DESIGN AND IMPLEMENTATION OF A CMOS CHIP FOR PROLOG

Vason P. Srini, Jerric V. Tam, Tam M. Nguyen,
Bruce K. Holmer, Yale N. Patt, and Alvin M. Despain

Computer Science Division
University of California, Berkeley, CA 94720

ABSTRACT

We have designed and fabricated a high performance VLSI chip for executing Prolog programs using a 1.4 micron CMOS technology with two layers of metal. This chip implements a tagged architecture with hardware support for five stacks. The 32-bit data path of the chip contains a fast ALU, 64 registers in four groups, five counters, and six non-master/slave registers. The control is microprogrammed and uses a 512 X 160 bit ROM with four pages for fast microbranching. The chip operates at a cycle time of 100 ns (worst case) and has a size of 10 mm X 9 mm. A semicustom design methodology employing Mentor and NCR tools has been used in this design. The challenges involved in the design, verification, routing, and fabrication of the chip are described.

Table of contents

List of Figures

1. INTRODUCTION

The current trend in computer architecture is to develop high performance architectures that execute programs used in artificial intelligence (AI) in general and expert systems in particular. Lisp and Prolog are two of the major languages used by the AI community. The Aquarius project at Berkeley has been addressing the problem of designing and building high performance processors for Prolog since 1983. Three Prolog systems have been designed and two of them have been constructed. The three Prolog systems are coprocessors to a host such as the NCR 9300 system or the SUN workstation. They are different implementations of the Warren Abstract Machine [10, 11]. The Warren Abstract Machine involves translating Prolog programs to an intermediate language, called W-code, and from there to the machine language of the host processor. Machine instructions are then interpreted by the host microcode to control the data path of the host microengine.

Fast execution of Prolog programs requires architectural support for procedure calls, unification, and backtracking. Although hardware support for procedure calls is available in many of the commercially available 32-bit microprocessors, not much was known about supporting backtracking and unification in hardware until 1983. Since unification requires knowledge about the data types of the terms being unified, a tagged architecture is almost a necessity. Otherwise, the execution of Prolog programs is slowed down significantly. A tagged architecture for Prolog has been developed at Berkeley based on the Warren Abstract Machine (WAM) [10, 11]. The first system developed at Berkeley translates the instructions of the abstract machine [11], called W-code, directly to the microcode of a special purpose processor designed for interpreting W-code. The architecture for the special purpose processor, called Prolog Machine (PLM), has been designed by Dobry [4]. The PLM includes only those features that are deemed necessary by the the results of extensive simulation. The PLM has been constructed using TTL parts and runs at a cycle time of 100 ns. The PLM is connected to a host processor, an NCR 9300 system, to do I/O, floating point calculations, and diagnostics. The performance of PLM for benchmark programs and comparison to other systems have been described by Dobry [5].

The second system generates vertical microcode for a general purpose processor, the NCR 9300 system, from W-code [7]. Three significant pieces of software are used in transforming Prolog programs to executable NCR/32-000 microcode: a Prolog compiler, a microcode compiler, and an assembler. The

study showed the importance of tags and special purpose architecture and compared the performance of NCR/32 [1], with that of the TTL implementation.

The third implementation is a VLSI design of PLM. This report describes the design of a 32-bit microprocessor which combines the architectural features of the PLM with a static CMOS implementation to create a processor with high regularity, low power dissipation, and a small instruction set. The design has been implemented using a semicustom methodology with standard cells supplied by NCR corporation. Macrocells such as ALU, register file, counter, and ROM have been designed and used in the processor. The chip is constructed using a 1.4 micron feature size CMOS process with two levels of metal for interconnection. The chip requires 120 pins and has a size of 10 mm X 9 mm. It is housed in a pin grid array package with 192 pins and dissipates 2 watts. The chip is designed to be a coprocessor for workstations such as SUN-3 and NCR-Tower. Since Prolog programs are memory intensive, a cache is assumed between the chip and the host processor with a read time of 100 ns for cache hits. The availability of a cache allowed us to move the maintenance control unit and the instruction prefetch unit off chip. The chip is designed so that it can be interfaced to standard buses such as VME and MULTIBUS-II. A block diagram of a system using the chip is shown in *Figure 1*.

The chip design and simulation have been complex because of the nature of Prolog, the 100 ns cycle time, and its large size. The need to support backtracking and unification in hardware, and the use of cdr-coding for lists and structures have contributed to the complexity of the architecture. Although the PLM architecture has a small number of instructions (< 60), many (> 16) of which require several (> 9) cycles to execute. The data path has to support six simultaneous register transfers and communicates address and data to memory in a single microcycle. This requires a minimum of 8 buses. We had to tradeoff space to achieve the 100 ns cycle time. For example, instead of using 8 buses that would run the length of the chip to support the register transfers, we used 11 buses of which three run the entire length of the chip. This approach reduced the bus capacitance without taking extra area for buses. The complexity of the input and output parts of blocks is also reduced. It also increased the number of possible parallel transfers within a single microinstruction. The price we paid for this is 8 extra bits in the microinstruction. The second layer metal in the CMOS process is used for the buses, VDD, and GND. The details of the PLM architecture are shown in Section 2.

Achieving the 100 ns cycle time presented many challenges. For example, the critical path in the data path contains a register file, an ALU, and a register. To achieve the 100 ns cycle time, a 16 word register file with a read time of 30 ns and a 28-bit ALU with an add/subtract time of 40 ns are needed. The microsequencer's critical path contains a ROM and next microaddress calculation circuit. This requires a ROM (512 X 160) with a read access time of 40 ns. The next microaddress must be generated in 42 ns based on the status information supplied by the data path. Furthermore, the logic in more than 300 LSI and MSI chips occupying two hex size boards in the TTL version of PLM have to be put in a single chip. The top level view of the chip is shown in Figure 2. It contains the major units of the chip and the logical connections between the units. The design and implementation of the data path is described in Section 3. Section 4 describes the microsequencer. The design of the ROM, microinstruction register, and the generation of interface signals are discussed in Section 5.

The major challenges came in the verification of the design and routing the chip. The design verification process is described in Section 6. A hierarchical methodology is employed by NCR in routing the chip. The size of the design (> 20,000 gates) presented significant challenges in routing the chip. It is described in Section 7.

To simplify the debugging process, static circuits are used everywhere except the ROM. The ROM precharges output lines to achieve the 40 ns access time. The buses used in the chip are also static. A number of features have been added to support testing. For example, the microinstruction register (MIR) and the registers containing the machine status are LSSD registers. Some generic microinstructions have been added to read the contents of registers and to set values to them. Seven pins have been added to the chip for testing.

The chip uses a two phase nonoverlapping clock. The clock skew is controlled by running the two clock phases through the length of the chip using 20 micron wide metal lines and distributing the phases locally in each of the blocks. Overall, the semicustom design methodology allowed us to achieve the desired performance by redesigning the macrocells in some of the blocks without affecting others. The design time has also been reduced because of the use of standard cells. However, changes have to be made to the commercially available tools before they could be used in the design of the chip. This caused significant delays in simulation and physical design.

2. MICROARCHITECTURE

The microarchitecture of the chip contains hardware for manipulating five stacks. These stacks form the data space of a Prolog process. The code is kept in a separate area. The separation of code and data is intended for the efficient management of memory, changing clauses in the code space, and fast access to data bases. The organization of the data space and the pointers to manipulate the stacks are shown in *Figure 3*. The **control stack** (hereafter called "the stack") is the area in memory used for storing control information. Two kinds of objects may appear on the stack: *environments*, and *choice points*. An environment represents the saved state of a Prolog clause: it contains pertinent register values, and what are known as "permanent" variables. Permanent variables are variables needed by more than one goal in the body of a clause; they must be saved so that succeeding goals can access them.

A choice point is a group of data words containing sufficient information to restore the state of a computation if a goal fails, and to indicate the next clause to try. Choice points are placed on the stack by special instructions when entering a procedure containing multiple clauses that can unify with the current goal. Choice points support backtracking, a feature unique to Prolog. Choice points contain the pointers shown in *Figure 3*, state registers, argument registers, and continuation pointer. The choice point frame and the environment frame are shown in Appendix 1.

The **heap** is the area of data memory used for the storage of lists and structures, which are too cumbersome to be kept in environments on the control stack. It is allocated incrementally like a stack, and deallocated in variable size blocks.

The **trail** is an area used for keeping track of variable bindings. When a variable becomes bound during the course of a Prolog program, it may become necessary to undo the binding when backtracking is done. Thus some method is needed for keeping track of all bindings that are to be undone when the current goal fails so that the variables they refer to can be unbound again.

The **PDL** is a small stack created for the unification of nested structures and nested lists. The **H2space** is an area of memory used for global variables, system tables such as symbol table, process table, and page table. It will be used extensively in the execution of concurrent processes resulting from AND parallelism and OR parallelism.

The data and code memories are word-addressable with 28-bit addresses. In a 32-bit word, 28 bits are used for storing data and addresses. The most significant four bits are for tags: 2 bits for data types, 1 bit for cdr-coding, and 1 bit for garbage collection. Since tags are not used in arithmetic operations, the ALU performs 28-bit add/subtract. The data types and their representation are shown in Appendix 1.

The pointers needed to manipulate the stacks are also shown in *Figure 3*. In addition there are argument registers, temporary registers, and 16 registers for storing constants. Many of the stack pointers are actually 28-bit counters. This allows further concurrency in the microarchitecture since increments and decrements need not go through the ALU. A block diagram of the microarchitecture is shown in *Figure 4*.

The instruction set supported by the microarchitecture contains ten classes. They are shown in Table 1. The dynamic frequency count of these instructions for a class of benchmark programs, maximum number of data transfers, and the execution cycles are also included in Table 1 to show their relative importance. Additional details can be found in Dobry [5]. The procedure control instructions create choice points and manage them. The indexing instructions act as filters to prevent the execution of clauses which the compiler can determine will not unify with the invoking goal. The clause control instructions sequence between subgoals in the body of a clause, invoke builtin functions, create an environment, and remove an environment from stack. The get instructions unify the calling subgoal's arguments with the head of an invoked subgoal. The put instructions load the argument registers prior to invoking a subgoal. The unify instructions construct structures on the heap (write mode), and unify the structures (read mode). The arithmetic and logical instructions perform arithmetic operations on 28-bit numbers and logical operations on 32-bit numbers. The jump instructions look at the state of equal and less than flags and jumps to a specified location. The load and store instructions read from memory and store in memory. The miscellaneous instructions is the last group and allows booting the system, reseting the stack pointers, and halting the processor by looping on a microstate.

In addition to the above instructions, there are six fundamental operations (primitives) to support Prolog. They are fail, trail, dereference, decdr, bind, and unify. They are not available to the programmer. The fail operation restores the machine state when a failure occurs during unification. The trail operation manages the trail stack during binding if a variable is to be trailed. The dereference operation follows the chain of pointers which occur due to binding of variables to other variables during unification. The decdr

operation supports cdr-coding of lists and structures. It is used to fetch the next element from a list or structure. The bind operation stores the data value at a given address. It may call the trail operation to see if the binding must be trailed. The unify operation unifies arbitrary Prolog items, binding variables as required. It uses the PDL during the unification of nested lists and structures.

A simulator has been written in C for the microarchitecture. The simulator accepts W-code and produces the state of the architecture for each cycle. The state information includes the contents registers MDR, R, T, T1, MAR, S, N, and the condition codes. The simulator can also produce the stimulants for the QUICKSIM simulator of Mentor's IDEA system for the logical simulation of the chip. The structure of the simulator is described in Appendix 1.

3. DATA PATH DESIGN AND IMPLEMENTATION

A hierarchical design methodology employing semicustom tools was chosen since we wanted to do a single chip implementation of the microarchitecture in the shortest time possible. The availability of 1.4 micron CMOS technology with two layers of metal and commercial design tools (Mentor Graphics' IDEA Station), and the support from NCR for routing and fabricating the chip were the additional factors in our choice of the methodology. The logic level design and simulation, timing simulation, and architecture development took place in Berkeley. We used NETED and SYMED programs of IDEA station to do the schematic capture. NCR-Fort Collins designed the cells and the macrocells for the chip, routed the chip and fabricated it. The details of this physical design are described in Section 7.

The design of the chip involved three major parts: data path, microsequencer, and the support circuitry. We expected the chip area to be dominated by the data path and so concentrated on making it regular. Since our goal is not just to come up with one processor for Prolog, we also concentrated on designing the data path so that parts of it can be reused in other designs.

The 32-bit wide data path is dominated by registers and counters. This can be seen in its floorplan for a bit-slice, shown in *Figure 5*. Each block in the data path can transfer data to many destinations simultaneously. We use tristate buffers controlled by a microbit for each destination. Each block has a multiplexer to select one of the buses as its source. There is a microbit for each source. The modular design of the blocks simplified checking and routing. Since the bus delays have to be reduced as much as possible to

allow enough time for the activities in a block, considerable effort went into the design of buses.

BUS DESIGN

The total number of buses and the length of each bus are two important criteria in the bus design for the chip. If the bus length is kept to a minimum bus transfer time can be reduced because of reduced bus capacitance. The number of buses affects the chip area in the data path. Reduced number of buses also decreases the routing complexity and decreases the area needed for the buses. The placement of the blocks in the data path and the number of simultaneous register transfers in the microstates determine the number of buses and their length.

We analyzed the register transfers in microstates and the eight buses used in the TTL version of PLM to come up with an initial placement for the blocks in the data path and the number of buses. Two different bus designs have been proposed and evaluated. The first bus design kept the buses used in the TTL design. Three more buses are added to maintain compatibility with the microstates of the TTL design, diagnose the chip, and to support future additions to the microcode while implementing some of the built-in functions. The second proposal contained a total of eight buses. It maintained compatibility with the present microstates of the TTL design but did not maintain compatibility with the buses in the TTL design. We felt that this lack of compatibility could cause microstate compatibility problems in the future when new microinstructions are added. So we have decided on the first design. The placement of the blocks and the eleven buses connecting the blocks are shown in Figure 5. We used a program to detect bus conflicts in each of the microstates during the bus design. This program is a key tool to be used in adding new microstates to the ROM. The second level of metal is used for the buses. Bus routing is simplified by butting the blocks as shown in Figure 5. The top level diagram of the data path showing the connections between the blocks is included in Appendix 1.

The design details of ALU, counters, registers, PDL, and register files are now described.

ALU DESIGN

The data path contains an ALU for doing arithmetic and logical operations. It performs arithmetic on 28-bit numbers and compares on 32-bit quantities. The design objective is to finish the longest operation

(add/subtract) in the least possible time using the smallest area possible. The clocking scheme allows 40 ns after phase 0 goes low for the ALU to complete add/subtract operation. This timing constraint requires the use of some parallel carry generation scheme. We use the P-circuit [9] with pre-conditioning and post-conditioning [2, 3] circuits to generate the carry.

The implementation of the ALU using standard cells and a semicustom design methodology imposes some constraints on the designer. For example, the fan-in, in addition to fan-out of the basic cells should not be large on the critical path; otherwise, the delay in the circuit would be large.

The ALU comprises four blocks: an input block transforming inputs according to the control signals, a compare block testing whether the inputs are equal, a parallel prefix calculation block generating the propagate (P) and generate (G) signals needed for carry calculation, and a sum block supplying the final result. The top level view of the ALU is shown in Figure 6. The functions of the ALU are similar to that of the AS181 chip, but it uses a fast carry evaluation method to achieve high performance.

In the input block, the P and G signals for each bit are generated according to the control signal and inputs. The compare block passes the inputs through exclusive OR gates and then tests to see if all the outputs are zero. The testing is done using a tree of NAND and NOR gates and is performed in parallel with other ALU operations. We could have put the testing circuitry after the sum block, but our approach removed the testing from the critical path. However, the alternative method would allow us to test for zero output which might be generated from logic or increment/decrement operations. This testing cannot be done by our approach. There is no loss of functionality since the TTL version does not test for zero output.

Considerable amount of effort went into the design of the parallel prefix calculation block. The basic architecture is derived from the works of Ladner and Fischer [9]. The pre-conditioning and post-conditioning circuits invented by Despain [2] are incorporated to reduce the fan-outs in the design of the block.

NCR standard cells are used to implement the ALU and some new cells are added to optimize the critical path. At each iteration, NCR software tools are used to identify the potential critical paths, and then intensive SPICE simulation runs are used to obtain better estimates on the delay. We have designed an ALU that performs add/subtract with a worst case (VDD = 4.5V, Temperature = 80 C) delay of 37 ns.

Since the PLM uses the four most significant bits for tags and cdr coding, the ALU needs to separate them from the real data if overflow detection is to be done. So, the ALU performs arithmetic operations on the least significant 28 bits, while the the logical and comparison operations are done on 32-bit inputs. However, there is still anomaly over the secondary tagging. Currently, the ALU tests for overflow from bit 28, while the secondary tagging takes place on bit 27 and bit 28. No small integer (26-bit integer) overflow is caught since the additional logic needed to check the data type will slow down the ALU.

Since only 28 bits, instead of 32 bits are needed in the carry calculation, further optimization based on the Q-circuit ideas proposed by Despain [3] is implemented. The general idea is to use circuits with fewer gate delay levels to drive those with more levels, and thus try to absorb the propagation delay which is both unavoidable and significant in MOS circuitry. The control to the ALU consists of 4 bits, S3, S2, S1, and S0. In addition, a mode bit M controls whether an arithmetic or a logical operation is performed. A carry in bit (Cn) from the microinstruction controls whether the initial carry is zero. The functions implemented by the ALU are shown in Table 2.

Note that there are many unused entries in the table and it is possible to do further optimization.

COUNTER DESIGN

The data path contains five counter blocks: H, H2, T, S, and MAR. These counters are important for high performance. Since data space and code space addresses are 28-bit wide in our architecture, the counter blocks do only 28-bit counts. However, they can store 32-bit values. The counters are used for pushing and popping the stacks in the data space. Each counter block contains a 32-bit master/slave register and counting logic. A carrylookahead scheme is used to achieve a worst case count delay of 60 ns. The 28-bit counter is implemented in three stages. The first stage contains 8-bit carrylookahead circuit. The other two stages contain 10-bit carrylookahead circuits. The carry ripples through the three stages.

Each counter can perform four functions: load in new data from the selected bus, increment by one, decrement by one, and hold current value. The design objective for the counter is to obtain the new counting value within the specified time limit, using the smallest area possible.

Our clocking scheme allows a worst case delay of 60 ns for the counting logic alone (70 ns for counting and transfer). This timing constraint requires the use of some carry lookahead scheme. For

efficient use of area, regularity among the 28 counting bits is desirable. We started our design from the basic counting equations to examine the possible logic circuit organizations. First, we viewed the counter as an adder; that is, incrementing is adding a positive one and decrementing is adding a negative one (in 2's complement). Consider the following basic equations for addition with carry lookahead:

$$G_i = A_i B_i$$
$$P_i = A_i \oplus B_i$$
$$C_{i+1} = G_i + P_i C_i$$
$$S_i = A_i \oplus B_i \oplus C_i$$

where A and B are the values to be added; G is carry *generate* signal; P is the carry *propagate* signal; C is the carry; and S is the resulting sum.

Let A be the data currently stored in the D flip flops (DFF), let B and C_0 (the initial carry in bit) combined to be the value to be added to A. For counting up, assign $B_i = 0$ and $C_0 = 1$ to obtain the effect of adding a positive one. When counting down, set $B_i = 1$ and $C_0 = 0$ to add a negative one. There is an active low control signal UP, which specifies the counting direction. The counter increments when UP is low and decrements when UP is high. By substituting UP for B_i,

$$S_i = A_i \oplus UP \oplus C_i$$
$$C_{i+1} = A_i C_i \overline{UP} + (A_i + C_i) UP$$

Unrolling the recurrences for C_i, we get

$$C_i = (A_{i-1} A_{i-2} \cdots A_0) C_0 \overline{UP} + (A_{i-1} + \cdots + A_0) UP + C_0 UP$$

The equation above describes the logic of the carry circuit for an *n-bit* counter block with *n-bit* carry-lookahead. C_0 is the initial carry into this block and C_n is the carry out of this block. The resulting values are stored in the sum bits S_0 through S_{n-1}. The logic circuit of this n-bit block is made up of 2 different bitslices, each bitslice contains the suming logic, the DFF to store the result, and the generation of carry into the succeeding bit. The carry generation logic for the bitslices are shown below.

Bitslice type 'a', used for bit 0:

$$\text{cnand} = \overline{C_0 * UP}$$

$$\text{andline}_1 = A_0$$

$$\text{orline}_1 = A_0$$

$$C_1 = \text{cnand} * \overline{A_0\ UP} * A_0\ C_0$$

Bitslice type 'b', used for bits i = 1 to n-1:

$$\text{andline}_{i+1} = \text{andline}_i\ A_i$$

$$\text{orline}_{i+1} = \text{orline}_i + A_i$$

$$C_{i+1} = \text{cnand}\ \overline{(\text{andline}_{i+1}\ C_0)}\ \overline{(\text{orline}_{i+1}\ UP)}$$

An *n-bit* block consists of one bitslice 'a' and *n-1* bit slices 'b'. We call each such block a *stage*. The 28-bit counter is made up of three blocks, where the values for *n* are 8, 10, and 10, respectively, with the 8-bit block being the least significant. The carry out of the first stage is connected to the carry in of the 2nd stage, and the carry out of the 2nd stage is in turn tied to the carry in of the 3rd stage. The top level view of the counter is shown in Figure 7.

The critical paths in this circuit are *orline* which ORs all the bits of A (the *andline* is similar but operates a bit faster), and the carry propagation from one stage to the next. The counter is broken up into stages to take advantage of the fact that the *orline* is local to each of the three stages, and since values of A are available immediately, these lines operate in parallel. There is some delay associated with driving the control signal UP through all 28 bits (also, $\overline{UP}$ is the carry into the first stage). These two delays are synchronized so that the control signal will be stable at about the same time as the *orline* into the last bit of the first stage, which together generate the carry out of the first stage. From this point on, we only have to be concerned about the delay of generating the carry out of the 2nd stage into the 3rd stage. In short, the logic is constructed in a way which minimizes the number of gates in the carry path.

With respect to the VLSI methodology, our counter design has a number of advantages. First, the layout is fairly regular with only two different types of bitslices, using 3 of one type and 25 of the other. Second, very simple 2-input and 3-input gates (standard cells) are used, which require much smaller area and switch faster than their higher fan-in equivalents. And finally, routing is significantly simpler because

the number of inputs and outputs of each bitslice is fairly constant, and because most of the inputs come from the outputs of the immediately preceding bitslice.

PDL AND EXCEPTION HANDLING

The Push Down List (PDL) is a LIFO data structure which has 16 locations of temporary storage for the pointers to the Prolog structures. It has two parts, PDL left (PDLl) and PDL right (PDLr). Each part is 32 bits wide. During unification of structures, each location contains a pointer to the next deeper nesting level of the structure. Although we believe that it is highly unlikely that structures in Prolog programs are nested deeper than 16 levels (in our benchmarks, they are nested no more than 10 levels deep), we have designed the architecture to detect and to handle the potential overflow. The PDL address calculation logic (PDLACL) in Figure 5 manages the PDL. The top level diagram of PDL is shown in Figure 8.

The PDLACL has two markers, called TOP and BOTTOM. As their names suggest, TOP and BOTTOM mark the top and the bottom of the PDL, respectively. Both markers are initialized to zero at the start of the structure unification. During normal operation, a PDL *push* increments TOP (modulo 16) before storing data into PDL at TOP, and a *pop* decrements TOP after the data from PDL at TOP has been read. In our scheme, an overflow occurs when TOP and BOTTOM both point to the same place in PDL and there is an attempt to write into it (a *push* operation). When this happens, a hardwired address to the overflow handler routine in the control ROM is selected instead of the normal next microaddress. The overflow handler routine increments the BOTTOM marker, moves *one* location (both PDLl and PDLr) from the PDL at the BOTTOM out into the stack in the data space in memory, and jumps back into the normal unification microcode. Upon exiting the handler routine, BOTTOM now points to one location above TOP. If a push operation is done after an overflow without an intervening pop, another overflow will occur and BOTTOM will again be incremented by one.

After an overflow has occurred, PDL *pop*s will function normally as TOP will be decremented (modulo 16) each time. When TOP is again equal to BOTTOM and a PDL read request is present, PDL underflow signal becomes active and the address to the underflow handler will be selected as the next microaddress. The underflow handler restores *one* location of the PDL (both PDLl and PDLr) from the stack, and decrements BOTTOM pointer.

There is a single bit D flip-flop to remember that a previous overflow has occurred. This bit is set when the first overflow occurs and remains set until all overflow data in stack has been restored into PDL, at which time it will be cleared by the underflow handler.

Since we believe that overflow rarely occurs, the detection and handling mechanisms are designed to require minimal additional hardware and microcode, and such that performance in normal situation would not be affected. In terms of additional hardware, the scheme presented above requires four latches, four 2-input MUXs, one D flip-flop, and about a dozen simple gates used in the comparison and decoding logic for the control signals from MIR. If the PDL does not overflow, all instructions operate at the same number of clock cycles as the TTL version without any exception handling. The detection mechanism is transparent and requires no additional microstates. In the event of an overflow or an underflow, approximately ten extra cycles are required for the exception handler to execute.

REGISTER DESIGN

The data path contains five non-master/slave registers. These registers are used for storing the arguments supplied by an instruction (ARG1 and ARG2_3), memory data register (MDR), result from ALU (R), processor status register (PSW), and scratchpad (T1). Each register contains an input multiplexer (MUX), transparent latches, and output tri-state drivers. The input MUX is used to select input to the register from different buses. The output tri-state drivers are used as multiple read ports of the register. In between, there is the transparent latch which is used as a storage element.

To support the data structures used in the PLM architecture, some registers provide functions that manipulate the most significant 6 bits which include the tag bits and the CDR bit. For example, register T1 is capable of clearing the most significant 6 bits or the most significant 4 bits, which corresponds to providing a short integer or clearing the primary tag and secondary tag bits. Another example is the MDR which provides means to change the CDR bit and tag bits using data from the microinstruction. It is also one of the most complex blocks in the data path. The symbol of MDR is shown in Figure 9. The MDR block also manipulates the tag bits and the cdr bit from other sources. The tag bits of MDR can be loaded from any one of six buses. The cdr bit can be loaded from any one of eight places. The details of MDR in Figure 10 shows the various sources for the cdr bit and tags. The MDR block allows data to be transferred to and

from memory with appropriate tags. For example, the tag of MDR can be set to the tag of T1 register and the cdr bit of MDR can be set to that of T register in one cycle.

All registers are written during phase 1 and read during phase 0. One way to do this is by doing an AND operation on the clock phase and control signal from the microinstruction, and driving the clock inputs of the 32 latches using a huge buffer. This implementation introduces local clock skew because the delay of driving 32 clock inputs is quite large. The second way reduces the clock skew and it is done by performing the AND operation in every bit. However, because of space considerations, we decided to use the first approach.

STATIC BUS

To support multiple parallel transfers in the data path, we want to read and write registers in the same cycle. Since registers use transparent latches, the output of registers have to be disabled after phase 0. This will leave the buses in high impedance state after phase 0. The way we solve the problem is by introducing a static bus circuit to buses which are involved in the reading and writing of registers.

The static bus circuit consists of an inverter and a tri-state buffer. They are connected to form a latch which will be enabled after phase 0 goes low. Together with registers, the static bus circuit acts as a master slave flip-flop with the register as master and static bus as slave. Under this scheme, reading and writing to registers in the same cycle is possible without introducing the space penalty of using master slave flip-flops in the registers.

REGISTER GROUPS

The register groups in the chip are basically RAMs containing 16 words, each 32 bits long. There are four register groups each containing 16 registers. The first three register groups have only one input and one output. The first group is used for storing constants and the base addresses of the five stacks shown in *Figure 3*. This would allow experimenting with different sizes for the stacks in the data space. The heap usually occupies a good part of the data space. The pushdown list (PDL) is supported by using two register groups. The left and right parts are stored in the two groups. If a structure has more than 16 levels of nesting then the the bottom entry will overflow to memory.

The fourth group is used for argument registers (eight in all) and state registers such as E, B, TR, Heap backtrack pointer (HB), and continuation pointer (CP). The remaining three registers are used for book-keeping. The argument registers support fast execution of procedure calls and also data communication to the external environment. This register group has three inputs and two outputs. A multiplexer is used to select an input. It has two read ports so that two different pointers, for example, stack and trail, can be manipulated simultaneously. A detailed diagram of this register group is shown in Figure 11. RAM cells in this register group have two read ports and one write port. Two separate address decoders are provided for the two read ports. One of them is also used for the write port. Instead of having only one source for the address as in the cases of other RAM's, there are four ways the address for the register group can be generated. Three ways are used to generate the address for the lower eight words and the fourth way is used for addressing the upper eight words in the register group. The three ways use the lower three bits of Arg1, Arg2, and three address bits from the microinstruction. The fourth way uses the same three address bits from the microinstruction. The four ways of generating addresses are controlled by two bits from the microinstruction. Each of the two read ports has its own addressing bits supplied by the microinstruction. So, they are independent of each other. This register group is one of the important blocks since it is used often.

A macrocell has been designed for the register groups with a worst case read time of 30 ns. The implementation is similar to that of the non-master/slave registers except that the transparent latches are replaced by an array of 1-bit RAM cells. The read (write) enable control signal, an output of the decode logic, and phase 0 (phase 1) of the clock are put through an AND gate as in the case of registers. The storage part of the macro cell is organized as an array of 32 rows and 16 columns with 1-bit static RAM cells. Eight RAM cells in each row are connected to a bit line bus. Each bit line bus is connected to a huge tristate buffer that drives an output bus. To reduce the read time, p-devices are used to precharge the bit line buses. The transistors in the 1-bit RAM cell are sized so that they can pull down the bit line bus in less than 30 ns and the p-device can pullup the bus in less than 50 ns.

The current access time for the register groups is less than 30 ns (worst case) with the exception of the dual ported register group which is about 35 ns (worst case). Since the dual ported register group is not in the critical path, the extra delay does not affect the cycle time.

COLLISION DETECTION

The data space is divided into four parts to contain global heap, heap, stack, and trail. The starting addresses for the four parts are stored in the constant RAM block of the data path. As data items are entered into the global heap it is possible to exceed the space allocated and go into the heap area. A similar kind of situation can happen between heap and stack, and stack and trail. These are called collisions and they have to be detected and reported to the host system.

To detect collisions in parallel with the data transfers in the data path, parallel hardware is included in the data path. The top 15 bits of H2, H, and S are compared with the base values for heap, stack, and TR respectively. If the two are identical then there is a collision and a signal is generated and stored in the program status word (PSW). The comparison is done on 15 bits instead of 28 bits since not enough time is available during register transfers in phase 1. Note that since we are not comparing 28-bit addresses the collisions are detected at the page level, where a page is 8K words in this context.

4. MICROCONTROL DESIGN

The microcontrol comprises a microsequencer and a status unit. The microsequencer supplies the address of the next microinstruction to be executed. A block diagram of it is shown in Figure 12. To keep the design simple it supports just one level of microsubroutine and one level of interrupt. Two 9-bit registers, microreturn pointer (urp) and control microreturn pointer (curp) are included in it to store return addresses. Fast microbranching is supported by partitioning the ROM into four pages and using logic to modify the two most significant bits (page bits) of the next microaddress seed. The micro page select (upage_select) logic modifies the page bits according to the current status and directives from the microinstruction.

The next microaddress is selected from different sources according to the current status and directives from the microinstruction in the micro program counter select (uPCselect) circuit. The potential sources for the next microaddress are: modified next address seed, new opcode, arg1 register, subroutine rom, microreturn register (urp), and control microreturn register (curp).

Both upage_select and uPCselect circuits have been designed using the tree-height reduction method proposed by Kuck [8]. Although the tradeoffs involved (e.g. different basic cells have different fan-out

capacity, there might be too many cells needed, etc.) are too complicated to obtain the optimal circuit, we used approximate circuit breakdowns and obtained good performance.

Since only the most significant two bits (tag bits) of the seed need to be modified by the upage_select circuit, there are four possible ways of accomplishing the modification. We have a choice of generating the encoded version or decoded version (i.e. either have a two bit output or a four bit output) of the page number. The two bit version is heavily favored since it needs fewer component counts. It also runs faster than the four bit version since it is necessary to change the output of the four bit version back to the two page bits.

Optimizing the design of the uPCselect presented some challenges. The implementation produces a two bit encoded signals and then decodes them to four control signals. The only consideration used in the optimization is the reduced component count. It is believed that fewer components indeed would lead to faster circuits, but it is not clear whether the time saved would be more than the additional decoding time needed.

We have designed a uPCselect circuit with a delay of 35 ns. The selected next microaddress is then supplied to the external_mux block in Figure 12. It is also stored in the control microreturn pointer (curp) register. If no interrupts and exceptions are present then the address supplied by the uPCselect logic is sent to the ROM latch. This completes the operation of the microsequencer and the total time available to the microsequencer is 42 ns. If exceptions occur then the address of the exception handler routine is supplied to the ROM latch. If an external interrupt occurs, then the address supplied along with the interrupt signal is supplied to the ROM latch. The top level diagram of the microsequencer and the tree circuits of upage_select and uPCselect are shown in Appendix 1. We implemented the two circuits using PLAs and random logic and selected the latter because of its speed.

STATUS

The status unit contains the current state of the PLM. The state information includes condition codes, tag bits of MDR, T, T1, cdr bit of MDR, and tags of selected argument registers. The condition codes are generated during the previous cycle. They are latched into the status unit during phase 0. The status unit delivers the state information quickly to the microsequencer. The 18 bits of the status unit are stored in

LSSD [6] registers so that the chip can be tested by initializing the chip to a known state. The unit contains a shadow register block and an LSSD block. The shadow register block stores the two most significant bits (tag bits) of the registers AX0 - AX7 in the the dual ported register block of the data path. The shadow register block is written into during phase 1 when a write to AX0 - AX7 is performed in the the dual ported register. The contents of shadow are available to the microsequencer within 10 ns from the time phase 0 goes high. This fast delivery of state information is needed to meet the microsequencer timing constraints.

5. ROM, MIR, and INSTREN DESIGN

ROM

One of our goals is to design the ROM with a read access time of 40 ns. The NCR design team supplied the ROM as a macrocell. The circuit diagram of the ROM and the timing diagram are shown in Figures 13 and 14. The ROM is organized as a NOR array with 128 rows and 640 columns. The 640 columns are divided into 160 groups with 4 columns in each group corresponding to the four pages. The least significant seven bits of the ROM address specify the row to be read. The most significant two bits specify the column. The worst case read time depends on the output capacitance on the 7 input NAND gate. This capacitance increases as the number of zeros stored in a row.

The ROM uses a precharge scheme to reduce the read time. The reading takes place during phase 1 and the values of the 160 bits are supplied to a latch. During phase 0 the value in the latch is sent to the microinstruction register (MIR). The values of eight bits at the end of a word in the latch are also supplied to output pad drivers for communicating them to the cacheboard.

MICROCODE GENERATION

Most of the microcode for the chip is generated from the microarchitecture simulator using programs. The microcode is stored in the ROM. Almost 300 locations in the ROM are used to implement the PLM instructions. The state diagrams for the instructions are included in Appendix 1. The remaining 212 locations are used for builtin functions, initialization, and debugging. The microinstructions are 160 bits long in the chip compared to 144 in the TTL version of PLM. This is because the number of buses in the chip and the implementation of PDL are different from the TTL version. There are also additional blocks in

the chip to handle heap/stack and stack/trail collisions.

To generate microcode, programs are written in "AWK" and "C". One program determines the buses to be used for each microstate so that bus conflicts would not arise. A second set of programs are written to generate values for the fields of a microinstruction corresponding to the data transfer part of the microstate flow chart. Another program generates the ROM address for the next microinstruction from the next state part of the flow chart.

MIR DESIGN

The MIR contains the current microinstruction. It is implemented as an LSSD [6] register. The contents of MIR are supplied to the data path and microsequencer for the entire cycle if the chip is not in test mode. Since each bit in MIR has to drive logic in 32 bitslices, buffers are needed to reduce the delay. The buffers on MIRD block of Figure 2 are designed so that within 8 ns of phase 0 going high the control point values will be available to the farthest bitslice in the data path. The chip can be put in the test mode by asserting the TEST1 pin in Figure 15. In the test mode the MIR can be loaded with data on the SHIFTIN1 pin by shifting it using the SHIFTA clock. Any microinstruction can be loaded into the MIR and executed. The results can be observed by reading them using the MEMDATBUS.

INSTREN DESIGN

The PLM instructions usually take several cycles to execute. It is possible to prefetch the next instruction for most of the PLM instructions to avoid delays in starting the next instruction. The cycle at which prefetch can be performed is indicated by the microbits PREF1 and PREF2. The PREF1 bit when asserted indicates that the next PLM instructions opcode and first argument can be fetched from the cacheboard. The INSTREN pin of the chip in Figure 15 is used to communicate the prefetch signal to the cacheboard. The cacheboard supplies the instruction and a 32-bit argument within 10ns from receiving INSTREN. The PREF2 bit when asserted indicates that the second and third arguments can be fetched during the cycle if the opcode of the next instruction indicates that arguments two and three are needed. Three bits (bits 4, 5, and 6) of the 8-bit opcode indicate the number of arguments and the size of an instruction in bytes. The INSTREN signal is then generated and communicated to cacheboard. The data supplied by the

cacheboard in response to INSTREN is stored in ARG2_3 block of Figure 5.

6. DESIGN VERIFICATION

The verification of the design has been the most complex and time consuming activity. We did functional and timing simulation to verify the design. The complexity of the design prohibited us from starting the functional simulation at the chip level. The use of master/slave registers, latches, two phase clock, and a complex next microaddress selection scheme based on tags and condition code required us to start the timing simulation at the block level.

FUNCTIONAL SIMULATION

A hierarchical methodology is used in the functional simulation. The individual blocks of the chip; units such as data path, sequencer, MIR, and status; and the entire chip formed the three levels of the hierarchy.

We used the QUICKSIM program of IDEA station for simulation. Each block in the data path and microsequencer is functionally simulated by applying all possible values for the control inputs coming from MIR. The functional simulation of the ALU is carried out using two programs. All the functions of the ALU are exercised by using a given operand for the A and B inputs of the ALU in the first program. The add and subtract operations of the ALU are performed for a set of patterns by the second program. All functions of the remaining blocks have been exercised by a select set of input data.

Following the functional simulation of individual blocks, entire units in *Figure 4* are simulated. Exhaustive simulation is not possible because of the large number of inputs. For example, the microsequencer has 18 inputs from the status unit and its operation is dependent on these inputs. But it is not possible to make a simulation run for each combination of values. So, programs have been devised to reduce the number of simulation runs. Programs have also been written to check the results of the simulation runs. Simulating the data path as an unit presented a number of challenges because of the diversity of the blocks. We first identified 14 classes of transfers that can take place in the data path based on the register transfers in the microcode. For each of these classes we used a set of data values on the input buses and observed the outputs.

Functional simulation of the entire chip has been done using benchmark programs. The simulation and verification process used the programs shown in *Figure 16*. The ROM of the chip is first loaded with the microcode. The execution of benchmark programs is simulated by reading the sequence of microinstructions from the ROM for each PLM instruction and supplying the control points to the other units within the chip. Inputs to the chip are supplied on the MEMDATBUS and outputs are observed on the MARBUS and MEMDATBUS. For each cycle the contents of the key registers and the next microaddress are saved. This state information is compared with the output of the microarchitecture simulator for each cycle when it is executing the same set of programs.

The set of benchmarks used to verify the design is shown in Appendix 1. An example of a set of stimulants supplied to the QUICKSIM and the output from it are also shown in Appendix 1.

TIMING

The need for reading and updating a register in the same cycle dictated a two phase clocking scheme. For example, data can be read from T and the register group, CONSTRAM, that contains STACKbase during phase0. During phase1 the add operation can be performed in the ALU and the result stored in T. We used delayed branching and a pipeline register for the microinstruction (MIR) so that the data path and the microsequencer can operate in parallel.

To understand the timing issues in all the blocks of the data path, 14 different classes of data transfers are identified in the microarchitecture. For each of the 14 classes timing diagrams are drawn using the two phase nonoverlapping clock. The registers and register files are read during phase 0 and written into during Phase 1. The ALU is combinational and it is assumed that it will supply the result 10 ns before phase 1 goes low. In the case of master/slave registers, the master is written into during phase 1. The transfer from master to slave and reading the slave occurs during phase 0. A set of timing diagrams is included in Appendix 1. This set is useful for cache board designers and also interfacing the chip to standard buses.

TIMING SIMULATION

The timing simulation has been done using NODEDELAY, PATHDELAY, and QUICKSIM programs with estimated capacitances to determine the delay through the blocks. The timing information is used in redesigning the blocks to achieve the 100ns cycle time. The timing simulation programs are run for each of the blocks in the data path and microsequencer to calculate the delay in each block. The blocks have been redesigned by adding bigger drivers, or changing the circuits to reduce the delay and to meet the timing constraints. For example, the ALU initially took 70 ns to do ADD operation since we used four input gates. We redesigned the ALU by using 2 input NAND and NOR gates, OR-AND-INVERT(OAI 121) gate, and AND-OR-INVERT (AOI21) gate. The SPICE run on the ALU showed a worst case delay of 37 ns.

The results obtained from the timing simulation on individual blocks are used to determine the time delay for each of the 14 classes of data transfers in the data path. Whenever the calculated time delay exceeded the specified value the blocks have been redesigned to meet the specifications.

The timing simulation on the blocks of the first version of the microsequencer resulted in a complete redesign since the estimated delay exceeded the 42 ns constraint by almost 50%. Another unit of the chip that had to be completely redesigned to satisfy timing requirement was the MIR. The LSSD registers of the MIR have been redesigned to reduce the time delay.

7. PHYSICAL LEVEL DESIGN

The area of the chip, if it could be routed conventionally, has been estimated at nearly 1.8 cm on a side, and our cycle time target of 100 ns would be missed by at least 50%. Rather than accepting this poor result, NCR developed a new approach to implement the physical design. The goal is to develop a placement and routing methodology that would enable us to build very complex structured designs (ones with more than 20,000 gates) with sufficient density to achieve efficient manufacturability and high performance.

The physical level design work for the chip is done by the Advanced Development group at NCR-Microelectronics in Ft. Collins, Co. NCR undertook this project with the desire to explore the problems associated with physical design of next-generation semicustom products. The PLM architecture was especially attractive, because the design requirements of the chip anticipated those of a growing class of

advanced processors. It was, nonetheless, beyond the capabilities of current commercial semicustom design systems. Implementation of the PLM architecture has provided an opportunity to explore hierarchical, cell based design representations for complex, structured logic.

The concept of cell based design for VLSI is itself a form of hierarchical representation. Cells or blocks (both parameterizable and fixed) are designed with low level elements and transistors. Then, with some systems, the chip design can be based only on a behavioral abstraction to the cell level. Currently, however, the most common approach to physical layout is to extract a single netlist from a design at the cell level, without any further hierarchy. This so-called "flat" netlist is used by an automatic placement and routing program to produce a physical layout.

If conventional routing techniques are used to route a highly structured architecture like PLM, one finds cells associated with a given block distributed throughout the layout. A skilled designer, aware of the structure of the logic knows, for example, that data paths can be bit sliced; he can place the logic associated with a given bit very compactly, and efficiently for performance. However, a "flat" router has no apriori knowledge which permits such efficiencies. Scattering of highly related logic increases the wire length between cells, and in turn increases the die size and the path delays of the final chip. These factors can lead to very unsatisfactory results. It is possible with most tools to "seed", or specify the location of cells, to improve interconnection during automatic routing. This is, first of all, a very tedious procedure, but in addition, it has been our experience that seeding a placement is generally inefficient, and increases the die size over an unseeded placement.

TOPHER (TwO Phase HiErarchical Routing)

A hierarchical physical design procedure has been developed to address the problems and goals identified above. It was implemented for the chip as an extension of NCR's commercial cell based design system (VISYS), and progressed in the following way. The design was physically partitioned into blocks corresponding to the logical blocks shown in *Figure 4*. On first pass routing, the bit slices were routed inside each block, so that they would align with (pitch match) adjoining blocks. Design verification was done on the individual blocks, which are fairly manageable in size, allowing for easy detection of layout problems. The blocks were then wired together in a second pass through the router, after the chip pad cells

were added to the layout. The finished layout was then submitted to a full chip level design verification cycle very similar to that used for a conventionally placed and routed ASIC design. The two phase hierarchical routing is called **TOPHER**. An important aspect of TOPHER has to do with the use of the metal interconnect layers, and the layers on which ports are defined. NCR's commercial cell library uses 2 levels of metal for interconnect and power routing; metal-2 is routed vertically over the cells for reduced wire-to-cell ratio, and elimination or reduction of feed-through cells. Since metal-2 is largely free of area penalty, it is important to maximize metal-2 usage. Consequently, the commercial cell library has all port connections on metal-2, using metal-1 as the primary horizontal routing layer and for power and ground wiring. Metal-1, therefore, runs parallel to cell rows.

If this approach is used in the first pass of TOPHER routing for a block, a significant number of metal-2 wiring tracks would be consumed just by the input and output pins of the cells. These tracks would then be unavailable during the second pass of routing. To reduce the number of blocked tracks, we modified the cell library to have all inputs and outputs on the polysilicon level, instead of on metal-2. Outputs on polysilicon are, however, undesirable because of the high resistance associated with this material. It is most advantageous therefore, for this first-pass routing problem, to have input pins on the polysilicon layer, horizontal connections and power/ground on metal-1, and output pins on metal-2. Since the router supported only two routing levels, outputs had to be initially routed on the polysilicon layer, so all of the pins could be defined on only two levels (poly and metal 1). An NCR proprietary post-layout processing tool is used to automatically convert all polysilicon connected to output ports back to the metal-2 layer. This eliminated resistance at the beginning of a net, and provided significantly better performance than would a library actually designed with poly outputs.

CELL ROUTING

The process for cell routing begins with identification of the critical nets, the block level port locations, and the number of cell rows to be used. In order for each bit slice in a block to align to adjacent bit slices throughout the data path, the cells are manually placed. In this way, cells are positioned so that block level port connections would be approximately aligned to the predefined bus order in a bit slice. Block level port connections are the ports to be wired to in the second pass routing. They are either input or output ports to standard cells. The tedious manual procedure is due to the lack of schematic-capture-level

seeding capability in the router. A recent version of the router greatly automates this step, but it was unavailable at the time the blocks were routed for the chip.

BLOCK ROUTING

After cell placement, block control lines and other critical nets are identified and wired first, followed by the automatic routing of the remaining nets. By splitting the wiring into two groups, the critical nets could be more efficiently wired using an interactive mode of the router if desired, thereby guiding the route to the highest performance implementation possible. Upon completion of a block route, poly paths connected to output ports are converted to the metal-2 layer, as described earlier. The result is that the blocks are wired with three levels of interconnect: poly, metal 1 and metal 2.

Next, the blocks are loaded on a layout workstation. The graphics manipulation programs and an interactive editor are used to add block level port definition structures and metal-2 power buses. Three sets of power buses are placed in each block to provide an adequate power distribution network. The buses are placed next to bit 0, between bits 15 and 16, and between bit 31 and the control logic. At this time nets that did not route to completion are also finished and the blocks are now ready for verification.

In addition to the blocks in the data path of the chip, there are additional functional blocks associated with status and microsequencer units. These blocks are automatically placed and routed after defining the port locations. These port locations are selected to minimize excessive wiring in the global routing pass. Since many inputs to these blocks are from external pads, the ports are also placed to match the desired pin-out of the chip. The only other constraint placed on the router for these modules is their width which should not exceed that of the data path.

ANALYSIS

For each block, post layout timing analysis and network verification are performed by a set of NCR proprietary software tools known collectively as VITA. Additional process design rule and electrical rule checking is also performed to assure that the layout topology is correct. The VITA tool Interconnect Analysis (IA) is used to extract resistance and capacitance values that are back annotated into the design files so that functional simulation can be based on real values. For the entire chip, only those pins connected to block level ports still used estimated capacitance values after this back annotation.

The network comparison tool, NETCMP, is used to verify that the layout matches the netlist at the cell level. NETCMP does not do a transistor-by-transistor check, but it verifies the accuracy of connectivity between cells. The run time is considerably faster than the other, more detailed, check but gives comparable results if the cells are known to be correct. The design rule checks (DRC) and electrical rules checks (ERC) are run next, followed by a transistor level check between layout and netlist (LVS). This latter usually runs without reporting errors if any discrepancies found by the VITA programs have been fixed correctly.

GLOBAL ROUTING

The global routing of the chip is done by the second pass of TOPHER. The blocks of the data path, status, and microsequencer are wired together using the two metal layers. Power and ground routing is done prior to signal routing. As mentioned earlier, the bit slices for the chip are laid out to align when blocks are placed next to each other in the second pass. These bit slices are then wired together using metal-2. Control lines in this design run orthogonally to the data buses, and are in metal-1. Miscellaneous signals and pad connections are wired in either metal layer, in order to complete the layout.

8. CONCLUSION

The semicustom approach to logic design and TOPHER approach to physical layout has resulted in a chip design with 40,000 equivalent gates, and eleven 32-bit buses, on a die under 1 cm per side (approximately 130,000 square mils) having a simulated cycle time under 100 ns. A layout of the chip without global routing is shown in Figure 17. Note that almost 70% of the area is used up by the data path, and only 25% by the ROM and the microcontrol. The logic design, simulation, and verification effort involved the equivalent of five engineers working for 2 years. The physical design process involved two engineers, one working half time, for 1.4 years. We accomplished this using a conventional two-level router, and the chip is constructed in a manner compatible with existing design verification tools.

ACKNOWLEDGEMENT

The physical level design was done by Maurice Moll of NCR Fort Collins. We appreciate his efforts and those of Dan Ellsworth at NCR Fort Collins. We are thankful to Tep Dobry, the designer of the TTL version of the PLM, for explaining the microarchitecture and answering our questions related to the architecture and the microcode. We are thankful to Chien Chen for designing the ALU, Allen Wei for writing programs to check bus conflicts in microcode, Jim Testa for designing parts of the microsequencer and generating microcode from flow charts, Harold Crafts of NCR for developing the cell library, and Tara Weber for microcode generation programs. The comments and suggestions provided by the members of the Aquarius project are also appreciated.

This work is partially funded by Defense Advance Research Projects Agency (DOD) and monitored by Naval Electronics System Command under contract No. N00039-84-C-0089, NCR Corporation, Dayton, Ohio, and National Science Foundation. Equipment and other support for the project has been provided by DEC, NCR, Apollo, ESL, and Xenologic.

REFERENCES

1. NCR Corporation, *NCR/32 General Information*. 1983.

2. A. M. Despain, "Lecture Notes, CS 257," *CS Division, UC Berkeley*, (Fall 1984).

3. A. M. Despain, "Notes on Computer Architecture for High Performance," *New Computer Architecture, Academic Press*, (1984).

4. T. Dobry, Y. Patt, and A. M. Despain, "Design Decisions Influencing the Microarchitecture For A Prolog Machine," *Proceedings of the MICRO 17*, (October 1984).

5. T. Dobry, A. M. Despain, and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture," *Proceedings of the 12th Intl. Symposium on Comp. Arch.*, (June 1985).

6. E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the Design Automation Conference, pp 462 - 468*, (1977).

7. B. Fagin, Y. Patt, V. P. Srini, and A. M. Despain, "Compiling Prolog Into Microcode: A Case Study Using the NCR/32-000," *Proceedings of the MICRO 18*, (December 1985).

8. D. Kuck, "The Structure of Computers and Computations, Vol. 1," *John Wiley Press, New York*, (1978).

9. R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *Journal of ACM, Vol. 27, No. 4, pp 831 - 838.*, (October 1980).

10. E. Tick and D. H. D. Warren, *Towards a Pipelined Prolog Processor*, SRI International, Menlo Park, CA (August 1983). Technical Report.

11. D. H. D. Warren, *An Abstract Prolog Instruction Set*, SRI International, Menlo Park,CA (1983). Technical Report.

Table 1 VLSI-PLM Instruction Set Summary

Instructions	Cycles	Max. Number of Transfers	Dynamic Instruction † Frequency (%)
try_me_else	21	7	3.89
try	17	7	1.12
retry_me_else	3	7	2.63
retry	3	7	0.62
trust_me_else	5	6	2.08
trust	5	6	0.59
cut	8	4	2.24
cutd	1+7*l	5	0.04
fail	19+3*l	4	0.13
switch_on_term	5+d	3	5.46
switch_on_structure	11+d+4*l	4	0.36
switch_on_constant	10+d+4*l	3	0.14
allocate	6	6	4.37
call	1	4	3.08
proceed	3	4	2.51
execute	1	1	0.42
deallocate	5	4	3.19
escape	variable	6	3.48
get_list	3+t+d	5	5.15
get_structure	4+t+d	5	4.69
get_variable	5+u+d	6	4.82
get_constant	1+u+d	5	1.90
get_value	5+u+d	4	2.49
get_nil	2+u+d	3	0.91
put_value	4	5	9.37
put_constant	2	3	2.17
put_variable	4	9	3.34
put_unsafe_value	10+d	5	2.31
put_list	3	3	0.67
put_structure	3	3	0.26
put_nil	2	3	0.03
unify_variable	6+c+d	6	9.01
unify_cdr	5	5	4.07
unify_value	9+2*d+c+u	3	4.79
unify_nil	4+d+u	4	4.39
unify_constant	4+d+c+u	3	1.27
unify_void	2+8*l	5	2.04
add	3	6	0.1
sub	4	4	0.42
mult	85+number of ones in the multiplier	7	0.01
and	3	5	0.51
or	3	5	0.0
eor	3	5	0.0

jump	1	1	0.0
jumpxn	1	1	0.0
jle	5	4	0.0
jlt	5	4	0.0
jeq	5	4	0.25
memread	4	5	0
memwrite	3	5	0
coderead	4	3	0
codewrite	3	3	0
loadn	2	2	0
dereference	2+3*links	4	0.95
reset	5	5	0
noop	1	1	0
halt	1	0	0
boot	22	5	0

† Dynamic Instruction Frequency is obtained by executing the Big Benchmark Set and averaging the results.

where:

c - time for a decdr operation (= 2).
d - time for a dereference operation (= 5).
t - time for a trail operation (= 4).
u - time for a unify operation (= 3 + optional trail).
l - the number of loop iterations (>= 1).

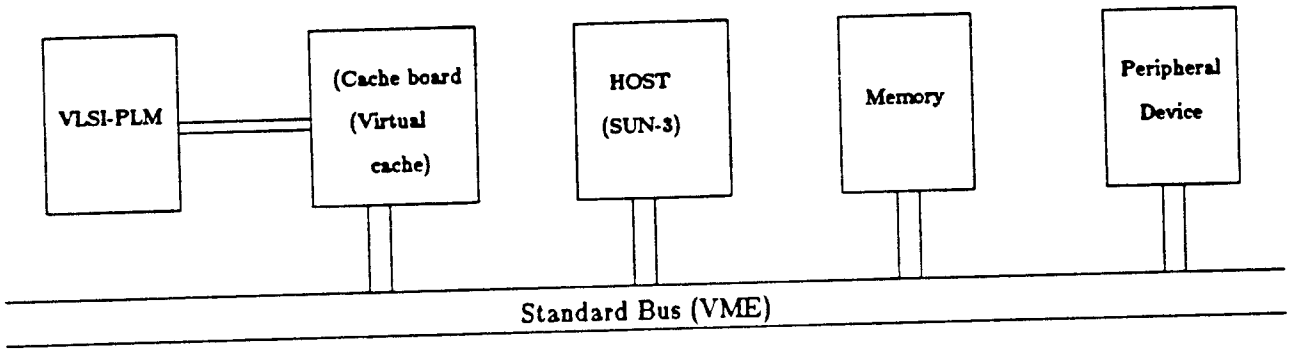


FIGURE 1 **BERKELEY PROLOG SYSTEM**

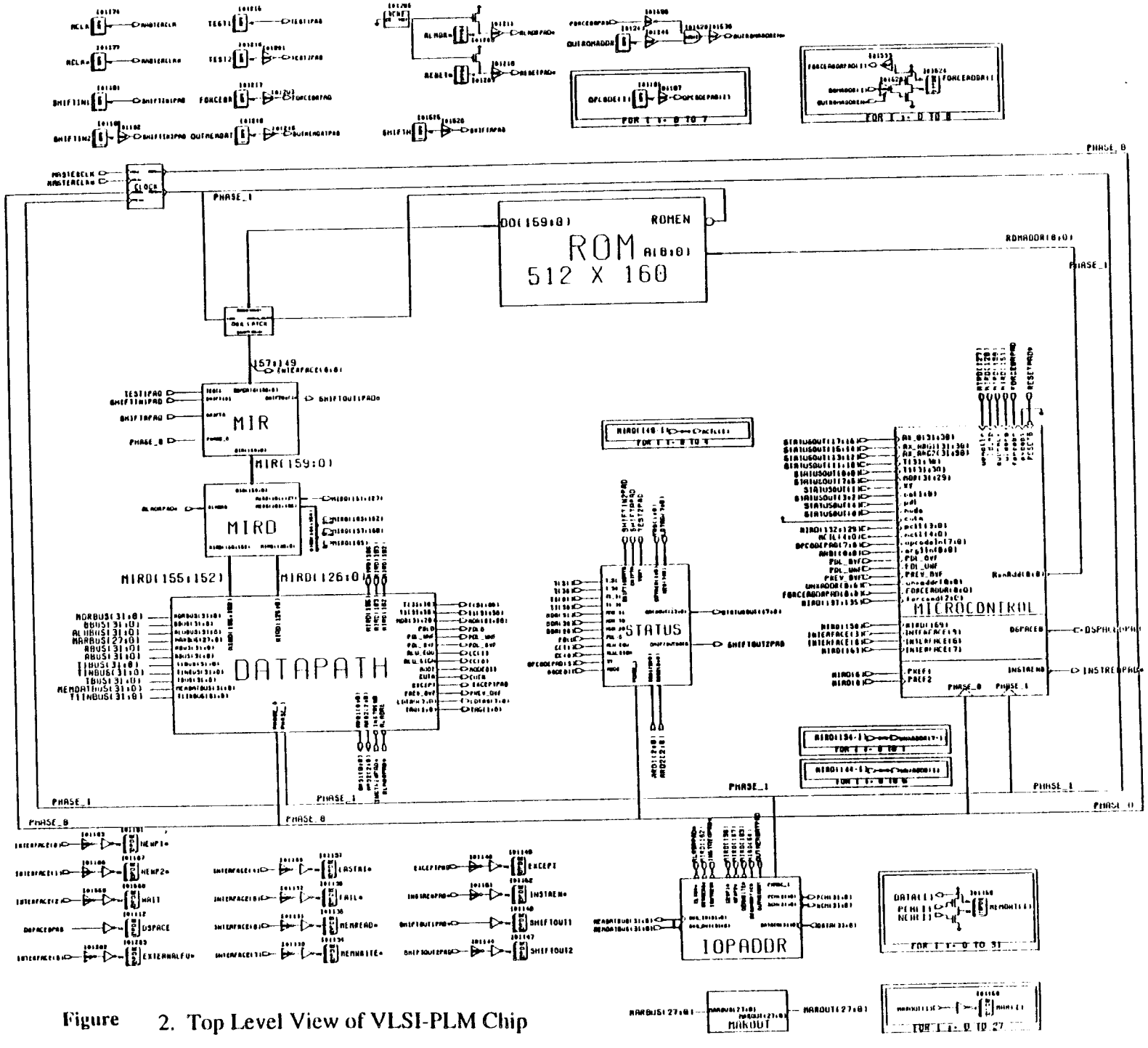


Figure 2. Top Level View of VLSI-PLM Chip

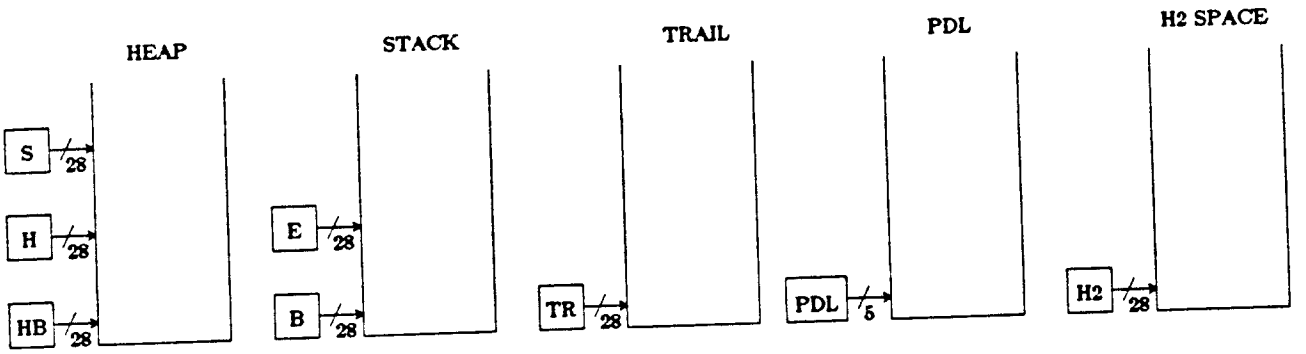


FIGURE 3 **STACKS IN THE DATA SPACE AND POINTERS**

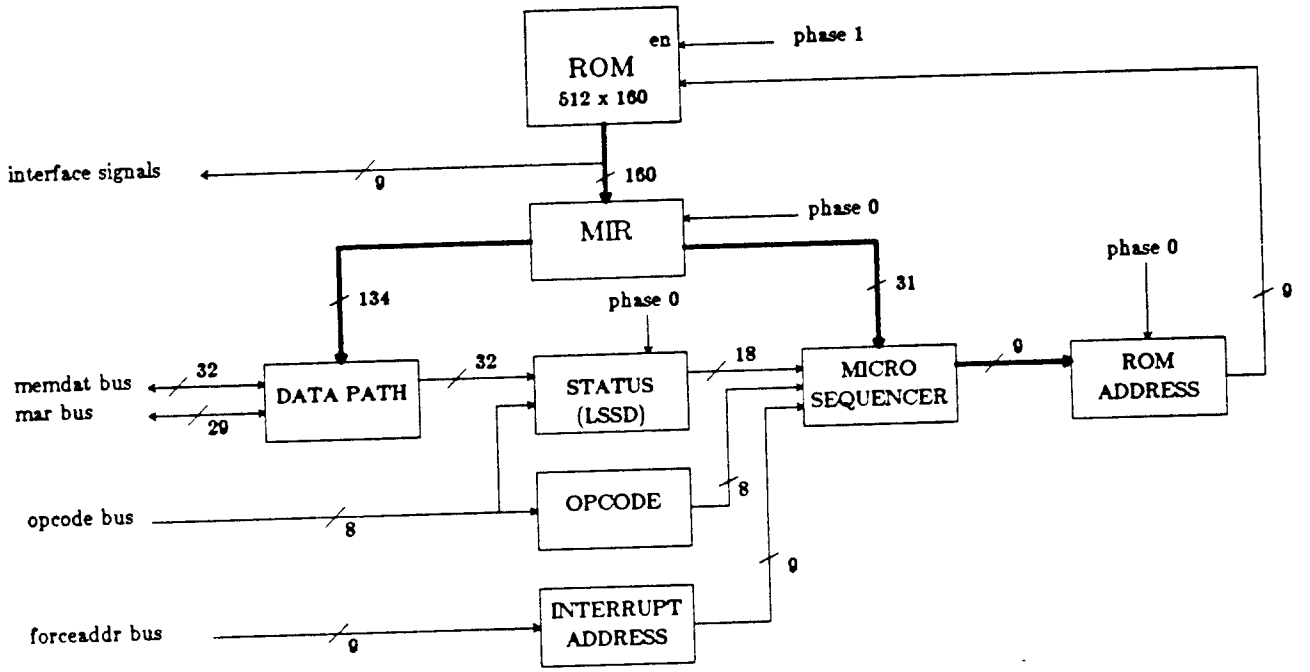


FIGURE 4 **BLOCK DIAGRAM OF VLSI-PLM**

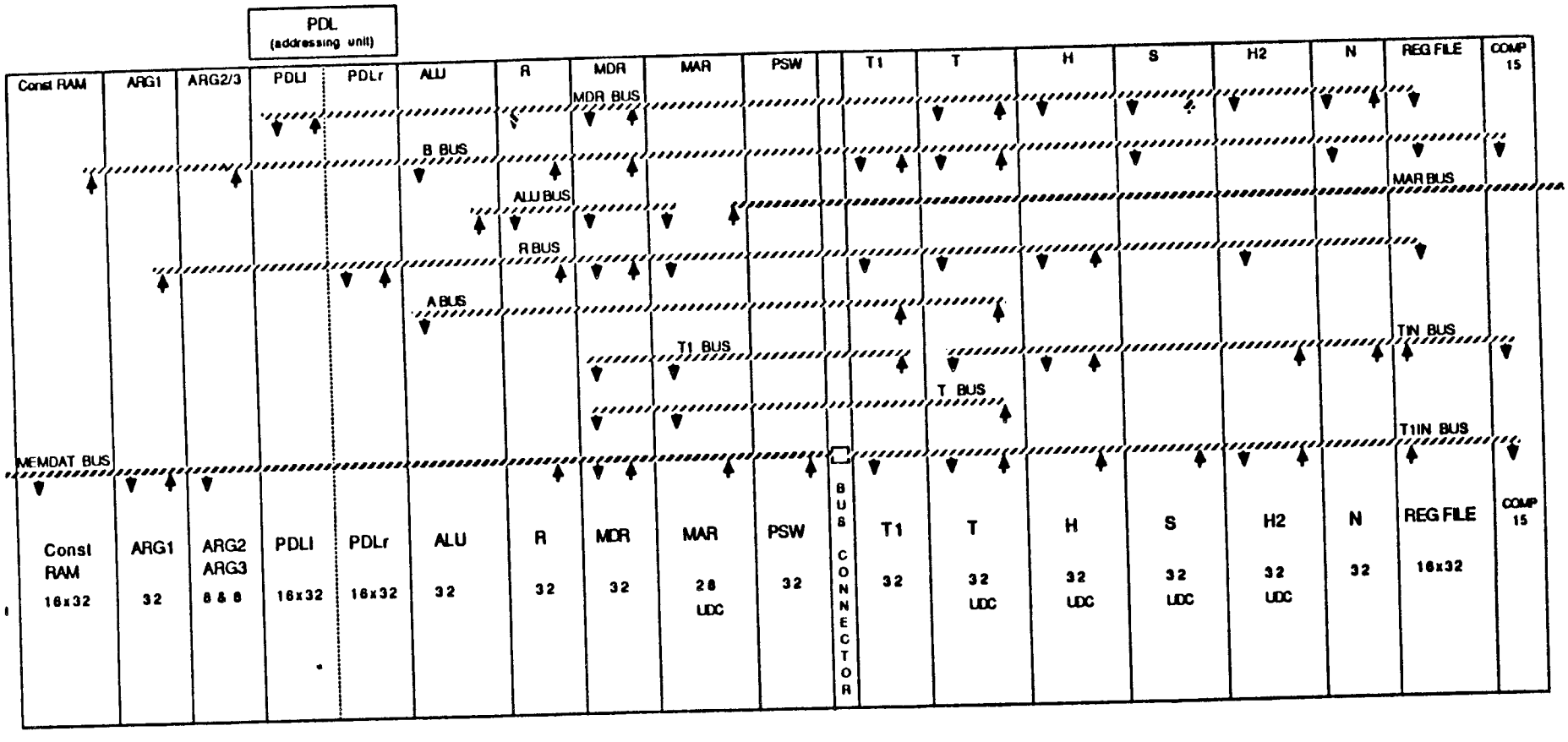


Figure 5 **Floorplan of Datapath**

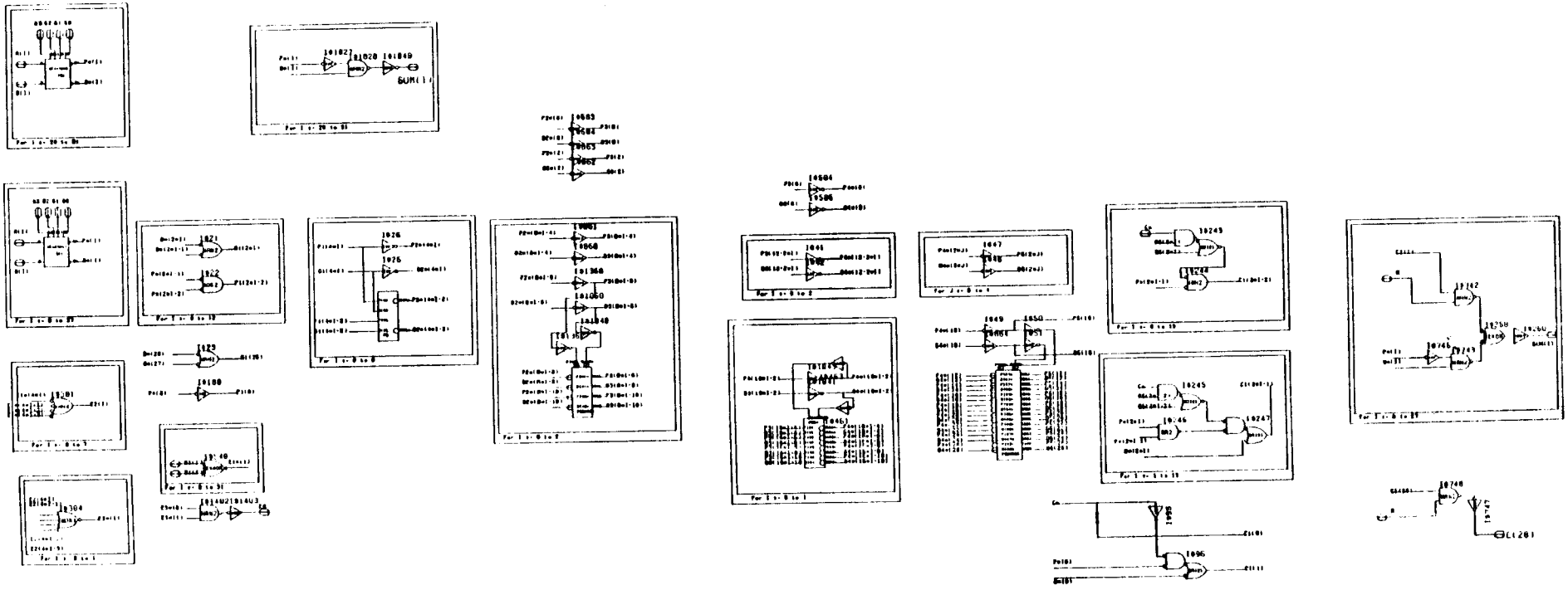


Figure 6. ALU's Top Level Diagram

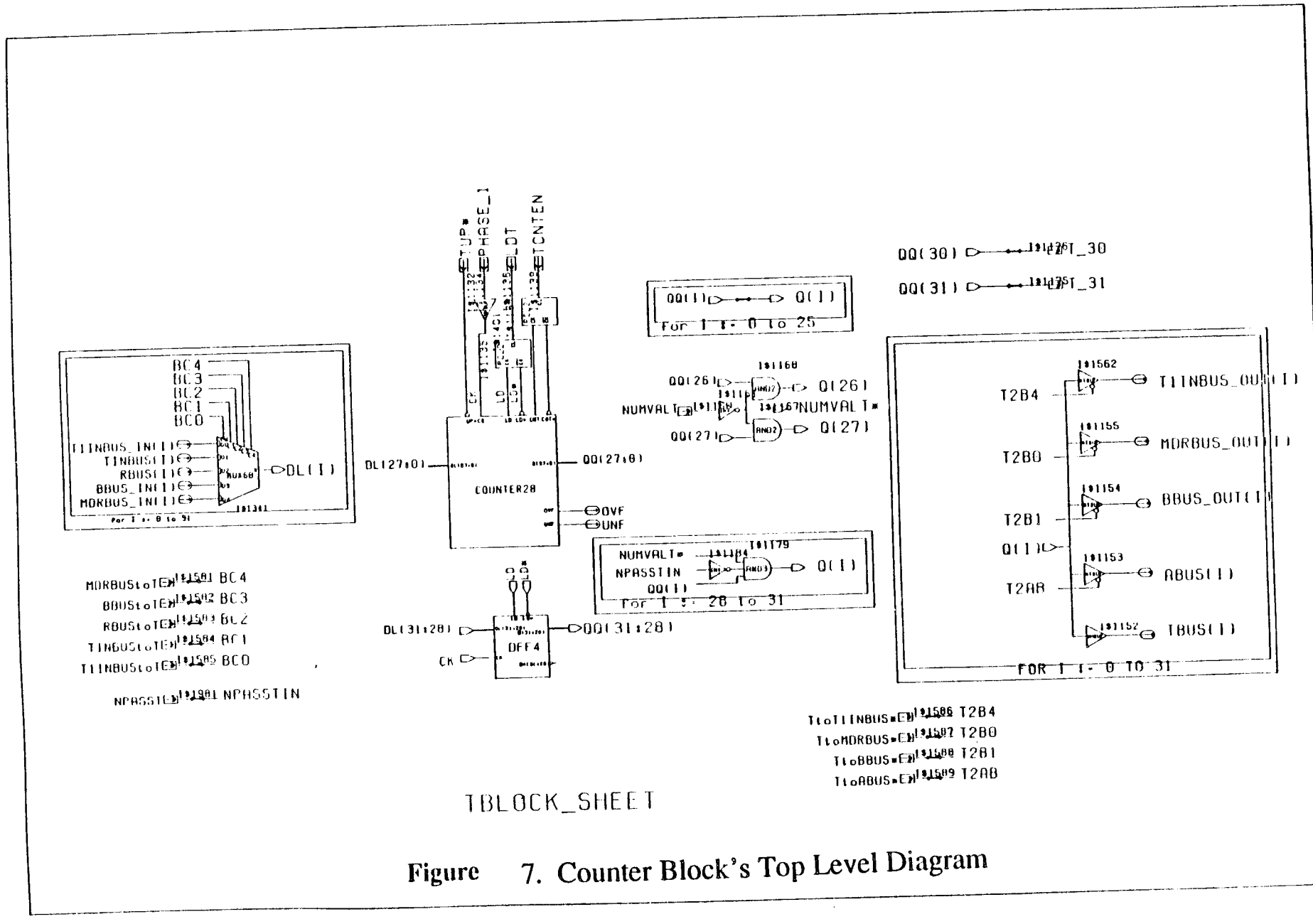


Figure 7. Counter Block's Top Level Diagram

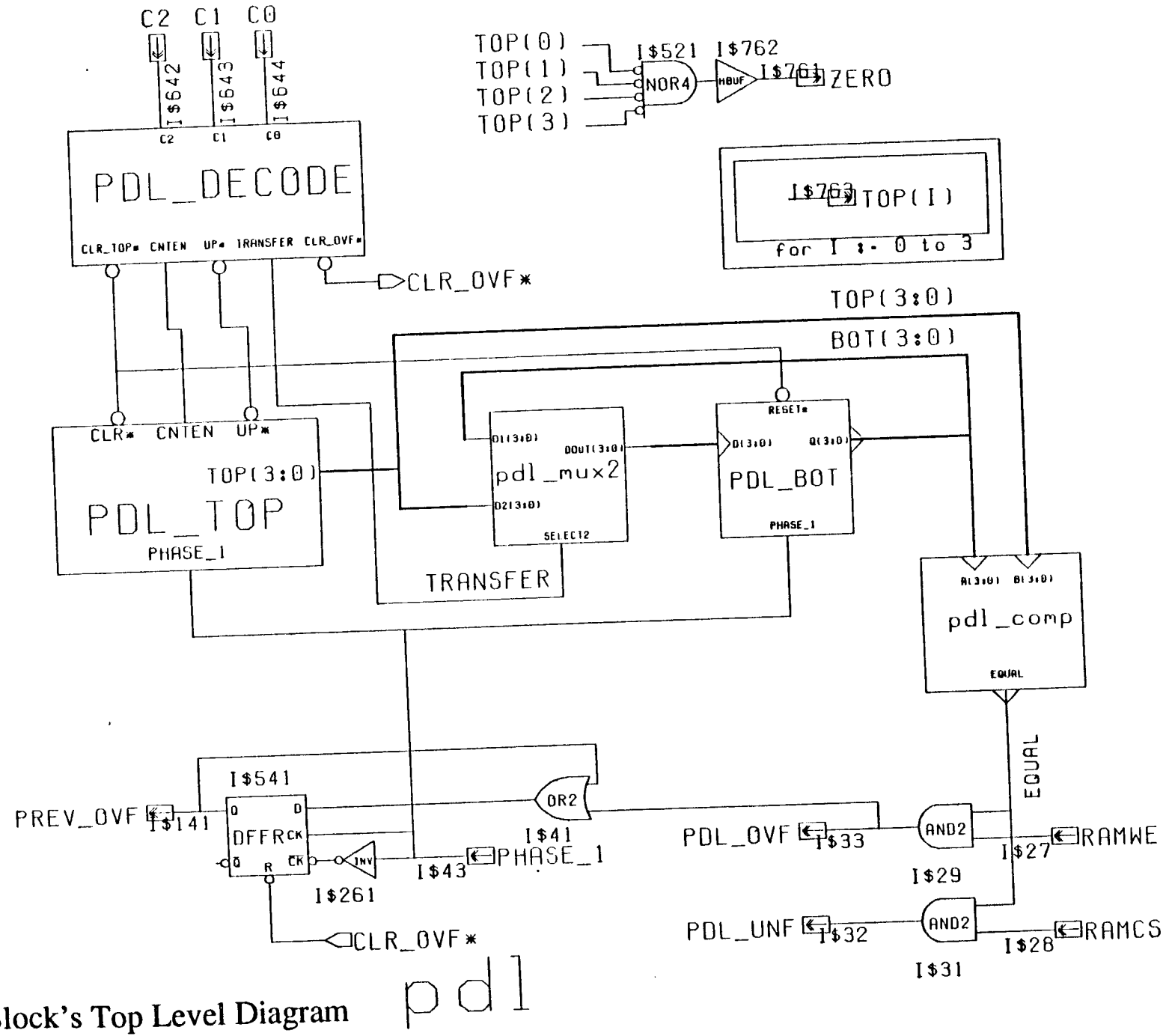


Figure 8. PDL Block's Top Level Diagram

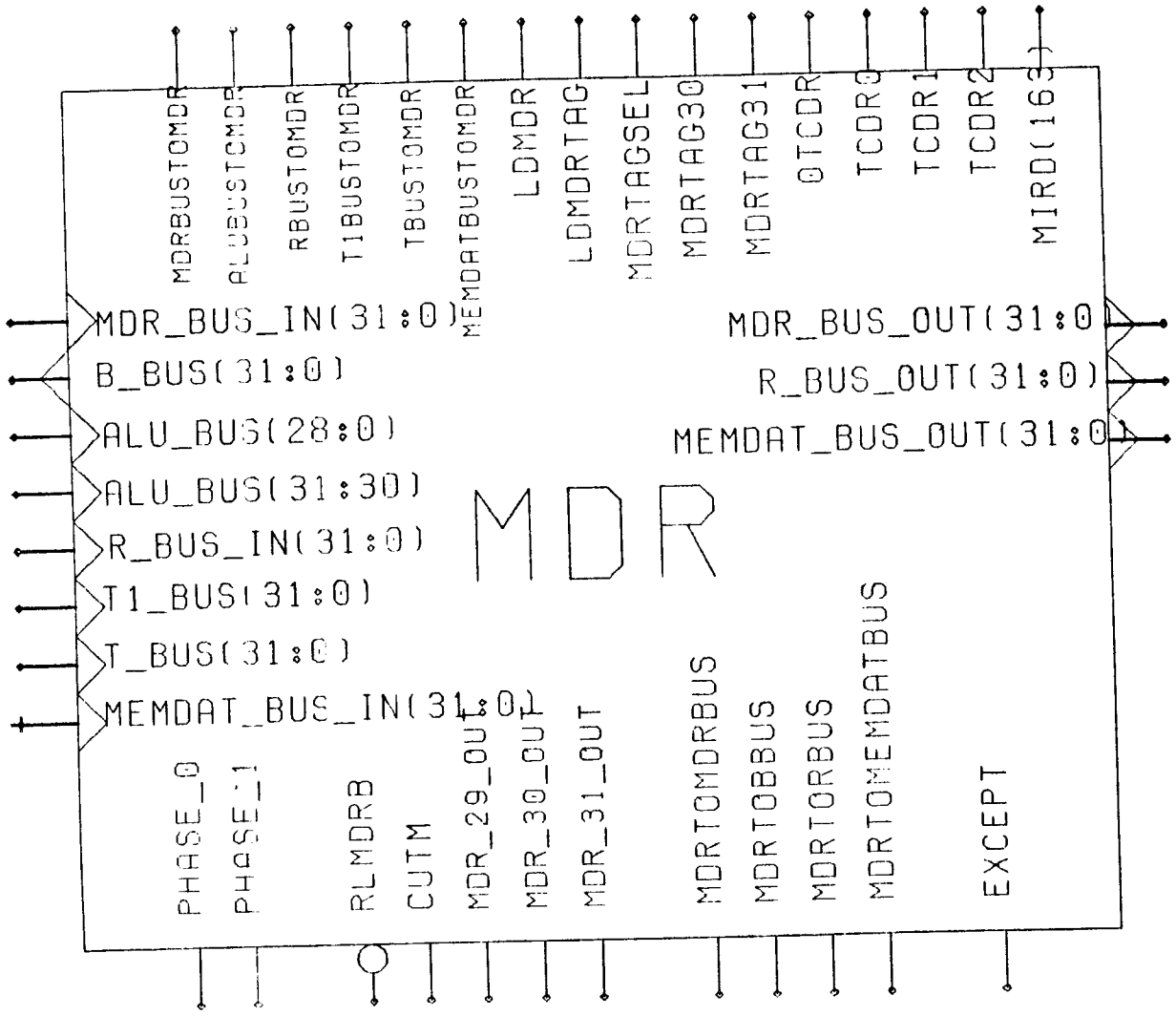
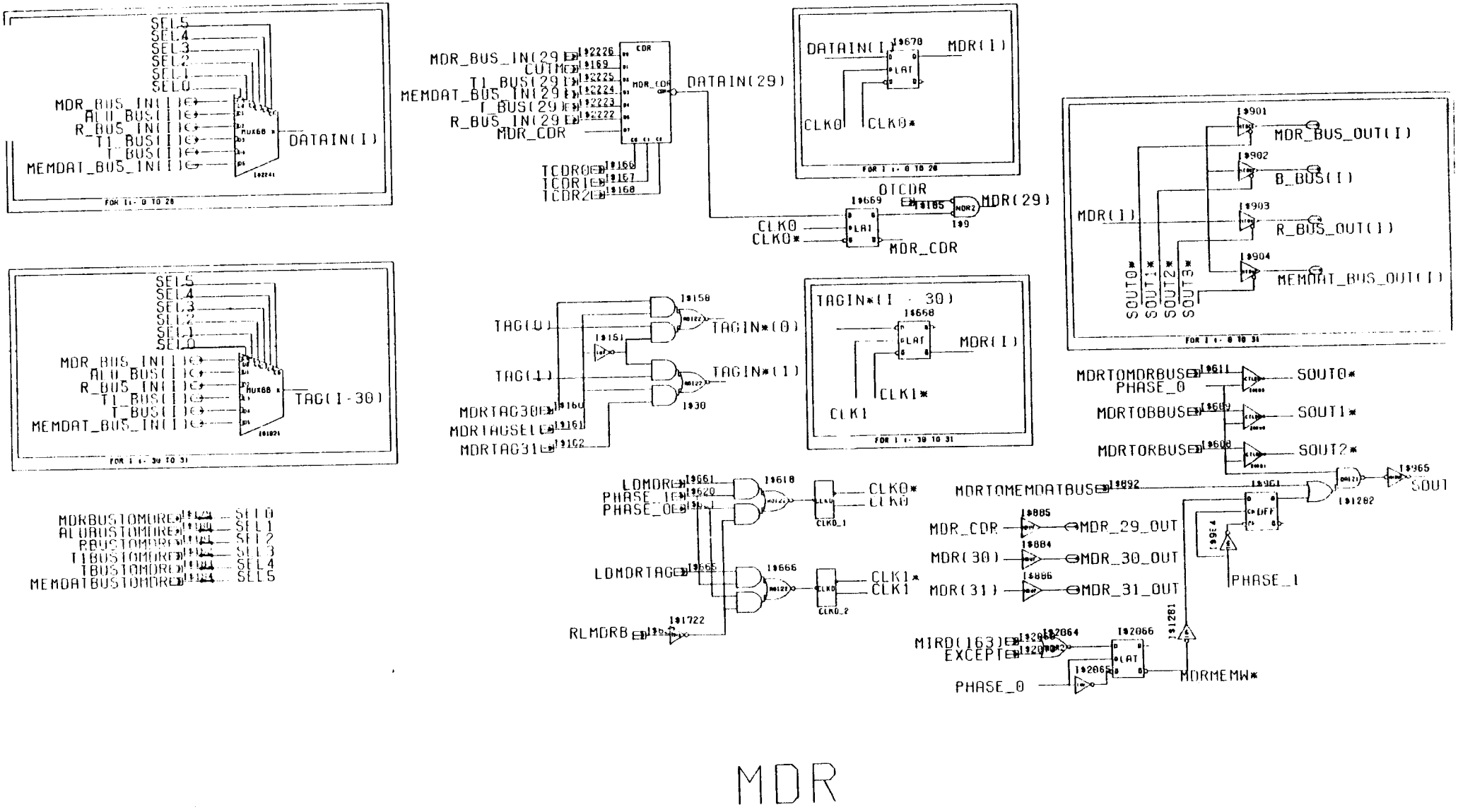
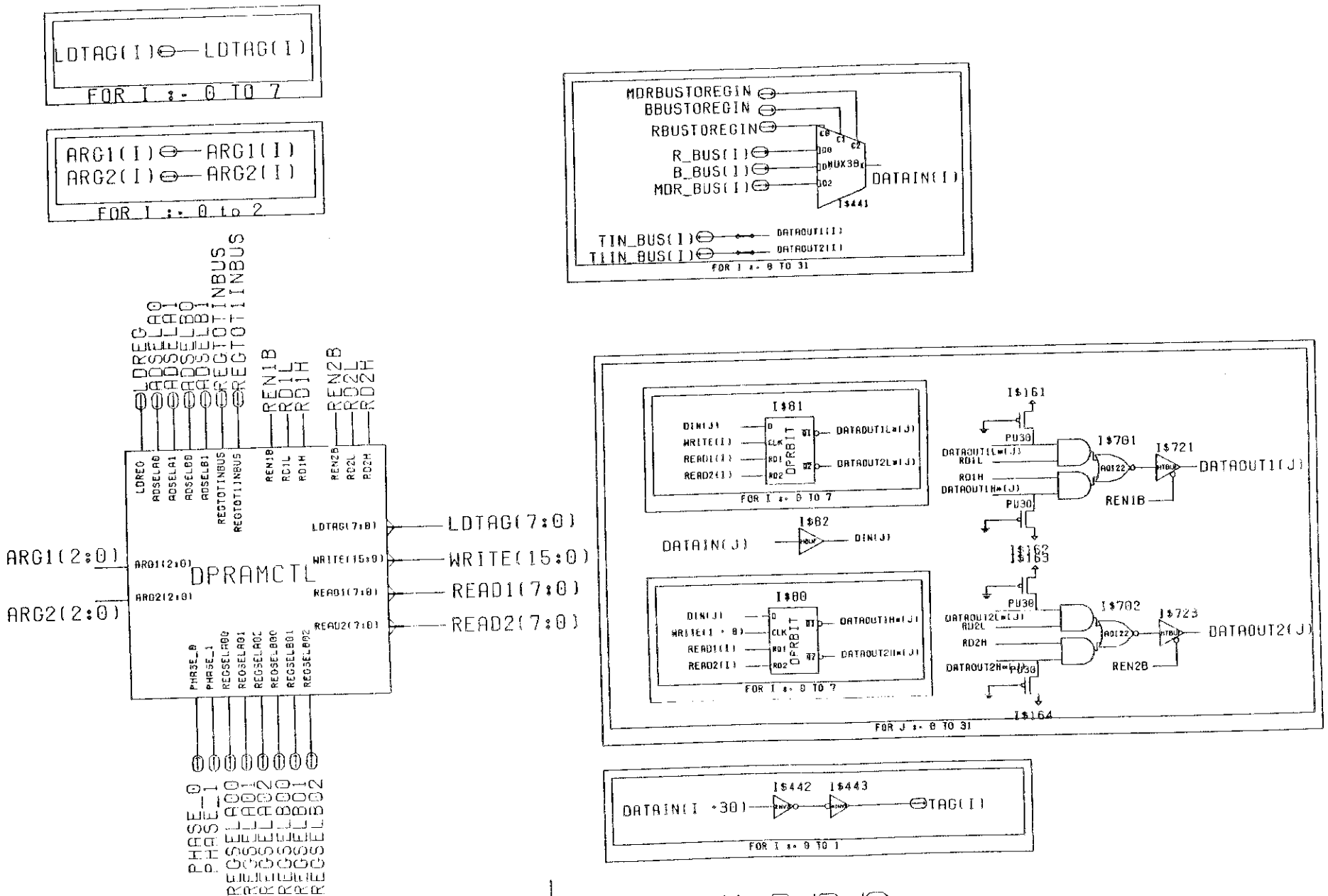


Figure 9. MDR Block's Symbol



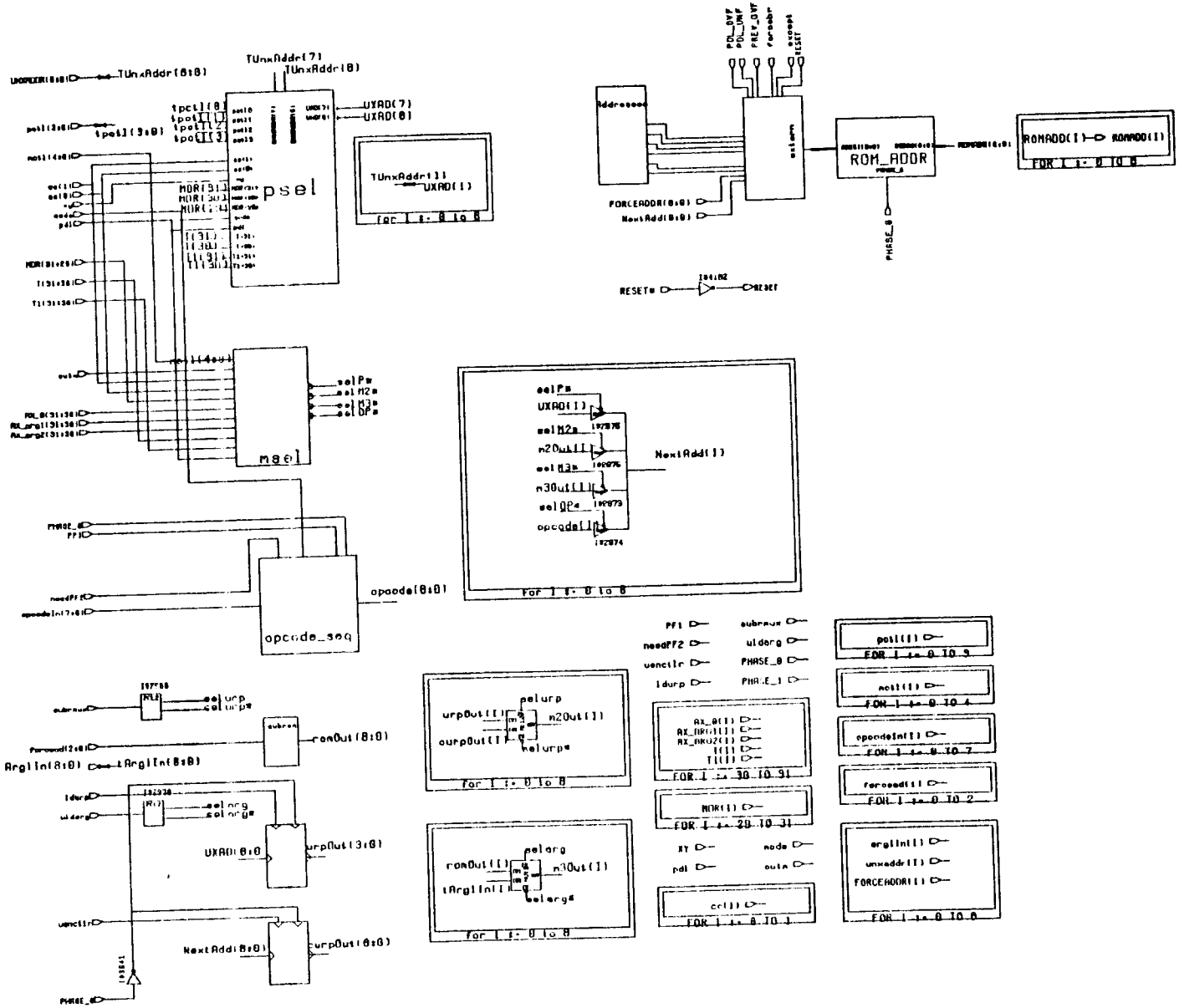
MDR

Figure 10. MDR Block's Details



dpramarr

Figure 11. Register Files's Details



microsequencer

Figure 12. Block Diagram of the Microsequencer

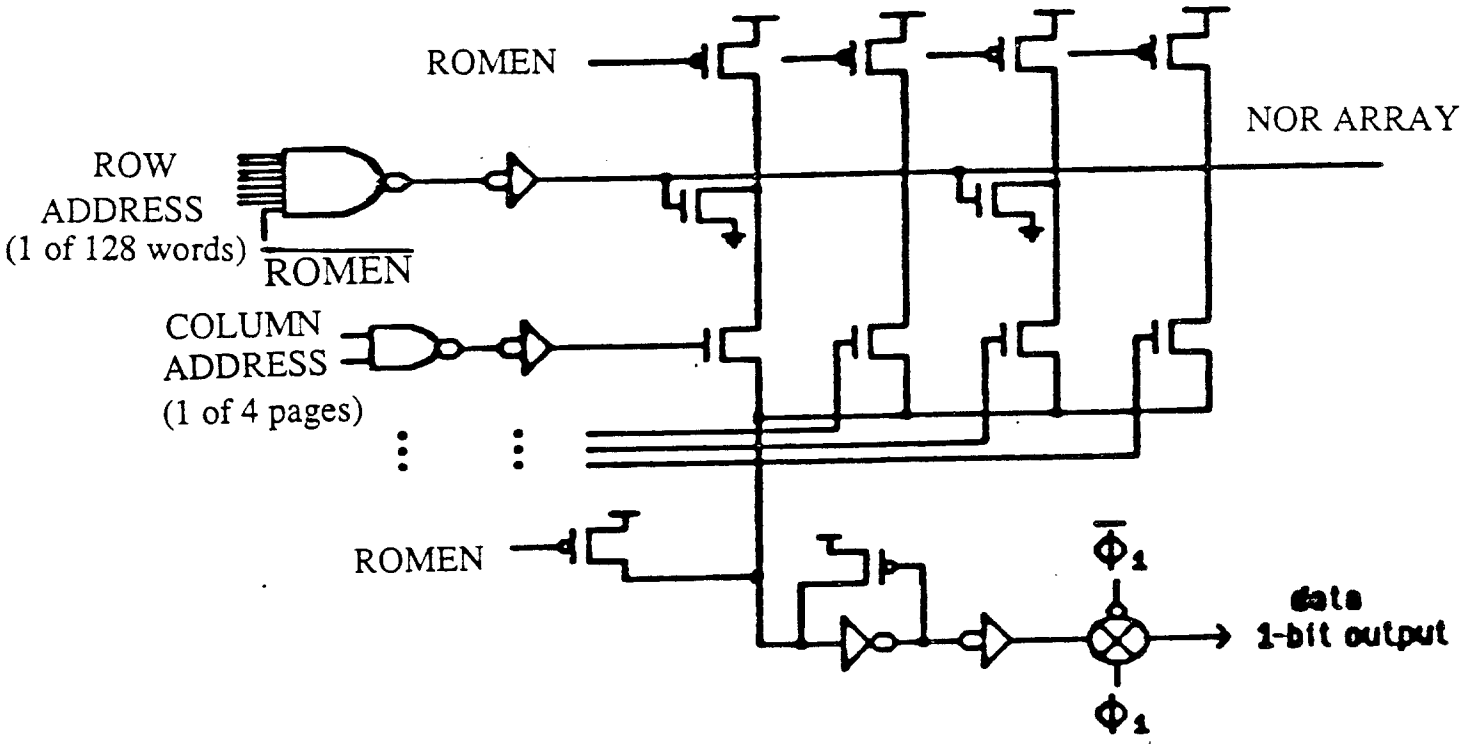


Figure 13. Circuit Diagram of ROM

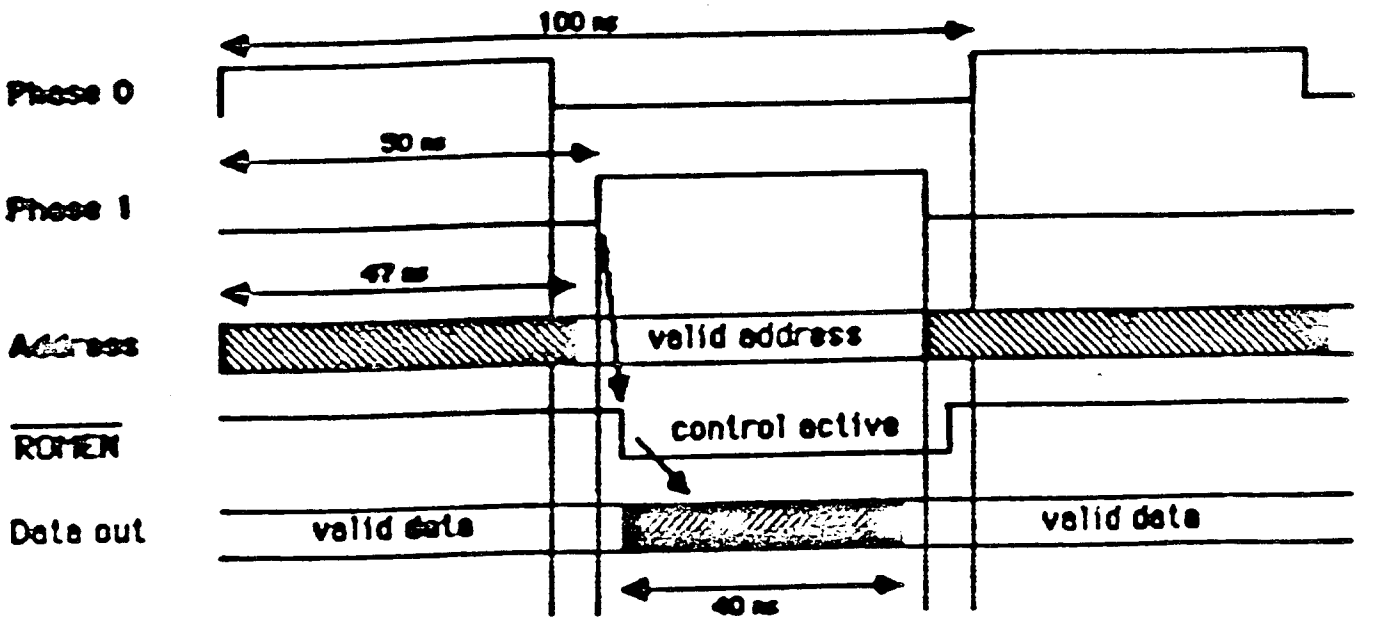


Figure 14. Timing Diagram of ROM

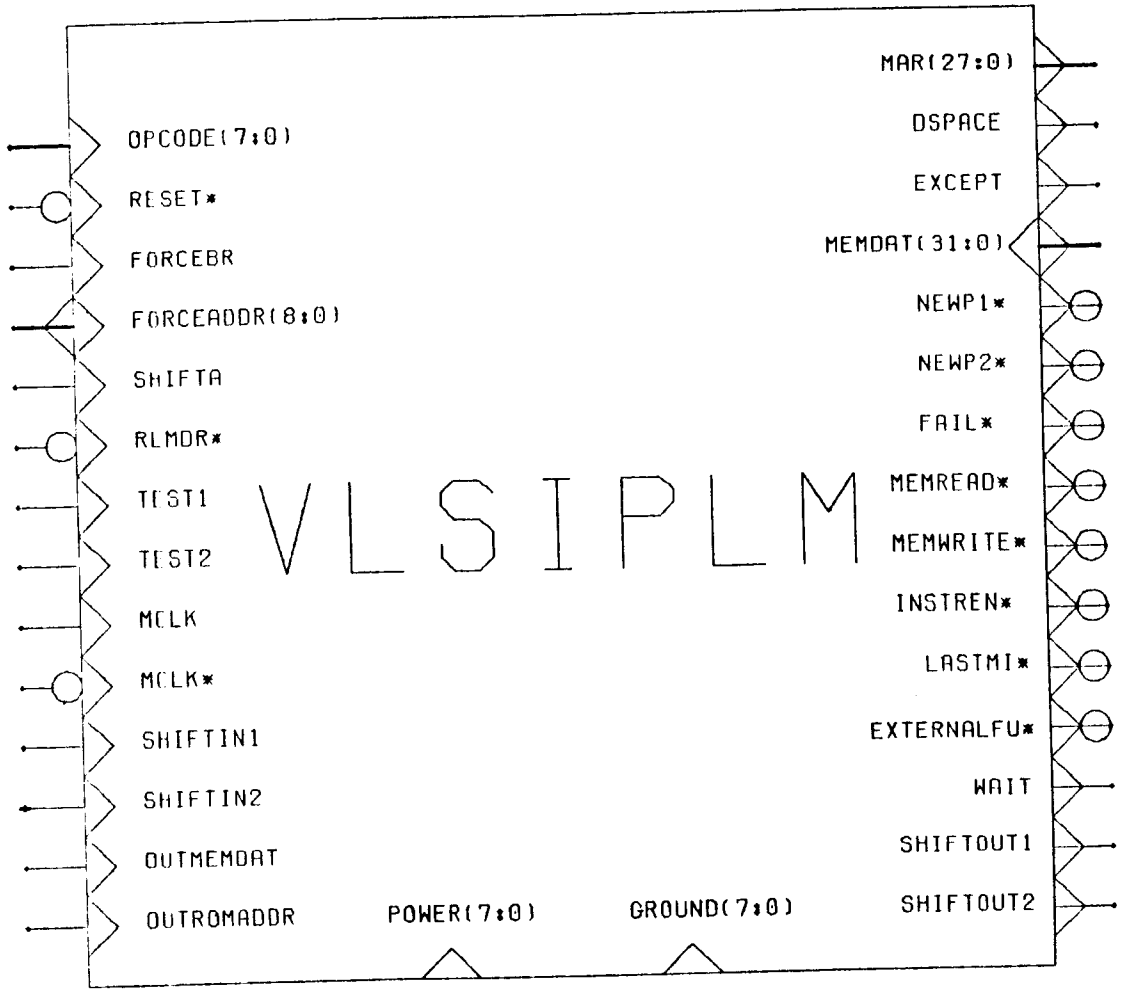


Figure 15. Pinout of the Chip

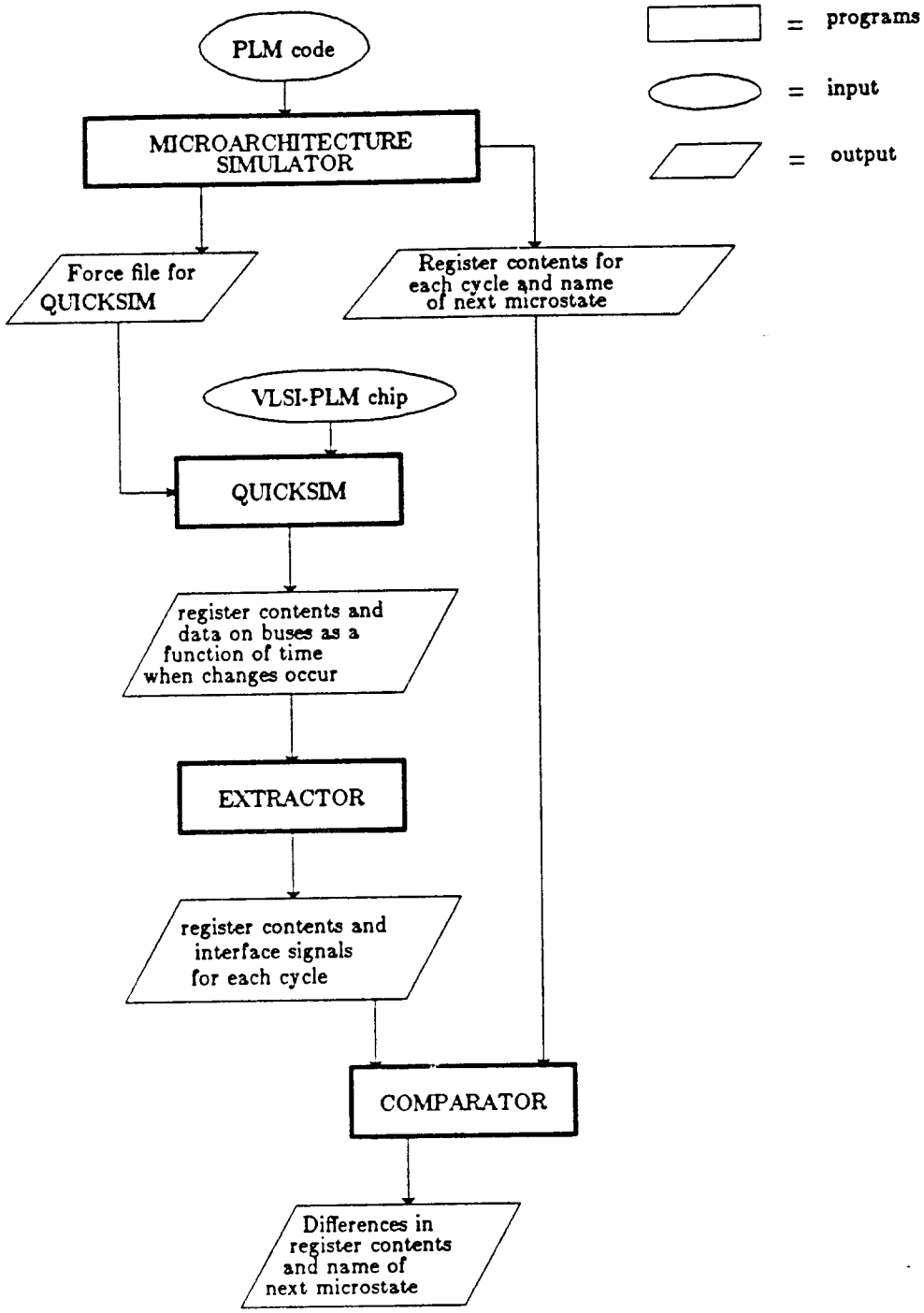


FIGURE 16 **DESIGN VERIFICATION SYSTEM**

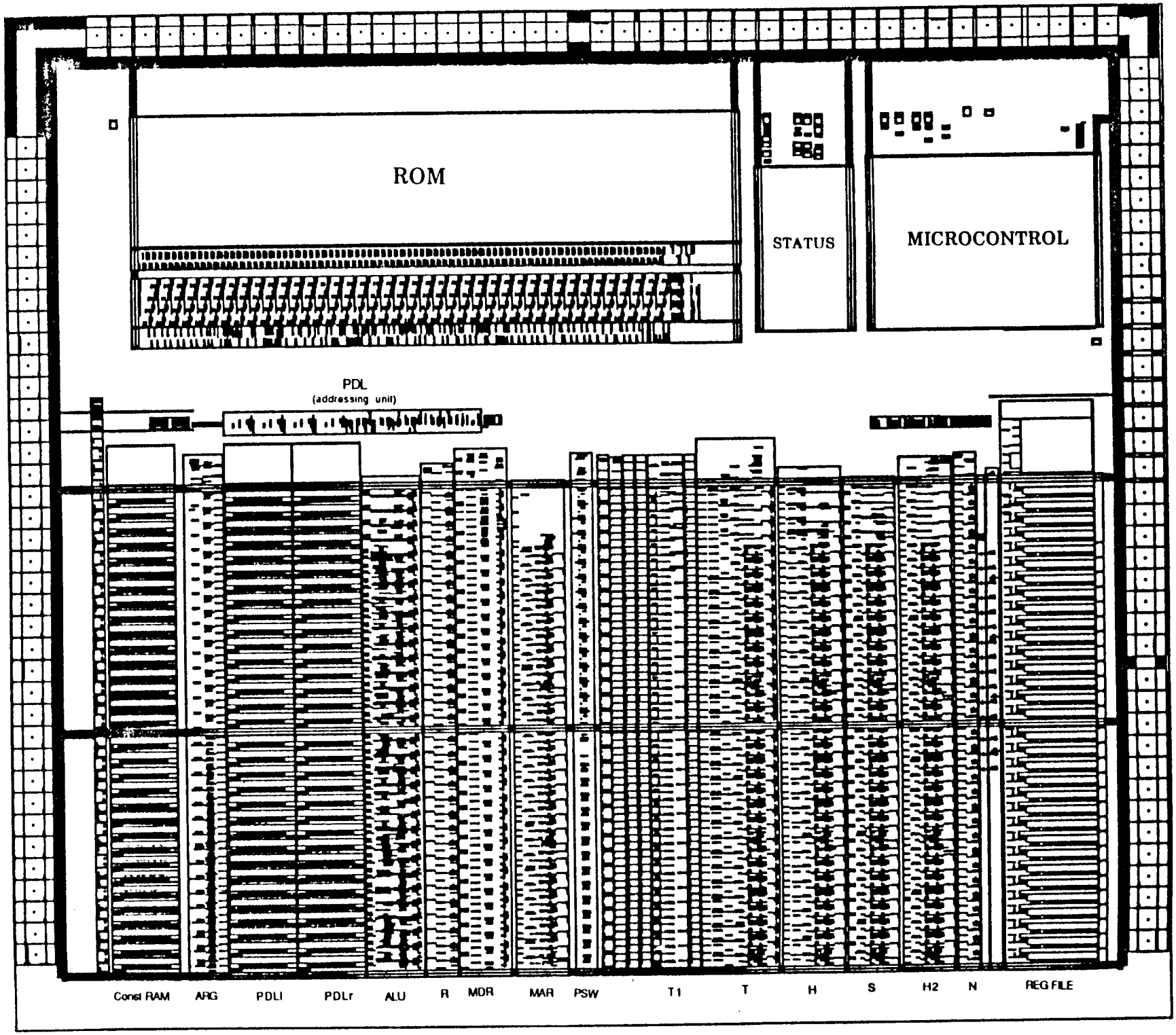


Figure 17 VLSI-PLM Block Placement

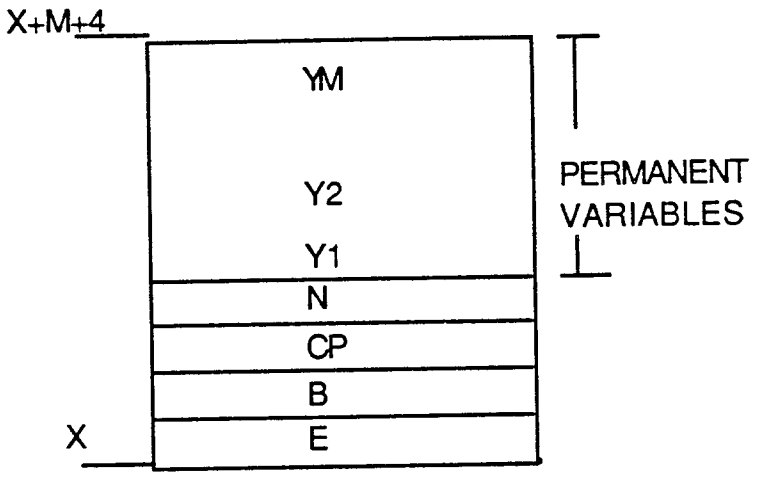


Figure 18 Environment Frame

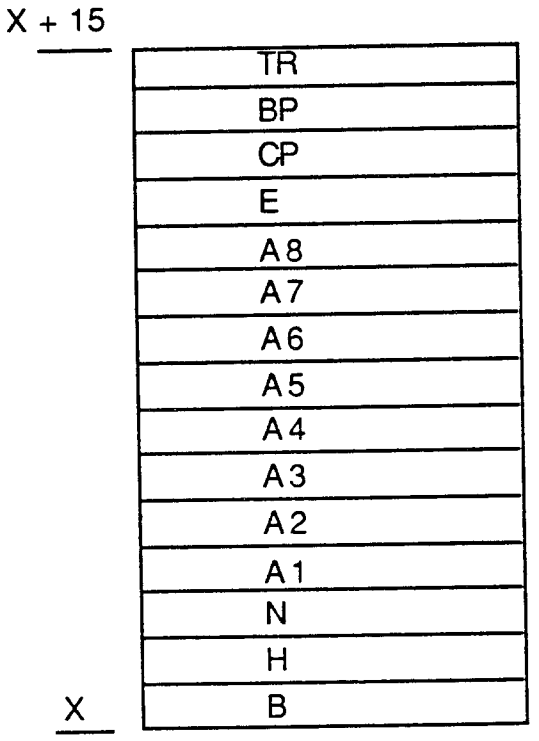
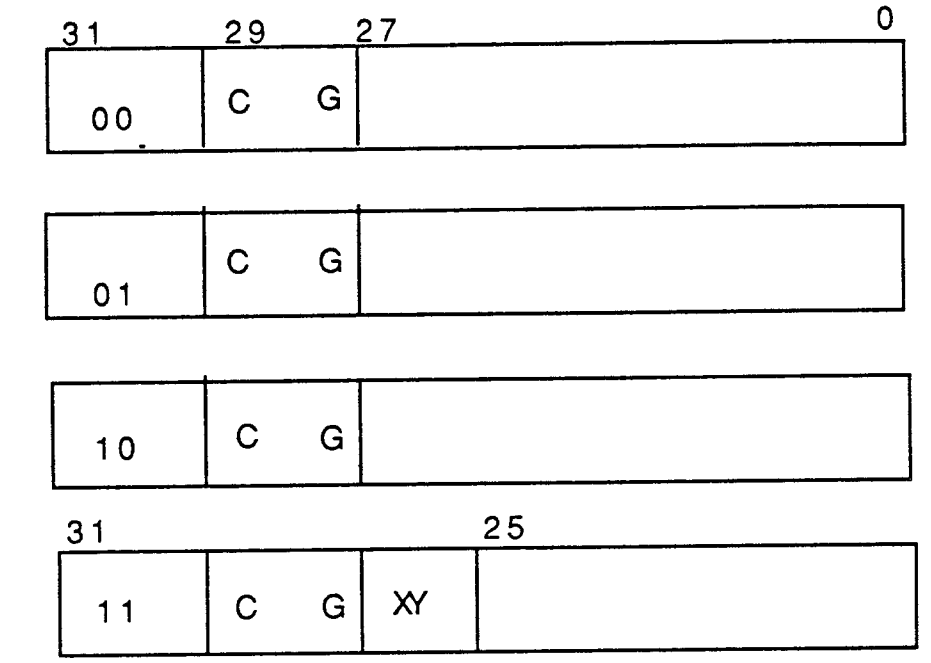


Figure 19 Choice Point Frame



C CDR bit

G Garbage bit

XY = 00 small integer
01 other numbers
10 atom
11 NIL

Figure 20 Data Representation in VLSI-PLM

APPENDIX 1

This appendix has six parts. The first part describes the contents of the environment frame, choice point frame, and the tagging scheme. The second part gives execution flow diagrams for each instruction implemented on the VLSI-PLM. The third part discusses the details of the microarchitecture level simulator. A block diagram of the simulator is included along with its description. The sheets of the entire data path are given in the fourth part. The busses used by each block and the control signals needed to operate the data path are also shown in the sheets. The fifth section contains the top level sheet of the microsequencer and the details of the micropage select circuit and the microprogramcounter circuit. The final part contains timing diagrams of the various chip interface signals.

1. STACK FRAMES AND TAGGING SCHEME

The stack contains both environments and choice points (see section 2). Figure 18 shows the structure of an environment frame. Besides permanent variables, environments also contain the values of certain registers which must be preserved across the execution of a Prolog clause. The following registers are saved in an environment frame (ordered from low to high memory):

E : location of last environment on stack
B : location of last choice point (and the current value of the cut bit)
CP : where to continue once this clause succeeds
N : number of permanent variables in last environment.

Choice point frames contain sufficient information to restore the state of a computation if a goal fails, and to indicate the next clause to try. Figure 19 shows the structure of a choice point frame. Choice points contain the following register values (ordered from low to high memory):

B : location of previous choice point
H : the current top of the heap
N : number of permanent variables in current environment
An : the contents of the argument registers (8 registers)
E : location of current environment on stack
CP : address of next clause to execute should this one succeed
BP : address of next clause to try should current goal fail
TR : the current top of trail.

The four data types of the VLSI-PLM are implemented as shown in Figure 20. Two primary tag bits identify the data type. The four basic data types are reference (variables), constant (atoms, integers, and other numerics), list, and structure. In addition to the primary tags, there is also a cdr-bit and a garbage collection bit. The cdr-bit allows compact representation of lists.

The constant data type also requires a secondary tag field to distinguish between small integers, atoms, and nil. The secondary field is not fixed by the hardware of the VLSI-PLM. The values given in the diagram are typical.

2. DESCRIPTION OF INSTRUCTIONS

The VLSI-PLM implements the PLM instruction set [1] along with support for external (host) processing. To lessen the performance impact that external procedures impose, a minimal set of general purpose instructions have also been included. These instructions are used to implement nearly all of the procedures that otherwise would require host processing.

Figure 21 shows the execution flow for each instruction. Execution flow of an instruction begins at the top of the diagram and exits at the bottom. Some instructions have a second entry point, indicated by the entry labeled with **need pf2** (pf2 is short for prefetch2). This alternate entry point is used whenever an instruction needs the second/third argument which has not yet been fetched.

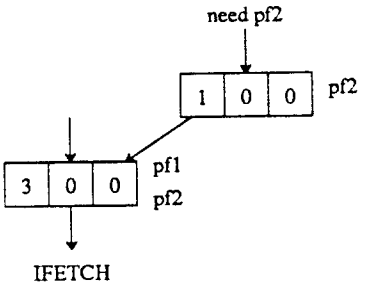
Each block in the flow diagrams represents a basic block in the microcode. The left-most number in each block gives the number of microstates in that block (which is equal to the execution cycles given no

memory read/write delays). The middle number is the number of memory reads performed in that block, and the right-most number is the number of memory writes. To the right of some blocks are notes indicating operations done by that block. **Pf1, pf2, newp1,** and **newp2** indicate opcode prefetch, argument two/three prefetch, update PC (program counter in prefetch unit), and add offset to PC, respectively. Microbranches are also labeled to specify the conditions true for a given path.

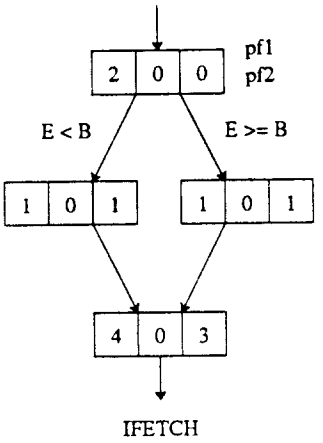
A complete description of each instruction will not be given here. For more information see [1].

1. B. Fagin and T. Dobry, "The Berkeley PLM Instruction Set: An Instruction Set for Prolog," *UCB Research Report*, CS Division, University of California, Berkeley, (September 1985).

add



allocate



and

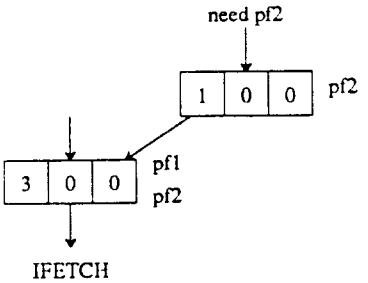
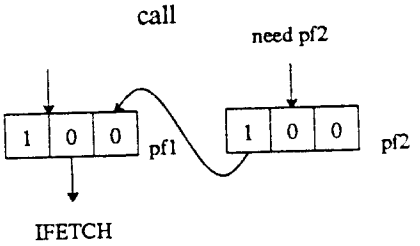
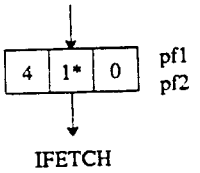


Figure 21 State Diagrams for the Instructions

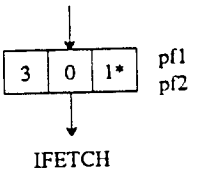
call



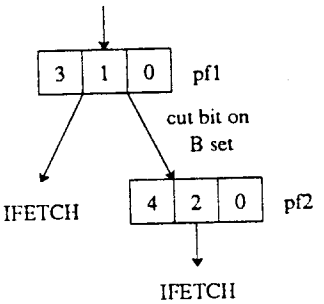
coderead



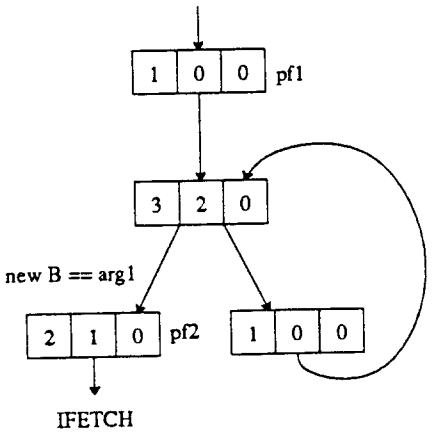
codewrite



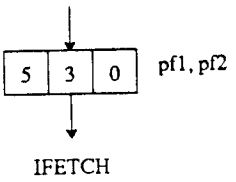
cut



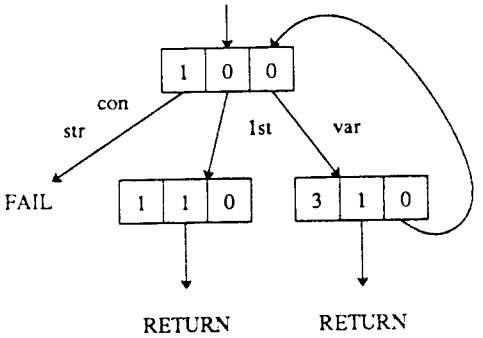
cutd



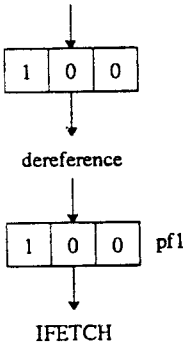
deallocate



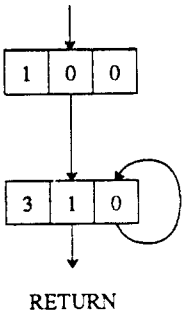
decdr



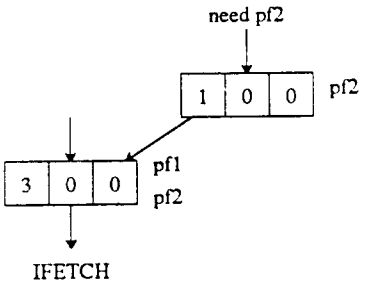
deref



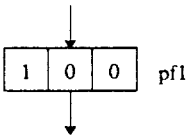
dereference

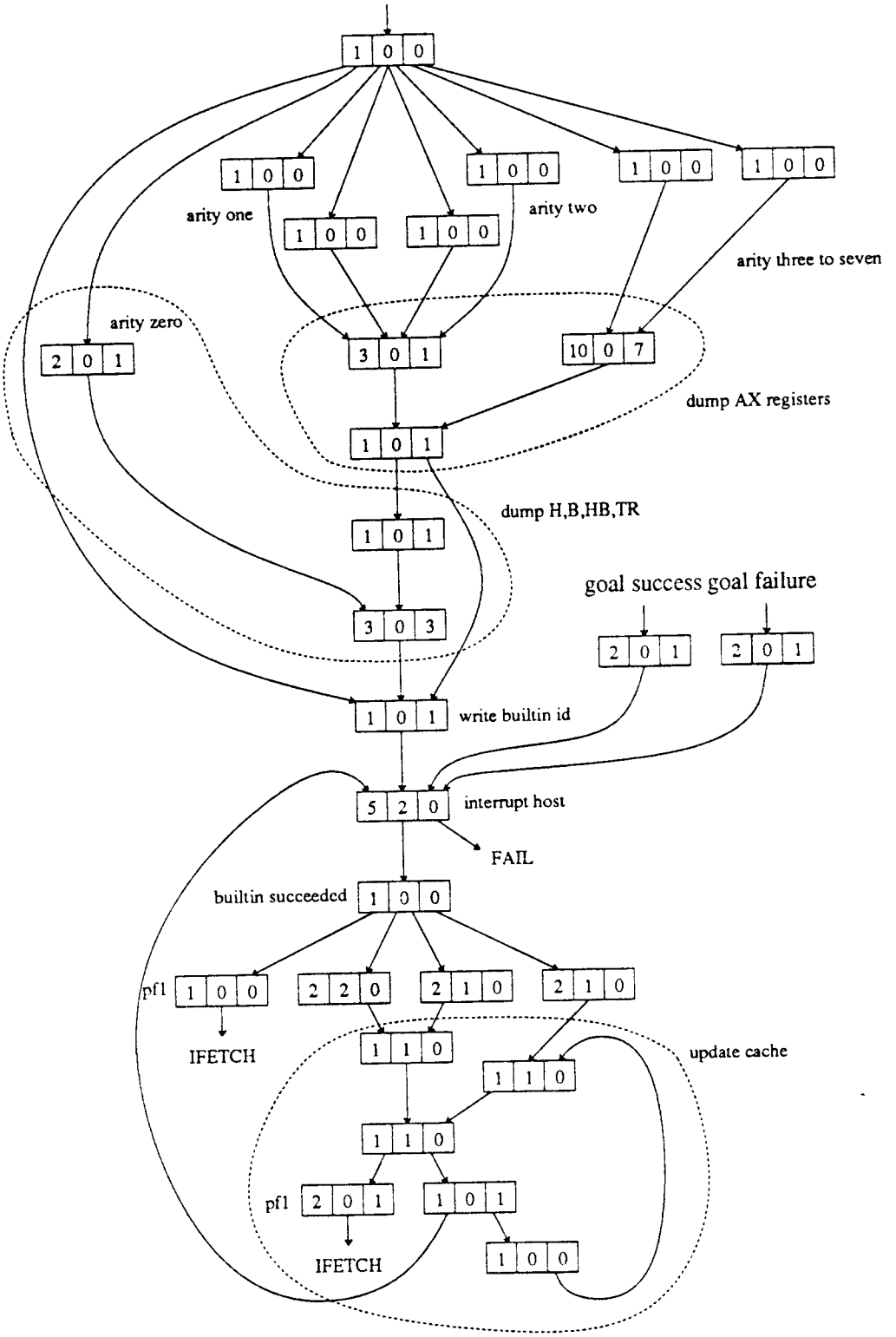


eor

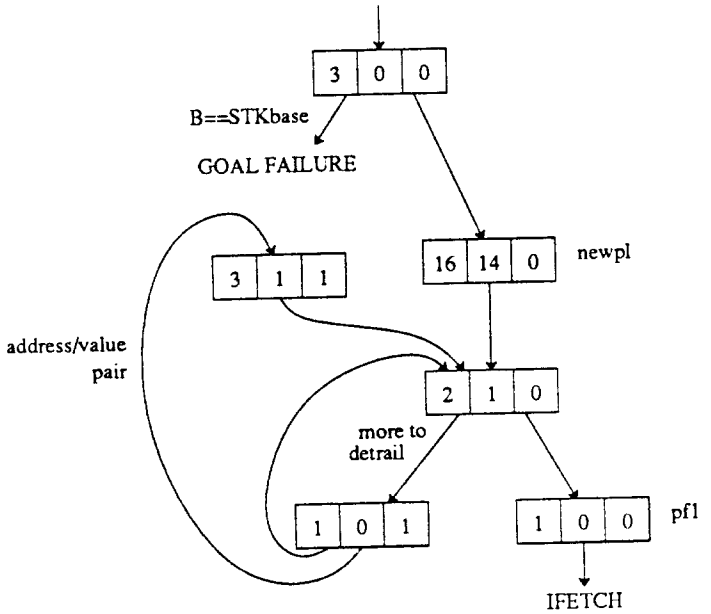


execute

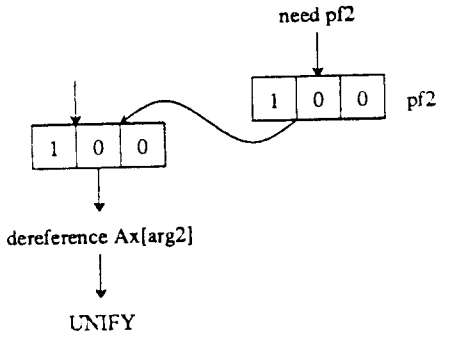



external escape
1 0 0
1 0 0
arity one
1 0 0
1 0 0
1 0 0
arity two
1 0 0
1 0 0
arity three to seven
arity zero
2 0 1
3 0 1
10 0 7
dump AX registers
1 0 1
dump H,B,HB,TR
1 0 1
goal success goal failure
3 0 3
2 0 1
2 0 1
1 0 1
write builtin id
5 2 0
interrupt host
FAIL
builtin succeeded
1 0 0
pf1
1 0 0
2 2 0
2 1 0
2 1 0
IFETCH
1 1 0
update cache
1 1 0
1 1 0
pf1
2 0 1
1 0 1
IFETCH
1 0 0

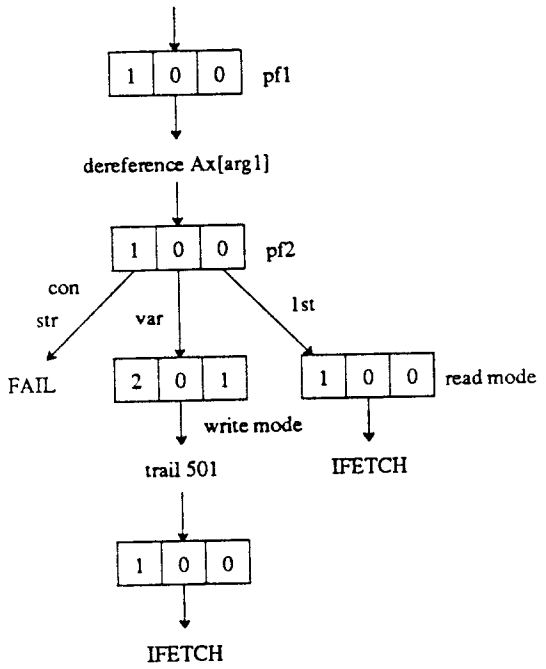
fail



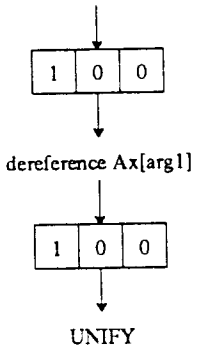
get_constant



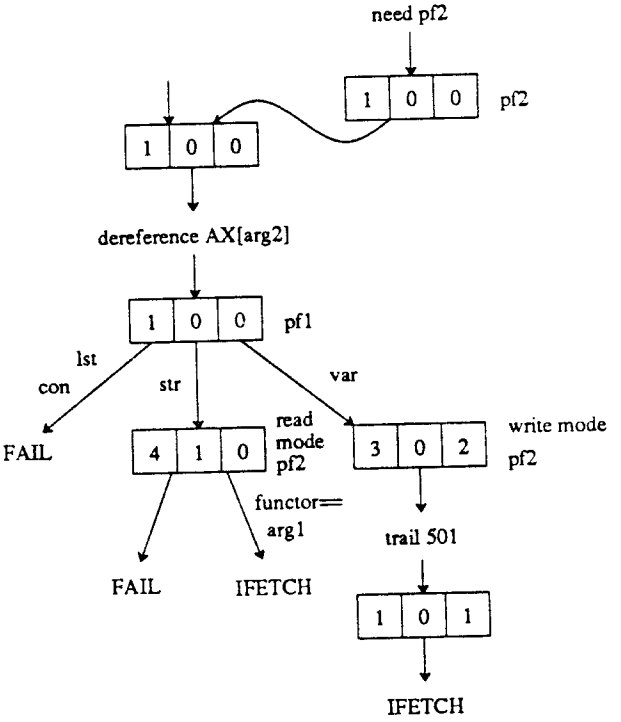
get_list



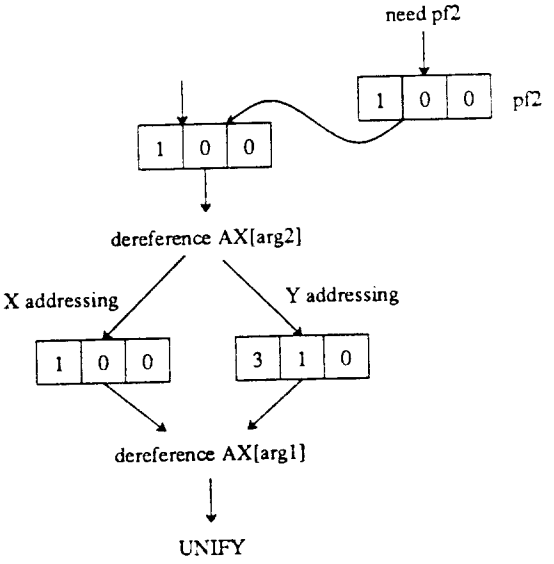
get_nil



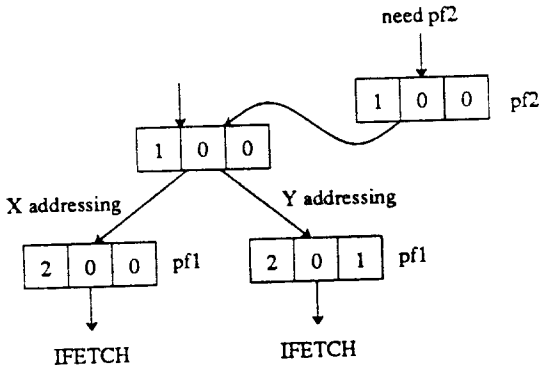
get_structure



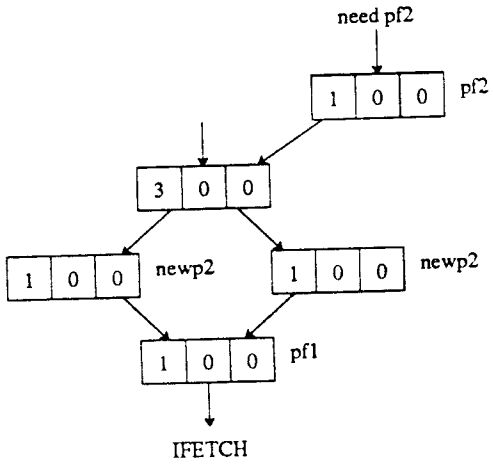
get_value



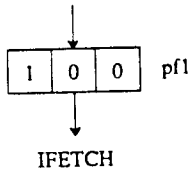
get_variable



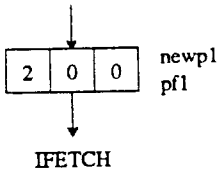
jeq/jlt/jle



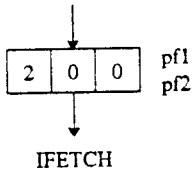
jump



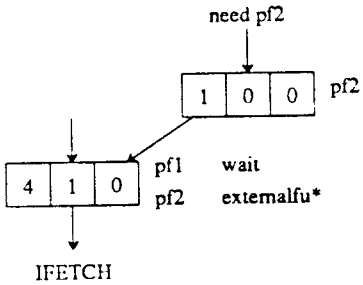
jumpxn



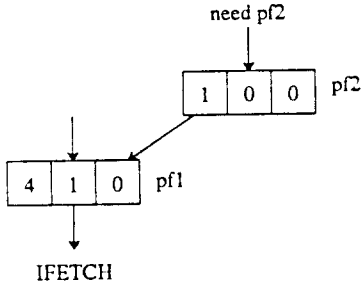
loadn



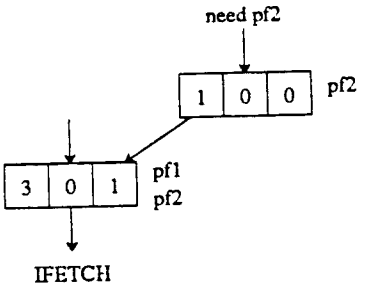
lock



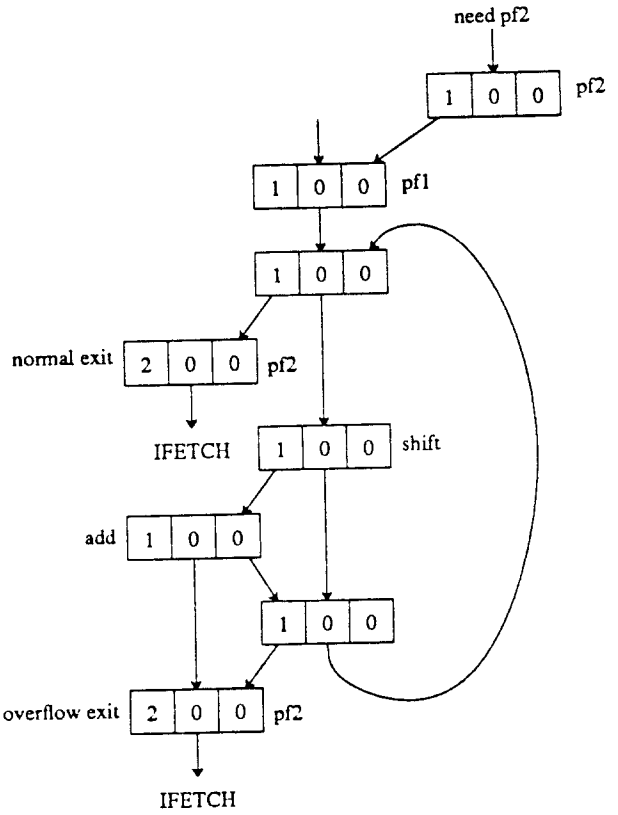
memread



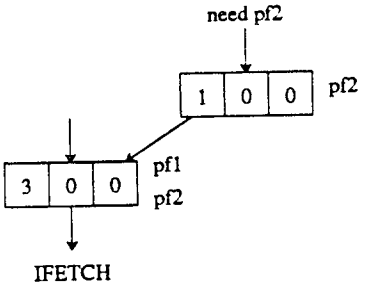
memwrite



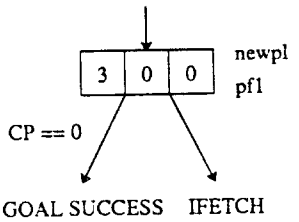
mult



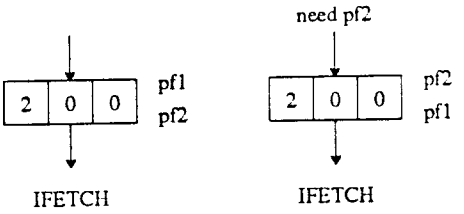
or



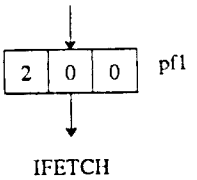
proceed



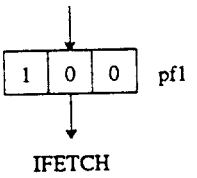
put_constant



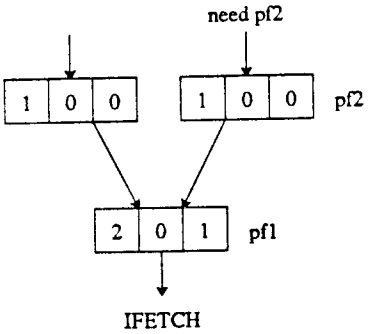
put_list



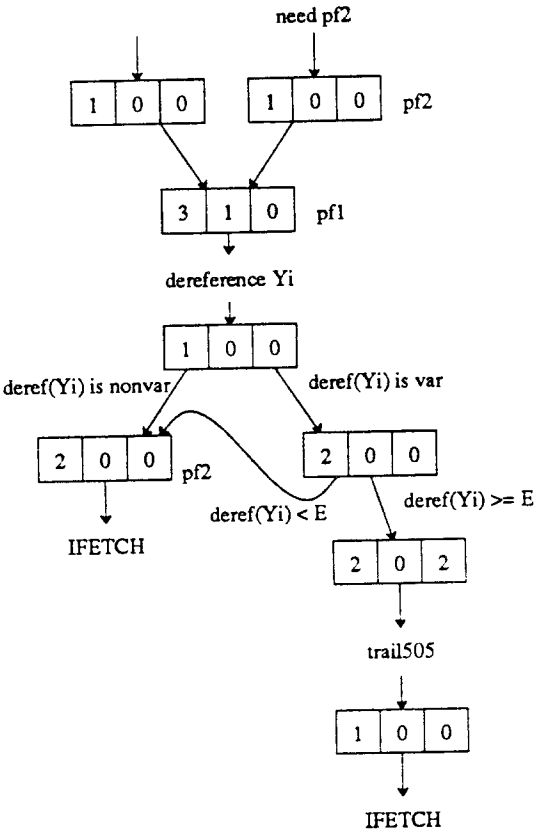
put_nil



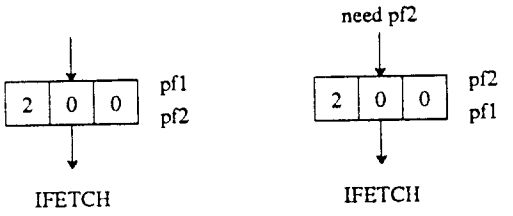
put_structure



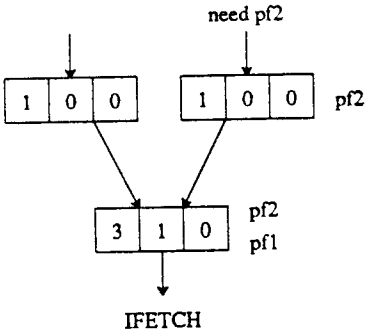
put_unsafe_value



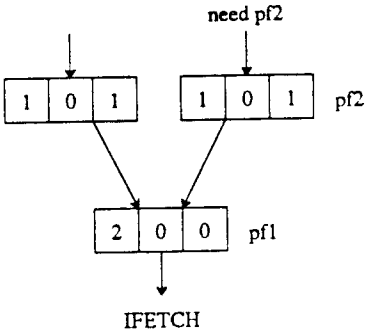
put_value X



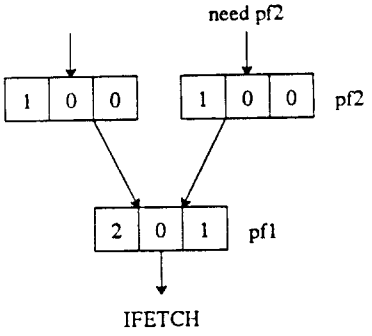
put_value Y



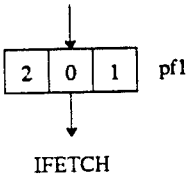
put_variable X



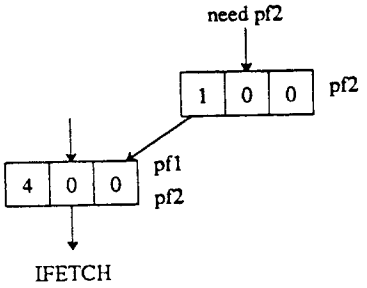
put_variable Y

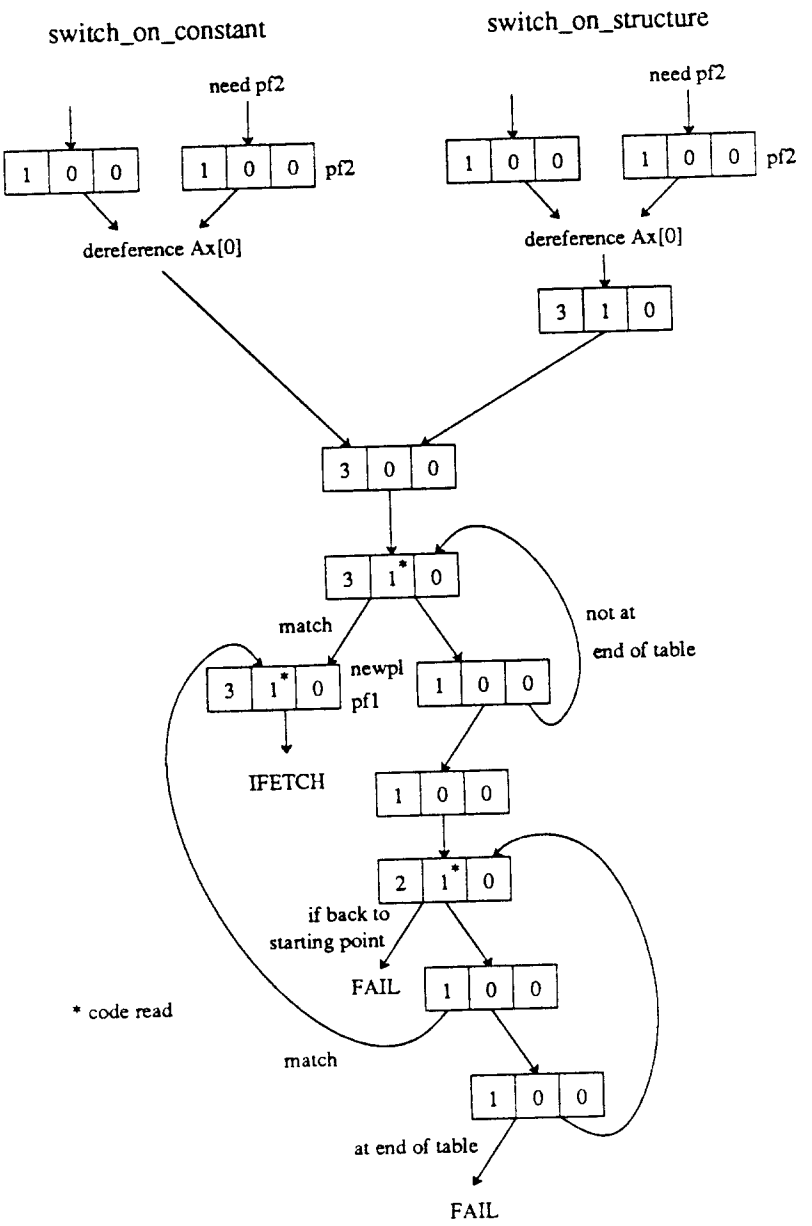


retry/retry_me_else

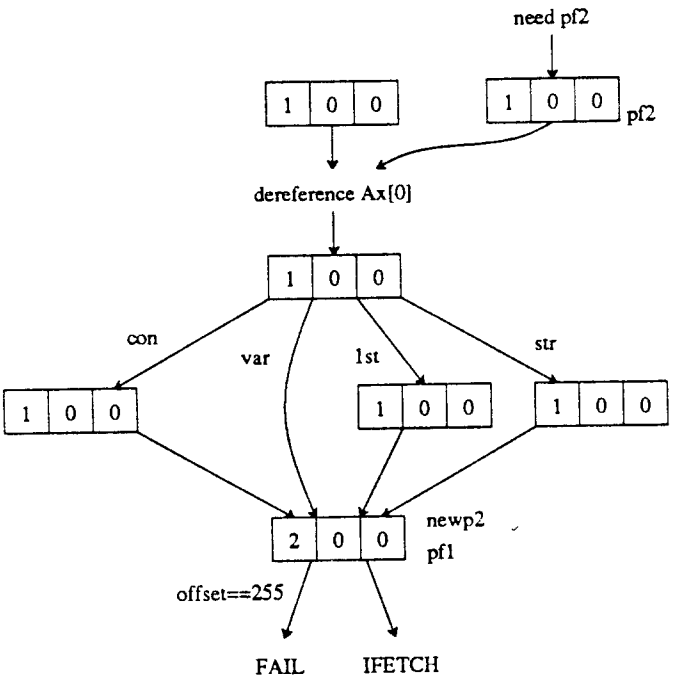


sub

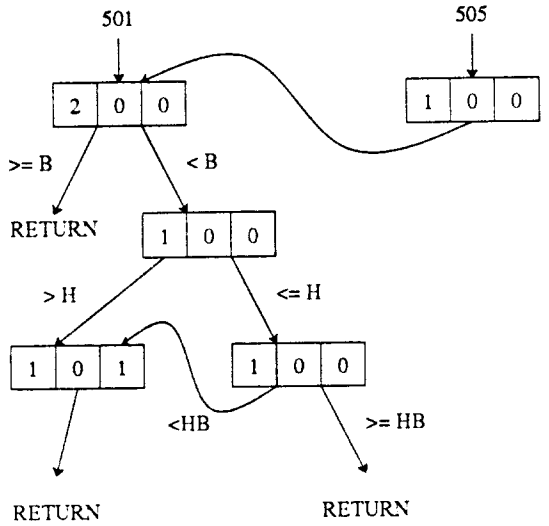



switch_on_constant
switch_on_structure
need pf2
need pf2
1 0 0
1 0 0 pf2
1 0 0
1 0 0 pf2
dereference Ax[0]
dereference Ax[0]
3 1 0
3 0 0
3 1* 0
match
not at
end of table
3 1* 0
newpl
pf1
1 0 0
IFETCH
1 0 0
2 1* 0
if back to
starting point
FAIL
1 0 0
* code read
match
1 0 0
at end of table
FAIL

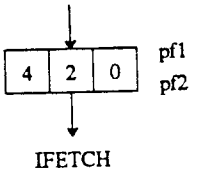
switch_on_term



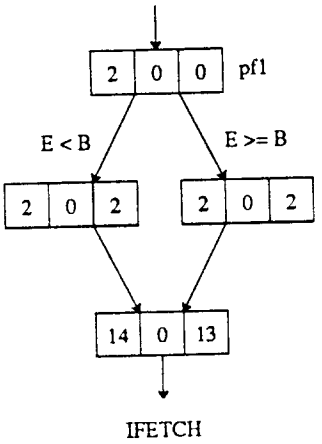
trail

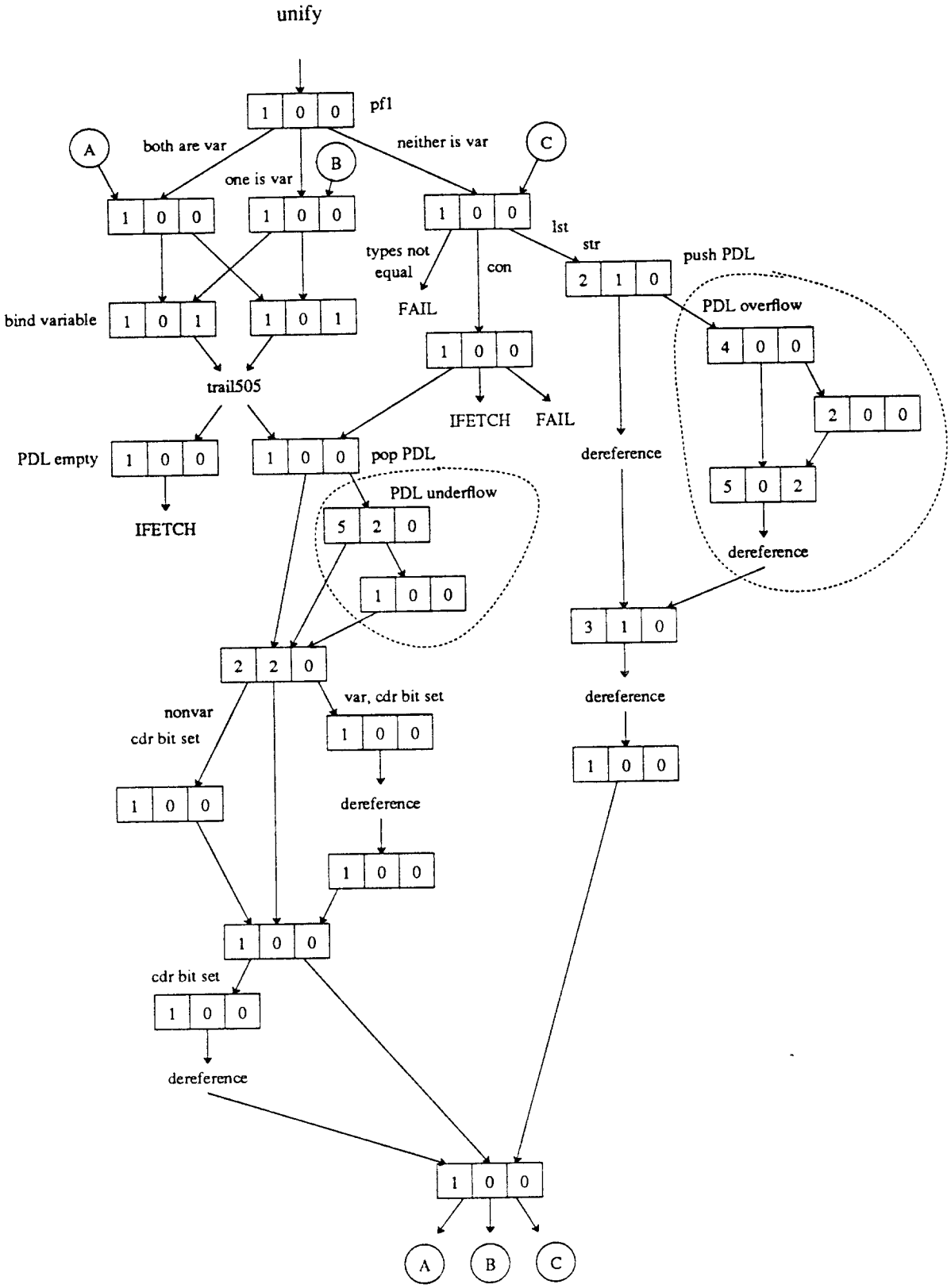


trust_me_else/trust

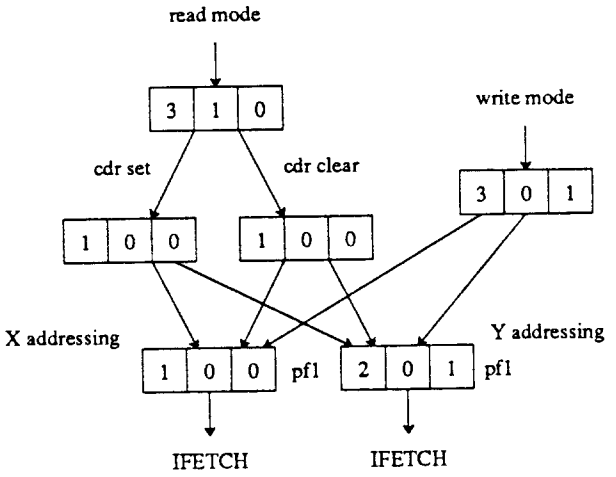


try/try_me_else

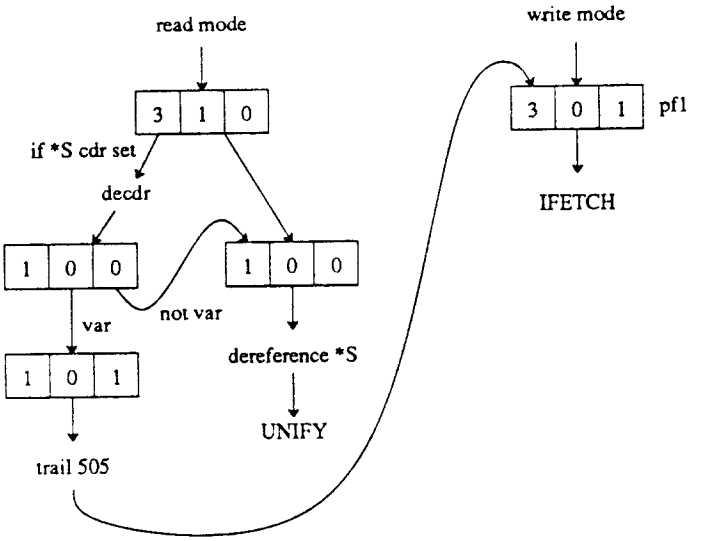



unify
1 0 0
pf1
A
both are var
B
one is var
neither is var
C
1 0 0
1 0 0
1 0 0
lst
str
2 1 0
push PDL
types not equal
con
FAIL
PDL overflow
bind variable
1 0 1
1 0 1
4 0 0
2 0 0
1 0 0
trail505
IFETCH
FAIL
5 0 2
dereference
PDL empty
1 0 0
1 0 0
pop PDL
dereference
IFETCH
PDL underflow
5 2 0
1 0 0
3 1 0
2 2 0
dereference
var, cdr bit set
nonvar
cdr bit set
1 0 0
1 0 0
dereference
1 0 0
1 0 0
cdr bit set
1 0 0
dereference
1 0 0
A
B
C

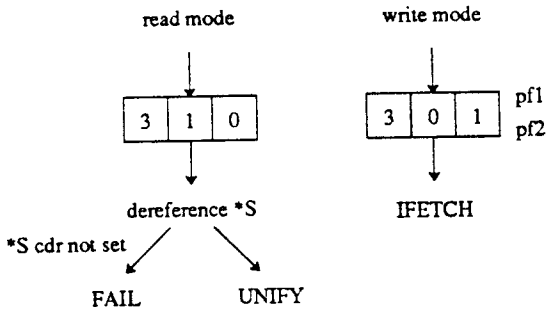
unify_cdr



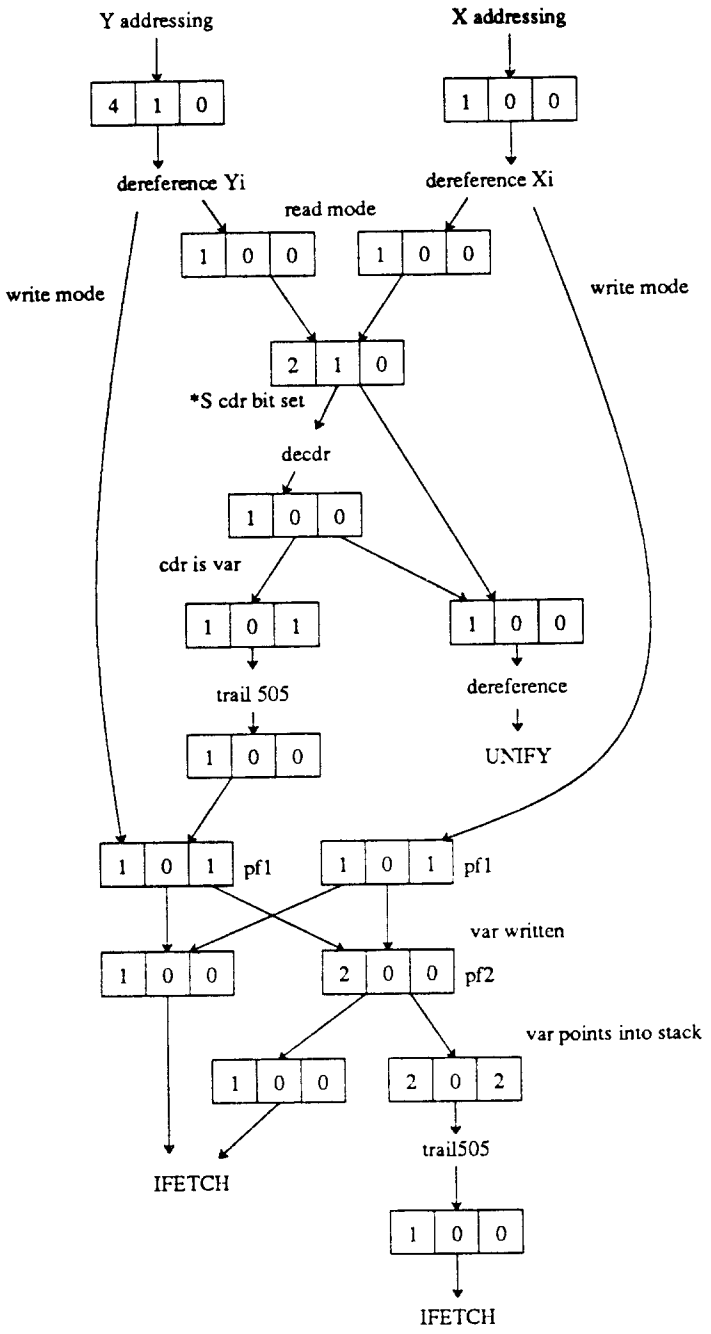
unify_constant



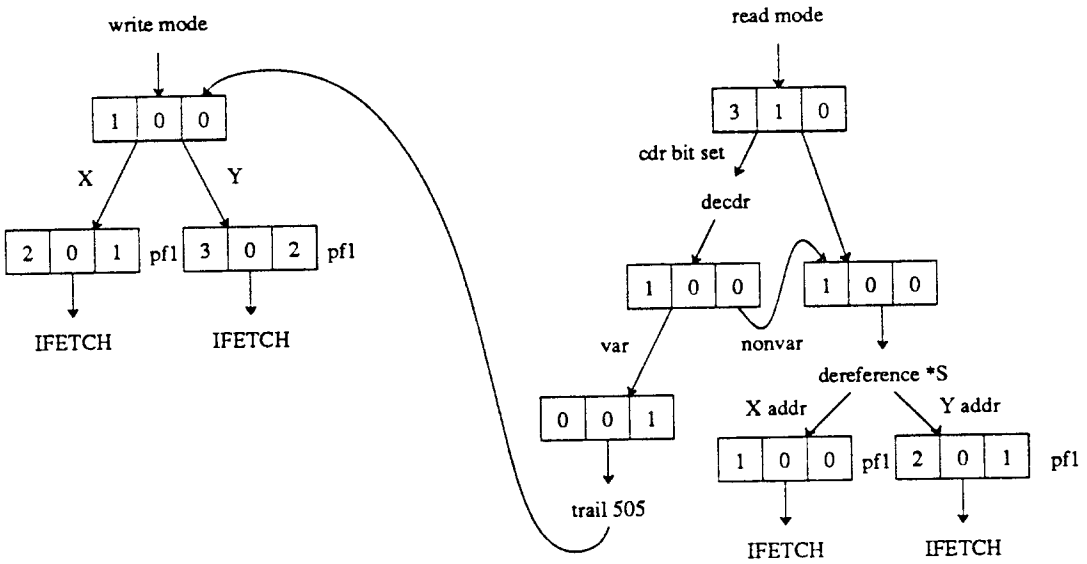
unify_nil


read mode
write mode
3
1
0
3
0
1
pf1
pf2
dereference *S
IFETCH
*S cdr not set
FAIL
UNIFY

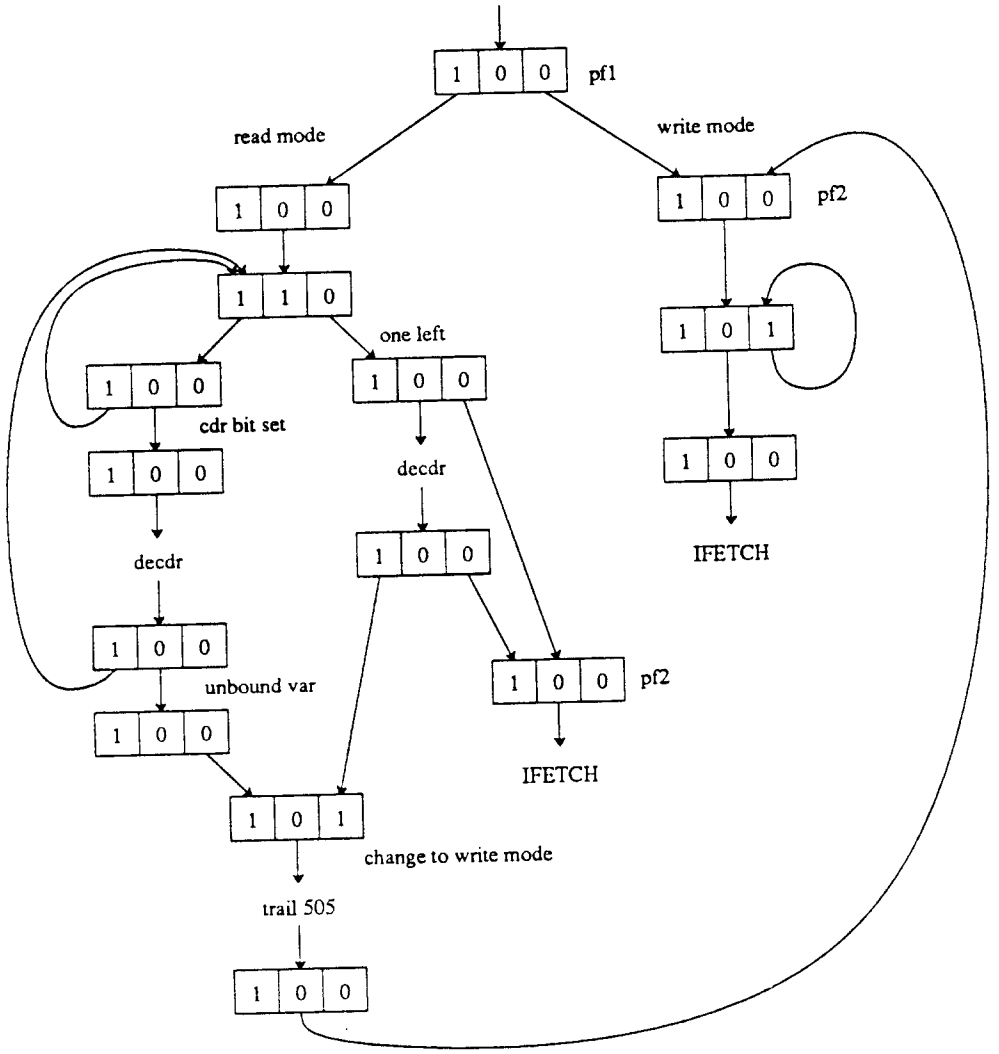
unify_value


Y addressing
X addressing
4 1 0
1 0 0
dereference Yi
dereference Xi
read mode
1 0 0
1 0 0
write mode
write mode
2 1 0
*S cdr bit set
decdr
1 0 0
cdr is var
1 0 1
1 0 0
trail 505
dereference
1 0 0
UNIFY
1 0 1 pf1
1 0 1 pf1
var written
1 0 0
2 0 0 pf2
var points into stack
1 0 0
2 0 2
trail505
IFETCH
1 0 0
IFETCH

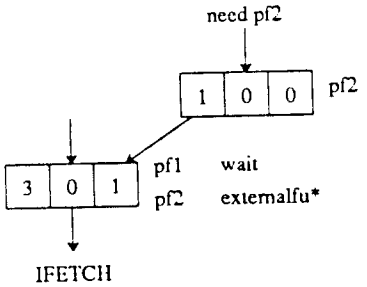
unify_variable



unify_void



unlock



3. REGISTER LEVEL SIMULATOR

A register level simulator was used for the debugging of the microcode and for producing input for the gate level simulator. The register level simulator is based directly on the microcode ROM bits. The ROM bits are compiled into C code which models the behavior of the chip (at the register transfer level). The result of compiling the ROM is a collection of C functions, one for each microstate, that is called by the main loop of the register level simulator (see Figure 22). The body of the simulator contains code to simulate memory, instruction prefetching, and external (host) processing. The simulator also contains code for statistics gathering, and debugging (single stepping, break pointing, etc.). A very useful feature of the register level simulator is the ability to produce Quicksim input. This allowed non-trivial programs to be used for gate level simulation.

A large number of benchmark programs were used for testing the microcode and the chip design. Many of the programs are standard Prolog benchmarks, others are programs developed as part of the Aquarius research project at Berkeley, and the remainder are small programs written specifically for exercising various states or branches in the microcode. Because of the large number of programs available, only a small number of them were used to generate input for the gate level simulator. This subset of the benchmarks was carefully chosen to exercise all parts of the datapath and microcode. The fact that the step-by-step values of the chip registers and interface pins given by both the gate level and register level simulators exactly match for this subset indicates that the register level simulator faithfully simulates the behavior of the datapath. All other benchmarks were run on the register level simulator and the results were compared with running these benchmarks on standard Prolog systems (C-prolog and Quintus Prolog).

The gate level simulator used was Quicksim (part of Mentor's IDEA system). This simulator requires input specifying the logical values of various points in the chip as a function of time. We were able to simulate the entire chip by specifying the values on input pins (clock, opcode, and memory data pins) as a function of time. To run benchmarks on this simulator, the register level simulator was used to produce the necessary input. An example of the Quicksim input is given in Figure 23. A program was used to filter the output of Quicksim and produced an output file giving step-by-step values of certain registers and output pins (see Figure 24). This output could then be directly compared with the corresponding output of the register level simulator.

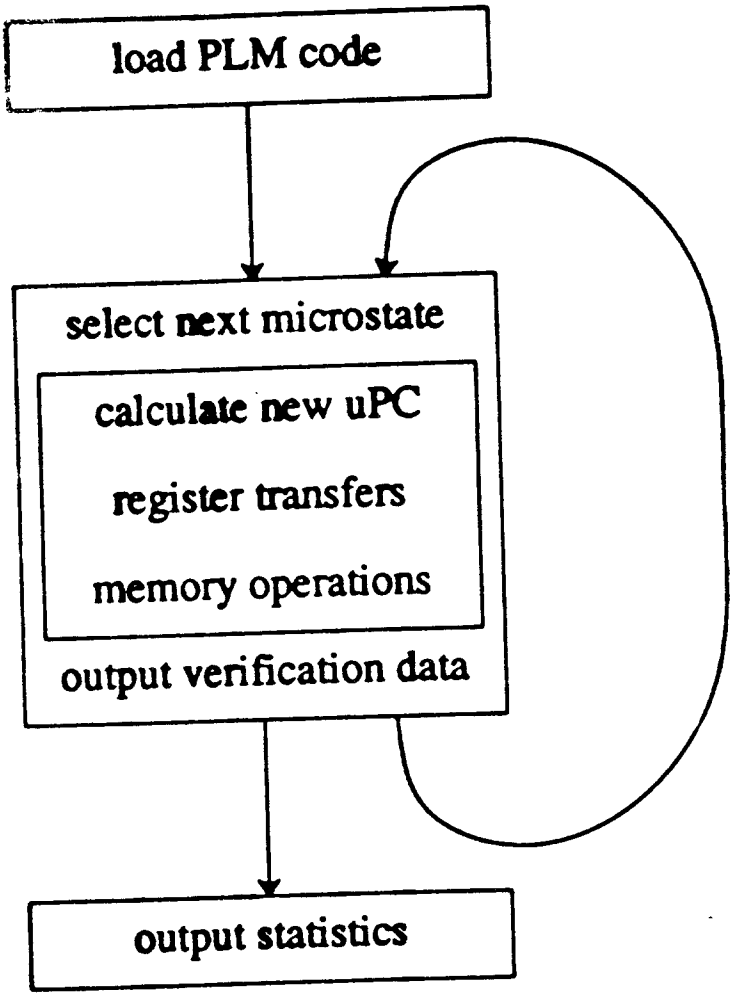


Figure 22 Block Diagram of the RTL Simulator

```
### 1 ###
# put_list X1
# boot00
run 100

### 2 ###
# put_list X1
# boot01
run 100

### 3 ###
# put_list X1
# init00
# memread
force MEMDAT 0ffffc00 65
run 100

### 4 ###
# put_list X1
# init01
# memread
force MEMDAT 0ffffc10 65
run 100

### 5 ###
# put_list X1
# init02
# memread
force MEMDAT 0fffffff 65
run 100

### 6 ###
# put_list X1
# init03
# memread
force MEMDAT 0ffffe00 65
run 100

### 7 ###
# put_list X1
# init04
# memread
force MEMDAT 00000001 65
run 100
```

Figure 23 Sample Input to Gate Level Simulator

```
### 8 ###
# put_list  X1
# init05
# memread
force MEMDAT 00000020 65
run 100

### 9 ###
# put_list  X1
# init06
# memread
force MEMDAT 00001000 65
run 100

### 10 ###
# put_list  X1
# init07
# memread
force MEMDAT cfffffff 65
run 100

### 11 ###
# put_list  X1
# init08
# memread
force MEMDAT 00040000 65
run 100

### 12 ###
# put_list  X1
# init09
# memread
force MEMDAT 00080000 65
run 100

### 13 ###
# put_list  X1
# init10
# memread
force MEMDAT 00000004 65
run 100

### 14 ###
# put_list  X1
# init11
```

```
# memread
force MEMDAT 0000000f 65
run 100

### 15 ###
# put_list  X1
# init12
# memread
force MEMDAT 00000002 65
run 100

### 16 ###
# put_list  X1
# init13
# memread
force MEMDAT 00000000 65
run 100

### 17 ###
# put_list  X1
# init14
# memread
force MEMDAT fffffffd 65
run 100

### 18 ###
# put_list  X1
# init15
# memread
force MEMDAT 000000ff 65
run 100

### 19 ###
# put_list  X1
# reset00
run 100

### 20 ###
# put_list  X1
# reset01
run 100

### 21 ###
# put_list  X1
# reset02
```

```
run 100

### 22 ###
# put_list  X1
# reset03
run 100

### 23 ###
# put_list  X1
# reset04
# prefetch(1)
force OPCODE 00000013 20
force MEMDAT 00000000 35
run 100

### 24 ###
# put_list  X1
# put_list00
run 100

### 25 ###
# put_list  X1
# unify_cdr03
# prefetch(1)
force OPCODE 00000040 20
force MEMDAT c80000b5 35
run 100

### 26 ###
# unify_constant  a
# unify_constant_write00
run 100

### 27 ###
# unify_constant  a
# unify_constant_write01
# prefetch(1)
force OPCODE 00000040 20
force MEMDAT c80000b6 35
run 100

### 28 ###
# unify_constant  a
# ifetch
run 100
```

```
### 29 ###
# unify_constant  b
# unify_constant_write00
run 100

### 30 ###
# unify_constant  b
# unify_constant_write01
# prefetch(1)
force OPCODE 00000002 20
force MEMDAT ffffffff 35
run 100

### 31 ###
# unify_constant  b
# ifetch
run 100

### 32 ###
# unify_nil
# unify_nil_write00
# prefetch(1)
force OPCODE 00000013 20
force MEMDAT 00000001 35
run 100

### 33 ###
# unify_nil
# unify_nil_write01
run 100

### 34 ###
# unify_nil
# ifetch
run 100

### 35 ###
# put_list  X2
# put_list00
run 100

### 36 ###
# put_list  X2
# unify_cdr03
# prefetch(1)
```

```
*** 1 ***
boot00
Memdat = 0ffffc00  MDR   = 0ffffc00  T     = fffffff
T1   = fffffff  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = fffffff  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 2 ***
boot01
Memdat = fffffff  MDR   = 0ffffc00  T     = fffffff
T1   = fffffff  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = 0ffffc00  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 3 ***
init00
memread
Memdat = 0ffffc00  MDR   = 0ffffc00  T     = 0ffffc00
T1   = 0ffffc00  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = 0ffffc01  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 4 ***
init01
memread
Memdat = 0ffffc10  MDR   = 0ffffc10  T     = 0ffffc00
T1   = 0ffffc00  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = 0ffffc02  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 5 ***
init02
memread
Memdat = 0fffffff  MDR   = 0fffffff  T     = 0ffffc00
T1   = 0ffffc00  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = 0ffffc03  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 6 ***
init03
memread
Memdat = 0ffffe00  MDR   = 0ffffe00  T     = 0ffffc00
T1   = 0ffffc00  R     = fffffff  H     = fffffff  S     = fffffff
MAR  = 0ffffc04  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1
```

Figure 24 Sample Output of Gate Level Simulator

```
*** 18 ***
init15
memread
Memdat = 000000ff  MDR   = 000000ff  T     = 0ffffc00
T1    = 0ffffc00  R     = fffffff  H     = fffffff  S     = fffffff
MAR   = 0ffffc10  N     = fffffff  Mode  =      2  CC    =      2
Arg1  = fffffff  Arg2_3 = -1

*** 19 ***
reset00
Memdat = fffffff  MDR   = 000000ff  T     = 0ffffc00
T1    = 0ffffc00  R     = 00000000  H     = fffffff  S     = fffffff
MAR   = 0ffffc10  N     = 00000000  Mode  =      2  CC    =      0
Arg1  = fffffff  Arg2_3 = -1

*** 20 ***
reset01
newp1
Memdat = 00000000  MDR   = 00000000  T     = 0ffffc00
T1    = 0ffffc00  R     = 00040000  H     = fffffff  S     = fffffff
MAR   = 0ffffc10  N     = 00000000  Mode  =      2  CC    =      0
Arg1  = fffffff  Arg2_3 = -1


*** 23 ***
reset04
pref1
instren
lastmi*
Memdat = 00000000  MDR   = 00000000  T     = 00040000
T1    = 00040000  R     = 00001000  H     = 00001000  S     = 00001000
MAR   = 00001000  N     = 00000000  Mode  =      0  CC    =      0
Arg1  = 00000000  Arg2_3 = -1

*** 24 ***
put_list00
Memdat = fffffff  MDR   = 00001000  T     = 00040000
T1    = 00040000  R     = 00001000  H     = 00001000  S     = 00001000
MAR   = 00001000  N     = 00000000  Mode  =      1  CC    =      2
Arg1  = 00000000  Arg2_3 = -1

*** 25 ***
unify_cdr03
pref1
instren
```

4. SHEETS OF THE DATA PATH

The data path is in several sheets with the control lines and power connected at the top and bottom. The buses of the data path are connected to the blocks on the sides. A list of microbits is also given. The microbits are grouped according to the blocks they control.

Mir	Mird	Control
Constant RAM:		
0	0	contobbus
1	1	padd0
2	2	padd1
3	3	padd2
4	4	padd3
5	5	memdattocon
Arg1:		
6	6	pref1
7	7	arg1torbus
8	8	arg1tomemdatbus
Arg23:		
9	9	pref2
10	10	arg2tobbus
11	11	arg3tobbus
PDL-left and right:		
12	12	pdlc0
13	13	pdlc1
14	14	pdlc2
15	15	ramwe
16	16	ramcs
ALU:		
17	17	s3
18	18	s2
19	19	s1
20	20	s0
21	21	m
22	22	cn
R:		
23	23	mdrbustor
24	24	alubustor
	25	ldr
25	26	rtobbus
26	27	rtorbus
27	28	rtomemdatbus
MDR:		
28	29	mdrbustomdr
29	30	alubustomdr
30	31	rbustomdr
31	32	t1bustomdr
32	33	tbustomdr
33	34	memdatbustomdr
34	35	ldmdr

35	36	ldmdrtag
36	37	mdrtagsel
37	38	mdrtag30
38	39	mdrtag31
39	40	otcdr
40	41	tcdr0
41	42	tcdr1
42	43	tcdr2
43	44	mdrtomdrbus
44	45	mdrtorbus
45	46	mdrtomemdatbus
46	47	mdrtobbus
MAR:		
47	48	alubustomar
48	49	rbustomar
49	50	t1bustomar
50	51	tbustomar
51	52	ldmar
52	53	marcnten
53	54	marup*
54	55	martomemdatbus*
55	56	diagnostics
Bus Connector:		
56	57	t1inbustomemdatbus
57	58	memdatbustot1inbus
T1:		
58	59	bbustot1
59	60	rbustot1
60	61	t1inbustot1
	62	ldt1
61	63	npasst1
62	64	numvalt1
63	65	t1tobbus
64	66	t1toabus
T:		
65	67	mdrbustot
66	68	bbustot
67	69	rbustot
68	70	tinbustot
69	71	t1inbustot
	72	ldt
70	73	tcnten
71	74	tup*
72	75	npasst
73	76	numvalt
74	77	ttomdrbus*
75	78	ttobbus*
76	79	ttoabus*
H:		
77	80	mdrbustoh
78	81	rbustoh

79	82	tinbustoh
	83	ldh
80	84	hcnten
81	85	hup*
82	86	htot1inbus*
83	87	htorbus*
84	88	htotinbus*
S:		
85	89	mdrbustos
86	90	bbustos
	91	lds
87	92	scnten
88	93	sup*
89	94	stomdrbus*
90	95	stot1inbus*
H2:		
91	96	mdrbustoh2
92	97	rbustoh2
93	98	t1inbustoh2
	99	ldh2
94	100	h2cnten
95	101	h2up*
96	102	h2totinbus*
97	103	h2tot1inbus*
N:		
98	104	mdrbuston
99	105	bbuston
	106	ldn
100	107	ntotinbus
101	108	ntomdrbus
Register File:		
102	109	mdrbustoregin
103	110	rbustoregin
104	111	bbustoregin
105	112	ldreg
106	113	adsela0
107	114	adsela1
108	115	adselb0
109	116	adselb1
110	117	regsela00
111	118	regsela01
112	119	regsela02
113	120	regselb00
114	121	regselb01
115	122	regselb02
116	123	regtotinbus
117	124	regtot1inbus
Collision Mux:		
118	125	setcollision
119	126	collision If zero then select T1INBUS else TINBUS

Microsequencer:

120	127	uenctlr	Usually one, i.e. load curp.
121	128	ldurp	

Pselect:

122	129	pctl0
123	130	pctl1
124	131	pctl2
125	132	pctl3
126	133	unxad8
127	134	unxad7

Subroutine ROM:

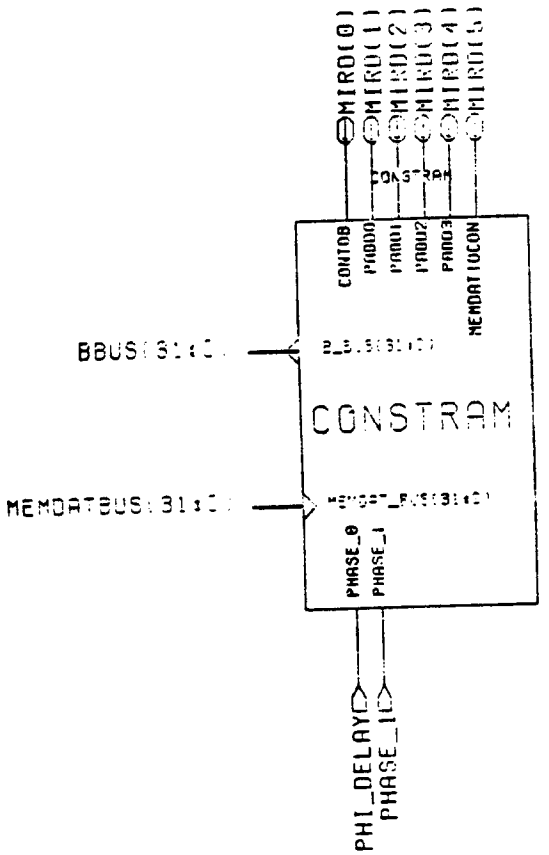
128	135	forcead0
129	136	forcead1
130	137	forcead2
131	138	unxad6
132	139	unxad5
133	140	unxad4
134	141	unxad3
135	142	unxad2
136	143	unxad1
137	144	unxad0

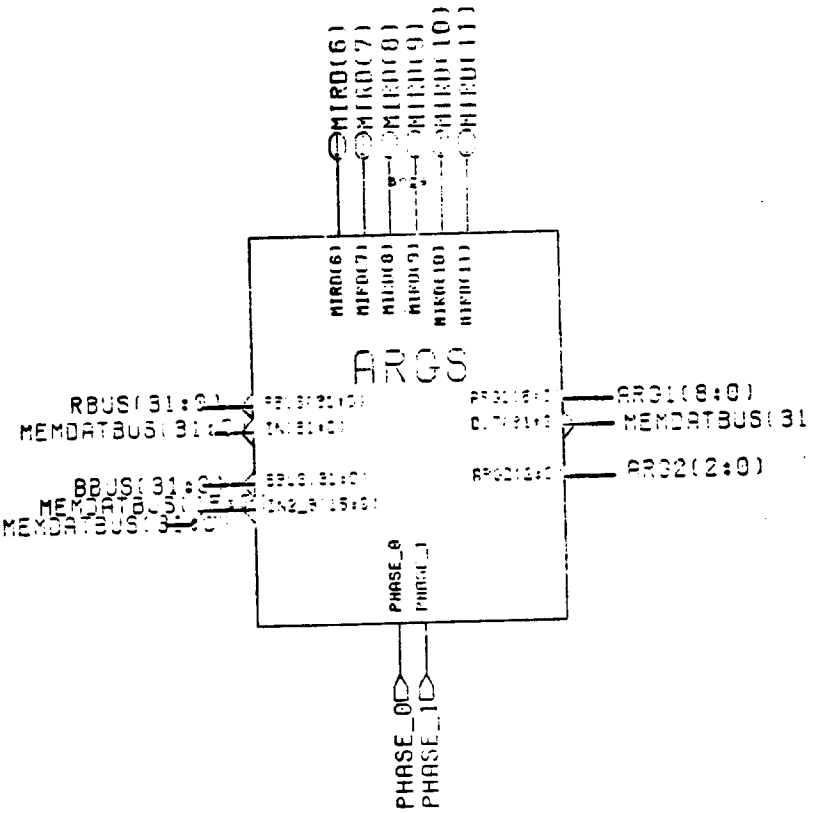
Microprogramcounter Select:

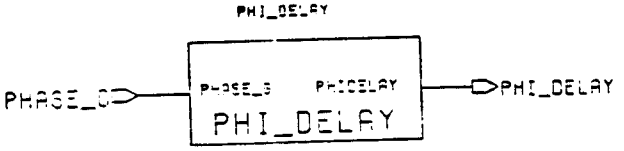
138	145	mctl4	
139	146	mctl3	
140	147	mctl2	
141	148	mctl1	
142	149	mctl0	
143	150	subrmux	If zero then urp else curp
144	151	uldarg	
145	152	ldmode	
146	153	mode	
147	154	ldcutm	
148	155	cut	

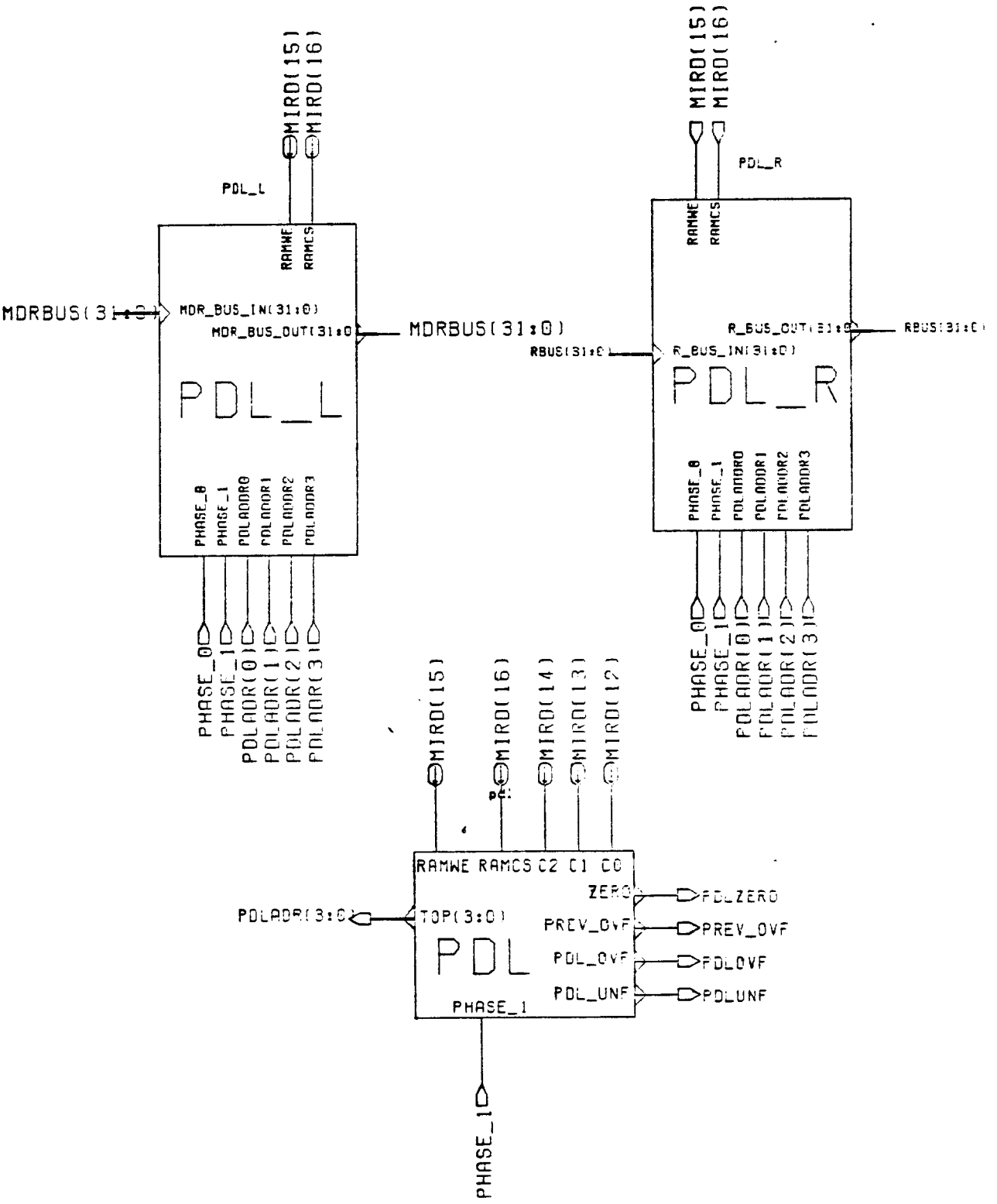
Interface Signals:

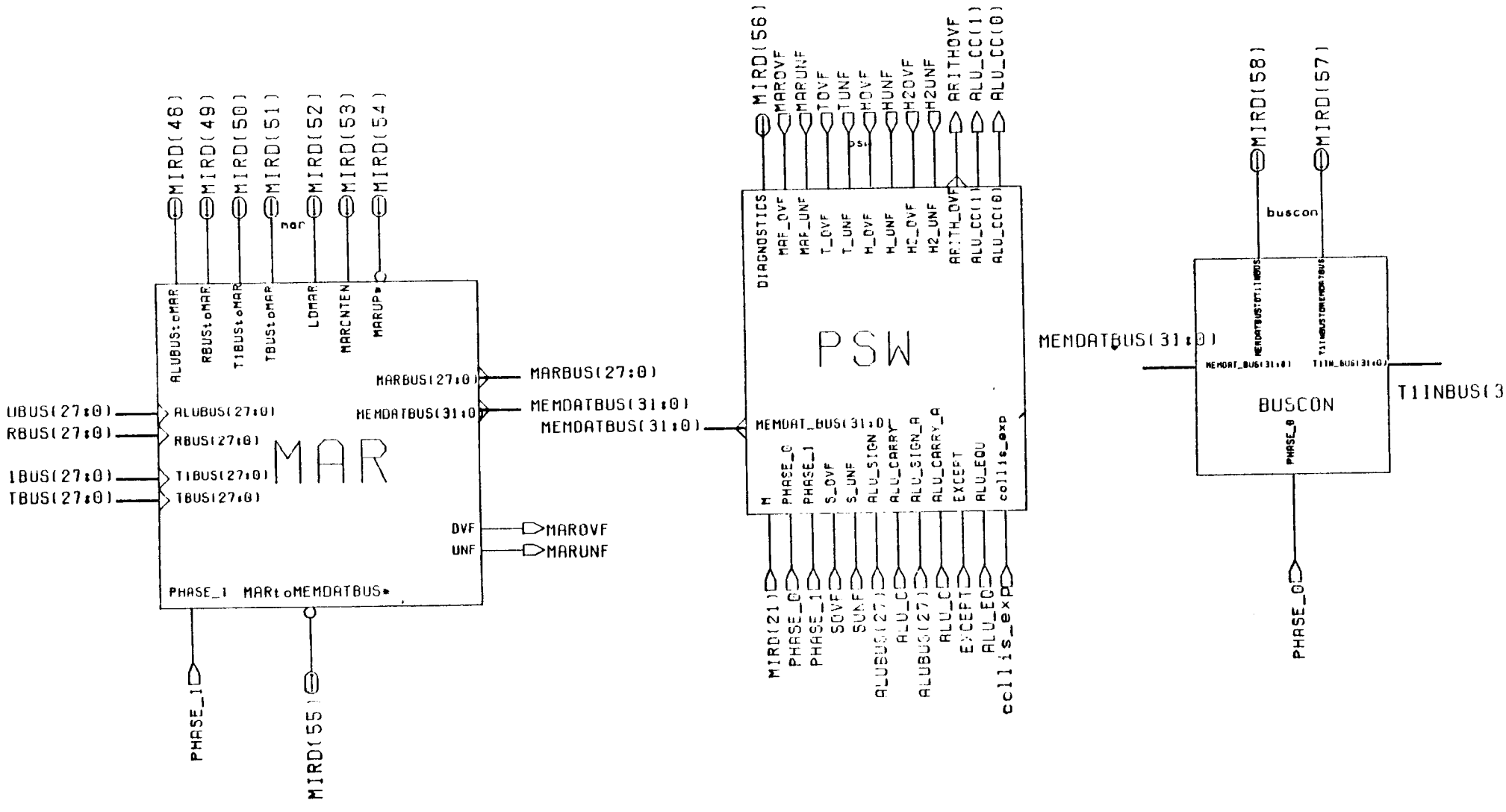
149	156	newp1*
150	157	newp2*
151	158	wait
152	159	dspace
153	160	lastmi*
154	161	fail*
155	162	memread*
156	163	memwrite*
157	164	externalfu*
158	165	ttot1inbus*
159	166	parity

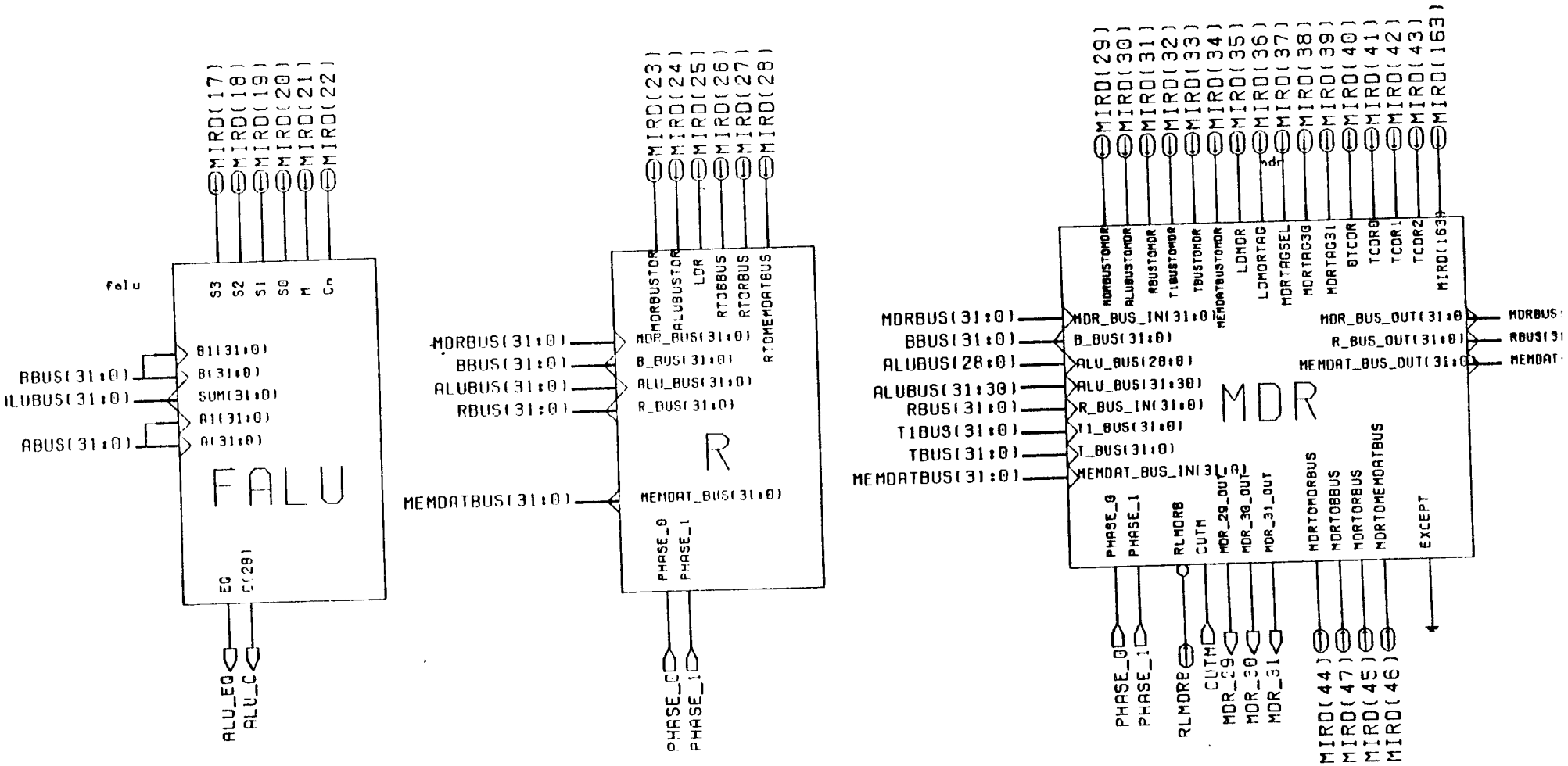

MIRD(0)
MIRD(1)
MIRD(2)
MIRD(3)
MIRD(4)
MIRD(5)
CONSTRAM
CONTOB
PADD0
PADD1
PADD2
PADD3
MEMDATTOCON
BBUS(31:0)
CONSTRAM
MEMDATBUS(31:0)
PHASE_0
PHASE_1
PHI_DELAY
PHASE_1

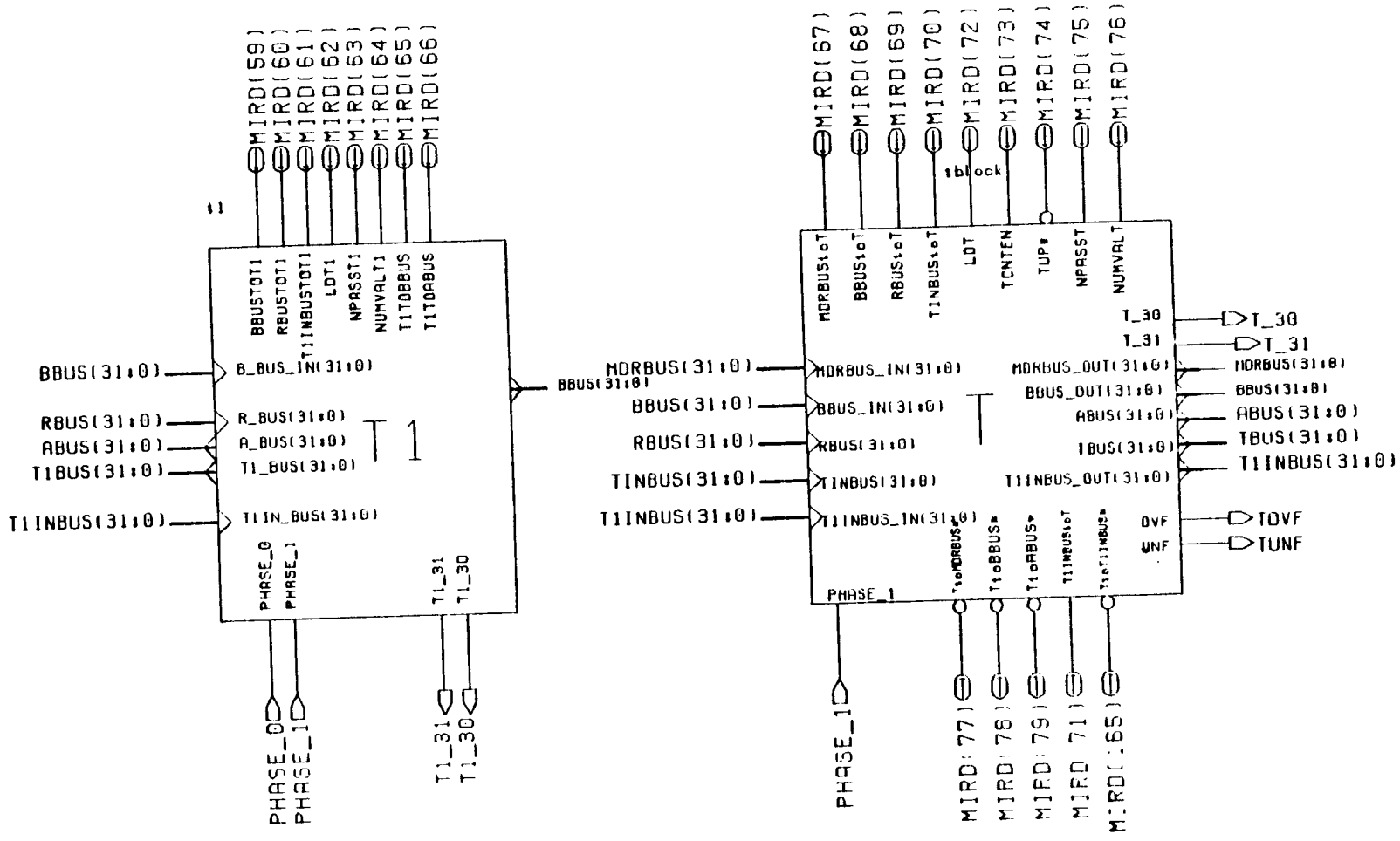

MIRD(6)
MIRD(7)
MIRD(8)
MIRD(9)
MIRD(10)
MIRD(11)
ARGS
RBUS(31:0)
MEMDATBUS(31:0)
BBUS(31:0)
ARG1(8:0)
MEMDATBUS(31
ARG2(2:0)
PHASE_0
PHASE_1
PHASE_0
PHASE_1

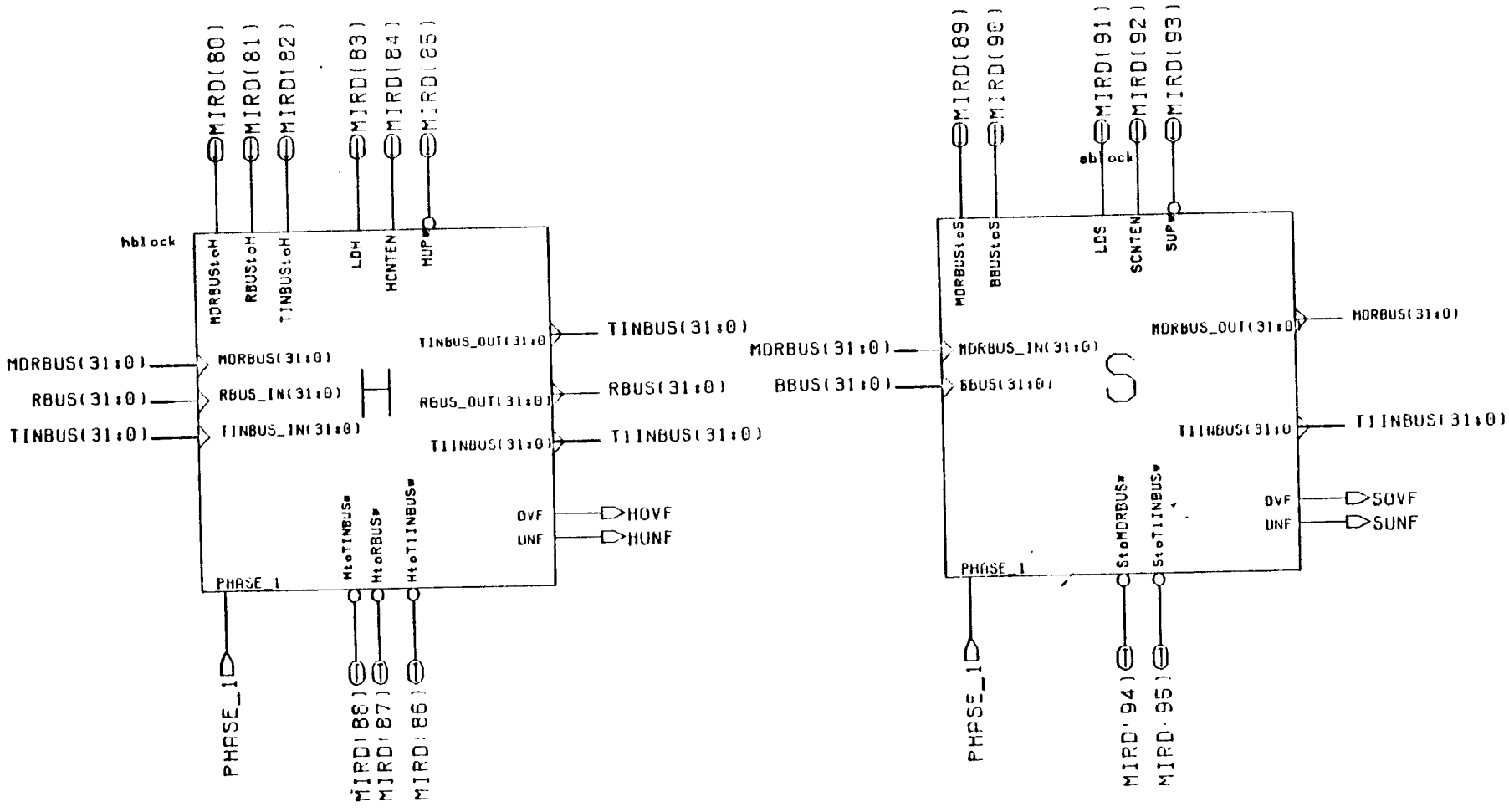

PHI_DELAY
PHASE_0
PHASE_0
PHIDELAY
PHI_DELAY
PHI_DELAY

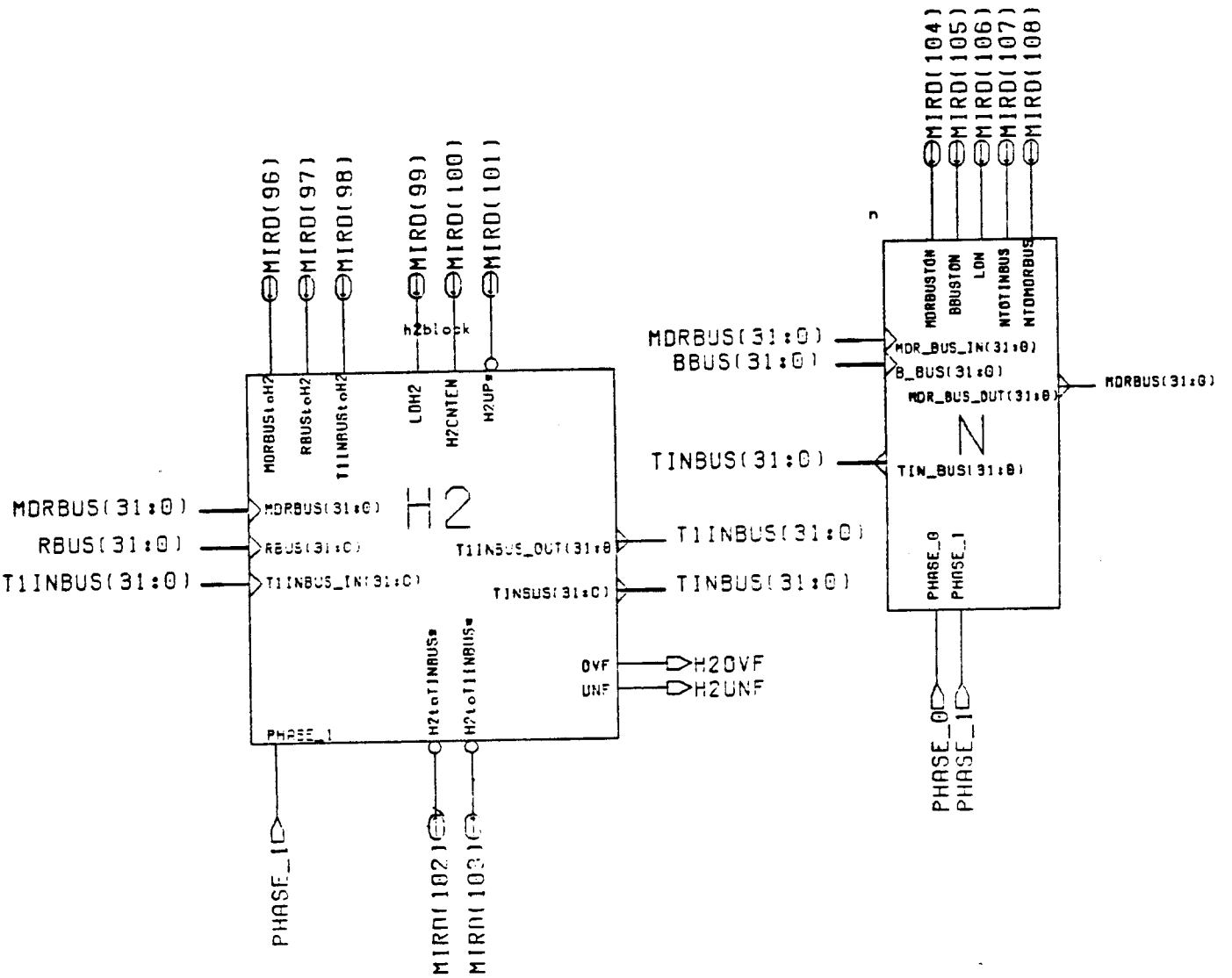

MIRD(15)
MIRD(16)
PDL_L
RAMWE
RAMCS
MDRBUS(31:0)
MDR_BUS_IN(31:0)
MDR_BUS_OUT(31:0)
PDL_L
PHASE_0
PHASE_1
PDLADR(0)
PDLADR(1)
PDLADR(2)
PDLADR(3)
MIRD(15)
MIRD(16)
PDL_R
RAMWE
RAMCS
RBUS(31:0)
R_BUS_IN(31:0)
R_BUS_OUT(31:0)
RBUS(31:0)
PDL_R
PHASE_0
PHASE_1
PDLADR(0)
PDLADR(1)
PDLADR(2)
PDLADR(3)
MIRD(15)
MIRD(16)
MIRD(14)
MIRD(13)
MIRD(12)
RAMWE RAMCS C2 C1 C0
PDLADR(3:0)
TOP(3:0)
PDL
PHASE_1
ZERO
PREV_OVF
PDL_OVF
PDL_UNF
PDLZERO
PREV_OVF
PDLOVF
PDLUNF
PHASE_1

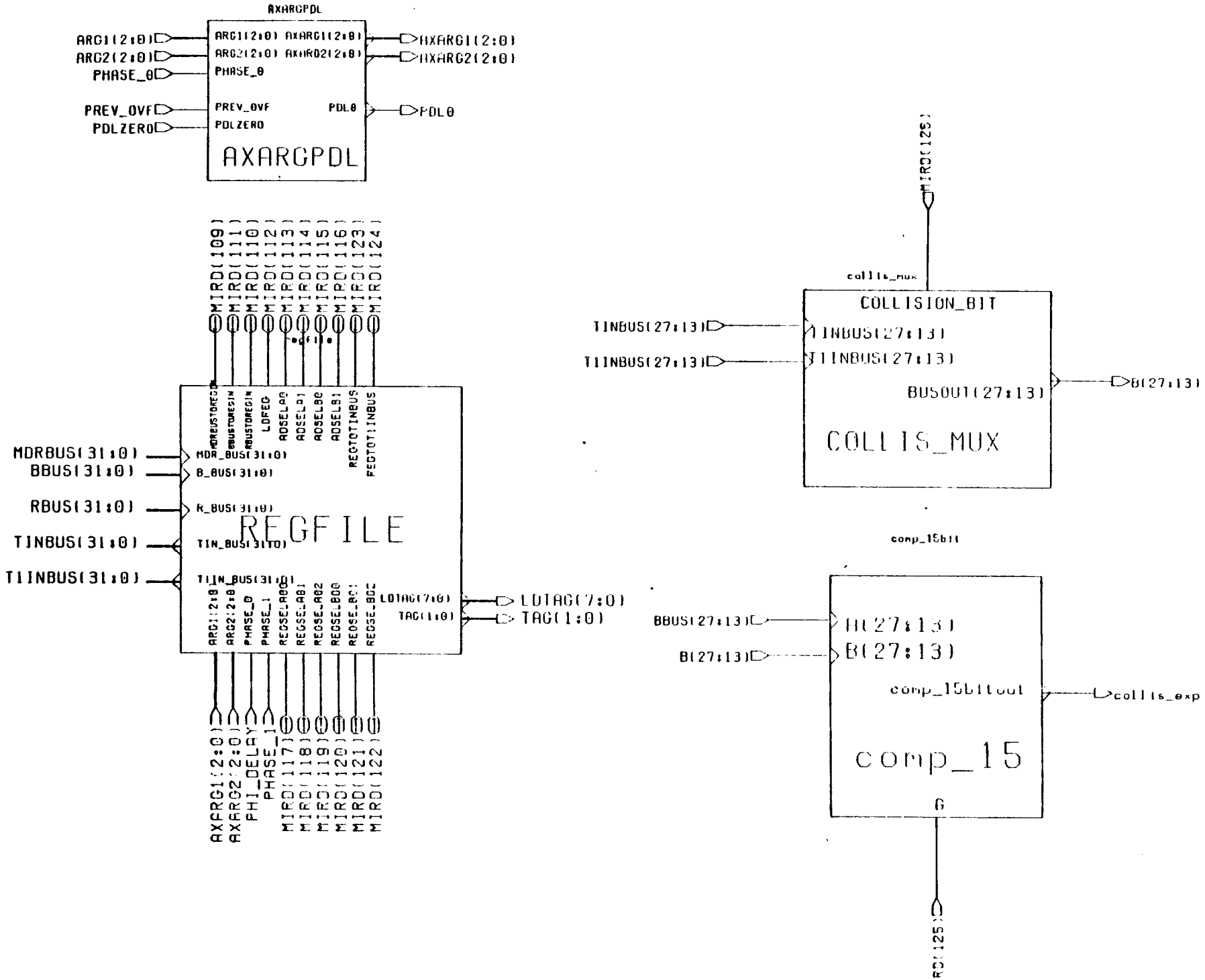

MIRD(48)
MIRD(49)
MIRD(50)
MIRD(51)
MIRD(52)
MIRD(53)
MIRD(54)
mar
ALUBUS:oMAR
RBUS:oMAR
T1BUS:oMAR
TBUS:oMAR
LOMAR
MARCNTEN
MARUP*
MARBUS(27:0)
MEMDATBUS(31:0)
UBUS(27:0)
RBUS(27:0)
1BUS(27:0)
TBUS(27:0)
ALUBUS(27:0)
RBUS(27:0)
T1BUS(27:0)
TBUS(27:0)
MAR
DVF
UNF
MAROVF
MARUNF
PHASE_1
MARtoMEMDATBUS*
PHASE_1
MIRD(55)
MIRD(56)
MAROVF
MARUNF
TOVF
TUNF
MOVF
MUNF
M2OVF
M2UNF
ARITHOVF
ALU_CC(1)
ALU_CC(0)
psw
DIAGNOSTICS
MAF_OVF
MAF_UNF
T_OVF
T_UNF
M_OVF
M_UNF
M2_OVF
M2_UNF
ARITH_OVF
ALU_CC(1)
ALU_CC(0)
PSW
MEMDAT_BUS(31:0)
MEMDATBUS(31:0)
M
PHASE_0
PHASE_1
S_OVF
S_UNF
ALU_SIGN
ALU_CARRY
ALU_SIGN_A
ALU_CARRY_A
EXCEPT
ALU_EQU
collis_exp
MIRD(21)
PHASE_0
PHASE_1
SOVF
SUNF
ALU_C
ALU_C
EXCEPT
ALU_EQU
collis_exp
MIRD(58)
MIRD(57)
buscon
MEMDATBUS(31:0)
MEMDAT_BUS(31:0)
T1INBUS(31:0)
BUSCON
PHASE_0
T1INBUS(3
PHASE_0

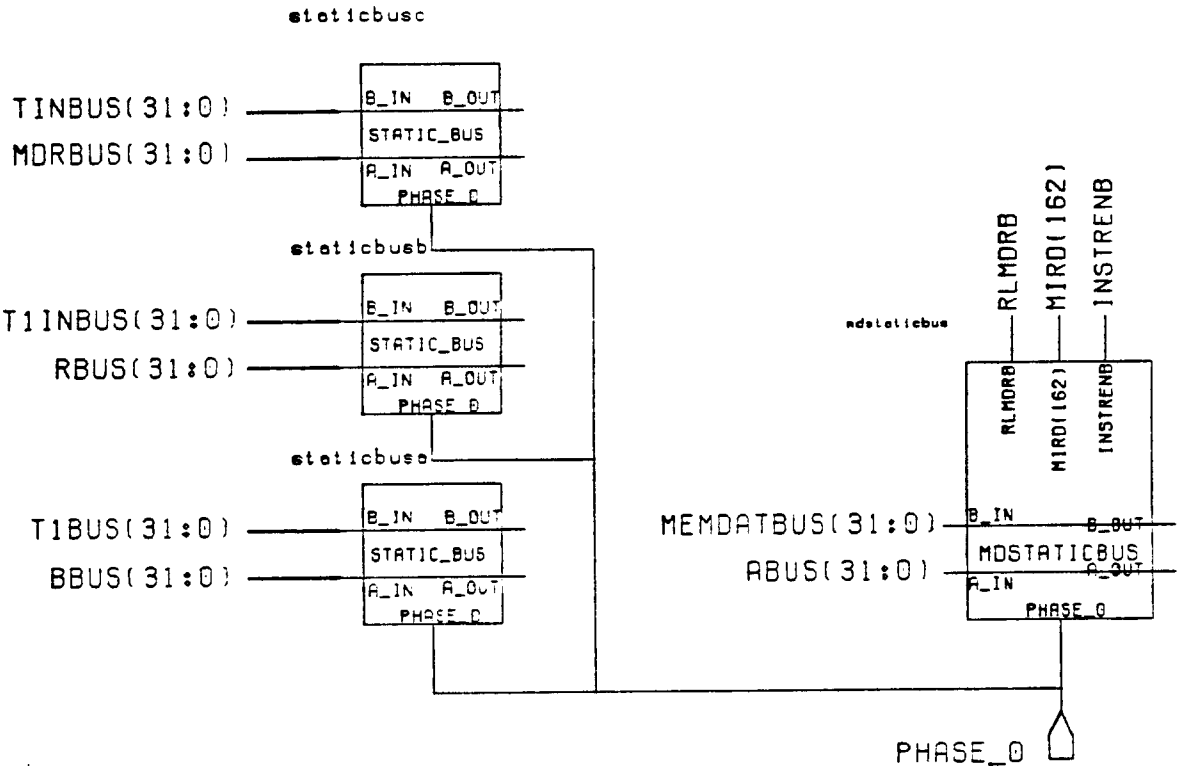

MIRD(17)
MIRD(18)
MIRD(19)
MIRD(20)
MIRD(21)
MIRD(22)
falu
S3
S2
S1
S0
M
Cn
BBUS(31:0)
ALUBUS(31:0)
ABUS(31:0)
B(31:0)
SUM(31:0)
A(31:0)
FALU
EQ
ALU_EQ
ALU_C
MIRD(23)
MIRD(24)
MIRD(25)
MIRD(26)
MIRD(27)
MIRD(28)
MDRBUSTOR
ALUBUSTOR
LDR
RTOBBUS
RTORBUS
RTOMEMDATBUS
MDRBUS(31:0)
BBUS(31:0)
ALUBUS(31:0)
RBUS(31:0)
MEMDATBUS(31:0)
MDR_BUS(31:0)
B_BUS(31:0)
ALU_BUS(31:0)
R_BUS(31:0)
R
MEMDAT_BUS(31:0)
PHASE_0
PHASE_1
MIRD(29)
MIRD(30)
MIRD(31)
MIRD(32)
MIRD(33)
MIRD(34)
MIRD(35)
MIRD(36)
MIRD(37)
MIRD(38)
MIRD(39)
MIRD(40)
MIRD(41)
MIRD(42)
MIRD(43)
MIRD(163)
MDRBUSTOMDR
ALUBUSTOMDR
RBUSTOMDR
T1BUSTOMDR
TBUSTOMDR
MEMDATBUSTOMDR
LDMDR
LDMDRTAG
MDRTAGSEL
MDRTAG30
MDRTAG31
BTCDR
TCDR0
TCDR1
TCDR2
MIRD(163)
MDRBUS(31:0)
BBUS(31:0)
ALUBUS(28:0)
ALUBUS(31:30)
RBUS(31:0)
T1BUS(31:0)
TBUS(31:0)
MEMDATBUS(31:0)
MDR_BUS_IN(31:0)
B_BUS(31:0)
ALU_BUS(28:0)
ALU_BUS(31:30)
R_BUS_IN(31:0)
T1_BUS(31:0)
T_BUS(31:0)
MEMDAT_BUS_IN(31:0)
MDR
MDR_BUS_OUT(31:0)
R_BUS_OUT(31:0)
MEMDAT_BUS_OUT(31:0)
MDRBUS
RBUS(3
MEMDAT
PHASE_0
PHASE_1
RLMDRB
CUTM
MDR_29_OUT
MDR_30_OUT
MDR_31_OUT
MDRTOMDRBUS
MDRTOBBUS
MDRTORBUS
MDRTOMEMDATBUS
EXCEPT
MIRD(44)
MIRD(47)
MIRD(45)
MIRD(46)

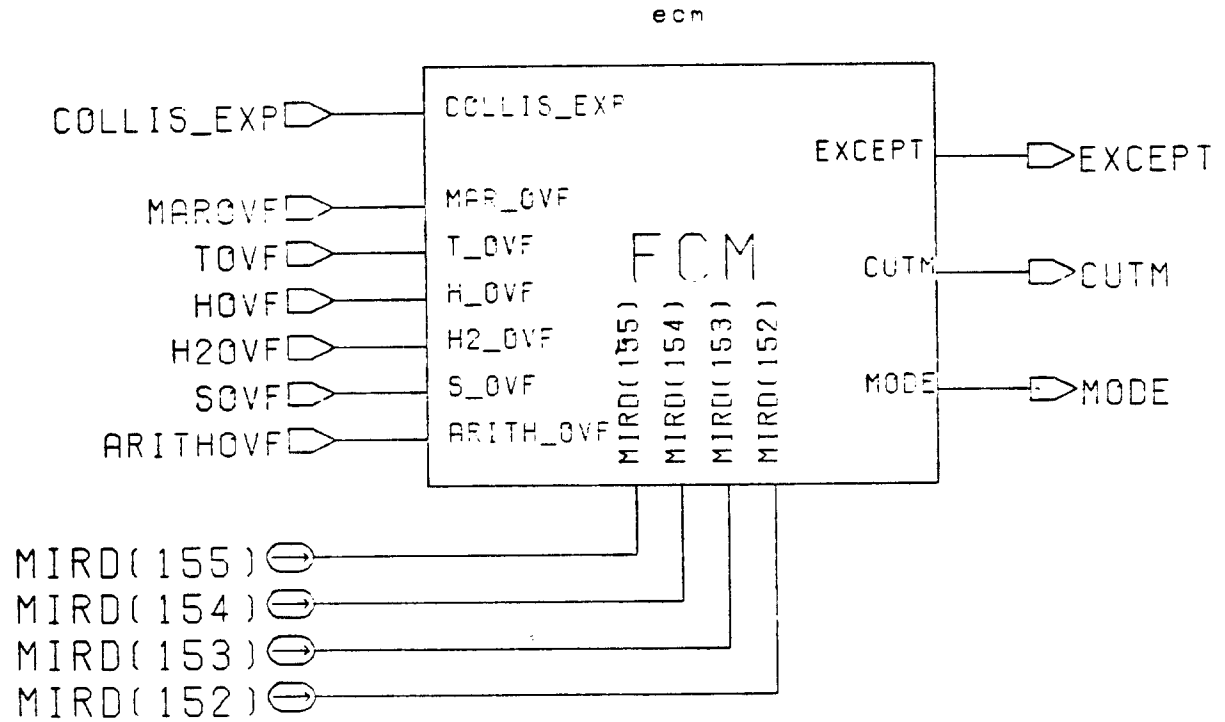

MIRD(59)
MIRD(60)
MIRD(61)
MIRD(62)
MIRD(63)
MIRD(64)
MIRD(65)
MIRD(66)
t1
BBUSTOT1
RBUSTOT1
T1INBUSTOT1
LDT1
NPASST1
NUMVALT1
T1TOBBUS
T1TOABUS
BBUS(31:0)
B_BUS_IN(31:0)
RBUS(31:0)
R_BUS(31:0)
ABUS(31:0)
A_BUS(31:0)
T1BUS(31:0)
T1_BUS(31:0)
T1INBUS(31:0)
T1IN_BUS(31:0)
BBUS(31:0)
PHASE_0
PHASE_1
T1_31
T1_30
PHASE_0
PHASE_1
T1_31
T1_30
MIRD(67)
MIRD(68)
MIRD(69)
MIRD(70)
MIRD(72)
MIRD(73)
MIRD(74)
MIRD(75)
MIRD(76)
tblock
MDRBUStoT
BBUStoT
RBUStoT
T1INBUStoT
LDT
TCNTEN
TUP*
NPASST
NUMVALT
MDRBUS(31:0)
MDRBUS_IN(31:0)
BBUS(31:0)
BBUS_IN(31:0)
RBUS(31:0)
RBUS(31:0)
TINBUS(31:0)
TINBUS(31:0)
T1INBUS(31:0)
T1INBUS_IN(31:0)
T_30
T_31
MDRBUS_OUT(31:0)
BBUS_OUT(31:0)
ABUS(31:0)
TBUS(31:0)
T1INBUS_OUT(31:0)
OVF
UNF
T_30
T_31
MDRBUS(31:0)
BBUS(31:0)
ABUS(31:0)
TBUS(31:0)
T1INBUS(31:0)
TOVF
TUNF
PHASE_1
T1toMDRBUS*
T1toBBUS*
T1toABUS*
T1INBUStoT
T1toT1INBUS*
PHASE_1
MIRD(77)
MIRD(78)
MIRD(79)
MIRD(71)
MIRD(65)

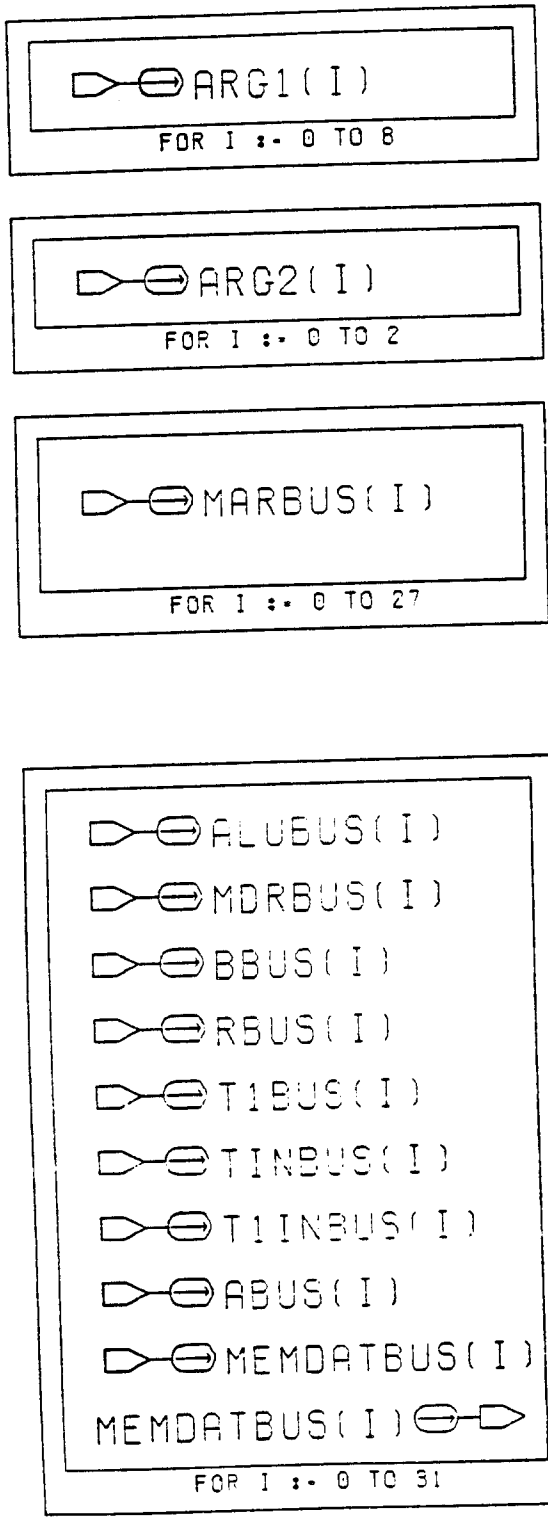

MIRD(80)
MIRD(81)
MIRD(82)
MIRD(83)
MIRD(84)
MIRD(85)
hblock
MDRBUStoH
RBUStoH
TINBUStoH
LDH
HCNTEN
HUP
MDRBUS(31:0)
RBUS(31:0)
TINBUS(31:0)
MDRBUS(31:0)
RBUS_IN(31:0)
TINBUS_IN(31:0)
TINBUS_OUT(31:0)
RBUS_OUT(31:0)
TIINBUS(31:0)
TINBUS(31:0)
RBUS(31:0)
TIINBUS(31:0)
OVF
UNF
HOVF
HUNF
HtoTINBUS
HtoRBUS
HtoTIINBUS
PHASE_1
MIRD(88)
MIRD(87)
MIRD(86)
MIRD(89)
MIRD(90)
MIRD(91)
MIRD(92)
MIRD(93)
sblock
MDRBUStoS
BBUStoS
LDS
SCNTEN
SUP
MDRBUS(31:0)
BBUS(31:0)
MDRBUS_IN(31:0)
BBUS(31:0)
MDRBUS_OUT(31:0)
TIINBUS(31:0)
MDRBUS(31:0)
TIINBUS(31:0)
OVF
UNF
SOVF
SUNF
StoMDRBUS
StoTIINBUS
PHASE_1
MIRD(94)
MIRD(95)


MIRD(96)
MIRD(97)
MIRD(98)
MIRD(99)
MIRD(100)
MIRD(101)
h2block
MDRBUStoH2
RBUStoH2
T1INBUStoH2
LOH2
H2CNTEN
H2UP
MDRBUS(31:0)
RBUS(31:0)
T1INBUS(31:0)
MDRBUS(31:0)
RBUS(31:0)
T1INBUS_IN(31:0)
H2
T1INBUS_OUT(31:0)
TINBUS(31:0)
OVF
UNF
H2toTINBUS
H2toT1INBUS
PHASE_1
H2OVF
H2UNF
MIRD(102)
MIRD(103)
PHASE_1
MIRD(104)
MIRD(105)
MIRD(106)
MIRD(107)
MIRD(108)
MDRBUSTON
BBUSTON
LON
NTOTINBUS
NTOMDRBUS
MDRBUS(31:0)
BBUS(31:0)
MDR_BUS_IN(31:0)
B_BUS(31:0)
MDR_BUS_OUT(31:0)
MDRBUS(31:0)
TINBUS(31:0)
TIN_BUS(31:0)
N
PHASE_0
PHASE_1
PHASE_0
PHASE_1


AXARGPDL
ARG1(2:0)
ARG2(2:0)
PHASE_0
PREV_OVF
PDLZERO
AXARG1(2:0)
AXARG2(2:0)
PDL0
AXARGPDL
MIRD(109)
MIRD(111)
MIRD(110)
MIRD(112)
MIRD(113)
MIRD(114)
MIRD(115)
MIRD(116)
MIRD(123)
MIRD(124)
MDRBUS(31:0)
BBUS(31:0)
RBUS(31:0)
TINBUS(31:0)
TIINBUS(31:0)
REGFILE
LDTAG(7:0)
TAG(1:0)
MIRD(125)
collis_mux
COLLISION_BIT
TINBUS(27:13)
TIINBUS(27:13)
BUSOUT(27:13)
B(27:13)
COLLIS_MUX
comp_15bit
BBUS(27:13)
B(27:13)
comp_15bitout
collis_exp
comp_15
G


staticbusc
TINBUS(31:0)
MDRBUS(31:0)
B_IN B_OUT
STATIC_BUS
A_IN A_OUT
PHASE_0
staticbusb
T1INBUS(31:0)
RBUS(31:0)
staticbusa
T1BUS(31:0)
BBUS(31:0)
mdstaticbus
RLMDRB
MIRD(16?)
INSTRENB
MEMDATBUS(31:0)
ABUS(31:0)
MDSTATICBUS
PHASE_0


ecm
COLLIS_EXP
MAROVF
TOVF
HOVF
H2OVF
SOVF
ARITHOVF
MAR_OVF
T_OVF
H_OVF
H2_OVF
S_OVF
ARITH_OVF
FCM
MIRD(155)
MIRD(154)
MIRD(153)
MIRD(152)
EXCEPT
CUTM
MODE
MIRD(155)
MIRD(154)
MIRD(153)
MIRD(152)


ARG1(I)
FOR I :- 0 TO 8
ARG2(I)
FOR I :- 0 TO 2
MARBUS(I)
FOR I :- 0 TO 27
ALUBUS(I)
MDRBUS(I)
BBUS(I)
RBUS(I)
T1BUS(I)
TINBUS(I)
T1INBUS(I)
ABUS(I)
MEMDATBUS(I)
MEMDATBUS(I)
FOR I :- 0 TO 31
PHASE_0
PHASE_1
EXCEPT
CUTM
MODE
PREV_OVF
PDL0
T_31
T(31)
T_30
T(30)
T1_31
T1(31)
T1_30
T1(30)
MDR_31
MDR(31)
MDR_30
MDR(30)
MDR_29
MDR(29)
PDLOVF
PDL_OVF
PDLUNF
PDL_UNF
ALU_CC(0)
ALU_SIGN
ALU_CC(1)
ALU_EQU

5. SHEETS OF THE MICROSEQUENCER

The top level sheet of the microsequencer is enclosed along with the details of the micropage select logic and next microprogramcounter select logic. The logic equations are derived from a C program.

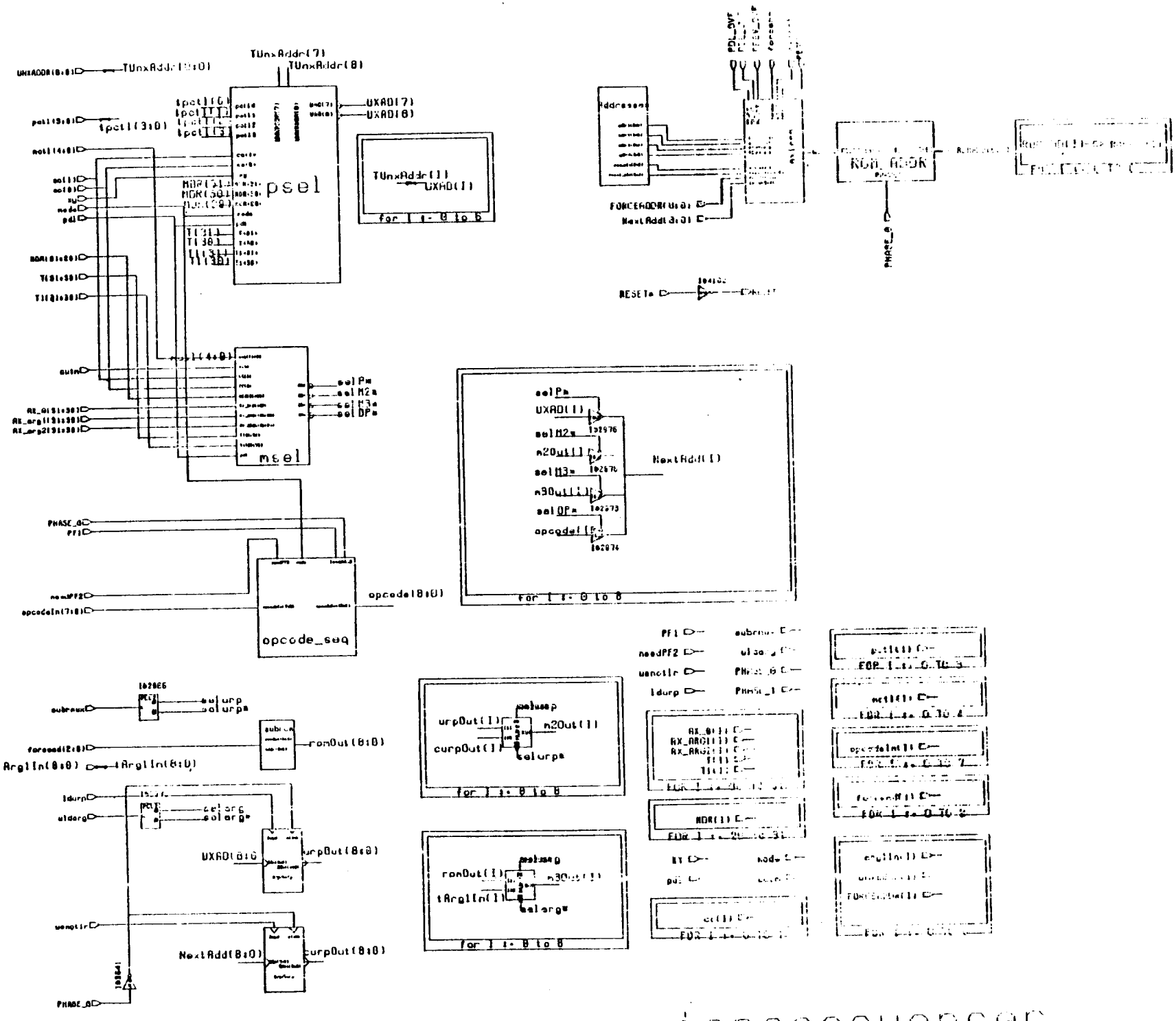
5.1. PAGE SELECT

The logic for this unit is given by the C program shown below.

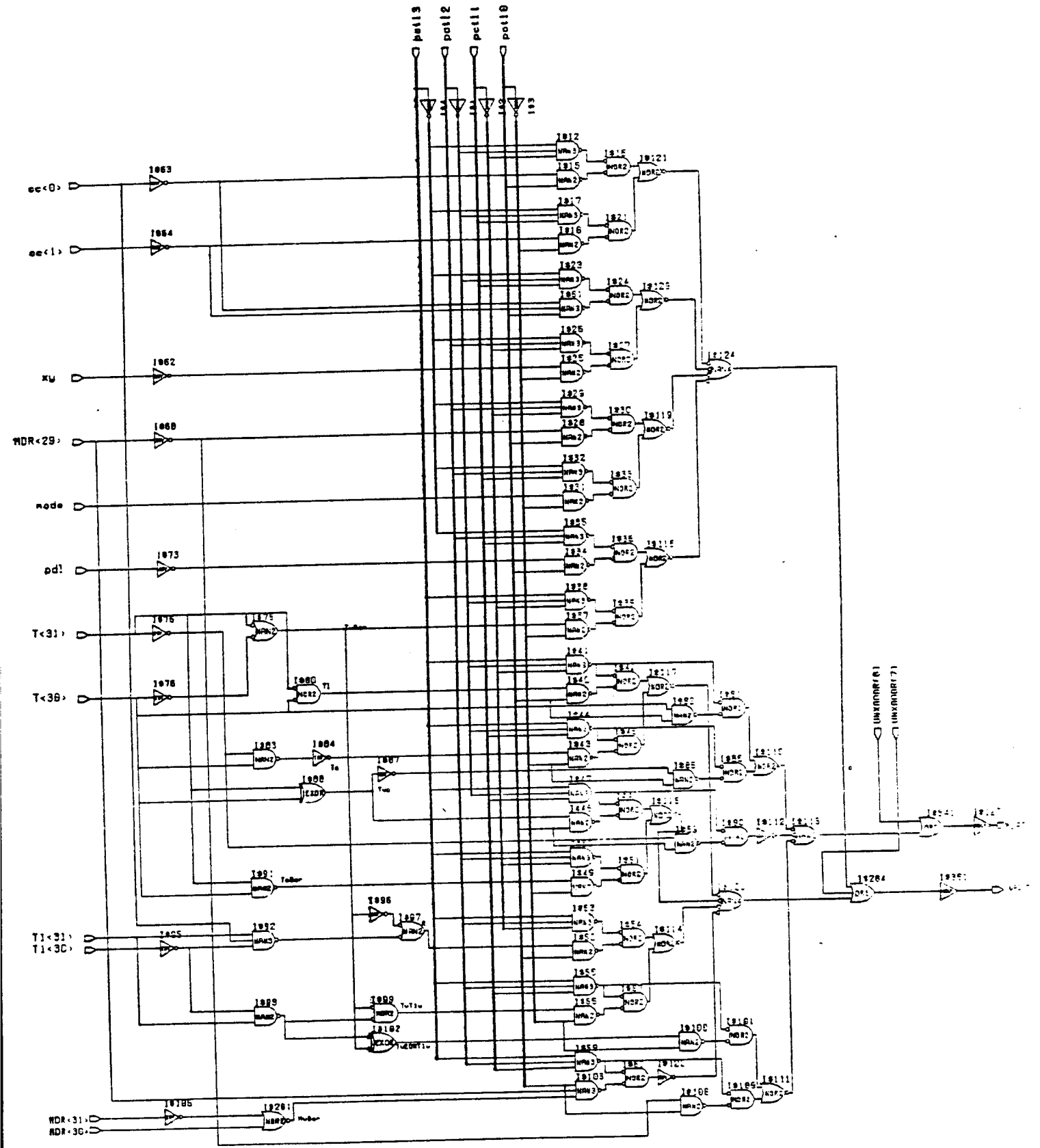
```
psel( P, p1, p2, p3, p4 )
 int P,p1,p2,p3,p4;

{
  switch( P )
  { case  0 : return( p1 );
    case  1 : if( cc == 1 ) return( p1 ); /*  cond <  */
                    else return( p2 );
    case  2 : if( cc == 2 ) return( p1 ); /*  cond == */
                    else return( p2 );
    case  3 : if( cc != 0 ) return( p1 ); /*  cond <= */
                    else return( p2 );
    case  4 : if( XY ) return( p1 );
                    else return( p2 );
    case  5 : if( MDR & tcdr ) return( p1 );
                    else return( p2 );
    case  6 : if( mode == read ) return( p1 );
                    else return( p2 );
    case  7 : if( PDL == 0 ) return( p1 );
                    else return( p2 );
    case  8 : switch( type( T ))
                    {  case tvar : return( p1 );
                       case tstr :
                       case tcon :
                       case tlst : return( p2 );  };
    case  9 : switch( type( T ))
                    {  case tvar : return( p1 );
                       case tlst : return( p2 );
                       case tcon :
                       case tstr : return( p3 );  };
    case 10 : switch( type( T ))
                    {  case tvar : return( p1 );
                       case tstr : return( p2 );
                       case tcon :
                       case tlst : return( p3 );  };
    case 11 : switch( type( T ))
                    {  case tcon : return( p1 );
                       case tvar : return( p2 );
                       case tlst : return( p3 );
                       case tstr : return( p4 );  };
    case 12 : switch( type( T ))
                    {  case tcon : return( p1 );
                       case tstr :
                       case tvar :
                       case tlst : return( p2 );  };
    case 13 : if(( type( T ) == tvar ) && ( type( T1 ) == tvar ))
            {  if( cc == 1 ) return( p2 );
               else return( p1 ); }
```

```
        else if( type( T ) == tvar ) return( p1 );
        else return( p2 );
case 14 : if(( type(T)!=tvar) && (type(T1)!=tvar )) return( p1 );
        else if(( type(T)==tvar) && (type(T1)==tvar )) return( p2 );
        else return( p3 );
case 15 : if( MDR & tcdr )
                    if( type(MDR) != tvar ) return( p1 );
                    else return( p2 );
                  else return( p3 );
default : return( -1 );   };   }
```



microsequencer


pctl3
pctl2
pctl1
pctl0
ec<0>
ec<1>
xu
MDR<29>
mode
pdl
T<31>
T<30>
T1<31>
T1<30>
MDR<31>
MDR<30>
UNXMDR(6)
UNXMDR(7)

psel

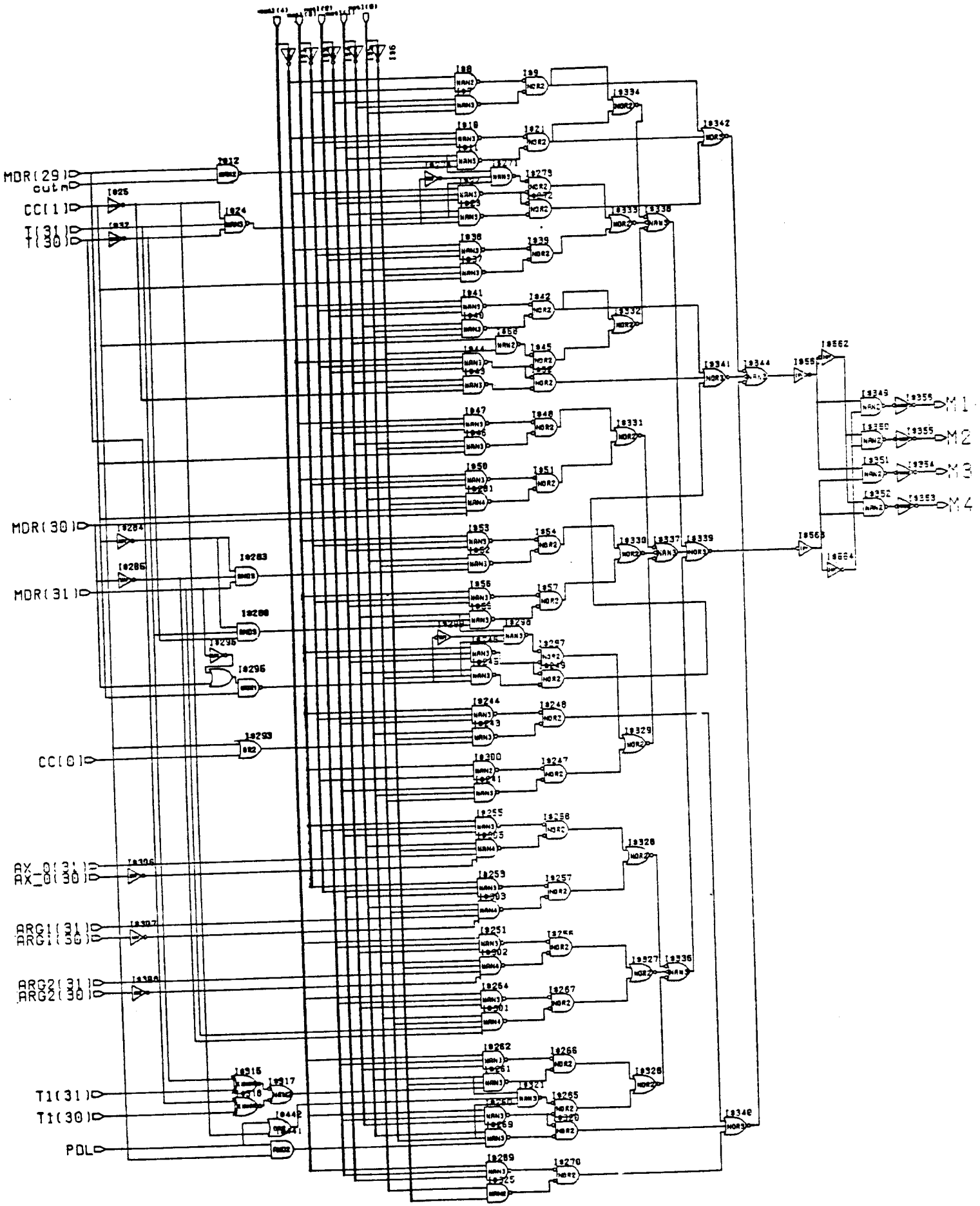
5.2. MICROPROGRAMCOUNTER SELECT

The logic for this unit is given by the C program shown below.

```
msel( M, m1, m2, m3, m4 )
 int M,m1,m2,m3,m4;

{
  switch( M )
  { case  0 : return( m1 );
    case  1 : return( m4 );
    case  2 : if( MDR & cutm ) return( m1 );
                    else return( m4 );
    case  3 : if(( cc != 2 ) && ( type(T) == tvar)) return( m3 );
                    else return( m2 );
    case  4 : if( cc == 2 ) return( m3 );
                    else return( m1 );
/*  case  5 : */
    case  41 : if( cc == 2 ) return( m4 );
                    else return( m1 );
/*  case  6 : */
    case  5 : if( cc == 2 ) return( m3 );
                    else return( m4 );
/*  case  7 : */
/* NOTE: m2 really means m1  */
    case  6 : if( MDR & tcdr ) return( m3 );
                    else return( m2 );
/*  case  8 : */
    case  7 : if((MDR&tcdr)&&((type(MDR)!=tvar)&&(type(MDR)!=tlst)))
                  return( m3 );
                else return( m1 );
/*  case  9 : */
/* NOTE: m2 really means m1  */
    case  71 : if(( type(MDR) == tvar )  && !(MDR&tcdr)) return( m3 );
                    else return( m2 );
/*  case  10: */
/* NOTE: m2 really means m1  */
    case  72 : if(( type(T) == tvar )  && !(MDR&tcdr)) return( m3 );
                    else return( m2 );
/*  case  11: */
    case  8 : if(( MDR&tcdr ) && ( type(MDR) != tvar ))
                  return( m3 );
                else return( m2 );
/*  case  12: */
    case  9 : if( cc != 0 ) return( m2 );
                    else return( m1 );
/*  case  13: */
    case 10 : return( m3 );
/*  case  14: */
/* NOTE: m2 really means m1  */
    case 11 : if( type( AX[0] ) == tvar ) return( m3 );
                    else return( m2 );
/*  case  15: */
/* NOTE: m2 really means m1  */
    case 12 : if( type( AX[arg1] ) == tvar ) return( m3 );
                    else return( m2 );
```

```
/*   case  16: */
/* NOTE: m2 really means m1  */
    case 121 : if( type( AX[arg2] ) == tvar ) return( m3 );
                    else return( m2 );
/*   case  17: */
/* NOTE: m2 really means m1  */
    case 122 : if( type( MDR ) == tvar ) return( m3 );
                    else return( m2 );
/*   case  18: */
    case 13 : if( type( T ) != type( T1 )) return( m3 );
                    else return( m1 );
/*   case  19: */
    case 14 : if( cc != 2 ) return( m3 );
                    else if( PDL == 0 ) return( m4 );
                    else return( m1 );
/*   case  20: */
    case 15 : return( m2 );
    default : return( -1 );   }; }
```


MDR(29)
cutm
CC(1)
T(31)
T(30)
MDR(30)
MDR(31)
CC(0)
AX_0(31)
AX_0(30)
ARG1(31)
ARG1(30)
ARG2(31)
ARG2(30)
T1(31)
T1(30)
POL
M1
M2
M3
M4

6. TIMING DIAGRAMS

The chip has several interface signals to assist system designers. Timing diagrams are included to show the interaction between these signals and the external environment. Interfacing the chip to a cache or a standard bus require an understanding of these timing diagrams. Since the program counter for the VLSI-PLM is not on chip, any interface to the chip must contain a program counter and logic to do instruction prefetching and partial decoding.

VLSI-PLM Pinout

The pinout of the VLSI-PLM consists of 102 signals described below. These signals are described as Input (signals to the VLSI-PLM), Output (signals from the VLSI-PLM), and I/O (a bi-directional signal with high impedance state). In addition, there are 9 VDD pins and 9 GND pins. The chip is packaged in a 168 lead pin grid array (PGA). All control signals must be available by 15ns after the rising edge of MCLK. All data is assumed to be available during MCLK* following the assertion of the appropriate control signal except for memory write which is supplied during the next cycle.

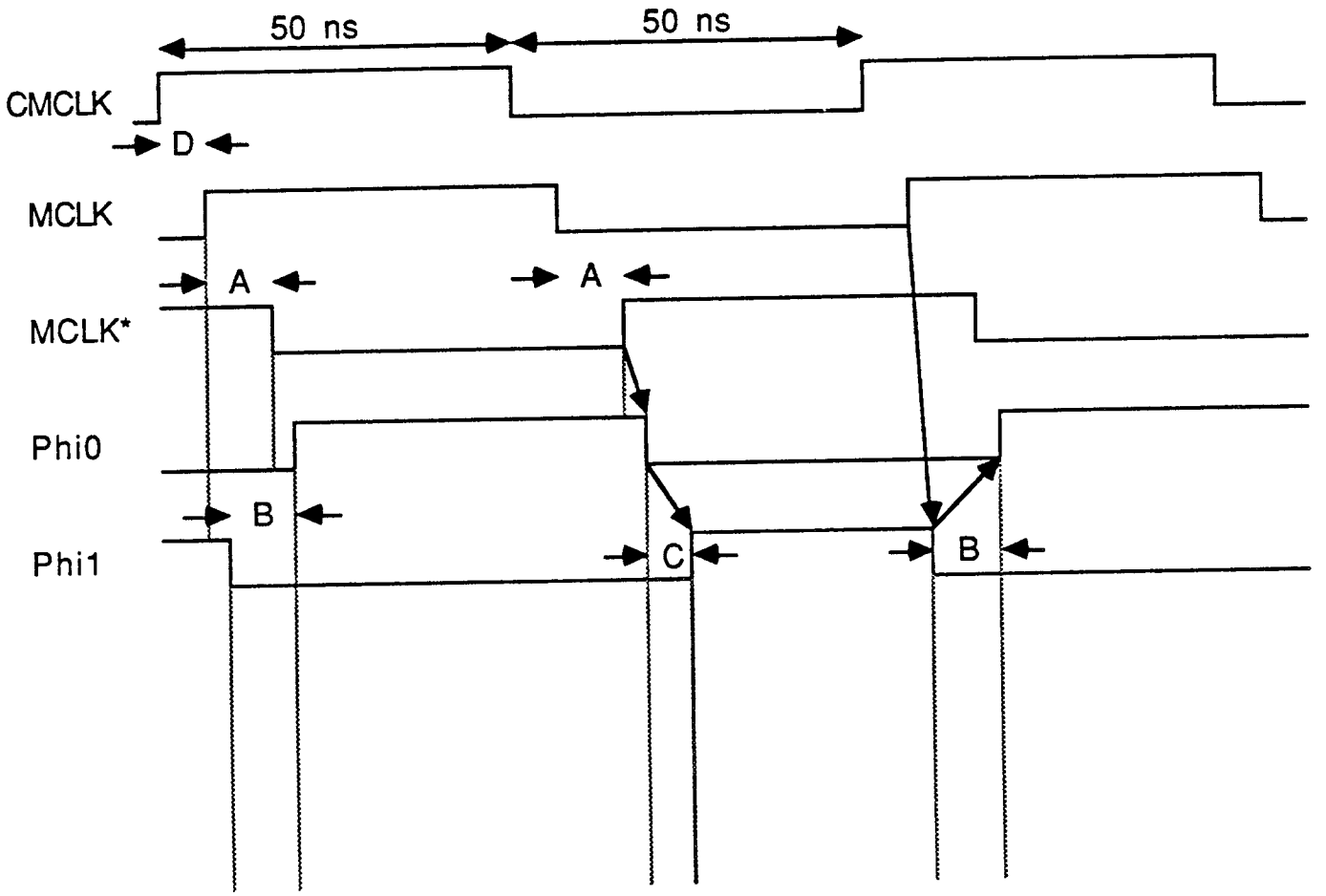
MAR<27..0>	(Output) A 28 bit memory address (usually virtual).
DSPACE	(Output) The most significant address bit for memory access. DSPACE is 1 for access to the Data Space and 0 for access to the Code Space (for Code Space items to be used as data). This signal and MAR bus forms the memory address bus.
EXCEPT	(Output) A one cycle long status signal indicating that an exception has occurred on the VLSI-PLM. The cache board generates an interrupt to the host. The VLSI-PLM supplies the interrupt driver with information on the cause of the exception by sending the contents of PSW on the MEMDAT bus. There will be a one cycle delay in communicating the PSW except for collision exception in which case it is supplied on the next cycle. This signal has the highest priority. The interface between the VLSI-PLM and the cache board must enforce the priority of this signal.
MEMDAT<31..0>	(I/O) The primary data path to memory. Memory read/write data to/from the MDR passes on this bus, as well as instruction arguments to arg1, arg2, and arg3 during instruction prefetch; and new values for the P register (either 32 bit or 8 bit for newp1 or newp2 respectively).
OPCODE<7..0>	(Input) The path for the 8 bit opcode from the prefetch buffer to the instruction register used during instruction prefetch by the VLSI-PLM.
NEWP1*	(Output) A one cycle long control signal to tell the Prefetch Unit that the MEMDAT bus holds a 32 bit value to reload the P register.
NEWP2*	(Output) A one cycle long control signal to tell the Prefetch Unit that the MEMDAT bus holds an 8 bit value to be added to the P register.
FAIL*	(Output) A one cycle long control signal to tell the Prefetch Unit that failure has occurred and that the prefetch buffer is to be flushed. The Prefetch Unit then waits for a NEWP1.
MEMREAD*	(Output) A one cycle long control signal to request a memory read. At the beginning of the cycle, the MAR bus has the memory address. The VLSI-PLM can expect to be able to latch the data from the MEMDAT bus towards the end of the MCLK* cycle (See the discussion below for more information on clocking). If the data is in the write buffer or there is a cache miss the cache board will stop the VLSI-PLM by freezing MCLK on the high level during the next cycle.
MEMWRITE*	(Output) A one cycle long control signal to request a memory write. The cache board latches the MAR and MEMDAT busses on the next rising edge of MCLK. If the signal is asserted during cycle t the data on MAR and MEMDAT buses will be latched during the rising edge of cycle t+2.
INSTREN*	(Output) A one cycle long control signal to request a transfer of data from the prefetch buffer. The data on the MEMDAT bus may be latched during MCLK*. It is the VLSI-PLM's responsibility to keep track of whether this is a prefetch1 (opcode and arg1) or a prefetch2 (arg2 and arg3). The MEMDAT bus should not be used for other transfers during this cycle.
RESET*	(Input) An arbitrarily long but synchronized control signal to the VLSI-PLM to reset. This signal will load the constant RAM with data from the communication page of memory (next to last page of data space with a page size of 2K

	bytes) and initialize the machine registers.
LASTMI*	(Output) A one cycle long control signal indicating that the last microinstruction of a PLM instruction is in execution. That is, end of macro instruction execution.
FORCEBR	(Input) A one cycle long control signal to inform the VLSI-PLM to do a forced microbranch to the address on the FORCEADDR bus.
FORCEADDR<8..0>	(I/O) A nine bit bus to transfer the forced microbranch address to the VLSI-PLM when FORCEBR is asserted and to output the contents of the ROM latch when OUTROMADDR is asserted.
PRECHARGE	(Input) A one cycle long control signal to inform the VLSI-PLM that the prcharge circuit of the register files and ROM should be enabled.
MCLK/MCLK*	(Input) The Master 100ns clock for the VLSI-PLM. The VLSI-PLM may assume that all data transfer requests (MEMREAD*,MEMWRITE*, and INSTREN*) occur in one cycle. If the cache board is unable to do this, due to a cache miss or buffer full or empty, MCLK will tick one more time and then stop with a High level. Once the data is available, MCLK will resume. If both MCLK and MCLK* are supplied then they will be used as phase 0 and phase 1 of a two phase nonoverlapping clock.
RLMDR*	(Input) A one cycle long control signal to reload the MDR register of the VLSI-PLM once data is available for a memory read after a cache miss. MCLK will resume 150ns after RLMDR* goes away. (Check this since the PLM uses 175ns)
TEST1	(Input) A one cycle long control signal to initiate the scan of microinstruction register (MIR) as a part of testing the VLSI-PLM.
TEST2	(Input) A one cyle long control signal to initiate the scan of status bits in the status unit of VLSI-PLM.
SHIFTA	(Input) Clock for shifting data into the master register of LSSD.
SHIFTIN1	(Input) Data for the first scan path controlled by TEST1.
SHIFTIN2	(Input) Data for the second scan path controlled by TEST2.
SHIFTOUT1	(Output) Data output from the first scan path controlled by TEST1.
SHIFTOUT2	(Output) Data output from the second path controlled by TEST2.
EXTERNALFU*	(Output) A one cycle long control output to the cacheboard indicating that a builtin function is to be executed by an external functional unit. If this signal and LASTMI are asserted at the same time then it indicates that the transfer of all the data to the cachboard for the execution of the external builtin function has been completed.
OUTMEMDAT	(Input) Control signal from the cache board indicating that the 32 pads of the MEMDAT bus should be in the output mode. The pads will also be in the output mode when MEMWRITE*, NEWP1*, NEWP2*, or diagnostics (internal signal) is asserted. The 32 pads will be in the input mode if MEMREAD*, RLMDR*, or INSTREN* is asserted. If none of the above signals for the input or output is asserted then the 32 pads will be in high impedance state. This signal is an asynchronous one. It is provided for reading the contents of the blocks in data path during the VLSI-PLM testing or debugging the hardware when MCLK is frozen in the high level (stays in phase 0) and a microinstruction is shifted into the microinstruction register.
WAIT	(Output) A one cycle long control signal to the cache board indicating that the chip is halted (looping on a microinstruction).
OUTROMADDR	(Input) A one cycle long control input to the VLSI-PLM requesting the contents of the ROM latch to be output on FORCEADDR0 - FORCEADDR8 pins. The

	next microinstruction address is in the ROM latch. This signal puts the 9 pads of FORCEADDR in output mode. The 9 pads will be in the input mode when FORCEBR signal is asserted. If both OUTROMADDR and FORCEBR are not asserted then the pads will be in the high impedance state.
POWER	There are 9 power pins.
GROUND	There are 9 ground pins.

Clocking Scheme

Rev 1 6/25/86
Rev 2 5/30/87



Delay A. External inverter delay. The two are equal if pull up & down times are the same for the external inverter. (4 ns).

Delay B. Non-overlap time after end of Phi1 and before start of Phi0. This is affected by the external inverter delay (5 ns).

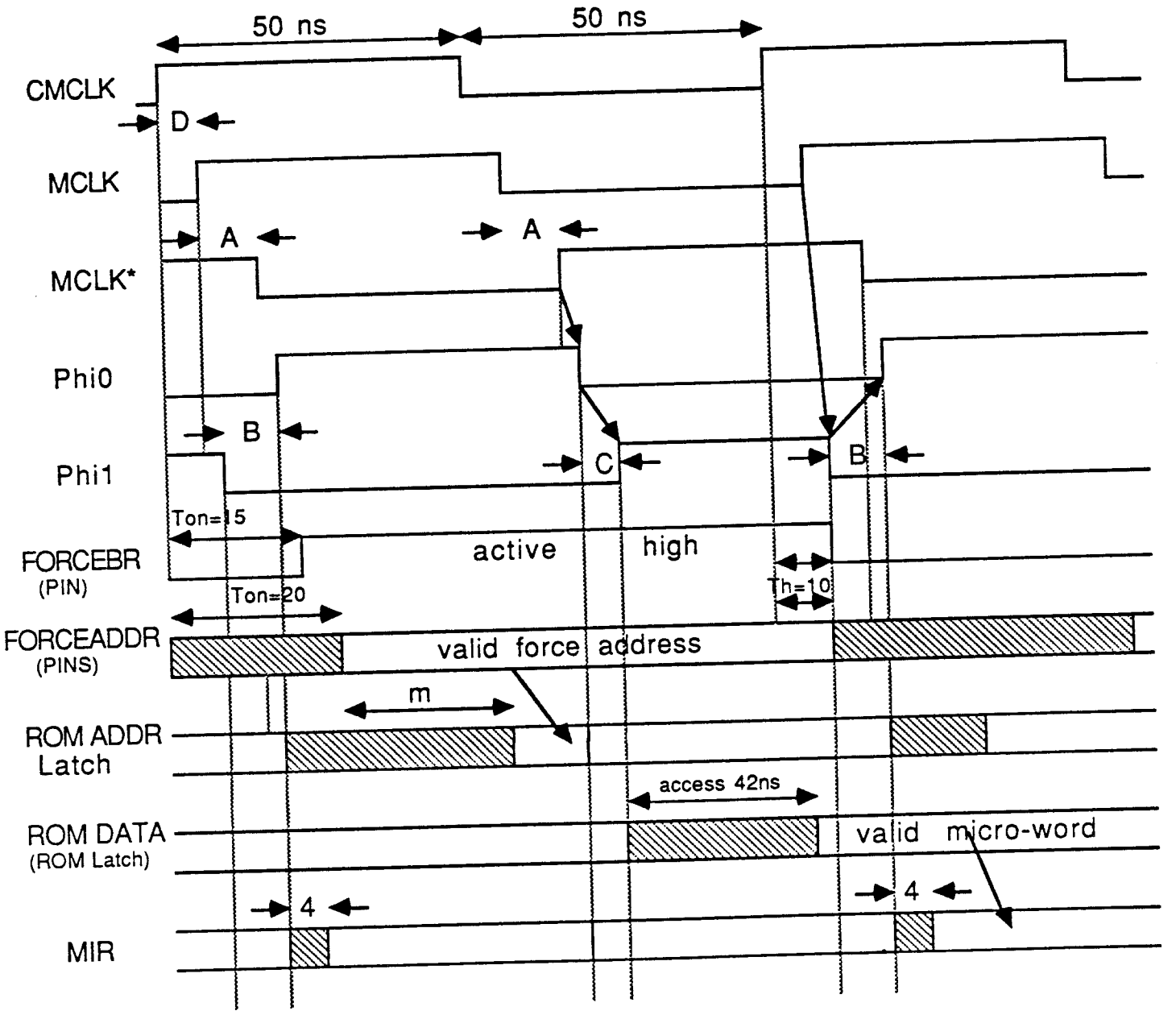
Delay C. Non-overlap time after end of Phi0 and before start of Phi1. This is affected by the internal driver delay (5 ns).

Delay D Delay of Cache clock to split into MCLK and MCLK* (4 ns).

Rise of Phi0 (Phi1) is caused by fall of Phi1 (Phase0).
Fall of Phi0 (Phi1) is caused by rise of MCLK* (MCLK).

FORCE BRANCH TIMING

Rev 2 7/16/86
Rev 3 5/30/87
Rev 4 6/02/87



Note:

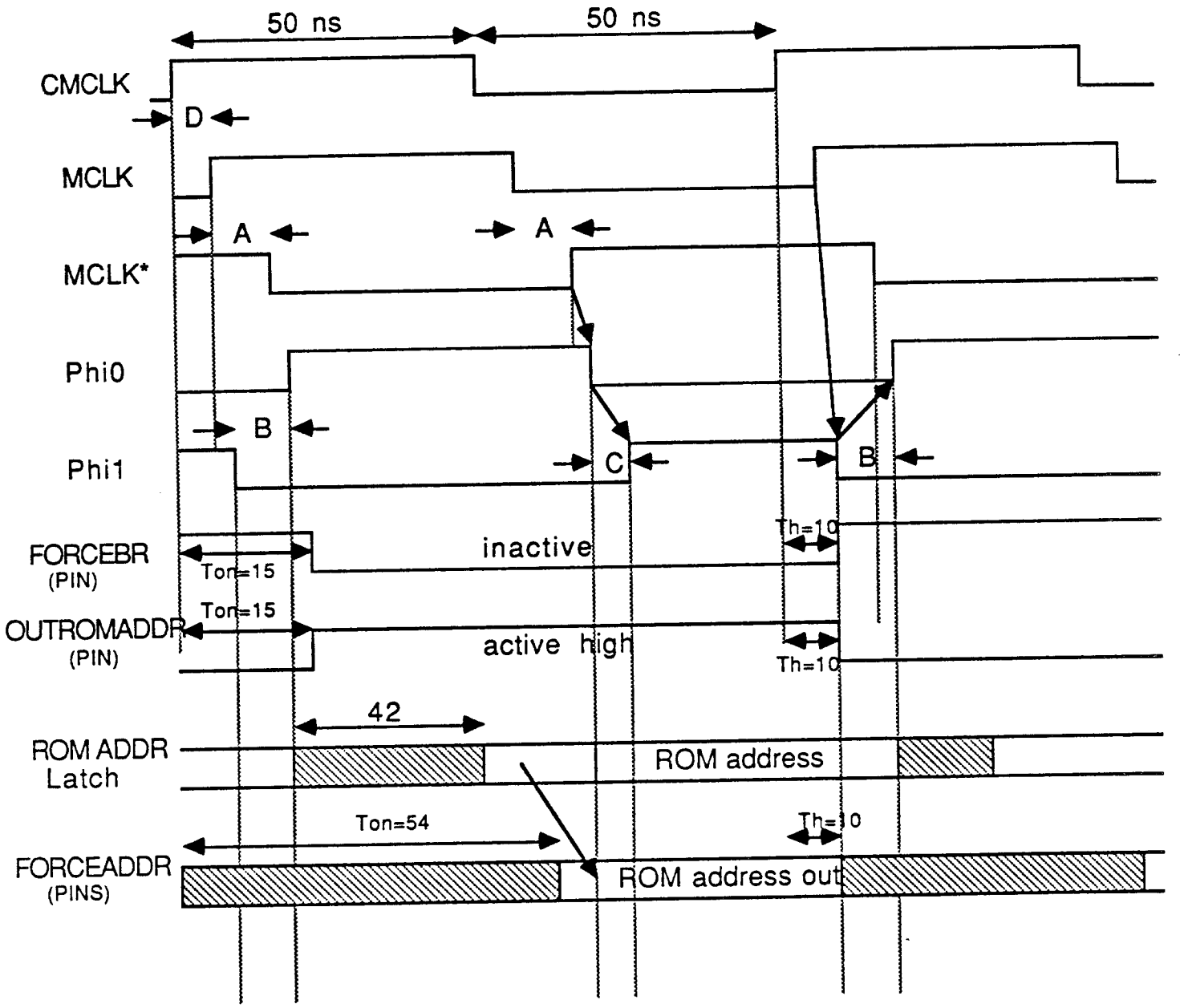
'm' is the mux time delay for selecting the Force Address as the next micro-address.

ROM Address is latched in on Phi0. ROM data latch is written in Phi1 and data is valid at end of Phi1. MIR latches in the microword in Phi0.

ROM data is available before end of Phi1, and stays valid until end of Phi0.

FORCE BRANCH LOW
(ROM ADDRESS OUT)

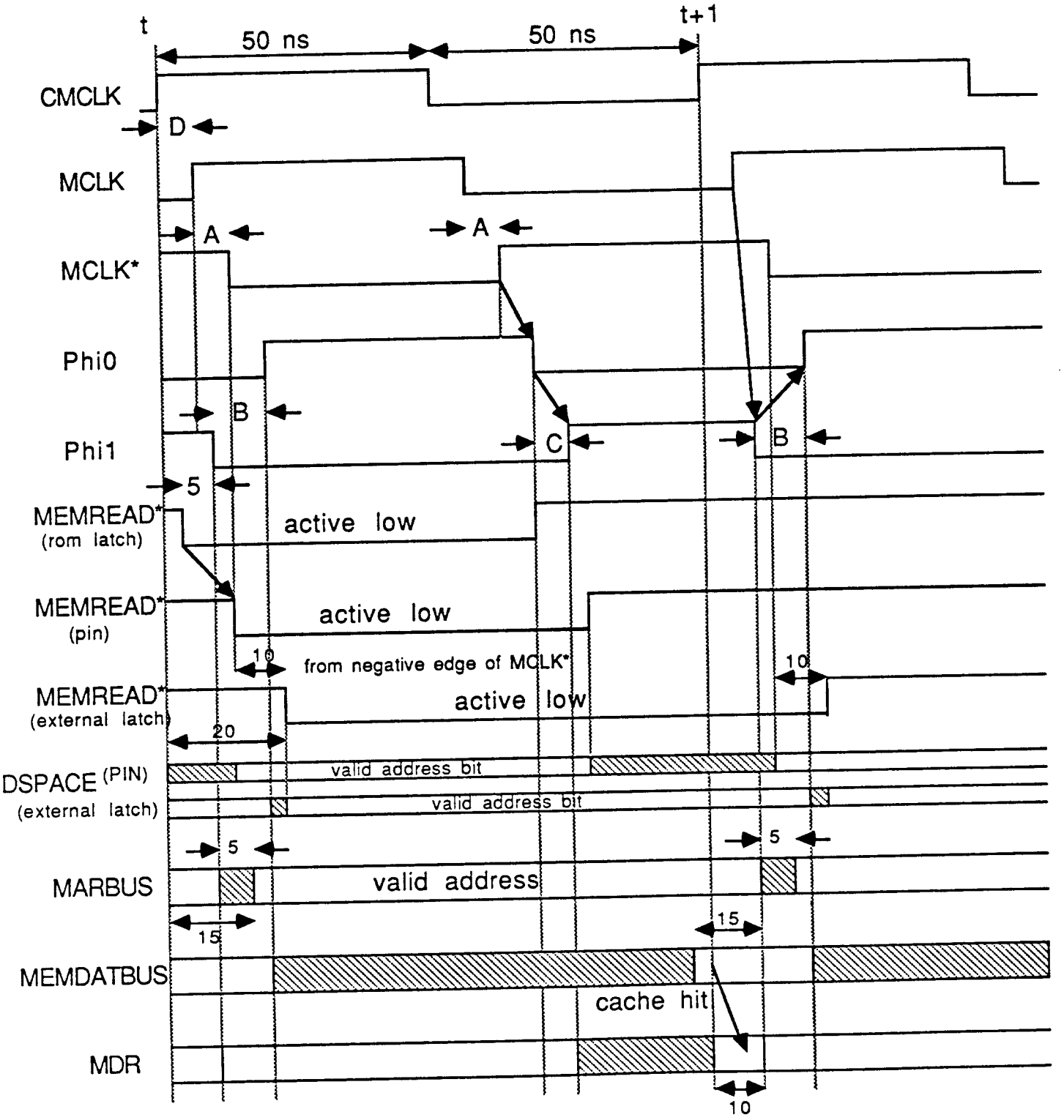
Rev 1 6/04/87
Rev 2 6/09/87



Cacheboard specs: FORCEBR (from CB) valid in ≤15ns from rise of CMCLK.
When FORCEBR is low, CB expects ROM Address out in ≤54ns.
OUTROMADDR comes from interface board, causes chip to open FORCEADDR I/O pads for output and to drive out 9-bit ROM address.
FORCEBR must be low when OUTROMADDR is high.

MEMREAD TIMING
CACHE HIT

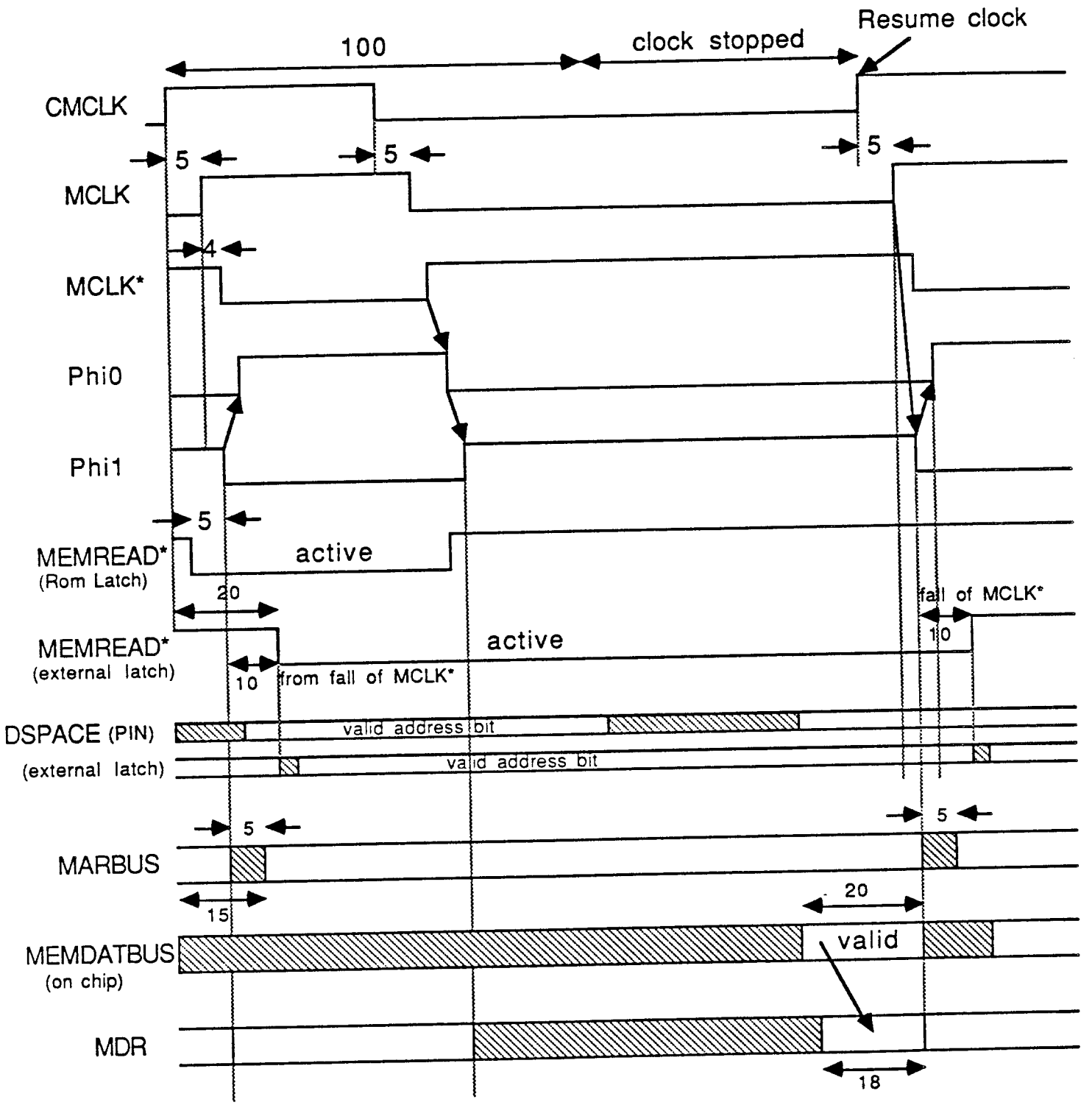
Rev 5 5/30/87
Rev 6 6/04/87



MEMREAD* signal and address must be available to cache board no later than 20ns after rising edge of CMCLK to allow sufficient time for stopping clock in the case of a cache miss.

MEMREAD TIMING
CACHE MISS

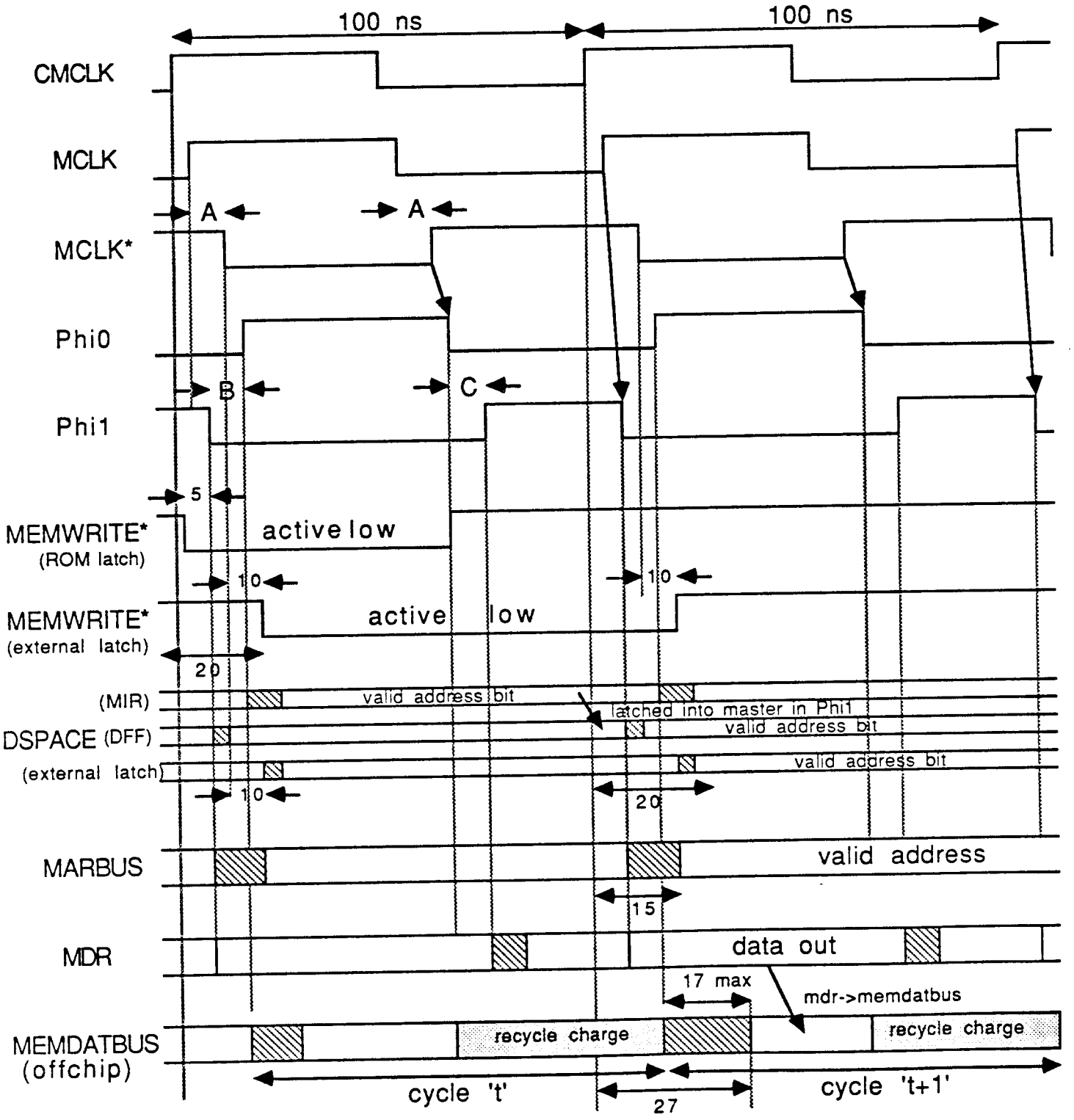
Rev 3	7/24/87
Rev 4	5/30/87
Rev 5	6/04/87



Cache Board Specs: when cache-miss occurs, CMCLK is kept from rising until data is available. Phi1 will be high when stopping clock.

MEMWRITE TIMING

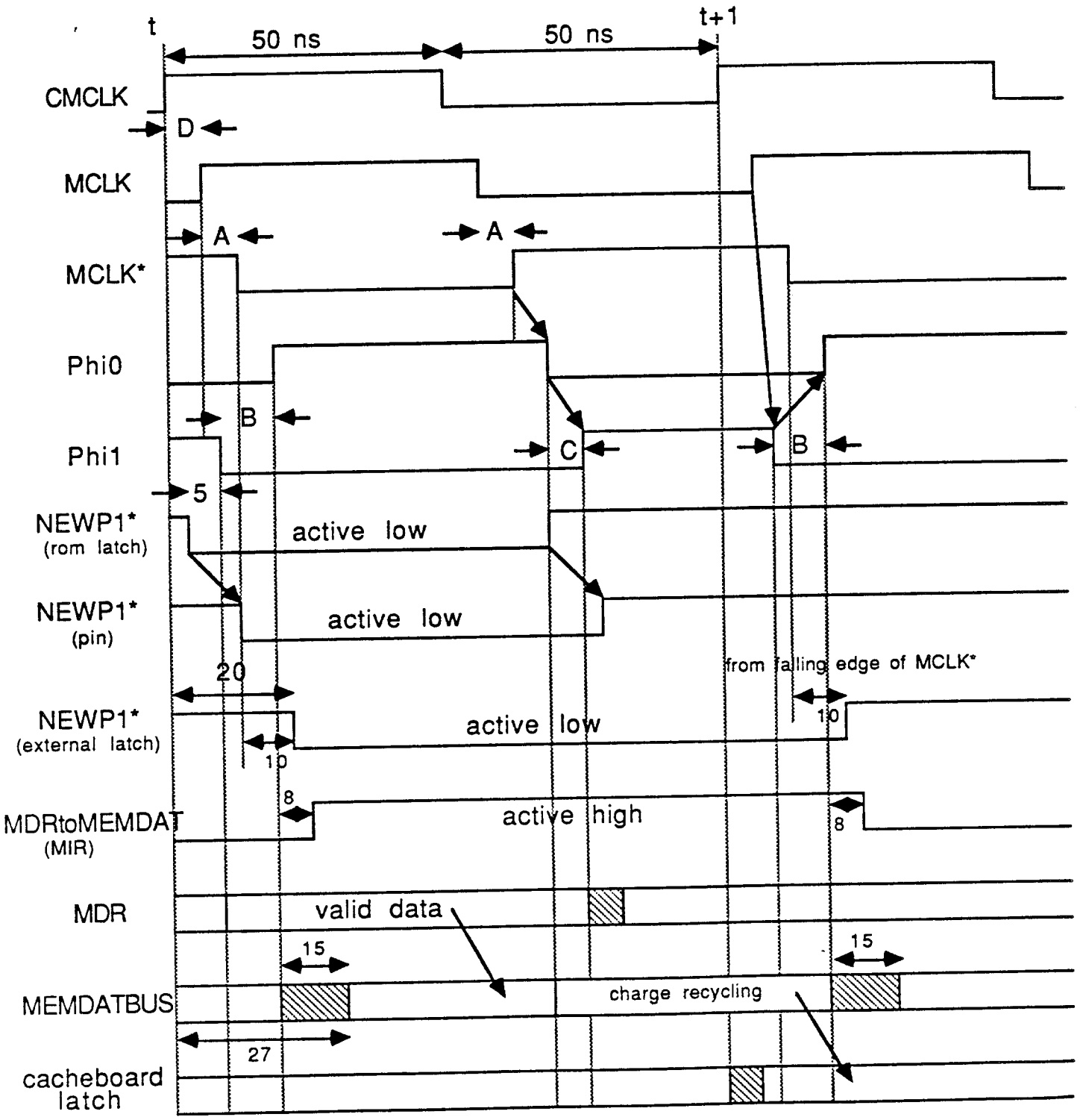
Rev 5 6/03/87
Rev 6 6/09/87



Cacheboard specs: cycle t, MEMWRITE* valid ≤20ns;
cycle t+1, Address valid ≤20ns, Memdatbus valid ≤27 ns from rise of CMCLK.

NEWP1 TIMING

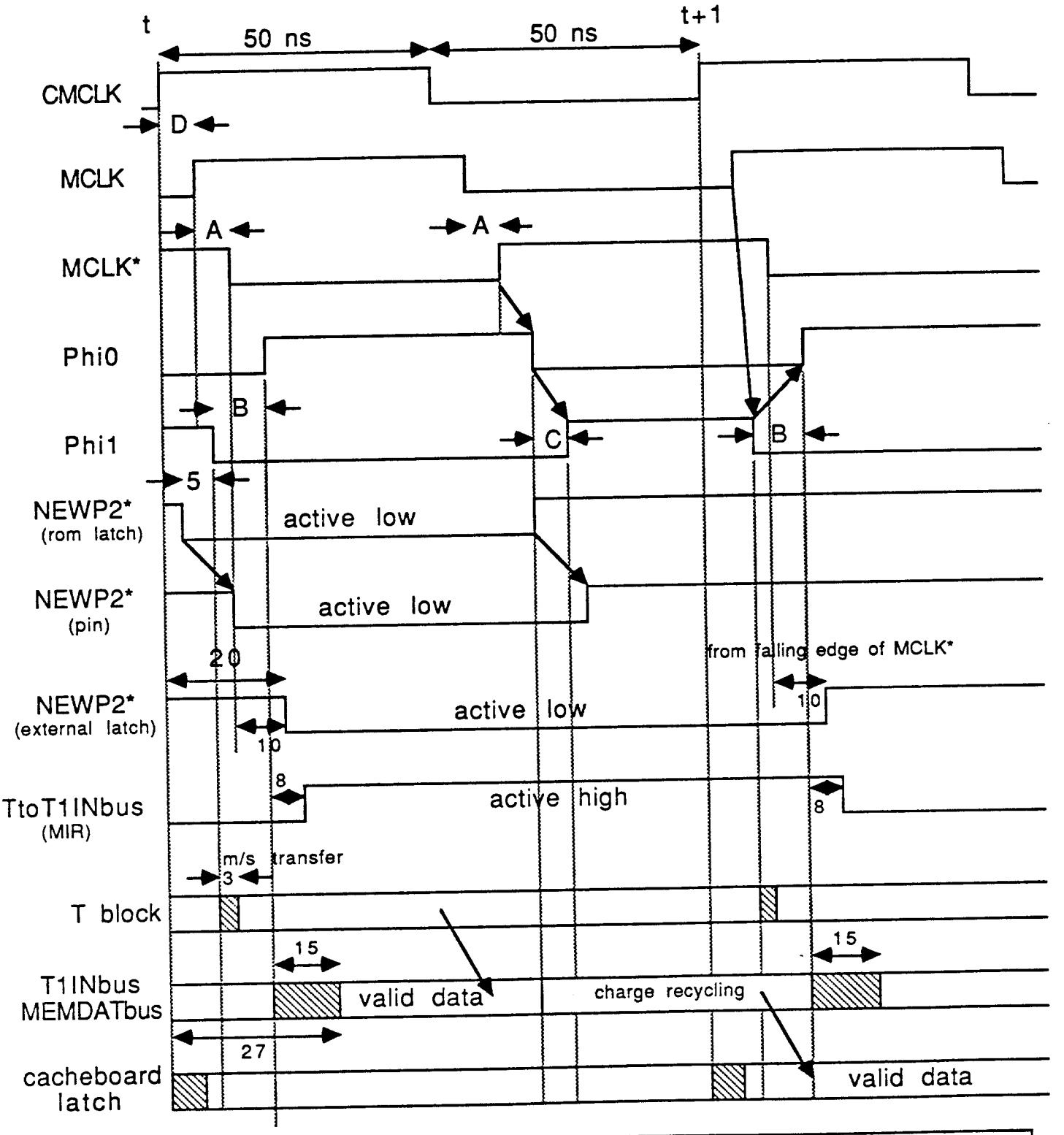
Rev 3 7/24/86
Rev 4 5/30/87
Rev 5 6/05/87



Cache board specs: NEWP1* valid in ≤20ns, data from MEMDATBUS valid in ≤27ns (after rise of CMCLK). Data going offchip can also come from R & CP (Regfile). R -> Memdatbus; CP -> T1inbus -> Memdatbus (longest delay). In any case, Memdatbus must be valid ≤15ns after rise of Phi0.

NEWP2 TIMING

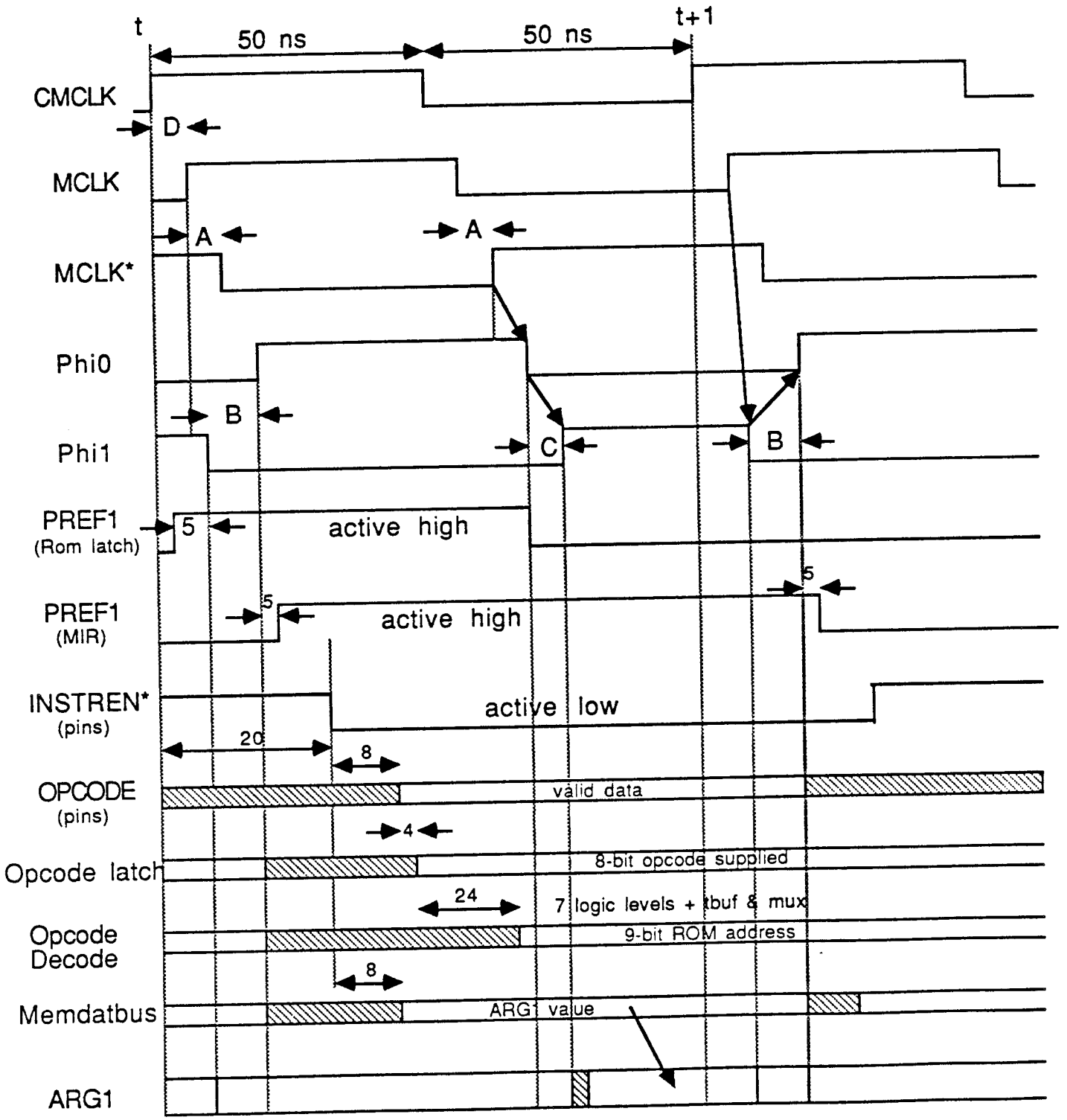
Rev 3 5/30/87
Rev 4 6/09/87



Cache board specs: NEWP2* valid in ≤20ns, data from MEMDATBUS valid in ≤27ns (after rise of CMCLK). T block Master/Slave transfer is done in Phase1*. T -> T1inbus -(bus connector)-> Memdatbus. Memdatbus must be valid ≤15ns after rise of Phi0.

PREF1 TIMING

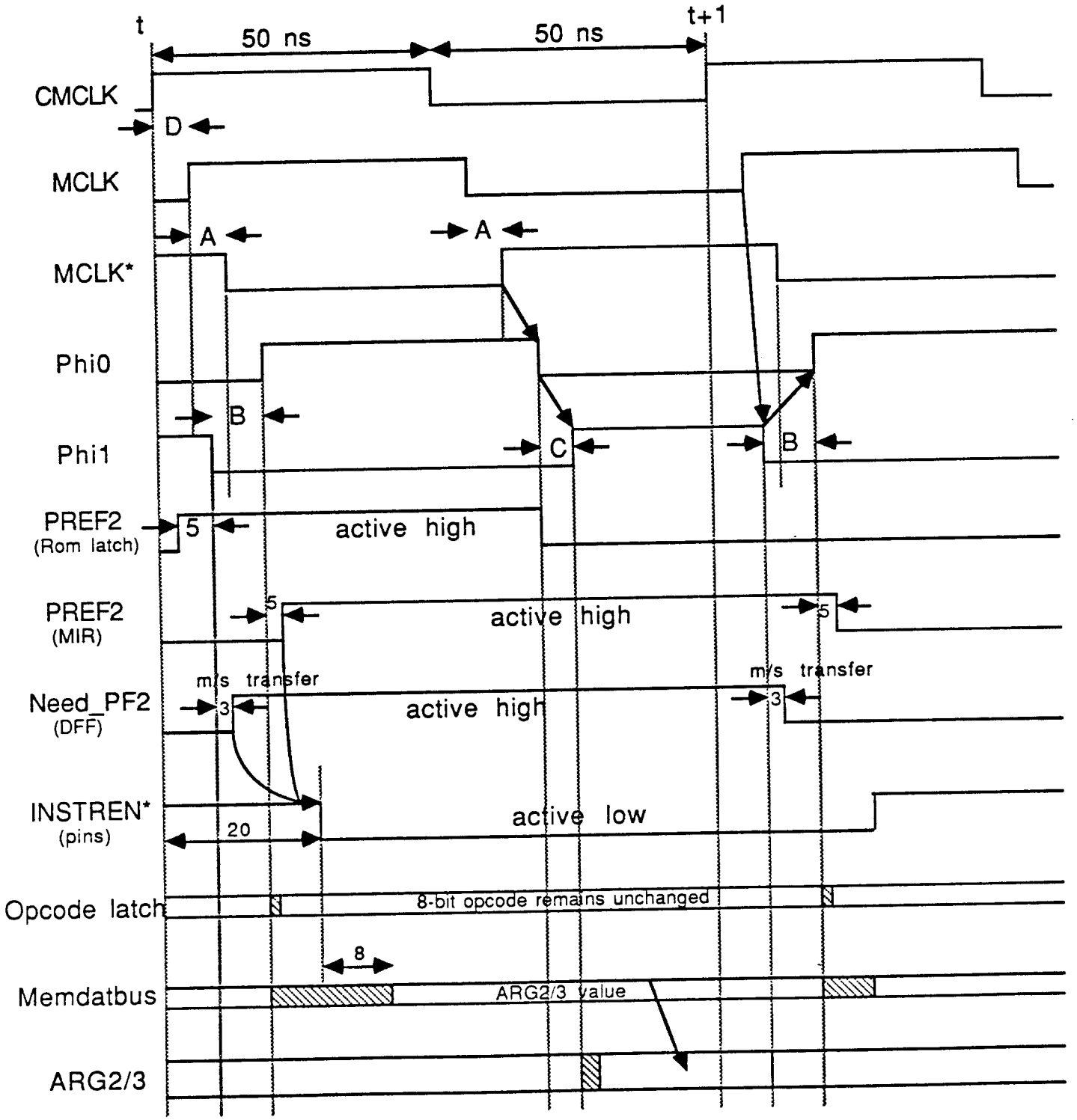
Rev 5 6/04/87
Rev 6 6/09/87



Cacheboard specs: INSTREN* valid in ≤20ns after rise of CMCLK. Cacheboard provides OPCODE and ARG1 ≤8ns after INSTREN* is valid.
Opcode pins and latch are 8-bits. Opcode goes thru Opcode Decode to become 9-bit ROM address (0:8), with bit 3 modified & bit 8 added.

PREF2 TIMING

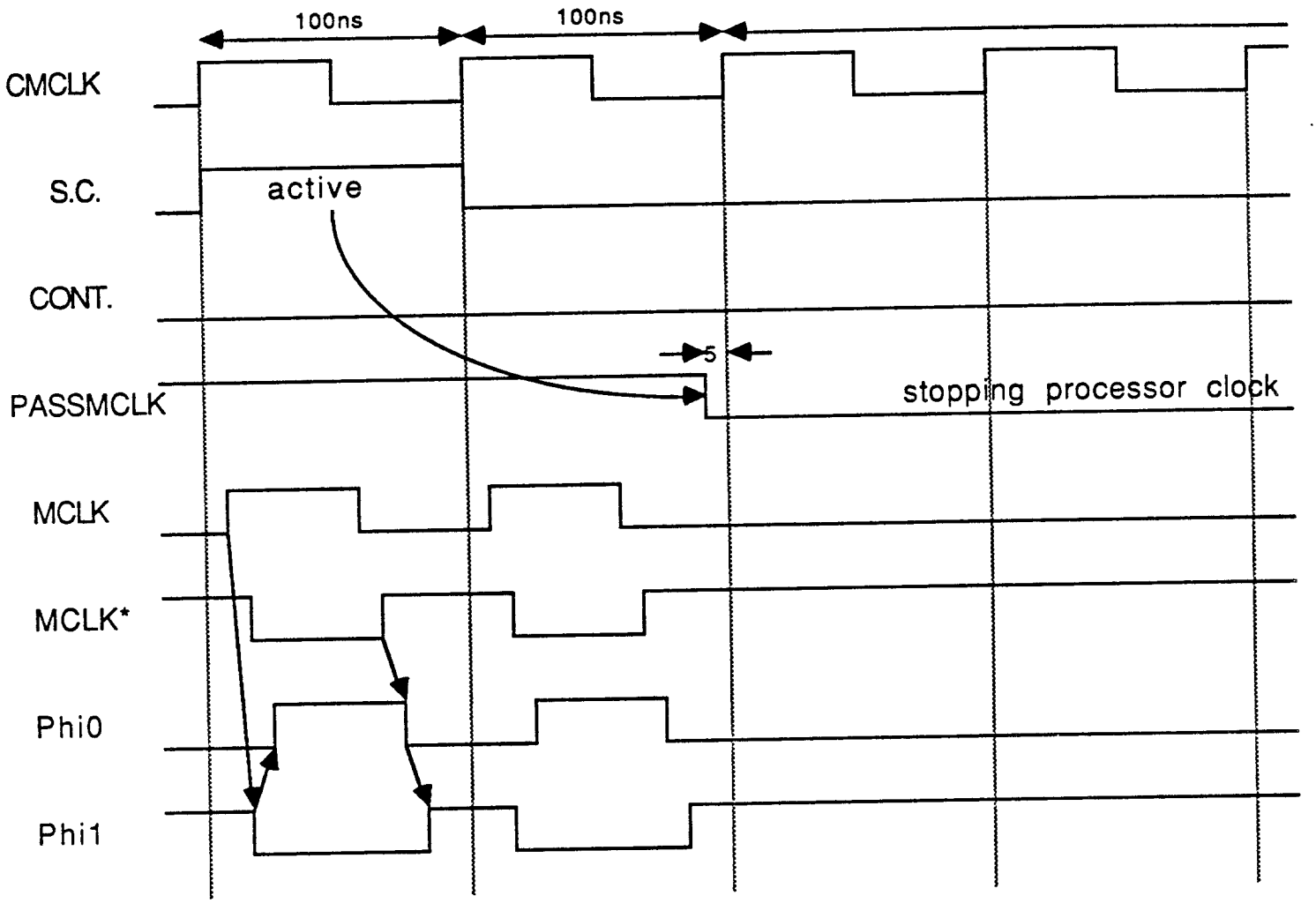
Rev 3 6/04/87
Rev 4 6/09/87



Cacheboard specs: INSTREN* valid in ≤20ns after rise of CMCLK. OPCODE and ARG2/3 from Cacheboard are valid ≤8ns after INSTREN* is valid.
Least significant 8 bits of Memdatbus go into ARG2, the next 8 bits into ARG3. (ARG2 & ARG3 both output into the least significant 8 bits of Bbus.

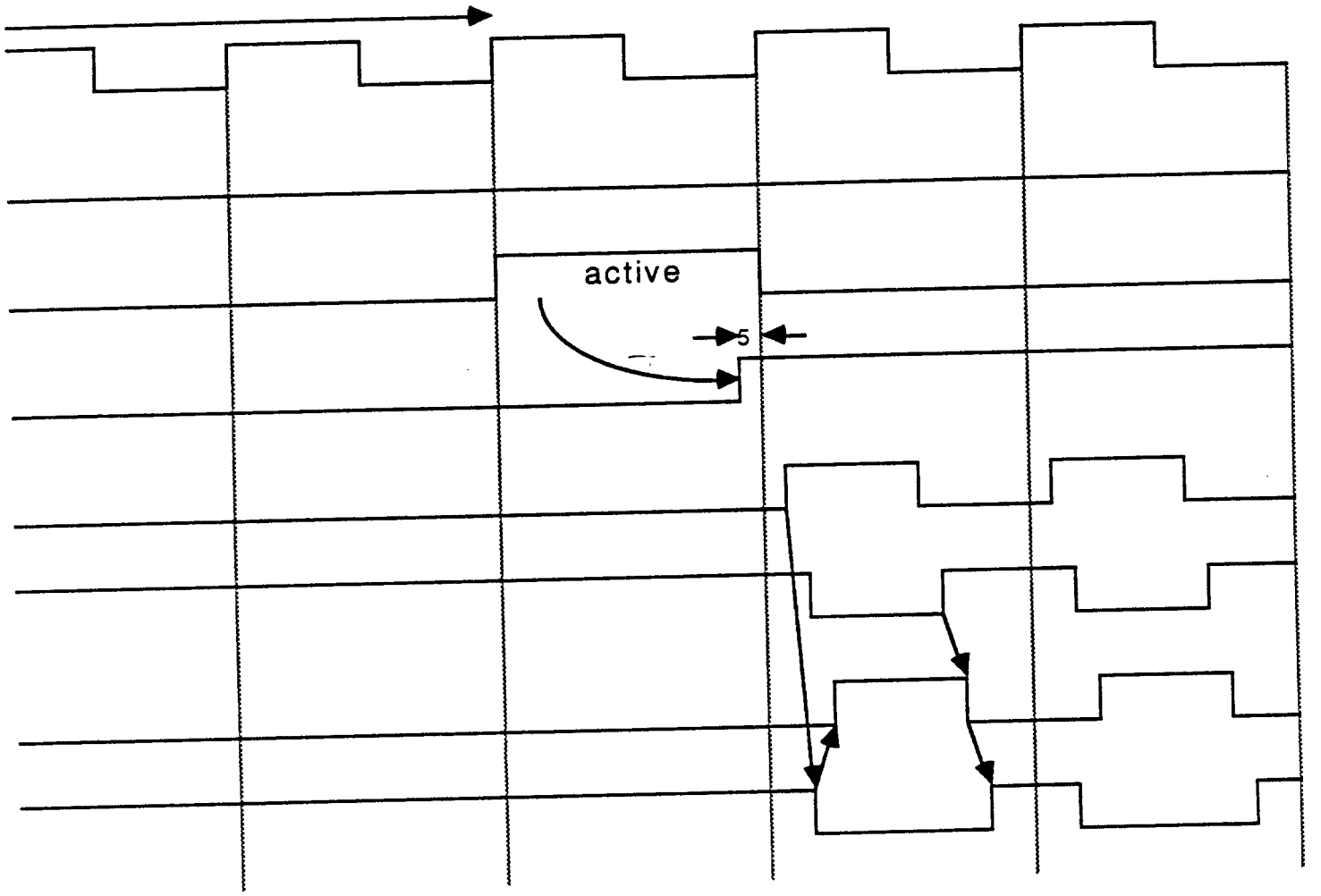
SINGLE CYCLE TIMING

Rev 2 7/22/8
Rev 3 6/09/8



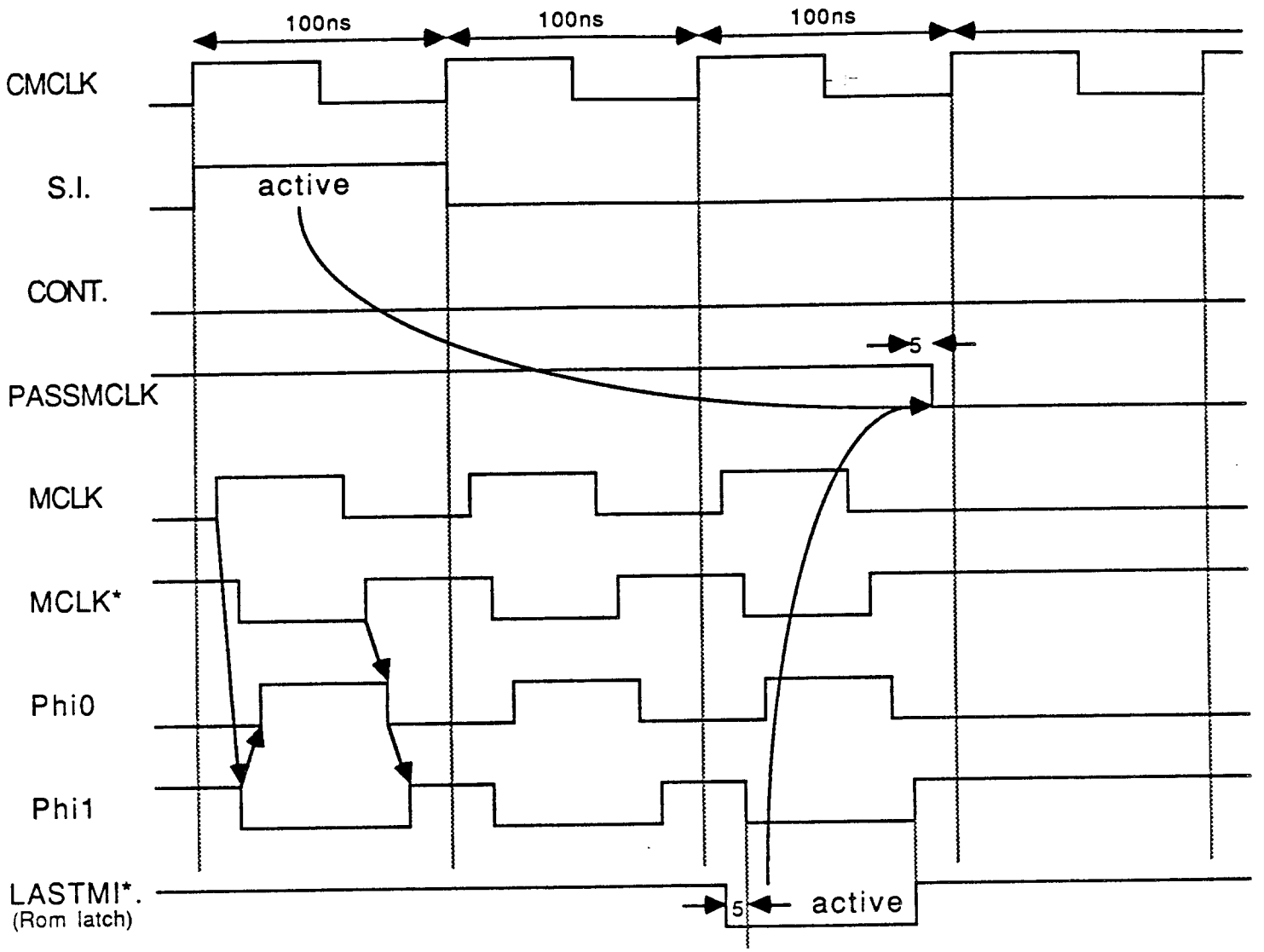
Cacheboard Specs: Single Cycle (SC) is valid in cycle t of CMCLK. MCLK runs 1 more cycle (t+1), then gets stopped (PASSMCLK low). Some number of cycles later, Continue is active to reactivate MCLK (PASSMCLK high). SC may also be valid for another single cycle execution. PASSMCLK must settle well before the rising edge of CMCLK for MCLK to be stopped and reactivated properly.

6
7

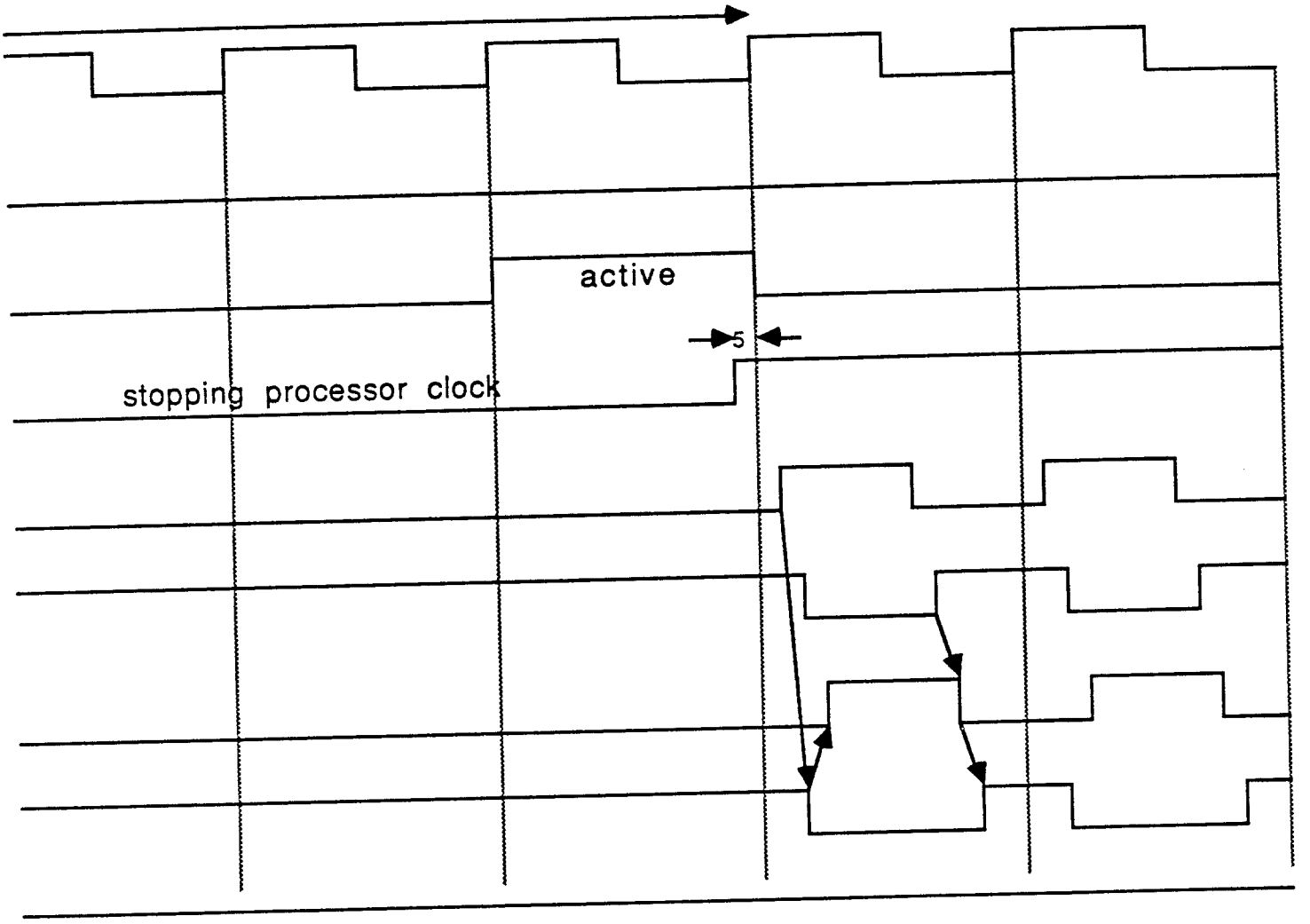


SINGLE INSTRUCTION

Rev 1 6/09/87



Cacheboard Specs: Single Instruction (SI) is valid in cycle t of CMCLK. MCLK runs several more cycles until LASTMI* is low, then gets stopped (PASSMCLK low). Some number of cycles later, Continue is active to reactivate MCLK (PASSMCLK high). SI may also be valid for another single instruction execution. PASSMCLK must settle well before the rising edge of CMCLK for MCLK to be stopped and reactivated properly.


active
5
stopping processor clock