Behavioral Synthesis from VHDL Using Structured Modeling

Joseph S. Lis
Daniel D. Gajski

Technical Report #91-05
January, 1991

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-7063

Abstract

This dissertation describes work in behavioral synthesis involving the development of a VHDL Synthesis System **VSS** which accepts a VHDL behavioral input specification and performs technology independent synthesis to generate a circuit netlist of generic components. The VHDL language is used for input and output descriptions. An intermediate representation which incorporates signal typing and component attributes simplifies compilation and facilitates design optimization.

A **Structured Modeling** methodology has been developed to suggest standard VHDL modeling practices for synthesis. Structured modeling provides recommendations for the use of available VHDL description styles so that optimal designs will be synthesized.

A design composed of generic components is synthesized from the input description through a process of Graph Compilation, Graph Criticism, and Design Compilation. Experiments were performed to demonstrate the effects of different modeling styles on the quality of the design produced by VSS. Several alternative VHDL models were examined for each benchmark, illustrating the improvements in design quality achieved when Structured Modeling guidelines were followed.

UNIVERSITY OF CALIFORNIA

IRVINE

Behavioral Synthesis from VHDL Using Structured Modeling

DISSERTATION

submitted in partial satisfaction of the requirements for the degree

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Joseph Stephen Lis

Dissertation Committee:

Professor Daniel D. Gajski, Chair

Professor Lubomir Bic

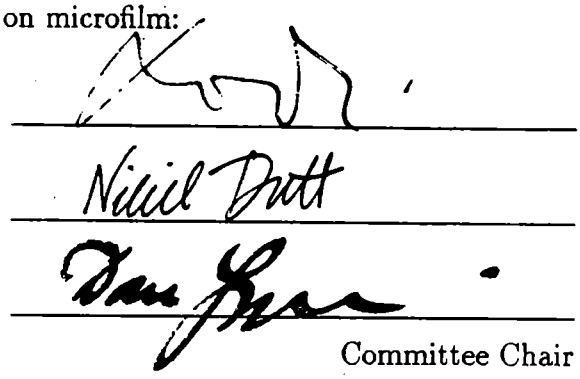
Professor Nikil D. Dutt

1991

The dissertation of Joseph Stephen Lis is approved,

and is acceptable in quality and form for

publication on microfilm:



Committee Chair

University of California, Irvine

1991

Dedication

This dissertation is dedicated to my parents, Stephen J. and Virginia M. Lis, who have given me the strong foundations of love and support necessary to handle the ups and downs of life.

This dissertation is also dedicated to my grandmother Jeannette Lis, whose constant prayers and encouragement have been a source of inspiration for me throughout this ordeal.

Contents

List of Figures

List of Tables

Acknowledgements

When I first arrived at U.C. Irvine after being lured from the flatlands of the Midwest by Prof. Daniel Gajski, the CADLAB consisted of one empty room with a bookshelf bolted to the wall and a single desk in the middle of the room. After numerous iterations of partition construction, workstation setups, and dreaded "moving" escapades, I feel that I have been a part of the establishment of a first-class operation over the past four years. Along the way, Prof. Gajski has taught me a great deal about our common obsession of behavioral synthesis, as well as a thing or two about life in general. Although we haven't always agreed on how to get from point A to point B, I respect and admire his dedication and vision, and I feel privileged to have had the opportunity to work with him. Thank you, Prof. Gajski; I hope you have the same good memories of our accomplishments here as I know I will.

I would like to thank my committee members, and Prof. Lubomir Bic and Prof. Nikil Dutt, for their interest, guidance and probing questions. I have shared the unique experiences of graduate study with Nikil when we were colleagues during my days at the University of Illinois. The fact that he has remained involved with the behavioral synthesis projects initiated by Prof. Gajski is testimonial to the fact that something worthwhile must be happening at UCI. Thank you for your advice and friendship.

I would like to acknowledge the members of the CADLAB fraternity whose comaraderie have made my graduate research experience not only tolerable, but most enjoyable. Thanks to Allen Wu, Tedd Hadley, Nels Vander Zanden, Sanjiv Narayan, Frank Vahid, Elke Rundensteiner, Rajesh Gupta, Loganath Ramachandran, Viraphol Chaiyakul, and Jim Fradkin for your contributions to this work, and for keeping me honest by putting my software through the acid test. In addition, the many hours of volleyball, midnight snack excursions, and day to day small talk you've shared with me have helped to preserve my sanity during the trying times of graduate study.

I would also like to express my appreciation for the efforts of Bob Larsen of Rockwell International who has shown enthusiasm and strong support for the design synthesis work at U.C. Irvine. Bob has been instrumental in pulling the necessary "corporate strings" so that we could validate our work using industrial examples. Thanks for all you've done, Bob.

Another valuable resource at UCI has been the computer support group, with special thanks to Sam Horrocks. I appreciate the time you've spent installing new software, fixing networking problems, and dealing with the demands placed on the machines which have been vital to the operation of CADLAB.

I know I wouldn't have been able to climb this mountain without the counsel and support I received from my family. I thank you for being there when I needed that encouraging word.

This work was supported through grants from Texas Instruments, TRW and funding from SRC 90-DJ-146 and NSF MIP-8922851 contracts.

Curriculum Vitae

Education:

1991: Ph.D. in Computer Science
- Department of Information and Computer Science
- University of California, Irvine
- Dissertation Topic: Behavioral Synthesis from VHDL
- Advisor: Professor Daniel D. Gajski

1985: M.S. in Computer Science
1983: B.S. in Computer Science
- Department of Computer Science
- University of Illinois at Urbana-Champaign
- Master's Thesis Advisor: Professor Daniel D. Gajski

Experience:

1. 1987–91: Research Assistant, Department of Information and Computer Science, University of California, Irvine
2. 1985–86: Member of the Technical Staff, Gould Inc., Gould Research Center, Electronic & Computer Systems Laboratory, Rolling Meadows, IL
3. 1984–85: Teaching Assistant, Department of Computer Science, University of Illinois at Urbana-Champaign
4. 1982–84: Summer Engineering Intern, Northrop Corporation, Defense Systems Division, Rolling Meadows, IL

Refereed Conference and Workshop Papers:

Potasman, R., Lis, J., Nicolau, A., and Gajski, D., "Percolation Based Synthesis". In *Proceedings of the 27th Design Automation Conference* (pp. 444-449), Orlando, FL, June 1990.

Lis, J. and Gajski, D., "Structured Modeling for VHDL Synthesis", invited speaker, *Fourth International Workshop on High-Level Synthesis*, Kennebunkport, ME

Lis, J. and Gajski, D., "VHDL Synthesis Using Structured Modeling". In *Proceedings of the 26th Design Automation Conference* (pp. 606-609), Las Vegas, NV, June 1989.

Lis, J. and Gajski, D., "Synthesis from VHDL". In *Proceedings of the International Conference on Computer Design* (pp. 378-381), Rye Brook, NY, October 1988.

Technical Reports:

Lis, J. and Gajski, D., "Structured Modeling for VHDL Synthesis", *Technical Report 89-14*, University of California at Irvine, June 1989

Lis, J. and Gajski, D., "VHDL Design Representation in the VHDL Synthesis System", *Technical Report 89-15*, University of California at Irvine, June 1989

Lis, J. and Gajski, D., "VSS: A VHDL Synthesis System", *Technical Report 88-13*, University of California at Irvine, May 1988

Invited Talks:

"Design Synthesis from VHDL", tutorial presented at the *1989 VHDL Users' Group Meeting*, Redondo Beach, CA, October 22, 1989

"VHDL Synthesis Using Structured Modeling", invited speaker, *VHDL Users' Group Meeting*, Las Vegas, NV, June 29, 1989

VHDL Synthesis System prototype, invited demonstrations, *27th Design Automation Conference* (Orlando, FL 1990), *26th Design Automation Conference* (Las Vegas, NV 1989), *25th Design Automation Conference* (Anaheim, CA 1988)

Honors, Professional Activities and Societies:

- Regents' Dissertation Fellowship, University of California at Irvine, Winter Quarter 1990
- Dean's List, College of Engineering, University of Illinois at Urbana-Champaign, 7 semesters (Fall 1980 through Fall 1983)
- Reviewed papers for IEEE/ACM Design Automation Conference, IEEE Conference on Computer-Aided Design
- Member IEEE Computer Society, ACM SIGDA, Tau Beta Pi

Abstract of the Dissertation

Behavioral Synthesis from VHDL Using Structured Modeling

by

Joseph Stephen Lis

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1991

Professor Daniel D. Gajski, Chair

This dissertation describes work in behavioral synthesis involving the development of a VHDL Synthesis System **VSS** which accepts a VHDL behavioral input specification and performs technology independent synthesis to generate a circuit netlist of generic components. The VHDL language is used for input and output descriptions. An intermediate representation which incorporates signal typing and component attributes simplifies compilation and facilitates design optimization.

A **Structured Modeling** methodology has been developed to suggest standard VHDL modeling practices for synthesis. Four design models currently understood and used in practice by designers have been identified: *combinational logic*, *functional descriptions* (involving clocked components such as counters), *register transfer* (data path) descriptions, and *behavioral* (instruction set processor) designs. Structured modeling provides recommendations for the use of available VHDL description styles (*structural*, *dataflow* and *behavioral*) so that optimal designs will be synthesized.

A design composed of generic components is synthesized from the input description through a process of Graph Compilation, Graph Criticism, and Design

Compilation. Graph Compilation parses the VHDL input description into an internal Control/Data Flow Graph representation. The Graph Critic removes inefficiencies introduced by certain language constructs and makes local optimizations in the flow graph structure.

The Design Compilation process involves a collection of algorithms which map the internal representation to a corresponding structural implementation. Portions of the input description may be modeled using different Structured Modeling design models; the Design Compiler will apply the appropriate synthesis algorithm to each section. Behavioral descriptions are processed using algorithms which consider the interrelated effects of storage and function unit allocation on interconnect and total chip area. The VSS system generates a VHDL *structural netlist* for the data path, and a *state table* which captures control information.

Experiments were performed to demonstrate the effects of different modeling styles on the quality of the design produced by VSS. Several alternative VHDL models were examined for each benchmark, illustrating the improvements in design quality achieved when Structured Modeling guidelines were followed.

Chapter 1

Problem Description

1.1 Introduction

In order to successfully exploit new VLSI design technologies, the problems of rapid prototyping of new systems and redesigning old parts must be solved. To solve these problems, a new generation of design tools that capture human design knowledge must be developed. Unfortunately, the knowledge required to translate functional specifications to structural representations and structural representations into physical design is not sufficiently well understood to allow the development of computer-aided design (CAD) tools based on simple algorithms. To make the problem even more complicated, the functional specifications are often incomplete and given with conflicting design goals.

High-level or *behavioral synthesis* involves the transformation of a specification of the behavior required of a hardware system to be designed given a set of constraints on the implementation into a structure that implements the behavior and satisfies the constraint requirements. The desired "black box" behavior of a design is represented by a mapping of the system's inputs to its outputs over time. A programming language [GK84, JR+89] or a *hardware description language* (HDL) such as ISPS [Bar81]

or VHDL [IEE87] is used to specify this behavioral description. The description contains as little detail about the system's implementation as possible. Examples of such specifications are an instruction set for a special purpose processor, a set of register transfers for an application specific integrated circuit (ASIC) data path, or boolean equations used to describe combinational logic. Constraints are expressed in terms of limitations on the time (the individual clock cycle or total execution time), the area (a total design area or a specified set of functional units, storage elements and interconnects) and/or the power attributes of the design to be implemented.

The resultant structure produced by behavioral synthesis is a set of interconnected components often represented as a *netlist*. Depending on the level of abstraction and corresponding component library, the netlist can consist of component primitives which are complex processors or memories, microarchitecture components such as ALUs, registers and multiplexors, or in some cases simple transistors and wires.

The design process proceeds through several stages of an abstraction hierarchy [GK83], from the algorithmic specification down to layout mask information used to implement the design in silicon. At each level of this process, the specification is refined to add implementation details which are further refined at subsequent levels until the design is completed. High-level synthesis performs the first phase of this refinement to produce a *register-transfer level* (RTL) structure. The details of this design process will be elaborated in subsequent chapters of this dissertation.

The behavioral synthesis task is complicated by the fact that it is difficult to develop a general purpose synthesis system that will produce quality results for a variety of target applications. Existing systems have focused their capabilities on a

restricted domain so as to reduce the complexity of the design task. Unfortunately, these systems are often too specialized or inflexible to apply to a majority of real world designs. A synthesis methodology which addresses these needs has the following primary requirements: well-defined *design models* and *modeling practices*, a *flexible design representation*, and an *extendable system framework*.

In order to successfully perform behavioral synthesis, the abstract functionality expressed in the input description must be mapped onto a physical implementation or architecture that has a known *model* of execution or computation. One design model which is appropriate for every situation is difficult to define; it may be more feasible for a synthesis tool to produce designs targeted to a set of a few design models representing a majority of real world designs. When using an existing language or developing one's own language for the representation of a particular design model, a semantics must be determined for synthesis. This will allow the tools to use a consistent method of interpreting what the designer meant when a particular construct of the language is used, and more importantly, how the tool will interpret this statement. Because there can be no guarantee of uniqueness of descriptions written in a particular language, it is necessary to establish *modeling practices*. These guidelines should provide a consistent interpretation of the language constructs used by the designer (in writing the models) and the synthesis tool (in synthesizing the description).

The synthesis process requires an intermediate *design representation* or data base which captures the intent of the behavioral description. This format can be manipulated by the synthesis system and transformed into a structure consistent with the chosen design model. Synthesis tools usually begin as an implementation of an initial idea or algorithm. As this approach is refined, the design representation must be flexible enough to satisfy the information storage and manipulation requirements

of new algorithms. Similarly, the synthesis *system framework* should allow for easy integration of new modules which operate on this common design representation.

1.2 Contributions

This thesis describes an approach to behavioral synthesis which uses the VHDL language [IEE87], the IEEE standard language for hardware description. An examination of the issues involved in behavioral modeling is presented, as well as an evaluation of various modeling practices and their effects on the quality of a synthesized design. To demonstrate the feasibility of this approach, the implementation of the **VHDL Synthesis System (VSS)** will be discussed. The novel contributions of this work are described below.

1.2.1 Use of the VHDL Language for Synthesis

While VHDL has been used for modeling for the purpose of simulation and has been adapted for use as a front end language to existing synthesis systems, this work is novel in that from the outset, VHDL was selected as the input and output specification language for synthesis. The syntax of the language has been preserved, and the underlying design model of VHDL has been studied. A synthesis semantics has been developed which uses existing language constructs to represent common hardware models and characteristics.

1.2.2 Design Models

Our synthesis system supports four design models: *combinational logic*, *functional* descriptions (involving clocked components such as counters), *register transfer* (instruction set or data path) descriptions, and *behavioral* (processor) designs. Previous synthesis systems have been limited by a narrow problem domain (e.g., digital signal processing (DSP) compilers such as Cathedral II [DR+86]). While these systems are effective in synthesizing designs in this restricted subset, a majority of real world designs cannot be processed by such systems (e.g., interface or glue logic, designs controlled by asynchronous events).

A potential application for behavioral synthesis is a design description consisting of some previously designed modules as well as portions of behavioral specification. Alternatively, the description could consist of portions which are to be targeted to different design models, requiring that different synthesis algorithms be applied to different portions of the description. The collection of design models used in this work allows for synthesis of a broader range of designs.

1.2.3 Design Representation

An internal design representation was developed to allow for the mapping of VHDL behavioral models (representing different design models) to a common internal format (CDFG) which can be manipulated by synthesis tools. This design representation can be manipulated via behavioral level transformations which recognize design optimizations early in the synthesis process.

1.2.4 Structured Modeling

This work introduces a new methodology termed **Structured Modeling** which was developed to provide modeling guidelines for use of an existing language (VHDL). Structured modeling provides a synthesis semantics for VHDL which identifies preferred representations that will synthesize to high quality designs. This dissertation will illustrate how the quality of a design as well as the complexity of the synthesis process are directly related to the style of description chosen to represent a particular design model.

1.2.5 Synthesis framework

The implementation of this design methodology allows for the integration of the different procedures and the development of control strategies required for synthesis of each design model. The VSS system framework facilitates installation of new subtask algorithms and modules which operate on the common design representation. Thus, a collection of techniques can be made available to the designer, allowing some interactivity in the design process through the selection of synthesis procedures to be applied.

1.3 Thesis overview

This thesis is organized as follows. Chapter 2 discusses the main issues involved in behavioral synthesis. Chapter 3 surveys previous work in the areas of behavioral modeling, its application to synthesis, and in particular, the use of VHDL for these

purposes. Chapter 4 presents the Structured Modeling methodology. Chapter 5 details the design representation developed. Chapter 6 describes the organization and major components of the VHDL Synthesis System prototype. Chapter 7 presents the results of experiments performed using Structured Modeling guidelines to develop models synthesized by the VSS system. Chapter 8 summarizes the accomplishments of this research and outlines future work.

Chapter 2

Synthesis Design Process

There are several concepts which influence the behavioral synthesis approach described in this thesis. This chapter presents the major issues which form the foundation of our approach, and it provides a common terminology which will be used to compare existing behavioral synthesis approaches to this work.

2.1 Design Process

The design process for behavioral synthesis is shown in Figure 2.1. The tasks which compose this process are described in the following subsections.

2.1.1 Representation Compilation

Representation compilation [TW+88] involves parsing a design description and translating it into an internal representation. This representation organizes information extracted from the input specification necessary for synthesis. It is created, manipulated, and optimized by the synthesis system so that a netlist or other output

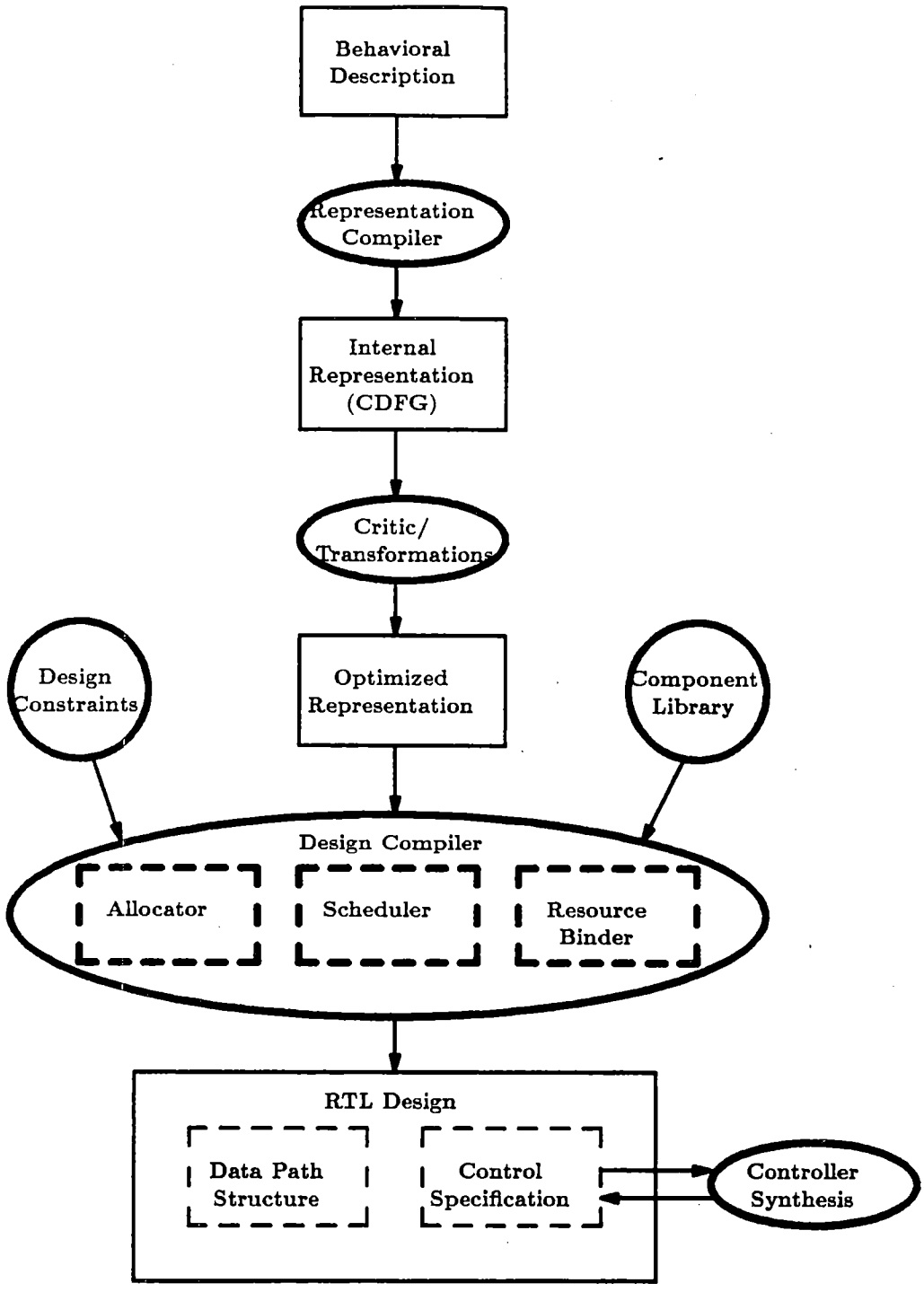


Figure 2.1: Behavioral Synthesis Design Process

specification can be produced. Different optimizations are applied to this representation depending on the design style and design goals.

One common design representation used in several synthesis systems is the **control/data flow graph (CDFG)** [OG86] or *value trace* [McF78]. The *control flow*

```
entity EX is
   port (B,C,D,F,H,I: in BIT_VECTOR(7 downto 0);
      E,G: out BIT_VECTOR(15 downto 0));
end EX;
architecture BEHAVIOR of EX is
begin
   process
      variable A: BIT_VECTOR(15 downto 0);
      begin
         A := (B + C) * D;
         E := F * (B + C);
         G := A + (H - I);
      end process;
end BEHAVIOR;
```

Figure 2.2: Behavioral Description

graph represents sequencing information. Each "state" in the behavioral description is represented as a sequence of actions to be performed, and based on the evaluation of

a condition, the next state to which execution is to be advanced is indicated. Control dependencies implied in the semantics of the behavioral description (for example, loop and if-then-else constructs) are preserved in the control flow graph. Figure 2.2 shows an example input description using the VHDL hardware description language. Figure 2.3 presents a corresponding flow graph representation.

The sequence of actions to be performed (arithmetic, logical, shifting operators) is represented using *data flow graphs.* A data flow graph indicates data dependencies that exist between variable accesses in assignment statements.

The data flow graph exposes the parallelism in the input description. A control flow node representing a state or basic block [LDSM80] will have a data flow graph associated with it.

Most input languages are *procedural*; they describe data manipulations with assignment statements and organize sequences of these statements into blocks using standard control constructs for sequential execution, conditional execution and iteration. An execution ordering is implied by this language paradigm which is maintained in the design representation using *control* and *data dependencies.* Control dependencies are used to sequence control of the design between sequences of assignment statements. Data dependencies ensure that variable assignments and accesses occur in the order specified in the input description.

2.1.2 Optimization of the Internal Representation

Once the design representation is created, global optimizations such as standard language compiler data flow analysis (dead code elimination, constant propagation,

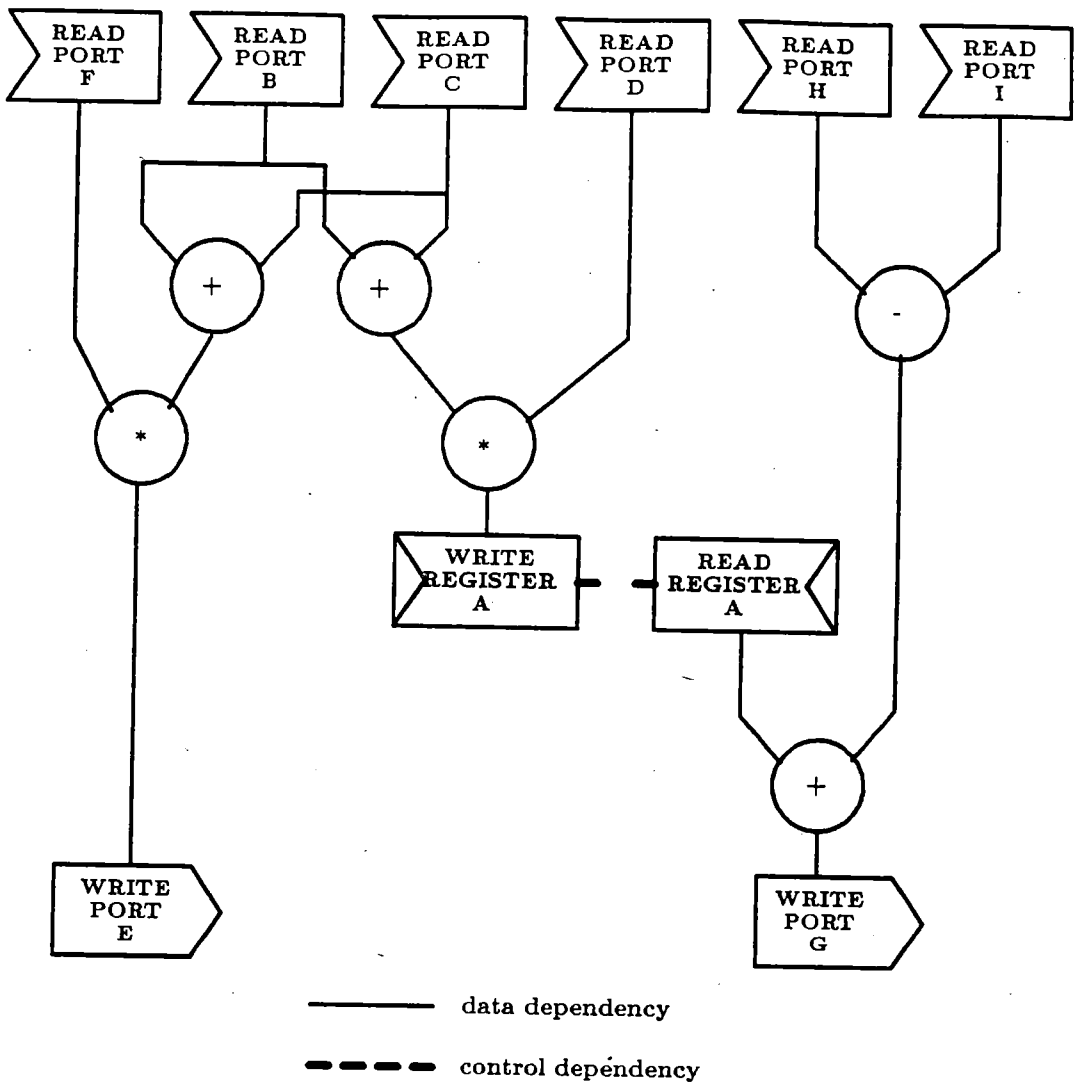


Figure 2.3: Flow Graph Representation

common subexpression elimination, inline expansion of procedures, loop unrolling) are often applied [Tri87, TW+88]. Figure 2.4 shows the results of applying such optimizations to the flow graph of Figure 2.3.

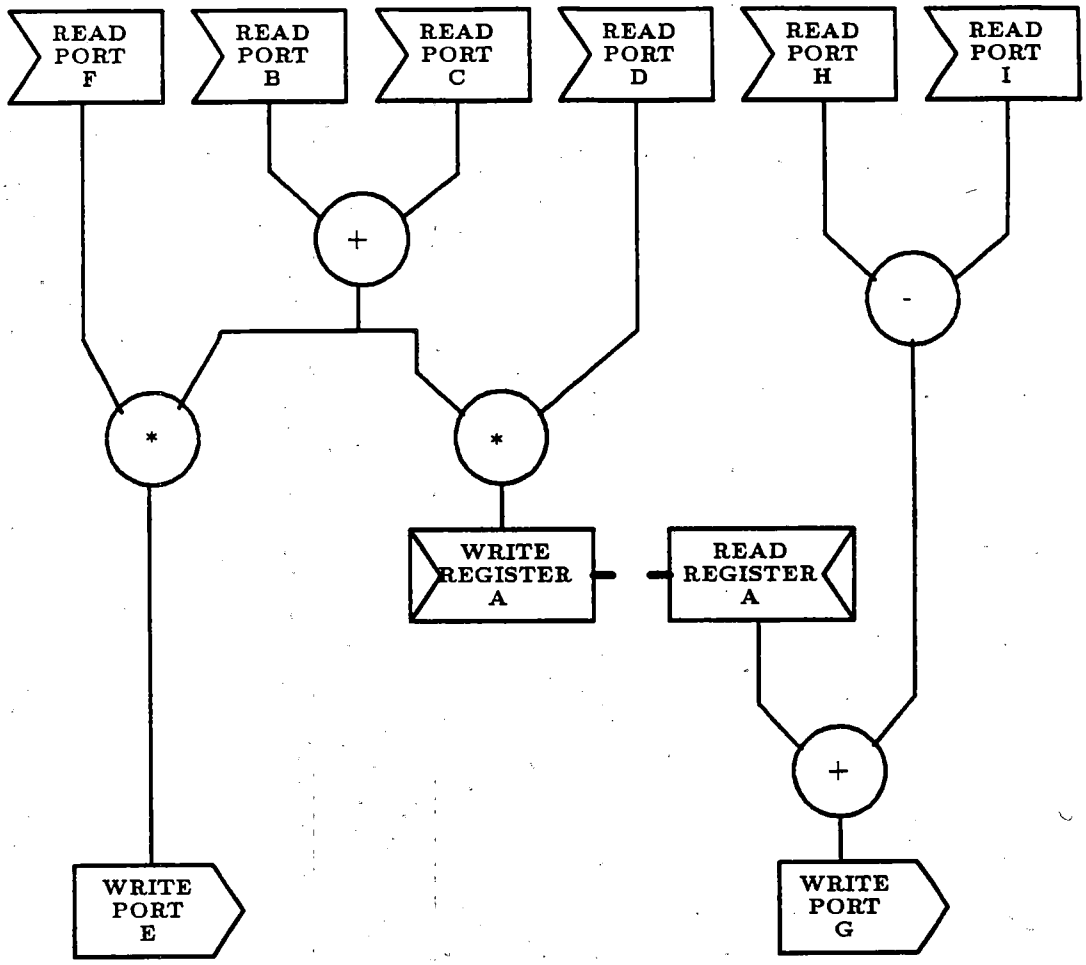


Figure 2.4: Flow Graph after Optimization

In addition, local, hardware-specific transformations such as the identification of signals and registers provide additional information to aid the synthesis task. The

local transformations can be applied to the internal representation by a *graph critic* module [Tri87, LG88] to replace the behavioral information derived from the input description with information which is relevant to the synthesis process (for example, the identification of a variable used as a clock signal). This optimization task simplifies the process of mapping hardware components to the operations in the internal representation.

2.1.3 Allocation

A high-level synthesis system generally assumes that *generic* or technology-independent functional units from a defined library are available to execute the abstract operators in the input description. Libraries provide estimates of component area and propagation delays. One such generic library is GENUS [Dut88].

Resource allocation determines the number and type of functional units, storage elements and communication paths to be used in the design. In some high-level synthesis systems, the designer supplies the allocation [BG87, DN89, Kow85]. Other systems [T+83, GK84, PPM86] provide only component costs via the component library which the scheduler uses to determine the allocation necessary to satisfy scheduling constraints. In this case, the resource binding task generates the actual allocation required.

Figure 2.5 shows two possible allocations for the example. The first is a maximally parallel allocation where there is one unit for every operator in the behavioral description. Alternatively, a user specified allocation is shown which corresponds to the minimal area point of the design space curve for this example.

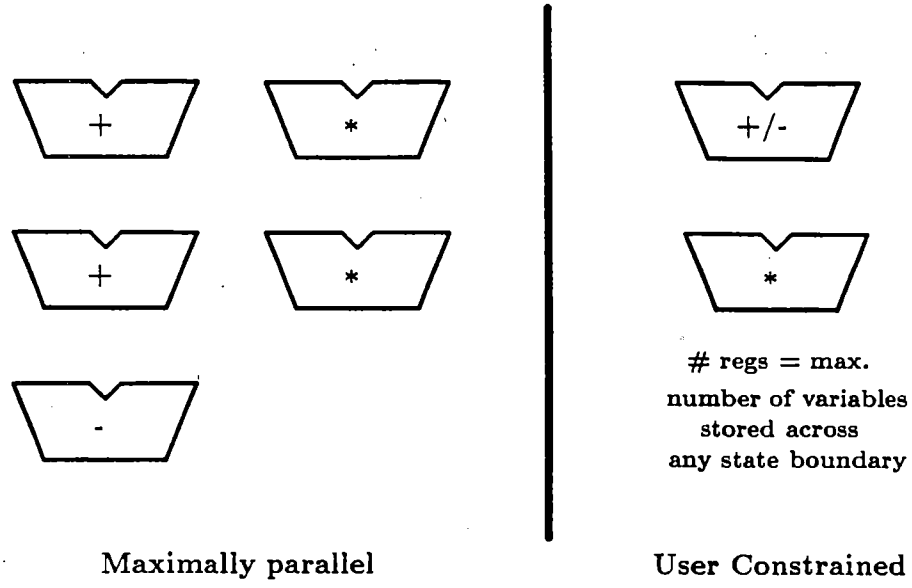


Figure 2.5: Allocation

2.1.4 Scheduling

Scheduling performs *state binding*, or the assignment of operations in the behavioral description to control steps. This state binding must maintain the correct execution order as specified in the input description. If the scheduling reorders operations such that a variable is overwritten before its previous value needs to be accessed, an incorrect result is produced. Scheduling is a critical step in the synthesis process [GDP86] which affects the interrelated task of component allocation. The scheduler tries to execute as many operations as possible in each machine state (i.e., extract as much parallelism as possible).

There are two main approaches to scheduling based on the constraints supplied. In the first case, a limit on time is imposed, either by a specification of the machine's

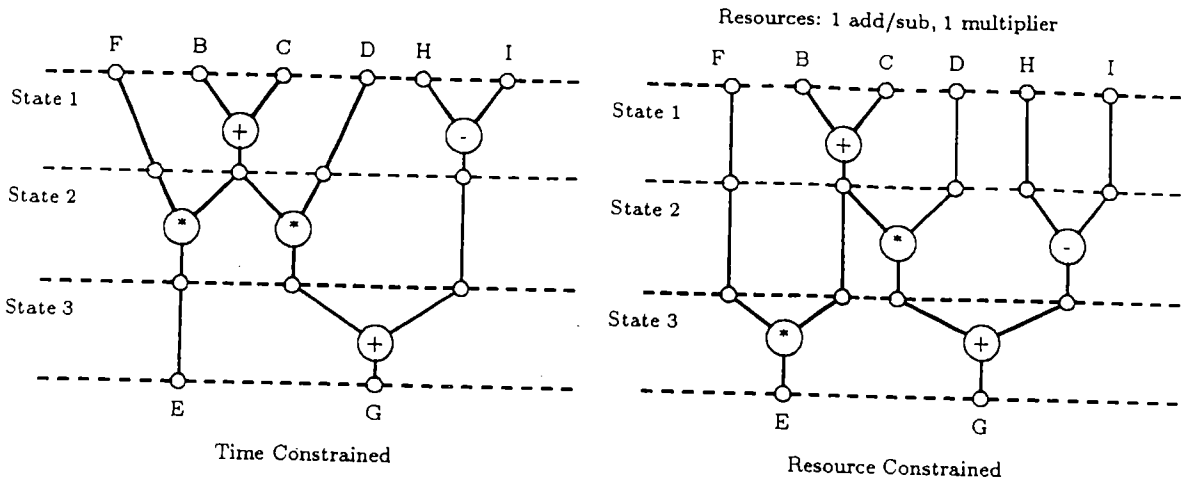


Figure 2.6: Scheduling

clock cycle duration or by the specification of the total execution time. Here, the design must sequence through as few states as possible when executing the behavior. If the scheduler is constrained by a fixed number of machine states, this implies that the scheduler determines the necessary hardware resources required to meet the imposed time constraints (based on function unit, storage element and interconnect costs), if this schedule is possible. A second approach limits the resources the scheduler may use in any state. In this case, the scheduler tries to maximize the utilization of available resources in order to minimize the number of control steps required. Figure 2.6 illustrates one schedule using each of the constraint approaches.

2.1.5 Resource Binding

The task of *binding* or *behavior-to-structure mapping* assigns specific instances of functional, storage and interconnect units to the abstract operations and variables in the behavioral description (or more correctly, the design representation). This task also decides how each component and connection of the data path is to be realized, given possibly several alternatives from the component library.

After state binding maps operations to states, *unit binding* maps each operation to a component which performs the desired function during the particular state. Operators which are not executed in the same state can be mapped to the same functional unit if they are considered compatible (for example, some synthesis systems [TS83] allow the merging of "+" and "*" operators into a common unit, while other systems associate a high cost with such a merge that would favor two separate units).

If a variable is used in more than one state, *register binding* must be performed to assign this variable to a storage element. *Lifetime analysis* [ASU86] is often performed for each variable in the input description to determine which variables can share a storage element. A variable is "live" from the time of its definition to the time of its last use; the variable is dead from the time of its last use to the time of its next definition. Each variable may be mapped to a separate register, or the number of registers can be optimized by sharing (mapping several variables to the same register if the "live" periods of the variables do not overlap).

Connectivity binding allocates a connection between hardware components in order to perform the required data and control transfers. Connections can be *point-to-point* (using a multiplexor component at function unit and register inputs to select

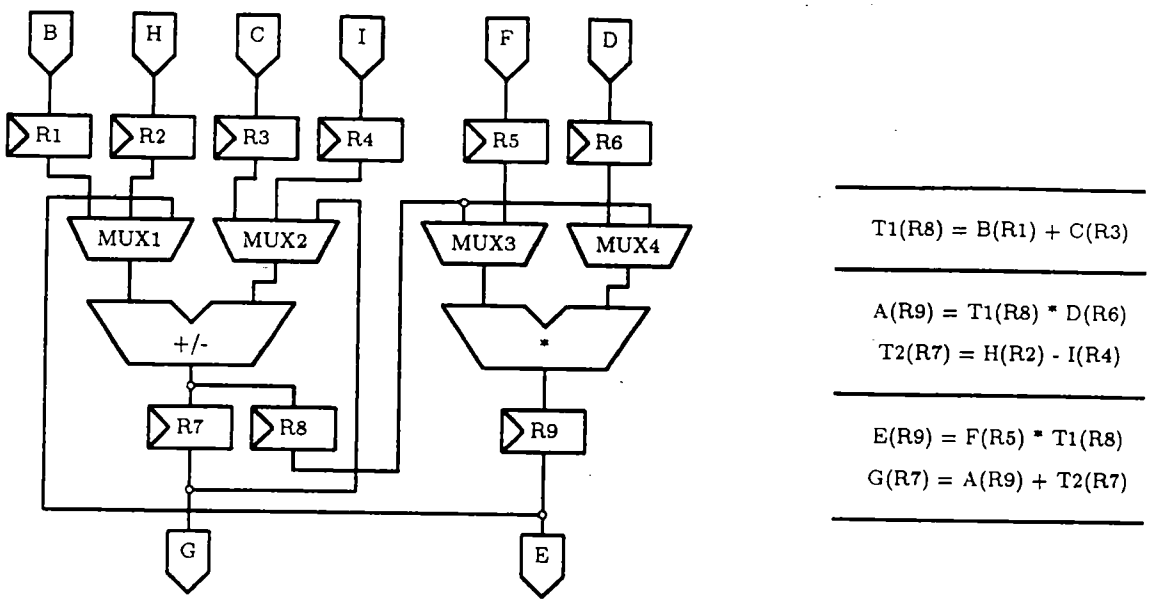
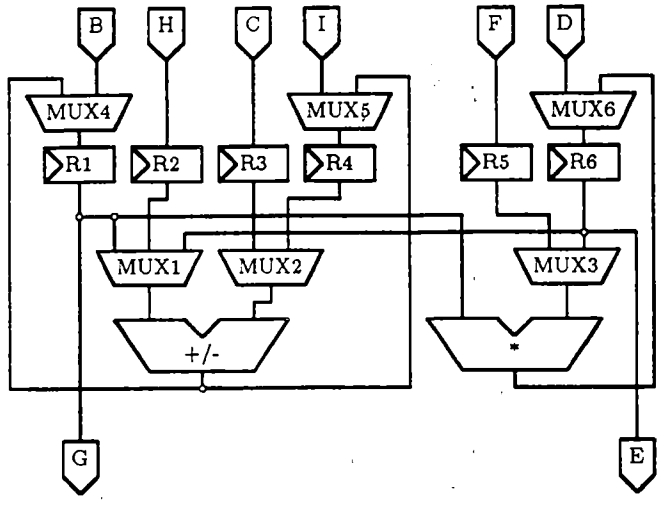


Figure 2.7: Resource Binding

one of several data inputs) or *bused* (equivalent to multiplexors at inputs and outputs, where one of several function unit or register outputs can be selected and transmitted on a wire or bus). The bus model allows greater interconnection sharing by creating a larger number of paths for the same number of connections. Connectivity binding can take advantage of the commutativity of operators by performing operand exchanges which will minimize interconnect.

Figure 2.7 shows one possible resource binding for the example using the resource constrained schedule. The number of registers in this example (9) can be reduced through register sharing since the minimum number of registers required is equal to the maximum number of variables to be stored across any state boundary in the

schedule of Figure 2.6 (6). Also, an operand exchange at the multiplier inputs will result in an interconnect savings. A resource binding with these modifications is shown in Figure 2.8.



T1(R1) = B(R1) + C(R3)

A(R6) = T1(R1) * D(R6)
T2(R4) = H(R2) - I(R4)

E(R6) = F(R5) * T1(R1)
G(R1) = A(R6) + T2(R4)

Figure 2.8: Register Merging and Operand Exchange

Notice that by sharing registers, additional interconnect is required (six multiplexors versus four in the previous design). Cost functions trading off storage element area for interconnect will guide the high-level synthesis system when making such resource binding decisions.

2.1.6 Controller synthesis

The results of state assignment must be captured in some format so that a controller which sequences the data path as required can be generated. In essence,

this description is a behavioral description for the controller. One common format for this specification is a set of boolean equations specifying each required control signal. The controller description can be oriented toward the realization of some specific control logic implementation such as random logic gates, a programmable logic array (PLA), or a microprogram sequencer. This specification can be input to logic synthesis systems such as those described in the next subsection, or a finite state machine compiler [Kin87] to produce the hardware which implements the controller.

Current State	Condition	Next State	Ops
1	True	2	MUX1_SELECT = INPUT0, MUX2_SELECT = INPUT0 ALU_OP = ADD MUX4_SELECT = INPUT0 R1_LOAD = 1
2	True	3	MUX3_SELECT = INPUT1, MUX6_SELECT = INPUT1 R6_LOAD = 1 MUX1_SELECT = INPUT1, MUX2_SELECT = INPUT1 ALU_OP = SUB MUX5_SELECT = INPUT1 R4_LOAD = 1
3	True	-	MUX3_SELECT = INPUT0, MUX6_SELECT = INPUT1 R6_LOAD = 1 MUX1_SELECT = INPUT2, MUX2_SELECT = INPUT1 ALU_OP = ADD MUX4_SELECT = INPUT0 R1_LOAD = 1

Figure 2.9: Symbolic Microcode

A more abstract description of the control unit can be captured in the form of **symbolic microcode** or a **state table** [DHG89, Har87]. For each machine state,

one or more triplets specify actions to be performed. Each triplet consists of a *condition* under which operations are performed, a *next state* transition, and a *set of operations*. Figure 2.9 illustrates the state table which controls and sequences the design of Figure 2.8 to perform the desired function.

2.1.7 Logic and Layout Synthesis

The register-transfer design produced by high-level synthesis is passed to lower level design tools which perform further optimizations and technology mapping and eventually produce an implementation in silicon. Logic synthesis systems such as MILO [VG88], MIS [BRSVA87] and SOCRATES [GBdH86] transform a functionally correct design consisting of generic components into one that has been optimized to meet a designer's constraints for a given component library.

Logic synthesis employs two techniques: *refinement* and *optimization* [VG88]. Refinement involves transforming the input description into an initial design; the input description (usually in the form of boolean equations) is minimized using algebraic techniques. *Technology mapping* maps the minimized equations to a technology-specific design at the gate level consisting of logic gate and flipflop components. Optimization transforms the initial design into one that meets some set of constraints (time, area, power). *Critical paths* (the longest path from input to output in a circuit, or a path which has a critical time constraint) are examined and optimized using strategies which make tradeoffs to meet the specified constraints.

Layout synthesis produces silicon layout from the gate level descriptions generated by logic synthesis. Automatic layout consists of two primary tasks: determining

the position of components on the layout surface, called *placement*, and interconnecting the components with wiring, a problem termed *routing* [PL88]. Components at the layout level can be *standard cells* (predesigned blocks of silicon which implement a logic function), elements of a *gate array* (a regular array of simple logic gates), or *custom layout* (cells designed by hand or by generators which tailor the cells to supplied parameters [LG87]).

2.2 Modeling Methodology

Ideally, the purpose of behavioral modeling is to describe the functionality of a design while remaining independent of a particular implementation. The design model to which the design is to be mapped must first be selected. An appropriate modeling style which reflects this design model is then used to develop the behavioral model. Once the designer's intent is captured in the behavioral description, the synthesis tool can use available methodologies to create a design targeted to the appropriate technology. The goal is that this model should not have to be changed significantly as the design is targeted to new implementations.

Figure 2.10 shows the typical use of modeling or specification in current design practice. Initial design descriptions are textual specifications which enumerate the desired features of the design to be implemented. This textual specification is often incomplete with respect to information necessary for the design process (especially for synthesis); for example, timing information at the level of detail necessary for automated synthesis is often omitted. A high level behavioral model may be written from the initial specification; however, this model is primarily used to verify the

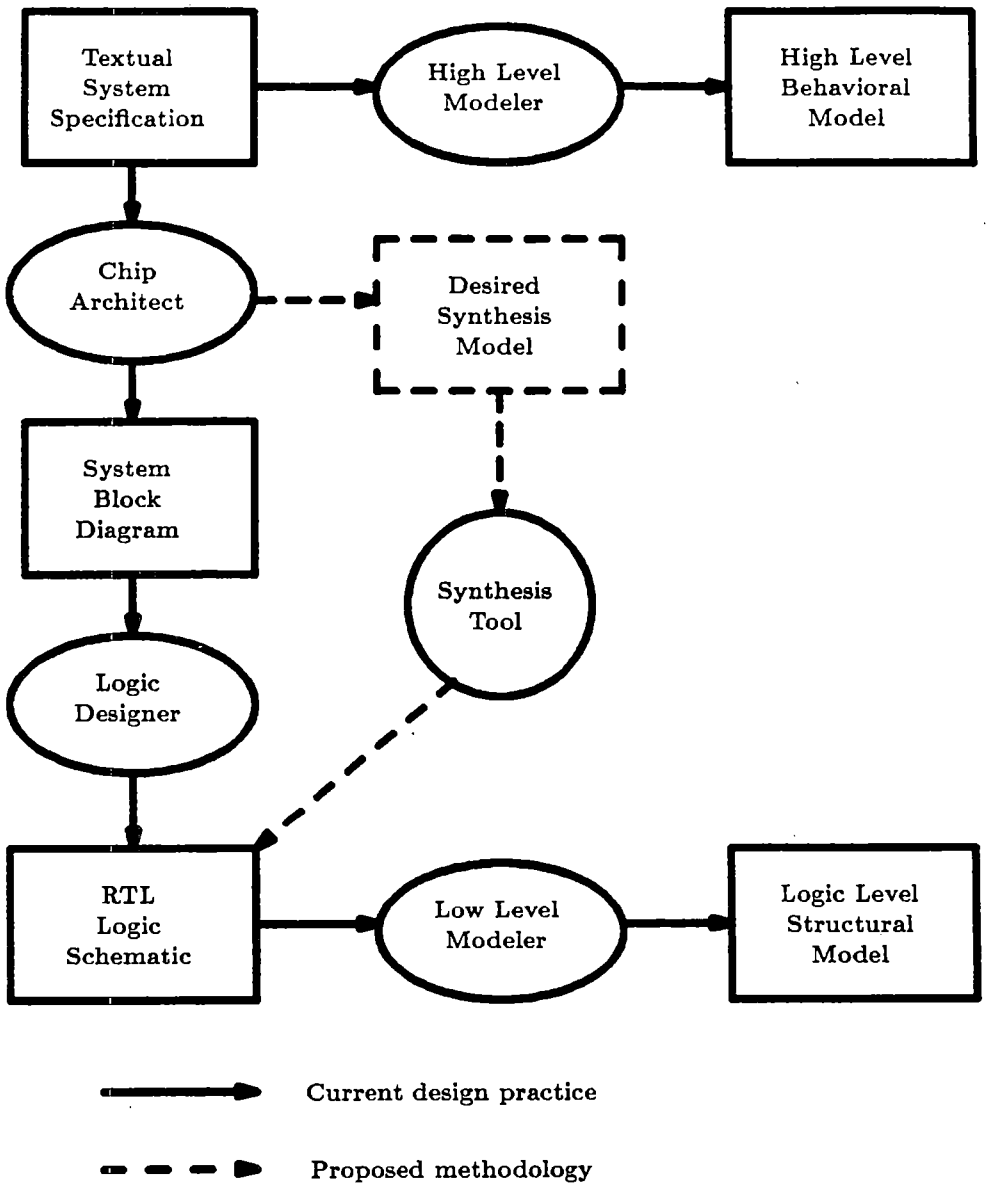


Figure 2.10: Typical Design Practice

conceptuality of the design through simulation and will not map directly to an efficient implementation.

The Chip Architect relies on his experience and expertise to interpret the specification and provide the missing information necessary to produce a logic or register-transfer level design. The initial specification is refined and partitioned to produce a System Block Diagram. This refined textual description and top level schematic are then passed on to the Logic Designer who implements the design using components from a selected component library. Researchers in behavioral synthesis have been working over the past two decades to capture and encode the design knowledge employed by the Chip Architect and Logic Designer in tools which attempt to automate this design process, or at least allow the designer to evaluate several alternative designs without having to manually design each one.

Once an initial design at the RTL or logic level has been completed, a low level behavioral model which reflects the structure and organization of the logic schematic is then developed to aid in the verification of the design. This model is inappropriate as input to behavioral synthesis since most of the design decisions have already been made.

Herein lies another major issue in synthesis from a behavioral specification: there currently exists no standard methodology for development of a synthesizable behavioral model. The focus of the high level modeler is to develop a functionally correct simulation model without concern for how easily that model can be synthesized. The low level modeler extracts the logic model from an already completed design. A level of modeling is missing which captures the desired functionality of a design using a modeling style which can be efficiently mapped to hardware. Given this model, a

designer can use a synthesis tool which will appropriately interpret the model and aid in the production of several design alternatives which approach human quality. The focus of the design effort can then be shifted from interpretation of the design specification requirements to improvement of portions of the design which require specialized human design knowledge that has not been sufficiently captured in the synthesis tool.

2.3 Definition of Design Models

Every high-level synthesis system assumes an underlying design model or *target architecture* for the synthesized structure. This section defines several commonly used design models and describes the operation of each model.

2.3.1 Combinational Logic Model

The design model for *combinational logic* consists of a network of logic gates. In this model, concurrent evaluation of all signal values is assumed. It is often desirable to specify point to point timing constraints in this model for optimization purposes (for example, critical path timing constraints). The most common method used to describe combinational logic designs is boolean equations.

2.3.2 Functional Model

The *functional* design model consists of combinational logic as well as storage elements (registers, counters). It may include a mixture of synchronous and asynchronous events which trigger operations in the data path (for example, the loading of storage elements). It cannot be guaranteed that these events are mutually exclusive; an asynchronous event such as a register reset can occur concurrently with a synchronous load of the same register. The functional design can be described in VHDL using block and process statements. Thus, this design model represents a functional partitioning of the design into one or more functional blocks. The complete operation of a hardware component can be described under the effects of several events in one functional block. Alternatively, the effect of each event can be described individually in a functional block, resulting in a distributed description of a component's exclusive functions across functional blocks.

2.3.3 Register Transfer Model

Register transfer descriptions involve the specification of operations to be performed within a processor for each machine state of a design. A common method for describing this behavior uses a **state table** [DHG89]. This model describes the designs using a temporal partitioning, rather than a structural or functional partitioning.

For each state, one or more triplets specify actions to be performed. Each triplet is composed of a *condition*, a *next state* specification, and a set of *operations*. The condition tests a boolean expression. Within each state, one or more conditions may

evaluate to true. The actions corresponding to each true condition are performed in the state. If the result of the test is true, a specified set of operations or register transfers is performed. Finally, control is transferred to the specified next state upon completion of the current state operations.

2.3.4 Control Unit/Data Path Model

The design model to which most behavioral synthesis tools map their designs is the *control unit/data path* model shown in Figure 2.11 [PL88].

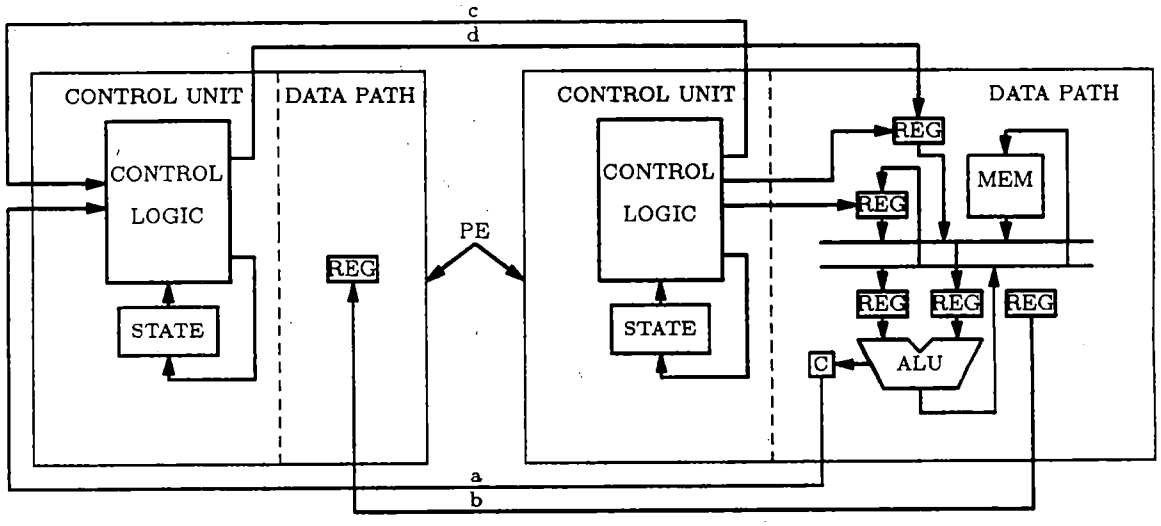


Figure 2.11: Control Unit/Data Path Design Model

A design in this model is composed of communicating processing elements (PEs). Each PE consists of a Control Unit (CU) and Data Path (DP). Because a behavioral

description may require one or several machine cycles (states) to execute the desired function, the microarchitecture implementation uses the DP to perform computations and the CU to sequence the machine through the necessary states and control the operations performed in the DP for each state. The CU contains a state register for storing the current state of the machine and control logic which controls the DP and communicates with other PEs. The DP consists of storage elements (registers, counters, memories) and functional units (ALUs, counters, shifters) and interconnect units (multiplexers, buses).

Access to registers, units or I/O ports is controlled by the CU. If several buses are used as sources to a storage or functional unit, a selector controlled by the CU must be added to the input. Some DP models use only point-to-point connection with selectors only and no buses. Processes also communicate via global signals. PEs communicate through DP ports to the CU or DP (nets *a* and *b* in Figure 2.11) or through CU ports to the CU or DP (nets *c* and *d*).

Note that in this model, an adder may be represented as a PE with no CU but with a DP (having one output port, two input ports, and no storage elements). Similarly, a flip-flop can be modeled as a DP with no functional units or as a CU with no DP and no control logic. Thus, this model is complete in the sense that it can model any synchronous digital system.

The composition of the data path is another feature which varies among high-level synthesis systems. Some systems target the design to a *fixed architecture* which consists of a standard microprocessor or a standard data path organization. For example, a strip architecture convention may be used where functional units are placed between two buses which are used for inter-component communication.

2.4 Hardware Description Languages

There are two choices for selecting a language for behavioral modeling: either an existing language can be used, or a language can be created to suit the particular application. The advantage of using an existing language is that there are often support utilities such as compilers, syntax parsers or simulators which aid in the verification of descriptions written in the language. A particular semantics is often associated with these languages in the context of their primary use; for example, a language used primarily for simulation will associate a meaning with each language construct that affects the execution of the underlying simulator. The disadvantage of using a language for a purpose other than its primary application is that there is often a mismatch between the available language features and the desired hardware attributes to be modeled. Certain language features may have no hardware realization; conversely, it may be difficult or impossible to model a particular hardware attribute given a fixed set of language constructs.

The alternative of creating one's own synthesis language alleviates the problem of modeling attributes of the selected design model(s). The language can be tailored to the explicit needs of the synthesis or simulation application. If the language has an unambiguous syntax and semantics, behavioral models written in the language would serve as important documents of the design decisions made and the conceptual operation of the design throughout its lifetime. However, the problem of standardization and portability of models written in a variety of non-standard languages presents several difficulties. First, additional effort would have to be expended in building verification tools or in translating this description into other languages which have

such support. In addition, the ability to transmit design information between independently developed tools will be lost if a standard interchange format is not used.

A consequence of using high level languages for behavioral modeling is that at this higher level of abstraction, it is possible to describe the same functionality using several language style or construct alternatives. This presents a fundamental problem for the synthesis tool – recognition of this equivalent functionality which should be mapped to the same hardware implementation. Current approaches to behavioral synthesis often restrict the use of the language via subsets in order to avoid the difficulties of equivalent descriptions.

2.5 Design Representation

Behavioral synthesis cannot be accomplished by a straightforward mapping of language constructs to RTL components. Information necessary for synthesis must be extracted from the input specification and organized in a representation which can be interpreted and transformed by algorithms within a synthesis tool. The design representation stores the status of the partial design as it is constructed and modified by the synthesis subtasks. It is possible that several versions and alternatives for the design will need to be maintained, and the hierarchy of the design has to be managed.

The design representation maintains the following views of the design: *behavior*, *structure* and *control*. The synthesis process begins with a representation of the input behavior (in the form of a flow graph, for instance) and completes by producing a *partial design* which consists of a structural netlist (a graph of interconnected components) and a control specification. At intermediate steps in the synthesis process,

these views of the design exist concurrently and are interrelated. There is a need to maintain behavior-to-structure links; for example, the resource binder needs to know which variables have been previously bound to a register to determine if the current variable can be bound to that register. The questions of unified versus orthogonal design representations of behavior, structure and control [CT88], as well as the degree of linkage between behavioral and structural views [BTK88] is an active research topic in high level synthesis.

Chapter 3

Previous Work

This chapter surveys previous work in the areas of behavioral modeling, its application to synthesis, and in particular, the use of VHDL for these purposes.

3.1 Armstrong's Process Graph Model

Armstrong [Arm88, Arm89] illustrates how VHDL can be used to model hardware at the various levels of abstraction. His work focuses on methods for representing various behavioral aspects of chip level modeling. At this level, a component is a complete VLSI chip such as a microprocessor, memory chip, or UART. The chip is modeled as a single entity (not constructed hierarchically from more basic primitives) which performs a sequence of micro-operations coded in an HDL. The model defines the input/output response of the device by specifying the algorithm the chip is to implement. Because logic signals flow in parallel, any hardware model must include a provision for concurrency of execution. The VHDL language handles this notion of simultaneity with the use of the process statement. Each process represents a block of logic, with all processes executing in parallel.

3.1.1 Design Representation

Armstrong defines a graph representation termed the *process model graph*. Nodes of the graph represent a partitioning of the functionality of the model into subfunctions. Arcs between nodes represent intercommunication between processes via signals. A timing specification may be associated with each arc indicating the delay associated with the transmission of a signal from one process to another. Figure 3.1 shows the general process graph model.

The process or subfunction represented by each node in the process model graph may be decomposed further according to functionality. For example, a node representing a register with synchronous load and asynchronous clear attributes can be modeled by two processes, one representing the effects of the load operation, the other reflecting the effects of the clear operation. In Figure 3.1, Process 3 is decomposed into three functional blocks: F1, F2 and F3.

3.1.2 Use of VHDL

Armstrong uses VHDL process statements to develop behavioral models for each process graph node. Signals appearing in the process sensitivity list are used to model timing delays and input/output or sequencing relationships between subfunctions. A VHDL process becomes activated on a change of value of any signal appearing in it's sensitivity list; thus, any change of a signal value produced by the behavioral description of a process graph node will cause the execution of the behavioral model for other process graph nodes which have this signal in their sensitivity list.

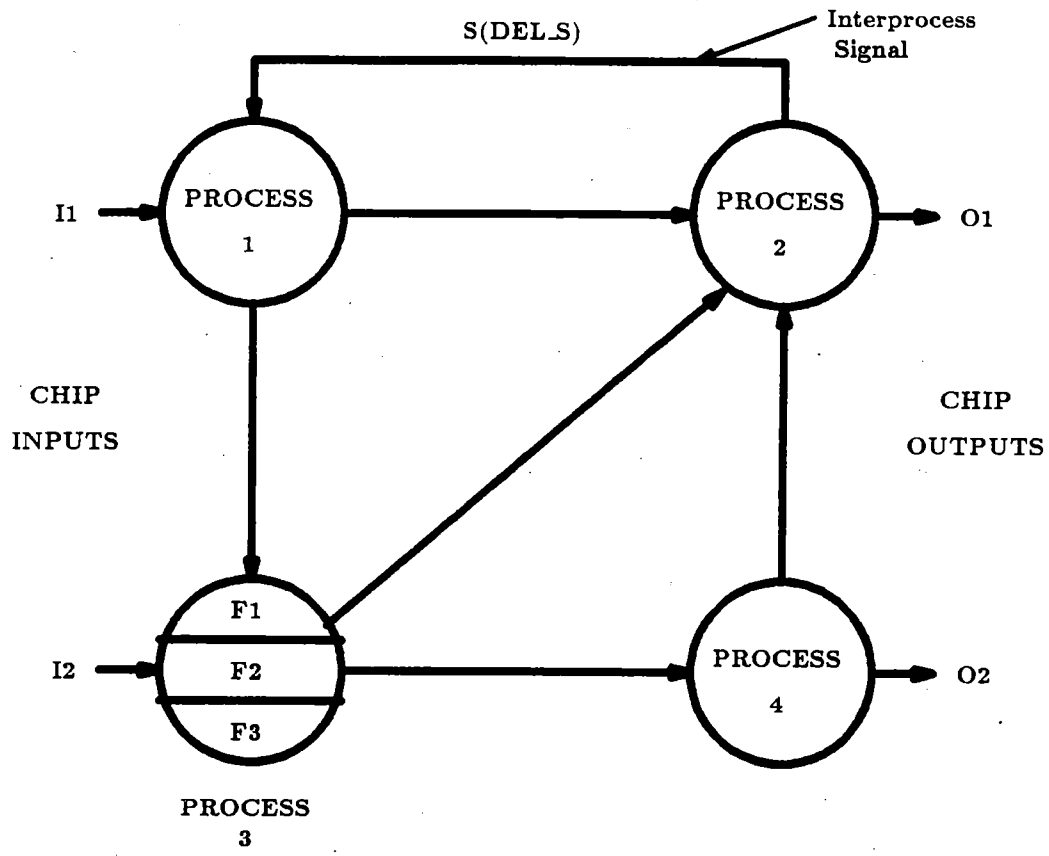


Figure 3.1: Armstrong's Process Model Graph

If a process graph node is functionally decomposed, each function of the node can be modeled using a process statement. Alternatively, since each signal assignment statement in VHDL can be considered a concurrent process, a process graph node function can also be modeled using the VHDL *guarded signal* construct. In this VHDL statement, a boolean expression or *block guard* which evaluates to TRUE enables the assignment of a data value to the signal; otherwise, no assignment is performed.

3.2 VSYNTH

3.2.1 Synthesis System

The VSYNTH system [Bha86, HKL89] provides a VHDL input interface to the existing MIMOLA Synthesis System (MSS) [Z+80]. It seeks to improve upon the drawbacks of the MIMOLA system: to remove the burden of decomposing the behavioral description into operations to be performed in individual control states, and to perform global data flow analysis which minimizes the required number of operators and storage elements. Figure 3.2 shows a block diagram of the VSYNTH system.

A *Process Graph Analyzer* accepts a VHDL behavioral input and generates a process graph by decomposing each statement and expression into a simple form (one operator and at most two operands). Compiler techniques (constant folding, local code optimization, code motion, common subexpression elimination) are used to optimize this description. The *Control State Generator* partitions the process graph into control states, introducing parallelism where possible. A reverse transformation from

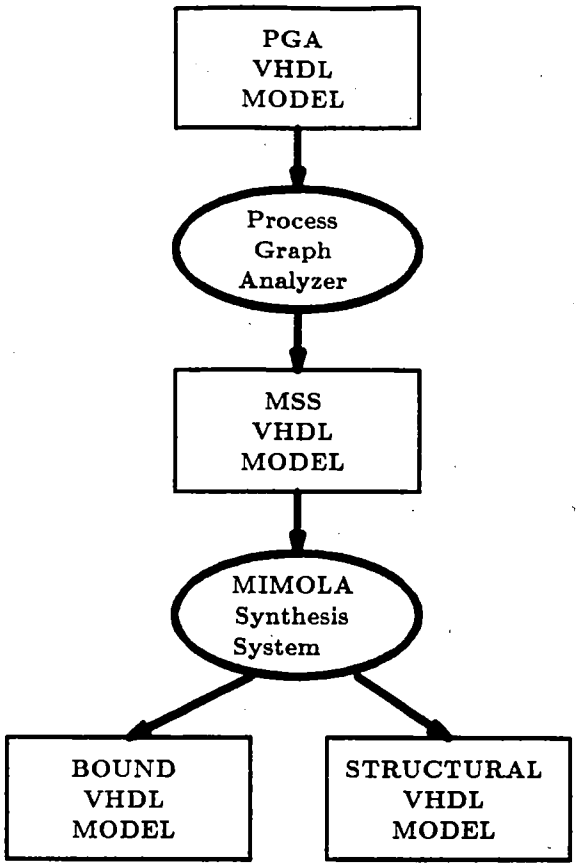


Figure 3.2: VSYNTH System Block Diagram

the process graph to legal VHDL syntax is performed by a translator. This description consists of a set of process statements, with each process describing operations to occur in a single control state.

The MIMOLA design system is intended to be an interactive design aid. To that end, the Design Representation is a database where the output of the Process Graph Analyzer is stored for designer interaction. The designer is allowed to modify the Design Representation by adding hardware bindings or constraints before presenting

the description to MSS for synthesis. When MSS is invoked, an implementation is generated by binding hardware components to operators and variables in the representation. A statistical analyzer provides information such as component utilization to aid the designer in determining constraints to meet design goals.

3.2.2 Use of VHDL

The VSYNTH system uses VHDL to represent four models of hardware, where *models* in this context refer to description styles. There are two input description styles, the *Process Graph Analyzer (PGA) Model* and the *MIMOLA Synthesis System (MSS) Model.* The output description styles include a *Structural Model* and a *Binding Model.*

The PGA Model uses a single VHDL process description with behavioral constructs (excluding wait statements). A restricted form of subprograms are allowed; either the procedure must be represented using a single control state data flow graph, or the multi-state subprogram must be in-line expanded into the PGA model. Variables declared local to the process are used to represent wire connections and therefore are not bound to storage elements. Signals declared within the architecture body of which the process is a part represent storage elements which retain their value across executions of the process.

An MSS Model is generated from the PGA model by the Process Graph Analyzer. The MSS Model describes the input behavior in terms of a finite state machine design model. This description consists of arbitrarily complex expressions whose arguments are storage devices (registers, memories). Several restrictions are placed on the use of VHDL to describe this model:

- At most one assignment to each storage element is permitted in each state
- A next state assignment to the variable RP is required in each state
- The finite state machine is represented using a single case statement within a single process statement.
- Only if, signal assignment and variable assignment constructs are allowed within the case alternatives. The if statement allows specification of control flow within the data path (i.e., assignment to the same variable or signal under exclusive conditions).
- As in the PGA model, signals represent wires, and variables represent storage elements.
- Attributes are used to specify component attributes (e.g., the functions to be performed by an ALU) and synthesis tool directives.

The Structural Model uses the VHDL structural description style to represent the netlist of interconnected components synthesized by the MIMOLA system. Use of the generic construct in VHDL allows for the specification of parameterizable templates for component classes such as multiplexors. This facilitates the use of a general model for components with similar functionality which differ only in attributes such as bit width or number of inputs. The control for the synthesized design is represented either as a hardwired control box with its behavior specified using VHDL signal assignments (boolean equations). An alternative representation of the control is a microstore component, where attributes of the control store are specified using the generic construct, and the contents of the store are specified in a constant declaration.

To indicate the binding of operations and variables in the MSS model to hardware generated by the MIMOLA system, the Binding Model replaces behavioral operators with VHDL functions whose name and attributes reflect component bindings. The function name is formed from the corresponding component name in the Structural Model. An *instantiation* parameter identifies the particular instance (in the event of multiple instantiations of the component). Other parameters include the function being performed by the component instance for this operation, and the component inputs.

3.3 IBM VHDL Design System

The IBM VHDL Design System [Sau87] consists of a collection of tools which use the VHDL language for hardware description, design management, simulation and synthesis. The tool set is built around the VHDL Version 7.2 language with language extensions added (e.g., memoried signal assignments, simulator test case control statements), some of which were a part of the IEEE Draft Standard that was evolving at this time. This system has been in production use within IBM. Research on the evaluation of VHDL for high-level synthesis has been conducted at the T.J. Watson Research Center [CT88, CST88].

3.3.1 Synthesis System

The Synthesis Subsystem [Sau87] performs register-transfer level synthesis of concurrent VHDL statements. This system maps VHDL operators in the description to primitive RTL elements (logic gates, ALUS, registers, multiplexors, etc.) which

are in the component library that can be operated on by subsequent logic synthesis tools. The design model to which this system is targeted is a Level Sensitive Scan Design (LSSD) methodology [EW77], a strategy which that allows test stimuli to be loaded into registers which are surrounded by combinational logic in order to facilitate observability and testability of potential faults in the circuit. VHDL memoried signal assignments (guarded signals in IEEE standard VHDL) are mapped to LSSD registers, while non-memoried signal assignments are mapped to combinatorial components. These components are technology independent; the generic design is later refined and mapped to technology specific components in the Logic Transformation System (LTS) [Ben83].

3.3.2 Use of VHDL

The use of VHDL for synthesis in the IBM VHDL Synthesis System is restricted to concurrent signal assignments. A restricted template form is used to specify a clocked LSSD register; each register update is modeled as a memoried signal assignment, where the block guard is the clocking signal.

Camposano et. al. [CST88] evaluate the feasibility of high-level synthesis from a behavioral, sequential description in VHDL. Their evaluation is based on the high-level synthesis design process associated with the Yorktown Silicon Compiler [BC+88].

3.3.3 Design Representation

The design representation proposed by Camposano and Tabet [CT88] for synthesis of behavioral VHDL consists of two models: a *hierarchy* model and a *behavior*

and structure model. The hierarchy model is a directed acyclic graph (DAG) which reflects the behavioral nesting of VHDL processes and procedures.

Because the behavioral and structural domains of both the data path and control portions of a design are interrelated, the behavior and structure design model presented consists of four graphs with links established between the graphs. The *data flow graph (DFG)* represents the operations and data dependencies present in the input behavioral description. A *control flow graph (CFG)* consists of nodes which represent the same operations found in the DFG; edges represent predecessor-successor relationships in the control sequencing of the operations rather than the input/output relationships of the DFG edges. The *data path graph (DPG)* consists of nodes which represent functional, storage and interconnect units and edges which reflect the interconnection of these units (i.e., a netlist). A *control automaton graph (CAG)* represents the state transition graph for the finite automaton to be implemented in the design, where nodes represent machine states and edges the state transitions.

Four types of links between the various behavior/structure graphs are created and manipulated by different synthesis tasks. The links between a CFG and DFG are derived from the explicit sequencing found in the VHDL input description. Scheduling involves the construction of the CAG and the association of CFG nodes to state nodes in the CAG. Resource binding introduces links between the DFG and DPG as behavioral operators and their input and output variables are assigned to hardware units; in addition, links between the CAG and DPG indicate function and data select signals that must be supplied to the data path during each state of the design's execution.

3.4 Physical Design using VHDL

Researchers at the University of Pittsburgh and Pennsylvania State University have cooperated in the development of a system which performs layout synthesis from a restricted form of VHDL behavioral models [LMOI89].

3.4.1 Synthesis System

The design system consists of three types of tools: *simulation*, *decomposition and transformation* and *translation*. Input to the system is in the form of VHDL text or a schematic graphical description. The output produced is a two dimensional gate matrix layout (in form compatible with the layout system MAGIC [SHMO86]).

3.4.2 Use of VHDL

The Pittsburgh/Penn State system currently processes only restricted VHDL models consisting of VHDL data flow and structural description styles. The purpose of restricting the use of VHDL to concurrent statements is to emphasize modeling the design at the structural level. For example, registers are specified at the gate level in terms of boolean equations (VHDL concurrent signal assignment statements). This fulfills their requirements of accurately and consistently mapping a VHDL construct to a primitive component at the level of physical design. Primitive component "behaviors" are instantiated as building blocks in a structured hierarchy. Work is in progress on the extension of this limited subset of VHDL toward the ability to synthesize more complex VHDL behavioral descriptions.

3.4.3 Design Representation

The VHDL input description is parsed into an *intermediate VHDL format (IVF)* which is used primarily for simulation. A second *gate level description language (GLUE)* is used for circuit descriptions within the physical design tools.

3.5 Summary

In summarizing the existing approaches to synthesis which use VHDL in various capacities, the following observations can be made:

1. *The efforts which use VHDL for the primary purpose of simulation utilize a modeling style which does not lend itself to synthesis.*

   While the style of VHDL description used for modeling, such as those developed by Armstrong, may correctly simulate the behavior of the hardware at the behavioral level, it presents several problems when viewed from the synthesis perspective. First, the separation of the description of a single component into several process statements complicates the task of collecting and identifying attributes to be associated with that component. Second, this description style relies on the VHDL simulator's notion of a container to assign the correct value to a signal at any given time based on one or more drivers. A container represents signal nets as well as registers, making the task of identifying these entities difficult for the compiler. Often, complicated language constructs are used to combine these drivers which result in a suboptimal design when mapped to logic components. Finally, the modeling practices such as those developed for Process Graph descriptions intermix signals and variables which are used to

sequence the model with those which are involved in data computations. The synthesis tool requires a clear distinction between data and control operations in order to produce a design of acceptable quality.

2. *The design models to which existing systems are targeted are limited.*
   Current systems tend to limit the applications or design models which can be processed by the tool to one of the design models mentioned in the previous chapter. The Pittsburgh/Penn State physical design system is currently targeted to a gate level (combinational) design model. The IBM VHDL Synthesis System uses VHDL for a limited functional design model. The adaptation of VHDL to the Yorktown Silicon Compiler system is targeted to VHDL behavioral descriptions only. The MIMOLA system has a specific design model to which it is targeted, namely, a synchronous finite state machine composed of a control unit and data path. Other commercial systems such as Synopsis or SilcSyn [BFR85] target VHDL process level descriptions to designs which fall more under the realm of logic synthesis (combinational and limited functional as in [Sau87] rather than high-level synthesis.

3. *VHDL has been attached as a front end description language to existing systems.*
   For example, the VSYNTH system adapts VHDL to the input and output requirements and models of the existing MIMOLA system. It does not take complete advantage of the language features. For features of the MIMOLA system and its design methodology where VHDL does not fit the systems needs appropriately, the descriptions are clumsy. The authors admit in [HKL89] that their use of VHDL functions to indicate synthesis bindings may not be the most straightforward way of representing such information.

4. *There is no well defined synthesis semantics for VHDL.*

   The work by Camposano et. al. analyzes the feasibility of attaching *some* synthesis semantics to each of the VHDL behavioral constructs. However, the published works stops short of presenting modeling situations in hardware design where these constructs could be used.

From these observed shortcomings of current efforts to synthesize from VHDL, the following goals of the research presented in this thesis can be stated:

- Identification of design models to be modeled using VHDL.
- Development of a set of guidelines for writing such models with the primary intent of synthesis.
- Definition and development of a framework which uses VHDL as the primary design description and interchange format and can accommodate a variety of design styles and tasks.
- Evaluation of the effects of modeling style on the quality of the synthesized design.

The remainder of this dissertation will present the details of the approach that has been taken to achieve these goals.

Chapter 4

Structured Modeling

This chapter describes a proposed modeling style for the use of the VHSIC Hardware Description Language (VHDL) in design synthesis. The operations and underlying assumptions of four design models currently understood and used in practice by designers are described. These design models include: *combinational logic*, *functional descriptions* (involving clocked components such as counters), *register transfer* (data path) descriptions, and *behavioral* (instruction set or processor) designs. We will illustrate the various uses of the VHDL description styles (*structural*, *dataflow* and *behavioral*) to represent characteristics of each of these design models. This chapter identifies how the VHDL language can be used for synthesis in VSS. Through the use of signal typing and attribute annotations, it will be shown how a VHDL description for simulation can be enhanced to provide necessary information for synthesis. The structural, dataflow and behavioral description styles of VHDL will be investigated. Emphasis is placed on how VHDL constructs should be used in order to synthesize optimal designs.

4.1 VHDL

4.1.1 Introduction

VHDL [IEE87] is the IEEE standard language for hardware description. However, the VHDL language does not guarantee uniqueness of descriptions; designs can be described in several ways and at several different levels of abstraction. The process of defining the conventions used to create these different descriptions is called *modeling.* Unfortunately, models perfectly suitable for one application can be unsuitable for another.

VHDL can be used in three basic application areas: simulation, fault modeling and test generation, and synthesis and silicon compilation. Each application area requires a different modeling style which satisfies the particular goals of of the application.

The goal of simulation is to validate the correctness of the description by measuring output response to input stimuli. Thus, generation of correct values on all signal lines over time is the most important goal. A secondary goal is the efficiency of the simulation; examining only the parts of the design affected by changes in the input values reduces the complexity and run time of the simulation. A high level (algorithmic or process-level) description is preferable for this application since such a model captures the high-level functionality.

In fault modeling, a fault is injected into the model. This fault is then sensitized, and its effects are propagated to an observable output in the description. Sensitization and propagation involves tracing data paths through the description. Consequently,

a structural or dataflow description is better suited to this application since it more closely reflects the structure of the hardware to be tested.

For synthesis, the primary objective is to process an algorithmic description in order to generate a structural description of components from a given library. Here, emphasis is placed on the proper connection of pins on components to implement the desired functionality.

4.1.2 Description Styles

VHDL provides three description styles: *structural*, *dataflow*, and *behavioral*. The structural description consists of component declarations, interconnect signal declarations, and component instantiations with port maps. This description style is suitable for describing a captured schematic after a design is completed, and it should be used to describe the design generated by a behavioral synthesis tool.

The dataflow description style is not as closely tied to the actual structural implementation of the design. This description style allows for the specification of concurrent events (data transformations and register transfers) under the control of synchronous (clock) or asynchronous signals. It can be used for combinatorial or functional logic models. The synthesis tool must optimize the design for a given component library. In the case of functional logic, components and connections are shared in time. The machine states are already specified in the description using conventions of the modeling style such as one block statement per state.

Behavioral descriptions are void of any implementation detail. They specify output values in terms of input values over time using an abstract algorithm. The

statements execute sequentially in the order of their occurrence. A synthesis tool must allocate components, schedule operations into machine states, and interconnect components for these specifications.

4.1.3 Design Model

The underlying design model assumed for a VHDL description is the control unit/data path model that was described in section 2.3 (Figure 2.11).

The *design entity* is the primary hardware abstraction in VHDL. It represents a portion of the hardware design that has well-defined inputs and outputs and performs a well-defined function. A design entity may represent an entire system, a sub-system, a board, a chip, a macro-cell, a logic gate, or any level of abstraction in between. A *configuration* can be used to describe how design entities are put together to form a complete design as shown in Figure 4.1.

A design entity may be described in terms of a hierarchy of *blocks*, each of which represents a portion of the whole design. The top-level block in such a hierarchy is the design entity itself; such a block is an *external* block that resides in a library and may be used as a component of other designs. Nested blocks in the hierarchy are *internal* blocks, defined by **process** or **block** statements. A structural, dataflow or behavioral description style can be used to express the functionality of an internal block.

Successive decomposition of a design entity into components, and binding of those components to other design entities that may be decomposed in like manner,

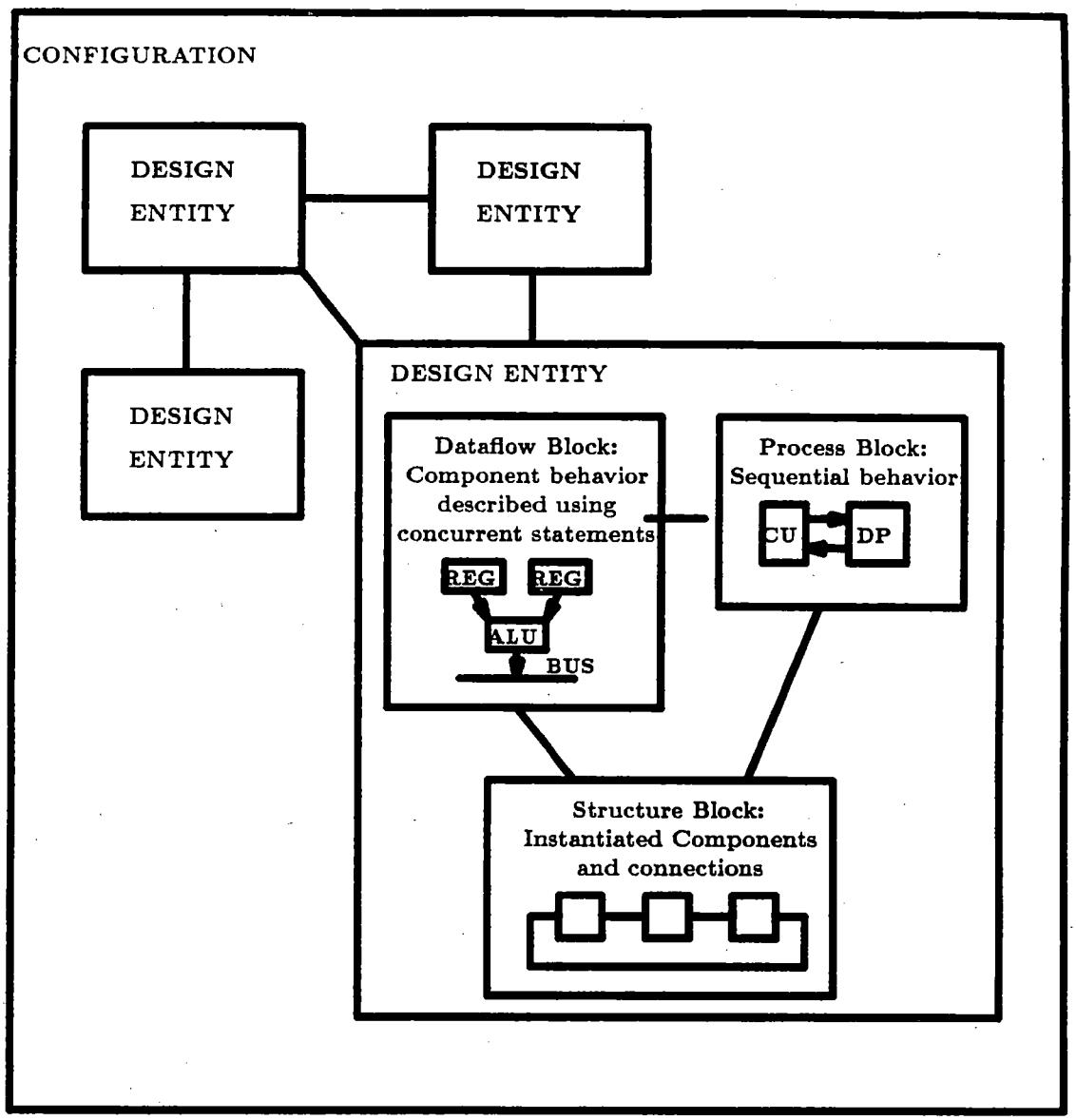


Figure 4.1: VHDL Design Hierarchy

results in a hierarchy of design entities representing a complete design. Such a collection of design entities is called a *design hierarchy.* The bindings necessary to identify a design hierarchy can be specified in a configuration of the top-level entity in the hierarchy. The design hierarchy concept is illustrated in Figure 4.1.

A VHDL description which represents such a design hierarchy is shown in Figure 4.2. Each design entity description is composed of two major sections: the *entity block* and the *architecture body.* The entity block contains the specification of external input/output port connections to the hardware to be designed.

The architecture body defines the body (structure and/or behavior) of a design entity. It specifies the relationships between inputs and outputs of the design entity, and may be expressed using a mixture of the three styles mentioned previously (structural, dataflow, behavioral).

4.1.4 Design Model Representation

The three description styles (behavioral, dataflow, structural) use concurrent statements to describe a portion of the complete design model shown above. Each concurrent statement in a VHDL description may be used to describe a piece (one or more components) of a design. Alternatively, more than one statement can be used to describe the functionality of the same design section if the behaviors are non-overlapping (exclusive).

The design sections represented by the concurrent statements communicate via global signals. These signals are defined in the declaration section of the architecture body. A global signal may be read (input) to several blocks or processes, but should

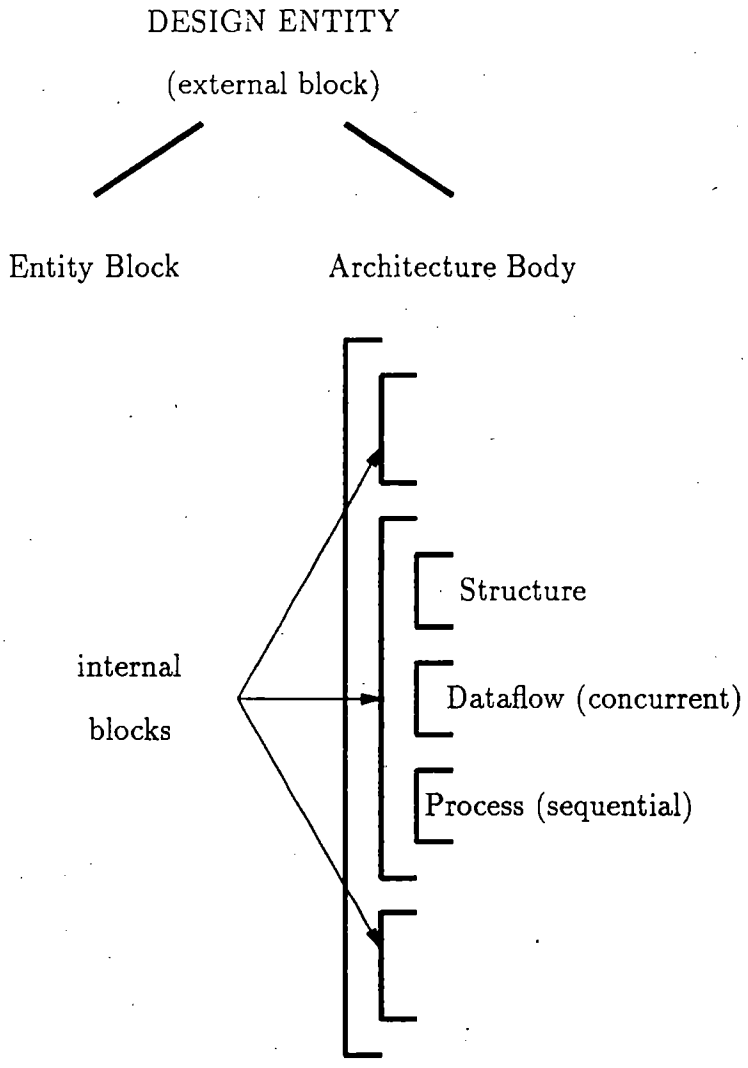


Figure 4.2: VHDL Design Entity Block Structure

be written to (updated by) only one block or process at any given time. In the event that it is desirable to have more than one active driver for a signal simultaneously (to model a bus, for example), a *resolution function* must be written and associated with the signal to determine its proper value for simulation.

Behavior

A VHDL description using the behavioral style consists of *process statements* and *concurrent procedure calls.* The most straightforward mapping of process statements representing behavior in algorithmic form to hardware is a microarchitecture implementation which uses the complete control unit/data path design model. Control constructs (IF, CASE and LOOP statements) are implemented via control unit sequencing. Variables within a process may represent storage components or interconnect wires. Local signals are used to communicate between the CU and DP. Assignment to variables occur in the order in which they appear in the specification, implying data dependencies between statements.

Interprocess communication follows these conventions:

1. The following subtypes are defined for descriptions to be used for synthesis:

   subtype data **is** BIT;
   subtype control **is** BIT;

   Signals of type data are used to interface with the data path. Signals of type control interface with the CU.

2. By default the following signal types/accesses are allowed:

   Input

   - signal/port reads within the data path description
   - conditional bit signals input to the descriptions of control logic

Output

- constant signals output from control logic (boolean, binary, integer)
- computed signals output from DP

Timing is expressed as a part of the output signal assignments. Data computations within the process are made with variable assignment statements.

Dataflow

Dataflow descriptions consist of *concurrent signal assignment statements.* They describe only the data path portion of the VHDL design model. The data path is a structure of components, where each component is described by one or more statements. Conditional signal assignments represent control embedded in the data path. The ordering of conditional clauses within these assignment statements indicates the priority of the events (as specified in the conditional expressions which selects the value to be assigned).

Structure

The VHDL structural design style utilizes *component instantiation* and *generate* statements. Here, the data path portion of the design model is described through the instantiation and interconnection of component primitives or previously defined design entities.

4.1.5 Mixture of VHDL Design Styles

This section illustrates a mixture of the VHDL structural, dataflow and behavioral description styles in a single description. Figure 4.3 shows a block diagram for a controlled counter functional description adapted from [Arm89].

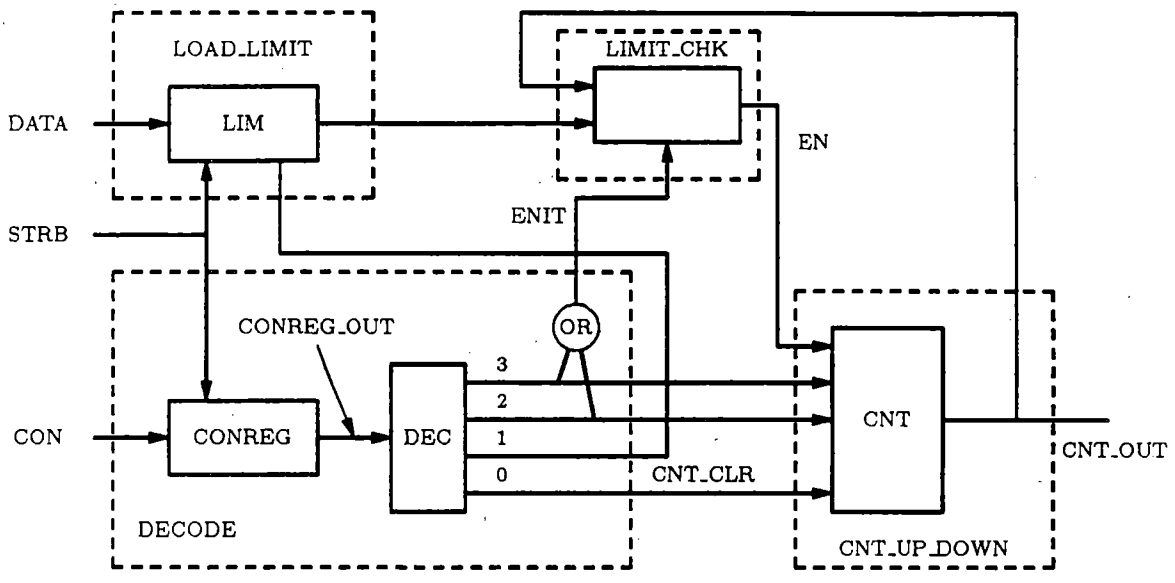


Figure 4.3: Controlled Counter Block Diagram

The operation of the controlled counter can be described as follows. On the rising edge of the STRB signal, an internal control register CONREG is loaded with the value on CON. The CONREG value is decoded to perform one of four functions: clear the counter, load a limit register, count up to a limit, or count down to a limit. The counter runs synchronously under an input clock, and the counting functions are

enabled by the internal signal EN. The DATA value is loaded into the limit register LIM on the falling edge of STRB if the control register contains the value '00'.

The VHDL description is shown in Figure 4.4.

This description consists of four concurrent statements, each of which describes a portion of the design: the decoding of the CONREG value, the loading of the limit register (LIM), the asynchronous clear and the synchronous up/down count of the counter (CTR), and a limit test.

The DECODE block statement describes the functionality of more than one functional block (the CONREG register and the decoder). A structural description style is used which specifies component declarations, interconnect signal declarations, component instantiations, and component interconnection (via the *port map* clause of the component instantiation statement).

The VHDL dataflow description style is used for the description of blocks LOAD_LIMIT and CNT_UP_OR_DOWN. The block guard is used to enable an update of the LIM and CNT register values. Note that these descriptions do not explicitly specify the structure of the components to be used in the implementation. However, the format of the guarded and conditional signal assignment statements suggest a mapping to storage elements (registers, counters) under conditional control.

The LIMIT_CHK block is described behaviorally with a process statement. This particular description represents a conditional signal assignment to the EN signal modeled using a behavioral IF statement.

```
entity CONTROLLED_CTR is
   port (
      CLK,STRB: in BIT;
      CON: in BIT_VECTOR(1 downto 0);
      DATA: in BIT_VECTOR(3 downto 0);
      CNT_OUT: out BIT_VECTOR(3 downto 0));
end CONTROLLED_CTR;

architecture MIXED of
   CONTROLLED_CTR  is

   subtype nibble is BIT_VECTOR(3 downto 0);
   signal CONSIG: nibble := B"0000";
   signal LIM: nibble register := B"0000";
   signal ENIT: BIT := '0';
   signal EN: BIT := '0';
   signal CNT: nibble  register := B"0000";
   signal CNT_CLR: BIT;

begin

DECODE: block (STRB = '1')

component register
   port (D: in BIT_VECTOR(1 downto 0);
         CLK: in BIT;
         Q: out BIT_VECTOR(1 downto 0));
end component;
component decoder
   port (D: in BIT_VECTOR(1 downto 0);
         Q: out BIT_VECTOR(3 downto 0));
end component;
component or2
   port (A,B: in BIT;
         O: out BIT);
end component;
signal CONREG_OUT: BIT_VECTOR(1 downto 0);

begin

CONREG: register
   port map (CON,CLK, CONREG_OUT);
DEC: decoder
   port map (CONREG_OUT,CONSIG);

OR_1: or2
   port map (CONSIG(2),CONSIG(3),
      ENIT);
   CNT_CLR <= CONSIG(0);
end block} DECODE;

LOAD_LIMIT: block (CONSIG(1)='1'
   and STRB='0' and not STRB'STABLE)
begin

   LIM <= guarded DATA after 10 ns;

end block LOAD_LIMIT;

CNT_UP_DOWN: block ((CLK = '1' and
   not CLK'STABLE) or (CNT_CLR = '1'))
begin
   CNT <= guarded
      B"0000" after 5 ns
         when CNT_CLR = '1' else
      CNT when EN = '0' else
      CNT + B"0001" after 12 ns
         when CONSIG(2) = '1' else
      CNT - B"0001" after 12 ns
         when CONSIG(3) = '1' else
      CNT;

end block CNT_UP_DOWN;

LIMIT_CHK: process (ENIT,CNT)
begin
  if ((CNT /= LIM) and (ENIT = '1'))
   then
      EN <= '1' after 12 ns;
   else
      EN <= '0' after 5 ns;
   end if;

end process LIMIT_CHK;

CNT_OUT <= CNT;

end MIXED;
```

Figure 4.4: VHDL Description of Controlled Counter

4.2 Problems for Synthesis Posed by VHDL

The VHDL language provides the designer with a powerful description language with many alternative ways to model the same functionality. When viewed from a synthesis perspective, this presents several problems, including:

- Identification of storage elements and signals
- Language constructs with no hardware realization
- Collection and identification of component attributes
- Specification of asynchronous events
- Use of multiple blocks/processes to describe one component
- Functional versus temporal partitioning of the design functionality
- Processing of slices of a bit vector quantity (e.g., the update of selected bits of a control word register)
- Hierarchical decomposition of the design into communicating processes using a mixture of description styles and design models

For example, Figure 4.5 shows a VHDL description which uses separate statements to model the asynchronous clear and synchronous up/down count of a controlled counter [Arm89].

The drivers (CNT1, CNT2) generated to represent the effects of each event on the register's output value are combined using a conditional signal assignment statement MUX1. Note that MUX1 is a virtual component which should have no hardware realization. The sole purpose of the statement is to collect the multiple drivers for simulation such that the value of OUT_TMP is properly updated.

```
Architecture PROCESS_IMPL of CONTROLLED_CTR is

    signal CLK,EN: BIT;
    signal CONSIG: BIT_VECTOR(0 to 3);
    signal OUT_TMP,CNT1,CNT2: BIT_VECTOR(0 to 3);

 CLEAR_CTR: block (CONSIG(0) = '1' and not CONSIG(0)'stable)
    begin
       CNT1 <= guarded "0000" after CLRDEL;
    end block CLEAR_CTR;

 CNT_UP_OR_DOWN: process (CLK,EN)

    variable CNT: BIT_VECTOR(0 to 3);
    variable CLKE: BOOLEAN;

    begin
       if EN'stable then
          if EN = '0' then
      CLKE := TRUE;
          else
      CLKE := FALSE;
          end if;
       end if;
       if (CLK = '1' and not CLK'stable and CLKE) then
          if (CONSIG(2) = '1') then
      CNT := INC(CNT);
          else if (CONSIG(3) = '1') then
      CNT := DEC(CNT);
   end if;
       end if;
       CNT2 <= CNT after CNTDEL;
    end process CNT_UP_OR_DOWN;

 MUX1: OUT_TMP <= CNT1 when not CNT1'quiet else
   CNT2;

 end block PROCESS_IMPL;
```

Figure 4.5: VHDL Controlled Counter Chip Model

Two approaches may be taken to translate this behavioral description into hardware: direct mapping of VHDL constructs to appropriate microarchitecture components, or recognition of certain VHDL construct patterns as a representation of a particular hardware concept. If a straightforward mapping of VHDL constructs is performed, inefficient hardware will often result as shown in Figure 4.6.

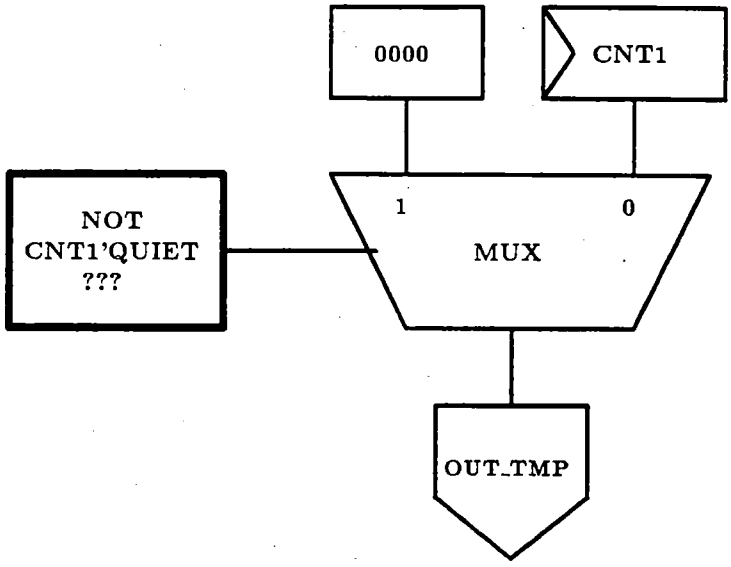


Figure 4.6: Virtual Multiplexor Problem

In the above example, an unnecessary multiplexor will be introduced when mapping the MUX1 statement to hardware, with each driver as a data input and complicated selection logic. A sophisticated logic critic would then be needed to transform this design into an optimal one (i.e., a register with up/down count and clear control inputs). The latter method of translation requires identification of the type of signals used to select the input driver. Since VHDL allows the designer to express the same functionality in many different ways, the task of developing a rule set which recognizes all valid VHDL representations of a desired set of hardware concepts

would be extremely difficult, if not impossible. The compilation process becomes simplified if the descriptions are not allowed to contain such virtual components.

4.3 Structured Modeling for Synthesis

4.3.1 Introduction

The quality of a design as well as the complexity of the synthesis process are directly related to the style of description chosen to represent a particular design model. Certain VHDL constructs or description styles are better suited to describe a particular design model than others. Because VHDL allows the designer several ways of describing the same functionality, it is important to set standard modeling practices for designers using VHDL. These standards should guarantee high quality of synthesized design, while divergence from the standard will result in a description that is simulatable, but a synthesized design that is not optimal.

The following sections describe the design models supported within the VSS system. For each model, the level of abstraction or type of input specification is identified. The VHDL modeling practices for each model are then presented.

4.3.2 Combinational Logic

Design Model

The design model for combinational logic consists of a network of logic gates. The most common method used to describe combinational logic designs is boolean

equations. In this model, concurrent evaluation of all signal values is assumed. A boolean equation representation facilitates synthesis tasks such as algebraic minimization (e.g., MIS [BRSVA87]) or optimization (e.g., SilcSyn [BFR85]).

A combinational logic design involves path delays through the interconnected components. When specifying timing constraints, the combinational logic model should be able to express input to output timing for critical path constraints. These constraints guide the synthesis tool in selecting the appropriate components when tradeoffs are possible. In some instances, the designer may wish to specify more detailed timing constraints on particular operators or paths between some internal points in the design.

VHDL Alternatives

One alternative in VHDL for expressing the combinational logic model is a dataflow description. The combinational circuit can be represented as a set of boolean equations in the form of concurrent assignment statements. Figure 4.7(a) illustrates a dataflow description of a full adder.

The dataflow description offers the following advantages:

1. The description style would be familiar to designers who generally think of design at this level in terms of boolean equations.
2. The description is readable - a straightforward mapping exists between operators and logic components.
3. In performing synthesis, the description is easily translatable to netlist format (either EDIF or structural VHDL, for example).

```
entity FULL_ADDER is
   port (X,Y: in BIT;
         CIN: in BIT;
         SUM: out BIT;
         COUT: out BIT);
end FULL_ADDER;
 range 0 to 3 := 0;

architecture DATA_FLOW_IMPL of
   FULL_ADDER is
-- local signal declarations
   signal S1,S2,S3: BIT;
begin
   S1 <= X xor Y;
   SUM <= S1 xor CIN after 3 ns;
   S2 <= X and Y;
   S3 <= S1 and CIN;
   COUT <= S2 or S3 after 5 ns;
end DATA_FLOW_IMPL;
```

(a) Dataflow Description

```
architecture BEHAVIORAL_IMPL
   of FULL_ADDER is
begin
   process (X,Y,CIN)
      variable S: BIT_VECTOR(1 to 3);
      variable NUM,I: INTEGER

   begin
      S := X & Y & CIN;
      for I := 1 to 3 loop
         if (S(I) = '1') then
            Num := Num + 1;
         end if;
      end loop;
      case Num is
         when 0 => COUT <= '0'; SUM <= '0';
         when 1 => COUT <= '0'; SUM <= '1';
         when 2 => COUT <= '1'; SUM <= '0';
         when 3 => COUT <= '1'; SUM <= '1';
      end case;
   end process;
end BEHAVIORAL_IMPL;
```

(b) Behavioral Description

Figure 4.7: VHDL Full Adder Descriptions

Note that timing information is associated with output signal assignments only. If the VHDL description is to remain correct for simulation, timing constraints cannot be specified for internal signals using the after clause mechanism. This is due to the fact that all concurrent assignment statements have their drivers evaluated at the current simulation time using the current value of all signals. Thus, a new value for an internal signal which becomes effective after some delay will not contribute to the computation of a new output value (evaluated at the current simulation time) which depends on it.

An alternative way to describe the functionality of combinational logic is an algorithmic description as shown in the example of the full adder in Figure 4.7(b)

While expressing the same behavior as the dataflow description, the algorithmic description has the following deficiencies:

- The algorithmic description is not the natural way to think of logic. Operators manipulate variables (integers) with extended ranges (number representations) other than boolean. The algorithm requires manipulations of index and other variables. Type conversions from bit quantities to integer and back to perform a counting operation clutter the description and contribute to a suboptimal design generated by the synthesis tool.
- Synthesis yields inefficiencies. When the VHDL algorithmic description is used as input for synthesis, the logic that is designed will initially contain some unnecessary hardware. This results from the translation of language constructs associated with simulator efficiency such as the type conversions mentioned above, or control constructs such as loops which were meant to represent replication of a design section. Additional effort must be spent in the synthesis process to recognize inefficiencies in the design. Some of the inefficiency may never be removed because of costly global optimization.

The following modeling practices for combinational logic are recommended:

Proposition 1

Use the dataflow model for synthesis of combinational logic.

Proposition 2

Use an **after** clause only for assignments made to output signals. This delay represents the maximum allowed delay from any input to the next particular output, and it will be used as a constraint during synthesis.

4.3.3 Functional Model

Design Model

The functional design model consists of combinational logic as well as storage elements (registers, counters). It may include a mixture of synchronous and asynchronous events for loading storage elements. An event is defined as the transition of a clock or any other signal. It cannot be guaranteed that these events are mutually exclusive; an asynchronous event such as a register reset can occur concurrently with a synchronous load of the same register.

The design is a structure of functional blocks such as ALUs, shift registers, counters, comparators, memories and buses. Each block performs transformations on its inputs with or without latching or storing. Each block is a combinatorial function or a finite state machine (FSM) where the state is determined by the values in storage elements.

The controlled counter [Arm89] shown in Figure 4.3 is an example of such a design. On the rising edge of the STRB signal, an internal control register CONREG is loaded with the value on CON. The CONREG value is decoded to perform one of four functions: clear the counter, load a limit register, count up to a limit, or count down to a limit. The counter runs synchronously under an input clock, and the counting functions are enabled by the internal signal EN. The DATA value is loaded into the limit register LIM on the falling edge of STRB if the control register contains the value '00'.

VHDL Alternatives

The functional design can be described in VHDL using block or process statements. When modeling such a design, one or more functional blocks can be described with one block or process. The counting function of the counter in Figure 4.3 is described by the block in Figure 4.8(a). The same function is described by process statement in Figure 4.8(b).

```
CNT_UP_OR_DOWN: block (CLK = '1' and not CLK'STABLE)
begin
   CNT <= guarded
       CNT when EN = '0' else
       CNT + "0001" after INCDEL when CONSIG(2) = '1' else
       CNT - "0001" after INCDEL when CONSIG(3) = '1' else
       CNT;
end block CNT_UP_OR_DOWN;
```

(a) Block Statement Representation

```
CNT_UP_OR_DOWN: process (CLK,CONSIG(2),CONSIG(3),EN)
   variable CNT_REG: BIT_VECTOR(3 downto 0);
begin
   if (CLK = '1' and not CLK'STABLE) then
      if (EN = '1') then
 if (CONSIG(2) = '1') then
    CNT_REG := CNT_REG + "0001";
 elsif (CONSIG(3) = '1') then
    CNT_REG := CNT_REG - "0001";
         end if;
      end if;
   end if;
   CNT <= CNT_REG after INCDEL;
end process CNT_UP_OR_DOWN;
```

(b) Process Statement Representation

Figure 4.8: VHDL Functional Descriptions

Modeling each functional block with more than one process may become difficult if the description is to remain simulatable. If assignments are made to the same signal in multiple processes such that the signal may have multiple drivers, a *resolution function* is required to determine the appropriate value of the signal. The solution to this problem, proposed by Armstrong [Arm89], is to introduce a virtual multiplexor outside of both processes. This solution, although acceptable in simulation, is difficult to implement in real hardware. Thus, multiprocess modeling of the same functional block should not be used for synthesis.

Functional blocks can be described with more than one VHDL block statement. However, the behavior described in each block statement should be independent of other blocks. Examples of exclusive functions are the synchronous up counting and asynchronous reset of a synchronous up-counter with asynchronous reset. Furthermore, assignment to the same guarded signal under different guard expressions (representing different clocks) in different VHDL blocks should not be allowed. Although two guard expressions (i.e., two clocks) can be mutually exclusive, controlling selection of the input signals to the same register may generate timing hazards.

To achieve uniformity, the timing should be assigned only to output signals according to Proposition 2. For each functional block, the following four timing constraints can be used:

1. the clock cycle, specified with a VHDL attribute statement,
2. propagation delay from inputs to (clocked or asynchronously controlled) storage elements. Since this path can contain only combinational logic, a local signal can be defined to designate the storage element input data value. A timing

specification (either an attribute or possibly an after clause) can be used for a signal assignment to this local signal.

3. propagation delay from storage elements to outputs, and
4. propagation delay from inputs to outputs (in the case where there are no storage elements on the path from input to output).

In order to properly connect VHDL declared signals to components in the given library, all signals should be typed. The following five types should be defined: **clock, set, reset, test, data** and **control**. Typing will be used to identify the function of event signals appearing in in the block guards. The merging of assignments to the same variable in different blocks is possible during the synthesis process since signal types are known and synchronous/asynchronous behavior is clearly distinguished.

The following guidelines should be followed when developing a functional model description for synthesis:

Proposition 3

One or more functional blocks should be described by one VHDL block statement. Several block statements could be used to describe exclusive behavior (synchronous and asynchronous behavior of the same functional block).

Proposition 4

The guard expression should contain only signals of type clock, set or reset.

Proposition 5

All signals should be typed. Signal types should include clock, reset, set, test, data and control.

4.3.4 Register Transfer Model

Design Model

Register transfer descriptions involve the specification of operations to be performed within a PE (as shown in the design model of Figure 2.11) for each machine state of a design. For each state, one or more triplets specify actions to be performed. Each triplet is composed of a *condition*, a *next state* specification, and a set of *operations*. The condition tests a boolean expression. Within each state, one or more conditions may evaluate to true. The actions corresponding to each true condition are performed in the state. If the result of the test is true, a specified set of operations or register transfers is performed. Finally, control is transferred to the specified next state upon completion of the current state operations.

Figure 4.9 illustrates a simple example of a state table which specifies the conditional statement **if** X = 0 **then** A = A + 1 **else** B = A + B.

Timing in register transfer descriptions is dependent on two parameters: the clock cycle duration, and the maximum time required to perform all operations specified for any state. In this case, it is not necessary to supply timing information in the statements which represent register transfers. If the clock cycle is supplied by the user (using a VHDL attribute for the design entity), the synthesis system will attempt to select units which will perform the desired operations in each state within the specified clock cycle. If the clock cycle duration is not specified, the fastest components are selected from the available library, and the clock cycle duration is determined by the longest delay path in the design necessary to implement any state.

Current State	Condition	Next State	Ops
S0	True	S1	cond <= (X = 0);
S1	cond	S2	
	cond'	S3	
S2	True	S4	A <= A + 1;
S3	True	S4	B <= A + B;
S4			

Figure 4.9: Register Transfer State Table

In VHDL, block statements may be used to represent the state table using the following conventions:

1. Every block represents a different state.
2. The block guard specifies clock, while the body of the block sets the state variable to the appropriate next state and performs operations under the desired conditions.

Figure 4.10 shows the corresponding block description for the state table of Figure 4.9. This VHDL block representation allows for the expression of parallelism. Concurrent actions may be specified for a given condition within the block statement.

A second use of the block representation to describe the register transfer state table is shown in Figure 4.11.

```
clock_edge <= CLK = '1' and not CLK'STABLE;

State_0: block (clock_edge)
begin
   state <= guarded S1 when (state = S0) else state;
   cond <= (X = 0) when (state = S0) else cond;
end block State_0;

State_1: block (clock_edge)
begin
   state <= guarded
      S2 when (state = S1 and cond) else
      S3 when (state = S1 and not cond) else state;
end block State_1;

State_2: block (clock_edge)
begin
   state <= guarded S4 when (state = S2) else state;
   A <= guarded A + "0001" when (state = S2) else A;
end block State_2;

State_3: block (clock_edge)
begin
   state <= guarded S4 when (state = S3) else state;
   B <= guarded A + B when (state = S3) else B;
end block State_3;
```

Figure 4.10: State Table Block Description

This description separates the state transition portion of the description (associated with the control unit) from the register transfers to be performed in each state (data path operations). While this description simulates properly, it has one difficulty from the synthesis perspective: identification of the clock. Assignment to the state variable is made via guarded signal assignments in which the current state, rather than a common clock, is used. The time interval that elapses between changes in the state (the clock period) is modeled with the after clause. The data operations appearing in block statements are also clocked by the state. This description is difficult to synthesize since the clock for register assignments is not explicitly specified.

```
State_1: block (state = S0)
begin
   state <= guarded S1 after CLK_PERIOD;
end block;
State_2: block (state = S1 and cond)
begin
   state <= guarded S2 after CLK_PERIOD;
end block;
State_3: block (state = S1 and not cond)
begin
   state <= guarded S3 after CLK_PERIOD;
end block;
State_4: block (state = S2 or state = S3)
begin
   state <= guarded S4 after CLK_PERIOD;
end block;
```

(a) state transitions

```
F0: block (state = S0)
begin
   cond <= guarded (X = '0');
end block;
F2: block (state = S2)
begin
   A <= guarded A + "0001";
end block;
F2: block (state = S3)
begin
   B <= guarded A + B;
end block;
```

(b) data operations

Figure 4.11: State Transitions/Register Transfers Description

The description of the state table using the VHDL behavioral description style (process statement) is shown in Figure 4.12(a).

Here, each process represents a state. Problems associated with this representation with respect to synthesis include:

1. One signal variable per state is required. Since each process is triggered by a change in the state variable found in its sensitivity list, detection of this signal change and state decoding are difficult to implement.

2. The same storage element may need to be updated in more than one process. Using block statements, this can be handled with guarded signal assignments; the process, however, provides no clean method of expressing this concept. Variables are local to the process and can be used to represent a storage element within one process only. Guarded signal assignments are not allowed

```
architecture P1 of STATE_TBL is
   signal S0,S1,S2,S3,S4: BIT;
   signal S4_1,S4_2: BIT;
   signal A,A1,B1: BIT_VECTOR(3 downto 0);
begin
   State_0: process (S0)
   begin
      cond <= (X = 0);
      S1 <= not S1 after CLK_PERIOD;
   end process State_0;

   State_1: process (S1)
   begin
      if (cond) then
         S2 <= not S2 after CLK_PERIOD;
      else
         S3 <= not S3 after CLK_PERIOD;
      end if;
   end process State_1;
                end if;
   State_2: process (S2)
   begin
      A1 <= A + "0001";
      S4_1 <= not S4 after CLK_PERIOD;
   end process State_2;

   State_3: process (S3)
   begin
      B1 <= A + B;
      S4_2 <= not S4 after CLK_PERIOD;
   end process State_3;

   A <= A1 when not A1'QUIET else
        A;
   B <= B1 when not B1'QUIET else
B;
   S4 <= S4_1 when not S4_1'QUIET else
         S4_2 when not S4_2'QUIET else S4;
end P1;
```

(a) process graph description

```
architecture P2 of STATE_TBL is
   type STATE_VAL is (S0,S1,S2,
      S3,S4);signal cond: BOOLEAN;
   signal cond: BOOLEAN;
   signal state: STATE_VAL;
   signal new_state: STATE_VAL;
begin

process(state)

begin

      ...
      when S0 => cond <= (X = 0);
                 new_state <= S1;
      when S1 => if (cond) then
                    new_state <= S2;
                 else
                    new_state <= S3;

     when S2 => A := A + 1;
                new_state <= S4;
      when S3 => B := A + B;
                 new_state <= S4;
      when S4 => ...
   end case;

state <= new_state
            after CLK_PERIOD;

end process;

end P2;
```

(b) sigle process

Figure 4.12: Alternative VHDL State Table Descriptions

within processes. Virtual muxes must be added to accommodate the update of the same signal in more than one state. This introduces unnecessary hardware which violates good design practice.

A second use of the process statement to represent register transfers is shown in Figure 4.12(b). The single process contains a case statement to specify an instruction set like description. This description can't express parallelism for operations associated with one condition since the process is inherently sequential. On the other hand, if we assume for synthesis that all statements appearing within a case alternative are executed in parallel, the VHDL simulation of the input description will not reflect the true behavior of the synthesized design. The solution is to use additional signals of type wire. The following VHDL code fragment illustrates the equivalent sequential statements for the concurrent interchange of the values of A and B:

```
variable A,B: BIT;
signal temp: BIT;

temp <= A;
A := B;
B := A;
```

In order to describe register transfer designs for synthesis, the following modeling practice is recommended:

Proposition 6

Each state of a register transfer design should be described with block statements containing condition, next state assignment and all register transfers with the clock specified in the guard expression. Alternatively, a single process with a case statement can be used.

4.3.5 Behavioral Design

Design Model

The design model shown in Figure 2.11 is also assumed for the algorithmic design model. A behavioral description allows the designer to describe the design as a black box with well defined interfaces. Variables within a description can be allocated storage by default, or the synthesis system can determine which variables require storage. As in the combinational model, input to output timing is expressed.

VHDL Alternatives

Figure 4.13 shows a simple VHDL behavioral description. The process statement is the only suitable method in VHDL for expressing behavior in algorithmic form. Each VHDL process will be synthesized into a CU/DP pair. Data computations within the process are made with variable assignment statements. Its similarity to a programming language allows for the coding of algorithms using typical control constructs (IF, CASE, FOR and WHILE loops).

Input to output timing is expressed as a part of the output signal assignments. The wait statement can be used within the process statement to express timing. A statement of the form

wait until <condition>

will model a design state which loops on itself until the specified condition evaluates to TRUE. The state table entry for this state will advance the state register to the

```
architecture BEHAVIOR of STATE_TBL is
   signal B_port: BIT_VECTOR(3 downto 0);
begin

process (X)
   variable A,B: BIT_VECTOR(3 downto 0);
begin
   if (X = '0') then
      A := A + "0001";
   else
      B := B + A;
   end if;
   B_port <= B after 20 ns;
end process;
end block;
```

Figure 4.13: Behavioral Description Using VHDL Process Statement

next state in sequence when the condition is TRUE. The second form of the wait statement,

wait for <time>

models a design state which loops on itself for the specified time duration. For synthesis, the time duration must be a multiple of a known quantity of time such as a clock cycle. This model requires a count variable initially set to zero which is incremented on every execution of the state. When the count reaches the specified number of clock cycles, the state register is advanced to the next state.

The recommended modeling practice for algorithmic design can be summarized as follows:

Proposition 7

Behavioral designs are modeled by VHDL process statements. Signal assignments are used to represent output port assignments. Signals may also be used to hold temporary values (for example, the swapping of register contents) in order to model concurrent events within the sequential process.

4.3.6 Summary

This chapter presented the details of a proposed structured modeling methodology which does not restrict VHDL to a particular subset but recommends several writing styles for different design models. This methodology is based on the following principles:

1. Appropriate constructs in VHDL should be used for appropriate levels of design.
2. Guard expressions for block statements are used to represent clocks, or signals that enable storage.
3. Unguarded signal assignments should be used to model wires. Guarded signal assignments should be used for register and bus assignments. These constructs should not be mixed so that the model remains consistent for synthesis.
4. Design hierarchy and partitioning should be reflected in the description, although not with the same granularity.
5. It is believed that this structured modeling methodology will result in reduced modeling effort, allow portability of models, and facilitate synthesis of high quality designs.

Appendix A presents the VHDL coding practices and conventions for Structured Modeling as implemented in the VHDL Synthesis System.

Chapter 5

Design Representation

This chapter describes internal representation of the VHDL input description and the synthesized structural description used in the VSS system. The Design Representation consists of two views of the design: the *behavioral* view which is captured in a **Control/Data Flow Graph (CDFG)** representation, and a *structural* view which is maintained in the form of a **GENUS Partial Design** representation. The use of this representation to capture characteristics of four different design models (combinational, functional, register transfer, behavioral) will be illustrated.

5.1 Control/Data Flow Graph

5.1.1 Introduction

A *design representation* or *data base* is the internal representation used by a synthesis tool. It organizes information extracted from the input specification necessary for synthesis. This representation is created, manipulated, and optimized by the system so that a netlist or other output specification can be produced.

One common design representation used in several synthesis systems is the **control/data flow graph** [OG86]. The *control flow graph* represents sequencing information. Each "state" in the behavioral description is represented as a sequence of actions to be performed, and based on the evaluation of a condition, the next state to which execution is to be advanced is indicated. Control dependencies implied in the semantics of the behavioral description (for example, loop and if-then-else constructs) are preserved in the control flow graph.

The sequence of actions to be performed (arithmetic, logical, shifting operators) is represented using *data flow graphs.* A data flow graph indicates data dependencies that exist between variable accesses in assignment statements. The data flow graph exposes the parallelism in the input description. A control flow node representing a state will have a data flow graph associated with it.

5.1.2 Motivation

In synthesis, we are interested in generating a structural description of components from a given library from a behavioral description. Here, we are interested in properly connecting all pins on all components instead of observing signal values on some of the pins. The behavioral description must be parsed into a design representation which can be operated on by a variety of synthesis tools. This design representation should be well defined and should capture uniquely the functionality and intention of several equivalent behavioral descriptions in a format appropriate for synthesis. The representation must allow for the transformation of behavioral information (simulatable functionality) to structural information (library components and their attributes).

This section details the corresponding internal representation (control and/or data flowgraph) produced as the VSS input compiler parses each VHDL statement. The various interpretations of VHDL statements used to represent characteristics of each of the design models mentioned in our structured modeling methodology (combinational, functional, register transfer, behavioral) will be illustrated.

5.1.3 VHDL Design Representation in VSS

This subsection describes how each VHDL statement is processed by the VHDL Synthesis System (VSS) in order to generate and maintain an internal representation appropriate for synthesis. The **control/data flow graph (CDFG)** which is used as this internal representation is constructed as each statement is parsed. The portions of data and control flow graphs corresponding to the statements in a block or process are appropriately interconnected according to the design style used in the VHDL description.

Structural Description Style

A designer can specify an initial design, fully or partially, using a structural description mixed with behavior. When sections of the design are described using structural VHDL (for example, previously synthesized modules), these portions are copied intact to the output produced by the VSS system. The partial structural description is enhanced with additional components necessary to implement the sections of the design described using the data flow and behavioral styles.

When synthesis is completed, the VSS system produces a VHDL structural description of the design, using component declarations and instantiations derived from an Intelligent Component Data Base (ICDB) [Che90]. VHDL behavioral models for these components are available from the data base.

Dataflow Description Style

The dataflow description style emphasizes the flow of information between storage and gating elements.

Concurrent Statements

Concurrent statements are used to define interconnected blocks (components, possibly of different complexity) that jointly describe the overall behavior or structure of a design. Concurrent statements execute asynchronously with respect to each other. The following concurrent statements are found in VHDL:

```
concurrent_statement ::=
     block_statement
   | process_statement
   | concurrent_procedure_call
   | concurrent_assertion_statement
   | concurrent_signal_assignment_statement
   | component_instantiation_statement
   | generate_statement
```

Block Statement

The primary VHDL construct used for the dataflow description style is the **block** statement. A block statement defines an internal block representing a portion of a design. It has the following syntax:

```
block_statement ::=
   block [ (guard_expression) ]
      block_header
      block_declarative_part
   begin
      block_statement_part
end block;

block_header ::=
   [ generic_clause
   [ generic_map_aspect; ] ]
   [ port_clause
   [ port_map_aspect; ] ]

block_declarative_part ::=
   { block_declarative_item }

block_statement_part ::=
   { concurrent_statement }
```

The optional *guard_expression* defines an implicit signal GUARD which is of type BOOLEAN for simulation. If the guard_expression evaluates to TRUE, all signal assignments with a **guarded** qualifier appearing in the block_statement_part will have their RHS evaluated, and a driver is placed on the event queue to update the signal values at the appropriate time. For synthesis, the guard_expression is used to specify a synchronous or asynchronous event which results in a signal update.

The *block_header* explicitly identifies certain values or signals that are to be imported from the enclosing environment into the block and associated with formal generics or ports.

The *block_declarative_part* defines all local signals, types and subtypes, constants, components and attributes.

One or more concurrent statements constitute the *block_statement_part.* Blocks may be hierarchically nested to support design decomposition [IEE87]. The block statement groups together other concurrent statements such as signal assignments

which assign values to signals. Nested blocks are flattened for synthesis to facilitate resynthesis with optimization.

The flow graph representation for a block statement in shown in Figure 5.1.

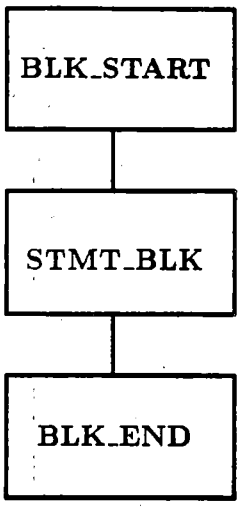


Figure 5.1: Block Statement Flowgraph Representation

It consists of BLK_START and BLK_END demarcation nodes, and a STMT_BLK node which represents the body of the block statement. The data flow graphs generated for each concurrent statement appearing in the block are associated with the STMT_BLK.

Signal Assignment

A signal assignment statement is used to assign or update values for a signal driver. The basic format of an assignment statement is the following:

target <= [**guarded**] <RHS-expression>

Each assignment made to a target or left hand side (LHS) signal/variable is represented by a WRITE node in the flow graph. Similarly, each access of a signal or variable appearing as a part of the right hand side (RHS) expression of an assignment statement is represented by a READ node.

READ and WRITE nodes for signals of can be of type PORT, REGISTER or WIRE (WIRE is the default for any variable declared as a SIGNAL). If a signal is of mode *internal* (that is, it was declared locally within some block statement) and a WRITE and READ node for that signal are connected when DFG sections are merged, the nodes can be coalesced, producing a signal net of type WIRE.

Conditional Signal Assignment

The *conditional signal assignment* statement has the following syntax:

```
signal <= [ guarded ] { <waveform> when <condition> else }
          <waveform> ;

<waveform> ::= <expression> [ after <delay> ]
```

The conditional signal assignment will occur in one of the following forms:

a) signal <= <waveform> ;

This is the simplest form of assignment statement. The VHDL simulator interprets this statement as a directive to compute the value of <expression> and schedule the activation of this driver for the signal value at time <current-simulation-time> + <delay> (if no delay is specified, the driver is activated immediately).

From the CDFG perspective, a dataflow graph is constructed for the RHS expression, and the result is input to a WRITE node for the signal. Associated with each graph arc (connection) is a **signal type** (bus, register, port, wire), **mode**

(in/out/inout (for ports only), internal), **bit width** (number of bits), and **representation** (integer, floating point, 1's complement, 2's complement, sign/magnitude). The optional delay specification indicates the time which elapses between the READ of all signals/variables which appear on the RHS of the assignment statement and the appearance (WRITE) of the updated expression value at the register/port/wire represented by the signal. Figure 5.2 shows a typical signal assignment statement and the corresponding flowgraph with delays.

```
entity EXAMPLE is
port (B,C: in BIT_VECTOR(3 downto 0);
  ...
architecture EX of EXAMPLE is
  ...
 signal A: BIT_VECTOR(3 downto 0);
  ...
  A <= B + C after 3 ns;
  ...
 end EX;
```

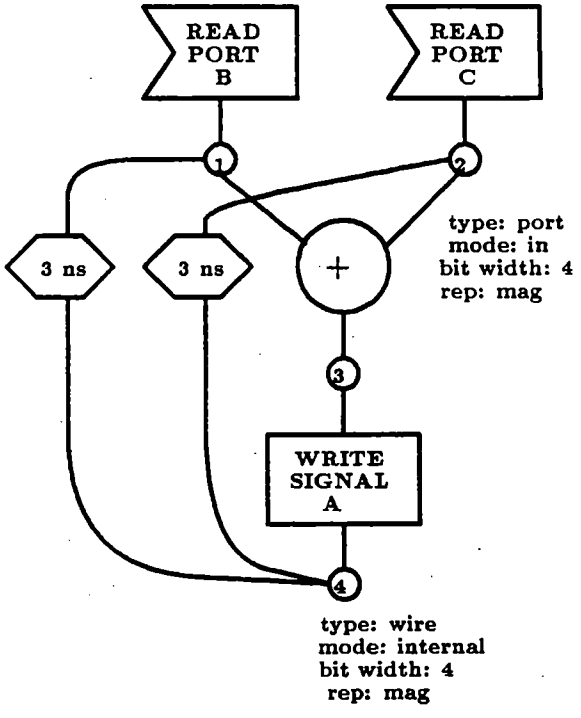


Figure 5.2: A Simple Conditional Signal Assignment

b) signal <= **guarded** <waveform> ;

The *guarded assignment* involves the conditional assignment of the evaluated <waveform> to the signal based on the value of the **guard expression** which appears at the beginning of the enclosing VHDL block statement. When the guard expression evaluates to TRUE, the VHDL simulator activates the signal driver and places its value on the simulator event queue so that the signal is updated at the specified simulation time.

For the purposes of CDFG generation and synthesis, a guarded signal assignment is used for signals declared with the **bus** or **register** signal kind qualifier. A data flow graph is generated for the RHS expression and is connected to the true input of a CHOOSE-VALUE node. The CHOOSE-VALUE node represents the selection of a data element based on the value of a guard (select) input. The guard input is a data flow graph representing the block guard expression. The output of the CHOOSE-VALUE node is used as the input to a WRITE node for the signal. Figure 5.3 shows an example of this construct.

If the signal is declared as a bus, the CHOOSE-VALUE will be mapped to a tri-state driver for the bus signal. If the signal is a register under a guard expression of type CLOCK, the CHOOSE-VALUE will be removed, and the select line will be connected to the clock input of the WRITE_REG node. The function of each signal appearing in the guard expression is determined by its signal type. In the case of multiple signals in the guard expression (clock and set, for example), an optimization step will connect each signal to the appropriate control input.

```
c) signal <= [ guarded ]
      waveform1 when condition1 else
      waveform2 when condition2 else
      ...
      waveformM when conditionN else
      waveformN;
```

```
entity CONTROLLED_CTR is
  port (CLK: in CLOCK;
        DATA: in BIT_VECTOR(3 downto 0);
        ...
architecture DATAFLOW of
  CONTROLLED_CTR is
  ...
signal CNT: BIT_VECTOR(3 downto 0) register;
  ...
CNT_UP: block (CLK = '1' and not CLK'STABLE)
  begin
  CNT <= guarded CNT + "0001" after 10 ns;
  ...
end DATAFLOW;
```

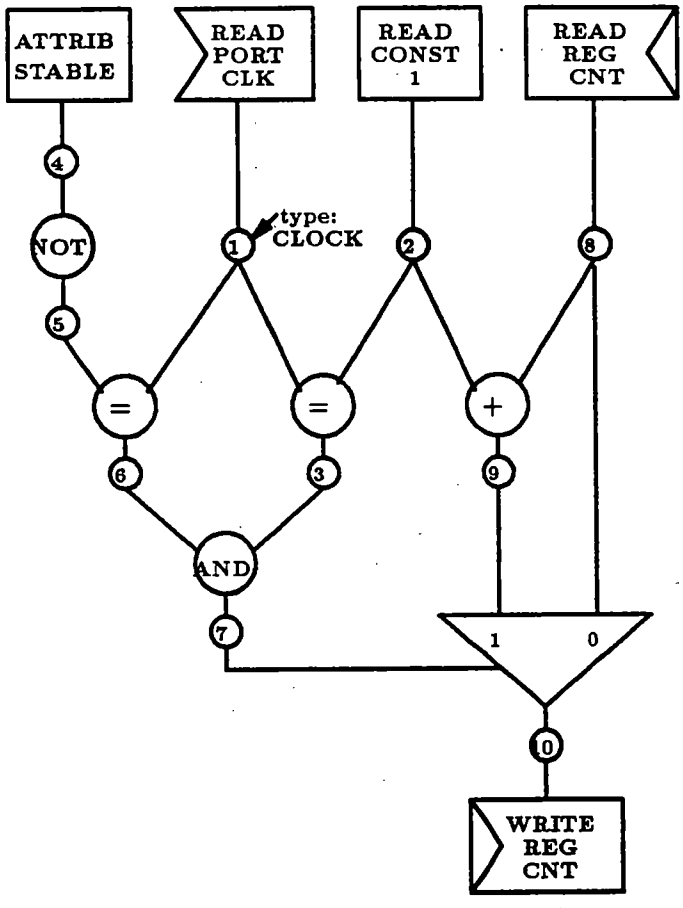


Figure 5.3: Guarded Signal Assignment

This statement corresponds to a nested if arrangement of assignments to the same signal based on different boolean conditions. The VHDL simulator will evaluate waveform/condition pairs in the order in which they appear and will schedule the assignment of the first waveform value to the signal when its associated condition evaluates to true.

The conditional assignment statement can be useful in representing an assignment to a signal based on prioritized conditions. For example, the statement in Figure 5.4 might be used to represent a register for which the CLEAR is of highest priority, followed by PRESET and CLOCKed assignment. Figure 5.4 shows the flowgraph generated for the statement.

A chain of CHOOSE-VALUES is constructed to form the data flow graph for the nested if construct. The bottom most CHOOSE-VALUE is guarded by the first condition encountered, the CHOOSE-VALUE above the bottom one is guarded by the next condition, etc. The output of the bottom most CHOOSE-VALUE is connected to the WRITE node input.

Selected Signal Assignment

The format of the *selected signal assignment* is shown in Figure 5.5. This is equivalent to the case statement available as a sequential statement within the process construct. The choices are exclusive conditions (either integer or boolean values) such that only the waveform matching the value of the <expression> is evaluated and scheduled for assignment by the VHDL simulator. Figure 5.5 shows the flowgraph generated for the general form of this statement.

```
block (CLR = '0' or SET = '1' or CLK = '1')
    begin
     A <= guarded
          '0' when (CLR = '0') else
          '1' when (SET1 = '1') else
            DATA when (CLK = '1') else
            A;
    end block;
```

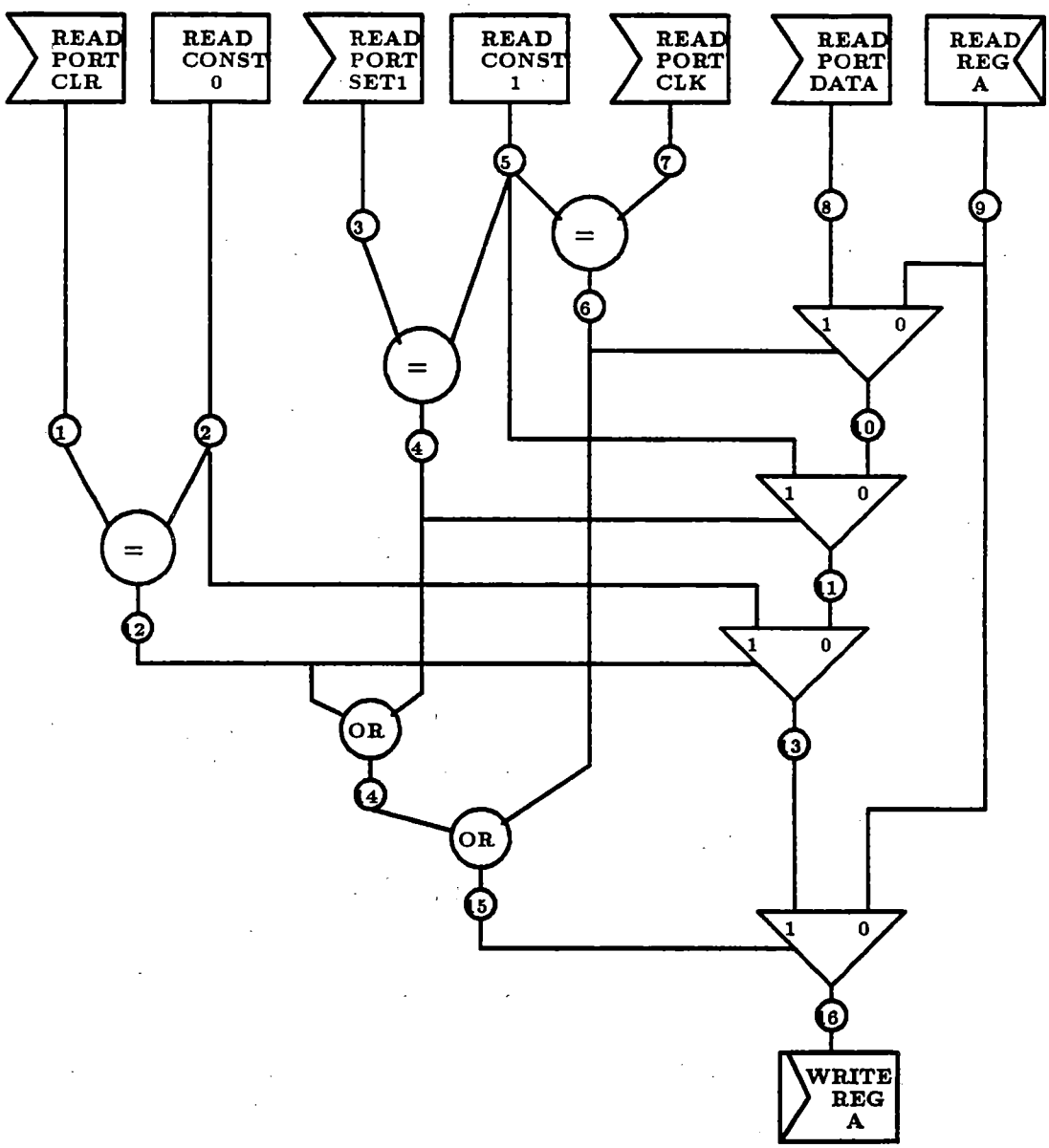


Figure 5.4: Conditional Signal Assignment

The data flow graph construct associated with this statement is the multiple input CHOOSE-VALUE guarded by the <expression>. Each waveform will have a corresponding data flow graph generated for its expression value, and the guard test for each input will be stored in the input net.

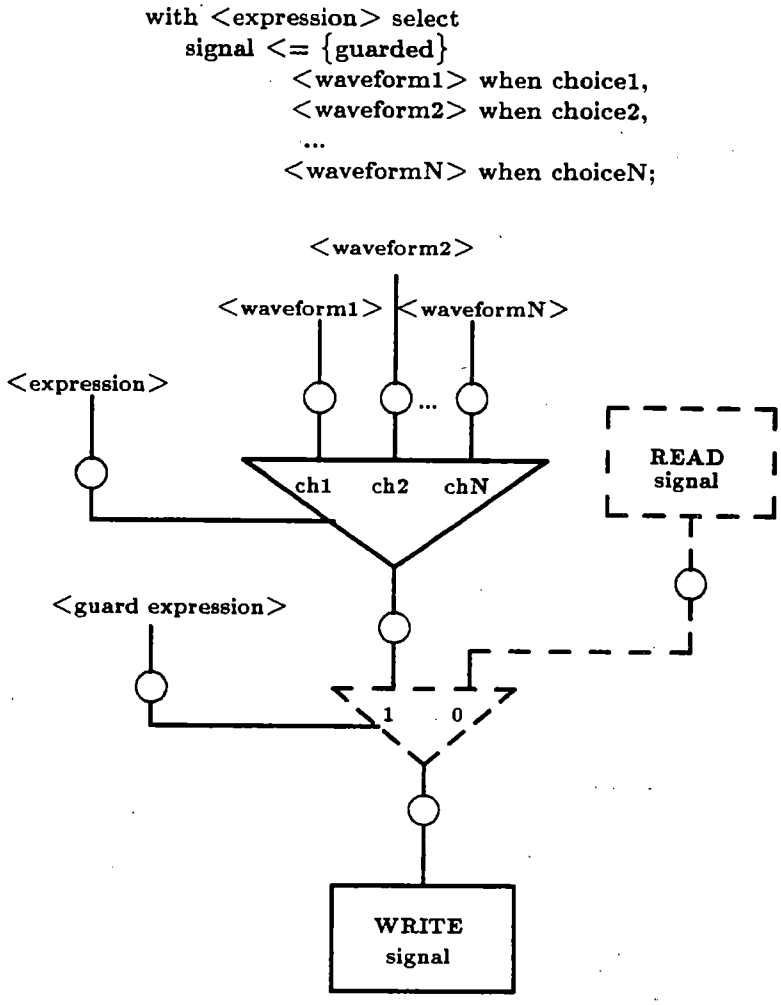


Figure 5.5: Selected Signal Assignment

Behavioral Description Style

A *behavioral description* is a sequentially executed, procedural style of code typical of common programming languages. A behavioral specification specifies, with any desired degree of precision, what a device does (its function) without specifying how it does it (its structure) [CAD87].

Process Statement

The primary VHDL construct used for the behavioral description style is the **process** statement. A process statement defines an independent sequential process representing the behavior of some portion of the design. It has the following syntax:

```
process_statement ::=
   process [ (sensitivity_list) ]
      process_declarative_part
   begin
      process_statement_part
end process;

process_declarative_part ::=
   { process_declarative_item }

process_statement_part ::=
   { sequential_statement }
```

The execution of a process statement consists of the repetitive execution of its sequence of sequential statements. After the last statement in the sequence of statements of a process statement is executed, execution will immediately continue with the first statement in the sequence of statements [IEE87].

A *sensitivity list* may be specified for each process. By specifying a sensitivity list of one or more signals, the process statement is assumed to contain an implicit wait statement as the last in the sequence of statements. This wait statement will suspend execution of the process statement until an event (change) occurs involving

one of the signals in the sensitivity list. The sensitivity list is ignored by the VSS synthesis tool.

The *process_declarative_part* defines all local signals, variables, types and subtypes, constants and attributes.

One or more sequential statements comprise the *process_statement_part*. The sequential statements which may appear in the description are listed in the next section.

The flow graph representation for an example process statement is shown in Figure 5.6. Note that a STMT_BLK node is a control node which has an associated data flow graph. These data flow graphs are constructed for sequential signal and variable assignment statements.

Sequential Statements

The sequence of statements within a process statement may contain one or more of the following statement types:

```
sequential_statement ::=
     wait_statement
   | signal_assignment_statement
   | variable_assignment_statement
   | procedure_call_statement
   | if_statement
   | case_statement
   | loop_statement
```

As mentioned above, data flow graph sections for assignments of values to signals and variables are created as in the case of concurrent signal assignments and associated with STMT_BLK nodes. Control flow graph sections are created for each of the behavioral control constructs. These control flow graph sections are nested

```
process
 begin
  while (stop = '0')
  PI := M(CR)(15 downto 0);
   S := PI(15 downto 3);
   case PI(2 downto 0) is
     when 0 => CR := M(S);
     when 1 => Acc := Acc - M(S);
       ...
     when 6 => if (Acc < 0) then
                  CR := CR + 1;
     when 7 => stop <= '1';
   end case;
  end loop;
end process;
```

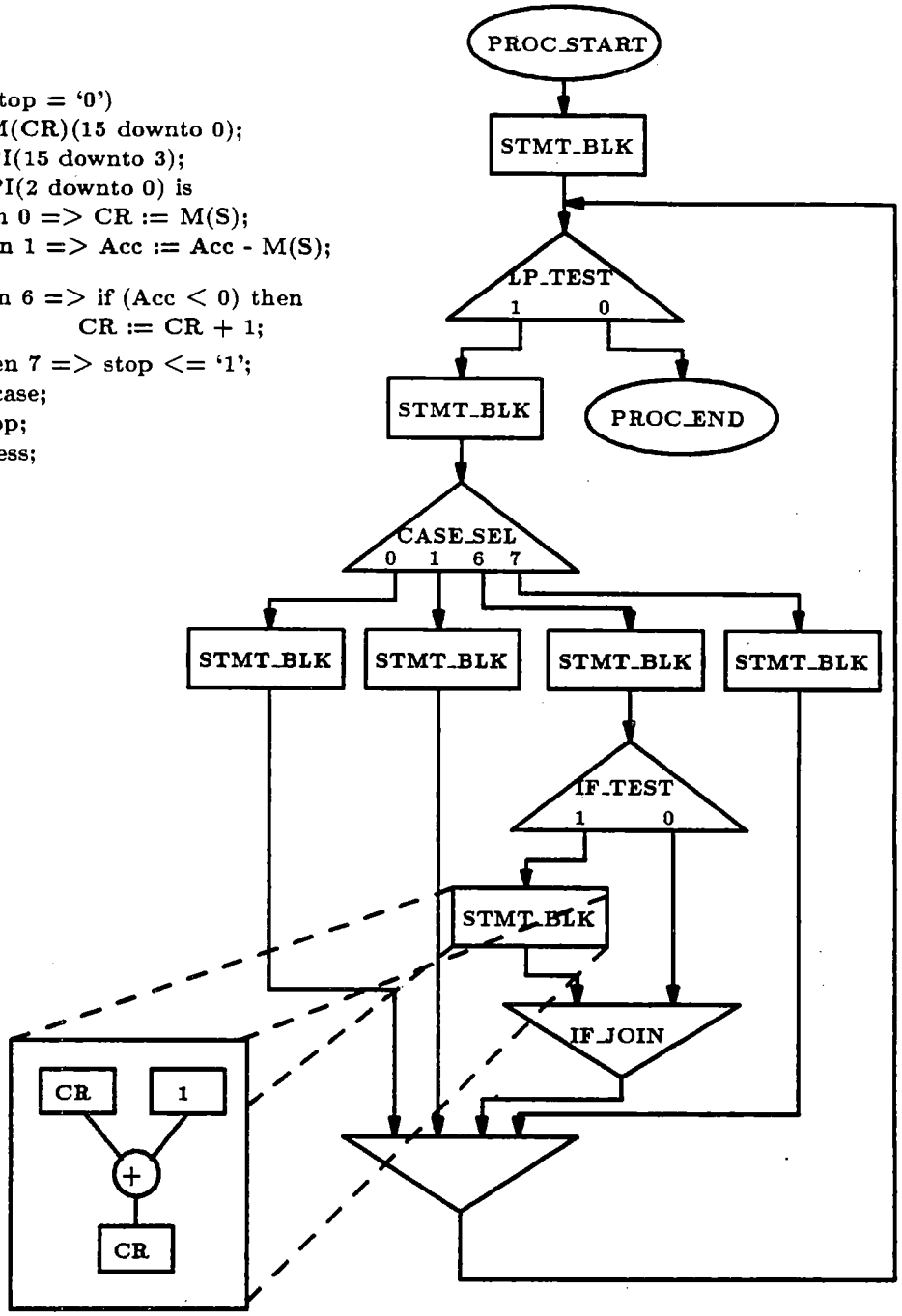


Figure 5.6: Process Statement Flowgraph Representation

and interconnected to model the flow of control implicit in the sequential, behavioral description.

Signal Assignment

The syntax of the signal assignment statement for a sequential process is identical to form (a) of the conditional signal assignment in a concurrent block. A data flow graph similar to the representation generated for a concurrent signal assignment (see Figure 5.2) is created.

Variable Assignment

A variable assignment statement replaces the current value of a variable with a new value specified by an expression. The statement has the following syntax:

target := <expression> ;

This statement cannot use the **after** clause to specify timing relationships as in the signal assignment statement. A data flow graph is generated to represent the variable assignment.

If Statement

One construct used to model conditional execution in the VHDL process statement is the **if** statement. The if statement performs a conditional branch based on the value of a boolean signal.

The control flow graph section created to represent the if statement consists of three parts: (1) a TEST (or SELECT) node which selects the control branch to be taken based on the test signal; (2) for each control branch, one or more control nodes representing a sequence of statements to be performed in that branch; (3) a JOIN

node which signifies the end of each conditional branch and connects to the flowgraph section for the next sequential statement. Figure 5.7 shows the control flow graph sections created for the if construct.

```
if (boolean_expression)
   then
     seq_of_statements_1
   else
     seq_of_statements_2
   end if;
```

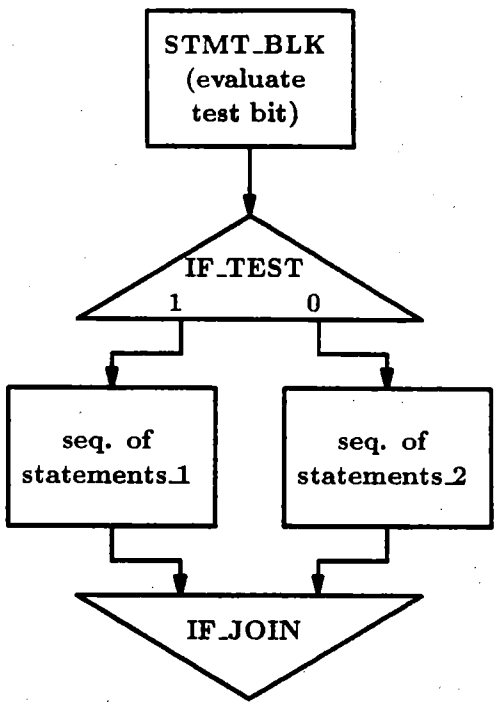


Figure 5.7: If Statement

Case Statement

The case statement selects between two or more conditional branches based on the value of an integer select signal. Figure 5.8 shows the flowgraph representation for the case statement.

```
case (integer_expression) is
    when choice_1 =>
      seq_of_statements_1
    ...
    when choice_N =>
      seq_of_statements_N
  end case;
```

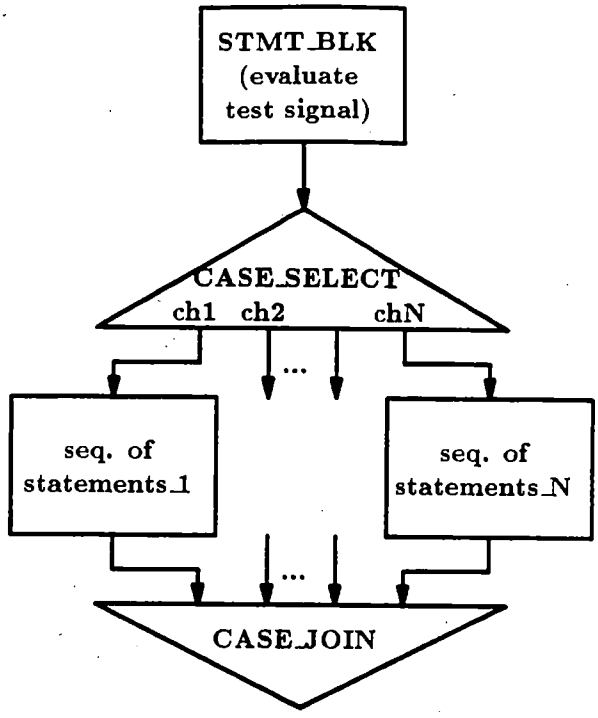


Figure 5.8: Case Statement

For Loop

A loop statement includes a sequence of statements that is to be executed repeatedly, zero or more times.

The **for** loop construct uses an index variable whose value steps through a specified range for each iteration of the loop. The index variable is set to the first value in the range prior to entering the loop. A test is made to determine if the index value is within the range; if so, the loop body is entered. Once the loop body statements are executed, the index variable assumes the next value in the specified

range, and control is returned to the loop entry test. If the test returns FALSE, control passes to the next sequential statement. Figure 5.9 shows the for loop representation.

```
for identifier in discrete_range loop
    seq_of_statements
end loop;
```

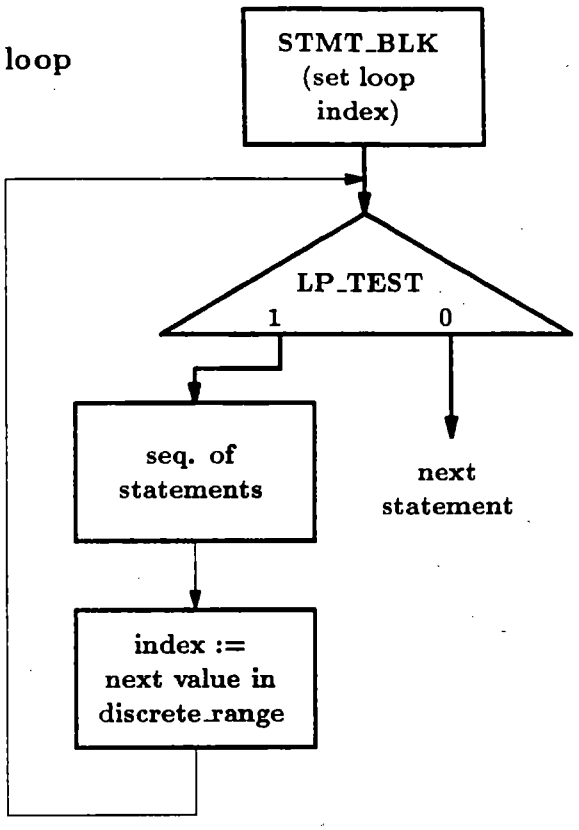


Figure 5.9: For Loop Statement

While Loop

The **while** loop construct tests a boolean condition, and if it is TRUE, passes control to the first control node of the flowgraph section implementing the sequence of statements for the loop body. Once the loop body statements are executed, control returns to the condition test which is repeated. If the condition evaluates to FALSE,

control passes to the sequential statement following the while loop. Figure 5.10 shows the representation corresponding to a while loop.

```
while (boolean_expression) loop
    seq_of_statements
end loop;
```

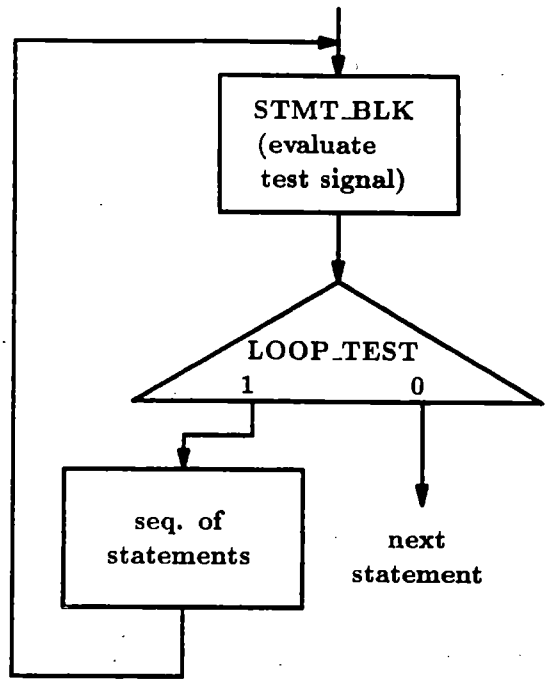


Figure 5.10: While Loop Statement

Procedure Call

The procedure call has the following syntax:

procedure_name (<parameter_list>);

Procedure calls are used in a VHDL description to invoke a procedure body consisting of sequential statements which are used one or more times in the description. Figure 5.11 shows the flow graph representation for a procedure call.

The procedure call may be processed in one of two ways:

proc_name (<parameter_list>);

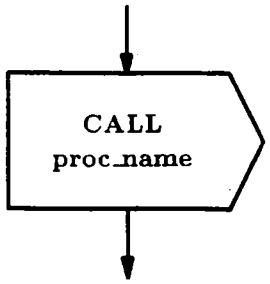


Figure 5.11: Procedure Call Statement

1. In-line expansion of each call may be performed, where the statements of the procedure body are substituted for the procedure call statement. A template flowgraph created for the procedure body is inserted, with actual parameters replacing occurrences of formal parameters. When this description is synthesized, each procedure call invocation can be mapped to available hardware in the data path, or a microcode implementation in control can be implemented. Annotations in the VHDL description will determine the implementation style.
2. The procedure body is treated as a description of a block in the design. A flowgraph is created for the procedure body. Hardware is synthesized for this description, and each procedure call supplies the values of actual parameters as inputs to the procedure body hardware.

Wait Statement

The *wait* statement has the following syntax:

```
wait [ <condition_clause> ] [ <timeout_clause> ] ;

   <condition_clause> ::= until <boolean_expression>
   <timeout_clause>   ::= for <time_expression>
```

wait until cond_expr
for time;

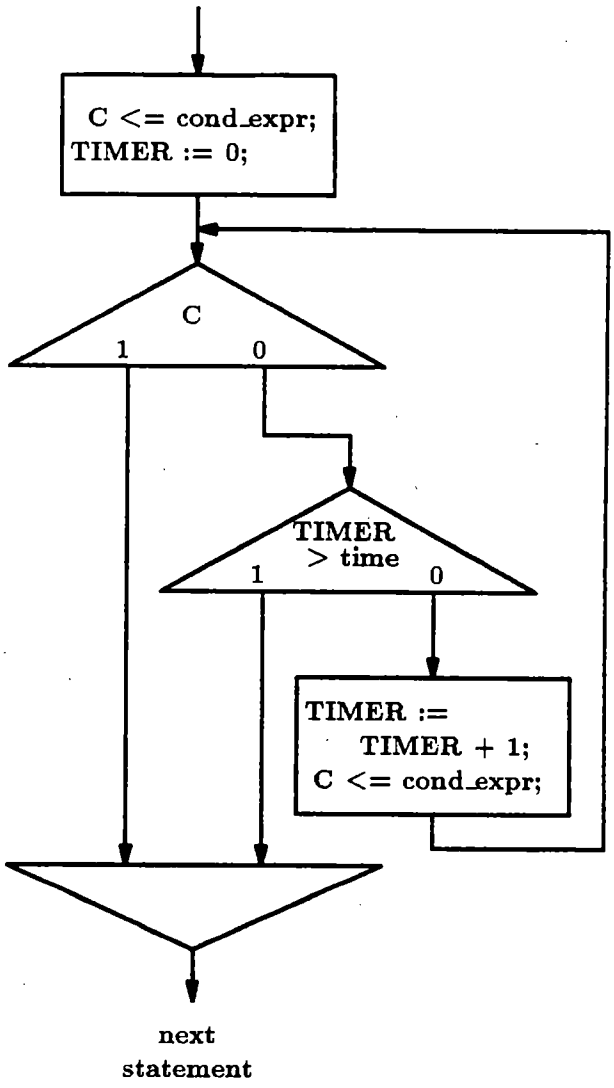


Figure 5.12: Wait Statement

A wait statement is used to suspend the execution of a process statement until a specified condition is TRUE, or a timeout period elapses. Figure 5.12 shows the control flow graph sections created for a wait statement with condition and timeout clauses. This statement is implemented in control and is synchronized with the system clock; time is measured in multiples of the clock period.

5.1.4 Annotations

In some instances, it is necessary to indicate to the VSS system which design process should be used for a given VHDL description. This is accomplished through the use of *annotations* in the form of special VHDL comments as shown below:

VSS: functional description

Annotations are used in the following situations:

1. To indicate the structured modeling style used in the VHDL description.
2. To indicate that a CFG to DFG transformation is to be applied to a process control construct (IF, CASE or LOOP). For example, it may be desirable to unwind a loop, where iterations are flattened into a sequence of assignments, rather than implementing indexing or conditional tests in control.
3. To denote a next state in process descriptions. This can be used to define state boundaries for a register transfer description consisting of a sequence of assignment statements.

5.2 Partial Design Representation

The *partial design representation* stores the data path structure and the control specification produced by VSS. The data path structure is represented using the three levels of hierarchy of a GENUS [Dut88] component description; these levels will be described briefly in the next section. The control unit specification is derived from information stored in the flow graph design representation. State and resource bindings performed by the scheduling and resource binding modules of the Design Compiler annotate the flow graph with state and component assignments for register transfer and behavioral designs. This information can be extracted, formatted, and presented to the designer in the form of Behavioral Intermediate Format (BIF) [DHG89] state tables.

5.2.1 GENUS Partial Design Representation

The GENUS generic component library consists of three levels of hierarchy: generators, component classes, and component instances. A *generator* is used to represent a family of similar components and instances. The generator descriptor maintains a list of all possible parameters and a specification of each operation performed by the generated component. The *component class* is the product of a call to a parent generator with a particular set of parameters. For example, a 4-bit register component class is generated by calling the register generator with a bit width parameter of 4. The component class representation is maintained by the VSS tool as a part of the design data base which stores the partial design being synthesized. *Instances* are "carbon copies" of a parent component class, distinguished by an unique

name. Each instance corresponds to an actual component in the partial design. The instance inherits its attributes from the parent component; consequently, the primary information stored in this level of the GENUS hierarchy involves the connectivity of the instance.

It is often desirable to represent a hierarchical decomposition of a design, where the top level of the hierarchy contains a small number of components which have been constructed from more primitive elements. For example, the VHDL structural description style allows for a hierarchical description using *entity/architecture* pairs to describe portions of the design and *configuration* statements to indicate the decomposition of component instances. This hierarchy must be distinguished from the hierarchy present in the GENUS component representation; the latter is associated with the representation of a single component, while the former refers to the representation of the entire design which at any intermediate level may consist of a mixture of simple GENUS component instances and groupings of instances which are expanded in lower levels. The configuration specification indicates the expansion of a complex component in the higher levels of the design hierarchy in terms of more primitive components, where the leaf level components in the hierarchy are constructed using the most basic elements (pure GENUS components).

The GENUS component representation has been adapted and extended to suit the requirements of partial design representation in VSS. In order to represent the partial design hierarchy as it is synthesized from the VHDL input description, an *entity* object has been defined to represent a collection of component instances at any level of the abstraction. Components within an entity may themselves be entities, allowing for structural decomposition of the partial design. Associated with each entity object are lists of GENUS component classes and GENUS component instances.

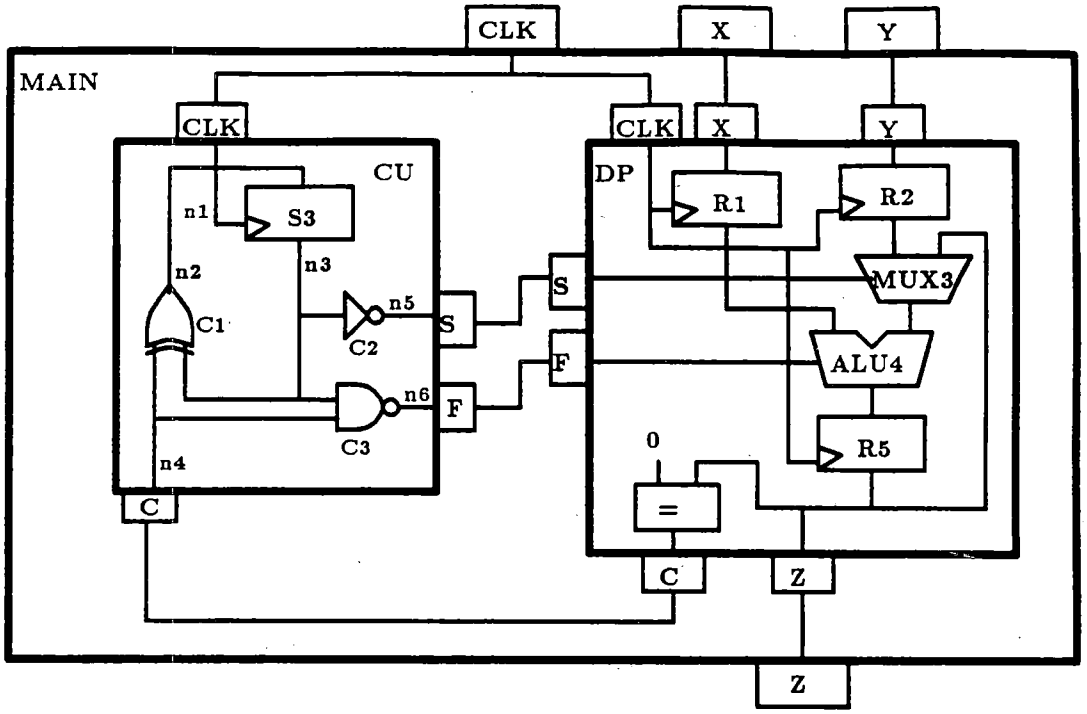
If a component in this entity is to be decomposed further, links are maintained in an entity hierarchy. Figure 5.13 shows an example partial design netlist and the corresponding representation in terms of entity objects and GENUS component instances. The entity object hierarchy is easily translatable into the entity/architecture pair hierarchy of a VHDL structural description or netlist.

Available component generators are introduced to VSS using a GENUS generator input parser. This parser reads a textual file of GENUS generic component generator specifications written using the LEGEND language [Dut90] and produces an internal data structure which maintains the generator information. Upon completion of and integration with the Intelligent Component Data Base [Che90], this information will be obtained through data base queries.

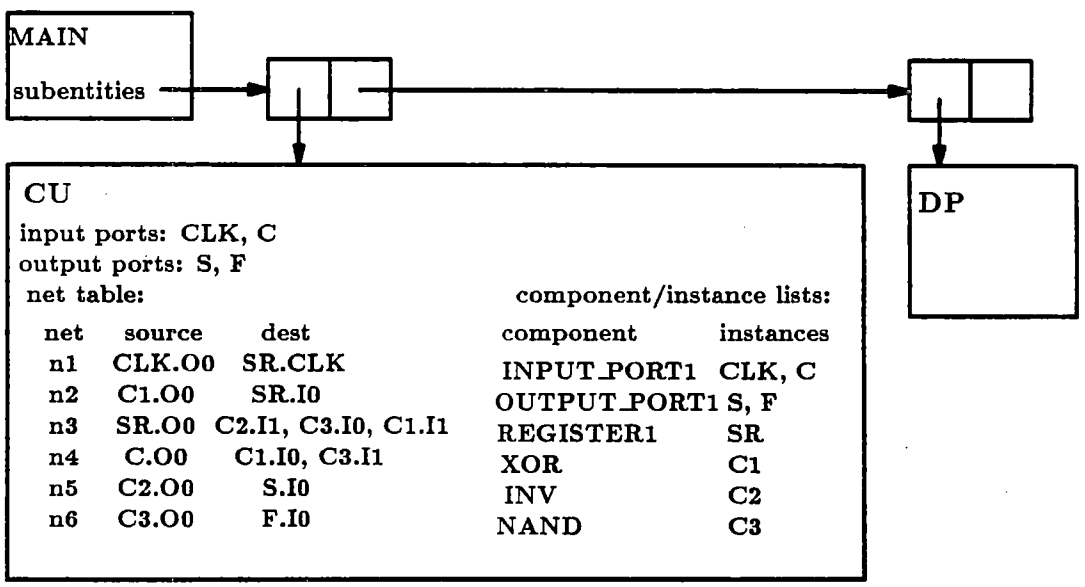
5.2.2 State Tables

The format used to capture the sequencing information present in the control flow graph and present it to the designer in readable form is the Behavioral Intermediate Format (BIF) [DHG90]. These tables are derived from a traversal of the control/data flow graph; branching conditions for loop, if and case conditional constructs become the conditional event under which data flow and/or sequencing operations are performed. The task of state scheduling annotates the CDFG with state binding information; this information is reflected in the current and next state assignments specified in the BIF tables.

Upon completion of the scheduling task of synthesis of a VHDL behavioral model, an *operation based* state table (OBST) is generated. The behavioral operations that are performed in each state are specified in the action field of the table.



Hierarchical Structure



Entity Object Hierarchy

Figure 5.13: GENUS Partial Design Entity Object

The *unit based* state table (UBST) captures the assignment of behavioral operators to units in the partial design structure on a state-by-state basis after the resource allocation and binding tasks have been performed. For each state, all units which perform an operation in that state are listed in the action field. The unit performing the operation, its operands, the operation to be performed, and the destination are specified for each unit usage. This specification links the behavioral operator from the CDFG to a specific instance of a functional or storage element in the GENUS Partial Design Representation.

Chapter 6

Synthesis System Framework

This chapter describes the system architecture of the VHDL Synthesis System (VSS), providing details of its major components. The block diagram of the VSS system is shown in Figure 6.1.

The VSS system consists of four subcomponents: a **Graph Compiler** component, a **Representation Optimization** component, a **Design Compiler** component, and an **Output Generation** component. The *Graph Compiler* module accepts a VHDL description and generates the Control Data Flow Graph (CDFG) internal representation which is operated on by subsequent components of the VSS system. Various local and global transformations are applied to the CDFG within the *Representation Optimization* component. These optimizations restructure the internal representation in order to facilitate efficient synthesis as performed by the Design Compiler.

The optimized flow graph is then processed by the *Design Compiler*. The Design Compiler consists of a collection of algorithms which perform the allocation, scheduling and resource binding tasks of high-level synthesis. An appropriate sequence of synthesis procedures is determined by the selected design model and by directives

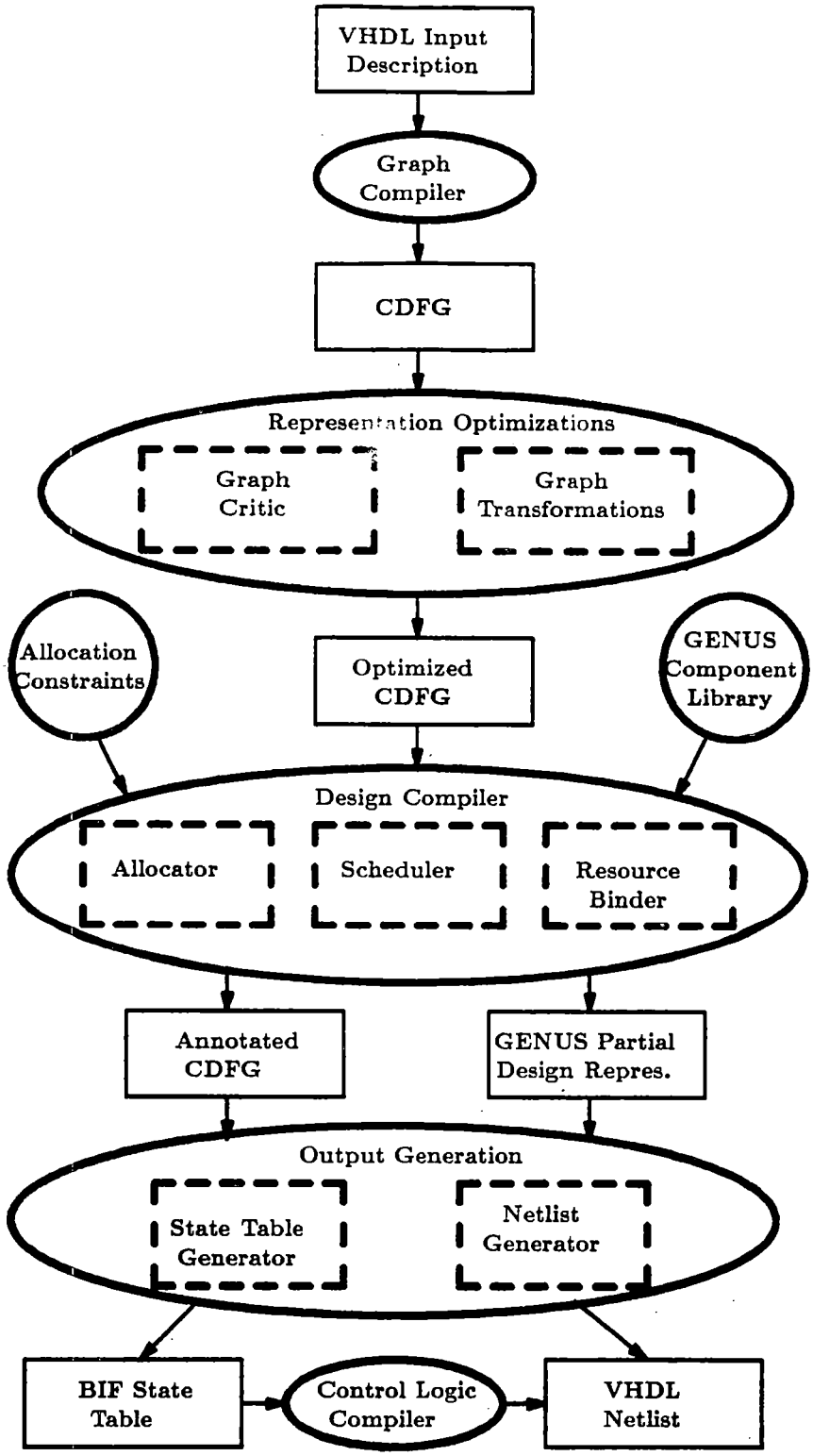


Figure 6.1: VSS Block Diagram

specified through annotations in the input description which are recorded in the design representation. Allocation constraints are supplied to the Design Compiler algorithms via a textual file. As each subtask within the Design Compiler completes, the synthesized structure of the design is generated using components from the GENUS generic component library [Dut88]. This structural view of the design is maintained in the GENUS Partial Design Representation described in section 5.2.1. In addition, the CDFG is annotated with information that relates the binding of behavioral operators and variables to the corresponding structural components.

Results of the synthesis process to be examined by the designer are created by the *Output Generation* module. The GENUS Partial Design Representation is presented in textual form via a VHDL structural description. If a multiple state design is produced, a specification of the control sequencing required is generated in the textual form of a BIF [DHG89] unit based state table derived from the annotated CDFG.

A *Control Logic Compiler* utility derives a description of the behavior of control unit components from a BIF state table. This specification is synthesized to produce an implementation of the control unit, thereby completing the design structure.

6.1 Graph Compiler

The *Graph Compiler* parses the VHDL input description into the Control/Data flow graph representation used internally in the VSS system. The Graph Compiler operates in one of two modes, *concurrent* or *sequential*, depending on the VHDL concurrent statement (block, signal assignment, process or procedure) currently being

processed. The primary difference in these operation modes involves the processing and interconnection of signal or variable assignment statements. When processing a VHDL block statement, the assumed design model dictates that signal assignments are concurrent; therefore, the Graph Compiler does not introduce data dependencies between the update of a signal and subsequent accesses of that signal in the same block. Conversely, in sequential mode, the design model requires the enforcement of data dependencies in a sequence of assignment statements occurring within a VHDL process statement; in this case, data dependency arcs are introduced in the data flow graph representation.

Each VHDL concurrent statement is processed in order of occurrence in the input description, producing a corresponding Control and/or Data flow graph representation. The hierarchy of the VHDL description (e.g., nesting of block and process statements) is preserved in the internal representation. Annotations encountered in the input description are used to guide the Graph Compiler in the following cases:

- *Typing of signals* - if the designer wishes to bind a variable in the description to a hardware element (register, bus, etc.), an annotation appearing just prior to the signal/variable definition (see section 5.1.4) will appropriately type all references in the representation.
- *Selection of design style* - if it is desired to implement a block or process using a particular design model, a comment annotation appearing just prior to the statement will result in the selection of the appropriate Graph Compiler mode and application of subsequent optimizations and transformations.

- *Control construct transformations* - if such a directive is encountered in the description, the annotated sequential statement will be marked, and transformations will be applied to convert this statement to an equivalent concurrent (data flow) representation.

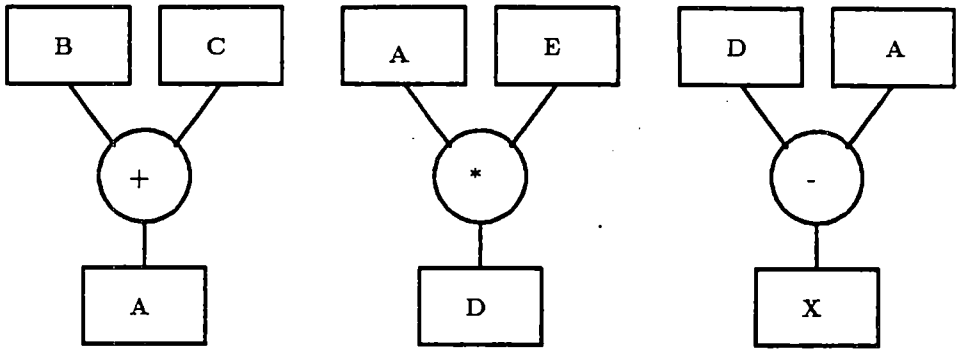
6.1.1 Block Statement Compilation

For signal assignments appearing in a block statement, flowgraph sections generated for each statement are interconnected once all statements have been processed. This corresponds to the concurrent data flow style where all operations are assumed to be executed in parallel. Variables appearing on the left hand side (LHS) of an assignment statement are assigned the value of the variable prior to the execution of the block statement. Figure 6.2 shows a VHDL code fragment consisting of several concurrent assignment statements, the flow graphs created for each statement, and the final interconnected flow graph.

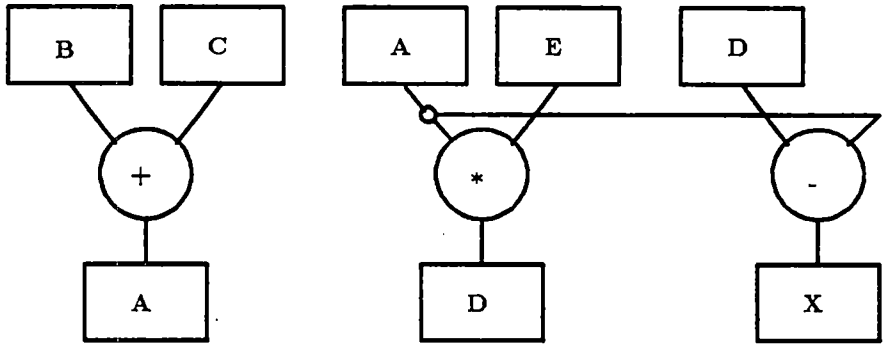
The sections of DFG representing each signal assignment will be appropriately interconnected based on the signal type. It is the *signal kind* that will define whether a VHDL signal (container) represents a memory element, port, bus or wire. Guarded signal assignments indicate that the assignment target signal is of signal kind REGISTER (if the block guard is of type CLOCK, SET or RESET) or BUS. Unguarded signal assignments signify SIGNAL (or wire) targets when access to these signals occurs within the scope of the block being processed (i.e., the signal must be defined within the current block, and a READ and WRITE of the signal occur in the block); otherwise, the signal access will be mapped to a PORT in subsequent Design Compiler processing.

```
A <= B + C;
D <= A * E;
X <= D - A;
```

VHDL Concurrent Statements



Individual Statement Flow Graphs



Interconnected Flow Graphs

Figure 6.2: Block Statement Compilation

Structured Modeling recommends that for concurrent (block level) descriptions, a single block should be used for a signal update, or multiple blocks are allowed to specify mutually exclusive updates to a signal. In the case where multiple blocks are used to describe exclusive functionality of a component, Design Compilation may require the flattening of these multiple blocks into a single DFG to facilitate mapping to GENUS components. In order to accomplish this, the signal kind is used to determine the interconnect protocol which results when multiple sources for the same VHDL signal are encountered within a DFG section.

Multiple WRITEs (sources) to a signal of signal kind SIGNAL indicate that a WIRED-OR node should be created with each WRITE node as an input. Any READ nodes for this signal should be connected to the output of the WIRED-OR node. This DFG construct will be mapped to a wired-or connection during design compilation.

Similarly, recognition of multiple WRITEs to a signal of signal kind BUS should produce a BUS node to which all WRITE nodes are connected. Since each WRITE node for a signal of type bus was created when a guarded signal assignment was made, each input is controlled by some guard. This flow graph pattern will be mapped to a bus connection, where each CHOOSE-VALUE controlling a WRITE input becomes a tri-state bus driver.

Accesses to signals of type register are merged into a single WRITE access node. The inputs are muxed on the data input if they are synchronous, or are applied to different inputs (e.g., load and clear) if they are asynchronous.

The compilation algorithm for concurrent statements is summarized in the procedure **interconnect_concur_stmts** shown in Figure 6.3 below.

```
interconnect_concur_stmnts ()
{
   merge duplicate READ (SIGNAL,REGISTER,CONSTANT) nodes

   for (each WRITE node)
      switch (signal kind)
         case SIGNAL   : if (WIRED-OR node does not exist)
                            create WIRED-OR node
                         attach data input of WRITE node as input to
                            WIRED-OR node
         case BUS      : if (BUS node does not exist)
                            create BUS node
                         attach data input of WRITE node (from a
                            CH-VALUE node) as input to BUS node
         case REGISTER: if (another WRT_REG node for same var exists)
                            merge WRT_REG nodes, connecting appropriate
                               control lines

   look at all WRITE nodes and appropriately connect them to READ
      nodes for the same signal
   invoke Graph Critic
}
```

Figure 6.3: Concurrent Statement Processing Algorithm

6.1.2 Process Statement Compilation

Unlike concurrent statements which are interconnected once all statements in the block have been processed, sequential statements appearing within a process statement are interconnected as they are encountered. Each control flow graph section corresponding to a sequential statement (STMT_BLK, if, case, loop, wait and procedure call) has a single entry point and single exit point. As these statements are processed, the exit point of the previous statement is connected to the entry point of the current statement. Since the control flow graph sections of most sequential statements are hierarchically constructed from other sequential statements, a stack is

used to maintain the control flow node to which the current control flow node is to be attached.

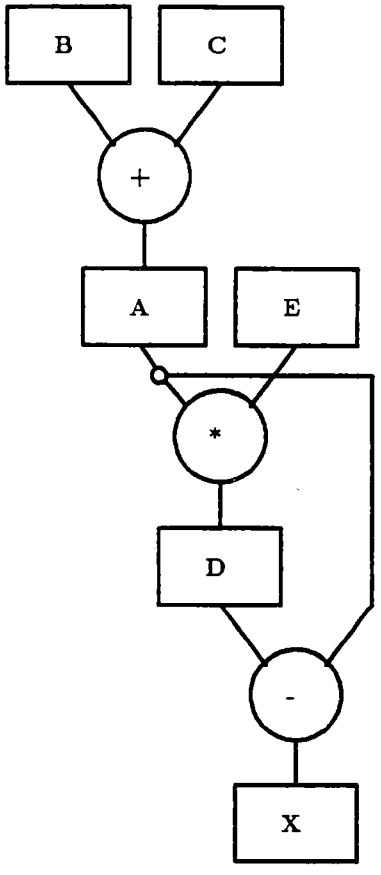
When processing conditional branching constructs (IF, CASE and LOOP statements), the CU/DP design model used in VSS assumes that the branching condition evaluates to either a BOOLEAN value (in the case of IF and LOOP statements) or an integer/binary value in a discrete range (in the case of CASE statements). If a branching condition is an expression consisting of one or more operations, a DFG which computes this expression value must occur in a STMT_BLK preceding the decision node. The value is assigned to a temporary variable created by the Graph Compiler, and the decision node is annotated with the name of the variable on which the conditional depends.

Assignment statements are associated with the current STMT_BLK. Thus, a sequence of assignment statements is grouped initially into the same STMT_BLK control node until state binding is performed by the scheduling subtask of design compilation. A STMT_BLK is created if the current CFG node is not a STMT_BLK when an assignment statement is encountered.

For signal or variable assignments appearing in a process statement, data flow graph sections are generated for each statement. The location of the last update (WRITE) of all signals and variables is maintained. Variables appearing on the left hand side (LHS) of an assignment statement are assigned this last update value. If a value is updated and subsequently accessed within the same STMT_BLK, the data flow WRITE and READ nodes, respectively, are interconnected.

```
A := B + C;
D := A * E;
X := D - A;
```

VHDL Variable Assignment Statements



Interconnected Sequential Statements

Figure 6.4: Compilation of Variable Assignments in a Process

Figure 6.4 shows the same VHDL code fragment from the previous section as it would appear in a process statement consisting of several variable assignment statements. Notice that the sequential nature of the process imposes data dependencies on the variable accesses, resulting in a different interconnected flow graph.

6.2 Representation Optimizations

6.2.1 Graph Critic

Because VHDL allows the designer to express the same functionality in many different abstract ways, a **Graph Critic** module is needed to transform these various representations into an unique representation which captures the hardware concept being described. The initial parsing of the VHDL input description into the CDFG Design Representation produces an abundance of DFG expression trees which are derived from event specifications such as block guards and condition clauses of signal assignment statements. These expressions represent attributes of signals and hardware components, rather than boolean functions which require logic gate implementation. Left unoptimized, these expression trees will be mapped to unnecessary logic. In addition, the interconnection of DFG sections which represent individual VHDL statements often requires additional manipulation of the CDFG representation.

The Graph Critic contains two rule sets which perform local optimizations. *Cleanup rules* eliminate redundant constructs in the flowgraph. For example, the WRITE of a signal of type REGISTER followed by the READ of that signal will be represented as a WRITE node connected to a READ node via a data dependency arc.

One Graph Critic rule recognizes such a pattern and merges the READ and WRITE nodes into a single node. Figure 6.5 illustrates the operation of this rule.

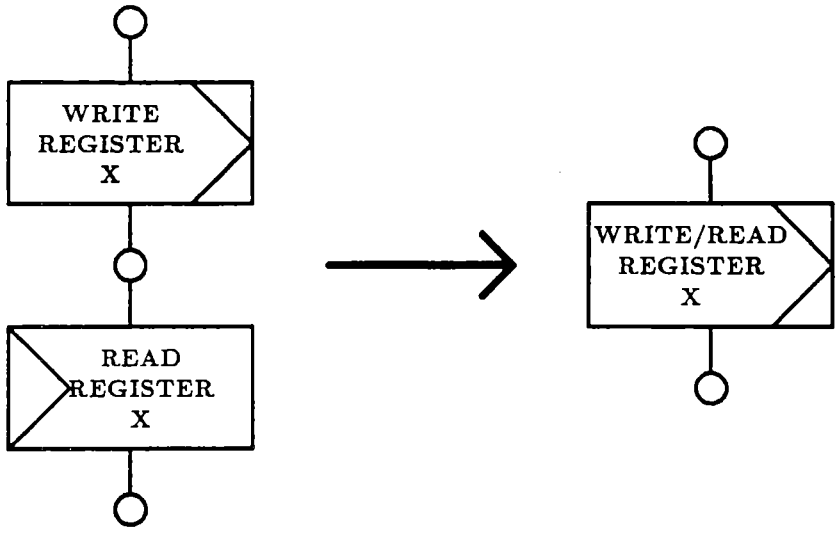


Figure 6.5: Graph Critic Cleanup Rule

Optimization rules systematically replace behavioral constructs with those which more closely resemble library components and their attributes. For example, a rising edge event must be described as follows in VHDL: (X = '1' and not X'STABLE). The Graph Compiler will initially produce a data flow graph containing comparison and logic operation nodes for this expression. A Graph Critic rule will then be applied, replacing the expression tree with a POSITIVE EDGE sensitivity attribute on the output arc of the READ X node. Figure 6.6 shows the results of applying this optimization rule.

The Graph Critic is applied to each STMT_BLK which contains data flow nodes upon completion of graph compilation for that data flow block. This optimization

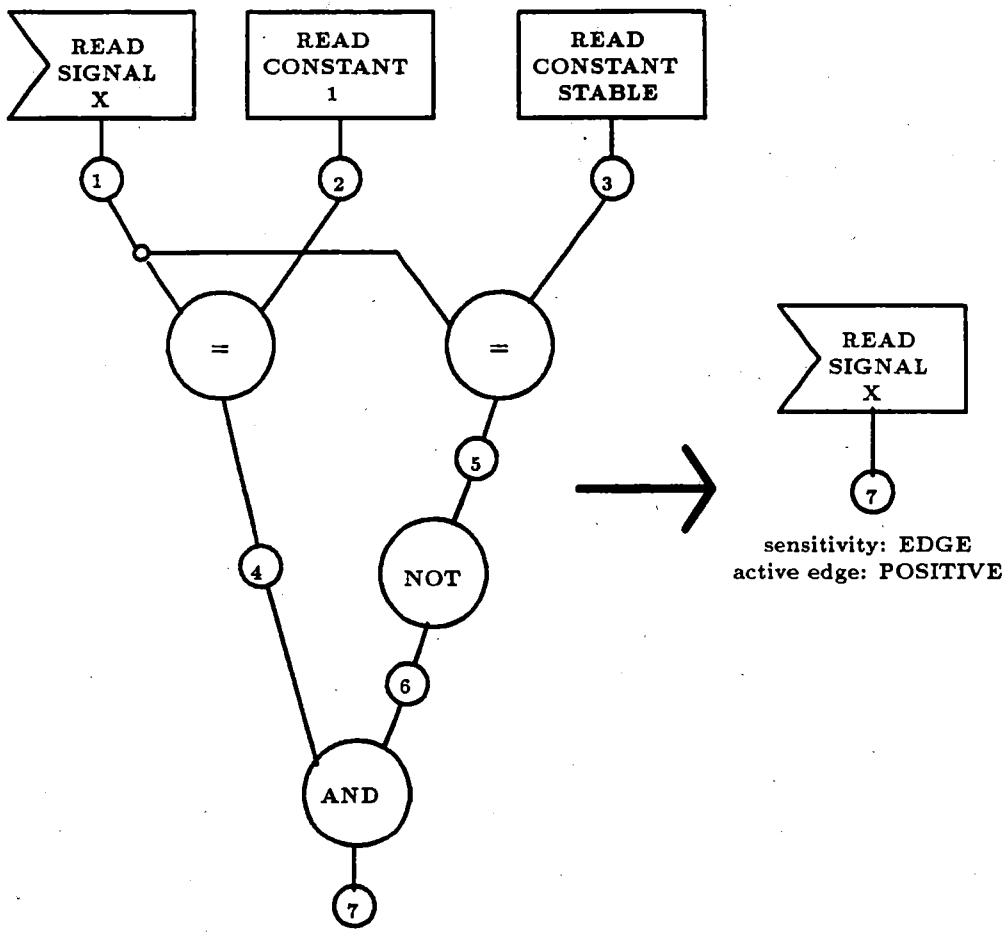


Figure 6.6: Graph Critic Optimization Rule

task simplifies the assignment of generic logic components to corresponding operation nodes in the flowgraph representation.

6.2.2 Graph Transformations

When appropriate, global transformations are applied to the flow graph representation. Flow graph transformations aid the synthesis process by facilitating the application of Design Compiler algorithms in the following situations:

- *Transforming descriptions which are not written using the preferred Structured Modeling guidelines for the intended target design model.* For example, a Functional design which is described using VHDL process constructs will initially be represented as a control flow graph with embedded data flow. Synthesis of such descriptions will yield a design that is of poorer quality than one which is generated from a concurrent data flow representation.
- *Making architectural tradeoffs in the design to be synthesized.* Due to area or speed constraints placed on the design, the designer may wish to evaluate the effects of isolating control logic or embedding this control logic in the data path. Isolated control logic will result when synthesizing a Behavioral description in VSS which will be mapped to a control unit/data path design model. Embedded control logic is produced when a Functional description consisting of conditional signal assignments is synthesized.
- *Identifying situations where resources can be shared under mutually exclusive conditions.* As mentioned earlier in the Design Synthesis Process chapter, one goal of synthesis is to extract the maximum amount of parallelism in a design in order to share resources. Resources can be shared under two conditions: (1)

they are not used in the same machine state, or (2) if they are used within the same state, they must not be used under the same condition (as determined by conditional branching in control flow or conditional signal assignment in data flow). While determining shareability in the former case is straightforward, the latter case presents difficulties in identification of shareable resources.

By applying transformations, the CDFG can be converted to a preferred representation that is efficiently processed by the Design Compiler so that a higher quality design will result. The transformations described below are examples of such optimizations which have been integrated into the VSS framework.

Control Flow to Data Flow Transformations

In order to determine the impact of graph transformations motivated by the the first two situations presented above, a package of CDFG transformations was developed [Gup91]. These transformations are applied to control flow constructs which have been annotated by the designer in the input description. Such statements are appropriately marked in the design representation such that the transformation package will process them as the CFG is traversed.

The following types of transformations can be applied:

1. *IF and CASE statements*

   This transformation will form a DFG section equivalent to a conditional signal assignment for every variable to which an assignment is made in any conditional branch of the construct. Such a DFG section consists of a CHOOSE-VALUE node that provides alternative data values to a WRITE node for the variable,

one value per conditional branch of the transformed control construct. The expression which selected the conditional branch in the control construct is used to select the corresponding data value which is assigned in that control branch. Figure 6.7 illustrates the application of this transformation.

2. *FOR LOOP statements*

   This transformation can be applied to the CFG section which represents a FOR loop with known iteration bounds. The transformation performs loop unrolling through replication of the DFG that represents the actions found in the loop body. For each copy of the loop body, references to the loop index are replaced with the appropriate value for that iteration. Data dependencies between iterations are introduced in the expanded flowgraph.

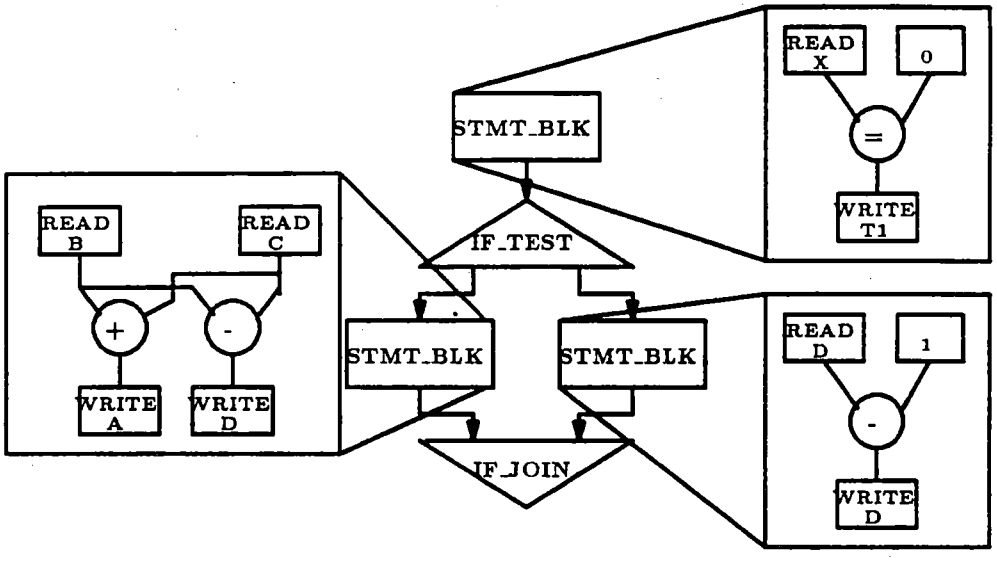
Upon completion of the transformations, a corresponding concurrent data flow representation will be constructed to replace the marked control constructs in the CDFG Design Representation.

Component Synthesis Algorithm

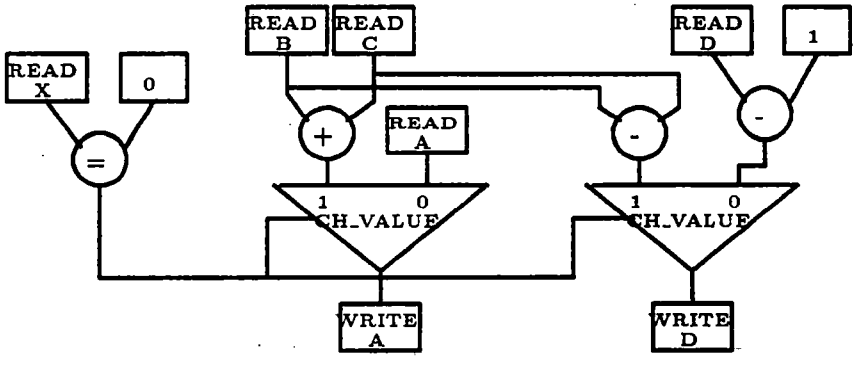
Functional descriptions describe the behavior of one or more RTL components; often, these components perform multiple functions (for example, an arithmetic logic unit, or ALU). These descriptions don't require scheduling; they are, in effect, *single state designs* in which the occurrence of an event signifies entrance into the state. Action(s) to be performed are determined via the selection of a function to be performed by some functional unit or a data value to be passed through an interconnect unit (multiplexor, bus). The Functional modeling style of Structured Modeling advocates the use of VHDL conditional signal assignment statements for these descriptions,

```
if (X = 0) then
  A := B + C;
  D := B - C;
else
  D := D + 1;
end if;
```

VHDL Input Description



Control Data Flow Graph Representation



Transformed Concurrent Data Flow Graph

Figure 6.7: IF Statement Transformation

where the target of the assignment is the output of the component. These statements often consist of the selection of one expression from among several alternatives based on the value of one or more conditions.

One method of synthesizing such a design would be to supply functional units which would perform the necessary operations to produce a value for each expression alternative and each condition expression; the appropriate value for assignment would then be selected based on the conditional value(s). This would often result in an inefficient design since most of the computed values would not be used. If it can be determined that there is mutually exclusive selection of only one of these alternatives at any time, then designs which share functional units across mutually exclusive expression alternatives can be evaluated. These designs would tend to show a reduction in area and improvement in the utilization of components.

In order to improve the quality of Functional designs, the following problems need to be addressed:

- Complex functions supported by RTL components (for example, ALUs) should be utilized by the synthesis process, even when the VHDL input descriptions contain language operators that do not correspond directly to these functions.
- Mutually exclusive operators within the input description should be mapped to the same component when the cost of such a mapping indicates an improvement in the design quality.

A Component Synthesis Algorithm (CSA) [RGB90] was developed to perform these optimizations on the Design Representation. CSA operates on a data flow graph (DFG) generated from a VHDL Functional description. Two optimization procedures are applied to the DFG:

1. *Functionality Recognition* - This procedure merges DFG expression subtrees into single function nodes for which there is an available component to perform that function. It is driven by the components available in the library and the functions they can perform. Component library specific information is maintained in a functionality table which stores the DFG expression subtree patterns corresponding to the available component functions.

   Functionality recognition solves the *functionality mismatch* problem. Language operators (or a sequence of language operators) are mapped to component functions. For example, the expression A+B+1 maps to the complex ALU function ADD-INCREMENT.

2. *Component Mapping* - This procedure solves the problem of merging mutually exclusive DFG operator nodes into multi-function operator nodes using a clique partitioning approach. Costs associated with the merging of DFG operation nodes are computed in terms of gate counts associated with the corresponding functional unit that will implement the DFG operation node. These costs functions take into account additional decoding logic that will be required to select the function of a multi-function unit as well as connection (multiplexor) cost.

Each operator node in the resultant DFG can now be mapped to an appropriate component from the supplied library (in this case, a GENUS generic component).

6.3 Design Compiler

The *Design Compiler* performs the central synthesis task of mapping the behavior captured in the CDFG Design Representation to a structure specified in the

form of a GENUS Partial Design Representation which implements the desired functionality. A block diagram of the Design Compiler subsystem is shown in Figure 6.8.

The Design Compiler consists of three major components: an *Allocator*, a *Scheduler* and a *Resource Binder*. Input to this subcomponent consists of the optimized CDFG Design Representation and user supplied allocation constraints. User specified allocation constraints are entered using a textual file which determines the number and types of resources (function units, registers and interconnect units) to be used by the Scheduler and Resource Binder modules. The constraint file is parsed by the *Constraint Input Parser*, and the appropriate components are instantiated in the GENUS Partial Design Representation.

As described in section 2.1, the Scheduler performs the assignment of operations to control steps given the constraints of a unit allocation. These state bindings are recorded through annotations made to the CDFG. As the Resource Binder creates or upgrades function, storage and interconnect units, entity, component class and component instance information is added to the GENUS Partial Design Representation. The binding of DFG operations, data accesses and DFG node interconnections to GENUS component instances and component connections are also recorded in the CDFG Design Representation.

Within the VSS framework, Design Compilation results are maintained in the GENUS Partial Design Representation and the annotated CDFG. In order to allow the designer to review these results in textual form, BIF state tables are generated by the *State Table Generator*. Results of Scheduling are reflected in the Operation Based State Table, while resource bindings are shown in the Unit Based State Table.

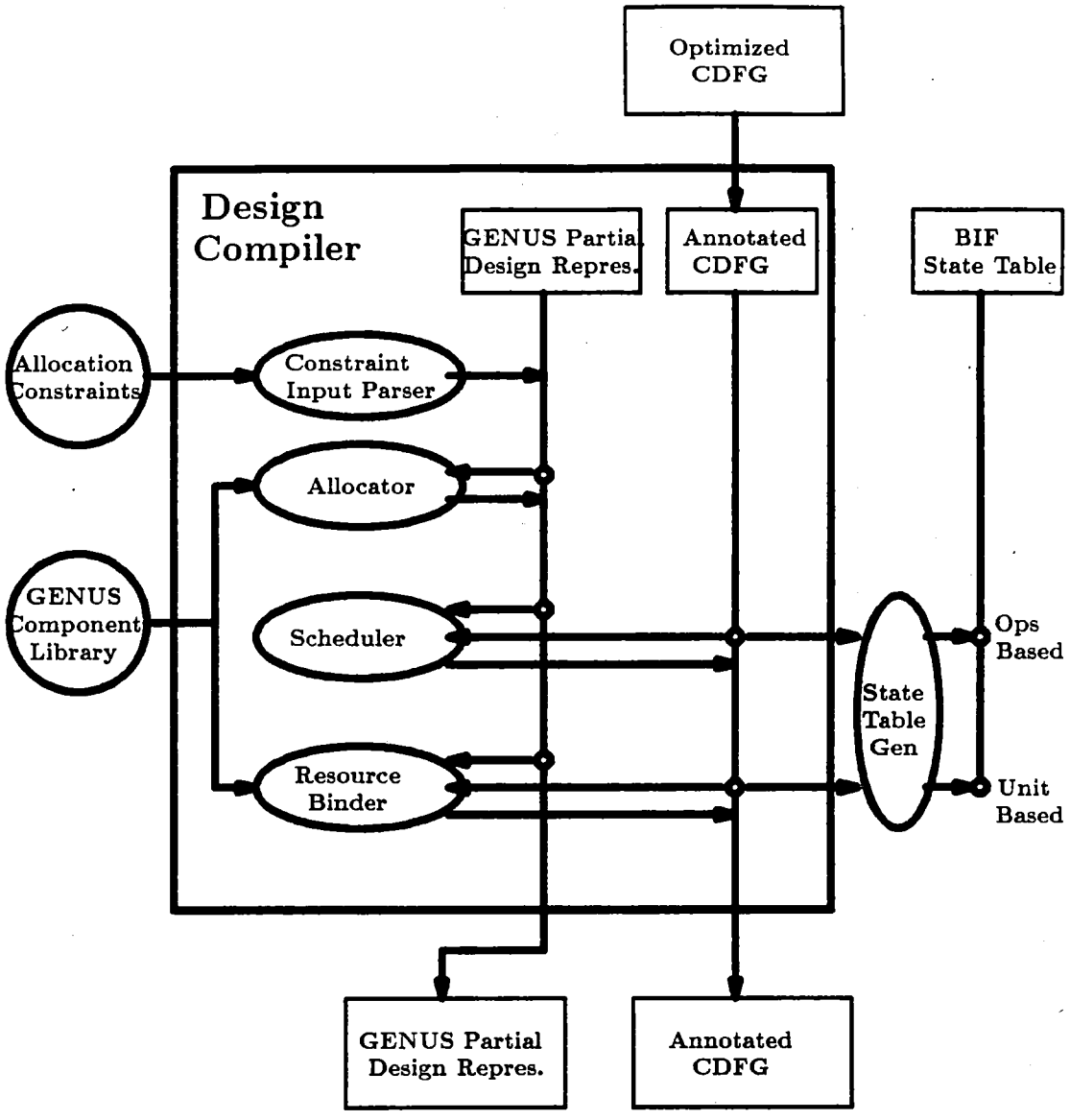


Figure 6.8: Design Compiler

The various Structured Modeling design models are processed differently within the Design Compiler. For example, in processing Behavioral designs, an explicit ordering of the allocation, scheduling and resource binding tasks is maintained, while the allocation and resource binding tasks for Functional designs are performed in the same procedure. The following subsections will detail the processing steps performed by the Design Compiler for each design model.

6.3.1 Combinational and Functional Design Compilation

The concurrent nature of the CDFG Design Representation for Combinational and Functional designs implies that there will be no opportunity to share hardware resources among operators. Unlike Behavioral designs where the Resource Binder performs a many-to-one mapping of operation nodes to a hardware component, synthesis of these designs involves a mapping of each DFG node to a single or combination of components available in the GENUS library. This underlines the importance of Representation Optimizations which are applied to minimize the number of operation and data access nodes in the DFG sections. The Graph Critic and Graph Transformations perform the majority of the optimization work on Combinational and Functional Designs.

Because concurrent descriptions can be considered single state designs in which one or more events trigger one or more actions in the data path, there is no concept of machine states as defined by a system clock. Consequently, the Scheduler module of the Design Compiler is not required for Combinational and Functional Designs.

Thus, the primary task of the Design Compiler in the case of Combinational and Functional designs becomes the allocation and binding of appropriate GENUS components to each DFG node. As each DFG node is processed, the Allocator extracts parameters for attributes such as bit width, a functionality list, and edge sensitivity and passes them to GENUS generators. The GENUS component library server instantiates and returns the desired component class (ALU, register, multiplexor, etc.) with the minimal functionality required. Instances of this component class are instantiated by the Resource Binder, and the mapping of DFG nodes to these component instances is annotated in the design representation. The GENUS Partial Design Representation maintains this structural view of the design.

As mentioned in the Graph Compilation section, the hierarchy of the VHDL input description as defined by the nesting of block statements is reflected in the hierarchy of the CDFG Design Representation. One problem a hierarchical representation poses to the Design Compiler is that of multiple assignments to the same signal. If this signal is to represent one storage element, mapping each WRITE node to an unique GENUS register (in order to maintain a structural hierarchy corresponding to that of the Design Representation) will produce redundant hardware (and most likely, an incorrect design). Conversely, creating a "flattened" structure by mapping all accesses to a signal to the same component without first restructuring the Design Representation may lead to an inefficient and/or incorrect design.

To avoid these problems, the designer can direct the Resource Binder to process the Design Representation in one of two modes:

- *flattened*
- *hierarchical*

In flattened mode, all DFGs corresponding to VHDL blocks as specified in the input description are combined into a single DFG. This DFG is then mapped to GENUS components.

When the hierarchical mode is selected, the hierarchy of the input description is preserved; each block is mapped to GENUS components individually. Here, it is assumed that all assignments made to a signal occur within a single block. A VHDL entity/architecture pair is created for each DFG block. The partial design hierarchy is maintained using the entity object of the GENUS Partial Design Representation as described in section 5.2.1.

6.3.2 Register Transfer and Behavioral Design Compilation

6.3.2.1 Allocation

The Allocation module allows the designer to input the number and types of resources (function units, registers and interconnect units) to be used when synthesizing a register transfer or behavioral design. The following component attributes can be specified: operation class, operation types, bit width and operation delay. The operation delay can be expressed in terms of fractions or multiples of a clock cycle; this allows for chaining of operations in the same machine state or multi-cycle operations. These units are entered into the GENUS Partial Design Representation, and this information is accessed by the Scheduler module in order to determine the available units.

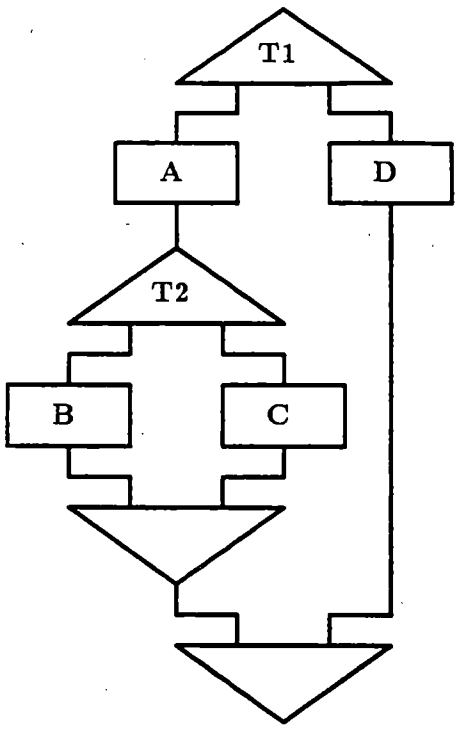
6.3.2.2 Scheduling

Two schedulers have been developed and integrated into the VSS framework: a *Mobility Based Scheduler* and the *Percolation Based Scheduler.*

Mobility Based Scheduler

The primary scheduler used in VSS is a variant of the SLICER [PG87] scheduler which calculates the as-soon-as-possible (ASAP) and as-late-as-possible (ALAP) schedules in order to determine the range of machine states to which an operation can be assigned. The scheduler actually consists of two parts: a macro scheduler which traverses the CFG and assigns states to control point nodes, and the SLICER scheduler which is applied to all STMT_BLKs encountered. The first state to be assigned to the STMT_BLK is passed to the SLICER scheduler, along with the DFG nodes in the STMT_BLK, and the scheduled STMT_BLK is returned.

Two techniques can be employed for the assignment of states across conditional branches. Figure 6.9 presents a simple conditional branch example with the schedule produced using each of these techniques. The first technique assigns unique states in each conditional branch. This results in a control strategy in which a conditional test is made in one machine state, and based on the result of this test, a branch is made to the first state of the appropriate branch. No actions are performed in this branching state. The advantage of this scheme is that in the current state, no knowledge is required of previous conditional values which resulted in the entry into this state. A disadvantage is that an overhead in the number of states is required, since operations in different conditional branches will never share the same state assignment, even though they are mutually exclusive. Alternatively, each conditional branch can be



STATE	COND	ACTIONS	NEXT
1	T1 = 1 T1 = 0		2 6
2	TRUE	A	3
3	T2 = 1 T2 = 0		4 5
4	TRUE	B	7
5	TRUE	C	7
6	TRUE	D	7
7			

UNIQUE STATES ACROSS BRANCHES

STATE	COND	ACTIONS	NEXT
1	T1 = 1 T1 = 0	A D	2 3
2	T1 = 1 & T2 = 1 T1 = 1 & T2 = 0	B C	3 3
3			

SAME INITIAL STATE IN EACH BRANCH

Figure 6.9: State Assignment Across Conditional Branches

labelled beginning with the same state assignment. This approach offers the savings in the number of machine states with the cost of increasing the complexity of the condition evaluation associated with each mutually exclusive action in the same state.

Percolation Based Scheduler

A Percolation Based Scheduling algorithm [PLNG90] has been integrated into the VSS system framework. This scheduler is used to perform scheduling on VHDL input descriptions which consist primarily of loops. Percolation scheduling utilizes techniques which compact flow graphs beyond basic block (straight line code segment) limits, potentially resulting in an order of magnitude speedup over serial execution. In order to schedule under resource constraints, the *optimal schedule* (without constraints) is first determined. Next, heuristics are added to map the optimal schedule onto a system with limited resources. Starting from the optimal schedule is a key feature of this approach because it provides a realistic lower bound to the scheduler which can be used to tune the heuristics employed to determine the resource constrained schedule.

Percolation Scheduling is a system of semantics-preserving transformations that convert an original *program graph* [1] into a more parallel one. Its core consists of 4 transformations (Move-op,Move-cj,delete and unify) which are defined in terms of adjacent nodes in the program graph. The transformations are atomic and thus can be combined with a variety of guidance rules (heuristics) to direct the optimization process. Repeatedly applying the transformations allows data-independent operations

[1] Here, program-graph is an extension of the conventional notion of control-flow graphs, in that a node may contain one or more operations, including conditional-jumps. The program-graph corresponds to an execution model in which all operations in a node can execute in parallel. If conditional jumps are present in the node, their evaluation combines to yield the unique successor node that executes next. The exact mechanisms by which control-flow is determined in a node is unimportant in this discussion. The interested reader is referred to [EN89].

to "percolate" towards the top of the program graph from the various parts of the code—hence the name Percolation Scheduling. Operations are packed together in nodes (states) as PS is applied to a program graph, thereby yielding more parallel code. The details of the transformations deal with maintaining the semantic integrity of all affected paths. Detailed discussions of percolation scheduling and its extension to multicycle and pipelined operations can be found in [AN88] and [PLNG90].

The optimal schedule without constraints is obtained using the OPT procedure. OPT is a loop parallelization technique for a loop which does not contain conditional jumps in its body. It applies to both unicycle and pipelined operations. The idea behind OPT is simple: the loop is incrementally unwound. As new iterations are brought in, operations are allowed to migrate upwards (without regard to iteration boundaries) in the *expandable program-graph* formed by the unwinding of the loop. This migration is only limited by the data-dependencies between the operations. If all operations in the loop body are either involved in some data-dependence cycle, or depend on operations involved in such a cycle, then a repeating *pattern* will provably emerge after a polynomial (and in practice small) number of iterations have been parallelized in this manner. This pattern will then repeat, as long as more iterations are forthcoming, so that in effect more unwinding of the loop will *not* yield further parallelism. Replacing the original loop body with the pattern discovered (proper *startup* and *wind-down* code is trivially derived in the process of finding the pattern) then yields a compact expression of the *maximum* parallelism available in the original loop, subject to the given data-dependencies and operation latencies.

The operation of the Percolation Scheduling algorithm can be summarized as follows:

1. *Find the optimal schedule.* Scheduling begins from the optimal schedule (the schedule without resource constraints). This schedule is derived by the OPT algorithm explained above.

2. *Find each operations' mobility and reorder operations.* If the number of operations in one state exceeds the number of available resources, some operations have to be delayed. The mobility [PG86] of each operation has been chosen as the criterion for delaying operations. Operations with higher mobility are delayed first because their delay will not necessarily "stretch" the schedule. After finding the mobility of each operation, the operations are sorted in non-decreasing order. The last operations in this order are the first to be delayed (if necessary).

3. *Make reservations.* The scheduler deals with two kinds of machines: pipelined and non-pipelined. In the pipelined version, operations are scheduled assuming that the execution unit can handle a new operation every cycle (state). In that way, it is not necessary to wait for the execution unit to flush before issuing the next operation. The non-pipelined version requires such a wait; therefore, states are reserved so that latency times are not violated. This procedure is responsible for the insertion of "empty" states where needed.

4. *Adjust state l.* This procedure delays operations from state l due to resource constraints. An operation which has to be delayed is moved to the next available state in the program.

5. *Percolate operations from l's successor.* After (possibly) delaying some operation from state l, there is a possibility that some of the operations from l's successor will percolate up. This percolation of operations is due to the addition of "new" states between the "original" l and its successor. As a result,

there are cases in which we can hoist operations from l's successor. Operations are moved up if data dependencies are preserved and resources are available in earlier states.

6.3.2.3 Resource Binding

Two algorithms which perform the resource binding task of high-level synthesis for Behavioral designs can be invoked by the VSS Design Compiler: a *Frequency Based Binder* and a *Gain Based Binder.* These algorithms trade off the extent of the design state space being examined at one time (from a single state through the entire state space) with the execution time of the algorithm and the resultant design quality. The main goal of the Frequency Based binder is the assignment of operators which share similar connection patterns (input and output) to the same functional unit such that the amount of interconnect is minimized. The Gain Based Binder weighs the effects of making a binding which offers local gain in the current processing step on the potential gain that can be achieved upon completion of the design after this binding has been made.

Frequency Based Binder

Algorithm Overview

The Frequency Based Binder creates input/output connection patterns for each operation in the DFG. A usage frequency (a measure of the reuse of common connection patterns) is used to establish the order in which patterns will be considered for binding to units. Binding costs consider the tradeoffs of adding functionality to existing components versus instantiating new components.

A DFG section associated with a STMT_BLK which has been annotated with state bindings is used as the input to the Frequency Based Binding algorithm. *Patterns* are created for each operation. These patterns are tuples consisting of the following information:

- assigned state of operation
- operation type
- inputs
- outputs
- control flow condition under which operation is performed

The input/output connection patterns for each operation in the flow graph are examined to compute a *usage frequency* for each pattern. This measure is used to determine the order in which operations are to be considered for binding to functional units. Candidate patterns are sorted by their usage frequency; those with a higher usage frequency correspond to operations which will tend to reuse existing connections. In this manner, the most frequently used components will be allocated first, and a larger number of operations will be bound to them.

After an operation is selected for binding, an appropriate unit to which this operation is to be bound is chosen. A binding cost is computed for each existing unit that is available during the operation's assigned time step. This cost consists of two components:

1. *functionality cost* - the cost (in terms of area) associated with adding the desired functionality to the unit such that the operation under consideration can be performed by that unit.
2. *connection cost* - an estimate of the interconnection cost based on the rough placement of datapath components in the eventual layout. This cost function is described in more detail in the following section.

One alternative which is always evaluated is to create a new unit; if no existing unit is available, a new unit is instantiated. Binding is then performed by updating the Partial Design representation with the necessary unit and connection information. The above procedure iterates until all patterns are bound.

The allocator/binder generates a unit based state table reflecting the binding of operations to units and the required control that is to be supplied to all components in each machine state. In addition, the interconnected register transfer structure of the datapath is produced.

Figure 6.10 presents an outline of the Frequency Based Binding algorithm.

Pattern Creation

The following processing options affect the types of patterns created and the statistics generated by the pattern creation function:

1. *operator commutativity* - whether or not commutativity of operations is to be considered affects the common inputs count.
2. *register sharing* - if storage units are to be shared as function units are, patterns are created for READ/WRITE variable accesses.
3. *operator classes* - this parameter determines if units of different classes (e.g., LOGICAL (and,or,nand,nor,not), ADDING (+,-), MULT, DIV) are to be considered mergeable into the same functional unit.

Usage Frequency Function

The usage frequency is computed by a weighted sum of the frequency of the operation type and the number of common connections for a particular operator. The

```
allocate_and_bind()

{
   create_pattern_list();
   pattern = first pattern on list;
   while (there are patterns to bind)
      {
      j = assigned_state(pattern);

      /* Examine all existing components and compute the cost for */
      /* binding of current operation to available components.    */

      for (i = 1 to num_of_units)
         if (unit i available in state j)
            cost[i] = functionality_cost() + connection_cost();
         else
            cost[i] = MAX_COST;

      /* Compute cost of creating a new unit. */

      cost[num_units + 1] = new_unit_cost() + connection_cost();

      /* Select unit with smallest associated cost as candidate   */
      /* for binding.                                             */

      unit = min_cost_unit();
      update_partial_design();

      /* process remaining patterns */

      pattern = next pattern on list;
      }
}
```

Figure 6.10: Frequency Based Binding Algorithm

function used to compute the usage frequency is shown in Figure 6.11. The *number_of_same_ops* function returns the total count of operations of the same type in the design specification (for example, addition operations). The *number_of_pattern_matches* function returns the number of patterns which match the current pattern in the number of places specified by the second argument (where places are defined as input1, input2 or output of the operation).

---

usage_frequency(op,pattern)

```
{
   w1 = 0.7;
   w2 = 0.3;
   w3 = 1;
   w4 = 2;
   w5 = 3;

   uf = (w1 * number_of_same_ops(op)) + (w2 *
((w3 * number_of_pattern_matches(pattern,1)) +
 (w4 * number_of_pattern_matches(pattern,2)) +
 (w5 * number_of_pattern_matches(pattern,3)));
   return(uf);
}
```

Figure 6.11: Usage Frequency Cost Function

Microarchitecture Connection Cost Function

The microarchitecture connection cost function currently used by the Frequency Based binder is shown in Figure 6.12. This function estimates a point-to-point interconnect cost for a specified source to destination connection. An arbitrary cost is assigned for connections in which either the source or destination unit is not yet bound. If other connections exist to the destination input, an estimate of the multiplexing cost for multiple inputs is included in the cost computation.

```
m_arch_connection_cost(src,dest,input)
{
   /* find units associated with src, dest fg nodes; if units     */
   /* have not yet been bound, find_src_unit returns -1           */

   src_index = find_src_unit(src);
   dest_index = find_dest_unit(dest);

   if (src_index == -1 || dest_index == -1)
      /* assign UNBOUND_COST */
      cost = UNBOUND_CONN_COST;
   else
      if (connection_exists(src_index,dest_index))
 cost = 0;
      else
 if (dest has no other input connections)
    cost = SINGLE_CONN_COST;
 else
    /* If adding a new conn requires a new multiplexor    */
    /* with additional inputs, add a MUX_UPGRADE_COST.    */
    if (# of dest input conns % 2 == 0 ||
                # dest input conns == 1)
       cost = MUX_UPGRADE_COST + SINGLE_CONN_COST;
    else
       cost = SINGLE_CONN_COST;
   return(cost);
}
```

Figure 6.12: Microarchitecture Connection Cost Function

Layout Architecture Model

A strip layout architecture is assumed for the data path. The component width (pitch) for each bit slice is fixed; the height of the component may vary. Each component has a fixed number of tracks (13) running vertically in metal2 over each bit slice of the component. These tracks are used to route the interconnect between components. As components are created, they are placed in a column. This column is arranged by decreasing bit width of the component (the largest components are at the top of the column).

Layout Connection Cost Function

The layout connection cost function is shown in Figure 6.13. It seeks to minimize the track density across the entire design. A count of tracks which cross each cell boundary is maintained.

---

layout_connection_cost()

```
{
   for (i = each possible position of new component)
      {
      for (j = each existing component)
 density[j] = number of tracks crossing lower boundary;
      cost[i] = MAX(density[1..<# components>]);
      }

   return(MIN(cost[1..<# components>]));
}
```

Figure 6.13: Layout Connection Cost Function

As a new component is to be inserted in the sorted data path column, each possible placement of the new component is evaluated. The track density at each component's lower boundary is computed by counting the number of tracks used to make connections to components in lower rows of the column. After all possible placements are evaluated, the minimum cost is returned, and the component is placed in the row of the layout which yields this minimum connection cost.

The procedure is physically restricted by the fixed number of tracks per bit slice; when this limit is reached, the data path must be partitioned. This partitioning is not done within the allocation process. It is handled later within the SLAM [WCG90] partitioner. The cost function can be extended to include a penalty associated with

this condition in order to influence the allocator/binder to seek to increase the number of units in order to minimize track density.

Gain Based Binder

Algorithm Overview

The Gain Based Binder performs the binding of variables and operations in a behavioral description to storage, functional and interconnect units in a data path. The behavioral description is represented by a data flow graph which reflects the data dependencies and operation sequence inherent in the description. It is assumed that scheduling has been performed.

A data flow graph annotated with state bindings is used as the input to the gain based binding algorithm. Figure 6.14 shows a behavioral description and a corresponding flowgraph representation which indicates the results of scheduling.

A clique partitioning approach is used on the vertices of a *compatibility graph* G (with V vertices and E edges) into K (<= V) disjoint cliques which cover the graph. Figure 6.15 presents the compatibility graph corresponding to the behavioral description shown above.

Each vertex of the compatibility graph represents an operation or variable access in a data flow graph. An edge between vertices indicates that the operations (variables) represented by the nodes can be bound to the same hardware unit. Operations are compatible if they are of the same operation class (for example, if addition and subtraction are of operation class ADDING and multiplication is of class MULTIPLYING, addition and subtraction are mergeable while addition and multiplication are not) and if they are not assigned to the same control state. Variables

```
u1 := u * dx;
u2 := 5 * x;
u4 := u1 - u2;
u6 := u * u4;
```

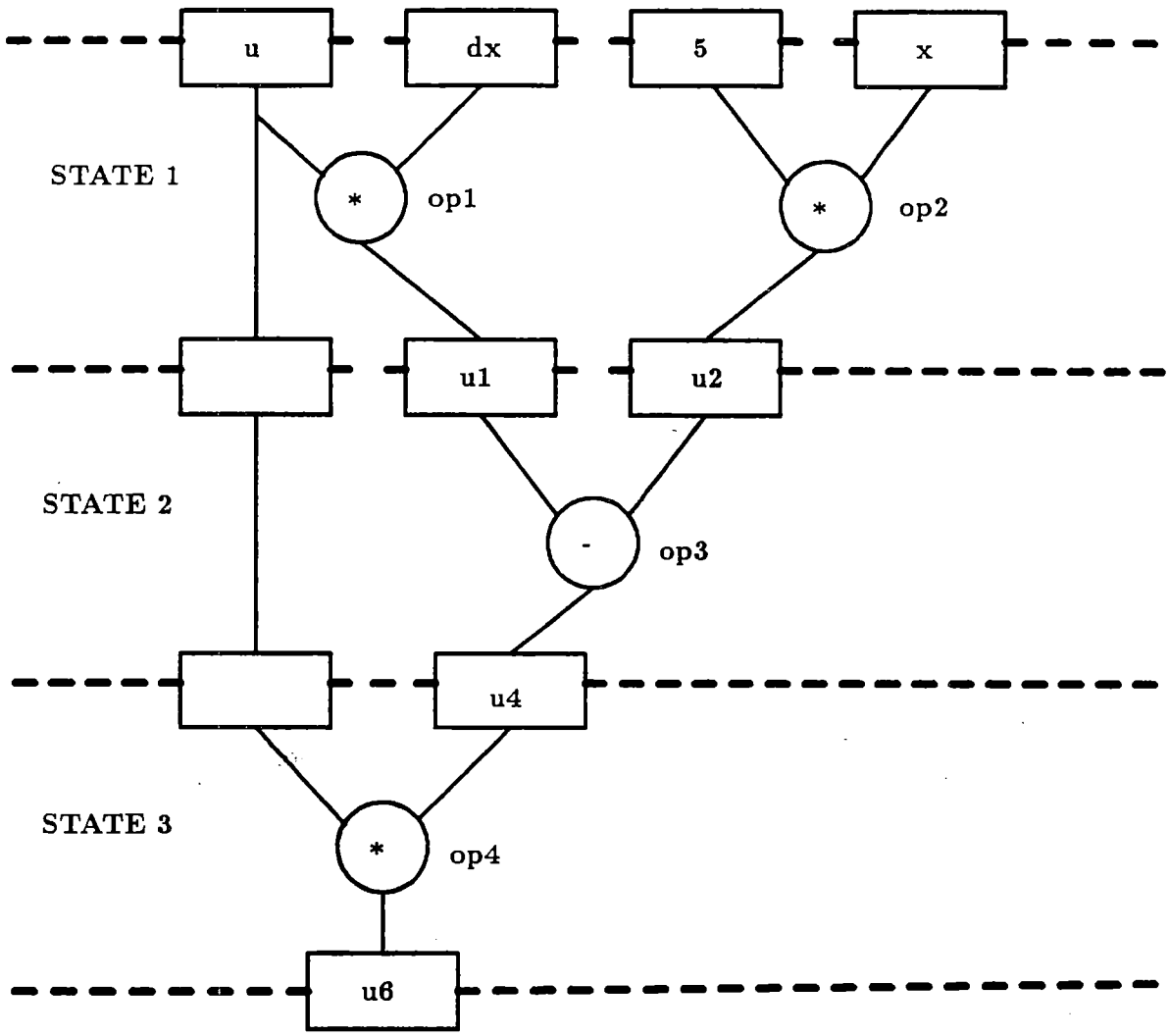


Figure 6.14: Gain Based Binding Example

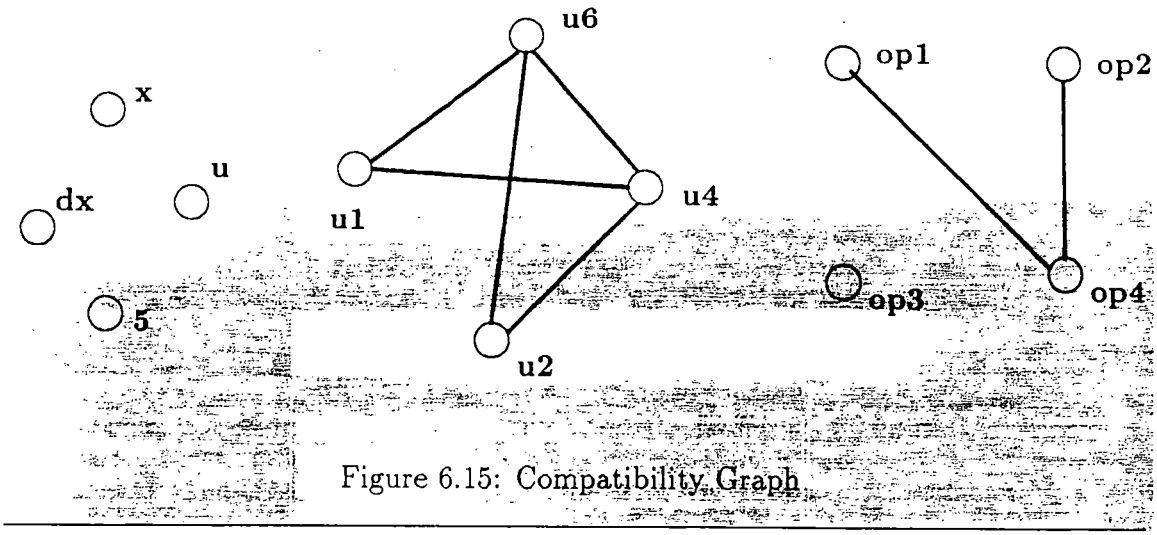


Figure 6.15: Compatibility Graph

are considered compatible (mergeable) if their lifetimes do not overlap. Thus, the compatibility graph can be represented as separate graphs, one for operations and another for variables, or it can be thought of as a single graph in which there will never be edges between operation and variable nodes.

The cliques formed for the compatibility graph represent function units (storage elements) in the partial design to which the member operations (variables) of the cliques are bound.

A *gain value* is associated with each edge in the compatibility graph. This gain value indicates a potential connection cost savings in the partial design if the elements represented by the vertices connected by the edge are assigned to the same hardware unit (the gain value may be 0,1,2). For example, if two operations have a common input and are mergeable, the gain value associated with the edge in the compatibility graph is 1. Figure 6.16 illustrates this concept by showing the potential gain that can be achieved if op1 and op4 are merged. Since the operation nodes have

a common data input, one multiplexor input is saved. This savings is annotated on the compatibility graph as indicated by the weight of 1 on the edge between nodes op1 and op4.

A *clique forest* is constructed to generate all possible cliques. The clique forest is constructed in a bottom-up fashion; the first (leaf) level of the clique tree consists of the single nodes of the compatibility graph. The second level consists of nodes which represent a compatible pair of leaf nodes; each leaf node of the pair becomes a child node of these nodes. Subsequent levels of the clique forest are constructed by examining each clique of the previous level. The leaf nodes compatible with each element of the clique are examined. If a new leaf element is compatible with all elements of the clique, a new clique is added. This procedure is repeated until all cliques are generated. Figure 6.17 shows the clique forest generated for the example presented above. The first row represents the leaf cliques, the second row compatible pairs of leaf nodes, and the third row, compatible triplets. Arcs between the nodes in this diagram represent the derivation of a larger clique by adding one new leaf node to a clique from the previous level.

A gain value for each clique is then computed. Like the gain value associated with edges in the compatibility graph, this metric reflects the possible cost savings in the partial design which would result if the hardware element represented by this clique is used.

The cliques are then sorted by their gain value. Several strategies may be employed to select a *seed* clique which begins the selection process for covers (sets of disjoint cliques). For example, the cliques containing the largest number of elements or those with the largest associated gain values may be used as seeds. Once a seed is

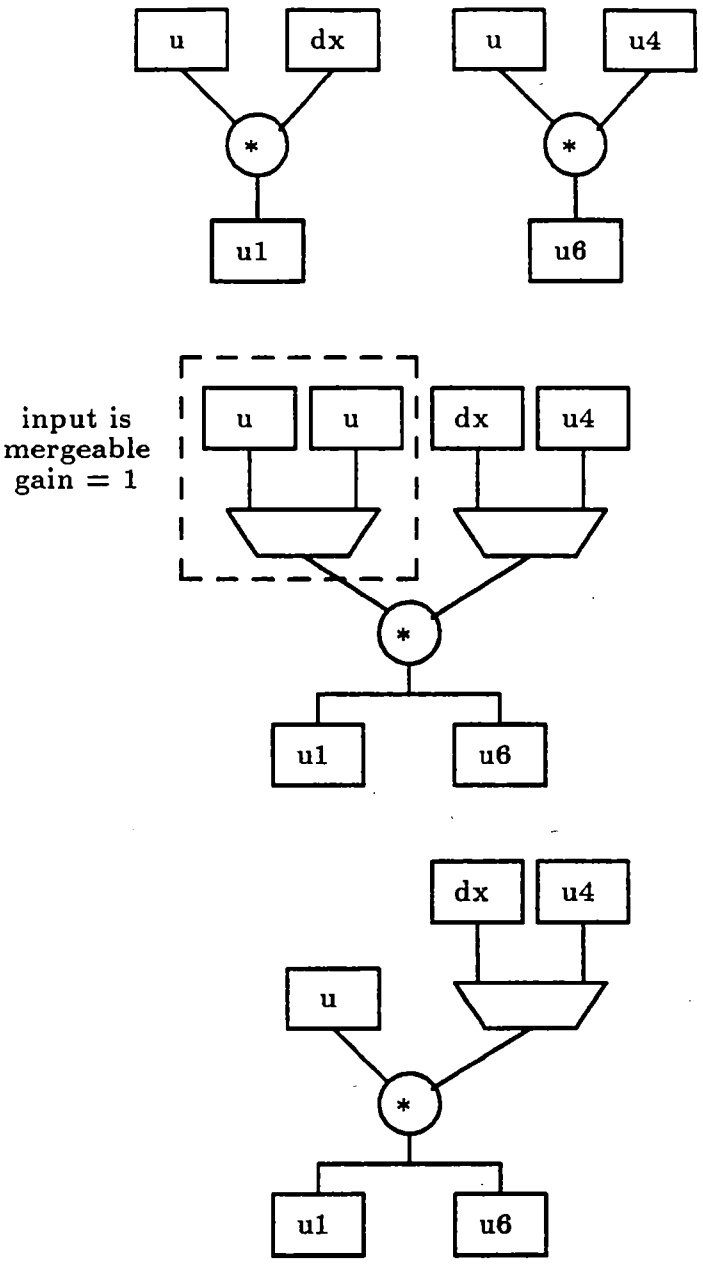


Figure 6.16: Computation of Gain Value

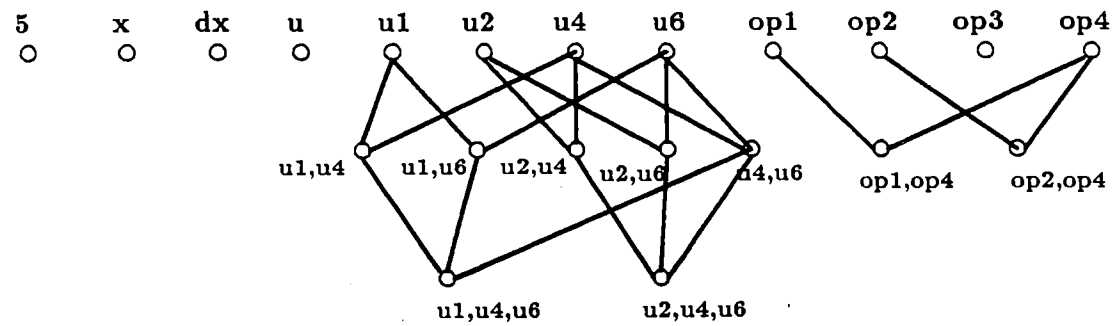


Figure 6.17: Clique Forest

selected, the remaining available cliques which do not contain an element common to the seed are examined to form a complete cover of the compatibility graph vertices. Each cover becomes a partial design binding alternative.

Algorithm Details and Alternatives Investigated

The following parameters had an influence on the decisions made during the clique formation procedure and the selection of cliques used to form clique covers of the compatibility graphs:

1. *Determination of Cliques.* The degree to which operation nodes were to be considered compatible depended on the determination of which operations could be merged into the same clique (and ultimately, the same functional unit). This parameter could be varied from the extremes of allowing the merging of any two operators to the more realistic level which allows merging of selected arithmetic/logic operations found in typical ALUs.

2. *Selecting a seed clique.* The results of cover formation were influenced greatly by the selection of the seed clique. Intuitively, it would seem preferable to select a seed clique which contained the largest number of elements, or had the largest associated gain values. Due to ties in the cost function values at critical decision points in the procedure, the results obtained did not always validate this assumption.

3. *Cover formation strategies.* The following cover formation strategies were examined:

   - *first fit* - The first clique which satisfied the selection criteria at the current processing step was used.
   - *user selected partial cover* - The designer was allowed to select one or more cliques from the clique forest. The Gain Based Binder would then select the remaining cliques necessary to complete the compatibility graph cover.
   - *exhaustive search* - The algorithm attempted to examine all possible compatibility graph covers. Naturally, the computational expense of this option was prohibitive in most cases.

4. *Cost functions.* Several cost functions were used to evaluate each design produced by the pairing of a variable cover with an operator cover. These functions included:

   - maximum gain - a summation of the gains associated with each clique belonging to the selected cover.
   - number of multiplexors and multiplexor inputs
   - function unit area cost

6.4 Control Logic Compiler

Upon completion of the Design Compilation phase for Behavioral designs, a BIF unit based state table is produced which serves as a behavioral description of the controller. In order to generate either a boolean equation description (in the form of VHDL concurrent signal assignment statements) or control unit structure (in the form of a VHDL structural description) in the current VSS system framework, a Control Logic Compiler module is invoked. This tool generates a VHDL dataflow description from the BIF state table. A second pass through the VSS system targeted to the Functional design model will produce a structural description consisting of logic gates and a state register. Either specification of the control unit is incorporated into the GENUS Partial Design representation for the design.

6.5 Interface to Logic Synthesis

In order to pass the results produced by VSS to the MILO system [VG88] technology mapping and optimization, the GENUS Partial Design Representation must be captured in textual form. The selected interchange format is a VHDL structural netlist. In addition, the coupling of MILO to the Intelligent Component Database (ICDB) [Che90] requires that the netlist must be preprocessed in order to install the design in the database and provide links to the necessary component information required by the logic synthesis tools. Figure 6.18 shows the modules required for integration of VSS with the MILO microarchitecture and logic optimization tool.

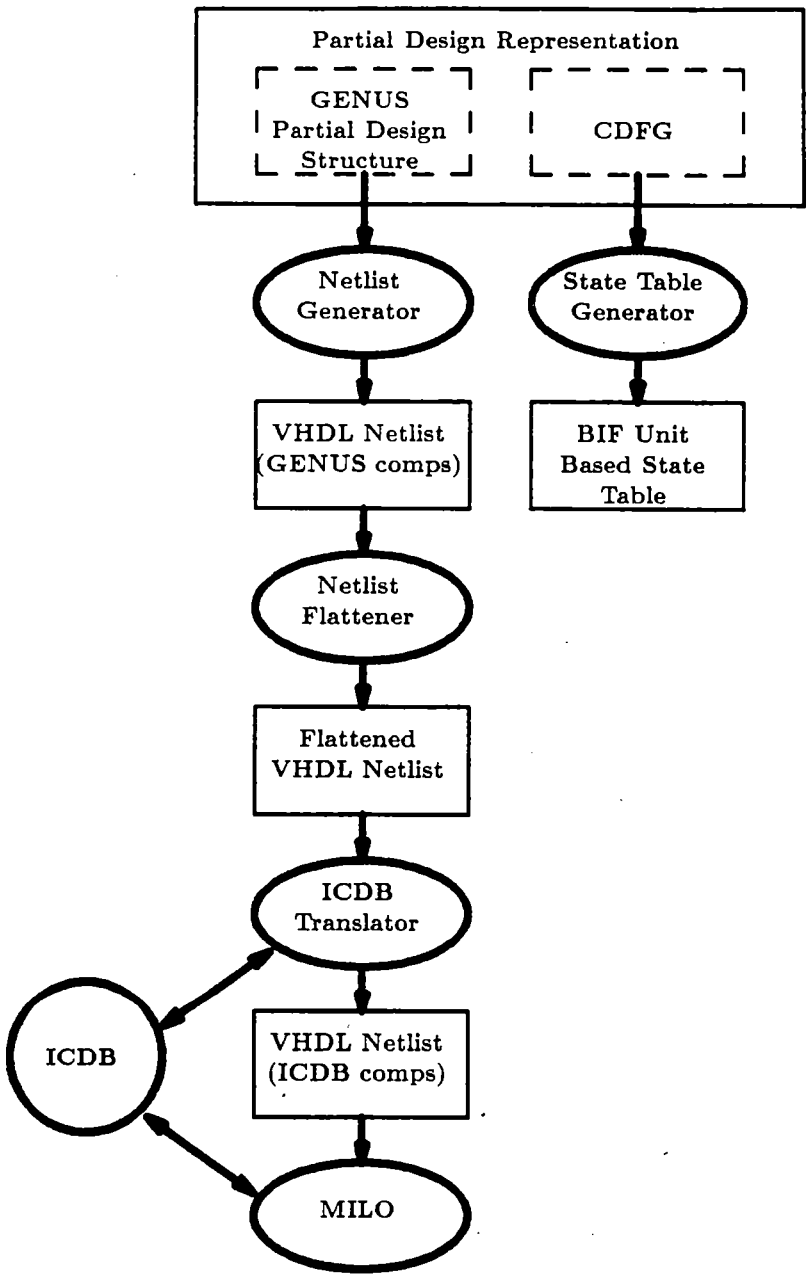


Figure 6.18: VSS Interface to Logic Synthesis

6.5.1 Netlist Generator

The Netlist Generator processes the GENUS Partial Design Representation in order to generate a text file for the VHDL structural netlist specification of the synthesized design. A VHDL entity/architecture pair is used to represent each level of the design's structural hierarchy. If the netlist contains hierarchy, a VHDL configuration statement is generated to link components in higher levels of the hierarchy with their corresponding decomposition as defined in lower level entity/architectures.

6.5.2 State Table Generator

The BIF State Table Generator traverses the control flow graph and embedded data flow graphs to produce either an operation based or unit based state table. Each VHDL process which is synthesized to a control unit/data path structure will have BIF tables generated for the behavior of the control unit.

6.5.3 Netlist Flattener

Since the MILO system can currently process only a single level of hierarchy in the input netlist, the design produced by VSS must first be flattened. This is accomplished using a Netlist Flattener. Given a hierarchical VHDL netlist, all complex components are replaced with their equivalent structure in terms of the lowest level components, in this case GENUS generic components. The output of the flattener is a single entity/architecture VHDL structural description.

6.5.4 ICDB Translator

Some VHDL structural descriptions produced by VSS will consist either of a structural hierarchy or a mixture of structural components and behavior of components. For example, a Functional design processed by the Component Synthesis Algorithm will generate a behavioral description of a random logic component which provides the function select lines for a multi-function unit. Similarly, the VHDL description of a control unit produced by the Control Logic Compiler might be present in the VHDL netlist generated by VSS. For each component found in the netlist, the MILO system requires that a boolean equation level description of the component must be present in the Intelligent Component Database (ICDB).

In order to enter these component behavioral descriptions in the database and identify the components with their corresponding descriptions, the ICDB Translator is invoked. This module will accept two input formats: either a flattened netlist produced by the Netlist Flattener, or a netlist with at most two levels of hierarchy, the topmost level consisting of a VHDL structural description with leaf levels (such as the function select logic or control unit components) described using VHDL concurrent dataflow statements. A modified VHDL netlist is produced which renames components so that they can be appropriately accessed through ICDB queries. Appropriate ICDB calls are made to generate the behavioral descriptions corresponding to each component necessary for processing by MILO.

6.6 Simulation Interface

VHDL input descriptions have been simulated using the Vantage Analyst commercial VHDL simulator [Van90]. At present, input test patterns are generated manually. A post-processing step adds VHDL models for the GENUS components used in the VHDL structural netlist description produced by VSS synthesis. The structural description can then be exercised with the same test pattern stimuli to verify the functional correctness of the synthesized design.

6.7 User Interface

A graphical display under either the Suntools or X Windows environments provides the capability to view results generated by the synthesis process such as hierarchical CDFGs and VHDL structural netlists. The utility accepts two input formats: either a file containing a textual "netlist" format of a CDFG, or a VHDL structural netlist file. Options such as node expansion, window panning, highlighting of node sources and destinations and zooming allow the designer to visually examine the Design Representation created by VSS.

Figure 6.19 shows a display generated by this utility.

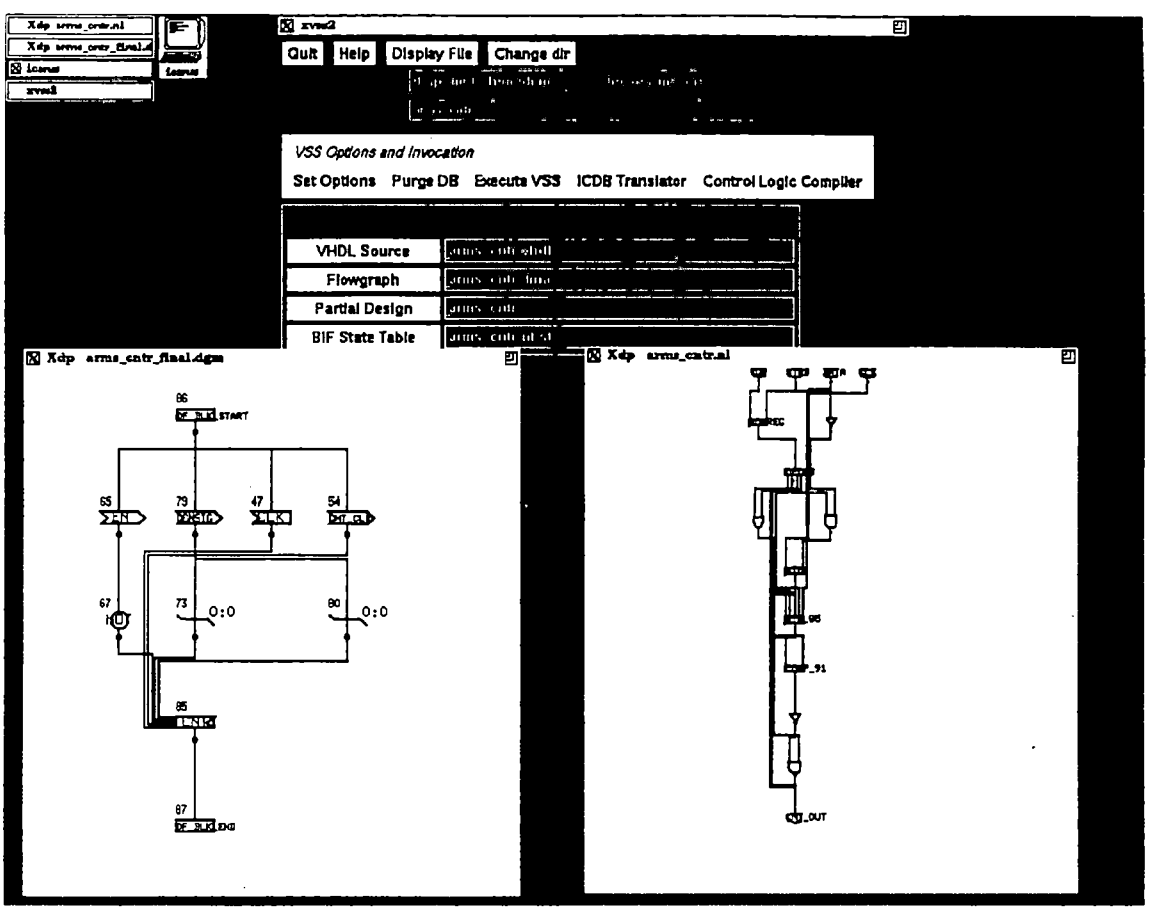


Figure 6.19: Flowgraph and Netlist Display Utility

Chapter 7

Experiments

This chapter presents experiments performed using the VHDL Synthesis System (VSS) described in the previous chapter. Table 7.1 lists the benchmarks which were used to verify the operation of VSS and validate the modeling guidelines of Structured Modeling. A brief description of each benchmark is provided which indicates the variety of designs which can be synthesized by VSS. In addition, a count of VHDL source lines in the input description is specified which gives an indication of the complexity of the benchmark.

The experiments listed in this table were used to demonstrate the effect of different modeling styles on the quality of the design produced by VSS. Results produced for four representative benchmarks (Rockwell counter, DRACO, AM2910 and 8251 USART) are presented in this chapter. Several alternative VHDL models are examined for each benchmark, and it is shown that when applicable Structured Modeling considerations are applied to models which do not comply with the standards, the quality of the design is improved. Another aspect to be considered in these experiments is a comparison of the human versus synthesized designs, since the examples

Benchmark	Description	VHDL source lines	
		Functional	Behavioral
Mark1	simple CPU		30
HAL	diff. eqn. computation		35
Rockwell counter	count sequence	40	42
Elliptic filter	DSP data path		40
Bus interface	microprocessor peripheral	45	
Booth multiplier	multiplication algorithm		54
FACET	data-dependent data ops		55
Armstrong counter	up/down counter to limit	92	
AM2910	microprogram controller	165	260
AM2901	bit-sliced ALU	333	
Multi-process arbiter	bus arbitration	440	267
DRACO	peripheral interface chip	845	657
8251	USART		953

Table 7.1: Benchmarks Synthesized by VSS

examined in detail model commercial or application specific circuits. A third motivation for these experiments is to observe the effects on the quality of the synthesized design when using a Functional versus a Behavioral modeling style.

7.1 Rockwell Counter

This experiment was conducted as a part of a case study which investigated the design process and synthesis tools used in the UC Irvine CADLAB design environment [GLVW90], of which VSS is a part. The benchmark was supplied by Rockwell International as a member of the Silicon Research Consortium (SRC).

Although this benchmark is of minimal complexity, we sought to investigate the following objectives in synthesizing this benchmark using VSS:

- Given a VHDL description developed without knowledge of the Structured Modeling, what modeling styles and VHDL language construct preferences would be used by the modeler?
- Could VSS synthesize the specification provided? If so, what processing steps were required? If not, could Structured Modeling practices be used to rewrite the description such that it would be synthesizable by VSS?
- What differences in the quality of the produced design would result from the application of various Representation Optimizations?
- What modeling style (Functional or Behavioral) is appropriate for this type of design?

7.1.1 Problem Description

A block diagram of this conceptual design is shown in Figure 7.1. There are four input and one output ports used for external communication. CLK is the system clock. RST is a one bit control line (active high) which indicates that a synchronous reset is to be performed. LDE is a one bit control line (active high) which indicates that a data value DTI (an integer in the range 0 to 4095) is to be loaded into the counter.

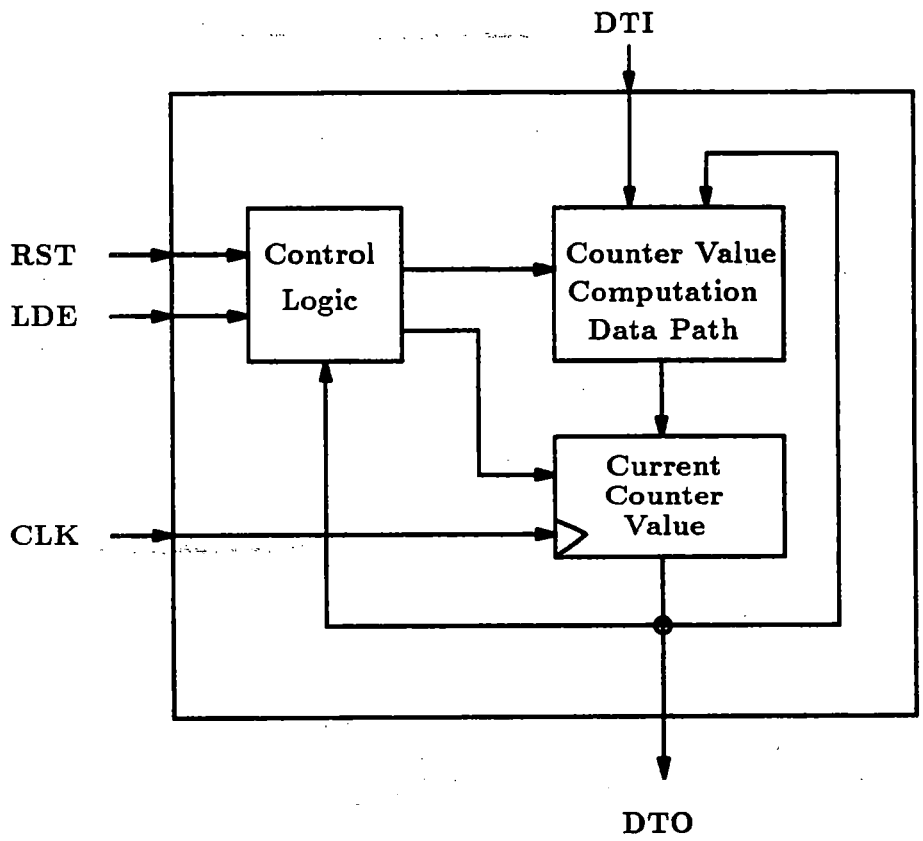


Figure 7.1: Rockwell Counter Block Diagram

The circuit to be synthesized has the following specification:

- The counter has a start count of 0 and a terminal count of 3327.
- For each clock (CLK) strobe, the counter increases by 208. If the count is greater than 3327, the counter will start at the previous beginning of the count plus 26 (in this case 0 + 26); if the previous beginning of the count plus 26 is greater than 207, then the count will start at the previous sequence plus 1.
- Portions of the first two count sequences are shown in Figure 7.2. The complete counting pattern of the counter consists of a total of 26 sequences. The counter counts the first column of the first sequence top to bottom, then the second column, and so on. When it reaches 3327, it will wrap around back to 0.
- The counter also has an active high load enable (LDE), which loads a data value (DTI) synchronously with the rising edge of the clock. The state machine must adjust to the new state so as to keep the same counting sequence.
- The counter must also have a synchronous reset (RST).

7.1.2 VHDL Behavioral Model

A VHDL description of the Rockwell counter written using the VHDL behavioral design style can be found in Appendix B.

This model consists of a nested IF statement which represents the conditional assignment to the counter value DTO_REG. If transformations are not applied to this description, VSS will target it to a CU/DP implementation. In this case, the synthesized design would consist of 8 machine states. The reason for this is that the CU/DP design model requires one state for the evaluation of a conditional branch expression, and a second state for branching based on the expression value.

```
0000 0026 0052 0078 0104 0130 0156 0182  0208   0001 0027 ... 0183 0209
0208 0234                            0390        0209
0416 0442                            0598        0417
0624 0650                            0806        0625
0832 0858                            1014        0833
1040 1066                            1222        1041
1248 1274                            1430        1249
1456 1482                            1638        1457
1664 1690                            1846        1665
1872 1898                            2054        1873
2080 2106                            2262        2081
2288 2314                            2470        2289
2496 2522                            2678        2497
2704 2730                            2886        2705
2912 2938                            3094        2913
3120 3146 3172 3198 3224 3250 3276 3302          3121
-------------------------------------------      ----------------------
3328 3354 3380 3406 3432 3458 3484 3510          3329 3355 ... 3511
```

Figure 7.2: Rockwell Counter Count Sequence

The application of Control Flow to Data Flow transformations restructures the CDFG Design Representation such that it will be mapped to a Functional design model. As the synthesis results show, this representation results in a cleaner, more efficient design than would be achieved by targeting this design to the CU/DP design model. When transformations are applied, the control of the conditional assignment has been moved from the control unit into the data path. In this example, the expressions used to determine branching conditions are generated by logic found in the data path. For the Behavioral model, these conditions are evaluated in the appropriate machine state and are stored in latches, and the latched values are input to control logic that implements a finite state machine (FSM). This FSM requires several clocks to perform the conditional branching as specified in each IF and ELSIF clause in the input description. After transformations are applied, the design becomes a concurrent model in which all conditional expressions are evaluated simultaneously; these

signals are used to select the appropriate data value to be assigned to DTO_REG on each clock.

7.1.3 VHDL Functional Model

The process description using sequential statements was converted to a description with concurrent statements in order to conform to Structured Modeling guidelines. An explanation of our reasoning for this modeling style is given in the next subsection.

The following modifications were necessary in order to convert the process level description into a synthesizable functional description:

1. Assignment to the output port is made via a signal assignment. This follows the Structured Modeling practice of using variables to represent values involved in data operations (which may require storage elements) and signals to represent the transfer of stored values (via wires) to the output port.
2. The Graph Compiler for VSS does not perform constant propagation optimizations currently. In order to reduce the amount of unnecessary hardware that would be generated for computations such as the addition and subtraction of constants, these optimizations were performed manually on the input description.

The equivalent dataflow (block) description which is preferred when using our Structured Modeling methodology can be found in Appendix B.

7.1.4 Structured Modeling Considerations

This design is classified as a *Functional* description in the Structured Modeling design style taxonomy. This functional design is a "single state" design where several conditions are tested on each clock event. The counter is a synchronous design; depending on the control signal values (RST and LDE), a reset, load or count operation will occur (with reset given the first priority, load second and count third).

In investigating the alternatives for modeling a Functional design using VHDL, Structured Modeling favors a VHDL block description as the most appropriate method for describing such a design. Some reasons for the preference in this example include:

1. Clock (CLK) and reset (RST) signals can be identified using subtypes defined within our VHDL synthesis package.
2. The VHDL block statement provides a convenient template which allows the synthesis tool to identify the storage class and function of various signals. Following the Structured Modeling guidelines, the block guard is used to represent an event such as a positive edge transition of the CLK signal.
3. Conversely, the process description seems to be more appropriate for describing sequential, multi-state designs. The design model for such designs consists of a cleanly partitioned control unit/data path pair. The process description for the benchmark of this particular case study presents the following problems for synthesis:
4. Identification of clocked storage elements is difficult. While the VHDL dataflow description style provides the guarded signal assignment in which the clock event can be expressed in the block guard (an explicit control of any assignment made to the guarded signal), no comparable construct exists in the VHDL

behavioral description style. Thus, it is impossible for the synthesis tool to distinguish which variables in a Behavioral (process) description should be mapped to registers and which should be mapped to wires. This ambiguity led to the establishment of the Structured Modeling convention that variables defined within a process are mapped to registers, while signals are used for inter-process communication.

5. Assignments to the same signal/variable are distributed among conditional branches. Unlike the dataflow descriptions, where a single conditional assignment statement is used to enumerate all assignments of data values to a variable under a corresponding condition, the behavioral description distributes variable assignment information throughout the description. Therefore, the synthesis tool must collect all of these assignments made under all conditions and make an assignment to a single variable. In the Rockwell counter example, assignment to the DTO_REG variable is made in every conditional branch.

7.1.5 VSS Synthesis Results

Functional Model Processing

The Functional (block) model of the Rockwell counter benchmark shown in Appendix B was first synthesized by VSS without invoking the Component Synthesis Algorithm (CSA). This design is referred to as **rw_cntr_func** in the following discussion. A second run was performed which invoked the CSA algorithm. This design is called **rw_cntr_func_csa**.

Figure 7.3 shows the netlist composed of GENUS generic components produced by the VSS system for the rw_cntr_func_csa design. The right half of the schematic shows the data path synthesized to perform the counter value computations. Currently, constants are treated as single word ROMs in the VSS system. The left half of the schematic consists of glue logic used to select data inputs and ALU functions. The COMPARATOR LOGIC block consists of random logic used to compute the conditional bits derived from the conditional expressions of the VHDL input description.

Figure 7.4 presents a VHDL behavioral description produced by the CSA algorithm. This description specifies the behavior of the ALU select logic. VSS processes this description in the Combinational mode in order to generate a gate level structure for this and other SELECT LOGIC components.

Behavioral Model Processing

Three experiments were performed to synthesize the Behavioral (process) description of the Rockwell counter:

1. VSS was invoked on the Behavioral input description (this design will be called **rw_cntr_beh**). Since no transformation were applied to this model, the Mobility Scheduler and Frequency Based Resource Binder were invoked to process the Behavioral design. A multi-state design results, requiring a control unit generated from the BIF state table description.
2. A second run (identified as **rw_cntr_beh_trans** in the subsequent discussion) applied the CFG to DFG transformations; this results in a design in which processing is completed using the Functional Design Compiler.

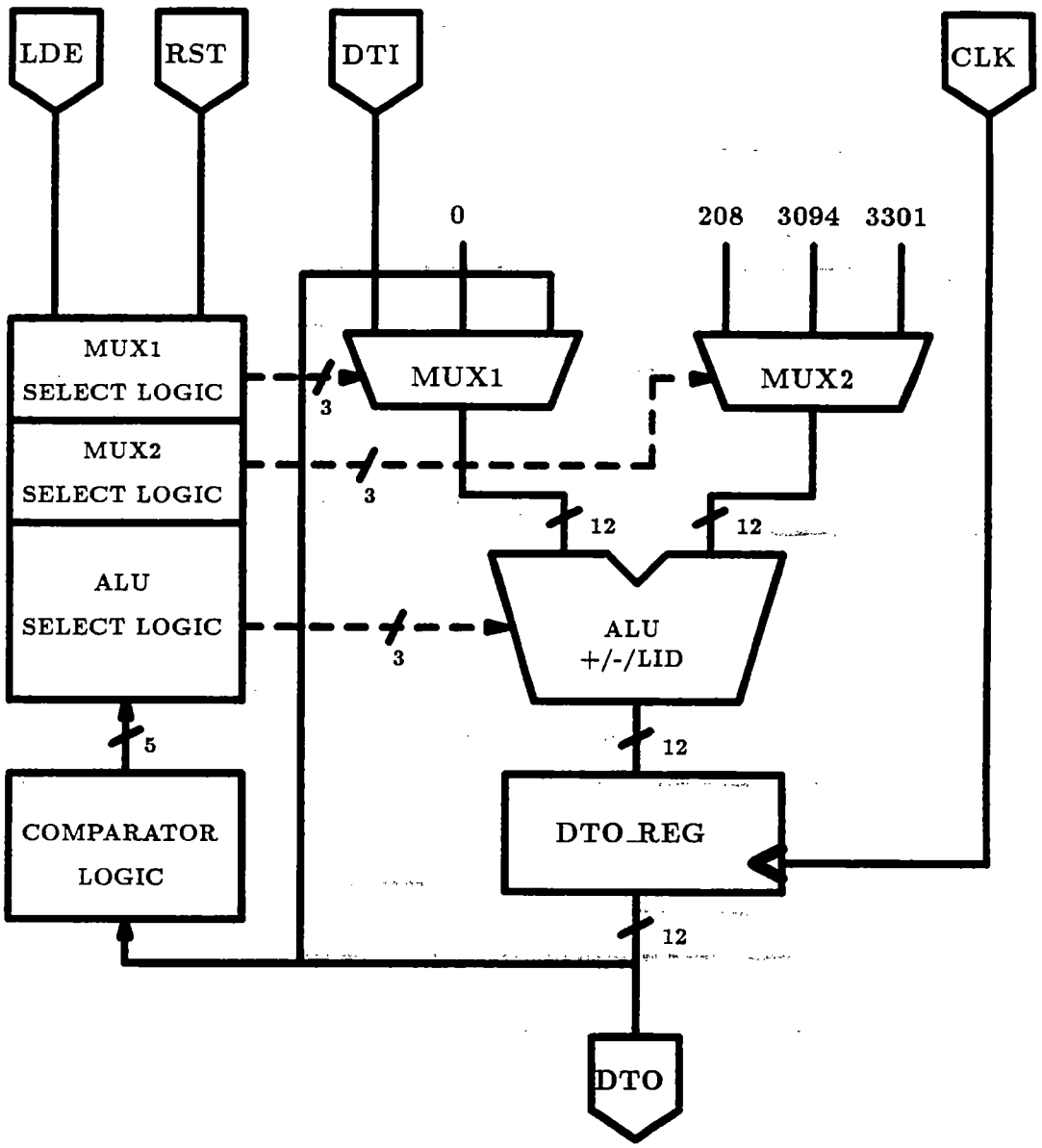


Figure 7.3: Structure Produced by VSS for Functional Model

```
-- ======================================================
-- TRUTH TABLE:
-- ======================================================
--        C62| Function
-- ------------------------------------------------------
--       00010 | ADD
--       00001 | SUB
--       00000 | SUB
--       10000 |LID
--       01000 |LID
--       00100 |LID
-- ------------------------------------------------------

entity  TRUTH_TABLE84 is
  port (C62:  in BIT_VECTOR(4 downto 0);
        CADD,CSUB,LID:  out BIT) ;
end TRUTH_TABLE84;

--VSS: design_style COMBINATIONAL

architecture dataflow of  TRUTH_TABLE84 is
begin

CADD <=
  ((not C62(4)) and (not C62(3)) and (not C62(2)) and
   C62(1) and (not C62(0)));
CSUB <=
  ((not C62(4)) and (not C62(3)) and (not C62(2)) and
   (not C62(1)) and C62(0)) or
  ((not C62(4)) and (not C62(3)) and (not C62(2)) and
   (not C62(1)) and (not C62(0)));
LID <=
  (C62(4) and (not C62(3)) and (not C62(2)) and
   (not C62(1)) and (not C62(0))) or
  ((not C62(4)) and C62(3) and (not C62(2)) and
   (not C62(1)) and (not C62(0))) or
  ((not C62(4)) and (not C62(3)) and C62(2) and
   (not C62(1)) and (not C62(0)));
end dataflow;
```

Figure 7.4: VHDL Description of the ALU Function Select Logic

3. A third run (**rw_cntr_beh_trans_csa**) applies CSA to the transformed behavioral model.

Figure 7.5 shows the structure synthesized for the VHDL Behavioral model of the Rockwell Counter (rw_cntr_beh). The control unit appears at the far left of the schematic. As with the select logic generated for the Functional design to which the CSA algorithm was applied, a gate level implementation of the control unit was also synthesized.

Figure 7.6 shows the data path generated for the transformed description of the rw_cntr_beh_trans design. Note that all select lines for the function unit and multiplexor select logic have been embedded in the data path.

Finally, Figure 7.7 shows the data path generated for the transformed description of the rw_cntr_beh_trans_csa design.

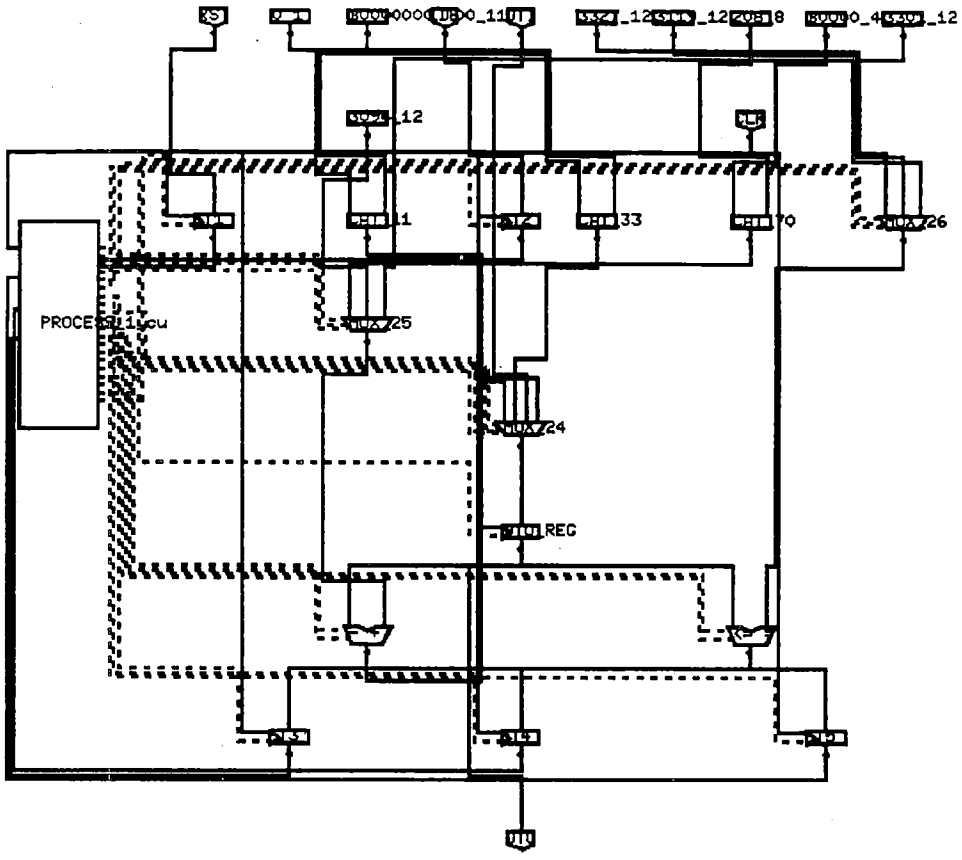


Figure 7.5: Design for Rockwell Counter Behavioral Description (CU/DP)

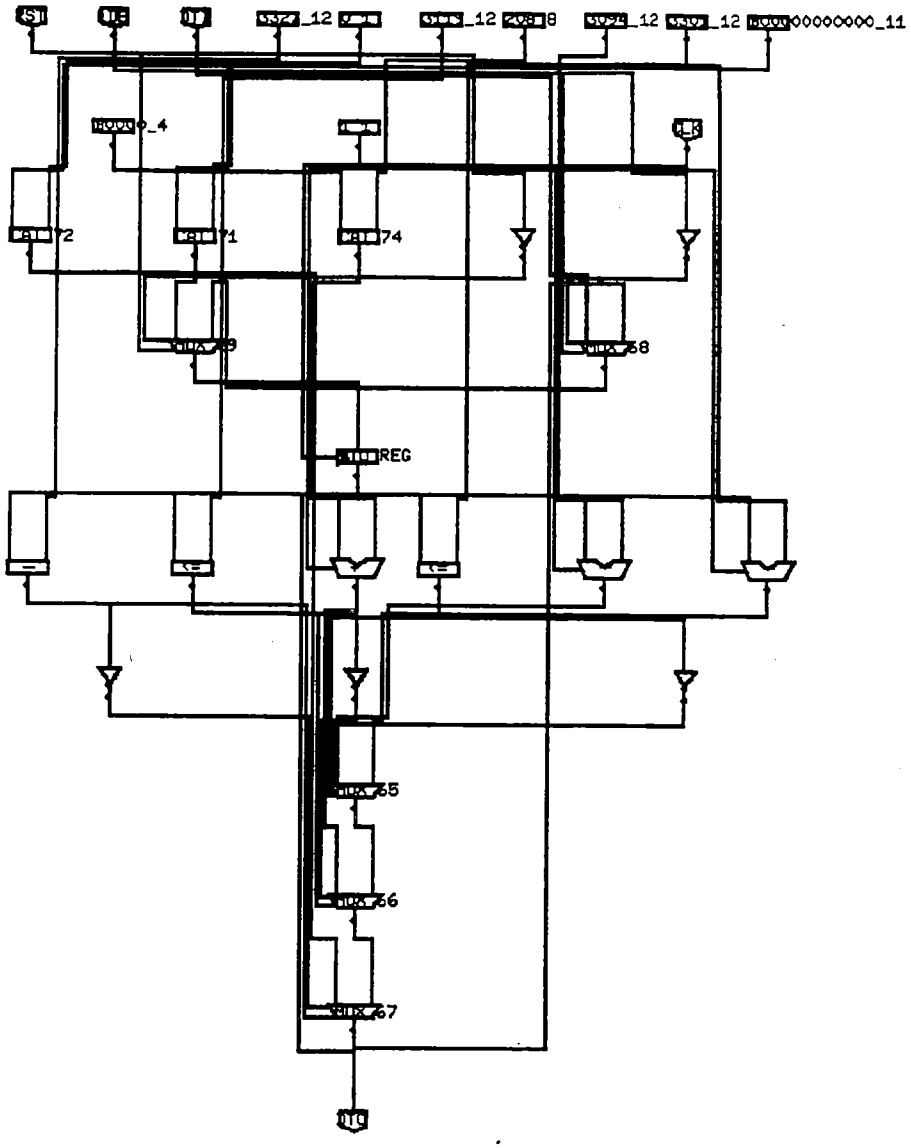


Figure 7.6: Design for Transformed Behavioral Description

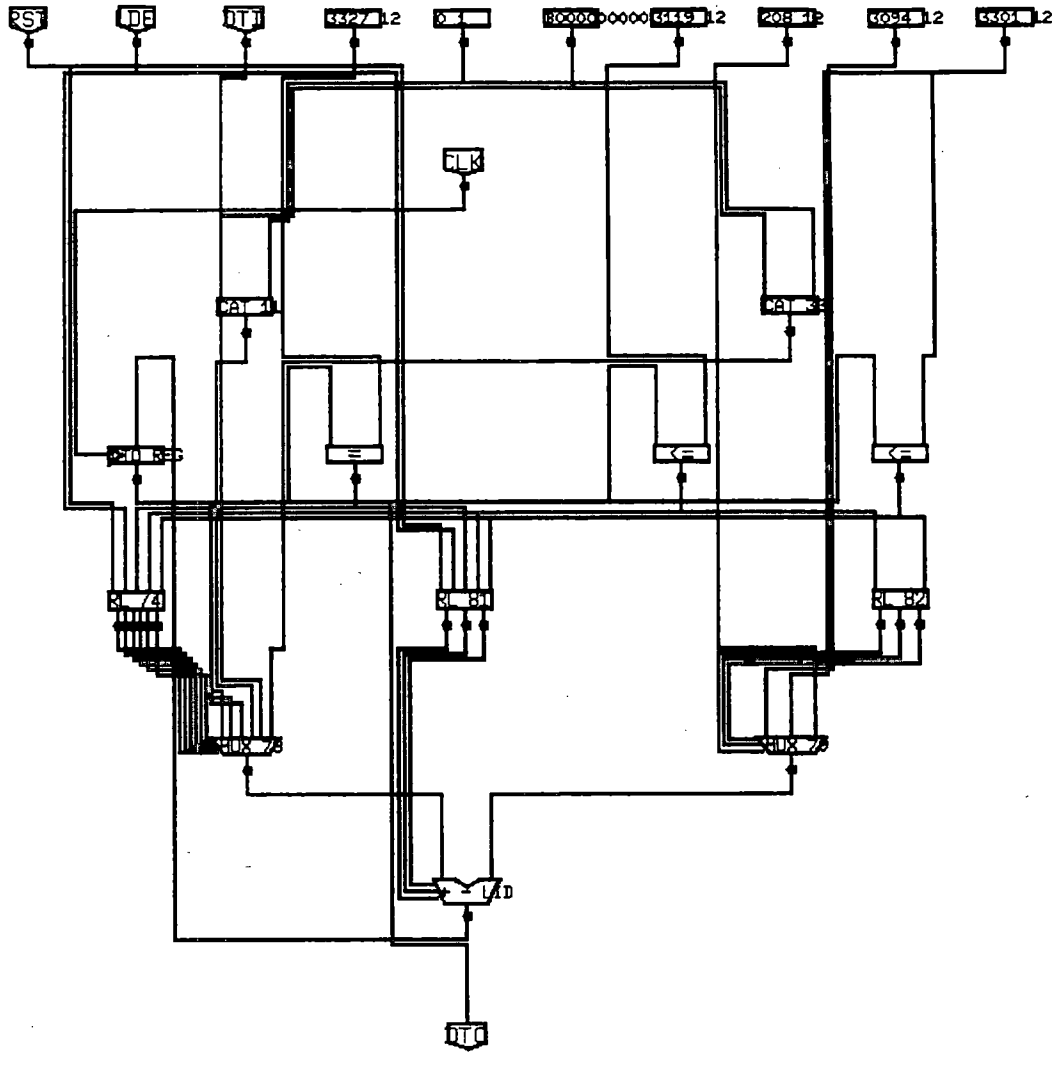


Figure 7.7: Design for Transformed Behavioral Description with CSA

7.1.6 Analysis

Table 7.2 summarizes the VSS processing options and design metrics achieved for each of the five experiments performed using the Functional and Behavioral VHDL models of the Rockwell Counter. The transistor count metrics have been partitioned into counts associated with functional units (FU), comparator units (COMP), multiplexors (MUX), registers and other storage elements (REG) and random logic (RL).

Several observations can be made in analyzing these results:

1. Due to the relatively small size of this example, the transistor count is dominated by the ALU, adder/subtractor and comparator components used (these units account for 33 to 62 percent of the transistor count in the numbers shown). Thus, the CSA algorithm improves the design in terms of transistor count by merging functional units.
2. The design synthesized from the Functional model without applying CSA has a negligibly smaller transistor count than the run which applied CSA. This result can be explained by looking at the random logic portions of the designs. CSA produces select logic which is suboptimal (has not been processed by logic synthesis).
3. The Behavioral design transistor count is actually increased by 2% when CFG to DFG transformations are applied. This can be attributed to the fact that the Behavioral design used a single comparator unit which could be shared across machine states, while the transformed design requires three concurrent comparisons. However, it is interesting to note that after CSA is applied to the transformed design, the result is comparable in quality to the design synthesized for the Functional model using CSA.

VSS Processing Options					
Model	**Struc. Mod. Style**	**CSA**	**CFG Trans.**	Scheduler	Resource Binder
rw_cntr_func	Functional	no	no	none	flat DFG
rw_cntr_func_csa	Functional	yes	no	none	flat DFG
rw_cntr_beh	Behavioral	no	no	Mobility	Freq. Based
rw_cntr_beh_trans	Behavioral	no	yes	none	flat DFG
rw_cntr_beh_trans_csa	Behavioral	yes	yes	none	flat DFG

Design Metrics						
Model	**Transistor Count**					
	FU	**COMP**	**MUX**	**REG**	**RL**	**TOTAL**
rw_cntr_func	1140	1008	672	576	80	3476
rw_cntr_func_csa	672	1008	720	576	506	3482
rw_cntr_beh	792	420	1080	960	362	3614
rw_cntr_beh_trans	1140	1008	960	576	10	3694
rw_cntr_beh_trans_csa	672	1008	1032	576	202	3490

Table 7.2: VSS Results for the Rockwell Counter Benchmark

4. Another fact which indicates that the use of the behavioral description style was inappropriate for this design is that in order to perform one count in the rw_cntr_func_csa and rw_cntr_beh designs, more clocks per count must be supplied to the Behavioral design versus the Functional design. This is due to the fact that the CU/DP architecture of the rw_cntr_beh design requires extra states to determine next state sequencing.

7.2 DRACO

This experment involves another industrially design chip developed at Rockwell International.[1] The design is more substantial in complexity, allowing for investigation of the following attributes of the VHDL Synthesis System and Structured Modeling:

- The modeling of the functionality of the design using temporal partitioning (grouping all operations which effect any component as a result of the occurrence of an event) versus a functional partitioning (collecting all operations which occur for a component or a group of components over all time).
- What techniques or modeling guidelines are required to properly model various storage elements (registers, buses, wires) within the same description.
- How will the partitioning of the VHDL input description (by using a hierarchy of blocks or processes) be reflected in the synthesized design.

[1]Rockwell International has granted U.C. Irvine permission to study the DRACO design for educational purposes.

7.2.1 Problem Description

DRACO is a peripheral interface Application Specific Integrated Circuit (ASIC) developed by Rockwell International for numerical control applications. The behavioral model was generated from a data sheet of the fabricated chip. which consisted of a description of the chip's input-output functionality, its physical and operational characteristics, and a functional block diagram. The data sheet contained very little abstract behavioral information. The VHDL behavioral model was developed through reverse engineering of the data sheet description, supplemented by further consultation with designers of the DRACO ASIC at Rockwell International [GD90].

A block diagram of the DRACO chip is shown in Figure 7.8. The primary function of the DRACO chip is to interface 16 I/O ports to a microprocessor's 8 bit multiplexed address/data bus and control signals. The chip consists of three main functional blocks: the *address decoder section* (ADRDEC), the *checksum/parity/error computation section* (CSPARITY), and the *input/output interface section* (IO).

Functional Partitioning

The ADRDEC block performs the following functions:

- latches the address byte and its associated parity bit
- generates and compares the parity on the address
- decodes the address to generate control signals
- implements the electronic key, used to control the loading of the chip's configuration

The CSPARITY block consists of hardware which generates and validates checksums and parity bits associated with incoming and outgoing data. A configuration

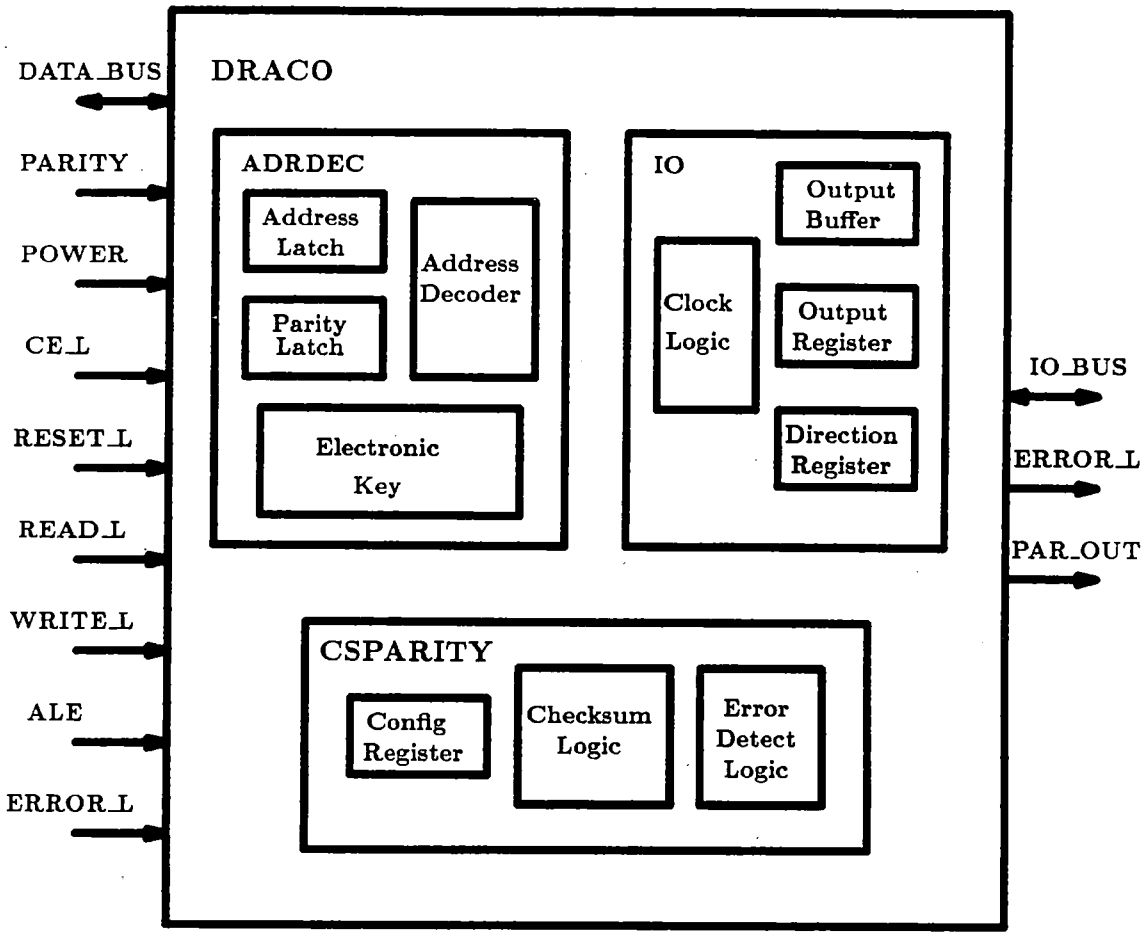


Figure 7.8: DRACO Block Diagram

register in this block selects the various parity and checksum error checking options available on the DRACO chip.

The IO interface consists of 16 bidirectional ports, and the appropriate selection logic to enable staggered output of the chip and to control the direction of data flow.

Temporal Partitioning

The behavior of the DRACO chip can be modeled using a state diagram consisting of the following eight states:

1. Reset
2. Chip Enable
3. Address Cycle
4. Read Cycle
5. Write Cycle
6. Idle
7. Chip Disabled
8. Power Off

Figure 7.9 shows the state transitions possible between these states.

These eight states are combined to perform three primary operations:

- allow data to be READ out of the chip
- WRITE data into the internal registers of the chip
- set the configuration of the DRACO chip

For a data access out of DRACO, the chip passes through the Address Cycle and the Read Cycle. For this to occur, the following events take place: the address appears on the address/data bus, an address latch enable (ALE) signal goes low, a read enable (READ_L) signal goes low, and data is placed on the address/data bus.

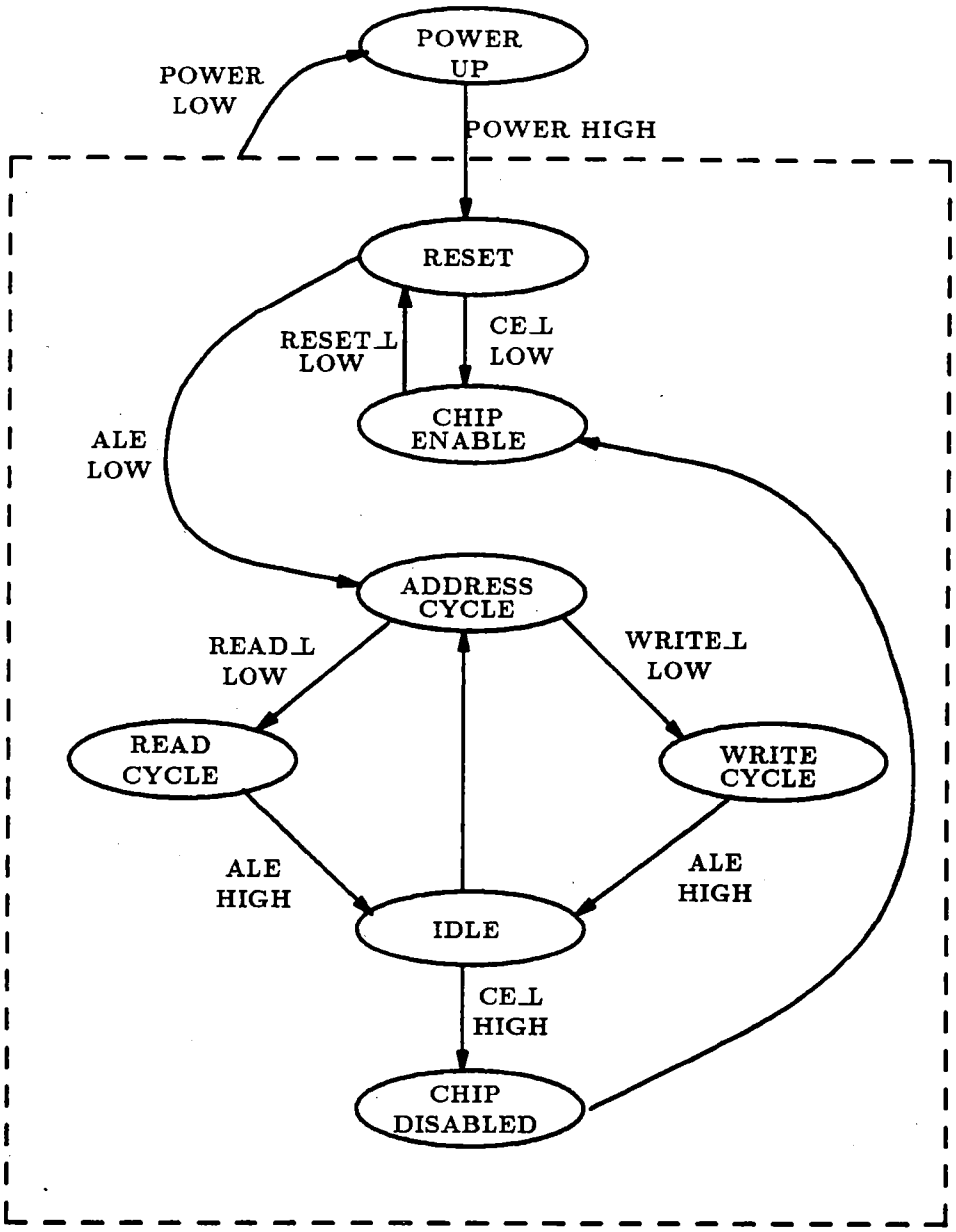


Figure 7.9: DRACO State Diagram

For a write to DRACO, the chip sequences through the Address Cycle and the Write Cycle. In order for this operation to execute, the address first appears on the address/data bus, ALE transitions to low, a write enable (WRITE_L) signal falls, data appears on the address/data bus, and WRITE_L rises. As ALE goes low, the address (if valid) is latched into DRACO. When WRITE_L rises, data is written into the registers of DRACO.

Setting the configuration of the DRACO chip involves the unlocking of the electronic key, the writing of the configuration into DRACO, and the relocking of the electronic key. The electronic key is unlocked by writing a specific data value into a specific location. The configuration value is also written to a specific address. Writing to an illegal address or writing an illegal data value to the electronic key will relock the key.

7.2.2 Structured Modeling Considerations

Several Behavioral and Functional models of the DRACO chip were developed to observe the differences in the design quality of the synthesized results. The following subsections outline the differences of each model. A model name will be associated with each model, and the model will be referred to by that name in subsequent discussions.

Behavioral Model

draco_beh

This model was the first VHDL model derived from the data sheet specification of DRACO. It consists of a model written with VHDL behavioral constructs. The model uses a temporal partitioning of the design into the eight states mentioned previously. VHDL process statements were used to model each behavioral state; all operations which occur in each state were modeling in the same process. This description will be synthesized with and without the application of Graph Transformations.

Functional Models

draco_rw_schem

This is a VHDL structural description derived directly from the logic schematic produced by Rockwell International. No synthesis of this model was performed; it was used as a point of reference for comparison of gate and transistor counts of the synthesized designs.

draco_logic

A Functional description was derived which described each component found in the Rockwell logic schematic using VHDL concurrent assignment statements. As the description was organized into a hierarchy using VHDL block statements, the partitioning followed that of the schematic. Consequently, this model closely reflected

the structure of the logic schematic. This resulted in a cleaner partitioning of assignments made under common events (as reflected in block guards used to trigger guarded signal updates).

draco_func2

This Functional model was created by first translating the original process level model to an equivalent block model and then applying Structured Modeling guidelines. In this model, a direct translation of the assignment statements found in the Behavioral description was made in place (for example, assignments to the same signal were made in various places in the description; separate concurrent assignment statements were formed in the equivalent Functional description). These multiple assignment statements were then combined, conditional clauses were coalesced, and an attempt was made to collect assignments made under similar conditions and guard conditions into the same VHDL block statement. Thus, this model differs from the draco_logic model in that the criteria used for partitioning the model into blocks was assignment under common conditions, rather than common structure as reflected in the Rockwell schematic.

draco_func3

This model is a higher level Functional model which used temporal partitioning of the DRACO functionality into the three primary operations outlined earlier. Each operation was modeled using a VHDL block statement.

7.2.3 Analysis

The number of RTL components in this design makes it impractical to include the synthesized schematic in this context. Table 7.3 compares the results of VSS synthesis for the various DRACO models. In addition to the breakdown of transistor count by unit type, the last column of the Design Metrics table shows a ratio of the total transistor count of each model to the count produced for the draco_rw_schematic which serves as a basis of comparison to human quality design.

The Functional draco_logic model has a structural implementation that is actually 7% smaller than the human design. This can be traced to additional flip flops and random logic in the draco_rw_schem design which are used to generate clocks for the storage elements on the DRACO chip. This special clocking logic was not specified in the original data sheet from which the VHDL model was generated; therefore, it does not appear in the draco_logic model.

The draco_func2 and draco_func3 models are roughly 70% larger than the human quality design. Factors which contribute to this difference include:

- Conditional expressions used to test for equality comparisons (for example, a check that the address bus is carrying a particular hexadecimal value) are implemented using full comparators by VSS. When logic optimization is performed on comparator units for which one input is a constant (as would be the case in the address comparison example above), a simpler gate level structure would result. As can be seen in the DRACO results table, the draco_func2 and draco_func3 designs contain three times the number of comparison units as does the draco_rw_schem design.

VSS Processing Options					
Model	**Struc. Mod. Style**	**CSA**	**CFG Trans.**	Scheduler	Resource Binder
draco_logic	Functional	no	no	none	hier. DFG
draco_func2	Functional	no	no	none	hier. DFG
draco_func3	Functional	no	no	none	hier. DFG
draco_beh	Behavioral	no	no	Mobility	Freq. based
draco_beh_trans	Behavioral	no	yes	none	hier DFG

Design Metrics							
Model	**Transistor Count**						**Ratio**
	FU	**COMP**	**MUX**	**REG**	**RL**	**TOTAL**	
draco_rw_schem	240	2240	64	3696	1088	7328	1.00
draco_logic	240	2212	32	3264	1080	6828	0.93
draco_func2	480	6300	1440	3552	1144	12916	1.76
draco_func3	480	6636	672	3168	1238	12194	1.66
draco_beh	5040	504	6942	6960	10616	30062	4.10
draco_beh_trans	480	6328	15488	4320	2106	28722	3.92

Table 7.3: VSS Results for the DRACO Benchmark

Based on experiments run using VSS and MILO [VG88] which compared the transistor counts of full (equality) comparator components to a gate level implementation, it was shown that the gate level implementation was 50% smaller than the comparator unit on the average. If this logic optimization is taken into account in the draco_func2 (which consists of 56 8-bit comparators and 2 1-bit comparators) and the draco_func3 (59 8-bit comparators, 2 1-bit comparators) models, then the optimized transistor counts would be reduced by approximately 3200 transistors. This translates into designs which are within 34 and 21 percent, respectively, of the draco_rw_schem standard of comparison.

- The draco_func2 and draco_func3 designs contain one more functional unit and more random logic than does the draco_rw_schem design. VSS translates boolean expressions found in the VHDL input description into a straightforward, suboptimal gate level implementation. This random logic is reduced by logic optimization. Since the MILO system groups certain regular components (ALUs, comparators, etc.) along with logic gates in order to optimize random logic, it is difficult to determine the effect of logic optimization on the various categories of components (FU, COMP, MUX, REG, RL) shown in the VSS results. However, preliminary results reported on processing of the draco_func2 and draco_func3 models by MILO indicate a reduction of approximately 40 to 50% in the amount of random logic remaining after optimization. Taking this improvement into consideration for the RL category, the draco_func2 model transistor count is reduced by 515 transistors (improving it in comparison to the draco_rw_schem design by 8%), and the draco_func3 transistor count is reduced by 558 transistors (a reduction of 9% when compared to draco_rw_schem) netlist.

The sizes of the designs produced by VSS from the Behavioral model are significantly larger than those generated for the Functional model. This can be attributed to the following factors:

- The VSS Graph Compiler does not perform compiler optimizations such as common subexpression elimination. This results in replicated conditional expressions in the CDFG which are mapped to redundant control logic in the structure. Using the same estimation of improvement of the random logic transistor count as above, the draco_beh model would be reduced by 5300 transistors after logic optimization, while the draco_beh_trans model will be reduced by 1050 transistors.
- The partitioning of the VHDL input description into processes can at times result in duplication of functional units. Because each process is mapped to a separate CU/DP architecture, the design model will not allow sharing of resources across process partitions. Thus, VSS cannot detect parallelism across these process partitions. In this example, a checksum calculation is performed in two processes; because it cannot be determined that these computations are mutually exclusive, dedicated resources are allocated in each CU/DP associated with the processes. The Functional model utilizes a functional partitioning which shares the checksum computation resources.
- In the behavioral model, registers are being allocated for variables which should be wires. This is a result of the difficulty in determining which variables should have storage allocated to them and which should be implemented as wires. In order to make this distinction, Structured Modeling guidelines were established which map each variable within a process to a register. Signals (which cannot be declared local to a process) are defined in the block which encloses the process

and are used for inter-process communication. In this example, the VSS system incorrectly binds 14 variables which should be wires to registers (accounting for 816 transistors), and the ADD_DATA bus to a register (accounting for another 384 transistors).

- The CFG to DFG transformations introduce a substantial amount of multiplexing. This is due to the fact that the description consists of a large number of conditional (IF) statements. Any assignment made to a variable in any conditional branch will be transformed into a DFG representation consisting of a tree of CHOOSE_VALUE nodes (mapped to multiplexor components). This multiplexor tree is then used to select the appropriate data value based on the branching conditions.

7.3 AM2910 Microprogram Controller

7.3.1 Problem Description

The Am2910 microprogram controller is an address sequencer which controls the sequence of execution of microinstructions stored in microprogram memory [SBN80]. A block diagram of the Am2910 is shown in Figure 7.10. In addition to the capability of sequential access, it provides conditional branching to any microinstructions within its 4096-microword range. A last-in, first-out stack provides microsubroutine return linkage and looping capability. There are five levels of nesting allowed for microsubroutines. Microinstruction loop-count control is provided with the count capacity of 4096.

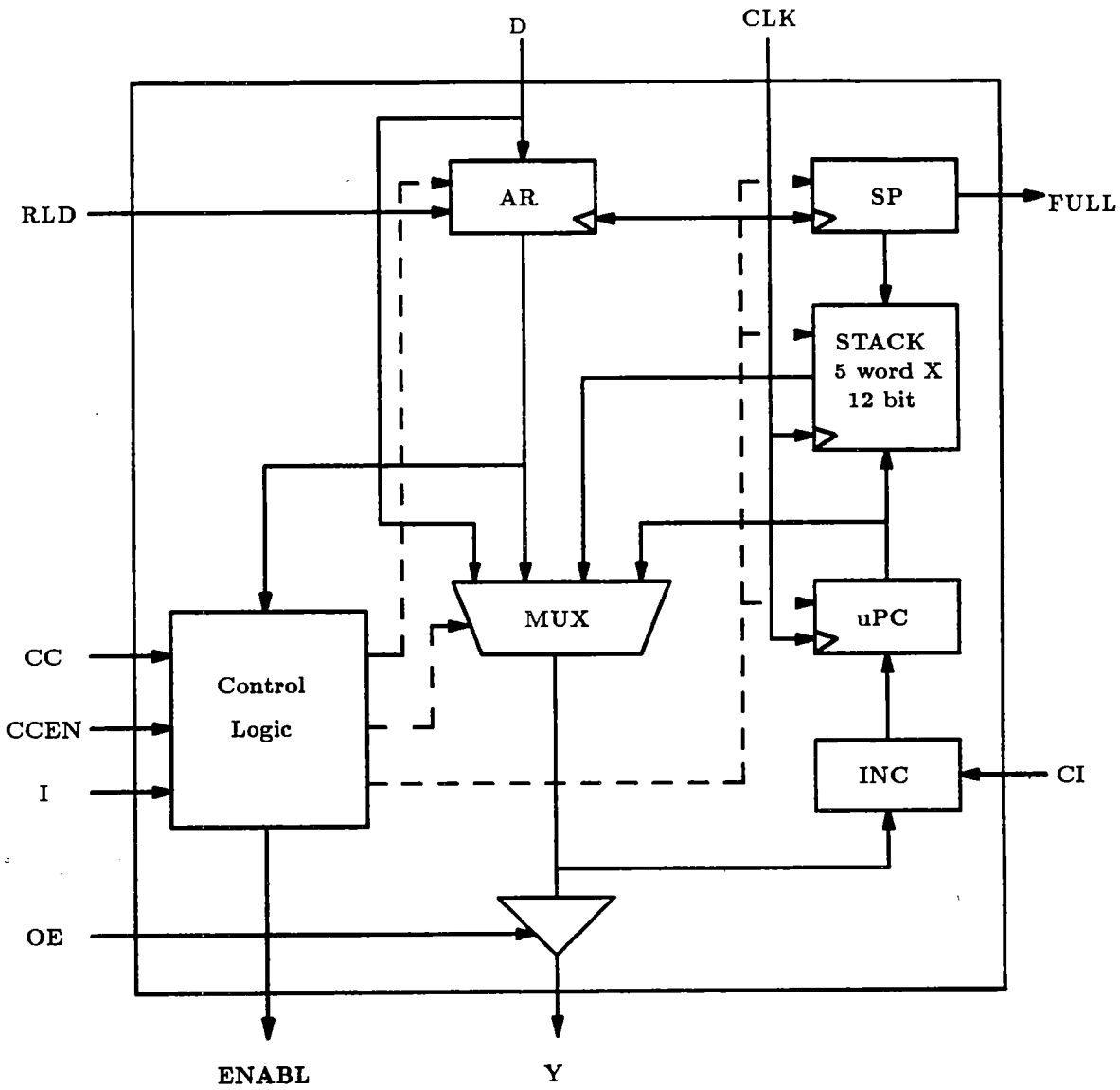


Figure 7.10: Am2910 Block Diagram

During each microinstruction, the microprogram controller provides a 12-bit address from one of four sources: (1) the microprogram address register (uPC), which usually contains an address which is one greater than the previous address; (2) an external (direct) input (D); (3) a register/counter (AR) which retains data loaded during a previous microinstruction; or (4) a five-deep last-in, first-out stack (STACK).

7.3.2 Structured Modeling Considerations

Because this design is a microprogram sequencer which decodes an input instruction, the Am2910 can be modelled in a straightforward fashion using a Behavioral description with a CASE statement. The block diagram can be partitioned easily into a data path consisting of the registers and register file (STACK), a multiplexor, and an increment unit. When viewed from the Functional perspective, this design can also be modeled as a single state machine in which on each state, appropriate control signals are applied to the storage and data select units such that the appropriate actions are performed. Thus, an equivalent Functional can also be used to represent the operation of each component based on conditions evaluated by the Control Logic.

The equivalent Functional and Behavioral VHDL descriptions were developed for this design. They can be found in Appendix B. The Functional model (am2910_func) consists of one conditional signal assignment per storage element or data wire found in the block diagram. At the time of these experiments, the VSS system did not have CASE statement CFG to DFG transformations available; consequently, the model which used the CASE statement (referred to as **am2910_case** in subsequent discussions) was rewritten using a nested IF construct (the **am2910_if** model) so that transformations could be applied.

7.3.3 Analysis

Table 7.4 compares the results of VSS synthesis for the various models. The following observations can be made:

VSS Processing Options					
Model	**Struc. Mod. Style**	**CSA**	**CFG** Trans.	Scheduler	Resource Binder
am2910_func	Functional	no	no	none	flat DFG
am2910_if	Behavioral	no	no	Mobility	Freq. based
am2910_if_trans	Behavioral	no	yes	none	flat DFG
am2910_case	Behavioral	no	no	Mobility	Freq. based

Design Metrics							
Model	**Transistor Count**						**States**
	FU	**COMP**	**MUX**	**REG**	**RL**	**TOTAL**	
am2910_func	754	5250	1824	1152	596	9576	1
am2910_if	300	0	1912	3936	18378	24526	126
am2910_if_trans	2246	1890	9600	1920	368	16024	1
am2910_case	198	42	1604	3552	5002	10398	97

Table 7.4: VSS Results for the AM2910 Benchmark

- These results show that given the CU/DP model to which VSS targets the Behavioral designs, the number of states required for the am2910_if design is 30% larger than that of the am2910_case design. This is a result of the control unit model used in VSS and the difference in mapping the IF statement

versus the CASE statement onto this model. In the am2910 benchmark, the am2910_case model uses a CASE statement with 16 alternatives; the equivalent nested IF description (am2910_if) uses 15 levels of 2-way branch decision nodes. The CU/DP model used by VSS computes the test value used to determine branching in the state prior to the state in which the branch actually occurs. The CASE branch can occur in two states: one state to evaluate the value of the test condition, and one state to direct execution to the first instruction of the appropriate branch. However, for the equivalent nested IF branching condition, the am2910_if model requires 2 to 30 states to execute (depending on the value of the condition bits and the availability of units for evaluation of these conditional bit values).

- It is also interesting to note that the am2910_case model, even without logic optimization, is within 10% of the transistor count of the am2910_func model.

7.4 8251 USART

7.4.1 Problem Description

The Intel 8251A is a programmable communication interface chip [TS85] or Universal Synchronous/Asynchronous Receiver/Transmitter (USART) designed for data communications between microprocessors. The 8251A is used as a peripheral device and is programmed by the CPU to operate using a variety of serial data transmission techniques. The USART accepts data characters from the CPU in parallel format and converts them into a continuous serial stream for transmission. Simultaneously, it can receive serial data streams and convert them into parallel data characters for

the CPU. The USART will signal the CPU whenever it can accept a new character for transmission, or whenever it has received a character for the CPU.

A block diagram of the 8251A is shown in Figure 7.11.

7.4.2 Structured Modeling Considerations

A Behavioral model for the 8251A was written using three processes: MAIN, TRANSMIT and RECEIVE. The areas of the design modeled by each process are indicated in Figure 7.11 by the dashed boxes.

7.4.3 Analysis

Due to the size of the input description, each of the three processes in the Behavioral model were processed by VSS separately. The transistor counts of the synthesized results for the MAIN and TRANSMIT processes are shown in Table 7.5.

These descriptions use VHDL CASE and WHILE loop constructs which are not processed currently by the transformations. Consequently, the effects of transformations on this design cannot be evaluated. The results presented here are preliminary in the sense that the design cannot be evaluated as thoroughly as in the previous examples. However, the synthesis of the 8251A has aided in the verification of the VSS software.

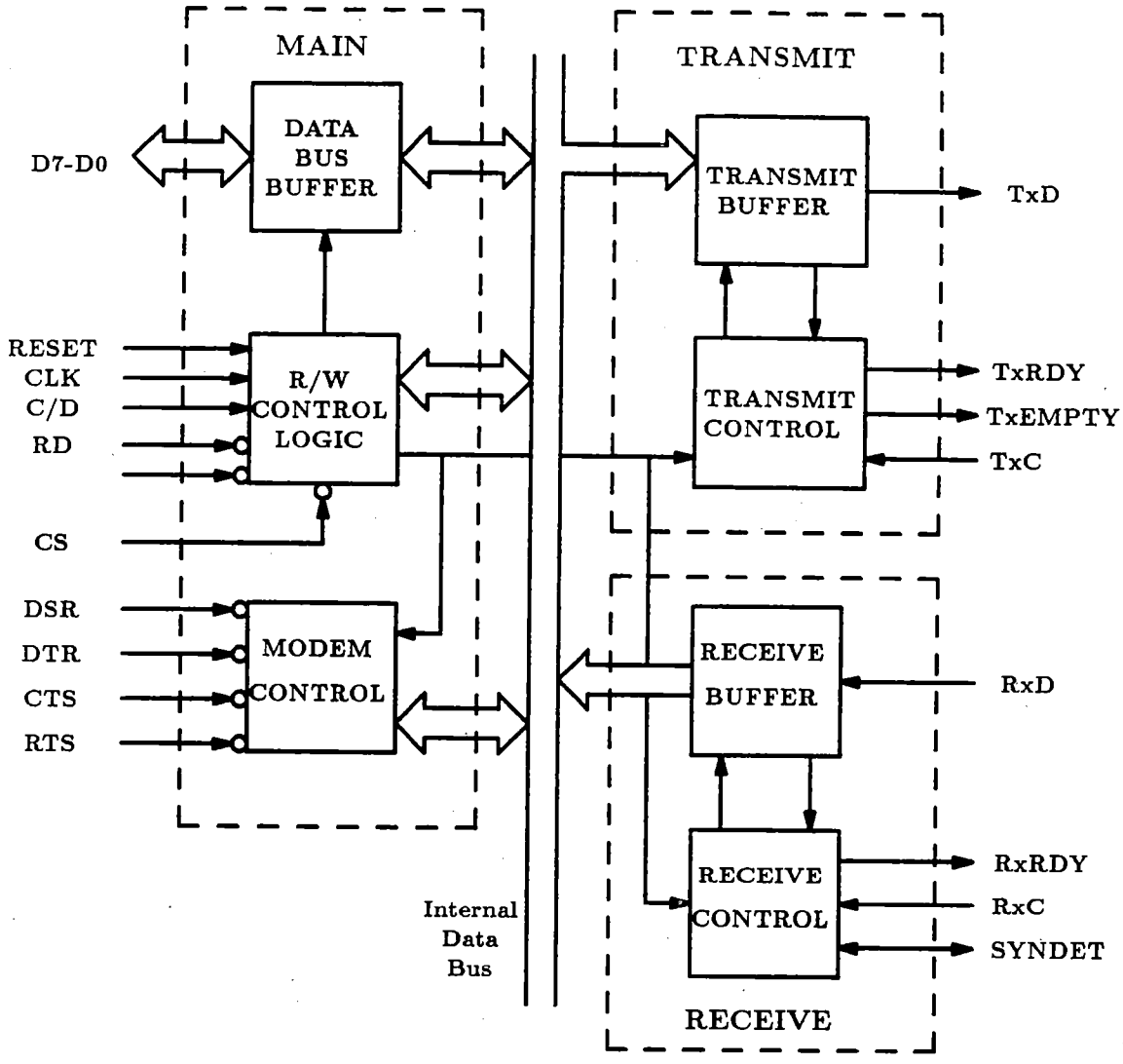


Figure 7.11: 8251A Block Diagram

VSS Processing Options					
Model	**Struc. Mod. Style**	**CSA**	**CFG** Trans.	Scheduler	Resource Binder
main	Behavioral	no	no	Mobility	Freq. Based
transmit	Behavioral	no	no	Mobility	Freq. based

Design Metrics							
Model	**Transistor Count**						**States**
	FU	**COMP**	**MUX**	**REG**	**RL**	**TOTAL**	
main	0	574	8428	9936	4202	23140	91
transmit	240	49	1316	5664	3880	11149	40

Table 7.5: VSS Results for the 8251A Benchmark

Chapter 8

Conclusions

8.1 Summary of Contributions

This dissertation has presented an approach to behavioral synthesis which uses the VHDL language for the modeling of the input behavior as well as the structure of the synthesized design. An examination of the issues involved in behavioral modeling was presented. This motivated the need for the development of a Structured Modeling methodology which suggests standard VHDL modeling practices for synthesis. These modeling practices were applied to several design examples in order to evaluate the various modeling practices and their effects on the quality of the synthesized design.

To demonstrate the feasibility of this approach, the implementation of the VHDL Synthesis System (VSS) was discussed. A synthesis framework was developed with a Control/Data Flow Graph and Partial Design Representation at its core. This framework provides the opportunity to incorporate various synthesis algorithms which can be evaluated in a common design environment. Experiments were performed to demonstrate the effects of different modeling styles on the quality of the design produced by VSS. Several alternative VHDL models were examined for each

benchmark, illustrating the improvements in design quality achieved when Structured Modeling guidelines were followed.

Through this work, we have substantiated the following claims that were established as the objectives of this research:

- VHDL can be used as a language for synthesis if the proper semantics are well defined.
- The Structured Modeling methodology serves as a useful guideline for synthesis from VHDL in the context of the VSS synthesis framework.
- The modeling style used in the synthesis input behavioral description has a direct effect on the quality of the synthesized output.
- With the appropriate application of representation transformations and optimizations, human quality design can be achieved.

8.2 Future Work

While the VHDL Synthesis System (VSS) has served as a valuable tool for the evaluation of our synthesis methodologies, several improvements can be made to this framework. Principal among these enhancements are:

1. *Incorporation of additional representation optimizations.* By performing standard compiler optimization techniques such as common subexpression elimination, constant folding, and in-line procedure expansion, more optimal designs would result at the register-transfer level.

2. *Automated design model and transformation selection.* Development of a mechanism or strategy which will select the appropriate design model and representation for a supplied VHDL model would relieve the designer of making these choices manually. These choices could be suggested by the tool, allowing the designer to override these options if he/she so chooses.

3. *Alternative input specification formats.* Because the design representation was developed with the intent of being a general purpose format for capturing necessary information for synthesis, it is possible to map other existing hardware description languages (or those under development in related work at U.C. Irvine such as BIF [DHG89] or SpecCharts [VNG90]) to this representation. This would allow for alternative information interchange formats between synthesis tools.

4. *System level synthesis.* VSS can be adapted to the processing of system level specifications [VNG90], where VHDL or other hardware description languages are used to specify a set of chips which communicate via protocols.

5. *Incorporation of testability measures.* Industrial concerns of design verification and fault diagnosis has spawned interest in the possibility of incorporating testability measures and practices into the synthesis design process.

6. *Feedback from logic and layout synthesis.* While the cost functions used to make design decisions in VSS are influenced by transistor counts and other lower level parameters of the synthesized design, a tighter coupling with a layout synthesis system would ensure that high level synthesis decisions have the appropriate effects on the layout generated.

7. *Specification of timing constraints.* The design representation used within VSS requires additional enhancements to allow for the expression of timing relationships. Since the mechanisms used to express timing in VHDL are not well defined, development of a semantics of the VHDL timing constructs is needed.

Bibliography

[AN88] A. Aiken and A. Nicolau. Perfect Pipelining: A New Loop Parallelization Technique. In *Proc. of the 1988 European Symp. on Programming*, 1988.

[Arm88] J. Armstrong. Modeling with HDLs. *IEEE Design and Test*, February 1988.

[Arm89] J. Armstrong. *Chip Level Modeling with VHDL.* Prentice-Hall, 1989.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bar81] M. Barbacci. Instruction Set Processor Specifications (ISPS): the Notation and its Applications. *IEEE Transactions on Computers*, C-30(1), January 1981.

[BC+88] R. Brayton, R. Camposano, et al. The Yorktown Silicon Compiler. In D. Gajski, editor, *Silicon Compilation.* Addison Wesley, 1988.

[Ben83] J. Bendas. Design through Transformations. In *20th Design Automation Conference*, 1983.

[BFR85] T. Blackman, J. Fox, and C. Rosebrugh. The SILC Silicon Compiler: Language and Features. In *22nd Design Automation Conference*, 1985.

[BG87] F. Brewer and D. Gajski. Knowledge-Based Control in Micro-Architecture Design. In *24th Design Automation Conference*, 1987.

[Bha86] J. Bhasker. Process Graph Analyzer: A Front End Tool for VHDL Behavioral Synthesis. In *Proceedings of the 10th Annual Honeywell International Computer Sciences Conference*, 1986.

[BRSVA87] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A.Wang. MIS: A Multiple-Level Logic Optimization System. *IEEE Transactions on Computer-Aided Design*, CAD-6(6), November 1987.

[BTK88] R. Blackburn, D. Thomas, and P. Koenig. CORAL II: Linking Behavior and Structure in an IC Design System. In *25th Design Automation Conference*, 1988.

[CAD87] CAD Language Systems Inc. *VHDL Tutorial for IEEE Standard 1076 VHDL*, 1987.

[Che90] G. Chen. An Intelligent Component Database for Behavioral Synthesis. In *27th Design Automation Conference*, 1990.

[CST88] R. Campasano, L. Saunders, and R. Tabet. High-Level Synthesis from VHDL. Technical Report RC 14282, IBM Research Division, T.J. Watson Research Center, December 1988.

[CT88] R. Campasano and R. Tabet. Design Representation for the Synthesis of Behavioral VHDL Models. Technical Report RC 14282, IBM Research Division, T.J. Watson Research Center, December 1988.

[DHG89] N. Dutt, T. Hadley, and D. Gajski. BIF: A Behavioral Intermediate Format for High Level Synthesis. Technical Report 89-03, University of California at Irvine, September 1989.

[DHG90] N. Dutt, T. Hadley, and D. Gajski. An Intermediate Representation for Behavioral Synthesis. In *27th Design Automation Conference*, 1990.

[DN89] S. Devadus and R. Newton. Algorithms for Hardware Allocation in Data Path Synthesis. *IEEE Transactions on Computer-Aided Design*, CAD-8(7), July 1989.

[DR+86] H. DeMan, J. Rabaey, et al. Cathedral II: A Silicon Compiler for Digital Signal Processing. *IEEE Design and Test*, December 1986.

[Dut88] N. Dutt. GENUS: A Generic Component Library for High Level Synthesis. Technical Report 88-12, University of California at Irvine, September 1988.

[Dut90] N. Dutt. LEGEND: A Language for Generic Component Library Description. In *IEEE International Conference on Computer Languages*, 1990.

[EN89] K. Ebcioglu and A. Nicolau. A Global Resource-constrained Parallelization Technique. In *Proc. ACM SIGARCH ICS-89: Int. Conf. on Supercomputing*, 1989.

[EW77] E. Eichelberger and T. Williams. A Logic Design Structure for LSI Testability. In *14th Design Automation Conference*, 1977.

[GBdH86] D. Gregory, K. Bartlett, A. deGeus, and G. Hatchel. SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic. In *23rd Design Automation Conference*, June 1986.

[GD90] R. Gupta and N. Dutt. Behavioral Modeling of DRACO: A Peripheral Interface ASIC. Technical Report 90-13, University of California at Irvine, June 1990.

[GDP86] D. Gajski, N. Dutt, and B. Pangrle. Silicon Compilation: A Tutorial. In *Proceedings of the Custom Integrated Circuits Conference*, 1986.

[GK83] D. Gajski and R. Kuhn. New VLSI Tools. *IEEE Computer*, December 1983.

[GK84] E. Girczyc and J. Knight. An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. In *International Conference on Computer Design (ICCD84)*, October 1984.

[GLVW90] D. Gajski, J. Lis, N. VanderZanden, and A. Wu. Synthesis from VHDL: Rockwell-Counter Case Study. Technical Report 90-09, University of California at Irvine, April 1990.

[Gup91] R. Gupta. Transformations for Behavioral Synthesis. Master's thesis. Dept. of Electrical and Computer Engineering, University of California, Irvine, January 1991.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8, 1987.

[HKL89] P. Harper, S. Krolikoski, and Oz Levia. Using VHDL as a Synthesis Language in the Honeywell VSYNTH System. In *Ninth International Symposium on Computer Hardware Description Languages (CHDL89)*, 1989.

[IEE87] IEEE. *VHDL Language Reference Manual, Draft Standard 1076/B*, June 1987.

[JR+89] J.Y. Jou, S. Rothweiler, et al. BESTMAP: Behavioral Synthesis from C. In *International Workshop on Logic Synthesis*, May 1989.

[Kin87] C. Kingsley. The Implementation of a State Machine Compiler. In *24th Design Automation Conference*, 1987.

[Kow85] T. Kowalski. *An Artificial Intelligence Approach to VLSI Design.* Kluwer Academic Publishers, 1985.

[LDSM80] D. Landeskov, S. Davidson, B. Shriver, and P. Mallett. Local Microcode Compaction Techniques. *Computing Surveys*, 12(3), September 1980.

[LG87] S. Lin and D. Gajski. LES: A Layout Expert System. In *24th Design Automation Conference*, June 1987.

[LG88] J. Lis and D. Gajski. Synthesis from VHDL. In *International Conference on Computer Design (ICCD88)*, October 1988.

[LMOI89] S. Levitan, A. Martello, R. Owens, and M. Irwin. Using VHDL as a Language for Synthesis of CMOS VLSI Circuits. In *Ninth International Symposium on Computer Hardware Description Languages (CHDL89)*, 1989.

[McF78] M. McFarland. The Value Trace: A Data Base for Automated Digital Design. Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, December 1978.

[OG86] A. Orailoglu and D. Gajski. Flow Graph Representation. *23rd Design Automation Conference*, June 1986.

[PG86] B. Pangrle and D. Gajski. State Synthesis and Connectivity Binding for Microarchitecture Compilation. In *International Conference on Computer-Aided Design*, 1986.

[PG87] B. Pangrle and D. Gajski. Slicer: A State Synthesizer for Intelligent Silicon Compilation. In *International Conference on Computer Design (ICCD87)*, 1987.

[PL88] B. Preas and M. Lorenzetti. *Physical Design Automation of VLSI Systems.* Benjamin/Cummings, 1988.

[PLNG90] R. Potasman, J. Lis, A. Nicolau, and D. Gajski. Percolation Based Synthesis. In *27th Design Automation Conference*, 1990.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. MAHA: a Program for Datapath Synthesis. In *23rd Design Automation Conference*, 1986.

[RGB90] E. Rundensteiner, D. Gajski, and L. Bic. The Component Synthesis Algorithm: Technology Mapping for Register Transfer Descriptions. In *International Conference on Computer-Aided Design*, 1990.

[Sau87] L. Saunders. The IBM VHDL Design System. In *24th Design Automation Conference*, 1987.

[SBN80] D. Siewiorek, C. Bell, and A. Newell. *Computer Structures: Principles and Examples.* McGraw-Hill, 1980.

[SHMO86] W. Scott, R. Hamachi, R. Mayo, and J. Ousterhout. Berkeley CAD Tools User's Manual. Technical Report UCB/CSD 86/272, University of California at Berkeley, 1986.

[T+83] D. Thomas et al. Methods of Automatic Data Path Synthesis. *IEEE Computer*, December 1983.

[Tri87] H. Trickey. Flamel: A High-Level Hardware Compiler. *IEEE Transactions on Computer-Aided Design*, CAD-6(2), March 1987.

[TS83] C. Tseng and D. Siewiorek. Facet: A Procedure for the Automated Synthesis of Digital Systems. In *20th Design Automation Conference*, 1983.

[TS85] W. Triebel and A. Singh. *The 8086 Microprocessor Architecture, Software and Interfacing Techniques.* Prentice-Hall, 1985.

[TW+88] C. Tseng, Ruey-Sing Wei, et al. Bridge: A High Level Synthesis System in Industry. In *25th Design Automation Conference*, 1988.

[Van90] Vantage Analysis Systems. *Vantage Analysis Systems Analyst User's Guide*, September 1990.

[VG88] N. VanderZanden and D. Gajski. MILO: A Microarchitecture and Logic Optimizer. In *25th Design Automation Conference*, 1988.

[VNG90] F. Vahid, S. Narayan, and D. Gajski. Synthesis from Specifications: Basic Concepts. In *TECHCON'90*, 1990.

[WCG90] C. H. Wu, G. D. Chen, and D. Gajski. Silicon Compilation from Register Transfer Schematics. In *International Symposium on Circuits and Systems*, 1990.

[Z+80] G. Zimmerman et al. MDS - The MIMOLA Design Method. *Journal of Digital Systems*, IV(3), 1980.

Appendix A:

VHDL Coding Practices for Structured Modeling

In order for the VSS system to synthesize a VHDL input description, a set of Structured Modeling conventions were established. The following coding practices should be adhered to for the modeling of storage elements and intercommunication signals between design entities.

1. Variables defined within a process are mapped to registers. Signals defined within a block with no signal_kind specification are mapped to wires. Signals defined within a block of signal_kind REGISTER (specified using a VSS annotation) are mapped to registers. Signals defined within a block of signal_kind BUS (specified using a VSS annotation) are mapped to a bus component.
2. Signals defined globally within the main architecture body are considered to be intercommunication signals which will be mapped to wires. Similarly, signals defined within a block (which are not of signal_kind REGISTER or BUS) are treated as intercommunication signals within the scope of that block (for example, interconnections between sub-blocks defined within this current block).

3. Each defined global signal must have ONLY ONE source block/process and AT LEAST ONE destination block/process. If either of these conditions is violated, VSS will have a problem in making the interconnections between blocks/processes.

4. The global signal may not be READ from and WRITTEN to in the same block or process. This is due to the fact that it is difficult within VSS to recognize bus, register and inout port components which are accessed in this manner in the same design entity.

   If a value is to be READ and WRITTEN to in the same process (block), a variable (signal of signal kind REGISTER or BUS) should be used to perform any data manipulations within the process (block). If this value is to be transmitted to other processes (blocks) in the design, an assignment of this value to a global signal should be made at the end of the process (block). This global signal can then be read in other processes (blocks).

   Multiple updates (WRITEs) to the signal may occur within that process in various conditional branches. A conditional or selected signal assignment can be used in the block to assign values to a signal under different conditions.

5. All signals/variables defined within the scope of a process or block should have an unique name. Use of the same signal/variable name locally in two different blocks/processes is not supported in VSS.

6. If selected bit updates (WRITEs) to a signal/variable representing a storage element are to be made, the n-bit BIT_VECTOR signal/variable should be modeled as n 1-bit signals/variables. These 1-bit accesses may be collected in one process/block back into a BIT_VECTOR form using a concatenation operator. The output of this concatenation operator may be assigned to a

global signal which can communicate this value to other processes/blocks. A selected bit READ access of this BIT_VECTOR can be made using the global signal.

For example:

```
architecture ex of example is

  signal X: BIT_VECTOR(7 downto 0);

begin

   block1: block

      --VSS: signal_kind REGISTER
      signal X_7,X_6,X_5,X_4,X_3,X_2,X_1,X_0: BIT;

   begin

      X_7 <= '1';
      X_4 <= '0';
      ...

      X <= X_7 & X_6 & X_5 & X_4 & X_3 & X_2 & X_1 & X_0;

   end block block1;

   block2: block
      signal Y: BIT;
   begin

      Y <= X(5) or X(2);

   end block block2;

end ex;
```

7. Registers are modeled in a concurrent dataflow (block) model as follows:

   - The signal representing the register must be annotated as being of signal kind register (either using a VSS comment annotation just prior to the declaration of the signal, or using the signal_kind qualifier REGISTER in the signal definition).

- To model a simple clocked register, a guarded signal assignment is used. The clocking event should appear in the block guard of the block in which the assignment is made.
- To model asynchronous events which affect the register signal, use a guarded conditional assignment statement. The event which triggers the action should appear in the block guard. The block guard may consist of more than one event expressions ORed together.
- If there are multiple events which may cause an assignment to the signal, a waveform clause for each value/event pair should be used. The order in which these waveform clauses appears determines the priority of the events.
- Where possible, signals in the block guard which are used to generate the event expression should be typed to identify their purpose. Types currently recognized by VSS include CLOCK, SET and RESET.

For example, the following VHDL code fragment models a register update with an asynchronous clear which overrides a synchronous count:

```
signal CLK: CLOCK;
signal X: RESET;

block ((CLK = '1' and not CLK'STABLE) or X = '1')

   signal REG: bit register;

begin

    REG <= guarded
       '0' when (X = '1') else
       REG + 1 when CNT_UP = '1' else
       REG;

end block;
```

8. Buses are modeled in a concurrent dataflow (block) model as follows: WRITEs to the bus are made using a single conditional signal assignment to a signal of signal_kind BUS. The condition associated with each waveform clause should be used to enable the data value specified in that waveform clause to be assigned to the bus signal. The following example illustrates a signal assignment which represents a bus:

```
block

--VSS: signal_kind BUS
   signal a_bus: BIT_VECTOR(7 downto 0);

begin

   a_bus <=
      data1 when enable1 = '1' else
      data2 when enable2 = '1' else
      a_bus;

end block;
```

9. The use of inout ports are not currently supported in VSS. These ports must be modeled as a pair of input/output ports (for example, an inout port 'A' should be modeled as A_in and A_out). READs of the ports use the input port; WRITEs use the output port.

10. VSS will handle single and two-dimensional arrays. Single dimensional arrays (BIT_VECTORs) are mapped to model n-bit signals and registers. Two-dimensional arrays are mapped to a MEMORY or REGISTER_FILE GENUS component.

    The following type and signal/variable declarations should be used to model a MEMORY or REGISTER_FILE:

```
type MEMORY is array (INTEGER range <>)
   of BIT_VECTOR(11 downto 0);
variable STACK : MEMORY(5 downto 0);      -- stack register file
```

The above declarations define a MEMORY with 6 words. Each word consists of 12 bits. The type MEMORY is a special type recognized by VSS. Signals/variables defined to be of this type are mapped to the appropriate GENUS component.

11. VSS does not support the following VHDL language features:

    - enumerated types
    - aliases
    - CONSTANT declarations
    - null statements
    - exit statements
    - return statements
    - loop statement with no iteration scheme, i.e.,

```
loop
   sequence of statements
end loop;
```

Appendix B:

VHDL Benchmark Descriptions

This appendix contains selected VHDL source descriptions either discussed within this dissertation or used as benchmarks to verify the operation of the VHDL Synthesis System (VSS) and the concepts of Structured Modeling.

Behavioral Rockwell Counter Model

```
------------------------------------------------------------------
--
-- Rockwell Counter Benchmark
--    Modified Behavioral (process) description
--    Copyright (c) 1990 by Joe Lis
--
------------------------------------------------------------------

use work.bit_functions.all;

entity RWC is
   port (CLK : in CLOCK;
         RST : in RESET;
         LDE : in BIT;
         DTI : in INTEGER range 0 to 4095;
         DTO : out INTEGER range 0 to 4095
         );
end RWC;

--VSS: design_style BEHAVIORAL

architecture BEH of RWC is

begin

process (CLK)

variable DTO_REG: INTEGER range 0 to 4095;

begin

--VSS: transform
   if (RST = '1') then DTO_REG := 0;
   elsif (LDE = '1') then DTO_REG := DTI;
   elsif (DTO_REG = 3327) then DTO_REG := 0;
   elsif (DTO_REG <= 3119) then DTO_REG := DTO_REG + 208;
   elsif (DTO_REG <= 3301) then DTO_REG := DTO_REG - 3094;
   else DTO_REG := DTO_REG - 3301;
   end if;

DTO <= DTO_REG;

end process;

end BEH;
```

Functional Rockwell Counter Model

```
------------------------------------------------------------
--
-- Rockwell Counter Benchmark
--    Functional (block) description
--    Copyright (c) 1990 by Joe Lis
--
------------------------------------------------------------

use work.bit_functions.all;

entity rw_cntr_func is
   port (CLK : in CLOCK;
         RST,LDE : in BIT;
         DTI : in INTEGER range 0 to 4095;
         DTO : out INTEGER range 0 to 4095
 );
end rw_cntr_func;

--VSS: design_style FUNCTIONAL

architecture FTNAL of rw_cntr_func is
begin

main: block (CLK = '1' and not CLK'STABLE)

--VSS: signal_kind REGISTER
signal DTO_REG: INTEGER range 0 to 4096;
signal A0,A1,A2,A3: BIT;

begin
A0 <= '1' when (RST = '0') AND (LDE = '1') else '0';
A1 <= '1' when (RST = '0') AND (A0 = '0') AND (DTO_REG = 3327) else '0';
A2 <= '1' when (RST = '0') AND (A0  = '0') AND (A1 = '0') AND
               (DTO_REG <= 3119) else '0';
A3 <= '1' when (RST = '0') AND (A0 = '0') AND (A1 = '0') AND
      (A2 = '0') AND (DTO_REG <= 3301) else '0';

with (RST & A0 & A1 & A2 & A3)  select
DTO_REG <=  guarded
  0               when B"10000"|B"00100",
  DTI             when B"01000",
  DTO_REG + 208   when B"00010",
  DTO_REG - 3094  when B"00001",
  DTO_REG - 3301  when B"00000",
  DTO_REG         when others;

DTO <= DTO_REG;

end block;
end FTNAL;
```

AM2910 Functional Model

```
------------------------------------------------------------------
--
-- AM2910 Microprogram Sequencer
-- Functional Model
--
-- Source: Adapted from an ISPS description in
--         "Computer Structures: Principles and Examples
--         by Siewiorek, Bell and Newell
--
--    Copyright (c) 1990 by Joe Lis
--
------------------------------------------------------------------

use work.bit_functions.all;

entity AM2910 is
   port (
        CLK: in CLOCK;                          -- clock
        CI : in BIT;                            -- carry in
        CC : in BIT;                            -- condition code
        CCEN : in BIT;                          -- cond. code enable
        RLD: in BIT;                            -- R register load
        D: in BIT_VECTOR(11 downto 0);          -- direct inputs
        I: in BIT_VECTOR(3 downto 0);           -- 2910 instruction
        OE: in BIT;                             -- output enable
        Y_OUT: out BIT_VECTOR(11 downto 0);      -- output instruction word
        ENABL: out BIT_VECTOR(2 downto 0); -- enable conditions
        FULL: out BIT                           -- stack full flag
);
end AM2910;

-- VSS: design_style FUNCTIONAL

architecture DATAFLOW of AM2910 is

begin

main: block(CLK = '1' and not CLK'STABLE)

--VSS: signal_kind REGISTER
   signal uPC: BIT_VECTOR(11 downto 0);    -- microprogram counter
--VSS: signal_kind REGISTER
   signal AR  : BIT_VECTOR(11 downto 0);    -- address register
--VSS: signal_kind REGISTER
   signal SP : BIT_VECTOR(2 downto 0);     -- stack pointer

   type MEMORY is array (INTEGER range <>) of BIT_VECTOR(11 downto 0);
   signal STACK : MEMORY(5 downto 0);      -- stack register file
   signal FAIL: BIT;                         -- CC fail flag
   signal Y: BIT_VECTOR(11 downto 0);    -- Y output signal
```

```
begin
        ENABL <=
           B"011" when (I = B"010") else
           B"101" when (I = B"110") else
           B"110";

        FAIL <= (not CCEN) and CC;

        Y <=
           D                    when ( ((I = X"1") and (FAIL = '0')) or
                                (I = X"2") or
                                ((I = X"3") and (FAIL = '0')) or
                                ((I = X"5") and (FAIL = '0')) or
                                ((I = X"6") and (FAIL = '0')) or
                                ((I = X"7") and (FAIL = '0')) or
                                ((I = X"9") and (AR /= B"000000000000")) or
                                ((I = X"B") and (FAIL = '0')) or
                                ((I = X"F") and (AR = B"000000000000") and
(FAIL = '1'))
         ) else
  B"000000000000" when (I = X"0") else
  uPC             when (((I = X"1") and (FAIL = '1')) or
                        ((I = X"3") and (FAIL = '1')) or
                        (I = X"4") or
                        ((I = X"6") and (FAIL = '1')) or
                        ((I = X"8") and (AR = B"000000000000")) or
                        ((I = X"9") and (AR = B"000000000000")) or
                        ((I = X"A") and (FAIL = '1')) or
                        ((I = X"B") and (FAIL = '1')) or
                        (I = X"C") or
                        ((I = X"D") and (FAIL = '0')) or
                        ((I = X"F") and (AR = B"000000000000") and
                             (FAIL = '0')) or
                        ((I = X"F") and (AR /= B"000000000000") and
                              (FAIL = '0'))
                             ) else
  AR              when (((I = X"5") and (FAIL = '1')) or
                        ((I = X"7") and (FAIL = '1'))
                             ) else
  STACK(BIN_TO_INT(SP))       when (
                              ((I = X"8") and (AR /= B"000000000000")) or
                              ((I = X"A") and (FAIL = '0')) or
                              ((I = X"D") and (FAIL = '1')) or
                              ((I = X"F") and (AR /= B"000000000000") and
                              (FAIL = '1'))
                              ) else
          Y;

       SP <= guarded
          B"000"      when (I = X"0") else
          SP + B"001" when (
                           ((SP /= B"100") and ((I = X"1") or (I = X"4"))) or
                           ((I = X"8") and (SP /= B"000") and
                            (AR = B"000000000000")) or
```

```
                              ((SP /= B"000") and (FAIL = '0') and
                               ((I = X"A") or (I = X"B") or (I = X"D")))
                              ) else
            SP;

         FULL <=
            '1' when ((I = X"0") or
                      ((I = X"1") and (SP /= B"100")) or
                      ((I = X"4") and (SP /= B"100")) or
                      ((I = X"8") and (SP /= B"000") and
                       (AR = B"000000000000")) or
                      ((I = X"A") and (FAIL = '0') and (SP /= B"000")) or
                      ((I = X"B") and (FAIL = '0') and (SP /= B"000")) or
                      ((I = X"B") and (FAIL = '0') and (SP /= B"000"))
                     ) else
            '0';

   STACK(BIN_TO_INT(SP)) <=
      uPC  when ((I = X"1") or (I = X"4") or (I = X"5")) else
      STACK(BIN_TO_INT(SP));

   AR <= guarded
      D                          when (((I = X"4") and (FAIL = '0')) or
                                       (I = X"C")
                                      ) else
             AR - B"000000000001"   when (((I = X"8")
                                       and (AR /= B"000000000000"))
         ) else
             AR;

         uPC <= guarded Y + (B"00000000000" & CI);

         Y_OUT <=
            Y when (OE = '0') else
            B"000000000000";

   end block;

   end DATAFLOW;
```

AM2910 Behavioral Model

```
------------------------------------------------------------------
--
-- AM2910 Microprogram Sequencer
-- Behavioral Model
--
-- Source: Adapted from an ISPS description in
--         "Computer Structures: Principles and Examples
--         by Siewiorek, Bell and Newell
--
--    Copyright (c) 1990 by Joe Lis
--
------------------------------------------------------------------

use work.bit_functions.all;

entity AM2910 is
   port (
        CLK: in CLOCK;                      -- clock
        CI : in BIT;                        -- carry in
        CC : in BIT;                        -- condition code
        CCEN : in BIT;                      -- condition code enable
        RLD: in BIT;                        -- R register load
        D: in BIT_VECTOR(11 downto 0);      -- direct inputs
        I: in BIT_VECTOR(3 downto 0);       -- 2910 instruction
        OE: in BIT;                         -- output enable
        Y: out BIT_VECTOR(11 downto 0);     -- output instruction word
        ENABL: out BIT_VECTOR(2 downto 0); -- enable conditions
        FULL: out BIT                       -- stack full flag
        );
end AM2910;

-- VSS: design_style BEHAVIORAL

architecture BEHAVIOR of AM2910 is


-- output instruction wd signal
   signal Y_sig: BIT_VECTOR(11 downto 0);
-- enable conditions signal
   signal ENABL_sig: BIT_VECTOR(2 downto 0);
   signal FULL_sig: BIT;          -- stack full flag

begin

--VSS: transform
process

   variable uPC: BIT_VECTOR(11 downto 0);    -- microprogram counter
   variable AR  : BIT_VECTOR(11 downto 0);    -- address register
   variable SP : BIT_VECTOR(2 downto 0);     -- stack pointer
```

```
   type MEMORY is array (INTEGER range <>) of BIT_VECTOR(11 downto 0);
   variable STACK : MEMORY(5 downto 0);      -- stack register file
   variable FAIL: BIT;                         -- CC fail flag
   variable Y_var: BIT_VECTOR(11 downto 0); -- output instruction wd signal

begin

        if (I = B"010") then
           ENABL <= B"011";
        elsif (I = B"110") then
           ENABL <= B"101";
        else
           ENABL <= B"110";
        end if;

        FAIL := (not CCEN) and CC;

        case I is
           when X"0" =>                          -- JZ instruction
              Y_var := B"000000000000";
              SP := B"000";
              FULL <= '1';
           when X"1" =>                          -- CJS instruction
              if (FAIL = '1') then
                 Y_var := uPC;
              else
                 Y_var := D;
              end if;

              -- push

              if (SP = B"100") then
                 FULL <= '0';
              else
                 FULL <= '1';
                 SP := SP + B"001";
              end if;
              STACK(BIN_TO_INT(SP)) := uPC;

           when X"2" =>                          -- JMAP instruction
              Y_var := D;
           when X"3" =>                          -- CJP instruction
              if (FAIL = '1') then
                 Y_var := uPC;
              else
                 Y_var := D;
              end if;
           when X"4" =>                          -- PUSH instruction
              Y_var := uPC;

              -- push

              if (SP = B"100") then
                 FULL <= '0';
```

```
      else
         FULL <= '1';
         SP := SP + B"001";
      end if;
      STACK(BIN_TO_INT(SP)) := uPC;

      if (FAIL = '0') then
         AR := D;
      end if;
   when X"5" =>                    -- JSRP instruction
      if (FAIL = '1') then
         Y_var := AR;
      else
         Y_var := D;

         -- push

         if (SP = B"100") then
            FULL <= '0';
         else
            FULL <= '1';
            SP := SP + B"001";
         end if;
         STACK(BIN_TO_INT(SP)) := uPC;
      end if;
   when X"6" =>                    -- CJV instruction
      if (FAIL = '1') then
         Y_var := uPC;
      else
         Y_var := D;
      end if;
   when X"7" =>                    -- JRP instruction
      if (FAIL = '1') then
         Y_var := AR;
      else
         Y_var := D;
      end if;
   when X"8" =>                    -- RFCT instruction
      if (AR = B"000000000000") then
         Y_var := uPC;

         -- pop

         if (SP /= B"000") then
            SP := SP + B"001";
            FULL <= '1';
         end if;
      else
         Y_var := STACK(BIN_TO_INT(SP));
         AR := AR - B"000000000001";
      end if;
   when X"9" =>                    -- RPCT instruction
      if (AR = B"000000000000") then
         Y_var := uPC;
```

```
      else
         Y_var := D;
         AR := AR - B"000000000001";
      end if;
   when X"A" =>                        -- CRTN instruction
      if (FAIL = '1') then
         Y_var := uPC;
      else
         Y_var := STACK(BIN_TO_INT(SP));

         -- pop

         if (SP /= B"000") then
            SP := SP + B"001";
            FULL <= '1';
         end if;
      end if;
   when X"B" =>                        -- CJPP instruction
      if (FAIL = '1') then
         Y_var := uPC;
      else
         Y_var := D;

         -- pop

         if (SP /= B"000") then
            SP := SP + B"001";
            FULL <= '1';
         end if;
      end if;
   when X"C" =>                        -- LDCT instruction
      Y_var := uPC;
      AR := D;
   when X"D" =>                        -- LOOP instruction
      if (FAIL = '1') then
         Y_var := STACK(BIN_TO_INT(SP));
      else
         Y_var := uPC;

         -- pop

         if (SP /= B"000") then
            SP := SP + B"001";
            FULL <= '1';
         end if;
      end if;
   when X"E" =>                        -- CONT instruction
      Y_var := uPC;
   when X"F" =>                        -- TWB instruction
      if (AR = B"000000000000") then
         if (FAIL = '1') then
            Y_var := D;
         else
            Y_var := uPC;
```

```
                    -- pop

                    if (SP /= B"000") then
                       SP := SP + B"001";
                       FULL <= '1';
                    end if;
                 end if;
              else
                 if (FAIL = '1') then
                    Y_var := STACK(BIN_TO_INT(SP));
                 else
                    Y_var := uPC;

                    -- pop

                    if (SP /= B"000") then
                       SP := SP + B"001";
                       FULL <= '1';
                    end if;
                 end if;
                 AR := AR - B"000000000001";
              end if;
        end case;

        uPC := Y_var + (B"00000000000" & CI);

-----------------------------------------------
-- assignment to external ports
-----------------------------------------------

   if (OE = '0') then
      Y <= Y_var;
   else
      Y <= B"000000000000";
   end  if;

end process;

end BEHAVIOR;
```

Appendix C:

GENUS Component Transistor Count

The following table lists the transistor counts used to evaluate the results of VSS synthesis experiments.

In the table, n refers to the number of bits, and i represents the number of data inputs.

The SIMPLE_ALU component performs the following functions: ADD, SUB, LID (left identifier), RID (right identifier), AND, OR, LNOT (invert left input), RNOT (invert right input).

GENUS Component	Trans. per bit	GENUS Component	Trans. per bit
ADDER	32(n - 1) + 16	NAND3	6
ADD_SUB	34(n - 1) + 12	NAND4	8
ADD_SUB_LID_RID	56	NOR2	4
ALU	100	NOR3	6
AND	6	NOR4	8
BUFFER	4	OR	6
COMPAR_EQ	14	REGISTER	48
COMPAR_LGE	35	REGISTER_FILE	54
CONSTANT	0	SHIFTER	12
DECODER	20	SIMPLE_ALU	92
EXTRACT	0	SUB	34(n - 1) + 12
INC_DEC	18	TRI_STATE	12
LATCH	32	UP_COUNTER	52
MUX	$6i + 2log_2\ i$	UP_DOWN_COUNTER	58
NOT	2	XOR	10
NAND2	4	XNOR2	10