Defining and Implementing a Multilevel Design Representation With Simulation Applications

by

John A. Nestor

DRC-01-07-82

April, 1982

Defining and Implementing a Multilevel Design Representation With Simulation Applications

M.S. Research Project Report

by

John A. Nestor

19 October 1981

Department of Electrical Engineering

Carnegie-Mellon University

Pittsburgh, Pennsylvania

This research was supported in part by the Hewlett Packard Company, the U.S. Army Research Office under Grant DAAG/29/79/C/0213, and the National Science Foundation under Grant ENG 78-25755.

Table of Contents

List of Figures

Acknowledgments

I owe thanks to many people for their advice and encouragement while I was completing this project. My advisor, Professor Don Thomas, provided me with the opportunity to work on an interesting project and deserves much credit for his encouragement, patience, and guidance throughout this project.

Credit is also due to the other members of my committee, Professor Steve Director and Dr. Mario Barbacci. Their willingness to provide help and advice has been greatly appreciated. Mario Barbacci deserves special mention for modifying the ISPS simulator and related software to interface with my software.

This project was very much a part of the CMU-DA research project. Interactions with the members of this group were both valuable for my project and personally enjoyable. In this group I would especially like to thank Gary Leive, Mike McFarland, Lou Hafer, Rich Cloutier, Jin Kirn, Andy Nagle, Alice Parker, Ted Kowalski, Mike Meyer, Dave Gatenby, John Lertola, Ron Dirusso, and Chia-Jeng Tseng.

An interesting summer spent with the Hewlett-Packard Design Aids group gave me some perspective on the more immediate needs of computer-aided-design in an industrial setting. Discussions with Bill Haydamack, Beatriz Infante, and Ravi Apte were particularly helpful.

Finally, I would like to thank my family and friends in Tennessee, Pennsylvania, and elsewhere, whose support and encouragement helped me enormously.

1. Introduction

1.1 Motivation

Research in digital systems computer aided design (CAD) has resulted in the development of computer-based tools that aid designers in many parts of the design process, including specification, documentation, verification of design correctness, and fabrication. Some particular CAD tools include simulators, logic synthesis programs, design rule checking programs, and programs that provide automatic fabrication of designs.

While these tools have very different functions, they all require a description of a design to work with. Traditionally, each CAD tool has required a description of a design in its own special input language or format. The cost and inconvenience of translating between different design descriptions to use different CAD tools has motivated the development of *design representations* that can be used as a common design description by several tools.

Design representations have been defined at several different levels of abstraction to describe a design at different stages of the design process. For example, an integrated circuit design is described at the most detailed level of abstraction in terms of the geometric information required to specify its fabrication. An example of such a design representation is Caltech Intermediate Form (CIF), which was developed to describe the layout of NMOS integrated circuits [Mead 80]. On the other hand, the same design might be described for specification and simulation purposes at a much higher level of abstraction that describes its behavior but contains little or no information about its implementation. An example this type of design representation is ISPS, which describes a design's behavior procedurally [Barbacci 81]. Design representations at intermediate levels of abstraction describe a design in terms of abstractions such as transistors, gates, registers, and higher level components.

1.1.1 The Hierarchy of Design Representations

A loose hierarchy of design representations has emerged, organized by level of abstraction. From the top level down, this hierarchy typically includes the following levels of abstraction:

- *Behavioral Level:* Describes the behavior of a digital system without specifying its implementation.
- *Functional Logic, or Register-Transfer Logic Level:* Describes a digital system implementation in terms of abstract components and their interconnections.

- *Structural Logic Level:* Describes a digital system implementation in terms of physically realizable components, such as integrated circuit packages or layout cells, and their interconnections.

- *Gate Level:* Describes a digital system implementation in terms of combinational logic components and their interconnections.

- *Circuit Level:* Describes a digital system implementation in terms of transistors and other primitive components and their interconnections.

- *Physical Layout Level:* Describes a digital system implementation in terms of the physical fabrication technology.

1.1.2 Multilevel Representations

A design representation that incorporates design representations at more than one level of abstraction is called a *multilevel representation.* Levels of abstraction in a multilevel representation are maintained in a hierarchical fashion so that components of a design at one level of abstraction can be related to corresponding components at a different level. There are two primary motivations for establishing a multilevel design representation:

1. The development of new CAD tools that require as input a description of a design at more than one level of abstraction.

2. The need for a methodology that can consistently manage large design representations.

Several new CAD techniques have been developed that require more than one level of abstraction as input. One of these techniques is multilevel simulation. Multilevel simulation is motivated by the high cost of accurate simulation of digital designs. In general, simulation at a low level of abstraction is accurate but slow and expensive. Simulation at a higher level of abstraction is less expensive but also less accurate. The goal of multilevel simulation is to simulate a design at more than one level of abstraction simultaneously. Parts of a design which are considered critical can be simulated at a low level of abstraction while the remainder of the design is simultaneously simulated at higher, less expensive levels of abstraction. The resulting simulation provides a tradeoff between the speed of high level simulations and the accuracy of low level simulations.

Multilevel simulation between the gate and circuit level has been the topic of extensive research. Multilevel simulators that mix gate and circuit levels of abstraction include SAMSON [Sakallah 80] and SPLICE [Newton 79]. In addition, multilevel simulation has been explored at higher levels of abstraction. Example simulators that work at higher levels of abstraction are SABLE [Hill 79] and SLIDE [Altman 80].

Another class of CAD applications using multiple levels of abstraction that has been explored less extensively involves measuring and relating information between levels of representation. For example, implementation, cost, and timing information may be measured at one level of abstraction and related to a higher level. This information can then be used to verify the correctness of the lower level representation through simulation and analysis.

A multilevel representation is also a potential tool for managing and maintaining consistency in large design representations. A generally recognized trend in digital integrated circuit design is the rapid increase in size and complexity of designs due to improvements in technology. The result of this increase in complexity is that it has become very difficult to maintain consistency between design representations at different levels. In addition, traditional CAD tools cannot handle very large design representations. A multilevel representation should be able to enforce consistency between design representations and aid in partitioning a desing representation into smaller, more managable pieces when needed.

Recent multilevel representations have evolved to meet the needs of single tools, most notably multilevel simulators. For example, the SAMSON multilevel simulator uses an input language that describes designs at the gate and circuit levels. Similarly, the SABLE and SLIDE systems describe designs at higher levels of abstraction. These multilevel design representations apply primarily to middle and low levels of abstraction; very little work has been done with higher levels of abstraction such as the behavioral level.

The CMU Design Automation (CMU-DA) project involves research about the problems associated with automating the design of digital systems. Previous research in this project has resulted in the development of software that automatically transforms a behavioral specification of a digital system into lower and lower level design representations until the final result is the description of a completed design at the structural logic level of abstraction. This project and associated software are described in more detail in the following section.

An interesting feature of the CMU-DA software is the definition and use of design representations at high levels of abstraction starting with an algorithmic behavioral representation. These representations provide a unique opportunity to define and build a multilevel representation that incorporates the behavioral level of abstraction. A multilevel representation that is based on a behavioral representation provides the possibility of several new applications. For example, information from a completed design may be extracted and related to the behavioral representation for high-level analysis and evaluation. In addition, it provides the basis for developing a multilevel

simulation technique that mixes a procedural behavioral simulation with lower level simulation. Also, a properly defined multilevel representation insures consistency between the behavioral representation and lower level representations.

This report describes the definition and implementation of a testbed multilevel representation based on the design representations generated by the current CMU-DA software. In addition, it describes the implementation of an application of the multilevel representation, timing abstraction, which allows behavioral simulation using timing information from a completed design.

1.2 Overview

The following sections of this repcrt describe the definition and implementation of a multilevel representation for the CMU-DA design representations. Section 2 describes the CMU-DA synthesis system in detail to provide the necessary background information for the project. Section 3 examines different ways to relate different design representations together and defines a specific method for the CMU-DA design representations. Section 4 describes the implementation in software of a simplified version of this multilevel representation, and Section 5 describes the implementation of an application of the multilevel representation, timing abstraction. Section 6 shows some simulation results of this application. Finally, Section 7 makes some conclusions about the complete project and presents some ideas for future work involving multilevel representations.

2. Background: The CMU-DA Project

Research in the CMU Design Automation (CMU-DA) project has concentrated on problems associated with automatical designing a digital system from a high-level description of that system's behavior. This research has resulted in the development of a methodology for automatic design of digital systems. This methodology was initially proposed in [Siewiorek 76]; a more recent description can be found in [Director 81]. Over the past five years this methodology has evolved substantially, but many of its underlying principles have remained the same. Some of these principles include:

- *Technology Independence.* The methodology should not be tied to a single specific technology, so that it can support many new and evolving technologies without major changes.

- *Design Space Exploration.* The methodology should support the exploration of design alternatives to find the "best" design in the space of possible designs given user constraints of cost, size, speed, etc.

- *Design by Successive Refinement.* The design process should be broken down into several tasks which are performed successively in a top-down fashion, starting with a behavioral representation of a design and creating successively more detailed design representations until a complete design is specified.

The present CMU-DA design methodology is shown in Figure 2-1. The design process is broken down into three groups of tasks: Optimization, Analysis, and Synthesis.

High level optimization and analysis tasks work with the behavioral specification of the design. Snow [Snow 78] and McFarland [McFarland 78, McFarland 81] have defined several transformations that can be applied to a behavioral specification that can improve performance of a design. The *Global Optimization* task performs these transformations where appropriate.

Design Style Selection is an analysis task. A design style is a particular approach to designing a digital system. Thomas [Thomas 77] has identified several different design styles including bus style, distributed logic, bit-slice microprocessor, and others. Design style selection analyzes a behavioral description and decides which design style is most appropriate for that description.

The desirability of several optimizing transformations is dependent on the particular design style chosen for a design. For this reason, Global Optimization and Design Style Selection must interact to be most effective.

Synthesis tasks transform the optimized behavioral description into an actual design. These tasks are Data-Memory Allocation, Logic Synthesis and Module Selection, and Control Allocation.

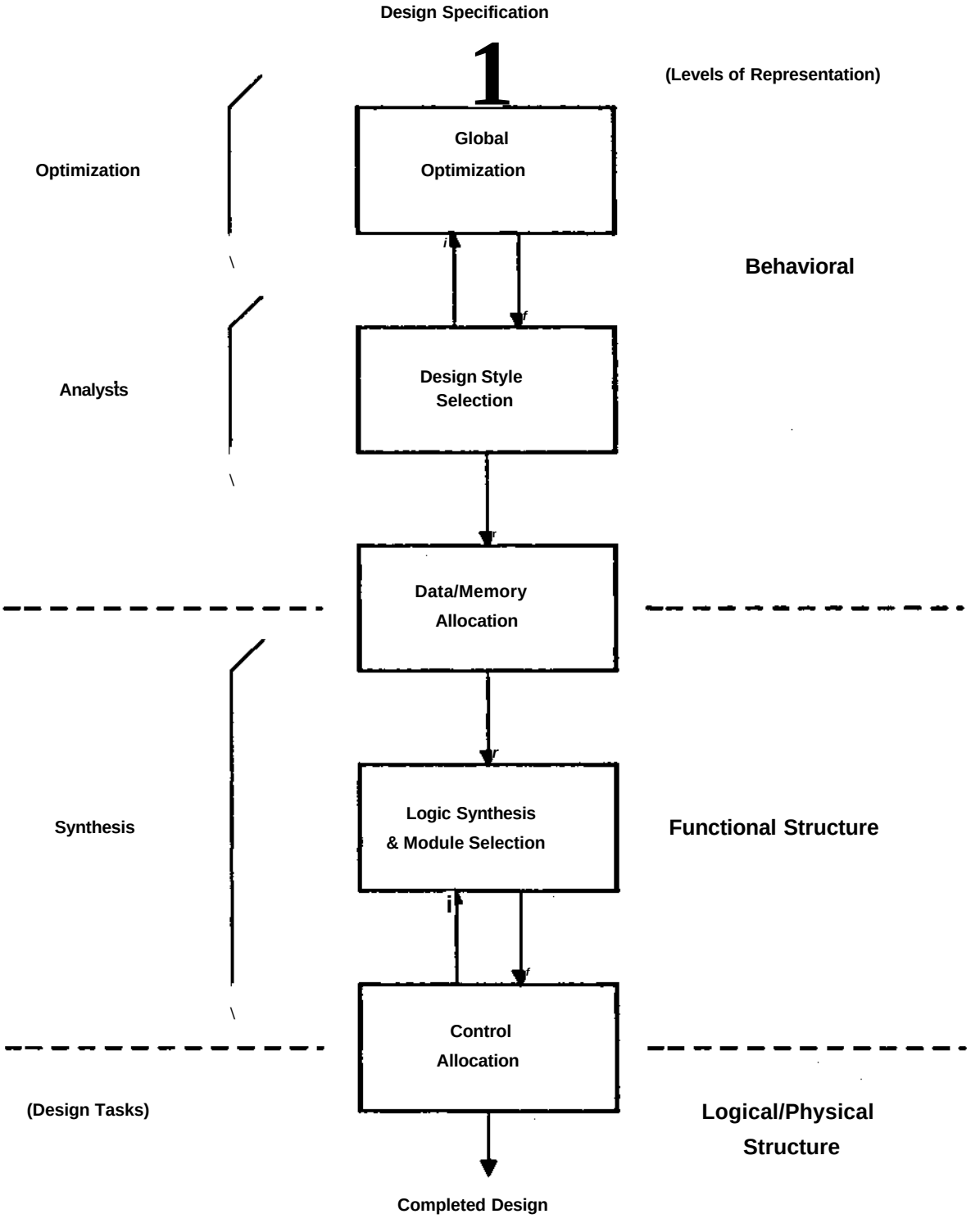

Figure 2-1: The CMU-DA Design Methodology

A major assumption in the CMU-DA design methodology is that digital system designs can be separated into a data part and a control part. *Data-Memory Allocation* generates the data part of the design in the design style specified by Design Style Selection. It specifies the data part design in terms of technology-independent abstract components such as registers, memories, multiplexers, and arithmetic and combinational operators. Control of the design is specified following data-memory allocation in terms of abstract operations that act on the components in the data part.

Logic Synthesis and Module Selection (LSMS) transforms abstract components specified by the data-memory allocator into realizable components in the desired design technology. These realizable components are called modules. Those abstract components which do not map directly into modules in the target technology must be transformed by the LSMS process into other components which do. LSMS is applied to the data part of the design following data-memory allocation and to the control part of the design following the next phase of the design methodology, control allocation.

Control Allocation generates the control part of the design and ties it to the data part of the design to complete the CMU-DA design process. Control allocation transforms the abstract control operations generated by data-memory allocation into specific signals to the components in the data part and adds components that form a control part to generate these signals in the proper sequence. The LSMS task must interact with the control allocation task to map components added for the control part of the design into the modules of the desired technology.

The result of this design methodology is a complete design specified in terms of modules from a particular technology. Previous research in high-level design using this methodology has used predefined sets of modules such as integrated circuit packages and integrated circuit standard cells. More recent CMU-DA research involves extending the current design methodology to include the development of new sets of modules for integrated circuits. This research has concentrated on developing circuit level tools for synthesis, optimization, and simulation of integrated circuit modules and is described in more detail in [Director 81].

2.1 Design Representations in the CMU-DA project

Corequisite with the definition of a methodology for automatic design is the definition of design representations. The CMU-DA design methodology uses three major levels of representation: Behavior, Functional Structure, and Logical/Physical Structure. These levels of representation correspond roughly to the behavioral, functional logic, and structural logic representations described in Section 1. The extension of this methodology to include lower levels of design will require the addition of design representations at the gate, circuit, and physical layout level.

The *Behavioral* representation specifies the desired behavior of q digital design without specifying its implementation. The behavioral representation used in the CMU-DA project is ISPS [Barbacci 81], a hardware description language resembling block structured programming languages such as ALGOL and PASCAL, but with additional features added to allow easier description of hardware behavior. ISPS describes register and memory storage elements as variables; behavior is specified by procedures that operate on these variables. The ISPS Simulator [Barbacci 80] allows a user to simulate and verify ISPS behavioral descriptions. An example of ISPS, a fragment of a description of the DEC PDP-8, is shown in Figure 2-2. This description shows some storage element variable declarations and an ISPS procedure that describes the effective address calculation implemented by a PDP-8.

The *Functional Structure* representation specifies the initial structural design of the data part and an abstract description of the operations that control the data part. Components in the data part represent primitive components that can be realized in most technologies such as registers, memories, multiplexers, and arithmetic operators. Control is specified by a sequence of operations on these components plus operations which control what sequence these operations are evoked.

The *Logical/Physical Structure* representation specifies the completed design in terms of components from the target technology plus any programming information that is required for components such as read only memories and programmable logic arrays. This representation makes no distinction between the data and control parts of the design, although information about the control part design may allows us to predict its performance using the abstraction of microcode operations.

2.2 CMU-DA Experimental Logic Synthesis Software

Several of the tasks in the CMU-DA design methodology have been implemented in computer programs. McFarland [McFarlandMS] has demonstrated some optimizing transformations as a precursor to implementing the Global Optimization task. Lawson [Lawson 78] has implemented the Design Style Selection task. Of particular interest, however, are the implemented synthesis tasks. Together these programs make up a system that allows a design to be created automatically from a behavioral specification.

```
•• Mp.state ••

MXHemory[0:4095]<0:11>,

••Pc.state"

PC\Program.Counter<0:ll>.
cpage\current.page<0:4>.
        .
        .
        .

••Instruction.Format••

1\instruction<0:ll>t

        op\operation.code<0:2>   := 1<0:2>,
        1b\indirect.bit<>        := 1<3>,
        pb\page.O.bit<>          := 1<4>,
        pa\page.address<0:6>     := 1<5:11>,

                .
                .
                .

••Address.Calculation"

eadd\effective.address<O:ll> :«
       Begin
       Decode pb c>
              Begin
              0 := eadd ='00000 0 pa
              1 := eadd = cpage 6 pa
              End Next
       If 1b «>
               Begin
               If eadd<0:8> Eql #001 => M[eadd] = H[eadd] + 1 Next
               eadd * H[eadd]
               End
       End,
```

Figure 2-2: An ISPS Example: PDP-8 Effective Address Calculation

2.2.1 The Distributed Design Style Data-Memory Allocator

In 1977 Hafer [Hafer 77] developed a program that implements the data-memory allocation task for the distributed design style. The data-memory allocator program uses as input Register Transfer Machine (RTM) tables, which were originally intended to drive the ISPS simulator [Barbacci

79, Barbacci 80]. RTM tables are generated from an ISPS description and contain a symbol table describing registers, memories, and temporaries and a statement table that describes behavior in terms of operations for an imaginary 3-address machine called the Register Transfer Machine. The data-memory allocator uses these tables to establish the defined registers and memories in the data part design; it then allocates operators, multiplexers, and interconnections to realize the remainder of the data part.

The data-memory allocator represents the data part it designs in terms of a directed graph called the *path graph.* The nodes of this graph represent the components of the data part design, e.g. registers, memories, and operators and interconnection elements. The edges of the graph represent interconnections between these elements and consequently the flow of data between them.

The control algorithm for the data part design is represented by a table called the Microsequence Table, which is derived from the RTM statement table. This table is a sequence of "data[91] operations on path graph components and "control" operations which describe the sequencing of the control algorithm. Data operations are represented in a "three address** format of destination and source components for the operation, similar to the intermediate languages used in some compilers [Aho 77]. However, each operation the microsequence table also refers to the specific operator and interconnection components used in the operation. Control operations implement ISPS control constructs for sequencing: conditional execution, procedure activation, etc.

Together, the path graph and the microsequence table make up the functional structure representation used by the CMU-DA software. The Data-Memory Allocator writes them in a tabular form in an ASCII file called the path graph file.

2.2.2 Logic Synthesis and Module Selection: SYNNER and the Module Database

Leive [Leive 81] has implemented a program called SYNNER[1] that performs the Logic Synthesis and Module Selection Task. SYNNER transforms components in a functional structure data path description into available components in the desired technology. The actual information about components of a technology are stored in a module database; SYNNER extracts this information for each technology when needed. Technologies presently supported by the module data base are TTL and RCA CMOS integrated circuit packages as well as Sandia integrated circuit standard cells.

---

[1] An acronym for Synthesizing Designer

SYNNER generates output in the form of an augmented path graph file. The path graph table is updated to represent the transformed data part of the design, including identification of the physical components they represent. The microsequence table is updated to relate to the modified path graph components.

2.2.3 The Control Allocator

Nagle [Nagle 80] and Cloutier [Cloutier 80] have implemented the Control Allocation task. This program uses as input the augmented path graph and microsequence table from SYNNER. controller and microcode to control the data part of the design. Information about what signals are required to control different components in the design are extracted from the module data base. The control allocator then synthesizes a controller whic! , supplies these signals in the proper sequence.

The control allocator first performs some optimizations on the microsequence table. Two optimizations of interest are performed. First, procedures defined in the microsequence table that are only called once are substituted inline at location in the microsequence table where they are called. Second, transfers of control such as LEAVE and RESTART are transformed into simple jump instructions.

When these optimizations on the microsequence table are complete, it constructs a controller and ties it into the data part of the design. Presently only one type of controller is generated, a microprogrammed sequencer that uses a counter for normal sequencing and a lookup table for conditional branches. If any procedures remain in the microsequence table after inline substitution, hardware is added to the controller to implement microcode procedure calls and returns, in addition to branches.

Because later phases of the control allocator require explicit information about the components that implement both the data and control parts of the design, LSMS is required for the components of the controller and is incorporated into the control allocator at this point. This task binds actual components to the controller, but does not perform many of the other transformations found in SYNNER, since the basic controller design is assumed by the later control allocator phases. It is also interesting to note that at this point the width of the controller hardware has not been fixed because the microcode word has not yet been formatted. However, the later phases of control allocation require the exact signals that operate on the controller knowledge of the components is required for these phases. For this reason the control allocator chooses hardware for the controller that can be replicated later to increase its width to fit the microcode word width.

The control allocator then breaks down Operations in the microsequence table into primitive operations called micro-operations. Operations that control sequencing in the microsequence table are broken down into micro-operations on the controller part of the design; other operations are broken down into operations on the data part of the design. Micro-operations are a collection of even more primitive operations on components in the design called device primitives; all device primitives in one micro-operation are executed in the same controller clock period. Information which maps device primitives into actual signals which control design components is stored in the module database; the control allocator extracts this information as needed.

The control allocator then determines potential parallelism between these micro-operations. This information is used to optimize micro-operations and format them into words in the microprogram. This formatting completes the control allocation task.

The completed design generated by the control allocator makes up the CMU-DA system's logical/physical structure representation. This representation is output as another, further augmented path graph file. The path graph table now contains the control part of the design as well as the data part; controller components are specified by path graph nodes like other components. Also, the file contains additional tables that specify programming for the control part ROMs.

2.2.4 Verifying CMU-DA Software Performance and Correctness

The performance of the design system up to but not including the control allocator has been examined in detail in several different efforts. An initial experiment generated designs for the data part of the DEC PDP-8/E [Parker 79] using the TTL and Sandia Standard Cell technologies. The intent of this experiment was to compare data part designs generated by the CMU-DA programs to actual designs in the same technologies. The TTL CMU-DA design was compared to the original DEC PDP-8 design. The CMU-DA implementation used about 30% more TTL chips than the original DEC PDP-8 implementation. The Sandia Standard Cell design was also compared to a standard cell design implemented by humans. This comparison was less favorable.

To validate the performance SYNNER alone, a design experiment was conducted that compared SYNNER's performance of the LSMS task to that of humans performing the same task [Leive 81]. This experiment used as design examples data path designs for a truncated PDP-8 design and a vending machine coin changer mechanism. Both of these examples were generated by the data-memory allocator. For both examples, the LSMS task was evaluated using two different technologies, again TTL and Sandia Standard Cells. SYNNER's performance on these two examples was

compared to the human designs in both Sandia Standard Cells and TTL. The results of this experiment showed that the performance of SYNNER was comparable to that of human designers.

A more recent project to validate the correctness of designs generated by the entire collection of CMU-DA synthesis programs including the control allocator is now nearing completion [DiRusso 81]. The goal of this project is to fabricate a working TTL PDP-8 using a design specified by the CMU-DA programs. This project has uncovered several errors in the designs generated by the system, particularly in the Control Allocator program. These errors are primarily program bugs and not conceptual errors. The fabrication project also made clear several parts of the control allocator output for which the logical/physical structure design specification is incomplete. In particular, control inputs to chips in the design are specified only implicitly, and bit steering logic for the control part of the design is not specified at all. It is expected that future versions of the control allocator will correct these deficiencies.

2.2.5 Some Conclusions About the CMU-DA Synthesis Software

The present CMU-DA synthesis programs provided a first cut at automatically generating designs from an ISPS behavioral description. However, these designs are by no means optimal. The limitation of designs to the distributed design style is a contributing factor, as is the limitation of the control part design style to a microprogrammed design style.

Furthermore, design representations were defined in a somewhat ad-hoc manner. In particular, the definition of the path graph and microsequence table contains many irregularities that make representation of some types of designs very difficult. The final path graph design file is also somewhat incomplete in that it does not describe control connections or controller bit-steering logic. Finally, the design representations at all levels contain artifacts of ISPS the RTM tables which are not really relavent to digital design. All of these factors detract from the overall performance of the current CMU-DA programs. Current research in the CMU-DA project is defining new design representations and synthesis strategies for which much better results are anticipated; some of these design representations will be discussed later in this report.

Something that the current CMU-DA software does provide is a collection of design representations at several levels of abstraction. The CMU-DA system is unique in that it provides high-level design representations starting at the behavioral level to describe a design implementation. These design representations provide a good testbed on which to define and implement a multi-level design representation that links together these high level representations. The following sections describe the definition of such a representation and its implementation.

3. Defining a Multilevel Representation in the CMU-DA System

The design representations developed by the CMU-DA synthesis software provide a set of representations at several levels of abstraction. The existence of these representations and programs to create and manipulate designs with them provides the opportunity to define and implement a multilevel representation that ties them together. These representations are not ideal for establishing a multilevel representation, but developing such a representation allows us to develop a testbed multilevel representation. In addition, in defining such a multilevel representation we can examine the information in the current CMU-DA design representations and make some suggestions for improvements for future design representations.

3.1 Relating Design Representations in the CMU-DA System

To define a multilevel representation, we must examine each level of representation in the CMU-DA hierarchy, find common features in each representation, and define a way to relate them together. We can relate two kinds of features in design representations: structural features and behavioral features.

Structural features in a design description are the components of the description that reflect the structure of the design. Structural features in the current CMU-DA behavioral representation are minimal; they consist solely of ISPS storage element variables that represent registers and memories in the completed design. All connections between the register and memory variables are defined implicitly by transfer operations in the procedural part of the ISPS description. This is appropriate, for a behavioral description is not intended to specify structure. Structural features in functional structure and logical/physical structure descriptions are much more common because these descriptions reflect the actual structure of the design. These structural features consist of registers, memories, operators, and the interconnections that tie them together.

Behavioral features are features of a design description which we use to model the behavior of a design. At each level of representation, behavior can be modelled by the current state of the description and operations that change the current state to a new state. Behavioral features are operations in the description that we can identify that change the state of the design and so describe its behavior. In an ISPS description behavior is modelled directly by the ISPS procedures in the description operating on the storage element variables of the description. The state of an ISPS

description is determined by state of the procedural description and the values of the variables. Operations that change the state of the design are simply the operations in the ISPS description. In the functional structure representation, we model behavior by the behavior of abstract components such as registers and operators plus the behavior of an unimplemented control part that is specified by operations in the microsequence table. The state of the functional structure description is specified by the state of the abstract components and the state of the microsequence table. Operations we can identify in the functional structure description that change the state of the design are the operations in the microsequence table and the operations that describe the behavior of the abstract components of the design. In the logical/physical structure representation, we model behavior of the design by the behavior of the specific components in the control and data part together and the state of these components. Operations in the logical/physical structure description that change the state of the design are the operations that describe the behavior of these speci.ic components. To relate behavioral features between designs, we must find operations in the different representations that cause the design to enter corresponding states at each level of abstraction and relate them together.

3.1.1 Relating Structural Features Between Design Representations

One approach to defining a multilevel representation is to relate only structural features between levels of representations. In this approach, the primitive components at each level of representation are decomposed into groups of more primitive components that are connected together until at the bottom level the most primitive components correspond to modules in the desired technology. In this way, design representations are related together by a nested hierarchy of components. This is the approach used by SABLE [Hill 79], a system for multilevel design specification and simulation developed at Stanford University. SABLE specifies a design in terms of *components.* Each component has associated with it an algorithmic behavioral description and a structural description. The structural description describes the component in terms of interconnected lower level components. In SABLE a design is described as a nested hierarchy of components. At the top level, a design is described as a single component. This single component is partitioned into a collection of lower level components, each of which is also partitioned into still lower level components. Since interconnection and behavior are specified for components at all levels of representation, SABLE provides an excellent environment for multilevel simulation. Each component of a design description may be simulated using its behavioral description or by simulating the components specified by its structural description.

SABLE has several attractive features. Decomposing designs structurally means that the design

representation is very general. The levels of representation used in a SABLE description can be defined by the user. A well-defined component nesting and connection methodology makes designs well structured and facilitates multilevel simulation. However, this approach is not appropriate for use with the CMU-DA design representations. The CMU-DA design representations are much less general than those supported by SABLE, and they do not necessarily decompose hierarchically by structure. While some high level structural decomposition is certainly appropriate in a design automation synthesis system, the current CMU-DA synthesis programs avoid the issue and do not define designs in terms of nested components. In addition, components described in lower level CMU-DA design representations do not necessarily decompose hierarchically into lower level components. In particular, a component at the functional structure level of description may not map directly into components at the logical/physical structure level of description. For example, SYNNER may split a functional structure component into more than one logical/physical structure component. Conversely, it may merge two functional structure components into one logical/physical component. The existence of these transformations do not allow a nested hierarchy of components such as that used by SABLE.

Finally, the CMU-DA system relies heavily on the partitioning of a design into a data part and a microprogrammed control part that implements an algorithm very similar to the initial ISPS description. While this discards much of the generality provided by SABLE, it makes relating behavioral features between design representations very attractive. If a multilevel representation is established using only structural information it is not possible to take advantage of this similarity.

3.1.2 Relating Behavioral and Structural Features Between Design Representations

Another approach to defining a multilevel representation that is more appropriate for the CMU-DA design representations is to relate both structural and behavioral features. In this method no attempt is made to specify higher level descriptions *completely* in terms of lower level descriptions, as in the SABLE hierarchy. Rather, this method establishes *correspondences* between the description's structural and behavioral features. This approach assumes a separation between the data part and the control part of a design, so that behavioral as well as structural features can be extracted from the design representations.

In this approach structural features are related by identifying and relating components in each level of representation that have corresponding components or implementations of components in other levels. For example, in the CMU-DA system storage elements defined in the ISPS behavioral representation have corresponding register and memory implementations in the lower level functional

structure and logical/physical structure representations. Likewise, data operators in the functional structure representation have operator implementations in the lower level logical/physical structure representation.

Behavioral features are related by identifying and relating together corresponding control operations at each level of representation that define changes in design state that can be related between levels. In a behavioral representation like an ISPS description these operations appear as high level operations like those found in a programming language. At lower levels these operations are tied more and more specifically to the design description until at the lowest level they can be represented as microcode operations or just as states in the control part of the design.

A technique similar to this approach has been used to formally verify that microcode implements a behavior specified by a higher level description [Darringer 79]. In this work a *simulation relation* specifies corresponding points in behavioral and microcode descriptions and the correspondence between description states that should exist at these points. This simulation relation is then used to verify the equivalence of the two descriptions by establishing that symbolically simulating the behavioral description and microcode between two of these points results in the specified corresponding states.

While it may initally seem difficult to relate together behavioral features between design representations in the CMU-DA system, examination of the representations and the relationships between them show that it is relatively straightforward. Due to the style of implementation of designs generated by the CMU-DA system, the ISPS behavioral algorithm and lower level control algorithms correspond closely. This is due to the close mapping of the initial behavioral specification into similar control algorithms at lower levels. In particular, we can relate control and data operations in the ISPS behavioral algorithm to their implementations as operations in the functional structure microsequence table and micro-operations in the logical/physical structure microcode. While there is not an exact correspondence between these control algorithms, the transformations that cause them to be different are known and can be compensated for.

3.2 A Multilevel Representation for CMU-DA

We now define specific relations between representations as *links.* Reflecting the two kinds of information related between representations, we define two kinds of links: structural links and behavioral links. Structural links relate structural features common to different levels of representation, while behavioral links relate behavioral features common to different levels of

representation. These links are built between adjacent levels of design representations. With these links the design representations form a single multilevel representation. Figure 3-1 shows this multilevel representation as a hierarchy of design representations and a collection of structural and behavioral links that relate them together.

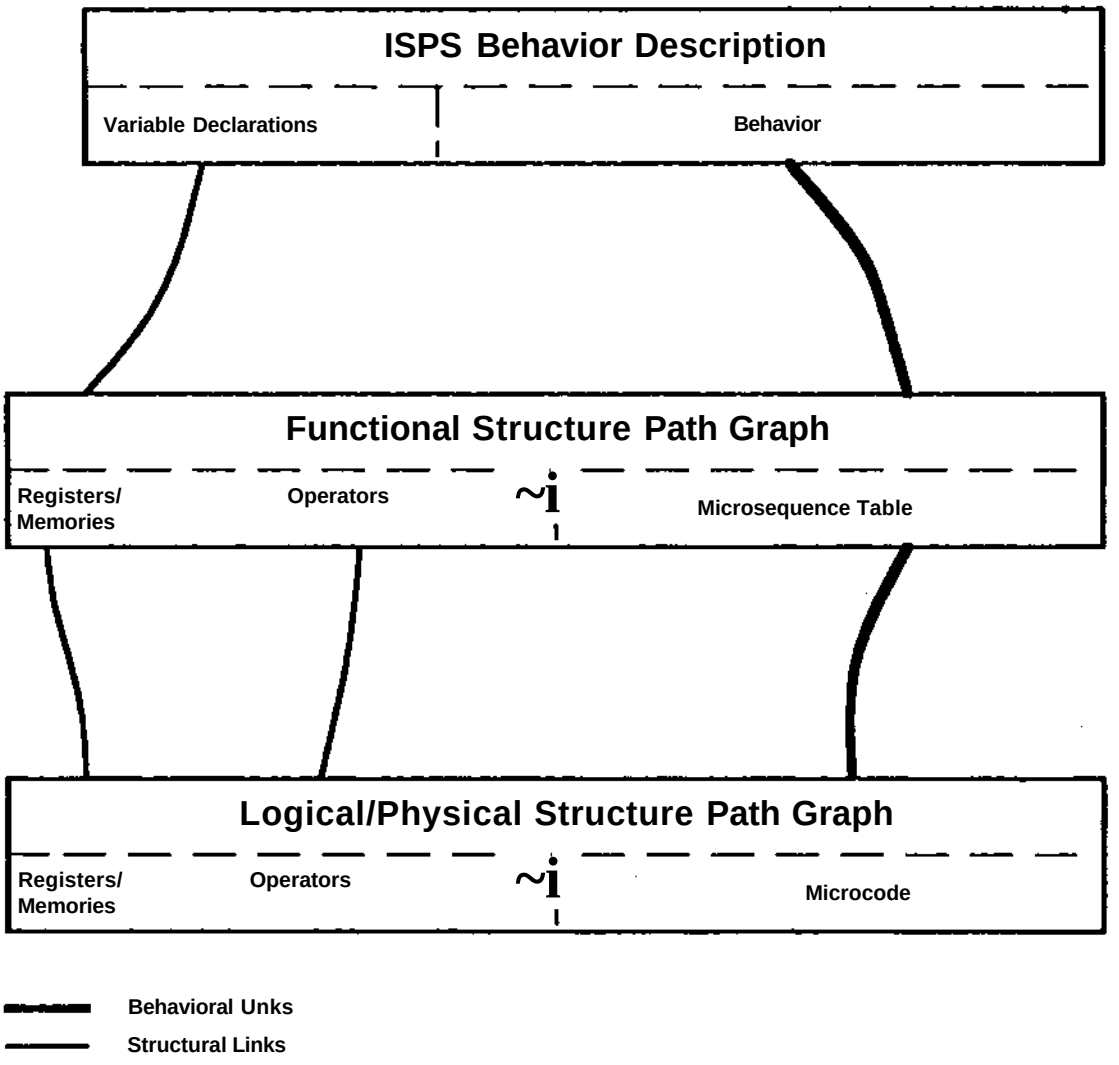

Figure 3-1: The CMU-DA Multilevel Representation

3.2.1 Structural Links

Structural links relate structural features between each level of representation. Between the behavioral and functional structure descriptions, structural links relate ISPS storage element variables to the corresponding functional structure register and memory path graph nodes that implement them. In the present CMU-DA synthesis programs, there is a one-to-one correspondence between ISPS register and memory storage elements and register and memory nodes in the functional structure path graph. Structural links between the behavioral and functional structure levels therefore link single ISPS storage element variables to single functional structure registers and memories.

Between the functional structure and logical/physical structure design representations, structural links relate together several kinds of components, including registers, memories, and operators. Since components in the logical/physical structure representation are implementations of components in the functional structure representation, it seems easy to relate them together. However, to implement the functional structure components as logical/physical structure components, the LSMS task may transform them in several ways so that exact correspondences do not result. For example, functional structure components may be split to fit into physically realizable components in the best way possible. Conversely, multiple components may be merged into single components to reduce the count of components. For this reason, structural links between components in the functional structure and logical/physical structure descriptions must at times link together more than one component in either direction.

3.2.2 Behavioral Links

Behavioral links relate together behavioral features from each level of representation. Behavioral features of a design representation are generally operations that change the state of the design. In the ISPS these features are simply ISPS operations. In the functional structure representation, behavioral features are microsequence table operations. In the logical/physical structure, behavioral features are microcode operations. Behavioral links relate together the operations in each representation that describe corresponding changes in state at each level of description.

Behavioral links between the behavioral and functional structure descriptions relate ISPS data and control operations to corresponding operations in the functional structure microsequence table. These links are one to one, providing that optimizations have not changed the functional structure microsequence table.

However, the synthesis process performs a few optimizing transformations on the control algorithm initially described in the ISPS which have an impact on behavioral links. For example, many ISPS procedures are substituted in the microsequence table inline, so that not all ISPS procedures map into corresponding microsequence table and microcode subroutines. Procedure call operations in the ISPS for these procedure have no corresponding call operations in the microsequence table, and so no behavioral link will exist for these call operations. However, the body of the substituted procedure remains in the microsequence table. Therefore, behavioral links should relate the operations in the body of the ISPS procedure to the operations in the microsequence table at the site of the inline substitution.

In addition, ISPS describes several control constructs that imply transfer of control without an explicit operator. ISPS procedures, for example, do not use an explicit return operation. However, the lower level design representation do describe these implicit operations explicitly. It is therefore necessary to somehow link these implicit ISPS operations to the corresponding implicit operations that implement them. This introduces a new class of behavioral links between behavioral and functional structure descriptions, called *implicit links.* Implicit links relate ISPS control constructs that imply operations in their implementation to the explicit microsequence operations that implement them.

Behavioral links between the functional structure and logical/physical structure descriptions link functional structure microsequence operations to the logical/physical structure microcode micro-operations that implement them. These links make no assumption about micro-operation ordering. Since more than one micro-operation may be generated to implement a single microsequence operation, a single microsequence operation can be linked to more than one micro-operation.

Figure 3-2 shows behavioral linking between the behavioral and functional structure levels for a statement from the ISPS fragment of Figure 2-2. The ISPS is shown on the left, while the corresponding microsequence table operations are on the right. Explicit links between the two levels are shown as solid lines, while implicit links are shown as dashed lines.

The major operation in this example is a DECODE conditional execution operation. This operation is implemented in the microsequence table as a SELECT operation, which specifies a conditional branch to one of two microsequence labels L # 11 and L # 13. The microsequence operation referred to by label L#11 implements the first conditional action specified by the ISPS DECODE statement, while the one referred to by L# 13 implements the second. In this example both of these operations are concatenate operations. The microsequence statement immediately following the statement

referred to by label L#11, the SJOIN operation, implements a branch to the end of the conditional execution action at the microsequence statement labelled L# 14.

The ISPS DECODE operation is linked with both explicit and implicit links. The explicit operation generated from the DECODE operation is the microsequence SELECT operation, so an explicit link is established between the two operations. The SJOIN following the implementation of the first branch of the SELECT operation is generated implicitly by the implementation of the DECODE, so an implicit link is established between them. The transfer (=) and concatenate (@) operations in the ISPS link to the respective microsequence concatenate operations (CONC) that implement them. Explicit links are established between these operations.

Figure 3-3 st ows an example of a microsequence operation and the corresponding microcode that is related to it by behavioral links. This example is also taken from the PDP-8 design and shows a call to the effective address procedure and its implementation as micro-operations on the control part of the design. Each micro-operation is represented by the microcode rom address and a list of the device primitives (e.g. LOAD, SELECT, etc.) that operate on components in the path graph, identified here by numeric labels. In this example, a procedure call is implemented by two successive micro-operations, so the behavioral link ties both to the CALL microsequence operation.

In this section we examined methods of establishing a multilevel representation and defined a particular approach to establishing a multilevel representation that is appropriate to the CMU-DA system. In particular, the multilevel representation establishes *links* between common features in both the structural and behavioral parts of each design representation. The next section describes a program that implements a simplified version of this multilevel representation.

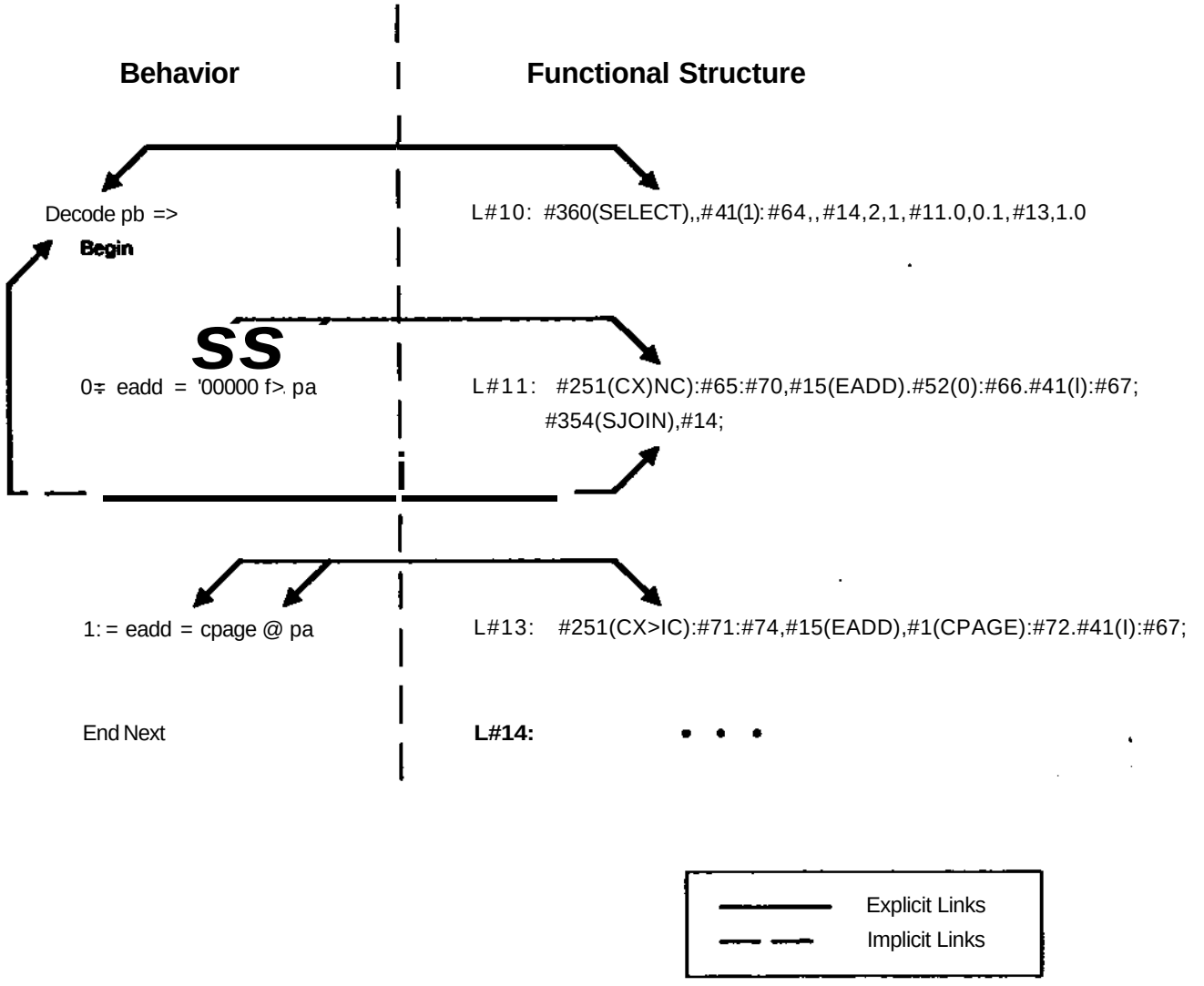

Figure 3-2: Behavioral Links: ISPS Behavioral - Functional Structure Microsequence Table

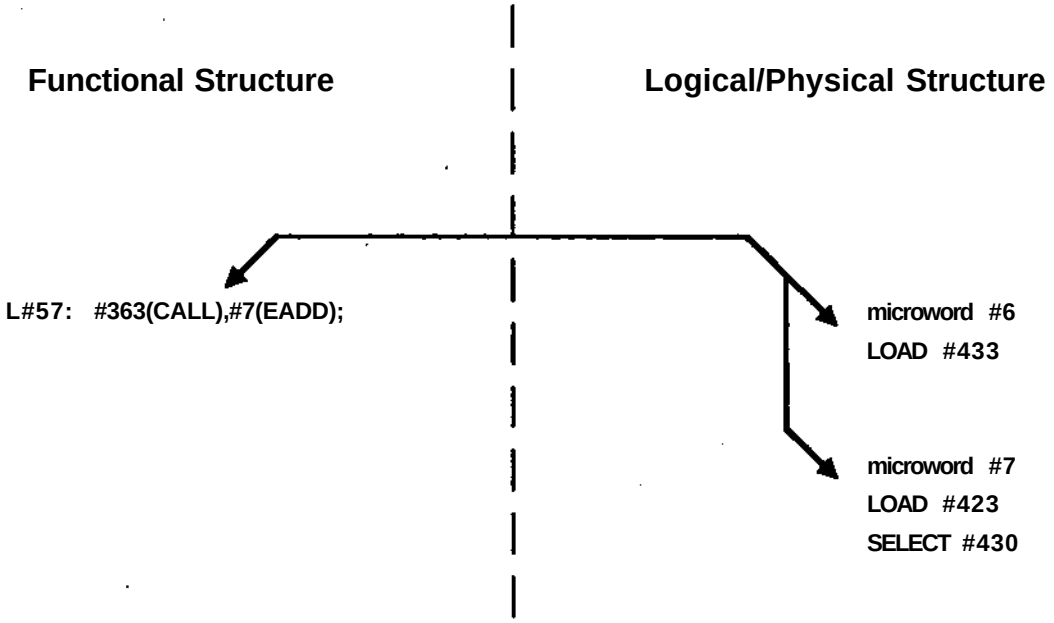

Figure 3-3: Behavioral Links: Functional Structure Microsequence Table • Logical/Physical Structure Microcode

4. Implementing a Multilevel Representation in the CMU-DA System

This section describes the software implementation of a multilevel representation for the CMU-DA design representations. A program has been developed that reads the CMU-DA design representations and builds a multilevel representation that ties them together as a large data structure.

4.1 The Multilevel Representation Simplified

A major problem in implementing a multilevel representation is the size of the representation. For more than a trivial design the design representations generated by the CMU-DA system are quite large. Building data structures to support these representations completely and linking them together is at this point impractical. While work is presently underway to define a database for building and manipulating large multilevel design representations [Meyer 81], the only reasonable short term solution was to remove some of the information from the multilevel representation.

The major simplification to the multilevel representation is the removal of most of the functional structure representation. The functional structure path graph is removed altogether. The functional structure microsequence table is replaced by the microsequence table output of the control allocator, which references components at the logical/physical structure level. Structural links are established directly between storage elements described in the ISPS description and register and memory components described in the logical/physical structure description.

Behavioral links are simplified also. Specifically, only transfer operations that write into ISPS-defined storage elements and control operations are linked between the ISPS description and the microsequence table. Behavioral links between the microsequence table and microcode are maintained as they were previously defined. Figure 4-1 shows the simplified multilevel representation and the links between the levels of representation.

4.2 Data Structures for Multilevel Representation

To build a multilevel representation each level of representation and the links between them must be defined and built as data structures. Design representations and links between them have been defined as PASCAL data structures. Many of the ideas for design representation data structures are drawn from the different CMU-DA synthesis program data structures, notably SYNNER [Leive

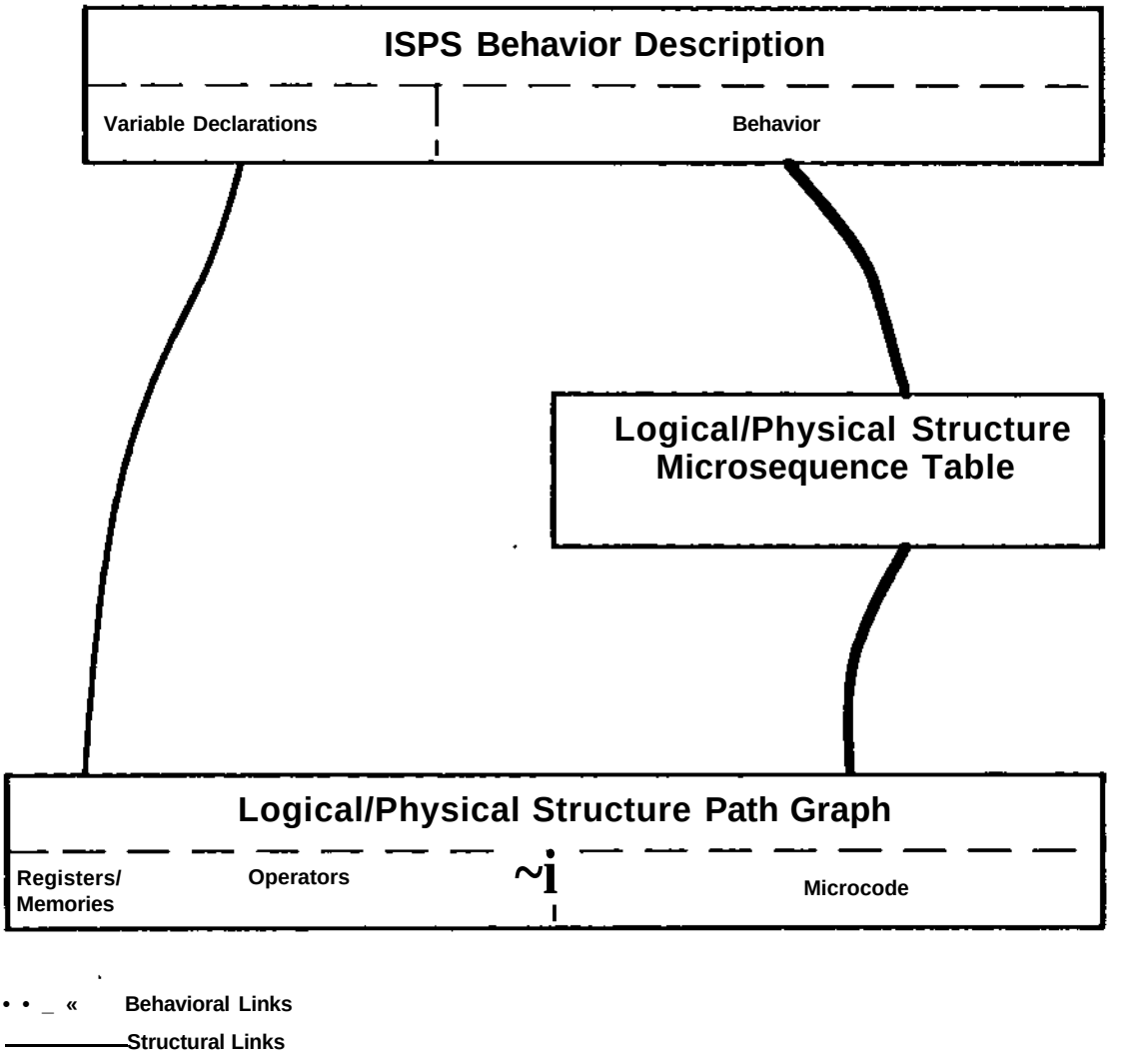

Figure 4-1: The Simplified CMU-DA Multilevel Representation

81, Leive 80] and the control allocator [Cloutier 80]. Ideas for the GDB data structures are drawn from the GDB-RTM translator used with the ISPS simulator and a PASCAL implementation of an ISPS Parser written by Mario Barbacci.

The ISPS behavioral description is represented by a parse tree built from the ISPS parser output, the Global Data Base (GDB) tree. The GDB tree is output by the ISPS parser in an ASCII file. Its format describes a parse tree in a list notation very similar to that used by LISP. GDB tree nodes are described by single records. Node brothers are maintained in a linked list to allow multiple sons for any node. Each tree node points to its first and last son and its immediate brother.

The logical/physical structure path graph is represented by a large doubly-linked list of records representing path graph nodes, where each path graph node represents a component in the design. Interconnection for the path graph node is represented by fields that point to linked lists of records that point to interconnecting path graph nodes. Other information maintained in the record includes node type, operation, physical component used, etc.

The microsequence table is also represented by a doubly-linked list of records, each of which represents a microsequence operation. Fields in these records contain operation code, lists of source and destination path graph nodes, references to other microsequence table operations for control operations, and other information.

Records representing microcode micro-operations are organized by microcode instruction. Each microcode instruction corresponds to a microcode rom location and is represented by a linked list maintained in order of rom address. Since more than one micro-operation may occupy the same instruction, micro-operation records for each address are maintained in a linked list attached to each microcode instruction record. Each microcode operation record maintains a list of records that represent the device primitives that implement the micro-operation.

Behavioral links between microsequence operation records and micro-operations are implemented by fields in both types of records. A field in each microsequence operation record points to the first micro-operation corresponding to that microsequence statement. Micro-operation records for any subsequent micro-operations that implement the microsequence operation are maintained in a linked list by a field in each micro-operation record. Another field in each micro-operation record points back to the microsequence record to link each micro-operation back to its origin.

Behavioral and structural links between the behavioral description and the path graph and microsequence table are maintained as records in separate linked lists in order to allow searching through the multilevel representation by the established links.

Structural link records link GDB tree records for declarations of ISPS entities[2] to the path graph nodes that implement these storage elements. Since more than one path graph node may implement a storage element defined in an ISPS description, path graph nodes that implement a GDB entity are maintained as a linked list of records that are attached to each structural link record.

---

[2] ISPS entities represent storage elements, procedure names, and sometimes both a procedure name and a storage element.

Behavioral link records link GDB node records of ISPS transfer and control operations to corresponding microsequence operation records, both implicit and explicit. There is one behavioral link record for each GDB operation that is linked. Since a GDB operation may conceivably be linked both explicitly and implicitly, these records can support both links at once.

4.3 BLINK: A Program to Implement the Multilevel Representation

A program to read design representation files and build the multilevel representation data structures has been implemented and is called Blink.[3] Blink reads the design files generated by the CMU-DA synthesis programs and builds data structures to represent the design representations at the behavioral and logic al/physical structure levels. It then establishes the behavioral and structural links between them to form a multilevel representation. It should be noted that this is a rather odd way to establish a multilevel representation when the design representations are generated by an automatic synthesis system. Normally in such a system the multilevel representation would be generated and maintained as each design is generated, rather than building it after the fact. However, for the most part the present CMU-DA synthesis programs do not retain this information in the design representations they build and there was no clean way to modify them to maintain a multilevel representation. For this reason, most of the behavioral and structural links are established after they are generated by the synthesis programs. It is expected that the methodology for establishing a multilevel representation developed in this project will be applied in future CMU-DA software so that the multilevel representation will be built up as a design is developed.

The one exception in retaining link information is the control allocator. The control allocator maintains a data structure that links microsequence operations to corresponding microcode operations and has been modified to write this data structure into a separate output file. Blink reads this file after it reads the GDB and path graph files and builds the microsequence table and microcode data structures and the behavioral links between them.

To establish the other structural and behavioral links, Blink traverses the GDB tree, path graph, and microsequence data structures in two passes. The first pass traverses the GDB tree and builds structural link records for each declared ISPS entity. When a reference to an entity is encountered, it is tied to the appropriate structural link, so that the list of structural links doubles as a symbol table for

[3] An acronym for Behavioral Linking

ISPS entities. For each ISPS storage element entity encountered, the path graph is searched for corresponding register and memory path graph nodes. If found, they are tied to the structural link record, completing the structural link. To aid in the following behavioral linking task, structural link records for ISPS procedure entities that have corresponding procedures implemented in the microsequence table also point to the entry points of the corresponding microsequence procedure.

The second pass over the GDB tree traverses each ISPS procedure that is implemented in the microsequence table and microcode. Concurrent with each ISPS procedure traversal is a traversal of the corresponding microsequence table procedure. The traversal of each data structure starts at the entry point of a procedure. The GDB subtree is recursively traversed in execution order starting at this entry point. Each time the GDB traversal encounters a transfer or control operation node, Blink attempts to find a corresponding operation in the microsequence table. This is done by moving the position of the microsequence table traversal forward until an operation is found that corresponds to a transfer or control operation in the ISPS. If the current operations in the GDB and microsequence table match, then Blink establishes a behavioral link between the corresponding records and continues the traversal. Traversing conditional operations requires special handling, as does traversing procedure calls that have been replaced by inline substitution. In conditional operations, each branch of the conditional must be traversed and linked, with normal traversal resuming at the end of the conditional. When a procedure is expanded inline, Blink must traverse the procedure body and link the corresponding microsequence operations that appear inline.

Blink provides a command interpreter to interface with the program user. The command interpreter allows the user to read design representation files, invoke behavioral linking and applications routines, and interactively display the design representations and the relations between them.

In this section we have discussed the implementation of a simplified version of the multilevel representation that was described in Section 3. Blink, a PASCAL program, reads design representations from the ISPS parser and the output of the CMU-DA programs and builds a multilevel representation of the design and it's behavioral description using dynamic data structures. An application routine that uses this multiple representation to measure information in the completed design and relate it back to the behavioral description is described in the next section.

5. Timing Abstraction for Behavioral Simulation: An Application of the Multilevel Representation

The multilevel design representation developed by Blink has several potential applications. An interesting set of applications involves measuring the performance of a completed design and relating it back to the original behavioral specification. For example, the multilevel representation can be used to extract timing information from a completed design's microcode and relate it back to the original control algorithm specified in the behavioral description. *Timing Abstraction* measures and extracts timing information from the completed design microcode and adds it to a corresponding behavioral description to allow verification and simulation of features of a low-level design description using the behavioral representation and some added information. This information can be added in a way such that the extracted timing information can be used to simulate timing at the behavioral level. The advantage of this approach is the use of an inexpensive high-level simulator to predict the performance of a lower level design. The timing abstraction task has been implemented using the multilevel representation implemented for this project. It generates as output an ISPS GDB file suitable for use with the ISPS simulator but with the low level timing information added. The ISPS simulator has been modified to utilize this added information during simulation.

5.1 Instrumenting an ISPS Simulation with Timing Information

5.1.1 ISPS Simulation and Timing

The ISPS Simulator [Barbacci 80] performs high-level behavioral simulations of ISPS descriptions. The ISPS simulator emulates an imaginary three-address computer called the *Register Transfer Machine* (RTM) [Barbacci 79]. It is driven by *RTM tables* which consist of a table of RTM operations and a symbol table of ISPS storage element and procedure variables. RTM tables are compiled from an ISPS description by a GDB to RTM translator called GDBRTM. The Register Transfer Machine simulates ISPS parallelism by dynamically maintaining multiple streams of RTM operations for parallel sections of an ISPS description and switching execution between them.[4]

The ISPS simulator maintains timing information in two independent units of time, one maintained by the simulator, the other maintained under user control in the ISPS description. The time count of *steps* corresponds to RTM operations executed and is incremented each time an RTM instruction is

[4]However, since any parallelism defined in an ISPS description is discarded by the present CMU-DA software, this feature of the simulator is not used in this protect. It would not be difficult to add parallelism to the multilevel representation if necessary.

executed. The time count of *intervals* is controlled in the ISPS description. The interval time count is controlled using the predefined ISPS qualifier PTIME and predefined ISPS procedures such as DELAY, WAIT, and TIME.WAIT.

The PTIME qualifier is attached to ISPS procedures and specifies a time of execution for a procedure. Each time the ISPS simulator executes this procedure the interval time counter is increased by the time specified in the PTIME qualifier. The PTIME qualifier is intended to give a rough estimate of timing at the procedure level. Because it relates timing information to the body of the procedure and not the internal operations, the PTIME qualifier is a rather inexact way to add timing information to a simulation.

The DELAY procedure causes a simulation delay specified by its argument. This gives the ISPS writer explicit control over simulator interval timing. The WAIT and TIME.WAIT predefined procedures are intended to implement delays that wait for conditions.

5.1.2 Adding Timing Information to ISPS Simulations

The multilevel representation relates the ISPS behavioral representation to the completed logical/physical representation by behavioral links. To extract timing information from the logical/physical structure representation we must define how to measure this information and add it to a behavioral description. We add timing information at points which affect the externally visible state of the ISPS simulation. Specifically, these points are changes in the value of register and memory values caused by ISPS transfer operations and changes in the flow of control caused by ISPS control operations.

We break down an ISPS description into straight-line sequences ending with these operations. These sequences are called *timing blocks.* The notion of timing blocks is similar to that of basic blocks used in compiler optimization [Aho 77]. Basic blocks define straight line segments of code that can be entered only at the beginning of the block and left only at the end. For this reason, basic blocks end with operations that change flow of control or immediately precede an operation that is the target of a branch. Timing blocks differ from this definition in that timing blocks can end with transfer operations as well as control operations. To fit timing blocks into a high level procedural language like ISPS, the implicit transfer of control operations associated with high level control constructs must be considered and represented somehow. For example, consider the end of an ISPS procedure. Although there is no explicit return operation, the return must be implemented in the lower level representations. This is analogous to the problem of linking implicit operations with

behavioral links described in section 3. In fact, the timing abstraction algorithm uses behavioral links to identify timing blocks in the ISPS.

The microcode that implements the ISPS description can be broken down into corresponding timing blocks. Since these timing blocks are straight-line segments of microcode, the execution time of each timing block is proportional to its length in microcontroller steps. This time value can be measured and related back to corresponding timing blocks in the ISPS.

Timing blocks break an ISPS description down into segments that separate transfer operations and changes in flow of control. Since they are straight line segments of code, the execution time of an arbitrary timing block is constant provided that it is implemented in the logical/physical structure microcode as a straight line segment.[5] Since we can add execution tim ≥s to an ISPS description independent of flow of control using timing blocks, we can simulate timing in an ISPS description at the behavioral level completely and accurately. However, we must define a method to add timing information to the ISPS behavioral representation.

Although we have described several methods to control timing in the ISPS simulator, they are not really suited for adding timing information in timing block form. The PTIME qualifier can only be attached to procedures. Calls to the DELAY predefined procedure could be added at the ends of timing blocks, but would require substantial editing of the GDB tree to insert. Instead, the timing information is added to the GDB tree as user-defined information.

The ISPS and GDB syntax allow the user to add application dependent information to an ISPS description using two different mechanisms: ISPS *qualifiers* and GDB *attributes.* ISPS qualifiers are a way of adding user defined information to a source ISPS description. Qualifiers may be attached to entity declarations and certain operations in an ISPS description. Unfortunately, qualifiers cannot be attached to all ISPS constructs. In particular, information that is implicit in the ISPS description but which may be referenced in parsed form in the GDB is not available to qualifiers. Like predefined procedures, adding qualifiers to an existing GDB tree can require substantial editing.

Attributes add information to arbitrary nodes of the GDB tree but have no corresponding appearance in the source ISPS description. Attributes are maintained as separate data structures which are attached to nodes of the GDB tree, so that no editing of the actual GDB tree is necessary to

[5]While this is generally, the case, there are a few exceptions. For example, consider the multiply (*) operation. In a shift-and-add multiply implementation, the length of time required to execute this operation will be dependent on the values of the operands.

add attributes. Due to their generality and ease of addition, attributes were chosen to add timing information to behavioral descriptions.

Timing Attributes add the timing information to the GDB tree at the end operations of timing blocks. *Explicit Timing Attributes* add timing information for timing blocks whose end operations appear explicitly in the ISPS description. *Implicit Timing Attributes* add timing information for timing blocks whose end operations are described implicitly, as described previously. Timing attributes are added to the GDB tree in standard format.

To illustrate timing attributes, figure 5-1 shows a small fragment of the PDP-8 ISPS description and a corresponding GDB tree fragment with timing attributes added. Explicit timing attributes appear in the GDB in the form !7!<time-value>!. This attribute implies that in simulation <time-valu';> should be added to the simulation clock when the operation the timing attribute is attached to executes. In figure 5-1 there are 3 explicit timing attributes, attached to the DECODE operator and the two assignment operators. Each of these timing attributes has a time value of 1, so that the simulation time is incremented by one when each of these operations is executed. Implicit timing attributes appear in the GDB in the form !8!<time-valu$>!. This attribute implies that in simulation <time-value> should be added to the simulation clock when the operation the implicit timing attribute refers to is executed. In figure 5-1, only one implicit timing attribute appears. It is attached to the first alternative of the DECODE statement and implies that there is a branch operation at the end of this alternative (following the first assignment operation) that takes one time unit to execute, so that the simulation clock should be incremented by 1 when this branch operation occurs.

5.1.3 Accuracy Considerations for Timing Abstraction

Timing Abstraction extracts the timing information from the microcode of a design and adds it to a corresponding behavioral description. This timing information is proportional to the number of cycles the microcode controller has executed between operations that relate to the behavioral description. However, since microcode is almost always optimized, there are cases where some accuracy is lost in timing abstraction.

Microcode optimizations change the order of operations in the microcode, so that the order of operations in the behavioral representation may not always reflect the true ordering of operations in the actual design. In the current timing abstraction algorithm exact timing information is lost for some of the reordered operations. The effect of operation reordering is that timing blocks for these operations are combined. Simulation timing remains accurate at the ends of these timing blocks, but the blocks are larger, making the resolution of timing in the simulation more grainy.

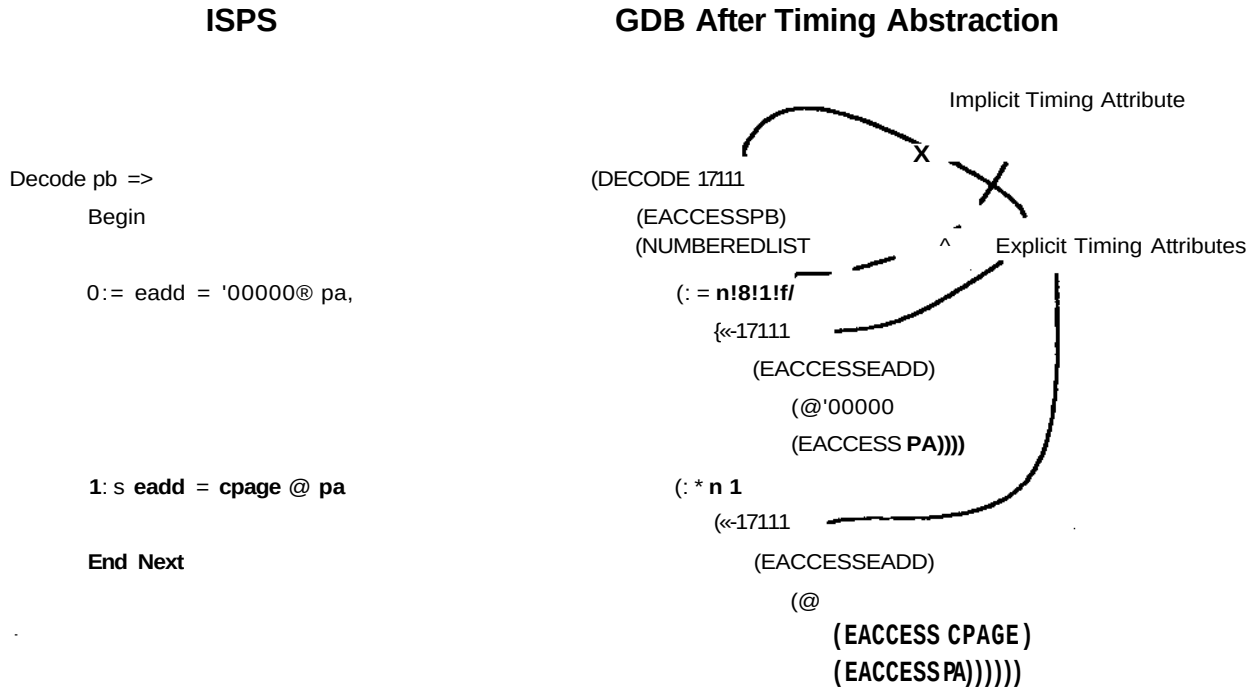

Figure 5-1: An Example of GDB Timing Attributes

Microcode optimizations are constrained to be within straight line segments of microcode [Nagle 80] and do not include control operations. For this reason, the effects of these optimizations on timing abstraction are local and the resulting loss of accuracy is usually small. Furthermore, timing between key operations such as control operations remains accurate, so that inaccuracies due to optimizations are not cumulative.

5.2 Implementing Timing Abstraction

Timing information is extracted from timing blocks in the completed design's microcode. Since the cycle time of the microcontroller is known and fixed, the execution time of these timing blocks can be extrapolated as the microcontroller cycle time multiplied by the number of words of microcode executed in the sequence. The approach used in determining execution times for timing abstraction is to walk through the microcode for each timing block and determine the length of the microcode sequence in words. This value is then associated with the corresponding timing block at the behavioral level using timing attributes.

The timing abstraction task has been implemented as an application routine in the Blink multilevel representation program. The basic form of the timing abstraction routine is a walk down the

microsequence table linked-list that identifies timing blocks in the list. It is assumed that the microcode occurs in ascending order corresponding to the order of the microsequence list, so that an execution order traversal is not necessary for conditional operations as it is in the behavioral linking algorithm. Each time a timing block is encountered, the length of the timing block in microwords is calculated. This value is proportional to the execution time of the timing block. If an optimized micro-operation has not preceded the timing block, then this difference value is added to the GDB tree in the form of a timing attribute using the behavioral link at the end of the timing block. If the behavioral link is an explicit link, then an explicit timing attribute is used. If it is an implicit link, an implicit timing attribute is used instead. When this walk of the microsequence table is complete, the GDB tree is output with added attributes in the standard GDB file format.

5.3 Modifications to the ISPS Simulator to Support Timing Abstraction

To support simulation with the added timing attributes, it was necessary to modify the ISPS simulator and support software. In particular, the GDB-RTM translator was modified to recognize timing attributes and add the timing information they contained to the RTM tables, and the ISPS simulator was modified to use this added timing information to update its simulation clocks. These modifications were implemented by Mario Barbacci.

Adding time information to the RTM tables required a modification of the RTM statement table definition. The RTM statement table can be viewed as a fixed array of records, with one record for each RTM operation. Timing information from the PTIME qualifier was previously conveyed to the simulator through a field unused for procedure beginning operations but used in other applications. Since timing information from timing attributes could conceivably be added to any arbitrary RTM operation, they required the addition of another field, called the ATIME field.

The GDB-RTM translator was modified to search for timing attributes in the appropriate places and place the values associated with them into the ATIME field of the appropriate RTM operations. The GDB-RTM translator generates RTM operations by recursively traversing the GDB tree and calling several code generation routines. Modifying it to find timing attributes required adding code to these routines to search for timing attributes on the appropriate tree nodes and add time values to the ATIME fields of corresponding RTM operations. The ISPS Simulator was then modified to update simulation interval clocks when RTM operations with a nonzero ATIME field are encountered.

This section defined timing abstraction, a multilevel representation application, and its

implementation as a routine in Blink, the multilevel representation program. Timing abstraction extracts timing information from the completed design microcode and associates it with portions of the corresponding ISPS behavioral description. Modifications to the ISPS simulator and associated information allow simulation using this timing information in the ISPS simulator. The following section describes some results simulation using timing abstraction.

6. Some Simulation Results for Timing Abstraction

This section presents some examples of ISPS simulation using timing abstraction. The example chosen for simulation here is a CMU-DA generated design of a DEC PDP-8 minicomputer implemented with TTL integrated circuits. This design is presently being fabricated in another project [DiRusso 81], which makes it attractive for our use because results of simulation with timing abstraction could be compared with timing of the actual hardware. However, at this writing the fabrication project is incomplete, so comparisons between simulated and actual timing have been postponed.

The examples of ISPS simulations shown in the following figures are transcripts from the ISPS simulator. These transcripts show simulation traces and the commands that set up and started the traces. Lines in the simulation trace show three kinds of information:

1. Trace identification - Whether a line in the trace is a breakpoint or a trace point. Breakpoints print a line in the simulation trace and halt simulation. They are set in these simulations to show procedure start and finish (the commands that restart the simulation at these breakpoints have been edited out of the figures to make them more readable.) Trace points are printed when a value is written into an ISPS variable.

2. Simulation Time - The current simulation time for each line in simulator timing intervals, which correspond to microcontroller clock cycles.

3. Action - The event that causes the line to be printed. These events include the start and finish of procedures and transfer operations that write a value into an ISPS variable flagged for tracing.

All numbers in the simulation transcripts are octal. These simulation transcripts have been edited to make them more readable. In particular, ISPS commands that do not relate directly to the simulation have been omitted. In addition, ISPS step times have been edited out of the simulation traces. Unedited transcripts of these same simulations are shown in the appendix.

The simulation traces presented in this section show two different simulations using the PDP-8. The first simulation shows exhaustive simulation of a small portion of the PDP-8 design that calculates the effective address used by memory reference instructions. This task is described in the PDP-8 ISPS description as a single procedure that was shown previously in Figure 2-2. The second simulation shows a simulation of the entire PDP-8 design executing a small program.

6.1 PDP-8 Effective Address Calculation

The simulation traces shown in Figures 6*1 -6-6 document the different simulation traces encountered in exhaustive simulation of the different cases of PDP-8 effective address calculation. Effective address calculation in the PDP-8 translates the 9 bits of the PDP-8 instruction register allotted for memory reference in PDP-8 instructions into a 12 bit address.

Memory in the PDP-8 is divided into pages of 128 words. Effective address calculation allows direct reference of memory locations in the bottom page, page 0, and the page that the program is currently executing, the current page, using a page select bit of the instruction register, pb. Words in each page are specified by the page address field of the instruction register, pa. If the indirect address bit of the instruction register, ib, is set, the contents of this memory location specified by pb and pa becomes the effective address; otherwise the effective address is the address of the location specified by pb and pa.

An additional feature of PDP-8 effective address calculation is autoindexing. Memory locations 8*15 are called autoindex registers. When these locations are referenced indirectly, they are incremented before the indirect address is calculated. A more detailed explanation of PDP-8 addressing modes can be found in [DEC 73].

The ISPS procedure eadd of Figure 2-2 implements this address calculation as specified for the PDP-8. Simulation traces in the effective address simulation show the simulation times that the eadd ISPS procedure starts and finishes, as well as the times that the register eadd and memory m are written into and the value that is written at these points. At completion, the eadd register should contain the result of the effective address calculation. In these simulations the eadd procedure is started directly by a command to the ISPS simulator after the instruction register i has been initialized to cause the desired addressing mode. Figures 6-1 - 6-3 show respectively simulation traces for calculation of direct, indirect, and autoindexed addresses using page 0 addressing. Figures 6-4 - 6-6 show the corresponding simulation traces for calculation of addresses using current page addressing. For each of these simulation case, the simulation timing is slightly different due to the different microcode executed in the completed design to compute the effective address.

```
>setvalue 1b$0
>setva1ue pb=0
>setvalue pa=45          I effective address should be #45
>start eadd

B: t « 0          BEGIN
T: t - 3         EADD=#0045
B: t « 6          END
```

Figure 6-1: Page 0 Direct Mode Effective Address Simulation

```
>setva1ue 1b=l
>setvalue pb«0
>setvalue pa=45
>setvalue m[45]-55       1 effective address should be #55
>start eadd

B: t - 0          BEGIN
T: t - 3         EADD==#0045
T: t - 8         EADD==#0055
B: t « 9          END
```

Figure 6-2: Page 0 Indirect Mode Effective Address Simulation

```
>setvalue ib=l
>setvalue pb=0
>setvalue pa=12          1 note that m[#12] 1s an autoindex register
>setvalue m[12]=66       I effective address should be #67
>start eadd

B: t * 0          BEGIN
T: t « 3         EADD=#0012
T: t * 10        M[#12]=#0067
T: t « 13        EADD=#0067
B: t * 14         END
```

Figure 6-3: Page 0 Autoindex Mode Effective Address Simulation

6.2 PDP-8 Multiply Program Example

The following simulation trace shows the simulation of the entire PDP-8 ISPS description as it executes a small program stored in its memory. Figure 6-7 shows the program used in the simulation, a small program adapted from an example in [DEC 73] that multiplies the numbers located in octal

```
>setvalue 1b-0
>setvalue pb-1
>setva1ue pa=20
>setvalue cpage-3        I effective address should be #620
>start eadd

B: t « 0          BEGIN
T: t - 3         EADD*#0620
B: t * 5          END
```

Figure 6-4: Current Page Direct Mode Effective Address Simulation

```
>setvalue 1bˢl
>setvalue pb*l
>setvalue paˢ20
>setva1ue cpage*3
>setvalue m[620]*700     (effective address should be #700
>start eadd

B: t • 0          BEGIN
T: t • 3         EADD=#0620
T: t « 7         EADD-00700
B: t « 8          END
```

Figure 6-5: Current Page Indirect Mode Effective Address Simulation

```
>setvalue ib=l
>setvalue pb*l
>setvalue paˢ12
>setvalue cpage=0
>setva1ue m[12]*333      I effective address should be #334
>start eadd

B: t « 0          BEGIN
T: t « 3         EADD=#0012
T: t « 10        M[#12]=#0334
T: t « 12        EADD=#0334
B: t = 13         END
```

Figure 6-6: Current Page Autoindex Mode Effective Address Simulation

memory locations 210 and 211 and stores the result in octal memory location 212. This program is interesting for simulation because it uses several of the more common PDP-8 instructions. It implements multiplication by adding one multiplicand to itself in a loop. A simulation trace for the execution of this program with multiplicand values 4 and 5 is shown in Figure 6-8. Simulator commands load 4 and 5 into the memory locations reserved by the program for multiplicands and start the PDP-8 simulation (Simulator commands loading the multiply program have been omitted).

The simulation trace follows write operations into the link-accumulator register LAC, its subset register AC, and the memory M. Each entry shows the simulation time in microcontroller steps and the value that is written into the register or memory location at that time. The result value is written into the product memory location at simulation time t = 381. Simulation halts after this value is written (the AC register is cleared simultaneously).

```
Addr    Contents          PDP-8 Assembly Code
*****************************************************

                •30                         ; set origin to 30
30      0213            mult                ; for Indirect subroutine call
                •200                        ; set origin to 200
200     7300    start.  cla ell             ; clear link & ac
201     1211            tad a               ; move a Into ac
202     3205            dca '. + 3          ; store It 1n temp
203     1212            tad b               ; move b Into ac
204     4430            jms 1 30            ; call mult subroutine
205     0000            0000                ; temp value
206     3210            dca prduct          ; move result to prduct
207     7402            hit                 ; all done
210     0000            prduct,             ; Where to store result
211     0000            a.                  ; Where to get 1st multiplicand
212     0000            b.                  ; Where to get 2nd multiplicand
213     0000    mult.   0000                ; data/return address pointer
214     7041            cia ( cma 1ac )     ; set ac to -b
215     3223            dca mtally          ; move it to mtally
216     1613            tad 1 mult          ; add temp (a) to ac
217     2223            1sz mtally          ; increment
220     5216            jmp . - 2           ; if mtally <> 0 loop back
221     2213            1sz mult            ; set return address to temp+1
222     6613            jmp 1 mult          ; return
223     0000            mtally,0000         ; loop counter
```

Figure 6-7: A PDP-8 Multiply Program

This section showed some results of simulating a PDP-8 using the ISPS simulator with timing information extracted from a design generated by the CMU-DA synthesis programs. Simulation traces showed simulation of the effective address calculation implemented by the PDP-8, as well as simulation of the entire PDP-8 executing a small program to do multiplication. The following section concludes this report with some conclusions about the multilevel representation and some ideas for future work.

```
>setvalue m[211]=4
>setvalue m[212]=5
>startniu1t        I defined to start the PDP-8 simulation with PO#200

T:  t • 8          AC=#0000
T:  t « 26         LAC=#00004
T:  t • 39         M[*205]=#0004
T:  t - 39         AC=#0000
T:  t - 54         LAC=#00005
T:  t - 71         M[#213]=#0205
T:  t « 78         AC=#7772
T:  t • 79         LAO#07773
T:  t « 94         M[#223]*#7773
T:  t - 94         AO#0000
T:  t « 112        LAC=#00004
T:  t * 128        M[#223]=#7774
T:  t • 162        LAC=#00010
T:  t « 178        M[#223]=#7776
T:  t - 212        LAC=#00014
T:  t « 228        M[#223]-#7776
T:  t = 262        LAC=#00020
T:  t - 278        M[#223]=#7777
T:  t = 312        LAC=#00024
T:  t « 328        M[#223]=#0000
T:  t « 349        M[#213]=#0206
T:  t « 381        M[#210]=#0024
T:  t « 381        AC=#0000
```

Figure 6-8: PDP-8 of Multiply Program Simulation

7. Conclusions

This report has described the definition, design, and implementation of a multilevel design representation for the CMU-DA design synthesis software and an associated application of the multilevel representation, timing abstraction. The basis of the multilevel representation is the establishment of relations between levels of representation called links. These links relate structural and behavioral features between adjacent levels of representation. Structural links relate components such as registers and memories to their implementations in lower level representations. Behavioral links relate operations at which changes in corresponding states can be identified between levels of representation. A software package has been implemented that reads design representations for a specific design and builds the multilevel representation as a large data structure. An application routine that extracts timing information from the bottom level of the multilevel representation and attaches it to its corresponding behavioral representation for simulation purposes has also been implemented. This added information is used by the ISPS simulator to provide high level behavioral simulation with timing at the level of the low level design.

7.1 Limitations of the Multilevel Representation

This project successfully demonstrated the implementation of a testbed multilevel representation for the present CMU-DA synthesis software. However, as presently implemented the multilevel representation has several shortcomings. Problems with the multilevel simulation stem from three major sources:

1. Limitations due to the multilevel representation definition.
2. Limitations due to the multilevel representation implementation.
3. Limitations inherited from the CMU-DA design representations.

7.1.1 Limitations in the Multilevel Representation Definition

The definition of the multilevel representation definition is limited in three ways. The first and probably most important limitation is the exclusion of lower levels of abstraction in the multilevel representation. The CMU-DA software used in this project defines design representations at the structural logic level and above. It does not define design representations for the gate, circuit, or physical layout levels of abstraction. Since defining these design representations is in itself a major project, establishing a multilevel representation definition that extends to these lower levels of abstraction has not been addressed.

The second major limitation of the multilevel representation definition is due to assumptions about the behavioral description and the CMU-DA software's implementation of that behavioral description. In particular, behavioral descriptions are assumed to describe a single process that is implemented as a data part driven by a synchronous microprogrammed or state machine controller. Behavioral links between the behavioral and lower level representations are dependent on the assumption that operations that cause changes in state at each level of representation can be related together. Relating these operations together was relatively simple in the multilevel representation implemented in this project because of the straightforward (at the expense of design performance) manner in which the CMU-DA synthesis software generates designs. However, relating operations between levels for a less naive design would not be as easy.

Several optimizations may take place in the design process which eliminate or modify parts of the design as it was described by the ISPS description. Data part-control part tradeoffs, "code motion"* in the control algorithm, and common subexpression elimination are just a few of the possible optimizations that may take place in design process. It is inevitable when so many optimizations are applied that some relations between the behavioral and lower level design specifications will be lost. However, it is expected that if these relations are maintained as the optimizations take place the degradation of links between levels of representation will be graceful, in that some key relations will be maintained independent of the optimizations and others may be able to recognize and account for them. Even though an optimized design may not be completely related between levels of representation, relations between key state-change operations such as control operations and operations involving externally visible variables will remain in some form, so that much useful information will remain in a multilevel representation for even a highly optimizing design system.

Finally, the current definition of the multilevel representation is limited in that it has no provision for describing multiple processes or parallelism in a design at any level other than the behavioral level. It will be necessary in future design systems to describe multiple processes in the behavioral description and their implementation in the design. The multilevel representation could be modified to support multiple processes by maintaining information that decomposes the behavioral representation into partitions that describe single processes. Parallelism is not described in the present definition due to an artifact of the CMU-DA synthesis software in that it serializes all parallelism found in an ISPS description and establishes parallelism in the microcode using an algorithm of its own. Multilevel representations that use design representations that implement parallelism as specified by the behavioral specification will need to handle parallelism. Supporting multiple processes and parallelism should not be difficult; it is anticipated that they will be added to future design representations that support parallelism and multiple processes.

7.1.2 Limitations in the Multilevel Representation Implementation

The major limitation of the multilevel representation due to implementation is the size of the multilevel representation program, Blink. For example, Blink occupies about 126K words of memory with the full multilevel representation for the PDP-8 design example described in Section 6. It is expected that program size could be reduced somewhat by tuning the Blink code, but even then it would be difficult to handle much larger designs due to the complexity of design representations. Lower level design representations require increasingly more space to represent, so that representing any more than a very small design in this way is impractical. It will be necessary, therefore, to implement more than a toy multilevel representation as a database system rather than as a collection of data structures. Meyer [Meyer 81] has addressed this topic.

Another shortcoming of the multilevel representation's implementation is the manner in which it builds the multilevel representation. Links between representations are established in Blink primarily by searching through the different design representations. While this poses some interesting problems, it is not a desirable approach for handling large design representations due to the complexity of the searching. Future multilevel representations should be built up incrementally as new design representations are created at each level of abstraction, so that this step should be unnecessary.

7.1.3 Limitations in the Multilevel Representation due to CMU-DA Design Representations

The most glaring limitations to the multilevel representation are inherited from the design representations of the CMU-DA synthesis software. The initial functional structure representation, the path graph, was defined to describe data path designs generated by the distributed design style data-memory allocator. The logical/physical structure representation, an extended path graph, was defined to describe the completed design. These descriptions have two major shortcomings: Parts of the description are incomplete in that they do not completely or only implicitly describe parts of the design, and parts of the description are very irregular, making them difficult to use or extend. The effect of these limitations is that at this time the multilevel representation is only usable with designs in the distributed design style such as those generated by the data-memory allocator. Attempting to use the current multilevel representation software with other types of designs without modifying the design representations substantially would be very difficult. Current research in the CMU-DA project is defining new design representations. It is expected that most of the problems encountered in the current design representations will be corrected in these new representations.

7.2 Future Work

There are several issues involving multilevel representations that deserve further research. The multilevel representation software implemented for this project provides some immediate opportunities for future work. In addition, experience in implementing and defining this software also suggests some issues that should be considered in a "second system" multilevel representation to replace the current software. Finally, applications that use the multilevel representation need to be defined and implemented.

7.2.1 Immediate Applications of the Multilevel Representation Software

The multilevel representation software that has been implemented for this project provides a collection o. data structures that allow interactive access to corresponding features of the behavioral and logical/physical structure design representations generated by the CMU-DA software. While new design representations will probably make this software obsolete shortly, there are several immediate applications of the multilevel representation software that would be interesting for short-term experimental research.

Some of these applications include:

- *Top Down Design Measurement* It may be desirable to use the multilevel representation to automatically measure and feed back implementation and performance information to higher level analysis and synthesis programs in the CMU-DA system. The timing abstraction task implemented in this project was an example of one such measurement; others might measure cost, size, and other metrics and associate this information with the behavioral representation. The multilevel representation provides a testbed on which some of these measurement tasks might be implemented and evaluated.

- *Other applications of the multilevel representation software.* The multilevel representation software is structured in such a way that design representation data structures and code for each level of design representation can be separated easily. These data structures are sufficiently general that with slight modification they could be used separately without the complete multilevel representation for several CMU-DA related tasks.

7.2.2 Issues in Designing Future Multilevel Representations

There are several issues which warrant future work in designing a "second system" multilevel representation. We will discuss in detail two of them: Designing a multilevel representation to use new design representations currently being implemented in the CMU-DA project, and extending multilevel representations to include lower levels of representation.

7.2.2.1 The Value Trace: A New Design Representation Methodology

The scope and utility of the multilevel representation implemented in this project is limited by its basis on the current CMU-DA design representations. The definition of new design representations provides the opportunity to replace the current multilevel representation with a more useful one. Definition of new design representation is currently underway as part of the current effort in the CMU-DA project to generate a second set of synthesis programs. The new CMU-DA design representations show a substantial departure from the ones defined by the previous software. A new representation that has major impact is the *Value Trace* (VT). Snow [Snow 78] noted that algorithmic descriptions of behavior such as ISPS add unnecessary limitations and artifacts to a behavioral description. The VT was developed in an effort to remove these limitations and artifacts from the behavioral representation and provide a basis for applying optimizations and other transformations. The VT is similar to data flow graphs [Aho 77] used for code optimization by optimizing compilers but retains control information that a data flow graph does not. Nodes in the VT graph represent operations from the ISPS and *values* generated by these operations.

The values described by the value trace replace the register and memory variables found in ISPS but also include the intermediate values generated in ISPS expressions. Value nodes corresponding to register and memory variables in the ISPS are normally not constrained to be implemented in a design in the same way that they were initially described. If it is necessary to specify certain register and memory variables that must be implemented with directly corresponding registers and memories in the design, they are declared as *global variables,* which are constrained to be implemented as they are described in the ISPS description. This construct allows a design to remove the noncritical bindings of values to registers which could cause artifacts of the ISPS description to appear in the design.

The major impact of this approach is that transfer operations, which were important in the previous multilevel representation definition, often become meaningless in a VT-oriented multilevel representation because there are no corresponding operations (or variables for them to write into) in the lower level design representations. Instead, the important behavioral features of the VT are the operation nodes. These operations correspond directly to operations in the ISPS description and include both data and control operations. A multilevel representation between the ISPS and lower design representations must center its behavioral links on these operations.

Another issue of interest in VT-oriented design representations is optimization and its effects. Several optimizing transformations have been defined that operate on the VT. A multilevel design

representation must be tolerant of these optimizations while maintaining links between the ISPS behavior, the VT, and lower level design representations.

A multilevel representation using the VT and related lower level could be implemented in a manner very similar to the one described in this report for the present CMU-DA design representations, although an additional level of representation, the VT level, would be added. Behavioral links would still link control operations and data operations as they do in the present design representation. However, transfer operations would not be linked, since they would have no corresponding operations in the VT or lower level descriptions. The biggest difference in the old and new multilevel representations would be in the definition of structural links. In a VT-based design system there is no requirement that non-global ISPS register and memory variables be bound to actual registers and memories in the actual design. For this reason, direct structural correspondences cannot be made between ISPS variables and registers and memories in lower level representations. Instead, each variable maps to several value nodes of the value trace that correspond to values that were once assigned into the ISPS variables. Each value node may then correspond to a register or memory in the functional structure representation that describes the design implementation. Figure 7-1 shows the hierarchy of design representations and links that the new multilevel representation would use.

The VT and related design representations are currently being implemented as a collection of programs to support future CMU-DA synthesis software. It is expected that these design representations will incorporate the multilevel representation described **above.**

7.2.2.2 Adding Lower Levels of Abstraction to the Multilevel Representation

A major issue which has not been addressed by this project is the extension of the multilevel representation defined in this report to include lower levels of representation. Such an interface is necessary if the multilevel representation is to be truly useful as a design tool. Low level design representations have been well defined by the development of numerous CAD tools used in integrated circuit design and fabrication. As lower level design tasks are integrated into the framework of the CMU-DA synthesis system, the multilevel representation will need to be extended to include lower level design representations. At these levels of abstraction it will probably be impractical to attempt to relate behavioral features as was defined in the current multilevel representation. Instead, logic level and lower level representations should probably be decomposed structurally into the lower level components that they represent. A complete multilevel representation would then relate behavioral and structural information between high level and medium level representations and relate mainly structural information between medium level and low level representations.

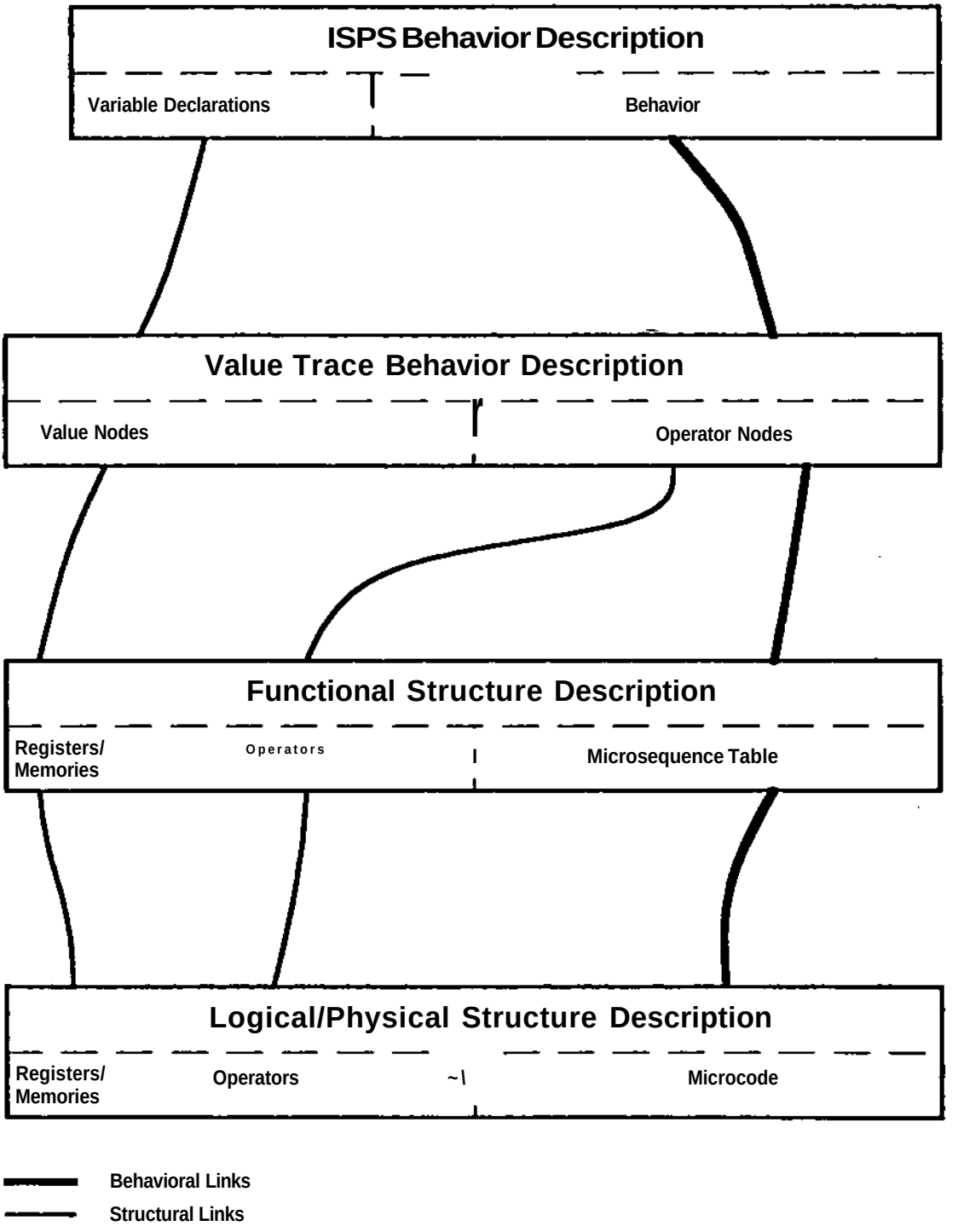

Figure 7-1: The VT-based Multilevel Representation

7.2.3 Future Applications of Multilevel Representations

The unique feature of the multilevel representation defined in this report is that it allows information about the behavior of a design representation to be related between levels as well as structural information. There are many possible applications of this feature. Two of particular interest are design feedback for a large design system and a new approach to multilevel simulation.

7.2.3.1 Design Feedback for Analysis and Synthesis of Designs

The multilevel representation described in this report allows access to the internal features of design representations at multiple levels of abstraction. This capability could be exploited to measure and compare design performance, cost, and implementation information and relate it back to higher levels of abstraction. One such application that was demonstrated in this report was timing abstraction, which extracted timing information from the logical/physical structure description and added it to the corresponding behavioral representation; other measurements have been proposed. Additional parameters that it might be possible to measure and attribute to different parts of higher level descriptions are cost and size of different parts of the design, as well as complexity of interconnection between these parts.

These information obtained by such measurements could be used for several applications. For example, this information could be fed back into logic and synthesis programs to direct them toward better implementations of designs. In addition, measurements collected from several different designs might make it possible to define *predictors* that predict the bounds on cost, speed, and size of an arbitrary design specification using its behavioral specification before a corresponding design is implemented. These predictors could then be used to aid either human or automatic designers in partitioning and implementing a design from its behavioral specification. Leive [Leive 81] has done some preliminary work in this area with the LSMS task. Design measurements and feedback using the multilevel representation would allow this work to be continued and extended.

7.2.3.2 A New Approach to Multilevel Simulation

Multilevel simulation has been suggested as a technique for reducing the cost of simulating large designs while retaining some of the accuracy associated with low-level simulation. Previous efforts at multilevel simulation such as SABLE have decomposed a design representation structurally into components that can be simulated at different levels of abstraction. This decomposition made it possible to mix simulation of the system at the top level of abstraction with lower levels. The multilevel representation defined by this project suggests the possibility of a different approach for multilevel simulation that will allow mixing simulation of the behavioral level with lower levels of representation.

The multilevel representation defined in this project relates together operations in each level of abstraction that cause changes in the state of each level as behavioral links. A possible multilevel simulation scheme would use these behavioral links as points where the level of simulation can be changed. Since these operations cause state changes in each level of representation, these points provide a uniform interface between levels of simulation. Changing the level of simulation would then require translation of the state of one level of simulation to another level of simulation. This approach differs considerably from previous multilevel simulation approaches. In these approaches, simulation states for each level of simulation are maintained concurrently; translation between levels of simulation is passed by interconnections of components.

A major concern in implementing a multilevel simulation in this fashion is how to translate from one level of simulation to the next when a change in level is specified. It is necessary at these points to translate the state of the current simulation to the state of the new level of simulation. Parts of this translation are straightforward. The behavioral links specified in the multilevel representation help in this translation, but there are complications. For example, if the design implements procedures, then part of the state of the design is the hardware that records the procedure calling sequence. There must be sufficient knowledge in the multilevel simulator about the control part of the design so that the state of the calling sequence hardware can be translated to a behavioral level simulation calling sequence and vice versa, or the simulation must return to the previous level of simulation before the calling sequence if changed.

This proposal for a multilevel simulation technique is unique in that it allows the behavioral level to be mixed with lower levels of abstraction in simulation. However, further analysis and examination will be required to determine whether it is feasible to implement.

I. Appendix: Complete Simulation Transcripts

This appendix contains the complete, unedited simulation transcripts for the simulations which are described in Section 6.

Effective Address Simulation

```
.IHITIA

CMU10A 8.4/DEC 6.02A-VM 19:56:59 TTY131

.log n750jn23
JOB 30 CMU10A 8.4/DEC 6.02A-VM TTY131
Other jobs logged in as N750JN23: 34
1958    25-Aug-81       Tue

.ru s1m
ISP SIMULATOR V10.4


Sequential Simulation? [YES]:
Type HELP for Help
Type tCtC to Interrupt Simulation Loops
Latest News: 10 Jun 81

>l              Effective Address PDP8 Simulation Example
>
>time
>daddress
>
>trace eadd m
>break eadd
>abreak eadd
>
>!      Page 0 Direct Addressing
>
>setva1ue op«0
>setva1ue 1b«0
>setva1ue pb«O
>setvalue pa-45         1 effective address should be #45
>
>bva1ue i
Not a Command
>va1ue i

I*#0045
>
>start eadd

B: t - 0+1      BEGIN

*c
```

```
T: t = 3+0      EADD=#0045
B: t = 6+0       END

**c

Simulation Completed
Run Time(Milliseconds)=17
RTM OPS EXECUTED=8
>>
>
>!      Page 0 Indirect
>
>setvalue ib=1
>setvalue pb=0
>setvalue pa=45
>setvalue m[pa]=55              ! effective address should be #55

%WARNING:       PA is Unknown or Invalid
>setvalue m[45]=55
>
>start eadd

B: t = 0+1       BEGIN

*c

T: t = 3+0      EADD=#0045
T: t = 8+0      EADD=#0055
B: t = 9+0       END

**c

Simulation Completed
Run Time(Milliseconds)=21
RTM OPS EXECUTED=13
>>
>!      Page 0 Autoincrement/Indirect Addressing
>
>setvalue ib=1
>setvalue pb=0
>setvalue pa=12          ! note that m[#12] is an autoincrement register
>setvalue m[12]=66       ! effective address should be #67
>v

>value i

I=#0412
>start eadd

B: t = 0+1       BEGIN

*c

T: t = 3+0      EADD=#0012
```

```
T: t • 11+0     M[#12]«#0067
T: t - 13+0     EADD«#0067
8: t • 14+0      END

••c

Simulation Completed
Run Time(Mi111seconds)-26
RTM OPS EXECUTED-16
»
>l      Current Page Direct Addressing
»
>setva1ue 1b*0
>setvalue pb«l
>setva1ue pa*20
>setva1ue cpage*3       I effective address should be #320
»
>start eadd

B: t - 0+1       BEGIN

•c

T: t • 3+0      EADD*#0620
B: t • 5+0       END

••!1 oops, I meant #6201
•c

Simulation Completed
Run Time(Mmiseconds)-21
RTM OPS EXECUTED-7
»
>l      Page 1 Indirect
>
>setvalue 1b«l
>setvalue pb-1
>setva1ue pa-20
>setva1ue cpage*3

>setva1ue m[620]>700    (effective address should be #700

»start eadd

B: t • 0+1       BEGIN

•c
T: t - 3+0      EADD*#0620
T: t • 7+0      EADD«#0700
B: t • 8+0       END

••c

Simulation Completed
Run Time(Ni111seconds)«2l
```

```
RTM OPS EXECUTED-12
»
>I      Current Page Autoincrement/Indirtct
»
>setvalue ib-1
>setva1ue pb«l
>setvalue pa-12
>setva1ue cpage-0
>setva1ue m[12]•333     I effective address should be #334
»
>start eadd

B: t • 0+1       BEGIN

•C

T: t « 3+0       EADD«#0012
T: t - 10+0      M[#12>#0334
T: t - 12+0      EADD-00334
B: t - 13+0       END

••c

Simulation Completed
Run Time(Milliseconds)«27
RTM OPS EXECUTED-15
»exit

EXIT
```

Multiply Program Simulation

```
.IMITIA

CMU10A 8.4/DEC 6.02A-VM 15:26:15 TTY132

.log n750jn23
JOB 26 CMU10A 8.4/DEC 6.02A-VM TTY132
Other jobs logged in as N750JN23: 25
1526    26-Aug-81       Wed

.ru sim
ISP SIMULATOR V10.4


Sequential Simulation? [YES]:
Type HELP for Help
Type +C+C to Interrupt Simulation Loops
Latest News: 25 Aug 81

>echo
>read mult

»define teliproduct-value m[210] $
```

```
»define tel1a*value m[211] $
»define tellb*value m[212] $
»>
»define startmult*setva1ue pc-200
start interpret $
»>
»radix octal
»>
»time
»daddress
»>
»>
»setvalue m[30]-0213
»>
»setvalue m[200]>7300  ! start.      cla ell          ; clear link & ac
»setvalue m[201]«1211  !       tad a           ; move a into ac
»setvalue m[202]>3205  !             dca . • 3       ; store it in temp
»setvalue m[203]>1212  !             tad b           ; move b into ac
»setvalue m[204]>4430  !             jms i 30        ; call mult subroutine
»setvalue m[205]«0000  !             0000            ; temp value
»setvalue m[206]«3210  !             dca prduct      ; move result to prduct
»setvalue m[207]«7402  !             hit             ; all done
»setvalue m[210]«0000  ! prduct,
»setvalue m[211]«0000  ! a,
»setvalue m[212]→0000  ! b,
»setvalue m[213]'0000  ! mult. 0000           ; return address (also points to data)
»setvalue m[214]→7041  !       cia    ( cma iac )    ; set ac to -b
»setvalue m[215]>3223  !       dca mtally      ; move it to mtally
»setvalue m[216]«1613  !       tad i mult      ; add temp (a) to ac
»setvalue m[217]→2223  !       isz mtally      ; increment
»setvalue m[220]>5216  !       jmp . - 2       ; if mtally <> 0 loop back
»setvalue m[221]↗2213  !       isz mult        ; set return address to temp + 1
»setvalue m[222]«5613  !       jmp i mult      ; return
»setvalue m[223]↗0000  ! mtally. 0000         ; loop counter
»35 Lines Read
>
>trace m lac ac
>
>tella

>
M[#211]«f0000

»setvalue m[211]-4
>
>tellb

>
M[f212]-fOOOO
»setvalue m[212]-5
>
>startmult

»>
T: t - 8+0      AOJ0000
```

```
T: t • 26+0     LAOJ00004
T: t • 39+0     M[#205]«#0004
T: t • 39+1     AOfOOOO
T: t • 54+0     LAOJ00005
T: t • 71+0     M[#213]-#0206
T: t • 78+0     AC-#7772
T: t • 79+0     LAC«#07773
T: t « 94+0     M[#223]«#7773
T: t • 94+1     AC*#OOOO
T: t • 112+0    LAOJ00004
T: t - 128+0    M[#223]-#7774
T: t - 162+0    LAC-#00010
T: t - 178+0    M[#223]-#7776
T: t • 212+0    LAC-#00014
T: t - 228+0    M[#223]-#7776
T: t • 262+0    LAC*#00020
T: t « 278+0    M[#223]«#7777
T: t • 312+0    LAC*#00024
T: t • 328+0    M[#223]«#0000
T: t - 349+0    M[*213]«#0206
T: t • 381+0    M[#210]«f0024
T: t « 381+1    AC-#0000
Simulation Completed
Run Time(Milliseconds)«239
RTM OPS EXECUTED-805
»tellproduct

➤
M[#210]«#0024

»ex1t

EXIT
```

References

[Aho 77] A.V. Aho and J.D. Ullman.
Principles of Compiler Design.
AddisorvWesley, Reading, MA, 1977.

[Altman 80] Altman, Arthur, and Alice Parker.
The Slide Simulator: A Facility for the Design and Analysis of Computer Interconnections.
In *Proceedings of the 17th Design Automation Conference,* pages 148-155. ACM, IEEE, September, 1980.

[Barbacci 79] Barbacci, Mario R.
The Register Transfer Machine.
Technical Report, Department of Computer Science, Carnegie-Mellon University, September, 1979.

[Barbacci 80] Barbacci, Mario R., Nagle, Andrew W., Northcutt, J. Duane.
An ISPS Simulator.
Technical Report, Departments of Computer Science and Electrical Engineering, Carnegie-Mellon University, January, 1980.

[Barbacci 81] Barbacci, Mario R.
Instruction set processor specifications (ISPS): The notation and its applications.
IEEE Transactions on Computers C-30(1):24-40, Jan., 1981.

[Cloutier 80] Cloutier, Richard J.
Control Allocation: The Automated Design of Digital Controllers.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, April, 1980.

[Darringer 79] Darringer, John A.
The Application of Program Verification Techniques to Hardware Verification.
In *Proceedings of the 16th Design Automation Conference,* pages 375-381. IEEE/ACM, 1979.

[DEC 73] Digital Equipment Corporation.
Introduction to Programming.
Digital Equipment Corporation -, Maynard, Massachusetts, 1973.
(PDP-8 Programming Manual).

[Director 81] S.W. Director, A.C. Parker, D.P. Siewiorek and D.E. Thomas.
A Design Methodology and Computer Aids for Digital VLSI Systems.
IEEE Transactions on Circuits and Systems CAS-28(7):634-645, July, 1981.

[DiRusso 81] DiRusso, Renato.
Implementation of a Design Using the CMU-DA System.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, October, 1981.

[Hafer77j Hafer.Lou.
Data-Memory Allocation in the Distributed Design Style.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, December, 1977.

[Hill 79] Mill, Dwight and William vanCleemput.
SABLE: A Tool for Generating Structured, Multi-Level Simulations.
In *Procedings of the 16th Design Automation Conference,* pages 272-279. IEEE/ACM, 1979.

[Lawson 78] Lawson, Gregory L.
Design Style Selector, An Automated Computer Program Implementation.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, August, 1978.

[LeiveSO] Leive, Gary W.
The C.vfU Design Automation System: The SYNNER's Salvation.
Technical Report, Department of Electrical Engineering, Carnegie-Mellon University, September, 1980.
Internal Report (SYNNER Maintenance Document).

[Leive 81] Leive, Gary W.
The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.
PhD thesis, Carnegie-Mellon University, January, 1981.

[McFarland 78] Me Far land, Michael C.
The Value Trace: A Data Base for Automated Digital Design.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, December, 1978.

[McFarland81] Me Far land, Michael C.
Mathematical Models for Formal Verification in a Design Automation System.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, July, 1981.
Available as a Technical Report, Design Research Center, Carnegie-Mellon University.

[Mead 80] C. Mead and L. Conway.
Introduction to VLSI Systems.
Addison-Wesley, 1980.

[Meyer 81] Meyer, Michael.
Databse Considerations for a Multi-Level Design Automation System.
Master's thesis, Department of Electrical Engineering, Carnegie-Mellon University, October, 1981.

[Nagle80] Nagle, Andrew W.
Automated Design of Digital-System Control Sequencers from Register-Transfer Specifications.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, December, 1980.

[Newton 79] Newton, A. Richard.
Techniques for the Simulation of Large-Scale Integrated Circuits.
IEEE Transactions on Circuits and Systems CAS-26(9):741-749, September, 1979.

[Parker 79] A.C. Parker, D.E. Thomas, D.P. Siewiorek, M.R. Barbacci, G. Leive and J. Kim.
The CMU Design Automation System: an example of automated data path design.
In *Proceedings of the 16th Design Automation Conference,* pages 73-80. ACM SIQDA and IEEE Computer Society DATC, June, 1979.

[Sakallah80] Sakallah, K. and Dii 3Ctor, S. W.
An Activity-Directed Circuit Simulation Algorithm.
In *Proc. IEEE ICCC$_t$* pages 1032-1035. IEEE, 1980.

[Siewiorek 76] Siewiorek, Daniel P. and Mario R. Barbacci.
The CMU RT-CAD System: An Innovative Approach to Computer Aided Design.
In AFI PS Conference Proceedings, Vol. 45, pages 643-655. AFIPS, 1976.

[Snow 78] Snow, Edward A.
Automation of Module Set Independent Register-Transfer Level Design.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, April, 1978.

[Thomas 77] Thomas, Donald E., Jr.
The Design and Analysis of an Automated Design Style Selector.
PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, March, 1977.