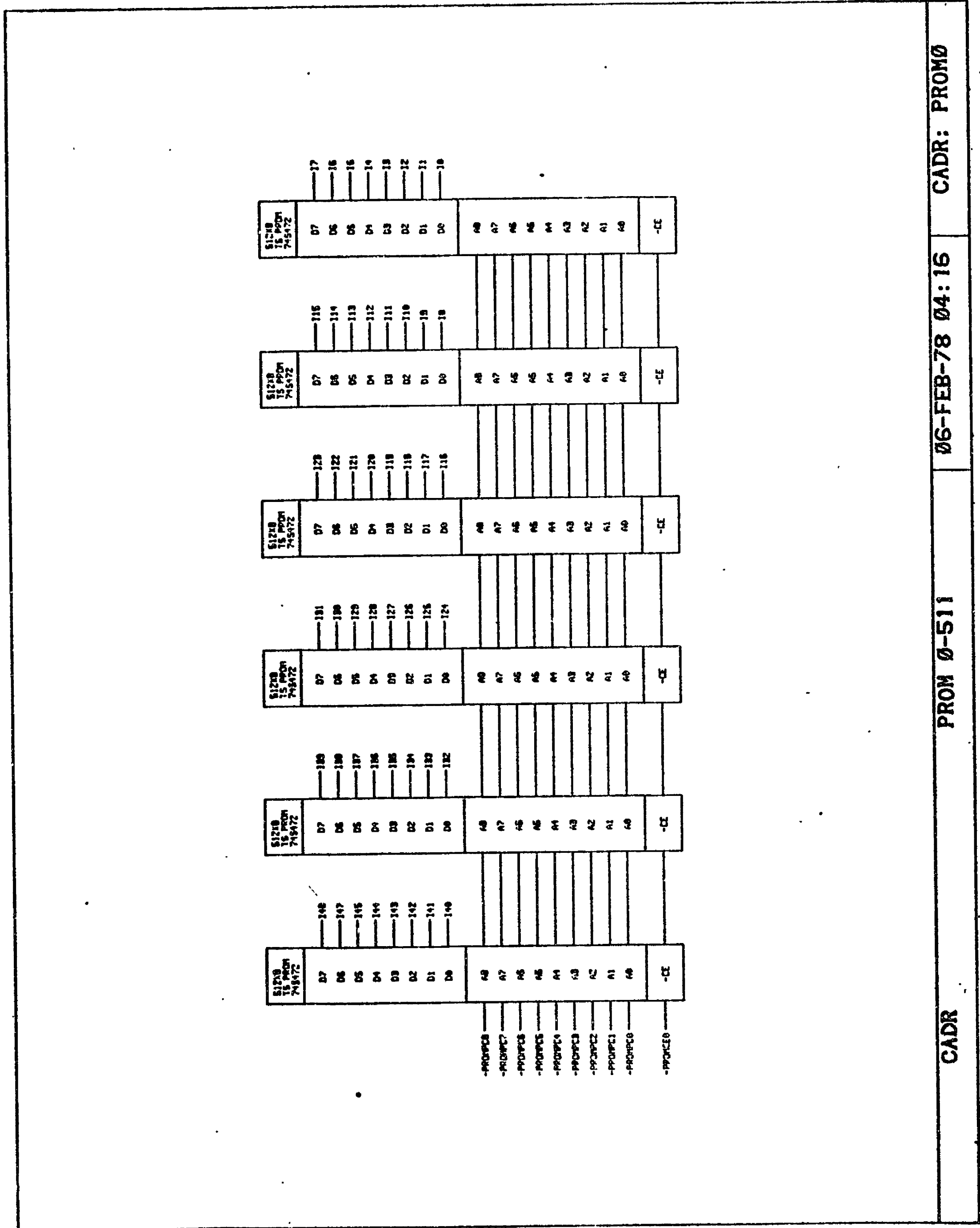
17-AUG-78 Ø9:1Ø

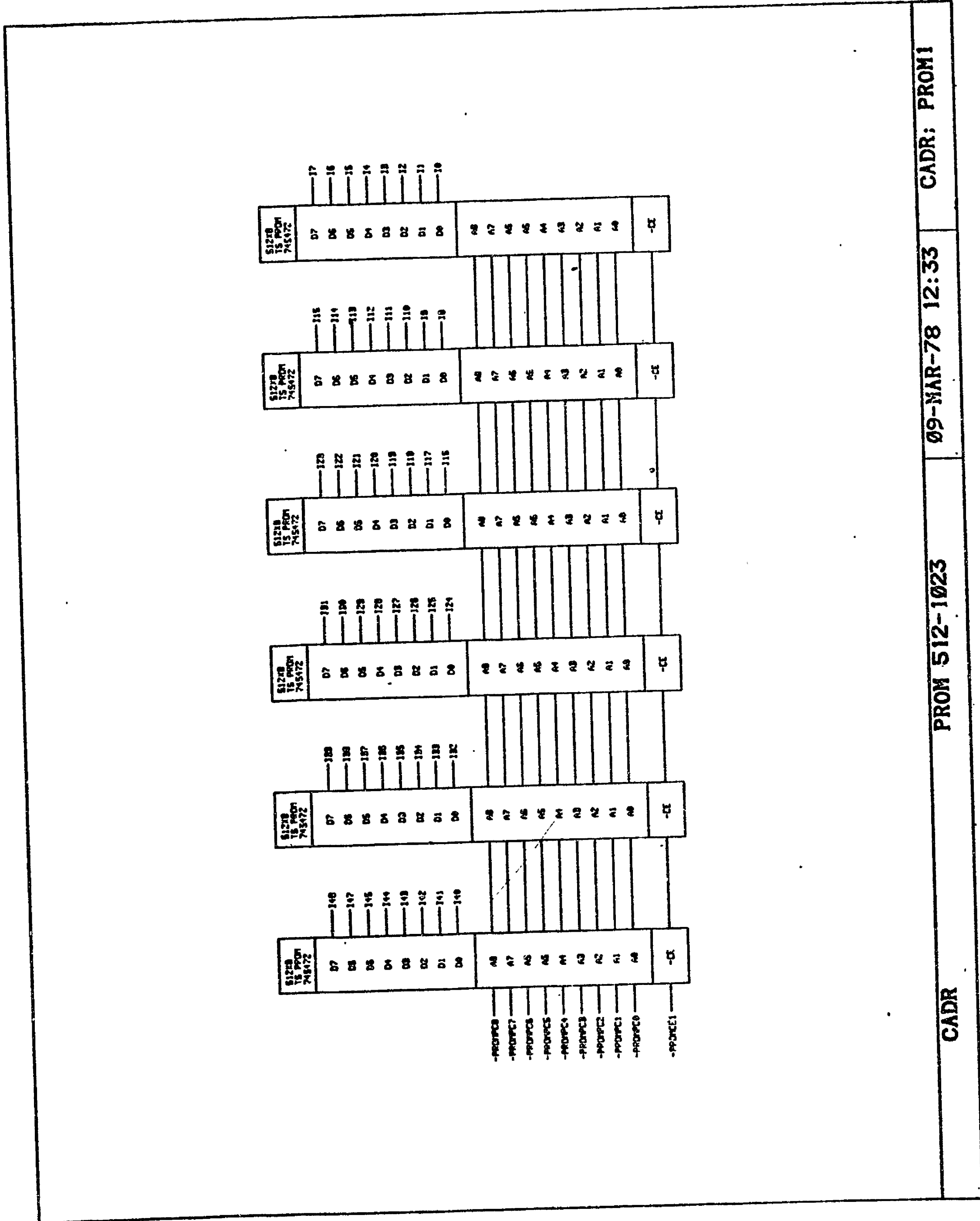
CADR: IRAMØ2


CADR
RAM 12K-16K. Ø-11
17-AUG-78 Ø9:11
CADR: IRAMØ3

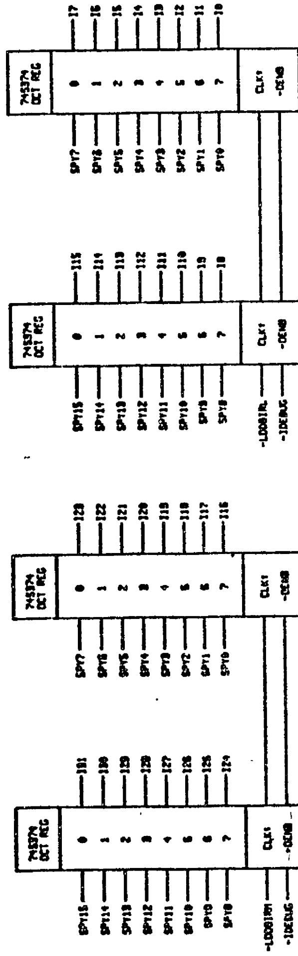

CADR
INSTRUCTION WRITE REGISTER
Ø2-FEB-78 14:Ø2
CADR: IWR

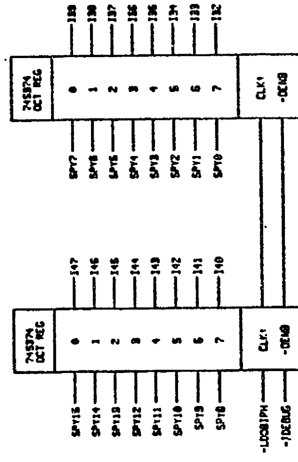

CADR
PROM CONTROL
27-FEB-78 03:45
CADR: PCTL


CADR: PROMØ
Ø6-FEB-78 Ø4:16
PROM Ø-511
CADR



CADR
PROM 512-1023
Ø9-MAR-78 12:33
CADR; PROM1





CADR

PDP11 DEBUG INSTRUCTION

26-JAN-78 16:52

CADR: DEBUG

Microinstruction Modification and Main Instruction Register

The fetched microinstruction word can potentially be modified by the result of the previously executed microinstruction. This is implemented on the IOR print by inclusively ORing the main output bus (OBx) with the output of control memory (Ixx) to produce the IOBxx modified signals. The main processor instruction register on the IREG print conditionally selects either the modified or unmodified result, depending upon whether the instruction modification feature has been activated. The clocking of a new instruction into to the IR register marks the initiation of execution for the new microinstruction.

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

74S32 OR GATE

CADR	INST. MODIFY OR	22-JAN-78 Ø6:19	CADR: IOR

CADR	INSTRUCTION REGISTER	24-JAN-78 07:34	CADR: IREG

IR Decoding

For the most part, the processor is designed so that the initial phase of microinstruction execution is independent of the details of the particular instruction. The decoding of the IR fields, then, as shown on the SOURCE print is not in the critical path of cycle timing.

The four main instructions are decoded from a pair of IR bits, providing the IRBYTE, IRALU, IRJUMP and IRDISP conditions. All of these are inhibited if the execution of this cycle is NOP'd.

The SRCxxx conditions specify a particular source of M bus data. These are activated only if a normal M memory location is not specified (IR31 set).

The destination codes are not needed until quite late in a cycle and are decoded in several stages. First, only the IRBYTE and IRALU instructions specify destinations, producing the DEST signal. Next, only destination codes with bit IR25 clear specify and M register address (less than 32.) producing DESTM. These, of course, also specify a functional destination, which is decoded from IR19 to IR22. If IR23 is on, the functional destination is to either the VMA or MD register, and the memory subroutine further decodes this field.

CADR	SOURCE. DEST. OP DECODE	16-AUG-78 00:44	CADR: SOURCE

A Memory

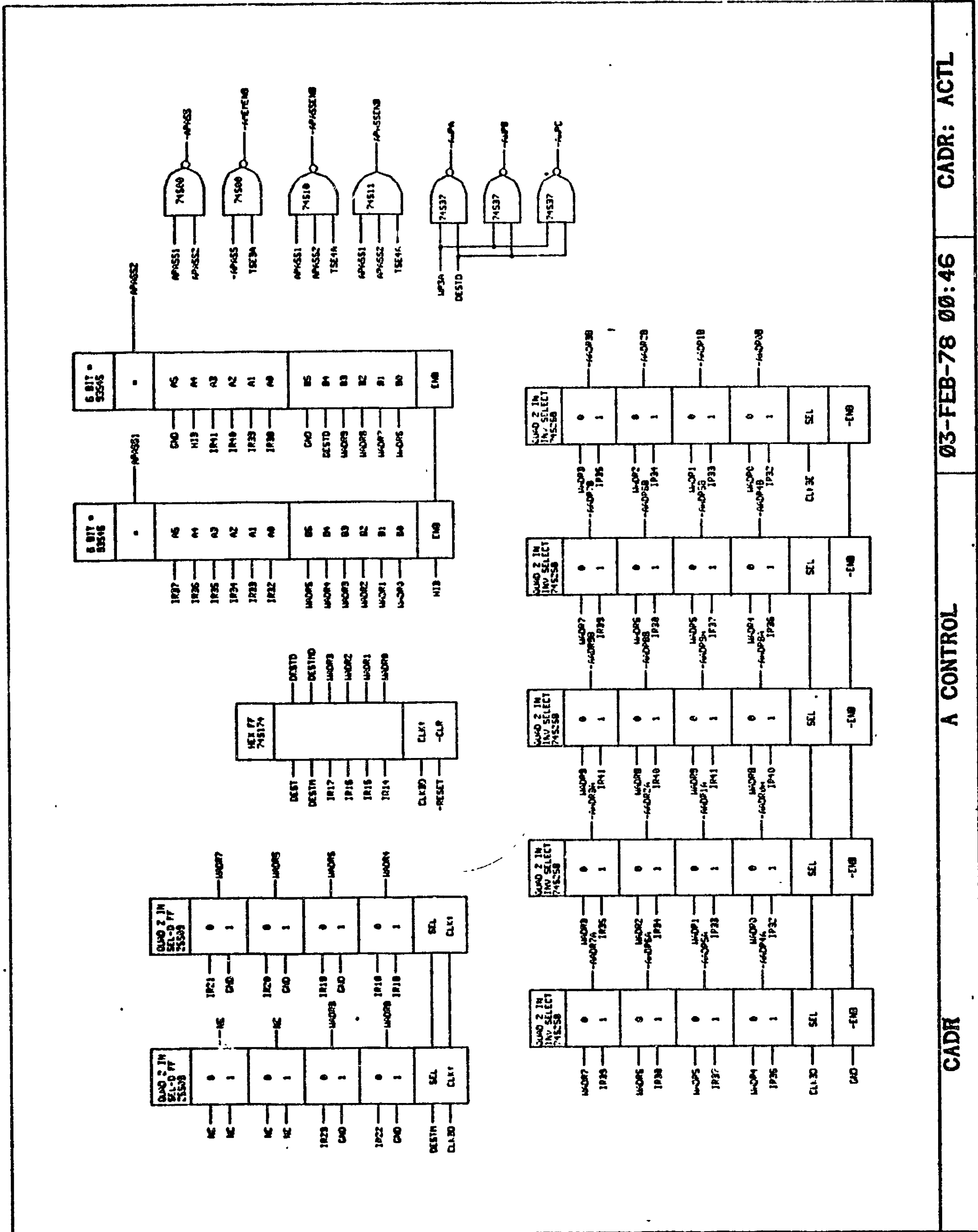
All instructions begin with the fetching of data from the M and A busses. Regardless of whether the data will be used, data is fetched from all of the memories in the machine at the beginning of each cycle.

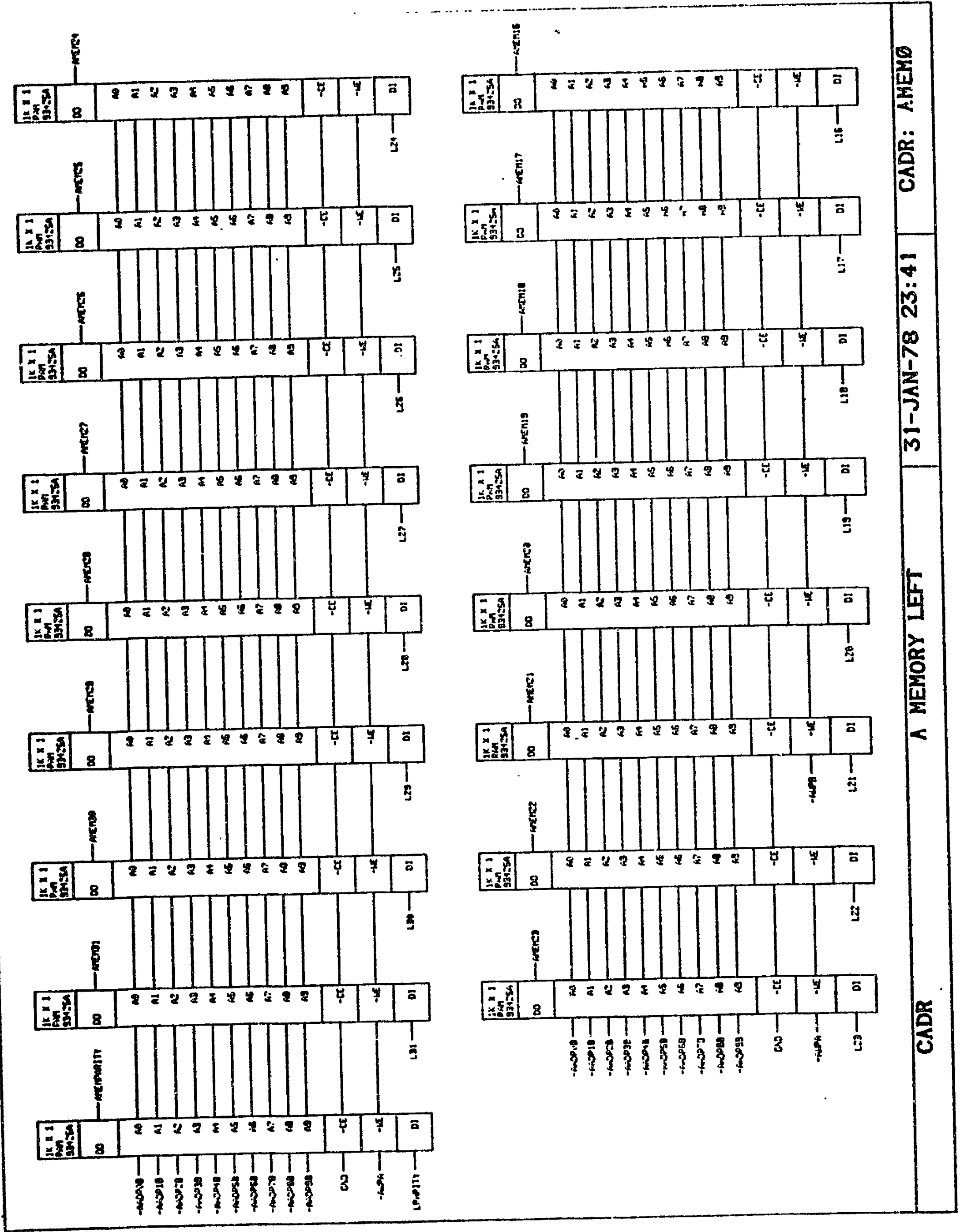
In the A memory, the data fetch begins with the high phase of the clock selecting IR bits 41-32 for driving the A memory address lines (AADRxx), as shown on the ACTL print. Simultaneously, the A memory address is compared with the A memory write address from the previous cycle, in order to detect the situation in which a read is being performed on a location which has yet to be written. The APASS signal indicates that this condition has occured.

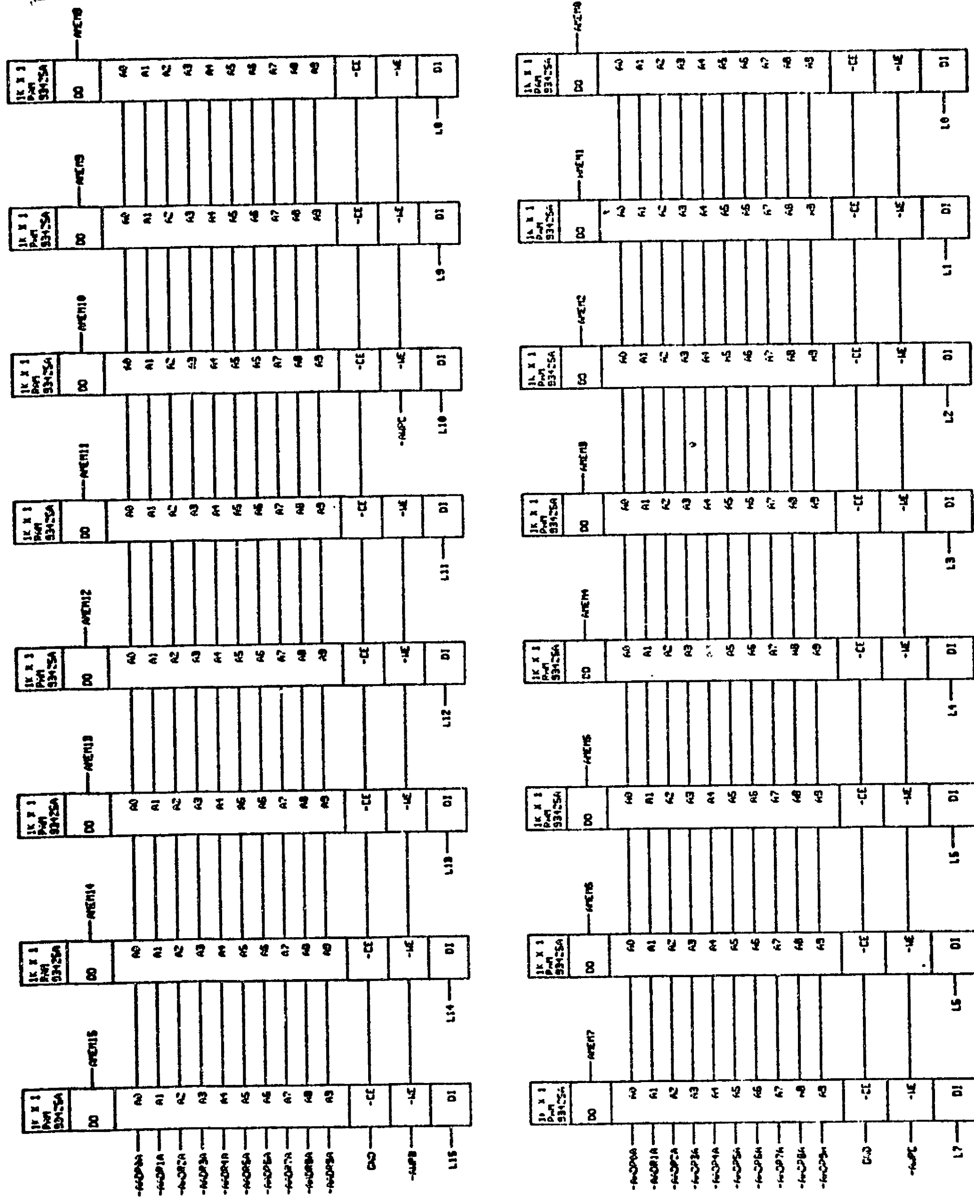
After the address access time of the A memory rams (shown on AMEM0 and AMEM1), the AMEMxx signals are valid at the input of the A memory latch (print ALATCH). Either this data or the contents of the L register (which stores the yet to be written result of the previous instruction) is driven onto the A bus as determined by the APASS signal (AMEMENB and APASSENB on ACTL).

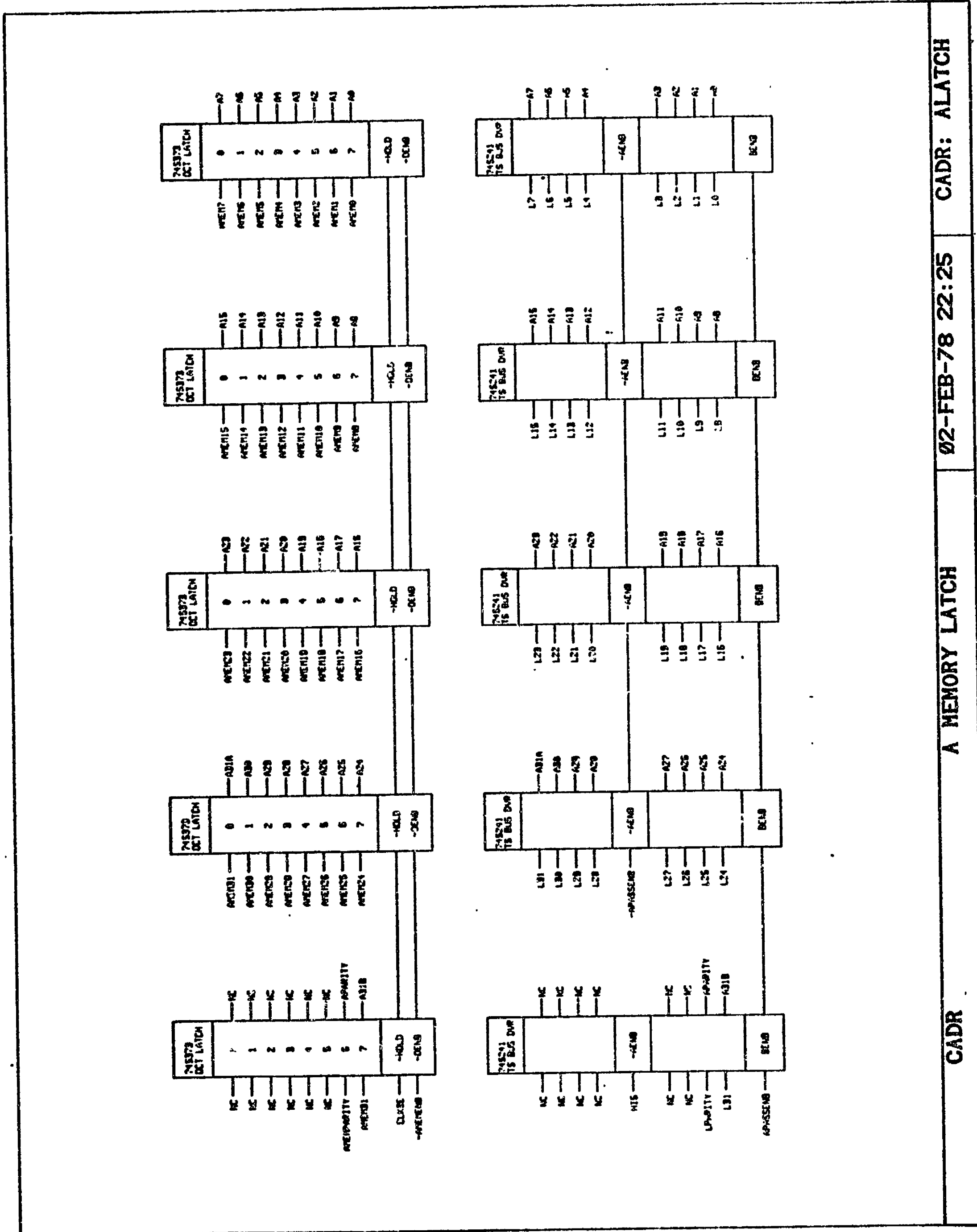
When the main clock falls, the A memory latches close, holding the output of the A memory for the remainder of the cycle. Data from the L register will of course be valid throughout the cycle.

Simultaneously, the A memory address selector (ACTL) selects the previous cycle's write address as the A memory address. If the previous cycle specified an A memory destination (DEST) then an A memory write pulse is generated (AWPx) writing the data from the previous cycle.


CADR
A CONTROL
03-FEB-78 00:46
CADR: ACTL


CADR
A MEMORY LEFT
31-JAN-78 23:41
CADR: AMEMØ

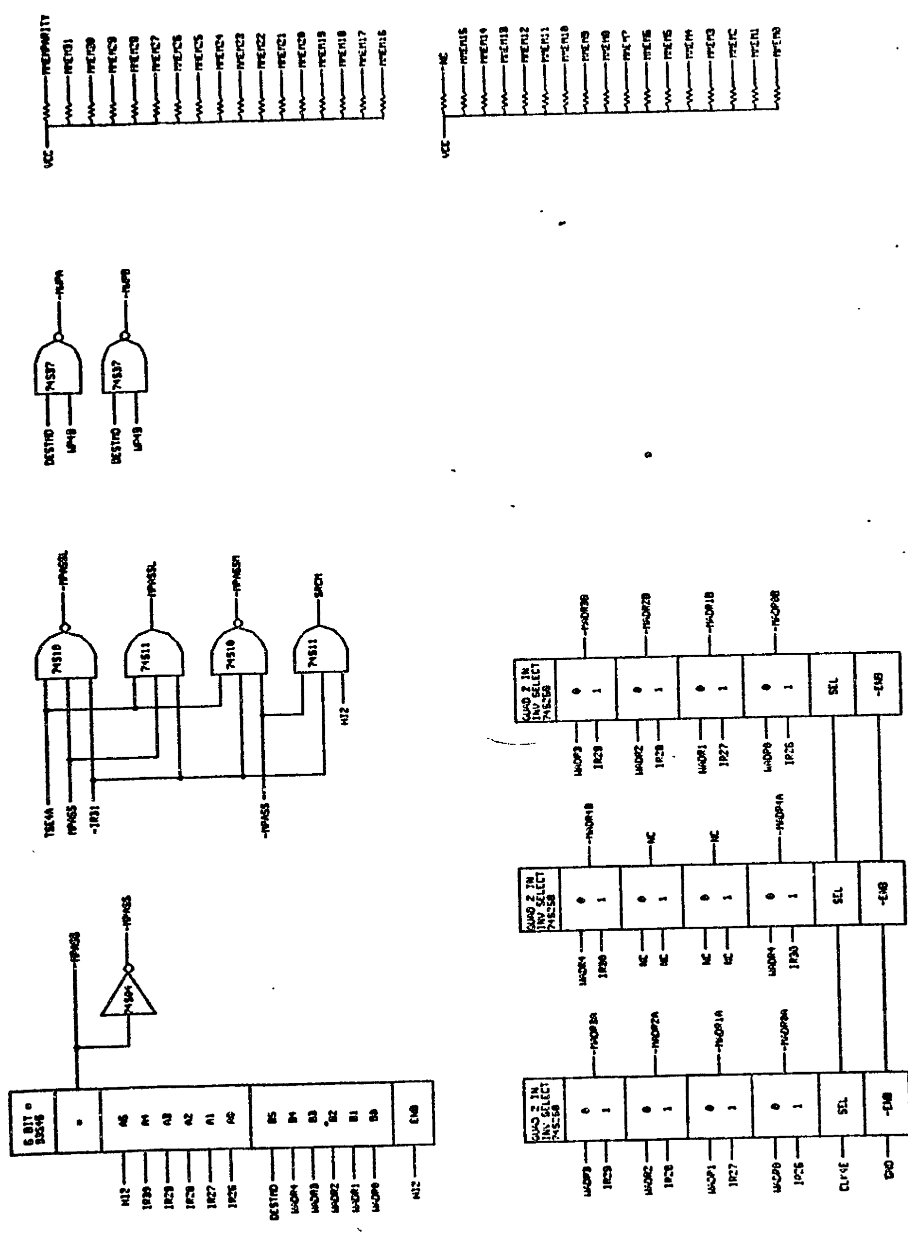

CADR: AMEM1
31-JAN-78 23:42
A MEMORY RIGHT
CADR

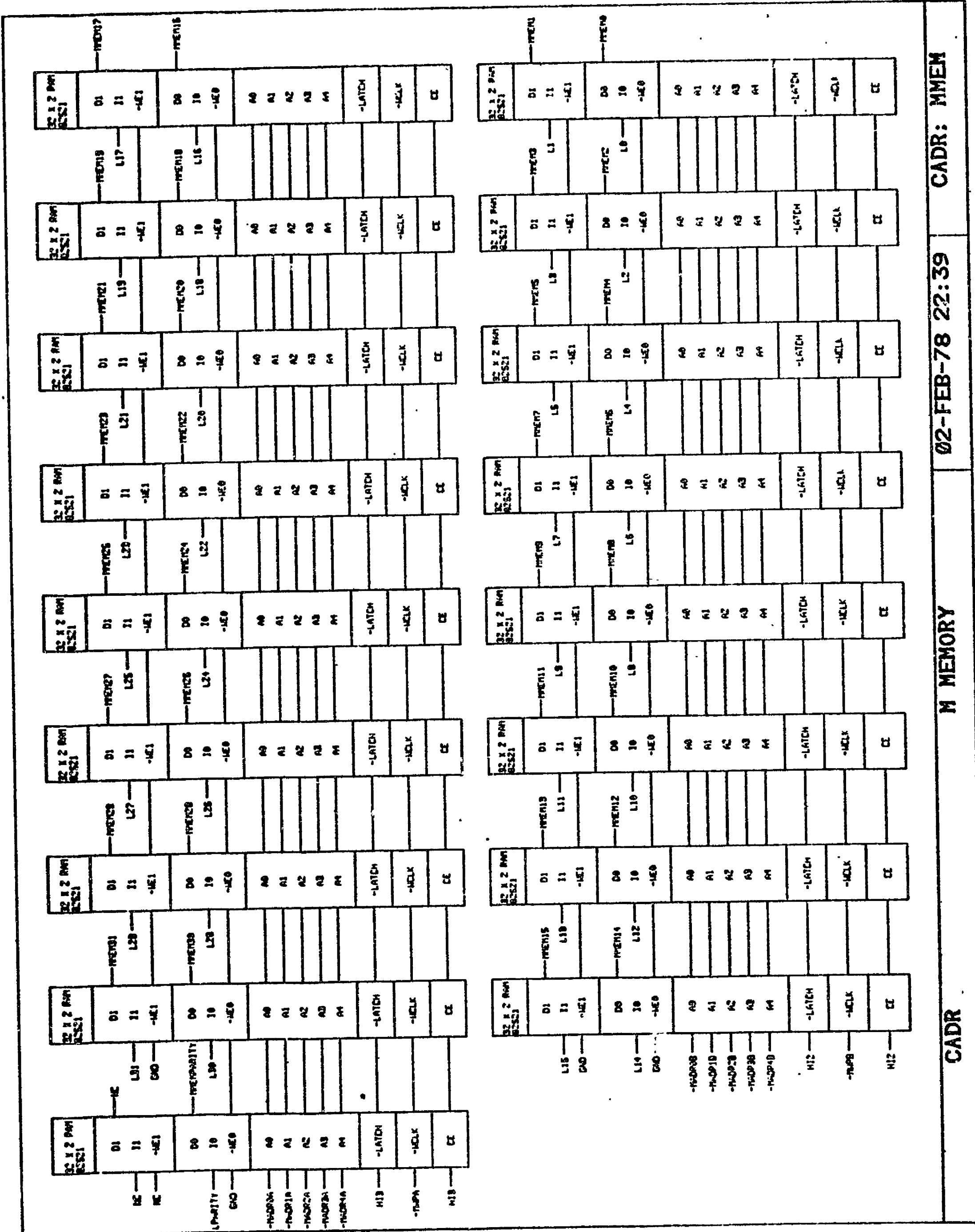

CADR: ALATCH
Ø2-FEB-78 22:25
A MEMORY LATCH
CADR

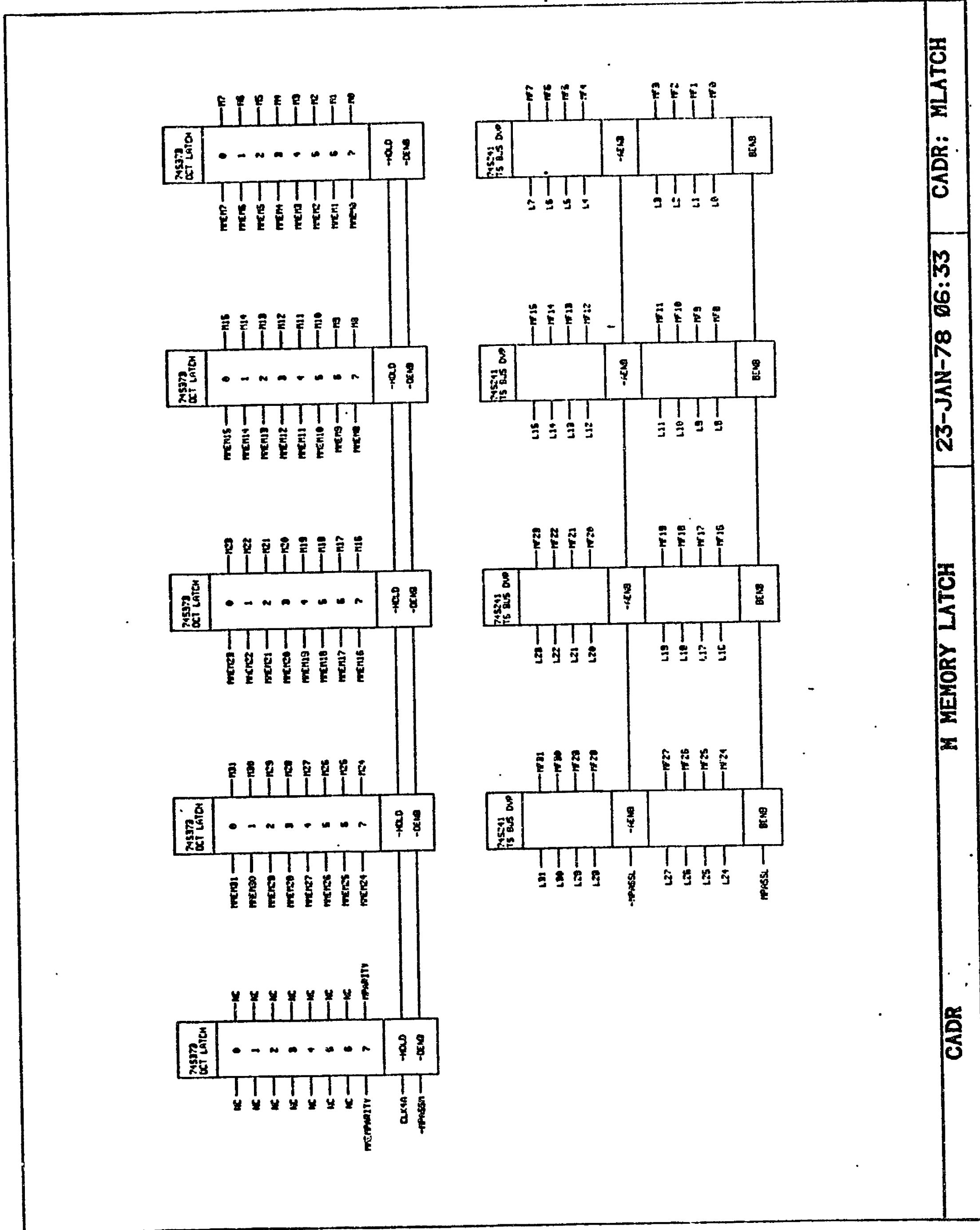
M Memory

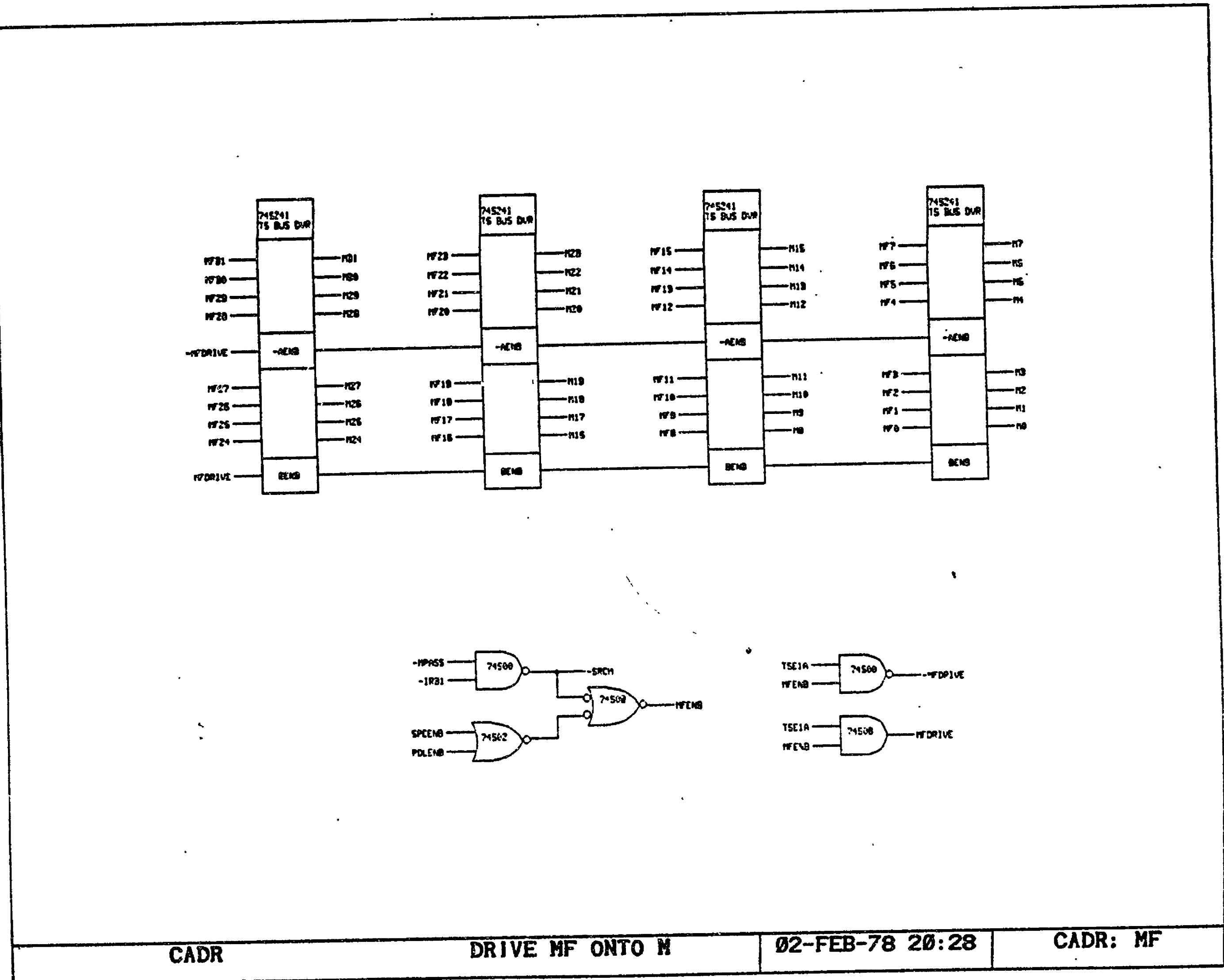
Timing of M memory fetches and writes is very similar to timing of the A memory, and is shown on the MCTL, MMEM, MLATCH prints.

The major difference is a two-level buffering of the M bus data, introduced for loading reasons. The M bus is split into two parts: a short bus, unbuffered for "slow" sources, and a long bus, buffered, for "fast" sources. The short bus, labeled Mxx, is driven from the M memory latches, the PDL buffer latches, the SPC latches, and from the buffered outputs of the long bus. The long bus, labelled MFxx is driven from all of the remaining M bus sources, including the L register output, which is used for the pass around path. The MFxx to Mxx buffers are shown on the MF print.


CADR: MCTL
06-FEB-78 03:59
M CONTROL
CADR


CADR
M MEMORY
Ø2-FEB-78 22:39
CADR: MMEM


CADR
M MEMORY LATCH
23-JAN-78 06:33
CADR: MLATCH


CADR
DRIVE MF ONTO M
02-FEB-78 20:28
CADR: MF

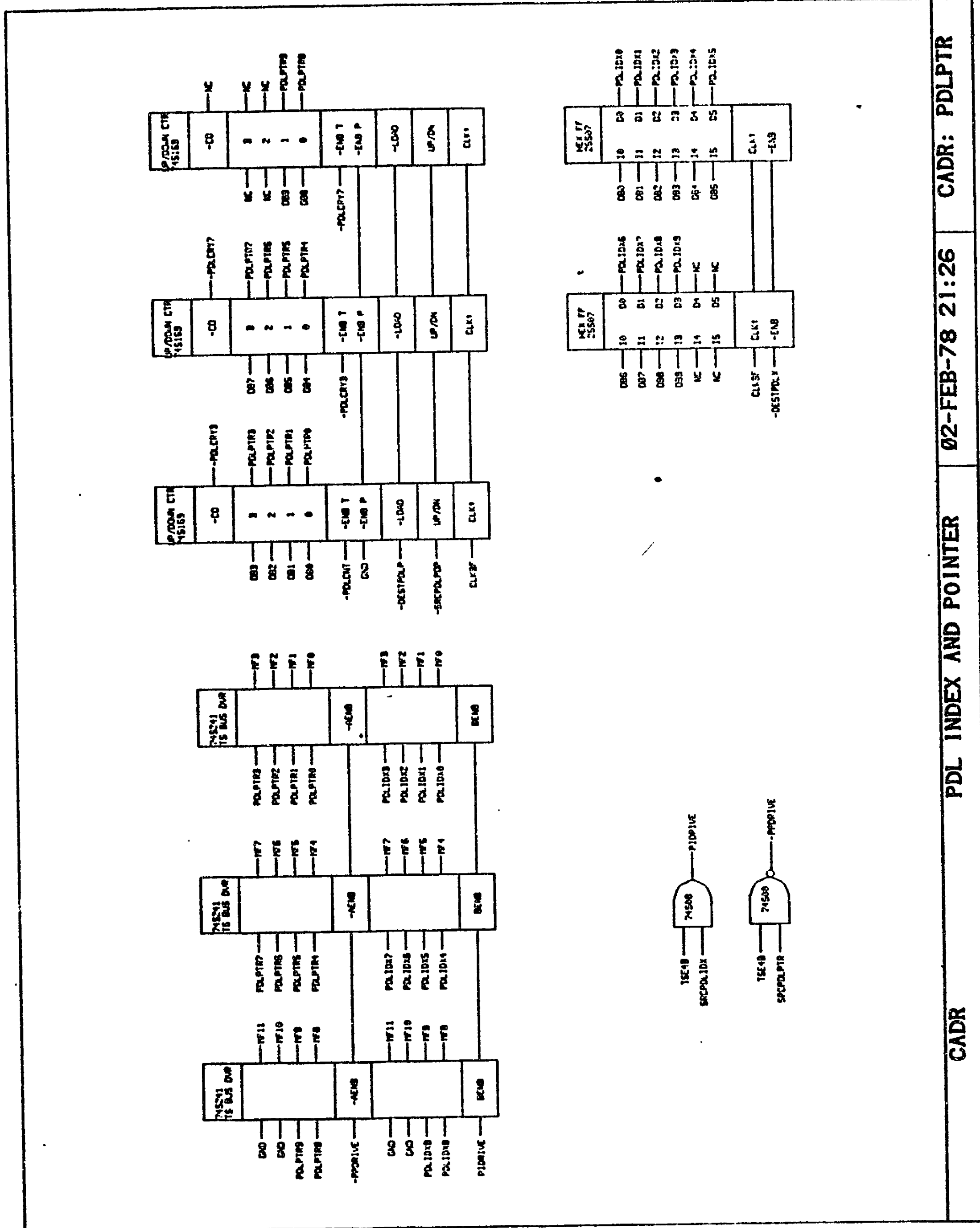
Stack Buffer

The stack buffer is addressed from one of two registers, the PDLPTR or the PDLIDX. The PDLCTL print shows the address selector for the buffer. During the first half of a cycle, the buffer performs a read from either the address contained in the PDLPTR or the address contained in the PDLIDX controlled by IR bit 30. The output of the buffer memories (shown on PDL0 and PDL1) drive the inputs of the latches shown on the PLATCH print. When the clock falls, this set of latches holds the output of the buffer for the remainder of the cycle. The latch output is driven onto the Mxx lines as an M source if the M source field contains either SRCPDL or SRCPDLPOP.

The stack buffer address lines also change when the clock falls, switching the address so that the results of the previous cycle's write in the stack buffer may be performed. If a write is to occur, the PWPx write pulses are produced.

The PDLIDX and PDLPTR registers are conditionally loaded directly from the output bus at the rising edge of the main clock. In addition, the PDLPTR register can conditionally count up or down by one to perform the push and pop operations. Due to the delay in performing the write until the succeeding cycle, the post-decrement pop and pre-increment push occur automatically due to the intervening rising clock edge. Since pushing and immediately poping data on the stack is a rather meaningless excercise, there is no facility in the stack buffer for performing data passarounds. As a result, it is possible to produce nominally correct, although inefficient, code which will function incorrectly.

Either the PDLPTR or PDLIDX may be read onto the M bus as fast sources buffered through the MF buffers.

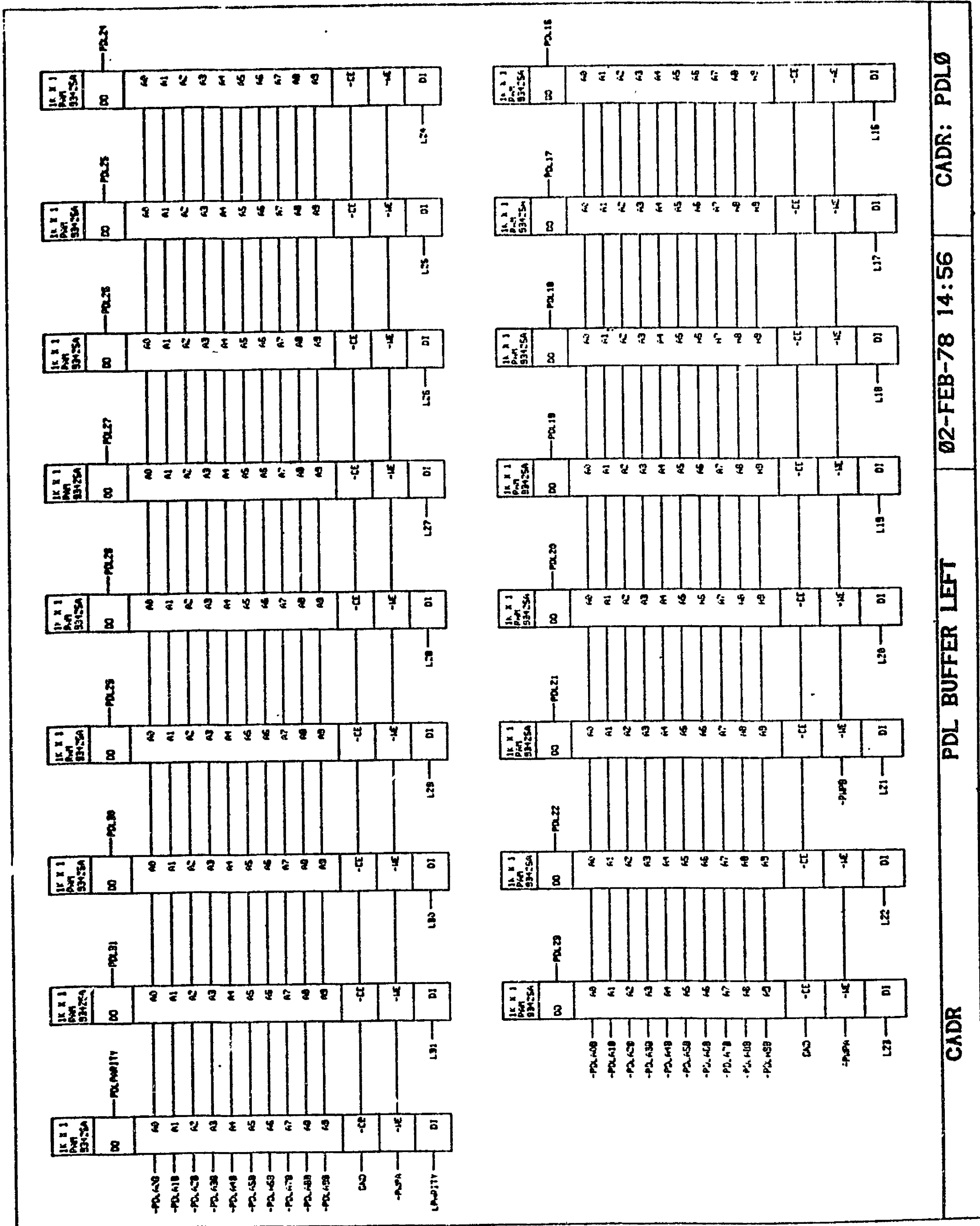

CADR
PDL INDEX AND POINTER
Ø2-FEB-78 21:26
CADR: PDLPTR

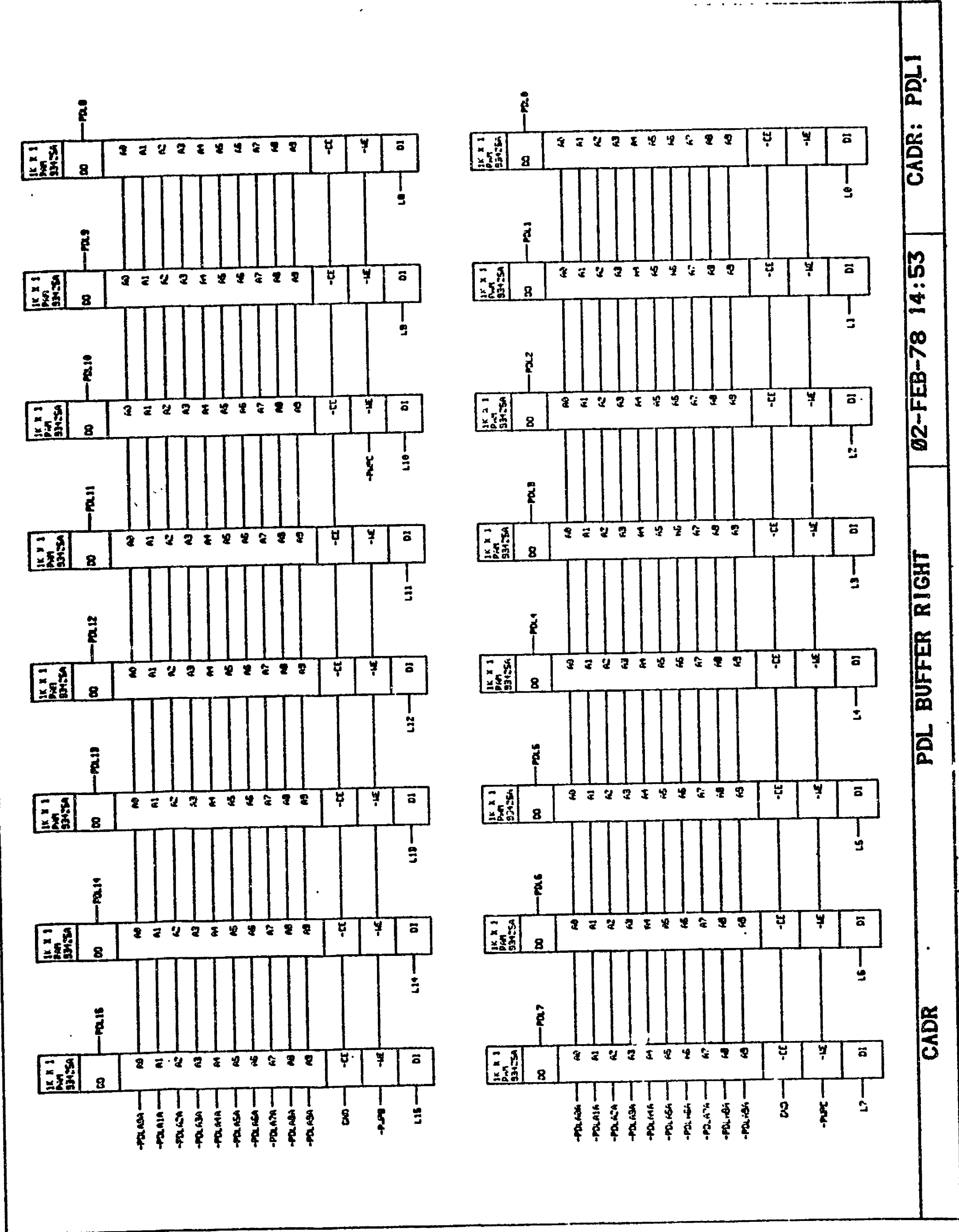
CADR

PDL BUFFER CONTROL

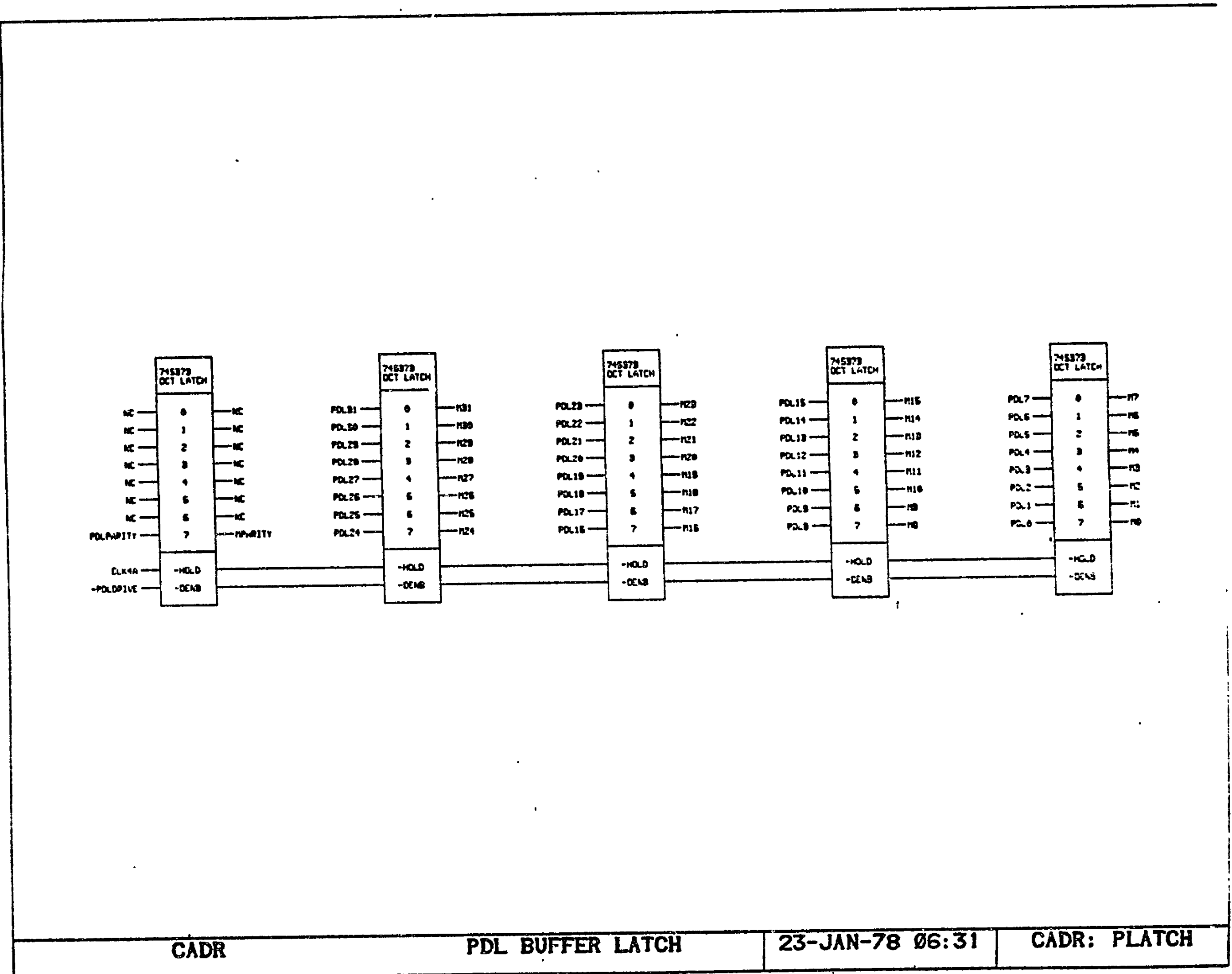
15-FEB-78 00:09

CADR: PDLCTL


CADR
PDL BUFFER LEFT
Ø2-FEB-78 14:56
CADR: PDLØ



CADR
PDL BUFFER RIGHT
Ø2-FEB-78 14:53
CADR: PDL1


CADR
PDL BUFFER LATCH
23-JAN-78 Ø6:31
CADR: PLATCH

The Shifter/Masker

The A and M bus data drive both the ALU and the shifter/masker. The shifter is controlled by a five bit shift field generated on the SMCTL print. The shift is usually specified from IR4-IR0, but this selection can be modified in two ways. In the selective deposit instruction, the shift of the M source is forced zero, by setting IR bit 12 in a byte instruction. In instructions referencing macroinstruction byte streams of 8 or 16 bit words, the shift can be modified in bits 4 and 3 by the contents of the macro program counter (LCC) if the miscellaneous function 3 is selected (IR bits 10 and 11 on). This allows instructions referencing a word containing this data to be automatically shifted to examine the correct byte with no extra cycles for performing shifts.

The specified five bit shift field is used to control a 32 x 32 shift matrix set up as a rotator, as shown on the SHIFT0 and SHIFT1 prints.

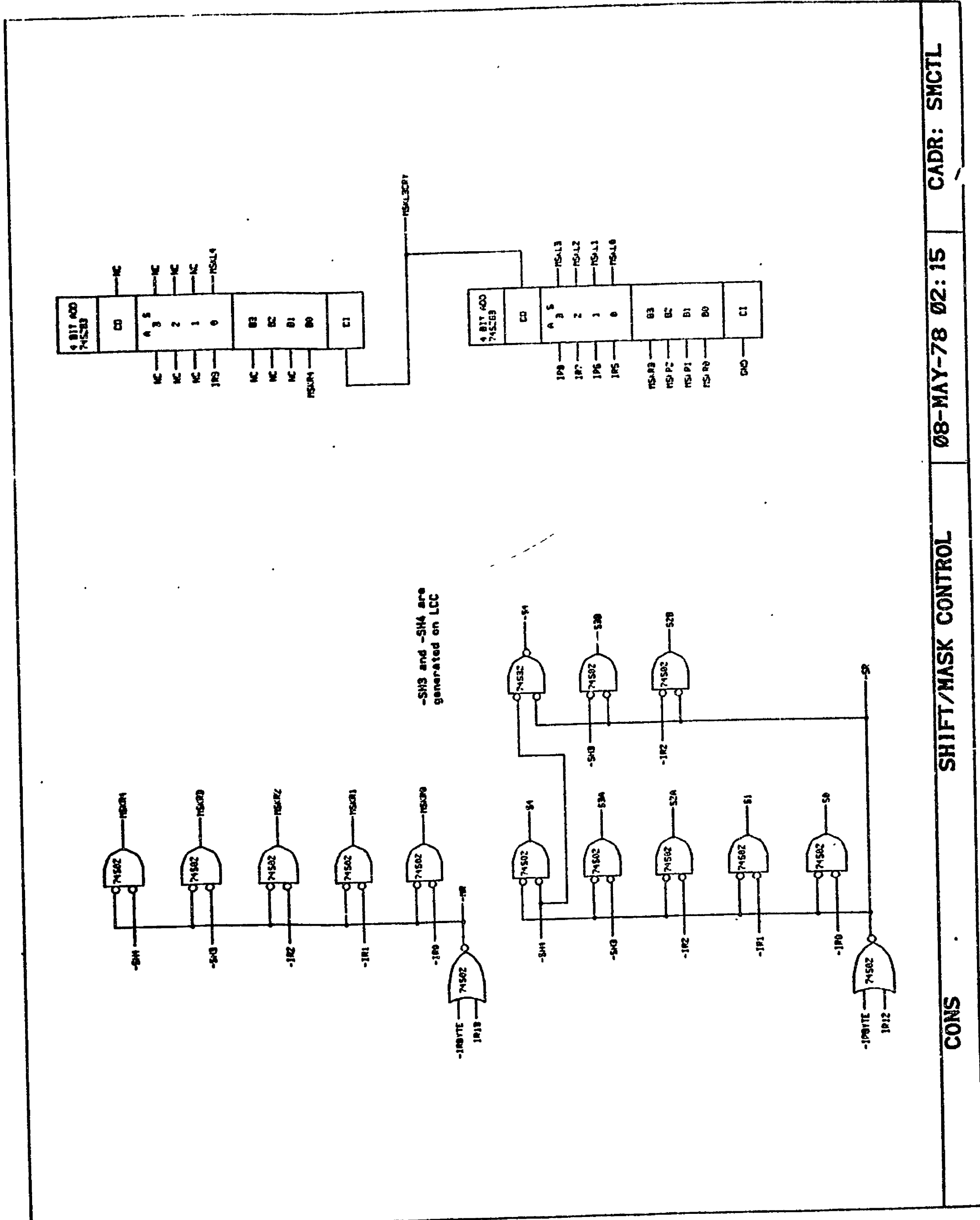
The output of the shifter (Rxx) is used to drive both the masker and the dispatch table address inputs.

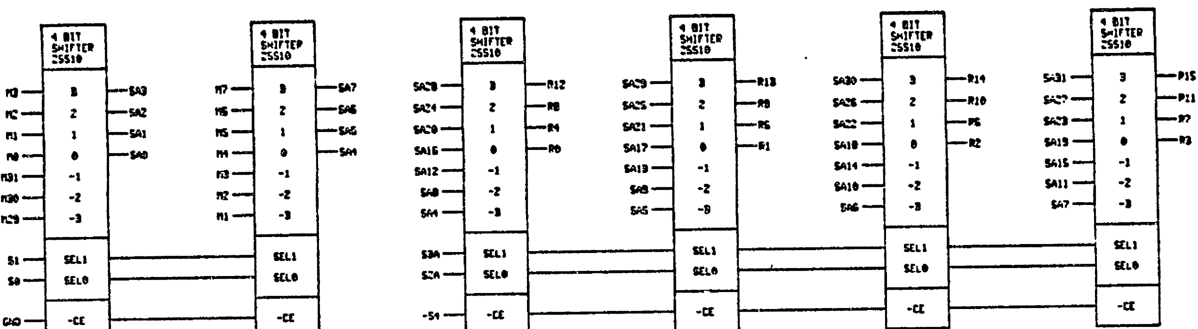
The masker performs a bitwise selection between the output of the rotator and the A bus data. The selection is based on a third 32 bit number, the mask (MSKxx). This number is produced from a set of two 32 x 32 programmable read only memories (print MSKGEN), whose contents are OR'd. Each set contains a triangular bit pattern, working from left to right in one set, and from right to left in the other.

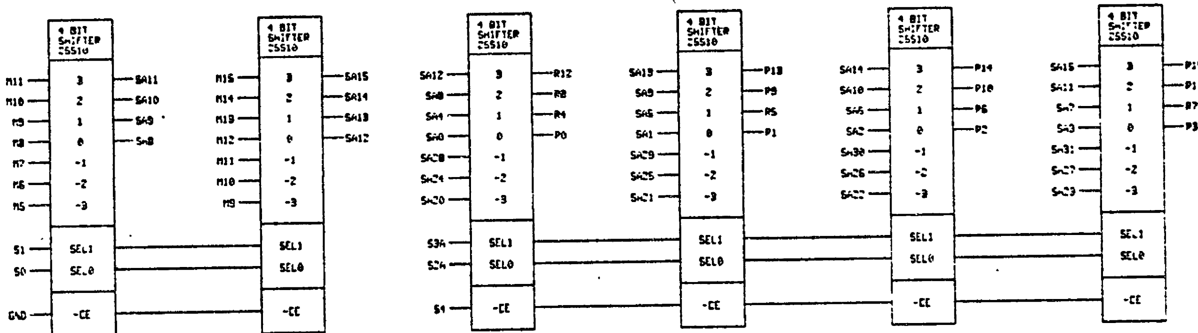
The address inputs to these proms represent the left and right bit positions of the selected field. The right bit position is either zero or the same as the input to the shift matrix, as controlled by bit 13 in a byte instruction.

The left bit position is always produced by adding the contents of IR9 to IR5 to the right bit position. Thus, these IR bits control the width of the masked field.

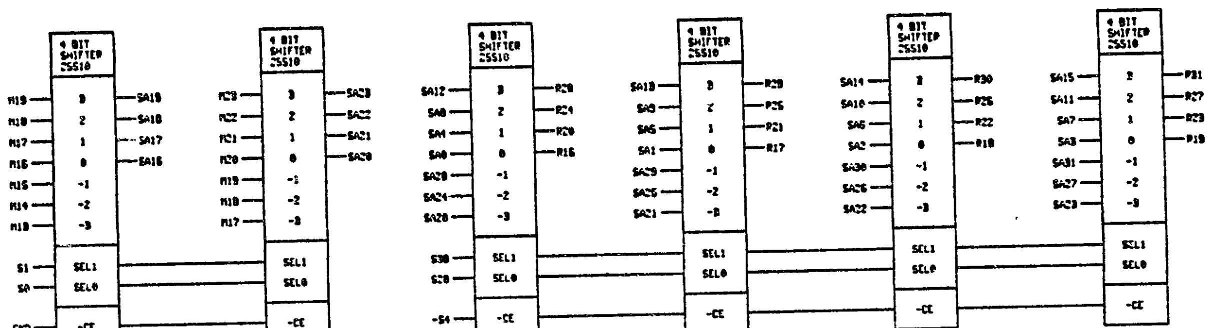
The actual masking operation requires both polarities of the mask, and is performed with the non-inverting and/or gate as shown on the MASK print. This is one of the more inefficient areas of the processor from a package count standpoint, primarily because there is no package efficient and fast device for implementing the bitwise select operation.

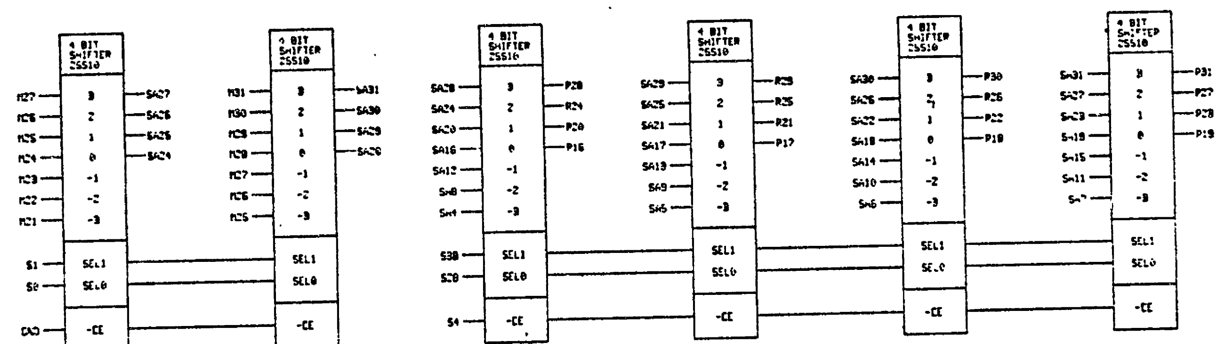

CONS
SHIFT/MASK CONTROL
Ø8-MAY-78 Ø2:15
CADR: SMCTL
-SH3 and -SH4 are generated on LCC
4 BIT ADD 74S283





CADR	SHIFTER RIGHT	22-JAN-78 23:46	CADR: SHIFTØ



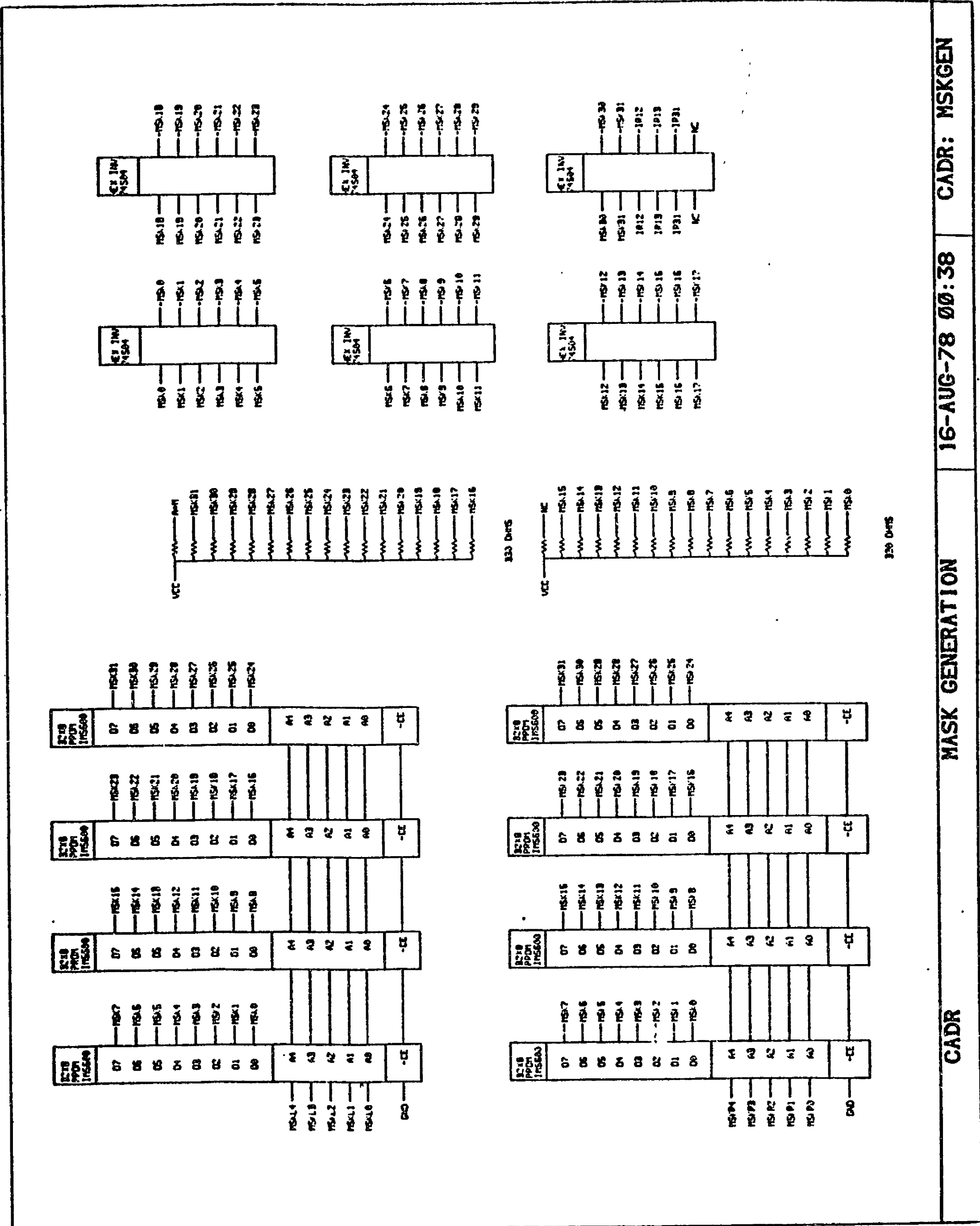


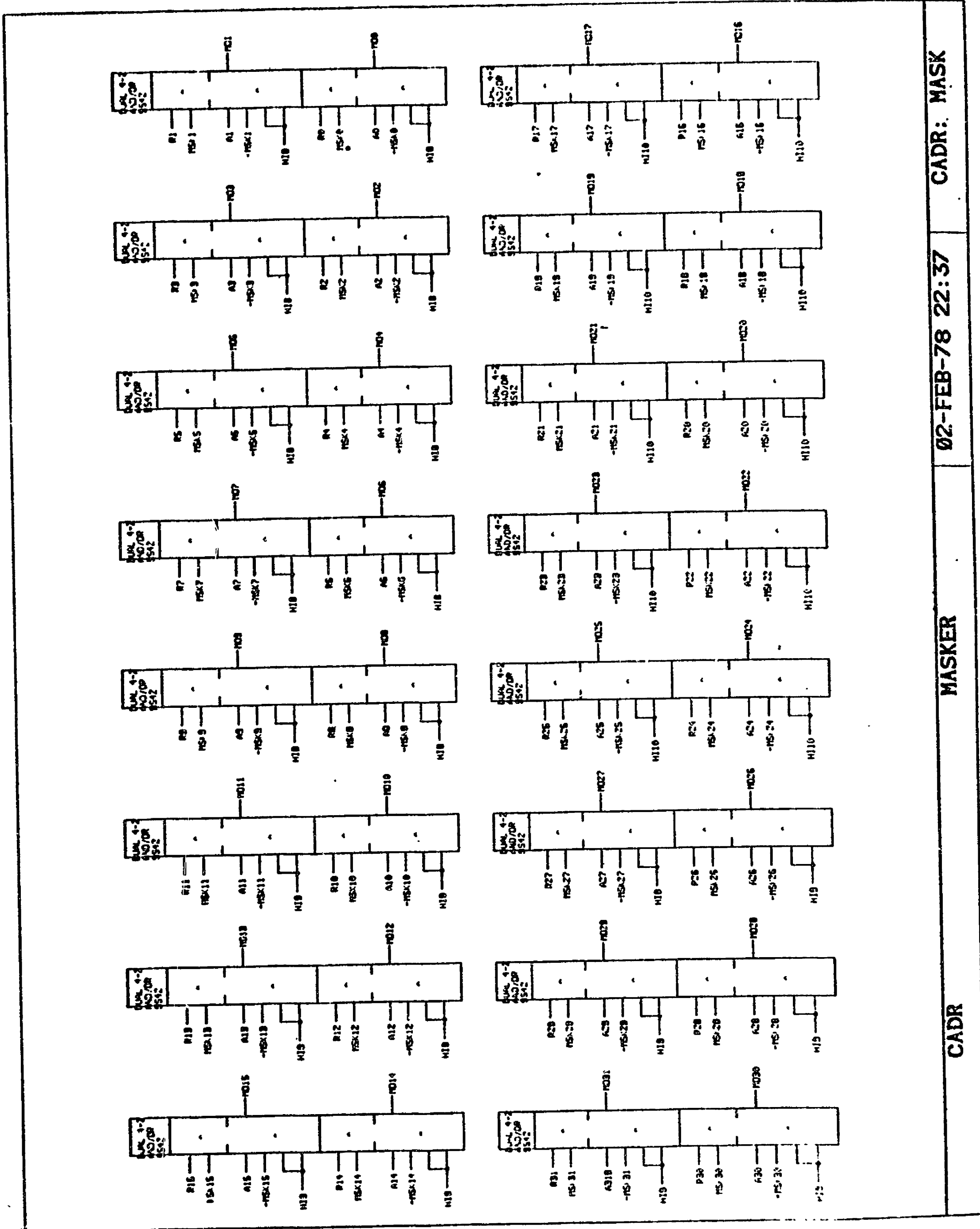
CADR

SHIFTER LEFT

22-JAN-78 23:48

CADR: SHIFT1

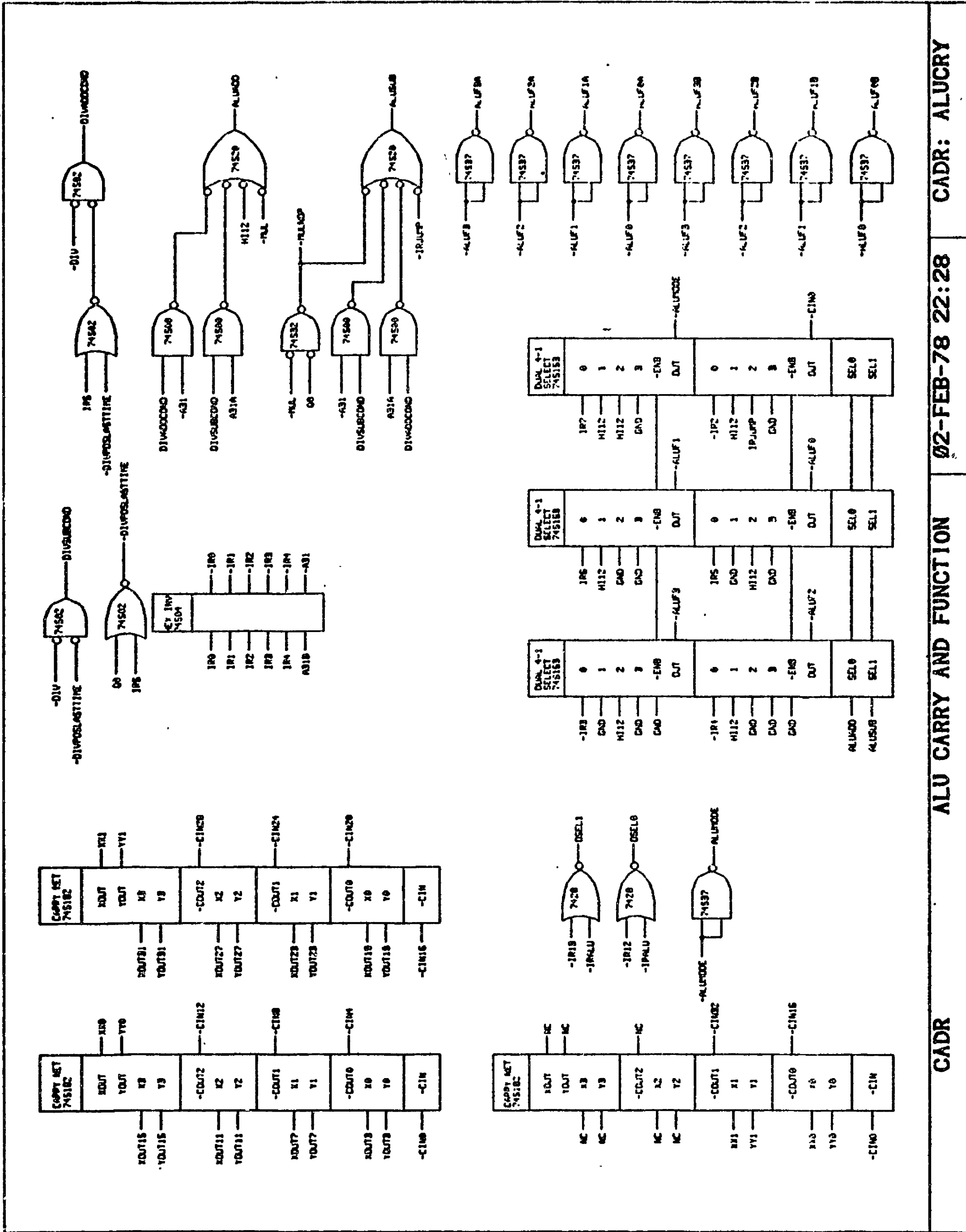

CADR
MASK GENERATION
16-AUG-78 00:38
CADR: MSKGEN

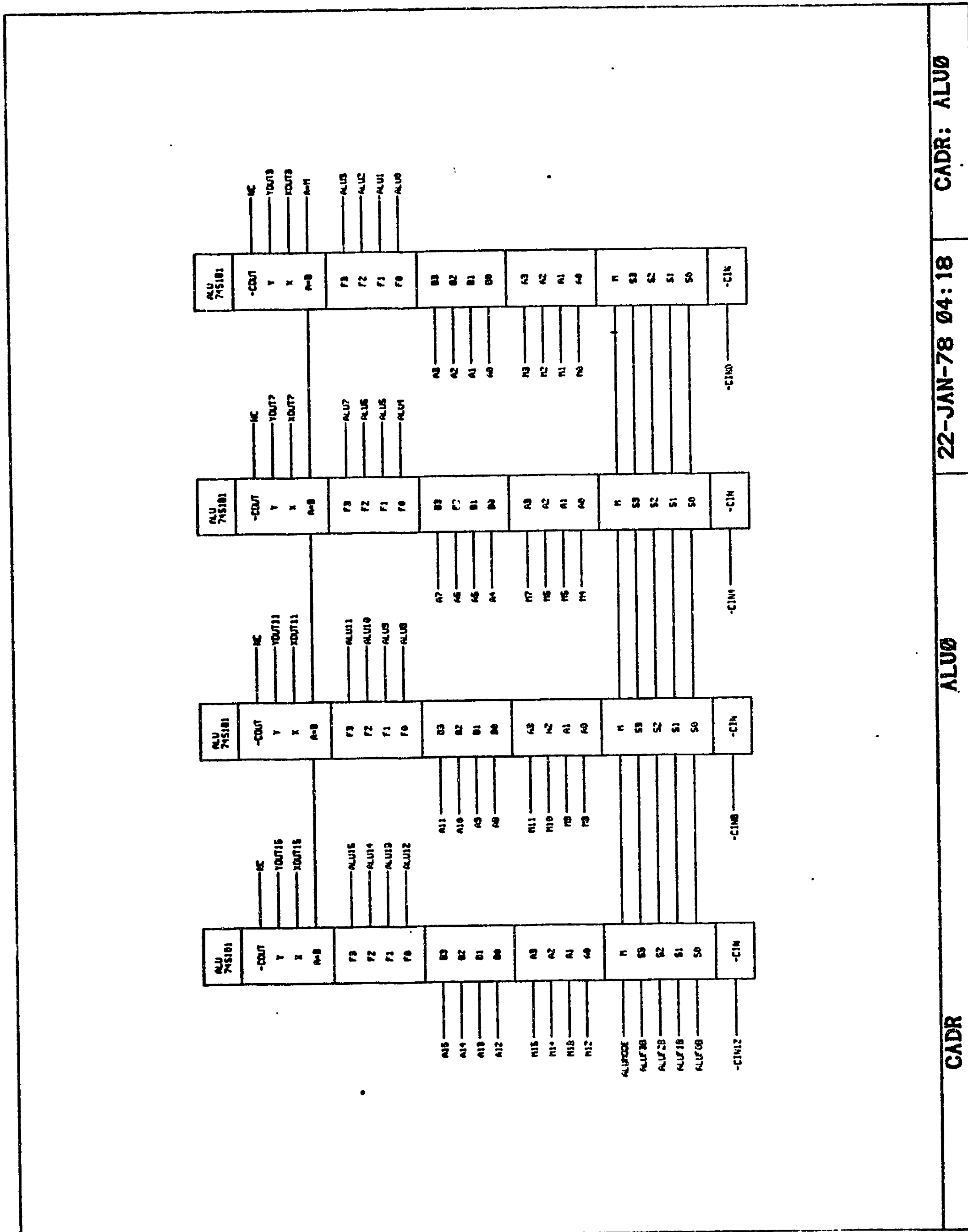

CADR: MASK
02-FEB-78 22:37
MASKER
CADR

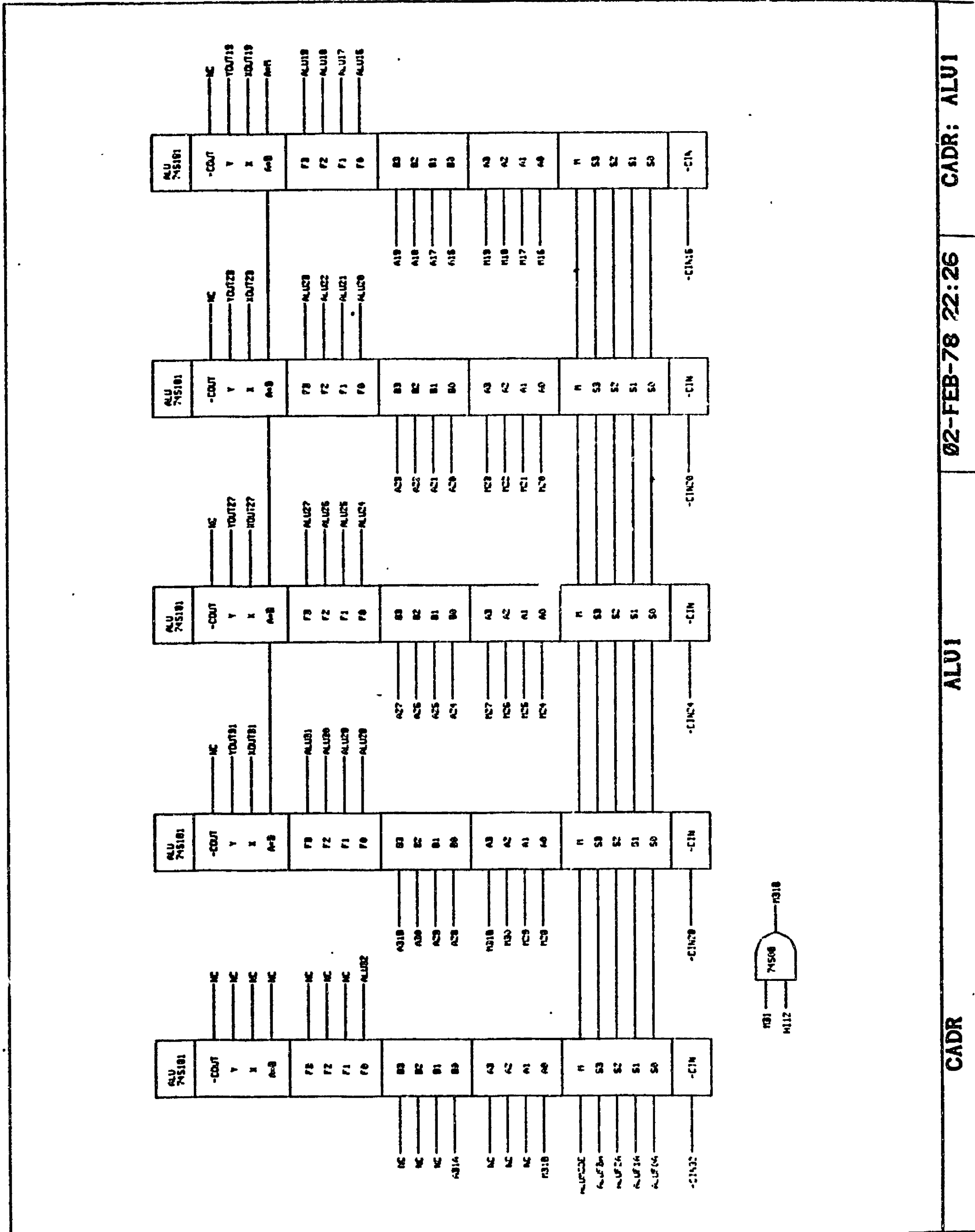
The ALU

The ALU performs the basic arithmetic in the machine. It consists of a set of nine 4 bit arithmetic chips, with two levels of fast carry extenders. The A and M busses drive the two inputs of the ALU directly. The function performed by the ALU is controlled by the set of selectors shown on the ALUCRY print. For the normal ALU instruction, the function, mode, and carry inputs are controlled by bits in the IR.

Three "hardwired" functions are also provided, by other selector inputs. One is a hardwired add of the A and M bus data, invoked by both multiply step and divide step under the appropriate circumstances (controlled by the ALUADD signal). Another is the subtract operation, invoked conditionally by the divide operation, and unconditionally by all jump instructions (ALUSUB). The jump instruction subtracts the A and M bus data, allowing an examination of the sign and equality condition code outputs of the ALU. The third hardwired function simply passes A memory data through, and is used as the NOP cycle during an inactive multiply step (both ALUSUB and ALUADD asserted).


CADR
ALU CARRY AND FUNCTION
Ø2-FEB-78 22:28
CADR: ALUCRY

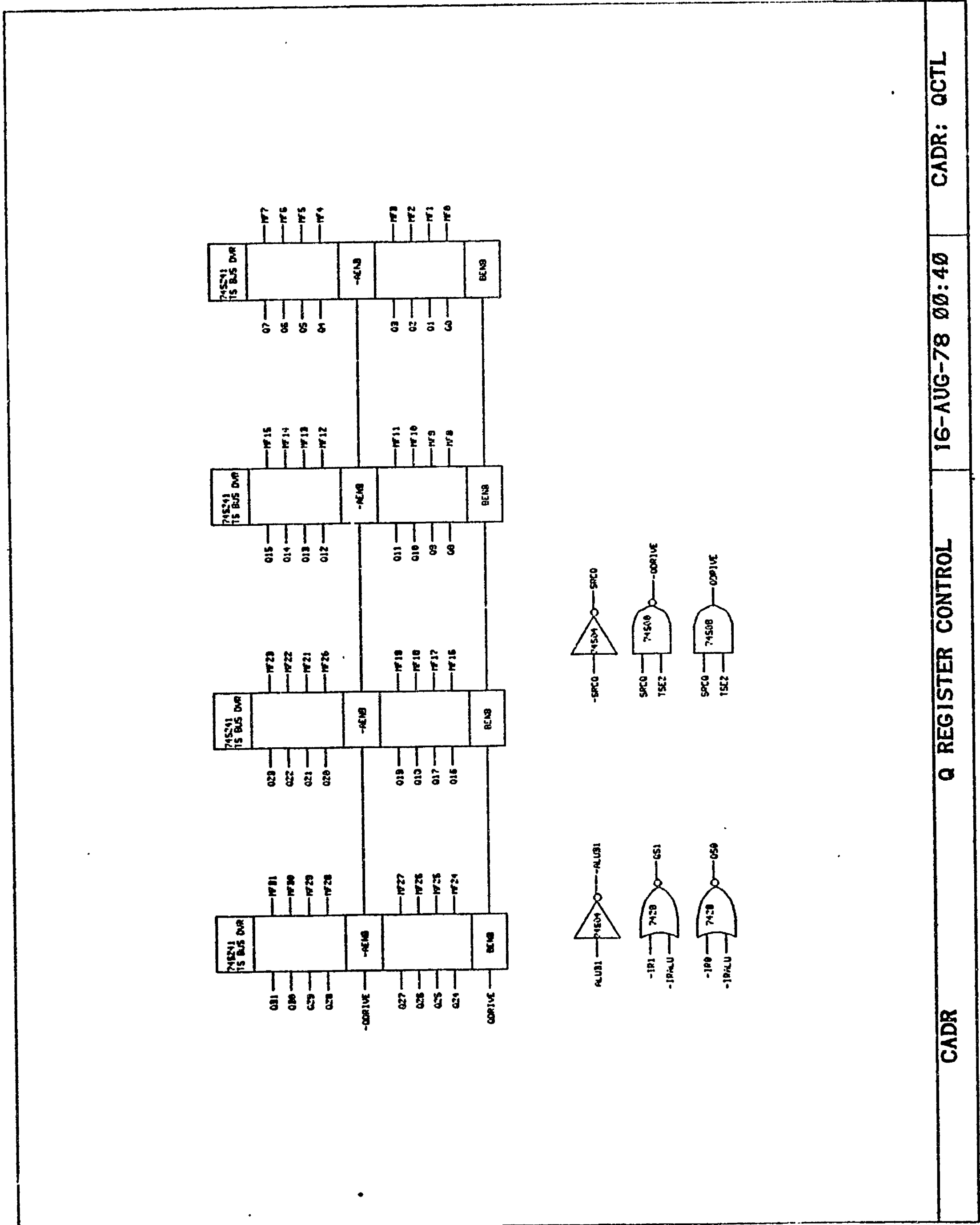

CADR: ALUØ
22-JAN-78 Ø4:18
ALUØ
CADR

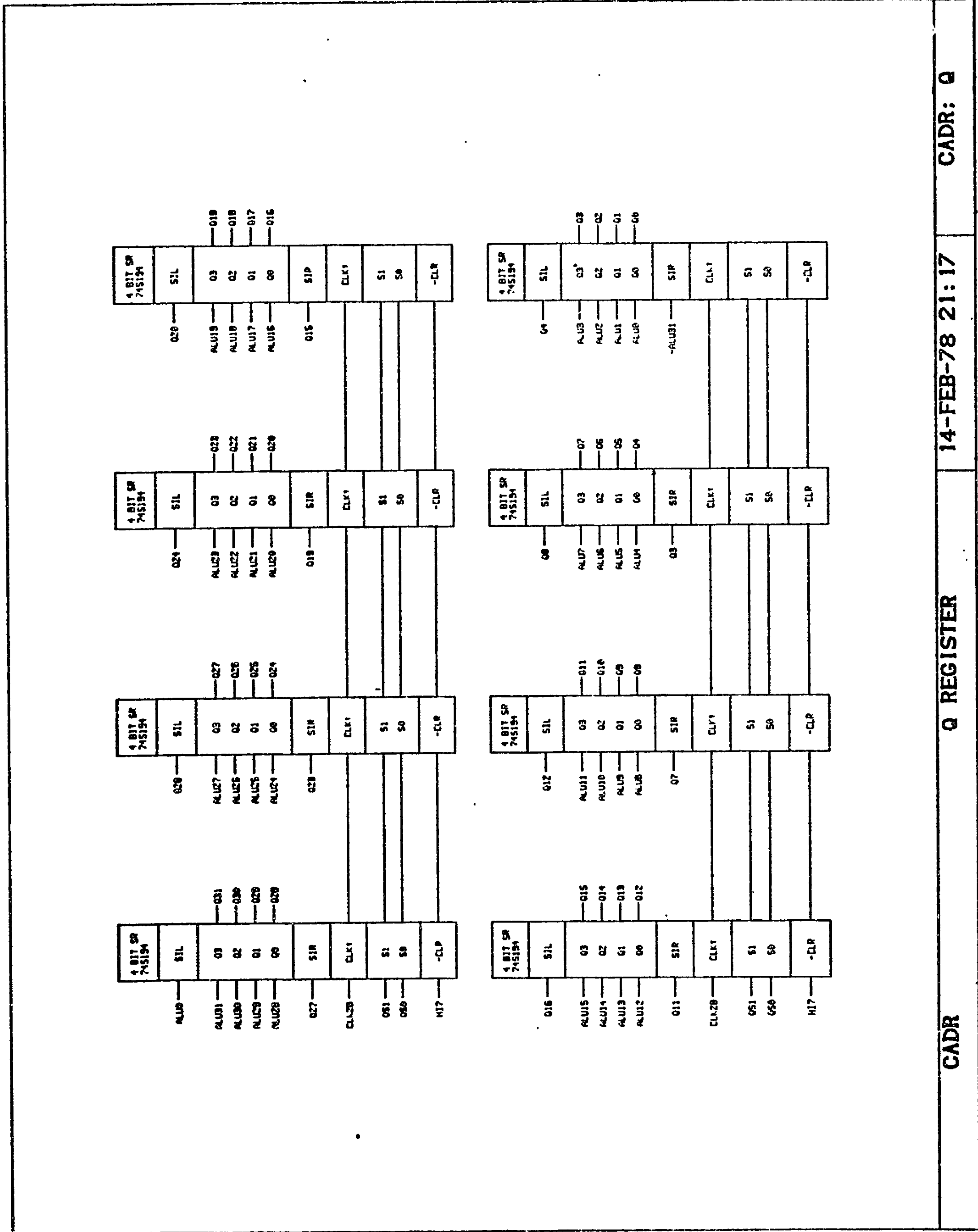

CADR: ALU1
Ø2-FEB-78 22:26
ALU1
CADR

The Q Register

The Q register forms the remainder of the multiply-step and divide-step hardware. It stores low order product bits as they are produced by multiply step, and is used to store both dividend and quotient bits during a divide step operation. It consists of a 32 bit shift register which can be loaded from the ALU output, shifted left, or shifted right. It is controlled by IR1 and IR0 in an ALU instruction (QCTL). It acts as a fast source on the M bus, and is altered on the rising edge of the main clock.

Shifts in from the left are from the low order bit of the ALU (useful for multiply step), while shifts in from the right are from the complement of the sign of the ALU output, which is the correct data for the partial quotient in a divide step.

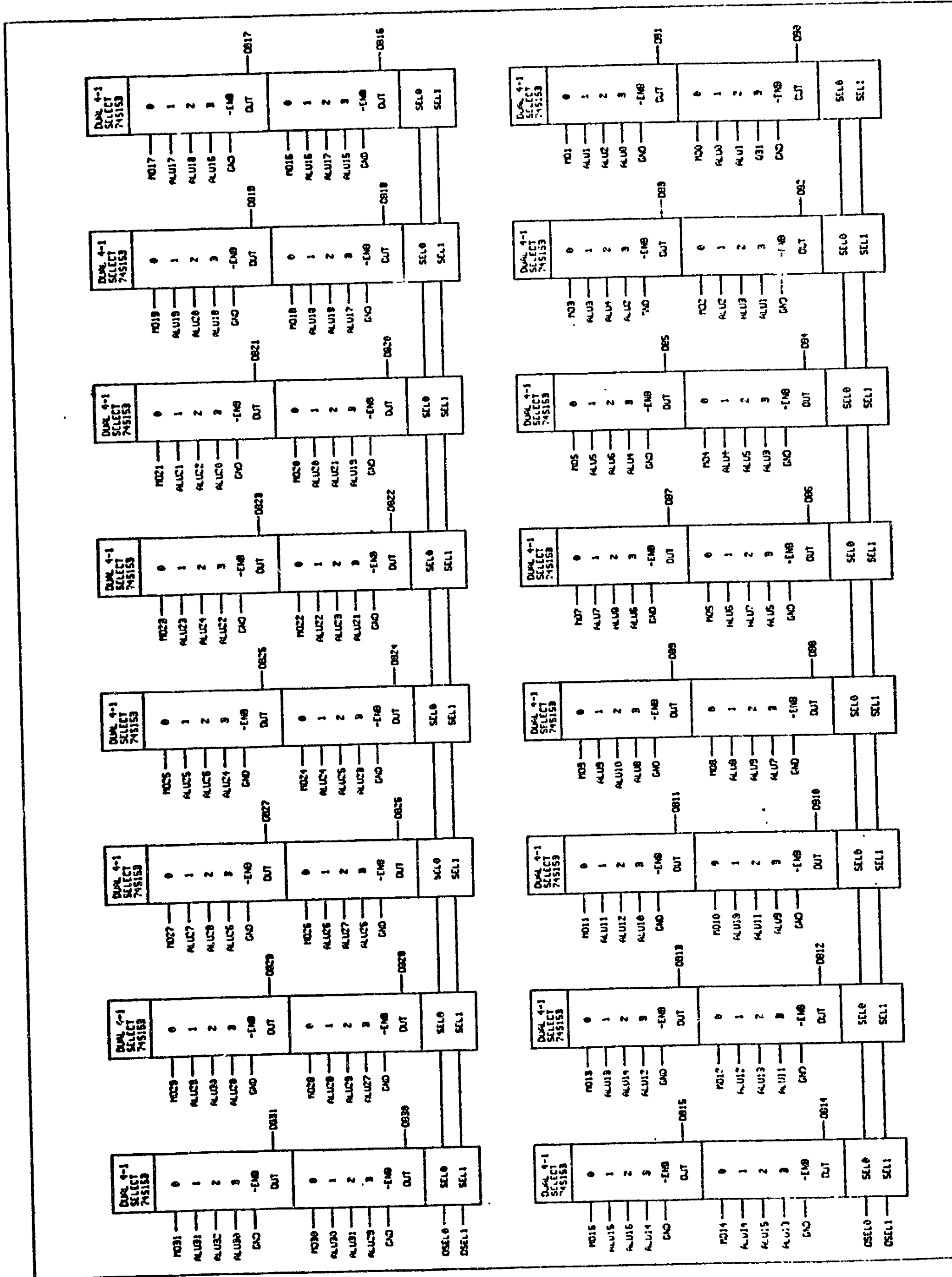

CADR: QCTL
16-AUG-78 ØØ:4Ø
Q REGISTER CONTROL
CADR


CADR: Q
14-FEB-78 21:17
Q REGISTER
CADR

The Output Bus

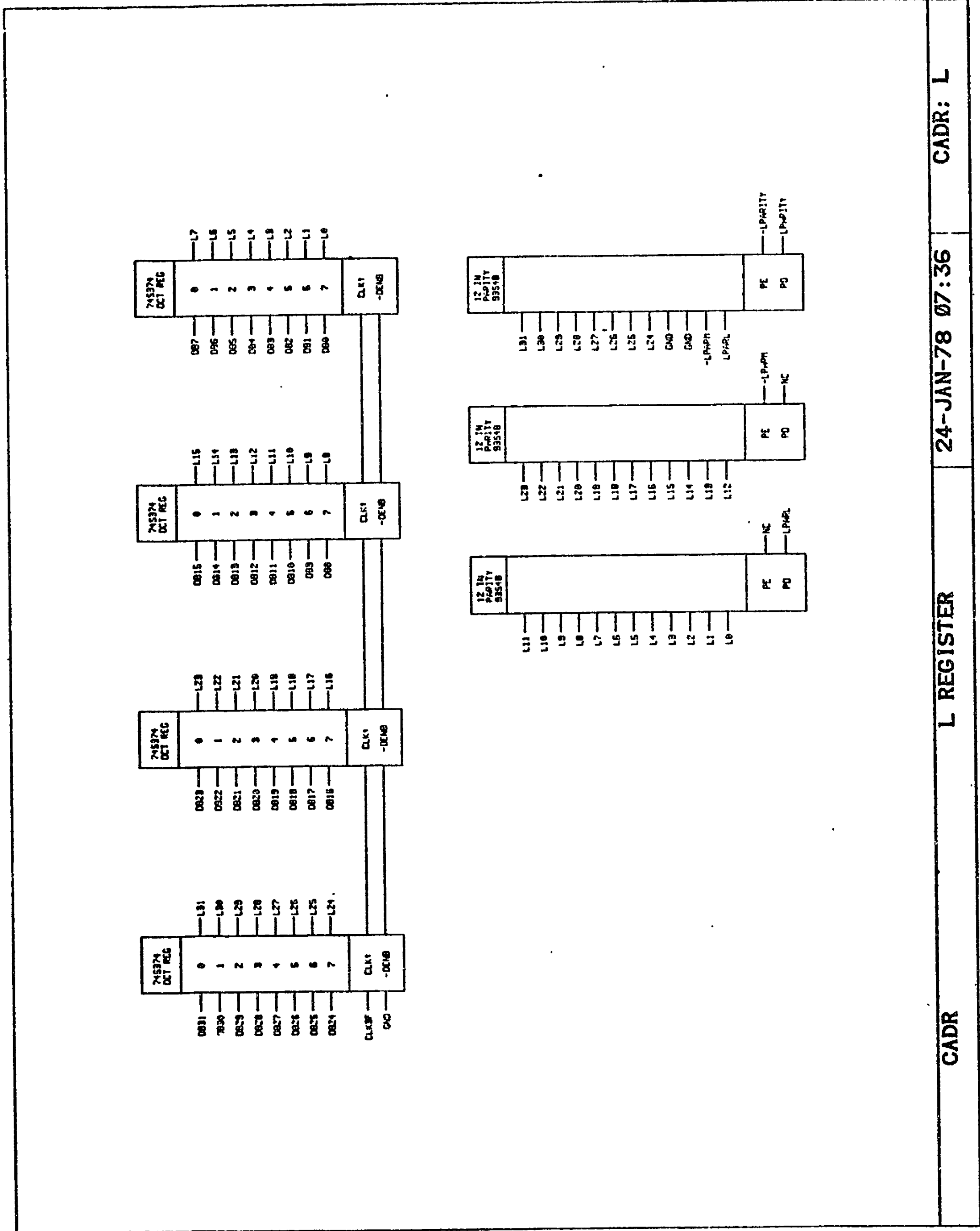
The primary result of either BYTE or ALU instructions is placed on the output bus (OBx). The output bus is heavily loaded, and drives all of the processor registers, except the Q, which are clocked on the rising edge of the main clock. During BYTE instructions, the output bus selector is forced to pass data from the masker (ALUCRY print). Otherwise, the selector is controlled by bits 12 and 13 in an ALU instruction. The selector may select either the unshifted ALU output, which is the normal case, or the ALU output shifted left by one, with the low bit coming from the high order bit of the Q, or the ALU output shifted right by one, with the high order bit coming from the one bit ALU extension.

The output bus bit shifting paths are again chosen to make the multiply step and divide step operations occur in a single cycle.


CADR
OUTPUT SELECTORS
22-JAN-78 23:11
CADR: OB

The L Register

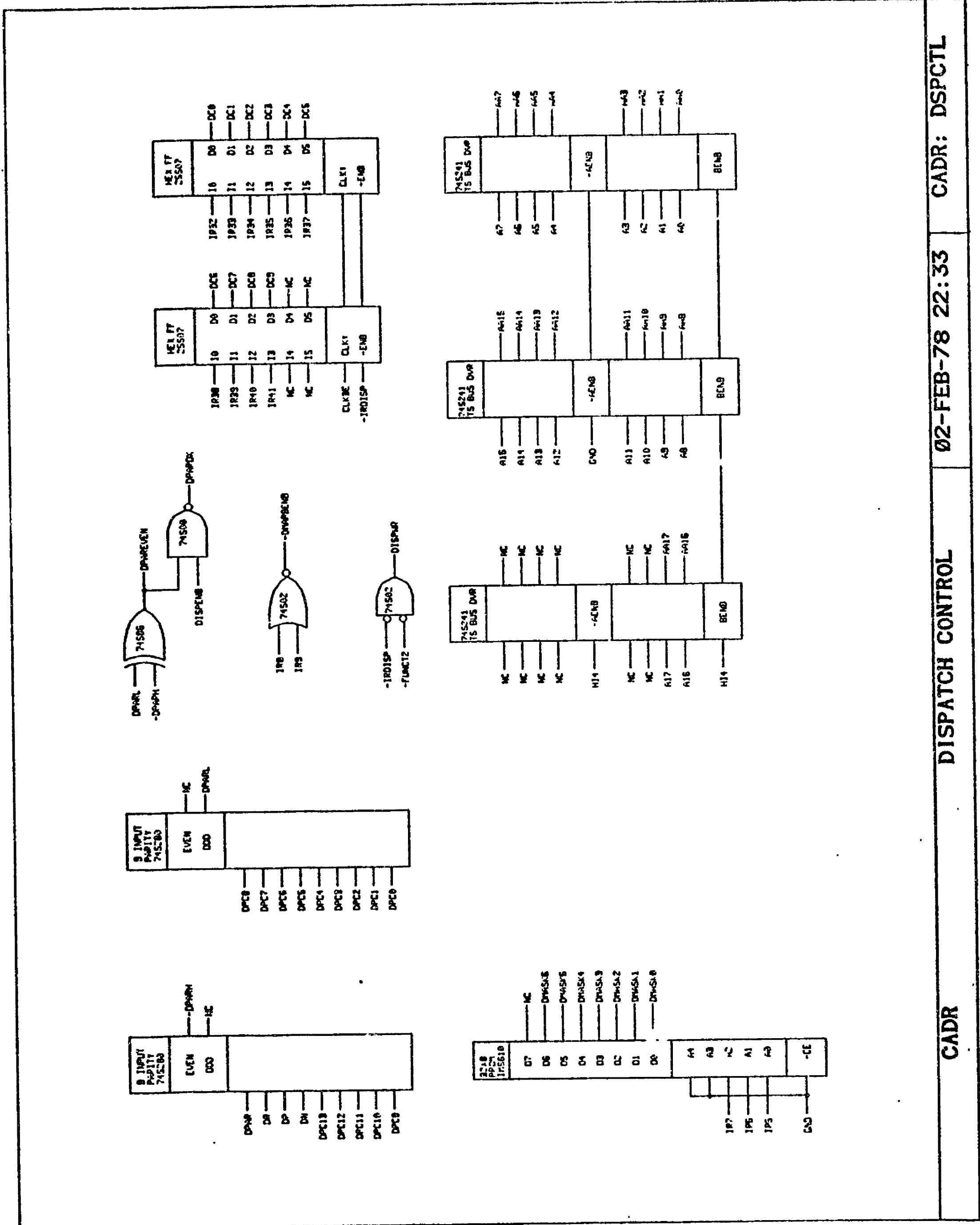
The L register provides a clocked version of the output bus data from the previous cycle. It is used as the source of write data for all of the memories which are written with write pulses in the second half of the succeeding cycle, and as a source of operand data on the A or M bus when the passaround logic is activated by sequential write-read instructions to the same memory location.

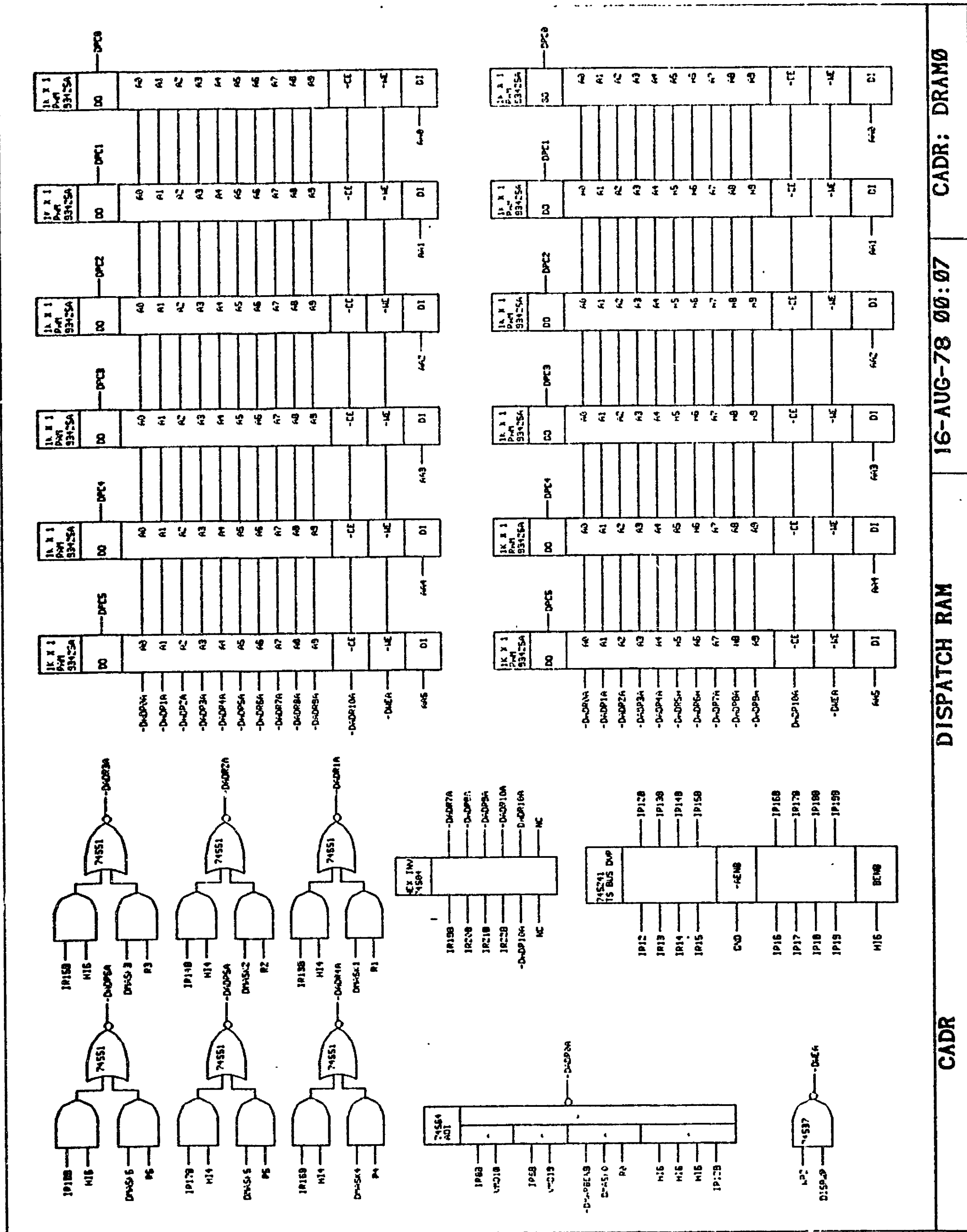

CADR: L
24-JAN-78 Ø7:36
L REGISTER
CADR

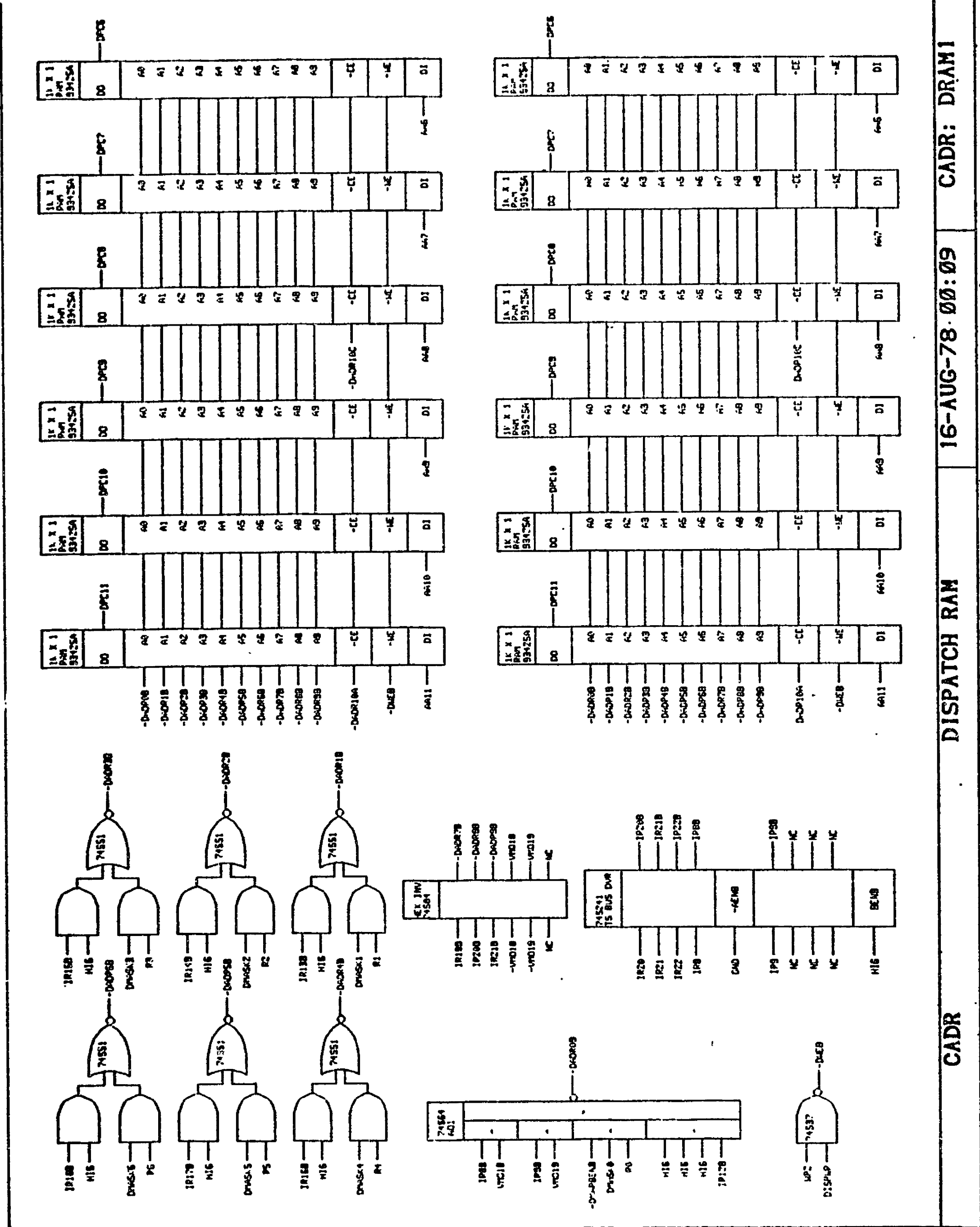
The Dispatch Memory

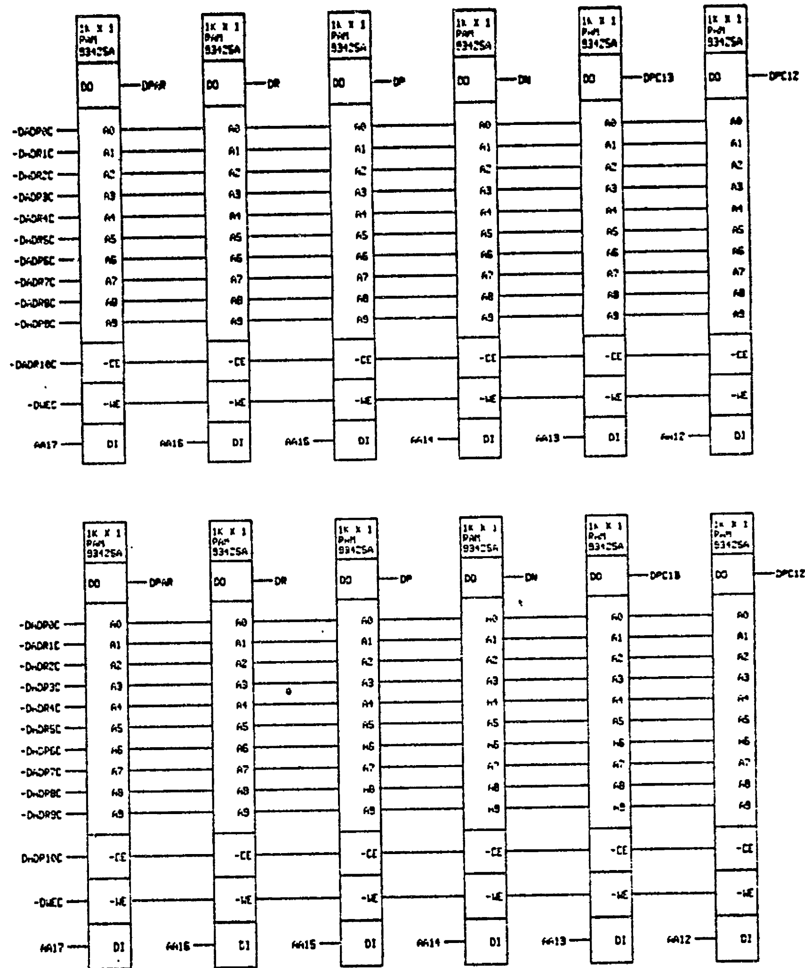
The dispatch memory is used to produce new program counter values as a result of data flowing through the processor. Outputs of the shifter (Rxx) are AND'd with a dispatch byte length mask, and OR'd with an eleven bit field from the IR to form the dispatch memory address, DADRxx (left hand side of DRAM0,1,2). Bit zero of the dispatch address additionally may be taken from the output of the memory map (see below). The mask used for selecting the dispatch byte length is located on the DSPCTL print, as is the buffers for A bus data which is used for writing into dispatch memory. The dispatch memory, unlike all other memories in the machine, is loaded in the same instruction which specifies the write, since its output is needed late in the cycle and cannot be latched as can the other processor memory outputs.

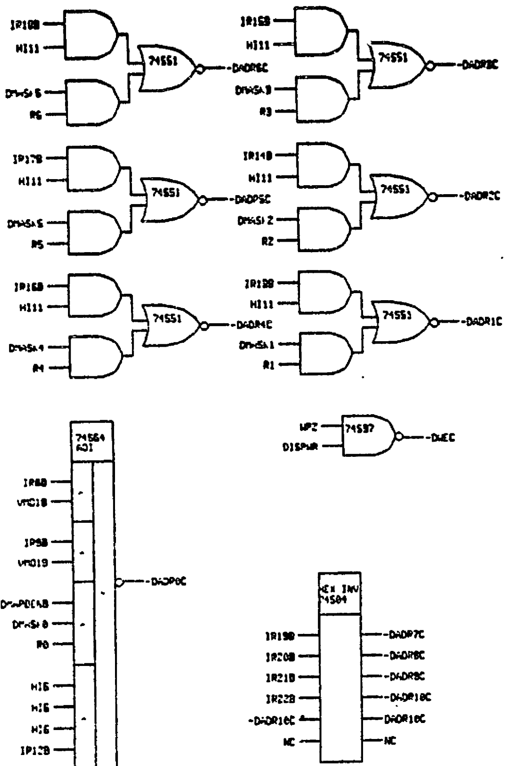
Output of the dispatch memory is a new 14 bit program counter value and three additional bits, DN, DP, and DR, which control the type of transfer performed (see control discussion below).


CADR: DSPCTL
02-FEB-78 22:33
DISPATCH CONTROL
CADR


CADR
DISPATCH RAM
16-AUG-78 ØØ:Ø7
CADR: DRAMØ


CADR
DISPATCH RAM
16-AUG-78 ØØ:Ø9
CADR: DRAM1





CADR

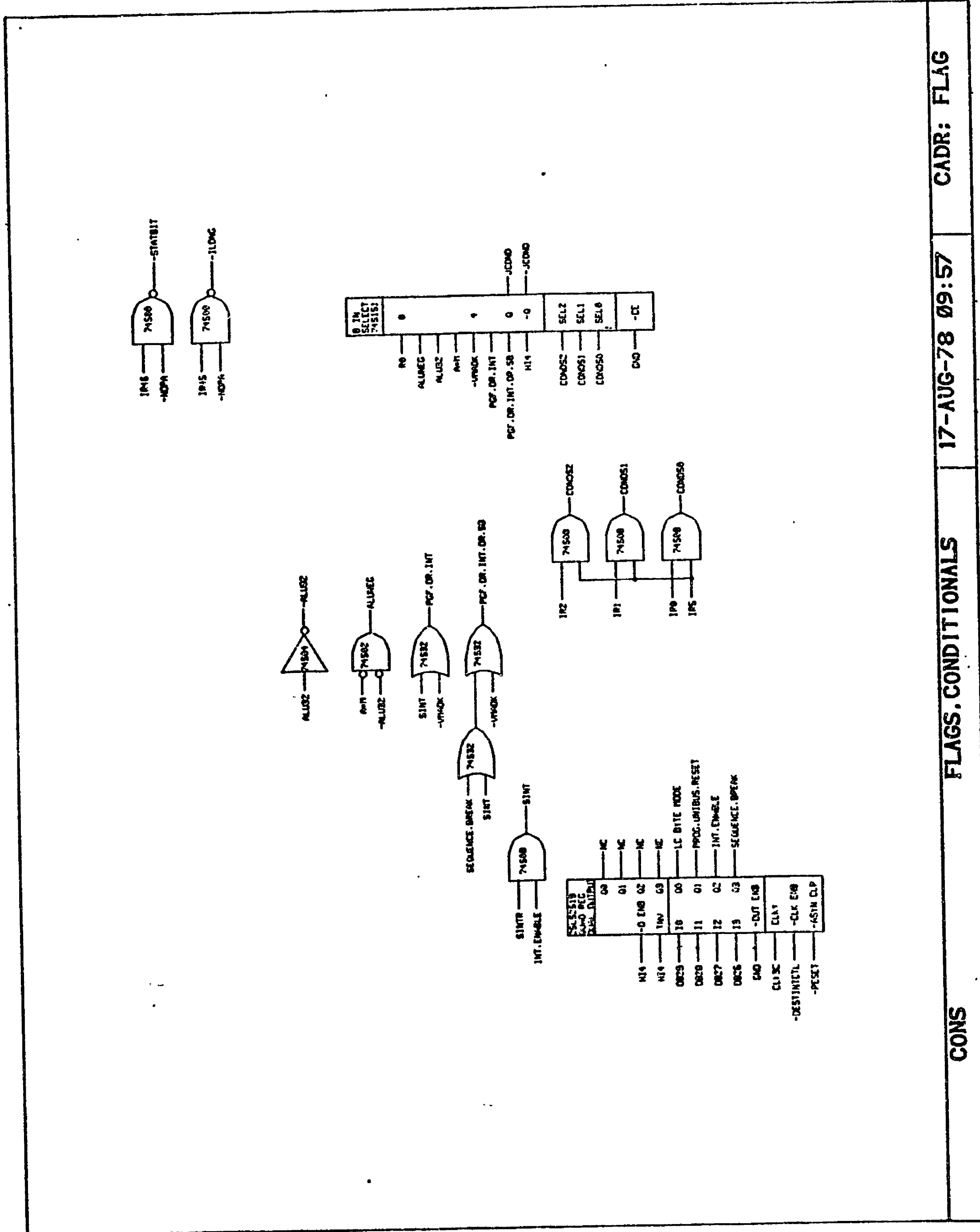
DISPATCH RAM

16-AUG-78 00:10

CADR: DRAM2

Jump Conditions

The FLAG print shows the jump condition selector, which is used to determine what controls the success of a conditional jump. If IR bit 5 is zero, the low bit of the rotator output (R0) is selected, and the field IR4-0 is used for controlling the rotator. Otherwise, bits 2-0 control the jump selector, which examines the ALU sign and equal zero outputs in various combinations to provide arithmetic compares. The selector also has access to the page fault condition (-VMAOK) and combinations of it with interrupt and sequence break conditions, so that these combinations may be easily tested in a single cycle.

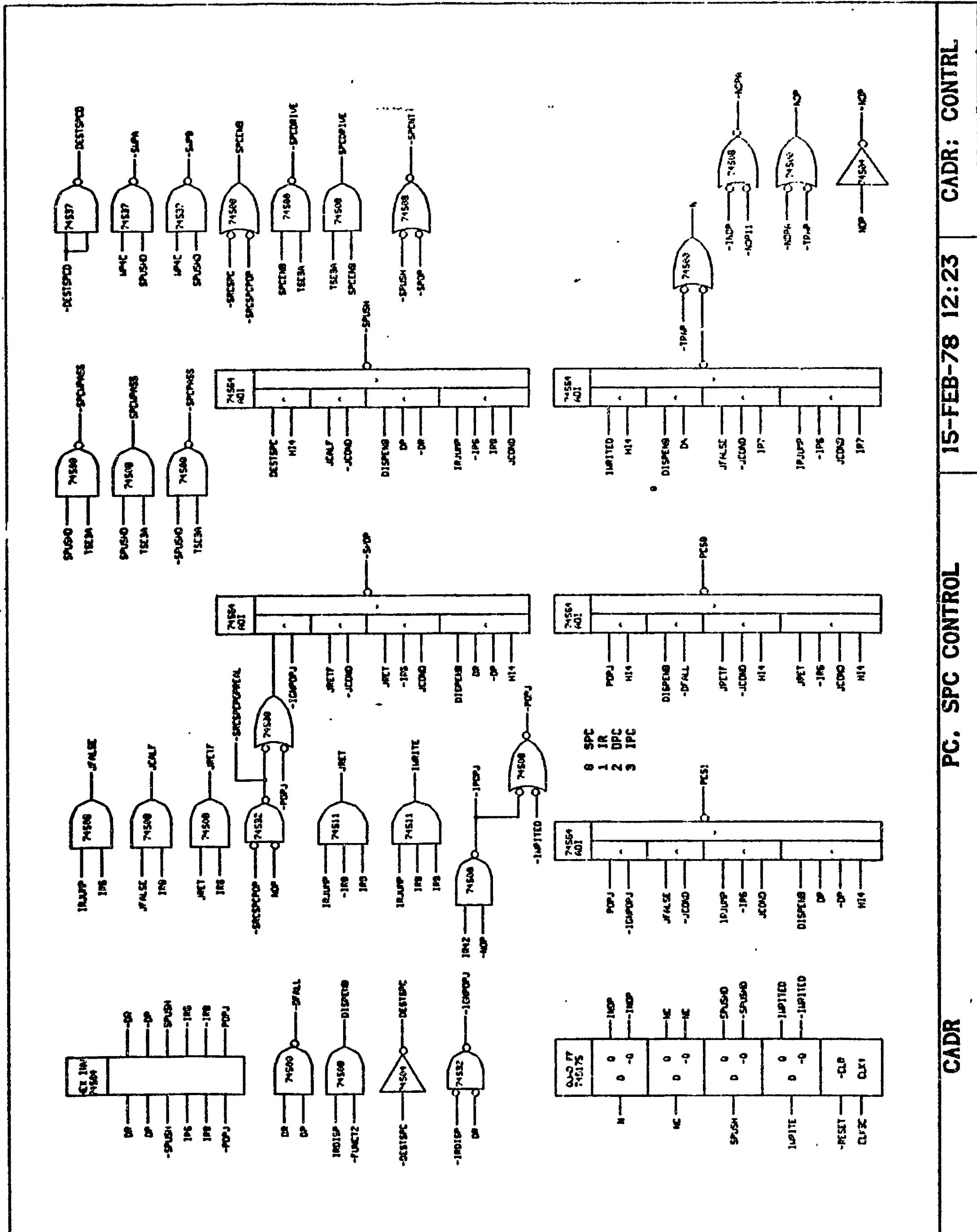

CADR: FLAG
17-AUG-78 09:57
FLAGS.CONDITIONALS
CONS

Flow of Control

The CONTRL print implements the different types of jump and dispatch transfers. Three sources influence the transfer of control: the jump instruction data, the dispatch instruction data, and the POPJ-AFTER-NEXT bit.

The jump instruction is further decoded into special cases prior to the availability of the condition output. For the PC source, the jump selects between IPC (no jump), IR (jump or call) and SPC data (return). If the transfer is taken, the value of IR7 is selected for N, which controls execution of the following instruction.

The dispatch instruction similarly specifies with DP and DR, the PC source and N outputs, selecting the new PC from DPCxx (output of the dispatch memory), IPC (fall through) or SPC (return). The N bit is driven directly from the DN output of the dispatch memory.

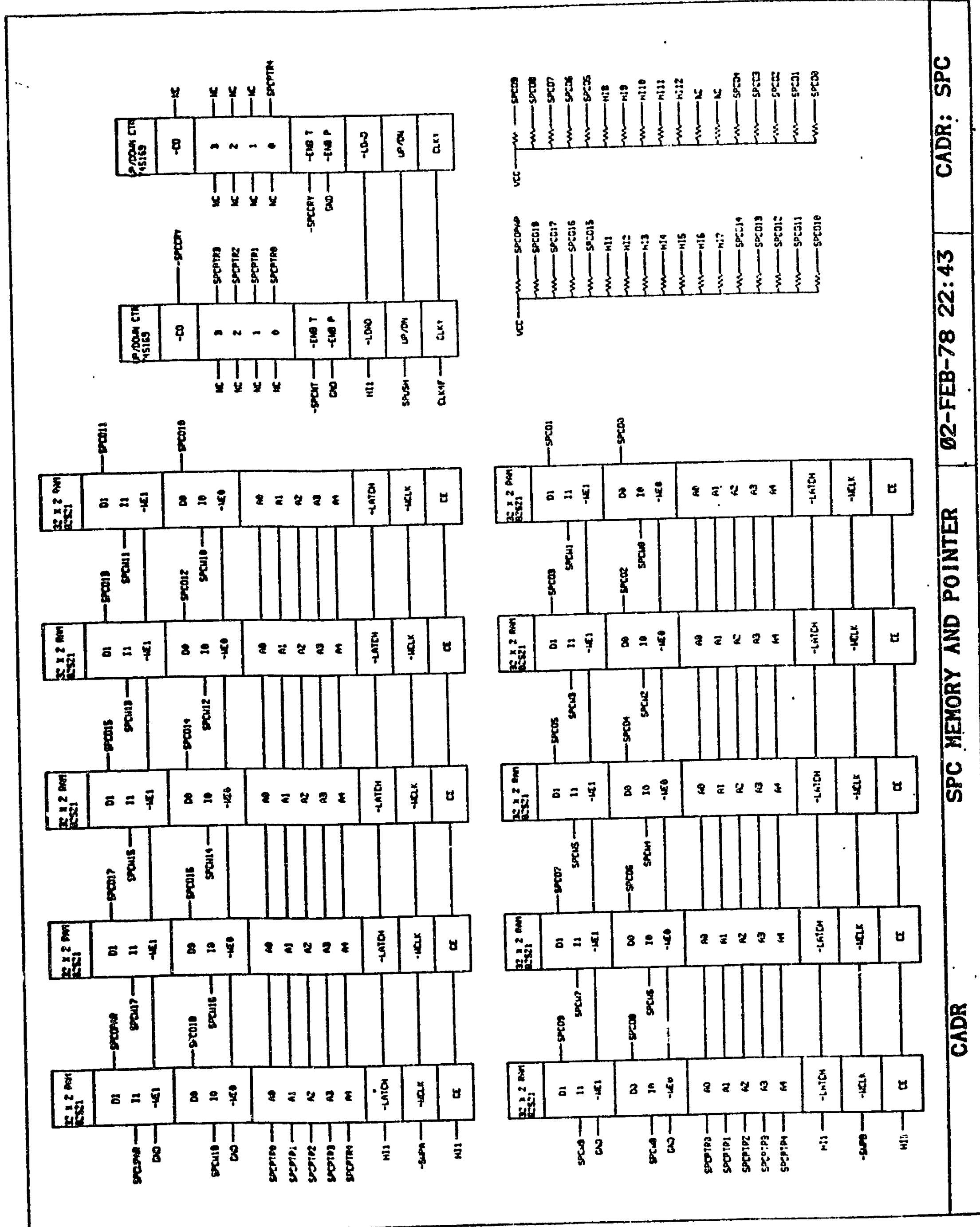

CADR
PC, SPC CONTROL
15-FEB-78 12:23
CADR: CONTRL

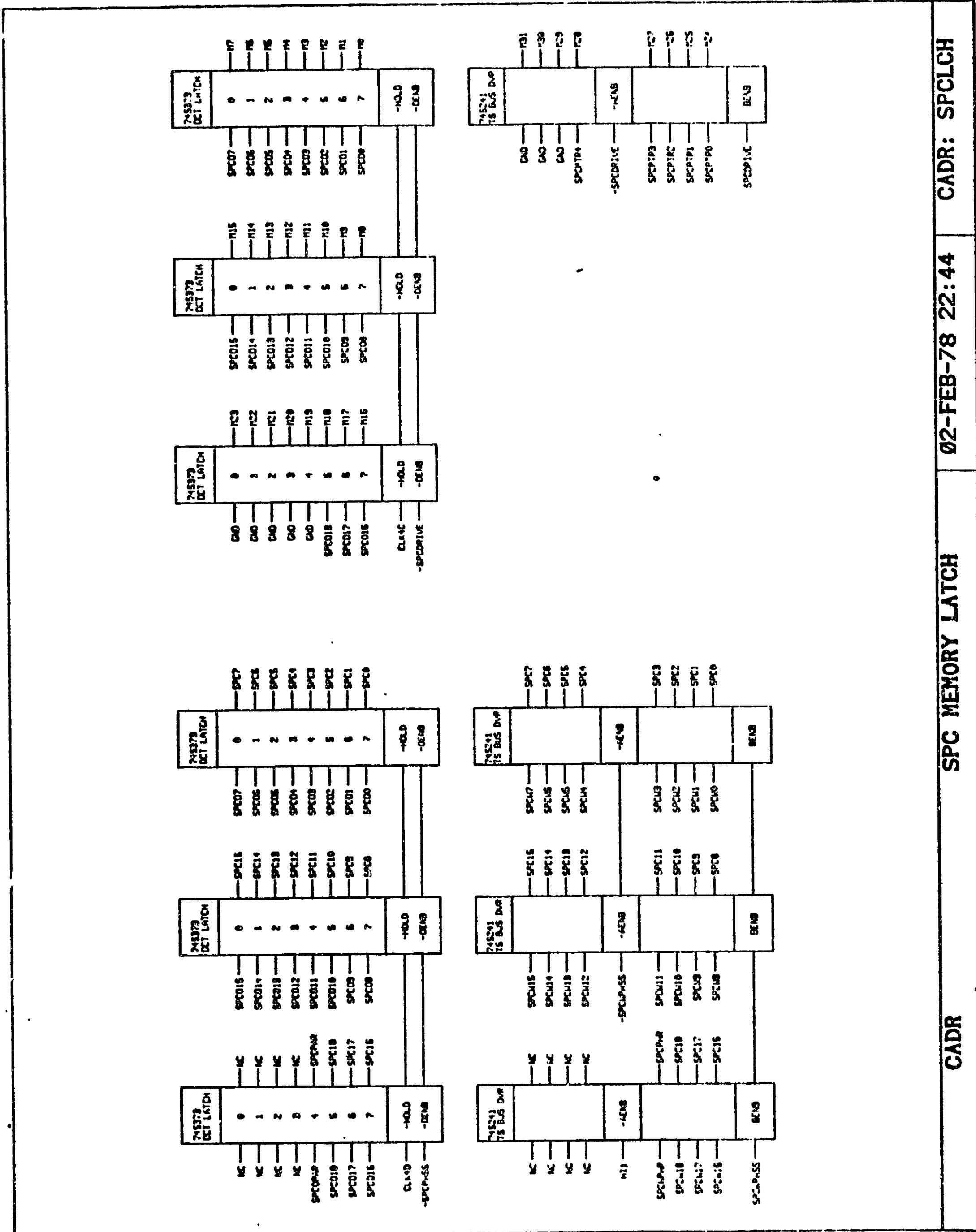
Microcode Subroutine Return Stack

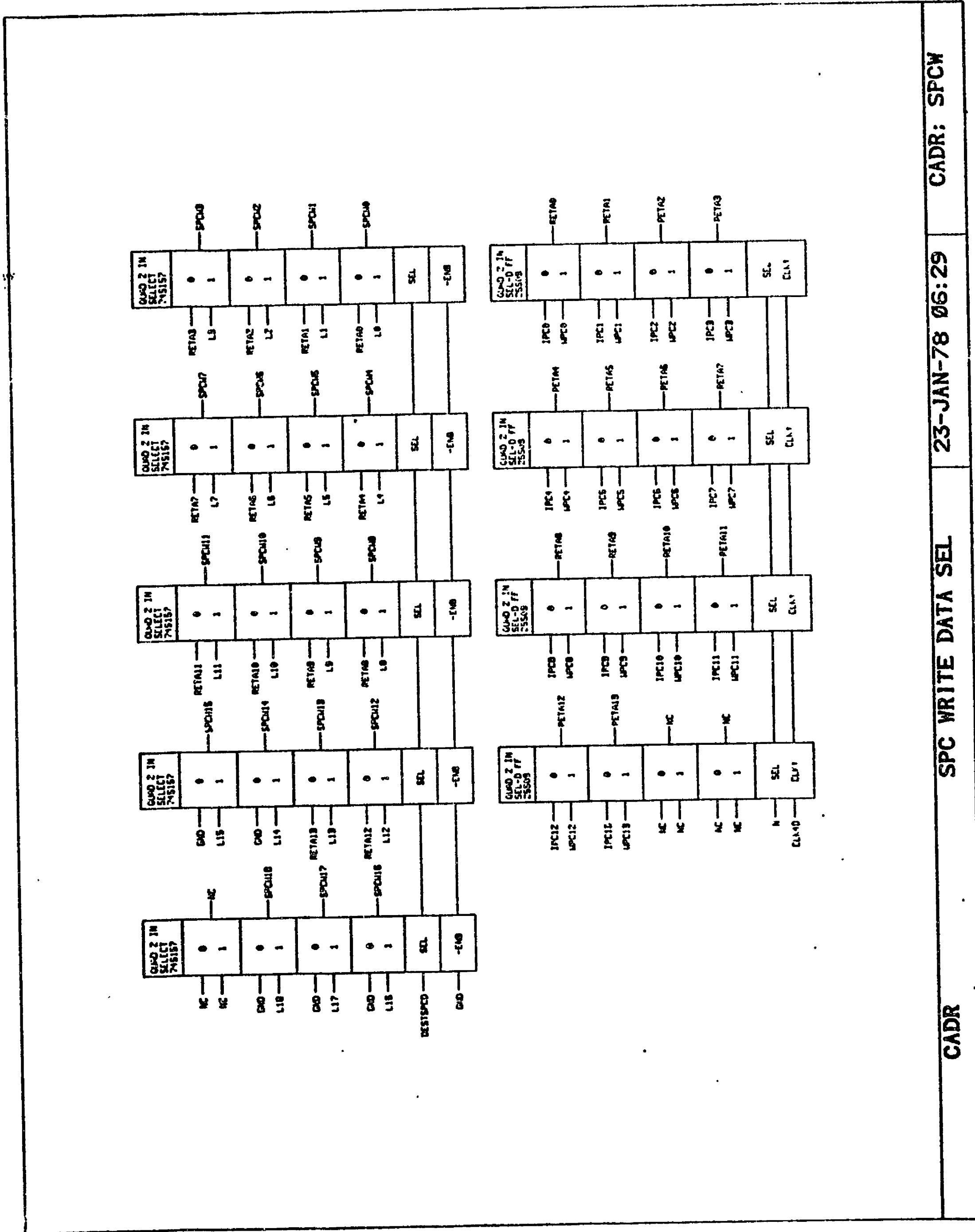
The return address for micro subroutine calls is kept in the 32 entry SPC memory. This memory is addressed by a five bit counter, the SPCPTR. On micro subroutine calls, this counter is advanced, and the subroutine return address is written into the memory. This address may be one of four quantities, the current PC, the incremented PC, the previous PC, or the L register contents.

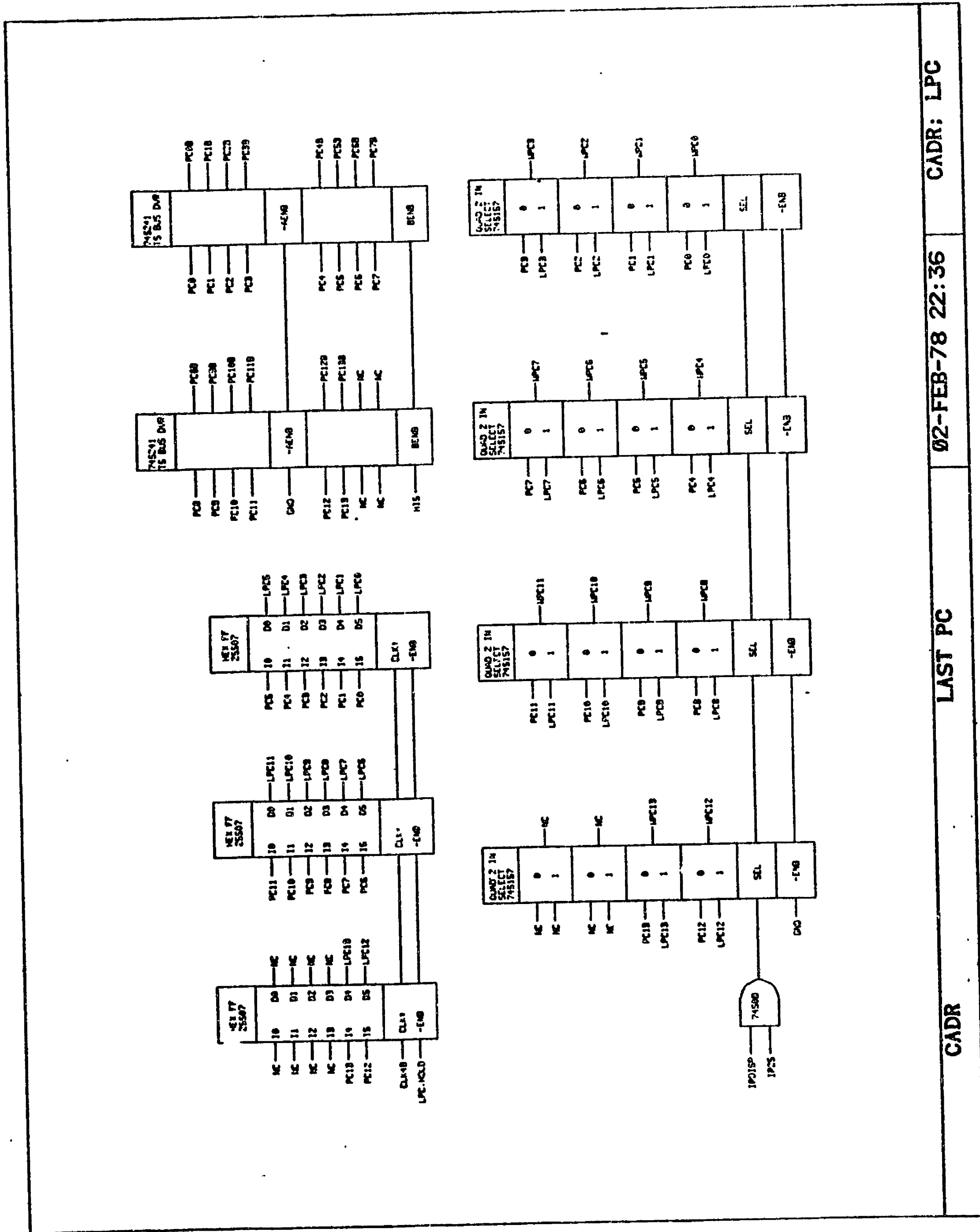
The decision as to which PC to write on a call is based on whether the call specified execution of the following instruction (PC or IPC) and on whether it is an error catching dispatch, with the intention of allowing retry (LPC). Writing into the memory from the L register allows normal restoration of processor state information, as well as allowing the machine to set the special flags in the left half of the SPC register. One of these flags activates the macroinstruction prefetch mechanism when a return is executed with that bit set.

The data contained in the SPC memory is read in two independent paths, one driving the M bus, and the other driving the inputs to the next PC selector (NPC). This allows the stack data to be read from the main data paths, and to be used as a return address in jump and dispatch instructions. Pass around paths are provided on the SPC output path to the NPC selector, but are missing on the M memory path, since the data is available from other registers in the machine after an SPC write cycle.


CADR
SPC MEMORY AND POINTER
Ø2-FEB-78 22:43
CADR: SPC

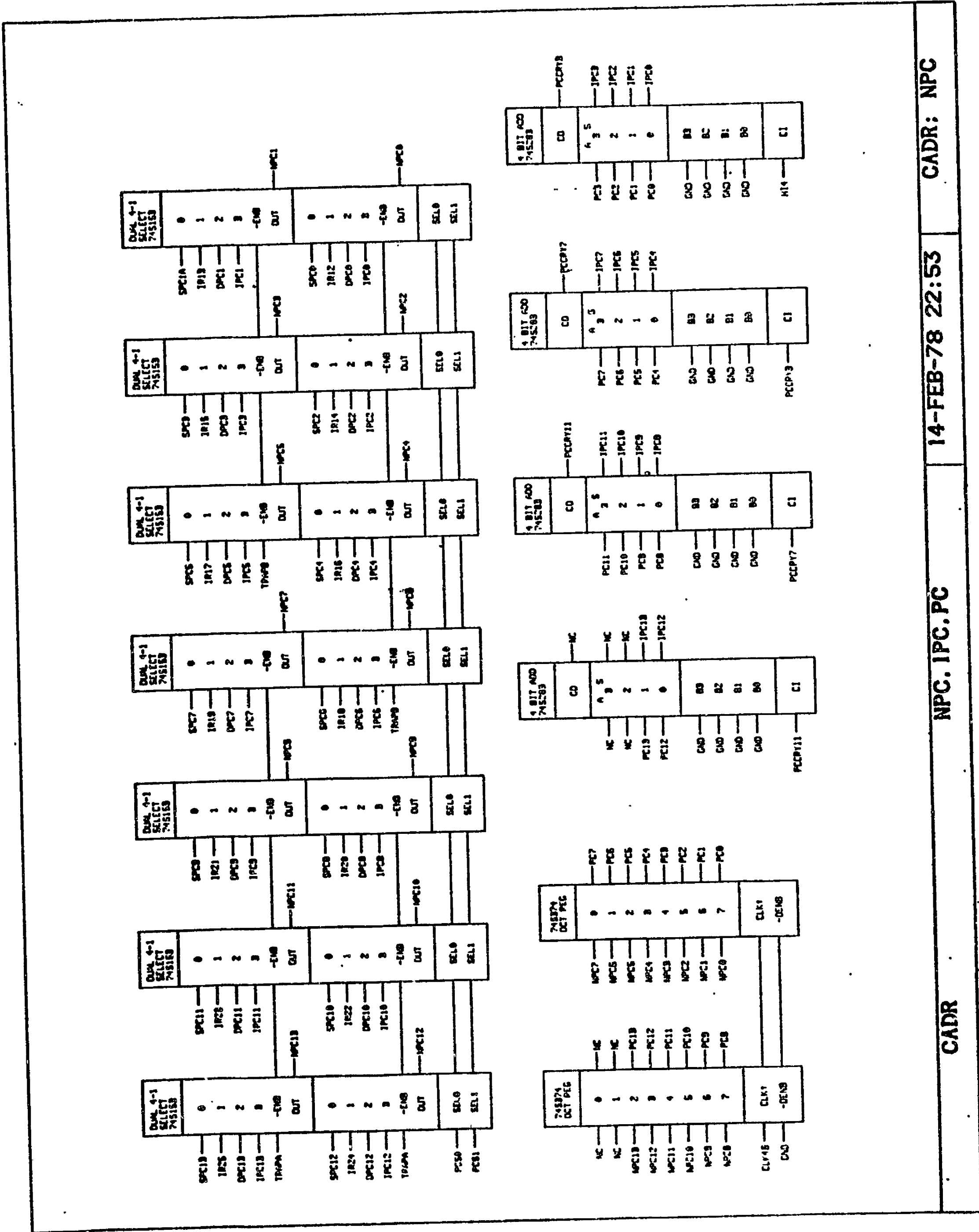

CADR
SPC MEMORY LATCH
02-FEB-78 22:44
CADR: SPCLCH


CADR
SPC WRITE DATA SEL
23-JAN-78 06:29
CADR: SPCW


CADR: LPC
Ø2-FEB-78 22:36
LAST PC
CADR

Next PC Selector

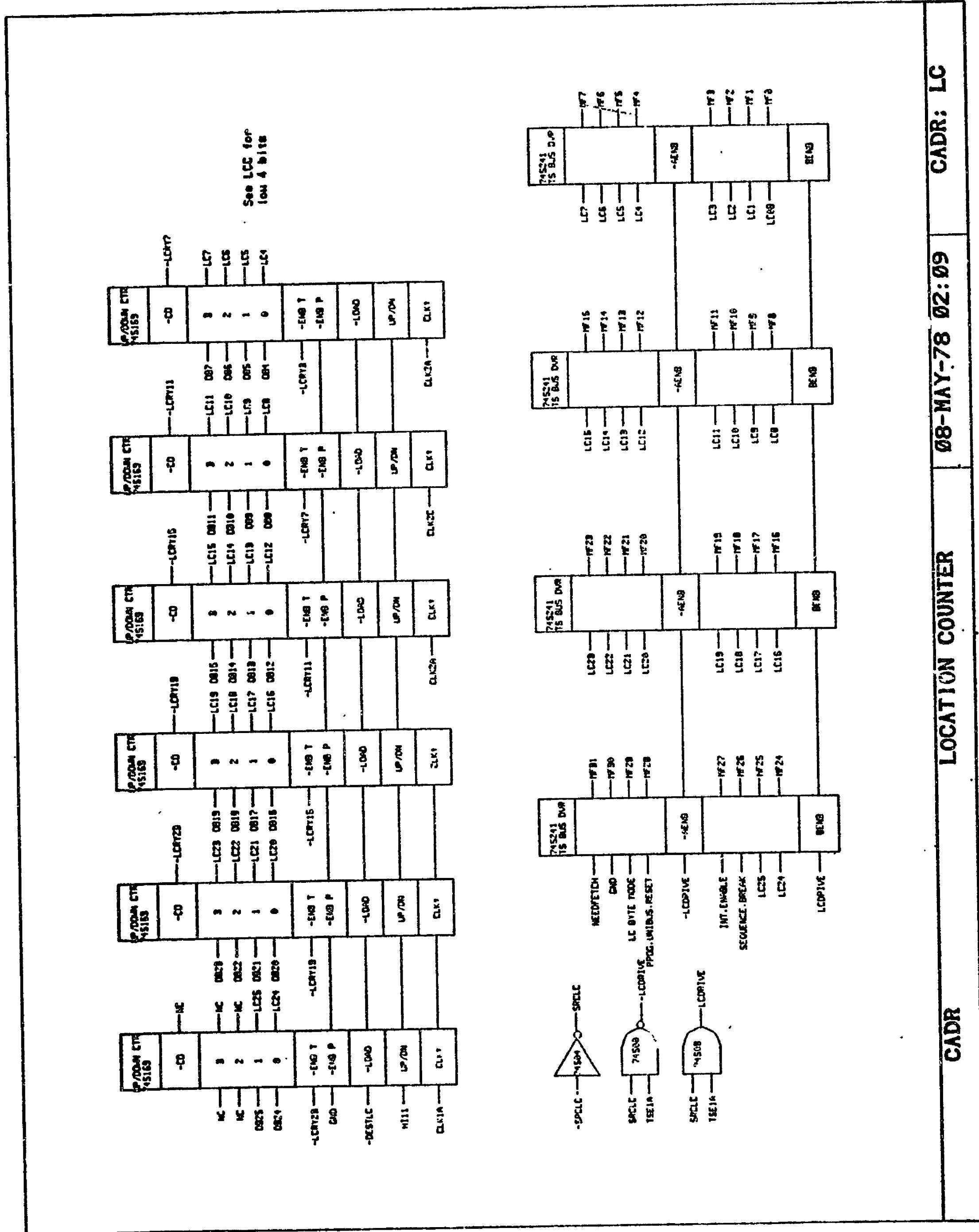
The NPC selector determines the location of the next microinstruction fetch. It loads one of four potential sources for the next program counter into the PC register, whose contents directly drive the microinstruction memory addresses. The four sources are the output of the SPC stack (SPC), the instruction register (IR), the dispatch memory outputs (DPC) and the incrmented PC (IPC). The PC may also be forced to a zero by the TRAP signal, used for initial bootstrap, and main memory parity errors. The output of the NPC selector is loaded on each clock into the PC register, and the incremented PC is developed with an adder chain from the PC register.


CADR: NPC
14-FEB-78 22:53
NPC.IPC.PC
CADR

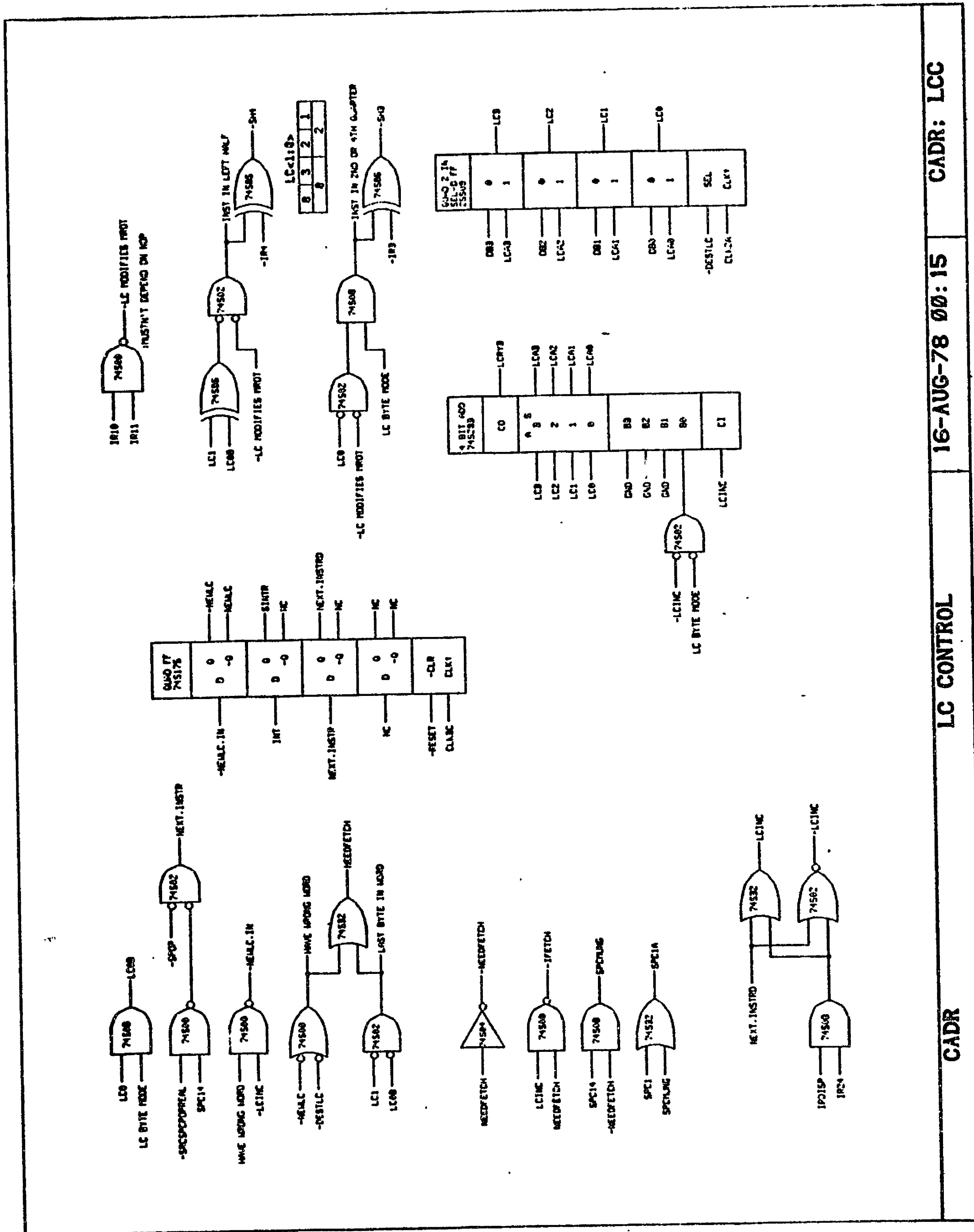
The LC register and Instruction Prefetch

The LC register provides an efficient mechanism for executing 16 bit and 8 bit macro instruction streams with the processor. LC contains the byte address of the next macroinstruction to be executed. When the SPC return memory flag specifying the end of a macroinstruction execution is set, the contents of LC are incremented by one or two (depending on the 8/16 bit instruction flag). Then, if the last byte of the macro instruction was just used, a main memory fetch is initiated, loading the memory address register from the LC register, shifted right by two.

During the course of macroinstruction execution, the LC register contents influence execution of all microinstructions with the miscellaneous field set to three. This specifies that the shift and mask select for the current operation is operating on a macro instruction word, and the value of the shift field is to be modified according to the low order bits of the LC, resulting in the selection of the proper half or quarter of the macro instruction word as data in the modified microinstruction.



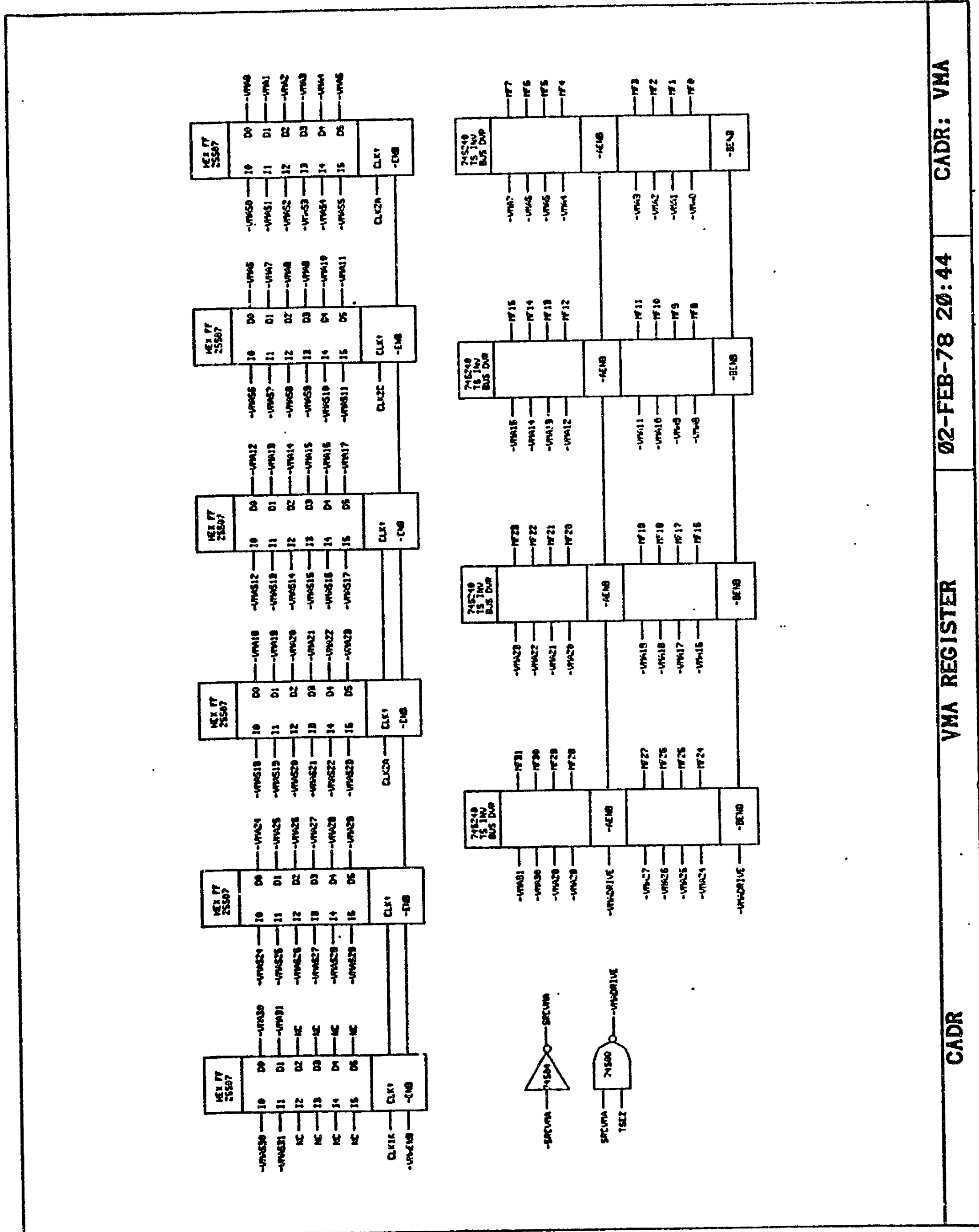
CADR
LOCATION COUNTER
Ø8-MAY-78 Ø2:Ø9
CADR: LC
See LCC for low 4 bits

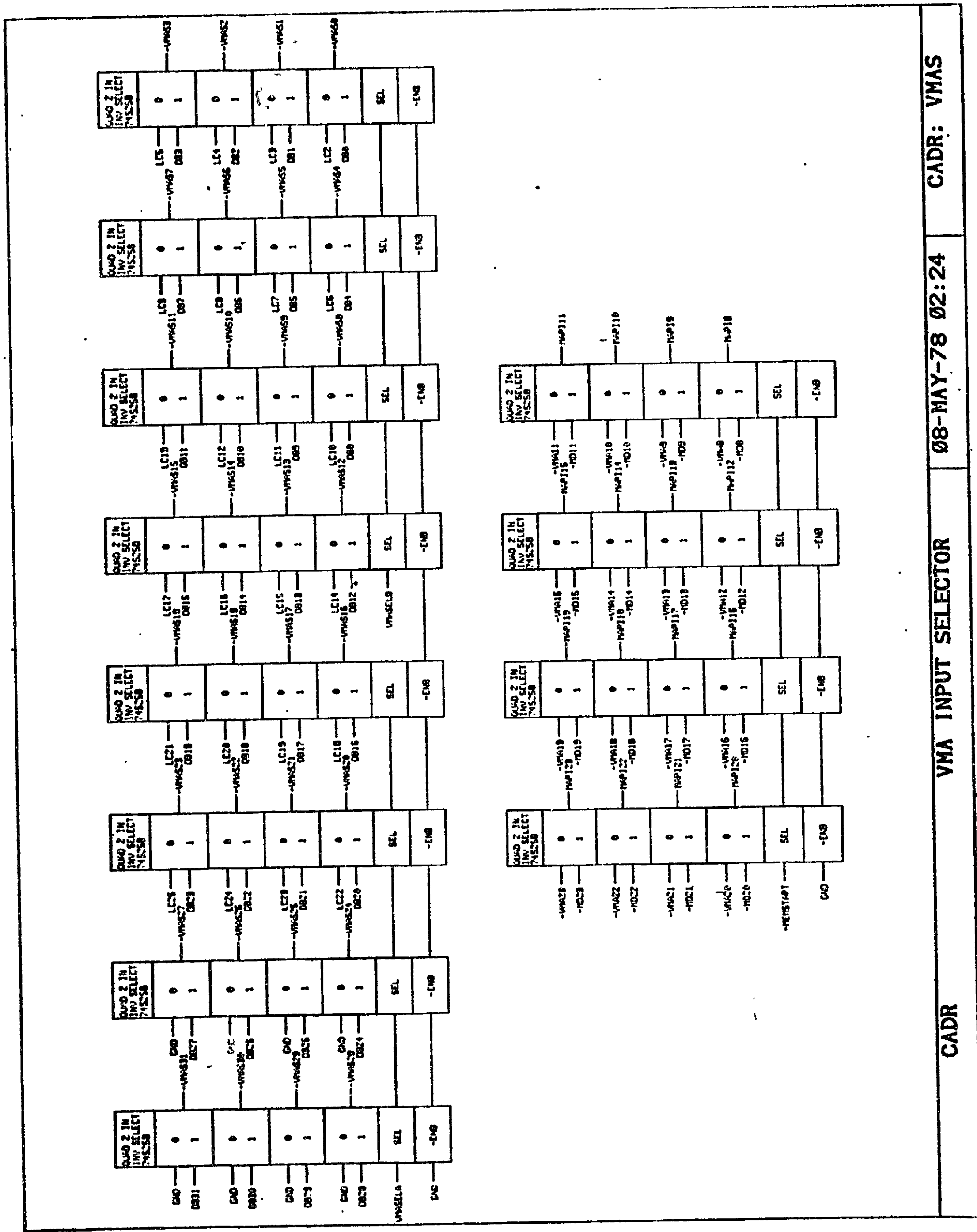

CADR: LCC
16-AUG-78 00:15
LC CONTROL
CADR

The VMA and VMA Selector

The VMA is a 32 bit register which holds the virtual memory address being fetched by the processor. It is normally loaded from the output bus, but may also be loaded during the macroinstruction prefetch sequence from the (shifted) LC register contents. The output of the VMA may drive the MF bus for access from the main data paths.

During one cycle prior to the initiation of a main memory cycle, the output of the VMA drives the virtual memory map inputs though the MAPI selector. This takes bits 23 though 8 (the virtual page number) and maps them into a physical page number. Bits 7 through 0 are passed directly to the memory system as an offset within the page.

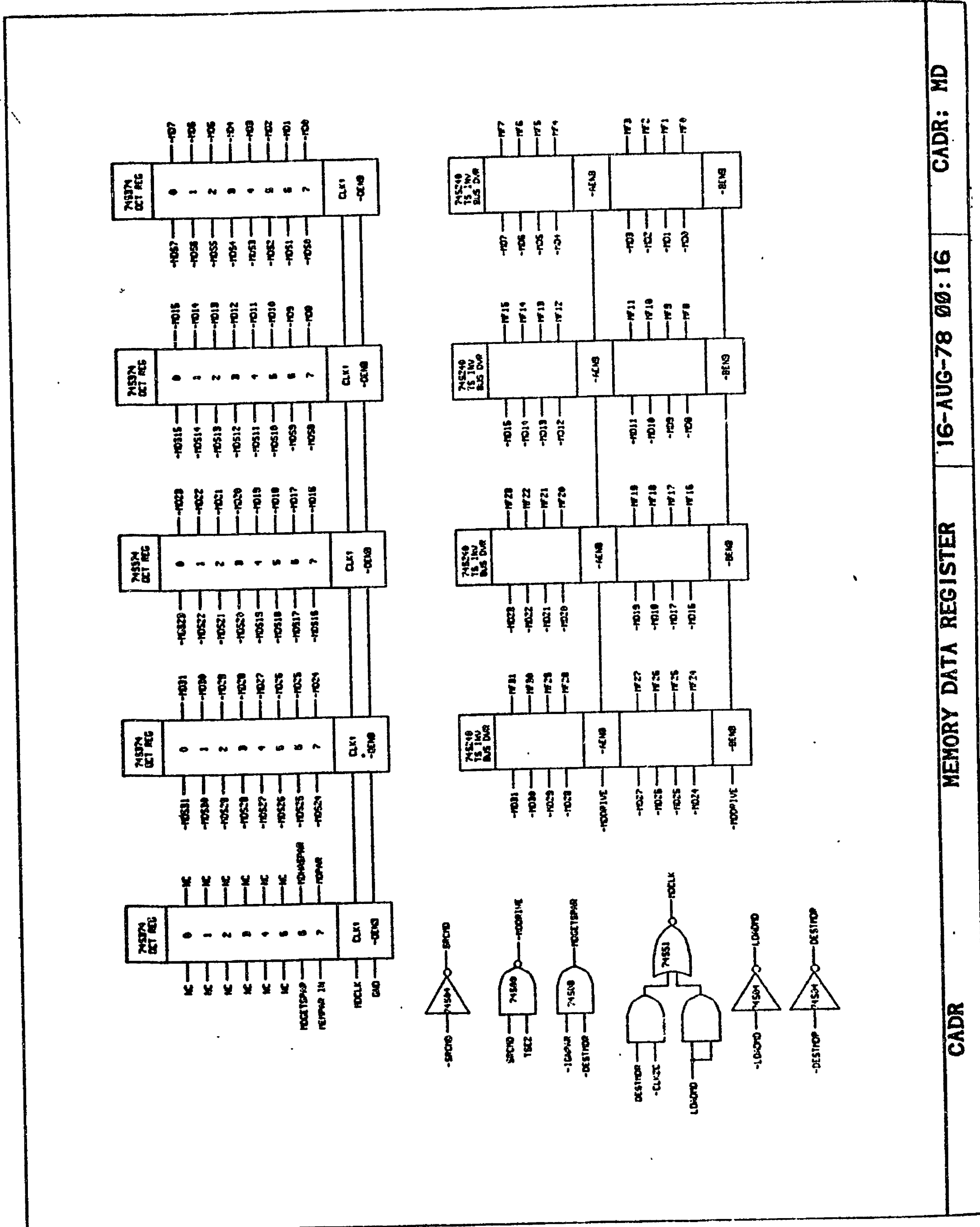

CADR
VMA REGISTER
Ø2-FEB-78 2Ø:44
CADR: VMA

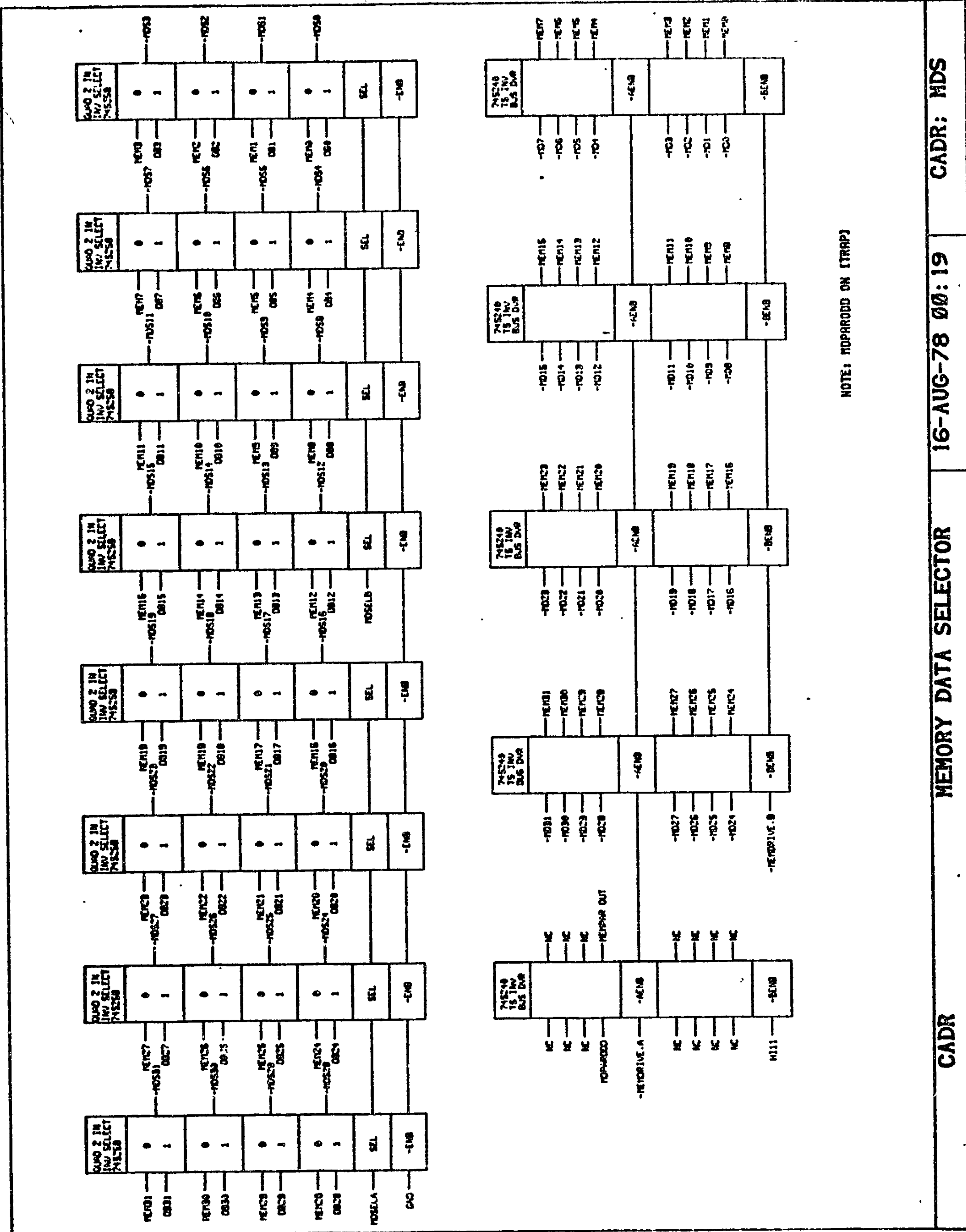

CADR: VMAS
Ø8-MAY-78 Ø2:24
VMA INPUT SELECTOR
CADR

The MD and the MD Selector

The MD is a 32 bit register holding the contents of a main memory word which is being read or written. Data is loaded into the MD from either the main memory system (through the MEM bus) or from the output bus. Output of the MD register drives the MEM bus (for memory writes), the MF bus for access from the main processor data paths, and the virtual page map input selector.

Access to the virtual memory map is provided so that the operation of checking the data type and newspace/oldspace pointer location can be effectively done simultaneously. Normally the map input selector is driven from the MD register, allowing newly fetched data coming into the MD to be looked up in the map and drive the main data paths for the dispatch instruction on the data type simultaneously.


CADR: MD
16-AUG-78 00:16
MEMORY DATA REGISTER
CADR


CADR
MEMORY DATA SELECTOR
16-AUG-78 00:19
CADR: MDS

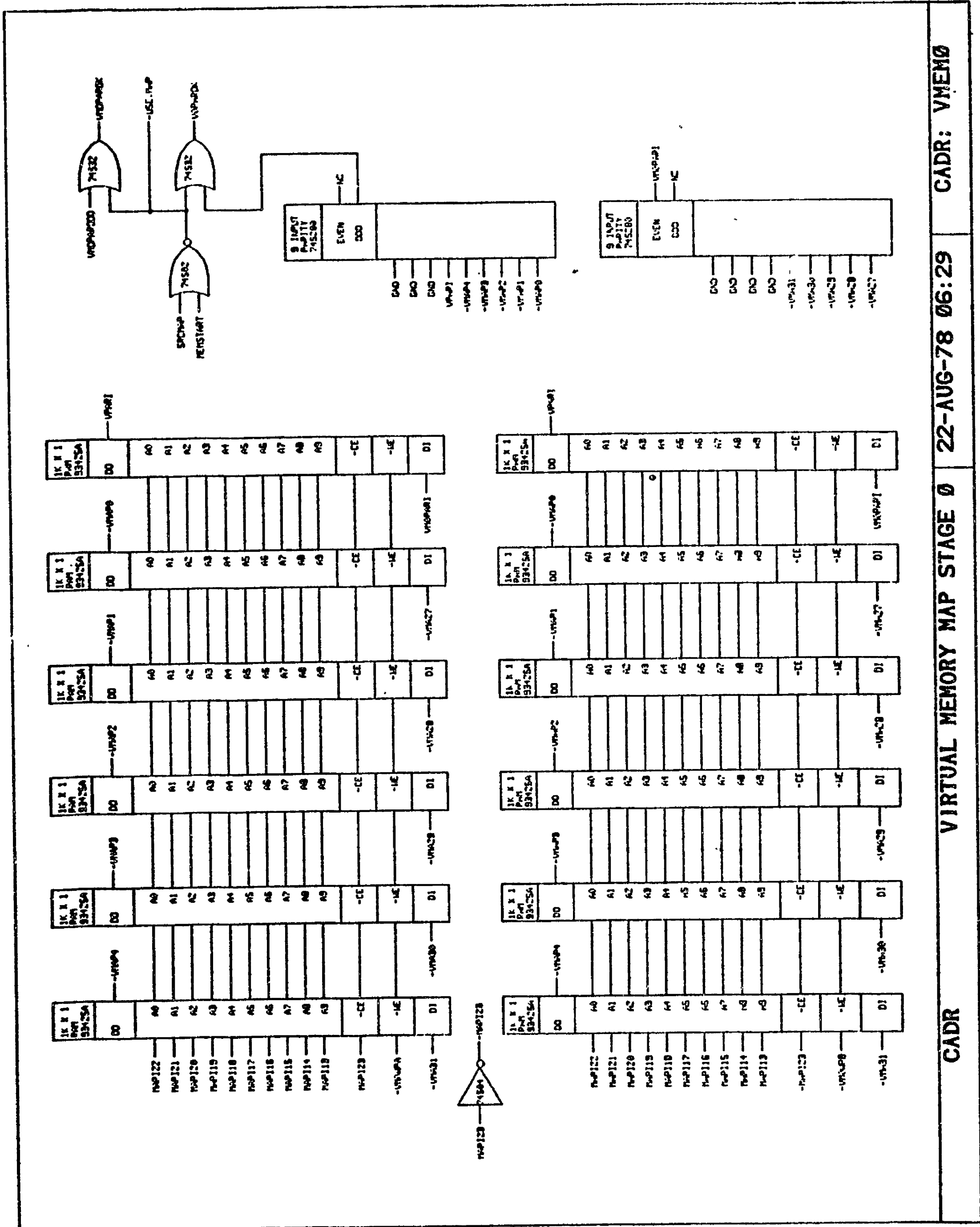
First and Second Level Maps

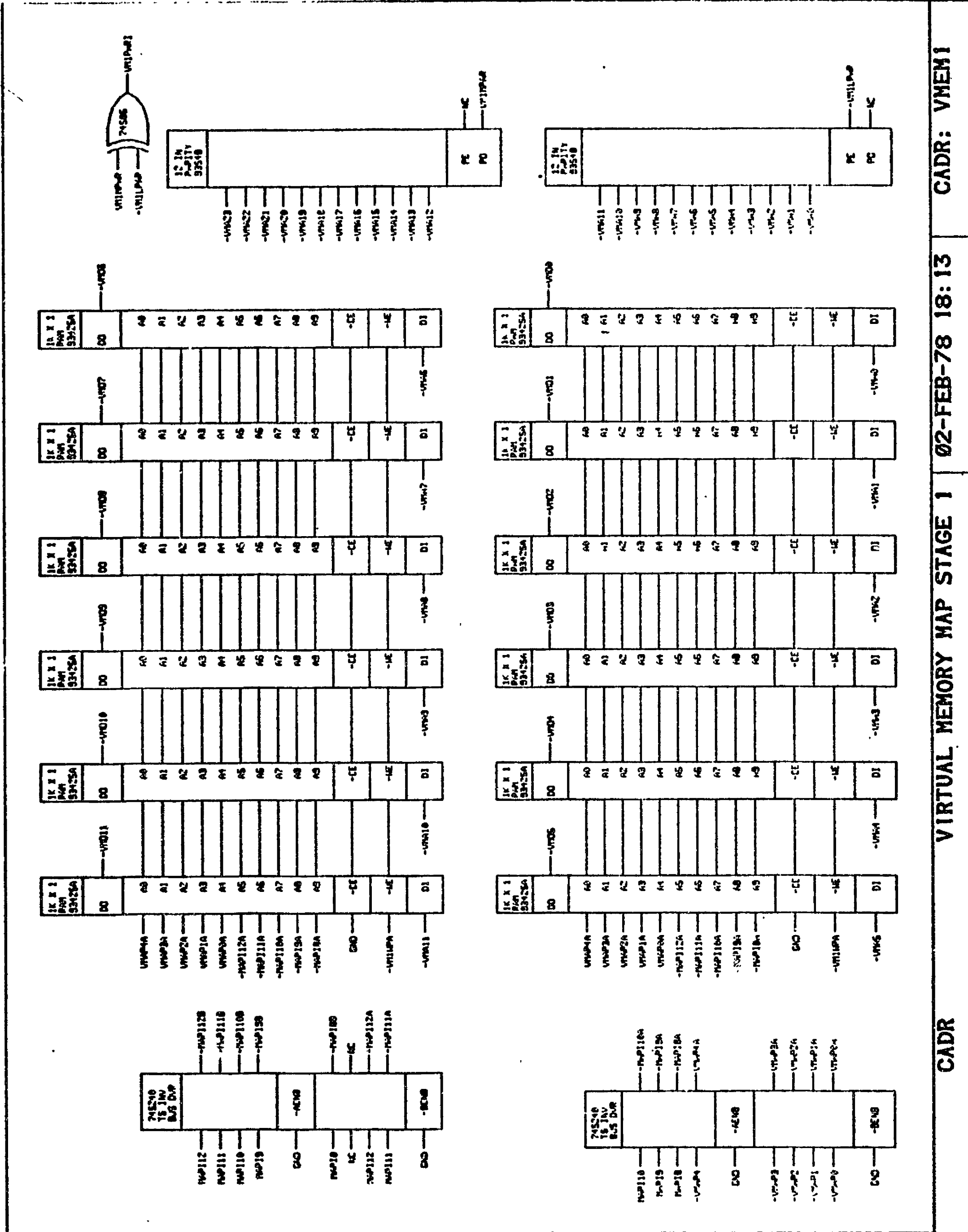
The virtual memory map is implemented as a two level table lookup map. The map input selector drives either the MD or VMA contents onto the MAPI lines. Bits 23 through 13 address a 2K by 5 memory, the first level map on the VMEM0 print. Of the possible 2048 entries in this table, a maximum of 31 are non-zero, producing non-zero outputs on the five VMAP lines. The five VMAP signals combine with the map input bits 12 to 8 to address the second level map, on prints VMEM1 and VMEM2. The second level map is a 1K by 24 bit memory, holding map entries.

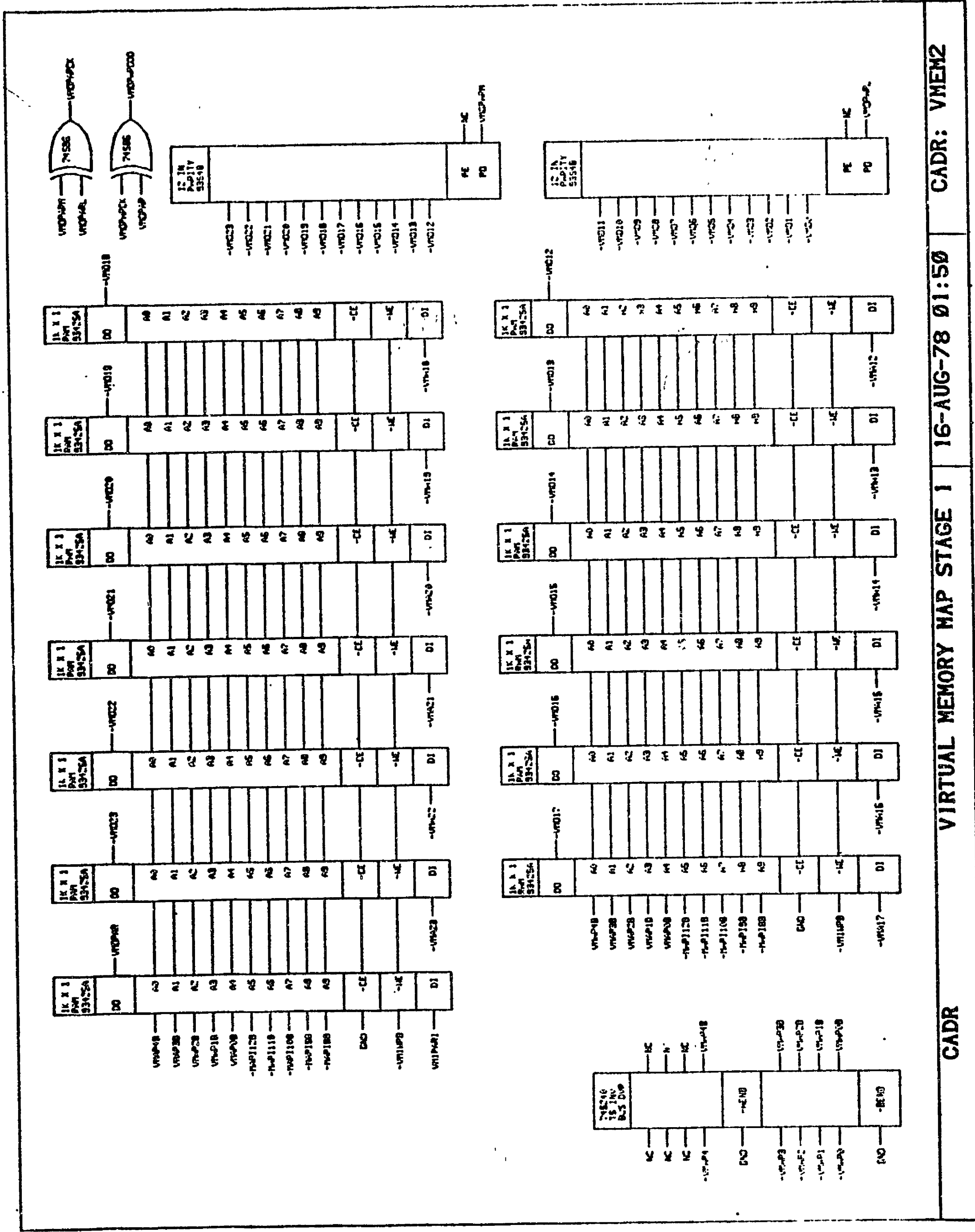
On a main memory read or write cycle, the map inputs are driven from the VMA, and the map outputs, after settling, are held in the latches shown on the VMEMDR print, such that the address sent to the memory system is stable for the duration of the memory cycle. The final memory address is formed from the low 8 bits of the VMA and the low 14 bits of the map outputs.

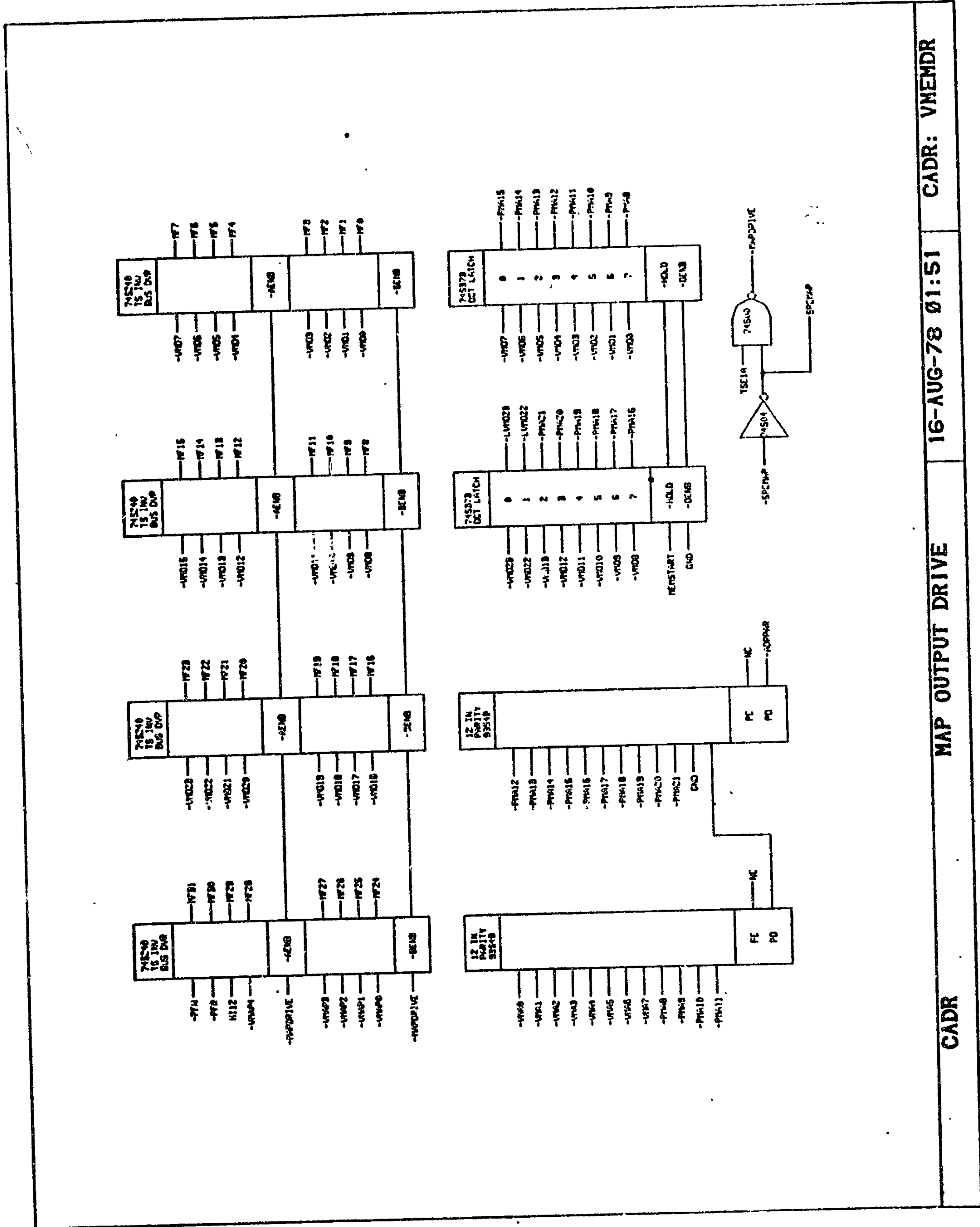
The VMEMDR print also contains drivers for the map outputs onto the MF bus, so that the main data paths can access the map data. The intermediate VMAP data is also driven onto the MF bus so that the processor can distinguish first from second level map misses in the paging microcode.

Writing of the map is done by addressing the map in the normal manner, by loading a particular page entry into the MD register, and then writing into the memories the data held in the VMA register.


CADR
VIRTUAL MEMORY MAP STAGE Ø
22-AUG-78 Ø6:29
CADR: VMEMØ


CADR
VIRTUAL MEMORY MAP STAGE 1
Ø2-FEB-78 18:13
CADR: VMEM1


CADR
VIRTUAL MEMORY MAP STAGE 1
16-AUG-78 01:50
CADR: VMEM2

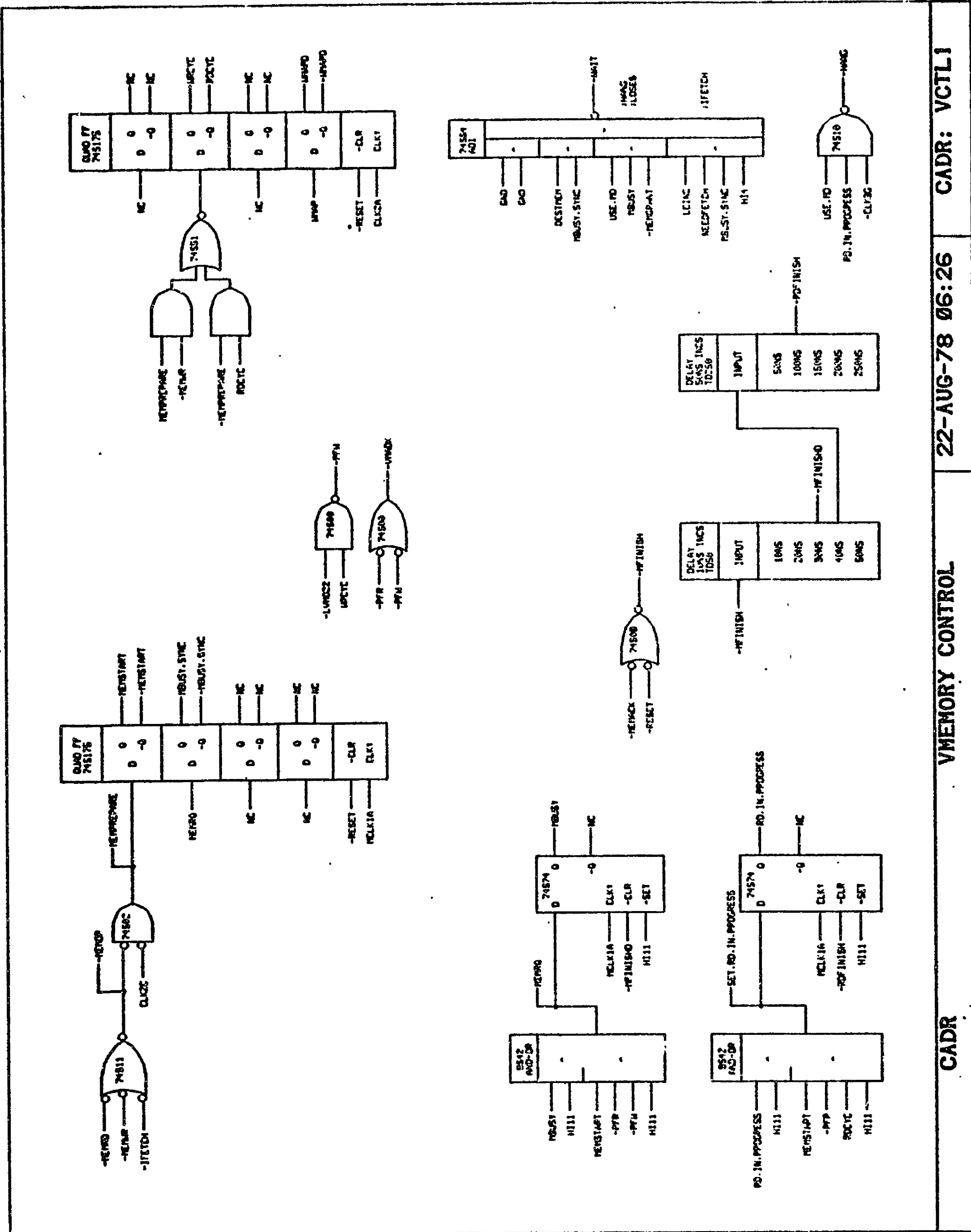

CADR
MAP OUTPUT DRIVE
16-AUG-78 01:51
CADR: VMEMDR

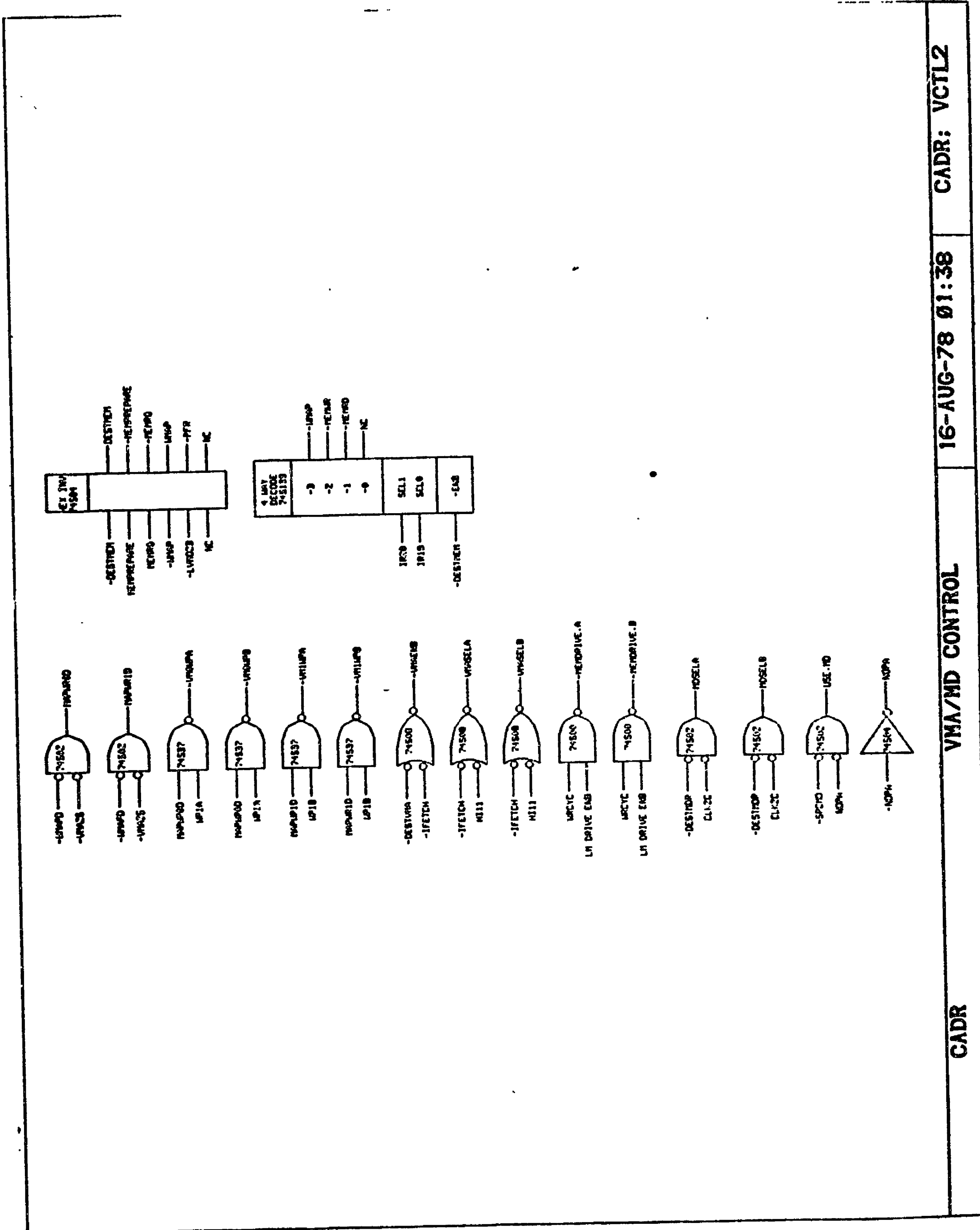
Memory Control Logic

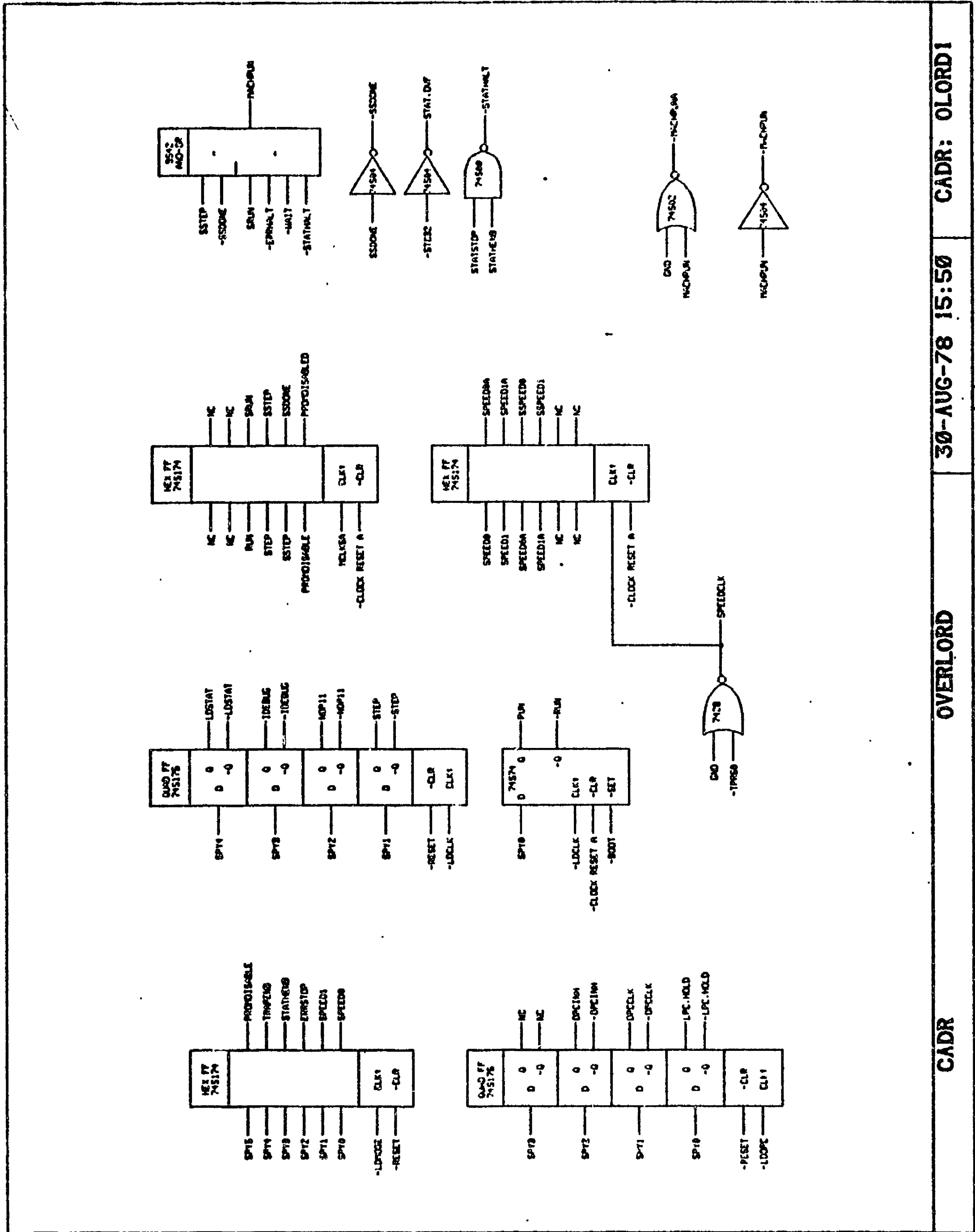
The timing interface between the processor and the memory system is handled on the VCTL1 and VCTL2 prints. These handle the timing of map references, syncronization between the asynchronous bus and the clocking of the processor, and detection of the cases when the processor clock must be delayed pending arrival of data from the main memory system.

Cycles are initiated with either a read, write, or instruction fetch (-MEMRD, -MEMWR, -IFETCH) is required, and results in setting of the MEMSTART flipflop. During the MEMSTART cycle, the map inputs are gated from the VMA.

Assuming there was no map miss or protect violation, the request is sent to the main memory system (MEMRQ) and memory busy is set (MBUSY). On read cycles, RD.IN.PROGRESS is set. Execution of microinstructions continues normally, unless one of the gates driving HANG or WAIT causes the processor clock generator to momentarily halt, after detecting the potential interference between the program executing and the state of the memory system.


CADR
VMEMORY CONTROL
22-AUG-78 06:26
CADR: VCTL1


CADR: VCTL2
16-AUG-78 01:38
VMA/MD CONTROL
CADR


CADR
OVERLORD
30-AUG-78 15:50
CADR: OLORD1

CADR	OVERLORD	31-AUG-78 08:31	CADR: OLORD2