USING A COMPUTER TO DESIGN
COMPUTER INSTRUCTION SETS

by

Frederick Marion Haney

Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pennsylvania
May 20, 1968

Submitted to the Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects
Agency of the Office of the Secretary of Defense (SD-146)
and is monitored by the Air Force Office of Scientific
Research. Distribution of this document is unlimited.
ABSTRACT

The purpose of the research described in this thesis is to develop a method for using a computer to design computer instruction sets and to experiment with instruction set design strategies. One result is GIS (Generalized Instruction Set) -- a model of the concept instruction set. The primary result -- ISDS (Instruction Set Design System) -- is a hierarchy of IPL-V subroutines that perform the bookkeeping, specification, and analysis required to construct a GIS representation of an instruction set in the memory of a computer. Also described is a heuristic program, written in ISDS, that determines its own strategy for designing an optimal instruction set for a given programming environment. Evaluation is provided by an instruction mix and a scoring polynomial whose coefficients specify the relative cost and value of addressing features. The program can design instruction sets with almost any addressing feature used in existing computers, but the instruction sets are restricted primarily to a single instruction format. Twelve examples are presented to show how various instruction sets can be obtained by altering the input, altering evaluation parameters, and extending the system. The examples include design strategies for the instruction formats of the IAS computer, Whirlwind I, and the Univac 1108.
ACKNOWLEDGEMENTS

I want to thank five people for their help in the preparation of this thesis. I am very grateful to Professor Gordon Bell, who suggested the problem and, as my thesis advisor, contributed much of his time and many valuable suggestions. I am also grateful to Professor Allen Newell, who reviewed my work at several stages and made valuable suggestions as to the direction it should take, and to Professors Alan Perlis and Richard Van Horn, who suggested significant improvements in the written thesis. Finally, a very warm "thank you" to my wife, Barbara, for typing two drafts of the thesis.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TITLE PAGE</td>
<td>i</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>iv</td>
</tr>
<tr>
<td>ILLUSTRATIONS</td>
<td>vi</td>
</tr>
<tr>
<td>INTRODUCTION,</td>
<td>1</td>
</tr>
<tr>
<td>- Statement of the Problem and Primary Results</td>
<td></td>
</tr>
<tr>
<td>- Architectural Design of Computers</td>
<td></td>
</tr>
<tr>
<td>- The Computer as a Design Tool</td>
<td></td>
</tr>
<tr>
<td>CHAPTER I. INSTRUCTIONS FOR COMPUTERS</td>
<td>9</td>
</tr>
<tr>
<td>- Definition of the Problem</td>
<td></td>
</tr>
<tr>
<td>- Classification of Instructions</td>
<td></td>
</tr>
<tr>
<td>CHAPTER II. SELECTION AND EVALUATION OF INSTRUCTION SETS</td>
<td>25</td>
</tr>
<tr>
<td>- Evaluation of Computer Performance</td>
<td></td>
</tr>
<tr>
<td>- Selection of an Instruction Set</td>
<td></td>
</tr>
<tr>
<td>CHAPTER III. USING A COMPUTER TO DESIGN INSTRUCTION SETS</td>
<td>53</td>
</tr>
<tr>
<td>- Theory of Design</td>
<td></td>
</tr>
<tr>
<td>- Requirements of a Computer-based Design System</td>
<td></td>
</tr>
<tr>
<td>- Needs Analysis</td>
<td></td>
</tr>
<tr>
<td>- Activity Analysis</td>
<td></td>
</tr>
<tr>
<td>- Outline of a Solution</td>
<td></td>
</tr>
<tr>
<td>CHAPTER IV. GENERALIZED INSTRUCTION SET</td>
<td>73</td>
</tr>
<tr>
<td>- Developing a Model for Instruction Sets</td>
<td></td>
</tr>
<tr>
<td>- Description of GIS</td>
<td></td>
</tr>
<tr>
<td>- The Use of GIS to Represent Instructions</td>
<td></td>
</tr>
<tr>
<td>CHAPTER V. A SYSTEM FOR DESIGNING INSTRUCTION SETS</td>
<td>85</td>
</tr>
<tr>
<td>- Introduction</td>
<td></td>
</tr>
<tr>
<td>- A Data Structure for GIS</td>
<td></td>
</tr>
<tr>
<td>- System Routines</td>
<td></td>
</tr>
<tr>
<td>CHAPTER VI. USING ISDS TO DESIGN INSTRUCTION SETS</td>
<td>115</td>
</tr>
<tr>
<td>- The Basic Strategy -- T5</td>
<td></td>
</tr>
<tr>
<td>- Adding Features to the Basic Instruction Set</td>
<td></td>
</tr>
<tr>
<td>- Evaluation Routine -- T51</td>
<td></td>
</tr>
<tr>
<td>- Examples</td>
<td></td>
</tr>
<tr>
<td>CHAPTER VII. CONCLUSION</td>
<td>223</td>
</tr>
<tr>
<td>- Evaluation of ISDS</td>
<td></td>
</tr>
<tr>
<td>- Implications of the Research</td>
<td></td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS (continued)

Future Work on ISDS
The Future of Design Systems

BIBLIOGRAPHY .............................................. 236
# ILLUSTRATIONS

<table>
<thead>
<tr>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relationships Between the Design Variables</td>
<td>39</td>
</tr>
<tr>
<td>Asimow’s Model of the Design Process</td>
<td>57</td>
</tr>
<tr>
<td>Address Interpretation Algorithm</td>
<td>79</td>
</tr>
<tr>
<td>Form-Variable</td>
<td>90</td>
</tr>
<tr>
<td>Attribute and Value Symbols for GIS</td>
<td>92</td>
</tr>
<tr>
<td>Other Symbols for GIS</td>
<td>93</td>
</tr>
<tr>
<td>Hierarchy of Routines and Data in ISDS</td>
<td>95</td>
</tr>
<tr>
<td>Data Structures for Design Procedures</td>
<td>104</td>
</tr>
<tr>
<td>Table of Reserved Words</td>
<td>110</td>
</tr>
<tr>
<td>Optimization in ISDS</td>
<td>128</td>
</tr>
<tr>
<td>Whirlwind I</td>
<td>132</td>
</tr>
<tr>
<td>Computer Output, Example 1</td>
<td>135</td>
</tr>
<tr>
<td>Computer Output, Example 2</td>
<td>142</td>
</tr>
<tr>
<td>Computer Output, Example 3</td>
<td>147</td>
</tr>
<tr>
<td>IAS Computer</td>
<td>152</td>
</tr>
<tr>
<td>Computer Output, Example 4</td>
<td>155</td>
</tr>
<tr>
<td>Computer Output, Example 5</td>
<td>161</td>
</tr>
<tr>
<td>Computer Output, Example 6</td>
<td>165</td>
</tr>
<tr>
<td>Computer Output, Example 7</td>
<td>169</td>
</tr>
<tr>
<td>1108 Instruction Format</td>
<td>179</td>
</tr>
<tr>
<td>Computer Output, Example 8</td>
<td>181</td>
</tr>
<tr>
<td>Computer Output, Example 9</td>
<td>187</td>
</tr>
<tr>
<td>Computer Output, Example 10</td>
<td>196</td>
</tr>
</tbody>
</table>
ILLUSTRATIONS (continued)

<table>
<thead>
<tr>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Output, Example 11</td>
<td>205</td>
</tr>
<tr>
<td>Computer Output, Example 12</td>
<td>215</td>
</tr>
<tr>
<td>ISDS as a Design Model</td>
<td>230</td>
</tr>
</tbody>
</table>
INTRODUCTION

STATEMENT OF THE PROBLEM AND PRIMARY RESULTS

The purpose of this thesis is to develop an automated method for designing computer instruction sets that perform optimally with respect to a given class of programming tasks. The primary result is not a complete solution to the problem, but a programming system for implementing instruction set design strategies. The system (ISDS -- Instruction Set Design System) is intended to be used initially as a tool for experimenting with various design strategies and ultimately as a means for implementing the strategies that turn out to be most useful.

This thesis presents an analysis of the problem of building the design system, the details of the system, and examples and discussion showing how the system can be used. The main results are as follows:

1. Generalized Instruction Set (GIS) - a method for representing the functional capabilities of a computer.
2. The form-variable -- a data structure which is appropriate for representing GIS in the memory of a computer.
3. A set of operators for manipulating form-variables and performing tasks which are generally useful in the computer design process. These operators are written in IPL-V and are used as subroutines in an IPL-V program. They make it possible to write, in IPL-V, design strategies
and design evaluation strategies at a conceptual level which is appropriate to the computer design problem.

4. Examples and discussion of how ISDS can be used to write design strategies. The primary result is a heuristic program, written in ISDS, which determines its own strategy for optimizing a computer with respect to a given programming environment.

Many of the problems encountered in this research are related to the use of the computer as a design tool. These problems and their solutions are discussed here in the hope of adding to the rather limited body of knowledge about computer-aided design. The system described in Chapter V is experimental from the point of view of computer-aided design and from the point of view of instruction set design. It is intended as a vehicle for experimentation with the design of instruction sets and for modeling useful approaches.

The remainder of this introductory chapter discusses the design problem in more detail and explains the organization of the remaining chapters of the thesis.

ARCHITECTURAL DESIGN OF COMPUTERS

The design of a digital computer is an extremely complex process. Like most design processes it passes through many different levels of specification, working gradually from an abstract specification toward a finished product. In this paper we will be concerned
primarily with the particular level of computer specification which Amdahl [2] has referred to as architecture:

"The term architecture is used here to describe the attributes of a system as seen by the programmers, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

At the architectural level the important computer design decisions concern the memory, the instruction set, and input/output devices. Some of the properties of the memory which must be specified are its size, whether it is to have fixed or variable word length, word size (if fixed), and byte size. The instruction set must provide programmers with means for retrieving and storing information in memory and for performing operations on stored information.

The instruction set also provides means for manipulating the peripheral devices which are connected to the computer. Ideally the information retrieval and operational aspects of the instruction set coincide with the needs of the user; i.e. the machine's operators correspond approximately to the primitive operators in the user's language for describing algorithms. If the power of the instruction set is greater than the user requires, the user pays a penalty equal to the cost of the excess power. If the power of the instruction set is less than the user requires, the user pays the price of awkward and inefficient programs.

At the architectural level the design of a computer is compli-
cated enough. At levels of more detailed specification the complexity of the problem explodes to a staggering degree. A single decision at the architectural level, for example a single operation, may require complicated circuitry in the instruction set processor. The design of the circuitry may involve hundreds of decisions and many of these circuitry decisions may involve decisions about specific components. Like any design process, however, the process of designing a computer is much more complicated than a mere specification of each of the parts. The individual parts are related to each other in such complicated ways that it is not possible, in general, to make isolated decisions. Each decision depends on previous decisions and helps to determine subsequent decisions. Decisions may have to be revised as subsequent decisions provide more information about the final product.

The most recent generation of computers includes machines having several types of memory, six (possibly more) data formats, index registers, indirect addressing, several types of special registers, and many operations on data. Since it has become almost impossible for humans to deal adequately with the information processing and decision problems involved in designing a computer, it has become desirable, almost necessary, to transfer some of the human designer's responsibilities to computers. Computers have already proved to be useful as drafting aids and as simulators of proposed computers. [31] [46] Many of the computations involved in the design of logic, circuitry, and components have been relegated to computers. [19] [59] There has
been very little use of computers, however, at higher levels of the
design process. One system which is currently being developed at
Carnegie-Mellon University will design circuits from a register-
transfer description of a computer. Although this system may
eliminate much of the busy work of circuit design it cannot be used
until most of the architectural-level decisions about a computer
have been made. The specification of the architecture of a com-
puter is an important part of the design problem since it is at the
architectural level that the computer first interacts with its en-
vironment. To specify the functional characteristics of a computer
it is necessary to consider the tasks which the computer will perform
and the externally imposed constraints on the design of the computer.
The goal of this thesis is an automated method for designing the "best"
computer architecture, or instruction set, which satisfies a given set
of environmental constraints. The method, or system, can also be used
to model strategies that were used to design existing instruction sets.
Though difficult to carry out completely because of the difficulty
of comparing computers, this approach can be useful in determining
the parameters that were considered in the design of an existing
instruction set.

This thesis presents a method for designing computer architecture.
The main concern is with specification of the instruction set of the
computer -- i.e. with the selection of the operations and addressing
capabilities. This problem is necessarily intertwined with another
problem, which is central to the thesis -- the problem of determining
what information is required to communicate instruction to a computer and how the information should be represented to permit efficient execution of stored programs.

Two factors account for much of the complexity of the selection of an instruction set. One is the number of solutions to the problem. Different combinations of memory size, memory structure, addressing capabilities, and operations account for a large number of possible instructions. This is a problem which must be dealt with by any system which designs instruction sets. One of the results of this study is that there are relatively simple strategies which make it possible to find appropriate instruction sets with very little effort.

A second complicating factor is the difficulty of evaluating the finished product. It is desirable in any design system to be able to evaluate the partially specified product at various stages of its design. But the problem of measuring the performance of an instruction set is roughly equivalent to the problem of evaluating an entire computer system -- a problem which is far from solved, though it has received much attention in computer literature. The difficulty seems to be that "performance" of a computer is meaningful only with respect to specific tasks which the computer must perform. It is proper to ask, "How long does it take computer x to run program y with input z?" It is probably meaningless, however, to ask, "How good is computer x?"

Any adequate evaluation procedure must be based on the tasks the computer will perform. This raises the question of how the tasks should be expressed to the design program. The most ideal method would be
to present tasks in a high-level language. But if the design program is to perform accurate evaluations it must know how the computer will execute the given tasks -- i.e. the design program must be able to write programs. Although simple program-writing programs have been implemented [61] it is not yet feasible to include a program-writer as part of a design system.

Although it is not feasible to automatically match computers precisely to a given task environment, there are some approximate methods which may be satisfactory in many cases. These methods correspond to the techniques which are used in practice to evaluate computer systems -- simulation, instruction mixes, benchmark programs, etc. The use of these methods to match computers and tasks is discussed in Chapter II. One of the hopes for the research presented here is that it will contribute indirectly to the improvement of evaluation techniques by making it possible to experiment with many different methods.

THE COMPUTER AS A DESIGN TOOL

Throughout this thesis the general design problem is viewed as the gradual transformation of an abstract object into a specific object. At one extreme the solution to a design problem can be thought of as an abstract object which fulfills the needs and satisfies the constraints given in the statement of the problem. At the most specific level the solution is a physical object. Although there may be a continuum of levels of abstraction between the most abstract and the most specific forms of a solution there are several intermediate points which commonly receive special attention. Some of
these are the designer's initial concepts of a solution, sketches, blue prints, prototypes, and models of solutions. Although a computer may be a useful aid at any level of a design process, most uses of the computer as a design tool have been at the lower levels where most of the important aspects of the solution are already specified. In the last five years some research has been done in the direction of using the computer at higher levels of the design problems. (See the discussion in Chapter III.) Much of this research has been on interactive systems in which the computer performs computations and displays the result to a human designer who uses the new information to specify more details of the solution. There are only a few examples of programs that make high-level, possibly "creative", design decisions; no general techniques for writing such a system have been developed. [8] [25] There is a need for research in the general area of automated design systems and computer-aided design. Many questions must be answered before the computer can become a generally useful design tool. What, for example, is the appropriate way to represent a partially specified object in the memory of a computer? Is there a universally applicable representation? Must design systems be interactive? How much intelligence can a design system have? Although this thesis does not deal directly with these issues, it is likely that the knowledge gained from a number of relatively ad hoc experimental systems will point the way toward a multi-purpose automated design system.
CHAPTER I

INSTRUCTIONS FOR COMPUTERS

One of the main concerns of this thesis is the selection of instructions for computers. The present chapter provides the background information necessary for an understanding of the problem of designing an instruction set. The discussion is in three sections:

1. Definition of the design problem.
2. Examination and classification of existing instruction sets.
3. Discussion of general design considerations.

DEFINITION OF THE PROBLEM

For the purposes of this thesis the instruction set design problem is assumed to be the following:

Given: a) Information about a class of programmable tasks
     b) A fixed technology -- i.e., a set of technological capabilities and costs

Find the processor, or instruction set, which performs all tasks in the given class for the least cost. Specify the solution at the following level of detail:

Memory
   size of main memory, word organization, (fixed word length vs. variable word length), structure of processor registers, data representation methods for specifying addresses

Instructions
   operations, encoding of stored-program instructions

The solution to the design problem, then, is a functional description of a computer similar to the description in a programmer's manual. Although the goal is to specify an instruction set to the level of detail indicated above, we will assume that the implementation
of the functional capabilities we specify has some of the global properties of existing computers. The memory, for example, can be considered to be a linear array of zero-one bits. Any regular structure may be imposed on the bits of memory. For example, they may be grouped according to a fixed word size.

We will also assume that a computer is driven by a program which is stored in main memory. A program in this sense is a set of characters which can be interpreted, by the computer's instruction processor. The computer performs no action until the instruction processor reads a portion of the stored program and translates the information into a series of actions. The instruction processor determines the memory location of the next instruction either automatically or from information contained in the last instruction. This fetch-execute process continues until the instruction processor is stopped either internally or by an instruction.

In the above context an instruction is any unit of information which can cause the instruction processor to perform an action or a series of actions. The set of all such units of information is the instruction set of the computer. Chapter IV gives a precise model for the concept of an instruction set. In the present discussion we will indicate some of the information contained in the instruction set of a computer and how that information can be represented for practical purposes.

A typical instruction in a modern computer might specify two addresses in main memory and one operation. When the instruction is executed the instruction processor obtains the information in the
two addresses, performs the operation using the stored information as operands, and returns the result of the operation to one of the operand addresses. We note then that if we are to completely specify the instruction set of a computer we must specify the manner in which information in memory is addressed. We also note that by the definition of instruction set every combination of operation-address-address is an instruction. For purposes of readability and generality it is desirable to represent classes of instruction rather than single instructions. A class of instructions can be represented by an instruction form. Each category of information in the instruction form can be thought of as a variable over a precisely defined domain of values. Specific instructions are obtained from instruction forms by specifying one value for each variable in the form. For some simple computers the entire instruction set can be represented by a single instruction form. For most computers however it is necessary to represent the instruction set by a set of instruction forms. In such a case we will refer to the set of all instructions obtainable from a given instruction form as an instruction group. The instruction set processor of a computer must be able to interpret each instruction group. In particular the processor must know for any given instruction the form for the instruction and the exact location within the instruction of each variable. This information is also an important part of the specification of an instruction set.

Our definition of the instruction set design problem is a restricted version of the design problem as it occurs in practice. The main reason for this restriction is that our goal is to design
the functional capabilities of computers independent of their imple-
mentation. We will consider briefly how the problem, as we have stated
it, is related to the more general design problem. In the general
problem the memory and instruction set must be specified as in the
restricted version of the problem. However in practice it is also
necessary to specify some details about the operation of devices
which can be controlled by the computer. It is generally necessary,
for example, to specify how input/output information is transmitted.
In the restricted version of the problem we will be concerned with
external devices only to the extent that they influence the instruc-
tion set. For example, with regard to an output device we will
specify only what information (operation) is required to start the
device and (if necessary) to specify the memory locations to be trans-
mitted. We will adopt the same approach toward certain functions such
as overflow checking, interrupts, and sense switches, which vary
greatly from one machine to another. We will recognize these items
as components of a computer and we will construct our design system
in such a way that it can build computers having these features, but
we will specify only the existence of such features and the information
that is required to envoke them. This will be our position on many
such issues since we are concerned only with designing the broad
functional capabilities of computers.

CLASSIFICATION OF INSTRUCTIONS

A computer can perform only the functions which can be expressed
in the language of its instruction set. In a sense the capabilities
of a computer are defined by its instructions. Within the class of stored-program computers which we are considering it is possible to identify a set of general capabilities which every instruction set must possess. These requirements are a direct result of the fact that both data and instructions are stored in memory and can be retrieved only by their addresses. An instruction set must perform the following functions:

1. Store and retrieve data in memory
2. Perform operations on data
3. Fix the sequence in which stored instructions are executed
4. Control external devices and transmit data for I/O devices

In the following sections we will consider some of the methods used in existing computers to meet these requirements.

**Storage and Retrieval of Information in Memory**

Before the instruction set processor can alter data in the memory of a computer it must know which portions of the memory are to be affected. We will assume that all memories are divided into addressable units of information. This means that the bits of the memory are somehow partitioned into relatively small clusters of contiguous bits and that the clusters are in a one-one correspondence with a set of numbers (or, more generally, a set of character strings). The number (or character string) associated with each bit cluster is called the address of the bit cluster. When a programmer wants the instruction processor to obtain or alter data in memory he provides the processor with the address of the data. Computers vary as to
the size of an addressable cluster of bits. In general the smallest
addressable unit of memory is either a character (or byte) or a word.
A "character" represents enough bits to encode a single symbol of some
alphabet, whereas a word is usually large enough to contain arithmetic
values to the degree of accuracy required of the computer (eight
decimal digits, for example). The principles of communicating
memory addresses to the processor are similar for character and word
addresses.

In the following sections we will consider the most important
addressing methods. For a more detailed discussion of these methods
and their variations the reader may refer to a standard text on machine
language programming or computer design. (See, for example, [5] and
[37])

Immediate, Direct, and Indirect Addressing. It is standard
practice to think of a computer instruction as containing an operation
and, possibly, one or more "addresses". However the "addresses" of
an instruction are not necessarily memory addresses of the desired
data operands. It may be more accurate to call the "addresses" of
an instruction data references. There are three standard ways in
which data references are interpreted in existing computers:

1. Immediate Reference -- The data reference itself, i.e.
   information in the instruction, is used as data in the
   execution of the instruction.

2. Direct Addressing -- The data reference is interpreted as
   the memory address of data to be used in the execution of
the instruction

3. Indirect Addressing -- The data reference is interpreted as the address of a memory location which contains a second data reference which is either direct or indirect. If the second data reference is direct the data is obtained with "one level of indirectness." Since the second data reference may be indirect it is possible to reference data via any number of levels of indirectness.

Instructions may operate on more than one character or word. If this is the case then the processor must observe conventions for locating groups of characters or words when given only a single memory address. Generally this is less of a problem when addressing multiple words than when addressing multiple characters. Multiple-word instructions are sufficiently rare that the practice is to define for each multiple word instruction the precise set of affected words. For multiple character instructions, however, it is sometimes desirable to specify within the data reference the number of characters in the operand. Whether the characters reside in addresses above or below the data reference is established either by convention or by explicit information within the data reference. An alternative to explicit mention of the operand size is a special bit or character in the memory which defines the boundaries of operands. The word mark of the IBM 1401 and the flag bit of the IBM 1620 serve this purpose.

Indexing and Augmented Addressing. A data reference may appear explicitly in an instruction. For purposes of greater programming flexibility or because of certain design considerations (which are
discussed in Chapter II), however, it is sometimes desirable to compute values of data references as functions of several arguments. The two most frequently used techniques are called indexing and augmented addressing. \[37\] The functions used to compute values of data references by these methods are, respectively, addition and bit-string concatenation. An indexed data reference is the sum of an integer which is explicit in the data reference and the contents of an "index" register. If the machine has more than one index register the address of the desired register is a part of the data reference. An augmented address is obtained by concatenating (left or right as set by convention) an explicit portion of the data reference with the contents of a register or memory location which is specified either by convention or by an address in the data reference.

There are many useful combinations and variations of the techniques mentioned above. We will not consider these methods since our purpose at this stage is merely to explore the primary alternatives that should be available in a system for designing instruction sets.

**Operations on Data**

Operations are associated with particular data types. For example, a computer having binary arithmetic data and floating point decimal arithmetic must have sets of operations for both classes of data. Most of the arithmetic data formats in existing machines are either fixed point (binary, or decimal) or floating point (binary,
or decimal). To achieve greater accuracy in arithmetic operations some fixed-word-length machines have operations that treat 2 or 3, possibly more, words as a single arithmetic operand.

In addition to arithmetic operations many computers have operations that treat data as logical information (string of bits) and as character information (string of alphabetic characters). There are 16 binary logical operations that can be performed on 2 logical operands. If a computer permits logical data it generally must provide at least a theoretically complete set of logical operators. A set of logical operators is "complete" in this sense if all 16 operations can be expressed as combinations of the operators in the set. The operators not and and, for example, form a complete set. The subtract operation is complete for arithmetic operations. In addition to the binary logical operations many computers also permit shifting or rotating of a string of bits. There are numerous variations of shifting. The operations on character strings usually involve moving character data from one memory location to another or editing operations for the purpose of making printed output more human-readable.

Controlling the Sequence of Instruction Execution

The usefulness of digital computers derives largely from two characteristics:

1. They are capable of repeating a sequence of operations any number of times.

2. They can distinguish between various cases in information and execute different instructions accordingly.

Sequence-setting instructions are frequently called branching
instructions. The simplest branch instruction is an **unconditional branch**, which explicitly gives the address of the next instruction to be executed. The unconditional branch by itself gives a computer the power to loop -- i.e., to repeat a sequence of instructions. In order to exit from loops and execute different instructions for different cases of input data, however, a computer must have at least one **conditional branch instruction**. Most conditional branch instructions cause the instruction sequencer to proceed either to the next instruction in memory or to the address contained in the conditional branch instruction depending upon the setting of one or more internal switches. Hence in the general case the conditional branch instruction must provide the instruction processor with both the condition being tested and the conditional **branch address**. The internal switches, or conditions, being tested are normally set as the result of an operation or an instruction which explicitly sets a condition. In some cases the condition is set and tested in the same instruction, so that the setting of the condition is implicit and does not necessarily involve the setting of a hardware switch. The two modes of setting and testing conditions are illustrated by the following examples:

1. Set switch x if y = 0. Branch to z if switch x is on.  
   (Two instructions)

2. Branch to z if y = 0. (One instruction)

The information required by the processor is the same in both cases -- the addresses y and z and the name of the switch, x. The second example seems to have the advantage of requiring only one instruction, however, in many modern machines there are not enough bits in
an instruction to express \( y \) and \( z \) in a single command.

Since case 2 above can be considered to be a special case of case 1 we will use case 1 as a model of conditional branching. That is, we will assume for purposes of a model that a hardware switch stores the result of each condition test and that branches occur or fail to occur depending upon the status of the switch. In particular cases, of course, it is also necessary to establish conventions for the setting and re-setting of conditions. Generally conditions are reset either when they are queried or when they occur in a branch instruction.

Some computers have a more elaborate branch instruction which, in addition to causing a branch, causes the address of the instruction following the branch to be saved in a register. This instruction is useful for linkage to subroutines. A return or branch back instruction when the subroutine is completed causes normal sequencing to be resumed.

One other instruction, the execute instruction, should be included in a discussion of sequence-controlling instructions. An execute specifies to the processor the address of a memory location, either main memory or register memory, which contains a single instruction. The execute instruction causes the single instruction at the specified memory location to be executed. Control then returns to the instruction following the execute. In some machines portions of the executed instruction are altered automatically by the contents of specified memory location so that the programmer can automatically set values of parameters in the executed instruction.
Control of External Devices

An instruction set must contain commands for controlling external devices. The primary responsibility of these commands is to start input/output devices, to locate data to be transmitted to output devices, and to locate input regions for input devices. In some machines the input and output regions are fixed so that region addresses are not required in I/O commands. Most instruction sets include some instructions for controlling mechanical features of I/O devices, such as line spacing, card selection, etc. and instructions which test, (in the manner of the conditional branch) the mechanical status of various features of external devices (page overflow, last-card switch, punch check, etc.).

Associated with each type of instruction discussed above is a certain minimal amount of information which must be supplied to the instruction processor before it can perform the function. In order to interpret an add operation, for example, the processor must have the following information:

1. The add operation code
2. The locations of the operands
3. The memory location which is to receive the sum
4. The address of the next instruction to be executed

Since the address of the next instruction may depend upon the status of a condition, the instruction may contain 2 next-instruction addresses -- one which holds if the condition is satisfied and one which holds if the condition is not satisfied. In general, then, there are 5 addresses, an operation, and a condition associated with a
single binary operation. For practical reasons, however, it is desirable to maximize the number of operations per bit in the instruction stream. The length of a single instruction is usually restricted to a single word length in a fixed-word-length machine. Since the information required to specify 5 addresses generally requires more than a single word, most machines use fewer than 5 addresses. The omitted addresses are assigned fixed interpretations. For example, the next-instruction addresses might be assumed to be the address of the current instruction plus one. This trade-off between addressing flexibility and the practical need to maximize information content is one of the central issues in the design of an instruction set. Bucholz [10] identifies it as the most important single design consideration. We will consider this problem in more detail in Chapter II. In the remainder of this section we present addressing arrangements that have been used in the past.

Three-Address Formats. The three address format has two operand addresses, A and B, and a result address, C. The contents of A and B are combined under the operation and the result is stored in location C.

Two-Address Formats. Many of the so-called "data processing" computers use the two-address instruction format. A typical instruction in a two-address machine has an operation code and two operand addresses, A and B. The contents of A and B are combined according to the operation and the result replaces the contents of location A or location B. Instructions are executed sequentially until a "branch"
instruction alters the sequence. Branch instructions in a two-address computer generally have a different instruction format than data operations since only one address is necessary.

A second type of two-address format, sometimes called the one-plus-one format, uses one operand address, A, and one next instruction address, B. [37] The contents of A are combined with the contents of an accumulator and the result replaces the contents of the accumulator. The next instruction is taken from address B.

The One-Address Format. The one-address format uses a single operand address, A, for data operations. The contents of A are combined with the contents of an accumulator and the result is stored in the accumulator. Instructions are fetched sequentially from memory until a branch instruction alters the sequence. A branch instruction in a single address computer can have the same format as a data operation since only one address need be specified. The accumulator in a single-address computer is usually a single register capable of storing each of the computer's data formats. In cases where it is desirable to have a different accumulator or an extension (M-Q) of the accumulator for certain operations (as is sometimes true of multiply/divide operations) the accumulator is implied by the operation code of an instruction.

The Zero-Address Format. The zero-address, or stack, computer was designed primarily for use in symbol manipulation and rapid compilation of algebraic programming languages. The accumulator of a zero-address computer is a hardware pushdown stack, which can contain a large number of operands. The only information required in a data
operation instruction is the operation to be performed. The top two operands in the accumulator are combined under the operation. The operands are removed from the accumulator and the result becomes the new top element of the accumulator. It should be noted, however, that some instructions in a stack machine must contain at least one address, since it is necessary to tell the processor which portions of memory are to be placed in the stack. Branch instructions in a stack machine also require at least one address unless they are performed indirectly via the accumulator.

Some computer instructions cannot be classified simply as op-code and 1-address, op code and 2-addresses, etc. The main reason is that it is sometimes desirable to perform, in one instruction, operations which are much more complicated than the normal arithmetic and logical operations. It is possible, though not always economically feasible, to provide high level operations which normally would require long subroutines or programs. (A square-root operator, for example). A more feasible device for providing high-level operations is the micro-instruction. A sequence of micro-instructions is executed by an operate instruction. An operate instruction contains a sequence of program bits. Each bit of the program portion is associated with a certain primitive operation such as add, clear, shift, etc. The processor interprets the program bits sequentially, executing the associated primitive operation for each program bit that is set to 1. It is also possible to execute primitives conditionally; i.e. the operation associated with a bit is performed only if the bit is set to 1 and a certain condition was satisfied by a previous operation. The PDP8, for example, contains two operate instructions each capable
of executing eight primitive operations on the accumulator and the
link (a one-bit register associated with the accumulator). The prim-
itive operations include shift left, shift right, shift 2 left, shift
2 right, complement accumulator, complement link, clear accumulator
and clear link. Micro-instructions provide a compact representation
for a sequence of operations. In a single-address machine like the
PDP-8 there is no room in the operate instruction format for an address.
Hence micro-instructions affect only the accumulator or the link. In
general an operate instruction with no parameters can perform more
micro-instructions than an operate with input parameters. Between
the operate instruction and the single-primitive instructions there
are instructions that perform 2 or 3 primitive operations. One
example is the store instruction of The Rice Institute Computer which
has two operation codes -- one causes a store; the second can be used
to set the two "tag" bits to any configuration. The "tag" bits are
used by the processor to interrupt instruction execution in various
ways depending upon the tag configuration. Although micro-instructions
and multiple-operation instruction can be very powerful, most computer
instructions have one operation and 0, 1 or 2 addresses.
CHAPTER II

SELECTION AND EVALUATION OF INSTRUCTION SETS

In current practice the design of an instruction set differs from most design processes in a very important way: it is relatively free of the iteration or "looping" that characterizes most design processes because it is very difficult to predict the performance of an instruction set. This is not to say that it is possible to proceed directly from a set of constraints to an optimal instruction set. As in any design problem the interactions between design parameters are very complex, and local decisions frequently have an effect on subsequent or, possibly, previous decisions. However once a feasible solution is discovered it is very difficult to improve it for the simple reason that no satisfactory evaluation technique exists. Most design processes involve specify-evaluate-re-specify cycles (Grason [25] calls this process melioration) in which the desired object, or a portion of the desired object, is gradually improved. This process would be desirable in the design of an instruction set, but with current methods it is impracticable. There are no completely satisfactory methods for evaluating an instruction set without implementing it. Moreover, it is generally meaningless to evaluate a single aspect of an instruction set. The only satisfactory method of evaluation used to data is simulation in the task environment. Once an instruction set is specified, a set of representative jobs from the task environment is programmed for the proposed machine and the programs are
executed by a "simulation" program with estimated instruction timings. In general if the instruction set is changed it is necessary to re-
program the sample tasks and to construct a new simulator for the new instruction set. For these reasons simulation has been used primarily in the final stages of the design process and most in-
struction sets have been designed in a relatively loop-free manner.

Although there are no useful theoretical methods, there are approximate methods for evaluating computer performance and practical rules-of-thumb for selecting instruction sets for various classes of tasks. The evaluation techniques attempt to extract enough significant information about the computer and the task environment to estimate how well the computer will perform. Most of the rules-
of-thumb for selection are based on observed characteristics of the task environment and instruction features that have been useful in the environment in the past. It is also possible to identify the trade-offs between parameters which should be considered in the de-
sign of an instruction set. The purpose of this chapter is to pre-
sent the methods that can be used to evaluate computers and the prac-
tical and theoretical considerations involved in the selection of an instruction set. We will begin with a discussion of evaluation of computer performance. The methods described here are not new. They have been used but with only limited success. Calingaert [12] compares many of the methods and shows some of their limitations.

EVALUATION OF COMPUTER PERFORMANCE

There are two generally respected true measurements of computer
performance:

1. The time it takes a computer to execute a task or set of tasks.

2. Thruput -- some measure of the volume of work performed by a computer in a fixed time. [3]

In practice, of course, information obtained in a timing or throughput analysis must be weighed together with system costs, the cost of preparing data and algorithms, and the users resources and requirements. Execution timings and throughput analysis are distinct measurements. Program timings are relatively local measurements, whereas throughput analysis gives a more global measurement of system performance. Throught analysis includes consideration of multi-programming capabilities, set-up times, overlapping, and executive time. Although these factors are important the main problem in measuring performance is the problem of determining execution time for single programs or classes of problems. We will concentrate primarily on this problem.

One way to obtain a true execution time for a task on a computer is to execute an actual program with actual input data. We will regard these three components -- execution, the program, and data -- as the important components in a system evaluation. In a true evaluation each component must be measured precisely. It is possible, however, to obtain approximate evaluations by approximating the values of one or more of the components. We will describe several approximation techniques in the following sections.

Approximations to Input Data

First let us consider the problem of selecting data for a pro-
gram to be tested on a given computer. There is almost always a statistical element involved in such a test, since it is often impossible to test the program for all cases. Given any set of test data or actual data we must ask, "How representative is this data of actual data over a long period of time?" It is also important to ask, "How sensitive is the execution time to variations in the input data?"

The execution time for some programs is relatively independent of variations in input data. On the other hand, execution time for a single case may depend directly on values of input variables. In such cases it may be very difficult to determine a representative set of input data. This illustrates the importance of basing evaluations on actual data. It is meaningless to evaluate computers independent of programs and data.

Any set of test data is an approximation to live data. In most cases, however, it is probably possible to select, by careful analysis, a relatively small but representative set of test data. Careful selection and statistical analysis of data must be a part of any accurate evaluation technique. There have been attempts to base evaluation on general information about input data. This approach may be useful for some relatively simple data processing tasks, but in general the validity of an evaluation is proportional to the validity of the test data.

**Approximating the Programming Environment**

We have already pointed out that one way to obtain precise performance evaluations is to test actual programs. In practice this may be impossible if evaluation is desired for a large number of com-
plex programs or a large, loosely defined class of programs. In such cases one feasible approach is to perform execution timings on a "representative" subset of the programs. The selection of a "representative" subset, of course, may be very difficult since it involves the comparison of programs. Joslin and Aiken [33] have illustrated elaborate techniques for partitioning classes of programs into subsets in such a way that any member of a given subset is typical of the subset. A performance estimate for the entire class of programs is obtained by combining estimates for programs from each subset with weighting factors to allow for the relative importance of each subset. The programs which are finally selected for testing are called benchmark problems. According to Joslin [32] benchmark programs must be representative of "types of processing", "time requirements", "equipment requirements", and "the order of the problems (priority)." As with most approximation schemes the finer the partition the more accurate -- and expensive -- the evaluation. The main problem in such an evaluation is to find a relatively coarse partition that yields a useful evaluation.

Frequently in practice computers are evaluated by testing their performance on a few standard programs called kernels. The kernel method and the benchmark method differ in degree; the benchmark problem, if properly selected, is more likely to represent a portion of the task environment than a kernel. [3] [12] Extreme caution must be used when the kernel and benchmark methods are used to compare computers. In many cases the best algorithm for a task depends upon the machine which is to be used. Hence a benchmark or kernel problem yields an
accurate comparison of two computers only if the program for each computer is the best algorithm for the computer and if it makes the best possible use of the computer's features.

**Approximating Execution**

It is often desirable (or necessary) to estimate program execution times without running the program on an actual computer. In some cases it is possible to obtain approximate execution times from general information about the program and the computer being evaluated. The most useful substitute for actual execution is simulation. (See, for example ref. [46].) Simulation, in this sense, is the programming of one computer to behave like another. The main attraction of simulation is that it can be used to evaluate a computer before it is constructed. The most important drawback of the simulation approach is that it is very difficult to write a program that accurately mimics the behavior of another computer in every detail. In practice simulation programs usually duplicate the gross behavior of a system. Although it may be possible to simulate every detail of the execution of each instruction it may be much more practical, though less reliable, to simulate only the most important aspects of each instruction.

We have described the most useful techniques for approximating actual data, programs, and execution. These methods are often used in combination. One commonly used method attempts to evaluate performance on the basis of actual data, an approximate program, and approximated execution. The object of this method is to determine the number of executions for each of the computer's instructions.
during an execution of the program. This type of information may be obtained easily if the program is already written for a machine with approximately the same instructions as the machine being evaluated. The instruction frequencies, together with instruction timings for the machine being evaluated, give an estimate of total execution time for the program. A table of instruction classes and their associated frequencies is called an instruction mix. (See [12], [26], [63], [53].) To be useful an instruction mix should be based on actual data. That is the relative frequency of each class of instructions should be determined by tracing the execution of a program on actual data. There are hardware devices which facilitate the collection of this type of information, [3] [22].

Instruction mixes have been compiled for various classes of task environments. Knight [35], for example, gives mixes for scientific and commercial applications. There are, however, several serious objections to using an instruction mix taken on one computer as a basis for evaluation of another computer. As we have pointed out, the best algorithm for solving a problem may depend upon the computer being used. It may turn out that the best mix of instructions on the computer being evaluated is different from the mix on which the evaluation is based. Even if the instruction repertoires of the two machines are almost identical the value of an instruction mix is questionable. As Calingaert [12] points out, it is extremely difficult to compare even single instructions that vary in such details as operand length and recomplementation. Another objection to instruction mixes is that they make it almost impossible to take into account the
the effects of instruction look-ahead mechanisms used in some modern computers. For our purposes mixes can be of some use, however, because all of the machines we design can be assumed to have the same technology. Chapter VI illustrates the use of an instruction mix as the evaluation function in a closed-loop design strategy. In general, however, instruction mixes are another form of approximate evaluation.

A discussion of evaluation methods would not be complete without at least some mention of several very simple rule-of-thumb methods that have been used. Computer evaluations have sometimes been based on single attributes such as memory size, memory speed, add time, average instruction execution time, etc. Knight [35] gives a simple formula for "computing power", \( F_{ni} \), "the number of equivalent operations performed per second." The formula is:

\[
F_{ni} = M \times \frac{10^{12}}{t_c \cdot TI/O}
\]

In this formula \( M \) is memory size; \( t_c \), the internal speed of the processor; and \( TI/O \), processor idle time. All of these simple evaluations are very crude because they give, at best, minor consideration to programs and data. The parameter \( t_c \) in Knight's formula is based on an instruction mix, but in order to obtain an accurate value of \( t_c \) for a given class of programs it would be necessary to obtain an accurate instruction mix by executing actual programs. Hence Knight's formula if used carefully is approximately equivalent to the instruction mix approach.

Although there is room for improvement in existing evaluation techniques it seems likely that the ultimate measurement of a computer...
will always be on-the-job performance and that any simpler evaluation will be subject to error, the reliability depending to some extent upon the amount of work involved in the evaluation.

SELECTION OF AN INSTRUCTION SET

The problem of selecting an instruction set fits the currently accepted paradigm of a design process. We will consider some of the details of design theory in Chapter III. In this section we will deal with the following types of considerations which are typical of design problems:

1. A problem is initially identified because some need must be met. In our case the need is for a set of operations for a computer.

2. A solution to the problem meets the specified need, but it must also function effectively in a certain environment. The statement of the problem may explicitly specify certain constraints which must be satisfied. As the solution is specified in more detail new interactions of the solution with the environment become apparent to the designer, thus introducing new constraints which must be satisfied.

3. The specification of the solution proceeds through various levels of abstraction, until it is complete. The specification process, though serial, may involve many trial and error loops. During the process the many complex interactions between different parts of the solution must be considered. The purpose of this section is to analyze the
instruction set design problem in the light of the considerations mentioned above. What needs must an instruction set meet? What is the environment of an instruction set? What effects do environment have on instruction set design? What are the "parts" of an instruction set? How do the "parts" interact during the design process? Not all of these questions can be answered completely. But if we are to build an automated system for designing instruction sets we must have at least partial answers to the above questions.

Computer literature contains no thorough protocols of the instruction set design process. The most complete analysis is Buchholz's essay in *Planning a Computer System: Project Stretch*.[10] Buchholz's article deals largely with the evolution of instruction sets toward the single-address format, a result, presumably, of increased memory sizes and increased numbers of operations. Many of the parameters considered in the design of the IBM 360 are described in reference [9]. These articles offer a mild contrast to Buchholz's point of view since the 360, a successor to Project Stretch, was given a two-address instruction format at the expense of increased complication in addressing. Considerations in the design of other instruction sets are only incidental in computer literature. In the remainder of this chapter we will attempt to identify the most important factors. A more complete list can be compiled only when tools are available for detailed experimentation with various features of instruction sets in different task environments.

One useful approach to design is to attempt to separate the
solution into a collection of relatively self-contained parts. This technique is emphasized by Alexander. [1] Although we cannot generally hope to isolate totally independent subproblems, it is often possible to identify "clusters" of design parameters that seem to enter into the design process as single, relatively self-contained entities.

The design of automobile tires and automobile engines, for example, can be considered as relatively, though not totally, independent. But the design of a carburetor or electrical system cannot be isolated from the engine design because there is a relatively high degree of interaction between engine and carburetor, or engine and electrical system.

In addition to dealing with this type of analysis Alexander considers an automated system for partitioning design problems (partially specified solutions) into "clusters."

In the case of an instruction set it is possible to identify the following categories of sub-problems:

1. Structure of memory
2. Data formats
3. Operations
4. Addressing features
5. Instruction encoding

Many of the design activities within each of these categories can be treated as sub-problems. For example, the specification of a data format, word size, or address displacement size can be treated as a subproblem. The five variables listed above can be considered as both independent and dependent variables during the design process. Each is dependent in the sense that its value depends on other vari-
ables and independent in the sense that it helps to determine the value of other variables. To consider the entire design problem we must consider two additional variables which we assume always to be independent -- the task environment and the technological environment.

Before we proceed with our discussion of design variables, let us make some clarifying remarks about the task environment. By task environment we mean the totality of programs which the computer will have to execute. Frequently we may think of the task environment as a class of problems, such as algebraic computation, data processing, etc. We must acknowledge, however, that, in almost any such task environment, high level languages are the modus operandi. It is possible that a significant percentage of execution time will be devoted to the compilation of object programs, which is a symbol manipulation-data processing application. This need not be the case if programs are compiled on another machine or if production run time for each program far exceeds total compile time required to get the program to production status. However in many installations compiling is recognized as an important activity. Compiler writers have become aware that certain machine features are valuable aids in a compiler type of program. Randell and Russell, for example, [54], use a single-address meta-language to compile algebraic expressions. From the point of view of efficient compiling the instruction set of the base computer should resemble their meta-language. This type of consideration was obviously recognized by the designers of the Burroughs B5000 computer. One of the most efficient algorithms for compiling algebraic expressions makes use of pushdown stacks to store operators and operands. The B5000, designed expressly for rapid compilation of
algebraic expressions, contains hardware pushdown stacks for storage of operands and operators. The translate instruction of the IBM 360 provides hardware recognition of specified characters in a character string. This is another example of a hardware feature intended to aid compiler-writers. So far, however, no computer stands out as a successful blend of the features required for both efficient compiling and efficient execution of object programs. There is much room for improvement in this area of instruction set design.

There is another way in which the use of high-level languages affects the design of a computer. For most modern computers the lowest practical language level is a symbolic language whose instructions correspond approximately to machine instructions. If the computer instruction set is very complex then the program which translates the symbolic language to machine language must be very complex. This illustrates a trade-off which must be treated delicately and probably deserves more attention than it has received in the past. Added complexity in an instruction set may be desirable from the point of view of application programming and writing language translators, but each added complexity increases the complexity of the symbolic language translator and, ultimately, the ratio of translate time to production time. The answer to this dilemma seems to be an instruction set which is powerful from the application programmer's point of view and yet classically simple from the point of view of writing a machine language program to translate the symbolic language to machine language.

We will now proceed with our discussion of the instruction set design variables. In the remainder of this chapter we will show how
each of the five dependent variables depends upon the other design
variables. The discussion is general; it is not intended as a detailed
description of all considerations in the design of an instruction set.
Our purpose is to analyze the types of relationships that exist be-
tween the design variables. These relationships are summarized in
fig. 1. In Chapters III, IV, and V the relationships between design
variables will enter into our analysis and synthesis of an instruction
set design system.

Memory Structure

The variables most active in determining memory structure are the
task environment, technology, and data formats. In order to specify
memory structure we must specify the following components of the
memory: size, word structure, (fixed or variable) and memory type.

Memory Size. Memory size is determined primarily by the task
environment and cost constraints imposed by technology. The memory
must be large enough to accommodate most programs, together with their
data, of the task environment. We say most programs because it is
usually possible to deal with exceptionally large programs by means
of auxiliary memory devices or program overlays. It is desirable,
however, to avoid the costly information swapping that becomes necessary
when auxiliary memory devices are used.

There is considerable variation in the amount of memory required
by different types of tasks. Complicated symbol-manipulation or
matrix manipulation programs may require more than a million characters
for program and data. At the other extreme, some data processing
programs may fit very comfortably into fewer than 1,000 memory
RELATIONSHIPS BETWEEN THE DESIGN VARIABLES

Figure 1
locations. Many installations have functioned successfully with computers having 4,000 or fewer locations in memory. At this end of the scale, however, the most constraining factor is very likely to be the memory space required for compiler and assembly programs. The IBM 1401, for example, is available with a 4,000 character memory, but it is very difficult to implement a thorough and efficient assembly program for the 1401 in 4,000 storage locations. It is not practical to implement higher level languages like Algol or Fortran on a computer with such small memory capacity. In general the memory of a computer should be large enough for any task in the task environment. If exceptions seem necessary the cost of swapping information between main memory and auxiliary memory must be weighed against the cost of additional main memory locations.

We have mentioned that memory size is constrained by technological costs. One of these costs is the cost of addressing memory locations. As memory size increases either the number of bits required to specify an address or the amount of internal processing required to compute an address increases. This point illustrates an interaction between memory size and instruction size. If word size is fixed, then, unless augmented addressing is used, memory size is constrained to be smaller than the number of locations that can be addressed in the portion of the instruction format which remains after all are assigned. Conversely, of course, instruction size can be treated as a variable depending upon memory size. The amount of dependence between word size and memory size depends largely on the size of addressable units of the memory. Changes in the size of an 8-bit byte memory will cause more repercussions in the design than changes in 64-bit word addressable
memory containing the same number of bits.

**Word Structure.** One important decision that must be made with respect to the word structure of a memory is whether the word length is to be fixed or variable. A variable word length memory has the advantage of compact representation of data and programs. For very irregular types of data the structure of a fixed word memory may result in inefficient utilization of the memory. The cost of a variable word-length memory is the time and memory space required to define data boundaries. There are two methods for defining data fields in a variable word-size memory. One is to include "field length" as a parameter in each address. This approach requires bits in the instruction format and additional internal processing during each memory access. The other solution is to provide a special character, or a special bit in memory, to identify data boundaries. The cost of this approach is extra instructions in the instruction set to manipulate boundary markers and execution-time processing to set, clear, and search for boundary markers. Ultimately the decision between fixed and variable word size depends upon the "regularness" of data and, perhaps, programs in the task environment. Records in a "data processing" application are often naturally variable in length. Most of the data elements in a symbol manipulation or matrix manipulation application may be similar enough to warrant a single word length. The designer must use the characteristics of data in the task environment to weigh the cost of extra processing in a variable-word memory against the cost of wasted space in a fixed-word memory.

Another consideration regarding word structure is that more in-
struction bits are required to address byte-addressable memory than
word-addressable memory. Byte addressable memory is desirable, however,
if large amounts of information must be stored in small fields.

In a variable-word memory, addressable units are usually bytes,
or characters. The number of bits per character is determined by the
number of characters in the computer's character set and the scheme
used to encode characters. In general it is possible to encode $2^n$
characters in $n$ bits. The number of characters required in the character
set, of course, depends directly upon the task environment. In a
fixed-word-length memory the character or byte size is determined as
above, but the word size should be an even multiple of byte size so
that byte and word boundaries coincide.

Considerable attention must be given to the selection of word
size in a fixed-word-length machine. Word size is very closely re-
lated to the data formats of an instruction set. In most cases the two
are probably designed in parallel, since the various data formats must
be stored in single words or word groups as efficiently as possible.
Usually the length of each data format is equal either to the word
length or to a multiple of the word length. Word length, therefore,
is determined largely by the number of bits required to represent
arithmetic data with the accuracy required by the task environment.

Word length is restricted by two factors: The cost of accessing
each bit and the cost of inefficient memory utilization if a signifi-
cant percentage of stored quantities do not require the entire word
length.

Memory type. Memory type may be either main memory or high-speed
register. Many modern computers contain a limited number of registers in which operations can be performed much more rapidly than in main memory. When, during execution of a program, several quantities are used repeatedly in the computation it may be possible to obtain very efficient execution by allowing the frequently-used operands to remain in high-speed registers. The number of registers should be optimal with respect to the number of operands which are used frequently in the same program, so that bits reserved for addressing registers will not be wasted.

We are not concerned with register memory because of its increased speed. Our real concern with high-speed registers vs. main memory is the effects on the instruction set when the decision is made to include high-speed registers in a computer for memory or for indexing. This seemingly simple decision has far-reaching effects on the design of an instruction set because it is necessary to include "complete" sets of separate register operations for each data type. It is also important to include operations to interface between registers and main memory. The IBM 360 illustrates the repercussions of the decision to include high-speed registers. The 360 has 16 "general registers" and 4 floating point registers. Many of the 360's arithmetic operations can be performed in three modes: register to register, main memory to main memory, and main memory to register.

The main arguments in favor of high-speed registers are increased efficiency of execution and limited bit requirements for addressing. The constraining factors are additional cost of registers and increased complexity of the instruction set.
Data Formats

The data formats of a computer are the interpretations that the processor can give to the bits of data fields in memory. The same string of bits may be interpreted differently, depending upon whether the processor is executing a fixed-point arithmetic operation, a floating-point operation, or a non-arithmetic operation. The choice of data formats depends largely on the types of information which must be stored in the computer. Most computers have a limited number of data formats, the standard one being fixed point numbers, floating point numbers, alphameric words, and logical fields. Some computers allow arithmetic fields of several sizes. Since representation of data is, in some sense, the core of every programming problem there is a gradual trend toward more flexible data structures in computers. Perlis [49] and Standish [65] have considered the problem of increasing the flexibility of data structures in programming languages. Their work points up the need for greater flexibility of data in computers. Lesser [38] has proposed one solution to the problem—a computer organization that separates the task of data access from operations on data. The efficiency with which a task can be performed on a given computer depends largely on the efficiency with which the algorithm and its data can be stored in the memory of a computer.

In the discussion of word size we pointed out that the size of data formats is closely related to word size in a fixed-word machine. The length of a number usually equals the word length of the computer.

When a new data format is added to a computer it must be accompanied by a "complete" set of operations. It may also be necessary to add transfer operations to convert the new data format to old for-
mats, and vice versa. These operations require additional hardware and additional space in the instruction format for operation codes.

Operations

The selection of operations for an instruction set is influenced to some extent by each of the other six design variables, the task environment and data formats being the most important factors. Some operations must be provided with each data format. Add and subtract are almost always provided for numerical formats. A logically complete set of operations must be provided for logical data if it is permitted. For completeness' sake it is often important to include transfer functions between some data types. It may never be necessary to translate numerical fields to logical fields, but transfers between arithmetic types are almost necessary. Once enough operations have been included to satisfy all completeness requirements it may be desirable to add more operations in order to increase the efficiency with which algorithms can be executed on the machine. There are many variations on the fundamental operations. For arithmetic operations for example, each basic operation such as add can be supplemented by add negative, add absolute, or add negative absolute. Although one or two logical operations can form a complete set, there are 16 logical operations. There are many variations of shift and rotate operations. The utility of every operation depends upon the frequency with which it will be used in the task environment. From a theoretical point of view only one operation is necessary. [10] From a practical point of view an operation should be included if its cost in hardware and instruction space is less than the ultimate cost of "programming around" its absence.
If the task environment requires an operation that is relatively expensive to perform with the operations already in the computer then the operation should be added to the instruction set. If the instruction sequence "multiply by minus one, add" occurs frequently in the task environment, then add negative should be added to the instruction set. The branch on index high and branch on index low instructions of the IBM 360 were obviously included because they represent the increment-and-test combination of instructions which appears in almost every program loop.

Perhaps the best way to select operations for a task environment is to search the environment for primitive and compound operations that occur frequently. This "subroutinizing" process can be performed on a detailed representation of algorithms in the task environment. A "subroutine" can be made an operation in the computer if it is frequently used and if its parameters fit into the instruction format. The latter condition is important. Many useful sequences of instructions cannot be operations because they require too many parameters. The sequence load-add-store, for example cannot be an operation in a single-address computer because it requires two distinct parameters—the load-store address and the add address.

The preceding discussion suggests two ways in which the selection of operations could be automated. One approach is to write algorithms of the task environment in a sub-machine-level language like RTL (Register Transfer Language).[17] The subroutinization procedure could then be used to select the most useful machine-level operations. A slightly different approach would be to select a set of operations which is minimal in the sense that all operations in the set are re-
quired by completeness considerations. These operations could then be used to express algorithms in the task environment; the subroutin- 
ization process can be applied to these algorithms.

Certain decisions about the memory of a computer may have some impact on the set of operations. In a variable word-length computer it may be necessary to include instructions for manipulating data-boundary bits. If high-speed registers are included it is necessary to include operations to transfer data to and from the registers and to perform operations on data in the registers. In a similar manner the addressing method affects the set of operations, since a single-address computer must have some means for transferring data from memory to the accumulator and back. This is frequently -- though not necessarily -- done by load and store operations. The load operation can be replaced by the sequence clear accumulator, add.

The number of operations in an instruction set is limited by hardware costs and the space available for operation codes in the instruction format. Operations are relatively inexpensive in some modern computers because they are implemented by micro-programming rather than by additional hardware. Moreover the number of operations can be doubled by the addition of one bit to the operation code. Hence the design of a computer is relatively insensitive to changes in the set of operations. On the other hand, the performance of the computer may depend largely on how well its operations are matched to the needs of the task environment.

Addressing

We will deal with the following aspects of addressing: number of
addresses per instruction, immediate addressing, indirect addressing, indexing, and augmented addressing. Addressing features may be determined at least in part by any of the six other design variables.

**Addresses per Instruction.** Unless augmented addressing is used the number of addresses per instruction is determined by memory size and the amount of space in the instruction format. An address field of at least $n$ bits is required to address $2^n$ memory locations. This means, for example, that it takes 16 bits to address 64,000 memory locations, 20 bits to address 1,000,000 locations. As Buchholz[10] emphasizes, the size of memories in modern computers, together with practical word sizes and the number of operations almost restricts instructions to the single-address format. This need not be the case, of course, for intermediate-sized computers.

The number of addresses per instruction can be an important factor in determining the efficiency with which algorithms can be expressed in the machine language. For example algebraic expressions can be evaluated very naturally by single address instructions. If two-address instructions are used the second address is wasted in many instructions; i.e. the information content of the instruction stream is low compared to a program with single-address instructions. On the other hand many data processing applications require a large number of different arithmetic totals. In such applications it is convenient to keep totals in main memory accumulators and to address both the desired accumulator and the detail data field in a two-address instruction. On a single address computer such applications are characterized by frequent occurrence of the load-operate-store sequence, which re-
in a low information rate in the instruction stream. It appears, then, that in certain cases the number of addresses per instruction should be determined by the task environment rather than by memory size and word size. If the required number of addresses does not fit into the instruction format then indexing or augmented addressing can be used to reduce the amount of addressing information which must be specified in each instruction. This is precisely the approach used in the IBM 360, which has the two-address format. Each main-memory address consists of a 4-bit base register address and a 12-bit displacement. The contents of the specified base register are added to the displacement to obtain the actual address. The objection to this method is that programs cannot have direct access to all memory locations at once. In order to access a segment of memory a programmer must initialize a base register to the lowest address in the desired segment.

**Indexing and Indirect Addressing.** Indexing and indirect addressing can be invaluable aids in programs which must compute the location of data fields within complex data structures. Indexing and single-level indirect addressing are approximately equivalent. Both can be very useful in operations like searching linked lists, table-lookup, array accessing, and de-blocking variable length records. All of these examples involve compound data structures in which components are identified by their location within the data structure rather than by their individual addresses. Multi-level indirect addressing makes it possible to compute addresses by means of very complex accessing functions.
Indexing requires space in the instruction format to specify which memory locations are to be added to the displacement portion of the address. Existing computers have only a limited number of memory locations which can be used in indexing. In the IBM 1401, for example, three main memory locations are reserved for indexing. Two bits of each address specify either no indexing, index 1, index 2, or index 3. Some computers have special registers for indexing. The limit on the number of index registers that can be used efficiently seems to be approximately 16 or 32. Even if only one index register is available it is necessary to include a bit in each indexable address to specify whether or not indexing occurs. If index registers are different from main memory or other registers already in the computer it is necessary to include special operations for index registers, including, in general; load, store, and add operations.

Single-level indirect addressing requires only one bit in each address to specify whether the memory reference is direct or indirect. Multi-level indirect addressing poses a different problem since each quantity in a chain of indirect addresses must contain a bit which is used only to specify whether the next memory access is direct or indirect. The cost of these extra bits must be weighed against the desirability in the task environment of multi-level indirect addressing.

Immediate Addressing. Many applications require the use of a large number of constant data fields whose values can be anticipated by the programmer. It is frequently necessary to initialize counters to pre-specified values, increment counters by constant quantities, etc. In applications of this sort immediate addressing can be a valuable
aid because it eliminates the need for a special memory location for each program constant. The cost of immediate addressing is the cost of a slightly different accessing mechanism and one extra bit in the instruction format to specify whether or not immediate addressing is in effect.

**Instruction Encoding.** Instruction encoding is the interpretation which the instruction processor assigns to bits of the instruction stream. The instruction formats must convey all information necessary to invoke the various capabilities of the processor. To some degree the encoding problem is a hardware design problem. One of the main objectives is to find an encoding scheme that permits efficient decoding at execution time. Some of the hardware considerations, however, are reflected onto the overall design of the instruction set. Two opposing factors must be considered for any encoding scheme: information rate and ease of decoding. In variable-word-length computers the information rate of the instruction stream is relatively high. In fixed-word-length machines, however, some care must be taken to prevent an inefficient instruction stream. In most fixed-word-length machines it is much easier to access words or half-words than arbitrary sets of characters or bits. For this reason it is desirable that every instruction occupy one word or one half-word. In some cases it may be necessary to use 3 half-words or 2 full words. It is sometimes necessary, moreover, to use more than one instruction length, since the information requirements of different instructions can vary greatly. A branch operation, for example requires only one address. In a two-address format much of the space in a branch instruction is
wasted if all instructions have the same length. In order to maximize the information rate it may be necessary to use several instruction lengths. The IBM 360 has three instruction lengths: half-word, full-word, and three half-words, corresponding to register-register, register-memory, and memory-memory instructions.

Another method for increasing the information rate of the instruction stream is to assign codes to combinations of values instead of giving each instruction parameter its own field. Suppose, for example, that one parameter of an instruction has 3 values and another, 5. If each parameter is assigned its own field, 5 bits are required. But if each of the 15 combinations of the two parameters is assigned a code, only 4 bits are needed. This approach has only limited application because it complicates the decoding process. From the point of view of efficient decoding each bit or set of bits should be associated with a single function of the processor.

This completes our discussion of selection and evaluation of instruction sets. This chapter has been a statement of the problem we wish to automate. In Chapter III we will use the parameters and relationships described in this chapter, together with the other relevant considerations, to synthesize an approach to automated design of instruction sets.
CHAPTER III

USING A COMPUTER TO DESIGN INSTRUCTION SETS

The purpose of this chapter is to analyze the problem of designing an instruction set design system and to synthesize the conceptual features of a solution. The demands on our system fall into two categories. In Chapters I and II we considered the specific problem of designing instruction sets. This analysis, however, leaves some questions unanswered. How, for example, should we deal with the fact that various design strategies apparently lead to different solutions? What interactions between design variables must our system be prepared to handle? To answer questions like these we must examine the general characteristics of design processes. In the first two sections of this chapter we will consider various models of design and the applicability of computers to design problems. In sections 3 and 4 we will consider the demands which our design system must meet and the constraints which must be satisfied by a solution. The final section of this chapter outlines a solution to the problem. The details of the solution are presented in Chapters IV and V.

THEORY OF DESIGN

In this section we will deal with two types of models: a model of the design process and a model of a solution to a design problem. The model of the design process appears in most design literature, although different authors emphasize different aspects of the model.
The model of a solution is due primarily to Alexander.[1]

In its most general form our model of the design process is the statement that design is the transformation of an abstract object into a specific object.[15] Design is a process of specification of a solution to a design problem. At the most abstract level the solution can be considered as an object which meets the needs of the design problem and optimally satisfies the design criteria. At the most specific level the solution is a physical object. Between these two extremes there may be many intermediate levels of specification. The progress of a solution from abstract to specific is not necessarily direct. It may involve backtracking to an earlier stage and re-specification of part of the solution. Each step of this process involves two sub-processes: 1) specification of concepts and 2) analysis.[40] If analysis of a proposed solution shows that it does not perform optimally or does not meet the constraints of the problem then the solution -- or part of the solution -- must be re-specified.

The observation that it is sometimes desirable to re-specify part of a solution is very important. In general the parts of a solution are very interdependent; changes in one part affect the performance of other parts in very complex ways. In order to cope with this aspect of a design problem the designer must have a deep understanding of the relationships between the parts of his proposed solution -- he must understand the structure of the problem. Mann [40] refers to this aspect of a design problem as "design problem hierarchy." Mann identifies three specific hierarchical levels: "system", "component", and "element." Alexander in Synthesis of
Form [1] deals primarily with the problem of identifying relationships between design parameters. His approach is to identify relatively independent subsets of design parameters that can be treated as design subproblems. Alexander has developed a program which reads information about pairwise relationships between design parameters and determines "clusters" of parameters that can be treated independently. This "hierarchical decomposition" process requires a complementary "hierarchical re-composition" process to guarantee unity of the solution. Berholtz and Bierstone [8] show how Alexander's program decomposes the problem of architectural design of houses.

We see that the designer must understand not only the structure of the design process but also the structure of his particular problem. Grason [25] emphasizes this point in his breakdown of the design process. He identifies three aspects of the design process:

1. Statement of the problem
2. Structure -- organizing the solution into manageable subgroups
3. Synthesis

It is possible to identify many more steps in the design process than we have indicated above. Two authors -- Schmit [58] and Asimow [4] -- provide us with more details. Schmit identifies six aspects of the design process:

1. Need
2. Criteria
3. Concept
4. Model
5. Analysis of the model
6. Evaluation

In Schmit's model of the design process need, concept, and model represent three increasingly specific levels of specification. The criteria are properties or performance requirements which the solution must have in addition to fulfilling the need. Analysis of the model is an attempt to predict the behavior of the solution. Evaluation is comparison of the predicted behavior to the required behavior as specified by the criteria. If the model fails to meet some design requirements then the designer must re-specify either part of the design concept or part of his model.

Asimow [4] gives a detailed flow-chart of the design process in terms of 25 individual steps. Asimow's chart is reproduced in fig. 2. It contains all of the steps that have been mentioned above and several steps that the other authors have not mentioned. It also allows for "design problem hierarchy" by assigning separate steps to the treatment of subsystems, components, and parts.

Other authors, rather than modeling the entire design process, have focused on particular aspects of design. Eastman [21] describes the study of design protocols as an aid to modeling the methods of human designers. He is concerned largely with the manner in which people determine and apply relevant constraints and the manner in which people select "alternatives" for design parameters. Grason [25] deals with the effects of the strategy in the design of houses. We have already pointed out the importance of relationships between design parameters. Assuming that design is a serial process, we will call any scheme for treating the parameters of a design problem a design strategy. In general, different design strategies result in different
solutions. Grason has described a two-part system consisting of a strategy selection program and a strategy implementation program. His purpose is to see how changes in strategy affect the solution to the design problem. Grason points out that a design strategy involves the manner in which dependencies between variables are treated, as well as the sequence in which variables are specified. He identifies six methods for varying design strategies:

1. Sequence switching
2. Independent parallelization
3. Dependent parallelization
4. Local looping—redo one design activity
5. Pure looping
6. Melioration—redo a combination of design activities

The purpose of this section has been to present the various models of design processes. We can see that in most respects the models described above are compatible. They differ primarily in the amount of detail they contain, Asimow's model being specified in greatest detail. For our purposes the importance of these models lies in their emphasis on structure and the specification-analysis process. In the next section we will see how the models described above influence the design of an automated design system.

**REQUIREMENTS OF A COMPUTER-BASED DESIGN SYSTEM**

Different phrases are used in computer and design literature to distinguish between various types of design systems which make use of computers. The term **computer-aided design** is frequently used. However, to many people "computer-aided" connotes a man-computer inter-
active system in which the computer is an aid to a human designer. There are other ways to make significant use of computers in a design process. In many cases computers are used in the latter design stages for model simulation. In other cases it may be possible to perform large portions of a design procedure without human interaction. Schmit [58] uses the phrase "automated optimal design" to refer to a system which performs specification, analysis, evaluation, and re-specification. We will use the phrase computer-based design to refer to any design method that relies heavily on a computer.

We will consider four fundamental requirements of a computer-based design system:

1. Internal representation
2. Input/Output -- Information Storage and Retrieval
3. Computation
4. Strategy -- decision making capability

Representation

Design is essentially a process of specification of objects. It may be possible to use computers in some steps of a design process without representing objects in memory. However, if a computer is to help specify objects or perform detailed analyses it must contain information about proposed solutions of the design problem. From our previous discussion of structure we see that a representation for a partially specified object must store information about parts of the solution and relationships between the parts. A data-link list is a useful device for representing this type of information in the memory of a computer. Another useful device, which differs only in implementation, is the plex developed by Ross. [55] [56] A plex is com-
posed of \textit{n-component elements} each of which contains a variable number of data-link cell pairs. A link may point to another n-component element or to a plex. An n-component element or a linked list can store all pertinent attributes of a \textit{part} of a solution, as well as pointers to related parts and information about relationships. Grason [25] uses a similar representation for houses but with a theoretical twist. He uses the nodes of a directed graph to represent rooms of a house. The edges of the directed graph represent spatial relationships between the rooms of a house. This representation has the advantage that various theorems of graph theory apply to the dual of the original graph to give results on the existence of feasible solutions to design problem.

In general the selection of a representation depends heavily on the form of the particular objects the designer wishes to represent. Grason's results, for example, show how for specific problems it may be possible to gain great advantage by careful selection of a representation.

\textbf{Input/Output - Information Storage and Retrieval}

Like any programming system, a computer-based design system must provide an input/output capability. There are two general types of information that must be input to a design system: 1) constraints and 2) partially specified objects. It is very important that the language used to specify inputs be close to the designer's language for describing his problem. [40] Since many design problems can be described graphically much emphasis has been placed on graphic input/output for computer-based design systems. Sutherland's \textit{Stetchpad} [67]
was the first system to show the value of graphic communication in a
design system. Sketchpad permits a user to draw objects, by means
of a light pen, on a photo-sensitized screen. The user may also specify
constraints on parts of an object and relationships between parts.
Sketchpad finds an object that satisfies the constraints, if possible,
and displays the object on the screen. Designers using this approach
have been able to observe the dynamic behavior of partially specified
objects such as complex mechanical linkages.\[14][41] Much of the effort
toward useful graphic input/output systems has been motivated by the
needs of computer-based design systems. \[66] [64] [23] [31]

In some cases it may be more useful to use descriptive language
for input rather than graphic input. Ledley [37] shows how infix
operator notation can be used to construct a syntactic description
of a house. A similar approach can be used for many objects. Ross and
Rodriquez [56] deal with the problems of language in a general design
system. Their primary purpose is to describe a method for translating
descriptions of partially specified objects into plex structures in
the memory of a computer.

Input and output of constraints and objects are only half of the
communication problem in a design system. Once a partially specified
object is stored in memory it is necessary to access information about
the object in order to analyze and evaluate the object. It is also
necessary to alter and add parts as an object is re-specified. Hence
a design system must include information storage and retrieval mech-
anisms. The storage and retrieval needs of a system are determined
by the objects to be stored and the data structure used to store them.
The information retrieval language of a design system should permit
a user to store and retrieve parts of a solution naturally and with a minimal amount of specification. Users must also be able to retrieve information about constraints and relationships between the parts of an object. In a system with graphic displays the information retrieval operations display objects on a screen. In a closed system, however, the purpose of the information operations is to make parts of an object available to programs which perform analysis and evaluation.

In the following section we will discuss the computational needs of a computer-based design system. In the present section on input/output we should note, however, that a design system must have some language for specifying computations and communicating results either to the user or to other computations. The computational language must be matched to the computations that are performed on objects. It should also be integrated with the information storage and retrieval language. Ideally the information retrieval language should be embedded in the computational language in such a way that the information required to access a part of an object can appear as a single operand whenever the part is involved in a computation.

**Computation**

Automated design systems will release designers from many of the busy-work computations that must be performed during the design process. A computation on a given part of a solution, for example, may have to be re-done every time the part is re-specified. In most design problems there are computations that must be performed many times. These computations can be built into the design system as subroutines or operators. This approach is particularly useful when
the system is intended for the design of a limited class of closely related problems such as the design of houses or buildings. In a more general design system it may not be possible to anticipate the useful computations. In such a case the system should contain a mechanism which allows users to define their own computations and invoke them whenever they are required.

Another type of computation which is useful in design processes is simulation. According is Asimow [4] one of the primary roles of computers in design should be to simulate models of solutions. The problems involved in simulating a solution are so unique that several programming languages have been developed for the sole purpose of performing computer simulations.

Strategy

Strategy -- the order of treating variables and the manner of dealing with their interactions -- is an important part of every design process. Design can be thought of as a sequence of alternating steps of specification and analysis, or, as Mann and Coons [41] state it, "short bursts of high creative activity followed by longer periods of systematic analysis, followed by more peaks of decision." One of the important questions in the design of a design system is "How much decision power, if any should the system have?" Many authors feel that humans should decide which concepts and alternative values to use in a solution and that computers should perform analysis. [41] [64] [40] [14] [15] [56] They argue that man, because of his experience and his social viewpoint is better able to make appropriate design decisions, whereas computers are well-suited to the "busy-work" of analysis. This type of argument is the motivation for the man-machine
interactive design systems. Ross and Rodriguez [56] argue that no closed system can meet all the requirements of a design system. This may be true in most cases. It should not be argued, however, that computers should not perform some of the "creative" design activities. It seems doubtful that computers could perform the very high level design steps of needs analysis and selection of the design concepts, but at the lower levels of specification it has already been demonstrated that computers can perform some of the "creative" work of a human designer. In Schmit's automated optimal design approach, for example, the computer performs both analysis and re-specification. Eastman [21] suggests that the computer should contain the relative parameters and their alternatives. He also proposes a design system that "learns" and improves its strategy by remembering previous experience. Grason's proposed system contains a program which determines the strategy most suited to solving the given design problem.

Most of the attempts at automated-creative-design have an applied artificial intelligence flavor. The contemporary design systems being used in production are interactive systems. Jacks [31] discusses the difference between the two approaches as follows:

"It could be argued that for each uncertainty a program could be written to analyze the situation, then man would not be needed to aid the process. The strong point of man-machine communication via graphic consoles, however, is that for any given problem one may ask which parts are easily solved by the computer and which are best solved by man. This results in programs being written which have decision points in them at which the man at the console can be asked for advice."

Jacks' argument for the interactive approach can also be construed as an argument for getting computers to handle as much "creativity" as possible. One of the results of the research reported in this
thesis should be some insight as to how much design creativity can be expected of a computer.

In the following two sections we will combine the analysis of Chapters I and II with the first two sections of Chapter III to formulate a precise statement of our problem. Asimow [4] identifies needs analysis and activity analysis as important stages in the formulation of the problem statement. Needs analysis identifies the major requirements of a solution. Activity analysis is a "black-box" study of the "boundary conditions" which a solution must satisfy.

NEEDS ANALYSIS

The primary need which we established in Chapter II is the need for better matching of instruction sets to particular task environments. A closely related need is the need for better understanding of instruction sets and how their utility is affected by incremental changes. In order to meet these needs we must develop a system which satisfies the following requirements:

1. The system should provide a rapid method for synthesizing instruction sets so that the specification-analysis loop can be closed.

2. The system should deal adequately with the complexities of the design problem. In particular it should provide a means for dealing with different design strategies.

ACTIVITY ANALYSIS

In this section we will proceed with our statement of the problem by imagining a solution as it functions in the appropriate environment.
Our purpose is to define the desired outputs of the system, the inputs, constraints, and criteria for goodness.[4]

Output

The output from our system is to be a computer instruction set. In particular we want to specify:

1. Memory
   a. Size
   b. Structure
2. Data formats
3. Operations
4. Addressing methods
5. Instruction encoding

Inputs

Physical Inputs. Clearly our system will require computer time and storage.

Human Resources. A certain amount of human time will be required for the preparation of data and strategy.

Information. The system must be given information about the task environment, design strategy, and constraints on the desired instruction set.

Economic. The economic inputs to the system are the costs of computer time and preparation of input information.

Constraints on Inputs and Outputs
One constraint on the output from our system is that it must comply to some degree with the current notion of an instruction set. We also require output that completely describes the behavior of the instruction set and can be easily understood by humans.

The human time and effort required to prepare inputs should be minimized. The input languages should be natural for human use. The time required to prepare information about task environment and constraints should be on the order of minutes or hours. The limit on time required for preparation of strategy information is more difficult to assess. Intuitively input preparation time should be small compared to the generality of the strategy.

At the outset of the project the computers available were the IBM 360/67 and the CDC G-21. The only language available for a project of this nature on the 360 was *I, a fairly primitive linked-list language. [45] Languages available on the G-21 were IPL-V [44] and Formula Algol. [50] The normal mode of operation of the 360 was card input with two or three turns per day. The normal mode on the G-21 was teletype input with a run-time limit of three minutes per turn and a possibility for twenty, or thirty turns per day. IPL-V on the G-21 also provides a save-for-restart feature which permits continuity between separate runs from teletype. The G-21 provides space for 8,000 IPL-V cells. IPL-V was chosen as the base language because of the three languages available, it provides the most appropriate data structure and operations.

Environmental Constraints

We have already stated that our system should be capable of
closed-loop design. This means that it must provide some means for
analysis and evaluation. The only evaluation methods available are
the methods described in Chapter II under Evaluation of Instruction
Sets.

Criteria of Goodness

There are two important measures for our system: 1) "realness"
of the instruction sets it produces and 2) generality. Clearly
the system has a practical value only if it can produce useful in-
struction sets. The second measurement -- generality -- is a central
issue in this thesis. One of our purposes is to investigate the
feasibility of a closed design system as opposed to an interactive
system. We must show that a closed system can design all instruction
sets of practical value and that the overhead costs of anticipating
all cases are less, in the long run, than the cost of human effort in
an interactive system. The generality of the system is approximately
the ease with which the system can be made to design instruction sets
in all regions of computer space. If we can find a simple way to use
the system to generate a great variety of useful instruction sets then
the system has a great deal of generality. On the other hand if it
turns out to be fairly difficult to design instruction sets in just
one or two regions of computer space we will be inclined to say that
the system lacks generality and that the interactive approach is to
be favored.

OUTLINE OF A SOLUTION

In this section we will propose a solution to the problem of
automating the design of instruction sets and show how our proposed solution meets the needs and constraints which we established in the preceding sections of this chapter. The details of the solution are described in the remaining chapters of the thesis.

Our system (ISDS) has three parts:

1. A model of the concept of an instruction set.
2. An internal representation for storing instances of the model in the memory of a computer.
3. Two sets of IPL-V subroutines:
   a. Operators for storing and retrieving parts of an instruction set
   b. Operators for performing the tasks that occur frequently in the design of an instruction set.

The operators can be used to specify parts of an instruction set. The second category of operators can then be used, together with the information retrieval routines, to analyze the partially specified instruction set. Control for the design process is provided by the usual methods of IPL-V. IPL-V instructions decide, on the basis of the results of analysis, which parts should be specified and how they should be specified. Control then passes to the necessary specification routines. A strategy, then, is an IPL-V program in which the functions of information storage and retrieval and utility operators are performed by system subroutines. A strategy can use any desirable form of input information. Output is provided by a system subroutine which prints an instruction set, using English names to identify parts, in approximately the same format used in programming manuals. The semantic
content of the output is obtained from the model of an instruction set.

Let us see how the proposed solution meets the needs stated in the needs analysis. The system allows the user complete flexibility with regard to the amount of task environment information used by a strategy. Apparently the system can also be used to study effects of incremental changes by applying a strategy to two similar partially-specified instruction sets. Clearly the control provided by a strategy can close the specification-analysis design loop, provided of course that a satisfactory method for analysis can be found. Finally, in the needs analysis we required that a system should deal with the complexities of the design problem and the effects of strategy. It should be clear that our system is sufficiently flexible to allow any design strategy. On the other hand, the system isolates the functions of control, analysis, and specification, so that any attempt to impose more structure on the system would build some strategy into the system. The organization of ISDS is isomorphic to the organization of an interactive system, the difference being that control and specification are performed by a program rather than by a man. The information and utility routines could be used without change in an interactive system for designing instruction sets.

We will now see how our proposed solution meets the needs which were revealed by the activity analysis.

The output of the system will be tailored explicitly to the requirements stated in the activity analysis. Except for information about the memory, the output will be organized by instruction group, showing the addressing, formats, operations, and encoding for each
group. No specific provision is made for inputting constraints and task environment information. The system provides strategy information by means of an IPL-V program which determines the sequence in which variables are specified and the manner in which dependencies are handled.

The "realness" of instruction sets produced by the system depends upon the accuracy of the model. The model must reflect all characteristics of existing instruction sets. It should be simple, yet inclusive; and if our system is to innovate, the model must contain useful instruction sets which have not been invented. The semantics of the model must be precisely defined so that instances of the model are unambiguous. (Note, however, that ambiguity is not undesirable so long as the user has a satisfactory interpretation of his instance of the model.)

The language used to specify strategies will be relatively efficient. Much of the work will be done by system routines. Most of the IPL-V instructions will merely provide input parameters to the system routines. Some instructions will be required to perform computations and to determine the flow of control in the strategy. Special analysis and evaluation routines are also written in IPL-V with the aid of the system routines. IPL-V has two distinct advantages as a base language for strategies:

1. It provides a data structure close to the data structures required by the problem.

2. It provides natural mechanisms for stacking, or saving parts of a partially specified object.

It is difficult to predict to what extent the constraints on
human, physical, and economic resources will be met by our proposed solution. There is no reason to suspect a bottleneck, however, or a particular constraint that cannot be met satisfactorily.

Whether or not our proposed system can "close the design loop" depends upon the availability of adequate analysis and evaluation strategies. One possible approach to analysis is simulation. It is possible to implement a simulator for our model that would simulate any instance of the model. This would be an improvement over existing methods since one simulator would suffice for all test models. The disadvantage of this approach is that tasks must be re-programmed for every new instruction set. The most hopeful approach to closed loop design seems to be to use existing evaluation techniques to develop heuristics for measuring the value of an instruction set with respect to a given task environment.

Our criteria for goodness are "realness" and "generality." Presumably the model of an instruction set can be made sufficiently real and general. The remaining question -- whether or not sufficiently general design strategies can be written -- can be answered best by experimentation with the system. In the domain of general-purpose computers especially, it seems likely that a relatively simple strategy can account for most existing machines.

We will now describe the details of our solution to the instruction set design problem. Chapter IV describes GIS -- the generalized model of an instruction set. Chapter V describes the internal representation for GIS and the information processing and utility routines of ISDS.
The purpose of this Chapter is to describe a model of the concept instruction set. In keeping with the objectives stated in Chapter I we are interested primarily in modeling the functional capabilities of computers. In order to satisfy the needs of our design system a model should meet the following requirements:

1. Generality -- it should be possible to obtain almost every existing instruction as a special case of the model.

2. Simplicity -- Many computers contain very ad hoc instructions. Whenever possible our model should contain the concepts behind these instructions. The alternative -- to include instructions explicitly in the model -- should be used sparingly.

3. Isomorphism -- The elements of the model should correspond to the elements of computer instructions. The model should provide a syntactic description of computer instructions. For purposes of completeness, however, the model should also have a semantic interpretation.

Because of the wide range of techniques for implementing instructions it is not possible to find a simple model that captures the semantic details of a large number of real instruction sets. It would be difficult, for example, to find two computers with add
instructions similar in every detail. Add instructions vary in such details as operand length, operand format (position of the algebraic sign, for example), round-off, truncation, and overflow. Even at the syntactic level it is difficult to model all instructions. Input/output features and special hardware features are given only minor consideration in our model because of the great variation even in the language used to express these functions. For definitional purposes we will attach semantics to our model, but the semantics can be altered as required when the model is used to represent instruction sets in our design system. The primary purpose of the model is to describe the syntax of instructions, GIS is a grammar for machine languages.

GIS (Generalized Instruction Set) is based on 13 computers representing all of the important types of instruction sets. (See [7], [11], [16], [20], [28], [29], [30], [34], [42], [47], [48], [51]; [52],)

DESCRIPTION OF GIS

The language used to describe GIS is similar to the Backus Normal Form representation of a language with the exception that elements of a concatenation may be used but are not required. That is< A > < B > in our description would normally be written< A >< B > |< A > |< B > |∅ . In the case when A and B are integers we will assume that the concatenation is uniquely reversible. We can assume, for example, that a space remains between the two integers.
Computer

Syntax.

<computer> ::= <memory> <instruction length> <byte size> <word size> <order code>

<memory> ::= "integer"

<instruction length> ::= "integer"

<byte size> ::= "integer"

<word size> ::= "integer"

<order code> ::= <instruction group> | <order code> <instruction group>

Semantics. A <computer> is the most general object which can be modelled by GIS. A <computer> consists of an <order code> and four attributes of the memory structure -- memory size, word size, byte size and instruction length. By the convention which we established above, none of these attributes are required. In some cases not all of the attributes are meaningful. Word size, for example, is omitted from a description of a variable-word-length machine. An <order code> is a list of instructions for the computer.

Instruction Group

Syntax.

<instruction group> ::= <simple instruction> | <instruction group> <simple instruction>

Semantics. An <instruction group> corresponds to one instruction format for a computer. An instruction format can be thought of as a concatenation of variables which has meaning (to the instruction pro-
processor) for any assignment of legal values to the variables. A <simple instruction> is the functional unit of the computer's language. A simple instruction causes the processor to obtain the specified operands and perform one operation such as add, move, etc. Most computer instructions are simple instructions. Some more complex instructions, including micro-instructions, are modelled by a sequence of simple instructions.

Simple Instruction

Syntax.

<simple instruction> ::= <operation> <left operand part> <right operand part> <result part> <condition part> <if next-instruction part> <else next instruction part>

(In the syntax for <operation> , L stands for left operand, R for right operand.)

<operation> ::= absolute value | neg absolute value | + | - | * | \n  compare | shift | move | rotate | execute | f | t | l | \n  ~L | R | \n  | | V | \n  \phi | (empty) | normalize | tally
  no-operation | convert | complement | load | store | halt | input/output

<left operand part> ::= <address>

<right operand part> ::= <address>

<result part> ::= <address>

<if next-instruction part> ::= <address>

<condition part> ::= "any bit string"

<else next-instruction part> ::= <address>

Semantics. The <simple instruction> is the operational unit of an instruction set. The instruction processor interprets a <simple instruction> as follows:

1. The operands associated with the <left operand part> and
the <right operand part> are obtained by means of the Address Interpretation Algorithm (AIA). (See discussion under "address.")

2. The <operation> is performed on the operands and the result is placed in the memory location obtained by applying the Address Interpretation Algorithm to the <result part>. Various processor internal conditions may be set to preserve features of the result.

3. The <condition part> identifies an internal state of the instruction processor. If the processor is in the specified state then the AIA is applied to the <if next-instruction part> and the <simple instruction> at that memory location is presented to the processor to be executed next.

4. If the processor is not in the state specified by the <condition>, the AIA is applied to the <else next-instruction part> and the result is taken as the location of the next instruction to be executed.

Address

Syntax.

<address> ::= <switch> <displacement> <index list> <interpretation>

<switch> ::= on off

<index list> ::= <index> | <index list> <index>

Semantics. When the processor interprets a <left operand part> or a <right operand part> it expects to find an operand. When it interprets a <result part>, an <if next-instruction part>, or an <else next-instruction part> it expects to find an address. Since the
procedures for obtaining operands and addresses are almost identical, the processor is required to have only one address interpretation algorithm (AIA). The result of the algorithm, however, may be treated as an operand or as an address depending upon which type of quantity the processor expects to find. The AIA is described in figure 3. The method of the algorithm is to obtain the value of a data reference from the <index list> and the <displacement>. The <interpretation> specifies additional information about the data reference. If the data reference is indirect the algorithm calls itself recursively until a direct or immediate data reference stops the recursion. At this point the value of the address has been obtained. If the processor requires an address it uses the value obtained by the AIA. If the processor requires an operand it refers to the <interpretation> most recently used by the AIA. If the <mode> of the <interpretation> is immediate, the value becomes the operand. If the <mode> is direct the processor obtains the operand from the memory location specified by the value.

The <index> 's may be used either for relative addressing or for augmented addressing. This means that the values of the <index> 's may be combined with the <displacement> under any combination of subtract, add, or concatenate operations.

Displacement

Syntax,

<displacement> ::= <integer> <adjustment>

<adjustment> ::= <switch> <when> <add or subtract><address>

<when> ::= before|after
ADDRESS INTERPRETATION ALGORITHM

Figure 3
\texttt{<add or subtract> ::= add|subtract}

\textbf{Semantics.} If a \texttt{<displacement>} contains no \texttt{<adjustment>} then the value of the \texttt{<displacement>} is the \texttt{<integer>}. An \texttt{<adjustment>} with \texttt{<switch> off} has no effect on a \texttt{<displacement>}. An \texttt{<adjustment>} with \texttt{<switch> on} has the following effect.

If the \texttt{<when>} is \texttt{before}:

The value of the \texttt{<adjustment>}'s \texttt{<address>} is added or subtracted (as specified by the \texttt{<add or subtract>}) to the \texttt{<integer>} of the \texttt{<displacement>}, and the result \texttt{replaces} the \texttt{<integer>} in memory. The result is taken as the value of the \texttt{<displacement>}. 

If the \texttt{<when>} is \texttt{after}:

The value of the \texttt{<displacement>} is the value of its \texttt{<integer>}. The value of the \texttt{<adjustment>}'s \texttt{<address>} is added or subtracted to the \texttt{<integer>} of the \texttt{<displacement>} and the result replaces the \texttt{<integer>}. The next time the \texttt{<integer>} is referenced it will have the "adjusted" value.

\textbf{Index}

\textbf{Syntax.}
\texttt{<index> ::= <switch> <combine> <address> <adjustment>}
\texttt{<combine> ::= <add or subtract> | concatenate}

\textbf{Semantics.} If the \texttt{<switch>} of an \texttt{<index>} is \texttt{off}, the value of the \texttt{<index>} is undefined. If the \texttt{<switch>} is \texttt{on} then the value of the \texttt{<address>} is obtained. If the value of the \texttt{<address>} is a quantity in memory (rather than an immediate quantity) then the quantity is affected by the \texttt{<adjustment>} in the same manner that the \texttt{<integer>} of a \texttt{<displacement>} is affected by the \texttt{<adjustment>} of the \texttt{<displacement>}.
An <index> always occurs in a list of indexes (which may, however, contain only one index). The value of each <index> is added, subtracted or concatenated with the partial value of a data reference. The initial value of a data reference is the value of the <displacement>.

**Interpretation**

**Syntax.**

<interpretation> ::= <storage type> <reference> <field length> <data type> <mode>

<storage type> ::= register | main memory | auxiliary memory | temporary memory

<reference> ::= high order | low order

<field length> ::= <integer>

<data type> ::= arithmetic fixed | arithmetic floating | logical | byte

<mode> ::= immediate | direct | indirect

**Semantics.** The value of an <address> can be used in many ways by the instruction processor to determine an operand for an instruction. The <interpretation> of an <address> tells the processor how it should use the value of the address to obtain an operand.

The <storage type> of an <interpretation> distinguishes between the different logical or physical types of memory.

When a data reference identifies a single position of an operand field a <reference> tells whether the remainder of the operand field occupies addresses above (low order) or below (high order) the data reference.

The <field length> of an <interpretation> gives the number of memory units in the operand field. A memory unit is the smallest
addressable unit of the memory type specified by the memory type of the interpretation.

The <data type> tells the instruction processor how to interpret the bits of an operand field.

The <mode> specifies which of three methods is used to obtain an operand from the data reference of an <address>. If the <mode> is immediate the data reference is used as the operand. If the <mode> is direct the data reference is the memory location of the operand. If the <mode> is indirect the data reference specifies a memory location which contains an <address>.

THE USE OF GIS TO REPRESENT INSTRUCTIONS

GIS is a grammar for a very general machine language. Every instance of a <simple instruction> can be thought of as an instruction in a "GIS machine." By restricting the domain of each GIS variable we can use GIS to represent many different instruction sets. In Chapter V we present an internal representation for GIS which allows a more compact representation of an instruction set because it permits us to indicate the required GIS variables and their domains instead of enumerating all instructions. This representation describes instruction formats, rather than instructions. It also contains encoding information; i.e. it shows which bits of an instruction format are assigned to each variable.

In the remainder of this section we will show how GIS represents the different types of instructions used in modern computers.

The most important part of GIS is the <simple instruction>. It models the information requirements of almost every computer instruction.
However no instruction explicitly contains all parts of a simple instruction. For practical reasons some of the variables are assigned implicit values. A three-address instruction, for example, may have the GIS format

<operation> <left operand part> <right operand> <if next-instruction part>

The <result part> field is implicitly equal to the <left operand part> location or the <right operand part> location. The condition has implicit value unconditional and the <else next-instruction> address has <switch> value off.

The two-address format has the GIS representation

<operation> <left operand part> <right operand part>

The <result part> is implicitly the location of the <left operand part> or the <right operand part>. The value of the <if next-instruction part> is unconditionally the contents of the instruction register (the next instruction in memory). The <switch> of the <else next-instruction part> is off.

The one-address format has the representation

<operation> <right operand part>

The interpretation of the one-address format is similar to the interpretation of the two-address format with the added implication that the <left operand part> refers to the accumulator. In a stack machine only the <operation> appears and the <left operand part> and the <right operand part> refer to the first and second elements in the stack. The <result part> refers to the stack top.

To illustrate the use of GIS to represent instructions we will consider the GIS representation of the structure of the RX instruc-
tion format of the IBM System/360. RX instructions have two operands—one contained in a general register and one contained in main memory. The main memory address is relative to two registers—an index register X2 and a base register B2.

\[
\text{RX} :: = \text{operation} \quad \text{left operand part} \quad \text{right operand part}
\]

\[
\text{left operand part} :: = \text{displacement} \quad \text{interpretation}
\]

\[
\text{displacement} :: = \text{integer}
\]

\[
\text{right operand part} :: = \text{index} \quad \text{index} \quad \text{displacement} \quad \text{interpretation}
\]

\[
\text{index} :: = \text{displacement} \quad \text{interpretation = register}
\]

The primary purpose of GIS is to define a relatively inclusive portion of "computer space" as the domain of an automated design system. It is highly desirable, ultimately, that the domain of the system be open, or extendable. One consideration in the design of the internal representation for GIS is the desirability of extending the syntax. If a simple, homogenous data structure can be used to represent GIS then the syntax can be extended or altered in any way useful in a design procedure, providing, of course, that the procedure does not rely on the original interpretation.
INTRODUCTION

This chapter describes a data structure for GIS and a system of operators that can be used to write instruction set design strategies. The base language for the system is IPL-V [44] (Information Processing Language). The main disadvantage of embedding our system in an existing programming language is that strategy writers must know two programming systems. The strategy writer must know IPL-V in order to write the control instructions between design procedures, and he must know the operators in our design system. This is a price that has to be paid, however, in any design system that makes strategy-level decisions.

One of the goals in the design of this system has been to minimize the amount of IPL knowledge required to use the system. (We distinguish here between a user and a strategy-writer. A user presents data to a strategy.) It is possible to write strategies that accept easily constructed symbolic input and provide human-readable output. The system has also been designed in such a way that it could be adapted to an interactive environment. In order to understand the function of the system routines the user need know only the method by which IPL-V subroutines obtain their parameters and return their results. IPL-V is a stack machine. It has one accumulator, H0, which is a pushdown stack. Every operator in the system described in their Chapter obtains its input parameters from H0 and returns its outputs (if any) to H0.
The operators consume their inputs; i.e. the inputs are removed from HO. Each input or output is denoted by its position in HO. The top element is denoted by (0), the second by (1), the third by (2), etc.

A DATA STRUCTURE FOR GIS

Requirements of a Data Structure

A data structure for GIS should represent an instruction set in terms of its instruction groups. An enumeration of individual instructions would require too much storage space to be useful in practice. The data structure should represent each instruction group in terms of variables, the domain of values for each variable, the location within the format of each variable, etc. For example, an arithmetic instruction group in a single address machine might contain an operation code, an address, and a bit which specifies whether operations are fixed point or floating point. The data structure should represent this instruction group in terms of three variables -- operation code variable, data type variable, and right operand address variable. The operation code variable should specify the operations allowed for the group, the code for each operation, and the location within the instruction format of the operation code. The data type variable should specify the two data types -- fixed point and floating point -- the bit code for each type (0 or 1), and the bit of the instruction format used for data type. The right operand variable should specify the number of bits required to encode the right operand and their location in the instruction format. This example illustrates the following general requirements of an efficient data structure for GIS:
1. The data structure must be capable of representing each object which can appear in an instruction. It should also provide for the various useful means of specifying the domain of a variable. In some cases it may be desirable to enumerate the values which a variable can assume. It is also possible, however, to represent variables by an explicit value or by the number of bits they require in the instruction format.

2. The data structure should preserve the structure of GIS. It must be recursively defined in order to represent an address. In general the data structure should represent each object in terms of its component parts as defined by GIS.

3. The data structure should distinguish between implicit and explicit use of an object. It may be desirable to represent the details of an object that has an implicit value -- the accumulator of a single-address machine, for example.

4. The data structure should be as simple and homogenous as possible. It would be possible to design an elaborate data structure for each object in GIS. The effort required for information processing with such a complex structure would be prohibitive. If a simple data structure capable of representing every object in GIS can be found then the number of information processing routines required and the amount of run time spent on information storage and retrieval can be minimized. The simpler data structure is also desirable.
from the user's point of view because it requires less knowledge of the system and because it provides more generality and flexibility in data representation.

The Form Variable

In this section we will describe the form variable -- a data structure for GIS.

\[
<\text{form variable}> :: = <\text{description list}> <\text{form}>
\]

\[
<\text{description list}> :: = <\text{attribute}> <\text{value}> | <\text{description list}> <\text{attribute}> <\text{value}>
\]

\[
<\text{attribute}> :: = '\text{IPL symbol}'
\]

\[
<\text{value}> :: = '\text{IPL symbol}'
\]

\[
<\text{form}> : <\text{attribute}> <\text{item variable}> | <\text{form}> <\text{attribute}> <\text{item variable}>
\]

\[
<\text{item variable}> :: = <\text{description list}> <\text{item value}>
\]

\[
<\text{item value}> :: = '\text{IPL symbol}' | <\text{list}> | <\text{form variable}> | <\text{item variable list}>
\]

\[
<\text{list}> :: = <\text{description list}> '\text{IPL symbol}' | <\text{list}> '\text{IPL symbol}'
\]

\[
<\text{item variable list}> : = <\text{description list}> <\text{item variable}> | <\text{item variable list}> <\text{item variable}>
\]

Any GIS object that has more than one part can be represented by a \(<\text{form variable}>\). The \(<\text{form}>\) contains an \(<\text{attribute}>\) item variable pair for each part. The attribute is an IPL symbol which represents the name of the part. The item variable contains the part in its \(<\text{value}>\). The description list of the \(<\text{item variable}>\) may contain additional information about the \(<\text{item value}>\). The \(<\text{description list}>\) may be used to indicate which type of \(<\text{item value}>\) is contained in an \(<\text{item variable}>\). The item variable \(<\text{description list}>\) may also distinguish between explicit and implicit mention of variables. The
structure of a form variable is illustrated in figure 4.

The four types of <item values> which may appear in an <item variable> give the form variable the required ability to represent instruction variables. The domain of a variable can be defined by a list of values, an explicit IPL symbol, or the number of bits required to encode the variable. An <item variable list> is used to represent an <index list>; each <index> is represented by an <item variable>.

The <form variable> satisfies our requirement of homogeneity. It provides a simple, compact, internal representation for GIS. Since <item value> may be a <form variable> the <form variable> has the recursive power required to represent GIS.

System Symbols

All symbols in an IPL program must satisfy a restrictive set of syntactic rules. The IPL interpreter does not accept mnemonic symbols. Every attribute or value which can appear in GIS must have an associated IPL symbol. Similarly every attribute or value which can be used to specify a descriptive type or a specification type (implicit or explicit) must have a translation in the language of IPL. Figures 5 and 6 give the translations for the symbols of GIS and the additional symbols required by the design system described in this thesis. The set of symbols is open in the sense that any legal IPL symbol can be used in a design strategy. The input and output routines described in the section Design Procedures translate symbolic input to IPL symbols, and vice versa. Initially these routines are prepared to translate only the symbols which appear in
FORM VARIABLE

NAME—DESCRIPTION LIST

ATTRIBUTE

ITEM VARIABLE—DESCRIPTION LIST

VALUE

ATTRIBUTE

ITEM VARIABLE—DESCRIPTION LIST

VALUE

ATTRIBUTE

ITEM VARIABLE—DESCRIPTION LIST

VALUE

Figure 4
figures 5 and 6. The translation tables for the input and output routines, however, can easily be extended for new symbols. The translation tables and the procedure for extending them are included with the description of the input and output routines.

The GIS attribute and value symbols correspond to parts of GIS. The system attribute and value symbols are used in form variable or item variable description lists to specify additional information about a GIS part. The value of the instruction length attribute is an IPL integer. The attribute description type may have the value field description (an integer representing the number of bits assigned to a GIS part), form variable, integer, list or symbol. The attribute specification type may have the value explicit or implicit. The old value list and initial value attributes are used by the item variable and value preserve and restore routines B31-B39. The constraint type attribute may have the value NO, N1, N2, N3, or N4. These symbols can be used to specify the nature of constraints. They may be associated with numeric constraints or with constraint lists.

SYSTEM ROUTINES

The operators described in this section were chosen to meet the following needs:

1. The structure of the form variable demands certain primitive operators for storing and retrieving item variables and item values and for handling item description lists.

2. The recursive nature of GIS, especially the definition of address, requires some operators that store and retrieve
### ATTRIBUTE SYMBOLS FOR GIS

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>A11</td>
</tr>
<tr>
<td>adjustment</td>
<td>A12</td>
</tr>
<tr>
<td>byte size</td>
<td>A13</td>
</tr>
<tr>
<td>combine</td>
<td>A10</td>
</tr>
<tr>
<td>condition</td>
<td>A14</td>
</tr>
<tr>
<td>data type</td>
<td>A15</td>
</tr>
<tr>
<td>displacement</td>
<td>A16</td>
</tr>
<tr>
<td>else</td>
<td>A17</td>
</tr>
<tr>
<td>if</td>
<td>A19</td>
</tr>
<tr>
<td>index</td>
<td>A35</td>
</tr>
<tr>
<td>index list</td>
<td>A20</td>
</tr>
<tr>
<td>integer</td>
<td>A21</td>
</tr>
<tr>
<td>left operand part</td>
<td>A22</td>
</tr>
<tr>
<td>memory size</td>
<td>A23</td>
</tr>
<tr>
<td>interpretation</td>
<td>A24</td>
</tr>
<tr>
<td>mode</td>
<td>A36</td>
</tr>
<tr>
<td>operation</td>
<td>A25</td>
</tr>
<tr>
<td>order code</td>
<td>A26</td>
</tr>
<tr>
<td>reference</td>
<td>A27</td>
</tr>
<tr>
<td>result</td>
<td>A28</td>
</tr>
<tr>
<td>right operand part</td>
<td>A29</td>
</tr>
<tr>
<td>storage type</td>
<td>A30</td>
</tr>
<tr>
<td>switch</td>
<td>A31</td>
</tr>
<tr>
<td>when</td>
<td>A32</td>
</tr>
<tr>
<td>word size</td>
<td>A33</td>
</tr>
</tbody>
</table>

### VALUE SYMBOLS FOR GIS

#### Operation Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>V3</td>
</tr>
<tr>
<td>-</td>
<td>V4</td>
</tr>
<tr>
<td>*</td>
<td>V5</td>
</tr>
<tr>
<td>÷</td>
<td>V6</td>
</tr>
<tr>
<td>abs</td>
<td>V1</td>
</tr>
<tr>
<td>-abs</td>
<td>V2</td>
</tr>
<tr>
<td>compare</td>
<td>V7</td>
</tr>
<tr>
<td>shift</td>
<td>V8</td>
</tr>
<tr>
<td>move</td>
<td>V9</td>
</tr>
<tr>
<td>execute</td>
<td>V10</td>
</tr>
<tr>
<td>f</td>
<td>V11</td>
</tr>
<tr>
<td>t</td>
<td>V12</td>
</tr>
<tr>
<td>↓</td>
<td>V13</td>
</tr>
<tr>
<td>↑</td>
<td>V14</td>
</tr>
<tr>
<td>~L</td>
<td>V15</td>
</tr>
</tbody>
</table>

#### Figure 5

![Image of Figure 5]
<table>
<thead>
<tr>
<th>Storage type Symbols</th>
<th>Data type Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>auxiliary V56</td>
<td>fixed point V60</td>
</tr>
<tr>
<td>main V57</td>
<td>floating point V61</td>
</tr>
<tr>
<td>register V58</td>
<td>logical V62</td>
</tr>
<tr>
<td>temporary V59</td>
<td>byte (char) V63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mode Symbols</th>
<th>Other GIS Value Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>immediate V72</td>
<td>on V50</td>
</tr>
<tr>
<td>direct V73</td>
<td>off V51</td>
</tr>
<tr>
<td>indirect V74</td>
<td>before V52</td>
</tr>
<tr>
<td></td>
<td>after V53</td>
</tr>
<tr>
<td></td>
<td>high order V54</td>
</tr>
<tr>
<td></td>
<td>low order V55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System attribute Symbols</th>
<th>System Value Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction length A34</td>
<td>explicit V64</td>
</tr>
<tr>
<td>description type A50</td>
<td>field description V65</td>
</tr>
<tr>
<td>specification type A52</td>
<td>form variable V66</td>
</tr>
<tr>
<td>old value list A54</td>
<td>implicit V67</td>
</tr>
<tr>
<td>initial value A55</td>
<td>integer V68</td>
</tr>
<tr>
<td>constraint type A60</td>
<td>list V69</td>
</tr>
<tr>
<td></td>
<td>symbol V71</td>
</tr>
</tbody>
</table>

Figure 6
data through a chain of form variables.

3. It is often necessary in a design process to change parts of a partially specified object. Parts may be altered, deleted, or restored to previous values. The system should contain preserve, restore, and delete operators for item and item variables.

4. Certain primitive arithmetic operations such as count items, integer logarithm, and exponentiation are needed frequently in the design of an instruction set.

5. Some high-level operators are common to most instruction set design strategies. These operators include counting bits for an instruction group, output, and adding parts to instructions.

The information storage and retrieval routines, together with the preserve, restore, and delete routines, should provide a user with a natural "notebook" system for recording the progress of a partially specified instruction set. The user should be able to specify an operation on any part of a partially specified instruction set with at most three or four parameters.

In the following sections the operators of ISDS are organized hierarchically starting with the primitive form-variable operators and proceeding to the higher level design procedures. Figure 7 illustrates the organization of ISDS operators. Within each class operators are organized according to the data structures they affect.

Form-Variable Primitives

The form-variable primitive operators are the basis of the
HIERARCHY OF ROUTINES AND DATA IN ISDS

search routine

- basic strategy
  - form-variable routines
  - form-variable structures
  - GIS

- operators and evaluation routine
  - strategy-level utility routines
  - utility routines
    - IPL-V symbols and integers

Figure 7
information processing portion of the design system. They provide a "complete" set of one-level form variables. The primitives are "complete" in the sense that they provide the needed operations for the four parts of a form variable -- the description list, the item description list, the item variable, and the value. The form-variable chain operators described in the following section use the form-variable primitives to perform multi-level operations. The primitive operators are designed to meet the needs of the form-variable data structure. They are totally independent of GIS and any other specific type of information which can be stored in form variables. They can be thought of as a self-contained form-variable programming system.

**Description List Operators.** The name of a form variable or item variable is the name of the "head cell" of an IPL list structure. The head cell contains the name of the description list for the structure. Hence the IPL description list operators J10 - J16 can be used to process form variable and item variable description lists. It is sometimes useful, however, to process an item variable description list without first obtaining the name of the item variable. Operators B3, B5, B11 and B12 provide this facility.

**B3 FIND THE VALUE OF ATTRIBUTE (0) OF ATTRIBUTE (1) OF (2).** The item variable for attribute (1) of form-variable (2) is located. If attribute (1) does not occur on form-variable (2), output (0) is J3 and B3 exits with the IPL cell H5 set to -. If the item variable is located, B3 uses J10 to find the value of attribute (0) on the item variable description list. If attribute (0) does not occur on the description list, output (0) is J4 and H5 is set -. If J10 finds a value output (0) is the value and H5 is set +.

**B5 ASSIGN (1) TO ATTRIBUTE (0) OF ATTRIBUTE (2) OF (3).** The item variable for attribute (2) of form-variable (3) is located.
If attribute (2) does not occur on form-variable (3) an item variable is created and added to (3) as the item variable of attribute (2). The value (1) is assigned to attribute (0) on the item variable description list by the IPL primitive J11.

**B11 ERASE ATTRIBUTE (0) OF ATTRIBUTE (1) OF (2).** If attribute (1) has an item variable on form variable (2) attribute (0) on the item variable description list is erased by means of the IPL primitive J14. If attribute (1) does not occur on (2), B11 exits with H5-.

**B12 ERASE ALL ATTRIBUTES OF ATTRIBUTE OF ATTRIBUTE (0) ON (1).** Identical to B11 except that J15 is used instead of J14.

**Item Operators.** The item operators perform operations on items (values) of a form-variable. The value to be affected is uniquely identified by the name of a form variable and the attribute symbol for the item. Unless specified otherwise the item operators exit with no output and H5- if a required attribute does not occur on the form variable.

**B2 FIND THE VALUE OF ATTRIBUTE (0) OF (1).** The item variable of attribute (0) is located on form-variable (1). The value of the item variable is placed in H0 as output (0). If attribute (0) does not occur on form-variable (2), there is no output and H5 is set -.

**B5 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF (2).** If attribute (0) occurs on form-variable (2), input (1) becomes the value of the item variable. If attribute (0) does not occur on the form variable then an item variable is created and assigned to attribute (0) and (1) becomes the value of the new item variable. Input (1) may be the name of an IPL data term, an IPL symbol, the name of a list, or the name of a form-variable.

**B6 ADD (1) AT THE FRONT OF THE VALUE LIST OF ATTRIBUTE (0) OF (2).** If attribute (0) occurs on form-variable (2), input (1) is added to the value -- which is assumed to be the name of a list -- by J64. If attribute (0) does not occur on the form-variable an item variable is created with a list as its value and input (1) as the first symbol on the list.

**B7 ADD (1) AT THE END OF THE VALUE LIST OF ATTRIBUTE (0) OF (2).** Identical to B6 except that J65 is used instead of J64.
B9 **TEST IF (1) IS ON THE VALUE LIST OF ATTRIBUTE (0) OF (2).**
If attribute (0) occurs on form-variable (2) its value -- assumed to be a list -- is obtained and tested by J77 for an occurrence of the symbol (1). If attribute (0) does not occur H5 is set -.

B10 **DELETE (1) FROM THE VALUE LIST OF ATTRIBUTE (0) OF (2).**
If attribute (0) occurs on form-variable (2) input (1) is deleted from the value list by J69. If attribute (0) does not occur H5 is set -.

B31 **PRESERVE ATTRIBUTE (0) OF (1).** The cell containing the value of attribute (0) on form-variable (1) is preserved by the IPL instruction 41HO. The value of attribute (0) can be replaced by the new value and restored later by B32. There is no limit to the number of times B31 can be applied to an item.

B32 **RESTORE ATTRIBUTE (0) OF (1).** The cell containing the value of attribute (0) of form-variable (1) is restored by the IPL instruction 31HO.

B35 **DELETE ATTRIBUTE (0) OF FORM VARIABLE (1).** The cell containing the attribute (0) the form of form-variable (1) and the following cell (which names the item variable) are returned to available space. Neither the item variable nor the value is affected.

B36 **SAVE INITIAL VALUE OF ATTRIBUTE (0) OF (1).** The value of attribute (0) of form variable (1) is assigned as the value of attribute A55 on the item variable description list of attribute (0).

B37 **RESTORE ATTRIBUTE (0) OF (1) TO INITIAL VALUE.** The value of attribute A55 on the item description list of attribute (0) of form-variable (1) is assigned as the value of attribute (0)

B41 **GET INITIAL VALUE OF ATTRIBUTE (0) OF (1).** The value of attribute A55 on the item description list of attribute (0) of form-variable (1) is placed in the IPL accumulator, HO.

**Item Variable Operators.** It is sometimes convenient to use an item variable (an item value and its descriptive information) more than once in a data structure. This requires routines for processing item variables. Some item variable routines are also useful when a form-variable is processed by generating its values to a subprocess. Since
the subprocess must distinguish between different types of values, it
is necessary to generate the item variables of the form-variable rather
than the values. (See B45 under Processing Form Variables.)

B44 CREATE ITEM VARIABLE. No inputs. Two cells are removed
from the available space list and linked together to form
an item variable. Output (0) is the name of the item
variable.

B30 GET VALUE OF ITEM VARIABLE (0). The value of item variable
(0) is placed (input) in HO.

B43 ASSIGN (0) TO ITEM VARIABLE (1). Input (0) becomes the value
of item variable (1). Input (0) must be an IPL symbol or the
name of a data term, list, or form-variable.

B29 FIND THE ITEM VARIABLE OF ATTRIBUTE (0) OF (1). Output (0)
is the name of the item variable of attribute (0) of form
variable (1).

B40 ASSIGN ITEM VARIABLE (1) AS THE VALUE OF ATTRIBUTE (0) OF (2).
Input (1) becomes the item variable of attribute (0) of form-
variable (2). If attribute (0) occurred previously on form-
variable (2) the old item variable is not erased.

B33 PRESERVE ITEM VARIABLE (0) OF (1). Preserved names of item
variables of attribute (0) of form variables (1) are stored
in an "old variable " list. The name of the "old variable"
list for attribute (0) is the value of attribute (0) on the
description list of form variable (1). If the "old variable"
list does not exist it is created by B33. The "old variable"
list is preserved and the name of the current item variable
of attribute (0) becomes the first symbol.

B34 RESTORE ITEM VARIABLE (0) OF (1). The top element of the
"old variable" list is assigned by B40 as the item variable
of attribute (0) on form variable (1). The "old variable"
list is restored.

B38 SAVE INITIAL ITEM VARIABLE OF ATTRIBUTE (0) OF (1). The
name of the item variable of attribute (0) of form-variable
(1) becomes the value of attribute (0) on the description
list of the "initial variable" list. The name of the "initial variable" list is the value of attribute A55 on the form
variable description list, If A55 has no value the "initial variable" list is created.

B39 RESTORE ATTRIBUTE (0) OF (1) TO INITIAL ITEM VARIABLE. The
name of the initial item variable of attribute (0) of form-
variable (1) is obtained from the "old variable" list and
assigned as the current item variable of attribute (0). The
previous item variable is not altered or erased.

B42 **GET INITIAL ITEM VARIABLE OF ATTRIBUTE (0) OF (1)**. Output (0) is the name of the initial item variable of attribute (0) of form variable (1).

**Processing form variables.**

B1 **CREATE FORM-VARIABLE**. An empty form variable is created of cells from the available space list. Output (0) is the name of the new form-variable.

B45 **GENERATE THE ITEM VARIABLES OF (1) FOR SUBPROCESS (0)**. The subprocess is executed for every item variable of the form variable. The name of the item variable is input (1) to the subprocess. Input (0) is the attribute of the item variable. (See the IPL-V manual for conventions on the use of generators.

B45 provides a convenient method for performing general processes, such as printing and searching, on form-variables. If different types of values are to be processed by different subroutines the attribute symbols to be processed by each subroutine can be stored on a list associated with the subroutine. A routine written in this manner can have considerable recursive power since for values of type form-variable the routine can call itself.

**Form Variable Chain Operators**

Unlike the form-variable primitives the form-variable chain operators make use of information about GIS. They make it possible to store and retrieve parts of an address by naming the address and the attribute of the part. All of the chain operators process two or three-level form-variable structures. The more complex operations such as preserve and restore are not provided for multi-level form-variable structures. All chain operators create the necessary intermediate form-variables if they do not already exist.
Multi-Level Value Operations

B13 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF THE DISPLACEMENT OF ADDRESS (2). If address (2) has a displacement, input (1) is assigned as the value of attribute (0) on the form-variable which represents the displacement. If address (2) does not have a displacement, a form variable is created and assigned to (2) as the value of attribute A16. Input (1) becomes the value of attribute (0) on the new form-variable.

B14 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF THE ADJUSTMENT OF THE DISPLACEMENT OF ADDRESS (2).

B15 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF THE INTERPRETATION OF ADDRESS (2).

B25 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF THE FORM-VARIABLE WHICH IS THE VALUE OF ATTRIBUTE (2) OF FORM-VARIABLE (3).

B16 FIND THE VALUE OF ATTRIBUTE (0) OF THE DISPLACEMENT OF ADDRESS (1).

B17 FIND THE VALUE OF ATTRIBUTE (0) OF THE ADJUSTMENT OF THE DISPLACEMENT OF ADDRESS (1).

B18 FIND THE VALUE OF ATTRIBUTE (0) OF THE INTERPRETATION OF ADDRESS (1).

B26 FIND THE VALUE OF ATTRIBUTE (0) OF THE FORM-VARIABLE WHICH IS THE VALUE OF ATTRIBUTE (1) OF FORM-VARIABLE (2).

Multi-Level description List Operations.

B19 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) ON THE ITEM VARIABLE DESCRIPTION LIST OF ATTRIBUTE (2) OF THE DISPLACEMENT OF ADDRESS (3).

B21 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) ON THE ITEM DESCRIPTION LIST OF ATTRIBUTE (2) OF THE ADJUSTMENT OF THE DISPLACEMENT OF ADDRESS (3).

B23 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) ON THE ITEM DESCRIPTION LIST OF ATTRIBUTE (2) OF THE INTERPRETATION OF ADDRESS (1).

B27 ASSIGN (1) AS THE VALUE OF ATTRIBUTE (0) OF THE ITEM DESCRIPTION LIST OF ATTRIBUTE (2) OF THE FORM-VARIABLE WHICH IS THE VALUE OF ATTRIBUTE (3) OF FORM-VARIABLE (4).

B20 FIND THE VALUE OF ATTRIBUTE (0) ON THE ITEM DESCRIPTION LIST OF ATTRIBUTE (1) OF THE DISPLACEMENT OF ADDRESS (2).
Utility Operators

The operators described in this section perform operations that are useful in any strategy for designing instruction sets.

R1  INTEGER LOGARITHM OF (0). Input (0) is the name of an integer data term. Output (0) is the smallest integer greater than or equal to the logarithm (base 2) of the input (0). Output (0), in effect, gives the number of bits required to encode (0) items.

R2  INTEGER LOGARITHM AND DIFFERENCE OF (0). Same as R1 except a second output (1) gives the difference between input (0) and 2 exp (output (0)).

R3  COUNT SYMBOLS ON (0). Input (0) is the name of a list. Output (0) is the number of symbols (not counting the head.cell) on the list (0).

R4  COUNT BITS TO ENCODE LIST (0). Input (0) is the name of a list. Output (0) is the number of bits required to encode the symbols on list (0).

R5  COUNT BITS TO ENCODE AND DIFFERENCE. Same as R4 except that a second output (1) gives the number of items that can be added to list (0) such that the items on list (0) can still be encoded by output (0) bits.

R6  2 EXP ((0)). Input (0) is the name of a non-negative integer data term. Output (0) is the name of an integer data term equal to 2 raised to the input (0) power. Output (0) is the number of items that can be encoded by (0) bits.

Design Procedures

The operators described in this section are tools for performing high-level tasks which are useful in the analysis and synthesis of
instruction sets. Almost any high-level design procedure must contain some restrictions and assumptions. Yet for experimental purposes it is desirable to have a set of operators for the important sub-tasks of a design process. The operators described here perform many of these tasks. Although they are based on some restrictions and assumptions they are general enough to permit a wide range of experimentation with design strategies. In the remainder of this section, each design procedure is described together with its restrictions and assumptions.

The design procedures permit the full generality of GIS. However the data structure is restricted in the sense that a specific data structure is associated with each GIS object. This requirement is not particularly restrictive. It simplifies the coding of the operators, reduces the space they require and allows them to run much faster than would be possible with the full generality of form-variable structures. The data structure for each part is shown in figure 8. When a part may have two representations the representation used is indicated in the description list of the item variable containing the part.

A <computer> is a form-variable having the name of the <order code> as one of its values. The <order code> is a list of instruction groups. Each instruction group is a list of <simple instructions>.

Computations. Most of the computations which are performed in the design of an instruction set involve counting bits in instruction formats. It is necessary, for example, to determine the length of each instruction and the number of bits required for the operation code. Many elaborate strategies can be used to encode operations.
### Data Structures for Design Procedures

<table>
<thead>
<tr>
<th>form-variable</th>
<th>values</th>
<th>value type</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td>word size</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>byte size</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>memory size</td>
<td>integer</td>
</tr>
<tr>
<td></td>
<td>order code</td>
<td>list of instruction groups</td>
</tr>
<tr>
<td>simple instruction</td>
<td>operations</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>left op part</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>right op part</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>condition</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>if next-instruction</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>result part</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>else next instruction</td>
<td>form-variable</td>
</tr>
<tr>
<td>address</td>
<td>switch</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>disp</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>index list</td>
<td>list of item variables</td>
</tr>
<tr>
<td></td>
<td>interpretation</td>
<td>form-variable</td>
</tr>
<tr>
<td>displacement</td>
<td>integer</td>
<td>integer, field description</td>
</tr>
<tr>
<td></td>
<td>when</td>
<td>integer, field description</td>
</tr>
<tr>
<td></td>
<td>add or subtract</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>form-variable</td>
</tr>
<tr>
<td>adjustment</td>
<td>switch</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>when</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>add or subtract</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>form-variable</td>
</tr>
<tr>
<td>index</td>
<td>switch</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>combine</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>address</td>
<td>form-variable</td>
</tr>
<tr>
<td></td>
<td>adjustment</td>
<td>form-variable</td>
</tr>
<tr>
<td>interpretation</td>
<td>storage type</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>reference</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>field length</td>
<td>integer, field description</td>
</tr>
<tr>
<td></td>
<td>data type</td>
<td>list, field description</td>
</tr>
<tr>
<td></td>
<td>type size</td>
<td>integer, field description</td>
</tr>
</tbody>
</table>

Figure 8
There is no a priori reason, except perhaps for ease in decoding, for assigning the same bits of every instruction to the operation code. It is always possible, for example, to have one instruction with a one-bit operation code. However, many general purpose computers use the same bits of every instruction format for operation code. This is the method used in the operators of this section.

From the point of view of efficient information transfer, it is desirable to encode combinations of values of instruction variables. From the point of view of efficient decoding, on the other hand, it is desirable to encode each instruction variable in a separate field of the instruction format. The latter practice is used most frequently. Operation codes, however, usually contain more information than the type of operation to be performed. The strategy of the operators presented here is to assign an operation code to each combination of operation, data type, and storage type. This means, for example, that add-register-register and add-memory-memory are assigned separate operation codes.

T21 TALLY OPERATION-DATA TYPE-STORAGE TYPE COMBINATION FOR COMPUTER (0). For each instruction group the following quantities are multiplied to form an operation total, for the group: number of operations, number of data types in each address part, number of storage types in each address part. Output (0) is the sum of the products for individual instruction groups.

T22 TALLY BITS TO ENCODE GROUP (0). Each part of instruction group (0) is treated as if it is to be assigned its own field in the instruction for; i.e., no combinations are encoded as in T21. The number of bits required to encode each part is determined according to its representation as follows:

<table>
<thead>
<tr>
<th>representation</th>
<th>bits required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
<tr>
<td>list</td>
<td>integer logarithm (base 2) of</td>
</tr>
</tbody>
</table>
representation

- field description
- form-variable
- list of item variables

bits required
- the number of items on the list
- value of the field description (an integer)
- sum of totals for individual values
- sum of totals for each value

Output (0) is the name of an integer data term containing the total number of bits required to encode the instruction group (0).

T23 **TALLY OPERATION, STORAGE, TYPE, AND DATA TYPE BITS OF INSTRUCTION GROUP (0)**. Output (0) is the number of bits required to encode the operations, storage types, and data types of instruction group (0). Output (0) is determined by the method of T22 except that only operations, storage types, and data types are considered.

T25 **TALLY NON-OPERATION CODE BITS FOR INSTRUCTION GROUP (0)**. T25 uses the method of T22 to determine the number of bits required to encode instruction group (0), except for operations, data types, and storage types. T25 calls T22 and T23. Output (0) is the difference between the results of T23 and T22.

The total number of bits required for an instruction group is equal to the sum of the integer logarithm (base 2) of the output of T21 and the output of T25. This number is generally less than the output of T22. The important operators above are T21 and T25. T22 and T23 are used in T25, but are unlikely to be useful by themselves.

T6 **COMPUTE THE NUMBER OF ADDRESSES PER INSTRUCTION FOR COMPUTER (0)**. Assumes that computer (0) contains a word size. (A variable-word computer can also have word boundaries). Computer (0) is partially specified to any extent. T6 applies T21 to determine the number of operation code bits required in each instruction format. Assuming that the most desirable size for each instruction is one word length, T6 subtracts the number of operation code bits from the word length and determines the number of displacements (one displacement = integer log₂ (memory size) that can be accommodated by the remainder of the word. Output (0) gives the number of addresses per instruction. Output (1) gives the number of unused bits which will remain in the word if one-word-length instructions are used.

T6 is used to obtain an estimate of the number of words per instruction for a computer. There are cases in which it would be
undesirable to use the result of T6 at face value to fix the number of addresses per instruction. For example if T6 is applied to a computer with one highly structured address it may specify a two-address machine. In some cases, however, a symmetric addressing structure may not fit into one word. T6 considers only displacement size in determining address sizes. Hence it must be used cautiously. Several iterations may be required to obtain a sufficiently powerful addressing structure.

T6 operates on the assumption that instruction power increases with the number of addresses per instruction. This seems to have been a consideration in the design of many general purpose computers. Although there are many other rationales for selecting addressing structures (see Chapter II), T6 permits a wide range of experimentation.

T7 CREATE A (1)-ADDRESS STRUCTURE FOR COMPUTER (O). Each instruction group of computer (O) is assumed to have a result part. Using the result part as a prototype, T7 creates (1)' (an integer data term equal to 0-5) addresses for the instruction group. The result part is deleted unless input (I) is greater than or equal to 3. Addresses are assigned according to (I) as follows:

<table>
<thead>
<tr>
<th>Input (I)</th>
<th>Addresses Assigned</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no addresses</td>
</tr>
<tr>
<td>1</td>
<td>right op</td>
</tr>
<tr>
<td>2</td>
<td>right op and left op</td>
</tr>
<tr>
<td>3</td>
<td>right op, left op, and result</td>
</tr>
<tr>
<td>4</td>
<td>right op, left op, result, if next-instruction part</td>
</tr>
<tr>
<td>5</td>
<td>right op, left op, result, if next-instruction part, else next-instruction part</td>
</tr>
</tbody>
</table>

T6 and T7 can be used in the early stages of a design strategy to construct the skeleton of an instruction set. Any unused bits in the instruction format can then be assigned by the remainder of
T52 GENERATE THE SIMPLE INSTRUCTIONS OF COMPUTER (1) FOR SUB-
PROCESS (0). Routine (0) is executed for each simple in-
struction of computer (1). (See [44] for conventions on
construction and use of generators.)

Input/Output. Many types of input data can be useful in a
design strategy. The input should specify requirements on the desired
instruction set and any constraints which must be satisfied. The
input routine described here translates a symbolic representation of
a partially specified <computer> into the internal format of an IPL
list structure. This means that requirements must be expressed in
terms of GIS. A limited amount of constraint information can be
included in the description lists of form variables and item variables.
The following input program is part of an input translation program
written by Professor A. Newell of Carnegie-Mellon University.

E20 TRANSLATE SYMBOLIC INPUT TO AN IPL LIST STRUCTURE. E20
translates symbolic statements in linear list notation to
their IPL internal equivalents. Input is in the form of
a sequence of assignment statements having the following
syntax:

<assignment> ::= <left side> <-<right side> ;

<left side> ::= <name>

<right side> ::= <symbolic form variable> | <symbolic item
variable>

<name> ::= <letter or digit> | <name> <letter or digit>

<letter or digit> ::= 1|2|...|9|A|B|...|Z

The symbolic form variable or item variable of the <right side>
is translated to an IPL list structure. E20 associates an IPL name
with the <name> of the left side and assigns the IPL name as the
internal name of the list structure.

A symbolic form variable has the following syntax:
To make input more readable a symbolic word is associated with each symbol of GIS and the internal representation. These "reserved attributes" and "reserved values" are illustrated in Table 1. E20 translates each reserved word into its associated IPL symbol. An IPL symbol preceded by " _0 " is transferred directly to the internal translation.

The attribute symbol for each value on a form-variable appears on a <form element list> followed by a <symbolic item variable> , the symbolic representation of the item variable containing the value of the attribute. A symbolic item variable has the following syntax:

<symbolic item variable> ::= <description> <symbolic value name>

<symbolic value> ::= <list name> | <integer name> | <form-variable name> | <list> <symbolic form variable>

$list name$ ::= <name> "IPL symbol"

<integer name> ::= "IPL data term"

<form variable name> ::= <name>

<list> ::= <symbol> | <list> <symbol>

<symbol> ::= "reserved value" "IPL symbol"
TABLE OF RESERVED WORDS

<table>
<thead>
<tr>
<th>Reserved Symbol</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>addsub</td>
<td>A10</td>
</tr>
<tr>
<td>adr</td>
<td>A11</td>
</tr>
<tr>
<td>adj</td>
<td>A12</td>
</tr>
<tr>
<td>byte</td>
<td>A13</td>
</tr>
<tr>
<td>cond</td>
<td>A14</td>
</tr>
<tr>
<td>data</td>
<td>A15</td>
</tr>
<tr>
<td>disp</td>
<td>A16</td>
</tr>
<tr>
<td>else</td>
<td>A17</td>
</tr>
<tr>
<td>length</td>
<td>A18</td>
</tr>
<tr>
<td>if</td>
<td>A19</td>
</tr>
<tr>
<td>index</td>
<td>A20</td>
</tr>
<tr>
<td>integer</td>
<td>A21</td>
</tr>
<tr>
<td>leftop</td>
<td>A22</td>
</tr>
<tr>
<td>mem</td>
<td>A23</td>
</tr>
<tr>
<td>access</td>
<td>A24</td>
</tr>
<tr>
<td>opr</td>
<td>A25</td>
</tr>
<tr>
<td>code</td>
<td>A26</td>
</tr>
<tr>
<td>ref</td>
<td>A27</td>
</tr>
<tr>
<td>result</td>
<td>A28</td>
</tr>
<tr>
<td>rightop</td>
<td>A29</td>
</tr>
<tr>
<td>memtyp</td>
<td>A30</td>
</tr>
<tr>
<td>switch</td>
<td>A31</td>
</tr>
<tr>
<td>when</td>
<td>A32</td>
</tr>
<tr>
<td>word</td>
<td>A33</td>
</tr>
<tr>
<td>abs</td>
<td>V1</td>
</tr>
<tr>
<td>negabs</td>
<td>V2</td>
</tr>
<tr>
<td>+</td>
<td>V3</td>
</tr>
<tr>
<td>-</td>
<td>V4</td>
</tr>
<tr>
<td>*</td>
<td>V5</td>
</tr>
<tr>
<td>/</td>
<td>V6</td>
</tr>
<tr>
<td>comp</td>
<td>V7</td>
</tr>
<tr>
<td>shift</td>
<td>V8</td>
</tr>
<tr>
<td>move</td>
<td>V9</td>
</tr>
<tr>
<td>exec</td>
<td>V10</td>
</tr>
<tr>
<td>F</td>
<td>V11</td>
</tr>
<tr>
<td>T</td>
<td>V12</td>
</tr>
<tr>
<td>L3</td>
<td>V13</td>
</tr>
<tr>
<td>L4</td>
<td>V14</td>
</tr>
<tr>
<td>NEG</td>
<td>V15</td>
</tr>
<tr>
<td>L6</td>
<td>V16</td>
</tr>
<tr>
<td>L7</td>
<td>V17</td>
</tr>
<tr>
<td>L8</td>
<td>V18</td>
</tr>
<tr>
<td>L9</td>
<td>V19</td>
</tr>
</tbody>
</table>
### TABLE OF RESERVED WORDS

<table>
<thead>
<tr>
<th>Reserved Symbol</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>V20</td>
</tr>
<tr>
<td>L11</td>
<td>V21</td>
</tr>
<tr>
<td>L12</td>
<td>V22</td>
</tr>
<tr>
<td>L13</td>
<td>V23</td>
</tr>
<tr>
<td>L14</td>
<td>V24</td>
</tr>
<tr>
<td>L15</td>
<td>V25</td>
</tr>
<tr>
<td>OR</td>
<td>V26</td>
</tr>
<tr>
<td>NCP</td>
<td>V27</td>
</tr>
<tr>
<td>NORMFL</td>
<td>V28</td>
</tr>
<tr>
<td>tally</td>
<td>V29</td>
</tr>
<tr>
<td>branch</td>
<td>V30</td>
</tr>
<tr>
<td>on</td>
<td>V30</td>
</tr>
<tr>
<td>off</td>
<td>V51</td>
</tr>
<tr>
<td>before</td>
<td>V52</td>
</tr>
<tr>
<td>after</td>
<td>V53</td>
</tr>
<tr>
<td>hi</td>
<td>V54</td>
</tr>
<tr>
<td>lo</td>
<td>V55</td>
</tr>
<tr>
<td>aux</td>
<td>V56</td>
</tr>
<tr>
<td>main</td>
<td>V57</td>
</tr>
<tr>
<td>reg</td>
<td>V58</td>
</tr>
<tr>
<td>temp</td>
<td>V59</td>
</tr>
<tr>
<td>fix</td>
<td>V60</td>
</tr>
<tr>
<td>float</td>
<td>V61</td>
</tr>
<tr>
<td>log</td>
<td>V62</td>
</tr>
<tr>
<td>char</td>
<td>V63</td>
</tr>
<tr>
<td>form</td>
<td>A50</td>
</tr>
<tr>
<td>reserv</td>
<td>A51</td>
</tr>
<tr>
<td>type</td>
<td>A52</td>
</tr>
<tr>
<td>bits</td>
<td>A53</td>
</tr>
<tr>
<td>exp</td>
<td>V64</td>
</tr>
<tr>
<td>desc</td>
<td>V65</td>
</tr>
<tr>
<td>var</td>
<td>V60</td>
</tr>
<tr>
<td>imp</td>
<td>V67</td>
</tr>
<tr>
<td>int</td>
<td>V68</td>
</tr>
<tr>
<td>list</td>
<td>V69</td>
</tr>
</tbody>
</table>

Table 1 (continued)
Description lists in form-variables and item variables may be used to specify which type of value is used and how constraints are to be interpreted.

The "quote" symbol preceding the name of a <symbolic item variable> or a <form variable name> causes E20 to mark the variable "local." The advantage of this is that the entire form-variable structure for the input computer conforms to the IPL definition of "list structure", so that the IPL routine J150 can be used to print the entire structure. This can be a valuable aid in debugging design strategies. All operators described in this chapter preserve the "list structureness" of the internal representation so that J150 is applicable at any stage of a design strategy.

T11 PRINT FORM-VARIABLE STRUCTURE (O). Each attribute of the form-variable (O) is printed, followed by its value. If the value is a form variable, T11 is called recursively to print the new form-variable. The form variable structure is assumed to observe the conventions described at the beginning of this section for representing parts of GIS.

T12 PRINT COMPUTER (O). Prints the memory size, word size, and byte size of computer (O) and then generates the simple instructions of computer (O) for T11.

Both T11 and T12 are included for debugging purposes. Their output is formatted to identify parts only by their attribute symbols.

T30 PRINT MANUAL FOR COMPUTER (O). Prints memory size, word size, and byte size; then prints the following information for each instruction group:
   1. Instruction length
   2. Names of participating addresses
   3. Bits assigned to operation code
   4. Operations for each data type - storage type combination, together with a binary operation code.
   5. Bits assigned to each remaining part, the "tree" name of the part (in terms of GIS names) and an encoding for each element if the part is a list.

T30 is applicable to any GIS representation of a computer which meets
the structural requirements described earlier in this Chapter and
the additional requirement that each instruction group must have
an instruction length assigned as the value of attribute A34 on
its description list. (See T26). T30 translates each IPL symbol
of computer (0) to a symbolic name, by means of a translation
table (See T70). The binary operation codes are obtained by trans-
lating decimal codes to binary notation. (See T32 and T33).

T70 TRANSLATE SYMBOL (0). If (0) has a value on the description
list of D90 then the translation of (0) is obtained from
the list named by the value and entered in the current print
line. If the line is full it is printed and cleared and
the translation is entered. If (0) has no translation on
D90 the symbol (0) is entered in the print line.

T33 CONVERT (0) TO ITS (1) BIT BINARY EQUIVALENT. Input (0)
names a positive decimal integer. Input (0) is converted
to its binary equivalent of the length given by input (1).
Input (1) is an integer data term. The bits of the binary
equivalent of input (0) are elements of the list output (0),
the first element representing the high order bit. Each s
ymbol on the list output (0) is either NO (for 0) or N1
(for 1).

T32 ENTER THE (1) BIT BINARY EQUIVALENT OF (0). Converts (0)
to its (1) bit binary equivalent using T33 and enters the
result in the current print position.

T26 SET INSTRUCTION LENGTH FOR EACH GROUP OF COMPUTER (0).
Each instruction group of computer (0) is treated separately.
If a group already has an instruction length (attribute A34)
it is ignored. Otherwise the instruction length becomes the
length of the smallest half-word-length multiple which can
accomodate the bits of the instruction group. (Word mul-
tiples are used in the case of an odd word length.) Bit
counting is done by T21 and T25.

Optimization. The operators described in the preceding section
are the basis of ISDS. They can be used to implement entire strategies
or ad hoc operators that affect only part of a computer. Given a
collection of operators it seems reasonable to ask what effect each
has on the performance of the instruction set and how different
sequences of operators affect performance. Evaluation of performance is very difficult. It seems likely, however, that there are heuristic evaluation techniques that can be used at least to obtain a deeper understanding of instruction set design. In this section we describe a mechanism which permits experimentation with various evaluation techniques and with various strategies.

T75  **GENERATE COMBINATIONS OF LENGTH (2) FROM LIST (1) FOR SUBPROCESS (0).** Input (1) is the name of a list of symbols. Input (2) is a positive integer data term. On each generation cycle a new combination of length (2) is formed of the symbols on list (1). The symbols of the combination are placed on a list which is output (0). Subprocess (0) is executed for every combination.

T76  **OPTIMIZE COMPUTER (1) WITH RESPECT TO EVALUATION ROUTINES (0) AND SEQUENCES OF LENGTH (3) FROM OPERATOR LIST (2).** Input (2) names a list whose symbols are the names of operators which conform to the following conventions:

1. The operator's only input is the name of a computer.

2. Before altering the input computer the operator tests whether or not it is applicable. If the operator cannot be applied it exits with H5-.

3. If an operator is applicable it preserves every part of the input computer which it will affect before it is applied. After application H5 is set +.

4. Associated with the operator is a restore routine which returns the input computer to its status prior to the call on the operator.

The name of the restore routine appears as the value of the operator name on the description list of (2). Sequences of operator names are generated to a "strategy executor", which executes each operator of the sequence, calls evaluation routine (0) whose only output is a numeric data term, and retains the strategy receiving the highest score. After a score is obtained for each strategy the restore routine for each operator is executed in order to return computer (0) to its original status. Operators are restored in reverse order of application. After all strategies have been scored the strategy receiving the highest score is applied permanently. The process continues until every operator of every sequence in inapplicable.
ISDS can be used in at least three ways to design instruction sets:

1. The operators can be used in an interactive system.
2. The operators can be embedded in IPL programs to form ad hoc instruction set design strategies.
3. The operators can be used to construct strategy-level operators to be applied by an executive routine which heuristically determines its own strategy.

In an interactive system ISDS could be used to perform bookkeeping and analysis. If the system contained a graphic display device the user could specify parts of an instruction set and use the operators of ISDS to perform the bit-counting computations required for analysis. An interactive system should allow the user to construct his own high-level operators for analysis or specification. The most desirable level of interaction between the user and ISDS is at the level of the user's needs. Ideally a user should be able to specify his requirements in simple language, perhaps via a graphic console. A powerful design strategy, written in ISDS, would then display, on the console, a solution to the design problem. If the user is not satisfied with the solution he re-specifies his requirements or alters some parameters in the strategy and requests a new solution. This process continues until the system specifies a solution that meets the user's needs and satisfies the necessary constraints.
The second method for using the system -- to implement ad hoc design strategies -- raises an issue of generality: "How many strategies would it take to design all possible instruction sets?" If the system is used to implement specific strategies then its usefulness depends largely on the range of computers generated by a given strategy. If one strategy, though restricted in domain, can design a relatively wide range of useful instruction sets then the system would fulfill a purpose. On the other hand, if each strategy can design only in a very limited domain, then the system would provide little advantage to a designer. To be most useful the system should accept a wide range of inputs and design a wide range of instruction sets. For this reason we will abandon the use of ad hoc strategies in favor of the third alternative -- a heuristic program that determines its own strategy.

The general idea of this approach is to use the operators of Chapter V to construct "strategy-level" operators. A "strategy-level" operator specifies one part of an instruction set. It performs as much analysis as possible in order to make an "intelligent" specification. The executive routine (T76; see Chapter V) executes sequences of strategy-level operators and evaluates the resulting partially specified instruction set. After evaluation the instruction set is restored and a new sequence of operators is applied. The sequence resulting in the highest score is applied permanently and the process continues until no operator is applicable. In addition to a total score for each partially-specified instruction set the evaluate routine also computes a cost for the instruction set. If the cost exceeds a pre-specified limit the current strategy is rejected. (Strategy here means "sequence of
operators.") To get the above process started an operator (T5) transforms the input into an instruction set with a primitive addressing structure. The strategy-level operators add as many features as possible to the basic instruction set.

The heuristic approach outlined above can be used to build a single system capable of designing any instruction set which can be represented in GIS. To do this the system must contain an operator to specify each part of GIS. Since there are different strategies and analysis techniques for specifying some values the system may require more than one operator for some parts of GIS. In general there must be an operator for every method of specifying every part of GIS. Such a general collection of operators, coupled with the search technique T76, can, given the appropriate evaluation strategy, design an instruction set which can be represented in GIS.

In the remainder of this chapter we present a restricted version of the system described above. The system described here is limited in the sense that only the most frequently used instruction set features are included and only one operator is provided for each feature. The system has four parts:

1. The basic strategy (T5).
2. Operators that perform analysis and add GIS features to an instruction set.
3. A polynomial-type evaluation routine with variable cost and value coefficients.
4. The executive routine (Optimize Computer -- T76; see p. 114)

In spite of the limited set of operators the program described here can design a wider range of instruction sets than could be expected
of a fixed strategy. Variability is provided by two features: 1) input to the program and 2) the values of the cost and value coefficients. Symbolic input is in the form described in Chapter V. The use of the cost and value coefficients is described later in this chapter. In the remainder of this Chapter we describe the three parts of the system -- the basic strategy, the operators, and the evaluation routine. The final section contains examples illustrating the use of the system.

THE BASIC STRATEGY -- T5

The heuristic program accepts two types of input: 1) values of parameters for the evaluation routine and 2) a partially-specified instruction set in the format of the input language described in Chapter V. The partially-specified instruction set must contain the following information:

1. Memory Size
2. Word Size
3. One simple instruction including at least one operation and a "result" address which specifies the data type.

The input may specify any GIS instruction set that satisfies the above requirements and represents GIS parts by the required data structure as shown in figure 8.

The purpose of T5 is to transform the initial partially-specified instruction set into an instruction set that satisfies the following fundamental constraints:

1. Each instruction must fit into one word length
2. It must be possible to address every memory location
4. A single-address machine must provide access to the accumulator (e.g., load and store).

If the input instruction set does not have a branch instruction, T5 adds one. Then T5 gives the instruction set a basic addressing structure as follows:

1. Compute the number of addresses per instruction using T6.
   (p. 106)

2. Assign addresses to each instruction group as follows:

   Number of addresses = 0
   
   Number of bits available for addresses > 12:
   
   Use segment size of 4096 characters, 4 bits for base registers in each address; as many addresses as possible

   Number of bits available ≤ 12:
   
   One address; largest possible segment size; augmented addressing

   Number of addresses = 1:
   
   One address per instruction; number of bits = \(\log_2\) (memory size).

   Number of addresses = 3 or more:
   
   Use half-word addresses; call T5 recursively (only once)

If T5 creates a single-address instruction set it adds load and store operations if they are not present in the input.

ADDING FEATURES TO THE BASIC INSTRUCTION SET

Once T5 has constructed an instruction set which meets the basic requirements listed above, the search routine, T76, "optimizes" the instruction set by a gradual process of specification and evaluation. Each specification operator adds one GIS part to the instruction set. In this section we present two sets of operators. The first set
contains operators which aid in the construction of the second set -- the routines that add GIS parts to an instruction set.

Utility Operators

T58 GENERATE AVAILABLE BITS AND SIMPLE INSTRUCTIONS OF COMPUTER (1) FOR SUBPROCESS (0). For each instruction group of computer (1) subprocess (0) is executed with the following inputs:

Input (0) = (word size) -- (number of bits required to encode the group -- determined by T25 and T21).

Input (1) = first simple instruction of the group.

T42 FIND THE SMALLEST NUMBER OF UNUSED BITS IN ANY INSTRUCTION GROUP OF COMPUTER (0). Uses T58 to find the number of unused bits for each group. Output (0) is the smallest number generated by T58.

T65 TEST IF EVERY INSTRUCTION GROUP OF COMPUTER (0) CAN BE ENCODED IN ONE WORD LENGTH. Compares the number of bits required to encode each instruction group of computer (0) to the word size of computer (0). If the number of bits exceeds the word size T65 exits with H5-. Otherwise H5 is set +.

T42 and T65 perform similar functions. T42 is used in operators in which analysis precedes specification. T65 is used in operators in which specification precedes analysis (generate and test).

T40 ADD A (1) BIT INDEX REGISTER SPECIFICATION TO EACH NON-REGISTER OPERAND OF COMPUTER (0). Input (1) is the name of an integer data term specifying the number of bits to be used in each operand for the new index specification. A (1) bit index specification is added to each operand (right op or left op) of computer (0) which is not of storage type register. T40 is used in T5 if base registers are required and in T68. (See p. 122.) If an address has an index list, then the new index specification is added at the end of the list. Otherwise T40 creates an index list with the new specification as the only symbol on the list.

T41 RESTORE COMPUTER (0) FROM T40. For each non-register right op and left op of computer (0) T40 removes the last index specification. If the index list contains more than one symbol T41 deletes the last symbol. If the index list contains only one symbol T41 erases the index item variable and deletes it from the operand.
T63  ADD ONE OPERATION FROM LIST (2) TO EACH SIMPLE INSTRUCTION FOR DATA TYPE (1) OF COMPUTER (0). For each simple instruction of computer (0) with a right operand of data type (1), T63 does the following:

1. Preserve the operation list
2. Assign a new copy of the operation list
3. Add to the new operation list the first symbol on list (2) which does not already appear on the operation list.

If every symbol on list (2) appears on every operation list of format (1) then no symbols are added and H5 is set -.
Otherwise symbols are added as described above and H5 is set +.

T64  RESTORE INSTRUCTIONS OF TYPE (1) OF COMPUTER (0) FROM T63. The operation list of each simple instruction of computer (0) with a right operand of data type (1) is restored.

Strategy Operators

Each operator described in this section adds one GIS part to a partially specified instruction set. Since the sequence of application of these operators is governed by T76 each operator must meet the requirements of T76; that is, each operator must preserve every affected part of the instruction set and must have an associated restore routine. If an operator is not applicable it sets H5-.

T56  ADD INDIRECT ADDRESSING TO COMPUTER (0). If any operand of computer (0) contains indirect addressing T56 exits with H5-. Otherwise T56 uses T42 to determine whether or not the word size of computer (0) can accommodate the additional bit or bits required for indirect addressing. If not exit with H5-. Otherwise add the indirect symbol to the mode list of every operand. If an operand does not have a mode list a new mode list is provided.

T53  RESTORE COMPUTER (0) FROM T56. Removes indirect addressing from the mode list of every operand of computer (0). If the mode list contains only two symbols (eg. direct, indirect) the mode list is deleted.

T43  ADD GENERAL REGISTERS TO COMPUTER (0). If the number of available bits as determined by T42 is not positive T43
exits with H5-. If any simple instruction of computer (0) has a left operand T43 exits with H5+. If any right operand of computer (0) has an index specification then the number of bits added for general registers must equal the number of bits in the index specification. If the word size cannot accommodate this number of bits (as determined by T42) T43 exits with H5-. Otherwise a register left operand is created for each simple instruction having a right operand. If computer (0) contains no index specification the number of bits in the left operand specification is equal to the number of available bits or to 4, whichever is smaller.

T60  RESTORE COMPUTER (0) FROM T43. Deletes and erases the left operand of every simple instruction of computer (0) having a right operand.

T61  ADD I/O OPERATIONS TO COMPUTER (0). If computer (0) has no operand of data type 'character' T61 exits with H5+. Otherwise T61 copies the 'character' operand and creates a new instruction group with a 'character' right operand and four dummy I/O operations.

T62  RESTORE COMPUTER (0) FROM T61. Erases and deletes the last instruction group of computer (0).

T66  ADD ONE LOGIC OPERATION TO COMPUTER (0). Uses T63 to add one operation from a list of logical operations to each operand of type 'logic'.

T67  RESTORE COMPUTER (0) FROM T66. Calls T64 for computer (0) with input (1) equal to 'logic'.

T53  ADD FIXED POINT ARITHMETIC OPERATIONS TO COMPUTER (0). Uses T63 to add one arithmetic operation to each operand of type 'fixed point' of computer (0).

T54  RESTORE COMPUTER (0) FROM T53. Calls T64 for computer (0) with input (1) equal to 'fixed point'.

T68  ADD INDEXING TO COMPUTER (0). If computer (0) has an index specification or a register operand specification the number of bits in the new index specification must equal the number of bits used previously to specify indexing or a register operand. If this number of bits can be added to each operand (determined by T42) then the addition is accomplished by T40. If not T68 exits with H5-. If computer (0) has no index specification or register operand then the number of bits added by T40 for the new index specification is equal to the smaller of 4 and the number of available bits divided by the number of addresses per instruction. (Integer division)
The restore routine for T68 is T41, the restore routine for T40.
EVALUATION ROUTINE -- T51

The evaluation routine associates a value and a cost with each feature of a partially specified instruction set and computes a total value and a total cost for the instruction set. The cost and value functions can be thought of as polynomials in which some of the variables are 0-1 boolean variables. A boolean variable has value 1 if the associated feature is present in the instruction set, 0 otherwise. The variables associated with addressing features are as follows:

1. Index variable -- the number of index items on the index list of each operand.
2. Address mode -- a 0-1 variable indicating the presence or absence of indirect addressing.
3. Left op variable -- a 0-1 variable indicating the presence or absence of a register left operand.
4. I/O variable -- a 0-1 variable indicating the presence or absence of an instruction group with I/O operations.
5. Operation variable -- the number of symbols on operation list for all simple instructions.

The coefficients of the evaluation polynomials are IPL integer data terms. They are set to zero initially but may be altered to any integer value when the system is loaded for execution. The same is true of the data term C20 which specifies the maximum allowable cost. The coefficients for addressing features are as follows:

<table>
<thead>
<tr>
<th>Feature Description</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost constraint</td>
<td>C20</td>
</tr>
<tr>
<td>index value</td>
<td>C21</td>
</tr>
<tr>
<td>index cost</td>
<td>C22</td>
</tr>
<tr>
<td>indirect address value</td>
<td>C23</td>
</tr>
<tr>
<td>indirect address cost</td>
<td>C24</td>
</tr>
</tbody>
</table>
T51 evaluates addressing features by multiplying each variable by its cost and value coefficients and adding the product to the total cost and total value, respectively. The value of a boolean variable is 0 or 1, depending upon whether or not the instruction set contains the feature associated with the variable. For example, if the instruction set contains indirect addressing the address mode variable is equal to 1 and the coefficients C23 and C24 are added to total value and total cost, respectively.

To evaluate operations for an instruction set T51 uses an instruction mix. Any mix may be used. The input format for the instruction mix is illustrated below:

```
C90 <- "data type" (value coefficient, cost coefficient, operation, ...
(value coefficient, cost coefficient, operation, ...
"data type" (value coefficient, cost coefficient, operation, ...
(value coefficient, cost coefficient, operation, ...
]().
```

The "data type" in a mix may be fix, float, log, or char. A data type may be used more than one time in a mix to place different weights on groups of operations for the same data type. The cost and value for each operation in the instruction set are determined by searching each list of the instruction mix for the appropriate data type and operation. If the operation is in the mix its value and cost are given
by the first two symbols on the list in which the operation occurs.
If an operation is not in the mix its cost and value are 0.

Input (0) to T51 is the name of a partially specified computer.
Output (0) is the total score for addressing features and operations for the instruction set. T51 also computes the total cost of the instruction set by means of the cost coefficients. If the total cost exceeds the maximum cost, C20, then H5 is set -. Otherwise H5 is set +.

T51 is a relatively simple evaluation strategy. There are at least three reasons for using the type of evaluation procedure used by T51:

1. Since our evaluation routine operates in a closed loop of a program we are forced to use a "feature extraction" approach. More reliable evaluation techniques such as simulation, benchmarks, kernels, etc. are impractical for our present purpose.

2. The method of T51 is similar to a frequently used evaluation technique -- the instruction mix approach. T51 is an extension of the instruction mix approach since it assigns values to addressing features as well as to types of instructions.

3. A design problem is frequently stated in terms of optimizing a performance function subject to cost constraints. Hence T51 provides a practical method for posing a problem to the design system.

The inputs to our program can be classified as follows:
1. Partially-Specified Instruction Set
2. Values of cost and value coefficients -- parameters of T51
3. Instruction Mix, including cost and value coefficients for each group of operations

The partially-specified instruction set specifies memory size, word size, and the data formats of the instruction set. It may specify additional features which are to occur in the completely specified instruction set. The coefficients of T51 and of the instruction mix specify the relative cost and value to the user of the addressing features and the various operations. Hence the coefficients of T51 have an important role in the specification of a problem to the design program. The user can "request" a particular feature by assigning a relatively large number to the value coefficients of the feature. The operators of ISDS will determine the appropriate value for the feature and add it to the instruction set, provided that the new partially-specified instruction set satisfies all constraints. Similarly the user can restrict the use of a feature by assigning a large number to the cost of the feature. If the cost of a feature exceeds the maximum cost, C20, the feature will not be added to the instruction set.

The cost and value coefficients of T51 and of the instruction mix can be used to experiment with different strategies. At each stage of the optimization T76 applies the operator which results in the greatest improvement in the partially-specified instruction set. It is possible to vary strategy by altering the relative values of the coefficients.

This concludes our description of the heuristic instruction set.
design program. The organization of the program is illustrated in figure 9. In the remainder of this Chapter we present 12 examples of instruction sets designed by the program.
Figure 9
EXAMPLES

The 12 examples presented here are organized into three groups as follows:

1. Examples 1-3:
The ISDS version of the Whirlwind I instruction set [70] and two variations obtained by adjusting the coefficients of the evaluation routine.

2. Examples 4-7:
The ISDS version of the IAS instruction set [13] and two variations. Examples 7 shows each step in the optimization of the instruction set of examples 6.

3. Examples 8-12:
Example 8 illustrates ISDS's first attempt to design the Univac 1108 instruction set. Examples 9 and 10 are variations of example 1 obtained by altering the symbolic input and the coefficients of T51. Examples 11 and 12 illustrate the procedure for adding new operators to ISDS inorder to improve ISDS's version of the 1108 instruction set. The instruction set designed by ISDS in example 12 contains all of the parts of the 1108 instruction format.

The computer printout for each example contains the following information:

1. The initial partially-specified instruction set in symbolic format.

2. Values of the cost and value coefficients.

3. The strategy adopted by ISDS together with the cumulative
cost and value at each stage.

4. The instruction set designed by ISDS.

The cost and value coefficients are loaded in the format of IPL-V integer data terms. The printed values show the values that were presented to the system at load-time. The coefficients C31-C48 are the cost and value coefficients of the instruction mix. The mix used in these examples is as follows:

\[
\begin{align*}
\text{C90} & \leftarrow \text{FIX}( (C31, C32, V3, V4, V5, V6, C51, C52) \\
& \quad \quad \quad \quad (C33, C34, V1, V2, V7, V29, C53, C54)) \\
& \text{FLOAT}( (C35, C36, V3, V4, V5, V6, C55, C56) \\
& \quad \quad \quad \quad (C37, C38, V7, V28, C57, C58)) \\
& \text{LOG}( (C39, C40, V11, V15, V20, V27, C59, C60) \\
& \quad \quad \quad \quad (C41, C42, V8, V12, V13, V14, V16, V17) \\
& \quad \quad \quad \quad (C43, C44, V18, V19, V21, V23, V24, V25, V26)) \\
& \text{CHAR}( (C45, C46, V10, V9, V7, V31, C61, C62)) \\
\end{align*}
\]

The symbols Vxx are GIS operation symbols. (See p. 76) The symbols C50-C62 are dummy operation symbols added to increase the number of operations that can be added to instruction sets designed by ISDS.

The printed output of each example shows the name of the operator applied at each step of the optimization. The operators which add addressing features are described on pp. 121-122. The program used to produce the examples described here contains one operator to add instructions from each list of the instruction mix. These operators, in the order that their associated operation lists appear in the above instruction mix, are T72, T73, T74, T82, T83, T77, T78, T79.

The instruction set in each example is printed in the format of T30. (See p. 112.) Dummy operations are printed in the form Cxx. The symbols L3, L4, L6, L7, L8, L9, L11, L12, L13, L14, L15 represent, respectively, the logic operations \( \downarrow, \downarrow, \downarrow, \downarrow, R, \uparrow, /, =, R, \supset, L, \subset \). (See p. 76.)
Each GIS part is identified in the printed output by the "tree name" of the part. The name "a*b*c" is read "the c of the b of the a."

Examples 1-3

This set of examples illustrates the use of ISDS to reconstruct a design strategy for the Whirlwind I instruction set. [70] The actual instruction set of Whirlwind I is presented, in the format of T30, in figure 10.

Whirlwind I is a single-address computer with a memory of 2,048 16-bit words. Each instruction contains a 5-bit operation code and an 11-bit address specification. The 5 operation code bits permit 32 operations. However, Whirlwind I has 34 operations because the sixth bit of the instruction is used in the encoding of the four shift operations. This is possible because the shift operations, since they apply to the accumulator, require no address.

Example 1. The printed output for example 1 is presented in figure 11. The symbolic input specifies the memory size (K32 = 65536), word size (N16 = 16) and byte size (N8 = 8). The first instruction group specifies add and subtract operations for fixed point arithmetic data. The second group specifies a move operation for character data. The last line of the symbolic input invokes the design program with the symbolic input "computer" as its argument.

In this example all of the cost coefficients are set to zero. The value coefficients of all addressing features are set to zero, so, that only operations and an I/O instruction group will be added during the optimization phase of the design strategy. Since only one 11-bit address specification can be used in the 16-bit word, T5 assigns
Whirlwind I

Memory Size = 2048 words
Word Size = 16 bits

***Instruction Group 1***

| op code | rt op |

---Operation Group 1---

right op: char;

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>select I/O</td>
<td>00000</td>
</tr>
<tr>
<td></td>
<td>00001</td>
</tr>
<tr>
<td>block transfer in</td>
<td>00010</td>
</tr>
<tr>
<td>read</td>
<td>00011</td>
</tr>
<tr>
<td>block transfer out</td>
<td>00100</td>
</tr>
<tr>
<td>record</td>
<td>00101</td>
</tr>
</tbody>
</table>

Bits 5 to 15
rt op*displacement*integer

***Instruction Group 2***

| op code | rt op |

---Operation Group 1---

right op: logic

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum digits (no carry)</td>
<td>00110</td>
</tr>
<tr>
<td>store</td>
<td>01000</td>
</tr>
<tr>
<td>store digits</td>
<td>01001</td>
</tr>
<tr>
<td>shift l. &amp; round (6 bit code)</td>
<td>110110</td>
</tr>
<tr>
<td>shift l. &amp; hold (6 bit code)</td>
<td>110111</td>
</tr>
<tr>
<td>shift r. &amp; round (6 bit code)</td>
<td>111000</td>
</tr>
<tr>
<td>shift r. &amp; hold (6 bit code)</td>
<td>111001</td>
</tr>
<tr>
<td>scale (6 bit code)</td>
<td>111000</td>
</tr>
<tr>
<td>cycle left &amp; clear (6 bit code)</td>
<td>111100</td>
</tr>
<tr>
<td>cycle l. &amp; hold</td>
<td>111101</td>
</tr>
</tbody>
</table>

Figure 10
<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply digits</td>
<td>1111</td>
</tr>
<tr>
<td>scale</td>
<td>11101</td>
</tr>
<tr>
<td>exchange</td>
<td>01101</td>
</tr>
</tbody>
</table>

Bits 5 to 15
right op*displace*ntger

***Instruction Group 3***

<table>
<thead>
<tr>
<th>op code</th>
<th>rt op</th>
</tr>
</thead>
</table>

Bits 0 to 4
op code

---Operation Group 1---
right op: fixed point

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>check</td>
<td>01011</td>
</tr>
<tr>
<td>add B reg to x</td>
<td>01100</td>
</tr>
<tr>
<td>clear and add</td>
<td>10000</td>
</tr>
<tr>
<td>clear and sub</td>
<td>10001</td>
</tr>
<tr>
<td>add</td>
<td>10010</td>
</tr>
<tr>
<td>subtract</td>
<td>10011</td>
</tr>
<tr>
<td>clear &amp; add magnitude</td>
<td>10100</td>
</tr>
<tr>
<td>special add</td>
<td>10101</td>
</tr>
<tr>
<td>tally</td>
<td>10110</td>
</tr>
<tr>
<td>difference of magnitudes</td>
<td>10111</td>
</tr>
<tr>
<td>multiply &amp; round</td>
<td>11000</td>
</tr>
<tr>
<td>multiply &amp; hold</td>
<td>11001</td>
</tr>
<tr>
<td>divide</td>
<td>11010</td>
</tr>
</tbody>
</table>

Bits 5 to 15
right op*displace*ntger

***Instruction Group 4***

<table>
<thead>
<tr>
<th>op code</th>
<th>rt op</th>
</tr>
</thead>
</table>

Bits 0 to 4
op code

---Operation Group 1---
right op: char

Figure 10 (continued)
### Figure 10 (continued)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>branch</td>
<td>01000</td>
</tr>
<tr>
<td>branch conditional</td>
<td>01110</td>
</tr>
<tr>
<td>branch (subroutine)</td>
<td>01111</td>
</tr>
</tbody>
</table>

Bits 5 to 15

\[ rt \text{ op*displacement*integer} \]
OUTPUT FOR EXAMPLE 1

COMPUTER=\((MEM (\times K32) CODE (\times ORDERS) WORD (\times N16) BYTE (\times N8))\)
ORDERS=\('GROUP1, 'GROUP2\);
GROUP1=\('I1\);
I1=\((OPR ((+, -)) RESULT ('ADR1))\);
ADR1=\((ACCESS ((DATA ((F\times 1))))))\);
GROUP2=\('I2\);
I2=\((OPR ((MOVE)) RESULT ('ADR2))\);
ADR2=\((ACCESS ((DATA ((CHAR))))))\);

COST CONSTRAINT 0
INDEX VALUE 0
INDEX COST 0
MODE VALUE 0
MODE COST 0
LEFTOP VALUE 0
LEFTOP COST 0
I/O VALUE 50
I/O COST 0
OPERATION VALUE 10
OPERATION COST 0
C31 5
C32 0
C33 5
C34 0
C35 1
C36 0
C37 1
C38 0
C39 5
C40 0
C41 1
C42 0
C43 1
C44 0
C45 1
C46 0

STRATEGY:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T61</td>
<td>152</td>
<td>0</td>
</tr>
<tr>
<td>T73</td>
<td>242</td>
<td>0</td>
</tr>
<tr>
<td>T72</td>
<td>302</td>
<td>0</td>
</tr>
<tr>
<td>T79</td>
<td>345</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 11
MEMORY SIZE IS 2048 WORDS
WORD SIZE IS 16 BITS
BYTE SIZE IS 8 BITS

*****INSTRUCTION GROUP 1*****
INSTRUCTION LENGTH 16 BITS

**OP CODE (RIGHTOP)**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>00000</td>
</tr>
<tr>
<td>-</td>
<td>00001</td>
</tr>
<tr>
<td>ABS</td>
<td>00010</td>
</tr>
<tr>
<td>-ABS</td>
<td>00011</td>
</tr>
<tr>
<td>COMP</td>
<td>00100</td>
</tr>
<tr>
<td>TALLY</td>
<td>00101</td>
</tr>
<tr>
<td>C53</td>
<td>00110</td>
</tr>
<tr>
<td>C54</td>
<td>00111</td>
</tr>
<tr>
<td>*</td>
<td>01000</td>
</tr>
<tr>
<td>÷</td>
<td>01001</td>
</tr>
<tr>
<td>C51</td>
<td>01010</td>
</tr>
<tr>
<td>C52</td>
<td>01011</td>
</tr>
</tbody>
</table>

**RIGHTOP: FIXED POINT**

**RIGHTOP: DISPLACEMENT, INTEGER**

Figure 11 (continued)
### Instruction Group 2

**Instruction Length 16 Bits**

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>RIGHTOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>01100</td>
</tr>
<tr>
<td>STORE</td>
<td>01101</td>
</tr>
<tr>
<td>EYEC</td>
<td>01110</td>
</tr>
<tr>
<td>COMP</td>
<td>01111</td>
</tr>
<tr>
<td>C61</td>
<td>10000</td>
</tr>
<tr>
<td>C62</td>
<td>10001</td>
</tr>
</tbody>
</table>

**Bits 0 to 4**

**Rightop Byte**

**Bits 5 to 15**

**Rightop*Displacement*Integer**

---

*Figure 11 (continued)*
***** INSTRUCTION GROUP 3 *****
INSTRUCTION LENGTH 16 BITS.

BITS 0 TO 4
OP CODE

10010

-----OPERATION GROUP 1-----

OPERATION CODE
BRNCH 10010

BITS 5 TO 15
IF*DISPLACEMENT*INTEGER*

Figure 11 (continued)
### Instruction Group 4

**Instruction Length**: 16 bits

**Op Code**: `RIGHTOP`

#### Bits 0 to 4

<table>
<thead>
<tr>
<th>Bit Mask</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/0</code></td>
<td><code>10011</code></td>
</tr>
<tr>
<td><code>/0</code></td>
<td><code>10100</code></td>
</tr>
<tr>
<td><code>/0</code></td>
<td><code>10101</code></td>
</tr>
<tr>
<td><code>/0</code></td>
<td><code>10110</code></td>
</tr>
</tbody>
</table>

#### Bits 5 to 15

<table>
<thead>
<tr>
<th>Bit Mask</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>RIGHTOP*DISPLACEMENT*INTEGER*</code></td>
<td></td>
</tr>
</tbody>
</table>
a single-address format. T5 also adds a branch operation and load
and store operations.

Since the addition of I/O operations results in the greatest
increase in the value of the instruction set the first step in the
optimization phase is to add I/O operations by means of T61. The
next step is to add the operations abs, -abs, comp, tally, C53, C54.
This is done by T73. Each of these operations adds a value of 15 to
the score for the instruction set -- 10 for operation value and 5 be-
cause the mix coefficient of the operation group, C33, is set to 5.
Hence T73 results in an increase of 90 in the total score. The opera-
tors T72 and T79 are applied in a similar manner to add one more group
of arithmetic operations and one group of character operations bring-
ing the total value of the instruction set to 346.

The instruction set designed by ISDS has the same format as the
actual Whirlwind I. The number of bits required for operation code
are the same; however the ISDS version of Whirlwind contains only
23 operations, compared to 34 for the actual Whirlwind. Twelve of
the operations (including I/O) of the ISDS version correspond directly
to operations in the actual Whirlwind. The six dummy operations can
be associated with six operations in the actual Whirlwind I. Hence
the ISDS version of Whirlwind I accounts, at best, for 18 of the 34
operations in Whirlwind. This failure of ISDS is due to the limited
number of operations in GIS. One reason for this is the variation
in operations of different computers. GIS contains only the operations
that appear in a large number of computers. This shortage of operators
can be remedied, as we have indicated partially in this example, by
the addition of new operations via the instruction mix. One operation
that should be added to the present example is a conditional branch. The branch instruction format does not permit bits for conditions. In the actual Whirlwind, conditional branching is provided by a special operation which is missing from GIS.

There is one other important difference between the ISDS version of Whirlwind I and the actual Whirlwind. Four instructions of the actual Whirlwind contain 6-bit operation codes. All of the counting and output routines of ISDS assume that the same bits of each instruction format are used for operation code.

Example 2. This example (figure 12) is a variation on example 1 obtained by setting the address mode value coefficient equal to 100. The result is an instruction set with a 4-bit operation code, a 15-bit address specification and one bit for direct or indirect addressing.

The basic instruction set produced by T5 contains 5 operations requiring 3 bits for operation code. The indirect-direct bit is added first during the optimization phase because it results in the highest score. Hence only 1 bit remains for new operation codes. This bit is used to encode I/O operations supplied by T61 and one group of arithmetic operations supplied by T73. No additional operators can be applied at this stage because all bits of the instruction word have been used. Although the completed instruction set has a powerful addressing method its set of operations is very limited.

Example 3. Example 3 is a variation of example 1 obtained by assigning the value 100 to the value coefficient for indexing. The printed output for example 3 is shown in figure 13. The only operator
OUTPUT FOR EXAMPLE 2

```
COMPUTER += (((MEM (mK32) CODE (ORDERS) WORD (mN16) BYTE (mN8))))
ORDERS += ('GROUP1, 'GROUP2);
GROUP1 += ('11);
I1 += (((OPR ((+, -)) RESULT ('ADR1)));
ADR1 += (((ACCESS (((DATA ((FIX)))))));
GROUP2 += ('12);
I2 += (((OPR ((MOVE)) RESULT ('ADR2)));
ADR2 += (((ACCESS (((DATA ((CHAR)))))));
COMPUTER; mT39,

| COST CONSTRAINT | 0 |
| INDEX VALUE     | 0 |
| INDEX COST      | 0 |
| MODE VALUE      | 100 |
| MODE COST       | 0 |
| LEFTOP VALUE    | 0 |
| LEFTOP COST     | 0 |
| I/O VALUE       | 50 |
| I/O COST        | 0 |
| OPERATION VALUE | 10 |
| OPERATION COST  | 0 |
| C31              | 5 |
| C32              | 0 |
| C33              | 5 |
| C34              | 0 |
| C35              | 1 |
| C36              | 0 |
| C37              | 1 |
| C38              | 0 |
| C39              | 5 |
| C40              | 0 |
| C41              | 1 |
| C42              | 0 |
| C43              | 1 |
| C44              | 0 |
| C45              | 1 |
| C46              | 0 |

STRATEGY:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T56</td>
<td>162</td>
<td>0</td>
</tr>
<tr>
<td>T61</td>
<td>252</td>
<td>0</td>
</tr>
<tr>
<td>T73</td>
<td>342</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Figure 12
MEMORY SIZE IS 2048 WORDS  
WORD SIZE IS 16 BITS  
BYTE SIZE IS 8 BITS  

***INSTRUCTION GROUP 1***  
INSTRUCTION LENGTH 16 BITS  

BITS 0 TO 3  
OP CODE  

RIGHTOP: FIXED POINT  

OPERATION | CODE  
---|---  
+ | 0000  
- | 0001  
ABS | 0010  
-ABS | 0011  
COMP | 0100  
TALLY | 0101  
C53 | 0110  
C54 | 0111  

BITS 4 TO 4  
RIGHTOP=O OPERAND MODE  

DIRECT | INDIRECT  
---|---  
0 | 1  

BITS 5 TO 15  
RIGHTOP=DISPLACEMENT INTEGER  

Figure 12 (continued)
| OP CODE | RIGHTOP | B YTE | OPERATION GROUP 1---
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>1000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>1001</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **BITS 0 TO 3**
- **OP. CODE**
- **RIGHTOP:**
- **BYTES:**
- **OPERATION GROUP 1---**
- **OPERATION**
- **CODE**

- **DIRECT**
- **INDIRECT**

- **BITS 4 TO 15**
- **RIGHTOP**
- **OP. DETAIL**
- **MODE**
- **DIRECT**
- **INDIRECT**

---

**Figure 12 (continued)**
***INSTRUCTION GROUP 3***

INSTRUCTION LENGTH 16 BITS

BIT 0 TO 3

OP CODE

---OPERATION GROUP 1---

OPERATION CODE

BRNCH 1010

BIT 4 TO 14

IF DISPLACEMENT INTEGER

*Figure 12 (continued)*
**INSTRUCTION GROUP 4**

INSTRUCTION LENGTH: 16 BITS

| OP CODE | RIGHTOP1
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BITS 0 TO 3</td>
<td></td>
</tr>
</tbody>
</table>

---

**OPERATION GROUP 1**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>1011</td>
</tr>
<tr>
<td>I/O</td>
<td>1100</td>
</tr>
<tr>
<td>I/O</td>
<td>1101</td>
</tr>
<tr>
<td>I/O</td>
<td>1110</td>
</tr>
</tbody>
</table>

- **BITS 4 TO 4**
- **RIGHTOP*OP DETAIL*MODE**
- **DIRECT**
  - 0
- **INDIRECT**
  - 1

- **BITS 5 TO 15**
- **RIGHTOP*DISPLACEMENT*INTEGER**

Figure 12 (continued)
OUTPUT FOR EXAMPLE 3

```plaintext
COMPUTER=((MEM (X32) CODE QUORDS) WORD (N16) RYTE (MNR));
ORDERS=((GROUP1, 'GROUP2));
GROUP1=('I1);
I1=((OPR ((+, -)) RESULT ('ADR1));
ADR1=((ACCESS ((DATA ((FIX))));
GROUP2=('I2);
I2=((OPR ((MOVE)) RESULT ('ADR2));
ADR2=((ACCESS ((DATA ((CHAR))));
COMPUTER; T39..
```

| COST CONSTRAINT | 0 |
| INDEX VALUE     | 100 |
| INDEX COST      | 0 |
| MODE VALUE      | 0 |
| MODE COST       | 0 |
| LEFTOP VALUE    | 0 |
| LEFTOP COST     | 0 |
| I/O VALUE       | 50 |
| I/O COST        | 0 |
| OPERATION VALUE | 10 |
| OPERATION COST  | 0 |
| C31              | 5 |
| C32              | 0 |
| C33              | 5 |
| C34              | 0 |
| C35              | 1 |
| C36              | 0 |
| C37              | 1 |
| C38              | 0 |
| C39              | 5 |
| C40              | 0 |
| C41              | 1 |
| C42              | 0 |
| C43              | 1 |
| C44              | 0 |
| C45              | 1 |
| C46              | 0 |

**STRATEGY:**

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T68</td>
<td>162</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 13
MEMORY SIZE IS 2048 WORDS
WORD SIZE IS 16 BITS
BYTE SIZE IS 8 BITS

****INSTRUCTION GROUP 1****
INSTRUCTION LENGTH 16 BITS

BITS 0 TO 2
   OP CODE

RIGHTOP: FIXED POINT

OPERATION CODE
   + 000
   - 001

BITS 3 TO 4
   RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 5 TO 15
   RIGHTOP*DISPLACEMENT*INTEGER*

Figure 13 (continued)
### Instruction Group 2

**Instruction Length**: 16 bits

<table>
<thead>
<tr>
<th>Bits 0 to 2 (op code)</th>
<th>Bit 3 to 4 (right shift)</th>
<th>Bits 5 to 15 (right shift)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>STORE</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Operation Group 1**

Figure 13 (continued)
Figure 13 (continued)
applied is T68, which assigns the two available bits to indexing; i.e. the computer has 4 index registers. The number of operations is too restricted to be useful, since only three bits are available for operation code. In this example a two-step look-ahead strategy would have designed a more useful instruction set since it would have obtained a higher score by using one of the two available bits for new operations and the other for indexing.

The method used by ISDS to assign operation codes is similar to the method in which codes were assigned to the Whirlwind operations. That is, the codes were assigned to operations in sequence with no particular effort to find similar encodings for similar operations. The operation codes in the examples, however, do not agree with the actual codes since ISDS receives no information about encoding and makes no attempt to find an efficient encoding.

Examples 4-7

The examples in this group illustrate the use of ISDS to design instruction sets similar to the IAS instruction set. The actual IAS instruction set is shown in figure 14. The original specification of the instruction set contains no operation codes or input/output operations. However, six bits are reserved for operation codes, so that input/output operations can easily be added.

The IAS computer has a word size of 40 bits. A word size of 33 bits was required to provide the accuracy demanded by the task environment. Since the memory was to contain only 4,096 words (requiring 12 bits for encoding) only 17 to 18 bits were required for each instruction depending upon whether 5 bits or 6 were used to encode
IAS

Memory size = 4096
Word size = 40

* * * Instruction Group 1 * * *

| op code | rt op |

--- Operation Group 1 ---
right op: fixed point

Operation
clear and add
clear and sub
clear and add abs
clear and sub abs
add
sub
add abs
sub abs
clear & add reg
clear & load reg
clear & mult reg
clear reg & divide
store

Bits 6 to 17
rt op* displacement* integer

* * * Instruction Group 2 * * *

| op code | if |

--- Operation Group 1 ---
right op: char

Operation
branch rt
branch left
branch rt ≥ 0
branch left ≥ 0

Figure 14
Operation
replace rt
replace left

Bits 6 to 17
if*displacement*integer

***Instruction Group 3***

|op code|

Bits 0 to 5
op code

---Operation Group 1---

Operation
shift rt
shift left

Figure 14 (continued)
operations. The word size of 40 bits was chosen to permit two instructions per word. This meant that each instruction contained 20 bits, 2 of which were unused.

Example 4. This example shows ISDS's attempt to duplicate the design of the IAS instruction set. The symbolic input specifies memory size (K32 = 4096), word size (N40 = 40) and byte size (N4 = 4). The single instruction group contains two operations, + and -, for fixed-point arithmetic. The symbolic input for this example represents the smallest partially-specified instruction set that can be supplied to the design program.

The value coefficients specify a value for every addressing feature. The instruction mix coefficients, except for C45, are equal to zero, although this is not necessary since operations are added only for data types specified in the symbolic input. That is, no data formats are added to an instruction set by ISDS. Indexing, indirect addressing (mode), and left operand are assigned costs of 1. Since the cost constraint is equal to zero none of these addressing features will be added to the instruction set in spite of the fact that the instruction format contains unused bits.

In this example the basic strategy, T5, first determines that the 40-bit word can accommodate three address specifications. T5 then sets the instruction length equal to 20 bits (=1/2 x 40) and calls itself recursively. On the second call of T5 there is room in the instruction format for only one address specification. T5 specifies a single-address format of 20 bits and adds a branch instruction
OUTPUT FOR EXAMPLE 4


<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T73</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>T72</td>
<td>75</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 15
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

**** INSTRUCTION GROUP 1 ****
INSTRUCTION LENGTH 20 Bits

BITS 0 TO 3
OP CODE

RIGHTOP: FIXED POINT

OPERATION CODE

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0000</td>
</tr>
<tr>
<td>-</td>
<td>0001</td>
</tr>
<tr>
<td>ABS</td>
<td>0010</td>
</tr>
<tr>
<td>-ABS</td>
<td>0011</td>
</tr>
<tr>
<td>COMP</td>
<td>0100</td>
</tr>
<tr>
<td>TALLY</td>
<td>0101</td>
</tr>
<tr>
<td>C53</td>
<td>0110</td>
</tr>
<tr>
<td>C54</td>
<td>0111</td>
</tr>
<tr>
<td>*</td>
<td>1000</td>
</tr>
<tr>
<td>/</td>
<td>1001</td>
</tr>
<tr>
<td>C51</td>
<td>1010</td>
</tr>
<tr>
<td>C52</td>
<td>1011</td>
</tr>
</tbody>
</table>

BITS 4 TO 15
RIGHTOP: DISPLACEMENT * INTEGER

Figure 15 (continued)
*****INSTRUCTION GROUP 2*****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

OPERATION GROUP 1

OPERATION
BRNCH

BITS 4 TO 7
CONDITION

BITS 8 TO 19
IF*DISPLACEMENT*INTEGER*

Figure 15 (continued)
Figure 15 (continued)
and load and store operations. Although ISDS assigns two instructions to each word the strategy used by ISDS is slightly different from that used by Von Neumann in IAS. The specification sequence for the actual IAS was:

memory size → word size (33) → instruction length (18)
→ word size (40) → instruction length (20)

For ISDS the sequence is:

memory size → word size (40)
→ addresses/instruction (3)
→ instruction length (20)

The difference between these strategies is that word size was variable in the original strategy, whereas word size is treated as an input to our design program. Word size can be treated as a variable by ISDS, provided that an operator is added to perform the required analysis and specify a value for word size. The same is true of memory size.

As in the Whirlwind I examples there is considerable discrepancy in the operations in the actual IAS and the version of IAS designed by ISDS. The actual IAS contains 21 operations. The ISDS version of IAS contains 15 operations, 11 of which (making use of the dummy operations) occur in IAS. Moreover the ISDS version uses only 4 bits for operation code. The reason for this is the limited number of operations provided by GIS. In this case the shortage of operations could be corrected by adding operations to the instruction mix or by adding an operator that reserves bits for the operation code.
Example 5. This example is a two-operand version of the IAS instruction set. This instruction set in figure 16 was obtained by setting the cost coefficients of addressing features equal to zero. The first two operators applied add arithmetic operations. The third operator, T56, adds an indirect-direct address indicator to the right operand address specification. T43 adds a left operand to the instruction format. Since T43 specifies 3 bits for a register left operand the computer has 8 general registers. The instruction set designed by ISDS in this example has a limited set of operations but it provides powerful addressing capabilities by making efficient use of instruction bits.

Example 6. This example (figure 17) is a variation of example 4. Indexing was given a larger value than a left operand. The result is a variation of IAS with indirect addressing and 8 index registers. This instruction set offers powerful addressing features, but a limited set of operations.

Example 7. Example 7 (figure 18) is the same as example 6 with the exception that ISDS was modified to print the partially-specified instruction set at each stage of the optimization. The sequence of operators is:

1. T73 -- add arith. operations
2. T72 -- add arith. operations
3. T56 -- add indirect addressing
4. T68 -- add indexing
OUTPUT FOR EXAMPLE 5

```
COMPUTER+((MEM (k32) CODE (ORDERS) WORD (N40) BYTE (N4)));
ORDERS+('GROUP1);
GROUP1+('II);
I1+((OPR ((+,-)) RESULT (ADR1)));
ADR1+((ACCESS (((DATA ((FIX))))) ));
COMPUTER: T39..
```

COST CONSTRAINT 0
INDEX VALUE 10
INDEX COST 0
MODE VALUE 20
MODE COST 0
LEFTOP VALUE 10
LEFTOP COST 0
I/O VALUE 50
I/O COST 0
OPERATION VALUE 1
OPERATION COST 0
C31 5
C32 0
C33 5
C34 0
C35 0
C36 0
C37 0
C38 0
C39 0
C40 0
C41 0
C42 0
C43 0
C44 0
C45 0
C46 0

STRATEGY:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T73</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>T72</td>
<td>75</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>95</td>
<td>0</td>
</tr>
<tr>
<td>T43</td>
<td>105</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 16
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

**** INSTRUCTION GROUP 1 ****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

--- OPERATION GROUP 1 ---
RIGHTOP: FIXED POINT; LEFTOP: REG;

OPERATION    CODE
+            0000
-            0001
ABS          0010
-ABS         0011
COMP         0100
TALLY        0101
C53          0110
C54          0111
*            1000
/            1001
C51          1010
C52          1011

BITS 4 TO 6
LEFTOP*DISPLACEMENT*INTEGER*

BITS 7 TO 7
RIGHTOP*OP DETAIL*MODE*
DIRECT 0
INDIRECT 1

BITS 8 TO 19
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 16 (continued)
***INSTRUCTION GROUP 2***
INSTRUCTION LENGTH 20 BITS

**TOP CODE: IDENTIFY CONDITION**

---OPERATION GROUP 1---
OPERATION CODE
BRNCH 1100

BITS 0 TO 3
OP CODE

BITS 4 TO 7
CONDITION

BITS 8 TO 19
IF DISPLACEMENT INTEGER

Figure 16 (continued)
INSTRUCTION GROUP 3
INSTRUCTION LENGTH 20 BITS

OP CODE: 1111

OPERATION GROUP 1:

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>1101</td>
</tr>
<tr>
<td>STORE</td>
<td>1110</td>
</tr>
</tbody>
</table>

BITS 0 TO 3
OP CODE

BITS 4 TO 15
IF DISPLACEMENT INTEGER

Figure 16 (continued)
OUTPUT FOR EXAMPLE 6

COMPUTER = (((MEM (\#K32) CODE ("ORDERS")) WORD (N=0)) BYTE (N4)));
ORDERS = ('GROUP1);
GROUP1 = 'II);
II = (((OPR ((+, -)) RESULT ("ADR1")));
ADR1 = ((ACCESS (((DATA (\{FIX\}))));
COMPUTER = T39.

COST CONSTRAINT 0
INDEX VALUE 20
INDEX COST 0
MODE VALUE 20
MODE COST 0
LEFT TOP VALUE 10
LEFT TOP COST 0
I/O VALUE 50
I/O COST 0
OPERATION VALUE 1
OPERATION COST 0
C31  5
C32  0
C33  5
C34  0
C35  0
C36  0
C37  0
C38  0
C39  0
C40  0
C41  0
C42  0
C43  0
C44  0
C45  1
C46  0

STRATEGY:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T73</td>
<td>51</td>
<td>0</td>
</tr>
<tr>
<td>T72</td>
<td>75</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>93</td>
<td>0</td>
</tr>
<tr>
<td>T68</td>
<td>115</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 17
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

*****INSTRUCTION GROUP 1*****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

----OPERATION GROUP 1-----
RIGHTOP: FIXED POINT;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td>0001</td>
</tr>
<tr>
<td>ABS</td>
<td>0010</td>
</tr>
<tr>
<td>-ABS</td>
<td>0011</td>
</tr>
<tr>
<td>COMP</td>
<td>0100</td>
</tr>
<tr>
<td>TALLY</td>
<td>0101</td>
</tr>
<tr>
<td>C53</td>
<td>0110</td>
</tr>
<tr>
<td>C54</td>
<td>0111</td>
</tr>
<tr>
<td>*</td>
<td>1000</td>
</tr>
<tr>
<td>/</td>
<td>1001</td>
</tr>
<tr>
<td>C51</td>
<td>1010</td>
</tr>
<tr>
<td>C52</td>
<td>1011</td>
</tr>
</tbody>
</table>

BITS 4 TO 6
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 7 TO 7
RIGHTOP*OP DETAIL*MODE*

<table>
<thead>
<tr>
<th>DIRECT</th>
<th>INDIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

BITS 8 TO 19
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 17 (continued)
### INSTRUCTION GROUP 2

**INSTRUCTION LENGTH 20 BITS**

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>CONDITION*</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>000000000000</td>
<td>1100</td>
</tr>
</tbody>
</table>

**OPERATION GROUP 1**

<table>
<thead>
<tr>
<th>OPERATION CODE</th>
<th>BRANCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>1100</td>
</tr>
</tbody>
</table>

*Figure 17 (continued)*
*****INSTRUCTION GROUP 3*****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

----OPERATION GROUP 1----
OPERATION    CODE
MOVE         1101
STORE        1110

BITS 4 TO 15
IF*DISPLACEMENT*INTEGER*

Figure 17 (continued)
OUTPUT FOR EXAMPLE 7

<table>
<thead>
<tr>
<th>Mode</th>
<th>Value</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mode 1</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Mode 2</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>Leftop</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>I/O</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>Operation</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>C31</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>C32</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C33</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>C34</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C35</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C37</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C38</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C39</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C40</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C41</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C42</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C43</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C44</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C45</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C46</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

STRATEGY:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Total Value</th>
<th>Total Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>773</td>
<td>51</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 18
MEMORY SIZE IS 4096 WORDS.
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

****INSTRUCTION GROUP 1****
INSTRUCTION LENGTH 20 BITS

**OP CODE [RIGHTOP1]**

BITS 0 TO 3
OP CODE

**RIGHTOP: FIXED POINT**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0000</td>
</tr>
<tr>
<td>-</td>
<td>0001</td>
</tr>
<tr>
<td>ABS</td>
<td>0010</td>
</tr>
<tr>
<td>-ABS</td>
<td>0011</td>
</tr>
<tr>
<td>COMP</td>
<td>0100</td>
</tr>
<tr>
<td>TALLY</td>
<td>0101</td>
</tr>
<tr>
<td>C53</td>
<td>0110</td>
</tr>
<tr>
<td>C54</td>
<td>0111</td>
</tr>
</tbody>
</table>

BITS 4 TO 15
RIGHTOP: DISPLACEMENT*INTEGER

****INSTRUCTION GROUP 2****
INSTRUCTION LENGTH 20 BITS

**OP CODE [IF*CONDITION1]**

BITS 0 TO 3
OP CODE

**OPERATION GROUP 1**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BRNCH</td>
<td>1000</td>
</tr>
</tbody>
</table>

BITS 4 TO 7
CONDITION

BITS 8 TO 19
IF*DISPLACEMENT*INTEGER

****INSTRUCTION GROUP 3****
INSTRUCTION LENGTH 20 BITS

**OP CODE [IF*CONDITION1]**

BITS 0 TO 3
OP CODE

**OPERATION GROUP 1**

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>1001</td>
</tr>
<tr>
<td>STORE</td>
<td>1010</td>
</tr>
</tbody>
</table>

Figure 18 (continued)
Figure 18 (continued)
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

***INSTRUCTION GROUP 1***
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

RIGHTOP: FIXED POINT:

OPERATION CODE

+ 0000
- 0001
ABS 0010
-ABS 0011
COMP 0100
TALLY 0101
C53 0110
C54 0111
* 1000
/ 1001
C51 1010
C52 1011

BITS 4 TO 15
RIGHTOP=DISPLACEMENT+INTEGER

***INSTRUCTION GROUP 2***
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

-----OPERATION GROUP 1-----
OPERATION CODE

BRNCH 1100

BITS 4 TO 7
CONDITION

BITS 8 TO 19
IF*DISPLACEMENT+INTEGER

***INSTRUCTION GROUP 3***
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

-----OPERATION GROUP 1-----

Figure 18 (continued)
<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>1101</td>
</tr>
<tr>
<td>STORE</td>
<td>1110</td>
</tr>
</tbody>
</table>

BITS 4 TO 15
IF*DISPLACEMENT*INTEGER*
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

*****INSTRUCTION GROUP 1*****
INSTRUCTION LENGTH 20 BITS

<table>
<thead>
<tr>
<th>OPERAND CODE</th>
<th>OPERATION GROUP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>+</td>
</tr>
<tr>
<td>0001</td>
<td>-</td>
</tr>
<tr>
<td>0010</td>
<td>ABS</td>
</tr>
<tr>
<td>0011</td>
<td>-ABS</td>
</tr>
<tr>
<td>0100</td>
<td>COMP</td>
</tr>
<tr>
<td>0101</td>
<td>TALLY</td>
</tr>
<tr>
<td>0110</td>
<td>C53</td>
</tr>
<tr>
<td>0111</td>
<td>C54</td>
</tr>
<tr>
<td>1000</td>
<td>*</td>
</tr>
<tr>
<td>1001</td>
<td>/</td>
</tr>
<tr>
<td>1010</td>
<td>C51</td>
</tr>
<tr>
<td>1011</td>
<td>C52</td>
</tr>
</tbody>
</table>

BITS 0 TO 3

<table>
<thead>
<tr>
<th>INSTRUCTION GROUP 2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>OPERAND CODE</th>
<th>CONDITION GROUP 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100</td>
<td>BRNCCH</td>
</tr>
</tbody>
</table>

BITS 0 TO 3

<table>
<thead>
<tr>
<th>INSTRUCTION GROUP 3</th>
</tr>
</thead>
</table>

Figure 18 (continued)
BITS 0 TO 3
    OP CODE

-----OPERATION GROUP 1-----
OPERATION   CODE
MOVE        1101
STORE       1110

BITS 4 TO 15
    IF DISPLACEMENT INTEGER

Figure 18 (continued)
MEMORY SIZE IS 4096 WORDS
WORD SIZE IS 40 BITS
BYTE SIZE IS 4 BITS

****INSTRUCTION GROUP 1****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

RIGHTOP: FIXED POINT:

OPERATION CODE
+
-
ABS
-ABS
COMP
TALLY
C53
C54
*
/
C51
C52

0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011

BITS 4 TO 6
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 7 TO 7
RIGHTOP*OP DETAIL*MODE*

BITS 8 TO 19
RIGHTOP*DISPLACEMENT*INTEGER*

****INSTRUCTION GROUP 2****
INSTRUCTION LENGTH 20 BITS

BITS 0 TO 3
OP CODE

-----OPERATION GROUP 1-----

OPERATION CODE
BRANCH

1100

BITS 4 TO 7
CONDITION*

BITS 8 TO 19
DISPLACEMENT*INTEGER*

****INSTRUCTION GROUP 3****
INSTRUCTION LENGTH 20 BITS

Figure 18 (continued)
BITS 0 TO 3
OP CODE

-----OPERATION GROUP 1-----

OPERATION CODE
MOVE 1101
STORE 1110

BITS 4 TO 15
IF*DISPLACEMENT*INTEGER*

Figure 18 (continued)
Examples 8-12

The examples in this group illustrate attempts by ISDS to create a design strategy for the Univac 1108. The instruction format of the 1108 is illustrated in figure 19. The "partial word processes" of the 1108 are not included in GIS. The four "partial word" bits are used to specify that the operation is to affect only the certain bits in the operand field. Each combination of the "partial word" bits selects a different portion of the word to be affected.

With the exception of partial word processes, all of the addressing features of the 1108 are contained in GIS. The four-bit left operand specifies one of 16 general registers. The main memory right operand may be modified by indexing (any one of the 16 general registers) or indirect addressing. The contents of an index register are separated into two parts -- an index value and an adjustment. The adjustment is added to the index value if the adjustment switch is on.

The arithmetic data formats of the 1108 are as follows:

1. half-word
2. third-word
3. single precision floating point
4. double precision floating point
5. single precision fixed point
6. double precision fixed point

Although the 1108 has a 6-bit operation code it contains 150 operations, since the partial word bits can be used to encode certain operations. None of the examples in this group contain as complete a set of operations as the 1108. Some of the examples, however, contain more than
1108 INSTRUCTION FORMAT

Bits 0 to 5
   op code

Bits 6 to 9
   partial word process

Bits 10 to 13
   left op*displacement*integer

Bits 14 to 17
   right op*index*address*displacement*integer

Bits 18 to 18
   right op*index*adjustment*switch
      - off 0
      - on 1

Bits 19 to 19
   right op*mode
      - direct 0
      - indirect 1

Bits 20 to 35
   right op*displacement*integer

Figure 19
32 operations so that 6 bits are required for operation codes. Hence although ISDS does not contain all of the operations of the 1108, it is able to design the 1108 instruction format.

Example 8. This example (figure 20) illustrates a first attempt to construct a design strategy for the 1108. The symbolic input specified the memory size \((K_{32} = 65536)\), word size \((N_{36} = 36)\), and byte size \((N_8 = 8)\). The input instruction set contains an instruction group for every data type of GIS. All of the value coefficients have positive values. The cost constraint is equal to 20. Since the index cost coefficient is the only positive cost coefficient, the cost constraint restricts the instruction set to 2 index register specifications. (Double indexing).

Since the symbolic input contains 15 operations (requiring only 4 bits for encoding) the basic strategy, T5, finds that the 36-bit word can accommodate a 4-bit operation code and two 16-bit address specifications. Hence, T5 creates a two-address instruction set. No additional addressing features can be added since the operation code and the two 16-bit displacements use every bit of the instruction word.

Example 9. This example (figure 21) is an attempt to improve example 8 by adding operations to the symbolic input. The result is shown in figure 21. In this case the initial partially-specified instruction set contains 17 operations. Since these operations require 5 bits for operation code it is not possible to fit two main memory address specifications into each instruction. T5 specifies
OUTPUT FOR EXAMPLE 8

COMPUTER=(((MEM (H32) CODE ('ORDERS) WORD (N6) BYTE (N8)));
ORDERS=('GROUP1,'GROUP2,'GROUP3,'GROUP4);
GROUP1=('I1);
I1=(((OPR ((+,-,*,/,COMP)) RESULT ('ADR1)));
ADR1=((ACCESS (((DATA ((FIX)))))));
GROUP2=('I2);
I2=(((OPR ((+,-,*,/,COMP)) RESULT ('ADR2)));
ADR2=(((ACCESS (((DATA ((FLOAT)))))));
GROUP3=('I3);
I3=(((OPR ((NEG,AND,OR,NOP)) RESULT ('ADR3)));
ADR3=(((ACCESS (((DATA ((LOG)))))));
GROUP4=('I4);
I4=(((OPR ((MOVE)) RESULT ('ADR4)));
ADR4=(((ACCESS (((DATA ((CHAR)))))));
COMPUTER; T139..

COST CONSTRAINT 20
INDEX VALUE 10
INDEX COST 10
MODE VALUE 20
MODE COST 0
LEFTOP VALUE 10
LEFTOP COST 0
I/O VALUE 50
I/O COST 0
OPERATION VALUE 1
OPERATION COST 0
C31 1
C32 0
C33 1
C34 0
C35 1
C36 0
C37 1
C38 0
C39 1
C40 1
C41 0
C42 1
C43 1
C44 0
C45 1
C46 0

STRATEGY:

OPERATOR TOTAL VALUE TOTAL COST

Figure 20
MEMORY SIZE IS 65536 WORDS
WORD SIZE IS 36 BITS
BYTE SIZE IS 8 BITS

***INSTRUCTION GROUP 1***
INSTRUCTION LENGTH 36 BITS

**OP CODE**

**RIGHTOP** : FIXED POINT; **LEFTOP** : FIXED POINT;

**OPERATION** | **CODE**
---|---
+ | 0000
- | 0001
* | 0010
/ | 0011
COMP | 0100

IMPLIED VALUE

**LEFTOP** * **OPERATION** * **LEFTOP**

**LEFTOP** * **DISPLACEMENT** * **INTEGER**

**RIGHTOP** * **DISPLACEMENT** * **INTEGER**

Figure 20 (continued)
### Instruction Group 2

**Instruction Length 36 Bits**

**Op Code**: LEFTOP | RIGHTOP

**Operation Group 1**

**Rightop**: Float Point; **Leftop**: Float Point;

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0101</td>
</tr>
<tr>
<td>-</td>
<td>0110</td>
</tr>
<tr>
<td>*</td>
<td>0111</td>
</tr>
<tr>
<td>/</td>
<td>1000</td>
</tr>
<tr>
<td>COMP</td>
<td>1001</td>
</tr>
</tbody>
</table>

**Implied Value**

**Leftop**

<table>
<thead>
<tr>
<th>Bits 4 to 19</th>
<th><strong>Leftop</strong> + <strong>Displacement</strong> + <strong>Integer</strong></th>
</tr>
</thead>
</table>

| Bits 20 to 35 | **Rightop** + **Displacement** + **Integer** |

---

*Figure 20 (continued)*
**INSTRUCTION GROUP 3**

**INSTRUCTION LENGTH 36 BITS**

OP CODE | LEFTOP | RIGHTOP
---|---|---

---OPERATION GROUP 1---

RIGHTOP: LOGIC; LEFTOP: LOGIC;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>1010</td>
</tr>
<tr>
<td>AND</td>
<td>1011</td>
</tr>
<tr>
<td>OR</td>
<td>1100</td>
</tr>
<tr>
<td>NOP</td>
<td>1101</td>
</tr>
</tbody>
</table>

IMPLIED VALUE

LEFTOP*OP DETAIL*

BITS 4 TO 19

LEFTOP*DISPLACEMENT*INTEGER*

BITS 20 TO 35

RIGHTOP*DISPLACEMENT*INTEGER*

Figure 2Q (continued)
****INSTRUCTION GROUP 4****
INSTRUCTION LENGTH 36 BITS

**OP CODE:LEFTOP:RIGHTOP**

BITS 0 TO 3
   OP CODE

-----OPERATION GROUP 1-----
RIGHTOP: BYTE; LEFTOP: BYTE;

OPERATION       CODE
    MOVE        1110

IMPLIED VALUE
LEFTOP*OP DETAIL*

BITS 4 TO 19
LEFTOP*DISPLACEMENT*INTEGER*

BITS 20 TO 35
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 20 (continued)
Figure 20 (continued)
OUTPUT FOR EXAMPLE 9

COMPUTER+((MEM ((X32) CODE ('ORDERS) WORD ('N36) BYTE ('N8)));
ORDERS+('GROUP1, 'GROUP2, 'GROUP3, 'GROUP4);
GROUP1+('I1);
I1+((OPR ((+,-,/,COMP,A35)) RESULT ('ADR1)));
ADR1+((ACCESS (((DATA ((FIX)))))));
GROUP2+('I2);
I2+((OPR ((+,-,/,COMP,A35)) RESULT ('ADR2)));
ADR2+((ACCESS (((DATA ((FLOAT)))))));
GROUP3+('I3);
I3+((OPR ((NEG,AND,OR,VOP)) RESULT ('ADR3)));
ADR3+((ACCESS (((DATA ((LOG)))))));
GROUP4+('I4);
I4+((OPR ((MOVE)) RESULT ('ADR4)));
ADR4+((ACCESS (((DATA ((CHAR)))))));

COMPUTER: T39.

COST CONSTRAINT: 10
INDEX VALUE: 10
INDEX COST: 10
MODE VALUE: 20
MODE COST: 0
LEFTTOP VALUE: 10
LEFTTOP COST: 0
I/O VALUE: 50
I/O COST: 0
OPERATION VALUE: 1
OPERATION COST: 0
C31: 1
C32: 0
C33: 1
C34: 0
C35: 0
C36: 0
C37: 1
C38: 0
C39: 1
C40: 0
C41: 1
C42: 0
C43: 1
C44: 0
C45: 1
C46: 0

STRATEGY:

<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T61</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>T43</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>160</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 21
\begin{table}
\centering
\begin{tabular}{ccc}
\hline
T77 & 172 & 0 \\
T78 & 184 & 0 \\
T68 & 194 & 10 \\
T73 & 202 & 10 \\
T79 & 210 & 10 \\
T82 & 216 & 10 \\
T83 & 222 & 10 \\
T72 & 226 & 10 \\
T74 & 230 & 10 \\
\hline
\end{tabular}
\caption{Figure 21 (continued)}
\end{table}
MEMORY SIZE IS 65536 WORDS
WORD SIZE IS 36 BITS
BYTE SIZE IS 8 BITS

*****INSTRUCTION GROUP 1*****
INSTRUCTION LENGTH 36 BITS

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>LEFTTOP</th>
<th>RIGHTTOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>000000</td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>000001</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>000010</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>000011</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>000100</td>
<td></td>
</tr>
<tr>
<td>ABS</td>
<td>000101</td>
<td></td>
</tr>
<tr>
<td>-ABS</td>
<td>000110</td>
<td></td>
</tr>
<tr>
<td>TALLY</td>
<td>000111</td>
<td></td>
</tr>
<tr>
<td>C53</td>
<td>001000</td>
<td></td>
</tr>
<tr>
<td>C54</td>
<td>001001</td>
<td></td>
</tr>
<tr>
<td>C51</td>
<td>001010</td>
<td></td>
</tr>
<tr>
<td>C52</td>
<td>001011</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFTTOP<em>DISPLACEMENT</em>INTEGER*</td>
</tr>
<tr>
<td>RIGHTTOP<em>INDEX</em>ADDRESS<em>DISPLACEMENT</em>INTEGER*</td>
</tr>
<tr>
<td>RIGHTTOP<em>OP</em>DETAIL<em>MODE</em></td>
</tr>
<tr>
<td>DIRECT</td>
</tr>
<tr>
<td>INDIRECT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OPERATION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHTTOP<em>DISPLACEMENT</em>INTEGER*</td>
</tr>
</tbody>
</table>

Figure 21 (continued)
*****INSTRUCTION GROUP 2*****
INSTRUCTION LENGTH 36 BITS

BITS 0 TO 5
OP CODE

-----OPERATION GROUP 1-----
RIGHTOP: FLOAT POINT; LEFTOP: REG;

OPERATION       CODE
+               001100
-               001101
*               001110
/               001111
COMP            010000
ABS             010001
NORM            010010
C57             010011
C58             010100
C55             010101
C56             010110

BITS 6 TO 9
LEFTOP*DISPLACEMENT*INTEGER*

BITS 10 TO 13
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 14 TO 14
RIGHTOP*OP.DETAIL*MODE*

DIRECT          0
INDIRECT        1

BITS 15 TO 30
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 21 (continued)
**FIGURE 21** (continued)

**INSTRUCTION GROUP 3**

**INSTRUCTION LENGTH 36 BITS**

Top Code | Bits 0 to 5
---|---
LEFTOP | RIGHTOP

---OPERATION GROUP 1---

**RIGHTOP**: LOGIC; **LEFTOP**: REG;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>010111</td>
</tr>
<tr>
<td>AND</td>
<td>011000</td>
</tr>
<tr>
<td>OR</td>
<td>011001</td>
</tr>
<tr>
<td>NOP</td>
<td>011010</td>
</tr>
<tr>
<td>SHIFT</td>
<td>011011</td>
</tr>
<tr>
<td>TRUE</td>
<td>011100</td>
</tr>
<tr>
<td>V13</td>
<td>011101</td>
</tr>
<tr>
<td>V14</td>
<td>011110</td>
</tr>
<tr>
<td>V16</td>
<td>011111</td>
</tr>
<tr>
<td>V17</td>
<td>100000</td>
</tr>
<tr>
<td>V18</td>
<td>100001</td>
</tr>
<tr>
<td>V19</td>
<td>100010</td>
</tr>
<tr>
<td>V21</td>
<td>100011</td>
</tr>
<tr>
<td>V23</td>
<td>100100</td>
</tr>
<tr>
<td>V24</td>
<td>100101</td>
</tr>
<tr>
<td>V25</td>
<td>100110</td>
</tr>
<tr>
<td>FALSE</td>
<td>100111</td>
</tr>
<tr>
<td>C59</td>
<td>101000</td>
</tr>
<tr>
<td>C60</td>
<td>101001</td>
</tr>
</tbody>
</table>

**BITS 6 TO 9**

LEFTOP *DISPLACEMENT* INTEGER

**BITS 10 TO 13**

RIGHTOP *INDEX* ADDRESS *DISPLACEMENT* INTEGER

**BITS 14 TO 14**

RIGHTOP *OP DETAIL* MODE

<table>
<thead>
<tr>
<th>MODE</th>
<th>DIRECT</th>
<th>INDIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**BITS 15 TO 30**

RIGHTOP *DISPLACEMENT* INTEGER

Figure 21 (continued)
### Instruction Group 4

**Instruction Length:** 36 Bits

**Top Code:** [Left Top] Right Top

---

**Operation Group 1**

**Right Top:** Byte; **Left Top:** Reg;

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>101010</td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
</tr>
<tr>
<td>EXEC</td>
<td>101100</td>
</tr>
<tr>
<td>COMP</td>
<td>101101</td>
</tr>
<tr>
<td>C61</td>
<td>101110</td>
</tr>
<tr>
<td>C62</td>
<td>101111</td>
</tr>
</tbody>
</table>

- **Bits 6 to 9:** Left Top: Displacement*Integer*
- **Bits 10 to 13:** Right Top: Index*Address*Displacement*Integer*
- **Bits 14 to 14:** Right Top: Op Detail*Mode*
- **Bits 15 to 30:** Right Top: Displacement*Integer*

---

**Figure 21 (continued)**
Figure 21 (continued)
*****INSTRUCTION GROUP 6*****
INSTRUCTION LENGTH 36 BITS

**OP CODE**=**LEFTOP**+**RIGHTOP**

---OPERATION GROUP 1---

**LEFTOP**: BYTE;**RIGHTOP**: REG;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>110001</td>
</tr>
<tr>
<td>I/O</td>
<td>110010</td>
</tr>
<tr>
<td>I/O</td>
<td>110011</td>
</tr>
<tr>
<td>I/O</td>
<td>110100</td>
</tr>
</tbody>
</table>

**RIGHTOP**: DISPLACEMENT,INTEGER,

**RIGHTOP**: INDEX,ADDRESS,DISPLACEMENT,INTEGER,

**DIRECT**

**INDIRECT**

Figure 21 (continued)
a single-address format. The first three operators during the optimization phase add I/O operations, general registers, and indirect addressing. T77 and T78 add logic operations from the instruction mix. T68 adds 4 bits for an index register specification since the instruction set already contains the 16 general registers specified by T43. Only one index specification is added because the index cost equals the cost constraint. The remaining operators of the strategy add operations from the instruction mix. Since the number of operations in the completed instruction set exceeds 32, 6 bits are required for operation code. The instruction set shown in figure 21 has the same general format as the 1108 instructions, with the exception that it does not contain "partial word" bits or an index adjustment.

Example 10. The only difference between this example (see figure 22) and example 9 is that the cost constraint in this example is set to 20 instead of 10. This allows T43 to add two index specifications instead of one as in example 9.

Example 11. This example illustrates the procedure for adding an operator to ISDS in order to specify an index adjustment. The goal is to add an index adjustment to the instruction set in example 9. The new operators required are as follows:

T110 ADDAN ADJUSTMENT SWITCH TO THE FIRST INDEX ON EACH RIGHT OP OF COMPUTER (0). T110 is applicable only if the number of available bits, as determined by T42, is greater than zero and if the partially-specified instruction set contains no index adjustment. If both conditions are satisfied the simple instructions of computer (0) are generated to a subroutine which adds an adjustment switch to the first index on each right operand.

T111 RESTORE COMPUTER (0) FROM T110. Deletes the adjustment switch from the first index of each right operand.
OUTPUT FOR EXAMPLE 10

COMPUTER: ((MEM $K32) CODE (ORDERS) WORD ($N36) BYTE ($N8));
ORDERS: (GROUP1, GROUP2, GROUP3, GROUP4);
GROUP1: ('I1);
I1: ((OPR (+,-,*,/), COMP, ABS)) RESULT ('ADR1));
ADR1: ((ACCESS ((DATA ((FIX))))));
GROUP2: ('I2);
I2: ((OPR (+,-,*,/), COMP, ABS)) RESULT ('ADR2));
ADR2: ((ACCESS ((DATA ((FLOAT))))));
GROUP3: ('I3);
I3: ((OPR ((NEG, AND, OR), NOP)) RESULT ('ADR3));
ADR3: ((ACCESS ((DATA ((LOG))))));
GROUP4: ('I4);
I4: ((OPR ((MOVE)) RESULT ('ADR4));
ADR4: ((ACCESS ((DATA ((CHAR))))));
COMPUTER: 'T39..

COST CONSTRAINT: 20
INDEX VALUE: 10
INDEX COST: 10
MODE VALUE: 20
MODE COST: 0
LEFTOP VALUE: 10
LEFTOP COST: 0
I/O VALUE: 50
I/O COST: 0
OPERATION VALUE: 1
OPERATION COST: 0
C31: 1
C32: 0
C33: 1
C34: 0
C35: 1
C36: 0
C37: 1
C38: 0
C39: 1
C40: 0
C41: 1
C42: 0
C43: 1
C44: 0
C45: 1
C46: 0

STRATEGY:

OPERATOR       TOTAL VALUE       TOTAL COST

T61            90               0

Figure 22
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T43</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>160</td>
<td>0</td>
</tr>
<tr>
<td>T77</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td>T78</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>T68</td>
<td>194</td>
<td>10</td>
</tr>
<tr>
<td>T68</td>
<td>204</td>
<td>20</td>
</tr>
<tr>
<td>T73</td>
<td>212</td>
<td>20</td>
</tr>
<tr>
<td>T79</td>
<td>220</td>
<td>20</td>
</tr>
<tr>
<td>T82</td>
<td>226</td>
<td>20</td>
</tr>
<tr>
<td>T83</td>
<td>232</td>
<td>20</td>
</tr>
<tr>
<td>T72</td>
<td>236</td>
<td>20</td>
</tr>
<tr>
<td>T74</td>
<td>240</td>
<td>20</td>
</tr>
</tbody>
</table>

Figure 22 (continued)
MEMORY SIZE IS 65536 WORDS
WORD SIZE IS 36 BITS
BYTE SIZE IS 8 BITS

****INSTRUCTION GROUP 1****
INSTRUCTION LENGTH 36 BITS

TOP CODE | LEFTOP | RIGHTOP

BITS 0 TO 5
OP CODE

-----OPERATION GROUP 1-----
RIGHTOP: FIXED POINT; LEFTOP: REG;

OPERATION     CODE
+            000000
-            000001
*            000010
/            000011
COMP          000100
ABS           000101
-ABS          000110
TALLY         000111
C53           001000
C54           001001
C51           001010
C52           001011

BITS 6 TO 9
LEFTOP*DISPLACEMENT*INTEGER*

BITS 10 TO 13
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 14 TO 17
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 18 TO 18
RIGHTOP*OP DEFAUL*MODE*

DIRECT
INDIRECT 0
1

BITS 19 TO 34
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 22 (continued)
### Instruction Group 2

**Instruction Length 36 Bits**

**Op Code (left to right):**

**Operation Group 1**

**Rightop:** Float Point; **Leftop:** Reg;

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>001100</td>
</tr>
<tr>
<td>-</td>
<td>001101</td>
</tr>
<tr>
<td>*</td>
<td>001110</td>
</tr>
<tr>
<td>/</td>
<td>001111</td>
</tr>
<tr>
<td>COMP</td>
<td>010000</td>
</tr>
<tr>
<td>AHS</td>
<td>010001</td>
</tr>
<tr>
<td>NORM</td>
<td>010010</td>
</tr>
<tr>
<td>C57</td>
<td>010011</td>
</tr>
<tr>
<td>C58</td>
<td>010100</td>
</tr>
<tr>
<td>C55</td>
<td>010101</td>
</tr>
<tr>
<td>C56</td>
<td>010110</td>
</tr>
</tbody>
</table>

**Bits 6 to 9:**

<table>
<thead>
<tr>
<th>Leftop<em>Disp</em>Integer*</th>
</tr>
</thead>
</table>

**Bits 10 to 13:**

| Rightop*Index*Address*Disp*Integer* |

**Bits 14 to 17:**

| Rightop*Index*Address*Disp*Integer* |

**Bits 18 to 18:**

**Rightop*Op Detail*Mode**

<table>
<thead>
<tr>
<th>Direct</th>
<th>Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bits 19 to 34:**

| Rightop*Disp*Integer* |

---

Figure 22 (continued)
### Instruction Group 3

**Instruction Length 36 bits**

<table>
<thead>
<tr>
<th>OP Code</th>
<th>LeftTop</th>
<th>RightTop</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OP CODE</td>
<td>LEFTOP</td>
</tr>
</tbody>
</table>

#### Bits 0 to 5

**Operation Group 1**

**Operation Code**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>010111</td>
</tr>
<tr>
<td>AND</td>
<td>011000</td>
</tr>
<tr>
<td>OR</td>
<td>011001</td>
</tr>
<tr>
<td>NOP</td>
<td>011010</td>
</tr>
<tr>
<td>SHIFT</td>
<td>011011</td>
</tr>
<tr>
<td>TRUE</td>
<td>011100</td>
</tr>
<tr>
<td>V13</td>
<td>011101</td>
</tr>
<tr>
<td>V14</td>
<td>011110</td>
</tr>
<tr>
<td>V16</td>
<td>011111</td>
</tr>
<tr>
<td>V17</td>
<td>100000</td>
</tr>
<tr>
<td>V18</td>
<td>100001</td>
</tr>
<tr>
<td>V19</td>
<td>100010</td>
</tr>
<tr>
<td>V21</td>
<td>100011</td>
</tr>
<tr>
<td>V23</td>
<td>100100</td>
</tr>
<tr>
<td>V24</td>
<td>100101</td>
</tr>
<tr>
<td>V25</td>
<td>100110</td>
</tr>
<tr>
<td>FALSE</td>
<td>100111</td>
</tr>
<tr>
<td>C59</td>
<td>101000</td>
</tr>
<tr>
<td>C60</td>
<td>101001</td>
</tr>
</tbody>
</table>

#### Bits 6 to 9

**LeftOp: Displacement Integer**

#### Bits 10 to 13

**RightOp: Index Address Displacement Integer**

#### Bits 14 to 17

**RightOp: Index Address Displacement Integer**

#### Bits 18 to 19

**RightOp: CP Detail Mode**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT</td>
<td>0</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Bits 19 to 34

**RightOp: Displacement Integer**

---

Figure 22 (continued)
## Instruction Group 4

**Instruction Length**: 36 Bits

### Bits 0 to 5

- **Op Code**: LEFTOP | RIGHTOP

### Bits 6 to 9

- LEFTOP | DISPLACEMENT | INTEGER

### Bits 10 to 13

- RIGHTOP | INDEX | ADDRESS | DISPLACEMENT | INTEGER

### Bits 14 to 17

- RIGHTOP | INDEX | ADDRESS | DISPLACEMENT | INTEGER

### Bits 18 to 19

- RIGHTOP | OP | DETAIL | MODE

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>101010</td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
</tr>
<tr>
<td>EXEC</td>
<td>101100</td>
</tr>
<tr>
<td>COMP</td>
<td>101101</td>
</tr>
<tr>
<td>C61</td>
<td>101110</td>
</tr>
<tr>
<td>C62</td>
<td>101111</td>
</tr>
</tbody>
</table>

### Bits 19 to 34

- RIGHTOP | DISPLACEMENT | INTEGER

---

**Figure 22 (continued)**
*****INSTRUCTION GROUP 5*****
INSTRUCTION LENGTH 36 BITS

BITS 0 TO 5
OP CODE

-----OPERATION GROUP 1-----
OPERATION CODE
BRANCH

BITS 6 TO 9
CONDITION

BITS 10 TO 25
IF DISPLACEMENT INTEGER

Figure 22 (continued)
### Instruction Group 6

**Instruction Length:** 36 bits

<table>
<thead>
<tr>
<th>Bits 0 to 5</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>100001</td>
</tr>
<tr>
<td>I/O</td>
<td>100010</td>
</tr>
<tr>
<td>I/O</td>
<td>100011</td>
</tr>
<tr>
<td>I/O</td>
<td>110100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 6 to 9</th>
<th>Left: Displacement, Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 10 to 13</td>
<td>Right: Index, Address, Displacement, Integer</td>
</tr>
<tr>
<td>Bits 14 to 17</td>
<td>Right: Index, Address, Displacement, Integer</td>
</tr>
<tr>
<td>Bits 18 to 19</td>
<td>Right: OP Detail, Mode</td>
</tr>
</tbody>
</table>

| Bits 19 to 34 | Right: Displacement, Integer |

**Table:**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>100001</td>
</tr>
<tr>
<td>I/O</td>
<td>100010</td>
</tr>
<tr>
<td>I/O</td>
<td>100011</td>
</tr>
<tr>
<td>I/O</td>
<td>110100</td>
</tr>
</tbody>
</table>

*Figure 22 (continued)*
The steps required to use these operators in the heuristic program are as follows:

1. Load T110 and T111
2. Add the symbol T110 to V90, the list of operators executed by T76.
3. Make T111 the value of symbol T110 on the description list of V90 to identify T111 as the restore routine for T110.
4. Add code to the evaluation routine, T51, to evaluate an adjustment. This means adding two new coefficients C47 (value) and C48 (cost) for an adjustment switch. These coefficients must also be added to a list for T190 which prints initial values of the coefficients. The evaluation strategy is to add the cost and value coefficients to their respective totals if and only if the partially specified instruction set contains at least one adjustment switch.

No change is required in the counting routines or the output routines because these routines provide for all GIS features.

The results of adding the above operators to the heuristic program are illustrated in figure 23. The instruction set in figure 23 has all of the parts of the 1108 instruction format except for the 4 partial-word bits.

Example 12. This example shows the final step required to get ISDS to design the instruction format of the 1108. The only 1108 feature which does not appear in figure 23, example 11, is the partial word feature. In this example we describe an operator that adds the partial word feature and the steps needed to add the operator to our
OUTPUT FOR EXAMPLE 11

```plaintext
COMPUTER=({MEM,32,CODE,ORDERS,WORD,36, BYTE,8})
ORDERS=({GROUP1,GROUP2,GROUP3,GROUP4})
GROUP1=('11')
I1=({OPR,+,*,/,COMP,ABS}) RESULT ('ADR1'))
ADR1=({ACCESS,DATA,(FIX)})
GROUP2=('12')
I2=({OPR,+,*,/,COMP,ABS}) RESULT ('ADR2'))
ADR2=({ACCESS,DATA,(FLOAT)})
GROUP3=('13')
I3=({OPR,NEG,AND,JR,NOP}) RESULT ('ADR3'))
ADR3=({ACCESS,DATA,(LOG)})
GROUP4=('14')
I4=({OPR,MOVE}) RESULT ('ADR4'))
ADR4=({ACCESS,DATA,(CHAR)})
```

<table>
<thead>
<tr>
<th>COST CONSTRAINT</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX VALUE</td>
<td>10</td>
</tr>
<tr>
<td>INDEX VALUE</td>
<td>10</td>
</tr>
<tr>
<td>MODE VALUE</td>
<td>20</td>
</tr>
<tr>
<td>MODE Cost</td>
<td>0</td>
</tr>
<tr>
<td>LEFT TOP VALUE</td>
<td>10</td>
</tr>
<tr>
<td>LEFT TOP COST</td>
<td>0</td>
</tr>
<tr>
<td>I/O VALUE</td>
<td>50</td>
</tr>
<tr>
<td>I/O COST</td>
<td>0</td>
</tr>
<tr>
<td>OPERATION VALUE</td>
<td>1</td>
</tr>
<tr>
<td>OPERATION COST</td>
<td>0</td>
</tr>
<tr>
<td>C31</td>
<td>1</td>
</tr>
<tr>
<td>C32</td>
<td>1</td>
</tr>
<tr>
<td>C33</td>
<td>1</td>
</tr>
<tr>
<td>C34</td>
<td>0</td>
</tr>
<tr>
<td>C35</td>
<td>1</td>
</tr>
<tr>
<td>C36</td>
<td>0</td>
</tr>
<tr>
<td>C37</td>
<td>1</td>
</tr>
<tr>
<td>C38</td>
<td>0</td>
</tr>
<tr>
<td>C39</td>
<td>1</td>
</tr>
<tr>
<td>C40</td>
<td>0</td>
</tr>
<tr>
<td>C41</td>
<td>1</td>
</tr>
<tr>
<td>C42</td>
<td>0</td>
</tr>
<tr>
<td>C43</td>
<td>1</td>
</tr>
<tr>
<td>C44</td>
<td>0</td>
</tr>
<tr>
<td>C45</td>
<td>1</td>
</tr>
<tr>
<td>C46</td>
<td>0</td>
</tr>
<tr>
<td>C47</td>
<td>10</td>
</tr>
<tr>
<td>C48</td>
<td>0</td>
</tr>
</tbody>
</table>

**STRATEGY:**

Figure 23
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T61</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>T43</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>160</td>
<td>0</td>
</tr>
<tr>
<td>T77</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td>T78</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>T68</td>
<td>194</td>
<td>10</td>
</tr>
<tr>
<td>T110</td>
<td>204</td>
<td>10</td>
</tr>
<tr>
<td>T73</td>
<td>212</td>
<td>10</td>
</tr>
<tr>
<td>T79</td>
<td>220</td>
<td>10</td>
</tr>
<tr>
<td>T82</td>
<td>226</td>
<td>10</td>
</tr>
<tr>
<td>T83</td>
<td>232</td>
<td>10</td>
</tr>
<tr>
<td>T72</td>
<td>236</td>
<td>10</td>
</tr>
<tr>
<td>T74</td>
<td>240</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 23 (continued)
Memory size is 65536 words
Word size is 36 bits
Byte size is 8 bits

Instruction length 36 bits

***Instruction Group 1***

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>LEFTOP</th>
<th>RIGHTOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>000001</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>000010</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>000011</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>COMP</td>
<td></td>
</tr>
<tr>
<td>001001</td>
<td>ABS</td>
<td></td>
</tr>
<tr>
<td>001010</td>
<td>-ABS</td>
<td></td>
</tr>
<tr>
<td>001111</td>
<td>TALLY</td>
<td></td>
</tr>
<tr>
<td>001000</td>
<td>C53</td>
<td></td>
</tr>
<tr>
<td>001001</td>
<td>C54</td>
<td></td>
</tr>
<tr>
<td>001010</td>
<td>C51</td>
<td></td>
</tr>
<tr>
<td>001011</td>
<td>C52</td>
<td></td>
</tr>
</tbody>
</table>

Figure 23 (continued)
### Instruction Group 2

**Instruction Length:** 36 bits

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>001100</td>
</tr>
<tr>
<td>-</td>
<td>001101</td>
</tr>
<tr>
<td>*</td>
<td>001110</td>
</tr>
<tr>
<td>/</td>
<td>001111</td>
</tr>
<tr>
<td>COMP</td>
<td>010000</td>
</tr>
<tr>
<td>ABS</td>
<td>010001</td>
</tr>
<tr>
<td>NORM</td>
<td>010010</td>
</tr>
<tr>
<td>C57</td>
<td>010101</td>
</tr>
<tr>
<td>C58</td>
<td>010110</td>
</tr>
<tr>
<td>C55</td>
<td>010111</td>
</tr>
<tr>
<td>C56</td>
<td>010110</td>
</tr>
</tbody>
</table>

- **Bits 0 to 5:** OP code
- **Operation Group 1:**
  - **Rightop:** Float point; **Leftop:** Reg;

- **Bits 6 to 9:**
  - Leftop: Displacement: Integer
- **Bits 10 to 10:** Rightop: Index: Adjustment: Switch
  - On: 0
  - Off: 1
- **Bits 11 to 14:** Rightop: Index: Address: Displacement: Integer
- **Bits 15 to 15:** Rightop: Op Detail: Mode
  - Direct: 0
  - Indirect: 1
- **Bits 16 to 31:** Rightop: Displacement: Integer

---

*Figure 23 (continued)*
*****INSTRUCTION GROUP 3*****
INSTRUCTION LENGTH 36 BITS

OP CODE | LEFTOP | RIGHTOP
--- | --- | ---

---OPERATION GROUP 1-----
RIGHTOP: LOGIC; LEFTOP: REG;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>010111</td>
</tr>
<tr>
<td>AND</td>
<td>011000</td>
</tr>
<tr>
<td>OR</td>
<td>011001</td>
</tr>
<tr>
<td>NOP</td>
<td>011010</td>
</tr>
<tr>
<td>SHIFT</td>
<td>011110</td>
</tr>
<tr>
<td>TRUE</td>
<td>011111</td>
</tr>
<tr>
<td>V13</td>
<td>011101</td>
</tr>
<tr>
<td>V14</td>
<td>011110</td>
</tr>
<tr>
<td>V16</td>
<td>011111</td>
</tr>
<tr>
<td>V17</td>
<td>100000</td>
</tr>
<tr>
<td>V18</td>
<td>100001</td>
</tr>
<tr>
<td>V19</td>
<td>100010</td>
</tr>
<tr>
<td>V21</td>
<td>100011</td>
</tr>
<tr>
<td>V23</td>
<td>100100</td>
</tr>
<tr>
<td>V24</td>
<td>100101</td>
</tr>
<tr>
<td>V25</td>
<td>100110</td>
</tr>
<tr>
<td>FALSE</td>
<td>100111</td>
</tr>
<tr>
<td>C59</td>
<td>101000</td>
</tr>
<tr>
<td>C60</td>
<td>101001</td>
</tr>
</tbody>
</table>

BITS 6 TO 9
LEFTOP*DISPLACEMENT*INTEGER*

BITS 10 TO 10
RIGHTOP*INDEX*ADJUSTMENT*SWITCH*
ON 0
OFF 1

BITS 11 TO 14
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER*

BITS 15 TO 15
RIGHTOP*OP DETAIL*MODE*
DIRECT 0
INDIRECT 1

BITS 16 TO 31
RIGHTOP*DISPLACEMENT*INTEGER*

Figure 23 (continued)
**********INSTRUCTION GROUP 4********
INSTRUCTION LENGTH 36 BITS

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>LEFT TOP</th>
<th>RIGHT TOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>101010</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>101100</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>101101</td>
<td></td>
</tr>
<tr>
<td>CA1</td>
<td>101110</td>
<td></td>
</tr>
<tr>
<td>CA2</td>
<td>101111</td>
<td></td>
</tr>
</tbody>
</table>

BITS 0 TO 5

OP CODE

RIGHT TOP: BYTE; LEFT TOP: REG

OPERATION CODE

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVE</td>
<td>101010</td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
</tr>
<tr>
<td>EXEC</td>
<td>101100</td>
</tr>
<tr>
<td>COMP</td>
<td>101101</td>
</tr>
<tr>
<td>CA1</td>
<td>101110</td>
</tr>
<tr>
<td>CA2</td>
<td>101111</td>
</tr>
</tbody>
</table>

BITS 6 TO 9

LEFT TOP * DISPLACEMENT * INTEGER*

BITS 10 TO 10

RIGHT TOP * INDEX * ADJUSTMENT * SWITCH*

ON
OFF

BITS 11 TO 14

RIGHT TOP * INDEX * ADDRESS * DISPLACEMENT * INTEGER*

BITS 15 TO 15

RIGHT TOP * OP DETAIL * MODE*

DIRECT
INDIRECT

BITS 16 TO 31

RIGHT TOP * DISPLACEMENT * INTEGER*

Figure 23 (continued)
Figure 23 (continued)
*****INSTRUCTION GROUP 6*****
INSTRUCTION LENGTH 36 BITS

<table>
<thead>
<tr>
<th>OP CODE</th>
<th>LEFTOP</th>
<th>RIGHTOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>110001</td>
<td>I/O</td>
</tr>
<tr>
<td>I/O</td>
<td>110010</td>
<td>I/O</td>
</tr>
<tr>
<td>I/O</td>
<td>110011</td>
<td>I/O</td>
</tr>
</tbody>
</table>

**OPERATION GROUP 1**

RIGHTOP: BYTE; LEFTOP: REG;

<table>
<thead>
<tr>
<th>OPERATION CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
</tr>
<tr>
<td>I/O</td>
</tr>
<tr>
<td>I/O</td>
</tr>
<tr>
<td>I/O</td>
</tr>
</tbody>
</table>

BITS 6 TO 9
LEFTOP: DISPLACEMENT INTEGER

BITS 10 TO 10
RIGHTOP: INDEX ADJUSTMENT SWITCH

BITS 11 TO 14
RIGHTOP: INDEX ADDRESS DISPLACEMENT INTEGER

BITS 15 TO 15
RIGHTOP: OP DETAIL MODE

BITS 16 TO 31
RIGHTOP: DISPLACEMENT INTEGER

DIRECT

INDIRECT

Figure 23 (continued)
heuristic program,

T112  **ADD PARTIAL WORD PROCESSES TO COMPUTER (0)**. Not applicable if computer (0) contains partial word processes or if the number of available bits as determined by T42 is not positive. Otherwise T112 adds a partial word specification to each simple instruction that contains a right operand. The number of bits for partial word processes is 4 or the number of available bits, whichever is smaller.

T113  **RESTORE COMPUTER (0) FROM T112**. Deletes the partial word specification from every simple instruction of computer (0) that contains a right operand.

The steps required to add these operators to the heuristic program were as follows:

1. Load T112 and T113.
2. Add T112 to V90, the operator list for T76.
3. Make T113 the value of attribute T112 on the description list of V90.
4. Add an evaluation strategy to T51 for partial word processes. The new value and cost coefficients, C71 and C72, are added to their respective totals if and only if the partial word symbol, C73, occurs on the partially specified instruction set.
5. The coefficients C71 and C72 must be added to the print list for T90.
6. The partial word symbol, C73, must be added to list D71 to indicate that every occurrence of C73 has description form field description.
7. The words "partial word process" must be assigned as the translation of C73 on the translation tree, D90.

Items 6 and 7 above are necessary because partial word processes
are not included in GIS. Once the symbol C73 has been added to D71, however, all of the counting routines include partial word bits.

The final ISDS version of the 1108 is illustrated in figure 24. This instruction set has a limited number of operations but its instruction parts are identical to the 1108.
OUTPUT FOR EXAMPLE 12

```
COMPUTER=((MEB (132) CODE (ORDERS) WORD (N36) BYTE (N8)));
ORDERS=('GROUP1', 'GROUP2', 'GROUP3', 'GROUP4');
GROUP1=('1111');
I1=((OPR ((-, -, *, /, COMP, ABS)) RESULT ('ADR1')));
ADR1=((ACCESS (((DATA ((FIX)))))));
GROUP2=('1212');
I2=((OPR ((-, -, *, /, COMP, ABS)) RESULT ('ADR2')));
ADR2=((ACCESS (((DATA ((FLOAT)))))));
GROUP3=('1313');
I3=((OPR ((NE3, AND, OR, NOP)) RESULT ('ADR3')));
ADR3=((ACCESS (((DATA ((LOG)))))));
GROUP4=('1414');
I4=((OPR ((MOVE)) RESULT ('ADR4')));
ADR4=((ACCESS (((DATA ((CHAR)))))));
```

| COST CONSTRAINT | 10 |
| INDEX VALUE     | 10 |
| INDEX COST      | 10 |
| MODE VALUE      | 20 |
| MODE COST       | 0  |
| LEFTTOP VALUE   | 10 |
| LEFTTOP COST    | 0  |
| I/O VALUE       | 50 |
| I/O COST        | 0  |
| OPERATION VALUE | 1  |
| OPERATION COST  | 0  |
| C31              | 1  |
| C32              | 0  |
| C33              | 1  |
| C34              | 0  |
| C35              | 1  |
| C36              | 0  |
| C37              | 1  |
| C38              | 0  |
| C39              | 1  |
| C40              | 0  |
| C41              | 1  |
| C42              | 0  |
| C43              | 1  |
| C44              | 0  |
| C45              | 1  |
| C46              | 0  |
| C47              | 10 |
| C48              | 0  |

Figure 24
<table>
<thead>
<tr>
<th>OPERATOR</th>
<th>TOTAL VALUE</th>
<th>TOTAL COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>T61</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>T43</td>
<td>140</td>
<td>0</td>
</tr>
<tr>
<td>T56</td>
<td>160</td>
<td>0</td>
</tr>
<tr>
<td>T77</td>
<td>172</td>
<td>0</td>
</tr>
<tr>
<td>T78</td>
<td>184</td>
<td>0</td>
</tr>
<tr>
<td>T68</td>
<td>194</td>
<td>10</td>
</tr>
<tr>
<td>T110</td>
<td>204</td>
<td>10</td>
</tr>
<tr>
<td>T73</td>
<td>212</td>
<td>10</td>
</tr>
<tr>
<td>T79</td>
<td>220</td>
<td>10</td>
</tr>
<tr>
<td>T82</td>
<td>226</td>
<td>10</td>
</tr>
<tr>
<td>T83</td>
<td>232</td>
<td>10</td>
</tr>
<tr>
<td>T72</td>
<td>236</td>
<td>10</td>
</tr>
<tr>
<td>T74</td>
<td>240</td>
<td>10</td>
</tr>
<tr>
<td>T112</td>
<td>241</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 24 (continued)
MEMORY SIZE IS 65536 WORDS
WORD SIZE IS 36 BITS
BYTE SIZE IS 8 BITS

*****INSTRUCTION GROUP 1*****
INSTRUCTION LENGTH 36 BITS

BITS 0 TO 5
OP CODE

-----OPERATION GROUP 1-----
RIGHTOP: FIXED POINT; LEFTOP: REG;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>000000</td>
</tr>
<tr>
<td>-</td>
<td>000001</td>
</tr>
<tr>
<td>*</td>
<td>000010</td>
</tr>
<tr>
<td>/</td>
<td>000011</td>
</tr>
<tr>
<td>COMP</td>
<td>000100</td>
</tr>
<tr>
<td>ABS</td>
<td>000101</td>
</tr>
<tr>
<td>-ABS</td>
<td>000110</td>
</tr>
<tr>
<td>TALLY</td>
<td>000111</td>
</tr>
<tr>
<td>C33</td>
<td>001000</td>
</tr>
<tr>
<td>C54</td>
<td>001001</td>
</tr>
<tr>
<td>C51</td>
<td>001010</td>
</tr>
<tr>
<td>C52</td>
<td>001011</td>
</tr>
</tbody>
</table>

BITS 6 TO 9
PARTIAL WORD PROCESS

BITS 10 TO 13
LEFTOP*DISPLACEMENT*INTEGER

BITS 14 TO 14
RIGHTOP*INDEX*ADJUSTMENT*SWITCH
ON 0
OFF 1

BITS 15 TO 16
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER

BITS 19 TO 19
RIGHTOP*OP DETAIL*MODE
DIRECT 0
INDIRECT 1

BITS 20 TO 35
RIGHTOP*DISPLACEMENT*INTEGER

Figure 24 (continued)
---INSTRUCTION GROUP 2---
INSTRUCTION LENGTH 36 BITS

TOP CODE: LEFTTOP | RIGHTTOP

### BITS 0 TO 5

**OP CODE**

### OPERATION GROUP 1

RIGHTTOP: FLOAT POINT; LEFTTOP: REG;

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>001100</td>
</tr>
<tr>
<td>-</td>
<td>001101</td>
</tr>
<tr>
<td>*</td>
<td>001110</td>
</tr>
<tr>
<td>/</td>
<td>001111</td>
</tr>
<tr>
<td>COMP</td>
<td>010000</td>
</tr>
<tr>
<td>ABS</td>
<td>010001</td>
</tr>
<tr>
<td>NORM</td>
<td>010010</td>
</tr>
<tr>
<td>C57</td>
<td>010011</td>
</tr>
<tr>
<td>C58</td>
<td>010100</td>
</tr>
<tr>
<td>C59</td>
<td>010101</td>
</tr>
<tr>
<td>C56</td>
<td>010110</td>
</tr>
</tbody>
</table>

### BITS 6 TO 9

PARTIAL WORD PROCESS

### BITS 10 TO 13

LEFTTOP | DISPLACEMENT | INTEGER

### BITS 14 TO 14

RIGHTTOP | INDEX | ADJUSTMENT | SWITCH

<table>
<thead>
<tr>
<th>ON</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### BITS 15 TO 18

RIGHTTOP | INDEX | ADDRESS | DISPLACEMENT | INTEGER

### BITS 19 TO 19

RIGHTTOP | OP DETAIL | MODE

<table>
<thead>
<tr>
<th>DIRECT</th>
<th>INDIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### BITS 20 TO 35

RIGHTTOP | DISPLACEMENT | INTEGER

---

Figure 24 (continued)
### INSTRUCTION GROUP 3

**INSTRUCTION LENGTH 36 BITS**

** Bits 0 to 5 **

**OP: CODE | LEFT: LEFT | RIGHT: RIGHT **

** OPERATION GROUP 1 **

**RIGHT: LOGIC | LEFT: REG **

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>010111</td>
</tr>
<tr>
<td>AND</td>
<td>011000</td>
</tr>
<tr>
<td>OR</td>
<td>011011</td>
</tr>
<tr>
<td>NOP</td>
<td>011001</td>
</tr>
<tr>
<td>SHIFT</td>
<td>011010</td>
</tr>
<tr>
<td>TRUE</td>
<td>011100</td>
</tr>
<tr>
<td>V13</td>
<td>011101</td>
</tr>
<tr>
<td>V14</td>
<td>011110</td>
</tr>
<tr>
<td>V16</td>
<td>011111</td>
</tr>
<tr>
<td>V17</td>
<td>100000</td>
</tr>
<tr>
<td>V18</td>
<td>100001</td>
</tr>
<tr>
<td>V19</td>
<td>100010</td>
</tr>
<tr>
<td>V21</td>
<td>100011</td>
</tr>
<tr>
<td>V23</td>
<td>100100</td>
</tr>
<tr>
<td>V24</td>
<td>100101</td>
</tr>
<tr>
<td>V25</td>
<td>100110</td>
</tr>
<tr>
<td>FALSE</td>
<td>100111</td>
</tr>
<tr>
<td>C59</td>
<td>101000</td>
</tr>
<tr>
<td>C60</td>
<td>101001</td>
</tr>
</tbody>
</table>

** Bits 6 to 9 **  

**PARTIAL WORD ADDRESS**

** Bits 10 to 13 **  

**LEFT: LEFT | DISPLACEMENT: INTEGER**

** Bits 14 to 14 **  

**RIGHT: RIGHT | INDEX: ADJUSTMENT | SWITCH**

<table>
<thead>
<tr>
<th>ON</th>
<th>OFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

** Bits 15 to 18 **  

**RIGHT: RIGHT | INDEX: ADDRESS | DISPLACEMENT: INTEGER**

** Bits 19 to 19 **  

**RIGHT: RIGHT | OP: DETAIL | MODE**

<table>
<thead>
<tr>
<th>DIRECT</th>
<th>INDIRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

** Bits 20 to 35 **  

**RIGHT: RIGHT | DISPLACEMENT: INTEGER**

---

*Figure 24 (continued)*
### Instruction Group 4

**Instruction Length:** 36 bits

<table>
<thead>
<tr>
<th>Bits 0 to 5</th>
<th>Operation</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEFTOP: REG;</td>
<td>MOVE</td>
<td>101010</td>
</tr>
<tr>
<td>STORE</td>
<td>101011</td>
<td></td>
</tr>
<tr>
<td>EXEC</td>
<td>101100</td>
<td></td>
</tr>
<tr>
<td>COMP</td>
<td>101101</td>
<td></td>
</tr>
<tr>
<td>C61</td>
<td>101110</td>
<td></td>
</tr>
<tr>
<td>C62</td>
<td>101111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bits 6 to 9</th>
<th>Partial Word Process*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits 10 to 13</td>
<td>Leftop: Displacement: Integer*</td>
</tr>
<tr>
<td>Bits 14 to 14</td>
<td>Rightop: Index: Adjustment: Switch*</td>
</tr>
</tbody>
</table>

**On:** 0  
**Off:** 1

| Bits 15 to 18 | Rightop: Index: Address: Displacement: Integer* |
| Bits 19 to 19 | Rightop: Op Detail: Mode* |

**Direct:** 0  
**Indirect:** 1

| Bits 20 to 35 | Rightop: Displacement: Integer* |

---

*Figure 24 (Continued)*
<table>
<thead>
<tr>
<th>Bits 0 to 5</th>
<th>Condition</th>
<th>Bits 6 to 9</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP CODE</td>
<td></td>
<td>BRNCH</td>
<td>110000</td>
</tr>
</tbody>
</table>

Figure 24 (continued)
*****INSTRUCTION GROUP 6*****
INSTRUCTION LENGTH 36 BITS

BITS 0 TO 5
OP CODE

---OPERATION GROUP 1---
RIGHTOP: BYTE; LEFTOP: REG;

OPERATION CODE

<table>
<thead>
<tr>
<th>OPERATION</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O</td>
<td>110001</td>
</tr>
<tr>
<td>I/O</td>
<td>110010</td>
</tr>
<tr>
<td>I/O</td>
<td>110011</td>
</tr>
<tr>
<td>I/O</td>
<td>110100</td>
</tr>
</tbody>
</table>

BITS 6 TO 9
PARTIAL WORD PROCESS

BITS 10 TO 13
LEFTOP*DISPLACEMENT*INTEGER

BITS 14 TO 14
RIGHTOP*INDEX*ADJUSTMENT SWITCH

<table>
<thead>
<tr>
<th>VALUE</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ON</td>
<td>0</td>
</tr>
<tr>
<td>OFF</td>
<td>1</td>
</tr>
</tbody>
</table>

BITS 15 TO 18
RIGHTOP*INDEX*ADDRESS*DISPLACEMENT*INTEGER

BITS 19 TO 19
RIGHTOP*OP DETAIL*MODE

<table>
<thead>
<tr>
<th>MODE</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT</td>
<td>0</td>
</tr>
<tr>
<td>INDIRECT</td>
<td>1</td>
</tr>
</tbody>
</table>

BITS 20 TO 35
RIGHTOP*DISPLACEMENT*INTEGER

Figure 24 (continued)
EVALUATION OF ISDS

In this section we will attempt to evaluate ISDS with respect to the following criteria:

1. Generality
2. Ability to cope with complexity of the design problem
3. Ability to match an instruction set to a specified task environment
4. Usefulness
5. The system as a model of a design process

Generality

GIS gives ISDS the ability to represent a wide range of instruction sets. GIS contains most of the addressing features that are used in existing computers. The primary limitation of GIS, as the examples show, is its restricted set of operations. It would be difficult to extend the set of operations meaningfully since in most cases the operations not contained in GIS are ad hoc operations used in relatively few computers. These operations are frequently designed to take advantage of special hardware features or peculiarities in the implementation of a computer. ISDS's shortage of operations could be overcome by a mechanism for defining new operations or by an operator that reserves part of an instruction format for unnamed operation
Example 12 shows that it can be relatively simple to incorporate instruction parts that do not occur in GIS. In some cases new parts can be added by placing their symbols on the list associated with the appropriate method for processing the part.

Although ISDS can represent a wide variety of instruction sets, there are limits on the range of instruction sets that ISDS can design. In the program described earlier the range of strategies is limited because word size and memory size are always taken as input parameters. Example 1 shows that this assumption may eliminate some useful strategies. The number of strategies that can be implemented by ISDS is also limited by the excessive time required at each stage of the optimization phase to select the optimal operator. Since the look-ahead is limited to one step (see discussion under Usefulness of ISDS) ISDS executes simple strategies. To discover more complex strategies ISDS should be provided with a mechanism for restricting the number of operators attempted at each stage of the optimization.

The range of instruction sets is also limited by assumptions made by some of the ISDS operators. The two most serious restrictions in this category are the reliance of a uniform operation code and uniform instruction formats. More complex operators would be required to design the IBM 360 or the IBM 7030 (STRETCH), since these machines have several instruction formats.

ISDS is extremely flexible with respect to the different types of operators that can be constructed. Any type of analysis or specification can be performed. The system is open in the sense that no structure is imposed on the set of operators. Utility opera-
tors can be constructed using the operators already in the system and can be added as required. Strategy operators can also be added easily; the name of each new strategy operator must be added to the search list for T76.

Ability of the System to Cope With the Complexity of the Design Problem

In theory, ISDS can be used to construct an operator to cope with any type of inter-relationship between design variables. In practice the amount of analysis that can be performed by each operator is restricted since each operator may be executed at least partially at each step of the design strategy. The practical limitations on operators are discussed in the section Usefulness of the System. Even though the operators perform relatively simple analysis, ISDS is able to design some relatively complex instruction sets (see example 12, for example) by a serial process that is similar to the process used by a human designer.

One drawback of the heuristic program described earlier is that the optimization loop contains no smaller, inner loops. When an operator is called it analyzes the partially-specified instruction set and specifies one new GIS part. Ideally each specification stage should contain some inner loops to guarantee that the operator specifies an optimal value for the new GIS part. For example when T68 -- add indexing -- is called it could call the evaluation routine for several different index specifications to determine an optimal index specification. This type of optimization would require a more elaborate evaluation routine, but it would add a new dimension to strategies determined by the program.
Ability to Match an Instruction Set to a Task Environment

ISDS is limited in this area by the difficulty of obtaining performance measurements for an instruction set. One motivation in the design of the search routine (T76) was to isolate the function of evaluation so that any evaluation strategy can be "plugged" into the system. The feature extraction, or mix, approach seems most promising for closed-loop design. Ultimately it is possible that evaluation could be performed by an automated GIS simulator, but, as we pointed out earlier, this approach requires a "program-writer."

The system can be useful even if present techniques, such as simulation, are used for evaluation. For example, the designer can pose a problem to ISDS on the basis of his knowledge of the requirements of a solution. ISDS returns solution to the problem in a matter of minutes. The ISDS solution can then be further specified and evaluated by any available method. If the evaluation process reveals weaknesses in the instruction set the designer can re-submit the problem to ISDS with minor changes in the input parameters. This use of ISDS is illustrated by examples 8-12. In this type of environment ISDS functions as an intelligent generator of feasible solutions. The savings in time to the designer is great, but no loss of performance occurs because of the difficulty of automatically evaluating an instruction set.

Usefulness of ISDS

One of the critical measurements of a system like ISDS is its practicality -- the cost of using the system weighed against the
time savings afforded by use of the system. The operators described in this thesis require approximately 7,000 IPL cells (14,000 G-20 words). The examples in Chapter VI require between 500,000 and 1,400,000 IPL interpretation cycles, or up to 35 minutes of G-20 execution time. Much of this time is spent in "applicability analysis" by operators. For this reason the time required to execute T76 for operator sequences of length greater than one is prohibitive. In one test with six operators T76 required over 12 minutes to select one operator sequence of length 2. The reason for this is that the number of attempted operator applications increases geometrically with the length of the operator sequence applied. Although the practical restriction to operation sequences of length one permits a wide spectrum of strategies, it rules out some interesting strategies. In some cases "look-ahead" is important. For example it is more advantageous to use an extra bit for 5 operations worth 2 points per operation than for indirect addressing at 8 points. T76 can detect such a strategy only if it uses operator sequences of length 5. Although this is impractical in the current implementation it is possible that a different organization could reduce the amount of analysis performed by each operator. One approach may be for the executive routine to perform analysis and make the results available to every operator. "Side effects" may make this approach difficult to implement, however.

Another objection to the present organization of the executive routine (T76) is that each operator is applied, or attempted, at every stage of the optimization. The program would be more efficient if it contained a method for selecting a few "high expectation"
operators to attempt at each stage of the optimization. This may be
difficult since presumably the only information about the desired
instruction set is provided by the scores computed by the evaluation
routine.

On the positive side, experiments with the system indicate that
it can relieve a designer of an enormous amount of "busy-work".
In one example the routine R3 (count symbols on list (0)) was executed
over 2,500 times. Even some of the higher level routines such as
T21 (find the number of operation codes) and T42 (find available bits)
may be executed hundreds of times in the course of a solution. This
savings in busy-work is one of the primary justifications for any
design system.

ISDS as a Model of a Design Process

One of the purposes of this research has been to provide some
general insight into the design process. The results of the research
support the conclusions that the models of design described in
Chapter IV can be taken as the basis for an automated design system.
ISDS conforms to the models of design in the following respects:

1. Input to ISDS expresses demands for solution to a problem,
   including constraint information.

2. The system makes use of a "design concept" -- GIS. In-
   formation about the design concept is contained in the
   strategy level operators.

3. An instruction set proceeds gradually from its initial form --
   the symbolic input -- toward a completed product. Each stage
   involves analysis and specification and, possibly, "de-
specification" and re-specification. Analysis is performed by operators as well as by the evaluation routine. All specification is performed by operators, as is "de-specification", or restoring, when required. Even though operator sequences are restricted by practical considerations to length 1, much specification and re-specification occur at T76 searches at each stage of the strategy for the operator that yields the most improvement in the instruction set.

The general design process of ISDS is illustrated in figure 25.

IMPLICATIONS OF THE RESEARCH

The most important conclusion that can be drawn from the research reported here is that a "closed" design system is feasible. This is not to say that interactive systems should be abandoned. The design problem considered here is not so complicated as many problems for which interactive systems are a valuable aid. The execution time required by ISDS suggests that current technology cannot accommodate closed systems for much more complex design problems. It is likely, however, that some time in the future closed systems will be feasible for problems that are presently solved with the aid of human interaction. In the remainder of this section we will consider three aspects of ISDS which may have general applicability in the design of closed design systems.

Model of a Solution

A certain amount of information about the solution space must be built into any closed design system. This information must not
"build out" useful solutions. Information can be "built in" by means of data structure, construction routines, and design strategy. The most non-restrictive approach seems to be to build in very general data structures and a minimum of strategy. This approach places solution information in the construction routines which are relatively self-contained and easily added or altered.

Operators

The main reason for ISDS's ability to design a broad range of instruction sets seems to be its use of very powerful operators. These operators do as much analysis as possible before specification. Each operator uses information contained in the partially specified instruction set to determine how it should alter the instruction set. This procedure minimizes the amount of search required of the system.

It is possible to distinguish three types of analysis in ISDS:

1. Applicability Analysis
2. Determination Analysis
3. Evaluation Analysis

Applicability Analysis is the activity performed by each operator as it determines whether or not it can be applied to the partially specified instruction set. In many cases, for example, it is necessary to test whether or not there are unused bits in the instruction format for a new specification.

Determination Analysis is analysis that precedes specification. Its purpose is to compute as accurately as possible from information in the partially specified instruction set the new value to be posited
by an operator. The importance in ISDS of this type of analysis suggests that the frequently used model of design as:

\[
\text{specification} \rightarrow \text{analysis} \quad \text{specification} \leftarrow \text{analysis}
\]

should be:

\[
\downarrow \text{analysis} \rightarrow \text{specification} \rightarrow \text{analysis}
\]

where the first type of analysis is determination analysis and the second is evaluation analysis.

**Statement of a Design Problem**

Any design system must provide a means for posing a design problem. In general this means that the user must have some method for stating his requirements and the constraints that a solution must satisfy. This can be a difficult problem since the user would like to state his requirements in a high-level language, whereas the design program must be able to translate the input specifications into specific criteria for a solution. For example, a program cannot design a "computer for commercial data processing" unless it has been given a great deal of information about computers and commercial data processing. In a design system it would be very difficult to maintain a store of knowledge large enough to permit users to state their requirements in a high-level language. One way to guarantee that the design system contains enough information about a user's demands is to express requirements in terms of a program that evaluates solutions to a problem. The design system uses the evaluation program to direct its search for an acceptable solution. Clearly there must be some rapport between the design system and the evaluation routine; the design system must not submit apples to an evaluation routine for
oranges. It seems likely, however, that the necessary rapport can be established at least within particular design environments.

The use of the evaluation routine in ISDS illustrates the above method for specifying the requirements of a solution. Any evaluation routine can be supplied by a user. However, the parameters of the evaluation routine T51 offer a simple means for a user to specify his requirements. T51 has the necessary rapport with the underlying design system because it is written in the same language and contains information about all objects which the system can construct. This method of problem specification may be useful in many problem-solving systems.

FUTURE WORK ON ISDS

In this section we will summarize the shortcomings of ISDS and propose ways that they can be overcome.

In all examples that have been run on ISDS the main difference between the ISDS instruction set and the desired instruction set has been in the set of operations. The best way to remedy this is probably to add a large number of operations to GIS.

GIS presently makes no attempt to find an efficient encoding for operations. An encoding scheme should try to minimize the effort required to interpret instructions. This may frequently mean that an instruction set should have more than one format for operation code. This raises the general point that the current ISDS makes some assumptions where a good user system should permit generality. For example, design strategies should be prepared to determine memory size
and word size, rather than requiring the values as input parameters. One useful approach to increasing the flexibility of ISDS may be to add a mechanism for building upon the existing system. The steps performed in examples 11 and 12 to add operators to the system could be performed by a program. This type of mechanism would be particularly desirable if the system were to be used interactively.

ISDS could be used in an interactive environment, but it would be impractical to perform high-level analysis or specification interactively with the present system because of the time required. Before the system could be useful interactively, at a high level, methods must be devised to reduce the search time required by the executive routine. This may be the most challenging problem. It seems desirable to use some technique like differences in GPS to give the search much more direction. Yet in the current version of ISDS the only information about the "desired object" is a single number obtained upon each evaluation. It seems that any attempt to direct the search would remove some strategy variability from the system. One feasible compromise may be to use the value coefficients to determine several operators at each stage which seem likely to result in greatest improvement. Certain operators, such as add indirect addressing, are unlikely to be applied more than one time and could be removed from consideration after one application. If the number of strategies considered at each stage of the optimization can be reduced to 3 or 4, the system can probably develop sophisticated strategies.

Another important area of experimentation is experimentation with evaluation techniques. The instruction mix approach described in this thesis can be refined by using multi-step strategies and more
groups in the instruction mix. Other evaluation techniques can be tested by means of ISDS. One especially interesting project would be to construct a simulator for all GIS instructions. Such a device would provide a common base for comparison of different GIS instruction sets.

THE FUTURE OF DESIGN SYSTEMS

Much experimentation will be required before closed design systems can be useful in practice. One natural vehicle for such experimentation is the interactive system. Closed design sub-systems can be embedded in an interactive system. The user-designer still provides information to the system, but, whenever possible, closed sub-systems intervene to perform part of the design strategy and return results to the user. A system organized on this basis can be used ultimately to construct a completely closed system.

In the past the computer's role in design has been that of a tool for performing analysis. It has been felt that the most useful attribute of the computer is its ability to perform large volumes of computation. However the computer is also a valuable search tool because it can construct and evaluate a large number of potential solutions in a short time period. The maximum benefit of computers to designers will be attained when computers perform both the analysis and the specification and search inherent in a design problem.
BIBLIOGRAPHY


Selections on Benchmark Results. *Computers and Automation* 15


35. Knight, K. E. A Fast Sort of Country. A Study of Technological
Carnegie Institute of Technology, Pittsburgh, Pa., 1965.

36. Kopp, C. G. and Wertz, J. B. Designer and Computer: An Engineer-
pp. 8-14.

37. Ledley, R. S. *Programming and Utilizing Digital Computers.*

38. Lesser, V. A Machine Organization that Separates the Function
of Computation from Data Access. Stanford University, Califor-
nia, 1967.

*Design Quarterly* (Design and the Computer) 66/67, pp. 31-39.

40. Mann, R. W. Engineering Specifications for a Man-Computer System


42. A Manual for the Rice Institute Computer. The Rice Institute,
Houston, Texas, 1958.


1964.

of Technology, Pittsburgh, Pa., 1967.

46. Nielsen, N. R. The Analysis of General Purpose Computer Time-
1966.

47. Operational Characteristics of The Processors for the Burroughs
B5000. Burroughs Manual 5000-21005-D.

Manual DED-10-GBAA-D.


