Experimental Evaluation of the Certification-Trail Method

Gregory F. Sullivan,[1] Dwight S. Wilson,[2] Gerald M. Masson,[3]
Mamoru Itoh,[4] Warren W. Smith, Jonathan S. Kay[5]

Dept. of Computer Science, Johns Hopkins Univ., Baltimore, MD 21218

Abstract

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [1, 2, 3, 4]. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the method. The method is applied to algorithms for the following problems: huffman tree, shortest path, minimum spanning tree, sorting, and convex hull. Our results reveal many cases in which an approach using certification-trails allows for significantly faster overall program execution time than a basic time redundancy-approach.

We also examine algorithms for the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. We implemented and analyzed answer-validation solutions for two types of priority queues. In both cases, the algorithm which performs answer-validation is substantially faster than the original algorithm for computing the answers.

Next we present a probabilistic model and analysis which enables comparison between the certification-trail method and the time-redundancy approach. The analysis reveals some substantial and sometimes surprising advantages for the certification-trail method.

[1]Research partially supported by NSF Grants CCR-8910569 and CCR-8908092 and an IBM Technology Interchange Program Grant.
[2]Research partially supported by NSF Grant CCR-8910569 and an IBM Technology Interchange Program Grant.
[3]Research partially supported by NASA Grant NSG 1442 and an IBM Technology Interchange Program Grant.
[4]Visiting Scholar, Matsushita Electronic Components Co.
[5]Currently at Dept. of Computer Science, University California San Diego

Finally we discuss the work our group has performed on the design and implementation of fault injection testbeds for experimental analysis of the certification trail technique This work employs two distinct methodologies: software fault injection (modification of instruction, data, and stack segments of programs on a Sun Sparcstation ELC and on an IBM 386 PC) and hardware fault injection (control, address, and data lines of an Motorola MC68000-based target system pulsed at logical zero/one values). Our results indicate the viability of the certification trail technique. We also believe the tools we have developed provide a solid base for additional exploration.
Keywords: Software fault tolerance, certification trails, error monitoring, design diversity, data structures.

1 Introduction

Certification trails are a recently introduced and promising approach to fault-detection and fault-tolerance [1, 3]. In this paper, we report on a comprehensive attempt to assess experimentally the performance and overall value of the method. We have implemented several fundamental algorithms together with versions of the algorithms which generate and utilize certification trails. Specifically, algorithms for the following problems are analyzed: huffman tree, shortest path, minimum spanning tree, sorting, and convex hull. Our results reveal many cases in which an approach using certification trails allows for significantly faster overall program execution time than a basic time redundancy approach.

We also examine algorithms for the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. For this paper we implemented and analyzed answer-validation solutions for two abstract data types. The first solution is for a simplified priority queue which allows insert, min and deletemin operations, and the second solution is for a priority queue which allows insert, min, delete and deletemin operations. In both cases, the algorithm which performs answer-validation is substantial faster than the original algorithm for computing the answers.

This paper next presents a simple probabilistic model and analysis which enables comparison between the certification-trail method and the time-redundancy approach. The analysis shows that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the method has both a a smaller probability of error and a smaller probability of undetected error. Surprisingly, the analysis also reveals the intriguing result that the certification-trail method often can display superior performance even when the method has the same execution time or a longer execution time than the time-redundancy approach. This superior behavior stems from the typical assymetry of the execution times of the first and second executions in the certification-trail method.

The paper next discusses the work our group has performed on the design and implementation of fault injection testbeds. This work employs two distinct methodologies: software fault injection and hardware fault injection. The software fault injection tool is similar to an interactive debugger but more accurately can be considered an interactive bugger. It allows programs to be halted and faults to be injected by direct modification of the stack, data and instruction segments of a program. Output can then be captured and characterized.

The hardware fault injector is based on injecting faults into an operating microprocessor. The injection is performed by explicitly setting one or more pins of the microprocessor to logical zero and/or logical one values. The timing and duration of the pin setting is under control of a supervisory processor. The testbed also includes a multi-processor system. This system consists of three processors which are connected to one another pairwise by shared banks of dual ported memory. We plan to use this system to conduct evaluation of systems which utilize concurrent execution of algorithms using the certification-trail method.

2 Introduction to Certification Trails

To explain the essence of the certification-trail technique for software fault tolerance, we will first discuss a simpler fault-tolerant software method. In this method the specification of a problem is given and an algorithm to solve it is constructed. This algorithm is executed on an input and the output is stored. Next, the same algorithm is executed again on the same input and the output is compared to the earlier output. If the outputs differ then an error is indicated, otherwise the output is accepted as correct. This software fault tolerance method requires additional time, so-called time redundancy

[32, 52]; however, it requires no additional software. It is particularly valuable for detecting errors caused by transient fault phenomena. If such faults cause an error during only one of the executions then either the error will be detected or the output will be correct. The second possibility, of undetected faults, occurs when the output of the execution is unaffected by the faults.

A variation of the above method uses two separate algorithms, one for each execution, which have been written independently based on the problem specification. This technique, called N-version programming [16, 12] (in this case N=2), allows for the detection of errors caused by some faults in the software in addition to those cause by transient hardware faults and utilizes both time and software redundancy. Errors caused by software faults are detected whenever the independently written programs do not generate coincident errors.

The certification-trail technique is designed to obtain similar types of error-detection capabilities but expend fewer resources. The central idea, as illustrated in Figure 1, is to modify the first algorithm so that it leaves behind a trail of data which we call a *certification trail.* This data is chosen so that it can allow the the second algorithm to execute more quickly and/or have a simpler structure than the first algorithm. As above, the outputs of the two executions are compared and are considered correct only if they agree. Note, however, we must be careful in defining this method or else its error detection capability might be reduced by the introduction of data dependency between the two algorithm executions. For example, suppose the first algorithm execution contains an error which causes an incorrect output and an incorrect trail of data to be generated. Further suppose that no error occurs during the execution of the second algorithm. It still appears possible that the execution of the second algorithm might use the incorrect trail to generate an incorrect output which matches the incorrect output given by the execution of the first algorithm. Intuitively, the second execution would be "fooled" by the data left behind by the first execution. The definitions we give below exclude this possibility. They demand that the second execution either generate a correct answer or signal that an error has been detected in the data trail.

3 Formal Definition of a Certification Trail

In this section we will give a formal definition of a certification trail and discuss some aspects of its realizations and uses.

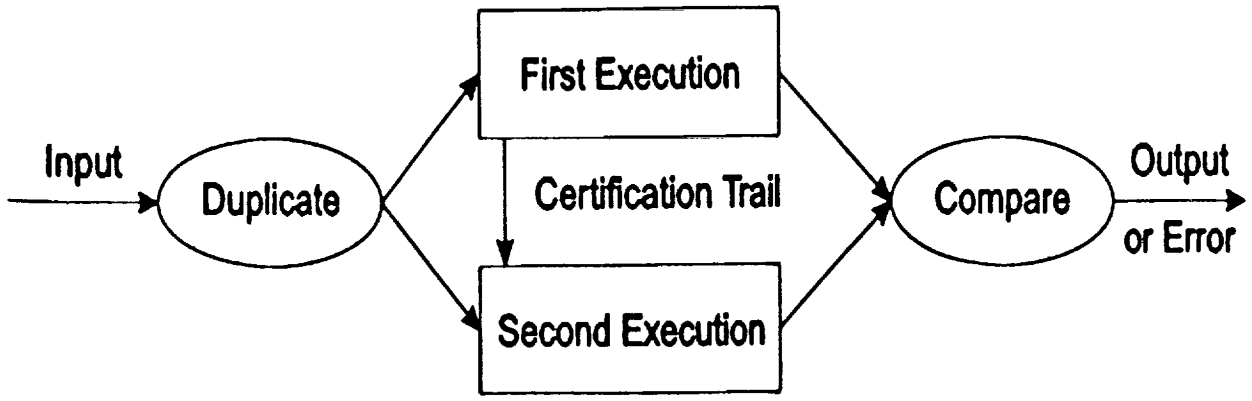


Figure 1: Certification trail method.

Definition 3.1 A problem **P** is formalized as a relation, i.e., a set of ordered pairs. Let **D** be the domain (that is, the set of inputs) of the relation **P** and let **S** be the range (that is, the set of solutions) for the problem. We say an algorithm **A** solves a problem **P** iff for all $d \in \mathbf{D}$ when d is input to **A** then an $s \in \mathbf{S}$ is output such that $(d, s) \in \mathbf{P}$.

Definition 3.2 Let $\mathbf{P} : \mathbf{D} \rightarrow \mathbf{S}$ be a problem. A solution to this problem using a *certification trail* consists of two functions F_1 and F_2 with the following domains and ranges $F_1 : \mathbf{D} \rightarrow \mathbf{S} \times \mathbf{T}$ and $F_2 : \mathbf{D} \times \mathbf{T} \rightarrow \mathbf{S} \cup \{\text{error}\}$. **T** is the set of *certification trails.* The functions must satisfy the following two properties:

(1) for all $d \in \mathbf{D}$ there exists $s \in \mathbf{S}$ and there exists $t \in \mathbf{T}$ such that
$F_1(d) = (s, t)$ and $F_2(d, t) = s$ and $(d, s) \in \mathbf{P}$
(2) for all $d \in \mathbf{D}$ and for all $t \in \mathbf{T}$
either $(F_2(d, t) = s$ and $(d, s) \in \mathbf{P})$ or $F_2(d, t) = \text{error}$.

We also require that F_1 and F_2 be implemented so that they map elements which are not in their respective domains to the error symbol. The definitions above assure that the error-detection capability of the certification-trail approach is similar to that obtained with the simple time-redundancy approach discussed earlier. (That is, if transient hardware faults occur during only one of the executions then either an error will be detected or the output will be correct.) It should be further noted, however, the examples to be considered will indicate that this new approach can also save overall execution time.

Throughout this section we have assumed that our method is implemented with software, however, it is clearly possible to implement the method with assistance from dedicated hardware. The degree of diversity or independence achieved when using certification trails depends on how they are used. A fuller discussion of this and of the relationship between certification trails and other approaches to software fault tolerance is contained in the expanded version of [1].

4 Generalized Priority Queue

Before we present our example algorithms which use certification trails we must discuss the notion of an abstract data type. An abstract data type has a well defined data object or set of data objects, and an abstract data type has a carefully defined finite collection of operations that can be performed on its data object(s). Each operation takes a finite number of arguments (possibly zero), and some but not all operations return answers.

Some of the algorithms presented in the next section use the priority queue abstract data type. In addition, later in this paper the answer-validation problem for two variants of the priority queue are presented. Therefore, we now describe the priority queue. The data consists of a set of ordered pairs. The first element in these ordered pairs is referred to as the item number and the second element is called the key value. Ordered pairs may be added and removed from the set, however, at all times the item numbers of distinct ordered pairs must be distinct. It is possible, though, for multiple ordered pairs to have the same key value. In this paper the item numbers are integers between 1 and n, inclusive. Our default convention is that i is an item number, k is a key value and h is a set of ordered pairs. A total ordering on the pairs of a set can be defined lexicographically as follows: $(i,k) < (i',k')$ iff $k < k'$ or ($k = k'$ and $i < i'$). The abstract data types we will consider support a subset of the following operations.

member(i) returns a boolean value of true if the set contains an ordered pair with item number i, otherwise returns false.

insert(i,k) adds the ordered pair (i,k) to the set. We require that no other pair with item number i be in the set.

delete(i) deletes the unique ordered pair with item number i from the set. We require that a pair with item number i be in the set initially.

changekey(i, k) is executed only when there is an ordered pair with item number i in the set. This pair is replaced by (i, k).

deletemin (or deletemax) returns the ordered pair which is smallest (or largest) according to the total order defined above and deletes this pair. If the set is empty then the token "empty" is returned.

min (or max) returns the ordered pair which is smallest (or largest) according to the total order defined above. If the set is empty then the token "empty" is returned.

predecessor(i) returns the item number of the ordered pair which immediately precedes the pair with item number i in the total order. If there is no predecessor then the token "smallest" is returned. We require that a pair with item number i be in the set initially.

If an operation violates one of the requirements described above then it is considered to be ill-formed. Also, if an operation has the wrong number or type of arguments it is considered to be ill-formed.

Many different types and combinations of data structures can be used to support different subsets of these operations efficiently.

5 Examples of the Certification Trail Technique with Timing Data

In this section we evaluate the use of certification trails for five well-known and significant problems in computer science: the convex hull problem, the minimum spanning tree problem, the shortest path problem, the Huffman tree problem, and the sorting problem. We have implemented algorithms for these problems together with other algorithms which generate and use certification trails.

We provide a full description of the algorithm for the convex hull problem which generates a certification trail and a full description of the algorithm which uses that trail. This material has not appeared in our previous publications [1, 3]. Because of space considerations the discussion of three of the other algorithms is abbreviated, but references to previous publications or technical reports which describe the algorithms more fully are given. The treatment of the sort algorithm is brief but is detailed enough for the interested reader to implement the certification-trail method.

The algorithms we have choosen to implement are not always the algorithms which have the smallest asymptotic time complexity. Often the asymptotically fastest algorithms have large constants of proportionality which make them slower on the data sizes we examined. We modified and used some programs from major software distributions such as quicker-sort from a Berkeley Unix distribution. Other algorithms were based on textbook discussions. It should be stressed here that this research is exploratory and we hope to further increase our corpus of algorithm and data-structure implementations.

5.1 Systems used for timing data

We have collected timing data for the algorithms considered using a Sun workstation, an IBM 386 PC and a Motorola 68000-based system.

The SUN machine utilized was a SPARCstation ELC with 16MB of RAM. The system was run as a standalone machine in single user mode during the timing experiments. Timing data was obtained through the getrusage() system call; the user times are reported in the data.

Some of the algorithms were also run on an MSDOS machine: a Northgate 386/33 with 8MB of RAM. The programs were compiled using DJGPP, DJ Delorie's port of the GNU GCC compiler to MSDOS. This compiler uses a DOS extender to allow programs to run in protected mode; thus nearly all of the 8MB in the machine was available, thereby allowing data sets comparable in size to those used on the Sun. The programs required no change to run under MSDOS, though the data generators required minor modification because the drand48() family of random number generators was not available.

Finally some of the algorithms were also run Motorola M68000-based target system. In addition to the MC68000 microprocessor which served as the cpu, the system was also was comprised of 512K bytes of RAM, 512 bytes of ROM, and numerous I/O modules to support serial and parallel communication. A timer module is also included in the system which uses the 4Mhz clock as a reference so as to provide execution time data for experiments. This system is discussed in Section 10 relative to fault injection experiments.

5.2 Explanation of timing data table entries

Much of the data presented in the timing table is essentially self-explanatory relative to the certication trail technique and algorithms considered. However, a brief discussion of the table entries is appropriate.

The *Basic Algorithm* timing data refers to the execution time of the algorithm in producing the output without the generation of the certification trail. All timing data is listed in seconds.

The *Generate Certif.* timing data refers to the execution time of the algorithm in producing the output with the additional overhead of generating the certification trail.

The *Use Certif.* timing data refers to the execution time of the algorithm in producing the output while using the certification trail.

The *Compare* timing data refers to the time necessary to compare the outputs from both two Basic Algorithm runs or from a Generate Certification Trial run and a Use Certification Trail run. (Obviously, the value of the comparison would be the same in each case.) For the some of the experiments, the data was too small to calculate and is therefore listed as 0.00. In other experiments, the comparison was included in the algorithm execution timing data and therefore is not separately listed.

The *Total Basic* timing data is twice the Basic Algorithm timing data plus the Comparison time (when available) so as to evaluate the classical time-redundancy approach.

The *Total Certif.* timing data is the sum of the Generate Certif. timing data and the Use Certif. data and Comparison data (when available) so as to evaluate the certification trail approach.

The *% Savings* data is percentage of the execution time savings which is gained by using the certification trail method as compared to the classical time redundancy method.

For the Huffman tree data, the input size for the Huffman tree program is the number of nodes. Each node is given a frequency, chosen uniformly from the integers $\{1, 2, \ldots, n\}$. n was selected to be the number of nodes, but in fact it's value does not affect the running time of the algorithm. In order for the algorithm to execute correctly, the sum of the frequencies must not cause an arithmetic overflow. The certification trail method will detect this.

For the minimum spanning tree and shortest path tables, there are two numbers associated with the input size, the first is the number of vertices in the graph, the second the number of edges. A graph with the required edges is selected uniformly from the set of all such graphs, then tested for connectedness. The algorithms will function regardless of connectedness, but allowing graphs that are not connected would introduce undesirable variation in the timing data.

For the convex hull tables, the input size is the number of points in the data set. The points are chosen uniformly from the set of points with integer coordinates between 0 and 30,000.

For the sorting tables, sorting was timed in two ways. The first set of results were obtained by sorting integers. To generate a trail, an integer tag is added to each input integer and an array of these pairs passed to the sort function. After sorting, the "data" integers are placed in an array, and the "tag" integers are placed on the certification trail. Thus, the sort call looks the same as a normal sort function. The time to massage the data in this manner is included in the cost of the call. This method resulted in only a small speedup, because of the overhead involved in massaging the data, and because the sort routine must swap pairs of integers instead of single integers. The integers were chosen uniformly over the range 0 to 1,000,000.

The second method was to sort an array of pointers to structures. In this case it was assumed that the structure contained a field that would serve as the tag. The sort program needed only to fill in this field, and not copy the structures to a second array. This method results in dramatic speedups. Integer keys were used, though a more complex key will work as well (in fact, a more complex key is very likely to increase the speedup achieved).

For the priority queue and generalized priority queue tables, the input size n is the number of commands executed. The item numbers range from 1 to n (ie. there are as many item numbers as there are commands). The commands are not chosen with equal probability, but rather the first $n/2$ are weighted toward insert operations while the second half are weighted toward the other operations, the weightings remaining the same for all runs. This weighting is necessary in order to force a large queue.

The timing data displayed in the tables should be considered not only relative to the overall efficiencies of the certification trail method relative to classical time redundancy but also relative to the probabilistic analysis given in Section 9 in which we show that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the certification trail method has both a a smaller probability of error and a smaller probability of undetected error.

5.3 Convex Hull Example

The convex hull problem is a fundamental one in computational geometry. Our certification trail solution is based on a solution due to Graham [24] which is called Graham's Scan. For basic definitions in computational geometry see the text of Preparata and Shamos[46]. For simplicity in the discussion which follows we will assume the points are in so called general position, e.g., no three points are colinear. It is not hard to remove this restriction.

Definition 5.1 The *convex hull* of a set of points, S, in the Euclidean plane is defined as the smallest convex polygon enclosing all the points. This polygon is unique and its vertices are a subset of the points in S. It is specified by a counterclockwise sequence of its vertices.

Figure 2(c) shows a convex hull for the points indicated by black dots. The algorithm given below constructs the convex hull incrementally in a counterclockwise fashion. Sometimes it is necessary for the algorithm to "backup" the construction by throwing some vertices out and then continuing. The first step of the algorithm selects an "extreme" point and calls it p_1. The next two steps sort the remaining points in a way which is depicted in Figure 2(a). It is not hard to show that after these three steps the points when taken in order, $p_1, p_2, \ldots, p_n$, form a simple polygon; although this polygon may not be convex. It is possible to think of the algorithm as removing points from this simple polygon until it becomes convex. The main FOR loop iteration adds vertices to the polygon under construction and the inner WHILE loop removes vertices from the construction. A point is removed when the angle test performed at line 6 reveals that it is not on the convex hull because it falls within the triangle defined by three other points. A "snapshot" of the algorithm given in Figure 2(b) shows that q_5 is removed from the hull. The angle formed by q_4, q_5, p_6 is less than 180 degrees. This means, q_5 lies within the triangle formed by q_4, p_1, p_6. (Note, $q_1 = p_1$.) In general, when the angle test is performed if the angle formed by q_{m-1}, q_m, p_k is less than 180 degrees then q_m lies within the triangle formed by q_{m-1}, p_1, p_k. Below it will be revealed that this is the main fact that our certification trail relies on. When the main FOR loop is complete the convex hull has been constructed.

Algorithm CONVEXHULL(S)
Input: Set of points, S, in R^2

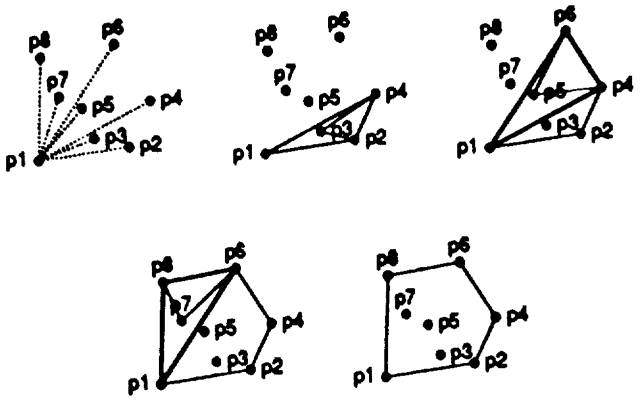


Figure 2: Convex hull example.

```
Output: Counterclockwise sequence of points in $R^2$ which define convex hull of $S$
1 Let $p_1$ be the point with the largest $x$ coordinate (and smallest $y$ to break ties)
2 For each point $p$ (except $p_1$) calculate the slope of the line through $p_1$ and $p$
3 Sort the points (except $p_1$) from smallest slope to largest. Call them $p_2, \ldots, p_n$
4 $q_1 := p_1$; $q_2 := p_2$; $q_3 := p_3$; $m = 3$
5 FOR $k = 4$ to $n$ DO
6    WHILE the angle formed by $q_{m-1}, q_m, p_k$ is $\geq 180$ degrees DO $m := m - 1$ END
7    $m := m + 1$
8    $q_m := p_k$
9 END FOR
10FOR $i = 1$ to $m$ DO, OUTPUT($q_i$) END FOR
END CONVEXHULL
```

First execution: In this execution the code CONVEXHULL is used. The certification trial is generated by adding an output statement within the WHILE loop. Specifically, if an angle of less than 180 degrees is found in the WHILE loop test then the four tuple consisting of q_m, q_{m-1}, p_1, p_k is output to the certification trail. The table below shows the four tuples of points that would be output by the algorithm when run on the example in Figure 2. The points in the table are given the same names as in Figure 2(a). The final convex hull points $q_1, \ldots, q_m$ are also output to the certification trail. Strictly speaking the trail output does not consist of the actual points in R^2. Instead, it consists of indices to the original input data. This means if the original data consists of $s_1, s_2, \ldots, s_n$ then rather than ouput the element in R^2 corresponding to s_i the number i is output. It is not hard to code the program so that this is done.

Point not on convex hull	Three surrounding points
p_5	p_4, p_1, p_6
p_4	p_3, p_1, p_6
p_7	p_6, p_1, p_8

Second execution: Let the certification trail consist of a set of four tuples, $(x_1, a_1, b_1, c_1), (x_2, a_2, b_2, c_2), \ldots, (x_r, a_r, b_r, c_r)$ followed by the supposed convex hull, $q_1, q_2, \ldots, q_m$. The code for CONVEXHULL is not used in this execution. Indeed, the algorithm performed is dramatically different than CONVEXHULL.

It consists of five checks on the trail data.

- First, the algorithm checks for $i \in \{1, \ldots, r\}$ that x_i lies within the triangle defined by a_i,b_i, and c_i.
- Second, the algorithm checks that for each triple of counterclockwise consecutive points on the supposed convex hull the angle formed by the points is less than or equal to 180 degrees.
- Third, it checks that there is a one to one correspondence between the input points and the points in $\{x_1, \ldots, x_r\} \cup \{q_1, \ldots, q_m\}$.
- Fourth, it checks that for $i \in \{1, \ldots, r\}$, a_i, b_i, and c_i are among the input points.
- Fifth, it checks that there is a unique point among the points on the supposed convex hull which is a local extreme point. We say a point q on the hull is a *local extreme* point if its predecessor in the counterclockwise ordering has a strictly smaller y coordinate and its successor in the ordering has a smaller or equal y coordinate.

If any of these checks fail then execution halts and "error" is output. As mentioned above, the trail data actually consists of indices into the input data. This does not unduly complicate the checks above; instead it makes them easier. The correctness and adequacy of these checks must be proven. Because of space limitations we shall not give the proof here.

Time complexity: In the first execution the sorting of the input points takes $O(n \log(n))$ time where n is the number of input points. One can show that this cost dominates and the overall complexity is $O(n \log(n))$.

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.74	0.79	0.11	0.03	1.51	0.93	38.41
20000	1.65	1.75	0.23	0.06	3.36	2.05	39.28
50000	4.64	4.79	0.59	0.14	9.42	5.52	41.40
100000	9.95	10.32	1.19	0.28	20.18	11.79	41.57

Table 1: Huffman Tree on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.09	1.32	0.32	0.10	2.28	1.74	23.68
20000	2.38	2.91	0.63	0.21	4.97	3.75	24.55
50000	7.01	8.80	1.59	0.50	14.52	10.89	25.00

Table 2: Huffman tree on 386/33

It is possible to implement the second execution so that all five checks are done in $O(n)$ time. /papers/certify3/tabdata /papers/certify3/tabdataChecking that a point lies within a triangle is a geometric calculation that can be done in constant time. Comparing the angle formed by three points to 180 degrees can be done in constant time. The third and fourth checks can be done in $O(n)$ because the certification trail contains indices into the input data as described above. The uniqueness of the "local extreme" can also be checked in linear time.

5.4 Minimum Spanning Tree Example

This classic problem has been examined extensively in the literature and an historical survey is given in [25]. Our approach is applied to a variant

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.26	1.29	0.13	0.01	2.53	1.43	43.47
20000	2.71	2.81	0.31	0.01	5.43	3.13	42.35
50000	7.41	7.48	0.70	0.01	14.83	8.19	44.77
100000	15.76	15.87	1.43	0.01	31.53	17.31	45.09

Table 3: Convex Hull on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.79	1.88	0.15	0.01	3.59	2.04	43.18
20000	3.86	4.08	0.31	0.01	7.73	4.40	43.08
50000	10.51	11.16	0.78	0.01	21.03	11.95	43.18
100000	22.40	23.97	1.64	0.01	44.81	25.62	42.83

Table 4: Convex Hull on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
100,1000	0.04	0.05	0.01	0.00	0.08	0.06	25.00
200,2000	0.10	0.12	0.02	0.00	0.20	0.14	30.00
500,5000	0.30	0.31	0.06	0.00	0.60	0.37	38.33
1000,10000	0.68	0.72	0.13	0.00	1.36	0.85	37.50
1500,15000	1.10	1.14	0.19	0.00	2.20	1.33	39.55
2000,20000	1.51	1.58	0.27	0.00	3.02	1.85	38.74
2500,25000	1.97	2.00	0.35	0.00	3.94	2.35	40.36

Table 5: Minimum Spanning Tree on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
100,1000	0.04	0.03	0.01	0.00	0.08	0.04	50.00
200,2000	0.08	0.08	0.02	0.00	0.16	0.10	37.50
500,5000	0.26	0.24	0.06	0.00	0.52	0.30	42.31
1000,10000	0.59	0.56	0.13	0.00	1.18	0.69	41.53
1500,15000	0.93	0.90	0.20	0.00	1.86	1.10	40.86
2000,20000	1.29	1.28	0.28	0.00	2.58	1.56	39.53
2500,25000	1.67	1.65	0.36	0.00	3.34	2.01	39.82

Table 6: Shortest Path on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.23	0.40	0.06	0.01	0.47	0.47	0.00
20000	0.51	0.86	0.13	0.01	1.02	1.00	1.96
50000	1.38	2.35	0.35	0.02	2.78	2.72	2.15
100000	2.96	4.97	0.76	0.04	5.92	5.73	3.20

Table 7: Integer sorting on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.02	1.18	0.14	0.04	2.08	1.36	34.62
20000	2.16	2.49	0.29	0.08	4.40	2.86	35.00
50000	5.67	6.48	0.73	0.22	11.56	7.43	35.73
100000	11.74	13.48	1.57	0.44	23.92	15.49	35.24

Table 8: Integer Sort on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.32	0.33	0.03	0.01	0.65	0.37	43.07
20000	0.71	0.72	0.07	0.01	1.43	0.80	44.05
50000	1.97	1.99	0.18	0.02	3.96	2.19	44.69
100000	4.32	4.37	0.38	0.05	8.69	4.80	44.76

Table 9: Pointer sorting on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	1.08	1.15	0.07	0.03	2.19	1.25	42.92
20000	2.41	2.41	0.16	0.07	4.89	2.64	46.01
50000	6.37	6.38	0.42	0.22	12.96	7.02	45.83
100000	13.29	13.33	0.89	0.43	27.01	14.65	45.76

Table 10: Pointer Sort on 386/33

Size	Basic Algorithm	Generate Certif.	Use Certif.	Compare	Total Basic	Total Certif.	% Saving
10000	0.86	0.83	0.14	0.01	1.73	0.98	43.35
20000	1.92	1.87	0.28	0.01	3.85	2.16	43.89
50000	5.32	5.37	0.69	0.02	10.64	6.08	42.85

Table 11: Data structs on Sun

Size	Basic Algorithm	Generate Certif.	Use Certif.	Total Basic	Total Certif.	% Saving
8	0.075	0.091	0.026	0.151	0.117	28.7
16	0.215	0.248	0.054	0.430	0.302	42.4
32	0.561	0.629	0.111	1.122	0.740	51.6
64	1.330	1.468	0.224	2.660	1.692	57.2
128	3.120	3.398	0.450	6.240	3.848	62.2
256	7.225	7.783	0.903	14.450	8.686	66.4
512	16.270	17.388	1.808	32.540	19.196	69.5

Table 12: Huffman Tree on 68000-based system

Size		Basic	Generate	Use	Total	Total	% Saving
Nodes	Edges	Algorithm	Certif.	Certif.	Basic	Certif.	
10	15	0.053	0.054	0.055	0.106	0.109	-2.5
10	20	0.071	0.072	0.073	0.142	0.145	-1.7
10	25	0.088	0.089	0.090	0.176	0.179	-1.5
50	75	0.320	0.323	0.309	0.639	0.632	1.2
50	100	0.423	0.427	0.400	0.846	0.826	2.3
50	125	0.492	0.496	0.464	0.984	0.960	2.5
100	150	0.652	0.658	0.602	1.305	1.260	3.6
100	200	0.874	0.881	0.789	1.748	1.671	4.6
100	250	1.036	1.045	0.938	2.073	1.983	4.5
500	750	3.588	3.617	3.047	7.176	6.664	7.7
500	1000	4.780	4.817	3.955	9.560	8.772	9.0
500	1250	5.656	5.698	4.717	11.311	10.415	8.6
1000	1500	7.474	7.533	6.115	14.949	13.649	9.5
1000	2000	9.902	9.977	7.919	19.803	17.895	10.7
1000	2500	11.830	11.917	9.517	23.660	21.434	10.4
1500	2250	11.415	11.503	9.157	22.830	20.660	10.5
1500	3000	14.967	15.077	11.802	29.933	26.879	11.4

Table 13: Min Spanning Tree on 68000-based system

of the Prim/Dijkstra algorithm [47, 18] as explicated in [54]. We provide a definition of the problem below. For more information on the graph theoretic terminology used in this problem and others the reader may consult [54, 17].

Definition 5.2 Let $G = (V, E)$ be a graph and let w be a positive rational valued function defined on E. A ***subtree*** of G is a tree, $T(V', E')$, with $V' \subseteq V$ and $E' \subseteq E$. We say T ***spans*** V' and V' is spanned by T. If $V' = V$ then we say T is a spanning tree of G. The weight of this tree is $\sum_{e \in E'} w(e)$. A minimum spanning tree is a spanning tree of minimum weight.

The problem is to input a graph with edge weights and output a minimum spanning tree. The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in [20]. This algorithm however appears to have a large constant of proportionality. Other asymptotically fast algorithms [22] also appear to be handicapped by large constants of proportionality. A fuller discussion of the two algorithms we employ for generation and use of a certification trial is given in [1].

5.5 Shortest Path Example

This is another classic problem which has been examined extensively in the literature. Our approach is applied to a variant of the Dijkstra algorithm [18] as explicated in [54]. We are concerned with the single source problem, i.e., given a graph and a vertex s, find the shortest path from s to v for every vertex v.

The algorithm for this problem which has the fastest asymptotic time complexity uses fusion trees and is given in the same paper which we cited earlier when considering the minimum spanning tree problem[20]. This algorithm however appears to have a large constant of proportionality. Our solution employing the certification trail method is very closely based on the solution we gave for the minimum spanning tree problem [1].

5.6 Huffman Tree Example

This is another old algorithmic problem and one of the original solutions was found by Huffman[30]. It has been used extensively to perform data compression through the design and use of so called Huffman codes. These codes are prefix codes which are based on the Huffman tree and which yield excellent data compression ratios. The tree structure and the code design are based on the frequencies of individual characters in the data to be compressed. Here we are concerned exclusively with the Huffman tree. See [30] for information about the coding application.

Definition 5.3 The Huffman tree problem is the following: Given a sequence of frequencies (positive integers) $f[1], f[2], \ldots, f[n]$, construct a tree with n leaves and with one frequency value assigned to each leaf so that the weighted path length is minimized. Specifically, the tree should minimize the following sum: $\sum_{l_i \in \mathrm{LEAF}} \mathrm{len}(i) f[i]$ where LEAF is the set of leaves, $\mathrm{len}(i)$ is the length of the path from the root of the tree to the leaf l_i, $f[i]$ is the frequency assigned to the leaf l_i.

The method we employ to generate and use a certification trail is detailed in the following technical report [2].

5.7 Sorting Example

This important problem has a massive literature. In this section we will discuss how to apply the certification trail approach to the sorting problem. Let us assume that the sorting algorithm takes as input an array of n elements and outputs an array of n elements. The algorithm is supposed to place the data into non-decreasing order.

To design a certification trail algorithm we must discover the nature of the data that should be included in the certification trail to allow quick computation of the final output sorted array. Suppose that we decide to use the output array itself as the certification trail. We note that it is easy to check that this array is in non-decreasing order by simply performing a single pass over the array. Unfortunately, it is considerably more difficult to make sure that this array contains exactly the same elements as the original input array. Indeed, this problem has a lower bound time complexity of $\Omega(n \log(n))$ in a comparison based model.

Because of this difficulty we use the permutation of the elements defined by the input and output data arrays as the certification trail. To compute this permutation we allocate a new array of size n called permute which is initialized by setting its ith element to i. (Alternatively, we add a new field to pre-existing structures when structures are being sorted.) Each time the sort algorithm exchanges two elements the corresponding elements in the permute array are also exchanged. (If structures are being used then this happens automatically.) This approach works with all sort algorithms which are based on exchanging array elements. The code below shows how the permute array is used to rapidly recompute the final sorted output array and how the permute array itself is checked.

Algorithm SORT USING TRAIL
Input: Arrays indata[1..n] and permute[1..n]
Output: outdata[1..n] containing the data in indata sorted into non-decreasing order

The first part of the algorithm checks that the permute values are in the proper range and constructs the output array.

```
FOR $i := 1$ to $n$ DO
    IF permute[$i$] $> n$ or permute[$i$] $< 1$
        THEN OUTPUT("Error: not a permutation") STOP
        ELSE outdata[$i$] := indata[permute[$i$]]
END FOR
```

The next part of the algorithm checks that the output array is properly ordered.

```
FOR $i := 2$ to $n$ DO
    IF outdata[$i-1$] > outdata[$i$] THEN OUTPUT("Error: decreasing value") STOP
END FOR
```

The final part of the algorithm checks that the permute array defines a proper permutation, i.e., each element is mapped to exactly one element.

```
FOR $i := 1$ to $n$ DO present[$i$] = FALSE END
FOR $i := 1$ to $n$ DO
    IF present[permute[$i$]] = TRUE
        THEN OUTPUT("Error: not a permutation") STOP
        ELSE present[permute[$i$]] := TRUE
END FOR
END SORT USING TRAIL
```

Our experimental work on the Sun was based on a variant of quicksort [26] which is called quickersort [50]. The implementation of this algorithm that we used was provided by a Berkeley UNIX software distribution for the Sun. Our experimental work on the IBM PC was based on a quicksort algorithm implemented as part of a Gnu library of functions.

6 Answer-Validation Problem for Abstract Data Types

The next few sections of this paper are concerned with the answer-validation problem for abstract data types. This kind of problem was originally proposed in [3] and provides a basis for applying the certification-trail method to wide classes of algorithms. Because of space limitations we will not discuss the details of how this can be done.

Below, we define the answer-validation problem. Next, we give two example algorithms for the answer-validation problems. The first algorithm is for a priority queue which allows insert, min and deletemin operations. The second algorithm is for a priority queue which allows insert, min, delete and deletemin operations. In the next section experimental data on the execution times of these algorithms is presented.

For each abstract data type we define an *answer-validation* problem. Intuitively, the answer validation problem consists of checking the correctness of a sequence of supposed answers to a sequence of operations performed on the abstract data type. More formally, the input to the answer-validation problem is a sequence of operations on the abstract data type together with the arguments of each operation. In addition, the sequence contains the supposed answers for each of the operations which return answers. In particular, each supposed answer is paired with the operation that is supposed to return it. Examples of such inputs are given in the columns labelled "Operation" and "Answer" table 15.

The output for the answer-validation problem is the word "correct" if the answers given in the input match the answers that would be generated by actually performing the operations. The output is the word "incorrect" if the answers do not match. It is also useful to allow the output word to say "ill-formed". This output is used if the sequence of operations is ill-formed, e.g., an operation has too many arguments or an argument refers to an inappropriate object.

The answer-validation problem is similar to the idea of an acceptance test which is used in the recovery-block approach [48, 6] to software fault tolerance. The main difference is that an answer-validation problem is dependent upon a sequence of answers, not just an individual answer. Hence, if an incorrect answer appears in the sequence, it may not be detected immediately. It is guaranteed, however, that an incorrect answer will be detected at some point during the processing of the entire sequence. By allowing for this latency in detection, it is possible to create a much more efficient procedure for solving the answer-validation problem.

The most important aspect of the answer-validation problem is that it is often possible to check the correctness of the answers to a sequence of operations much more quickly than actually calculating what the answers should be from scratch. In other words, the answer-validation problem has a smaller time complexity than the original abstract-data-type problem. This speedup is very useful in fault-detection applications.

It is possible to run an answer-validation algorithm for some abstract data type concurrently with some algorithm which uses the abstract data type. The answer-validation algorithm could act as a monitor making sure that all interactions with the abstract data type are handled correctly. This is valuable because many algorithms spend a large fraction of their time operating on abstract data types. Note, the overhead of this monitor is less than the overhead of actually performing the data-type operations a second time.

7 Answer Validation for Priority Queue

We will first consider the priority-queue abstract data type which allows only three operations: insert, min and deletemin. An example of a sequence of such operations appears in table 14. Many different data structures can be used to implement priority queues including heaps [61]; and balanced search trees such as AVL trees [5], red-black trees [27], or b-trees [13]. It is possible to process a sequence of $O(n)$ operations in $O(n\log(n))$ time using the data structures above. Furthermore, there is a lower bound of $\Omega(n\log(n))$ because it is possible to sort using a priority queue. Remarkably, the answer-validation problem can be solved using only $O(n)$ time, as documented below.

The algorithm which we present in this section is the same as that given in [3]. It is necessary to include a description of this algorithm because the algorithm in the next section (which has not appeared before) builds on this algorithm.

Each operation is time-stamped, i.e., the operations are assigned integers sequentially starting with 1 which is easy to do with a counter. The answer-validation algorithm uses a stack called answerstack. The contents of this stack are illustrated in table 14. The top of the stack is on the left in table 14.

Let us consider the kinds of tests that an answer-validation algorithm

Time	Operation	Answer	Insert time	Stack used in validation
1	insert(6,300)			
2	insert(2,404)			
3	insert(3,250)			
4	deletemin	(3,250)	3	(3,250,4)
5	insert(10,248)			
6	insert(12,245)			
7	insert(4,260)			
8	min	(12,245)	6	(12,245,8), (3,250,4)
9	insert(13,140)			
10	insert(5,142)			
11	deletemin	(13,140)	9	(13,140,11), (12,245,8), (3,250,4)
12	deletemin	(5,142)	10	(5,142,12), (12,245,8), (3,250,4)
13	deletemin	(12,245)	6	(12,245,13),(3,250,4)
14	deletemin	(10,248)	5	(10,248,14),(3,250,4)
15	deletemin	(4,260)	7	(4,260,15)

Table 14: Sequence of Priority Queue operations illustrating answer validation algorithm

for a priority queue might perform. Suppose (i,k) is the answer to some min or deletemin operation. Further, suppose (i′,k′) was the answer to a previous min or deletemin operation. If the priority queue is correct then either (i,k)$\geq$(i′,k′) or (i,k) was inserted after the answer (i′,k′) was given. ** multiple insertions possible?* This suggests that the time of insertion for an element and the time of an answer should be recorded and the algorithm below does this. Unfortunately, if an algorithm compares an ordered pair which has been given as an answer against all previous answers then the algorithm complexity is at least $O(m^2)$. To avoid this a stack called the answerstack is used. The answerstack was designed to allow many comparisons to be done implicitly and thus the overall complexity of the many tests is reduced.

Algorithm for Answer Validation for Priority Queue

Input: Sequence of m operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: Array called *inserttime* indexed by item number. Array elements contain either "absent" or a time-stamp. Array called *keyvalue* indexed by item number. Array elements contain either "absent" or a key value. Initially, each element in these two arrays contains "absent". Stack of ordered triples called *answerstack*. Each ordered triple has the following form: first element is an item number, second element is a key value, and third element is a time-stamp. answerstack is initially empty.

First phase: In this phase we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): If inserttime[i]$\neq$"absent" then output "ill-formed" and stop. Otherwise, let inserttime[i] = currenttime and let keyvalue[i]=k.

min (i,k): (where (i,k) is the supposed answer to the deletemin operation.) If inserttime[i]="absent" or keyvalue[i]$\neq$k then output "ill-formed" and stop.

Otherwise, let (i′,k′) be the item number and key value of the triple on the top of answerstack (if there is one). Repeatedly pop the stack until (i,k)<(i′,k′) or until answerstack is empty.

If answerstack is empty then push the triple (i,k,currenttime) onto answerstack and process the next priority queue operation.

If answerstack is non-empty then let the top element be (i′,k′,answertime′). If inserttime[i]<answertime′ then output "incorrect" and stop. Otherwise, push the triple (i,k,currenttime) onto answerstack and process the next priority queue operation.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those described for the min operation. However, just before processing the next priority queue operation, let inserttime[i]="absent" and let keyvalue[i]="absent".

Second phase: In this phase we operate on the items which have been inserted but have never been deleted.

Scan the array inserttime and for each item number for which inserttime[i]≠"absent" construct an ordered triple (i,keyvalue[i],inserttime[i]). Call this set of ordered triples remainders.

Use a bucket sort to sort the triples in remainders by their time-stamps, i.e., the third element of the ordered triple.

Merge the triples in remainders together with the triples in answerstack so that they are all ordered by their time-stamps, i.e., the third element of the ordered triple.

Scan the combined triples to determine if there exist two triples which satisfy the following: inserttime[i]<answertime′ and (i,keyvalue[i])<(i′,k′); where one triple is from remainders and has the form (i,keyvalue[i],inserttime[i]) and where the other triple is from answerstack and has the form (i′,k′,answertime′);

If these two triples exist then output "incorrect" and stop. Otherwise output "correct" and stop.

Theorem 7.1 *The algorithm for answer validation of the priority queue abstract data type is correct.*

Theorem 7.2 *The answer validation algorithm for priority queue has a time complexity of $O(n)$ for processing a sequence of $O(n)$ operations.*

For proofs of these theorems see [3].

8 Answer Validation for Generalized Priority Queue

We next consider the priority-queue abstract data type which allows four operations: insert, min, deletemin, and delete. An example of a sequence of such operations appears in table 15.

The algorithm to solve the validation problem for this data type is an enhanced version of the algorithm given above for the data type which allowed only three priority-queue operations.

Algorithm for Answer Validation for Generalized Priority Queue

Input: Sequence of m operations together with arguments and supposed answers for the priority-queue data type.
Output: "correct", "incorrect" or "ill-formed"

Declarations: All the declartions used in the earlier algorithm are used again. In addition, a collection of sets called *stacksets* are used. Each set in stacksets consists of a set of item numbers (possibly the empty set). There is a one-to-one correspondence between the sets in stacksets and the ordered triples in answerstack. Initially, answerstack consists solely of the ordered triple (0,-∞,-1). Also initially, stacksets contains exactly one set which is the empty set and which corresponds to (0,-∞,-1).

First phase: In this phase we process each operation as it appears serially using the following rules:

Let currenttime refer to the time-stamp of the operation being processed.

insert(i,k): Perform the same actions as those given earlier for the insert operation. In addition, add the item number i to the set in stacksets corresponding to the top element in answerstack.

min (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those given earlier for the min operation. In addition, if any elements are popped off of answerstack then the sets in stacksets corresponding to these elements are unioned together to form a new set. This new set is placed in correspondence with the new top element of answerstack.

deletemin (i,k): (where (i,k) is the supposed answer to the deletemin operation.) Perform the same actions as those given for the min operation described immediately above. In addition, remove the item number i from the set in stacksets which contains it. Further, before processing

Time	Operation	Answer	Insert time	Stack used in validation		
1	insert(5,310)					(0,-∞,-1) {5}
2	insert(6,210)					(0,-∞,-1) {5,6}
3	insert(8,280)					(0,-∞,-1) {5,6,8}
4	min	(6,210)	2			(6,210,4) {5,6,8}
5	insert(9,190)					(6,210,4) {5,6,8,9}
6	min	(9,190)	5		(9,190,6),	(6,210,4) {5,6,8,9}
7	insert(2,275)				(9,190,6), {2},	(6,210,4) {5,6,8,9}
8	delete(8)		3		(9,190,6), {2},	(6,210,4) {5,6,9}
9	insert(12,170)				(9,190,6), {2,12},	(6,210,4) {5,6,9}
10	insert(14,400)				(9,190,6), {2,12,14},	(6,210,4) {5,6,9}
11	deletemin	(12,170)	9	(12,170,11),	(9,190,6), {2,14},	(6,210,4) {5,6,9}
12	insert(3,290)			(12,170,11), {3},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
13	insert(7,330)			(12,170,11), {3,7},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
14	insert(15,200)			(12,170,11), {3,7,15},	(9,190,6), {2,14},	(6,210,4) {5,6,9}
15	delete(9)		5	(12,170,11), {3,7,15},	(9,190,6), {2,14},	(6,210,4) {5,6}
16	deletemin	(15,200)	14		(15,200,16), {2,3,7,14},	(6,210,4) {5,6}
17	delete(7)		13		(15,200,16), {2,3,14},	(6,210,4) {5,6}
18	deletemin	(6,210)	2			(6,210,18) {2,3,5,14}
19	delete(14)		10			(6,210,18) {2,3,5}

Table 15: Sequence of Priority Queue operations illustrating answer validation algorithm

the next priority queue operation, let inserttime[i]="absent" and let keyvalue[i]="absent".

delete(i): If inserttime[i]="absent" or keyvalue[i]="absent" then output "ill-formed" and stop.

Otherwise, let inserttime=inserttime[i] and let k=keyvalue[i]. Next, let inserttime[i]="absent" and let keyvalue[i]="absent".

Now, let (i′,k′,answertime′) be the ordered triple which corresponds to the set in stacksets containing item number i. Next, remove item number i from the set which contains it.

If answertime′>inserttime and (i,k)>(i′,k′) then output "incorrect" and stop.

If answertime′>inserttime and (i,k)≤(i′,k′) then process the next priority queue operation.

If (i′,k′,answertime′) is the top element of answerstack then process the next priority queue operation.

Let (i″,k″,answertime″) be the element immediately above (i′,k′,answertime′) on answerstack.

If (i,k)>(i″,k″) then output "incorrect" and stop. Otherwise, process the next priority queue operation.

Second phase: In this phase we operate on the items which have been inserted but have never been deleted.

For this phase one performs the same operations as the second phase described earlier.

Theorem 8.1 *The algorithm above for answer validation of the priority queue abstract data type is correct.*

Theorem 8.2 *The answer validation algorithm above for priority queue has a time complexity of $O(n)$ for processing a sequence of $O(n)$ operations.*

Proofs omitted for space reasons. It is clear that a priority queue with operations insert, delete, max, deletemax can also be validated in linear time by changing the appropriate signs in the algorithm above.

Definition 8.3 Consider a sequence of priority queue operations together with arguments and supposed answers. The sequence may contain the following operations: insert, delete, min, deletemin, max, and deletemax.

Based on this sequence we define a new sequence called a *minimum sequence.* This sequence differs from the original sequence as follows: Each max operation and answer pair is removed from the sequence. Each deletemax operation and answer pair is replaced by a delete(i) operation where i is the item number given in the answer to the deletemax operation. Each other operation remains the same.

We also define a *maximum sequence.* This sequence differs from the original sequence as follows: Each min operation and answer pair is removed from the sequence. Each deletemin operation and answer pair is replaced by a delete(i) operation where i is the item number given in the answer to the deletemin operation. Each other operation remains the same.

Theorem 8.4 *Consider a sequence of priority queue operations together with arguments and supposed answers. The sequence may contain the following operations: insert, delete, min, deletemin, max, and deletemax. The answers given for this sequence are correct if and only if the answers given for the corresponding minimum and maximum sequences are both correct.*

This theorem allows us to define an algorithm which solves the answer-validation problem for general priority queue.

9 Probabilistic Model

We will now present a simple probabilistic model with accompanying analysis which will permit a comparison between of our certification-trail method and the classical time-redundancy approach [32, 52]. The analysis shows that when the certification-trail method has a smaller execution time than the time-redundancy approach it yields strictly superior performance. This means the certification trail method has both a a smaller probability of error and a smaller probability of undetected error. Surprisingly, the analysis also reveals the intriguing result that the certification-trail method often can display superior performance even when the method has the same execution time or a longer execution time than the time-redundancy approach. This superior behavior stems from the typical assymetry of the execution times of the first and second executions in the certification-trail method.

We make the following assumptions.

i. Errors are distributed exponentially with parameter λ.

ii. If errors occur during only one phase of the execution, then they are detected.

iii. If errors occur in both phases of an execution they are not detected.

For solutions to a problem with run times a and b, we therefore have:

$$\begin{aligned}
Pr\{correct\} &= e^{-\lambda(a+b)} \\
Pr\{detected\} &= e^{-\lambda a}(1-e^{-\lambda b}) + e^{-\lambda b}(1-e^{-\lambda a}) \\
&= e^{-\lambda a} + e^{-\lambda b} - 2e^{-\lambda(a+b)} \\
Pr\{undetected\} &= (1-e^{-\lambda a})(1-e^{-\lambda b}) \\
&= 1 - e^{-\lambda a} - e^{-\lambda b} + e^{-\lambda(a+b)} \\
&= 1 - Pr\{correct\} - Pr\{detected\}
\end{aligned}$$

Given two solutions for a problem, we say that the first is strictly superior to the second iff:

$$\begin{gathered}
Pr_1\{correct\} \geq Pr_2\{correct\} \quad and \quad Pr_1\{undetected\} < Pr_2\{undetected\} \\
or \\
Pr_1\{correct\} > Pr_2\{correct\} \quad and \quad Pr_1\{undetected\} \leq Pr_2\{undetected\}
\end{gathered}$$

This implies that the run time of the first solution is no greater than that of the second solution.

Observation 1 *Suppose there are two solutions (using certification trails) to a problem, such that each solution runs in two phases, and the combined run times of phases is the same for both solutions. Then the solution with the greater time imbalance between phases is strictly superior.*

Proof: Let $2a =$ the run time . Let $a + b$ the run length of the first phase of the first method, and $a + c$ be the run time of the first phase of the second method. Then the second phases have times of $a - b$ and $a - c$ respectively. Assume $b < c$.

Since the total run time is the same for both solutions, we have $Pr_1\{correct\} = Pr_2\{correct\} = e^{-\lambda 2a}$, so we need only show that $Pr_1\{detected\} < Pr_2\{detected\}$, ie.

$$\begin{aligned}
e^{-\lambda(a+b)}(1 - e^{-\lambda(a-b)}) + e^{-\lambda(a-b)}(1 - e^{-\lambda(a+b)}) &< e^{-\lambda(a+c)}(1 - e^{-\lambda(a-c)}) + e^{-\lambda(a-c)}(1 - e^{-\lambda(a+c)}) \\
e^{-\lambda(a+b)} + e^{-\lambda(a-b)} &< e^{-\lambda(a+c)} + e^{-\lambda(a-c)} \\
e^{-\lambda b} + e^{\lambda b} &< e^{-\lambda c} + e^{\lambda c}
\end{aligned}$$

Setting $x = e^{\lambda b}$ and $y = e^{\lambda c}$ we want

$$\begin{aligned}
x + \frac{1}{x} &< y + \frac{1}{y} \qquad \text{for } 1 \leq x < y \\
\frac{1}{x} - \frac{1}{y} &< y - x \\
\frac{y - x}{xy} &< y - x
\end{aligned}$$

Corollary 1 *Given a basic algorithm for a problem, a certification trail method is superior to running the basic algorithm twice if the total run time is no greater than twice that of the basic algorithm.*

The above statements apply to the situation of a single execution of a solution. A more interesting case is to iterate the solution until no errors are reported, that is we either arrive at the correct answer, or have undetected errors.

Let $Pr_{iter}\{correct\}$ denote the probability of finding a correct solution in the iterated scheme and $Pr_{iter}\{undetected\}$ denote the probability of accepting an incorrect run.

Note that we repeat a run only when errors are detected, so if we obtain the correct answer on the $n-th$ run, the previous $n-1$ runs must have resulted in detected errors. Thus it is clear that:

$$\begin{aligned}
Pr_{iter}\{correct\} &= Pr\{correct\} \sum_{i=0}^{\infty} Pr\{detected\} \\
&= \frac{Pr\{correct\}}{1 - Pr\{detected\}}
\end{aligned}$$

Similarly,

$$Pr_{iter}\{undetected\} = \frac{\mathrm{Pr}\{undetected\}}{1 - Pr\{detected\}}$$

For the iterated scheme, we will say that one method is superior to another if the probability of obtaining the correct answer is larger. Obviously if a method is superior in the single run sense, it must be superior in the iterated case. However it is possible for one method to be superior to another in the iterated scheme, but not in the single run scheme. This means that a certification trail method may be better than running a basic algorithm twice, even if the certification trail takes longer to run!

Suppose we have a basic algorithm A with running time a for a particular problem, and a certification trail method with phases running in times b and c. Given b, how small must c be, for the certification trail to be superior?

We require:

$$\begin{aligned}
\frac{Pr_{cert}\{correct\}}{1 - Pr_{cert}\{detected\}} &> Pr_{basic}\{correct\}1 - Pr_{basic}\{detected\} \\
\frac{e^{-\lambda(b+c)}}{1 - e^{-\lambda b} - e^{-\lambda c} + 2e^{-\lambda(b+c)}} &> \frac{e^{-\lambda 2a}}{1 - 2e^{-\lambda a} + 2e^{-\lambda 2a}} \\
e^{-\lambda(b+c)} - 2e^{-\lambda(a+b+c)} &> e^{-\lambda 2a} - e^{-\lambda(2a+b)} - e^{-\lambda(}2a + c) \\
e^{-\lambda c}(e^{-\lambda b} + e^{-\lambda 2a} - 2e^{-\lambda(a+b)}) &> e^{-\lambda 2a}(1 - e^{-\lambda b})
\end{aligned}$$

Note that $b > a$, so $e^{-\lambda b} + e^{-\lambda 2a} - 2e^{-\lambda(a+b)}$ must be positive. So,

$$\begin{aligned}
e^{-\lambda c} &> \frac{e^{-\lambda 2a}(1 - e^{-\lambda b})}{e^{-\lambda b} + e^{-\lambda a}(1 - e^{-\lambda b})} \\
c &< -\frac{1}{\lambda}\ln \frac{e^{-\lambda 2a}(1 - e^{-\lambda b})}{e^{-\lambda b} + e^{-\lambda 2a}(1 - e^{-\lambda b})}
\end{aligned}$$

Since the argument to ln is strictly between 0 and 1, c is well defined for any choice of a, b, and λ.

In addition to the probability of correctness, we would like to know the expected running time using the iterated approach. Fortunately, this is easily determined.

Our probability of stopping on a particular execution is $Pr\{correct\} + Pr\{undetected\} = 1 - Pr\{detected\}$. Therefore with that probability we stop on the first execution, with probability $Pr\{detected\}(1 - Pr\{detected\})$ we stop on the second execution, and in general we stop on the nth execution with probability $(1 - Pr\{detected\})(Pr\{detected\})^{n-1}$. This gives us an expected number of iterations of,

$$(1 - Pr\{detected\}) \sum_{i=0}^{\infty} (i+1) Pr\{detected\}^i$$

Now,

$$\sum_{i=0}^{\infty} (i+1) x^i = \frac{1}{(1-x)^2}$$

so we find that the expected number of iterations is,

$$\frac{1}{1 - Pr\{detected\}}$$

Multiplying the run time of a single iteration will give us the expected running time.

Table 16 shows information for running a basic algorithm. The run time of a basic algorithm is set to 1 unit of time. The basic algorithm is run twice and the results compared, we assume that comparator is fast enough so that the time it takes is negligible (this is justified by the experimental results), and that it is error free. We compute

i. Prob. Correct - The probability that both phases are error free.

ii. Prob. Detected - The probability that exactly on of the phases contains an error.

iii. Prob. Undetected - The probability that both of the phases contain errors.

iv. Iterated Prob Correct - If the basic algorithm is iterated (each iteration is two runs), this is the probability that the terminating result is correct.

v. Expected Runtime - The expected run time of the algorithm in the iterated model. For the basic algorithm this is twice the expected number of iterations.

Tabel 17 illustrates the "breakeven" point for the certification trail approach. Given a value for λ and a run time b of a trail generating algorithm. The breakeven point for the run time of the trail checking algorithm is the

λ	Basic Algorithm	Prob Correct	Prob. Detected	Prob. Undetected	Iter. Prob. Correct	Expected Runtime
0.01	1	0.980199	0.019702	0.000099	0.999899	2.040197
0.10	1	0.818731	0.172213	0.009056	0.989060	2.416081
1.00	1	0.135335	0.465088	0.399576	0.253005	3.738935

Table 16: Balanced Probabilites

λ	Generate Trail	Breakeven Trail Checker
0.01	1.10	0.909050
0.01	1.50	0.666111
0.01	2.00	0.498750
0.10	1.10	0.908683
0.10	1.50	0.661128
0.10	2.00	0.487505
1.00	1.10	0.905504
1.00	1.50	0.614107
1.00	2.00	0.379885

Table 17: Certification checker breakeven points

point at which the iterated probability of correctness is the same as for the "basic" algorithm (which has a run time of 1).

Run times less than this will result in the certification trail solution being superior. It is interesting to notice that in the total length of the solution at the breakeven point is greater than 2, ie. running the basic algorithm twice.

Table 18 is similar to the first one, the difference being that this examines the behavior of certification trail methods for different run times of the two phases. The meaning of the other columns is identical to the meaning in the table for basic algorithms. Of interest is the row $\lambda = 1.00, b = 1.50, c = 0.25$. Compare this with the first table for $\lambda = 1.00$. We see that the certification method has a greater probability of being correct for a single run and the total run time is shorter than twice the basic algorithm, yet the expected iterated run time is larger!

10 Fault Injection Experiments

A series of hardware fault injection experiments have been conducted during which combinations of the address, data, and control lines of a Motorola

C-2

λ	Generate Certif.	Use Certif.	Prob Correct	Prob. Detected	Prob. Undetected	Iter. Prob. Correct	Expected Runtime
0.01	1.10	0.25	0.986591	0.013382	0.000027	0.999972	1.368311
0.01	1.10	0.50	0.984127	0.015818	0.000055	0.999945	1.625716
0.01	1.10	0.75	0.981670	0.018248	0.000082	0.999917	1.884387
0.01	1.50	0.25	0.982652	0.017311	0.000037	0.999962	1.780827
0.01	1.50	0.50	0.980199	0.019727	0.000074	0.999924	2.040248
0.01	1.50	0.75	0.977751	0.022138	0.000111	0.999886	2.300937
0.01	2.00	0.25	0.977751	0.022199	0.000049	0.999949	2.301082
0.01	2.00	0.50	0.975310	0.024591	0.000099	0.999899	2.563028
0.01	2.00	0.75	0.972875	0.026977	0.000148	0.999848	2.826245
0.10	1.10	0.25	0.873716	0.123712	0.002572	0.997065	1.540590
0.10	1.10	0.50	0.852144	0.142776	0.005080	0.994074	1.866490
0.10	1.10	0.75	0.831104	0.161369	0.007527	0.991025	2.205976
0.10	1.50	0.25	0.839457	0.157104	0.003439	0.995920	2.076175
0.10	1.50	0.50	0.818731	0.174476	0.006793	0.991771	2.422703
0.10	1.50	0.75	0.798516	0.191419	0.010065	0.987553	2.782653
0.10	2.00	0.25	0.798516	0.197008	0.004476	0.994426	2.802021
0.10	2.00	0.50	0.778801	0.212359	0.008841	0.988776	3.174033
0.10	2.00	0.75	0.759572	0.227330	0.013098	0.983049	3.559087
1.00	1.10	0.25	0.259240	0.593191	0.147568	0.637254	3.318513
1.00	1.10	0.50	0.201897	0.535609	0.262495	0.434755	3.445370
1.00	1.10	0.75	0.157237	0.490763	0.352000	0.308770	3.632888
1.00	1.50	0.25	0.173774	0.654383	0.171843	0.502793	5.063409
1.00	1.50	0.50	0.135335	0.558990	0.305674	0.306876	4.535047
1.00	1.50	0.75	0.105399	0.484698	0.409903	0.204539	4.366374
1.00	2.00	0.25	0.105399	0.703338	0.191263	0.355283	7.584379
1.00	2.00	0.50	0.082085	0.577696	0.340219	0.194374	5.919905
1.00	2.00	0.75	0.063928	0.479846	0.456226	0.122902	5.286897

Table 18: Unbalanced Probabilites

M68000-based target system were pulsed with selected signals of various types and durations while in the process of executing algorithms. In addition to the MC68000 microprocessor which served as the cpu, the target also was comprised of 512K bytes of RAM, 512 bytes of ROM, and numerous I/O modules to support serial and parallel communication. A timer module is also included in the target which uses the 4Mhz clock as a reference so as to provide execution time data for experiments. Finally, a simple operating system is resident in the ROM of the target which provides programming and operational support.

The fault injection testbed on which these experiments were performed is illustrated as the configuration shown in Figure 3. In addition to the target system, the fault injection testbed contains other modules which perform the fault injection and data acquisition functions under instruction from the Operations Control Console. By means of RS232C, SCSI, and GPIB interfaces, a Macintosh IICX serves as the Operations Control Console permitting fault injections to be precisely executed and resulting error data to be recorded for later analysis by a SUN SPARCstation 2.

The Operations Control Console also communicates over a VMEbus with the Testbed Controller which is responsible for overall testbed operation. The primary component of the Testbed Controller is a MC68030-based unit with 8 Mbytes of SRAM to store error data from fault injection runs as communicated to it over the VMEbus from the data acquisition module. The Testbed Controller also is similarly responsible for the operations of the fault injection module as determined by commands from the Operations Control Console.

The fault injection module and the data acquisition module have access via edge connector pins to the lines of the target system selected for injection and monitoring, respectively. The fault injections are precisely triggered after some operator determined delay following the appearance of an operator pre-selected set of bits on either the address lines of the address bus or the data lines of the data bus. Similarly, the durations and frequencies of the injections are also controlled by the operator. The injections emanate from a bank of programmable function generators included in the fault injection module. The precision with which fault conditions are triggered and injected permits the resulting error conditions which are observed to be repeated (if necessary) for further monitoring/analysis. The data acquisition module is also triggered by the same address or data bits that activated the fault injection module. However, there is no delay associated with the data acquisition function; transfer of the signals on the lines being monitored by the data acquisition module to the memory of the Testbed Controller commences immediately the data acquisition module's activation. Data monitored by the data acquisition module is transmitted directly onto VME bus and then written into the SRAM of the Testbed Controller.

10.1 Fault injection and error classification in MC68000 target system

To generally indicate the details of the fault injection experiments using the target system, the injections and resulting errors can be summarized and displayed at the Operations Control Console as illustrated in Figure 4.

In the example illustrated in Figure 4, the trigger address for the injection was selected by the operator to be address 1019E (hexadecimal) in the first version of Huffman tree program which was to generate both the output and the certification trail. The actual injection consisted of holding the lower 4 bits of the data bus at logical zero starting 2 microseconds after the recognition of the trigger address by the fault injection module and then maintaining the logical zero on these lines for various durations lasting between 1 and 10 microseconds. For this example, we see that 5 distinct error conditions resulted depending on the duration of the injection. The details of data errors classified as type 2 and type 3 are beyond the scope of this discussion. Suffice it to say that each such type of data error observed in this particular experimental run could be interpreted as an inconsistent labeling of nodes in the certification trail passed to the second program. In each case, however, it should be emphasized that the execution of the second program utilizing the certification trail detected the error. The other errors listed in Figure 4 can be categorized as address errors and illegal instructions.

Our purpose in presenting Figure 4 is only to illustrate an example of a fault injection run with a subsequent error analysis and classification. In general, the errors resulting from injections into the target system could be classified as:

- No error.
- Data output errors
- Certification trail errors
- Addressing errors
- Data value errors

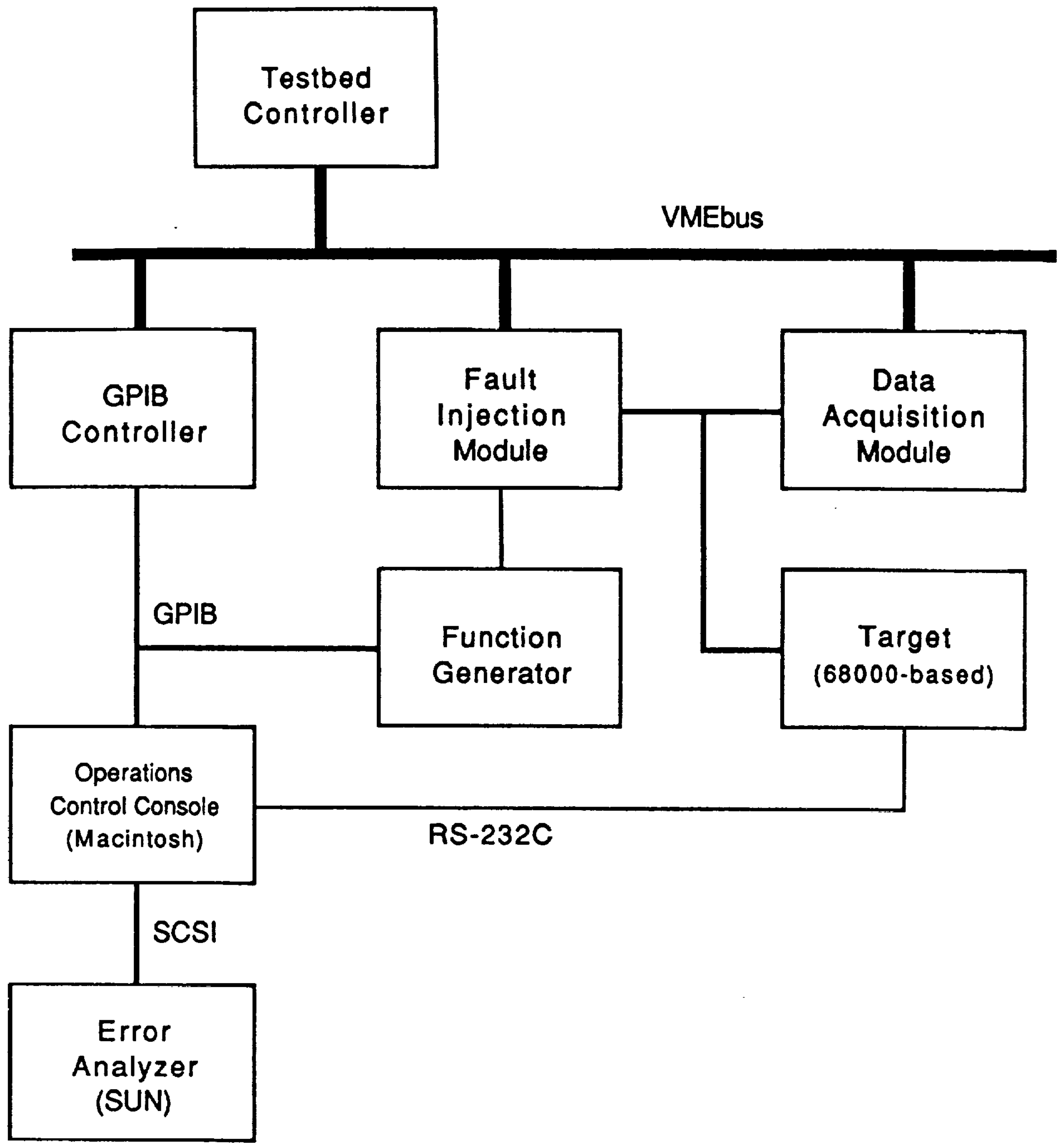


Figure 3: Hardware fault injection testbed for MC68000-based target system

```
----------------------------------------------------------------------------
Fault              Delay   Width    Error
----------------------------------------------------------------------------
XXXX XXX0          0 us    .1 us    no error
                           .2       no error
                           .3       no error
                           .4       ADDR TRAP ERROR
                           .5       ADDR TRAP ERROR
                           1        ADDR TRAP ERROR
                           2        ADDR TRAP ERROR
                           4        ADDR TRAP ERROR
                           4.5      ADDR TRAP ERROR
                           5        data_error.2
                                    Certification Error: Inconsistent Labels
                           5.5      data_error.2
                                    Certification Error: Inconsistent Labels
                           6        data_error.3
                                    Certification Error: Inconsistent Labels
                           7        data_error.3
                                    Certification Error: Inconsistent Labels
                           8        data_error.3
                                    Certification Error: Inconsistent Labels
                           9        data_error.3
                                    Certification Error: Inconsistent Labels
                           10       ILLEGAL INSTRUCTION
----------------------------------------------------------------------------
```

Figure 4: Example of output displayed at Operations Control Console for fault injection run for Huffman tree algorithm program

- Halt generated
- Reset generated
- Non-termination of program
- Program mutilation

Currently, the testbed tools are being expanded to produce automated injections using suites of fault conditions on the target system.

Software fault injection experiments were also performed in which instructions, data, and stack contents were modified using both the Sun Sparcstation and the 386 machine with which the previously detailed timing data was collected. The details of these fault injection experiments will be presented in a companion document.

11 Concluding Discussion

This paper experimentally supplements two previous *FTCS* papers [1, ?] which theoretically explore the new fault tolerance technique referred to as the certification trail method. We have presented experimental timing data which illustrates the advantages of the certification trail technique over classical time redundancy. We have further presented analytical results which further support the significance of the certfication trail technique.

References

[1] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Digest of the 1990 Fault Tolerant Computing Symposium*, pp. 423-431, IEEE Computer Society Press, 1990.

[2] Sullivan, G.F., and Masson, G.M., "Using certification trails to achieve software fault tolerance," *Department of Computer Science Technical Report JHU 89/26*, Johns Hopkins University, Baltimore, Maryland, 1989.

[3] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Digest of the 1991 Fault Tolerant Computing Symposium*, pp. 240-247, IEEE Computer Society Press, 1991.

[4] Sullivan, G.F., and Masson, G.M., "Certification trails for data structures," *Department of Computer Science Technical Report JHU 90/17*, Johns Hopkins University, Baltimore, Maryland, 1990.

[5] Adel'son-Vel'skii, G. M., and Landis, E. M., "An algorithm for the organization of information", *Soviet Math. Dokl.*, pp. 1259-1262, 3, 1962.

[6] Anderson, T., and Lee, P., *Fault tolerance: principles and practices*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[7] Andrews, D., "Software fault tolerance through executable assertions," *Rec. 12th Asilomar Conf. Circuits, Syst., Comput.*, pp. 641-645, 1978, Nov. 6-8.

[8] Andrews, D., "Using excutable assertions for testing and fault tolerance," *Dig. 9th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 102-105, 1979, June 20-22.

[9] Avizienis, A., "Fault tolerance by means of external monitoring of computer systems," *Proceedings of the 1981 National Computer Conference*, pp. 27-40, AFIPS Press, 1980

[10] Avizienis, A., "Design diversity - the challenge of the eighties," *Digest of the 1982 Fault Tolerant Computing Symposium*, pp. 44-45, IEEE Computer Society Press, 1982.

[11] Avizienis, A., and Kelly, J., "Fault tolerance by design diversity: concepts and experiments," *Computer*, vol. 17, pp. 67-80, Aug., 1984.

[12] Avizienis, A., "The N-version approach to fault tolerant software," *IEEE Trans. on Software Engineering*, vol. 11, pp. 1491-1501, Dec., 1985.

[13] Bayer, R., and McCreight, E., "Organization of large ordered indexes", Acta Inform., pp 173-189, 1, 1972.

[14] Blough, D., and Masson, G., "Performance analysis of a generalized concurrent error detection procedure," *IEEE Trans. on Computers* vol. 39, Jan., 1990.

[15] Blum, M., and Kannan, S., "Designing programs that check their work", *Proceedings of the 1989 ACM Symposium on Theory of Computing*, pp. 86-97, ACM Press, 1989.

[16] Chen, L., and Avizienis A., "N-version programming: a fault tolerant approach to reliability of software operation," *Digest of the 1978 Fault Tolerant Computing Symposium*, pp. 3-9, IEEE Computer Society Press, 1978.

[17] Cormen, T. H., and Leiserson, C. E., and Rivest, R. L., *Introduction to Algorithms* McGraw-Hill, New York, NY, 1990.

[18] Dijkstra, E. W., "A note on two problems in connexion with graphs," *Numer. Math. 1*, pp. 269-271, Sept., 1959.

[19] Eifert, J.B., and Shen, J.P., "Processor monitoring using asynchronous signatured instruction streams," *Dig. 14th Int. Conf. Fault-Tolerant Comput.*, pp. 394-399, 1984, June 20-22.

[20] Fredman, M. L., and Willard, D. E., "Trans-dichotomous algorithms for minimum spanning trees and shortest paths," *Proc. 31st IEEE Foundations of Computer Science*, pp. 719-725,1990.

[21] Fredman, M. L., and Saks, M. E., "The cell probe complexity of dynamic data structures," *Proc. 21st ACM Symp. on Theo. Comp. 1989*, pp. 109-122, 2, 1986.

[22] Gabow, H. N., Galil, Z., Spencer, T., and Tarjan, R. E., "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," *Combinatorica 6*, pp. 109-122, 2, 1986.

[23] Gabow, H. N., and Tarjan, R. E., "A linear-time algorithm for a special case of disjoint set union," *J. of Comp. and Sys. Sci.*, 30(2), pp. 209-221, 1985.

[24] Graham, R. L., "An efficient algorithm for determining the convex hull of a planar set", *Information Processing Letters*, pp. 132-133, 1, 1972.

[25] Graham, R. L., and Hell, P., "On the history of the minimum spanning tree problem," *Ann. Hist. Comput.*, pp. 43-47, Jan., 1985.

[26] Hoare, C. A. R., "Quicksort," *Computer Journal*, pp. 10-15, 5(1), 1962.

[27] Guibas, L. J., and Sedgewick, R., "A dichromatic framework for balanced trees", *Proceedings of the Nineteenth Annual Symposium on Foundations of Computing*, pp. 8-21, IEEE Computer Society Press, 1978.

[28] Gunneflo, U., Karlsson, J., and Torin, J., "Evaluation of error detection schemes for using fault injection by heavy-ion radiation," *Dig. of the 1989 Fault Tolerant Computing Symposium*, pp. 340-347, June, 1989.

[29] Huang, K.-H., and Abraham, J., "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. on Computers*, pp. 518-529, vol. C-33, June, 1984.

[30] Huffman, D., "A method for the construction of minimum redundancy codes", *Proc. IRE*, pp 1098-1101, 40, 1952.

[31] Iyengar, V.S. and Kinney, L.L., "Concurrent fault detection in microprogrammed control units," *IEEE Trans. Comput.*, vol. C-34, pp. 810-821, Sept. 1985.

[32] Johnson, B., *Design and analysis of fault tolerant digital systems* Addison-Wesley, Reading, MA, 1989.

[33] "Fault tolerant FFT networks," *Dig. of the 1985 Fault Tolerant Computing Symposium*, June, 1985.

[34] Kane, J.R. and Yau, S.S., "Concurrent software fault detection," *IEEE Trans. Software Eng.* , vol. SE-1, pp. 87-99, March 1975.

[35] Komlòs, J., "Linear verification for spanning trees", *Proceedings of the 1984 Symposium on Foundations of Computing*, pp. 201-206, IEEE Computer Society Press, 1984.

[36] Lee, Y.H. and Shin, K.G., "Design and evaluation of a fault-tolerant multiprocessor using hardware recovery blocks," *IEEE Trans. Comput.*, vol. C-33, pp. 113-124, Feb. 1984.

[37] Lu, D., "Watchdog processor and structural integrity checking," *IEEE Trans. Comput.*, vol. C-31, pp. 681-685, July 1982.

[38] Mahmood, A., Lu, D.J. and McCluskey, E.J., "Concurrent fault detection using a watchdog processor and assertions," *Proc. 1983 Int. Test Conf.*,, pp. 622-628, Oct., 1983.

[39] Mahmood, A. Ersoz, A. and McCluskey, E.J., "Concurrent system level error detection using a watchdog processor," *Proc. 1985 Int. Test Conf.*, pp. 145-152, Nov., 1985.

[40] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors - a survey," *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[41] Mahmood, A., and McCluskey, E., "Concurrent error detection using watchdog processors", *IEEE Trans. on Computers*, vol. 37, pp. 160-174, Feb., 1988.

[42] Nair, V., and Abraham, J., "General linear codes for fault-tolerant matrix operations on processor arrays," *Dig. of the 1988 Fault Tolerant Computing Symposium*, pp. 180-185, June, 1988.

[43] Namjoo, M., and McCluskey, E., "Watchdog processors and capability checking," *Digest of the 1982 Fault Tolerant Computing Symposium*, pp. 245-248, IEEE Computer Society Press, 1982.

[44] Namjoo, M. "Techniques for concurrent testing of VLSI processor operation," *Dig. 1982 Int. Test Conf.*, pp. 461-468, Nov., 1982.

[45] Namjoo, M. "CERBERUS-16: An architecture for a general purpose watchdog processor," *Dig. Papers 13th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 216-219, June, 1983.

[46] Preparata F. P., and Shamos M. I., *Computational geometry: an introduction*, Springer-Verlag, New York, NY, 1985.

[47] Prim, R. C., "Shortest connection networks and some generalizations," *Bell Syst. Tech. J.*, pp. 1389-1401, Nov., 1957.

[48] Randell, B., "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, pp. 220-232, June, 1975.

[49] Schmid, M., Trapp, R., Davidoff, A., and Masson, G., "Upset exposure by means of abstraction verefication," *Dig. of the 1982 Fault Tolerant Computing Symposium*, pp. 237-244, June, 1982.

[50] Sedgewick, R., "Implementing quicksort programs," *Communications of the ACM*, pp. 847-857, 21(10), 1978.

[51] Shen, J.P. and Schuette, M.A., "On-line self-monitoring using signatured instruction streams," *Proc. 1983 Int. Test Conf.,*, pp. 275-282, Oct., 1983.

[52] Siewiorek, D., and Swarz, R., *The theory and practice of reliable design*, Digital Press, Bedford, MA, 1982.

[53] Sridhar, T. and Thatte, S.M., "Concurrent checking of program flow in VLSI processors," *Dig. 1982 Int. Test Conf.*, pp. 191-199, Nov., 1982.

[54] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[55] Tarjan, R. E., "Efficiency of a good but not linear set union algorithm," *J. ACM*, 22(2), pp. 215-225, 1975.

[56] Tarjan, R. E., "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. of Comp. and Sys. Sci.*, 18(2), pp. 110-127, 1979.

[57] Tarjan, R. E., and Leeuwen, J. van, "Worst-case analysis of set union algorithms," *J. ACM*, 31(2), pp. 245-281, 1984.

[58] Tarjan, R. E., "Applications of path compression on balanced trees", *J. ACM*, pp. 690-715, Oct., 1979.

[59] Tomas, S. P. and Shen, J. P., "A roving monitoring processor for detection of control flow errors in multiple processor systems," *Proc. IEEE Int. Conf. Comput. Design: VLSI Comput.*, pp.531-539, Oct., 1985.

[60] Taylor, D., "Error Models for robust data structures," *Dig. 20th Annu. Int. Symp. Fault Tolerant Comput.*, pp. 416-422, 1990 June 26-28.

[61] Williams, J. W. J, "Algorithm 232 (heapsort)," *Commun. of ACM*, vol.7, pp. 347-348, 1964.

[62] Yau, S.S, and Chen, F.-C., "An approach to concurrent control flow checking," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 126-137, March 1980.

APPENDIX A

DATA ACQUISITION MODULE TECHNICAL MANUAL Ver. 1.0

THE TABLE OF CONTENTS

1. The Experimental System Overview

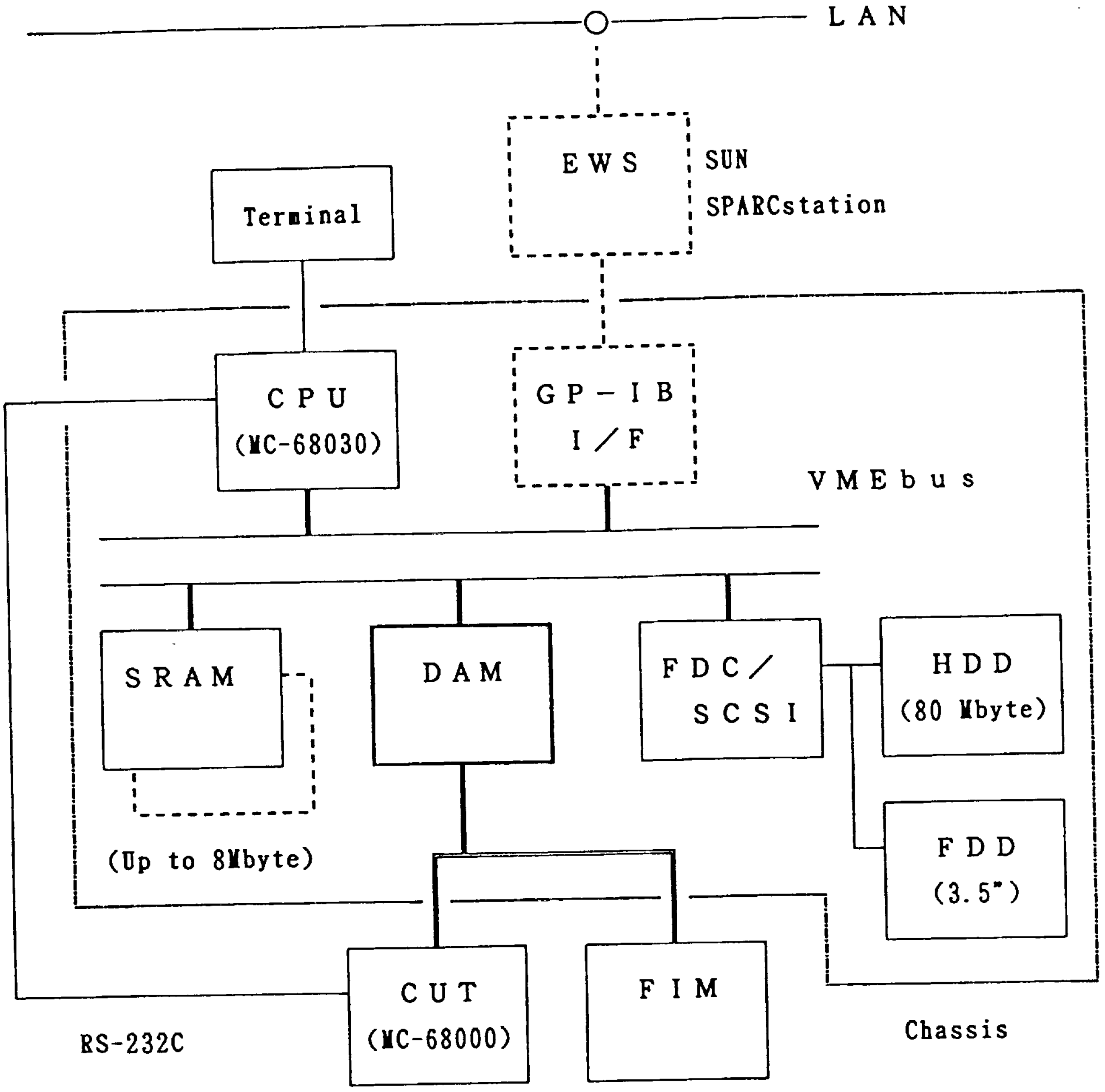
This system provides an experimental environment for recording and analyzing upset data in computer systems. This chapter provides the information on the system configuration and general hardware description.

1.1 System Configuration

This experimental system is mainly based on the VMEbus and controlled by the 68030 CPU board. The VMEbus provides a master-slave, asynchronous non- multiplexed data transfer medium. The target system (CPU Under Test) and the Fault Injection Module are connected by its local bus.

Fig.1.1 shows the experimental configuration. This system's features include:

- 68030 CPU Board
- Up to 8 Mbyte SRAM Memory Modules
- Floppy Disk and SCSI Bus Controller (FDC/SCSI)
- 80 Mbyte Hard Disk and 3.5" Floppy Disk Drive
- OS–9 Operating System
- Chassis with power supply, cooling fans, and motherboard
- Data Acquisition Module
- CPU Under Test (MC68000 Educational Computer Board)
- Fault Injection Module
- (GP–IB I/F Controller)
- (SUN SPARCstation)



DAM: Data Acquisition Module
CUT: CPU Under Test (Target System)
FIM: Fault Injection Module

Fig.1.1 Experimental Configuration

1.2 General System Description

This section briefly describes the general description of each module of the experimental system. For detailed information, refer to the user's manuals on specific modules.

- 68030 CPU Board
  - SYS68K/CPU-33XN (Force Computers Inc.)
  - 68030 CPU with 16.7 MHz clock frequency.
  - Not equipped with the Floating Point Coprocessor.
  - 32-bit high speed DMA controller for data transfers.
  - 1 Mbyte of shared dynamic RAM.
  - Two multiprotocol serial I/O channels.
  - Up to 2 Mbyte EPROM and up to 512 Kbyte SRAM/EEPROM.
  - Real Time Clock with calendar and on-board battery backup.
  - Full 32 bit VMEbus master/slave interface.
- Memory Module
  - SYS68K/SRAM-6 (Force Computers Inc.)
  - 2 Mbyte SRAM on SRAM-6.
  - Battery backup for SRAM devices.
  - 55ns(typical) Read/Write Access Time.
  - Jumper selectable access address and address modifier code.
  - VMEbus intereface supporting 32 data and 32 address lines.
- Floppy Disk and SCSI Bus Controller
  - SYS68K/ISCSI-1 (Force Computers Inc.)
  - 68010 CPU for local control.
  - 68450 DMA Controller for local transfers.
  - SCSI bus interface with the NCR5386S SCSI bus controller.

- SHUGART compatible floppy interface with the WD1772 FDC.
- All I/O signals available on P2 connector.
- VMEbus interface supporting A24:D16, D8.

- Mass Storage Module
  - SYS68K/MSM-84 (Force Computers Inc.)
  - Only VME P1 backplane is required.
  - 64 Pin flat cable is used to connect P2 of the ISCSI-1.
  - Floppy Disk Driver (Toshiba ND352)
    * Disk Size and Capacity: 3.5", 1.0 Mbyte
    * Number of Tracks: 160
    * Access Time: 79 ms (average)
  - Hard Disk (Quantum PRO80S)
    * Disk Size and Capacity: 3.5", 84 Mbyte
    * Number of Cylinders and Heads: 834, 6
    * Seek Time: 19 ms (average)

- OS-9 Operating System
  - Professional OS-9 (Microware Systems Corporation)
  - Multitasking, real time operating system.
  - UNIX-like shell and a hierarchical directory/file structure.
  - C Compiler, Assembler/Linker, and User-state Debugger.
  - μMACS screen-oriented text editor.

- Chassis with power supply, cooling fans, and motherboard
  - SYS68K/TARGET-32 (Force Computers Inc.)
  - 19", 7U chassis.
  - 500 W power supply to drive VMEbus and mass storage memory.
  - Cooling systems with four fans.
  - 20 slot J1-J2 VMEbus Motherboard.

- Data Acquisition Module
  - Up to 8 Mbyte address space.
  - Jumper selectable address modifier code.
  - 32 Input Channels with data selectors.
  - VMEbus compatible data transfers supporting A24:D32, D8.
  - VMEbus Master bus control (Non-slot 1)
- CPU Under Test
  - MC68000 Educational Computer Board (Motorola Inc.)
  - 4 MHz MC68000 16-bit CPU.
  - 32 Kbyte of DRAM and 16 Kbyte firmware ROM/EPROM monitor.
  - Two serial ports provided for a terminal and a host.
- Fault Injection Module
  - Hardware fault injections on IC pin lines.
  - Single/multiple faults of stuck/bridging types with fault duration varying from 250 ns to ~~ms.~~ 64μs .
  - Application program generated fault injection.

1.3 System Customization

This section describes the system customization required to implement the upset analysis experimental system. This also provides information on the programming of peripherals.

- SYS68K/CPU-33XN
  - OS-9/68000[1] EPROM Installation
    - * Remove VMEPROM[2] and install EPROMs for OS-9.
    - * High — Socket J6, Low — Socket J4
  - EPROM Type Selection
    - * 27512 EPROM
    - * Jumperfield B1: 1 to 12, 6 to 7
  - Interfacing PI/T2 User I/O Port
    - * Device: MC68230 Parallel Interface/Timer (PI/T)
    - * Accessible via the 8-bit local I/O bus. Table 1.1 shows the register layout of PI/T2.
    - * User I/O port is available on P2 of VMEbus, shown in Table 1.2.
  - The Address Map
    - * The address map of this CPU board is listed in Table 1.3.
    - * A24: D32, D24, D16, D8 area: SRAM-6, ISCSI-1
- SYS68K/SRAM-6
  - Address Modifier Selection
    - * Standard Supervisor/Non-privileged Data Access
    - * Address Modifier Code: 3D, 39
    - * Jumperfield B4: 4 to 15, 2 to 17
  - VMEbus Interface
    - * A24: D32, D16, D8
    - * Standard Address Mode (A24)

    * Address: $XX000000 — $XX2000000 (2 Mbyte)
    * Jumperfield B3: 18 to 15, 20 - 30 to 13 - 3

- SYS68K/ISCSI-1
  - Address Modifier Selection
    * Standard Non-priviledged/Supervisory program and data Access.
    * Address Modifier Code: 3A, 39, 3E, 3D
    * Jumperfield B22: 5 to 2, 6 to 1
  - VMEbus Interface
    * A24: D16, D8
    * Address: $XXA00000 — $XXA1FFFF (128 Kbyte)
    * Jumperfield B21: 2 to 17, 4 - 7 to 15 - 12

Table 1.1 PI/T2 Register Layout

ADDRESS	REGISTER	DESCRIPTION
FF800E00	PIT2 PGCR	Port General Control Register
FF800E01	PIT2 PSRR	Port Service Request Register
FF800E02	PIT2 PADDR	Port A Data Direction Register
FF800E06	PIT2 PACR	Port A Control Register
FF800E08	PIT2 PADR	Port A Data Register
FF800E0A	PIT2 PAAR	Port A Alternate Register
FF800E0D	PIT2 PSR	Port Status Register

Table 1.2 PI/T2 User I/O Interface Signals

PIN No.	PORT No.	IN/OUT	P2/J2 No.	SIGNAL
4	PA0	OUT	A29	READY*
5	PA1	OUT	C29	LW/B*
6	PA2	OUT	A30	SLCT0*
7	PA3	OUT	C30	SLCT1*
8	PA4	IN	A31	ENB0*
9	PA5	IN	C31	ENB1*
10	PA6		A32	
11	PA7		C32	
13	H1		A27	
14	H2		C27	
15	H3		A28	
16	H4		C28	

Table 1.3 The Address Map

START (HEX)	END (HEX)	SPACE	DESCRIPTION
00000000	003FFFFF	1.0 MB	Shared Memory
00400000	F9FFFFFF	3.9 GB	A32: D32, D24, D16, D8
FA000000	FAFFFFFF	16.0 MB	Message Broadcast Area
FB000000	FBFEFFFF	15.9 MB	A24: D32, D24, D16, D8
FBFF0000	FBFFFFFF	64.0 KB	A16: D32, D24, D16, D8
FC000000	FCFEFFFF	15.9 MB	A24: D16, D8
FCFF0000	FCFFFFFF	64.0 KB	A16: D16, D8
FD000000	FFFFFFFF		System Area

[1]OS-9 and OS-9/68000 are trademarks of Microware Systems Corporation.
[2]VMEPROM is a PDOS based real time monitor.

2. Data Acquisition Module

When the fault is injected from the fault injection module, the data acquisition module is activated and activity data on 8 or 32 observation points are synchronously sampled with the clock of the target system and written into the SRAM memory module.

2.1 Hardware Overview

Basically, the data acquisition module generates the address signals from the clock of the target system and transfers the sampled data to the memory module via the VMEbus.

A block diagram is shown in Fig.2.1. This board consists of the following functional blocks:

- Clock Control (CKCTRL)
- Address Generator (ADDGEN)
- Address Modifier Selector (AMS)
- Address Bus Buffers (ABUF)
- Data Transfer Control (DTCTRL)
- Input Channel Selectors (INSLCT)
- Data Bus Buffers (DBUF)
- Bus Master Control (BUSMST)

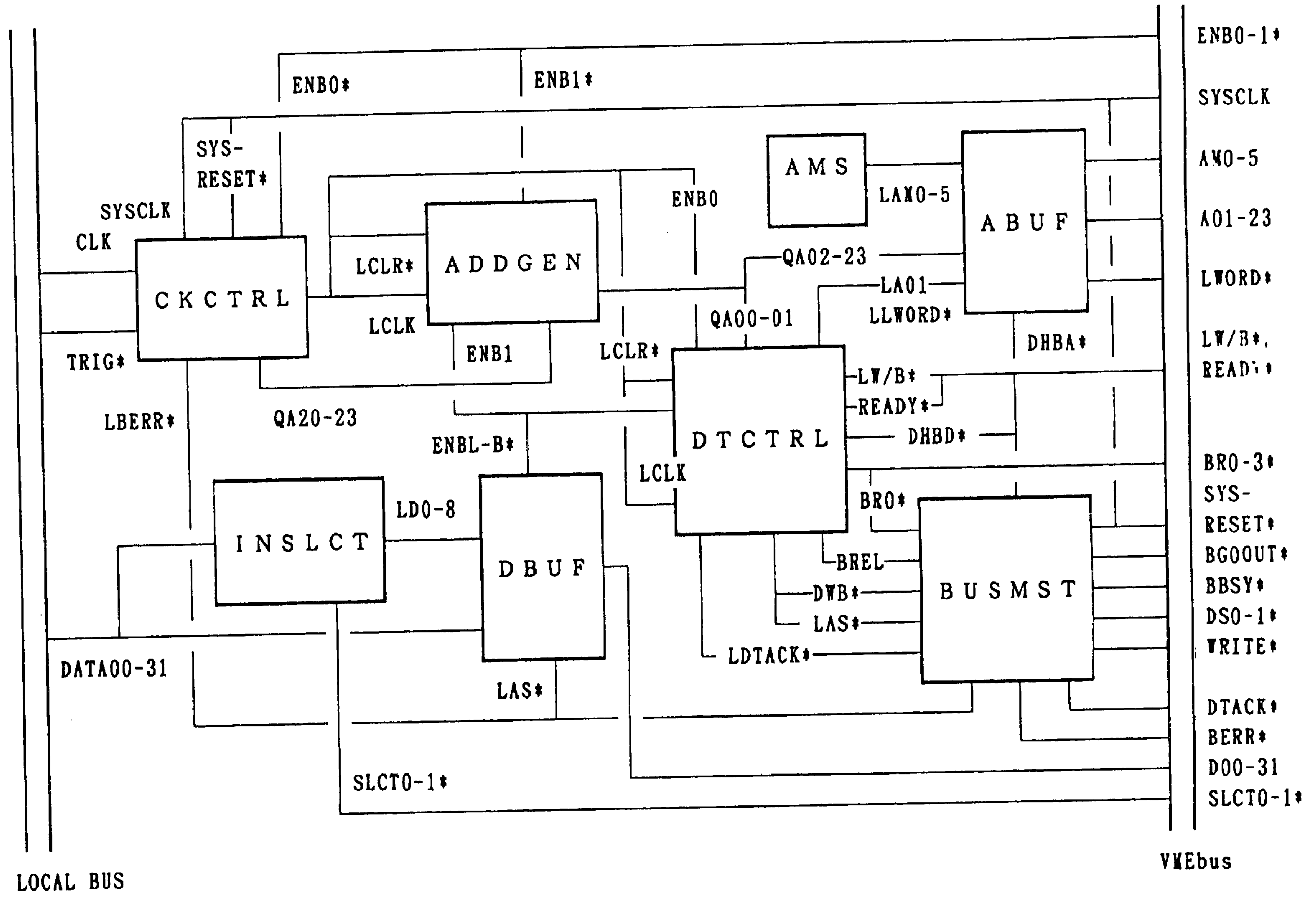


Fig.2.1 Functional Block Diagran

2.2 Clock Control

- Recording Clock Selector
  - J1-1, IC1-1
  - Selectable by bit 1 and 2 of J1.
    * Clock of CPU Under Test: bit 1: ON, bit 2: OFF
    * 16MHz VME System Clock: bit 1: OFF, bit 2: ON
- Clock Frequency Divider
  - J1-2, IC2
  - Selectable by bit 3 – 7 of J1 as shown in Table 2.1.
- Qualifier Trigger
  - IC1-2, IC3-1, IC10-1
  - Trigger: Fault injection signal transferred from FIM.
  - The trigger is enabled when ENB1 is high.
- Clear Control
  - R1, IC1-3, IC16-1
  - Generate Clear Signal for the Clock Control, Address Generator, and Data Transfer Control.
  - Reset Signals: System Reset, Bus Error, and End Address.
- End Address Selection
  - J2-1
  - End address: `$XX0FFFFF` – `$XX7FFFFF`
  - Selectable by bit 1 – 4 of J2-1 as shown in Table 2.2.

Table 2.1 Frequency Division Settings

Division	bit 3	bit 4	bit 5	bit 6	bit 7
1	ON	OFF	OFF	OFF	OFF
2	OFF	ON	OFF	OFF	OFF
4	OFF	OFF	ON	OFF	OFF
8	OFF	OFF	OFF	ON	OFF
16	OFF	OFF	OFF	OFF	ON

Table 2.2 End Address Selection

End Address	bit 1	bit 2	bit 3	bit 4
$XX0FFFFF	ON	OFF	OFF	OFF
$XX1FFFFF	OFF	ON	OFF	OFF
$XX3FFFFF	OFF	OFF	ON	OFF
$XX7FFFFF	OFF	OFF	OFF	ON

2.3 Address Generator

- Address Signal Generator
  - IC4, IC5, IC6, IC7, IC8, IC9
  - Implement 24-bit synchronous binary counter using a carry-look-ahead circuit.
  - Maximum clock frequency is calculated as follows:
    $f_{MAX} = 1/(CLKtoRCOt_{PLH} + ENTt_{SU})$
  - Address Space
    * Up to 8 Mbyte Address Space. Refer to Table 2.3.
    * Start address: $XX000000 (fixed)
    * End address: $XX0FFFFF – $XX7FFFFF (selectable)
- Counter Status Output
  - IC10-2
  - When counters are enabled to count, ENB1* is asserted.

Table 2.3 Address Space and End Address

Address Space	End Address
1 Mbyte	$XX0FFFFF
2 Mbyte	$XX1FFFFF
4 Mbyte	$XX3FFFFF
8 Mbyte	$XX7FFFFF

2.4 Address Bus Buffers and Address Modifier Selector

- Address Bus Buffers
  - IC12, IC13, IC14
  - Three transparent D-latches (74AS573) interface local address signals with the VMEbus address bus.
  - DHBA* places the 24-bit outputs in either a normal logic state or a high-impedance state.
- Address Modifier Selector
  - J2-2, RN, IC11
  - 6-bit Codes: Used for an additional decoding parallel to the address signals.
  - Address Mode: Supports the standard address mode (A24) for supervisor or nonpriviledged memory access.
    - 3E: Standard Supervisor Program Access
    - 3D: Standard Supervisor Data Access
    - 3A: Standard Non-Priviledged Program Access
    - 39: Standard Non-Priviledged Data Access
  - Selectable by bit 5 - 10 of J2 as shown in Table 2.4.

Table 2.4 Address Modifier Codes and Settings

HEX	Binary	bit 5	bit 6	bit 7	bit 8	bit 9	bit 10
3E	111110	OFF	OFF	OFF	OFF	OFF	ON
3D	111101	OFF	OFF	OFF	OFF	ON	OFF
3A	111010	OFF	OFF	OFF	ON	OFF	ON
39	111001	OFF	OFF	OFF	ON	ON	OFF

2.5 Data Transfer Control

- Data Transfer Bus Control
  - ENB1, DWB*
    - IC10-3, IC15-1
    - When READY* asserted, both ENB1 and DWB* are latched to be active.
    - LCLR* resets the outputs.
  - LAS*
    - R2, IC10-4, IC15-2, IC17-1, -2
    - When READY* asserted, LAS* is set to be active.
    - During data transfers, LAS* is asserted by LCLK and reset by LDTACK*.
  - LA01, LDS0–1*, LLWORD*
    - IC16-2, -3, -4, IC18-1, -2, IC30-1, -2, -3, IC33-1
    - When LW/B* is high (long word mode), LDS0*, LDS1*, LA01, and LLWORD* are set to low during data transfers.
    - When LW/B* is low (byte mode), LLWORD* is set to high and other signals respond as follows:
      LDS0* = QA00, LDS1* = –QA00, LA01 = QA01
- Data Bus Buffer Control
  - IC17-3, -4, IC18-4, -5
  - Long Word Mode (LW/B* is high)
    - During DHBD* is active, ENBL* is asserted and ENBB* is de-asserted.
  - Byte Mode (LW/B* is low)
    - During DHBD* is active, ENBB* is asserted and ENBL* is de-asserted.

- Bus Release Control
  - IC31-1
  - Support Release On Request (ROR) operation.
    - Bus request signals (BR0–3*) will assert BREL to release BBSY* at the end of the current data transfer.

2.6 Input Channel Selector and Data Bus Buffers

- Input Channel Selector
  - IC10-5, -6, IC19, IC20, IC21, IC22
  - Implement 32-to-8 data selectors using four 4-bit data selectors.
  - Data selection is controlled by the two select inputs (SCLT0–1*) as shown in Table 2.5.
- Data Bus Buffers
  - Long Word Mode
    - IC23, IC24, IC25, IC26
    - Four transparent D-latches (74AS573) interface 32-bit input data with the 32-bit VME data bus (D00–31).
    - When LAS* is taken low, the outputs are latched to retain the data that was set up. Refer to Table 2.6.
    - ENBL* places the 32-bit outputs in either a normal logic state or a high-impedance state.
  - Byte Mode
    - IC27, IC32
    - Two transparent D-latches (74AS573) interface 8-bit local data bus (LD0–7) with the 16-bit VME data bus (D00-15).
    - When LAS* is taken low, the outputs are latched to retain the data that was set up. Refer to Table 2.6.
    - ENBB* places the 16-bit outputs in either a normal logic state or a high-impedance state.

Table 2.5 Input Channel Selection

SLCT0*	SLCT1*	LD7	LD6	LD5	LD4	LD3	LD2	LD1	LD0
high	high	28	24	20	16	12	08	04	00
high	low	29	25	21	17	13	09	05	01
low	high	30	26	22	18	14	10	06	02
low	low	31	27	23	19	15	11	07	03

Table 2.6 (a) Active Portions of Data Bus

DS1*	DS0*	A01	LWORD*	D24-31	D16-23	D08-15	D00-07
low	low	low	low	byte 0	byte 1	byte 2	byte 3
high	low	high	high				byte 3
low	high	high	high			byte 2	
high	low	low	high				byte 1
low	high	low	high			byte 0	

Table 2.6 (b) Data Organization in Memory

Operand	Byte Address
byte 0	$XXX....XX00
byte 1	$XXX....XX01
byte 2	$XXX....XX10
byte 3	$XXX....XX11

2.7 VMEbus Master Control

- Master Bus Controller
  - IC28, IC29
  - VME 1220[1] provides two device chip set for non-slot 1 master bus controller.
  - Initiating a Bus Request
    - Drive BR0* low after receiving DWB* and LAS* asserted.
  - Arbitration
    - After receiving BG0IN* from daisy chained VMEbus grants, local arbiter arbitrates between DWB* and BG0IN.
      - If DWB* wins the arbitration (i.e. DWB* occurs before BG0IN*), BBSY* will be asserted.
      - If BG0IN* wins, local arbiter will drive BG0OUT*, which passes the bus grant down the daisy chain to adjacent master in the system.
  - Data Transfer
    - Local master does not access the bus until the previous master has relinquished control of bus, which occurs when AS*, DTACK* and BERR* are de-asserted.
    - Support Address Pipelining using DHBA* and DHBD*.
      - Broadcast the address of the next bus cycle while the data transfer of the current cycle is occuring, i.e. DTACK* and DSn* are still low.
      - DHBA* is enabled as soon as AS* is disabled.
      - When DTACK* goes high, signifying the end of the current data cycle, DHBD* enables the data buffers for the next data cycle.
    - WRITE* is latched during address pipelining to hold its level.
  - Bus Release
    - Supports Release On Request (ROR) protocol via BREL.
      - Release the data transfer bus whenever another module requires it.

- External bus request will assert BREL to release BBSY* at the end of the current data transfer. Refer to section **2.5**.
- If no bus requests are pending, the BREL will be kept de-asserted and the local master maintains BBSY* low to perform continuous VMEbus data transfer cycles.

[1] PLX Technology, 625 Clyde Ave., Mountain View, CA 94043

3. Interface Signals

3.1 VMEbus Interface

This section provides information on VMEbus interface. Table 3.1 and Table 3.2 list P1/J1 and P2/J2 pin assignments respectively. The P1 connector includes all the signals required for the 68000. The P2 connector provides expansion of both address and data buses to 32 bits and also provides 96 pins for user I/O lines.

The data transfer bus is very similar to the 68000's native buses except the following signals. Long word (LWORD*) is asserted for 32-bit data transfers. The 6-bit address modifier (AM0 - AM5) allows the type of access to be specified. The bus error signal (BERR*) is typically used to indicate a memory error.

The interrupt bus has seven interrupt request lines (IRQi*), an interrupt acknowledge (IACK*), and a daisy-chained priority signal (IACKIN*, IACKOUT*). Each of seven lines corresponds to an interrupt priority level.

The arbitration bus provides four levels of arbitration. For each level, there is a bus request signal (BRi*) and a bus grant daisy chain (BGiIN*, BGiOUT*). The utility bus consists of SYSCLK, SYSRESET*, SYSFAIL*, ACFAIL*, and power supplies.

Table 3.1 VMEbus P1/J1 Pin Assignments

PIN No.	P1/J1 ROW A	P1/J1 ROW B	P1/J1 ROW C
1	D00	BBSY*	D08
2	D01	BCLR*	D09
3	D02	ACFAIL*	D10
4	D03	BG0IN*	D11
5	D04	BG0OUT*	D12
6	D05	BG1IN*	D13
7	D06	BG1OUT*	D14
8	D07	BG2IN*	D15
9	GND	BG2OUT*	GND
10	SYSCLK	BG3IN*	SYSFAIL*
11	GND	BG3OUT*	BERR*
12	DS1*	BR0*	SYSRESET*
13	DS0*	BR1*	LWORD*
14	WRITE*	BR2*	AM5
15	GND	BR3*	A23
16	DTACK*	AM0	A22
17	GND	AM1	A21
18	AS*	AM2	A20
19	GND	AM3	A19
20	IACK*	GND	A18
21	IACKIN*	SERCLK	A17
22	IACKOUT*	SERDAT*	A16
23	AM4	GND	A15
24	A07	IRQ7*	A14
25	A06	IRQ6*	A13
26	A05	IRQ5*	A12
27	A04	IRQ4*	A11
28	A03	IRQ3*	A10
29	A02	IRQ2*	A09
30	A01	IRQ1*	A08
31	−12VDC	+5VSTDBY	+12VDC
32	+5VDC	+5VDC	+5VDC

Table 3.2 VMEbus P2/J2 Pin Assignments

PIN No.	P2/J2 ROW A	P2/J2 ROW B	P2/J2 ROW C
1		+5VDC	
2		GND	
3		RESERVED	
4		A24	
5		A25	
6		A26	
7		A27	
8		A28	
9		A29	
10		A30	
11		A31	
12		GND	
13		+5VDC	
14		D16	
15		D17	
16		D18	
17		D19	
18		D20	
19		D21	
20		D22	
21		D23	
22		GND	
23		D24	
24		D25	
25		D26	
26		D27	
27		D28	
28		D29	
29	READY*	D30	LW/B*
30	SLCT0*	D31	SLCT1*
31	ENB0*	GND	ENB1*
32		+5VDC	

3.2 Input Channels

The input channels consit of data channels (DATA00–31), clock (CLK), and trigger signal (TRIG*). Table 3.3 shows the pin assignments of the input channels.

Table 3.3 Input Channel Pin Assignments

PIN	DAM Signal	ECB Signal	PIN	DAM Signal	ECB Signal
(a)			(b)	GND	GND
(c)			(d)	GND	GND
1	DATA04	D04	2	DATA03	D03
3	DATA05	D05	4	DATA02	D02
5	DATA06	D06	6	CLK	4M-CLK
7	DATA07	D07	8	DATA14	D14
9	DATA08	D08	10	DATA15	D15
11	DATA09	D09	12	TRIG*	FIEN*[1]
13	DATA10	D10	14	DATA01	D01
15	DATA11	D11	16		E
17	DATA12	D12	18		AS*
19	DATA13	D13	20		UDS*
21	DATA00	D00	22		LDS*
23	DATA31	A15	24	DATA16	R/W*
25	DATA30	A14	26	DATA29	A13
27	DATA28	A12	28		FC2
29	DATA27	A11	30		FC1
31	DATA26	A10	32		FC0
33	DATA25	A09	34	DATA17	A01
35	DATA24	A08	36	DATA18	A02
37	DATA22	A06	38	DATA19	A03
39	DATA23	A07	40	DATA20	A04
41	DATA21	A05	42		DTACK*
43		8M-CLK	44		6800IRQ*
45		1M-CLK	46		VMA*

[1]FIEN*: Fault Injection Enable, a signal transferred from the fault injection module.

Appendix A   Schematic Diagrams

A.1 Clock Control

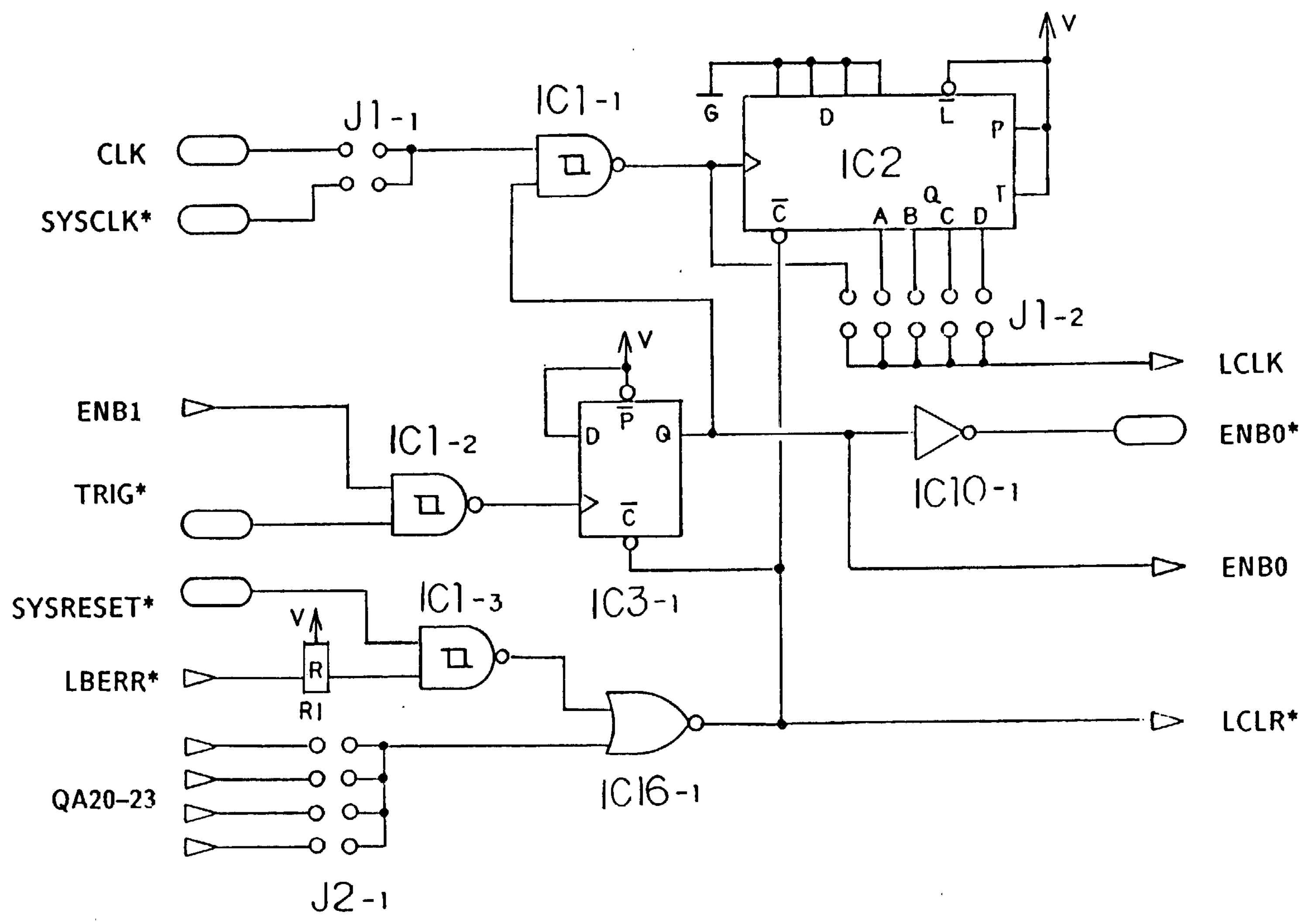
A.2 Address Generator

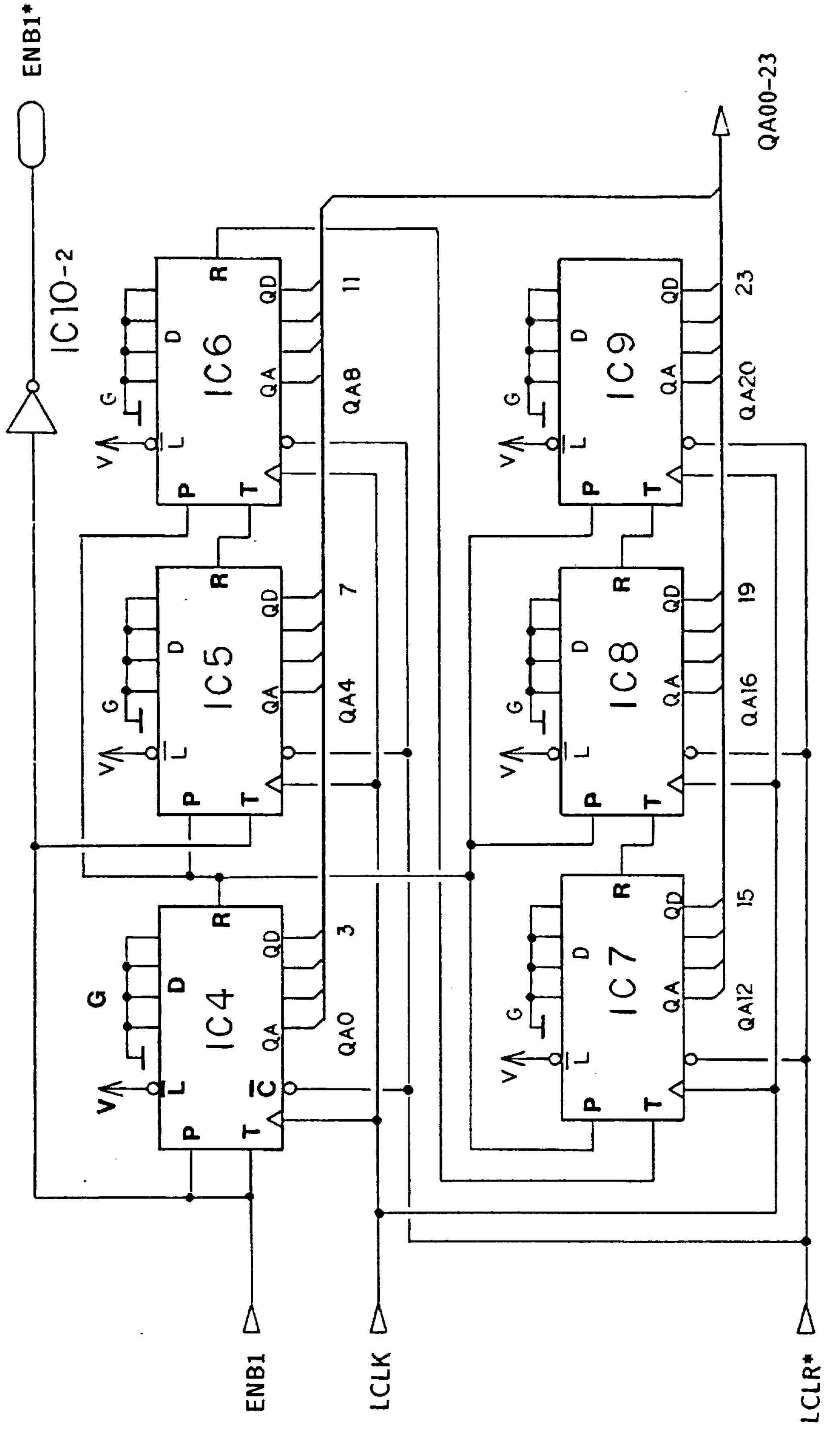
A.3 Address Bus Buffers and Address Modifier Selector

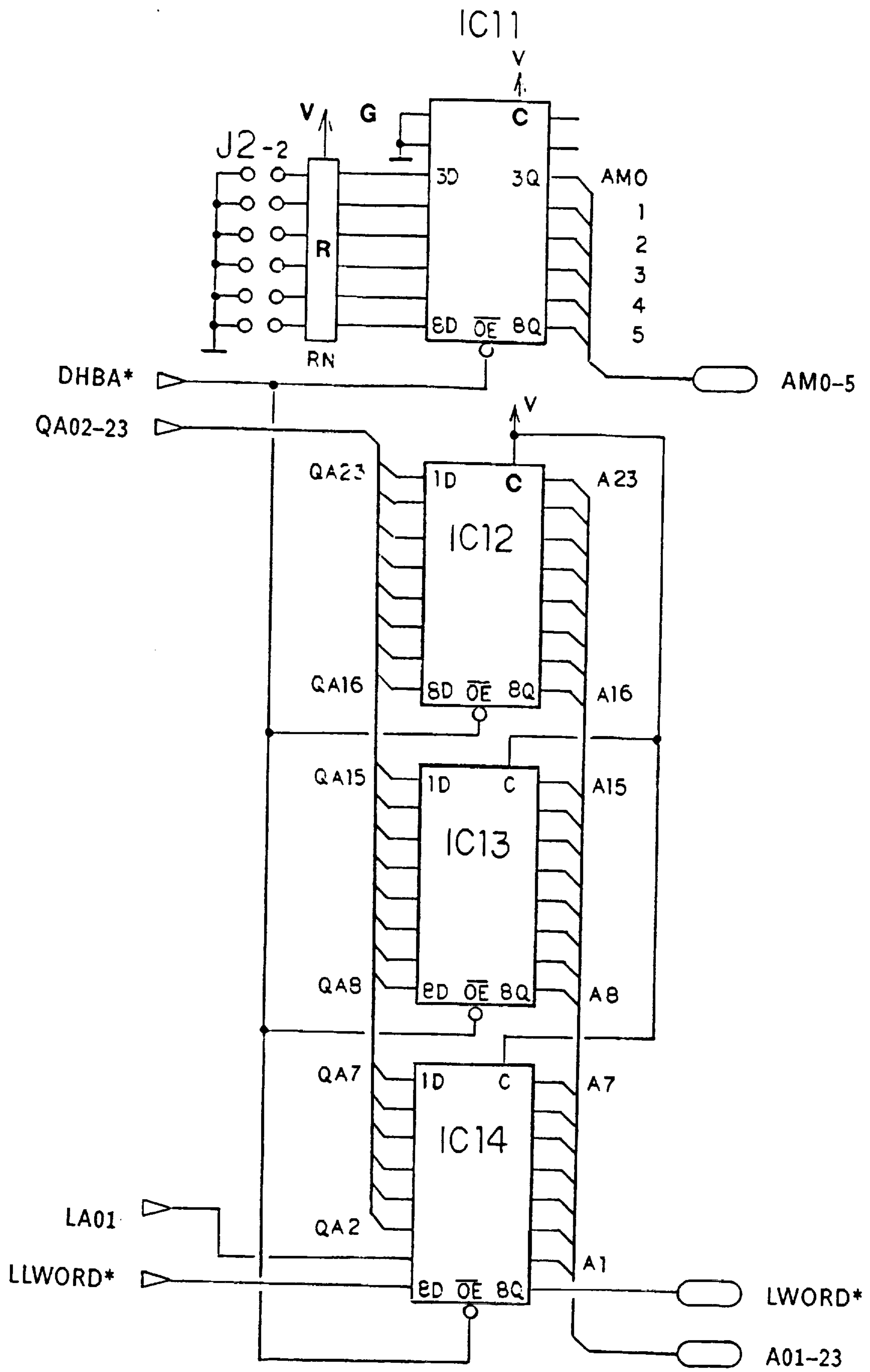
A.4 Data Transfer Control

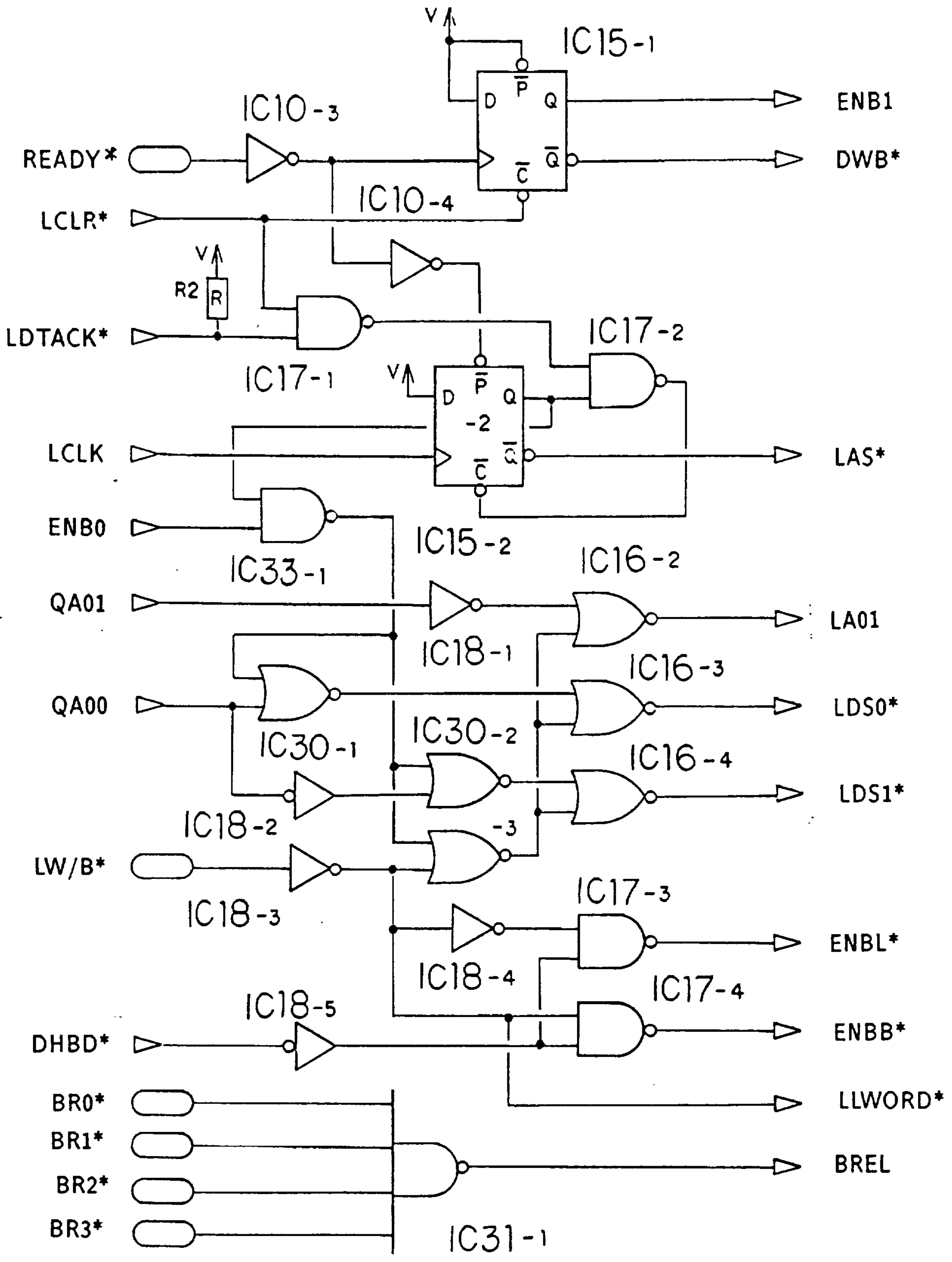
A.5 Input Channel Selector and Data Bus Buffers

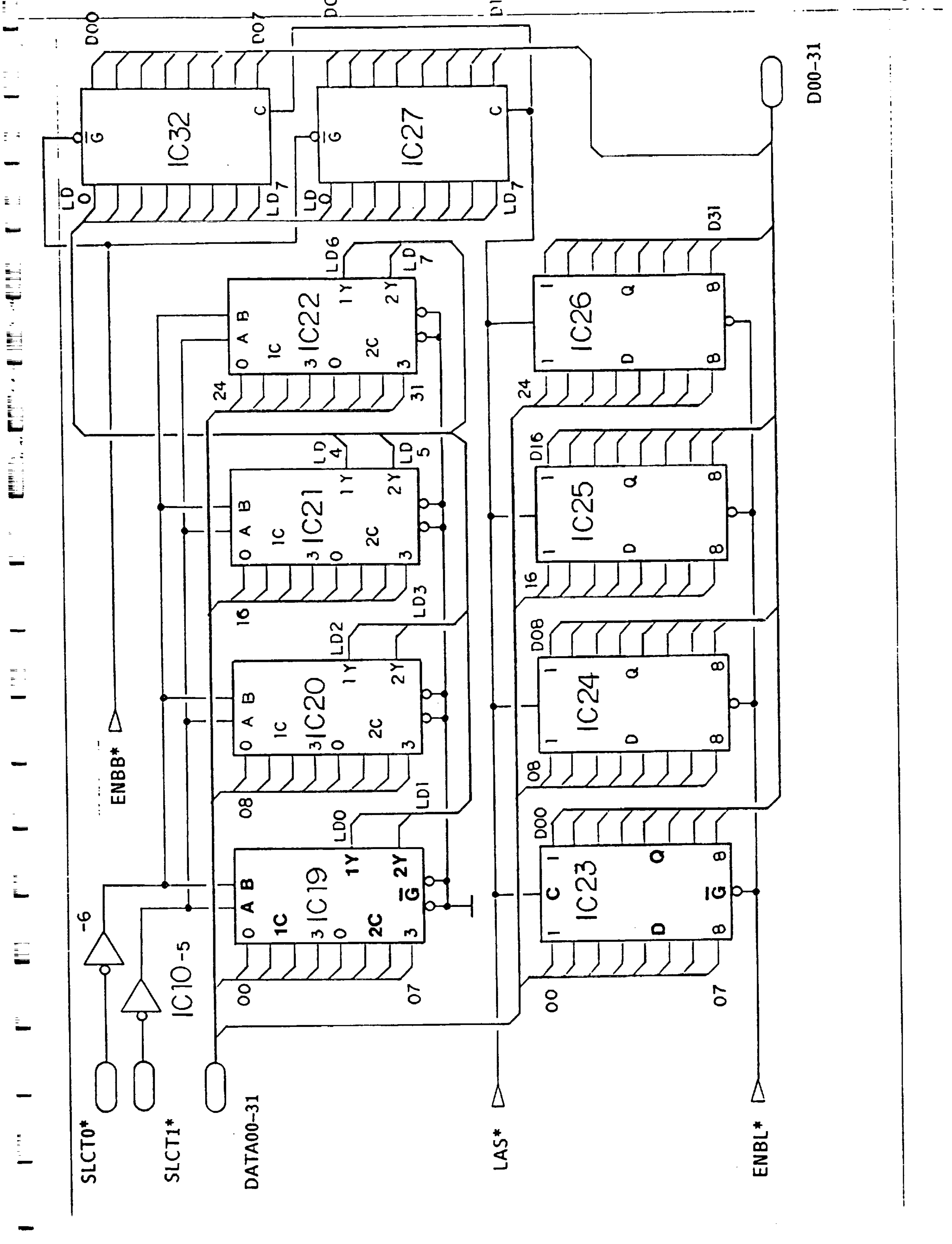
A.6 VMEbus Master Control


CLK
SYSCLK*
J1-1
IC1-1
G
D
L
P
T
IC2
C
Q
A B C D
V
J1-2
LCLK
ENB1
TRIG*
IC1-2
D
P
Q
C
IC3-1
IC10-1
ENB0*
ENB0
SYSRESET*
LBERR*
R
R1
IC1-3
IC16-1
LCLR*
QA20-23
J2-1

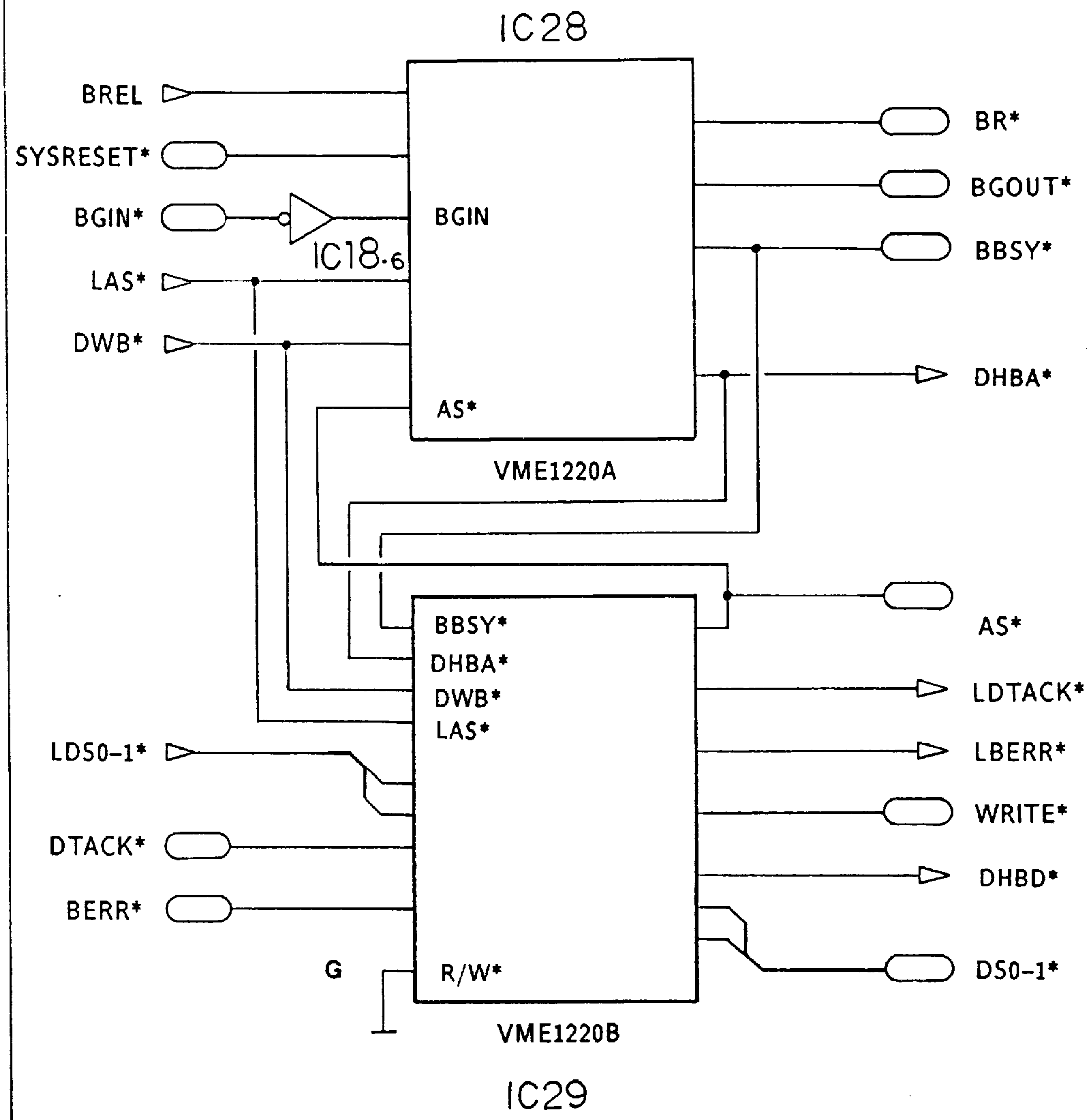

ENB1*
IC10-2
QA00-23
IC4
IC5
IC6
IC7
IC8
IC9
G
D
V
L
P
T
R
C̄
QA
QD
QA0
3
QA4
7
QA8
11
QA12
15
QA16
19
QA20
23
ENB1
LCLK
LCLR*


IC11
V
C
V
G
J2-2
3D
3Q
AM0
1
2
3
4
5
R
8D
OE
8Q
RN
DHBA*
AM0-5
QA02-23
V
QA23
1D
C
A23
IC12
QA16
8D
OE
8Q
A16
QA15
1D
C
A15
IC13
QA8
8D
OE
8Q
A8
QA7
1D
C
A7
IC14
LA01
QA2
A1
LLWORD*
8D
OE
8Q
LWORD*
A01-23


IC15-1
V
D
P̄
Q
Q̄
C̄
ENB1
DWB*
IC10-3
READY*
IC10-4
LCLR*
V
R2
R
LDTACK*
IC17-1
IC17-2
V
D
P̄
Q
-2
Q̄
C̄
LCLK
LAS*
ENB0
IC33-1
IC15-2
IC16-2
QA01
LA01
IC18-1
IC16-3
QA00
LDS0*
IC30-1
IC30-2
IC16-4
LDS1*
IC18-2
-3
LW/B*
IC18-3
IC17-3
ENBL*
IC18-4
IC17-4
IC18-5
ENBB*
DHBD*
LLWORD*
BR0*
BR1*
BREL
BR2*
BR3*
IC31-1



SLCT0*
SLCT1*
DATA00–31
LAS*
ENBL*
ENBB*
D00–31
IC10–5
–6
IC19
IC20
IC21
IC22
IC23
IC24
IC25
IC26
IC27
IC32
D00
D07
D08
D15
D16
D31
LD0
LD1
LD2
LD3
LD4
LD5
LD6
LD7
00
07
08
16
24
31
A
B
1C
2C
1Y
2Y
G
C
D
Q


IC28
BREL
SYSRESET*
BGIN*
LAS*
DWB*
BGIN
IC18-6
AS*
VME1220A
BR*
BGOUT*
BBSY*
DHBA*
AS*
BBSY*
DHBA*
DWB*
LAS*
LDS0-1*
DTACK*
BERR*
G
R/W*
LDTACK*
LBERR*
WRITE*
DHBD*
DS0-1*
VME1220B
IC29

Appendix B Parts List

Table B.1 DAM Parts List (1)

LABEL	Part Number	Pins	DESCRIPTION
IC1	74LS132	14	Quadruple Schmitt NAND gates
IC2	74LS161A	16	Synchronous 4-bit counter
IC3	74AS74	14	Dual D-type F/Fs
IC4	74LS161A	16	Synchronous 4-bit counter
IC5	74LS161A	16	
IC6	74LS161A	16	
IC7	74LS161A	16	
IC8	74LS161A	16	
IC9	74LS161A	16	
IC10	74LS04	14	Hex inverters
IC11	74AS573	20	Octal D-type transparent latches
IC12	74AS573	20	
IC13	74AS573	20	
IC14	74AS573	20	
IC15	74AS74	14	Dual D-type F/Fs
IC16	74AS02	14	Quadruple 2-input NOR gates
IC17	74AS00	14	Quadruple 2-input NAND gates
IC18	74AS04	14	Hex inverters
IC19	74LS153	16	Dual 4-to-1 data selectors

Table B.2 DAM Parts List (2)

LABEL	Part Number	Pins	DESCRIPTION
IC20	74LS153	16	Dual 4-to-1 data selectors
IC21	74LS153	16	
IC22	74LS153	16	
IC23	74AS573	20	Octal D-type transparent latches
IC24	74AS573	20	
IC25	74AS573	20	
IC26	74AS573	20	
IC27	74AS573	20	
IC28	VME1220A	24	VMEbus master controller
IC29	VME1220B	24	(Non-slot 1, P-45)
IC30	74AS02	14	Quadruple 2-input NOR gates
IC31	74LS20	14	Dual 4-input NAND gates
IC32	74AS573	20	Octal D-type transparent latches
IC33	74AS00	14	Quadruple 2-input NAND gates

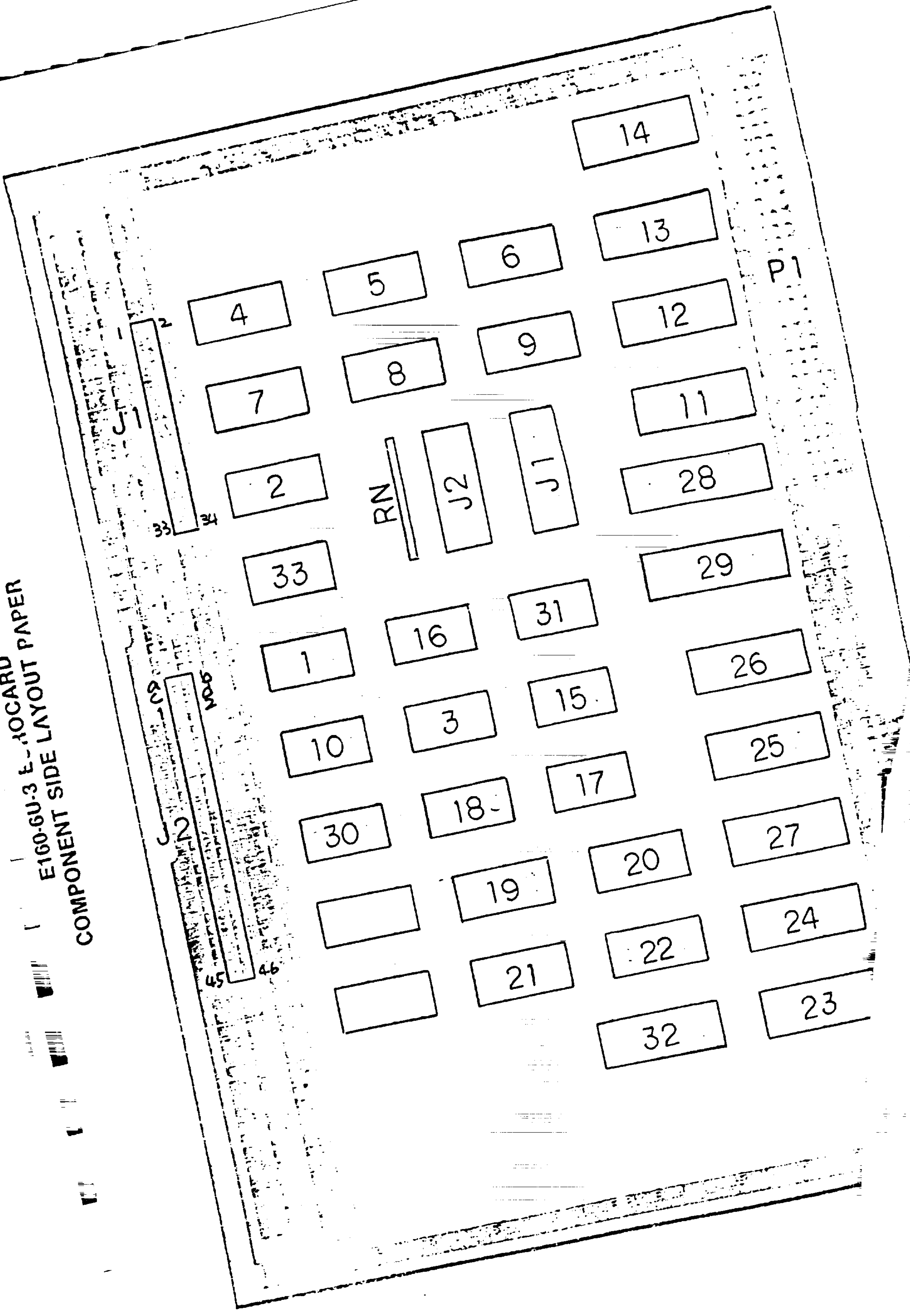
Appendix C DAM Board Layout

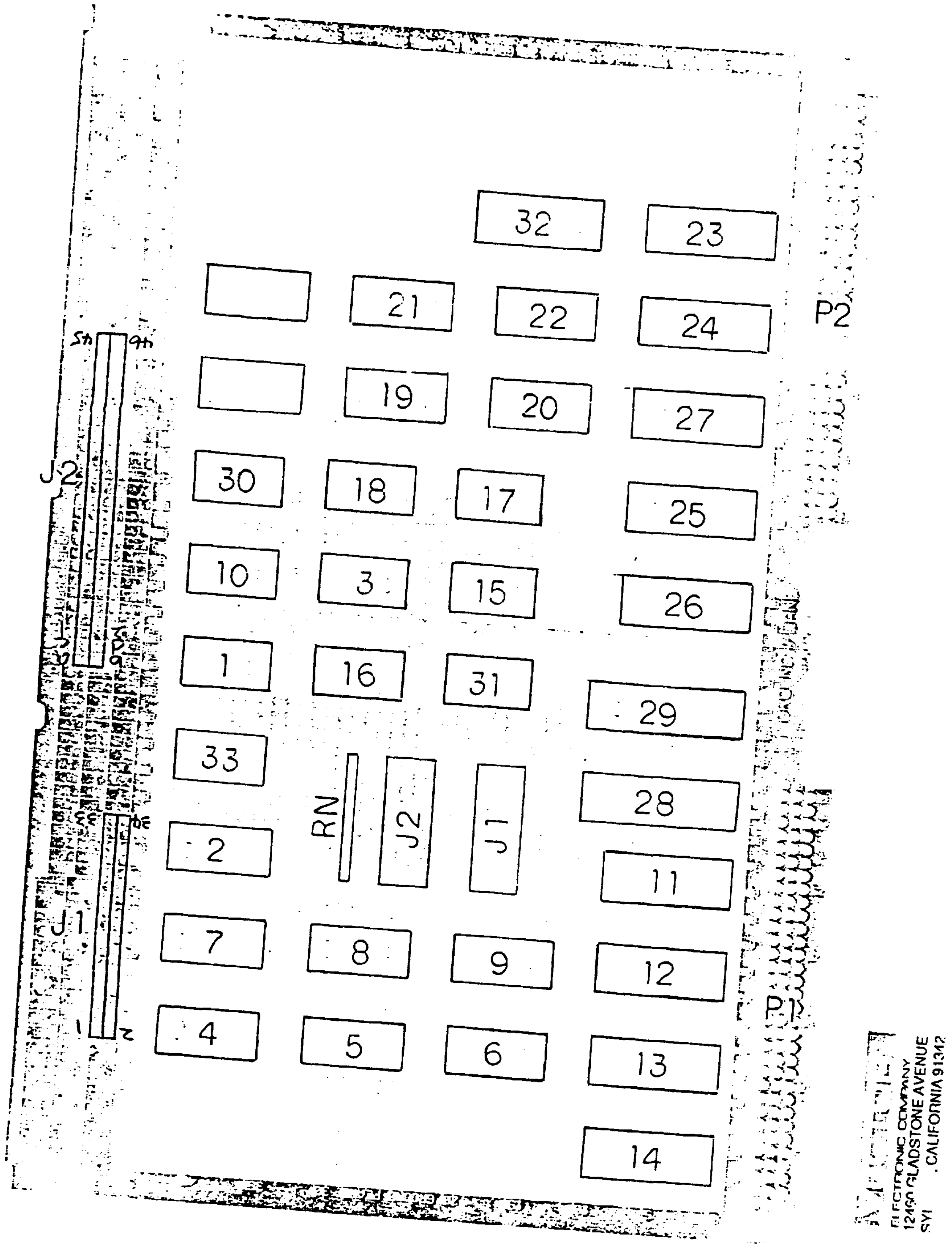
C.1 Component Side Layout

C.2 Wiring Side Layout

LA52P1

E160-6U-3 E[illegible]ROCARD
COMPONENT SIDE LAYOUT PAPER





ORIGINAL PAGE IS
OF POOR QUALITY

Appendix D Copies of Data Sheets

D.1 VME 1220 Non-Slot 1 VMEbus Master Controller

VME 1210/1220

June 1990

Slot 1 and Non-Slot 1 VMEbus Master Controllers

Distinctive Features

- **VME 1210 provides two device chip set for slot 1 master bus controller and single level arbiter**
- **VME 1220 provides two device chip set for non-slot 1 master bus controller**
- **Integrates 48ma and 64ma VMEbus signals:AS*,DS0*,DS1*,WRITE*,BR*,BBSY***
- **Integrates input hysteresis buffers**
- **Supports Release When Done (RWD) and Release On Request (ROR) protocols**
- **Supports address pipelining, block transfers, and early BBSY* release**
- **Available in Commercial, Industrial and Military temperature ranges**

Programmable Version Available

If the VME 1210/1220 does not match the requirements of the design, a programmable version is available (the PLX 464) which allows the user to customize all inputs, outputs and logic. Programming is performed using industry standard tools such as ABEL™ and CUPL™ software and commonly available PLD programming hardware. Contact PLX for a data sheet on the PLX 464 and other PLX products.

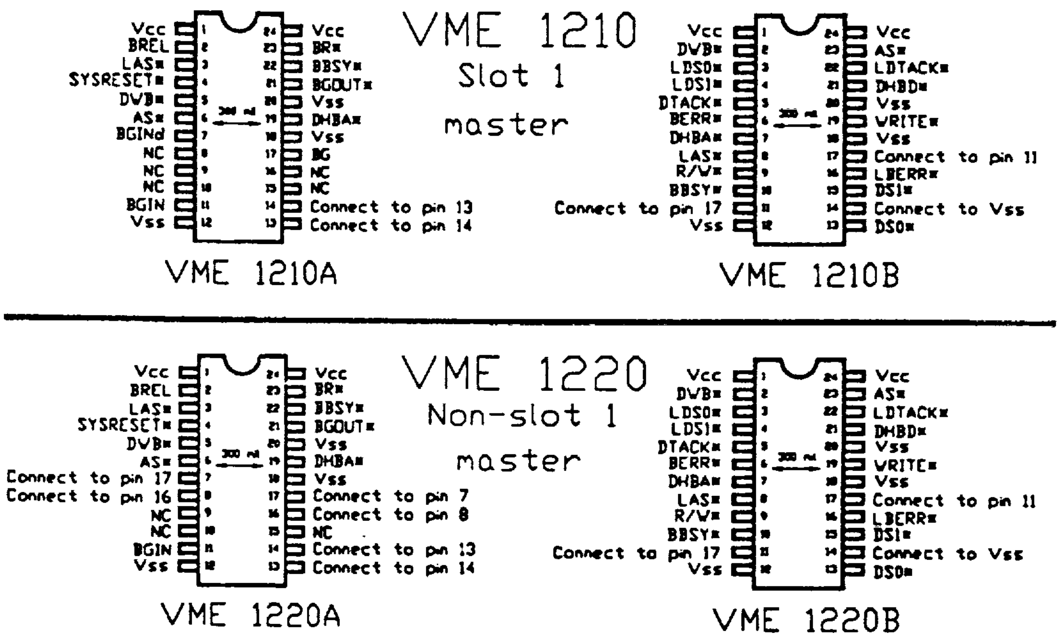
Applications

- VMEbus masters residing in slot 1 boards (VME 1210)
- VMEbus masters residing in non-slot 1 boards (VME 1220)

General Description

The VME 1210: The VME 1210 is comprised of the VME 1210A and the VME 1210B for slot 1 applications. The devices are CMOS and packaged in 24 pin 300 mil wide DIPs or 28 pin J-type LCCs. The VME 1210A provides bus requesting, local arbitration, and single level system arbitration. The VME 1210B functions as the VMEbus controller. The requester initiates a VMEbus request from the local master's bus request for a data or interrupt cycle. The bus controller controls the bus after initiation of a bus cycle and relinquishes the bus at the end of the bus cycle. The bus controller supervises the handshaking between the local master CPU and the slave modules.

The VME 1220: The VME 1220 is comprised of the VME 1220A and the VME 1220B for non-slot 1 applications. The devices are CMOS and packaged in 24 pin 300 mil wide DIPs or 28 pin J-type LCCs. The VME 1220A provides bus requesting and local arbitration. The VME 1220B functions as the VMEbus controller. The requester initiates a VMEbus request from the local master's bus request for a data or interrupt cycle. The bus controller controls the bus after initiation of a bus cycle and relinquishes the bus at the end of the bus cycle. The bus controller supervises the handshaking between the local master CPU and the slave modules.



VME 1210
Slot 1
master
VME 1210A: 1 Vcc, 2 BREL, 3 LAS*, 4 SYSRESET*, 5 DVB*, 6 AS*, 7 BGINd, 8 NC, 9 NC, 10 NC, 11 BGIN, 12 Vss, 13 Connect to pin 14, 14 Connect to pin 13, 15 NC, 16 NC, 17 BG, 18 Vss, 19 DHBA*, 20 Vss, 21 BGOUT*, 22 BBSY*, 23 BR*, 24 Vcc
VME 1210B: 1 Vcc, 2 DVB*, 3 LDS0*, 4 LDS1*, 5 DTACK*, 6 BERR*, 7 DHBA*, 8 LAS*, 9 R/W*, 10 BBSY*, 11 Connect to pin 17, 12 Vss, 13 DS0*, 14 Connect to Vss, 15 DS1*, 16 LBERR*, 17 Connect to pin 11, 18 Vss, 19 WRITE*, 20 Vss, 21 DHBD*, 22 LDTACK*, 23 AS*, 24 Vcc
300 mil
VME 1220
Non-slot 1
master
VME 1220A: 1 Vcc, 2 BREL, 3 LAS*, 4 SYSRESET*, 5 DVB*, 6 AS*, 7 Connect to pin 17, 8 Connect to pin 16, 9 NC, 10 NC, 11 BGIN, 12 Vss, 13 Connect to pin 14, 14 Connect to pin 13, 15 NC, 16 Connect to pin 8, 17 Connect to pin 7, 18 Vss, 19 DHBA*, 20 Vss, 21 BGOUT*, 22 BBSY*, 23 BR*, 24 Vcc
VME 1220B: 1 Vcc, 2 DVB*, 3 LDS0*, 4 LDS1*, 5 DTACK*, 6 BERR*, 7 DHBA*, 8 LAS*, 9 R/W*, 10 BBSY*, 11 Connect to pin 17, 12 Vss, 13 DS0*, 14 Connect to Vss, 15 DS1*, 16 LBERR*, 17 Connect to pin 11, 18 Vss, 19 WRITE*, 20 Vss, 21 DHBD*, 22 LDTACK*, 23 AS*, 24 Vcc

Figure 1. Pinout of VME 1210/1220 (DIPs)

ORIGINAL PAGE IS OF POOR QUALITY

Patent Pending
ABEL is a trademark of Data I/O Corp.
CUPL is a trademark of Logical Devices, Inc.

PRECEDING PAGE BLANK NOT FILMED

in Description

VME 1220A

Pin # LCC	Pin # DIP	Signal	Type	Function
3	2	BREL	I	Active high; Bus release signal indicating BBSY* can be released.
4	3	LAS*	I	Active low; Address strobe from local master.
5	4	SYSRESET*	I	Active low; VMEbus System Reset.
6	5	DWB*	I	Active low; Device wants bus, local master requests control of bus.
7	6	AS*	I	Active low; VMEbus Address Strobe.
9	7	-	I	Connect to pin 17 (DIP) or pin 20 (LCC).
10	8	-	I	Connect to pin 16 (DIP) or pin 19 (LCC).
11	9	NC	I	No Connect.
12	10	NC	I	No Connect.
13	11	BGIN	I	Active high; Inverted VMEbus Bus Grant In signal, BGIN*.
14,21, 24	12,18, 20	Vss		Chip Ground.
16	13	-	O	Connect to Pin 14 (DIP) or Pin 17 (LCC).
17	14	-	I	Connect to Pin 13 (DIP) or Pin 16 (LCC).
18	15	NC	O	No Connect.
19	16	-	O	Connect to pin 8 (DIP) or pin 10 (LCC).
20	17	-	O	Connect to pin 7 (DIP) or pin 9 (LCC).
23	19	DHBA*	O	Active low; Device has bus address, address buffer enable.
25	21	BGOUT*	O	Active low; VMEbus Bus Grant Out signal.
26	22	BBSY*	I/O	Active low, 48 mA open collector; VMEbus Bus Busy signal.
27	23	BR*	O	Active low, 48 mA open collector; VMEbus Bus Request signal.
2,28	1,24	Vcc		+5 V Chip Power
1,8, 15,22	-	NC	-	No Connect.

PRECEDING PAGE BLANK NOT FILMED

Pin Description

VME 1210B and VME1220B

Pin # LCC	Pin # DIP	Signal	Type	Function
3	2	DWB*	I	Active low; Device wants bus, local master wants control of VMEbus.
4	3	LDS0*	I	Active low; Lower data strobe from local master.
5	4	LDS1*	I	Active low; Upper data strobe from local master.
6	5	DTACK*	I	Active low; VMEbus Data Transfer Acknowledge, data is valid during a read cycle or data has been accepted from the bus during a write cycle.
7	6	BERR*	I	Active low; VMEbus Error signal.
9	7	DHBA*	I	Active low; Device has bus address, address buffer enable.
10	8	LAS*	I	Active low; Address strobe from local master.
11	9	R/W*	I	Active high/low; Read or write cycle from local master.
12	10	BBSY*	I	Active low; VMEbus Busy, local master controls bus.
13	11	-	I	Connect to pin 17 (DIP) or pin 20 (LCC).
14,21, 24	12,18, 20	Vss		Chip Ground.
16	13	DS0*	O	Active low; 64ma VMEbus lower Data Strobe signal, indicates valid data on bus.
17	14	-	I	Connect to Vss.
18	15	DS1*	O	Active low; 64ma VMEbus upper Data Strobe signal, indicates valid data on bus.
19	16	LBERR*	O	Active low; Open collector signal, bus error to local master.
20	17	-	O	Connect to pin 11 (DIP) or pin 13 (LCC).
23	19	WRITE*	O	Active low; 48ma VMEbus Write signal, indicates bus read or write cycle.
25	21	DHBD*	O	Active low; Device has bus data, data buffer enable.
26	22	LDTACK*	O	Active low; Open collector signal, data acknowledge to local master.
27	23	AS*	O	Active low; 64mA VMEbus Address Strobe signal, indicates valid address on bus.
2,28	1,24	Vcc		+5 V Chip Power
1,8, 15,22	-	NC	-	No Connect.

VME 1210/1220 Timing Waveforms

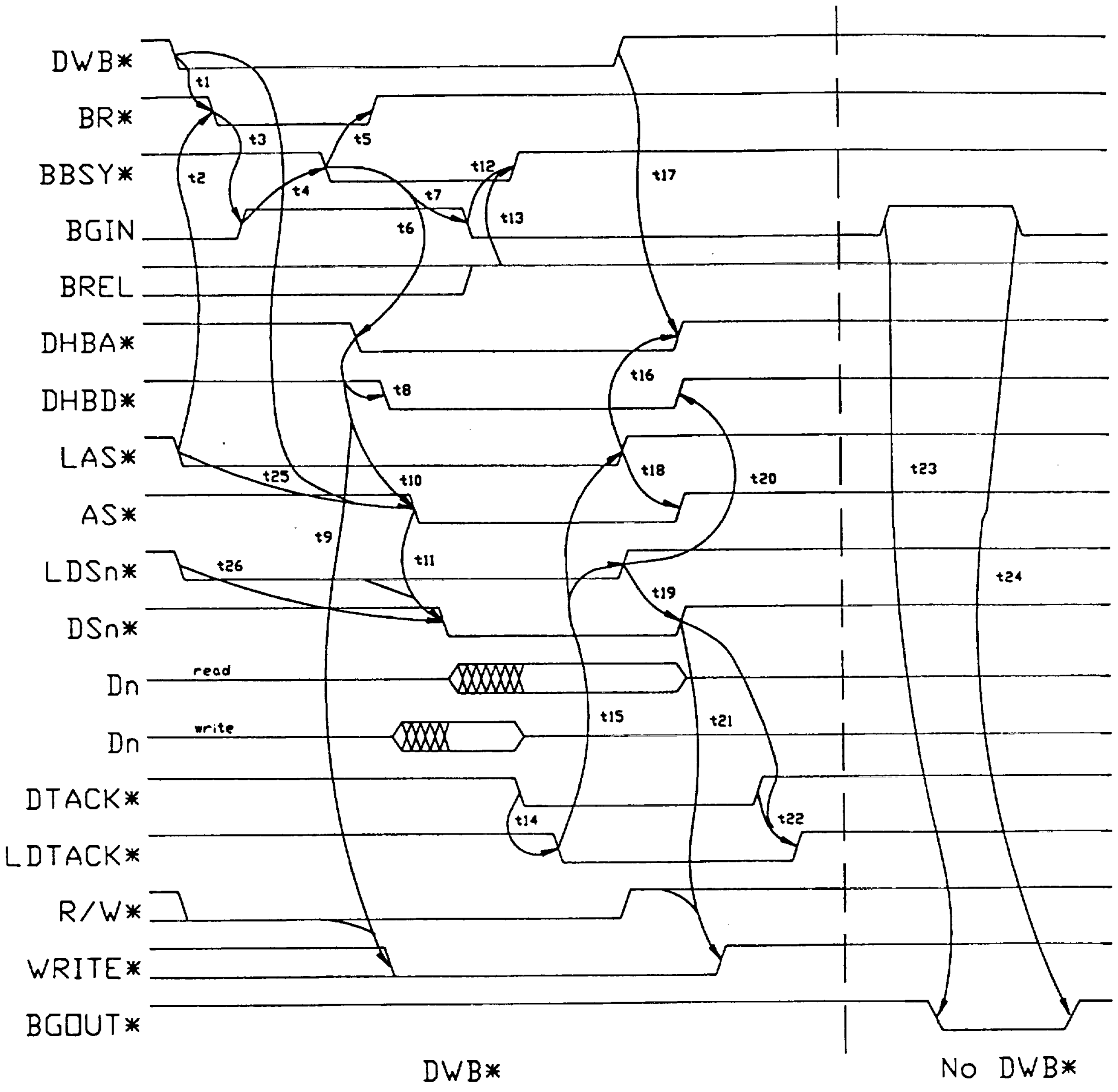


Figure 5. Timing Diagram

6-8

PRECEDING PAGE BLANK NOT FILMED

Timing Specifications

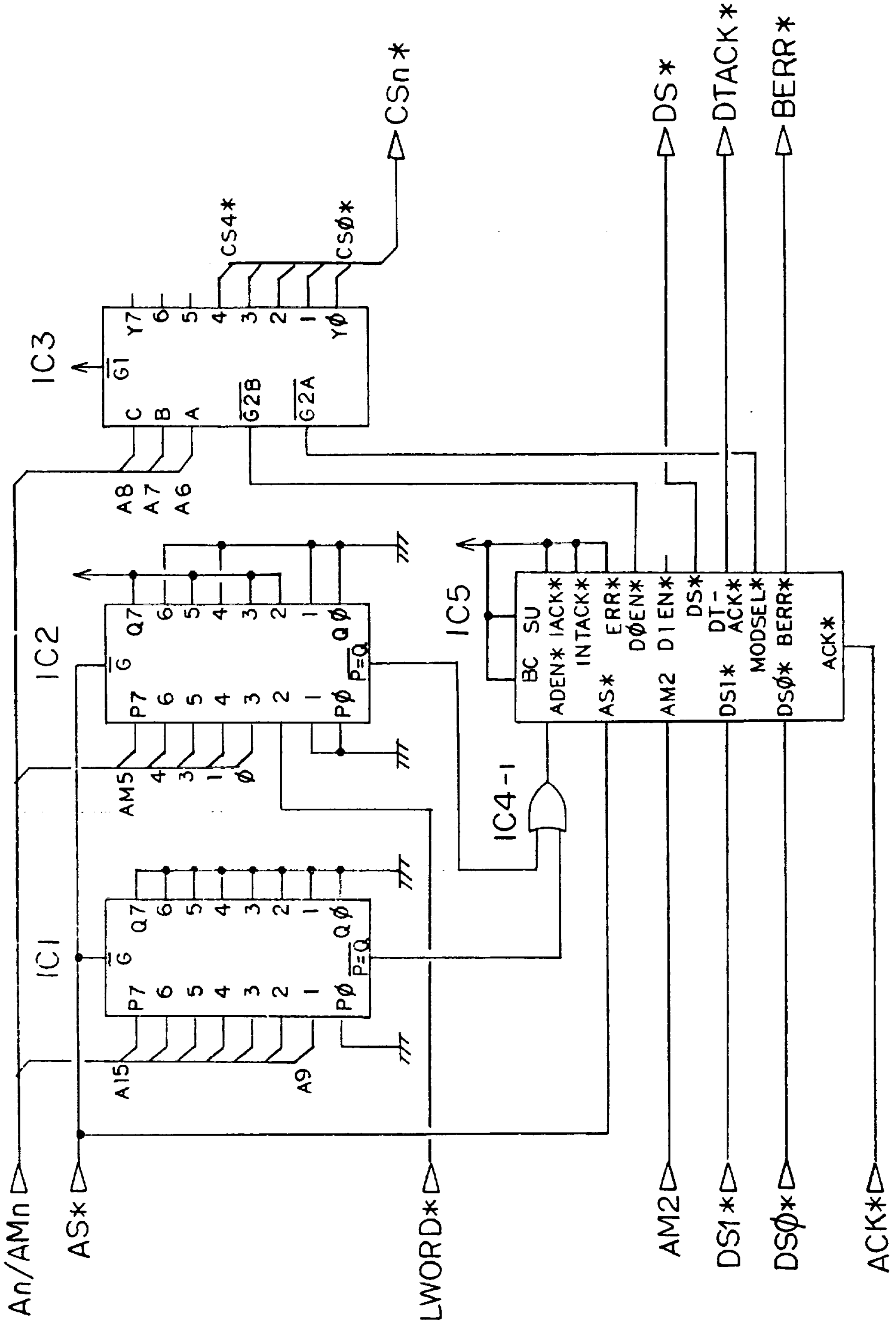
Timing Parameters	Signals	Max. Time(ns) unless otherwise specified		Description
		C-45	M-65	
t1	DWB* to BR* asserted	90	130	If DWB* is asserted after LAS*
t2	LAS* to BR* asserted	90	130	If LAS* is asserted after DWB*
t3	BR* to BG asserted	0	0	VME 1210 only when internal BR* generated (BG connected to BGIN)
		45	65	VME 1210 only when external BR* received (BG connected to BGIN)
		System arbiter time	System arbiter Time	VME 1220 only
t4	BGIN to BBSY* asserted	125	185	VME 1210 only, includes delay line: 55ns for M-65, 45ns for M-55, 35ns for C-45, 40ns for C-35, 60ns for C-25 part
		135	195	VME 1220 only
t5	BBSY* to BR* negated	45	65	
t6	BBSY* to DHBA* asserted	45	65	
t7	BBSY* to BGIN negated	45 max	65	VME 1210 only
		35 min	55 min	
		System arbiter time	System arbiter time	VME 1220 only
t8	DHBA* to DHBD* asserted	45	65	
t9	DHBA* to WRITE* asserted	45	65	Conditional upon R/W* value
t10	DHBA* to AS* asserted	90 70 (min.)	130	Ensures 35ns minimum address to AS* and data to DSn* set up times
t11	AS* to DSn* asserted	45	65	
t12	BGIN to BBSY* negated	80 max	120 max	VME 1210 only;
		70 min	110 min	VME 1210 only; t7min + t12min ≥ 90 ns min. BBSY* assertion
		135 max	195 max	VME 1220 only
		105 min	165 min	VME 1220 only. (see note below)
t13	BREL to BBSY* negated	45	65	Valid only when BREL is asserted after BGIN is negated
t14	DTACK* to LDTACK* asserted	45	65	
t15	LDTACK* to LAS*/LDSn* negated	@ Local master	@ Local master	Local master's time to negate strobes
t16	LAS* to DHBA* negated	45	65	If DWB* already negated
t17	DWB* to DHBA* negated	45	65	If LAS* already negated
t18	LAS* to AS* negated	50	72	
t19	LDSn* to DSn* negated	50	72	
t20	LDSn* to DSn* negated	50	72	
t21	DSn* to WRITE* negated	45	65	Ensures 10ns hold time
t22	DSn*/DTACK* to LDTACK* negated	45	65	Earliest negation of DSn* or DTACK* causes LDTACK* to be negated.
t23	BGIN to BGOUT* asserted	90	130	VME 1220 only
		25+d,35+d,45+d	55+d,65+d	VME 1210 only
t24	BGIN to BGOUT* negated	45	65	
t25	Latest of LAS*/DWB* to AS* asserted	135	195	Assertion time when already have bus (BBSY* asserted).
t26	Latest of DHBD*/LDS* to DS* asserted	45	65	Assertion time when already have bus (BBSY* asserted)

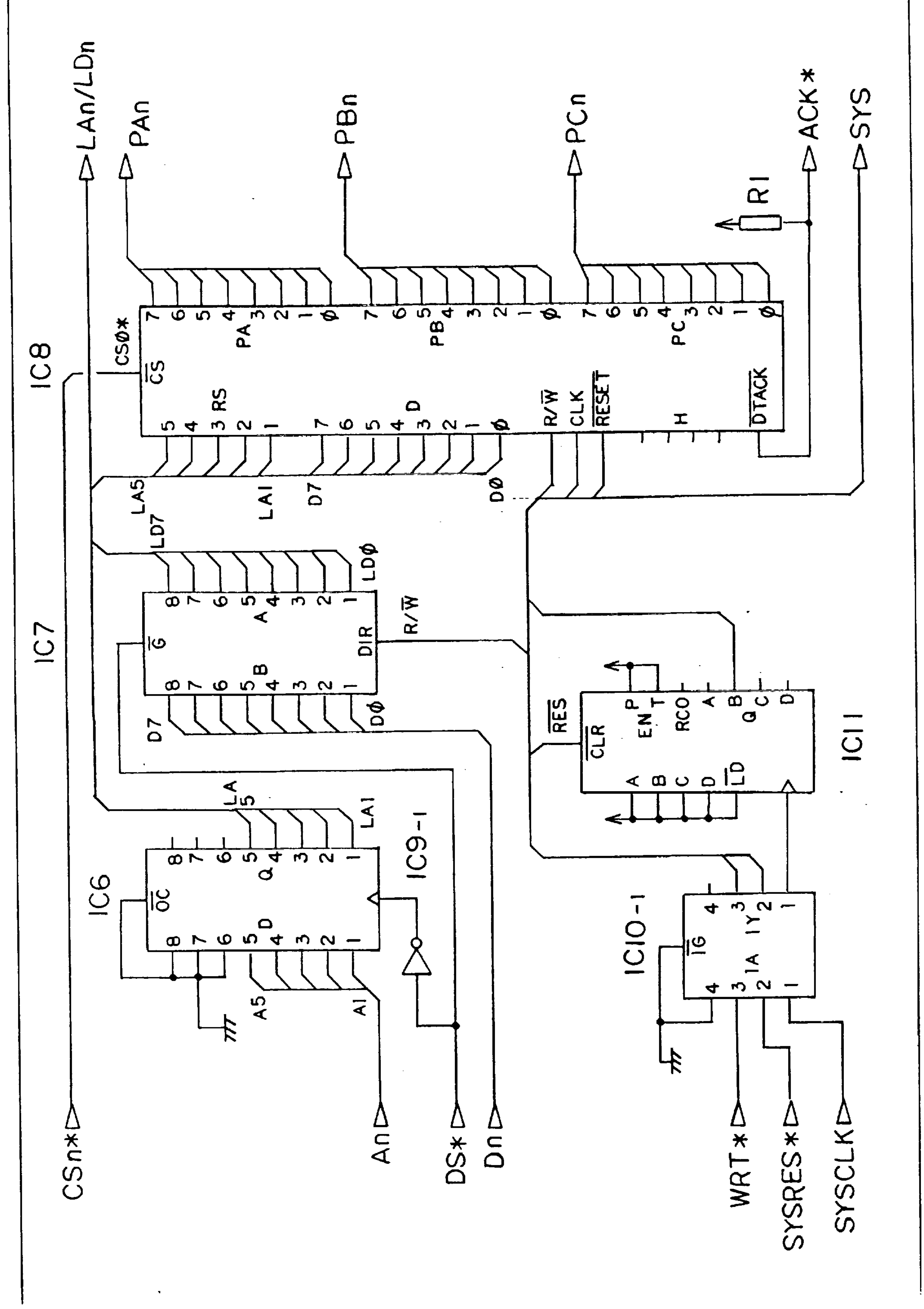
Note:
BBSY* is guaranteed to be asserted for a minimum of 90 ns in the VME 1210A devices and the C-45 device of the VME 1220A, even if BGIN is negated immediately after BBSY* is asserted. For the C-35 and C-25 VME 1220A devices, the sum of the system arbiter "BBSY* asserted to BGIN* negated" time and the t12 minimum time on the VME 1220A must be greater than 90 ns. Generally, this time will be taken up completely by the system arbiter time, however, if not, a delay line can be connected between pins 8 and 16 (DIP) or pins 10 and 19 (LCC) on the VME 1220A device to guarantee the 90 ns minumum. For example, if the system arbiter "BBSY* asserted to BGIN* negated" time was 35ns (min), no delay line would be needed for the C-35 VME 1220A device, since 35 + 75 > 90. However, a 10 ns delay line would be required for the C-25 VME 1220A.

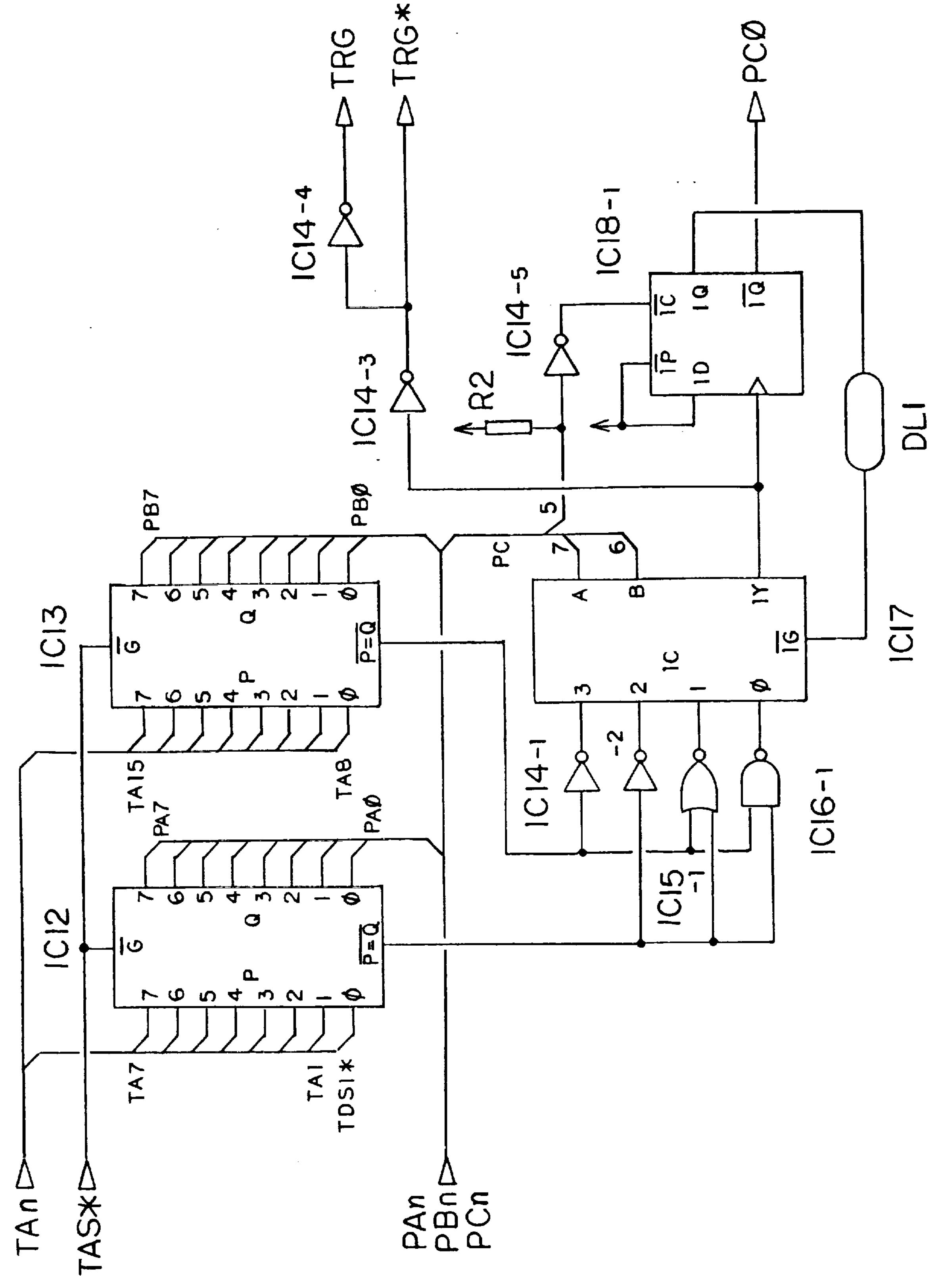
APPENDIX B

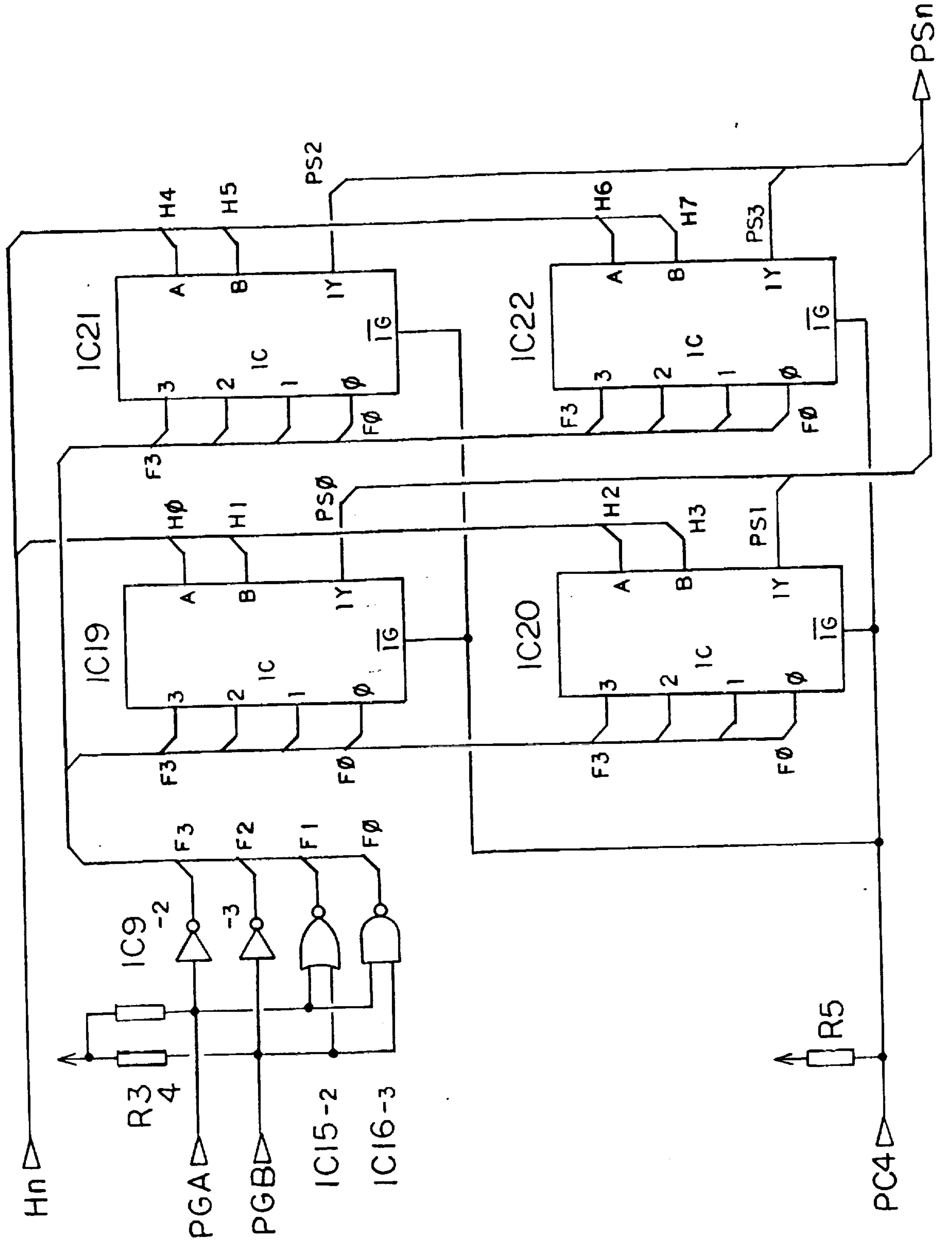
FAULT INJECTION MODULE SCHEMATIC DIAGRAMS

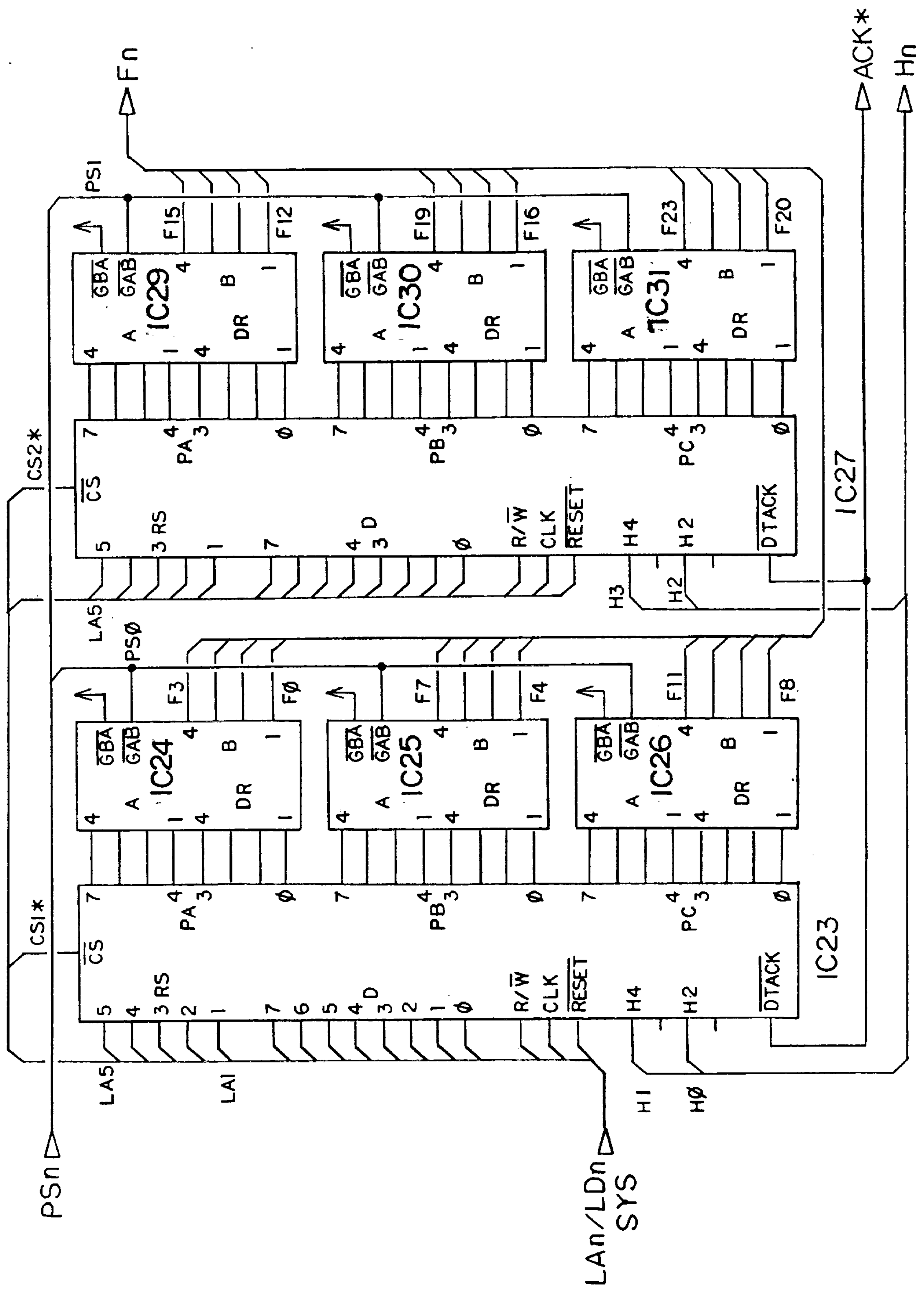
Ver. 1.0

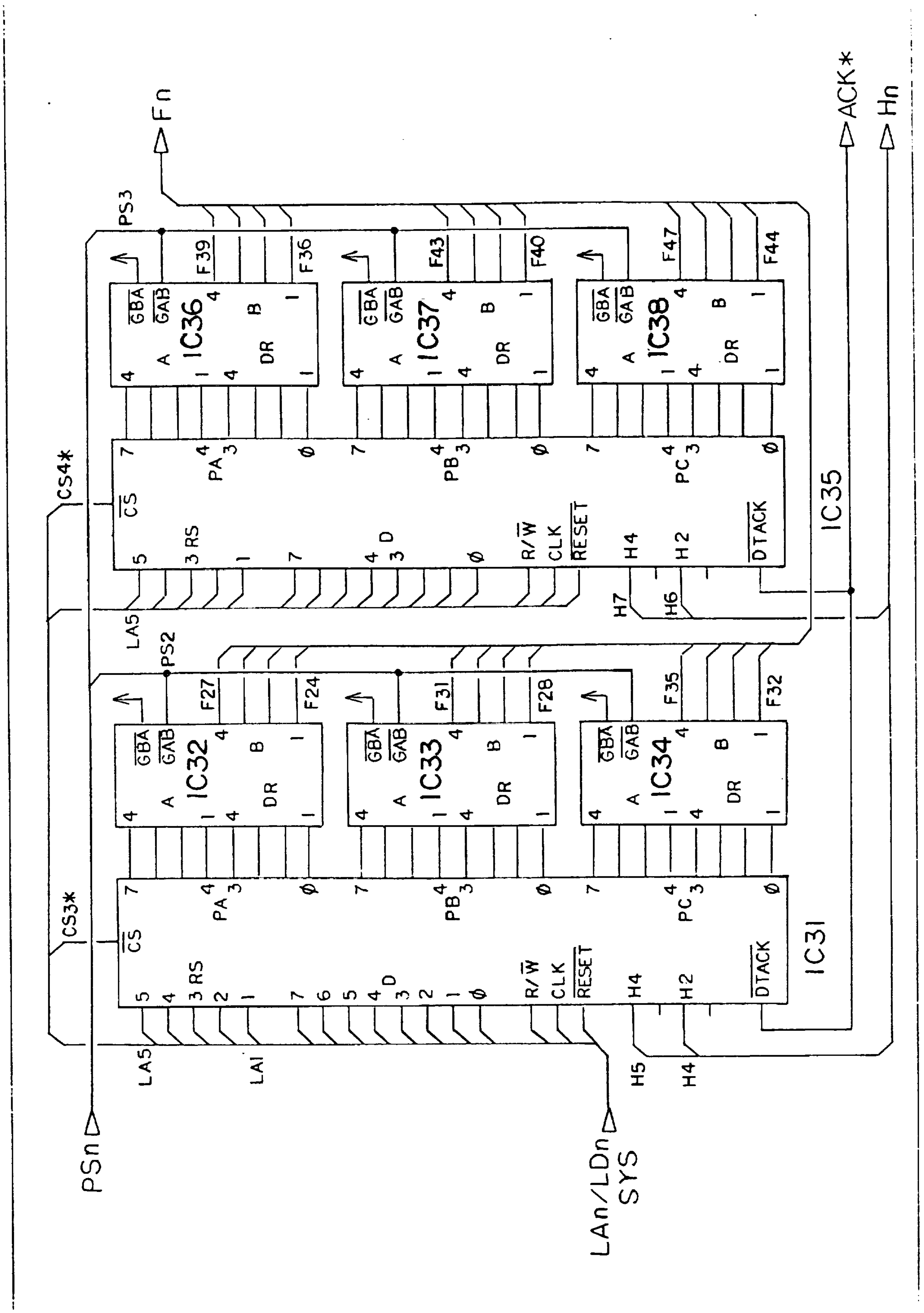

IC1
IC2
IC3
IC4-1
IC5
A15
A9
AM5
4
3
1
Ø
A8
A7
A6
P7
6
5
4
3
2
1
PØ
G
Q7
QØ
P=Q
C
B
A
G1
G2B
G2A
Y7
YØ
CS4*
CSØ*
CSn*
BC
SU
ADEN*
IACK*
INTACK*
AS*
ERR*
DØEN*
AM2
D1EN*
DS*
DS1*
DT-ACK*
MODSEL*
DSØ*
BERR*
ACK*
DTACK*
An/AMn
LWORD*


CSn*
An
DS*
Dn
WRT*
SYSRES*
SYSCLK
LAn/LDn
PAn
PBn
PCn
ACK*
SYS
R1
IC6
IC7
IC8
IC9-1
IC10-1
IC11
CS0*
CS
RS
D
PA
PB
PC
R/W
CLK
RESET
H
DTACK
LA5
LA1
D7
D0
LD7
LD0
G
A
B
DIR
OC
Q
A5
A1
RES
CLR
EN
P
T
RCO
LD
QC
1G
1A
1Y


TRG
TRG*
PC∅
IC14-4
IC14-3
IC14-5
IC18-1
R2
1C
1Q
1P
1D
DL1
PB7
PB∅
PC
5
7
6
A
B
1Y
1G
IC17
IC13
G
Q
P
P=Q
TA15
TA8
PA7
PA∅
IC12
TA7
TA1
TDS1*
IC14-1
-2
IC15 -1
IC16-1
TAn
TAS*
PAn
PBn
PCn


Hn
PGA
PGB
IC15-2
IC16-3
R3
4
IC9
-2
-3
F3
F2
F1
FØ
IC19
IC20
IC21
IC22
1C
A
B
1Y
1G
3
2
1
Ø
HØ
H1
H2
H3
H4
H5
H6
H7
PSØ
PS1
PS2
PS3
PSn
R5
PC4


Fn
ACK*
Hn
PS1
PSØ
CS1*
CS2*
F15
F12
F19
F16
F23
F20
F3
FØ
F7
F4
F11
F8
IC24
IC25
IC26
IC29
IC30
IC31
IC23
IC27
GBA
GAB
A
B
DR
CS
RS
PA
PB
PC
D
R/W
CLK
RESET
H4
H2
DTACK
LA5
LA1
H3
H2
H1
HØ
PSn
LAn/LDn
SYS



PSn
LAn/LDn SYS
Fn
ACK*
Hn
CS3*
CS4*
PS2
PS3
LA5
LA1
H5
H4
H6
H7
IC31
IC32
IC33
IC34
IC35
IC36
IC37
IC38
F24
F27
F28
F31
F32
F35
F36
F39
F40
F43
F44
F47
GBA
GAB
A
B
DR
CS
RS
D
R/W
CLK
RESET
H4
H2
DTACK
PA
PB
PC

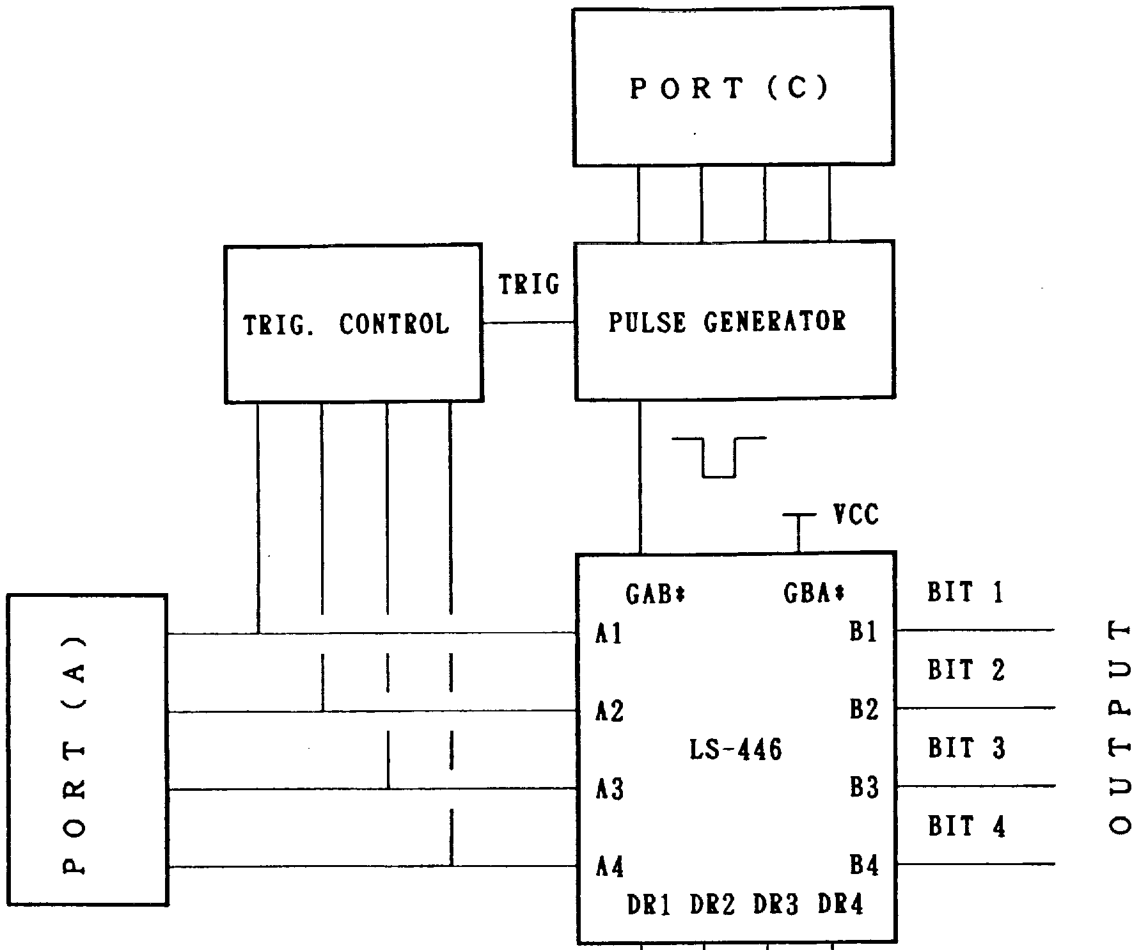
Fault Injection Module
Parts List (1)

Ref No.	Part Number	Size	Description
IC1	SN74ALS520	20	8-bit Identity Comparator
IC2	SN74ALS520	20	8-bit Identity Comparator
IC3	SN74ALS138	16	3 to 8 Decoder
IC4-1	SN74ALS32	14	Quad 2-Input OR Gates (1/4)
IC5	VME 2000	24[1]	Slave Module Interface Device
IC6	SN74F374	20	Octal D-Type Flip-Flops
IC7	SN74LS645-1	20	Octal Bus Transceivers
IC8	MC68230 P8	48	Parallel Interface/Timer (PIT-0)
IC9-1	SN74ALS04B	14	Hex Inverters (1/6)
IC10-1	SN74LS244	20	Octal Buffers (1/2)
IC11	SN74ALS161B	16	4-bit Binary Counter
R1		8[2]	R Network, seven 4.7kΩ (1/7)
IC12	SN74ALS520	20	8-bit Identity Comparator
IC13	SN74ALS520	20	8-bit Identity Comparator
IC14-1	SN74ALS04B	14	Hex Inverters (1/6)
IC14-2	SN74ALS04B	14	Hex Inverters (2/6)
IC14-3	SN74ALS04B	14	Hex Inverters (3/6)
IC14-4	SN74ALS04B	14	Hex Inverters (4/6)
IC14-5	SN74ALS04B	14	Hex Inverters (5/6)
IC15-1	SN74ALS02	14	Quad 2-Input NOR Gates (1/4)
IC16-1	SN74ALS01	14	Quad 2-Input NAND Gates (1/4)
IC17	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC18-1	SN74ALS74A	14	Dual D-Type Flip-Flops (1/2)
R2		8	R Network, seven 4.7kΩ (2/7)
DL1	RWT050P	14	50ns Delay Line

[1] 300mil 24 pin DIP
[2] Single-in-line package

Fault Injection Module
Parts List (2)

Ref No.	Part Number	Size	Description
IC9-2	SN74ALS04B	14	Hex Inverters (2/6)
IC9-3	SN74ALS04B	14	Hex Inverters (3/6)
IC15-2	SN74ALS02	14	Quad 2-Input NOR Gates (2/4)
IC16-2	SN74ALS01	14	Quad 2-Input NAND Gates (2/4)
IC19	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC20	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC21	SN74ALS153	16	Dual 1 of 4 Data Selectors
IC22	SN74ALS153	16	Dual 1 of 4 Data Selectors
R3		8	R Network, seven 4.7kΩ (3/7)
R4		8	R Network, seven 4.7kΩ (4/7)
R5		8	R Network, seven 4.7kΩ (5/7)
IC23	MC68230 P8	48	Parallel Interface/Timer (PIT-1)
IC24	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC25	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC26	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC27	MC68230 P8	48	Parallel Interface/Timer (PIT-2)
IC28	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC29	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC30	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC31	MC68230 P8	48	Parallel Interface/Timer (PIT-3)
IC32	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC33	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC34	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC35	MC68230 P8	48	Parallel Interface/Timer (PIT-4)
IC36	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC37	SN74LS449	16	Bus Transceviers w/ Bit dir.
IC38	SN74LS449	16	Bus Transceviers w/ Bit dir.



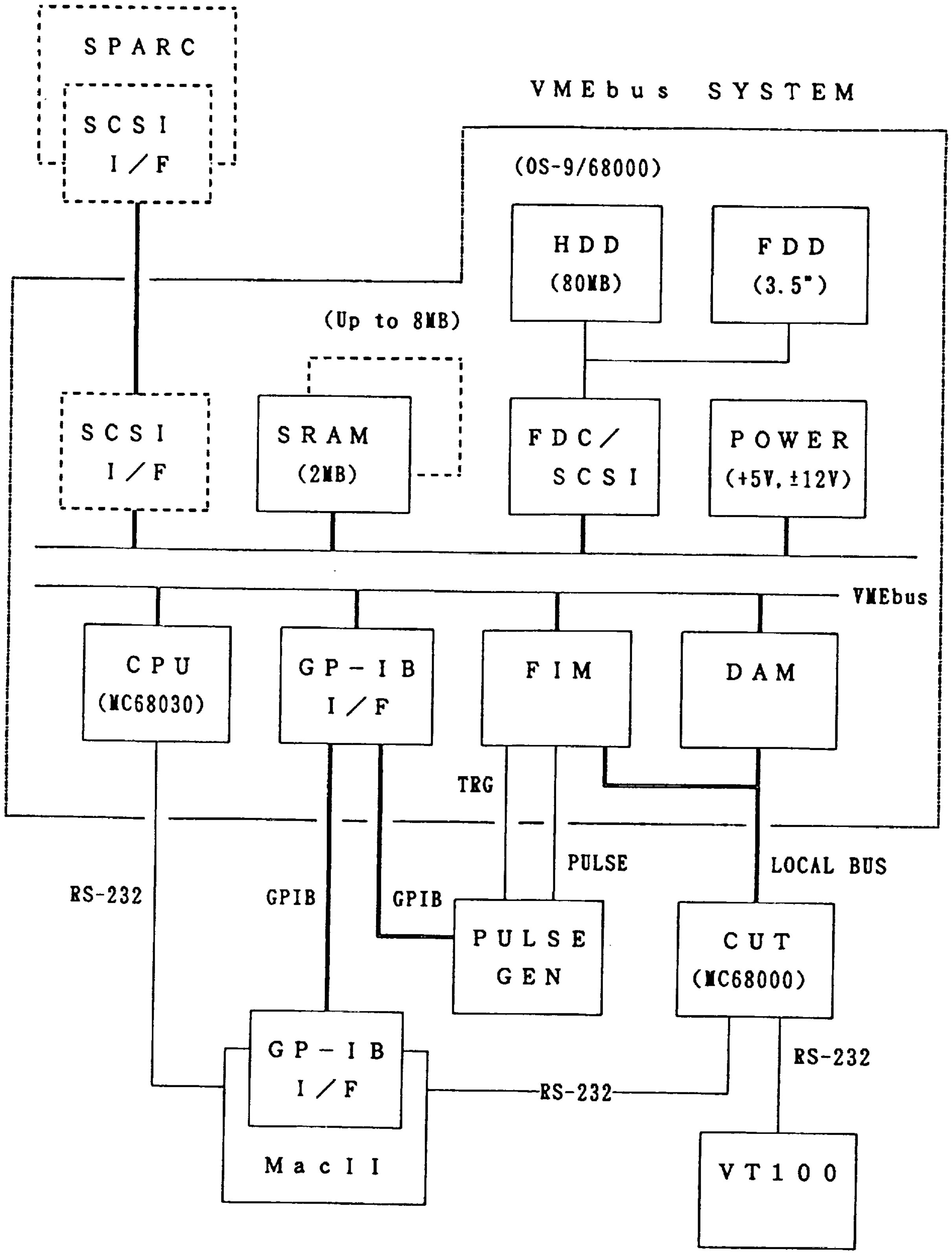
LS-446

GBA*	GAB*	DRn	OPERATION
H	H	X	ISOLATION
H	L	H	A* TO B
H	X	L	ISOLATION

FIG. FAULT INJECTION MODULE (4-BIT)

Additional Components for the New Experimental System

Part No.	Manufacturer	Description	Cost ($)
MZ 7500	MIZAR	GPIB Interface Board for VMEbus	695.00
	MIZAR	Single Cable for MZ 7500	75.00
MacII488	IOtech	GPIB Controller Board for Mac II	535.00
PFG5105	Tektronix	Pulse Generator (demo)	2,471.25
PFG5105	Tektronix	Pulse Generator (new)	2,800.75
TM5006	Tektronix	Prog. Mainframe (demo)	851.25
FIM	JHU	48ch Fault Injector	
Mac II	Apple	Macintosh II	
SPARC	Sun Micro.	SPARCstation work station	



FAULT INJECTION EXPERIMENTAL CONFIGURATION

Targeted Features of the Fault Injection Module

- **Fault Injector**
  - Provides 48 channels with bit-definable outputs using four PI/T (MC68230) and twelve bus transceiver (74LS446) chips.
  - Supports three output states (0, 1, and Z[1]) on each channel.
  - 2ch pulse generator is installed as a source of fault injections.
  - Supports single/multiple faults of stuck-at-0/1 types with duration varying from 40 ns to 99.9 ms.
- **Word Recognizer**
  - Provides a versatile trigger source for the fault injection and data acquisition.
  - Implements 16-bit word recognizer using a MC68230 PI/T and two 74LS686 magnitude comparators.

[1]Z: High-impedance