ISDS—A program that designs computer instruction sets

by F. M. HANEY

Scientific Data Systems
El Segundo, California

INTRODUCTION

ISDS (Instruction Set Design System), a program that designs instruction languages for computers, is the result of research aimed at gaining a better understanding of computer-assisted design and, in particular, automated design of computers. The primary goal of the research was to develop techniques for writing programs that solve design problems without intervention by human designers. This paper describes a program that solves a specific design problem—the selection of an order code for a computer—but the general approach can be easily adapted to other design problems.

ISDS contains a generalized model of a computer instruction set and solves a design problem by filling in details of the model, analyzing the result with respect to the requirements of the given problem, and selecting instances of the model that best meet the requirements of the problem.

The model used by ISDS is GIS (Generalized Instruction Set) which is capable of representing a broad range of computer instruction sets, including most of the features of existing computers.

The programs that make up ISDS operate at several levels, the lowest of which is used to manipulate the tree structures storing GIS representations of instruction sets. Other programs in ISDS perform computations useful in analysis of instruction sets, select values for single values of the instruction set, analyze an entire instruction set, and determine the optimal method of selecting parts of the instruction set.

This paper is organized into four sections:
1. A discussion of design theory and the basis for the ISDS approach to constructing design programs.
2. A description of GIS, the model of an instruction set that ISDS uses as its basic design concept in solving a problem.
3. A description of the programs that make up ISDS and the actual operation of ISDS, including an example of an instruction set designed by ISDS.
4. A summary of the results of experimentation with ISDS.

Formalizing the design process

Before programs that simulate design process can be considered, the complex nature of this process must be understood. Many models of the design process have been proposed, but for the most part they are the same in content if not in detail. However, two men have adequately expressed the complexity of design process—Asimow and Alexander.

Asimow considers design as a process of specification during which the solution to a design problem is gradually transformed from an abstraction into a physical reality. At each step the solution is analyzed. If any part of the solution fails to meet some requirement of the design problem, or if other decisions lead to better solutions, some parts of the solution may have to be re-specified. Dealing primarily with engineering
design, Asimow identifies over 25 different steps in the design process, each dealing with a different level of detail.

Alexander's view of design is consistent with Asimow's, although Alexander places greater emphasis on the relationships between design variables which must be considered at each step. The value of any part of a solution depends on, and may help to determine, the value of other parts of the solution. Since design is generally a serial process, the designer must be aware of these interactions and be careful about the sequence in which he makes design decisions. A particular method for treating the relationships between design variables is called "design strategy."

For many design problems, the truly creative part of the process seems to take place in the very early stages when the "design concept" is formed. This is the most abstract form of the solution, except, of course, for the more abstract functional descriptions.

Some design problems consist of complex sub-problems that require creative design, but for many, once the design concept is formed the solution is a relatively simple process of specifying details in such a way that the resulting solution meets the requirements of the problem. The fact that many computer instruction sets are so similar, suggests that the instruction set design problem is one for which most solutions can be generated by a single design concept.

This observation is the basis of an approach to writing a program that designs instruction sets. The trick lies in providing the program with an appropriate design concept that is general enough to include a broad range of instruction sets, but it must contain enough information to guarantee that the program can transform the concept into a solution in a reasonable amount of time.

GIS: A design concept for instruction sets

Existing instructions for computers have many common features. A typical instruction occupies one word of the computer memory and consists of several fields of information, each encoded in a particular set of bits. Most computers have a field containing a code for an operation the computer is to perform when it executes the instruction. An instruction may be comprised of one or more fields containing addresses of locations in memory which embody information to be used during execution of the instruction. Some computers allow special methods, such as indexing and indirect addressing, for specifying data in the main memory of the computer. The purpose of GIS is to organize as many of these features as possible into a single, general model of a computer instruction set.

Since GIS is a model for a type of language, it can be described in the notation of Backus Normal Form. The complete description of GIS is rather detailed since it includes almost all of the features that have been used in instruction sets and a detailed description of the meaning of each syntactic feature of GIS.

For illustration, part of a GIS representation of an instruction is:

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

The <operation> part of an instruction in GIS may be one of a list of 36 operations including:

add, subtract, multiply, divide, compare, branch shift, move logical operations, and others.

The <left operand part>, <right operand part>, <result part>, <if part>, and <else part> are <addresses>.

An <address>, in turn, consists of many parts including displacement information, indexing, indirect addressing, bits to distinguish between references to various types of memory such as main memory or register memory, and other special techniques for specifying memory locations.

Each part of an instruction has an interpretation. The right and left operand parts specify operands which are to participate in the operation. The <result part> specifies an address where the result of an operation is to be stored. The <condition part> specifies some internal condition which may be set as the result of the operation. The <if part> specifies the address of the next instruction provided that the internal condition is satisfied and the <else part> specifies the address of the next instruction if the internal condition is not satisfied.

In most instruction sets, some of the GIS parts have implicit values. For example, in a single-address instruction format one of the operand addresses is always assumed to refer to the accumulator. The same is true of the result address. The if and else instruction addresses are assumed to refer to the next instruction in memory. To completely specify an instruction set by means of GIS, it is necessary to indicate whether each instruction part is implicit or explicit. The assumed value must also be specified for implicit parts.
while, for explicit parts, the parts of the instruction format used to encode the value of the part must be precisely specified.

GIS can be used to represent almost any instruction format in use in existing computers. From a syntactic point of view its primary limitation is its list of operations, which is necessarily restrictive since some operations in actual computers deal with special features and cannot be generalized. From a semantic point of view, GIS is not capable of all the subtle nuances assigned to certain instructions in some computers. For example, GIS makes no distinction between post-indexing and pre-indexing. In most cases, however, these subtleties have little effect on the design of the syntax of the instruction language which is of primary concern.

The most important attribute of GIS so far as the design program is concerned is that it is a design concept for instruction sets which it appears to represent at an appropriate level.

GIS meets the requirement of generality because it contains all the important addressing methods as special cases. It can be used to represent single address instructions, double- or triple-address instructions, memory-to-register instructions, and register-register operations, as well as others.

Another requirement is that a program using GIS as its model of an instruction set should be able, without a great deal of effort, to generate instruction sets that are plausible solutions to a design problem. GIS possesses this feature in the sense that any instance of the GIS model is indeed a valid instruction set.

**ISDS: The design program**

The first step in the construction of ISDS was the selection of a method for storing GIS representations of instruction sets in the memory of a computer. The Backus Normal Form representation of GIS suggests a tree-like data structure. The structure actually used, called a "form-variable", is an IPL-V (Information Processing Language-V) list structure containing each instruction part identified by name and an attribute-value description list for each part to store important information about the part (whether it is implicit, whether the specification is a list of possible values or the number of bits needed to encode the time, and other descriptive information.)

All of the programs of ISDS are written in IPL-V, the primary reason being that IPL-V contains instructions for manipulating the tree-like data structure that is most appropriate for representing GIS instruction sets in the memory of a computer. However, the form-variable is a slightly more specialized data structure than the IPL-V list structure. Hence it was necessary to write a set of programs for manipulating form-variables.

These form-variable routines add items to form-variables, delete items, search for items, find attribute values on item description lists, and insert and delete attribute values on item description lists. The form-variable is a recursive data structure since an item may be a single value, a list of values or another form-variable.

The form variables of ISDS are at the lowest level of a hierarchy of routines (see Figure 1*) and are the building blocks of other routines in the sense that the higher-level routines make use of them to store new items in an instruction set, search for an item, and so on.

The form-variable routines are general in that they contain no information about instruction sets, GIS, or any aspects of the design process but are merely bookkeeping programs. ISDS contains another set of programs that are general in the sense that they perform the numerous computational tasks that must be undertaken during the design of an instruction set. These tasks include counting the number of items on a list and determining the number of bits required to encode a list of items.

At the level above the form-variable and computational routines, ISDS contains routines that add single parts to an instruction set. One such routine, for

---

example, adds a specified number of bits for designating index registers in a memory address. The number of bits is an input to this routine.

This routine performs no analysis, but merely the bookkeeping required to add a new part to an instruction set. The analysis required to determine the number of bits to be added for indexing is performed at the next level of the hierarchy. One routine, for example, adds indexing to the address references of an instruction set. For this routine the number of bits is not specified. The routine performs the analysis to determine the number of bits to be specified and then calls its counterpart which adds the specified number of bits. The routines which add specified parts to an instruction set are called “strategy-level utility routines”. The routines which perform analysis and call for specific parts to be added are called “operators”.

The routines in the higher level of ISDS are much more specialized than the low-level form-variable routines that can be used to represent many different kinds of objects. At the next higher level, the strategy-level utility routines are intended specifically for constructing instruction sets although they could be used in any design strategy since they have no decision power. Some decision power begins at the level of the operators which are based on a particular view of the relationships between the different parts of an instruction set. Each operator uses the values of certain parts of the instruction set to determine the value of some new part. The types of possible relationships are illustrated in Figure 2.

In many cases, the relationship between parts of the instruction set are relatively obvious, but different results could be obtained with a different set of operators.

So far, nothing has been said about how the operators of ISDS are applied. One way is to write a program consisting of a sequence of calls on the operators. Operators that might be called, for example, are the address operator (which selects the number of addresses per instruction and the size of each address), the indexing operator, the indirect addressing operator, the arithmetic operator, and the logical instruction operator. (This program would be a specific design strategy for the instruction set design problem.) It must be recalled that a design strategy is a particular method for selecting the parts of a solution to a design problem. In particular, a design strategy is a specific choice of the independent variables that determine each part of the solution, together with a particular sequence in which the design decisions were made. As was pointed out, the operators represent a particular view of the independent variables and their influence on each part of the instruction set. The operators could have been used to write a set of different design strategies. Instead, however, a heuristic program that would determine its own strategy according to the demands of the design problem was written:

The statement of the design problem to this program consists of the following information:

1. An optional GIS representation of a particular instruction set containing features which must be included in the final product.
2. A cost-value matrix which assigns a relative cost and value to each instruction feature of GIS. The cost-value matrix also specifies a maximum cost for the instruction set.
3. Optional constraints on instruction features.
4. Memory size, word size, and byte size of the computer.

The heuristic design program consists of two routines; a basic strategy and a search routine. The basic strategy uses the memory size and word size to determine the number of addresses in each instruction and the general format of each address (whether it is a memory reference or an address augmented by a base register, page bit, etc.).

After this basic strategy has provided a starting point, the search routine adds one instruction part at
The following inputs were presented to the heuristic program:

1. A cost value matrix as follows:

<table>
<thead>
<tr>
<th></th>
<th>Cost</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indexing</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Indirect Addressing</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>General Registers</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Partial Word Address</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Extra Operations</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Permanent Adjustment To Index Registers</td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>


3. Required operations of add, subtract, multiply, divide, compare, and absolute value for fixed point and floating point arithmetic.

4. Required operations of "negate", "and", "or", and "no operation for logical data."

5. A move operation.

6. Memory size and word size of 65536 words and 36 bits respectively.

The basic strategy determines that 16 bits are required for each main memory address. Since five bits are needed to encode the required operations, there is only room in an instruction word for one address without some augmented addressing scheme. The basic strategy can specify augmented addressing, but for this case it specifies a single, main memory address specification of 16 bits. The search strategy specifies additional instruction features in the following sequence: general registers, indirect addressing, additional operations, additional operations, indexing, a permanent adjustment to an index register after indexing, operations, operations, partial word addressing. The resulting instruction set has the following format:

```
0  5  6  9 10 13 14 15  18  19 20 35
```

This format is almost identical to the format of the Univac 1108 computer, however, the instruction set designed by ISDS is not. The primary difference is in the number of operations in the two instruction sets. The 1108 permits over 150 operations, whereas the ISDS instruction set contains only 52 operations.
The instruction sets also differ in their interpretation of some of the instruction features. However, this example shows that ISDS is capable of designing an instruction language that in its essential features resembles the instruction language of the Univac 1108.

It is interesting to note in the above example that if only 16 or fewer operations are required in the statement of the problem, then the basic strategy assigns four bits for the operation code and the remaining 32 bits permit two 16-bit memory references. In this case the search routine would not be able to apply any of the operators since every bit of the instruction word is used by the basic strategy. This illustrates a practical value of the present heuristic program; i.e., it permits a designer to learn by experimentation how the different design variables interact and how minor changes in one part affect the final product.

SUMMARY

Working with ISDS indicates that for some design problems it is plausible to write programs that solve the design problem without human intervention. In general, the approach consists of the following steps:

1. Select a design concept—a model of solutions to the design problem.
2. Select a data structure for instances of the design concept.
3. Create operators that perform analysis and specify single parts of the model.
4. Create programs that use cost, value and constraint information from the statement of the problem to apply the operators in some sequence that results in a solution to the problem.

This process, as it is applied in ISDS, is illustrated in Figure 4.

To be of practical use, a design program based on the ISDS approach would require a more sophisticated search strategy than the one used in the present version of ISDS. In general, it is probably possible to find clever ways of selecting the operators to be applied without actually trying every one. Any such scheme would give the search much more direction and enable the program to evaluate strategies of depth greater than one.

The approach to automated design described is of limited use in many practical design problems. However, as designers experiment with interactive design systems they are likely to discover problems for which the so-called creative effort is relatively routine. For such problems, the approach of ISDS offers the prospect of more efficient automation than can be achieved in an interactive system.

REFERENCES

1. M. ASIMOW  
*Introduction to design*  
Prentice Hall, 1962 Englewood Cliffs N J

2. C. ALEXANDER  
*Notes on the synthesis of form*  
Harvard University Press 1964 Cambridge Mass