A Framework for Synthesis from VHDL

by

Sandeep R. Shah

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

[illegible]

Dr. J. G. Tront, Chairman

J. R. Armstrong

Dr. J. R. Armstrong

[illegible]

Dr. D. S. Ha

W. R. Cyre

Dr. W. R. Cyre

December, 1991

Blacksburg, Virginia

C. ~

5655
V855
1991
S523
C.2

A Framework for Synthesis from VHDL

by

Sandeep Shah

Dr. Joseph G. Tront

Electrical Engineering

(ABSTRACT)

This thesis describes the design and implementation of a hardware synthesis system based on design descriptions provided in VHDL. Several aspects of the synthesis problem are examined. These include the design of an internal format to represent multiple levels of design information, algorithms for synthesis, optimizations, and verification of the synthesis process. Key features of this system include the ability to synthesize models that span a wide range of design description abstraction levels. The synthesis system internal format contains data structures for algorithmic, dataflow, as well as structural VHDL constructs. This framework for performing synthesis over a wide range of abstraction levels is the novel feature of this system. Optimizations for register-transfer level (dataflow) models are discussed along with their implementation. The design and implementation of the synthesis library, which contains information about the hardware components available to perform the synthesis, is also discussed.

The output of the synthesis system is in the form of two files, an RNL format netlist and a purely structural VHDL netlist. In order to produce the actual hardware layout, the RNL netlist must be input to VPNR, a standard cell place and route system. The structural VHDL may be simulated to verify the synthesis process. Results of mixed level synthesis are provided.

Acknowledgements

I would like to thank Dr. Joseph G. Tront for the support and guidance that he has provided. Working under him has enabled me to learn much about the many disciplines in the Computer Engineering field. I thank him for his patience and encouragement especially when I was not productive. I would like to thank Dr. James R. Armstrong for allowing me to work on the VHDL Validation Suite project and broaden my knowledge of VHDL. I would also like to thank Dr. Dong S. Ha and Dr. Walling R. Cyre for serving on my committee.

I am grateful to my family for their encouragement and mental support during my college career. Without them, I could not have completed my degree. I also wish to thank Bob Lineberry, Chang Cho, Manish Modi, Chakravarthy Kosaraju, Balraj Singh and Anil Sama for their friendship and interesting talks. I have learned much from them. Finally, I wish to thank God for providing so much joy and happiness in my life. My life and work would be meaningless without Him.

Table of Contents

Chapter 1
Introduction

Trends in today's electronics industry indicate that component complexity doubles every few years. With this kind of exponential growth, design methodologies must be frequently updated in order to keep up with the growing amount of information contained in each design description. Designs may be composed of hundreds of thousands of gates or millions of transistors. Thus, the design cycle becomes time consuming and prone to errors. It becomes necessary to design at a higher abstraction level than that which is currently used. Over the last several years, hardware description languages have gained acceptance and popularity in the design community. The VHSIC Hardware Description Language (VHDL), now designated as IEEE Std 1076-1987 [1], is a very powerful language that allows the user to model hardware at a behavioral or functional level, as well as at the traditional register-transfer and gate levels.

Using the behavioral constructs available in VHDL, a design may be simulated and verified without explicitly stating the structure of the hardware. High-level synthesis is a process which consists of taking the behavioral specification of a system along with a set of constraints and goals to be satisfied (e.g., minimize for size or speed), and to find a structure that implements the behavior while still satisfying the goals and constraints. Current synthesis research is being preformed from languages such as ISPS [2], HardwareC [3], V [4], VHDL [5,6], as well as other hardware description languages. Most of these systems don't allow integrating levels of design at the source level (i.e., mixing structural, dataflow, and algorithmic models).

There are several advantages in performing automated synthesis. If the design process is automated, the length of the design cycle is reduced and is less prone to errors. An automated synthesis system also allows the user to search the design space. Thus, the designer can explore trade-offs between cost, speed, power, etc. IC technology becomes accessible to designers who may not be experts in IC design. Designing at a higher level allows the designer to focus on functionality rather than detail.

This thesis examines the advantages of automated synthesis and the tasks involved in the synthesis process. The focus is on VTsynth, a synthesis system that performs synthesis from VHDL descriptions. The internal format of the VTsynth system contains a single representation which contains data structures to represent all levels of design information. Therefore, synthesis can be performed on models that span several levels of the design hierarchy. The advantages of the intermediate format and the design of the tools that make up the system will be described. One of the main objectives of this project is to develop a framework for synthesis that: 1) includes constructs to handle structural, dataflow, and algorithmic VHDL, and 2) is easily expandable to incorporate other synthesis algorithms. A big issue in current software design is the upgradability and maintainability of software. This issue was taken into consideration while designing this synthesis system.

Figure 1 shows a block diagram of the VTsynth synthesis system. The system is implemented on an Intel 80386 based personal computer operating under the MS-DOS operating system. The software is written in the C programming language and compiled using the Microsoft C compiler [7]. The parser and internal format were adapted from Yacc and C source code for a VHDL simulator from the University of Pittsburgh [30].

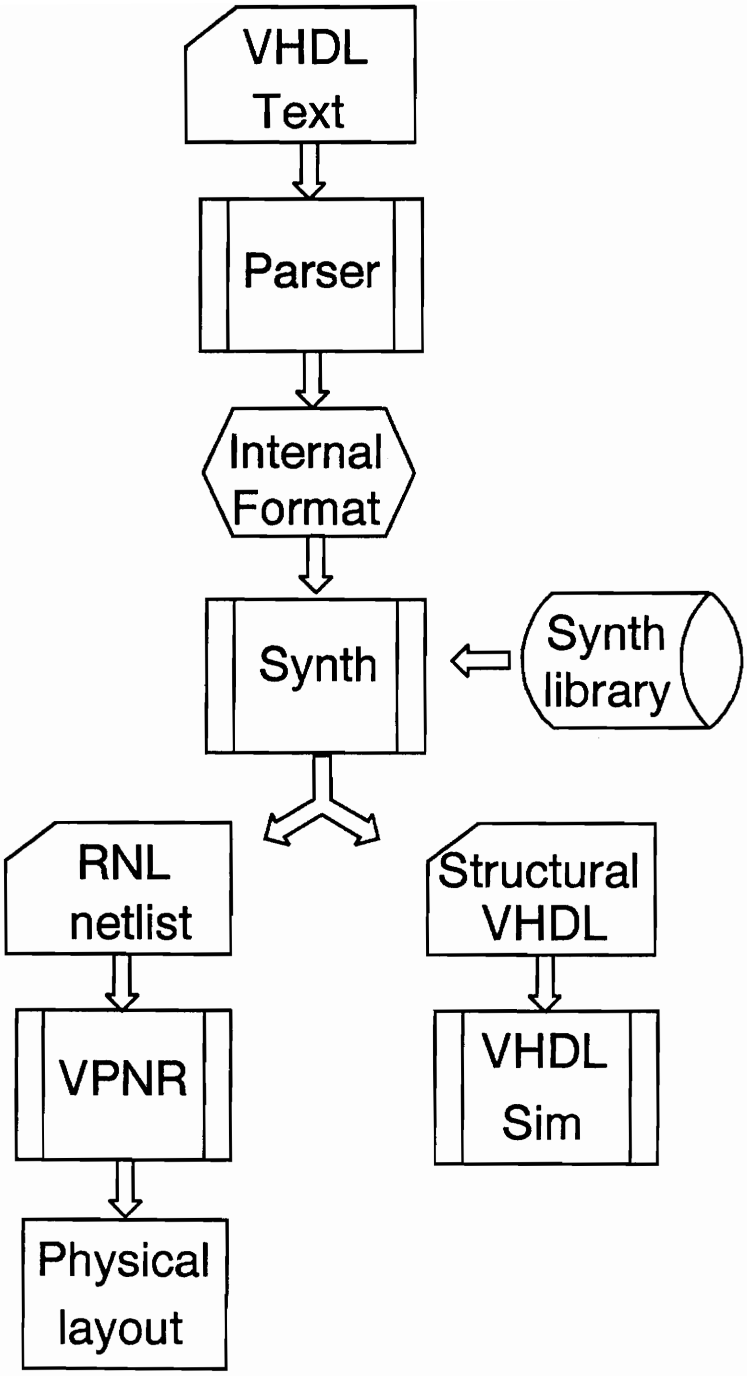


Figure 1. Diagram of the VTsynth Synthesis System

The Yacc and C descriptions were updated to include support for synthesis. The synthesis library contains information such as the set of components available during synthesis as well as the size and speed characteristics of those components. The output of the synthesis program is in the form of two files, an RNL format netlist and a purely structural VHDL model. The RNL [31] netlist is sent to the VPNR (Vanilla Place aNd Route System) [19] software package, provided by MCNC (Microelectronics Center of North Carolina), in order to obtain a physical layout. The structural VHDL model may be simulated to verify the synthesis process. Also, the designs output by the synthesis system may be used in future models for synthesis.

The VTsynth system is designed for users who desire physical layouts from VHDL descriptions. The user must be familiar with the VHDL language as defined by the IEEE Standard VHDL Language Reference Manual (LRM) [1]. The models may be structural, dataflow, or algorithmic, provided they only contain constructs contained in the VHDL subset for synthesis described in Chapter 3. Structural parts of a model may contain components from the synthesis standard cell library (described in Appendix A) or previously synthesized models.

1.1 Thesis Organization

The remainder of the thesis is organized as follows: Chapter 2 describes a collection of synthesis systems from the literature, several of which are still being developed. Chapter 3 reviews the modeling constructs of VHDL and examines their suitability for synthesis. Advantages and disadvantages of using VHDL for synthesis are also discussed. The subset of VHDL implemented by the VTsynth program is also given in this chapter. Chapter 4 describes the tasks involved in synthesis. Synthesis from different types of models are examined, along with the their internal representation, optimizations, and transformations leading to a physical layout. An example is presented which compares the synthesis of a structural view, dataflow view, and algorithmic view of the same model. The design of the VTsynth system is described in Chapter 5. Included in this chapter is a description of the Yacc and Lex descriptions of the parser, the format of the internal data structures, the synthesis library, the algorithms for performing synthesis, and a description of the program output. The process of obtaining physical layouts and verifying the synthesis results are also discussed.

Chapter 6 details a set of results produced by the VTsynth system. The input VHDL description, the resulting RNL netlist and structural VHDL, as well as the layouts generated by VPNR are shown for several examples. These models include a purely structural model, a dataflow model, an algorithmic model, and several mixed-level models. Synthesis results from models of varying sizes are given along with some system execution statistics.

Chapter 7 describes how the VTsynth system may be enhanced in the future. The internal data format is currently able to store loop constructs within sequential processes, but the program is not yet able to generate a structure from such descriptions. A useful addition to this program would be the implementation of scheduling and allocation algorithms, some of which are described in Chapter 4, to increase the high-level synthesis capabilities of the program. The system currently uses fixed delay values for components in the synthesis library. These delay values are the worst case delay for a component with a fan-out of two inverters. A more accurate value may be calculated using knowledge of the exact fan-out available from the resulting netlist. This is also reserved for future implementation.

Chapter 8 is the user's manual for the VTsynth synthesis system. Included in this chapter is information on how to execute the program, a description of the command line parameters, and information on how to create a custom library for synthesis.

1.2 Definitions:

Listed below are some terms used in this paper:

High-Level synthesis or *Algorithmic-Level synthesis* is defined as a transformation from a behavioral specification to a structural one. The constructs used in the behavioral specification are similar to procedural constructs found in high-level programming languages such as Pascal, C, or Ada.

Logic synthesis is a transformation from a boolean specification (logic equations) to a technology dependent layout.

Behavioral modeling is a style of modeling which describes how a system functions or a mapping of the input to the output. The models may be algorithmic or dataflow.

Dataflow modeling uses register-transfer operations to form a behavioral description of the model.

A *structural model* is a netlist of interconnected components that make up a system.

Chapter 2
Background

Automated synthesis has been the object of research for over a decade. The emphasis on high-level synthesis has increased greatly over the last five years, as evident by the number of papers in recent computer literature [2,3,4,5,6]. The terminology has also changed over the years and is still sometimes ambiguous in current literature. Behavioral synthesis, high-level synthesis, and register-level synthesis may have the same meaning to one person, while having different meanings to another. A decade ago, the term "silicon compiler" implied a program that would compile a normal programming language into a layout, rather than into object code. Of course, no such program existed at the time. Soon after that, the term was used to mean the use of regular methods for converting logic into layout (PLA generator, gate arrays, gate matrices, etc.). As synthesis systems started to work at a higher abstraction level, a "silicon compiler" came to mean a system that constructs custom layouts from high-level languages. Today, there is still some confusion regarding the definitions of terms such as *silicon compiler* or *synthesis*. Terms such as behavioral synthesis, high-level synthesis, algorithmic-level synthesis, register-level synthesis, logic synthesis, etc. appear frequently in the synthesis literature. There seems to be much overlapping in the definitions of these terms. Definitions of the terminology used in this thesis were presented in Chapter 1.

There have been publications about several aspects of the synthesis task: these include languages for synthesis (or subsets of existing languages), algorithms for scheduling and allocation, intermediate formats for synthesis, style specific synthesis, synthesis for testability, etc. Described below are a few of the academic and industrial publications that have made significant contributions to the advancement of high-level synthesis.

One of the first synthesis programs was a system called Bristle Blocks [25]. This system accepted an ISP description of a processor and produced a complex MOS layout. Although that system never produced completely working layouts, it paved the way for ones that did.

2.1 Carnegie Mellon System Architect's Workbench

There have been dozens of papers and even a book about "The System Architect's Workbench" [2], the synthesis system being developed at Carnegie Mellon University. This system consists of several programs combined together to form a high-level synthesis system. The input to the system is an ISPS behavioral description. The behavioral description is translated to an internal format called the Value Trace (VT). The tools within the Workbench work on the VT internal format, altering it, or adding information when necessary. The output of this system is a register-transfer level data path and abstract controller. The output may then be sent to module generators, logic synthesis tools, and physical design tools. The Workbench offers two separate synthesis paths, one for general purpose synthesis and one for microprocessor synthesis.

ISPS is a computer hardware description language for describing the behavior of system level components. The basic memory element in ISPS is a *carrier*. A carrier may be a "bit structure" (comparable to bit_vector in VHDL) or a "word structure" (comparable to an array of bit_vectors in VHDL). ISPS contains constructs for sequential as well as concurrent execution, support for procedures and procedure calls, several looping constructs, and mechanisms for conditional execution. These constructs are comparable to the behavioral constructs of VHDL. However, the System Architect's Workbench contains no mechanisms to implement partial binding of structure at the input description level. Therefore, there is no capability of mixing structure and behavior at the source level. This may only be accomplished by manipulating the intermediate form.

In their book "Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench" [2], the authors from Carnegie Mellon University admit that languages such as VHDL and Verilog are preferable to the ISPS language because of the capabilities of mixing structural and behavioral models. Currently, they are developing VHDL and Verilog compilers in order to accept these languages as input to the Workbench.

2.2 IBM Yorktown Silicon Compiler

The Yorktown Silicon Compiler [4] is a synthesis system developed at the IBM T. J. Watson Research Center at Yorktown Heights, N.Y. The synthesis system accepts a behavioral model written in the V language (to maintain compatibility with other research within IBM) and translates it to the Yorktown Intermediate Format (YIF). The internal format represents control flow and data flow within the behavioral model. This format contains the necessary constructs to model sequential and concurrent system behavior and is stored internally as matrices and linked lists. As with the Carnegie Mellon system, structural models may not be mixed with behavioral models in the original V specification.

2.3 Honeywell VSYNTH System

The Honeywell VSYNTH system [6] is an extension of earlier work first begun in Germany that was based on the language MIMOLA [8]. The system consists of two major subsystems: the Process Graph Analyzer (PGA) and the MIMOLA Synthesis System (MSS). The PGA performs optimizations, control state generation, and storage allocation on the input and prepares the information in order to send it to the MIMOLA Synthesis System. The system only accepts sequential constructs. Only one VHDL process is allowed in a model. No constructs for concurrent modeling or structural modeling are supported.

2.4 The Olympus Synthesis System

The Olympus Synthesis System [3] is a synthesis system being developed at Stanford University. The input to the system is a language developed at Stanford called HardwareC. It is a high-level hardware description language with C-like syntax. The language contains several modeling semantics which are similar to those found in VHDL. These include procedural constructs as well as structural constructs which allow interconnection of modules. The building blocks of HardwareC are similar to those of algorithmic VHDL. These include blocks, processes, procedures, and functions. In the Olympus system, the HardwareC model goes through several synthesis steps and a technology mapping step. The output of the system is a netlist of interconnected components from a predefined library. Since HardwareC was designed for synthesis, it does not contain the robust simulation primitives found in VHDL. These include user defined types, signal attributes, inertial and transport delays, etc.

2.5 Other synthesis systems

The synthesis systems described so far have been instrumental in forming a base for synthesis research and tool development. There are several other synthesis system developments which have made significant contributions to synthesis research, but are not described here in detail. The first published results of synthesis from VHDL were from the University of California at Irvine. Their system, called VSS (VHDL Synthesis System) [5], accepts as input a subset of VHDL and translates it into an intermediate format called BIF (Behavioral Intermediate Format). Other synthesis systems include the ADAM [32] system from the University of Southern California, the Cathedral [33] system from the University of Leuven, Belgium, and the HAL [34] system from Carleton University.

2.6 Summary

The preceding literature survey shows that several synthesis efforts have been made in the past and synthesis continues to be an object of much current research. Several issues have hindered the emergence of powerful synthesis systems. In the past, it has been difficult to build on previous work since most of the synthesis research used internally developed hardware description languages. These include ISPS, V, HardwareC, VHDL, and a host of other languages not described here. Now, with the standardization of VHDL, the potential for building on previous work becomes much greater. Also, the languages used for synthesis in the past were designed specifically for synthesis and lacked semantics for accurate simulation. VHDL overcomes this limitation since it's simulation semantics are very well defined. Thus, VHDL provides a rich set of modeling constructs which are necessary for simulation as well as synthesis.

Since its standardization by the IEEE in 1987, VHDL has grown in popularity in the design community, as is evident by the number of VHDL simulators and design tools currently available. Because of its rich set of modeling constructs and its acceptance in the design community, it was selected as the input language for the VTsynth synthesis system.

Chapter 3
VHDL and Synthesis

VHDL is a very powerful hardware description language used to model digital systems. It contains constructs that can be used to model hardware from as high as the system level down to the gate level. Hardware models may span a wide range of the design hierarchy. A diagram of the design hierarchy is given in Figure 2. Each slice in the pyramid represents a level of abstraction. Moving down the abstraction hierarchy results in increasing the amount of detail that must be dealt with. At each level of abstraction, except the layout level, there exists primitives to describe the behavior and structure of the circuit. Table 1 lists the structural and behavioral primitives available at each level of the design hierarchy.

Table 1. Primitives within the Design Hierarchy [9]

Level	Structural Primitives	Behavioral Primitives
PMS	CPUs, memories, buses	Performance specifications
Chip	Microprocessors, RAMs, ROMs, UARTs, parallel ports	I/O response, algorithms, micro-operations
Register	Registers, ALUs, counters, MUXs	Truth tables, State tables, micro-operations
Gate	Gates, flip-flops	Boolean equations
Circuit	Transistors, R, L, C	Differential equations
Physical Layout	Geometrical objects	None

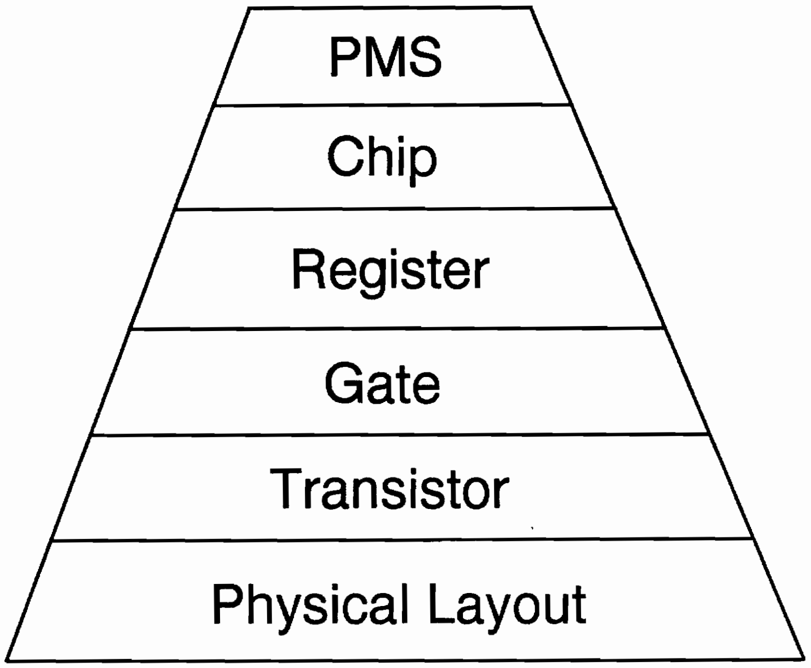


Figure 2. The Abstraction Hierarchy [9]

VHDL designs may consist of procedural constructs similar to those of high-level programming languages as well as structural constructs similar to those used to describe a netlist of components. This broad capability not only allows for a wide range of modeling techniques but also poses a great challenge for those designing tools for VHDL. Version 7.2 of the language was released in 1985 and the IEEE Standard VHDL (IEEE Std 1076-1987) [1] was approved in 1987. However, only recently have VHDL simulators [10,11,12,13] become available that implement a significant portion of the language and handle models of reasonable sizes. This is mainly due to the vastness and complexity of the language. The input to the VTsynth synthesis system is a subset of the IEEE Std 1076-1987 version of VHDL. The specific subset is described in detail in a later section.

Because the available modeling constructs span several abstraction levels, VHDL seems to be a good language for modeling hardware. But there are several issues that must be resolved before using VHDL for synthesis. First, the language has many constructs that are defined primarily for use in simulation. These include several signal attributes, abstract and user-defined data types, file operation functions, constructs to report messages during simulation, arbitrary waveforms, etc. These constructs may have no meaning in synthesis. Second, VHDL sequential statements are assumed to execute in zero time. This is easily handled by a simulator with an event queue, but is impossible when trying to allocate physical components to operations. All physical components have some delay associated with them. Third, because of the numerous modeling constructs available, the parser needed to read an input VHDL text file and the data structures needed to store the language will be very large. These issues that make VHDL difficult for synthesis are resolved by designing a subset of VHDL for synthesis and developing a design methodology for synthesis. The next section examines the use of VHDL for synthesis and defines restrictions imposed for synthesis.

3.1 VHDL as a Synthesis Language

VHDL is inherently a very large language allowing a very large range of modeling techniques. The vastness of the language can be illustrated by comparing it to some other languages. Listed below in Table 2 is a comparison of the modified Backus-Naur Form of four languages: Caltech Intermediate Format (CIF) [14], MODULA-2 [15], ANSI C [7], and VHDL [1]. The column labeled "Production Rules" shows the number of production rules in the language. An example of a production rule in VHDL is shown in Figure 3. A production rule consists of a left-hand side and a right-hand side separated by "::=". The left-hand side is called a *nonterminal symbol*. The right-hand side lists all the possible *formulations* that can exist to satisfy this rule. Each formulation is separated by a vertical bar ('|') character (meaning OR). The column labeled "Formulations" is a sum of all the formulations in the production rules of the BNF syntax.

```
sequential_statement ::=
     wait_statement
      | assertion_statement
      | signal_assignment_statement
      | variable_assignment_statement
      | procedure_call_statement
      | if_statement
      | case_statement
      | loop_statement
      | next_statement
      | exit_statement
      | return_statement
      | null_statement
```

Figure 3. VHDL Production Rule

Table 2. BNF Comparison of Four Languages

Language	Production Rules	Formulations
CIF	32	60
MODULA-2	72	147
ANSI C	79	211
VHDL	217	440

Note the large number of production rules and possible formulations for VHDL. This greatly increases the size of the parser needed to process the language. This requires the parser to handle a very large amount of data since there are a very large number of correct VHDL sentences. Thus, the complexity of the full VHDL language parser becomes several times more complex than say, a C language compiler.

Complexity is not the only reason to limit the user to a subset for synthesis. Several VHDL constructs have no meaning in the synthesis domain. For example, the VHDL constructs for library manipulation and file interaction through the operating system are meaningless in the synthesis domain. Also, several constructs are specific to simulation and have little correspondence to actual hardware. For example, the "assert" statement is used to test a condition and report a message string to the user. This is generally used to monitor the value of signals during simulation. Several signal attributes also fit into the simulator specific category. Defining a subset of VHDL for synthesis prevents these types of constructs from appearing in models that will be synthesized.

3.2 Subset of VHDL for Synthesis

The VHDL subset is designed to provide modeling flexibility. Constructs were selected from the various abstraction levels in order to demonstrate that mixing levels of design was possible during synthesis. Structural constructs were included in the subset so that a user may instantiate components from the library, or instantiate components that were previously synthesized. Dataflow constructs were included in the subset in order that the user may translate a truth table or Karnaugh map into a dataflow model and subsequently obtain a physical layout. Optimizations, which are discussed later, are made on the dataflow constructs in order to minimize the layout area and maximize the speed. Algorithmic constructs such as sequential signal and variable assignment statements, IF-THEN-ELSE statements, and CASE statements are included in the subset in order to show that the internal design representation is able to store all levels of design from the algorithmic to structural. Currently, no optimizations are preformed on the algorithmic models.

Several VHDL constructs were purposely excluded from the synthesis subset. The reasons for excluding them from the subset vary. Some constructs were excluded from the subset because their semantics are simulator specific. These include constructs such as the "assert" statement which is used to report messages during simulation. Also, signal attributes are not allowed at this time. Several signal attributes have no meaning in the synthesis domain. For example, the attribute 'LAST_EVENT returns a value of type TIME which represents the simulation time at which the signal was last assigned (the time at which 'EVENT was last true). It can be easily seen that these types of constructs were designed to ease simulation, but have no meaning in the synthesis domain. Therefore, a subset of VHDL was developed for synthesis which included structural, dataflow, and a number of algorithmic constructs, but excluded a portion of the language based on the reasons given above.

The parser shown schematically in Figure 1 accepts as input any models written in IEEE Standard VHDL. Section 5.1 describes how the stand alone parser is validated using models out of [9]. The internal data format contains data structures to store structural, dataflow, as well as most algorithmic VHDL constructs. The exact list of VHDL constructs that are supported by the internal format is given in Table 3. An error message is issued if a user tries to synthesize a model using VHDL constructs other that those for which the internal format is defined. The internal format is able to store more VHDL constructs than are currently synthesizable. The subset of VHDL which is currently synthesizable is shown in Table 4.

Table 3. VHDL constructs supported by the internal format

entity declaration
port clause (interface signals)
architecture body declaration
constant declaration
variable assignment expression
signal declaration
variable declaration
component declaration
label declaration
configuration declaration
component specification
port map clause
guarded signal specification
indexed name
slice name
logical expression
simple expression
relational expression
aggregates
signal assignment statement
single waveform element
if-then-else statements
case statement
loop statement
block statement
process statement
concurrent signal assignment statement
conditional signal assignment
component instantiation statement

Table 4. VHDL Subset for Synthesis

entity declaration
port clause (interface signals)
architecture body declaration
variable assignment expression
signal declaration
variable declaration
component declaration
component specification
logical expression
simple expression
signal assignment statement
single waveform element
block statement
process statement
concurrent signal assignment statement
component instantiation statement
if statement
case statement

3.3 Data Objects

There are three classes of objects in VHDL: constants, variables, and signals. The VTsynth system handles all these classes. Signals may appear in an interface description (i.e., ports) or an architecture body (i.e., local signals). Interface signals have a name, a data type, and a mode. A signal mode may be **in**, **out**, **inout**, **buffer**, or **linkage**. Local signals have a name and a data type. VHDL allows variable declarations only within process blocks. Variables have a name and an associated data type. The data types currently accepted by the VTsynth system are shown in Table 5. The operators supported for these data types are shown in Table 6.

Table 5. VHDL Data Types Supported

Data Type	Value
BIT	{0, 1}
BOOLEAN	{True, False}
BIT_VECTOR(m to n)	Array (m to n) of {0, 1}
INTEGER	up to 32 bits

Table 6. VHDL Operators supported by VTsynth

Operator	Argument Types Allowed
+/-	Integer
NOT	Bit, Boolean
+	Integer
-	Integer
*	Integer
/	Integer
AND	Bit, Boolean
OR	Bit, Boolean
XOR	Bit, Boolean
NAND	Bit, Boolean
NOR	Bit, Boolean
&	Bit, Bit_vector

3.4 Summary

A summary of VHDL was presented in this chapter. It was seen that VHDL is a very powerful language that is capable of modeling hardware at a variety of abstraction levels including algorithmic, dataflow, and structural levels. A comparison of VHDL to other languages showed that VHDL is indeed a very large and rich language in terms of modeling constructs. The need for creating a subset of VHDL for synthesis was presented. This was followed by a description of the actual subset of VHDL used in the implementation of the VTsynth system.

Chapter 4
Tasks involved in Synthesis

As mentioned earlier, VHDL contains constructs to model at the structural, dataflow, and algorithmic levels. The method by which physical layouts are derived from each modeling style (i.e., structural, dataflow, and algorithmic) is quite different. This section describes the process by which physical layouts are obtained from structural, dataflow, and algorithmic models. Algorithms for synthesis and optimization are also presented for dataflow and algorithmic models. The chapter concludes with an example of synthesis. Three descriptions of a "ones_counter" from [9] are presented. The structural, dataflow, and algorithmic models of the ones_counter are functionally equivalent, but are modeled using different VHDL constructs. Synthesis is performed on all three descriptions and the results discussed.

4.1 Synthesis from Structural Models

A structural model is essentially a netlist of interconnected components. In order to obtain a physical layout from a structural model, layouts for each of the components must be available. The layouts of the individual components may reside in the synthesis library, or may be previously synthesized models which reside outside the synthesis library.

A structural netlist may be represented in several forms, each containing the same information. Figure 4 shows the textual VHDL description of an XOR gate made with ANDs, ORs, and INVERTERs. Figure 5 shows its equivalent RNL description, and

Figure 6 shows its schematic representation. The keyword *macro* in the RNL description corresponds to the keyword *entity* in the VHDL description. The macro name in the RNL description is the same as the entity name in VHDL (xor_gate). The port list of the VHDL model appears in the RNL description after the macro name. The local signals of the VHDL model are declared in the RNL netlist using the *local* statement. Note that the components declared in the VHDL model are not declared in the RNL netlist. A fundamental assumption in the whole translation process is that the components used in the structural model exist in the library used by the VTsynth system. The VHDL component instantiation statements are translated to their equivalent RNL form (e.g., "A: i1 port map (a, ai);" in VHDL is translated to "(in a ai)" in RNL). A physical layout is obtained automatically by sending the RNL netlist of Figure 5 to VPNR, an automated cell place and route package. The layout produced by VPNR is shown in Figure 7.

The RNL description of the XOR gate of Figure 5 is slightly different than the model of the XOR gate in the VPNR standard cell library named *xors*. The XOR gate of Figure 5 is composed of inverters, AND gates, and OR gates. The gate named *xors* in the VPNR library is made up of complex gates. In a structural model, the user may choose to instantiate either model of the XOR gate. The primary advantage of using the VTsynth system with structural models is to obtain layouts without having to learn a lower level layout language. Thus, a user is able to obtain physical layouts from a purely structural VHDL model consisting of components from the VPNR library without having to learn RNL or the details of IC layout. The VHDL is automatically translated into RNL by the process described above. The RNL netlist is then sent to VPNR which produces a Magic [20] description of the layout.

```
entity xor_gate is
     port (a,b : in bit;
           q   : out bit);
end xor_gate;


architecture structural of xor_gate is

signal ai,bi,ci,di : bit;
component i1 is
     port (a : in bit;
           b : out bit);
end component;

component a2 is
     port (a,b : in bit;
           c : out bit);
end component;

component o2 is
     port (a,b : in bit;
      c : out bit);
end component;

begin

A: i1 port map (a,ai);
B: i1 port map (b,bi);
C: a2 port map (a,bi,ci);
D: a2 port map (b,ai,di);
E: o2 port map (ci,di,q);

end structural;
```

Figure 4. Structural description of an XOR gate

```
(macro xor_gate (a b q)
(local ai bi ci di)
(i1 a ai)
(i1 b bi)
(a2 a bi ci)
(a2 b ai di)
(o2 ci di q)
)
```

Figure 5. An RNL description of an XOR gate

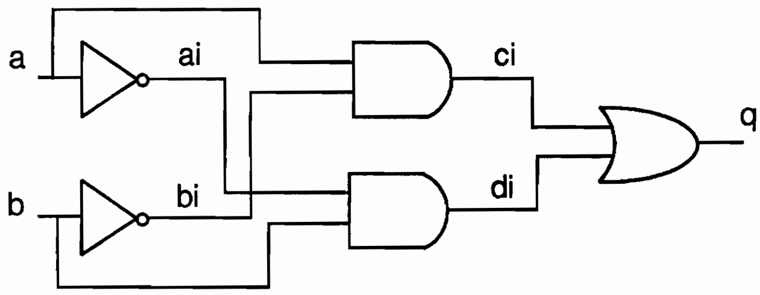


Figure 6. Schematic of XOR gate in Figure 5

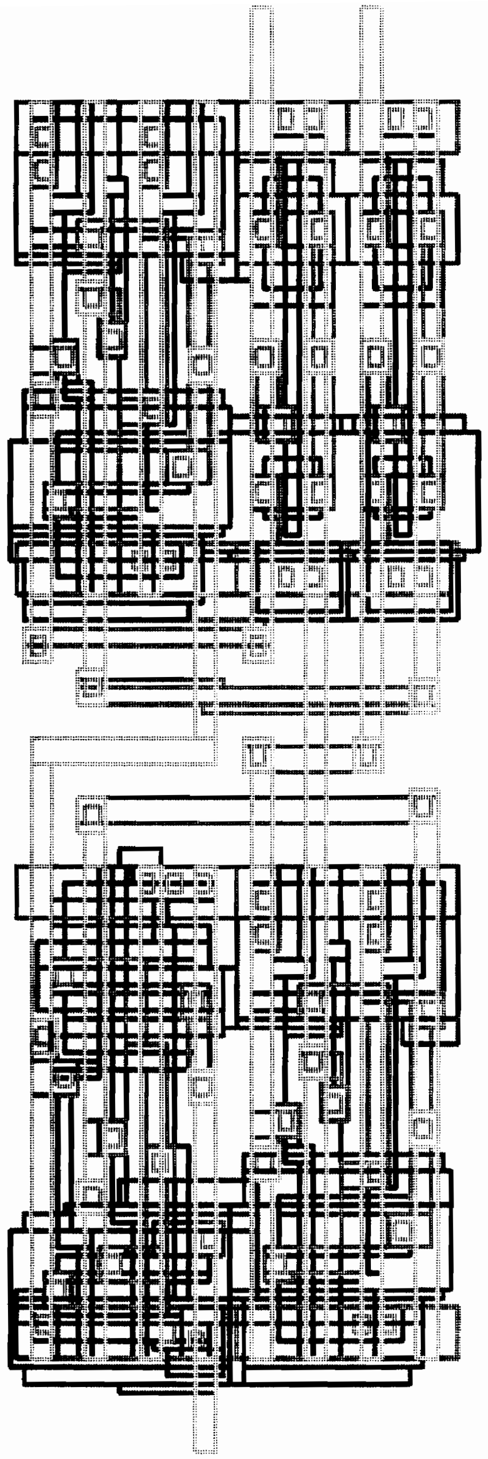


Figure 7. Resulting VPNR layout for XOR gate structural model

4.2 Synthesis from Dataflow Models

A dataflow model contains statements which describe behavior using operations such as AND, OR, NAND, XOR, NOT, +, -, /, etc. and the signal assignment operator (<=). In order to obtain a physical layout from a dataflow model, several steps must be taken. These steps include reading the VHDL text, storing the VHDL into internal data structures, extracting the logic, optimizing it, and mapping the logic to physical components from the available library.

The dataflow statements are first parsed and stored in the internal format. A statement is stored internally as a linked list of operations. Figure 8 shows the graphical internal representation of a statement. The internal node structure contains a pointer to the next statement in the block or process (or a NULL value if it is the last statement), pointers to the first and last operations in the statement, a text field for the label (if one exists), and a field to store the statement type (i.e., concurrent or sequential). Figure 9 shows how a series of operations within a statement are linked together. The statements are stored internally as a linked list of nodes.

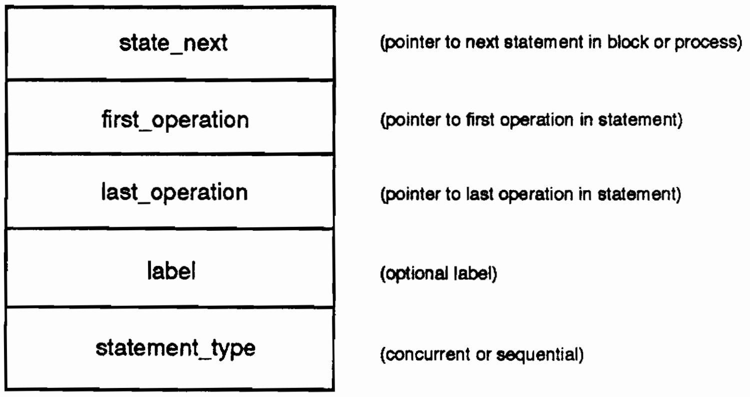

state_next
(pointer to next statement in block or process)
first_operation
(pointer to first operation in statement)
last_operation
(pointer to last operation in statement)
label
(optional label)
statement_type
(concurrent or sequential)

Figure 8. Internal Representation of a Statement

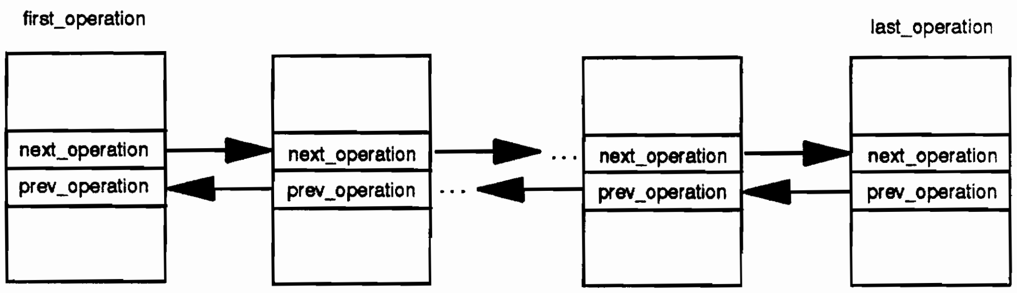


Figure 9. List of Operations within a Statement

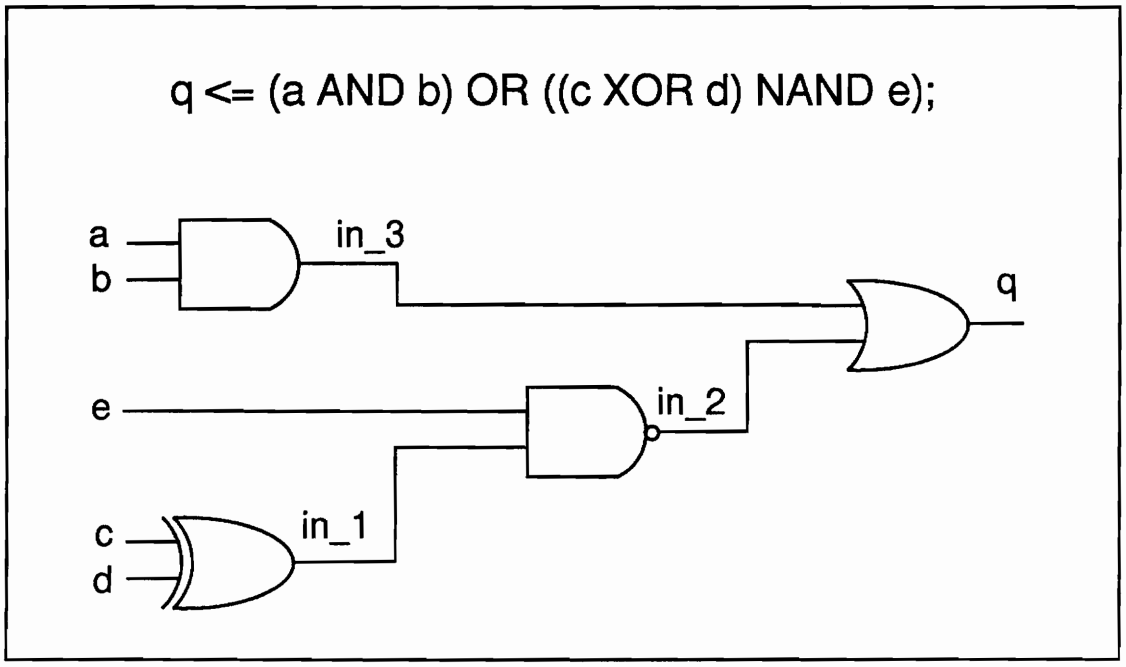


Figure 10. Schematic for a VHDL Dataflow Statement

Figure 10 shows an example of a dataflow statement and its equivalent schematic representation. The operations within a statement are stored in order of precedence. The order of precedence of each operator is given in the VHDL Language Reference Manual. For the example of Figure 10, the precedence of operations would be as follows: XOR, NAND, AND, and OR. In this case, all four operations have equal precedence, so parenthesis are used to determine the order. The parser determines the order of operations, data dependencies (which signals are dependent on previous operations), and creates internal signals to propagate the output of one operation to the input of another. Internal signals are designated by the prefix ***in_*** followed by an integer. For the example of Figure 10, in_1, in_2, and in_3 are the internal signals generated by the system.

The internal data structure representing each of the operations within a statement is shown in Figure 11. The structure contains information about the operation name, operation type, pointers to the previous and next operations (NULL for first and last operations), the input signal list, and the output signal list.

operation_name	(name of operation)
operation_type	(type of operation)
operation_type_ord_value	(integer value assigned to operation type)
next_operation	(pointer to next operation)
previous_operation	(pointer to previous operation)
input_signal_list	(list of input signals to operation)
output_signal_list	(list of output signals of operation)

Figure 11. Internal Representation of an Operation

The internal data structure for an operation allows for unary operations (such as NOT), binary operations (such as AND, OR, XOR, etc.), as well as complex operations which have multiple inputs and outputs. Thus, the same internal format is used to store all operations, irrespective of the number of inputs and outputs. The internal representation of the statement of Figure 10 is shown in Figure 12. The leftmost node (XOR) represents operation of highest precedence in the statement while the rightmost operation (OR) represents operation of lowest precedence.

The statement is stored as a linked list of nodes. The structure of each node is given in Figure 11. For the example of the XOR operation (leftmost node), "XOR" represents the name of the operation, "xors" represents the name of the component in the library that this function is mapped to, "5" is the ordinal value assigned to this operation and is used for sorting. The "next_operation" field points to the "NAND" node since it follows the XOR operation in precedence. The "previous_operation" is NULL since XOR is the first operation in the statement. Signals "c" and "d" represent the inputs to the XOR operation and "in_1" represents the internally generated signal which holds the output of the XOR function.

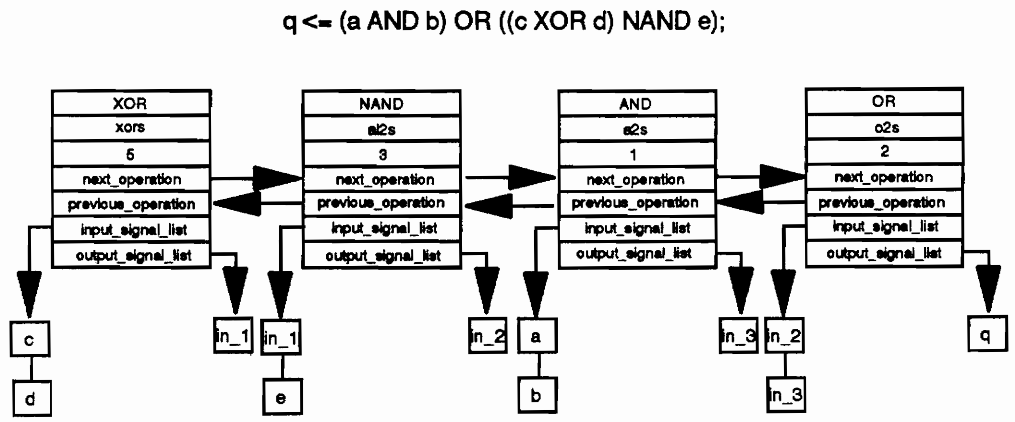


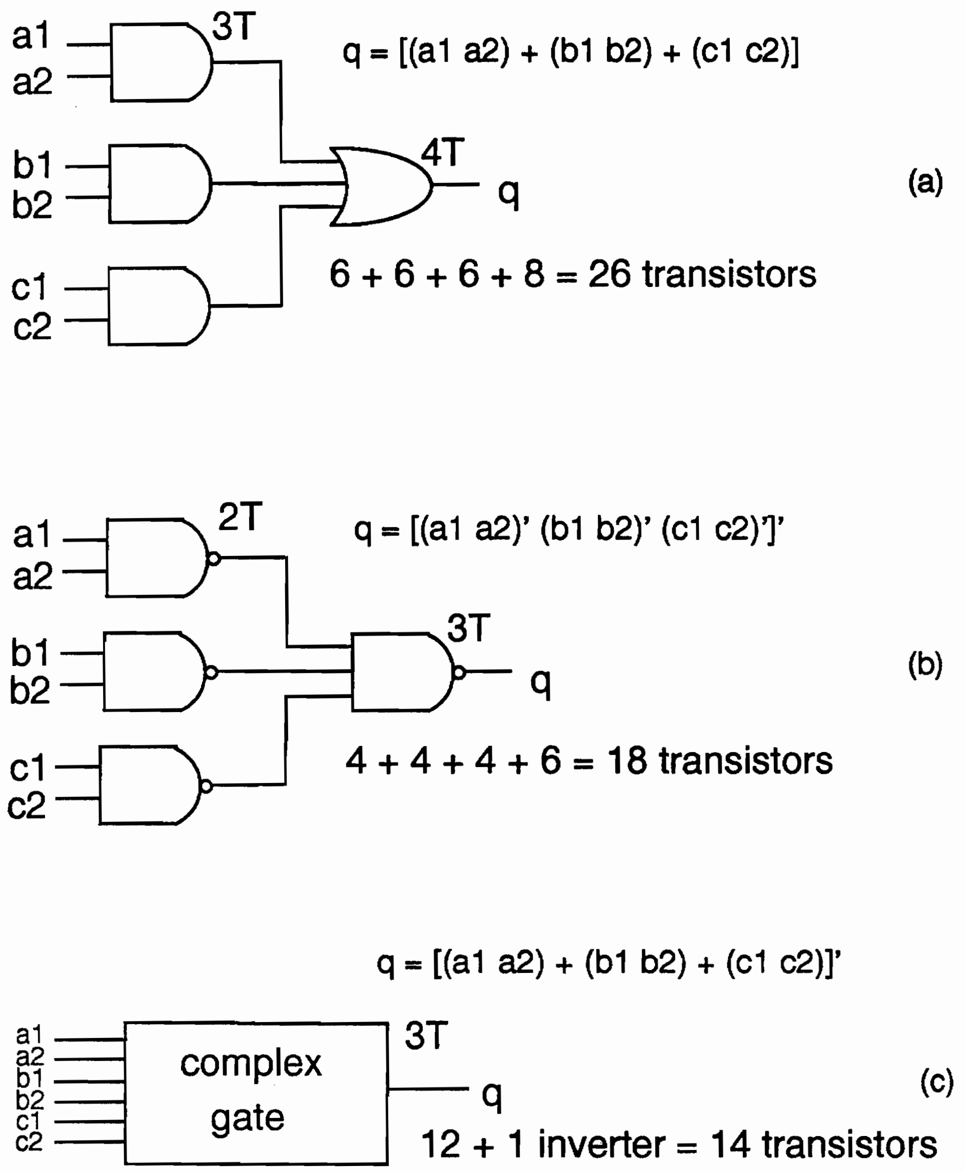
Figure 12. Internal representation of an example dataflow statement

The behavioral operations must be allocated to hardware units. For example, the statement "q <= a AND b;" would be mapped to a 2-input AND gate with inputs a and b, and output q. Several optimizations may be performed at this level. Blocks of logical operations may be mapped to complex gates that implement the same function while reducing delay and area. This optimization is known as ***subexpression elimination***. The optimization can best be described through an example. Consider the following statement:

```
q <= (a1 AND a2) OR (b1 AND b2) OR (c1 AND c2);
```

A direct translation of the statement into AND and OR gates would yield the circuit in Figure 13(a). The CMOS layout of a 2-input AND gate consists of six transistors and results in a three transistor delay from the input to the output. (The term *transistor delay* refers to the time it takes for the signal to propagate through a transistor). The layout of a 3-input OR gate consists of eight transistors and results in a four transistor delay. The resulting circuit for the statement consists of 26 transistors and a worst case delay of seven transistors from input to output. Translating the AND-OR logic into NAND-NAND logic, as shown in Figure 13(b), results in a smaller and faster circuit because the CMOS layout of a 2-input NAND gate consists of only four transistors instead of the six in a 2-input AND gate. The resulting NAND-NAND implementation consists of 18 transistors and a worst case delay of five transistors. A further improvement in size and speed can be made using a complex gate, shown in Figure 13(c), to implement the desired logic function. A comparison of the three resulting circuits is given in Table 7.

```
q <= (a1 AND a2) OR (b1 AND b2) OR (c1 AND c2);
```



a1
a2
b1
b2
c1
c2
3T
4T
q
q = [(a1 a2) + (b1 b2) + (c1 c2)]
6 + 6 + 6 + 8 = 26 transistors
(a)
a1
a2
b1
b2
c1
c2
2T
3T
q
q = [(a1 a2)' (b1 b2)' (c1 c2)']'
4 + 4 + 4 + 6 = 18 transistors
(b)
q = [(a1 a2) + (b1 b2) + (c1 c2)]'
a1
a2
b1
b2
c1
c2
complex gate
3T
q
12 + 1 inverter = 14 transistors
(c)

Figure 13. Optimization for Dataflow Model

Table 7. Comparison of the Circuits of Figure 13

Circuit	Components	Size (# transistors)	Delay (# transistors)
(a)	AND-OR gates	26	7
(b)	NAND-NAND gates	18	5
(c)	Complex gate	14	4

The NAND-NAND implementation results in a 31% reduction in size and a 29% speedup compared to the AND-OR implementation. The complex gate implementation results in a circuit with a 46% reduction in size and a 43% speedup over the AND-OR implementation. This speedup and size reduction occur whenever a part of a statement can be mapped to a an equivalent complex gate in the synthesis library. A complete list of complex gates and their corresponding logic functions is given in Appendix A. If logic for a complex gate is imbedded within a statement, it is automatically extracted during the synthesis process as a part of the dataflow optimizations (described in chapter 5). For example, consider the following statement:

```
q <= ((a1 NOR x) AND a2) OR (b1 AND b2) OR (c1 AND c2);
```

The resulting RNL netlist consists of the complex gate described in Figure 13(c) and a NOR gate as shown below in Figure 14. The output of the NOR gate is an internally generated signal name which is also used as an input to the complex gate.

```
...
(oi2 a1 x in_1)                        /* NOR gate */
(aoi222 in_1 a2 b1 b2 c1 c2 q)         /* Complex gate */
...
```

Figure 14. RNL description of optimized statement

4.2 Synthesis from Algorithmic (High-Level) models

Algorithmic level models are composed of VHDL constructs similar to those of high-level programming languages. Examples of such constructs include if-then-else statements, loop statements, sequential assignment statements, case statements, etc. Obtaining layouts from such descriptions is a very challenging task and is the object of much current research. Even though the internal format does contain data structures to store algorithmic constructs, the VTsynth system is not able to produce physical layouts from most algorithmic level VHDL constructs at the present time. The high-level synthesis algorithms are presented here, but their incorporation into the VTsynth system is reserved for future work.

Currently, the VTsynth synthesis system is able to handle process blocks with a limited number of sequential constructs. These include sequential assignment statements, variable assignment statements, simple if-then-else statement (no nested ifs), and the simple case statement.

No optimizations are currently made on algorithmic level constructs. A simple scheduling scheme is implemented to demonstrate the integration of high-level synthesis into the VTsynth system. The simple scheduling algorithm consists of scheduling one operation per control step. Currently, no attempt is made to extract parallelism out of the sequential VHDL process. The data path synthesis also employs a very simple algorithm. Each operation is bound to a hardware unit. No attempt is made to allow sharing of resources by separate operations.

Figure 15 shows an example of a VHDL process. Assume signals A, B, C, SUM, and DIF to be declared earlier as integers.

```
. . .
p: process (CLK) begin

     if (CLK = '1') then
          SUM <= A + B;
          DIF <= SUM - C;
     end if;
end process p;
. . .
```

Figure 15. Example of a VHDL process

```
p1: add4 port map (state_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3,
                   sum_0, sum_1, sum_2, sum_3);

p2: sub4 port map (state_1, sum_0, sum_1, sum_2, sum_3, b_0, b_1, b_2, b_3,
                   dif_0, dif_1, dif_2, dif_3);
```

Figure 16. Synthesized Data Path for the Process of Figure 15

Figure 16 shows a structural VHDL description of the data path that is generated from the example process if Figure 15. The data path consists of two components, *add4* and *sub4*. The add4 component represents a 4-bit adder and the sub4 component represents a 4-bit subtracter. The first signal in the port map of each component represents the component enable. The controller, which is presently not synthesized automatically, would sequentially enable each component according to the sequence described in the process. In this case, the controller would take as input the process sensitivity list (CLK) as well as the if condition, and generate the enable signals *state_0* and *state_1* accordingly. An RNL netlist is also generated during the synthesis process. If components "add4" and "sub4" were available in the synthesis library, then the generated RNL netlist could be used to obtain a physical layout. A diagram of the data path and controller is given in Figure 17.

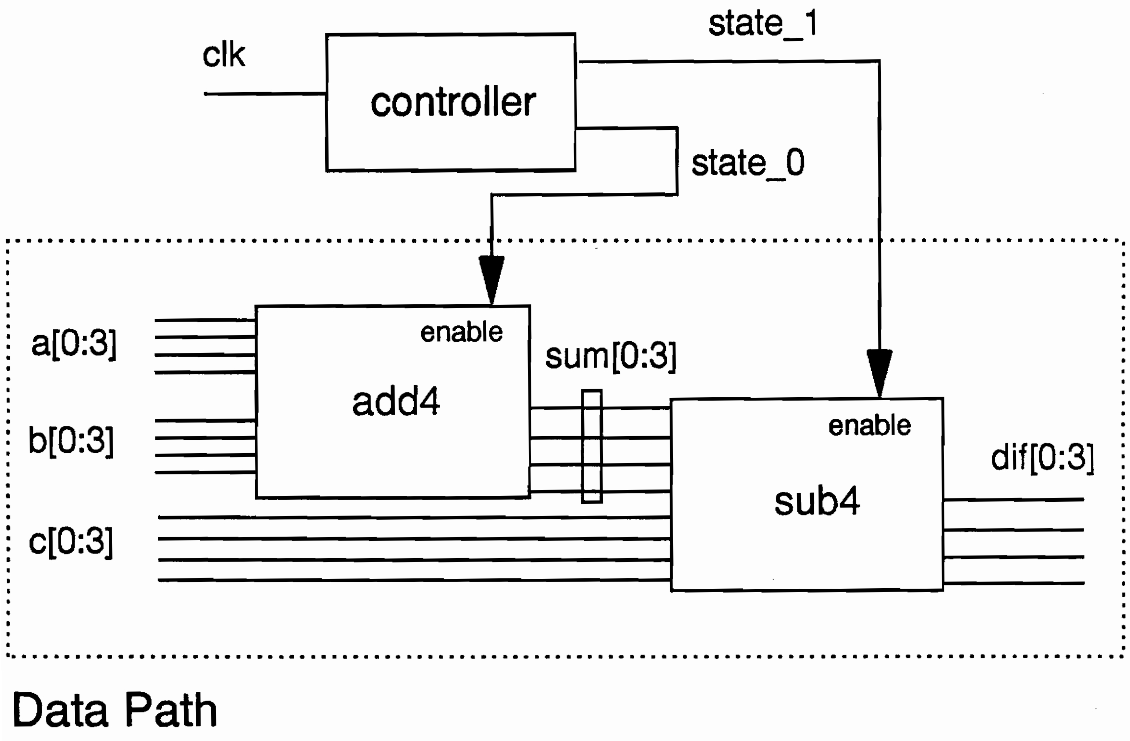


Figure 17. Data Path and Controller for Process of Figure 15

The hardware currently synthesized by the VTsynth system from algorithmic constructs is not optimal by any means. The simple synthesis algorithms are implemented to show that the control flow and data flow within a sequential process can be represented by the internal format. Thus, models containing structural, dataflow, as well as algorithmic models may be mixed during synthesis. The optimizations and algorithms for scheduling and allocation are described in the following sections, but their implementation into the VTsynth system is reserved for future work.

4.2.1 High-Level Optimizations

In order to improve the quality of resulting layouts, several optimizations must be performed to the algorithmic specification. Since the VHDL constructs involved in algorithmic models are similar to those of high-level programming languages, the optimizations are also similar to those of programming language optimizing compilers. These optimizations include dead code elimination, constant propagation, common subexpression elimination, inline expansion of procedures, loop unrolling, etc. [16]. Several books on compiler design discuss these optimizations in detail [17,18].

Several transformations more specific to hardware may also be performed. For example, a multiplication by 1/2 (or division by 2) may be replaced by a right shift operation. the statement, "i := i + 1" may be implemented by an increment operation instead of using an adder with 'i' and '1' as inputs. Detailed descriptions of the types of transformations mentioned here may be found in [4].

4.2.2 Scheduling and Allocation

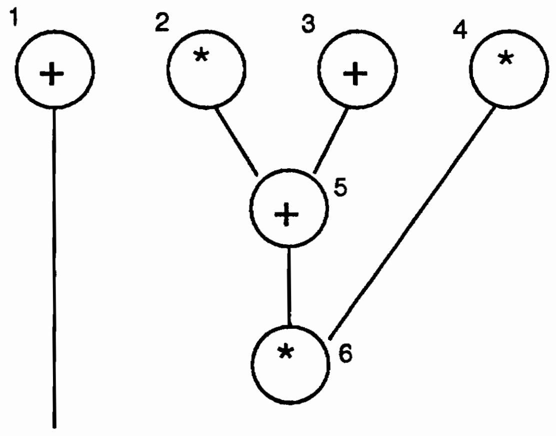
Two very important steps in high-level synthesis are *scheduling* and *allocation.* Scheduling consists of assigning operations to control steps. Allocation consists of assigning operations to hardware. Scheduling and allocation are closely interrelated and interdependent operations. The goal of scheduling algorithms is to minimize the number of control steps needed to implement an algorithm, given limits on the available hardware resources. The goal of allocation is to minimize the amount of hardware needed.

The major problem in the scheduling and allocation tasks is the extremely large number of design possibilities. Finding an optimal solution is a difficult problem. Many synthesis subtasks, including scheduling and allocation even with hardware constraints, are known to be NP-hard [16].

4.2.3 Algorithms for Scheduling

The goal of scheduling algorithms is to find the optimal balance between the time required to complete a set of VHDL algorithm statements and the amount of hardware necessary. For example, if two independent "add" operations needed to be performed, then there are two possible methods of scheduling the operations. The operations may be scheduled concurrently using two separate adders, or they may be scheduled sequentially using the same adder. Several scheduling algorithms have been developed to solve this synthesis problem [16]. Two scheduling algorithms are presented here to demonstrate the role of scheduling in the synthesis process.

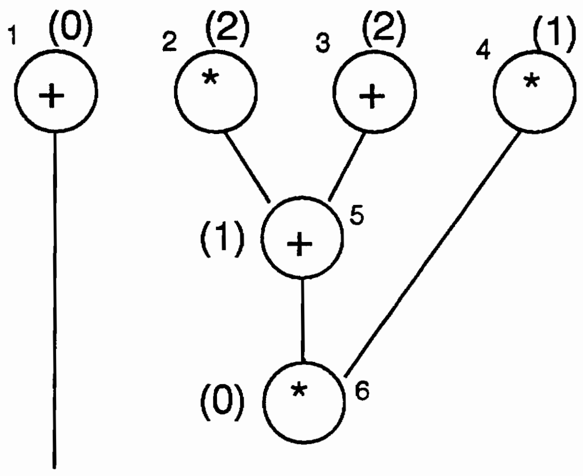
ASAP (As Soon As Possible) *scheduling* is the simplest type of scheduling algorithm. It assumes that the number of functional units has been specified. Operations are sorted topologically according to data dependencies. For example, if operation x2 is constrained to follow operation x1 because of data dependency, then x2 will follow x1 in the topological order. Operations are taken from the list in order and each is put into the earliest control step possible. Figure 18 shows a dataflow graph and its ASAP schedule. The problem with this algorithm is that no priority is given to operations in the critical path (in this case, the path which has the most operations). Operation 1 is scheduled before operations 2 and 3, which are on the critical path. This results in a less than optimal schedule.



Step	*	+
1	2	1
2	4	3
3		5
4	6	

Figure 18. ASAP Scheduling

A *list schedule* uses a more global criterion for selecting the next operation to be scheduled. For each control step to be scheduled, the operations that are available to be scheduled in that control step are ordered by some priority function. Figure 19 shows a list schedule for the dataflow graph of Figure 18. The priority function is the length of the path from the operation to the end of the block, and is represented by the number in the parentheses. Note that the resulting schedule takes one less step to complete when compared to the ASAP schedule.



Step	*	+
1	2	3
2	4	5
3	6	1

Figure 19. A List Schedule

4.2.4 Algorithms for Allocation

Allocation consists of mapping operations onto functional units, assigning values to registers, and providing interconnections between operators and registers using buses and multiplexers. The optimization goal is to minimize some objective function, such as total interconnect length, total hardware cost, or critical path delays. Allocation algorithms may be classified into two types: *iterative*, and *global.*

Iterative algorithms select an operation, value, or an interconnection to be assigned, makes the assignment, and then repeats the process for the remaining items to be assigned. Selection rules determine the next operation, value, or interconnect to be selected. *Global allocation* techniques combine graph theory and mathematical programming. In these graphs, elements are assigned to hardware and represented as nodes. The elements may be operations, values, or interconnections. Arcs between two nodes signify that the two nodes share the same hardware. The problem then becomes a graph theory problem of finding a minimal number of cliques that cover the graph. (A *clique* is a set of nodes in a graph in which all of the nodes are connected to each other). The objective is to find a valid solution that minimizes some cost function. Finding an optimal solution requires an exhaustive search. Since this can be very computationally expensive, heuristics may be used to reduce the search space.

4.3 Synthesis of an Example

Synthesis from structural, dataflow, and algorithmic models can best be understood through an example. The example used for synthesis is the model of a ones_counter found in [9]. Architectural descriptions for structural, dataflow, and algorithmic styles are shown and the synthesis results compared. The interface description for the ones_counter is given below in Figure 20.

```
entity ones_counter is
        port (A : in bit_vector (0 to 2);
              C : out bit_vector (0 to 1));
end ones_counter;
```

Figure 20. Interface description of ones_counter

Three architectural bodies for the ones_counter are presented. Figure 21 shows a structural description of the ones_counter. Figure 22 shows a data flow implementation while Figure 23 shows an algorithmic description of the model. All three models are functionally equivalent, i.e., they produce the same output for a given input.

```
architecture structural of ones_counter
        component MAJ3 is
                port (x : in bit_vector (0 to 2);
                        z : out bit);
        end component;

        component OPAR3 is
                port (x : in bit_vector (0 to 2);
                        z : out bit);
        end component;

begin
        comp_1: MAJ3 port map (A, C(1));
        comp_2: OPAR3 port map (A, C(0));
end structural;
```

Figure 21. Structural VHDL model of a ones_counter

```
architecture dataflow of ones_counter is
begin
        C(1) <= (A(1) and A(0)) or (A(2) and A(0)) or
                        (A(2) and A(1))

        C(0) <= (A(2) and not A(1) and not A(0))
                or (not (A(2) and not A(1) and A(0))
                or ((A(2) and A(1) and A(0))
                or (not (A(2) and A(1) and not A(0))
end dataflow;
```

Figure 22. Dataflow VHDL model of ones_counter

```
architecture algorithmic of ones_cnt is
begin
process (A)
        variable num: integer range 0 to 3;
begin
        num := 0;
        for i in 0 to 2 loop
          if A(i) = '1' then
            num := num + 1;
          end if;
        end loop;

        case num is
          when 0 => C <= "00";
          when 1 => C <= "10";
          when 2 => C <= "01";
          when 3 => C <= "11";
        end case;
end process;
```

Figure 23. Algorithmic VHDL model of a ones_counter

Synthesis from the structural model:

The structural model consists of two components, MAJ3 and OPAR3, as shown in Figure 24. The components MAJ3 and OPAR3 may be previously synthesized models or models generated by hand. If these components are the results of previous synthesis, then the original models must have been written using constructs allowed by VTsynth. The structural model would consist of component instantiation statements (AND and OR gates for MAJ3, and XOR gates for OPAR3). The dataflow model would consist of logical operations and assignment statements. RNL descriptions of MAJ3 and OPAR3 may also be generated by hand. This is done by creating an RNL netlist utilizing components in the VPNR library. A manually generated RNL netlist is shown in Figure 25. Two views of MAJ3 are presented, MAJ3_1 and MAJ3. Since the function for MAJ3 is implemented using a complex gate and an inverter, it is the preferred configuration because it is faster and smaller than the one utilizing three 2-input AND gates and a 3-input OR gate. When the optimal description of MAJ3 is used, then the resulting layout of the structural description of the ones_counter consists of 42 transistors with a 10 transistor delay.

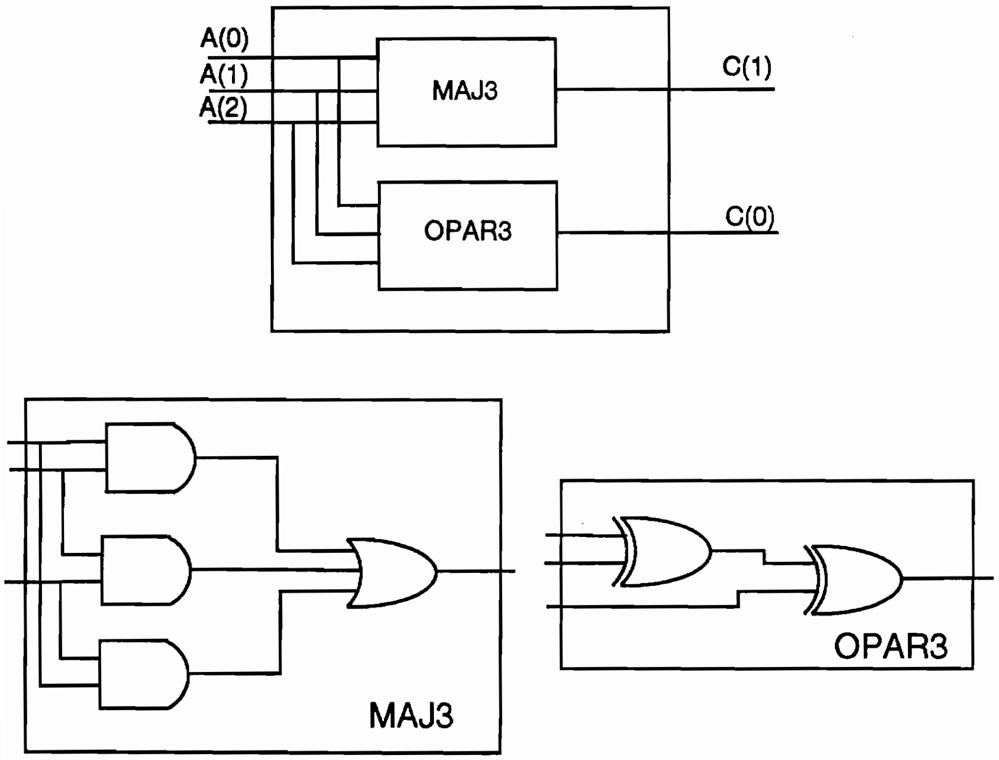


Figure 24. Structural view of ones_counter

```
(macro MAJ3_1 (x0 x1 x2 q)
(local in_1 in_2 in_3)
(a2 x0 x1 in_1)
(a2 x1 x2 in_2)
(a2 x0 x2 in_3)
(o3 in_1 in_2 in_33 q)
)
```

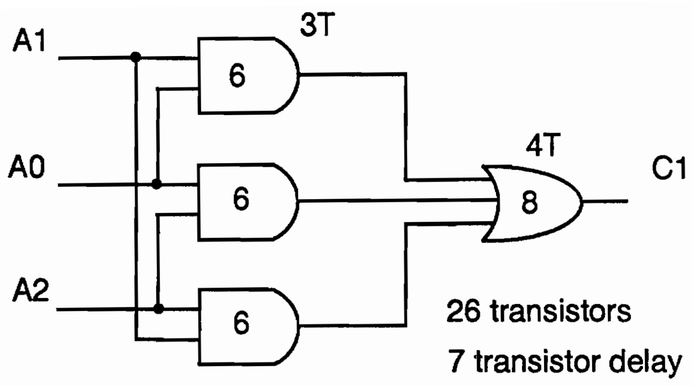
```
(macro MAJ3 (x0 x1 x2 q)
(local in_1)
(aoi222 x0 x1 x2 in_1)
(i1 in_1 q)
)
```

```
(macro OPAR3 (a0 a1 a2 q)
(local i1)
(xor a0 a1 i1)
(xor i1 a2 q)
)
```

Figure 25. A manually generated RNL netlist for MAJ3 and OPAR3

Synthesis from the dataflow model:

The dataflow description for the ones_counter was presented in Figure 22. A schematic diagram of the circuit is shown in Figure 26. A direct translation of the description into AND and OR gates yields a circuit which consists of 74 transistors with a total delay of 10 transistors. The circuit may be optimized by using a complex gate to implement the C(1) function and using XOR gates to implement the C(0) function.



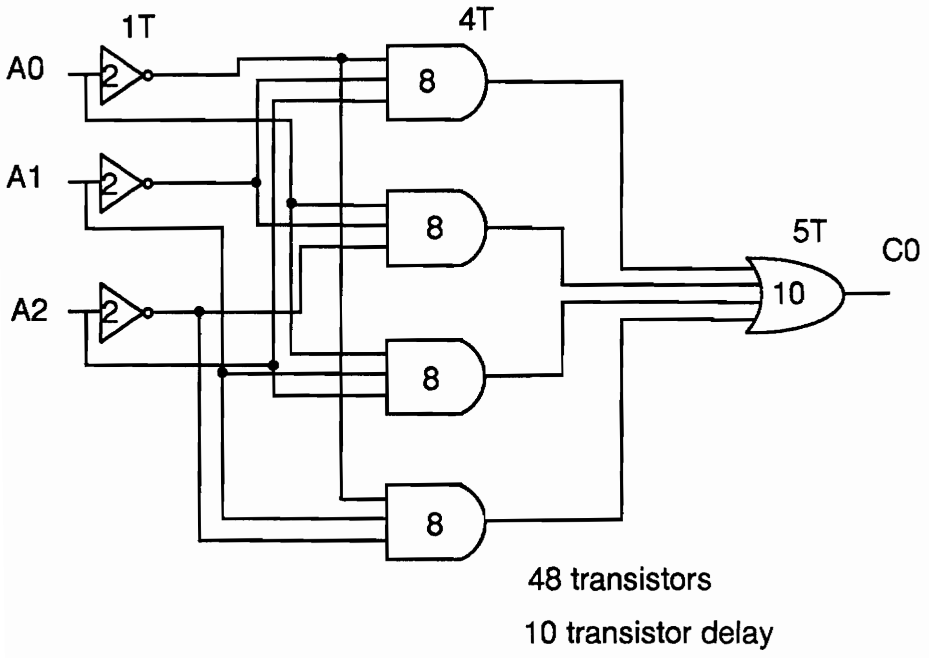


Figure 26. Logic schematic of ones_counter

Circuit	Unoptimized		Optimized	
	Size	Delay	Size	Delay
C1	26	7	14	4
C0	48	10	28	10

Figure 27. Comparison of optimized and unoptimized ones_counter

Figure 27 shows the results of the optimized circuit versus the unoptimized circuit. Either circuit may result from the structural or dataflow models. If the cells of the structural model are not optimal, then the resulting layout will not be optimal. If the synthesis system does not perform optimizations at the dataflow level, then the results will also not be optimal. In order to obtain efficient layouts from a synthesis system, the following two things must be true. First, the layouts for the standard cells must be optimal. Second, the synthesis system must have a knowledge of what components are available and a method of mapping behavioral statements into optimal hardware.

Synthesis from the algorithmic model:

The algorithmic model of the ones_counter consists of higher level modeling constructs such as a FOR loop, IF statement, and CASE statement. In order to obtain a structure from this model, each of the modeling constructs must be mapped to hardware. Figure 28 shows a possible hardware configuration for this algorithm. A counter and comparator are used to implement the FOR loop, a multiplexer is used to implement the IF statement, and two multiplexers are used to implement the CASE statement. Signals A and C, and variable NUM are mapped to registers. An adder is used to implement the add function inside the loop.

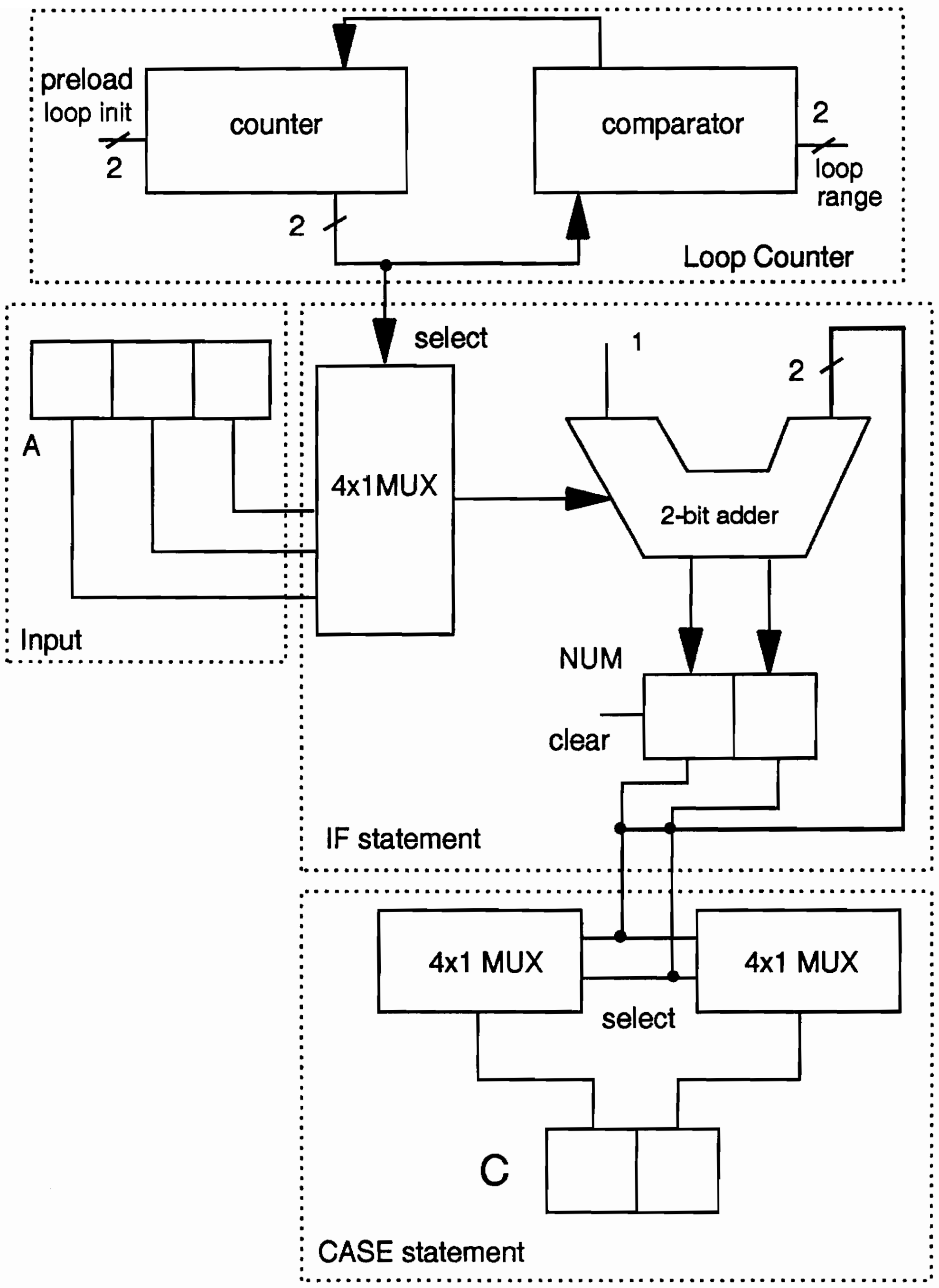


Figure 28. Structure obtained from Algorithmic view of ones_counter

Note that since no structure was implied or specified by the algorithm, the modeling constructs were mapped to general purpose components. The loop control was mapped to a counter to increment the loop variable and a comparator to determine when the loop ends. The IF statement was mapped to a multiplexer and the add function was allocated to an adder. The CASE statement was mapped to two multiplexers since the output is two bits wide. The 4x1 MUXs were used for the CASE statement since there were four alternatives for the CASE expression.

It can be easily seen that this implementation requires much more hardware to implement than the structural and dataflow models. The counter, comparator, multiplexors, etc. may have been unnecessary for this circuit if more information about the structure was specified.

Note that the CASE statement is functioning as a type conversion routine. It is assigning the integer value of variable 'num' to the bit_vector C. In the synthesis algorithm, integer variables get mapped to bit_vectors. So the output of the synthesized variable 'num' can be tied directly to the bit_vector C. Thus, two multiplexers would be saved.

One method of reducing the generation of some unnecessary hardware is to define a set of functions that the synthesis system recognizes. These can include type conversion functions among other things. The CASE statement could then be modeled as a function which is recognized by the synthesizer as with the function:

```
C <= int_to_bitvec (num);
```

Function intval() (used to convert a bit_vector to an integer) could also be included in this set of functions. This would allow easy type conversion between integers and bit_vectors. During modeling and simulation, type conversions must be performed when operating on different data types. During synthesis, all data types get mapped to bits and bit_vectors. Therefore, no type conversions are necessary.

This example has shown the process by which hardware may be obtained from algorithmic descriptions. It was seen that the amount of hardware generated from the algorithmic description was much greater than that generated from the dataflow or structural models. Using predefined functions that are recognized by the synthesizer is one way to increase the efficiency of the generated designs. Automating the algorithmic optimizations described in section 4.2.1 is another way of eliminating unnecessary hardware from a design. In general, the more detail that is specified about the nature of the circuit, the more optimal will be the synthesis.

Chapter 5
Design of the VHDL Synthesis System

The VTsynth synthesis system consists of a set of programs used to obtain a physical layout from a VHDL description. The input to the VHDL synthesis system is a text file containing statements from the subset of VHDL described in Chapter 4. The program that reads the VHDL text is called a *parser*. The parser reads the text and stores the information in data structures in memory. A description of the parser and the format of the internal data structures are presented in this chapter. Information contained in the synthesis library, algorithms to process the statements, optimizations, and a description of the program output are also topics covered in this chapter. The chapter concludes with a description of the procedure used to obtain the final physical layout using VPNR [19] and Magic [20].

5.1 VHDL Parser

The VHDL parser is generated using the parser generator ***bison*** [21]. Bison is a program very similar to the Unix program Yacc (yet another compiler compiler). The BNF description of VHDL was adapted from the University of Pittsburgh's vsim VHDL simulator. This section describes how Bison was used to generate the parser and store the intermediate format.

Bison is a parser generator or a program to convert a grammatical specification of a language into a parser that will parse statements in the language. Bison provides a way to associate meanings with the components of the grammar in such a way that as the parsing takes place, the meaning can be evaluated as well. First, a grammar is written which specifies the syntax of the language. This grammar can be directly obtained from Appendix A of the IEEE Standard VHDL LRM which defines the entire syntax of VHDL in BNF format. Second, each production rule of the grammar can be augmented with an action - the series of actions to perform when an instance of that grammatical form is found in the program being parsed. In this case, the actions will store the VHDL as an intermediate form. The Bison compiler was ported from an Ultrix Vax to an MS-DOS-based 80386 PC. The source code for the Bison compiler was compiled using the Microsoft C compiler.

The program ***Lex*** [22] generates a lexical analyzer in a manner similar to the way Yacc and Bison generate parsers. A *lexical analyzer* reads the input being parsed and breaks it up into meaningful chunks for the parser. This lexical chunk is referred to as a *token.* A token can be a VHDL keyword, a number, a name (such as an entity or architecture name), an identifier, an operator, etc.

Bison reads in the BNF format of VHDL as input and outputs a file containing the parser. This parser is a function with the name yyparse(). The function yyparse() repeatedly calls the lexical analyzer for tokens, recognizes the syntactic structure in the input, and performs the semantic actions as each grammatical rule is recognized. The main VTsynth program calls function yyparse() until the end of file is reached. The semantic actions that are performed when a grammatical rule is recognized are written in C. The actions are in the form of functions that are called when a rule is recognized. These functions reside in C source files and are compiled and linked with the Bison generated parser and lexical analyzer.

In order to validate the parser generated by Bison, another parser was generated which performed no actions when grammatical rules were recognized, except to report syntax errors. The parser was then used to check syntax on models which were known to be correct. The test cases were previously simulated versions of models from [9] consisting of the MARK2, RAM, UART, INT_CONT and I8212 models. All the models passed the syntax check. When syntax errors were purposely inserted in the models, the parser correctly flagged the errors. Although all the VHDL constructs were not formally tested in the validation process, the models that were used to test the parser did exercise most of the VHDL modeling constructs.

5.2 Specification of the Internal Form

The internal format of the VTsynth system is a graph based format which allows for representation of multiple levels of design. The same format is used to represent structural, dataflow, as well as algorithmic modeling constructs. Thus, items such as component declarations, component instantiations, assignment statements, if-then-else statements, case statements, loop statements, etc. may all be represented within the framework of this format. Figure 30 shows a high-level view of the internal representation. The root node represents the entity structure. The corresponding C code for the entity structure is shown below in Figure 29. This data structure contains a pointer to the port list (I/O signals), a pointer to the first architecture that describes this entity, and a pointer to any subsequent entities that may be declared. Several architecture descriptions may be given for the same entity. Descriptions of the architecture structure and the internal representation of statements inside an architecture body are described in the following sections.

```
struct ENTITY_STRUC
{
  char        ent_name[MAX_NAME_LEN];  /* the entity name */
  port_ptr    ent_port;                /* entity port list   */
  ent_ptr     ent_next;                /* pointer to the next top level entity  */
  arch_ptr    ent_arch_ptr;            /* pointer to architectures for this entity  */
};
```

Figure 29. Data Structure for the Entity Declaration

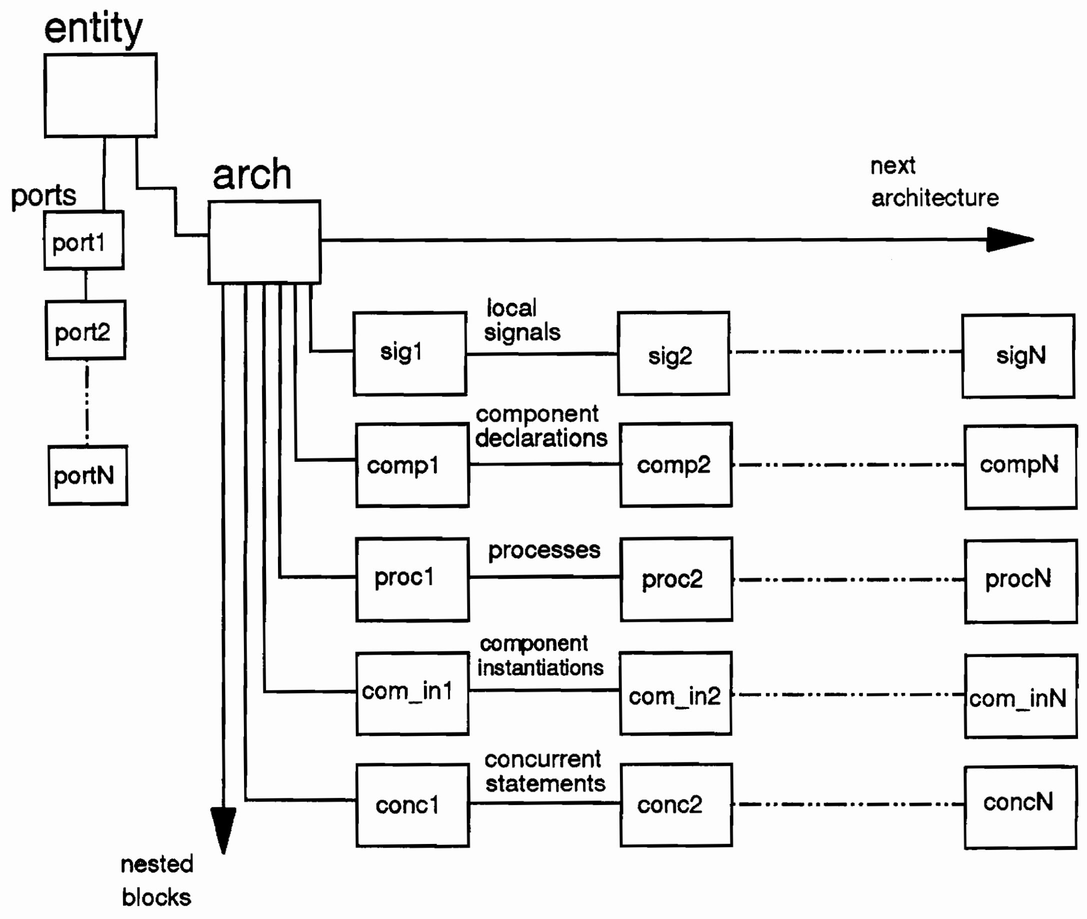


Figure 30. Diagram of the VTsynth internal format

5.2.1 Architecture Structure

The architecture body of a VHDL description contains local signals, component declarations, component instantiation statements, concurrent assignment statements, and processes. The internal format is able to represent all of these declarations and statements. Figure 30 (of the previous section) shows the organization of the information contained in the architecture. Each type of statement is stored as a linked list of nodes, with each node representing a complex data structure. For example, all local signals are stored in a linked list, all component declarations are stored in a linked list, all processes are stored in a linked list, etc. Thus, the synthesis algorithm needs only to traverse a list when performing synthesis on constructs of a certain type. The C code to represent the architecture declaration is shown in Figure 31.

```
struct ARCHITEC_STRUC
{
  char        arch_name[MAX_NAME_LEN];  /* the architecture name  */
  int         arch_flag;                /* flag word  */
  arch_ptr    arch_next;                /* next architecture for this entity  */
  ent_ptr     arch_ent_back;            /* a 'back-pointer' to the entity  */
  struct BLOCK_STRUC arch_block;        /* architecture block */
};
```

Figure 31. Data Structure for Architecture Declaration

The body of the architecture, which is referred to as *arch_block* in the above architecture structure definition (line 7 of Figure 31), is also a complex data structure called *BLOCK_STRUC*. Since the architecture body is essentially a top level block, the data structure for the architecture body is the same as the structure used to represent all concurrent blocks and process blocks which may exist in the architecture body. The C code for this data structure, called BLOCK_STRUC, is shown below in Figure 32.

```
struct BLOCK_STRUC
{
  int          block_type;           /* type of block, either process or conc. */
  label_ptr    block_lab_ptr;        /* pointer to this block's unique label */
  label_ptr    block_labels;         /* local block labels */
  port_ptr     block_signals;        /* local block signals */
  comp_temp_ptr  block_comp_temp;      /* component template pointer */
  block_ptr    block_next;           /* pointer to next block at this level */
  block_ptr    block_sub;            /* pointer to subordinate blocks */
  block_ptr    block_last;           /* pointer to block up in the hierarchy */
  block_ptr    block_proc;           /* pointer to processes within this block */
  state_ptr    block_state;          /* pointer to statements within this block */
  comp_in_ptr  block_comp_in;        /* pointer to component instans. */
  arch_ptr     block_arch_back;      /* back pointer to the parent architecture */
  sig_list_ptr block_sens;           /* sensitivity list for process */
  sig_list_ptr block_int_sigs;       /* internally used signal list for process */
};
```

Figure 32. Data Structure for Blocks and Processes

5.2.2 Signal Structure

A single data structure, shown in Figure 33, is used to represent signals and variables. Signals may be declared in entity port declarations (I/O signals) or in the architecture body declarations (local signals). Variables may only be declared within a process (as defined by the VHDL LRM). The data structure is presently able to handle any signal or variable of type bit, boolean, bit_vector, or integer. An integer is mapped to a bit_vector when translating to a hardware structure. If a range constraint is specified with an integer, then the integer is mapped to a bit_vector with the width specified by the range constraint. For example, the statement "variable a: integer range 0 to 200;" would be mapped to bit_vector (0 to 7). If a range constraint is not specified with an integer, then the integer is mapped to a bit_vector with a default width. The default width is a constant in the file "data_strc.h" and may be changed by the user.

```
struct SIGNAL_STRUC
{
  char        sig_name[MAX_NAME_LEN];  /* signal name */
  u_char      sig_type;          /* signal type (predefined type definition) */
  u_char      sig_mode;          /* in, out, inout, buffer, linkage, local */
  sig_ptr     sig_map;           /* pointer to real signal in port map */
  int         sig_bv_hwm;        /* bit width for bit vector */
  union
  {
    block_ptr sig_block;         /* pointer back to block where signal is def. */
    ent_ptr   sig_ent;           /* pointer to entity for port def. */
  } sig_back;
};
```

Figure 33. Data Structure for Signals and Variables

5.2.3 Process Structure

As stated in Section 5.2.1, the data structure for a process is the same as the one for concurrent blocks and the architecture body. Actually, the format of the data structure is the same but the information represented is different. The process structure stores information such as the process label, process sensitivity list (versus guarded signal list in the block structure), local variables (as opposed to local signals), sequential statements (as opposed to concurrent statements), as well as pointers to other processes and the architecture and entity in which the process is defined. A graphical view of the process data structure is given in Figure 34.

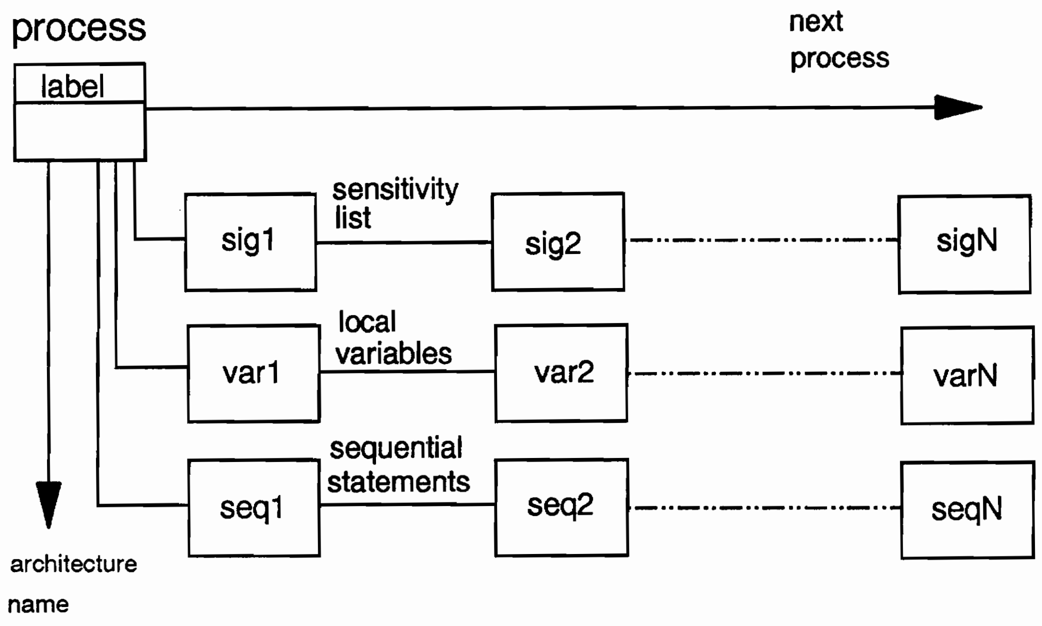


Figure 34. Internal Structure of a Process

5.3 Design of the Synthesis Library

The synthesis library contains a list of all the components available for synthesis and the characteristics such as the size and speed of each component. The library is implemented as a C include file. This allows all the component information to be local to the synthesis program. Thus, the program does not have to read a database at run time since the library data is a part of the compiled executable program. The performance of the program is optimal since all the data is local. One disadvantage to this approach is that the program must be recompiled if the design library is modified. The recompliation is easily accomplished with one command using the *make* file in the VTsynth directory. The entire recompilation takes less than five minutes on the system described in Chapter 1.

The library consists of a list of C "#define" directives. The gate name, type, size, and delay value are provided for each component in the library. The delay value corresponds to the worst case delay of the component driving two inverter loads. The size of the cell is based on a layout of the cell using MOSIS 3 micron technology. Figure 35 shows the information for two gates, a 2-input AND gate, and a complex gate. The logical function implemented by a gate may be determined by the gate name. For example, gate "a2" is a 2-input AND gate. Similarly, gate "aoi31" is a complex gate implementing AND-OR-INVERT logic. The AND gate has three inputs. The OR function gets its input from the AND gate and an external input. All the components in the present VPNR library are listed in Appendix A along with their size and speed characteristics. Currently, only the component name and type are used in the synthesis process. The values for the sized and speed of the component may be used in the future to calculate the size and speed of the synthesized circuit.

```
   #define GATE_AND                  1        /* gate type */
   #define GATE_AND_STR              "a2"     /* name in VPNR cell lib */
   #define GATE_AND_DELAY            3557     /* delay in ps -> 3.557 ns */
   #define GATE_AND_SIZE             4176     /* size in square microns */
...
...
   #define COMPLEX_AOI222            15       /* gate type */
   #define COMPLEX_AOI222_STR        "aio222" /* name in VPNR cell lib */
   #define COMPLEX_AOI222_DELAY      6052     /* delay in ps -> 6.052 ns */
   #define COMPLEX_AOI222_SIZE       8352     /* size in square microns */
...
```

Figure 35. Format of Component Information in Synthesis Library

5.4 The Synthesis Process

A description of the parser was given in Section 5.1. It was stated that the parser stored valid VHDL constructs in the internal format described in Section 5.2. The synthesis process consists of traversing the graph form of the VHDL constructs and producing a hardware structure corresponding to the operations encountered. The methods used by VTsynth for generating the two output files (structural VHDL and RNL netlist) differ from each other. The structural VHDL file consists of component declarations as well as component instantiations and interconnections. The RNL netlist only consists of component instantiations and interconnections. Therefore, the graph representing the internal format must be traversed twice in order to generate the structural VHDL but only once to generate the RNL. In the first pass, VTsynth determines which components are used in the synthesized design. The VHDL component declarations are then output to the structural VHDL file. The second pass is used to output the component instantiation and interconnection information to the structural VHDL and RNL files. Pseudocode for the synthesis process is shown below.

```
program VHDL_Synthesis

procedure parse_input begin
        if (valid_input) then
                optimize_if_possible;
                store_in_internal_format;
        else
                report ("Statement not allowed in VHDL synthesis subset");
        end if;
end procedure parse_input;

begin (* main *)

        parse_input;

        output_entity_declaration_to_structural_VHDL;
        output_entity_declaration_to_RNL;

        output_architecture_declaration_to_structural_VHDL;

        output_local_signals_to_structural_VHDL;
        output_local_signals_to_RNL;

        output_internally_generated_signals_to_structural_VHDL;
        output_internally_generated_signals_to_RNL;


(* pass 1 *)

        for (all_component_declaration_statements) begin
                output_component_declaration_to_structural_VHDL;
        end for;


        for (all_concurrent_statements) begin
                for (all_operations_in_statement) begin
                        bind_operation_to_component;
                        if (component_not_declared) then
                                output_component_declaration_to_structural_VHDL;
                        end if;
                end for;
        end for;


        for (all_processes) begin
                for (all_sequential_process_statements) begin
                        for (all_operations_in_statement) begin
                                bind_operation_to_component;
                                if (component_not_declared) then
                                        output_component_declaration_to_structural_VHDL;
                                end if;
                        end for;
                end for;
        end for;

(* pass 2 *)


        for (all_component_instantiation_statements) begin
                output_component_instantiation_to_structural_VHDL;
                output_component_instantiation_to_RNL;
        end for;


        for (all_concurrent_statements) begin
                for (all_operations_in_statement) begin
                        output_component_instantiation_to_structural_VHDL;
                        output_component_instantiation_to_RNL;
        end for;
    end for;

    for (all_processes) begin
        state := 0;
        for (all_sequential_process_statements) begin
            for (all_operations_in_statement) begin
                output_component_instantiation_to_structural_VHDL;
                output_component_instantiation_to_RNL;
                state := state + 1;
            end for;
        end for;
    end for;
```

5.5 Description of the Synthesis Output

The output of the VTsynth is in the form of two files. One is a purely structural VHDL file and the other is an RNL format netlist. Both these files represent purely structural netlists. The RNL netlist may be sent to VPNR in order to produce the physical layout. The structural VHDL may be simulated to verify the synthesis process. The structural VHDL file contains a port declaration corresponding to the port declaration of the unsynthesized model, declarations of all the components used, and component instantiations with the appropriate port maps containing the interconnect information. The port declarations of the pre-synthesis and post-synthesis models will differ if the original model contained any data types other than bit. An example of this translation is given below. Figure 36 shows the pre-synthesis port list while Figure 37 shows the port list after synthesis.

```
port (a,b : in bit;
      q   : out bit;
      d, e, f : in integer;
      x, y :    out integer);
```

Figure 36. Unsynthesized Port List

```
port (
      a : in bit;
      b : in bit;
      d_0 : in bit;
      d_1 : in bit;
      d_2 : in bit;
      d_3 : in bit;
      e_0 : in bit;
      e_1 : in bit;
      e_2 : in bit;
      e_3 : in bit;
      f_0 : in bit;
      f_1 : in bit;
      f_2 : in bit;
      f_3 : in bit;
      q : out bit;
      x_0 : out bit;
      x_1 : out bit;
      x_2 : out bit;
      x_3 : out bit;
      y_0 : out bit;
      y_1 : out bit;
      y_2 : out bit;
      y_3 : out bit);
```

Figure 37. Synthesized Port List

There are several transformations that were made to the original port list. First, all bit_vectors were expanded into individual bits. The name of the bit corresponds closely to the original name in the bit_vector. For example, bit x(0) in a bit_vector x maps to bit x_0. Second, the mode of each signal is preserved. If a signal was originally of mode *inout*, then the signal will be of mode *inout* in the synthesized design. Finally, the ordering of the signals in the port list is rearranged. All of the input signals are listed before the output signals. This is a convention used by VPNR and is thus incorporated in the synthesis process.

The remainder of the output consists of component instantiation statements. Structural components of the pre-synthesis VHDL model are translated directly into component instantiation statements in the resulting VHDL and RNL netlists. Behavioral constructs are synthesized into structural components and output in VHDL and RNL formats. It is the user's responsibility to ensure that components used in structural VHDL models either exist in the VPNR library or are the output of previously synthesized models. If a user instantiates components that are non-existent, then VPNR will not be able to generate the physical layout.

5.6 VPNR and Magic

VPNR (Vanilla Place aNd Route System) is a CAD tool developed by MCNC (Microelectronics Center of North Carolina) which automatically translates netlist descriptions of logic circuits into physical layouts using predesigned standard cells.

VPNR is a very powerful automatic layout tool. It comes with a database of standard cells which were designed using the VLSI editor "Magic". Currently, only the VPNR cell library may be used as input to the VPNR place and route package. Information such as the shape, connection points, etc. of the standard cells is in the data base. The VPNR toolset also contains a tool which reads a "Magic" description of a standard cell and generates a database entry for that cell. Thus, a user-designed cell is easily added to the VPNR database provided the designer strictly adheres to the design style used in the original VPNR database. The user may also write macros containing components from the VPNR database. These macros can be added to the database resulting in a hierarchical database of components. The macros will be expanded automatically by the netlist translator when encountered in an RNL description.

The VPNR software contains two netlist translators, one to translate a HILO netlist into VPNR and the other to translate an RNL netlist into VPNR. The RNL netlist translator is used to translate the RNL netlist generated by VTsynth to VPNR format. This same netlist may also be used for switch level simulation, test generation, etc. The VPNR automated place and route system consists of three basic subprograms that perform:

- combined placement and loose global routing of cells,
- detailed global routing of the signal nets,
- detailed channel routing.

The combined placement and global routing subprogram receives a netlist of standard cells, places the cells in rows, and makes a tree of connections for each electrical net (global routing). The detailed global routing subprogram then performs iterative improvements on the design and prepares the data for detailed channel routing. The router computes where to put the wires so that the resulting layout occupies the least amount of area. VPNR provides a choice of two channel routers [19]: a greedy channel router, and a left-edge based router with channel compaction. The output of the VPNR is a "Magic" layout file. Magic may be used to graphically view the layout or it may be used to generate the CIF description of the layout without actually invoking the Magic graphics display.

The entire place and route process is executed with one command called "vpnr" with the RNL netlist filename as a command line argument. For example, if the input VHDL file to VTsynth was *adder.vhd*, then the output of VTsynth would be *adder.str* (structural VHDL) and *adder.net* (RNL netlist). The VPNR command line would be, "vpnr adder.net". VPNR will create a directory named *adder.magic* containing the layout information.

Chapter 6
Synthesis Results

The VTsynth system has been used to generate layouts from several VHDL models. Results of synthesis from several examples are presented in this chapter. Included in this section are the input VHDL, the resulting RNL and structural VHDL netlist files, and the VPNR generated layouts for several examples.

Table 8. VTsynth Results

	# Lines VHDL	# Operations	CPU Time	Memory	# components	# transistors
STR	70	5	2 sec.	644	5	28
DFL	25	25	3	2.6 K	19	130
ALG	20	5	3	800	5	N/A
ADR	50	49	4	5 K	29	276
MED	132	196	6	18 K	116	1104
LRG	338	530	12	35 K	318	3040
VLG	1000	1600	52	55 K	1000	12000

Table 8 gives a tabulated summary of several examples synthesized by VTsynth. The table contains information about the model such as the model name (e.g., STR, DFL, ALG, etc.), the number of lines in the input VHDL description, the number of operations in the model (e.g., component instantiations, logical operations such as NAND, NOR, etc.), the CPU time to generate the RNL and structural VHDL netlists, the memory required the synthesize the model, the number of components in the synthesized design, and the number of transistors in the resulting design.

The model named STR is a purely structural VHDL description. It contains five component instantiation statements. Since this is a purely structural model, VTsynth translates it directly to an RNL description. The structural VHDL model output by VTsynth turns out to be the same as the input model. The model named DFL is a dataflow description containing 25 logical operations such as AND, OR, XOR, etc. VTsynth determined that 19 components were needed to synthesize the dataflow description. Six complex gates were used in the design. An unoptimized design would have used 25 components to perform the 25 logical operations found in the input description.

The model named ALG contains algorithmic VHDL constructs which include a process statement and sequential assignment statements. VTsynth generated a netlist of components containing adders and subtracters when synthesizing this model. The controller to schedule the enabling of the components is not yet synthesized automatically. Since the layout of the adder and subtracter are not available in the VPNR library, no physical layouts can presently be generated for the algorithmic models.

The ADR model is a register-transfer level description of a 4-bit carry-lookahead adder. It was obtained from [26] and translated into VHDL. The input VHDL description contained 49 logical operations such as AND, OR, etc. VTsynth was able to synthesize this model in four seconds using 5K bytes of memory space. The resulting netlist contained 29 components. Thus, 49 operations were mapped to 29 components using the optimizations described in Chapter 4. The RNL netlist was sent to VPNR for placement and routing. VPNR was successful in generating a layout for the netlist in about two minutes. Using the standard cell library described in Appendix A, the layout of unoptimized adder was 30.56 square mils in size. The layout of the optimized adder was only 16.41 square mils resulting in a design which is almost half the size of the unoptimized adder.

The MED, LRG, and VLG models are larger versions of the 4-bit carry-lookahead adder of the ADR model. MED is essentially four of the 4-bit adders, LRG is ten 4-bit adders, and VLG contains 35 of the 4-bit adders. These models were synthesized to see the effect of large models on performance and memory requirements for VTsynth. It was observed that there was a constant overhead for CPU and memory for each execution of VTsynth independent of model size. After the initial overhead, the CPU time and memory requirements were fairly linearly proportional to the input file size.

The remainder of this chapter shows actual results from the VTsynth system. Example 1 shows synthesis from a model containing structural as well as dataflow VHDL constructs. First, the input VHDL file is given. The two pages that follow are the output produced by VTsynth for the given VHDL model. These are the RNL and structural VHDL generated by VTsynth. The final page of Example 1 is Figure 38, the layout generated by VPNR for this example.

Example 2 shows how VTsynth currently handles algorithmic constructs. The input VHDL file, ALG from Table 8, contains a process statement and sequential assignment statements. The output generated by VTsynth for this model follows the input VHDL description. Note that the add4 component was used to implement the '+' operation and the sub4 component was used to implement the '-' operation. No layout can presently be generated from this netlist since layouts for the add4 and sub4 components are not available in the VPNR library.

Example 3 is the 4-bit carry-lookahead adder described above as ADR in Table 8. The input VHDL description, the netlists generated by VTsynth, and the layout generated by VPNR are given. In order to realize the optimizations that were performed, a careful analysis of the input VHDL and output RNL description must be performed. Note that the c2 assignment statement (c2 <= (pr1 AND gn1) OR (gn1 AND Cin);) was synthesized using one component aoi22. The statement could have been synthesized using two AND gates and an OR gate which would have resulted in a larger and slower design. Instead, a complex gate was used to implement the function.

Example 1: Synthesis from Structural and Dataflow VHDL

Input VHDL file containing structural and dataflow VHDL constructs

```
entity inv is
   port (a : in bit;
        b : out bit);
end inv;

architecture beh of inv is
begin
end;

entity and2 is
   port (a,b : in bit;
        c : out bit);
end and2;

architecture beh of and2 is
begin
end;

entity or2 is
   port (a,b : in bit;
        c : out bit);
end or2;

architecture beh of or2 is
begin
end;


entity mixed_level is
   port (a,b : in bit;
        q   : out bit;
        G, H, I : in bit;
        J, K, L : inout bit);
end mixed_level;


architecture behav of mixed is

signal ai,bi,ci,di : bit;

component i1 is
   port (a : in bit;
        b : out bit);
end component;
```

Input VHDL file containing structural and dataflow VHDL constructs (cont.)

```
component a2 is
   port (a,b : in bit;
        c : out bit);
end component;

component o2 is
   port (a,b : in bit;
        c : out bit);
end component;

label A,B,C,D,E,p;
for A,B : i1 use entity inv(beh);
for C,D : a2 use entity and2(beh);
for E : o2 use entity or2(beh);

begin

-- structural

A: i1 port map (a,ai);
B: i1 port map (b,bi);
C: a2 port map (a,bi,ci);
D: a2 port map (b,ai,di);
E: o2 port map (ci,di,q);


-- dataflow

J <= (G and H) or (I and B);
K <= ((I or G) nor J) and H;
L <= G xor H;

end behav;
```

Synthesized RNL netlist (VTsynth output)

```
(macro mixed (a b g h i j k l q )
(local _10 _9 ai bi ci di )
(o2  ci di q )
(a2  b ai di )
(a2  a bi ci )
(i1  b bi )
(i1  a ai )
(ao22 g h i b j )
(o2 i g _9 )
(oi2 _9 j _10 )
(a2 _10 h k )
(xor g h l )
)
```

Synthesized structural VHDL netlist (VTsynth output)

```
entity mixed is
   port (
        a : in bit;
        b : in bit;
        g : in bit;
        h : in bit;
        i : in bit;
        j : inout bit;
        k : inout bit;
        l : inout bit;
        q : out bit);
end mixed;

architecture pure_structural of mixed is

   signal  _10, _6, _9, ai, bi, ci, di : bit;

component o2 is
   port (
        a : in bit;
        b : in bit;
        c : out bit);
end component;

for e: o2 use entity or2(beh);

component a2 is
   port (
        a : in bit;
        b : in bit;
        c : out bit);
end component;

for c: a2 use entity and2(beh);

for d: a2 use entity and2(beh);

component i1 is
   port (
        a : in bit;
        b : out bit);
end component;

for a: i1 use entity inv(beh);

for b: i1 use entity inv(beh);
```

Synthesized structural VHDL netlist (cont.)

```
begin

e: o2 port map (ci, di, q);
d: a2 port map (b, ai, di);
c: a2 port map (a, bi, ci);
b: i1 port map (b, bi);
a: i1 port map (a, ai);

L1: ao22 port map (g, h, i, b, j);
L2: o2 port map (i, g, _9);
L3: oi2 port map (_9, j, _10);
L4: a2 port map (_10, h, k);
L5: xor port map (g, h, l);

end pure_structural;
```

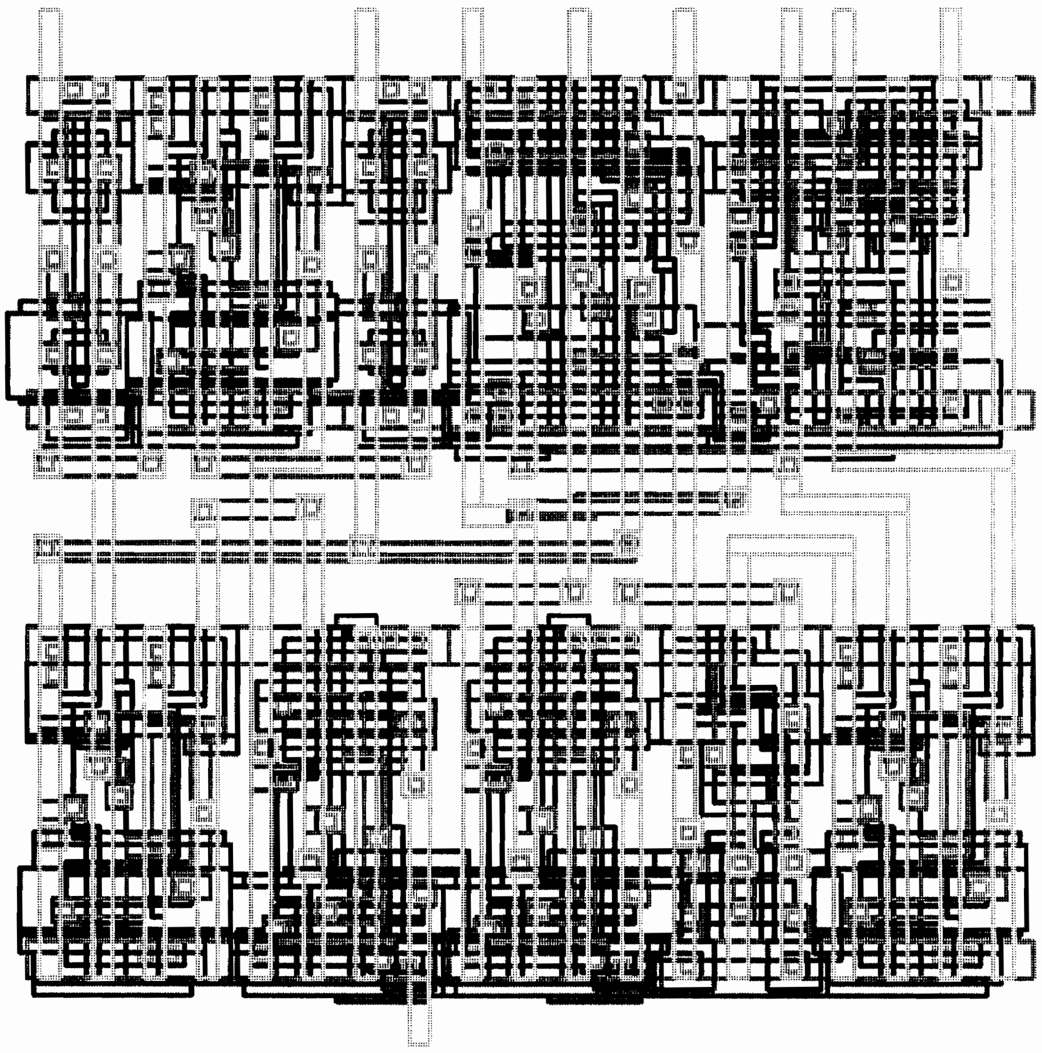


Figure 38. VPNR layout for Example 1

Example 2: Synthesis from Algorithmic VHDL

Input VHDL file containing algorithmic VHDL constructs

```
entity alg is
      port (a, b, c : in integer;
          d, e, f : out integer;
          x, y :    out integer);
end alg;

architecture behavior of alg is
      label p;
begin
p: process (a, b, c)
begin
      x <= a + b;
      y <= a - c;
      d <= a - b;
      e <= c - b;.
      f <= b + c;

end process;
end behavior;
```

Synthesized RNL netlist (VTsynth output)

```
(macro alg (a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3 c_0 c_1 c_2 c_3
        d_0 d_1 d_2 d_3 e_0 e_1 e_2 e_3 f_0 f_1 f_2 f_3
        x_0 x_1 x_2 x_3 y_0 y_1 y_2 y_3 )
(add4 state_0 a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3 x_0 x_1 x_2 x_3 )
(sub4 state_1 a_0 a_1 a_2 a_3 c_0 c_1 c_2 c_3 y_0 y_1 y_2 y_3 )
(sub4 state_2 a_0 a_1 a_2 a_3 b_0 b_1 b_2 b_3 d_0 d_1 d_2 d_3 )
(sub4 state_3 c_0 c_1 c_2 c_3 b_0 b_1 b_2 b_3 e_0 e_1 e_2 e_3 )
(add4 state_4 b_0 b_1 b_2 b_3 c_0 c_1 c_2 c_3 f_0 f_1 f_2 f_3 )
)
```

Synthesized structural VHDL netlist (VTsynth output)

```
entity alg is
   port (
       a_0 : in bit;
       a_1 : in bit;
       a_2 : in bit;
       a_3 : in bit;
       b_0 : in bit;
       b_1 : in bit;
       b_2 : in bit;
       b_3 : in bit;
       c_0 : in bit;
       c_1 : in bit;
       c_2 : in bit;
       c_3 : in bit;
       d_0 : out bit;
       d_1 : out bit;
       d_2 : out bit;
       d_3 : out bit;
       e_0 : out bit;
       e_1 : out bit;
       e_2 : out bit;
       e_3 : out bit;
       f_0 : out bit;
       f_1 : out bit;
       f_2 : out bit;
       f_3 : out bit;
       x_0 : out bit;
       x_1 : out bit;
       x_2 : out bit;
       x_3 : out bit;
       y_0 : out bit;
       y_1 : out bit;
       y_2 : out bit;
       y_3 : out bit);
end alg;
```

Synthesized structural VHDL netlist (cont.)

```
architecture pure_structural of alg is

begin

P1: add4 port map (state_0, a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3,
                   x_0, x_1, x_2, x_3 );
P2: sub4 port map (state_1, a_0, a_1, a_2, a_3, c_0, c_1, c_2, c_3,
                   y_0, y_1, y_2, y_3 );
P3: sub4 port map (state_2, a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3,
                   d_0, d_1, d_2, d_3 );
P4: sub4 port map (state_3, c_0, c_1, c_2, c_3, b_0, b_1, b_2, b_3,
                   e_0, e_1, e_2, e_3 );
P5: add4 port map (state_4, b_0, b_1, b_2, b_3, c_0, c_1, c_2, c_3,
                   f_0, f_1, f_2, f_3 );


end pure_structural;
```

Example 3: Synthesis of a 4-bit Carry-Lookahead Adder

Input VHDL file containing RTL description of Adder

```
entity adder4 is
      port (y1, y2, y3, y4 : in bit;
           z1, z2, z3, z4 : in bit;
           Cin : in bit;
           cout : out bit;
           Propagate : out bit;
           s1, s2, s3, s4 : out bit);
end adder4;

architecture dataflow of adder4 is

signal gn1, gn2, gn3, gn4, pr1, pr2, pr3, pr4 : bit;
signal c2, c3, c4 : bit;

begin

gn1 <= y1 and z1;
pr1 <= y1 xor z1;
s1 <= pr1 xor Cin;

c2 <= (pr1 and gn1) or (gn1 and Cin);

gn2 <= y2 and z2;
pr2 <= y2 xor z2;
s2 <= pr2 xor c2;

c3 <= (Cin and gn2 and gn1) or (gn2 and pr1 and gn1) or (gn2 and pr2);

gn3 <= y3 and z3;
pr3 <= y3 xor z3;
s3 <= pr3 xor c3;

c4 <= (Cin and gn3 and gn2 and gn1) or (pr1 and gn3 and gn2 and gn1) or
    (gn2 and pr2 and gn3) or (gn3 and pr3);

gn4 <= y4 and z4;
pr4 <= y4 xor z4;
s4 <= pr4 xor c4;

Cout <= (gn4 and gn3 and gn2 and gn1) or (pr2 and gn4 and gn3 and gn2) or
      (gn3 and pr3 and gn4) or (gn4 and pr4);

Propagate <= pr1 and pr2 and pr3 and pr4;

end dataflow;
```

Synthesized RNL netlist for Adder (VTsynth output)

```
(macro adder4 (cin cout propagate s1 s2 s3 s4 y1 y2 y3 y4 z1 z2 z3 z4 )
(local in_15 in_16 in_25 in_26 in_27 in_28 in_37 in_38 in_39 in_40 in_43
    in_44 c2 c3 c4 gn1 gn2 gn3 gn4 pr1 pr2 pr3 pr4 )
(a2 y1 z1 gn1 )
(xor y1 z1 pr1 )
(xor pr1 cin s1 )
(aoi22 pr1 gn1 gn1 cin c2 )
(a2 y2 z2 gn2 )
(xor y2 z2 pr2 )
(xor pr2 c2 s2 )
(aoi33 cin gn2 gn1 gn2 pr1 gn1 in_15 )
(a2 gn2 pr2 in_16 )
(o2 in_15 in_16 c3 )
(a2 y3 z3 gn3 )
(xor y3 z3 pr3 )
(xor pr3 c3 s3 )
(a4 cin gn3 gn2 gn1 in_25 )
(a4 pr1 gn3 gn2 gn1 in_26 )
(o2 in_25 in_26 in_27 )
(aoi32 gn2 pr2 gn3 gn3 pr3 in_28 )
(o2 in_27 in_28 c4 )
(a2 y4 z4 gn4 )
(xor y4 z4 pr4 )
(xor pr4 c4 s4 )
(a4 gn4 gn3 gn2 gn1 in_37 )
(a4 pr2 gn4 gn3 gn2 in_38 )
(o2 in_37 in_38 in_39 )
(aoi32 gn3 pr3 gn4 gn4 pr4 in_40 )
(o2 in_39 in_40 cout )
(a2 pr1 pr2 in_43 )
(a2 in_43 pr3 in_44 )
(a2 in_44 pr4 propagate )
)
```

Synthesized structural VHDL netlist (VTsynth output)

```
entity adder4 is
   port (
       cin : in bit;
       cout : out bit;
       propagate : out bit;
       s1 : out bit;
       s2 : out bit;
       s3 : out bit;
       s4 : out bit;
       y1 : in bit;
       y2 : in bit;
       y3 : in bit;
       y4 : in bit;
       z1 : in bit;
       z2 : in bit;
       z3 : in bit;
       z4 : in bit);
end adder4;

architecture pure_structural of adder4 is

   signal  in_15, in_16, in_25, in_26, in_27, in_28, in_37, in_38,
        in_39, in_40, in_43, in_44, c2, c3, c4, gn1, gn2, gn3, gn4,
        pr1, pr2, pr3, pr4 : bit;


begin

L0: a2 port map (y1, z1, gn1);
L1: xor port map (y1, z1, pr1);
L2: xor port map (pr1, cin, s1);
L3: aoi22 port map (pr1, gn1, gn1, cin, c2);
L4: a2 port map (y2, z2, gn2);
L5: xor port map (y2, z2, pr2);
L6: xor port map (pr2, c2, s2);
L7: aoi33 port map (cin, gn2, gn1, gn2, pr1, gn1, in_15);
L8: a2 port map (gn2, pr2, in_16);
L9: o2 port map (in_15, in_16, c3);
L10: a2 port map (y3, z3, gn3);
L11: xor port map (y3, z3, pr3);
L12: xor port map (pr3, c3, s3);
L13: a4 port map (cin, gn3, gn2, gn1, in_25);
L14: a4 port map (pr1, gn3, gn2, gn1, in_26);
L15: o2 port map (in_25, in_26, in_27);
L16: aoi32 port map (gn2, pr2, gn3, gn3, pr3, in_28);
L17: o2 port map (in_27, in_28, c4);
L18: a2 port map (y4, z4, gn4);
L19: xor port map (y4, z4, pr4);
```

Synthesized structural VHDL netlist (cont.)

```
L20: xor port map (pr4, c4, s4);
L21: a4 port map (gn4, gn3, gn2, gn1, in_37);
L22: a4 port map (pr2, gn4, gn3, gn2, in_38);
L23: o2 port map (in_37, in_38, in_39);
L24: aoi32 port map (gn3, pr3, gn4, gn4, pr4, in_40);
L25: o2 port map (in_39, in_40, cout);
L26: a2 port map (pr1, pr2, in_43);
L27: a2 port map (in_43, pr3, in_44);
L28: a2 port map (in_44, pr4, propagate);

end pure_structural;
```

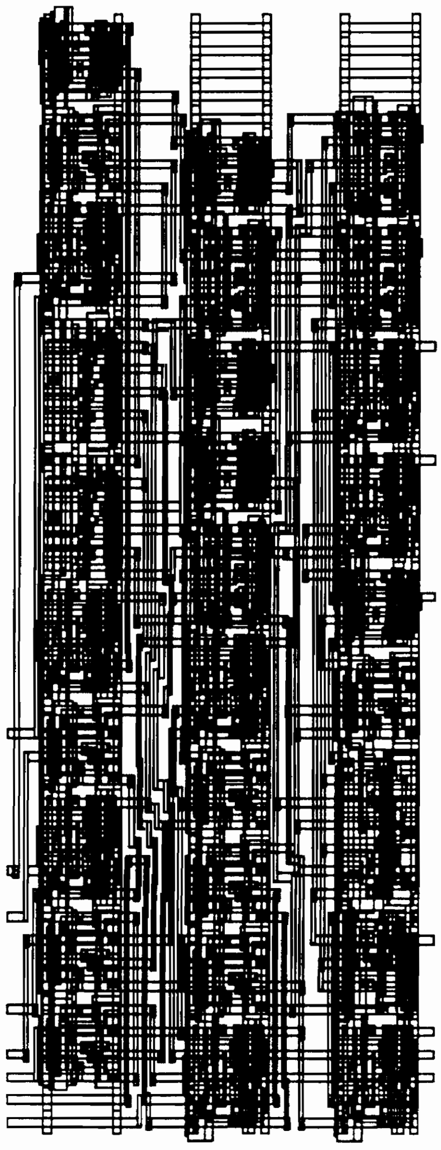


Figure 39. VPNR layout of 4-bit Adder

Chapter 7
Future Work

The VTsynth synthesis system is currently able to synthesize models written using the VHDL subset described in Chapter 3. Physical layouts have been produced from structural and dataflow models. VTsynth is able to generate a netlist from algorithmic models containing sequential statements within VHDL processes. No physical layouts are currently produced by the system for algorithmic models since the components needed for high-level synthesis are not available in the VPNR library.

The VTsynth system may be enhanced in several ways in order to make it a more complete synthesis system. Currently, the synthesis library only consists of gates and flip-flops. More components which support high-level synthesis need to be added. Components such as registers, adders, multipliers, ALUs, etc. need to be added before high-level synthesis can be performed. Scheduling and allocation algorithms must also be implemented along with high-level optimizations.

Some enhancements may also be made at the circuit level. Currently, delay values for components are constant values calculated based on a fanout of two inverters. In actuality, the delay of a component is based on the load that it drives (fanout). A more realistic calculation of the delay can be made by determining the actual fanout which is available from the synthesized netlist. This information would then be back annotated into the structural VHDL output by VTsynth.

Synthesis also needs to be an interactive process. The designer needs to be able to communicate the constraints imposed on the design to the synthesis system. These constraints include information such as: which set of components or library is available to choose parts from, what kind of optimizations to perform (or not perform), and the defaults for synthesis (e.g., map all integer variables to 8-bit registers or 32-bit registers). Currently, the VTsynth user interface consists only of a few command line options (such as the entity(architecture) to synthesize and the names of the input files). Future enhancements to the system should include a user interface which interactively allows the user to communicate with the synthesis process.

The subset of VHDL defined for the VTsynth synthesis system may also be updated to include support for more abstract VHDL constructs. As more sophisticated synthesis algorithms are developed, methods of generating hardware from these types of descriptions are also likely to develop. These methods can then be incorporated into the VTsynth synthesis system. This includes techniques for synthesizing algorithmic constructs such as loop statements and signal attributes which are not presently supported by VTsynth.

Verification of the synthesis process still has some issues for future research. These include automatic insertion of delay values into the structural VHDL model generated by VTsynth and formal verification of the pre-synthesis and post-synthesis models.

Several high-level and RTL level optimizations may be added in the future. Currently, only AND-OR logic is optimized and mapped into complex gates. In the future, NAND-NAND and NOR-NOR logic may also be optimized and mapped into complex gates.

Chapter 8
VTsynth User's Manual

This chapter describes the process by which VTsynth is executed. Included are descriptions of all of the steps necessary to generate the netlists and the physical layouts using VTsynth and VPNR.

The Vtsynth system may be executed on an IBM compatible PC running DOS 3.3 or higher. The input to VTsynth is a VHDL model written using the synthesis subset described in Chapter 3. The command line to execute Vtsynth is as follows:

vtsynth [-e entity.architecture] input_file_list

The brackets ([]) denote that the parameter is optional. Therefore, the "-e entity.architecture" parameter is optional. Multiple file names may be entered at the command line. Every file specified on the command line will be processed. If the "-e entity.architecture" parameter is missing, then no synthesis will be performed on the input files. The input files will only be checked for syntax errors. Running VTsynth without the -e option allows the user to ensure that the input model is syntactically correct before synthesis.

Multiple entities and architectures may be placed in the input VHDL files. Only the entity.architecture pair specified on the input command line will be synthesized. The command line to synthesize the 4-bit adder (ADR) of Chapter 6 is:

```
vtsynth -e adder4.dataflow adder.vhd
```

The input file name is "adder.vhd". The entity name is "adder4" and the architecture name is "dataflow".

The format of the VTsynth command line may be displayed by entering "vtsynth" without any parameters. The program recognizes that no parameters were given and outputs the format of the command. The following lines would be displayed:

```
   ** VTsynth VHDL Synthesis System, Version 1.0 **
usage: vtsynth [-e entity.architecture] files.vhd
```

Error Reporting:

Vtsynth performs much error checking and reporting during analysis and synthesis. The types of error messages include warnings, recoverable errors, and fatal errors. Warnings are issued to inform the user as to what is happening during synthesis. For example, a message such as "line 24: Warning - mapping integer to bit_vector (31 downto 0)" is an example of a warning message. Recoverable errors are ones which allow the parsing to continue after the error is flagged. An example of a recoverable error is, "line 15: 'EVENT attribute is not allowed". Fatal errors are ones which cause parsing to stop. These may be certain syntax errors, memory or stack overflow, etc. All warning and error messages are accompanied with a line number corresponding to the location of the error in the source file. The warning and error messages are designed to be self explanatory.

Running VPNR:

A successful execution of VTsynth will result in the creation of two output files, an RNL netlist and a structural VHDL netlist. The RNL netlist may then be sent to VPNR for placement and routing in order to generate the physical layout. The VPNR software is presently only available for VAX computers running the Ultrix operating system. Therefore, the RNL netlist generated by VTsynth must be moved to the VAX environment containing the VPNR software. The command to run VPNR in the Ultrix environment is as follows:

```
vpnr file.net
```

where file.net is the name of the VTsynth generated RNL netlist. VPNR generates a directory file.magic which contains a Magic description of the physical layout.

The method of synthesizing VHDL models using VTsynth and VPNR was described. The commands for executing Vtsynth in order to generate the RNL and structural VHDL netlists was shown. Also, the process of using VPNR to generate the physical layout from the RNL netlist was described.

References

[1] *IEEE Standard VHDL Language Reference Manual*, IEEE, New York, NY, March 1988.

[2] Donald E. Thomas, et. al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publishers, Boston, MA, 1990.

[3] Giovanni De Micheli, David Ku, Frederic Mailhot, Thomas Truong, "The Olympus Synthesis System", *IEEE Design and Test*, October 1990, pp. 37-53.

[4] R. K. Bryton, R. Camposano, G. De Micheli, R. H. J. M. Otten, J. van Eijndhoven, "The Yorktown Silicon Compiler", *Silicon Compilation*, Addison-Wesley, Reading, MA, 1988, pp. 204-310.

[5] J. Lis, D. D. Gajski, "VHDL Synthesis Using Structured Modeling", *Proceedings of the 16th Design Automation Conference*, 1989, pp. 606-609.

[6] Paul Harper, Stanley Krolikoski, Oz Levia, "Using VHDL as a Synthesis Language in the Honeywell VSYNTH System", *CHDL '89*, 1989, pp. 315-330.

[7] Microsoft C 5.0 Programming Language Manuals, Microsoft Corporation, 1987.

[8] G. Zimmerman, "The MIMOLA Design System : A Computer-Aided Processor Design Method", *Proceedings of the 16th Design Automation Conference*, 1979, pp. 53-58.

[9] James R. Armstrong, *Chip-Level Modeling with VHDL*, Prentice Hall, New Jersey, 1989.

[10] *MCC CAD VHDL System User's Guide*, MCC, 1989.

[11] *Vantage Spread Sheet Analyst User's Guide*, Vantage Analysis Systems, June 1989.

[12] *Intermetrics VHDL User's manual*, Intemetrics, 1988.

[13] *Synopsys VHDL User's Manual*, Synopsys, 1990.

[14] Carver Mead, Lynn Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, MA, 1980.

[15] Niklaus Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Hills, NJ, 1976.

[16] M. C. McFarland, A. C. Parker, R. Camposano, "Tutorial on High-Level Synthesis", *Proceedings of the 25th Deisgn Automation Conference*, 1988, pp. 330-336.

[17] Alfred V. Aho, Ravi Sethi, Jeffrey Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1988.

[18] William A. Barrett, Rodney M. Bates, David A. Gustafson, John D. Couch, *Compiler Construction - Theory and Practice*, SRA, Chicago, IL, 1986.

[19] *VPNR Users Guide*, Microelectronics Center of North Carolina (MCNC), 1988.

[20] *Magic User's Guide*, University of California, Berkley, 1986.

[21] *Bison User's Manual*, GNU Software Foundation, 1987.

[22] M. E. Lesk, E. Schmidt, "Lex - a lexical analyzer generator", *Unix Programmer's Manual*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[23] Axel T. Schreiner, H. George Friedman, Jr., *Introduction to Compiler Construction with UNIX*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1985.

[24] P. G. Paulin, J. P. Knight, "Scheduling and Binding Algorithms for High-Level Synthesis", *Proceedings of the 26th Deisgn Automation Conference*, 1989, pp. 1-6.

[25] Steven M. Rubin, *Computer Aids for VLSI Design*, Addison Wesley, 1987, pp. 115-160.

[26] Neil Weste, Kamran Eshraghian, *Principles of CMOS VLSI Design*, Addison-Wesley, Reading, MA, 1985.

[27] Daniel Gajski, *Silicon Compilation*, Addison-Wesley, Reading, MA, 1988.

[28] Charles H. Roth, *Fundamentals of Logic Design*, West Publishing Company, St. Paul, MN, 1985.

[29] Brian W. Kernighan, Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ, 1984, pp. 223-287.

[30] VCOMP source code, University of Pittsburgh, 1988.

[31] *RNL User's Guide*, University of Washington, 1987.

[32] Rajiv Jain, et. al., "Experience with the ADAM Synthesis System", *Proceedings of the 26th Deisgn Automation Conference*, 1989, pp. 56-61.

[33] J. Rabaey, et. al., "Cathedral II: A Synthesis System for Multiprocessor DSP Systems", *Silicon Compilation*, Addison-Wesley, Reading, MA, 1988.

[34] P. G. Paulin, J. P. Knight, E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis", *Proceedings of the 23rd Deisgn Automation Conference*, 1986, pp. 263-270.

Appendix A

Components in the VPNR Standard Cell Library

Component Name	Function	Size[1]	Delay[2] (ns)
a2s	2-input AND	87 x 48	3.700
a3s	3-input AND	87 x 48	5.135
a4s	4-input AND	87 x 60	5.327
ai2s	2-input NAND	87 x 36	2.143
ai3s	3-input NAND	87 x 48	2.543
ai4s	4-input NAND	87 x 60	3.481
aoi2111s	NOT [(a1 AND a2) OR b OR c OR d]	87 x 72	6.438
aoi211s	NOT [(a1 AND a2) OR b OR c]	87 x 60	4.639
aoi21s	NOT [(a1 AND a2) OR b]	87 x 48	2.818
aoi2211s	NOT [(a1 AND a2) OR (b1 AND b2) OR c OR d]	87 x 84	7.376
aoi221s	NOT [(a1 AND a2) OR (b1 AND b2) OR c]	87 x 72	5.332
aoi2221s	NOT [(a1 AND a2) OR (b1 AND b2) OR (c1 AND c2) OR d]	87 x 108	8.426
aoi222s	NOT [(a1 AND a2) OR (b1 AND b2) OR (c1 AND c2)]	87 x 96	5.990
aoi22s	NOT [(a1 AND a2) OR (b1 AND b2)]	87 x 60	3.283

1. Size of standard cells based on 2 micron SCMOS technology [19].
2. Delay values assume a load of two inverters.

aoi31s	NOT [(a1 AND a2 AND a3) OR b]	87 x 60	3.730
aoi32s	NOT [(a1 AND a2 AND a3) OR (b1 AND b2)]	87 x 72	4.132
aoi33s	NOT [(a1 AND a2 AND a3) OR (b1 AND b2 AND b3)]	87 x 84	4.631
dff	D flip-flop	87 x 108	13.396
i1s	Inverter	87 x 24	1.753
i2s	Inverter	87 x 24	0.976
i4s	Inverter	87 x 36	0.735
i8s	Inverter	87 x 48	0.586
o2s	2-input OR	87 x 48	4.010
o3s	3-input OR	87 x 48	5.797
o4s	4-input OR	87 x 60	8.510
oai211s	NOT [(a1 OR a2) AND b AND c]	87 x 60	4.169
oai21s	NOT [(a1 OR a2) AND b]	87 x 48	3.033
oai221s	NOT [(a1 OR a2) AND (b1 OR b2) AND c]	87 x 72	5.035
oai222s	NOT [(a1 OR a2) AND (b1 OR b2) AND (c1 OR c2)]	87 x 84	5.124
oai22s	NOT [(a1 OR a2) AND (b1 OR b2)]	87 x 60	3.760
oai31s	NOT [(a1 OR a2 OR a3) AND b]	87 x 60	4.576
oai32s	NOT [(a1 OR a2 OR a3) AND (b1 OR b2)]	87 x 72	5.010
oai33s	NOT [(a1 OR a2 OR a3) AND (b1 OR b2 OR b3)]	87 x 84	5.448
oi2s	2-input NOR	87 x 36	2.057

oi3s	3-input NOR	87 x 60	3.508
oi4s	4-input NOR	87 x 72	4.848
xnors	2-input XNOR	87 x 60	4.389
xors	2-input XOR	87 x 60	4.163

Vita

Sandeep Ramesh Shah was born in Bombay, India on January 9, 1965. At the age of seven, he immigrated to the United States with his family. During grade school and high school, he acquired an interest for electronics and computers. After taking two years of high school electronics, he decided to pursue a degree in Electrical Engineering at Virginia Tech.

After completing a B.S. degree in Electrical Engineering, he decided to continue his education at Virginia Tech and pursue an M.S. degree in the same field. Under the guidance of Dr. Tront and Dr. Armstrong, he found himself moving into the CAD, VHDL, and VLSI areas of Computer Engineering. He is currently a candidate for the M.S. degree in Electrical Engineering.

His research interests include design automation for VLSI, VHDL, synthesis, compiler design, and software engineering.

Sandeep Shah