1978

The value trace : a data base for automated digital design

S. J. McFarland
Carnegie Mellon University

THE VALUE TRACE:
A DATA BASE FOR AUTOMATED DIGITAL DESIGN

by

Michael C. McFarland, S.J.

DRC-01-04-80

December 1978

THE VALUE TRACE:
A Data Base
for
Automated Digital Design

by
Michael C. McFarland, S.J.

Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

December 9, 1978

This project was funded in part by the National Science Foundation, grant »GJ-32758X

Abstract

This report describes the design and implementation of an abstract, graph-like representation of digital hardware behavior, called the Value-Trace. This representation is especially useful for automaled synthesis of digital hardware because it shows explicitly **the** data-flow and control-flow relationships implied in the, functional specification. These relationships are shown to be of great imporlwce in recognizing feasible optimizations at several levels of a hierarchical automated synthr$j$ system.

We discuss a number of issues regarding th[A] representation itself, including what kinds of information should be included and how it is best represented. Then we describe the design and performance of a program which translate from the hardware description language **ISPS to** the Value Trace. Finally we show by meam of an example how global transformations on the Value Trace can be used to improve the design which would be produced by **the** automated system.

Table of Contents

Acknowledgements

I would like to thank my advisor, Prof. Alice Parker, for introducing me to the RT-CAD project, and for her many helpful suggestions in developing this work. I would also like to thank Prof. Daniel Siewiorek and Dr. Edward Snow, who originally developed the idea of the VT, for explaining the concept to me and for making many very helpful suggestions about what an implementation might look like. Many thanks too to Dr. Mario Barbacci, who helped me to understand the requirements a design representation must meet if it is to be useful in a comprehensive design automation system and for his many useful programming tips.

I am very much indebted to all of the members of the RT-CAD research group, especially Prof. Don Thomas, Andy Nagle, Lou Hafer, Gary Leive, Vittal Kini, Greg Lawson, and Richard Cloutier, whose many comments and sharp questions helped me to understand much better the issues involved in this work.

Finally, I would like to acknowledge the cheerful help and support of my community, the Christian Brothers of Central Catholic High School. Their kindness and understanding has certainly made this work much easier.

1. Background

1.1. Top-Down Design

The process of designing a digital system generally begins with the specification of the function which the system is to perform, together with certain constraints which it must satisfy. For example, a design group might be given the task of designing a sixteen-bit signed/unsigned integer multiplier which operates in under 100ns and has a failure rate less than 50ppm per thousand hours when operated in the temperature range of -55 to +125 deg. C. Along with these "hard" constraints, which the design must satisfy, there are usually other factors, such as cost and testability in this example, which should be optimized within the limits imposed by the initial specifications and constraints.

An intelligent approach to such a design problem does not begin by immediately laying out components in some haphazard fashion. First an overall design philosophy must be developed. This involves choosing the general structure of the system, e.g. whether the multiplication is to be done combinatorial^ with an array of processors, or partial products are to be generated and accumulated over successive time steps, or some combination of the two. At this level the basic style of implementation should also be chosen if it is not part of the specification, e.g. whether off-the-shelf TTL chips will be used or a custom LSI chip should be designed.

There are several reasons for making these decisions first. For one thing, they are the most abstract, being based more on the nature of the specified function and constraints and can be made at least preliminarily before anything is known about the detailed implementation, whereas none of the more concrete implementation issues can even be approached until the basic structure of the system is known. Furthermore these high level choices affect the entire system and therefore have the largest impact on system cost and performance. No amount of juggling data paths and control steps in a basically serial multiplier, for example, will make it perform as fast as a fully parallel array multiplier. Once the overall shape of the design has been set, the design process can proceed, laying out the data paths, processing elements, storage registers and controller in more and more detail until the components in the design correspond to the concrete, hardware-level elements in the chosen implementation, whether they be MSI registers and multiplexers or gate-level LSI standard cells. Finally the details of the physical layout must be worked out, including the placement of components, the routing of interconnections, and the translation of all the necessary information into a format appropriate for the actual construction of the device, e.g. mask layouts for ICs or PC boards.

Of course, the above description is an oversimplification. Even after the basic design has been chosen, it may be found as more and more detail is added that some constraint is not being met, or that some other important parameter is clearly not optimal. Then it is necessary to back up, change some of the higher-level decisions and try again. Any practical design methodology must make provision for iterations through various levels of the process. To keep this to a minimum, it should also be possible to incorporate some lookahead into fhe system. Even though they are basically abstract, the high level decisions can be made more intelligently if certain details about the lower level implementation can be anticipated.

Until recently, design-automation techniques have only been applied at the lower end of the design process. There are a number of well-developed techniques for chip layout and wire routing at the hardware level, and for gate- and state-minimization at the logic-design level. See, for example [Lewi77], [Breu72], and the Proceedings of any of the recent Design Automation Conferences. There are in addition some computer-aided design (CAD) programs which deal with systems at a somewhat higher level, where the primitive elements correspond roughly to MSI-type components such as registers and ALUs. This is normally referred to as the Register-Transfer (RT) level. Two CAD systems designed to work at the RT-level are the LOGOS system [Heat72], and the IBM ALERT system [Frie69} However, even these systems require that the basic data path and control structures be given as part of the input, so that they do little in the way of high-level exploration of alternative structures.

Yet there is a great need for design aids at an even higher level of abstraction. With recent developments in semiconductor fabrication techniques, digital systems can be fabricated cheaply in a single integrated-circuit package which are far more complex than anything a human designer can comprehend and control even at the RT-level ([Casw78]). As a result, design costs are now the dominant factor in the development of digital hardware, making general-purpose high-volume applications far more attractive, and discouraging the development of complex special-purpose low-volume LSI except where cost is not considered a major factor. Reliability, testability, and verification of designs are also becoming critical problems because of the lack of structured, high-level design aids which could provide systematic checking, verification and documentation at each level.

1.2. The RT-CAD System

It is the ultimate goal of the Register-Transfer Level Computer-Aided Design Project (RT-CAD) under development at Carnegie-Mellon University [Siew76] to automate the entire design process described above. The basic structure of the system is shown in Fig. 1, due to Leive ([Leiv77]). The basic inputs to the system at the top level are the specification of

RT-CAD OPERATIONAL OVERVIEW

A. DESCRIPTION PROCESSING

B. DESIGN PROCESSING

C. MODULE DATA BASE SYSTEM

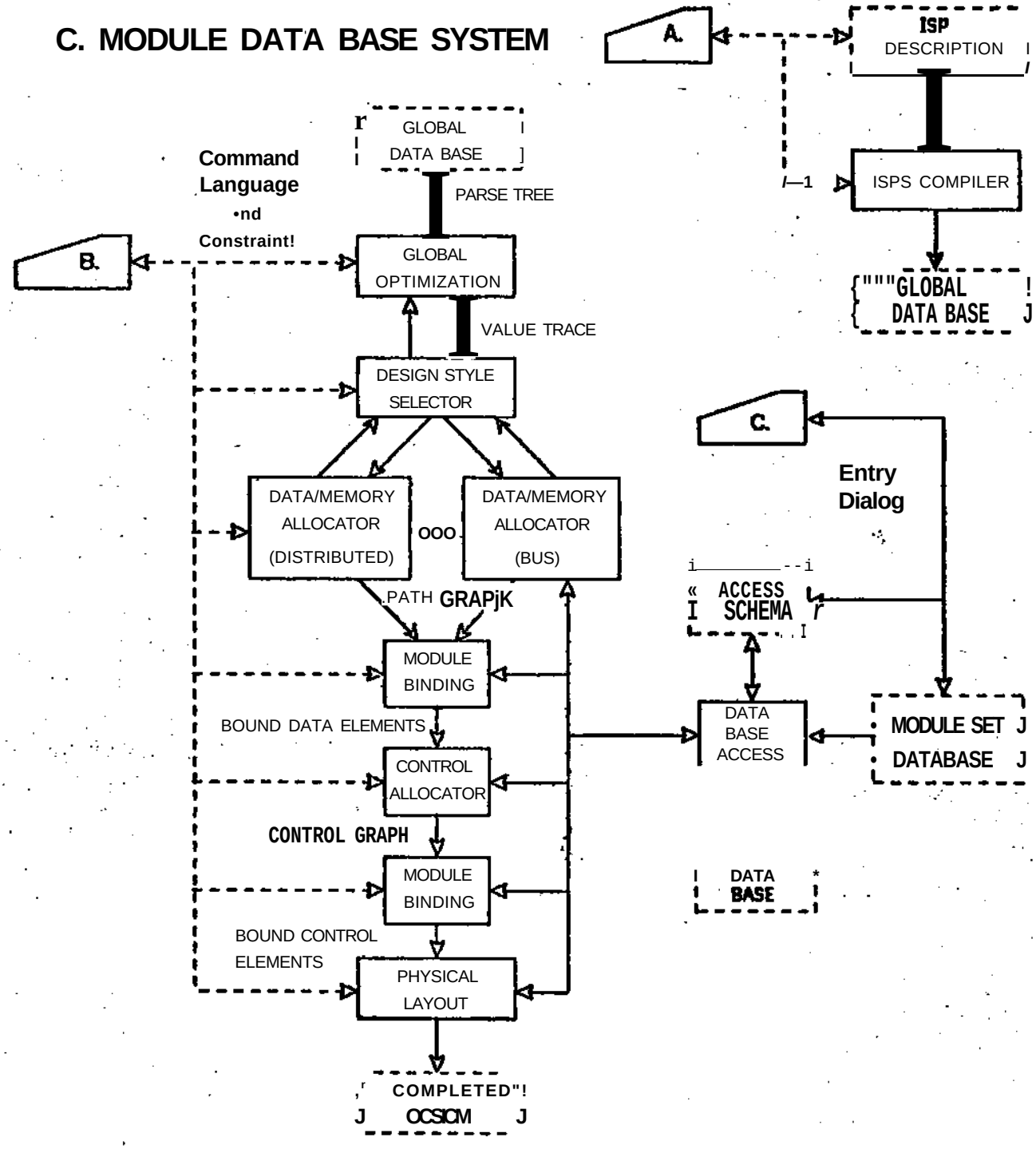


Figure h The RT-CAD System

the algorithm which the system is to implement plus certain constraints which the system must satisfy and other factors which should be optimized in the design. The behavioral specification is written in the ISPS computer description language described in [Barb77a], which is a refinement of the ISP notation originally presented in [Bell71]. ISPS bears a close resemblence to such structured programming languages as ALGOL and PASCAL, which makes the development of correct and comprehensible hardware algorithms relatively easy. At the same time ISPS allows the specification of certain factors, such as word length and methods of sequencing and control transfer, which are ignored in programming but which are very important in hardware design. The existence of of an ISPS simulator, [Barb77b], is also an important factor in the evaluation and verification of ISPS descriptions, as described, for example in [Barb77c].

The other input required by the system is a description of the different types of hardware modules in which the design could be implemented. These might include microprocessor chip sets, TTL, CMOS, standard cells to be fabricated in a custom LSI design, or specialized module sets. These descriptions comprise the Module Data Base and are described in [Leiv77]. Although the Module Data Base is used mainly at the lower end of the system, where the actual mapping into hardware takes place, a certain amount of information about the characteristics of the available implementations must be known at each level so that realistic evaluations of design alternatives can be made.

The top level of the system, the global optimizer, is intended to take the functional description and design specifications provided by the designer and perform certain optimizations on the algorithm itself which can be counted on to help bring the eventual implementation closer to the given specifications. This assumes that certain transformations can be recognized at this level of abstraction which can be determined to improve any available implementation according to the designer-specified criteria. Snow in [Snow78] has identified such transformations and has shown how they correspond to the kind of tradeoffs and optimizations which are used in real designs.

At the next level, that of the Design Style Selector, the overall structure of the system is chosen, whether it be bus-oriented, central accumulator, pipelined, or some other style, along with the basic type of implementation, e.g. microprocessor or distributed logic. Thomas in [Thom77] has discussed the feasibility of automating the design process in this top-down manner, has presented experimental results showing the importance of the design-style selection to the eventual result of the design process, and has suggested a method for automating the design-style selection based on measurements made on the algorithm supplied by *he* designer. Lawson in [La*s78} has implemented a measurement and decision package similar to that suggested by Thomas.

From the Design Style Selector down, the system proceeds by step-wise refinement similar to the methods used in structured programming as described in [DijK68] *inter alia,* adding more and more detail at each level until the elements of the design correspond to the basic hardware primitives of the chosen module set. The data path allocator lays out the structure of the data part of the design in terms of storage elements (registers, memorites and flags), interconnections (links, busses and multiplexers), and processing elements (adders, ALUs etc.). An allocator for the distributed design style has been built by Hafer as described in [Hafe78].

The path graph produced by the data path allocator is presented to a module-binding program which assigns physical modules from the chosen module set to each data path element, as described in [Leiv77]. The control allocator must then lay out a system which generates signals to control the data path elements so that they perform the given algorithm. Such an allocator is under development along lines described in [Nagl78a] and [Nagl78b} Once the controller has been laid out, it also must have modules bound to it. Finally, the description of the entire system in terms of modules and interconnections must go through a translator which generates the necessary input for a low-level design program, such as a PC board layout or IC mask generation system.

1.3. Tho Global Optimizer

The work presented here relates primarily to the top level of the system, known as the Global Optimizer. The objective has been to implement a data base which represents the abstract features of the system and which supports the type of analysis and transformation to be done at that level. As such it is basically an extension and realization of the work described in [Snow78]. Therefore, in order to see what requirements the database must satisfy it is necessary to look more closely at the task the Global Optimizer must perform.

At the same time, it has been determined that it is highly desirable to unify the entire system as much as possible by having it operate off of a common representation of the system being designed. There are a number of reasons for this. First of all, the system must allow communication between the various levels in both directions, i.e. not only from the top down, but also frow the bottom up in order to support both the feedback and feedforward aspects of the design process decribed above. Secondly, verification done at a higher level should not have to be repeated later on when the design becomes more detailed. This would be easier to arrange if later stages simply add detail to the representation generated earlier. Then only the details added at each level would have to be checked. Thirdly, there are a number of activities which are applicable at several different levels, including certain optimizations and measurements and certain types of human inputs to the system. It is

certainly desirable if possible to avoid having to reimplement these each time for a different data structure. Finally, translating from one data structure to another at each level adds significantly to the overhead of the system, both in terms of its initial development and in its actual operation.

tt is hoped, therefore, that the database described here can support the activities of the lop half of the system, at least down through the data path and control allocators. With that in mind, it will be necessary to consider the problems encountered at those levels of the system also.

First we will consider the Global Optimizer. The pioneering work in searching out the alternative implementations available for a given ISP description was done by Barbacci as described in [Barb73]. He was able to identify certain transformations, such as series-to-parallel and parallel-to-series rearrangements of the control, and the elimination of unnecessary activities, which had a large impact on the speed and cost of the eventual design. He also found that transformations applied to the innermost loops were the ones which had the highest payoffs, since they affected activities performed many times. The present RT-CAD system builds on this original work, but seeks to extend it in various ways, which gives rise to new problems. First, the EXPL system built by Barbacci was designing with a specific implementation in mind. Therefore there was no design style selection. Also, since the module-set used, RTM modules, [Bell72], was made up of fairly large-scale functional and control units, the ISP-level primitives often mapped directly into RTM units, although clever algorithms were developed to optimize bus usage. Finally, in EXPL the hardware-descriptive aspects of ISP were taken as at least an initial specification of the hardware structure of the system. That is, variables in ISP were mapped directly into hardware storage elements, the sequencing and branching was done as specified in the ISP and so on. Therefore, there was no time when the system was dealing with a purely abstract algorithm, and ail the transformations could be evaluated in terms of some specific implementation.

The system currently under development, as described by Snow, regards the ISP description simply as the specification of an algorithm for which an optimal hardware structure must be found. Evaluating the transformations at such a level of abstraction becomes more of a problem. Some of the transformations proposed can be counted on to improve the system, under all circumstances. One example would be constant-folding. Usually it would be much simpler to provide a constant 8 than to provide a constant 3 and a constant 5 and add them whenever an 8 is needed. Likewise it is almost always easier to take the rightmost bit of a word than to compute its remainder modulo 2. Other transformations can also be seen to have this global nature. Pulling a constant operation out of a loop will

generally increase speed without adding to cost. Eliminating redundant activities, i.e. operations which always produce the same result as another operation which has already been performed, and activities which do not affect the algorithm at all, are other examples. However, most constraints are conflicting and most transformations involve tradeoffs, using more processing elements and storage, for instance, in order to increase speed. To evaluate the desirability of the transformations, then, it is necessary to have some relative estimates of factors like cost and speed. This in turn requires that the global optimizer perform preliminary control-step partitioning and the allocation of storage and processing elements. Therefore it can be seen that the global transformations cannot be completed until at least a rough partitioning and allocation has been performed. Likewise the Design-Style Selector needs information about the allocation of storage elements and the sequencing and parallelism in order to perform its measurements. Lawson took these from the ISP description in his implementation, but in general what is shown in the ISP might not accurately reflect the ultimate implementation.

Seen in detail, the Global Optimizer will probably appear more as in Fig. 2. The high-level allocation and partitioning will certainly be improved by later stages of the system, but will at least provide a starting point for those stages, as the original ISP description does now, and will probably be at least roughly indicative of the overall shape of the final design.

The types of transformations which could be performed by the general Global Optimizer are detailed in Chapter 4 of [Snow78]. Since many of these involve rearranging the algorithm while preserving its results, the problem is in many ways analogous to the problem of recognizing and performing code optimization in an optimizing compiler, as described in [Aho77] and [Wulf75]. For example, common subexpression reduction, which changes the code so as to store intermediate results which are used several times rather than recomputing them each time; constant folding, which finds operations which have a constant result and simply substitutes the constant for their result; and code motion, which can be used to shorten loops by moving outside the loop operations which are constant for each iteration, can all be applied to a hardware algorithm in the same way that they are applied to code. Transformations such as these, which reduce the amount of computation required while using little or nothing in the way of additional resources, are representative of the kind of transformations which might be done at the top level before any partitioning.

It is also desirable at this level to identify and eliminate operations which have no effect at all on the algorithm. For example, if the result of an addition is stored in a certain variable and that result is overwritten by another operation before the variable is ever referenced, the first addition could be eliminated. This again would improve the design independent of any implementation details because it eliminates at no cost a possible source of delay and a

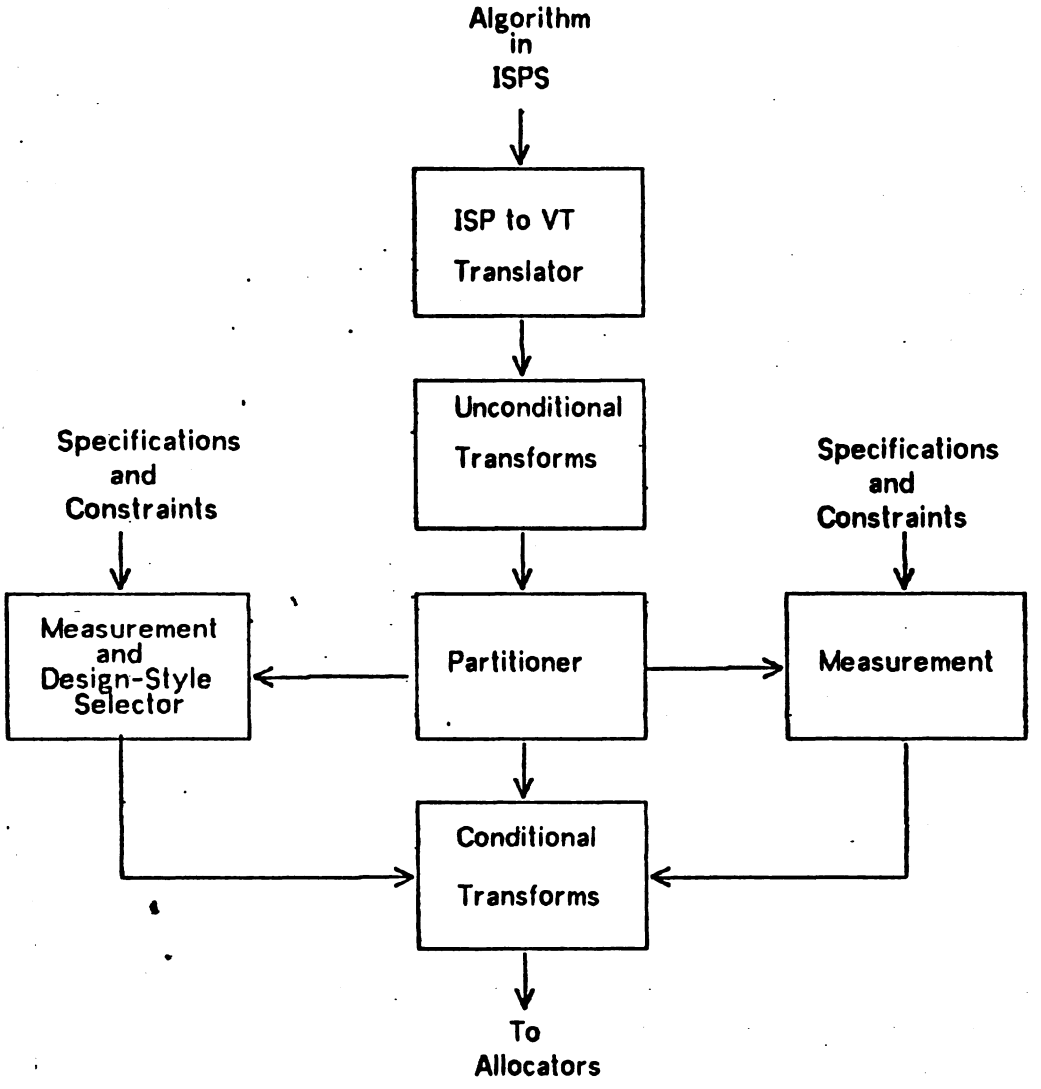


Figure 2: Detail of the Top Level of the RT-CAD System

use of resources.

The optimizer at the top level, therefore, should identify those operations which do not affect or are not affected by their local context and try to eliminate them, simplify them, or move them to a less critical context. This means that it must be able to determine for each operation where its inputs are produced and where its results are needed.

At the partitioner level, operations are divided into timesteps. It is at this point that the system must find potentially parallel operations, since these are the ones that could be

assigned to the same timestep. Operations can be put in parallel if the result of one is not needed in order to perform the other. Therefore it is again important to identify for each operation where its inputs are produced and where its outputs are used.

A simple approach to the partitioning problem would be to produce a maximally parallel design, since operations can always be serialized if desired later on. However, it seems that it would be worthwhile to try to balance the number and type of operations in each step, since this will result in the fewest processing elements for a given degree of parallelism and thus the lowest cost for a given speed. There should be enough information to do this at least approximately at the partitioner level if some simple assumptions can be made about how ISP primitives will be translated into hardware operations. Note that this is very close to the microcode packing problem discussed in [Toko77], In some ways the partitioner can be thought of as assigning non-conflicting micro-operations to microwords.

For the purposes of evaluating the conditional transformations discussed below and for performing the measurements needed by the Design-Style Selector and later stages of the program, some assumptions must be made about the assignment of storage elements to values which must be held between timesteps. To some extent the system may follow the ISP description in this regard, but it may wish to do some optimization here too. Also there may be values which are buried in ISP expressions, and which are therefore never assigned to a variable, which nevertheless will have to be stored somewhere in the actual computation. The value used as an index into an array is a good example. It would have to be held in an address register while the array access is being done.

The conditional transformations are those which rearrange the sequencing and control flow so as to improve certain parameters at the expense of others. The simplest example is the serial-parallel tradeoff. A more parallel implementation will generally be faster but more costly, since more processing elements and data paths will be needed to do more operations at once. There are other transformations which are more difficult to recognize and perform because they involve more than one control context, so that most optimization programs do not attempt them. However, they could have a large impact on the quality of a design and how well it meets its objectives, especially in simplifying the control. For instance, if a set of operations organized as a procedure is called from only one or two sites in the main algorithm, it would simplify the control if the procedure were simply expanded inline at each calling site, since no branching or subroutine linkage would be necessary. There are optimizations which could be applied to conditional branches also. A faster implementation of a branch can often be found by computing each branch in parallel and selecting among them at the end. Sometimes what appears to be a control branch in the original algorithm can actually be implemented with a data selector in hardware. The statement "IF I THEN c«-a+b

ELSE c«-a+c," for example, can be performed by multiplexing b or c into an adder, using I as the selector.

Recognizing and performing the above transformations requires a Knowledge of where data values are produced and used, as before, but also a detailed analysis and possibly a rearrangement of the control-flow. A good discussion of this as it applies to code optimization appears in [Hech77].

Other transformations could be applied at this point which enhance the modularity of the design. This might be desirable for greater testability among other things. One such transformation would be to find a group of operations which are strongly dependent among themselves but which communicate very little with other parts of the system and form them into an independent control context. Again a Knowledge of both the data and control dependencies would be needed for this.

Once the partitioning has been done and the transformations made, the output is to be passed on to the Data-Path Allocator. Hafer, from his earlier work in building an allocator, has pointed out a number of problems which are of interest at the allocator level. One is minimizing the number of storage elements by sharing registers among various variables and temporaries, similar to the register-allocation problem in [Wulf75]. This requires Knowing the "lifetime" of variables, that is how long they must be stored once they are computed. Once the registers have been assigned, the data paths must be laid out. Again it is desirable to share them as much as possible. For this an analysis of variable and operator connectivity must be made. At this level also, then, easy access should be provided to the sources of each operation and the uses of each output.

Processing elements must also be laid out at this level. In order to reduce cost in terms of data paths and processors, the allocator should be able to change the partitioning to reduce parallelism or assign operations to different time steps.

The path-graph produced by the Data-Path Allocator, along with the partitioned set of operations and information on the modules used to implement the data part, are passed to the control allocator. Nagle in [Nagl78c] has identified the issues which are important at the control allocator level and the types of information which the control allocator needs. One problem is again in recognizing potential parallelism, not only in parallel data operations, but also where the destination of a branch can be computed in parallel with some other activities. Another difficulty for the control allocator is that control-flow information must be made more explicit than it is in the ISP itself. For example, the ends of loops must be clearly marked and sequences of procedure calls must be followed in order to determine how complex the return mechanism must be, and where inline expansion would be desireable.

2. The VT:A Data Base for Optimized Design

2.1. The Notion of a Value Trace

It is most important for the transformation and allocator programs to have easy access to information about where the values of variables in the algorithm are produced and where they are used, as can easily be seen from the discussion in the previous section. This is a common problem in code optimization also, and a formalism for displaying such relationships, called a *data flow graph* was first proposed by Allen and Cocke in 1970. (See [Alle76] for a recent discussion of their work.) The nodes of a data flow graph correspond to operations which take certain values as inputs and produce new values. The arcs connecting the nodes represent the generation and use of data, i.e. there is a directed arc from node A to node B if node A produces data which is required as an input to node B. Therefore the data flow arcs represent the necessary precedence relations among the operations, thus making explicit all constraints on the movement or reordering of operations.

Data flow analysis has proved to be very useful in the optimization of so-called "straight-line code," that is, sequences of operations for which control can only enter at the beginning and leave at the end. In the presence of more complicated control constructs such as branching, loops and procedure calls, simple data flow analysis is not sufficient. A number of algorithms have been developed which take into account the effects of control flow, as discussed in [Aho77] and [Hech77], but these tend to be cumbersome and expensive and are not frequently used in practice.

The transformations proposed in the RT-CAD system must be able to operate in the presence of and even manipulate control structures. Therefore pure data flow analysis is not sufficient, though it must be incorporated. To meet this need, Snow in [Snow78] has proposed a new representation called a *Value Trace* or VT. A VT in Snow's formalism is a pirected Acyclic Graph very much like a data flow graph except that control constructs have been retained and translated into their data flow equivalents.

A VT is actually a collection of such graphs, each of which represents a *VT-body.* A VT-body is a set of operations which can be evoked, entered of left as a unit. ISP constructs which translate into VT-bodies are procedures, labelled blocks and the bodies of loops.

The nodes of a VT correspond to operations or *activities.* These include the usual unary and binary operations, operations which change or access fields in words or words in arrays, control operations such as procedure or block invocations, called *VT instantiations,* and conditional branches. Each operation is seen as requiring a list or vector of *values* as inputs

and producing one or more values as outputs. The arcs of the graph, then, correspond to values in the sense that an arc runs from node A to node B whenever a value produced by A is part of the input vector to B.

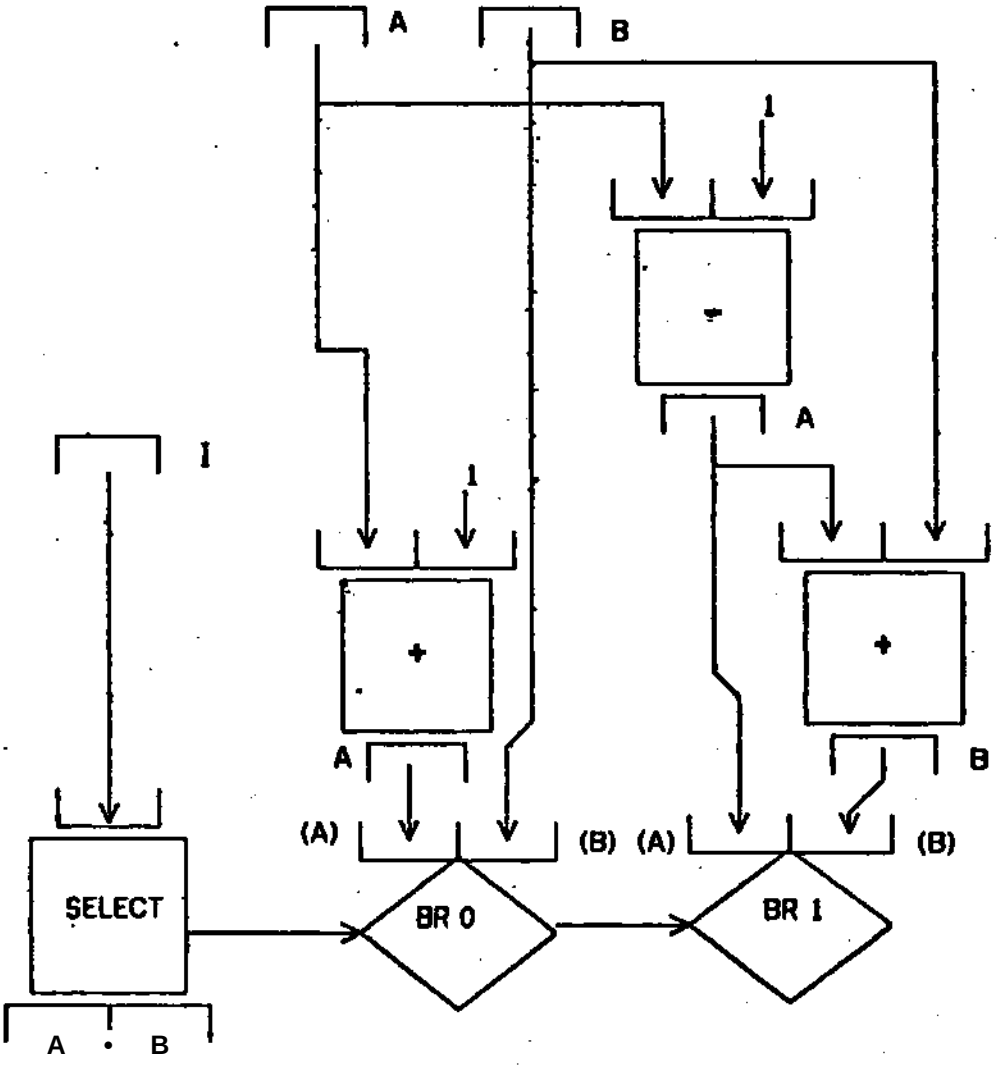
• A block invocation or procedure call simply looks like an activity to its local context. It has *an* input vector, which is made up of all the values which are required by the procedure or * block, not just any parameters explicitly passed, and an output vector which is made up of all the values which could be produced by the invocation. Thus data flow optimizations can be performed in the vicinity of a procedure call without fear of hidden side effects, since all the required variables and possible effects of the procedure are made explicit.

The most complicated structure in the VT is a *Select,* which is the VT equivalent of a conditional branch. From the data flow point of view, a branch selects among a number of different input vectors which are computed in parallel. The example in Fig. 3 should make) this clear. There an ISPS fragment and its translation into a VT-type graph are shown. The Select produces new values for A and B. What those new values are computed to be depends * on which branch is taken. In one branch, B is unchanged. However, since B could potentially be changed in the Select, the Select must be shown to produce a new value for B. This means * that any operation following the Select which references B cannot safely be moved before the Select.

2.2. The VT Augmented

When the integration of the VT into the RT-CAD system was considered, two major concerns surfaced. First there was the problem of identification and verification. After an algorithm has been transformed and a design produced, it should still be possible to relate elements in the design to their analogs in the original functional description. The VT as proposed had sought to abstract away much of the information that would have allowed such verification. Secondly, it became apparent that there was not enough explicit control information in the VT to make measurement, partitioning, and data and control allocation tractable. Three possible solutions were considered. One was to build a parallel control graph of some sort. However, there was a problem with maintaining parallel structures through the transformation of the VT. Either each transformation would have to have defined with it a parallel transformation of the control graph or the control graph would have to be rebuilt each time. Maintaining such parallel structures is cumbersome in general, as was noted in [Barb73]. Another possible alternative was to work with the "pure[1]* VT up to a certain point and then to reinsert all sequencing and control information from scratch. Unfortunately, searching through all possible arrangements of sequencing and control is simply far too complex a task to undertake. The Optimizer should not be bound by the way

```
DECODE I =>
   BEGIN
     0:-   <-A+1,
     1:=(A*-A-1 next B«-B+A)
    END
```



Figurs 3: The VT of a SELECT

the algorithm is originally expressed, but there is much useful information in the ISP which can be used as a starting point for an intelligent search of the alternatives. Also there are many cases where control constructs which have a simple representation in the ISP and a simple implementation in the final design can only be represented in a very tortured manner in a data flow type graph. A LEAVE, for instance, which jumps through several levels of

procedure calls, would have to be shown as conditionally altering the output vector of each level, whereas in reality it is a simple jump or return.

The third alternative and the one taken in this implementation of the VT is to add the control and sequencing information to the VT itself. This makes the structure somewhat more complicated and difficult to manipulate, but avoids the worse problems of keeping parallel structures or having to rebuild all the information later.

Sequencing information is preserved in the VT by always maintaing the "operator nodes," which correspond to activities in the original VT notation, in a linear list in executable order. This list is maintained and rearranged through all the transformations. There is also provision for stringing operators together in conceptual control steps. A whole new class of transformations can be defined which deal with the reordering of operators and their assignment to different control steps. Partitioning can then be defined in terms of these transformations rather than the optimal ordering of an unordered set with constraints. Such an incremental approach to the partitioning problem cannot be counted on to always give an optimal solution, but at least the problem is reduced to a manageable size. Also, since the VT always maintains some partitioning information, there is some flexibility in when various transformations and measurements can be done.

Branches are represented as both data selectors and alternative control-flows. Each branch, besides having an input vector, points to a list of operations which are invoked when that branch is selected. If the construct shown in Fig. 3 were translated into a VT in this representation it would have added to it an ordering of the operator nodes and pointers from each branch node to the operators associated with it. Obviously the designer or transformation program has the option of moving operations outside the select so that they are always executed. Thus the select can be implemented as either a control branch or a data selector, as will be shown in a later example.

Other control constructs are represented with both their data flow and control effects made explicit. Stop, Delay and Wait operators, for example, are included because they affect the sequencing, though they do not produce new values. A Leave, which causes a jump out of a named context, requires as inputs all the outputs of that named context, but of course returns no values to the context left. It is put in the VT so that it can later be translated by the control allocator into a jump or return, depending on the complexity of the calling sequence.

Identification information, qualifiers, and attributes are also kept in the VT representation. This is done in two ways. First, each VT has a list of descriptors with it, one for each "entity", i.e. process name, register, memory, flag, map, and VT-body declared in the original

ISP. It is intended that more information be added to this list as more detail is specified in the design. It may well be that enough information can be put here to fully specify the path graph. Discussions are under way regarding this point. Secondly, each value input to or produced within a VT-body has a descriptor, called an "outnode", associated with it which identifies the carrier and name associated with it in the original ISP. Again this information can be changed or added to as the design proceeds.

2.3. Information Contained in the VT

Conceptually the VT has two parts. The first contains the descriptors for all of the entities declared in the ISP. These are called "entity nodes". There is a basic structure common to all entity nodes; but in addition certain types of entities have fields added which are used only for that type. .

The basic entity node structure, and the one which is used for all carriers, has the following contents:

ENTITY NODE

1. Name (from ISP)
2. Number (for identification in the VT)
3. Flags
4. List of Attributes
5. List of Qualifiers
6. Entity in which it is Declared
7. Arithmetic Type
8. Range of Word Indices (for an array)
9. Range of bit indices (for any carrier)

The entity node for a Sectionlist Header, which is a name given to a collection of declarations, contains the first six fields shown above and also has a list of pointers to all the entities declared in it.

The entity node for a VT-body has all the fields in the basic entity structure plus the following:

10. Output Values (list of all carriers which may be changed in this VT-body)
11. Input Values (list of all carriers whose values are needed in this VT-body)
12. Formal Connections (List of all formal parameters of a. procedure-type VT-body)
13. Activation List (List of all VT-bodies called

from this one, either directly or indirectly)
14. Calling List (List of all operators which invoke this VT-body)

A mapping defined in the ISP is retained in the list of declarations for identification purposes although it does not correspond to any hardware register or array. For a mapping, the following fields are added to the basic entity node

10. Target (base element to which this map refers)
11. Bit Offset (from rightmost bit of target)
12. Word factor (amount by which the word size of an array is multiplied or divided in the mapping)
13. Word Offset (from the first word of the target array)

The second part of the VT holds information about the operations which perform the algorithm. They are organized by VT-bodies. Each entiiy node which represents a VT-body is also used as a header for the list of operator nodes in that VT-body. The operator nodes in a VT-body *are* stored in order in a doubly-linked list pointed to by this header.

Operator nodes hold the following information:

OPERATOR NODE

1. Operator Type (opcode)
2. Identification Number
3. VT-body header
4a. Arithmetic Type (for an arithmetic operator)
4b. VT-body invoked (for a call, restart, etc.)
4c. List of Branches (for a SELECT)
5. Input List (List of values used as inputs)
6. Output List (List of values produced as outputs in outnode format)
7. Previous Operator Node
8. Next Operator Node in List
9. Qualifiers
10. Branch Containing this Operator
11. Next operator in this timestep
12. Attribute List

For identification and efficient searching, each value which is produced by an operator, each input to a VT-body, each formal parameter to a procedure-type VT-body, each output from a VT-body and each constant has a value descriptor node associated with it called an outnode. The following information is stored for each outnode:

OUTNODE

1. Register (where the value is stored if this is determined)
2. Operator by which this value is produced or VT-body with which it is associated
3. Name Attached to this Value (if any)

3a. Value (for a constant)

4. Size (number of bits)
5. List of Operators Using this Value
6. Identification Number and type

As noted above, each select points to a list of branches. Each branch is represented by a node with the following format:

BRANCH NODE

1. Value(s) of Conditional for which this Branch is Selected
2. Input List (as represented in Fig. 3)
3. Select for which this is a Branch
4. List of Operators in this Branch
5. Qualifiers for this Branch

As an example of what these nodes would look like, consider the following ISP:

```
C:=Begin
   **Storage**
   A[0:15]<23:0>,
   B<23:0>,
   B1<11:0>:=B<23:12>,
   I<3:0>

   **Behavior**
   F:=  Begin !US!
        B←A[I] next
        B←B+F1()
        End,
   F1<4:0>:=Begin
            F1←2*I
            End
   End
```

If this were translated into a VT, entity nodes would be built for C, A, B, B1, I, F, and F1. Since C is a sectionlist head, its node would have a list of all entities declared in it. In this

case, the list would include all the other entities, while each of the other entities would point back to C as the entity in which it was declared.

A, B, and I are carriers. Since A is an array, it would have fields for both its word structure (0:15) and its bit structure (23:0). The others would show only a bit structure.

Bl is a mapped entity, so in addition.to the fields used to describe a carrier, its entity node would include the additional fields used to describe a mapping. In particular, it would show B. as the target of the mapping and a bit offset of 12.

F and Fl are VT-bodies. Consequently their entity nodes would include the extra fields used only for VT-bodies. F would be specified to have inputs of A and I, outputs of B and Fl, and an activation of FL The arithmetic type of F would be unsigned. Fl would have an input of I_9 an output of Fl, no activation, and a call from the CALL operator in F. Note that besides having a behavior connected with it, Fl can also hold a value. Therefore it has a bit structure like any carrier.

The entity nodes for the VT-bodies F and Fl would also act as headers for the behvioral descriptions associated with those VT-bodies. Thus the entity node for F would point to a list of three operators, an array-read, a CALL, and an addition. The array-read would take as inputs the outnodes A and I, which are inputs to F, and produce a 24 bit output which is assigned to B. The call would take I and constant 2 as inputs and produce a five bit output assigned to Fl. Finally, the last operator in the list, the addition, would take as inputs the outputs of the previous two operators and produce a 24 bit output assigned to B. The output B of the VT-body would refer to this last value as its source, while the output Fl would refer to the output of the CALL operator as its source.

More elaborate examples appear in the Appendix, along with instructions on how to read them.

Of course all references to other nodes are implemented as address pointers to those nodes, so that the information structure can be easily searched. Lists which are of variable size and which may be changed by transformations are implementaed as linked lists. This saves the moving and rebuilding of tables every time elements are added or deleted, a problem noted in [Barb73],

In summary, there are basically five types of information represented in the VT: data flow, control flow, resource usage, context, and identification and qualifiers. Therefore the same information can be accessed a number of different ways. Moreover, most accessing mechanisms are two-way in order to make searches as easy as possible. For example, for a given operator, there are pointers to all the values which it needs as inputs, and these all

point to the operators which produce them. Conversely, for any given value, there are also pointers back to all the places where it is used. Thus the effects of changing a value can be easily checked.

2.4. Tho ISP-to-VT Translator

A program to generate VTs from ISP descriptions has been implemented and is available on the PDP-10D at CMU. The current executable version is kept as dskc:vt[x375lh0kJ The program is written in BLISS-10 <[Wulf71]). This is an attractive language for building large production version systems because it provides high-level control constructs while allowing the programmer access to all the features of the particular target machine. It is thus possible to exercise considerable control over such details as the implementation of data structures, the way loops are translated and so on in order to produce more efficient code.

The VT translation is actually performed by doing a symbolic execution of the ISP description. First a descriptor (an entity node) is built for each named entity in the ISP and for each potential VT-body. Since an invocation of a VT-body is built as an operator, requiring certain inputs and producing certain outputs, the graph for one VT-body cannot be built until all the inputs and outputs of the VT-bodies called by it are known. But a certain VT-body, call it procedure A, may invoke another, say block B, while B may cause a Restart of A, in which case neither can be translated. To resolve this impass, during the building of each VT-body descriptor, an initial pass is made through the ISP description of its behavior to pick up all the variables referenced by it, all the variables which it alters, and all the other VT-bodies which it calls.

After all the descriptors are built, the transitive closure of all the VT-body input, output and activation lists are taken. That is, the inputs to a VT-body must include not only those carriers referenced by its own operators but also those referenced by any VT-body which it calls. Similarly, the output list of each VT-body must be augmented to include all the outputs of VT-bodies which it calls, and the activation list to include the activations of all VT-bodies which it activates. For example, if A calls B and B calls C, then C must be on A's activation list. And if C writes in register I, any VT-body which calls A must see that A could produce a new value of I. Therefore I must be on A's output list.

Actually, all of the carriers referenced in a VT-body may not be needed as inputs. A carrier may be overwritten before it is referenced, in which case its old value is never used. But this is not known until the data-flow analysis is completed. Therefore, when a VT-body has been translated, its input list must be pruned, and all calls to it which have already been translated must be updated to reflect the change in the input list. To minimize this, after all

the input and output lists have been built, the VT-bodies are sorted in ascending order of the number of calls to them. In this way VT-bodies which are called from other VT-bodies tend to get translated before the operators which call them.

At this point the translation can procede. The basic approach is straightforward. Each time a carrier is written into, a pointer to the descriptor for the new value is stored in the entity node for that carrier. Also the old value is pushed onto a stack. Therefore at any point in the translation the entity nodes hold the current "state" of the machine, namely the current value for each carrier, while the stack allows any previous state of the machine to be recovered. Whenever an operator takes a carrier as an input, the entity node for that carrier is looked up, and a pointer to its current value is put in the input list for that operator.

Translating branches requires translating several alternative series of operations, all beginning from the same state of the machine. This is done by marking the stack, translating the first branch, "rolling back" the state of the machine to its original state by popping the statck until the mark is encountered, translating the second branch and so on. All values changed anywhere in any branch must be remembered and added to the output of the ENDSELECT.

When the translation is completed, the VT is dumped out in symbolic form either on the teletype or in an ASCII file, as specified by the user. The output file, which has default extension .VTR, contains enough information to rebuild the VT data-structures in core, but does not contain all the cross references described in the previous section. It is primarily designed to be machine-readable; but with some coaching it can be made intelligible. A description of the format is included in Appendix A.

An attempt was made to make the program as general as possible, especially in its ability to handle the many modes of variable-access and mappings which are allowed in ISPS. This required considerable error-checking and special-case handling. To simplify later stages of the design system, all accesses to ISP-defined variables are translated into hardware-type accesses on the real data carriers. For example, if A is defined to be a certain field of register 8 (a mapping in ISPS), then accesses to A are translated to accesses to the appropriate fields of B. Similarly, if an array A is mapped onto an array B, all accesses to A are translated into accesses to the the appropriate word and field of B.

All accesses to a field of a word are translated into a standard form using an (offset,size) type of specification. That is, any field in a word is defined by its offset from the rightmost bit in the word and its size in bits. This provides a simple, consistent notation which transforms linearly under ISPS-type mappings.

These word and field accesses are shown in the VT using four new operators, array-read [r], array-write [w], bit-read <r> and bit-write <w>.

The array read inputs two values. The first is the old value of the array. The second is the index of the word selected, counting the first word in the array as zero. The output is then a word from the array. However, it is possible for the output size to be greater than the word size of the array. In this case the hardware will have to access more than one word.

The bit-read operator takes as its first input the value of the carrier (full word as defined) and as its second input an offset. The offset is the number of bits from the right of the carrier that the bit field is to begin. Thus a <r> operator with inputs of A and 4 and an output size of 8 produces as a value an 8-bit field starting with the fifth bit from the right of A. If A were defined as A<18:0>, it would produce the value A<11:4>

The word-write operator accesses a word in an array, modifies all or part of it and produces a new array with that one word modified. It takes four inputs. The first is the old value of the array. The second is the index of the word to be modified, from 0 as before. The third is the value to be inserted in the word. The size of this value determines how many bits are to be changed. The fourth input is the offset from the rightmost bit of the word to be modified where the field to be modified begins.

The bit-write operator has three inputs. The old carrier value, the new partial-word value to be inserted into the old word, and the offset from the rightmost bit. Thus a <w> operator with inputs (1) A<0:12>, (2) x with a size of 3, and (3) 2, will produce a new A as its output with A<8:10> equal to x and the rest equal to the old value of A.

Operators needed to convert indices and offsets from the ISP defined values to the standard values used by the bit- and word- read- and write- operators are generated automatically during the VT translation.

The program needed to implement all these features turned out to be somewhat lengthy. It comprises some 6000 lines of BLISS code, including documentation. The executable code occupies about 20K words on the PDP-10 when loaded with the debuggers. Much of the bulk, fortunately, is for special case handling so that it does not get executed often.

Programming the translator was made much simpler by the existence of the ISPS compiler and the format of its output. The compiler parses the ISP description and outputs a file with a .GOB extension, which is in the form of a tree. This Global Data Base (GOB) file is actually what is input to the VT program. A module has also been provided to rebuild the GOB in core

and this was used as the front end of the VT translator. The GDB structure is simple and clean. Its tree format makes searches simple to do using recursive subroutines.

3. Global Transforms on the VT

3.1. Typical Global Transforms

There are four basic types of transformations that are possible on the VT at the global level. The simplest are those which just eliminate operations which do not really affect the outcome of the algorithm. These are easy to find using the use-list for output values. If an operator does not affect synchronization by causing a WAIT, DELAY, or STOP and does not cause a jump in control, and its values are not referenced anywhere, i.e. its use-list is empty, then it can be eliminated. Performing the elimination only involves taking the operator out of the sequencing lists and eliminating its inputs from the use-lists of the values they reference.

Somewhat more elaborate are those transformations which involve replacing one value with an equivalent one. Constant folding is one example. The output of an operator with constant inputs can be replaced by a constant. Or an operator with one constant input could be replaced by a simpler but equivalent operator, as a division by constant two could be replaced by a shift right. Another example of transformations at this level is redundant operator elimination. That is a transformation which locates pairs of operators which always produce the same result and eliminates the later of the two. Before the second operator is deleted by the first type of transformation described above, all uses of the second operator have to be transferred to the first and the use-list of the first must be updated to reflect this.

At about the same level of difficulty but involving different considerations, are the third type of transformations, those which involve moving operators with respect to one another in the sequence list. This is the way operators may be put in parallel or put in a more efficient order. An operator may be moved backward as long as (1) it does not move into another control context (branch or VT-body), (2) it does not move past any of the operators which produce values which it requires as inputs, and (3) it does not move past any operators which enforce synchronization. Similarly, an operator may move forward provided that (1) it does not move into another control context, (2) it does not move up to or beyond any of the operators which use its outputs as inputs, and (3) it does not move past any operators which cause synchronization. These conditions are easy to read from the VT by scanning the operator list between the present and proposed position of the operator and using the data flow and context information (branch and VT-body pointers) provided for each operator. Moving the operator is simply a matter of rearranging the sequencing lists.

Transformations of the fourth type are those which operate across control contexts. Such transformations move operators into and out of VT-bodies and select bodies, combine or factor select operators, build VT-bodies inline, and create new VT-bodies. These are far more difficult to recognize than the other classes of transformations and are harder yet to actually perform; but they are by far the richest and most powerful set of transforms available. In fact, it would hardly be an exaggeration to say that these transformations are what change the Global Optimizations from a set of clean-up routines to an intelligent design-optimizer. Most such transformations can be recognized by checking the inputs and outputs of the VT-body, branch or select to be changed and the inputs, outputs and context of the operator to be moved, if there is one. Performing such transformations requires a fair amount of bookkeeping, since inputs, outputs, sequence lists, and context pointers for both the operator and the control context must in general be changed, but it is reasonably straightforward nonetheless.

In order to gain some experience with transformations operating on the VT, a set of routines was built which implements one of each of the classes of transformations described above.

The first routine deletes an operator which is no longer needed. It assumes that the routine which calls it has determined or arranged that the outputs of the operator are unused. This routine then takes the inputs of the operator off of the use-lists of the values which it has as inputs, removes it from the sequencing lists and releases the space it occupied.

The routine chosen to represent the second class, a redundant operator eliminator, was implemented so as to be able to search a VT-body and find all truly redundant activities and eliminate them in a fairly efficient manner. Thus it can operate with a minimum of direction. First it sorts all of the operators in the VT-body by type. This cuts down the size of the search considerably. Then for each pair of operators of the same type it checks their equivalence. If they are equivalent, it transfers uses of the outputs of the second to uses of the corresponding outputs of the first and eliminates the second. Two operators are determined to be equivalent if (1) they are of the same type, (2) they have the same values as inputs, (3) they are not in mutually exclusive branches, (4) they have the same arithmetic type or they represent calls to the same VT-bodies, and (5) their outputs are of the same size.

The equivalence of the operators was simple to check from information stored in each operator node. No lifetime analysis was needed to see if a variable referenced by both was changed in between the two references, and no thorough context search was needed either.

since both contained pointers to their enclosing branches.

The routine that implemented this transformation along with routines it required to do its checking and rearrangements are contained in less than 150 lines of BLISS. It has been found that it can check through a fairly large VT in minimal time.

An example of the third type of transformation is a routine which takes as inputs two operators and checks whether the first can be moved in parallel with the second. To do this it checks the context to make sure that they are in the same branch or in no branch, then scans the operator list between the two to make sure that no synchronizing operators are passed and no operators are in between which reference outputs of the operator to be moved. It leaves it to the calling routine to decide how to move the operator. Furthermore, if the movement is blocked somehow, it returns a number indicating the reason why, so that the calling routine may decide whether it should try to remove the blockage.

As an example of the cross-context type of transformations, a routine which identifies and performs select-motion, i.e.* finding an operation which is performed in each branch of a select and deferring it until after the branches have merged. It is believed that this is the most complicated single transformation defined on the VT because it involves rearranging the select, which is the most complicated structure in the VT. The routine is designed to take as inputs a select node and an operator in the select and determine whether there are corresponding operators in all the other branches that can be moved. If so, it performs the select motion, if not it returns the reason for failure. It is so structured that it can be used in a search for possible candidates for select motion. It operates as follows:

1. Determine if the operator (call it f) can be moved to the end of its branch.

2. Find an output of the select which comes from f. For each other branch find out if the same output is produced by the same type of operator. If so, determine if that operator can be moved to the end of its branch.

3. If all these operators have been found and can be moved, the select motion can take place. F must be moved outside the select and the select changed so that it selects among possible inputs to f. To do this, delete all of the operators except f. Add what had been the inputs of each of those operators to the inputs of its corresponding branch node and delete the outputs of those operators from the inputs of their respective branch nodes. Similarly, add fs inputs to the outputs of the select and delete f's outputs from the output list of the select. Fix up the use-lists to reflect these changes. Change all later operators using the outputs deleted from the select to use the outputs of f.

4. Move f in the sequence list until it follows the select. Fix up the context pointers.

5. Clean up the select. This involves a routine which checks whether a pair of

inputs to each branch node always come from the same value, in which case it collapses them into one, and which checks whether a certain input in each branch always comes from the same value outside the select, in which case it removes it from the select and fixes up all uses of that output of the select to use the original value.

It can be seen that the transformation involves a good deal of rearrangement. Conceptually, it would be easier to think of the transformation as operating on a table whose rows corresponded to the individual branches, with the final row containing the select outputs, and whose columns corresponded to different elements of the input and output vectors. Then eliminating an input from each branch would correspond to deleting a column, and so on. Unfortunately, it is not easy to delete a column from a table in practice. Rather, table access was simulate using routines which, given a list head and an index i, either find or delete the ith element from the list.

The select motion operation with all its attendant routines was implemented in about 250 lines of BLISS, which makes it of manageable size once organized properly. There was no great difficulty in making it work.

3.2. MIN: An Example

The following example illustrates how the transformations described above operate and how they might result in more efficient hardware.

Figure 4 shows an ISP for an algorithm to find the minimum of an array, taken from [Mann74]. It is a restricted use of the kind of sorting algorithm which converts an n^2 operation into *an* nlog(n) one. It is efficient when an array access costs no more than a comparison. At the beginning the list to be searched is stored in the top half of the array. Starting at the top, the algorithm takes pairs of elements, compares them, and stores the smaller below the current bottom of the list. When it reaches the first element the minimum is stored there.

As written, the algorithm is not an efficient implementation, since it recomputes the index and reaccesses the same element from the array for S[2*i] and S[2*i-1]. Figure 5 shows this.[1] There are four array-reads done where two would suffice and four multiplications by 2 where one only is needed. Figure 6 shows the VT structure after redundant operator elimination has been performed. Note that the number of array accesses and multiplications has been cut, leading to a faster operation.

[1] The actual VT outputs for this example are in Appendix A

```
! ISP of an algorithm to find the minimum of an array
! The array is stored in the upper half of S, beginning at S[n+1]

"min:»  BEGIN

        ••INPUTS**
        n<5:0>,              Inumber of words in the array
        S[0:127]<15:0>,      !The array and scratchpad

        **OUTPUT**
        z<15:0>,             !The minimum value

        **INDEX**
        i<5:0>,

        **THE.ALGORITHM**
start- BEGIN
        i«-n next
        loop:-  BEGIN
                IF i EQL «OO -> <z«-S[O] next stopO ) next
                DECODE (S[2*i-lJ LEQ S[2*i]) ->
                        BEGIN
                        O\false:-  S[i-1]←S[2*i],
                        I\true :=  S[i-i]←S[2*i-1]
                        END next
                i«-i-l next
                RESTART loop
                END
        END
END
```

Figure 4: ISP of MIN

It can be seen from Fig. 6 that the second select invokes one of two sets of operations which lead to an array-write. But really what is done is to store one of two values in a certain place in the array. Applying the select motion does this. Now the select simply gates

one of two values into the array. This would result in a much simpler and probably faster control scheme, since it could be implemented combinationally with a multiplexer. The result of the transformation is shown in Figure 7.

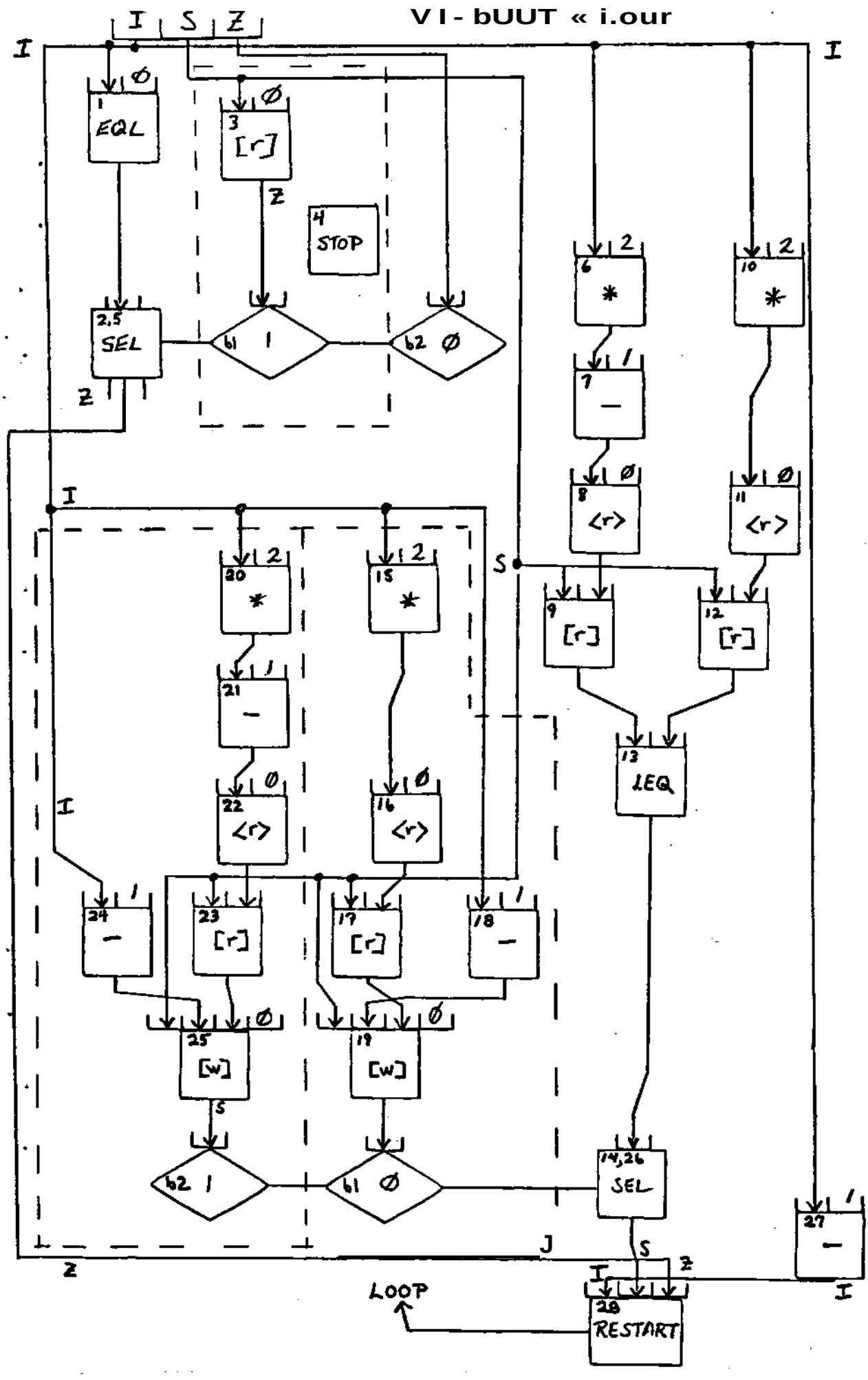
3.3. Verification

One of the features which make high-level design automation attractive is its potential for producing verifiably correct designs. Once the algorithm is itself verified, it should not be necessary to reverify the entire system at each level. Rather each transformation should be checked to ensure that it does not affect the behavior of the system and each new level of detail added should be shown to faithfully implement the specifications and to handle correctly the new issues that enter at that level, e.g. register sharing at the data path allocator level. It is of concern, then, to be able to show that the global transformations preserve the behavior of the system.

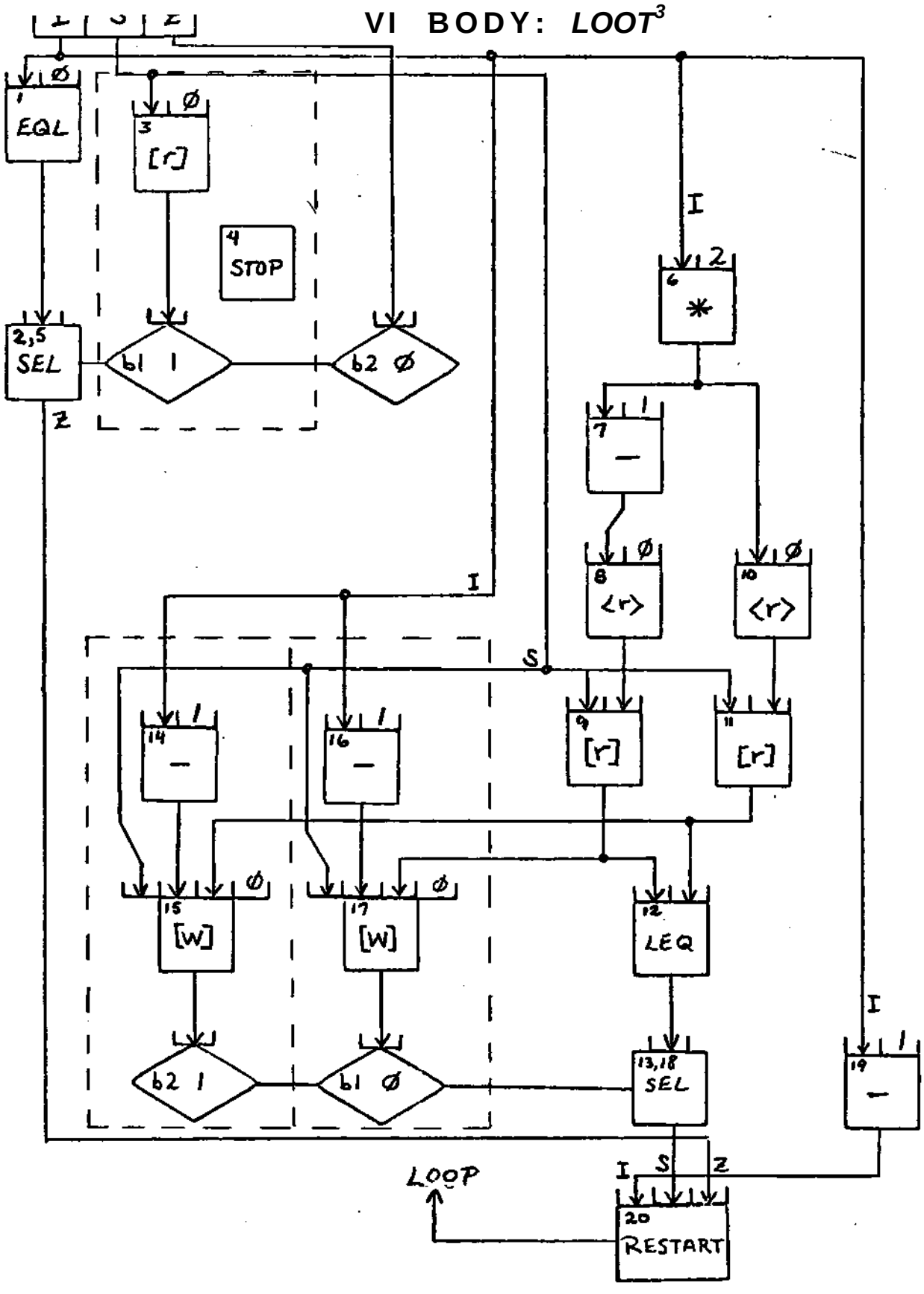
Proving the correctness of a program involving data structures involves operation at two different levels, as described, for example in [Hoar72]« First it must be shown that the abstract transformation preserves certain assertions about the abstract structures. Then it must be shown that the actual implementation preserves the identity between the actual structures and the abstract structures.

As an example of how a transformation might be verified, consider the redundant operator elimination. On the abstract level it must be shown that removing the second operator leaves the behavior intact. There are two ways in which an operator can affect behavior. One is by producing new values and the other is by causing changes in timing. This transformation is not applied to selects or synchronization operators, so we need only show that values are preserved. Now two operators of the same type with the same inputs produce the same outputs. If the outputs are of the same size, then they are identical. As long as we can show that whenever the second operator is invoked, the first must have been invoked already, then the outputs of the first can replace the outputs of the second. Since both operators are in the same VT-body, as long as they are in the same branch or the first is in a branch in which the second is nested, this is true. But the routine checks for this, so on the abstract level it does indeed preserve the behavior.

On the implementation level, it is necessary to identify each list or pointer which could include the operator to be deleted and show that it is adjusted correctly. When other parts of the structure are changed, a similar check must be made for them. First, all inputs using the output of the second operator are changed to point to the first. But the use-list points back to these inputs, so it must be shown that the routine transfers every input from the



Figur* 5: The Original VT for MIN



Figurs 6: The VT for MIN after Redundant Operator Elimination

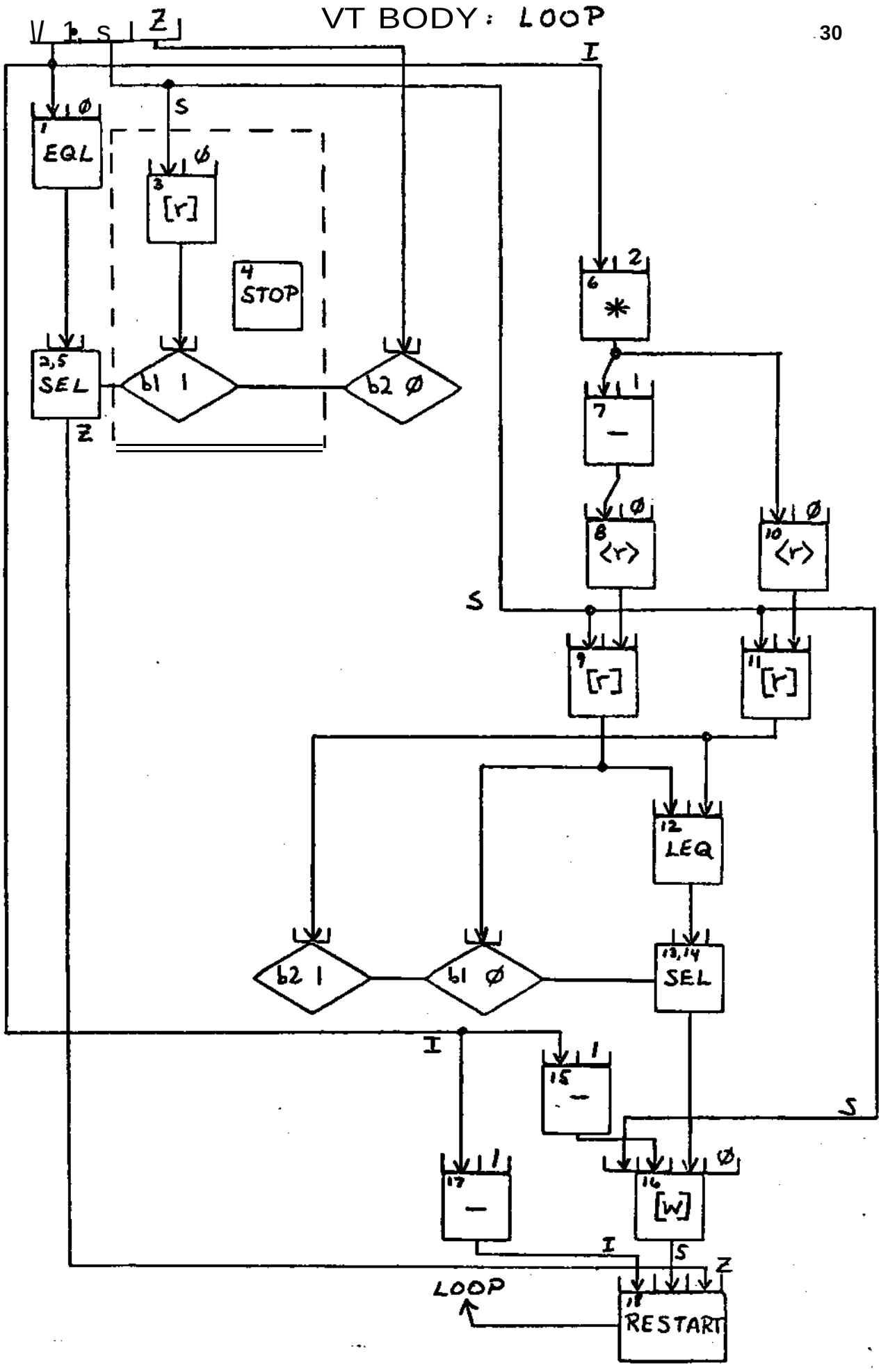


Figure 7: The VT for MIN after Select Motion

second operator's use-list so that it points to the first operator, and that the first operator's use-list is updated to reflect this. Then it must be shown That all pointers on the sequencing and branch lists which could point to the second operator are rearranged after it is removed. These are the double links in the linear list of operators and possibly the pointer to the operators in its branch. Since the routine does change all these, the actual data structure does still reflect the abstract structure.

3.4. Preliminary Evaluation of the VT

Studies in the optimization of algorithms have shown that when a great deal of searching needs to be done, it is often more efficient to sort the elements first, so that the searching is much simpler. Many examples of this appear, for instance, in [Aho7flJ This is what the VT is designed to do. A fully functional optimizer and partitioner would have to look for the same information on contexts, data dependencies, side-effects of procedure calls, and so on over and over again. The VT attempts to find all these things once and hold them, even through transformations. In this case the searches involved are multi-dimensional, so that there must be a number of different accessing mechanisms, each organized in a different manner. The structure is necessarily complicated. On the basis of experience gained with the transformations described above, it appears that a good compromise has been reached on the amount of information stored in the VT. All of the information there is useful and frequently used, and the addition of any more would make the structure not only more unwieldy but virtually impossible to control under a transformation like select motion.

It might seem that certain transformations would be easier to perform on the GOB. But there are two problems with that. The very feature that makes the GDB so clean and easy to use, namely that much of the information is stored implicitly in the tree structure through the relationships of nodes to one another, also tends to make certain searches and arbitrary changes difficult or impossible. Information about which operators preceed others and about whether operators are in the same branch is clear from a top-down search, starting from a root common to both, but is not clear at all from a bottom-up search. Therefore all searches must start from the top *oi* the tree and all contexts passed through must be remembered. Furthermore, since the GDB only references variable names it is not immediately clear when values and operations are identical.

To see the implications of this, consider again the redundant operator elimination. It would not be possible with the GDB to presort the operators by type, since then as operators were taken off of the sorted list, it would not be known how they were reached from the root. Therefore instead of a series of small lists to be searched, it would be necessary to compare every operator with every other one,' an n! algorithm. Furthermore, even though two

operators referenced the same variable, all operations occurring in between them would have to be checked to determine whether they could have affected that variable. And if one of the intervening operations was a procedure call, that whole procedure would have to be checked to determine whether it could have changed the value of the variable. Finally, even if the transformation could be recognized, it could not always be displayed in the GDB format. If the operator to be eliminated were part of an expression, for example, its output would not have a name, so that there would be no way of referencing it from another part of the tree without redefining the GDB and removing some of its nicer properties.

Table I shows some statistics on the operation of the VT translator. The first two columns show the comparative sizes of the VT and the GDB in words of memory, exclustze of text and identifiers, which are shared by the two structures. It can be seen that for small descriptions they are about the same, but for larger ISPs the VT grows faster. The relation for large descriptions is that the VT grows as the square of the GDB. The reason is primarily the way large ISP descriptions are written. They are made highly modular, as they should be for various reasons, with many layers of procedure calls. Moreover, there is little partitioning usually, so that many of the middle and high level routines have access to most of the variables and can alter most of the variables. As a result, in the VT these are translated into VT-bodies with huge input and output lists, and all calls to them also have large input and output lists. Moreover, if these routines are called inside selects, those selects also have large lists of inputs and outputs. The effects multiply as the description goes to higher and higher levels.

The size factor does make the VT somewhat more awkward for the largest ISPs. On the other hand, it is not wasted space. Usually the actual implementation is not as modular as the ISP. One valuable use of the VT would be to perform inline expansion of VT-bodies in order to simplify the control. The analysis of where variables are used which is contained in the VT could also be used to do a better partitioning, rearranging VT-bodies and selects so as to cut down the connectivity. This could simplify the data paths, make partitioning onto chips and boards easier, and make the design more testable.

The next two columns in Table I represent the processing time for each ISP. It can be seen, that this is fairly small and only grows linearly with the size of the ISP.

The last two columns show the time to look through the entire VT for possible transformations and perform them if found. The column "Redun" refers to the redundant operation elimination while "Selmo" refers to the select motion. It can be seen that the search through the entire description takes little time. Even "Redun," which checks all possible combinations of operators for equivalence only increases in time linearly with the

size of the GDB.

At this point, probably the greatest disadvantage of the VT is its unfamiliarity. It is a different structure from what people are used to and even more importantly, understanding it requires looking at algorithms in a different way. Designers seldom think in terms of "values" or data flow. This problem will be alleviated if a good user interface can be provided so that people can gain experience in manipulating the VT. It is especially important to provide simple, flexible accessing mechanisms which can be used either interactively or as procedures called from other programs.

ISP	GDwds	VTwds	Pass1	Pass2	Redun	Selmo
MIN	630	548	.35s	.60s	.33s	.45s
MINIS	1723	2137	1.06s	1.75s	.75s	.35s
PDP8	3889	5530	4.68s	4.61s	1.61s	.40s
VIDEO	5838	9042	5.45s	6.00s	2.05s	.23s

4. Future Work

4.1. Accessing Routines

A standard set of accessing routines to search or extract specific information from VTs should be built. The system should include a module to build a VT in core from the file representation, display specified parts of it, and save it when requested. There should be routines which find specified VT-bodies or operators and give various types of information about them as requested. It should be possible to call these routines from another program, but there should also be a command interpreter to allow interactive use.

4.2. Transformations

A full complement of VT tranformation routines should be built. These also should be accessible through a command interpreter. Initially they will probably be used interactively. However, the ultimate goal is to implement a system which will take an ISP description along with design criteria and apply the transformations selectively so as to optimize the design according to the given criteria.

One approach would be to generate and evaluate alternative descriptions as was done in EXPL. However, it is generally felt that the design space is just too large to make such an approach fruitful. The alternative would be to find types of measurements on the VT that would make it possible to predict what kinds of transformations would pay off, an approach

somewhat similar to the one taken in the Design-Style Selector. Much work needs to be done in this area.

4.3. Measurements

The measurement package built by Lawson must be reimplemented on the VT, There are two possible approaches. One would be to make the measurements directly from the VT. This would be simple enough, since the same information is there as is in the GDB and is just as available. However, it would mean rewriting his search routines. The other would be to build his data structure again from what is in the VT, and then use the same search and measurement routines. This would be a quick fix-up, but less efficient in the long run.

4.4. The Partitioner

An intelligent partitioner must be built. There are two parts to this problem. First it must be determined what features in the partitioning would lead to the best possible implementations of the data path and controller designs for any set of design criteria. Secondly, algorithms must be found which recognize ways that the VT can be rearranged so as to optimize those features identified as desireabie. Serial-parallel tradeoffs are important here, but so are register and processing-element useage, modularity and a number of other factors. How are these recognized and how do they interact? To what extent is it possible to predict the speed, cost, etc., of the final design from the shape of the VT at this level?

4.5. Data-Path Allocator

There are a number of possibilities which must be looked at at this level. It is hoped that the VT will provide much useful information to allow optimization of register usage, data path sharing and so on. For example, the lifetime of a value is easy to find from the list of uses associated with that value. Values with nonoverlapping lifetimes may share storage elements. Once registers have all been assigned, the interconnectivity of the data paths can be found by scanning the operator list and finding where the value of each input is stored and where the output is stored. Registers may be reassigned and processing elements allocated so as to simplify the data paths.

As noted earlier, it would be advantageous to include all the information about the path graph in the VT. This would make cross-referencing operations and the data path elements which perform them much simpler and also allow certain measurements and transformations to be repeated at this level where appropriate.

The allocator in cooperation with the module binder should make sure that operators in the

VT at this level correspond to operations which *are* actually performed by the hardware elements. This would mean redefining or factoring certain VT operators. This can be done in the VT. In fact, in some cases it has already been done by the VT translator. One example is the representation of array accesses.

4.6. Control Allocator

If the VT handed down by the data path allocator is in the form of a series of concrete operations on hardware elements, each one referring to definite storage elements for its sources and destinations, then the control design can concentrate on microcode or control state-and-signal optimization. In fact, many microcode optimizers and machine code generators take as inputs lists of "quadruples," each containing an opcode, sources and destination, along with a list of possible successors for branches. See for example [Dasg74]. Actually, even at the highest level the VT operator nodes are in this form. The difference is that not all operations correspond to hardware primitives and not all values are bound to storage elements.

At the control-design level, some reordering of the operations and reworking of the control is still desirable. It is hoped that information in the VT will prove useful for recognizing these also.

It must be remembered that this outline of future directions is tentative at best. Consideration of the possibilities for using the VT at other levels of the system has only just begun, and no real experimentation has been done as yet. However, the kind of information needed at the partitioner and allocator levels does seem to be readily accessible in the VT. This fact and the desirability of unifying the system by having all levels share a common representation of the design make further analysis and experimentation with the VT seem worthwhile.

Appendix A: The VT File Format

The ISP-to-VT translator saves the VT in an ASCII file. The first line of the file is a header line which includes *he* name of the file, and the time and date generated. This is followed by three sections. The first has all the entity nodes, the second a list of all constants used in the VT_f and the third the actual behavioral description organized by VT-bodies. Each section has a two or three line heading, with each line beginning with an "!". The first line indicates which section it is, i.e. "Entity Declarations," "Constants," or "VT-bodies," The remaining line(s) give the definitions of the various fields used in each descriptor.

There are certain conventions which are observed throughout the VT file. They are as follows:

- V indicates an empty field, except when used as an opcode
- A number contained in "<>" indicates a word or field size
- "7" means "contained in" or "of," referring to the context to which something belongs
- *V▷^{N} indicates a jump or other transfer of control, with the destination specified by the label following
- Inputs to branches and operators are enclosed in "()".
- All entities in the entity section, constants in the constant section, operators in a given VT-body, and outnodes in a given list are numbered consecutively. Each such "ID" number is prefaced by a letter indicating what that object is. These IDs are the principle means of cross-referencing in the VT.
- Attributes and qualifiers attached to entities, operators and branches appear immediately following the objects they describe. Attributes are in GDB format, as described in [Barb77a]. Qualifiers are either identifiers or strings or pairs of the form identifienstring, and are enclosed in "{}".

Entity Declarations

The first field of an entity declaration gives its ID. The ID letter indicates what kind of entity it is, "s" for a sectionlist-header, "r" for a carrier, "m" for a map, and "v" for a VT-body. The next field is the entity in which this entity was declared. For example, a formal parameter of a procedure which is VT-body v5 would have Zv5 in the "Decl in" field.

The next field is a collection of 18 flag bits, shown exactly as they are stored in the entity node. Numbering from the right, with the rightmost bit called bit 0, the meanings of the flags are as follows.

Position	Set if
0	Declared as a Process
1	A Map
2	AVT-body
3	A Slisthead
4	Has a bit structure
5	Has a word structure
6	VT-body contains a delay or wait
7	VT-body contains a LEAVE anywhere but at the end
8	VT-body can be called with a RESTART
9	VT-body can be RESUMEd
10	VT-body is body of a REPEAT in ISP
11	Carrier is defined as GLOBAL
12	Word size of mapped array is multiplier by wordfactor
13	Word size of mapped array is divided by wordfactor
14	VT-body can be part of a loop
15-18	For internal use

The next two fields are the word structure and bit structure exactly as given in the ISP. The last field on the first line is the name attached to that entity in the ISP description.

Entities which are maps have a second line in their descriptions. The first line is the ID of the "target," that is, the carrier onto which the mapping is made. The next two fields apply only to mapped arrays. If the wordsize of the map is different from the wordsize of the target, the first of these two fields, the wordfactor, tells how it is changed. A "*" indicates that it is multiplied by the following integer, while a "/" indicates that it is divided. Thus if an *array* of sixteen bit words is mapped onto an array of eight bit words, the word factor would be *2. The next field gives the offset from the lowest-numbered word in the target where the map begins.

The last field on the second line for a map is the bit offset, the offset from the rightmost bit of the target where the mapped word begins.

Constants

Each constant in the list has three fields. The first is its ID, prefaced by a "c" to indicate a constant, the second is its (decimal) value, and the third its size in bits. Note that constants with the same value but different sizes are considered to be distinct.

VT-bodies

Each VT-body in the last section has a header, which is just a repetition of the description of the VT-body from the Entities section, followed by a list of all its formal parameters and

inputs in outnode format, followed by a list of its operators, followed finally by a list of its putputs in outnode format.

Each outnode is described by four fields. The first is its ID. The letter at the beginning of the ID indicates the type of value which it describes as follows: "f" for a formal parameter of a VT-body$_f$ V for a VT-body input, "o" for a VT-body output, and "p" for a value produced by an operator.

The second field contains the ID of the carrier in which the value is stored, if any has been assigned. Initially this comes directly from the ISP. The next field is the size of the value in number of bits. The final field gives the name associated with the value in the ISP, if any.

Operators are specified by ID, Opcode, Inputs, and Outputs. Context information is shown by position in the list of operators and special tags inserted to show where branches begin and end. The first field for any operator is its ID, always prefaced by the letter "x". The next field is its opcode. The opcodes of operators which correspond to ISP primitives are denoted by the same symbol or string used in ISP, e.g. + and RESTART. Operators defined especially for use in the VT and their designations are as follows:

Opcode.	Meaning
[r]	Array read
[w]	Array write
<r>	Bit field read
<w>	Bit field write
CALL	Call procedure
SELECT	Beginning of Select
ENDSEL	End of Select
DIVERGE	Begin Parallel Branch
MERGE	End Parallel Branch
PADO	Pad left with O's
PADS	Sign extend

If the operator references another VT-body (with a CALL, RESTART, RESUME, or LEAVE), the ID of the VT-body prefaced with an "©" comes next. Also given are the first four letters of the name of that VT-body to aid in identification.

Next all the inputs to the operator are listed. Each input of course references an outnode connected with some value, so the list of inputs is actually a list of IDs referring to various outnocles, each enclosed in "()". Each outnode is referred to by the ID of the operator or VT-body to which it belongs, a "." and the ID of the outnode itself. Thus xl2.p2 refers to the second output of operator xl2, while v6.i3 refers to the third input of VT-body v6.

Constants have a "*" before the V, since they are not attached to any other construct. If the value has a name associated with it, the first four letters of the name are also given, separated from the ID by a V. For a constant the value is given, separated by an "».

After the input list, there is a list of all the outputs for that operator, one per line in the outnode format described above.

A SELECT construct must show not only the inputs and outputs for the select and each branch, but also which operators are associated with each branch. The listing of operators associated with the select are begun with a dummy SELECT operator. It has no inputs and no outputs, serving mainly to mark the list.

Each branch in the select begins with a line having 6>bi in the first field, where i is the number of the branch, and 7xn in the next field, where xn is the ENDSEL operator of which this is a branch. The last field is of the form [range], where "range" indicates the value(s) of the selector input for which this branch is taken.

After the first branch line are listed all the operators which are in that branch. At the end of this list is a line marking the end of the branch. This line starts with two fields, «bi, and 2xn, which correspond to the numbers at the beginning line of the branch. Then there are listed all the inputs to that branch in the same format as inputs to an operator.

After all of the branches have been listed, the select is closed with an ENDSEL operator. This corresponds to the real select node in the VT. It has one input, the selector, and its outputs are all the values produced by the select.

The other construct which refers to lists of operators is the DIVERGE-MERGE pair, which translates the ISPS V. The beginning of the parallel branches is marked by a DIVERGE operator with no inputs or outputs. The beginning of each parallel branch is marked by a line of the form ^>di 7,xn, where the following operators belong to the ith parallel branch of the diverge operator xn. This branch runs until the line @di+l Zxn is encountered. The entire construct is ended with a MERGE operator, again with no inputs or outputs.

The VT files for the original MIN algorithm and it two transformed versions follow.

"VALUE TRACE;MIN.VTRlX338m41]; 2 Dec 78;4:22 PM.

```
!                    ENTITY DECLARATIONS
!Typ/ID DecI  in     FLAGS                Uord Structure  Bit Structure   Na
!     Map            To      Ud Fact Ud Offs Bit Offs

si       *          888688068080881088     *               *              MI
rl       %sl        668000000000010000     *               <5:B>          N;
r2       %sl        088888880000110000     [8:1273         <15:0>         S;
r3       %sl        000008000000010000     >V              <15:B>         Z%
r4       2sl      * 088888800000310000     >V              <5:8>          I;
v5       %sl        088800000001000100     Vf              >\             ST
v6       %vS        088168888161888188     it              ft             LO


j                              CONSTANTS
I                    ID        Value   Size
                     cl        8       <G>
                     c2        8       <2>
                     c3        8       <7>
                     c4        2       <3>
                     c5        1       <2>
                     c6        8       <8>

j                              VTBOOIES
ii0        OPCODE   (INPUTS)
           OUTPUT:  ID        CARRIER SIZE    NAME

v6         ' n v w  688188888161888168        >V              *              LO
                    il        r4      <6>     I
                    i2        r2      <1G>    S
                    i3        r3      <1G>    Z
xl         EQL  •   (vG.iljI)         (Vf.cl-8)
                    pi        it      <1>     it
x2         SELECT
ebl        %x5      [0THERU3
x3          irJ     (v6.i2:S)         Ut.c2-8)
                    pi        r3      <1G>    Z
x4         STOP
=bl        %x5      (x3.pl:Z)
teb2       %x5      18]
=b2         %x5     (v6.i3:Z)
x5         ENDSEL   (xl.pl)
                    pi        r3      <1G>    Z
xG         it       (*.c4-2)          (v6.il:I)
                    pi        it      <9>     *
x7         -        (x6.pl)           (*.c5-l)
```

```
                    pi      ft      <10>    ft
x8      <r>         (x7.pl)         (ft.c6-8)
                    pi      ft      <7>     ft
x9      [rl         (v6.i2:S)       (x7.pl)
                    pi      ft      <1G>    ft
xlB     ft          (>v.c4-2)       (v6.il:l)
                    pi      ft      <9>     ft
xll     <r>         (xl8.pl)        (ft.c6=0)
                    pi      ft      <7>     ft
xl2     tr]         (v6.i2:S)       (xlB.pl)
                    pi      ft      <16>    ft
xl3     LEQ         (x9.pl)         (xl2.pl)
                    pi      ft      <1>     ft
xl4     SELECT
ebl     2x26        [0]
xl5     ft          (ft.c4-2)       (v6.il:l)
                    pi      ft      <9>     ft
xlG     <r>         (xl5.pl)        (ft.c6=B)
                    pi      ft      <7>     ft
xl7     tr]         (vG.i2:S)       (xl5.pl)
                    pi      ft      <16>    ft
xl8     —           (vG.il:l)       (ft.c5-l)
                    pi      ft      <7>     ft
xl9     [Ml         (v6.i2:S)       (xl8.pl)        (xl7.pl)        (ft.c6=0)
                    pi      r2      <1G>    S
=bl     %x2G        (xl9.pl:S)
eb2     %x2G        til •
x28     it          (ft.c4-2)       (v6.il:l)
                    pi      ft      <9>     ft
x21     -           (x20.pl)        (ft.c5-l)
                    pi      ft      <18>    ft
x22     <r>         (x21.pl)        (ft.c6-0)
                    pi      ft      <7>     ft
x23     [r]         (v6.i2:S)       (x21.pl)
                    pi      ft      <16>    ft
x24     _           (vG.il:l)       (>v.c5»l)
                    pi      ft      <7>     it
x25     [ш]         (v6.i2:S)       (x24.pl)        (x23.pl)        (ft.c6-8)
                    pi      r2      <16>    S
«b2     %x26        (x25.pl:S)
x26     ENDSEL      (xl3.pl)
                    «pl     r2      <1>     S
x27     —           (vG.il:l)       (ft.c5-l)
                    pi      r4      <G>     I
x2S     RESTART     evG:LOOP        (x27.pl:l)      (x2G.pl:S)      (x5.pltZ)
                    ol      r3      <1G>    Z
                    o2      r2      <1G>    S
                    o3      r4      <6>     I

v5      Xsl         888888880001000100      ft      1       ST
                    il      r4      <G>     I
                    i2      r2      <1G>    S
                    i3      r3      <16>    Z
```

xl	CALL	ev6:L00P		(v5.il:l)	(v5.i2:S)	(v5.13:Z)
		Pi	r3	<1G> Z		
		P2	r2	<1G> S		
		P3	r4	<6> I		
x2	LEAVE-	ev5:STAR		(xl.p3:I)	(xl.pltZ)	(xl.p2fS)
		ol	r4	<G> I		
		o2	r3	<16> Z		
		o3	r2	<1G> S		

3UVN 3ZIS U3!HdV3 01 UndinO
(smdNi) 3ao3do on
S310091A i
88T888TBI8BB8BT8B8 S^A% 9A
88T688TBTB888BTBB8 S^A% 9^A
88I888T0000000888B l*% SA
0068T0000000000000 T^s% *?**
000010608880000088 1^s% 3
80001T00000000888B 1^s% 2
000010006000000008 1^s% 1
B8BI88888888888888 * X^s
S1NV1SN03
8Z1.S an|e_A ai
SN0UVUV133Q A1I1N3
SOVld
*UV 8i7SBT^s8Z AON ^t [Tt?UU8££X] HiA'VNIU^s33Vai 301VA

```
                pi      ft      <18>    ft
x8      <r>     (x7.pl)         (ft.c6«0)
                pi      ft      <7>     ft
x9      [r]     (v6.i2:S)       (x7.pl)
                pi      ft      <16>    ft
xl8     <r>     (x6.pl)         (ft.c6«B)
                pi      ft      <7>     ft
xll     [rJ.    (vB,i2:S)       (x6.pl)
                pi      ft      <16>    ft
xl2     LEQ     (x9.pl)         (xll.pl)
                pi      ft      <1>     ft
xl3     SELECT
ebl     2x18    [0]
xl4     —       (v6.il:I)       (ft.c5-l)
                pi      ft      <7>     ft
xl5     M       (v6.i2:S)       (xl4.pl)        (xll.pl)        (*.c6-B)
                pi      r2      <16>    S
=bl     2x18    (xl5.pl:S)
eb2     2x18    [U
xlG     -       (v6.il:I)       (ft.c5-l)
                pi      ft      <7>     ft
xl7     [u]     (v6.i2:S)       (xl6.pl)        (x9.pl)         (ft.c6=0)
                pi      r2      <16>    S
«b2     2x18    (xl7.pl:S)
xl8     ENOSEL  (xl2.pl)
                pi      r2      <1>     S
xl9     -       (v6.il:I)       (ft.c5-l)
                pi      r4      <6>     I
x20     RESTART sv6:L00P        (xl9.pl:I)      (xl8.pl:S)      (x5.pl:Z)
                ol      r3      <16>    Z
                o2      r2      <16>    S
                o3      r4      <6>     I

v5      2sl     • 088000088081880100    ft      *               ST
                il      r4      <6>     I
                12      r2      <1G>    S
                i3      r3      <16>    Z
xl      CALL    ev6:L00P        (v5.il:I)       (v5.i2:S)       (v5.i3:Z)
                pi      r3      <16>    Z
                p2      r2      <16>    S
                p3      r4      <6>     I
x2      LEAVE   ev5:STAR        (xl.p3:I)       (xl.pltZ)       (xl.p2:S)
                ol      r4      <6>     I
                o2      r3      <16>    Z
                o3      r2      <16>    S
```

• VALUE TRACE,niNB.VTR[X338firi41];24 Nov 78;12:13 PM.

```
!                    ENTITY DECLARATIONS
!Typ/ID Decl in       FLAGS                  Uord Structure  Bit Structure   Na
!    hap              To      Ud Fact Ud Offs Bit Offs

si      ft       888888888808601088         ft              *               MI
rl      %sl      886880800000010000         ft              <5:0>           N;
r2      %s\      800080000000110000         [8:1271         <15:8>          Sj
r3      Xsl      886880088688818888         ft              <15:0>          Z;
r4      %sl      800000000000816000         ft              <5:8>           I;
v5      %sl      888888000001000100         ft              ft              ST
vB      *v5      088188886181888188         ft              *               LO

!                          CONSTANTS
!                  ID      Value   Size
                1 cl      8       <G>
                  c2      8       <2>
                  c3      8       <7>
                  c4      2       <3>
                  c5      1       <2>
                  cG      8       <8>

!                          VTBOOIES
!IO     OPCODE  (INPUTS)
        OUTPUT: ID      CARRIER SIZE    NAME

v6      %v5     888188688181888188      *               ft              LO
                i1      r4      <6>     I
                i2      r2      <1G>    S
                J3      r3      <1G>    Z
xl      EQL     (v6.il:I)       (ft.cl><0)
                pi   .   ft     <1>     ft
x2      SELECT
ebl     2x5     [OTHERU]
x3      [r]     (v6.i2:S)       (ft.c2>-8)
                pi   ;   r3     <1G>    Z
x4      STOP
«bl     %x5     (x3.pl:Z)
eb2     %x5     [01
«b2     %*S     (vG.i3:Z)
x5      ENDSEL  (xl.pl)
                pi      r3      <16>    Z
x6      ft      (ft.c4-2)       (vG.il:I)
                pi      ft      <9>     ft
x7      -       (x6.pl)         (ft.c5-l)
```

```
                    pi      ft      <18>    ft
x8      <r>         (x7.pl)         (ft.c6«B)
                    pi      ft      <7>     ft
x9      [r]         U6.i2:S)        (x7.pl)
                    pi      ft      <16>    ft
xie     <r>         (x6.pl)         (ft.cG=8)
                    pi      >v      <7>     ft
xll     [rl         (v6.i2:S)       (x6.pl)
                    pi      ft      <16>    ft
xl2     LEQ         (x9.pl)         (xll.pl)
                    pi      ft      <1>     ft
xl3     SELECT
ebl     2x14        10}
=bl      %xl4       (xll.pl)
eb2     %xl4        [11
=b2      %xl4       (x9.pl)
xl4     ENDSEL      (xl2.pl)
                    pi      ft      <16>    ft
xl5     —           (vG.il:I)       (ft.c5-l)
                    pi      ft      <7>     ft
xlG      tu]        (vG.12:S)       (xl5.pl)        (xl4.pl)        (ft.c6»B)
                    pi      r2      <16>    S
xl7     —           (vG.il:I)       (ft.c5-l)
                    pi      r4      <6>     I
V18     RESTART     ev6:L00P        (xl7.pl:I)      (xlG.pl:S)      (x5.pl:Z)
                    ol      r3      <16>    Z
                    o2      r2      <16>    S
                    o3      r4      <6>     I

v5      *9l         eeeeeeeeeeeieeeiee      ft      *               ST
                    il      r4      <6>     I
                    \2      r2      <16>    S
                    • 3     r3      <1G>    Z
xl      CALL        ev6:L00P        (v5.il:I)       (v5.i2:S)       (v5.i3:Z)
                    pi      r3      <16>    Z
                    p2      r2      <16>    S
                    p3      r4      <6>     I
x2      LEAVE       @v5:STAR        (xl.p3:I)       (xl.pl:Z)       (xl.p2:S)
                    ol      r4      <G>     I
                    o2      r3      <16>    Z
                    o3      r2      <16>    S
```

References

[Aho74] Aho, Alfred V., James E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Mass., 1974.

[Aho77] Aho, Alfred V. and Jeffrey D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977.

[Alle76] Allen, F. E. and Cocke, J., "A Program Data Flow Analysis Procedure," *CfiCM,* Vol. 19, No. 3, March, 1976, pp. 137-147.

[Barb73] Barbacci, Mario R., "Automated Exploration of the Design Space for Register Transfer (RT) Systems," Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, 1973.

[Barb77a] Barbacci, Mario R., Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek, "The ISPS Computer Description Language,* technical report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Barb77b] Barbacci, Mario R. and Andrew W. Nagle, "An ISPS Simulator," technical report, Department of Computer Science, Carnegie-Mellon University, 1977.

[Barb77c] Barbacci, Mario R. and Daniel P. Siewiorek, "Evaluation of the CFA Test Programs Via Formal Computer Descriptions," *Computer,* Vol. 10, No. 10, October, 1977.

[Bell71] Bell, C. Gordon and Allen Newell, *Computer Structures: Readings and Examples,* McGraw-Hill, New York, 1971.

[Bell72] Bell, C. Gordon, John Grason, and Allen Newell, *Designing Computers and Digital Systems,* Digital Press, Maynard, Mass., 1972.

[Breu72] Breuer, M. A. (ed.), *Digital System Design Automation, Volume I: Theory and Techniques,* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[Casw78] Caswell, Hollis L et al., "The Oregon Report: Basic Technology," *Computer,* Vol. 11, No. 9, September, 1978, pp. 10-18.

[Dasg74] Dasgupta, Subrata and John Tartar, "Some Aspects of Parallelism in Microprograms," Department of Computing Science, University of Alberta, 1974.

[Dijk68] Dijkstra, Edsger, "Notes on Structured Programming," Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* Academic Press, New York, 1968.

[Frie69] Friedman, T. 0. and S. C. Yang, "Methods Used in an Automatic Logic Design Generator (ALERT)," *IEEE Trans. Computers,* Vol. C24, 1969, pp. 593-614.

[Hafe78} Hafer, Louis J. and Alice C. Parker, "Register-Transfer Level Digital Design Automation: The Allocation Process," *Proceedings of the ISth Design Automation Conference,* Las Vegas, Nevada, June, 1978, pp. 213-219.

[Heat72] Heath, F. G., "The LOGOS System," *IEE Conference on CAD,* 1972, pp. 225-230.

[Hech77] Hecht, Matthew S., *Flow Analysis of Computer Programs,* North-Holland, New York, 1977.

[Hoar72] Hoare, C. A. R., "Proof of Correctness of Data Representations," *Acta Informatica,* Vol. 1, No. 4, November, 1972, pp. 271-281.

[Law$78] Lawson, Gregory Lee, "Design Style Selector, An Automated Computer Program Implementation," M. S. Project Report, Department of Electrical Engineering, Carnegie-Mellon University, 1978.

[Leiv77] Leive, Gary W., "The Binding of Modules to Abstract Digital Hardware Descriptions," Ph.D. Thesis Proposal, Department of Electrical Engineering, Carnegie^Mellon University, 1977.

[Lewi77] Lewin, Douglas, *Computer-Aided Design of Digital Systems,* Crane Russak, New York, 1977.

[Mann74] Manna, Zohar, *Mathematical Theory of Computation,* McGraw-Hill, New York, 1974.

[Nagl78a] Nagle, Andrew W., "Automatic Design of Sequencers for the Control of Digital Systems," Ph.D. Thesis Proposal, Department of Electrical Engineering, Carnegie-Mellon University, 1978.

[Nagl78b] "Automatic Synthesis of Microcontrollers," Bell Telephone Laboratories! Holmdale, New Jersey, 1978.

[Nagl78c] Nagle, Andrew W., "Specification of a Control Graph for Design Automation," Unpublished Report to RT-CAD Research Group, Carnegie-Mellon University, Oct. 19, 1978.

[Siew76] Siewiorek, Daniel P. and Mario R. Barbacci, "The CMU RT-CAD System, An Innovative Approach to Computer Aided Design," National Computer Conference, New York, 1976.

[Snow78] Snow, Edward A., "Automation of Module Set Independent Register-Transfer Level Design," Ph.D. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, 1978.

[Thom77] Thomas, Donald E., "The Design and Analysis of an Automated Design

Style Selector," Ph.D. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, 1977.

[Toko77] Tokoro, Mario et al., "An Approach to Microprogram Optimization Considering Resource Occupancy and Instruction Formats," Proceedings of the 10th Annual Workshop on Microprogramming, October, 1977, pp. 92-108.

[Wulf71] Wulf, William A., D. B. Russell and A. Nico Haberman,"Bliss: a Language for Systems Programming," *CACM*, Vol. 1, No. 12, December, 1971, pp. 780-790.

[Wulf77] Wulf, William A., Richard K. Johnson, Charles B. Weinstock, Steven O. Hobbs, and Charles M. Geschke, *The Design of an Optimizing Compiler*, Elsevier, New York, 1977.