1979

Automated synthesis of digital hardware

Alice C. Parker
Carnegie Mellon University

Louis Hafer

AUTOMATED SYNTHESIS OF DIGITAL HARDWARE

by

Alice C. Parker[*] and Louis Hafer[*]

DRC-18-12-79

May 1979

Index Terms

logic design, automated synthesis, computer-aided design, design automation, register-transfer level design, hardware specification, ISP, hardware-descriptive language

620.0042
028.
DRC-10-12-79

ABSTHACT

This paper describes a portion of the register transfer level computer aided design (RT-CAD) research at Carnegie-Mellon University. This part of the research involves the design and implementation of a data-memory allocator, consisting of a set of algorithms and data structures which synthesize hardware at the logical level from an ISPL language description. Preliminary results indicate the allocator's performance compares favorably with a human designer when designing an elevator controller and a reduced PDP-8/E.

1.0 INTRODUCTION

The motivation behind the research described in this paper is to enhance the digital designer's capabilities by producing more powerful design tools. Digital logic design has progressed to the point that the operation of the logic can be functionally expressed by a variety of hardware descriptive languages, ISP being one of the more widely used behavioral languages. Functional simulators exist and are useful for verifying system operation and performance measurements (Barb77a). Thus, the state of the art in digital design is such that the next addition to design aids should be a synthesis program; a program that can design the STRUCTURE of a digital system, given its FUNCTION or

BEHAVIOR as input. Along with automated synthesis of hardware comes the problem of producing optimal or near optimal designs to meet the design constraints. The research reported on here is aimed at understanding the design or synthesis process so that it can be automated. A major goal of this project is to produce logic-level hardware designs from ISP descriptions in a non-optimal fashion, to better understand the automated design process. A parallel goal of related research by the same group is to develop discrete optimization algorithms and techniques to be applied in a more complex and powerful synthesis package.

This paper describes the operation and performance of one of the synthesis routines - termed an "allocator.[11] This routine designs logic in the distributed design style from an ISPL description. The distributed design style is a way of designing digital logic characterized by many storage locations and operators interconnected in a non-regular manner, with little sharing of hardware. The distributed design style is often implemented with TTL circuits or on a single IC. It is most often used to produced designs where speed of operation is a desirable goal. The ISPL* language used as input for this allocator allows the designer to specify the behavior of digital hardware, without describing the structure.

2.0 THE DESIGN AUTOMATION SYSTEM

---

*The allocator described here used ISPL (Barb76) but has been converted to ISPS (Barb 77b), a more powerful, newer version of ISP.

In order to discuss the role of the allocator and results of the synthesis effort, an overview of the RT-CAD (Register-Transfer level Computer-Aided-Design) system is presented here.   This overview was originally published in (Snow78a).

RT-CAD OVERVIEW

The ultimate goal of the RT-CAD project is to provide a technology-relative, structured-design aid to help the hardware designer explore a larger number of possible design implementations. Inputs to the system are a behavioral description of the system to be designed, an objective function which specifies the user's optimization criteria, and a library specifying the hardware components available to the design system.   The components of the RT-CAD system are shown in Figure 2.1 and discussed below.

The RT-CAD system differs from other design automation systems in that it operates from a behavioral specification. Such specification provides a model that, while accurately characterizing the input-output behavior of a piece of hardware, does not necessarily reflect its internal structure. The design process is one of binding implementation decisions in a top-down manner as a design proceeds through the RT-CAD system. More and more structural detail is frozen at each level until a complete hardware specification is obtained, the most influential design parameters being bound first in order to cut down the design search space.   The functions of the design system components which bind these implementation decisions are described

below.

GLOBAL OPTIMIZER. The global optimizer applies high-level transformations to a design's behavioral representation after translating it from ISPS notation (Barb77b) to an abstract design representation called the value trace (Snow78b). The transformations have a significant impact on the cost, performance, and other parameters of the designs to which they are applied.

DESIGN STYLE SELECTOR. By considering the various module sets that can be used (e.g., TTL vs. a microprocessor), the design constraints imposed (e.g., cost, speed), and the structure of the algorithm to be designed (e.g., pipeline data flow), the design style selector decides on the specific style of design to be employed (e.g., bit-slice microprocessor, MOS microprocessor, SSI/MSI logic). Earlier work (Thom77b) shows this to be an influential decision in terms of cost and speed tradeoffs. When the style is selected, the design is passed to an allocator specific to the design style. Initial research into the design style selection process has been completed (Thom 77a) and an automatic design style selector is currently being programmed.

PARTITIONER. The partitioner groups operations from the abstract design representation into control steps. This effectively binds the control flow for the design. Tradeoffs between the data and control parts are made at this level.

DATA/MEMORY (DM) ALLOCATOR. The function of the DM allocators is to decide the number and type of data operators, multiplexers, and registers needed to implement the data part of the design. They are style specific in that they embody analytic and heuristic knowledge about a style (e.g., the trade-offs involved in the design of a TTL system), but they do not have access to the specific details of each module set. The output of the allocator is a data path graph whose nodes are elements such as adders or registers. The edges represent actual circuit interconnections.

CONTROL ALLOCATOR: The control allocator generates a sequential state machine to control the data paths produced by the DM allocator. The control allocator has the option of designing the control unit around control philosophies such as microprogramming, programmed logic arrays, random logic, etc. The output of the control allocator is a control path graph whose nodes represent control states, and whose edges represent permissible state transitions.

MODULE BINDER. The module binder selects physical modules from the module set library to implement a design's data and control path graphs. The library contains descriptions of the components available to the design system and may be freely updated so that it is kept current with respect to advances in module technology. This dynamic aspect of the module set library provides for the technology-relative aspects of the RT-CAD system.

PHYSICAL LAYOUT PROCESSOR. This component partitions the system into printed circuit boards or chips, decides the placement of components, routes interconnections, and prepares engineering documentation.

Research is currently underway into the design of all of the system components described above. In addition, the problem of integrating them into a coherent design system is being investigated.

Research described in this paper has focused on a synthesis routine - the data-memory allocator. Although the control allocation effort is just beginning, some ideas as to the nature of the problems to be solved have been posed. The generation of control hardware is analogous to the problem of generation of microcode, with its inherent computational complexity, but there is one difference. Generation of hardware introduces another set of variables into the optimization routines. Not only are microinstructions generated, but the control hardware itself must be designed and optimized.

Previous work on data-memory allocation techniques includes the Macromodule and Register Transfer Module allocators (Barb75) and the allocation algorithm developed by Rege (Rege7*0. In each of these, however, a small, tightly constrained module set was assumed available for implementing the device description. Detailed knowledge of the physical structure of the modules was fixed in the allocation algorithms to take advantage of implementation tricks peculiar to each module set. The allocator described here uses more abstract logical modules and algorithms which are independent of detailed knowledge of

the modules used to implement the physical hardware. This in turn allows greater freedom in future attempts at optimization, due to the richer physical module set which can be used.

3.0 OVERVIEW OF THE DATA-MEMORY ALLOCATOR

After the ISPL functional description of the system to be designed has been manipulated by the higher level design routines to better achieve the price/performance objectives provided by the user, it is used as input to the allocation routines. The allocators fall into two categories, data-memory and control. The data-memory allocators basically perform a mapping function from the algorithmic description to the data part of the hardware implementation. The data part consists of the data storage elements, data operators, and data paths necessary to implement the operations specified in the algorithmic description. It should be noted that due to the characteristics of the ISPL language this mapping may be multi-valued in either direction, rather than a simple one-to-one translation. The control allocators perform a slightly different function, mapping the timing, sequencing, and branching information implicit in the ISPL description onto control states, control signals, and conditional branching signals to control the data part. Again, the mapping is not a simple one.

The allocator described here is a data-memory allocator for the distributed logic design style. As pointed out earlier, the allocator itself is technology relative and the mapping onto specific integrated

circuit packages is performed by the data base module, which is updated as new packages are added to the module sets. It should be understood that the process referred to as allocation throughout the remainder of this paper is a logical allocation in terms of data storage elements, a set of operator primitives defined by the 1SPL language, data paths, and multiplexers.

The first version of the allocator is experimental, and it performs only minor optimizations on the allocated hardware. Rather, it has been designed to investigate:

* the feasibility of performing the mapping from ISPL to hardware

* the independence of the allocator from a specific technology

* information necessary to the design process which cannot be expressed in ISPL

* the types of data structures needed for allocation

* bounds on the size of the ISPL description that can be processed by the system

* exceptional constructs allowed in ISPL which may be difficult or impossible to design or implement in hardware

* types of error checking that can be performed by the allocator

* areas where optimizations are possible in future, more sophisticated allocators.

In addition, the allocator has been designed as a possible skeletal structure for future allocators in order to standardize input/output formats and data structures.

In the following description of allocator structure and function, an attempt is made to extrapolate details learned through implementation into a basic philosophy of allocator design. The procedure used by the allocator might be compared to a two pass compilation. The first pass may be considered a syntax or feasibility check. The allocator inputs a parsed ISPL description, constructs data structures analogous in function to symbol tables, and enforces various constraints necessary to insure that the data storage locations, logical mappings*, and input/output interface characteristics specified in the description can be implemented in hardware. If no errors are encountered, it proceeds to allocate the basic data storage structures called for in the description, and any additional data paths, storage, and operators necessary to implement variable accessing schemes described by the logical mapping facility of ISPL. The second pass may be considered the semantic phase with the activity of code generation replaced by the allocation of data paths, operators, and additional storage as needed to implement the functions specified in the ISPL description. Parallelism analysis is performed at several levels to warn the user of error conditions and determine constraints relating to optimization of hardware. The allocation is then completed by the addition of multiplexing where required.

---

*The mapping capabilities in the ISPL language will be discussed later in Appendix A.

It is important to realize, however, that allocation differs from compilation in that in a compilation one is concerned with implementing the specified data operations on a fixed data part whose capabilities are known a priori. In allocation, the allocator must be able to recall and utilize the capabilities of a data part which is being dynamically created. The allocator thus works from the inside out, first creating the data storage and access structures, and then adding the necessary data paths and operators to perform the described data operations. In addition, the output of the allocator is a non-planar directed graph, rather than a linear list of compiled instructions.

The detailed description of the allocator will focus on the allocator inputs, data structures, algorithms, and outputs.

3.1   Basic Concepts in Allocation of Data Paths

Before proceeding to the detailed description of the allocator two fundamental concepts should be defined. These are the process and the operation path.

A process is defined as a sequence of actions occurring within a control environment which is independent of and asynchronous with respect to other control environments. No assumptions can be made about the relative timing of events across process boundaries, and no resources other than those explicitly specified by the input functional description may be shared by multiple processes. Complications introduced by allowing multi-process descriptions will

be explored in detail in later sections.

The operation path is defined, in the most general case, to consist of two data sources, an operator, and the data paths leading from each source to the operator. This is the basic conceptual unit produced by the allocator to implement an action specified in the functional description, although only those elements of the operation path necessary to perform the particular action are actually allocated. (An example would be a two's complement negation. Only one source, the operator, and a connecting data path are necessary; the other source and data path remain null.) Considerations motivating the choice of this fundamental unit include:

* The compiled ISPL functional description used as input to the allocator is expressed in the same form, using two sources and an operation code

* The operation path is the minimal unit which can be considered when analyzing for possible parallel usage conflicts during optimization to reduce the number of allocated operators.

No destination for the result value is included in the operation path as the destination and the data path from the operator to the destination are not subject to the same usage constraints as the operation path. Throughout the following description of the allocator, the term path will be used to refer to the operation path as defined here. If the destination and the data path from the operator to the destination are to be included, the combination will be referred to as the extended path. The physical data path between a source and a destination will be referred to as a link or data link.

3.2 Allocator Inputs

The primary input to the allocator is the compiled ISPL functional description, in the form of a symbol table and a statement table. A small portion of the ISPL description of an elevator controller is shown in Fig. 3-1, along with relevant portions of the symbol table and statement table. This description will be used as a running example in describing the allocator. Detailed documentation of the ISPL language and compiler may be found in (Barb76).

The compiler tables are augmented with information supplied by the user in a "technical file." This file contains two types of information:

* global information about the description which cannot be specified in ISPL

* interface information to be passed to the module data base for selection of specific IC^fs. This information describes the desired logical and electrical characteristics of the inputs and outputs of the device being allocated.

Relevant portions of the technical file for the elevator controller are shown in Fig. 3.2. The PROCESS specifications indicate there are two processes in the complete description, LOOK.FOR.CALLS (the example ISP of Fig. 3.1) and MAIN. The INPUT specifications describe the desired electrical characteristics of the input lines, and the OUTPUT specifications present the characteristics of the output lines.

3.3 Allocator Data Structures

The allocator builds seven major data structures for use during the allocation of an ISPL description.

The process table contains one entry for each process specified in the technical file. Each entry contains a listing of all variables and called procedures used in the process. The called procedure table contains one entry for each called procedure declared by the user in the ISPL description. Data recorded in the table for each procedure includes the number of locations where the procedure is called and an indicator if it is shared by two or more processes. The allocator symbol table contains one entry for each variable which will be allocated a node in the path graph by the allocator. Each entry records all physical information relevant to the entry type for use by the allocator during logical allocation and by the module data base during physical allocation. As in the called procedure table, an indicator flags the entry if access to it is shared between processes. These three tables, in conjunction with the compiler symbol table, comprise the portion of the data structures which function as symbol tables for the allocator.

The path graph contains the complete allocated or designed data part of the device and is the major output of the allocator. As the name implies, the form of this structure is a directed graph, with nodes describing the variables, operators, switches, and data links, and the actual data flow described by the directed edges. In the general case, this graph may be non-planar, cyclic, and disconnected. The basic path graph groups are described in Fig. 3.3. A complete

path graph for the example ISP description of Fig. 3-1 is shown in
Fig. 3.4.

The path table is the major working data structure of the
allocator. It is dynamically created and purged for each process that
is allocated, and contains all paths allocated for the process
organized in the form of a three-dimensional sparse matrix. Each
dimension is indexed by one of the defining nodes of the path, which
are the two sources and the operator. This allows the allocator to
access information about storage locations and operators without
exhaustive searching.

The path parallelism table, also created and cleared for each
process, contains the record of all concurrent uses of the paths
recorded in the path table, stored in the form of path pairs. It is
used to perform parallelism error checking at the variable level. The
operator table, although conceptually a symbol table, is used
primarily as a more convenient record of allocated operators than the
path graph. It is updated at the end of each process from the path
table before the path table is purged.

Although the internal implementation of the data structures is
not of interest here, certain overall characteristics should be
discussed. Due to the diversity and quantity of information required
by the allocator, and the need to access this information in a variety
of orderings with a variety of keys, the tables are heavily cross
referenced and contain varying amounts of redundant information. This
diversity of structure allows the allocator to efficiently locate

information and relate it to both the original 1SPL description and the allocated data path. As an example, the path graph and the path table are highly redundant. During the allocation of a process, both contain complete information on the paths allocated to implement the process actions. However, locating a particular path in the path graph is a combinatorial search, involving the comparison of all possible pairs of output links from each source to determine if data links exist from each source to the same operator. The operator must then be checked to determine if it can perform the desired function on the data. Locating this same path in the path table is a linear search, involving only the location of a particular element in the three-dimensional matrix.

The dynamic nature of the path data structures allows the allocator to handle ISPL descriptions of any size, providing they are suitably divided into processes. This is because the total size of the data structures is limited only by the amount of core memory available to the allocator, and the purging of the path table and path parallelism structures after each process allows a large percentage of this space to be reused. Only the path graph and operator table increase in size monotonically from process to process.

A rough measure of the size of the data structures can be obtained in terms of the average number of 36 bit memory words required for a variable or a statement table operation. Approximately 45 words of storage are used per variable, and 55 words per statement table operation. Additional storage for parallelism information is

allocated at the rate of roughly five words per pair of parallel paths, with four words of overhead for each parallel level or branch.

3.4 Allocator Algorithms

3.4.1 Global perspective

The allocation of an ISPL description begins with the allocator performing several checks and preparatory actions under the heading preliminary processing. Implementation restrictions on mappings (see Appendix A) are enforced, statistical measures are gathered, the technical file is processed, and array memories (RAMS) are given a uniform accessing structure. Next, a variable and procedure usage analysis is performed to determine which processes use the variables and procedures to be allocated. Synthesis begins with the initialization of the path graph by allocating path graph nodes for the declared variables and mappings. A universal variable access mechanism is set up to access these nodes. All the above actions are performed for the entire ISPL description, completing the "global* phase of the allocation. The second phase of the allocation might be termed the "process" phase, with the actions of extended jcaJUl allocation, parallelism analysis, and optimization being performed independently for each process. Path allocation and parallelism analysis are performed for each statement table entry in the process, while optimization is performed over the process as a whole. When this process phase has been completed for all processes in the description, some final processing of the path graph as a whole is done to place multiplexers where needed.

3.4.2 Preliminary processing

The allocator begins the processing of the ISPL description by enforcing the implementation restrictions on logical mappings. The mapping capability of ISPL allows the writer to define a register or memory, then to "map" another register or memory over the first. The base memory can then be accessed either by its name or the mapped memory name. The bit width and number of words in the base and mapped memories need not be the same as long as the _total_ number of bits is equivalent. The mapping capability and the constraints imposed by the allocator are defined in Appendix A. The major objectives considered in developing these constraints were:

* To be able to implement the addressing and access algorithms implicitly defined by the mappings with a reasonable amount of hardware, both in the data part and in the control part.

* Since the addressing and access algorithms are implicitly defined, rather than explicitly specified, to develop a regular structure capable of implementing all mappings by creating a well defined basic pattern.

* Since mapping definitions can be cascaded, to be able to cascade the structure used to implement each mapping.

* To restrict the use of the mapping construct as little as possible.

Mappings satisfying the constraints defined in the appendix allow an addressing and access structure compatible with the first three objectives. Mappings not satisfying these constraints are revealed as fatal errors detected during the mapping check, causing the allocator

to print an error message identifying the erroneous mappings and halt.

If no errors are detected, the allocator reads the PROCESS section of the technical file to determine the identity and quantity of the processes declared by the user. The INPUT and OUTPUT sections of the technical file are now processed and the information obtained stored in the allocator symbol table.

The final step in the preliminary processing is the assignment of memory address (MAR) and data (MDR) registers to the declared array (RAM) memories. The ISPL language does not require explicit declaration of MARs and MDRs, whereas in a hardware implementation a large amount of flexibility is gained in the design of the control part by the presence of these registers. They are therefore automatically assigned, using temporary registers generated by the allocator. If they are later found redundant, they are removed.

3.4.3 Variable and procedure usage analysis

The allocator now analyses the statement table to determine the variables and procedures used in each process. If no variables or procedures could be shared between processes, the task of allocation would be greatly simplified, as each process could be treated essentially as a separate device description with a totally independent data part. The existence of variables and procedures shared between independent, asynchronous control environments introduces several problems.

First, access to the resource (variable or procedure) must be arbitrated in some manner to prevent two processes from attempting to use the resource simultaneously. While the actual arbitration structure will be created by the control allocator, information must be left specifying the process that a data link is associated with. This is necessary to allow the control allocator to create the proper access arbitration structure for the resource- Identifying the links, however, is the least of the problems to be dealt with.

The second difficulty is associated with variables shared between processes. From a veiwpoint internal to a particular process, a shared variable must be regarded as subject to random change by the other processes sharing the variable and cannot be used by the optimization routines inside a given process.

The third difficulty arises when shared procedures are present. Again, from a viewpoint internal to a process, the hardware used to implement the procedure will be subject to random unavailability due to use by other processes. The allocator is capable of two actions in this case. It can absorb the procedure into each process by the creation of separate hardware implementations of the procedure internal to each process, or it can treat the procedure as if it were itself a separate process and allocate an independent hardware incarnation whose use is arbitrated by the control allocator.

Information collected during the usage analysis is stored in the appropriate symbol tables. In addition, statistics are collected on the total number of operations specified in the statement table.

3.4.4 Allocation of declared variables and mappings

The path graph is now initialized by allocating the nodes representing declared variables and mappings. All variables declared by the user are allocated nodes in the graph, representing storage, inputs or outputs. When the basic variables have been allocated, the multiplexers, demultiplexers, operators, registers, and data links needed to implement the addressing and access structures specified by any logical mappings are determined and allocated.

In order that the nodes in the path graph can be easily related to the variables specified in the ISPL statement table, and to overcome several massive difficulties in the form of the tables, a universal variable access pointer is created for each variable and mapping declared by the user. These pointers contain all information needed to create data paths to and from the variable in the path graph, and also indicate any additional actions necessary to access the value of the variable when it is implemented in hardware.

This completes the portion of the allocation referred to in the global perspective as the global phase of the allocation.

Extended path allocation

The allocator now begins to allocate the extended paths necessary to implement the actions specified in the functional description. For each process specified by the user, there is an entry in the process table. The allocator obtains the starting point of the process in the ISPL statement table and begins a statement by statement allocation.

The first step in allocating the extended path is obtaining the access pointers for the sources specified in the statement. Any actions necessary to make the variable value physically accessible, such as reading it from array memory, are allocated and recorded at this time. If both sources must be obtained from arrays, a check is made for accessing conflicts and, if necessary, temporary storage is allocated to hold the first value while the second is accessed. The access pointers are manipulated as required to reflect these actions.

The access pointers and the operation code specified in the statement table entry are now used to access the path table to determine if the desired path has already been created. If it has not, an operator with the required size and operation is created and entered in the path graph, along with data links from the sources to the operator, and the path is entered in the path table. If the path has already been allocated, the bit width of the operator is examined and it is expanded if necessary.

To complete the allocation of the statement table entry, the allocator obtains and examines the access pointer for the specified destination. If the destination is a variable declared by the user, completion of the extended path is straightforward. A data link is created between the path output and the destination, if it is not already present, and any operations necessary to physically store the value, such as writing an array, are allocated and recorded. If the destination specified is a temporary variable generated by the ISPL compiler, a more complicated series of actions is initiated to

determine if a temporary is necessary, or if the value can be left at the path output for direct input to the next path in sequence.

When the destination has been fixed and a data link created if needed, the necessary hardware actions for the extended path are recorded by adding information to the statement table specifying the exact location in the path graph of the sources, operator, destination, and data links used to implement the instructions. This information is recorded as each extended path is allocated. The number of extended paths used to implement an entry in the statement table can vary from zero to seven, depending on the operation and the types of variables used as the sources and destination.

3.4.5 Temporary allocation

Temporary variables are generated and used by the 1SPL compiler using the criterion "if a temporary is available of the proper size and not currently holding a value, reuse it; otherwise, create a new one." This criterion is unacceptable for hardware allocation, however, due to the fact that it does not recognize if the temporary is in use in a parallel sequence of operations. The allocator must decide whether to remove the temporary, or replace it with a temporary generated by the allocator in a manner compatible with physical hardware. The allocator makes this decision based on an estimation of the stability of the source variables for the operation.

A source is defined as unstable if the value it contains could change before the result of the operation is used. Such a change could cause the output of the path to be undefined or erroneous when it is used in subsequent operations. Cases where this might occur are:

* The source is shared between processes and therefore subject to random change.

* The source resides in the MDR of an array memory. It is impossible to determine in the general case if the value in the MDR will change. A worst case example can plausibly be developed here which would require N statement lookahead, a value trace of the MAR, and could still not be resolved because of possible parallel actions.

* The source is a previously allocated temporary register.

If either source variable satisfies one of the above criteria, a temporary of the proper size is allocated to hold the result of the operation, replacing the temporary generated by the ISPL compiler. The variable access pointer of the allocator temporary is then bound to the compiler temporary and used for subsequent accesses of the result value. An example of this sequence is the implementation of the operations described in statements 21 and 22 of the example statement table shown in Fig. 3.He). The path graph for these statements is the sequence on the left hand side of Fig. 3.4(b). The temporary labeled TREG1 in the path graph was generated to replace the compiler temporary JTFAAA used in the statement table. The reason for the generation of the temporary is that the variable CAR.FLOOR is

shared with the other process, MAIN, in the full elevator controller description.

If both sources are stable, however, the allocator simply removes the compiler temporary and creates an access pointer specifying the output of the path as the place to obtain the result value. This provides the allocator with the capability to generate combinatorial logic networks to implement complex actions in the functional description. An example of this action is shown in Fig. 3.5 for a complex combinational logic expression. The compiler has created the temporaries JTRAAC and JTRAAD. These are removed and the resulting path graph displays an OR-AND-OR combinatorial network.

3.^.6 Special cases

Several special cases are worth mentioning. Subfield accesses of variables and logical negation, which are operations in the ISPL statement table, are easily performed in hardware. These do not require the full processing just described, but are handled by simple manipulation of the access pointers. The ISPL control operations using data values as conditionals require only the source processing, so that access pointers may be created to allow the control allocator to access the data value. Finally, operations commonly available as register functions (INC/DEC, CLEAR, shift operations) are recognized and the required function is added to the register characteristics in the allocator symbol table.

3.4.7 Parallelism analysis

After each extended path has been allocated, a parallelism check is performed, both at the variable level and at the path level. All extended paths that are concurrently active with the extended path just allocated are recorded in the path parallelism stack. The new extended path is compared with the stack entries to determine path pairs active in parallel and to check for variables potentially in use at the same time by different actions in the ISP. Path pairs are recorded in the path parallelism table for future use in optimization routines.

Variable parallel usage errors are noted and a warning is issued to the user. Conditions which generate a warning are:

* Destination - destination conflicts: attempts to write a storage location concurrently using the results of different paths.

* Source - destination ambiguities: use of the same variable as a source and a destination in separate parallel actions. This could produce differing results depending on the relative order in which the parallel actions were executed.

* Array access conflicts:

  * Attempts to read and write the same array memory concurrently.

  * Attempts to use the same address path in two separate parallel actions.

No more than a warning can be generated, however, as the allocator cannot detect synchronization mechanisms specified by the user in the ISPL description which could resolve the ambiguity or conflict.

3.4.8 Optimization

After a process has been allocated as described above, the path graph contains a worst case allocation of the process in terms of operators and temporary storage. Optimization could be performed at this time using the parallelism information in the path parallelism table and the record of allocated paths in the path table. As an example optimization to demonstrate the feasibility of .such operations, reduction of temporary registers has been implemented using the criterion "combine the temporaries if they are compatible in size and no parallel or sequential usage conflicts exist."

After the optimization routines, the path table and path parallelism table are cleared and the allocator begins again with extended path allocation for all remaining processes. After all have been allocated, the second or process phase of the allocation is complete.

3.4.9 Final processing

After the process allocation is completed, a final examination of the path graph is made to place multiplexers where needed for switching data links. When this activity is completed, the allocation is complete.

3.5 Allocator Outputs

The primary allocator output is the path graph, which is used by the data base module, in conjunction with the allocator symbol table to map IC's onto the logical allocation. Because the succeeding routines have not yet been written, the allocator also produces a human readable version of the process table, the called procedure table, the operator table, the allocator symbol table, and the path graph.

4.0 Measuring Allocator Performance

The allocator described here is only beginning to be tested, so the results presented here are preliminary. Also, in the future, the actual hardware is to be allocated by the module data base, which is currently being designed. However, by performing a hand allocation of IC chips for path graph designs, we have obtained some preliminary results. Two designs have been done by the allocator. The first is part of an elevator controller and is described in Figs. 3.1, 3.2 and 3.4. The second is a reduced version (minus I/O) of the PDP-8/E. A non-optimal hand mapping of integrated circuits onto the allocator output logic diagram has been done, and estimates of chip count and cost were made.

The elevator controller was part of the design style experiment of (Thom77b), which provides a controlled measure of how well the allocator performs in relationship to human designers. The results are shown in Table 4.1. The cost figures for the allocator design were derived using Thomas' cost estimate:

$$C = \text{(total chip cost)} + (\$3 \text{ overhead/chip})$$

One can see that the cost of the automated design falls in the same order of magnitude as the cost of the designs produced for the design experiment, with no significant optimizations attempted by the allocator.

It is difficult to compare the automated PDP-8/E data path design with the original DEC (DEC72) design for three reasons. First, the ISPL description input to the allocator declares as registers some values the PDP-8/E uses but never stores explicitly in registers, such as the effective address. These show up as registers in the allocator's design. Also, the allocator designs distributed logic, and the DEC design was done in the central-accumulator design style (For a discussion of design styles, see (Thom77a)). Finally, the DEC design fortunately has assumed a boundary between the control and data-memory parts of the design, but the boundary is different from that imposed on the allocator by the ISPL description. Thus some tests, flags, and registers which must be declared explicitly in the ISPL description are part of the control in the DEC design. In spite of these differences, estimates of chip count indicate that the allocator produces a path graph which could use 31–38% more integrated circuit chips than the human designers used for the data paths and registers. Of course, these estimates were made using the same 1970 technology chip set the DEC designers had to deal with. The 30J excess hardware can be found in multiplexers which connect the registers, the extra registers declared in the ISPL description, and duplicated operators like increment and add. Much of this can be

attributed to the way in which the ISPL description had to be written, and some of these constraints will not be present in future ISPS descriptions. However, other chips can only be eliminated when optimization algorithms operate at some stage of the design process. It has not been shown whether this design operates at comparable speeds to the DEC processor. The complete block diagram of the allocator output can be found in Appendix B, along with the implementation information used to make the chip count estimates, and the PDP-8/E ISPL description.

One interesting point to be illustrated is the differences in the design seen even from the block diagram level. This is shown in Figure 4.1. There are two reasons for the differences. First, as stated previously, the design styles are different. Second, the multiplexing is used in different ways. In the DEC version, the operators are shared, and are even used to provide no-op paths from one register to another. In the CMU version, only registers are shared and use multiplexed inputs. The ISPL language is partially the source of this disparity. In ISPL, the user can repeatedly use register A as a destination from various sources. However, the expressions A+B and C+D do not imply (or discount) a single adder. Other differences in the design include the use of multiplexers for shifting in the DEC design, and use of true/complement 0/1 chips for creating complements. "Oring" of the MQ and AC registers in the DEC version is done within the multiplexing hardware. Constants are often created in one place and gated over already existant data paths to the registers. In the CMU version, these constants are multiplexed at the

register inputs. The lower (31J) figure of excess chips was obtained by using the CLEAR input on registers instead of a leg of the multiplexers to set some registers to zero.

One final difference is the treatment of the Link FF and Accumulator register as a single register in the CMU version. This is done because of the way the PDP-8/E ISPL description was written. Further analysis of this design is in progress and includes a human implementation of the control. Comparisons of the DEC and CMU data-path speeds will then be possible.

One more comparison has been possible. The Design Style Selector module is now running (Laws78) and makes cost and speed estimates for the input ISPS descriptions. It produced a worst-case cost estimate for the PDP-8/E data paths of $294.88, using a $3/chip cost estimate. Usng the same cost estimate on the 64 chip DEC design produces a cost of $192 and for the 84 chip CMU design $252. The cost differences illustrate the degree of optimizations performed by human and machine.

5.0 Conclusions and Discussion

The results described in this paper lead to the following conclusions:

    * The basic experimental allocator will function successfully as the base for an expanded allocator with optimization capabilities.

    * Specific module set information is not necessary to produce a

reasonable design.

* The size of the 1SPL description which can be handled is
  limited only by the amount of core memory available to the
  allocator.

* The allocator can detect user constructs in the ISPL
  description which will produce complex hardware or unreliable
  operation.

* The allocator will also be capable of significant optimization
  within the framework of the data structures described in this
  paper.

In addition, we have concluded that a large part of the complexity of
the allocator is due to:

* The ability to allocate multi-process ISPL descriptions.

* The lack of one-to-one correspondence between the compiled
  statement table operations and the actions necessary to
  implement the operations in hardware.

* The availability of the ISPL logical mapping facility.

* The production of temporaries by the ISPL compiler.

In addition, multi-process allocation would be considerably
easier from the control viewpoint if arbitration mechanisms for shared
variables and procedures were explicit in the ISPL descriptions.

Unfortunately, it is unreasonable to expect these good results to
hold for more complex designs, as the larger size allows greater
freedom to improve the design through optimization. Optimization
research is, therefore, a significant part of the design automation

effort.

The conversion to ISPS is expected to alleviate the complex-mapping problems, the compiler temporaries will no longer exist, and the technical file will not be needed.

6.0 References

(Barb75) Barbacci, M.R. and Siewiorek, D.P., "The CMU RT-CAD System: An Innovative Approach to Computer Aided Design," AF1PS Conference Proceedings, vol. 45, pp. 643-655, 1976.

(Barb76) Barbacci, M.R., "The Symbolic Manipulation of Computer Descriptions: ISPL Compiler and Simulator," Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1976.

(Barb77a) Barbacci, M.R., et. al, "Architecture Research Facility: ISP Descriptions, Simulation and Data Collection," Proceedings, 1977 National Computer Conference, Dallas, Texas, June 1977.

(Barb77b) Barbacci, M.R., Barnes, G.E., Cattell, R.G. and Siewiorek, D.P., "The ISPS Computer Description Language," technical report, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1977.

(DEC72)   Digital Equipment Corporation, "PDP-8/E Maintenance Manual, vol. 1, no. DEC-8E-HR1B-D, 1972.

(Hafe78) Hafer, L.J. and Parker, A.C., "Register-Transfer Level Automatic Digital Design: The Allocation Process," Proceedings of the 15th Design Automation Conference, Las Vegas, Nevada, June 1978.

(Laws78) Lawson, Gregory L., "Design Style Selector, an Automated Computer Program Implementation," M.S. Project Report, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, PA, August 1978.

(Leiv77) Leive, G., "The Binding of Modules to Abstract Digital Hardware Descriptions," Thesis Proposal, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, PA, 1977.

(Park78) Parker, A.C. and Hafer, L.J., "The Application of a

Hardware Descriptive Language for Design Automation,^M Proceedings of the Third Jerusalem Conference on Information Technology, August 1978.

(Rege74)   Rege, S.L., "Designing Variable Data Format Modules With Cost-Performance Tradeoffs,[1]* Ph.D. Thesis, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, PA, 1974.

(Snow78a) Snow, E.A., Siewiorek, D.P. and Thomas, D.E., "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations and Design Tradeoffs,[11] Proceedings of the 15th Design Automation Conference, Las Vegas, Nevada, June 1978.

(Snow78b) Snow, E.A., "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. dissertation, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, PA, 1978.

(Thom77a) Thomas, D.E., "The Design and Analysis of an Automated Design Style Selector," Ph.D. dissertation, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, PA, 1977.

(Thom77b) Thomas, D.E* and Siewiorek, D.P., "Measuring Designer Performance to Verify Design Automated Systems," Design Automation Conference Proceedings, vol. 14, pp. 411-418, 1977.

Appendix A: Mappings

The ISPL language contains a generalized logical mapping facility allowing the user to declare a logical component in terms of a previously declared physical or logical component. The only restriction imposed by the ISPL compiler is that the physical size (total bits) of each side of the declaration be equivalent. For mappings where the mapped variable is a register, this restriction is sufficient. When the mapped variable is defined as an array structure (memory), however, additional restrictions must be imposed to insure that a reasonable hardware implementation of the mapping can be created. Consider the following example mapping, which will be used to define the terms used in the equation defining the mapping constraints.

Al [0:1, 8:9, 14:15] <0:7 > ;   1 declaration of base memory

AA2 [5, 9, 12] <0:15> : = Al [0:1, 8:9, 14:15] <0:7 > ;

                              ! mapping declaration

Define the following terms:

main declaration	::=	declaration of Al
mapping primary	::=	left half of the mapping declaration (AA2)
mapping secondary	::=	right half of the mapping declaration (Al) (may in general be all of the main declaration or a subset of the addressing space defined by the main declaration)
$bitcnt_p$	::=	bit size of primary word - 16 bits in this example
$bitcnt_s$	::=	bit size of secondary word - 8 bits in this example
d	::=	$lg_2 (bitcnt_p/bitcnt_s) - 1$ in this example
$b0_p$	::=	lowest address of primary - 5 in this example

$b0_s$ ::=   lowest address of secondary - 0 in this example

adr_p ::=   address from primary address space

adr_s ::=   address from secondary address space

With these definitions, any mapping satisfying the equation

$$(adr_p + 5) + d = adr_s \quad \text{where } 6 = (b\,0_s + - d) - b\,Q\,T_p = \text{constant}$$

$$\text{and } + \text{ is the logical shift operation}$$

can be implemented with at most an addition and wired shift in the address path , a multipiexor-demultiplexor pair to provide the necessary gating in and out of the MDR of the main definition, and, for the case where $bitcnt_p > bitcnt_s$, a MDR for the primary to assemble the primary word while the necessary number of memory accesses are made in the main memory.

The path graph representation of the example mapping as allocated by the data-memory allocator is attached as fig. A-1•

APPENDIX B

CMU PDP-8 Design

The next 4 pages contain the automated PDP-8 design. All connections
contain the number and position of bits specified. Output wires not
attached to any other blocks are used by the control to test for
certain conditions.   MUX connections not specified indicate that
register bits in those positions are to remain unchanged.

Because DEC includes the LINK logic in the control module rather than
the data path module, we are able to reduce the complexity of the
accumulator input multiplexing. The link multiplexing belongs in the
control chip count.

Two chip counts have been done.   The first was a "worst-case"
allocation, which was done by fitting chips to the path graph just as
it stands. This allocation used 90 chips. The only alteration was to
exclude the link logic. The second, or "intelligent" allocation also
removed some legs on muxes, assuming registers could clear themselves.

A. DESCRIPTION PROCESSING
B. DESIGN PROCESSING
C. MODULE DATA BASE SYSTEM
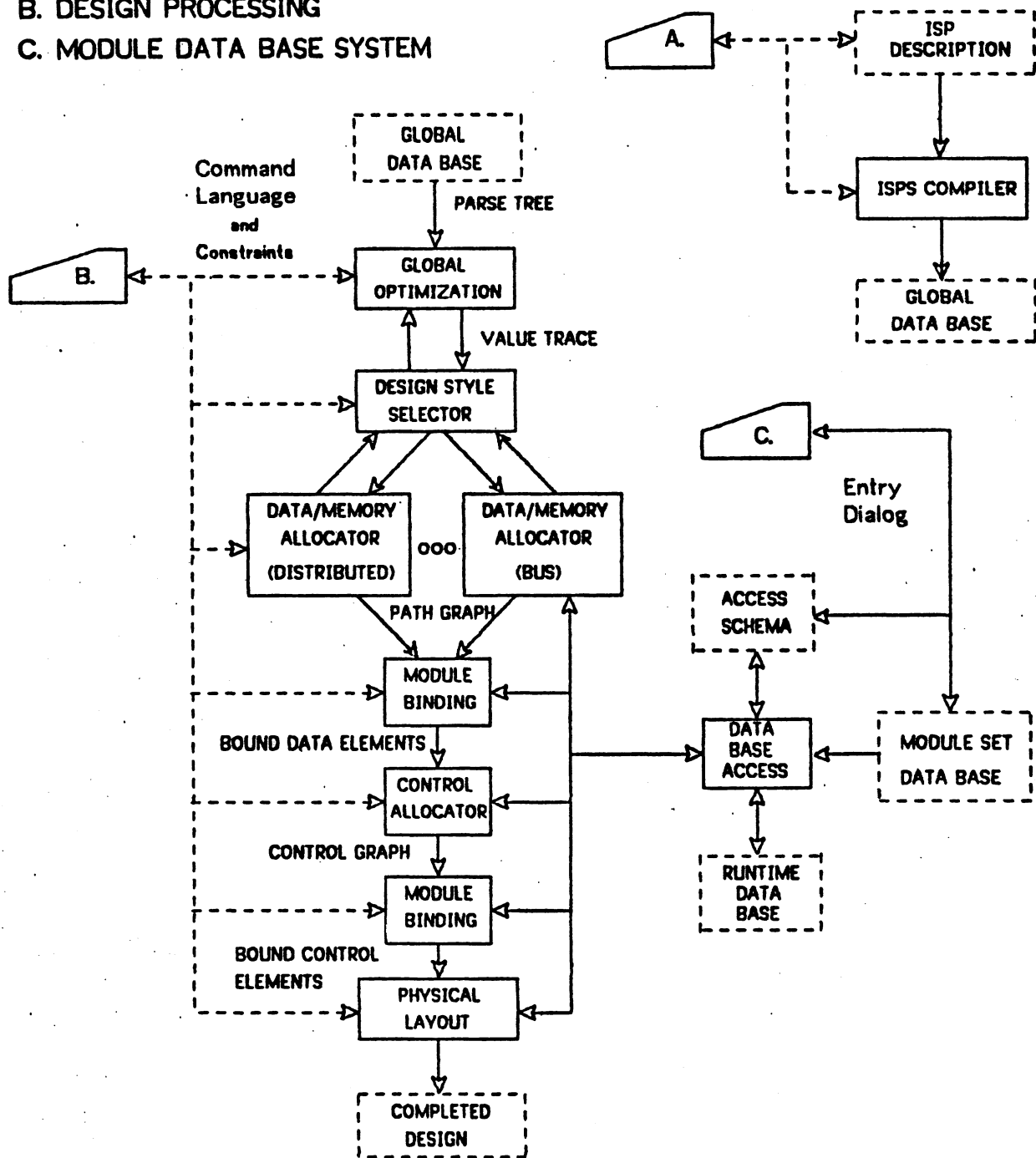
FIG. 2.1   RT-CAD SYSTEM OVERVIEW

```
! Declarations for Simple.elevator.control !

        car.floor<2:0> ;            !floor car is on
        car.call[15:0]<> ;          !8 floors, 1 bit each u/d
macro top.floor := 7$


    ! Declarations for Look.for.calls !

        scan.floor<2:0> ;           !scan inputs from 8 floors
        up.call<> ;                 !input used with scan
        down.call<> ;               !input used with scan
        button<> ;                  !input used with scan

    ! end declarations for Look.for.calls !

Look.for.calls := (

        !floorscan controls multiplexors whose inputs
        !are the call buttons.  The outputs of the multiplexors
        !are upcall, downcall, and button

    scan.floor ← 0 next
    Next.floor := (
        !upcall at scanned floor ?
        (IF up.call => car.call[1@scan.floor] ← 1) next
        !down call at scanned floor ?
        (IF down.call => car.call[0@scan.floor] ← 1) next
        !button in car pushed for floor ?
        (IF button => (
                !decide wether up or down call
                DECODE car.floor GTR scan.floor =>
                car.call[1@scan.floor] ← 1;
                car.call[0@scan.floor] ← 1
                )) next
    scan.floor ← scan.floor + 1 next
    (IF scan.floor LEQ top.floor => next.floor))
    !leave if all floors scanned; otherwise look at next floor
    next look.for.calls
    );
```

Fig. 3.1(a) ISPL Source Code for LOOK.FOR.CALLS

INDEX	TYPE	FLAGS	DEF	BLK	LBL	BCNT	UCNT	PNAtiE UOROSjBITStwME(POSITIO
3	2	18888888	8	8	8	1	1	•BUTTON'
5	1	18888888	8	8	8	1	28	•CAR.CA*C8<17):17(8)]
6	2	18888888	8	8	8	3	1	•CAR.FL'<B(2):2(B)>
15	2	18888888	8	8	8	1	1	'OOUN.C*
22	4	18881188	8	8	3	8	8	•LOOK.F*
30	4	18888188	8	8	6	8	8	•NEXT.F'
33	2	18888888	8	8	8	3	1	'SCAN.F'<8(2):2(8)>
37	2	18888888	8	8	8	1	1	•UP.CAL'
41	3	18888881	B	8	8	1	8	B
42	3	18B88B81	8	8	8	1	8	1
44	3	18888881	8	8	8	3	8	7
47	18	18888881	8	8	8	1	8	'XTFAAA*
52	7	18888881	8	8	8	4	8	•XTRAAC

Fig. 3.Kb) ISPL Compiler Symbol Table for LOOK.FOR.CALLS

INDEX	LABEL	FLAG	OPC00E	DEST	. SOURCE1	SOURCE2	MERGE	PATHS
5		0	'CLEAR	"SCAN.F*				
				(33)				
6	'NEXT.F'	1	'SMERGE*					
	(3B)							
7		8	'ISP '				4	
10		0	'IF '		'UP.CAL*		13	13,11
					(37)			
11		0	'CONC	"XTRAAC	I'SCAN.F'			
				(52)(	42)(	33)		
12		0	'WRITE	" CAR. CA"	XTRAAC	1		
				(5)(	52)(	42)		
13		8	'SMERGE'					
14		0	'IF '		'DOUN.C		17	17,15
					(15)			
15		0	'CONC	"XTRAAC	0'SCAN.F'			
				(52) (	41) <	33)		
16		8	'WRITE	"CAR.CA"	XTRAAC	1		
				(5)(	52) (	42)		
17		0	'ST1ERGE'					
20		0	'IF '		'BUTTON'		31	31,21
					(3)			
21		0	'GTR	"XTFAAA"	CAR.FL"	SCAN.F'		
				(47)(	G)(	33)		
22		0	'BRANCH*		'XTFAAA'		30	23,2G
					(47)			
23		0	'CONC	"XTRAAC	I'SCAN.F'			
				(52)(	42)(	33)		
24	'	0	'WRITE	"CAR.CA"	XTRAAC	1		
				(5)(	52)(	42)		
25		0	'JOIN '				30	
26		0	'CONC	"XTRAAC	0'SCAN.F'			
				(52)(	41)(	33)		
27		0	'WRITE	"CAR.CA"	XTRAAC	1		
				(5)(	52)(	42)		
30		0	'StiERGE'					
31		0	'SMERGE'					
32		0	'INCR	" SCAN. F"	SCAN. F'			
				(33) (	33)			
33		0	'LEO.	"XTFAAA"	SCAN,F'	7		
				{ 47)(	33)(	44)		
34		0	'IF '		'XTFAAA'		36	36,35
					(47)			
35		4	'JOIN '		'NEXT.F*		6	
					(30)			
.36		8	'SMERGE'					
37		0	'NOOP '		'NEXT.F'		6	
					(30)			

Fig. 3.He) ISPL Compiler Statement Table for LOOK.FOR.CALLS

! technical file for elevator control

PROCESS t   MAIN , LOOK.FOR.CALLS ;

INPUT :

! inputs for LOOK.FOR.CALLS

```
        up.call - level down / elec ttl ,
        down.call • level down / elec ttl ,
        button • level down / elec ttl ;
```

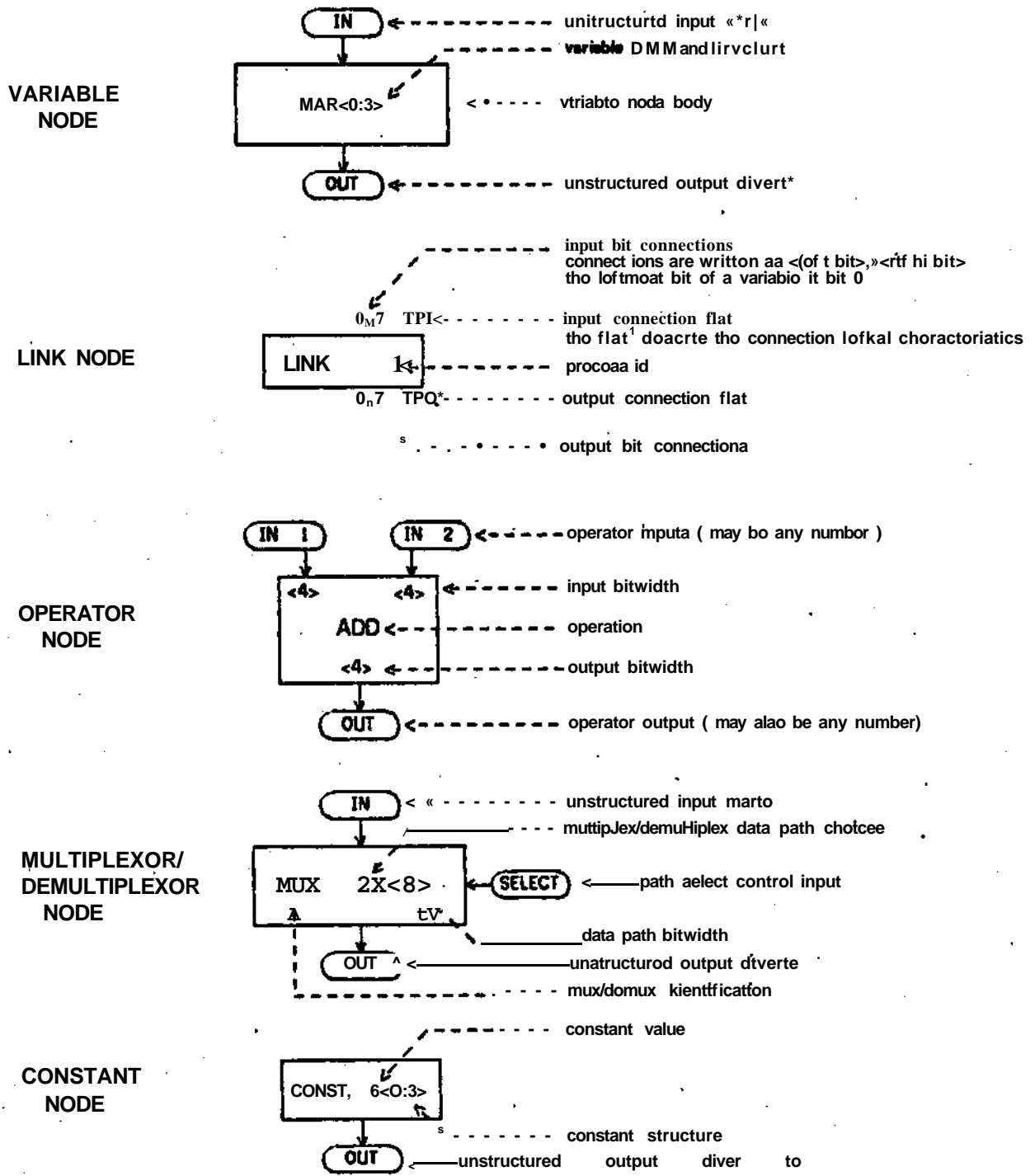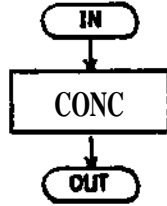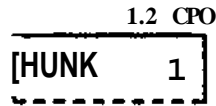Fig. 3.2 Technical File for LOOK.FOR.CALLS

VARIABLE NODE

IN ◄ ─ ─ ─ ─ ─ ─ ─ ─ unitructurtd input «*r|«
─ ─ ─ ─ ─ ─ ─ variable DMM and lirvclurt

MAR<0:3> ◄ • ─ ─ ─ ─ vtriabto noda body

OUT ◄ ─ ─ ─ ─ ─ ─ ─ ─ unstructured output divert*

─ ─ ─ ─ ─ ─ ─ ─ input bit connections
connect ions are writton aa <(of t bit>,»<rtf hi bit>
tho loftmoat bit of a variabio it bit 0

0_M7 TPI< ─ ─ ─ ─ ─ ─ input connection flat
tho flat[1] doacrte tho connection lofkal choractoriatics

LINK NODE

LINK k ─ ─ ─ ─ ─ ─ ─ procoaa id

0_n7 TPQ* ─ ─ ─ ─ ─ ─ output connection flat

s ─ ─ ─ • ─ ─ ─ • output bit connectiona

OPERATOR NODE

IN 1   IN 2 ◄ ─ ─ ─ ─ operator mputa (may bo any numbor)

<4>   <4> ◄ ─ ─ ─ ─ ─ input bitwidth

ADD ◄ ─ ─ ─ ─ ─ ─ ─ operation

<4> ◄ ─ ─ ─ ─ ─ ─ ─ output bitwidth

OUT ◄ ─ ─ ─ ─ ─ ─ ─ operator output (may alao be any number)

MULTIPLEXOR/ DEMULTIPLEXOR NODE

IN < « ─ ─ ─ ─ ─ ─ ─ unstructured input marto
─ ─ ─ muttipJex/demuHiplex data path chotcee

MUX 2X<8> ◄ SELECT ◄ ─ path aelect control input

A   tV

─ data path bitwidth

OUT ^ ◄ ─ unatructurod output dtverte

─ ─ ─ ─ ─ ─ ─ ─ ─ mux/domux kienttfication

CONSTANT NODE

─ ─ ─ ─ ─ ─ ─ constant value

CONST, 6<O:3>

s ─ ─ ─ ─ ─ ─ ─ constant structure

OUT ─ unstructured   output   diver   to

FIG. 3.3(a)   **BASIC PATH GRAPH GROUPS**

CONCATENATION
NODE

HALF LINK
NODE

1.2 CPO

[HUNK   1]

uaad lo apoctfy on* aida of a connection
for laiar *v*9* by tha control allocator
codaa ara idafttical to full Knka

CONNECTION FLAG CODE

C :   complamantad
T: truo
P: paraflal
S .   aarial
I :   input
0 .'   output

FIG. 3.3(b)   BASIC PATH GRAPH GROUPS
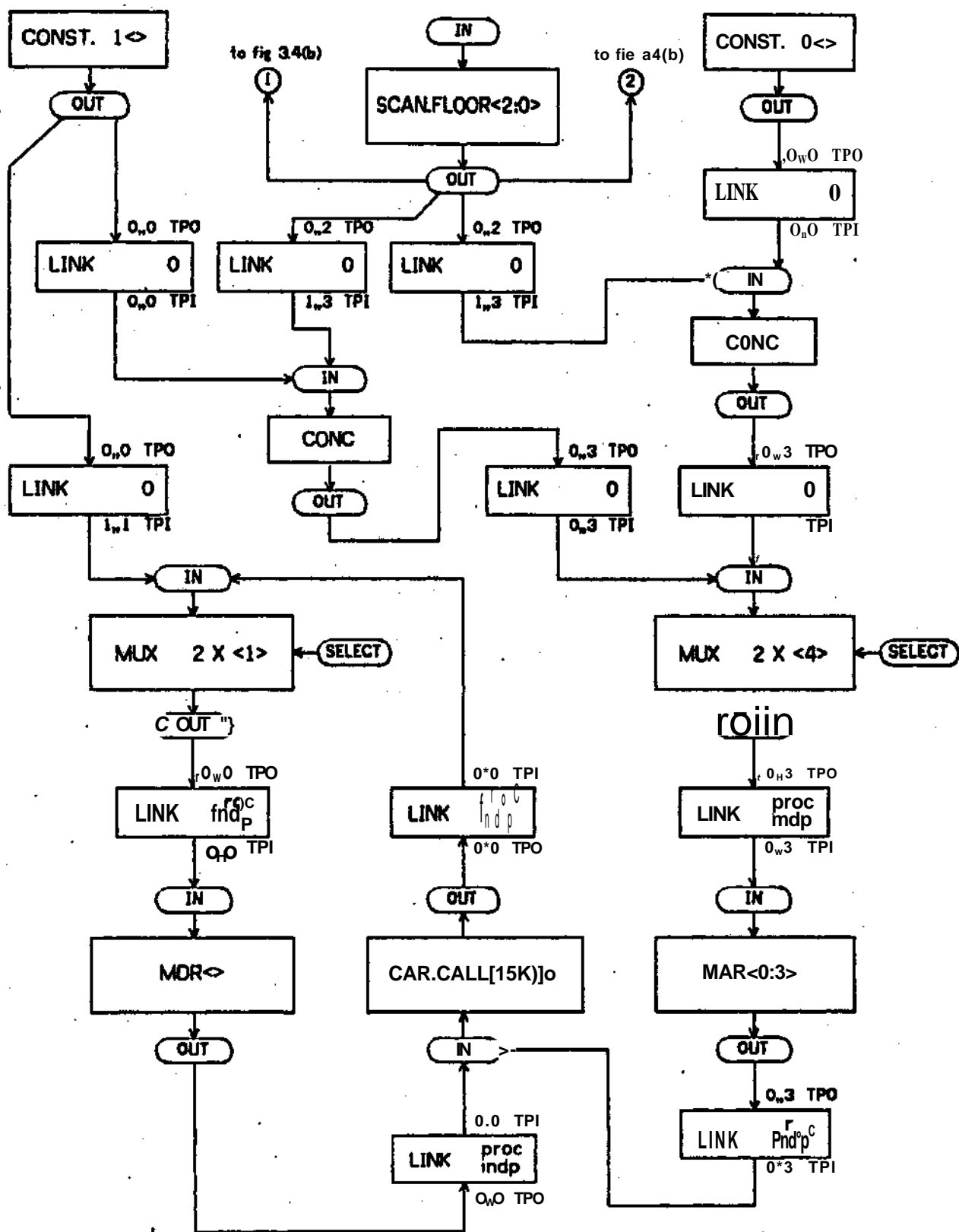
FIG. 3.4(a)   PATH GRAPH FOR LOOK.FOR.CALLS
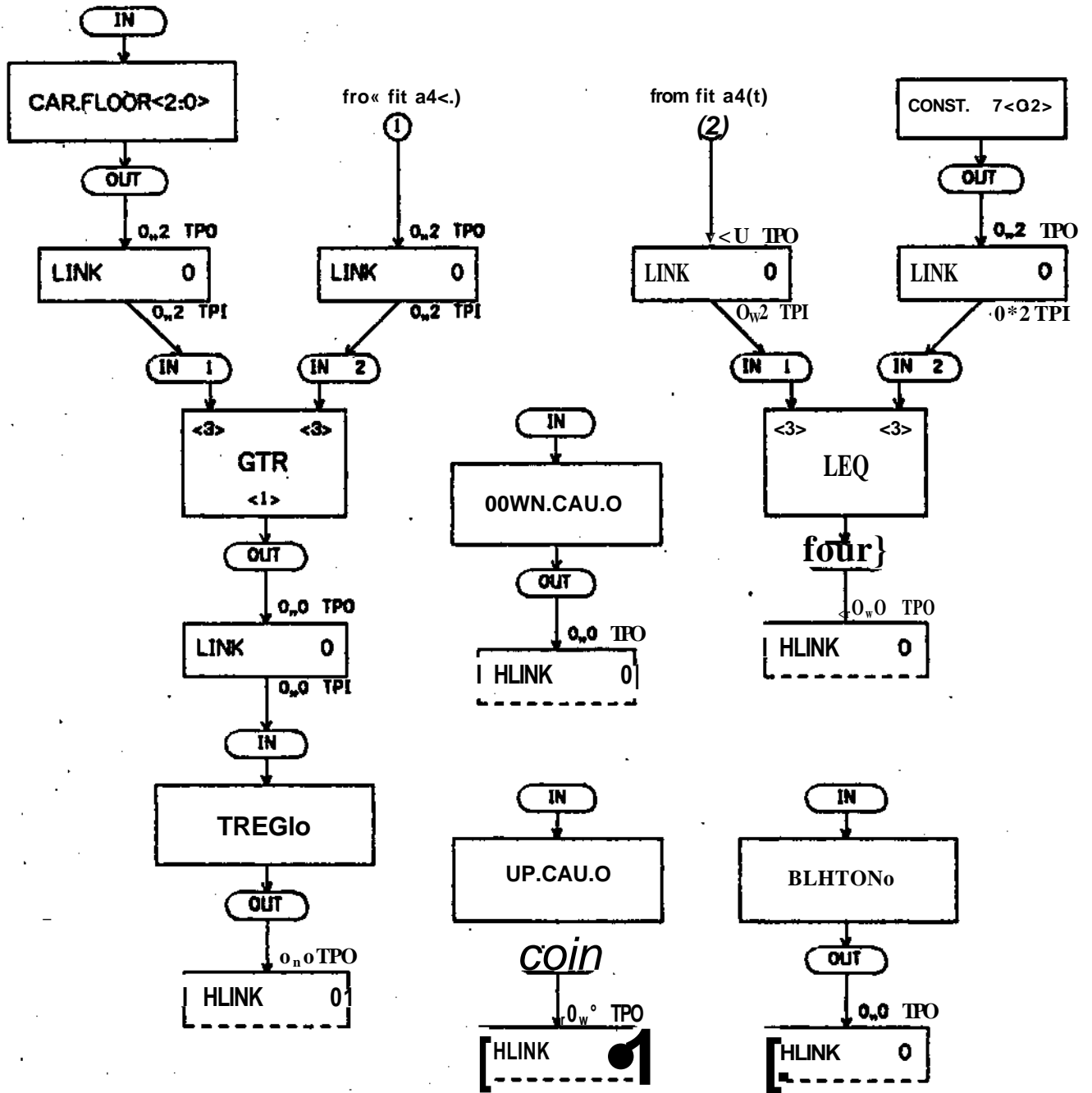
FIG. 3.4(b)   PATH GRAPH FOR L00K.F0RCALL5

```
      big.reg<0:63> ;                        ! big register
      Iittle.reg<0:15> :« big.reg<Si23> ;    ! tittle register
      a<7:0> ;                               ! extra registers
      b<7:8> 5
```

blah :- (a «- b AND NQT(b OR a) OR N0T(big.reg<lB$25> OR little.reg))

Fig* 3.5(a) ISPL Source Text

INDEX	TYPE	FLAGS	DEF	BLK	LBL	BCNT	WCNT	PNAME	WORDS;BITS:NAME(POSIT101
1	2	1BBB88BB	8	0	B	10	1	•A	'<B(7):7(B>>
2	2	18888888	B	0	8	10	1	•B	'<B(7):7(B)>
3	2	18888B88	8	0	B	100	1	•BIG.RE'<77(77):8(8)>
4	2	18100000	3	8	8	20	1	*BIG.RE^t<27(17):18(8)>
5	4	10001108	8	8	44	0	0	•BLAH '
12	2	18B188BB	4	B	0	20	1	•LITTLE'<17(17):0<8)>
48	5	100008B1	0	0	0	28	0	65,,46
43	7	18888881	0	8	0	10	0	•XTRAAC*
44	7	18888881	0	B	e	20	0	•XTRAAO'

Fig. 3.5(b) ISPL Compiler Symbol Table for BLAH


```
44   'BLAH  '   1    'SMERGE'
     (     5)
45              8    •ISP   '                                   18
46              B    •CR    ''XTRAAC •B      ' 'A      '
                            {   43)(      2)(      1)
47              8    •NOT   ' •XTRAAC •XTRAAC
                            (     43)(     43)
58              8    •ANO   ' •XTRAAC •B      ' •XTRAAC
                            (     43)(      2)(     43)
51              0    •RBYTE '•XTRAAO'•BIG.RE'  65,,46
                            (     44)(      3)(     48)
52              8    'CR    '•XTRAAO'•XTRAAO' 'LITTLE'
                            (     44)(     44)(     12)
53              B    •NOT   '•XTRAAD* •XTRAAD'
                            (     44)(     44)
54              0    •CR    ' •A      '•XTRAAC 'XTRAAO'
                            (      1)(     43)(     44)
55              8    •RETURN'          'BLAH  '                 44
                                       (     5)
```


Fig. 3.5(c) ISPL Compiler Statement Table for BLAH

PATH GRAPH FOR BLAH
FIG. 3.5(d)
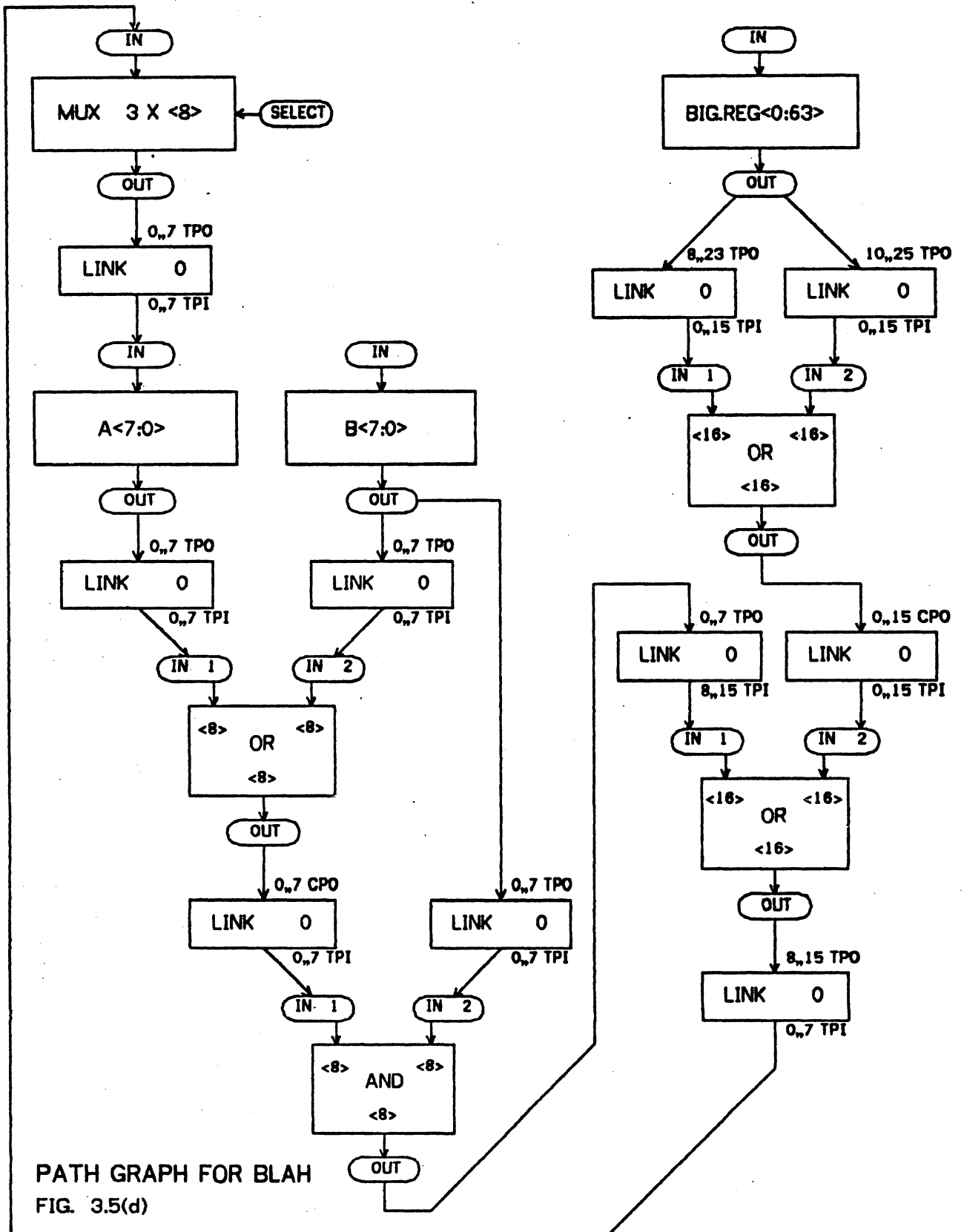
Cost for data part of process LOOK.FOR.CALLS as
implemented for Thomas' design experiment:

     Designer 10 : $32.49
     Designer 5 : $71.45

Cost for data part of process LOOK.FOR.CALLS as
implemented from allocator path graph ✭ :

               $47.63

✭ The same subset of the TTL chip family used in the
design experiment was used to implement the allocator
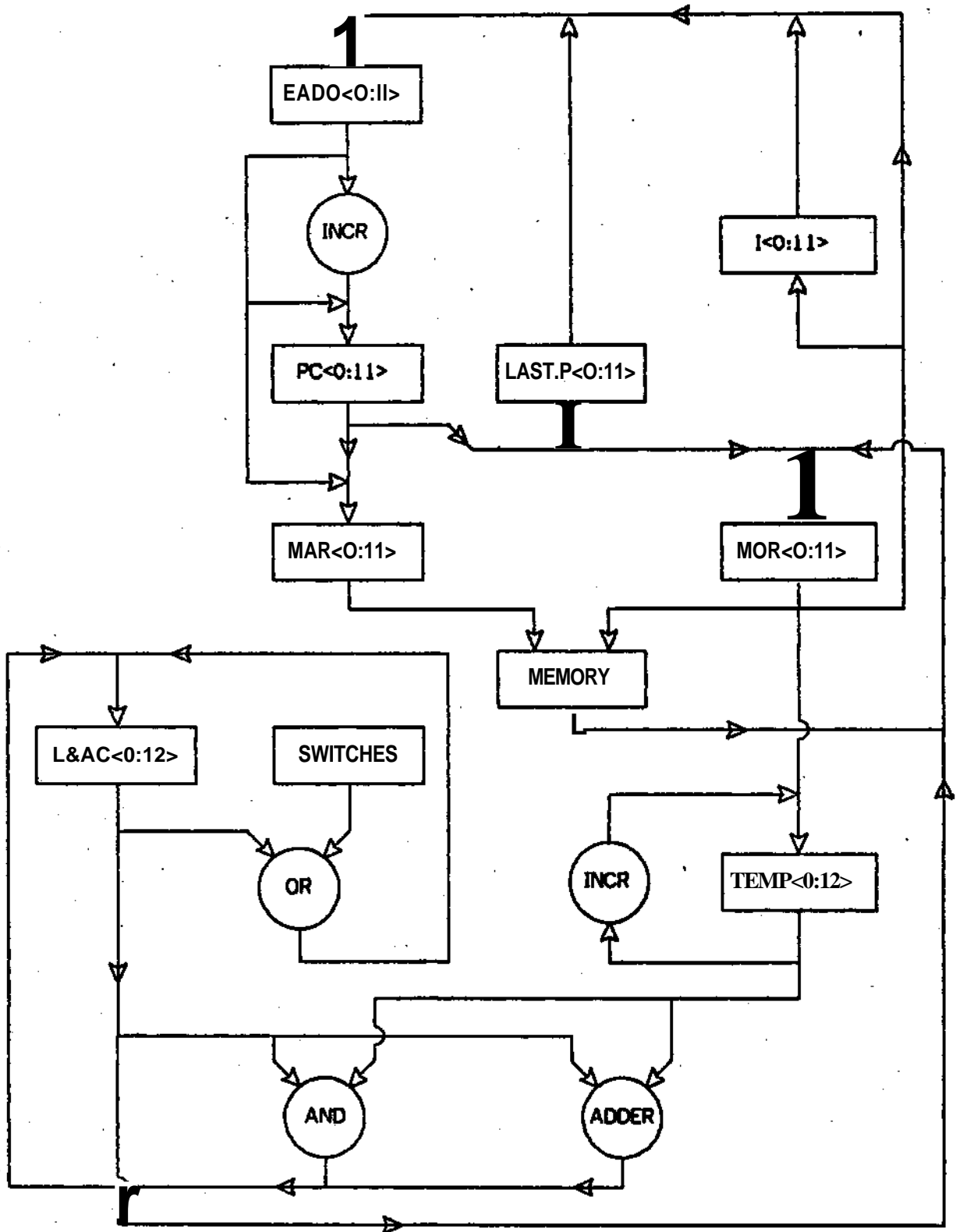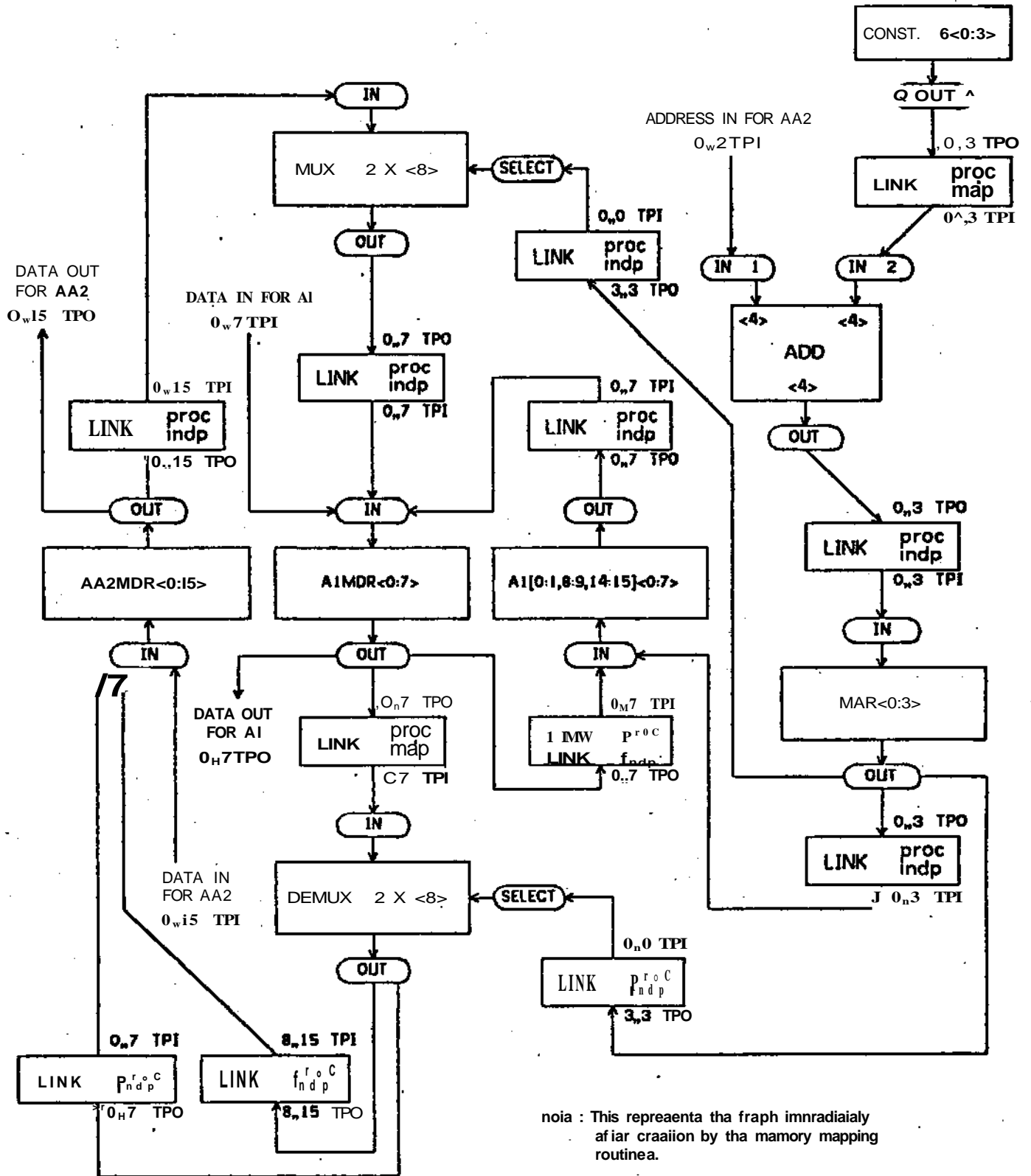path graph

                    Table 4.1

FIG. 4.1   BLOCK DIAGRAM OF CMU PDP8/E DESIGN

FIG. A1   PATH GRAPH FOR MEMORY MAPPING EXAMPLE

CMU - PDP-8 Chip Count
Intelligent Allocation

The following chip count was taken from the part of the automated design corresponding to the DEC PDP-8 M8300 "major resistors."[11] The chips were allocated by hand using an intelligent allocation. For example, if the allocator specified a 7 to 1 13 bit MUX, and the MUX could be decomposed into a simpler structure, it was*

Part #	(TI TIL Equiv.)	Quantity	Part of Path Graph Implemented
74153		6	MUX 4 x <12>*
8271	74194	3	AC<0:11>
7474 (%)		1	TREG3 <0>
7402		1	0R<12>
7404		1	
74153		7*	MUX 3 x <13>
8271	74194	3	TREG3 <1:12>
8271	74194	3	LAST.P<0:11>
7483		3	INCR<13>
7404		2	
7420		1	EQL<12>
7430		1	
7483		4	ADD<14>
7400		3	AND<12>
7404		2	
74153		6	MUX 4 x <12>
8271	74194	3	TREG1 <0:11>
74153		6	MUX 3 x <12>
8271	74194	3	PC<0:11>
74153		6	MUX 3 x <12>
8271	74194	3	TREG2 <0:11>
7483		3	INCR<13>
8271	74194	3	EADD<0:11>
8266	74157	3	MUX 2 x <12>**
7404		2	
7400		1	EQL<9>
7430		1	
	Total	84	

*   Reduced from a 7 x <13>MUX due to exclusion of link logic and allowing
     AC to be able to clear itself.

**   Reduced from a 3 x <12>MUX due to clear on register eliminating († leg
     of this MUX.

DEC - PDP-8/E Chip Count
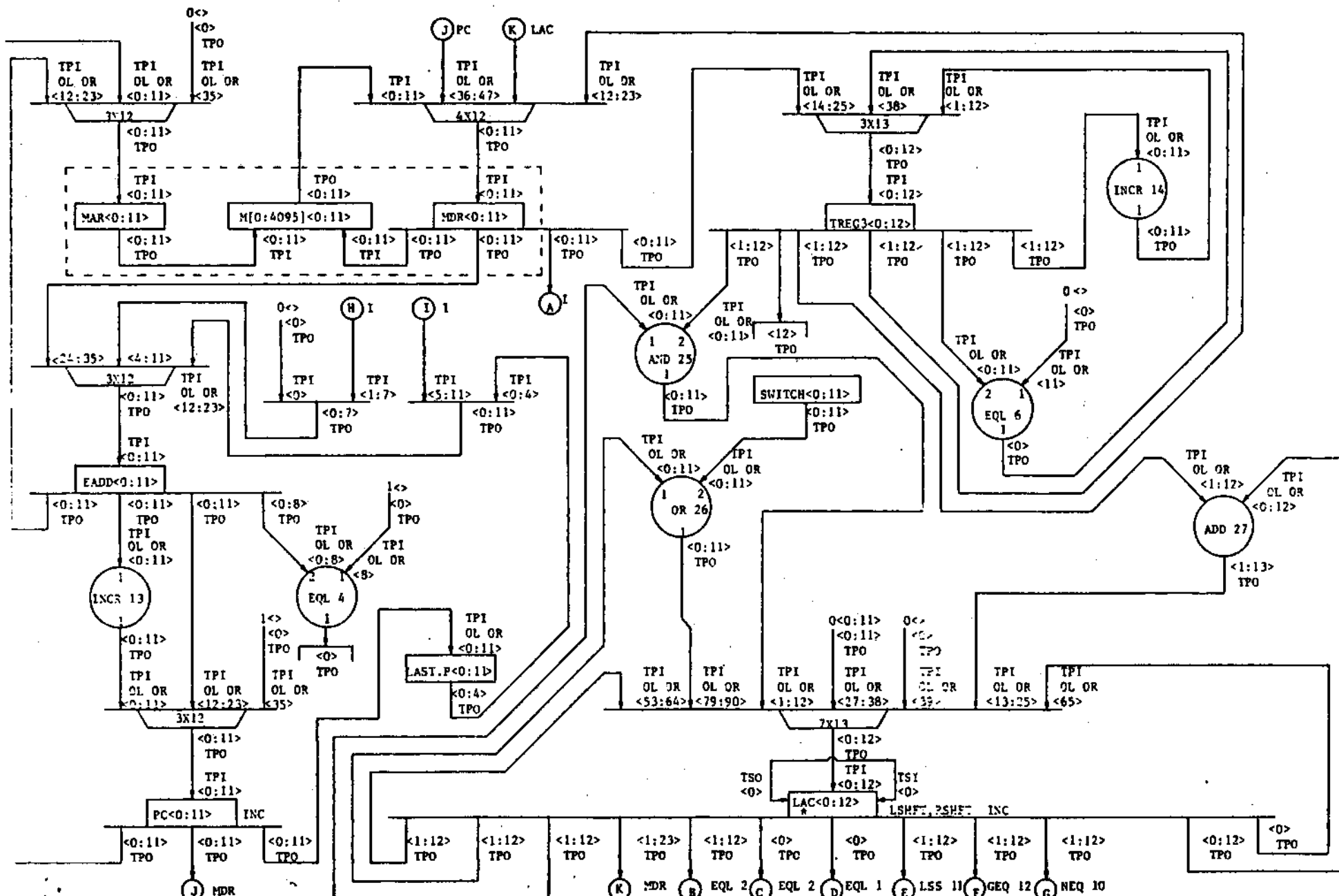
The following chip count was taken from M8300 "major registers."

Part #	/TI ^TTL equiv. /	Quantity	Function
7400		3	Quad 2 input NAND
7402		1	Quad 2 input NOR
74H04		2	Hex inverter
7420		1	Dual 4 input NAND
7430		1	8 input NAND
74H87		3	4 bit true/complement
7483		3	4 bit binary full adders
84151		12	1 of 8 MUX
74153		6	dual 4 to 1 MUX
8271	74194	15	4 bit universal shift reg.
8266	74157	8	Quad 2 to 1 MUX
8235	74H87	3	4 bit true/complement
8881	7401	6	Quad 2 input NAND

Total 64 iiitegrated circuit chips

Figure Bl(a) The Allocated PDP8.