22
11-18-76
to NTIS

UCID- 17299

Lawrence Livermore Laboratory

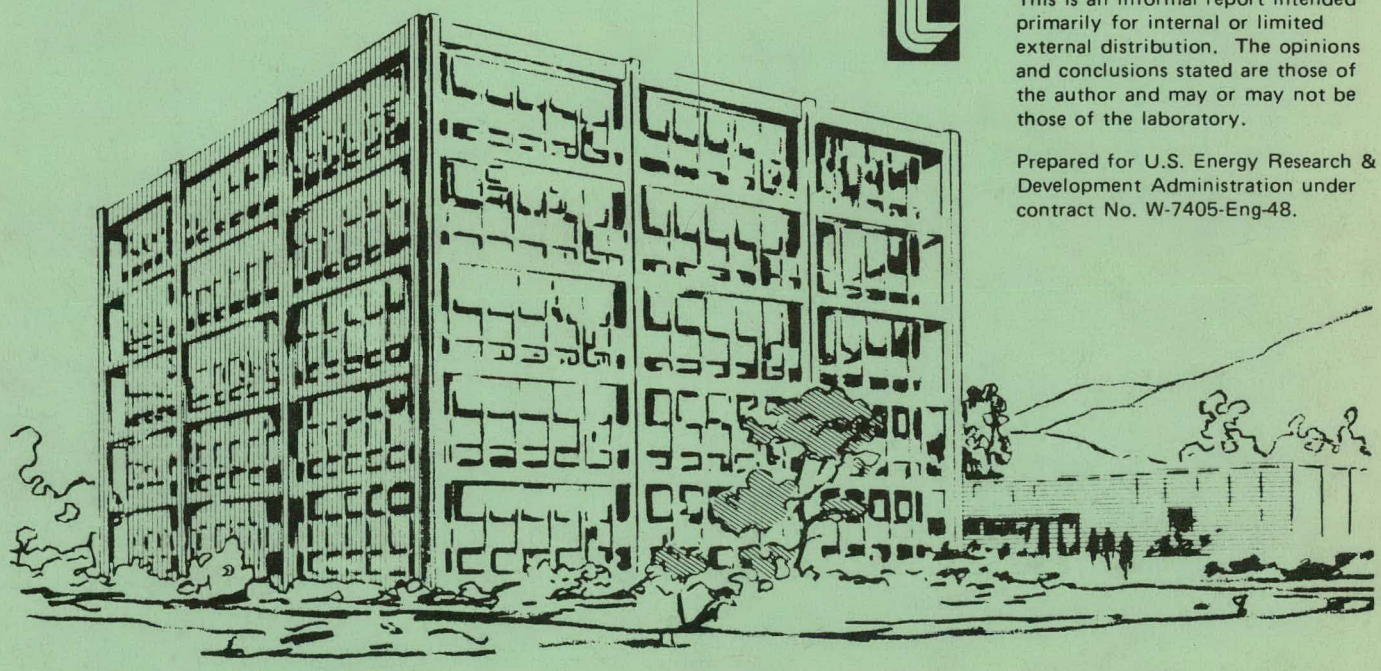
THE PRELIMINARY DESIGN OF AN ADVANCED PROGRAMMABLE DIGITAL FILTER NETWORK FOR LARGE PASSIVE ACOUSTIC ASW SYSTEMS

Thomas McWilliams, Lawrence C. Widdoes, Jr., Lowell Wood

30 September 1976

Prepared for The Naval Systems Division, Office of Naval Research
Arlington, Virginia
Under ONR Order No. N00014-76-F-0023

MASTER



Prepared for U.S. Energy Research & Development Administration under contract No. W-7405-Eng-48.

DISCLAIMER

LAWRENCE LIVERMORE LABORATORY

SS76-151

THE PRELIMINARY DESIGN OF AN ADVANCED PROGRAMMABLE DIGITAL FILTER NETWORK FOR LARGE PASSIVE ACOUSTIC ASW SYSTEMS

An Interim Report on Research Work in
Advanced Programmable Digital Filter Network Technology

Reported by: Thomas McWilliams
Lawrence C. Widdoes, Jr.
Lowell Wood

Special Studies Group
Physics Department

30 September 1976

Prepared for: The Naval Systems Division
Office of Naval Research
Arlington, Virginia

Under: ONR Order #N00014-76-F-0023

FOREWORD

This is an account of research work in advanced programmable digital filter network technology performed during the latter part of FY76 and FY76T by the Special Studies Group of the LLL Physics Department for the Office of Naval Research, under ONR Order #N00014-76-F-0023, along the lines specified in LLL Phys. Prop. 76-101, which was submitted to ONR in March 1976. This document reports satisfactory completion of all the items of this proposal's Work Statement, and the successful accomplishment of additional, related tasks which position this research project to maintain a very aggressive pace in FY77, given adequate funding.

The work reported herein was performed by Harlan Lau, Richard McWilliams, Thomas McWilliams, Joseph Simpson, Lawrence C. Widdoes, Jr., and Lowell Wood, of the Special Studies Group, with research sub-contact assistance from Paul Levine and Kottappuram Mohiuddin of Stanford University's Electrical Engineering and Computer Science Departments, supervised by Professor Forest Baskett.

Section Page

Section Page

Section Page

Section Page

1. Introduction

This report describes the design of an extremely high performance programmable digital filter of novel architecture, the LLL Programmable Digital Filter (LLL Filter).

Essentially all of the perceived Navy requirements for advanced digital processing systems may be effectively addressed with parallel processing systems, in which relatively independent processing units work in parallel on portions or sub-divisions of the entire problem, exchanging information with each other during the course of processing. This is the case whether one is concerned primarily with fleet defense (in which various processors might provide local control and monitoring of sensors or weapons systems while sharing information with each other on the time-varying aspects of attack and defense parameter spaces, both within single ships and between them), with SOSUS (in which each hydrophone array might have its own powerful processing unit exchanging filtered information with essentially identical units in all other stations monitoring a common region of the ocean, for coherent processing techniques such as aperture synthesis, or for accuracy enhancement or reliability purposes), or weather prediction (in which each processor might handle meteorological data acquisition and time-advanced extrapolation for its own, relatively small section of the simulated air-ocean envelope, exchanging interface condition information with those of its fellow processors responsible for adjacent sections).

Moreover, the enormous demands on digital processing power which Navy requirements, of which the foregoing are only examples, impose on modern digital processing technology appear to be most fully satisfied in the foreseeable future *only* by extensive use of parallel processing techniques and hardware. The doubling time for raw processing power from single processing unit superprocessors (for example, the CDC 6600/7600 series) has been increasing steadily over the last decade, and presently appears to be more than 4 years, a sharp contrast to the 1.5 year figure characteristic of the late 50s and early 60s. Parallel processing systems, on the other hand, are capable of indefinitely great extension in raw processing power with essentially zero technological risk and time lag, and moreover, with advance knowledge of system performance and thus cost-effectiveness

We have therefore undertaken to determine the optimal structure of a parallel processing system for addressing the specific Navy application centering on the advanced digital filtering of passive acoustic ASW data of the type obtained from the SOSUS net.

1.1 Advantages of Parallel Processors

For problems which involve algorithms amenable to parallel processing ([Amdahl 1967], [Ball 1962], [Carroll 1967], [Flynn 1966], [Katz 1970]), parallel architectures can offer certain major advantages over sequential architectures. The advantages result from the modularity inherent in parallel architectures. These advantages can be categorized as advantages of *reliability*, *economy*, and *size*.

The advantage of reliability has been discussed extensively (for example, see [Barker 1975] or [Hamer-Hodges 1973]); failure of a single module may not entail failure of the entire system if the module failure can be detected and the module replaced by a duplicate under program control.

Of primary importance among the advantages of economy are the economies of scale in the construction phase; by repeating the construction of a single processing element many times, the total cost per processing element may be greatly reduced.

A second economy of scale comes in the design phase. Theoretically, the design cost per processing element is reduced asymptotically to zero as the processing element is replicated. Actually, any real parallel processor must include some design costs per processing element which grow as the number of processing elements is increased, but these costs may be negligible.

A third important economy has been overlooked in previous parallel processor design efforts; it is the potentially reduced time lag between the freezing of the system design and the delivery of the first operational system. Although this time lag may include both hardware and software contributions, the software contribution will be neglected in this analysis. Essentially, by replicating a relatively simple processing element many times and using a regular interconnection network, the lag time mentioned can be made very small; it is virtually independent of the processing power of the total system. As a result, the semiconductor technology used in a properly designed parallel processor can be nearly state-of-the-art, whereas the technology used in a more complex processing structure must be considerably more out of date. This time-lag phenomenon will continue to seriously degrade the cost-effectiveness of delivered complex systems as long as advancing semiconductor technology continues to provide exponentially more cost-effective components, but may be essentially eliminated in advanced parallel processing systems.

One additional economy has also been overlooked in the past; this economy results from the freedom of the parallel processor designer to choose the most cost-effective processing element structure independent of the processing power of the element. Cost-effectiveness of sequential processor structures is not constant over all levels of processing power. Although the specific shape of the cost-effectiveness curve depends upon the technology available and upon the characteristics of the target problem domain, for any specific technology and problem domain the cost-effectiveness curve has a finite number of broad maxima. Because the design of a digital processing system must be aimed not only toward maximum cost-effectiveness, but toward some minimum processing power, designers of single processor sequential systems have not been able to utilize structures with possibly higher cost effectiveness but lower processing power. On the other hand, the designer of a parallel processor may be able to achieve a total cost-effectiveness which is nearly the same as the cost-effectiveness of the processing element, and since the processing element may not be constrained to have a large minimum processing power, to achieve higher total cost-effectiveness.

Independent of these economic advantages is the advantage of size; regardless of whether it is economically feasible to build increasingly powerful sequential systems, at some point it becomes physically impossible (with state-of-the-art technology) to build these machines. It can be argued

that sequential systems of almost arbitrary speed can be built given enough resources, and so the advantage of size reduces to the advantage of economy. However, from a practical viewpoint, at some point the cost of a sequential system increases so rapidly with speed that this argument is moot, and in addition, there are theoretical limits both in physics and mathematics to the speed of sequential machines, and these limits do not apply to parallel processors working in appropriate problem domains. This advantage of parallel processor structures is important because for the forseeable future it will be desirable to build systems with more total processing power; numerical weather prediction with its real-time constraints is an obvious example.

These arguments about the advantages of parallel processors are applicable without modification only if the target problem domain can utilize with high efficiency each processor in a parallel processor system of arbitrary size. The suitability of various problems for parallel processing has been the subject of much academic contention ([Amdahl 1967], [Flynn 1966], [Minsky 1971]). Unfortunately, only a few parallel architectures have proven economically viable, so there has been little impetus to develop new algorithms for exotic parallel machine architectures. We believe that the computational simulations of many large physical problems, for example, the optimal SOSUS digital filtering problem, are so well-suited for parallel processor architectures and so important, that any one such simulation alone is sufficient justification for the intensive development of such digital processing technology.

2. System Overview

The LLL Programmable Digital Filter consists of high-performance processors that execute independent instruction streams and access a common main memory via a crossbar interconnection network (crossbar). All of main memory is uniformly accessible by every processor.

The crossbar arbitrates access by all processors to 16 *block storage modules* (BSMs) which are interleaved on either the most significant or the least significant address bits (manually selectable). Ignoring conflicts, approximately 1 micro-second is required to accomplish a memory read of four 36-bit words.

The crossbar contains facilities for logically disconnecting (amputating) any processor. Amputation of processor P_i can be invoked by any other processor P_j. In order to prevent processors, errant due to either hardware or software reasons, from performing spurious amputations, an amputator must, by convention, pass elaborate software correctness tests (which will involve confirmation by other processors).

The programmable digital filter has been optimized to include 16 processors. Each processor contains a novel dual cache, which buffers the interconnection network against processor accesses to instructions and local variables. Processors do not have local memory. No connections exist between processors except through the crossbar.

Interprocessor communication takes place in main memory; memory management hardware allows protection of interprocessor communication. Interprocessor synchronization is accomplished by a combination of primitive mechanisms including interrupts, which can be sent from any processor to any one other processor over the crossbar, special mutual exclusion hardware, which is addressed as memory, read-modify-write capability in the crossbar, and special memory access modes (specified in the virtual-to-real map) which force some memory accesses to bypass the caches.

An extremely high-level instruction set improves the individual processor performance by reducing the number of instructions which need to be executed. Furthermore, natural addressing modes are complex, and therefore the processor implementation separates addressing and execution into three parallel micro-processors. The instruction set is horizontally micro-programmed in writeable control store, and can therefore be extensively modified to reduce execution time and code size for specific applications.

A large virtual memory space is provided in order to allow the architecture and software to remain fixed while memory costs decline and real memory size increases.

2.1 System Configuration

Figure 2.1-1 shows an overview of the LLL Programmable Digital Filter.

Main memory is divided into a number of *block storage modules* (BSMs) that can be simultaneously and independently accessed by any of the processors. When two or more processors demand access to any one BSM, memory contention logic establishes a queue. The queueing discipline is such that no processor can access a given memory BSM twice before a processor desiring to access that BSM is allowed to access it once.

Each processor communicates with the crossbar over two unidirectional 25-bit cables. The

crossbar communicates with memory over two unidirectional 50-bit cables. Internally, the crossbar switch is 25 bits serial in each direction.

Main memory provides a path for interprocessor communication. Interprocessor synchronization is accomplished by means of munch registers, which appear as memory locations, hardware queues, which are accessed as memory locations, read-modify-write capability in the crossbar, and inter-processor interrupts. Interrupt requests are sent through the crossbar and are handled by the *interrupt controllers*. Whenever a processor is interrupted by its associated interrupt controller, it performs memory accesses to determine the nature of the interrupt.

Input/output is accomplished in two ways. For low speed I/O devices such as terminals, data is transferred by the writing and reading of the I/O control words, which are addressed as memory, and are located in the various memory controllers. Each low-speed I/O device is attached to some specific interrupt controller, and thus can interrupt one processor. The interrupted processor may then forward the interrupt. High-speed I/O devices (for example, disks) are handled by a direct memory access (DMA) port, which communicates with main memory in the same way as all the processors do.

We summarize the the major characteristics of the system architecture as follows:

- Multiple (16) identical processors execute independent instruction streams.
- Every processing element can uniformly address all system memory through a (25-bit serial) crossbar switch.
- Each processing element has dual private caches to reduce contention for main memory, to reduce average memory access time, and to insure that system performance does not seriously degrade as more processing elements (and therefore a bigger and slower interconnection network) are added.
- Each processing element can direct an interrupt to any other processing element.
- Munch registers, hardware queues, and read-modify-write memory capability are available for synchronization.
- The virtual-to-real memory maps include access mode bits which allow efficient sharing of data and instructions.

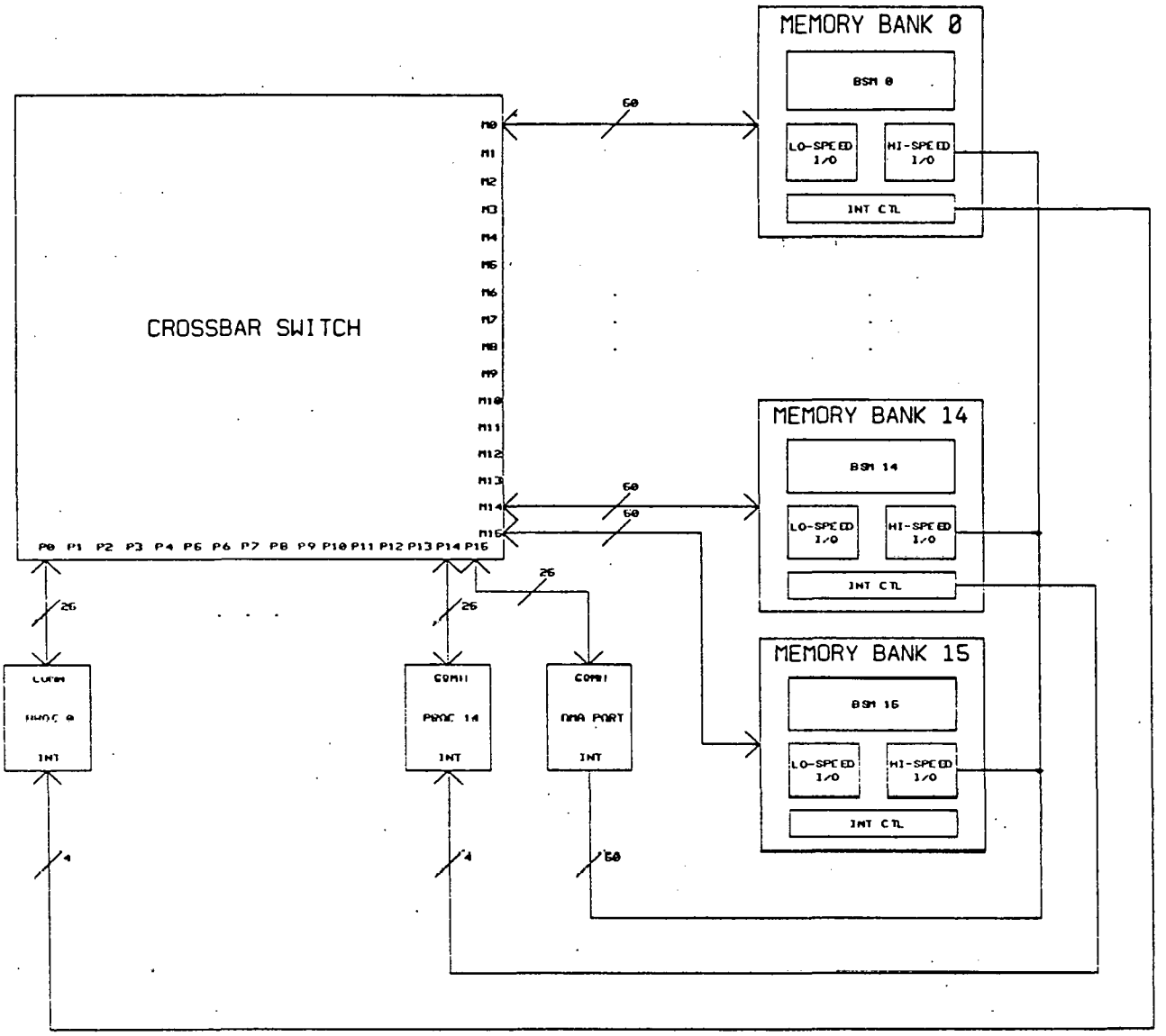


Figure 2.1-1
System Configuration

2.2 Processor Organization

The processors shown in Figure 2.1-1 are complete high-performance computing elements which could be used in either a uniprocessor or multiprocessor configuration; they are extremely cost effective in either environment.

The processor architecture and design are described in Section 3 and Section 4. The basic processor organization is shown in Figure 2.2-1.

Each processor has dual high-speed caches; one contains only instructions, and the other contains data. Writes *ordinarily* do not update main memory, but affect only the caches (see Section 3.1 and Section 3.3 for full detail).

A virtual-to-real address map in each processor translates addresses generated by instructions into addresses used by the hardware, and also defines access modes for memory pages. A page can be tagged as not cacheable, in which case it is never placed in the cache, and all writes to the page then write through to main memory.

The Instruction Box (IBOX) contains a general-purpose micro-programmed sequencer, which executes out of writeable control store. The IBOX performs all operations required to decode instructions and fetch operands. In particular, the IBOX performs the virtual to physical address translation, implements the various memory access modes, handles communication with the crossbar, and fields interprocessor interrupts.

The IBOX also controls the Execution Box (EBOX). The EBOX performs all arithmetic and logical operations except those involved in addressing. The organization of the EBOX is similar to that of the IBOX; it contains a micro-programmed controller and internal registers. The EBOX is designed for high-speed floating point arithmetic; its floating point algorithms allow three rounding modes; true stable rounding, ceiling rounding, and floor rounding.

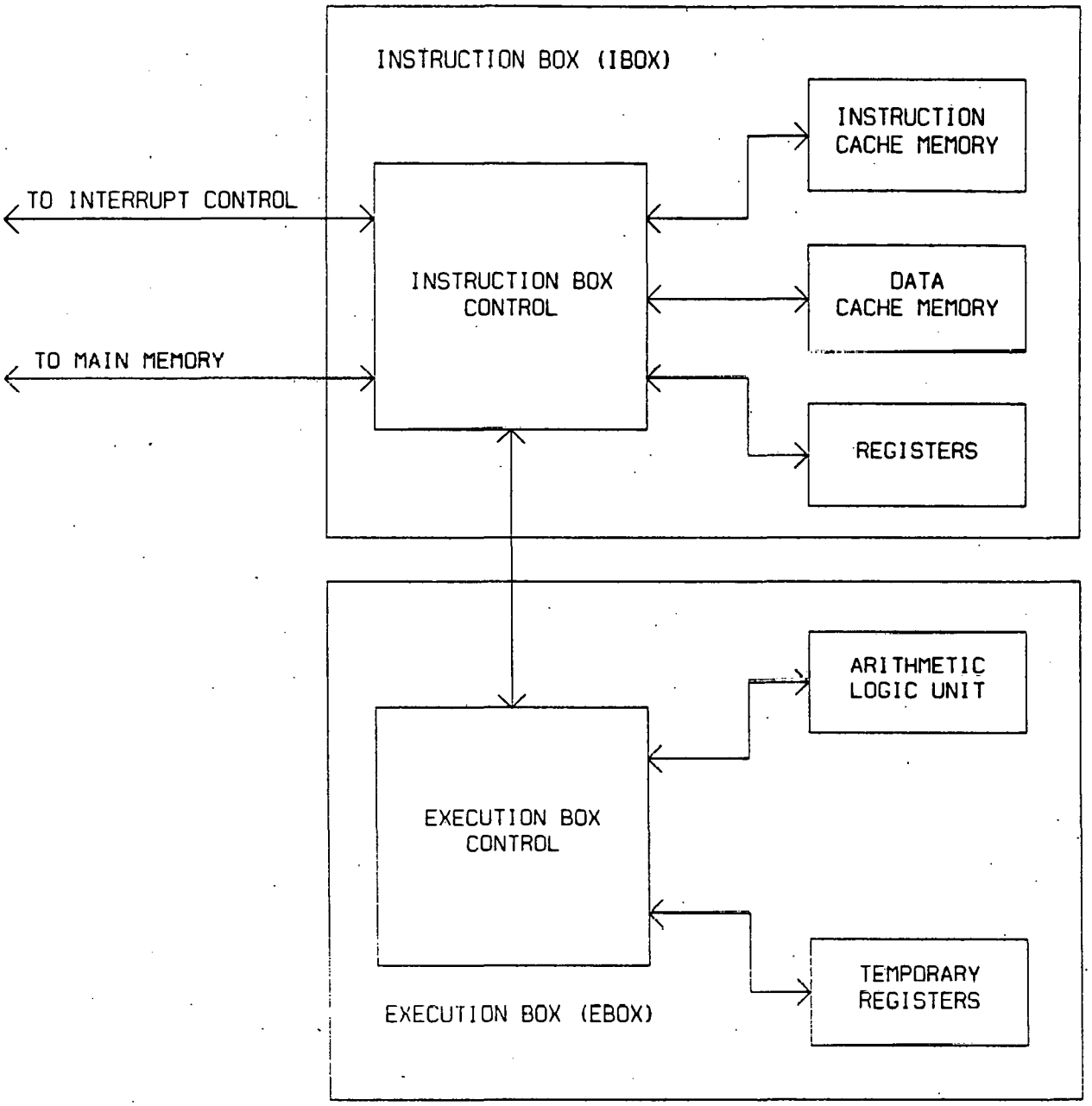


Figure 2.2-1
Processor Organization

3. Processor Architecture

We summarize the processor's major architectural features as follows:

- A very large (2^{28} word) virtual address space to allow each processor to uniformly address any system memory of feasible size in the forseeable future.
- Efficient mechanisms for allowing the executive to communicate with user processes.
- A high-level instruction set ideally suited for compilers.
- An instruction set specifically tailored to reduce the frequency of pipeline interlocks in a high-performance implementation.
- The capability to perform three-operand instructions through the use of a unique "T-field" descriptor.
- Comprehensive floating-point capability, including three rounding modes and the option to trap on excess pre- or post-normalization.
- The capability to directly perform operations on operands of 4 precisions: quarter-word, half-word, single-word, and double-word.
- Special instructions for dealing with the multiprocessor environment.

Certain processor implementation details are included in this section for clarity; processor implementation is fully described in Section 4.

3.1 Caches

Each processor has a private cache; this cache reduces memory contention and reduces access time for areas of locality, thereby lowering the performance requirements for the switching network and main memory.

The cache is implemented in two parts; the *instruction cache*, and the *data cache*. Both caches can be read simultaneously, allowing instructions representable in one word, requiring only one execution cycle, and having at most one memory operand to be executed continuously at a rate of one instruction per cache cycle (approximately 100 nano-seconds); the instruction set has been optimized so that instructions of this type predominate dynamically. Each cache is set-associative, with a set size of 4 and a capacity of 4K words (1K lines of 4 words each).

The instruction cache retains only locations accessed as instructions, and the data cache retains locations accessed as operands of an instruction. (Note that instruction words may be accessed as data.) The hardware insures that no memory word is contained in both caches as follows: Instructions are always fetched from the instruction cache. If a necessary instruction is not resident in the instruction cache, then a 4-word line is fetched from the data cache or memory, in that priority, and is evicted from the data cache. If the line was marked as having been altered in the data cache, then it is written out to memory. The instruction cache contains no mark bit; writes and data reads always access the data cache. If a necessary data line is not resident in the data cache, then it is fetched from the instruction cache or memory, in that

priority, and is evicted from the instruction cache. This discipline insures that no memory word is contained in both caches simultaneously, with the disadvantage that it forces slow transitions between writing and executing or executing and writing any block of instructions.

The cache uses *physical addresses* to tag entries, allowing the software to switch virtual address spaces without sweeping the cache, and eliminating the problem of clogging the cache with multiple copies of shared read-only data.

For communication or synchronization reasons, it will be necessary at times to insure that certain variables are not present in the cache of a specific processor. Access modes may serve this purpose, as described in Section 3.3, but in addition two special instructions are provided: The instruction "KILL DATA V,L" sweeps the data cache, writing to memory (if marked) and invalidating every entry which has a virtual address U such that $V \le U \le V+L-1$ (L is assumed to be a count of quarter-words). The instruction "KILL INSTR V,L" performs an identical function for the instruction cache (in which no entry is ever marked). The instruction "KILL DATA INSTR" performs both sweeps.

For reasons of efficiency, it may be convenient to avoid invalidating the cache residents swept by the KILL instructions. A special instruction is provided for this purpose: The instruction "UPDATE DATA V,L" sweeps the data cache, writing to memory (if marked) every entry which has a virtual address U such that $V \le U \le V+L-1$ (L is assumed to be a count of quarter-words). No analogous instruction is provided for the instruction cache, since instruction cache entries cannot be marked.

Depending upon the magnitude of L in these KILL and UPDATE instructions, the hardware may sweep the entire cache instead of individually sweeping each location in the specified range.

No instructions are provided which, when executed on processor P_i, cause the cache of processor P_j $(i \neq j)$ to be swept. This necessary function will be accomplished by directing a special interrupt from P_i to P_j which causes P_j to sweep its own cache.

3.2 Virtual Memory

The LLL Filter uses paging to map 30-bit virtual addresses to 30-bit real addresses (although the particular implementation of the LLL Filter described in Section 4 uses only 28-bit real addresses).

The virtual-to-real address map is shown in Figure 3.2-1. A virtual address space is uniquely identified by the contents of the *segment base register*, which is the main memory address of the *segment pointer table* for the address space, or is a pointer to the disk address of same. The segment pointer table is a contiguous list of *segment table pointers*. Each segment table pointer is either the main memory address of a *segment table*, or the disk address of same, or is null, indicating that the segment table does not exist. Each segment table is a contiguous list of *page table pointers*. Each page table pointer is either the main memory address of a *page table*, or the disk address of same, or is null, indicating that the page table does not exist. Each page table contains a list of *page table entries*. Each page table entry contains either the main memory address of a *page*, or the disk address of same, or is null, indicating that the page does not exist.

An address translation in general involves three memory references, one to the segment pointer table, one to the segment table and one to a page table; the segment base register is a hardware register inside the processor. A *page map* in each processor contains (for the most recently used pages) the complete translation from virtual page address to real page address.

The processor contains two hardware page maps; one translates addresses of locations accessed as instructions, and one translates addresses of locations accessed as data. Each page map is implemented as a set-associative memory with a set size of four and a capacity of 64 entries, therefore 128 address translations can be stored simultaneously in the processor. An entry may be stored in both page maps.

The processor hardware actually contains *two* segment base registers, EXEC_SEG_BASE_REG, and USER_SEG_BASE_REG; an instruction may conveniently specify that either be used in mapping each memory operand of an instruction (see the discussion of the M bit in Section 3.7.3.1.2). Each page map entry contains a bit called the *base bit*, which identifies which of the two segment base registers the entry is associated with. The address space specified by EXEC_SEG_BASE_REG will be called the *executive address space*, and the address space specified by USER_SEG_BASE_REG will be called the *user address space*.

Whenever a segment base register is altered, all page map entries associated with that segment base register must be invalidated: The instruction "WRITE EXEC JUMP X,J" loads EXEC_SEG_BASE_REG with X, invalidates all page map entries associated with EXEC_SEG_BASE_REG, and jumps to location J. The instruction "WRITE USER JUMP X,J" loads USER_SEG_BASE_REG with X, invalidates all page map entries associated with USER_SEG_BASE_REG, and jumps to location J.

In user mode, any reference to the executive address space causes a trap to the executive trap vector at address REF_EXEC. The executive may refer to the user address space without trapping.

Whenever a necessary translation is not resident in a page map, the necessary entry is fetched from memory and placed in the page map. A page map resident may be evicted in this process, but page map residents need not be written to memory when evicted. Whenever an entry is fetched from memory, the *reference bit* is set in the page table entry in memory; this reference bit is used by the operating system in the page replacement algorithm.

The data cache page map contains a *mark bit* for each entry. When a write occurs, if the page written is unmarked in the data cache page map, then the mark bit is set in the appropriate page table entry in memory and in the data cache page map. If the page written is marked in the data cache page map, then the page table entry in memory is not modified. Mark bits are not necessary in the instruction cache page map since all writes are done to the data cache.

Whenever the executive needs to modify page table entries to reflect the changing configuration of real memory, a protocol must be invoked which removes invalidated page table entries from the two page maps of each processor. The hardware refills the page maps directly from main memory, bypassing the caches, therefore invalidated page table entries need not be removed from the caches. Special instructions are provided for removing entries from both page maps simultaneously: For example, the instruction "KILL USER MAP V" will remove any entry in the instruction cache page map or the data cache page map which maps virtual address V in the user address space to any real address. The protocol mentioned above then requires that the processor P_i, executing the operating system, interrupt each processor P_j which may have in its page maps the entry to be modified, and cause each such P_j to execute a KILL USER MAP instruction.

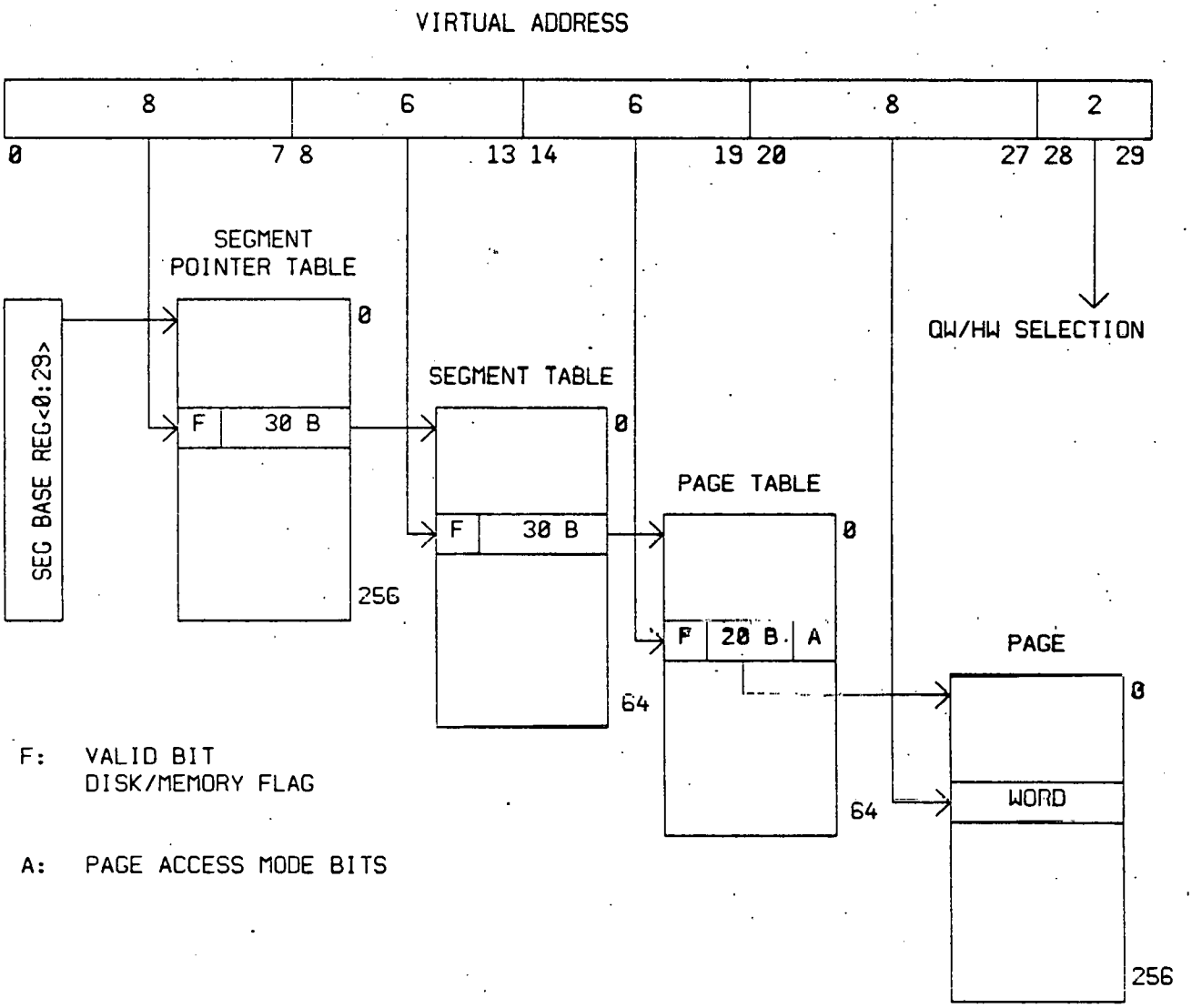


Figure 3.2-1
Virtual to Real Address Translation

3.3 Memory Access Modes

Each page table entry includes bits which specify the access modes of the page. The names and meanings of these bits are as follows:

Instructions. If this bit is false, then a hard trap to the executive at trap vector NOT_INSTRUCTION will occur when a location from this page is accessed as an instruction.

Data. If this bit is false then a hard trap to the executive at trap vector NOT_DATA will occur when a location from this page is accessed as an operand of an instruction.

Read-through. If this bit is true, then any read of a location on this page will cause a memory access to occur; the resulting data will be placed in the cache if and only if the location is already a cache resident.

Write-only. If this bit is true, any read from a location on this page will cause a hard trap to the executive at trap vector WRITE_ONLY.

Write-allocate. If this bit is true, then any write miss will allocate a cache entry and the data will be written into the allocated entry. Write hits will simply update the cache entry. If this bit is false, then a write miss will not allocate a cache entry.

Write-through. If this bit is true, then any write will update memory. If the write is a write hit, then cache will be updated as well. If the write is a write miss, then if and only if the write-allocate bit is true, a cache entry will be allocated and written.

The combination in which both write-allocate and write-through are false is reserved to mean "read-only". A write to a read-only page will cause a hard trap to the executive at trap vector READ_ONLY.

Combinations of these bits allow us to obtain many useful access modes, of which the following are examples:

Local-data (data $\wedge$ write-allocate) A cache miss caused by reading an operand from a local-data page causes the four-word block containing the missed word to be read over the switching network and placed in the data cache. *Writes to local-data pages do not write through to main memory.* Whenever it is important that the memory shadow of a local-data page be made identical to the cache, the "UPDATE DATA" or "KILL DATA" instruction must be executed to update main memory. It is intended that the private variables of a process be identified as local-data pages; cache sweeping will be necessary if the process ever moves to another processor.

Cached-read-data (data) A cache miss in a cached-read-data page causes the missed word to be read over the switching network and placed in the cache. No writes are allowed to a cached-read-data page; such a page is created by writing it as a local-data page, executing the instruction "UPDATE DATA" or "KILL DATA", and finally changing the appropriate page table entries to convert the page into a cached-read-data page. A cached-read-data page is destroyed by destroying the access route to the page, that is, by destroying all information about it in page tables in memory, and removing it from all page maps. Although locations from a cached-read-page may be resident in the cache, they will be replaced by new cache residents. Since locations from a cached-read page can not be marked in any cache, no cache sweep is necessary to destroy such a page.

Static-code (instructions). A static-code page is similar to a cached-read-data page, that is, it is cached, created, and destroyed in the same way as a cached-read-data page. However, locations on a static-code page can be accessed only as instructions. It is intended that shared routines will be identified as static-code.

Dynamic-code (instructions ∧ data ∧ write-allocate). In order to avoid the large overhead of cache sweeping and page-table modification, some programs may write dynamic-code pages and execute them immediately. Dynamic-code pages are the same as local-data pages, except that locations from these pages may be accessed both as instructions and as data.

Shared-data (data ∧ read through ∧ write-through). Words from shared-data pages are never placed in the cache. A write to a shared-data page writes through to main memory without writing in cache (write-allocate is false), and a read from a shared page reads directly from main memory. I/O registers and munch registers (see Section 3.4) are on shared-data pages. In addition, locations which are heavily shared by multiple processors are on shared pages, eliminating the necessity to perform repeated cache sweeps when passing small amounts of data between processors.

3.4 Synchronization

Several mechanisms are provided to allow efficient process synchronization: interrupts, read-modify-write memory capability, munch registers, and hardware task queues.

3.4.1 Interrupts

Each BSM_i contains one *interrupt controller*, which is directly attached to processor P_i by four *interrupt lines*, INT_LINE<0:3>, as shown in Figure 2.1-1. The function of the interrupt controller is to receive interrupts from I/O devices (both low- and high-speed), which are directly connected to the interrupt controller, and from processors, which send interrupts through the crossbar, and to assert the interrupt lines accordingly.

The interrupt controller contains four 36-bit registers, INT_REG[0:3]<0:35>, which can be accessed over the crossbar as memory locations. The sole function of the interrupt controller is to set INT_LINE<i> if and only if INT_REG[i]<j>=1 for some j. Each I/O device is connected to one bit of one INT_REG; the I/O device interrupts by setting that bit. No I/O device is connected to INT_REG[0]. Any processor P_i may interrupt any other processor P_j by setting some bit in P_j's INT_REG[0]. Specifically, "SET INTERRUPT J,I" executed by any processor sets location J to (J or I) using a read-modify-write memory access. By convention, when P_i interrupts P_j, P_i will set bit i in P_j's INT_REG[0].

Whenever INT_LINE<k> to processor P_j is asserted, P_j compares its current priority (PRIO) to k, which is the priority of the interrupt. If and only if PRIO is less than k, P_j will acknowledge the interrupt by resetting a bit in its interrupt register INT_REG[k] under micro-code control. If more than one INT_LINE is asserted, then the INT_LINE with the higher priority will be acknowledged first.

After acknowledging the interrupt, P_j interrupts to the executive at a specific *interrupt vector*, the address of which depends upon the identity of the I/O device or processor which caused the interrupt; that identity is fully determined by the index of the bit in INT_REG[k] which caused the interrupt and which P_j reset in acknowledgement. Section 3.7.4.10 contains a complete description of flow of control during an interrupt after interrupt acknowledgement.

3.4.2 Read-Modify-Write

The crossbar network has the capability to perform read-modify-write memory cycles. This capability is used to implement special instructions such as "TEST AND SET", and "INTERRUPT", and to implement hardware queues. Normal instructions which access a memory location as both a source and the destination do not use read-modify-write memory access capability.

To perform a read-modify-write memory access, processor P_i, under micro-code control, sends a read-modify-write request to the crossbar. The crossbar causes the addressed memory module to read and returns the data to P_i. The crossbar prevents any other processor from accessing the selected memory module until P_i returns a write.

3.4.3 Munch Registers

We borrow the concept of munch registers from Steele ([Steele 1975]). Associated with each processor is at least one munch register. Munch registers are identified by their page table entries as being shared-data. The instruction "MUNCH SKIP NOT FULL ADR M,V" executed by processor P_i translates V into a real address R and writes R into the munch register at address M. The munch register controller allows R to be written into M if and only if *no other munch register contains R*, otherwise the controller writes zero into M. After writing to M, P_i reads M and skips if and only if the result is non-zero, that is, if and only if there was no conflict.

Munch registers can also be read and written with normal memory-reference instructions, in particular, a munch register M is returned to the free state by writing zero into it. Note that the munch register controller always checks conflicts on writes to munch registers, even in the case in which zero is being written to the munch register.

Munch registers are designed primarily to allow processors to enqueue on very small data elements without wasting storage by having a separate flag for each element. Munch registers are implemented as an associative memory with special control logic connected to a memory port. Any munch register is accessible by any processor, but munch registers will be allocated by software to processors, and that allocation will be enforced by the memory mapping hardware. There are enough munch registers to allocate several to each processor.

Note that the executive will update the munch registers when evicting or re-loading munched pages.

3.4.4 Hardware Queues

There exist several hardware queues which are addressed as memory locations. Special instructions such as "QUEUE" and "DEQUEUE" manipulate the hardware queues by using read-modify-write memory accesses. For example, when processor P_i performs a "QUEUE SKIP NOT FULL ADR Q,X" instruction, in a read-modify-write cycle, it reads the state of the hardware queue at address Q, and if the queue is not full, places X on the queue and skips to ADR. If the queue is full, then P_i places nothing on the queue (writing to a dummy location in the queue controller in order to satisfy the crossbar that the read-modify-write cycle has been completed) and does not skip.

Hardware queues allow the rapid dispatching of tasks without the necessity of using munch registers or TEST AND SET instructions. Both FIFO and LIFO queues are being provided.

3.5 Status

The hardware register STATUS_REG<0:35> contains both processor and user status. The processor status can be accessed only in executive mode, whereas the user status can be accessed in either executive or user mode.

3.5.1 Processor Status

The processor status portion of STATUS_REG is accessible only by means of the instructions "READ FULL STATUS", and "WRITE FULL STATUS JUMP"; these instructions read or write the entire STATUS_REG, including both processor and user status. The processor identity (PROCESSOR_ID<0:35>) is a unique number for each physical processor; it is considered part of the processor status and is read with the instruction "READ PROC ID". The execution of any of these instructions in user mode causes a hard trap to the executive at trap vector address STATUS_ACCESS.

The fields included in the processor status are as follows:

SP_ID<0:4>
Stack pointer identity. This field is the address of the register used as the stack pointer in some instructions. The stack limit is always the next contiguous register. SP refers to the stack pointer register, and SL refers to the stack limit register.

EXEC_FILE<0:1>
Executive register file. This field is the index of the register file used for operands and addressing in the executive address space. (See Section 3.7.2 for reserved file indices.)

USER_FILE<0:1>
User register file. This field is the index of the register file used for operands and addressing in the user address space. Furthermore, when executing in the executive address space, the lowest 32 single-words of the address space refer to these registers, not to real memory locations. (See Section 3.7.2 for reserved file indices.)

USE_SHADOW
Use shadow registers. If this bit is set, then memory addresses 0 to 127 (the first 32 single-words of the virtual address space), when mapped in the user address space, actually access memory locations; otherwise, these memory addresses access the user register file.

PRIO<0:2>
Processor priority. Interrupts with priority less than or equal to this number will not interrupt the processor.

EXEC_MODE
Executive mode. The executive is currently in execution if and only if this bit is set; privileged instructions may be executed without trapping.

TRACE_TRAP
Trace trap. After any instruction, perform a hard trap to the executive at trap vector address TRACE. The effects of changing this bit do not appear until *after* the instruction following the instruction which changes the status word.

3.5.2 User Status

The user status portion of STATUS_REG is accessible in either user mode or executive mode, only by means of the instructions "READ USER STATUS", and "WRITE USER STATUS JUMP". This portion of the STATUS_REG will also be called USER_STATUS_REG.

The fields included in USER_STATUS_REG are as follows:

COND<0:4>
Arithmetic condition codes negative, zero, overflow, carry-out, and underflow. Every floating-point and integer operation may set these condition codes. Only floating-point operations set underflow.

INT_TRAP
Allow integer overflow traps. Integer overflow will soft trap to the trap vector at address INT_OVFL.

FLOAT_TRAP
Allow floating-point underflow and overflow user traps. Floating-point underflow will soft trap to the trap vector at address FLOAT_UNDFL. Floating-point overflow will soft trap to the trap vector at address FLOAT_OVFL.

PRE_LIMIT<0:5>
Prenormalization limit. If a floating-point number is prenormalized more than this amount and PRE_TRAP is true, then a soft trap will occur to the trap vector at address PRE_OVFL. The value PRE_LIMIT<0:5>=63 is reserved by the hardware to mean "never trap".

POST_LIMIT<0:5>
Postnormalization limit. If a floating-point number is postnormalized more than this amount and POST_TRAP is true, then a soft trap will occur to the trap vector at address POST_OVFL. The value POST_LIMIT<0:5>=63 is reserved by the hardware to mean "never trap".

3.6 Input/Output

The processor performs I/O by manipulating I/O registers which are logically located in the main memory address space and physically located in the I/O controllers.

Each I/O device (both low- and high-speed) has a direct connection to its I/O registers (which are located in one I/O controller). Protection of I/O devices from access by unauthorized processes is accomplished by using the memory protection facilities (Section 3.3). I/O registers must be marked in each page map as shared-data so that they will not be placed in the cache.

As explained in Section 3.4.1, each I/O controller can interrupt only one processor, and therefore each I/O device can directly interrupt only one processor. However, any processor receiving an interrupt may forward that interrupt to any other processor by means of the interprocessor interrupt facility.

3.7 Instruction Set Definition

The processor executes instructions which are from one to three 36-bit words in length. With certain restrictions on the addressing modes, many instruction types can operate on 9, 18, 36, or 72 bit operands, called quarter-word (qw), half-word (hw), single-word (sw), and double-word (dw), respectively.

We first consider the justifications for a 36-bit word (as opposed to a 32-bit word). First, without devastating changes, the LLL Filter instruction format would not fit into 32-bits. Furthermore, it is important for an entire address to fit in a single word, and for there to be room left in the word to specify an index register and an indirect bit (as in the PDP-10). Finally, a 36-bit word allows reasonably large addresses to be packed in a half-word; a 32-bit word does not.

The disadvantages of a 36-bit word are (1) that it is incompatible with a number of machines, and (2) that it makes addressing standard 8-bit bytes difficult. In answer to the second problem, the LLL Filter allows quarter-word addressing (a quarter-word is a 9-bits); considering the exponentially decreasing cost of memory, it seems reasonable to waste the extra bit in those applications which cannot find a use for it.

In order to allow more efficient utilization of memory, the LLL Filter includes the PDP-11 feature which allows most instructions to operate on multiple operand sizes; in this case the sizes are quarter-word (9-bits), half-word (18-bits), single-word (36-bits), and double-word (72-bits). One major problem with multiple operand types is the necessity to shift addresses; the IBM-370 and PDP-11 can spend a large fraction of their time shifting array indices. To overcome this problem, the LLL Filter includes addressing modes which automatically allow an index to be shifted left 0, 1, 2, or 3 places; this feature makes it convenient for a compiler to work with arrays composed of any of the basic operand types.

Another design goal was to simplify the task of writing a compiler that produces compact and efficient code. All operand addressing in the LLL Filter is completely symmetrical, that is, every operand uses the same address computation procedure. The LLL Filter also provides the reverse form of all non-commutative operations, and allows indexing off of local variables on the stack. Because of the operand addressing symmetry, a compiler can perform code generation almost independently of deciding which variables are to be on the stack and which are to be in high-speed registers.

The most important single design goal was to allow convenient access to a very large address space; such an address space may allow a new architecture to survive for a long period even in face of exponentially decreasing memory costs, thus amortizing the expensive software development effort.

The LLL Filter architecture includes multiple-word instruction formats (one to three 36-bit words) in order to allow sufficiently powerful instructions that the code density lost in specifying large addresses is not important. Using the LLL Filter instruction format, the total number of bits needed to represent a program is in general less than the number needed to represent the same program on the IBM-370, and approximately equal to the number needed on the PDP-10. Section 3.7.5 gives a number of examples to substantiate this claim.

The instruction set is horizontally micro-coded in writeable control store. The instruction set definition which follows is fixed in some respects, for example, in the operand addressing modes, but the data paths in the implementation are sufficiently general and the control store is large enough that the instruction set can be extensively modified, either by the inclusion of new special

purpose instructions, or by the replacement of existing instructions; such modification simply involves writing new micro-code.

3.7.1 Notation and Conventions

Bits in a word, quarter-words, and half-words are numbered from left to right (most significant to least significant). The bits in a word are numbered from 0 to 35, and subfields in a word are referenced by the notation X<i:j>, where i is the bit number of the high-order bit in the field, and j is the bit number of the low-order bit of the field. Using this notation, the quarter words in a word X are X<0:8>, X<9:17>, X<18:26>, and X<27:35>; these quarter-words are numbered 0, 1, 2, and 3, respectively.

In a number of places in the description, a field is used as a signed two's complement number. If F is such a field, then the notation SIGNED_F (or simply S_F) refers to F considered as a two's complement number.

Some instructions operate on a pair of data objects, such as two quarter words, or two single words. If X is the first object of such a pair, then second one is refered to as NEXT_X. X and NEXT_X are contiguous, that is, if X and NEXT_X are addresses of objects of length L quarter words, then NEXT_X = X + L.

3.7.2 Registers and Memory

The processor hardware includes 4 stacks of 32 registers each, REG_FILE[0:3][0:31]<0:35>.

REG_FILE[USER_FILE<0:1>][0:31] and REG_FILE[EXEC_FILE<0:1>][0:31] will sometimes be called USER_R[0:31] and EXEC_R[0:31], respectively. R[0:31] will mean USER_R[0:31] if EXEC_MODE=0, and will mean EXEC_R[0:31] if EXEC_MODE=1.

Certain instructions make use of a *stack pointer* and *stack limit* register, called SP and SL, respectively. SP will mean R[SP_ID], and SL will mean R[SP_ID+1], where SP_ID is the stack pointer identity field in the STATUS_REG.

Registers can be addressed as memory; the lowest 32 single-word addresses of the executive address space refer to EXEC_R[0:31], and the lowest 32 single-word addresses of the user address space refer to USER_R[0:31].

REG_FILE[0] is dedicated for use by the hardware and micro-code. REG_FILE[0][0:31] will also be called TEMP[0:31], since it contains many hardware temporary locations. In the following sections we will refer to some registers in REG_FILE[0] by name as follows:

EXEC_SEG_BASE_REG	Executive segment base register.
USER_SEG_BASE_REG	User segment base register.

REG_FILE[0] can be accessed by the executive by setting USER_FILE<0:1>=0 and referencing the registers as memory locations in the user address space.

REG_FILE[1:3] are not dedicated; it is intended that they will contain executive and user registers.

The instruction set gives hardwired functions to some registers, as shown below:

R[0]	no short indexing allowed
R[1]	no short indexing allowed
R[2]	no short indexing allowed
R[3]	program counter (PC)
R[4]	low-order word of temporary register RTA (RTA[0])
R[5]	high-order word of temporary register RTA (RTA[1])
R[6]	low-order word of temporary register RTB (RTB[0])
R[7]	high-order word of temporary register RTB (RTB[1])
R[9]	general purpose register
.	.
.	.
.	.
R[30]	general purpose register (receives first parameter of trap)
R[31]	general purpose register (receives second parameter of trap)

The registers RTA and RTB can be used as a third address in some instructions, as explained in Section 3.7.3.

The instruction set can manipulate the R registers as easily as memory locations, and special instructions are provided for saving and restoring R registers during interrupts, traps, and subroutine calls.

Unless otherwise specified, all addresses in this description are quarter-word addresses. Directly addressable main memory consists of 2^{30} quarter-words which can also be accessed as half-words, single-words, or double-words.

In order to facilitate computing with data of multiple precisions (qw, hw, sw, and dw), instructions are included for each precision. Some instruction types operate on only a subset of the possible precisions, for example, floating point instructions operate only on single-word and double-word operands. Most instructions assume that both source operands and the destination are of the same precision, although some instructions are provided for converting from one precision to another.

Half-word operands must lie on half-word boundaries, single-word operands on single-word boundaries, and double-word operands on double-word boundaries. Any violation of this boundary rule will cause a hard trap to the executive trap vector at address BOUNDARY_ERROR. The registers in the register file are considered to lie on contiguous single-word boundaries. Instructions must lie on single-word boundaries.

Note that a quarter-word add, for example, specifying R[16] and R[17] as source operands and R[18] as the destination operand, will add the high-order quarter-word of R[16] to the high-order quarter-word of R[17], and store the quarter-word result in the high-order quarter word of R[18].

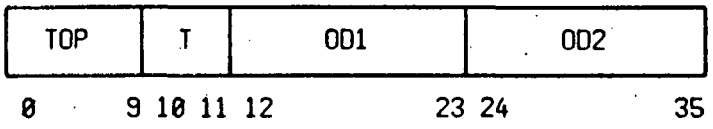
3.7.3 Instruction Formats

Every instruction is either one, two, or three 36-bit words in length. The first instruction word includes the opcode, and specifies part or all of the address computation for the operands. The

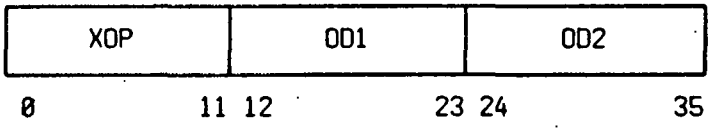
second and third instruction words are used for long immediate constants and for extended addressing.

Four basic instruction formats apply to the first word of an instruction, as follows:

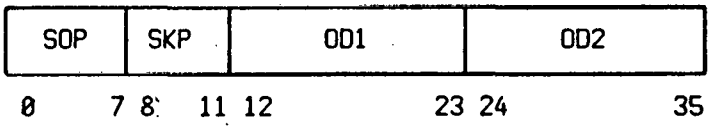
Three-Address Instruction



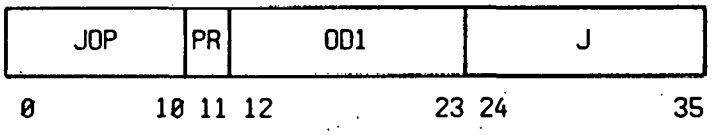
Two-Address Instruction



Skip Instruction



Jump Instruction



TOP, XOP, SOP, and JOP are opcodes. OD1 and OD2 are general Operand Descriptors; they specify general operands which can be memory locations, registers, or constants. (It should be noted that the address computation algorithm is identical for the OD1 and OD2 fields.) The T field specifies how to use the registers RTA and RTB as a third operand in the instruction. SKP and J specify a skip distance and a jump distance or jump address, respectively. PR specifies whether to use J as an offset to the PC or as the descriptor of a memory address (as are OD1 and OD2).

The three-address instruction format allows two general memory addresses to be specified, along with a third operand, either RTA or RTB. This instruction format provides most of the advantages of a true three-address format (that is, the elimination of "move" instructions to make copies of operands at the beginning of an expression), but costs only two bits in the instruction word.

The two-address instruction format allows two general memory addresses to be specified, and is primarily used in data transmission instructions (which have one source and one destination operand).

The skip instruction format allows a forward skip of from 0 to 7 words, or a backward skip of 1 to 8 words (from the location of the current instruction); it is useful for implementing small conditional loops and IF-THEN-ELSE statements.

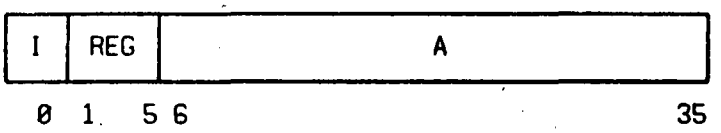
A jump instruction having PR=1 can jump anywhere in the range of PC+2047 to PC-2048 words (where PC is the address of the next instruction), and in that case requires no additional word to specify the jump address. If PR=0, J may specify any memory address, at the possible expense of requiring an additional instruction word.

3.7.3.1 General Operand Address Specification

We first consider some notation and conventions. If X is the address of a memory location, then M[X] will mean the contents of that location. The length of M[X] will be clear from context, it may be either quarter-word, half-word, single-word, or double-word.

Indefinite-level indirect addressing is denoted using the character "@", and is defined as follows: Let IAP (Indirect Address Pointer) be the *contents* of a register or memory location:

IAP: Format for Indirect Address Pointer



Then @IAP is an *address*, defined as follows:

I	REG	@IAP
0	=0	A
1	0	@M[A]
0	≠0	A+R[REG]
1	≠0	@M[A+R[REG]]

The evaluation of all operands (including the jump or skip destination) logically occurs *before the execution of the instruction* (and before the PC is updated).

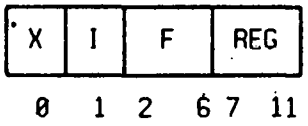
The evaluation of a general operand proceeds in two steps, which are discussed in the following sections.

3.7.3.1.1 Short-Operand Address Calculation

A *short operand* can be one of the 32 registers R[0:31], a memory location which is addressed as a short offset from a register, a short immediate constant, or several other entities. The name "short operand" derives from the fact that such operands require only a short descriptor in the instruction. An exact definition follows.

The 12-bit operand descriptor fields (OD1 and OD2) specify short operands, and may also specify extended indexing. They have the following format, where the bit numbers are relative to the origin of the field:

OD: Format for OD1 and OD2



These fields specify extended indexing (X), indirection (I), a short offset (F), and a register name (REG). A short operand (SO) is defined as the location specified by the fields I, F, and REG; it is evaluated as follows:

I	F	REG	Short Operand (SO)	Mode Name
0	0-31	0	R[F]	register-direct
1	0-31	0	M[@R[F]]	register-indirect
0	0-31	1	S_F	short-constant (-16 to +15)
1	0-31	1	0	short-zero
0	0-31	2	0-31	(reserved)
1	0-31	2	0-31	(reserved)
0	0-31	3-31	M[R[REG]+S_F*4]	short-indexed
1	0-31	3-31	M[@M[R[REG]]+S_F*4]	short-indexed-indirect

IF X=0, then the value of the operand described by OD is simply SO, as above. Addressing modes in which X=1 are described in the next section.

All memory address mapping is done in the own address space when calculating short operands.

Short-zero mode is provided only as an escape to allow absolute memory addressing; short-zero mode with X=1 addresses memory absolutely, as explained in the next section.

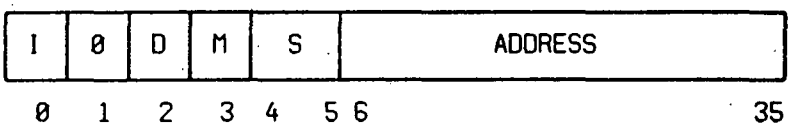
It is intended that all of the simple variables (i.e. local variables on the stack and own variables) be accessed directly in short-indexed mode. Short-indexed mode is of such utility that we call locations accessed using this mode *pseudo-registers* (or P registers).

The only variables that can not be conveniently addressed using the short-operand addressing modes are arrays and variables which are allocated at absolute addresses in memory. Such locations are accessed by using extended addressing modes, as described in the next section.

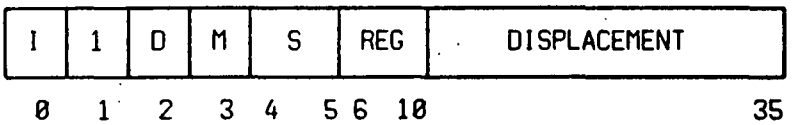
3.7.3.1.2 Extended Addressing

Extended addressing is specified by setting the X bit in the operand descriptor (OD1 or OD2). In extended addressing mode, the next word in the instruction stream is used in the operand calculation. This word is either the second or third word of the instruction, and has one of these formats:

E: Format for fixed-base extended addressing



V: Format for variable-base extended addressing



C: Format for long constant

Given that the X bit is set in the operand descriptor (OD1 or OD2), then, with one exception, the additional word in the instruction is used to calculate an extended address, and is interpreted either as fixed-base format (E), or variable-base format (V), depending upon the value of the V bit (bit 1) of the word itself. The exception noted is that if the operand descriptor specifies short-constant mode, then the additional word is interpreted as a long constant (C), and provides a 36-bit immediate constant which is used as the operand. This addressing mode is called long-constant mode. In the following discussion we will ignore long-constant mode.

The first step in the extended address calculation is to calculate the base address BASE to be used in the indexing operation. If the the additional word in the instruction has fixed-base format (E), then BASE is given by

$$BASE := ADDRESS$$

If the additional word in the instruction has variable-base format (V), then the register R[REG] contains the base address, and DISPLACEMENT is an additional offset as follows:

$$BASE := R[REG] + SIGNED\ DISPLACEMENT$$

Let SO be the short operand specified by the operand descriptor. If the indirect bit (I) in the extended word is zero, then the value of the operand addressed by using extended addressing is

$$M[BASE + SO*2^S]$$

If the indirect bit is one, then the value of the operand is

$$M[@M[BASE + SO*2^S]]$$

It should be noted that the extended addressing mode always includes an indexing operation, but that if short-zero is the short-operand addressing mode, then SO=0, and the address computed using extended addressing is just BASE. Note also that automatic address shifting occurs in extended addressing mode, that is, the value SO is shifted left by S bits (where S is a field in the extended word) before being added to BASE.

The M bit facilitates communication between the executive and the user, which operate in different address spaces, by allowing instructions executed by the executive to have either operand mapped in either the user or the executive address space. Only the *final* address mapping in the operand calculation procedure is affected by the M bit, as follows:

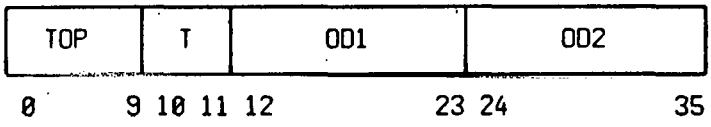
M	EXEC_MODE	Final Mapping Space
0	0	User address space.
0	1	Executive address space.
1	0	(Hard trap to REF_EXEC.)
1	1	User address space.

Table 3.7.3.1.2-1
M Bit Interpretation

The duplicate bit (D) specifies that the two operands of the instruction use the same extended instruction word; it simply inhibits the program counter from being incremented after the first operand is evaluated. This feature is useful when both operands are elements of the same array, but are accessed using different index registers.

3.7.3.2 Three-Address Instructions

Three-address instructions have the format:



The TOP field includes the opcode and specifies the precision (qw, hw, sw, or dw) of the operation.

Fields OD1 and OD2 are general operand descriptors, as described in Section 3.7.3.1; they may denote R registers, P registers, general memory locations, or immediate constants.

The two-bit T field specifies whether RTA or RTB is used as the third address of the instruction, where OD1 and OD2 specify the other two addresses. Specifically, the operation evoked by a three-address instruction is described using the names DEST, S1, and S2, for example, DEST←S1⊗S2, or DEST←S2⊗S1, where "⊗" means the operation evoked by the TOP field, and S2, S1, and DEST have meanings as shown in the following table. In this table, OP1 means the operand described by field OD1, and OP2 means the operand described by field OD2:

T	DEST	S1	S2
00	OP1	OP1	OP2
01	OP1	RTA	OP2
10	RTA	OP1	OP2
11	RTB	OP1	OP2

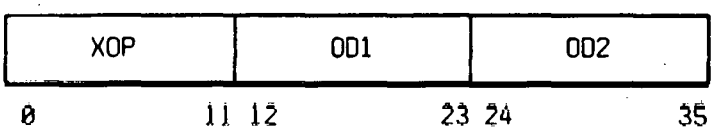
Table 3.7.3.2-1
T Field Meaning

These addressing modes are sufficient to allow any FORTRAN assignment statement except those of the form "A←B+C" or "A←B(I+J)*C(K+L)+D(M+N)*E(L+P)" to be evaluated with no move instructions to make copies of operands or to store away the result of the expression. The first exception clearly needs a full three address instruction if it is to be evaluated in one instruction, and the second requires a third RT register. Because of the binary nature of arithmetic operators, all other types of expressions require only two RT registers. For example, if two of the subscripts of the second example were the same, or if one the subscripts were a simple local variable, or were of the form "I+J+K", then two RT registers would be sufficient to evaluate the expression with no move instructions. In Section 3.7.5.2 some examples are given which show code using the RT registers.

Preliminary evidence suggests that for typical FORTRAN assignment statements, LLL Filter code using the RTA and RTB registers contains .5 to .7 times the instructions necessary for the PDP-10.

3.7.3.3 Two-Address Instructions

Two-address instructions have the format:

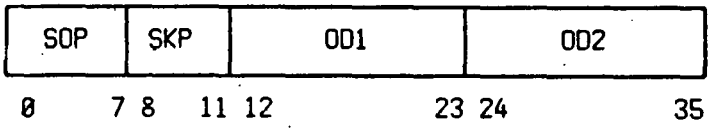


The XOP field includes the op-code and specifies the precision (qw, hw, sw, or dw) of the operation.

Fields OD1 and OD2 are general operand descriptors, as described in Section 3.7.3.1; they may denote R registers, P registers, general memory locations, or immediate constants.

3.7.3.4 Skip Instructions

Skip instructions have the format:



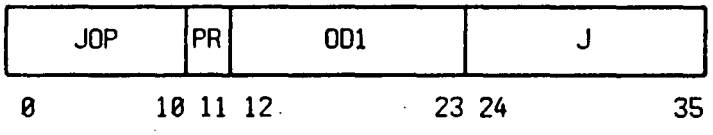
The SOP field includes the op-code, specifies the precision (qw, hw, sw, or dw) of the operation, and specifies the condition on which a skip will be taken.

Fields OD1 and OD2 are general operand descriptors, as described in Section 3.7.3.1; they may denote R registers, P registers, general memory locations, or immediate constants.

The SKP field contains a skip distance in words. If the skip condition is false at the end of the current instruction, then the next instruction to be executed is the next sequential instruction. If the skip condition is true, then the quarter-word address of the next instruction to be executed is PC+4*SIGNED_SKP, where PC is the address of the current instruction.

3.7.3.5 Jump Instructions

Jump instructions have the following format:



The JOP field includes the op-code, specifies the precision (qw, hw, sw, or dw) of the operation, and specifies the condition on which a jump will be taken.

Field OD1 is a general operand descriptor, as described in Section 3.7.3.1; it may denote an R register, P register, general memory location, or immediate constant.

The J field specifies a jump destination JUMPDEST. It is interpreted differently depending upon the value of the PC-relative (PR) bit. If the PR bit is one, then JUMPDEST is PC+4*SIGNED_J where PC is the address of the current instruction. If the PR bit is zero, then J is taken to be a general operand descriptor (OD2), and JUMPDEST is the *address* of the operand described by that operand descriptor.

Jumps to the user address space performed in executive mode hard trap to the executive at trap vector address JUMP_USER; all control transfers to the user address space must be performed by means of "TRAP EXEC", "RETURN FULL STATUS", and "WRITE FULL STATUS JUMP" (which may change the mode to user, then jump).

3.7.4 Instruction Descriptions

This section describes the instruction set which is currently being micro-coded for the LLL Filter. For the sake of clarity, we have not used a formal descriptive system, but have developed our own set of largely intuitive descriptive mechanisms and conventions.

Each instruction is defined by showing the *opcode string* of the instruction and the *operation* of the instruction. The opcode string contains *terms* which are separated from each other by one or more spaces and together uniquely define the instruction.

This section also describes sequences of operations which are not instructions (for example, the interrupt procedure). The opcode string column for such sequences shows a *function name* (in italics), and the function's formal parameters. A function defined in this way may be called from the definition of any instruction.

Curly brackets are sometimes used in writing terms of the opcode. Several strings (sub-terms) may be grouped in curly brackets and separated by commas, for example {Q,H,S,D}; this notation means that any one of the bracketted strings may be substituted in place of the brackets and everything enclosed in the brackets.

The curly-bracket notation may also be used in the operation column. Let X and Y represent any two curly-bracketted strings such that the number of sub-terms X_i of X is equal to the number of sub-terms Y_i of Y. Then if X appears in the opcode column, Y may appear in the operation column, with the following meaning: If an opcode is constructed by choosing X_i in place of the term X, then the operation of that opcode is formed by replacing Y by Y_i. In some cases, more than one curly-bracketted term is used in the opcode column; let W and X be two such terms. In this case, if curly-bracketted term Y appears in the operation column, Y corresponds to only one of W and X; that correspondence will not be formally specified, but will be obvious.

Undefined but intuitive functions appear in italics in the operation column.

The names OP1 (OPerand 1), OP2 (OPerand 2), S1 (Source 1), S2 (Source 2), and DEST (DESTination), have the meanings described in Section 3.7.3.2.

Let X represent any of the strings OP1, OP2, S1, S2, or DEST. Then ADDRESS_X means the memory address of X. Note that registers have memory addresses.

During the execution of one instruction, "PC" will mean the address of the instruction currently in execution, "PC_NEXT_INSTR" will mean the address of the next instruction in the execution sequence, and "PC_LAST_INSTR" will mean the address of the previous instruction in the execution sequence.

The LLL Filter instruction set includes "reverse operations" for all non-commutative instructions with two source operands and a destination operand, that is, instructions of the form "DEST←OP1 ⊗ OP2" where "⊗" is a non-commutative operator. A reverse operation is indicated by the inclusion of the term "V" in the opcode string. Reverse operations reverse the order of their source operands before performing the operation. For example, "SUB V OP1,OP2" means "OP1←OP2-OP1" whereas "SUB OP1,OP2" means "OP1←OP1-OP2".

Reverse operations are provided in order to allow evaluating "A←B ⊗ A" and "A←B ⊗ RTA" in one instruction, where A and B here represent memory addresses, RTA is a special temporary register (see Section 3.7.3.2), and "⊗" is a non-commutative operator.

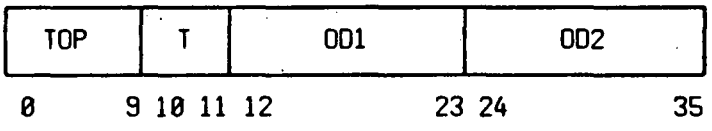
Note that the opcode strings shown in the following sections are not necessarily *assembler mnemonics*; they are simply unique names for the hardware operations. An assembler will allow omission of some terms and simplification of others; an intelligent assembler, for example, would infer the "V" term of the opcode string from the order of the three operands of the instruction.

3.7.4.1 Integer Instructions

Integers are represented in two's complement notation. All integer instructions operate on data of any integer precision, that is, quarter-word (Q), half-word (H), single-word (S), or double-word (D). The precision of the operation is indicated by including the appropriate term (Q, H, S, or D) after the opcode. For operations which take two operands, both operands must be of the same precision.

Integer operations are done in the precision of the source operands, except for extended precision operations (eg. "MULT L {Q,H,S,D}"), which are done in double precision.

3.7.4.1.1 Integer Arithmetic

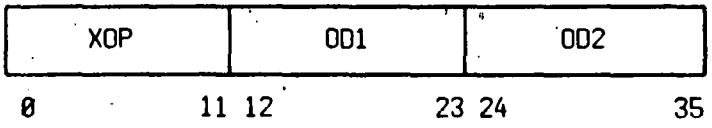


Reverse operations are provided for the non-commutative operations SUB, QUO, REM, and DIV.

Extended precision operations (eg. long multiply and long divide) are indicated by including the term "L" (Long) in the opcode string.

Opcode String				Operation
ADD			{Q,H,S,D}	DEST←S1+S2
SUB			{Q,H,S,D}	DEST←S1-S2
SUB		V	{Q,H,S,D}	DEST←S2-S1
MULT			{Q,H,S,D}	DEST←S1*S2
MULT	L		{Q,H,S,D}	(DEST,NEXT_DEST)←S1*S2
QUO			{Q,H,S,D}	DEST←S1/S2
QUO		V	{Q,H,S,D}	DEST←S2/S1
QUO	L		{Q,H,S,D}	DEST←(S1,NEXT_S1)/S2
QUO	L	V	{Q,H,S,D}	DEST←(S2,NEXT_S2)/S1
REM			{Q,H,S,D}	DEST←S1 *mod* S2
REM		V	{Q,H,S,D}	DEST←S2 *mod* S1
REM	L		{Q,H,S,D}	DEST←(S1,NEXT_S1) *mod* S2
REM	L	V	{Q,H,S,D}	DEST←(S2,NEXT_S2) *mod* S1
DIV			{Q,H,S,D}	DEST←S1/S2 NEXT_DEST←S1 *mod* S2
DIV		V	{Q,H,S,D}	DEST←S2/S1 NEXT_DEST←S2 *mod* S1
DIV	L		{Q,H,S,D}	DEST←(S1,NEXT_S1)/S2 NEXT_DEST←(S1,NEXT_S1) *mod* S2
DIV	L	V	{Q,H,S,D}	DEST←(S2,NEXT_S2)/S1 NEXT_DEST←(S2,NEXT_S2) *mod* S1

3.7.4.1.2 Increment and Decrement



The increment (INC) and decrement (DEC) instructions provide the capability to perform either of the operations OP1←OP2+1 or OP1←OP2-1 in one instruction.

Opcode String		Operation
INC	{Q,H,S,D}	OP1←OP2+1
DEC	{Q,H,S,D}	OP1←OP2-1

3.7.4.2 Floating Point Instructions

Floating point precisions are single-word (S), and double-word (D), whereas integer precisions are quarter-word (Q), half-word (H), single-word (S), and double-word (D). The floating point arithmetic instructions require one floating point precision to be specified, and the floating point translation instructions require either a floating point precision and an integer precision or two floating point precisions to be specified.

Single-precision floating point numbers have the following format:

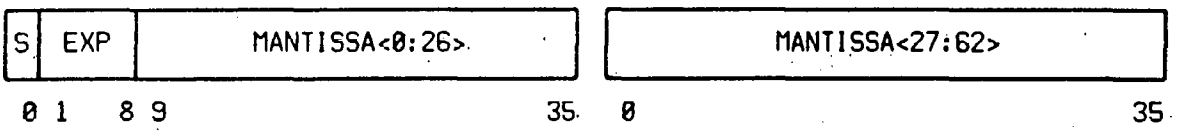
Single-Precision Floating Point Number

where S is the sign, EXP is an excess-128 exponent of 2, and MANTISSA is a normalized binary fraction.

If X is a positive floating point number (single or double precision), then the floating point number -X is represented by the two's complement of X, so that integer comparison operations yield the correct results for floating point operands.

Double-precision floating point numbers have the following format:

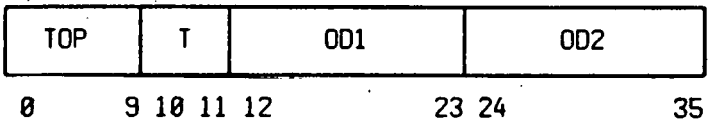
Double-Precision Floating Point Number



where S, EXP, and MANTISSA represent the sign, exponent, and mantissa of the double-precision floating point number, as above.

Any floating point operation may be either floor rounded (FR), ceiling rounded (CR), or stable rounded (SR) (see [Kahan 1973]); these modes are indicated by including the appropriate characters as a term in the opcode string. Floor rounding yields the closest floating point number less than the true result (equivalent to truncation since the number system is two's-complement), ceiling rounding yields the closest floating point number greater than the true result, and stable rounding yields the closest floating point number if that number is unique, otherwise it yields the closest floating point number with a "0" as the least-significant bit.

3.7.4.2.1 Floating Point Arithmetic



Most floating point arithmetic instructions combine two operands of one floating point precision, and store into a destination of the same floating point precision. The operation precision is indicated by including the appropriate character in the opcode string.

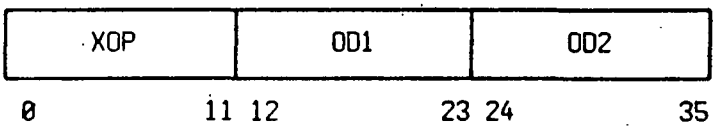
Long floating multiply (FMULT L) takes two single-word floating point numbers and multiplies them to form a double-word floating point number. Long floating divide (FDIV L) divides a double-word by a single-word and produces a single-word.

Reverse operations are provided for the non-commutative operations FSUB and FDIV.

As explained above, the terms "FR", "CR", and "SR" in the opcode string imply floor rounding (truncation), ceiling rounding, and stable rounding, respectively. For example, "FMULT FR S" means "multiply single-precision floating point numbers with truncation."

Opcode String				Operation
FADD		{FR,CR,SR}	{S,D}	DEST←S1+S2
FSUB		{FR,CR,SR}	{S,D}	DEST←S1-S2
FSUB	V	{FR,CR,SR}	{S,D}	DEST←S2-S1
FMULT		{FR,CR,SR}	{S,D}	DEST←S1*S2
FMULT	L	{FR,CR,SR}		(DEST,NEXT_DEST)←S1*S2
FDIV		{FR,CR,SR}	{S,D}	DEST←S1/S2
FDIV	V	{FR,CR,SR}	{S,D}	DEST←S2/S1
FDIV	L	{FR,CR,SR}		DEST←(S1,NEXT_S1)/S2
FDIV	L V	{FR,CR,SR}		DEST←(S2,NEXT_S2)/S1

3.7.4.2.2 Floating Point Translation



The floating point translation instructions translate floating point to integer, integer to floating point, and floating point to floating point, in each case performing floor rounding, ceiling rounding, or stable rounding.

Floating point numbers may be of any floating point precision, that is, single-word (S), or double-word (D), and integer numbers may be of any integer precision, that is, quarter-word (Q), half-word (H), single-word (S), or double-word (D). In addition to the floor-rounding (FR), ceiling-rounding (CR), and stable-rounding (SR) terms, each floating point translation opcode string includes a two character precision term; the first character specifies the destination precision, and the second character specifies the source precision. For example, "FLOAT SR SD" means "translate with stable rounding a double-word integer to a single-word floating point number." For symmetry reasons, all translate instructions include rounding modes.

Opcode String			Operation
FIX	{FR,CR,SR}	{Q,H,S,D}{S,D}	OP1←*fix*(OP2)
FLOAT	{FR,CR,SR}	{S,D}{Q,H,S,D}	OP1←*float*(OP2)
TRANS	{FR,CR,SR}	S D	OP1←*float_trans*(OP2)
TRANS	{FR,CR,SR}	D S	OP1←*float_trans*(OP2)

3.7.4.3 Arithmetic Compare Instructions

The arithmetic compare instructions compare two operands, possibly incrementing, decrementing, or adding to the destination operand, and skip (-8 to +7 words from the location of the current instruction), jump (anywhere), or trap (to a fixed virtual address) conditionally on the outcome of the comparison. Throughout these sections, PC refers to the address of the current instruction.

With two exceptions, the arithmetic compare instructions assume that both operands are of single-word length. These exceptions are "SKIP {*COND*} {Q,H,S,D}", and "JUMP {*COND*} 0 {Q,H,S,D};" each allows specification of the length of the operands (Q, H, S, or D). Both operands must be of the same length.

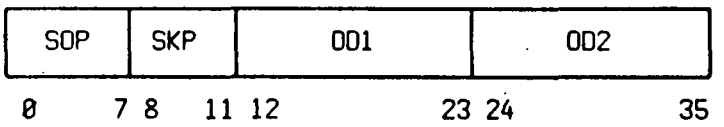
Every arithmetic compare instruction performs integer comparison. The format of floating point numbers guarantees that integer comparison produces the correct results for floating point operands. On the other hand, some arithmetic compare instructions add to the destination operand, and this addition is *integer* addition; those particular instructions are not intended to be used with floating point operands.

In the instruction definitions which follow, we have used "{*COND*}" in the opcode strings to represent "{N,G,E,GE,L,NE,LE,A}", abbreviations for the eight conditions on which an arithmetic compare instruction can skip or jump; these abbreviations mean never, greater, equal, greater or equal, less, not equal, less or equal, and always, respectively. "{*COND*}" is also used as a function symbol (with obvious meaning) in the description column of these opcodes.

The opcode strings in these instructions may include the terms in the following table, and these terms uniformly have the meanings shown:

Opcode Term	Meaning
INC	Add one before comparison.
DEC	Subtract one before comparison.
0	The comparison is with 0.

3.7.4.3.1 Arithmetic Compare and Skip

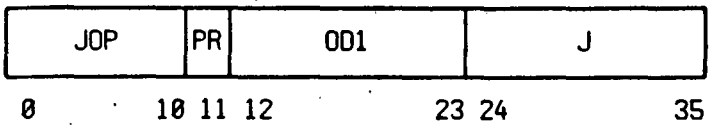


The field SKP in these instructions specifies a 4-bit (signed) skip distance (in words). Depending upon the result of the compare instruction, the next instruction to be executed is either at PC, or at PC+4*SIGNED_SKP.

These instructions are important in that they allow two general operands to be specified in a compare instruction. The SKP field of 4 bits in many cases eliminates the need for including a jump instruction after the compare.

Opcode String			Operation
INC SKIP	{*COND*}		OP1←OP1 + 1 if OP1 {*COND*} OP2 then PC←PC+4*SIGNED_SKP
DEC SKIP	{*COND*}		OP1←OP1 - 1 if OP1 {*COND*} OP2 then PC←PC+4*SIGNED_SKP
SKIP	{*COND*}	{Q,H,S,D}	if OP1 {*COND*} OP2 then PC←PC+4*SIGNED_SKP

3.7.4.3.2 Arithmetic Compare and Jump



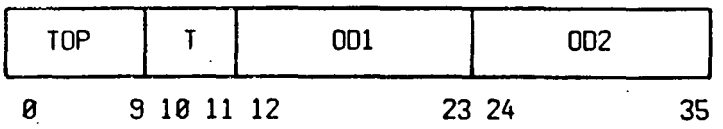
In the following instruction definitions, JUMPDEST refers to the jump destination. As described in Section 3.7.3.5, JUMPDEST is computed in one of two ways, depending upon the value of the PC-relative flag (PR). If PR is true, then J is taken to be a signed 12-bit PC offset, and JUMPDEST is PC+4*SIGNED_J. If PR is false, then J is taken to be a general operand descriptor (see Section 3.7.3.1), and JUMPDEST is the result of evaluating that operand descriptor. In either case, JUMPDEST is computed *before the execution of the arithmetic-compare-and-jump instruction.*

Note that the 12-bit PC relative jump (PR true) is included only to increase code density. All instructions in this section can be written with PR true or PR false; this symmetry makes the jump length decision relatively orthogonal to other decisions in code generation.

These instructions allow only one general operand address (OD1), since the field of the instruction normally reserved for a second operand descriptor (OD2) instead contains the jump address.

Opcode String	Operation
INC JUMP {*COND*}	OP1←OP1 + 1 if OP1 {*COND*} NEXT_OP1 then PC←JUMPDEST
DEC JUMP {*COND*}	OP1←OP1 - 1 if OP1 {*COND*} NEXT_OP1 then PC←JUMPDEST
INC JUMP {*COND*} 0	OP1←OP1 + 1 if OP1 {*COND*} 0 then PC←JUMPDEST
DEC JUMP {*COND*} 0	OP1←OP1 - 1 if OP1 {*COND*} 0 then PC←JUMPDEST
JUMP {*COND*} 0 {Q,H,S,D}	if OP1 {*COND*} 0 then PC←JUMPDEST
JUMP	PC←JUMPDEST (note: this is the same instruction as "JUMP A 0")

3.7.4.3.3 Arithmetic Compare and Set Flag



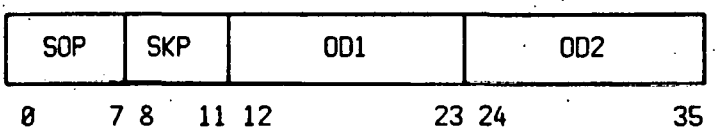
These instructions perform an arithmetic comparison and set the destination to all zeroes or all ones depending upon the result; zeroes indicate false and ones indicate true.

The source operands may be of any integer length (Q, H, S, or D). *The destination operand is always a single word.*

Opcode String	Operation
SET FLAG {COND} {Q,H,S,D}	DEST←S1 {*COND*} S2

3.7.4.4 Logical Operations

3.7.4.4.1 Logical Testing



The logical test instructions test a group of flags (OP1) under a mask (OP2) and conditionally skip (-8 to +7 words from the location of the current instruction) depending upon the result. The operands can be any integer length (Q, H, S, or D), but the flags and mask must be of the same length.

The opcode strings in the following instruction definitions contain the terms in the following table, and these terms have the meanings shown:

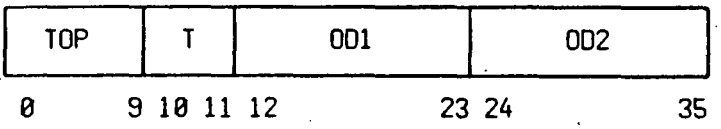
Opcode Term	Meaning
CT	Complement OP1 before anding (ie. use Complement with True).
Z	Skip if the result is Zero.
NZ	Skip if the result is Non-Zero.

If OP1 is a word of flags, and OP2 is a mask which selects a subset of the flags, then these instructions can be used to test various combinations of the flags, as follows:

Opcode	Meaning
AND SKIP Z	Skip if no selected flag is set.
AND SKIP NZ	Skip if any selected flag is set.
AND CT SKIP Z	Skip if all selected flags are set.
AND CT SKIP NZ	Skip if not all selected flags are set.

Opcode String					Operation
AND		SKIP	{Z,NZ}	{Q,H,S,D}	if (OP1∧OP2) {=,≠} 0 then PC←PC+4*SIGNED_SKP
AND	CT	SKIP	{Z,NZ}	{Q,H,S,D}	if (*not*(OP1)∧OP2) {=,≠} 0 then PC←PC+4*SIGNED_SKP

3.7.4.4.2 Logical Assignment

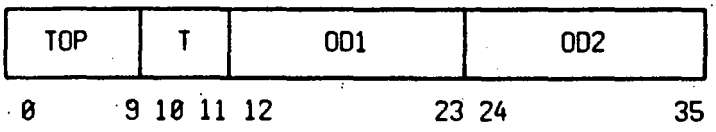


The logical assignment instructions perform a logical operation on S1 and S2 and assign the result to DEST. The operands of logical assignment operations may be any integer length (Q, H, S, or D), but both operands must be of the same length.

The terms CT and TC are used with the following meaning: CT implies that S1 is complemented before the logical operation (use Complement and True), and TC implies that S2 is complemented before the logical operation (use True and Complement).

Opcode String			Operation
AND		{Q,H,S,D}	DEST←S1∧S2
AND	TC	{Q,H,S,D}	DEST←S1∧*not*(S2)
AND	CT	{Q,H,S,D}	DEST←*not*(S1)∧S2
NOR		{Q,H,S,D}	DEST←*not*(S1)∧*not*(S2)
OR		{Q,H,S,D}	DEST←S1∨S2
OR	TC	{Q,H,S,D}	DEST←S1∨*not*(S2)
OR	CT	{Q,H,S,D}	DEST←*not*(S1)∨S2
NAND		{Q,H,S,D}	DEST←*not*(S1)∨*not*(S2)
XOR		{Q,H,S,D}	DEST←S1 *xor* S2
EQV		{Q,H,S,D}	DEST←*not*(S1 *xor* S2)

3.7.4.4.3 Shift and Rotate



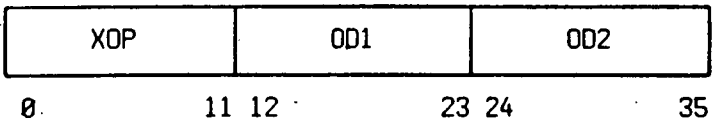
The shift and rotate instructions take operands which are any integer length (Q, H, S, or D). The shift count is always a single-word.

All shift and rotate instructions are non-commutative, therefore each instruction is provided in reverse form.

The term "A" (Arithmetic) in the opcode string implies that the operation is arithmetic, otherwise the operation is logical.

Opcode String					Operation
SHIFT	{LEFT,RIGHT}			{Q,H,S,D}	DEST←S1 logical {LEFT,RIGHT} shifted by S2
SHIFT	{LEFT,RIGHT}		V	{Q,H,S,D}	DEST←S2 logical {LEFT,RIGHT} shifted by S1
SHIFT	{LEFT,RIGHT}	A		{Q,H,S,D}	DEST←S1 arithmetic {LEFT,RIGHT} shifted by S2
SHIFT	{LEFT,RIGHT}	A	V	{Q,H,S,D}	DEST←S2 arithmetic {LEFT,RIGHT} shifted by S1
ROT	{LEFT,RIGHT}			{Q,H,S,D}	DEST←S1 rotated {LEFT,RIGHT} by S2
ROT	{LEFT,RIGHT}		V	{Q,H,S,D}	DEST←S2 rotated {LEFT,RIGHT} by S1

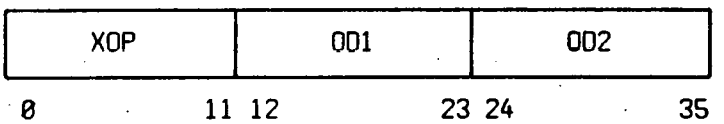
3.7.4.4.4 BIT REVERSE



BIT REVERSE reverses the bits in a quarter-word, half-word, single-word, or double-word.

Opcode String		Operation
BIT REVERSE	{Q,H,S,D}	OP1←*bit_reverse*(OP2)

3.7.4.4.5 Bit Counting

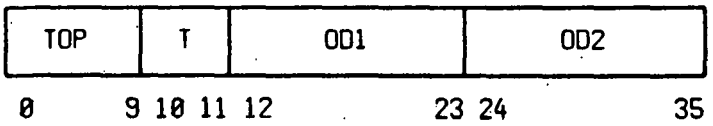


BIT COUNT counts the number of one bits in an operand; it is useful for counting the number of elements in a set, where bits in a word represent elements in a set, as in common implementations of PASCAL.

BIT FIRST finds the bit number of the first one bit of an operand; it is useful for computing the index of the first element of a set.

Opcode String		Operation
BIT COUNT	{Q,H,S,D}	OP1←(number of one bits in OP2)
BIT FIRST	{Q,H,S,D}	OP1←(index of the first one bit in OP2) (The search is from the left to the right.)

3.7.4.4.6 BIT EXTRACT

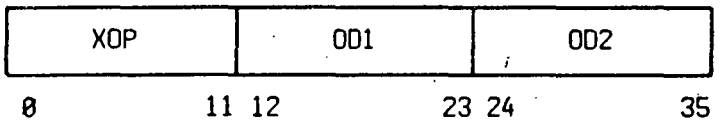


BIT EXTRACT was suggested by Professor John McCarthy; it is particularly useful for extracting a set of flags from a word in order to do an N-way branch on them. S1, S2, and DEST are assumed to be of the same length.

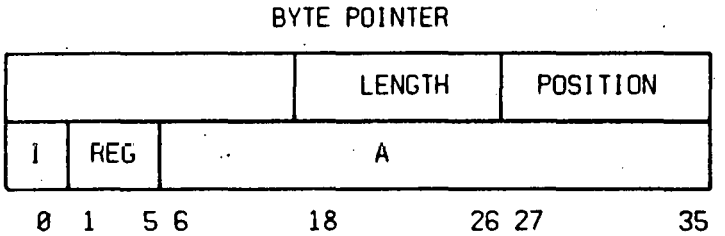
BIT EXTRACT is non-commutative, and is therefore provided in reverse form.

Opcode String	Operation
BIT EXTRACT {Q,H,S,D}	DEST is set to the value obtained by extracting the bits in S1 that correspond to the ones in S2, then squeezing them to the right in DEST.
BIT EXTRACT V {Q,H,S,D}	DEST is set to the value obtained by extracting the bits in S2 that correspond to the ones in S1, then squeezing them to the right in DEST.

3.7.4.5 Byte Pointer



The byte pointer instructions operate on bit-strings of arbitrary size (less than or equal to 36 bits), which are called *bytes*. These instructions all use a two word BYTE POINTER, which has the format:



LENGTH is the size of the byte, and POSITION is the bit-number of the first bit in the byte. The second word of the BYTE POINTER is a standard Indirect Address Pointer (see Section 3.7.3.1), which evaluates to the address of the word which contains the byte.

The LENGTH and POSITION fields are each 9 bits long, therefore quarter-word instructions can be used to manipulate them. The LENGTH and POSITION fields must specify a byte contained entirely within a word. When incrementing a BYTE POINTER, the hardware adds LENGTH to POSITION, then, if the result is greater than 35, sets POSITION to 0 and increments A. Byte-adjustment is similar.

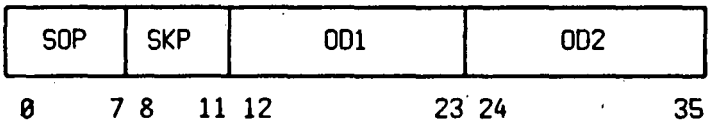
The function *byte* takes an argument which is the address of a byte pointer. The value of *byte*(X) is the bit string described by the byte pointer X.

Opcode String	Operation
LBYTE	Load BYTE OP1←*byte*(OP2)
DBYTE	Deposit BYTE BYTE(OP1)←OP2
ADJ BYTEP	ADJust BYTE Pointer OP1←OP1 byte-adjusted by OP2
LBYTE INC	Load BYTE and INCrement OP1←BYTE(OP2) OP2←OP2 byte-incremented
DBYTE INC	Deposit BYTE and INCrement BYTE(OP1)←OP2 OP1←OP1 byte-incremented

3.7.4.6 List Manipulation

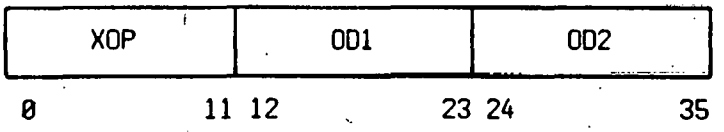
The list manipulation instructions operate on lists which have two-word list headers, where the first word points to the first element of the list, and the second word points to the last element of the list. An empty list is represented by zero in the first word of the list header. These lists are assumed to be linked together by the first word of each element; the last element contains a zero link.

3.7.4.6.1 Skipping List Instructions



Opcode String	Operation
LIST POP SKIP EMPTY	Remove an element from the head. (OP2,NEXT_OP2) is the list header. OP1 gets the address of the first element of the list. If the list is empty, then the instruction skips. if OP2 = 0 then PC←PC+4*SIGNED_SKP else begin OP1←OP2 OP2←M[OP2] end
LIST POP SKIP NOT EMPTY	Remove an element from the head. (OP2,NEXT_OP2) is the list header. OP1 gets the address of the first element of the list. If the list is not empty, then the instruction skips. if OP2 ≠ 0 then begin PC←PC+4*SIGNED_SKP OP1←OP2 OP2←M[OP2] end

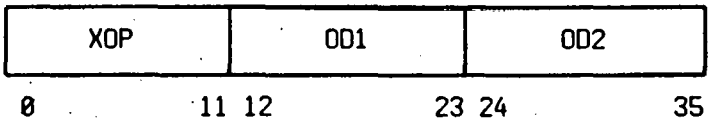
3.7.4.6.2 Non-Skipping List Instructions



Opcode String	Operation
LIST PUSH	Add an element to the head. (OP1,NEXT_OP1) is the list header. OP2 points to the element to be added to the head of the list. M[OP2]←OP1 if OP1 = 0 then NEXT_OP1←OP2 OP1←OP2
LIST APPEND	Add an element to the tail. (OP1,NEXT_OP1) is the list header. OP2 points to the element to be added to the tail of the list. M[OP2]←0 if OP1 = 0 then OP1←OP2 NEXT_OP1←OP2
LIST POP TRAP	Remove an element from the head. (OP2,NEXT_OP2) is the list header. OP1 gets the address of the first element of the list. If the list is empty, then the instruction soft traps to the trap vector at address LIST_UNDFL. if OP2 = 0 then soft_error(LIST_POP,PC) else begin OP1←OP2 OP2←M[OP2] end

3.7.4.7 Data Transfer

3.7.4.7.1 Block Transfer



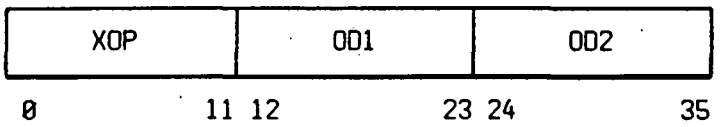
The block transfer (BLT) instruction transfers a block of data from one location in memory to another.

(OP2,NEXT_OP2) is the descriptor of the source block. This descriptor has double-word length; the first word is the address of the block, and the second word is the length of the block in quarter words. OP1 is the address of the destination block.

The operands of a BLT are continuously updated so that if an interrupt occurs during a BLT, the BLT can be restarted. It is therefore important that *the values of the operands not be used to calculate their own addresses.*

Opcode String	Operation
BLT {Q,H,S,D}	BLock Transfer. for I←0 step {1,2,3,4} until NEXT_OP2-{1,2,3,4} do M[OP1+I]←M[OP2+I] OP2←OP2+NEXT_OP2 OP1←OP1+NEXT_OP2 NEXT_OP2←0

3.7.4.7.2 Move and Exchange



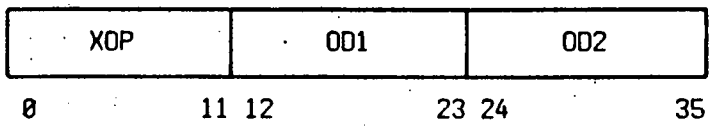
The "MOV" instructions move an operand of any integer length (Q, H, S, or D) to another operand of any integer length. The source and destination lengths are specified by including the appropriate characters together in the opcode string, with the destination length preceding the source length.

In addition, the "MOV" opcode strings may include special terms which specify the move type as shown in the opcode descriptions below. For example, "MOV N DS" means "negate a single precision integer and move it to a double precision integer."

EXCH assumes that the both operands are of the same precision.

Opcode String	Operation
MOV {Q,H,S,D}{Q,H,S,D}	OP1←OP2
MOV S {Q,H,S,D}{Q,H,S,D}	OP1←*sign_extend*(OP2)
MOV {1,2, ... ,8}	for I←1 step 1 until {1,2, ... ,8} do M[ADDRESS_OP1+I-1]← M[ADDRESS_OP2+I-1] (Note that MOV 1 and MOV 2 are the same as MOV S S and MOV D D.)
MOV C {Q,H,S,D}	OP1←*not*(OP2)
MOV N {Q,H,S,D}	OP1←*twos_negative*(OP2)
MOV M {Q,H,S,D}	OP1←*abs*(OP2)
MOV A	OP1←ADDRESS_OP2
MOV A OP1	OP1←(address specified by OD1 in the instruction at OP2)
MOV A OP2	OP1←(address specified by OD2 in the instruction at OP2)
MOV A REAL	OP1←*real_address*(ADDRESS_OP2)
EXCH {Q,H,S,D}	OP1↔OP2

3.7.4.8 Stack Manipulation



The stack manipulation instructions conditionally hard error trap on the result of the comparison of the stack pointer with the stack limit register. The trap location is a fixed location in virtual space, STACK_MANIP.

The "PUSH {UP,DOWN} TRAP" instructions push an operand of integer length (Q, H, S, or D) onto a stack and trap conditionally depending upon the outcome of a comparison. Stacks may grow either upward or downward; "PUSH UP" pushes onto an upward-growing stack and "PUSH DOWN" pushes onto a downward-growing stack. One operand, call it OP, is assumed to be a single-word stack pointer, and the stack limit is NEXT_OP. The length of the stack entry is specified by a term in the opcode string.

Opcode String				Operation
ADD		TRAP		if (OP1+OP2) > NEXT_OP1 then hard_error(STACK_ADJUST,ADDRESS_OP1) else OP1←OP1 + OP2
SUB		TRAP		if (OP1-OP2) < NEXT_OP1 then hard_error(STACK_ADJUST,ADDRESS_OP1) else OP1←OP1 - OP2
PUSH	UP	TRAP	{Q,H,S,D}	PUSH UP and TRAP if overflow if (OP1+{1,2,3,4}) > NEXT_OP1 then hard_error(STACK_ADJUST,ADDRESS_OP1) else begin M[OP1]←OP2 OP1←OP1+{1,2,3,4} end
PUSH	DOWN	TRAP	{Q,H,S,D}	PUSH DOWN and TRAP if overflow if (OP1-{1,2,3,4}) < NEXT_OP1 then hard_error(STACK_ADJUST,ADDRESS_OP1) else begin M[OP1]←OP2 OP1←OP1-{1,2,3,4} end
POP	UP		{Q,H,S,D}	POP an UPward stack. OP2←OP2-{1,2,3,4} OP1←M[OP2]
POP	DOWN		{Q,H,S,D}	POP a DOWNward stack. OP2←OP2+{1,2,3,4} OP1←M[OP2]

3.7.4.9 Subroutine Linkage

The subroutine linkage mechanism is designed to allow the efficient implementation of high-level block structured languages such as PASCAL; it explicitly implements call-by-value and call-by-reference.

In a block structured language, a display is often used to implement references to upper levels in the stack. The active display is maintained in the R registers; it consists of a pointer to the stack frame of each procedure which is at a lower lexical level than the currently active procedure. When a procedure at a lower lexical level returns, the display registers above the level of the called procedure must be restored to their state at the time of the call. For example, consider a procedure CALLER on lexical level 3 which calls a procedure CALLED on lexical level 1. CALLER first saves the old display register, DISPLAY[1], allocates a new frame on the stack, then sets DISPLAY[1] to point to the new frame. During the execution of CALLED, DISPLAY[2] and above are not needed, and therefore can be used for any other purpose, providing they are restored before CALLED exits. The per-procedure-call overhead in maintaining the display is then one memory write to save the old display register, one register write to set up the new display register, and one memory read to restore the old display register. During the execution of a procedure on lexical level I, I registers are required to hold its display; all registers above the level of the current display register can be used for local variables, providing they are restored on return.

In the LLL Filter, an efficient mechanism is provided for passing parameters to subroutines through the registers, rather than on the stack. The parameter instruction (PAR) is used to save a register on the stack, and to place a parameter in that register. This operation represents essentially the same overhead as pushing parameters on the stack, but has the advantage that it leaves the parameters in the registers for efficiency.

To understand the (PAR) instruction, it is first necessary to understand the format of the current stack frame. Before a procedure can be called, storage on the current stack frame must be allocated for the callee's parameters, the old stack frame pointer, and the return program counter, as shown in Figure 3.7.4.9-1. It will be convenient for the caller to allocate this extra space on its stack frame when it is first invoked, allowing enough room for the largest routine call which it will make. The allocation will thus be made far enough in advance so that pipeline interlocks normally will not occur (indexing off of a recently altered register will cause the pipeline to interlock). Furthermore, allocation in advance will save the expense of performing multiple allocations and deallocations, one pair for each call.

Figure 3.7.4.9-2 shows an example procedure call which passes three parameters A, B, and C, where A and C are call by value, and B is call by reference. Figure 3.7.4.9-3 shows the called procedure (CALLED), which uses two local registers and allocates 10 words on its stack. NEW_SF is the stack frame register for CALLED. The operations preformed by the subroutine linkage instructions are shown as comments in the example. The exact definition of the instructions is given in the sections which follow.

If the contents of a register used to pass a parameter are known to useless after the subroutine call, then the parameter can be MOVed to the register, and the register need not be restored, saving the overhead of one save and one restore.

This parameter passing method requires a register for each parameter passed to a procedure. One possible code-generation technique is to assign 8 registers to be used for passing parameters;

if a procedure has more than 8 parameters, it will push the rest of the parameters onto the stack. Furthermore, it will be efficient to have two types of temporary registers for use in procedures; the first type will be used to hold local variables, which are saved and restored when a procedure is entered and exited, and the other type will never be saved, but will be used for holding temporary results and calling bottom-level procedures (which call no other procedures).

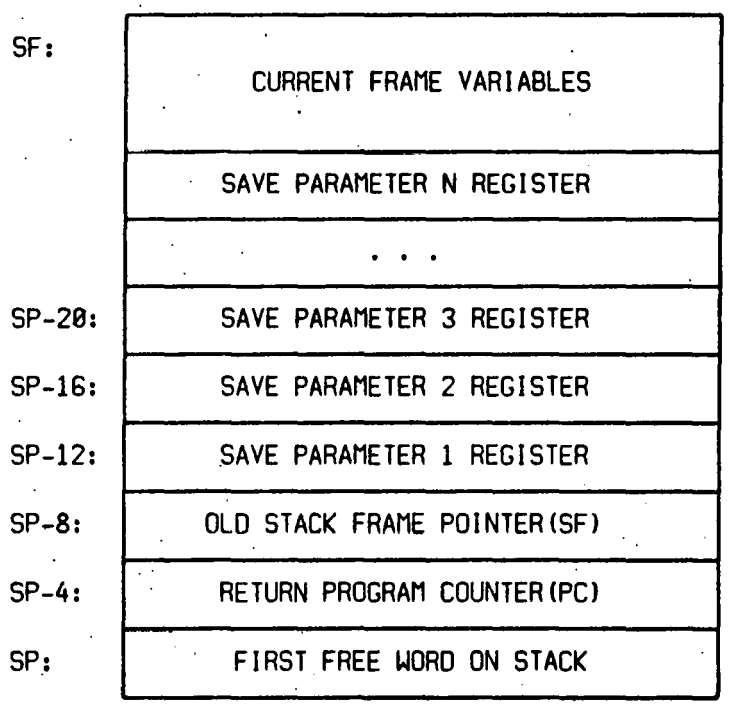


Figure 3.7.4.9-1
Current Stack Frame

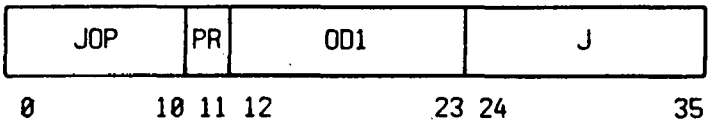
```
        PAR 1           P_REG,A         !M[SP-12]←R[P_REG]
                                        !R[P_REG]←M[A]

        PAR A 2         P_REG-1,B       !M[SP-16]←R[P_REG-1]
                                        !R[P_REG-1]←B

        PAR 3           P_REG-2,C       !M[SP-20]←R[P_REG-2]
                                        !R[P_REG-2]←M[C]

        JUMP SUB        NEW_SF,CALLED   !M[SP-8]←R[NEW_SF]
                                        !M[SP-4]←PC+4
                                        !R[NEW_SF]←SP
                                        !PC←CALLED

        MOV 3           P_REG-2,-20(SP) !R[P_REG-2]←M[R[SP]-20]
                                        !R[P_REG-1]←M[R[SP]-16]
                                        !R[P_REG]←M[R[SP]-12]
```

Figure 3.7.4.9-2
Example Procedure Call

```
CALLED: ALLOC 2         NEW_SF+1,#40    !M[SP]←R[NEW_SF+1]
                                        !M[SP+4]←R[NEW_SF+2]
                                        !SP←SP+40

        !"ROUTINE BODY"

        RETURN SUB 2    NEW_SF+1,NEW_SF !R[NEW_SF+1]←M[R[NEW_SF]]
                                        !R[NEW_SF+2]←M[R[NEW_SF]+4]

                                        !PC←M[R[NEW_SF]-4]
                                        !SP←R[NEW_SF]
                                        !R[NEW_SF]←M[R[NEW_SF]-8]
```

Figure 3.7.4.9-3
Example of Called Procedure

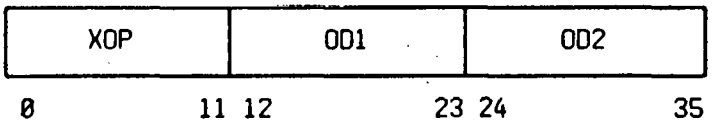
3.7.4.9.1 Jump to Subroutine



OP1 is the stack frame register. The JUMP SUB instruction saves on the stack the return program counter and the old stack frame register (OP1), and sets the new stack frame register (OP1) equal to the stack pointer.

Opcode String	Operation
JUMP SUB	JUMP to SUBroutine M[SP-8]←OP1 M[SP-4]←PC_NEXT_INSTR OP1←SP PC←JUMPDEST

3.7.4.9.2 Subroutine Context Switching



PAR saves the value of a register (OP1) in one of eight parameter-save areas on the current stack frame, and loads OP1 with a value parameter, OP2. PAR A is identical except it loads OP1 with the address of OP2.

ALLOCATE is used by the called procedure to allocate OP2 words on the stack, and to save 1 to 8 registers (sequentially, starting with OP1) at the beginning of the new stack frame.

RETURN SUB restores 1 to 8 registers (sequentially, starting with OP1) from the beginning of the current stack frame, restores the PC from the previous stack frame, sets the SP to the value of the current stack frame pointer (OP2), and restores the previous stack frame pointer from the previous stack frame.

Opcode String		Operation
PAR	{1,2, ... ,8}	subroutine PARameter M[SP-8-{1,2, ... ,8}*4]←OP1 OP1←OP2
PAR A	{1,2, ... ,8}	subroutine PARameter Address M[SP-8-{1,2, ... ,8}*4]←OP1 OP1←ADDRESS_OP2
ALLOCATE	{1,2, ... ,8}	ALLOCATE stack and save registers if SP > (SL+OP2*4) then hard_error(STACK_ADJUST,SP_ID*4) else begin for I←1 step 1 until {1,2, ... ,8} do M[SP+I*4-4]← M[ADDRESS_OP1+I*4-4] SP←SP+OP2 end
RETURN SUB	{0,1,2, ... ,8}	RETURN from SUBroutine and restore registers. for I←1 step 1 until {0,1,2, ... ,8} do M[ADDRESS_OP1+I*4-4]← M[OP2+I*4-4] PC←M[OP2-4] SP←OP2 OP2←M[OP2-8]

3.7.4.10 Traps and Interrupts

This section describes trap instructions, soft-error traps, hard-error traps, and interrupts.

Traps and interrupts use *trap vectors*. A trap vector includes a new PC and possibly a status word; those values are loaded into the processor during a trap after the previous state of the machine has been saved.

The trap instructions allow trapping within the current mode (TRAP SELF), or trapping to the executive (TRAP EXEC). TRAP SELF does not save the status register, but places the addresses of OP1 and OP2 into R[30] and R[31] (after saving them); it is intended to be used as a two-parameter subroutine call. TRAP EXEC saves the status register and gets a new status register from the trap vector; it also places the addresses of OP1 and OP2 in R[30] and R[31], but without saving those registers. TRAP EXEC is intended to be used to implement monitor calls; the executive will reserve R[30] and R[31] to receive parameters. The TRAP opcodes define the trap vector addresses; each instruction type has 64 different opcodes, each of which traps to a unique trap vector. The TRAP SELF trap vectors are contiguous in both the user and executive virtual address spaces, starting at address TRAP_SELF_ADR, and the TRAP EXEC trap vectors are contiguous in the executive address space starting at address TRAP_EXEC_ADR (they do not exist in the user address space). Both TRAP USER and TRAP EXEC save the PC of the *next instruction* (some types of traps save the PC of the current instruction); a return will thus not re-execute the trap instruction.

Some types of instruction execution errors (for example, integer overflow) will cause a soft error trap. A soft error traps to a fixed trap vector address (which depends upon the identity of the error) in the current address space. A soft error trap saves the USER_STATUS_REGISTER (and sets a new USER_STATUS_REGISTER from the trap vector), if the trap occurs in user mode, but saves the STATUS_REGISTER (and sets a new STATUS_REGISTER from the trap vector), if the trap occurs in executive mode. The soft error trap routine also saves on the stack the PC of the next instruction and one or more parameters, the nature of which is specific to the type of error. Returns from soft error traps will usually be to the next instruction, since most instructions with soft errors complete execution before trapping. Cases in which the trapping instruction needs to be re-executed are handled by passing the PC of the trapping instruction as a parameter.

Other types of instruction execution errors (for example, writing a read-only page) will cause a hard error trap. A hard error traps to a fixed trap vector address (which depends upon the identity of the error) in the *executive* address space. Hard errors occurring in the executive trap to different locations than hard errors occurring in the user. A hard error trap saves one or more parameters, the PC of the trapping instruction, and the STATUS_REG; the save area is simply the stack defined by the new STATUS_REG, which is obtained from the trap vector. The STATUS_REG value in the trap vector will also set the processor into executive mode. As with soft errors, the nature and number of the parameters saved is specific to the type of error. Most hard errors cause abortion of an instruction before any results are written; those instructions can be re-executed.

Two special hard errors may occur during traps or interrupts: page fault, and stack overflow. These errors trap again to special hard error trap vectors, passing parameters which allow the proper execution and return of trap which encountered the error. The special hard error handler PAGE_FAULT_IN_TRAP must not encounter a page fault error, and the hard error handler SP_OVFL must not encounter a stack overflow error.

An interrupt is similar to a hard error, but no parameter is saved. An interrupt is initiated when one of the four interrupt lines is asserted; if the priority of the interrupt is higher than PRIO, then the interrupt is accepted and the processor, under micro-code control, finds an interrupt vector address (INT.VECTOR) in main memory (where it was stored by the interrupting device). The processor at the same time resets the interrupt bit which caused the interrupt line to be asserted. Interrupts are tested immediately before execution of an instruction; at that time PC is the address of the next instruction to be executed.

Three return instructions handle all returns from traps or interrupts; RETURN REGS, RETURN USER STATUS, and RETURN FULL STATUS restore only registers, only the user status, and the full status, respectively. RETURN REGS handles returns from TRAP SELF, RETURN USER STATUS handles returns from soft error traps and RETURN FULL STATUS handles returns from hard error traps, TRAP EXEC, and interrupts. Both RETURN USER STATUS and RETURN FULL STATUS allow OP1 to specify the number of locations to be popped off of the stack.

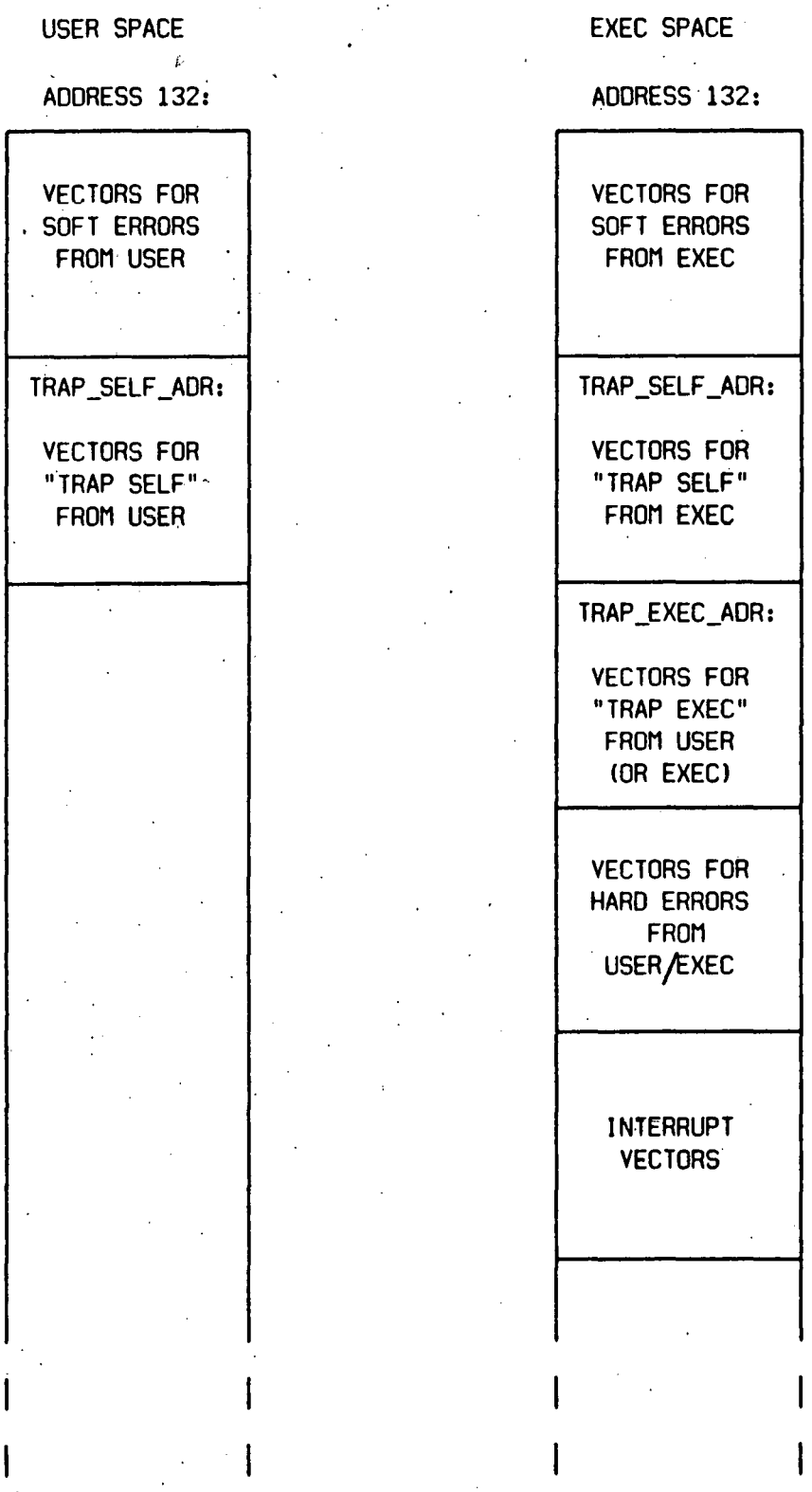


Figure 3.7.4.10-1
User and Executive Address Spaces

Vector for TRAP SELF from user:

HANDLER ADDRESS

Vector for TRAP SELF from executive:

HANDLER ADDRESS

Vector for soft error from user:

HANDLER ADDRESS
NEW USER_STATUS_REG

Vector for soft error from executive :

HANDLER ADDRESS
NEW STATUS_REG

Vector for hard error from user or executive:

HANDLER ADDRESS FOR USER HARD ERROR
NEW STATUS_REG FOR USER HARD ERROR
HANDLER ADDRESS FOR EXEC HARD ERROR
NEW STATUS_REG FOR EXEC HARD ERROR

Vector for interrupt:

HANDLER ADDRESS
NEW STATUS_REG

Figure 3.7.4.10-2
Trap Vector Formats

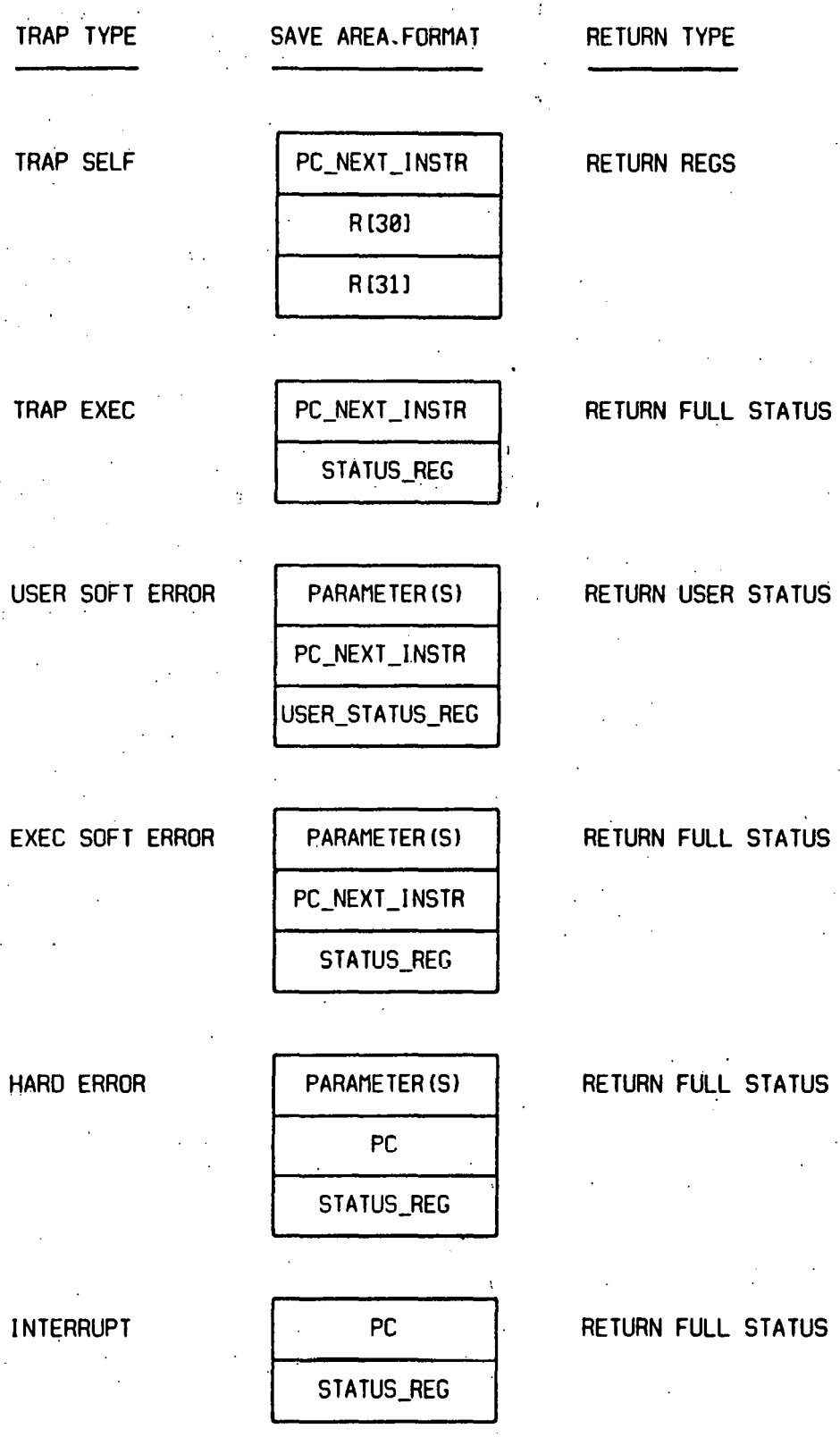


Figure 3.7.4.10-3
Save Area Formats

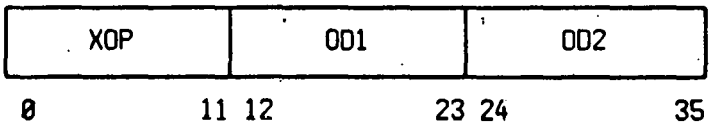
Trap Address	Error Condition	Parameters
INT_OVFL	integer overflow	PC
ZERO_DIVIDE	divide by zero	PC
LIST_UNDFL	list underflow	PC
FLOAT_UNDFL	floating underflow	PC
FLOAT_OVFL	floating overflow	PC
POST_OVFL	postnormalization overflow	PC
PRE_OVFL	prenormalization overflow	PC

Figure 3.7.4.10-4
Soft Error Trap Addresses

Trap Address	Error Condition	Parameters
TRACE	trace trap	PC
PAGE_FAULT_IN_TRAP	page fault during trap	page address trap address trap parameter
SP_OVFL	SP overflow in trap	trap address
PAGE_FAULT	page fault	page address
STACK_ADJUST	stack overflow	stack register adr
EXECUTE_USER	execute to user space from exec	PC
JUMP_USER	jump to user space from exec	PC
REF_EXEC	reference to exec space from user	PC
STATUS ACCESS	accessing processor status by user	PC
ILLEGAL_INSTR	illegal instruction	PC
NOT_INSTRUCTION	page at PC is not instruction type	PC
NOT_DATA	operand page is not data type	PC
WRITE_ONLY	reading a write-only page	PC
READ_ONLY	writing a read-only page	PC
BOUNDARY_ERROR	data/instruction boundary error	PC

Figure 3.7.4.10-5
Hard Error Trap Addresses

3.7.4.10.1 Trap Instructions



Opcode String		Operation
TRAP SELF	{0,1,2, ... ,63}	M[SP]←PC_NEXT_INSTR M[SP+4]←R[30] M[SP+8]←R[31] R[30]←ADDRESS_OP1 R[31]←ADDRESS_OP2 SP←SP+12 PC←M[TRAP_SELF_ADR+{0,1,2, ... ,63}*4] if SP > SL then SP_ovfl(SP_ID*4)
TRAP EXEC	{0,1,2, ... ,63}	TEMP[1]←STATUS_REG EXEC_MODE←1 STATUS_REG← M[TRAP_EXEC_ADR+{0,1,2, ... ,63}*8+4] M[SP]←PC_NEXT_INSTR M[SP+4]←TEMP[1] R[30]←ADDRESS_OP1 R[31]←ADDRESS_OP2 PC←M[TRAP_EXEC_ADR+{0,1,2, ... ,63}*8] SP←SP+8 if SP > SL then SP_ovfl(SP_ID*4)

3.7.4.10.2 Soft-Error Trap

Opcode String	Operation

soft_error(TRAP_ADR,PAR)	if page fault
	  in (M[SP],M[SP+4],M[SP+8])
	then page_fault_in_trap(
	  TRAP_ADR,PAR)
	M[SP]←PAR
	M[SP+4]←PC_NEXT_INSTR
	if EXEC_MODE
	then M[SP+8]←STATUS_REG
	else M[SP+8]←USER_STATUS_REG
	PC←M[TRAP_ADR]
	SP←SP+12
	if SP > SL
	then SP_ovfl(SP_ID*4)

3.7.4.10.3 Hard-Error Traps

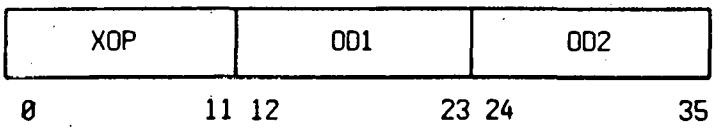
Opcode String	Operation
hard_error(TRAP_ADR,PAR)	TEMP[1]←STATUS_REG if EXEC_MODE then begin STATUS_REG←M[TRAP_ADR+12] M[SP+4]←PC PC←M[TRAP_ADR+8] end else begin EXEC_MODE←1 STATUS_REG←M[TRAP_ADR+4] M[SP+4]←PC PC←M[TRAP_ADR] end M[SP]←PAR M[SP+8]←TEMP[1] SP←SP+12 if SP > SL then SP_ovfl(SP_ID*4)
page_fault_in_trap(TRAP_ADR,PAR)	TEMP[1]←STATUS_REG if EXEC_MODE then begin STATUS_REG←M[SOFT_ERROR_PAGE_FAULT+12] M[SP+8]←PC_NEXT_INSTR PC←M[PAGE_FAULT_IN_TRAP+8] end else begin EXEC_MODE←1 STATUS_REG←M[SOFT_ERROR_PAGE_FAULT+4] M[SP+8]←PC_NEXT_INSTR PC←M[PAGE_FAULT_IN_TRAP] end M[SP]←TRAP_ADR M[SP+4]←PAR M[SP+12]←TEMP[1] SP←SP+16 if SP > SL then SP_ovfl(SP_ID*4)

Opcode String	Operation
SP_ovfl(PAR)	TEMP[1]←STATUS_REG if EXEC_MODE then begin STATUS_REG←M[STACK_OVFL+12] M[SP+4]←PC PC←M[STACK_OVFL+8] end else begin EXEC_MODE←1 STATUS_REG←M[STACK_OVFL+4] M[SP+4]←PC PC←M[STACK_OVFL] end M[SP]←PAR M[SP+8]←TEMP[1] SP←SP+12

3.7.4.10.4 Interrupt

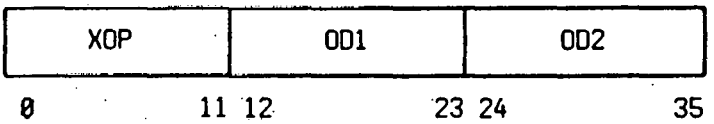
Opcode String	Operation
interrupt(INT_VECTOR)	TEMP[1]←STATUS_REG EXEC_MODE←1 STATUS_REG←M[INT_VECTOR+4] M[SP]←PC M[SP+4]←TEMP[1] SP←SP+8 PC←M[INT_VECTOR] if SP > SL then SP_ovfl(SP_ID*4)

3.7.4.10.5 Trap and Interrupt Returns



Opcode String	Operation
RETURN REGS	Return and restore registers. (Return from TRAP SOFT.) PC←M[SP-12] R[30]←M[SP-8] R[31]←M[SP-4] SP←SP-12
RETURN FULL STATUS	Return and restore full status. (Return from interrupt, hard error, or TRAP EXEC.) PC←M[SP-8] STATUS_REG←M[SP-4] SP←SP-OP1
RETURN USER STATUS	Return and restore user status. (Return from soft error.) PC←M[SP-8] USER_STATUS_REG←M[SP-4] SP←SP-OP1

3.7.4.11 Cache Control



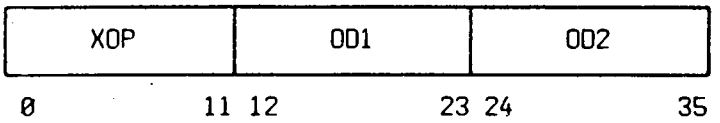
The cache control instructions have been described in Section 3.1. If a very large sweep range is specified in a cache control instruction, the processor will choose to sweep the entire cache instead of sweeping each location in the range.

For efficiency reasons, a special instruction is provided to sweep both the instruction cache and the data cache simultaneously.

Opcode String	Operation
UPDATE DATA	Sweep through the data cache (for OP2 quarter-words), starting at virtual address OP1, and writing back changed locations.
KILL DATA	Same as UPDATE DATA, except that the words in the cache in the given range are also invalidated, so that future references to them will be made to memory.
KILL INSTR	Sweep through the instruction cache (for OP2 quarter-words), invalidating each location starting at virtual address OP1.
KILL DATA INSTR	Same as KILL DATA followed by KILL INSTR.

3.7.4.12 Page Map Control

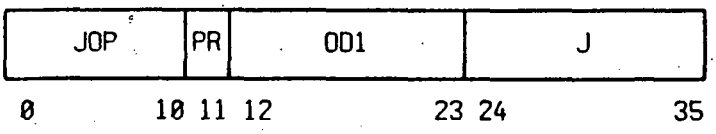
3.7.4.12.1 KILL MAP



The page map control instructions have been described in Section 3.2. KILL MAP deletes a specific entry from both page maps. KILL MAP EXEC deletes all executive address space entries in the page map, and KILL MAP USER deletes all user address space entries in the page map.

Opcode String	Operation
KILL EXEC MAP	Invalidate the entry in the associative map that corresponds to the executive virtual address M[OP1].
KILL USER MAP	Invalidate the entry in the associative map that corresponds to the user virtual address M[OP1].
KILL ALL EXEC MAP	Invalidate all executive address space entries in the page map.
KILL ALL USER MAP	Invalidate all user address space entries in the page map.

3.7.4.12.2 Writing Segment Base Registers

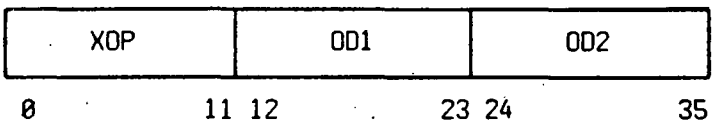


These instructions allow writing either segment base register. A jump is included to allow writing the executive to write its own segment base register (which affects the instruction address space for the executive). Execution of WRITE EXEC JUMP will cause all executive address space entries to be deleted from the page map. Execution of WRITE USER JUMP will cause all user address space entries to be deleted from the page map.

Opcode String	Operation
WRITE EXEC JUMP	EXEC_SEG_BASE_REG←OP1 PC←JUMPDEST
WRITE USER JUMP	USER_SEG_BASE_REG←OP1 PC←JUMPDEST

3.7.4.13 Status Register Control

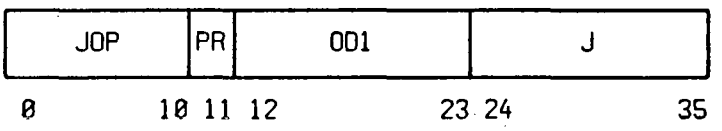
3.7.4.13.1 Read Status



The full processor status and the processor ID are accessible only in executive mode.

Opcode String	Operation
READ FULL STATUS	OP1←STATUS_REG
READ USER STATUS	OP1←USER_STATUS_REG
READ PROC ID	OP1←PROCESSOR_ID

3.7.4.13.2 Write Status

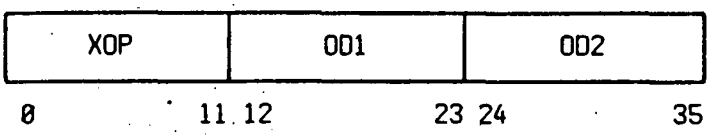


The processor status register is accessible only in executive mode. A jump is provided after the load so that the executive can load a user's status register and jump to the user in one instruction. The M bit cannot be set in the jump destination of these or any other jump instructions.

Opcode String	Operation
WRITE FULL STATUS JUMP	STATUS_REG←OP1 PC←JUMPDEST
WRITE USER STATUS JUMP	USER_STATUS_REG←OP1 PC←JUMPDEST

3.7.4.14 Synchronization

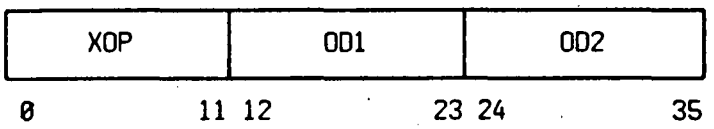
3.7.4.14.1 SET INTERRUPT



Interrupts have been described in Section 3.4.1. A processor P_i may direct an interrupt to processor P_j by setting bit i in P_j's interprocessor interrupt word using a read-modify-write memory cycle. OP1 and OP2 are assumed to be single-word operands.

Opcode String	Operation
SET INTERRUPT	(using read-modify-write cycle) OP1←OP1∨OP2
RESET INTERRUPT	(using read-modify-write cycle) OP1←OP1∧*not*(OP2)

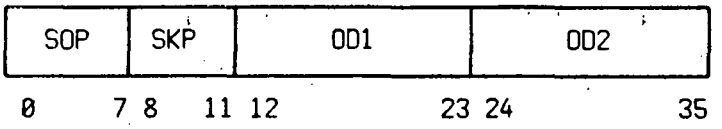
3.7.4.14.2 Test and Set/Reset



TEST AND SET and TEST AND RESET allow the setting and resetting of single-word flags using a read-modify-write memory cycle.

Opcode String	Operation
TEST AND SET	(using read-modify-write cycle) OP1←OP2 OP2←-1
TEST AND RESET	(using read-modify-write cycle) OP1←OP2 OP2←0

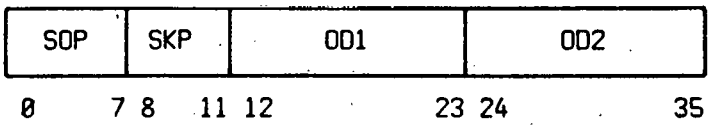
3.7.4.14.3 Munch Registers



Munch registers have been described in Section 3.4.3. These instructions allow a munch register to be set if and only if there is no conflict (that is, no other munch register equals OP2). If a conflict exists, the munch register controller writes a zero into the munch register. The instruction definitions assume that OP1 is a munch register.

Opcode String	Operation
MUNCH SKIP OK	if *no_conflict* then begin OP1←OP2 PC←PC+SIGNED_SKP end else OP1←0
MUNCH SKIP NOT OK	if *no_conflict* then OP1←OP2 else begin PC←PC+SIGNED_SKP OP1←0 end

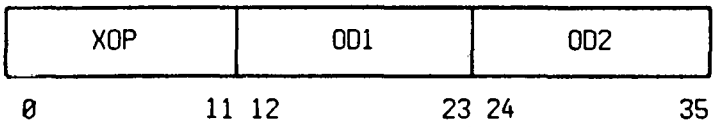
3.7.4.14.4 Hardware Queues



This instructions have been described in Section 3.4.4. The definitions assume that QUEUE.X is a hardware queue at location ADDRESS_X. The processor uses a read-modify-write memory cycle to both determine whether the queue is full (empty) and to enqueue (dequeue) an entry if and only if such enqueueing (dequeueing) is possible. Both LIFO and FIFO queues are provided; they are distinguished by their addresses.

Opcode String	Operation
QUEUE SKIP FULL	(using read-modify-write cycle) if *not_full* then QUEUE.OP1←OP2 else PC←PC+SIGNED_SKP
QUEUE SKIP NOT FULL	(using read-modify-write cycle) if *not_full* then begin QUEUE.OP1←OP2 PC←PC+SIGNED_SKP end
DEQUEUE SKIP EMPTY	(using read-modify-write cycle) if *not_empty* then OP1←QUEUE.OP2 else PC←PC+SIGNED_SKP
DEQUEUE SKIP NOT EMPTY	(using read-modify-write cycle) if *not_empty* then begin OP1←QUEUE.OP2 PC←PC+SIGNED_SKP end

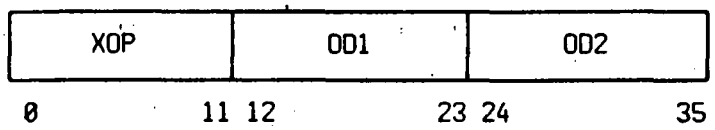
3.7.4.15 Control Store



When the processor is powered-up, an LSI-11 console machine initializes the control memories in the processor. The following instructions allow the *operating system* to alter the control memories.

Opcode String	Operation
WRITE ISEQ	Word OP1 in the ISEQ control gets OP2.
WRITE PSEQ	Word OP1 in the PSEQ control gets OP2.
WRITE ESEQ	Word OP1 in the ESEQ control gets OP2.
WRITE DECODE RAM	Word OP1 in the DECODE RAM gets OP2.
WRITE DATA CACHE LRU	Word OP1 in the DATA CACHE LRU DECODE RAM gets OP2.
WRITE INSTR CACHE LRU	Word OP1 in the INSTR CACHE LRU DECODE RAM gets OP2.
WRITE DATA ADR TRN LRU	Word OP1 in the DATA ADDRESS TRANSLATION LRU DECODE RAM gets OP2.
WRITE INSTR ADR TRN LRU	Word OP1 in the INSTR ADDRESS TRANSLATION LRU DECODE RAM gets OP2.

3.7.4.16 Miscellaneous



Opcode String	Operation
WAIT	Wait for interrupt.
HALT	Stop processor OP1.
START	Start processor OP1, if halted, else does nothing.
RESET	Reset I/O devices and switch.
EXECUTE	Execute OP1 in the address space of OP1.
AMPUTATE	Lock processor OP1 out of the switch.

3.7.5 Sample Programs

This section presents sample programs which for comparison are coded in several assembly languages, including assembly language for the LLL Filter.

The purpose of this section is to indicate the density of compiled code for the LLL Filter, to suggest the relative execution speed of the LLL Filter compared with existing machines, and to clarify the LLL Filter instruction set.

3.7.5.1 Assembly Language Specification

This section presents a brief, informal description of the assembly language which is used for the sample programs included in this report.

An assembly language statement may have five main fields, as follows:

LABEL OPCODE GOTO OPERANDS COMMENTS

The LABEL and COMMENTS fields are self-explanatory. The remaining fields are described in the following sections.

3.7.5.1.1 OPCODE Field

The OPCODE field contains an opcode string, as described in Section 3.7.4, or an abbreviated form of the opcode string. An opcode string may be abbreviated by the deletion of certain terms; the assembler fills in default values for these terms. The following list shows the assembler defaults for opcode string terms:

Term	Assembler Default
{S,D}	S
{FR,CR,SR}	SR

For example, the assembler expands the opcode string "FDIV" into "FDIV SR S", meaning "single-word floating divide with stable rounding."

3.7.5.1.2 GOTO Field

The GOTO field is used for any instruction which includes a skip or a jump destination. The GOTO field contains the name of the destination instruction.

3.7.5.1.3 OPERANDS Field

The OPERANDS field specifies the operands of the instruction. The operand names RTA, RTB, PC, SP, and SL are reserved words which indicate special R registers, as shown in Section 3.7.2. The notation RX means R[X].

Operands are written in the order shown in Table 3.7.3.2-1. In instructions having two operands, the order of the operands is OP1, OP2. In instructions having three operands, the operands are written "DEST,OP1,OP2."

3.7.5.2 Use of the T Field

The main use of the T format instructions is in the evaluation of expressions. The following examples compare LLL Filter code and PDP-10 code in the evaluation of expressions.

Expression	LLL Filter	# Words	PDP-10	# Words
A←A+B		1		2
	ADD A,B		MOVE R0,B ADDM R0,A	
A←B+C		2		3

	ADD RTA,B,C
	MOV A,RTA
			MOVE R0,B
			ADD R0,C
			MOVEM R0,A	

A←B+C-D		2		4

	ADD RTA,B,C
	SUB A,RTA,D
			MOVE R0,B
			ADD R0,C
			SUB R0,D
			MOVEM R0,A	

A←A*B+C*D		3		6

	MULT RTA,A,B
	MULT RTB,C,D
	ADD A,RTA,RTB
			MOVE R0,A
			MULT R0,B
			MOVE R1,C
			MULT R1,D
			ADD R0,R1
			MOVEM R0,A	

A←B*(C(J)-D(K))		4		6

	SUB RTA,C(J),D(K)
	MULT A,RTA,B
			MOVE R0,J
			MOVE R1,K
			MOVE R2,C(R0)
			SUB R2,D(R1)
			MULT R2,B
			MOVEM R2,A	

A←B(I+J)*C(K+L)+D(M+N)*E(L+P)		12		14

	ADD RTA,I,J
	ADD RTB,K,L
	MULT RTA,B(RTA),C(RTB)
	MOV R1,RTA
	ADD RTA,M,N
	ADD RTB,L,P
	MULT RTA,D(RTA),E(RTB)
	ADD A,RTA,R1
			MOVE R0,I
			ADD R0,J
			MOVE R1,K
			ADD R1,L
			MOVE R2,B(R0)
			MULT R2,C(R1)
			MOVE R0,M
			ADD R0,N
			MOVE R1,L
			ADD R1,P
			MOVE R3,D(R0)
			MULT R3,E(R1)
			ADD R2,R3
			MOVEM R2,A	

This last example might seem a little unlikely, but it was given because except for the statement "A←B+C", it is the only expression that can not be evaluated with no "MOV" instructions, because each of the four subscripts need the RT registers, and each of the products need the RT registers for their results. If even one of the four subscripts takes one more or one less operation, then the expression can be evaluated with no "MOV" instructions.

3.7.5.3 Compiled Treesort Comparisons

This section compares compilations of the Treesort algorithm. The first compilation shown is the output of a hypothetical simple compiler compiling BLISS for the LLL Filter. The second compilation is the output of the BLISS-10 compiler compiling BLISS for the PDP-10. The third compilation is the output of the BLISS-11 compiler compiling BLISS for the PDP-11. Each of the first three compilations is shown for two cases, called case NO REGS and case REGS, which correspond to the cases in which the variables T, J, K, and N are declared to be OWN variables and REGISTER variables, respectively. The last compilation is the output of the FORTRAN-H compiler compiling a FORTRAN version of the same algorithm for the IBM-370/168. This compilation was performed using the full optimization capability of FORTRAN-H (OPT=2).

The following table summarizes the important static parameters of the compilations.

	# INSTRUCTIONS	# BITS	DATA CACHE CYCLES
LLL Filter (NO REGS)	33	1584	81
LLL Filter (REGS)	33	1584	19
BLISS-10 (NO REGS)	63	2268	60
BLISS-10 (REGS)	42	1512	19
BLISS-11 (NO REGS)	63	1376	63
BLISS-11 (REGS)	58	1216	31
FORTRAN-H 370/168	84	2432	51

3.7.5.3.1 BLISS Treesort Algorithm

This section presents the Treesort algorithm which is compiled for several machines in the following sections. The listing shown declares T, J, K, and N to be registers.

```
MODULE=
BEGIN

REGISTER T,J,K,N;
LABEL L1,L2;
OWN  A[61];

INCR I FROM 2 TO .N DO BEGIN
   K←.I;
   J←.I;
   T←.A[.I];
   L1: DO BEGIN
      J←.J/2;
      IF .T LEQ .A[.J] THEN LEAVE L1;
      A[.K]←.A[.J];
      K←.J;
   END UNTIL .J EQL 1;
   A[.K]←.T;
END;

DECR I FROM .N-1 TO 1 DO BEGIN
   T←.A[.I+1];
   A[.I+1]←.A[1];
   K←1;
   J←2;
   L2: WHILE .J LEQ .I DO BEGIN
      IF .J LSS .I THEN BEGIN
         IF (.A[.J+1] GTR .A[.J]) THEN J←.J+1;
      END;
      IF .A[.J] GTR .T THEN BEGIN
         A[.K]←.A[.J];
         K←.J;
         J←2*.J;
      END ELSE LEAVE L2;
   END;
   A[.K]←.T;
END;

END ELUDOM;
```

3.7.5.3.2 LLL Filter Compilation

This section presents the output of a hypothetical non-optimizing compiler compiling the above BLISS program for the LLL Filter. Along with each assembly language instruction is shown the number of data cache cycles required for the instruction for each of case NO REGS and case REGS, and the length of the instruction in words.

The assembly language output is identical for case NO REGS and case REGS, therefore only one listing is shown.

				NO REGS # DATA CACHE CYCLES	REGS #DATA CACHE CYCLES	# 36-BIT INSTR WORDS
	MOV		I,#2	2	0	1
	SKIP LE	L1	I,N	2	0	1
	JUMP	L4		0	0	1
L1	MOV		K,I	3	0	1
	MOV		J,I	3	0	1
	MOV		T,A(I)	4	1	2
L2	SHIFT RIGHT A		J,#1	2	0	1
	SKIP LE	L3	T,A(J)	3	1	2
	MOV		A(K),A(J)	5	3	2
	MOV		K,J	3	0	1
	SKIP NE	L2	J,#1	1	0	1
L3	MOV		A(K),T	4	2	2
	INC SKIP G	L4	I,N	3	0	1
	JUMP	L1		0	0	1
L4	DEC		I,N	3	0	1
	JUMP LE 0	L11	I	1	0	1
L5	MOV		T,A+1(I)	4	1	2
	MOV		A+1(I),A+1	4	3	2
	MOV		K,#1	2	0	1
	MOV		J,#2	2	0	1
L6	SKIP LE	L7	J,I	2	0	1
	JUMP	L10		0	0	1
L7	SKIP GE	L8	J,I	2	0	1
	SKIP LE	L8	A+1(J),A(J)	3	2	3
	ADD		J,#1	2	0	1
L8	SKIP LE	L10	A(J),T	3	1	2
	MOV		A(K),A(J)	5	3	2
	MOV		K,J	3	0	1
	SHIFT LEFT A		J,#1	2	0	1
	SKIP G	L10	J,I	2	0	1
	JUMP	L7		0	0	1
L10	MOV		A(K),T	4	2	2
	INC JUMP G 0	L5	I	2	0	1
L11						
			TOTAL:	81	19	44

3.7.5.3.3 BLISS-10 Compilation for PDP-10

Following is the code generated by the BLISS-10 compiler compiling the above BLISS program for the PDP-10 for the case in which T, J, K, and N are not declared to be registers.

```
        MOVEI   17,2
        MOVEM   17,I
L1      CAMLE   17,N
        JRST    L4
        MOVEM   17,K
        MOVEM   17,J
        MOVE    4,A(17)
        MOVEM   4,T
L2      MOVE    5,J
        ASH     5,-1
        MOVEM   5,J
        MOVE    6,A(5)
        CAMLE   6,T
        JRST    L3
        MOVE    10,K
        MOVE    11,J
        MOVE    12,A(11)
        MOVEM   12,A(10)
        MOVEM   11,K
        CAIE    11,1
        JRST    L2
L3      MOVE    4,K
        MOVE    5,T
        MOVEM   5,A(4)
        AOJA    17,L1
L4      MOVE    6,N
        SOJ     6,0
        MOVE    17,6
        JUMPLE  17,L9
L5      MOVE    7,A+1(17)
        MOVEM   7,T
        MOVE    10,A+1
        MOVEM   10,A+1(17)
        MOVEI   12,1
        MOVEM   12,K
        MOVEI   11,2
        MOVEM   11,J
L6      CAMGE   17,J
        JRST    L8
        CAMG    17,J
        JRST    L7
        MOVE    5,J
        MOVE    6,J
        MOVE    7,A(6)
        CAML    7,A+1(5)
        JRST    L7
        AOJ     5,0
        MOVEM   5,J
L7      MOVE    5,J
        MOVE    6,T
        CAML    6,A(5)
        JRST    L8
        MOVE    10,K
        MOVE    12,A(5)
        MOVEM   12,A(10)
        MOVEM   5,K
        ASH     5,1
        MOVEM   5,J
        JRST    L6
L8      MOVE    10,K
        MOVE    12,T
        MOVEM   12,A(10)
        SOJG    17,L5
L9      .
```

Following is the code generated by the BLISS-10 compiler compiling the above BLISS program for the PDP-10 for the case in which T, J, K, and N are declared to be registers.

```
        MOVEI   13,2
L1      CAMLE   13,14
        JRST    L2
        MOVE    15,13
        MOVE    16,13
        MOVE    17,A(13)
L3      ASH     16,-1
        CAMG    17,A(16)
        JRST    L4
        MOVE    4,A(16)
        MOVEM   4,A(15)
        MOVE    15,16
        CAIE    16,1
        JRST    L3
L4      MOVEM   17,A(15)
        AOJA    13,L1
L2      MOVE    13,14
        SOJ     13,0
        JUMPLE  13,L5
L6      MOVE    17,A+1(13)
        MOVE    6,A+1
        MOVEM   6,A+1(13)
        MOVEI   15,1
        MOVEI   16,2
L7      CAMLE   16,13
        JRST    L10
        CAML    16,13
        JRST    L11
        MOVE    11,A(16)
        CAMGE   11,A+1(16)
        AOJ     16,0
L11     CAML    17,A(16)
        JRST    L10
        MOVE    12,A(16)
        MOVEM   12,A(15)
        MOVE    15,16
        MOVE    1,16
        ASH     1,1
        MOVE    16,1
        JRST    L7
L10     MOVEM   17,A(15)
        SOJG    13,L6
```

3.7.5.3.4 BLISS-11 Compilation for PDP-11

Following is the code generated by the BLISS-11 compiler compiling the above BLISS program for the PDP-11 for the case in which T, J, K, and N are not declared to be registers.

```
        MOV     #T,R$0
        MOV     #J,R$3
        MOV     #K,R$1
        MOV     #2,-(SP)
        BR      L$6
L$5:    MOV     @SP,@R$1
        MOV     @SP,@R$3
        MOV     @SP,R$5
        ASL     R$5
        MOV     A(R$5),@R$0
L$7:    ASR     @R$3
        MOV     @R$3,R$2
        MOV     R$2,R$5
        ASL     R$5
        CMP     A(R$5),@R$0
        BGE     L1
        MOV     @R$1,R$4
        ASL     R$4
        MOV     A(R$5),A(R$4)
        MOV     R$2,@R$1
        CMP     R$2,#1
        BNE     L$7
L1:     MOV     @R$1,R$4
        ASL     R$4
        MOV     @R$0,A(R$4)
        INC     @SP
L$6:    CMP     @SP,@#N
        BLE     L$5
        MOV     @#N,R$5
        DEC     R$5
        MOV     R$5,R$2
        BR      L$13
L$12:   MOV     R$2,R$5
        ASL     R$5
        MOV     A+2(R$5),@R$0
        MOV     @#A+2,A+2(R$5)
        MOV     #1,@R$1
        MOV     #2,@R$3
L$14:   MOV     @R$3,R$5
        CMP     R$5,R$2
        BGT     L2
        BGE     L$18
        MOV     R$5,R$4
        ASL     R$4
        ASL     R$5
        CMP     A+2(R$4),A(R$5)
        BLE     L$18
        INC     @R$3
L$18:   MOV     @R$3,R$5
        ASL     R$5
        CMP     A(R$5),@R$0
        BLE     L2
        MOV     @R$1,R$4
        ASL     R$4
        MOV     A(R$5),A(R$4)
        MOV     @R$3,@R$1
        MOV     R$5,@R$3
        BR      L$14
L2:     MOV     @R$1,R$4
        ASL     R$4
        MOV     @R$0,A(R$4)
        DEC     R$2
L$13:   BGT     L$12
```

Following is the code generated by the BLISS-11 compiler compiling the above BLISS program for the PDP-11 for the case in which T, J, K, and N are declared to be registers.

```
        MOV     #2,-(SP)
        BR      L$6
L$5:    MOV     @SP,R$3
        MOV     R$3,R$4
        MOV     R$3,R$1
        ASL     R$1
        MOV     A(R$1),R$5
L$7:    ASR     R$4
        MOV     R$4,R$1
        ASL     R$1
        CMP     R$5,A(R$1)
        BLE     L1
        MOV     R$3,R$0
        ASL     R$0
        MOV     A(R$1),A(R$0)
        MOV     R$4,R$3
        CMP     R$4,#1
        BNE     L$7
L1:     MOV     R$3,R$1
        ASL     R$1
        MOV     R$5,A(R$1)
        INC     @SP
L$6:    CMP     @SP,R$2
        BLE     L$5
        DEC     R$2
        MOV     R$2,@SP
        BR      L$13
L$12:   MOV     @SP,R$2
        ASL     R$2
        MOV     A+2(R$2),R$5
        MOV     @#A+2,A+2(R$2)
        MOV     #1,R$3
        MOV     #2,R$4
L$14:   CMP     R$4,@SP
        BGT     L2
        BGE     L$18
        MOV     R$4,R$1
        ASL     R$1
        MOV     R$4,R$2
        ASL     R$2
        CMP     A+2(R$1),A(R$2)
        BLE     L$18
        INC     R$4
L$18:   MOV     R$4,R$2
        ASL     R$2
        CMP     A(R$2),R$5
        BLE     L2
        MOV     R$3,R$1
        ASL     R$1
        MOV     A(R$2),A(R$1)
        MOV     R$4,R$3
        MOV     R$2,R$4
        BR      L$14
L2:     MOV     R$3,R$0
        ASL     R$0
        MOV     R$5,A(R$0)
        DEC     @SP
L$13:   BGT     L$12
```

3.7.5.3.5 FORTRAN-H Compilation for IBM-370/168

Following is the code generated by the FORTRAN-H compiler compiling a FORTRAN version of the above BLISS program for the IBM-370/168 with full optimization enabled.

```
C1      DC      XL4'1'
C2      DC      XL4'2'
C4      DC      XL4'4'
C8      DC      XL4'8'

        USING   R13

        L       7,C1
        L       0,C8
        ST      0,Q01
        L       9,C2
        L       10,C4
L1      CR      9,11
        BC      2,L3
        LR      2,9
        ST      9,K
        ST      9,J
        L       14,Q01
        L       6,A(14)
        LR      8,2
        L       10,C2
L20     L       2,J
        SRDA    2,32
        DR      2,10
        ST      3,J
        LR      5,3
        SLL     5,2
        L       11,A(5)
        CR      6,11
        BC      3,L2
        ST      8,K
        BC      15,L30
L2      LR      5,8
        SLL     5,2
        ST      11,A(5)
        LR      8,3
        CR      3,7
        BC      7,L20
        ST      8,K
L30     L       11,N
        L       10,C4
        L       2,K
        SLL     2,2
        ST      6,A(2)
        L       0,Q01
        AR      0,10
        ST      0,Q01
        AR      9,7
        BC      15,L1
L3      L       9,N
        SR      9,7
        BC      12,L80
L4      LR      5,9
        SLL     5,2
        L       6,A+4(5)
        L       3,A+4
        ST      3,A+4(5)
        LR      3,7
        ST      7,K
        LR      8,3
        L       11,C2
L50     CR      11,9
        BC      12,L5
        ST      8,K
        BC      15,L40
L5      CR      11,9
        BC      10,L70
L6      LR      3,11
        SLL     3,2
        L       2,A+4(3)
        C       2,A(3)
        BC      12,L70
L7      AR      11,7
L70     LR      3,11
        SLL     3,2
        L       10,A(3)
        CR      10,6
        BC      3,L8
        ST      8,K
        BC      15,L40
L8      LR      3,8
        SLL     3,2
        ST      10,A(3)
        LR      8,11
        SLL     11,1
        BC      15,L50
L40     L       2,K
        SLL     2,2
        ST      6,A(2)
        SR      9,7
        BC      2,L4
L80     DC      0H
```

3.7.5.4 Hand-Coded Quicksort Comparisons

This section compares hand-coded versions of a particular rendition of the Quicksort algorithm. This version of the Quicksort algorithm comes from [Sedgewick 1975] pg. 329.

The following table summarizes the results of these comparisons:

	# INSTRUCTIONS	# BITS
LLL Filter	53	2916
PDP-10	63	2268

It is instructive to compare the inner loops of the various Quicksort programs, and these are marked.

It should be noted that the LLL Filter code has not been highly optimized; by using absolute addresses for arrays, most multiple-word instructions can be reduced to single-word instructions, and furthermore, constants can be shared, eliminating duplicate versions in line.

3.7.5.4.1 ALGOL-W Quicksort Algorithm

This section presents in ALGOL-W the Quicksort algorithm which is hand-coded in the following sections.

Certain liberties have been taken with the ALGOL-W language. Specifically, "INFINITY" is assumed to be a reserved word, the operator ":=:" is the exchange operator, and a macro facility is assumed (eg. "DEFINE N=400;").

```
BEGIN DEFINE N=400; DEFINE M=9;

BEGIN INTEGER ARRAY A(0::N+1);
       INTEGER ARRAY STACK(0::2*(ENTIER(LN((N+1)/(M+2))))+1);
       INTEGER P,L,R,I,J,V,T;

A(0):=-INFINITY;
A(N+1):=INFINITY;

       P:=0; L:=1; R:=N;
PART:  I:=L; J:=R+1; V:=A(L);
       WHILE I<J DO BEGIN
                I:=I+1; WHILE A(I)<V DO I:=I+1;
                J:=J-1; WHILE A(J)>V DO J:=J-1;
                A(J):=:A(I);
                END;
       A(I):=:A(J);
       A(J):=:A(L);
       IF R-J>J-L THEN GO TO RBIG;
       IF J-L<=M THEN GO TO POP;
       IF R-J<=M THEN GO TO LEFT;
       P:=P+2;
       STACK(P):=L;
       STACK(P+1):=J-1;
RIGHT: L:=J+1;
       GO TO PART;
RBIG:  IF R-J<=M THEN GO TO POP;
       IF J-L<=M THEN GO TO RIGHT;
       P:=P+2;
       STACK(P):=J+1;
       STACK(P+1):=R;
LEFT:  R:=J-1;
       GO TO PART;
POP:   L:=STACK(P);
       R:=STACK(P+1);
       P:=P-2;
       IF P>=0 THEN GO TO PART;
INSERT:FOR I:=2 UNTIL N DO
                BEGIN
                V:=A(I); J:=I-1;
                WHILE A(J)>V DO BEGIN A(J+1):=A(J); J:=J-1; END;
                A(J+1):=V;
                END;

END;
END.
```

3.7.5.4.2 LLL Filter Hand-Coding

This section presents a version of the above ALGOL-W program hand coded in LLL Filter assembly language. We assume that P, L, R, I, J, and V are stored in R registers.

```
        MOV             A,#-INFIN
        MOV             A+N+1,#INFIN
        MOV             P,#0
        MOV             L,#1
        MOV             R,#N
PT      MOV             I,L
        INC             J,R
        MOV             V,A(L)

      *** INNER LOOP FOLLOWS ***

L1      ADD             I,#1
        SKIP L      L1  A(I),V
L2      SUB             J,#1
        SKIP G      L2  A(J),V
        EXCH            A(J),A(I)
        SKIP L      L1  I,J

      *** END OF INNER LOOP ***

        EXCH            A(J),A(I)
        EXCH            A(J),A(L)
        SUB             RTA,R,J
        SUB             RTB,J,L
        SKIP LE     L11 RTA,RTB
        JUMP        RB
L11     SKIP G      L7  RTB,#M
        JUMP        PP
L7      SKIP G      L9  RTA,#M
        JUMP        LF
L9      MOV             STACK+2(P),L
        DEC             STACK+3(P),J
        ADD             P,#2
RT      INC             L,J
        JUMP        PT
RB      SKIP G      L10 RTA,#M
        JUMP        PP
L10     SKIP LE     RT  RTB,#M
        INC             STACK+2(P),J
        MOV             STACK+3(P),R
        ADD             P,#2
LF      DEC             R,J
        JUMP        PT
PP      MOV             L,STACK(P)
        MOV             R,STACK+1(P)
        SUB             P,#2
        JUMP GE 0   PT  P
IN      MOV             I,#2
        SKIP LE     L6  I,#N
        JUMP        L3
L6      DEC             J,I
        MOV             V,A(I)
        SKIP LE     L5  A(J),V
L4      MOV             A+1(J),A(J)
        SUB             J,#1
        SKIP G      L4  A(J),V
L5      MOV             A+1(J),V
        INC SKIP G  L3  I,#N
        JUMP        L6
L3      .
```

3.7.5.4.3 PDP-10 Hand-Coding

This section presents a version of the above ALGOL-W program hand coded by John Reiser in PDP-10 assembly language. We assume that P, L, R, I, J, and V are stored in registers, and we call those registers RP, RL, RR, RI, RJ, and RV. In addition, we use the names RT1, RT2, and RT3 to refer to distinct temporary registers.

```
NINF    -2**35

        MOVE    RT1,NINF
        MOVEM   RT1,A
        MOVMM   RT1,A+N+1
        MOVEI   RP,STACK-1
        MOVEI   RL,1
        MOVEI   RR,N
PART    MOVEI   RI,(RL)
        MOVEI   RJ,1(RR)
        MOVE    RV,A(RL)

  *** INNER LOOP FOLLOWS ***

L1      CAMLE   RV,A+1(RI)
        AOJA    RI,L1
L2      CAMGE   RV,A-1(RJ)
        SOJA    RJ,L2
        MOVE    RT1,A-1(RJ)
        EXCH    RT1,A+1(RI)
        MOVEM   RT1,A-1(RJ)
        CAILE   RJ,2(RI)
        JRST    L1

  *** END OF INNER LOOP ***

        MOVE    RT1,A-1(RJ)
        EXCH    RT1,A+1(RI)
        EXCH    RT1,A(RL)
        MOVEM   RT1,A-1(RJ)
        MOVEI   RT2,(RR)
        SUBI    RT2,(RJ)
        MOVEI   RT3,(RJ)
        SUBI    RT3,(RL)
        CAIGE   RT3,2(RT2)
        JRST    RBIG
        CAIG    RT3,M
        JRST    POP
        CAIG    RT2,M
        JRST    LEFT
        PUSH    RP,RL
        MOVEI   RT1,-2(RJ)
        HRLM    RT1,(RP)
RIGHT   MOVEI   RL,(RJ)
        JRST    PART
RBIG    CAIG    RT2,M
        JRST    POP
        CAIG    RT3,M
        JRST    RIGHT
        PUSH    RP,RJ
        HRLM    RR,(RP)
LEFT    MOVEI   RR,(RJ)
        JRST    PART
POP     TLNN    RP,-1
        JRST    INSERT
        HLRZ    RR,RL
        JRST    PART
INSERT  MOVEI   RI,RN
        SOJLE   RI,OUT
TOP     MOVE    RV,A(RI)
        CAMG    RV,A+1(RI)
        JRST    BOT
        MOVEI   RJ,1(RI)
        CAMLE   RV,A+1(RJ)
        AOJA    RJ,.-1
        MOVSI   RT1,A+1(RI)
        HRRI    RT1,A(RI)
        BLT     RT1,A-1(RJ)
        MOVEM   RV,A(RJ)
BOT     SOJG    RI,TOP
OUT     .
```

4. Implementation

4.1 Processing Element

The major features of the processing element implementation are as follows:

- State-of-the-art high-speed ECL logic.
- Triple micro-controllers, two for fetching instructions and operands, and one for executing instructions.
- An instruction set defined in a writeable control store which can be dynamically modified to accomodate the special requirements of some codes.
- Special data paths for the rapid execution of floating-point instructions.
- Hamming-coded main memory to allow the use of cost-effective 4K-bit and 16K-bit RAM chips.

The processing element is shown in Figure 2.2-1. The entire processing element, including control store, requires approximately 4000 ECL 10K ICs. The processing element cycles in 100 nano-seconds, with register-to-register and register-to-memory integer adds proceeding in pipeline mode at 100 nano-seconds per instruction. With stable rounding, floating addition takes 6 cycles, and floating multiply takes 11 cycles. With truncation, floating addition takes 5 cycles, and floating multiply takes 10 cycles.

The processing element contains three independent micro-programmed processors, which are designated the P-sequencer, the I-sequencer, and the E-sequencer. The P-sequencer does the basic instruction decode which takes care of the different operand types, and register operands. The I-sequencer calculates memory, indexed, and indirect operands, in addition to controlling things like cache misses and the interaction with the switch. The E-sequencer executes all of the basic instructions, once the P and I sequencers have fetch the operands, and scheduled the write(s) for the result(s). All three of the sequencer's have writeable control stores, which can be dynamically changed.

In this discussion "macro-instruction" ("macro-operation") will mean the sequence of micro-instructions executed by the three sequencers to emulate a user-level instruction.

Drawings in general will be referenced by an abbreviation which is given in all capital letters. For example the drawing for the instruction box has the abbreviation IBOX.

The drawings are the output of an advanced computer-aided design system; they are a *hierarchical* representation of the machine. In general, a single page is the definition of a *macro-body* included in a drawing at a higher level; the definition may use macro-bodies which are defined at a lower level. The name of a macro-body appears inside the body at the call site; it is also the title of the body definition. Most macro-body definitions are one page, although multiple-page definitions are allowed. Multiple-page definitions are indicated by placing a page number (for example, "1/2") in the title of each drawing of the definition.

Lines in the drawings represent bundles of signals. The notation X<i:j> means the bundle of signals X<i>, X<i+1>, ... , X<j>. The notation X:Y:Z means the bundle of signals (or vectors of

signals) X, Y, and Z, in that order. Special "merger" bodies are also used to bundle separately named signals.

The parameter passing mechanism is similar to that of ALGOL; actual parameters may be passed to a macro-body where it is used (paramters are bundles of signals) and the body definition may refer to those parameters by their formal names. Global signals will be declared, although no declarations have yet been made on these drawings. Any macro-body can refer to global signals which are delared at a higher level.

The definitions of most low-level bodies are not shown in this report, although an appendix contains some low-level definitions.

4.1.1 IBOX/EBOX Communication

This section describes the signals which connect the IBOX and EBOX. In the logic diagrams, all signals connecting the IBOX an EBOX are prefixed with the character "X". Times in parentheses indicate when the signal is available in the sender's reference frame.

4.1.1.1 IBOX to EBOX Signals

START ADR<0:11> (T40)
Starting address in the EBOX of the sequence of micro-operations which emulate the current instruction.

A OP<0:35> (T50)
Operand to the EBOX. A OP is normally the operand described by OD1.

B OP<0:35> (T50)
Operand to the EBOX. B OP is normally the operand described by OD2.

USE A OP (T50)
This signal allows the IBOX to wrap the EBOX result around into the A input. If this signal is not set and the EBOX is reading an operand from the IBOX, then the operand read into the A input is simply the result of the last EBOX cycle.

USE B OP (T50)
This signal allows the IBOX to wrap the EBOX result around into the B input. If this signal is not set and the EBOX is reading an operand from the IBOX, then the operand read into the B input is simply the result of the last EBOX cycle.

BRANCH TAKEN (T50)
During conditional branch instructions, this signal indicates that the IBOX took the branch.

BRANCH COND<0:2> (T50)
During conditional branch instructions, these signals indicate the one of eight branch conditions coded in the instruction.

A OP LOW ADR<0:1> (T50)
The least-significant two bits of the A operand address. These bits are used in quarter-word and half-word operations.

B OP LOW ADR<0:1> (T50)
The least-significant two bits of the B operand address. These bits are used in quarter-word and half-word operations.

DEST LOW ADR<0:1> (T50)
The least-significant two bits of the destination operand address. These bits are used in quarter-word and half-word operations.

KILL EBOX (T50)
Stop the EBOX unconditionally.

PAUSE EBOX (T50)
This signal can be tested by the EBOX and if asserted, will cause a soft stop to occur.

4.1.1.2 EBOX to IBOX Signals

USING OPS (T4)
This signal indicates to the IBOX that if the input operands are not ready for the EBOX, then the EBOX clock should be stopped until the input operands become ready.

OPS TAKEN (T10)
This signal indicates to the IBOX that the input operands have been loaded into the EBOX and therefore the IBOX operand registers can be reloaded.

RESULT DATA<0:35> (T20)
The result of a sequence of micro-operations.

TRAP (T20)
The instruction in execution has trapped.

RESULT (T20)
A result is available on RESULT DATA<0:35>.

DONE (T20)
The EBOX is done with the current sequence of operations and is ready to accept a new starting address.

INTERRUPT IBOX (T20)
Interrupt the IBOX. Several cycles are wasted in cleaning up the IBOX to prepare for an IBOX/EBOX dialogue.

WRONG BRANCH (T21)
The IBOX took the wrong direction on the conditional branch currently in execution.

4.1.2 Instruction Box

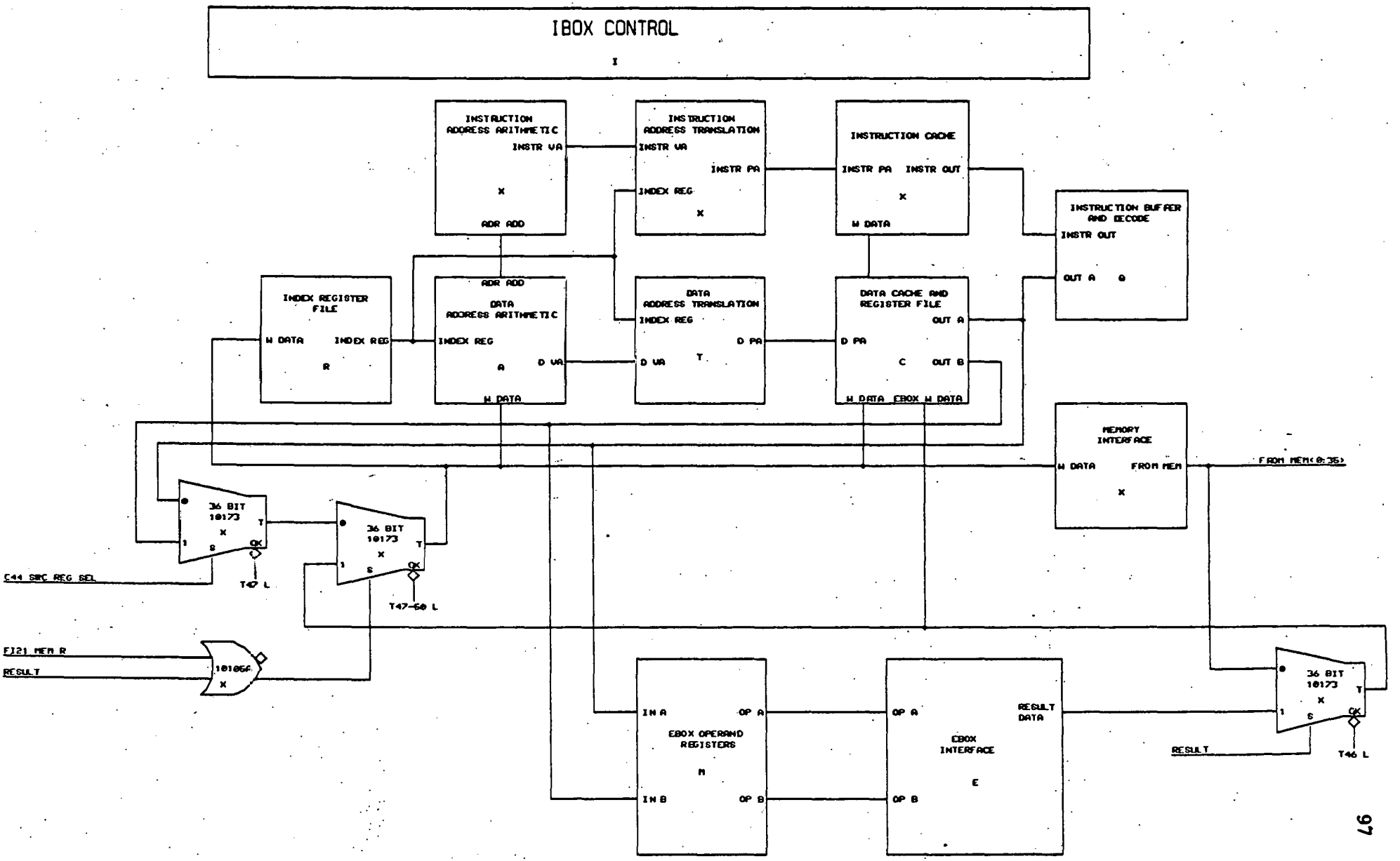
The instruction box (IBOX) controls the fetching of instructions and operands, the interaction with the crossbar switch to read and write main memory, and all I/O operations.

The IBOX has two caches, one for instructions and one for data, which each hold 4K words. The main reasons for having two caches is that it doubles the cache bandwidth, and simplifies the scheduling of cache operations, since the instruction prefetch logic has its own dedicated cache. A given word of memory can only be in one of the two caches at a time. When ever a miss occurs in one of the caches, the other cache is checked for that word. If it is found there, then it is moved from the one cache to the other. In addition, the instruction cache does not have any modify bits, so if a modified word is moved from the data cache to the instruction cache, then it is also written back to main memory.

The main register stack is 128 words by 36-bits, which contains the three sets of registers for the user, and a set of temporary registers for use by the IBOX. All of the registers are stored three times, which allows three different registers to be read out at the same time. During each micro-cycle, one register write and three reads may be done.

One of the register stacks exists in the Index Register File macro, and is used for index operations. The other two are in the Data Cache and Register File macro, which are used for reading register operands for instructions.

The Instruction Address Arithmetic, Instruction Address Translation, Instruction Cache, and Instruction Buffer and Decode macros all have to do with prefetching instructions. The Index Register File, Data Address Arithmetic, Data Address Translation, and Data Cache and Register File macros are used for the calculation of operands. Memory Interface allows memory read and write operations to be done to the switch. One of its more interesting features is that it puts hamming codes on the data before it goes to the switch, and checks and corrects it when it comes back. That way, if there is an error introduced any place between the processor and the memory, it can be corrected if its a single error, and detected if its a double error. The EBOX Operand Register macro holds the next pair of operands for the EBOX, and the EBOX Interface macro just specifies the interconnections between the IBOX and the EBOX



Instruction Box (IBOX)

4.1.2 Instruction Box Pipeline Timing

The IBOX Pipeline Timing shows an example of the parallelism which results in the IBOX when a series of contiguous instructions are executed, each of which requires a single EBOX execution cycle. Each box in the figure represents a 100ns event.

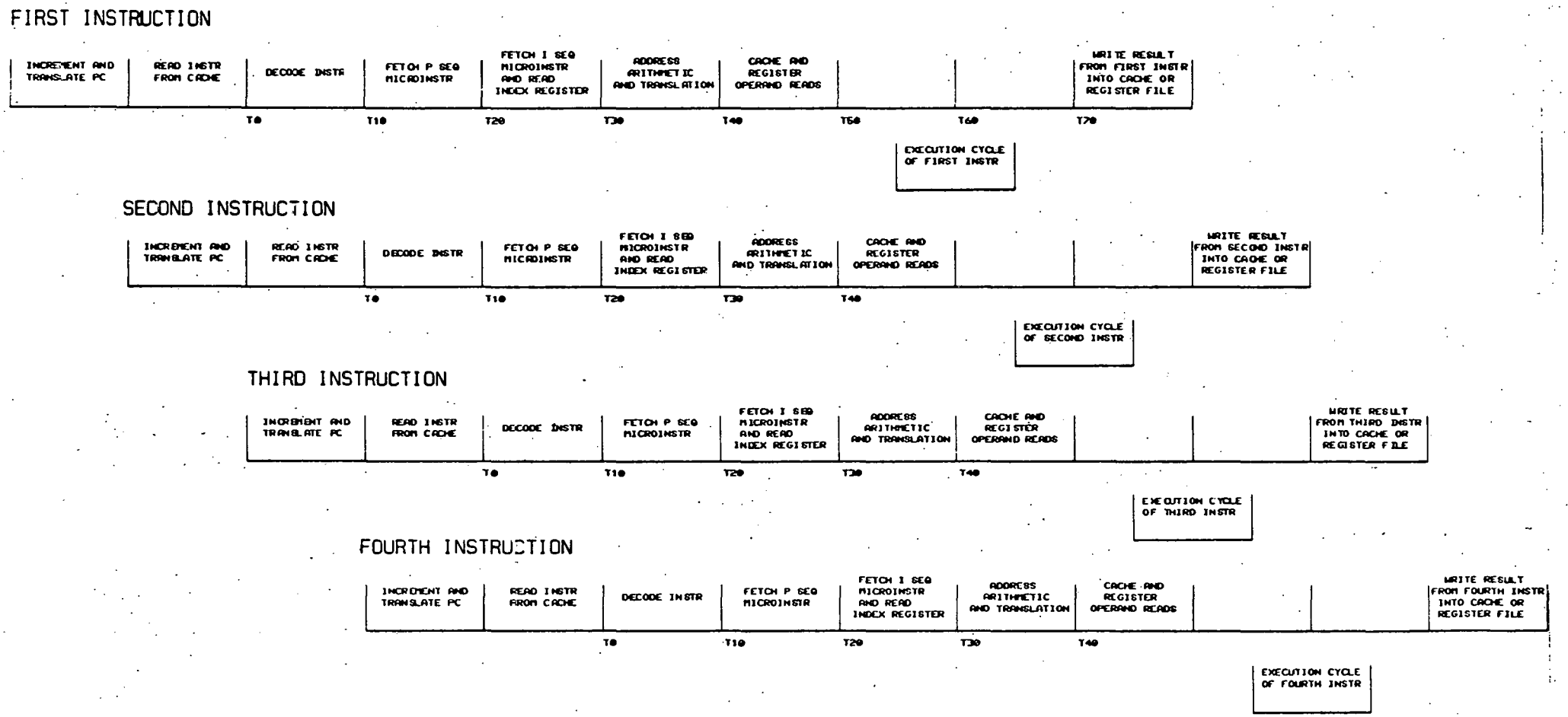
The prefetch logic fetches an instruction every cycle, as long as the pipeline can use the instructions. The prefetch logic looks at the instructions as they are decoded, and if it sees an unconditional branch, it takes it. If it sees a conditional short PC relative branch or skip backwards, then it assumes that it is a loop, and also jumps backwards. In all other cases, it fetches the next instruction assuming the branch is false. When the conditional branch is executed, if the prefetch logic went the wrong way, the pipeline if flushed, and the processor starts fetching instructions the other direction.

Once the instruction is decoded, the next step is to fetch the P-sequence micro-instruction for the instruction. The P-sequence micro-instruction then specifies a starting address in the I-sequencer, and calculates register addresses for the register operands. Depending on the operand formats for the specific instruction, and the specific addressing modes used, a number of P-sequence and I-sequence micro-instructions may be done.

After an I-sequencer micro-instruction is executed, there is a two stage pipe. The first stage of the pipe calculates addresses and does a virtual to real address translation. The virtual to real address translation was not done in parallel with the cache read so that the page size could be smaller than the size of the chips used to implement the cache, which are 1K bit ECL RAMs. The second stage of the pipe can then do two register reads, or a register read and a cache read. If a register is read as a memory location, then the hardware automatically reads the correct register.

After the operands of the instruction are read, then a half cycle is allowed for the operands to get to the EBOX. The EBOX then executes the instruction taking some number of cycles, and writes the result(s) back. The addresses of the result(s) have already been scheduled at this time, and hardware logic actually does the writes. If a write conflicts with what the IBOX wants to do during a given cycle (i.e. the IBOX wants to do a cache read, and the EBOX wants to do a cache write), then the clock for the IBOX is stoped for a cycle, and the write occurs. For most addressing modes, the IBOX does not need to write into the cache or the general register file, so very few write conflicts should occur.

There is a set of comparators which take care of the cases where a result of one instruction is used in one of the next two instructions, which causes the appropriate data to bypass the cache or register file, with no loss in time. The only place where execution time is lost is where an instruction tries to index off of a rescently generated result, in which case up to three cycles may be lost. Because of this, it is 2 cycles faster to index off a local variable on your stack, than it is to load it into a register and then index off of it once.

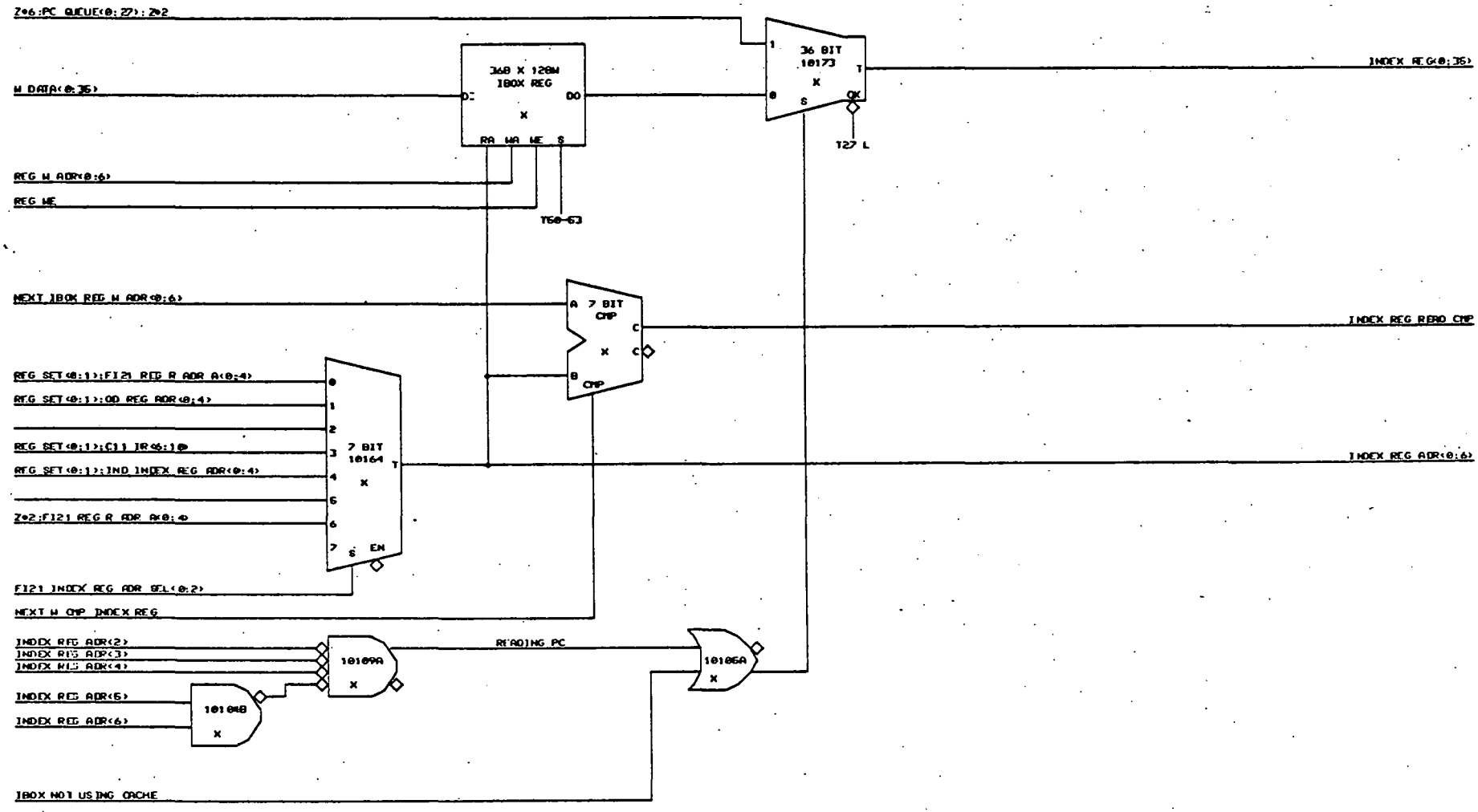


IBOX Pipeline Timing (IBOXT)

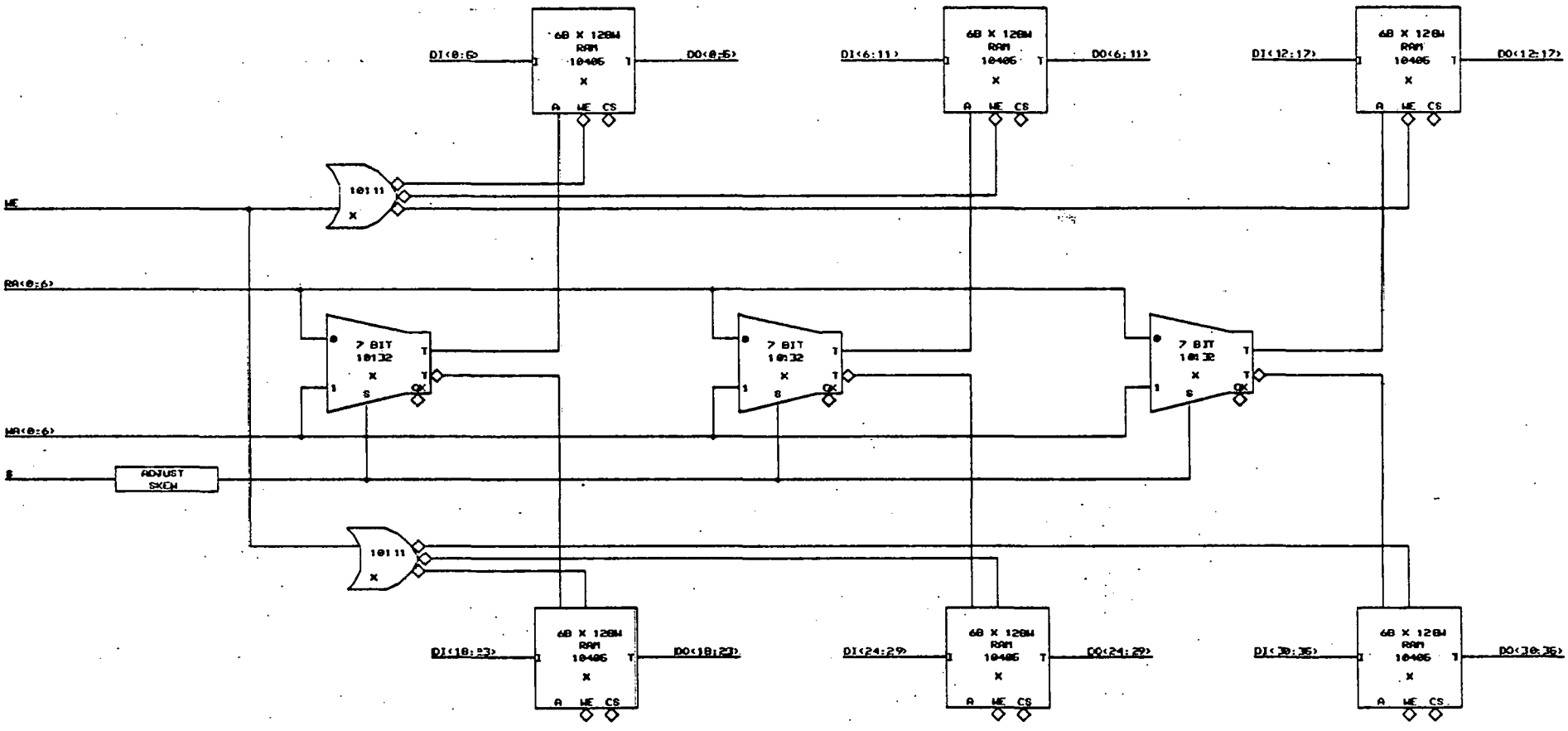
4.1.2.1 Index Register File

The index register file is used for reading registers which are used in address arithmetic, such as in index operations and register indirection. The multiplexer is used to determine the source of the register address, and the comparator is used to detect that the next cycle is writing into the register being read, to allow the appropriate data to bypass the index register file, saving a cycle.

The IREGM drawing shows how the 36B x 128W register file is implemented.



Index Register File (IREGF)

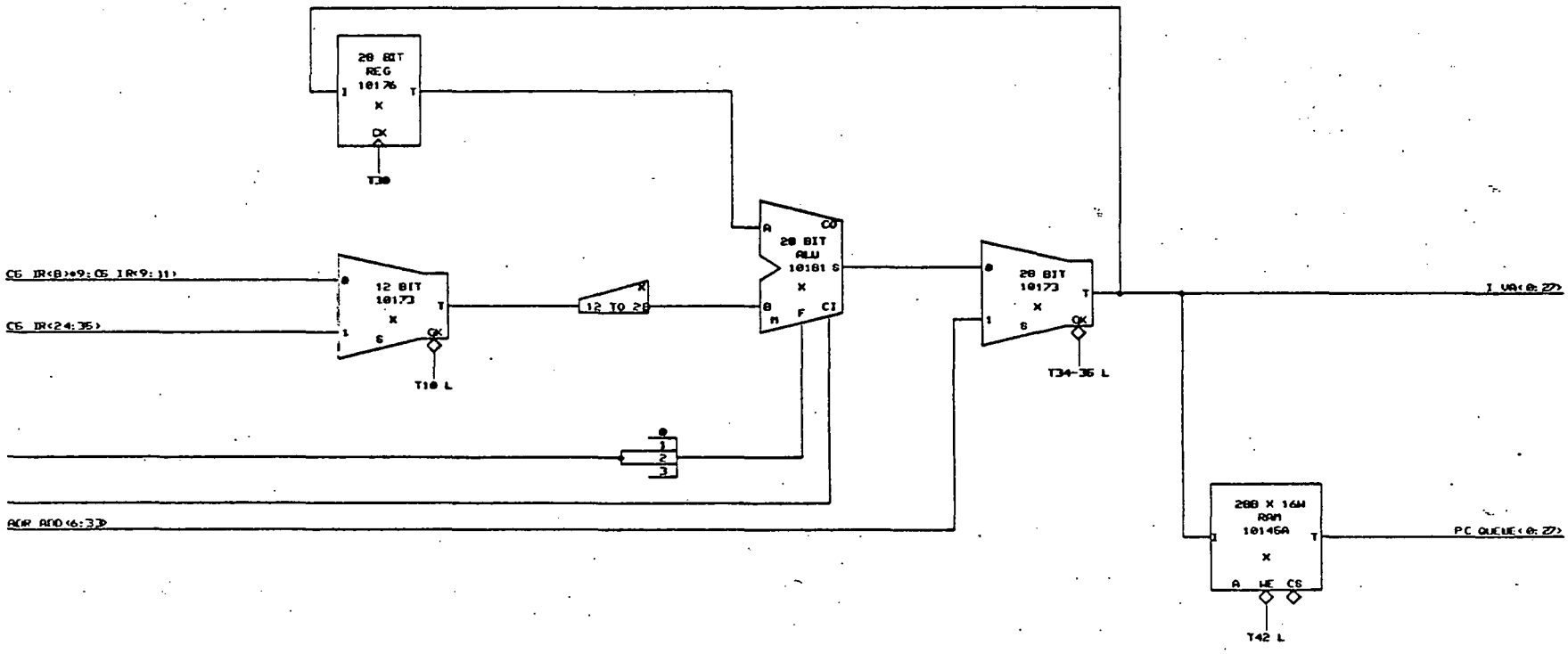


36B X 128W IBOX REG (IREGM)

4.1.2.2 Instruction Address Arithmetic

The Instruction Address Arithmetic logic calculates the address of the next instruction to be fetched if it PC+4 (next word), or the destination of a PC relative skip or short jump. In all other cases, the Data Address Arithmetic logic is used to calculate the address of the next instruction.

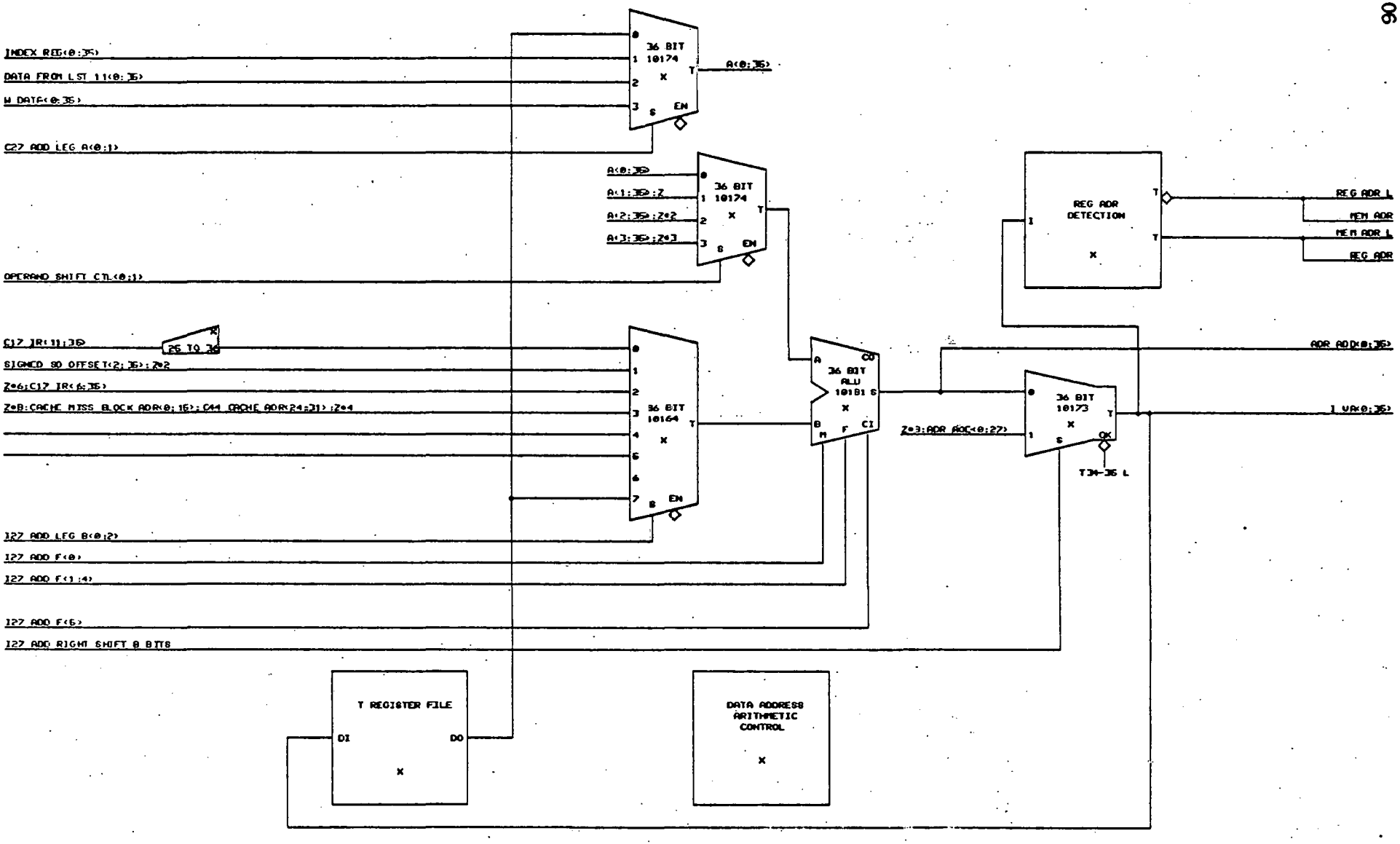
The 28B x 128W RAM is used to remember the PC of all instructions in the pipe, in case one of themgets an error, or the pipeline gets flushed for some reason.



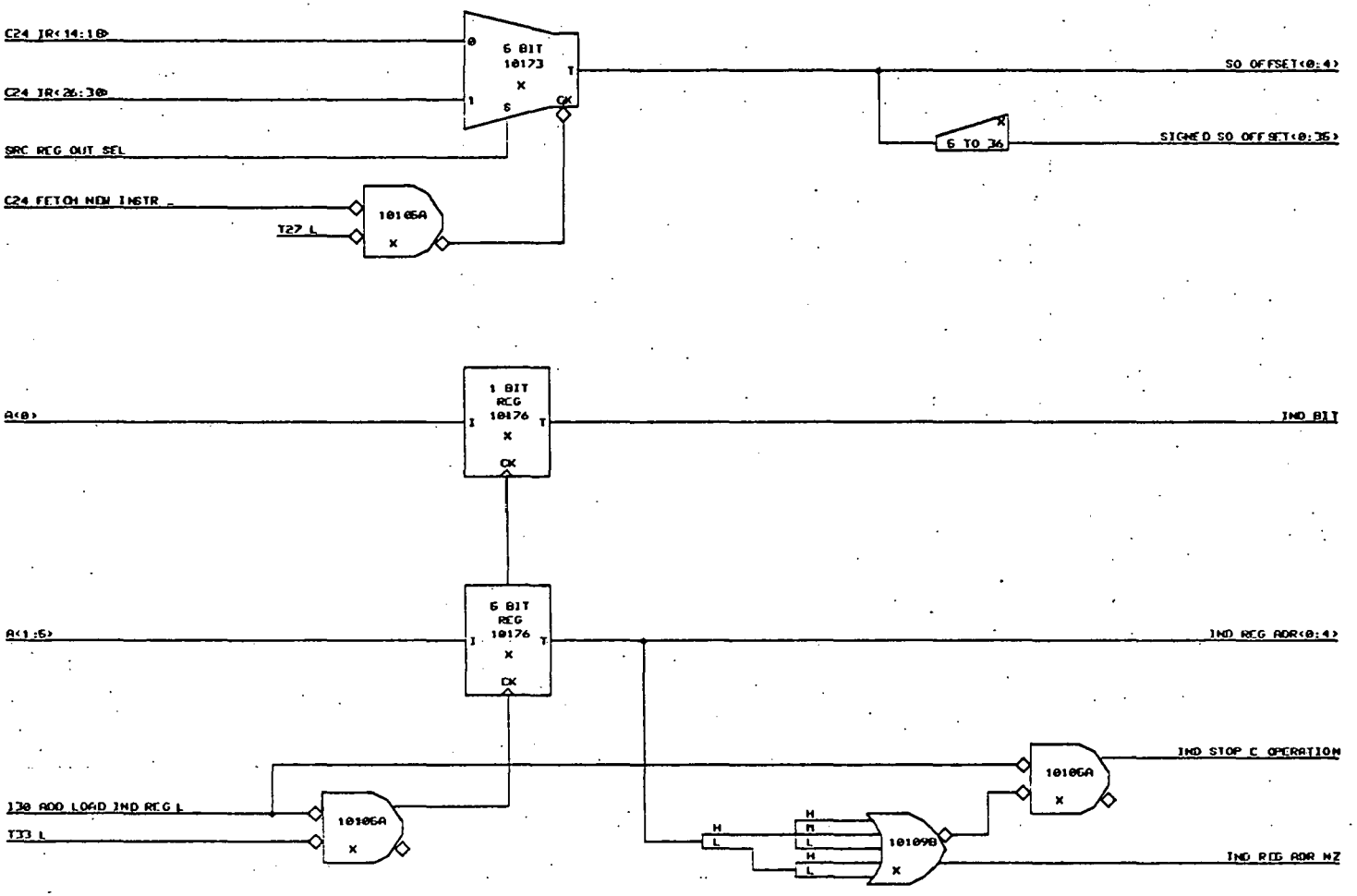
Instruction Address Arithmetic (IADRA)

4.1.2.3 Data Address Arithmetic

The Data Address Arithmetic logic does all of the non-register operand address calculations. It contains a set of 16 36-bit temporary registers (see the T REGISTER FILE macro), which can be used in the calculation of addresses. The REG ADR Detection logic detects if the address generated is a register, which causes the cache read to be automatically turned into a register read.



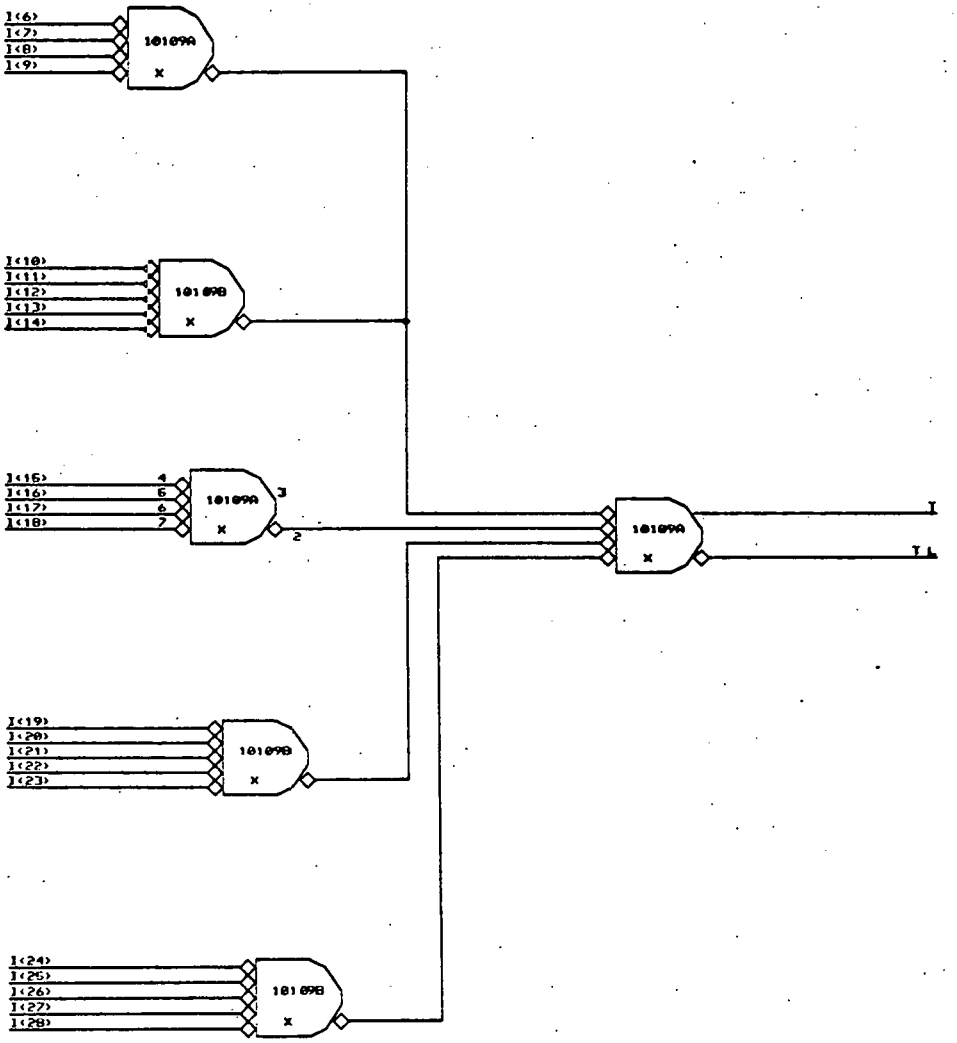
Data Address Arithmetic 1/2 (ADRAR1)



Data Address Arithmetic 2/2 (ADRAR2)

4.1.2.3.1 Register Address Detection

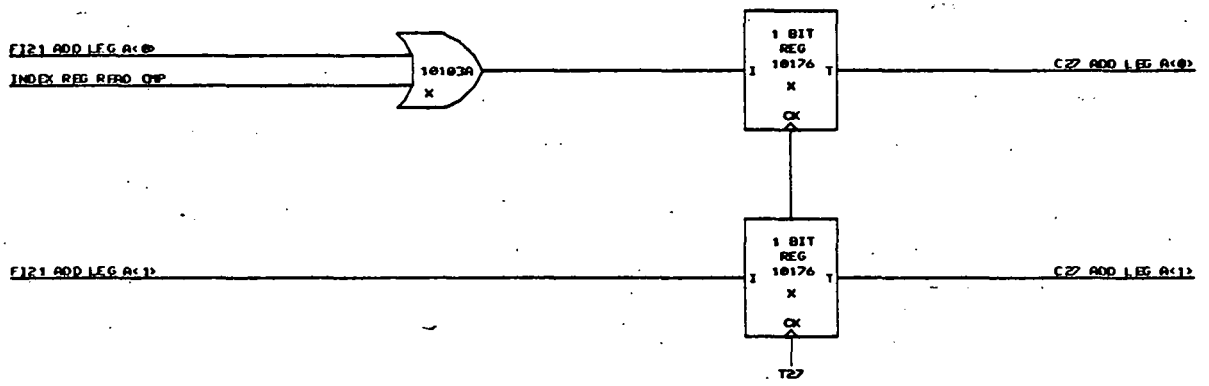
The Register Address Detection logic checks to see of the memory address is in the range of 0 to 127, in which case it is a register address.



REG ADR Detection (RADRD)

4.1.2.3.2 Data Address Arithmetic Control

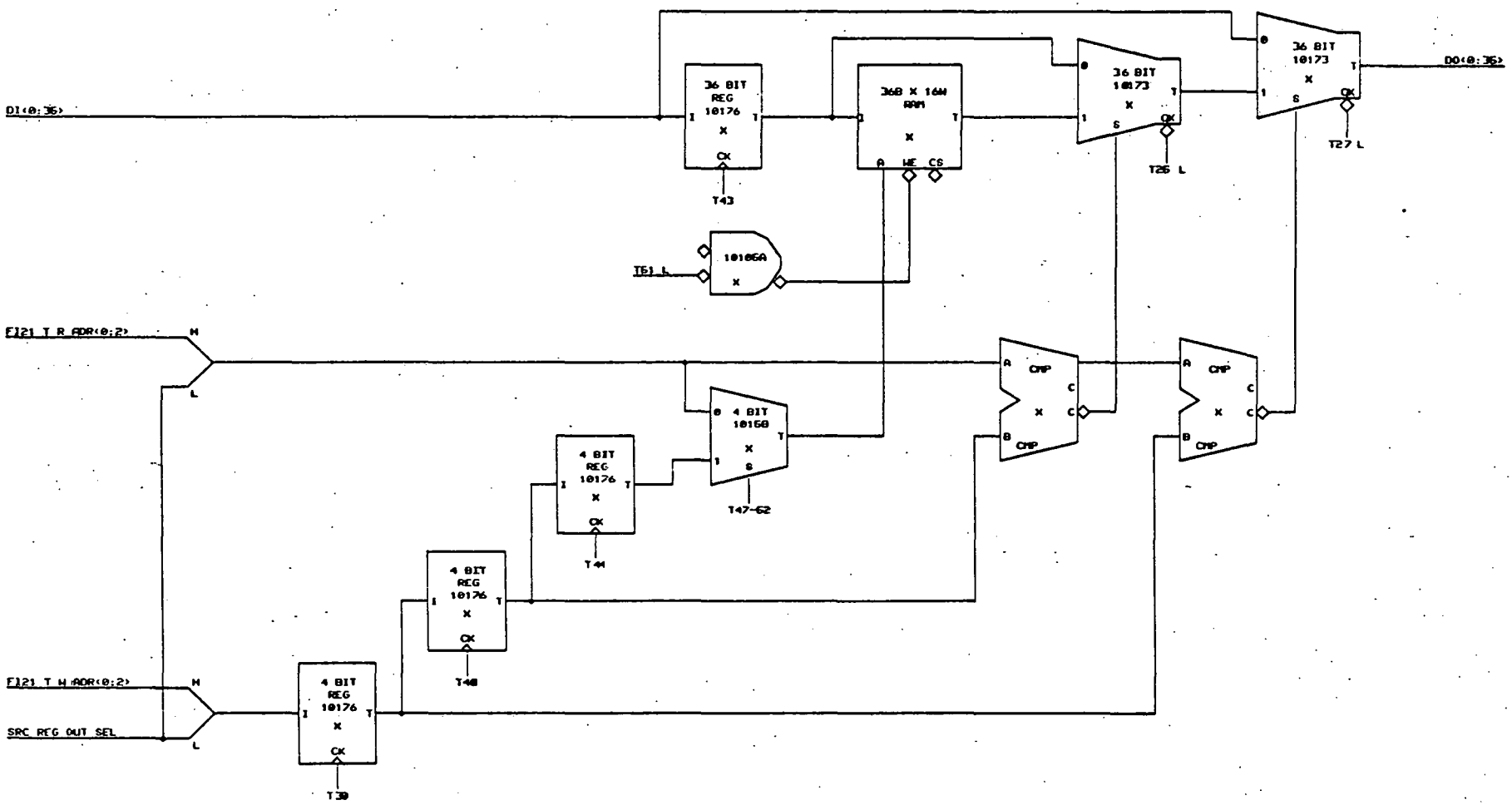
The Data Address Arithmetic Control causes the write data (WDATA) bus to be selected in the Data Address Arithmetic logic, if the word being read out of the Index Register File is being written the next cycle.



Data Address Arithmetic Control (ADRARC)

4.1.2.3.3 T Register File

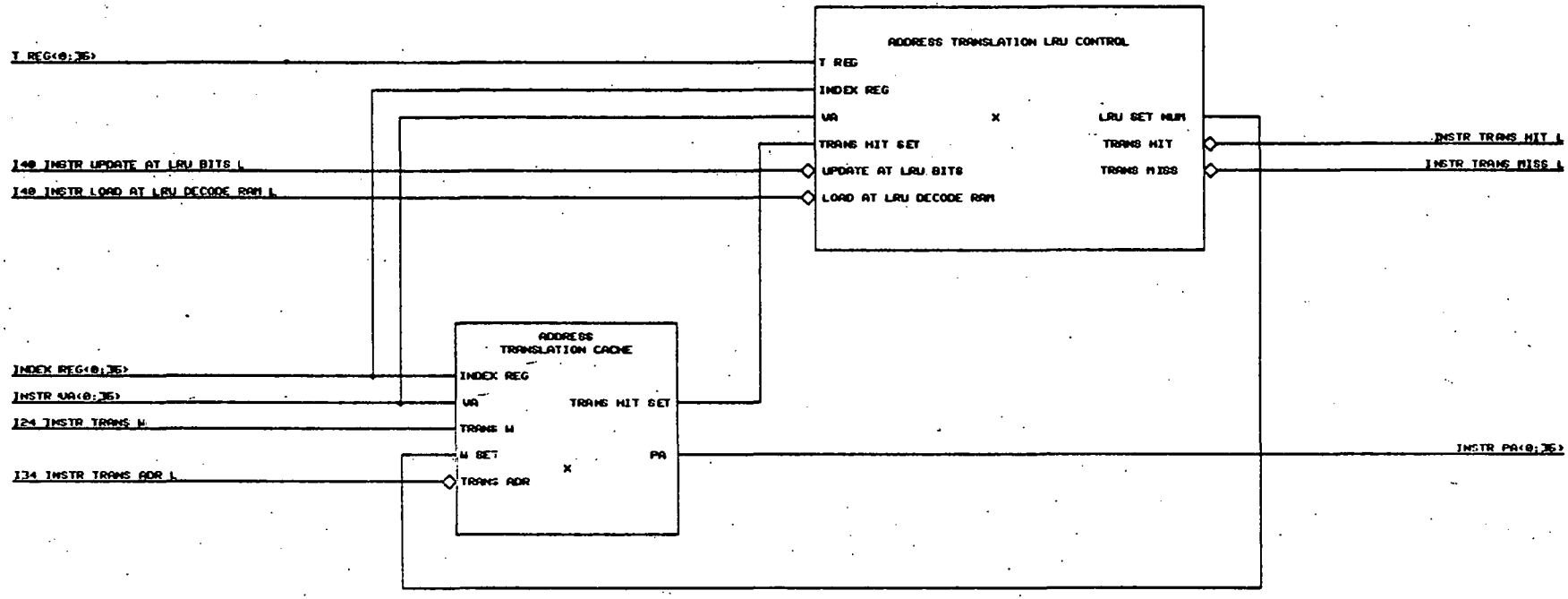
The T Register File is a set of 16 registers for use in calculating addresses. They are written into from the output of the data address arithmetic adder, and can be read into the A or B leg of the adder. The control of this register file is particularly complicated because results to be written into it have to be delayed for two cycles, in case a micro-interrupt occurs, and the instruction doing the write has to be canceled.



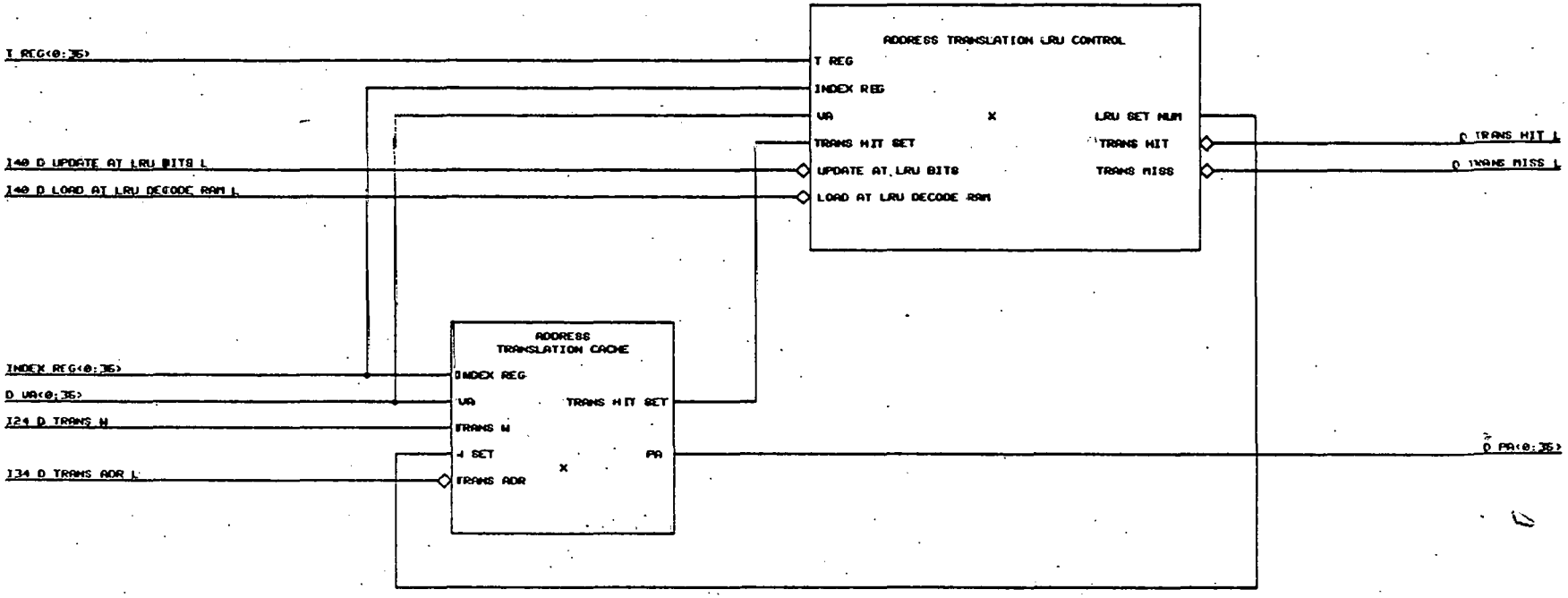
T Register File (TREGF)

4.1.2.4 Instruction and Data Address Translation

The Instruction Address Translation and Data Address Translation logic translates virtual addresses for the instruction and data caches into physical addresses. The address translation is done by a lookup in a small set associative cache, which has 64 words, and a set size of 4. Because of the very large address space (30-bits), this method was prefered to the more conventional method of using a direct mapping cache for the address translation. Since different data is stored in the two address translation caches, up to 128 different page translations can be kept in the processor.



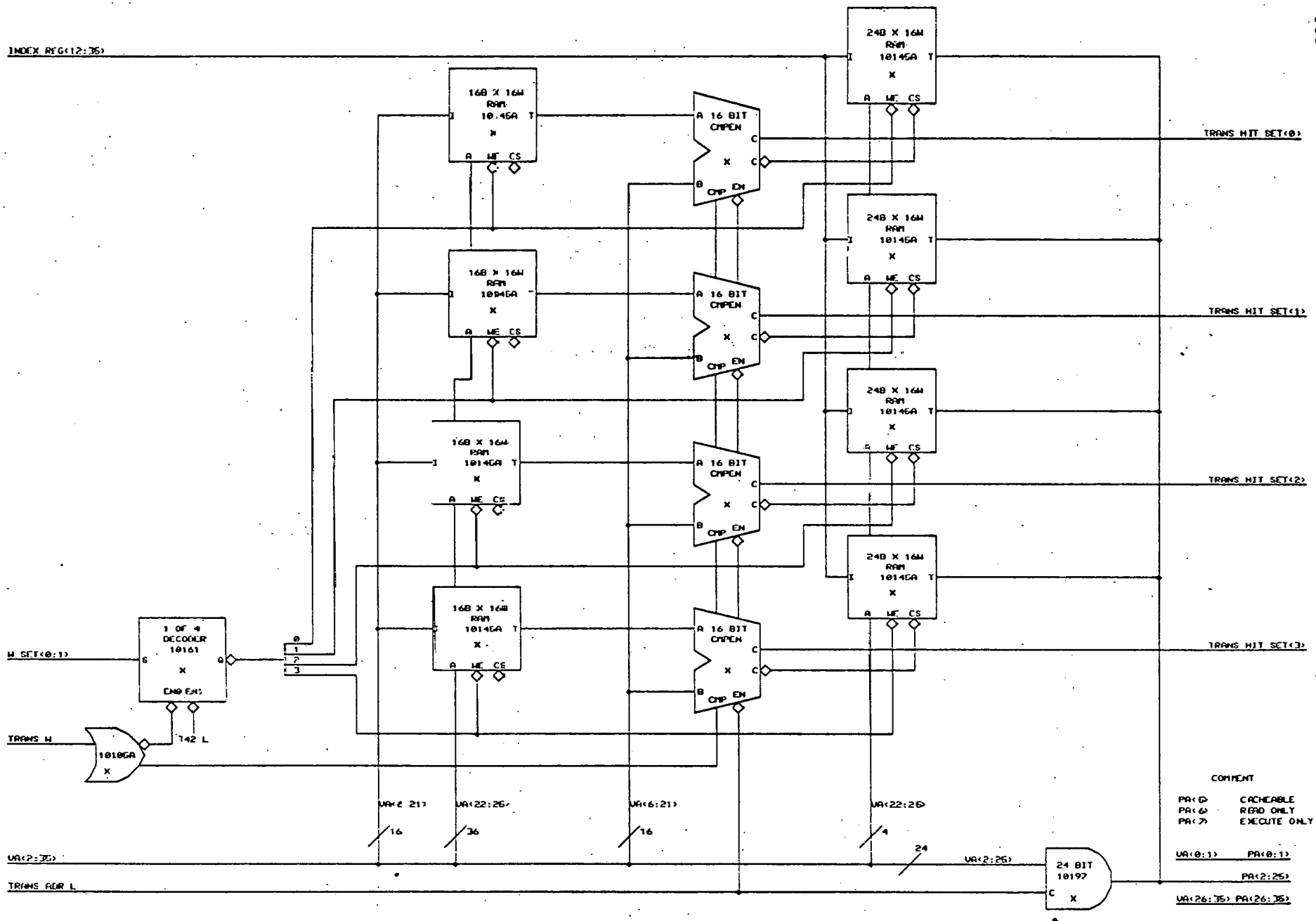
Instruction Address Translation (IADRTN)



Data Address Translation (DADRTN)

4.1.2.4.1 Address Translation Cache

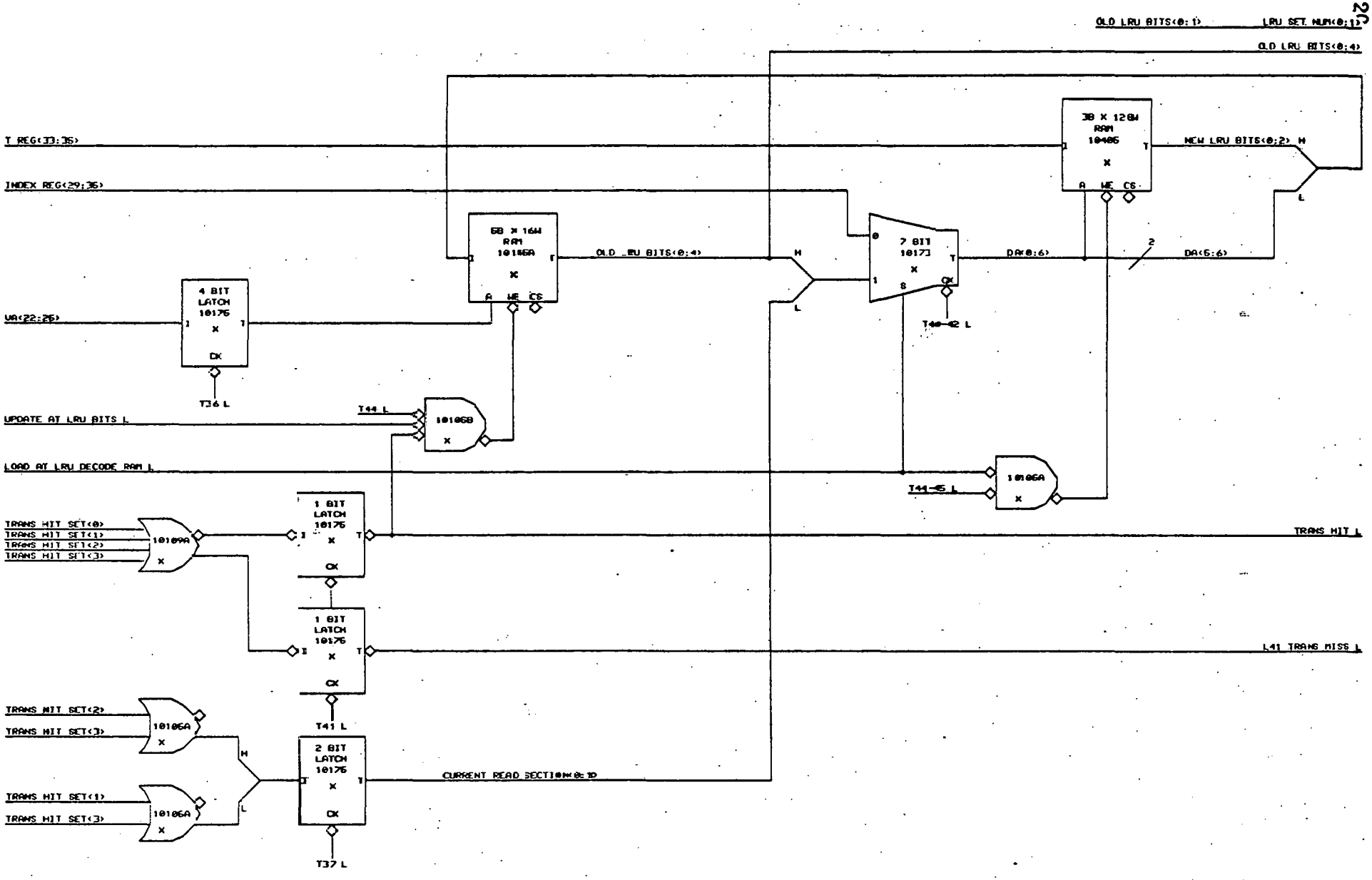
The Address Translation Cache is a standard set associative cache with a set size of 4, and with 64 total entries. The input to the cache is the bus VA<0:35>, where VA<6:35> contains the address to be translated. The way the cache operates is to look up four words based on VA<22:25>, and to compare the address stored there to VA<6:21>. If one of those words match, then the physical address stored in that location is read out. Otherwise, the address translation required is not stored in the cache, and a micro-interrupt occurs.



Address Trarslation Cache (ADRTRN)

4.1.2.4.2 Address Translation LRU Control

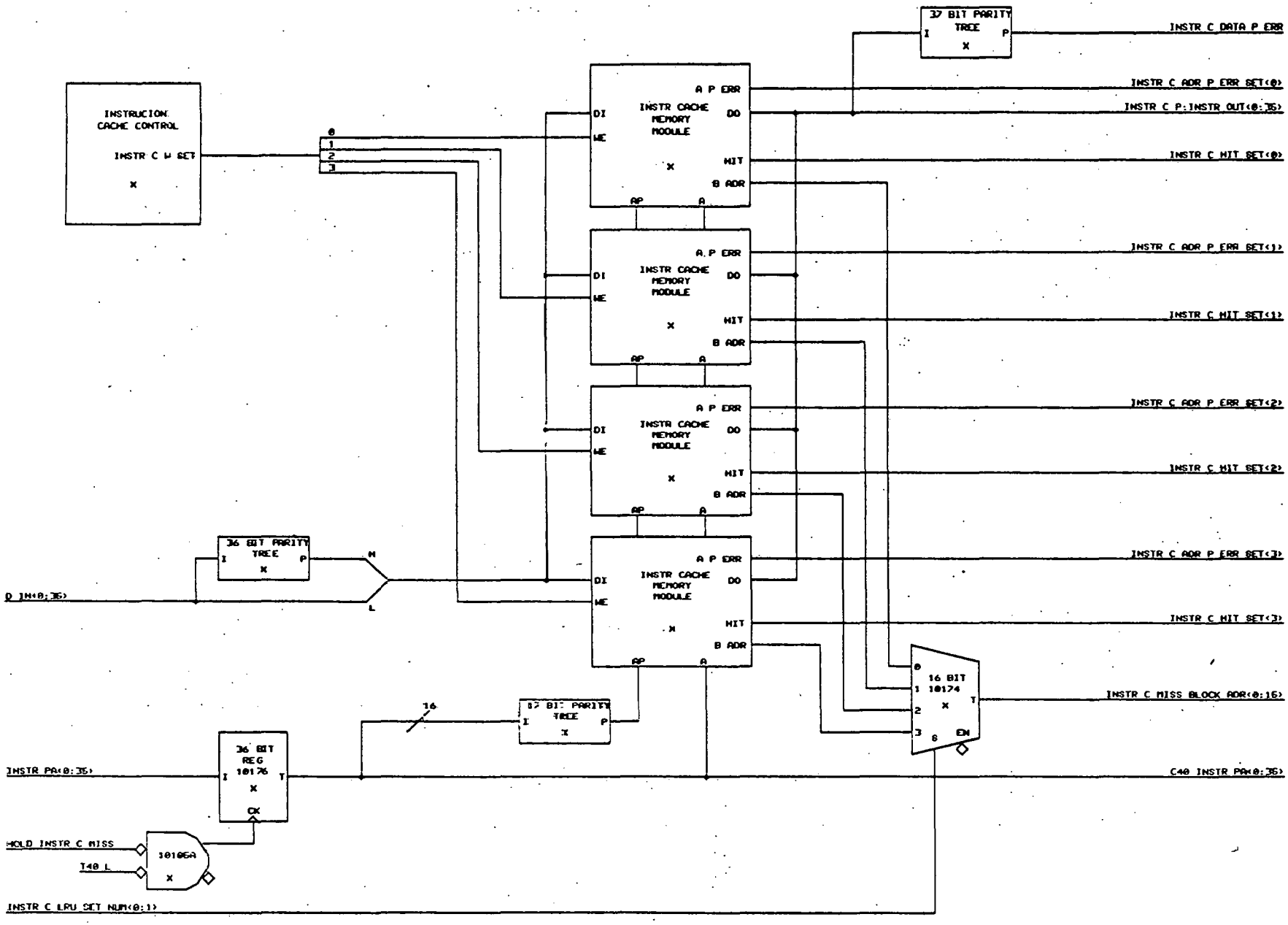
The Address Translation LRU Control keeps track of the least rescently used word in each set in the Address Translation Cache, so that when an element needs to be replaced in the cache, that word can be the one. The way this is implemented is as follows. For each set in the cache, there are five bits stored, two of which specify the most rescently used word, two which give the least rescently used word, and one which tells the order of the other two words. In order to update these five bits on a reference to the cache then, these five bits and two bits which tell which word is currently being referenced are fed into the address lines of a RAM which is programmed to give the new five bits for this set. It should be noted that the two bits which give the most rescently referenced word are just the current word being referenced, so they do not need to be generated by the RAM.



Address Translation LRU control (ATLRU)

4.1.2.5 Instruction Cache Memory

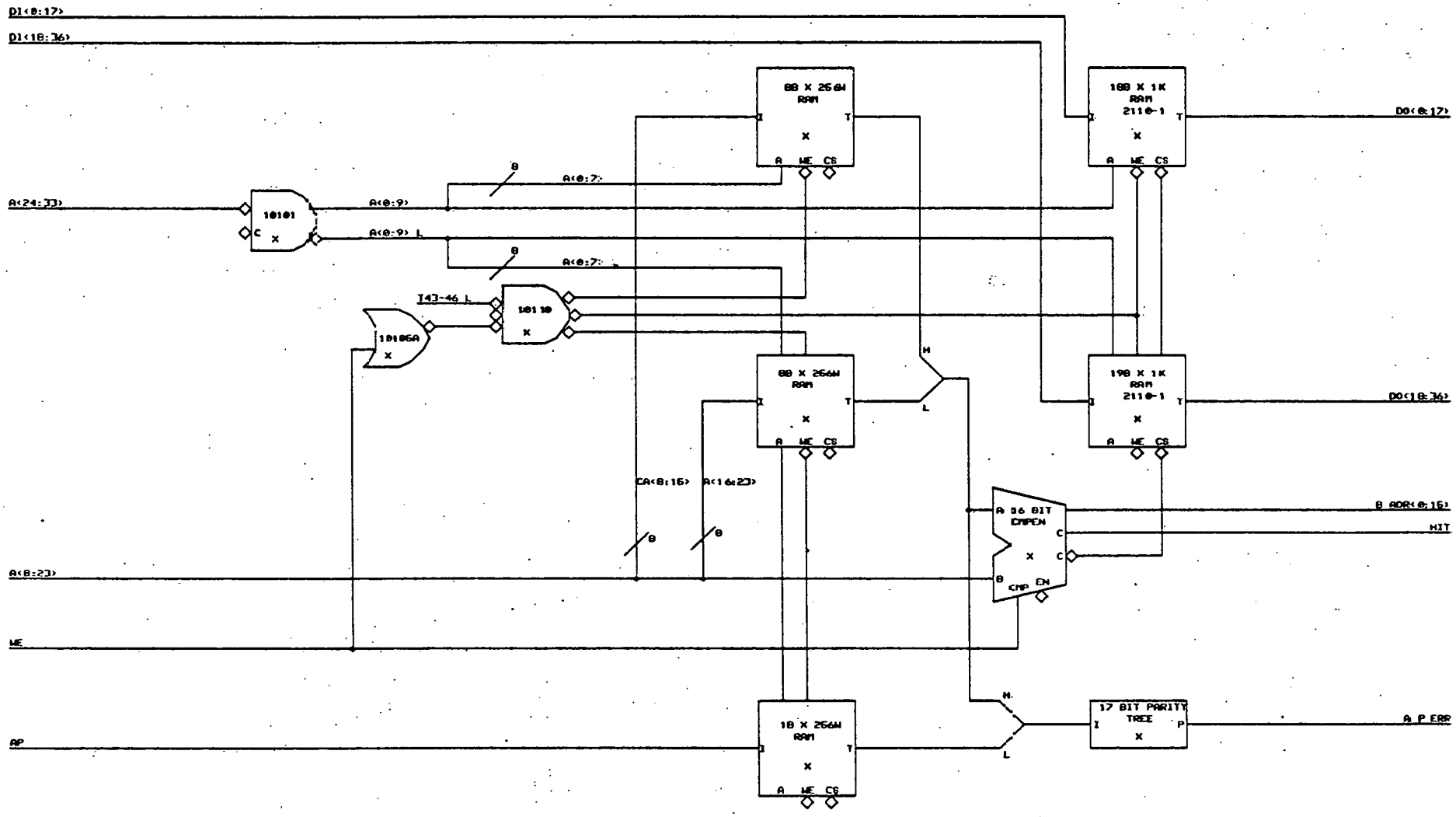
The Instruction Cache Memory and the Data Cache Memory are both basically the same, and are conventional set associative cache organizations. They each hold 4K 36-bit words, and have a set size of 4. The instruction cache does not have a modify bit, so writes to it must also go to memory. The data cache has a modify bit for every four words, and words are always transfered between the caches and main memory in groups of 4.



Instruction Cache Memory (ICACHE)

4.1.2.5.1 Instruction Cache Memory Module

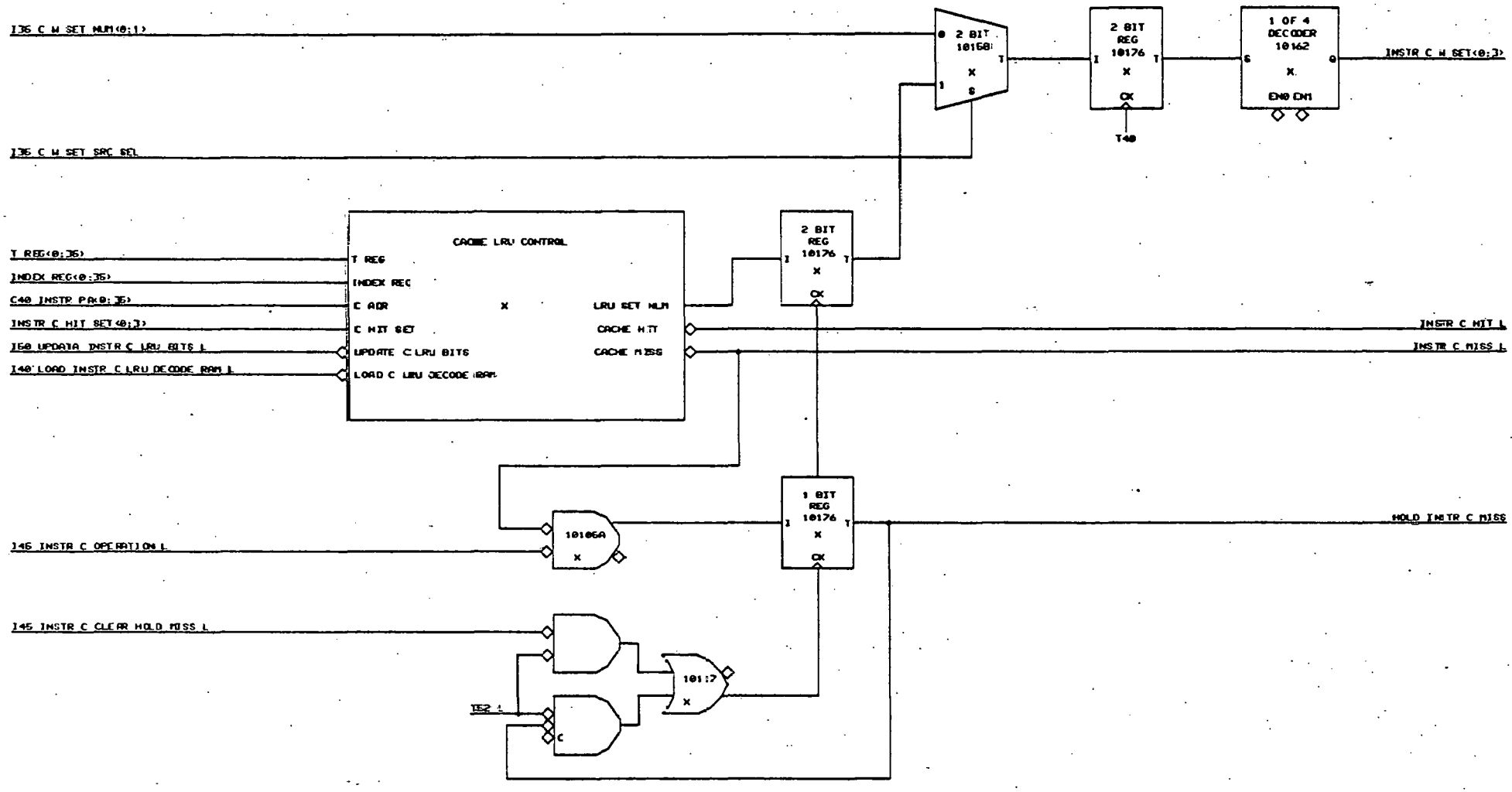
The Instruction Cache Memory Module implements one set of the instruction cache. Since words are always transfered between the cache and memory four at a time (called a line), the high order address bits only need be stored in the cache for every fourth word. The two 8B x 256W RAMs are used to store the high order 16 bits of the physical address for a line. The 18B x 1K and 19B x 1K RAMs store the data words plus parity. The 1B x 256W RAM stores the parity bit for the physical addresses.



INSTR Cache Memory Module (IMEMMD)

4.1.2.5.2 Instruction Cache Control

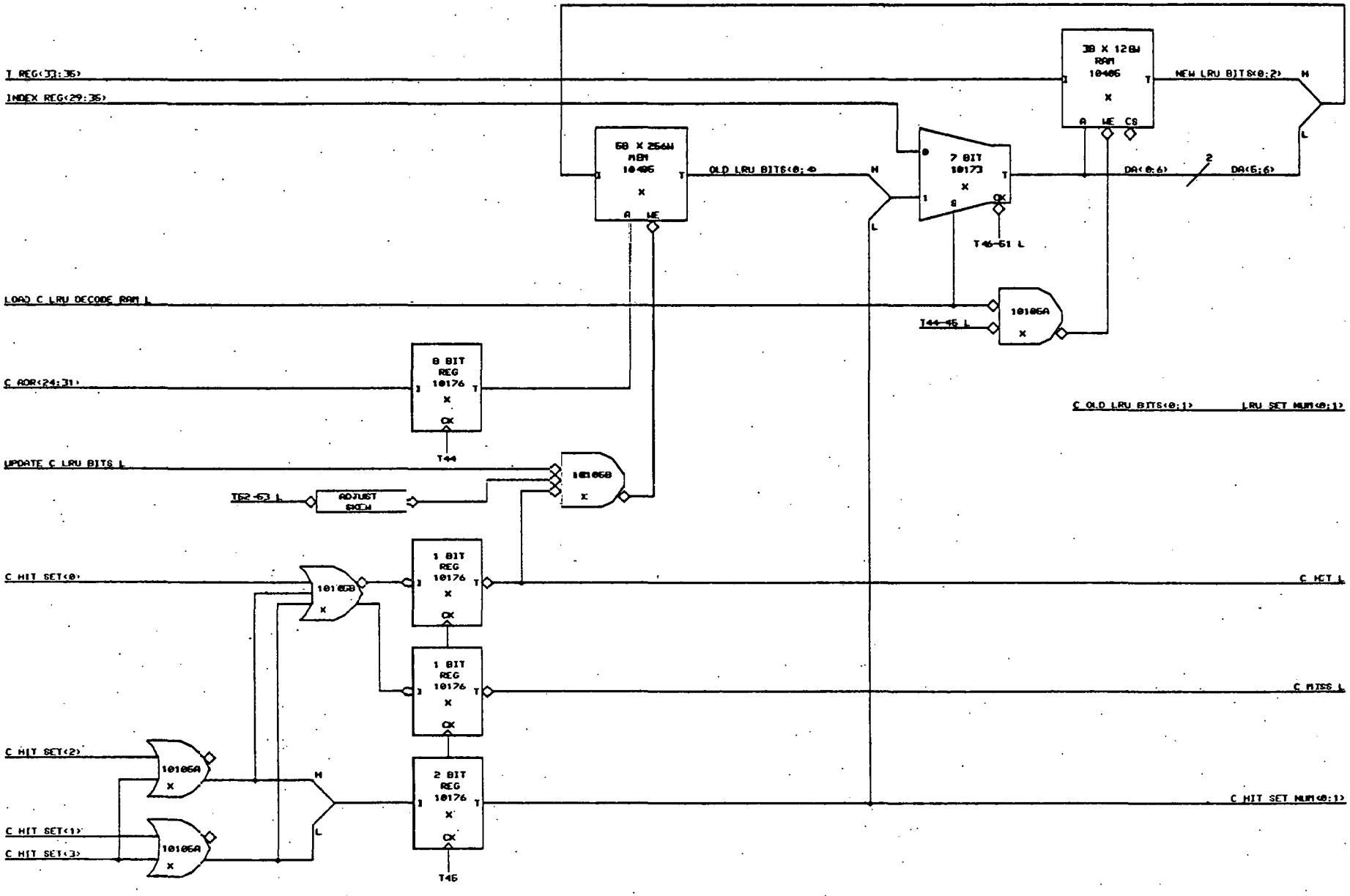
The Instruction Cache Control asserts the signal HOLD INSTR C MISS if an instruction cache miss occurs. It also selects which set is to be written into on a cache miss.



Instruction Cache Control (ICACC)

4.1.2.5.2.1 Cache LRU Control

The Cache LRU Control is almost identical to the Address Translation LRU Control, with the only mail difference being that it has to keep track of 1024 lines, instead of 64.

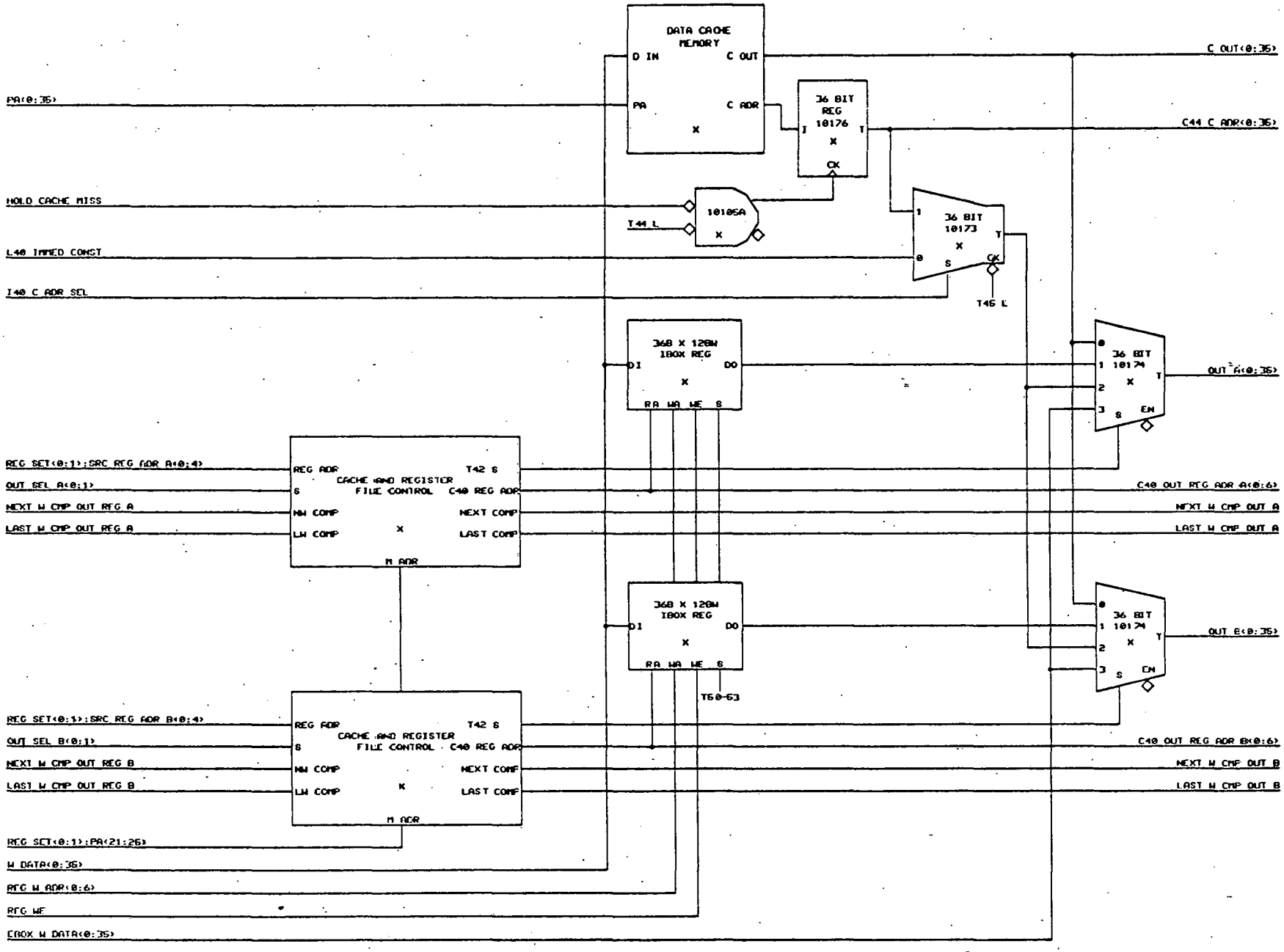


Cache LRU Control (CLRUC)

4.1.2.6 Data Cache and Register File

The Data Cache and Register File contains a cache memory for data, plus two register files, which both contain the four processor register sets. The outputs OUT A and OUT B are perfectly symmetrical, and both can read a cache location, a register, or an immediate constant. If a register is addressed as memory, then if the word was being read out of OUT A, the one register file will be used to read the register, otherwise the other register file will be used.

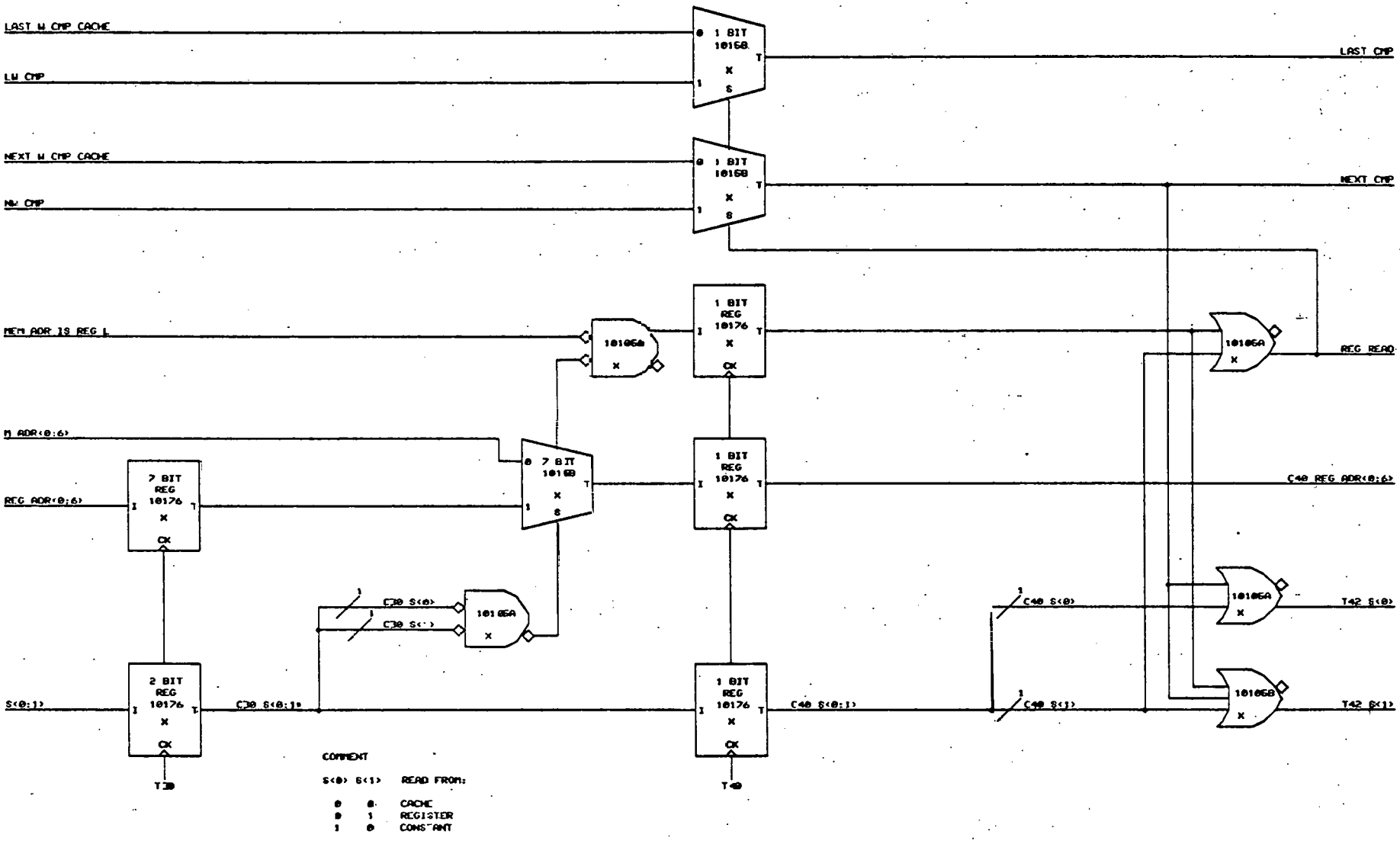
The EBOX has two operand registers, OP A and OP B. When the I-sequencer is calculating an operand to be put in OP A, it normally uses OUT A, and if it is calculating an operand for OP B, it used OUT B. The P-sequencer can then read a register operand on the other output, allowing two operands to be read per micro-cycle, with no conflict in the data paths being used.



Data Cache and Register File (CRFILE)

4.1.2.6.1 Cache and Register File Control

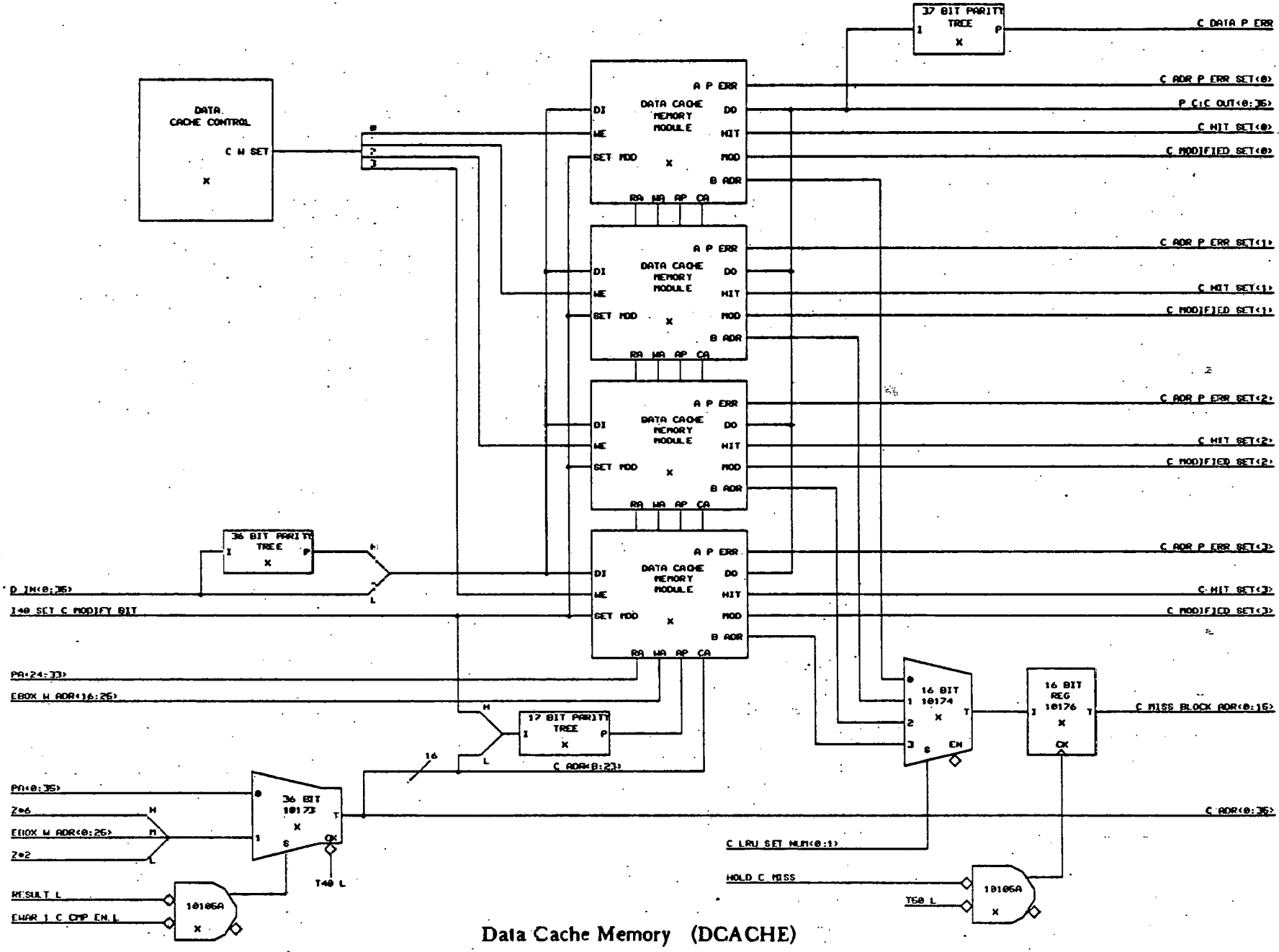
There are two Cache and Register File Controls, one for OUT A and one for OUT B. They control the output multiplexers to take care of when a register is read as a memory location, and which write compares happen the result of one instruction is used by one of the following instructions, which cause the W DATA bus to be selected on the output.



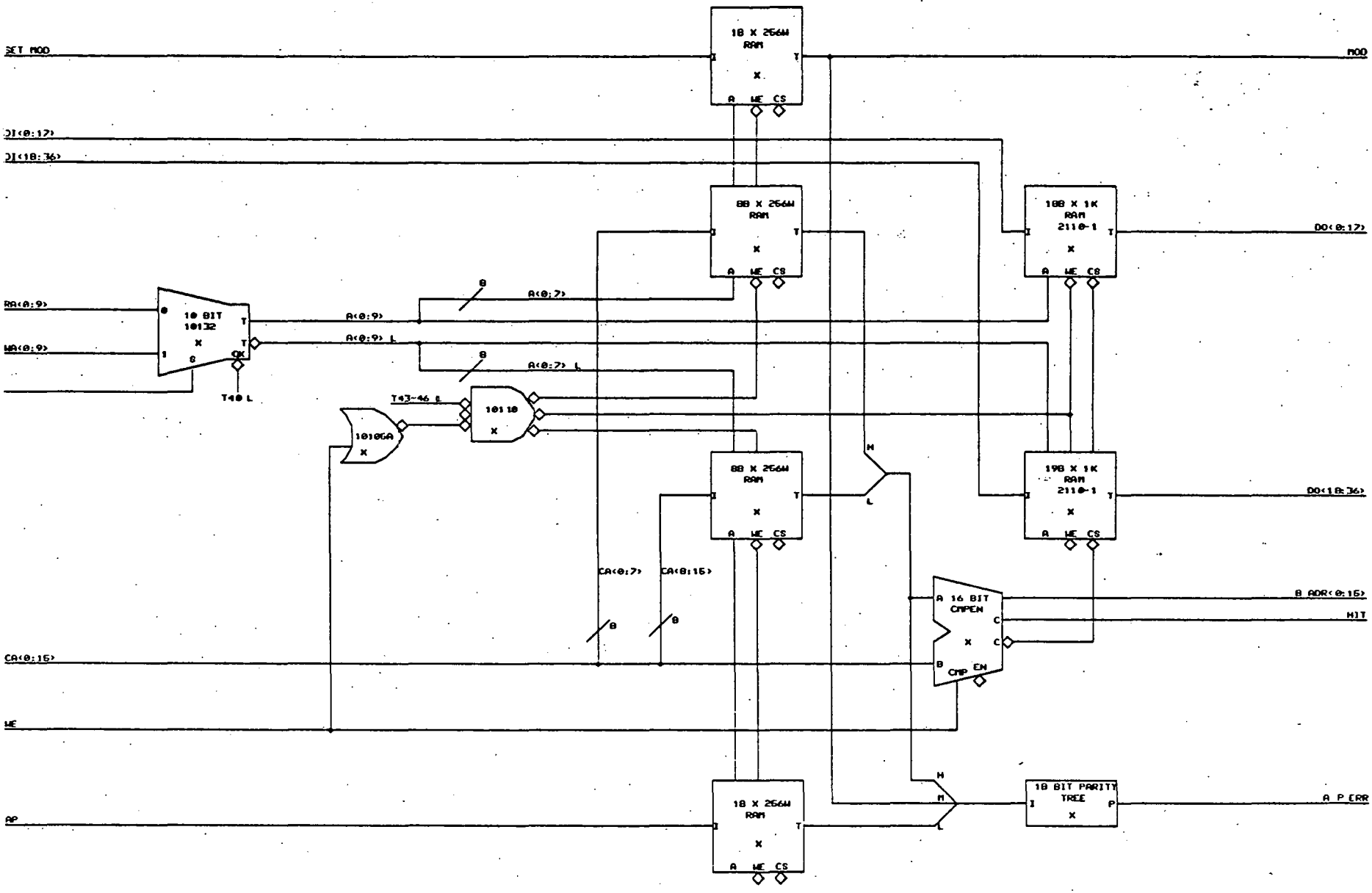
Cache and Register File Control (CRFCTL)

4.1.2.6.2 Data Cache Memory

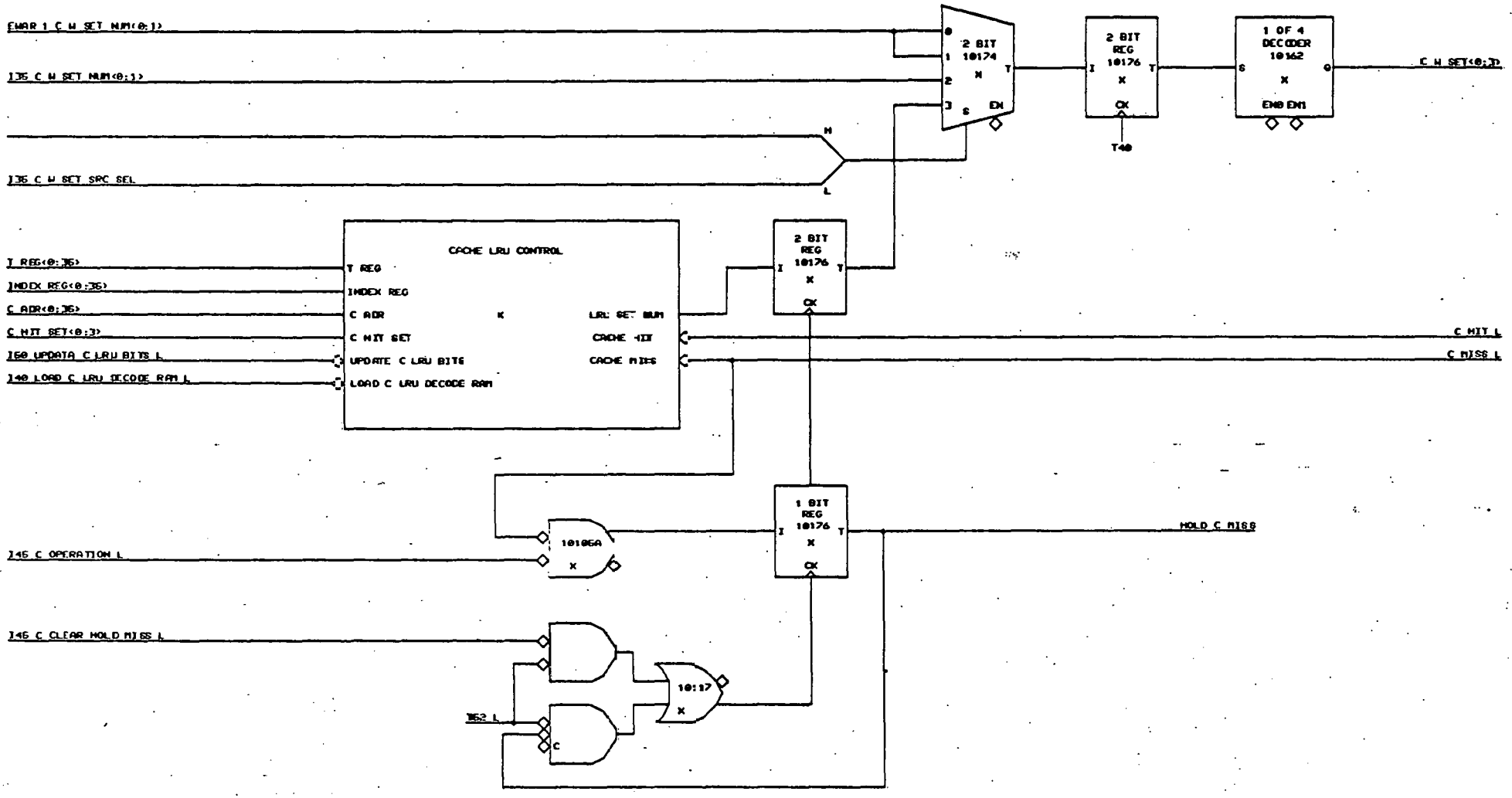
The Data Cache Memory is very similar to the instruction cache, with the main difference being that a bit is stored for each line in the cache, indicating that it has been written into.



Data Cache Memory (DCACHE)



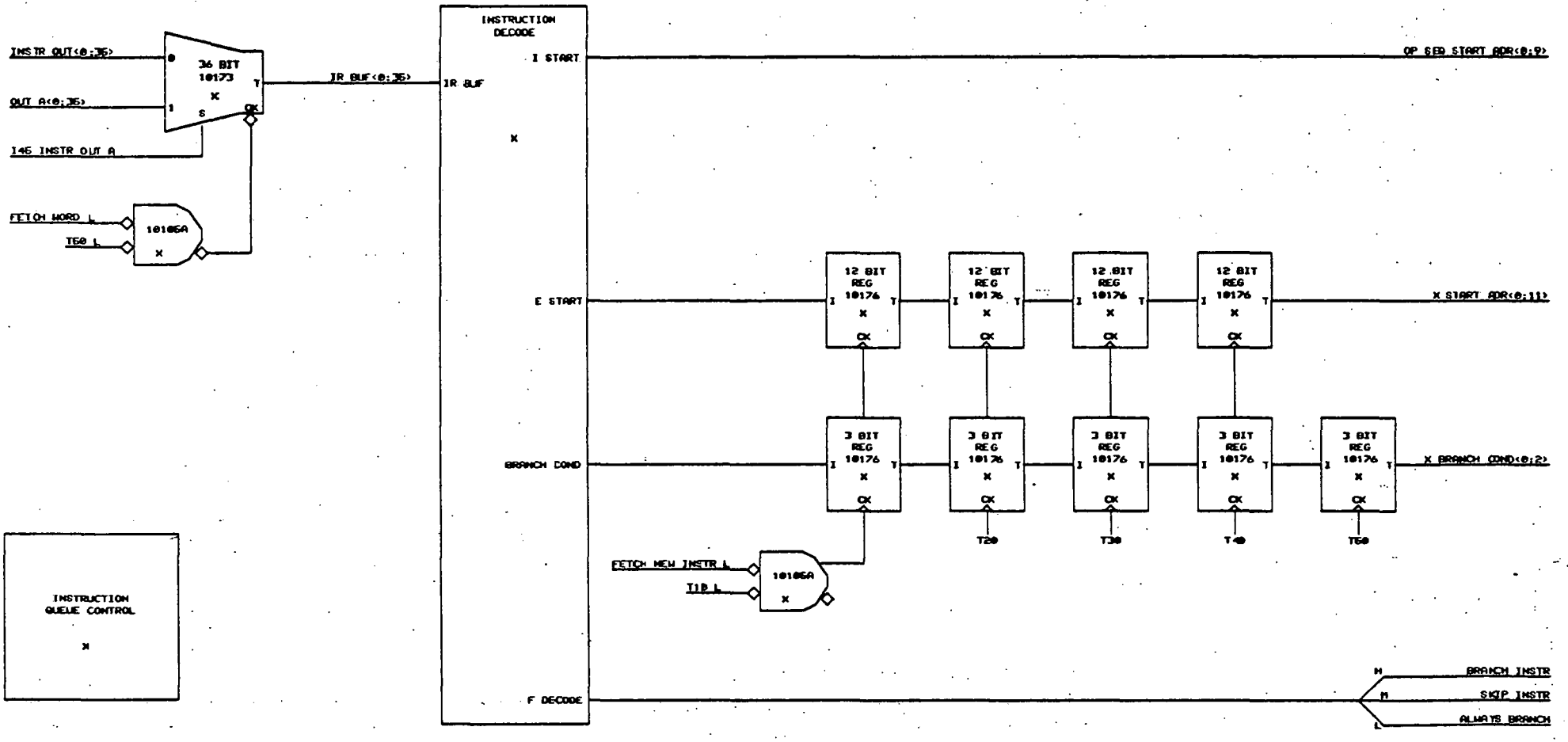
Data Cache Memory Module (MEMMOD)



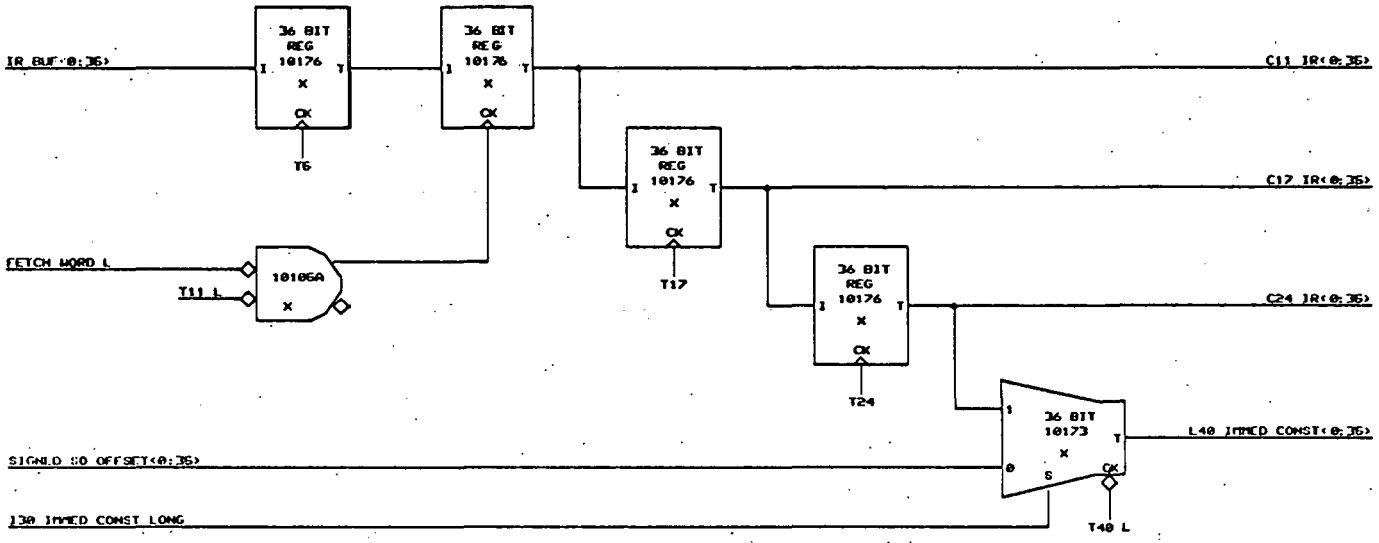
Data Cache Control (DCACC)

4.1.2.7 Instruction Buffer and Decode

The Instruction Buffer and Decode buffers instructions so that they are around during later cycles in the pipeline, and decodes them, to find out the starting address in the P-sequencer and the E-sequencer. It also recognizes branch and skip instructions for the prefetch logic.



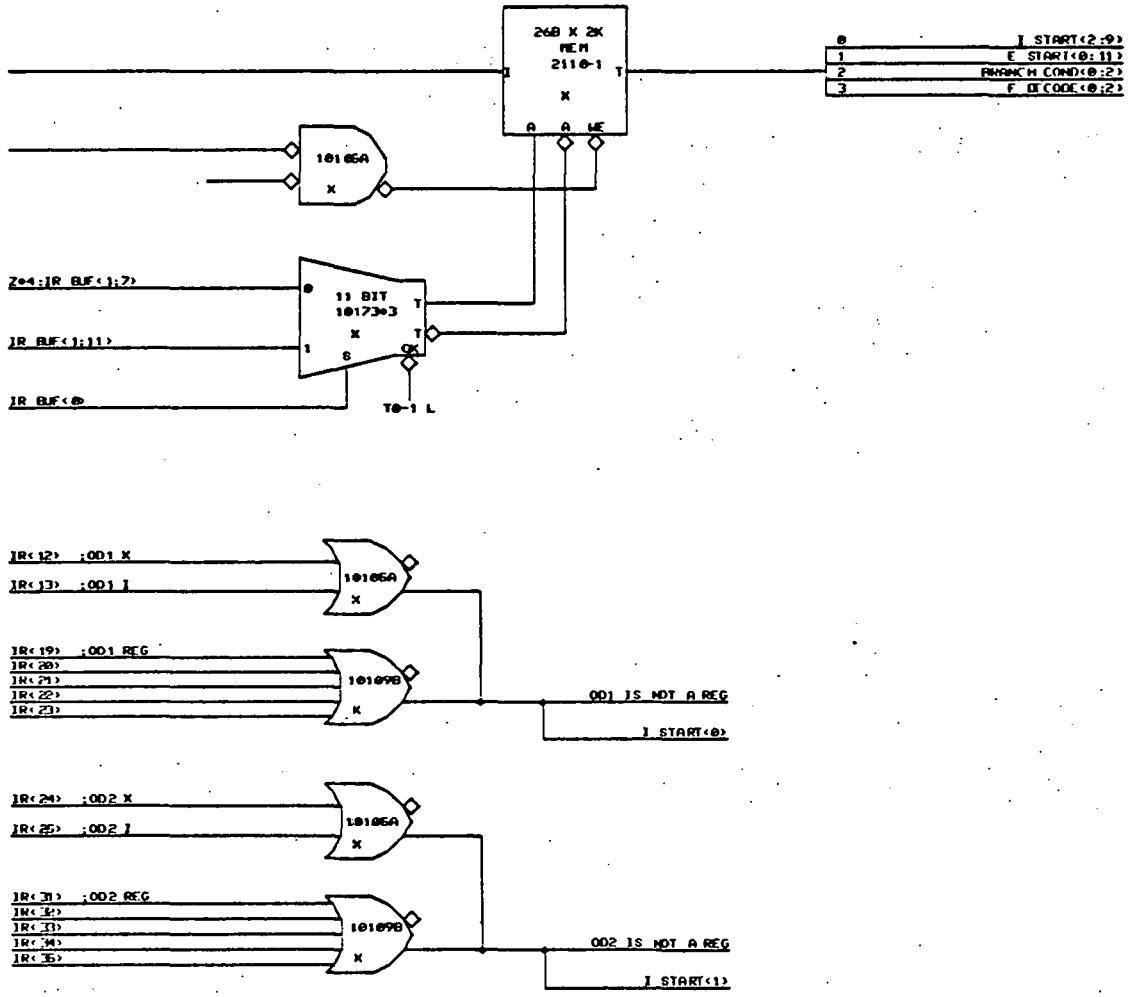
Instruction Buffer and Decode 1/2 (INBD1)



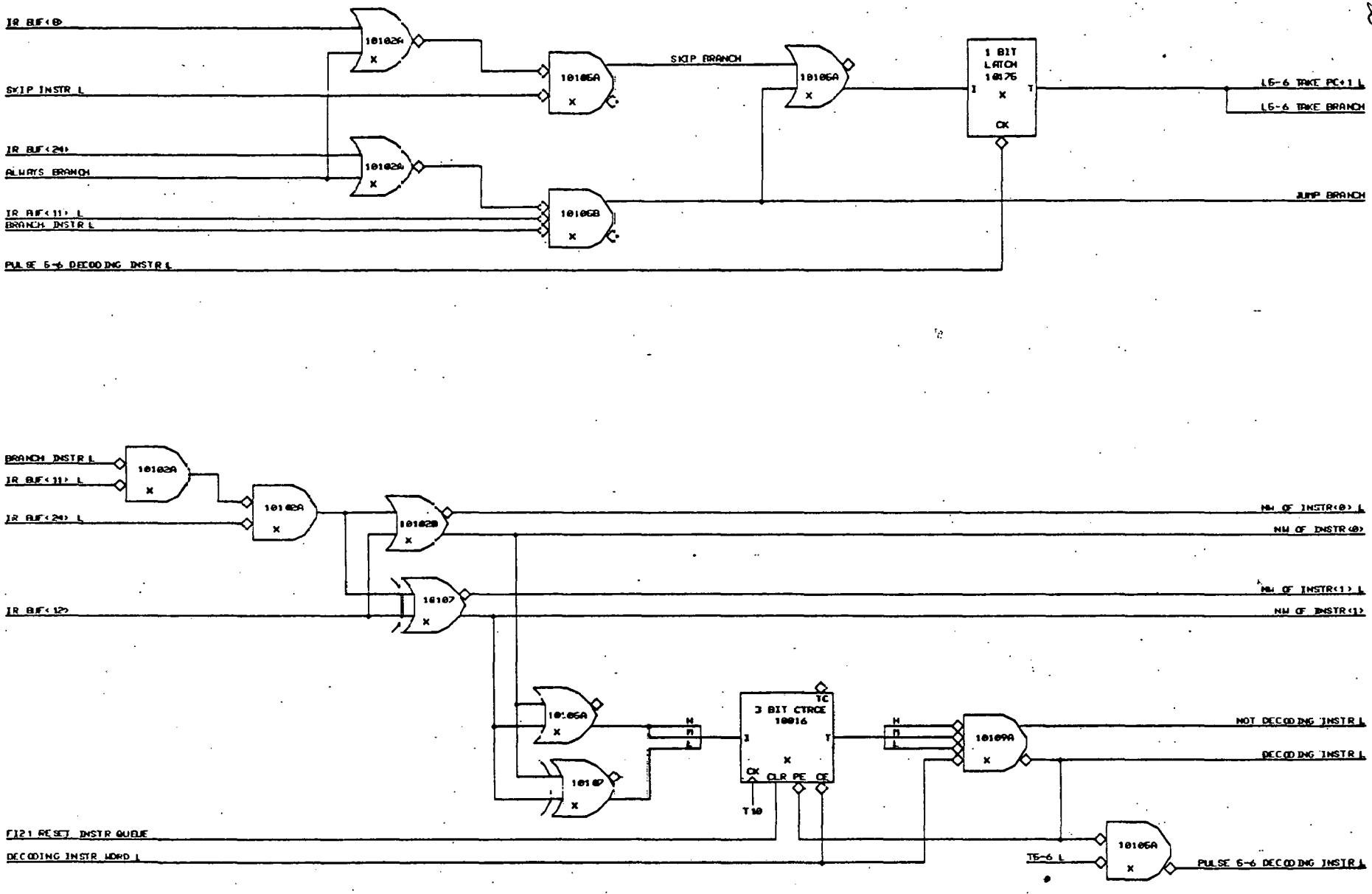
Instruction Buffer and Decode 2/2 (INBD2)

4.1.2.7.1 Instruction Decode

The Instruction Decode logic decodes instructions mainly by looking the opcode up in a 28B x 2K word RAM. The top bit of the opcode is used to tell if the instruction is a skip instruction or not, meaning that exactly half of all of the opcodes will always be skips. If it is a skip instruction, then Z*4:IR BUF<1:7> is fed into the decode RAM, otherwise IR BUF<1:11> is put into it. This also means that 128 of the non-skip opcodes are unusable, but this seemed a reasonable price to pay for being able to use a 2K decode RAM, rather than a 4K one.



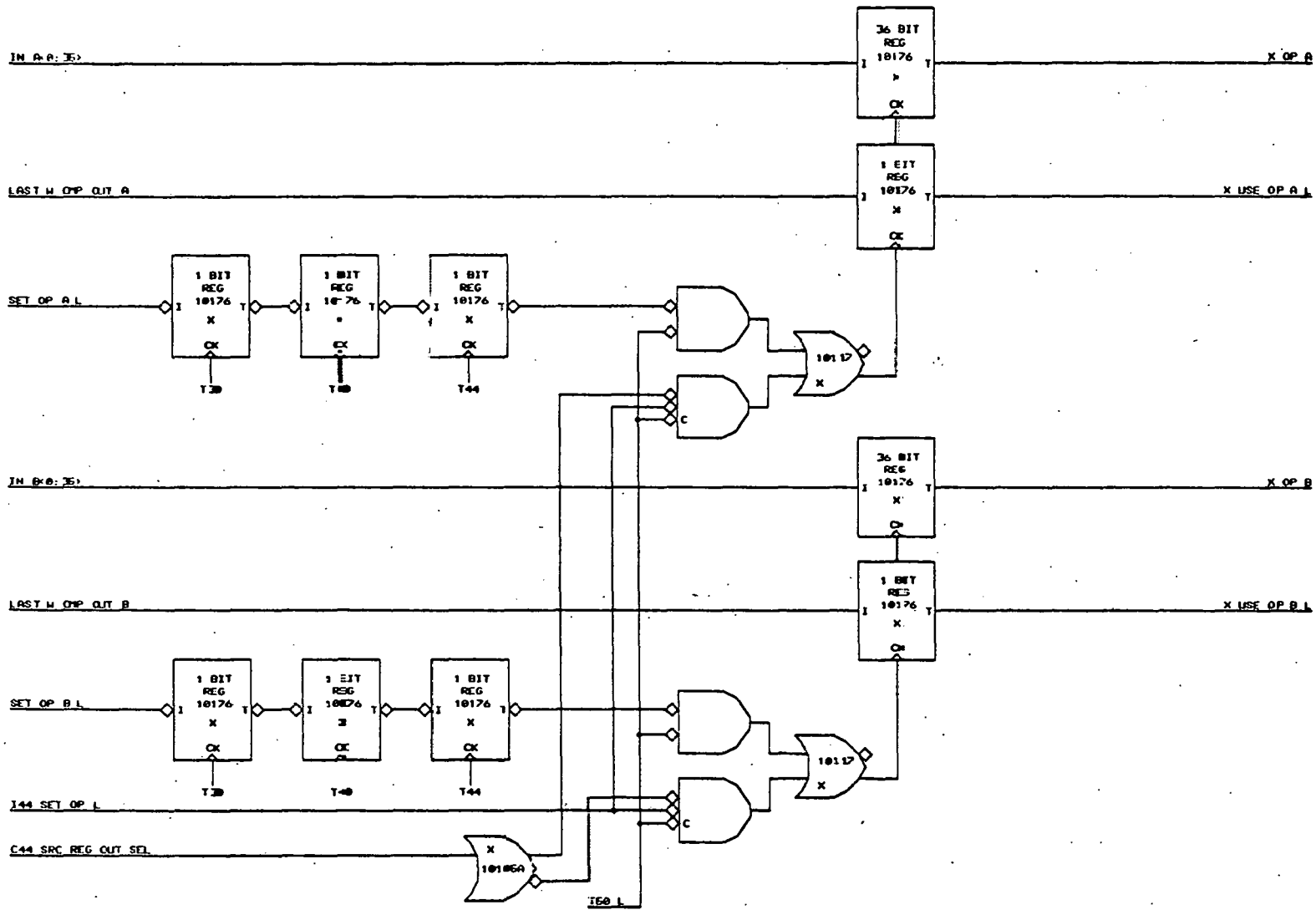
Instruction Decode 1/2 (DECOD1)



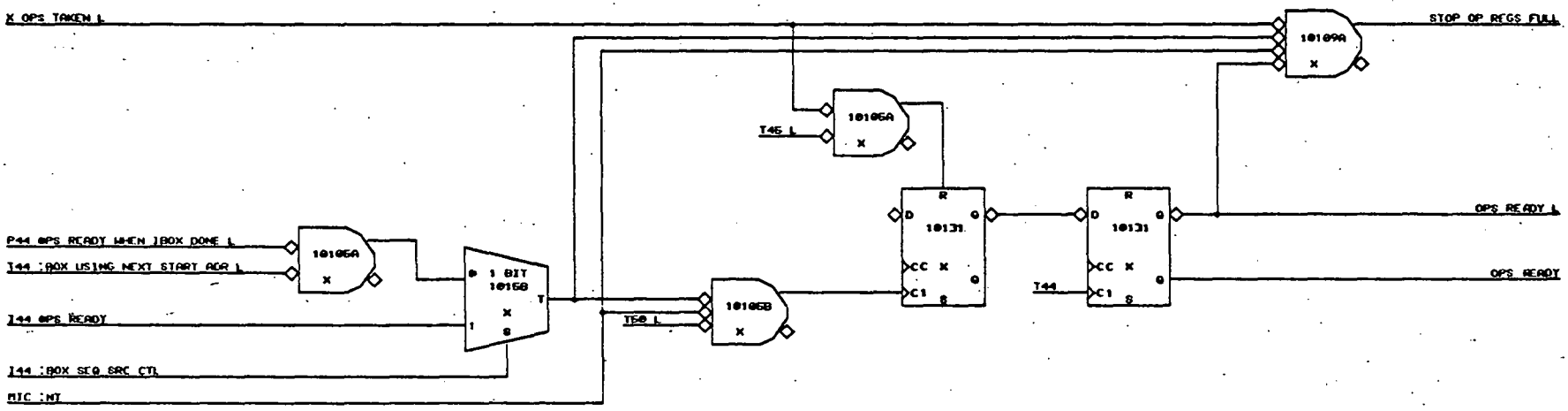
Instruction Decode 2/2 (DECOD2)

4.1.2.8 EBOX Operand Registers

The EBOX Operand Registers are used to hold the next set of operands for the EBOX. If the IBOX gets further than two operands ahead of the EBOX in fetching instructions and operands, then it stops and waits. If the EBOX is done with a given instruction, and the operands for the next instruction are not ready, then it waits. The EBOX Operands Ready Control keeps track of when operands are ready, and when the EBOX takes them.



EBOX Operand Registers (EOPREG)

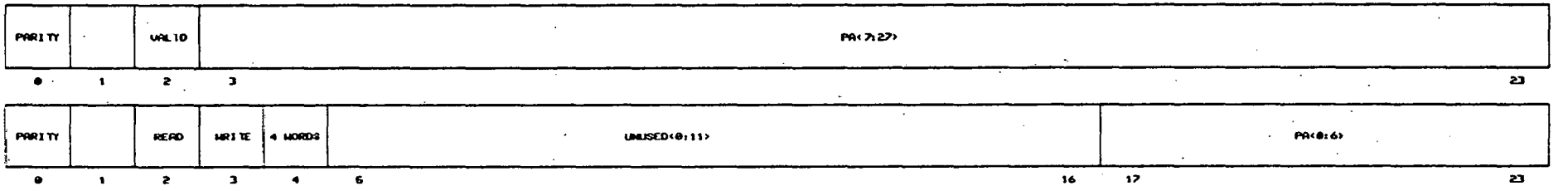


EBOX Operands Ready Control (EOPRDY)

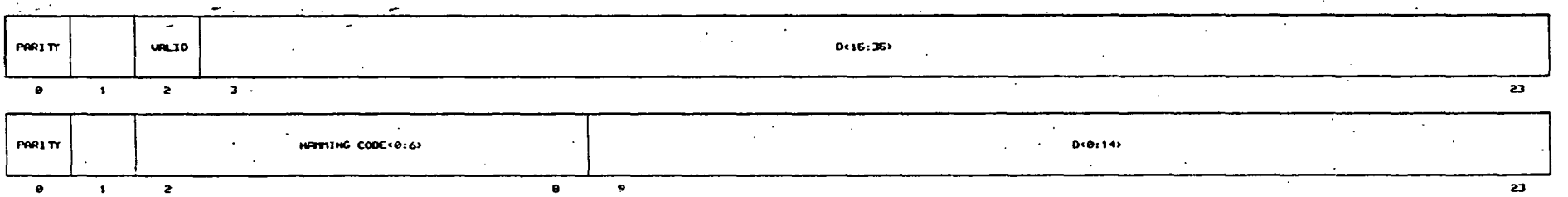
4.1.2.9 Memory Interface

The Memory Interface controls communication between the processor and the switch, and takes care of generating hamming codes, and correcting errors. The format for the switch control words is shown in drawing IOFORM. An I/O operation is started by sending an I/O control word to the switch, which specifies a memory address, whether it is a read, write, or both (a read-modify-write operation), and whether 1 or 4 words are to be transfered. If it is a read operation, the processor just sits and waits for the data to come back. On writes, the processor waits until the switch sends a control word back with its VALID bit set, which signals that the processor has a direct path to memory opened, and to start sending data.

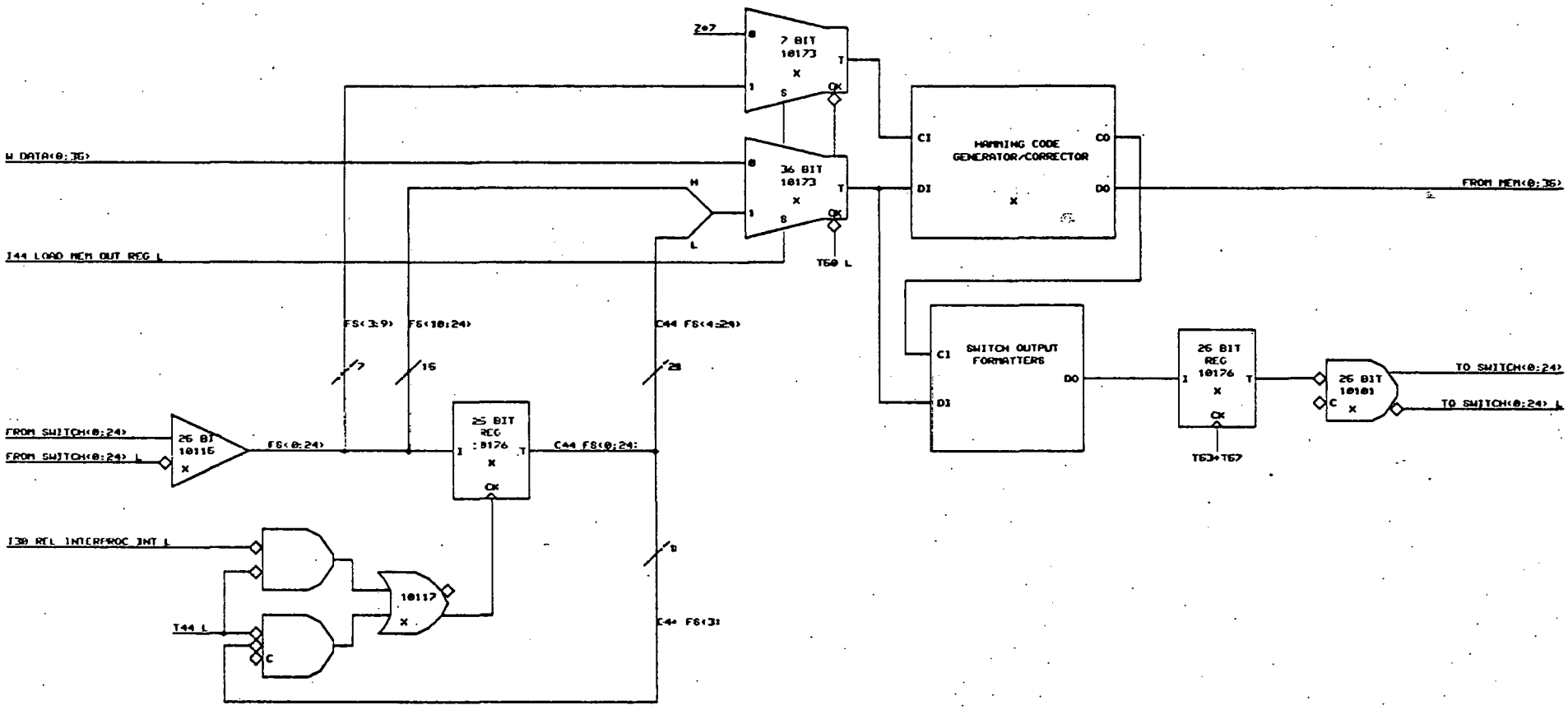
DATA CONTROL WORD



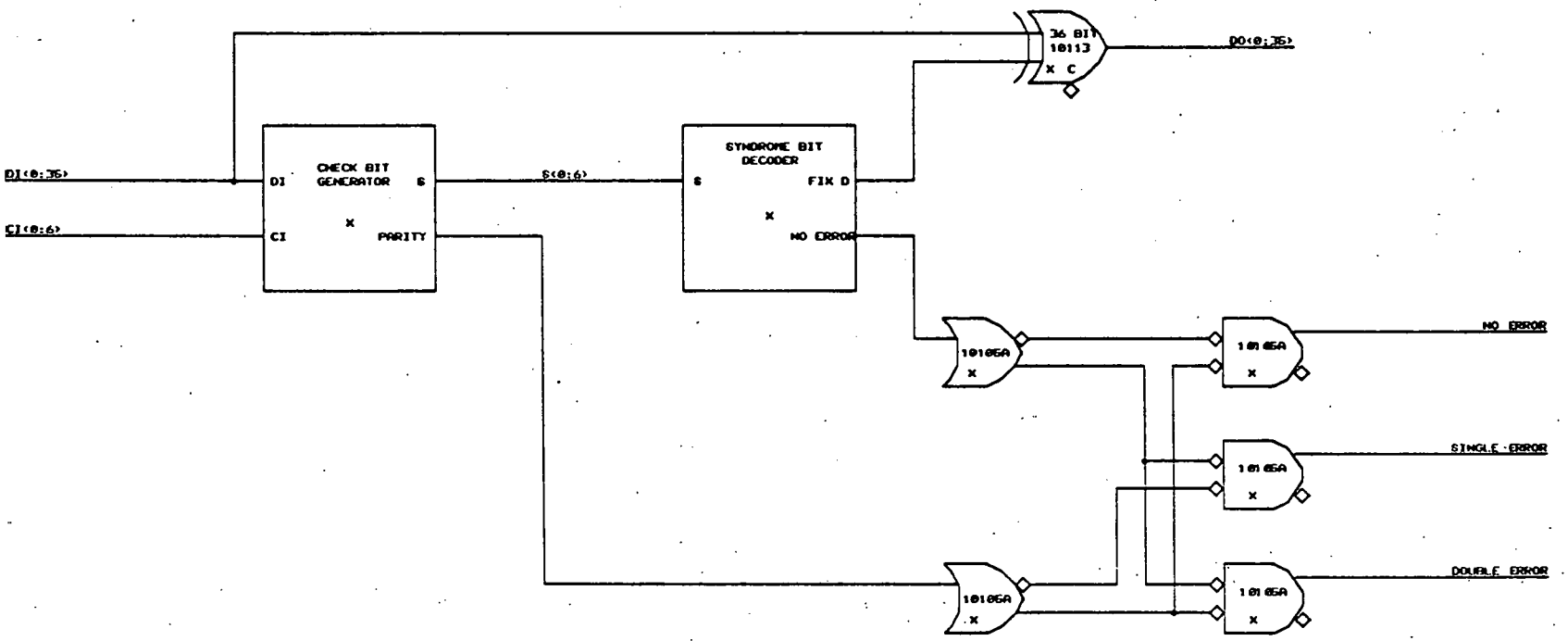
DATA WORD



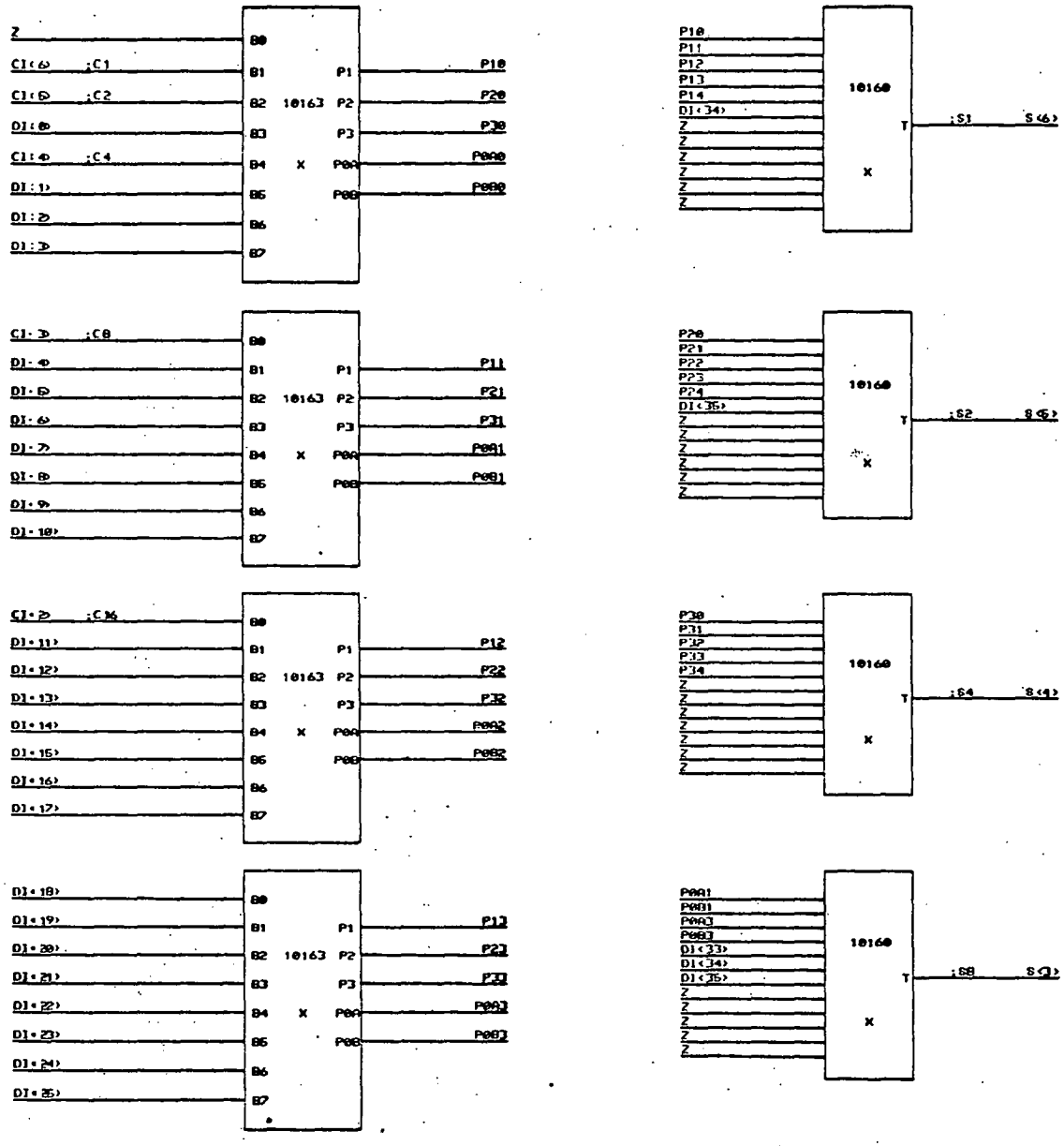
Switch Control Word Formats (IOFORM)



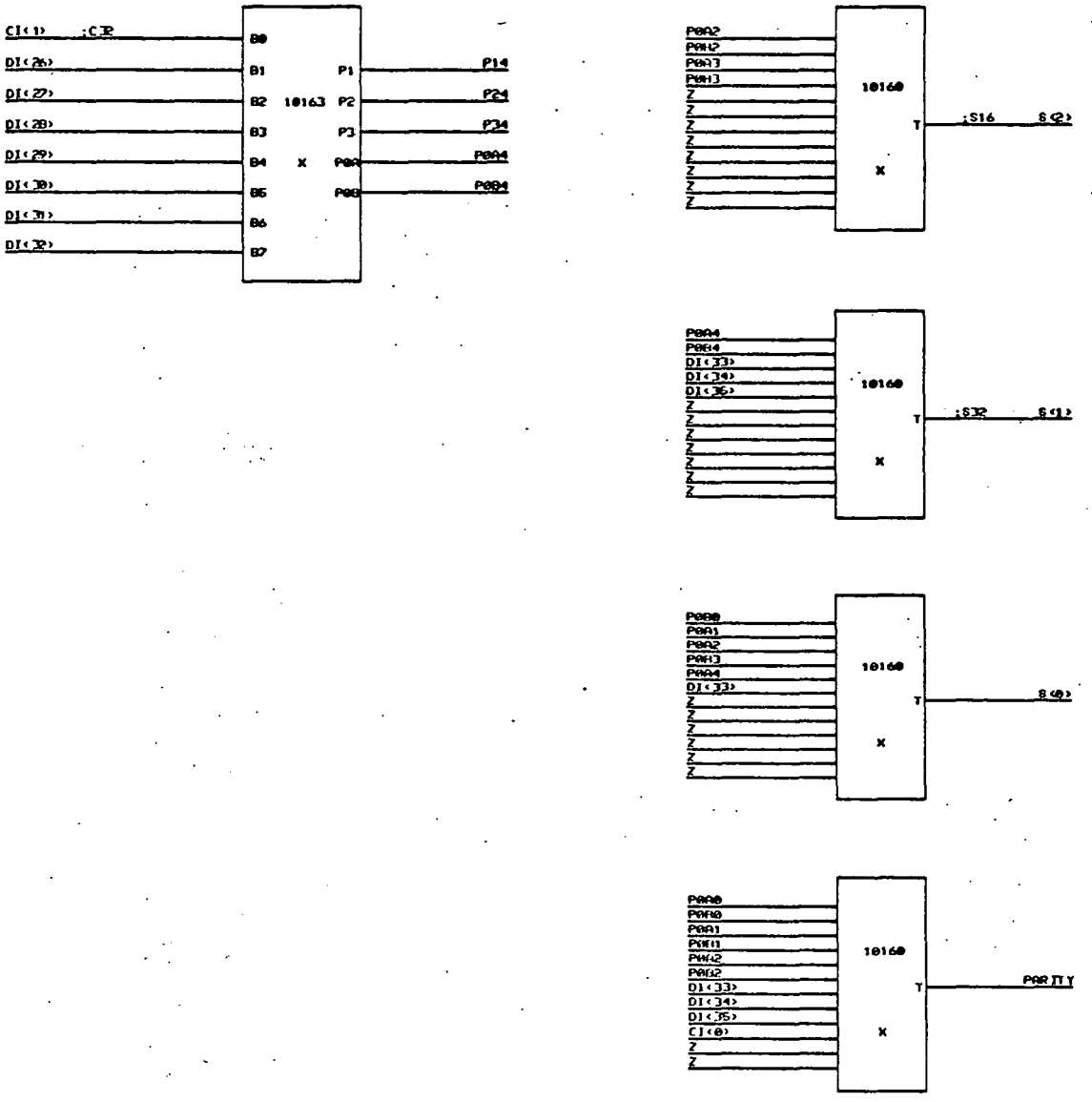
Memory Interface (MFACE)



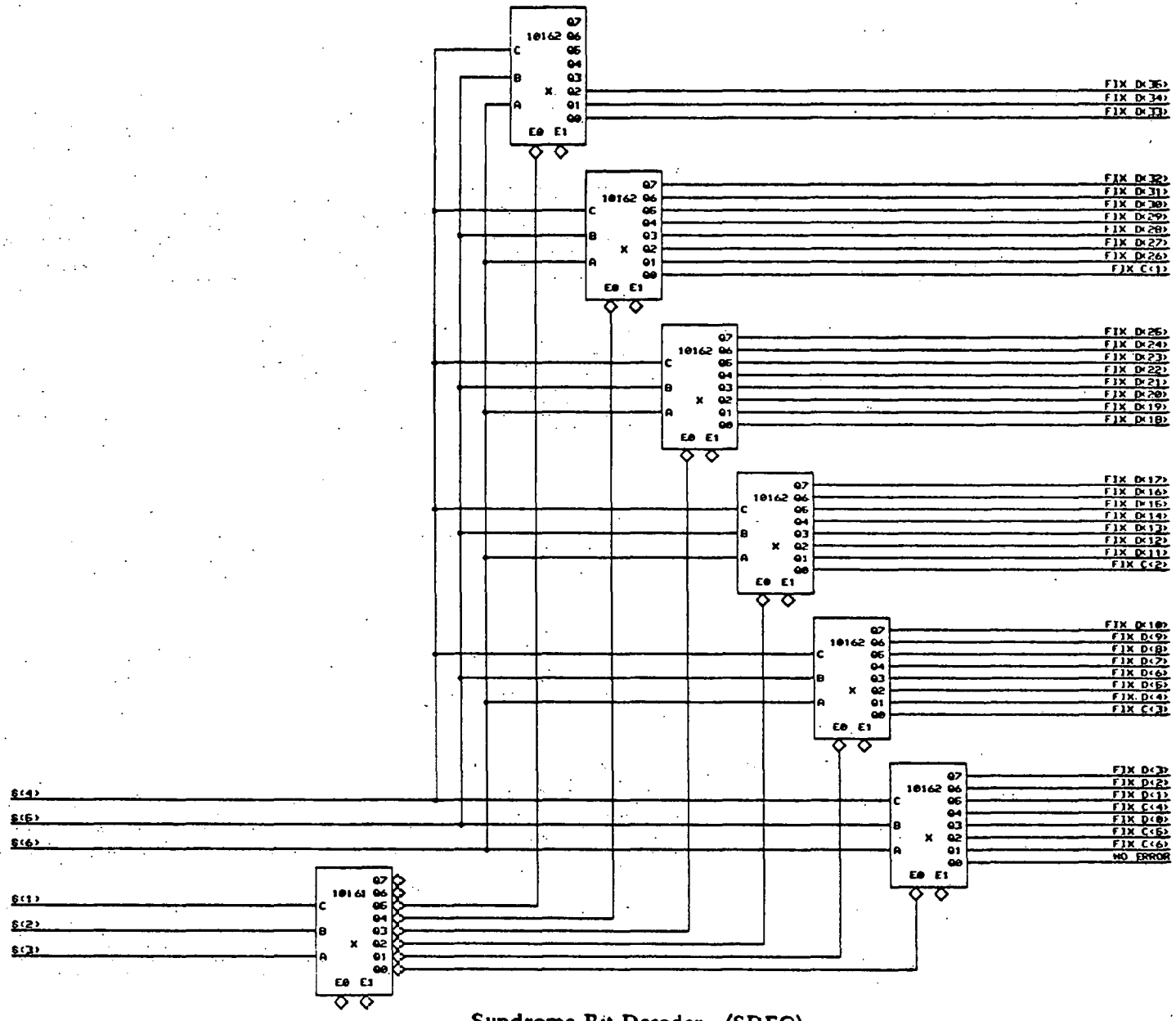
Hamming Code Generator/Corrector (HAMCOD)



Check Bit Generator 1/2 (CHKGN1)



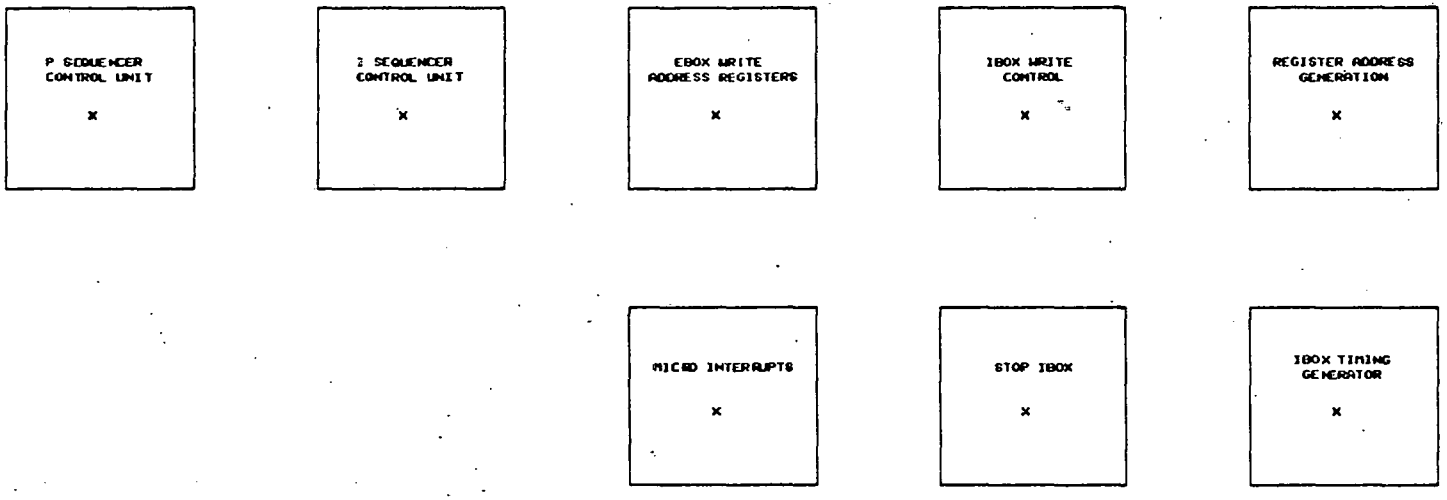
Check Bit Generator 2/2 (CHKGN2)



Syndrome Bit Decoder (SDEC)

4.1.2.10 IBOX Control

The major sections of the IBOX Control are shown in drawing IBOXC. The following sections will go into detail about what each of these sections do. In addition to these sections, there is a section which gives the flow of control of the prefetch logic.

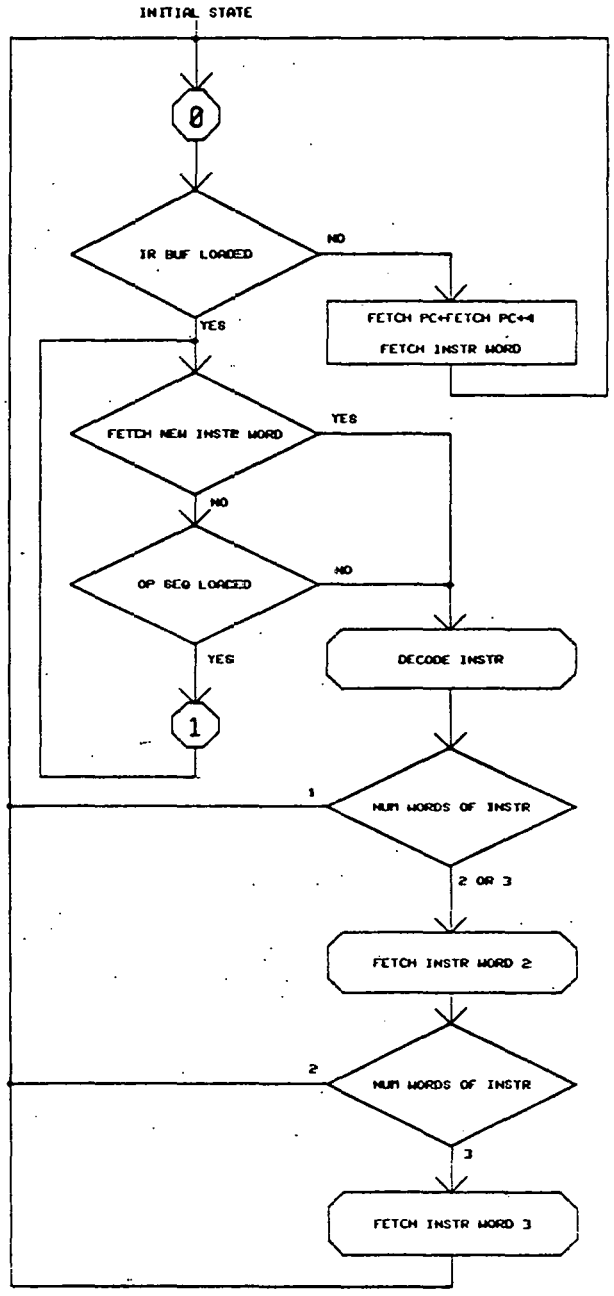


IBOX Control (IBOXC)

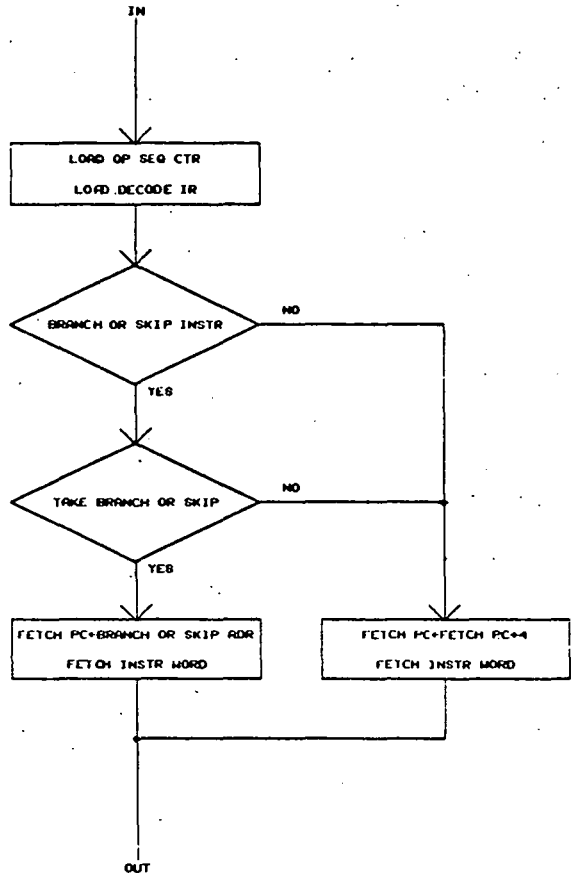
4.1.2.10.1 Instruction Prefetch Control

In addition to the three micro-sequencers in the machine, there is a hardware control unit called the Instruction Prefetch Control, which keeps fetching instructions ahead of the P-sequencer, in order to keep the pipeline full. The basic flow of control is shown in drawing FLOWF1.

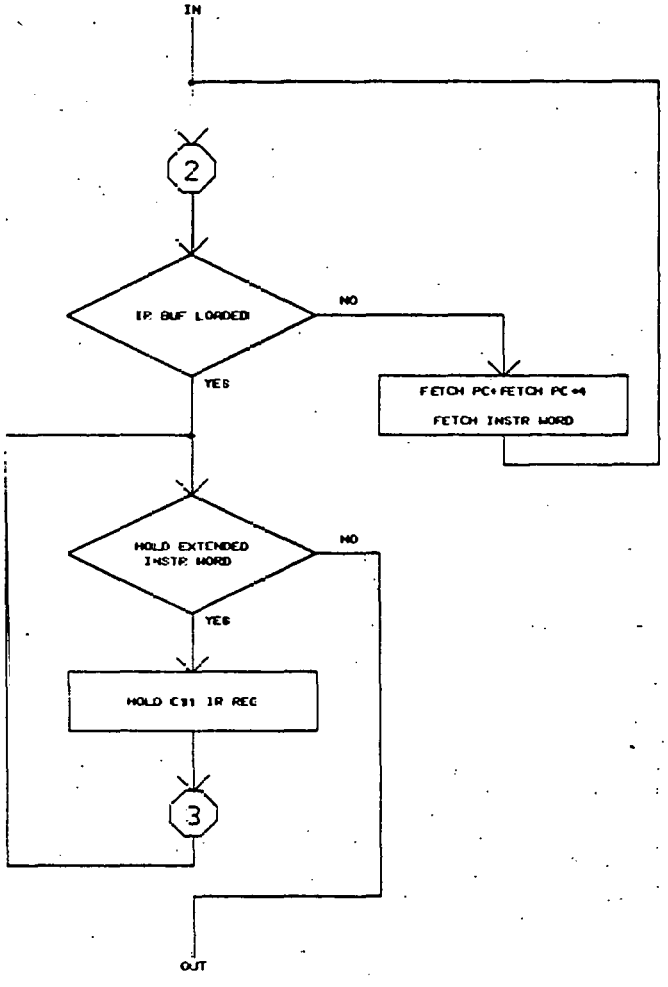
The sequencer goes from one state to the next every micro-cycle, where states are represented by octagons, with the state number shown inside. The rectangular boxes represent actions to be preformed, and the diamonds represent conditionals. The rectangular boxes with cut off corners represent macro calls to the macros defined in drawings FLOWF2, FLOWF3, and FLOWF4.



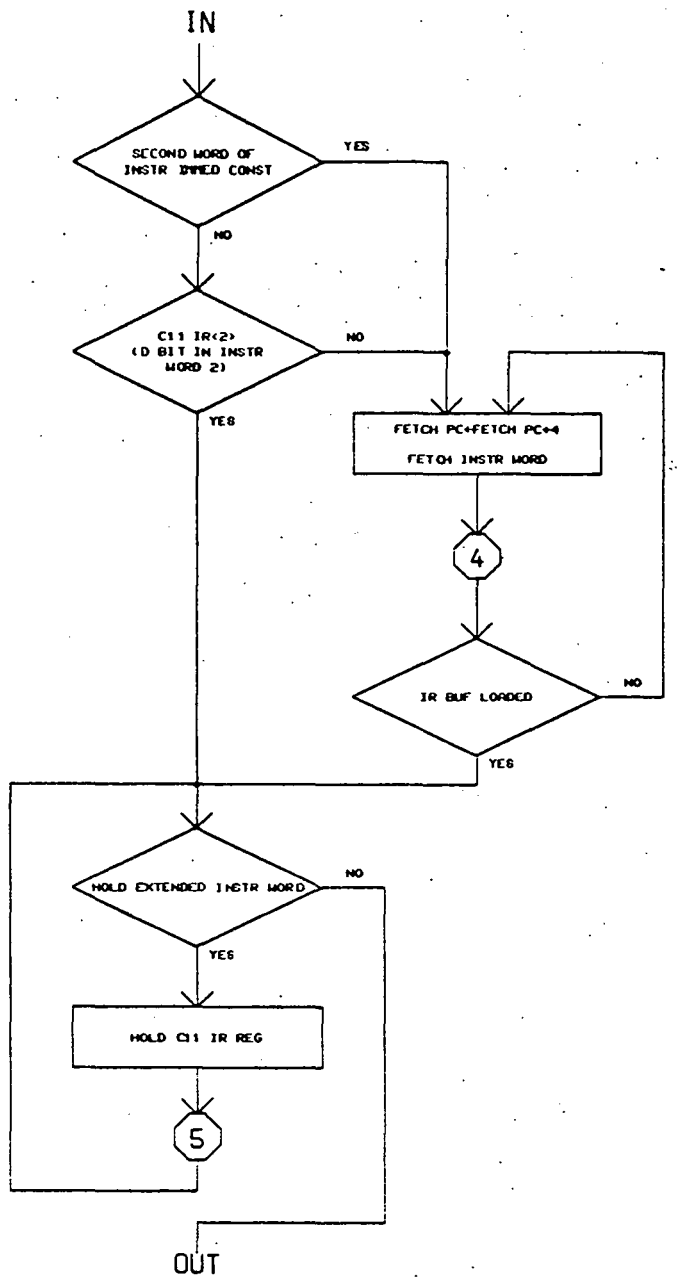
Instruction Prefetch Control (FLOWF1)



Decode INSTR (FLOWF2)



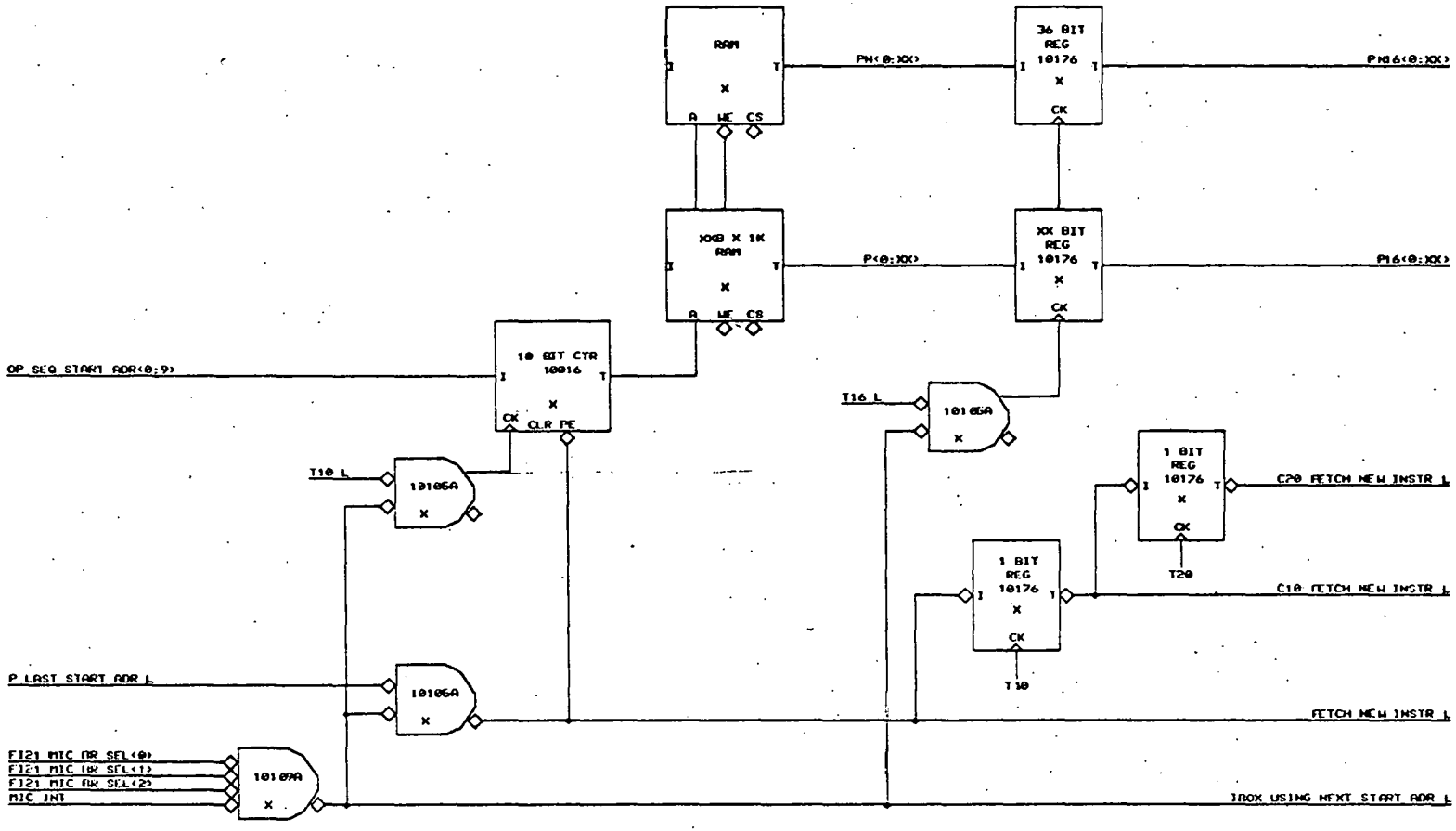
Fetch INSTR Word 2 (FLOWF3)



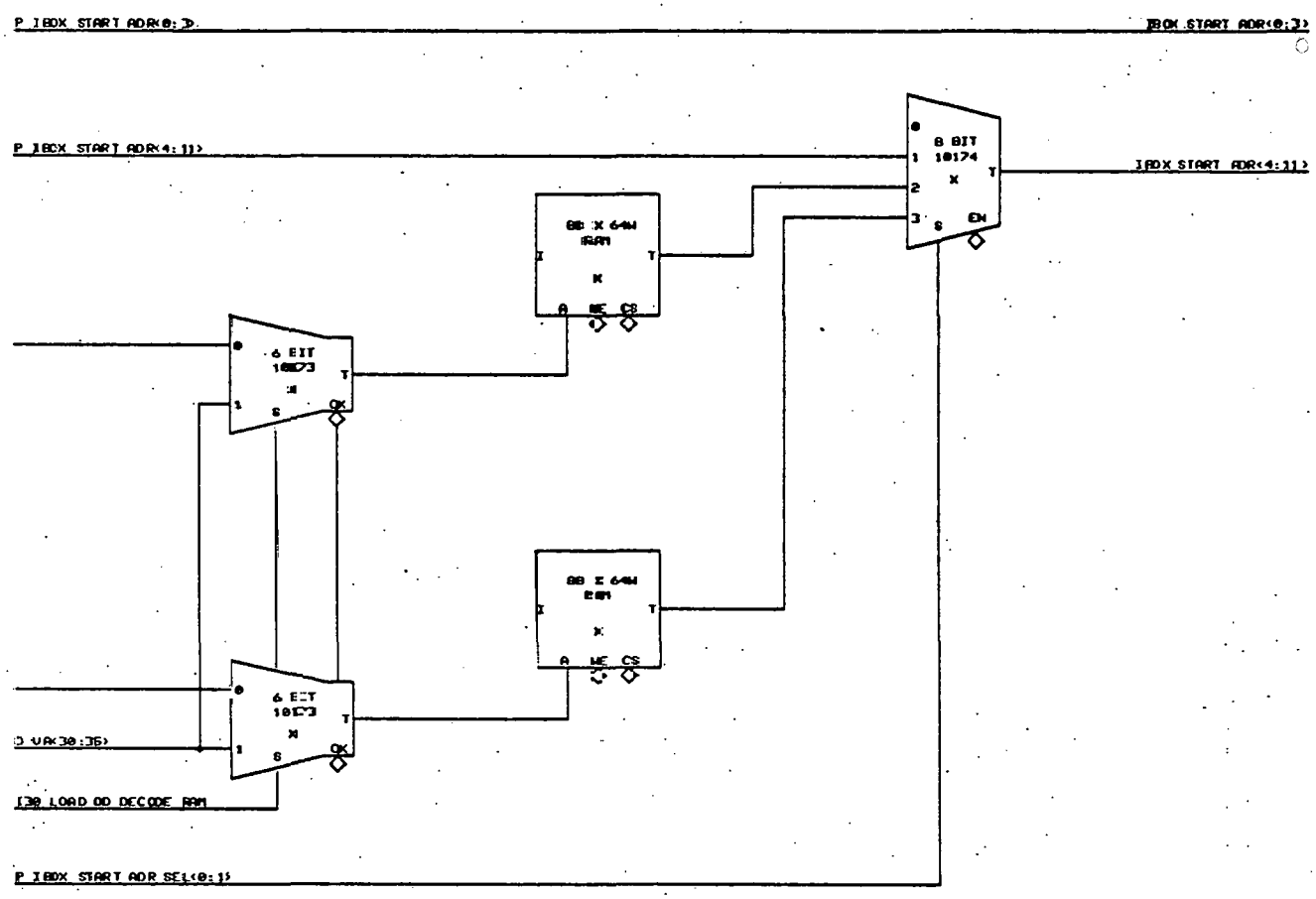
Fetch INSTR Word 3 (FLOWF4)

4.1.2.10.2 P-Sequencer Control Unit

The P-Sequencer Control Unit is started at address OP SEQ Start ADR<0:9>, which is generated by the decode RAM. For a given instruction, it can only execute sequential micro-instructions. Its main function is to take care of the difference between the many different formats for the operands of instructions, and to fetch all register operands, which the I-Sequencer fetches memory operands.



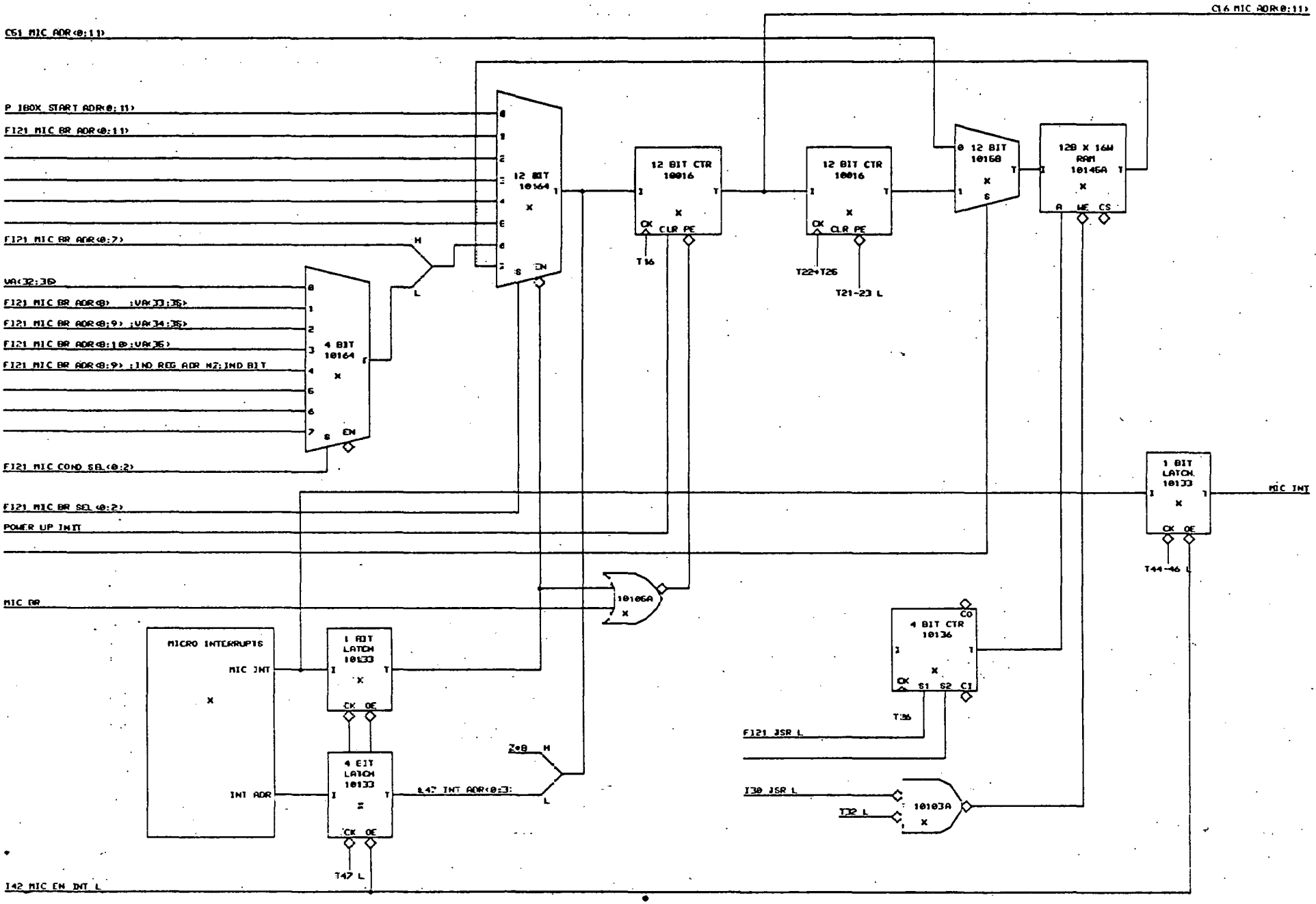
P Sequencer Control Unit 1/2 (PSEQ1)



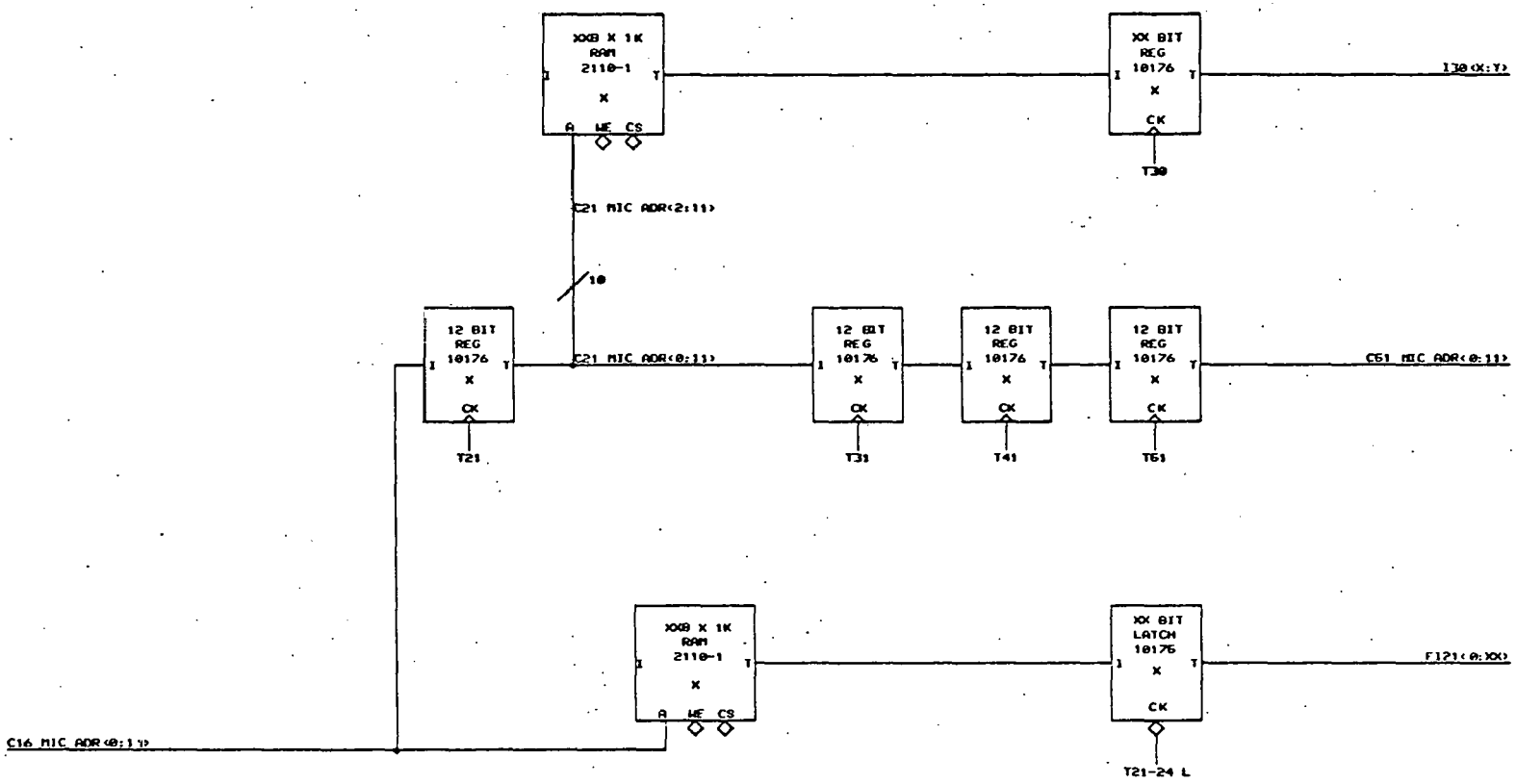
P Sequencer Control Unit 2/2 (PSEQ2)

4.1.2.10.3 I-Sequencer Control Unit

The I-Sequencer Control Unit is the main work horse in the IBOX, and is a powerful micro-programed controler. It can branch anywhere in its control store, can execute nested subroutines up to 16 levels deep, and can preform micro-interrupts, which stack their return address. The control store is divided into two parts, a fast and slow part. The only difference is the time at which the control bits come out. The fast signals are designated FI21 since they come out around time 21, and the slow signals are designated I30.



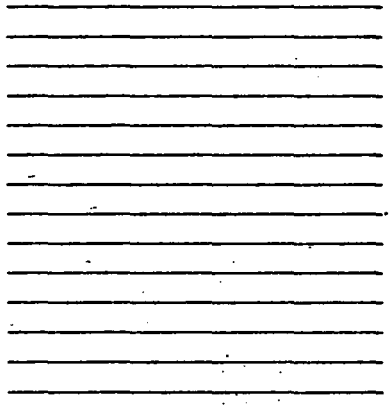
I Sequencer Control Unit 1/3 (ISEQ1)



I Sequencer Control Unit 2/3 (ISEQ2)

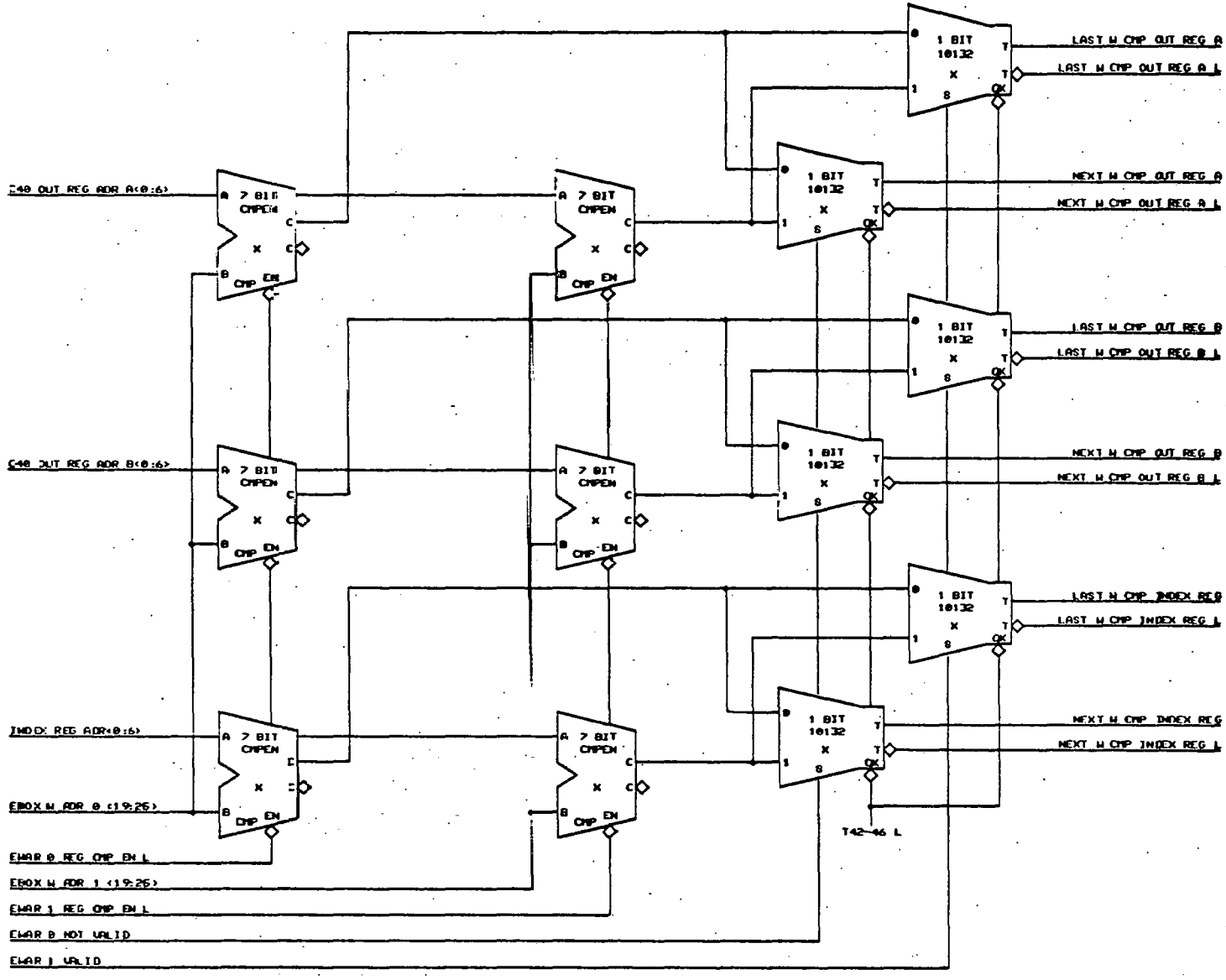
COMMENT

MICRO WORD BIT ASSIGNMENTS

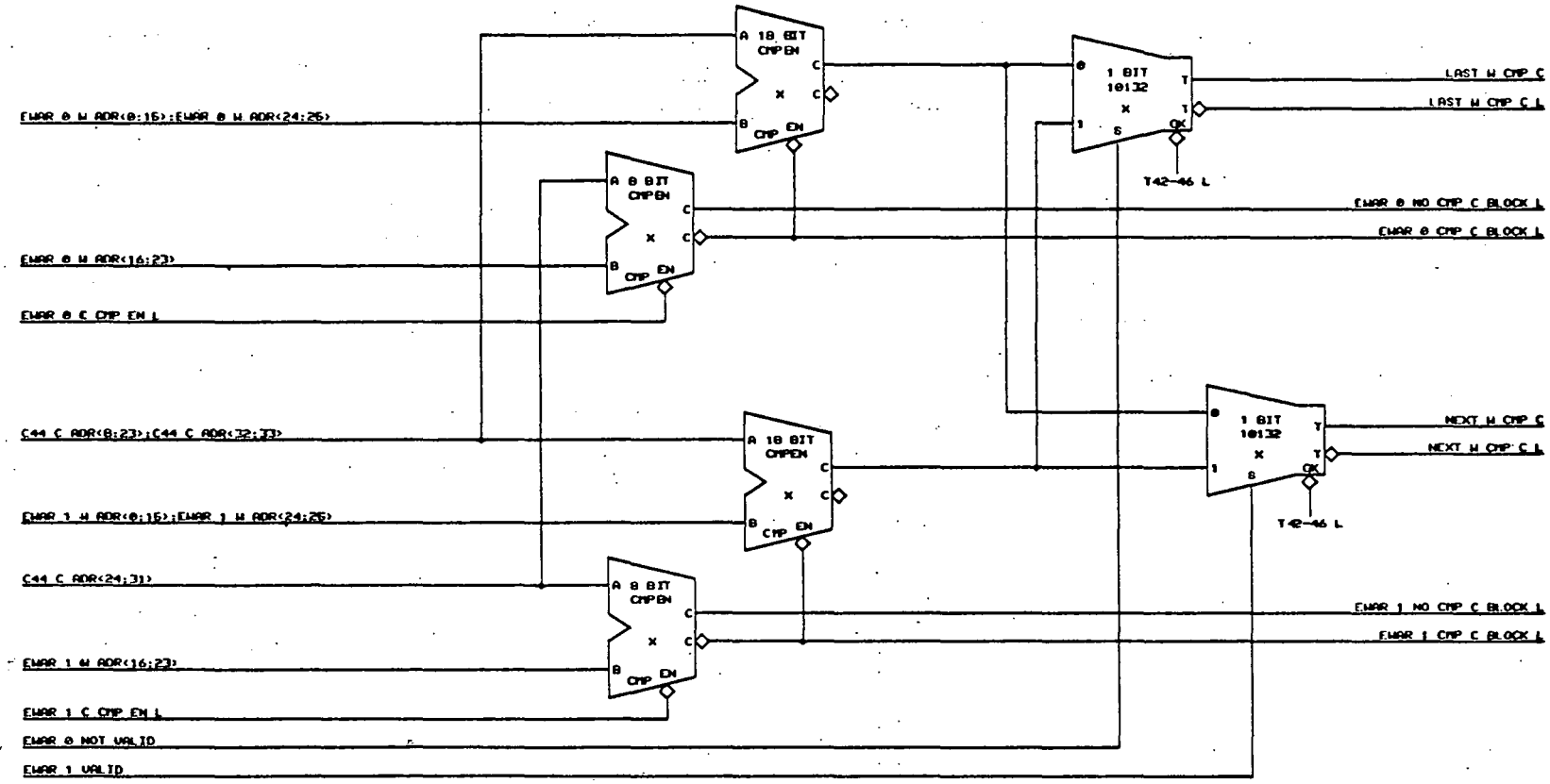


4.1.2.10.4 EBOX Write Address Registers

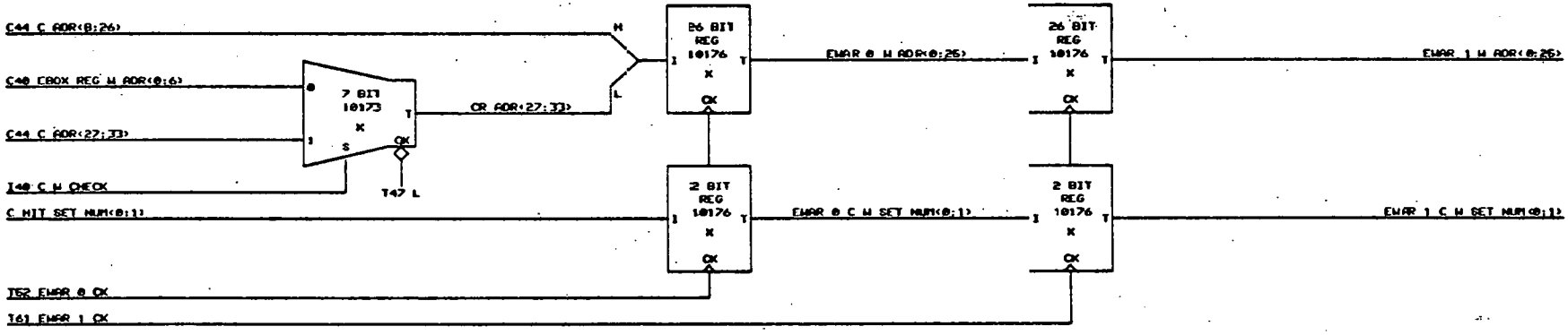
The EBOX Write Address Registers are used to keep track of pending writes from the EBOX into either the cache, the register file, or to memory. There are two write address registers, which allow the IBOX to schedule up to two writes ahead of the EBOX. If the IBOX tries to schedule a third write, then it is stoped until the EBOX does a write, freeing up one of the registers. It has a set of four comparators for each of its two write address registers, which compare the address of the words currently being read from the three register stacks and the cache, to the addresses which have pending writes. If one of the comparators compare, then signals are asserted which cause the IBOX to either wait for the write to occur, or to take the data directly from the output of the EBOX. For example, if the IBOX is reading an operand for one instruction, and it finds out that it is the result of the previous instruction, rather than reading the operand from memory or a register file, it sets a bit in the EBOX operand register saying for the EBOX to use the result of the previous instruction, rather than the contents of the EBOX operand register.



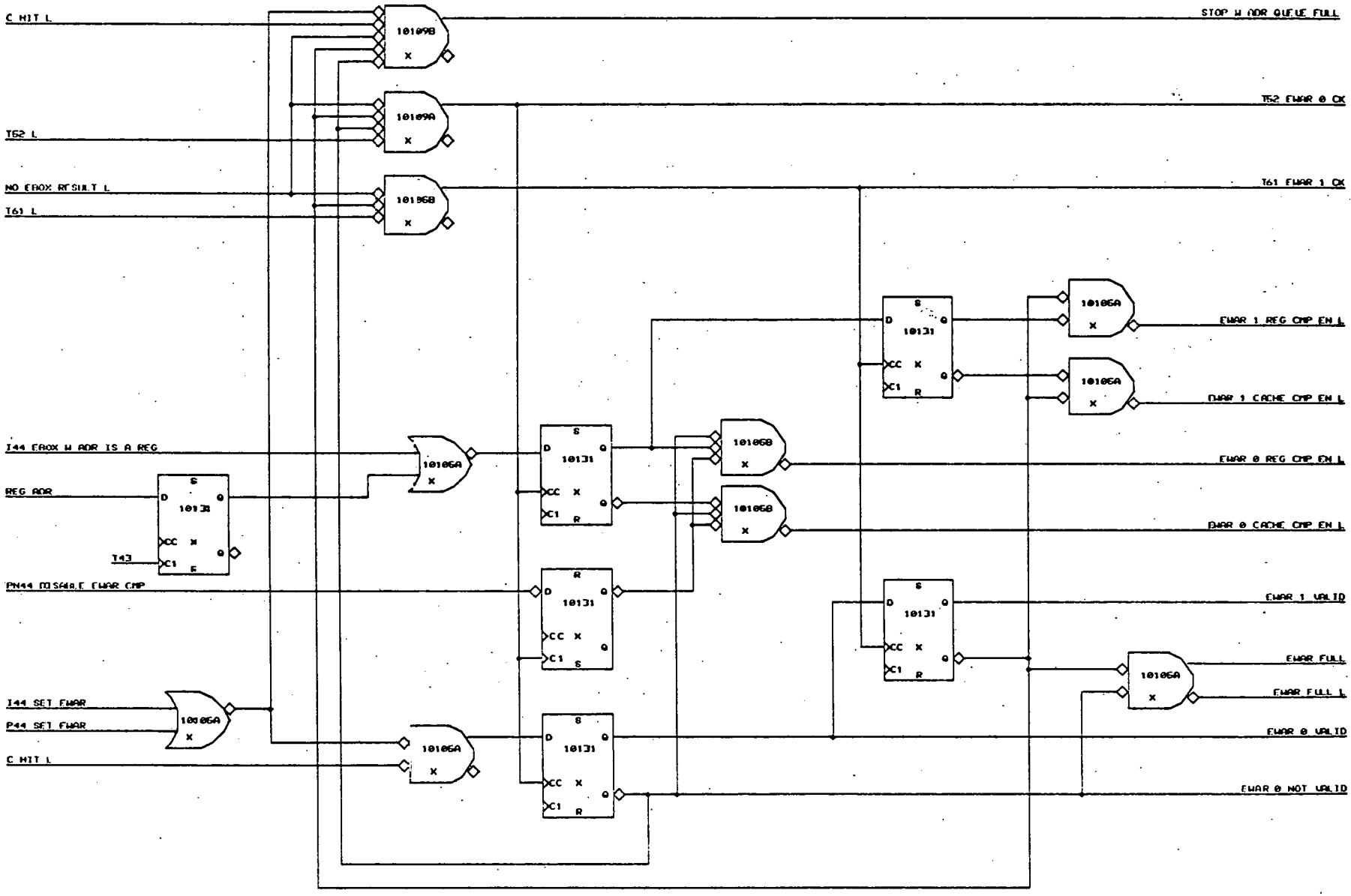
EBOX Write Address Registers 1/4 (EWAR1)



EBOX Write Address Registers 2/4 (EWAR2)



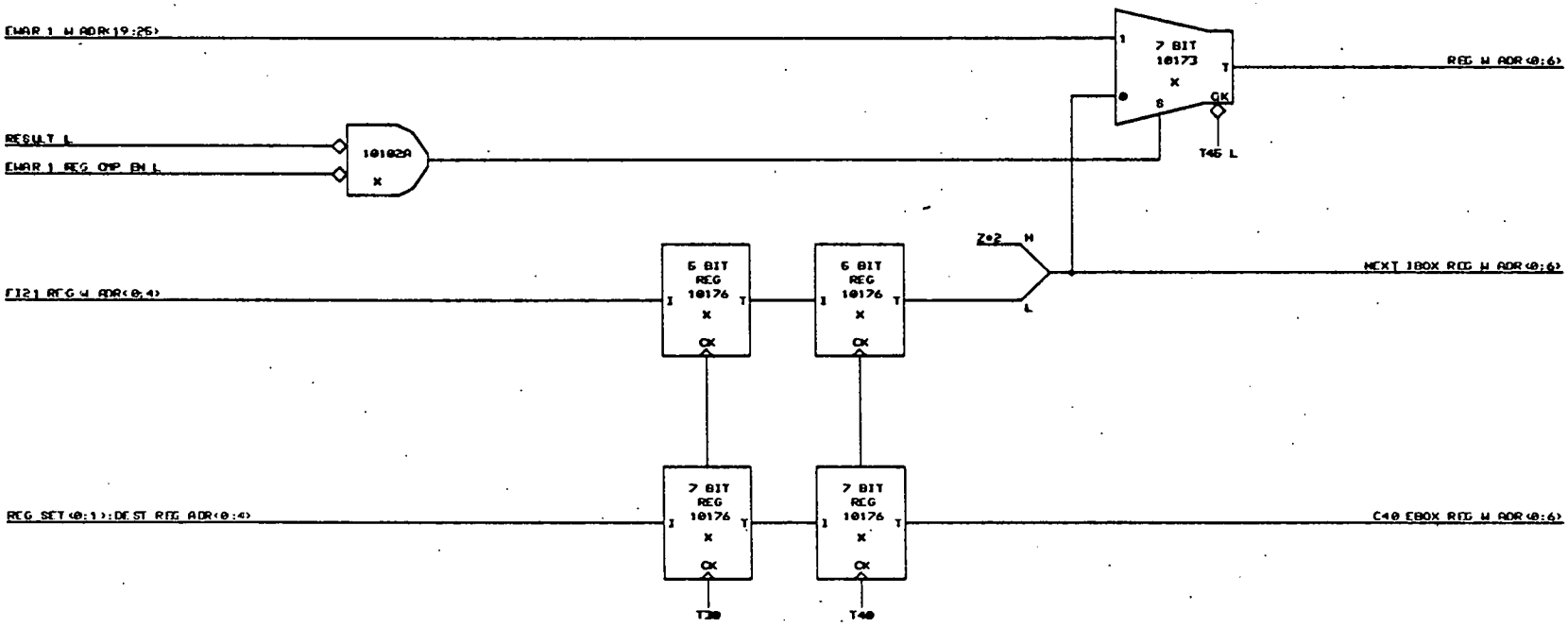
EBOX Write Address Registers 3/4 (EWAR3)



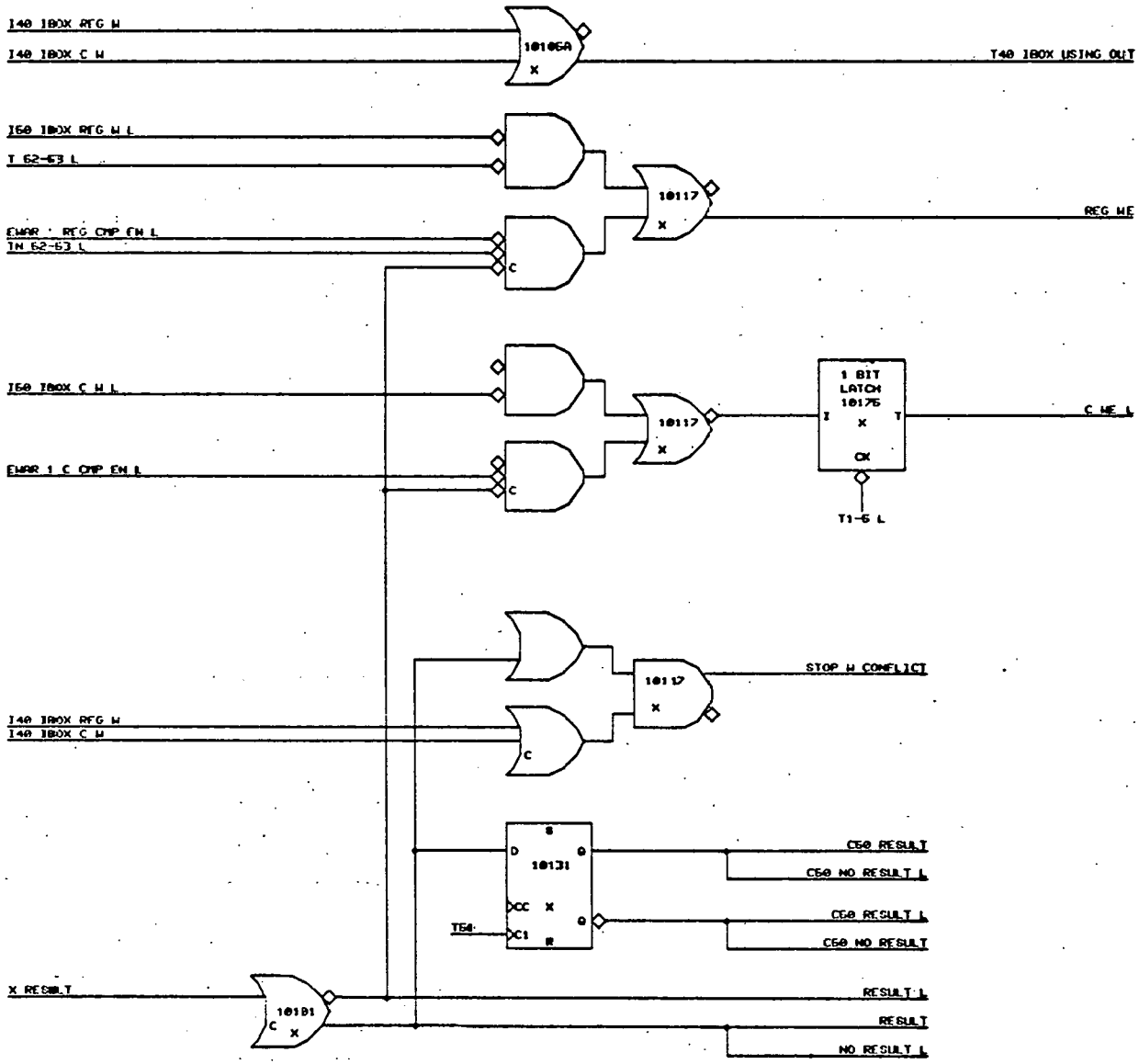
EBOX Write Address Registers 4/4 (EWAR4)

4.1.2.10.5 IBOX Write Control

The IBOX Write Control controls the writing into the cache and register files. The IBOX is structured such that only one thing can be written into either the cache or the general register file at one time. The T register file is completely separate, and can be written in parallel.



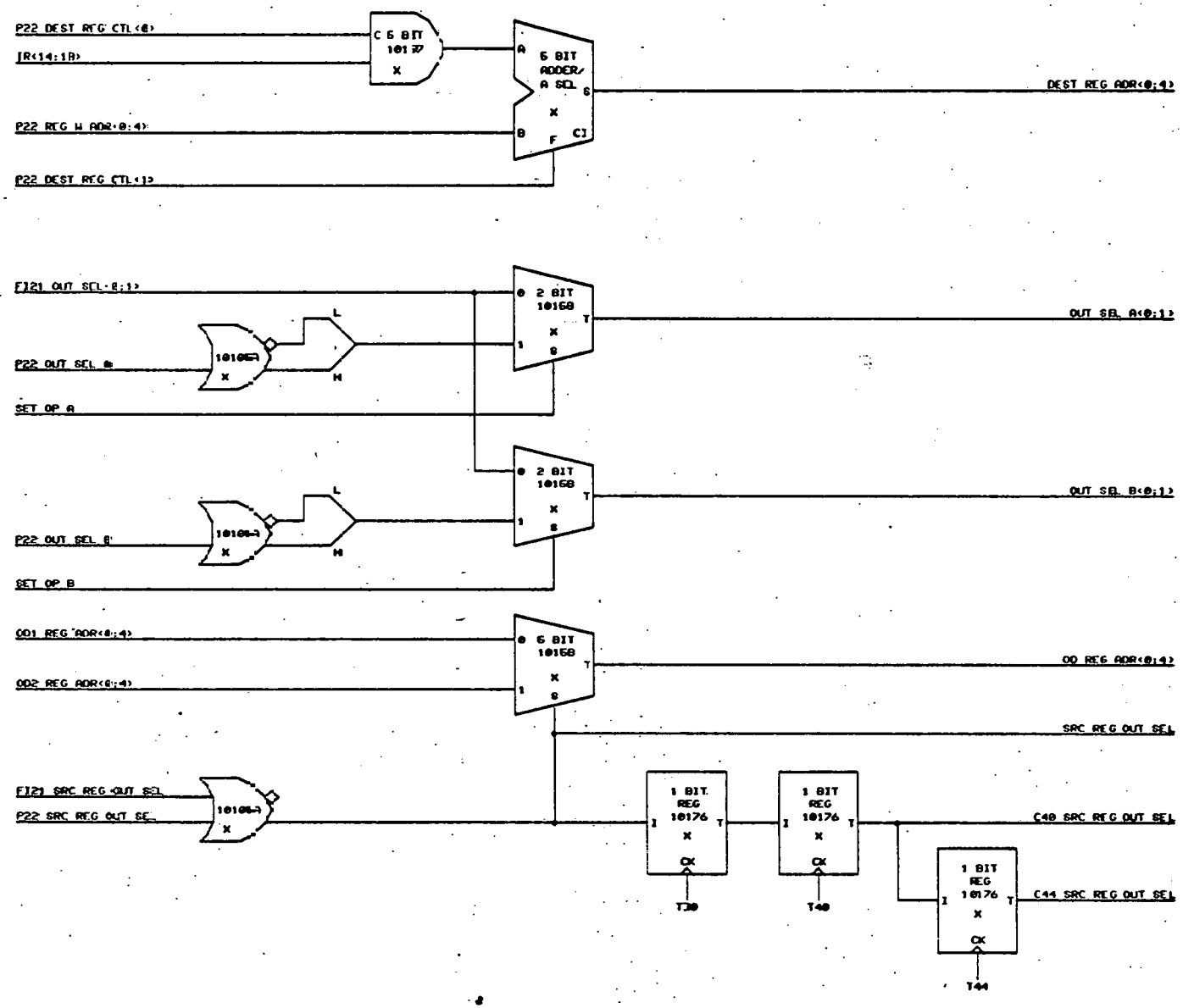
IBOX Write Control 1/2 (IWC1)



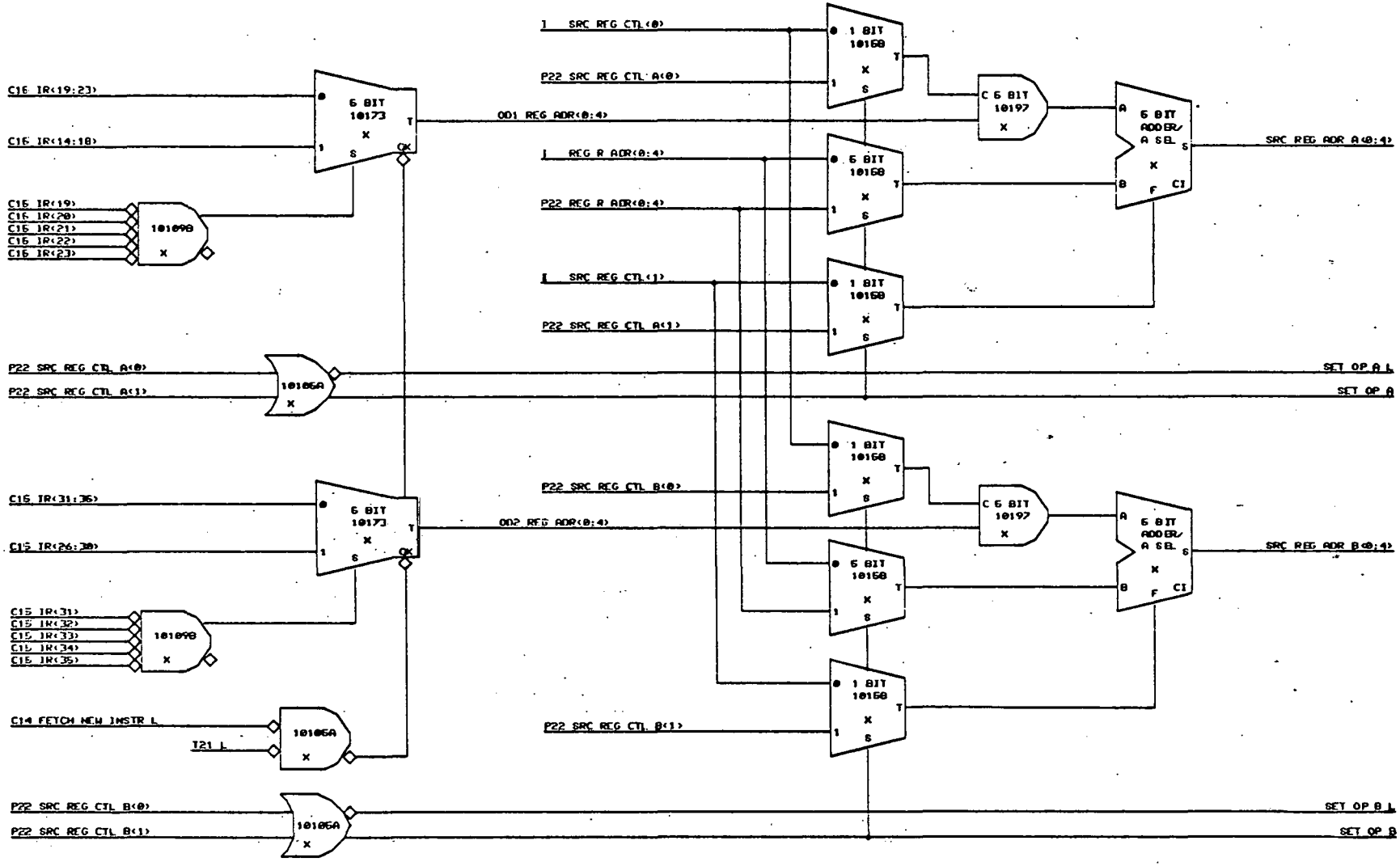
IBOX Write Control 2/2 (IWC2)

4.1.2.10.6 Register Address Generation

The Register Address Generation logic is used to calculate all register addresses for operands address as registers. Since the registers are in the address space, they can also be addressed by using the Data Address Arithmetic logic if some fancy operations want to be preformed, but that ties up the cache. The Register Address Generation logic is used by both the P-Sequencer and the I-Sequencer.

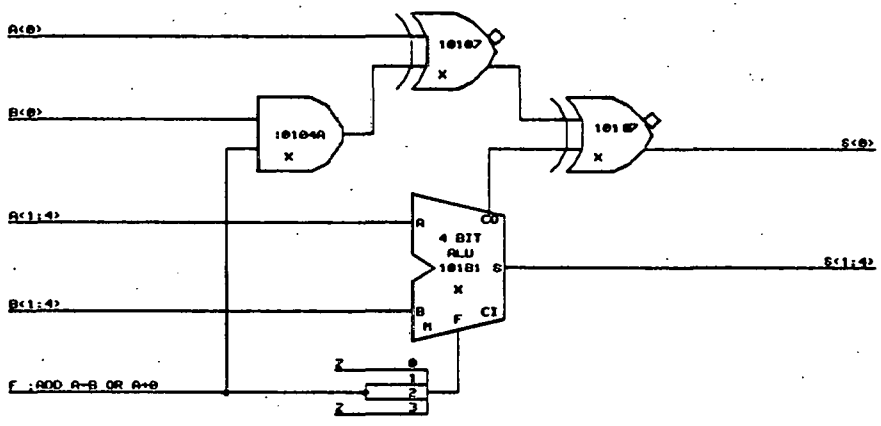


Register Address Generation 1/2 (RAG1)



Register Address Generation 2/2 (RAG2)

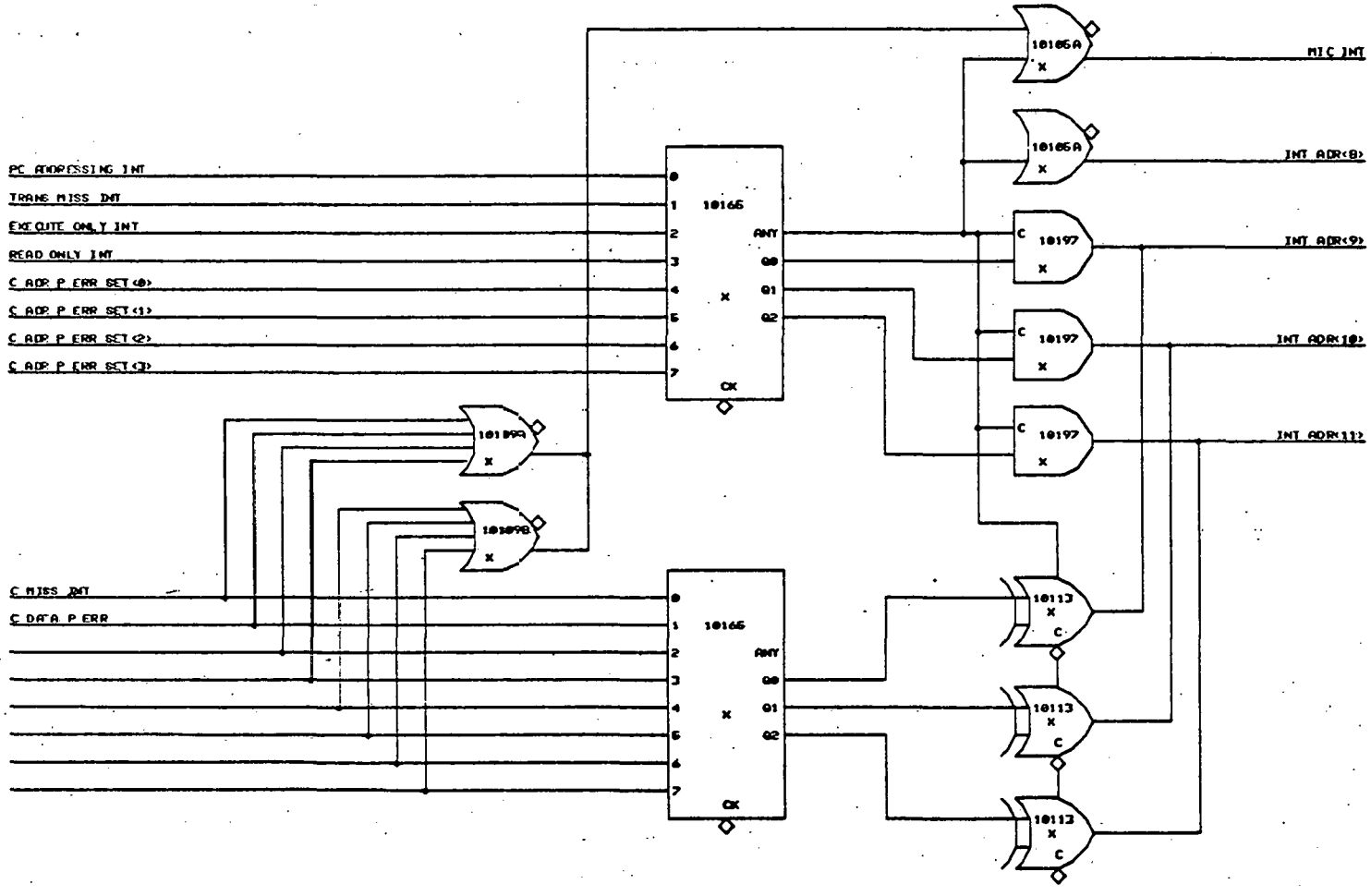
177



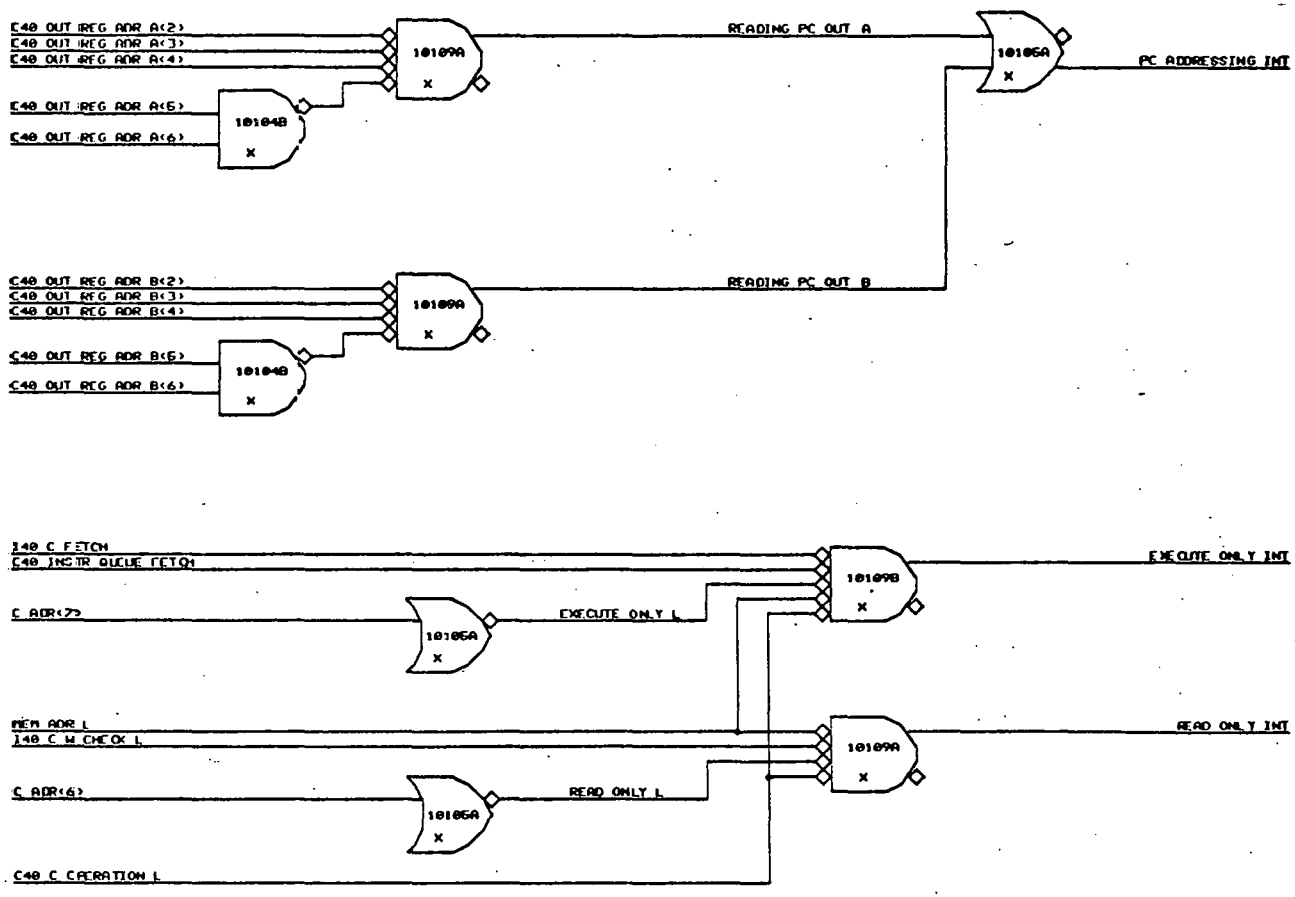
5 Bit Adder/A SEL (ADASEL)

4.1.2.10.7 Micro Interrupts

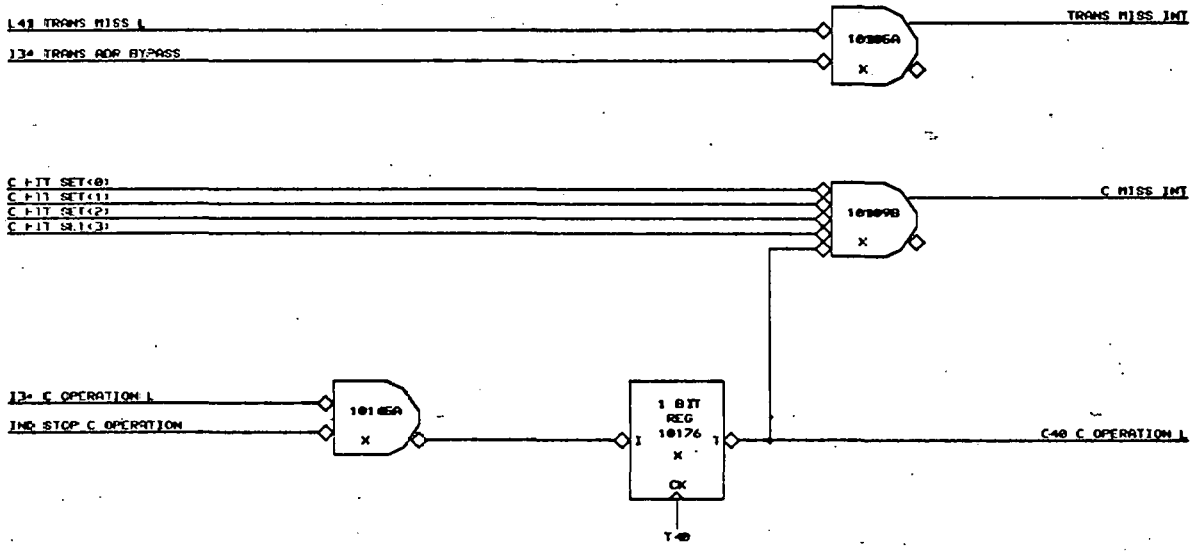
Micro Interrupts allow various conditions to interrupt the I-Sequencer to be handled, such as a cache or page fault miss. When this happens, the micro-program PC is stored on the subroutine stack, and instructions start getting fetched at the micro-interrupt address assigned to that particular interrupt. The various micro-interrupts are all fed into a priority encoder, which comes out with the address of the highest priority interrupt.



Micro Interrupts 1/3 (MICI1)



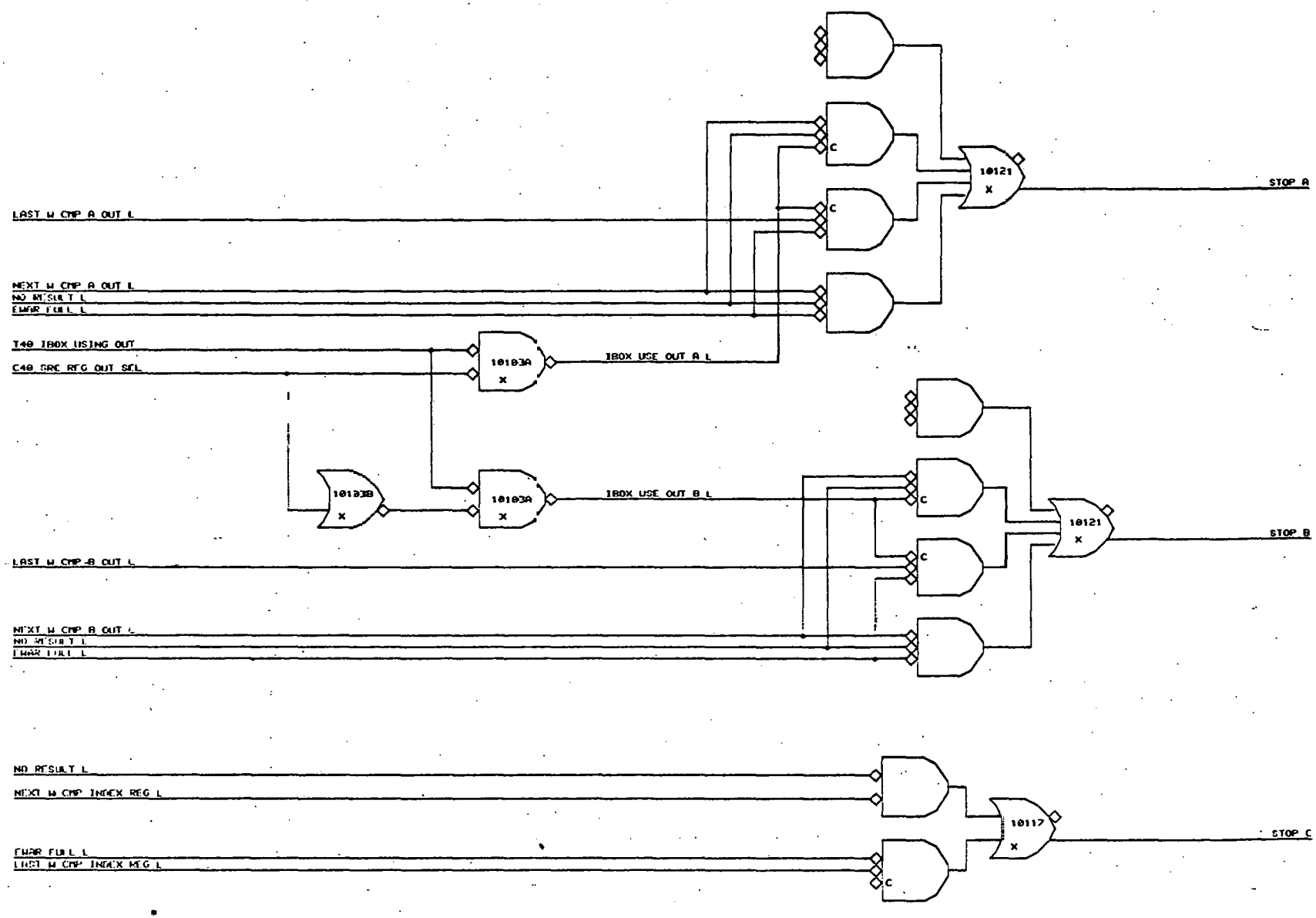
Micro Interrupts 2/3 (MICI2)



Micro Interrupts 3/3 (MICI3)

4.1.2.10.8 Stop IBOX

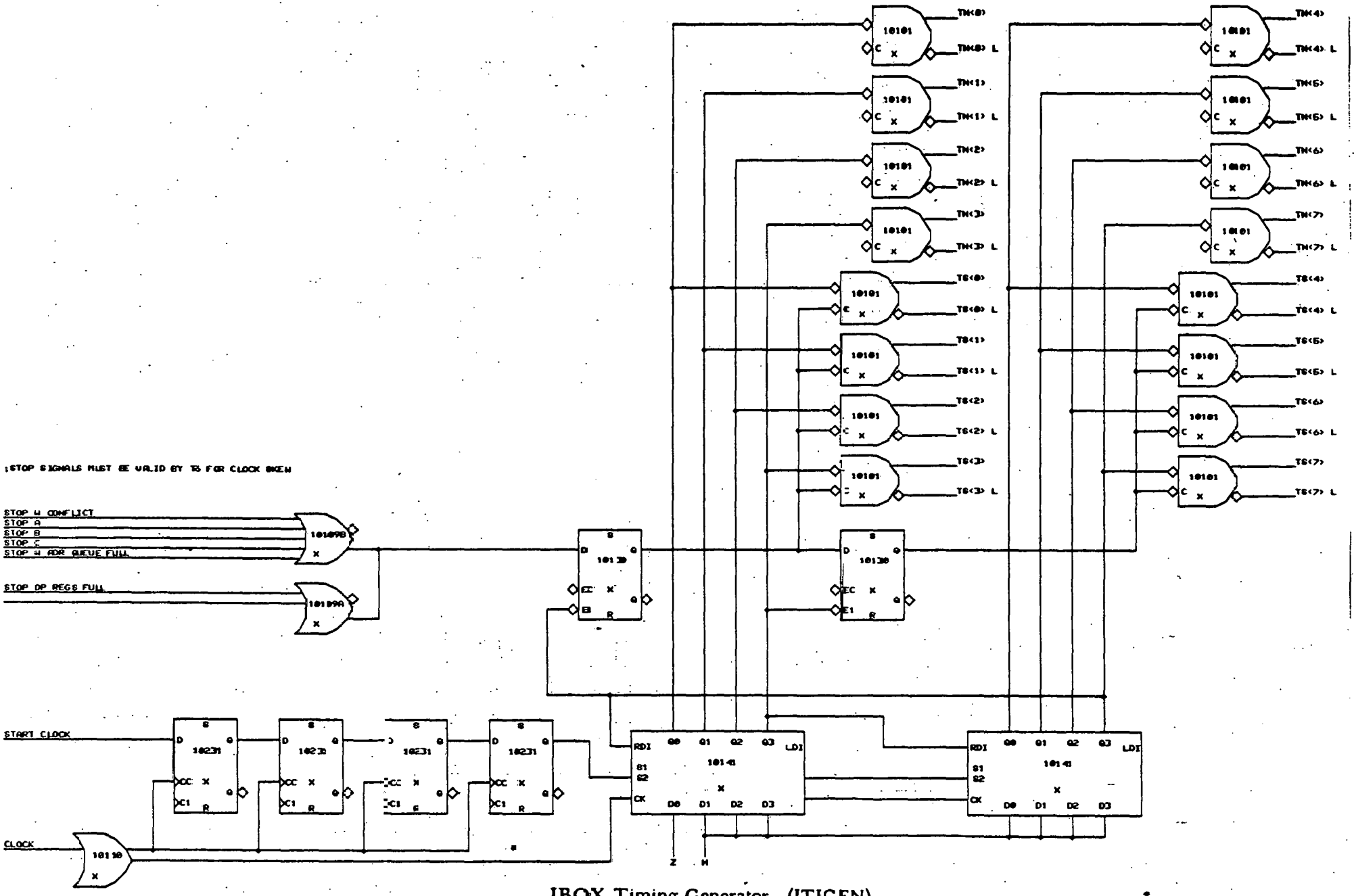
The Stop IBOX logic detects the conditions which cause the IBOX to stop its clock and wait for some event to occur.



Stop IBOX (STOPBX)

4.1.2.10.9 IBOX Timing Generator

The IBOX Timing Generator generates the eight phases of the clock used in the IBOX. It consists of an eight bit circular shift register which is initialized to the sequence 01111111, and it just circulates the zero around. The shift register is never really stoped, but when the IBOX wants to stop its clock, it just disables the output drivers on the shift register.



IBOX Timing Generator (ITIGEN)

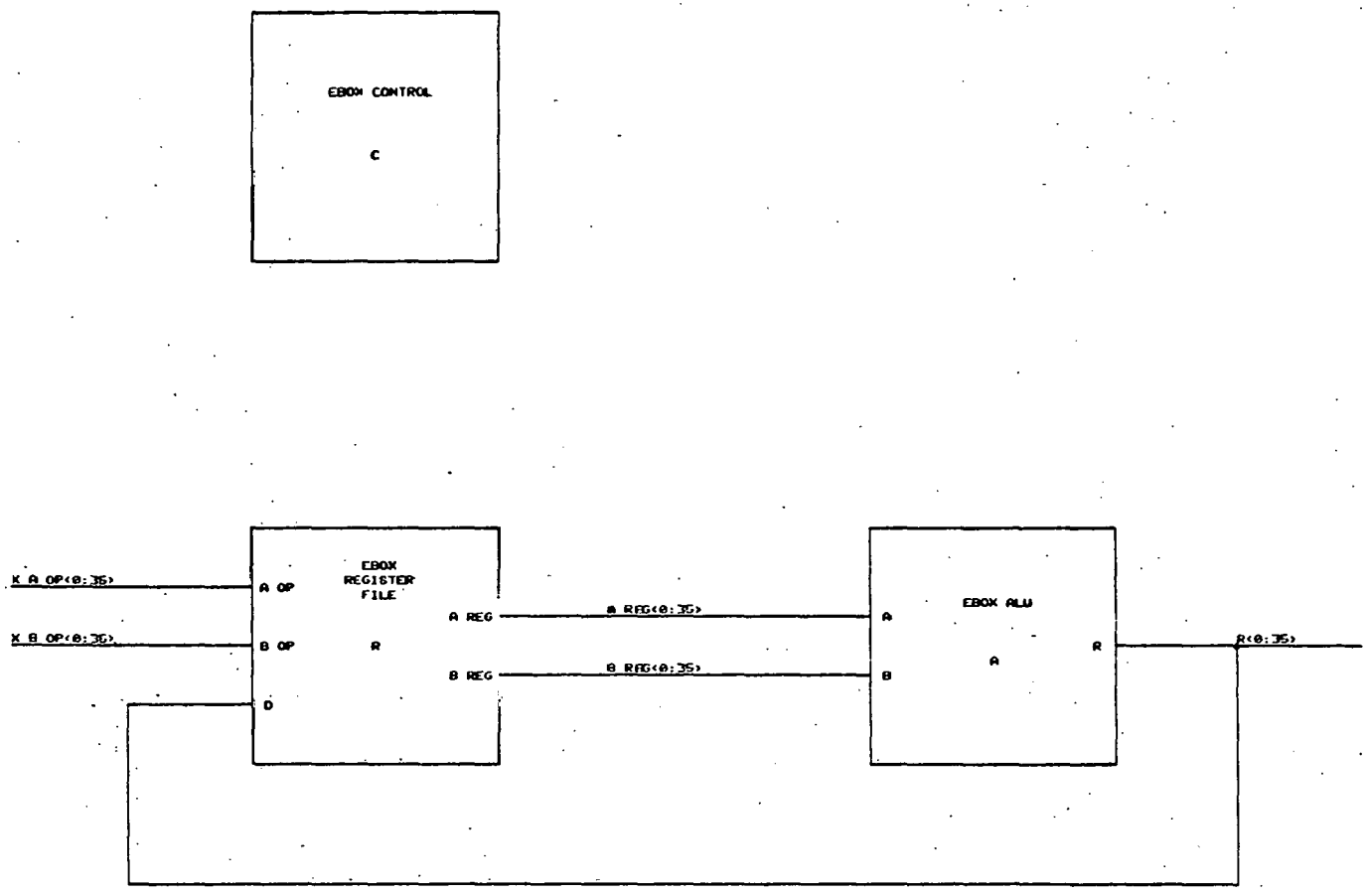
4.1.3 Execution Box

The function of the Execution Box (EBOX) is to perform variable-precision arithmetic and logical operations for the IBOX; it executes one micro-instruction each 100 nano-seconds. EBOX can be decomposed into the EBOX ALU (EBXALU), the EBOX Register File (EREGF), and the EBOX Control (EBXCTL).

The EBXALU performs arithmetic and logical functions on two operands read during each cycle from the register file.

The EREGF contains 32 read/write registers. During a single micro-cycle, any two registers can be read for use as input to the EBXALU. Furthermore, during a micro-cycle two input operands from the IBOX can be written into any even-odd pair of registers, or the result of the EBXALU operation can be written into any register, or one operand from the IBOX *and* the result of the EBXALU operation can be written into even-odd pair of registers.

The EREGF also can shift quarter-words and half-words into position for the EBXALU, can sign-extend floating point numbers, and can deliver zero operands.



Execution Box (EBOX)

4.1.3.1 EBOX Register File

- The EBOX Register File (EREGF) stores initial and intermediate operands for use by the EBXALU during a sequence of micro-operations.

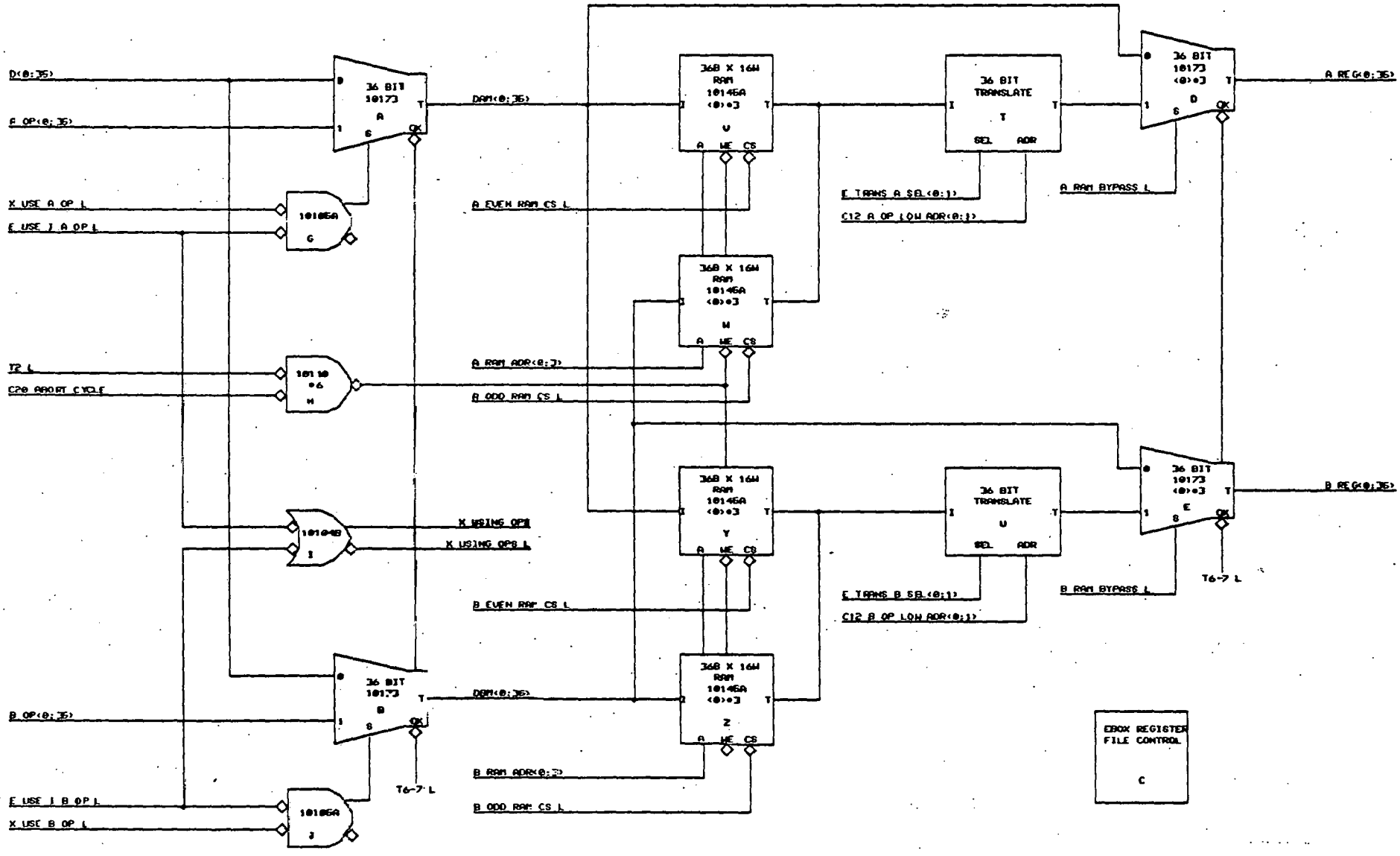
The EREGF contains two duplicate banks each of 32 36-bit registers (R[0:31]). Identical data is always written into both banks. During a single micro-cycle, the IBOX A and B input operands can be written into any even-odd pair of registers (A into an even register and B into an odd register), or the result of the EBXALU operation can be written into any register, or one of the IBOX input operands can be written into a register (only an even register for the A operand, and only an odd register for the B operand) *and* the result from EBXALU can be written into the other register of the even-odd pair.

Since the first cycle of a micro-instruction sequence normally takes two input operands from the IBOX, the result of the previous cycle (ie., the last cycle of the previous micro-instruction sequence) can not be saved in the EREGF.

Because the two register banks contain identical data, any *two* registers may be read out during a micro-cycle for use as input to the EBXALU. In the case of a micro-instruction which reads the result of the preceding operation (or a micro-instruction which reads the A or B input operands from the IBOX), the necessary data is bussed around the register banks, therefore, although writes physically occur one cycle late, they logically occur on time, except as noted below.

Each operand read out of the EREGF can be independently translated. The available translation modes are: straight through, floating point sign extension, left justification of a quarter-word, and left justification of a half-word. Operands which are bussed around the register banks (as described above) cannot be translated.

The EREGF also has the capability to deliver zero operands on either the A or B output independently by disabling the register file output.



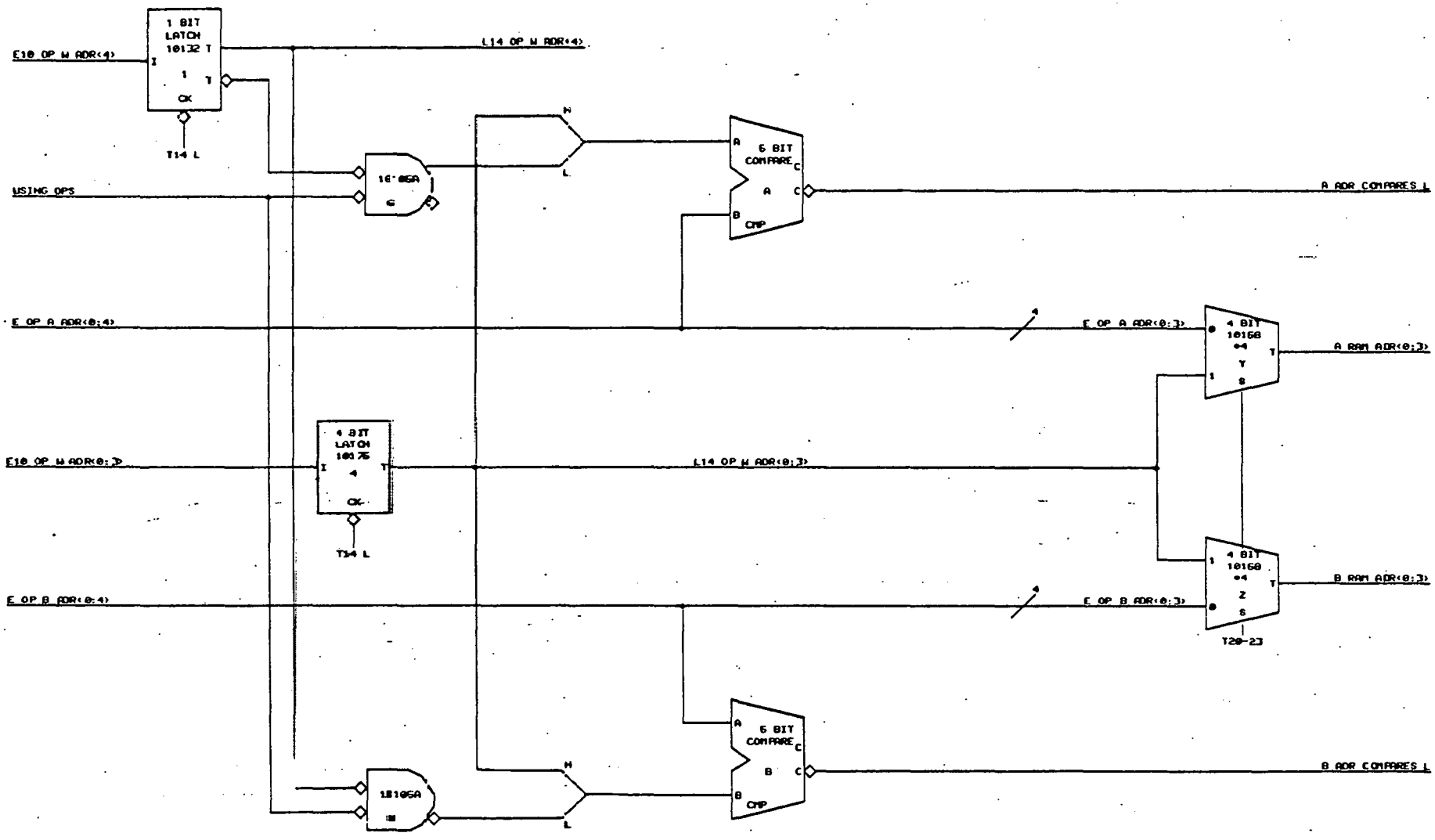
EBOX Register File (EREGF)

4.1.3.1.1 EBOX Register File Control

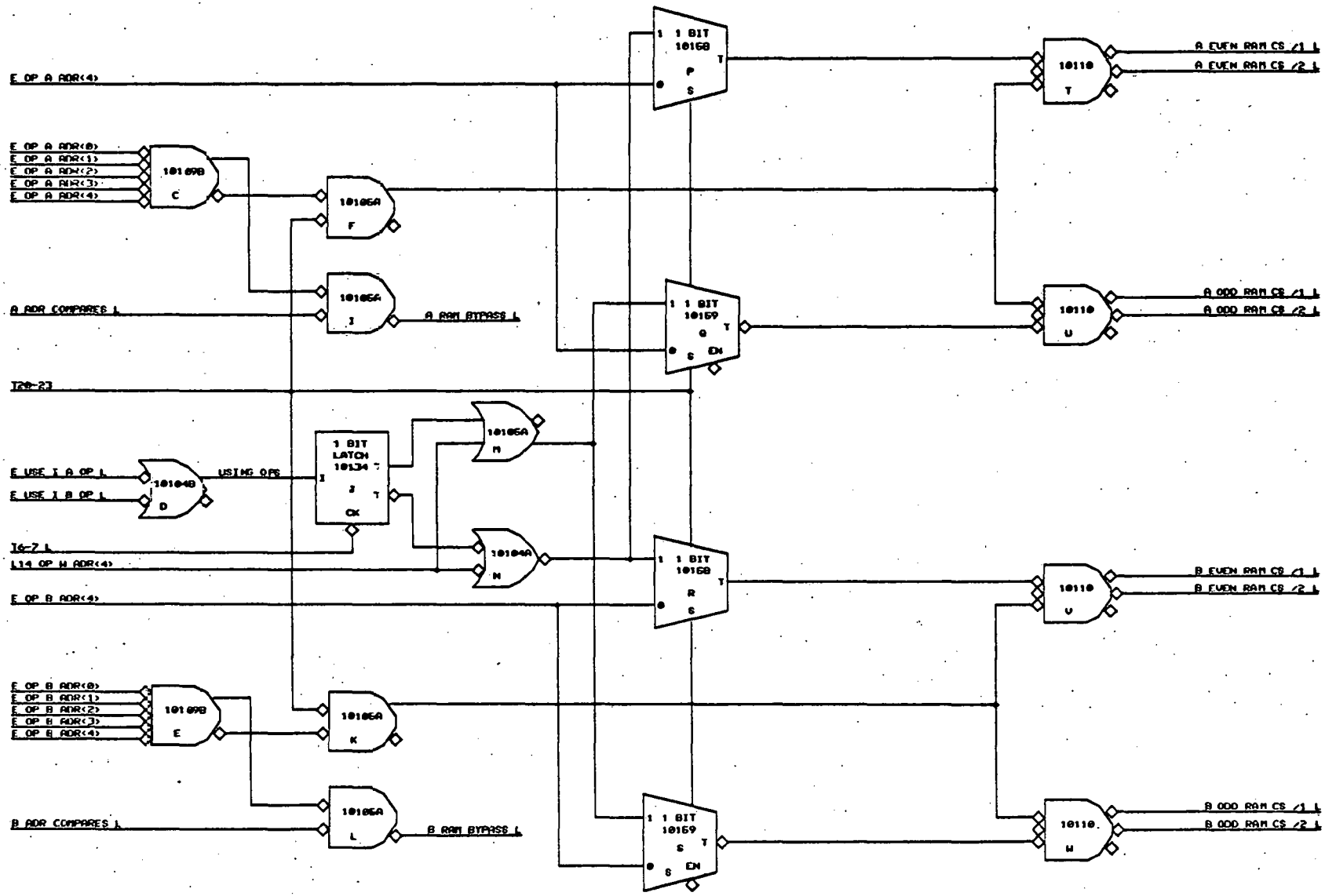
The EBOX Register File Control (ERFC) primarily detects when the current micro-instruction is attempting to use a value which will not be written into the register file until the next cycle, and in that case commands the EREGF to bus the data around the register banks.

The ERFC also detects when R[0] is being read out (on either the A or B output) and commands the EREGF to output the value zero. R[0] can therefore not be used to contain data.

The ERFC also controls the chip select lines for the EREGF so that either one or two values may be written into the duplicate register banks.



EBOX Register File Control 1/2 (ERFC1)



EBOX Register File Control 2/2 (ERFC2)

4.1.3.1.2 36 Bit Translate

The 36 Bit Translate (TRANS) is used on each output leg of the EREGF. Each TRANS is independent and has the capability to perform four different translations as follows:

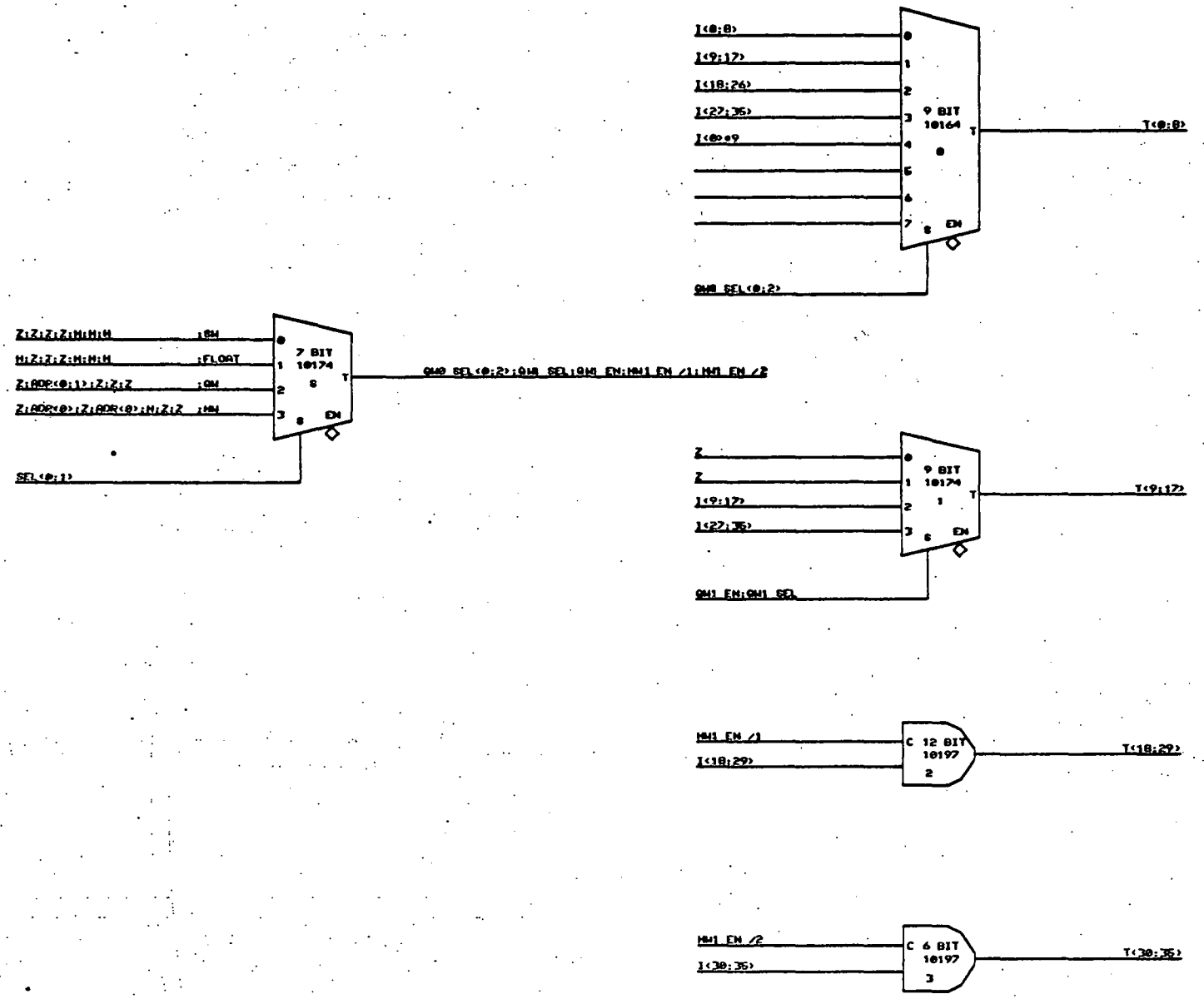
1. Straight through. The value passes straight through the TRANS without modification.

2. Sign extension of a floating point mantissa. Each bit of the exponent of a floating point number is replaced by bit 0 of the floating point number.

3. Quarter-word. One of four quarter words (depending upon the low-order address bits from the IBOX) is left justified, and the low-order quarter words are set to zero.

4. Half-word. One of two half words (depending upon the low-order address bits from the IBOX) is left justified, and the low-order half-word is set to zero.

The TRANS can not be used to modify the result of the preceding micro-instruction.



36 Bit Translate (TRANS)

4.1.3.2 EBOX ALU

The EBOX ALU (EBXALU) performs all arithmetic and logical operations for the EBOX. The EBXALU can be decomposed into the 3 Input Adder (3INADD), the Shift Box (SHFBOX), the Exponent Box (EXPBOX), the 36 Bit MUX Merge (MUXMRG), and the Q Register (QREG).

The 3INADD can add three operands, perform a few other logical functions on three operands, or perform general logical functions on two operands. The input operands to the 3INADD are A (the A output of EREGF), B (the B output of EREGF), and Q (the quotient register, QREG). Internally, the operands are shifted and multiplexed so that a single micro-cycle can do four bits of a multiply.

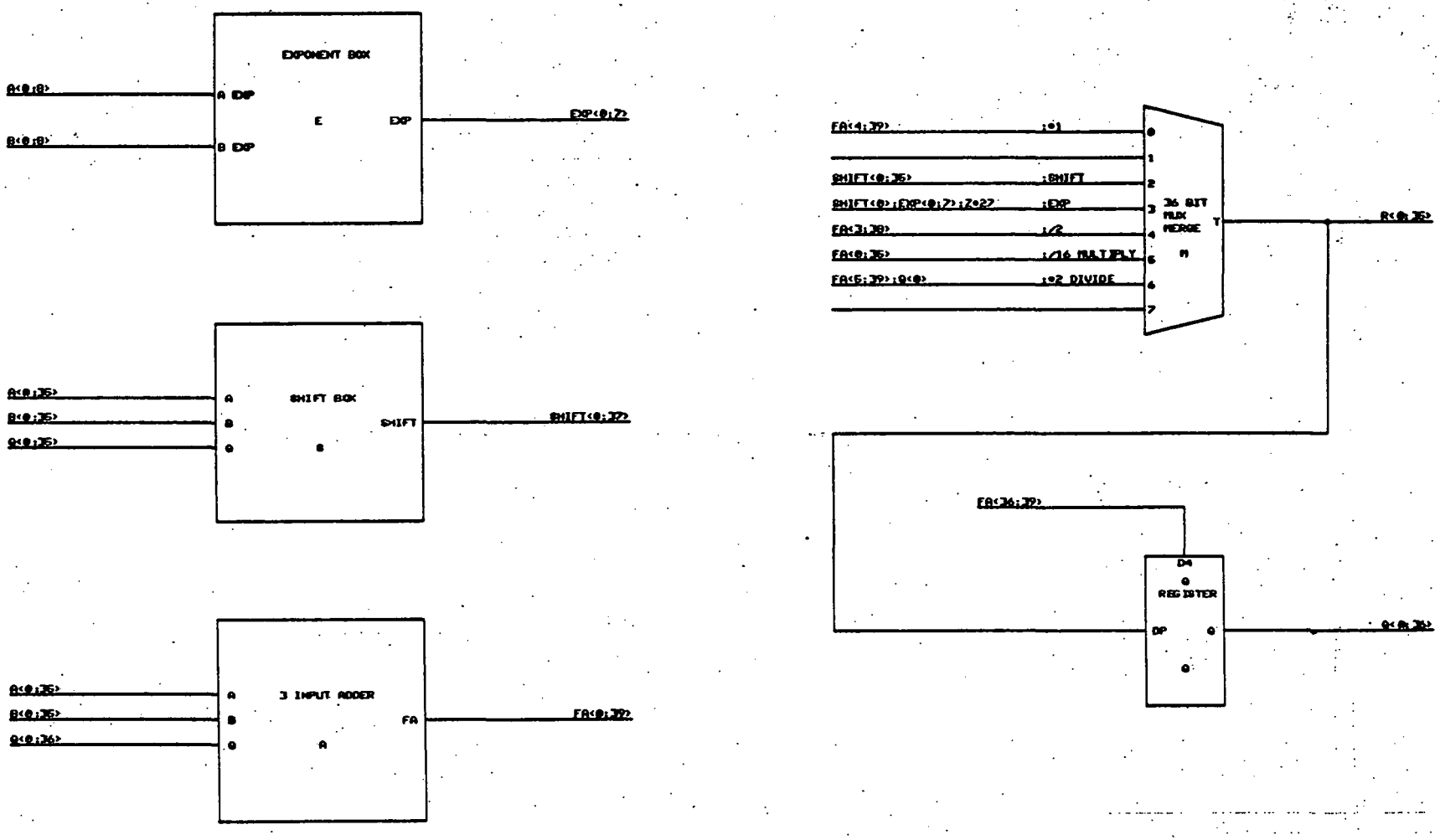
The SHFBOX can do arithmetic or logical left or (limited) right shifts of a double-word input onto a single word output. The three single-word inputs to the SHFBOX can be combined in various orders to accomplish single-word arithmetic or logical left right shifts or rotates of up to 36 bits in a single cycle.

The EXPBOX performs exponent arithmetic. The EXPBOX has its own internal registers, so that after loading the EXPBOX from the A and B operands, exponent arithmetic can proceed independently of the computations in the main data path.

The MUXMRG produces the one EBXALU output, R<0:35>. The inputs of the MUXMRG are from 3INADD, SHFBOX, and EXPBOX. Special inputs are provided for special functions; one input merges the exponent with the shifter output, one input does a multiply shift, and one input does a divide shift.

The MUXMRG also has the capability to selectively merge each quarter-word from the SHFBOX with the output of the 3INADD. This capability is used to merge the result of a quarter-word or half-word operation (which is shifted into place in the SHFBOX) back into the destination word (which passes unmodified through the 3INADD). In this case the destination low-order address bits control the MUXMRG.

The QREG holds the multiplier during a multiply sequence, and holds the dividend during a divide sequence. The QREG has shifting capability internally. The QREG can also be used to hold temporary results (for example, over the boundary between one micro-instruction sequence and the next).



EBOX ALU (EBXALU)

4.1.3.2.1 3 Input Adder

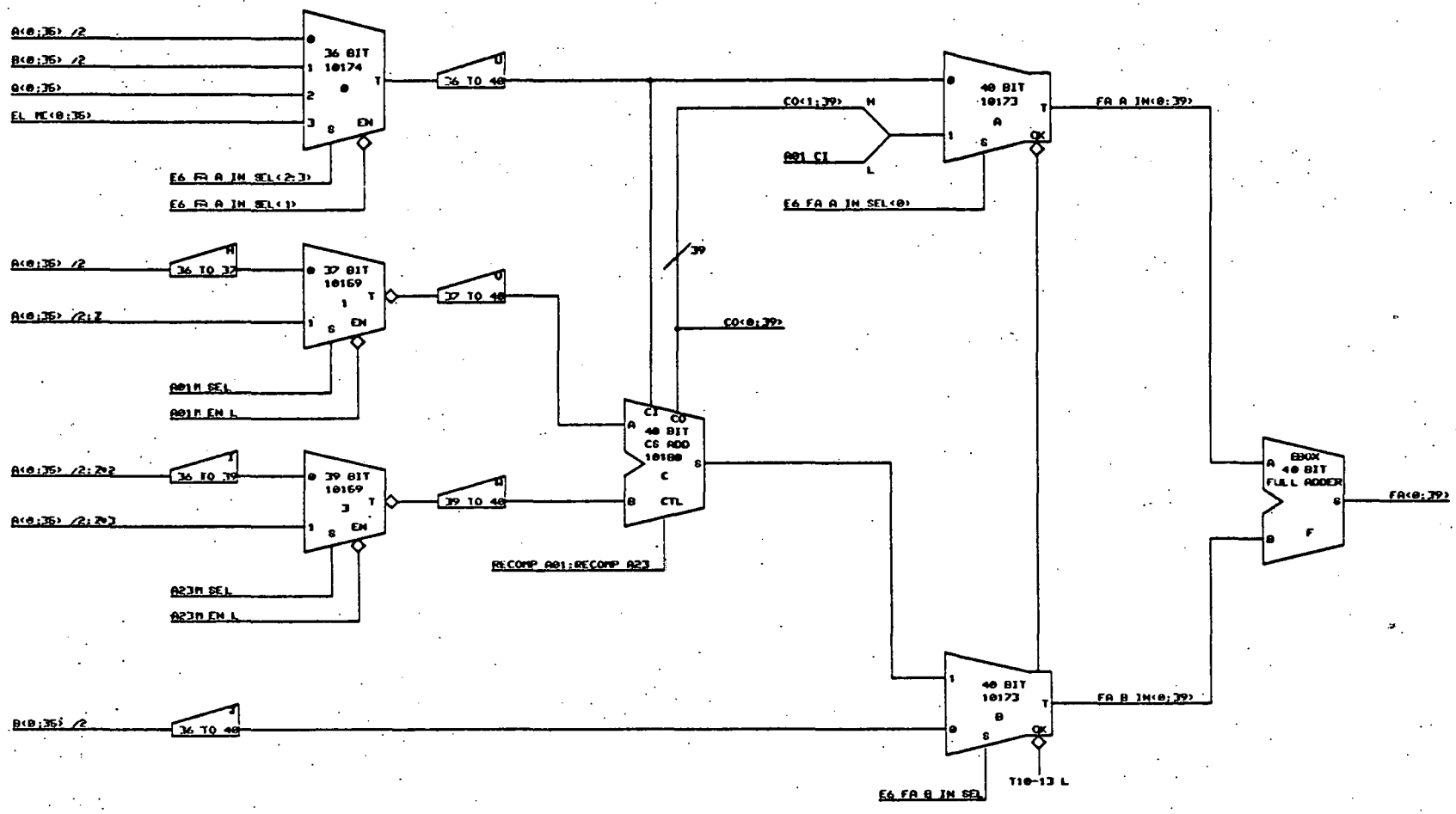
The 3 Input Adder (3INADD) has the capability to add three 36-bit numbers, to perform some other limited logical operations of three 36-bit numbers, or to perform general logical operations on two 36-bit numbers. The three-input addition capability is used primarily to produce 4 bits of a multiply operation in one micro-cycle.

The 3INADD can be decomposed into the Carry Save Adder (CSA), the EBOX Full Adder (EFA), and various multiplexers and multiplexer latches.

The CSA is an array of 20 ECL 10180 chips. The CSA forms the first two legs of the three-input adder. During a three-input add, the CSA adds three operands to produce a sum and carry vector output (each 40 bits long), and EFA adds those vectors to complete the add. Two legs of the CSA are dedicated to A, (or to shifted versions of A) which is the multiplicand in a multiply. The remaining leg of the CSA can receive A, B, Q, or a micro-code constant.

Each of the three inputs of the CSA can be independently set to zero. Furthermore, the 10180 has the capability to independently complement two of its inputs. These capabilities are used in the multiply micro-cycle.

Two-operand functions can be performed in the EFA. One leg of the EFA can receive A, B, Q, or micro-constant (in addition to carry out from the CSA), and the other leg of FA can receive only B (in addition to sum from the CSA). The EFA produces a 40-bit output.



3 INPUT Adder (3INADD)

4.1.3.2.1.1 EBOX 40 Bit Full Adder

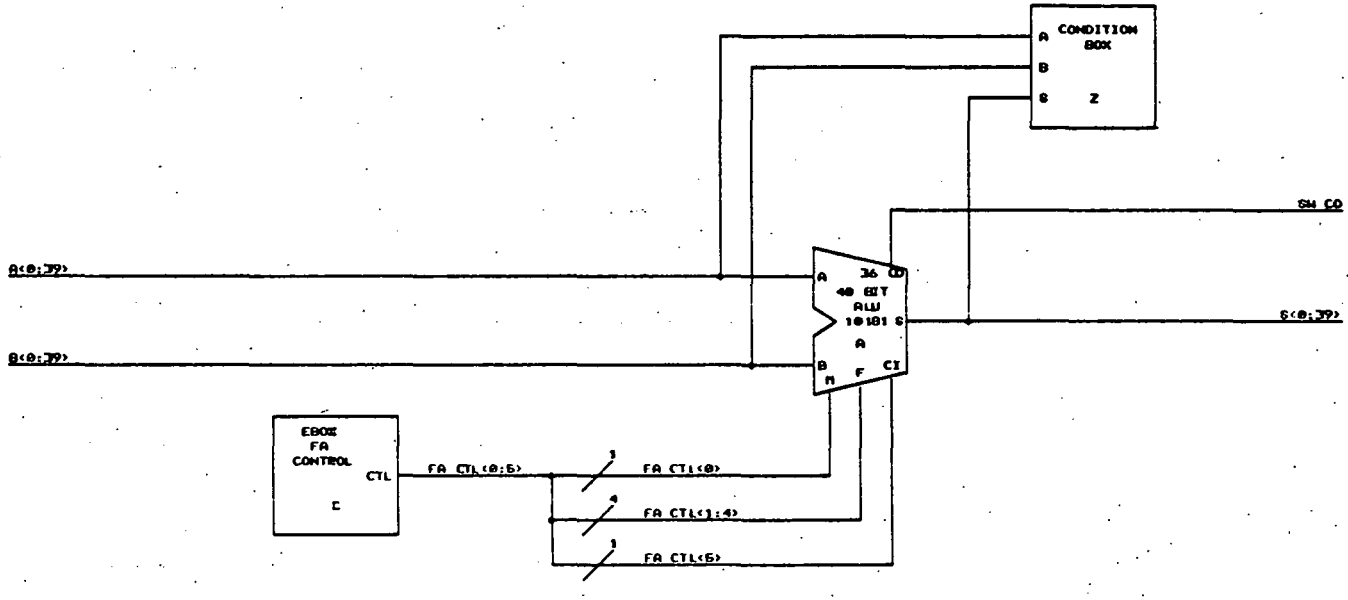
The EBOX 40 Bit Full Adder (EFA) can perform arithmetic and logical functions on two operands. It is constructed with 10181 ECL ALU chips and 10179 ECL carry-look-ahead units.

The EFA can be decomposed into the 40 Bit ALU 10181 (40ALU), the EBOX Full Adder Control (EFACTL), and the Condition Box (CBOX).

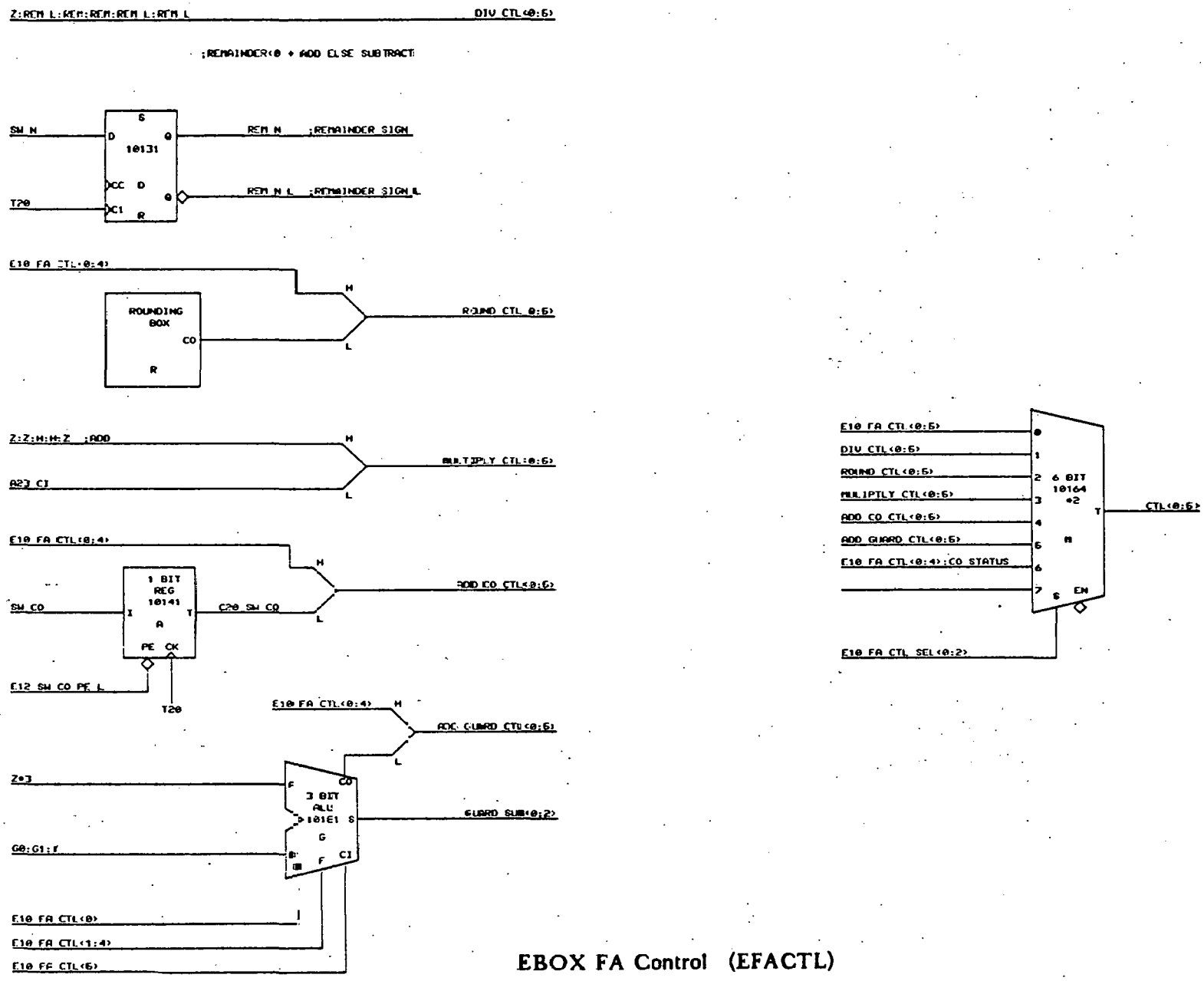
The 40ALU performs a full add in 24 nano-seconds worst case from the data inputs, neglecting wire delays. It also performs the full 10181 repertoire of logical and arithmetic functions.

The EFACTL controls the EFA, producing the mode, function, and carry-in signals for the 10181. The mode and function bits can come either from micro-code or from the divide logic. The carry-in bit can come from divide logic, rounding logic, multiply logic, carry-out of a previous cycle, guard-bit logic, or micro-code. The Rounding Box (ROUND) saves guard bits during floating point operations and generates a carry-in bit for the EFA depending upon guard bits and rounding mode.

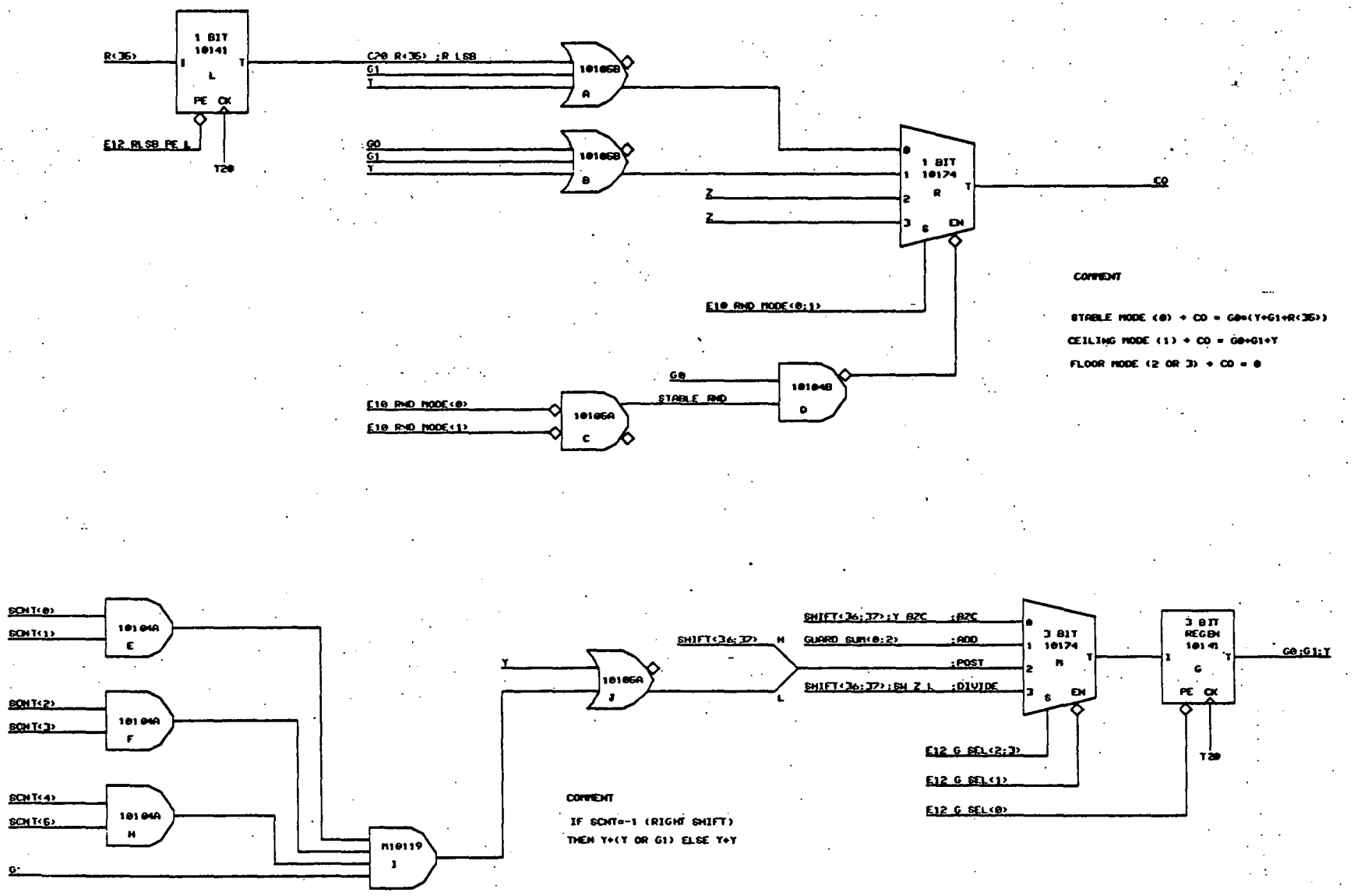
The CBOX detects single-word overflow, single-word negative, single-word zero, single-word less than or equal to zero, mantissa zero, and mantissa overflow. Single-word carry out is generated directly in the 40ALU. Since quarter-words and half-words are left justified and zero-filled in the TRANS, the single-word conditions are sufficient for testing quarter-word and half-word operations. Wrong Branch Logic (WRONB) combines the generated conditions with control bits received from the IBOX and determines whether the IBOX took the correct branch on a conditional branch instruction. If the IBOX took the wrong branch, then X WRONG BRANCH automatically becomes asserted.



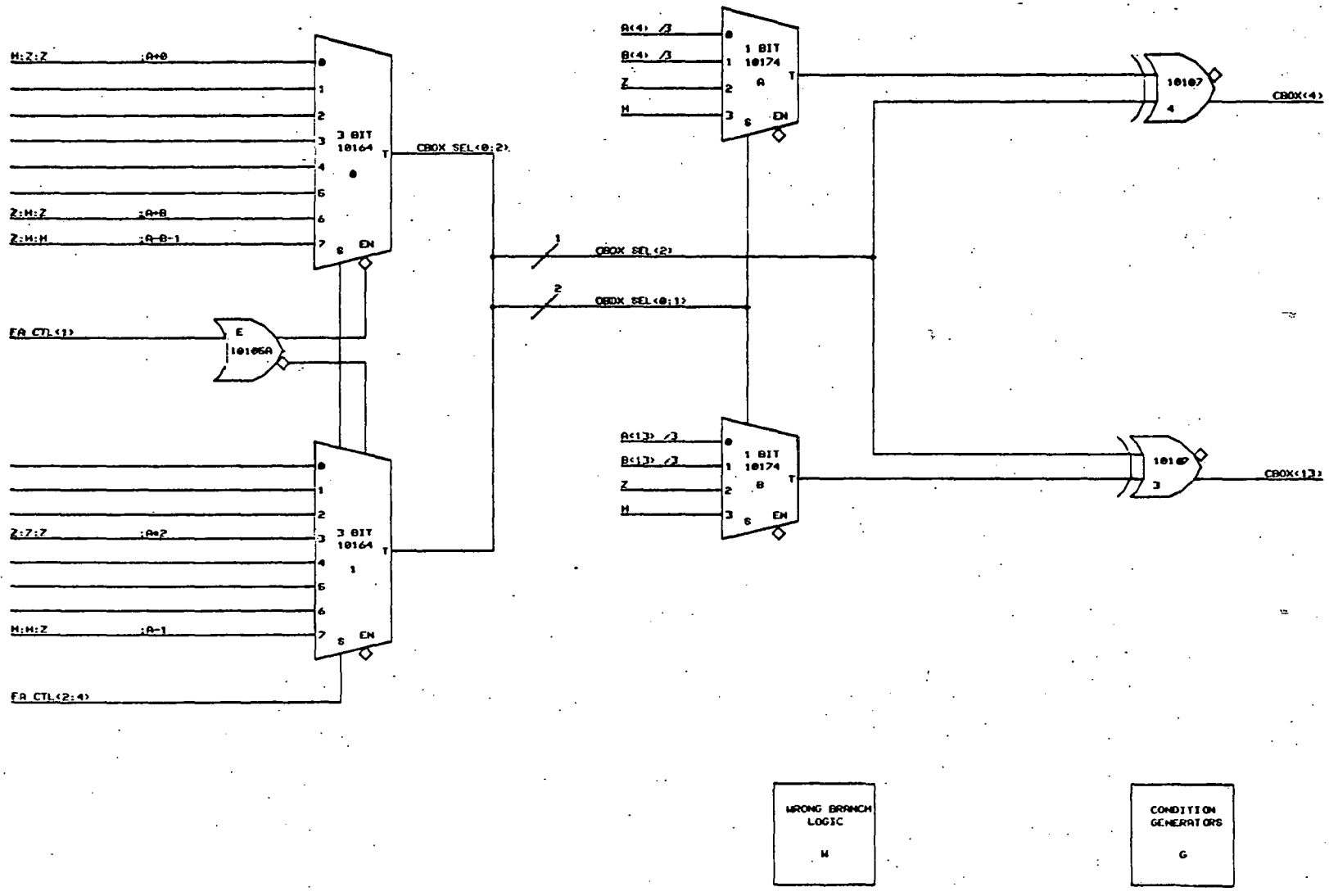
EBOX 40 Bit Full Adder (EFA)



EBOX FA Control (EFACTL)



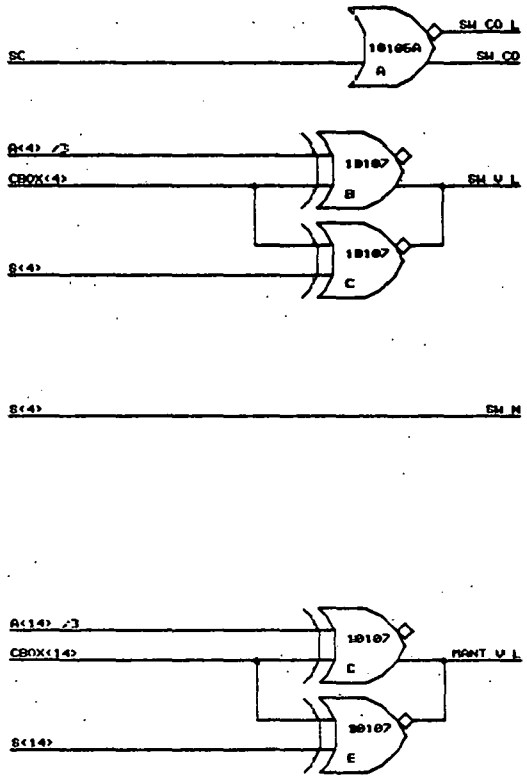
Rounding Box (ROUND)

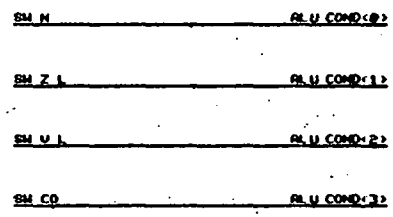


WRONG BRANCH LOGIC
M

CONDITION GENERATORS
G

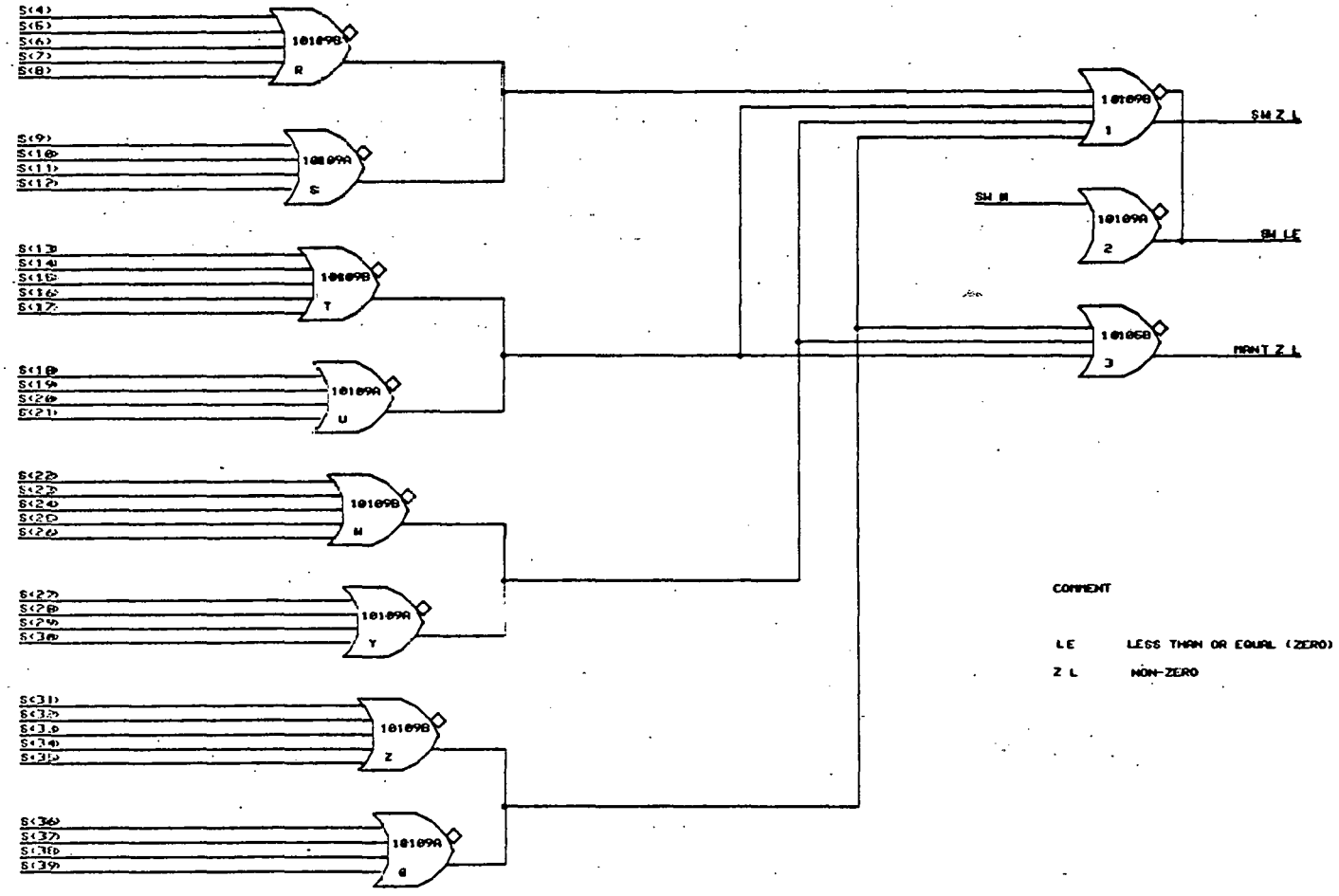
Condition Box (CBOX)



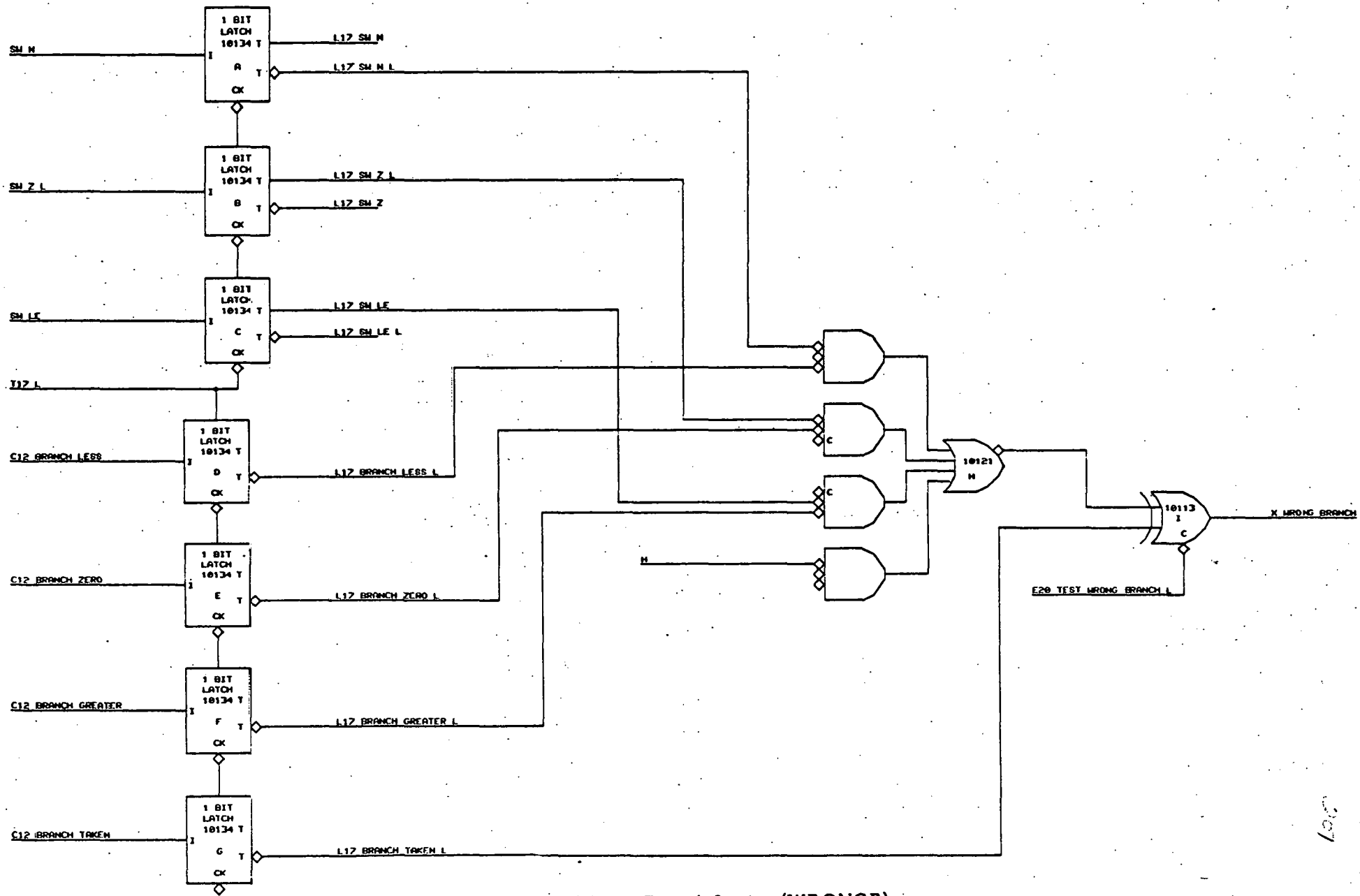


COMMENT	
MANT	MANTISSA
SW	SINGLE WORD
CO	CARRY OUT
V L	NOT OVERFLOW
N	NEGATIVE

Condition Generators 1/2 (CGEN1)



Condition Generators 2/2 (CGEN2)



Wrong Branch Logic (WRONGB)

4.1.4.2.1.2 Multiply Control

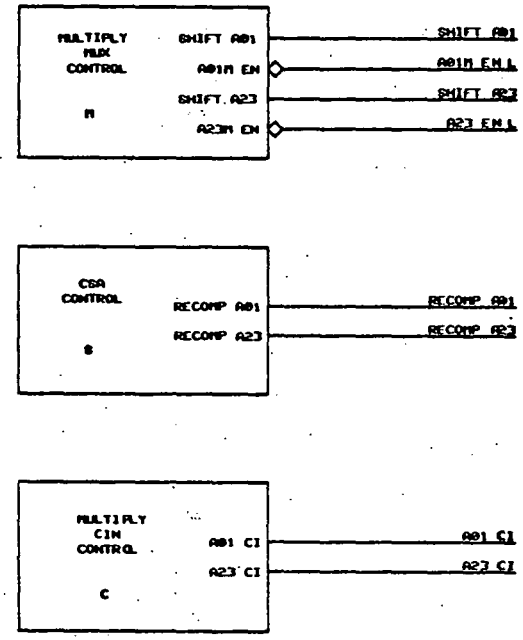
The Multiply Control module (MPYCTL) generates various control signals for use during a multiply cycle.

4 bits of the product are generated during each multiply cycle; the MPYCTL examines the 5 low order bits (including the carry out of the least-significant bit) of the Q register, and sets up the 3INADD to perform the multiply cycle. One leg of the three-input adder receives A (the multiplicand) or A*2, another leg receives A*4 or A*8, and the last leg receives B (the partial product). Each of the "A" legs is either added to or subtracted from the partial product.

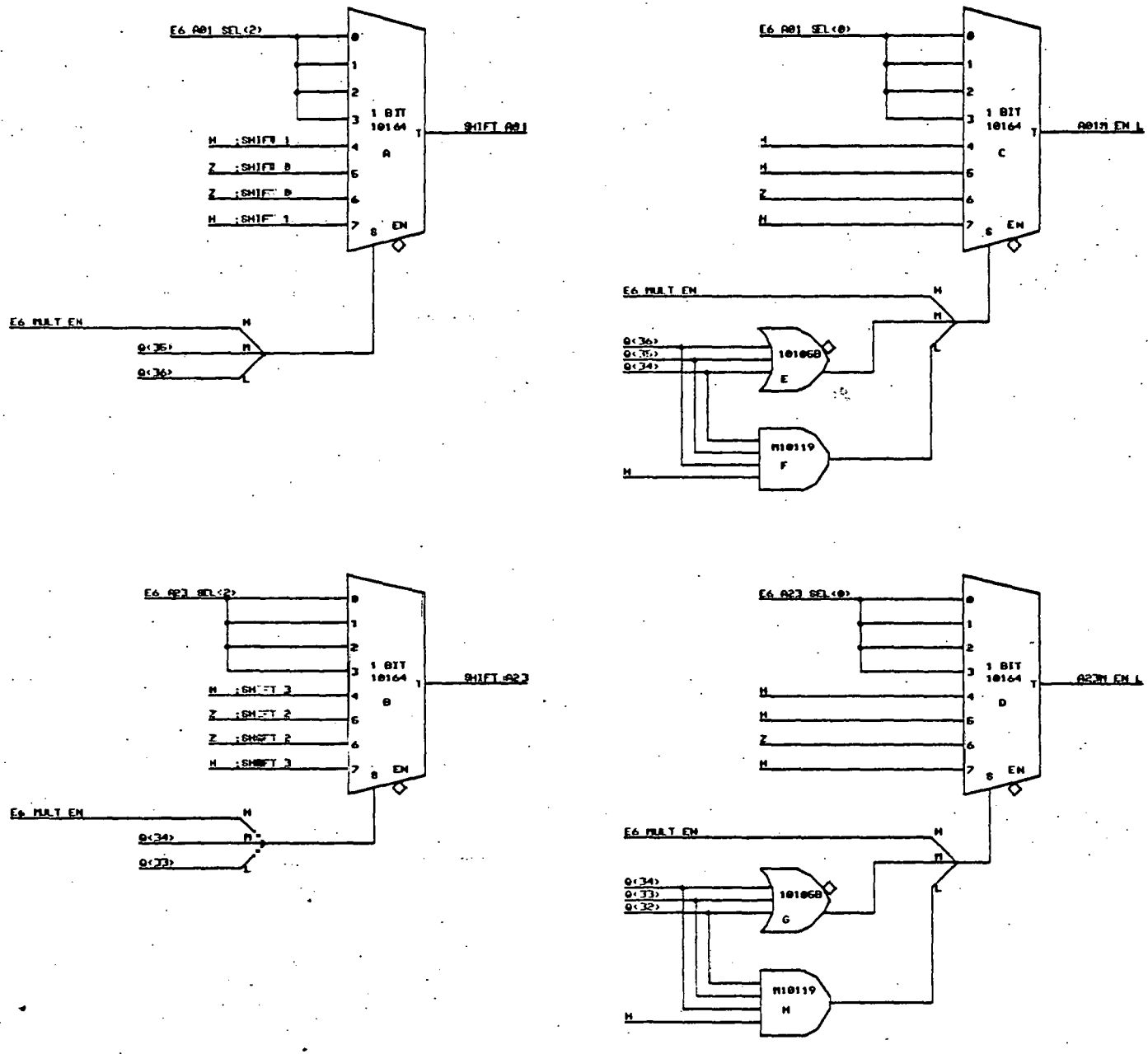
The table included in MPYCTL defines a 2-bit-per-cycle multiply algorithm. X0 represents the least significant bit of the Q register Q<35>, X1 represents Q<34>, and CI represents the carry out from the Q register from the previous cycle (Q<36>). F shows the function to perform, that is, PARTIAL_PRODUCT←PARTIAL_PRODUCT + F*MULTIPLICAND. Q and PARTIAL_PRODUCT are then shifted right by two bits and the cycle repeats. The other columns of the table show the values of various signals which are needed to implement F. The 4-bit-per-cycle algorithm is a direct extension of the 2-bit algorithm; two 2-bit cycles are performed in parallel using the 3INADD and examining 5 bits of Q instead of 3 bits of Q.

COMMENT

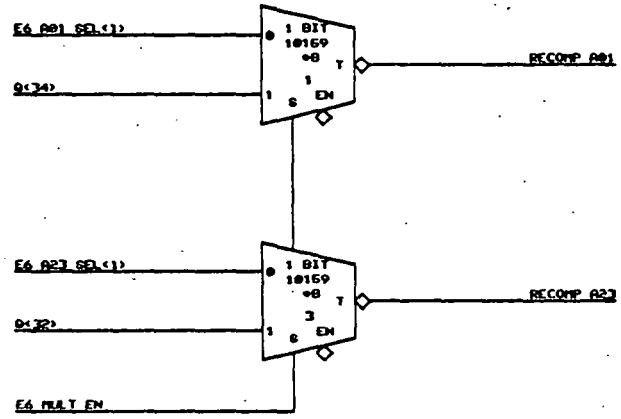
X0	X1	CI	F	SHIFT	10159 EN L	10180 RECOMP	MULT CI
0	0	0	0	1	1	1	1
0	0	1	+1	0	0	1	0
0	1	0	+1	0	0	1	0
0	1	1	+2	1	0	1	0
1	0	0	-2	1	0	0	1
1	0	1	-1	0	0	0	1
1	1	0	-1	0	0	0	1
1	1	1	0	1	1	0	0



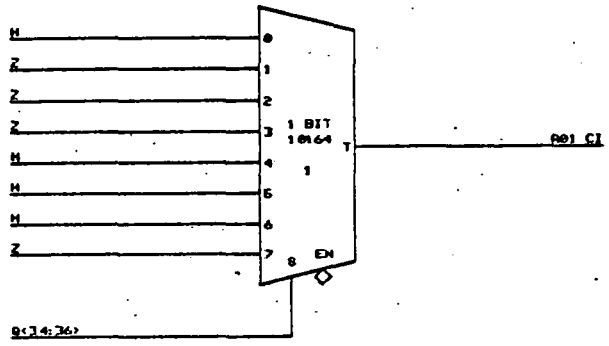
Multiply Control (MPYCTL)

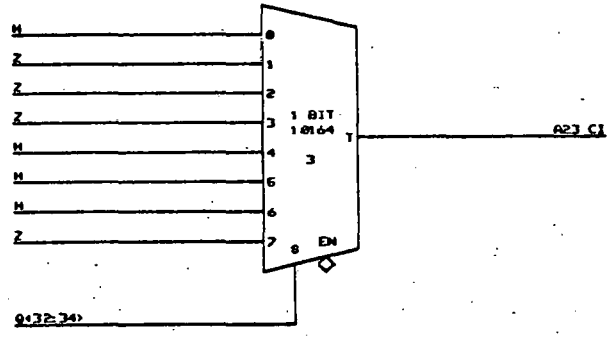


Multiply MUX Control (MMXCTL)



CSA Control (CSACTL)





Multiply CIN Control (MCICTL)

4.1.3.2.2 Shift Box

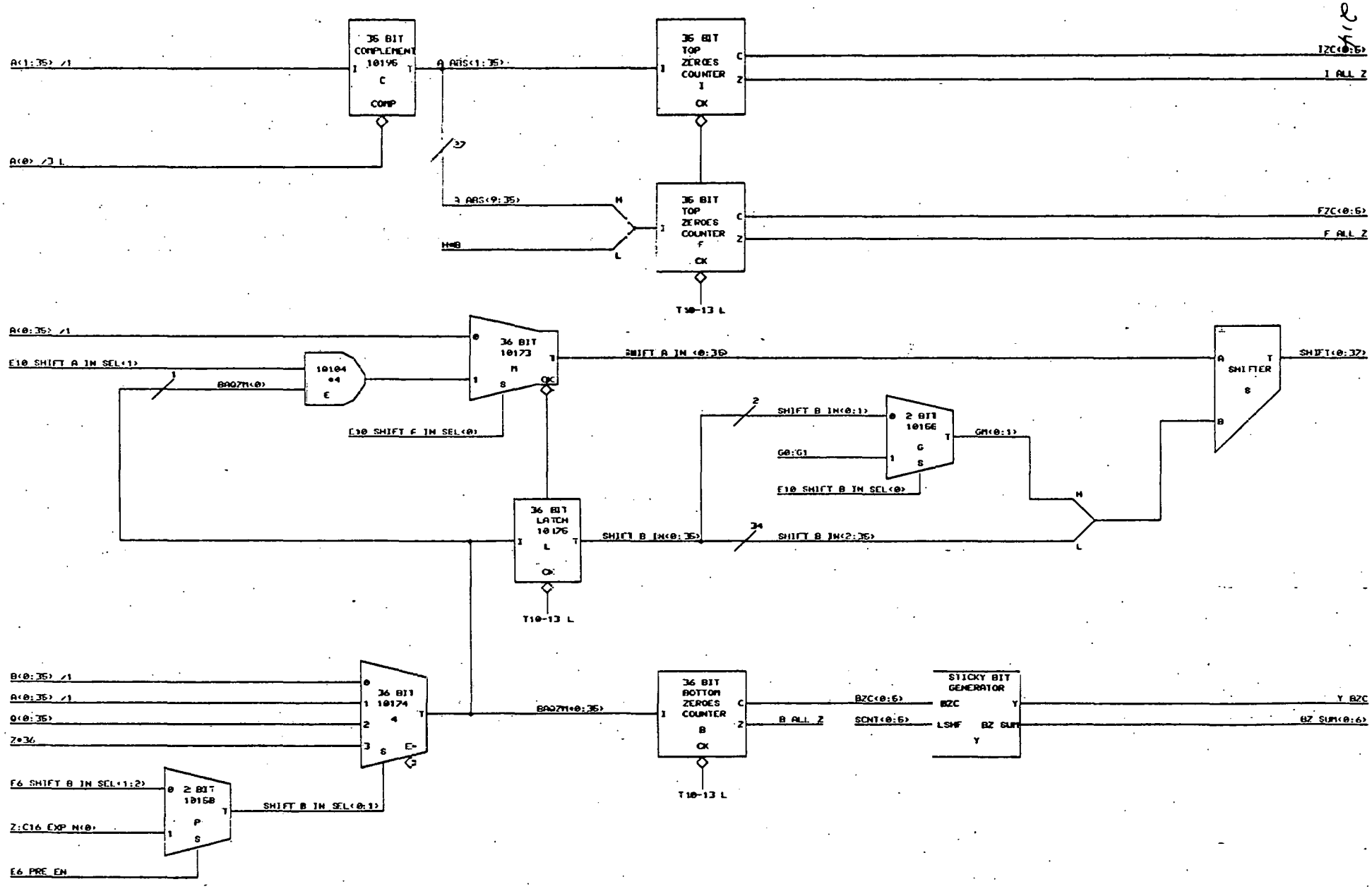
The Shift Box (SHFBOX) performs shifts in parallel with the arithmetic operation of the 3INADD. The SHFBOX can be decomposed into the Shifter (SHIFTR), the Sticky Bit Generator (STICKY), and various zeroes counters and multiplexers.

The SHIFTR takes two 36-bit input words, and can perform a left shift of 0 to 47 bits or a right shift of 1 to 16 bits onto a 38-bit output. The two low-order bits of the output become guard bits in floating-point operations. Guard bits may be merged into the SHIFTR input at the top of the low-order input word; this capability is used during floating point postnormalization.

STICKY examines the output of a zeroes counter (the 36 Bit Bottom Zeroes Counter) and determines whether all the bits lost (beyond the guard bits) in a right shift are zero; if any lost bit is a one, STICKY asserts the *sticky bit*, Y. [Kahan 1973] discusses the need for and use of the sticky bit.

Two 35 Bit Top Zeroes Counters (TZC) allow the contiguous zeroes (or ones) at the top of a floating point mantissa or an integer to be counted. The floating point count is useful during postnormalization.

A 36 Bit Bottom Zeroes Counter (BZC) counts the contiguous zeroes at the bottom of a number. This count is essential for generating the sticky bit Y.



Shift Box (SHFBOX)

4.1.3.2.2.1 Shifter

The Shifter (SHIFTR) takes two 36-bit operands as input and can shift them left 0 to 47 bits or right 1 to 16 bits, producing a 38-bit result (36 bits with two low-order guard bits for floating point operations).

The Shifter Control (SHFCTL) allows the shift count to come from various sources as follows:

- a QW3 holding register,
- a QW2 holding register,
- QW3 of the A register,
- QW2 of the A register,
- micro-constant,
- exponent ALU holding register,
- constant fields for special operations,
- top zeroes (ones) count of a mantissa, and
- top zeroes (ones) count of an integer.

In addition, many of these counts can be subtracted from 36 before being used. Subtraction of a count from 36 is necessary for simulating right shifts.

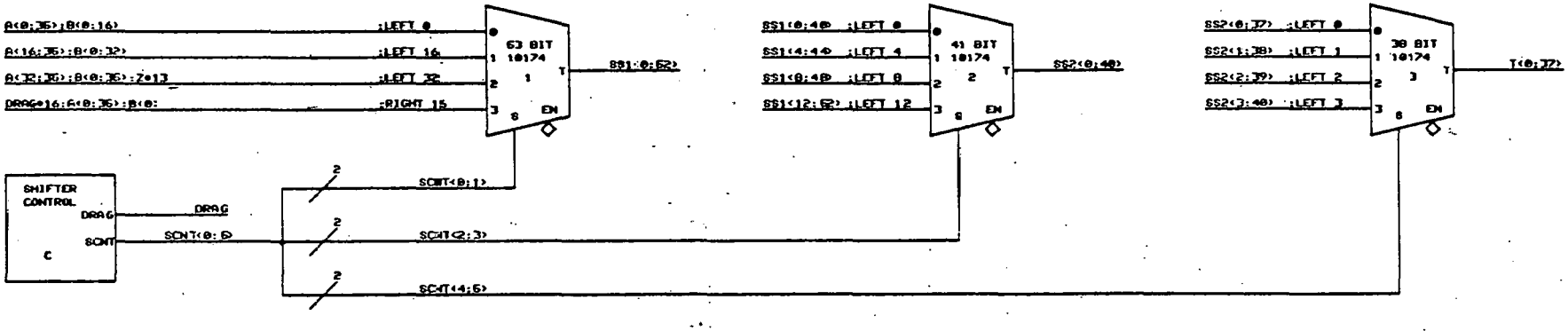
The SHIFTR is composed of three levels of multiplexers. The first level performs a shift of 0, 16 left, 32 left, or 16 right; the second level a shift of 0, 4, 8, or 12 left; and the third level a shift of 0, 1, 2, or 3 left.

COMMENT

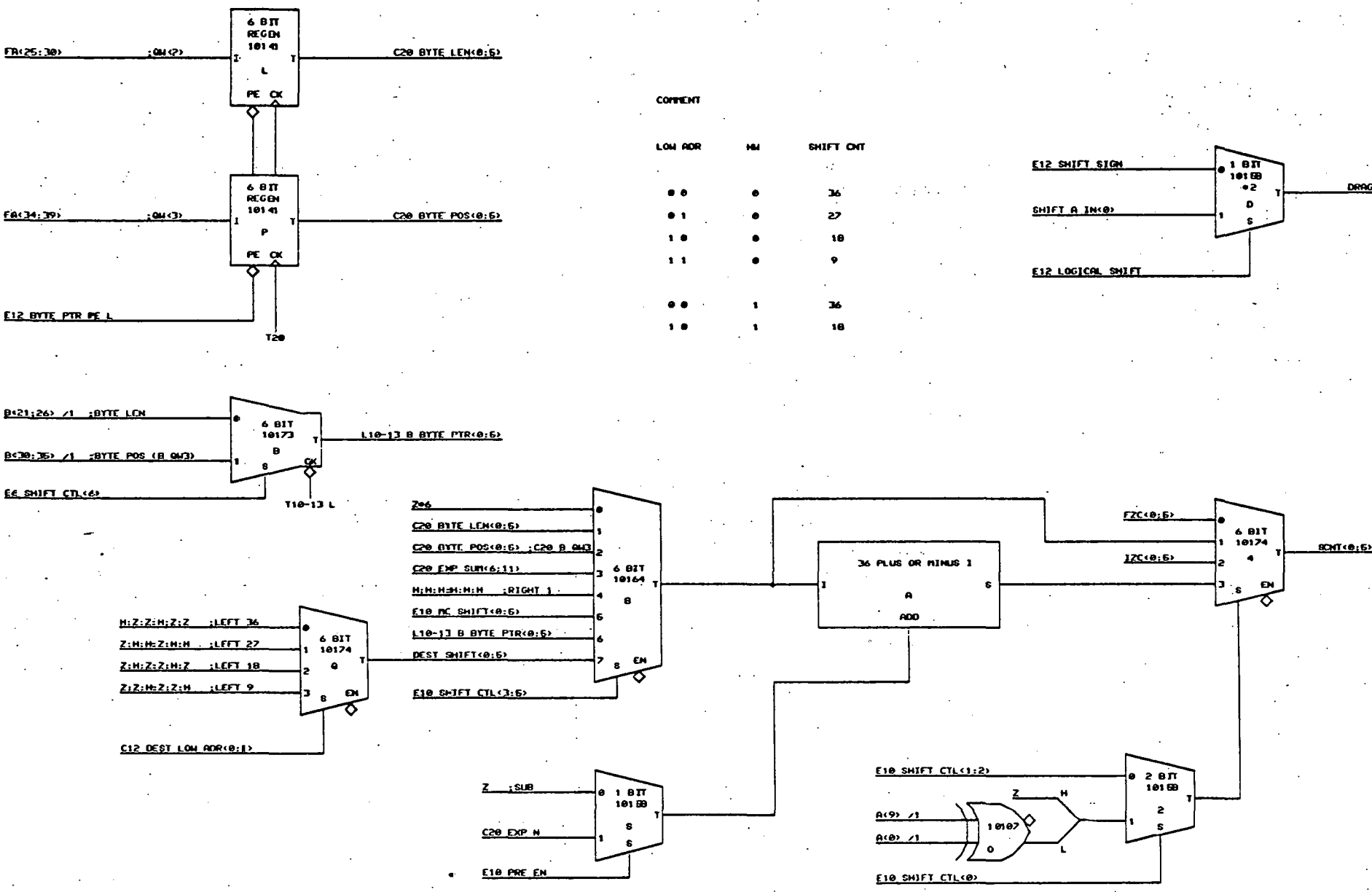
SHIFT RIGHT 1 TO 16 / SHIFT LEFT 0 TO 47

SCNT = 11 XX XX + SHIFT RIGHT 16-XXXX (1 TO 16)

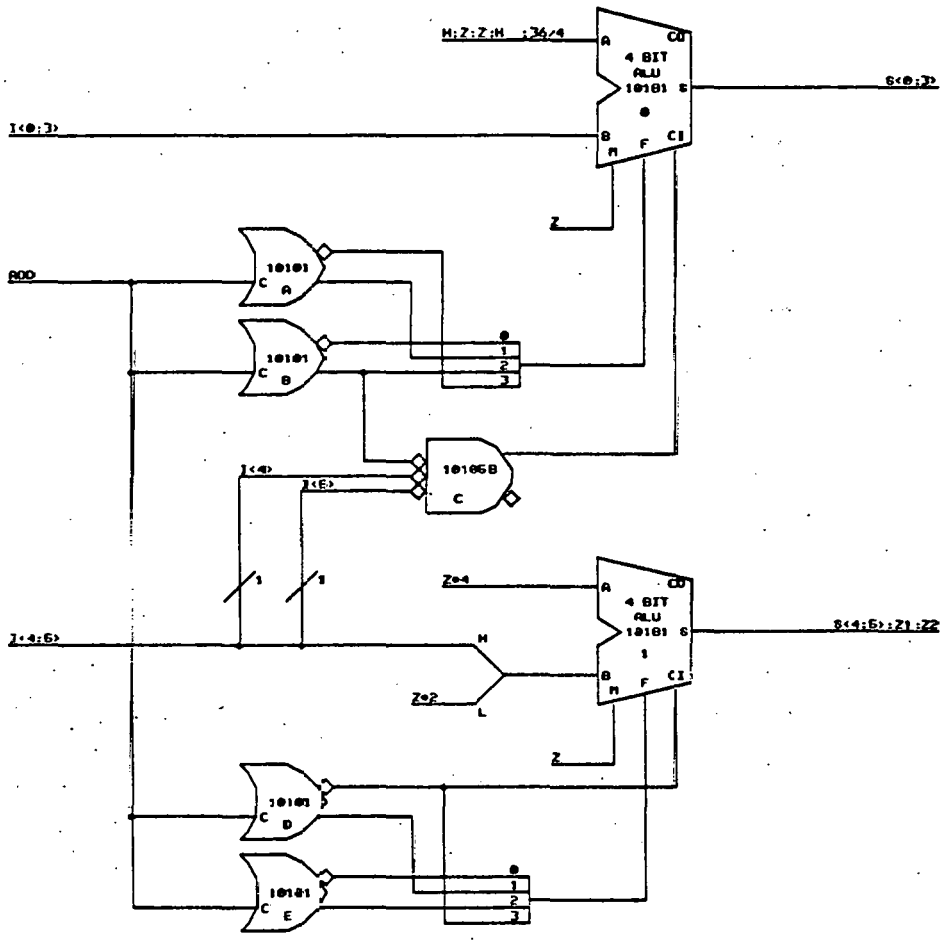
SCNT= YY XX XX (YY ≠ 11) + SHIFT LEFT YYXXXX (0 TO 47)



Shifter (SHIFTR)



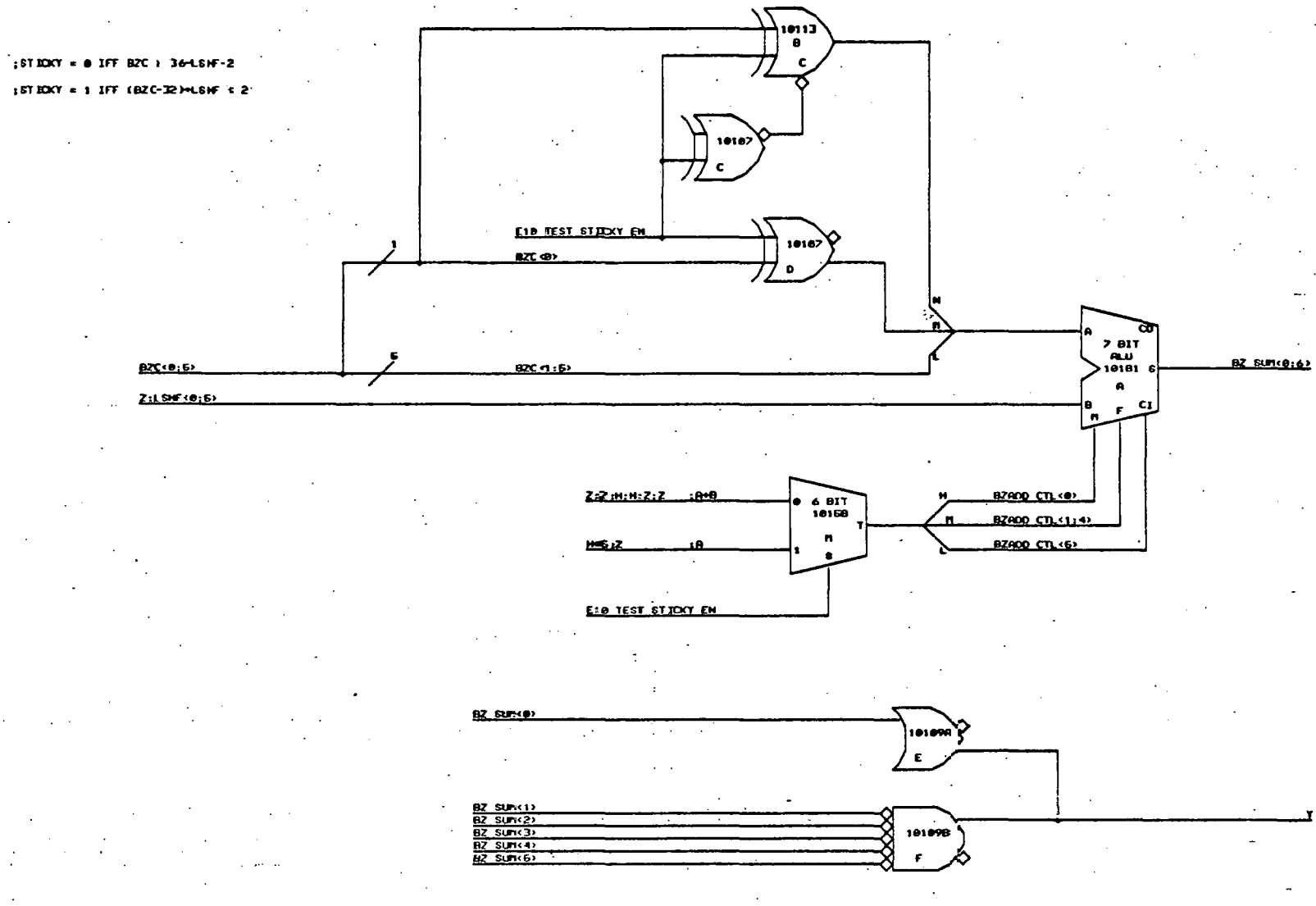
Shifter Control (SHFCTL)



36 Plus or Minus I (36PMI)

4.1.3.2.2.2 Sticky Bit Generator

The Sticky Bit Generator (STICKY) is used primarily during prenormalization of floating point numbers. During prenormalization, a number is right shifted and N bits are lost from the least-significant end. STICKY asserts the "sticky bit" if and only if the least significant N-2 lost bits are not all zero. (The most significant 2 lost bits become guard bits.) The need for and use of the sticky bit are explained in [Kahan 1973].



Sticky Bit Generator (STICKY)

4.1.3.2.3 Exponent Box

The Exponent Box (EXPBOX) performs exponent arithmetic in parallel with the operation of the EBXALU.

The exponent box receives operands from the EREGF and stores them for future use. Most floating point operations thus require a preliminary cycle in which the exponents are loaded into the EXPBOX. During the preliminary cycle, though, the QREG can be loaded. Furthermore, translations are not permitted until one cycle after the operands to be translated have been received from the IBOX.

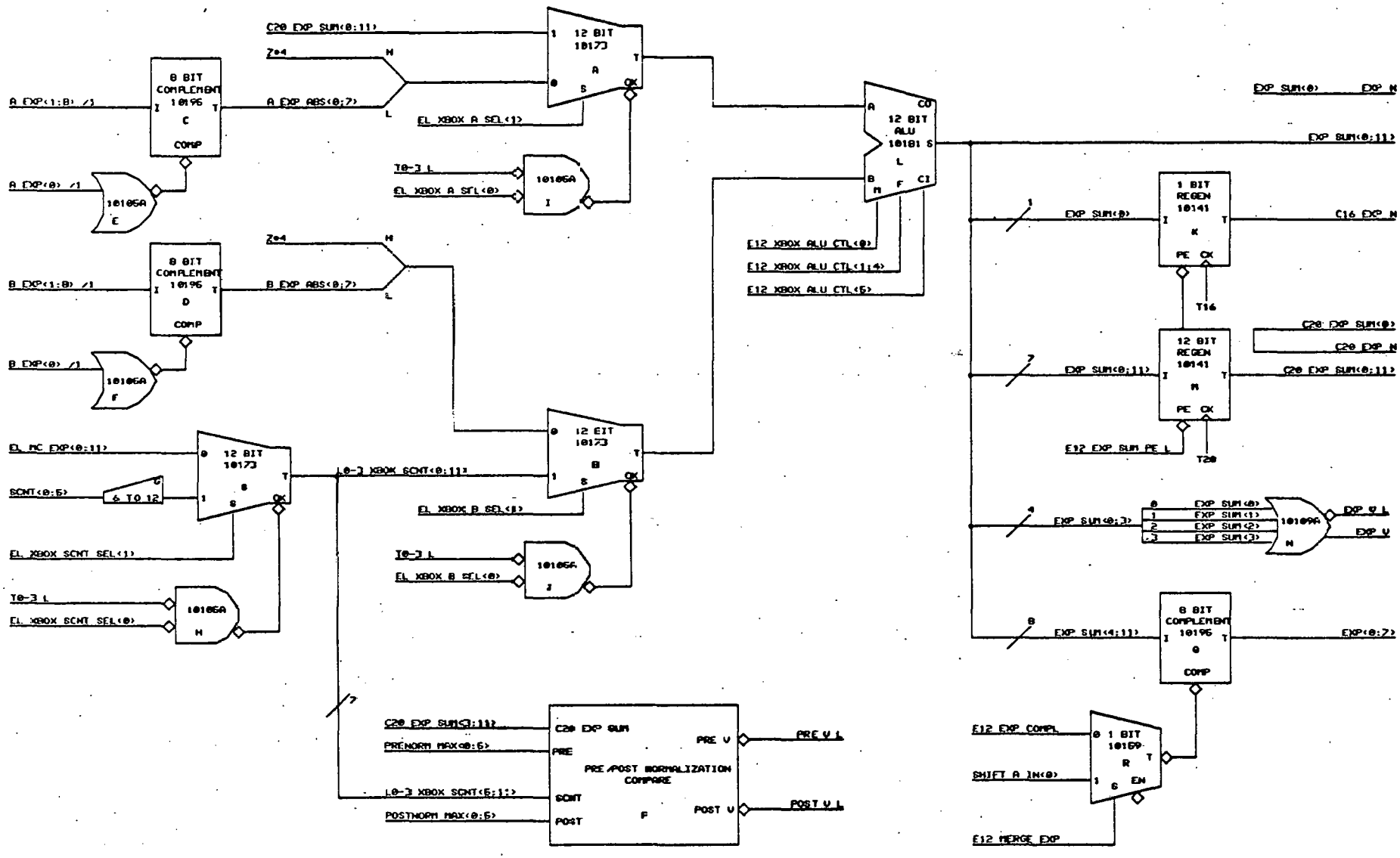
Complementers on the A and B input operands conditionally complement the exponent depending upon the sign of the mantissa (bit 0), producing the true excess-128 representation of the exponent, regardless of the sign of the floating point number.

The EXPBOX contains a 12-bit ALU which is controlled entirely by micro-code. The A leg of the ALU can come either from the A exponent complementer or from the latched ALU output. The B leg of the ALU can come either from the left shift count latched from the previous cycle, from the B exponent complementer, or from micro-code.

Since exponents in floating point numbers have only an 8-bit length, the 12-bit ALU allows exponent overflow or underflow to be carried until the last step of a floating point operation, by which time those conditions may disappear.

The output of the ALU can be saved in an output register (for input to the SHFCTL for prenormalization), or can be conditionally complemented by the sign of the input to the SHIFTR (in preparation for merging it with the SHFBOX output at the end of a floating point sequence).

The PPNCMP compares the left shift count from the SHIFTR with the postnormalization limit, and compares the ALU output register with the prenormalization limit. The signals generated by the PPNCMP are used in generating prenormalization and postnormalization error traps.



Exponent Box (EXPBOX)

COMMENT

LET D BE THE EXPONENT DIFFERENCE

-D>(PRENORM MAX L) IFF

-D>(-PRENORM MAX) -1 IFF

D<PRENORM MAX +1 IFF

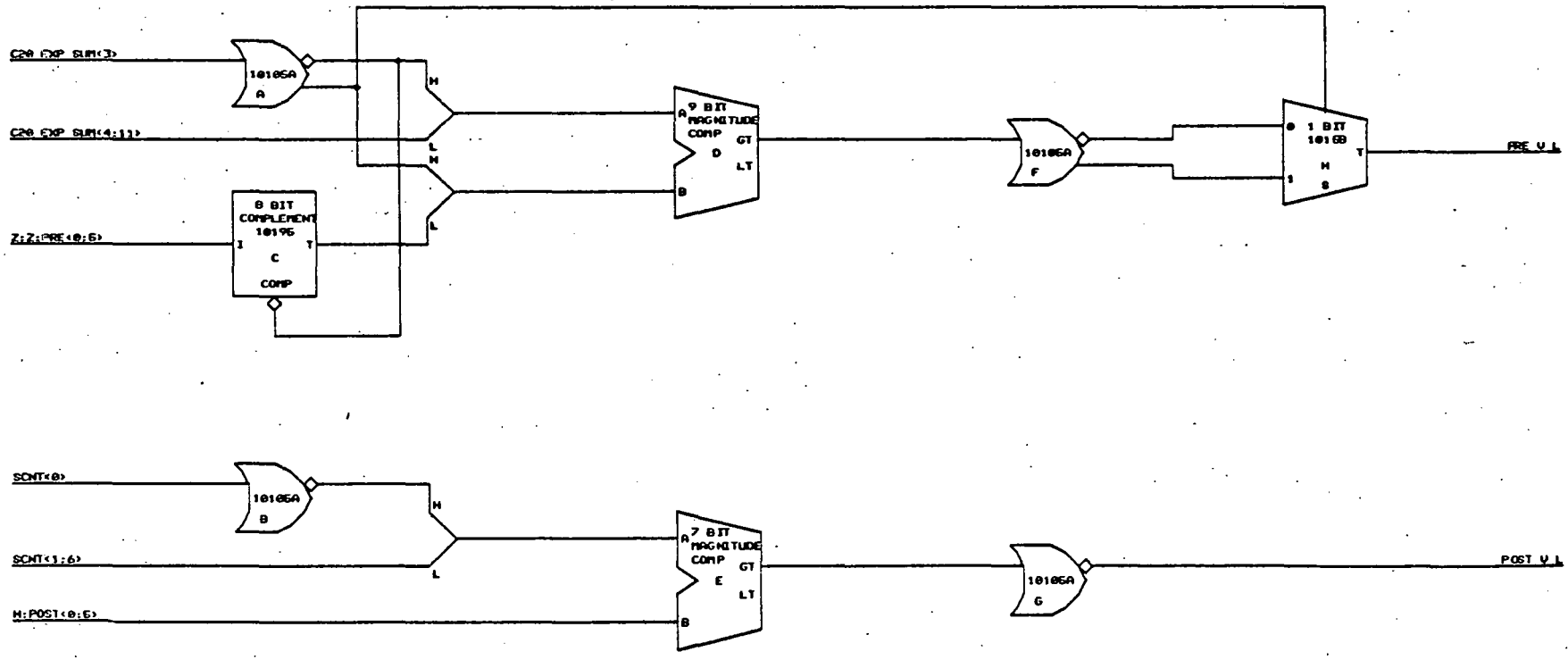
D<PRENORM MAX IFF

(D>PRENORM MAX) L IFF

PRE V L

NOTE: ADDING 64 FOR ARITHMETIC COMPARE

COMPLEMENTS THE MOST SIGNIFICANT BIT



Pre/Post Normalization Compare (PPNCMP)

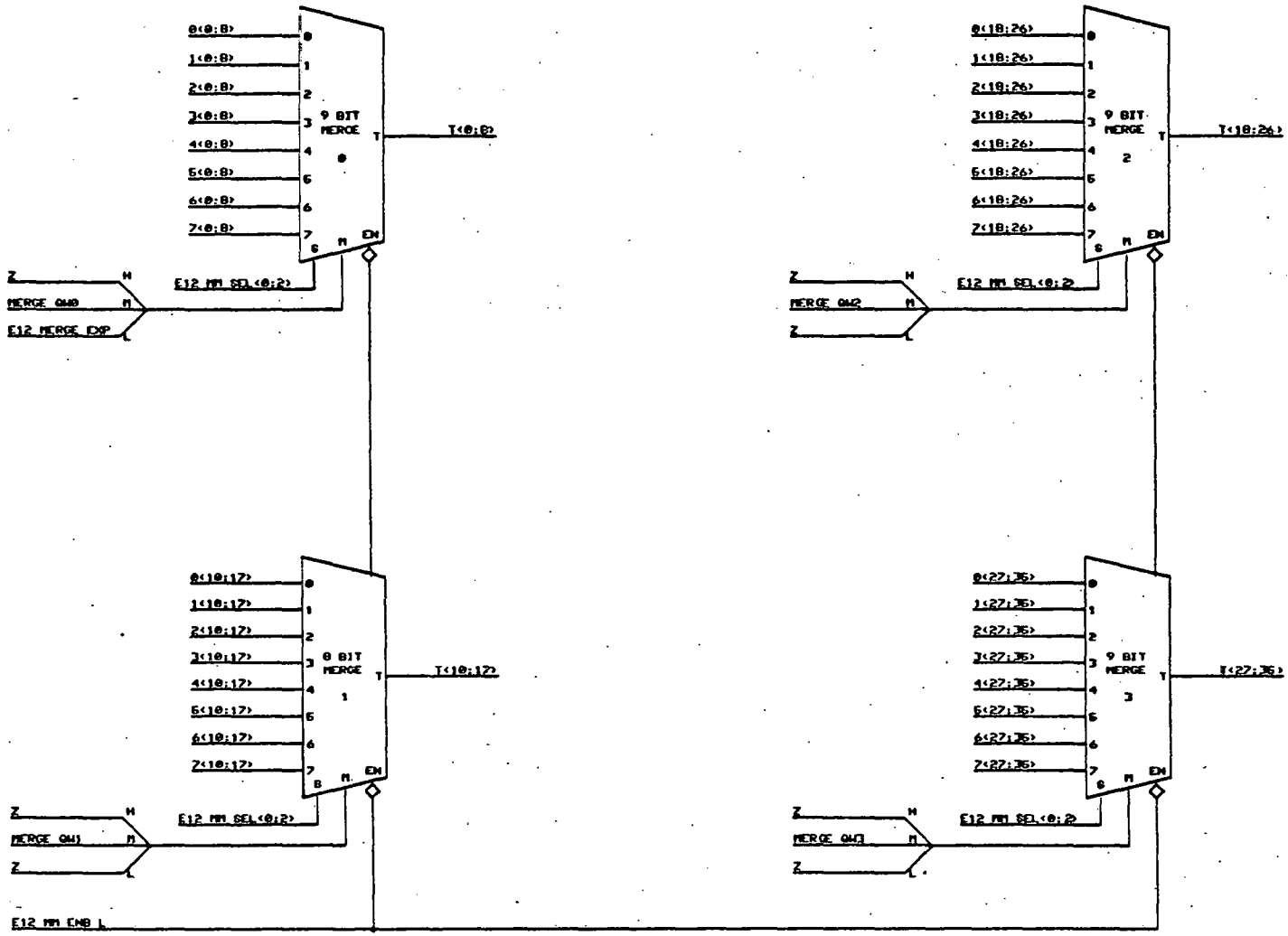
4.1.3.2.4 36 Bit MUX Merge

The 36 Bit MUX Merge (MXMRG), determines which of eight data paths is delivered as output to the EREGF or to the IBOX result register. The eight data paths are:

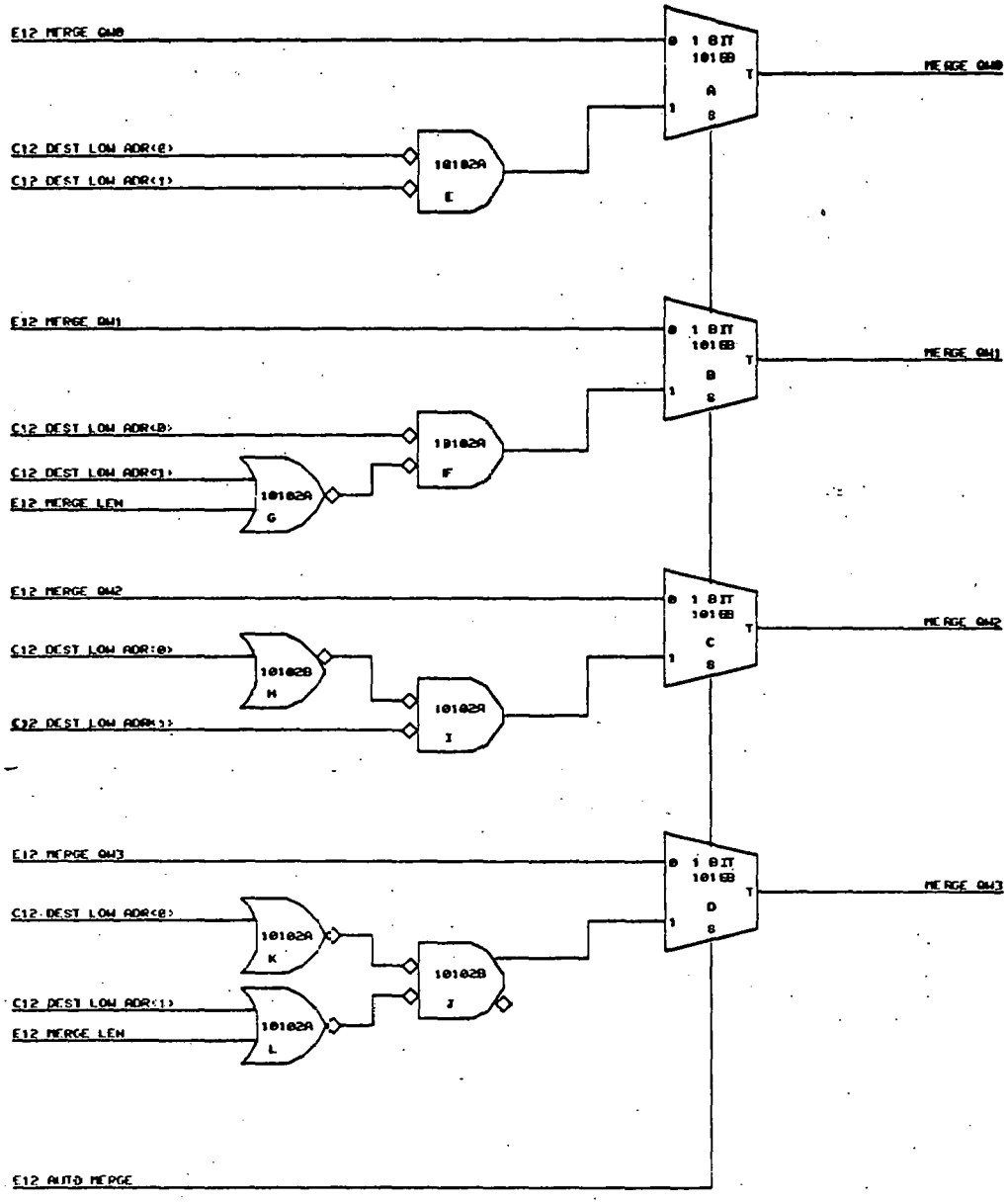
- The lower 36 bits of the output of the 3INADD shifted zero, left one, right four, or right one bit. The left-one shift is used during divide, and the right-four shift is used during multiply. The upper four bits of 3INADD are needed only during multiply operations.
- All zeroes.
- The output of SHFBOX.
- The output of EXPBOX.
- Miscellaneous fields from the EBOX.

The MXMRG also allows selective merging of each quarter-word of the SHFBOX with the output of the 3INADD. This capability can be controlled entirely by micro-code, in which case the micro-code can select the source of each output quarter-word independently, or by the address bits of the destination, which are supplied by the IBOX. Merging according to the address bits of the destination is necessary for quarter-word and half-word operations in which the result must be shifted into place and merged into the destination word.

The MXMRG also allows the exponent path to be merged with the output of the SHFBOX for producing final floating point results. In this case, the sign-extended mantissa comes through the SHFBOX and is merged with the exponent.



36 Bit MUX Merge 1/2 (MXMRG1)



36 BIT MUX Merge 2/2 (MXMRG2)

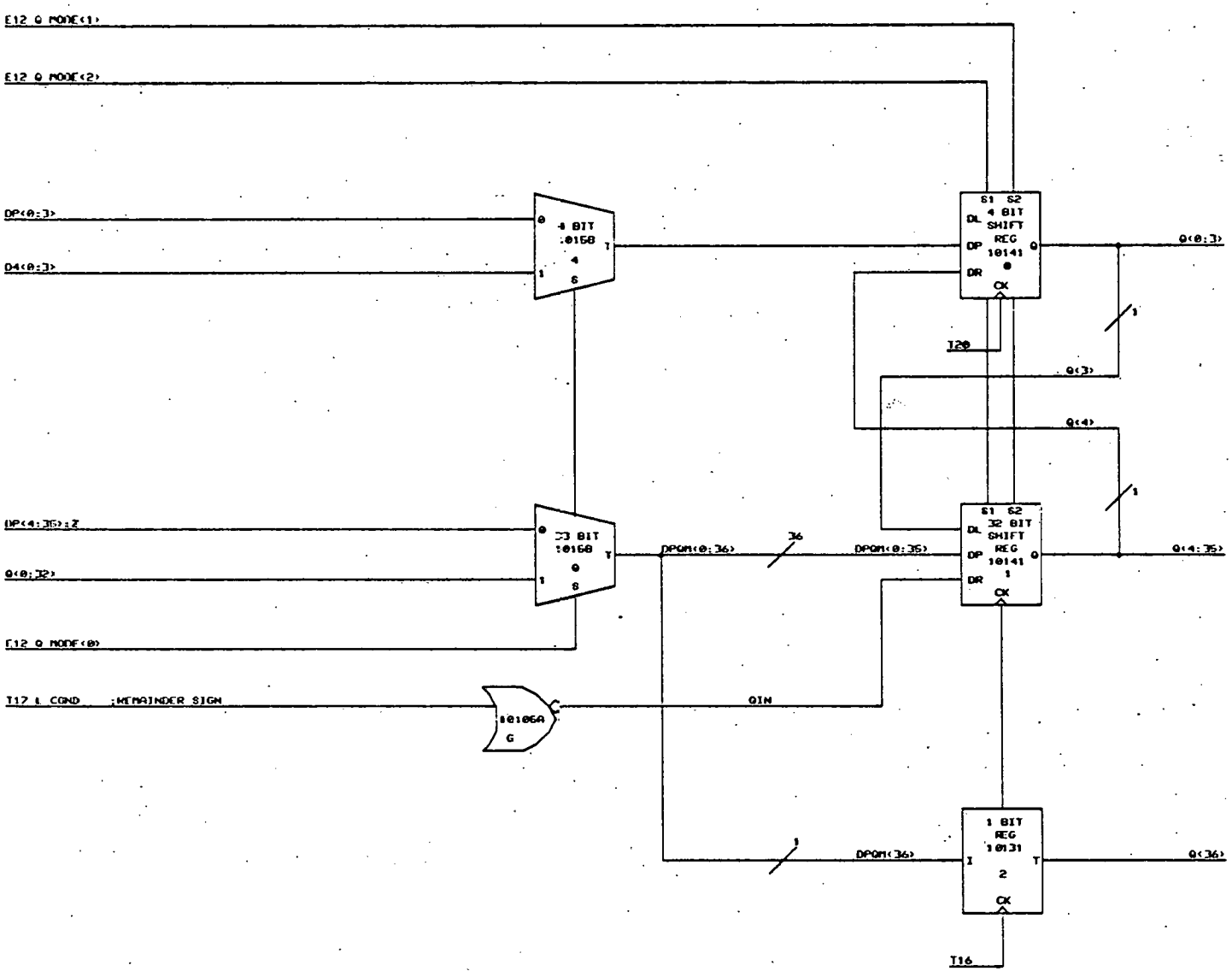
4.1.3.2.5 Q Register

The Q Register (QREG) is a 37-bit shift register (36 bits plus carry out of the least-significant bit) which is used to perform multiplication and division, and which also serves to hold temporary values. During multiplication, the QREG holds the multiplier, and during division the QREG holds the dividend.

The QREG is built of ECL 10141 universal shift registers. It has the capability to parallel load, shift right four, shift right 1, shift left one, or hold, all under micro-code control. The right-four shift is used during multiplication, and the left-one shift is used during division, as follows:

- Shifting right by 4. During multiplication, the QREG is initially loaded with the multiplier. The EBOX uses a multiplication algorithm that examines the multiplier and produces four bits of the product each micro-cycle. Each micro-cycle the QREG parallel loads from itself, moving the higher 33 bits into the lower 33 bits. This is physically equivalent to shifting right by 4. The 4 most significant bits loaded into the QREG are the 4 least significant bits coming out of the ALU. During a multiply these are the 4 least significant bits of the current partial product. After the last cycle, the QREG contains the low-order word of the product.

- Shifting left by 1. During division the Q register is initially loaded with the dividend. Each instruction cycle one new bit of the quotient is shifted into the least significant bit of the Q register.



Q Register (QREG)

4.1.3.3 EBOX Control

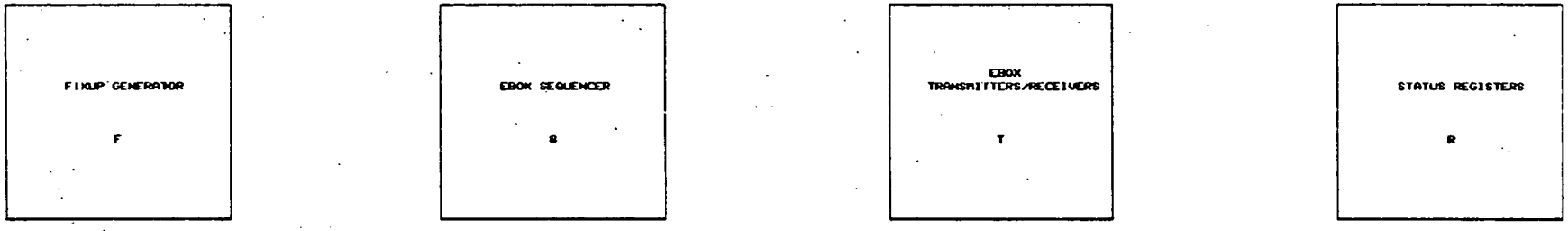
The EBOX Control (EBXCTL) includes all control logic and miscellaneous logic. It can be decomposed into the EBOX Sequencer (ESEQ), the Fixup Generator (FIXGEN), the Status Registers (STATUS), and the EBOX Transmitters/Receivers (EXCVR).

The ESEQ provides all sequencing control.

The FIXGEN produces the *fixup* signal. During some operations, such as floating point add, the cycle which is normally the last execution cycle may, in rare instances, generate a condition that requires further processing. In that case, the FIXGEN raises the fixup signal at the last possible instant, causing the EBOX to lose one cycle before continuing with the operation. If fixup is not asserted, then the operation will complete without wasting any cycles. This fixup capability allows conditions generated during the current execution cycle to affect the flow of control, without requiring that the next cycle be wasted to test conditions.

STATUS contains processor and user status registers.

The EXCVR handles receiving and transmitting most IBOX/EBOX communication signals.



EBOX Control (EBXCTL)

4.1.3.3.1 EBOX Sequencer

The EBOX Sequencer (ESEQ) controls the sequencing of the EBOX. The major components of the ESEQ are the 12 Bit Branch Address Merger (BRADRM), the EBOX Branch Condition MUX (EBCMUX), and the EBOX Control Store (EBXCS).

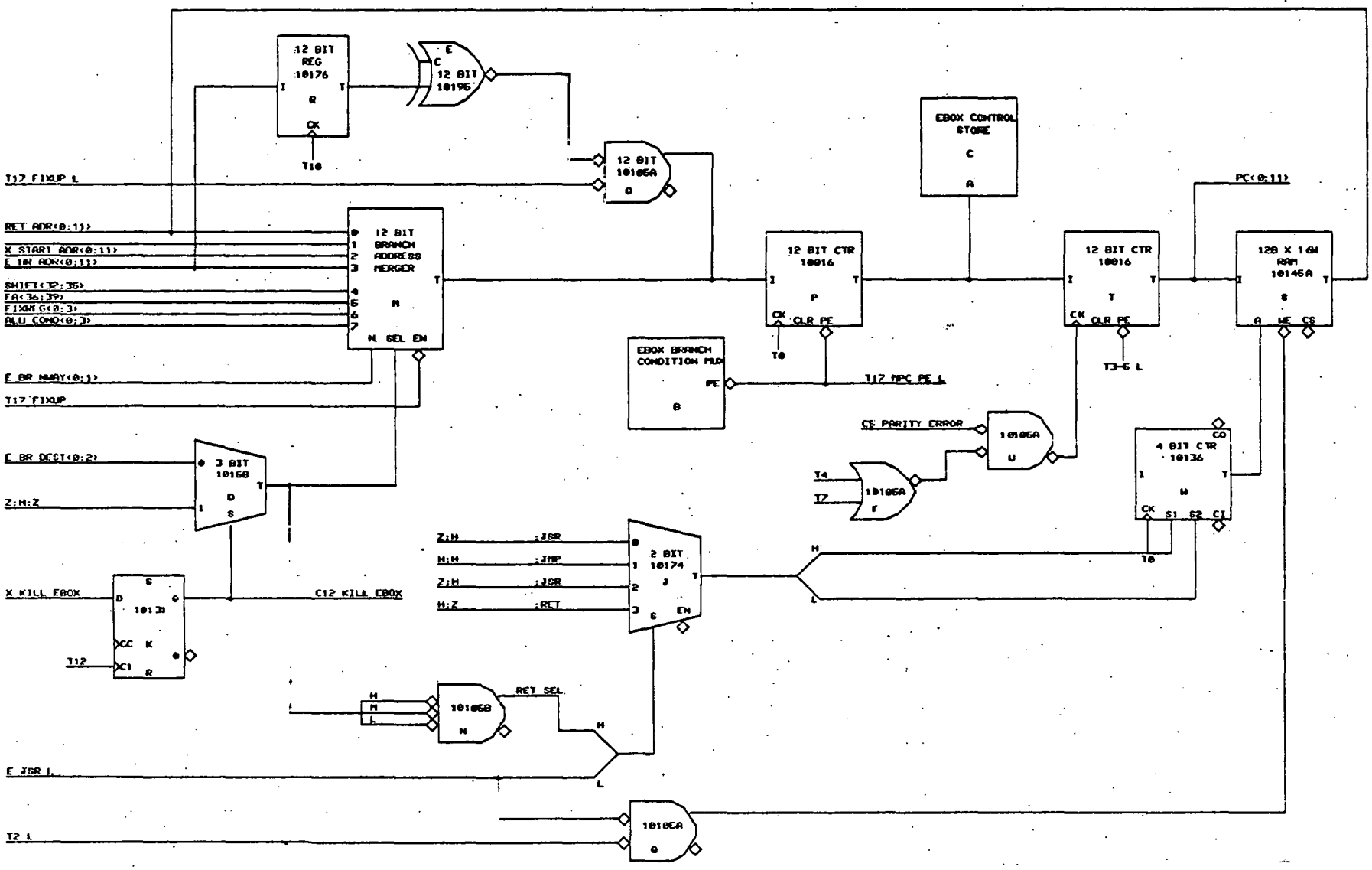
The BRADRM determines the source of the next micro-instruction. The possible micro-instruction address sources include a micro-subroutine return address, the IBOX-provided macro-operation starting address, and the micro-code branch address. Since micro-instructions are read out a full cycle before use, BRADRM must be set up approximately 1.25 cycles early.

The BRADRM allows an N-way branch (N = 2, 4, 8, or 16) on the low-order SHFBOX output, the low-order 3INADD output, the FIXREG output, or the conditions generated in the CBOX.

When FIXGEN asserts the fixup signal, a special branch address is forced into the micro-program counter to initiate a fixup sequence one full cycle later.

The control logic in the ESEQ allows any address input to the BRADRM to be used for a jump or a jump to subroutine.

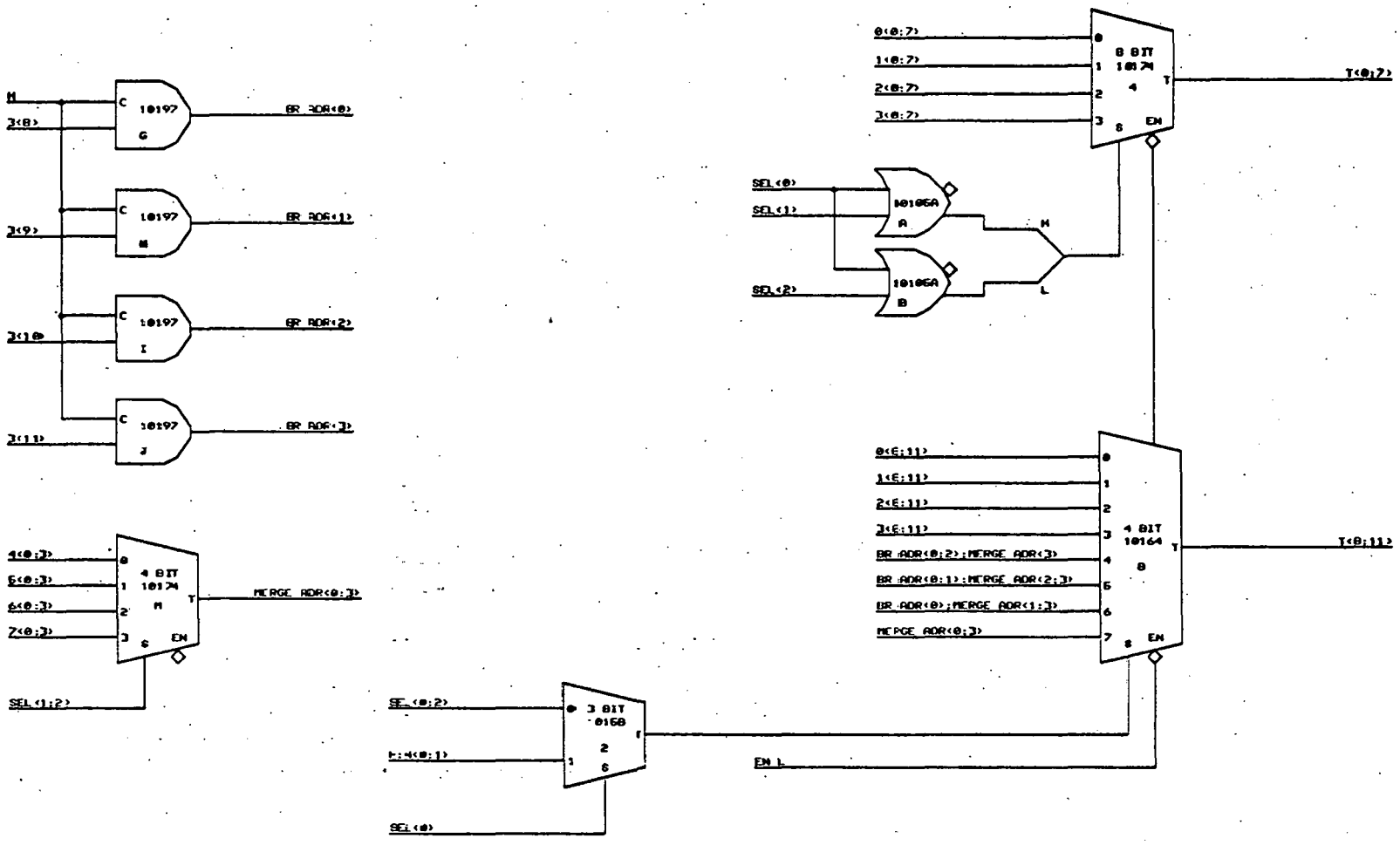
The EBCMUX determines whether the branch condition being tested by the micro-code is true, and if so, allows the micro-program counter to be loaded, otherwise the micro-program counter increments.



EBOX Sequencer (ESEQ)

4.1.3.3.1.1 12 Bit Branch Address Merger

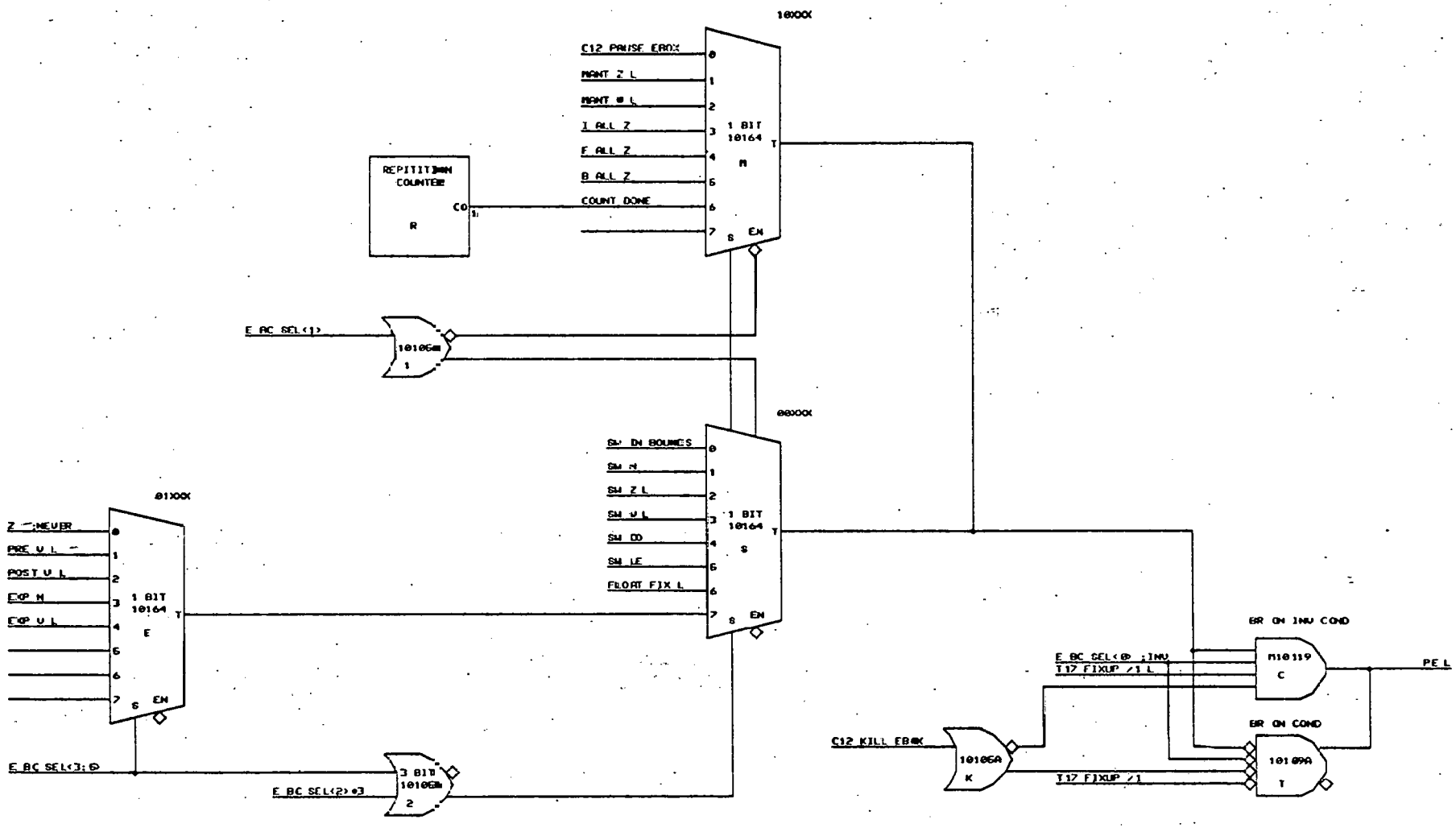
The 12 Bit Branch Address Merger (BRADRM) allows N-way (N = 2, 4, 8, or 16) branches on the value of any of four four-bit vectors. Depending upon N, the selected four-bit vector is shifted into place and substituted for the low-order bits of the branch address from the microcode.



12 Bit Branch Address Merger (BRADRM)

4.1.3.3.1.2 EBOX Branch Condition MUX

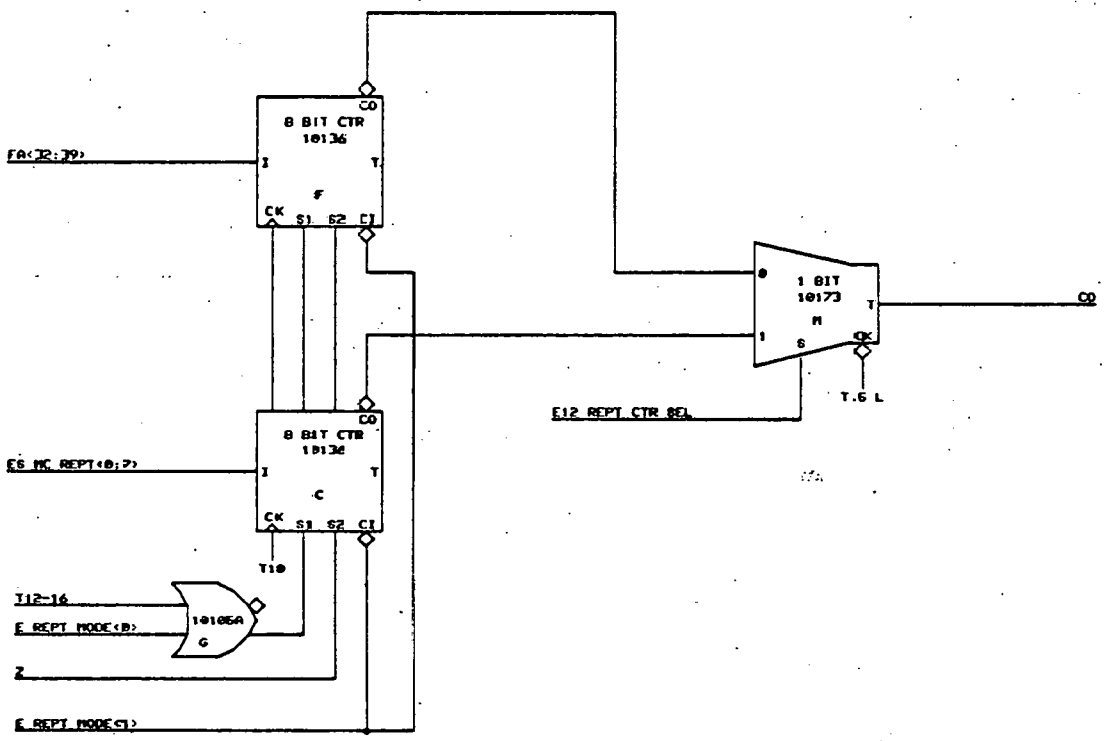
The EBOX Branch Condition MUX (EBCMUX) asserts the parallel load line on the micro-program counter if and only if the condition selected by the micro-code is true. EBCMUX allows any of 24 conditions to be tested, and allows those conditions to be inverted before testing. Testing of conditions for branching cannot be done during the cycle that the tested conditions are generated, but must be done during the next cycle, since the micro-program counter is loaded one cycle before execution commences.



EBOX Branch Condition MUX (EBCMUX)

4.1.3.3.1.2.1 Repitition Counter

The Repitition Counter (REPT) allows the micro-code to contain "FOR" loops. REPT can be loaded from either the 3INADD or the micro-code, and can be counted down and tested under micro-program control. REPT thus allows control constructs in the micro-code such as "branch if zero (non-zero) then decrement", and "branch if zero (non-zero) then load".



COMMENT

REPITION COUNTER ALLOWS:

MICROWORD SOURCE —

BRANCH IF ZERO/NON-ZERO THEN LOAD

BRANCH IF ZERO/NON-ZERO THEN DECREMENT

DATA PATH SOURCE —

SET UP DATA IN DATA PATH

BRANCH IF ZERO/NON-ZERO THEN LOAD

BRANCH IF ZERO/NON-ZERO THEN DECREMENT

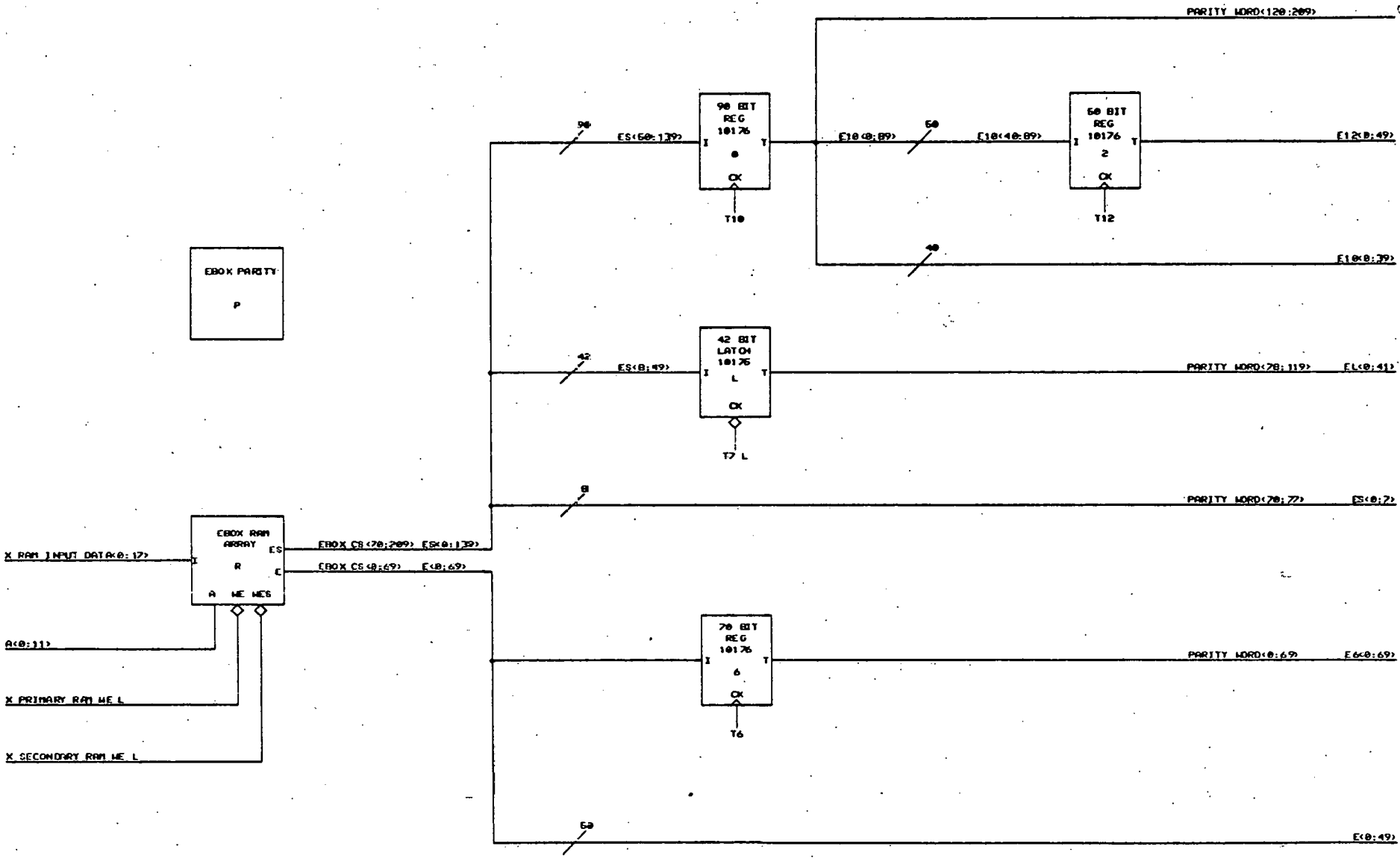
Repitition Counter (REPT)

4.1.3.3.1.3 EBOX Control Store

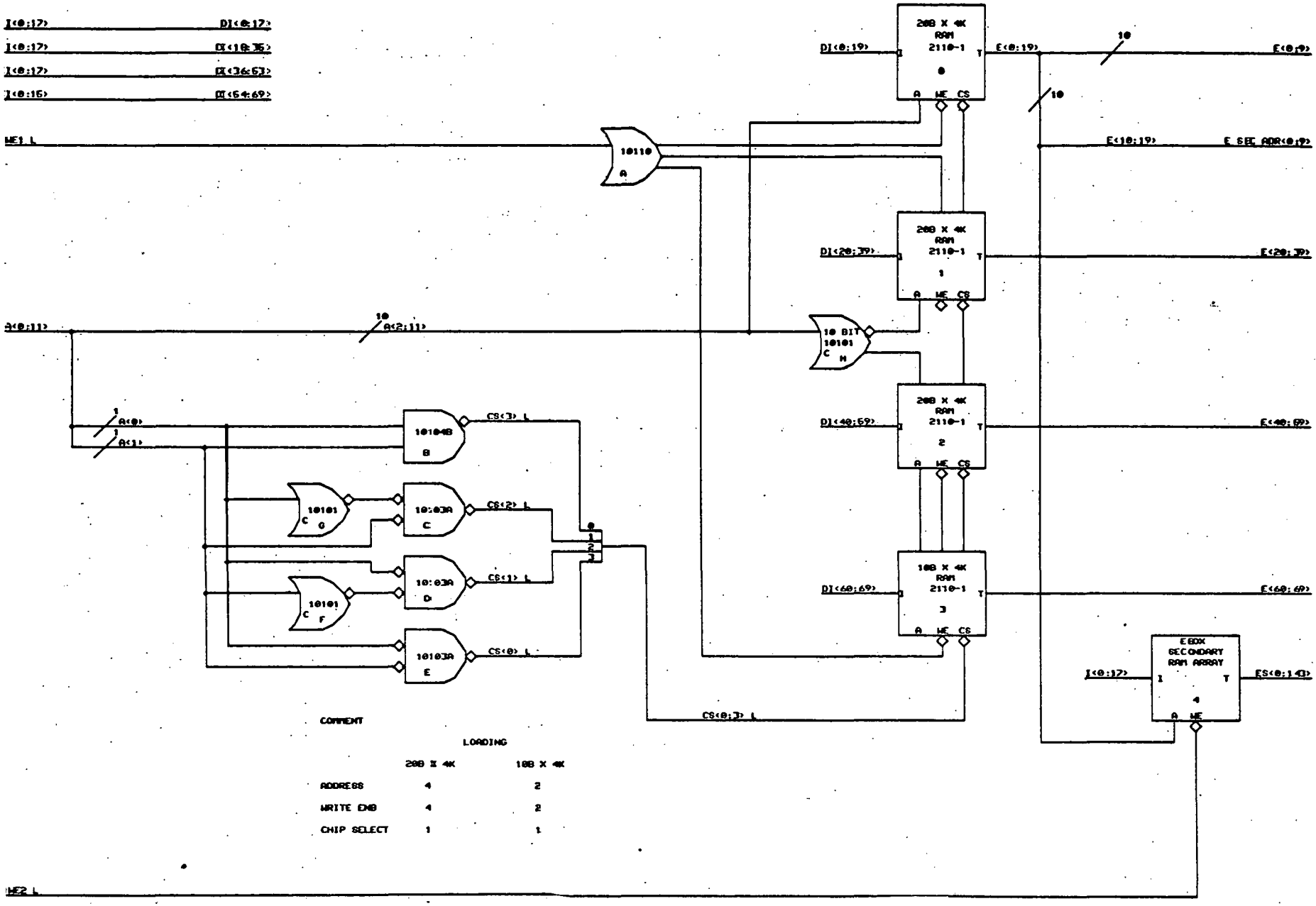
The EBOX Control Store (EBXCS) contains the EBOX writeable control store, various micro-instruction pipeline registers, and EBOX Parity (EBXPAR).

Control store for the EBOX is two-level for reasons of economy. The first level is addressed by the micro-program counter; it is 4K words deep by 70 bits wide. Ten bits of the output of the first level become the address bits for the second level, which is 1K words deep by 140 bits wide. In general, signals which are needed long before the micro-instruction execution commences must be located in the first level, and signals which are not needed until the execution starts can be located in the second level. This two-level control store allows the *sharing* between micro-instructions of subparts of common control words. With the aid of an intelligent micro-code assembler, the control store appears to be uniformly 4K words deep.

EBXPAR checks the parity of control store words and raises an error signal if a parity violation is detected.



EBOX Control Store (EBXCS)

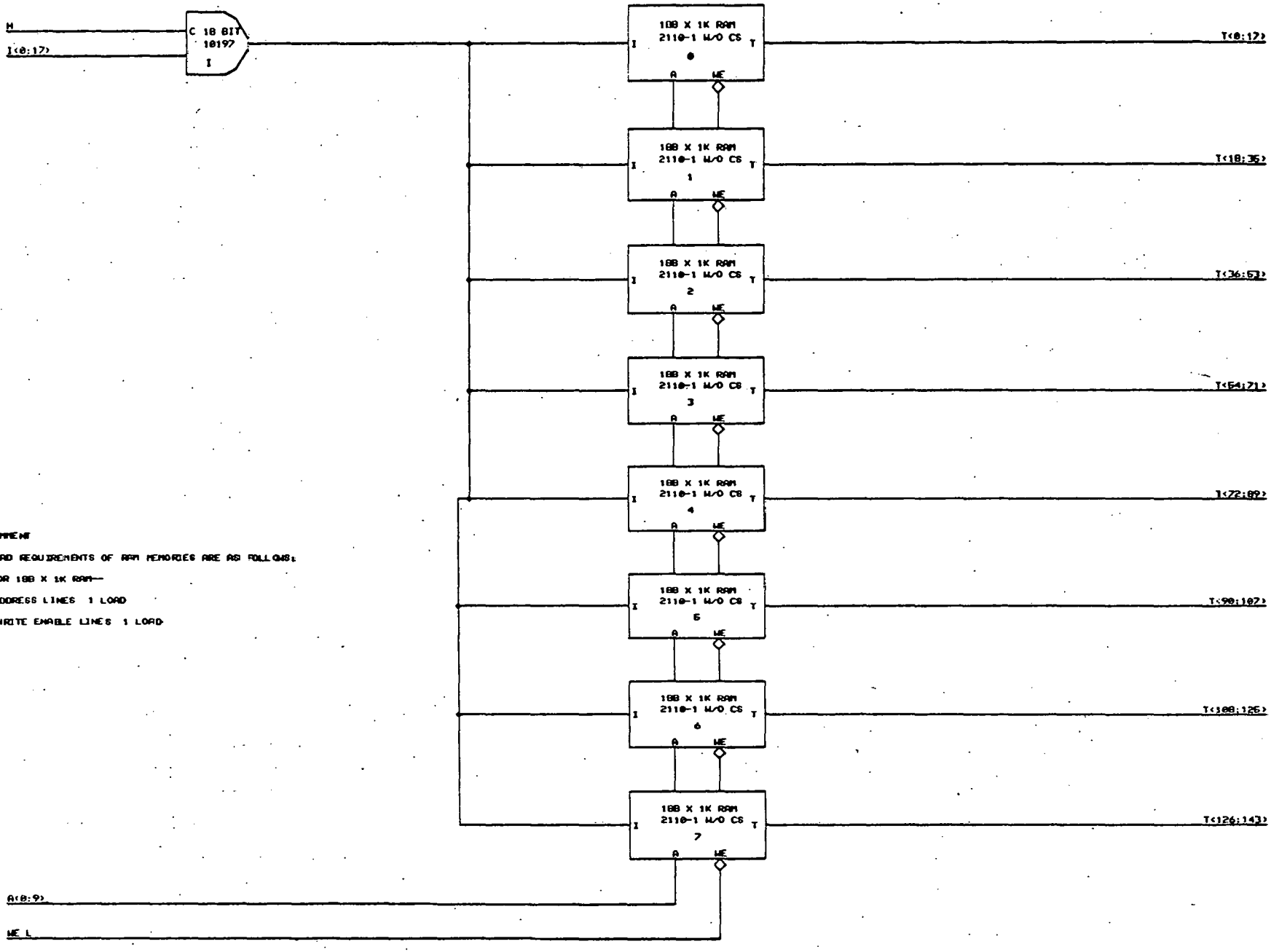


COMMENT

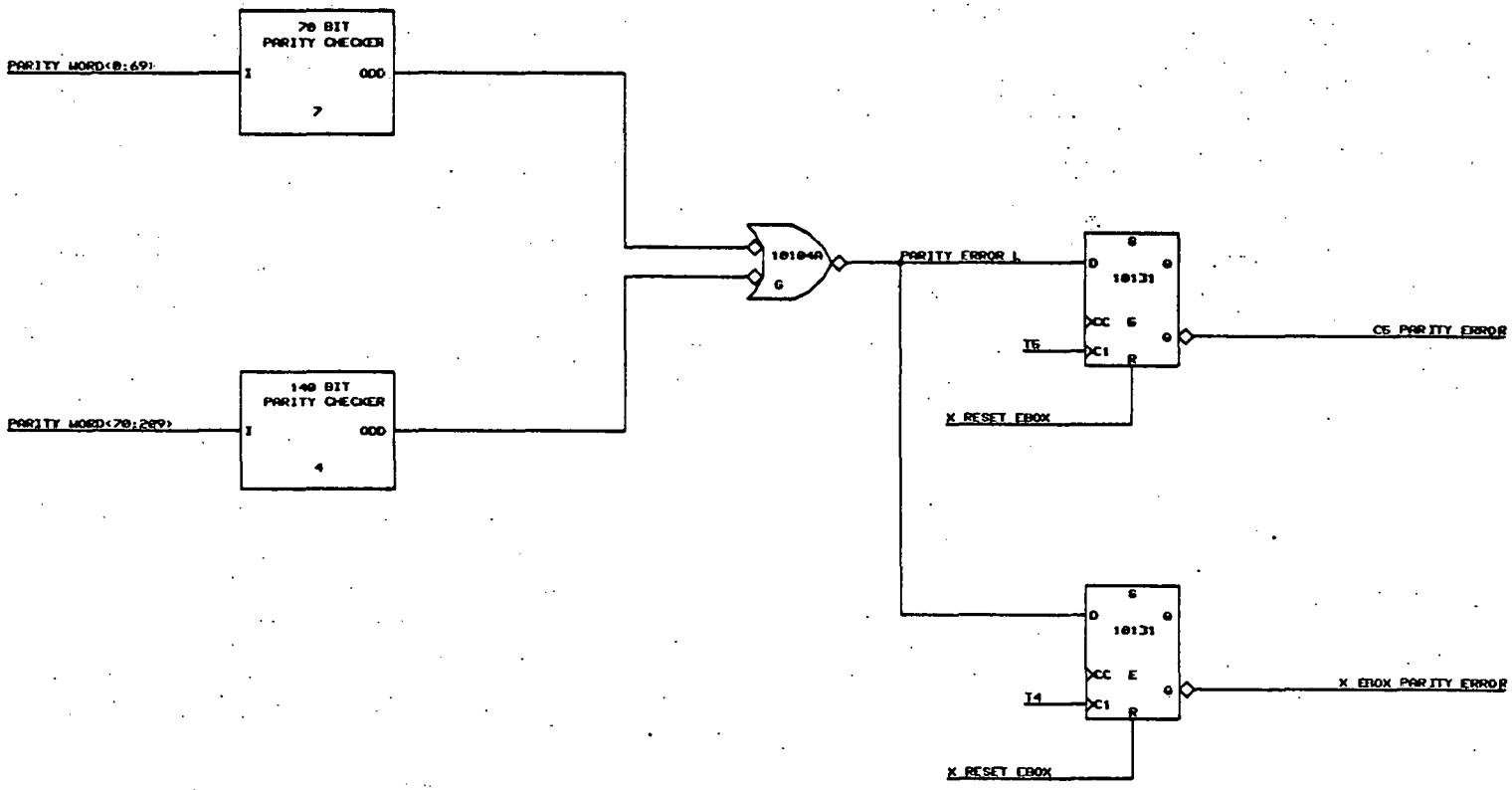
LOADING

	200 X 4K	100 X 4K
ADDRESS	4	2
WRITE ENB	4	2
CHIP SELECT	1	1

EBOX RAM Array (EBXRAM)



EBOX Secondary RAM Array (EBXSEC)



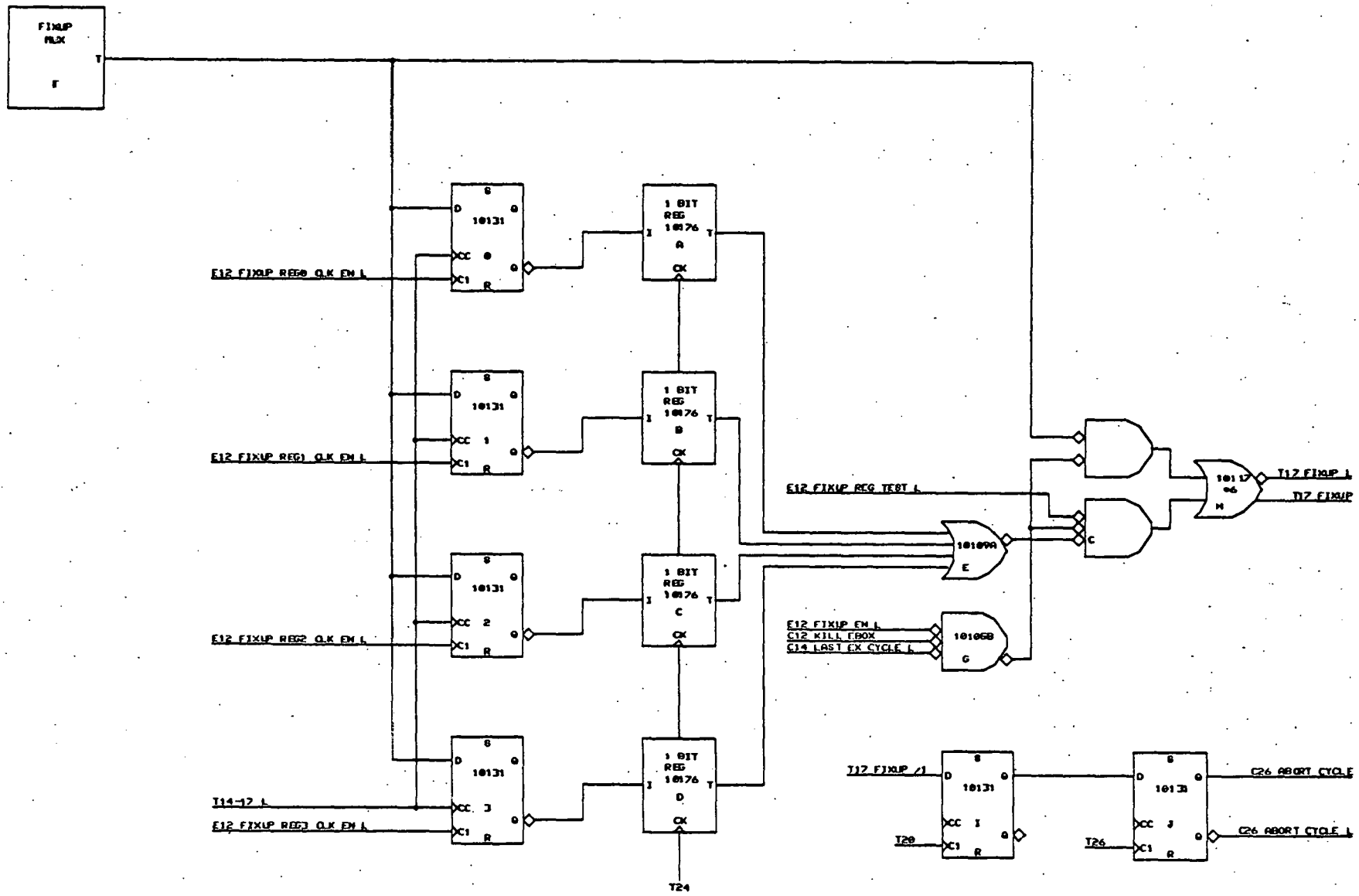
EBOX Parity (EBXPAR)

4.1.3.3.2 Fixup Generator

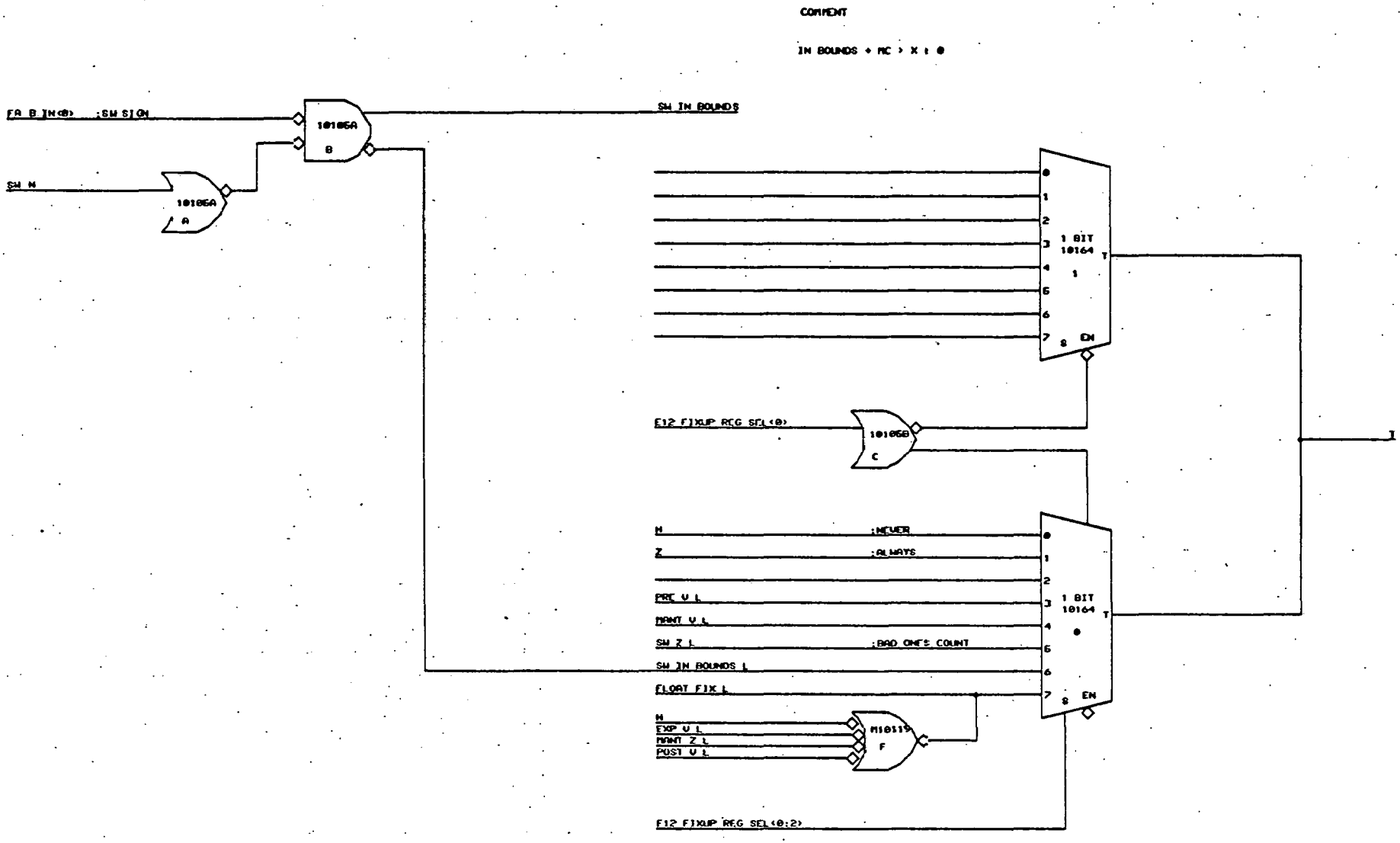
The purpose of the Fixup Generator (FIXGEN) is to sometimes assert the *fixup* signal and cause the EBOX to continue with the fixup micro-instruction sequence instead of starting a new operation sequence under command of the IBOX.

During execution cycles in the interior of a micro-instruction sequence, FIXGEN can store detected fixup conditions in any of four 1-bit registers, and can use the contents of those registers to assert fixup on the (tentatively) last cycle.

The Fixup Multiplexer (FIXMUX) multiplexes the fixup condition chosen by the micro-code. The output of the FIXMUX can be used to cause fixup during the current cycle, or can be stored for use later.



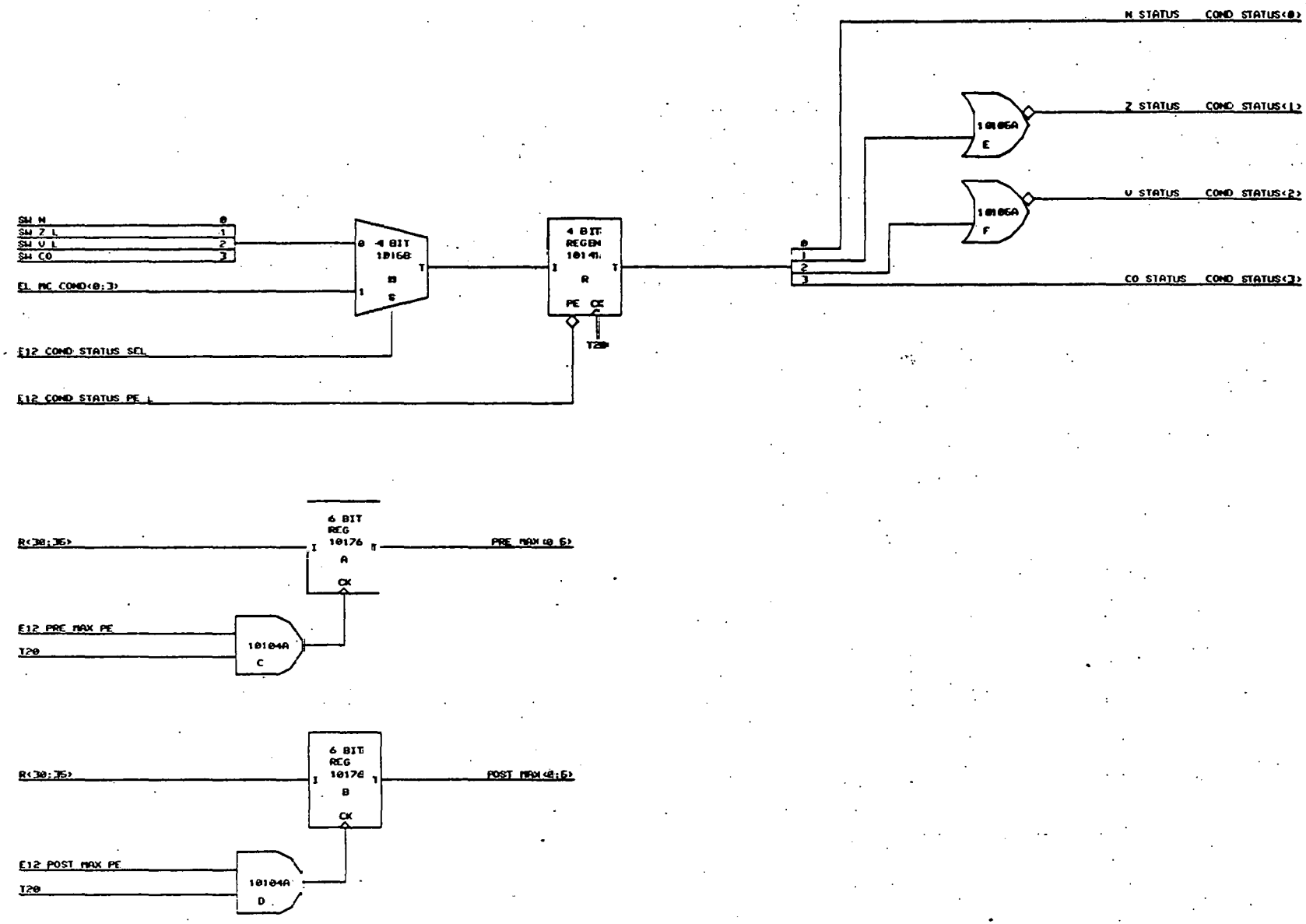
Fixup Generator (FIXGEN)



Fixup MUX (FIXMUX)

4.1.3.3.3 Status Registers

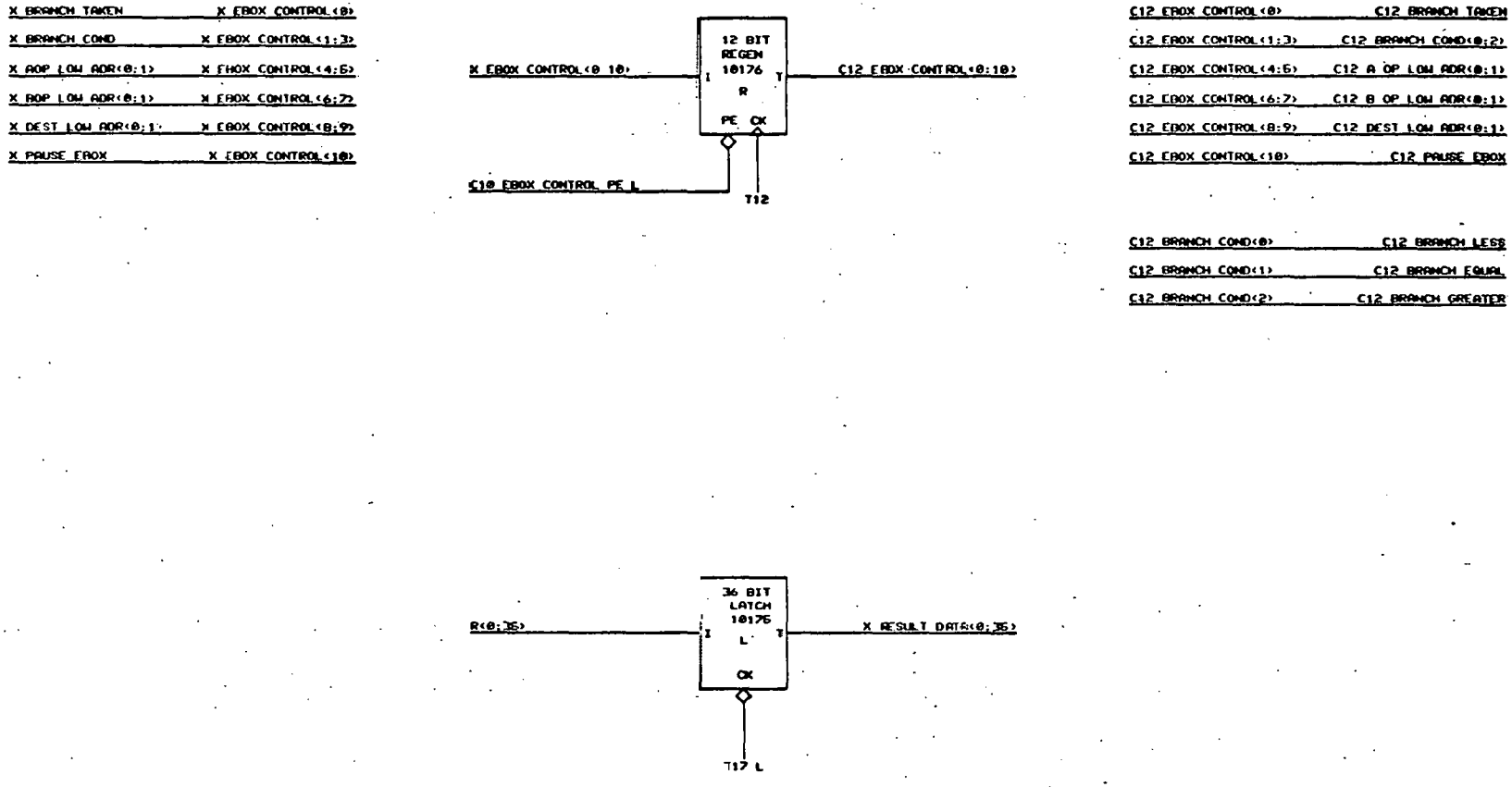
The Status Registers (STATUS) contains the processor and user status registers. These registers can be conditionally loaded under micro-program control.



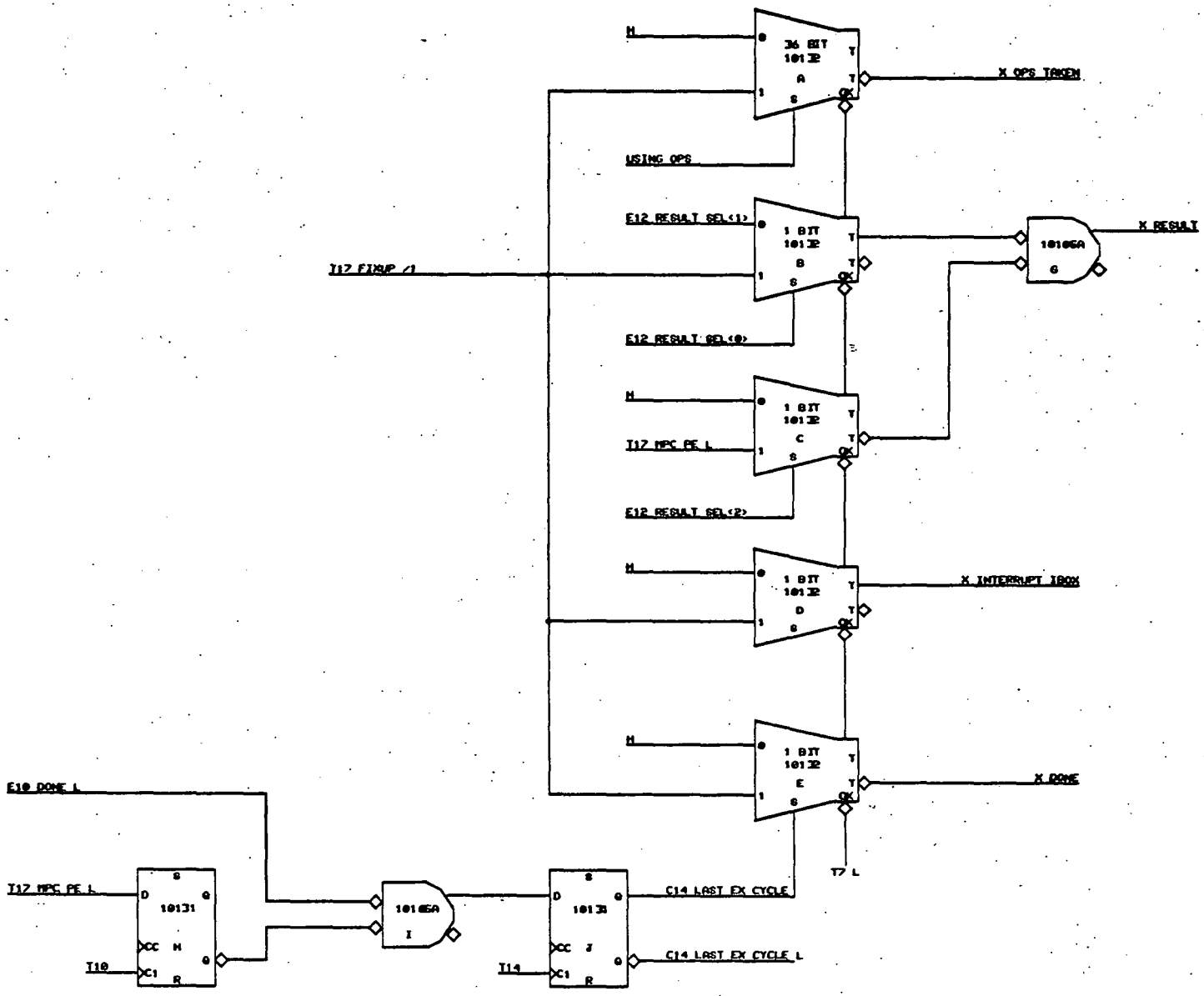
Status Registers (STATUS)

4.1.3.3.4 EBOX Transmitters/Receivers

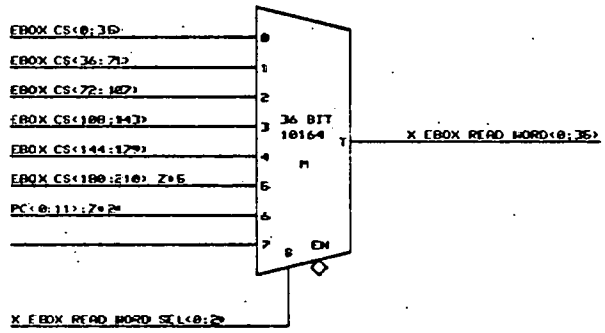
The EBOX Transmitters/Receivers (EXCVR) receives signals from the IBOX and transmits signals to the IBOX. Depending upon the fixup signal generated by FIXGEN, EXCVR will conditionally assert OPS TAKEN, RESULT, INTERRUPT IBOX, and DONE on the last execution cycle of a sequence.



EBOX Transmitters/Receivers 1/3 (EXCVR1)



EBOX Transmitters/Receivers 2/3 (EXCVR2)

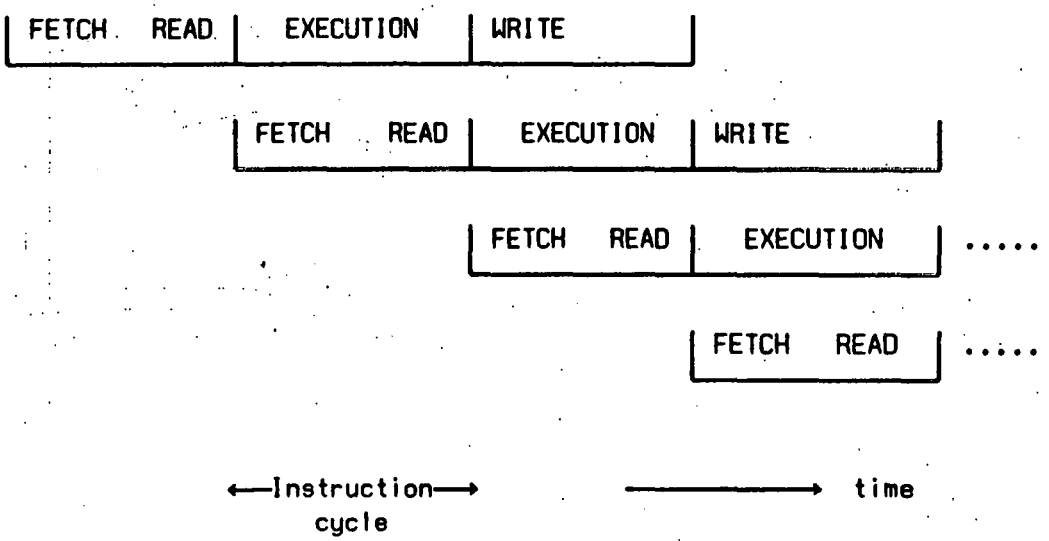


EBOX Transmitters/Receivers 3/3 (EXCVR3)

4.1.3.4 Timing

The EBOX is controlled by the IBOX, which specifies the operation and the operands for the EBOX. The IBOX provides the EBOX with the address of the first micro-instruction in the EBOX's control store. The EBOX performs the operation by executing the sequence of instructions from its control store beginning at the address specified by the IBOX. At the beginning of the last micro-instruction cycle of an operation, the EBOX raises the DONE flag. In response, the IBOX prepares the next address and operands of the first instruction of the next operation. This section describes the timing of a normal macro-operation.

A macro-operation consists of a sequence of micro-instructions as shown:



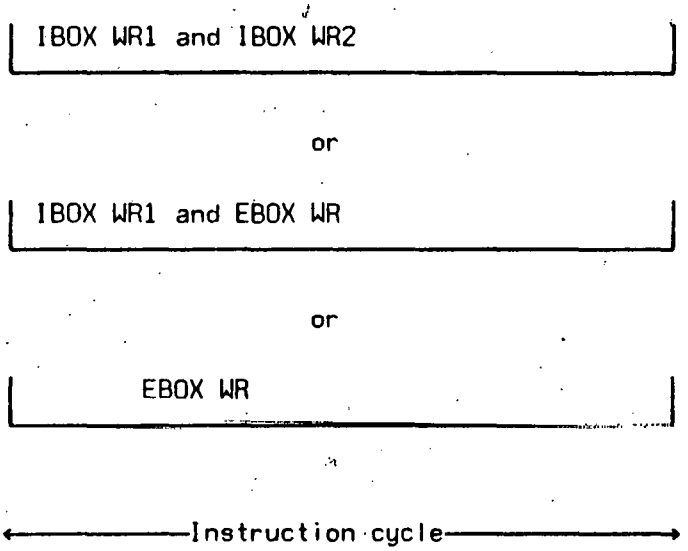
Sequential micro-instructions overlap; during a given instruction cycle, three operations occur in parallel:

1. During FETCH, the EBOX fetches the next micro-instruction from its control store and places it in the pipeline register.

2. During READ, the EBOX reads operands from its EREGF.

3. During EXECUTION, the EBOX executes the current micro-instruction from the pipeline register. The ALU produces a result by the end of the execution cycle. If the DONE bit of the micro-instruction is set, the DONE flag is raised at the beginning of the cycle.

During WRITE, either the IBOX or the EBOX may write into the EREGF.



The purpose of an IBOX write is to provide the operands for the next macro-operation. During the first half-cycle, the IBOX writes operand A and B into the same address of the two register banks. The register location written into is determined by the EBOX.

During any instruction in which the IBOX is not providing operands, or is providing only one operand, the EBOX may write data into its EREGF. The EBOX write also occurs during the first half-cycle.

At the end of an execution cycle, the result:

- is always available to be used as an operand for the next execution cycle, and
- is simultaneously written into the EREGF during the next execution cycle (unless two operands are received from the IBOX for the next execution cycle).

4.2 Interconnection Network

The processors are connected to memory by a serial/parallel crossbar interconnection switch (See Figure 2.1-1). Data is transmitted 24 bits at a time through the switch, taking two cycles per data word transmitted. Once it is through the switch, it is then transmitted fully word parallel to the memory's, since the relatively slow TTL logic in the memory's can not handle the high speed of the switch.

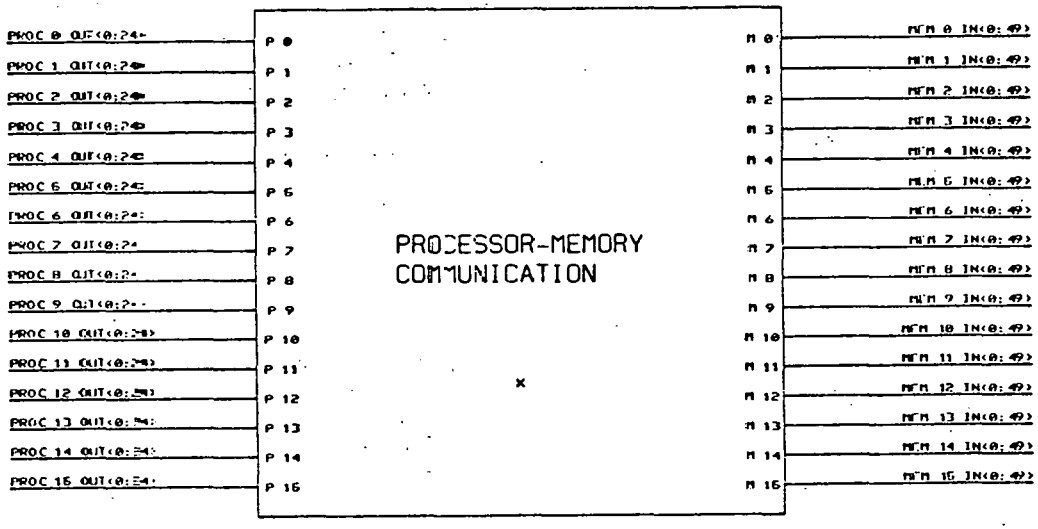
The memory is divided into 16 Block Storage Modules (BSMs). The BSMs are interleaved 4 ways on the low order bits of the real address word. When a processor does a read or write, four words are transmitted, except in cases where the data is tagged as not cacheable, in which case only one word is transmitted. Normally the address is transmitted once and the two low-order bits are permuted in order to obtain the addresses of four consecutive words in memory.

With N processors, the common store resembles an n-port memory because of the interconnection network, the structure of which allows each processor to simultaneously and independently access different BSMs. When two or more processors try to access the same BSM, the conflict is resolved by the memory contention control logic. This logic ensures that no processor can access a BSM twice before another processor desiring access can access it once. This effectively solves the deadlock problem which plagues some multiple processor systems, in which a higher priority processors locks out lower priority processors for an indefinite period of time.

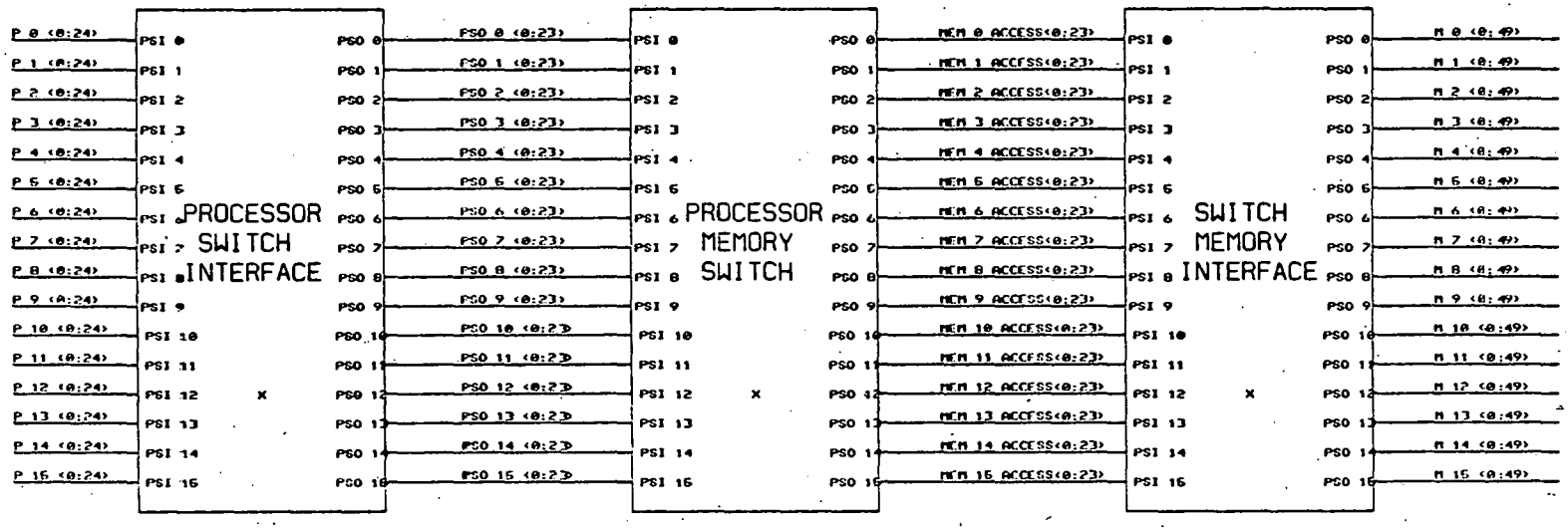
Each BSM has its own memory contention logic, the inputs of which are the request lines from each processor and the outputs of which are the select lines of the interconnection network. The request lines are activated by control logic monitoring the address lines of each processor. In a sixteen processor system, four of the address lines would be input to a 4-to-16 line decoder. The 16 output lines would indicate which of the 16 BSMs the processor desires to access.

As soon as a particular BSM becomes idle, the memory contention logic latches the 16 processor request lines for that BSM. It then proceeds to service the queued processors until the memory is again idle. The 16 output lines of the latch go to a 16-to-4 line priority decoder which determines which one of the processors is to be serviced first. The output lines of the priority encoder are connected to a latch, and to the select lines of the interconnection network, which routes the data from the selected processor to the BSM selected. At the end of a memory cycle when one processor has been serviced, the latch is released and the request is cleared.

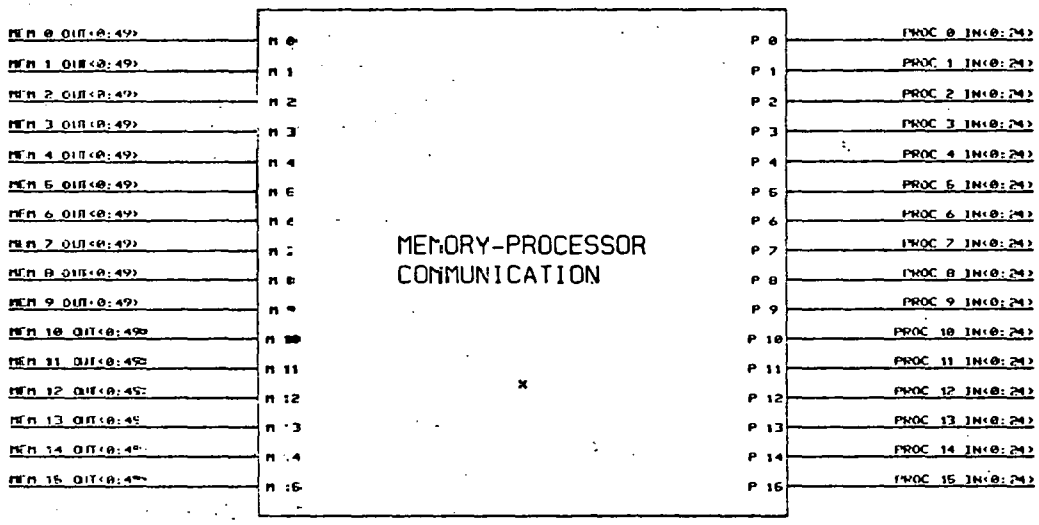
The priority encoder then elects the next processor to be serviced on the basis of the new data in the latches. This cycle continues until the latch is empty and all processors have been serviced. At this time the MEMORY IDLE line latches the next batch of processor requests and begins the next round of servicing processor requests.



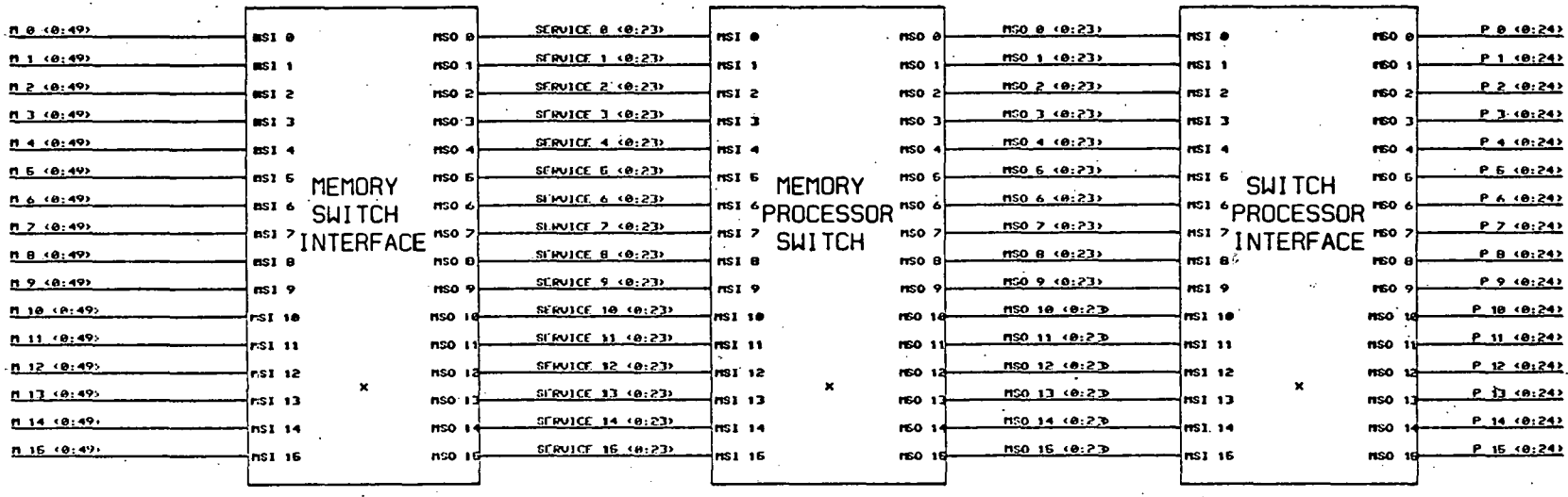
Processor-Memory Cross-Bar Switch (PMCOMN)



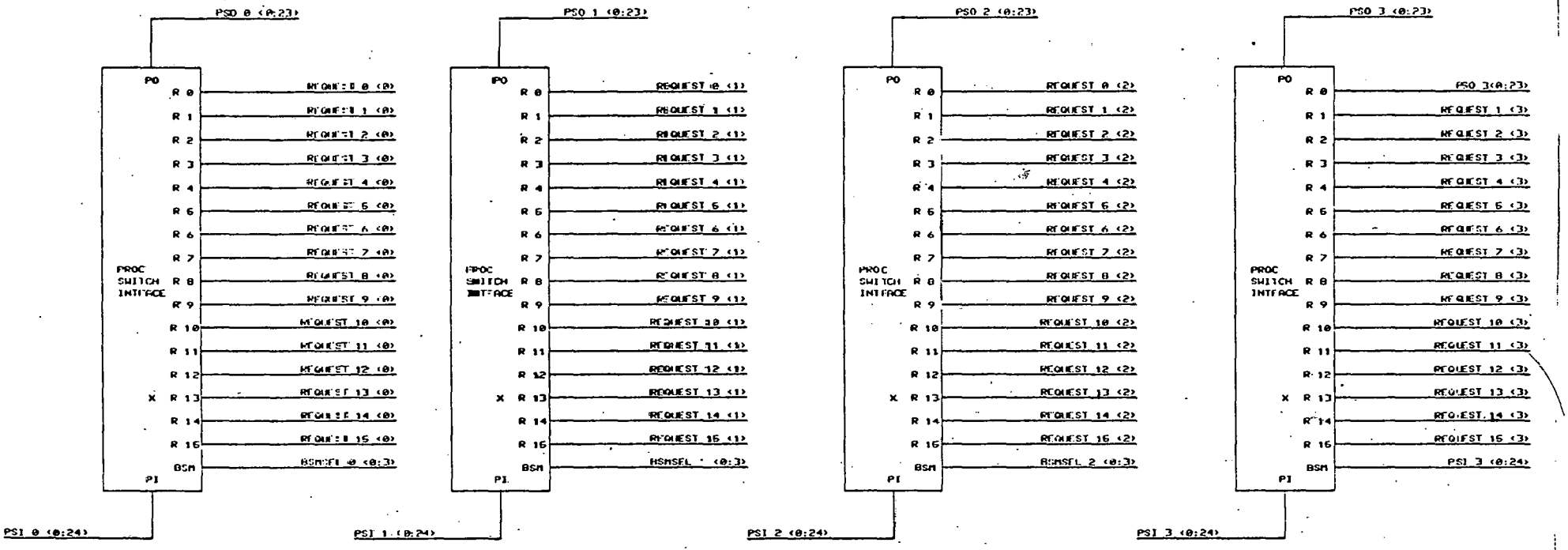
Processor-Memory Communication (PMEMSW)



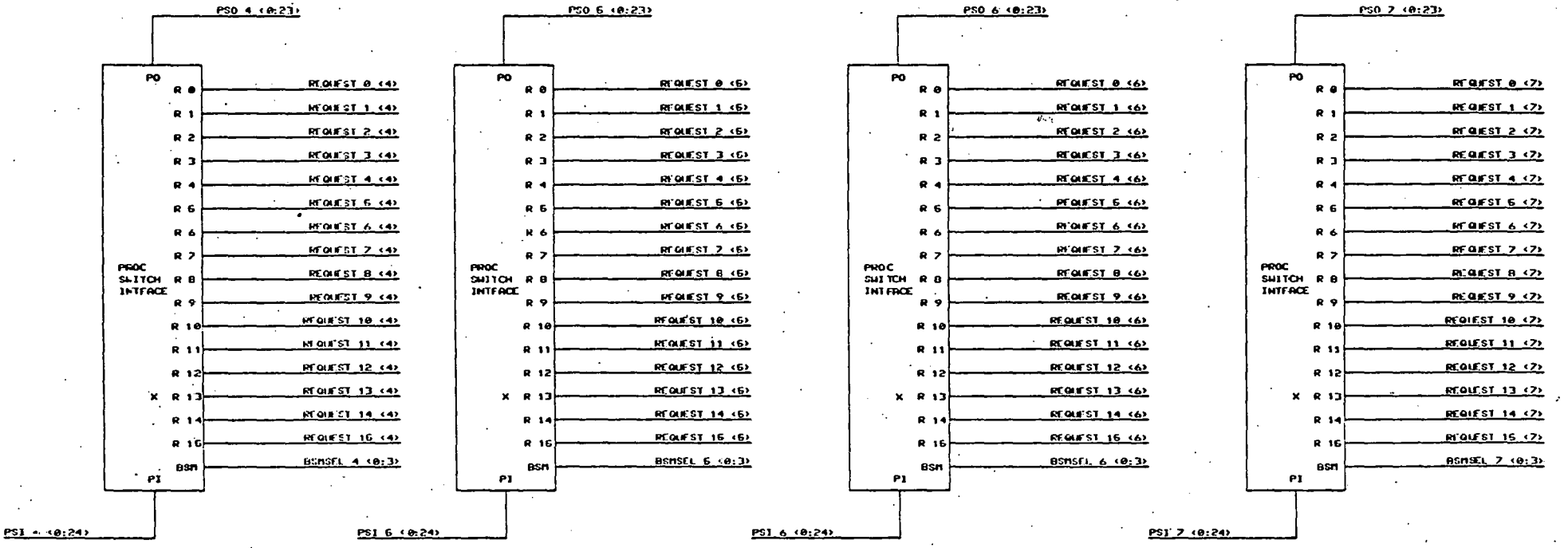
Memory-Processor Cross-Bar Switch (MPCOMN)



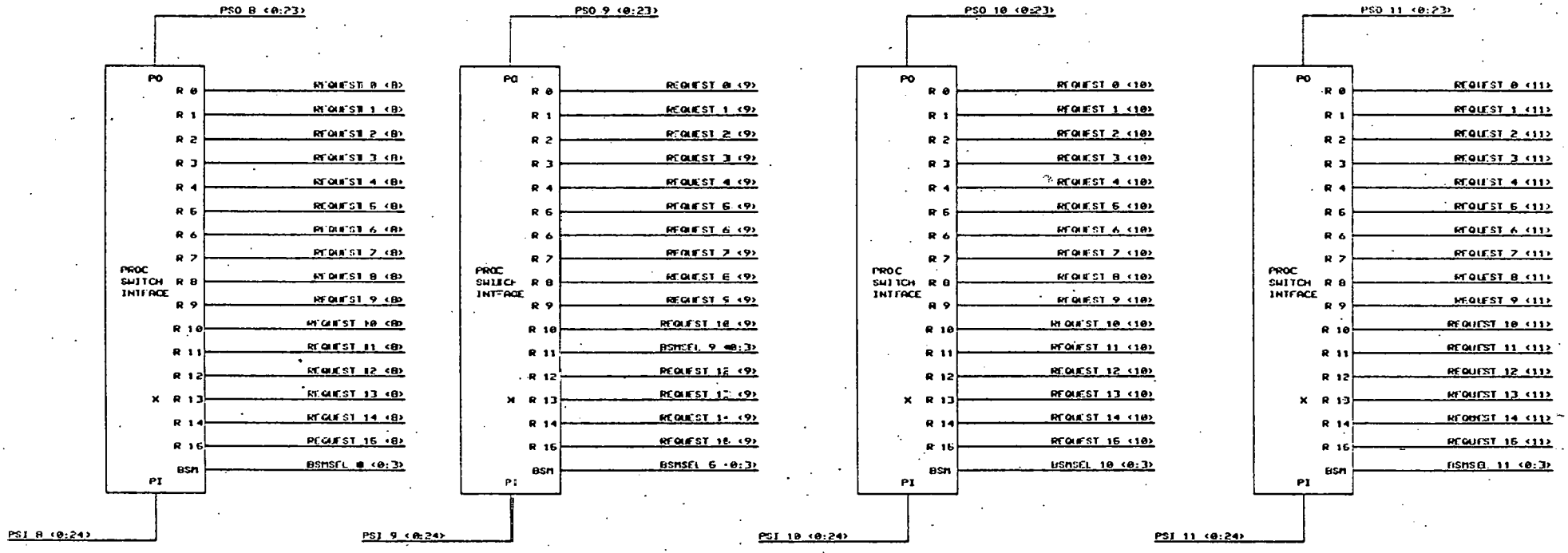
Memory-Processor Communication (MPRCSW)



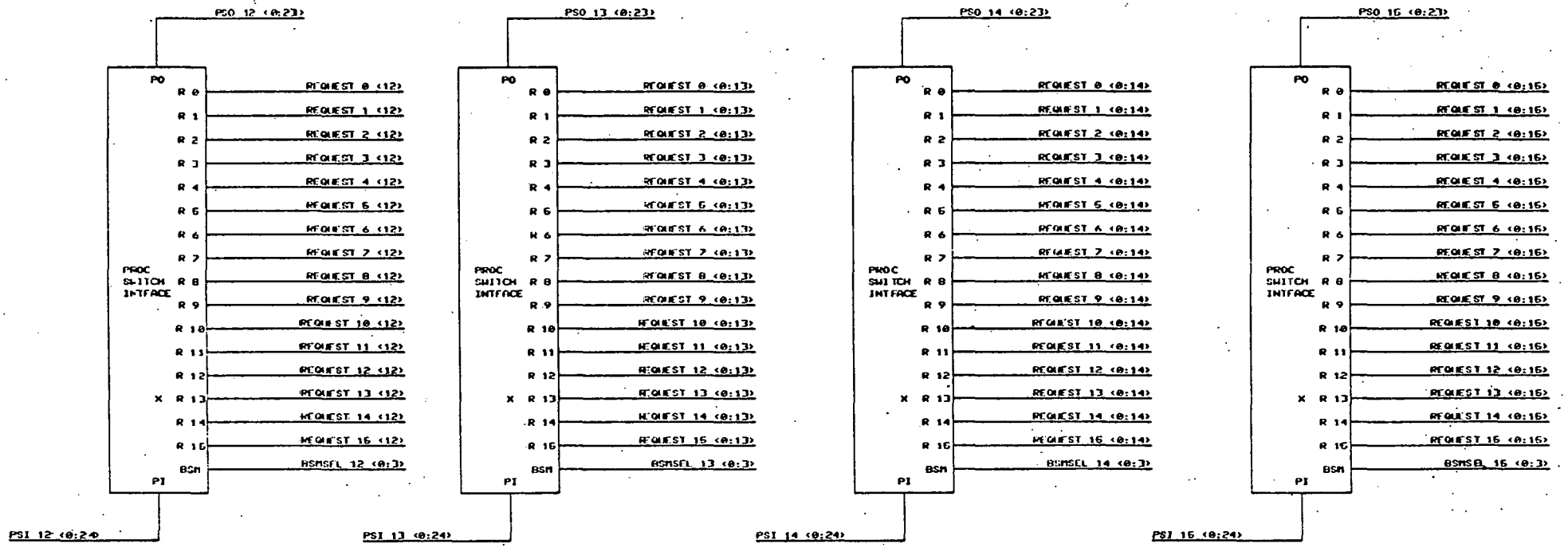
Processor Switch Interface 1/4 (PSINT1)



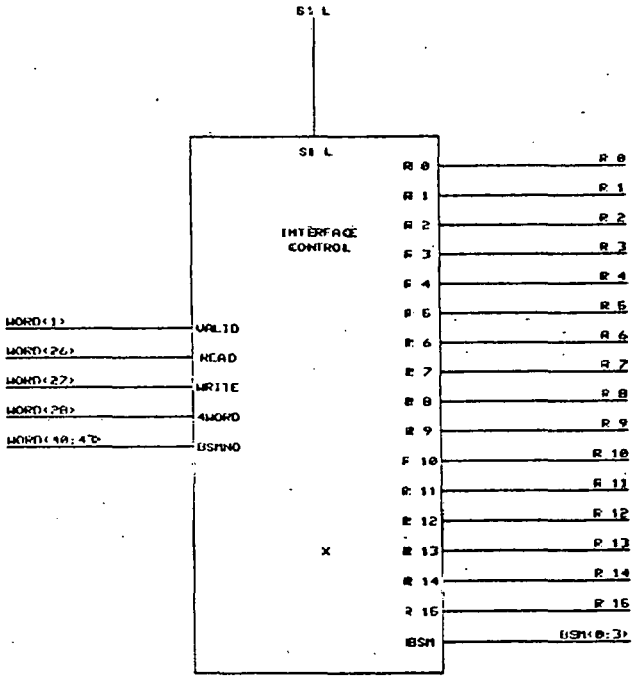
Processor Switch Interface 2/4 (PSINT2)

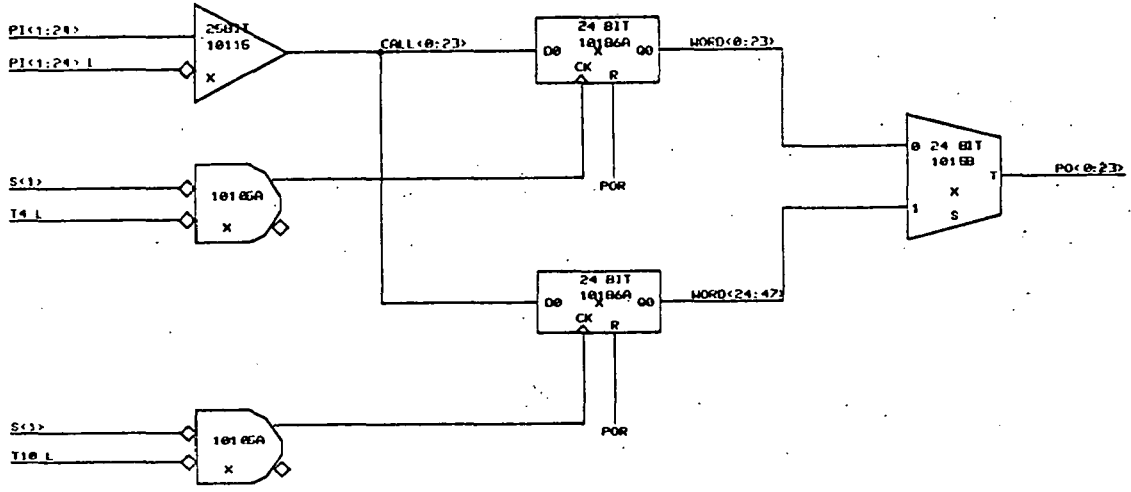


Processor Switch Interface 3/4 (PSINT3)

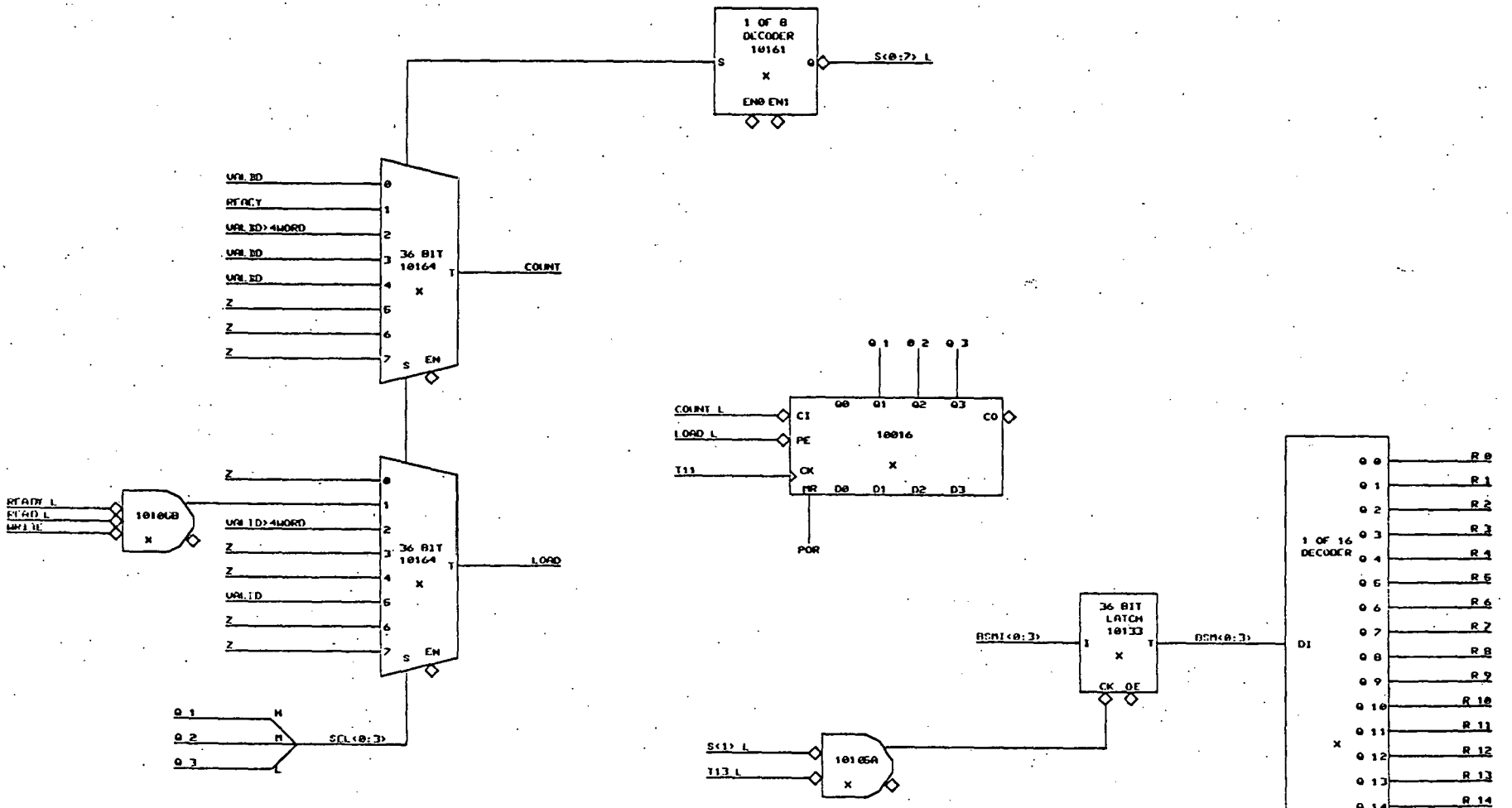


Processor Switch Interface 4/4 (PSINT4)

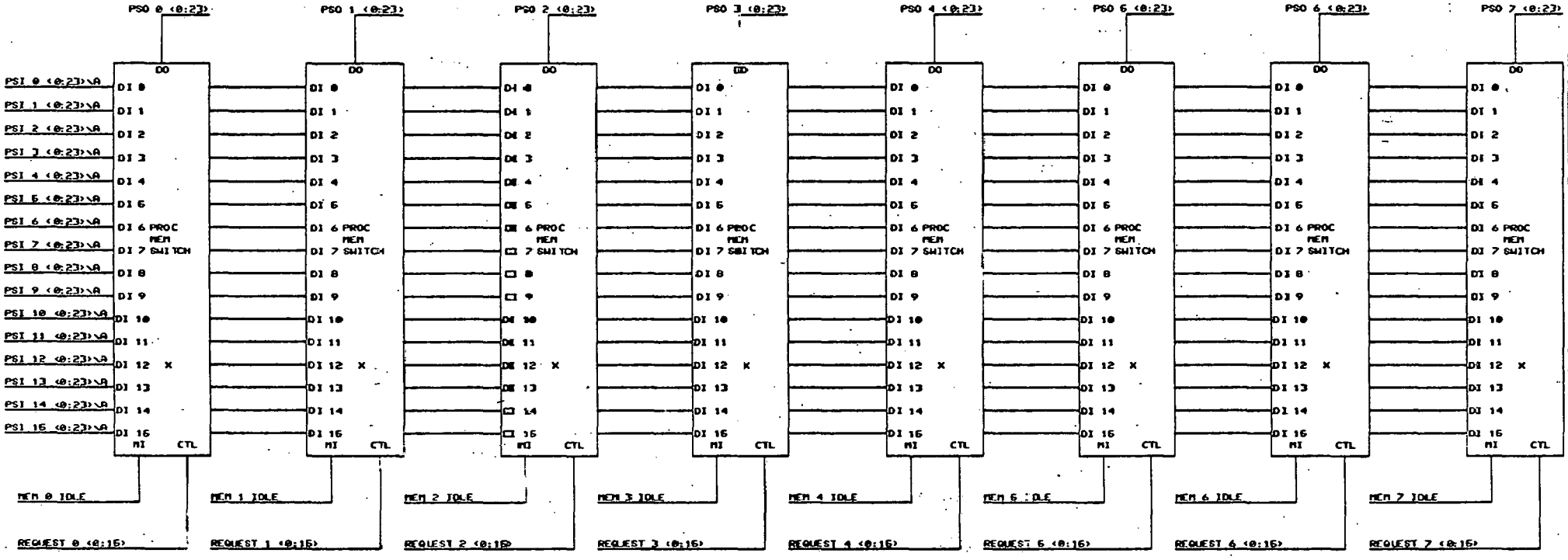




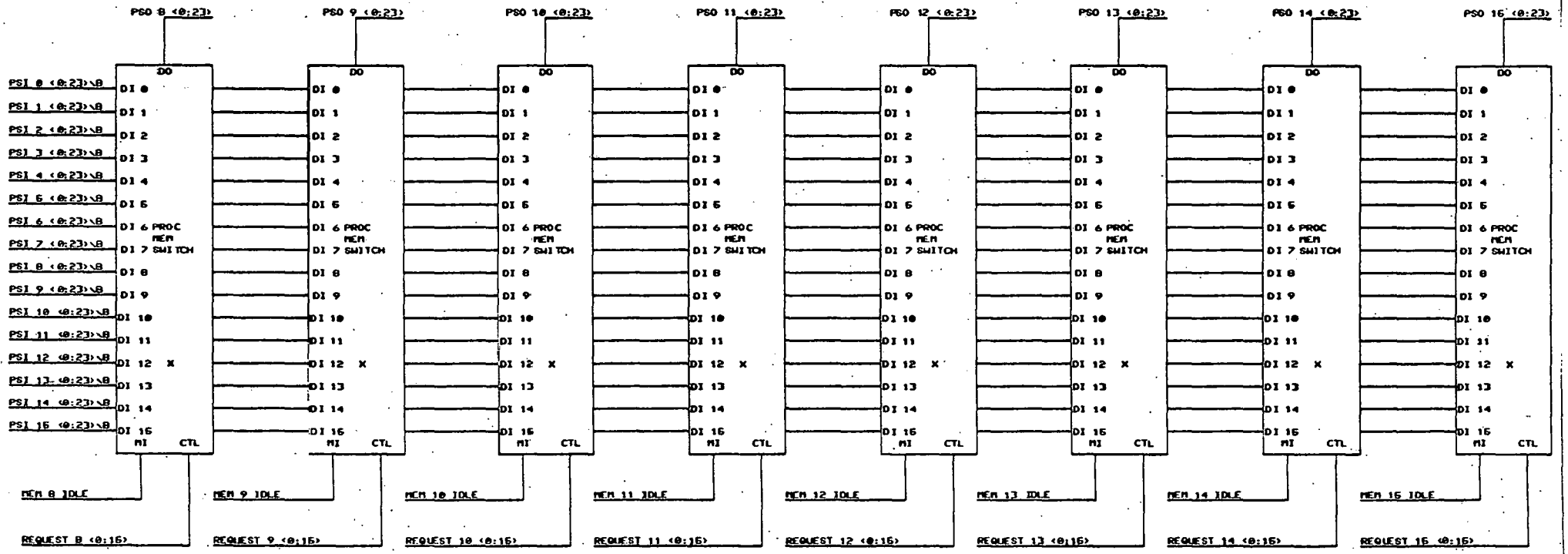
PROC Switch INTFACE (PSWINT)



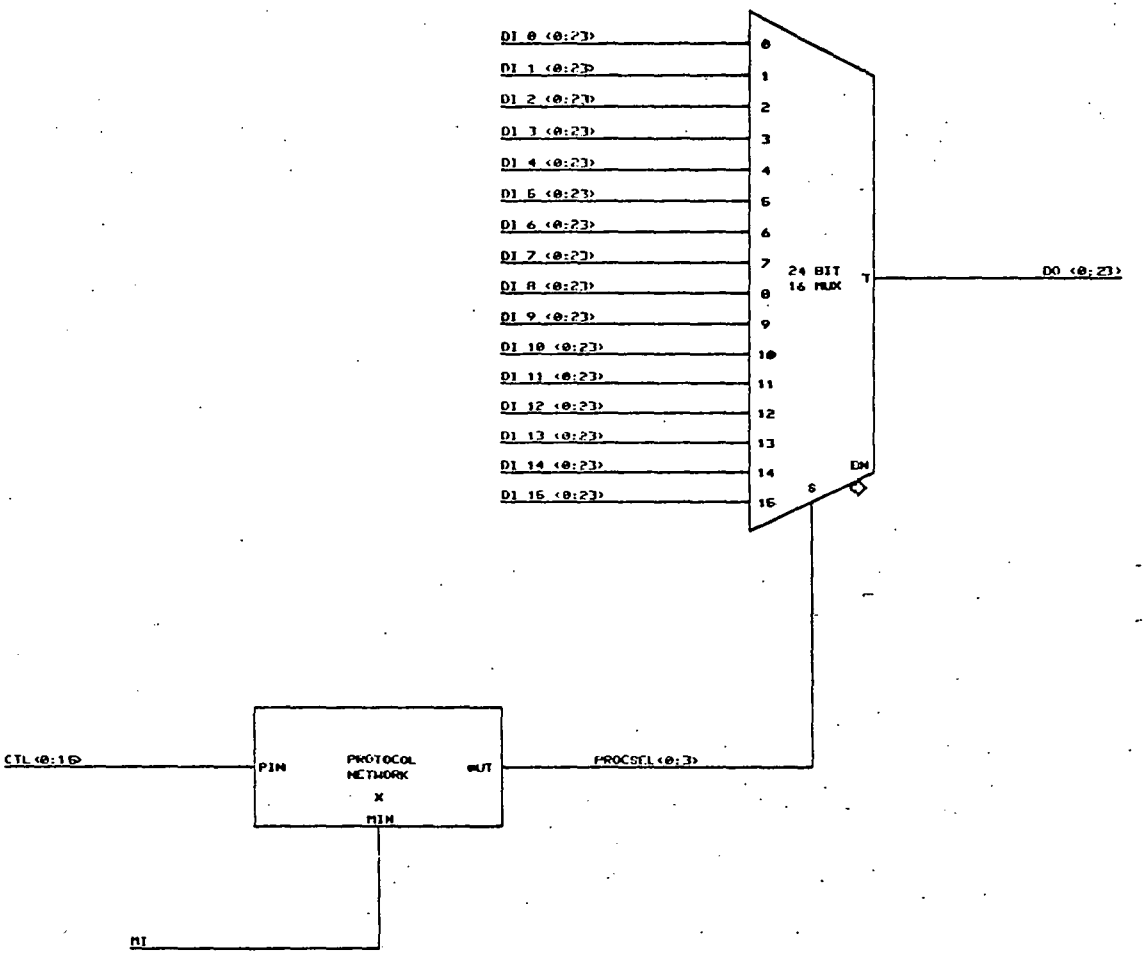
Interface Control (INTCTL)



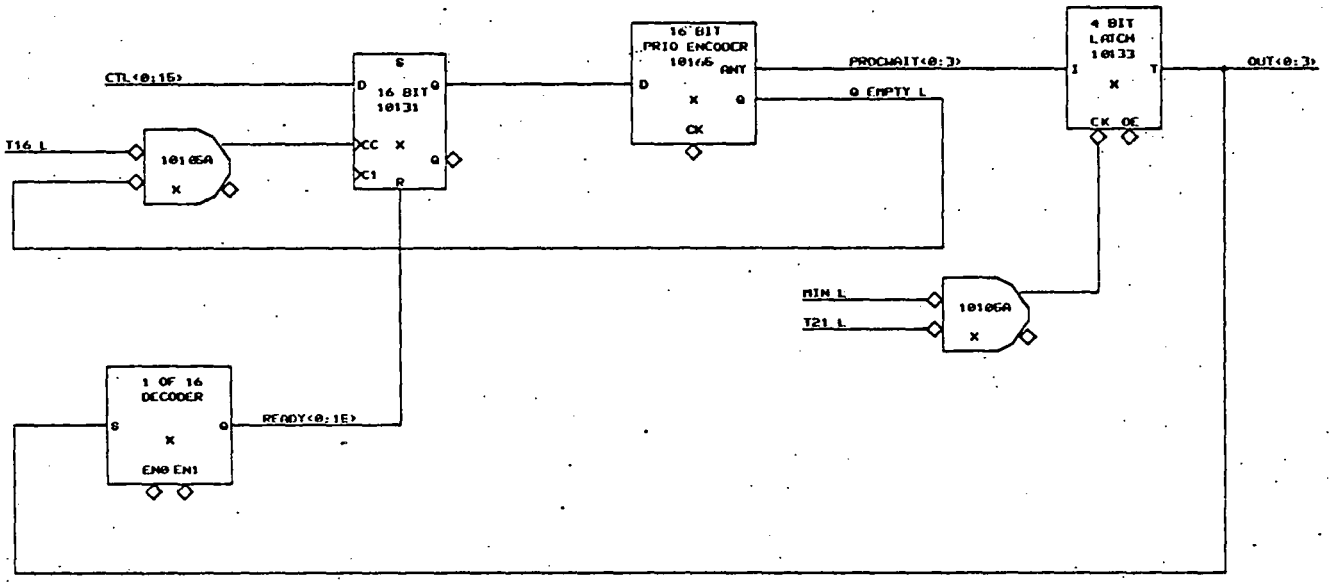
Processor Memory Switch 1/2 (PMSW1)



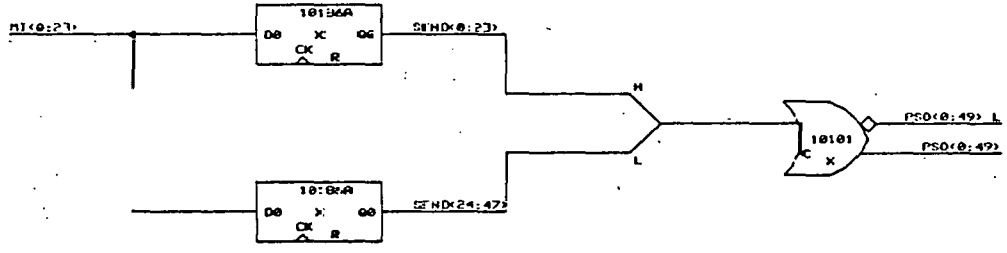
Processor Memory Switch 2/2 (PMSW2)



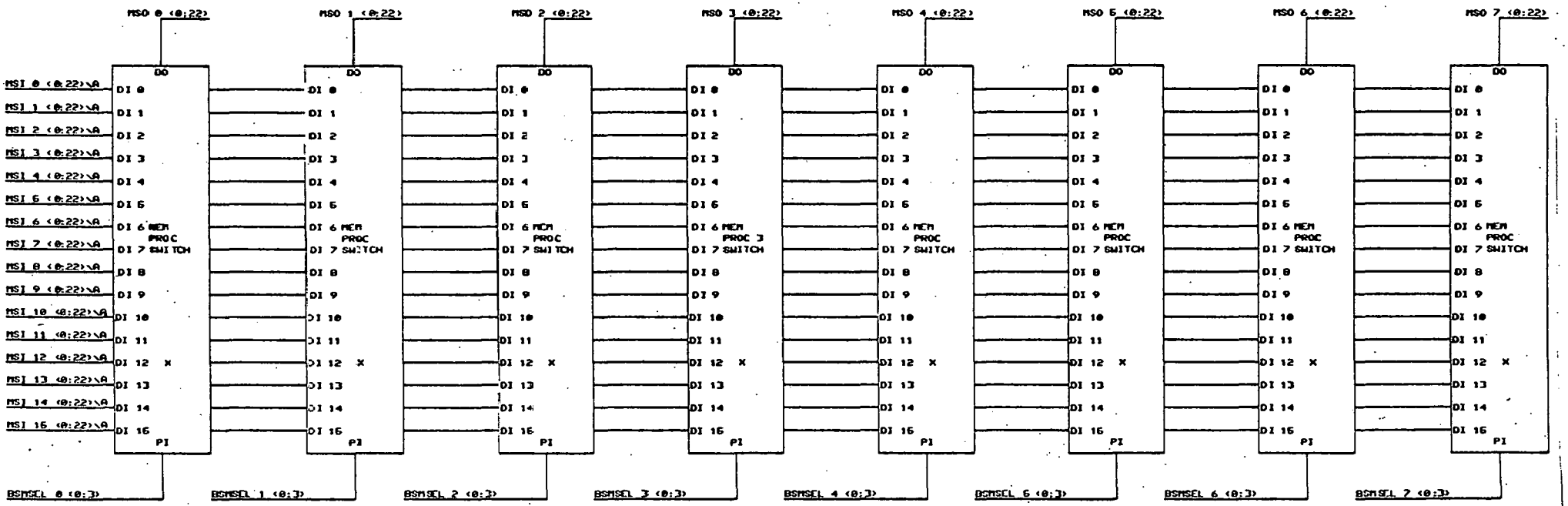
PROC MEM Switch (PMSMUX)



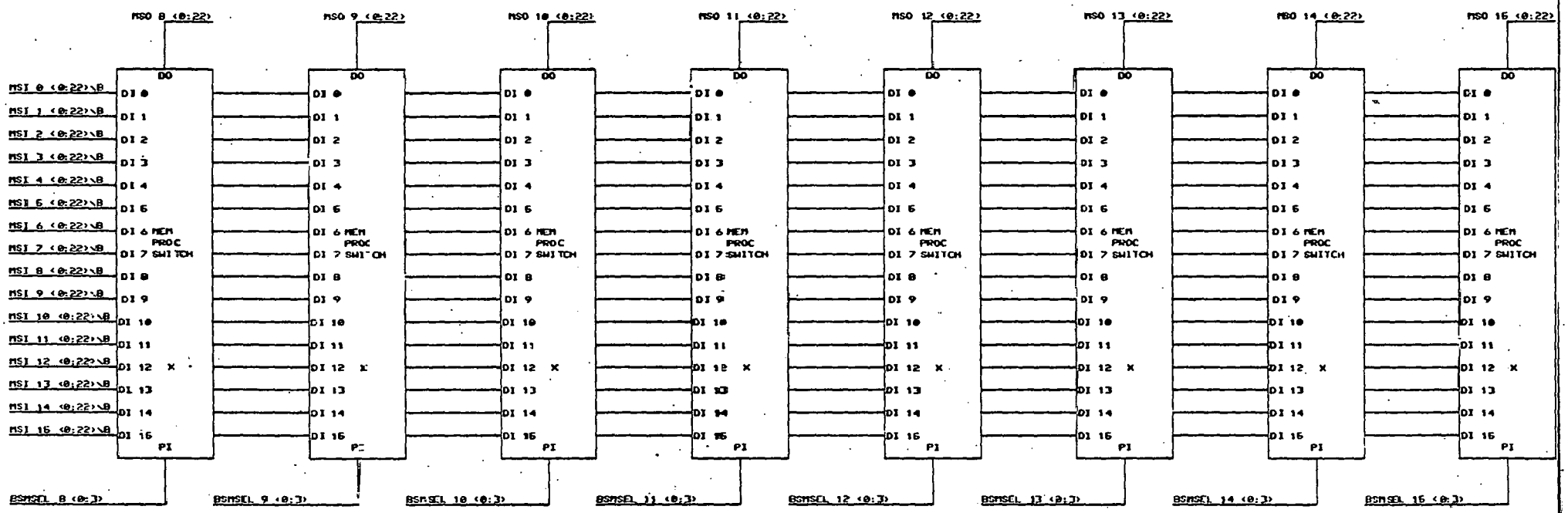
Protocol Network (PROCOL)



Switch MEM INTFACE (SWMINT)



Memory Processor Switch 1/2 (MPSW1)



Memory Processor Switch 2/2 (MPSW2)

5. Summary

The LLL Programmable Digital Filter is a high-performance multiprocessor having general purpose applicability and high programmability; it is extremely cost effective either in a uniprocessor or a multiprocessor configuration.

The important system characteristics of the LLL Filter are as follows:

- Multiple (16) identical processors execute independent instruction streams.
- Every processing element can uniformly address all system memory through a (25-bit serial) crossbar switch.
- Each processing element has dual private caches to reduce contention for main memory, to reduce average memory access time, and to insure that system performance does not seriously degrade as more processing elements (and therefore a bigger and slower interconnection network) are added.
- Each processing element can direct an interrupt to any other processing element.
- Munch registers, hardware queues, and read-modify-write memory capability are available for synchronization.
- The virtual-to-real memory maps include access mode bits which allow efficient sharing of data and instructions.

The architecture and instruction set of the individual processor has been optimized with regard to the multiple processor configuration. The important processor architecture features are as follows:

- A very large (2^{28} word) virtual address space to allow each processor to uniformly address any system memory of feasible size in the forseeable future.
- Efficient mechanisms for allowing the executive to communicate with user processes.
- A high-level instruction set ideally suited for compilers.
- An instruction set specifically tailored to reduce the frequency of pipeline interlocks in a high-performance implementation.
- The capability to perform three-operand instructions through the use of a unique "T-field" descriptor.
- Comprehensive floating-point capability, including three rounding modes and the option to trap on excess pre- or post-normalization.
- The capability to directly perform operations on operands of 4 precisions: quarter-word, half-word, single-word, and double-word.
- Special instructions for dealing with the multiprocessor environment.

6. References

Amdahl, G. M. 1967. "Validity of the single processor approach to achieving large scale computing capabilities," *Proc. AFIPS 1967 SJCC*, 30:483-85.

Ball, J. R. et al. 1962. "On the use of the SOLOMON parallel-processing computer," *Proc. AFIPS 1962 FJCC*, 22:137-46.

Barker, W. B. 1975. "A Multiprocessor Design," Bolt Beranek and Newman, Inc., Report BBN-3126, October 1975, 284 pp.

Carroll, A. B., and Wetherald, R. T. 1967. "Applications of parallel processing to numerical weather prediction," *J. of the ACM*, 14:591-614.

Flynn, M. J. 1966. "Very high-speed computing systems," *Proc. of the IEEE*, 54:1901-9.

Hamer-Hodges, K. J. 1973. "A Fault-Tolerant Multiprocessor Design for Real-Time Control," *Computer Design*, July 1973, 75-81.

Kahan, W. 1973. "Implementation of Algorithms. Part 1.", Technical Report 20, Department of Computer Science, University of California, Berkeley, California, 1973, 339 pp.

Kaplan and Winder, 1973. "Cache-based Computer Systems," *Computer*, March, 1975, 30-36.

Katz, J. H. 1970. "Matrix computations on an associative processor," *Parallel Processor Systems, Technologies, and Applications*, L. C. Hobbs ed., Spartan Books, Washington, D.C., 131-49.

Minsky, M., and Papert, S. 1971. "On some associative, parallel, and analogue computations," *Associative Information Techniques*, Elsevier, New York, New York, 1971.

Sedgewick, R. 1975. "Quicksort," Report No. STAN-CS-75-492, Stanford University Computer Science Department, May 1975, 352 pp.

Steele, G. L. 1975. "Multiprocessing compactifying garbage collection," *Communications of the ACM*, September 1975, Vol. 18, No. 9, 495-508.

A1. Abbreviations

This is a list of the abbreviations used throughout the design.

ABS	ABSOLUTE VALUE
ADD	ADDER
ADR	ADDRESS
BC	BRANCH CONDITION
BOC	BAD ONES COUNT (FLOATING POINT)
BR	BRANCH
BZ	BOTTOM ZEROES
BZC	BOTTOM ZEROES COUNT
C	CACHE
CI	CARRY IN
CK	CLOCK
CLR	CLEAR
CMP	COMPARE
CO	CARRY OUT
COMPL	COMPLEMENT
COND	CONDITION
CS	CHIP SELECT
CSA	CARRY-SAVE ADDER
CTL	CONTROL
CTR	COUNTER
DEC	DECREMENT
DEST	DESTINATION
DI	DATA IN
DIS	DISABLE
DO	DATA OUT
DP	DATA PARALLEL
DW	DOUBLE-WORD
E	E SEQUENCER MICRO INSTRUCTION FIELD
EBOX	EXECUTION BOX
EN	ENABLE
ERR	ERROR
EWAR	EBOX WRITE ADDRESS REGISTERS
EX	EXECUTION
EXP	EXPONENT
F	FUNCTION
FA	FULL ADDER
FS	FROM SWITCH
G	GREATER THAN (ZERO), CARRY GENERATE, GUARD
GE	GREATER THAN OR EQUAL TO (ZERO)
H	HIGH (ONE), HIGH (SIGNIFICANCE) BITS
HW	HALF-WORD
I	INPUT
I	I SEQUENCER MICRO INSTRUCTION FIELD
IBOX	INSTRUCTION BOX
IMMED	IMMEDIATE
INC	INCREMENT
IND	INDIRECT

INSTR	INSTRUCTION
INT	INTERRUPT
IR	INSTRUCTION REGISTER
IRS	SECOND OR THIRD WORD OF INSTRUCTION REGISTER
L	LESS THAN (ZERO), LOW (SIGNIFICANCE) BITS
LE	LESS THAN OR EQUAL TO (ZERO)
LEN	LENGTH
LRU	LEAST RECENTLY USED
LSB	LEAST-SIGNIFICANT BIT
M	MODE, MIDDLE (SIGNIFICANCE) BITS
MANT	MANTISSA
MC	MICRO-CONSTANT
MEM	MEMORY
MM	MULTIPLEXER MERGER
MPC	MICRO PROGRAM COUNTER
MUX	MULTIPLEXER
N	NEGATIVE
NE	NOT EQUAL TO (ZERO)
NUM	NUMBER
NW	NEXT WORD
NZ	NON-ZERO
OE	OUTPUT ENABLE
OP	OPERAND
OVFL	OVERFLOW
P	CARRY PROPAGATE
P	P SEQUENCER MICRO INSTRUCTION FIELD
PA	PHYSICAL ADDRESS
PC	PROGRAM COUNTER
PE	PARALLEL ENABLE
POS	POSITION
PROC	PROCESSOR
PRIO	PRIORITY
R	READ
REG	REGISTER
REL	RELEASE
REM	REMAINDER
REPT	REPITION
RND	ROUND
RNM	RENAMED
S	SELECT, SUM
SCNT	SHIFT COUNT
SEL	SELECT
SIN	SHIFTER INPUT
SKP	SKIP
SO	SHORT OPERAND
SRC	SOURCE
SW	SINGLE-WORD
TRANS	ADDRESS TRANSLATION
T	OUTPUT
TZC	TOP ZEROES COUNT
Q	OUTPUT

QW	QUARTER-WORD
UNDFL	UNDERFLOW
V	OVERFLOW
VA	VIRTUAL ADDRESS
W	WRITE
WE	WRITE ENABLE
X	TRANSMITTED
XBOX	EXPONENT BOX
Z	ZERO

A2. Micro-Code Conventions

$ INDICATES THE BEGINNING OF A FIELD DEFINITION

% INDICATES THE BEGINNING OF A MACRO DEFINITION

[] DELIMIT THE BODY OF A MACRO DEFINITION

, SEPARATES TERMS IN A MICRO-INSTRUCTION

; ENDS A MICRO-INSTRUCTION OR COMMENT

= SEPARATES A FIELD NAME FROM ITS VALUE

! INDICATES THE BEGINNING OF A COMMENT WHICH CONTINUES TO THE LINE FEED

ADD(X,Y) :ALEG("X"),BLEG("Y")
INDICATES THAT PARAMETER X AND Y OF ADD MACRO ARE TO BE SUBSTITUTED AS PARAMETERS OF THE ALEG AND BLEG MACRO RESPECTIVELY.

ALEG(X) :AIN"X"

INDICATES THAT PARAMETER X OF THE ALEG MACRO IS TO BE DIRECTLY SUBSTITUTED AS TEXT AFTER THE STRING "AIN"

* INDICATES DEFAULT VALUE OF FIELD SPECIFICATION.

COMMENT INDICATES THAT ALL TEXT UNTIL A SEMI COLON IS COMMENTS

A3. P-Sequencer Micro-Code Fields

```
$DEST REG CTL<0:1>

*       REG ADR         =1      !DEST REG ADR<0:4> = "REG ADR<0:4>"
        OD1             =2      !       "          = "IR<14:18>"
        ADD             =3      !       "          = "SUM OF ABOVE TWO FIELDS"

$IBOX START ADR<0:11>

*                       =0

$LAST START ADR

*                       =0
                        =1

$OPS READY WHEN IBOX DONE

*                       =0
                        =1

$OUT SEL A                      !SELECTS THE SOURCE FOR THE READ-ONLY DATA
                                !THE "OUT A" LINES.

*       REG             =0      !SOURCE A REGISTER
        CONST           =1      !IMMEDIATE CONSTANT OR CACHE ADR IF C ADR SEL
                                !IS SET

$OUT SEL B                      !SELECTS THE SOURCE FOR THE READ-ONLY DATA
                                !THE "OUT B" LINES.

*       REG             =0      !SOURCE A REGISTER
        CONST           =1      !IMMEDIATE CONSTANT OR CACHE ADR IF C ADR SEL
                                !IS SET

$REG R ADR<0:4>

*                       =0

$REG W ADR<0:4>

*                       =0

$SRC REG CTL A<0:1>

*                       =0      !DON'T SET OPERAND
        REG ADR         =1      !SRC (A OR B) REG ADR="REG R ADR<0:4>"
        OD              =2      !       "             ="OD REG ADR<0:4>"
        ADD             =3      !       "             =SUM OF ABOVE TWO FIELDS
```

```
$SRC REG CTL B<0:1>

*                       =0      !DON'T SET OPERAND
        REG ADR         =1      !SRC (A OR B) REG ADR="REG R ADR<0:4>"
        OD              =2      !       "             ="OD REG ADR<0:4>"
        ADD             =3      !       "             =SUM OF ABOVE TWO FIELDS

$SRC REG OUT SEL

        OD1             =0      !LET THE I SEQ CALCULATE THE OD1 ADDRESS
*       OD2             =1      !       "       "           OD2     "

$SET EWAR

*                       =0
                        =1      !SET THE EBOX WRITE ADDRESS REGISTER TO THE
                                !DESTINATION REGISTER ADDRESS
```

A4. P-Sequencer Micro-Code Macros

```
%A                  [6]
%A+1                [7]
%A+2                [8]
%A+3                [9]
%B                  [8]
%B+1                [9]
%B+2                [10]
%B+3                [11]
%D                  [SET EWAR=1,DEST REG CTL=OD1]
%D RT(AB)           [SET EWAR=1,REG W ADR=AB]
%D+N(N)             [SET EWAR=1,DEST REG CTL=ADD,REG W ADR="N"]
%DH                 [D,DISABLE EWAR CMP=1]
%DH RT(AB)          [D RT(AB),DISABLE EWAR CMP=1]
%DH+N(N)            [D+N(N),DISABLE EWAR CMP=1]
%DONE               [LAST START ADR=1]
%E(ADR)             [IBOX START ADR=ADR]
%H                  [DISABLE EWAR CMP=1]
%N(ODN)             [SRC REG OUT SEL=ODN]
%R D(ADR)           [R(ADR),DONE]
%R(ADR)             [OPS READY WHEN IBOX DONE=1,IBOX START ADR=ADR]
%S1                 [SRC REG CTL A=OD]
%S1 RT(AB)          [SRC REG CTL A=REG ADR,REG R ADR=AB]
%S1+N(N)            [SRC REG CTL A=ADD,REG R ADR=N]
%S2                 [SRC REG CTL B=OD,SRC REG OUT SEL=OD1]
%S2 RT(AB)          [SRC REG CTL B=REG ADR,REG R ADR=AB,SRC REG OUT SEL=OD1]
%S2+N(N)            [SRC REG CTL B=ADD,REG R ADR=N,SRC REG OUT SEL=OD1]
```

A5. P-Sequencer Micro-Code

```
!OD1=REG,OD2=REG
!DEST=QH          SRC 1=QHS          SRC 2=QHS

                          S1                ,R  (SW SRC);   !T=0
          D              ,S1                ,R D(NOP);

                          S1 RT(A)          ,R  (SW SRC);   !T=1
          D              ,S1                ,R D(NOP);

                          S1                ,R  (SW SRC);   !T=2
          D RT(A)        ,S1 RT(A)          ,R D(NOP);

                          S1                ,R  (SW SRC);   !T=3
          D RT(B)        ,S1 RT(B)          ,R D(NOP);
```

```
!OD1=REG,OD2=REG

!DEST=S          SRC 1=QHS        SRC 2=QHS

         D                ,S1              ,R D(SW SRC);   !T=0

         D                ,S1 RT(A)        ,R D(SW SRC);   !T=1

         D RT(A)          ,S1              ,R D(SW SRC);   !T=2

         D RT(B)          ,S1              ,R D(SW SRC);   !T=3

!DEST=S          SRC 1=D          SRC 2=S

                           S1               ,R  (SW SRC);   !T=0
         D                ,S1+N(1)         ,R D(NOP);

                           S1 RT(A)         ,R  (SW SRC);   !T=1
         D                ,S1 RT(A+1)      ,R D(NOP);

                           S1               ,R  (SW SRC);   !T=2
         D RT(A)          ,S1+N(1)         ,R D(NOP);

                           S1               ,R  (SW SRC);   !T=3
         D RT(B)          ,S1+N(1)         ,R D(NOP);

!DEST=S          SRC 1=S          SRC 2=D

                           S1               ,R  (SW SRC);   !T=0
         D                                 ,R D(W2 REG);

                           S1 RT(A)         ,R  (SW SRC);   !T=1
         D                                 ,R D(W2 REG);

                           S1               ,R  (SW SRC);   !T=2
         D RT(A)                           ,R D(W2 REG);

                           S1               ,R  (SW SRC);   !T=3
         D RT(B)                           ,R D(W2 REG);

!DEST=S          SRC 1=D          SRC 2=D

                           S1               ,R  (SW SRC);   !T=0
         D                ,S1+N(1)         ,R D(W2 REG);

                           S1 RT(A)         ,R  (SW SRC);   !T=1
         D                ,S1 RT(A+1)      ,R D(W2 REG);

                           S1               ,R  (SW SRC);   !T=2
         D RT(A)          ,S1+N(1)         ,R D(W2 REG);

                           S1               ,R  (SW SRC);   !T=3
         D RT(B)          ,S1+N(1)         ,R D(W2 REG);
```

```
!OD1=REG,OD2=REG

!DEST=D         SRC 1=S         SRC 2=S

        D+N(1)          ,S1             ,R  (SW SRC);   !T=0
        D                               ,R D(NOP);

        D+N(1)          ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D                               ,R D(NOP);

        D RT(A+1)       ,S1             ,R  (SW SRC);   !T=2
        D RT(A)                         ,R D(NOP);

        D RT(B+1)       ,S1             ,R  (SW SRC);   !T=3
        D RT(B)                         ,R D(NOP);


!DEST=D         SRC 1=D         SRC 2=S

        DH+N(1)         ,S1             ,R  (SW SRC);   !T=0
        D               ,S1+N(1)        ,R D(NOP);

        DH+N(1)         ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D               ,S1 RT(A+1)     ,R D(NOP);

        DH RT(A+1)      ,S1             ,R  (SW SRC);   !T=2
        D RT(A)         ,S1+N(1)        ,R D(NOP);

        DH RT(B+1)      ,S1             ,R  (SW SRC);   !T=3
        D RT(B)         ,S1+N(1)        ,R D(NOP);


!DEST=D         SRC 1=S         SRC 2=D

        DH+N(1)         ,S1             ,R  (SW SRC);   !T=0
        D                               ,R D(W2 REG);

        DH+N(1)         ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D                               ,R D(W2 REG);

        DH RT(A+1)      ,S1             ,R  (SW SRC);   !T=2
        D RT(A)                         ,R D(W2 REG);

        DH RT(B+1)      ,S1             ,R  (SW SRC);   !T=3
        D RT(B)                         ,R D(W2 REG);


!DEST=D         SRC 1=D         SRC 2=D

        DH+N(1)         ,S1             ,R  (SW SRC);   !T=0
        D               ,S1+N(1)        ,R D(W2 REG);

        DH+N(1)         ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D               ,S1 RT(A+1)     ,R D(W2 REG);

        DH RT(A+1)      ,S1             ,R  (SW SRC);   !T=2
        D RT(A)         ,S1+N(1)        ,R D(W2 REG);
```

```
        DH RT(B+1)      ,S1             ,R  (SW SRC);   !T=3
        D RT(B)         ,S1+N(1)        ,R D(W2 REG);

!DEST=D         SRC 1=4W        SRC 2=D

                        S1              ,R  (SW SRC);   !T=0
                        S1+N(1)         ,R  (W2 REG);
        DH+N(1)         ,S1+N(2)        ,R  (NOP);
        D               ,S1+N(3)        ,R D(NOP);

                        S1 RT(A)        ,R  (SW SRC);   !T=1
                        S1 RT(A+1)      ,R  (W2 REG);
        DH+N(1)         ,S1 RT(A+2)     ,R  (NOP);
        D               ,S1 RT(A+3)     ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=2
                        S1+N(1)         ,R  (W2 REG);
        DH RT(A+1)      ,S1+N(2)        ,R  (NOP);
        D RT(A)         ,S1+N(3)        ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=3
                        S1+N(1)         ,R  (W2 REG);
        DH RT(B+1)      ,S1+N(2)        ,R  (NOP);
        D RT(B)         ,S1+N(3)        ,R D(NOP);
```

```
!OD1=REG,OD2=REG

!DEST=4W        SRC 1=D         SRC 2=D

        DH              ,S1             ,R  (SW SRC);   !T=0
        D+N(1)          ,S1+N(1)        ,R  (W2 REG);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

        DH              ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D+N(1)          ,S1 RT(A+1)     ,R  (W2 REG);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

        DH RT(A)        ,S1             ,R  (SW SRC);   !T=2
        D RT(A+1)       ,S1+N(1)        ,R  (W2 REG);
        D RT(A+2)                       ,R  (NOP);
        D RT(A+3)                       ,R D(NOP);

        DH RT(B)        ,S1             ,R  (SW SRC);   !T=3
        D RT(B+1)       ,S1+N(1)        ,R  (W2 REG);
        D RT(B+2)                       ,R  (NOP);
        D RT(B+3)                       ,R D(NOP);


!DEST=4W        SRC 1=4W        SRC 2=D

                        S1              ,R  (SW SRC);   !T=0
                        S1+N(1)         ,R  (W2 REG);
        DH              ,S1+N(2)        ,R  (NOP);
        D+N(1)          ,S1+N(3)        ,R  (NOP);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R  (NOP);

                        S1 RT(A)        ,R  (SW SRC);   !T=1
                        S1 RT(A+1)      ,R  (W2 REG);
        DH              ,S1 RT(A+2)     ,R  (NOP);
        D+N(1)          ,S1 RT(A+3)     ,R  (NOP);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=2
                        S1+N(1)         ,R  (W2 REG);
        DH RT(A)        ,S1+N(2)        ,R  (NOP);
        D RT(A+1)       ,S1+N(3)        ,R  (NOP);
        D RT(A+2)                       ,R  (NOP);
        D RT(A+3)                       ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=3
                        S1+N(1)         ,R  (W2 REG);
        DH RT(B)        ,S1+N(2)        ,R  (NOP);
        D RT(B+1)       ,S1+N(3)        ,R  (NOP);
        D RT(B+2)                       ,R  (NOP);
        D RT(B+3)                       ,R D(NOP);

!DEST=4W        SRC 1=D         SRC 2=4W

                        S1              ,R  (SW SRC);   !T=0
                        S1+N(1)         ,R  (W2 REG);
```

```
DH              ,S2+N(2)        ,R  (NOP);
D+N(1)          ,S2+N(3)        ,R  (NOP);
D+N(2)                          ,R  (NOP);
D+N(3)                          ,R  (NOP);

                S1 RT(A)        ,R  (SW SRC);   !T=1
                S1 RT(A+1)      ,R  (W2 REG);
DH              ,S2+N(2)        ,R  (NOP);
D+N(1)          ,S2+N(3)        ,R  (NOP);
D+N(2)                          ,R  (NOP);
D+N(3)                          ,R D(NOP);

                S1              ,R  (SW SRC);   !T=2
                S1+N(1)         ,R  (W2 REG);
DH RT(A)        ,S2+N(2)        ,R  (NOP);
D RT(A+1)       ,S2+N(3)        ,R  (NOP);
D RT(A+2)                       ,R  (NOP);
D RT(A+3)                       ,R D(NOP);

                S1              ,R  (SW SRC);   !T=3
                S1+N(1)         ,R  (W2 REG);
DH RT(B)        ,S2+N(2)        ,R  (NOP);
D RT(B+1)       ,S2+N(3)        ,R  (NOP);
D RT(B+2)                       ,R  (NOP);
D RT(B+3)                       ,R D(NOP);
```

```
!OD1=REG,OD2=GENERAL

!DEST=QH          SRC 1=QHS        SRC 2=QHS

                        S1                ,R  (SW SRC);   !T=0
          D            ,S1                ,R D(NOP);

                        S1 RT(A)          ,R  (SW SRC);   !T=1
          D            ,S1                ,R D(NOP);

                        S1                ,R  (SW SRC);   !T=2
          D RT(A)      ,S1 RT(A)          ,R D(NOP);

                        S1                ,R  (SW SRC);   !T=3
          D RT(B)      ,S1 RT(B)          ,R D(NOP);
```

```
!OD1=REG,OD2=GENERAL

!DEST=S          SRC 1=QHS        SRC 2=QHS

        D                ,S1              ,R D(SW SRC);   !T=0

        D                ,S1 RT(A)        ,R D(SW SRC);   !T=1

        D RT(A)          ,S1              ,R D(SW SRC);   !T=2

        D RT(B)          ,S1              ,R D(SW SRC);   !T=3

!DEST=S          SRC 1=D          SRC 2=S

                          S1              ,R  (SW SRC);   !T=0
        D                ,S1+N(1)         ,R D(NOP);

                          S1 RT(A)        ,R  (SW SRC);   !T=1
        D                ,S1 RT(A+1)      ,R D(NOP);

                          S1              ,R  (SW SRC);   !T=2
        D RT(A)          ,S1+N(1)         ,R D(NOP);

                          S1              ,R  (SW SRC);   !T=3
        D RT(B)          ,S1+N(1)         ,R D(NOP);

!DEST=S          SRC 1=S          SRC 2=D

                          S1              ,R  (SW SRC);   !T=0
        D                                 ,R D(W2 SRC);

                          S1 RT(A)        ,R  (SW SRC);   !T=1
        D                                 ,R D(W2 SRC);

                          S1              ,R  (SW SRC);   !T=2
        D RT(A)                           ,R D(W2 SRC);

                          S1              ,R  (SW SRC);   !T=3
        D RT(B)                           ,R D(W2 SRC);

!DEST=S          SRC 1=D          SRC 2=D

                          S1              ,R  (SW SRC);   !T=0
        D                ,S1+N(1)         ,R D(W2 SRC);

                          S1 RT(A)        ,R  (SW SRC);   !T=1
        D                ,S1 RT(A+1)      ,R D(W2 SRC);

                          S1              ,R  (SW SRC);   !T=2
        D RT(A)          ,S1+N(1)         ,R D(W2 SRC);

                          S1              ,R  (SW SRC);   !T=3
        D RT(B)          ,S1+N(1)         ,R D(W2 SRC);
```

```
!OD1=REG,OD2=GENERAL

!DEST=D          SRC 1=S          SRC 2=S

        D+N(1)           ,S1              ,R  (SW SRC);   !T=0
        D(1)                              ,D  (NOP);

        D+N(1)           ,S1 RT(A)        ,R  (SW SRC);   !T=1
        D(1)                              ,D  (NOP);

        D RT(A+1)        ,S1              ,R  (SW SRC);   !T=2
        D RT(A)                           ,D  (NOP);

        D RT(B+1)        ,S1              ,R  (SW SRC);   !T=3
        D RT(B)                           ,D  (NOP);


!DEST=D          SRC 1=D          SRC 2=S

        DH+N(1)          ,S1              ,R  (SW SRC);   !T=0
        D                ,S1+N(1)         ,R D(NOP);
        DH+N(1)          ,S1 RT(A)        ,R  (SW SRC);   !T=1
        D                ,S1 RT(A+1)      ,R D(NOP);

        DH RT(A+1)       ,S1              ,R  (SW SRC);   !T=2
        D RT(A)          ,S1+N(1)         ,R D(NOP);

        DH RT(B+1)       ,S1              ,R  (SW SRC);   !T=3
        D RT(B)          ,S1+N(1)         ,R D(NOP);


!DEST=D          SRC 1=S          SRC 2=D

        DH+N(1)          ,S1              ,R  (SW SRC);   !T=0
        D                                 ,R D(W2 SRC);

        DH+N(1)          ,S1 RT(A)        ,R  (SW SRC);   !T=1
        D                                 ,R D(W2 SRC);

        DH RT(A+1)       ,S1              ,R  (SW SRC);   !T=2
        D RT(A)                           ,R D(W2 SRC);

        DH RT(B+1)       ,S1              ,R  (SW SRC);   !T=3
        D RT(B)                           ,R D(W2 SRC);


!DEST=D          SRC 1=D          SRC 2=D

        DH+N(1)          ,S1              ,R  (SW SRC);   !T=0
        D                ,S1+N(1)         ,R D(W2 SRC);

        DH+N(1)          ,S1 RT(A)        ,R  (SW SRC);   !T=1
        D                ,S1 RT(A+1)      ,R D(W2 SRC);

        DH RT(A+1)       ,S1              ,R  (SW SRC);   !T=2
        D RT(A)          ,S1+N(1)         ,R D(W2 SRC);

        DH RT(B+1)       ,S1              ,R  (SW SRC);   !T=3
```

```
        D RT(B)          ,S1+N(1)          ,R D(W2 SRC);

!DEST=D          SRC 1=4W          SRC 2=D

                          S1                ,R  (SW SRC);   !T=0
                          S1+N(1)           ,R  (W2 SRC);
        DH+N(1)           ,S1+N(2)          ,R  (NOP);
        D                 ,S1+N(3)          ,R D(NOP);

                          S1 RT(A)          ,R  (SW SRC);   !T=1
                          S1 RT(A+1)        ,R  (W2 SRC);
        DH+N(1)           ,S1 RT(A+2)       ,R  (NOP);
        D                 ,S1 RT(A+3)       ,R D(NOP);

                          S1                ,R  (SW SRC);   !T=2
                          S1+N(1)           ,R  (W2 SRC);
        DH RT(A+1)        ,S1+N(2)          ,R  (NOP);
        D RT(A)           ,S1+N(3)          ,R D(NOP);

                          S1                ,R  (SW SRC);   !T=3
                          S1+N(1)           ,R  (W2 SRC);
        DH RT(B+1)        ,S1+N(2)          ,R  (NOP);
        D RT(B)           ,S1+N(3)          ,R D(NOP);
```

```
!OD1=REG,OD2=GENERAL

!DEST=4W        SRC 1=D         SRC 2=D

        DH              ,S1             ,R  (SW SRC);   !T=0
        D+N(1)          ,S1+N(1)        ,R  (W2 SRC);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

        DH              ,S1 RT(A)       ,R  (SW SRC);   !T=1
        D+N(1)          ,S1 RT(A+1)     ,R  (W2 SRC);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

        DH RT(A)        ,S1             ,R  (SW SRC);   !T=2
        D RT(A+1)       ,S1+N(1)        ,R  (W2 SRC);
        D RT(A+2)                       ,R  (NOP);
        D RT(A+3)                       ,R D(NOP);

        DH RT(B)        ,S1             ,R  (SW SRC);   !T=3
        D RT(B+1)       ,S1+N(1)        ,R  (W2 SRC);
        D RT(B+2)                       ,R  (NOP);
        D RT(B+3)                       ,R D(NOP);


!DEST=4W        SRC 1=4W        SRC 2=D

                        S1              ,R  (SW SRC);   !T=0
                        S1+N(1)         ,R  (W2 SRC);
        DH              ,S1+N(2)        ,R  (NOP);
        D+N(1)          ,S1+N(3)        ,R  (NOP);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

                        S1 RT(A)        ,R  (SW SRC);   !T=1
                        S1 RT(A+1)      ,R  (W2 SRC);
        DH              ,S1 RT(A+2)     ,R  (NOP);
        D+N(1)          ,S1 RT(A+3)     ,R  (NOP);
        D+N(2)                          ,R  (NOP);
        D+N(3)                          ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=2
                        S1+N(1)         ,R  (W2 SRC);
        DH RT(A)        ,S1+N(2)        ,R  (NOP);
        D RT(A+1)       ,S1+N(3)        ,R  (NOP);
        D RT(A+2)                       ,R  (NOP);
        D RT(A+3)                       ,R D(NOP);

                        S1              ,R  (SW SRC);   !T=3
                        S1+N(1)         ,R  (W2 SRC);
        DH RT(B)        ,S1+N(2)        ,R  (NOP);
        D RT(B+1)       ,S1+N(3)        ,R  (NOP);
        D RT(B+2)                       ,R  (NOP);
        D RT(B+3)                       ,R D(NOP);

!DEST=4W        SRC 1=D         SRC 2=4W

                        S1              ,R  (SW SRC);   !T=0
                        S1+N(1)         ,R  (W2 SRC);
```

```
DH              ,N(OD1)         ,R  (W3 SRC);
D+N(1)          ,N(OD1)         ,R  (W3 SRC);
D+N(2)                          ,R  (NOP);
D+N(3)                          ,R D(NOP);

                S1 RT(A)        ,R  (SW SRC);   !T=1
                S1 RT(A+1)      ,R  (W2 SRC);
DH              ,N(OD1)         ,R  (W3 SRC);
D+N(1)          ,N(OD1)         ,R  (W4 SRC);
D+N(2)                          ,R  (NOP);
D+N(3)                          ,R D(NOP);

                S1              ,R  (SW SRC);   !T=2
                S1+N(1)         ,R  (W2 SRC);
DH RT(A)        ,N(OD1)         ,R  (W3 SRC);
D RT(A+1)       ,N(OD1)         ,R  (W4 SRC);
D RT(A+2)                       ,R  (NOP);
D RT(A+3)                       ,R D(NOP);

                S1              ,R  (SW SRC);   !T=3
                S1+N(1)         ,R  (W2 SRC);
DH RT(B)        ,N(OD1)         ,R  (W3 SRC);
D RT(B+1)       ,N(OD1)         ,R  (W4 SRC);
D RT(B+2)                       ,R  (NOP);
D RT(B+3)                       ,R D(NOP);
```

```
!OD1=GENERAL,OD2=REG

!DEST=QH          SRC 1=QHS          SRC 2=QHS

                                S2              ,R  (SW SRC);    !T=0
                                N(OD1)          ,D  (W1 SRC DEST);

                                S1 RT(A)        ,R  (SW SRC);    !T=1
                                N(OD1)          ,D  (SW DEST);

                                S2              ,R  (SW SRC);    !T=2
        D RT(A)                 ,S1 RT(A)       ,R D(NOP);

                                S2              ,R  (SW SRC);    !T=3
      • D RT(B)                 ,S1 RT(B)       ,R D(NOP);
```

```
!OD1=GENERAL,OD2=REG

!DEST=S          SRC 1=QHS        SRC 2=QHS

                                  S2              ,R D(SW SRC DEST);      !T=0

                                  S1 RT(A),S2     ,R D(SW DEST);   !T=1

          D RT(A)                 ,S2             ,R D(SW SRC);    !T=2

          D RT(B)                 ,S2             ,R D(SW SRC);    !T=3

!DEST=S          SRC 1=D          SRC 2=S

                                  S2              ,R  (SW SRC);    !T=0
                                  N(OD1)          ,E  (W2 SRC);
                                  N(OD1)          ,D  (WF DEST);

                                  S1 RT(A),S2     ,R  (NOP);       !T=1
                                  S1 RT(A+1)      ,R D(SW DEST);

                                  S2              ,R  (SW SRC);    !T=2
          D RT(A)                 ,N(OD1)         ,R D(W2 SRC);

                                  S2              ,R  (SW SRC);    !T=3
          D RT(B)                 ,N(OD1)         ,R D(W2 SRC);

!DEST=S          SRC 1=S          SRC 2=D

                                  S2              ,E  (SW SRC);    !T=0
                                  S2+N(1)         ,R D(W1 DEST);

                                  S1 RT(A),S2     ,R  (NOP);       !T=1
                                  S2+N(1)         ,R D(SW DEST);

                                  S2              ,R  (SW SRC);    !T=2
          D RT(A)                 ,S2+N(1)        ,R D(NOP);

                                  S2              ,R  (SW SRC);    !T=3
          D RT(B)                 ,S2+N(1)        ,R D(NOP);

!DEST=S          SRC 1=D          SRC 2=D

                                  S2              ,R  (SW SRC);    !T=0
                                  S2+N(1)         ,R  (W2 SRC);
                                  N(OD1)          ,D  (WF DEST);

                                  S1 RT(A),S2     ,R  (NOP);       !T=1
                                  S1 RT(A+1),S2+N(1),R D(SW DEST);

                                  S2              ,R  (SW SRC);    !T=2
          D RT(A)                 ,S2+N(1)        ,R D(W2 SRC);

                                  S2              ,R  (SW SRC);    !T=3
          D RT(B)                 ,S2+N(1)        ,R D(W2 SRC);
```

```
!OD1=GENERAL,OD2=REG

!DEST=D          SRC 1=S          SRC 2=S

                                  S2                ,R  (S2 DEST);  !T=0
                                                     D  (WF SRC DEST);

                                  S1 RT(A),S2       ,R  (S2 DEST);  !T=1
                                                     D  (WF DEST);

        D RT(A+1)                 ,S2               ,R  (SW SRC);   !T=2
        D RT(A)                                     ,D  (NOP);

        D RT(B+1)                 ,S2               ,R  (SW SRC);   !T=3
        D RT(B)                                     ,D  (NOP);


!DEST=D          SRC 1=D          SRC 2=S

                                  S2                ,R  (SW SRC);   !T=0
                                  N(OD1)            ,R  (W2 SRC DEST);
                                  N(OD1)            ,D  (WF DEST);

        H                         ,S1 RT(A),S2      ,R  (S2 DEST);  !T=1
                                  S1 RT(A+1),N(OD1),R D(WF DEST);

        DH RT(A+1)                ,S2               ,R  (SW SRC);   !T=2
        D RT(A)                   ,N(OD1)           ,R D(W2 SRC);

        DH RT(B+1)                ,S2               ,R  (SW SRC);   !T=3
        D RT(B)                   ,N(OD1)           ,R D(W2 SRC);


!DEST=D          SRC 1=S          SRC 2=D

                                  S2                ,R  (SW SRC);   !T=0
                                  S2+N(1)           ,R  (W2 SRC DEST);
                                                     D  (WF DEST);

                                  S1 RT(A),S2       ,R  (S2 DEST);  !T=1
                                  S2+N(1)           ,R D(WF DEST);

        DH RT(A+1)                ,S2               ,R  (SW SRC);   !T=2
        D RT(A)                   ,S2+N(1)          ,R D(NOP);

        DH RT(B+1)                ,S2               ,R  (SW SRC);   !T=3
        D RT(B)                   ,S2+N(1)          ,R D(NOP);


!DEST=D          SRC 1=D          SRC 2=D

                                  S2                ,R  (SW SRC);   !T=0
                                  S2+N(1)           ,R  (W2 SRC DEST);
                                  N(OD1)            ,D  (WF DEST);

        H                         ,S1 RT(A),S2      ,R  (S2 DEST);  !T=1
                                  S1 RT(A+1),S2+N(1),R D(WF DEST);
```

```
        DH RT(A+1)      ,S2             ,R  (SW SRC);   !T=2
        D RT(A)         ,S2+N(1)        ,R D(W2 SRC);

        DH RT(B+1)      ,S2             ,R  (SW SRC);   !T=3
        D RT(B)         ,S2+N(1)        ,R D(W2 SRC);
```

!DEST=D SRC 1=4W SRC 2=D

```
                        S2              ,R  (SW SRC);   !T=0
        H               ,S2+N(1)        ,R  (W2 SRC DEST);
                        N(OD1)          ,R  (W3 SRC);
                        N(OD1)          ,R  (W4 SRC);
                        N(OD1)          ,D  (WF3 DEST);

                        S1 RT(A),S2         ,R  (NOP);  !T=1
                        S1 RT(A+1),S2+N(1),R  (NOP);
        H               ,S1 RT(A+2),N(OD1) ,R  (S2 DEST);
                        S1 RT(A+3),N(OD1) ,R D(WF DEST);

                        S2              ,R  (SW SRC);   !T=2
                        S2+N(1)         ,R  (W2 SRC);
        DH RT(A+1)      ,N(OD1)         ,E  (W3 SRC);
        D RT(A)         ,N(OD1)         ,D  (W4 SRC);

                        S2              ,R  (SW SRC);   !T=3
                        S2+N(1)         ,R  (W2 SRC);
        DH RT(A+1)      ,N(OD1)         ,E  (W3 SRC);
        D RT(A)         ,N(OD1)         ,D  (W4 SRC);
```

```
!OD1=GENERAL,OD2=REG

!DEST=4W        SRC 1=D         SRC 2=D

        H               ,S2             ,R  (SW SRC DEST);      !T=0
                         S2+N(1)        ,R  (W2 SRC DEST);
                         N(OD1)         ,E  (W3 DEST);
                         N(OD1)         ,D  (W4 DEST);

        H               ,S1 RT(A),S2    ,R  (SW DEST);  !T=1
                         S1 RT(A+1),S2  ,R  (W2 DEST);
                         N(OD1)         ,E  (W3 DEST);
                         N(OD1)         ,D  (W4 DEST);

        DH RT(A)        ,S2             ,R  (SW SRC);   !T=2
        D RT(A+1)       ,S2+N(1)        ,R  (W2 SRC);
        D RT(A+2)       ,N(OD1)         ,E  (NOP);
        D RT(A+3)       ,N(OD1)         ,D  (NOP);

        DH RT(B)        ,S2             ,R  (SW SRC);   !T=3
        D RT(B+1)       ,S2+N(1)        ,R  (W2 SRC);
        D RT(B+2)       ,N(OD1)         ,E  (NOP);
        D RT(B+3)       ,N(OD1)         ,D  (NOP);

!DEST=4W        SRC 1=4W        SRC 2=D

                        S2              ,R  (SW SRC);   !T=0
                        S2+N(1)         ,R  (W2 SRC);
                        N(OD1)          ,R  (W3 SRC);
                        N(OD1)          ,R  (W4 SRC);
                        N(OD1)          ,E  (WF3 DEST);
                        N(OD1)          ,E  (W2 DEST);
                        N(OD1)          ,E  (W3 DEST);
                        N(OD1)          ,D  (W4 DEST);

                         S1 RT(A),S2        ,R  (NOP);   !T=1
                         S1 RT(A+1),S2+N(1),R  (NOP);
        H               ,S1 RT(A+2),N(OD1) ,R  (SW DEST);
                         S1 RT(A+3),N(OD1) ,R  (W2 DEST);
                         N(OD1)            ,E  (W3 DEST);
                         N(OD1)            ,D  (W4 DEST);

                         S2             ,R  (SW SRC);   !T=2
                         S2+N(1)        ,R  (W2 SRC);
        DH RT(A)        ,N(OD1)         ,R  (W3 SRC);
        D RT(A+1)       ,N(OD2)         ,R  (W4 SRC);
        D RT(A+2)                       ,E  (NOP);
        D RT(A+3)                       ,D  (NOP);

                         S2             ,R  (SW SRC);   !T=3
                         S2+N(1)        ,R  (W2 SRC);
        DH RT(B)        ,N(OD1)         ,R  (W3 SRC);
        D RT(B+1)       ,N(OD2)         ,R  (W4 SRC);
        D RT(B+2)                       ,E  (NOP);
        D RT(B+3)                       ,D  (NOP);

!DEST=4W        SRC 1=D         SRC 2=4W
```

```
                    S2              ,R  (SW SRC);   !T=0
                    S2+N(1)         ,R  (W2 SRC);
H                   ,S2+N(2)        ,R  (WF DEST);
                    S2+N(3)         ,R  (W2 DEST);
                    N(OD1)          ,E  (W3 DEST);
                    N(OD1)          ,D  (W4 DEST);

                    S1 RT(A),S2         ,R  (NOP);  !T=1
                    S1 RT(A+1),S2+N(1),R  (NOP);
H                   ,S2+N(2)        ,R  (SW DEST);
                    S2+N(3)         ,R  (W2 DEST);
                    N(OD1)          ,E  (W3 DEST);
                    N(OD1)          ,D  (W4 DEST);

                    S2              ,R  (SW SRC);   !T=2
                    S2+N(1)         ,R  (W2 SRC);
DH RT(A)            ,S2+N(2)        ,R  (NOP);
D RT(A+1)           ,S2+N(3)        ,R  (NOP);
D RT(A+2)                           ,E  (NOP);
D RT(A+3)                           ,D  (NOP);

                    S2              ,R  (SW SRC);   !T=3
                    S2+N(1)         ,R  (W2 SRC);
DH RT(B)            ,S2+N(2)        ,R  (NOP);
D RT(B+1)           ,S2+N(3)        ,R  (NOP);
D RT(B+2)                           ,E  (NOP);
D RT(B+3)                           ,D  (NOP);
```

```
!OD1=GENERAL,OD2=GENERAL

!DEST=QH        SRC 1=QHS       SRC 2=QHS

                                          E  (SW SRC);    !T=0
                          N(OD1)         ,R  (SW SRC);
                          N(OD1)         ,R D(W1 SRC DEST);

                          S1 RT(A)       ,R  (SW SRC);    !T=1
                          N(OD1)         ,R D(SW SRC DEST);

                                          E  (SW SRC);    !T=2
                          N(OD1)         ,R  (SW SRC);
        D RT(A)          ,S1 RT(A)       ,R D(NOP)

                                          E  (SW SRC);    !T=3
                          N(OD1)         ,R  (SW SRC);
        D RT(B)          ,S1 RT(B)       ,R D(NOP)
```

```
!OD1=GENERAL,OD2=GENERAL

!DEST=S          SRC 1=QHS       SRC 2=QHS

                                          ,E  (SW SRC);    !T=0
                          N(OD1)          ,R D(SW SRC DEST);

                          S1 RT(A)        ,R D(SW SRC);    !T=1
                          N(OD1)          ,D  (SW DEST);

                                           E  (SW SRC);    !T=2
          D RT(A)         ,N(OD1)         ,R D(SW SRC);

                                           E  (SW SRC);    !T=3
          D RT(B)         ,N(OD1)         ,R D(SW SRC);


!DEST=S          SRC 1=D         SRC 2=S

                                           E  (SW SRC);    !T=0
                          N(OD1)          ,R  (SW SRC);
                          N(OD1)          ,R  (W2 SRC);
                          N(OD1)          ,D  (WF DEST);

                          S1 RT(A)        ,R  (SW SRC);    !T=1
                          S1 RT(A+1),N(OD1) ,R D(SW DEST);

                                           E  (SW SRC);    !T=2
                          N(OD1)          ,R  (SW SRC);
          D RT(A)         ,N(OD1)         ,R  (W2 SRC);

                                           E  (SW SRC);    !T=3
                          N(OD1)          ,R  (SW SRC);
          D RT(B)         ,N(OD1)         ,R  (W2 SRC);


!DEST=S          SRC 1=S         SRC 2=D

                                           E  (SW SRC);    !T=0
                          N(OD1)          ,R  (SW SRC);
                                           R  (W2 SRC);
                          N(OD1)          ,D  (W1 DEST);

                          S1 RT(A)        ,R  (SW SRC);    !T=1
                                          ,R  (W2 SRC);
                          N(OD1)          ,D  (SW DEST);

                                           E  (SW SRC);    !T=2
                          N(OD1)          ,R  (SW SRC);
          D RT(A)                         ,R D(W2 SRC);

                                           E  (SW SRC);    !T=3
                          N(OD1)          ,R  (SW SRC);
          D RT(B)                         ,R D(W2 SRC);


!DEST=S          SRC 1=D         SRC 2=D
```

```
                                    E  (SW SRC);    !T=0
                N(OD1)             ,R  (SW SRC);
                                    E  (W2 SRC);
                N(OD1)             ,R  (W2 SRC);
                N(OD1)             ,D  (WF DEST);

                S1 RT(A)           ,R  (SW SRC);    !T=1
                S1 RT(A+1)         ,R  (W2 SRC);
                N(OD1)             ,D  (WF DEST);

                                    E  (SW SRC);    !T=2
                N(OD1)             ,R  (SW SRC);
                                    E  (W2 SRC);
D RT(A)         ,N(OD1)            ,R D(W2 SRC);

                                    E  (SW SRC);    !T=3
                N(OD1)             ,R  (SW SRC);
                                    E  (W2 SRC);
D RT(B)         ,N(OD1)            ,R D(W2 SRC);
```

```
!OD1=GENERAL;OD2=GENERAL

!DEST=D          SRC 1=S          SRC 2=S

                                          E  (SW SRC);    !T=0
                              N(OD1)     ,R  (SW SRC DEST);
                              N(OD1)     ,E D(W2 DEST);

                              S1 RT(A)   ,R  (SW SRC);    !T=1
                              N(OD1)     ,E  (SW DEST);
                                          D  (W2 DEST);

                                          E  (SW SRC);    !T=2
        D RT(A)               ,N(OD1)    ,R  (SW SRC);
        D RT(A+1)                        ,D  (NOP);

                                          E  (SW SRC);    !T=3
        D RT(B)               ,N(OD1)    ,R  (SW SRC);
        D RT(B+1)                        ,D  (NOP);

!DEST=D          SRC 1=D          SRC 2=S

                                         ,E  (SW SRC);    !T=0
                              N(OD1)     ,R  (SW SRC);
                              N(OD1)     ,R  (W2 SRC);
                              N(OD1)     ,E  (W1 DEST);
                              N(OD1)     ,D  (WF DEST);

                              S1 RT(A)   ,R  (SW SRC);    !T=1
                              S1 RT(A+1),N(OD1),R  (S2 DEST);
                              N(OD1)     ,D  (WF DEST);

                                          E  (SW SRC);    !T=2
        DH RT(A+1)            ,N(OD1)    ,R  (SW SRC);
        D RT(A)               ,N(OD1)    ,R  (W2 SRC);

                                          E  (SW SRC);    !T=3
        DH RT(B+1)            ,N(OD1)    ,R  (SW SRC);
        D RT(B)               ,N(OD1)    ,R  (W2 SRC);

!DEST=D          SRC 1=S          SRC 2=D

                                          E  (SW SRC);    !T=0
                              N(OD1)     ,R  (SW SRC);
                                          R  (W2 SRC);
                              N(OD1)     ,E  (W2 DEST);
                              N(OD1)     ,D  (WF DEST);

                              S1 RT(A)   ,R  (SW SRC);    !T=1
                                          R  (W2 SRC);
                              N(OD1)     ,E  (S2 DEST);
                              N(OD1)     ,D  (WF DEST);

                                          E  (SW SRC);    !T=2
        DH RT(A+1)            ,N(OD1)    ,R  (SW SRC);
        D RT(A)                          ,R D(W2 SRC);
```

```
                                        E  (SW SRC);   !T=3
     DH RT(B+1)      ,N(OD1)           ,R  (SW SRC);
     D RT(B)                           ,R D(W2 SRC);

!DEST=D          SRC 1=D          SRC 2=D

                                        E  (SW SRC);   !T=0
                     N(OD1)             ,R  (SW SRC);
                                        E  (W2 SRC);
                     N(OD1)             ,R  (W2 SRC DEST);
                     N(OD1)             ,D  (WF DEST);

                     S1 RT(A)           ,R  (SW SRC);   !T=1
                     S1 RT(A+1)         ,R  (W2 SRC);
                     N(OD1)             ,E  (S2 DEST);
                     N(OD1)             ,D  (WF DEST);

                                        E  (SW SRC);   !T=2
                     N(OD1)             ,R  (SW SRC);
     DH RT(A+1)                         ,E  (W2 SRC);
     D RT(A)         ,N(OD1)            ,R D(W2 SRC);

                                        E  (SW SRC);   !T=3
                     N(OD1)             ,R  (SW SRC);
     DH RT(B+1)                         ,E  (W2 SRC);
     D RT(B)         ,N(OD1)            ,R D(W2 SRC);

!DEST=D          SRC 1=4W         SRC 2=D

                                        E  (SW SRC);   !T=0
                     N(OD1)             ,R  (SW SRC);
                                        E  (W2 SRC);
                     N(OD1)             ,R  (W2 SRC);
                     N(OD1)             ,R  (W3 SRC);
                     N(OD1)             ,R  (W4 SRC);
                     N(OD1)             ,E  (WF2 DEST);
                     N(OD1)             ,D  (WF DEST);

                     S1 RT(A)           ,R  (SW SRC);   !T=1
                     S1 RT(A+1)         ,R  (W2 SRC);
                     S1 RT(A+2),N(OD1)  ,R  (S2 DEST);
                     S1 RT(A+3),N(OD1)  ,R D(WF DEST);

                                        E  (SW SRC);   !T=2
                     N(OD1)             ,R  (SW SRC);
                                        E  (W2 SRC);
                     N(OD1)             ,R  (W2 SRC);
     DH RT(A+1)      ,N(OD1)           ,R  (W3 SRC);
     D RT(A)         ,N(OD1)           ,R  (W4 SRC);

                                        E  (SW SRC);   !T=3
                     N(OD1)             ,R  (SW SRC);
                                        E  (W2 SRC);
                     N(OD1)             ,R  (W2 SRC);
     DH RT(B+1)      ,N(OD1)           ,R  (W3 SRC);
     D RT(B)         ,N(OD1)           ,R  (W4 SRC);
```

```
!OD1=GENERAL,OD2=GENERAL

!DEST=4W        SRC 1=D         SRC 2=D

                                          E  (SW SRC);    !T=0
                          N(OD1)         ,R  (SW SRC);
                                          E  (W2 SRC);
                          N(OD1)         ,R  (W2 SRC);
                          N(OD1)         ,E  (WF DEST);
                          N(OD1)         ,E  (W2 DEST);
                          N(OD1)         ,E  (W3 DEST);
                          N(OD1)         ,D  (W4 DEST);

                          S1 RT(A)       ,R  (SW SRC);    !T=1
                          S1 RT(A+1)     ,R  (W2 SRC);
                          N(OD1)         ,E  (SW DEST);
                          N(OD1)         ,E  (W2 DEST);
                          N(OD1)         ,E  (W3 DEST);
                          N(OD1)         ,D  (W4 DEST);

                                          E  (SW SRC);    !T=2
                          N(OD1)         ,R  (SW SRC);
        DH RT(A)                         ,E  (W2 SRC);
        D RT(A+1)        ,N(OD1)         ,R  (W2 SRC);
        D RT(A+2)                        ,E  (NOP);
        D RT(A+3)                        ,D  (NOP);

                                          E  (SW SRC);    !T=3
                          N(OD1)         ,R  (SW SRC);
        DH RT(B)                         ,E  (W2 SRC);
        D RT(B+1)        ,N(OD1)         ,R  (W2 SRC);
        D RT(B+2)                        ,E  (NOP);
        D RT(B+3)                        ,D  (NOP);

!DEST=4W        SRC 1=4W        SRC 2=D

                                          E  (SW SRC);    !T=0
                          N(OD1)         ,R  (SW SRC);
                                          E  (W2 SRC);
                          N(OD1)         ,R  (W2 SRC);
                          N(OD1)         ,R  (W3 SRC);
                          N(OD1)         ,R  (W4 SRC);
                          N(OD1)         ,E  (WF4 DEST);
                          N(OD1)         ,E  (W2 DEST);
                          N(OD1)         ,E  (W3 DEST);
                          N(OD1)         ,D  (W4 DEST);

                          S1 RT(A)       ,R  (SW SRC);    !T=1
                          S1 RT(A+1)     ,R  (W2 SRC);
        H                ,S1 RT(A+2),N(OD1),R (SW DEST);
                          S1 RT(A+3),N(OD1),R (W2 DEST);
                          N(OD1)         ,E  (W3 DEST);
                          N(OD1)         ,D  (W4 DEST);

                                          E  (SW SRC);    !T=2
                          N(OD1)         ,R  (SW SRC);
                                          E  (W2 SRC);
                          N(OD1)         ,R  (W2 SRC);
        DH RT(A)         ,N(OD1)         ,R  (W3 SRC);
```

```
        D RT(A+1)       ,N(OD1)         ,R  (W4 SRC);
        D RT(A+2)                       ,E  (NOP);
        D RT(A+3)                       ,D  (NOP);
                                         E  (SW SRC);   !T=3
                        N(OD1)          ,R  (SW SRC);
                                         E  (W2 SRC);
                        N(OD1)          ,R  (W2 SRC);
        DH RT(B)        ,N(OD1)         ,R  (W3 SRC);
        D RT(B+1)       ,N(OD1)         ,R  (W4 SRC);
        D RT(B+2)                       ,E  (NOP);
        D RT(B+3)                       ,D  (NOP);

!DEST=4W        SRC 1=D         SRC 2=4W

                                         E  (SW SRC);   !T=0
                        N(OD1)          ,R  (SW SRC);
                                         E  (W2 SRC);
                        N(OD1)          ,R  (W2 SRC);
                                        ,R  (W3 SRC);
                                        ,R  (W4 SRC);
                        N(OD1)          ,E  (WF DEST);
                        N(OD1)          ,E  (W2 DEST);
                        N(OD1)          ,E  (W3 DEST);
                        N(OD1)          ,D  (W4 DEST);

                        S1 RT(A)        ,R  (SW SRC);   !T=1
                        S1 RT(A+1)      ,R  (W2 SRC);
                                        ,R  (W3 SRC);
                                        ,R  (W4 SRC);
                        N(OD1)          ,E  (SW DEST);
                        N(OD1)          ,E  (W2 DEST);
                        N(OD1)          ,E  (W3 DEST);
                        N(OD1)          ,D  (W4 DEST);

                                         E  (SW SRC);   !T=2
                        N(OD1)          ,R  (SW SRC);
                                         E  (W2 SRC);
                        N(OD1)          ,R  (W2 SRC);
        DH RT(A)                        ,R  (W3 SRC);
        D RT(A+1)                       ,R  (W4 SRC);
        D RT(A+2)                       ,E  (NOP);
        D RT(A+3)                       ,D  (NOP);
                                         E  (SW SRC);   !T=3
                        N(OD1)          ,R  (SW SRC);
                                         E  (W2 SRC);
                        N(OD1)          ,R  (W2 SRC);
        DH RT(B)                        ,R  (W3 SRC);
        D RT(B+1)                       ,R  (W4 SRC);
        D RT(B+2)                       ,E  (NOP);
        D RT(B+3)                       ,D  (NOP);
```

A6. I-Sequencer Micro-Code Fields

```
$ADD F<0:5>                     !ADDRESS ARITHMETIC ADDER FUNCTION LINES
                                !(F<0> IS THE MODE CTR, F<1:4> IS THE FUNCTION
                                ! AND F<5> IS THE CARRY IN)

*       A+0             =0
        A+1             =1
        A+B             =12
        A+B+1           =13
        A-B-1           =18
        A-B             =19
        A*2             =24
        A*2+1           =25
        A-1             =30

        NA              =32
        NA AND B        =34
        NA AND NB       =36
        Z               =38
        NA OR B         =40
        B               =42
        A XNOR B        =44
        A AND B         =46
        NA OR NB        =48
        A XOR B         =50
        NB              =52
        A AND NB        =54
        MINUS ONE       =56
        A OR B          =58
        A OR NB         =60
        A               =62

$ADD LEG A<0:1>                 !CONTROLS LEG A ON ADDRESS ARITHMETIC ADDER

*                       =0      !PRE-FETCH PC.  USED BY INSTR QUEUE LOGIC
        INDEX REG       =1      !INDEX REG FILE.
        LSI 11          =2      !DATA FROM LSI-11
                        =3      !WRITE DATA BUS.  ONLY USED BY HARDWARE

$ADD LEG B<0:2>                 !CONTROLS LEG B ON ADDRESS ARITHMETIC ADDER

*                       =0      !BRANCH OFFSET FOR SHORT PC RELATIVE BRANCHES
        SO              =1      !SHORT OPERAND OFFSET
        VAR BASE        =2      !VARIABLE BASE OFFSET
        FIX BASE        =3      !FIXED BASE OFFSET
        C BLOCK ADR     =4      !CACHE MISS BLOCK ADDRESS
        T               =7      !T REGISTER
```

```
$ADD LOAD IND REG

*                       =0
                        =1      !LOAD THE INDIRECT BIT AND INDEX REGISTER FIELD
                                !FROM THE INDIRECT ADDRESS POINTER COMMING OUT
                                !OF THE INDEX REGISTER FILE INTO A SPECIAL
                                !REGISTER FOR THEM.

$ADD RIGHT SHIFT 8 BITS

*                       =0
                        =1      !RIGHT SHIFT THE OUTPUT OF THE ADDRESS
                                !ARITHMETIC ADDER BY 8 BITS.

$C ADR SEL

*                       =0      !ALLOW LONG IMMEDIATE CONSTANTS TO BE USED
                        =1      !FEED THE CACHE ADDRESS INTO THE LONG IMMEDIATE
                                !CONSTANT FIELD OF THE "OUT A" AND "OUT B"
                                !MULTIPLEXERS

$C CLEAR HOLD MISS

*                       =0
                        =1      !CLEAR THE HOLD CACHE MISS REGISTER

$C FETCH

*                       =0
                        =1      !THE CURRENT MEMORY READ IS FETCHING AN INSTR

$C OPERATION

*                       =0
                        =1      !THE CURRENT MICROINSTRUCION IS USING THE CACHE

$C W CHECK

*                       =0
                        =1      !CHECK THE CACHE TO SEE IF A WORD IS THERE SO
                                !THAT IT MAY BE WRITTEN IN THE FUTURE

$C W SET NUM<0:1>               !SPECIFIES A SPECIFIC CACHE SET TO BE WRITTEN
                                !INTO.  THIS IS ONLY USED BY DIAGNOSTIC PROGRAMS
*                       =0
                        =1
                        =2
                        =3
```

```
$C W SET SRC SEL

*                       =0      !WRITE INTO THE CACHE SET GIVEN BY
                                !"C W SET NUM<0:1>"
        LRU             =1      !WRITE INTO THE LRU CACHE SET

$DISABLE EWAR CMP

*                       =0
                        =1      !DISABLE THE EBOX WRITE ADDRESS REGISTER 0
                                !UNTIL IT IS SET AGAIN.

$EBOX W ADR IS A REG

*                       =0
                        =1      !THE EBOX WRITE ADDRESS TO BE PUT IN THE EWARS
                                !IS A REGISTER ADDRESS

$IBOX C W

*                       =0
                        =1      !CAUSES THE IBOX TO WRITE THE "W DATA"
                                !LINES INTO THE CACHE.

$IBOX REG W

*                       =0
                        =1      !CAUSES THE IBOX TO WRITE THE "W DATA"
                                !LINES INTO THE REGISTER FILE(S)

$IMMED CONST LONG

*                       =0
                        =1      !LONG IMMEDIATE CONSTANT

$INDEX REG ADR SEL<0:2>         !SELECTS THE SOURCE FOR THE ADDRESS OF THE
                                !INDEX REGISTER FILE

*       USER            =0      !"REG SET<0:1>:REG R ADR A<0:4>"
                                !USER'S REGISTER SPECIFIED BY THE MICROCODE

        OD              =1      !"REG SET<0:1>:OD REG ADR<0:4>"
        VB REG          =3      !"REG SET<0:1>:IRS<6:10>" VARIABLE BASE IND R
        IND             =4      !"REG SET<0:1>:IND INDEX REG ADR<0:4>"

        T               =6      !"Z*2:REG R ADR A<0:4>"
                                !IBOX TEMPORARY REGISTERS
```

```
$INSTR OUT A

*                       =0
                        =1      !SAYS THAT AN INSTRUCTION IS BEING READ
                                !OUT OF "OUT A", AND TO PUT IT IN THE
                                !INSTRUCTION QUEUE.

$LOAD AT LRU DECODE RAM

*                       =0
                        =1

$LOAD C LRU DECODE RAM

*                       =0
                        =1

$MEM R

*                       =0
                        =1

$MEM START R

*                       =0
                        =1      !START A MEMORY READ THROUGH THE SWITCH

$MEM START W

*                       =0
                        =1

$MIC BR ADR<0:11>               !MICRO BRANCH ADDRESS

*                       =0

$MIC BR SEL<0:2>

        START           =0      !"P IBOX START ADR<0:11>"
*       MIC BR          =1      !"MIC BR ADR<0:11>"

$MIC COND SEL<0:2>

*                       =0
                        =1
```

```
$MIC EN INT

*                       =0
                        =1      !ENABLE MICRO INTERRUPTS

$MIC JSR

*                       =0
                        =1      !DO A JUMP TO SUBROUTINE

$OUT SEL<0:1>                   !SELECTS THE SOURCE FOR THE READ ONLY DATA
                                !THE "OUT A" OR "OUT B" LINES.

        C               =0      !DATA BEING READ OUT OF THE CACHE
*       REG             =1      !SOURCE A REGISTER
        CONST           =2      !IMMEDIATE CONSTANT OR CACHE ADR IF C ADR SEL
                                !IS SET

$REG R ADR<0:4>                 !REGISTER READ ADDRESS

*                       =0

$REG W ADR<0:4>                 !REGISTER WRITE ADDRESS

*                       =0

$REL INTERPROC INT

*                       =0
                        =1      !INTER PROCESSOR INTERRUPT HAS BEEN SERVICED.
                                !ALLOW THE SWITCH TO SEND ANOTHER!

$RESET INSTR QUEUE

*                       =0
                        =1      !CLEAN IT OUT

$SET C MODIFY BIT

        NO              =0      !DON'T SET CACHE MODIFY BIT IF WRITE
*                       =1

$SET EWAR

*                       =0
                        =1      !SET EBOX WRITE ADDRESS REGISTER TO THE
                                !ADDRESS OF THE WORD BEING READ OUT OF THE
                                !CACHE NOW.
```

```
$SET OP

*                       =0
                        =1      !SET EBOX OPERAND REGISTER

$SRC REG CTL<0:1>

*       REG ADR         =1      !SRC (A OR B) REG ADR="REG R ADR<0:4>"
        OD1             =2      !       "             ="OD1 REG ADR<0:4>"
        ADD             =3      !       "             =SUM OF ABOVE TWO FIELDS

$SRC REG OUT SEL

*       A               =0
        B               =1

$SWITCH START W

*                       =0
                        =1      !START A MEMORY WRITE THROUGH THE SWITCH.

$T R ADR<0:2>                   !T REGISTER STACK READ ADDRESS

*                       =0

$T W ADR<0:2>                   !T REGISTER STACK WRITE ADDRESS

*                       =0

$TRANS ADR

*                       =0      !DO A VIRTUAL TO PHYSICAL ADDRESS TRANSLATION
                                !ON THE MEMORY ADDRESS
                        =1      !DON'T TRANSLATE THE MEMORY ADDRESS - DO
                                !ABSOLUTE MEMORY ADDRESSING

$UPDATE AT LRU BITS

*                       =0
                        =1      !UPDATE THE ADDRESS TRANSLATION LRU BITS

$UPDATE C LRU BITS

*                       =0
                        =1      !UPDATE THE CACHE LRU BIT

$W TRANS

*                       =0
                        =1      !WRITE INTO THE ADDRESS TRANSLATION CACHE
```

A7. I-Sequencer Micro-Code Macros

```
%A+B(LEG)                [ADD F=A+B,ADD LEG B=LEG]
%B(LEG)                  [ADD F=B,ADD LEG B=LEG]
%BR                      [MIC BR ADR]
%CR                      [C OPERATION=1,UPDATE C LRU BITS=1,OUT SEL=C]
%CR OP←F(F)              [CR,SET OP=1,INDEX TRANS(T),ADD F=B,ADD LEB B=F]
%CR OP←R+F(S,F)          [CR,SET OP=1,REG R ADR=S,INDEX TRANS(T),A+B(F)]
%CR OP←RS(SEL)           [CR,SET OP=1,INDEX TRANS(SEL)]
%CR OP←RS+F(SEL,F)       [CR,SET OP=1,INDEX TRANS(SEL),A+B(F)]
%CR R←F(F)               [CR,REG W(D),B(F),INDEX TRANS(T)]
%CR R←R(D,S)             [CR,REG W(D),INDEX TRANS(T),REG R ADR=S]
%CR R←R+F(D,S,F)         [CR,REG W(D),A+B(F),INDEX TRANS(T),REG R ADR=S]
%CR R←RS(D,SEL)          [CR,REG W(D),INDEX TRANS(SEL)]
%CR R←RS+F(D,SEL,F)      [CR,REG W(D),INDEX TRANS(SEL),A+B(F)]
%IND REG                 [ADD LOAD IND REG=1]
%INDEX TRANS(SEL)        [ADD LEG A=INDEX REG,TRANS ADR=1,INDEX REG ADR SEL=SEL]
%INDEX(SEL)              [ADD LEG A=INDEX REG,INDEX REG ADR SEL=SEL]
%REG W(ADR)              [IBOX REG W=1,REG W ADR=ADR]
%TR(S)                   [T R ADR=S]
%TW(D)                   [T W ADR=D]
%TWR(D,S)                [TW(D),TR(S)]
%T←F(F)                  [TW(D),B(F)]
%T←R(D,S)                [TW(D),INDEX(T),REG R ADR=S]
%T←R+F(D,S,F)            [TW(D),A+B(F),INDEX TRANS(T),REG R ADR=S]
%T←RS(D,SEL)             [TW(D),INDEX(SEL)]
%T←RS+F(D,SEL,F)         [TW(D),INDEX(SEL),A+B(F)]
```

A8. I-Sequencer Micro-Code

```
!EVALUATE A SHORT OPERAND (X=0)

!REG=0,I=0      REGISTER-DIRECT
        SET OP=1,SRC REG CTL=OD,DONE
!REG=0,I=1      REGISTER-INDIRECT
        CR R←RS(R1,OD),TW(T1),IND REG,JSR(REG IND OP)
!REG=1,I=0      SHORT-CONSTANT
        SET OP=1,OUT SEL=CONST,DONE
!REG=1,I=1      ILLEGAL
        BR=ILLEGAL OP
!REG=2          ILLEGAL
        BR=ILLEGAL OP
!REG=3-31,I=0   SHORT-INDEXED
        CR OP←RS+F(OD,SO),DONE
!REG=3-31,I=1   SHORT-INDEXED-INDIRECT
        CR R←RS+F(R1,OD,SO),JSR(MEM IND OP)
```

```
!EVALUATE A LONG OPERAND WITH A FIXED BASE (X=1,XI=0,M=0,V=0)

!REG=0,I=0      REGISTER-DIRECT

        CR OP←RS+F(OD,FIX BASE),DONE;

!REG=0,I=1      REGISTER-INDIRECT

        CR R←RS(R1,OD),TW(T1),IND REG,JSR(REG IND);
        CR OP←R+F(R1,FIX BASE),INDEX SHIFT,DONE;

!REG=1,I=0      LONG-CONSTANT

        SET OP=1,OUT SEL=CONST,IMMED CONST LONG=1,DONE;

!REG=1,I=1      LONG ABSOLUTE ADDRESSING

        CR OP←F(FIX BASE),DONE;

!REG=2          ILLEGAL

        BR=ILLEGAL OP;

!REG=3-31,I=0   SHORT-INDEXED

        CR R←RS+F(R1,OD,SO);
        BR=L1;                              !WAIT FOR CACHE READ

!REG=3-31,I=1   SHORT-INDEXED-INDIRECT

        CR R←RS+F(R1,OD,SO),JSR(MEM IND);

L1:     CR OP←R+F(R1,FIX BASE),INDEX SHIFT,DONE;
```

```
!EVALUATE A LONG OPERAND WITH A VARIABLE BASE (X=1,XI=0,M=0,V=1)

!REG=0,I=0      REGISTER-DIRECT

        T←RS+F(T1,OD,VAR BASE),INDEX SHIFT;
        CR OP←RS+F(VB REG,T),TR(T1),DONE;

!REG=0,I=1      REGISTER-INDIRECT

        CR R←RS(R1,OD),TW(T1),IND REG,JSR(REG IND);
        T←R+F(T1,R1,VAR BASE),INDEX SHIFT;
        CR OP←RS+F(VB REG,T),TR(T1),DONE;

!REG=1,I=0      LONG-CONSTANT

        SET OP=1,OUT SEL=CONST,IMMED CONST LONG=1,DONE;

!REG=1,I=1      LONG ABSOLUTE ADDRESSING

        CR OP←RS+F(VB REG,VAR BASE),DONE;

!REG=2          ILLEGAL

        BR=ILLEGAL OP;

!REG=3-31,I=0   SHORT-INDEXED

        CR R←RS+F(R1,OD,SO);
        BR=L1;

!REG=3-31,I=1   SHORT-INDEXED-INDIRECT

        CR R←RS+F(R1,OD,SO),JSR(MEM IND);

L1:     T←R+F(T1,R1,FIX BASE),INDEX SHIFT;
        CR OP←RS+F(VB REG,T),TR(T1),DONE;
```

```
!EVALUATION OF AN INDIRECT ADDRESS CHAIN WITH THE RESULT PUT IN R1.

MEM IND:                                    !GO INDIRECT THROUGH A MEMORY LOC.
        NOP;
        CR R←R(R1,R1),TW(T1),IND REG;
        INDRET(RI01,RI10,RI11);

REG IND:                                    !GO INDIRECT THROUGH A REGISTER.
        INDRET(RI01,RI10,RI11);

RI01:   CR R←R(R1,R1),TW(T1),IND REG;       !R=0, I=1
        INDRET(RI01,RI10,RI11);

RI10:   CR R←RS+F(R1,IND,T),TR(T1);         !R≠0, I=0
        RETURN;

RI11:   CR R←RS+F(R1,IND,T),TR(T1);         !R≠0, I=1
        NOP;
        CR R←R(R1,R1),TW(T1),IND REG;
        INDRET(RI01,RI10,RI11);
```

A9. E-Sequencer Micro-Code Fields

```
$A01 SEL<0:2>                    !MMXCTL. CSACTL.  SELECT CSA A01 INPUT.

        NA              =0
        NA*2            =1
        A               =2
        A*2             =3
        MINUS ONE       =4
        Z               =6

$A23 SEL<0:2>                    !MMXCTL. CSACTL.  SELECT CSA A23 INPUT.

        NA*4            =0
        NA*8            =1
        A*4             =2
        A*8             =3
        MINUS ONE       =4
        Z               =6

$AUTO MERGE                      !MXMRG2. EN MEANS IBOX CONTROLS MERGE

*       DIS             =0       !MERGING UNDER EBOX CONTROL
        EN              =1       !MERGING UNDER IBOX CONTROL
```

```
$BC SEL<0:5>                    !EBCMUX. BRANCH CONDITION SELECT

        SW IN BOUNDS    =0
        SW N            =1
        SW Z L          =2
        SW V L          =3
        SW CO           =4
        SW LE           =5
        FLOAT FIX L     =6
                        =7

*       NEVER           =8
        PRE V L         =9
        POST V L        =10
        EXP N           =11
        EXP V L         =12
                        =13
                        =14
                        =15

        PAUSE EBOX      =16
        MANT Z L        =17
        MANT V L        =18
        I ALL Z         =19
        F ALL Z         =20
        B ALL Z         =21
        COUNT DONE      =22
```

```
        SW IN BOUNDS L  =32
        SW N L          =33
        SW Z            =34
        SW V            =35
        SW CO L         =36
        SW LE L         =37
        FLOAT FIX       =38
                        =39

        ALWAYS          =40
        PRE V           =41
        POST V          =42
        EXP N L         =43
        EXP V           =44
                        =45
                        =46
                        =47

        PAUSE EBOX L    =48
        MANT Z          =49
        MANT V          =50
        I ALL Z L       =51
        F ALL Z L       =52
        B ALL Z L       =53
        COUNT DONE L    =54

$BR ADR<0:11>                   !EBXCTL. BRANCH ADDRESS

*                       =0

$BR DEST<0:2>                   !EBXCTL. BRANCH CONTROL

        RETURN          =0
                        =1
        START           =2
*       BRANCH          =3
        SHIFT           =4
        FA              =5
        FIXREG          =6
        ALU COND        =7
```

```
$BR NWAY<0:1>                       !EBXCTL. NUMBER OF BRANCH DESTINATIONS

        2 WAY           =0          !TWO-WAY BRANCH
        4 WAY           =1          !FOUR-WAY BRANCH
        8 WAY           =2          !EIGHT-WAY BRANCH
*       16 WAY          =3          !SIXTEEN-WAY BRANCH

$BYTE PTR PE                        !SHFCTL. LOAD QW2 AND QW3 OF R

*       HOLD            =0
        LOAD            =1

$SW CO PE                           !FACTL. LOAD SW CO

        HOLD            =0
*       LOAD            =1

$COND STATUS PE                     !STATUS. LOAD CONDITION CODES

*       HOLD            =0
        LOAD            =1

$COND STATUS SEL                    !STATUS. SELECT STATUS TO SAVE

*       COND CODES      =0          !NORMAL CONDITIONS CODES
        MC              =1          !MICRO-CONSTANT

$DONE                               !FIXGEN. LAST MICRO-CYCLE

*                       =0
                        =1

$EBOX CONTROL PE                    !EBOX2. LOAD CONTROL BITS FROM IBOX.

*       HOLD            =0
        LOAD            =1

$EXP COMPL                          !EXPBOX. COMPLEMENT EXPONENT

*                       =0
                        =1

$EXP SUM PE                         !EXPBOX. LOAD EXP SUM REGISTER

*       HOLD            =0
        LOAD            =1
```

```
$FA A IN SEL<0:3>                    !3INADD. SELECT FOR FA A LEG

*       A               =0
        B               =1
        Q               =2
        MC              =3
        Z               =4
        A CO            =8
        B CO            =9
        Q CO            =10
        MC CO           =11
        Z CO            =12

$FA B IN SEL                         !3INADD. SELECT FOR FA B LEG

*       B               =0      !OP B
        S               =1      !SUM OUTPUT FROM CSA

$FA CTL SEL<0:2>                     !FACTL. SELECT FA CTL SOURCE

*       EBOX            =0
        DIV             =1      !DIVISION
        RND             =2      !ROUNDING
        MULT            =3      !MULTIPLY
        SAVED CO        =4      !ADD CARRY (SAVED)
        GUARD           =5      !ADD GUARD
        CO STATUS       =6      !ADD CO FROM STATUS WORD
                        =7
```

```
$FA CTL<0:5>                    !FACTL. FA MODE/FUNCTION CONTROL

        A+0             =0
        A+1             =1
        A+B             =12
        A+B+1           =13
        A-B-1           =18
        A-B             =19
        A*2             =24
        A*2+1           =25
        A-1             =30

        NA              =32
        NA AND B        =34
        NA AND NB       =36
        Z               =38
        NA OR B         =40
        B               =42
        A XNOR B        =44
        A AND B         =46
        NA OR NB        =48
        A XOR B         =50
        NB              =52
        A AND NB        =54
        MINUS ONE       =56
        A OR B          =58
        A OR NB         =60
*       A               =62

$FIXUP EN                       !FIXREG. ENABLE FIXUP IF DONE

*       DIS             =0
        EN              =1
```

```
$FIXUP REG SEL<0:3>              !FIXREG. SELECT FIXUP REGISTER INPUT

*       NEVER           =0
        ALWAYS          =1
                        =2
        PRE V           =3
        MANT V          =4
        SW Z            =5
        SW IN BOUNDS    =6
        FLOAT FIX       =7

                        =8
                        =9
                        =10
                        =11
                        =12
                        =13
                        =14
                        =15

$FIXUP REG TEST                  !FIXREG. ENABLE TESTING OF FIXUP REGISTERS

*       NO REG          =0
        REG             =1

$FIXUP REG0 CLK EN               !FIXREG. ENABLE SETTING FIXUP REG 0

*       DIS             =0
        EN              =1

$FIXUP REG1 CLK EN               !FIXREG. ENABLE SETTING FIXUP REG 1

*       DIS             =0
        EN              =1

$FIXUP REG2 CLK EN               !FIXREG. ENABLE SETTING FIXUP REG 2

*       DIS             =0
        EN              =1

$FIXUP REG3 CLK EN               !FIXREG. ENABLE SETTING FIXUP REG 3

*       DIS             =0
        EN              =1
```

```
$G SEL<0:3>                         !ROUND. SELECT GUARD BIT INPUT AND MODE.

        BZC             =0          !USE BOTTOM ZEROES COUNT (EG. IN PRENORM)
        ADD             =1          !FLOATING ADD
        POST            =2          !POSTNORMALIZATION
        DIVIDE          =3          !FLOATING DIVIDE
        Z               =4          !CLEAR
*       HOLD            =8          !HOLD

$INTERRUPT IBOX                     !FIXGEN.

*       NEVER           =0
        NO FIX          =1          !IFF NO FIXUP

$JSR                                !EBXCTL. JUMP TO SUBROUTINE

*                       =0          !JMP OR RET.
                        =1          !JSR

$LOGICAL SHIFT                      !SHFCTL. LOGICAL/ARITHMETIC SHIFT

                        =0          !DRAG BIT IS SHIFT SIGN
*                       =1          !DRAG BIT IS SIGN OF SHIFT A IN

$MC COND<0:3>                       !STATUS. CONDITION MICRO-CONSTANT

*                       =0

$MC EXP<0:11>                       !EXPBOX. EXPONENT MICRO-CONSTANT

*                       =0

$MC REPT<0:7>                       !REPT.  REPTITION MICRO-CONSTANT

*                       =0

$MC SHIFT<0:5>                      !SHFCTL. SHIFT MICRO-CONSTANT

*                       =0

$MC<0:35>                           !3INADD. EBOX MICRO-CONSTANT

*                       =0

$MERGE EXP                          !MXMRG1. FOR EXP USE (MM SEL OR 1)

*                       =0
                        =1
```

```
$MERGE LEN                          !MXMRG2. QW OR HW MERGE.

*       QW              =0
        HW              =1

$MERGE QW0                          !MXMRG2. FOR QW0 USE (MM SEL OR 2)

*                       =0
                        =1

$MERGE QW1                          !MXMRG2. FOR QW1 USE (MM SEL OR 2)

*                       =0
                        =1

$MERGE QW2                          !MXMRG2. FOR QW2 USE (MM SEL OR 2)

*                       =0
                        =1

$MERGE QW3                          !MXMRG2. FOR QW3 USE (MM SEL OR 2)

*                       =0
                        =1

$MM EN                              !MXMRG1. ENABLE MUX MERGER OUTPUT

        DIS             =0
*       EN              =1

$MM SEL<0:2>                        !MXMRG1. SELECT MUX MERGER

*       FA              =0
                        =1
        SHIFT           =2
        EXP             =3
        FA/2            =4
        MUL             =5
        DIV             =6
                        =7

$MULT EN                            !MMXCTL. ENABLE MULTIPLY OPERATION

*       DIS             =0
        EN              =1      !Q REGISTER CONTROLS 3-INPUT ADDER

$OP A ADR<0:4>                      !ERFC1. EBOX A REGISTER ADR

*       Z               =0      !GARBAGE REGISTER
```

```
$OP B ADR<0:4>                  !ERFC1. EBOX B REGISTER ADR

*       Z               =0      !GARBAGE REGISTER

$OP W ADR<0:4>                  !ERFC1. EBOX WRITE REGISTER ADR

*       Z               =0      !GARBARGE REGISTER

$POST MAX PE                    !STATUS. LOAD MAX POSTNORM AMOUNT

*       HOLD            =0
        LOAD            =1

$PRE EN                         !SHIFTR/SHFBOX.  ENABLE PRENORMALIZATION

*       DIS             =0
        EN              =1

$PRE MAX PE                     !STATUS. LOAD MAX PRENORM AMOUNT

*       HOLD            =0
        LOAD            =1

$Q MODE<0:2>                    !Q. CONTROL LINES TO Q REG

*       LOAD            =0      !PARALLEL LOAD
        RIGHT 1         =1      !SHIFT RIGHT 1
        LEFT 1          =2      !SHIFT LEFT 1
        HOLD            =3      !HOLD
        RIGHT 4         =4      !SHIFT RIGHT 4

$RECOMP A01                     !CSACTL. COMPLEMENT A LEG OF CSA

                        =0
*                       =1


$RECOMP A23                     !CSACTL. COMPLEMENT B LEG OF CSA

                        =0
*                       =1
```

```
$REPT CTR MODE<0:1>                    !REPT. REPITITION CTR MODE

        LOAD            =1
        DEC             =2
*       HOLD            =3

$REPT CTR SEL                          !REPT. SELECT FA CTR OR MC CTR

        FA              =0      !FA CTR
*       MC              =1      !MICRO-CONSTANT CTR

$RESULT SEL                            !FIXGEN. CONTROL X RESULT SIGNAL

        ALWAYS          =0      !RESULT ALWAYS READY
        NO BRANCH       =1      !RESULT READY IFF NOT BRANCH
*       NEVER           =2      !RESULT NEVER READY
        DONE            =3      !RESULT READY IFF DONE AND NOT FIXUP
        NO FIX          =4      !RESULT READY IFF NOT FIXUP

$RLSB PE                               !ROUND. LOAD LS BIT OF R<0:35>

        HOLD            =0
*       LOAD            =1

$RND MODE<0:1>                         !ROUND. ROUNDING MODE
        STABLE          =0
        CEILING         =1
*       FLOOR           =2

$SHIFT A IN SEL<0:1>                   !SHFBOX. SELECT SHIFTER A INPUT

*       A               =0      !A INPUT GETS A OP
        Z               =2      !A INPUT GETS ZERO
        BAQZM SIGN      =3      !A INPUT GETS BAQZM SIGN

$SHIFT B IN SEL<0:2>                   !SHFBOX. SELECT SHIFTER B INPUT

*       B               =0
        A               =1
        Q               =2
        Z               =3
        GB              =4      !MERGE GUARD BITS
        GA              =5      !MERGE GUARD BITS
        GQ              =6      !MERGE GUARD BITS
        GZ              =7      !MERGE GUARD BITS
```

```
$SHIFT CTL<0:6>                 !SHFCTL. SELECT SCNT SOURCE
        FZC             =0

*       Z               =16     !001 0000
        C20 BYTE LEN    =18     !001 0010
        C20 BYTE POS    =20     !001 0100
        C20 B QW3       =20
        EXP SUM         =22     !001 0110
        RIGHT 1         =24     !001 1000
        MC              =26     !001 1010
        B BYTE LEN      =28     !001 1100
        B BYTE POS      =29     !001 1101
        B QW3           =29
        DEST            =30     !001 1110

        IZC             =32     !010 0000

        36-C20 BYTE LEN =50     !011 0010
        36-C20 BYTE POS =52     !011 0100
        36-C20 B QW3    =52
        36-EXP SUM      =54     !011 0110
        36-B BYTE LEN   =60     !011 1100
        36-B BYTE POS   =61     !011 1101
        36-B QW3        =61     !011 1101

        POST            =68     !100 0100  POSTNORMALIZE.

$SHIFT SIGN                     !SHFCTL. DRAG BIT FROM EBOX MICRO-CODE

*                       =0      !DRAG BIT=0
                        =1      !DRAG BIT=1

$TEST STICKY EN                 !STICKY. TEST BZC≥36-LSHF-2

        DIS             =0
*       EN              =1

$TEST WRONG BRANCH              !WRONGB.  TEST WRONG BRANCH TAKEN

*       DIS             =0
        EN              =1

$TRANS A SEL<0:1>               !EREGF. A OP TRANSLATION SELECT

*       SW              =0      !STRAIGHT THROUGH
        FLOAT           =1      !SIGN EXTEND FLOATING POINT
        QW              =2      !QW TRANSLATION
        HW              =3      !HW TRANSLATION
```

```
$TRANS B SEL<0:1>                    !EREGF. B OP TRANSLATION SELECT

*       SW              =0           !STRAIGHT THROUGH
        FLOAT           =1           !SIGN EXTEND FLOATING POINT
        QW              =2           !QW TRANSLATION
        HW              =3           !HW TRANSLATION

$USE I A OP                          !EREGF. USE IBOX A OP INSTEAD OF R

*                       =0           !USE R
                        =1           !USE IBOX OP

$USE I B OP                          !EREGF. USE IBOX B OP INSTEAD OF R

*                       =0           !USE R
                        =1           !USE IBOX OP

$XBOX A SEL<0:1>                     !EXPBOX. SELECT XBOX ALU A LEG

        A EXP           =0           !LOAD A EXP
        EXP SUM         =1           !LOAD EXP SUM
*       HOLD            =2           !HOLD
```

```
$XBOX ALU CTL<0:5>                !EXPBOX. EXPBOX ALU MODE/FUNCTION

        A+0             =0
        A+1             =1
        A+B             =12
        A+B+1           =13
        A-B-1           =18
        A-B             =19
        A*2             =24
        A*2+1           =25
        A-1             =30

        NA              =32
        NA AND B        =34
        NA AND NB       =36
        Z               =38
        NA OR B         =40
        B               =42
        A XNOR B        =44
        A AND B         =46
        NA OR NB        =48
        A XOR B         =50
        NB              =52
        A AND NB        =54
        MINUS ONE       =56
        A OR B          =58
        A OR NB         =60
*       A               =62

$XBOX B SEL<0:1>                  !EXPBOX. SELECT XBOX ALU B LEG

        B EXP           =0        !LOAD B EXP
        SCNT/MC         =1        !LOAD EXP SUM
*       HOLD            =2        !HOLD

$XBOX SCNT SEL<0:1>               !EXPBOX. XBOX SCNT REG SELECT

        MC              =0        !LOAD MC
        SCNT            =1        !LOAD SCNT
*       HOLD            =2;       !HOLD
```

A10. E-Sequencer Micro-Code Macros

```
%AUTO MERGE[
        SHIFT(A,Z,DEST),
        FA(Z,B,B),
        AUTO MERGE ENB,MM SEL=FA]

%BR NZ DEC(CTR,ADR)[
        BR DEST=BRANCH
        BC SEL=COUNT DONE L,
        BR ADR="ADR",
        REPT CTR SEL="CTR",
        REPT CTR MODE=DEC]

%BR Z DEC(CTR,ADR)[
        BR DEST=BRANCH
        BC SEL=COUNT DONE,
        BR ADR="ADR",
        REPT CTR SEL="CTR",
        REPT CTR MODE=DEC]

%BR(COND,ADR)[
        BR DEST=BRANCH,
        BR SEL="COND",
        BR ADR="ADR"]

%CHECK BOC(R)[
        CSA A IN A*2,CSA B IN Z,
        FA(A,S,A+B),
        FIX SAVE(SW Z,"R")]

%CSA(A,B,CI)[
        A01 SEL="A",
        A23 SEL="B"
        FA A IN SEL="CI"]

%DONE(COND)[
        OP W ADR=IW,
        BR DEST=START,
        BR SEL="COND",
        DONE=1]

%FA SEL(SOURCE)[
        FA CTL SEL="SOURCE"]

%FA(A,B,C)[
        FA IN("A","B"),
        FA CTL="C"]
```

```
!MERGE BACK QW OR HW
!SHIFT A LEFT AS PER ADDRESS
!PUT B THROUGH FA
!MERGE A INTO B

!REPT. BRANCH NOT ZERO AND DEC
!SET UP BRANCH ADDRESS MUX
!BRANCH IF COUNT NOT DONE
!TO BRANCH ADDRESS
!SELECT COUNTER OUTPUT
!DECREMENT SELECTED COUNTER

!REPT. BRANCH NOT ZERO AND DEC
!SET UP BRANCH ADDRESS MUX
!BRANCH IF COUNT DONE
!TO BRANCH ADDRESS
!SELECT COUNTER OUTPUT
!DECREMENT SELECTED COUNTER

!BRANCH TO ADR IF COND IS TRUE
!SET UP BRANCH ADDRESS MUX
!SET UP BRANCH CONDITION MUX
!INPUT TO BRANCH ADDRESS MUX

!CHECK FOR POST BAD ONES COUNT
!CSA GIVES (A*2) XOR A
!A*2 IS ON A LEG, A IS ON CI LEG
!FIXUP TO R IFF
!       A+((A*2) XOR A)=0

!SELECT CSA A, B, AND CI LEGS
!A01 SEL CONTROLS THE A LEG
!A23 SEL CONTROLS THE B LEG
!FA A IN SEL CONTROLS THE CI LEG

!DONE IFF COND
!MAKE SURE IW IS WRITE ADDRESS
!SELECT START ADR ON ADR MUX
!SET UP BRANCH CONDITION MUX
!DONE IFF COND

!SELECT SOURCE OF FA CTL

!SELECT FA A IN, FA B IN, FA CTL
```

```
%FIXUP INIT[                                    !CLEAR ALL FIXUP REGISTERS
        FIXUP REG SEL=NEVER,                    !SELECT FIXUP MUX TO CLEAR
        FIXUP REG0 CK EN=1,                     !ENABLE REG 0 CLOCK
        FIXUP REG1 CK EN=1,                     !ENABLE REG 1 CLOCK
        FIXUP REG2 CK EN=1,                     !ENABLE REG 2 CLOCK
        FIXUP REG3 CK EN=1]                     !ENABLE REG 3 CLOCK

%FIXUP SAVE(COND,R)[                            !SAVE FIXUP COND IN FIXUP REG R
        FIXUP REG SEL="COND",                   !SELECT FIXUP COND
        FIXUP REG"R" CK EN]                     !ENABLE CLOCK OF FIXUP REG R

%FIXUP(REG?,COND,ADR)[                          !FIXUP TO ADR IFF COND (OR REG)
        BR DEST=START,                          !SELECT START ADR
        BR ADR="ADR",                           !ADR IS FIXUP ADDRESS
        FIXUP REG SEL="COND",                   !SELECT FIXUP CONDITION
        FIXUP REG TEST="REG?",                  !CONDITIONALLY TEST FIXUP REGS
        FIXUP EN=1]                             !ENABLE FIXUP

%FLOAT SW OUT(R,FIX R,ADR)[                     !OUTPUT SW FLOATING RESULT
        OPS(IW,"R",Z),
        XA(A-B),                                !ADJUST EXPONENT BY SHIFT CNT
        SHIFT(A,Z,Z),R←SHIFT,MERGE EXP=1,       !MERGE EXPONENT INTO SHIFT OUT
        FA(A,B,A+0),LOAD COND,                  !TEST MANTISSA CONDITIONS
        FIXUP SAVE(FLOAT FIX,"FIX R"),          !SAVE FLOAT FIX CONDITION
        FIXUP(REG,FLOAT FIX,"ADR"),             !FIXUP ON REGS OR FLOAT FIX
        DONE(ALWAYS),RESULT(NO FIX)]            !RESULT IFF NO FIX

%FLOAT SW POST(A,B,FIX R)[                      !POSTNORMALIZE A:B, USE FIX R
        SHIFT("A","B",POST),R←SHIFT,            !POSTNORMALIZE BY FZC
        CHECK BOC("FIX R"),                     !CHECK BAD ONES COUNT
        G SEL=POST,                             !RECOMPUTE GUARD BITS
        XBOX SEL(EXP SUM,SCNT/MC,SCNT)]         !SET UP FOR EXPONENT ADJUST

%HOLD CO[                                       !HOLD CO IN CO REGISTER
        CO COND PE=HOLD]

%IA[    2]                                      !REG ADR FOR A OP FROM IBOX
%IB[    3]                                      !REG ADR FOR B OP FROM IBOX

%IC[    4]                                      !REG ADR FOR C OP FROM IBOX

%ID[    5]                                      !REG ADR FOR D OP FROM IBOX

%IW[    2]                                      !LAST INSTR MUST WRITE IW
```

```
%JSR(COND,ADR)[                         !JSR TO ADR IFF COND
        BR DEST=BRANCH,                 !SET UP BRANCH ADDRESS MUX
        BR SEL="COND",                  !SET UP BRANCH COND MUX
        BR ADR="ADR",                   !INPUT TO BRANCH ADDRESS MUX
        JSR=1]                          !ENABLE JSR

%LOAD BYTE PTR[                         !SET UP EXTERNAL BYTE PTR REG
        BYTE PTR PE=LOAD]

%LOAD CO[                               !LOAD CO
        SW CO PE=LOAD]                  !ENABLE LOADING OF CO REGISTER

%LOAD COND[                             !LOAD CONDITION STATUS
        COND STATUS SEL=COND CODES,     !SELECT COND STATUS INPUT
        COND STATUS PE=LOAD]            !ENABLE LOADING OF COND STATUS

%LOAD CONTROL[                          !LOAD CONTROL BITS FROM IBOX
        EBOX CONTROL PE=LOAD]

%LOAD REPT(CNT)[                        !REPT. LOAD REPITION COUNTERS
        REPT CTR MODE=LOAD,
        MC REPT="CNT"]

%MERGE(LEN)[                            !MERGE OPERAND INTO R
        MERGE LEN="LEN",                !QW OR HW MERGE
        AUTO MERGE]                     !ENABLE AUTO MERGE

%MULTIPLY[                              !SET UP MULTIPLY CYCLE
        TRANS A SEL=FLOAT,              !TRANSLATE MULTIPLICAND
        MULT EN=1,                      !ENABLE MULTIPLY CONTROL OF CSA
        FA(B CO,S,A+B),FA SEL(MULT),    !SET UP FA TO MULTIPLY
        Q MODE=RIGHT 4]                 !SHIFT MULTIPLIER RIGHT 4

%OPS(W,A,B)[                            !SET UP THREE REGISTER ADRS
        OP W ADR="W",                   !WRITE REGISTER ADDRESS
        OP A ADR="A",                   !READ REGISTER ADDRESS A
        OP B ADR="B"]                   !READ REGISTER ADDRESS B

%RESULT(COND)[                          !SET RESULT ON THREE CONDS:
        RESULT SEL="COND"]              !ALWAYS, NEVER, OR IFF NO FIXUP

%RET(COND)[                             !RET IFF COND
        BR DEST=RETURN,                 !SET UP BRANCH ADDRESS MUX
        BR SEL="COND"]                  !SET UP BRANCH COND MUX
```

```
%R←SHIFT[                                   !R OUTPUT GETS SHIFT
        MM SEL=SHIFT]                       !SELECT SHIFT ON MUX MERGER

%SHIFT(A,B,C)[                              !SHIFT AB CONTROLLED BY C
        SHIFT A IN SEL="A",                 !SELECT SHIFTER A LEG
        SHIFT B IN SEL="B",                 !SELECT SHIFTER B LEG
        SHIFT CTL="C"]                      !SELECT SHIFTER CONTROL

%START QW HW[                               !START QW OR HW INSTRUCTION
        TAKE,                               !TAKE A AND B OPS FROM IBOX
        SAVE CONTROL,                       !SAVE CONTROL SIGNALS FROM IBOX
        FIXUP INIT]                         !ALWAYS INITIALIZE FIXUP REG
%TAKE A[                                    !TAKE A OPERAND FROM IBOX
        USE I A OP=1]

%TAKE B[                                    !TAKE B OPERAND FROM IBOX
        USE I B OP=1]

%TAKE[                                      !TAKE A AND B OPS FROM IBOX
        TAKE A,
        TAKE B]

%TEST BOUNDS(CONSTANT)[                     !SET UP TEST FOR MC>X≥Ø
        MC=-"CONSTANT",                     !SET MC=-CONSTANT
        FA(MC,B,A+B)]                       !ADD MC TO B YIELDING B-MC

%TRANS(LEN)[                                !TRANSLATE A AND B
        TRANS A SEL="LEN",
        TRANS B SEL="LEN"]

%XA(C)[                                     !CONTROL EXPONENT BOX ALU
        XBOX ALU CTL="C"]

%XBOX SEL(A,B,S)[                           !SELECT EXPONENT BOX ALU INPUTS
        XBOX A SEL="A",                     !SELECT A INPUT
        XBOX B SEL="B",                     !SELECT B INPUT
        XBOX SCNT SEL="SCNT"]               !SELECT SCNT INPUT
```

A11. E-Sequencer Micro-Code

```
!ADD Q, ADD H

!*****************************************************************************

ADD Q:

OPS(4,Z,Z),                             !SET UP TO WRITE DESTINATION INTO R4
START QW HW;                            !START QW HW INSTRUCTION

OPS(5,IA,IB),TRANS(QW),                 !TRANSLATE OPERANDS
TAKE A,                                 !TAKE DESTINATION INTO 4
FA(A,B,A+B),LOAD COND,                  !ADD AND SAVE STATUS
FIXUP SAVE(SW V,0);                     !SAVE FIXUP CONDITION

OPS(IW,5,4),
MERGE(QW),                              !MERGE RESULT INTO DESTINATION
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER A RESULT
FIXUP(REG,NEVER,INT OVFL);              !FIXUP IFF OVFL

!*****************************************************************************

ADD H:

OPS(4,Z,Z),                             !SET UP TO WRITE DESTINATION INTO R4
START QW HW;                            !START QW HW INSTRUCTION

OPS(5,IA,IB),TRANS(HW),                 !TRANSLATE OPERANDS
TAKE A,                                 !TAKE DESTINATION INTO 4
FA(A,B,A+B),LOAD COND,                  !ADD AND SAVE STATUS
FIX SAVE(HW V,0);                       !SAVE FIXUP CONDITION

OPS(IW,5,4),
MERGE(HW),                              !MERGE RESULT INTO DESTINATION
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER A RESULT
FIXUP(REG,NEVER,INT OVFL);              !FIXUP IFF REG OVFL
```

```
!ADD S, ADD D

!*****************************************************************************

ADD S:
OPS(IW,IA,IB),TAKE,                     !TAKE BOTH OPERANDS FROM IBOX
FA(A,B,A+B),LOAD COND,                  !ADD AND SAVE STATUS
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER A RESULT
FIXUP(SW V,INT OVFL);                   !FIXUP IFF SW V

!*****************************************************************************

ADD D:

%MSA    [IA]                            !MOST SIGNIFICANT WORD OF OPERAND A
%MSB    [IB]                            !MOST SIGNIFICANT WORD OF OPERAND B
%LSA    [IC]                            !LEAST SIGNIFICANT WORD OF OPERAND A
%LSB    [ID]                            !LEAST SIGNIFICANT WORD OF OPERAND B

OPS(LSA,MSA,MSB),TAKE;                  !TAKE MOST SIGNIFICANT PARTS FIRST

OPS(6,LSA,LSB),TAKE,                    !TAKE LEAST SIGNIFICANT PARTS
FA(A,B,A+B),LOAD CO,                    !ADD AND SAVE CARRY
RESULT(ALWAYS);                         !DELIVER LEAST SIGNIFICANT RESULT

OPS(IW,MSA,MSB),
FA(A,B,A+B),FA SEL(SAVED CO),           !ADD
LOAD COND,                              !SAVE STATUS
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER A RESULT
FIXUP(SW V,INT OVFL);                   !FIXUP IFF SW V
```

```
!INC Q, INC H

!*****************************************************************************

INC Q:

OPS(Z,Z,Z),
START QW HW;                          !START QW HW INSTRUCTION

OPS(4,IB,Z),TRANS(QW),                !TRANSLATE OPERANDS
FA(A,Z,A+1),LOAD COND,                !INCREMENT AND SAVE STATUS
FIXUP SAVE(SW V,0);                   !SAVE FIXP CONDITION

OPS(IW,4,IA),
MERGE(QW),                            !MERGE QW INTO OUTPUT
DONE(ALWAYS),RESULT(ALWAYS),          !ALWAYS DELIVER THIS RESULT
FIXUP(REG,NEVER,INT OVFL);            !FIXUP IFF OVFL

!*****************************************************************************

INC H:

OPS(Z,Z,Z),
START QW HW;                          !START QW HW INSTRUCTION

OPS(4,IB,Z),TRANS(HW),                !TRANSLATE OPERANDS
FA(A,Z,A+1),LOAD COND,                !INCREMENT AND SAVE STATUS
FIXUP SAVE(SW V,0);                   !SAVE FIXUP CONDITION

OPS(IW,4,IA),
MERGE(HW),                            !MERGE HW INTO OUTPUT
DONE(ALWAYS),RESULT(ALWAYS),          !ALWAYS DELIVER THIS RESULT
FIXUP(REG,NEVER,INT OVFL);            !FIXUP IFF OVFL
```

```
!INC S, INC D

!*****************************************************************************

INC S:

OPS(IW,IA,Z),TAKE A,                    !USE ONLY OPERAND A
FA(A,Z,A+1),LOAD COND,                  !INCREMENT AND SAVE STATUS
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER THIS RESULT
FIXUP(NO REG,SW V,INT OVFL);            !FIXUP IFF SW V

!*****************************************************************************

INC D:

%MS     [1B]                            !MOST SIGNIFICANT WORD
%LS     [5]                             !LEAST SIGNIFICANT WORD

OPS(LS,Z,MS),TAKE B;                    !TAKE MS AS THE B OPERAND

OPS(Z,Z,LS),TAKE B,                     !TAKE LS AS THE B OPERAND
FA(Z,B,A+B+1),LOAD CO,                  !INCREMENT LOW HALF AND SAVE CARRY
RESULT(ALWAYS);                         !ALWAYS DELIVER THIS RESULT

OPS(IW,MS,Z),                           !USE THE SECOND HALF OF THE OPERAND
FA(A,B,A+0),FA SEL(SAVED CO),           !ADD THE SAVED CARRY OUT
LOAD COND,                              !SAVE THE STATUS
DONE(ALWAYS),RESULT(ALWAYS),            !ALWAYS DELIVER THIS RESULT
FIXUP(NO REG,SW V,INT OVFL);            !FIXUP IFF OVFL
```

```
FADD FR S:

%SMALL   [4]                                    !SMALLER OF IA AND IB
%IA+IB   [5]                                    !INITIAL RESULT IA+IB
%POST    [6]                                    !RESULT OF POSTNORMALIZATION

%PRE     [0]                                    !PRE OVERFLOW FIXUP REGISTER
%BOC     [1]                                    !BAD ONES COUNT FIXUP REGISTER
%FFIX    [2]                                    !FLOAT FIX FIXUP REGISTER

OPS(Z,IA,IB),TAKE,
XBOX SEL(A EXP,B EXP,HOLD),XA(A-B),             !SUBTRACT EXPONENTS
EXP SUM PE=LOAD,                                !SAVE EXPONENT DIFFERENCE
FIXUP INIT;                                     !INITIALIZE FIXUP REGISTERS

OPS(SMALL,IA,IB),TRANS(FLOAT,FLOAT),
PRE EN=1,SHIFT(Z,B,36-EXP SUM),R←SHIFT,         !PRENORMALIZE SMALLER
FIXUP SAVE(PRE V,PRE),                          !CHECK PRENORM OVERFLOW
G SEL=BZC,                                      !SAVE GUARD BITS
BR(EXP N,FADD FR S A SMALL);                    !BR ON EXP DIFFERENCE SIGN

OPS(IA+IB,IA,SMALL),TRANS(FLOAT,FLOAT),         !IB IS SMALLER
FA(A,B,A+B),FA SEL(GUARD),                      !ADD IA AND IB WITH GUARD BITS
G SEL=ADD,                                      !SAVE THE RECOMPUTED GUARD BITS
XA(A),EXP SUM PE=LOAD;                          !SAVE THE LARGER EXPONENT

FADD FR S JOIN:                                 !COME HERE IN BOTH CASES

OPS(POST,IA+IB,Z),
FLOAT SW POST(A,Z,BOC);                         !POSTNORMALIZE

FLOAT SW OUT(POST,FFIX,FADD FR S FIX);          !OUTPUT FLOATING RESULT POST

FADD FR S A SMALL:

OPS(IA+IB,SMALL,IB),TRANS(FLOAT,FLOAT),         !IA IS SMALLER
FA(A,B,A+B),                                    !ADD IA AND IB
XA(B),EXP SUM PE=LOAD,                          !SAVE THE LARGER EXPONENT
G SEL=ADD,                                      !RECOMPUTE GUARD BITS
BR(ALWAYS,FADD FR S JOIN);                      !RETURN TO FINISH FADD
```

```
FADD SR S:

%SMALL   [4]                                    !SMALLER OF IA AND IB
%IA+IB   [5]                                    !INITIAL RESULT IA+IB
%POST    [6]                                    !RESULT OF POSTNORMALIZATION
%ROUND   [7]                                    !RESULT OF ROUNDING

%PRE     [0]                                    !PRE OVERFLOW FIXUP REGISTER
%BOC     [1]                                    !BAD ONES COUNT FIXUP REGISTER
%FFIX    [2]                                    !FLOAT FIX FIXUP REGISTER
%RND V   [3]                                    !ROUNDING OVERFLOW FIXUP REG

OPS(Z,IA,IB),TAKE,
XBOX SEL(A EXP,B EXP,HOLD),XA(A-B),             !SUBTRACT EXPONENTS
EXP SUM PE=LOAD,                                !SAVE EXPONENT DIFFERENCE
FIXUP INIT;                                     !INITIALIZE FIXUP REGISTERS

OPS(SMALL,IA,IB),TRANS(FLOAT,FLOAT),
PRE EN=1,SHIFT(Z,B,36-EXP SUM),R←SHIFT,         !PRENORMALIZE SMALLER
FIXUP SAVE(PRE V,PRE),                          !CHECK PRENORM OVERFLOW
G SEL=BZC,                                      !SAVE GUARD BITS
BR(EXP N,FADD SR S A SMALL);                    !BR ON EXP DIFFERENCE SIGN

OPS(IA+IB,IA,SMALL),TRANS(FLOAT,FLOAT),         !IB IS SMALLER
FA(A,B,A+B),FA SEL(GUARD),                      !ADD IA AND IB WITH GUARD BITS
G SEL=ADD,                                      !SAVE THE RECOMPUTED GUARD BITS
XA(A),EXP SUM PE=LOAD;                          !SAVE THE LARGER EXPONENT

FADD SR S JOIN:                                 !COME HERE IN BOTH CASES

OPS(POST,IA+IB,Z),
FLOAT SW POST(A,Z,BOC);                         !POSTNORMALIZE

OPS(ROUND,POST,Z),
FA(A,B,A+B),FA SEL(RND),RND MODE=STABLE,        !PERFORM STABLE ROUNDING
FIXUP SAVE(MANT V,RND V);                       !CHECK ROUNDING OVERFLOW

FLOAT SW OUT(ROUND,FFIX,FADD SR S FIX);         !OUTPUT FLOATING RESULT ROUND

FADD SR S A SMALL:

OPS(IA+IB,SMALL,IB),TRANS(FLOAT,FLOAT),         !IA IS SMALLER
FA(A,B,A+B),                                    !ADD IA AND IB
XA(B),EXP SUM PE=LOAD,                          !SAVE THE LARGER EXPONENT
G SEL=ADD,                                      !RECOMPUTE GUARD BITS
BR(ALWAYS,FADD FR S JOIN);                      !RETURN TO FINISH FADD
```

```
FMULT FR S:

%MPCND   [1A]                                   !MULTIPLICAND
%MPYR    [1B]                                   !MULTIPLIER
%PROD    [4]                                    !PRODUCT REGISTER
%POST    [5]                                    !PRODUCT AFTER POSTNORMALIZE

%BOC     [0]                                    !BAD ONES COUNT FIXUP REGISTER
%FFIX    [1]                                    !FLOAT FIX FIXUP REGISTER

OPS(Z,MPCND,MPYR),TAKE
SHIFT(A,Z,Z),R←SHIFT,                           !PUT MULTIPLIER ON SHIFTER OUT
Q MODE=LOAD,                                    !LOAD Q REGISTER WITH MULTIPLIER
XBOX SEL(A EXP,B EXP,HOLD),XA(A+B),             !ADD EXPONENTS
EXP SUM PE=LOAD;

OPS(PROD,MPCND,Z),
XBOX SEL(EXP SUM,SCNT/MC,MC),MC EXP=128,        !CORRECT EXPONENT SUM
XA(A-B),EXP SUM PE=LOAD,
LOAD REPT(5),                                   !SET UP COUNTER
MULTIPLY;                                       !DO ONE MULTIPLY CYCLE HERE

FMULT FR S L1:

OPS(PROD,MPCND,PROD),
MULTIPLY,                                       !DO ANOTHER MULTIPLY CYCLE
BR NZ DEC(MC,FMULT FR S L1);                    !REPEAT MULTIPLY CYCLES

OPS(POST,PROD,Z),
FLOAT SW POST(A,Q,BOC);                         !POSTNORMALIZE A:Q

FLOAT SW OUT(POST,FFIX,FMULT FR S FIX);         !OUTPUT FLOATING RESULT POST
```

```
FMULT SR S:

%MPCND   [1A]                                   !MULTIPLICAND
%MPYR    [1B]                                   !MULTIPLIER
%PROD    [4]                                    !PRODUCT REGISTER
%POST    [5]                                    !PRODUCT AFTER POSTNORMALIZE
%ROUND   [6]                                    !RESULT OF ROUNDING

%BOC     [0]                                    !BAD ONES COUNT FIXUP REGISTER
%FFIX    [1]                                    !FLOAT FIX FIXUP REGISTER
%RND V   [3]                                    !ROUNDING OVERFLOW FIXUP REG

OPS(Z,MPCND,MPYR),TAKE
SHIFT(A,Z,Z),R←SHIFT,                           !PUT MULTIPLIER ON SHIFTER OUT
Q MODE=LOAD,                                    !LOAD Q REGISTER WITH MULTIPLIER
XBOX SEL(A EXP,B EXP,HOLD),XA(A+B),             !ADD EXPONENTS
EXP SUM PE=LOAD;

OPS(PROD,MPCND,Z),
XBOX SEL(EXP SUM,SCNT/MC,MC),MC EXP=128,        !CORRECT EXPONENT SUM
XA(A-B),EXP SUM PE=LOAD,
LOAD REPT(5),                                   !SET UP COUNTER
MULTIPLY;                                       !DO ONE MULTIPLY CYCLE HERE

FMULT FR S L1:

OPS(PROD,MPCND,PROD),
MULTIPLY,                                       !DO ANOTHER MULTIPLY CYCLE
BR NZ DEC(MC,FMULT SR S L1);                    !REPEAT MULTIPLY CYCLES

OPS(POST,PROD,Z),
FLOAT SW POST(A,Q,BOC);                         !POSTNORMALIZE A:Q

OPS(ROUND,POST,Z),
FA(A,B,A+B),FA SEL(RND),RND MODE=STABLE,        !PERFORM STABLE ROUNDING
FIXUP SAVE(MANT V,RND V);                       !CHECK ROUNDING OVERFLOW

FLOAT SW OUT(ROUND,FFIX,FMULT SR S FIX);        !OUTPUT FLOATING RESULT ROUND
```

```
!INC (SKIP,JUMP), DEC (SKIP,JUMP)

!*****************************************************************************

INC (SKIP,JUMP):

OPS(4,IA,Z),TAKE,                          !TAKE OP1 AS A OPERAND
LOAD CONTROL,                              !SAVE BRANCH CONDITION ETC
FA(A,B,A+1),                               !INCREMENT OP1
RESULT(ALWAYS);                            !ALWAYS DELIVER OP1+1

OPS(IW,4,IB),
FA(A,B,A-B),                               !COMPARE OP1+1 WITH OP2
LOAD COND,
TEST WRONG BRANCH=EN,                      !TEST WRONG BRANCH
DONE(SW V L);                              !DONE IFF NOT OVFL

BR(ALWAYS,INT OVFL);                       !NOT DONE SO GO TO OVERFLOW

!*****************************************************************************

DEC (SKIP,JUMP):

OPS(4,IA,Z),TAKE,                          !TAKE OP1 AS A OPERAND
LOAD CONTROL,                              !SAVE BRANCH CONDITION ETC
FA(A,B,A-1),                               !DECREMENT OP1
RESULT(ALWAYS);                            !ALWAYS DELIVER OP1-1

OPS(IW,4,IB),
FA(A,B,A-B),                               !COMPARE OP1-1 WITH OP2
LOAD COND,
TEST WRONG BRANCH=EN,                      !TEST WRONG BRANCH
DONE(SW V L);                              !DONE IFF NOT OVFL

BR(ALWAYS,INT OVFL);                       !NOT DONE SO GO TO OVERFLOW
```

```
!SKIP Q, SKIP H, SKIP S, SKIP D

!*****************************************************************************

SKIP Q:

OPS(Z,Z,Z),                                    !RECEIVE QW OPERANDS
START QW HW;

OPS(IW,IA,IB),TRANS(QW),                       !TRANSLATE QW OPERANDS
FA(A,B,A-B),                                   !COMPARE
TEST WRONG BRANCH=EN,                          !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT

!*****************************************************************************

SKIP H:

OPS(Z,Z,Z),                                    !RECEIVE HW OPERANDS
START QW HW;

OPS(IW,IA,IB),TRANS(HW),                       !TRANSLATE HW OPERANDS
FA(A,B,A-B),                                   !COMPARE
TEST WRONG BRANCH=EN,                          !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT

!*****************************************************************************

SKIP S:

OPS(IW,IA,IB),TAKE,                            !TAKE BOTH OPERANDS
LOAD CONTROL,                                  !LOAD BRANCH CONDITION ETC.
FA(A,B,A-B),                                   !COMPARE
TEST WRONG BRANCH=EN,                          !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT

!*****************************************************************************

SKIP D:

%MSA    [IA]                          !MOST SIGNIFICANT WORD OF OPERAND A
%MSB    [IB]                          !MOST SIGNIFICANT WORD OF OPERAND B
%LSA    [IC]                          !LEAST SIGNIFICANT WORD OF OPERAND A
%LSB    [ID]                          !LEAST SIGNIFICANT WORD OF OPERAND B

OPS(LSA,MSA,MSB),TAKE,                 !TAKE MOST SIGNIFICANT PARTS FIRST
LOAD CONTROL;                          !LOAD BRANCH CONDITION ETC.

OPS(6,LSA,LSB),TAKE,                   !TAKE LEAST SIGNIFICANT PARTS
FA(A,B,A-B),LOAD CO;                   !SUBTRACT AND SAVE CARRY

OPS(IW,MSA,MSB),
FA(A,B,A-B),FA SEL(SAVED CO),          !SUBTRACT
TEST WRONG BRANCH=EN,                  !TEST WRONG BRANCH
DONE(ALWAYS);                          !ALWAYS DELIVER A RESULT
```

```
!AND SKIP (Z,NZ) Q, AND SKIP (Z,NZ) H, AND SKIP (Z,NZ) S

!*****************************************************************************

AND SKIP (Z,NZ) Q:

OPS(Z,Z,Z),                                    !RECEIVE QW OPERANDS
START QW HW;

OPS(IW,IA,IB),TRANS(QW),                       !TRANSLATE QW OPERANDS
FA(A,B,A AND B),                               !AND THE OPERANDS
LOAD COND,
TEST WRONG BRANCH=EN,                          !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT

!*****************************************************************************

AND SKIP (Z,NZ) H:

OPS(Z,Z,Z),                                    !RECEIVE HW OPERANDS
START QW HW;

OPS(IW,IA,IB),TRANS(HW),                       !TRANSLATE HW OPERANDS
FA(A,B,A AND B),                               !AND THE OPERANDS
LOAD COND,
TEST WRONG BRANCH=EN,                          !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT

!*****************************************************************************

AND SKIP (Z,NZ) S:

OPS(IW,IA,IB),TAKE,                            !TAKE BOTH OPERANDS
LOAD CONTROL,                                  !LOAD BRANCH CONDITION ETC.
FA(A,B,A AND B),                               !AND THE OPERANDS
LOAD COND,
TEST WRONG BRANCHZ=EN,                         !TEST WRONG BRANCH
DONE(ALWAYS);                                  !NO RESULT
```

```
AND SKIP Z D:

%MSA     [IA]                    !MOST SIGNIFICANT WORD OF OPERAND A
%MSB     [IB]                    !MOST SIGNIFICANT WORD OF OPERAND B
%LSA     [IC]                    !LEAST SIGNIFICANT WORD OF OPERAND A
%LSB     [ID]                    !LEAST SIGNIFICANT WORD OF OPERAND B

OPS(LSA,MSA,MSB),TAKE,           !TAKE MOST SIGNIFICANT PARTS FIRST
LOAD CONTROL,                    !LOAD BRANCH CONDITION ETC.
FA(A,B,A AND B);                 !AND THE MOST SIGNIFICANT PARTS NOW

OPS(6,LSA,LSB),TAKE,             !TAKE LEAST SIGNIFICANT PARTS
FA(A,B,A AND B),                 !AND THE LEAST SIGNIFICANT PARTS
BR(SW Z,AND SKIP Z D L1);        !BRANCH IF (MSA AND MSB)=0

OPS(IW,Z,Z),
FA(A,B,MINUS ONE),               !PUT A NON-ZERO OUTPUT ON THE FA
LOAD COND,
TEST WRONG BRANCH=EN,            !TEST WRONG BRANCH
DONE(ALWAYS);                    !NO RESULT

AND SKIP Z D L1:                 !(MSA AND MSB)=0

OPS(IW,6,Z),                     !READ BACK (LSA AND LSB)
FA(A,B,A),                       !PUT OUT (LSA AND LSB) ON THE FA
LOAD COND,
TEST WRONG BRANCH=EN,            !TEST WRONG BRANCH
DONE(ALWAYS);                    !NO RESULT
```

```
AND SKIP NZ D:

%MSA     [IA]                    !MOST SIGNIFICANT WORD OF OPERAND A
%MSB     [IB]                    !MOST SIGNIFICANT WORD OF OPERAND B
%LSA     [IC]                    !LEAST SIGNIFICANT WORD OF OPERAND A
%LSB     [ID]                    !LEAST SIGNIFICANT WORD OF OPERAND B

OPS(LSA,MSA,MSB),TAKE,           !TAKE MOST SIGNIFICANT PARTS FIRST
LOAD CONTROL,                    !LOAD BRANCH CONDITION ETC.
FA(A,B,A AND B);                 !AND THE MOST SIGNIFICANT PARTS NOW

OPS(6,LSA,LSB),TAKE,             !TAKE LEAST SIGNIFICANT PARTS
FA(A,B,A AND B),                 !AND THE LEAST SIGNIFICANT PARTS
BR(SW Z L,AND SKIP NZ D L1);     !BRANCH IF (MSA AND MSB)≠0

OPS(IW,Z,Z),
FA(A,B,ZERO),                    !PUT A ZERO OUTPUT ON THE FA
LOAD COND,
TEST WRONG BRANCH=EN,            !TEST WRONG BRANCH
DONE(ALWAYS);                    !NO RESULT

AND SKIP NZ D L1:                !(MSA AND MSB)≠0

OPS(IW,6,Z),                     !READ BACK (LSA AND LSB)
FA(A,B,A),                       !PUT OUT (LSA AND LSB) ON THE FA
LOAD COND,
TEST WRONG BRANCH=EN,            !TEST WRONG BRANCH
DONE(ALWAYS);                    !NO RESULT
```

```
!SHIFT LEFT L Q, SHIFT LEFT L H

!*****************************************************************************

SHIFT LEFT L Q:

%D      [IA]                                    !DATA
%SCNT   [IB]                                    !SHIFT COUNT

OPS(Z,Z,Z),
START QW HW;

OPS(4,D,SCNT),TRANS(QW)
SHIFT(A,Z,B QW3),                               !SHIFT DATA
TEST BOUNDS(9),FIX SAVE(SW IN BOUNDS,0);        !SAVE 9>SCNT≥0 IN FIX REG 0

OPS(IW,4,Z),
MERGE(QW),                                      !MERGE QW INTO R
DONE(ALWAYS),RESULT(NO FIX),                    !DELIVER RESULT IFF NO FIX
FIXUP(REG,NEVER,SW LOGIC ZERO);                 !FIXUP IFF SCNT NOT IN BOUNDS
!*****************************************************************************

SHIFT LEFT L H:

%D      [IA]                                    !DATA
%SCNT   [IB]                                    !SHIFT COUNT

OPS(Z,Z,Z),
START QW HW;

OPS(4,D,SCNT),TRANS(HW)
SHIFT(A,Z,B QW3),                               !SHIFT DATA
TEST BOUNDS(18),FIX SAVE(SW IN BOUNDS,0);       !SAVE 9>SCNT≥0 IN FIX REG 0

OPS(IW,4,Z),
MERGE(HW),                                      !MERGE HW INTO R
DONE(ALWAYS),RESULT(NO FIX),                    !DELIVER RESULT IFF NO FIX
FIXUP(REG,NEVER,SW LOGIC ZERO);                 !FIXUP IFF SCNT NOT IN BOUNDS
```

```
!SHIFT LEFT L S, SHIFT LEFT L D
!*****************************************************************************

SHIFT LEFT L S:

%D      [1A]                                    !DATA
%SCNT   [1B]                                    !SHIFT COUNT

OPS(IW,D,SCNT),TAKE,
SHIFT(A,Z,B QW3),                               !SHIFT DATA
DONE(ALWAYS),RESULT(NO FIX),                    !DELIVER RESULT IFF NO FIX
TEST BOUNDS(36),                                !TEST 36>SCNT≥0
FIXUP(SW IN BOUNDS,SW LOGIC ZERO);              !FIXUP IFF SCNT NOT IN BOUNDS

!*****************************************************************************

SHIFT LEFT L D:

%D0     [1A]                                    !DATA WORD 0 (MOST SIGNIFICANT)
%SCNT   [1B]                                    !SHIFT COUNT
%D1     [1C]                                    !DATA WORD 1 (LEAST SIGNIFICANT)
%D0S    [5]                                     !D0 SHIFTED

OPS(D1,Z,IB),TAKE,                              !PREPARE TO ACCEPT IC
TEST BOUNDS(72),                                !TEST 72>SCNT≥0
LOAD BYTE PTR;                                  !SAVE SCNT FOR LATER

OPS(D0S,D1,Z),TAKE A,                           !ACCEPT IC
SHIFT(D1,Z,B QW3),                              !CREATE LOW ORDER WORD
BR(SW IN BOUNDS L,DW LOGIC ZERO),               !GIVE ZERO IF SCNT NOT IN BOUND
RESULT(NO BRANCH);                              !RESULT IFF SCNT IN BOUNDS

OPS(IW,D0,D1),                                  !SCNT IS IN BOUNDS
SHIFT(D0,D1,C20 B QW3),                         !CREATE HIGH ORDER WORD
DONE(ALWAYS),RESULT(ALWAYS);                    !ALWAYS DELIVER A RESULT
```

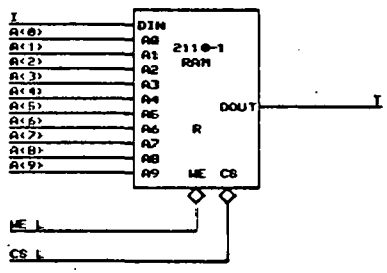
```
!LBYTE, DBYTE

!*****************************************************************************

LBYTE:

%BW     [IA]                                    !BYTE WORD
%BP     [IB]                                    !BYTE LEN, BYTE POS

OPS(4,BW,BP),TAKE,
SHIFT(A,Z,B BYTE POS),R←SHIFT;                  !LEFT JUSTIFY BYTE

OPS(IW,4,BP),
SHIFT(Z,A,B BYTE LEN),R←SHIFT,                  !SHIFT BYTE INTO RESULT WORD
DONE(ALWAYS),RESULT(ALWAYS);                    !ALWAYS DELIVER RESULT

!*****************************************************************************

DBYTE:

%DW     [IA]                                    !DESTINATION WORD = T:E:B
%BP     [IB]                                    !BYTE LEN, BYTE POS
%BW     [4]                                     !BYTE WORD = C:D:X

                                                !X=E, D=T, C=B

OPS(6,DW,BP),TAKE,                              !SET UP TO ACCEPT D
SHIFT(A,A,B BYTE POS),R←SHIFT,                  !R6 ← E:B:T
LOAD BYTE PTR;                                  !LOAD BYTE PTR REG FOR LATER

OPS(5,6,BP),
SHIFT(A,A,B BYTE LEN),R←SHIFT;                  !R5 ← B:T:E

OPS(7,4,5),TAKE A,                              !R4 ← BYTE WORD C:D:X
SHIFT(A,B,36-C20 BYTE LEN),R←SHIFT;             !R7 ← X:B:T

OPS(IW,7,Z),
SHIFT(A,A,36-C20 BYTE POS),R←SHIFT,             !RESULT ← T:X:B
DONE(ALWAYS),RESULT(ALWAYS);                    !ALWAYS DELIVER A RESULT
```

```
LBYTE INC:

%BW     [1A]                    !DATA WORD
%BP     [1B]                    !BYTE LEN, BYTE POS
%BL     [4]                     !BYTE LEN
%BA     [5]                     !BYTE POINTER ADDRESS
%LR     [6]                     !LBYTE RESULT
%NBP    [7]                     !NEW BYTE POINTER
%T1     [8]                     !POS+2*LEN
%T2     [9]                     !36-POS+2*LEN
%T3     [10]                    !BA ROTATED LEFT 6
%NBA    [11]                    !NEW BYTE POINTER ADDRESS

OPS(LR,BW,BP),TAKE,             !BEGIN LBYTE INTO LR
SHIFT(A,Z,B BYTE POS),R←SHIFT;  !LEFT JUSTIFY BYTE

OPS(LR,LR,BP),
SHIFT(Z,A,B BYTE LEN),R←SHIFT,  !SHIFT BYTE INTO LR
RESULT(ALWAYS);                 !ALWAYS DELIVER A RESULT

OPS(BL,Z,BP),
SHIFT(Z,B,MC),MC SHIFT=27,      !ALIGN BYTE LENGTH AS QW3
FA(Z,B,Z),MERGE QW3=1;          !CLEAR QW0, QW1, QW2

OPS(T1,BL,BP),TAKE B,           !BA ← BYTE POINTER ADDRESS
CSA(A*2,Z,B),FA(B CO,S,A+B);    !T1 ← POS+2*LEN

OPS(T2,Z,T1),
FA(MC,B,A-B),MC=36;             !T2 ← 36-POS+2*LEN

OPS(NBP,BL,BP),
FA(A,B,A+B),                    !NBP ← POS+LEN
BR(SW N,BYTE POS OVFL);         !BR IF POS+2*LEN>36

OPS(NBP,NBP,BP),
SHIFT(A,Z,Z),FA(Z,B,B),MERGE QW3=1,  !MERGE POS+LEN INTO BP QW3
RESULT(ALWAYS);                 !AND DELIVER NEW BYTE PTR

OPS(IW,BA,Z),
FA(A,B,A),                      !PASS BACK ADDRESS UNCHANGED
DONE(ALWAYS),RESULT(ALWAYS);

BYTE POS OVFL:                  !BYTE POSITION OVERFLOW

OPS(NBP,Z,BP),
SHIFT(Z,Z,Z),FA(Z,B,B),MERGE QW3=1,  !MERGE 0 INTO BYTE POS QW
RESULT(ALWAYS);                 !DELIVER NEW BYTE PTR

OPS(T3,BA,Z),
SHIFT(A,A,MC),MC SHIFT=6,R←SHIFT;  !ROTATE BA TO LEFT JUSTIFY

OPS(NBA,Z,T3),
FA(MC,B,A+B),MC=256;            !ADD 4 TO BA

OPS(IW,NBA,NBA),
SHIFT(A,B,MC),MC SHIFT=30,R←SHIFT,  !ROTATE NEW ADDRESS
DONE(SW V L),RESULT(ALWAYS);    !PASS ADDRESS. DONE IFF NOT OVFL

OPS(IW,Z,Z),
```

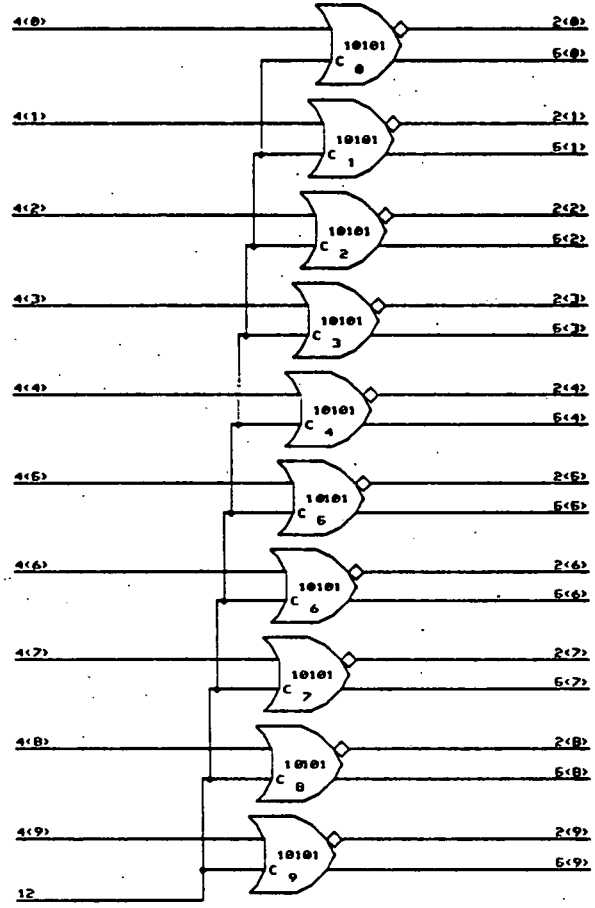
```
FA(MC,B,A),MC=BYTE PTR TRAP,
DONE(ALWAYS),TRAP;                          !ADDRESS OVERFLOW.  TRAP.
```

```
MSBIT:

%D      [IA]    !DATA
%BP     [IB]    !BYTE LEN,BYTE POS

OPS(4,D,BP),TAKE,
SHIFT(A,Z,BYTE POS),R←SHIFT;                    !LEFT JUSTIFY BYTE IN DATA WORD

OPS(5,4,Z),
FA(A,B,A+0),                                    !SET UP TO TEST BYTE SIGN
MM SEL=IZC;                                     !R5 ← IZC

OPS(IW,5,Z),
FA(A,B,A+1),                                    !INCREMENT IZC
DONE(SW N L),RESULT(DONE);                      !IF BYTE≥0 THEN DELIVER IZC+1

OPS(IW,Z,Z),                                    !BYTE<0
FA(Z,B,A),
DONE(ALWAYS),RESULT(ALWAYS);                    !DELIVER 0 RESULT
```

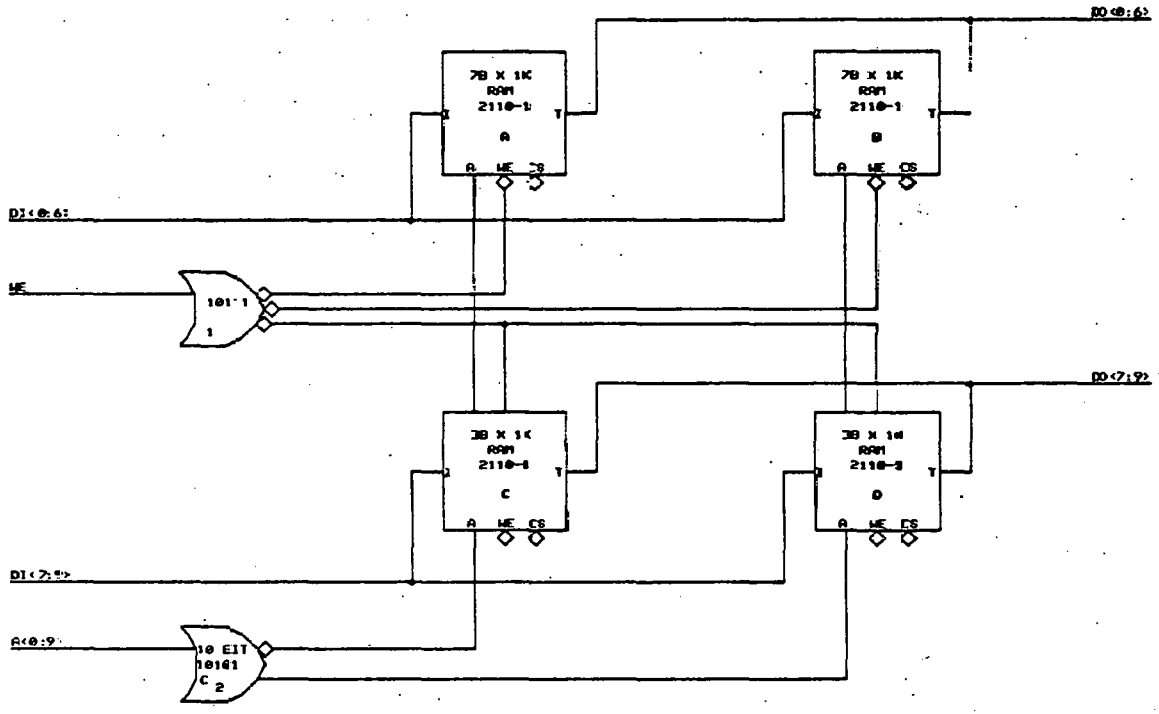
A12. Low-Level Macro Drawings



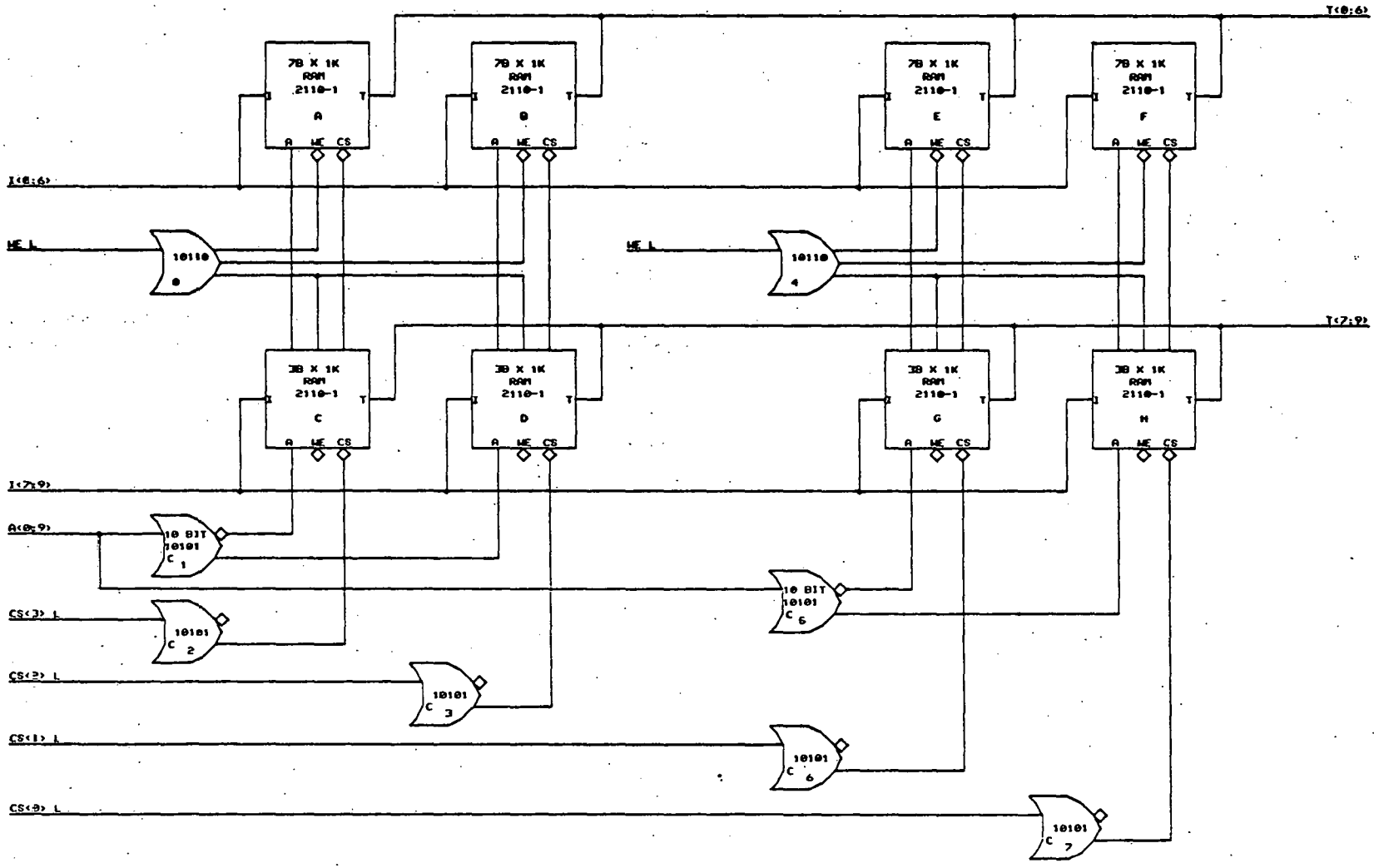
1B X 1K RAM 2110-1 (1KRAM)



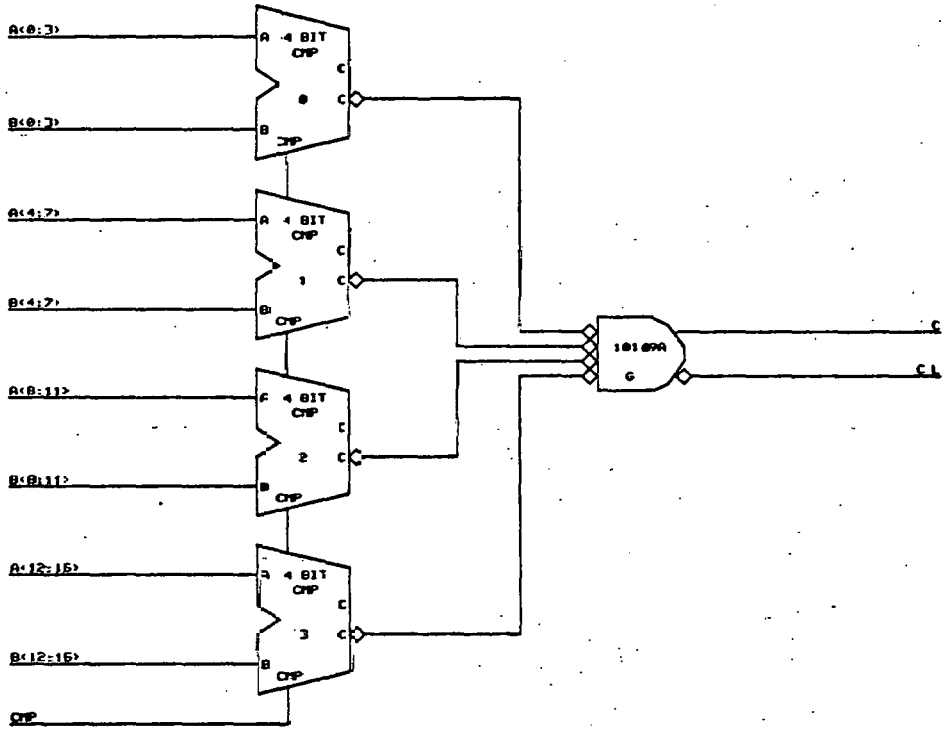
10 Bit 10101 (MAC101)



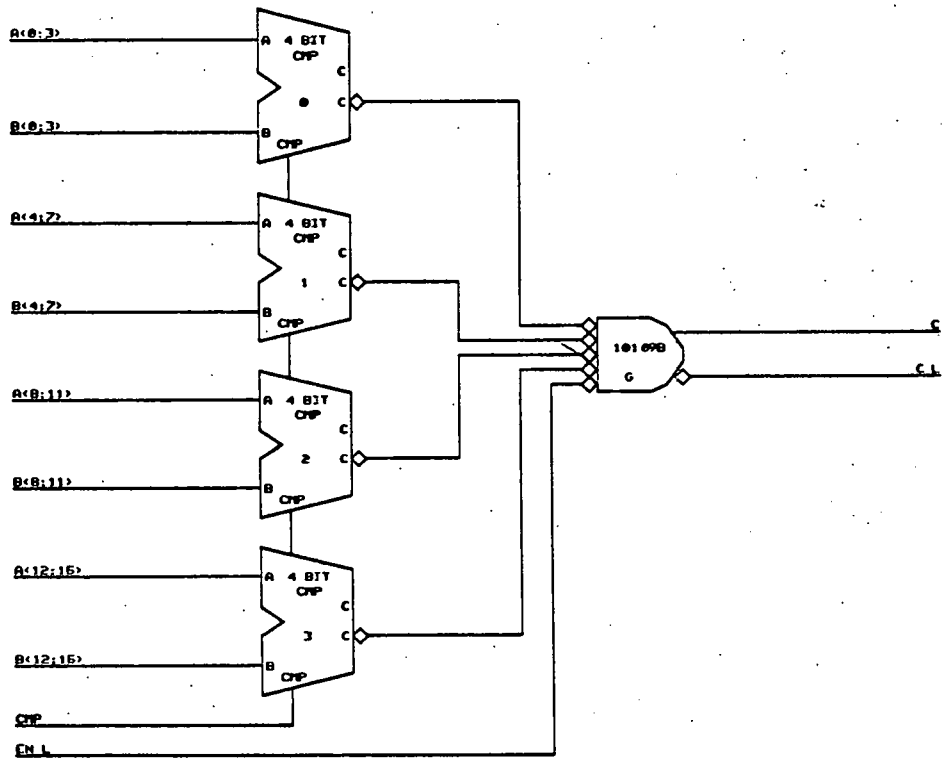
10B X 1K EBOX Control Store Cell (ESCEL2)



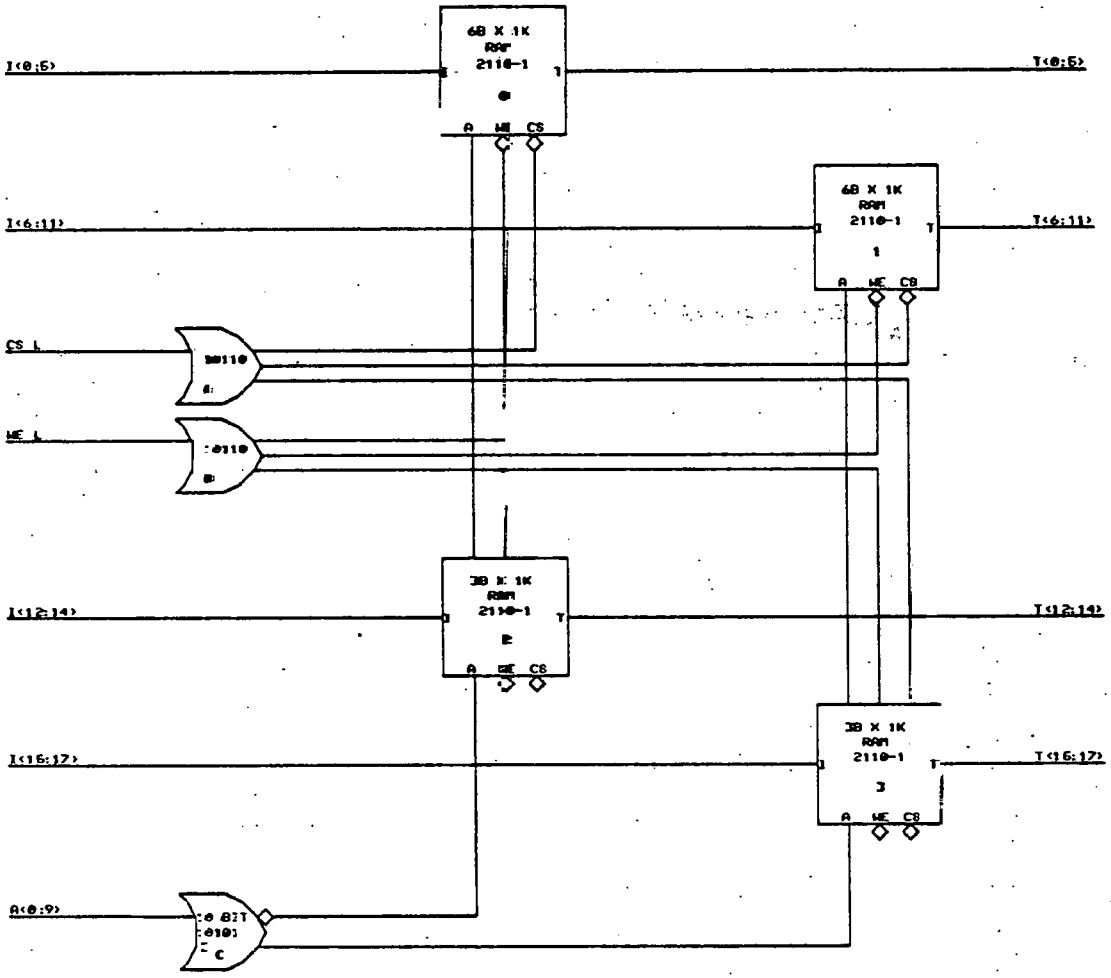
10B X 4K RAM 2110-1 (10BX4K)



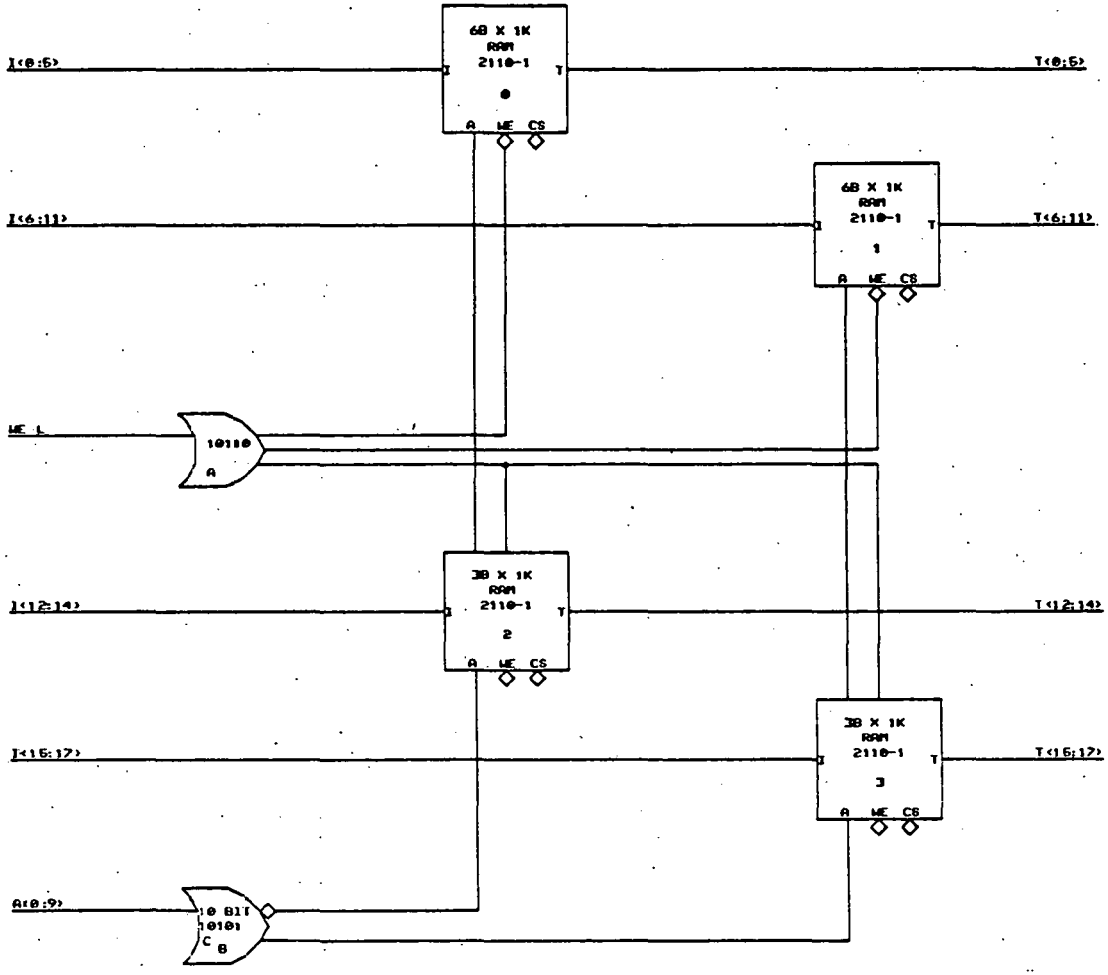
16 Bit CMP (16CMP)



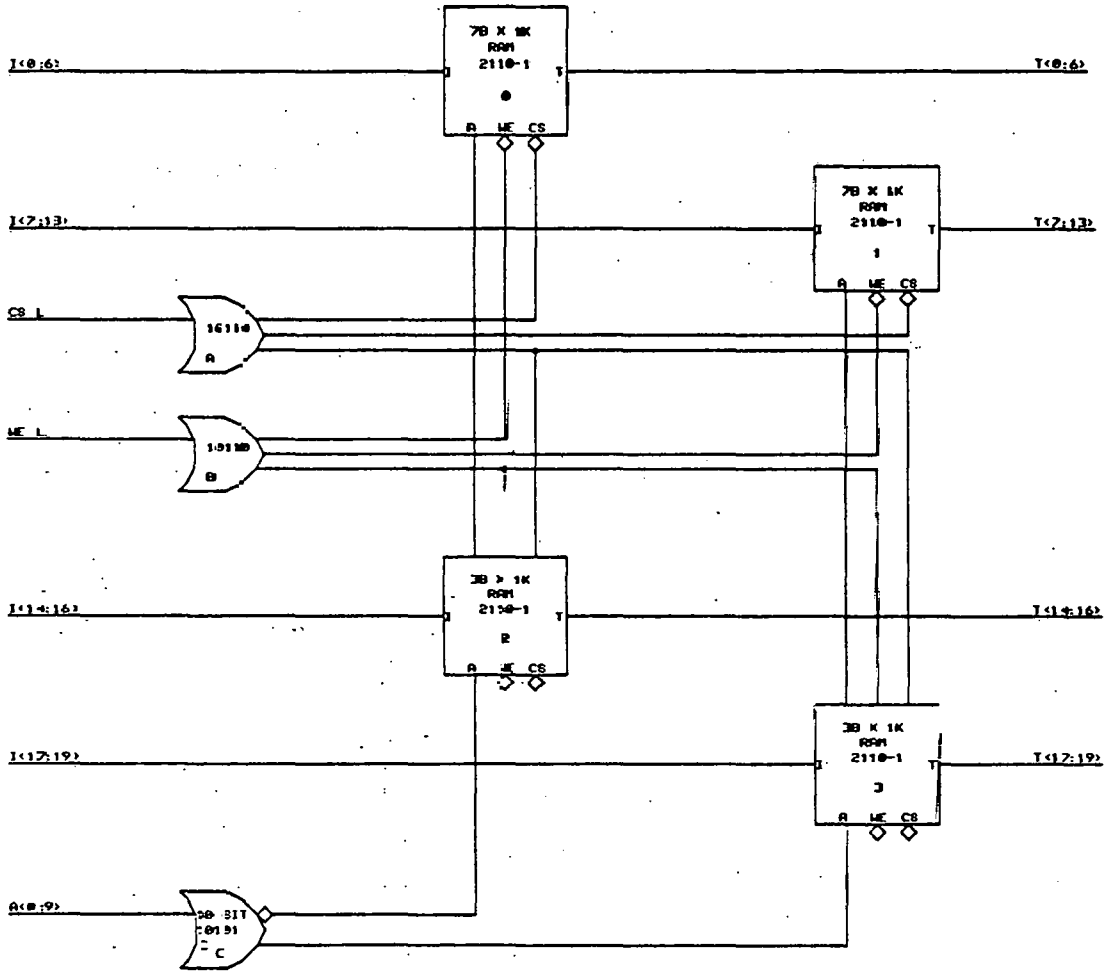
16 Bit CMPEN (16CMPE)



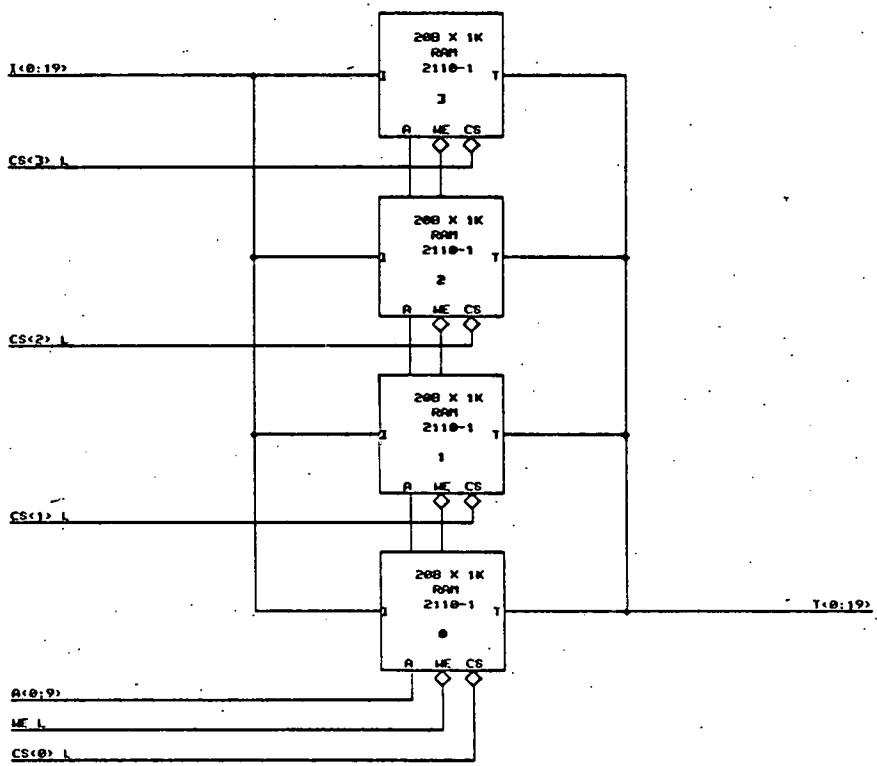
18B X 1K RAM 2110-1 (18BX1K)



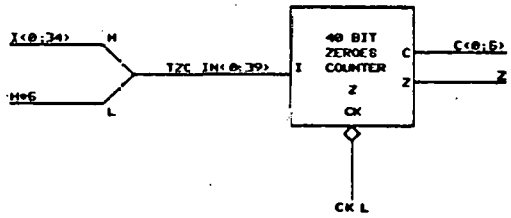
18B X 1K RAM 2110-1 W/O CS (18B1KW)



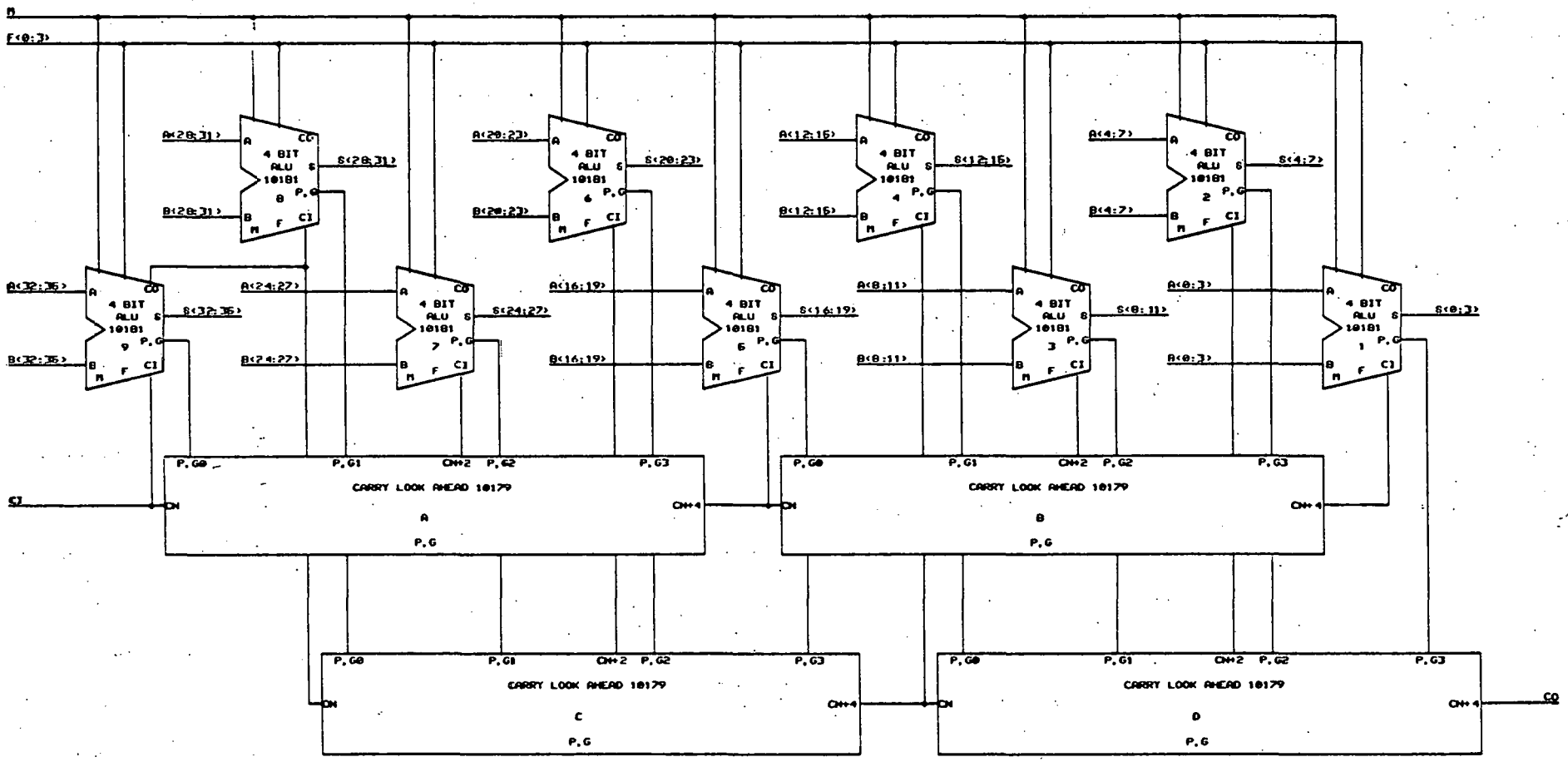
20B X 1K RAM 2110-1 (20BX1K)



20B X 4K RAM 2110-1 (20BX4K)

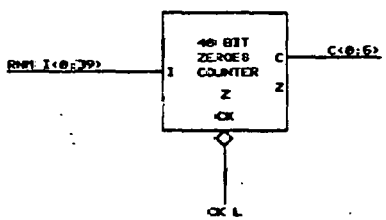


35 Bit Top Zeroes Counter (TZC)

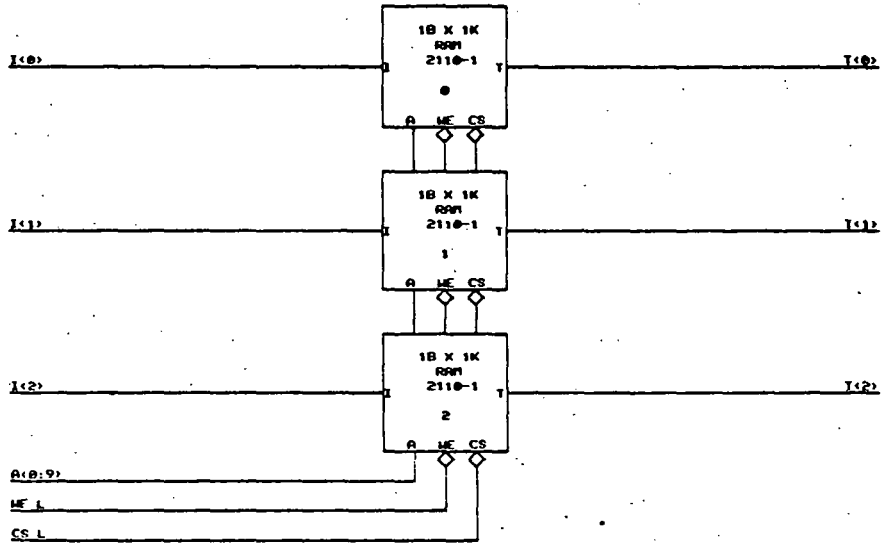


36 Bit ALU 10181 (36ALU)

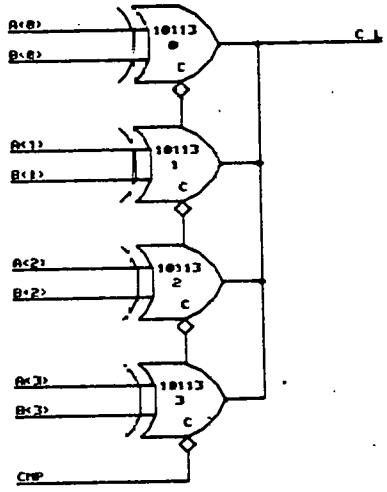
I<35>	RAM I<0>
I<34>	RAM I<1>
I<33>	RAM I<2>
I<32>	RAM I<3>
I<31>	RAM I<4>
I<30>	RAM I<5>
I<29>	RAM I<6>
I<28>	RAM I<7>
I<27>	RAM I<8>
I<26>	RAM I<9>
I<25>	RAM I<10>
I<24>	RAM I<11>
I<23>	RAM I<12>
I<22>	RAM I<13>
I<21>	RAM I<14>
I<20>	RAM I<15>
I<19>	RAM I<16>
I<18>	RAM I<17>
I<17>	RAM I<18>
I<16>	RAM I<19>
I<15>	RAM I<20>
I<14>	RAM I<21>
I<13>	RAM I<22>
I<12>	RAM I<23>
I<11>	RAM I<24>
I<10>	RAM I<25>
I<9>	RAM I<26>
I<8>	RAM I<27>
I<7>	RAM I<28>
I<6>	RAM I<29>
I<5>	RAM I<30>
I<4>	RAM I<31>
I<3>	RAM I<32>
I<2>	RAM I<33>
I<1>	RAM I<34>
I<0>	RAM I<35>
H	RAM I<36>
H	RAM I<37>
H	RAM I<38>
H	RAM I<39>



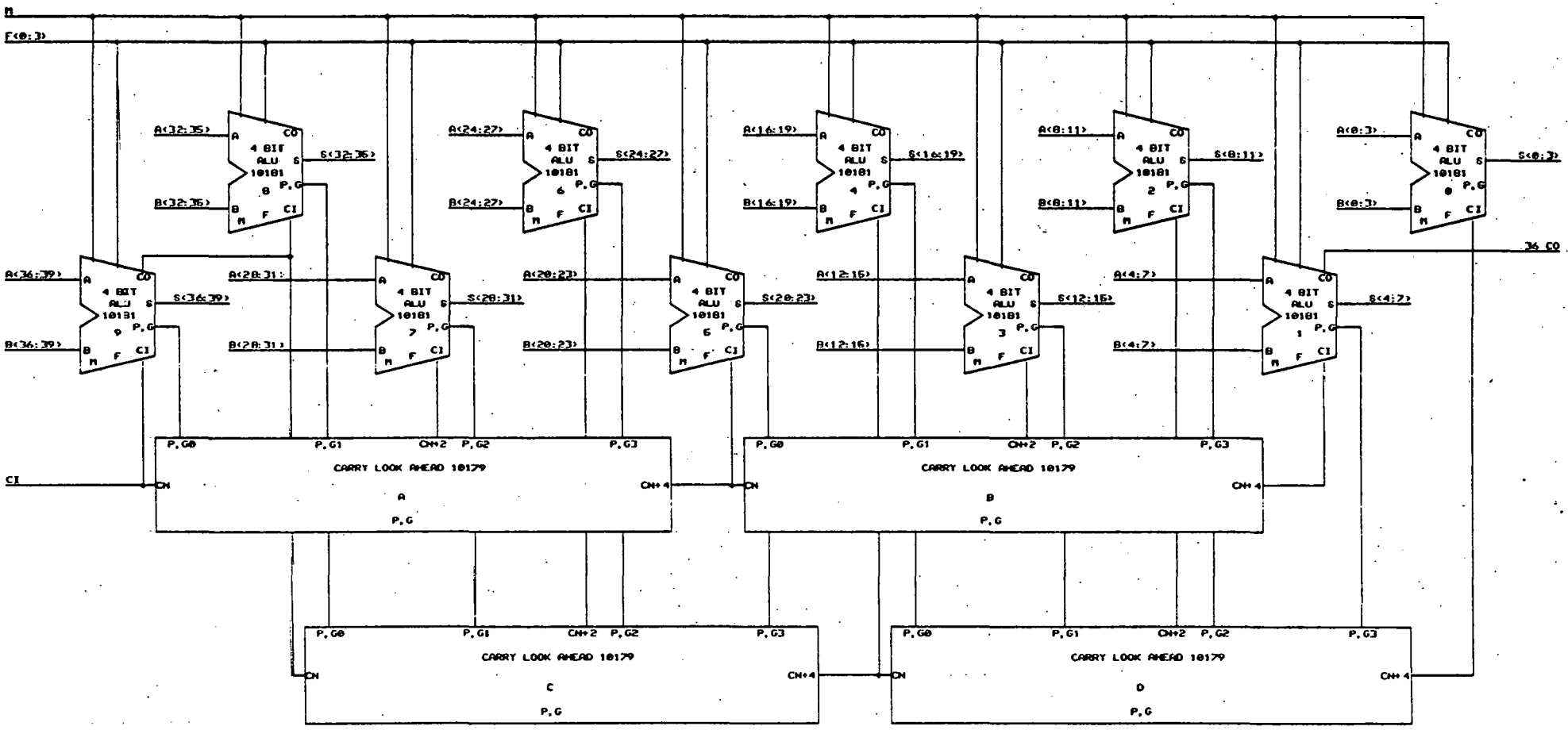
36 Bit Bottom Zeroes Counter (BZC)



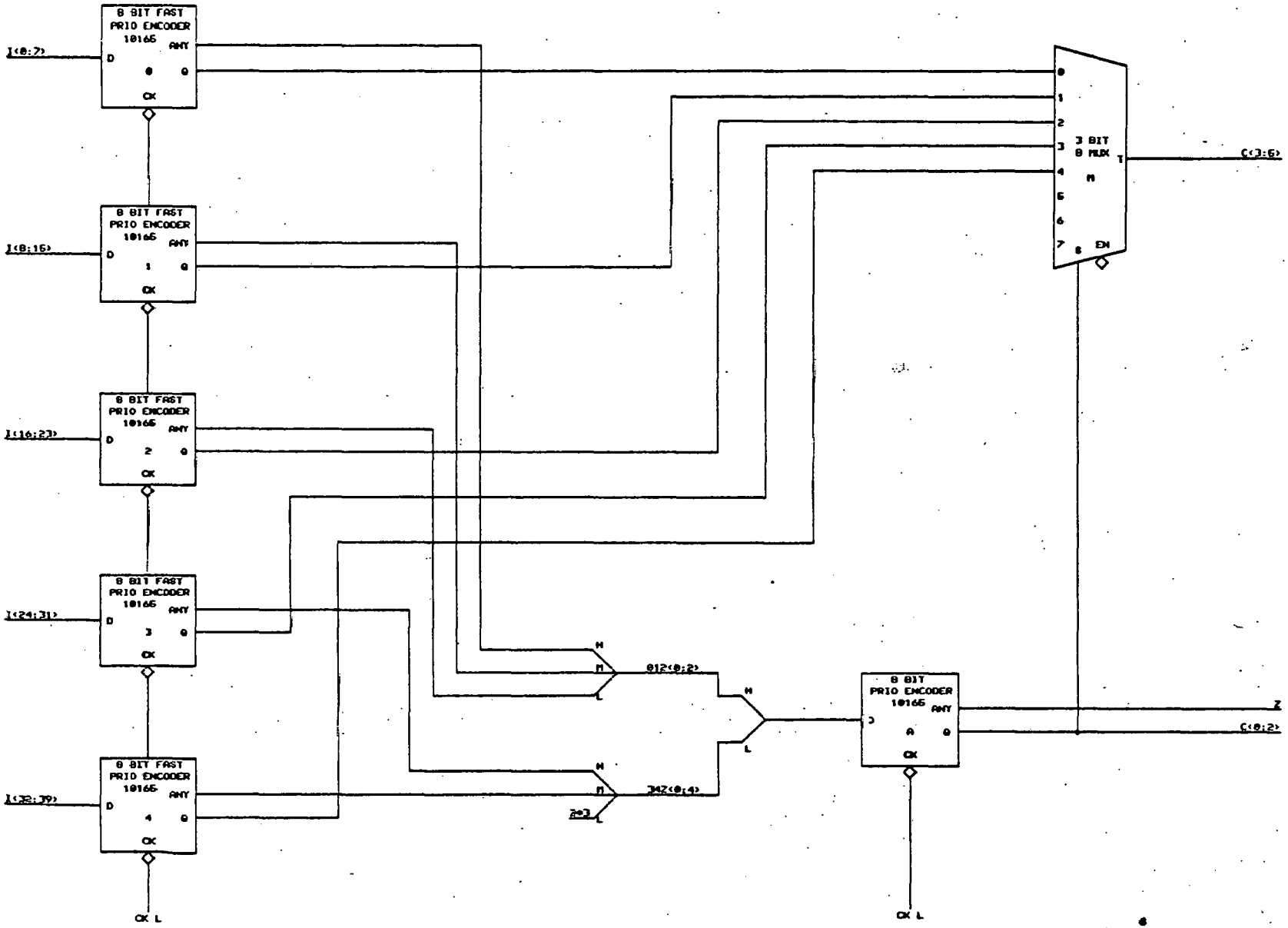
3B X 1K RAM 2110-1 (3BX1K)



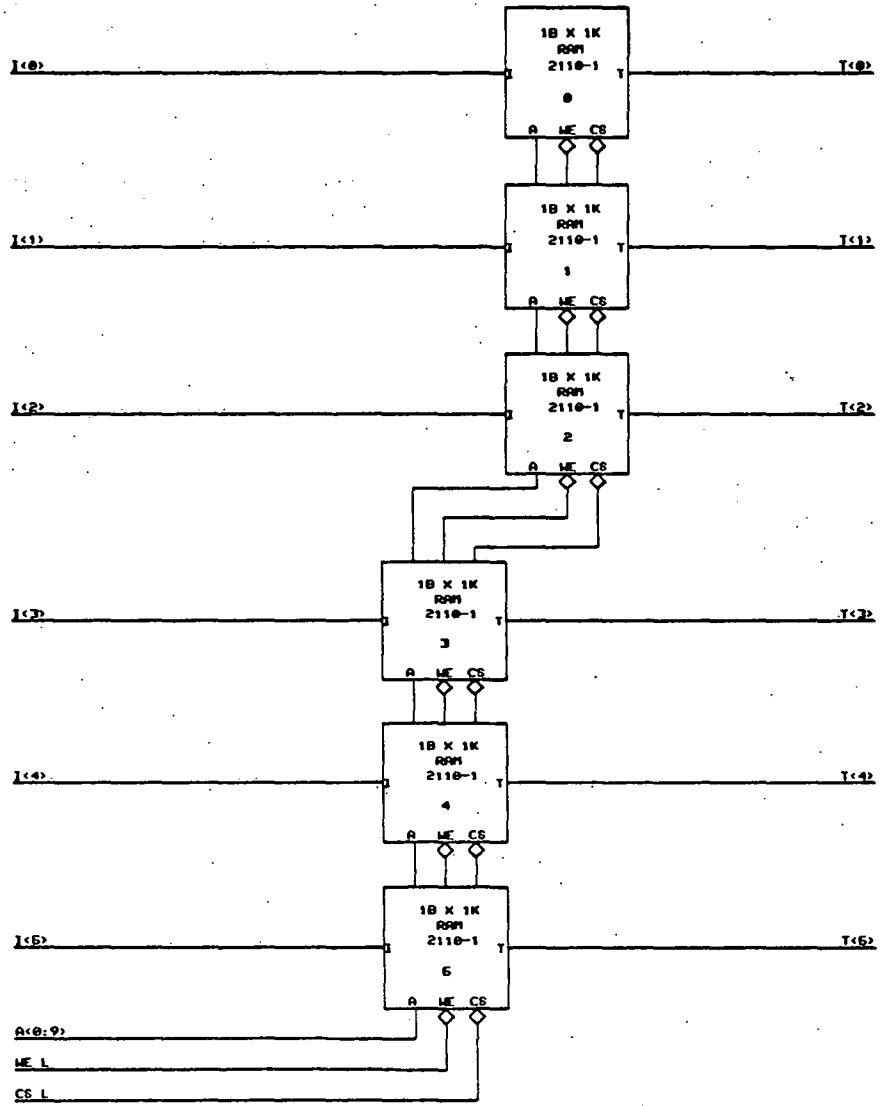
4 Bit CMP (4CMP)



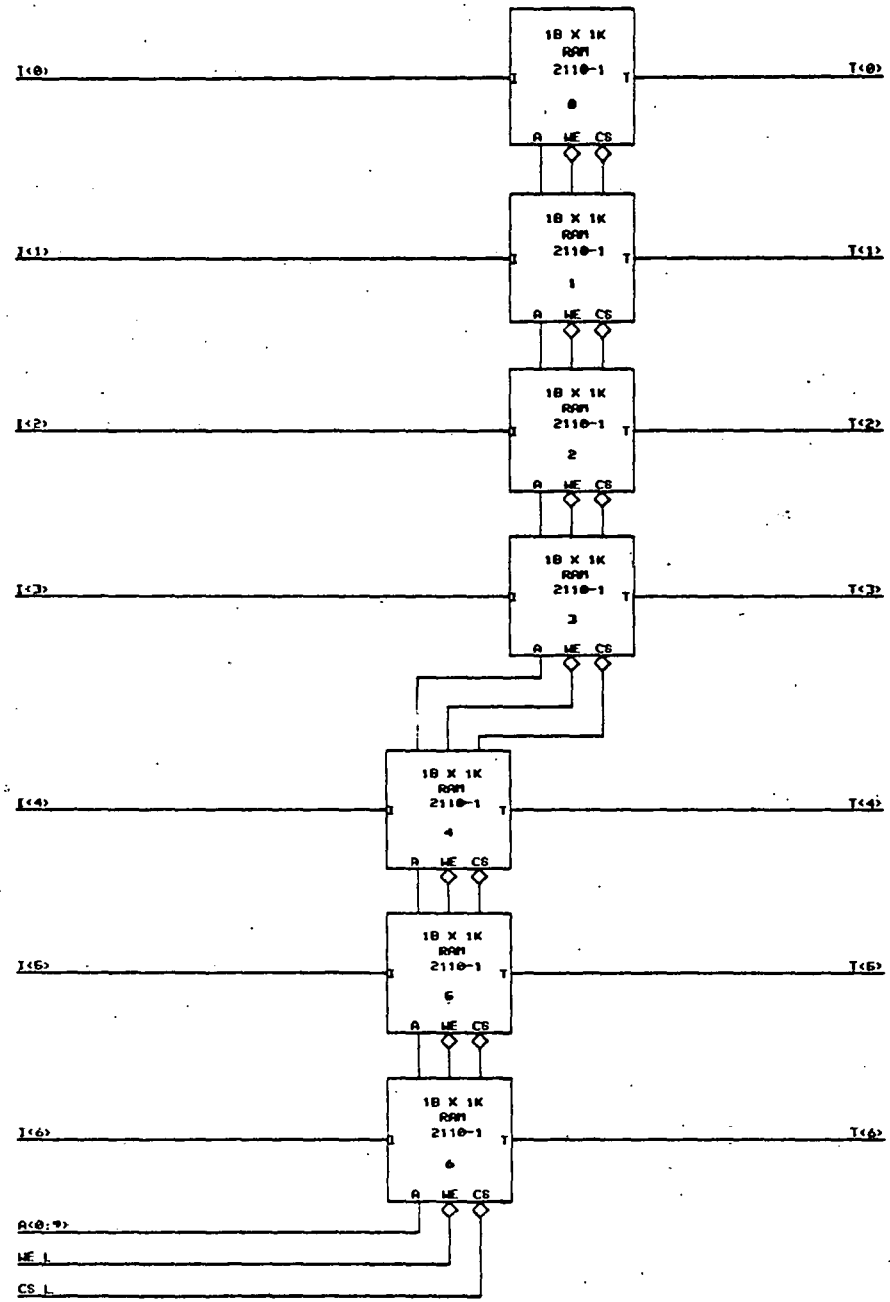
40 Bit ALU 10181 (40ALU)



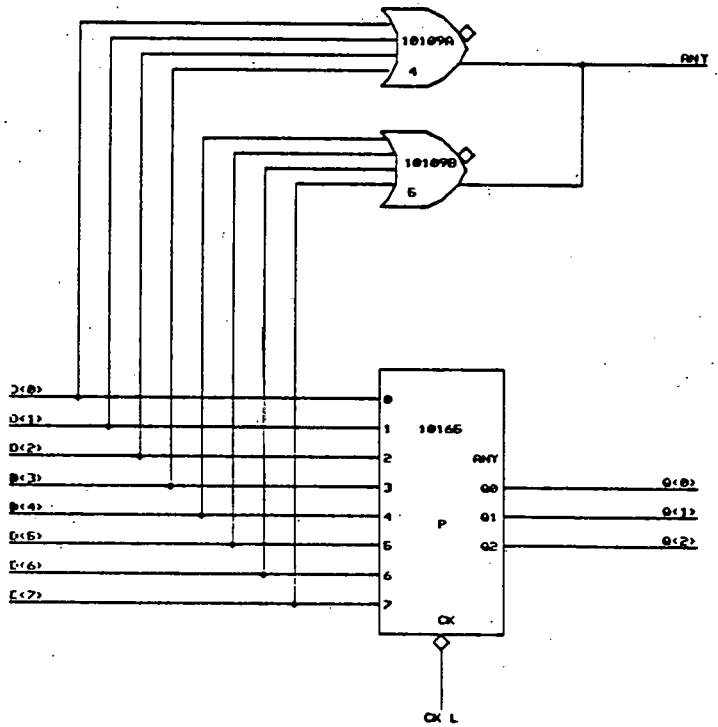
40 Bit Zeroes Counter (ZC)



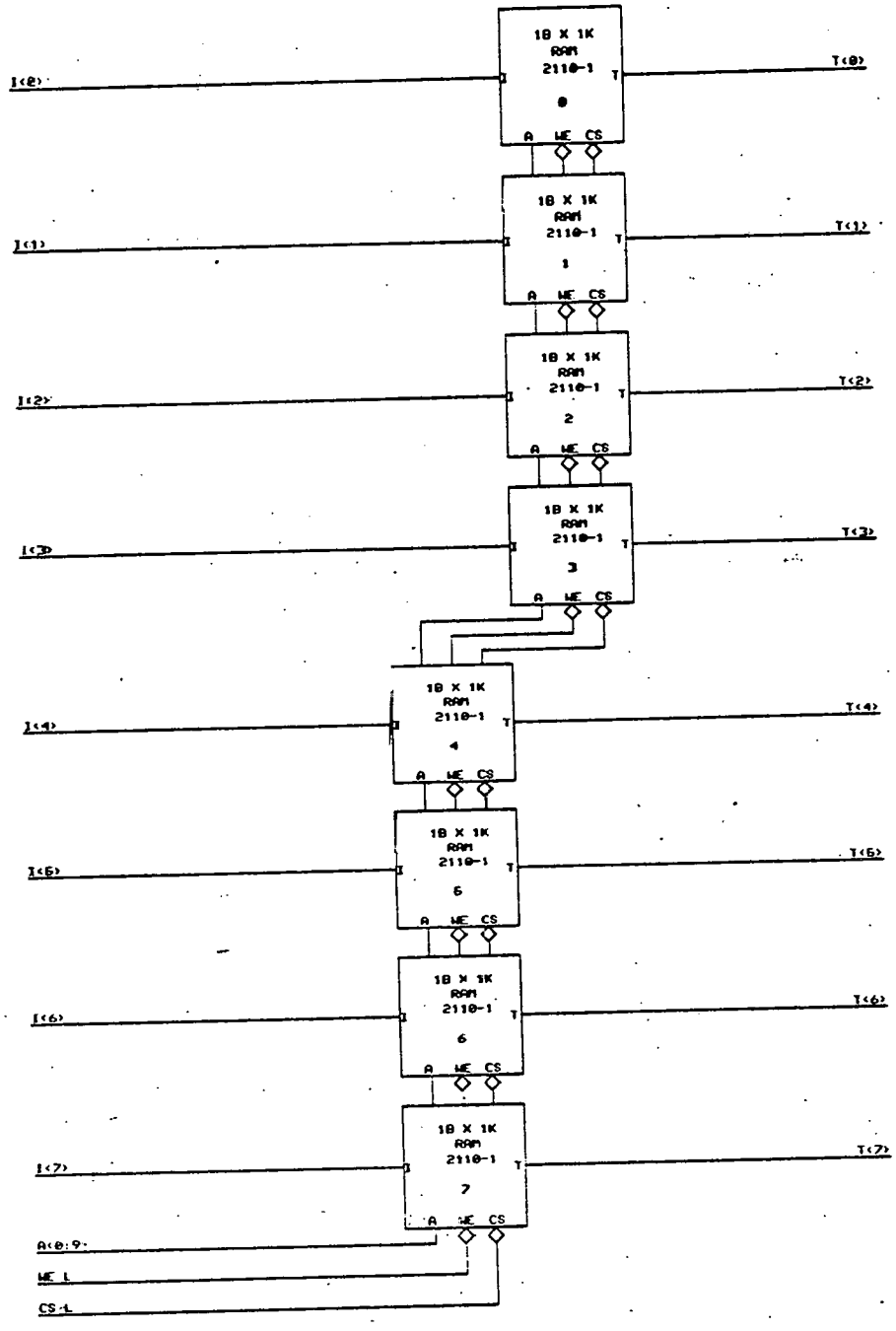
6B X 1K RAM 2110-1 (6BX1K)



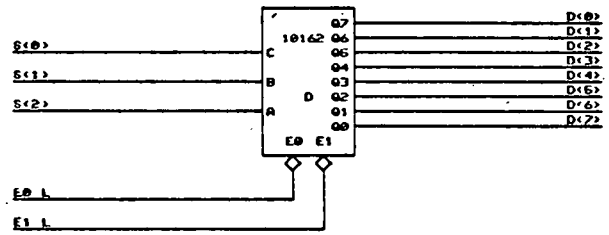
7B X 1K RAM 2110-1 (7BX1K)



8 Bit Fast Prio Encoder (FPRIO)



8B X 1K RAM 2110-1 (8BX1K)



M10162 (M10162)

DISTRIBUTION

O Group Files (5) L-75

TID (15)
TIC (27)

The Naval Systems Division
Office of Naval Research
Arlington, Virginia (5)

Technical Information Department
LAWRENCE LIVERMORE LABORATORY
University of California | Livermore, California | 94550