COMPILING A SYNCHRONOUS PROGRAMMING LANGUAGE INTO FIELD PROGRAMMABLE GATE ARRAYS

CENTRE FOR NEWFOUNDLAND STUDIES

TOTAL OF 10 PAGES ONLY
MAY BE XEROXED

(Without Author's Permission)

YING SHEN

Compiling a Synchronous Programming Language into Field Programmable Gate Arrays

By

©Ying Shen, B.Eng., M.Eng.

A thesis submitted to the School of Graduate Studies
in partial fulfilment of the requirements for
the degree of Master of Engineering

Faculty of Engineering and Applied Science
Memorial University of Newfoundland

May 1999

St. John's Newfoundland Canada

To my parents and my sisters

Abstract

This thesis shows how to compile a program expressed by a novel hardware description language, the State Machine Algol-Like Language (SMALL), into Field-Programmable Gate Arrays (FPGAs). A "netlist generator" for the SMALL language is created to transform a parallel Algorithmic State Machine (ASM) chart into a structural VHDL description. The one-hot encoding technique is used for the transformations. The structural VHDL description for the netlist is simulated and synthesised by Synopsys VSS (VHDL System Simulator) and Synopsys FPGA Compiler, respectively. The netlist is very simple and the components of the netlist consist of only D-type flip-flops and basic gates. The Design Manager of the Xilinx Alliance Series version 1.4 is used to produce configuration data for Xilinx FPGA chips. The Xilinx XC4000 family is employed as the target FPGA device.

The simulation results for several SMALL programs indicate that the netlist generator performs the specified requirements for all the statements and all the operators in the SMALL language.

Using the netlist generator and existing place-and-route tools makes the implementation of SMALL programs on FPGAs easy. This research offers a significant advance on the original SMALL implementation. Due to its simplicity and simple semantics, it is believed that the SMALL language will be widely used in many areas in the future.

Acknowledgements

I would like to take this opportunity to appreciate the help and assistance from many people in my study for the degree of Master of Engineering. Particularly, I would like to thank my supervisor, Dr. Theodore S. Norvell, for his guidance and financial support through the program. His broad knowledge in Electrical and Computer Engineering has greatly helped in every stage of my research program. The generous financial support from Memorial University is also greatly appreciated.

I am very grateful for the assistance provided by my fellow graduate students, namely Mohsin Riaz, Yaser El-Sayed and Zhikai Ding in using some software packages. Thanks are also due Dr. James J. Sharp, Dr. Mahmoud R. Haddara and Mrs. Moya Crocker for their help during my graduate studies at Memorial.

Special appreciation goes to my parents who have stayed in St. John's for a whole year to look after my son Kevin during my graduate studies. Their dedication is very helpful to my family. Finally, I would like to thank my husband Fanyu for his help and encouragement.

Contents

List of Figures

List of Abbreviations

ASM Chart	Algorithmic State Machine Chart.
ASIC	Application Specific Integrated Circuit.
CAD	Computer-Aided Design.
CLB	Configurable Logic Block.
CMC	Canadian Microelectronics Corporation.
DPGA	Dynamically Programmable Gate Array.
FPGA	Field Programmable Gate Array.
IOB	Input/Output Block.
SMALL	State Machine Algol-Like Language.
Synopsys FC	Synopsys FPGA Compiler.
Synopsys VSS	Synopsys VHDL System Simulator.
VHDL	VHSIC Hardware Description Language.
VHSIC	Very High Speed Integrated Circuit.

Chapter 1

Introduction

Several hundreds of hardware description languages have been presented in the past decades (Wodtko, 1987). Only some of these languages are designed to describe synchronous behavior. SMALL (State Machine Algol-Like Language) is a novel hardware description language that has been developed especially for both teaching and synchronous state-machine design purposes by Norvell (1997) at Memorial University of Newfoundland. It consists of a simple set of symbols and notations that replace schematic diagrams of digital circuits. SMALL, as a hardware-oriented programming language, could be applied into many areas as follows (Norvell, 1996 and 1997).

- It is oriented toward teaching sequential circuit design.
- It can be embedded in a larger specification language.
- It can aid design communication, hardware simulation and the transformation of designs from a higher level (such as state machine) to a lower level (such as gates).
- It can be used in design derivation from specification.
- It can be used in rapid development.

- The program written in SMALL can be directly compiled into hardware without human intervention.

Due to its simplicity in sequential circuit design, in formal semantics. and in time model, the SMALL language will be more widely used in the future and its further development becomes more important. The purpose of this thesis is to compile programs expressed in the SMALL language into Field-Programmable Gate Arrays (FPGAs), a technology that allows a design expressed in the SMALL language to be implemented without a conventional fabrication plant. The entire compilation process can be divided into the four stages shown in Figure 1.1. First, the text of a SMALL program is converted to a parallel Algorithmic State Machine (ASM) chart. At the second stage, the netlist for the SMALL program is created. The next stage is to synthesise from the structural VHDL (The VHSIC Hardware Description Language) netlist. The final stage implements the netlist using a specific Xilinx FPGA chip. It should be noted that the functional simulation is carried out in order to verify that the Netlist Generator performs the specified requirements before the third stage.

The Netlist Generator is written in the functional language Gofer (Jones, 1991, 1993, and 1994; Cunningham, 1995; and Wadler, 1995). The input to the Netlist Generator is a parallel ASM (Algorithmic State Machine) chart that is generated by the compiler's front-end. Its output is a structural VHDL source file describing a netlist composed of simple gates and flip-flops. Synopsys VSS (VHDL System Simulator) is used for simulation that verifies the netlist circuit created by the Netlist Generator matches the requirements. Synopsys FPGA Compiler (FC) is adopted for synthesising the gate-level

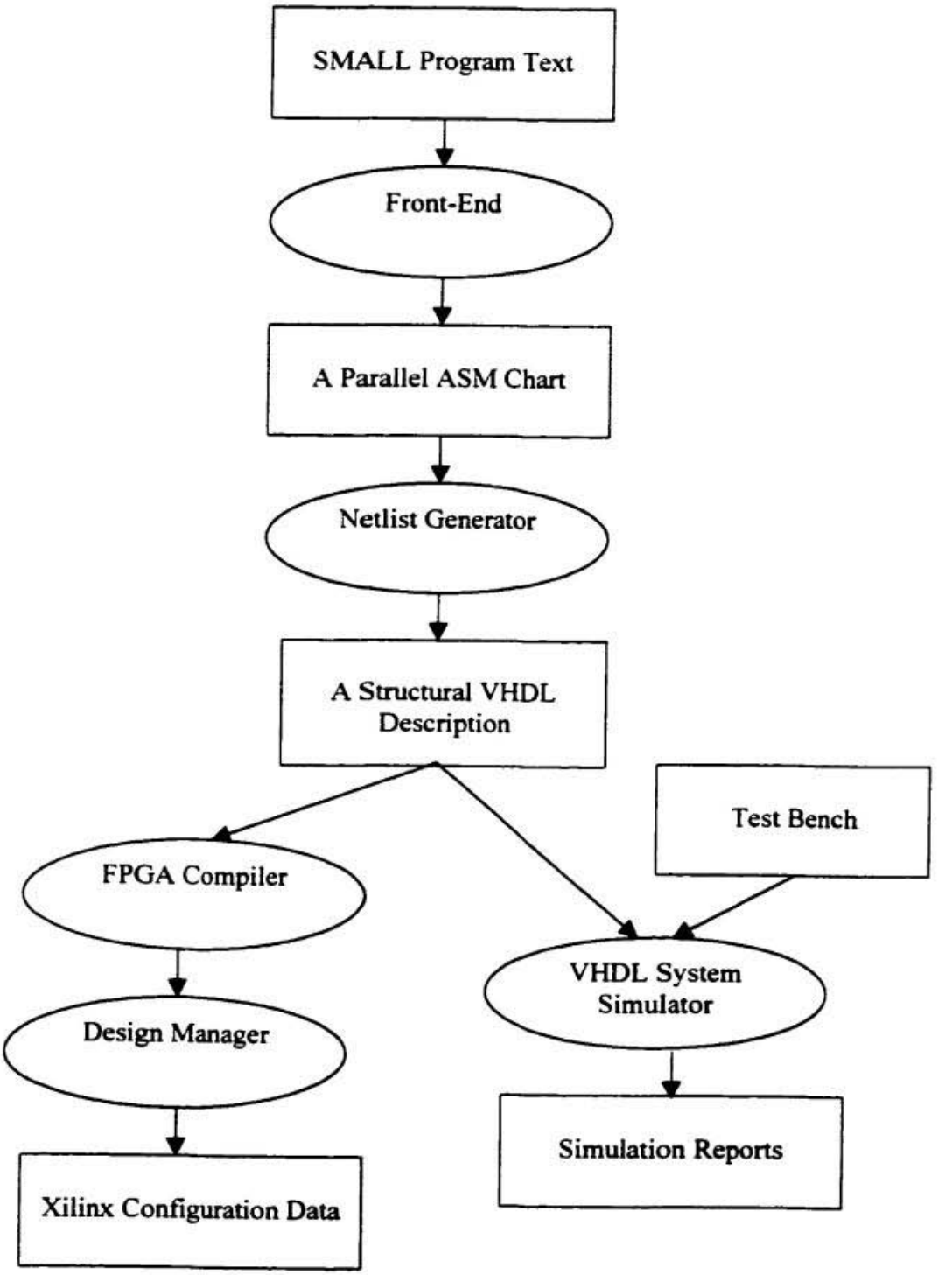


Figure 1.1: The Framework of This Thesis

VHDL description and automatic hardware generation. Design Manager of the Xilinx Alliance Series version 1.4 is employed to produce the configuration data for a specific Xilinx FPGA chip. The Xilinx XC4000 family FPGA logic is chosen as the target technology.

The rest of this thesis is organised as follows:

Chapter 2 provides an introduction to the SMALL programming language and discusses related approaches.

Chapter 3 describes in detail the design and implementation of the Netlist Generator for the SMALL language, including algorithm descriptions and translation circuits.

Chapter 4 gives the hardware implementation of the netlist for the SMALL language using a Field Programmable Gate Array (FPGA). Several examples are simulated to verify that the Netlist Generator produced in Chapter 3 meets the specified requirements.

Finally, Chapter 5 contains some concluding remarks.

Chapter 2

Literature Review

This chapter is intended to provide an introduction to the SMALL programming language and to review related work. An understanding of the original implementation of the SMALL language will be helpful in order to comprehend the Netlist Generator. An example of a SMALL program used for synthesis and simulation in later chapters is also presented.

2.1 SMALL: A Programming Language for State Machine Design

One essential contribution of this thesis is the creation of a Netlist Generator for the SMALL programming language. In order to understand the design and implementation of the Netlist Generator, this section briefly describes the SMALL programming language. The following is a review based on the SMALL documentation (Norvell, 1996).

2.1.1 The SMALL Language

SMALL stands for State Machine Algol-Like Language. Its relationship to ASM

charts is almost the same as the relationship between high-level software languages (like Algol) and flow charts. In the SMALL language, signals are used to carry information instantaneously and registers are used for storing information through time. The value of a signal is determined in the current clock period; however, the value of a register is the latest value assigned to it in a previous clock period. The following two forms are used for the declarations of the signals and the registers, respectively.

`sig` *signal_name T C*

`reg` *register_name T C*

In the above forms, `sig` and `reg` are keywords; *signal_name* and *register_name* are identifiers. *T* is a type that is either `bool` or `array` *n* `of` *T* at present and *C* is the scope of the declarations. In order to communicate with external world, the keyword `global` could be added to the beginning of the declarations.

The statements in the SMALL language include:

- *s ! expression* which places data on a signal *s*; this is called as "assert statement".
- *r <- expression* which places data in a register *r*; this is called as "assignment statement".
- `tick` which expresses the end of one clock and the start of the next.
- *statement_0 statement_1 ... statement_n* which represents a sequence of statements executed consecutively from left to right.
- `if` *expression* `then` *statement_0* `else` *statement_1* `fi` which is an alternative composition and means that when the *expression* is *true*, the *statement statement_0* is executed; otherwise, the *statement statement_1* is executed.

- `while` ***expression*** `do` ***statement*** `od` whose meaning is that the ***statement*** is executed if the ***expression*** is true; the ***expression*** is reevaluated in the clock period following the completion of the ***statement***.
- `repeat` ***statement*** `until` ***expression*** whose meaning is that the ***statement*** is executed, and if the ***expression*** is true in the last clock period of the ***statement***, the `repeat` statement is complete; otherwise, the loop is restarted in the next clock period.
- `par` ***statement_0*** || ***statement_1*** || ... || ***statement_n*** `rap` which is a parallel composition and which will end as soon as all the processes have ended.
- `skip` which simply ends immediately.
- `stop` which stops the interpreter.

In the SMALL language, the concept of duration is important. It is defined as the difference between the initial and final clock period by Norvell (1997). According to the definition, the duration of assert and assignment statements is 0; `tick` and `skip` have duration 1 and 0, respectively. The duration of a parallel statement is the maximum of the duration of the processes and that of a sequence of statements is the sum of the duration of each statement. In the sequence of statements, the final clock period of ***statement_i*** is also the initial clock period of ***statement_i+1***. For the alternative composition, whether ***statement_0*** or ***statement_1*** is chosen, the start time of the chosen statement is the same as the start time of the whole alternative composition. The finish time of the alternative composition depends on the finish time of ***statement_0*** when the value of the ***expression*** is `true` at the start time. In the case of a `false` value for the ***expression***, the finish time

of the alternative composition is the same as the finish time of *statement_1*. For the `while` statement, when the *expression* is true during a clock period, the *statement* is executed in the same clock period and the loop is restarted in the first clock period after the last clock period used by the *statement*; when the *expression* is `false`, the `while` statement ends immediately. Therefore, the duration of the `while` statement is related to its loops. Similar to the `while` statement, the duration of the `repeat` statement is also related to its loops. Both looping forms imply a delay immediately before the loop is restarted.

A number of the operators constitute expressions. The expressions are divided into seven categories, including identifiers, constants, unary expressions, binary expressions, subscript expressions, subarray expressions and array expressions. In the next section, an example will be given to illustrate the above statements and some operators in the SMALL language.

2.1.2 The Formal Semantics of SMALL

A specification is a predicate on behaviours. In SMALL, the starting time, the final time, and values of signals and registers over all time are used to describe the behaviours. The following variables representing times and time varying functions describe the behaviour of a command with a `global signal` $s : T_s$ and a `global register` $r : T_r$.

t : *xnat*	The starting time.
t' : *xnat*	The final time.
s : *wire*.T_s	The value of s during each clock period.

$\bar{s} : wire.bool$	Whether s is asserted during each clock period.
$r : wire.T_r$	The value of r during each clock period.
$\bar{r} : wire.bool$	Whether r receives a new value after each clock period.

In the above definition, *xnat* is a set of times and *wire.T* represents a function. *xnat* and *wire.T* are defined as follows:

$$xnat = nat \cup \infty$$
$$wire.T = xnat \rightarrow T$$

The formal semantics is determined by semantic equations which are presented in (Norvell, 1997). Their definitions include that primitive commands are specifications and that composition operators equal operators on specifications.

The following example specification obtained from (Norvell, 1997) can explain some simple semantics.

$$t' = t + 1 \ \wedge \ \bar{s}.t \wedge s.t = r.t \wedge \bar{r}.t \wedge r.(t+1) = 10$$

A command that implements this specification will have the following meanings:

- The duration of the command is 1.
- During its first clock period, the signal s has the same value as the register r.
- 10 will be assigned to r between its two clock periods.

2.1.3 An Example Program Written in SMALL

Figure 2.1 shows an example written in SMALL to illustrate many expressions and statements in the language. This example is obtained from (Norvell, 1996) and will be synthesised and simulated in Chapter 4.

```
par                                                                    //0
  global signal go : bool                                              //1
  global signal done : bool                                            //2
  global signal multiplier : array 4 of bool                           //3
  global signal multiplicand : array 4 of bool                         //4
  global reg p : array 8 of bool                                       //5
  while true do                                                        //6
    reg a: array 4 of bool                                             //7
    reg b: array 4 of bool                                             //8
    reg count: array 2 of bool                                         //9
    repeat // Initialize while waiting to start.                       //10
       p[4@4] <- 4 of 0                                                //11
       count <- 0 as 2 bits                                            //12
       a <- multiplicand                                               //13
       b <- multiplier                                                 //14
    until go                                                           //15
    tick                                                               //16
    repeat  // Invariant: the product thus far (multiplicand * the first  //17
          //count bits of multiplier) is the last 4+count bits of p.   //18
         // Form product of a and b[0].                                //19
       signal pp : array 4 of bool                                     //20
       if b[0] then pp !a else pp ! 4 of 0 fi                          //21
       // Add this to the most significant 4 bits of p.                //22
       signal sum : array 5 of bool                                    //23
       sum ! p[4@4] uplus pp // Replace the top 4 bits of the          //24
                    // product with the 5 bit sum. To make room, shift. //25
       p[5@3] <-sum                                                    //26
       p[3@0] <-p[3@1]                                                 //27
       b <- b[3@1] ++ [0]                                              //28
       count <- count plus (1 as 2 bits)                               //29
    until count = 3 as 2 bits                                          //30
    tick                                                               //31
    repeat done ! 1 until not go                                       //32
  od                                                                   //33
||                                                                     //34
  global signal go : bool                                              //35
  global signal done : bool                                            //36
  global signal multiplier : array 4 of bool                           //37
  global signal multiplicand : array 4 of bool                         //38
  go!1  multiplier![1,1,0,0]  multiplicand![0,1,0,0]                   //39
  tick                                                                 //40
  repeat skip until done                                               //41
rap                                                                    //42
```

Figure 2.1: A Sequential Multiplier

The example in Figure 2.1 is used to multiply two 4-bit numbers to produce an 8-bit result. It produces one partial product in each clock period. By shifting the partial product and adding the *multiplicand* if a bit in the *multiplier* is `1` in each clock period, the final product can be obtained after five clock periods. Many signal and register declarations and statements appear. For example, Line 0 to Line 42 is a parallel composition. Assert statements and assignment statements are used to send data to the `signals` or the `registers`. For example, the statements such as `p[4@4]<-4 of 0`, *multiplier* `! [1,1,0,0]`, and *multiplicand* `! [0,1,0,0]` initialise the partial product, the *multiplier*, and the *multiplicand*, respectively. There are many expressions and operators in the example. For instance, the expression, `p[4@4]` in Line 11, presents the segment of length 4 beginning at index 4 and the '`4 of 0`' expresses an array of length 4 that is the same as `[0,0,0,0]`. Because index 0 is used for the least significant bit, the expression '`1 as 2 bits`' in Line 29 means the 2 bits unsigned magnitude representation of `1` and has the value `[1,0]`. The operators such as '`uplus`' in Line 24, '`++`' in Line 28 and '`plus`' in Line 29 distribute to array of boolean level. From the above example, it is shown that SMALL has many distinguishing features that other hardware description languages lack. Compared with other hardware description languages, SMALL's simplicity will help it to be used in many practical areas.

2.2 The Original Implementation of the SMALL Language

The following paragraph from (Norvell, 1996) describes the original implementation of the SMALL language.

The current implementation is in Gofer version 2.30 and uses the prelude cc.prelude. Gofer is a functional language designed and implemented by Mark P. Jones and is very similar to Haskell. Extensive use is made of monads and an imperative programming style.

The implementation of the SMALL language is shown in Figure 2.2. The original implementation consists of all components other than the Netlist Generator. In Figure 2.2, the front-end is composed of Lexer, Parser, and Analyzer. The function of each component shown in Figure 2.2 is described in the following.

The lexer takes a source program written in the SMALL language as its input and outputs a token sequence. The parser creates an abstract syntax tree and finds all syntax errors. The analyzer creates an intermediate representation of the source program from the abstract syntax tree. It has several functions. The first is to check the source program for compile-time errors such as type errors and misuses of identifiers. The second is to produce a requirement table that is a list of all signals and registers, including the extra signals introduced for each process in order to coordinate termination of the parallel processes. These signals and registers are used to run the program. The last function is the generation of an ASM chart representation of the program. The ASM chart expresses the control-flow of the source program. It is a collection of nodes connected by directed edges. Nodes are divided into five varieties: state nodes, dummy nodes, assert nodes, assignment nodes, and condition nodes and are labelled by expressions. The next step is one of two alternatives. Either the executer simulates execution of the ASM chart with test inputs, or the Netlist Generator translates the ASM chart to a gate-level

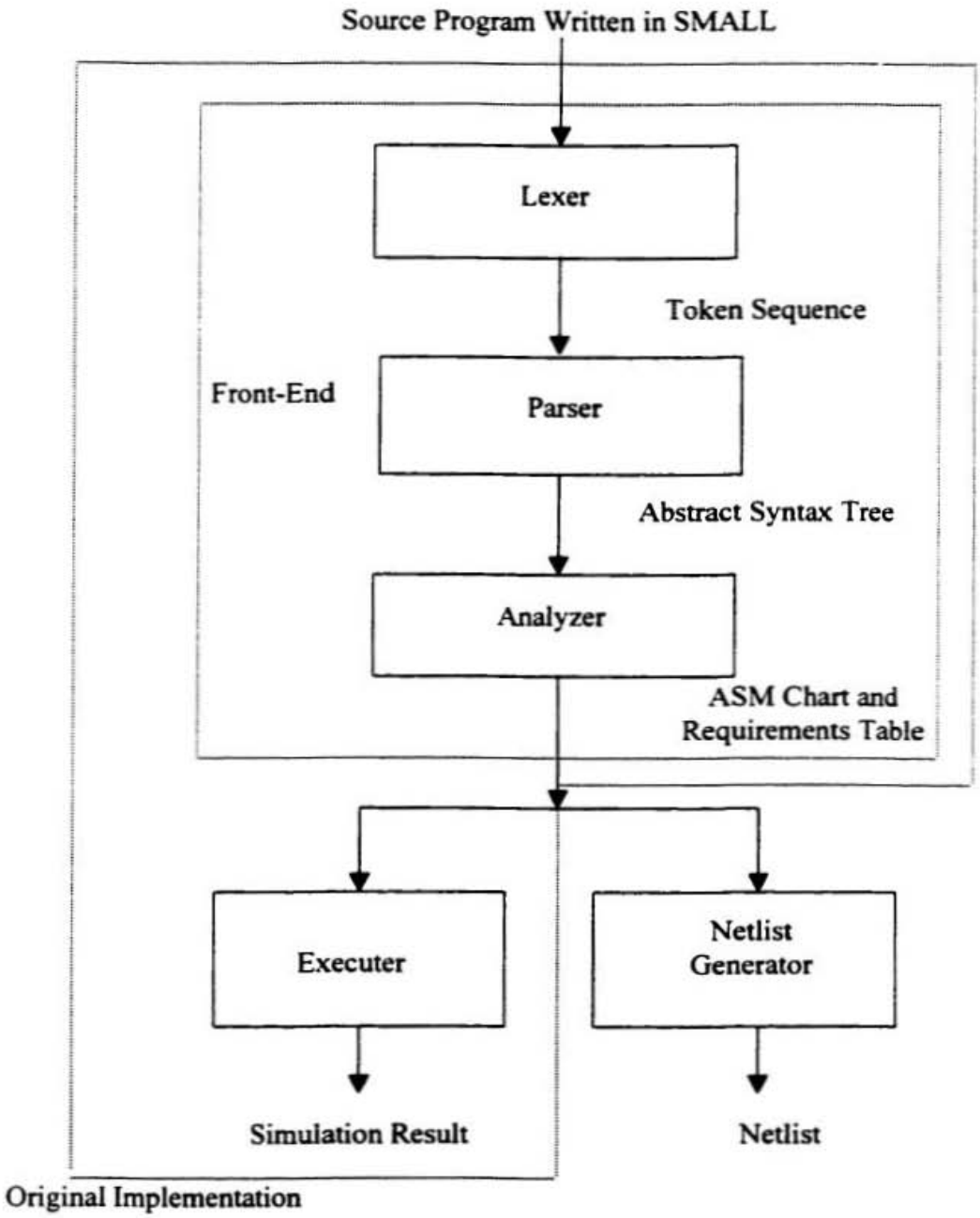


Figure 2.2: The Implementation of the SMALL Language

implementation in terms of gates and flip-flops.

2.3 The Main Contributions of This Thesis

One essential contribution of this thesis is the creation of a Netlist Generator for the SMALL programming Language. The ASM chart and requirements table is taken as the input to the Netlist Generator. The output will be a netlist composed of gates and flip-flops and expressed by a VHDL entity declaration and a structural architecture declaration. The Synopsys VSS is used for the functional simulation of the structural VHDL description to verify that the Netlist Generator performs the specified requirements. The detailed design and implementation of the Netlist Generator for the SMALL language will be presented in the following chapter.

Another contribution of this thesis is to implement such netlists with FPGAs. The Synopsys FPGA Compiler is adopted for the gate-level synthesis and logic optimisation. The Design Manager of the Xilinx Alliance Series is used to create configuration data for Xilinx FPGA chips. Some information about pre-placement and routing and post-layout for implementing a sequential multiplier shown in Figure 2.2 on a Xilinx XC4028EX-3-PG299 FPGA will be provided in Section 4.4.

2.4 Related Work

There has been some previous work on compiling a program written in a selected language into hardware, such as FPGAs and silicon. The difference between synchronous

and asynchronous methods for controlling the timing in circuits should be noted. In synchronous communication, a message is sent and received simultaneously, without communication delay. An asynchronous communication has a time delay when a message is sent and received. Usually, local handshaking signals are used in the asynchronous method; a global clock is used in the synchronous method. The following is a review of the related work.

An early work is done by Martin (1986). He developed a compiler from a concurrent programming language into delay-insensitive circuits that are obtained by a series of semantics-preserving transformations. The compilation is divided into the four steps: process decomposition, handshaking expansion, production-rule expansion, and operator reduction. The circuits are correct by construction.

The work of Weber *et al.* (1992a and 1992b) on Joy, a simple yet complete programming language for circuits, is similar to that of Martin. They have modelled delay-insensitive circuits as directed graphs and proved the correctness of their compilation algorithm. The resulting circuits are composed of some primary components; each of the components is described by a wiring diagram and a finite-state automaton.

Guo and Luk (1995) have provided a methodology for compiling a design expressed in the Ruby language into FPGAs. The Ruby language is intended for the specification and synthesis of the circuits described by relational abstractions of their behaviour (Rossen, 1990) and allows simple relations to be composed into more complicated ones by using higher-order functions. With mathematically based compilation tools, the

correct hardware can be produced from high-level descriptions very rapidly. The refinement module and the floorplanning module of their compilation system are also discussed in detail.

Another close work presented by Berry and Gonthier (1992) and Jagadeesan *et al.* (1995) has involved the hardware implementation of the ESTEREL synchronous programming language which is based on the "Synchrony Hypothesis" and has well-defined mathematical semantics. Berry and Gonthier have shown an algorithm to translate ESTEREL programs into deterministic automata. Berry (1995) has provided the direct hardware implementation that transforms ESTEREL programs into boolean circuits by using the current ESTEREL hardware or software compiling and verification technology.

LUSTRE is a synchronous declarative language devoted to real-time systems and is suitable for data path description. Thuau and Pilaud (1990) have divided the compilation of a LUSTRE program into two steps: expanding all the nodes to get a single system of equations and producing a finite automaton. The verification tool LESAR is developed for proving the correctness of a circuit.

The VHDL (Lipsett *et al.*, 1989; Patterson, 1994) and Verilog (Thomas and Moorby, 1991) languages provide the digital system designer with accurate description and modelling of a system at a wide range of levels, from the highest behavioural level of abstraction to the lowest structural gate level. Many people are interested in circuit synthesis with VHDL or Verilog. Greaves (1995) has worked on defining formal

semantics for the Verilog language that is used for simulation and compilation into hardware. Gschwind and Salapura (1995) have proposed a VHDL design methodology for FPGAs to integrate the modelling, verification and implementation processes.

Hehner *et al.* (1998) presented two new ways to implement ordinary programs with logic gates. They have adopted local delays and given a framework for the proof of correct circuits in terms of a formal semantics for programs and circuits.

The most closely related work is conducted by Page and his group (Page and Luk, 1991; He *et al.*, 1993; and Page, 1996). They have compiled programs written in a subset of occam into FPGAs. A "normal form" method (He *et al.*, 1993) is used for processing into a netlist and for proving the correctness of their compilation. Page and Luk (1991) have described a prototype compiler written in the functional language SML that transforms Handel, an occam-like language, to a netlist. Page (1996) has further implemented his hardware-software codesign by compiling the Handel language into a netlist, which is suitable for Xilinx Dynamically Programmable Gate Arrays (DPGAs) technology.

This research is also related to and influenced by the above work. We have focused on compiling programs in the SMALL language into a netlist, which can be loaded into a Xilinx FPGA. Compared with the work of Page *et al.*, the main difference is that, Page *et al.* considered the implementation of control hardware for channel communication and used a flip-flop in their assignment control hardware; this research has not. This is

because assignment statements and assert statements in SMALL have a duration of 0 rather than 1.

The one-hot encoding is used as the encoding technique in this thesis. It uses one flip-flop per state node. Gschwind and Salapura (1995) have proved that one-hot encoding is both the fastest encoding and one of the smallest representations as it makes the best use of the flip-flops on an FPGA.

Chapter 3

Design and Implementation of the Netlist Generator for SMALL

The purpose of this thesis is to compile programs expressed in the SMALL language into FPGAs. To do so, a Netlist Generator for the SMALL language has been created to transform parallel ASM charts to structural VHDL files. The ASM chart is obtained after the source SMALL program is processed by the front-end stage shown in Figure 2.2. The VHDL file describes a netlist that is a collection of devices and wires implementing the program. The devices in our netlist consist of simple gates and D-type flip-flops. The Netlist Generator is programmed using the functional language Gofer (Jones, 1994; Cunningham, 1995; and Wadler, 1995). Based on (Norvell, 1998), this chapter will give detailed design and implementation for the Netlist Generator.

3.1 Specification for the Netlist Generator

The input to the Netlist Generator is an ASM chart. The output will be a VHDL file composed of an entity declaration for a device and a structural architecture for the same device. This section will provide an example ASM chart and the data-flow diagram for

the Netlist Generator. An example VHDL description will be given in Section 3.9.

3.1.1 An Example ASM Chart

In order to illustrate ASM charts, Figure 3.1 shows a simple SMALL program describing a parity generator and some test inputs. The function of the program is that at each clock period, the *outBit* is the parity of those *inBits* that have been seen in the previous clock periods. A timing diagram of the program is given in Figure 3.2. It can be seen that in Figure 3.2, the output named *outBit* is `[0,0,1,1,0]` if the input named *inBit* is `[0,1,0,1,1]` in the first five clock periods.

```
global sig inBit: bool
global sig outBit: bool
par
      while true do
          repeat
               outBit ! 0
          until inBit
          tick
          repeat
               outBit ! 1
          until inBit
     od
||
     inBit ! 0       tick
     inBit ! 1       tick
     inBit ! 0       tick
     inBit ! 1       tick
     inBit ! 1       tick
rap
```

Figure 3.1: The Example SMALL Program

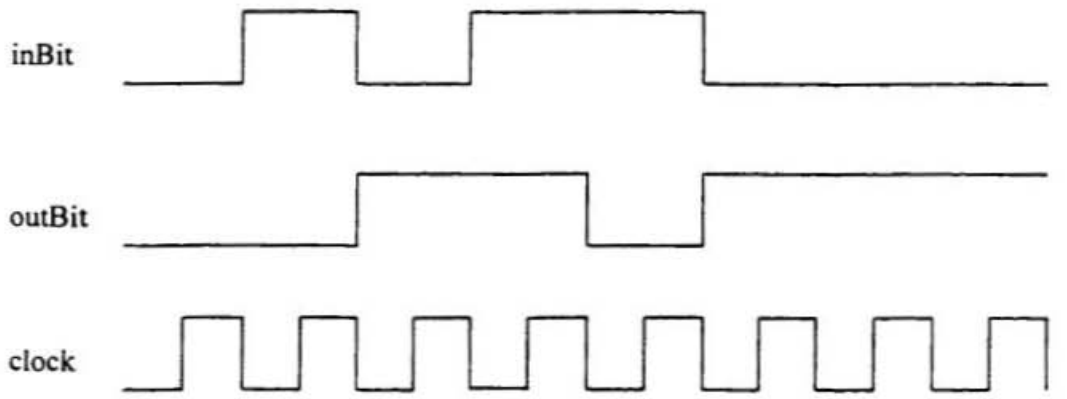


Figure 3.2: A Timing Diagram of the Program Shown in Figure 3.1

This program shown in Figure 3.1 can be converted to a parallel ASM chart form by a front-end that has been implemented by Norvell. It is noted that a requirements table is included in the parallel ASM chart in this chapter. The ASCII descriptions of the ASM chart of the SMALL program are shown in Figure 3.3, in Figure 3.4, and in Figure 3.5. A diagram of the ASM chart appears in Figure 3.6.

```
("inBit",Signal "inBit"True of type bool)
("outBit",Signal "outBit"True of type bool)
("#r*1",Signal "#r*1"True of type bool)
("#l*0",Signal "#l*0"True of type bool)
```

Figure 3.3: The ASCII Form of the Example ASM Chart: Entities

ASM charts are composed of "Entities", "Nodes", and "Edges". The collection of entities shown in Figure 3.3 is called a "requirements table" and lists all the registers and signals used for communication across space or time in the source program. The requirements table provides the following information for each entity:

- The name of the entity.
- Whether the entity is a signal or register.
- Whether the entity is local or global.
- For a signal entity, the active value of the signal.
- The entity's type, that is, bool or an array.

It is noted that signals "#r*1" and "#l*0" shown in Figure 3.3 are introduced in order to coordinate termination of the parallel processes.

Figure 3.4 represents a collection of nodes. These nodes are divided into five kinds: state nodes, dummy nodes, assert nodes, assignment nodes, and condition nodes. For each node, the node number and the node type are needed. Additional expressions are used for the assert nodes, the assignment nodes, and the condition nodes. Assert nodes and assignment nodes are labelled with two expressions representing the target signal or register and the value. The expression labelling a condition node is a boolean and is used to determine which branch is taken. Condition nodes are also labelled with the numbers of the nodes to branch to. Their detailed descriptions are given in Section 3.6, which discusses the generation of circuits for nodes.

```
The ASM Chart is
36
GRAPH Labels
(35,%CondNd (#l*0,((22,17),bool)) then 31 else 34)
(34,%StateNd)
(33,%CondNd (#r*1,((22,17),bool)) then 31 else 32)
(32,%StateNd)
(31,%DummyNd)
(30,%AssertNd(#r*1,((22,17),bool))!(True,((22,17),bool)))
(29,%AssertNd(#l*0,((22,17),bool))!(True,((22,17),bool)))
(28,%StateNd)
(27,%AssertNd(inBit,((27,17),bool))!(True,((27,25),bool)))
(26,%StateNd)
(25,%AssertNd(inBit,((26,17),bool))!(True,((26,25),bool)))
(24,%StateNd)
(23,%AssertNd(inBit,((25,17),bool))!(False,((25,25),bool)))
(22,%StateNd)
(21,%AssertNd(inBit,((24,17),bool))!(True,((24,25),bool)))
(20,%StateNd)
(19,%AssertNd(inBit,((23,17),bool))!(False,((23,25),bool)))
(18,%StateNd)
(17,%StateNd)
(16,%CondNd (inBit,((20,28),bool)) then 14 else 15)
(15,%DummyNd)
(14,%DummyNd)
(13,%AssertNd(outBit,((19,27),bool))!(True,((19,36),bool)))
(12,%DummyNd)
(11,%StateNd)
(10,%StateNd)
(9,%CondNd (inBit,((16,28),bool)) then 7 else 8)
(8,%DummyNd)
(7,%DummyNd)
(6,%AssertNd(outBit,((15,27),bool))!(False,((15,36),bool)))
(5,%DummyNd)
(4,%CondNd (True,((13,23),bool)) then 2 else 3)
(3,%DummyNd)
(2,%DummyNd)
(1,%DummyNd)
(0,%StateNd)
```

Figure 3.4: The ASCII Form of the Example ASM Chart: Nodes

```
Adjacency
(34,[30])
(32,[29])
(30,[35])
(35,[])
(29,[33])
(33,[])
(31,[])
(28,[30])
(3,[29])
(27,[28])
(26,[27])
(25,[26])
(24,[25])
(23,[24])
(22,[23])
(21,[22])
(20,[21])
(19,[20])
(0,[19,1])
(18,[1])
(14,[18])
(17,[12])
(15,[17])
(13,[16])
(16,[])
(12,[13])
(11,[12])
(7,[11])
(10,[5])
(8,[10])
(6,[9])
(9,[])
(5,[6])
(2,[5])
(1,[4])
(4,[])
```

Figure 3.5: The ASCII Form of the Example ASM chart: Edges

The edges shown in Figure 3.5 express a collection of references to successor nodes. They are very useful in connecting nodes. These edges and the nodes shown in Figure 3.4 form a node-labelled directed graph. Figure 3.6 shows the graph of the example ASM chart that represents the control-flow of the source program in Figure 3.1. In Figure 3.6, the first node is always a state node named Node 0. It represents the start of the program. In this example, a parallel statement appears at the beginning of the program so that a parallel branch is produced. Therefore, Node 0 has Node 1 and Node 19 as its successor nodes. For each `tick` statement, each `while` loop, and each `repeat` loop in the source program, the front-end will generate one state node. These state nodes represent the control state of the program. They are used to wait for the coming of the next clock period. Therefore, a timing delay is implied in the `while` loops and the `repeat` loops. For assert nodes, assignment nodes, and dummy nodes, no timing delay exists. Condition nodes are generated for `while` loops and `repeat` loops. Each condition node has two branch nodes as its successor.

Expressions come in seven forms: identifier expressions, constant expressions, unary expressions, binary expressions, subscripted array expressions, subarray expressions, and array building expressions. Each expression consists of its type and its location (line number and column number) in the source program. The detailed discussion about expressions will be given in the Section 3.7, which describes generating expression circuits.

It is believed that the clear understanding of the ASM chart is very important for understanding the netlist generation.

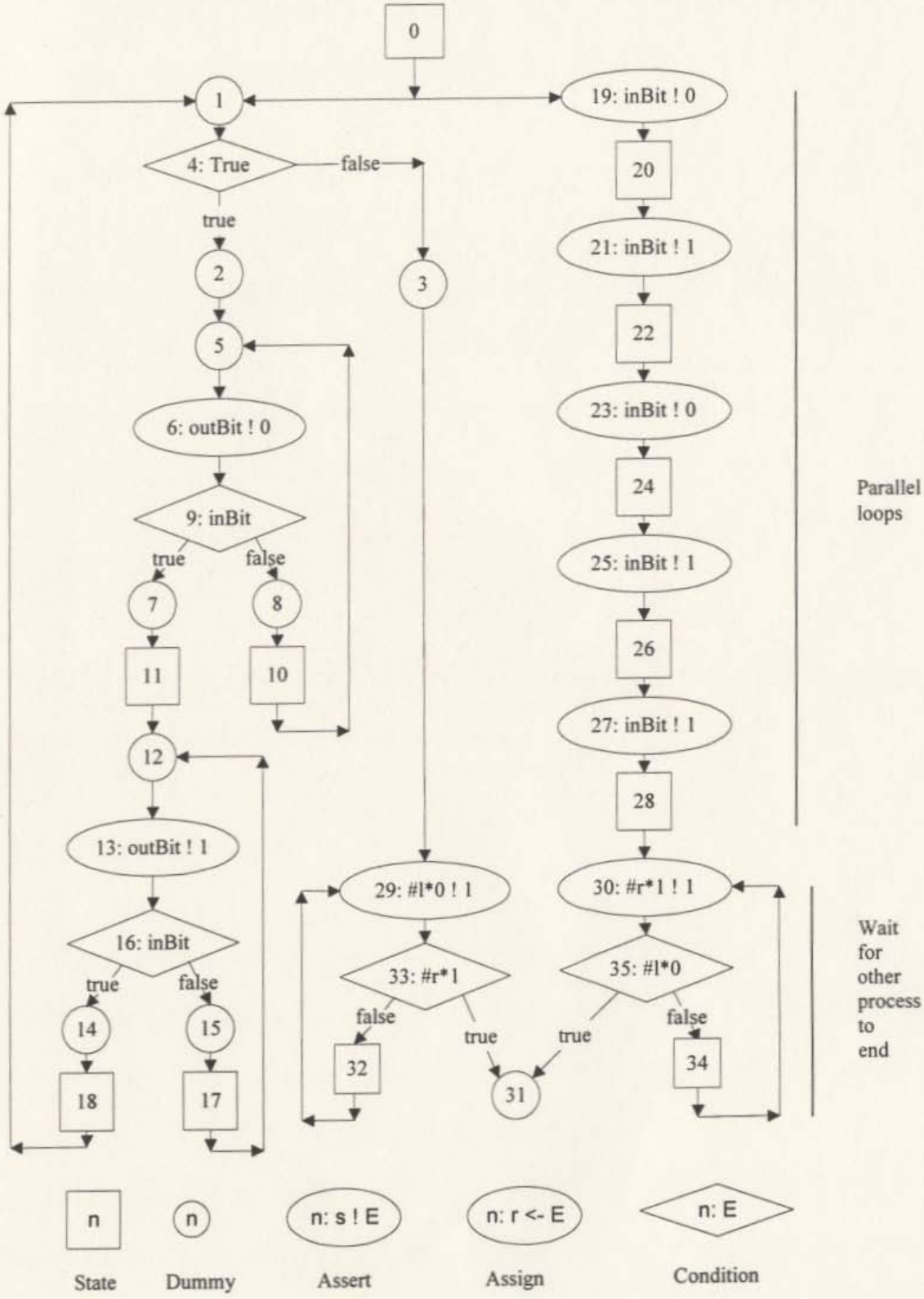


Figure 3.6: An Example ASM Chart

3.1.2 Data Flow Diagram for the Netlist Generator

Figure 3.7 gives the data flow diagram for the Netlist Generator. The processes for the Netlist Generator include the following:

- Generate the circuits of signals and registers according to "Entities" in Figure 3.3.
- Generate node circuits in terms of "Nodes" in Figure 3.4.
- Generate the circuits of expressions for the nodes that have the expressions.
- Link all the above circuits according to the edges of the graph in Figure 3.5.
- Form an output VHDL file representing the final circuits stored in the "Netlist" shown in Figure 3.7.

The following sections will describe the modules that implement the above processes, including the algorithms and the resulting circuits.

3.2 Utility Module

This module has defined some useful data types and useful functions. Gofer notations used in this module and other modules are explained in Appendix F. Especially, the *Assoc* type constructor is widely used in this thesis. Its definition is as follows:

type *Assoc a b* = [(*a*, *b*)]

In the above type synonym, *Assoc* is the name of a new type constructor; *a* and *b* are type variables representing the arguments of *Assoc*. [(*a*, *b*)] is a type expression. *Assoc a b* represents a finite map that associates members of type *a* with members of type *b*.

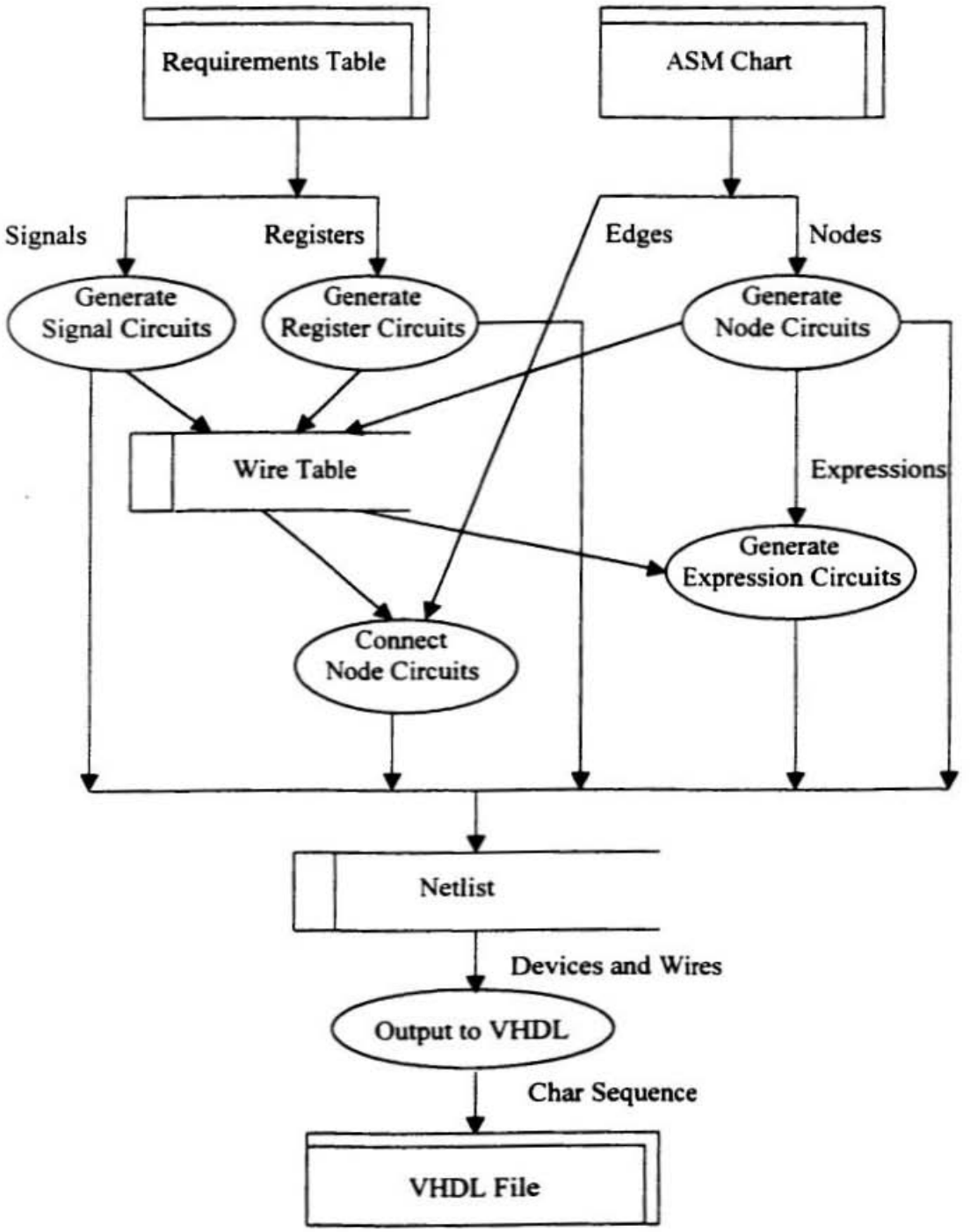


Figure 3.7: Data Flow Diagram for the Netlist Generator

3.3 Netlist Module

In this section, the data structure of the netlist and the types of gates and flip-flops used in the netlist are defined. The functions that create wires and connect devices are also described.

3.3.1 Netlist

The netlist consists of devices and wires. The devices chosen in this study include multi-input and one-output gates such as or-gates, and-gates and xor-gates, one-input and one-output gates such as not-gates and buffers, and D-type flip-flops.

For the devices, a data type named *DeviceKind* is defined in the following form.

```
data DeviceKind = DevK String                    ------ Device name
                      [(PortName, Bool)]         ------ Input ports
                      [PortName]                 ------ Output ports
```

The *String* represents the device name. The type named *PortName* is the same as *String* and is used to indicate whether the port is input or output. In the case of input ports, the **Bool** is used to describe whether the port can connect to more than one wire.

The devices used in the netlist are defined as follows:

```
andGate, orGate, xorGate, inverter, dFlipFlop, buffer :: DeviceKind
andGate = DevK "AND2" [("in", True)] ["out"]
orGate = DevK "OR2" [("in", True)] ["out"]
xorGate = DevK "XOR2" [("in", True)] ["out"]
inverter = DevK "INVERTER" [("in", False)] ["out"]
dFlipFlop = DevK "DFLIPFLOP" [("in", True)] ["out"]
buffer = DevK "BUF" [("in", False)] ["out"]
```

For the wires, the type *InOutLoc* is an enumerated type defined as

```
data InOutLoc =   In String          ---- Input wire
                | Out String         ---- Output wire
                | Local              ---- Local wire
```

The *String* in the above definition represents the wire name.

For the netlist, its type is defined as

```
data Netlist = NL Int                                  ---- The number of devices and wires
                  (Assoc DeviceID DeviceKind)          ---- Device map
                  (Assoc WireID InOutLoc)              ---- Wire map
                  (Assoc DeviceID (Assoc PortName [WireID]))   ---- Connection map
```

In the above form, the type named *DeviceID* and the type named *WireID* represent the identifiers of devices and wires, respectively. From the definition of the type *Netlist*, it can be seen that the data structure of the netlist shows the characteristics of devices and wires and the relationship between the device and its wires.

3.3.2 Functions

The functions exported from the Netlist module are called *createDevice*, *createLocalWire*, *createInputWire*, *createOutputWire*, and *connect*. They have the following meanings:

- *createDevice* is used to create devices.
- *createLocalWire*, *createInputWire*, and *createOutputWire* are used to create local wires, input wires, and output wires, respectively.
- *connect* is used to connect wires to ports of devices. This function needs to check that the device has the port and then to confirm that, unless permitted, the port is not already connected; otherwise the error information will be given.

3.4 Netlist State Module

This module defines some data types and some functions. For the Netlist state, the **tuple** type named *NLGState* is defined as (*Netlist*, *WireTab*, *ReqTab*). The type named *ReqTab* is intended to describe the characteristics of the signals and registers. The type named *WireTab* represents a function from Strings to *WireTabEntries*. *WireTab* and *WireTabEntries* are declared as follows:

data *WireTabEntry*

```
    =  WTELocSig  DeviceID      -- the identifier of the input gate.
                  WireID        -- the identifier of the wire named s.
     | WTEGlobSig DeviceID      -- the identifier of the input gate.
                  WireID        -- the identifier of the input wire named
                                -- assert_global_s.
                  WireID        -- the identifier of the wire named s.
     | WTELocReg  DeviceID      -- the identifier of the input gate named val.
                  DeviceID      -- the identifier of the input gate named assign.
                  WireID        -- the identifier of the wire named r.
     | WTEGlobReg DeviceID      -- the identifier of the input gate named val.
                  DeviceID      -- the identifier of the input gate named assign.
                  WireID        -- the identifier of the input wire named  val_global_r.
                  WireID        -- the identifier of the input wire named
                                -- assign_global_r.
                  WireID        -- the identifier of the wire named r.
     | WTENode    DeviceID      -- the identifier of the input gate named go.
                  WireID        -- the identifier of the output wire named done_N.
     | WTECondNode DeviceID     -- the identifier of the input gate named go.
                  WireID        -- the identifier of the done_N_then wire.
                  WireID        -- the identifier of the done_N_else wire.
                  Node          -- the identifier of the gate named thenNode
                  Node          -- the identifier of the gate named elseNode
     | WTEDataType String
     | SigWToDevice DeviceID String
     | RegWToDevice DeviceID DeviceID String
```

type *WireTab* = *Assoc String WireTabEntry*

The type named *WireTabEntry* contains information such as the identifiers of input gates and output wires for nodes, for local signals, and for local registers. For global signals and global registers, additional identifiers of input wires are needed. The information about signals and registers will be used in the generating signal and register circuits module and the generating expression circuits module. For condition nodes, additional identifiers of nodes are used to choose the control flow. The information about condition nodes is used in connecting node circuits module. The last three lines in *WireTabEntry* declaration are used to keep the names of signals and registers for output. Functions such as *createDeviceM*, *createInputWireM*, *createOutputWireM*, *createLocalWireM*, and *connectM* are the monad forms of the functions defined in the netlist module. The functions named *updateWireTabM* and *updateReqTabM* update the wire table and the requirements table.

3.5 A Module for Generating Signal and Register Circuits

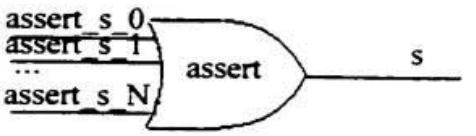
One of the processes for the Netlist Generator is to transform all the signals and registers used in the source program into their circuit descriptions. The algorithms for the transformation and the resulting circuits are shown in this section.

3.5.1 Generating Signal Circuits

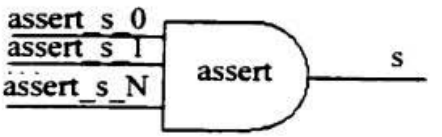
In the SMALL language, signals are used to carry information through space and the values of the signals are determined in the current clock period. Figure 3.8 shows the

circuit representations of the signals in the requirements table. It is noted that signals have an active level. The value of the active level of a signal depends on the default value that is the value of the signal in clock periods where it is not asserted. The default value of an active true signal is false and vice-versa. According to the definition and the assert node circuit, an or-gate is chosen in active-true signal circuit as shown in Figure 3.8(a) and Figure 3.8(c). An and-gate is used in active-false signal circuit as shown in Figure 3.8(b) and Figure 3.8(d). In Figure 3.8(c), an input wire named assert_global_*s*, an or-gate named *assert*, and its output wire named *s* are used to represent an active-true global signal *s*. The circuit representation of each active-false global signal *s* is shown in Figure 3.8(d). It consists of an input wire named assert_global_*s*, an and-gate named assert, and its output wire named *s*. The buffer and its output wire named global_*s* as shown in Figure 3.8(c) and Figure 3.8(d) are added for the need of the output VHDL file. In the specification of the Netlist Generator, every global signal is defined as the output port of the VHDL file that can not be used as an input to an internal component. Therefore, the signal *s* is used as both input and output and global_*s* is only used as an output port. The same goes for registers. It is noted that input wires named assert_*s*_0, assert_*s*_1, and assert_*s*_*N* are not generated here and only shown for illustrative purpose. They are from the outputs of the and-gates or or-gates produced in the circuits of the assert nodes that use the signals as the targets. Section 3.8 gives a simple example to explain the relationship between signal circuits and assert node circuits.

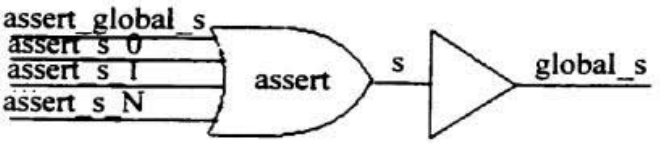
For a signal that has the type `array` *n* `of` *T*, the circuit representation of each element in the array should be generated. The identifiers of the gates named *assert*, the



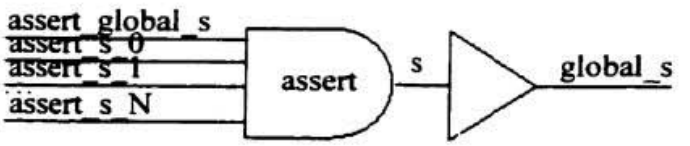
(a) Active_True Local Signal Circuit



(b) Active_False Local Signal Circuit



(c) Active_True Global Signal Circuit



(d) Active_False Global Signal Circuit

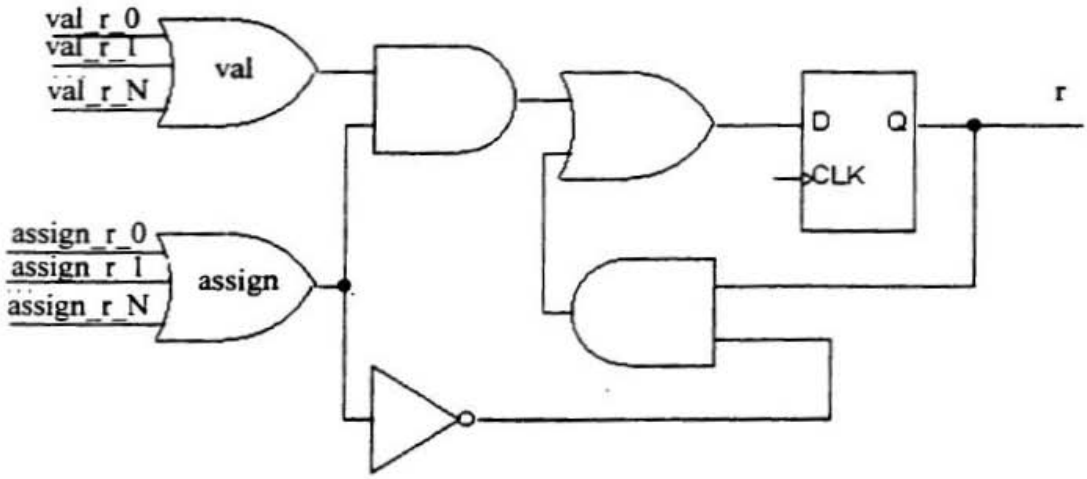
Figure 3.8: The Signal Circuits

input wires named assert_global_*s*, and the output wires named *s* shown in Figure 3.8 are added to the wire table when the circuit of each signal named *s* is created.

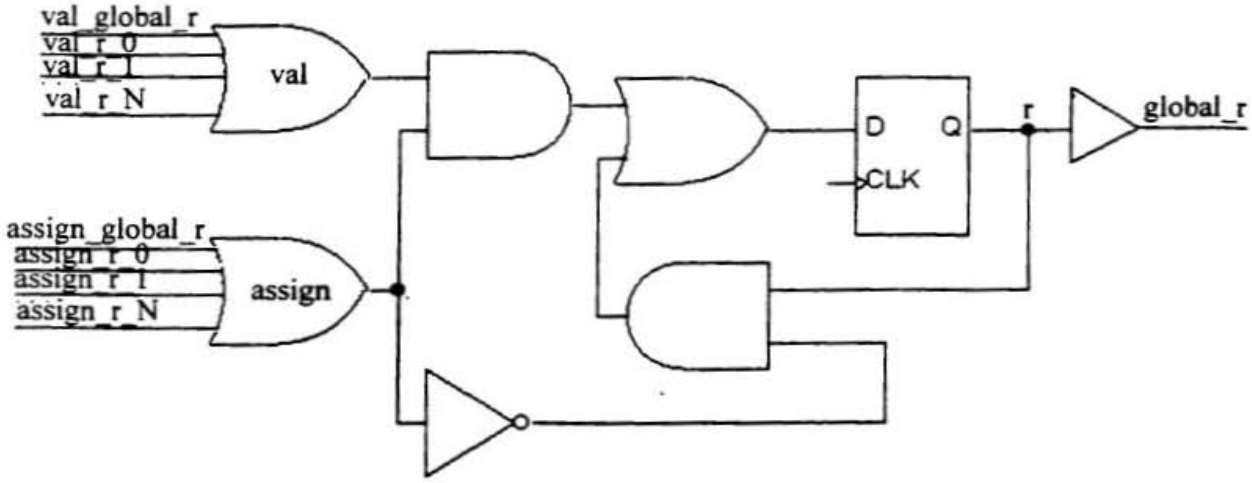
3.5.2 Generating Register Circuits

The circuits of the registers in the requirements table are generated in Figure 3.9. The D-type flip-flops are used in the circuits for the registers to store information. The value of a register is the latest value assigned to it in a previous clock period. For each local register named *r*, as shown in Figure 3.9(a), its transformation is composed of three or-gates, two and-gates, a not-gate, a D-type flip-flop, and several wires. In Figure 3.9(b), additional input wires named val_global_*r* and assign_global_*r*, a buffer, and a wire named global_*r* are needed for each global register named *r*. The buffer is generated for the same reason as the buffers in Figure 3.8. All flip-flops in the netlist use the same clock line. The clock is designed as a global VHDL signal. Similar to the transformations of the signals, input wires named val_*r*_0, val_*r*_1, val_*r*_*N*, assign_*r*_0, assign_*r*_1, and assign_*r*_*N* are not generated here and only shown for illustrative purpose. They are the output wires named val_*Target* and assign_*Target* produced in the circuits of the assignment nodes that use the register as the target (see Figure 3.13).

The transformation of a register that has the type `array` *n* `of` *T* is similar to the transformation of the signal that represents an array. As shown in Figure 3.9, the identifiers of the two input or-gates named *val* and *assign*, the input wires named val_global_*r* and assign_global_*r*, and the output wire named *r* are updated to the wire table when the circuit of each register named *r* is created.



(a) Local Register Circuit



(b) Global Register Circuit

Figure 3.9: The Register Circuits

3.6 A Module for Generating Node Circuits

This module generates the circuits of all the nodes in the ASM chart. Each node is labelled. Labels come in five varieties: dummy nodes, state nodes, assert nodes, assignment nodes, conditional nodes. The function named *genOneNdCircuit* generates the circuit for one node. When the circuits are generated for dummy nodes, state nodes, assert nodes, and assignment nodes, the identifiers of the input or-gate named *go* and the output wire named done_*N* are kept in the wire table. For condition nodes, the identifiers of the input or-gate named *go* and the two output wires named done_*N*_then and done_*N*_else and two nodes named *thenNode* and *elseNode* are stored in the wire table.

The dummy nodes are place-holders. The control flow goes directly through these nodes. The circuit of a dummy node simply consists of an or-gate and an output wire, as shown in Figure 3.10. The wires named done_p0, done_p1, and done_p*M* are not generated here and they are from the output wires that are produced in the circuits of the predecessor nodes. The same goes for all node types.

The state nodes represent the control state of the program. They are used to wait for the coming of the next clock period. The first node listed in the ASM chart is always a state node and represents the start of the program. As shown in Figure 3.11(a), its circuit is composed of a D-type flip-flop, an inverter, and three wires. The input wire is a true wire. It is required that the output wire should be true in the first clock period in order to start the program. Figure 3.11(b) gives the circuit generated for a state node numbered *N*.

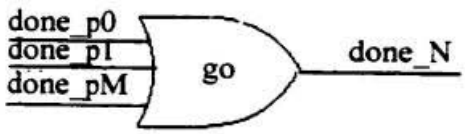
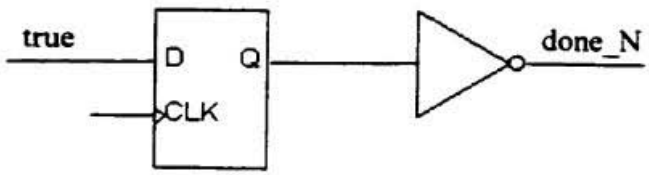
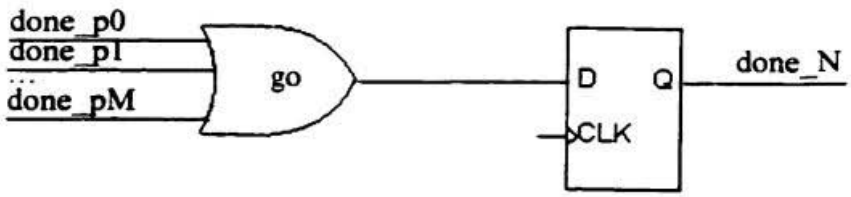


Figure 3.10: The Dummy Node Circuit



(a) The Initial State Node Circuit

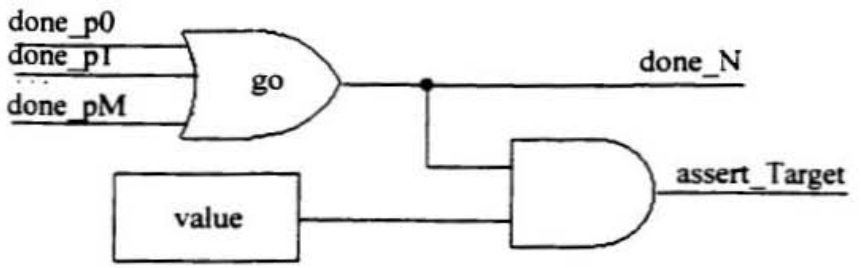


(b) The Circuit of the State Node Numbered N

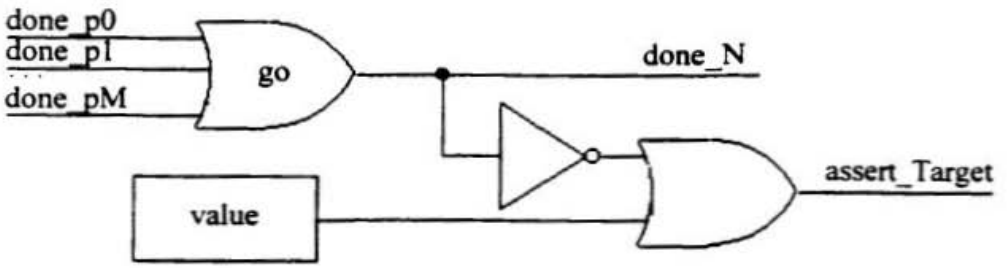
Figure 3.11: The State Node Circuits

The assert nodes are used to transfer information to signals. In the ASM chart, assert nodes are labelled with two expressions: target and value. The target expression represents the left-hand-side of an assert statement and will be a signal. The value expression indicates the right-hand-side of an assert statement and will be one of seven kinds of expressions described in the next section. Figure 3.12 gives the assert node circuits. Before generating the assert node circuits, the circuits of the two expressions are created. In Figure 3.12, the value expression circuits are represented by a box labelled *value* or by several boxes labelled *value0, value1, ... , valueN* in the case of an array target. As mentioned in generating signal and register circuits module, the output wire named assert_*Target* in Figure 3.12(a) will be the input of the or-gate in the circuit for the target signal of the assert node. The assert_*Target* output in Figure 3.12(b) will be the input of the and-gate in the circuit for the target signal of the assert node. The circuit in Figure 3.12(c) is generated for an assert node whose value expression is an array expression and whose target expression is an active-true array signal. The output wires named assert_*Target0*, assert_*Target1*, ... , assert_*TargetN* will be inputs to several or-gates in the circuit for the target signal of the assert node. Asserting an active-false array signal circuit is not shown. It can be obtained according to Figure 3.12(b).

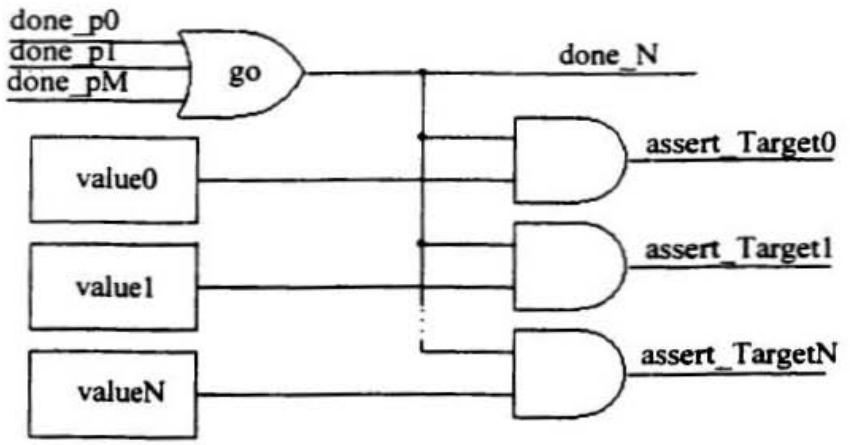
The assignment nodes are used to store information in registers. Similar to assert nodes, each assignment node is associated with a target expression and a value expression. The target expression represents the left-hand-side of an assignment statement and will be a register. The value expression indicates the right-hand-side of an assignment statement and will be one of seven kinds of expressions described in the next



(a) Asserting an Active-True Signal Circuit



(b) Asserting an Active-False Signal Circuit

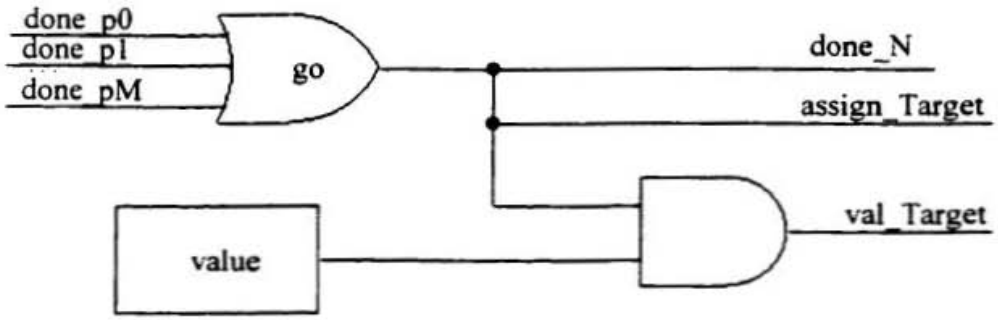


(c) Asserting an Active-True Array Signal Circuit

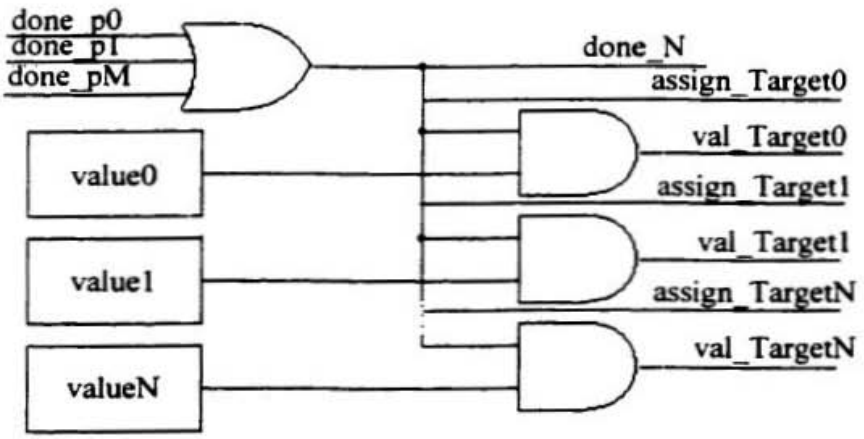
Figure 3.12: The Assert Node Circuits

section. The circuits shown in Figure 3.13 are used to represent the assignment nodes. Figure 3.13(a) represents an assignment node whose target expression is a bool type register. The box labelled *value* represents the circuit of the value expression of the assignment node. The wires named assign_*Target* and val_*Target* are the input wires of the or-gates named *assign* and *val* in the target register circuit, respectively. The wire named done_*N* of this node circuit will be the input wire of the or-gate in its successor node. The circuit shown in Figure 3.13(b) is generated for an assignment node that has an array type register and an array expression. Each of the boxes that are labelled *value*0, *value1*, ..., *valueN* represents a value output wire of the array expression. Each of the output wires named val_*Target0*, val_*Target1*, ... , val_*TargetN* will be the input wire of an or-gate named *val* in the target register circuit. The wires named assign_*Target0*, assign_*Target1*, ..., assign_*TargetN* will be the input wires of the several or-gates named *assign* in the target register circuit.

The condition nodes have associated a condition expression that will be true or false and two nodes called *thenNode* and *elseNode*. When the expression is true, the node named *thenNode* will be executed; when the expression is false, the node named *elseNode* will be executed. The circuit for each condition node is generated as shown in Figure 3.14. It is composed of an or-gate named *go*, two and-gates named *then* and *else*, an inverter, a box and several wires. The inverter is used to implement the choice of the control flow. In Figure 3.14, the box labelled *condition expression* represents the expression circuit of the condition node circuit. The output wire named done_*N*_then in the condition node circuit will be the input wire of the *go* gate of the node named



(a) Assigning a Register Circuit



(b) Assigning an Array Register Circuit

Figure 3.13: The Assignment Node Circuits

thenNode. The output wire named done_*N*_else in the condition node circuit will be the input wire of the *go* gate of the node named *elseNode*.

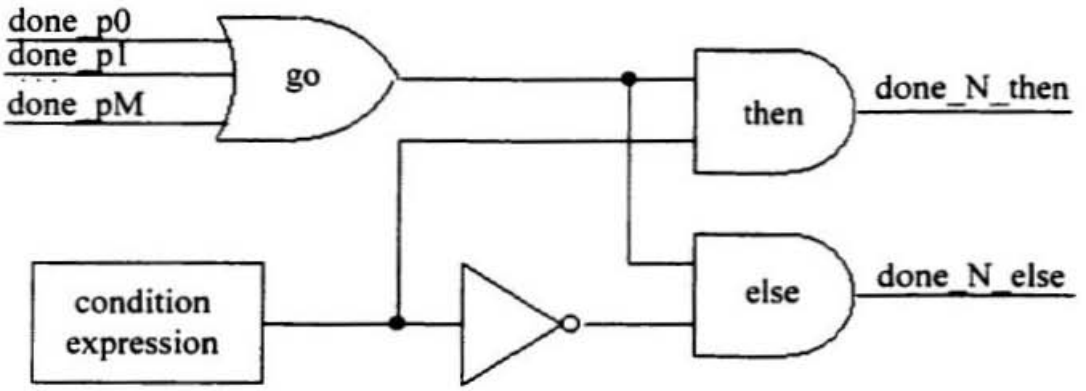


Figure 3.14: The Condition Node Circuit

3.7 A Module for Generating Expression Circuits

This module will transform all expressions that are associated with assert nodes, assignment nodes, and condition nodes into their corresponding circuits. When generating the circuits of these nodes, the expressions will be transformed into their circuit representations as a part of these node circuits. In the SMALL language, there are seven kinds of expressions: identifier expressions, constant expressions, unary expressions, binary expressions, subscript expressions, subarray expressions and array building expressions. Each expression is transformed into a circuit with the value outputs for the value of the expression and an overflow output that indicates whether overflow

occurred in the calculation of the value output. When adding or subtracting two's complement numbers and if the result is too large or too small to be represented in the specified range, we say that "overflow" occurs. The function named *genExpCircuits* is defined to implement the transformation. Its input parameter is the expression containing the type and the location and other information. Its outputs are wires representing the value outputs and the overflow output. For each kind of expression, the following gives its circuit representation.

The constant expression, `true`, is implemented by a true wire as the output value and a false wire as the overflow output. The constant expression, `false`, is implemented by a false wire as the output value and a false wire as the overflow output.

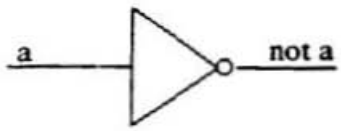
An identifier expression is a string referring to a signal or a register in the requirements table. The output value of the identifier expression referring to a bool type signal or register is the output named *s* or *r* of its signal or register circuit. The false wire is used to express the fact that no overflow occurs in generating the identifier expression circuit. In the case of the identifier referring to an array type signal or register, the output values will be a list of the outputs of every element in the array. The overflow output will be false wire.

A unary expression consists of an expression and an operator that is one of `not`, `-`, and `overflow`. The `not` expression is implemented by an inverter as shown in Figure 3.15(a). The `overflow` expression will be true if there is overflow in the evaluation of its operands. The overflow output is the disjunction of all overflow outputs of the

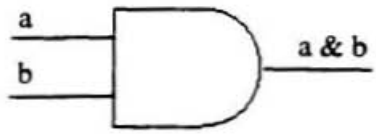
operands. The same goes for other expressions that have no specific explanation of overflow. The expression *-B* distributes to the array of boolean level and is two's complement negation. A ripple carry negater should be used for its implementation. For convenience, this thesis uses a ripple carry adder and inverters instead. The circuit will be discussed in array operations.

A binary expression contains two expressions and one operator that is one of &, `or`, `=>`, `<=`, `==`, `=/=`, `=`, `/=`, `++`, `plus`, `-`, and `uplus`. The former six operators and the `not` operator distribute to the bit level. Their meanings are given in (Norvell, 1998). Their expressions are implemented by the circuits shown in Figure 3.15. It is noted that the inputs in Figure 3.15 are the value outputs of the operands. The operators, `=` and `/=`, are used to compare two expressions. If the value outputs of the two expressions have the same value, the expression "*A* = *B*" will have a true result; otherwise, the expression "*A* /= *B*" will have a true result. Similar to the negation operator, the operators such as `++`, `plus`, `-`, and `uplus` distribute to the array of boolean level. The function named *distributelbM* is defined to implement these operations for one dimensional or multi-dimensional array operands. Because the results of multi-dimensional array operands can be obtained by applying these operators to the one-dimensional arrays, the following simply describes the definitions and the circuit implementations of these operators on one-dimensional arrays. Detailed definitions on these operators can be found in (Norvell, 1996 and 1998).

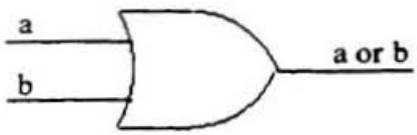
The expression, *A* ++ *B*, implements the catenation of two arrays. A function named *catM* is defined to generate circuits for this operation. The value outputs of the expression



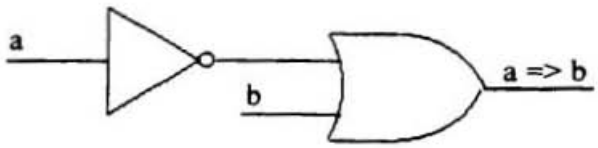
(a) Expression "not a" Circuit



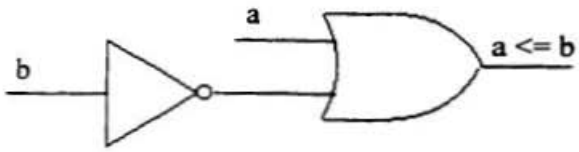
(b) Expression "a & b" Circuit



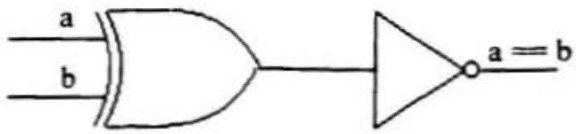
(c) Expression "a or b" Circuit



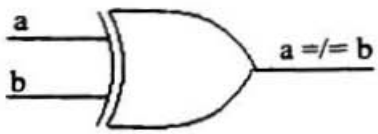
(d) Expression "a => b" Circuit



(e) Expression "a <= b" Circuit



(f) Expression "a == b" Circuit



(g) Expression "a =/= b" Circuit

Figure 3.15: The Expression Circuits

consist of the value outputs of the expression *A* as the start and the value outputs of the expression B as the end. The expression *A* `uplus` *B* implements the unsigned addition of two numbers. The result length is one more than the length of the longer of *A* and *B*. No overflow occurs in the above two operations. However, if there is overflow in evaluating *A* or *B*, the overflow outputs of the above two expressions are true. The expression, *A* `plus` *B*, carries out the two's complement addition of two numbers. The binary expression, *A* - *B*, implements the two's complement subtraction of two numbers. Functions named *uplusM*, *plusM*, and *minusM* are defined to generate circuits for these operations. The unary expression, - *B*, is used for the two's complement negation of a number. A function named *minusM* is defined for the expression that is the same as [false, false, ... , false] - *B*. A ripple carry adder shown in Figure 3.16 is generated for the circuits of the expressions with `uplus` and `plus` operators. A function named *genAdder* is defined to generate circuits for the ripple carry adder. In Figure 3.16, each box represents the circuit shown in Figure 3.17 and generated for a full adder. The following logic equations for the full adder are described in (Lenk, 1972).

$$\begin{aligned} S &= A\bar{B}\bar{C}_{in} + \bar{A}B\bar{C}_{in} + \bar{A}\bar{B}C_{in} + ABC_{in} \\ &= A \oplus B \oplus C_{in} \end{aligned}$$

$$C_{out} = AB + (A + B)C_{in}$$

In the above equations, *A* and *B* and C_{in} are the inputs; *S* and C_{out} represent the outputs for the sum and the carry; C_{in} is 0 for the addition of the least significant bits.

For operations with - *B* and *A* - *B*, a ripple carry subtracter should be used. However, for the convenience, this thesis uses a ripple carry adder and inverters to implement the

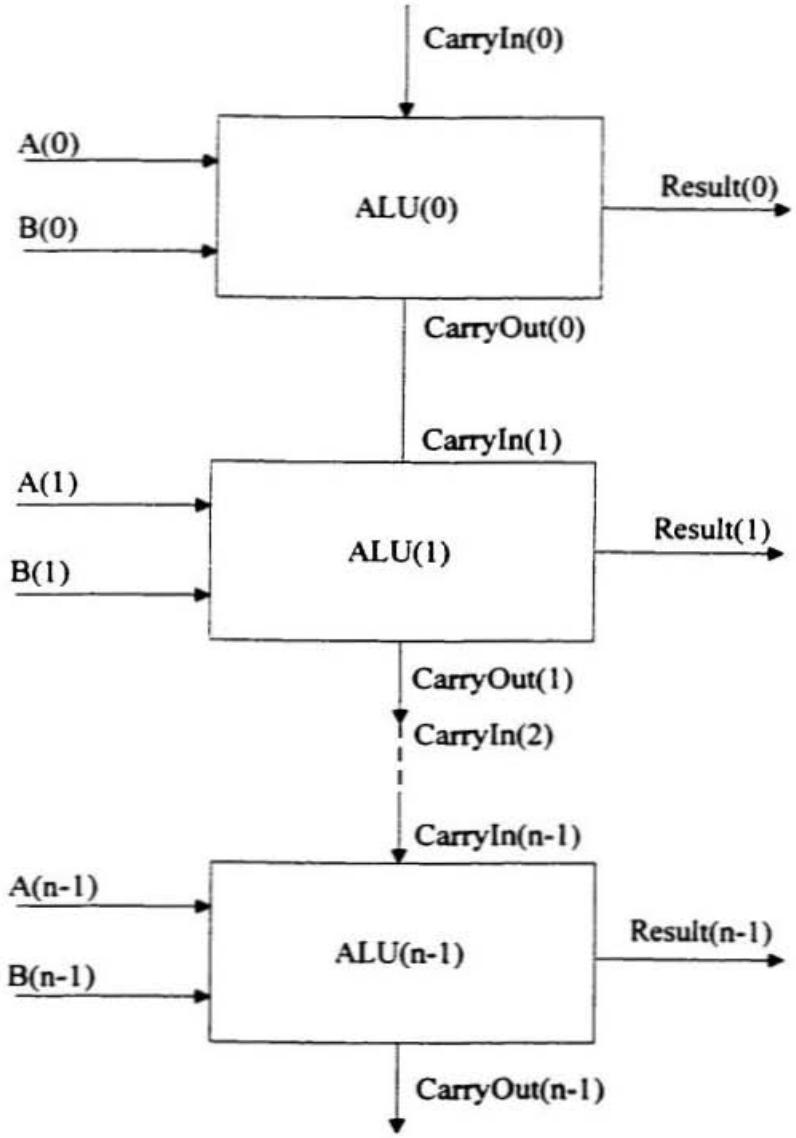


Figure 3.16: A Ripple Carry Adder Circuit

operations. Before using the function named *genAdder*, we implement the operation with `not` for every element in *B* and set the initial carry named *CarryIn0* to 1 to make the adder perform subtraction.

The overflow output for the operation with `uplus` is false. Figure 3.18 shows the overflow output circuit for the operation with `plus`. In Figure 3.18, *A*_Sign and *B*_Sign are the sign bits of the two operands; *S*_Sign is the sign bit of the sum; *Overflow* represents whether overflow occurs. The logic equation for Figure 3.18 is as follows:

$$Overflow = S\_Sign\ \overline{A\_Sign}\ \overline{B\_Sign} + \overline{S\_Sign}\ A\_Sign\ B\_Sign$$

Figure 3.18 is also used to generate the overflow output circuit for the expression *A*-*B* and the expression -*B*; however, the inverters should be used for the expression *B*. The logic equation is changed to the following form.

$$Overflow = S\_Sign\ \overline{A\_Sign}\ B\_Sign + \overline{S\_Sign}\ A\_Sign\ \overline{B\_Sign}$$

An array building expression is a list of expressions. The value outputs of the array building expression are a bundle of the value outputs of the expressions. An or-gate is used for disjoining all overflow outputs of the expressions in the array to get the overflow output of the array building expression.

A subscripted array expression contains an expression as an operand and a constant number as the subscript. The value output of the subscripted array expression is selected from the value outputs of the operand in terms of the constant number. The overflow output of the operand is the overflow output of the subscripted array expression.

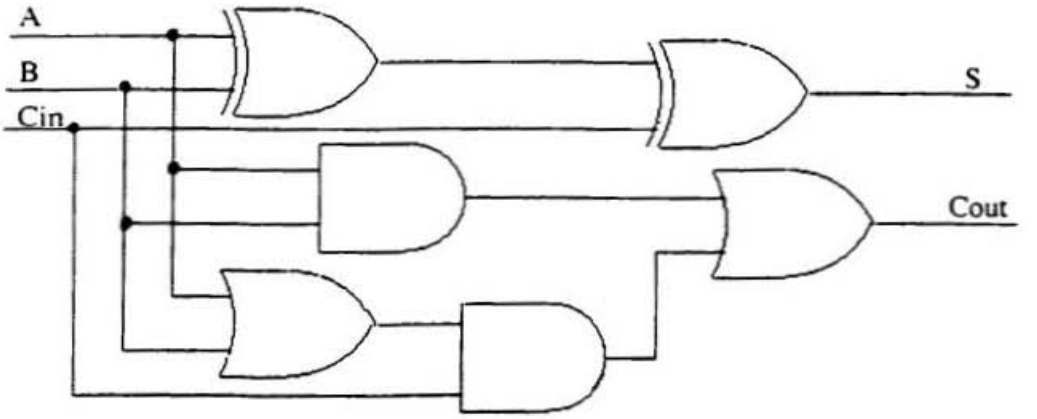


Figure 3.17: A Full Adder Circuit

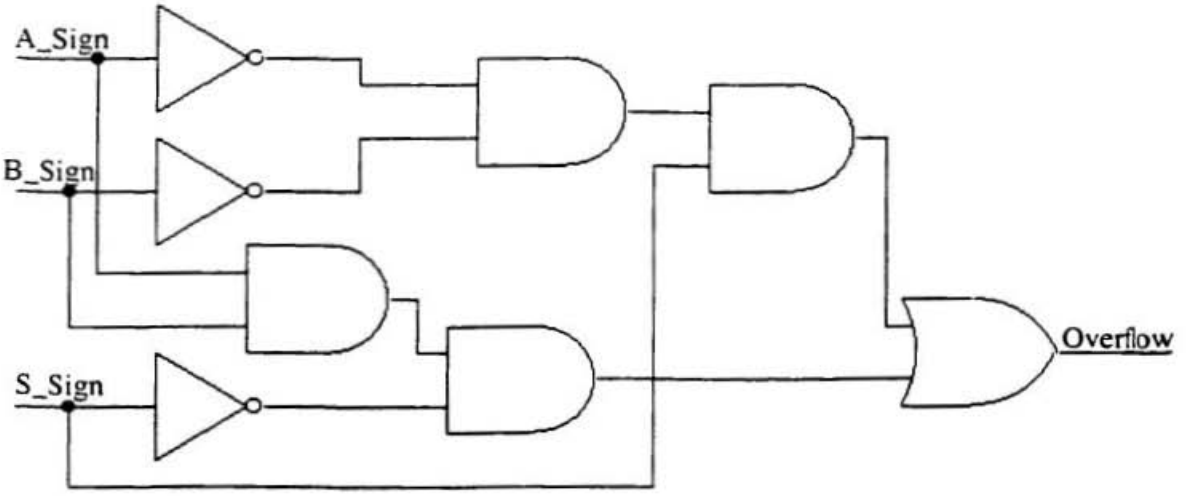


Figure 3.18: The Overflow Output Circuit for Operators: "plus" and -

A subarray expression is composed of an expression as an operand and two constant numbers as the length and as the starting subscript. The value outputs of the subarray expression are a subsequence of the value outputs of the operand in terms of the two constant numbers. The overflow output of the subarray expression is the same as the overflow output of the operand.

3.8 A Module for Connecting Node Circuits

This module links all the node circuits in terms of the edges in the ASM chart and the information from the wire table. The function named *connectNode* is defined in this module. The output wire named done_*N* of each node except for a condition node will be connected to every or-gate named *go* of its successor node. In the case of a condition node, the output wire named done_*N*_then will be linked to the go gate of its *thenNode* node and the output wire named done_*N*_else will be connected to the go gate of its *elseNode* node. Once the nodes are connected, the netlist is complete.

A simple SMALL example shown in Figure 3.19 is used to illustrate how to generate the circuits of signals and assert nodes and how to connect these circuits. Figure 3.20 gives an ASM chart for the SMALL program shown in Figure 3.19. Each node shown in Figure 3.20 is transformed into its corresponding circuit as shown in Figure 3.21. For example, the flip-flop labelled 1 and the inverter labelled 1 represent Node 0 and the or-gate labelled *go1* is generated to represent Node 1, a dummy node. In Figure 3.20, the done_*N* wire of each node is also labelled in Figure 3.21. According to the *done_N* wires, all the node circuits are connected to represent the circuit of the SMALL program shown

```
global sig a: bool          //0

  while true do             //1

       a ! true             //2
       tick                 //3
       a ! false            //4
       tick                 //5
       a ! true             //6

  od                        //7
```

Figure 3.19: A Simple SMALL Program

in Figure 3.19. From Figure 3.21, it can be seen that the or-gate labelled *assert* and a buffer are used to represent the circuit of global signal *a*. It has three inputs that are the outputs from three and-gates. Each of these and-gates is generated for one of three assert nodes. And-gates labelled 3, 4, and 5 are used for the assert statements of the SMALL program in Line 2, Line 4, and Line 6, respectively. For example, the or-gate labelled *go6* and the and-gate labelled 4 are used to represent Node 7 in Figure 3.20 that is an assert node and represents the assert statement in Line 4 in Figure 3.19. The or-gate labelled *go5* and the flip-flop labelled 2 generated for Node 6 are used for the `tick` statement in Line 3. The `tick` statement in Line 5 is converted to a state node labelled 8 and represented by the or-gate labelled *go7* and the flip-flop labelled 3. The or-gate labelled *go9* and the flip-flop labelled 4 represent the implicit `tick` statement in the `while` loop. The or-gate labelled *go2*, the inverter labelled 2, and two and-gates labelled 1, 2 are generated for Node 4 that is a condition node. The condition expression is always true so the done_2_then wire is always excuted.

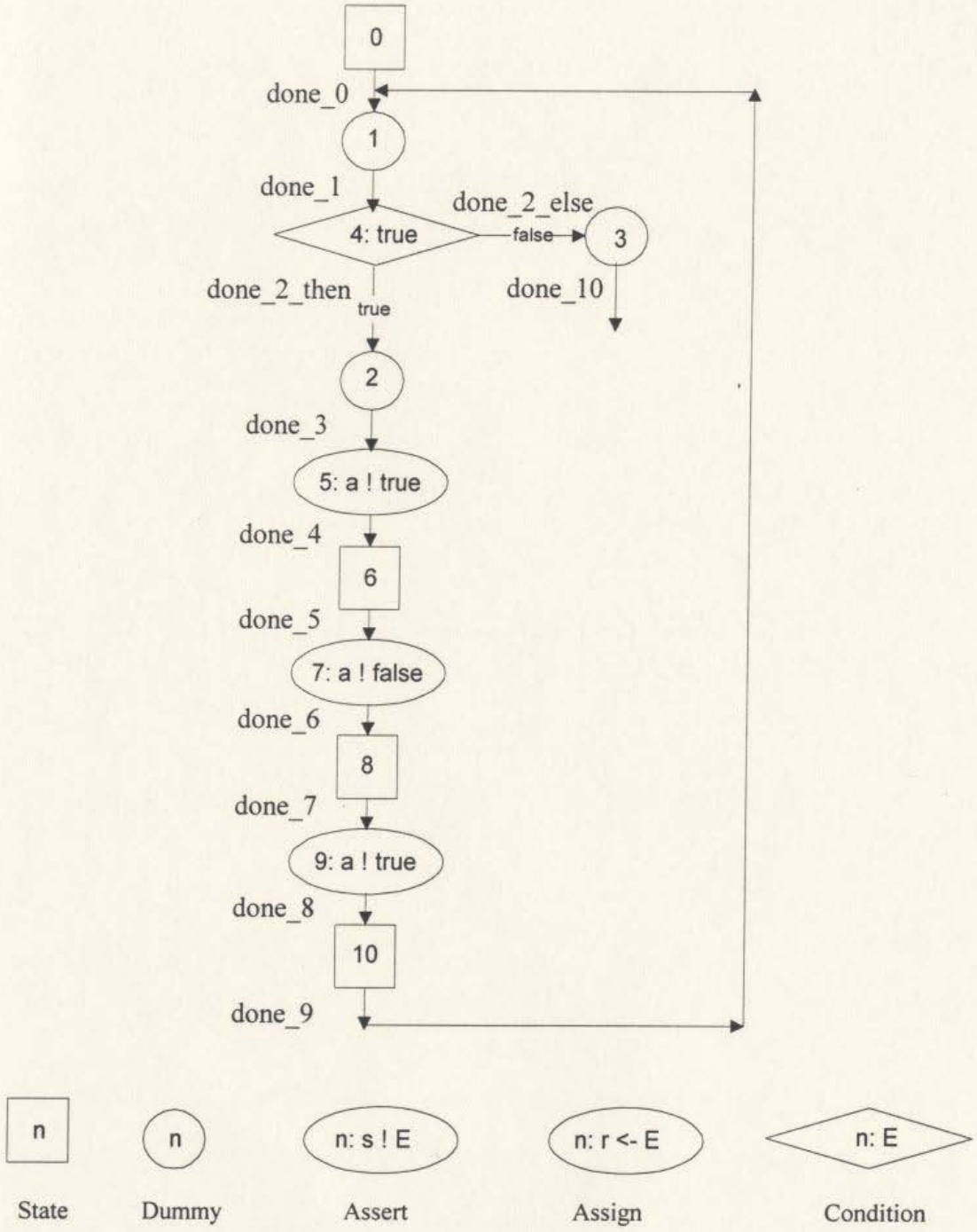


Figure 3.20: An ASM Chart for the SMALL Program Shown in Figure 3.19

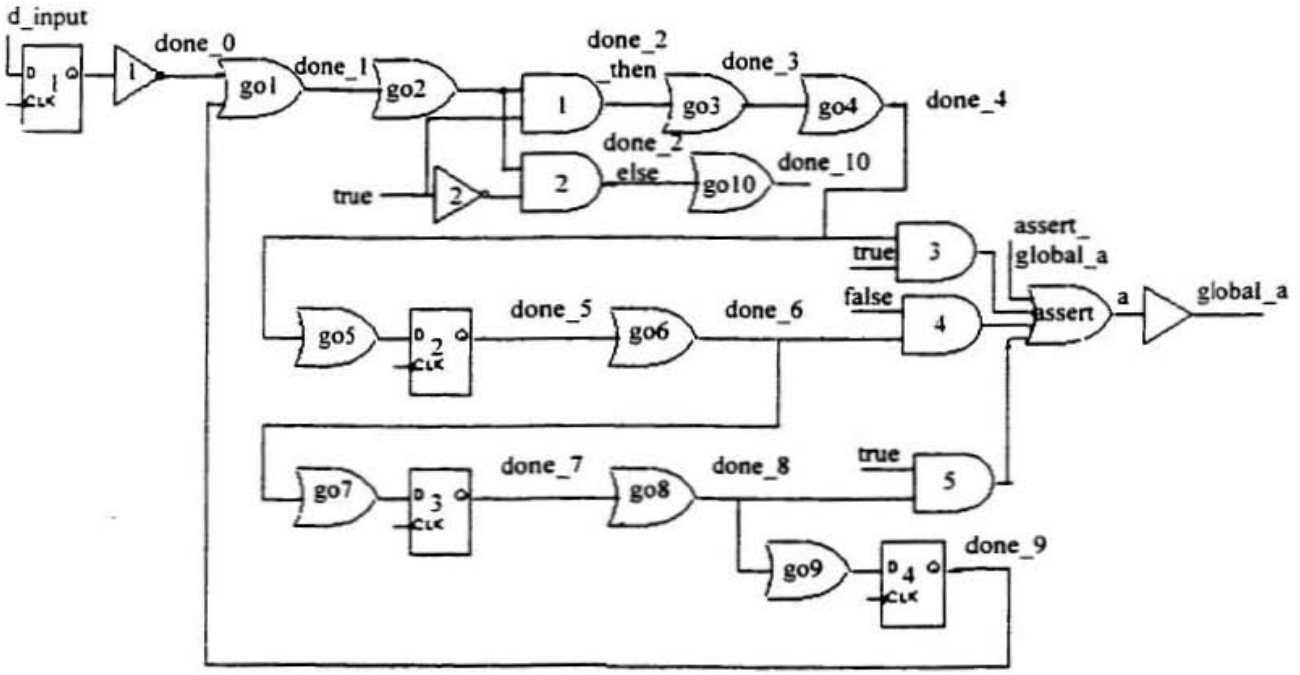


Figure 3.21: The Circuit of the SMALL Program Shown in Figure 3.19

3.9 Output to VHDL Module

Using the IEEE std_logic_1164 packages, this module produces a VHDL source file that represents the final netlist circuit generated by the above modules. The entity declaration of the VHDL file includes the name of the ports, the direction of its data flow, and the data type. The output ports are composed of the output of every global signal and register. The input ports include the clock, the power, the ground, the wire named assert_global_*s* for each global signal called *s*, and the wires named assign_global_*r* and

val_global_*r* for each global register called *r*. The input of the D-type flip-flop in the initial state node circuit is also considered as an input port. For multi-dimensional arrays, a package is used to declare several one-dimensional array types and these ports will have user-defined array types. The file contains a VHDL structural description of the components and their interconnections by signals. The components consist of and-gates, or-gates, xor-gates, not-gates, buffers, and D-type flip-flops. The and-gates, the or-gates, and the xor-gates have two-inputs and one-output. The not-gates and the buffers have one-input and one-output. Their VHDL descriptions will be given in the next chapter. An example VHDL output description for the SMALL program shown in Figure 3.1 is given in Figure 3.22.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_OUT1 is
 port (assert_global_outBit : in std_logic;
     assert_global_inBit : in std_logic;
     d_input : in std_logic;
     clock : in std_logic;
     power : in std_logic;
     ground : in std_logic;
     global_outBit : out std_logic;
     global_inBit : out std_logic);
end SMALL_OUT1;

architecture Structure of SMALL_OUT1 is

 signal  wire199, wire197, wire195, wire193, wire190, wire157, wire156,
       wire153, wire151, wire149, wire147, wire146, wire145, wire144,
       wire139, wire137, wire135, wire133, wire131, wire129, wire128,
       wire127, wire126, wire121, wire120, wire117, wire116, wire113,
       wire111, wire109, wire107, wire105, wire103, wire102, wire101,
       wire100, wire95, wire94, wire91, wire90, wire87, wire85,
       wire83, wire82, wire79, wire77, wire75, wire74, wire71,
```

```
        wire69, wire67, wire66, wire63, wire61, wire59, wire58,
        wire55, wire53, wire51, wire50, wire47, wire45, wire43,
        wire41, wire39, wire37, wire36, wire33, wire32, wire31,
        wire30, wire25, wire24, wire21, wire20, wire19, wire18,
        wire13, wire11, wire7, wire2: std_logic;

component OR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component AND2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component BUF
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component INVERTER
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component DFLIPFLOP
 port (clock, D : in std_logic; Q : out std_logic);
end component;

begin

 Device200: OR2 port map ( assert_global_inBit, wire199, wire2);
 Device198: OR2 port map ( wire55, wire197, wire199);
 Device196: OR2 port map ( wire63, wire195, wire197);
 Device194: OR2 port map ( wire71, wire193, wire195);
 Device192: OR2 port map ( wire79, wire87, wire193);
 Device191: OR2 port map ( assert_global_outBit, wire190, wire7);
 Device189: OR2 port map ( wire111, wire137, wire190);
 Device188: BUF port map ( wire43, wire11);
 Device187: BUF port map ( wire47, wire13);
 Device186: BUF port map ( wire41, wire18);
 Device185: BUF port map ( wire21, wire24);
 Device184: BUF port map ( wire45, wire30);
 Device183: BUF port map ( wire33, wire36);
 Device182: BUF port map ( wire53, wire50);
 Device181: BUF port map ( wire59, wire53);
 Device180: BUF port map ( wire61, wire58);
 Device179: BUF port map ( wire67, wire61);
```

```
Device178: BUF port map ( wire69, wire66);
Device177: BUF port map ( wire75, wire69);
Device176: BUF port map ( wire77, wire74);
Device175: BUF port map ( wire83, wire77);
Device174: BUF port map ( wire85, wire82);
Device173: BUF port map ( wire157, wire85);
Device172: BUF port map ( wire107, wire90);
Device171: BUF port map ( wire105, wire94);
Device170: BUF port map ( wire109, wire100);
Device169: BUF port map ( wire103, wire105);
Device168: BUF port map ( wire102, wire107);
Device167: BUF port map ( wire113, wire109);
Device166: BUF port map ( wire133, wire116);
Device165: BUF port map ( wire131, wire120);
Device164: BUF port map ( wire135, wire126);
Device163: BUF port map ( wire129, wire131);
Device162: BUF port map ( wire128, wire133);
Device161: BUF port map ( wire139, wire135);
Device160: BUF port map ( wire153, wire144);
Device159: BUF port map ( wire147, wire149);
Device158: BUF port map ( wire146, wire151);
Device155: INVERTER port map ( wire156, wire157);
Device154: DFLIPFLOP port map ( clock, d_input, wire156);
Device152: OR2 port map ( wire91, wire157, wire153);
Device143: AND2 port map ( wire145, wire144, wire147);
Device142: AND2 port map ( wire144, power, wire146);
Device141: INVERTER port map ( power, wire145);
Device138: OR2 port map ( wire151, wire121, wire139);
Device136: AND2 port map ( wire135, ground, wire137);
Device125: AND2 port map ( wire127, wire126, wire129);
Device124: AND2 port map ( wire126, wire2, wire128);
Device123: INVERTER port map ( wire2, wire127);
Device119: DFLIPFLOP port map ( clock, wire120, wire121);
Device115: DFLIPFLOP port map ( clock, wire116, wire117);
Device112: OR2 port map ( wire117, wire95, wire113);
Device110: AND2 port map ( wire109, power, wire111);
Device99: AND2 port map ( wire101, wire100, wire103);
Device98: AND2 port map ( wire100, wire2, wire102);
Device97: INVERTER port map ( wire2, wire101);
Device93: DFLIPFLOP port map ( clock, wire94, wire95);
Device89: DFLIPFLOP port map ( clock, wire90, wire91);
Device86: AND2 port map ( wire85, ground, wire87);
Device81: DFLIPFLOP port map ( clock, wire82, wire83);
Device78: AND2 port map ( wire77, power, wire79);
Device73: DFLIPFLOP port map ( clock, wire74, wire75);
```

```
  Device70: AND2 port map ( wire69, ground, wire71);
  Device65: DFLIPFLOP port map ( clock, wire66, wire67);
  Device62: AND2 port map ( wire61, power, wire63);
  Device57: DFLIPFLOP port map ( clock, wire58, wire59);
  Device54: AND2 port map ( wire53, power, wire55);
  Device49: DFLIPFLOP port map ( clock, wire50, wire51);
  Device46: AND2 port map ( wire45, power, wire47);
  Device44: OR2 port map ( wire149, wire37, wire45);
  Device42: AND2 port map ( wire41, power, wire43);
  Device40: OR2 port map ( wire51, wire25, wire41);
  Device38: OR2 port map ( wire32, wire20, wire39);
  Device35: DFLIPFLOP port map ( clock, wire36, wire37);
  Device29: AND2 port map ( wire31, wire30, wire33);
  Device28: AND2 port map ( wire30, wire11, wire32);
  Device27: INVERTER port map ( wire11, wire31);
  Device23: DFLIPFLOP port map ( clock, wire24, wire25);
  Device17: AND2 port map ( wire19, wire18, wire21);
  Device16: AND2 port map ( wire18, wire13, wire20);
  Device15: INVERTER port map ( wire13, wire19);
  Device9: BUF port map ( wire7, global_outBit);
  Device4: BUF port map ( wire2, global_inBit);
end Structure;
```

Figure 3.22: An Example VHDL Output Description

Chapter 4

Hardware Implementation of the SMALL Language with FPGAs

This chapter shows the hardware implementation of SMALL programs with Field Programmable Gate Arrays (FPGAs). CAD tools provided by Canadian Microelectronics Corporation (CMC) are used for the hardware implementation of the programs written in the SMALL language. First, the Netlist Generator described in Chapter 3 transforms the SMALL program to a structural VHDL file whose components are simple gates and clocked D-type flip-flops. Then a functional simulation of the VHDL description is carried out with the Synopsys VSS (VHDL System Simulation) (Xilinx, 1995). After the simulation, a graph in Waveform Viewer Window can be used to verify that the circuit performs the specified requirements. The Synopsys FPGA Compiler is used for synthesising and optimising the gate-level VHDL description. Finally, the Design Manager of the Xilinx Alliance Series version 1.4 (Xilinx, 1997b) is used to create configuration data for Xilinx FPGA chips. The Xilinx XC4000 family is employed as the target FPGA device. Several SMALL programs were compiled, simulated and synthesised to verify that the Netlist Generator performs the specified requirements for all statements and operators of the SMALL language.

4.1 Background

In order to provide background about hardware implementation of the SMALL program, this section introduces some basic knowledge of the VHSIC Hardware Description Language (VHDL) and Field Programmable Gate Arrays. The design procedure environment is also discussed.

4.1.1 VHDL: VHSIC Hardware Description Language

VHDL was developed by the Very High Speed Integrated Circuits (VHSIC) Program Office of the Department of Defense for use as a standard language in the microelectronics community during 1980's (Ott and Wilderotter, 1994). VHDL is a powerful language and can be used in many ways. In this study, VHDL is used for describing the circuit created by the Netlist Generator. The IEEE std_logic_1164 package is chosen because it is used in most new VHDL synthesis tools.

A structural description of a system contains the components and their interconnections by signals. Compared with behavioural descriptions, the structural description is more concrete and easier to synthesise into hardware. An example of the structural VHDL description has been given in Section 3.9. It is the netlist output of the program written in the SMALL language in Figure 3.1 and will be simulated and synthesised in later sections. The structural VHDL description includes an overall library statement, a "use" statement, an entity declaration, signal declarations, component declarations, and port map associations. The "library" specification is used in Synopsys simulation and synthesis. An entity declaration includes the name of each port, the

direction of its data flow, and the data type. A component declaration contains the name of the component and its ports. For each port, its name, data flow direction, and data type are declared. A signal declaration consists of the name and the data type of the signal. The body of the VHDL description starts with the "begin" keyword. The component instantiation statement is the basic unit in the description. Each instantiation is composed of a label, the name of the component and a mapping between the port names in the instantiation and the actual component. The signals used in the port map are the declared ports and declared signals.

In this study, VHDL is also used to describe a test bench file that provides a simple method for testing circuits. Many concurrent statements and signal assignment statements appear in the test bench file for providing stimuli to some input ports of the structural VHDL description output produced by the Netlist Generator.

4.1.2 Field Programmable Gate Arrays

A Field Programmable Gate Array (FPGA) is a programmable and reconfigurable logic device developed for rapid prototyping and implementation of digital systems in mid-1980s. Due to the short time-to-market and the low manufacturing cost, FPGAs are now very popular in hardware design. FPGAs are composed of programmable functional blocks surrounded by a programmable interconnection network. The user can customize the function of each block and its connections. The size, structure, number of blocks, and the interconnection are different among FPGA architectures. This difference in

architectures is determined by different programming technologies, different software, and different models of use (Trimberger, 1994).

Xilinx families of FPGAs are the most widely used FPGA families due to their features. The first commercially used FPGA series, the Xilinx XC2000, was introduced in 1985. Now more Xilinx generations such as XC3000, XC4000, and XC5000 exist. Although there are many variations on FPGA design, all the Xilinx FPGA families are composed of three main configurable components: an array of cells called configurable logic blocks (CLBs), a surrounding rectangular ring of input/output blocks (IOBs), and programmable interconnect resources. For each family member in a generation, the CLB array size is different. For example, XC4002A has 8×8 CLB array size and the CLB array size of XC4013 is 24×24. In the hardware implementation of the SMALL language, the XC4000 (Xilinx, 1997a) architecture is used as the target technology. As Xilinx's third-generation, the XC4000 family FPGAs are used as the programmable logic devices based on static memory resources. It is a popular and advanced technology for large design and for using placement and routing tools. Based on Oldfield and Dorf (1995) and Trimberger (1994), this section simply describes the Xilinx XC4000 architecture.

Figure 4.1 (Oldfield and Dorf, 1995) shows a simplified block diagram of the XC4000 CLB. It includes three combinational function generators (G', F', H'), two flip-flops, and their interconnect control logic. In Figure 4.1, F_1, F_2, F_3, F_4, G_1, G_2, G_3, G_4, C_1, C_2, C_3, C_4, and clock are the thirteen CLB inputs, and the four CLB outputs are XQ, X, YQ, Y. Each of the two main lookup-table-based function generators (F' and G') has four independent inputs. A third function generator (H') has three inputs (F', G', H_1) from the

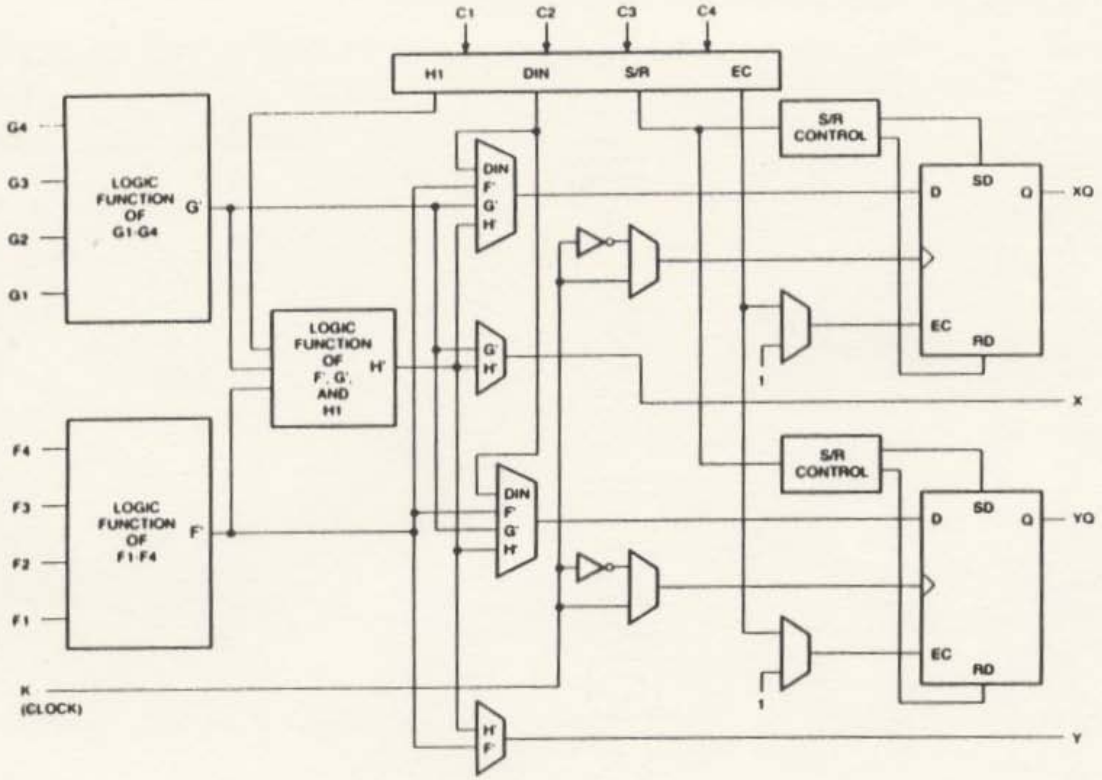


Figure 4.1: A Simplified Block Diagram of the XC4000 CLB

outputs of F' and G' and from outside the CLB. The two D-type flip-flops with the same clock (K) and clock enable (EC) inputs, a third common input (S/R) can get their inputs from the function generators or from external signals. The F' and G' lookup tables in each CLB can be configured as a 16×2 or 32×1 bits of memory cells. The flexibility and symmetry of the CLB architecture helps in placement and routing, during which inputs, outputs, and the functions can arbitrarily exchange positions within a CLB.

A block diagram of the XC4000 input/output block (IOB) is represented in Figure 4.2 (Trimberger, 1994). The IOB acts as the interface between external package pins and the

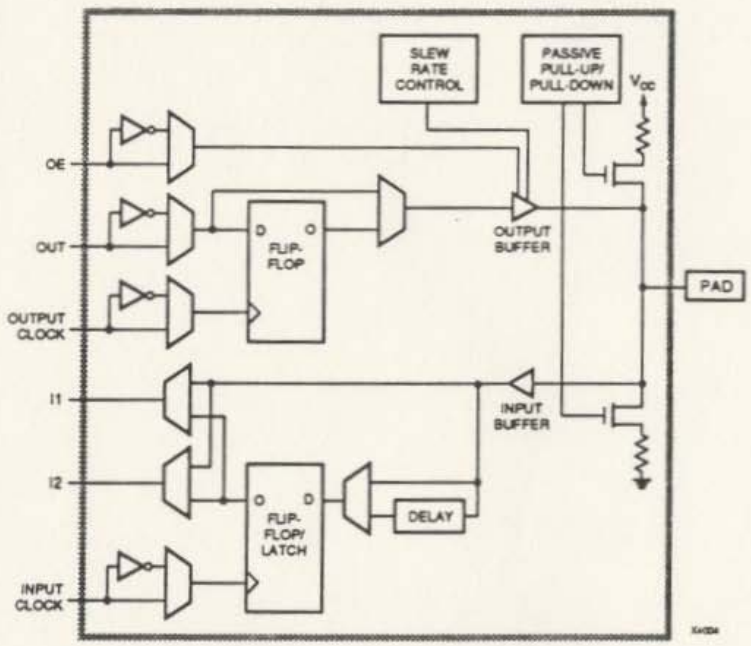


Figure 4.2: The XC4000 IOB

internal logic. Each IOB is used for one package pin. It can be arranged for input, output, or bi-directional signals. The XC4000 IOB contains test structure for testing internal logic or external logic. A useful function in the XC4000 IOB is that a master three-state control places all the I/O blocks in a high impedance mode when it is active.

Figure 4.3 (Trimberger, 1994) represents the XC4000 interconnect. There are many routing resources in the interconnect of any two points on the chip. Three kinds of interconnect, named as single-length lines, double-length lines, and long lines, are determined by the relative length of their metal segments. The single-length lines are a grid of horizontal and vertical lines at switchbox between each block. Double-length lines travel two CLBs before going to a switchbox. Long lines can be broken in the centre of the chip to give two half-long lines for better routability.

4.1.3 The Design Procedure and Environment

The entire process of compiling a SMALL program to an FPGA chip is divided into the following phases.

- Using the SMALL Simulator, an optional simulation of the SMALL program can be obtained.
- The SMALL program is compiled by the SMALL compiler to obtain a structural VHDL description that uses the IEEE std_logic_1164 package.
- The functional simulation of the structural VHDL description is carried out with Synopsys VSS to verify that the netlist circuit created by the Netlist Generator performs the specified requirements.

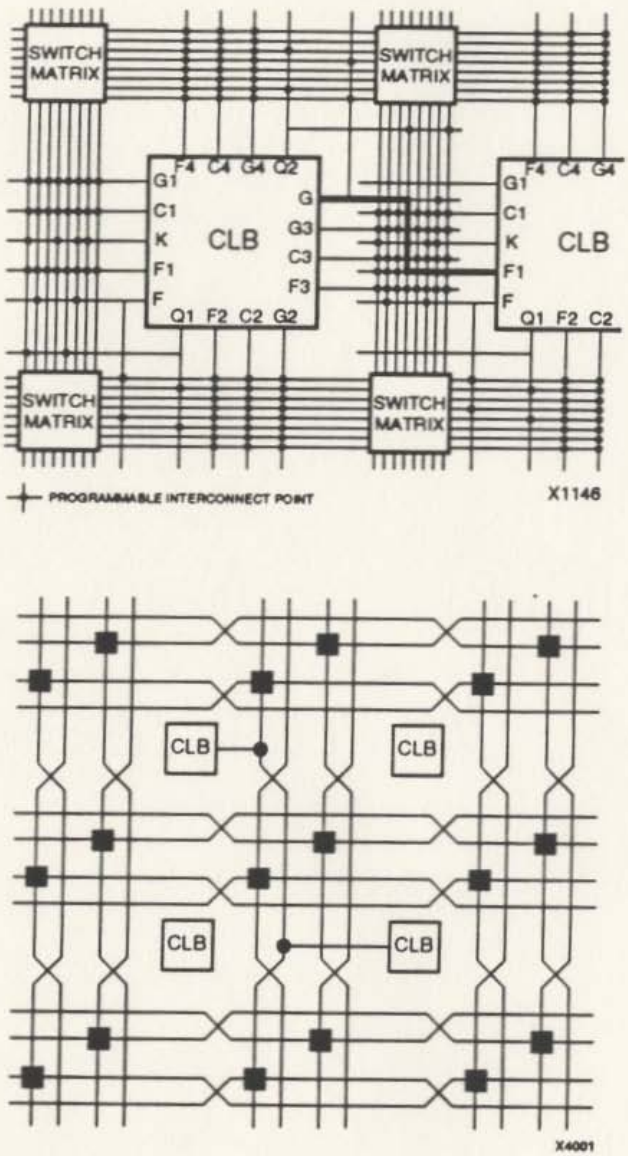


Figure 4.3: The XC4000 Interconnect

- Synopsys FPGA Compiler (FC) is used for the gate-level synthesis and logic optimization.
- The configuration data for a specific Xilinx FPGA chip, XC4028EX-3-PG299, is produced by the Design Manager of the Xilinx Alliance Series version 1.4.

4.2 Simulation with the Structural VHDL Description

This section shows the functional simulation results for several SMALL programs. The components used in the structural VHDL files of the SMALL programs and the test bench files for the simulations are also described.

4.2.1 Components

The components used in the architecture of the structural VHDL description include and-gates, or-gates, and xor-gates that have two-inputs and one-output, inverters and buffers that have one-input and one-output, and D-type flip-flops. Their VHDL descriptions are as follows.

- For an and-gate, the name of the component is AND2 and the port names are I1, I2, and O1. The following is the VHDL description of the and-gate.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity AND2 is
     port(I1, I2 : in std_logic;
          O1 : out std_logic);
end AND2;
```

```
architecture STRUCTURAL of AND2 is

begin
  O1 <= I1 and I2;
end STRUCTURAL;
```

- For an or-gate, the name of the component is OR2 and the port names are I1, I2, and O1. The VHDL description of the or-gate is similar to that for the and-gate.
- For an xor-gate, the name of the component is XOR2 and the port names are I1, I2, and O1. The VHDL description of the xor-gate is also similar to that for the and-gate.
- For a buffer, the name of the component is BUF and the port names are I1 and O1. The following is the VHDL description of the buffer.

```
 library IEEE;
use IEEE.std_logic_1164.all;

entity BUF is
      port( I1 : in std_logic;
            O1 : out std_logic);
end BUF;

architecture STRUCTURAL of BUF is

begin
  O1 <= I1;
end STRUCTURAL;
```

- For an inverter, the name of the component is INVERTER and the port names are I1 and O1. The VHDL description of the inverter is similar to that for the buffer.
- For a D-type flip-flop, the name of the component is DFLIPFLOP and the port names are clock, D, and Q. The clock of each D-type flip-flop in the netlist is the same. All the D-type flip-flops used in the netlist are defined as the falling-edge triggered flip-

flops. It is noted that when the input of the D-type flip-flop is 'U', its output is '0'. This is because the output of the D-type flip-flop used in the initial state node circuit is required to be false in the first clock period and to be true in the following clock periods. For this purpose, the D input of this D-type flip-flop is named d_input and defined as an input port in the VHDL output of the SMALL program. In the test bench, the d_input signal is set up with 'U' at the beginning and then becomes true before the second clock period and keeps true in the following clock periods. The following is the VHDL description of the D-type flip-flop.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DFLIPFLOP is
     port(clock, D : in std_logic;
          Q : out std_logic);
end DFLIPFLOP;

architecture STRUCTURAL of DFLIPFLOP is

 signal internal_state : std_logic;

begin

 FLIPFLOP : process(D,clock)

        begin

         if (D='U') then
           Q<= '0';
         elsif ( clock = '0' and not clock'stable) then

            Q <= D;
         end if;

        end process FLIPFLOP;

end STRUCTURAL;
```

4.2.2 Test Benches

Douglas and Thomas (1994) have defined the test bench as "the name given to the VHDL code that generates the input signals for the design being simulated and monitors the desired outputs." When simulating the VHDL output of the Netlist Generator, the VHDL description of the test bench is created to give test inputs to the netlist circuit. The test bench file is not synthesised. Figure 4.4 shows the VHDL description of the test bench used in simulating the VHDL output description that is shown in Figure 3.22. It mainly consists of entity declarations, component declarations, signal declarations, the statement part, and the configuration declaration. The entity name is declared as SMALL_TEST1 and no ports exist for this test bench. TEST is the identifier of the architecture. SMALL_OUT1 is the component name and is the same as the entity name shown in Figure 3.22. This component and that entity have the same ports. The signal declaration includes the signal name and the data type. For some signals, the signal initializations are also used. In this example, the power, ground, and clock signals are initialized to '1', '0', '0', respectively. The statement part includes concurrent statements that are composed of the instantiation of the SMALL_OUT1 component and signal assignment statements. The final part is the configuration declaration that configures the above part for testing the structural architecture of the SMALL_OUT1 (Navabi, 1993).

For each SMALL program, its simulation can be carried out in two ways. One is to use the parallel statement in the SMALL program for generating some input signals, as shown in Figure 3.1. In the corresponding test bench shown in Figure 4.4, we do not need to assign a value and a time to the input signal such as the assert_global_inBit signal.

```
library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_TEST1 is
end SMALL_TEST1;

architecture TEST of SMALL_TEST1 is

 component SMALL_OUT1
  port(assert_global_outBit : in std_logic;
       assert_global_inBit : in std_logic;
       d_input : in std_logic;
       clock : in std_logic;
       power : in std_logic;
       ground : in std_logic;
       global_outBit : out std_logic;
       global_inBit : out std_logic);

 end component;

signal assert_global_outBit: std_logic := '0';
signal assert_global_inBit: std_logic := '0';              //
signal global_outBit, global_inBit, d_input: std_logic;
signal ground: std_logic := '0';
signal clock: std_logic := '0';
signal power: std_logic := '1';
begin

AA1: SMALL_OUT1 port map (assert_global_outBit, assert_global_inBit, d_input,
clock, power, ground, global_outBit, global_inBit);
clock <= not clock after 10 ns;
d_input <= 'U', power after 5 ns;
--assert_global_inBit <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns,          //
--'1' after 80 ns, '0' after 100 ns, '1' after 120 ns;
end TEST;

configuration conf_SMALL_1 of SMALL_TEST1 is
 for TEST
 end for;
end conf_SMALL_1;
```

Figure 4.4: The Test Bench for the VHDL Output Shown in Figure 3.22

However, its circuit representation generated by the Netlist Generator is more complicated, as shown in Figure 3.22. The other is not to use the parallel statement in the SMALL program for some input signals, as shown in Figure 4.5. Its structural VHDL description shown in Figure 4.6 is not so complicated as the VHDL description shown in Figure 3.22. However, its corresponding test bench shown in Figure 4.7 is little different from the test bench shown in Figure 4.4. The difference is shown by using // in Figure 4.4 and in Figure 4.7. The output signal such as the global_outBit is the same in the two ways.

4.2.3 Simulation Results

Using the Synopsys VHDL System Simulator, the circuit produced by the Netlist Generator is simulated to verify its functionality. This section shows the functional simulation results for several SMALL programs. In all the simulation results, the clock period used for the simulations is 20 ns. The D-type flip-flops in the netlist circuits are defined as falling-edge triggered flip-flops.

Example 1: A Parity Generator

The source SMALL program is shown in Figure 3.1 and in Figure 4.5, respectively. Figure 4.8 represents the simulation result of the program in Figure 3.1 with its VHDL output description shown in Figure 3.22 and its test bench shown in Figure 4.4. Figure 4.9 shows the simulation result of the program in Figure 4.5 with its VHDL output

description shown in Figure 4.6 and its test bench shown in Figure 4.7. As described in Section 4.2.2, Figure 4.8 and Figure 4.9 are obtained in two ways for a parity generator. It can be seen that in both figures, the output named GLOBAL_OUTBIT is [0, 0, 1, 1, 0] if the input named GLOBAL_INBIT is [0, 1, 0, 1, 1] in the first five clock periods. This result is the same as the result that is obtained from the SMALL simulator, i.e. without the Netlist Generator. That is, at each falling-edge of the clock, the GLOBAL_OUTBIT is the parity of those GLOBAL_INBITs that have been seen in the previous clock periods (the parity of the empty sequence is 0) (Norvell, 1996).

```
global sig inBit: bool
global sig outBit: bool

while true do

     repeat
          outBit ! 0
     until inBit

     tick

     repeat
          outBit ! 1
     until inBit

od
```

Figure 4.5: A Parity Generator Written in the SMALL Language

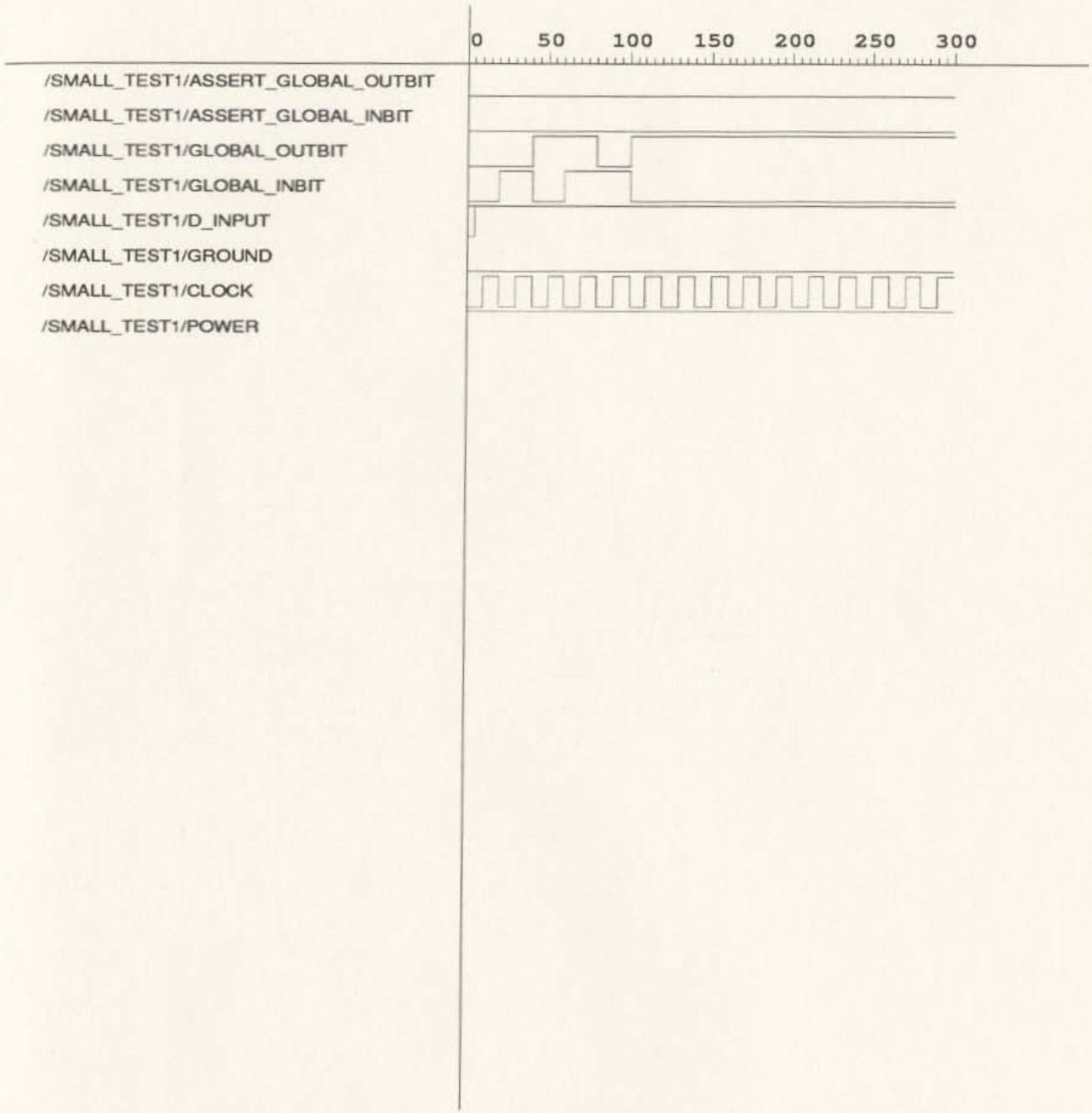
```
--The following is a VHDL source file representing the circuits stored
--in the Netlist store.

library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_OUT1 is
 port (assert_global_outBit : in std_logic;
      assert_global_inBit : in std_logic;
      d_input : in std_logic;
      clock : in std_logic;
      power : in std_logic;
      ground : in std_logic;
      global_outBit : out std_logic;
      global_inBit : out std_logic);

end SMALL_OUT1;

architecture Structure of SMALL_OUT1 is

 signal  wire96, wire79, wire78, wire75, wire73, wire71, wire69,
       wire68, wire67, wire66, wire61, wire59, wire57, wire55,
       wire53, wire51, wire50, wire49, wire48, wire43, wire42,
       wire39, wire38, wire35, wire33, wire31, wire29, wire27,
       wire25, wire24, wire23, wire22, wire17, wire16, wire13,
       wire12, wire7, wire2: std_logic;

component OR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component AND2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component BUF
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component INVERTER
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component DFLIPFLOP
 port (clock, D : in std_logic; Q : out std_logic);
```

```
 end component;
begin
  Device98: BUF port map ( assert_global_inBit, wire2);
  Device97: OR2 port map ( assert_global_outBit, wire96, wire7);
  Device95: OR2 port map ( wire33, wire59, wire96);
  Device94: BUF port map ( wire29, wire12);
  Device93: BUF port map ( wire27, wire16);
  Device92: BUF port map ( wire31, wire22);
  Device91: BUF port map ( wire25, wire27);
  Device90: BUF port map ( wire24, wire29);
  Device89: BUF port map ( wire35, wire31);
  Device88: BUF port map ( wire55, wire38);
  Device87: BUF port map ( wire53, wire42);
  Device86: BUF port map ( wire57, wire48);
  Device85: BUF port map ( wire51, wire53);
  Device84: BUF port map ( wire50, wire55);
  Device83: BUF port map ( wire61, wire57);
  Device82: BUF port map ( wire75, wire66);
  Device81: BUF port map ( wire69, wire71);
  Device80: BUF port map ( wire68, wire73);
  Device77: INVERTER port map ( wire78, wire79);
  Device76: DFLIPFLOP port map ( clock, d_input, wire78);
  Device74: OR2 port map ( wire79, wire13, wire75);
  Device65: AND2 port map ( wire67, wire66, wire69);
  Device64: AND2 port map ( wire66, power, wire68);
  Device63: INVERTER port map ( power, wire67);
  Device60: OR2 port map ( wire73, wire43, wire61);
  Device58: AND2 port map ( wire57, ground, wire59);
  Device47: AND2 port map ( wire49, wire48, wire51);
  Device46: AND2 port map ( wire48, wire2, wire50);
  Device45: INVERTER port map ( wire2, wire49);
  Device41: DFLIPFLOP port map ( clock, wire42, wire43);
  Device37: DFLIPFLOP port map ( clock, wire38, wire39);
  Device34: OR2 port map ( wire39, wire17, wire35);
  Device32: AND2 port map ( wire31, power, wire33);
  Device21: AND2 port map ( wire23, wire22, wire25);
  Device20: AND2 port map ( wire22, wire2, wire24);
  Device19: INVERTER port map ( wire2, wire23);
  Device15: DFLIPFLOP port map ( clock, wire16, wire17);
  Device11: DFLIPFLOP port map ( clock, wire12, wire13);
  Device9: BUF port map ( wire7, global_outBit);
  Device4: BUF port map ( wire2, global_inBit);
end Structure;
```

Figure 4.6: The Structural VHDL Output for a Parity Generator Shown in Figure 4.5

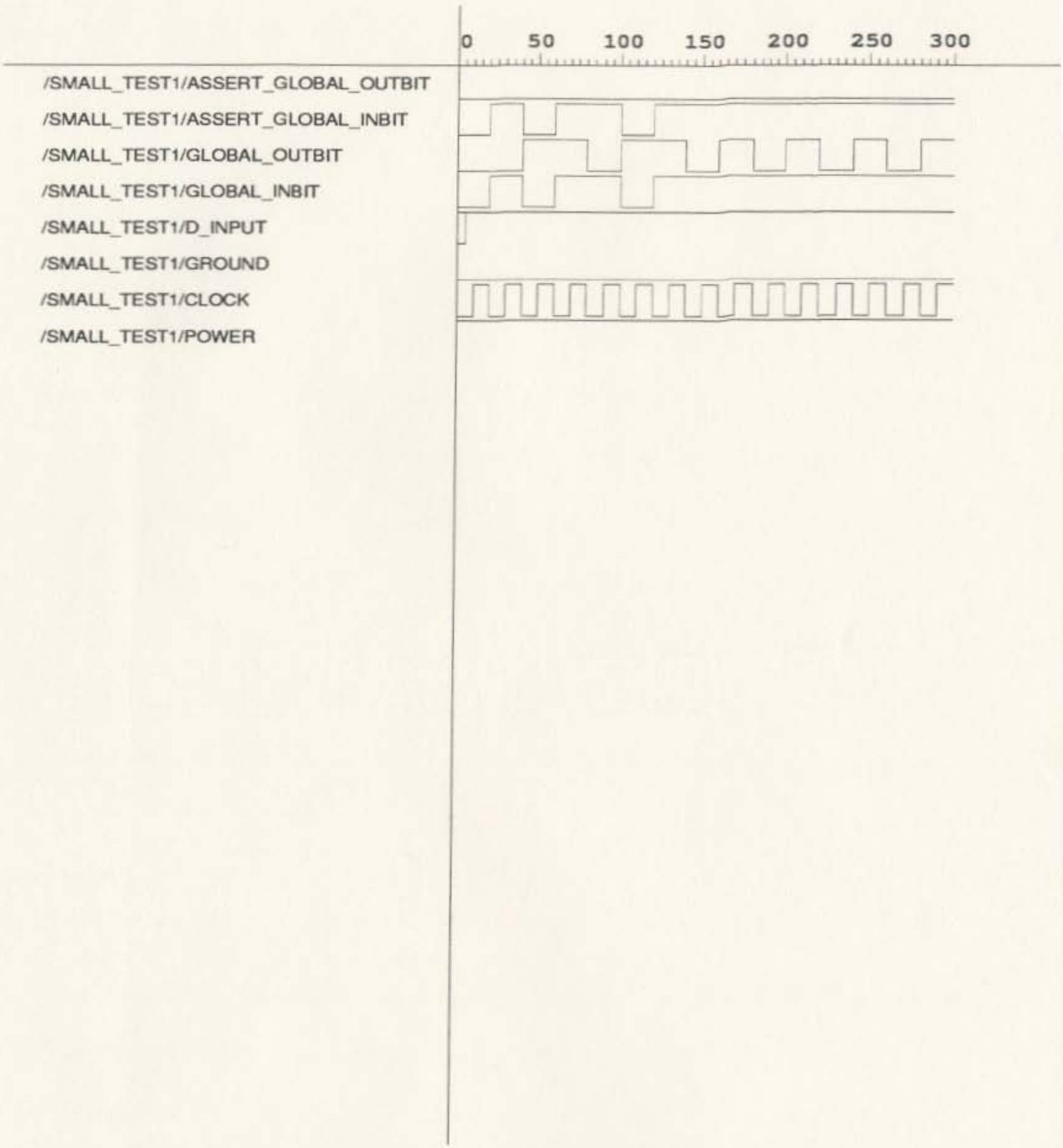
```
library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_TEST1 is
end SMALL_TEST1;

architecture TEST of SMALL_TEST1 is

 component SMALL_OUT1
  port(assert_global_outBit : in std_logic;
      assert_global_inBit : in std_logic;
      d_input : in std_logic;
      clock : in std_logic;
      power : in std_logic;
      ground : in std_logic;
      global_outBit : out std_logic;
      global_inBit : out std_logic);

 end component;

signal assert_global_outBit: std_logic := '0';
--signal assert_global_inBit: std_logic := '0';                    //
signal assert_global_inBit, global_outBit, global_inBit, d_input: std_logic;
signal ground: std_logic := '0';
signal clock: std_logic := '0';
signal power: std_logic := '1';
begin

AA1: SMALL_OUT1 port map (assert_global_outBit, assert_global_inBit, d_input,
clock, power, ground, global_outBit, global_inBit);
clock <= not clock after 10 ns;
d_input <= 'U', power after 5 ns;
assert_global_inBit <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns,
                       '1' after 80 ns, '0' after 100 ns, '1' after 120 ns;        //
end TEST;

configuration conf_SMALL_1 of SMALL_TEST1 is
 for TEST
 end for;
end conf_SMALL_1;
```

Figure 4.7: The Test Bench for the VHDL Output Shown in Figure 4.6



lsi2/mnt/projects/ying/cad/mosis0.5/XILINX/Synopsys/SMALL_TEST1.cougar.21
7/12/1998 13:22:44 Page 1,1 of 1,1

Figure 4.8: The Simulation Result of a Parity Generator Shown in Figure 3.1



lsi2/mnt/projects/ying/cad/mosis0.5/XILINX/Synopsys/SMALL_TEST1.cougar.21
7/12/1998 13:5:4 Page 1,1 of 1,1

Figure 4.9: The Simulation Result of a Parity Generator Shown in Figure 4.5

Example 2: A Sequential Multiplier

The sequential multiplier written in the SMALL language is shown in Figure 2.1. It implements the multiplication of two 4-bit numbers to produce an 8-bit product. The structural VHDL descriptions representing its netlist circuit and the test bench are given in Figure A.1 and in Figure A.2 of Appendix A, respectively. Figure 4.10 shows the simulation result when the multiplier and the multiplicand are 3 and 2 in the decimal system. The numbers displayed in Figure 4.10 are expressed in the hexadecimal system. For example, the multiplier named GLOBAL_MULTIPLIER and the product named GLOBAL_P are represented as C and 00 in the first clock period, respectively. After five clock periods, the product is reported as 60 in the hexadecimal system, i.e. [0, 1, 1, 0, 0, 0, 0, 0] as a binary array. Recalling that SMALL uses an lsb first encoding, this represents the number 6. Therefore, the output result of the Netlist Generator is verified.

Example 3: A Serial Adder

The serial adder written in the SMALL language is shown in Figure 4.11. It implements the addition of two numbers. The two statements shown in Figure 4.11, *d* <- *a* => *b* and *e* ! *a* <= *b*, are not useful for the adder. They are used to test the functions of the operators such as => and <=. The structural VHDL descriptions representing its netlist circuit and test bench are given in Figure B.1 and in Figure B.2 of Appendix B, respectively. Figure 4.12 shows the simulation result when the two additions named as

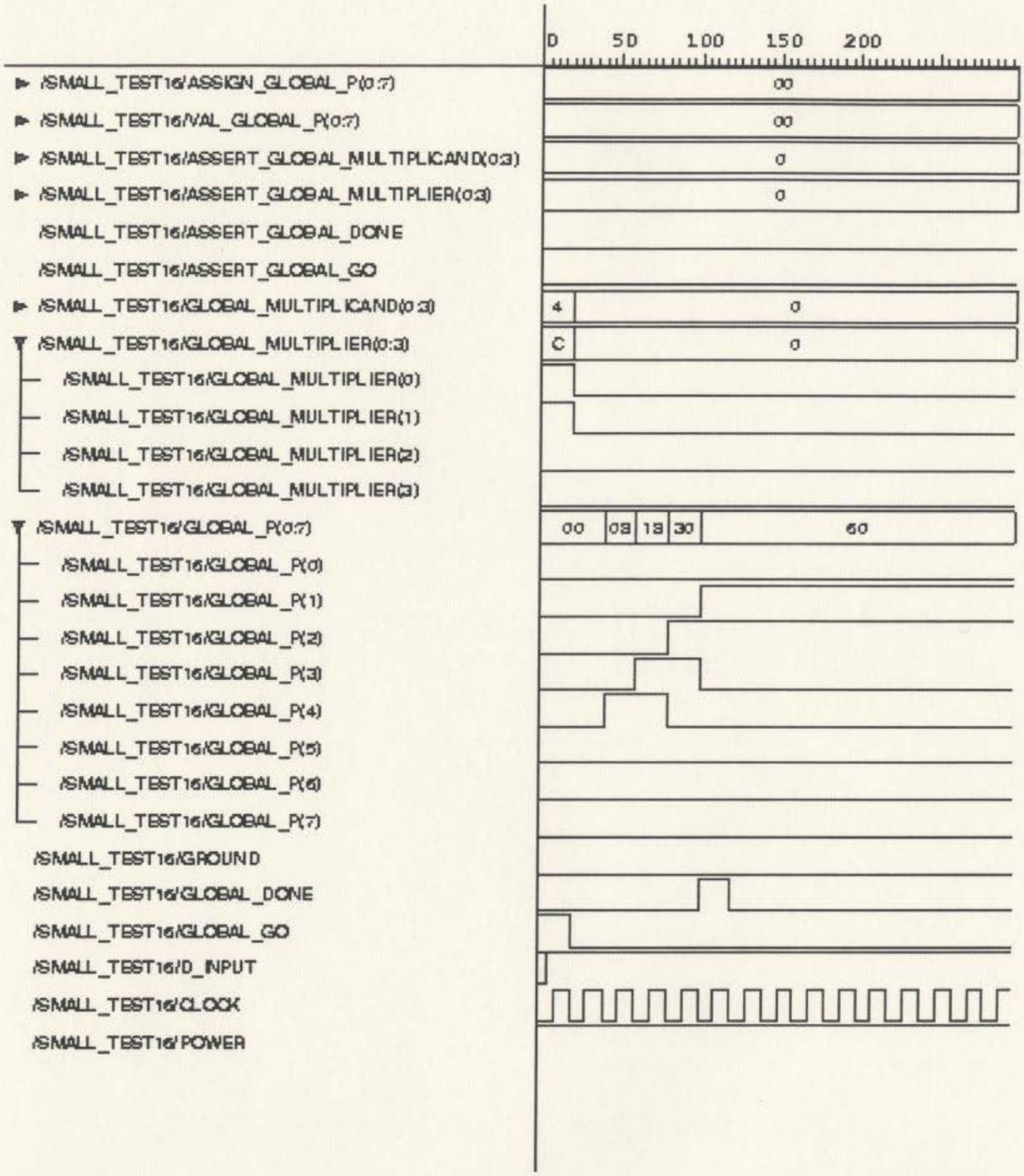


Figure 4.10: The Simulation Result of a Sequential Multiplier Shown in Figure 2.1

GLOBAL_A and GLOBAL_B are given as in the test bench shown in Figure B.2. The result can be verified using the SMALL simulator as shown in Figure 2.2.

```
global sig a : bool
global sig b : bool
global reg c : bool
global reg d : bool
global sig e : bool
global sig x : bool

while true do
        x ! a =/=b=/=c
        c <- a & b or a & c or b & c

        d <- a => b
        e ! a <= b

od
```

Figure 4.11: A Serial Adder Written in the SMALL Language

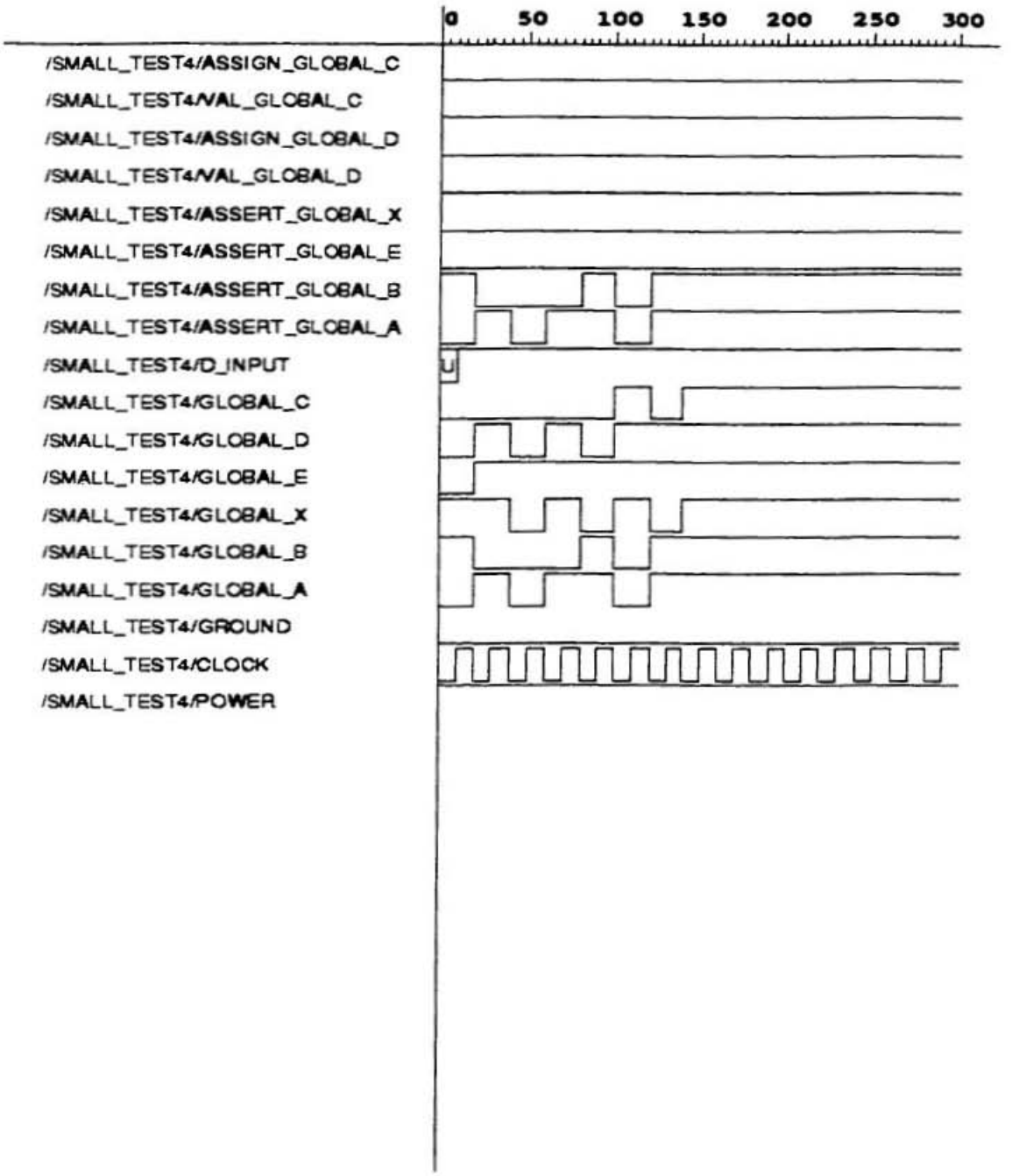


Figure 4.12: The Simulation Result of a Serial Adder Shown in Figure 4.11

Example 4: The Operation of Two-Dimensional Arrays

This example is used for verifying the operation of two-dimensional arrays. An example SMALL program is shown in Figure 4.13.

```
par

global signal a : array 2 of array 3 of bool
global signal b : array 2 of array 3 of bool
global signal c : array 2 of array 3 of bool
global signal d : bool
global register e : array 2 of array 3 of bool
global register f : array 2 of array 3 of bool

       c ! a & b
       d ! a /= b
       e <- a - b
       f  <- - c

       tick

||

global signal a : array 2 of array 3 of bool
global signal b : array 2 of array 3 of bool
global signal c : array 2 of array 3 of bool

       a[0] ! 5 as 3 bits
       a[1] ! 3 as 3 bits
       b[0] ! 5 as 3 bits
       b[1] ! 5 as 3 bits

rap
```

Figure 4.13: The SMALL Program for Example 4.

The structural VHDL description for the netlist generated from this example is too long to be shown in this thesis. The VHDL description for the user-defined data types in the example is given in Figure C.1. The VHDL description of the test bench for simulating the example is shown in Figure C.2. It is noted that in Figure C.1 two one-dimensional array types are declared instead of one two-dimensional array type. This is because some synthesis tools only support one-dimensional arrays (Naylor and Jones, 1997; Airiau *et al.*, 1994). Figure 4.14 represents the simulation result of the program shown in Figure 4.13. The result is confirmed using the SMALL simulator as shown in Figure 2.2.

Example 5: The Operation of Three-Dimensional Arrays

This example is used for verifying the operation of three-dimensional arrays. Its SMALL program and the VHDL descriptions for data types used in Example 5 and for the test bench are shown in Figure 4.15, in Figure D.1, and in Figure D.2 of Appendix D, respectively. For the same reason as Example 4, the structural VHDL description for Example 5 is not shown in this thesis. Figure 4.16 represents the simulation result of the program shown in Figure 4.15. Similar to Example 4, the VHDL description shown in Figure D.1 is used to declare all the user-defined data types for the operation of the three-dimensional arrays in Example 5. Three one-dimensional array types are declared instead of one three-dimensional array type.

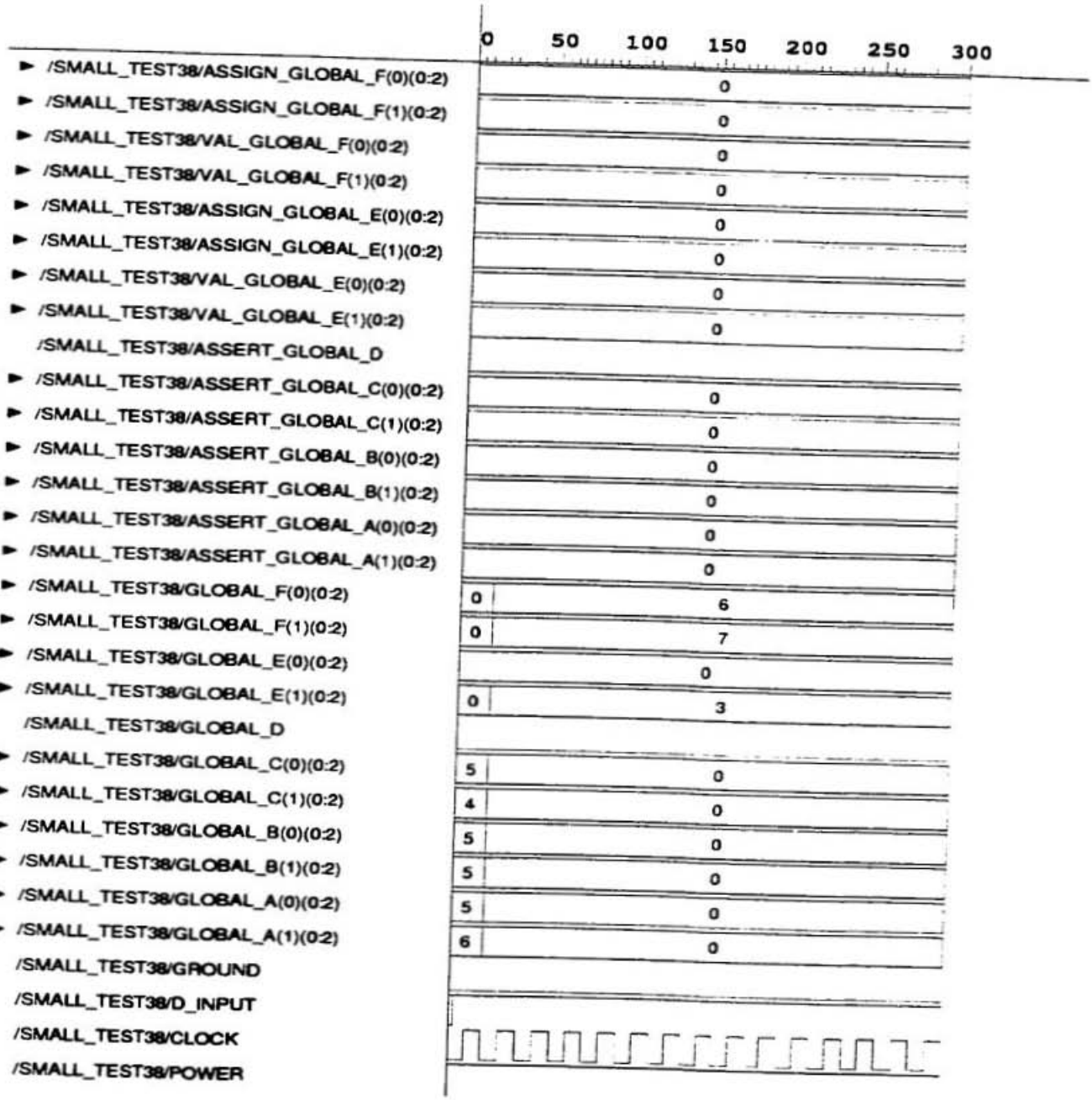


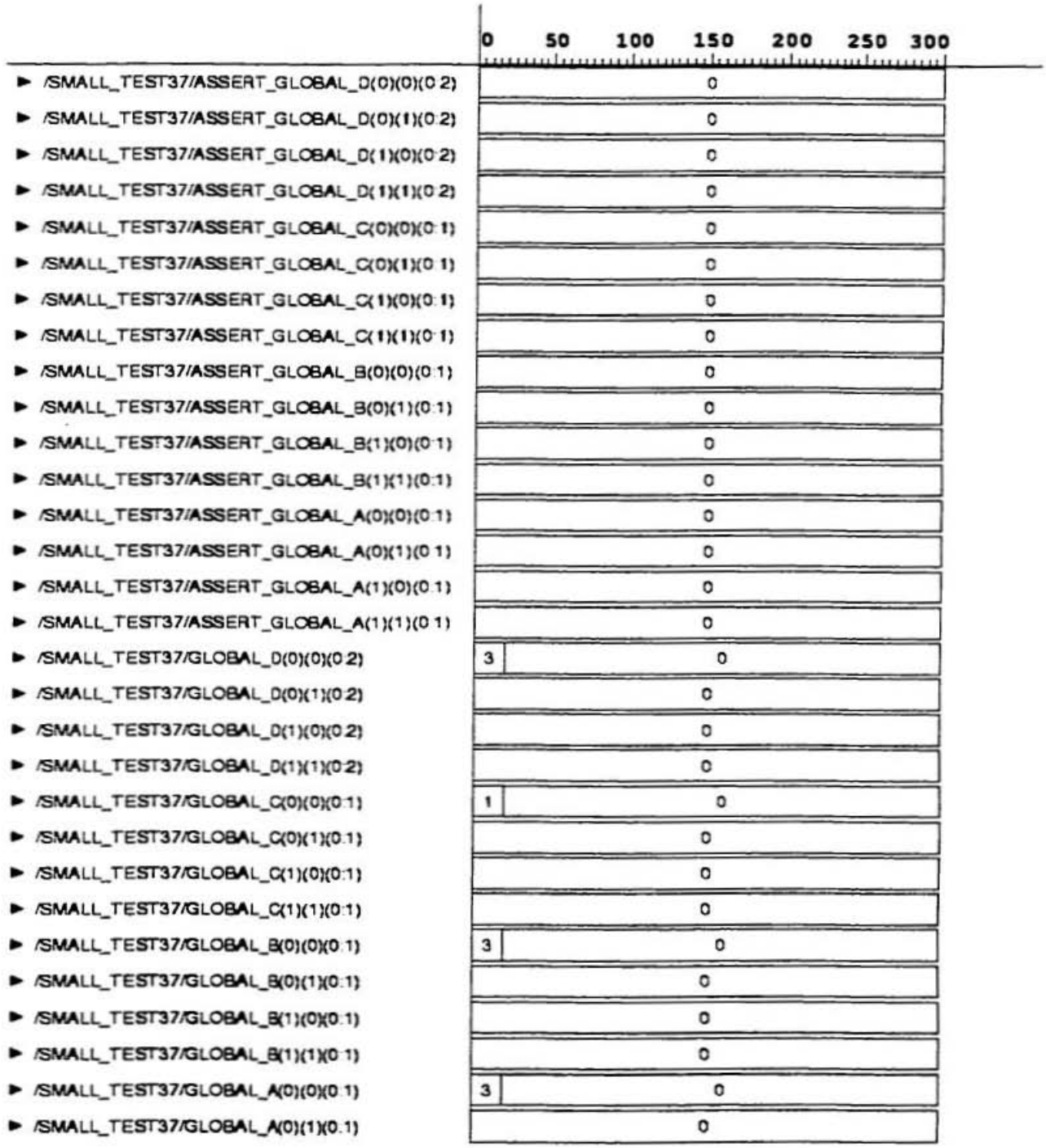
Figure 4.14: The Simulation Result of the SMALL Program Shown in Figure 4.13

```
par

global signal a : array 2 of array 2 of array 2 of bool
global signal b : array 2 of array 2 of array 2 of bool
global signal c : array 2 of array 2 of array 2 of bool
global signal d : array 2 of array 2 of array 3 of bool
global register e : array 2 of array 2 of array 3 of bool

     c ! a plus b
     d ! a uplus b

||

global signal a : array 2 of array 2 of array 2 of bool
global signal b : array 2 of array 2 of array 2 of bool
global signal c : array 2 of array 2 of array 2 of bool

     a[0][0] ! 3 as 2 bits
     b[0][0] ! 3 as 2 bits

rap
```

Figure 4.15: The SMALL Program for Example 5

Example 6: The SMALL Program in Pattern Matching

This example SMALL program, as shown in Figure 4.17, is applied to a simple pattern matching problem. In the example, a 1 by 3 image and a 1 by 2 pattern are used for simulation. The problem size is quite small because of limitations in the current version of the SMALL language, discussed in Chapter 5. The VHDL descriptions for the netlist circuit and for the test bench of Example 6 are shown in Figure E.1 and in Figure E.2 of Appendix E, respectively. Figure 4.18 represents the simulation result of Example 6. The image named GLOBAL_IMAGE(0:2) is chosen as [0, 1, 0] and three kinds of



lsi2/mnt/projects/ying/cad/mosis0.5/XILINX/Synopsys/SMALL_TEST37.cougar.1

5/12/1998 19:34:47 Page 1,1 of 2,1

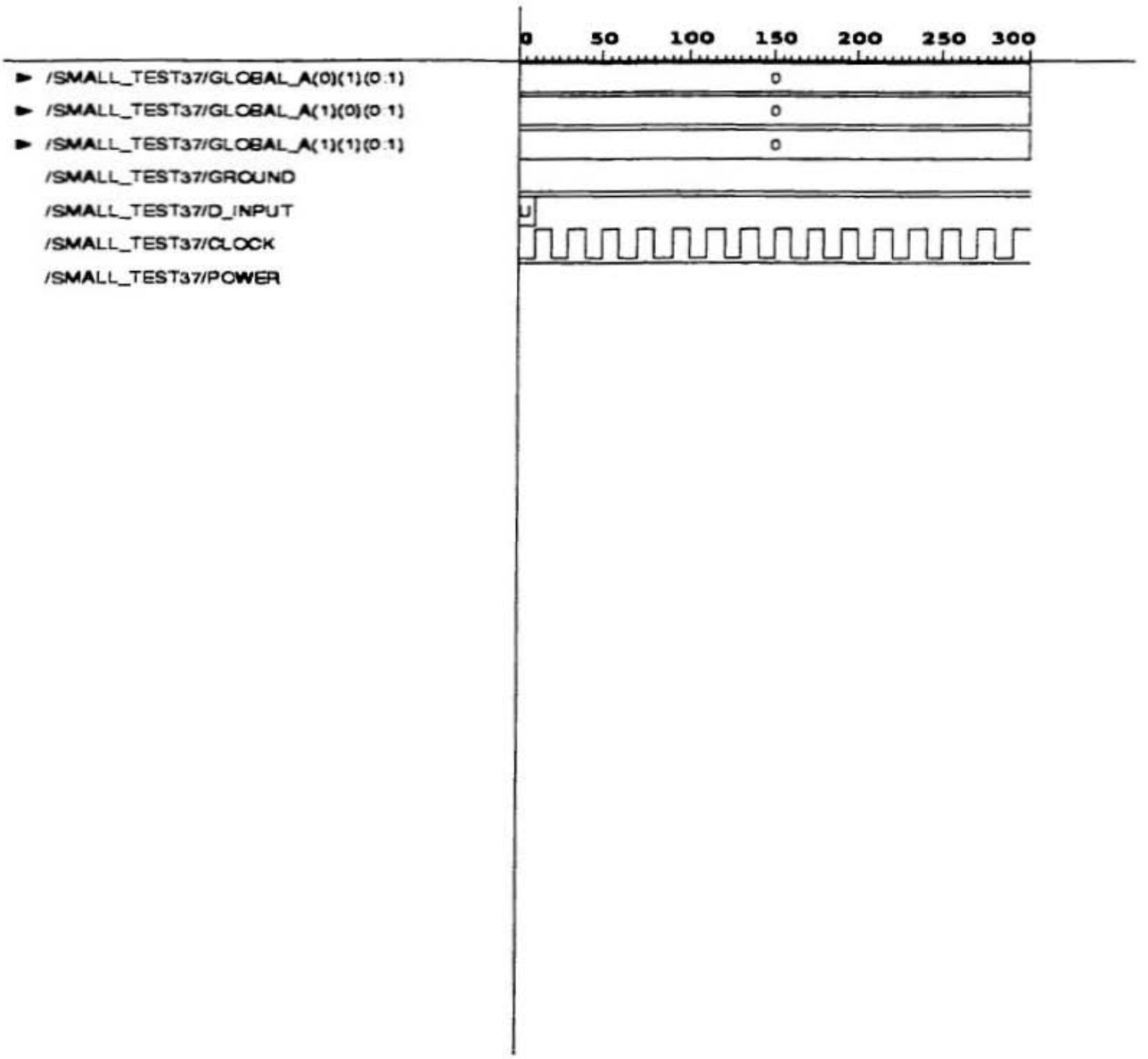


Figure 4.16: The Simulation Result of the SMALL Program Shown in Figure 4.15

patterns are tested. In Figure 4.18, when the pattern named GLOBAL_PATTERN(0:1) is [1, 1], the test result named GLOBAL_FOUND(0:1) is [0, 0]. This means the pattern is not found. When the pattern named GLOBAL_PATTERN(0:1) is [0, 1], the test result named GLOBAL_FOUND(0:1) is [1, 0]. This means the pattern is found at location 0. When the pattern named GLOBAL_PATTERN(0:1) is [1, 0], the test result named GLOBAL_FOUND(0:1) is [0, 1]. This means the pattern is found at location 1.

```
global signal image:array 3 of bool
global signal pattern : array 2 of bool
global signal found : array 2 of bool
global signal go : bool
global signal done : bool

par

  while true do

    while not go do skip od

      par          //Sequential search at location 0.

        if pattern[0] == image[0]

        then tick

          if pattern[1] == image[1]
          then found[0] ! true
          fi

        fi

      ||      //Sequential search at location 1.

        if pattern[0] == image[1]

        then tick
```

```
                if pattern[1] == image[2]
                then found[1] ! true
                fi
            fi

        rap

        done ! true

    od

||      //test code goes here.

    repeat
            image ! [0, 1, 0]
            pattern ! [1, 1]
            go ! true
    until done

    tick

    repeat
            image ! [0, 1, 0]
            pattern ! [0, 1]
            go ! true
    until done

    tick

    repeat
            image ! [0, 1, 0]
            pattern ! [1, 0]
            go ! true
    until done

    tick

rap
```

Figure 4.17: The SMALL Program Used in Pattern Matching

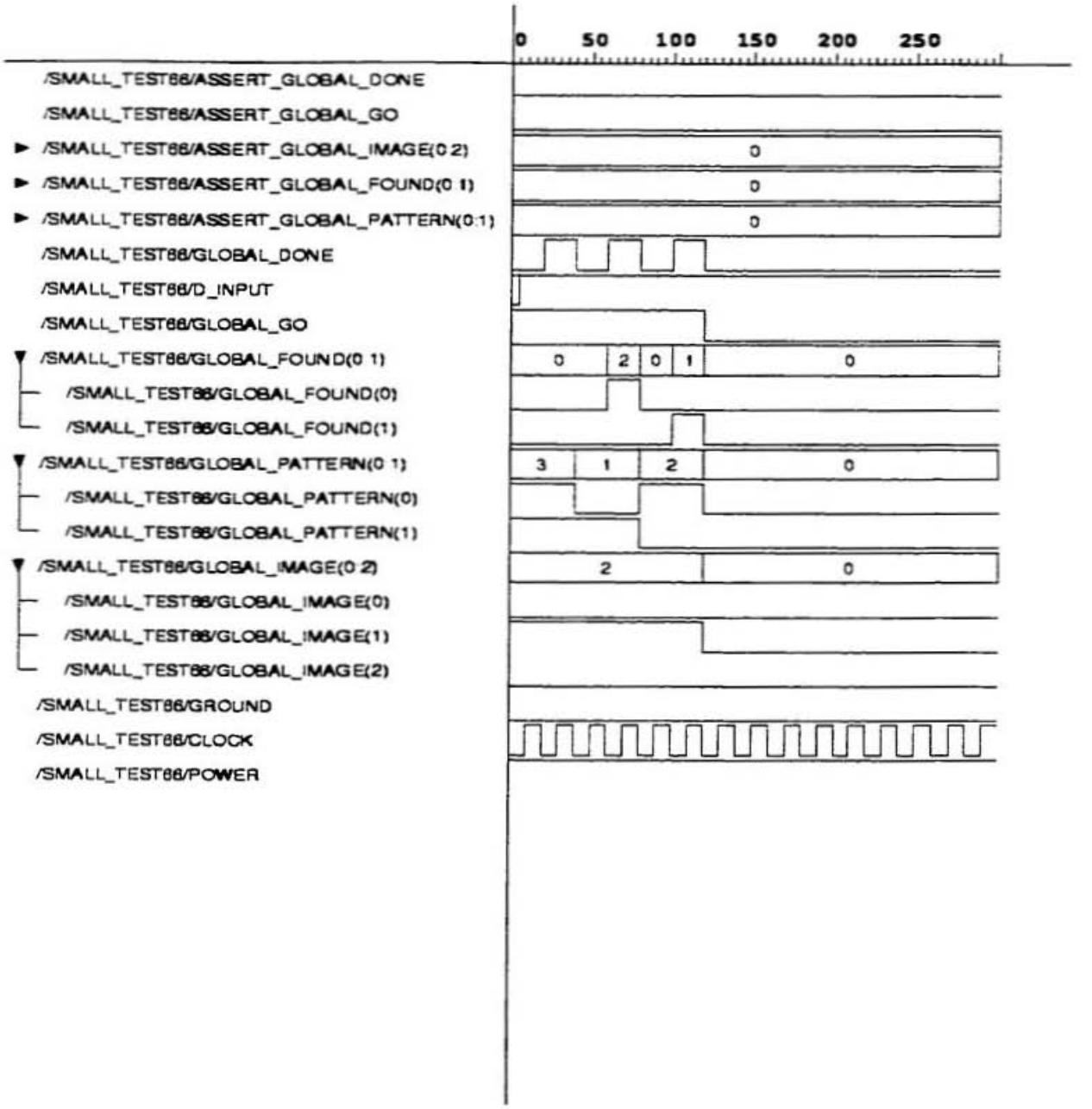


Figure 4.18: The Simulation Result of the SMALL Program Shown in Figure 4.17

The above examples are simple applications of the SMALL language. After the SMALL language enhancements and the Netlist Generator improvements, the SMALL language will be widely used in many applications.

4.3 Gate-Level Synthesis

The Synopsys FPGA Compiler is used for the gate-level synthesis. The synthesised gate-level design will be saved as an .sxnf file that is imported into the Design Manager of the Xilinx Alliance Series version 1.4 for place and route. Figure 4.19 and Figure 4.20 represent the synthesis results of the SMALL programs shown in Figure 3.1 and in Figure 2.1, respectively. In Figure 4.19, the circuit for a parity generator consists of twelve D-type flip-flops and other simple gates. In Figure 4.20, the circuit for a sequential multiplier is more complicated and can not be clearly seen because it is composed of hundreds of gates and tens of D-type flip-flops.

4.4 Hardware Implementation with FPGA

The Design Manager of the Xilinx Alliance Series version 1.4 is used to create configuration data for Xilinx FPGA chips. A Xilinx XC4028EX-3-PG299 FPGA is chosen as the target device. The SMALL program for a sequential multiplier is implemented with the above FPGA. The implementation process is composed of translating the design netlist, mapping the logic to CLBs, assigning all logic blocks to specific locations and interconnect elements on a die, and creating a configuration file that is used to program the FPGA.

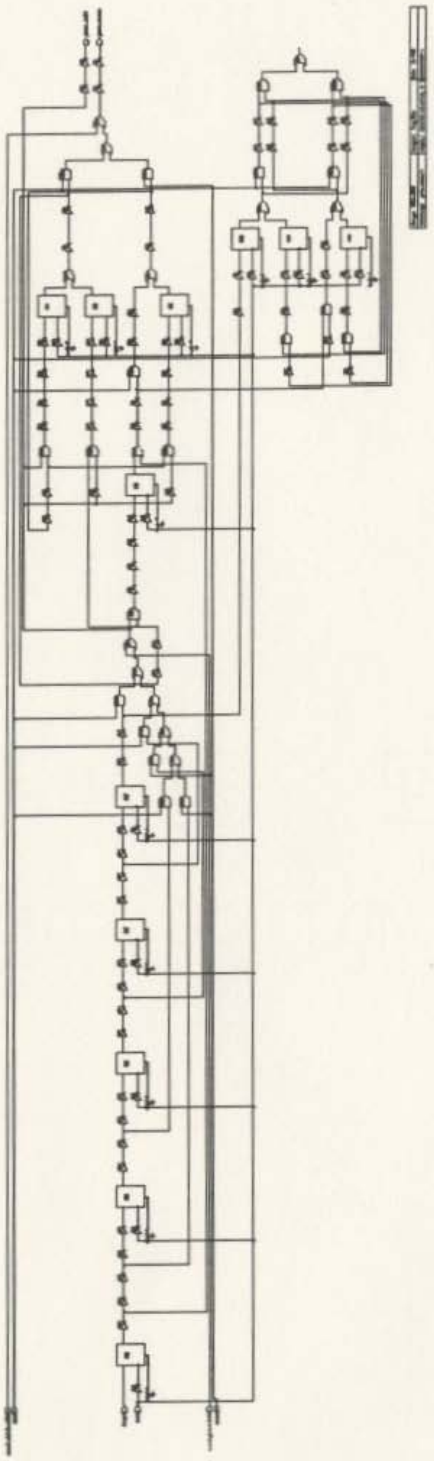


Figure 4.19: The Synthesis Result of the SMALL Program Shown in Figure 3.1

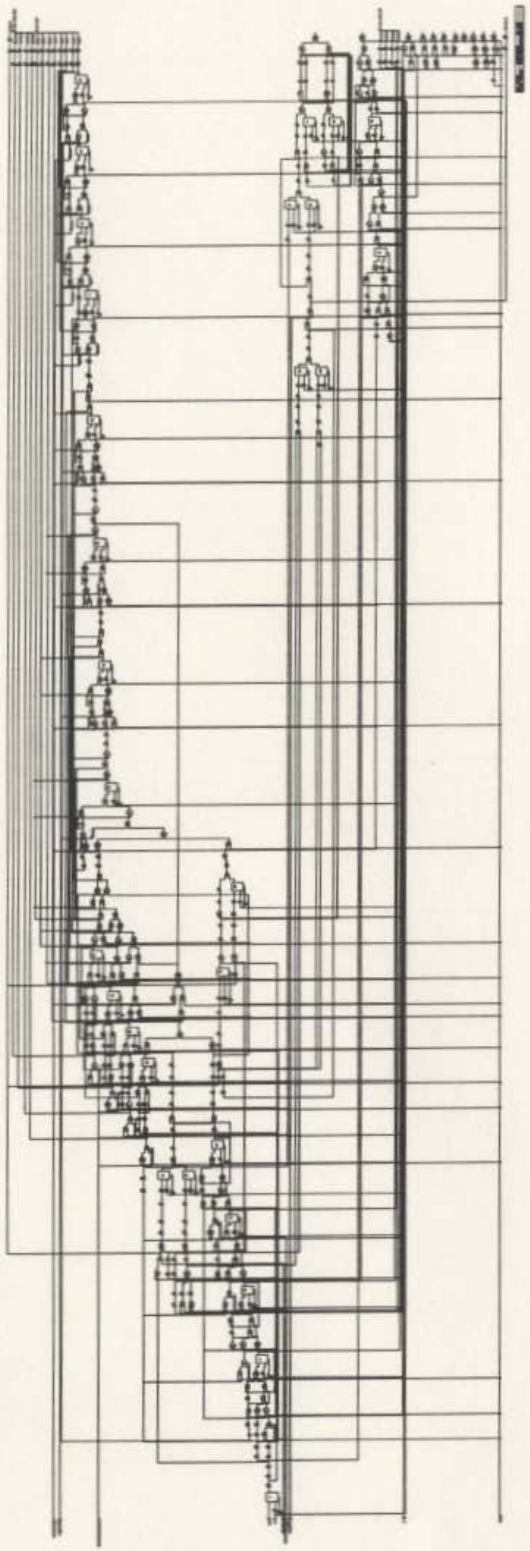


Figure 4.20: The Synthesis Result of the SMALL Program Shown in Figure 2.1

For the design named SMALL_OUT16, which is the structural VHDL output description of a sequential multiplier, the pre-placement and routing Synopsys area and timing reports show the following information:

Xilinx FPGA Design Statistics

FG Function Generators:	437
H Function Generators:	0
Number of CLB cells:	465
Number of Hard Macros and Other Cells:	0
Number of CLBs in Other Cells:	0
Total Number of CLBs:	465
Number of Ports:	48
Number of Clock Pads:	1
Number of IOBs:	47
Number of Flip Flops:	29
Number of 3-State Buffers:	0
Total Number of Cells:	513

The post-layout timing report file includes the following design information:

Design statistics:

Minimum period: 107.893ns (Maximum frequency: 9.268MHz)
Maximum net delay: 91.205ns

Chapter 5

Conclusions

Hardware compilation is a technique that allows the hardware design to be implemented by a purely software process. This approach is attracting increasing interest since it provides the designer the flexibility of introducing design changes and improvements, after implementation, when an FPGA chip is used. This thesis has presented the Netlist Generator and hardware implementation of a novel hardware description language SMALL. It has described the entire process for compiling the SMALL program into a Field-Programmable Gate Array.

The input to the Netlist Generator is a parallel ASM chart of a SMALL program. It is obtained from the SMALL program by a front-end that has been implemented by Norvell. The output of the Netlist Generator is a structural VHDL description. The function of the Netlist Generator is to transform the ASM chart to the structural VHDL description that represents the circuit stored in the netlist.

The algorithm of the transformations from an ASM chart to a netlist is simple.

Currently, the one-hot encoding technique is applied. When new statement forms are added to the SMALL language, the netlist generation method described in this thesis is not changed if the statement is transformed into one of the five kinds of nodes.

The netlist created by the Netlist Generator is also very simple. Its components consist of only D-type flip-flops and basic gates including two input and-gates, two input or-gates, two input xor-gates, inverters and buffers. All D-type flip-flops use the same clock line. The detailed circuit stored in the netlist is provided in this thesis.

Using the Synopsys VHDL System Simulation, the VHDL description for the netlist is simulated with a test bench to verify the functionality of the Netlist Generator. The simulation results for several SMALL programs show that the Netlist Generator performs the specified requirements for all the statements and all the operators in the SMALL language. For multi-dimensional arrays in the SMALL language, several one-dimensional array types are declared instead of one multi-dimensional array type. The gate-level synthesis and the place-and-route with a specific FPGA chip make designs expressed in the SMALL language easily implemented using FPGAs.

The Xilinx XC4000 series FPGAs are chosen as the target technology of the compilation process since these high-capacity programmable logic devices offer rapid prototyping and implementation of digital systems with the low manufacturing cost. An alternate technology could be used to implement the netlist created by the Netlist Generator. It includes full custom designs, standard cell designs, gate array designs, and ASICs, etc. VHDL is used for the target language. The structural VHDL description

representing the resulting the circuit stored in the netlist is simulated to verify that the Netlist Generator performs the specified requirement. It should be noted that the compiler's netlist is not dependent on the target language.

One current problem is that the SMALL compiler developed in Gofer in this thesis needs too much space and too much time. In order to reduce the compiling time, a compiled functional language, Haskell, is recommended to implement the SMALL compiler. Moreover, the change of the type constructor Assoc in the util module from a list of pairs to a balanced binary tree will save more compiling time and space.

Since the SMALL language is missing useful constructs such as modules and a parallel 'for' construct, only simple applications of the SMALL language are currently implemented with FPGA chips. For a large and more complicated application, language enhancements need to be carried out. For example, data types such as integer types, floating point types, enumerated types, and structures may be added to the current data types. In addition, computed array indices, more arithmetic operators, and more statements, including a parallel 'for' statement, are also desirable. The Netlist Generator will have some changes with the language enhancements. The compilation technique for these extensions and a proof of the compilation scheme should be developed.

Some optimisation opportunities exist. For the netlist created by the Netlist Generator, simplifying some gates will make the circuit more optimal and efficient. For example, when one of two inputs of an and-gate is false, the output of the and-gate is false so that a false wire can be used to represent the and-gate. When implementing the SMALL

programs with FPGAs, the Synopsys FPGA Compiler provides the capability to optimise the circuit for area and speed.

The Netlist Generator proposed in this thesis performs the specified requirements for the SMALL language. Its development and the hardware implementation of the SMALL language have offered a significant advance on the original SMALL implementation. It is believed that the SMALL language will be widely used in many areas in the future because of its simplicity and simple semantics, especially after the SMALL language enhancements and further improvements to the Netlist Generator.

REFERENCES

Airiau, R., Bergé, J.-M., and Olive, V. (1994). *Circuit Synthesis with VHDL*. Kluwer Academic Publishers, pp. 52-53.

Berry, G. (1995). *The Constructive Semantics of Pure Esterel*. Draft Version 2.0.

Berry, G. and Gonthier, G. (1992). " The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming, 19:87-152.*

Bird, R. (1998). *Introduction to Functional Programming using Haskell*, 2nd edition. Prentice Hall Press.

Cunningham, H. C. (1995). "Notes on Functional Programming with Gofer, " Technical Report UMCIS-1995-01

Greaves, D. (1995). " The CSYN Verilog Compiler and Other Tools," *Field-Programmable Logic and Applications*, eds. W. Moore and W. Luk, Springer, pp. 198-207.

Gschwind, M., and Salapura, V. (1995). "A VHDL Design Methodology for FPGAs," *Field-Programmable Logic and Applications*, eds. W. Moore and W. Luk, Springer, pp. 208-217.

Guo, S., and Luk, W. (1995). "Compiling Ruby into FPGAs, " *Field-Programmable Logic and Applications*, eds. W. Moore and W. Luk, Springer.

He, Jifeng, Page, I., and Bowen, J. (1993). " Towards a Provably Correct Hardware Implementation of Occam," *Correct Hardware Design and Verification Methods*, eds. G. J. Milne, and L. Pierre, Springer-Verlag, pp. 214-225.

Hehner, E. C. R., Norvell, T. S., and Paige, R. F. (1998). "High-Level Circuit Design," to be published in *Acta Information*.

Jagadeesan, L. J., Puchol, C., and Olnhausen, J. E. (1995). "A Formal Approach to Reactive Systems Software: A Telecommunications Application in ESTEREL," *Workshop on Industrial-Strength Formal Specification Techniques*, pp. 132-144.

Jones, M. P. (1991). "An Introduction to Gofer Version 2.20." Gofer On-Line Document.

Jones, M. P. (1993). "Gofer-Functional Programming Environment, Version 2.28." Gofer On-Line Document.

Jones, M. P. (1994). "The Implementation of the Gofer Functional Programming System, " Research Report YALEU/DCS/RR-1030, Yale University, New Haven, Connecticut, USA.

Lenk, J. D. (1972). *Handbook of Logic Circuits*, Reston Publishing Company, Inc., Reston, Virginia, pp. 83-113.

Lipsett, R., Schaefer, C., and Ussery, C. (1989). *VHDL: Hardware Description and Design*, Kluwer Academic Publishers.

Martin, A. J. (1986). "Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, " Distributed Computing, pp.226-234.

Navabi, Z. (1993). *VHDL: Analysis and Modelling of Digital Systems*, McGraw-Hill, Inc.,

Naylor, D., and Jones, S. (1997). *VHDL: A Logic Synthesis Approach*, Chapman & Hall.

Norvell, T. S. (1996). *SMALL Hardware Description Language*, unpublished.

Norvell, T. S. (1997). " SMALL: A Programming Language for State Machine Design," *Canadian Conference for Electrical and Computer Engineering*.

Norvell, T. S. (1998). *Specification, and (ange List for the Netlist Generator*, unpublished.

Oldfield, J. V., and Dorf, R. C. (1995). *Field Programmable Gate Arrays*, John Wieley & Sons, Inc., New York.

Ott, D. E., and Wilderotter, T. J. (1994). *A Designer's Guide to VHDL Synthesis*, Kluwer Academic Publishers, pp. 47-50.

Page, I. (1996). "Constructing Hardware-Software System from a Single Description," *Journal of VLSI Signal Processing*, pp. 87-107.

Page, I., and Luk, W. (1991). " Compiling occam into Field-Programmable Gate Arrays," *Field-Programmable Gate Arrays*, eds. W. Moore, and W. Luk, Abingdon EE &CS Books, pp. 271-283.

Patterson, D. A. (1994). *Computer Organisation & Design: The Hardware/Software Interface,* Morgan Kaufmann Publishers, Inc., San Francisco, California, pp.184-194.

Rossen, L. (1990). "Ruby Algebra," *Designing Correct Circuits,* eds. G. Jones, and M. Sheeran, Springer-Verlag, pp. 297-314.

Thomas, D. E., and Moorby, P. (1991). *The Verilog Hardware Description Language, Kluwer Academic Publishers.*

Thuau, G., and Pilaud, D. (1990). "Using the Declarative Language LUSTRE for Circuit Verification," *Designing Correct Circuits,* eds. G. Jones, and M. Sheeran, Springer-Verlag, pp. 313-326.

Trimberger, S. (1994). "SRAM Programmable FPGAs," *Field-Programmable Gate Array Technology,* eds. S.M. Trimberger, Kluwer Academic Publishers, pp. 15-90.

Wadler, P. (1995). "Monads for Functional Programming," *Advanced Functional Programming*, eds. J. Jeuring and E. Meijer, Springer Verlag, LNCS 925.

Weber, S., Bloom B., and Brown, G. (1992a). "Compiling Joy to Silicon," *Advanced Research in VLSI and Parallel Systems*, eds. T. Knight, and J. Savage, MIT Press, pp. 79-98.

Weber, S., Bloom B., and Brown, G. (1992b). " Compiling Joy into Silicon: An Exercise in Applied Structural Operational Semantics," *Semantics: Foundations And Applications,* Lecture Notes in Computer Science 666, eds J.W.de Bakker, W.-P.de Roever, and Rozenberg, Springer-Verlag.

Wodtko, A. (1987). "RT Languages in Goal-Oriented CAD Algorithms," *Hardware Description Languages,* eds. R. W. Hartenstein, Elsevier Science Publishers B. V. (North-Holland).

Xilinx. (1995). *Synopsys (XSI) for FPGAs Interface/Tutorial Guide.*

Xilinx. (1997a). *XC4000E and XC4000X Series Field Programmable Gate Arrays.*

Xilinx. (1997b). *ALLIANCE Series Software-Quick Start Guide.*

Appendix A

The VHDL Descriptions Representing the Netlist Circuit and the Test Bench for a Sequential Multiplier

```
--File name: mult_out.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL source file representing the circuits generated from a
--SMALL program that implements the multiplication of two 4-bit numbers.


library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_OUT16 is
 port (assign_global_p : in std_logic_vector(0 to 7);
       val_global_p : in std_logic_vector(0 to 7);
       assert_global_multiplicand : in std_logic_vector(0 to 3);
       assert_global_multiplier : in std_logic_vector(0 to 3);
       assert_global_done : in std_logic;
       assert_global_go : in std_logic;
       d_input : in std_logic;
       clock : in std_logic;
       power : in std_logic;
       ground : in std_logic;
       global_p : out std_logic_vector(0 to 7);
       global_multiplicand : out std_logic_vector(0 to 3);
       global_multiplier : out std_logic_vector(0 to 3);
       global_done : out std_logic;
       global_go : out std_logic);
end SMALL_OUT16;

architecture Structure of SMALL_OUT16 is

 signal  wire880, wire877, wire874, wire871, wire868, wire865, wire862,
       wire859, wire833, wire831, wire827, wire825, wire809, wire804,
       wire799, wire795, wire793, wire777, wire775, wire770, wire769,
       wire766, wire764, wire762, wire760, wire759, wire758, wire757,
```

wire752, wire750, wire748, wire746, wire744, wire742, wire740,
wire738, wire736, wire734, wire732, wire730, wire728, wire726,
wire724, wire722, wire720, wire718, wire716, wire714, wire712,
wire710, wire708, wire706, wire705, wire704, wire703, wire698,
wire697, wire694, wire693, wire690, wire688, wire686, wire684,
wire683, wire682, wire681, wire676, wire674, wire672, wire670,
wire668, wire666, wire664, wire662, wire660, wire658, wire656,
wire654, wire652, wire650, wire648, wire646, wire644, wire643,
wire642, wire641, wire640, wire639, wire632, wire631, wire630,
wire629, wire628, wire627, wire620, wire619, wire618, wire617,
wire616, wire615, wire608, wire607, wire606, wire605, wire604,
wire603, wire596, wire594, wire592, wire590, wire588, wire586,
wire584, wire582, wire580, wire578, wire576, wire574, wire572,
wire570, wire568, wire566, wire564, wire562, wire560, wire558,
wire556, wire554, wire553, wire552, wire551, wire550, wire549,
wire548, wire547, wire538, wire537, wire536, wire535, wire534,
wire533, wire526, wire525, wire524, wire523, wire522, wire521,
wire514, wire512, wire510, wire508, wire506, wire504, wire503,
wire502, wire501, wire496, wire494, wire492, wire491, wire488,
wire487, wire484, wire482, wire481, wire478, wire477, wire474,
wire472, wire470, wire468, wire466, wire464, wire463, wire462,
wire461, wire456, wire454, wire453, wire450, wire449, wire446,
wire444, wire442, wire440, wire438, wire436, wire434, wire432,
wire430, wire428, wire426, wire424, wire422, wire420, wire418,
wire417, wire414, wire412, wire410, wire408, wire407, wire406,
wire405, wire400, wire399, wire396, wire394, wire392, wire390,
wire388, wire386, wire385, wire382, wire381, wire380, wire379,
wire374, wire373, wire370, wire369, wire368, wire367, wire362,
wire360, wire358, wire356, wire354, wire352, wire350, wire348,
wire346, wire344, wire343, wire342, wire341, wire340, wire339,
wire338, wire330, wire329, wire328, wire327, wire326, wire325,
wire324, wire316, wire315, wire314, wire313, wire312, wire311,
wire310, wire302, wire301, wire300, wire299, wire298, wire297,
wire296, wire288, wire287, wire286, wire285, wire284, wire283,
wire282, wire274, wire273, wire272, wire271, wire270, wire269,
wire268, wire260, wire259, wire258, wire257, wire256, wire255,
wire254, wire246, wire245, wire244, wire243, wire242, wire241,
wire240, wire232, wire231, wire230, wire229, wire228, wire227,
wire226, wire218, wire217, wire216, wire215, wire214, wire213,
wire212, wire201, wire200, wire199, wire198, wire197, wire196,
wire195, wire183, wire182, wire181, wire180, wire179, wire178,
wire177, wire165, wire164, wire163, wire162, wire161, wire160,
wire159, wire147, wire146, wire145, wire144, wire143, wire142,
wire141, wire129, wire128, wire127, wire126, wire125, wire124,
wire123, wire111, wire110, wire109, wire108, wire107, wire106,

```
        wire105, wire93, wire92, wire91, wire90, wire89, wire88,
        wire87, wire75, wire74, wire73, wire72, wire71, wire70,
        wire69, wire55, wire50, wire45, wire40, wire33, wire28,
        wire23, wire18, wire11, wire6, wire3, wire1: std_logic;

 component OR2
  port (I1, I2 : in std_logic; O1 : out std_logic);
 end component;

 component AND2
  port (I1, I2 : in std_logic; O1 : out std_logic);
 end component;

 component XOR2
  port (I1, I2 : in std_logic; O1 : out std_logic);
 end component;

 component BUF
  port (I1 : in std_logic; O1 : out std_logic);
 end component;

 component INVERTER
  port (I1 : in std_logic; O1 : out std_logic);
 end component;

 component DFLIPFLOP
  port (clock, D : in std_logic; Q : out std_logic);
 end component;

begin

  Device883: BUF port map ( wire392, wire1);
  Device882: BUF port map ( wire396, wire3);
  Device881: OR2 port map ( val_global_p(4), wire880, wire141);
  Device879: OR2 port map ( wire586, wire744, wire880);
  Device878: OR2 port map ( assign_global_p(4), wire877, wire142);
  Device876: OR2 port map ( wire582, wire740, wire877);
  Device875: OR2 port map ( val_global_p(5), wire874, wire159);
  Device873: OR2 port map ( wire588, wire746, wire874);
  Device872: OR2 port map ( assign_global_p(5), wire871, wire160);
  Device870: OR2 port map ( wire582, wire740, wire871);
  Device869: OR2 port map ( val_global_p(6), wire868, wire177);
  Device867: OR2 port map ( wire590, wire748, wire868);
  Device866: OR2 port map ( assign_global_p(6), wire865, wire178);
  Device864: OR2 port map ( wire582, wire740, wire865);
```

```
Device863: OR2 port map ( val_global_p(7), wire862, wire195);
Device861: OR2 port map ( wire592, wire750, wire862);
Device860: OR2 port map ( assign_global_p(7), wire859, wire196);
Device858: OR2 port map ( wire582, wire740, wire859);
Device857: BUF port map ( wire724, wire212);
Device856: BUF port map ( wire722, wire213);
Device855: BUF port map ( wire726, wire226);
Device854: BUF port map ( wire722, wire227);
Device853: BUF port map ( wire728, wire240);
Device852: BUF port map ( wire722, wire241);
Device851: BUF port map ( wire730, wire254);
Device850: BUF port map ( wire722, wire255);
Device849: BUF port map ( wire646, wire354);
Device848: BUF port map ( wire648, wire356);
Device847: BUF port map ( wire650, wire358);
Device846: BUF port map ( wire652, wire360);
Device845: BUF port map ( wire654, wire362);
Device844: BUF port map ( wire390, wire367);
Device843: BUF port map ( wire370, wire373);
Device842: BUF port map ( wire394, wire379);
Device841: BUF port map ( wire382, wire385);
Device840: BUF port map ( wire410, wire399);
Device839: BUF port map ( wire414, wire405);
Device838: BUF port map ( wire408, wire410);
Device837: BUF port map ( wire407, wire412);
Device836: BUF port map ( wire420, wire417);
Device835: BUF port map ( wire432, wire420);
Device834: OR2 port map ( ground, wire833, wire422);
Device832: OR2 port map ( ground, wire831, wire833);
Device830: OR2 port map ( ground, ground, wire831);
Device829: BUF port map ( wire444, wire432);
Device828: OR2 port map ( ground, wire827, wire434);
Device826: OR2 port map ( ground, wire825, wire827);
Device824: OR2 port map ( ground, ground, wire825);
Device823: BUF port map ( wire770, wire444);
Device822: BUF port map ( wire468, wire449);
Device821: BUF port map ( wire466, wire453);
Device820: BUF port map ( wire470, wire461);
Device819: BUF port map ( wire464, wire466);
Device818: BUF port map ( wire463, wire468);
Device817: BUF port map ( wire474, wire470);
Device816: BUF port map ( wire508, wire477);
Device815: BUF port map ( wire506, wire481);
Device814: BUF port map ( wire510, wire501);
Device813: BUF port map ( wire504, wire506);
```

Device812: BUF port map (wire503, wire508);
Device811: BUF port map (wire560, wire510);
Device810: OR2 port map (ground, wire809, wire514);
Device808: OR2 port map (wire512, wire554, wire809);
Device807: BUF port map (wire574, wire560);
Device806: BUF port map (ground, wire562);
Device805: OR2 port map (ground, wire804, wire564);
Device803: OR2 port map (wire562, ground, wire804);
Device802: BUF port map (wire582, wire574);
Device801: BUF port map (wire594, wire582);
Device800: OR2 port map (ground, wire799, wire596);
Device798: OR2 port map (ground, ground, wire799);
Device797: BUF port map (wire686, wire656);
Device796: OR2 port map (ground, wire795, wire658);
Device794: OR2 port map (ground, wire793, wire795);
Device792: OR2 port map (ground, ground, wire793);
Device791: BUF port map (wire688, wire668);
Device790: BUF port map (wire690, wire681);
Device789: BUF port map (wire684, wire686);
Device788: BUF port map (wire683, wire688);
Device787: BUF port map (wire710, wire693);
Device786: BUF port map (wire708, wire697);
Device785: BUF port map (wire712, wire703);
Device784: BUF port map (wire706, wire708);
Device783: BUF port map (wire705, wire710);
Device782: BUF port map (wire722, wire712);
Device781: BUF port map (wire732, wire722);
Device780: BUF port map (wire740, wire732);
Device779: BUF port map (wire752, wire740);
Device778: OR2 port map (ground, wire777, wire742);
Device776: OR2 port map (ground, wire775, wire777);
Device774: OR2 port map (ground, ground, wire775);
Device773: BUF port map (wire766, wire757);
Device772: BUF port map (wire760, wire762);
Device771: BUF port map (wire759, wire764);
Device768: INVERTER port map (wire769, wire770);
Device767: DFLIPFLOP port map (clock, d_input, wire769);
Device765: OR2 port map (wire450, wire770, wire766);
Device756: AND2 port map (wire758, wire757, wire760);
Device755: AND2 port map (wire757, power, wire759);
Device754: INVERTER port map (power, wire758);
Device751: OR2 port map (wire764, wire698, wire752);
Device749: AND2 port map (ground, wire740, wire750);
Device747: AND2 port map (ground, wire740, wire748);
Device745: AND2 port map (ground, wire740, wire746);

```
Device743: AND2 port map ( ground, wire740, wire744);
Device737: AND2 port map ( ground, wire732, wire738);
Device735: AND2 port map ( ground, wire732, wire736);
Device733: OR2 port map ( ground, ground, wire734);
Device729: AND2 port map ( wire55, wire722, wire730);
Device727: AND2 port map ( wire50, wire722, wire728);
Device725: AND2 port map ( wire45, wire722, wire726);
Device723: AND2 port map ( wire40, wire722, wire724);
Device719: AND2 port map ( wire33, wire712, wire720);
Device717: AND2 port map ( wire28, wire712, wire718);
Device715: AND2 port map ( wire23, wire712, wire716);
Device713: AND2 port map ( wire18, wire712, wire714);
Device702: AND2 port map ( wire704, wire703, wire706);
Device701: AND2 port map ( wire703, wire6, wire705);
Device700: INVERTER port map ( wire6, wire704);
Device696: DFLIPFLOP port map ( clock, wire697, wire698);
Device692: DFLIPFLOP port map ( clock, wire693, wire694);
Device689: OR2 port map ( wire694, wire482, wire690);
Device680: AND2 port map ( wire682, wire681, wire684);
Device679: AND2 port map ( wire681, wire274, wire683);
Device678: INVERTER port map ( wire274, wire682);
Device675: AND2 port map ( wire668, wire260, wire676);
Device673: AND2 port map ( wire668, wire246, wire674);
Device671: AND2 port map ( wire668, wire232, wire672);
Device669: AND2 port map ( wire668, wire218, wire670);
Device665: AND2 port map ( wire656, ground, wire666);
Device663: AND2 port map ( wire656, ground, wire664);
Device661: AND2 port map ( wire656, ground, wire662);
Device659: AND2 port map ( wire656, ground, wire660);
Device653: AND2 port map ( wire594, wire644, wire654);
Device651: AND2 port map ( wire594, wire643, wire652);
Device649: AND2 port map ( wire594, wire631, wire650);
Device647: AND2 port map ( wire594, wire619, wire648);
Device645: AND2 port map ( wire594, wire607, wire646);
Device638: XOR2 port map ( wire632, wire642, wire643);
Device637: XOR2 port map ( wire352, wire201, wire642);
Device636: OR2 port map ( wire639, wire641, wire644);
Device635: OR2 port map ( wire352, wire201, wire640);
Device634: AND2 port map ( wire632, wire640, wire641);
Device633: AND2 port map ( wire352, wire201, wire639);
Device626: XOR2 port map ( wire620, wire630, wire631);
Device625: XOR2 port map ( wire350, wire183, wire630);
Device624: OR2 port map ( wire627, wire629, wire632);
Device623: OR2 port map ( wire350, wire183, wire628);
Device622: AND2 port map ( wire620, wire628, wire629);
```

```
Device621: AND2 port map ( wire350, wire183, wire627);
Device614: XOR2 port map ( wire608, wire618, wire619);
Device613: XOR2 port map ( wire348, wire165, wire618);
Device612: OR2 port map ( wire615, wire617, wire620);
Device611: OR2 port map ( wire348, wire165, wire616);
Device610: AND2 port map ( wire608, wire616, wire617);
Device609: AND2 port map ( wire348, wire165, wire615);
Device602: XOR2 port map ( ground, wire606, wire607);
Device601: XOR2 port map ( wire346, wire147, wire606);
Device600: OR2 port map ( wire603, wire605, wire608);
Device599: OR2 port map ( wire346, wire147, wire604);
Device598: AND2 port map ( ground, wire604, wire605);
Device597: AND2 port map ( wire346, wire147, wire603);
Device593: OR2 port map ( wire668, wire656, wire594);
Device591: AND2 port map ( wire362, wire582, wire592);
Device589: AND2 port map ( wire360, wire582, wire590);
Device587: AND2 port map ( wire358, wire582, wire588);
Device585: AND2 port map ( wire356, wire582, wire586);
Device583: AND2 port map ( wire354, wire582, wire584);
Device579: AND2 port map ( wire129, wire574, wire580);
Device577: AND2 port map ( wire111, wire574, wire578);
Device575: AND2 port map ( wire93, wire574, wire576);
Device571: AND2 port map ( ground, wire560, wire572);
Device569: AND2 port map ( wire316, wire560, wire570);
Device567: AND2 port map ( wire302, wire560, wire568);
Device565: AND2 port map ( wire288, wire560, wire566);
Device557: AND2 port map ( wire537, wire510, wire558);
Device555: AND2 port map ( wire525, wire510, wire556);
Device546: OR2 port map ( wire552, wire553, wire554);
Device545: INVERTER port map ( wire537, wire549);
Device544: INVERTER port map ( ground, wire548);
Device543: INVERTER port map ( wire344, wire547);
Device542: AND2 port map ( wire548, wire547, wire551);
Device541: AND2 port map ( wire551, wire537, wire553);
Device540: AND2 port map ( wire549, wire550, wire552);
Device539: AND2 port map ( ground, wire344, wire550);
Device532: XOR2 port map ( wire526, wire536, wire537);
Device531: XOR2 port map ( ground, wire344, wire536);
Device530: OR2 port map ( wire533, wire535, wire538);
Device529: OR2 port map ( ground, wire344, wire534);
Device528: AND2 port map ( wire526, wire534, wire535);
Device527: AND2 port map ( ground, wire344, wire533);
Device520: XOR2 port map ( ground, wire524, wire525);
Device519: XOR2 port map ( power, wire330, wire524);
Device518: OR2 port map ( wire521, wire523, wire526);
```

Device517: OR2 port map (power, wire330, wire522);
Device516: AND2 port map (ground, wire522, wire523);
Device515: AND2 port map (power, wire330, wire521);
Device511: OR2 port map (ground, ground, wire512);
Device500: AND2 port map (wire502, wire501, wire504);
Device499: AND2 port map (wire501, wire494, wire503);
Device498: INVERTER port map (wire494, wire502);
Device495: OR2 port map (wire484, ground, wire496);
Device493: AND2 port map (wire492, wire488, wire494);
Device490: INVERTER port map (wire491, wire492);
Device489: XOR2 port map (power, wire344, wire491);
Device486: INVERTER port map (wire487, wire488);
Device485: XOR2 port map (power, wire330, wire487);
Device483: OR2 port map (ground, ground, wire484);
Device480: DFLIPFLOP port map (clock, wire481, wire482);
Device476: DFLIPFLOP port map (clock, wire477, wire478);
Device473: OR2 port map (wire478, wire454, wire474);
Device471: AND2 port map (wire470, power, wire472);
Device460: AND2 port map (wire462, wire461, wire464);
Device459: AND2 port map (wire461, wire456, wire463);
Device458: INVERTER port map (wire456, wire462);
Device455: INVERTER port map (wire6, wire456);
Device452: DFLIPFLOP port map (clock, wire453, wire454);
Device448: DFLIPFLOP port map (clock, wire449, wire450);
Device445: AND2 port map (wire444, power, wire446);
Device441: AND2 port map (wire432, ground, wire442);
Device439: AND2 port map (wire432, ground, wire440);
Device437: AND2 port map (wire432, power, wire438);
Device435: AND2 port map (wire432, power, wire436);
Device429: AND2 port map (wire420, ground, wire430);
Device427: AND2 port map (wire420, ground, wire428);
Device425: AND2 port map (wire420, power, wire426);
Device423: AND2 port map (wire420, ground, wire424);
Device416: DFLIPFLOP port map (clock, wire417, wire418);
Device413: OR2 port map (wire418, wire400, wire414);
Device404: AND2 port map (wire406, wire405, wire408);
Device403: AND2 port map (wire405, wire11, wire407);
Device402: INVERTER port map (wire11, wire406);
Device398: DFLIPFLOP port map (clock, wire399, wire400);
Device395: AND2 port map (wire394, power, wire396);
Device393: OR2 port map (wire762, wire386, wire394);
Device391: AND2 port map (wire390, power, wire392);
Device389: OR2 port map (wire412, wire374, wire390);
Device387: OR2 port map (wire381, wire369, wire388);
Device384: DFLIPFLOP port map (clock, wire385, wire386);

```
Device378: AND2 port map ( wire380, wire379, wire382);
Device377: AND2 port map ( wire379, wire1, wire381);
Device376: INVERTER port map ( wire1, wire380);
Device372: DFLIPFLOP port map ( clock, wire373, wire374);
Device366: AND2 port map ( wire368, wire367, wire370);
Device365: AND2 port map ( wire367, wire3, wire369);
Device364: INVERTER port map ( wire3, wire368);
Device351: OR2 port map ( wire676, wire666, wire352);
Device349: OR2 port map ( wire674, wire664, wire350);
Device347: OR2 port map ( wire672, wire662, wire348);
Device345: OR2 port map ( wire670, wire660, wire346);
Device337: DFLIPFLOP port map ( clock, wire343, wire344);
Device336: AND2 port map ( wire342, wire344, wire341);
Device335: OR2 port map ( wire341, wire340, wire343);
Device334: INVERTER port map ( wire339, wire342);
Device333: AND2 port map ( wire339, wire338, wire340);
Device332: OR2 port map ( wire732, wire510, wire339);
Device331: OR2 port map ( wire738, wire558, wire338);
Device323: DFLIPFLOP port map ( clock, wire329, wire330);
Device322: AND2 port map ( wire328, wire330, wire327);
Device321: OR2 port map ( wire327, wire326, wire329);
Device320: INVERTER port map ( wire325, wire328);
Device319: AND2 port map ( wire325, wire324, wire326);
Device318: OR2 port map ( wire732, wire510, wire325);
Device317: OR2 port map ( wire736, wire556, wire324);
Device309: DFLIPFLOP port map ( clock, wire315, wire316);
Device308: AND2 port map ( wire314, wire316, wire313);
Device307: OR2 port map ( wire313, wire312, wire315);
Device306: INVERTER port map ( wire311, wire314);
Device305: AND2 port map ( wire311, wire310, wire312);
Device304: OR2 port map ( wire712, wire560, wire311);
Device303: OR2 port map ( wire720, wire572, wire310);
Device295: DFLIPFLOP port map ( clock, wire301, wire302);
Device294: AND2 port map ( wire300, wire302, wire299);
Device293: OR2 port map ( wire299, wire298, wire301);
Device292: INVERTER port map ( wire297, wire300);
Device291: AND2 port map ( wire297, wire296, wire298);
Device290: OR2 port map ( wire712, wire560, wire297);
Device289: OR2 port map ( wire718, wire570, wire296);
Device281: DFLIPFLOP port map ( clock, wire287, wire288);
Device280: AND2 port map ( wire286, wire288, wire285);
Device279: OR2 port map ( wire285, wire284, wire287);
Device278: INVERTER port map ( wire283, wire286);
Device277: AND2 port map ( wire283, wire282, wire284);
Device276: OR2 port map ( wire712, wire560, wire283);
```

```
Device275: OR2 port map ( wire716, wire568, wire282);
Device267: DFLIPFLOP port map ( clock, wire273, wire274);
Device266: AND2 port map ( wire272, wire274, wire271);
Device265: OR2 port map ( wire271, wire270, wire273);
Device264: INVERTER port map ( wire269, wire272);
Device263: AND2 port map ( wire269, wire268, wire270);
Device262: OR2 port map ( wire712, wire560, wire269);
Device261: OR2 port map ( wire714, wire566, wire268);
Device253: DFLIPFLOP port map ( clock, wire259, wire260);
Device252: AND2 port map ( wire258, wire260, wire257);
Device251: OR2 port map ( wire257, wire256, wire259);
Device250: INVERTER port map ( wire255, wire258);
Device249: AND2 port map ( wire255, wire254, wire256);
Device239: DFLIPFLOP port map ( clock, wire245, wire246);
Device238: AND2 port map ( wire244, wire246, wire243);
Device237: OR2 port map ( wire243, wire242, wire245);
Device236: INVERTER port map ( wire241, wire244);
Device235: AND2 port map ( wire241, wire240, wire242);
Device225: DFLIPFLOP port map ( clock, wire231, wire232);
Device224: AND2 port map ( wire230, wire232, wire229);
Device223: OR2 port map ( wire229, wire228, wire231);
Device222: INVERTER port map ( wire227, wire230);
Device221: AND2 port map ( wire227, wire226, wire228);
Device211: DFLIPFLOP port map ( clock, wire217, wire218);
Device210: AND2 port map ( wire216, wire218, wire215);
Device209: OR2 port map ( wire215, wire214, wire217);
Device208: INVERTER port map ( wire213, wire216);
Device207: AND2 port map ( wire213, wire212, wire214);
Device194: BUF port map ( wire201, global_p(7));
Device193: DFLIPFLOP port map ( clock, wire200, wire201);
Device192: AND2 port map ( wire199, wire201, wire198);
Device191: OR2 port map ( wire198, wire197, wire200);
Device190: INVERTER port map ( wire196, wire199);
Device189: AND2 port map ( wire196, wire195, wire197);
Device176: BUF port map ( wire183, global_p(6));
Device175: DFLIPFLOP port map ( clock, wire182, wire183);
Device174: AND2 port map ( wire181, wire183, wire180);
Device173: OR2 port map ( wire180, wire179, wire182);
Device172: INVERTER port map ( wire178, wire181);
Device171: AND2 port map ( wire178, wire177, wire179);
Device158: BUF port map ( wire165, global_p(5));
Device157: DFLIPFLOP port map ( clock, wire164, wire165);
Device156: AND2 port map ( wire163, wire165, wire162);
Device155: OR2 port map ( wire162, wire161, wire164);
Device154: INVERTER port map ( wire160, wire163);
```

```
Device153: AND2 port map ( wire160, wire159, wire161);
Device140: BUF port map ( wire147, global_p(4));
Device139: DFLIPFLOP port map ( clock, wire146, wire147);
Device138: AND2 port map ( wire145, wire147, wire144);
Device137: OR2 port map ( wire144, wire143, wire146);
Device136: INVERTER port map ( wire142, wire145);
Device135: AND2 port map ( wire142, wire141, wire143);
Device122: BUF port map ( wire129, global_p(3));
Device121: DFLIPFLOP port map ( clock, wire128, wire129);
Device120: AND2 port map ( wire127, wire129, wire126);
Device119: OR2 port map ( wire126, wire125, wire128);
Device118: INVERTER port map ( wire124, wire127);
Device117: AND2 port map ( wire124, wire123, wire125);
Device116: OR2 port map ( wire582, assign_global_p(3), wire124);
Device115: OR2 port map ( wire584, val_global_p(3), wire123);
Device104: BUF port map ( wire111, global_p(2));
Device103: DFLIPFLOP port map ( clock, wire110, wire111);
Device102: AND2 port map ( wire109, wire111, wire108);
Device101: OR2 port map ( wire108, wire107, wire110);
Device100: INVERTER port map ( wire106, wire109);
Device99: AND2 port map ( wire106, wire105, wire107);
Device98: OR2 port map ( wire574, assign_global_p(2), wire106);
Device97: OR2 port map ( wire580, val_global_p(2), wire105);
Device86: BUF port map ( wire93, global_p(1));
Device85: DFLIPFLOP port map ( clock, wire92, wire93);
Device84: AND2 port map ( wire91, wire93, wire90);
Device83: OR2 port map ( wire90, wire89, wire92);
Device82: INVERTER port map ( wire88, wire91);
Device81: AND2 port map ( wire88, wire87, wire89);
Device80: OR2 port map ( wire574, assign_global_p(1), wire88);
Device79: OR2 port map ( wire578, val_global_p(1), wire87);
Device68: BUF port map ( wire75, global_p(0));
Device67: DFLIPFLOP port map ( clock, wire74, wire75);
Device66: AND2 port map ( wire73, wire75, wire72);
Device65: OR2 port map ( wire72, wire71, wire74);
Device64: INVERTER port map ( wire70, wire73);
Device63: AND2 port map ( wire70, wire69, wire71);
Device62: OR2 port map ( wire574, assign_global_p(0), wire70);
Device61: OR2 port map ( wire576, val_global_p(0), wire69);
Device57: BUF port map ( wire55, global_multiplicand(3));
Device53: OR2 port map ( wire430, assert_global_multiplicand(3), wire55);
Device52: BUF port map ( wire50, global_multiplicand(2));
Device48: OR2 port map ( wire428, assert_global_multiplicand(2), wire50);
Device47: BUF port map ( wire45, global_multiplicand(1));
Device43: OR2 port map ( wire426, assert_global_multiplicand(1), wire45);
```

```
  Device42: BUF port map ( wire40, global_multiplicand(0));
  Device38: OR2 port map ( wire424, assert_global_multiplicand(0), wire40);
  Device35: BUF port map ( wire33, global_multiplier(3));
  Device31: OR2 port map ( wire442, assert_global_multiplier(3), wire33);
  Device30: BUF port map ( wire28, global_multiplier(2));
  Device26: OR2 port map ( wire440, assert_global_multiplier(2), wire28);
  Device25: BUF port map ( wire23, global_multiplier(1));
  Device21: OR2 port map ( wire438, assert_global_multiplier(1), wire23);
  Device20: BUF port map ( wire18, global_multiplier(0));
  Device16: OR2 port map ( wire436, assert_global_multiplier(0), wire18);
  Device13: BUF port map ( wire11, global_done);
  Device9: OR2 port map ( wire472, assert_global_done, wire11);
  Device8: BUF port map ( wire6, global_go);
  Device4: OR2 port map ( wire446, assert_global_go, wire6);
end Structure;
```

Figure A.1: The Structural VHDL Output Description for a Sequential Multiplier

```
--File name: mult_test.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description of the test bench that is used to test the circuit
--described in Figure A.1

library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_TEST16 is
end SMALL_TEST16;

architecture TEST16 of SMALL_TEST16 is

 component SMALL_OUT16
  port(assign_global_p : in std_logic_vector(0 to 7);
      val_global_p : in std_logic_vector(0 to 7);
      assert_global_multiplicand : in std_logic_vector(0 to 3);
      assert_global_multiplier : in std_logic_vector(0 to 3);
      assert_global_done : in std_logic;
      assert_global_go : in std_logic;
```

```
    d_input: in std_logic;
    clock : in std_logic;
    power : in std_logic;
    ground : in std_logic;
    global_p : out std_logic_vector(0 to 7);
    global_multiplicand : out std_logic_vector(0 to 3);
    global_multiplier : out std_logic_vector(0 to 3);
    global_done : out std_logic;
    global_go : out std_logic);
end component;

signal assign_global_p : std_logic_vector(0 to 7) := "00000000";
signal val_global_p :  std_logic_vector(0 to 7) := "00000000";
signal assert_global_multiplicand : std_logic_vector(0 to 3) := "0000";
signal assert_global_multiplier : std_logic_vector(0 to 3) := "0000";
signal assert_global_done : std_logic := '0';
signal assert_global_go : std_logic := '0';
signal global_multiplicand, global_multiplier: std_logic_vector(0 to 3);
signal global_p : std_logic_vector(0 to 7);
signal ground: std_logic := '0';
signal global_done, global_go, d_input : std_logic;
signal clock: std_logic := '0';
signal power: std_logic := '1';
begin

AA1: SMALL_OUT16 port map (assign_global_p, val_global_p,
assert_global_multiplicand, assert_global_multiplier, assert_global_done,
assert_global_go, d_input, clock, power, ground, global_p, global_multiplicand ,
global_multiplier, global_done, global_go);
clock <= not clock after 10 ns;
d_input <= 'U', power after 5 ns;
end TEST16;
configuration conf_SMALL_16 of SMALL_TEST16 is
 for TEST16
 end for;
end conf_SMALL_16;
```

Figure A.2: The VHDL Description of the Test Bench for a Sequential Multiplier

Appendix B

The VHDL Descriptions Representing the Netlist Circuit and the Test Bench for a Serial Adder

```
--File name: adder_out.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a structural VHDL description created by the netlist generator for
-- the serial adder in Figure 4.11.


library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_OUT4 is
 port (assert_global_x : in std_logic;
      assert_global_e : in std_logic;
      assign_global_d : in std_logic;
      val_global_d : in std_logic;
      assign_global_c : in std_logic;
      val_global_c : in std_logic;
      assert_global_b : in std_logic;
      assert_global_a : in std_logic;
      d_input : in std_logic;
      clock : in std_logic;
      power : in std_logic;
      ground : in std_logic;
      global_x : out std_logic;
      global_e : out std_logic;
      global_d : out std_logic;
      global_c : out std_logic;
      global_b : out std_logic;
      global_a : out std_logic);
end SMALL_OUT4;

architecture Structure of SMALL_OUT4 is
```

```
signal  wire133, wire132, wire129, wire127, wire125, wire123, wire122,
        wire121, wire120, wire115, wire113, wire111, wire109, wire107,
        wire105, wire103, wire101, wire99, wire97, wire95, wire93,
        wire91, wire89, wire87, wire85, wire83, wire81, wire79,
        wire77, wire75, wire74, wire71, wire69, wire67, wire65,
        wire64, wire61, wire59, wire58, wire53, wire48, wire42,
        wire41, wire40, wire39, wire38, wire37, wire36, wire24,
        wire23, wire22, wire21, wire20, wire19, wire18, wire7, wire2: std_logic;

component OR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component AND2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component XOR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component BUF
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component INVERTER
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component DFLIPFLOP
 port (clock, D : in std_logic; Q : out std_logic);
end component;

begin

 Device143: BUF port map ( assert_global_a, wire2);
 Device142: BUF port map ( assert_global_b, wire7);
 Device141: BUF port map ( wire61, wire58);
 Device140: BUF port map ( wire71, wire61);
 Device139: BUF port map ( wire81, wire71);
 Device138: BUF port map ( wire105, wire81);
 Device137: BUF port map ( wire127, wire105);
 Device136: BUF port map ( wire129, wire120);
 Device135: BUF port map ( wire123, wire125);
 Device134: BUF port map ( wire122, wire127);
```

```
Device131: INVERTER port map ( wire132, wire133);
Device130: DFLIPFLOP port map ( clock, d_input, wire132);
Device128: OR2 port map ( wire133, wire59, wire129);
Device119: AND2 port map ( wire121, wire120, wire123);
Device118: AND2 port map ( wire120, power, wire122);
Device117: INVERTER port map ( power, wire121);
Device114: AND2 port map ( wire105, wire111, wire115);
Device112: OR2 port map ( ground, wire109, wire113);
Device110: XOR2 port map ( wire24, wire107, wire111);
Device108: OR2 port map ( ground, ground, wire109);
Device106: XOR2 port map ( wire7, wire2, wire107);
Device102: AND2 port map ( wire99, wire81, wire103);
Device100: OR2 port map ( wire97, wire93, wire101);
Device98: OR2 port map ( wire95, wire91, wire99);
Device96: OR2 port map ( ground, ground, wire97);
Device94: AND2 port map ( wire24, wire7, wire95);
Device92: OR2 port map ( wire89, wire85, wire93);
Device90: OR2 port map ( wire87, wire83, wire91);
Device88: OR2 port map ( ground, ground, wire89);
Device86: AND2 port map ( wire24, wire2, wire87);
Device84: OR2 port map ( ground, ground, wire85);
Device82: AND2 port map ( wire7, wire2, wire83);
Device78: AND2 port map ( wire75, wire71, wire79);
Device76: OR2 port map ( ground, ground, wire77);
Device73: OR2 port map ( wire7, wire74, wire75);
Device72: INVERTER port map ( wire2, wire74);
Device68: AND2 port map ( wire61, wire65, wire69);
Device66: OR2 port map ( ground, ground, wire67);
Device63: OR2 port map ( wire2, wire64, wire65);
Device62: INVERTER port map ( wire7, wire64);
Device57: DFLIPFLOP port map ( clock, wire58, wire59);
Device55: BUF port map ( wire53, global_x);
Device51: OR2 port map ( wire115, assert_global_x, wire53);
Device50: BUF port map ( wire48, global_e);
Device46: OR2 port map ( wire69, assert_global_e, wire48);
Device35: BUF port map ( wire42, global_d);
Device34: DFLIPFLOP port map ( clock, wire41, wire42);
Device33: AND2 port map ( wire40, wire42, wire39);
Device32: OR2 port map ( wire39, wire38, wire41);
Device31: INVERTER port map ( wire37, wire40);
Device30: AND2 port map ( wire37, wire36, wire38);
Device29: OR2 port map ( wire71, assign_global_d, wire37);
Device28: OR2 port map ( wire79, val_global_d, wire36);
Device17: BUF port map ( wire24, global_c);
Device16: DFLIPFLOP port map ( clock, wire23, wire24);
```

```
  Device15: AND2 port map ( wire22, wire24, wire21);
  Device14: OR2 port map ( wire21, wire20, wire23);
  Device13: INVERTER port map ( wire19, wire22);
  Device12: AND2 port map ( wire19, wire18, wire20);
  Device11: OR2 port map ( wire81, assign_global_c, wire19);
  Device10: OR2 port map ( wire103, val_global_c, wire18);
  Device9: BUF port map ( wire7, global_b);
  Device4: BUF port map ( wire2, global_a);
end Structure;
```

Figure B.1: The Structural VHDL Output Description for a Serial Adder

```
--File name: adder_test.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description of the test bench that is used to test the circuit
--described in Figure B.1


library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_TEST4 is
end SMALL_TEST4;

architecture TEST4 of SMALL_TEST4 is

 component SMALL_OUT4
 port (assert_global_x : in std_logic;
     assert_global_e : in std_logic;
     assign_global_d : in std_logic;
     val_global_d : in std_logic;
     assign_global_c : in std_logic;
     val_global_c : in std_logic;
     assert_global_b : in std_logic;
     assert_global_a : in std_logic;
     d_input : in std_logic;
     clock : in std_logic;
     power : in std_logic;
     ground : in std_logic;
```

```
      global_x : out std_logic;
      global_e : out std_logic;
      global_d : out std_logic;
      global_c : out std_logic;
      global_b : out std_logic;
      global_a : out std_logic);
 end component;

signal assign_global_c, val_global_c: std_logic := '0';
signal assign_global_d, val_global_d: std_logic := '0';
signal assert_global_x :  std_logic := '0';
signal assert_global_e :  std_logic := '0';
signal assert_global_b, assert_global_a, d_input :  std_logic;
signal global_c, global_d, global_e, global_x, global_b, global_a : std_logic;
signal ground: std_logic := '0';
signal clock: std_logic := '1';
signal power: std_logic := '1';
begin

BA1: SMALL_OUT4 port map (assert_global_x, assert_global_e, assign_global_d,
        val_global_d, assign_global_c, val_global_c, assert_global_b,
        assert_global_a, d_input, clock, power, ground, global_x, global_e,
        global_d, global_c, global_b, global_a);
clock <= not clock after 10 ns;
d_input <= 'U', power after 10 ns;
assert_global_a <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns,
                   '1' after 80 ns, '0' after 100 ns, '1' after 120 ns;
assert_global_b <= '1', '0' after 20 ns, '0' after 40 ns, '0' after 60 ns,
                  '1' after 80 ns, '0' after 100 ns, '1' after 120 ns;
end TEST4;
configuration conf_SMALL_4 of SMALL_TEST4 is
 for TEST4
 end for;
end conf_SMALL_4;
```

Figure B.2: The VHDL Description of the Test Bench for a Serial Adder

Appendix C

The VHDL Descriptions Representing the Array Types and the Test Bench for the Operations of Two-dimensional Arrays

```
--File name: package4.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description used for defining the types of two-dimensional
--arrays in Example 4.


library IEEE;
use IEEE.std_logic_1164.all;

package P is

    type A236 is array (0 to 2) of std_logic;
    type B236 is array (0 to 1) of A236;
    type A235 is array (0 to 2) of std_logic;
    type B235 is array (0 to 1) of A235;
    type A234 is array (0 to 2) of std_logic;
    type B234 is array (0 to 1) of A234;
    type A119 is array (0 to 2) of std_logic;
    type B119 is array (0 to 1) of A119;
    type A118 is array (0 to 2) of std_logic;
    type B118 is array (0 to 1) of A118;
    type A117 is array (0 to 2) of std_logic;
    type B117 is array (0 to 1) of A117;
    type A77 is array (0 to 2) of std_logic;
    type B77 is array (0 to 1) of A77;
    type A76 is array (0 to 2) of std_logic;
    type B76 is array (0 to 1) of A76;
    type A41 is array (0 to 2) of std_logic;
    type B41 is array (0 to 1) of A41;
    type A40 is array (0 to 2) of std_logic;
    type B40 is array (0 to 1) of A40;
```

```
    type A5 is array (0 to 2) of std_logic;
    type B5 is array (0 to 1) of A5;
    type A4 is array (0 to 2) of std_logic;
    type B4 is array (0 to 1) of A4;

end P;
```

Figure C.1: The VHDL Description for Data Types Used in Example 4

```
--File name: example4_test.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description of the test bench used in Example 4.


library IEEE;
use IEEE.std_logic_1164.all;
use WORK.P.all;

entity SMALL_TEST38 is
end SMALL_TEST38;

architecture TEST38 of SMALL_TEST38 is

 component SMALL_OUT38
  port (assign_global_f: in B235;
     val_global_f: in B234;
     assign_global_e: in B118;
     val_global_e: in B117;
     assert_global_d : in std_logic;
     assert_global_c: in B76;
     assert_global_b: in B40;
     assert_global_a: in B4;
     d_input : in std_logic;
     clock : in std_logic;
     power : in std_logic;
     ground : in std_logic;
     global_f: out B236;
     global_e: out B119;
     global_d : out std_logic;
     global_c: out B77;
```

```
      global_b: out B41;
      global_a: out B5);

 end component;

 signal assign_global_f : B235:= ("000", "000");
 signal val_global_f : B234:= ("000", "000");
 signal assign_global_e : B118:= ("000", "000");
 signal val_global_e : B117:= ("000", "000");
 --signal assert_global_d : B112:= ("0000", "0000");
 signal assert_global_d : std_logic := '0';
 signal assert_global_c : B76:= ("000", "000");
 signal assert_global_b : B40 := ("000", "000");
 signal assert_global_a : B4 :=  ("000", "000");
 signal global_f : B236;
 signal global_e : B119;
 signal global_d : std_logic;
 signal global_c : B77;
 signal global_b : B41;
 signal global_a : B5;

 signal ground: std_logic := '0';
 signal d_input : std_logic;
 signal clock: std_logic := '0';
 signal power: std_logic := '1';
 begin

 AA1: SMALL_OUT38 port map ( assign_global_f, val_global_f,
         assign_global_e, val_global_e,
    assert_global_d, assert_global_c, assert_global_b,
    assert_global_a, d_input, clock, power, ground,global_f,
    global_e, global_d, global_c,global_b, global_a);
 clock <= not clock after 10 ns;
 d_input <= 'U', power after 3 ns;
 end TEST38;

 configuration conf_SMALL_38 of SMALL_TEST38 is
  for TEST38
  end for;
 end conf_SMALL_38;
```

Figure C.2: The VHDL Description of the Test Bench for Example 4

Appendix D

The VHDL Descriptions Representing the Array Types and the Test Bench for the Operations of Three-dimensional Arrays in Example 5

```
--File name: package5.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description used for defining the types of three-
-- dimensional arrays in Example 5.


library IEEE;
use IEEE.std_logic_1164.all;

package P is

    type A167 is array (0 to 2) of std_logic;
    type B167 is array (0 to 1) of A167;
    type C167 is array (0 to 1) of B167;
    type A166 is array (0 to 2) of std_logic;
    type B166 is array (0 to 1) of A166;
    type C166 is array (0 to 1) of B166;
    type A113 is array (0 to 1) of std_logic;
    type B113 is array (0 to 1) of A113;
    type C113 is array (0 to 1) of B113;
    type A112 is array (0 to 1) of std_logic;
    type B112 is array (0 to 1) of A112;
    type C112 is array (0 to 1) of B112;
    type A59 is array (0 to 1) of std_logic;
    type B59 is array (0 to 1) of A59;
    type C59 is array (0 to 1) of B59;
    type A58 is array (0 to 1) of std_logic;
    type B58 is array (0 to 1) of A58;
    type C58 is array (0 to 1) of B58;
    type A5 is array (0 to 1) of std_logic;
```

```
    type B5 is array (0 to 1) of A5;
    type C5 is array (0 to 1) of B5;
    type A4 is array (0 to 1) of std_logic;
    type B4 is array (0 to 1) of A4;
    type C4 is array (0 to 1) of B4;

end P;
```

Figure D.1: The VHDL Description for Data Types Used in Example 5

```
--File name: example5_test.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description of the test bench used in Example 5.


library IEEE;
use IEEE.std_logic_1164.all;
use WORK.P.all;

entity SMALL_TEST37 is
end SMALL_TEST37;

architecture TEST37 of SMALL_TEST37 is

 component SMALL_OUT37
  port(assert_global_d: in C166;
     assert_global_c: in C112;
     assert_global_b: in C58;
     assert_global_a: in C4;
     d_input : in std_logic;
     clock : in std_logic;
     power : in std_logic;
     ground : in std_logic;
     global_d: out C167;
     global_c: out C113;
     global_b: out C59;
     global_a: out C5);

 end component;
```

```
signal assert_global_d : C166 := (("000", "000"), ("000", "000"));
signal assert_global_c : C112:= (("00", "00"), ("00", "00"));
signal assert_global_b : C58 := (("00", "00"), ("00", "00"));
signal assert_global_a : C4 :=  (("00", "00"), ("00", "00"));
signal global_d : C167;
signal global_c : C113;
signal global_b : C59;
signal global_a : C5;

signal ground: std_logic := '0';
signal d_input : std_logic;
signal clock: std_logic := '0';
signal power: std_logic := '1';

begin

AA1: SMALL_OUT37 port map (
    assert_global_d, assert_global_c, assert_global_b,
    assert_global_a, d_input, clock, power, ground,
    global_d, global_c,global_b, global_a);
clock <= not clock after 10 ns;
d_input <= 'U', power after 10 ns;
end TEST37;

configuration conf_SMALL_37 of SMALL_TEST37 is
 for TEST37
 end for;
end conf_SMALL_37;
```

Figure D.2: The VHDL Description of the Test Bench for Example 5

Appendix E

The VHDL Descriptions Representing the Netlist Circuit and the Test Bench for Example 6

```
--File name: image_out.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description created by the netlist generator for
--Example 6.


library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_OUT66 is
 port (assert_global_done : in std_logic;
     assert_global_go : in std_logic;
     assert_global_found : in std_logic_vector(0 to 1);
     assert_global_pattern : in std_logic_vector(0 to 1);
     assert_global_image : in std_logic_vector(0 to 2);
     d_input : in std_logic;
     clock : in std_logic;
     power : in std_logic;
     ground : in std_logic;
     global_done : out std_logic;
     global_go : out std_logic;
     global_found : out std_logic_vector(0 to 1);
     global_pattern : out std_logic_vector(0 to 1);
     global_image : out std_logic_vector(0 to 2));
end SMALL_OUT66;

architecture Structure of SMALL_OUT66 is

 signal  wire501, wire499, wire496, wire494, wire491, wire489, wire486,
       wire484, wire481, wire479, wire476, wire474, wire455, wire444,
       wire433, wire424, wire422, wire419, wire417, wire392, wire391,
```

```
        wire388, wire386, wire384, wire382, wire381, wire380, wire379,
        wire374, wire372, wire370, wire368, wire367, wire366, wire365,
        wire360, wire358, wire357, wire354, wire352, wire350, wire349,
        wire348, wire347, wire342, wire340, wire339, wire336, wire335,
        wire332, wire330, wire328, wire327, wire326, wire325, wire320,
        wire318, wire317, wire314, wire312, wire310, wire308, wire306,
        wire305, wire304, wire303, wire298, wire296, wire295, wire292,
        wire291, wire288, wire286, wire284, wire283, wire282, wire281,
        wire276, wire274, wire273, wire270, wire268, wire266, wire264,
        wire262, wire260, wire258, wire256, wire255, wire252, wire251,
        wire250, wire249, wire244, wire243, wire240, wire239, wire238,
        wire237, wire232, wire230, wire228, wire227, wire224, wire222,
        wire220, wire218, wire216, wire214, wire212, wire210, wire208,
        wire206, wire204, wire202, wire200, wire198, wire196, wire195,
        wire194, wire193, wire188, wire187, wire184, wire183, wire180,
        wire178, wire176, wire174, wire172, wire170, wire168, wire166,
        wire164, wire162, wire160, wire158, wire156, wire154, wire152,
        wire151, wire150, wire149, wire144, wire143, wire140, wire139,
        wire136, wire134, wire132, wire130, wire128, wire126, wire124,
        wire122, wire120, wire118, wire116, wire114, wire112, wire110,
        wire108, wire107, wire106, wire105, wire100, wire99, wire96,
        wire95, wire92, wire90, wire88, wire86, wire84, wire82,
        wire81, wire78, wire77, wire76, wire75, wire70, wire69,
        wire66, wire65, wire64, wire63, wire58, wire56, wire54,
        wire52, wire48, wire43, wire38, wire33, wire26, wire21,
        wire14, wire9, wire4: std_logic;

component OR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component AND2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component XOR2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component BUF
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component INVERTER
 port (I1 : in std_logic; O1 : out std_logic);
```

```
end component;

component DFLIPFLOP
 port (clock, D : in std_logic; Q : out std_logic);
end component;

begin

 Device502: OR2 port map ( assert_global_image(0), wire501, wire4);
 Device500: OR2 port map ( wire130, wire499, wire501);
 Device498: OR2 port map ( wire174, wire218, wire499);
 Device497: OR2 port map ( assert_global_image(1), wire496, wire9);
 Device495: OR2 port map ( wire132, wire494, wire496);
 Device493: OR2 port map ( wire176, wire220, wire494);
 Device492: OR2 port map ( assert_global_image(2), wire491, wire14);
 Device490: OR2 port map ( wire134, wire489, wire491);
 Device488: OR2 port map ( wire178, wire222, wire489);
 Device487: OR2 port map ( assert_global_pattern(0), wire486, wire21);
 Device485: OR2 port map ( wire122, wire484, wire486);
 Device483: OR2 port map ( wire166, wire210, wire484);
 Device482: OR2 port map ( assert_global_pattern(1), wire481, wire26);
 Device480: OR2 port map ( wire124, wire479, wire481);
 Device478: OR2 port map ( wire168, wire212, wire479);
 Device477: OR2 port map ( assert_global_go, wire476, wire43);
 Device475: OR2 port map ( wire116, wire474, wire476);
 Device473: OR2 port map ( wire160, wire204, wire474);
 Device472: BUF port map ( wire88, wire52);
 Device471: BUF port map ( wire92, wire54);
 Device470: BUF port map ( wire262, wire56);
 Device469: BUF port map ( wire266, wire58);
 Device468: BUF port map ( wire86, wire63);
 Device467: BUF port map ( wire66, wire69);
 Device466: BUF port map ( wire90, wire75);
 Device465: BUF port map ( wire78, wire81);
 Device464: BUF port map ( wire112, wire95);
 Device463: BUF port map ( wire110, wire99);
 Device462: BUF port map ( wire114, wire105);
 Device461: BUF port map ( wire108, wire110);
 Device460: BUF port map ( wire107, wire112);
 Device459: BUF port map ( wire118, wire114);
 Device458: BUF port map ( wire126, wire118);
 Device457: BUF port map ( wire136, wire126);
 Device456: OR2 port map ( ground, wire455, wire128);
 Device454: OR2 port map ( ground, ground, wire455);
 Device453: BUF port map ( wire156, wire139);
```

```
Device452: BUF port map ( wire154, wire143);
Device451: BUF port map ( wire158, wire149);
Device450: BUF port map ( wire152, wire154);
Device449: BUF port map ( wire151, wire156);
Device448: BUF port map ( wire162, wire158);
Device447: BUF port map ( wire170, wire162);
Device446: BUF port map ( wire180, wire170);
Device445: OR2 port map ( ground, wire444, wire172);
Device443: OR2 port map ( ground, ground, wire444);
Device442: BUF port map ( wire200, wire183);
Device441: BUF port map ( wire198, wire187);
Device440: BUF port map ( wire202, wire193);
Device439: BUF port map ( wire196, wire198);
Device438: BUF port map ( wire195, wire200);
Device437: BUF port map ( wire206, wire202);
Device436: BUF port map ( wire214, wire206);
Device435: BUF port map ( wire224, wire214);
Device434: OR2 port map ( ground, wire433, wire216);
Device432: OR2 port map ( ground, ground, wire433);
Device431: BUF port map ( wire230, wire227);
Device430: BUF port map ( wire258, wire230);
Device429: BUF port map ( wire260, wire237);
Device428: BUF port map ( wire240, wire243);
Device427: BUF port map ( wire264, wire249);
Device426: BUF port map ( wire252, wire255);
Device425: OR2 port map ( wire244, wire424, wire260);
Device423: OR2 port map ( wire308, wire422, wire424);
Device421: OR2 port map ( wire286, wire268, wire422);
Device420: OR2 port map ( wire256, wire419, wire264);
Device418: OR2 port map ( wire352, wire417, wire419);
Device416: OR2 port map ( wire330, wire312, wire417);
Device415: BUF port map ( wire288, wire268);
Device414: BUF port map ( wire292, wire281);
Device413: BUF port map ( wire284, wire286);
Device412: BUF port map ( wire283, wire288);
Device411: BUF port map ( wire310, wire291);
Device410: BUF port map ( wire370, wire303);
Device409: BUF port map ( wire306, wire308);
Device408: BUF port map ( wire305, wire310);
Device407: BUF port map ( wire332, wire312);
Device406: BUF port map ( wire336, wire325);
Device405: BUF port map ( wire328, wire330);
Device404: BUF port map ( wire327, wire332);
Device403: BUF port map ( wire354, wire335);
Device402: BUF port map ( wire370, wire347);
```

```
Device401: BUF port map ( wire350, wire352);
Device400: BUF port map ( wire349, wire354);
Device399: BUF port map ( wire372, wire357);
Device398: BUF port map ( wire374, wire365);
Device397: BUF port map ( wire368, wire370);
Device396: BUF port map ( wire367, wire372);
Device395: BUF port map ( wire388, wire379);
Device394: BUF port map ( wire382, wire384);
Device393: BUF port map ( wire381, wire386);
Device390: INVERTER port map ( wire391, wire392);
Device389: DFLIPFLOP port map ( clock, d_input, wire391);
Device387: OR2 port map ( wire228, wire392, wire388);
Device378: AND2 port map ( wire380, wire379, wire382);
Device377: AND2 port map ( wire379, power, wire381);
Device376: INVERTER port map ( power, wire380);
Device373: OR2 port map ( wire386, wire358, wire374);
Device364: AND2 port map ( wire366, wire365, wire368);
Device363: AND2 port map ( wire365, wire360, wire367);
Device362: INVERTER port map ( wire360, wire366);
Device359: INVERTER port map ( wire43, wire360);
Device356: DFLIPFLOP port map ( clock, wire357, wire358);
Device346: AND2 port map ( wire348, wire347, wire350);
Device345: AND2 port map ( wire347, wire340, wire349);
Device344: INVERTER port map ( wire340, wire348);
Device341: OR2 port map ( ground, ground, wire342);
Device338: INVERTER port map ( wire339, wire340);
Device337: XOR2 port map ( wire4, wire21, wire339);
Device334: DFLIPFLOP port map ( clock, wire335, wire336);
Device324: AND2 port map ( wire326, wire325, wire328);
Device323: AND2 port map ( wire325, wire318, wire327);
Device322: INVERTER port map ( wire318, wire326);
Device319: OR2 port map ( ground, ground, wire320);
Device316: INVERTER port map ( wire317, wire318);
Device315: XOR2 port map ( wire9, wire26, wire317);
Device313: AND2 port map ( wire312, power, wire314);
Device302: AND2 port map ( wire304, wire303, wire306);
Device301: AND2 port map ( wire303, wire296, wire305);
Device300: INVERTER port map ( wire296, wire304);
Device297: OR2 port map ( ground, ground, wire298);
Device294: INVERTER port map ( wire295, wire296);
Device293: XOR2 port map ( wire9, wire21, wire295);
Device290: DFLIPFLOP port map ( clock, wire291, wire292);
Device280: AND2 port map ( wire282, wire281, wire284);
Device279: AND2 port map ( wire281, wire274, wire283);
Device278: INVERTER port map ( wire274, wire282);
```

Device275: OR2 port map (ground, ground, wire276);
Device272: INVERTER port map (wire273, wire274);
Device271: XOR2 port map (wire14, wire26, wire273);
Device269: AND2 port map (wire268, power, wire270);
Device265: AND2 port map (wire264, power, wire266);
Device261: AND2 port map (wire260, power, wire262);
Device257: OR2 port map (wire251, wire239, wire258);
Device254: DFLIPFLOP port map (clock, wire255, wire256);
Device248: AND2 port map (wire250, wire249, wire252);
Device247: AND2 port map (wire249, wire56, wire251);
Device246: INVERTER port map (wire56, wire250);
Device242: DFLIPFLOP port map (clock, wire243, wire244);
Device236: AND2 port map (wire238, wire237, wire240);
Device235: AND2 port map (wire237, wire58, wire239);
Device234: INVERTER port map (wire58, wire238);
Device231: AND2 port map (wire230, power, wire232);
Device226: DFLIPFLOP port map (clock, wire227, wire228);
Device223: OR2 port map (wire392, wire188, wire224);
Device221: AND2 port map (wire214, ground, wire222);
Device219: AND2 port map (wire214, power, wire220);
Device217: AND2 port map (wire214, ground, wire218);
Device211: AND2 port map (wire206, power, wire212);
Device209: AND2 port map (wire206, power, wire210);
Device207: OR2 port map (ground, ground, wire208);
Device203: AND2 port map (wire202, power, wire204);
Device192: AND2 port map (wire194, wire193, wire196);
Device191: AND2 port map (wire193, wire48, wire195);
Device190: INVERTER port map (wire48, wire194);
Device186: DFLIPFLOP port map (clock, wire187, wire188);
Device182: DFLIPFLOP port map (clock, wire183, wire184);
Device179: OR2 port map (wire184, wire144, wire180);
Device177: AND2 port map (wire170, ground, wire178);
Device175: AND2 port map (wire170, power, wire176);
Device173: AND2 port map (wire170, ground, wire174);
Device167: AND2 port map (wire162, power, wire168);
Device165: AND2 port map (wire162, ground, wire166);
Device163: OR2 port map (ground, ground, wire164);
Device159: AND2 port map (wire158, power, wire160);
Device148: AND2 port map (wire150, wire149, wire152);
Device147: AND2 port map (wire149, wire48, wire151);
Device146: INVERTER port map (wire48, wire150);
Device142: DFLIPFLOP port map (clock, wire143, wire144);
Device138: DFLIPFLOP port map (clock, wire139, wire140);
Device135: OR2 port map (wire140, wire100, wire136);
Device133: AND2 port map (wire126, ground, wire134);

```
  Device131: AND2 port map ( wire126, power, wire132);
  Device129: AND2 port map ( wire126, ground, wire130);
  Device123: AND2 port map ( wire118, ground, wire124);
  Device121: AND2 port map ( wire118, power, wire122);
  Device119: OR2 port map ( ground, ground, wire120);
  Device115: AND2 port map ( wire114, power, wire116);
  Device104: AND2 port map ( wire106, wire105, wire108);
  Device103: AND2 port map ( wire105, wire48, wire107);
  Device102: INVERTER port map ( wire48, wire106);
  Device98: DFLIPFLOP port map ( clock, wire99, wire100);
  Device94: DFLIPFLOP port map ( clock, wire95, wire96);
  Device91: AND2 port map ( wire90, power, wire92);
  Device89: OR2 port map ( wire384, wire82, wire90);
  Device87: AND2 port map ( wire86, power, wire88);
  Device85: OR2 port map ( wire96, wire70, wire86);
  Device83: OR2 port map ( wire77, wire65, wire84);
  Device80: DFLIPFLOP port map ( clock, wire81, wire82);
  Device74: AND2 port map ( wire76, wire75, wire78);
  Device73: AND2 port map ( wire75, wire52, wire77);
  Device72: INVERTER port map ( wire52, wire76);
  Device68: DFLIPFLOP port map ( clock, wire69, wire70);
  Device62: AND2 port map ( wire64, wire63, wire66);
  Device61: AND2 port map ( wire63, wire54, wire65);
  Device60: INVERTER port map ( wire54, wire64);
  Device50: BUF port map ( wire48, global_done);
  Device46: OR2 port map ( wire232, assert_global_done, wire48);
  Device45: BUF port map ( wire43, global_go);
  Device40: BUF port map ( wire38, global_found(1));
  Device36: OR2 port map ( wire270, assert_global_found(1), wire38);
  Device35: BUF port map ( wire33, global_found(0));
  Device31: OR2 port map ( wire314, assert_global_found(0), wire33);
  Device28: BUF port map ( wire26, global_pattern(1));
  Device23: BUF port map ( wire21, global_pattern(0));
  Device16: BUF port map ( wire14, global_image(2));
  Device11: BUF port map ( wire9, global_image(1));
  Device6: BUF port map ( wire4, global_image(0));

end Structure;
```

Figure E.1: The VHDL Description for the Netlist Circuit of Example 6

```
--File name: image_test.vhd
--Author: Ying Shen
--Data: Feb. 26,1999
--The following is a VHDL description of the test bench used in Example 6.


library IEEE;
use IEEE.std_logic_1164.all;

entity SMALL_TEST66 is
end SMALL_TEST66;

architecture TEST66 of SMALL_TEST66 is

component SMALL_OUT66
 port(
    assert_global_done : in std_logic;
    assert_global_go : in std_logic;
    assert_global_found : in std_logic_vector(0 to 1);
    assert_global_pattern : in std_logic_vector(0 to 1);
    assert_global_image : in std_logic_vector(0 to 2);
    d_input : in std_logic;
    clock : in std_logic;
    power : in std_logic;
    ground : in std_logic;
    global_done : out std_logic;
    global_go : out std_logic;
    global_found : out std_logic_vector(0 to 1);
    global_pattern : out std_logic_vector(0 to 1);
    global_image : out std_logic_vector(0 to 2));

end component;

signal assert_global_done: std_logic := '0';
signal assert_global_go: std_logic := '0';
signal assert_global_image: std_logic_vector(0 to 2) := "000";
signal assert_global_found: std_logic_vector(0 to 1) := "00";
signal assert_global_pattern: std_logic_vector(0 to 1) := "00";
signal global_done, d_input, global_go: std_logic;
signal global_found, global_pattern: std_logic_vector(0 to 1);
signal global_image: std_logic_vector(0 to 2);
signal ground: std_logic := '0';
signal clock: std_logic := '0';
signal power: std_logic := '1';
```

```
begin

AA1: SMALL_OUT66 port map ( assert_global_done, assert_global_go,
     assert_global_found, assert_global_pattern, assert_global_image,
     d_input, clock, power, ground, global_done, global_go,
     global_found, global_pattern, global_image);
clock <= not clock after 10 ns;
d_input <= 'U', power after 5 ns;
end TEST66;

configuration conf_SMALL_66 of SMALL_TEST66 is
 for TEST66
 end for;
end conf_SMALL_66;
```

Figure E.2: The VHDL Description for the Test Bench of Example 6

Appendix F
A Brief Introduction to Gofer

This section describes some Gofer features that are used in this thesis. More details about Gofer can be found in (Cunningham, 1995; Jones, 1991, 1993, and 1994).

F.1 What is Gofer?

Gofer is a functional programming environment (in other words, an interpreter) that was implemented by Mark P. Jones for his research activities (Jones, 1991). The language supported by Gofer is very similar to Haskell (Bird, 1998). It has many standard features of modern functional programming languages such as lazy evaluation, ploymorphic functions, higher-order functions, strong typing, pattern matching, and user-defined algebraic types. It also has a language feature called a class, that is used for inheritance and overloading.

F.2 Functions

The functions in Gofer are divided into standard functions and user-defined functions. All standard functions, for example, the division function named **div** or "/", are included as part of a large collection of functions called the 'standard prelude'. They are automatically loaded into the Gofer system while we start the Gofer interpreter. We can also define our own functions in the form of a text file that can be loaded and used by the Gofer system.

F.3 Data Types

Gofer supports simple types, list and tuple types, function types, and user defined types. It is noted that all function names must begin with a lower-case letter and all type names must begin with an upper-case letter.

The simple types include the following four types: **Bool**, **Char**, **Int**, and **Float**. They support boolean literals, character literals, integer literals, and floating point literals, respectively.

A function specified as having type *t1* -> *t2*, where *t1* and *t2* are types, takes an argument of type *t1* and returns a result of type *t2*.

[*t*] is the type of a list whose elements are lists of values of type *t*. The length of a list is variable. Therefore, **[Char]** represents the type of lists of characters. We can define the data type *String* as follows: **type** *String* = **[Char]** and use it to express string literals. It is noted that the above definition is a declaration for a type synonym. That is, the type alias *String* can be used in place of the specified type expression.

The tuple in Gofer is very similar to the structure in C. If *t1*, *t2*, ..., *tn* are types and $n \geq 2$, then (*t1*, *t2*, ..., *tn*) represents a type of *n*-tuples. For example, a type of 3-tuples, (*Netlist*, *WireTab*, *ReqTab*), is defined in this thesis.

The mostly used types in this thesis are user defined types declared by the keyword **data**. The definition of the types is as follows:

$$\textbf{data}\ \mathit{Datatype}\ \ a_1\ a_2 \cdots a_n = \mathit{constr}_1 \mid \mathit{constr}_2 \mid \cdots \mid \mathit{constr}_m$$

In the above definition,

- *Datatype* is the name of a new type constructor of arity *n* ($n \geq 0$).

- a_1 a_2 $\cdots$ a_n are distinct type variables representing the n arguments of the data type.
- $constr_1$, $constr_2$, $\cdots$, $constr_m$ ($m \geq 1$) describe the way in which the elements of the new data type are constructed.

For example, **data** *Colour* = *Red* | *Green* | *Blue* defines a new data type *Colour* that is an enumerated type with elements *Red*, *Green*, and *Blue*. The following recursive definition represents a binary tree data type Tree:

data *Tree* = *Empty* | *Node* **Int** *String Tree Tree*

Given the above definition, the constructor function Node takes four arguments that have types: **Int**, *String*, *Tree*, *Tree*, respectively and returns a *Tree*.

F.4 The Use of Monads

The monad concept is from category theory (Wadler, 1995). This thesis uses monad to model programs that make use of an internal state. For example, a new data type is defined as follows:

data *StateExTrans s a* = SET (*s* -> *Ok_Err s a*)

where *s* and *a* are type variables, and *Ok_Err* is defined as

data *Ok_Err s a* = *Ok s a String* | *Err String*

The functor and monad structures for the above state transformers are described by Norvell's program named *stateExMonad* (Norvell, 1996). The following function is

one of functions that have used the type *StateExTrans*.

```
getNetlist :: StateExTrans NLGState Netlist
getNetlist = SET( \( n, w, r) -> Ok (n, w, r) n "" )
```

F.5 The 'do' Notation

The **do** notation is used to provide a more attractive syntax for monadic programming. This thesis has used a lot of **do** notations in the following forms:

- **do** *expression*

  which can be translated to *expression.*

- **do** *expression*
  morelines

  which can be translated to *expression* `**bind**` (\ _ -> **do** morelines)

- **do** *pattern* <- *expression*
  morelines

  which can be translated to *expression* `**bind**` (\ *pattern* -> **do** morelines)

- **do** **let** declarationList
  morelines

  which can be translated to

```
let  declarationList
in   do morelines
```

In the above expressions, the `**bind**` operator is defined as

```
(a `bind` k) s = let (r, s') = a s
                 in k r s'
```