CORAL II: Linking Behavior and Structure in an IC Design System

Robert L. Blackburn, Donald E. Thomas, Patti M. Koenig

Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes a technique for maintaining very fine grained links between a behavioral specification and an automatically generated VLSI structural implementation. CORAL II execedes previous systems in the scope of design representations involved and the complexity of the relationships handled. The design representations used are described, as are the behavioral transformations that may be applied and the types of design choices that may be made. The complications introduced by these transformations and design decisions are discussed. Some possible applications of this work are outlined and an existing graphical interface for examining the synthesized design and its relationship to the behavioral specification is described.

1. Introduction

One problem with automated VLSI design systems is that the designer may not have a firm understanding of the design generated by the tools. This problem is aggravated by the complexity of designs that automated tools allow a designer to work with and by the fact that the designer is less intimately involved in the design process than he would be were he to do the design by hand. Also, the lack of detailed information relating the input specification to the synthesized design forces design tools to run open loop, the only feedback path being through the designer and his possibly imperfect understanding of the synthesized design. One way to overcome these difficulties is to generate highly detailed information about the relationship between the input and output design descriptions automatically during synthesis.

The CORAL II system described in this paper keeps track of relationships between the design specification read in by the design tools in the System Architect's Workbench [11] (the Workbench) and designs the tools produce. Figure 1-1 shows the general relationship of CORAL II to the design environment. Behavioral specifications expressed in procedural form are the primary input for the Workbench. Typical specifications are for large, complex parts such as microprocessors, digital signal processors, or digital feedback controllers. The Workbench applies behavioral transformations to improve the quality of the design, and then synthesizes a register-transfer level structure and a control sequence to implement the specified behavior. CORAL II collects information about the relationships between the input and output specifications as the design tools run.

CORAL II maintains very fine grained information about the correspondence between the basic components in each design representation by linking the individual operators and values in a behavioral specification to the specific submodules and operations that implement and store them. Furthermore, CORAL II works in a design system that is capable of producing a wide variety of implementations from a given input specification. For instance, a single behavioral specification may be implemented as a simple processor, a collection of concurrent processors, or a pipeline.

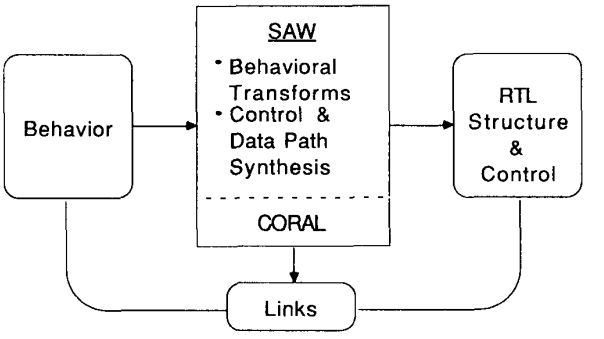

Figure 1-1: The CORAL II system, part of the System Architect's Workbench (SAW), maintains links between parts of the behavioral specification and the register-transfer level design output by the Workbench.

Previous systems only dealt with hierarchically decomposed structural representations and usually only with gate and transistor level designs. CORAL II pushes linking up into higher levels of abstraction, works with a mixture of behavioral and structural design representations, and maintains links despite behavioral transformations.

An automated design system that provides such information about the relationships between the specification and the synthesized design opens the door for a wide variety of design analysis tools and improvements in synthesis techniques. One important application of this information is formal verification of synthesized designs, a task which requires that corresponding values in the input and output designs be designated and proven equivalent. Another area is symbolic debugging of the synthesized designs, allowing the designer to control and probe a simulation of the synthesized design in terms of the original behavioral specification. Iterative design becomes a possibility when, after design analysis reveals a part of the design that does not meet expectations, the relational information can point to the part of the input specification that needs to be altered or re-implemented. One application that has been implemented and will be described in this paper is an interactive tool that graphically displays the relationships between the behavioral specification and the synthesized structure and control sequence.

2. Design Representations and Links

This work is concerned with the domain of design representations depicted in Figure 2-1. The Workbench design programs work from a behavior and structure specification and synthesize a behavior and structure implementation. In the structural domain, synthesis involves specifying submodules such as the ALU, register, and sequencer shown in the figure and their interconnections. The structural domain is hierarchical as indicated by the nesting of the implementation modules within the specification module. Each module has its own behavioral

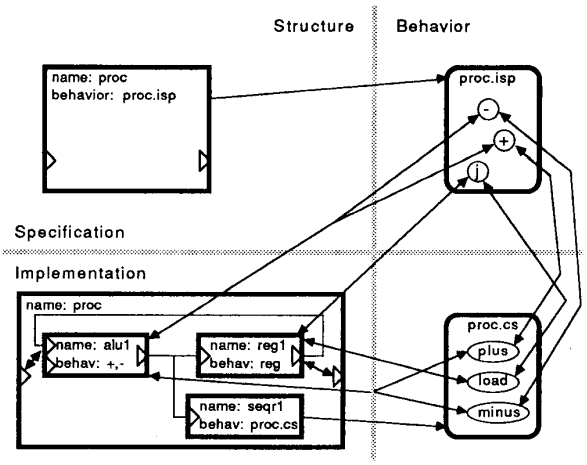

Figure 2-1: Design representations fall in both the structural and behavioral domains at both the specification and implementation levels. Links between components of the design representations are shown as arrows.

description.

In the context of the Workbench, specifications tend to be for parts like microprocessors and digital signal processors. The specification module is the part to be designed and the ISPS file describes the processor's behavior. Modules in the implementation are at the level of registers, ALUs, and MUXs.

The goal of CORAL II is to describe links between components of designs in the various sections of the space pictured in Figure 2-1; in particular, the links between elements in the behavioral specification and elements in the structural and behavioral parts of the implementation. The links from structural modules to their behavioral descriptions and parent module are easy to determine. The links connected to elements in the behavioral specification are difficult to keep track of because of the complexity of the relationships between those elements and elements in the implementation. The complex ways in which those relationships evolve add to the difficulty of the problem. The reasons for the complexity and a means for overcoming it will be discussed in Section 4.

2.1. ISPS

ISPS [1] is the language used to express the behavioral specification to the Workbench. ISPS bears a resemblance to procedural programming languages such as PASCAL; variables are declared and used to store values, arithmetic and logical operations are provided, if-then, decode (switch), and looping control constructs are available, subroutines are supported, etc.

The elements that the present work focuses on are the data and control operators and the values stored in variables. Values in ISPS may be referred to by specifying the line and token numbers for the particular appearance of the variable in which the value is stored. If the 22nd line in an ISPS file was:

```
PC = PC + 1
```

the value stored into PC at this point could be referred to as (22, 1), the value stored at the first token on line 22. Data and control operations may be specified in a similar manner; the "+" operator above may be referred to as (22, 4).

2.2. Value Trace

Value Trace [9] (VT) is an augmented dataflow language used as the internal representation in the Workbench (VT descriptions do not appear in Figure 2-1). A Value Trace is produced from ISPS by a compiler. The VT produced by the compiler has a dataflow graph for each of the basic blocks in the ISPS; each graph has a node for each operator in the ISPS. The flow of control between graphs is regulated by control operators corresponding to those found in the ISPS. While the order of execution within a basic block in ISPS is implicitly specified by the placement of operators within the text, execution within a VT dataflow graph is restricted only by the explicitly stated data dependencies. There are no variables in VT, only arcs along which values flow between operators.

A behavioral specification expressed in VT consists of a set of graphs, each graph has a set of operators, and each operator has a set of output values. All graphs, operators and values in a VT are numbered in order of appearance within their parent construct. Any operator in the design may be specified by giving its graph and operator numbers. Any value may be specified by giving its graph, operator, and output numbers.

2.3. Structure and Control Specification

The data path designs synthesized by the Workbench are expressed in SCS, the Structure and Control Specification language [12]. SCS describes the modules, their input and output ports, and the connections between them. For fairly simple modules such as an ALU, the module's functions are simply listed in its description. For more complex modules such as an entire microprocessor or a sequencer, the behavior is described in a separate file.

Sequencer behaviors are described in a separate file as a set of states in a state machine. The outputs of each state are described in terms of activations of the other modules in the structure.

Values in the SCS may be specified by naming a module that the value passes through or is stored in and the state at which the value reaches that module. Operations are specified by naming a module and the state(s) during which the module performs the operation.

2.4. Links

The purpose of this work is to describe the relationships between elements in the behavioral specification and elements in the implementation. We have described how operators and values may be identified by line and token numbers in the ISPS, by index numbers in the VT, and by module and state IDs in the implementation. To indicate that the operation at line 30, token 4 in the ISPS is linked to the operation performed by procr_7 at state 515 in the structure, we would write:

```
{(30, 4) : (procr_7, 515)}
```

Intermediate links to the Workbench's internal design representation may be expressed in a similar manner using the VT index numbers.

3. Prior Work

3.1. Symbolic Debugging

Symbolic debugging of programs is closely related to the present work. A symbolic debugger must have knowledge of the relationship between parts of the source language program and corresponding parts of the machine language version. The task of

keeping track of those relationships is usually fairly simple due to the straight forward translation algorithms used by most compilers.

Optimizing compilers, on the other hand, complicate the relationships between the two versions of the program by applying transformations to improve the machine code. Hennessy [4] and Zellweger [14] have reported research on building symbolic debuggers to work with optimizing compilers. Several of the techniques they used, particularly Hennessy's "tagging" of the internal statement tree, are applicable to the domain of hardware design.

Symbolic debuggers, including ones like Hennessy's, depend primarily on a statement map to indicate the relationship between the two program representations. A statement map is a list of the machine level instructions that correspond to the beginning of each statement in the high level language. This is sufficient because, even in an optimized program, operations are done predominantly in the order in which they appeared in the high level program and all operations are executed one at a time.

This convenient relationship between operations can not be depended upon in a hardware design system. In the particular case of the Workbench, the VT representation used as an intermediate form preserves very little of the original ordering of operations, order of execution being limited only by the data requirements of the operators. The hardware design tool is likely to reorder operations in a way that bears little resemblance to the original. The design tool may also schedule independent operations to execute in parallel and it may cascade operations that are data dependent on one another within a single, atomic control state. Consequently, knowing the next operation to be executed in the hardware tells far less about how the state of the hardware design should compare to the state of the behavioral specification.

3.2. SABLE

The SABLE environment was developed by D.D. Hill to support multi-level simulation of IC designs [5]. SDL is used to describe the instantiation of modules and the interconnections between them and ADLIB is used to describe the behavior of each module type. The designer initially specifies a design by defining a module and writing a description of its behavior. He refines the design by decomposing the module into a set of submodules and interconnections and specifying a behavior for each submodule type. The refined design may be simulated and the results compared to a simulation of the original specification.

The SABLE environment falls short of CORAL II in that the specification and implementation are only linked at the external ports. Because the linking information is so sparse, specifications and implementations can only be compared in their entirety; one cannot pick out a portion of the implementation to see if it correctly implements some portion of the specified behavior.

CORAL II indicates the relationship between the specification and the implementation at the finest granularity allowed by the current state of the design. These links are shown in figure 2-1 as the arrows between operations and values in the behavioral specification and modules and operations in the implementation.

3.3. Design Verification

Several research projects have addressed the problem of formally verifying that a given design satisfies the specification. The work reported by Carter and Joyner, et. al. [3, 6] was aimed at verifying that microcode written for an existing data path correctly implemented the instruction set of the computer being emulated. The system developed in that work accepted instruction set level descriptions (behavioral specifications) of the micro- and macro-machines and the micro-code to be verified. It also accepted a hand generated description of what the relationship between values in the two machines *should* be (represented in figure 2-1 by arrows from components of the behavior specification to components of the implementation). The system attempted to prove that the intended relationship between the two machines did exist.

This work is actually an example of an application of a system like CORAL II. CORAL II is part of a system which automatically generates a data path and microcode program to implement the specified behavior. It automatically generates a description of the links between the specification and implementation to support formal verification of the synthesized design.

3.4. Design Automation at USC

Research on relating different design representations in an automated design system has been done by Parker et. al. at USC [7, 8]. The systems developed there used either sets of bindings [7] or matrices of binary variables [8] to denote the relationship between values and operations in a data flow description and storage and operator modules in a structural description. The intention is that these data structures will be filled in by the synthesis tools as they create a new design representation from a specification. However, the reported work lacks the capability to describe the relationships between design representations when behavioral transforms are applied. CORAL II is designed to provide this capability.

3.5. CORAL I

A previous incarnation of the CORAL system was described in [2]. CORAL I was created as an initial attempt at achieving some goals of this work. CORAL I worked only with GDB (a parsed form of ISPS) and VT design representations, it linked only values, not operators, and it did not link the behavioral specification with the structure. CORAL II exceeds CORAL I in the range of design representations linked, the complexity of the Workbench transformations it can handle, and the completeness of the information provided.

3.6. Summary

There are many similarities between establishing the relationship needed to perform symbolic debugging and establishing the relationship that is desired between hardware design representations, especially when the debugger is designed to work with optimized programs. There are, however, substantial differences between the realm of programming and IC design, differences that require new solutions to be developed.

There are also similarities between CORAL II and the hardware projects described above. All four projects involved an attempt at describing the relationship between different design representations, but CORAL II goes beyond them in the completeness of the information provided and the complexity of the relationships it can describe.

4. CORAL II in the System Architect's Workbench

The Workbench is a package of programs that may be used to synthesize an RTL implementation for a specified behavior. This section will outline each of the three major phases in the Workbench: translation of the behavior specification to an internal format, behavioral transformations to improve the design, and actual synthesis of the control sequence and data path. This

section will also describe the steps CORAL II must take during each of these phases to gather information about the relationship between the behavioral specification and the synthesized control sequence and data path. Finally, we will describe how the information from the various design tools is combined into a statement of the relationship between specification and implementation.

4.1. ISPS to Value Trace Translation

The first stage in the Workbench translates the ISPS behavioral specification into the Workbench's internal format, Value Trace. This step operates much like normal program compilation. There is generally a one-to-one relationship between operators in the ISPS and operators in the VT.

For the purposes of this work, we would like to know the corresponding VT operator for every ISPS operator, and the corresponding VT value for every ISPS value. Similarly, we would like to know what, if any, ISPS operator (value) corresponds to each VT operator (value). The translation program "knows" which ISPS element it is looking at when it generates each VT operator and value. It has been modified to record this linking information during the translation process. The result is an auxiliary file generated by the translator that consists of lists of operator-operator pairs and value-value pairs. These lists cover every operator and value in the two descriptions.

4.2. Behavioral Transformations

The behavioral transformation package in the Workbench is much like the optimizing portion of an optimizing compiler. The transformations may be divided into four groups: data transforms, control flow transforms, code motion transforms, and process transforms. The data transforms include operations like redundant and dead operator elimination. The control flow transforms include operations like inline procedure expansion and its inverse, procedure formation, as well as some less standard operations such as *select* (case statement) combination. The code motion transforms can move operations into or out of a *select* construct. The process transforms are of a different character and are less commonly found in optimizing compilers. They can change a procedure into a process (*i.e.* make it an independently controlled entity) and change the pattern of communication between processes. More complete descriptions of the transforms may be found in [13].

Let us examine the case of a procedure formation transform. This transform may be applied to a part of an existing procedure as in Figure 4-1. Here, the first line of LOOP in 4-1(a) is left in place while the remainder of the procedure is removed to a new procedure called STAGE2 and replaced with a *call* to STAGE2 in 4-1(b).

The CORAL II system must have information about the correspondence between values and operators in the original VT and those in the VT resulting from the transformation. The transformation package gathers this information by means of a system of tags similar to Hennesy's graph labels. Whenever a new operator or value is created in the VT, a tag is attached to it giving the ID of the original operator or value from which the new one was created. In the case of procedure formation, the operator and values in the newly formed procedure correspond exactly to the operators and values removed from the original procedure. For example, the value (C, 4)* in Figure 4-1(b) will be tagged with

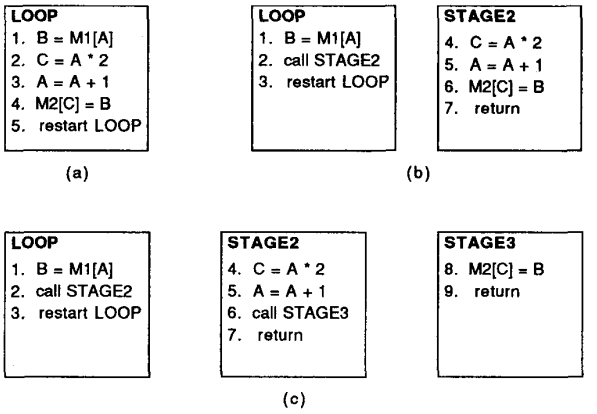

Figure 4-1: The original procedure (a), the two procedures that result from one application of procedure formation (b), and the three procedures that result from a second application (c).

(C, 2), the ID of the corresponding value in the original description in 4-1(a). The new *call* operator doesn't correspond to any operator in the original VT and so will have a blank tag. The *call* operator does, however, have three outputs (from the three variables modified within STAGE2) which correspond to the original values in LOOP. These three values, (A, 2), (C, 2), and (M2, 2), have the tags (A, 3), (C, 2), and (M2, 4) respectively.

When the transformation phase is completed, a file is written out containing the IDs in all the tags in the VT paired with the IDs of the new elements to which the tags are attached; thus, we have a record of the correspondence between the operators and values that existed before transformations were applied and those that exist after.

As another example, let us examine an application of the pipe stage creation transform. There must be at least three communicating processes for this transform to be applied so we will first apply procedure formation again to STAGE2 (see Figure 4-1(c)) and process formation to the three resulting procedures (see Figure 4-2(a)). Upon noting that the information passed from STAGE3 to STAGE2 is, in turn, passed directly on to LOOP, the pipe stage transform may be applied, resulting in the communication links shown in Figure 4-2(b). Note that STAGE3 now sends directly to LOOP and the *receive STAGE3* and *send LOOP* operators in STAGE2 have been removed. In this situation, we will say that the *send LOOP* operator in STAGE3 corresponds to both the *send STAGE2* operator in the original STAGE3 and the *send LOOP* operator in the original STAGE2, and that the values input to the *send LOOP* operator in STAGE3 correspond to the values input by the *send LOOP* operator in the original process STAGE2. These correspondences are noted by attaching tags with the appropriate identifiers to these elements as the transformation is done. For instance, the operator (send, 12) in Figure 4-2(b) will carry the tags (send, 14) and (send, 10).

4.3. Design Synthesis

The final stage in the Workbench creates a control sequence for operations in the design and a data path to carry out the control sequence's instructions. The synthesis program may elect to store the values that were originally stored in a single variable in the ISPS in different registers in the structure at different times; it may store values that were stored in different variables in the same register at different times; or, many values may not be stored at all

*A (<variable/operator>, <line number>) format will be used here for clarity

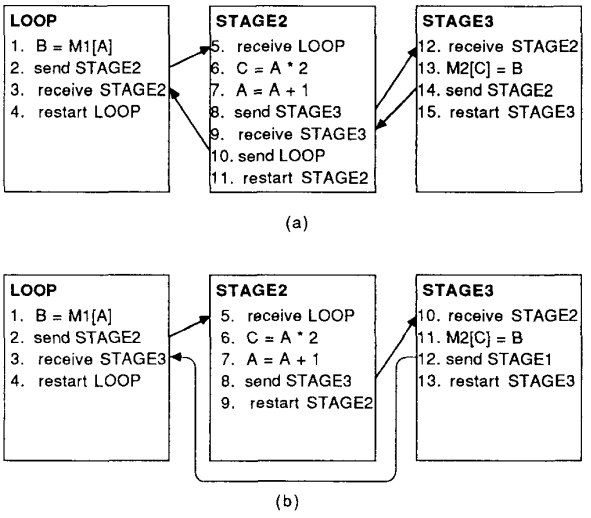

Figure 4-2: Three communicating processes before pipe stage creation (a) and after (b).

if they can be created and used in the same state. Similarly, multiple operations of different types may be implemented in a single module (*e.g.* an ALU). Operators that were sequentially ordered in the ISPS specification may legitimately be executed in a different order or in parallel in the synthesized implementation. All the effects of these decisions must be recorded for CORAL II.

Once again, the approach taken by CORAL II is based on a system of tags. The synthesis program was altered so that, as each VT operation is bound to a module, the VT operation's ID is recorded on a tag associated with the implementation level operation and the module in which the operation is performed. Similarly, as each value is bound to a register or a data path module such as a MUX, the value's ID and the state during which it enters the module are recorded on a tag attached to the module. When synthesis is complete, the resulting structure is examined and a file is written containing the information from the tags attached to each module. This file describes the relationship between the elements in the implementation and the operators and values in the VT.

4.4. Linking Things Together

After the Workbench has synthesized a data path to implement the specified behavior, we have three link files generated by the three stages of the Workbench. Each link file describes the relationship between the elements in the description read in and those in the description created by that stage.

A separate program sits off to the side of the other design tools and gathers the linking information generated by each of the three main phases of the Workbench. This program joins the input-output links from each phase and puts them together to form a web reaching from individual operators and values in the behavioral specification to individual modules and states in the synthesized structure and control sequence. The set of links that result from joining the separate link files together is described as a web because elements in any design representation may be linked to zero, one, or more elements in an adjacent representation. When this is compounded over the three layers of links that arise from the three stages of the Workbench, we see that the end-to-end relationships can be quite complex. The linking program can traverse this web to find the relationships between elements in the ISPS and individual modules and specific states in the implementation. These relationships are represented in Figure 2-1 by the arrows reaching from elements in the behavioral specification to modules and to elements in the sequencer's behavior in the implementation.

Having done all of the above, we have gained an explicit and detailed statement of the relationship between the input specification and the implementation created by a set of automated design tools. Having this relationship in hand, we open the door for a variety of multilevel design aids like those discussed in the next section.

5. Applications

The overriding goal of CORAL II is to improve the quality and usability of automated design tools by providing information about the relationships between different design representations. We will now discuss some of the potential applications of this information and describe one that has been realized.

5.1. Potential Applications

One potential application is in the area of formal verification of designs. As discussed above, this task involves attempting to prove that pairs of values in two different design representations will indeed be identical for all input sequences. Formal verification of automatically synthesized designs is potentially useful as long as the design tools themselves have not formally been proven correct. Proof of design correctness may be carried out by *symbolically* simulating each of the design representations and proving that the resulting expressions are equivalent. The most obvious advantage of having an explicit statement of the relationship between the two design representations is that it makes the task of choosing expressions to be proven quite simple. Moreover, since the difficulty of proving expressions equivalent grows exponentially with the length of the expression, and the expressions may be expected to grow exponentially with the number of statements between paired values, it is very important to find as many pairs of values to prove equivalent as possible. An automated means of finding pairs of values to compare has obvious advantages.

A second potential application is symbolic debugging of synthesized designs. The existence of the linking information produced by CORAL II allows one to build a symbolic debugger interface for a register-transfer level simulator. Such an interface would allow a designer to check the performance of the synthesized design without requiring that he first gain an understanding of the synthesized design detailed enough to allow him to set breakpoints and probe for values in terms of registers and hardware control states. Instead, the designer could control and examine the simulation of the structural implementation in terms of the ISPS behavioral description.

Yet another potential application is an iterative design system. In such a system, either a human or automatic design critic could examine a synthesized implementation and suggest that certain portions be implemented differently. With a system like CORAL II one can determine which parts of the specification are related to the portion of the implementation to be redone and which are not. This knowledge can be used to focus the changes on the selected parts of the design during the next pass. Alternately, if it is decided that a flaw in the implementation is the result of faulty specifications, the links recorded by CORAL II will indicate which portion of the specification needs to be changed.

5.2. SEE-SAW

All the above applications are suggestions for possible ways to use information about how a design specification and its implementation are related. We now turn to one application that has already been built, SEE-SAW.

SEE-SAW was developed to provide a visual aid to the designer in understanding the relationships between a behavioral specification, the synthesized structure, and the synthesized control sequence. SEE-SAW uses the information gathered by CORAL II to illustrate correspondences between items in the behavior, structure, and control. Graphical "tracing" enables the user to select items in any one of the three representations and see corresponding items highlighted in the other two.

SEE-SAW runs on a DEC VAXstation II/GPX Color Workstation using the X Window Manager. Separate X windows are used to display the ISPS, structure, and control sequence design files and to accept commands and display general information (see Figure 5-1). Each window is equipped with pull down menus of commands for manipulating the information in that window. Commands are provided to move around the text in the behavior and control windows and to manipulate the graphic display of the structure in its window. The control window has commands to control the types of trace information displayed, the colors used, screen dumps, etc. A box at the bottom of each window is used to display messages and prompt for input.

The linking information provided by CORAL II is used by SEE-SAW to find corresponding items in each of the windows. The designer uses the mouse to select an item in any of the three display windows and invoke a trace on that item. A trace uses the linking information to find all the corresponding items in the other windows and place them in a trace set together with the selected item. All the elements in the trace set are then highlighted. A trace example is shown in Figure 5-1. The user has selected the ALU module, procr_7, in the structure. All operations in the control sequence that affect procr_7 are highlighted as are all the operations in the ISPS that procr_7 implements (not all highlighted items are visible in the control sequence window).

SEE-SAW may be used by a designer to probe the synthesized design. By invoking a trace on a particular module he can see how it is used to implement the specified behavior. He can select a value in the ISPS and see where it is generated and stored in the structure and at what control states these actions take place. Selecting a memory access operation in the ISPS reveals not only the memory module where the information is stored, but also all the other processors and data path modules that are used to get the address and data to and from the memory for that particular operation. By examining the design descriptions in light of each other, the user can more easily gain an understanding of how the synthesized implementation functions.

6. Conclusions

This paper has described a means for keeping track of the relationships between design representations in an automated VLSI design system. The system deals with high level, algorithmic behavioral specifications and synthesized structural designs at the level of ALU's, registers, and multiplexers. The system keeps a very fined grained record of the relationship between design representations, tracking individual operators and values in each design, and does so despite behavioral transformations that may be applied in order to improve the quality of the resulting design.

A graphical designer's aid has been described. This tool helps designers understand and analyze an automatically synthesized design by interactively displaying the relationships between various parts of the the behavioral specification and the structure and control sequence created to implement it.

A system that maintains this kind of information opens the door for a variety of synthesis and analysis tools that can make use of a multi-level view of a VLSI design. These tools can help analyze the synthesized designs and verify their correctness, contribute to higher quality designs than is possible with a strict top-down procedure, and contribute to the designers understanding of the finished product. The capability to add these functions to automated design systems represents a significant advance in VLSI design technology.

Acknowledgements

This research was supported by the SRC under contract 86-01-068, by the CMU CAD Industrial Affiliates, and by a Hewlett-Packard / American Electronics Association Faculty Development Grant.

References

[1] M.R. Barbacci, G.E Barnes, R.G. Cattell, D.P. Siewiorek. *The ISPS Computer Description Language.* Technical Report, Carnegie-Mellon Univ., Aug., 1979.

[2] R.L. Blackburn, D.E. Thomas. Linking the Behavioral and Structural Domains of Representation in a Synthesis System. In *ACM/IEEE 22nd DAC Proc.*, pp. 374-380. IEEE Comp. Soc., June, 1985.

[3] W.C. Carter, W.H. Joyner Jr., D. Brand. Symbolic Simulation for Correct Machine Design. In *ACM/IEEE 16th DAC Proc.*, pp. 280-286. IEEE Comp. Soc., June, 1979.

[4] J. Hennesy. Symbolic Debugging of Optimized Code. *ACM Trans. on Prog. Lang. and Sys.* 4(3):323-344, 1982.

[5] D.D. Hill. *Language and Environment for Multi-Level Simulation.* PhD thesis, Stanford Univ., Mar., 1980.

[6] W.H. Joyner, W.C. Carter, G.B. Leeman. Automated Proofs of Microprogram Correctness. In *9th Annual Microprogramming Workshop Proc.*, pp. 51-56. IEEE, Sep., 1976.

[7] D.W. Knapp, A.C. Parker. *A Data Structure for VLSI Synthesis and Verification.* Technical Report CRI-85-19, Univ. of Southern California, Aug., 1985.

[8] A.C. Parker, L.J. Hafer. A Formal Method for the Specification, Analysis, and Design of Register-transfer level Digital Logic. *IEEE Trans. on CAD of Integrated Circuits and Sys.* CAD-2(1):8-18, Jan., 1983.

[9] E. Snow. *Automation of Module Set Independent Register-Transfer Level Design.* PhD thesis, Carnegie-Mellon Univ., Apr., 1978.

[10] D.E. Thomas, R.L. Blackburn, J.V. Rajan. Linking the Behavioral and Structural Domains of Representation for a Digital System Design. *IEEE Trans. on CAD of Integrated Circuits and Sys.* CAD-6(1):103-110, Jan., 1987.

[11] D.E. Thomas, E.M. Dirkes, R.A. Walker, J.V. Rajan, J.A. Nestor, R.L. Blackburn. The System Architect's Workbench. In *ACM/IEEE 25th DAC Proc.* IEEE Comp. Soc., June, 1988.

[12] J. Vasanthrajan. Design and Implementation of a VT-Based Multi-Level Representation. Master's thesis, Carnegie-Mellon Univ., Feb., 1982.

[13] R.A. Walker, D.E. Thomas. Design Representation and Transformation in the System Architect's Workbench. In *IEEE International Conf. on CAD Proc.* IEEE, Nov., 1987.

[14] P.T. Zellweger. *Interactive Source-Level Debugging of Optimized Programs.* PhD thesis, Univ. of California at Berkeley, May, 1984.

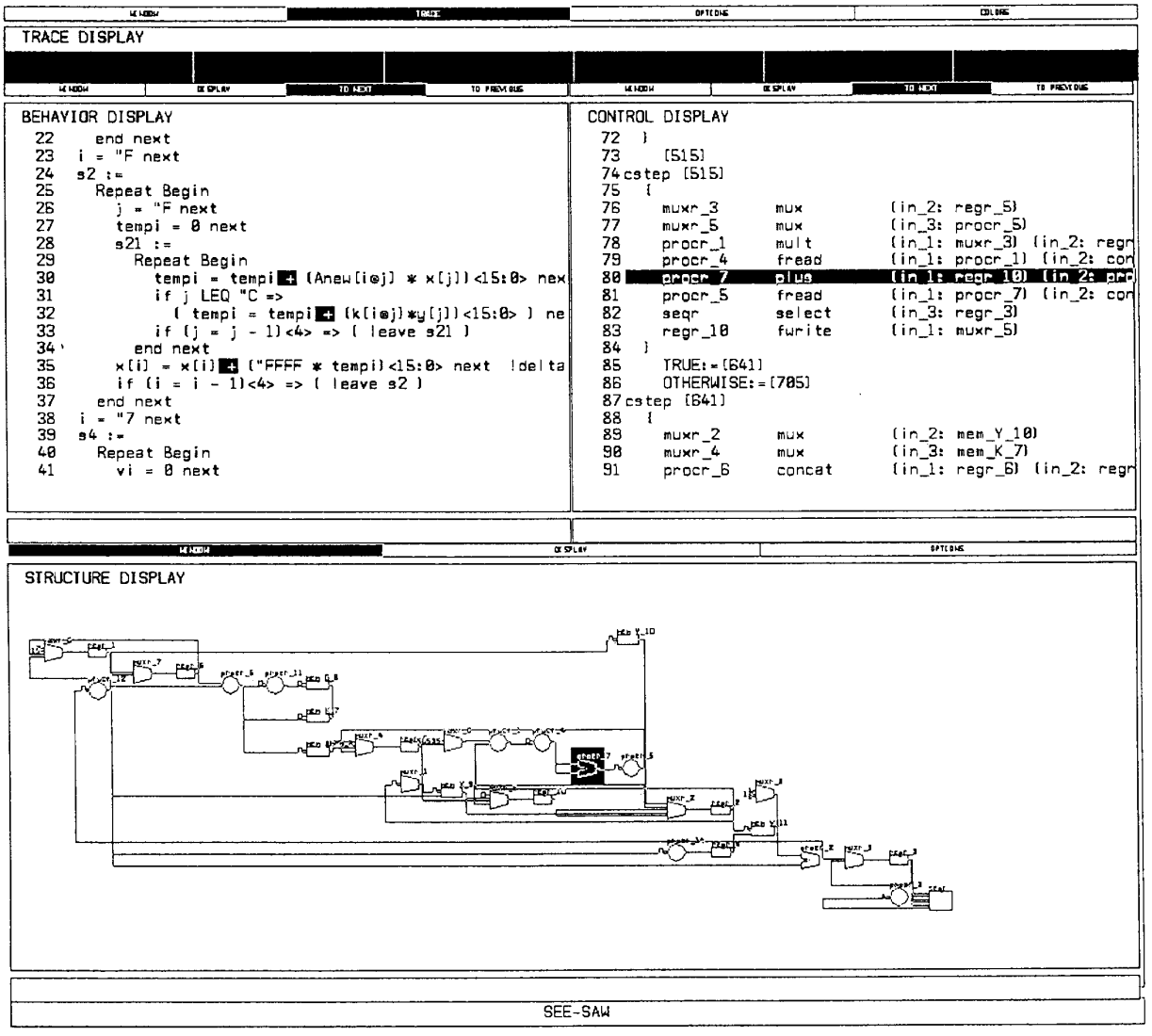

Figure 5-1: A SEE-SAW display with separate windows for the ISPS (top left), control sequence (top right) and structure (bottom). The ALU module, procr_7, has been seleted in the structure display and related items in the ISPS and control sequence have been highlighted.