A VHSIC
HARDWARE DESCRIPTION LANGUAGE COMPILER
FOR LOGIC CELL ARRAYS

by

Bing Liu

A thesis

presented to the University of Manitoba

in partial fulfillment of the

requirements of the degree of

Master of Science

in Electrical and Computer Engineering

Winnipeg, Manitoba, Canada

ISBN 0-315-71751-3

A VHSIC HARDWARE DESCRIPTION LANGUAGE COMPILER FOR LOGIC CELL ARRAYS

BY

BING LIU

A thesis submitted to the Faculty of Graduate Studies of the University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

ABSTRACT

This thesis presents a development of a VHSIC Hardware Description Language (VHDL) compiler for Logic Cell Arrays (LCAs). First, the concept of electronic circuit engineering and the electronic circuit development cycle using computer aided engineering (CAE) tools are reviewed; and the motivation of this research work is provided. Then, the architecture, design methodology and the significance of LCA are described. Thirdly, the VHDL is briefly reviewed and a VHDL architectural description subset for LCA is defined as the input of the compiler; the Xilinx Netlist Format (XNF) is chosen as the target of the compiler. Finally, the development, testing and verification of the VHDL compiler for LCA is described. Two of the examples implemented from the VHDL descriptions are presented to demonstrate the compiler.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to Dr. Witold Kinsner, my advisor, and Dr. Robert D. McLeod for their excellent guidance, endurable motivation and consistent support throughout the course of this research.

My thanks go to Dr. William L. Kocay for his comments and suggestions on my thesis. I am also grateful to Mr. J. Dickson, who helped me to simulate one of the examples, and all the other fellow students for their help.

I would also like to thank my wife who helped me a great deal during my writing of this thesis, my parents, and family for their encouragement.

The financial support of this research work provided by the National Science and Engineering Research Council (NSERC) of Canada through Dr. Kinsner's grant and Dr. McLeod's grant, Canadian Microelectronics Corporation, and the University of Manitoba is gratefully acknowledged. The LCA Xilinx Netlist Format Specification and the LCA External Netlist Tool Kit provided by Xilinx Inc. is also acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ASIC	Application Specific Integrated Circuit.
CAD	Computer Aided Design
CAE	Computer Aided Engineering
CAT	Computer Aided Testing
CAV	Computer Aided Verification
CLB	Configurable Logic Block.
DRC	Design Rule Checker
IC	Integrated Circuit
IOB	Input/Output Block
IVHDL	Internal VHDL representation
IXNF	Internal Xilinx Netlist Format
HDL	Hardware Description Language
LCA	Logic Cell™ Array
LFSR	Linear Feedback Shift Register.
PAL	Programmable Array Logic
PCB	Printed Circuit Board
PLD	Programmable Logic Device
SMB	Surface Mount Board
VHSIC	Very High Speed Integrated Circuit
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
XNF	Xilinx Netlist Format

CHAPTER 1

INTRODUCTION

The design of electronic systems always involves the design of electronic circuits, often in the form of either integrated circuits (ICs) using very large scale integration (VLSI) or very high speed ICs (VHSIC), or printed circuit boards (PCBs). The design process of electronic circuits can be considered as a transformation of a behavioural description of the circuit concepts into a physical description of the circuits suitable for implementation. A behavioural description is the highest level of abstraction, providing only the functional characteristics of circuits with no specified way of implementing them. For example, the behavioural description of a subtraction unit simply states that the output be the difference of the two inputs. For complex circuits, the transformation is achieved by a hierarchical decomposition from behavioural description to architectural description, then to physical description.

The architectural description is closer to the final implementation because it provides the necessary building blocks and the connections among the blocks with or without the information of how to implement the circuit. The architectural description defines the architecture of the circuit and its organization if the implementation information is contained. An architecture may be implemented in different approaches. For example, the architecture of the processor SPARC has two organizations implemented by two companies

respectively [RISC89]. The architectural description of a subtraction unit would show the negation modules, the summation modules, the carry lines, and the connections among them. The architectural description may have many different levels. The lower the level, the more the detailed building blocks and their connections become. The architectural description at the lowest level is used for placement and routing to generate physical descriptions.

The physical descriptions are the lowest level of circuit description, in which the geometrical representation of the circuits is provided. The physical description is also called circuit layout. At this stage, the circuits are ready to be implemented. Depending on the target technologies of circuit implementation, many different forms of physical description are possible. Common technologies include the use of standard components mounted on PCBs, or surface mount boards (SMBs), and semi-custom or full-custom integrated circuit (IC) fabricated on silicon dies, and programmable logic devices (PLDs).

While the behavioural description is textual, architectural and physical descriptions may be textual or graphical. Textual descriptions specify circuits in the forms of hardware description languages (HDLs) that are formalized modular languages. The graphic description is called schematic, which requires sketching programs that allow the user to physically place circuitry on a display screen.

As the circuit technology develops, the complexity of a circuit increases exponentially. This ever increasing circuit complexity has rendered manual intervention tedious, error-prone and time consuming. On the other hand, the demand for application specific IC (ASIC) is also increasing. As a result, computer-aided tools have been

developed to help implement and verify circuit designs. Electronic circuit engineering now utilizes computer-aided engineering (CAE) tools, including computer-aided design (CAD) tools, computer-aided verification (CAV) tools, and computer-aided test (CAT) tools, to reduce the design effort, turnaround time and design error, and at the same time to improve the design quality.

1.1. Electronic Circuit Engineering

The design process of electronic circuit is shown in Fig. 1.1. The concept of a circuit is first studied to establish the behaviour of the circuit. The behaviour is specified in behavioural description, which is used for modeling and prototyping to verify the specification, and for decomposition into architectural description [Kins86].

The architectural description is used for functional simulation to verify the circuit design at early stage. The building blocks are placed and the connections among the blocks are routed towards the target technology of implementing the circuits. Placement and routing generates the physical description of the circuit, i.e. the circuit layout.

The physical description is also used for simulation to verify the function and timing delay of the circuit. The circuit is manufactured using the circuit layout if the simulation is passed. The product must be tested before the acceptance.

1.1.1. Electronic Circuit Design

The CAE tools are used throughout the design process. The circuit design development cycle using CAE tools includes three stages: design entry, design

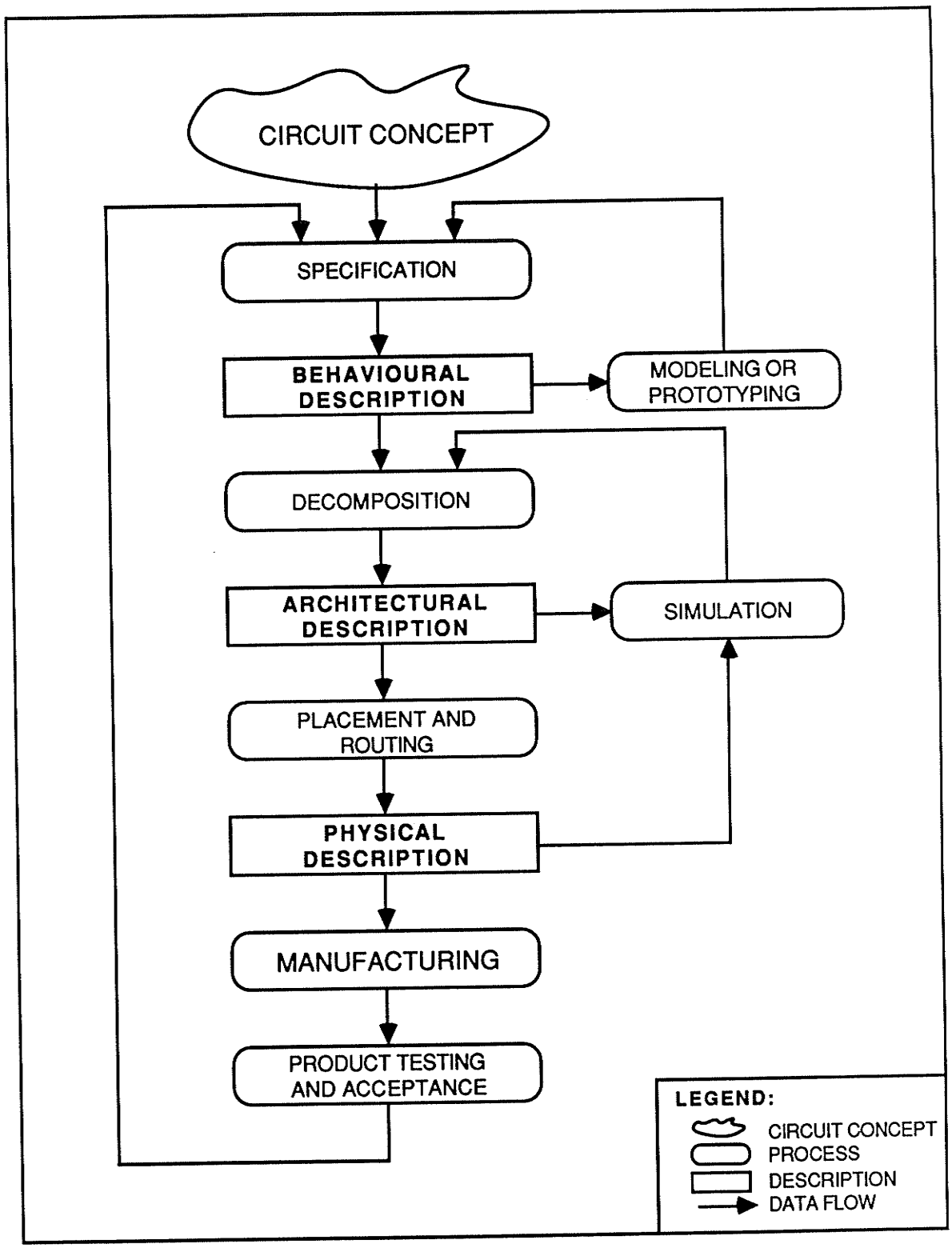

Fig. 1.1. Electronic circuit design process [after Kins86].

implementation and design verification. The CAE tools used at different stages are shown in Fig. 1.2.

The CAD tools include three groups: (i) schematic capture; (ii) HDL compilation; and (iii) placement and routing. Schematic capture tools are the sketching programs that users can place the building blocks on a display screen and draw the connections among the blocks. The schematics are converted to an architectural description. HDL compilation tools accept textual circuit descriptions. At the highest level the compiler decomposes the behavioural descriptions into architectural descriptions, and converts the architectural descriptions to a level for placement and routing. Placement and routing tools accept the architectural description at the lowest level , and generate the circuit layout.

The CAV tools include design rule checkers (DRCs) and simulators. Simulators accept textual descriptions and simulate the functions and timing delay of the circuits.

The circuit descriptions have to be formed into machine understandable formats, and entered into the computers in order to use CAE tools. This is the design entry stage. The CAE tools may accept the descriptions at the behavioural, architectural, or physical level in text using a text editor, or graphic using a schematic capture tool.

The CAE tools may perform decompositions, conversions, and placement and routing at the design implementation stage. If the design entries are at the behavioural level, HDL compilers decompose them into an architectural description. As design entries, architectural descriptions may be in textual or graphical forms. If graphical descriptions are entered into the computers, the schematic capture tools extract the circuit descriptions into a textual format. The architectural descriptions at high level must be converted into a lower

level suitable for placement and routing. The circuit designs can be implemented after the physical descriptions are generated by placement and routing.

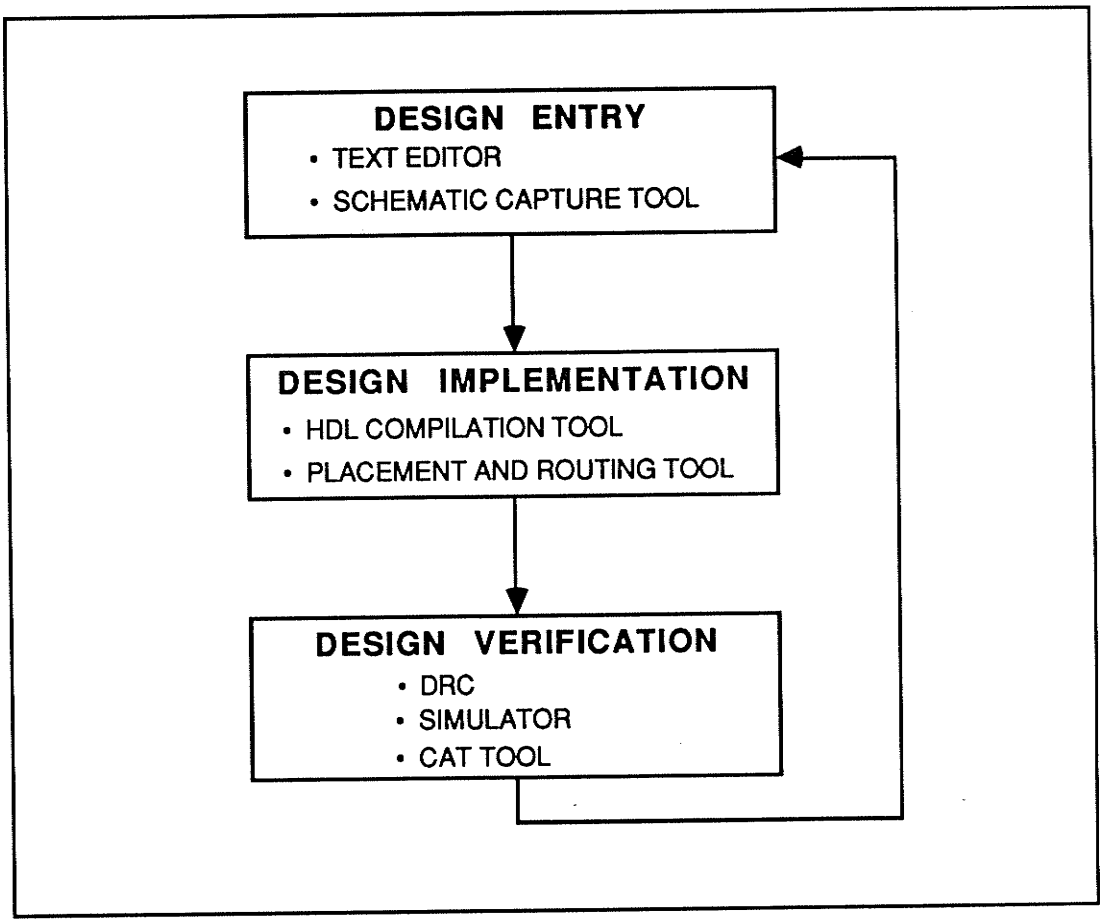

Fig 1.2. Electronic circuit development cycle using CAE tools.

While behavioural descriptions provide a great promise for the future, the decomposition from a behavioural description to an architecture that matches the function specification still remains difficult [Sang86]. More research is needed in this area. In general, it is also very difficult to obtain optimized layouts from automatic placement and routing due to the complexity of this problem. However, acceptable layouts may be obtained for the Programmable Logic Device (PLD) target technology because its regular

architecture provides constraints, thereby reducing complexity of the problem.

At the design verification stage, the DRC checks the design according to the design rules and reports design rule violations. Simulations must then be carried out at different levels to verify the functions of the designs. Design verification helps the designers to find out design faults at early design stages. Simulations provide design modelling at the behavioural description level, and worst-case analysis at the physical description level.

1.1.2. Programmable Logic Devices [Kins89]

The first PLD device, the Field-Programmable Logic Array (FPLA), was introduced by Signetics in the early 1970s. In 1975, Monolithic Memories followed with their Programmable Array Logic (PAL) devices [BiCo81, MoMe81]. Both PLDs were based on fusible-link technology. In the early 80s, the devices were used extensively in different practical industrial and experimental applications, requiring moderate random logic and small sequencers with high reliability. Later, implementation of more complex circuits demanded more complex PLDs, a number of which were introduced by Advanced Micro Devices, Fairchild Semiconductor, GE/Intersil, Harris Semiconductor, Intel, National Semiconductor, Texas Instruments, and newer companies such as Actel, Altera, Cypress Semiconductor, Exel Microelectronics, Lattice Semiconductor, and VLSI Technology [Marr86, Coll86a, Coll86b, Coll87, Meye87, Smit88, Free88, Actel88]. Many of those devices have advanced from the old fusible-link technology to floating-gate-based ultraviolet-erasable PLDs (UVEPLDs or simply EPLDs) and electrically erasable PLDs (EEPLDs) [Goet86], as well as from the old bipolar to CMOS technologies. A recent example of such devices is the CMOS electrically-configurable gate arrays

[AGGC88] which combine the flexibility of mask-programmable gate arrays and convenience of user-programmable PLDs. Notice that while the EPLDs cannot be reprogrammed *in situ*, the EEPLDs can be reprogrammed in place at the expense of slow programming times and high programming voltages.

The Logic Cell Array (LCA), introduced by Xilinx in 1985 [Land85b, CDFH86, Wynn86, Coll86a, Land87, Xili86a to Xili88] is a CMOS static-memory-based user programmable gate arrays. The LCA devices combine the flexibility of gate arrays with the instant (literally) availability of PLDs. The connections of the gate arrays are controlled by the memory cells so that LCAs are reprogrammable. Its density is between 1200 and 9000 gates with a 20,000 gate part announced for 1990.

1.1.3. Prototyping

Simulation verifies that the functions of the circuit according to the circuit definition. It also provides the worst-case analysis and verification of the critical timing path. However, simulation is limited to subsystem level [Shae87]. Prototyping can enhance simulation as the essential part of the circuit design process.

On the other hand, prototyping allows the circuit to be tested in the target system that it is intended for, and facilitates debugging of the circuit in real time, including the unpredictable timing of asynchronous events. Nevertheless, prototype implementations of complex electronic circuits are difficult. The task of building such prototypes can be simplified significantly by the use of electronic breadboarding.

An alternative implementation methodology has been provided by the Logic Cell

Array (LCA) technology, in which the wiring is done electrically rather than mechanically. Furthermore, once the system has been developed by using computer-aided tools, it can then be verified and modified easily, thus permitting the desired concept of analysis-by-implementation to be a realistic goal.

1.2. Motivation

PLDs have been used extensively to reduce the design complexity and turn around time. However, the flexibility of PLDs is limited because of their AND-OR plane architectures. Full-custom ICs can provide design flexibility, speed, reliability, small size, and security of designs, but they are very costly and time consuming. Gate arrays have higher flexibility and density than PLDs, and less design cost and turn around time than full-custom ICs. However, the design cost and turn around time of gate arrays are still much higher than those of PLDs.

LCAs have a great potential by combining the advantages of both PLDs and gate arrays. The density of LCA is up to 9000 or higher equivalent gates. Xilinx provides a development system supporting various schematic capture tools and logic synthesis tools such as PALASM. Although it is useful, PALASM has its limitation in general circuit design. At present, a general HDL is not available as a design entry option for LCA design.

Schematic capture tools have the direct visual representation. However, as the size of the design increases, graphical specifications become more difficult to create and modify. Hardware description languages are becoming increasingly popular as more

programmers become interested in hardware design and more designers become interested in software design. Textual descriptions can be created using a standard text editor, are easy to store and modify, and also serve as written documentation.

The VHSIC Hardware Description Language (VHDL) is an emerging standard for hardware description languages [Arms89]. The VHDL has the ability to describe circuits from as simple as a gate to as complex as a microprocessor. The VHDL is also able to describe the circuits in both behavioural and architectural description. The aim of this thesis is to develop a VHDL compiler integrated with the front end of the Xilinx development system in order to (i) facilitate the LCA design, and (ii) model an electronic circuit design using VHDL and prototype it using LCA instantaneously. This approach merges the advantages of both VHDL and LCA technologies.

1.3. Thesis Objectives

The objectives of this research work are:

1. To study Xilinx Logic Cell Arrays methodology, and its importance in the circuit design process.
2. To study hardware description languages (HDLs) and to select an HDL for LCA.
3. To design a VHDL compiler for LCA design.
4. To implement, test and verify the compiler.

Xilinx provides a circuit-layout tool. Its input language is described by the Xilinx Netlist Format (XNF). The exact format of XNF is proprietary and cannot be described in this thesis, but it is freely available from Xilinx to interested people.

The VHDL compiler developed in this thesis translates an architectural description subset of VHDL into the Xilinx netlist format, which is used by Xilinx circuit-layout tool. The design of the parser of the compiler is accomplished by using parser generation tools in Unix environment, which accept regular expressions of the grammar and produce the parser.

This compiler enables the designer to use hardware description language as the design entry option. Compared with the traditional graphical design entry approach, hardware description languages are easy to create and modify, and serve as documentation of the design. This compiler can also provide an interface with other VLSI design systems so that the LCA can be used for prototyping of the VLSI design. In addition, we anticipate that graphical design system will be unable handle the increasingly complexity of sophisticated system design. For this reason, the VHDL tool developed here will directly contribute to our efforts in promoting system level design space exploration in our laboratory.

1.4. Thesis Structure

Chapter 2 reviews the architecture of Xilinx LCAs, and the methodology of circuit design in LCAs. Chapter 3 gives an overview of VHDL. An architectural description subset of VHDL is defined as the input of the compiler. The output of the VHDL compiler is the Xilinx Netlist Format. Chapter 4 describes the design of the compiling algorithm. Chapter 5 represents the implementation, testing and verification result with two examples. Chapter 6 draws the conclusions of this research work and describes the possible further development and recommendations.

CHAPTER 2

XILINX LOGIC CELL ARRAY

Programmable gate arrays (PGAs) have become increasingly prominent in the past two years, and the signs are that this trend is set to continue [Micr89]. Programmable logic technology has been dominated by devices based on the PLA architecture for the last ten years. As IC technology has advanced, it has become possible to produce larger and larger PLAs. However, the efficiency of PLA utilization decreases as PLA size increases, and the fixed allocation of device pins to array and flip-flop outputs reduces the flexibility of the PLDs. The architecture of gate arrays is being adopted to large programmable devices to solve the problems.

Gate arrays are semi-custom devices based on any array of simple cells surrounded by an interconnection network. In standard gate arrays technology, the interconnection pattern is defined by metallization layers applied at the final stage of manufacture. PGAs dispense with this final stage by possessing a fixed interconnection network which includes programmable crosspoints or switches. The logic cell array (LCA) introduced by Xilinx in 1985 is one of the PGA families. This chapter provides a brief review of the architecture of the LCA, the devices of LCA , the LCA design methodology and the LCA development system.

2.1 Architecture of Xilinx Logic Cell Arrays

The logic cell array family is a group of high-density, high-performance, user-programmable gate arrays [CDFH86, HDKN87, Micr89, Xili89]. The LCA architecture is similar to that of other gate arrays, with an interior matrix of configurable logic blocks (CLBs) and surrounding ring of I/O interface blocks (IOBs). Interconnect resources with programmable routing sources occupy the channels between the rows and columns of logic blocks, and between the logic blocks and the I/O blocks. The architecture is shown in Fig. 2.1. Like a microprocessor, the LCA is a program-driven logic device. The functions of the configurable logic blocks and I/O blocks, and their interconnections, are controlled by a configuration program stored in the on-chip static memory. The crosspoints of the programmable routing sources are turned on or off by memory cells controlled switches.

There are three stages of utilizing an LCA device: i) configuration program generation, ii) configuration, and iii) operation. First, a configuration program is generated from the circuit design. Then, the configuration program is loaded either automatically from an external memory on power-up, or by a microprocessor on command as a part of system initialization. The methods of loading the configuration program are determined by logic levels applied to configuration mode selection pins at the time of configuration. The form of the data may be either serial or parallel, depending on the configuration mode. The programming data are independent of configuration mode selected. Finally, the configured LCA device functions just like an ASIC chip. The LCA device can be reconfigured by loading a new configuration program, since the configuration program is stored in an on-chip RAM like memory.

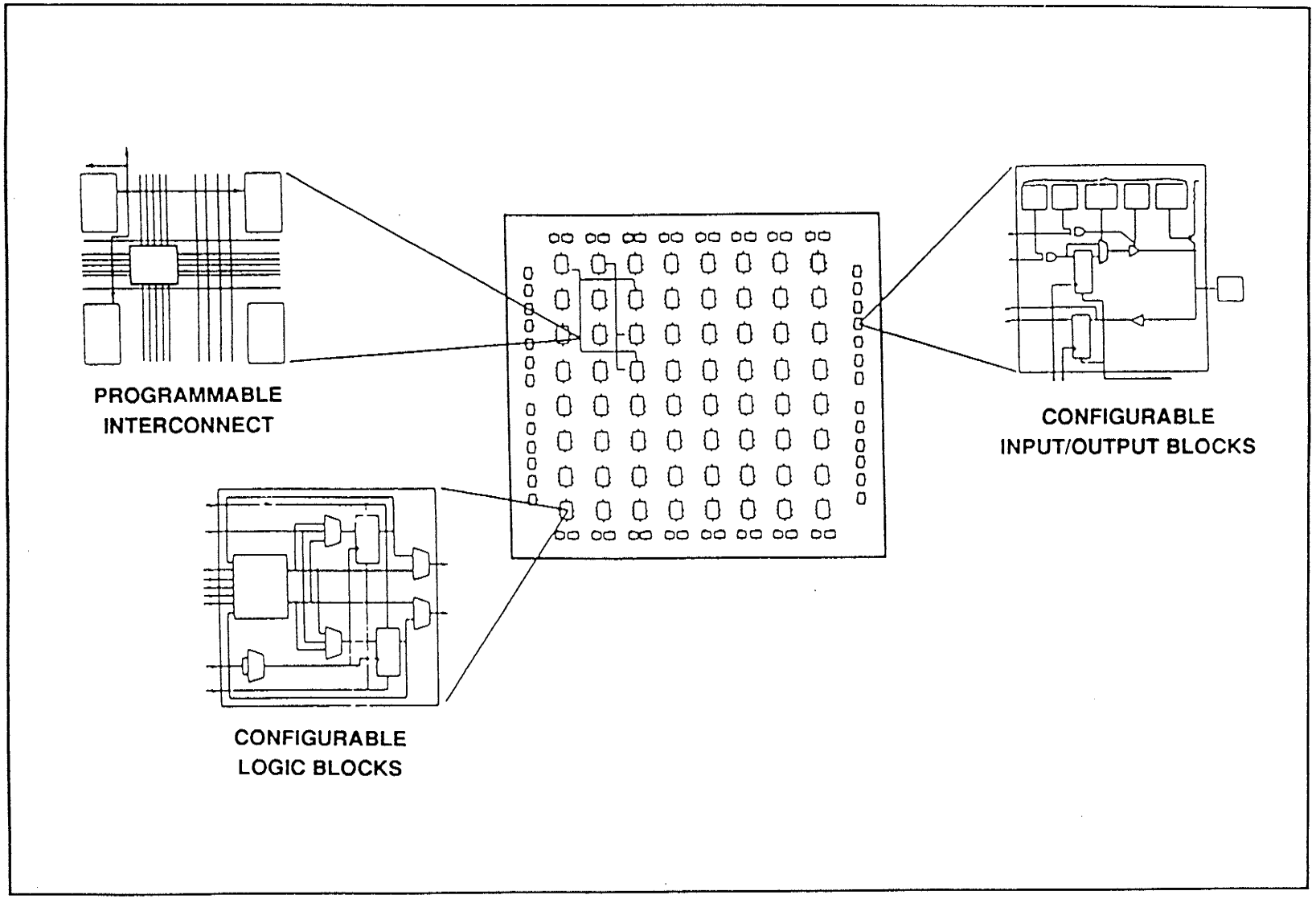

Fig. 2.1. The architecture of Logic Cell Arrays [from Xili89].

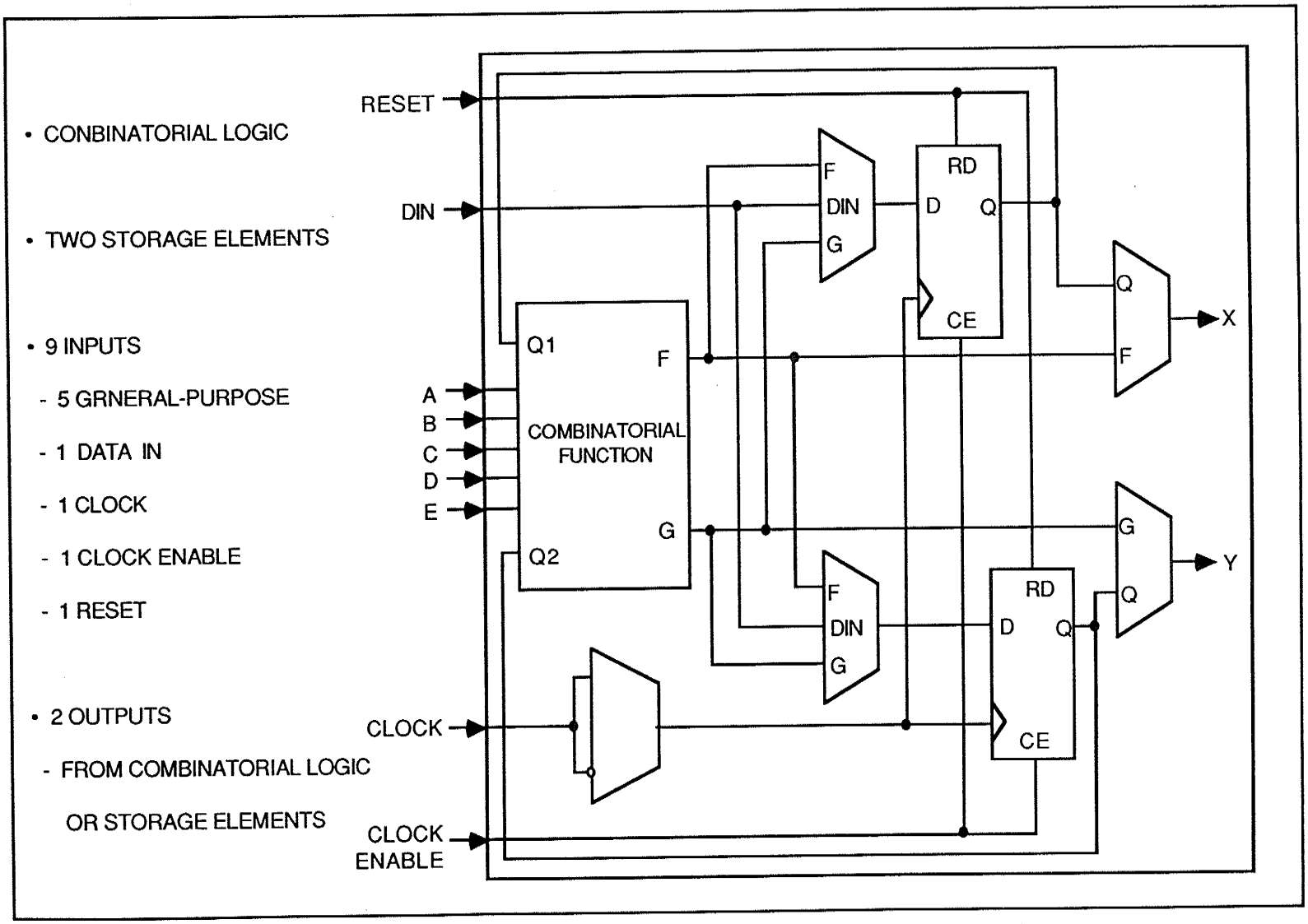

Fig. 2.2. Configurable Logic Block (CLB): 3000 Series [from Xili89].

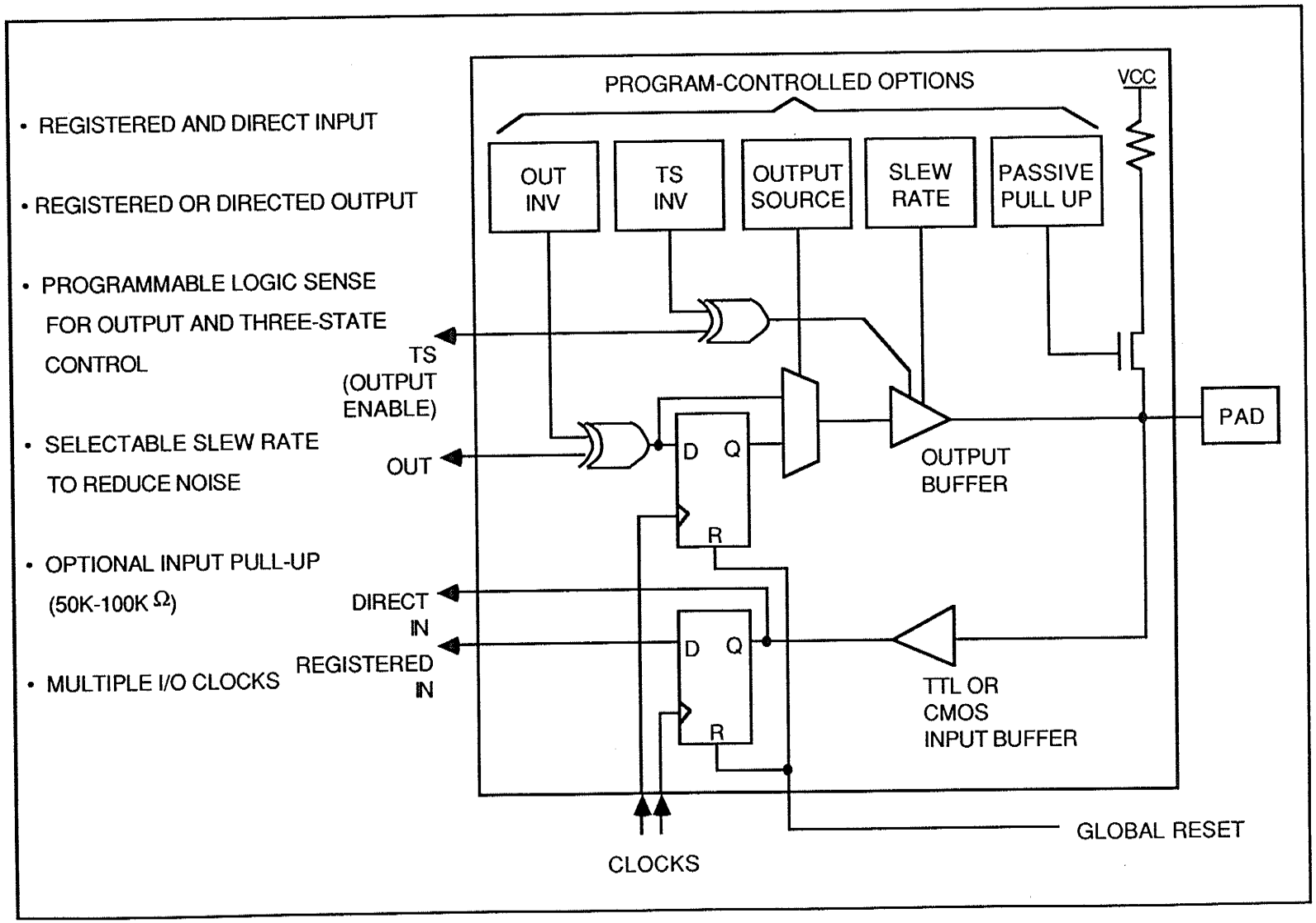

Fig. 2.3. Configurable Input/Output Block (IOB): 3000 Series [from Xili89].

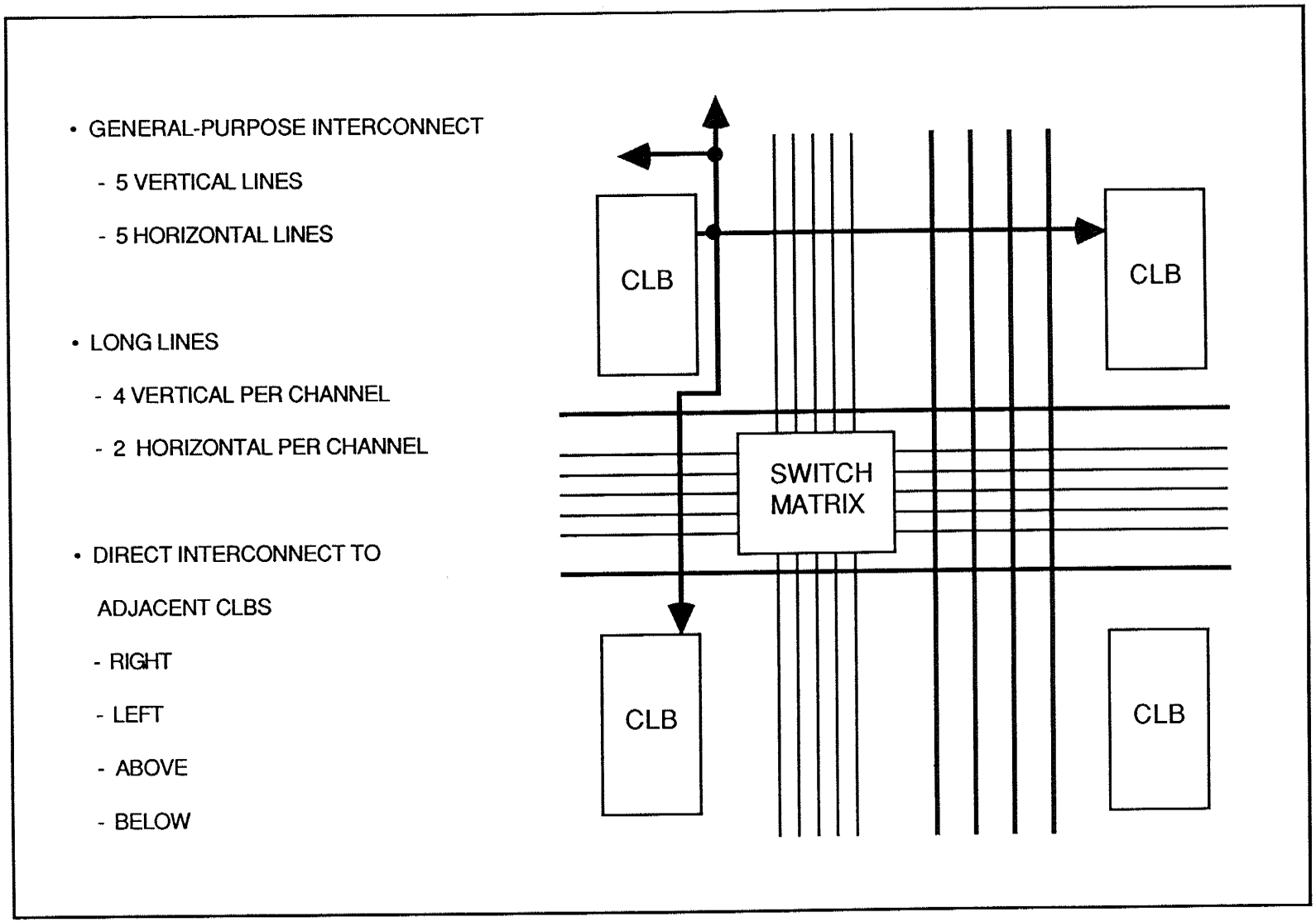

Fig. 2.4. Programmable Interconnect: 3000 Series [from Xili89].

2.1.1. Configurable Logic Blocks

The core of the LCA is a matrix of identical Configurable Logic Block (CLBs). Each CLB contains programmable combinatorial logic and storage registers as shown in Fig. 2.2. The combinatorial logic section of the block is capable of implementing any Boolean function of its input variables. The registers can be loaded from the combinatorial logic or directly from a CLB input. The register outputs can be inputs to the combinatorial logic via an internal feedback path.

2.1.2. Configurable Input/Output Blocks

The periphery of the LCA is made up of user programmable Input/Output Blocks (IOBs). Each block can be programmed independently to be an input, an output, or a bidirectional pin with three-state control as shown in Fig. 2.3. Inputs can be programmed to recognize either TTL or CMOS. Each IOB also includes flip-flops that can be used to buffer inputs and outputs.

2.1.3. Programmable Interconnections

The flexibility of the LCA is due to resources that permit program control of the interconnection of any two points on the chip, as shown in Fig. 2.4. Like gate arrays, the interconnection resources of LCA include a two-layer metal network of general purpose lines that run horizontally and vertically in the rows and columns between the CLBs. Programmable switches connect the inputs and outputs of IOBs and CLBs to nearby metal

lines. Crosspoint switches and interchanges at the intersections of row and columns allow signals to be switched from one path to another. Long lines run the entire length or width of the chip, bypassing interchanges to provide distribution of critical signals with minimum delay or skew. Adjacent CLBs can be connected by direct interconnections.

2.2. Xilinx LCA Devices

The first generation of LCAs introduced in 1985 has been incorporated as two devices in the XC2000 series in density ranging from 1200 to 1800 gates [CDFH86, Micr89, Xili89]. Each XC2000 CLB contains combinational logic and a signal storage element. The combinatorial logic of the block is capable of implementing any Boolean function of up to four variables, or any two independent functions of up to three variables in each. The storage element in the CLB can be configured as an edge-triggered flip-flop or a transparent latch. The XC2000 series IOBs include a register in the input path and a three-state buffer in the output path. The programmable interconnections include direct interconnects, five general purpose lines and two long lines per vertical channel and four general purpose lines and one long lines per horizontal channel.

The second generation of LCA introduced in 1987 is embodied in the XC3000 family [HDKN87, Micr89, Xili89]. It consists of five compatible devices ranging 2000 to 9000 gates densities. The XC3000-series CLB includes a wider combinatorial logic circuit, two storage elements, and dedicated logic that implements clock, reset, and I/O selection functions as shown in Fig 2.2. The nine inputs of the CLB include five general-purpose inputs to the combinatorial logic, and clock, clock enable, reset, and direct data

inputs to the register. The two outputs of the CLB can be driven by the combinatorial logic or the the Q output of the registers. Each combinatorial logic of the CLB can be configured in any one of the three modes. Any single Boolean function of five input variables can be implemented. A second option offers any two independent functions of four variables in each. The third option is to create two independent functions of four variables in each, multiplexed into a single function by the E input to the block. The two registers of the CLB are edge-triggered D-type flip-flops. Clocking is provided by the dedicated clock input of the block; the clock path has an option invert, allowing the flip-flops to be leading or falling-edge triggered at each block. Asynchronous reset and clock-enable inputs are also provided at each CLB. The D input to each flip-flop can be driven by the output of either of the two logic functions that can be generated in the combinatorial circuit, or by a direct data input to the block that bypasses the combinatorial logic.

Each IOB of XC3000 controls one pin. There are two registers in each IOB — one in the input path and one in the output path as shown in Fig. 2.3. Input signals pass through an input buffer to the D input of a register. The input register of the IOB can be configured as an edgetriggered flip-flop or a transparent latch. The output path includes a programmable invert for determining the polarity of the output signal. The signal then goes to the D input of the output register. This register is always an edgetriggered flip-flop. The direct or registered output signal can be sent to the output buffer, which is a three-state buffer.

The programmable interconnections of XC3000 include direct interconnects, five general purpose lines and four long lines per vertical channel and five general purpose lines

and two long lines per horizontal channel as shown in Fig. 2.4.

The XC4000 series, the third generation of LCA announced for 1990, are twice as fast as prior devices, and feature densities as high as 20,000 gates, on-chip static RAM and greatly improved utilization [Goer89, Wils89]. The devices of XC2000 series and XC3000 series are listed in Table 2.1.

Table 2.1. Xilinx Logic Cell Array device list.

DEVICES	XC2064	XC2018	XC3020	XC3030	XC3042	XC3064	XC3090
EQUIVALENT GATES	1200	1800	2000	3000	4200	6400	9000
CLBS (ROW X COL)	64 (8 X 8)	100 (10 X 10)	64 (8 X 8)	100 (10 X 10)	144 (12 X 12)	224 (16 X 14)	320 (20 X 16)
CLFS	128	200	128	200	288	448	640
FLIP-FLOPS	122	174	256	360	480	688	928
IOBS	58	74	64	80	96	120	144
PACKAGES	48DIP 68PLCC 68PGA	68PLCC 84PLCC 84PGA	68PLCC 84PLCC 84PGA	84PLCC 84PGA	84PLCC 84PGA 132PGA	132PGA	132PGA 175PGA
PROM SIZE (BITS)*	11,404	17,096	14,819	22,216	30,824	46,104	64,200
MEMORY (KBYTES)**	2,048	2,048	2,048	2,186	3,584	4,846	6,144

* The PROM size required to store the configuration program for the devices.

** The RAM of the CAD station required to design the devices using the LCA development system.

2.3. LCA Design Methodology

Integrated circuits of the LCA complexity need advanced CAE tools if they are to be used effectively. Like electronic circuit design using CAE tools, the LCA design process is partitioned into three main steps: entry, implementation, and verification [Micr89, Xili89]. The design process for LCA is shown in Fig. 2.5. An integrated development

system for design and implementation of LCA is provided by Xilinx. This provides the user with an effective, convenient, low risk method of logic design entry, simulation, configuration program generation and verification for single chip logic design.

The LCA development system is an open system — the designer can choose from among several popular CAE programs and workstations, allowing the use of existing familiar tools to develop LCA-based design. The development system is available for PC–AT and PS/2 personal computers, and their fully compatible clones. In addition, tools are also available for Sun, Apollo, and Vax/VMS workstations.

Many different schematic editors and simulators are available for entering and verifying circuit designs. To provide a bridge to existing CAE tools, Xilinx has defined an intermediate design description format called the Xilinx Netlist Format (XNF) [Xili88e]. Any design entry tools can be used to enter LCA designs if the output of the design entry can be translated into XNF file [Xili88f]. Similarly, any simulator can be used to verify LCA designs if the XNF file can be translated into the netlist format of that simulator.

2.3.1. Design Entry

An LCA design can be entered by using schematic capture, Boolean equations, or state machine equations in PALASM format. Xilinx supports various schematic capture tools, such as FutureNet DASH, Dasiy AIE or EED II, Mentor IDEA and OrCAD SDT. The design entry is translated into an XNF file. Other design entry methods can be supported with an appropriate XNF translator.

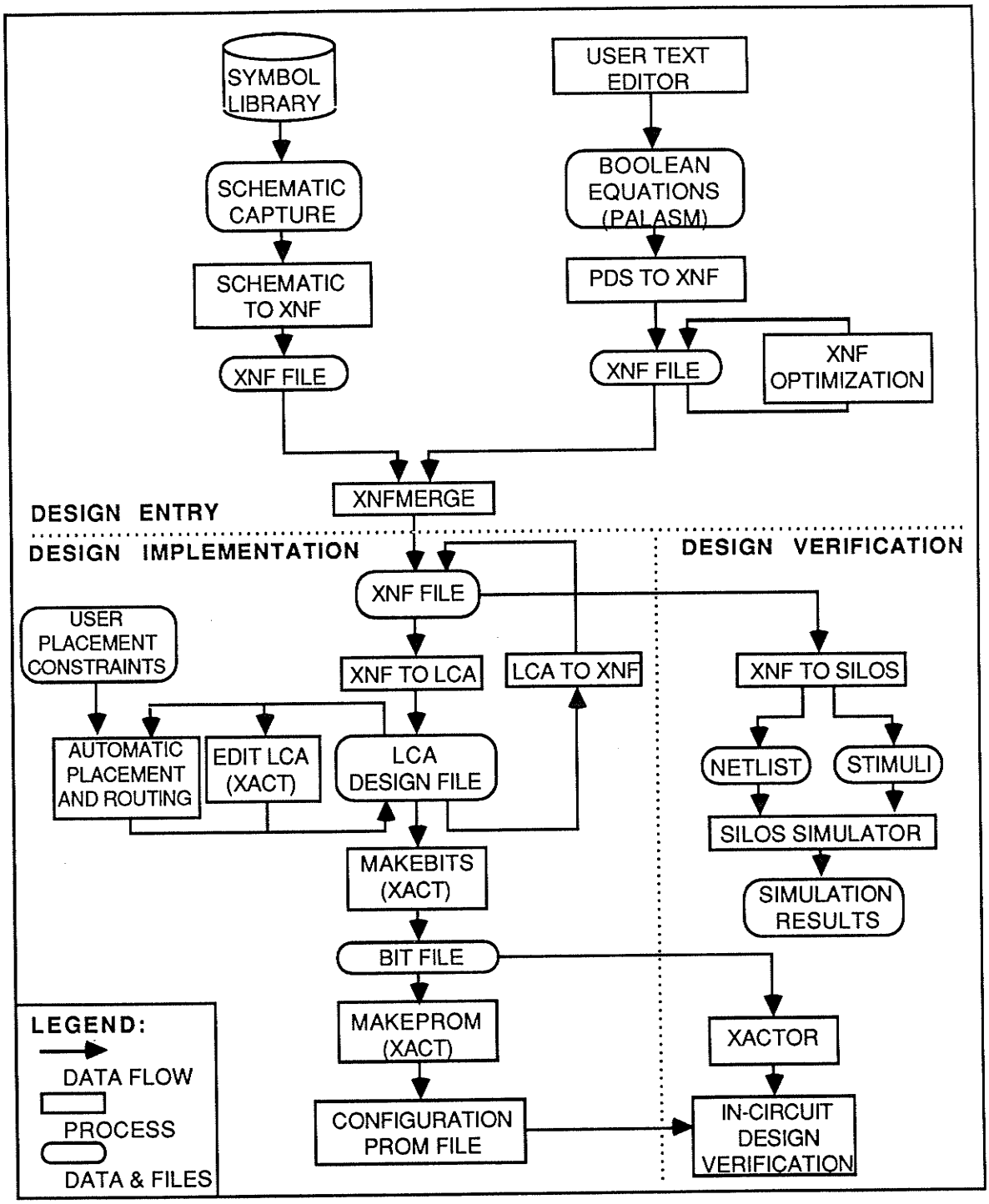

Fig. 2.5. LCA design process[after Xilinx].

Designers typically enter the designs hierarchically, by first creating a top-level description that defines the application in terms of major functional blocks. Lower-level descriptions decreasing the logic in each of the larger blocks are then entered, using marcos defined by the designer or supplied in the standard library. There is no limit to the number of levels within the hierarchical design. Two design examples will be seen later in Chapter 5.

2.3.2. Design Implementation

Implementing designs in LCA entails mapping the desired logic into the CLBs and IOBs of the LCA architecture. The automatic LCA implementation tools include logic reduction and logic partitioning, placement and routing, and design editor and configuration program generation [Micr89, Xili89]. Once a design has been translated into an XNF file, the design is mapped into the programmable resources of the LCA by a program called XNF2LCA. The XNF2LCA program performs two operations: logic reduction and logic partitioning. Logic reduction is the process of deleting unused logic from the design, permitting liberal use of the library, without any penalty when a macro has unused functions. The remaining logic is automatically partitioned into pieces that can be implemented within individual CLBs and IOBs. As much logic as possible is grouped into each block. The result is a design file called LCA.

Each CLB and IOB of the design is "placed" by assigning it to one of the discrete blocks within an LCA and is "routed" by specifying the programmable interconnection paths used to implement the connections among the blocks. Placement and routing can be

performed automatically with placement and routing software, interactively with graphic based design editor, or combination of the two. The automatic placement and routing (APR) program searches for the optimum placement by using the simulated annealing algorithm and then routes the nets that interconnect the blocks.

The core of the development system is the LCA design editor XACT. The XACT provides the designer with the ability to graphically enter, place, route, and manipulate LCA designs, and to generate the LCA configuration program data after the placement and routing. The generated configuration program data can be download through the download cable, which provides a convenient approach to program a breadboard or prototype unit.

2.3.3 Design Verification

After the implementation of the design, its operation must be verified. Both APR and XACT calculate the worst-case delays through both the logic block and the routing resources. A simulator PC-SILOS is also available from Xilinx with a converter that converts the XNF to the SILOS simulation netlist and a dummy simulation stimulus file. The stimulus file is tailored to define the input pattern and which signals to be examined and with what timing. The XNF file may come from the design entry or form an implemented design by converting the design to XNF file. Designs which have not been routed can be simulated with unit delay data. Routed designs include worst-case timing of functional blocks and their interconnections.

The verification can also be done by in-circuit testing tools such as Xilinx

XACTOR, or by IC testing devices such as AXIC tester.

2.4. Summary

The LCA technology provides a fast turn around time and high flexibility, and allows users to implement ASIC design easily and quickly. It also provides a practical approach for ASIC breadboarding. For educational purposes, the Xilinx development system demonstrates the whole process of digital system design. In this environment, an HDL should be available as an option for design entry. Since the development system is an open system with XNF as the interface, an HDL compiler can be developed to generate the XNF file.

CHAPTER 3

THE VHSIC HARDWARE DESCRIPTION LANGUAGE

The VHSIC Hardware Description Language (VHDL) is a newly-adopted IEEE standard hardware description language that supports architectural, dataflow and behavioural styles of design and documentation for digital systems. A number of CAE tools have been developed to support VHDL [AAIR88, LCAC88, Saun87]. The motivation for the development of the VHDL CAE tools has been the need for a standard medium of communication for transmitting hardware design data from one organization to another [DeGa86]. Such communication is necessary for the following four reasons [Shah86]: (i) from the viewpoint of the hardware component vendor, it allows formal specification of the behaviour of the component; (ii) from the viewpoint of a component user, it allows formal specification of the functionality required of the component for procurement purposes; (iii) from the viewpoint of the hardware design engineer, it provides a standard for sharing information within a design team, even among designers working at different levels of abstraction; and (iv) from the viewpoint of the CAE tool developer, it provides a wider user-base for tools, thereby creating a better return on investment.

Beginning in 1983, the U.S. Department of Defense sponsored the development of the VHSIC Hardware Description Language (VHDL). The original

intent of the language was to serve as a means of communicating designs from one contractor to another in the Very High Speed Integrated Circuit (VHSIC) program. However, the design of the language has received input from many individuals in computer industry and thus reflects a consensus of option as to what characteristics a hardware description language should have [Arms89].

In August 1985, Version 7.2 of the language was released by the Department of Defense, representing the completion of the first major stage of the language development. The IEEE sponsored a further development of VHDL, with the goal to establish a standard version of the language. In June 1987, eligible IEEE members voted to accept Version 7.2 as the standard version and in December 1987, it was officially so designated by the IEEE [VHDL88]. This chapter gives a brief overview of IEEE standard VHDL as well as a subset of VHDL for LCA design.

3.1. Features of VHDL

VHDL allows design and documentation of digital circuits from the system level to the gate level, and supports bottom-up as well as top-down design methodologies [Meye89]. Although designed to be independent of any underlying technology, design methodology, or environment tool, the language is extendible to various hardware technologies, design methodologies, and the varying information needs of design automation tools. In VHDL, not only the architecture of a circuit is defined, but its organization can be defined as well. The integration and unification of architecture and organization information is provided in VHDL.

VHDL describes the functionality and organization of hardware system at various levels of abstraction. The concept of *design entity* is the primary abstraction mechanism of the language [LMSh86]. Starting with design entity, we discuss the basic features of VHDL in this section.

3.1.1. Design Entities

One of the characteristics of hardware devices is that their functionality can be defined for the most part independent of the environment in which they operate. This characteristic allows components from various sources to be wired together to create new and more complex designs. VHDL reflects this characteristic in its overall organization, by emphasizing the ability to describe isolated components, called *design entities*, which can then be combined with other component descriptions to form more complex descriptions. In VHDL, a design entity consists of an interface description and one or more alternative body descriptions.

3.1.2. Interface Description

The interface contains a set of definitions common to alternative bodies. Such definitions capture the external view of hardware entity and specify communication channels between the design entity and the outside world. An example of interface description for a full adder is shown in Fig. 3.1. The interface description names the entity and describes its inputs and outputs. The port description of interface declares signals visible externally, including the mode of the signals (e.g., **in** or **out**) and the

type of the signals (e.g., BIT). BIT is a predefined signal type in VHDL.

```
entity FullAdder is
        port (X, Y, Cin: in Bit; Sum, Cout: out BIT);
end FullAdder;
```

Fig. 3.1. Interface description of a full adder.

3.1.3. Body Descriptions

After the interface description, the specification of the behaviour of the entity is required. In VHDL, three styles of body descriptions are possible: architectural, dataflow, and behavioural, as shown in Fig. 3.2. The three styles can be combined within an architectural body description. In general, the three styles are used separately as alternative body descriptions for a design entity.

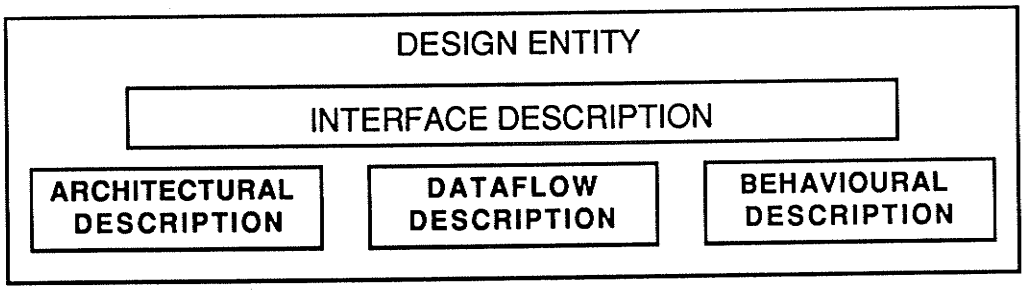

Fig. 3.2. Body descriptions of design entity.

3.1.3.1. Architectural Description

The architectural description captures the schematic view of hardware and consists primarily of interconnected components. The behaviour of the components is

externally defined. The architectural description involves (i) component declaration, which defines interface to components used in a design, and (ii) component instantiations, which create one or more instances of a declared component. An example of the architectural description of the full adder is shown in Fig. 3.3. Note that each instantiation has a unique label associate with it as well as a port map. The port map creates an association between the inputs and outputs of the components declared and the instantiation components by positions.

```
architecture Structure of FullAdder is

    component XOR2 port (Ain, Bin, Output);
    end component;
    component AND2 port (Ain, Bin, Output);
    end component;
    component OR2 port (Ain, Bin, Output);
    end component;

    signal A, B, C: Bit;

begin
    ADD0:     XOR2 port map (X, Y, A);
    ADD1:     XOR2 port map (A, Cin, Sum);
    CARRY0:   AND2 port map (A, Cin, B);
    CARRY1:   AND2 port map (X, Y, C);
    CARRY3:   OR2 port map (B, C, Cout);
end Structure;
```

Fig. 3.3. Architectural body description of a full adder.

3.1.3.2. Dataflow Description

The dataflow description specifies data transforms being performed in terms of concurrently executing register transfer level statements. It is more abstract than the architectural description.

One advantage of the architectural description is the ability to express the parallelism inherent in hardware operation. In contrast, purely architectural description cannot express what each component actually does, since component behaviour is externally defined. However, the dataflow description allows the simultaneous expression of parallelism and behaviour. For example, AHPL is a register transfer level language which has been used widely in industrial and educational area [HiPe87].

```
architecture Dataflow of FullAdder is

        signal A, B: Bit;

begin
        A <= X or Y;
        B <= A and Cin;
        Sum <= A xor Cin;
        Cout <= B or (X and Y);
end Dataflow;
```

Fig. 3.4. Dataflow description of a full adder body.

An example of dataflow description for the full adder is shown in Fig. 3.4. Dataflow descriptions are created through the use of concurrent signal assignment statements. Such statements execute in response to event or changes in signal values referenced within the statements.

3.1.3.3. Behavioural Description

The behavioural description, the most abstract style, specifies data transforms in terms of algorithms for computing output responses to input changes. Component networks modeled with concurrent signal assignment statements represent both design

architecture (in terms of component interconnection) and design behaviour (in terms of data transforms performed). However, it is sometimes necessary to describe behaviour without biasing that description toward a particular implementation. A pure behavioural description, entailing little or no architectural information, is more appropriate in such situations. For example, Mentor has developed the Behavioural Language [Ment84]. In VHDL behavioural description can be written by using process statement. An example of a behavioural description for an AND gate with its interface description is shown in Fig. 3.5.

```
entity AndGate is
   port ( inputs: in Bit_Vector(1 to 2);
          Result: out Bit);
end AndGate;

architecture Behaviour of AndGate is

begin
   process (Inputs)
          variable Temp: Bit;

   begin
          Temp := '1';
          for i in Inputs`Range loop
                 if Inputs(i) = '0' then
                        Temp := '0';
                        exit;
                 end if;
          end loop;
          Result <= Temp after 10ns;
   end process;

end Behaviour;
```

Fig. 3.5. Behavioural description of an AND gate with interface description.

A process statement contains a declarative part and a statement part. However, only sequential statements such as if, case, and loop statements can be written in its statement part. Such statements describe algorithms that specify how a component will respond to changes on input signals by causing changes on output signals.

A process statement is initially sensitive to an associated list of signals. An event on any of those signals will cause the process to execute, and will potentially cause the process to modify certain output signals.

3.1.4. Types

VHDL is strongly typed and supports a variety of data types. The data type classification is shown in Fig. 3.6. VHDL also allows the designer to define new data types as they are needed. For example, two new types BYTE and WORD can be created using type declaration as shown in Fig. 3.7.

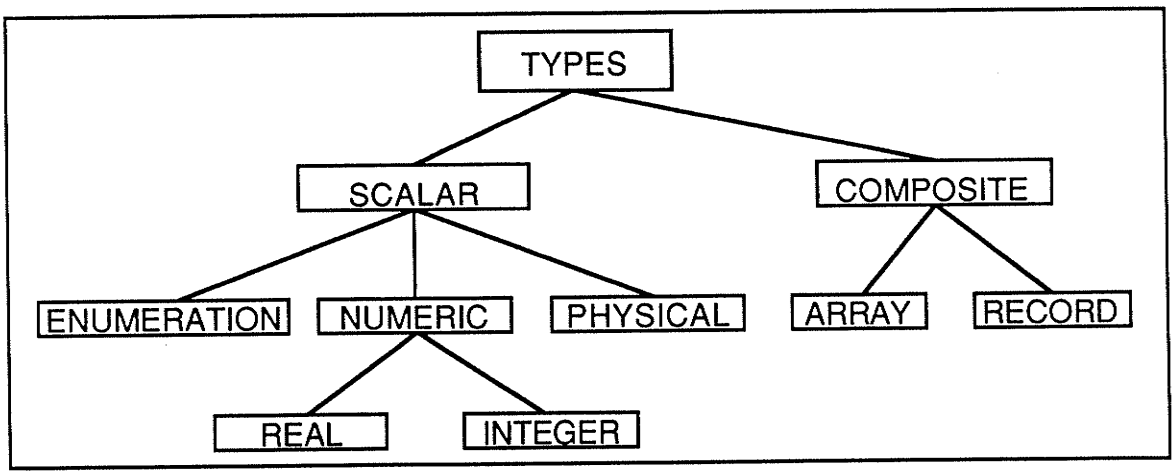

Fig. 3.6. VHDL type classification tree [after Arms87].

```
type BYTE is array (0 to 7) of BIT;
type WORD is array (0 to 15) of BIT;
```

Fig. 3.7. Type declaration.

INTEGER and REAL are two predefined numeric types with the standard numeric operation defined upon them; designers may define additional numeric types by specifying their range of values. Enumeration types include values that are either character literals or identifiers. VHDL predefines several enumeration types including type BOOLEAN, type BIT, and type CHARACTER. Standard logic operations are available on BOOLEAN or BIT objects. Designers may also define their own enumeration types. Physical types allow expression of measurements. A physical type declaration specifies a set of units, all defined in terms of some based unit, and all measuring the same quantity.

The two classes of composite types in VHDL are array and record types; elements of an array type must be the same, whereas elements of record type may differ. Predefined array types BIT_VECTOR and STRING represent arrays of bits and arrays of characters respectively. Logic operations defined on BITs are defined on BIT_VECTORs as well.

3.1.5. Signals

VHDL supports architectural and dataflow descriptions through the use of signals. Signals can be used to represent wires or buses in a architectural description or to represent data transmissions in a dataflow description. Signals may retain state and

as such may be used to represent memory elements such as flip-flops and registers.

Signals may have different types. Before their use, signals must be declared to define their names and the types associated with them. As shown in Fig. 3.8, an example of signal declaration signals RESET and CLOCK with type BIT, INPUTS and OUTPUTS with type BYTE, and ADDRESS_LINES with type WORD.

```
signal RESET, CLOCK : BIT;
signal INPUTS, OUTPUTS : BYTE;
signal ADDRESS_LINES : WORD;
```

Fig. 3.8. Signal declaration.

3.1.6. Packages

For frequently used declarations, a package may be created to avoid writing declarations repeatedly when they are needed. The package has a name associated with it. A package can be used to share declarations among many other design units, or collect declarations relating to a particular abstraction. Declarations in a package may be made visible by referring to the package. A package declaration example is shown in Fig. 3.9.

```
package BYTE_WORD is
        type BYTE is array (0 to 7) of BIT;
        type WORD is array (0 to 15) of BIT;
end;
```

Fig. 3.9. Package declaration.

3.2. Supported Subset

VHDL is a very complicated language. It is very difficult to develop a compiler which is able to accept the whole set of VHDL, especially for the decomposition from behavioural description to architectural description. It is a better approach to divide the task into several stages. We choose to develop a compiler that accepts a VHDL architectural description subset as the first step. Further development can be built up on this stage.

The VHDL subset supports the interface and architectural body description. Package declaration is also supported so that a group of type declarations can be referred by more than one entity. Pin declarations and several predefined signal types are added into the subset to support the LCA design. A User's Guide and the Backus–Naur (BN) form syntax description of the supported subset can be found in Appendix B and Appendix C.

3.2.1. Input/Output Pin Descriptions

In standard VHDL, the physical positions of interface ports cannot be specified. However, the I/O pin numbers of a design in LCA have to be explicitly defined for implementation using the LCA development system. If a I/O signal is not connected with a IOB or pin of the LCA, the logic partitioning and reduction program considers this signal no source. Thus, the signal and its loads are reduced. For this purpose, a pin description clause is added into interface description after the port description. For example, the full adder interface with its pin association description is

shown in Fig. 3.10.

```
entity FullAdder is
        port (X, Y, Cin : in Bit; Sum, Cout : out Bit);
        pins (X:    P11;
              Y:    P13;
              Cin:  P24;
              Sum:  P34;
              Cout: P35);
end FullAdder;
```

Fig. 3.10. Interface description of a full adder.

3.2.2. Types for LCA Design

Signals can have flags in an LCA design. The flags are used to control the automatic placement and routing (APR) process [Xili88f, Xili89]. "L" is used to tell the APR to route a signal through a long line across the row or column of the array. "C" tells APR that it is a critical route so that APR gives this signal the highest priority. "N" tells APR that it is a non-critical route so that APR gives it the lowest priority. "X" assures that this signal stays outside of CLBs. To cope with those flags, four predefined types BIT_L, BIT_C, BIT_N, and BIT_X are added into VHDL, corresponding to the flags L, C, N, and X.

3.3. Summary

This Chapter describes a sophisticated hardware description language, capable of supporting architectural, dataflow, and behavioural styles design and documentation for digital circuits from a single gate to a complex system. It is too

difficult to develop a compiler for the whole set of the language. A subset of VHDL needs to be adopted for a class of circuits design such as LCAs. As the first step, A VHDL architectural description subset is chosen for LCA design.

CHAPTER 4

A VHDL COMPILER FOR LCA DESIGN

Various VHDL design systems have been implemented to support the design of digital systems. There are basically two approaches to implement such a VHDL design system [Gilm86, Mars88]: i) a full-scale VHDL design system [Saun87], including VHDL analysis to compile the VHDL design files into a design library, synthesis to create the hardware layout and simulation to verify the design; and ii) an embedded VHDL design system in which the VHDL compiler is combined with an existing CAE system [AAIR88]. In this system, VHDL is used as an approach for design entry, with the compiler generating a description format acceptable by the CAE system, while the CAE tools are used to implement and verify the design originally described in VHDL.

The second approach has several advantages, such as the availability of simulation model library, schematic capture and fault simulation, and a large existing user base. Since we have a dedicated CAE system for LCA design, the second approach is selected in which the VHDL compiler for LCA design is embedded within the existing CAE tools.

The position of the compiler in the LCA design process as shown in Fig. 2.5. is shown by the shadowed area in Fig. 4.1. As we described in Chapter 2, the LCA development system supported by Xilinx has an interface with the high level entry and the

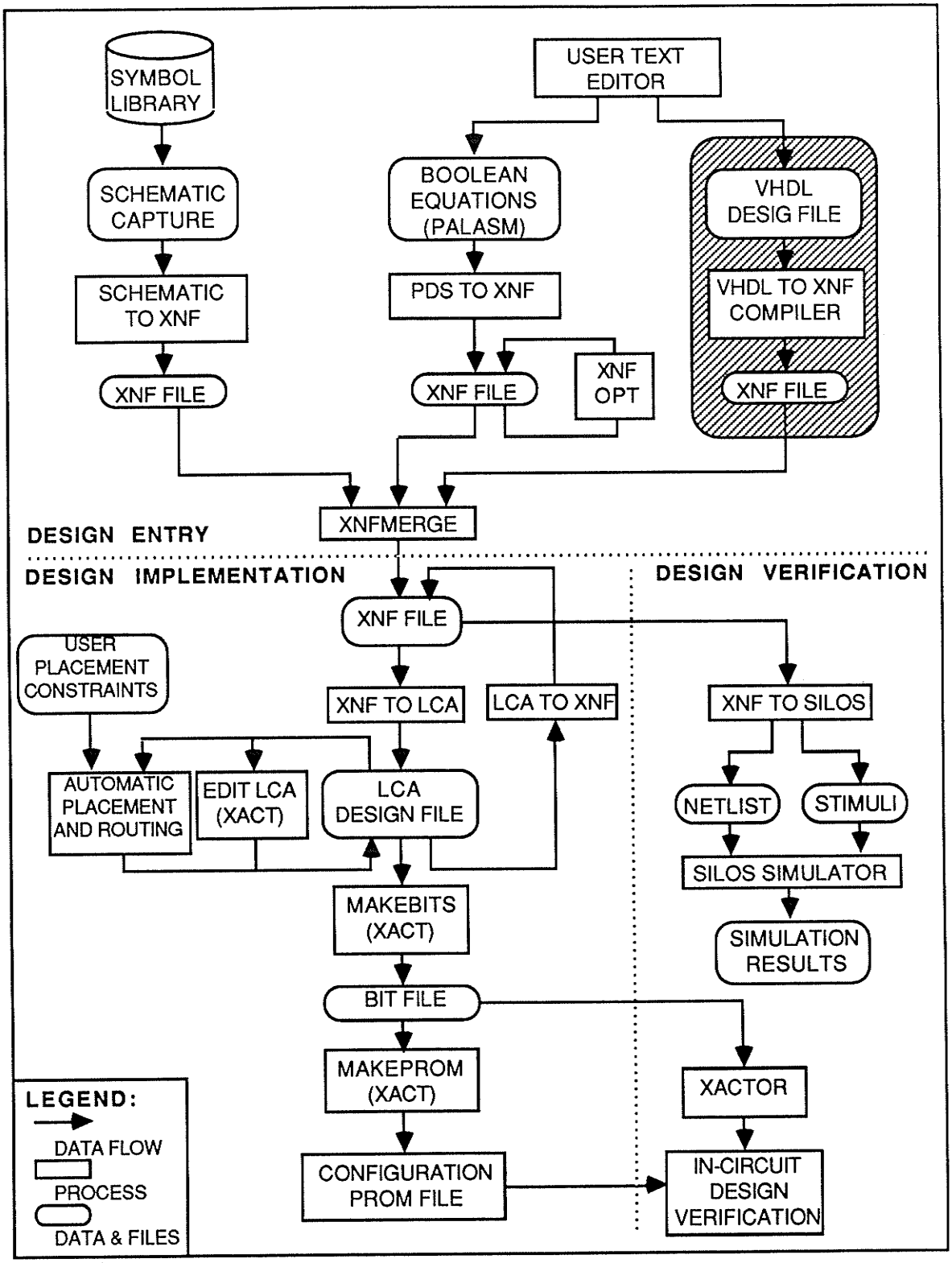

Fig. 4.1. The position of the VHDL compiler in the LCA design process.

simulation tools. The Xilinx Netlist Format (XNF) is accepted by the LCA development system as an interface for implementation and is generated by the LCA development system for simulation.

The VHDL compiler for LCA design is a one-pass compiler supporting a subset of standard VHDL for hierarchical architectural descriptions. It accepts VHDL design files and compiles the design files into the XNF files. Then, the design implementation and simulation are accommodated using the existing LCA development system. The structure of the compiler is shown in Fig. 4.2. The compiler consists of the following five components: a lexical analyzer, parser, stack machine, netlist generator and flattener.

VHDL design files are typed in and edited using a standard text editor such as VI in UNIX system. The lexical analyzer reads the VHDL design files and converts them into meaningful lexical chunks called tokens. The parser calls lexical analyzer to obtain the tokens and checks the syntax of the VHDL design files. If there are any syntax errors, they are reported by the parser. Otherwise, the parser generates a code for the stack machine. The code executed on the stack machine is a linear array of the instructions of the stack machine and parameters for the instructions. The stack machine generates an internal VHDL representation (IVHDL), and also reports any semantic errors. In the netlist generation stage, first the IVHDL is converted into an internal XNF (IXNF) representation which is subsequently translated to XNF file. Both IXNF and XNF are formats supported by Xilinx [Xili88e, Xili88f]. The XNF files in different hierarchical levels are stored in the design library. Each design entity in VHDL results in an XNF file in the design library. A standard library can also be established by adding the XNF files which are proved to

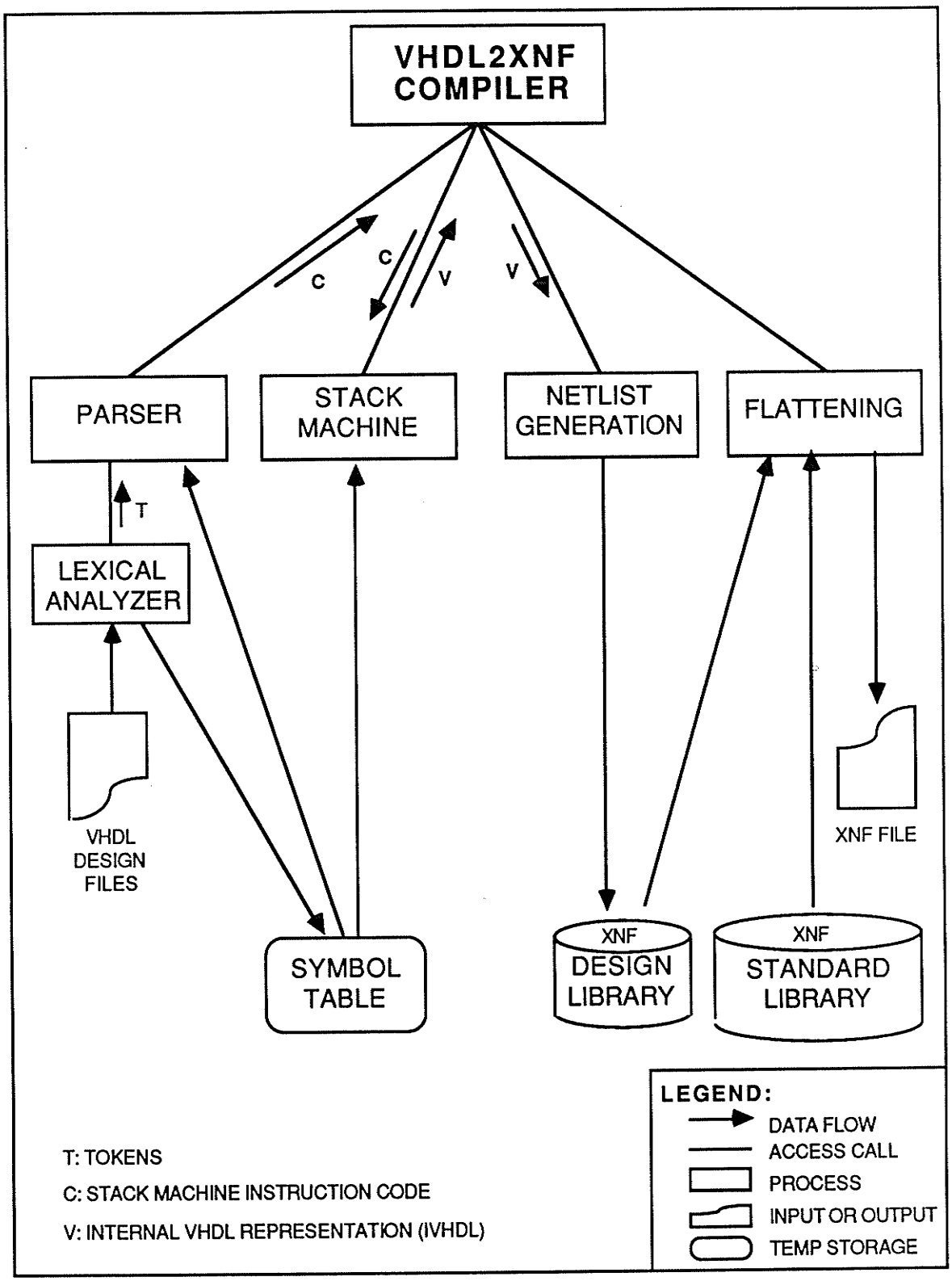

Fig. 4.2. The structure of the VHDL compiler for LCA design.

be eligible designs. Current standard libraries are provided by Xilinx. The library list is included in Appendix E. A program supported by Xilinx flattens the hierarchical XNF files. The flattening program reads the XNF files in the design library and the XNF files of the standard components in standard library, checks the syntax of the NXF files, and generates a flattened XNF file, which is used for implementation of the design.

4.1. Lexical Analysis

The lexical analysis is the first phase of the compiler. The lexical analyzer is a subroutine of the parser as shown in Fig. 4.2. Its main task is to read the VHDL design files and produce as output a sequence of tokens that the parser uses for syntax analysis. Each token represents a logically cohesive sequence of characters or string, such as an identifier, a keyword, a punctuation character, or an operator. In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a *pattern* associated with the token. The pattern is said to match each string in the set. The string forming a token is called the lexeme for the token. The lexemes are stored in a symbol table for later use. The lexical analysis uses the patterns to identify the tokens. Upon receiving a "get next token" request from the parser, the lexical analyzer reads the input files until it can identify the next token, and then passes the token to the parser.

The secondary task of the lexical analyzer includes stripping out comments and white spaces in the form of blanks, tabs, and newline characters from the VHDL design files, as well as counting line numbers which can be used in association with error

messages. The structure of the lexical analyzer is shown in Fig. 4.3.

There are tools for constructing lexical analyzer from special-purpose notations based on regular expressions. LEX [LEX86], a lexical analyzed generator tool is used to generate the lexical analyzer of the VHDL compiler.

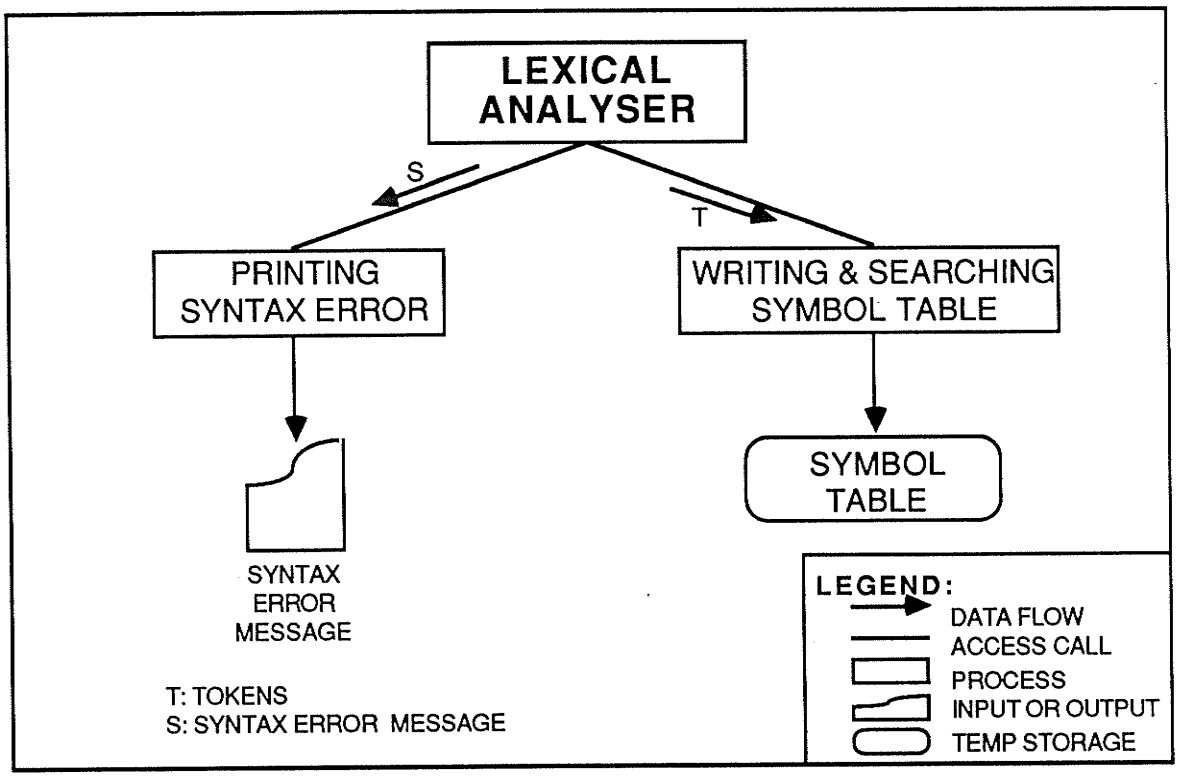

Fig. 4.3. The lexical analyzer of the VHDL compiler.

4.1.1. LEX — A Lexical Analysis Generator

LEX is used in the following manner. First, a specification of the lexical analysis is prepared by creating a program using regular expressions to describe all the patterns and fragments of C programs to be executed when a matching string is found. Then, the program is run through the LEX to produce a C program lex.yy.c. The program lex.yy.c

is the pattern-matching algorithm of the lexical analyzer.

The specification of the lexical analysis consists of three parts:

declarations

%%

patterns and actions

%%

auxiliary procedures

The declarations section includes declarations of variables, constants, and definitions. The patterns and actions are statements of the form:

p_1	$\{action_1\}$
p_2	$\{action_2\}$
...	...
p_n	$\{action_n\}$

where each p_i is a regular expression and each $action_i$ is a program fragment describing what action the lexical analyzer should take when pattern p_i matches a lexeme. In LEX, the actions are written in C; in general, however, they can be any implementation languages. The third section holds whatever auxiliary procedures are needed by the actions. The specification of the lexical analyzer can be found in Appendix F.

When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p_i. Then, it executes $action_i$. Typically, $action_i$ will return the control and the token to the parser. If the token is an identifier, the $action_i$ also

searches the symbol table and writes the identifier into the symbol table if it is not in the table.

4.2. Parsing

The second phase of the compiler is syntax analysis or parsing. The parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 4.2, and verifies that the string can be generated by the grammar for the source language. After the tokens pass the syntax checking, they are used to generate the output of the parser. The output of the parser of the VHDL compiler is a sample stack machine code which is a list of instructions and the parameters for the instructions. The tokens are used as the parameters in the code. The parser is also expected to report any syntax errors. The structure of the parser is shown in Fig. 4.4.

There are many different algorithms for syntax analysis and tools to generate parsers. Every programming language has rules that prescribe the syntactic structure of the well-formed programs. The syntax of programming language constructs can be described by context-free grammar or Backus-Naur Form (BNF) notation. YACC [YACC86], a parser generator that accepts a context-free grammar description and generates an LALR parser, is used to generate the parser of the VHDL compiler.

4.2.1. YACC — Yet Another Compiler Compiler

A parser can be constructed using YACC by creating a file containing a

specification of the parser first. Then, YACC transforms the file into a C program called y.tab.c using the LALR method [ASU186]. The program y.tab.c is a representation of an LALR parser, along with other C routines.

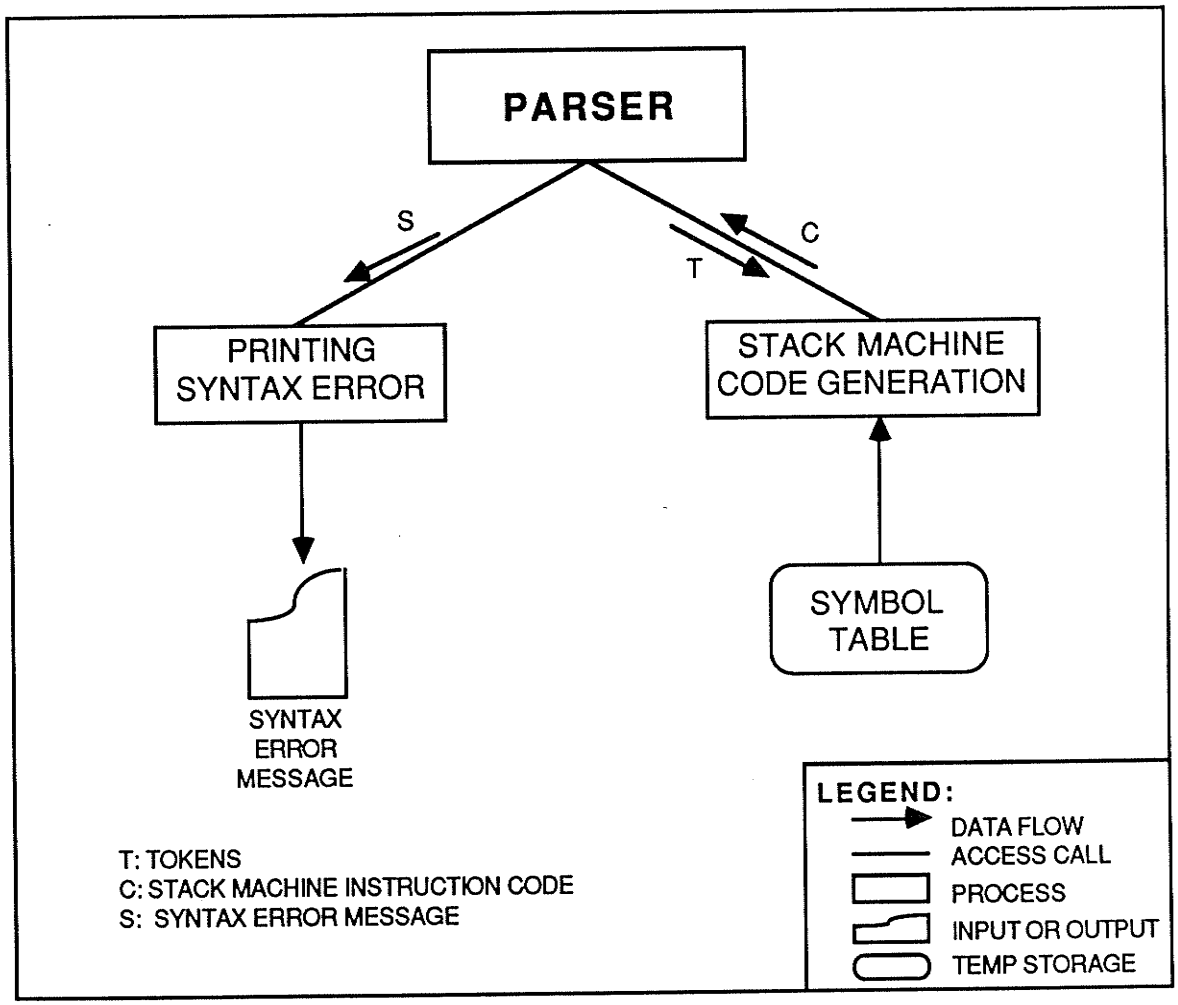

Fig. 4.4. The parser of the VHDL compiler.

A specification of the parser has three parts:

%{

C language declarations

YACC declaration: lexical tokens, grammar variables,

precedence and associativity information

%%

grammar and actions

%%

supporting C routines

The specification of the parser of the VHDL compiler can be found in Appendix E.

Alternate rules are separated by "|". Any grammar rule can have an associated action, which will be performed when an instance of that rule is recognized in the input. An action is a sequence of C statements enclosed in braces { and }. Within an action, $n (that is, $1, $2, etc.) refers to the value returned by the *n*-th component of the rule, and $$ is the value to be returned as the value of the whole rule.

The parser of the VHDL compiler has basically one kind of action which is to add new instructions and tokens as parameters in the code for the stack machine. Different grammar rule recognition result in different instructions added to the stack machine code.

4.3. Stack Machine

After the parser process, the result is a list of instructions for a stack machine. The stack machine is a simple computer. When an operand is encountered, it is pushed onto a stack; most operators operate on items on the top of the stack.

A stack machine results in simple compiler. It is just an array containing operators

and operands. The operators are the machine instructions; each is a function call with its arguments, if any, following the instructions. Other operands may already be on the stack. A stack machine is also easy to modify and expand by changing the instruction functions, or adding new instructions.

The structure of the stack machine is shown in Fig. 4.5. For example, to handle a signal declaration

signal X: BIT;

the following code is generated by the parser:

CODE	COMMENTS
idpush	*Push symbol table pointer onto stack;*
X	..... *identifier X;*
idpush	*Push symbol table pointer onto stack;*
BIT	.....*predefined type BIT;*
scode	*Create a Signal record to store the declaration.*

In the code, the boldface strings are stack machine instructions and the others are the parameters for the instructions. When this code is executed on the stack machine, the result is an internal VHDL representation of the signal declaration.

4.3.1. Stack Machine Instructions

The stack machine of the VHDL compiler has 46 instructions. When the VHDL subset is to be expanded, new instructions can be easily added into the existing instruction set. The instructions and their brief function descriptions are shown in Table 4.1.

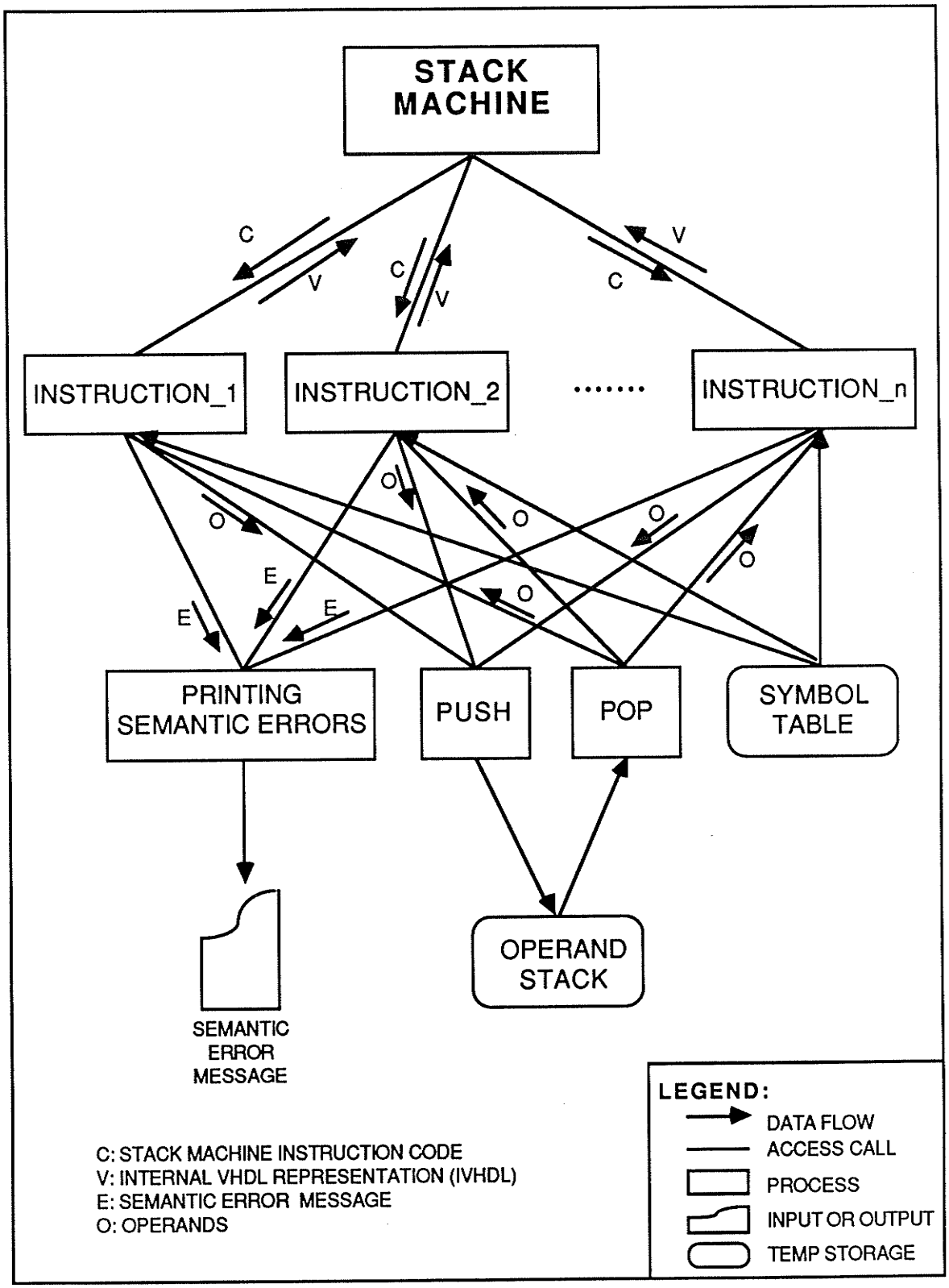

Fig. 4.5. The stack machine of the VHDL compiler.

Table 4.1. The stack machine instructions.

INSTRUCTION	FUNCTION	INSTRUCTION	FUNCTION
INTPUSH	PUSH INTEGER ONTO STACK	AND	LOGIC AND
IDPUSH	PUSH IDENTIFIER ONTO STACK	NAND	LOGIC NAND
POP	POP ID OR INT FROM STACK	OR	LOGIC OR
PCODE	PROCESS PACKAGE	NOR	LOGIC NOR
CHKPL	CHECK PACKAGE NEME	XOR	LOGIC XOR
UCODE	PROCESS USE-CLAUSE	EQ	EQUAL
ECODE	PROCESS ENTITY	NE	NOT EQUAL
CHKE	CHECK ENTITY NAME	GT	GREATER THAN
PPCODE	PROCESS PINS	GE	GREATER OR EQUAL
ACODE	PROCESS BODY	LT	LESS THAN
CHKA	CHECK BODY NAME	LE	LESS OR EQUAL
TCODE	PROCESS TYPE	NEGATE	NEGATIVE
CDCODE	PROCESS COMPONENT DECLA.	ADD	ADDITION
FPCODE	PROCESS FORMAL PORT	SUB	SUBTRACTION
IDFIRST	PROCESS IDENTIFIER	MUL	MULTIPLICATION
IDLIST	PROCESS IDENTIFIER LIST	DIV	DIVISION
SCODE	PROCESS SIGNAL	MOD	MODEL
CICODE	PROCESS COMPONENT INST.	REM	REMAININGG
CHKCL	CHECK COMPONENT LABEL	POEWR	POWER
SIMNAME	PROCESS SIMPLE NAME	ABSL	ABSOLUTE
FORCODE	PROCESS FOR STATEMNET	NOT	LOGIC NOT
IFCODE	PROCESS IF STATEMENT	EVAL	EVALUATE VARIABLE
CHKGL	CHECK GENERATE LABEL	STOP	STOP STACK MACHINE

INTPUSH, **IDPUSH** and **POP** are the three instructions that manipulate the operands stack. **STOP** halts the stack machine. The other instructions generate the IVHDL using the symbol table. The instructions on the left side of Table 4.1. process the declarations and statements, check the semantic of the VHDL design file. The instructions on the right side of Table 4.1. process the logic, relational and mathematical operations.

When the stack machine code generated by the parser is executed, the result is a group of records that contain the information from the VHDL design files. These records are denoted as Internal VHDL representations (IVHDL).

4.3.2. Internal VHDL Representation (IVHDL)

The Internal VHDL Representation (IVHDL) is a list of records called entity list. Some items in the records are sublists which are lists of some other records. The relation of the lists is shown in Fig. 4.6.

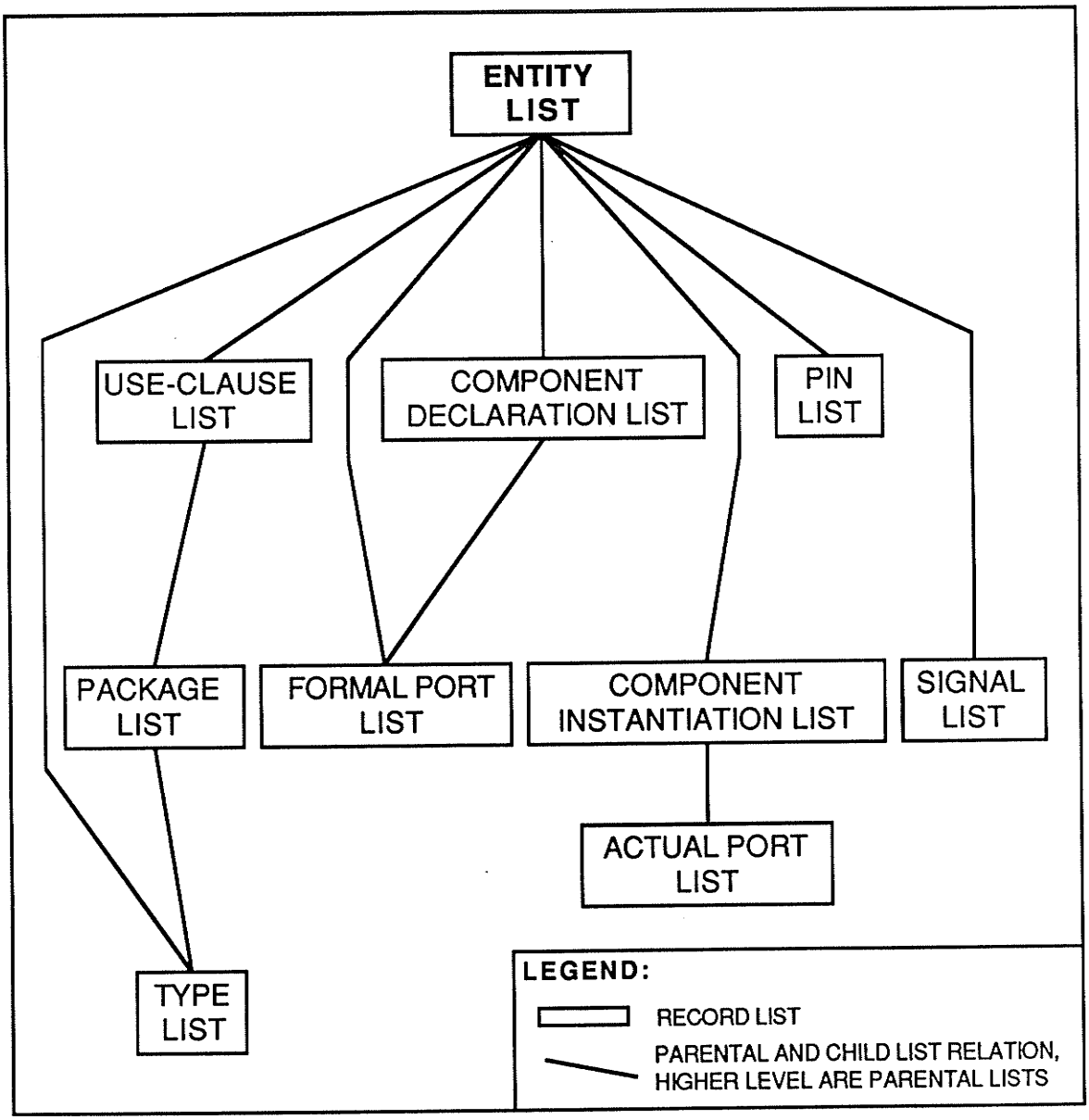

Fig. 4.6. Internal VHDL representation.

4.4. Xilinx Netlist Format Generation

After the IVHDL is obtained, it has to be converted into the internal Xilinx Netlist Format (IXNF) in order to generate the XNF files. Xilinx provides a program for the generation of the XNF file from IXNF.

The conversion starts from the entity list, since each design entity results in an XNF file. Then, the sublists in entity list are read, such as signal list, formal port list, pin list and component instantiation list. The actual port list is read when component instantiation is read. While the IVHDL is read, the corresponding IXNF is generated. The structure of the netlist generator is shown in Fig. 4.7.

4.4.1. Internal Xilinx Netlist Format

Xilinx provides the internal Xilinx Netlist Format (IXNF) and a program to generate the XNF files from the IXNF. The IXNF is also a group of record list that represents the XNF. WRITENET reads the lists and creates XNF files.

4.5. Flattening

The XNF files generated by WRITENET are still hierarchical files. Some XNF files are the components in the higher XNF files. The XNF files need to be flattened before the design can be implemented. Xilinx also provides the flattening program which reads all the XNF files in the design library and the used XNF files in standard libraries,

also checks the syntax of the XNF files and reports any XNF syntax errors.

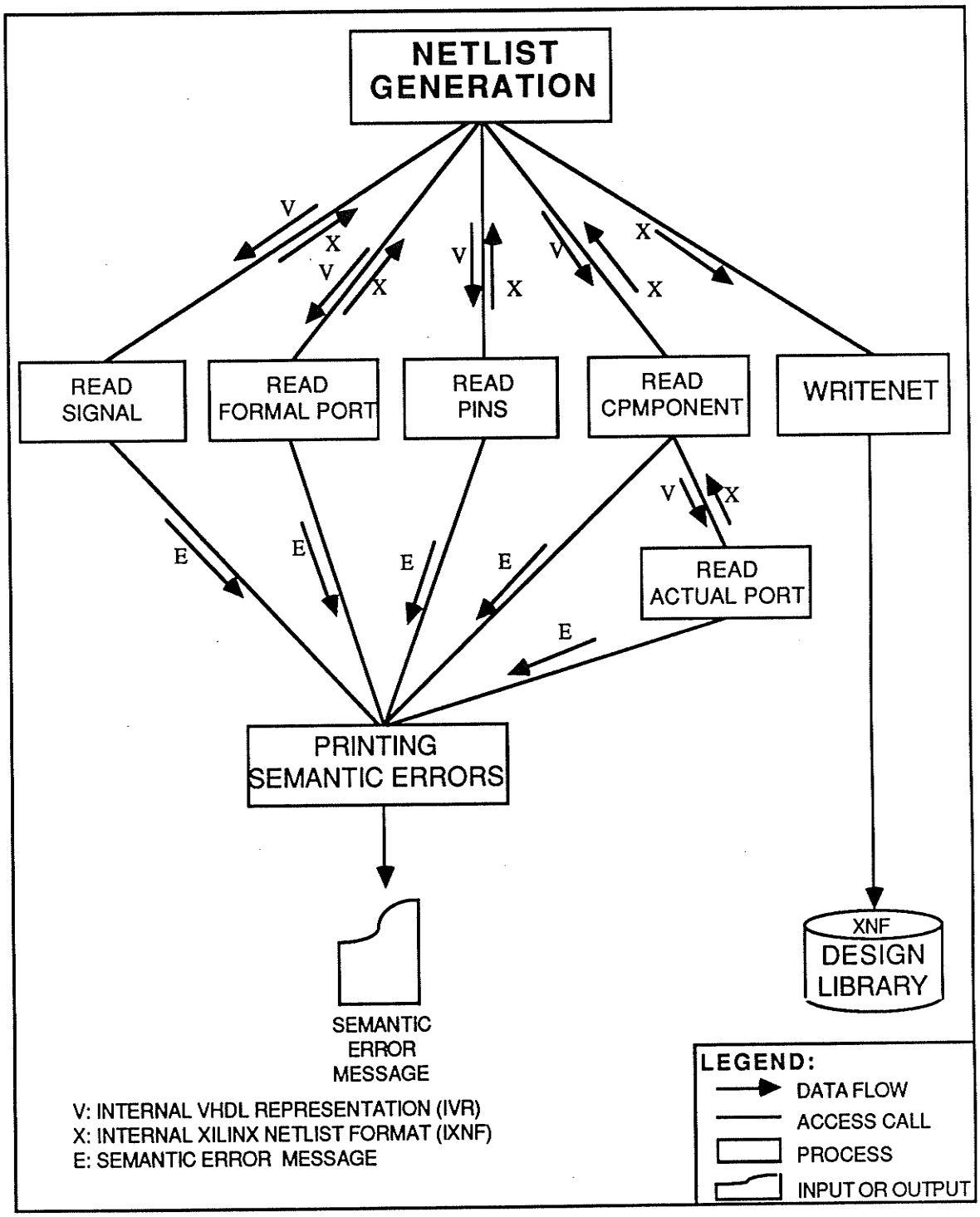

Fig. 4.7. Xilinx Netlist Format generation.

CHAPTER 5

IMPLEMENTATION, TESTING AND VERIFICATION

The VHDL compiler is implemented on a SUN workstation in the C language. The C program code has a total of 1371 lines. The LEX and YACC tools in the UNIX system are used to generate the lexical analyzer and the parser. The LEX specification has 75 lines, which results in 537 lines of C program code generated by LEX. The YACC specification has 285 lines, which results in 651 lines of C program code generated by YACC. The total 2559 lines of C code are compiled together with C program code provided by Xilinx for writing and flattening the XNF files. The sizes of the VHDL compiler for LCA are 98304 KBYTE for SUN3 and 106496 KBYTE for SUN4. The program lists can be found in Appendix E.

An IBM PC/AT compatible computer is used for the Xilinx LCA Development System. The PC and the SUN workstation communicate through Ethernet. The XNF files generated by the VHDL compiler can be transmitted from the SUN workstation to the PC for implementation.

The VHDL descriptions of circuit designs are compiled into XNF files. The circuit is implemented and verified using the XNF files. There are two criteria to verify the VHDL compiler: i) the XNF files generated by the VHDL compiler must follow the syntax

specification of the XNF; and ii), the LCA design in VHDL must function as required. During the implementation, the logic partitioning and reduction program reads the XNF files and checks syntax. Thus, the syntax of the XNF files is verified. The same designs are also entered using the schematic capture tool FutureNet, implemented and verified. The implementation and verification results of both designs in schematic and VHDL of the same circuit are compared in order to verify the function of the VHDL design. The designs can also be implemented in a demonstration board. Our testing shows the VHDL compiler works properly. Several examples are used to test and verify the compiler. The results of two examples, a 16-bit liner feedback shift register (LFSR) and a 4-bit ALU, are presented as follows.

5.1. Test 1: 16-bit Liner Feedback Shift Register

The first example is a 16-bit Liner Feedback Shift Register (LFSR). The polynomial of the LFSR is

$$X^{12} + X^{5} + 1.$$

A computer simulation of this LFSR is included in A.3. of Appendix A.

The LFSR is entered using the schematic capture tool FutureNet and implemented using the LCA automatic implementation tool. The schematic of the 16-bit LFSR is shown in Fig. 5.1. The symbols FDRD and FDS are standard cells form the standard library. The FDRD is a D flip-flop with reset. The FDS is a D flip-flop with set. The implementation is simulated using SILOS. The simulation result is included in A.4. of Appendix A. A graphic output of this simulation is also shown in Fig. 5.2. The

comparison of A.3. and A.4. of Appendix A shows the implementation works properly.

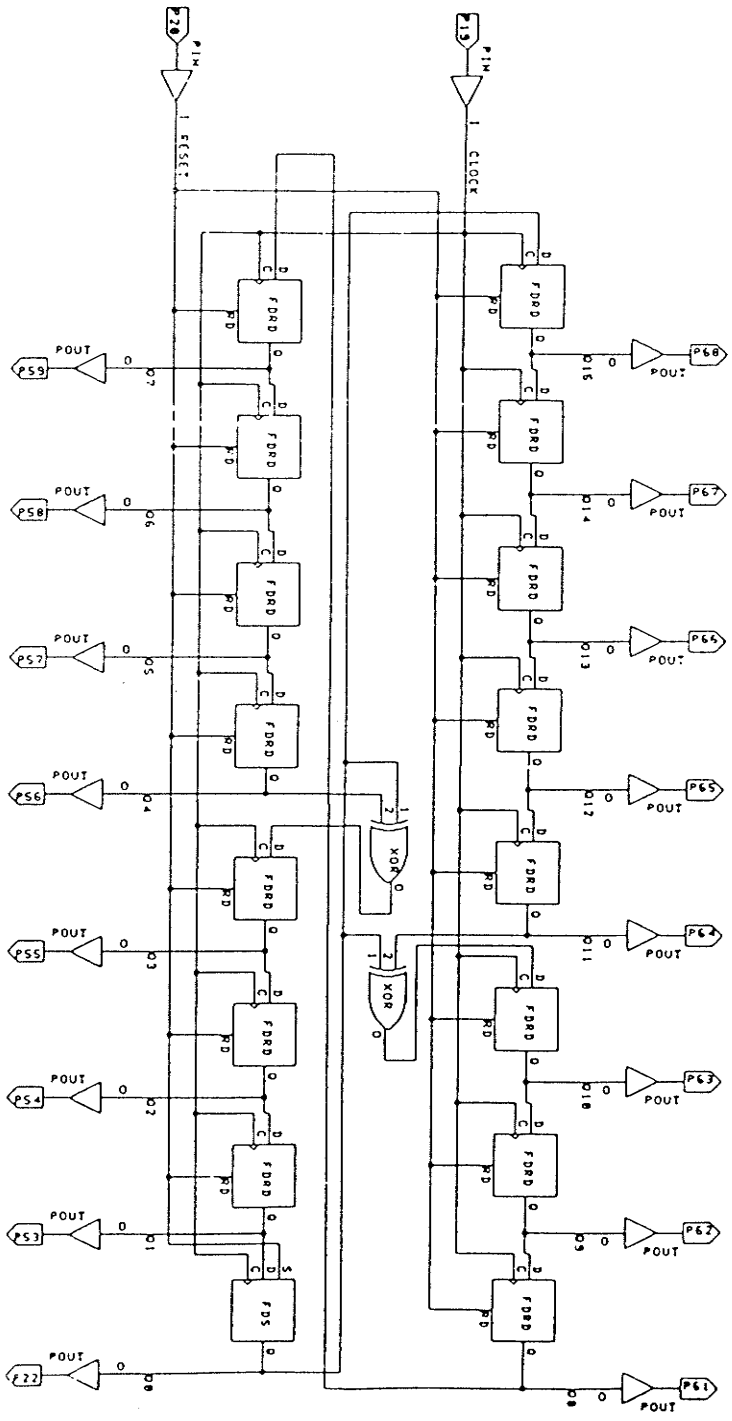

Fig. 5.1. The schematic of the 16-bit LFSR.

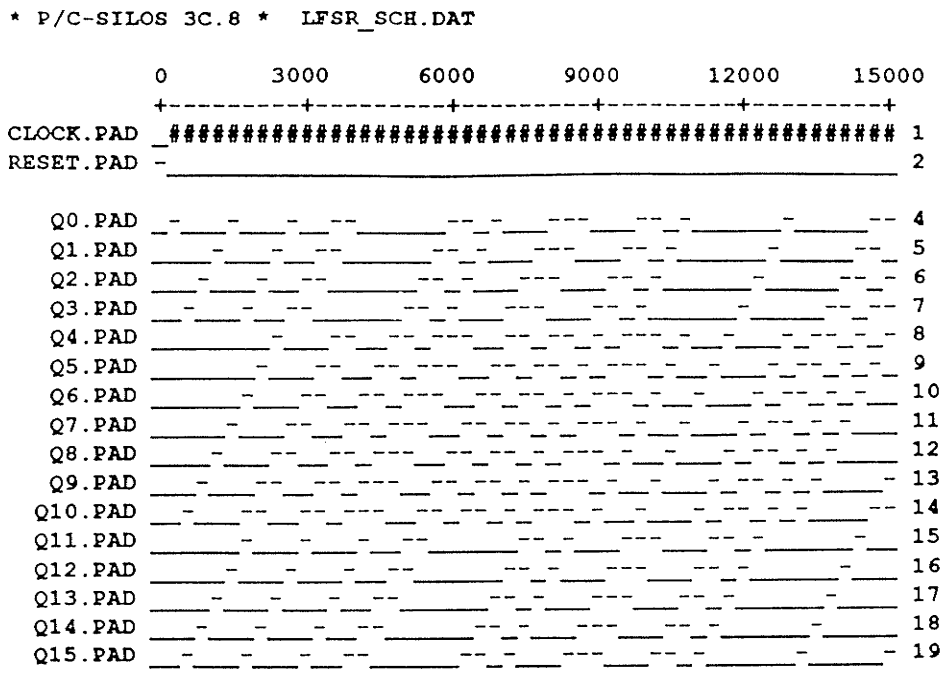

Fig. 5.2. The graphic output of the simulation of the 16-bit LFSR implemented from the schematic.

The LFSR displayed in Fig. 5.2. is described in VHDL description to test our VHDL compiler. The VHDL description of the LFSR is included in A.1. of Appendix A. This VHDL description is compiled by the VHDL compiler which generates the XNF file for implementation using the LCA automatic implementation tool. The LCA layout of the LFSR implemented from the VHDL description is shown in Fig. 5.3.

The LFSR implemented from the VHDL description was also simulated using SILOS. A graphic output of the simulation is shown in Fig. 5.4. The comparison of Fig. 5.2. and Fig. 5.4. shows the LFSR implemented from the VHDL description works properly.

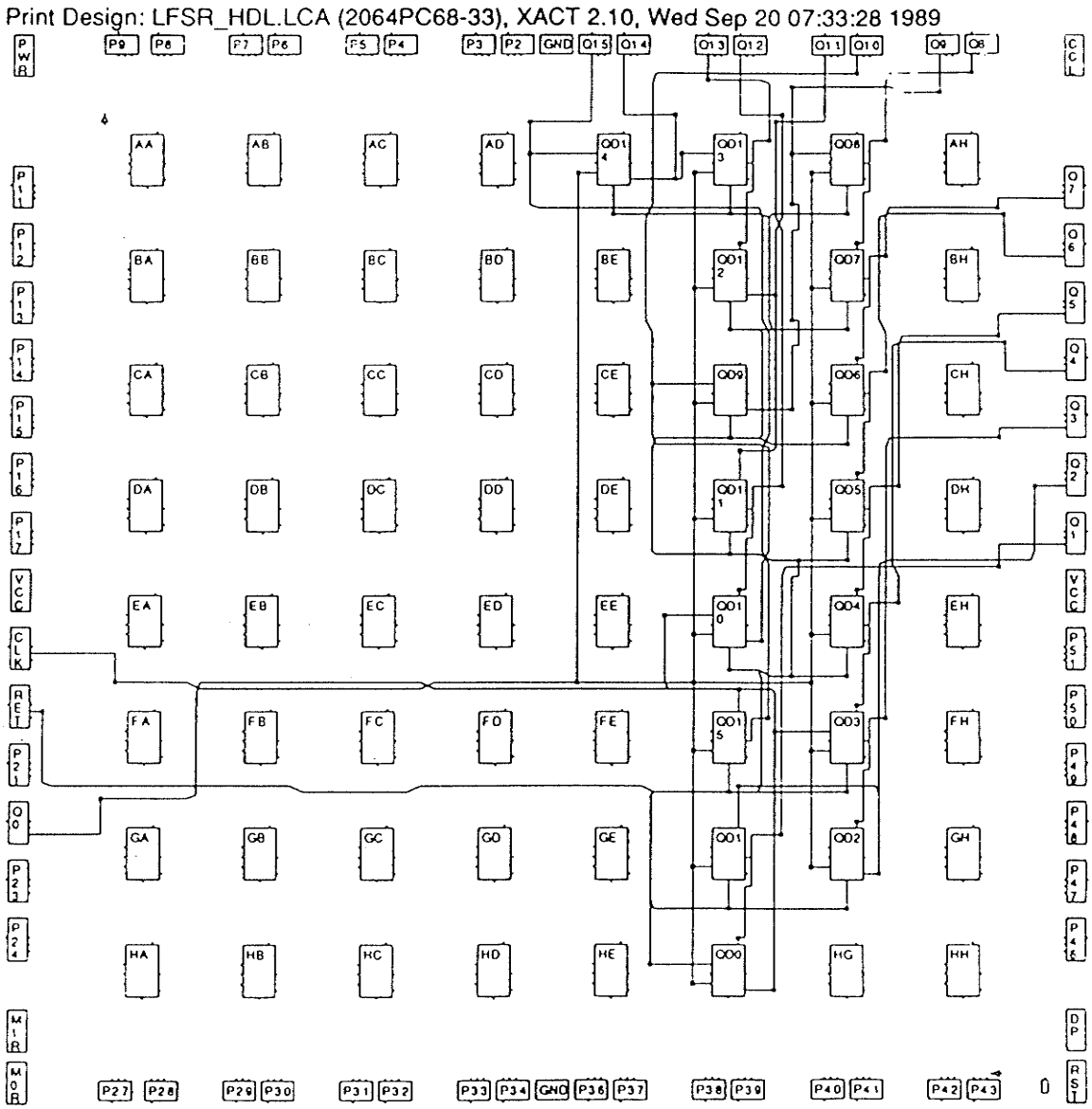

Fig. 5.3. The layout of the 16-bit LFSR implemented from the VHDL description.

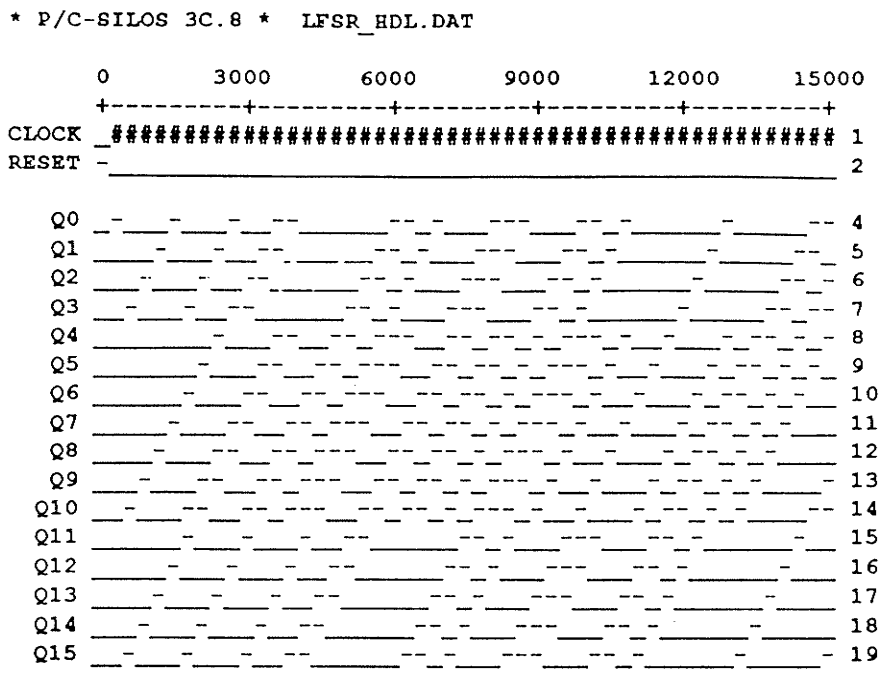

Fig. 5.4. The graphic output of the simulation of the 16-bit LFSR implemented from the VHDL description.

5.2. Test 2: 4-bit ALU

The second example is a 4-bit ALU with two control lines. The operation of the ALU is shown in Table 5.1. C1 and C0 are the two control lines, CIN is the carry in, I and Q are the two 4-bit inputs, COUT is the carry out, and F is the 4-bit output.

The ALU is entered using the schematic capture tool FutureNet and implemented using the LCA automatic implementation tool. The schematic of the 4-bit ALU is shown in Fig. 5.5. The symbols GADD and RD4 from the standard cell library are one-bit full

adder and 4-bit data register. The data register is used to store Q because of the limitation of the number of switches useable for inputs on the demonstration board used for testing the design.

Table 5.1. The operation of the 4-bit ALU.

CONTROL LINES		F
C1	C0	
0	0	F = Q + CIN
0	1	F= I + Q +CIN
1	0	F = ~I + Q + CIN
1	1	F = 1111 + Q + CIN

The implementation of the ALU was simulated using SILOS. A graphic output of the simulation is shown in Fig. 5.6. The implementation was also tested on a DEMO board, which shows the design works properly.

The VHDL description of the 4-bit ALU is written to test our VHDL compiler. The VHDL description of the ALU is included in A.2. of Appendix A. This VHDL description is compiled by the VHDL compiler which generates the XNF file for implementation using the LCA automatic implementation tool. The LCA layout of the ALU implemented from the VHDL description is shown in Fig. 5.7.

The ALU implemented from the VHDL description was simulated using SILOS and also tested on the demonstration board. A graphic output of the simulation is shown in Fig. 5.8. Both the testing on the demonstration board and the comparison of Fig. 5.6. and Fig. 5.8. show the ALU implemented from the VHDL description works properly.

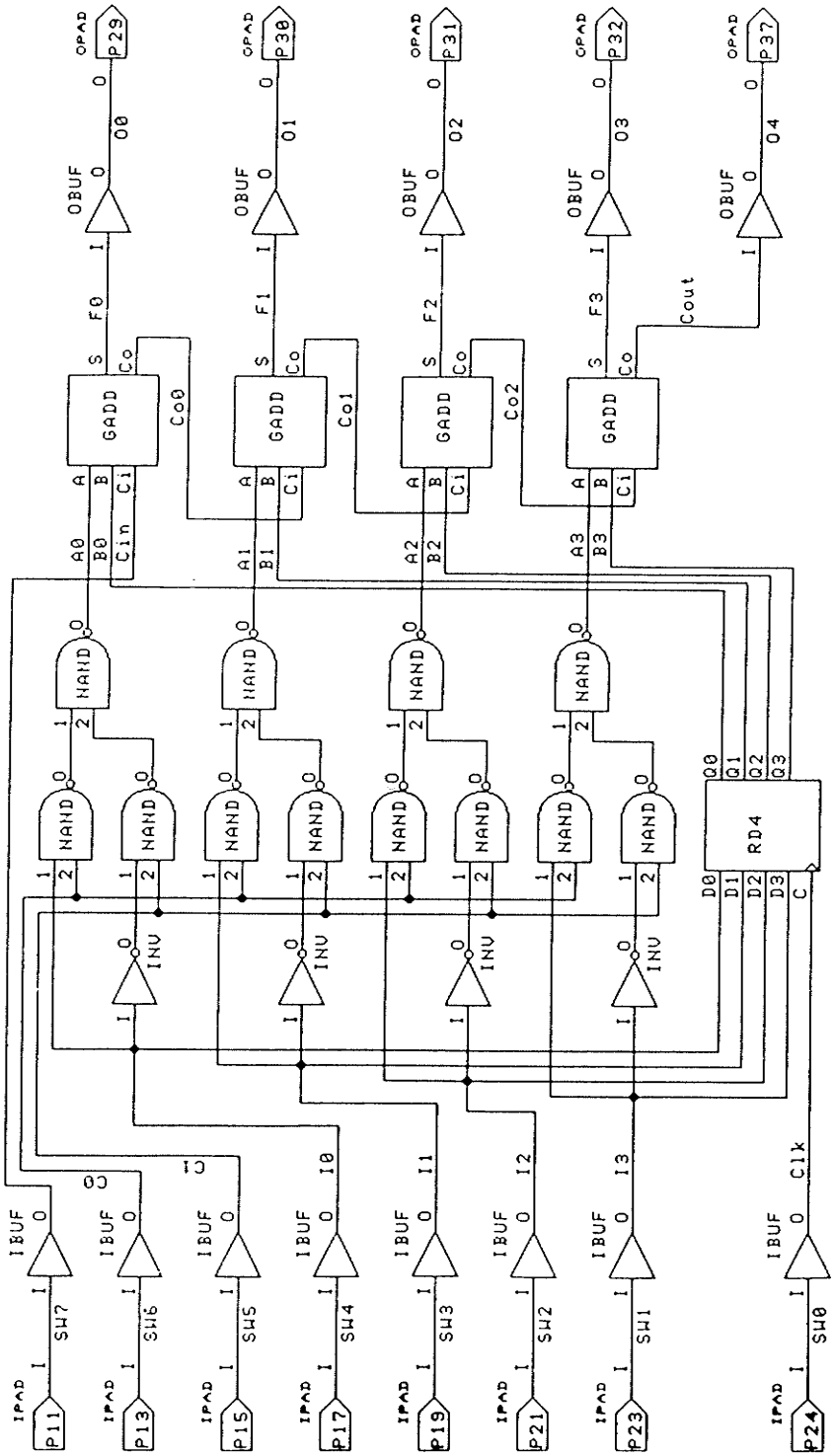

Fig. 5.5. The schematic of the 4-bit ALU.

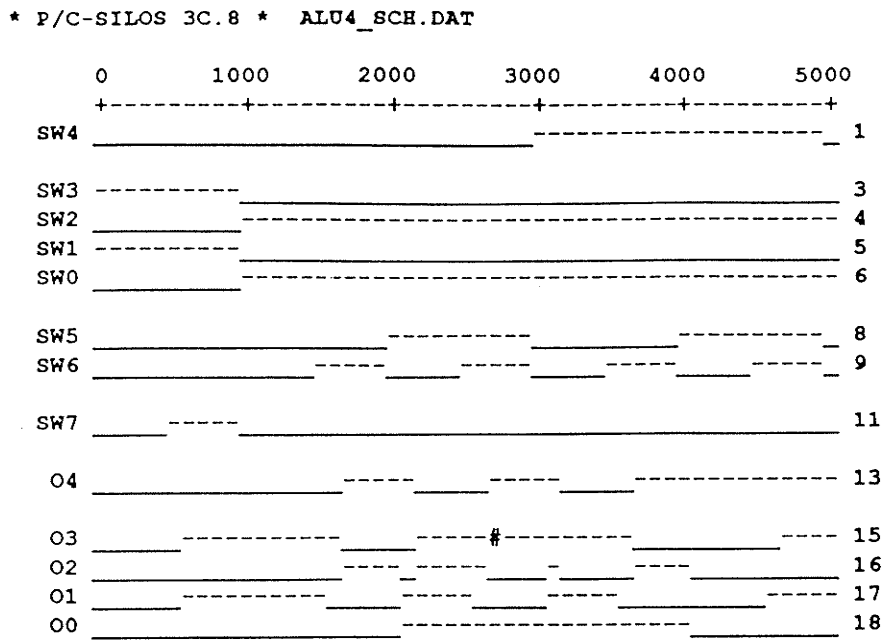

Fig. 5.6. The graphic output of the simulation of the 4-bit ALU implemented from the schematic.

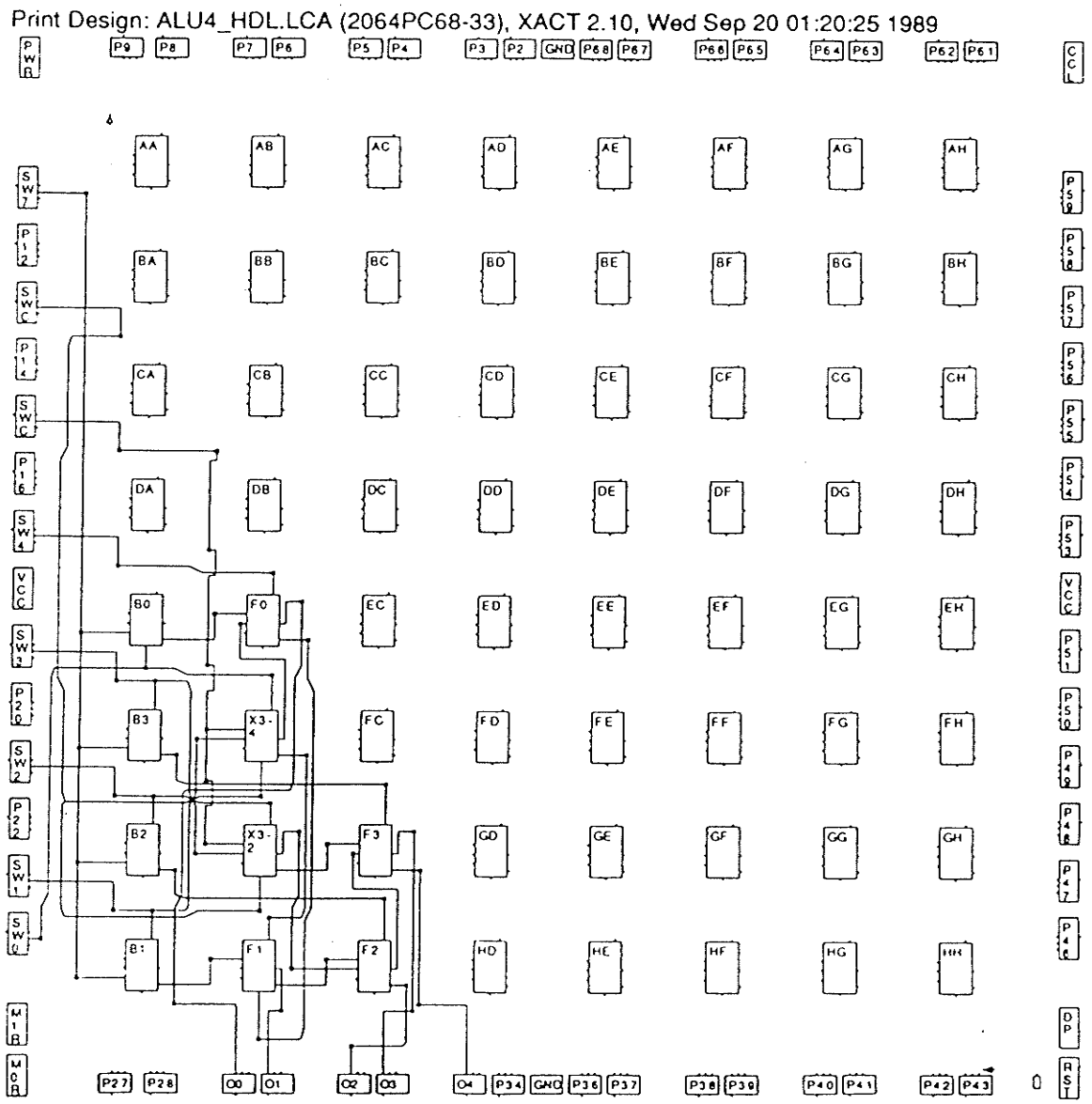

Fig. 5.7. The layout of the 4-bit ALU implemented from the VHDL description.

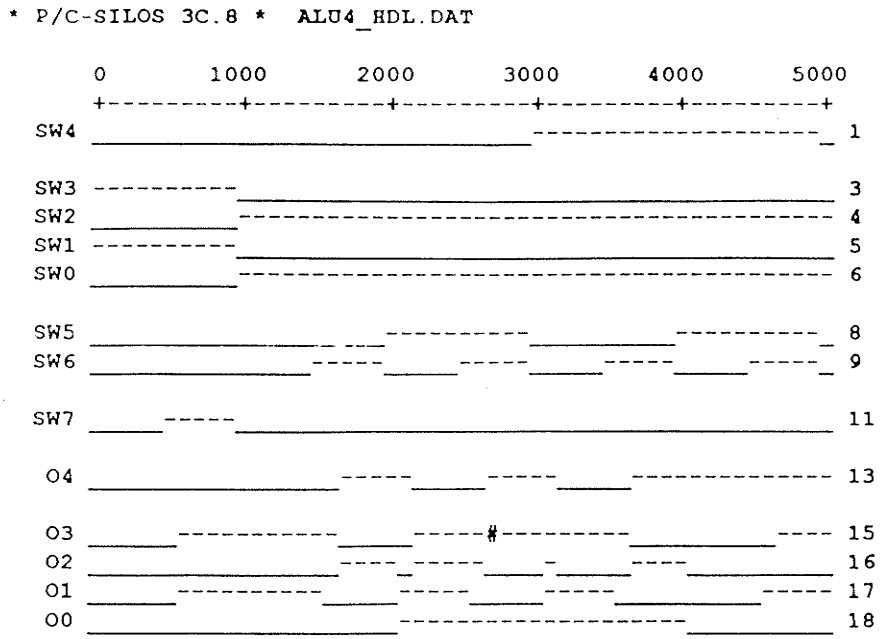

Fig. 5.8. The graphic output of the simulation of the 4-bit ALU implemented from the VHDL description.

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

An electrical circuit may include tens of hundreds of components. It is one of the most difficult problems to cope with the complexity in circuit design process. CAE tools have been developed to help the designers implement and verify their designs. A circuit description has to be entered into the computer in machine understandable forms in order to use CAE tools. Circuit description may be in textual format or graphical format, or both. Hardware Description Languages (HDLs), formalized modular languages to describe electronic circuits, are becoming increasingly popular because they are easy to create, modify and store.

The technology to implement a circuit design is also an important factor of design complexity. It is very difficult to obtain optimized circuit layout for a full custom VLSI, while acceptable layout may be obtained for semi-custom VLSI such as gate arrays and PLDs. The user programmable gate arrays LCA combines both the flexibility of gate arrays and the instant availability of PLDs. The LCAs are also reprogrammable.

The density of LCAs is from 1200 to 9000 equivalent gates. A dedicated CAE system is provided to support the LCA design. A circuit design can be entered using schematic tools or PALASM. However, the device like LCA should have the option of

using an HDL as design entry. In this thesis, a subset of the IEEE standard VHDL is chosen for LCA design. A VHDL compiler was developed to accept VHDL architectural descriptions and generate the XNF files for LCA implementation tools. The implementations from VHDL descriptions using the compiler are the same as those from schematics using the schematic capture tools supported by Xilinx.

This compiler enables the designer to use hardware description language as the design entry option. Compared with the traditional graphical design entry approach, hardware description languages are easy to create and modify, and serve as documentation of the design. This compiler can also provide an interface with other VLSI design systems so that the LCA can be used for prototyping of the VLSI design. In addition, we anticipate that graphical design system will be unable handle the increasingly complexity of sophisticated system design. For this reason, the VHDL tool developed here will directly contribute to our efforts in promoting system level design space exploration in our laboratory.

As demonstrated in the previous chapters, this research work has the following key contributions:

1. Studied the LCA technology and its design methodology. Its instant implementation is especially well suited for electronic breadboarding and educational purposes. An HDL is needed for LCA as a design entry option. The HDL for LCA can be compiled into XNF, which is the interface of the LCA implementation tools.
2. Studied HDLs, especially the VHDL. A VHDL architectural description subset is

chosen and adopted for LCA design.

3. Developed a VHDL compiler for LCA design. The compiler accepts the VHDL architectural description files and generates the XNF files. Tools such as LEX and YACC for compiler design and the stack machine structure are used in the compiler design for easy further development.
4. The compiler is tested and verified through examples. Two examples are demonstrated to show the compiler works properly.

Further development of the research work is recommended as follows:

1. The compiler can be expanded to accept VHDL dataflow and behavioural descriptions. The work should concentrate on the development of the synthesis form the dataflow and behaviour to architecture.
2. A platform can be developed for educational purpose. An AHPL compiler for LCA can be developed, since it is used widely in educational areas.
3. A platform can be built for VLSI design prototyping. The design entry method used for the VLSI design can be adopted for LCA. Thus, a VLSI design can be prototyped using LCAs easily before its fabrication.

REFERENCES

[AAIR88] R.D. Acosta, M. Alexandre, G. Imken, and B. Read, "The Role of VHDL in the MCC CAD System," *25th ACM/IEEE Design Automation Conference*, 1988, pp. 34-39..

[Acte88] *The ACT Family Products*. Sunnyvale (CA): Actel Inc. 1988

[AGGC88] K. El-Ayat, A.E. Gamal, R. Guo, J. Chang, E. Hamdy, J. McCollum, and A. Mohsen , "A COMS electrically configurable gate array," *IEEE Inter. Solid-State Circuits Conf.*, IEEE Cat. 88CH2562-7, 1988, pp. 76-77.

[Arms89] J.R. Armstrong, *Chip-level Modeling with VHDL*. Englewood Cliffs (NJ): Prentice Hall, 1989, 148pp.

[ASU186] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986, 796pp.

[AWSc86] J.H. Aylor, R. Waxman, and C. Scarratt, "VHDL—Feature Description and Analysis," *IEEE Design & Test*, April 1986, pp. 17-27.

[Bart88] D. Barton, "Behavioural Descriptions in VHDL," *VLSI System Design*, June 1988, pp. 28-33.

[BiCo81] J. Birkner and V. Coli, *PAL: Programmable Array Logic Handbook*. Sunnyvale (CA): Monolithic Memories, Inc. 1981

[Burs87] D. Bursky, "Enhanced programmable arrays challenge gate array density," *Electronic Design*, September 17, 1987, pp. 83-86.

[CDFH86] W.S. Carter, K. Duong, R.H. Freeman, H.-C. Hsieh, J.Y. Ja, J.E. Mahoney, L.T. Ngo, and S.L. Sze, "A user programmable reconfigurable logic array," *IEEE Custom Integrated Circuits Conf.*, IEEE Cat. CH2258P2/86, 1986, pp. 233-235.

[Coel88] D.R. Coelho, "VHDL: A Call for Standards," *25th ACM/IEEE Design Automation Conference*, 1988, pp. 40-47.

[Cole86] B.C. Cole, "Field-Programmable Logic: A new market force," *Electronics*, January 27, 1986, pp. 25-31.

[Coll86a] R. Collett, "ASICs: Take your pick," *Digital Design*, June 1986, pp. 29–36.

[Coll86b] R. Collett , "Programmable logic declares war on gate arrays," *Digital Design*, July 1986, pp. 32-39.

[Coll87] R. Collett, "Reports from the PLD front," *ESD: The Electronic System Design*, February 1987, pp. 46-54.

[DeGa86] A. Dewey and A. Gadient, "VHDL Motivation," *IEEE Design & Test*, April 1986, pp. 12-16.

[Donn85] J. Donnell, "Crosspoint switch: A PLD approach," *Digital Design*, July 1985, pp. 40-44.

[Free88] R. Freeman, "User-programmable gate arrays," *IEEE Spectrum*, Dec. 1988, pp. 32-35.

[Futu88] *DASH: Schematic Designer.* Chatsworth (CA): FutureNet. 1988

[GDPa86] D.D. Gajski, N.D. Dutt, and B.M. Pangrle, "Silicon Compilation (Tutorial)," *IEEE Custom Integrated Circuits Conf.,* CH2258-2/86, 1986, pp. 102-110.

[Gilm86] A. S. Gilman, "VHDL—The Designer Environment," *IEEE Design & Test*, April 1986, pp. 42-47.

[Goer89] R. Goering, "Can Xilinx challenge masked gate arrays?" *High Performance System,* Sept. 1989, pp. 13-14, 19

[Goet86] E. Goetting, "EEPROM-based ASIC propels programmable logic to new levels of complexity," *Electronic Design*, May 1, 1986, pp. 201-206.

[HDKN87] H.-C. Hsieh, K. Duong, J.Y. Ja, R. Kanazawa, L.T. Ngo, L.G. Tinkey, W.S. Carter, and R.H. Freeman, "A second generation user-programmable reconfigurable gate array," *IEEE Custom Integrated Circuits Conf.*, IEEE Cat. CH2430-7/87, 1986, pp. 515-521.

[HiPe87] F.J. Hill and G.R. Peterson, *Digital Systems: Hardware Organization and Design, 3rd ed.* Jhon Wiley & Sons, Inc., 1987, 601pp.

[Holl87] E.E. Hollis, *Design of VLSI Gate Array ICs.* Englewood Cliffs (NJ): Prentice-Hall, Inc., 1987, 507pp.

[KePi84] B.W. Kernighan and R. Pike, *The UNIX Programming Environment.* Englewood Cliffs (NJ): Prentice-Hall, 1984, 357pp.

[Kins86] W. Kinsner, *Computer-Aided Engineering of Electronic Circuits: An Introduction.* Technical Report MC86-2, Sept. 1986

[Kins89] W. Kinsner, *Background On PLDs.* Technical Report, University of Manitoba, 1989.

[Land85a] S. Landry, "Application-specific ICs relying on RAM implement almost any logic function," *Electronic Design*, October 31, 1985, pp. 123-131.

[Land85b] S. Landry, "Printer buffer proves RAM-based logic's strength and versatility," *Electronic Design*, November 14, 1985, pp. 139-144.

[Land87] S.L. Landry, " 'Designer' logic and symbols with Logic Cell Arrays, " *IEEE Micro*, February 1987, pp. 51-59.

[Laut86] D.P. Lautzenheiser, "Semicustom IC offer new possibilities for software protection," *EDN*, June 12, 1986, pp. 177-182.

[LCAC88] M. Loughzail, M. Cote, M. Aboulhamid, and E. Cerny, "Experience with the VHDL environment," *25th ACM/IEEE Design Automation Conf.*, CH2540-3/88, 1988, pp. 28-33.

[LEX86] *UNIX Programmer's Manual: LEX — A Lexical Analyzer Generator*. SUN Microsystems, 1986, pp. 119-140.

[LMSh86] R. Lipsett, E. Marschner, and M. Shahdad, "VHDL—The Language," *IEEE Design & Test*, April 1986, pp. 28-37.

[Marr86] K. Marrin, "PLDs slow advance of gate arrays in low-end designs," *Digital Design*, February 1, 1986, pp. 43-54.

[Mars88] E. Marschner, "VHDL Design Environment," *VLSI System Design*, September 1988, pp. 40-82.

[Ment84] *Behavioural Language*. Reference Manual, Mentor Graphics Corp., 1984.

[Meye87] E.L. Meyer, "Programmable logic overview," *VLSI Systems Design*, October 1987, pp. 62-70.

[Meye89] E. Meyer, "VHDL opens the road to top-down design," *Computer Design*, February 1, 1989, pp. 57-62.

[Micr89] "Programmable Gate Arrays: Devices and application," *Microprocessors and Microsystems*, vol. 13: Special Issue, June 1989.

[MoMe81] *The Programmable Array Logic*. New York: McGraw Hill, 1981 (2nd ed.)

[NaSa86] J.D. Nash and L.F. Saunders, "VHDL Critique," *IEEE Design & Test*,, April 1986 pp. 54-65.

[RISC89] *RISC: Recent Developments in Processor Design*. The 32th IEEE Videoconference Seminars via Satellite, Oct. 1989.

[Sang85] A.L. Sangiovanni-Vincentelli, "An overview of synthesis system," *IEEE custom Integrated Circuits Conf.*, CH2157-6/85, 1985, pp. 221-225.

[Saun87] L.F. Saunders, "The IBM VHDL Design System," *24th ACM/IEEE Design Automation Conference*, 1987, pp. 484-490.

[Shae87] D. Shaer, "Tools help you retain the advantages of using breadboards in gate-array design," *EDN*, March 18, 1987, pp. 81-88.

[Shah86] M. Shahdad, "An Overview of VHDL and Technology," *23rd Design Automation Conference*, 1986, pp. 320-326.

[SIMU88] *SILOS*. Menlo Park (CA): SIMUCAD Inc. 1988

[SLMS85] M. Shahdad, P. Lipsett, E. Marschner, K. Sheehan, H. Cohen, R. Waxman, and D. Ackle, "VHSIC Hardware Description Language," *Computer*, February 1985, pp. 94-103.

[SmBo87] D.E. Smith and T.B. Bowns, "Regain lost I/O ports with erasable PLDs," *Electronic Design*, March 19, 1987, pp. 151-156.

[Smit88] D. Smith, "User-programmable chips take on broader range of applications," *VLSI Systems Design*, July 1988, pp. 88-93.

[Tare87] R.S. Tare, *UNIX Utilities*. McGraw-Hill Book Company, 1987, 387pp.

[VHDL88] *IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987.* New York: IEEE Press, 1988.

[Waug88] T. Waugh, "Programmable array serves as a controller for dynamic RAMs," *EDN*, February 18, 1988, pp. 169-176.

[Wils89] R. Wilson, "Xilinx boosts array density," *Computer Design: News Edition,* August 1989, p. 1.

[Wynn86] P. Wynn, "In-Circuit Emulation for ASIC based designs," *VLSI Systems Design*, Oct. 1986, pp. 38-45.

[Xili86a] *The Programmable Gate Array Design Handbook.* San Jose (CA): Xilinx, Inc. 1986.

[Xili86b] *XACT: LCA Development System.* San Jose (CA): Xilinx, Inc. 1986.

[Xili88a] *Programmable Gate Arrays: Components and Development Systems.* San Jose (CA): Xilinx, Inc. 1988.

[Xili88b] *Programmable Gate Arrays: XC3000 Logic Cell Array Family.* San Jose (CA): Xilinx, Inc. 1988.

[Xili88c] *LCA Programmable Gate Array: User's Guide.* San Jose (CA): Xilinx, Inc. 1988.

[Xili88d] *The Programmable Gate Array Design Handbook, 2nd ed.* San Jose (CA): Xilinx, Inc. 1988.

[Xili88e] *LCA Xilinx Netlist Specification.* San Jose (CA): Xilinx, Inc. 1988.

[Xili88f] *LCA External Netlist Tool Kit.* San Jose (CA): Xilinx, Inc. 1988.

[Xili89] *The Programmable Gate Array Data Book.* San Jose (CA): Xilinx, Inc. 1989.

[YACC86] *UNIX Programmer's Manual: YACC — Yet Another Compiler Compiler.* SUN Microsystems, 1986, pp. 143-182.

APPENDIX A

EXAMPLES AND SIMULATION RESULTS

A.1. VHDL DESCRIPTION OF THE 16-BIT LFSR

```
--
-- This is the architectural description of a 16-bit LFSR.
-- It is used for testing the VHDL2XNF compiler, Version 2.
--
-- Bing Liu
-- Department of Electrical Engineering
-- University of Manitoba
--

package BUSES is
        type BUSS_16 is array (0 to 15) of BIT;
end;

use BUSES;
entity LFSR16 is
        port ( CLK :  in BIT;
               RET :  in BIT;
               Q :    out BUSS_16);
        pins ( CLK :  P19;
               RET :  P20;
               Q(0) : P22;
               Q(1) : P53;
               Q(2) : P54;
               Q(3) : P55;
               Q(4) : P56;
               Q(5) : P57;
               Q(6) : P58;
               Q(7) : P59;
               Q(8) : P61;
               Q(9) : P62;
               Q(10) : P63;
               Q(11) : P64;
               Q(12) : P65;
               Q(13) : P66;
               Q(14) : P67;
```

```
            Q(15) : P68);
end;

architecture LFSR_BODY of LFSR16 is
      component fdrd port (      D, C : in BIT;
                                 RD :   in BIT;
                                 Q :    out BIT);
      end component;     -- D flip-flop with reset from standard library
      component fds port (       D, C : in BIT;
                                 S :    in BIT;
                                 Q :    out BIT);
      end component;     -- D flip-flop with set from standard library
      component xor2 port (      I1, I2 : in BIT;
                                 O :    out BIT);
      end component;     -- 2 inputs XOR gate from standard library
      component ibuf port (      I :    in BIT;
                                 O :    out BIT);
      end component;     -- input buffer  from standard library
      component obuf port (      I :    in BIT;
                                 O :    out BIT);
      end component;     -- output buffer from standard library

      signal X1, X2, CLOCK, RESET : BIT;
      signal QD : BUSS_16;

begin
      for cnt in 0 to 15 generate
         obuf port map (QD(cnt), Q(cnt));
      end generate;

      ibuf port map (CLK, CLOCK);
      ibuf port map (RET, RESET);

      xor2 port map (QD(0), QD(4), X1);
      xor2 port map (QD(0), QD(11), X2);

      LFSR : for cnt in 0 to 15 generate
                  if cnt = 0 generate
                        fds port map (QD(cnt+1), CLOCK, RESET, QD(cnt));
                  end generate;
            if (cnt > 0) and (cnt < 3) generate
                  fdrd port map (QD(cnt+1), CLOCK, RESET, QD(cnt));
            end generate;
            if cnt = 3 generate
                  fdrd port map (X1, CLOCK, RESET, QD(cnt));
            end generate;
            if (cnt > 3) and (cnt < 10) generate
                  fdrd port map (QD(cnt+1), CLOCK, RESET, QD(cnt));
            end generate;
            if cnt = 10 generate
                  fdrd port map (X2, CLOCK, RESET, QD(cnt));
            end generate;
            if (cnt > 10) and (cnt < 15) generate
```

```
            fdrd port map (QD(cnt+1), CLOCK, RESET, QD(cnt));
        end  generate;
        if cnt = 15 generate
            fdrd port map (QD(0), CLOCK, RESET, QD(cnt));
        end  generate;
    end  generate LFSR;

end;
```

A.2. XNF FILE OF THE 16-BIT LFSR GENERATED FROM THE VHDL DESCRIPTION

```
LCANET, 1
PROG, FLATTEN, 1.00, Created from: lfsr16.hdl, Wed Sep 20 02:10:05 1989
PROG, vhdl2XNF,  2.00, Created from: lfsr16.hdl, Wed Sep 20 02:10:05 1989
PART, 2064pc68-33
SYM, Output-18, xor
PIN, 1, I, QD4
PIN, 2, I, QD0
PIN, O, O, X1
END
SYM, Output-17, xor
PIN, 1, I, QD11
PIN, 2, I, QD0
PIN, O, O, X2
END
SYM, DI-2-16, OR
PIN, 1, I, QD1
PIN, 2, I, RESET
PIN, O, O, DI-16
END
SYM, Q-1-16, DFF
PIN, C, I, CLOCK
PIN, D, I, DI-16
PIN, Q, O, QD0
END
SYM, Q-2-15, DFF
PIN, C, I, CLOCK
PIN, D, I, QD2
PIN, Q, O, QD1
PIN, RD, I, RESET
END
SYM, Q-2-14, DFF
PIN, C, I, CLOCK
PIN, D, I, QD3
PIN, Q, O, QD2
PIN, RD, I, RESET
END
```

```
SYM, Q-2-13, DFF
PIN, C, I, CLOCK
PIN, D, I, X1
PIN, Q, O, QD3
PIN, RD, I, RESET
END
SYM, Q-2-12, DFF
PIN, C, I, CLOCK
PIN, D, I, QD5
PIN, Q, O, QD4
PIN, RD, I, RESET
END
SYM, Q-2-11, DFF
PIN, C, I, CLOCK
PIN, D, I, QD6
PIN, Q, O, QD5
PIN, RD, I, RESET
END
SYM, Q-2-10, DFF
PIN, C, I, CLOCK
PIN, D, I, QD7
PIN, Q, O, QD6
PIN, RD, I, RESET
END
SYM, Q-2-9, DFF
PIN, C, I, CLOCK
PIN, D, I, QD8
PIN, Q, O, QD7
PIN, RD, I, RESET
END
SYM, Q-2-8, DFF
PIN, C, I, CLOCK
PIN, D, I, QD9
PIN, Q, O, QD8
PIN, RD, I, RESET
END
SYM, Q-2-7, DFF
PIN, C, I, CLOCK
PIN, D, I, QD10
PIN, Q, O, QD9
PIN, RD, I, RESET
END
SYM, Q-2-6, DFF
PIN, C, I, CLOCK
PIN, D, I, X2
PIN, Q, O, QD10
PIN, RD, I, RESET
END
SYM, Q-2-5, DFF
PIN, C, I, CLOCK
PIN, D, I, QD12
PIN, Q, O, QD11
```

```
PIN, RD, I, RESET
END
SYM, Q-2-4, DFF
PIN, C, I, CLOCK
PIN, D, I, QD13
PIN, Q, O, QD12
PIN, RD, I, RESET
END
SYM, Q-2-3, DFF
PIN, C, I, CLOCK
PIN, D, I, QD14
PIN, Q, O, QD13
PIN, RD, I, RESET
END
SYM, Q-2-2, DFF
PIN, C, I, CLOCK
PIN, D, I, QD15
PIN, Q, O, QD14
PIN, RD, I, RESET
END
SYM, Q-2-1, DFF
PIN, C, I, CLOCK
PIN, D, I, QD0
PIN, Q, O, QD15
PIN, RD, I, RESET
END
SYM, LFSR_BODY19, IBUF
PIN, O, O, RESET
PIN, I, I, RET
END
SYM, LFSR_BODY20, IBUF
PIN, O, O, CLOCK
PIN, I, I, CLK
END
SYM, LFSR_BODY21, OBUF
PIN, O, O, Q15
PIN, I, I, QD15
END
SYM, LFSR_BODY22, OBUF
PIN, O, O, Q14
PIN, I, I, QD14
END
SYM, LFSR_BODY23, OBUF
PIN, O, O, Q13
PIN, I, I, QD13
END
SYM, LFSR_BODY24, OBUF
PIN, O, O, Q12
PIN, I, I, QD12
END
SYM, LFSR_BODY25, OBUF
PIN, O, O, Q11
PIN, I, I, QD11
```

```
END
SYM, LFSR_BODY26, OBUF
PIN, O, O, Q10
PIN, I, I, QD10
END
SYM, LFSR_BODY27, OBUF
PIN, O, O, Q9
PIN, I, I, QD9
END
SYM, LFSR_BODY28, OBUF
PIN, O, O, Q8
PIN, I, I, QD8
END
SYM, LFSR_BODY29, OBUF
PIN, O, O, Q7
PIN, I, I, QD7
END
SYM, LFSR_BODY30, OBUF
PIN, O, O, Q6
PIN, I, I, QD6
END
SYM, LFSR_BODY31, OBUF
PIN, O, O, Q5
PIN, I, I, QD5
END
SYM, LFSR_BODY32, OBUF
PIN, O, O, Q4
PIN, I, I, QD4
END
SYM, LFSR_BODY33, OBUF
PIN, O, O, Q3
PIN, I, I, QD3
END
SYM, LFSR_BODY34, OBUF
PIN, O, O, Q2
PIN, I, I, QD2
END
SYM, LFSR_BODY35, OBUF
PIN, O, O, Q1
PIN, I, I, QD1
END
SYM, LFSR_BODY36, OBUF
PIN, O, O, Q0
PIN, I, I, QD0
END
EXT, RET, I,, LOC=P20, BLKNM=RET
EXT, Q9, O,, LOC=P62, BLKNM=Q9
EXT, Q8, O,, LOC=P61, BLKNM=Q8
EXT, Q7, O,, LOC=P59, BLKNM=Q7
EXT, Q6, O,, LOC=P58, BLKNM=Q6
EXT, Q5, O,, LOC=P57, BLKNM=Q5
EXT, Q4, O,, LOC=P56, BLKNM=Q4
```

```
EXT, Q3, O,, LOC=P55, BLKNM=Q3
EXT, Q2, O,, LOC=P54, BLKNM=Q2
EXT, Q15, O,, LOC=P68, BLKNM=Q15
EXT, Q14, O,, LOC=P67, BLKNM=Q14
EXT, Q13, O,, LOC=P66, BLKNM=Q13
EXT, Q12, O,, LOC=P65, BLKNM=Q12
EXT, Q11, O,, LOC=P64, BLKNM=Q11
EXT, Q10, O,, LOC=P63, BLKNM=Q10
EXT, Q1, O,, LOC=P53, BLKNM=Q1
EXT, Q0, O,, LOC=P22, BLKNM=Q0
EXT, CLK, I,, LOC=P19, BLKNM=CLK
EOF
```

A.3. COMPUTER SIMULATION OF THE 16_BIT LFSR

The following is the result of a computer simulation of the 16-bit LFSR

$$X^{12} + X^{5} + 1.$$

The list shows 200 clock cycles of running time.

```
1000000000000000
0001000000100001
0010000001000010
0100000010000100
1000000100001000
0001001000110001
0010010001100010
0100100011000100
1001000110001000
0011001100110001
0110011001100010
1100110011000100
1000100110101001
0000001101110011
0000011011100110
0000110111001100
0001101110011000
0011011100110000
0110111001100000
1101110011000000
1010100110100001
0100001101100011
1000011011000110
0001110110101101
0011101101011010
```

0111011010110100
1110110101101000
1100101011110001
1000010111000011
0001101110100111
0011011101001110
0110111010011100
1101110100111000
1010101001010001
0100010010000011
1000100100000110
0000001000101101
0000010001011010
0000100010110100
0001000101101000
0010001011010000
0100010110100000
1000101101000000
0000011010100001
0000110101000010
0001101010000100
0011010100001000
0110101000010000
1101010000100000
1011100001100001
0110000011100011
1100000111000110
1001001110101101
0011011101111011
0110111011110110
1101110111101100
1010101111111001
0100011111010011
1000111110100110
0000111101101101
0001111011011010
0011110110110100
0111101101101000
1111011011010000
1111110110000001
1110101100100011
1100011001100111
1001110011101111
0010100111111111
0101001111111110
1010011111111100
0101111111011001
1011111110110010
0110111101000101
1101111010001010
1010110100110101
0100101001001011
1001010010010110

0011100100001101
0111001000011010
1110010000110100
1101100001001001
1010000010110011
0101000101000111
1010001010001110
0101010100111101
1010101001111010
0100010011010101
1000100110101010
0000001101110101
0000011011101010
0000110111010100
0001101110101000
0011011101010000
0110111010100000
1101110101000000
1010101010100001
0100010101100011
1000101011000110
0000010110101101
0000101101011010
0001011010110100
0010110101101000
0101101011010000
1011010110100000
0111101101100001
1111011011000010
1111110110100101
1110101101101011
1100011011110111
1001110111001111
0010101110111111
0101011101111110
1010111011111100
0100110111011001
1001101110110010
0010011101000101
0100111010001010
1001110100010100
0010101000001001
0101010000010010
1010100000100100
0100000001101001
1000000011010010
0001000110000101
0010001100001010
0100011000010100
1000110000101000
0000100001110001
0001000011100010
0010000111000100

0100001110001000
1000011100010000
0001111000000001
0011110000000010
0111100000000100
1111000000001000
1111000000110001
1111000001000011
1111000010100111
1111000101101111
1111001011111111
1111010111011111
1111101110011111
1110011100011111
1101111000011111
1010110000011111
0100100000011111
1001000000111110
0011000001011101
0110000010111010
1100000101110100
1001001011001001
0011010110110011
0110101101100110
1101011011001100
1011110110111001
0110101101010011
1101011010100110
1011110101101101
0110101011111011
1101010111110110
1011101111001101
0110011110111011
1100111101110110
1000111011001101
0000110110111011
0001101101110110
0011011011101100
0110110111011000
1101101110110000
1010011101000001
0101111010100011
1011110101000110
0110101010101101
1101010101011010
1011101010010101
0110010100001011
1100101000010110
1000010000001101
0001100000111011
0011000001110110
0110000011101100

```
1100000111011000
1001001110010001
0011011100000011
0110111000000110
1101110000001100
1010100000111001
0100000001010011
1000000010100110
0001000101101101
0010001011011010
0100010110110100
1000101101101000
0000011011110001
0000110111100010
0001101111000100
0011011110001000
0110111100010000
```

A.4. SIMULATION OF THE IMPLEMENTATION OF THE LFSR

The following is the result of the simulation of the 16-bit LFSR implemented from the schematic. The simulator, SILOS uses the netlist translated from the XNF file of the LFSR. The XNF file is generated from the LCA design file which is the physical layout of the LFSR. The list also shows 200 clock cycles simulation. The clock speed is 300*ns*.

```
$  P/C-SILOS  3C.8 *   OUTPUT   09:37:41   Sept 20, 1989

$  LFSR_SCH.DAT

          QQQQQQQQQQQQQQQ
          0123456789111111
          ..........012345
          PPPPPPPPPP......
          AAAAAAAAAAPPPPPP
          DDDDDDDDDDAAAAAA
                    DDDDDD
TIME
  300     1000000000000000
  600     0001000000100001
```

900	0010000001000010
1200	0100000010000100
1500	1000000100001000
1800	0001001000110001
2100	0010010001100010
2400	0100100011000100
2700	1001000110001000
3000	0011001100110001
3300	0110011001100010
3600	1100110011000100
3900	1000100110101001
4200	0000001101110011
4500	0000011011100110
4800	0000110111001100
5100	0001101110011000
5400	0011011100110000
5700	0110111001100000
6000	1101110011000000
6300	1010100110100001
6600	0100001101100011
6900	1000011011000110
7200	0001110110101101
7500	0011101101011010
7800	0111011010110100
8100	1110110101101000
8400	1100101011110001
8700	1000010111000011
9000	0001101110100111
9300	0011011101001110
9600	0110111010011100
9900	1101110100111000
10200	1010101001010001
10500	0100010010000011
10800	1000100100000110
11100	0000001000101101
11400	0000010001011010
11700	0000100010110100
12000	0001000101101000
12300	0010001011010000
12600	0100010110100000
12900	1000101101000000
13200	0000011010100001
13500	0000110101000010
13800	0001101010000100
14100	0011010100001000
14400	0110101000010000
14700	1101010000100000
15000	1011100001100001
15300	0110000011100011
15600	1100000111000110
15900	1001001110101101
16200	0011011101111011
16500	0110111011110110

16800	1101110111101100
17100	1010101111111001
17400	0100011111010011
17700	1000111110100110
18000	0000111101101101
18300	0001111011011010
18600	0011110110110100
18900	0111101101101000
19200	1111011011010000
19500	1111110110000001
19800	1110101100100011
20100	1100011001100111
20400	1001110011101111
20700	0010100111111111
21000	0101001111111110
21300	1010011111111100
21600	0101111111011001
21900	1011111110110010
22200	0110111101000101
22500	1101111010001010
22800	1010110100110101
23100	0100101001001011
23400	1001010010010110
23700	0011100100001101
24000	0111001000011010
24300	1110010000110100
24600	1101100001001001
24900	1010000010110011
25200	0101000101000111
25500	1010001010001110
25800	0101010100111101
26100	1010101001111010
26400	0100010011010101
26700	1000100110101010
27000	0000001101110101
27300	0000011011101010
27600	0000110111010100
27900	0001101110101000
28200	0011011101010000
28500	0110111010100000
28800	1101110101000000
29100	1010101010100001
29400	0100010101100011
29700	1000101011000110
30000	0000010110101101
30300	0000101101011010
30600	0001011010110100
30900	0010110101101000
31200	0101101011010000
31500	1011010110100000
31800	0111101101100001
32100	1111011011000010

32400	1111110110100101
32700	1110101101101011
33000	1100011011110111
33300	1001110111001111
33600	0010101110111111
33900	0101011101111110
34200	1010111011111100
34500	0100110111011001
34800	1001101110110010
35100	0010011101000101
35400	0100111010001010
35700	1001110100010100
36000	0010101000001001
36300	0101010000010010
36600	1010100000100100
36900	0100000001101001
37200	1000000011010010
37500	0001000110000101
37800	0010001100001010
38100	0100011000010100
38400	1000110000101000
38700	0000100001110001
39000	0001000011100010
39300	0010000111000100
39600	0100001110001000
39900	1000011100010000
40200	0001111000000001
40500	0011110000000010
40800	0111100000000100
41100	1111000000001000
41400	1111000000110001
41700	1111000001000011
42000	1111000010100111
42300	1111000101101111
42600	1111001011111111
42900	1111010111011111
43200	1111101110011111
43500	1110011100011111
43800	1101111000011111
44100	1010110000011111
44400	0100100000011111
44700	1001000000111110
45000	0011000001011101
45300	0110000010111010
45600	1100000101110100
45900	1001001011001001
46200	0011010110110011
46500	0110101101100110
46800	1101011011001100
47100	1011110110111001
47400	0110101101010011
47700	1101011010100110
48000	1011110101101101

```
48300   0110101011111011
48600   1101010111110110
48900   1011101111001101
49200   0110011110111011
49500   1100111101110110
49800   1000111011001101
50100   0000110110111011
50400   0001101101110110
50700   0011011011101100
51000   0110110111011000
51300   1101101110110000
51600   1010011101000001
51900   0101111010100011
52200   1011110101000110
52500   0110101010101101
52800   1101010101011010
53100   1011101010010101
53400   0110010100001011
53700   1100101000010110
54000   1000010000001101
54300   0001100000111011
54600   0011000001110110
54900   0110000011101100
55200   1100000111011000
55500   1001001110010001
55800   0011011100000011
56100   0110111000000110
56400   1101110000001100
56700   1010100000111001
57000   0100000001010011
57300   1000000010100110
57600   0001000101101101
57900   0010001011011010
58200   0100010110110100
58500   1000101101101000
58800   0000011011110001
59100   0000110111100010
59400   0001101111000100
59700   0011011110001000
60000   0110111100010000
```

A.5. VHDL DESCRIPTION OF THE 4-BIT ALU

```
--
-- This is the architectural description of a 4-bit ALU
-- It is used for testing the VHDL2XNF compiler, Version 2.0.
--
-- Bing Liu
-- Department of Electrical Engineering
```

```
-- University of Manitoba
-

package BUSES is
        type CONTROL_LINES is array (0 to 1) of BIT;
        type BUSS_4 is array (0 to 3) of BIT;
end;

use BUSES;
entity ALU1 is
        port (A, B :            in BIT;
              Cin :             in BIT;
              Control :         in CONTROL_LINES;
              Output :          out BIT;
              Cout :            out BIT);
end;

architecture ALU1_BODY of ALU1 is
        component inv port (I : in BIT; O : out BIT);
        end component;      -- invertor from standard library
        component nand2 port (I1, I2 : in BIT; O : out BIT);
        end component;      -- 2 inputs NAND gate from standard library
        component gadd port (A, B, Ci : in BIT; S, Co : out BIT);
        end component;      -- 1-bit full adder from standard library

        signal A_not, X1, X2, X3 : BIT;

begin
        invertor :      inv port map (A, A_not);
        nand_g1 :       nand2 port map (A, Control(0), X1);
        nand_g2:        nand2 port map (A_not, Control(1), X2);
        nand_g3 :       nand2 port map (X1, X2, X3);
        full_adder:     gadd port map (X3, B, Cin, Output, Cout);
end;

use BUSES;
entity ALU4 is
        port ( SW5 :            in BIT;             -- control line 0
               SW6 :            in BIT;             -- control line 1
               SW4 :            in BIT;             -- carry in
               SW :             in BUSS_4;          -- input 0 to 3
               SW7 :            in BIT;             -- clock
               O :              out BUSS_4;         -- output 0 to 3
               O4 :             out BIT);           -- carry out
        pins ( SW7 :            P11;
               SW6 :            P13;
               SW5 :            P15;
               SW4 :            P17;
               SW(0) :          P24;
               SW(1) :          P23;
               SW(2) :          P21;
               SW(3) :          P19;
               O(0) :           P29;
```

```
            O(1) :          P30;
            O(2) :          P31;
            O(3) :          P32;
            O4 :            P33);
end;

architecture ALU_BODY of ALU4 is

    type INTERNAL_CARRY is array (0 to 2) of BIT;

    component ALU1
        port ( A, B :       in BIT;
               Cin :        in BIT;
               Control :    in CONTROL_LINES;
               Output :     out BIT;
               Cout :       out BIT);
    end component;      -- 1-bit slice ALU entity previously defined
    component rd4
        port ( D :          in BUSS_4;
               C :          in BIT;
               Q :          out BUSS_4);
    end component;      -- 4-bit data register from standard library
    component ibuf port ( I :  in BIT; O : out BIT);
    end component;      -- input buffer from standard library
    component obuf port ( I : in BIT; O : out BIT);
    end component;      -- output buffer from standard library

    signal      Carry : INTERNAL_CARRY;
    signal      I, F, B : BUSS_4;
    signal      C : CONTROL_LINES;
    signal      Cin, Cout, Clk : BIT;

begin
    for cnt in 0 to 3 generate
        ibuf port map (SW(cnt), I(cnt));
        obuf port map (F(cnt), O(cnt));
    end  generate;

    ibuf port map (SW5, C(0));
    ibuf port map (SW6, C(1));

    ibuf port map (SW4, Cin);
    ibuf port map (SW7, Clk);
    obuf port map (Cout, O4);

    Register : rd4 port map (I, Clk, B);

    ALUS : for cnt in 0 to 3 generate
       if cnt = 0 generate
          ALU1 port map (I(cnt), B(cnt), Cin, C, F(cnt), Carry(cnt));
       end  generate;
       if cnt = 3 generate
```

```
            ALU1 port map (I(cnt), B(cnt), Carry(cnt-1), C, F(cnt), Cout);
        end  generate;
        if (cnt > 0) and (cnt < 3) generate
            ALU1 port map (I(cnt), B(cnt), Carry(cnt-1), C, F(cnt), Carry(cnt));
        end  generate;
    end  generate ALUS;

end;
```

A.6. XNF FILE OF THE 4-BIT ALU GENERATED FROM THE VHDL DESCRIPTION

```
LCANET, 1
PROG, FLATTEN, 1.00, Created from: ALU4.HDL, Tue Sep  19 21:29:38 1989
PROG, vhdl2xnf, 2.00, Created from: ALU4.HDL, Tue Sep  19  21:29:37 1989
PART, 2064pc68-33
SYM, Output-21, NAND
PIN, 1, I, C0
PIN, 2, I, I3
PIN, O, O, X1-1
END
SYM, Output-20, NAND
PIN, 1, I, C1
PIN, 2, I, A_NOT-1
PIN, O, O, X2-1
END
SYM, Output-19, NAND
PIN, 1, I, X2-1
PIN, 2, I, X1-1
PIN, O, O, X3-1
END
SYM, S-5-18, XOR
PIN, 1, I, X3-1
PIN, 2, I, B3
PIN, 3, I, CARRY2
PIN, O, O, F3
END
SYM, AC-4-18, AND
PIN, 1, I, X3-1
PIN, 2, I, CARRY2
PIN, O, O, AC-18
END
SYM, BC-3-18, AND
PIN, 1, I, B3
PIN, 2, I, CARRY2
PIN, O, O, BC-18
END
SYM, Co-2-18, OR
```

```
PIN, 1, I, AB-18
PIN, 2, I, BC-18
PIN, 3, I, AC-18
PIN, O, O, COUT
END
SYM, AB-1-18, AND
PIN, 1, I, B3
PIN, 2, I, X3-1
PIN, O, O, AB-18
END
SYM, Output-17, NAND
PIN, 1, I, C0
PIN, 2, I, I2
PIN, O, O, X1-2
END
SYM, Output-16, NAND
PIN, 1, I, C1
PIN, 2, I, A_NOT-2
PIN, O, O, X2-2
END
SYM, Output-15, NAND
PIN, 1, I, X2-2
PIN, 2, I, X1-2
PIN, O, O, X3-2
END
SYM, S-5-14, XOR
PIN, 1, I, X3-2
PIN, 2, I, B2
PIN, 3, I, CARRY1
PIN, O, O, F2
END
SYM, AC-4-14, AND
PIN, 1, I, X3-2
PIN, 2, I, CARRY1
PIN, O, O, AC-14
END
SYM, BC-3-14, AND
PIN, 1, I, B2
PIN, 2, I, CARRY1
PIN, O, O, BC-14
END
SYM, Co-2-14, OR
PIN, 1, I, AB-14
PIN, 2, I, BC-14
PIN, 3, I, AC-14
PIN, O, O, CARRY2
END
SYM, AB-1-14, AND
PIN, 1, I, B2
PIN, 2, I, X3-2
PIN, O, O, AB-14
END
SYM, Output-13, NAND
```

```
PIN, 1, I, C0
PIN, 2, I, I1
PIN, O, O, X1-3
END
SYM, Output-12, NAND
PIN, 1, I, C1
PIN, 2, I, A_NOT-3
PIN, O, O, X2-3
END
SYM, Output-11, NAND
PIN, 1, I, X2-3
PIN, 2, I, X1-3
PIN, O, O, X3-3
END
SYM, S-5-10, XOR
PIN, 1, I, X3-3
PIN, 2, I, B1
PIN, 3, I, CARRY0
PIN, O, O, F1
END
SYM, AC-4-10, AND
PIN, 1, I, X3-3
PIN, 2, I, CARRY0
PIN, O, O, AC-10
END
SYM, BC-3-10, AND
PIN, 1, I, B1
PIN, 2, I, CARRY0
PIN, O, O, BC-10
END
SYM, Co-2-10, OR
PIN, 1, I, AB-10
PIN, 2, I, BC-10
PIN, 3, I, AC-10
PIN, O, O, CARRY1
END
SYM, AB-1-10, AND
PIN, 1, I, B1
PIN, 2, I, X3-3
PIN, O, O, AB-10
END
SYM, Output-9, NAND
PIN, 1, I, C0
PIN, 2, I, I0
PIN, O, O, X1-4
END
SYM, Output-8, NAND
PIN, 1, I, C1
PIN, 2, I, A_NOT-4
PIN, O, O, X2-4
END
SYM, Output-7, NAND
```

```
PIN, 1, I, X2-4
PIN, 2, I, X1-4
PIN, O, O, X3-4
END
SYM, S-5-6, XOR
PIN, 1, I, X3-4
PIN, 2, I, B0
PIN, 3, I, CIN
PIN, O, O, F0
END
SYM, AC-4-6, AND
PIN, 1, I, X3-4
PIN, 2, I, CIN
PIN, O, O, AC-6
END
SYM, BC-3-6, AND
PIN, 1, I, B0
PIN, 2, I, CIN
PIN, O, O, BC-6
END
SYM, Co-2-6, OR
PIN, 1, I, AB-6
PIN, 2, I, BC-6
PIN, 3, I, AC-6
PIN, O, O, CARRY0
END
SYM, AB-1-6, AND
PIN, 1, I, B0
PIN, 2, I, X3-4
PIN, O, O, AB-6
END
SYM, Q3-4-5, DFF
PIN, C, I, CLK
PIN, D, I, I3
PIN, Q, O, B3
END
SYM, Q2-3-5, DFF
PIN, C, I, CLK
PIN, D, I, I2
PIN, Q, O, B2
END
SYM, Q1-2-5, DFF
PIN, C, I, CLK
PIN, D, I, I1
PIN, Q, O, B1
END
SYM, Q0-1-5, DFF
PIN, C, I, CLK
PIN, D, I, I0
PIN, Q, O, B0
END
SYM, INVERTOR-4, INV
PIN, O, O, A_NOT-4
```

```
PIN, I, I, I0
END
SYM, INVERTOR-3, INV
PIN, O, O, A_NOT-3
PIN, I, I, I1
END
SYM, INVERTOR-2, INV
PIN, O, O, A_NOT-2
PIN, I, I, I2
END
SYM, INVERTOR-1, INV
PIN, O, O, A_NOT-1
PIN, I, I, I3
END
SYM, ALU_BODY6, OBUF
PIN, O, O, O4
PIN, I, I, COUT
END
SYM, ALU_BODY7, IBUF
PIN, O, O, CLK
PIN, I, I, SW7
END
SYM, ALU_BODY8, IBUF
PIN, O, O, CIN
PIN, I, I, SW4
END
SYM, ALU_BODY9, IBUF
PIN, O, O, C1
PIN, I, I, SW6
END
SYM, ALU_BODY10, IBUF
PIN, O, O, C0
PIN, I, I, SW5
END
SYM, ALU_BODY11, OBUF
PIN, O, O, O3
PIN, I, I, F3
END
SYM, ALU_BODY12, IBUF
PIN, O, O, I3
PIN, I, I, SW3
END
SYM, ALU_BODY13, OBUF
PIN, O, O, O2
PIN, I, I, F2
END
SYM, ALU_BODY14, IBUF
PIN, O, O, I2
PIN, I, I, SW2
END
SYM, ALU_BODY15, OBUF
PIN, O, O, O1
```

```
PIN, I, I, F1
END
SYM, ALU_BODY16, IBUF
PIN, O, O, I1
PIN, I, I, SW1
END
SYM, ALU_BODY17, OBUF
PIN, O, O, O0
PIN, I, I, F0
END
SYM, ALU_BODY18, IBUF
PIN, O, O, I0
PIN, I, I, SW0
END
EXT, SW7, I,, LOC=P11, BLKNM=SW7
EXT, SW6, I,, LOC=P13, BLKNM=SW6
EXT, SW5, I,, LOC=P15, BLKNM=SW5
EXT, SW4, I,, LOC=P17, BLKNM=SW4
EXT, SW3, I,, LOC=P19, BLKNM=SW3
EXT, SW2, I,, LOC=P21, BLKNM=SW2
EXT, SW1, I,, LOC=P23, BLKNM=SW1
EXT, SW0, I,, LOC=P24, BLKNM=SW0
EXT, O4, O,, LOC=P33, BLKNM=O4
EXT, O3, O,, LOC=P32, BLKNM=O3
EXT, O2, O,, LOC=P31, BLKNM=O2
EXT, O1, O,, LOC=P30, BLKNM=O1
EXT, O0, O,, LOC=P29, BLKNM=O0
EOF
```

APPENDIX B

USER'S GUIDE

Very High Speed Integrated Circuit Hardware Description Language (VHDL) is a language being developed to describe simple and complex hardware systems. It provides the capability to hierarchically describe systems by their architectures, by their dataflow, by their behaviours, or a combination of the three.

A VHDL architectural description is used to describe Xilinx Logic Cell Array (LCA) designs. The designs are mainly based on standard or custom cells and connections of these cells. The VHDL compiler can generate Xilinx Netlist Format (XNF) files for LCA implementation tools which generate the configuration program to configure an LCA device. This section defines the VHDL subset and provides the procedure to implementation a circuit design in LCA using the VHDL compiler.

B.1. SUPPORTED LANGUAGE SYNTAX

In VHDL, the primary element is called a design entity, which describes a system, subsystem, or cell. A design entity consists of two parts, an interface and an architecture body. The interface defines the input and output ports of the entity. The architecture body defines the function of the entity in one of three forms: architectural, dataflow, and

behavioral description, or combination of the three.

An architectural description consists solely of instances, which define the occurrence of some component which comprises a part of the entity. Components can be standard cells or entities which are defined elsewhere. The connection of instances is accomplished by referencing the ports on the instances to ports of the entity as well as to internal signals. signals are basically local points within an entity used to define electrically connected nodes.

Signals and ports are of a specific type. The simplest and most common predefined type is BIT. A BIT is a single line which can take on a single value and connect single ports together. Four other types BIT_X, BIT_C, BIT_N, and BIT_L are predefined to support LCA design. The X parameter on a signal tells the partitioning algorithm to make this signal an explicit LCA net. Any signal which does not have the X parameter specified for it may be pulled into a CLB logic function during partitioning. The X parameter is useful for controlling the partitioning algorithm in certain cases. The C parameter indicates that the signal is on a critical path and that all efforts should be made to minimize the delay through this signal. This affects the partitioning algorithm and is also passed along to the automatic placement and routing (APR) program. The N parameter indicates that the timing of the signal is non-critical and all other signals should take precedence over it. This affects the partitioning algorithm and is also passed along to the APR program. The L parameter is passed along to APR and tells APR that this signal should be routed using a long-line on the LCA. You can define new types such as an array of BITS of some arbitrary size for easily describing buses.

Ports also have a mode associated with them such as IN, OUT, DOT NOT (open collector). INOUT (bidirectional), and LINKAGE (unknown). The mode of a port can be used to check design errors such as two OUT ports connected together, nothing driving an IN port. A pin declaration associated with port is added into the VHDL to define the pin number of a I/O port.

The following describes the syntax of the portion of the VHDL language that is supported by the compiler. While it is not a majority portion, it still provides a sufficient method of describing LCA designs, and also a base for further development of the compiler. Note that a VHDL description tends to be self-documenting.

The form of the VHDL statement definitions is given using a simple variant of Backus-Naur Form:

1. Reserved keywords are given in boldface.
2. A vertical bar separates alternative items.
3. Square brackets enclose optional items.
4. Braces (i.e. {}) enclose a repeating item (zero or more repetition).

B.1.1. General Syntax

The computer parses the VHDL file into tokens. The end of line is simply considered to be a delimiter between tokens. Thus statements can span as arbitrary number of lines to improve readability. The maximum length of an input line is 80 characters.

B.1.1.1. Comments

Comments can appear anywhere in the file. The start of a comment is indicated by adjacent minus signs:

start_of_comment ::= --

The comment ends at the end of the line. Thus any text appearing after the start of a comment is ignored.

Example: signal X1, X2, X3 : BIT; --The part after the '--' is a comment.

B.1.1.2. Identifiers

Identifiers distinguish interfaces, bodies, signals. Identifiers must start with an alphabetic character (A-Z) with the rest of the name alphanumeric characters or the underscore '_':

identifier ::= letter {[underline] letter_or_digit}

Note that identifiers are treated case insensitive, i.e. lowercase letters and uppercase letters are considered to be identical. Also identifiers may not terminate with an underscore. An identifier must be a minimum of one character in length and a maximum of 80 characters in length. Identifiers may NOT be the same as one of the reserved keywords of VHDL. A list of reserved keywords can be found in Appendix C.

Example: Alpha Bus16 ID_example

B.1.1.3. Expressions

The full range of integer expressions is supported. It incorporates the usual precedence (multiplication before addition etc.). Expressions have the form:

```
expression ::=   relation {AND relation}
               | relation {OR relation}
               | relation {NAND relation} \
               | relation {NOR relation}
               | relation {XOR relation}
```

where the logic operators (AND, OR. ...) follow the normal Boolean rules. Relations are of the the forms:

```
relation ::= simple_expression [relational_operator simple_expression]

relational_operator ::=  =              -- equals
                       | /= | <>        -- not equals
                       | <              -- less than
                       | <= | =<        -- less than or equal to
                       | >              -- greater than
                       | >= | =>        -- greater than or equal to

simple_expression ::= [sign] term {adding_operator term}

adding_operator ::= + | -

term ::= factor {multiplying_operator factor}

multiplying_operator ::=  *             -- multiply
                        | /             -- divide
                        | MOD           -- modulus
                        | REM           -- remainder

factor ::=  primary [** primary]        -- to the power
          | ABS primary                 -- absolute value
```

```
                | NOT primary                    -- not

primary ::=  name
             | integer
             | (expression)
```

For the primary, the name must be a variable of type integer. Note that integer refers to both positive and negative integers.

```
Example:     6 + 2
             6 + 2 * i > 10                -- FALSE if i = 2
             (6 + 2) * i > 10              -- TRUE if i = 2
```

B.1.2. Interface Declaration

The interface declaration defines the input and output ports for an entity. The form of an interface declaration is:

```
interface_declaration ::=
        entity entity_name is
            port (formal_port_list);
            [ pins (pin_list); ]
        end [entity_name];
```

where the entity_name is an identifier. The entity_name after the **end** statement is optional but if present, must be the same as the first one.

B.1.3. Formal Port List

A formal port list describes the input and output ports for an interface declaration as

well as a component declaration. It is of the form:

formal_port_list ::= port_declaration {; port_declaration}

port_declaration ::= identifier_list : port_mode port_type

identifier_list ::= identifier {, identifier}

port_mode ::= **[in]** | **[dot]out** | **inout** | **linkage**

port_type :: = type_name | BIT

If no port_mode is specified, a default of **in** is assumed. The port_type must be either BIT of some user defined type.

B.1.4. Pin List

A pin list declaration assigns the ports of a entity to I/O pins of a LCA. Is is of the form:

pin_list ::= pin_association { ; pin_association }

pin_association ::= simple_name : PINNUM
| indexed_name : PINNUM

simple_name ::= identifier

indexed_name ::= identifier (simple_expression)

The PINNUM is the pin number of a LCA in the form of Pxx. The xx represents a two digits number.

Example:
```
entity DFF is
        port (reset, clock : in BIT; q, qB: out BIT);
        pins ( reset :      P12;
```

```
                clock :     P14;
                q :         P26;
                qB :        P27;)
        end DFF;
```

B.1.5. Architectural Body Declaration

Associated with an interface declaration is an architecture body describing the internals of the entity. The architectural description has the following form:

```
architectural_body_declaration ::=
        architecture body_name of entity_name is
                body_declarative_part
        begin
                set_of_statements
        end [body_name];

body_declarative_part ::= {body_declarative_item}

body_declarative_item ::=
                  component_declaration
                | signal_declaration
                | type_declaration
```

where body_name is unique identifier and entity_name is the name of the associated interface declaration. Note that the interface declaration must occur before the body declaration.

Example:

```
architecture A_body of four_bit_counter is
        component rdff
                port (d, reset, clock : in BIT; q, qb : out BIT);
        end component;

        signal t1, t2,t3, t4 : BIT;
```

```
begin
        bit0: port map rdff(t1, reset, clock, q0, t1);
        bit1: port map rdff(t2, reset, t1, q1, t2);
        bit2: port map rdff(t3, reset, t2, q2, t3);
        bit3: port map rdff(t4, reset, t3, q3, t4);
end A_body;
```

B.1.6. Component Declaration

Component declarations define the entities of instances declared in the set_of_statements of a body. It serves to document the design by having all declarations within the body where they are used. It also allows the use of components externally defined somewhere else. A component declaration is in the form:

```
component_declaration ::=
        component component_name port (formal_port_list)
```

where component_name is the name of the actual gate or entity.

Example: `component rdff port (d, reset, clock : in BIT; q, qb : out BIT);`

B.1.7. Signal Declaration

A signal declaration defines internal signals of an entity which are used in connecting instances together. The form of a signal declaration is:

```
signal_declaration ::= signal identifier_list : type_mark ;

identifier_list ::= identifier {, identifier}
```

```
type_mark ::= type_name
            | BIT
            | BIT_X
            | BIT_C
            | BIT_N
            | BIT_L
```

Example: **signal** t1, t2, t3, t4 : **BIT;**

B.1.8. Type Declaration

Currently the only supported user defined type is an array of bits of the form:

```
type_declaration ::=
    type type_name is array
                (simple_expression to simple_expression) of BIT;
```

where the two simple expressions define a start and end range of the array. Note that the range must be of type integer but can be either ascending or descending.

Example: **type** qbus **is array** (0 **to** 14) **of** BIT;

Thus, qbus is a type of fifteen bits starting at zero and ending at 14. Indexed names are used to refer to a single element of an array. Thus if R is of type qbus, R(2) refers to the third element of R.

B.1.9. Set of Statements

The set of statements of an architectural description defines the instances comprising an entity. The order of the statements is unimportant as it is the set and not the sequence of the statements which define the topology. The form is:

```
set_of_statements ::= architectural_statement {architectural_statement}
```

architectural_statement ::= component_instantiation_statement
| generate_statement

B.1.9.1. Component Instantiation Statement

A component instantiation statement creates one instance of a component in the body of an entity. It is of the form:

component_instantiation_statement ::=
[label:] **port map** component_name (actual_port_list);

actual_port_list ::= port_association {, port-association}

port_association ::= simple_name | indexed_name

The label is optional but is useful in identifying the instance. The actual port list must match the associated component formal port list by sequence in terms of number of ports and type of ports. A simple name is an identifier while an indexed_name refers to a single element of an array.

Example: bit0: rdff **port map** (t1, reset, clock, t1);

B.1.9.2. Generate Statement

The generate statement provides a means of selectively and repetitively calling a set of statements. This is very useful for defining regular structures. The generate statement has the form:

generate_statement ::=
[label:] generate_scheme **generate**

```
        set_of_statements
    end generate [label];

generation_scheme ::= for identifier in discrete_range
                    | if condition

condition ::= (expression)

discrete_range ::= simple_expression to simple_expression
```

If the optional label is included for the generation statement, it must appear both at the start and the end of the statement. The **for** generate scheme calls the set of statements the number of times indicated by the discrete range with the identifier taking on the present value of the discrete range:

Example:
```
bits: for i in 1 to 3 generate
          rdff port map ( t(i), reset, t(i-1), q(i), t(i) );
      end generate bits;
```
Thus the component instantiation is called three times with i equal to 1 for the first call, 2 for the second, and 3 for the last call.

The **if** generate scheme checks a Boolean expression (condition) and if **TRUE** (not zero), calls the set_of_statements. This is useful to take care of special conditions at the beginning or end of a regular structure.

Example:
```
bits: for i in 0 to 3 generate
          if i = 0 generate
              rdff( t(i), reset, clock, q(i), t(i);
          end generate;
          if i > 0 generate
              rdff( t(i), reset, t(i - 1), q(i), t(i):
          end generate;
      end generate bits;
```

This is equivalent to:

```
port map rdff( t(0), reset, clock, q(0), t(0);
port map rdff( t(i), reset, y(0), q(1), t(1);
port map rdff( t(2), reset, t(1), q(2), t(2);
port map rdff( t(3), reset, t(2), q(3), t(3);
```

B.1.10. Package Declaration

Package provides a means of declaring types once which can be used in several interface and body declarations. The form of a package declaration is:

```
package_declaration ::=
        package package_name is
                package_declarative_part
        end [package_name];

package_declarative_part ::= type_declaration; {type_declaration;}
```

Example:
```
package buses is
        type BYTE is array (7 to 0) of BIT;
        type WORD is array (15 to 0) of BIT;
end;
```

For an interface or a body to have access to the contents of a package, the "use" clauses are used. The clause must immediately precede each interface of body declaration where types are required. Note that if an interface declaration has access to the contents of a package, its associated body automatically has access to its contents (without another use clause required). The syntax of the with and use clause is:

```
use_clause ::= use package_name, { package_name };
```

Example:
```
use buses;
```

B.2. LCA Design Procedure Using the VHDL Compiler

The procedure of LCA design using the VHDL compiler is shown as follows in steps:

Step 1: Create and edit the VHDL design file on SUN workstation using the text editor **vi** or **textedit**. The filename must have the extension HDL.

Step 2: Run the VHDL compiler on SUN3 or SUN4:

vhdl2xnf3 [-Ppart] *filename*.HDL

for SUN3 and

vhdl2xnf4 [-Ppart] *filename*.HDL

for SUN4; or

vhdl2xnf3 or **vhdl2xnf4**

using the prompts provided by the compiler. The parameter part is the number of the LCA part to be used. The default is XC2064PC68. The compiler creates a XNF file with the name of the highest entity in the form of *entity_name*.XNF.

If there is no error reported by the compiler, then continue; otherwise, go back to **Step 1.**

Step 3: Transmit the file *entity_name*.XNF to the PC station for the LCA development system through ethernet.

Step 4: Use the LCA development system to implement and verify the design. See the LCA user's manuals for details.

If the design passes the verification, then continue; otherwise, go back to **Step 1.**

Step 5: Design the Printed Circuit Board using the LCA.

APPENDIX C

KEYWORD LIST

C.1. Keyword List of Supported VHDL Subset

abs	and	architecture	array
begin	component	dotout	end
entity	for	generate	if
in	inout	is	linkage
mod	nand	nor	not
of	or	out	package
pins	port	rem	signal
standard	to	type	use
xor			

C.2. Predefined Types for the Supported VHDL Subset

bit			
bit_x	bit_c	bit_n	bit_l

APPENDIX D

GRAMMAR OF SUPPORTED VHDL SUBSET

The form of the VHDL statement definitions is given using a simple variant of Backus-Naur Form:

1. Reserved keywords are given in boldface.
2. A vertical bar separates alternative items.
3. Square brackets enclose optional items.
4. Braces (i.e. {}) enclose a repeating item (zero or more repetition).

```
start_of_comment ::= --

identifier ::= letter {[underline] letter_or_digit}

expression ::=    relation {AND relation}
                | relation {OR relation}
                | relation {NAND relation} \
                | relation {NOR relation}
                | relation {XOR relation}

relation ::= simple_expression [relational_operator simple_expression]

relational_operator ::=  =          -- equals
                       | /= | <>    -- not equals
                       | <          -- less than
                       | <= | =<    -- less than or equal to
                       | >          -- greater than
                       | >= | =>    -- greater than or equal to
```

```
simple_expression ::= [sign] term {adding_operator term}

adding_operator ::= + | -

term ::= factor { multiplying_operator factor }

multiplying_operator ::=  *          -- multiply
                        | /          -- divide
                        | MOD        -- modulus
                        | REM        -- remainder

factor ::=  primary [** primary]     -- to the power
          | ABS primary              -- absolute value
          | NOT primary              -- not

primary ::=  name
           | integer
           | (expression)

interface_declaration ::=
           entity entity_name is
                  port  (formal_port_list);
                  [ pins  (pin_list); ]
           end  [entity_name];

formal_port_list ::= port_declaration {; port_declaration}

port_declaration ::= identifier_list : port_mode port_type

identifier_list ::= identifier {,  identifier}

port_mode ::= [in] | [dot]out | inout | linkage

port_type :: = type_name  |  BIT

pin_list ::=  pin_association { ; pin_association }

pin_association ::=  simple_name : PINNUM
                   | indexed_name : PINNUM
```

```
simple_name ::= identifier

indexed_name ::= identifier ( simple_expression )

architectural_body_declaration ::=
        architecture body_name of entity_name is
                body_declarative_part
        begin
                set_of_statements
        end [body_name];

body_declarative_part ::= {body_declarative_item}

body_declarative_item ::=
                  component_declaration
                | signal_declaration
                | type_declaration

component_declaration ::=
                component component_name port (formal_port_list)

signal_declaration ::= signal identifier_list : type_mark ;

identifier_list ::= identifier {, identifier}

type_mark ::= type_name
                | BIT
                | BIT_X
                | BIT_C
                | BIT_N
                | BIT_L

type_declaration ::=
        type type_name is array
                        (simple_expression to simple_expression) of BIT;

set_of_statements ::= architectural_statement {architectural_statement}

architectural_statement ::=  component_instantiation_statement
                           | generate_statement
```

component_instantiation_statement ::=
[label:] **port map** component_name (actual_port_list);

actual_port_list ::= port_association {, port-association}

port_association ::= simple_name | indexed_name

generate_statement ::=
[label:] generate_scheme **generate**
set_of_statements
end generate [label];

generation_scheme ::= **for** identifier **in** discrete_range
| **if** condition

discrete_range ::= simple_expression **to** simple_expression

condition ::= (expression)

package_declaration ::=
package package_name **is**
package_declarative_part
end [package_name];

use_clause ::= **use** package_name, { package_name };

APPENDIX E

STANDARD XNF LIBRARY LIST

E.1. List of 2000 Series Library

74-138	74-139	74-151	74-152	74-160
74-161	74-164	74-194	74-195	74-352
74-42	aclk	and2	and2b	and2b1
and2b2	and3	and3b	and3b1	and3b2
and3b3	and4	and4b	and4b1	and4b2
and4b3	and4b4	buf	c10bcprd	c10bcrd
c10bprd	c10jcr	c12jcr	c16bard	c16bcpr
c16bcprd	c16bcrd	c16bprd	c16budrd	c16jcr
c256fcrd	c2bcr	c2bcrd	c2bp	c2br
c2brd	c4bcp	c4bcr	c4bcrd	c4jcr
c6jcr	c8bcp	c8bcr	c8bcrd	c8jcr
d2-4	d2-4e	d3-8	d3-8e	fd
fdc	fdcr	fdcs	fdm	fdmr
fdmrd	fdms	fdmsd	fdr	fdrd
fds	fdsd	fdsrd	fjk	fjkrd
fjks	fjksd	fjksrd	frs	fsr

ft	ft0	ft0r	ft2	ft2r
ftp	ftprd	ftr	ftrd	fts
gadd	gclk	gcomp	geqgt	gmaj
gmux	gnd	gpar	gxor	gxtl
ibuf	inff	inv	ioff	ld
ldm	ldmrd	ldmsd	ldrd	ldsd
ldsrd	m3-1	m3-1e	m4-1	m4-1e
m8-1	m8-1e	nand2	nand2b	nand2b1
nand2b2	nand3	nand3b	nand3b1	nand3b2
nand3b3	nand4	nand4b	nand4b1	nand4b2
nand4b3	nand4b4	ndff	nor2	nor2b
nor2b1	nor2b2	nor3	nor3b	nor3b1
nor3b2	nor3b3	nor4	nor4b	nor4b1
nor4b2	nor4b3	nor4b4	obuf	obufz
or2	or2b	or2b1	or2b2	or3
or3b	or3b1	or3b2	or3b3	or4
or4b	or4b1	or4b2	or4b3	or4b4
rd4	rd8	rd8cr	rs4	rs8
rs8cr	rs8pr	rs8r	xnor2	xnor3
xnor4	xor2	xor3	xor4	

E.2. List of 3000 Series Library

74-138	74-139	74-151	74-152	74-160
74-161	74-162	74-163	74-164	74-194
74-195	74-352	74-42	aclk	and2
and2b	and2b1	and2b2	and3	and3b
and3b1	and3b2	and3b3	and4	and4b
and4b1	and4b2	and4b3	and4b4	and5
and5b	and5b1	and5b2	and5b3	and5b4
and5b5	buf	c10bcprd	c10bcrd	c10bprd
c10jcr	c12jcr	c16bard	c16bcp	c16bcpr
c16bcprd	c16bcrd	c16bprd	c16budrd	c16jcr
c256bcp	c256bcpr	c256bcr	c256bcrd	c256fcrd
c2bcp	c2bcprd	c2bcr	c2bcrd	c2bp
c2br	c2brd	c4bcp	c4bcprd	c4bcr
c4bcrd	c4jx	c4jxc	c4jxcr	c4jxcrd
c4jxrd	c6jcr	c8bcp	c8bcprd	c8bcr
c8bcrd	c8jcr	d2-4	d2-4e	d3-8
d3-8e	fd	fdc	fdcr	fdcrd
fdcs	fdm	fdmr	fdmrd	fdms
fdr	fdrd	fds	fjk	fjkrd
fjks	frs	fsr	ft	ft0
ft0r	ft0rd	ftp	ftprd	ftr
ftrd	fts	gadd	gclk	gcomp

gltgt	gmux	gnd	gxtl	ibuf
inff	inlat	inv	ld	ldrd
ldsd	lrs	m3-1	m3-1e	m4-1
m4-1e	m4-2	m8-1	m8-1e	nand2
nand2b	nand2b1	nand2b2	nand3	nand3b
nand3b1	nand3b2	nand3b3	nand4	nand4b
nand4b1	nand4b2	nand4b3	nand4b4	nand5
nand5b	nand5b1	nand5b2	nand5b3	nand5b4
nand5b5	ndff	nor2	nor2b	nor2b1
nor2b2	nor3	nor3b	nor3b1	nor3b2
nor3b3	nor4	nor4b	nor4b1	nor4b2
nor4b3	nor4b4	nor5	nor5b	nor5b1
nor5b2	nor5b3	nor5b4	nor5b5	obuf
obufz	oinv	or2	or2b	or2b1
or2b2	or3	or3b	or3b1	or3b2
or3b3	or4	or4b	or4b1	or4b2
or4b3	or4b4	or5	or5b	or5b1
or5b2	or5b3	or5b4	or5b5	osc
outff	outffz	pullup	rd4	rd4rd
rd8	rd8cr	rd8rd	rs4	rs4c
rs4cr	rs4crd	rs4rd	rs8	rs8c
rs8cr	rs8crd	rs8pr	rs8r	rs8rd
tbuf	xnor2	xnor3	xnor4	xnor5
xor2	xor3	xor4	xor5	

APPENDIX F

PROGRAM LIST

```
********  vhdi.h  ********************************************************************************************

/*
        Data structure for compilation program. Ivternal VHDL representation.
*/

#define TRUE 1
#define FALSE 0

typedef struct Symbol {                 /* symbol table entry */
        char *name;                     /* symbol name */
        int type;                       /* keywords, integer, identifier ... */
        int val;                        /* vlaue */
        struct Symbol *next;            /* to link to another */
} Symbol;

Symbol *install(), *lookup();

typedef union Datum {                   /* compiler stack type */
        int val;
        Symbol *symb;
} Datum;

extern Datum pop();

typedef int (*Inst)();                  /* compilation  instruction */
#define STOP (Inst) 0

/* compilation instruction code */
extern Inst prog[], *progp, *code();
extern intpush(), idpush(), execute();
extern pcode(), chkp(), wcode(), ucode(), ecode(), chke(), ppcode();
extern acode(), chka(), tcode(), cdcode(), fpcode();
extern idfirst(), idlist(), scode(), cicode(), chkcl(), apcode();
extern simname(), forcode(), ifcode(), chkgl();
extern eq(), ne(), lt(), le(), gt(), ge();
extern and(), nand(), or(), nor(), xor();
extern add(), sub(), mul(), div(), mod(), rem();
extern power(), negate(), abs(), not(), eval();
```

```
typedef struct Type {                   /* type declaration list */
        struct Type *next;              /* next on the linked list of type */
        char *name;                     /* type name */
        int int1,int2;                  /* type range */
} Type;

typedef struct Package {                /* package declaration list */
        struct Package *next;           /* next on the linked list */
        char *name;                     /* package name */
        Type *types;                    /* types in the package */
} Package;

typedef struct Clause {                 /* with-use clause list */
        Package *units;                 /* package name in with-use clause */
        struct Clause *next;            /* next package */
} Clause;

typedef struct Fport {                  /* formal port declaration list */
        struct Fport *next;             /* next no the linked list */
        char *name;                     /* formal port name */
        int mode;                       /* IN, OUT, ... */
        int type;                       /* BIT, TYPE_NAME */
        char *type_name;                /* if TYPE_NAME, then type name */
} Fport;

typedef struct Pport {                  /* pin list of an entity */
        struct Pport *next;             /* next on the linked list */
        char *name;                     /* formal port name */
        int index;                      /* index, -1 for simple name */
        char *pad;
} Pport;

typedef struct Cdecl {                  /* component declaration list */
        struct Cdecl *next;             /* next on the linked list */
        char *name;                     /* component name */
        int type;                       /* a component cell or an entity */
        Fport *ports;                   /* fomrmal port of the component */
} Cdecl;

typedef struct Aport {                  /* actual port declaration */
        struct Aport *next;             /* next on the linked list */
        char *name;                     /* actual port name */
        int type;                       /* signal or entity formal port */
        int index;                      /* index, -1 is simple name */
} Aport;

typedef struct Cinst {                  /* component instance list */
        struct Cinst *next;             /* next on the linked list */
        char *label;                    /* label */
        char *name;                     /* component name */
        int type;                       /* component cell or entity */
        Aport *ports;                   /* actual port */
} Cinst;
```

```
typedef struct Sig {                    /* signal declaration list */
        struct Sig *next;               /* next on the linked list */
        char *name;                     /* signal name */
        int type;                       /* BIT, X, N, C, L, TYPE_NAME*/
        char *type_name;                /* if TYPE_NAME, type name */
} Sig;

typedef struct Entity {                 /* entity list */
        struct Entity *next;            /* next on the linked list */
        char *name;                     /* entity name */
        Fport *ports;                   /* entity formal port */
        Pport *pins;                    /* entity pins */
        Clause *clause;                 /* with-use clause list */
        char *body;                     /* architectural body name */
        Cdecl *cdecls;                  /* component declaration list */
        Type *types;                    /* type declaration list */
        Sig *sigs;                      /* signal declaration list */
        Cinst *cinsts;                  /* component instance list */
} Entity;

#ifdef CODE
Entity *firstentity = NULL;
Package *firstpackage = NULL;
Type *firsttype = NULL;
Clause *firstclause = NULL;
Pport *firstpport = NULL;
Fport *firstfport = NULL;
Aport *firstaport = NULL;
Cinst *firstcinst = NULL;
Sig *firstsigl = NULL;
Cdecl *firstcdecl = NULL;
#else
extern Package *firstpackage;
extern Entity *firstentity;
extern Type *firsttype;
extern Clause *firstclause;
extern Pport *firstpport;
extern Fport *firstfport;
extern Aport *firstaport;
extern Cinst *firstcinst;
extern Sig *firstsigl;
extern Cdecl *firstcdecl;
#endif

******** vhdl.c ********************************************************************************************

/*****************************************************************************************************/
/*                                          vhdl2xnf                                                 */
/*                                                                                                   */
/*      AUTHOR:     Bing Liu                                                                         */
/*                  Dept. of Electrical and Computer Engineering                                     */
/*                  University of Manitoba                                                           */
```

```
/*                      Winnipeg, Manitoba, Canada                                  */
/*                      R3T 2N2                                                     */
/*      DATE:           Sept. 6, 1989                                               */
/*                                                                                  */
/*      REVERSION NUMBER:       2                                                   */
/*      REVERSION DATE:     Sept. 11, 1989                                          */
/*                                                                                  */
/*      DESCRIPTION:            This program reads a hierarchial VHDL structural    */
/*                      description design file and compiles it into unflattened    */
/*                      Xilinx Netlist Format (XNF) files. The XNF files are flattened */
/*                      by the FLATTEN module.                                      */
/*                                                                                  */
/*                      This program uses vhdl.h, init.c, symbol.c, parser.y, scanner.l, */
/*                      code.c, crenet.c, and modules form Xilinx, such as netread.c, */
/*                      netwrite.c, netutil.c, netparse.c, net.h and flatten.c.     */
/*                                                                                  */
/*      ARGUMENTS:  VHDL filename -- name of VHDL design file to be read.           */
/*                      parttype -- option, default is 2064pc68-33                  */
/*                                                                                  */
/*      EXTERNAL VARIABLES:                                                         */
/*              buff    -- buffer for string operation                              */
/*              yyin    -- file pointer to VHDL file, declared in LEX.              */
/*              linenio -- VHDL file line counter                                   */
/*              pgm_name-- the name of this compiler                                */
/*              pgm_ver     -- reversion number                                     */
/*              parttype-- LCA parttype number                                      */
/*              hdlname-- VHDL source filename                                      */
/*                                                                                  */
/************************************************************************************/

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

char *stralloc();

char *pgm_name;                 /* the name of this compiler */
char *pgm_ver = "2.00";         /* reversion number */
char *parttype = "2064pc68-33"; /* LCA part number */
char *vhdlname;                 /* VHDL source filename */
int lineno = 1;                 /* line counter of VHDL source file */
char buff[125];                 /* string operation buffer */
char msg[225];                  /* buffer for any message */
int vhdlerr = 0;                /* semantics error counter */
extern FILE *yyin;              /* pointer to VHDL source file, used by LEX */

main (argc, argv)  /* vhdl.c */
    int argc;
        char *argv[];
{
        pgm_name = argv[0];
        init();
```

```
if (argc==1) {                          /* No arguments - request file to read */
        hdr();                          /* print header */
        printf("VHDL Filename: ");      /* prompt for filename */
        argv[2] = gets(buff);           /* get filename */
 if((yyin=fopen(argv[2],"r"))==NULL) {  /* open file - check for errors */
                sprintf(buff, "%s.hdl", argv[2]);  /*check with extention*/
                if((yyin=fopen(buff,"r"))==NULL) {
                        idupper(buff);
                        if((yyin=fopen(buff,"r"))==NULL) {
                                printf("Can't open %s \n",argv[2]);
                                exit(0);
                        }
                }
                argv[2] = buff;
        }
        printf("LCA part type: ");      /* prompt fot parttype */
        argv[1] = gets(msg);            /* get parttype */
        if (msg[0] != '\0')
                parttype = stralloc(argv[1]);
        vhdlname = stralloc(argv[2]);   /* save VHDL filename */
        printf("VHDL design file is %s \n",argv[2]);
}
if (argc==3) {                          /* Correct # of arguments - open the file */
        hdr();                          /* print header */
        if((yyin=fopen(argv[2],"r"))==NULL) {   /* open file - check for errors */
                sprintf(buff, "%s.hdl", argv[2]);  /*check with extention*/
                if((yyin=fopen(buff,"r"))==NULL) {
                        idupper(buff);
                        if((yyin=fopen(buff,"r"))==NULL) {
                                printf("Can't open %s \n",argv[2]);
                                exit(0);
                        }
                }
                argv[2] = buff;
        }
        vhdlname = stralloc(argv[2]);   /* save VHDL filename */
        printf("VHDL design file is %s \n",argv[2]);
        strcpy(buff, argv[1]);
        if (buff[0] != '-' || buff[1] != 'p') {
                printf("\nIllegal option\n");
                exit(0);
        }
        parttype = &buff[2];
        parttype = stralloc(parttype);
}
if (argc==2) {                          /* Correct # of arguments - open the file */
        hdr(); /* print header */
        if((yyin=fopen(argv[1],"r"))==NULL) {   /* open file - check for errors */
                sprintf(buff, "%s.hdl", argv[1]);  /*check with extention*/
                if((yyin=fopen(buff,"r"))==NULL) {
                        idupper(buff);
                        if((yyin=fopen(buff,"r"))==NULL) {
```

```
                                        printf("Can't open %s \n",argv[1]);
                                        exit(0);
                                }
                        }
                        argv[1] = buff;
                }
                vhdlname = stralloc(argv[1]);      /* save VHDL filename */
                printf("VHDL design file is %s \n",argv[1]);
        }
        if (argc > 3) {  /* Too many inputs - issue error */
                hdr();  /* print header */
                printf("Error: Too many inputs.\nUsage: vhdl2xnf [-pparttype] <filename>\n");
                exit(0);
        }

    initcode();

        if (yyparse() !=1 ) {
                printf("\nPassing VHDL sytanx checking.\n");
                execute(prog);
        }
        else {
                printf("\nStoped due to VHDL sytanx error.\n");
                printf("Correct the error, please try again.\n");
                exit(0);
        }

        crenet();          /* create hierarchical Xilinx Netlist Files */
        if (vhdlerr > 0) {
                printf("\nStopped due to VHDL semantic error.\n");
                exit(0);
        }

        sprintf(buff, "%s.HNF", firstentity->name); /* highest level XNF name */
        flatten(buff); /* flatten hierarchical Xilinx Netlist Files */

        printf("\nCompilation successful!\n");
}
/* ---- end of main ----*/

/*****************************************************************************/
/*                                exceerror                                  */
/*                                                                           */
/* AUTHOR:      Bing Liu                                                     */
/*              Dept. of Electrical and Computer Engineering                 */
/*              University of Manitoba                                       */
/*              Winnipeg, Manitoba, Canada                                   */
/*              R3T 2N2                                                      */
/* DATE:        August 6, 1989                                               */
/*                                                                           */
/* REVERSION NUMBER:       2                                                 */
/* REVERSION DATE:  Sept. 11, 1989                                           */
/*                                                                           */
```

```
/* DESCRIPTION:  This program reads a hierarchial VHDL structural        */
/*                                                                       */
/* ARGUMENTS:  VHDL filename -- name of VHDL design file to be read.     */
/*                                                                       */
/*************************************************************************/

execerror (s, t)                        /* recover from run-time error */
    char *s, *t;
{
    warning(s, t);
        vhdlerr++;
}
/*---end of exerror---*/

/*************************************************************************/
/*                                  yyerror                              */
/*                                                                       */
/* AUTHOR:      Bing Liu                                                 */
/*              Dept. of Electrical and Computer Engineering             */
/*              University of Manitoba                                   */
/*              Winnipeg, Manitoba, Canada                               */
/*              R3T 2N2                                                  */
/* DATE:        August 6, 1989                                           */
/*                                                                       */
/* REVERSION NUMBER:        2                                            */
/* REVERSION DATE:  Sept. 11, 1989                                       */
/*                                                                       */
/* DESCRIPTION:  This program reads a hierarchial VHDL structural        */
/*                                                                       */
/* ARGUMENTS:  VHDL filename -- name of VHDL design file to be read.     */
/*                                                                       */
/*************************************************************************/

yyerror (s)                             /* called for yacc syntax error */
    char *s;
{
    warning (s, (char *) 0);            /* indicate syntax error line number */
     fprintf ( stderr, " near line %d\n", lineno);
}
/*---end of yyerror--*/

/*************************************************************************/
/*                                  warning                              */
/*                                                                       */
/* AUTHOR:      Bing Liu                                                 */
/*              Dept. of Electrical and Computer Engineering             */
/*              University of Manitoba                                   */
/*              Winnipeg, Manitoba, Canada                               */
/*              R3T 2N2                                                  */
/* DATE:        August 6, 1989                                           */
/*                                                                       */
/* REVERSION NUMBER:        2                                            */
/* REVERSION DATE:  Sept. 11, 1989                                       */
```

```
/*                                                                          */
/* DESCRIPTION:  This program reads a hierarchial VHDL structural           */
/*                                                                          */
/* ARGUMENTS:  VHDL filename -- name of VHDL design file to be read.        */
/*                                                                          */
/****************************************************************************/

warning (s, t)  /* print  warning message */
     char *s,  *t;
{
      fprintf (stderr,"\n%s: %s", vhdlname, s);
     if (t)
          fprintf (stderr, " %s", t);
         return;
}
/*---end of warning---*/

/****************************************************************************/
/*                                    hdr                                   */
/*                                                                          */
/* AUTHOR:      Bing Liu                                                    */
/*              Dept. of Electrical and Computer Engineering                */
/*              University of Manitoba                                      */
/*              Winnipeg, Manitoba, Canada                                  */
/*              R3T 2N2                                                     */
/* DATE:        Sept. 11, 1989                                              */
/****************************************************************************/

hdr()   /* Print header info */
{
      printf("\nvhdl2xnf, Version 2.00, November 2, 1989");
      printf("\n(c) Copyright 1989\nBing Liu");
      printf("\nDepartment of Electrical & Computer Engineering");
      printf("\nUniversity of Manitoba");
      printf("\nWinnipeg, Manitoba");
      printf("\nCanada R3T 2N2\n\n\n");
         return;
}
/*---end of hdr---*/

********     init.c     ***************************************************************************************************

#include "vhdl.h"
#include "x.tab.h"

static struct {                         /* Keywords */
         char *name;
         int kval;
} keywords[] = {
         "ABS",                         ABS,
         "AFTER",                       AFTER,
         "ALIAS",                       ALIAS,
```

```
"AND",                  AND,
"ARCHITECTURE",         ARCHITECTURE,
"ARRAY",                ARRAY,
"ASSERTION",            ASSERTION,
"ATTRIBUTE",            ATTRIBUTE,
"BEGIN",                BEGIN_V,        /* avoid conflict with lex default */
"BEHAVIORAL",           BEHAVIORAL,
"BODY",                 BODY,
"CASE",                 CASE,
"COMPONENT",            COMPONENT,
"CONNECT",              CONNECT,
"CONSTANT",             CONSTANT,
"CONVERT",              CONVERT,
"DOT",                  DOTOUT,
"DOWNTO",               DOWNTO,
"ELSE",                 ELSE,
"ELSIF",                ELSIF,
"END",                  END,
"ENTITY",               ENTITY,
"EXIT",                 EXIT,
"FOR",                  FOR,
"FUNCTION",             FUNCTION,
"GENERATE",             GENERATE,
"GENERIC",              GENERIC,
"IF",                   IF,
"IN",                   IN,
"INOUT",                INOUT,
"IS",                   IS,
"LINKAGE",              LINKAGE,
"LOOP",                 LOOP,
"MOD",                  MOD,
"MAP",                  MAP,
"NAND",                 NAND,
"NEXT",                 NEXT,
"NOR",                  NOR,
"NOT",                  NOT,
"NULL",                 NULL_V,         /* avoid conflict with lex default */
"OF",                   OF,
"OR",                   OR,
"OTHERS",               OTHERS,
"OUT",                  OUT,
"PACKAGE",              PACKAGE,
"PORT",                 PORT,
"PINS",                 PINS,
"RANGE",                RANGE,
"RECORD",               RECORD,
"REM",                  REM,
"REPORT",               REPORT,
"RESOLVE",              RESOLVE,
"RETURN",               RETURN,
"SEVERITY",             SEVERITY,
"SIGNAL",               SIGNAL,
"STANDARD",             STANDARD,
```

```
        "STATIC",               STATIC,
        "SUBTYPE",              SUBTYPE,
        "THEN",                 THEN,
        "TO",                   TO,
        "TYPE",                 TYPE,
        "UNITS",                UNITS,
        "USE",                  USE,
        "VARIABLE",             VARIABLE,
        "WHEN",                 WHEN,
        "WHILE",                WHILE,
        "WITH",                 WITH,
        "XOR",                  XOR,
        "BIT",                  BIT,
        "BIT_X",                BIT_X,
        "BIT_C",                BIT_C,
        "BIT_N",                BIT_N,
        "BIT_L",                BIT_L,
        "0",                    0
};

init()          /*install keywords, primitives and library in tables */
{
        int i;
        Symbol *s;
        for (i = 0; keywords[i].kval; i++)
                install (keywords[i].name, keywords[i].kval, 0);
        return;
}
/*---- end of init -----*/

********    symbol.c    ********************************************************************************************

#include "vhdl.h"
#include "x.tab.h"

Symbol *symlist = 0;                    /* symbol table: linkede list */

Symbol *lookup(s)                       /* find s in symbol table */
        char *s;
{
        Symbol *sp;
        for (sp = symlist; sp != (Symbol *) 0; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return (sp);
        return (0);         /* 0 ==> not found */
}
Symbol *install(s, t, d)                /* install s in symbol table */
        char *s;
        int t, d;
{
        Symbol *sp;
        char *emalloc();
```

```
        sp = (Symbol *) emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s) + 1);      /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->val = d;
        sp->next = symlist;                     /*put at front of list */
        symlist = sp;
        return sp;
}

char *emalloc(n)                        /* check return from malloc */
        unsigned n;
{
        char *p, *malloc();

        p = malloc(n);
        if (p == 0) {
                execerror("error 0: Out of memory", (char *) 0);
                exit(0);
        }
        return p;
}

********    parser.y    ********************************************************************************************

%{
#include "vhdl.h"
#define code2(c1, c2)           code(c1); code(c2)
#define code3(c1, c2, c3)       code(c1); code(c2); code(c3)
%}
%union {                    /* stack type */
        Symbol *symb;           /* symbol table pointer */
        Inst *inst;             /* translation instruction */
}
%token  <symb> AFTER ALIAS ARCHITECTURE ARRAY ASSERTION ATTRIBUTE
%token  <symb> BEGIN_V    /* avoid conflict with lex default */
%token  <symb> BEHAVIORAL BIT BODY
%token  <symb> CASE COMPONENT CONNECT CONSTANT CONVERT
%token  <symb> DOTOUT DOWNTO
%token  <symb> ELSE ELSIF END ENTITY EXIT
%token  <symb> FOR FUNCTION
%token  <symb> GENERATE GENERIC
%token  <symb> IF IN INOUT IS
%token  <symb> LINKAGE LOOP
%token  <symb> MAP
%token  <symb> NEXT
%token  <symb> NULL_V    /* avoid conflict with lex default */
%token  <symb> OF OTHERS OUT
%token  <symb> PACKAGE PORT
%token  <symb> RANGE RECORD REPORT RESOLVE RETURN
%token  <symb> SEVERITY SIGNAL STANDARD STATIC SUBTYPE
%token  <symb> THEN TO TYPE
%token  <symb> UNITS USE
```

```
%token  <symb> VARIABLE
%token  <symb> WHEN WHILE WITH     /*end of keywords list, not all are used*/
%token  <symb> PINS IDENTIFIER LABEL
%token  <symb> PINNUM BIT_X BIT_C BIT_N BIT_L INTEGER COM_NAME PACKAGE_NAME
%token  <symb> ENTITY_NAME TYPE_NAME BODY_NAME SIG_NAME FPORT_NAME
%type   <symb> type_name package_name
%type   <inst> generate_statement set_of_statements generation_scheme for
%type   <inst> architectural_statement component_instantiation_statement
%type   <inst> condition simple_expression term factor primary expression
%type   <inst> label relation if entity_name component_name generate
%right  '='
%left   OR NOR XOR
%left   AND NAND
%left   EQ NE LT LE GT GE       /* relational_operator */
%left   '+' '-'         /* adding_operator */
%left   '*' '/' MOD REM             /* multiplying_operator */
%left   UNARYMINUS NOT ABS
%left  POW                  /* popwer ** */
%%
list:       /* nothing */
          | list package_declaration
          | list use_clause ';'
          | list entity_declaration
          | list architectural_body_declaration
          | list error          { yyerrok;
                                  printf ("\n");
                                  return 1; /* stop when error*/ }
          ;
package_declaration:
           PACKAGE package_name IS
              package_declarative_part
           END ';' { code(pcode); }
          | PACKAGE package_name IS
              package_declarative_part
           END package_name ';' { code2(chkp, pcode); }
          ;
package_declarative_part:
           type_declaration
          | package_declarative_part type_declaration
          ;
use_clause:
           USE unit                 { code(ucode); }
          | use_clause ',' unit       { code(ucode); }
          ;
unit:
           package_name
          ;
package_name:
           IDENTIFIER               { code2(idpush, (Inst)$1); }
          ;
entity_declaration:
           ENTITY entity_name IS
                 entity_body
```

```
        END ';'                 { code(ecode); }
      | ENTITY entity_name IS
            entity_body
        END entity_name ';'     { code2(chke, ecode); }
      ;
entity_body:
        port_clause ';'
      | port_clause ';' pin_clause ';'
      ;
port_clause:
        PORT '(' formal_port_list ')'
      ;
pin_clause:
        PINS '(' pin_list ')'
      ;
pin_list:
        pin_association
      | pin_list ';' pin_association
      ;
pin_association:
        simple_name ':' PINNUM      { code3(idpush, (Inst)$3, ppcode); }
      | indexed_name ':' PINNUM     { code3(idpush, (Inst)$3, ppcode); }
      ;
architectural_body_declaration:
        ARCHITECTURE body_name OF entity_name IS
            body_declarative_part
        BEGIN_V
            set_of_statements
        END ';'                 { code(acode); }
      | ARCHITECTURE body_name OF entity_name IS
            body_declarative_part
        BEGIN_V
            set_of_statements
        END body_name ';'       { code2(chka, acode); }
      ;
body_name:
        IDENTIFIER              { code2(idpush, (Inst)$1); }
      ;
body_declarative_part:
        body_declarative_item
      | body_declarative_part body_declarative_item
      ;
body_declarative_item:
        type_declaration
      | component_declaration
      | signal_declaration
      ;
type_declaration:
        TYPE type_name IS ARRAY
            '(' simple_expression TO simple_expression ')'
                OF BIT ';' { code(tcode); }
      ;
```

```
component_declaration:
          COMPONENT component_name port_clause ';' { code(cdcode); }
        | COMPONENT component_name port_clause ';'
          END COMPONENT ';' { code(cdcode); }
        ;
component_name:
          IDENTIFIER     { $$ = code2(idpush, (Inst)$1); }
        ;
entity_name:
          IDENTIFIER     { $$ = code2(idpush, (Inst)$1); }
        ;
formal_port_list:
          port_declaration
        | formal_port_list ';' port_declaration
        ;
port_declaration:
          identifier_list ':' port_mode port_type    { code(fpcode); }
        ;
identifier_list:
          IDENTIFIER     { code3(idfirst, idpush, (Inst)$1); }
        | identifier_list ',' IDENTIFIER { code3(idlist, idpush, (Inst)$3); }
        ;
port_mode:
          IN             { code2(idpush, (Inst)$1); }
        | OUT            { code2(idpush, (Inst)$1); }
        | DOTOUT         { code2(idpush, (Inst)$1); }
        | INOUT          { code2(idpush, (Inst)$1); }
        | LINKAGE        { code2(idpush, (Inst)$1); }
        ;
port_type:
          type_name
        | BIT            { code2(idpush, (Inst)$1); }
        ;
signal_declaration:
          SIGNAL identifier_list ':' signal_type ';' { code(scode); }
        ;
signal_type:
          type_name
        | BIT            { code2(idpush, (Inst)$1); }
        | BIT_X          { code2(idpush, (Inst)$1); }
        | BIT_C          { code2(idpush, (Inst)$1); }
        | BIT_N          { code2(idpush, (Inst)$1); }
        | BIT_L          { code2(idpush, (Inst)$1); }
        ;
type_name:
          IDENTIFIER     { code2(idpush, (Inst)$1); }
        ;
set_of_statements:
          architectural_statement
        | set_of_statements architectural_statement
        ;
architectural_statement:
          component_instantiation_statement
```

```
        | generate_statement
        ;
component_instantiation_statement:
        label ':' component_name PORT MAP '(' actual_port_list ')' ';' {
                code2(cicode, chkcl);}
        | component_name PORT MAP '(' actual_port_list ')' ';'
                { code(cicode); }
        ;
actual_port_list:
        port_association                              { code(apcode); }
        | actual_port_list ',' port_association { code(apcode); }
        ;
port_association:
        simple_name
        | indexed_name
        ;
simple_name:
        IDENTIFIER    { code3(simname, idpush, (Inst)$1); }
        ;
indexed_name:
        IDENTIFIER '(' simple_expression ')' { code2(idpush, (Inst)$1); }
        ;
generate_statement:
        label ':' generation_scheme generate
                        set_of_statements
                  END generate label ';' {
                ($3)[1] = (Inst)$5;
                ($3)[2] = (Inst)$7;
                code(chkgl); }
        | generation_scheme generate
                set_of_statements
        END generate ';' {
                ($1)[1] = (Inst)$3;
                ($1)[2] = (Inst)$5; }
        ;
generate:
        GENERATE    { code(STOP); $$ = progp; }
        ;
label:
        IDENTIFIER    { $$ = code2(idpush, (Inst)$1); }
        ;
generation_scheme:
        for IDENTIFIER IN discrete_range { code2(idpush, (Inst)$2); }
        | if condition
        ;
for:    FOR    { $$ = code3(forcode, STOP, STOP); }
        ;
discrete_range:
        simple_expression TO simple_expression
        ;
if:     IF     { $$ = code3(ifcode, STOP, STOP); }
        ;
```

```
condition:
        expression
        ;
expression:
        relation
        | expression AND expression     { code(and); }
        | expression NAND expression    { code(nand); }
        | expression OR expression      { code(or); }
        | expression NOR expression     { code(nor); }
        | expression XOR expression     { code(xor); }
        ;
relation:
        simple_expression
        | simple_expression EQ simple_expression      { code(eq); }
        | simple_expression NE simple_expression      { code(ne); }
        | simple_expression LT simple_expression      { code(lt); }
        | simple_expression LE simple_expression      { code(le); }
        | simple_expression GT simple_expression      { code(gt); }
        | simple_expression GE simple_expression      { code(ge); }
        ;
simple_expression:
        term
        | '-' term %prec UNARYMINUS { $$ = $2; code(negate); }
        | simple_expression '+' simple_expression { code(add); }
        | simple_expression '-' simple_expression { code(sub); }
        ;
term:   factor
        | term '*' term { code(mul); }
        | term '/' term { code(div); }
        | term MOD term { code(mod); }
        | term REM term { code(rem); }
        ;
factor:
        primary
        | primary POW primary { code(power); }
        | ABS primary   { $$ = $2; code(abs); }
        | NOT primary   { $$ = $2; code(not); }
        ;
primary:
        IDENTIFIER      { $$ = code3(idpush, (Inst)$1, eval); }
        | INTEGER       { $$ = code2(intpush, (Inst)$1); }
        | '(' expression ')' { $$ = $2; }
        ;
%%

/* end of grammar */

********    x.tab.h    ***********************************************************************************************

typedef union {                 /* stack type */
        Symbol *symb;           /* symbol table pointer */
        Inst *inst;             /* translation instruction */
} YYSTYPE;
```

```
extern YYSTYPE yylval;
# define AFTER 257
# define ALIAS 258
# define ARCHITECTURE 259
# define ARRAY 260
# define ASSERTION 261
# define ATTRIBUTE 262
# define BEGIN_V 263
# define BEHAVIORAL 264
# define BIT 265
# define BODY 266
# define CASE 267
# define COMPONENT 268
# define CONNECT 269
# define CONSTANT 270
# define CONVERT 271
# define DOTOUT 272
# define DOWNTO 273
# define ELSE 274
# define ELSIF 275
# define END 276
# define ENTITY 277
# define EXIT 278
# define FOR 279
# define FUNCTION 280
# define GENERATE 281
# define GENERIC 282
# define IF 283
# define IN 284
# define INOUT 285
# define IS 286
# define LINKAGE 287
# define LOOP 288
# define MAP 289
# define NEXT 290
# define NULL_V 291
# define OF 292
# define OTHERS 293
# define OUT 294
# define PACKAGE 295
# define PORT 296
# define RANGE 297
# define RECORD 298
# define REPORT 299
# define RESOLVE 300
# define RETURN 301
# define SEVERITY 302
# define SIGNAL 303
# define STANDARD 304
# define STATIC 305
# define SUBTYPE 306
# define THEN 307
```

```
# define TO 308
# define TYPE 309
# define UNITS 310
# define USE 311
# define VARIABLE 312
# define WHEN 313
# define WHILE 314
# define WITH 315
# define PINS 316
# define IDENTIFIER 317
# define LABEL 318
# define PINNUM 319
# define BIT_X 320
# define BIT_C 321
# define BIT_N 322
# define BIT_L 323
# define INTEGER 324
# define COM_NAME 325
# define PACKAGE_NAME 326
# define ENTITY_NAME 327
# define TYPE_NAME 328
# define BODY_NAME 329
# define SIG_NAME 330
# define FPORT_NAME 331
# define OR 332
# define NOR 333
# define XOR 334
# define AND 335
# define NAND 336
# define EQ 337
# define NE 338
# define LT 339
# define LE 340
# define GT 341
# define GE 342
# define MOD 343
# define REM 344
# define UNARYMINUS 345
# define NOT 346
# define ABS 347
# define POW 348

******** scanner.l ********************************************************************************

%{
#include "vhdl.h"
#include "x.tab.h"
#include <ctype.h>
int lineno;
%}
%%
[ \t] { ;                          /* ignoring spaces and tabs */ }
"--".*\n { lineno++;               /* ignoring comments, but count lines */ }
```

```
[0-9]+ {                        /* integer token */
        int d;
        sscanf (yytext, "%d", &d);
        yylval.symb = install("", INTEGER, d);
        return (INTEGER);
        }

P[0-9]+ {               /* pin number token */
        Symbol *s;
        if ((s=lookup(yytext)) == 0)
                s = install(yytext, PINNUM, 0);
        yylval.symb = s;
        return (PINNUM);
        }

[a-zA-Z][a-zA-Z0-9]*_*[a-zA-Z0-9]* {    /* identifier token */
        Symbol *s;
        idupper(yytext);                /* convert all inentifier to upper case */
        if ((s=lookup(yytext)) == 0)
                s = install(yytext, IDENTIFIER, 0);
        yylval.symb = s;
        return (s->type);
        }
"="     {
        return (EQ);
        }
"/="|"<>"       {
        return (NE);
        }
"<"     {
        return (LT);
        }
"<="|"=<"       {
        return (LE);
        }
">"     {
        return (GT);
        }
">="|"=>"       {
        return (GE);
        }
\n      {       /* new line */
        lineno++;
        }
.   {   /* everything else */
        return(yytext[0]);
        }
%%

yywrap()
{
        code(STOP);
        printf ("Total %d lines have been read.\n", --lineno);
```

```
        return (1);
}

idupper(s)
        char *s;
{
        char c;

        for (c = *s; c != '\0'; c = *(++s))
                if ((c >= 'a') && (c <= 'z'))
                        *s = toupper(c);
        return;
}

******** code.c ********************************************************************************************

#include <stdio.h>
#include <string.h>
#include <math.h>

#define CODE

#include "vhdl.h"
#include "x.tab.h"

#define NSTACK 256
static Datum stack[NSTACK];                     /* interpreter stack */
static Datum *stackp;                           /* next free spot on stack */

#define NPROG 2000
Inst prog[NPROG];                               /* translater */
Inst *progp;                                    /* next free sport for code generation */
Inst *pc;                                       /* programm counter during execution */

initcode()                                      /* initialization for code generation */
{
        stackp = stack;
        progp = prog;
}

push(d)                                         /* push d on to stack */
        Datum d;
{
        if(stackp >= &stack[NSTACK])
                execerror("compiler error 1: stack overflow",(char *) 0);
        *stackp++ = d;
}

Datum pop()
{
        if(stackp <= stack)
                execerror("compiler error 2: stack underflow",(char *) 0);
```

```
        return *--stackp;
}

Inst *code(t)                                   /* install one translation or operand */
        Inst t;
{
        Inst *oprogp = progp;
        if(progp >= &prog[NPROG])
                execerror("compiler error 3: Design too big",(char *) 0);
        *progp++ = t;
        return oprogp;
}

execute(p)                                      /* run the translator */
        Inst *p;
{
        for(pc = p; *pc != STOP;)
                (*(*pc++))();
}

intpush()                                       /* push integer onto stack */
{
        Datum d;

        d.val = ((Symbol *)(*pc++))->val;
        push(d);
}

idpush()                                        /* push identifier onto stack */
{
        Datum d;
        d.symb = (Symbol *)(*pc++);
        push(d);
}

pcode()
{
        Datum d;
        Package *pk;

        d = pop();

        pk = (Package*) emalloc(sizeof(Package));
        pk->name = (char*) emalloc(strlen(d.symb->name) + 1);
        strcpy(pk->name, d.symb->name);
        d.symb->type = PACKAGE_NAME;

        pk->types = firsttype;
        firsttype = NULL;
        pk->next = firstpackage;
        firstpackage = pk;
}
```

```
chkp()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        if (strcmp(d1.symb->name, d2.symb->name) != 0)
                warning("Warn 1: Package name not match", d1.symb->name);
        push(d1);
}

ucode()
{
        Datum d;
        Clause *clp;
        Package *pk;

        d = pop();
        pk = firstpackage;

        if (d.symb->type != PACKAGE_NAME)
                execerror("Compiler Error 1: Undefined package in with clause", d.symb->name);
        clp = (Clause *) emalloc(sizeof(Clause));
        clp->units = pk;
        clp->next = firstclause;
        firstclause = clp;
}

ecode()
{
        Datum d;
        Entity *ep;


        d = pop();
        ep = (Entity *) emalloc(sizeof(Entity));
        ep->name = (char *) emalloc(strlen(d.symb->name) + 1);
        strcpy(ep->name, d.symb->name);
        d.symb->type = ENTITY_NAME;
        ep->clause = firstclause;
        firstclause = NULL;
        ep->ports = firstfport;
        firstfport = NULL;
        ep->pins = firstpport;
        firstpport = NULL;

        ep->next = firstentity;
        firstentity = ep;
}

chke()
{
```

```
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        if (strcmp(d1.symb->name, d2.symb->name) != 0)
                warning("WARN 2: Entity name not match", d1.symb->name);
        push(d1);

}

ppcode()
{
        Datum d1, d2, d3;
        Pport *pp;

        d3 = pop();     /* pin number */
        d2 = pop();     /* port name */
        d1 = pop();     /* index, if -1 simple name */

        if (d2.symb->type != FPORT_NAME)
                execerror("Compiler Error 2: Undefined entity port", d2.symb->name);
        pp = (Pport *) emalloc(sizeof(Pport));
        pp->name = (char *) emalloc(strlen(d2.symb->name) + 1);
        strcpy(pp->name, d2.symb->name);
        pp->pad = (char *) emalloc(strlen(d3.symb->name) + 1);
        strcpy(pp->pad, d3.symb->name);
        pp->index = d1.val;

        pp->next = firstpport;
        firstpport = pp;
}

acode ()
{
        Datum d1, d2;
        Entity *ep;
        Type *tp, *tp1;
        Clause *clp;

        d2 = pop();
        d1 = pop();

        ep = firstentity;
        while ((ep != NULL) && (strcmp(ep->name, d2.symb->name) != 0))
                ep = ep->next;
        if ( ep == NULL )
                execerror( "Compiler Error 3: Undefined entity", d2.symb->name);
        ep->body = (char *) emalloc(strlen(d1.symb->name) + 1);
        strcpy(ep->body, d1.symb->name);
        for (clp = ep->clause; clp != NULL; clp = clp->next) {
                for (tp = (clp->units)->types; tp != NULL; tp = tp->next) {
                        tp1 = (Type*) emalloc(sizeof(Type));
                        tp1->name = (char*) emalloc(strlen(tp->name) + 1);
```

```
                        strcpy(tp1->name, tp->name);
                        tp1->int1 = tp->int1;
                        tp1->int2 = tp->int2;
                        tp1->next = firsttype;
                        firsttype = tp1;
                }
        }
        ep->types = firsttype;
        firsttype = NULL;
        ep->cdecls = firstcdecl;
        firstcdecl = NULL;
        ep->sigs = firstsigl;
        firstsigl = NULL;
        ep->cinsts = firstcinst;
        firstcinst = NULL;
}

chka()
{
        Datum d1, d2, d3;

        d3 = pop();     /* body name at the end */
        d2 = pop();     /* entity name */
        d1 = pop();     /* bodyname at the begining */
        if (strcmp(d1.symb->name, d3.symb->name) != 0)
                warning("WARN 3: Body name not match", d1.symb->name);
        push(d1);
        push(d2);
}


tcode()
{
        Datum d1, d2, d3;
        Type  *tp;

        d3 = pop();
        d2 = pop();
        d1 = pop();

        tp = (Type*) emalloc(sizeof(Type));

        tp->name = (char*) emalloc(strlen(d1.symb->name) + 1);
        strcpy(tp->name, d1.symb->name);
        d1.symb->type = TYPE_NAME;
        tp->int1 = d2.val;
        tp->int2 = d3.val;
        tp->next = firsttype;
        firsttype = tp;
}

cdcode ()
{
```

```
        Datum d;
        Cdecl *cdp;

        d = pop();
        cdp = (Cdecl *) emalloc(sizeof(Cdecl));
        cdp->name = (char *) emalloc(strlen(d.symb->name) + 1);
        strcpy(cdp->name, d.symb->name);
        if (d.symb->type != ENTITY_NAME)
                if (d.symb->type == IDENTIFIER)
                        d.symb->type = COM_NAME;
        cdp->type = d.symb->type;
        cdp->ports = firstfport;
        firstfport = NULL;

        cdp->next = firstcdecl;
        firstcdecl = cdp;
}

fpcode ()
{
        Datum d1, d2, d3, d4;
        Fport *fp, *firstfp = NULL;

        d4 = pop();     /* port type */
        d3 = pop();     /* port mode */
        d1.val = TRUE;
        while (d1.val) {
                d2 = pop();     /* port name */
                d1 = pop();     /* TRUE more, FALSE last */
                fp = (Fport *) emalloc(sizeof(Fport));
                fp->name = (char *) emalloc(strlen(d2.symb->name) + 1);
                strcpy(fp->name, d2.symb->name);
                d2.symb->type = FPORT_NAME;
                fp->type = d4.symb->type;
                if (fp->type == TYPE_NAME) {
                        fp->type_name = (char *) emalloc(strlen(d4.symb->name) + 1);
                        strcpy(fp->type_name, d4.symb->name);
                }
                fp->mode = d3.symb->type;
                fp->next = firstfp;
                firstfp = fp;
        }
        for (fp = firstfp; fp != NULL; fp = firstfp) {
                firstfp = fp->next;
                fp->next = firstfport;
                firstfport = fp;
        }
}

idfirst()
{
        Datum d;
```

```
        d.val = FALSE;
        push(d);
}

idlist()
{
        Datum d;
        d.val = TRUE;
        push(d);
}

scode()
{
        Datum d1, d2, d3;
        Sig *sp;

        d3 = pop();     /* signal type */
        d1.val = TRUE;
        while (d1.val) {
                d2 = pop();     /* signal name */
                d1 = pop();     /* TRUE more, FALSE last */
                sp = (Sig *) emalloc(sizeof(Sig));
                sp->name = (char *) emalloc(strlen(d2.symb->name) + 1);
                strcpy(sp->name, d2.symb->name);
                d2.symb->type = SIG_NAME;
                sp->type = d3.symb->type;
                if (sp->type == TYPE_NAME) {
                        sp->type_name = (char *) emalloc(strlen(d3.symb->name) + 1);
                        strcpy(sp->type_name, d3.symb->name);
                }
                sp->next = firstsigl;
                firstsigl = sp;
        }
}

cicode()
{
        Datum d;
        Cinst *cip;

        d = pop();      /* component name */
        cip = (Cinst *) emalloc(sizeof(Cinst));
        cip->name = (char *) emalloc(strlen(d.symb->name) + 1);
        strcpy(cip->name, d.symb->name);
        cip->type = d.symb->type;
        cip->label = NULL;
        cip->ports = firstaport;
        firstaport = NULL;
        cip->next = firstcinst;
        firstcinst = cip;

}
```

```
chkcl()
{
        Datum d;
        Cinst *cip = firstcinst;
        d = pop();      /* component instance label */
        cip->label = (char *) emalloc(strlen(d.symb->name) + 1);
        strcpy(cip->label, d.symb->name);
}

apcode ()
{
        Datum d1, d2;
        Aport *ap;

        d2 = pop();     /* port name */
        d1 = pop();     /* index, if -1 simple name */

        ap = (Aport *) emalloc(sizeof(Aport));
        ap->name = (char *) emalloc(strlen(d2.symb->name) + 1);
        strcpy(ap->name, d2.symb->name);
        ap->type = d2.symb->type;
        ap->index = d1.val;

        ap->next = firstaport;
        firstaport = ap;
}

simname()
{
        Datum d;
        d.val = -1;
        push(d);
}

forcode()
{
        Datum d1,d2,d3;
        Inst  *savepc = pc;

        execute(savepc + 2); /* discrete range */

        d3 = pop();
        d2 = pop();
        d1 = pop();

        if(d3.symb->type != IDENTIFIER && d3.symb->type != VARIABLE)
                execerror("Compiler Error 4: Not a variable", d3.symb->name);
        d3.symb->type = VARIABLE;
        for(d3.symb->val = d1.val; d3.symb->val <= d2.val; d3.symb->val++) {
                execute(*((Inst **)(savepc))); /* set of statement */
        }
        pc = *((Inst **)(savepc + 1)); /* continue */
```

```
}

ifcode()
{
        Datum d;
        Inst  *savepc = pc;

        execute(savepc + 2);  /* condition */
        d = pop();
        if(d.val) {
                execute(*((Inst **)(savepc))); /* set of statement */
        }
        pc = *((Inst **)(savepc + 1)); /* continue */
}

chkgl()
{
        Datum d1, d2;
        d2 = pop();       /* end label */
        d1 = pop();       /* begin label */
        if (strcmp(d2.symb->name, d1.symb->name) != 0)
                warning("WARN 4: Label not match in generate sataement", d1.symb->name);
}

and()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (d1.val != 0 && d2.val != 0);
        push(d1);
}

nand ()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (d1.val == 0 || d2.val == 0);
        push(d1);
}

or()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (d1.val != 0 || d2.val != 0 );
        push(d1);
}
```

```
nor()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (d1.val == 0 && d2.val == 0);
        push(d1);
}

xor()
{
        Datum d1, d2;
        d1 = pop();
        d2 = pop();
        d1.val = ((d1.val == 0 && d2.val != 0) || (d1.val != 0 && d2.val == 0));
        push(d1);
}

eq()
{
        Datum d1,d2;
        d2 = pop();
        d1 = pop();
        d1.val = (d1.val == d2.val);
        push(d1);
}

ne()
{
        Datum d1,d2;

        d2 = pop();
        d1 = pop();
        d1.val = (d1.val != d2.val);
        push(d1);
}

gt()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        d1.val = (d1.val > d2.val);
        push(d1);
}

ge()
{
        Datum d1, d2;

        d2 = pop();
```

```
        d1 = pop();
        d1.val = (d1.val >= d2.val);
        push(d1);
}

lt()
{
        Datum d1,d2;

        d2 = pop();
        d1 = pop();
        d1.val = (d1.val < d2.val);
        push(d1);
}

le()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        d1.val = (d1.val <= d2.val);
        push(d1);
}

negate()
{
        Datum d;

        d = pop();
        d.val = -d.val;
        push(d);
}

add()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        d1.val += d2.val;
        push(d1);
}

sub()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        d1.val -= d2.val;
        push(d1);
}
```

```
mul()
{
        Datum d1,d2;

        d2 = pop();
        d1 = pop();

        d1.val *= d2.val;
        push(d1);
}

div()
{
        Datum d1, d2;

        d2 = pop();
        d1 = pop();
        if(d2.val == 0.0)
                execerror("Compiler Error 5: Division by zero",(char *) 0);
        else
                d1.val /= d2.val;
        push(d1);
}

mod()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        if(d2.val == 0.0)
                execerror("Compiler Error 5: Division by zero",(char *) 0);
        else
                d1.val = d1.val % d2.val;
        push(d1);
}

rem()
{
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        if(d2.val == 0.0)
                execerror("Compiler Error 5: Division by zero",(char *) 0);
        else {
                if (d2.val < 0)
                d1.val = (int) abs(d1.val) * (-1);
        d1.val = d1.val % d2.val;
        }
        push(d1);
}

power()
{
```

```
        Datum d1, d2;
        d2 = pop();
        d1 = pop();
        d1.val = (int) pow(d1.val, d2.val);
        push(d1);
}

absl()
{
        Datum d;
        d = pop();
        d.val = (int) abs(d.val);
        push(d);
}

not()
{
        Datum d;
        d = pop();
        d.val = (d.val == 0);
        push(d);
}

eval()    /* evaluate variable on stack */
{
        Datum d;
        d = pop();
        if(d.symb->type != VARIABLE)
                execerror("Compiler Error 6: Attempt to evaluate non-variable", d.symb->name);
        else
                d.val = d.symb->val;
        push(d);
}

********    crenet.c    ********************************************************************************************

#include <stdio.h>
#include <ctype.h>
#include "vhdl.h"
#include "x.tab.h"

#define DECL1
#include "net.h"

/*****************************************************************************************************************/
/*                                          crenet                                                               */
/*                                                                                                               */
/*      AUTHOR:     Bing Liu                                                                                     */
/*              Dept. of Electrical and Computer Engineering                                                     */
/*              University of Manitoba                                                                           */
/*              Winnipeg, Manitoba, Canada                                                                       */
/*              R3T 2N2                                                                                          */
```

```
/*      DATE:  Sept. 7, 1989                                                    */
/*                                                                              */
/*      REVERSION NUMBER:2                                                      */
/*      REVERSION DATE:    Sept. 12, 1989                                       */
/*                                                                              */
/*      DESCRIPTION: This module converts the data structure in vhdl.h into the data   */
/*              structure in net.h, writes the unflattened Xilinx Netlist Format (XNF) files using */
/*               the data structure in net.h.                                   */
/*                                                                              */
/*              This program uses vhdl.h, x.tab.h, and modules form Xilinx,     */
/*              such as netread.c, netwrite.c, netutil.c, netparse.c, net.h     */
/*                                                                              */
/*      ARGUMENTS:  None.                                                       */
/*                                                                              */
/*      EXTERNAL VARIABLES:                                                     */
/*              buff    -- buffer for string operation                          */
/*              msg     -- buffer for any message                               */
/*              netfn   -- name of the XNF file                                 */
/*              netfp   -- file being read from or written to                   */
/*              comnum -- componts number, used for constract SYM name          */
/*              body    -- body name of a architecture, used for SYM name       */
/*                                                                              */
/********************************************************************************/

char *netfn;
char *netfp;
extern char msg[255];
extern char buff[125];
extern vhdlerr;
int comnum;
char *body;
char *creatime;

crenet()
{
        Entity *ep;
        Sig *sip;
        Cinst *cip;
        Fport *fp;
        Pport *pp;
        SIG *s;
        char when[32];

        for (ep = firstentity; ep != NULL; ep = ep->next) {
                firstprog = NULL;
                lastprog = NULL;
                firstuser = NULL;
                lastuser = NULL;
                firstsym = NULL;
                firstsig = NULL;
                delsyms = NULL;
                delsigs = NULL;
                errcnt = 0;
```

```
                comnum = 0;
                body = ep->body;

                for (sip = ep->sigs; sip != NULL; sip = sip->next)
                        rdsigl(sip, ep->types);
                for (fp = ep->ports; fp != NULL; fp = fp->next)
                        rdfport(fp, ep->clause);
                for (pp = ep->pins; pp != NULL; pp = pp->next)
                        rdpport(pp);
                for (cip = ep->cinsts; cip != NULL; cip = cip->next)
                        rdcomp(cip, ep->cdecls, ep->types);
                if (strcmp(ep->name, firstentity->name) != 0)
                        sprintf(buff, "%s.XNF", ep->name);
                else
                        sprintf(buff, "%s.HNF", ep->name);
                netfn = stralloc(buff);
                timeofday(msg);
                creatime = stralloc(msg);
                if (vhdlerr == 0)
                        writenet();
        }
        return;
}
/*--end of crenet.c--*/

rdsigl(sip, tp)
        Sig *sip;
        Type *tp;
{
        SIG *s;
        int cnt;
        char *signame;

        switch (sip->type) {
                case BIT:
                        signame = stralloc(sip->name);
                        s = mksig(signame);
                        break;
                case BIT_N:
                        signame = stralloc(sip->name);
                        s = mksig(signame);
                        s->NONC = 1;
                        break;
                case BIT_X:
                        signame = stralloc(sip->name);
                        s = mksig(signame);
                        s->UEXP = 1;
                        break;
                case BIT_C:
                        signame = stralloc(sip->name);
                        s = mksig(signame);
                        s->CRIT = 1;
```

```
                        break;
                case BIT_L:
                        signame = stralloc(sip->name);
                        s = mksig(signame);
                        s->LONG = 1;
                        break;
                case TYPE_NAME:
                        while (tp != NULL && strcmp(tp->name, sip->type_name) != 0)
                                tp = tp->next;
                        if (tp == NULL) {
                                execerror ("Compiler Error 7: Undefined type", sip->type_name);
                                break;
                        }
                        for (cnt = tp->int1; cnt <= tp->int2; cnt++) {
                                sprintf(buff, "%s%d", sip->name, cnt);
                                signame = stralloc(buff);
                                s = mksig(signame);
                        }
                        break;
                default:
                        execerror("Compiler Error 8: Illegale signal", sip->name);
        }
        return;
}
/*--end of rdsigl--*/

rdfport(fp, clp)
        Fport *fp;
        Clause *clp;
{
        Type *tp;
        SIG *s;
        int cnt;
        char *signame;

        switch (fp->type) {
                case BIT:
                        signame = stralloc(fp->name);
                        s = mksig(signame);
                        break;
                case TYPE_NAME:
                        while (clp != NULL) {
                                tp = (clp->units)->types;
                                while (tp != NULL && strcmp(tp->name, fp->type_name) != 0)
                                        tp = tp->next;
                                if (tp == NULL)
                                        clp = clp->next;
                                else
                                        clp = NULL;
                        }
                        if (tp == NULL) {
                                execerror ("Compiler Error 9: Undefined type", fp->type_name);
                                break;
```

```
                        }
                        for (cnt = tp->int1; cnt <= tp->int2; cnt++) {
                                sprintf(buff, "%s%d\0", fp->name, cnt);
                                signame = stralloc(buff);
                                s = mksig(signame);
                        }
                        break;
                default:
                        execerror("Compiler Error 10: Illegal formal port", fp->name);
        }
        return;
}
/*--end of rdfport--*/

rdpport(pp)
        Pport *pp;
{
        SIG *s;

        if (pp->index != -1)
                sprintf(buff, "%s%d\0", pp->name, pp->index);
        else
                sprintf(buff, "%s\0", pp->name);
        s = fsigname(buff);
        if (s == NULL) {
                execerror("Compiler Error 11: Undefined formal port", pp->name);
                return;
        }
        s->UEXT = 1;
        s->lcaloc = stralloc(pp->pad);
        s->blkname = stralloc(buff);
        pp = pp->next;
        return;
}
/*--end of rdpport--*/

rdcomp(cip, cdp, tp)
        Cinst *cip;
        Cdecl *cdp;
        Type *tp;
{
        Fport *fp;
        Aport *ap;
        SYM *sym;

        while (cdp != NULL && strcmp(cip->name, cdp->name) != 0)
                cdp = cdp->next;

        if (cdp == NULL) {
                execerror ("Compiler Error 12: Undeclarated component", cip->name);
                return;
        }
        comnum++;
```

```
        sym = mksym();
        if (cip->label == NULL) {
                sprintf(buff, "%s%d", body, comnum);
                cip->label = buff;
        }
        sym->name = stralloc(cip->label);
        sym->symtype = symtype(cip->name);
        sym->typename = stralloc(cip->name);

        ap = cip->ports;
        fp = cdp->ports;
        while (fp != NULL && ap != NULL) {
                rdport(ap, fp, tp, sym);
                fp = fp->next;
                ap = ap->next;
        }
        return;
}
/*----end of rdcomp----*/

rdport(ap, fp, tp, sym)
        Aport *ap;
        Fport *fp;
        Type *tp;
        SYM *sym;
{
        SIG *s;
        PIN *p;
        char dir;
        int cnt;

        switch (fp->mode) {
                case IN:
                        dir = 'I';
                        break;
                case OUT:
                        dir = 'O';
                        break;
                default:
                        execerror ("Compiler Error 13: Illegal formal port direction", fp->name);
        }
        if (fp->type == BIT) {
                if (ap->index != -1)
                        sprintf(buff, "%s%d", ap->name, ap->index);
                else
                        sprintf(buff, "%s", ap->name);
                s = fsigname(buff);
                if (s == NULL) {
                        execerror("Compiler Error 14: Undefined actual port", ap->name);
                        return;
                }
                p = mkpin (s, sym);
```

```
                p->name = stralloc(fp->name);
                p->dir = dir;
                if (dir == A_INPIN)
                        s->NOLOAD = 0;
                else if (dir == A_OUTPIN) {
                        if (s->NOSRC)
                                s->NOSRC = 0;
                        else
                                s->MULTSRC = 1;
                }
                return;
        }
        if (fp->type == TYPE_NAME) {
                while (tp != NULL && strcmp(tp->name, fp->type_name) != 0)
                        tp = tp->next;
                if (tp == NULL) {
                        execerror ("Compiler Error 15: Undefined type", fp->type);
                        return;
                }
                for (cnt = tp->int1; cnt <= tp->int2; cnt++) {
                        sprintf(buff, "%s%d\0", ap->name, cnt);
                        s = fsigname(buff);
                        if (s == NULL) {
                                execerror("Compiler Error 16: Undefined actual port", buff);
                                return;
                        }
                        sprintf(buff, "%s%d\0", fp->name, cnt);
                        p = mkpin (s, sym);
                        p->name = stralloc(buff);
                        p->dir = dir;
                        if (dir == A_INPIN)
                                s->NOLOAD = 0;
                        else if (dir == A_OUTPIN) {
                                if (s->NOSRC)
                                        s->NOSRC = 0;
                                else
                                        s->MULTSRC = 1;
                        }
                }
                return;
        }
        execerror ("Compiler Error 17: Undefined formal prot", fp->name);
        return;
}
/*--end of rdport--*/

timeofday(buff)
        char *buff;
{
        long   clock;

        time(&clock);
        strcpy(buff, ctime(&clock));
```

```
        return;
}
/*---end of timeofday---*/

********    makefile    ********************************************************************************

YFLAGS = -d
OBJS = parser.o scanner.o vhdl.o code.o init.o symbol.o crenet.o netwrite.o netutil.o netparse.o
netread.o flatten.o

vhdl:   $(OBJS)
        cc  $(OBJS) -lm -ll -o vhdl2xnf

parser.o scanner.o vhdl.o code.o init.o symbol.o crenet.o:  vhdl.h

scanner.o vhdl.o code.o init.o symbol.o crenet.o:  x.tab.h

x.tab.h:  y.tab.h
        -cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
```