INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Automated VHDL Composition from AHPL

BY

Faisal Mohammad Zafar Soleja

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

Computer Science

August 1996

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by **FAISAL MOHAMMAD ZAFAR SOLEJA** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in COMPUTER SCIENCE.

THESIS COMMITTEE

Sadiq Sait · M

Dr. Sadiq M. Sait (Chairman)

٢٧/٩/٩٥

Dr. Muhammed S. Al – Mulhem (Co – Chairman)

Dr. Muhammad S. T. Benten (Member)

Dr. Habib Youssef (Member)

٢٧/٩/٩٥

Department Chairman

Dean, College of Graduate Studies

Date: 8-6-91

This thesis is dedicated to my three lovely daughters

Ayesha, Sara and Asma

Acknowledgments

First and foremost, all Praise is for Almighty Allah for having me guided at every stage of my life.

Acknowledgement is due to King Fahd University of Petroleum & Minerals for providing facilities and support to this work.

I am indebted to my thesis Chairman, Dr. Sadiq M. Sait, for his time, guidance and assistance. I am also highly thankful to my thesis Co-chairman and Department Chairman, Dr. Muhammed S. Al-Mulhem for all his sincere help and support. I would like also to place on record my great appreciation for the cooperation and suggestions extended by my committee members Dr. Muhammad S. T. Benten and Dr. Habib Youssef. My special thanks to all my thesis committee members, whose patience and forbearance have helped me to complete this work.

I wish to express my gratitude to my father, Dr. Mohammad Zafar Soleja, for his moral support, encouragement and motivation. To my mother and my sister I thank them for all their kind support. Lastly, I want to express my appreciation to my wife and my daughters for their tolerance and sacrifice.

Contents

List of Tables

List of Figures

Abstract

Name: Faisal Mohammad Zafar Soleja

Title: Automated VHDL Composition from AHPL

Major Field: Computer Science

Date of Degree: August 1996

Hardware description languages (HDLs) have been widely used for documentation, communication and verification. They have also been used as input specification languages to Design Automation (DA) systems which synthesize VLSI layouts. AHPL is an HDL that has been in use for the past three decades in modeling digital systems. Recently a language called VHDL (VHSIC Hardware Description Language) developed under the auspices of the United States Department of Defense Very High Speed Integrated Circuits Program, is rapidly emerging as the next generation Design Automation Language. However, because of its large size and sophistication, VHDL is a difficult language to learn and master. On the other hand, AHPL is a very concise language that can be mastered within few hours. The objective of this research is to develop a tool that will assist designers quickly learn and model in the VHDL language. In this thesis, we present a tool for the automatic composition of VHDL descriptions from their equivalent AHPL specifications. Central to the composition algorithms is a template which is the skeleton of a generic VHDL model consisting of a small subset of VHDL constructs that are sufficient to capture VHDL equivalent descriptions of any input AHPL model.

Master of Science Degree

King Fahd University of Petroleum and Minerals

Dhahran, Saudi Arabia

August 1996

خلاصة الرسالة

الاسم: فيصل محمد ظفر صوليجا

عنوان الرسالة: آلية VHDL المؤلفة من AHPL

التخصص: علم الحاسب الآلي

تاريخ الشهادة: اغسطس ١٩٩٦م

تستخدم لغات وصف الدوائر(HDLs) غالبا في التوثيق والاتصال و التحقق ، وكذلك تستخدم كلغة توصيف مدخلة الى أنظمة التصميم اللآلي (DA) المستخدمة في توليف نماذج الدوائر المتكاملة ذات النطاق الواسع جدا (VLSI). وخلال العقدين الماضيين استخدمت لغة (AHPL) – وهي من لغات وصف الدوائر (HDL) – في نمذجة الأنظمة الرقمية. ومنذ عهد قريب تم تطوير لغة اسمها VHDL (لغة VHSIC لوصف الدوائر) تحت رعاية برنامج وزارة الدفاع الامريكية لتطوير الدوائر المتكاملة ذات السرعات الفائقة. وهذه اللغة تأخذ طريقها بسرعة كلغة الجيل التالي لأنظمة التصميم الآلي. ولكن بسبب مدى ضخامتها وتطورها جعلت من VHDL لغة صعبة التعلم و الابداع. وبالمقابل فان AHPL لغة مختصرة يمكن التفنن فيها خلال ساعات قليلة. ان الهدف من هذا البحث هو تطوير أداة تسهل على المصممين سرعة التعلم و الابداع في لغة VHDL. نقدم في هذا البحث أداة تأليف آلية لتوصيف VHDL باستخدام مواصفات AHPL المكافئة لها.والجزء الرئيس في الخوارزمية المقترحة هو قالب المعايرة وهو مخطط هيكلي لنموذج VHDL توليدي. يتكون من مجموعة جزئية صغيرة من تركيبات VHDL كافية لاستخلاص وصف VHDL المكافئ لأي نموذج AHPL داخل. ويوضح أحد الأمثلة طريقة عمل المؤلف المقترح.

درجة ماجستير علوم

جامعة الملك فهد للبترول والمعادن

الظهران , المملكة العربية السعودية

اغسطس ١٩٩٦م

Chapter 1

Introduction

In the era of VLSI, the size and complexity of digital systems have grown tremendously. To effectively and efficiently design such systems many computer-aided design tools were introduced and among them was the development of the Hardware Description Languages (HDLs). Since then, HDLs have been extensively used to describe hardware for the purpose of simulation, modeling, testing, design, and documentation of digital systems. At first, the use of formal HDLs was confined to academic circles. They were used in the teaching of computer architecture concepts as well as the verification of new hardware through simulation. HDLs were not commercialized until early 1980's. Since then, they saw a wide use as interface between tools, as documentation of new designs, as interfaces to design databases, and as interfaces between IC manufacturers. The commercialization of HDLs brought with it the standardization wave. Standardization was motivated by several reasons, most

important among them were:

- the need to provide a common interface between CAD tools, this is to answer the portability issue,
- the need to have a multi-purpose language, that is, a language that is not confined to a single level of circuit description, and
- the requirement for a language to support the hierarchical description of large designs; this is indicated by the increasing level of density as well as complexity of new VLSI circuits.

It is this standardization effort that led to the advent of the VHDL language. VHDL is now the most important standard in the CAD community mainly because it satisfies the three requirements listed above [DG86]. Furthermore, VHDL is supported by efficient hierarchical simulators from the system level to the gate level and is becoming the language of choice for government, industry and academia participating in electronic research, business, and education.

The advent of VHDL has provided a foundation for the communities of design automation, design and test, and manufacturing, to reduce product cost through improved designer productivity and maintenance capabilities [Wax86]. A standard hardware description of a design has enabled developers of design workstation, test equipment, and design automation software to provide common data interface among their tools. Standard human-readable description of design data facilitates

exchange of data among designers, enabling them to work effectively with design tools [LSU89]. VHDL not only provides the feature for suitable descriptive medium but also a design tool. It also provides a standard textual means of description for hardware components at abstraction levels ranging from the logic gate level to digital system level [LMS86]. It provides precise syntax and semantics documentation for the hardware components, enabling design transfer both within and among organizations.

Another hardware description language that enjoyed and still enjoys widespread use (mostly in academia) is the AHPL language [HDL92a, HP73]. AHPL is a hardware description language based upon the notational conventions of APL. AHPL was developed in an academic environment to teach digital system design. Its basic structural similarity to high-level programming languages makes it accessible to wider class of users than would be the case with a more specialized hardware language. AHPL is neither a standard nor a multipurpose language. It is a language for the description of digital system at the Register Transfer level (RTL). AHPL is the easiest hardware language to teach/learn among all existing HDLs. VHDL on the other hand, can be quite intimidating for a first time user.

This thesis presents the design of a *Composer* which automatically generates a VHDL model from an equivalent AHPL input model. This *composition* is achieved in two stages with the help of a especially designed template and an external library of predefined functions. The *Composer* accurately obtains a VHDL model from any

AHPL input model. Both models are equivalent in their function and the level of description of hardware (RTL level) [SYBS93, SYBS94].

1.1 Motivation

The principle motivation for the development of such a *Composer* is three-fold:

1. Rapid prototyping of VHDL models.
2. Easy migration from AHPL-like languages to VHDL.
3. An educational tool to aid in the teaching of a large HDL such as VHDL (with a large variety of features and constructs).

The *Composer* will also help reverse engineer the high-level synthesis process. The AHPL language is a purely RTL language. The composition of VHDL models from their AHPL counterparts will provide insight into how best one can perform high-level synthesis of digital systems from VHDL descriptions. The composition process gives a good understanding of how one can perceive the hardware correspondence from VHDL model.

VHDL is a multi-purpose HDL and is among the hardest HDLs to master. On the other hand, AHPL is a single purpose language, is relatively small and is among the easiest to learn. We believe that the *Composer* developed in this thesis will assist AHPL modelers to easily migrate and learn VHDL. Furthermore in academia as well

as for novice VHDL users the *Composer* can play the beneficial role of a computer aided teaching tool in helping users quickly learn RTL modeling in VHDL. More experienced users can also find the *Composer* helpful to quickly generate a correct RTL-level VHDL model, so that they only need to worry about generating VHDL architectural descriptions at higher levels (behavioral, timing, performance issues).

The hardware design implemented using AHPL are usually small and concise. In contrast VHDL code tends to be verbose and lengthy due to the strong data-typing and data declaration features of VHDL for every element to be declared before use [NS86]. Although this feature is beneficial for comprehensive design documentation and readability, it discourages designers to migrate to VHDL. The *Composer* has ability to lessen burden of writing the VHDL code by generating entire architecture of the design including the initial declaration and the correct syntactic execution statements.

1.2 Literature Survey

Early work in this area mainly concentrated on methodology for using VHDL to model design at different levels. The facility of VHDL to describe hardware at several abstraction levels from the logic gate to the system level [LSU89] was encouraging factor for hardware designer to start modeling in VHDL. The increasing role of

VHDL as the preferred design language also encouraged researchers to use VHDL for high-level synthesis. Previous attempts at synthesizing from VHDL did not fully take the advantage of the VHDL capability to represent hardware design at varying levels.

VHDL provides suitable design environment for specifying, simulating and synthesizing digital hardware [NBD92]. The use of VHDL for synthesis has presented some problems as VHDL was designed mainly as a documentation language. As a result, researchers have defined various subsets for synthesis.

Although a lot of attention has recently been given to synthesis from other forms of hardware representation, the process of synthesis is not new. A great deal of useful work has been contributed on synthesis from behavioral specification [RKDV92], register transfer [NS90], and structural [LG89] level description. Interested readers are referred to [WC91] for an extensive survey of the research and development of synthesis systems.

In early development of behavioral synthesis, a VHDL Synthesis System (VSS) [LG89] was implemented to describe the methodology to generate structural design. Behavioral synthesis is defined as the translation of a behavioral description into a structural description. In VHDL, designs can be described in several ways and at several different levels of abstraction. The authors of the VSS introduced modeling styles that will allow for efficient generation of high quality designs. This synthesis system supports four design models: combinational logic, functional description,

register transfer description, and algorithmic designs. An interesting modeling approach was defined for describing the state machine design and to specify the operations to be processed for each machine state. Each state comprises one or more triplets that specify the actions to be performed. Each triplet is made up of a condition, a next state, and a set of operations. The VHDL guarded block statements are used to represent each machine state of a design and the block guard to specify the clocking mechanism.

In later development the concept of templates for synthesis from VHDL [NS90] was introduced. Two different VHDL templates for synthesis description styles were implemented: one for dataflow level of description and the other at more behavioral level. For both of the templates a general synthesis subset of VHDL construct was defined. This subset was used to design and describe the clocking schemes. The clocking mechanism used in each of the template was: explicit clocking in which a variable is used to keep track of the present state; and implicit clocking in which a boolean vector is used to determine the processing step. Although both of the templates are synthesizable and a hardware correspondence exists for them, their synthesis style does not fully provide the facility to easily define the complete state table of a finite state machine.

In [JW89], a prototype system called VCOMP was developed with the intention to simplify the introduction of VHDL for novice hardware designers. VHDL Composition System (VCOMP) is an environmental tool to provide the functions of a

tutorial and design developer for beginners in VHDL. This system allows users to effectively create hardware description of a system and formulate test vectors for simulation to verify correctness of the design. In general this system was confined in helping the hardware designer to learn the syntax of VHDL and to produce code for simulation and testing.

The VHDL language is designed to be efficiently simulated. Traditionally, designers have performed simulation with high-level simulation languages or in ad-hoc fashion by writing custom simulators in C or some other programming language. VHDL enables simulation across many different levels of abstraction. Although, VHDL presents many syntactic constructs for high-level system simulation [NS86], VHDL offers tremendous advantage of mixing high-level simulation with low-level simulation. This ability of VHDL to do simulation at multilevel allows rapid-prototyping process for system designs [SB92].

Simulation for VHDL models has often perceived to be a performance disadvantage compared to simpler and better established hardware description languages due to magnanimity and complexity of the language. Many VHDL simulators have been implemented to accelerate simulation performance by means of optimizing compilation and parallel simulation [WS92]. New compilation techniques are available to reduce runtime complexity and to promote concurrency in the VHDL models.

In view of VHDL's growing use and pervasive impact on electronic technology, literature survey has revealed that extensive work is being done in VHDL-related

research and development. Early work concentrated in demonstrating the variety of design approaches available in VHDL. Later work concentrated in more serious usage aspect of high-level synthesis from VHDL specification. Much useful work was done in synthesis of hardware correspondence from VHDL model but it was confined to subset of VHDL constructs. To lessen the burden of synthesis, the concept of templates was introduced. Templates provide a variety of styles for modeling in VHDL.

1.3 Conclusion

In this opening chapter, the significance of the thesis work was presented. The basic objective of the thesis was, quite simply, to design and implement a *Composer* that will assist in transformation of AHPL specification to its functionally corresponding equivalent VHDL model. The primary motivation for the development of the *Composer* was to provide the tool to quickly and easily migrate to VHDL language for people already familiar with AHPL-like languages. The success and the widespread use of the VHDL has encouraged hardware designers for using VHDL for high-level synthesis. The *composition* of VHDL model from its AHPL counterpart gives the opportunity to describe as how best one can perform high-level synthesis of digital system from VHDL description. The *composition process* provides a better

understanding of how one can perceive the hardware design of a VHDL model.

In the following chapter an overview of Hardware Description Languages (HDL) is presented. This is followed by a series of chapters describing in detail the whole *composition process*. The thesis ends with the conclusion chapter that summarizes the work.

Chapter 2

Hardware Description Languages

This chapter present the introductory overview of Hardware Description Languages (HDL). The chapter starts with the definition of HDL and specifies some of the current usages of HDL. This is followed by the general overview of the two concerned HDL namely, AHPL and VHDL. Also included in the chapter is a comparison analysis study of nine well known HDLs and a summary of their features.

2.1 Introduction to Hardware Description Languages

Hardware Description Language (HDL) is a notational medium for the precise capture of certain features of a hardware design. They are extensively used for the purpose of simulation, modeling, testing, design, and documentation of digital de-

sign. These languages provide a convenient and compact format for representing the output of various design stages.

Hardware description languages basically comprise of a set of symbols and notations for the representation of digital circuits. Some HDL also have the ability to model the hardware at one or more levels of abstraction. HDL software includes simulation programs for the hardware design verification, and synthesis programs for the facility for automatic hardware generation [Nav93].

Today's hardware description languages benefit from the efforts of designers of hardware description languages dating back to the mid 1960s. Figure 2.1 shows the evolution of hardware description languages over the last few decades. In following section an interesting look is provided at two of these HDLs, namely AHPL and VHDL. Finally in the last section a comparison summary is included of AHPL and VHDL against some other well known HDLs.

2.2 AHPL and VHDL

In this section a brief overview of both AHPL and VHDL is presented and discussed. The section is not intended to be a tutorial on either hardware description language. Interested readers are referred to [HP73] for details on AHPL, and to [Arm89, VHD88, Per93, Nav93, LSU89] for details on VHDL. The sole purpose here is to briefly introduce both languages and highlight some of their key elements which

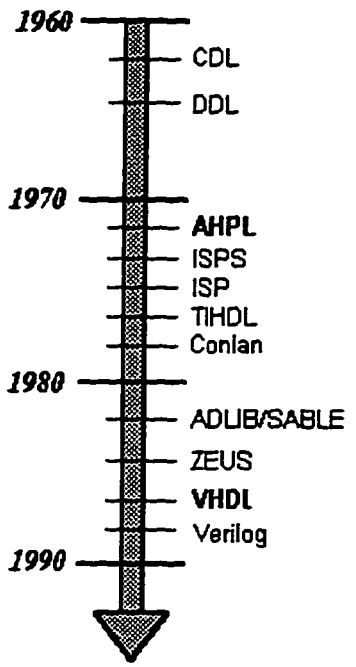

Figure 2.1: HDL Time Line.

are necessary for the description of the *Composer*.

2.2.1 AHPL Overview

AHPL is a hardware description language for describing the hardware at RTL level. AHPL came into existence in academic environment after the need arose for an easily manageable medium to teach digital system design. Since last two decades it has served a useful tool for teaching computer organization. The AHPL language is based on a subset notation of APL with many special added features to describe hardware specific details. The language is powerful enough to describe highly complex digital systems in an elegant way, yet it is the most compact and easiest to learn of all hardware description languages.

AHPL was designed as a hardware synthesis language and, therefore, is time-step oriented. In this language data assignment to registers are done synchronously by an implicit clock. AHPL does not provide any support for asynchronous hardware designs. In AHPL, digital designs are described using interactive concurrent modules. Iterative combinational networks such as adders, decoders, etc., can be described as Combinational Logic Units (CLUs). The language does not support timing mechanism and assignments of values to buses have immediate effect, while those to registers happen at the trailing edge of the clock. Figure 2.2 shows an AHPL description of a 4-bit multiplier of Figure 2.3 that will be later used as the vehicle to illustrate the composition system.

```
MODULE          : MULTIPLIER.
MEMORY          : AC1[4]; AC2[4]; COUNT[2]; EXTRA[5]; BUSY.
EXINPUTS        : DATAREADY; CLOCK; RESET.
EXBUSES         : INPUTBUS[8].
OUTPUTS         : RESULT[8]; DONE; BUSYOUT.
CLUNITS         : INC[2]<: INCR{2}.
CLUNITS         : ADD[5]<: ADDER{5}.
BODY SEQUENCE:  CLOCK

1    AC1,AC2<=INPUTBUS; EXTRA<=5$0; => (^DATAREADY)/(1).
2    BUSY<=1$1; => (^AC1[3])/(4).
3    EXTRA <=ADD[0:4](EXTRA[1:4];AC2).
4    EXTRA,AC1<=1$0,EXTRA,AC1[0:2]; COUNT<=INC(COUNT);
     => (^(&/COUNT))/(2).
5    RESULT=EXTRA[1:4],AC1; DONE=1$1; BUSY <=1$0; =>(5).
ENDSEQUENCE
     BUSYOUT=BUSY;
     CONTROLRESET(RESET)/(1).
END.
```

Figure 2.2: AHPL Description of a Multiplier.

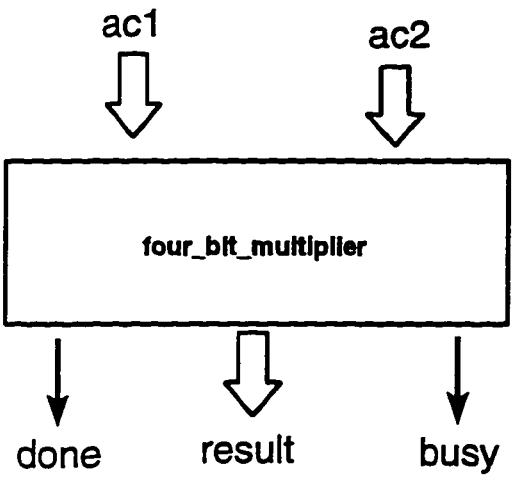

Figure 2.3: Block Diagram of 4-bit Multiplier.

As can be observed in Figure 2.2 every AHPL description consists of basically three parts:

- a declaration part
- procedure part
- non-procedural part

The declaration part consist of description of all the registers and buses. The procedural part describes the state machine, and this is followed by a non-procedural part. Referring to Figure 2.2 the numbered steps between the keywords SEQUENCE and ENDSEQUENCE form the procedural part defining the states of the sequential machine. In this part, a statement is active only when the machine is in the corresponding step (state). A step may have zero or more transfer or connection statements, followed by a conditional or unconditional branch statements. The destination of a transfer statement is always a memory element and that of the connection is non-memory like a *bus* or a set of output lines. A connection is active *throughout* the corresponding control step whereas a transfer is assumed to take place only at *trailing edge* of the clock in that step. The step numbers and branch statements define the sequencer of the control unit of the module. The non-procedural part follows the keyword ENDSEQUENCE. Statements in this part are always active regardless of the state of the control sequencer. Permanent connections between elements are to be described in this segment.

2.2.2 VHDL Overview

VHDL is a multipurpose hardware description language supporting several abstraction levels of design description, from the logic/gate level to the system/behavioral level. In VHDL the hardware objects are expressed using design constructs as either design entity, configuration, or package, with the internal code being declaration, specification, expressions, and statements.

The primary abstraction element in VHDL is the design entity since it represents the logic circuits from the very complex system to simple circuit units. The design entity consists of the interface description and one or more architectural bodies. The interface description specifies the entity name and defines the inputs and outputs of the entity design for the communication with the outside world. The architectural body gives the internal structure of the object as structural description, data flow description, a behavioral description, or a mixture of these.

The configuration construct is used to bind a named component to a specific architecture of a specific design entity. It is also useful to link subcomponents used in an architecture to lower level entity/architectures. The *package* concept originating from Ada provides a mechanism for encapsulating definitions and utility functions. The *package* contains a collections of declarations and programs that are frequently used. This construct (together with many others such as user defined data-types) is unique to VHDL among other HDLs.

The entities and component instantiations are primarily used for the structural decomposition of the system being modeled. While processes, subprograms, package, etc., are for the purpose of decomposing the behavior of the system.

In VHDL, sequential and concurrent circuit models are represented by '*process*' and '*block*' design elements respectively. Statements within a process are executed in sequence. However, each statement within a block is considered as a process, that is, all the statements of the block are executed whenever the block is allowed to run. The language has three classes of objects: constants, variables, and signals. Variables have no direct hardware correspondence whereas signals have equivalence in hardware. When a signal has more than one source it is defined with a resolution function to determine its correct source. VHDL supports a variety of data types and option for user-defined types. Finally VHDL supports the facility to maintain multiple design libraries to store user defined and system defined primitives and descriptions. It also allows the description of synchronous as well as asynchronous systems, and data transfers can be on the rising or falling edge of the clock.

2.2.3 AHPL versus VHDL

AHPL is simple and concise hardware description language. Beside being one of the smallest language, it provides a powerful capability to describe complex hardware design. AHPL has its limitation but it is good language to introduce for beginners to hardware design.

VHDL provides a foundation for the communities of design automation, design and test, and manufacturing to reduce product cost through improved designer productivity and maintenance capabilities. Being the standard hardware description language VHDL provides the designer and developers a common data interface among their tools. VHDL's rich features make it suitable not only as a descriptive medium but also as a design tool.

2.3 HDLs Analysis

In this section a short analysis of nine well known hardware description languages is presented. The analysis is based on the prominent features available by these nine hardware description languages. The nine selected hardware description languages are:

- IDL - Interactive Design Language
- TI-HDL - Texas Instrument Hardware Description Language
- CDL - Computer Design Language
- AHPL - A Hardware Programming Language
- ZEUS - Hardware Description Language
- CONLAN - A Consensus Language

- TEGAS - Test Generation and Simulation
- ISIS - Instruction Set Processor Specification
- VHDL - VHSIC Hardware Description Language

A detailed overview of AHPL and VHDL was presented in an earlier section. For brief outline of other HDLs, readers are encouraged to review [AWS86, HDL92a, HDL92b]. The comparison analysis for the nine HDLs was conducted on the basis of the following seven criteria:

- *Scope - the range of hardware*: the language support to describe and design at different levels.
- *Management of design*: support for different design methodologies, design abstraction and resuability.
- *Timing description*: capability to include timing mechanism in a design hierarchy and abstraction.
- *Architecture description*: capability to describe at different levels of design details. Also the control structures for statement execution.
- *Description of a design interface*: availability of a mechanism for defining external interface to design description.

- *Description of a design environment*: support for design tools and ability for environment definition.

- *Language extensibility*: flexibility for user-defined extension and support for future advances.

The comparison analysis of the nine HDLs is summarized in Table 2.1. In short all evaluated languages at least support gate-level design and synchronous sequential design. Most of the languages support hierarchy, modularity and libraries. Architectural description is possible in most of the languages. All of the languages can support more than one technology and more than half support multiple methodologies.

All these excellent hardware description languages in industry today cover various aspects of hardware design and description, but none matches VHDL's capabilities. The analysis has shown that no current hardware language capabilities, shortcomings and other characteristics had been overlooked in developing VHDL. VHDL has proven to be modern language that is complete and comprehensive. Furthermore, all the salient features of AHPL are available in VHDL, thus AHPL in terms of features forms a subset of VHDL.

Features Supported	IDL	TI-HDL	CDL	AHPL	ZEUS	CON-LAN	TE-GAS	ISPS	VHDL
Scope-range of hardware:									
Digital-system design	x	x		x	x	x	x	x	x
Gate-level design	x	x	x	x	x	x	x	x	x
Combinational design		x		x		x	x	x	x
Synchronous design	x	x	x	x	x	x	x	x	x
Asynchronous design		x				x	x	x	x
Mixed-mode		x					x		x
Management of design:									
Hierarchy	x	x		x	x	x	x	x	x
Modularity	x	x		x	x	x	x	x	x
Incremental compile		x			x				x
Libraries	x	x		x	x	x	x	x	x
Data abstraction					x	x			x
User type conv									x
Alternate description									x
Reusable design	x	x		x	x	x	x	x	x
Timing description:									
Timing at all levels	x	x		x	x	x	x	x	x
Specify timing data		x		x		x	x	x	x
User-defined data		x		x		x	x		x
Timing constraint		x					x		x
Propagation delay	x	x		x		x	x	x	x
Architectural description:									
Algorithmic description	x	x		x	x	x	x	x	x
Architecture description	x		x	x	x	x	x	x	x
Parallelism	x		x	x	x	x	x	x	x
User assertions						x		x	x
Generic components		x			x	x	x		x
Recursive structures				x	x	x			
Interface description:									
Explicit interface		x		x	x	x	x	x	x
Strongly typed interface					x	x			x
Design environment:									
Environment information								x	x
Language extensibility:									
User-defined data types					x	x			x
Design tool support						x		x	x
Multiple technologies	x	x	x	x	x	x	x	x	x
Multiple methodologies	x	x		x		x		x	x

Table 2.1: Comparison Analysis of HDLs.

2.4 Summary

This chapter presented introductory material of hardware description languages. The intention of the first part of the chapter was to give an overall understanding of the concept of hardware description language and their use in design environment. The second part of the chapter presented two hardware description languages, namely AHPL and VHDL. A brief overview of these languages and their many salient features were highlighted. In the last part nine well-known HDL language were selected and a comparison analysis was summarized.

Chapter 3

Translation of AHPL models to VHDL

The intent of this chapter is to provide a general knowledge to understand the *composition process*. The first part of the chapter describes the translation approach that was adopted in order to analyze the AHPL specification and to generate its corresponding equivalent VHDL model. The remaining part of this chapter is devoted to present and describe the VHDL construct that are available for the purpose of composing VHDL model. A comprehensive study was conducted to find the most suitable set of constructs that will best describe the AHPL specification.

In the next chapter the *composition process* is presented using the concept learnt from this chapter.

3.1 Translation Approach

AHPL is based on the fact that any digital system can be partitioned into a *data part* and a *control part* as shown in Figure 3.1. The data part consists of registers, buses, CLUs and some basic gates. The control part consists of logic which provides signals to control the operations in the data part. Additionally, the sequencing of control is influenced by the branching information fed back from the data part. Generally, the major role of the control part consists of a schedule defining each operation in the data part and determining the ordering and timing in which these operation take place. Similar to other HDLs, AHPL has its own set of conventions for transfers, connections, register indexing, etc.

VHDL does not require separation between the data flow and the control flow. If the designer/modeler desires to separate between data and control, he must explicitly incorporate control mechanism in the design model. A convenient method is by including *guard* in all the statements (a guarded statement is executed when the guard condition is true while all non-guarded statements as well as blocks whose guards are true are executed in parallel).

Conventionally, in AHPL, all transfers into registers take place at the trailing edge of the clock pulse. Also, transitions between states of the finite state machine take place at the trailing edge. However, transfers to buses or input/output lines, called *connections*, are active for the entire duration of the clock pulse.

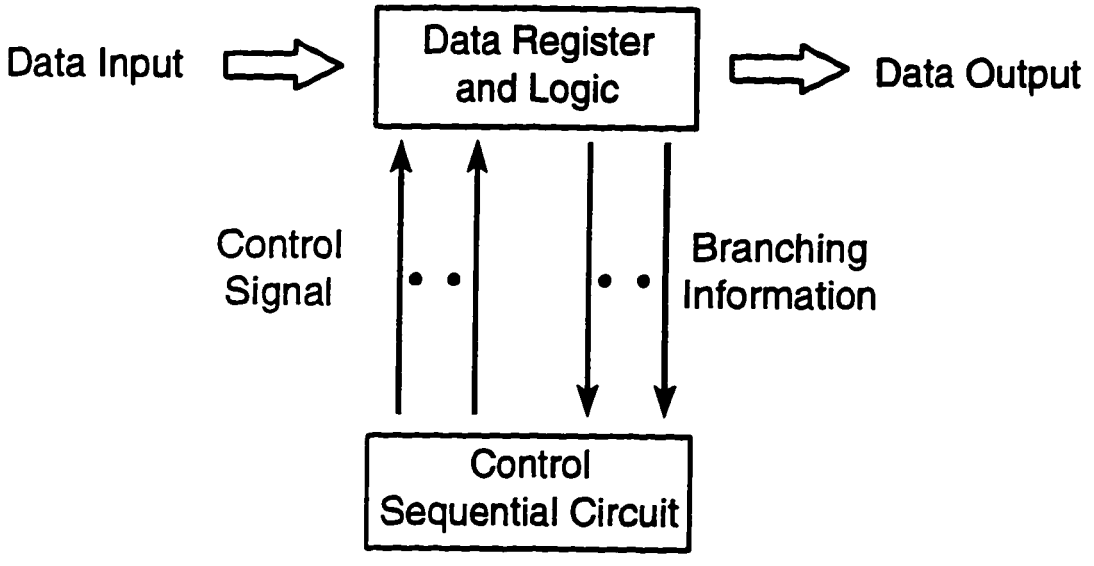

Figure 3.1: Data and Control Parts.

In the following sections a detail survey is made on some of the constructs available in VHDL. These constructs are introduced, with the justification that the essential features of AHPL can be mapped to its equivalent VHDL model. The realization of some of the important basic AHPL elements and features with their possible counter-parts is summarized at end of this chapter.

3.2 Mapping Concept

In this section several concepts are presented that are needed to be understood for purpose of realization of VHDL model from the AHPL model. These ideas are used in the building of the *Analyzer* and the *Composer* that are described later in the next chapter. Also included is a description of some of VHDL constructs that are required for the accomplishment of the task of translation.

3.2.1 Declarations

In VHDL, many features are available to facilitate the description of components and systems. A digital device in VHDL is represented as a *design entity.* In its simplest form, a *design entity* consist of two parts:

- *an entity declaration*
- *an architecture body*

As shown in Figure 3.2, the entity declaration begins with the keyword **entity** and defines the interface to describes the inputs/outputs (I/O) ports of the component, that is, what the outside world sees and communicates. Ports define communication channels between design entities and the outside world. A port definition involves description of its mode and type. The port's mode specifies the direction of information flow through the port. A port type specifies the set of values a port may assume.

The declarations of INPUTS, OUTPUTS, EXINPUTS, EXOUTPUTS and EXBUSES in AHPL module are taken care of by the interface description of the entity. These are declared as ports along with associated input or output mode. Registers declared as MEMORY in AHPL, and internal BUSES, are declared as *signals* inside the section that models the architecture. Table 3.1 gives the summary of how AHPL declaration elements are realized in VHDL.

Other external characteristics of a component, such as timing dependencies, can also be included in the entity interface of the component. The name of the component comes after the keyword **entity** and is followed by keyword **is**.

As shown in Figure 3.3, an architecture specification begins with the keyword **architecture**, which describe a possible internal functionality of the entity. This functionality depends on the input-output signals and other parameters that are specified in the entity interface declaration. A functional description of the component starts after the keyword **begin**.

```
entity component_name is
  input and output ports.
  physical and other parameters.
end component_name;
```

Figure 3.2: Entity Interface Declaration.

AHPL Element	VHDL Object
Input	Port in entity interface *in* mode
Output	Port in entity interface *out* mode
Exinput	Port in entity interface *in* mode
Exoutput	Port in entity interface *out* mode
Exbuses	Port in entity interface *in* mode
Memory	Signal in architecture body
Buses	Signal in architecture body

Table 3.1: AHPL Declaration Elements and their VHDL Counter-parts.

architecture identifier **of** component_name **is**
declarations.
begin
specification of the functionality of the component in terms of its input lines and as influenced by physical and other parameters
end identifier;

Figure 3.3: Architectural Specification.

In VHDL several architectural specifications with different identifiers can exist for one component with a given entity interface declaration. An architecture description could have three general styles of description: structural, dataflow, and behavioral or an unrestricted combination of all three. Structural description captures the schematic view of hardware and consist primarily of interconnected components. Dataflow description, a little more abstract, specifies data transform being performed in terms of concurrently executing RTL statements. Behavioral description, the most abstract, specifies data transforms in terms of algorithms for composing output responses to input changes. In this work the composed VHDL description that is produced is at the dataflow/RTL level, which is at the same level as the input AHPL description.

In VHDL, separation of the entity external interface from the internal architecture is a convenient feature. The entity structure can be viewed as describing the *black box property* and the architecture body as the *glass box property.*

In the declaration of *signals* in the architecture body it is often required to be declared as *resolved signal.* A signal which has more than one source is called a *resolved signal.* In such cases *resolved signal* must have a resolution function associated with it to resolve multiple sources into a single value for the signal. The resolution function is invoked every time the value of the resolved signal is updated. If two sources are driving the signal, then the resolution function is invoked with array of length two which contains the values of the two sources. Based on the

definition of the resolution, the returned result value is obtained by tying together the input source values. The resolution function used for the implementation of the *Composer* is a simple Wired_Or function which collects the values sent from all the input sources and applies the OR function to generate a single value.

3.2.2 Processes

The *process* construct of VHDL represents the fundamental method by which concurrent activities of a digital system are modeled. All processes are executed in parallel. A process statement has a declarative and statement parts. All the variables and constant objects are declared in the declaration part and these objects are initialized only once at the beginning of a simulation run. The statement part of a process is sequential and is always active. Each process statement defines a specific action, or behavior, to be performed. This behavior is defined by sequentially ordered execution statements in the process.

Statements in a process continue to execute until they are suspended. Once suspended, a process can be reactivated. One way a process can be reactivated is by designating a maximum time for the process to remain suspended. A common mechanism for suspending and subsequently conditionally activating a process is the use of *sensitivity list*. Following the keyword **process**, a list of signals in parentheses can be specified. This list is called *sensitivity list*, and the process is activated when an event occurs on any of these signals. When the program flow reaches the last

sequential statement, the process becomes suspended, although alive, until another event occurs on a signal that it is sensitive to. Figure 3.4 shows the example of a process that is controlled implicitly by the *sensitivity list.* This process, named Or_Process, has a sensitivity list of two signals, in1 and in2. Whenever any change of state happens for any of these signals, the process is activated, and the assignments in the statement part of the process are executed. In this case the output signal will obtain the result of the 'or' of the values of the signal in1 and in2.

Alternatively, the activation and suspension of the process can be controlled by a single construct called the *wait* statement. When the *wait* statement is executed inside a process, the process suspends and the conditions for its reactivation are set. There are three different kinds of conditions: *timeout*, *condition*, and *signal sensitivity*, and these kinds of conditions can be mixed together in the wait statement. In the *timeout* form, a maximum delay for the process to be suspended is defined and when the delay expires, the process is reactivated. In the second form, a *condition* must be true before the process can be resumed. The final form provides a list of signals which is similar to the *sensitivity list.* Whenever an event occurs on a signal to which the process is sensitive, the process is resumed. Figure 3.5 shows the example of a process that is controlled explicitly by the *wait* statement. In this example, the process statement contains one signal assignment statement and is followed by the *wait* statement. The signal assignment statement specifies that the signal output will obtain the result of 'or' of the values of the signal in1

```
Or_Process:
  process(in1,in2)
  begin
    output <= in1 or in2;
  end process;
```

Figure 3.4: Process Statement with Sensitivity List.

and in2. The wait statement suspends the process until there is an event on either of the signals in1 or in2, at which time execution resumes at the top of the process statement.

In summary, a process is always active but the statements within a process are executed in sequence, one after the other. Although processes have parallel with hardware, and provide a natural way of modeling it, they are not best suitable for purpose of defining AHPL statement actions. In AHPL, all register transfer and bus connections are performed concurrently in a single step whereas in the process all statement are done sequentially. A better VHDL construct is the *block* statement that is described in the next section.

3.2.3 Blocks

Similar to many programming languages VHDL provides a partitioning mechanism that allows the designer to logically group areas of the model. A *block statement* starting with keyword **block** and ending with keyword **end**, are used to group part of a design. A block can only be used within the architecture in which it has been designed and the statements within the block are executed concurrently.

Block structures are self-contained regions and each block may have a declarative part, which comes between the keywords **block** and **begin**. The declarative part may declare local signals, types, constants, etc. It may also define the interface to the block by using the **port** interface list.

```
Or_Process:
  process
  begin
    output <= in1 or in2;
    wait on in1, in2;
  end process;
```

Figure 3.5: Process Statement with Wait Statement.

Block statements have an interesting feature known as *guarded blocks*. Guards in VHDL provide a facility to control the operation of signal assignment statement within the block. These signal assignment statements are known as guarded signal assignment and are recognized by the keyword **guarded** between the <= and the expression part of the statement.

A guarded block contains a guard expression after the keyword **block**. When the guard expression is true, all of the guarded signal assignment statements are enabled, or turned on. When the guard expression is false, all of the guarded signal assignment are disabled, or turned off. Whenever, a block has a guard expression, the VHDL compiler also implicitly defines a signal variable with the name *guard*, which can be used inside the block to trigger other processes to occur. This signal is read only and cannot be updated. Figure 3.6 illustrates the latch model of a *D_flipflop* using the guarded block. There are two guarded signal assignment statements in this model. One is the statement that assigns a value to q, and the other is the statement that assigns a value to qb. When the *clk* signal has a value '1', the guard expression will be true, and the two guarded signal assignment statements are concurrently executed after their respective delay.

In summary blocks are very useful for partitioning the design into smaller and more manageable units. They allow the designer the flexibility to create large designs from smaller building blocks and provide a convenient method of controlling the execution of signal assignment statement. Based on this fact the VHDL block

```
d_flipflop: block(clk='1')
begin
    q <= guarded d after delay1
    qb <= guarded not(d) after delay2
end block d_flipflop;
```

Figure 3.6: Model of D_flipflop Using Guarded Block.

construct is best suitable for defining register transfers and bus connections in a AHPL step statement especially as each statement is executed concurrently within the block. Each transfer action can be translated as *guarded signal assignment statement* and the implicit *guard* is used to control the operation of signal assignment statement. The modeling of the AHPL register transfers and bus connections are described later in this chapter.

3.2.4 State Machine

To model the finite state machine of the digital system, a mechanism that represents the *state* of the system is required. The current state of the control sequencer is represented by a binary vector of size equal to the number of steps in the AHPL module is used. In sequential systems all but one bit of this vector are low. However, for a parallel machine, depending on the number of concurrent activities (states), more than one bit may be high. The index of the high bit identifies the current state of the machine.

3.2.5 Modeling Trailing Edge Transfers

VHDL provides several options to model negative edge triggering. If the state is modeled as a process, one possibility is to use a *wait* statement at the beginning of the process. The statement `wait until (clk='0')` when included at the beginning of the process ensures that the process is executed only when the clock changes, and

that the change is from '1' to '0', representing the trailing edge. VHDL does not allow simultaneous use of a sensitivity list and wait statements. The reason is because that might create a contradiction within a process at execution time. Therefore we cannot put the clock signal in the sensitivity list of the process and have a wait on clock statement inside the process. Modeling a state as a process is not feasible since a state generally has more than one data transfer, which must be executed in parallel. The correct representation in VHDL of these concurrent transfers is via the *block* construct. To make the execution of the transfers within the block happen at the trailing edge, it is sufficient to make the block guarded on a trailing edge condition. This is the solution adopted in this work. Figure 3.7 shows how trailing edge condition can be implemented in VHDL.

As mentioned in the last section, the guard will control execution of all the statements within the block. The guard condition `clock='0' and not clock'stable` assures that all the guarded assignment statements will be executed immediately when the clock changes its state from '1' to '0' i.e., the trailing edge condition. The attribute property *stable* is used to ensure that the clock value is stable for the guard to become true.

3.2.6 Modeling Transfers to Buses

Since transfers to buses in a state are also executed in parallel, the model is similar to the one above, except that the trailing edge condition is removed from the guard

```
trailing_edge: block(clock='0' and not clock'stable)
begin
    ...
    ...
    ...
end block trailing_edge.;
```

Figure 3.7: Trailing Edge Implementation.

of the block. This method allows the bus connection to be made any time during the clock high edge, (i.e., when that particular state is active).

3.2.7 Modeling Conditional Transfers

In AHPL, there are three possible forms of a clocked transfer and two forms of a connection statement. In a simplest clocked transfer there is only one single source which is merely transferred into the destination register. The remaining two form conditional transfers, namely *source control* and *destination control.* Similarly, bus connection is either of simple connection or *source control.*

The *source control* can be easily implemented by appending a series of *when* and *else* clause at the end of the signal assignment statement for each of multiple sources. For each of the possible sources, its condition could be appended at the end of its *when* clause. The transfer or the connection will only be executed when its conditional statement is true. On the other hand, the *destination control* is best implemented by use of the *if* structure. The *if* statement is used to select destination signal to be updated based on the truth of the condition. Unfortunately, the *if* is not appropriate as the implementation of the *composition process* uses *block* structure to represent actions part for each state and the *if* is a sequential statement whilst the *block* structure only allow concurrent statements. However, *block* structure does allow the use of *process* statement which could include the *if* statement. The use of *process* and *if* statement is awkward method to implement *destination control.* A

better solution is to create a nested *block* with its guard used to condition for the destination control condition.

3.3 Mapping Summary

In this chapter a detail study of VHDL features was conducted and proposal was presented on how declarations, state automaton, edge triggered transfers, and other processes in AHPL can be mapped to VHDL. Modeling styles for the trailing edge and, the register transfer and bus connection were proposed using a subset of VHDL constructs. Ideally the *process* construct seems sufficient to model AHPL features but as revealed it is not the most suitable. The VHDL *guarded blocks* provide a very convenient method to model various aspect of AHPL specification.

Table 3.2 summarizes some of the important basic AHPL elements and features and their possible counter-parts.

AHPL	VHDL
AHPL Module	*entity*
CLUs	*components*
MEMORY	*signals* in the architecture body
BUSES	*signals* in the architecture body
INPUT lines	*port* in entity declaration
OUTPUT lines	*port* in entity declaration
Concept of *state*	*variable bit_vector*
Trailing edge transition	*process* with '*wait until(clock='0')*' or *Guarded blocks* with trailing edge condition included in the *guard*
Connections to BUSES	*block* of statements without guard of edge condition
Transfer to Registers	*block* of statements with guard of edge condition
State Transitions	*process* with *wait* or *guarded block*
',' (catenation)	&
Reduction operator	VHDL predefined function
Source Control	Nested *when or else* clause
Destination Control	Nested *block* structure

Table 3.2: Basic AHPL Modeling Elements/Features and their VHDL Counterparts.

Chapter 4

The Composition Process

In the last chapter, VHDL language was examined and a translation approach was defined. Based on the translation approach a mapping scheme from AHPL description to VHDL was derived using a subset of VHDL constructs. In this chapter, the mapping concept defined in the last chapter are utilized in the development of the *composition process* that allows the translation of the AHPL specification to its functionally equivalent VHDL model. Along with the overview of the *composition process* a detail description is also given for all its components.

The chapter ends with an illustrative example demonstrating the *composition process.* During the discussion of the components of the *composition process*, parts of an example were used to show the intermediatory results that are generated before the final VHDL model is produced.

4.1 Composition Overview

In this section a general overview is presented of the proposed *composition process.* A brief overview is also given for the different components of the *composition process* and these components are discussed in more detail in the later sections.

The *composition process* is a method of composing the AHPL specification and generating its corresponding functionally equivalent VHDL model. It is performed in two stages using several pre-defined composition tools. The block diagram of the *composition process* is given in Figure 4.1.

In the first stage, the *Analyzer* performs the syntax check for the input AHPL specification and produce intermediatory output for the *Composer.* Initially, the design specification written in AHPL is analyzed to check for syntactic errors. The *Analyzer* scans the AHPL code line by line and reports of any syntax format that is not allowed. Although the AHPL code may pass the syntactic correctness but there may still remain some semantic errors which are not detected by the *Analyzer.* It is left up to the user to make sure that no semantic faults exist in the input AHPL code.

If the description is syntactically correct, the *Analyzer* produces the required data structures of the data and control paths of the design. The generated data structures undergo a series of transformations to produce a concise tabular data structure capturing both the data and control (finite state machine) paths.

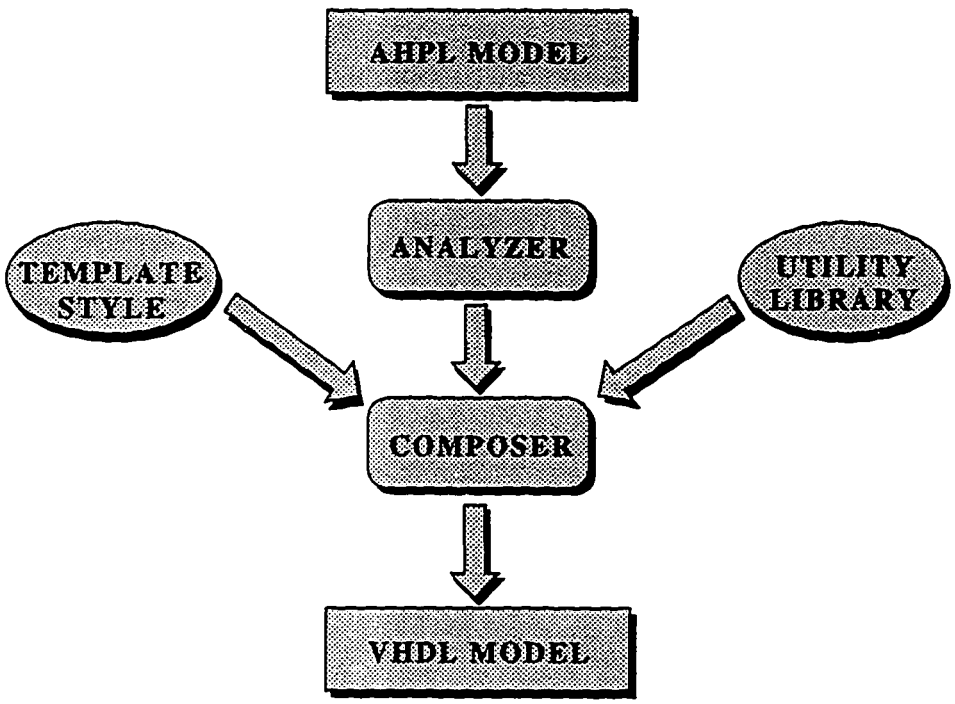

Figure 4.1: The Composition Process Steps.

In the second stage the *Composer* utilizes the output of the *Analyzer* with the especially designed composition tools to generate the required VHDL output model [LG89]. There are two composition tools, The VHDL Template and The Utility Library, that were especially developed in order to simplify the *composition process.* The template structure is described later in this chapter. The Utility Library being used by the *Composer* contains predefined utilities that assist in the translation of many AHPL feature and functions.

In the next few sections the components of the composition process are described in more details. The chapter concludes with an example of generated VHDL model for *4-bit Multiplier* example of Figure 2.2.

4.2 The Analyzer

The primary role of the *Analyzer* is to diagnose and analyze the AHPL specification for the control and data path and accordingly generate FSM. It is also used to check for the syntactic correctness of AHPL code. Figure 4.2 shows the block diagram of the *Analyzer.*

The *Analyzer* performs its task in two steps. In the first step, the AHPL code is passed through the *Syntactic Checker* for the verification of the syntax. The AHPL input specification is fed in to the *Syntax Checker* and output is forwarded to *Control & Data Path Generator.* For *Syntax Checker* to perform for syntax correctness, the

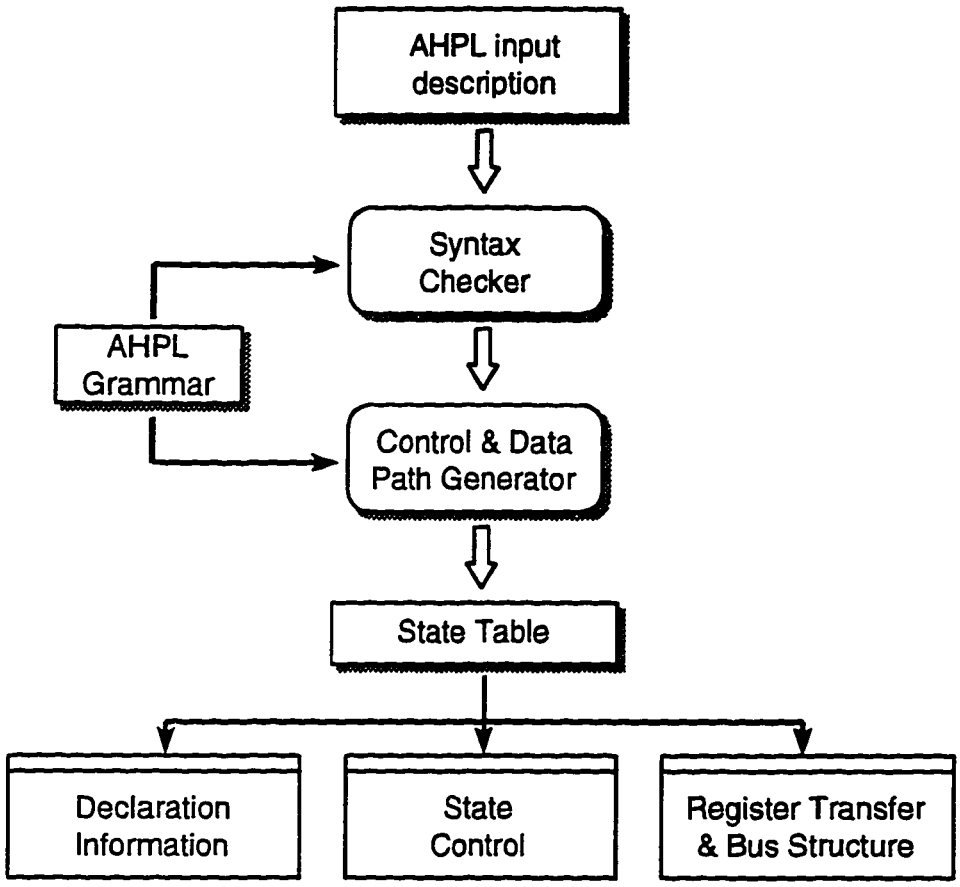

Figure 4.2: The Analyzer Block Diagram.

AHPL grammar is also provided.

In the second step the *FSM Control & Data Path Generator* produces state tables. The *Analyzer* studies the declaration part, procedural part, and the non-procedural part of the AHPL description sequentially and generates comprehensive reports that fully describe the control actions of the AHPL specification. These reports are in the form of three tables:

- Declaration Information.
- State Control.
- Register Transfers & Bus Structures.

For the purpose of demonstration an extended-FSM is produced to show the information generated by the *Analyzer*. The FSM diagram for the AHPL description of Figure 2.2 is shown in Figure 4.3. The actual output produced by the *Analyzer* are three state tables as shown in Tables 4.1, 4.2 and 4.3.

4.2.1 The Analyzer Implementation

The *Analyzer* was built using classic UNIX compiler tools, namely `lex` and `yacc` [LMB92]. These tools were chosen as they provide powerful and flexible programming development environment to represent AHPL grammar. AHPL is a context-free grammar in Backus-Naur form and can be easily be encoded using the `lex` and

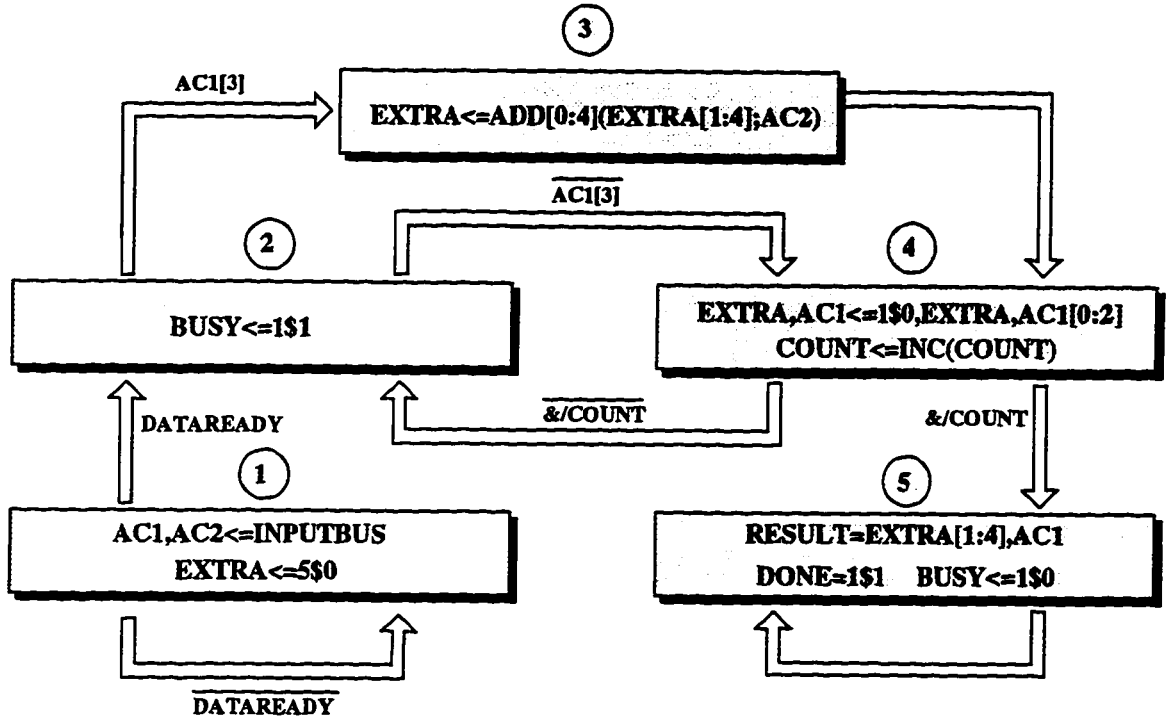

Figure 4.3: FSM Diagram of AHPL Model.

name	type	length	source_cnt
ac1	memory	4	2
ac2	memory	4	1
count	memory	2	1
extra	memory	5	3
busy	memory	1	2
dataready	exinput	1	0
clock	exinput	1	0
reset	exinput	1	0
inputbus	exbus	8	0
result	output	8	1
done	output	1	1
busyout	output	1	1
inc	clu=incr	2	0
add	clu=adder	5	0

Table 4.1: Declaration Information Table.

current_state	next_state	tran_cond
1	1	dataready='0'
1	2	
2	4	ac(3)='0'
2	3	
3	4	
4	2	and_red(count)='0'
4	5	
5	5	

Table 4.2: State Control Table.

state	type	dest_vect	dest_cond	sour_vect	sour_cond
1	register	ac1		inputbus(0 to 3)	
1	register	ac2		inputbus(4 to 7)	
1	register	extra		"00000"	
2	register	busy		'1'	
3	register	extra		add1_out(0 to 4)	
4	register	extra		0 & extra(0 to 3)	
4	register	ac1		extra(4) & ac1(0 to 2)	
4	register	count		inc1_out	
5	bus	result		extra(1 to 4) & ac1	
5	bus	done		'1'	
5	register	busy		'0'	

Table 4.3: Register & Bus Transfers Structure Table.

yacc tools. The yacc tool was used to generate the parser from the grammatical description of the AHPL language, while the lex tool was used for making lexical analyzer. The yacc tool in general 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 also be 'evaluated'.

The development of the *Analyzer* was performed in two steps. In the first step, the yacc tool was used for the specification of each rule or *production* of the AHPL grammar. For each rule, an action clause can be augmented – statement of what to do when an instance of that grammatical form is found in the AHPL code being parsed. This 'what to do' was written in C, with conventions for connecting the grammar to the C code. This part was used to define the semantic for the *Control & Data Path Generator* for purpose of extracting information for the state tables. In the second step, a lexical analyzer was created to read the input AHPL code being parsed and break it up into meaningful chunks for the parser. These lexical chunks are known as *tokens* and are used for defining of the keywords and symbols of the AHPL language. For each keyword and symbol of AHPL language recognized by the lexical analyzer a matching token was returned to the parser. There are two special types of tokens that are returned to parser: one for the string and one for the number. For both of these types the name corresponding to the token is return along with its value.

The complete lex and yacc programs for the *Analyzer* could be found in the

appendices. Figures 4.4 and 4.5 show a part of `lex` code and `yacc` production rule respectively. As can be seen in Figure 4.5, a `yacc` production rule comprises of two part. In the first part the definition of the grammar rule is stated and this is appended with an optional action part. The selected sample shows the production rule for the recognition of the 'module' head part of an AHPL code. The rule is looking for the keyword MODULE followed by the symbol ':' and an id string representing the name of the module. If this rule is satisfied, the action part of the production rule is executed and in this case it is simply printing the name of the module. For the `yacc` program to successfully match the different tokens, the lexical analyzer is intermediately invoked. For this particular production rule the three tokens are found by using the three lexical rules shown in Figure 4.4. The lexical rules are simply represented using the regular expression and if a match occurs its action part is executed. In all cases the name of the token is returned. For the tokens ID and CID its string value is also returned in the global variable id_val and for token INTEGER its integer value is returned in the global variable int_val.

4.2.2 State Table Generation

The `lex` and `yacc` tools used for the implementation of the *Analyzer* are initially used to perform the syntax check for each line of the AHPL code. The verified statements are then analyzed to extract all the information for the generation of the

```
id                  [a-zA-Z][a-zA-Z0-9]*
                    ⋮
%%
module|MODULE       { return MODULE; }
":"                 { return COLON; }
id                  { strcpy(id_val,yytext);
                      return ID; }
```

Figure 4.4: Sample Lex Code.

```
modhead     : MODULE COLON ID
                { printf("\nModule Name: %s\n",id_val); }
            ;
```

Figure 4.5: Sample Yacc Code.

state tables. Basically, the *Analyzer* breaks up the AHPL statements into smaller chunks and then recognizes the predefined structures. These structures are studied and the information extracted is used for the generation of state tables. These state tables are stored into specially design data structures tables.

The recognition and the extraction of the information are performed by the production rules of the yacc program. The yacc rules first breaks up each line of the AHPL code read and then recognizes the required pattern of tokens. Once this is achieved the action part of the rule uses the broken up elements to extract the required information. There are three general types of information that are needed for the extraction. The first type is the declaration information of the AHPL code; the second type for the state control and; the third type of information comprises register transfers and bus connections. These information are tabulated as shown in Tables 4.1, 4.2 and 4.3.

An AHPL module in general can be partitioned into three parts: the module head, declaration part and the module sequential part. The module sequential part can further be divided in two separate sections: procedural part and the non-procedural part. The *Analyzer* performs the scanning of the AHPL code in sequence for the module head, declaration part, and finally the procedural and non-procedural parts. For the module head, the *Analyzer* simply checks if it exists in the correct format and then extracts the name of the module. This name is used for the naming of its equivalent entity in VHDL model. For the declaration part, all the declared

elements are recorded along with its properties. As can be seen in Table 4.1, the name of each element is stored along with its type and length. Also recorded is the number of times an element is updated from different sources in the module sequential part. Initially, when scanning the declaration part, only the element name, type, and its length are recorded. The source count is updated later when the the module sequential part analysis is complete. The source count is simply calculated by counting the number of sources for each element in the Register & Bus Transfer Structure Table. For declaration elements of type INPUT, EXINPUT, EXBUSES and the CLU, the source count is by default 0, as no updates are allowed for these elements. This source count is needed for the purpose of determining which elements of the AHPL declaration need to be declared as resolved signals in the VHDL model.

The information extraction for control and data paths are performed simultaneously as the module sequential part is scanned. The *Analyzer*, for each step of the AHPL code in the procedural part, records the current state (or the current step number) and the next possible state(s). This state transition information is stored in State Control Table as shown in Table 4.2. The State Table contains all permissible next states for each state scanned. Also included for each transition any associated transition condition that is required. The *Analyzer* also assumes one of the next possible state for a step is the next higher step number. This assumption is implicit in the AHPL language and is also taken care of by the *Analyzer*.

For each step scanned in the procedural part, the *Analyzer* searches for all the register transfer and bus connection statements. For both of these transfers the basic information recorded is the state number in which transfer is made, type of the transfer (register or bus), the destination vector, and the source vector. For the destination vector and the source vector, any associated condition are also recorded respectively. The compiled information is stored in the Register & Bus Transfer Structure Table as shown in Table 4.3. The source vector expression extracted from AHPL are transformed into VHDL style expression format before storage. Although the yacc program does not include the transformer for the AHPL expression to VHDL expression, a simple routine could be built to do the required expression translation.

In summary, the *Analyzer* plays an important role of examining the AHPL code to extract and generate the control and the data paths of an AHPL specification. In addition, the *Analyzer* is also able to do the task of syntactic check. The semantic check is not performed by the *Analyzer* but this feature could be implemented. The *Analyzer* could be enhanced by embedding semantic check rules in the action part of the grammar production rules in the yacc program.

4.3 The Composer

The objective of the *Composer* is to use the state tables produced by the *Analyzer* to generate the final VHDL code. Basically, the *Analyzer* diagnoses the AHPL code and produces its break-up summaries. The *Composer* uses these summarized information to build the VHDL model. This task is accomplished by the use of composition tools that are described later in this chapter. Figure 4.6 shows the block diagram of the *Composer*. The tasks of the *Composer* is divided into three modules:

- *Declaration Transformer*
- *Controller Generator*
- *Data Path Builder*

In general, the *Composer* is a simple formator which produces output in a specified format. The template style defines the format for output VHDL. The modules of the *Composer* could easily be implemented using the *print* command of any high-level language.

In the following sub-section a detail description is presented of the three task of *Composer*. Also included for each module is the algorithm that may be built for its implementation.

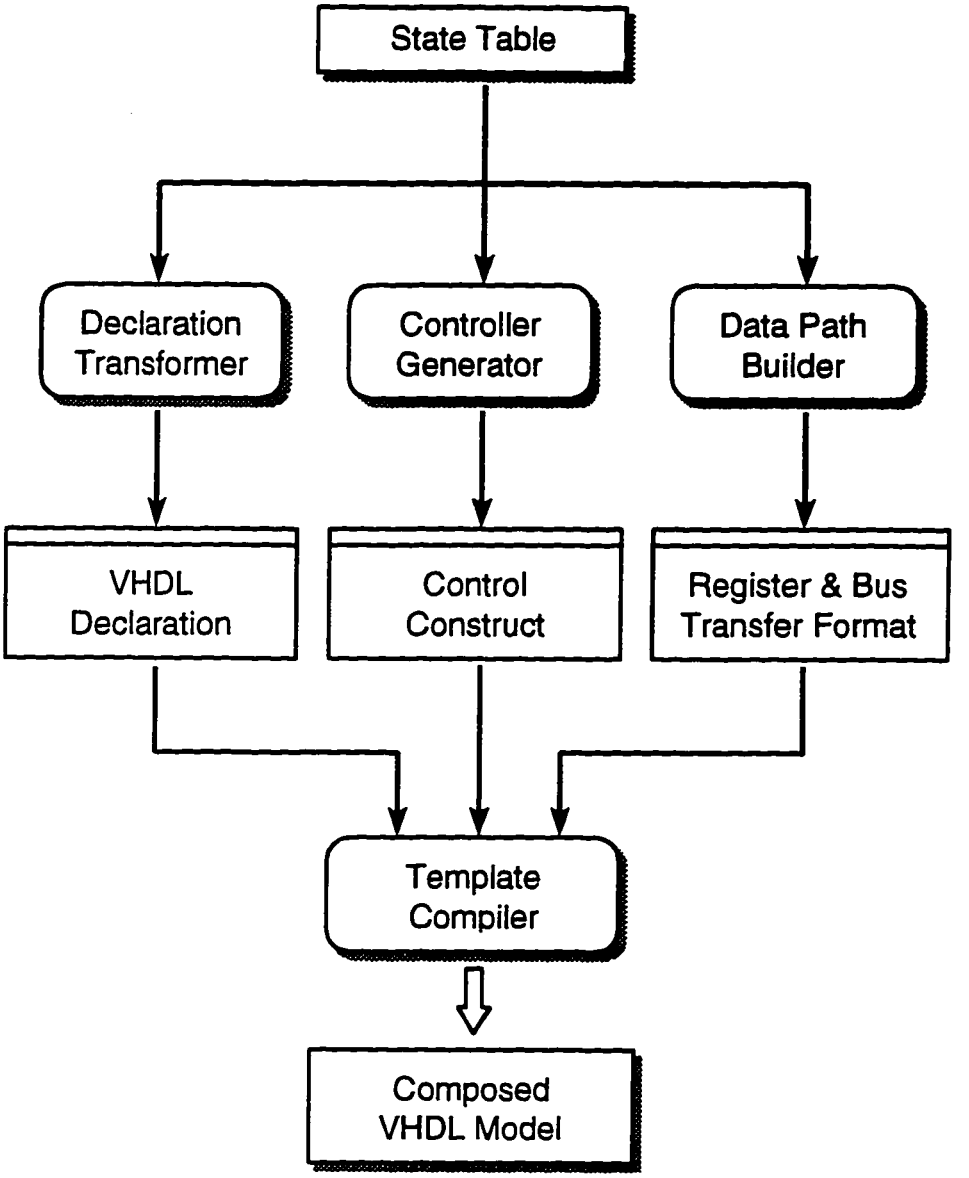

Figure 4.6: The Composer Block Diagram.

4.3.1 The Declaration Transformer

This is a simple module that uses the Declaration Information Table to produce corresponding VHDL declaration constructs. The declaration transformer reads the Declaration Information Table and based on its nature creates its translated VHDL format in the appropriate section of the VHDL model. Figure 4.7 show the algorithm of the declaration transformer.

The declaration transformer starts by searching for all the variable names of types INPUT, OUTPUT, EXINPUTS, EXOUTPUTS and EXBUSES. For each of these names, a corresponding declaration is written in the entity interface declaration section. These are declared as ports with their respective input or output mode. The modes allowable for the types OUTPUT and EXOUTPUT is *out* and, for types INPUT, EXINPUT and EXBUSES is *in*. Also for each of these AHPL declaration elements the length field of the Declaration Information Table is used to select the VHDL type needed to complete its declaration in the port format. These types are predefined in the Utility Library and are simply based on binary bit vectors of different lengths. The field source_cnt is ignored for these five types as the allowable source_cnt is 1 for types OUTPUT and EXOUTPUT, and, 0 for type INPUT, EXINPUT and EXBUSES.

For the remaining two types, MEMORY and BUSES, their declaration format is created in the declaration part of the architecture body. These two types are

```
Algorithm Declaration_Transformer

BEGIN
    infile=FILE("Declaration Information Table")
    WHILE NOT(EOF(infile))
        dec_rec=READ(infile)
        dec_type="mem_vec_"+dec_rec.length
        CASE dec_rec.type=(input,output,exinput,exouput,exbuses)
            IF dec_rec.type=(output,exouput)
                mode="out"
            ELSE
                mode="in"
            ENDIF
            /* Port Declaration */
            make_port_stm(dec_rec.name,mode,dec_type)
        CASE dec_rec.type=(memory,bus)
            IF dec_rec.source_cnt > 1
                res_fun="resolution_fun_"+dec_rec.length
                /* Resolved Signal Declaration */
                make_ressig_stm(dec_rec.name,res_fun,dec_type)
                /* Disconnection Declaration */
                make_dis_stm(dec_rec.name,dec_type)
            ELSE
                /* Signal Declaration */
                make_sig_stm(dec_rec.name,dec_type)
        CASE dec_rec.type=(clu)
            /* Component Declaration */
            comp_dec_stm(dec_rec)
            /* Component Specification */
            comp_spe_stm(dec_rec)
            /* Component Instantiation */
            comp_ins_stm(dec_rec)
        ENDCASE
    ENDWHILE
END Declaration_Transformer
```

Figure 4.7: Algorithm for Declaration Transformer.

simply declared as signals after declaration of control signals. Similarly, for both of these types, the length field is used to determine VHDL type needed from the Utility Library. For AHPL declaration which has source_cnt more than 1, they are declared as resolved signals. For each of the resolved signal an associated resolution function is also assigned and disconnection specification for that resolved signal is also created. The disconnection specification is described later in this chapter with the description of the template.

The final part of the Declaration Transformer involves the mapping of the CLU declaration into the VHDL format. The CLU mapping requires three steps and these step are described later in Section 4.4.3.

4.3.2 The Controller Generator

The information in the State Control Table along with the template style are used by the Controller Generator to produce the control mechanism in the VHDL model. The role of this module is also quite simple and can be easily implemented. The main part for the Controller Generator is the creation of the control block and the generation of the control transfer statements for each of the data blocks. The control block in conjunction with the embedded control transfer statement in the data blocks fully define the control path of an AHPL specification. The details of the control mechanism is described in Section 4.4.1. The algorithm of the Controller Generator is presented in Figure 4.8.

Algorithm *Controller_Generator*

ARRAY sour_vect[max_sour],sour_cond[max_sour]

BEGIN
infile=**FILE**("State Control Table")
/* Control Signal Declaration */
state_cnt=Σ unique(infile.current_state)
state_type="state_vec_"+state_cnt
res_fun="state_resolution_"+state_cnt
make_sig_stm("active_state",state_type)
make_ressig_stm("next_state",res_fun,state_type)
/* Control Block Generation */
blk_name="control"
blk_head="(clock='0' and not clock'stable) or reset='1')"
make_blk(blk_name,blk_head)
/* Guarded Assignment Statements for Control Block */
sour_vect[1]="$10_1 \ldots 0_{state\_cnt-1}$"
sour_cond[1]="reset='1'"
sour_vect[2]="next_state"
sour_cond[2]=""
grd_assig_stm("control","active_state",2,sour_vect)
sour_vect[1]="$0_1 \ldots 0_{state\_cnt}$"
sour_cond[1]=""
grd_assig_stm("control","next_state",1,sour_vect)
/* Guarded Assignment Statements for Data Blocks */
FOR i=1 to state_cnt
src_lst={List of source for current_state=i}
sour_cnt=0
FOR each src_vec $\in$ src_lst) **DO**
sour_cnt=sour_cnt+1
sour_vect[sour_cnt]=src_vec.next_state
sour_cond[sour_cnt]=src_vec.tran_cond
ENDFOR
blk_name="step_"+i
grd_assig_stm(blk_name,"next_state",sour_cnt,sour_vect)
ENDFOR
END *Controller_Generator*

Figure 4.8: Algorithm for Controller Generator.

For the control block, the Controller Generator requires the creation of the special control signal: active_state and next_state. The active_state signal is used to keep track of all the steps (or states) that are currently running. On the other hand, the next_state is used for recording the states that may become active in the next clock cycle. Both of these signals are declared as binary vectors with length equal to the total number of steps in the AHPL code. The guarded control block is created and the guard is defined as the trailing edge condition of the clock.

For each data block, the Controller Generator scans the state control table and accordingly writes the VHDL statement for the updation of the next_state signal. The generator checks the tran_cond field for any transfer condition. The expression in this field is used to build the *when* clause of the guarded signal assignment. In the case when there is only one possible future active state, no *when* clause is required. It is possible that a state may have several next states and in such case the guarded signal assignment is appended with *when* and *else* clauses. In general case the value assigned to the next_state vector will have only one of its bit high. The template design style supports concurrent processing of more than one data blocks and in such a case more than one bit could be high. As special case for the DEADEND AHPL statement the next_state field in the State Control Table is value 0 and the next_state is assigned bit vector of all 0s.

4.3.3 The Data Path Builder

This module uses the register transfer and bus connection information stored in the Register & Bus Transfer Structure Table to build a series of the data blocks. The template style also specifies the design of the data block structure and is used in writing the data block in the specified format. The algorithm for the Data Path Builder is shown in Figure 4.9. A detail description of how data path mapping is achieved is presented in Section 4.4.2.

The Data Path Builder starts by creating a data block for each state. It scans the Register & Bus Transfer Structure Table for all the records for a given state. The task of this module is divided into two parts. In the first part, all the transfers with the type *bus* are read and their corresponding guarded assignment statements are generated. In the second part, all the *register* type records are read and its information is used to write its guarded assignment statements. These guarded statements are written inside a nested block with its own guard. The guard of the outer block is defined using the step condition and for the inner block the trailing edge condition.

For *source control* register transfer and bus connection, there exist multiple sources. In such a case a dest_vect may be updated with one or more possible values. When creating the guarded assignment statement the sourc_cond field is used to generate the *when* and *else* clause statement to determine which source

```
ALGORITHM Data_Path_Builder

ARRAY  sour_vect[max_sour],sour_cond[max_sour]

BEGIN
    infile=FILE("Register & Bus Transfer Structure Table")

    state_cnt=Σ unique("State Control Table".current_state)
    FOR i=1 to state_cnt
        blk_name="step_"+i
        blk_head="(active_state("+i+")='1' and reset='0')"
        make_blk(blk_name,blk_head)       /* Data Block generation for each step */
        reg_lst={List of records of infile.type="register" for infile.state=i}
        bus_lst={List of records of infile.type="bus" for infile.state=i}

        IF NOT(EMPTY(bus_lst))      /* Checking for any Bus Connections */
            FOR each new bus_trf ∈ bus_lst DO
                src_lst={List of sources for bus_trf.dest_vect}
                sour_cnt=0
                FOR each src_vec ∈ src_lst DO
                    sour_cnt=sour_cnt+1
                    sour_vect[sour_cnt]=src_vec.sour_vect
                    sour_cond[sour_cnt]=src_vec.sour_cond
                ENDFOR
                blk_name="step_"+i
                grd_assig_stm(blk_name,reg_trf.dest_vect,sour_cnt,sour_vect)
            ENDFOR
        ENDIF
```

Figure 4.9: Algorithm for Data Path Builder.

```
        IF NOT(EMPTY(reg_lst))      /* Checking for any Register Transfers */
            blk_name="edge_"+i
            blk_head="(guard and clock='0' and not clock'stable)"
            make_blk(blk_name,blk_head)
            FOR each new reg_trf ∈ reg_lst DO
                IF (EMPTY(reg_trf.dest_cond))      /* Regular Register Transfer */
                    src_lst={List of sources for reg_trf.dest_vect}
                    sour_cnt=0
                    FOR each src_vec ∈ src_lst DO
                        sour_cnt=sour_cnt+1
                        sour_vect[sour_cnt]=src_vec.sour_vect
                        sour_cond[sour_cnt]=src_vec.sour_cond
                    ENDFOR
                    blk_name="edge_"+i
                    grd_assig_stm(blk_name,reg_trf.dest_vect,sour_cnt,sour_vect)
                ELSE                /* Destination Control Register Transfer */
                    sour_cnt=1
                    sour_vect[sour_cnt]=reg_trf.sour_vect
                    sour_cond[sour_cnt]=""
                    blk_name="cond_"+i
                    blk_head="(guard and "+ref_trf.dest_cond+")"
                    make_blk(blk_name,blk_head)
                    grd_assig_stm(blk_name,reg_trf.dest_vect,sour_cnt,sour_vect)
                ENDIF
            ENDFOR
        ENDIF
    ENDFOR
END Data_Path_Builder
```

Figure 4.9: Algorithm for Data Path Builder (cont).

vector is used.

For *destination control* register transfer, there will exist a condition in the dest_cond field. This condition will determine whether the register transfer must take place or not. This type of conditional transfer could easily be implemented by creating a nested block of guarded assignment statements. The guard of this nested block simply is the dest_cond ANDed with guard signal of the outer block.

4.3.4 Template Compiler

In the final stage of the *Composer* the Template Compiler collects all the transformed information generated by the three *Composer* modules. It then assembles the information together in style of the template. All the gathered information is formatted in the correct VHDL syntax. The final output of the Template Compiler represents the equivalent representation of the AHPL specification.

4.4 VHDL Template

In this section the design style of the VHDL is presented. The template is the skeleton of a generic VHDL model. This skeleton is defined to achieve accurate translation without creating complex VHDL code [NS90]. Using the input AHPL specification, the skeleton is customized (augmented) with the necessary details to compose a complete VHDL description functionally equivalent to the original

AHPL description. The clocking mechanism is explicitly defined in the template. The description style partitions the hardware of each module into data and control sections. Each section is implemented by a VHDL *guarded block* which uses the clocking scheme as the guard.

VHDL has a very rich variety of constructs for various applications. Since the AHPL model is an RTL functional description of the design, only a subset of VHDL is sufficient to fully capture an equivalent RTL description of the AHPL model. The template includes only those constructs that are needed for the accurate and correct translation from AHPL to VHDL. The designed template style is shown in Figure 4.10. The template accommodates the multi-way branching feature of AHPL.

The template starts with the inclusion of the library that contain all the necessary functions and definitions. This is followed by the entity declaration and its architectural description. The entity merely contains the name of the entity being described and all its input and output port specification.

In the template, the architectural description begins with declarations of essential control signals and the signals used by the architectural body. The declaration of all the data registers and buses are made along with the declaration of all CLU output signals. This is followed by the disconnection specification for all the resolved signals. The disconnection specification is required for the purpose of turning off the source drivers of the resolved signal that are not active. The disconnection specification only applies to drivers of guarded signal assignments. As in the template, all the

```
library util_lib;
use util_lib.all;
use util_Pkg.all;
entity template is
  port ( "input and output port specification");
end template;
architecture template_body of template is
  -- declaration of control signals.
  signal active_state:state_type;
  signal next_state:resolution_type state_type register;
  -- declaration of all data registers and buses.
  -- declaration of all CLU outputs.
  -- disconnection specification for all the
       resolved signals.
  -- component specification for each CLU used.
  -- configuration statements with binding indication
       for each component.
```

Figure 4.10: Template Style.

```
begin
  -- component instantiation statements.
  control:block ( "clock trailing edge & reset detection")
  begin
    -- update active_state vector.
    -- clear the next_state vector.
  end block control;
  step_i:block ( "step i and reset signal detection")
  begin
    -- guarded bus assignment statements.
    edge_i:block ( "clock trailing edge detection")
    begin
      -- guarded data register assignment statements.
    end block edge_i;
    -- set-up the next_state vector.
  end block step_i;
        :
  step_j:block ( "step j and reset signal detection")
  begin
    -- guarded bus assignment statements.
    edge_j:block ( "clock trailing edge detection")
    begin
      -- guarded data register assignment statements.
    end block edge_j;
    -- set-up the next_state vector.
  end block step_j;
  end_seq: block ( "reset signal detection")
  begin
    -- guarded data register and
         bus assignment statements.
  end block end_seq;
end template_body;
```

Figure 4.10: Template Style (cont).

signal assignments are made inside a guarded block, the source drivers of non-active blocks are turned off as their guard expression becomes FALSE. The effect is that only the array of values with active states are passed to the resolution function. The disconnection specification is followed by the declaration of all CLU component specifications along with its binding configuration statements.

The body of the architecture contains a control *block*, a series of data *blocks* (one for each step of the procedural part of the AHPL description), and a *block* for the non-procedural part (statements between the keywords ENDSEQUENCE and END AHPL). Every *block* construct contains a guard condition, which when true enables the execution of all the statements in the *block* body concurrently.

All the operations in the non-procedural part are contained in the last *block*. This block does not contain any control signals and it is always active. For asynchronous control reset facility, a special reset signal is appended to all the guard conditions of the control and data *blocks*.

4.4.1 Control Path Emulation

In the template, the architectural description contains declaration of two important control signals ACTIVE_STATE and NEXT_STATE. These two signals are used to store the present active states and the future active states. These two signals are of type binary vector of size equal to the number of steps in the AHPL description. The ith bit of the vector indicates whether control state i is active (bit=1) or not

(bit=0). More than one bit of the vector could be high. Since there may be several INITIAL_STATEs leading to this NEXT_STATE, NEXT_STATE is declared as a resolved signal. Every resolved signal requires a resolution function to enumerate its sources and assign the appropriate value to it. Under normal sequential execution of the states, one state will be updating the NEXT_STATE signal. In concurrent execution of states the sources of all the active states are combined together by an *or* resolution function and assigned to the NEXT_STATE. Following the control signals, all the memory and other elements of the declaration part of the AHPL description are declared in their respective VHDL representation.

The architectural body contains a control *block* whose basic role is to update the ACTIVE_STATE signal with the NEXT_STATE signal and to clear the NEXT_STATE vector at the trailing edge of the clock. Figure 4.11 shows the style of a generic control clock and Figure 4.12 an example of sample control block. As can be seen in Figure 4.12, the control block is guarded on the trailing edge condition. The control block contains two guarded signal assignment statements that are executed concurrently when the trailing edge condition becomes TRUE. The overall effect is the updation of the ACTIVE_STATE vector with the NEXT_STATE vector and the clearance of the NEXT_STATE vector.

```
control:block ( "clock trailing edge & reset detection")
begin
  -- update active_state vector.
  -- clear the next_state vector.
end block control;
```

Figure 4.11: Generic Control Block.

```
control:block ((clock='0' and not clock'stable)
              or reset='1')
begin
  active_state<=guarded "10000" when reset='1'
                  else next_state;
  next_state<=guarded "00000";
end block control;
```

Figure 4.12: Sample Control Block.

4.4.2 Data Path Mapping

The data *blocks* follow the control *block* with each *block* containing a set of operations and the next state specification corresponding to each step in the AHPL description.

The guard of the data *blocks* determine when a particular *block* is active depending on the individual bits of the ACTIVE_STATE.

The body of the data *block* contains all the bus connection assignment statements followed by a sub-block containing all the register transfer assignment statements. The guard of this sub-block condition is to enforce register transfer assignments only at trailing edge of the clock. The final part of the data *block* contains the next state specification corresponding to the branch statement of a step in the AHPL description. The design of a generic data block is shown in Figure 4.13 and an example of a sample block for an AHPL statement is shown in Figure 4.14.

The Sample Data Block of Figure 4.14 shows a step block with a nested block for bus connection and register transfer respectively. The outer step block uses the step condition as the guard, and is used for guarded assignments into data buses. The inner block ANDs the events on the reference with the step condition with the clock trailing edge condition for register transfers. The transfer into registers are also done by use of guarded assignment statements. Finally, the outer block also contains a guarded assignment statement for the updation of the next_state control parameter.

```
step_i:block ("step i and reset signal detection")
begin
  -- guarded bus assignment statements.
  edge_i:block ("clock trailing edge detection")
  begin
    -- guarded data register assignment statements.
  end block edge_i;
  -- set-up the next_state vector.
end block step_i;
```

Figure 4.13: Generic Data Block.

```
5    RESULT=EXTRA[1:4],AC1; DONE=1$1;
     BUSY <=1$0; =>(5).
```

```
step_5:block (active_state(5)='1' and reset='0')
begin
  result<=guarded extra(1 to 4) & ac1;
  done<=guarded '1';
  edge_5:block (guard and clock='0'
                    and not clock'stable)
  begin
    busy<=guarded '0';
  end block edge_5;
  next_state<=guarded "00001";
end block step_5;
```

Figure 4.14: Sample Data Block.

4.4.3 CLU Mapping

The mapping of Combination Logic Units (CLU) is done using predefined generic components. The generic components are defined in the Utility Library for each of the six common CLUs used in AHPL: INC, DEC, ADD, SUB, DCD and BUSFN. An instance of the generic component definition are used to declare each of the CLUs used.

The CLU mapping is not straight forward and is split into three steps. Figure 4.15 shows the steps involved in the declaration of a CLU of AHPL in VHDL. In the first step the component used for the CLU is declared along with the binding if all the inputs and the outputs. In the second step the component specification is done to specify which entity declaration in the Utility Library and which architecture of that design entity is to be selected. As shown, the component specification binds the instance 'incl' of the INC to the entity INCR and to its architecture INCR_BODY in the Util_Lib. In the last step, the actual instantiation is performed inside the design entity architecture body. The component instantiation statement identifies the CLU component and specifies which ports or signals in the design entity are connected to which ports in the CLU component.

AHPL:

```
CLUNIT:    INC[2]<:INCR{2}
```

VHDL:

Step 1: Component Declaration

```
component INC generic (inc_size: positive)
  port(incr_inp : in bit_vector(0 to incr_size-1);
       incr_out : out bit_vector(0 to incr_size-1));
```

Step 2: Component Specification

```
for inc1: INC use entity Util.Lib.INCR(INCR_BODY)
```

Step 3: Component Instantiation

```
inc1: INC generic map (incr_size=>2)
          port map (count,inc1_out);
```

Figure 4.15: CLU Mapping.

4.5 Utility Library

In order to generate concise VHDL descriptions, a library of utilities was implemented. This library is used by the *Composer* coding many necessary VHDL constructs. The library includes several predefined VHDL constructs:

- *Type Declarations*: Type definition of memory vector of different sizes.
- *Resolution Functions*: Functions to resolve output from multi-source for all type definitions.
- *Type transformation Functions*: Functions to map between different data types.
- *Generic Component*: Generic design entities for all the commonly used AHPL CLUs.

The availability of library facility allows the possibility of any future expansion or enhancement of the *composition process.*

4.6 An Illustrative Example

Figure 4.16 shows the composed VHDL model for the AHPL model of Figure 2.2. For explanation purpose all the statements are numbered and these line numbers are not part of the actual VHDL code.

The composed VHDL model basically consists of three parts: the utility library interface, entity description, and an architecture description part. Lines 1-3 define

```
1   library util_lib;
2   use util_lib.all;
3   use util_Pkg.all;
4   entity multiplier is
5               port (dataready,clock,reset : in mem_vec_1;
6                     inputbus : in mem_vec_8;
7                     result : out mem_vec_8;
8                     done,busyout : out mem_vec_1);
9   end multiplier;
10  architecture multiplier_body of multiplier is
11    signal active_state : state_vec_5;
12    signal next_state : state_resolution_5 state_vec_5 register;
13    signal ac1 : resolution_fun_4 mem_vec_4 register;
14    signal ac2 : mem_vec_4;
15    signal count : mem_vec_2;
16    signal extra : resolution_fun_5 mem_vec_5 register;
17    signal busy : resolution_fun_1 mem_vec_1 register;
18    signal inc1_out : mem_vec_2;
19    signal add1_out : mem_vec_5;
20    disconnect next_state : state_vec_5 after 0 ns;
21    disconnect extra : mem_vec_5 after 0 ns;
22    disconnect ac1 : mem_vec_4 after 0 ns;
23    disconnect busy : mem_vec_1 after 0 ns;
24    component INC generic (incr_size : positive);
25        port (incr_inp : in bit_vector(0 to incr_size-1);
26              incr_out : out bit_vector(0 to incr_size-1));
27    end component;
28    component ADD generic (adder_size : positive);
29        port (adder_inp1 : in bit_vector(0 to adder_size-1);
30              adder_inp2 : in bit_vector(0 to adder_size-1);
31              adder_out : out bit_vector(0 to adder_size));
32    end component;
33    for inc1:INC use entity Util_Lib.INCR(INCR_BODY);
34    for add1:ADD use entity Util_Lib.ADDER(ADDER_BODY);
35  begin
36    inc1: INC generic map (incr_size=>2)
37                port map (count, inc1_out);
38    add1: ADD generic map (adder_size=>4)
39                port map (extra(1 to 4),ac2,add1_out);
```

Figure 4.16: Composed VHDL Model of Multiplier.

```
40      control:block ((clock='0' and not clock'stable) or reset='1')
41      begin
42        active_state <= guarded "10000" when reset='1'
                       else next_state;
43        next_state<=guarded "00000";
44      end block control;
45      step_1:block (active_state(1)='1' and reset='0')
46      begin
47        edge_1:block (guard and clock='0' and not clock'stable)
48        begin
49          ac1<=guarded inputbus(0 to 3);
50          ac2<=guarded inputbus(4 to 7);
51          extra<=guarded "00000";
52        end block edge_1;
53        next_state<=guarded "10000" when dataready='0' else '01000";
54      end block step_1;
55      step_2:block (active_state(2)='1' and reset='0')
56      begin
57        edge_2:block (guard and clock='0' and not clock'stable)
58        begin
59          busy<=guarded '1';
60        end block edge_2;
61        next_state<=guarded "00100" when ac1(3)='1' else "00010";
62      end block step_2;
63      step_3:block (active_state(3)='1' and reset='0')
64      begin
65        edge_3:block (guard and clock='0' and not clock'stable)
66        begin
67          extra<=guarded add1_out(0 to 4);
68        end block edge_3;
69        next_state<=guarded "00010";
70      end block step_3;
```

Figure 4.16: Composed VHDL Model of Multiplier (cont).

```
71      step_4:block (active_state(4)='1' and reset='0')
72      begin
73        edge_4:block (guard and clock='0' and not clock'stable)
74        begin
75          extra<=guarded '0' & extra(0 to 3);
76          ac1<=guarded extra(4) & ac1(0 to 2);
77          count<=guarded inc1_out;
78        end block edge_4;
79        next_state<=guarded "01000"
              when (count(0)='0' or count(1)='0') else "00001";
80      end block step_4;
81      step_5:block (active_state(5)='1' and reset='0')
82      begin
83        result<=guarded extra(1 to 4) & ac1;
84        done<=guarded '1';
85        edge_5:block (guard and clock='0' and not clock'stable)
86        begin
87          busy<=guarded '0';
88        end block edge_5;
89        next_state<=guarded "00001";
90      end block step_5;
91      end_seq:block (reset='0')
92      begin
93        busyout<=guarded busy;
94      end block end_seq;
95    end multiplier_body;
```

Figure 4.16: Composed VHDL Model of Multiplier (cont).

the interface to the utility library and all the modules that are accessible. Lines 4-9 describe the entity along with its input and output ports. Lines 10-95 describe the architecture of the entity. The architecture is made-up of two sections: the declaration section (lines 11-34) and the body section (lines 36-94). The declaration section contains definitions of all the data registers and buses, the disconnection specification for the resolved signals, and component specifications for the CLU used. The body section contains instantiation statements (lines 36-39) for each CLU used followed by a *control* block (lines 40-44), a series of *step* blocks (lines 45-54, 55-62, 63-70, 71-80, 81-90) for AHPL steps 1 to 5 respectively, and *end_seq* block for the part between ENDSEQUENCE and END in the AHPL model.

The composed VHDL model was simulated to multiply two numbers and the simulated output waveform is shown in Figure 4.17. The two chosen numbers (3 and 4) were put on *inputbus* signal and the output is generated after ten clock pulses on signal *result*.

4.7 Summary

This chapter discussed an overview of the *composition process*. The general overview presented was followed by a comprehensive discussion of the the components of the composition system. The component discussed are: (a) The Analyzer, (b) The Composer, (c) The VHDL Template, and, (d) The Utility Library. The section

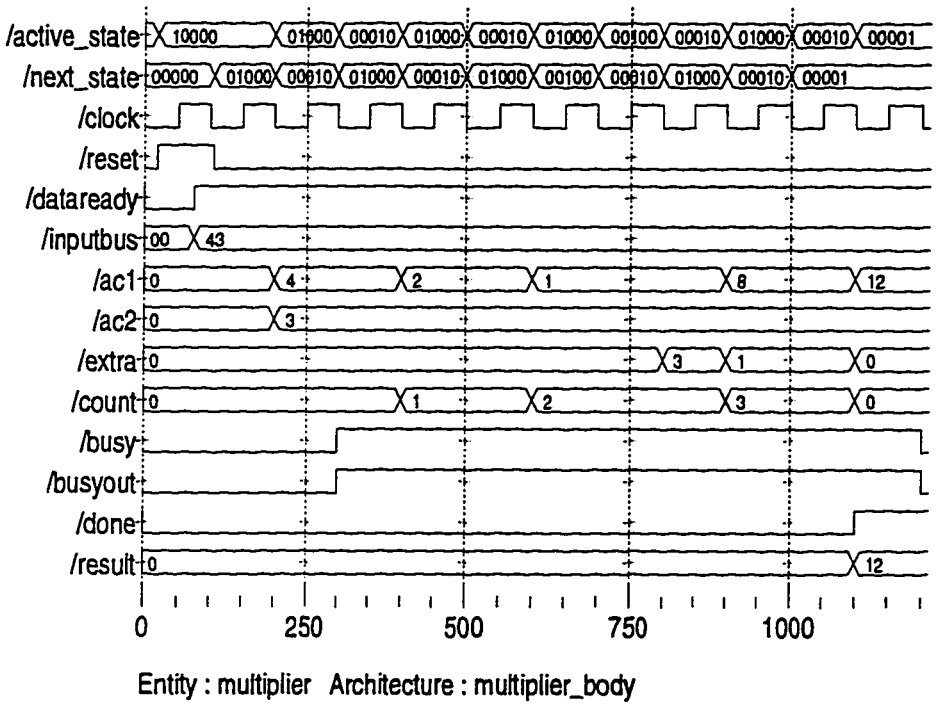

Figure 4.17: VHDL Simulated Output.

on *Analyzer* presented the implementation details and the process of generation of the corresponding state tables. These state tables comprise the intermediate results in a convenient format used by the *Composer*. The three modules that perform the three tasks of the *Composer* are: (a) The Declaration Transformer, (b) The Controller Generator and (c) The Data Path Builder, respectively.

Sections on Control Path Emulation, Data Path Mapping, and CLU Mapping, highlight the details of the The VHDL Template. Contents of the The Utility Library that contains several useful hardware components, and other declarations and function where elaborated. An illustrative example was used in the entire chapter to demonstrate the *composition process.*

Chapter 5

Conclusion and Future Work

In this thesis a *composition process* for the generation of the VHDL models from AHPL specifications was designed and implemented. The objective was achieved by the development of an *Analyzer* to examine and capture data and control paths of the AHPL specification and, a *Composer* to build the final output in the VHDL format. To accomplish the role of the *Composer* a VHDL template was designed which provided the skeleton structure of the produced VHDL model.

VHDL is a multi-purpose HDL and is among the hardest HDLs to master. On the other hand, AHPL is a single purpose language, is relatively small and is among the easiest to learn. The prime motivation of this work was to assist the designer familiar with AHPL-like languages to easily migrate to VHDL language and quickly start prototyping in VHDL. In addition, the *composition process* also provides useful understanding of the large variety of VHDL features and constructs.

The various aspects of the thesis were presented in four main chapters. The opening chapter, Chapter 1, discussed the significance of the work. The objective and motivation of the work were also highlighted.

Chapter 2 presented an introduction for the hardware description language (HDL). Two Languages, AHPL and VHDL were studied and an overview of them was presented. These two languages were compared against well-known HDLs and the comparison analysis has shown that VHDL provides extensive features that are useful for modern modeling techniques.

In Chapter 3 a detailed study was conducted on the important VHDL features. VHDL is a very large language and only subset of VHDL constructs are required for the fulfillment of mapping different AHPL elements and features into their corresponding counter-parts. The mapping of the AHPL elements and features such as on declarations, state automaton, edge triggered transfers, and other processes in AHPL were summarized in a table. The VHDL *guarded blocks* construct provides a very convenient method to model various aspect of AHPL specification.

Finally, the design implementation of the *composition process* was discussed in Chapter 4. In this core chapter, the mechanics of the *composition process* was described and a comprehensive discussion was presented on the four components of the *composition process*: (a) *The Analyzer*, (b) *The Composer*, (c) *The VHDL Template*, and, (d) *The Utility Library.* The *Analyzer* was used for the process of the generation of the corresponding state tables. These state tables comprise the in-

termediate results in a convenient format used by the *Composer.* The task of the *Composer* is accomplished by the three modules: (a) The Declaration Transformer, (b) The Controller Generator and (c) The Data Path Builder, respectively. The states tables are read by the three modules of the *Composer* to build the VHDL model as per the specification of the *VHDL Template.* The *VHDL Template* conceptualizes how the control path is emulated and how the data path and the CLU are mapped. The *Composer* is aided with a *Utility Library* which contains predefined generic CLU component definitions, user-defined types, resolution functions and other declaration.

The *composition process* described in this work provides a convenient tool to convert digital designs modeled in AHPL to their VHDL counter-part. The output result produced by the *composition process* envisages all the necessary concept of the AHPL specification.

Although the design implementation is well defined in this work, the *composition process* could be further enhanced and improved. The four components of the *composition process* design provide the facility to accommodate any expansion that may be required. The *Analyzer* could be enhanced to include semantic checks as well syntactic check for the input AHPL code. This could easily be implemented by embedding the necessary semantic rules in the action part of the production rules in the yacc program. The implementation of the expression builder in the *Analyzer* will also provide a convenient facility to convert directly the AHPL expressions into

VHDL expressions. The *Composer* and the *The VHDL Template* may also be re-defined to provide an alternative style for the composed VHDL model. The support of the *The Utility Library* provides the flexibility to include any user-defined types and functions.

Finally, the composition of VHDL models from AHPL specification provides useful insights as to how best one can perform high-level synthesis of digital system from VHDL description. The understanding of the *composition process* will assist the reverse engineering of the high-level synthesis process. Good prospects of future work in this area include the study of VHDL constructs for synthesis of its hardware.

Appendix A

```
cid         (inc|INC|dec|DEC|add|ADD|sub|SUB|dcd|DCD|busfn|BUSFN)[0-9]+
id          [a-zA-Z][a-zA-Z0-9]*
integer     [0-9]+

%%
module|MODULE                   { return MODULE; }
buses|BUSES                     { return BUSES; }
exbuses|EXBUSES                 { return EXBUSES; }
exinputs|EXINPUTS               { return EXINPUTS; }
inputs|INPUTS                   { return INPUTS; }
memory|MEMORY                   { return MEMORY; }
outputs|OUTPUTS                 { return OUTPUTS; }
exoutputs|EXOUTPUTS             { return EXOUTPUTS; }
clunits|CLUNITS                 { return CLUNITS; }
body|BODY                       { return BODY; }
sequence|SEQUENCE               { return SEQUENCE; }
endsequence|ENDSEQUENCE         { return ENDSEQUENCE; }
end|END                         { return END; }
nodelay|NODELAY                 { return NODELAY; }
null|NULL                       { return NULLSTMT; }
deadend|DEADEND                 { return DEADEND; }
controlreset|CONTROLRESET       { return CONTROLRESET; }
"."                             { return DOT; }
":"                             { return COLON; }
";"                             { return SEMICOLON; }
"<"                             { return LESS; }
">"                             { return GREATER; }
"["                             { return SQUARELEFT; }
"]"                             { return SQUARERIGHT; }
"+"                             { return PLUS; }
"*"                             { return STAR; }
```

```
"/"          { return SLASH; }
"~"          { return TILDA; }
"("          { return CURLYLEFT; }
")"          { return CURLYRIGHT; }
"="          { return EQUAL; }
"<="         { return LESSEQUAL; }
"!"          { return EXCLAIM; }
","          { return COMMA; }
"@"          { return ATTHERATE; }
"+/"         { return PLUSSLASH; }
"&/"         { return AMPERSLASH; }
"&"          { return AMPER; }
"$"          { return DOLLAR; }
"\"          { return BACKSLASH; }
"=>"         { return EQUALGREATER; }
{cid}        { strcpy(id_val,yytext);
               return CID; }

{id}         { strcpy(id_val,yytext);
               return ID; }

{integer}    { int_val=atoi(yytext);
               return INTEGER; }

[ \t\n]      ;
%%
```

Appendix B

```
%{
/* #define YYSTYPE double */
%}

%token MODULE BODY SEQUENCE ENDSEQUENCE END
%token CONTROLRESET MEMORY INPUTS OUTPUTS BUSES
%token EXINPUTS EXOUTPUTS EXBUSES CLUNITS
%token NODELAY NULLSTMT DEADEND LESSEQUAL EQUAL
%token DOT COLON SEMICOLON SLASH EQUALGREATER
%token LESS GREATER SQUARELEFT SQUARERIGHT
%token CURLYLEFT CURLYRIGHT AMPER PLUS ATTHERATE
%token AMPERSLASH PLUSSLASH TILDA STAR EXCLAIM
%token COMMA DOLLAR BACKSLASH INTEGER ID CID

%%

ahplprogram     : modhead DOT moddecls DOT modseq DOT
                    {printf("\nAHPL code is syntactically correct\n"); }
                ;

modhead         : MODULE COLON ID
                    { printf("\nModule Name : %s\n",id_val); }
                ;

moddecls        : moddecls DOT mdecl
                | mdecl
                ;

modseq          : BODY SEQUENCE COLON slrm DOT procpart
                        DOT ENDSEQUENCE noproc DOT END
                ;
```

```
mdecl           : type1 COLON id_dim_list
                | type2 COLON clu_dim_list
                ;

type1           : BUSES             { strcpy(ident_info.type,"BUSES");   }
                | EXBUSES           { strcpy(ident_info.type,"EXBUSES"); }
                | EXINPUTS          { strcpy(ident_info.type,"EXINPUTS"); }
                | INPUTS            { strcpy(ident_info.type,"INPUTS"); }
                | MEMORY            { strcpy(ident_info.type,"MEMORY"); }
                | OUTPUTS           { strcpy(ident_info.type,"OUTPUTS"); }
                | EXOUTPUTS         { strcpy(ident_info.type,"EXOUTPUTS"); }
                ;

type2           : CLUNITS           { strcpy(ident_info.type,"CLUNITS"); }
                ;

id_dim_list     : id_dim_list SEMICOLON id_dim
                                    { add_ident(ident_info); }
                | id_dim
                                    { add_ident(ident_info); }
                ;

id_dim          : ID                { strcpy(ident_info.name,id_val);
                                      ident_info.size=1;
                                      ident_info.lenght=1; }
                | ID dimension2     { strcpy(ident_info.name,id_val);
                                      ident_info.size=1; }
                | ID dimension1 dimension2  { strcpy(ident_info.name,id_val); }
                ;

clu_dim_list    : clu_dim_list SEMICOLON clu_dim
                                    { add_ident(ident_info); }
                | clu_dim
                                    { add_ident(ident_info); }
                ;

clu_dim         : CID               { strcpy(ident_info.name,id_val);
                                      ident_info.size=1;
                                      ident_info.lenght=1; }
                | CID dimension2    { strcpy(ident_info.name,id_val);
```

```
                                    ident_info.size=1; }
                | CID dimension1 dimension2 { strcpy(ident_info.name,id_val); }
                ;

dimension1      : LESS INTEGER GREATER
                                    { ident_info.size=int_val; }
                ;

dimension2      : SQUARELEFT INTEGER SQUARERIGHT
                                    { ident_info.lenght=int_val; }
                ;

slrm            : ID subs_range     { slrmval=id_val; }
                | ID                { slrmval=id_val; }
                ;

step_string : CURLYLEFT step_string1 CURLYRIGHT
                ;

step_string1    : step_string1 COMMA INTEGER
                        { br_info.step_num[++br_info.step_cnt]=int_val; }
                | INTEGER
                        { br_info.step_num[++br_info.step_cnt]=int_val; }
                ;

numb_string     : CURLYLEFT numb_string CURLYRIGHT
                | numb_string COMMA INTEGER
                | INTEGER
                ;

procpart        : procpart DOT stepnum steps
                  { if (br_next)
                     { br_info.step_num[++br_info.step_cnt]=step_num+1; }
                    for (i=1; i<=br_info.step_cnt; ++i)
                            printf("   next=>%d\n",br_info.step_num[i]); }
                | stepnum steps
                    { if (br_next)
                       { br_info.step_num[++br_info.step_cnt]=step_num+1; }
                      for (i=1; i<=br_info.step_cnt; ++i)
                              printf("   next=>%d\n",br_info.step_num[i]); }
                ;
```

```
noproc          : startstep SEMICOLON relation
                | startstep
                ;

stepnum         : INTEGER           { step_num=int_val;
                                      printf("step=>%d\n",step_num);
                                      br_next=1;
                                      br_info.step_cnt=0; }
                ;

steps           : NODELAY action
                | action
                | NULLSTMT
                | DEADEND           { br_next=0; }
                ;

action          : relation SEMICOLON branch
                | relation
                | branch
                ;

relation        : relation SEMICOLON relation1
                | relation1
                ;

branch          : EQUALGREATER glrm SLASH step_string
                | EQUALGREATER step_string              { br_next=0; }
                ;

relation1       : transfer
                | connection
                ;


transfer        : dlrm LESSEQUAL glrm
                | clhs LESSEQUAL glrm
                ;

connection      : dlrm EQUAL glrm
                ;
```

```
invok_list      : invok_list SEMICOLON cglrm
                | cglrm
                ;

bglrm           : bglrm EXCLAIM glrm1
                | glrm1
                ;

cglrm           : bglrm
                ;

dlrm            : dlrm EXCLAIM dlrm1
                | dlrm1
                ;

glrm            : bglrm STAR bglrm
                | bglrm
                ;

clhs            : dlrm STAR bglrm
                | clhs1
                ;

clhs1           : CURLYLEFT clhs CURLYRIGHT
                ;

dlrm1           : dlrm1 COMMA dlrm2
                | dlrm2
                ;

dlrm2           : CURLYLEFT dlrm CURLYRIGHT
                | slrm
                ;

glrm1           : glrm1 COMMA glrm2          { printf("SYM=,\n"); }
                | glrm2
                ;

glrm2           : glrm2 ATTHERATE glrm3      { printf("SYM=\n"); }
```

```
                | glrm3
                ;

glrm3           : PLUSSLASH glrm3          { printf("SYM=++\n"); }
                | glrm4
                ;

glrm4           : glrm4 PLUS glrm5         { printf("SYM=+\n"); }
                | glrm5
                ;

glrm5           : AMPERSLASH glrm5         { printf("SYM=&&\n"); }
                | glrm6
                ;

glrm6           : glrm6 AMPER glrm7        { printf("SYM=&\n"); }
                | glrm7
                ;

glrm7           : TILDA element            { printf("SYM= \n"); }
                | element
                ;

element         : ID CURLYLEFT invok_list CURLYRIGHT
                                                { printf("E=i\n"); }
                | ID subs_range CURLYLEFT invok_list CURLYRIGHT
                                           { printf("E=%s\n",id_val); }
                | INTEGER DOLLAR INTEGER        { printf("E=$\n"); }
                | BACKSLASH numb_string BACKSLASH { printf("E=s\n"); }
                | CURLYLEFT bglrm CURLYRIGHT    { printf("E=?\n"); }
                | slrm                     { printf("E=%s\n",slrmval); }
                ;

subs_range      : LESS range GREATER
                        SQUARELEFT range SQUARERIGHT
                | SQUARELEFT range SQUARERIGHT
                        LESS range GREATER
                | LESS range GREATER
                | SQUARELEFT range SQUARERIGHT
                ;
```

```
startstep           : CONTROLRESET CURLYLEFT glrm CURLYRIGHT
                            SLASH CURLYLEFT numb_string CURLYRIGHT
                    | CONTROLRESET CURLYLEFT numb_string CURLYRIGHT
                    ;

range               : range1 COLON INTEGER      { range_val2=int_val; }
                    | INTEGER                   { range_val1=int_val;
                                                  range_val2=-1;      }
                    ;

range1              : INTEGER                   { range_val1=int_val; }
                    ;

%%

#include <stdio.h>
#include <ctype.h>
#include "utility.h"

int                 i;
char                iname[20];
char                *itype;
int                 ilenght;

int                 step_num;
ident_rec   ident_info;
branch_rec br_info;
int                 br_next;

int                 int_val;
char                id_val[100];

#include "lex.yy.c"

char    *progname;
int     lineno = 1;

char        *slrmval;
int         range_val1,range_val2;
```

```
main(argc,argv)
        char *argv[];
{
        progname = argv[0];
        yyparse();
                printf("\nEnd of Parsing..\n");
                show_ident();
}

yyerror(s)
        char *s;
{
        warning(s, (char *) 0);
}

warning(s, t)
        char *s, *t;
{
        fprintf(stderr, "****** ATTENTION USER ****** \n");
        fprintf(stderr, "*** Syntax Error Occured *** \n");
        fprintf(stderr, "INVALID CODE NEAR : %s \n",yytext);
        fprintf(stderr, "************************** \n\n\n");
}
```

Bibliography

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

[AWS86] James H. Aylor, Ron Waxman, and Charles Scarratt. VHDL - Feature Description and Analysis. *IEEE Design and Test of Computers*, 3(2):17–27, April 1986.

[DG86] Allen Dewey and Anthony Gadient. VHDL Motivation. *IEEE Design and Test of Computers*, 3(2):12–16, April 1986.

[HDL92a] Three Decades of HDLs - Part 1. *IEEE Design and Test of Computers*, 9(2):69–81, June 1992.

[HDL92b] Three Decades of HDLs - Part 2. *IEEE Design and Test of Computers*, 9(3):54–63, September 1992.

[HP73] Fedrick J. Hill and Gerald R. Peterson. *Digital Systems: Hardware Organization and Design.* John Wiley & Sons, Inc., 1973.

[JW89] Paul R. Jordan and Ronald D. Williams. VCOMP: A VHDL Composition System. In *26th ACM IEEE Design Automation Conference*, pages 750–753, December 1989.

[LG89] Joseph S. Lis and Daniel D. Gajski. VHDL Synthesis Using Structured Modeling. In *26th ACM IEEE Design Automation Conference*, pages 606–609, December 1989.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly & Associates, Inc., 1992.

[LMS86] Roger Lipsett, Erich Marchner, and Moe Shahdad. VHDL - The Language. *IEEE Design and Test of Computers*, 3(2):28–41, April 1986.

[LSU89] Roger Lipsett, Carl F. Schaefer, and Cary Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publisher, 1989.

[Nav93] Zainalabedin Navabi. *VHDL Analysis and Modeling of Digital Systems.* McGraw-Hill, Inc., 1993.

[NBD92] Vijay Nagasamy, Neerav Berry, and Carlos Dangelo. Specification, Planning, and Synthesis in a VHDL Design Environment. *IEEE Design and Test of Computers*, 9(2):58–68, June 1992.

[NS86] J. D. Nash and Larry F. Saunders. VHDL Critique. *IEEE Design and Test of Computers*, 3(2):54–65, April 1986.

[NS90] Zainalabedin Navabi and John Spillane. Templates For Synthesis from VHDL. In *Proc. of the 1990 ASIC Seminar and Exposition*, September 1990.

[Per93] Douglas L. Perry. *VHDL*. McGraw-Hill, Inc., 2 edition, 1993.

[RKDV92] Jayanta Roy, Nand Kumar, Rajiv Dutta, and Ranga Vemuri. DSS: A Distributed High-Level Synthesis System. *IEEE Design and Test of Computers*, 9(2):18–32, June 1992.

[SB92] Mani B. Srivastava and Robert W. Brodersen. Using VHDL for High-Level Mixed-Mode System Simulation. *IEEE Design and Test of Computers*, 9(3):31–40, September 1992.

[SYBS93] Sadiq M. Sait, Habib Youssef, Muhammad S. T. Benten, and Faisal M. Z. Soleja. Automated VHDL Composition from AHPL. *Proceedings of the 5th International Conference on Microelectonics*, pages 220–224, December 1993.

[SYBS94] Sadiq M. Sait, Habib Youssef, Muhammad S. T. Benten, and Faisal M. Z. Soleja. Automated VHDL Composition from AHPL. *The Arabian Journal for Science and Engineering*, 19(4B):771–781, October 1994.

[VHD88] *IEEE Standard VHDL Language Reference Manual 1076-1987.* IEEE Press, March 1988.

[Wax86] Ron Waxman. The VHSIC Hardware Description Language - A Glimpse of the Future. *IEEE Design and Test of Computers*, 3(2):10–11, April 1986.

[WC91] Robert A. Walker and Raul Camposano. *A Survey of High-Level Synthesis Systems.* Kluwer Academic Publishers, 1991.

[WS92] John C. Willis and Daniel P. Siewiorek. Optimizing VHDL Compilation for Parallel Simulation. *IEEE Design and Test of Computers*, 9(3):42–53, September 1992.